# Contents

1 Introduction 3  
1.1 Productive Performance Not Performance Portability 3  
1.2 Phases in the Optimization Workflow 3  
1.3 Profiling and Tuning Your Code 4  
1.4 Source Code Examples 4  

2 Getting Started 5  
2.1 Remember Amdahl’s Law 5  
2.2 Locality Matters 5  
2.3 Rightsize Your Work 6  

3 Parallelization 7  
3.1 Use a Parallel Programming Language or API 7  
3.2 Parallelizing Compilers 7  
3.3 Parallel Libraries 7  

4 Intel® Iris® Xe GPU Architecture 9  
4.1 Xe-LP Execution Units (EUs) 9  
4.2 Xe-LP Dual Subslices 10  
4.3 Xe-LP Slice 12  
4.4 Intel UHD Architecture Parameters across Generations 13  
4.5 Xe-Core 13  
4.6 Xe-Slice 15  
4.7 Xe-Stack 15  
4.8 Xe-HPC 2-Stack Ponte Vecchio GPU 16  
4.9 Xe-HPG GPU 17  
4.10 Xe-Intel® Data Center GPU Flex Series 17  
4.11 Terminology and Configuration Summary 18  

5 GPU Execution Model Overview 21  

6 SYCL® Thread Mapping and GPU Occupancy 23  
6.1 nd_range 23  
6.2 Thread Synchronization 24  
6.3 Mapping Work-groups to Xe-cores for Maximum Occupancy 24  
6.4 Intel® GPU Occupancy Calculator 35  

7 Kernels 37  
7.1 Sub-groups and SIMD Vectorization 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Removing Conditional Checks</td>
<td>45</td>
</tr>
<tr>
<td>7.3 Registerization and Avoid Register Spills</td>
<td>48</td>
</tr>
<tr>
<td>7.4 Shared Local Memory</td>
<td>57</td>
</tr>
<tr>
<td>7.5 Pointer Aliasing and the Restrict Directive</td>
<td>65</td>
</tr>
<tr>
<td>7.6 Synchronization among Threads in a Kernel</td>
<td>69</td>
</tr>
<tr>
<td>7.7 Considerations for Selecting Work-group Size</td>
<td>80</td>
</tr>
<tr>
<td>7.8 Reduction</td>
<td>85</td>
</tr>
<tr>
<td>7.9 Kernel Launch</td>
<td>92</td>
</tr>
<tr>
<td>7.10 Executing Multiple Kernels on the Device at the Same Time</td>
<td>95</td>
</tr>
<tr>
<td>7.11 Submitting Kernels to Multiple Queues</td>
<td>98</td>
</tr>
<tr>
<td>7.12 Avoid Redundant Queue Construction</td>
<td>102</td>
</tr>
<tr>
<td>8 Using Libraries for GPU Offload</td>
<td>107</td>
</tr>
<tr>
<td>8.1 Using Performance Libraries</td>
<td>107</td>
</tr>
<tr>
<td>8.2 Using Standard Library Functions in DPC++ Kernels</td>
<td>109</td>
</tr>
<tr>
<td>8.3 Efficiently Implementing Fourier Correlation Using oneAPI Math Kernel Library (oneMKL)</td>
<td>113</td>
</tr>
<tr>
<td>9 Host/Device Memory, Buffer and USM</td>
<td>123</td>
</tr>
<tr>
<td>9.1 Performance Impact of USM and Buffers</td>
<td>123</td>
</tr>
<tr>
<td>9.2 Optimizing Memory Movement Between Host and Accelerator</td>
<td>127</td>
</tr>
<tr>
<td>9.3 Avoid moving data back and forth between host and device</td>
<td>131</td>
</tr>
<tr>
<td>9.4 Avoid Declaring Buffers in a Loop</td>
<td>135</td>
</tr>
<tr>
<td>9.5 Buffer Accessor Modes</td>
<td>138</td>
</tr>
<tr>
<td>10 Host/Device Coordination</td>
<td>147</td>
</tr>
<tr>
<td>10.1 Asynchronous and Overlapping Data Transfers Between Host and Device</td>
<td>147</td>
</tr>
<tr>
<td>11 Using Multiple Heterogeneous Devices</td>
<td>153</td>
</tr>
<tr>
<td>11.1 Overlapping Compute on Various Accelerators in the Platform</td>
<td>153</td>
</tr>
<tr>
<td>12 Compilation</td>
<td>157</td>
</tr>
<tr>
<td>12.1 Just-In-Time Compilation in DPC++</td>
<td>157</td>
</tr>
<tr>
<td>12.2 Specialization Constants</td>
<td>160</td>
</tr>
<tr>
<td>13 Optimizing Media Pipelines</td>
<td>165</td>
</tr>
<tr>
<td>13.1 Media Engine Hardware</td>
<td>165</td>
</tr>
<tr>
<td>13.2 Media API Options for Hardware Acceleration</td>
<td>167</td>
</tr>
<tr>
<td>13.3 Media Pipeline Parallelism</td>
<td>168</td>
</tr>
<tr>
<td>13.4 Media Pipeline Inter-operation and Memory Sharing</td>
<td>171</td>
</tr>
<tr>
<td>13.5 DPCPP-Blur Example</td>
<td>177</td>
</tr>
<tr>
<td>14 OpenMP Offloading Tuning Guide</td>
<td>179</td>
</tr>
<tr>
<td>14.1 OpenMP Directives</td>
<td>179</td>
</tr>
<tr>
<td>14.2 OpenMP Execution Model</td>
<td>180</td>
</tr>
<tr>
<td>14.3 Terminology</td>
<td>181</td>
</tr>
<tr>
<td>14.4 Compiling and Running an OpenMP Application</td>
<td>181</td>
</tr>
<tr>
<td>14.5 Offloading oneMKL Computations onto the GPU</td>
<td>185</td>
</tr>
<tr>
<td>14.6 Tools to Analyze Performance of OpenMP Applications</td>
<td>216</td>
</tr>
<tr>
<td>14.7 OpenMP Offload Best Practices</td>
<td>218</td>
</tr>
</tbody>
</table>
## 15 Debugging and Profiling

15.1 GPU Analysis with VTune™ Profiler ................................................. 285
15.2 Intel® Advisor GPU Analysis ................................................................. 295
15.3 Doing IO in the Kernel ................................................................. 295
15.4 Using the Timers ................................................................. 319
15.5 How to Use the Intercept Layer for OpenCL™ Applications .......... 323
15.6 Level Zero Tracer ................................................................. 326

## 16 GPU Analysis with Intel® Graphics Performance Analyzers (Intel® GPA)

16.1 Introduction ........................................................................... 329
16.2 Execution Unit Stall, Active and Throughput ......................... 330
16.3 Graphics Frame Analyzer .................................................. 336

## 17 Reference

347

## 18 Terms and Conditions

349
Welcome to the oneAPI GPU Optimization Guide. This document gives tips for getting the best GPU performance for oneAPI programs.
1.0 Introduction

Designing high-performance software requires you to think differently than you might normally do when writing software. You need to be aware of the hardware on which your code is intended to run, and the characteristics that control the performance of that hardware. Your goal is to structure the code such that it produces correct answers, but does so in a way that maximizes the hardware’s ability to execute the code.

oneAPI is a cross-industry, open, standards-based, unified programming model that delivers a common developer experience across accelerator architectures. A unique feature of accelerators is that they are additive to the main CPU on the platform. The primary benefit of using an accelerator is to improve the behavior of your software by partitioning it across the host and accelerator to specialize portions of the computation that run best on the accelerator. Accelerator architectures can offer a benefit through specialization of compute hardware for certain classes of computations. This enables them to deliver best results for software specialized to the accelerator architecture.

The primary focus of this document is GPUs. Each section focuses on different topics to guide you in your path to creating optimized solutions. The Intel® oneAPI toolkits provide the languages and development tools you will use to optimize your code. This includes compilers, debuggers, profilers, analyzers, and libraries.

1.1 Productive Performance Not Performance Portability

While this document focuses on GPUs, you may also need your application to run on CPUs and other types of accelerators. Since accelerator architectures are specialized, you need to specialize your code to achieve best performance. Specialization includes restructuring and tuning the code to create the best mapping of the application to the hardware. In extreme cases, this may require redesigning your algorithms for each accelerator to best expose the right type of computation. The value of oneAPI is that it allows each of these variations to be expressed in a common language with device-specific variants launched on the appropriate accelerator.

1.2 Phases in the Optimization Workflow

The first phase in using a GPU is to identify which parts of the application can benefit. This is usually compute-intensive code that has the right ratio of memory accesses to computation, and has the right data dependence patterns to map onto the GPU. GPUs include local memory and typically provide massive parallelism. This determines which characteristics of the code are most important when deciding what to offload.

The Intel Advisor tool included in the Intel oneAPI Base Toolkit is designed to analyze your code and help you identify the best opportunities for parallel execution. The profilers in Intel Advisor measure the data movement in your functions, the memory access patterns, and the amount of computation in order to project how code will perform when mapped onto different accelerators. The regions with highest potential benefit should be your first targets for acceleration.

GPUs often exploit parallelism at multiple levels. This includes overlap between host and GPU, parallelism across the compute cores, overlap between compute and memory accesses, concurrent pipelines, and vector computation. Using all of these levels of parallelism requires a good understanding of the GPU architecture and capabilities in the libraries and languages at your disposal.
**Keep all the compute resources busy.** There must be enough independent tasks to saturate the device and fully utilize all execution resources. For example, if the device has 100 compute cores but you only have one task, 99% of the device will be idle. Often you create many more independent tasks than available compute resources so that the hardware can schedule more work as prior tasks complete.

**Minimize the synchronization between the host and the device.** The host launches a kernel on the device and waits for its completion. Launching a kernel incurs overhead, so structure the computation to minimize the number of times a kernel is launched.

**Minimize the data transfer between host and device.** Data typically starts on the host and is copied to the device as input to the computation. When a computation is finished, the results must be transferred back to the host. For best performance, minimize data transfer by keeping intermediate results on the device between computations. Reduce the impact of data transfer by overlapping computation and data movement so the compute cores never have to wait for data.

**Keep the data in faster memory and use an appropriate access pattern.** GPU architectures have different types of memory and these have different access costs. Registers, caches, and scratchpads are cheaper to access than local memory, but have smaller capacity. When data is loaded into a register, cache line, or memory page, use an access pattern that will use all the data before moving to the next chunk. When memory is banked, use a stride that avoids all the compute cores trying to access the same memory bank simultaneously.

### 1.3 Profiling and Tuning Your Code

After you have designed your code for high performance, the next step is to measure how it runs on the target accelerator. Add timers to the code, collect traces, and use tools like VTune Profiler to observe the program as it runs. The information collected can identify where hardware is bottlenecked and idle, illustrate how behavior compares with peak hardware roofline, and identify the most important hotspots to focus optimization efforts.

### 1.4 Source Code Examples

Throughout the book, we use real code examples to illustrate optimization techniques. All the examples in this guide can be found at [https://github.com/oneapi-src/oneAPI-samples/tree/master/Publications/GPU-Opt-Guide](https://github.com/oneapi-src/oneAPI-samples/tree/master/Publications/GPU-Opt-Guide). Now it is the perfect time to download the examples and set up your environment by following the instructions in the README.md.

We try hard to keep the examples as simple, short and easy to follow as possible so optimization techniques in each example are not shadowed by the example itself and can be quickly grasped. Code snippets from the examples are referenced throughout the text. By replacing /examples with /GPU-Opt-Guide in the pathname listed right before a code snippet, one can easily locate the full example source that contains the snippet. As an example, the full source code for /examples/reduction/reduction.cpp is in GPU-Opt-Guide/reduction/reduction.cpp.

There is an old saying “I hear and I forget. I see and I remember. I do and I understand.”. It is strongly suggested that you pause to try the example on a real machine when a code snippet is encountered while reading the text.

Welcome to Intel® oneAPI GPU Optimization Guide!
2.0 Getting Started

If you only have time to read this far, then you at least need to know the three big concepts to optimize software for an accelerator.

2.1 Remember Amdahl’s Law

This may appear obvious, but it is the first step in making use of an accelerator. Amdahl’s law states that the fraction of time an application uses an accelerator $F_p$ limits the benefit of acceleration. The maximum speedup is bounded by $1/(1 - F_p)$. If you use the accelerator 50% of the time, you will get at most a $2 \times$ speedup, even with an infinitely powerful accelerator.

Note here that this is in terms of your program execution, not your program’s source code. The parallel kernels may represent a very small fraction of your overall source code, but if this is where you execution time is concentrated you can still do well.

2.2 Locality Matters

An accelerator often has specialized memory with a disjoint address space. An application must allocate or move data into the right memory at the right time.

Accelerator memory is arranged in a hierarchy. Registers are more efficient to access than caches, and caches are more efficient to access than main memory. Bringing data closer to the point of execution improves efficiency.

There are many ways you can refactor your code to get your data closer to the execution. They will be outlined in the following sections. Here, we focus on three:

1. Allocate your data on the accelerator, and when copied there, keep it resident for as long as possible. Your application may have many offloaded regions. If you have data that is common between these regions, it makes sense to amortize the cost of the first copy, and just reuse it in place for the remaining kernel invocations.

2. Access contiguous blocks of memory as your kernel executes. The hardware will fetch contiguous blocks into the memory hierarchy, so you have already paid the cost for the entire block. After you use the first element of the block, the remaining elements are almost free to access so take advantage of it.

3. Restructure your code into blocks with higher data reuse. In a two-dimensional matrix, you can arrange your work to process one block of elements before moving onto the next block of elements. For example, in a stencil operation you may access the prior row, the current row, and the next row. As you walk over the elements in a block you reuse the data and avoid the cost of requesting it again.
2.3 Rightsize Your Work

Data-parallel accelerators are designed as throughput engines and are often specialized by replicating execution units many times. This is an easy way of getting higher performance on data-parallel algorithms since more of the elements can be processed at the same time.

However, fully utilizing a parallel processor can be challenging. For example, imagine you have 512 execution units, where each execution unit had eight threads, and each thread has 16-element vectors. You need to have a minimum of $512 \times 8 \times 16 = 65536$ parallel activities scheduled at all times just to match this capacity. In addition, if each parallel activity is small, you need another large factor to amortize the cost of submitting this work to the accelerator. Fully utilizing a single large accelerator may require decomposing a computation into millions of parallel activities.
3.0 Parallelization

Parallelism is essential to effective use of accelerators because they contain many independent processing elements that are capable of executing code in parallel. There are three ways to develop parallel code.

3.1 Use a Parallel Programming Language or API

There are many parallel programming languages and APIs that can be used to express parallelism. oneAPI supports parallel program development through the Data Parallel C++ (DPC++) language. oneAPI also has a number of code generation tools to convert these programs into binaries that can be executed on different accelerators. The usual workflow is that a user starts with a serial program, identifies the parts of the code that take a long time to execute (referred to as hotspots), and converts them into parallel kernels that can be offloaded to an accelerator for execution.

3.2 Parallelizing Compilers

Directive-based approaches like OpenMP are another way to develop parallel programs. In a directive-based approach, the programmer provides hints to the compiler about parallelism without modifying the code explicitly. This approach is easier than developing a parallel program from first principles.

3.3 Parallel Libraries

oneAPI includes a number of libraries like oneTBB, oneMKL, oneDNN, and oneVPL that provide highly-optimized versions of common computational operations run across a variety of accelerator architectures. Depending on the needs of the application, a user can directly call the functions from these libraries and get efficient implementations of these for the underlying architecture. This is the easiest approach to developing parallel programs, provided the library contains the required functions. For example, machine learning applications can take advantage of the optimized primitives in oneDNN. These libraries have been thoroughly tested for both correctness and performance, which makes programs more reliable when using them.
4.0 Intel® Iris® Xe GPU Architecture

The Intel® Iris® Xe GPU family consists of a series of microarchitectures, ranging from integrated/low power (Xe-LP), to enthusiast/high performance gaming (Xe-HPG), data center/AI (Xe-HP) and high performance computing (Xe-HPC).

![Fig. 1: Intel® Iris® Xe family](image)

This chapter introduces Xe GPU family microarchitectures and configuration parameters.

4.1 Xe-LP Execution Units (EUs)

An Execution Unit (EU) is the smallest thread-level building block of the Intel® Iris® Xe-LP GPU architecture. Each EU is simultaneously multithreaded (SMT) with seven threads. The primary computation unit consists of a 8-wide Single Instruction Multiple Data (SIMD) Arithmetic Logic Units (ALU) supporting SIMD8 FP/INT operations and a 2-wide SIMD ALU supporting SIMD2 extended math operations. Each hardware thread has 128 general-purpose registers (GRF) of 32B wide.
Xe-LP EU supports diverse data types FP16, INT16 and INT8 for AI applications. The Intel® GPU Compute Throughput Rates (Ops/clock/EU) table compares the EU throughput rates of Xe-LP vs that of Intel® Gen 11 GPUs.

Table 1: Intel® GPU Compute Throughput Rates (Ops/clock/EU)

<table>
<thead>
<tr>
<th></th>
<th>Intel® Iris® Xe-LP</th>
<th>Gen 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>FP16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>INT32</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>INT16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>INT8</td>
<td>32 (DP4A)</td>
<td>NA</td>
</tr>
</tbody>
</table>

4.2 Xe-LP Dual Subslices

Each Xe-LP Dual Subslice (DSS) consists of an EU array of 16 EUs, an instruction cache, a local thread dispatcher, Shared Local Memory (SLM), and a data port of 128B/cycle. It is called dual subslice because the hardware can pair two EUs for SIMD16 executions.

The SLM is a 128KB High Bandwidth Memory (HBM) accessible from the EUs in the subslice. One important usage of SLM is to share atomic data and signals among the concurrent work-items executing in a subslice. For this reason, if a kernel’s work-group contains synchronization operations, all work-items of the work-group must be allocated to a single subslice so that they have shared access to the same 128KB SLM. The work-group size...
must be chosen carefully to maximize the occupancy and utilization of the subslice. In contrast, if a kernel does not access SLM, its work-items can be dispatched across multiple subslices.

The following table summarizes the computing capacity of a subslice.
### Table 2: Subslice computing capacity

<table>
<thead>
<tr>
<th>GPU Generation</th>
<th>EUs</th>
<th>Threads</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Iris Xe ICX</td>
<td>8</td>
<td>(7 \times 8 = 56)</td>
<td>(56 \times 8 = 448)</td>
</tr>
<tr>
<td>Intel Iris Xe-LP TGL</td>
<td>16</td>
<td>(7 \times 16 = 112)</td>
<td>(112 \times 8 = 896)</td>
</tr>
</tbody>
</table>

#### 4.3 Xe-LP Slice

Each Xe-LP slice consists of six (dual) subslices for a total of 96 EUs, up to 16MB L3 cache, 128B/cycle bandwidth to L3 and 128B/cycle bandwidth to memory.

![Fig. 3: Xe-LP slice](image-url)
4.4 Intel UHD Architecture Parameters across Generations

The following table summarizes the key architecture parameters in the current released products with Intel UHD Graphics:

<table>
<thead>
<tr>
<th>Generations</th>
<th>Threads per VE/EU</th>
<th>VEs/EUs per core/SubSlice</th>
<th>Xe-core/SubSlice</th>
<th>Total Threads</th>
<th>Total Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen9 (BDW)</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>168</td>
<td>1344</td>
</tr>
<tr>
<td>Intel Iris Xe ICL (Gen11)</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>448</td>
<td>3584</td>
</tr>
<tr>
<td>Intel Iris Xe-LP TGL (Gen12)</td>
<td>7</td>
<td>16</td>
<td>6</td>
<td>672</td>
<td>5376</td>
</tr>
</tbody>
</table>

4.5 Xe-Core

Unlike the Xe-LP and prior generations of Intel GPUs that used the Execution Unit (EU) as a compute unit, Xe-HPG and Xe-HPC use the Xe-core. This is similar to an Xe-LP dual subslice.

An Xe-core contains vector and matrix ALUs, which are referred to as vector and matrix engines.

An Xe-core of the Xe-HPC GPU contains 8 vector and 8 matrix engines, alongside a large 512KB L1 cache/SLM. It powers the Ponte Vecchio GPU. Each vector engine is 512 bit wide supporting 16 FP32 SIMD operations with fused FMAs. With 8 vector engines, the Xe-core delivers 512 FP16, 256 FP32 and 256 FP64 operations/cycle. Each matrix engine is 4096 bit wide. With 8 matrix engines, the Xe-core delivers 8192 int8, 4096 FP16/BF16 and 2048 FP32 operations/cycle. The Xe-core provides 1024B/cycle load/store bandwidth to the memory system.
Fig. 4: Xe-core
4.6 Xe-Slice

An Xe-slice contains 16 Xe-core for a total of 8MB L1 cache, 16 ray tracing units and 1 hardware context.

**Fig. 5: Xe-slice**

4.7 Xe-Stack

An Xe-stack contains up to 4 Xe-slice: 64 Xe-cores, 64 ray tracing units, 4 hardware contexts, 4 HBM2e controllers, 1 media engine, and 8 Xe-Link high speed coherent fabric. It also contains a shared L2 cache.

**Fig. 6: Xe-stack**
4.8 Xe-HPC 2-Stack Ponte Vecchio GPU

An Xe-HPC 2-stack Ponte Vecchio GPU consists of 2 stacks: 8 slices, 128 Xe-cores, 128 ray tracing units, 8 hardware contexts, 8 HBM2e controllers, and 16 Xe-Links.

Fig. 7: Xe-HPC 2-Stack
4.9 Xe-HPG GPU

Xe-HPG is the enthusiast or high performance gaming variant of the Xe architecture. The microarchitecture is focused on graphics performance and supports hardware-accelerated ray tracing.

An Xe-core of the Xe-HPG GPU contains 16 vector and 16 matrix engines. It powers the Intel® Arc GPUs. Each vector engine is 256 bit wide, supporting 8 FP32 SIMD operations with fused FMA. With 16 vector engines, the Xe-core delivers 256 FP32 operations/cycle. Each matrix engine is 1024 bit wide. With 16 matrix engines, the Xe-core delivers 4096 int8, 2048 FP16/BF16 and 1024 FP32 operations/cycle. The Xe-core provides 512B/cycle load/store bandwidth to the memory system.

An Xe-HPG GPU consists of 8 Xe-HPG-slice, which contains up to 4 Xe-HPG-cores for a total of 4096 FP32 ALU units/shader cores.

4.10 Xe- Intel® Data Center GPU Flex Series

Intel® Data Center GPU Flex Series (formerly codenamed ATS-M) come in two configurations. The 150W option has 32 Xe-cores on a PCIe Gen4 card. The 75W option has two GPUs for 16 Xe-cores (8 Xe-cores per GPU). Both configurations come with 4 Xe media engines, the industry’s first AV1 hardware encoder and accelerator for data center, GDDR6 memory, ray tracing units, and built-in XMX AI acceleration.

Intel® Data Center GPU Flex Series are derivatives of the Xe-HPG GPUs. An Intel® Data Center GPU Flex 170 consists of 4 Xe-HPG-slices for a total of 16 Xe-cores with 2048 FP32 ALU units/shader cores.
Targeting data center cloud gaming, media streaming and video analytics applications, Intel® Data Center GPU Flex Series provide hardware accelerated AV1 encoder, delivering a 30% bit-rate improvement without compromising on quality. It supports 8 simultaneous 4K streams or more than 30 1080p streams per card. AI models can be applied to the decoded streams utilizing Intel® Data Center GPU Flex Series’ Xe-cores.

Media streaming and delivery software stacks lean on Intel® oneVPL to decode and encode acceleration for all the major codecs including AV1. Media distributors can choose from the two leading media frameworks FFmpeg or GStreamer, both enabled for acceleration with oneVPL on Intel CPUs and GPUs.

In parallel to oneVPL accelerating decoding and encoding of media streams, oneDNN (oneAPI Deep Neural Network library) delivers AI optimized kernels enabled to accelerate inference modes in TensorFlow or PyTorch frameworks, or with the OpenVINO model optimizer and inference engine to further accelerate inference and speed customer deployment of their workloads.

4.11 Terminology and Configuration Summary

The following Architecture Terminology Changes table maps legacy GPU terminologies (used in Generation 9 through Generation 12 Intel® Core™ architectures) to their new names in the Intel® Iris® Xe GPU (Generation 12.7 and newer) architecture paradigm.

<table>
<thead>
<tr>
<th>Old Term</th>
<th>New Intel Term</th>
<th>Generic Term</th>
<th>New Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Unit (EU)</td>
<td>Xe Vector Engine</td>
<td>Vector Engine</td>
<td>XVE</td>
</tr>
<tr>
<td>Systolic/&quot;DPAS part of EU&quot;</td>
<td>Xe Matrix eXtension</td>
<td>Matrix Engine</td>
<td>XMX</td>
</tr>
<tr>
<td>Subslice (SS) or Dual Subslice (DSS)</td>
<td>Xe-core</td>
<td>NA</td>
<td>XC</td>
</tr>
<tr>
<td>Slice</td>
<td>Render Slice / Compute Slice</td>
<td>Slice</td>
<td>SLC</td>
</tr>
<tr>
<td>Tile</td>
<td>Stack</td>
<td>Stack</td>
<td>STK</td>
</tr>
</tbody>
</table>

The following Xe Configurations table lists the hardware characteristics across the Xe family GPUs.

<p>| Table 5: Xe Configurations |</p>
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Xe-LP (TGL)</th>
<th>Xe-HPG (Arc A770)</th>
<th>Xe-HPG (Data Center GPU Flex 170)</th>
<th>Xe-HPC (PVC 1 Stack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice count</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>XC (DSS/SS) count</td>
<td>6</td>
<td>32</td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td>XVE (EU) / XC</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>XVE count</td>
<td>96</td>
<td>512</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>Threads / XVE</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Thread count</td>
<td>672</td>
<td>4096</td>
<td>2048</td>
<td>4096</td>
</tr>
<tr>
<td>FLOPs / clk - single precision, MAD</td>
<td>1536</td>
<td>8192</td>
<td>4096</td>
<td>16384</td>
</tr>
<tr>
<td>FLOPs / clk - double precision, MAD</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>16384</td>
</tr>
<tr>
<td>FLOPs / clk - FP16 DP4AS</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>262144</td>
</tr>
<tr>
<td>LL cache size</td>
<td>3.84MB</td>
<td>16MB</td>
<td>8MB</td>
<td>up to 204MB</td>
</tr>
<tr>
<td>SLM size</td>
<td>6 × 128KB</td>
<td>32 × 128KB</td>
<td>16 × 128KB</td>
<td>64 × 128KB</td>
</tr>
<tr>
<td>FMAD, SP (ops / XVE / clk)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>SQRT, SP (ops / XVE / clk)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
5.0 GPU Execution Model Overview

The General Purpose GPU (GPGPU) compute model consists of a host connected to one or more compute devices. Each compute device consists of many GPU Compute Engines (CE), also known as Execution Units (EU) or Xe Vector Engines (XVE). The compute devices may also include caches, shared local memory (SLM), high-bandwidth memory (HBM), and so on, as shown in the figure General Purpose Compute Model. Applications are then built as a combination of host software (per the host framework) and kernels submitted by the host to run on the VEs with a predefined decoupling point.

![General Purpose Compute Model](image)

**Fig. 9:** General Purpose Compute Model

The GPGPU compute architecture contains two distinct units of execution: a host program and a set of kernels that execute within the context set by the host. The host interacts with these kernels through a command queue. Each device may have its own command queue. When a command is submitted into the command queue, the command is checked for dependencies and then executed on a VE inside the compute unit clusters. Once the command has finished executing, the kernel communicates an end of life cycle through "end of thread" message.

The GP execution model determines how to schedule and execute the kernels. When a kernel-enqueue command submits a kernel for execution, the command defines an index space or N-dimensional range. A kernel-instance consists of the kernel, the argument values associated with the kernel, and the parameters that define the index space. When a compute device executes a kernel-instance, the kernel function executes for each point in the defined index space or N-dimensional range.
An executing kernel function is called a work-item, and a collection of these work-items is called a work-group. A compute device manages work-items using work-groups. Individual work-items are identified by either a global ID, or a combination of the work-group ID and a local ID inside the work-group.

The work-group concept, which essentially runs the same kernel on several unit items in a group, captures the essence of data parallel computing. The VEs can organize work-items in SIMD vector format and run the same kernel on the SIMD vector, hence speeding up the compute for all such applications.

A device can compute each work-group in any arbitrary order. Also, the work-items within a single work-group execute concurrently, with no guarantee on the order of progress. A high level work-group function, like Barriers, applies to each work-item in a work-group, to facilitate the required synchronization points. Such a work-group function must be defined so that all work-items in the work-group encounter precisely the same work-group function.

Synchronization can also occur at the command level, where the synchronization can happen between commands in host command-queues. In this mode, one command can depend on execution points in another command or multiple commands.

Other types of synchronization based on memory-order constraints inside a program include Atomics and Fences. These synchronization types control how a memory operation of any particular work-item is made visible to another, which offers micro-level synchronization points in the data-parallel compute model.

Note that an Intel GPU device is equipped with many Vector Engines (VEs), and each VE is a multi-threaded SIMD processor. Compiler generates SIMD code to map several work-items to be executed simultaneously within a given hardware thread. The SIMD-width for a kernel is a heuristic driven compiler choice. Common SIMD-width examples are SIMD-8, SIMD-16, and SIMD-32.

For a given SIMD-width, if all kernel instances within a thread are executing the same instruction, the SIMD lanes can be maximally utilized. If one or more of the kernel instances choose a divergent branch, then the thread executes the two paths of the branch and merges the results by mask. The VE’s branch unit keeps track of such branch divergence and branch nesting.
6.0 SYCL* Thread Mapping and GPU Occupancy

The SYCL* execution model exposes an abstract view of GPU execution. The SYCL thread hierarchy consists of a 1-, 2-, or 3-dimensional grid of work-items. These work-items are grouped into equal sized thread groups called work-groups. Threads in a work-group are further divided into equal sized vector groups called sub-groups (see the illustration that follows).

**Work-item** A work-item represents one of a collection of parallel executions of a kernel.

**Sub-group** A sub-group represents a short range of consecutive work-items that are processed together as a SIMD vector of length 8, 16, 32, or a multiple of the native vector length of a CPU with Intel® UHD Graphics.

**Work-group** A work-group is a 1-, 2-, or 3-dimensional set of threads within the thread hierarchy. In SYCL, synchronization across work-items is only possible with barriers for the work-items within the same work-group.

6.1 nd_range

An nd_range divides the thread hierarchy into 1-, 2-, or 3-dimensional grids of work-groups. It is represented by the global range, the local range of each work-group.

![Thread hierarchy](image)

**Fig. 10:** Thread hierarchy

The diagram above illustrates the relationship among ND-Range, work-group, sub-group, and work-item.
6.2 Thread Synchronization

SYCL provides two synchronization mechanisms that can be called within a kernel function. Both are only defined for work-items within the same work-group. SYCL does not provide any global synchronization mechanism inside a kernel for all work-items across the entire nd_range.

- **“mem_fence”** inserts a memory fence on global and local memory access across all work-items in a work-group.
- **“barrier”** inserts a memory fence and blocks the execution of all work-items within the work-group until all work-items have reached its location.

6.3 Mapping Work-groups to Xe-cores for Maximum Occupancy

The rest of this chapter explains how to pick a proper work-group size to maximize the occupancy of the GPU resources. The example system is the Tiger Lake processors with Xe-LP GPU as the execution target. The examples also use the new terminologies Xe-core (XC) for Dual Subslice, and Xe Vector Engine (XVE) for Execution Unit.

From the **Key architecture parameters, Intel UHD Graphics** table, we summarize the architecture parameters for Xe-LP Graphics (TGL) GPU below:

<table>
<thead>
<tr>
<th></th>
<th>VEs</th>
<th>Threads</th>
<th>Operations</th>
<th>Maximum Work Group Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each Xe-core</td>
<td>16</td>
<td>$7 \times 16 = 112$</td>
<td>$112 \times 8 = 896$</td>
<td>512</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>$16 \times 6 = 96$</td>
<td>$112 \times 6 = 672$</td>
<td>$896 \times 6 = 5376$</td>
<td>512</td>
</tr>
</tbody>
</table>

The maximum work-group size is a constraint imposed by the hardware and GPU driver. You can query the maximum work-group using `device::get_info<cl::sycl::info::device::max_work_group_size>()` on the supported size.

Let’s start with a simple kernel:

```cpp
auto command_group = 
    [&](auto &cgh) {
        cgh.parallel_for(sycl::range<3>(64, 64, 64), // global range
            [=](item<3> it) { 
                // (kernel code)
            } );
    }
```

This kernel contains 262,144 work-items structured as a 3D range of 64 x 64 x 64. It leaves the work-group and sub-group size selection to the compiler. To fully utilize the 5376 parallel operations available in the GPU slice, the compiler must choose a proper work group size.

The two most important GPU resources are:
• Thread Contexts: The kernel should have a sufficient number of threads to utilize the GPU's thread contexts.

• SIMD Units and SIMD Registers: The kernel should be organized to vectorize the work-items and utilize the SIMD registers.

In a SYCL kernel, the programmer can affect the work distribution by structuring the kernel with proper work-group size, sub-group size, and organizing the work-items for efficient vector execution. Writing efficient vector kernels is covered in a separate section. This chapter focuses on work-group and sub-group size selection.

Thread contexts are easier to utilize than SIMD vector. Therefore, start with selecting the number of threads in a work-group. Each Xe-core has 112 thread contexts, but usually you cannot use all the threads if the kernel is also vectorized by 8 (112 x 8 = 896 > 512). From this, we can derive that the maximum number of threads in a work-group is 64 (512 / 8).

SYCL does not provide a mechanism to directly set the number of threads in a work-group. However, you can use work-group size and SIMD sub-group size to set the number of threads:

Work group size = Threads x SIMD sub-group size

You can increase the sub-group size as long as there are a sufficient number of registers for the kernel after widening. Note that each VE has 128 SIMD8 registers so there is a lot of room for widening on simple kernels. The effect of increasing sub-group size is similar to loop unrolling: while each VE still executes eight 32-bit operations per cycle, the amount of work per work-group interaction is doubled/quadrupled. In SYCL, a programmer can explicitly specify sub-group size using `intel::reqd_sub_group_size({8|16|32})` to override the compiler’s selection.

The table below summarizes the selection criteria of threads and sub-group sizes to keep all GPU resources occupied for TGL:

<table>
<thead>
<tr>
<th>Maximum Threads</th>
<th>Minimum group Size</th>
<th>Minimum Sub-group Size</th>
<th>Maximum Work-group Size</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>8</td>
<td>32</td>
<td>512</td>
<td>Threads x Sub-groupSize &lt;= 512</td>
</tr>
</tbody>
</table>

In general, choosing a larger work-group size has the advantage of reducing the number of rounds of work-group dispatching. Increasing sub-group size can reduce the number of threads required for a work-group at the expense of longer latency and higher register pressure for each sub-group execution.

### 6.3.1 Impact of Work-item Synchronization Within Work-group

Let’s look at a kernel requiring work-item synchronization:
This kernel is similar to the previous example, except it requires work-group barrier synchronization. Work-item synchronization is only available to work-items within the same work-group. You must pick a work-group local range using nd_range and nd_item. All the work-items of a work-group must be allocated to the same Xe-core, which affects Xe-core occupancy and kernel performance.

In this kernel, the local range of work-group is given as range(1, R, 128). Assuming the sub-group size is eight, let’s look at how the values of variable R affect VE occupancy. In the case of R=1, the local range group is (1, 1, 128) and work-group size is 128. The Xe-core allocated for a work-group contains only 16 threads out of 112 available thread contexts (i.e., very low occupancy). However, the system can dispatch 7 work-groups to the same Xe-core to reach full occupancy at the expense of a higher number of dispatches.

In the case of R>4, the work-group size will exceed the system-supported maximum work-group size of 512, and the kernel will fail to launch. In the case of R=4, an Xe-core is only 57% occupied (4/7) and the three unused thread contexts are not sufficient to accommodate another work-group, wasting 43% of the available VE capacities. Note that the driver may still be able to dispatch a partial work-group to an unused Xe-core. However, because of the barrier in the kernel, the partially dispatched work items would not be able to pass the barriers until the rest of the work group is dispatched. In most cases, the kernel’s performance would not benefit much from the partial dispatch. Hence, it is important to avoid this problem by properly choosing the work-group size.

The table below summarizes the tradeoffs between group size, number of threads, Xe-core utilization, and occupancy.

<table>
<thead>
<tr>
<th>Work-items</th>
<th>Group Size</th>
<th>Threads</th>
<th>Xe-core Utilization</th>
<th>Xe-core Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 × 64 × 128 = 524288 (R=1)</td>
<td>128</td>
<td>16</td>
<td>16/112 = 14%</td>
<td>100% with 7 work-groups</td>
</tr>
<tr>
<td>64 × 64 × 128 = 524288 (R=2)</td>
<td>128 × 2</td>
<td>2 × 16 = 32</td>
<td>32/112 = 28.6%</td>
<td>86% with 3 work-groups</td>
</tr>
<tr>
<td>64 × 64 × 128 = 524288 (R=3)</td>
<td>128 × 4</td>
<td>3 × 16 = 48</td>
<td>48/112 = 42.9%</td>
<td>86% with 2 work-groups</td>
</tr>
<tr>
<td>64 × 64 × 128 = 524288 (R=4)</td>
<td>128 × 4</td>
<td>4 × 16 = 64</td>
<td>64/112 = 57%</td>
<td>57% maximum</td>
</tr>
<tr>
<td>64 × 64 × 128 = 524288 (R&gt;4)</td>
<td>640+</td>
<td></td>
<td></td>
<td>Fail to launch</td>
</tr>
</tbody>
</table>
6.3.2 Impact of Local Memory Within Work-group

Let’s look at an example where a kernel allocates local memory for a work-group:

Listing 3: /examples/exec-model/local.cpp

```cpp
auto command_group =
  [&](auto &cgh) {
  // local memory variables shared among work items
  sycl::accessor<int, 1, sycl::access::mode::read_write,
  sycl::access::target::local>
  myLocal(sycl::range(R), cgh);
  cgh.parallel_for(nd_range(sycl::range<3>(64, 64, 128), // global range
  sycl::range<3>(1, R, 128) // local range
  ),
  [=](nghost<3> myGroup) {
    // (work group code)
    myLocal[myGroup.get_local_id()[1]] = ...
  })
`n```

Because work-group local variables are shared among its work-items, they are allocated in a Xe-core’s SLM. Therefore, this work-group must be allocated to a single Xe-core, same as the intra-group synchronization. In addition, you must also weigh the sizes of local variables under different group size options such that the local variables fit within an Xe-core’s 128KB SLM capacity limit.

6.3.3 A Detailed Example

Before concluding this section, let’s look at the hardware occupancies from the variants of a simple vector add example. Using Intel® Iris® Xe graphics from TGL platform as the underlying hardware with the resource parameters specified in Xe-LP (TGL) GPU.

Listing 4: /examples/exec-model/vec-add.cpp

```cpp
int VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b,
  IntArray &sum, int iter) {
  sycl::range num_items{a.size()};
  sycl::buffer a_buf(a);
  sycl::buffer b_buf(b);
  sycl::buffer sum_buf(sum.data(), num_items);
  auto start = std::chrono::steady_clock::now();
  auto e = q.submit([&](auto &h) {
    // Input accessors
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(num_items, [=](auto i) {
      for (int j = 0; j < iter; j++)
```n(continues on next page)
The VectorAdd1 above lets the compiler select the work-group size and SIMD width. In this case, the compiler selects a work-group size of 512 and a SIMD width of 32 because the kernel’s register pressure is low.

**Listing 5**: /examples/exec-model/vec-add.cpp

```cpp
int VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b, IntArray &sum, int iter) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = groups;
    size_t wg_size = 512;
    // get the max wg size instead of 512 size_t wg_size = 512;
    auto start = std::chrono::steady_clock::now();
    q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(
            sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(32)]] {
                size_t grp_id = index.get_group()[0];
                size_t loc_id = index.get_local_id();
                size_t start = grp_id * mysize;
                size_t end = start + mysize;
                for (int i = 0; i < iter; i++)
                    for (size_t i = start + loc_id; i < end; i += wg_size) {
                        sum_acc[i] = a_acc[i] + b_acc[i];
                    }
            });
    });
    auto end = std::chrono::steady_clock::now();
    std::cout << "VectorAdd2 completed on device - took " << (end - start).count()
                << " u-secs\n";
    return ((end - start).count());
} // end VectorAdd2
```
The VectorAdd2 example above explicitly specifies the work-group size of 512, SIMD width of 32, and a variable number of work-groups as a function parameter groups.

Dividing the number of threads by the number of available thread contexts in the GPU gives us an estimate of the GPU hardware occupancy. The following table calculates the GPU hardware occupancy using the TGL Intel® Iris® Xe architecture parameters for each of the above two kernels with various arguments.
The following VTune analyzer chart for VectorAdd2 with various work-group sizes confirms the accuracy of our estimate. The numbers in the grid view vary slightly from the estimate because the grid view gives an average across the entire execution.

The following timeline view gives the occupancy over a period of time. Note that the occupancy metric is ac-
curate for a large part of the kernel execution and tapers off towards the end, due to the varying times at which each of the threads finish their execution.

![Fig. 12: VectorAdd2 timeline view](image)

The kernel `VectorAdd3` shown below is similar to the kernels above with two important differences.

1. It can be instantiated with the number of work-groups, work-group size, and sub-group size as template parameters. This allows us to do experiments to investigate the impact of number of sub-groups and work-groups on thread occupancy.

2. The amount of work done inside the kernel is dramatically increased to ensure that these kernels are resident in the execution units doing work for a substantial amount of time.

Listing 6: `/examples/exec-model/vaddsync.cpp`

```cpp
template <int groups, int wg_size, int sg_size>
int VectorAdd3(sycl::queue &q, const IntArray &a, const IntArray &b, IntArray &sum, int iter) {
  sycl::range num_items{a.size()};

  sycl::buffer a_buf(a);
  sycl::buffer b_buf(b);
  sycl::buffer sum_buf(sum.data(), num_items);
  size_t num_groups = groups;
  auto start = std::chrono::steady_clock::now();
  q.submit([&](auto &h) {
    // Input accessors
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(
        sycl::nd_range<1>({num_groups * wg_size, wg_size}, [=](sycl::nd_item<1> index) {
          size_t grp_id = index.get_group()[0];
          size_t loc_id = index.get_local_id();
          size_t start = grp_id * mysize;
          size_t end = start + mysize;
          for (int i = 0; i < iter; i++)
            for (size_t i = start + loc_id; i < end; i += wg_size) {
              sum_acc[i] = a_acc[i] + b_acc[i];
            }
        }));
  });
  auto end = std::chrono::steady_clock::now();
  std::chrono::duration<double> diff = end - start;
  std::cout << diff.count() << std::endl;
}
```

(continues on next page)
The kernel VectorAdd4 is similar to the kernel VectorAdd3 above except that it has a barrier synchronization at the beginning and end of the kernel execution. This barrier is functionally not needed, but will significantly impact the way in which threads are scheduled on the hardware.

**Listing 7: /examples/exec-model/vaddsync.cpp**

```cpp
#include <iostream>

#include <chrono>

#include <sycl>

template <int groups, int wg_size, int sg_size>
int VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b, 
               IntArray &sum, int iter) {
    int num_items = a.size();

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = groups;
    auto start = std::chrono::steady_clock::now();
    q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(
            sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(sg_size)]] {
                index.barrier(sycl::access::fence_space::local_space);
                size_t grp_id = index.get_group()[0];
                size_t loc_id = index.get_local_id();
                size_t start = grp_id * mysize;
                size_t end = start + mysize;
                for (int i = 0; i < iter; i++) {
                    for (size_t t = start + loc_id; i < end; i += wg_size) {
                        sum_acc[i] = a_acc[i] + b_acc[i];
                    }
                }
            });
    },[]());
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "VectorAdd4<" << groups << "> completed on device - took "
              << (end - start).count() << " u-secs\n";
    return ((end - start).count());
} // end VectorAdd3
```

(continues on next page)
To show how threads are scheduled, the above two kernels are called with 8 work-groups, sub-group size of 8 and work-group size of 320 as shown below. Based on the choice of work-group size and sub-group size, 40 threads per work-group need to be scheduled by the hardware.

**Listing 8: /examples/exec-model/vaddsync.cpp**

```cpp
Initialize(sum);
VectorAdd3<8, 320, 8>(q, a, b, sum, 10000);
Initialize(sum);
VectorAdd4<8, 320, 8>(q, a, b, sum, 10000);
```

The chart from VTune below shows that the measured GPU occupancy for VectorAdd3 and VectorAdd4 kernels.

**Fig. 13: GPU occupancy VectorAdd3, VectorAdd4 kernels**

For the VectorAdd3 kernel, there are two phases for occupancies: 33.3% (224 threads occupancy) and 14.3% (96 threads occupancy) on a TGL machine that has a total of 672 threads. Since there are a total of eight work-groups, with each work-group having 40 threads, there are two Xe-cores (each of which have 112 threads) into which the threads of six work-groups are scheduled. This means that 40 threads each from four work-groups are scheduled, and 32 threads each from two other work-groups are scheduled in the first phase. Then in the second phase, 40 threads from the remaining two work-groups are scheduled for execution.

As seen in the VectorAdd4 kernel, there are three phases of occupancies: 45.3% (304 threads), 39.3% (264 threads), and 11.9% (80 threads). In the first phase, all eight work-groups are scheduled together on 3 Xe-cores,
with two Xe-cores getting 112 threads each (80 from two work-groups and 32 from one work-group) and one Xe-core getting 80 threads (from two work-groups). In the second phase, one work-group completed execution, which gives us occupancy of (304-40=264). In the last phase, the remaining eight threads of two work-groups are scheduled and these complete the execution.

The same kernels as above when run with a work-group size that is a multiple of the number of threads in a Xe-core and lot more work-groups gets good utilization of the hardware achieving close to 100% occupancy, as shown below.

```cpp
Listing 9: /examples/exec-model/vaddsync.cpp

1. Initialize(sum);
2. VectorAdd3<24, 224, 8>(q, a, b, sum, 10000);
3. Initialize(sum);
4. VectorAdd4<24, 224, 8>(q, a, b, sum, 10000);
```

This kernel execution has a different thread occupancy since we have many more threads and also the work-group size is a multiple of the number of threads in a Xe-core. This is shown below in the thread occupancy metric on the VTune timeline.

Note that the above schedule is a guess based on the different occupancy numbers, since we do not yet have a way to examine the per slice based occupancy numbers.

You can run different experiments with the above kernels to gain better understanding of how the GPU hardware schedules the software threads on the Execution Units. Be careful about the work-group and sub-group sizes, in addition to a large number of work-groups, to ensure effective utilization of the GPU hardware.
6.4 Intel® GPU Occupancy Calculator

In summary, a SYCL work-group is typically dispatched to an Xe-core. All the work-items in a work-group share the same SLM of an Xe-core for intra work-group thread barriers and memory fence synchronization. Multiple work-groups can be dispatched to the same Xe-core if there are sufficient VE ALUs, SLM, and thread contexts to accommodate them.

You can achieve higher performance by fully utilizing all available Xe-cores. Parameters affecting a kernel's GPU occupancy are work-group size and SIMD sub-group size, which also determines the number of threads in the work-group.

The Intel® GPU Occupancy Calculator can be used to calculate the occupancy on an Intel GPU for a given kernel, and its work-group parameters.
7.0 Kernels

A kernel is the unit of computation in the oneAPI offload model. By submitting a kernel on an iteration space, you are requesting that the computation be applied to the specified data objects.

In this section we cover topics related to the coding, submission, and execution of kernels.

7.1 Sub-groups and SIMD Vectorization

The index space of an ND-Range kernel is divided into work-groups, sub-groups, and work-items. A work-item is the basic unit. A collection of work-items form a sub-group, and a collection of sub-groups form a work-group. The mapping of work-items and work-groups to hardware vector engines (VE) is implementation-dependent. All the work-groups run concurrently but may be scheduled to run at different times depending on availability of resources. Work-group execution may or may not be preempted depending on the capabilities of underlying hardware. Work-items in the same work-group are guaranteed to run concurrently. Work-items in the same sub-group may have additional scheduling guarantees and have access to additional functionality.

A sub-group is a collection of contiguous work-items in the global index space that execute in the same VE thread. When the device compiler compiles the kernel, multiple work-items are packed into a sub-group by vectorization so the generated SIMD instruction stream can perform tasks of multiple work-items simultaneously. Properly partitioning work-items into sub-groups can make a big performance difference.

Let’s start with a simple example illustrating sub-groups:

```
Listing 10: /examples/sub-group/sub-group-0.cpp

1 q.submit([&](auto &h) {
2     sycl::stream out(65536, 256, h);
3     h.parallel_for(sycl::nd_range(sycl::range{32}, sycl::range{32}),
4         [=](sycl::nd_item<1> it) {
5             int groupId = it.get_group(0);
6             int globalId = it.get_global_linear_id();
7             sycl::ext::oneapi::sub_group sg = it.get_sub_group();
8             int sgSize = sg.get_local_range()[0];
9             int sgGroupId = sg.get_group_id()[0];
10            int sgId = sg.get_local_id()[0];

11                out << "globalId = " << sycl::setw(2) << globalId
12                   << " groupId = " << groupId
13                   << " sgGroupId = " << sgGroupId << " sgId = " << sgId
14                   << " sgSize = " << sycl::setw(2) << sgSize
15                   << sycl::endl;
16         });
17 });
```

The output of this example may look like this:
Each sub-group in this example has 16 work-items, or the sub-group size is 16. This means each thread simultaneously executes 16 work-items and 32 work-items are executed by two VE threads.

By default, the compiler selects a sub-group size using device-specific information and a few heuristics. The user can override the compiler’s selection using the kernel attribute `intel::reqd_sub_group_size` to specify the maximum sub-group size. Sometimes, not always, explicitly requesting a sub-group size may help performance.

```
Listing 11: /examples/sub-group/sub-group-1.cpp

q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{32}, sycl::range{32}),
                   [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(32)]] {
                   int groupId = it.get_group(0);
                   int globalId = it.get_global_linear_id();
                   sycl::ext::oneapi::sub_group sg = it.get_sub_group();
                   int szSize = sg.get_local_range()[0];
                });
```
![Image of a page from a document with code and text]

```cpp
int sgGroupId = sg.get_group_id()[0];
int sgId = sg.get_local_id()[0];

out << "globalId = " << sycl::setw(2) << globalId
     << " groupId = " << groupId
     << " sgGroupId = " << sgGroupId << " sgId = " << sgId
     << " sgSize = " << sycl::setw(2) << sgSize
     << sycl::endl;
```

The output will be:

<table>
<thead>
<tr>
<th>Device: Intel(R) Gen12HP</th>
</tr>
</thead>
<tbody>
<tr>
<td>globalId = 0 groupId = 0 sgGroupId = 0 sgId = 0 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 1 groupId = 0 sgGroupId = 0 sgId = 1 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 2 groupId = 0 sgGroupId = 0 sgId = 2 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 3 groupId = 0 sgGroupId = 0 sgId = 3 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 4 groupId = 0 sgGroupId = 0 sgId = 4 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 5 groupId = 0 sgGroupId = 0 sgId = 5 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 6 groupId = 0 sgGroupId = 0 sgId = 6 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 7 groupId = 0 sgGroupId = 0 sgId = 7 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 8 groupId = 0 sgGroupId = 0 sgId = 8 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 9 groupId = 0 sgGroupId = 0 sgId = 9 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 10 groupId = 0 sgGroupId = 0 sgId = 10 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 11 groupId = 0 sgGroupId = 0 sgId = 11 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 12 groupId = 0 sgGroupId = 0 sgId = 12 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 13 groupId = 0 sgGroupId = 0 sgId = 13 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 14 groupId = 0 sgGroupId = 0 sgId = 14 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 15 groupId = 0 sgGroupId = 0 sgId = 15 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 16 groupId = 0 sgGroupId = 0 sgId = 16 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 17 groupId = 0 sgGroupId = 0 sgId = 17 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 18 groupId = 0 sgGroupId = 0 sgId = 18 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 19 groupId = 0 sgGroupId = 0 sgId = 19 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 20 groupId = 0 sgGroupId = 0 sgId = 20 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 21 groupId = 0 sgGroupId = 0 sgId = 21 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 22 groupId = 0 sgGroupId = 0 sgId = 22 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 23 groupId = 0 sgGroupId = 0 sgId = 23 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 24 groupId = 0 sgGroupId = 0 sgId = 24 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 25 groupId = 0 sgGroupId = 0 sgId = 25 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 26 groupId = 0 sgGroupId = 0 sgId = 26 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 27 groupId = 0 sgGroupId = 0 sgId = 27 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 28 groupId = 0 sgGroupId = 0 sgId = 28 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 29 groupId = 0 sgGroupId = 0 sgId = 29 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 30 groupId = 0 sgGroupId = 0 sgId = 30 sgSize = 32</td>
</tr>
<tr>
<td>globalId = 31 groupId = 0 sgGroupId = 0 sgId = 31 sgSize = 32</td>
</tr>
</tbody>
</table>

The valid sub-group sizes are device dependent. You can query the device to get this information:
The valid sub-group sizes supported may be:

Device: Intel(R) Gen12HP
Subgroup Sizes: 8 16 32

Next, we will show how to use sub-groups to improve performance.

### 7.1.1 Vectorization and Memory Access

The Intel® graphics device has multiple VEs. Each VE is a multithreaded SIMD processor. The compiler generates SIMD instructions to pack multiple work-items in a sub-group to execute simultaneously in a VE thread. The SIMD width (thus the sub-group size), selected by the compiler is based on device characteristics and heuristics, or requested explicitly by the kernel, and can be 8, 16, or 32.

Given a SIMD width, maximizing SIMD lane utilization gives optimal instruction performance. If one or more lanes (or kernel instances or work items) diverge, the thread executes both branch paths before the paths merge later, increasing the dynamic instruction count. SIMD divergence negatively impacts performance. The compiler works to minimize divergence, but it helps to avoid divergence in the source code, if possible.

How memory is accessed in work-items affects how memory is accessed in the sub-group or how the SIMD lanes are utilized. Accessing contiguous memory in a work-item is often not optimal. For example:

```cpp
Listing 13: /examples/sub-group/sub-group-2.cpp
```

```cpp
cconstexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
        [=](sycl::nd_item<1> it) {
            int i = it.get_global_linear_id();
            i = i * 16;
            for (int j = i; j < (i + 16); j++) {
                data[j] = -1;
            }
        });
    q.wait();
```

This simple kernel initializes an array of 1024 x 1024 integers. Each work-item initializes 16 contiguous integers. Assuming the sub-group size chosen by the compiler is 16, 256 integers are initialized in each sub-group or thread. However, the stores in 16 SIMD lanes are scattered.
Instead of initializing 16 contiguous integers in a work-item, initializing 16 contiguous integers in one SIMD instruction is more efficient.

**Listing 14: /examples/sub-group/sub-group-3.cpp**

```cpp
constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}), [=](sycl::nd_item<1> it) {
        int i = it.get_global_linear_id();
        sycl::ext::oneapi::sub_group sg = it.get_sub_group();
        int sgSize = sg.get_local_range()[0];
        i = (i / sgSize) * sgSize + (i % sgSize);
        for (int j = 0; j < sgSize * 16; j += sgSize) {
            data[i + j] = -1;
        }
    });
});
```

We use memory writes in our examples, but the same technique is applicable to memory reads as well.

**Listing 15: /examples/sub-group/sub-group-4.cpp**

```cpp
constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}), [=](sycl::nd_item<1> it) {
        int i = it.get_global_linear_id();
        i = i * 16;
        for (int j = i; j < (i + 16); j++) {
            data[j] = data2[j];
        }
    });
});
```

This kernel copies an array of 1024 x 1024 integers to another integer array of the same size. Each work-item copies 16 contiguous integers. However, the reads from `data2` are gathered and stores to `data` are scattered. It will be more efficient to change the code to read and store contiguous integers in each sub-group instead of each work-item.

**Listing 16: /examples/sub-group/sub-group-5.cpp**

```cpp
constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);
```
auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N / 16}, sycl::range{32}),
                 [=](sycl::nd_item<1> it) {
        int i = it.get_global_linear_id();
        sycl::ext::oneapi::sub_group sg = it.get_sub_group();
        int sgSize = sg.get_local_range()[0];
        i = (i / sgSize) * sgSize + (i % sgSize);
        for (int j = 0; j < sgSize * 16; j += sgSize) {
            data[i + j] = data2[i + j];
        }
    });
});

You may have noticed that the sub-group size 16 was explicitly requested. When you use sub-group functions, it is always good to override the compiler choice to make sure the sub-group size always matches what you expect. Please also note that, at the time of writing, block load/store does not work with sub-group size 32 on current Intel hardware, so the group size explicitly requested must be 16 or smaller.
### 7.1.2 Data Sharing

Because the work-items in a sub-group execute in the same thread, it is more efficient to share data between work-items, even if the data is private to each work-item. Sharing data in a sub-group is more efficient than sharing data in a work-group using shared local memory, or SLM. One way to share data among work-items in a sub-group is to use shuffle functions.

#### Listing 18: /examples/sub-group/transpose.cpp

```cpp
constexpr size_t blockSize = 16;
sycl::buffer<uint, 2> m(matrix.data(), sycl::range<2>(N, N));

auto e = q.submit([&](auto &h)
{
    sycl::accessor marr(m, h);
    sycl::accessor<uint, 2, sycl::access::mode::read_write, sycl::access::target::local> barr1(sycl::range<2>(blockSize, blockSize), h);
    sycl::accessor<uint, 2, sycl::access::mode::read_write, sycl::access::target::local> barr2(sycl::range<2>(blockSize, blockSize), h);

    h.parallel_for(
        sycl::nd_range<2>(sycl::range<2>(N / blockSize, N), sycl::range<2>(1, blockSize)),
        [=](sycl::nd_item<2> it) [[intel::reqd_sub_group_size(16)]]
        {
            int gi = it.get_group(0);
            int gj = it.get_group(1);

            sycl::ext::oneapi::sub_group sg = it.get_sub_group();
            uint sgId = sg.get_local_id()[0];

            uint bcol[blockSize];
            int ai = blockSize * gi;
            int aj = blockSize * gj;

            for (uint k = 0; k < blockSize; k++)
            {
                bcol[k] = sg.load(marr.get_pointer() + (ai + k) * N + aj);
            }

            uint tcol[blockSize];
            for (uint n = 0; n < blockSize; n++)
            {
                if (sgId == n)
                {
                    for (uint k = 0; k < blockSize; k++)
                    {
                        tcol[k] = sg.shuffle(bcol[n], k);
                    }
                }
            }

            for (uint k = 0; k < blockSize; k++)
            {
                sg.store(marr.get_pointer() + (ai + k) * N + aj, tcol[k]);
            }
});
```
This kernel transposes a 16 x 16 matrix. It looks more complicated than the previous examples, but the idea is simple: a sub-group loads a 16 x 16 sub-matrix, then the sub-matrix is transposed using the sub-group shuffle functions. There is only one sub-matrix and the sub-matrix is the matrix so only one sub-group is needed. A bigger matrix, say 4096 x 4096, can be transposed using the same technique: each sub-group loads a sub-matrix, then the sub-matrices are transposed using the sub-group shuffle functions. This is left to the reader as an exercise.

DPC++ has multiple variants of sub-group shuffle functions available. Each variant is optimized for its specific purpose on specific devices. It is always a good idea to use these optimized functions (if they fit your needs) instead of creating your own.

### 7.1.3 Sub-group Size vs. Maximum Sub-group Size

So far in our examples, the work-group size is divisible by the sub-group size and both the work-group size and the sub-group size (either required by the user or automatically picked by the compiler are powers of two). The sub-group size and maximum sub-group size are the same if the work-group size is divisible by the maximum sub-group size and both sizes are powers of two. But what happens if the work-group size is not divisible by the sub-group size? Consider the following example:

**Listing 19: /examples/sub-group/sg-max-size.cpp**

```cpp
auto e = q.submit([&](auto &h) {
    sycl::stream out(65536, 128, h);
    h.parallel_for(sycl::nd_range<1>(7, 7),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(8)]] {
            int i = it.get_global_linear_id();
            sycl::ext::oneapi::sub_group sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgMaxSize = sg.get_max_local_range()[0];
            int sId = sg.get_local_id()[0];
            int j = data[i];
            int k = data[i + sgSize];
            out << "globalId = " << i << " sgMaxSize = " << sgMaxSize
                 << " sgSize = " << sgSize << " sId = " << sId
                 << " j = " << j << " k = " << k << sycl::endl;
        });
    q.wait();
}
```

The output of this example looks like this:

```
globalId = 0 sgMaxSize = 8 sgSize = 7 sId = 0 j = 0 k = 7
globalId = 1 sgMaxSize = 8 sgSize = 7 sId = 1 j = 1 k = 8
globalId = 2 sgMaxSize = 8 sgSize = 7 sId = 2 j = 2 k = 9
globalId = 3 sgMaxSize = 8 sgSize = 7 sId = 3 j = 3 k = 10
globalId = 4 sgMaxSize = 8 sgSize = 7 sId = 4 j = 4 k = 11
globalId = 5 sgMaxSize = 8 sgSize = 7 sId = 5 j = 5 k = 12
globalId = 6 sgMaxSize = 8 sgSize = 7 sId = 6 j = 6 k = 13
```

The sub-group size is seven, though the maximum sub-group size is still eight! The maximum sub-group size is actually the SIMD width so it does not change, but there are less than eight work-items in the sub-group, so the
sub-group size is seven. So be careful when your work-group size is not divisible by the maximum sub-group size. The last sub-group with fewer work-items may need to be specially handled.

7.2 Removing Conditional Checks

In Sub-groups and SIMD Vectorization, we learned that SIMD divergence can negatively affect performance. If all work items in a sub-group execute the same instruction, the SIMD lanes are maximally utilized. If one or more work items take a divergent path, then both paths have to be executed before they merge.

Divergence is caused by conditional checks, though not all conditional checks cause divergence. Some conditional checks, even when they do not cause SIMD divergence, can still be performance hazards. In general, removing conditional checks can help performance.

7.2.1 Padding Buffers to Remove Conditional Checks

Look at the convolution example from Shared Local Memory:

```cpp
Listing 20: /examples/slm/convolution-global.cpp

sycl::buffer<int> ibuf(input.data(), N);
sycl::buffer<int> obuf(output.data(), N);
sycl::buffer<int> kbuf(kernel.data(), M);
auto e = q.submit([&](auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    sycl::accessor kacc(kbuf, h, sycl::read_only);
    h.parallel_for(sycl::nd_range<1>({sycl::range(N), sycl::range(256)}),
               [=](sycl::nd_item<1> it) {
                  int i = it.get_global_linear_id();
                  int group = it.get_group()[0];
                  int gSize = it.get_local_range()[0];
                  int t = 0;
                  int _M = static_cast<int>(M);
                  int _N = static_cast<int>(N);

                  if ( (group == 0) || (group == _N / gSize - 1) ) {
                      if (i < _M / 2) {
                          for (int j = _M / 2 - i, k = 0; j < _M; ++j, ++k) {
                              t += iacc[k] * kacc[j];
                          }
                      } else {
                          if (i + _M / 2 >= _N) {
                              for (int j = 0, k = i - _M / 2;
                                   j < _M / 2 + _N - i; ++j, ++k) {
                                  t += iacc[k] * kacc[j];
                              }
                          } else {
                              for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
                                  t += iacc[k] * kacc[j];
                              }
                          }
                      }
                  }
              });
```
The nested if-then-else conditional checks are necessary to take care of the first and last 128 elements in the input so indexing will not run out of bounds. If we pad enough 0s before and after the input array, these conditional checks can be safely removed:

```
Listing 21
/examples/conditionals/convolution-global-conditionals.cpp

std::vector<int> input(N + M / 2 + M / 2);
std::vector<int> output(N);
std::vector<int> kernel(M);
srand(2009);
for (size_t i = M / 2; i < N + M / 2; ++i) {
    input[i] = rand();
}
for (size_t i = 0; i < M / 2; ++i) {
    input[i] = 0;
    input[i + N + M / 2] = 0;
}
for (size_t i = 0; i < M; ++i) {
    kernel[i] = rand();
}

{sycl::buffer<int> ibuf(input.data(), N + M / 2 + M / 2);
sycl::buffer<int> obuf(output.data(), N);
sycl::buffer<int> kbuf(kernel.data(), M);

auto e = q.submit([&](auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    sycl::accessor kacc(kbuf, h, sycl::read_only);
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{256}),
       [=](sycl::nd_item<1> it) {

(continues on next page)
7.2.2 Replacing Conditional Checks with Relational Functions

Another way to remove conditional checks is to replace them with relational functions, especially built-in relational functions. It is strongly recommended to use a built-in function if one is available. DPC++ provides a rich set of built-in relational functions like select(), min(), max(). In many cases you can use these functions to replace conditional checks and achieve better performance.

Consider the convolution example again. The if-then-else conditional checks can be replaced with built-in functions min() and max().

Listing 22: /examples/conditionals/convolution-global-conditionals-min-max.cpp
7.3 Registerization and Avoid Register Spills

7.3.1 Registers and Performance

Register is the fastest storage in the memory hierarchy. Keeping data in registers as long as possible is critical to performance. However, register space is limited and much smaller than memory space. The current generation of Intel® GPUs, for example, has 128 general-purpose registers, each 32 bytes wide by default for each XVE thread. Though the compiler aims to assign as many variables to registers as possible, the limited number of registers can be allocated only to a small set of variables at any point during execution. A given register can hold different variables at different times because different sets of variables are needed at different times. If there are not enough registers to hold all the variables, register can spill, or some variables currently in the registers can be moved to memory to make room for other variables.

In DPC++, the compiler allocates registers to private variables in work items. Multiple work items in a sub-group are packed into one XVE thread. By default, the compiler uses register pressure as one of the heuristics to choose SIMD width or sub-group size. High register pressures can result in smaller sub-group size (for example 8 instead of 16) if a sub-group size is not explicitly requested. It can also cause register spilling or cause certain variables not to be promoted to registers.

The hardware may not be fully utilized if sub-group size or SIMD width is not the maximum the hardware supports. Register spilling can cause significant performance degradation, especially when spills occur inside hot loops. When variables are not promoted to registers, accesses to these variables incur significant increase of memory traffic.

Though the compiler uses intelligent algorithms to allocate variables in registers and to minimize register spills, optimizations by developers can help the compiler to do a better job and often make a big performance difference.

7.3.2 Optimization Techniques

The following techniques can reduce register pressure:

- Keep live ranges of private variables as short as possible.
  Though the compiler schedules instructions and optimizes the distances, in some cases moving the loading and using the same variable closer or removing certain dependencies in the source can help the compiler do a better job.

- Avoid excessive loop unrolling.
  Loop unrolling exposes opportunities for instruction scheduling optimization by the compiler and thus can improve performance. However, temporary variables introduced by unrolling may increase pressure on register allocation and cause register spilling. It is always a good idea to compare the performance with and without loop unrolling and different times of unrolls to decide if a loop should be unrolled or how many times to unroll it.
- Prefer USM pointers.
  
  A buffer accessor takes more space than a USM pointer. If you can choose between USM pointers and buffer accessors, choose USM pointers.

- Recompute cheap-to-compute values on-demand that otherwise would be held in registers for a long time.

- Avoid big arrays or large structures, or break an array of big structures into multiple arrays of small structures.

  For example, an array of `sycl::float4`:

  ```cpp
  sycl::float4 v[8];
  ```

  can be broken into 4 arrays of `float`:

  ```cpp
  float x[8]; float y[8]; float z[8]; float w[8];
  ```

  All or part of the 4 arrays of `float` have a better chance to be allocated in registers than the array of `sycl::float4`.

- Break a large loop into multiple small loops to reduce the number of simultaneously live variables.

- Choose smaller sized data types if possible.

- Do not declare private variables as volatile.

- Share registers in a sub-group.

- Use sub-group block load/store if possible.

- Use shared local memory.

The list here is not exhaustive.

The rest of this chapter shows how to apply these techniques, especially the last four, and gives examples.

### 7.3.3 Choosing Smaller Data Types

Listing 23: /examples/registers/histogram32-long.cpp

```cpp
constexpr int blockSize = 256;
constexpr int NUM_BINS = 32;

std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
```
This example calculates histograms with a bin size of 32. Each work item has 32 private bins of unsigned long data type. Because of the large storage required, the private bins cannot fit in registers, resulting in poor performance.

With blockSize 256, the maximum value of each private histogram bin will not exceed the maximum value of an unsigned integer. Instead of unsigned long type for private histogram bins, we can use unsigned integers to reduce register pressure so the private bins can fit in registers. This simple change makes significant performance difference.

**Listing 24: /examples/registers/histogram32-int.cpp**

```cpp
constexpr int blockSize = 256;
constexpr int NUM_BINS = 32;
std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);
auto e = q.submit([&](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    for (int k = 0; k < NUM_BINS; k++) {
        for (int i = 0; i < 8; i++) {
            unsigned int c = x & 0x1FU;
            histogram[c] += 1;
            x = x >> 8;
        }
    }
    for (int k = 0; k < NUM_BINS; k++) {  
        hacc[k].fetch_add(histogram[k]);
    }
});
```
auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);

h.parallel_for(
    sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}),
    [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
        int group = it.get_group()[0];
        int gSize = it.get_local_range()[0];
        sycl::ext::oneapi::sub_group sg = it.get_sub_group();
        int sgSize = sg.get_local_range()[0];
        int sgGroup = sg.get_group_id()[0];

        unsigned int histogram[NUM_BINS]; // histogram bins take less storage
        // with smaller data type
        for (int k = 0; k < NUM_BINS; k++) {
            histogram[k] = 0;
        }
        for (int k = 0; k < blockSize; k++) {
            unsigned long x = sg.load(macc.get_pointer() + group * gSize * blockSize +
                                      sgGroup * sgSize * blockSize + sgSize * k);
            #pragma unroll
            for (int i = 0; i < 8; i++) {
                unsigned int c = x & 0x1FU;
                histogram[c] += 1;
                x = x >> 8;
            }
        }

        for (int k = 0; k < NUM_BINS; k++) {
            hacc[k].fetch_add(histogram[k]);
        }
    });
7.3.4 Do Not Declare Private Variables as Volatile

Now we make a small change to the code example:

Listing 25: /examples/registers/histogram32-int-volatile.cpp

```cpp
const expr int blockSize = 256;
const expr int NUM_BINS = 32;

std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&](auto &h) {
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}),
                   [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
        int group = it.get_group()[0];
        int gSize = it.get_local_range()[0];
        sycl::ext::oneapi::sub_group sg = it.get_sub_group();
        int sgSize = sg.get_local_range()[0];
        int sgGroup = sg.get_group_id()[0];

        volatile unsigned int histogram[NUM_BINS]; // volatile variables will not
        // be assigned to any registers

        for (int k = 0; k < NUM_BINS; k++) {
            histogram[k] = 0;
        }
        for (int k = 0; k < blockSize; k++) {
            unsigned long x = sg.load(
                macc.get_pointer() + group * gSize * blockSize +
                sgGroup * sgSize * blockSize + sgSize * k);
            #pragma unroll
            for (int i = 0; i < 8; i++) {
                unsigned int c = x & 0x1FU;
                histogram[c] += 1;
                x = x >> 8;
            }
        }
        for (int k = 0; k < NUM_BINS; k++) {
            hacc[k].fetch_add(histogram[k]);
        }
    });
```

The private histogram array is qualified as a volatile array. Volatile variables are not prompted to registers because their values may change between two different load operations.

There is really no reason for the private histogram array to be volatile, because it is only accessible by the local
7.3.5 Sharing Registers in a Sub-group

Now we increase the histogram bins to 256:

**Listing 26: /examples/registers/histogram256-int.cpp**

```cpp
constexpr int blockSize = 256;
constexpr int NUM_BINS = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            sycl::ext::oneapi::sub_group sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            unsigned int
                histogram[NUM_BINS]; // histogram bins take too much storage to be
            // promoted to registers
            for (int k = 0; k < NUM_BINS; k++) {
                histogram[k] = 0;
            }
            for (int k = 0; k < blockSize; k++) {
                unsigned long x =
                    sg.load(macc.get_pointer()) + group * gSize * blockSize +
                    sgGroup * sgSize * blockSize + sgSize * k);
                #pragma unroll
                for (int i = 0; i < 8; i++) {
                    unsigned int c = x & 0x1FU;
                    histogram[c] += 1;
                    x = x >> 8;
                }
            }
            for (int k = 0; k < NUM_BINS; k++) {
                hacc[k].fetch_add(histogram[k]);
            }
        });
```}

With 256 histogram bins, the performance degrades even with smaller data type unsigned integer. The storage of the private bins in each work item is too large for registers.
If the sub-group size is 16 as requested, we know that 16 work items are packed into one EU thread. We also know work items in the same sub-group can communicate and share data with each other very efficiently. If the work items in the same sub-group share the private histogram bins, only 256 private bins are needed for the whole sub-group, or 16 private bins for each work item instead.

**Fig. 15:** Each Work Item Has 256 Private Histogram Bins

**Fig. 16:** Sub-group Has 256 Private Histogram Bins
To share the histogram bins in the sub-group, each work item broadcasts its input data to every work item in the same sub-group. The work item that owns the corresponding histogram bin does the update.

**Listing 27:**
/examples/registers/histogram256-int-shared-private.cpp

```cpp
constexpr int blockSize = 256;
constexpr int NUM_BINS = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&](auto h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            sycl::ext::oneapi::sub_group sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            unsigned int histogram[NUM_BINS / 16]; // histogram bins take too much storage
            // to be promoted to registers
            for (int k = 0; k < NUM_BINS / 16; k++) {
                histogram[k] = 0;
            }

            for (int k = 0; k < blockSize; k++) {
                unsigned long x = sg.load(macc.get_pointer() + group * gSize * blockSize + sgGroup * sgSize * blockSize + sgSize * k);
                // subgroup size is 16
                #pragma unroll
                for (int j = 0; j < 16; j++) {
                    unsigned long y = sycl::group_broadcast(sg, x, j);
                    #pragma unroll
                    for (int i = 0; i < 8; i++) {
                        unsigned int c = y & 0xFF;
                        // (c & 0xF) is the workitem in which the bin resides
                        // (c >> 4) is the bin index
                        if (sg.get_local_id()[0] == (c & 0xF)) {
                            histogram[c >> 4] += 1;
                        }
                        y = y >> 8;
                    }
                }
            }
        });
    });
```

(continues on next page)
7.3.6 Using Sub-group Block Load/Store

Memory loads/stores are vectorized. Each lane of a vector load/store instruction has its own address and data. Both addresses and data take register space. For example:

Listing 28: /examples/registers/non-block-load-store.cpp

```cpp
constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
        [=](sycl::nd_item<1> it) {
            int i = it.get_global_linear_id();
            data[i] = data2[i];
        });
});
```

The memory loads and stores in the statement

```
data[i] = data2[i];
```

are vectorized and each vector lane has its own address. Assuming the SIMD width or the sub-group size is 16, total register space for addresses of the 16 lanes is 128 bytes. If each GRF register is 32-byte wide, 4 GRF registers are needed for the addresses.

Noticing the addresses are contiguous, we can use sub-group block load/store built-ins to save register space for addresses:

Listing 29: /examples/registers/block-load-store.cpp

```cpp
constexpr int N = 1024 * 1024;
int *data = sycl::malloc_shared<int>(N, q);
int *data2 = sycl::malloc_shared<int>(N, q);
memset(data2, 0xFF, sizeof(int) * N);

auto e = q.submit([&](auto &h) {
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
        [=](sycl::nd_item<1> it) [[intel::reqd_sub_group_size(16)]] {
            int x;
```

(continues on next page)
using global_ptr = 
    sycl::multiptr<int, sycl::access::address_space::global_space>;
int base = (it.get_group(0) * 32 + 
               sg.get_group_id()[0] * sg.get_local_range()[0]);
    x = sg.load(global_ptr(&(data2[base + 0])));
    sg.store(global_ptr(&(data[base + 0])), x);
});

The statements

```
x = sg.load(global_ptr(&(data2[base + 0]))); sg.store(global_ptr(&(data[base + 0])), x);
```

each loads/stores a contiguous block of memory and the compiler will compile these 2 statements into special
memory block load/store instructions. And because it is a contiguous memory block, we only need the starting
address of the block. So 8, instead of 128, bytes of actual register space, or at most 1 register, is used for the
address for each block load/store.

### 7.3.7 Using Shared Local Memory

If the number of histogram bins gets larger than, for example, 1024, there will not be enough register space for pri-
ivate bins even the private bins are shared in the same sub-group. To reduce memory traffic, the local histogram
bins can be allocated in the shared local memory and shared by work items in the same work-group. Refer to the
“Shared Local Memory” chapter and see how it is done in the histogram example there.

### 7.4 Shared Local Memory

Often work-items need to share data and communicate with each other. On one hand, all work-items in all work-
groups can access global memory, so data sharing and communication can occur through global memory. How-
ever, due to its lower bandwidth and higher latency, sharing and communication through global memory is less efficient. On the other hand, work-items in a sub-group executing simultaneously in a vector engine (VE) thread can share data and communicate with each other very efficiently, but the number of work-items in a sub-
group is usually small and the scope of data sharing and communication is very limited. Memory with higher
bandwidth and lower latency accessible to a bigger scope of work-items is very desirable for data sharing com-
munication among work-items. The shared local memory (SLM) in Intel® GPUs is designed for this purpose.

Each Xe-core of Intel GPUs has its own SLM. Access to the SLM is limited to the VEs in the Xe-core or work-
items in the same work-group scheduled to execute on the VEs of the same Xe-core. It is local to a Xe-core (or
work-group) and shared by VEs in the same Xe-core (or work-items in the same work-group), so it is called SLM.
Because it is on-chip in each Xe-core, the SLM has much higher bandwidth and much lower latency than global
memory. Because it is accessible to all work-items in a work-group, the SLM can accommodate data sharing and
communication among hundreds of work-items, depending on the work-group size.

It is often helpful to think of SLM as a work-group managed cache. When a work-group starts, work-items in the
work-group can explicitly load data from global memory into SLM. The data stays in SLM during the lifetime of the
work-group for faster access. Before the work-group finishes, the data in the SLM can be explicitly written back
to the global memory by the work-items. After the work-group completes execution, the data in SLM is also gone
and invalid. Data consistency between the SLM and the global memory is the program’s responsibility. Properly
using SLM can make a significant performance difference.
7.4.1 Shared Local Memory Size and Work-group Size

Because it is on-chip, the SLM has limited size. How much memory is available to a work-group is device-dependent and can be obtained by querying the device, e.g.:

```
Listing 30: /examples/slm/slm-size.cpp

std::cout << "Local Memory Size: "
    << q.get_device().get_info<sycl::info::device::local_mem_size>()
    << std::endl;
```

The output may look like:

**Local Memory Size: 65536**

The unit of the size is a byte. So this GPU device has 65,536 bytes or 64KB SLM for each work-group.

It is important to know the maximum SLM size a work-group can have. In a lot of cases, the total size of SLM available to a work-group is a non-constant function of the number of work-items in the work-group. The maximum SLM size can limit the total number of work-items in a group, i.e. work-group size. For example, if the maximum SLM size is 64KB and each work-item needs 512 bytes of SLM, the maximum work-group size cannot exceed 128.

7.4.2 Bank Conflicts

The SLM is divided into equally sized memory banks that can be accessed simultaneously for high bandwidth. The total number of banks is device-dependent. At the time of writing, 64 consecutive bytes are stored in 16 consecutive banks at 4-byte (32-bit) granularity. Requests for access to different banks can be serviced in parallel, but requests to different addresses in the same bank cause a bank conflict and are serialized. Bank conflicts adversely affect performance. Consider this example:

```
Listing 31: /examples/slm/slm-bank-s16.cpp

constexpr int N = 32;
int *data = sycl::malloc_shared<int>(N, q);

auto e = q.submit([&](auto &h) {
    sycl::accessor<int, 1, sycl::access::mode::read_write,
        sycl::access::target::local>
        slm(sycl::range(32 * 64), h);
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
        [=](sycl::nd_item<1> it) {
            int i = it.get_global_linear_id();
            int j = it.get_local_linear_id();
            slm[j * 16] = 0;
            it.barrier(sycl::access::fence_space::local_space);
            for (int m = 0; m < 1024 * 1024; m++) {
                slm[j * 16] += i * m;
                it.barrier(sycl::access::fence_space::local_space);
            }
        });
});
```
If the number of banks is 16, all work-items in the above example will read from and write to different addresses in the same bank. The memory bandwidth is $1/16$ of full bandwidth.

The next example instead does not have SLM bank conflicts and achieves full memory bandwidth because every work-item reads from and writes to different addresses in different banks.

**Listing 32: examples/slm/slm-bank-s1.cpp**

```cpp
constexpr int N = 32;
int *data = sycl::malloc_shared<int>(N, q);

auto e = q.submit([&](auto &h) {
    sycl::accessor<int, 1, sycl::access::mode::read_write,
        sycl::access::target::local>
    slm(sycl::range(32 * 64), h);
    h.parallel_for(sycl::nd_range(sycl::range{N}, sycl::range{32}),
        [=](sycl::nd_item<1> it) {
            int i = it.get_global_linear_id();
            int j = it.get_local_linear_id();

            slm[j] = 0;
            it.barrier(sycl::access::fence_space::local_space);

            for (int m = 0; m < 1024 * 1024; m++) {
                slm[j] += i * m;
                it.barrier(sycl::access::fence_space::local_space);
            }

            data[i] = slm[j];
        });
});
```

### 7.4.3 Data Sharing and Work-group Barriers

Let us consider the histogram with 256 bins example from the “Avoiding Register Spills” chapter once again.

**Listing 33:**

/examples/registers/histogram256-int-shared-private.cpp

```cpp
constexpr int blockSize = 256;
constexpr int NUM_BINS = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);
```
This example has been optimized to use the integer data type instead of long and to share registers in the subgroup so that the private histogram bins can fit in registers for optimal performance. If you need a larger bin size (e.g., 1024), it is inevitable that the private histogram bins will spill to global memory.
The histogram bins can be shared by work-items in a work-group as long as each bin is updated atomically.

**Listing 34: /examples/slm/histogram-slm-1024.cpp**

```
constexpr int NUM_BINS = 1024;
constexpr int blockSize = 256;

std::vector<unsigned long> hist(NUM_BINS, 0);
sycl::buffer<unsigned long, 1> mbuf(input.data(), N);
sycl::buffer<unsigned long, 1> hbuf(hist.data(), NUM_BINS);

auto e = q.submit([&](auto &h) {
    sycl::accessor macc(mbuf, h, sycl::read_only);
    auto hacc = hbuf.get_access<sycl::access::mode::atomic>(h);
    sycl::accessor<unsigned int, 1, sycl::access::mode::atomic, sycl::access::target::local>
        local_histogram(sycl::range(NUM_BINS), h);
    h.parallel_for(
        sycl::nd_range(sycl::range{N / blockSize}, sycl::range{64}),
        [=](sycl::nd_item<1> it) {
            int group = it.get_group()[0];
            int gSize = it.get_local_range()[0];
            sycl::ext::oneapi::sub_group sg = it.get_sub_group();
            int sgSize = sg.get_local_range()[0];
            int sgGroup = sg.get_group_id()[0];

            int factor = NUM_BINS / gSize;
            int local_id = it.get_local_id()[0];
            if ((factor <= 1) && (local_id < NUM_BINS)) {
                local_histogram[local_id].store(0);
            } else {
                for (int k = 0; k < factor; k++) {
                    local_histogram[gSize * k + local_id].store(0);
                }
            }
            it.barrier(sycl::access::fence_space::local_space);
        }
    );

    for (int k = 0; k < blockSize; k++) {
        unsigned long x =
            sg.load(macc.get_pointer() + group * gSize * blockSize +
                    sgGroup * sgSize * blockSize + sgSize * k);
        local_histogram[x & 0x3FFU].fetch_add(1);
        local_histogram[(x >> 16) & 0x3FFU].fetch_add(1);
        local_histogram[(x >> 32) & 0x3FFU].fetch_add(1);
        local_histogram[(x >> 48) & 0x3FFU].fetch_add(1);
    }
    it.barrier(sycl::access::fence_space::local_space);

    if ((factor <= 1) && (local_id < NUM_BINS)) {
        hacc[local_id].fetch_add(local_histogram[local_id].load());
    } else {
        for (int k = 0; k < factor; k++) {
            hacc[gSize * k + local_id].fetch_add(1);
        }
    }
```

(continues on next page)
When the work-group is started, each work-item in the work-group initializes a portion of the histogram bins in SLM to 0 (code in lines 21-27 in the above example). You could designate one work-item to initialize all the histogram bins, but it is usually more efficient to divide the job among all work-items in the work-group.

The work-group barrier after initialization at line 28 guarantees that all histogram bins are initialized to 0 before any work-item updates any bins.

Because the histogram bins in SLM are shared among all work-items, updates to any bin by any work-item has to be atomic.

The global histograms are updated once the local histograms in the work-group is completed. But before reading the local SLM bins to update the global bins, a work-group barrier is again called at line 43 to make sure all work-items have completed their work.

When SLM data is shared, work-group barriers are often required for work-item synchronization. The barrier has a cost and the cost may increase with a larger work-group size. It is always a good idea to try different work-group sizes to find the best one for your application.

You can find an example of an SLM version of a histogram with 256 bins in the Examples folder. You can compare its performance with the performance of the version using registers. You may get some surprising results, and think about further optimizations that can be done.

### 7.4.4 Using SLM as Cache

You may sometimes find it more desirable to have the application manage caching of some hot data than to have the hardware do it automatically. With the application managing data caching directly, whenever the data is needed, you know exactly where the data is and the cost to access it. The SLM can be used for this purpose.

Consider the following 1-D convolution example:

#### Listing 35: /examples/slm/convolution-global.cpp

```cpp
sycl::buffer<int> ibuf(input.data(), N);
sycl::buffer<int> obuf(output.data(), N);
sycl::buffer<int> kbuf(kernel.data(), M);

auto e = q.submit([&](auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    sycl::accessor kacc(kbuf, h, sycl::read_only);

    h.parallel_for(sycl::nd_range<1>(sycl::range(N), sycl::range{256}),
                   [=](sycl::nd_item<1> it) {
                    int i = it.get_global_linear_id();
```
The example convolves an integer array of 8192 x 8192 elements using a kernel array of 257 elements and writes the result to an output array. Each work-item convolves one element. To convolve one element, however, up to 256 neighboring elements are needed.

Noticing each input element is used by multiple work-items, you can preload all input elements needed by a whole work-group into SLM. Later, when an element is needed, it can be loaded from SLM instead of global memory.

**Listing 36:** /examples/slm/convolution-slm-cache.cpp

```c++
sycl::buffer<int> ibuf(input.data(), N);
sycl::buffer<int> obuf(output.data(), N);
sycl::buffer<int> kbuf(kernel.data(), M);

auto e = q.submit([&](auto &h) {
    sycl::accessor iacc(ibuf, h, sycl::read_only);
    sycl::accessor oacc(obuf, h);
    int group = it.get_group()[0];
    int gSize = it.get_local_range()[0];
    int t = 0;
    int _M = static_cast<int>(M);
    int _N = static_cast<int>(N);

    if (((group == 0) || (group == _N / gSize - 1)) {
        if (i < _M / 2) {
            for (int j = _M / 2 - i, k = 0; j < _M; ++j, ++k) {
                t += iacc[k] * kacc[j];
            }
        } else {
            if (i + _M / 2 >= _N) {
                for (int j = 0, k = i - _M / 2;
                     j < _M / 2 + _N - i; ++j, ++k) {
                    t += iacc[k] * kacc[j];
                }
            } else {
                for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
                    t += iacc[k] * kacc[j];
                }
            }
        }
    } else {
        for (int j = 0, k = i - _M / 2; j < _M; ++j, ++k) {
            t += iacc[k] * kacc[j];
        }
    }

    oacc[i] = t;
});
```
When the work-group starts, all input elements needed by each work-item are loaded into SLM. Each work-item,
except the first one and the last one, loads one element into SLM. The first work-item loads neighbors on the left of the first element and the last work item loads neighbors on the right of the last element in the SLM. If no neighbors exist, elements in SLM are filled with 0s.

Before convolution starts in each work-item, a local barrier is called to make sure all input elements are loaded into SLM.

The convolution in each work-item is straightforward. All neighboring elements are loaded from the faster SLM instead of global memory.

### 7.5 Pointer Aliasing and the Restrict Directive

Kernels typically operate on arrays of elements that are provided as pointer arguments. When the compiler cannot determine whether these pointers alias each other, it will conservatively assume that they do, in which case it will not reorder operations on these pointers. Consider the following vector-add example, where each iteration of the loop has two loads and one store.

```cpp
size_t VectorAdd(sycl::queue &q, const IntArray &a, const IntArray &b, IntArray &sum, int iter) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);

    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        auto e = q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(num_items,
                [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
        });
    }
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "Vector add completed on device - took " << (end - start).count()
               << " u-secs\n";
    return ((end - start).count());
} // end VectorAdd
```

In this case, the programmer leaves all the choices about vector length and the number of work-groups to the compiler. In most cases the compiler does a pretty good job of selecting these parameters to get good performance. In some situations it may be better to explicitly choose the number of work-groups and work-group sizes to get good performance and provide hints to the compiler to get better-performing code.
The kernel below is written to process multiple elements of the array per work-item and explicitly chooses the number of work-groups and work-group size. The `intel::kernel_args_restrict` on line 25 tells the compiler that the buffer accessors in this kernel do not alias each other. This will allow the compiler to hoist the loads and stores, thereby providing more time for the instructions to complete and getting better instruction scheduling. The pragma on line 27 directs the compiler to unroll the loop by a factor of two.

**Listing 38: /examples/restrict/vec-add-restrict.cpp**

```cpp
size_t VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
                  IntArray &sum, int iter) {
    sycl::range num_items(a.size());
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = d_selector.get_info<sycl::info::device::max_compute_units>();
    size_t wg_size = d_selector.get_info<sycl::info::device::max_work_group_size>();
    size_t num_groups = 1;
    size_t wg_size = 8;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                           [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(8)]] [[intel::kernel_args_restrict]] {
                            size_t loc_id = index.get_local_id();
                            // unroll with a directive
                            #pragma unroll(2)
                            for (size_t i = loc_id; i < mysize; i += wg_size) {
                                sum_acc[i] = a_acc[i] + b_acc[i];
                            }
                        });
        });
        q.wait();
        auto end = std::chrono::steady_clock::now();
        std::cout << "Vector add2 completed on device - took "
                   << (end - start).count() << " u-secs\n";
        return ((end - start).count());
    }
}
```

The kernel below illustrates manually unrolling of the loop instead of the compiler directive (the compiler may or may not honor the directive depending on its internal heuristic cost model). The advantage of unrolling is that fewer instructions are executed because the loop does not have to iterate as many times, thereby saving on the compare and branch instructions.
The kernel below shows how to reorder the loads and stores so that all loads are issued before any operations on them are done. Typically, there can be many outstanding loads for every thread in the GPU. It is always better to issue the loads before any operations on them are done. This will allow the loads to complete before the data are actually needed for computation.

```
size_t VectorAdd4(sycl::queue &q, const IntArray &a, const IntArray &b, 
    IntArray &sum, int iter) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = 1;
    size_t wg_size = 8;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                [=](sycl::nd_item<1> index) {
                    // Manual unrolling
                    size_t loc_id = index.get_local_id();
                    for (size_t i = loc_id; i < mysize; i += 16) {
                        sum_acc[i] = a_acc[i] + b_acc[i];
                        sum_acc[i + 8] = a_acc[i + 8] + b_acc[i + 8];
                    }
                });
        });
    }
    q.wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "Vector add3 completed on device - took " << (end - start).count() << " u-secs\n";
    return ((end - start).count());}
```
The following kernel has a restrict directive, which provides a hint to the compiler that there is no aliasing among the vectors accessed inside the loop and the compiler can hoist the load over the store just like it was done manually in the previous example.

Listing 41: /examples/restrict/vec-add-restrict.cpp

```cpp
size_t VectorAdd5(sycl::queue &q, const IntArray &a, const IntArray &b, 
    IntArray &sum, int iter) {
    sycl::range num_items(a.size());

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    size_t num_groups = 1;
    size_t wg_size = 8;
    auto start = std::chrono::steady_clock::now();

    for (int i = 0; i < iter; i++) {
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size), 
                [=](sycl::nd_item<1> index) {
                    // Manual unrolling
                    size_t loc_id = index.get_local_id();
                    for (size_t i = loc_id; i < mysize; i += 16) {
                        int t1 = a_acc[i];
                        int t2 = b_acc[i];
                        int t3 = a_acc[i + 8];
                        int t4 = b_acc[i + 8];
                        sum_acc[i] = t1 + t2;
                        sum_acc[i + 8] = t3 + t4;
                    }
                });
        });
    }

    return ((end - start).count());
} // end VectorAdd5
```
for (int i = 0; i < iter; i++) {
    q.submit([&](auto &h) {
        // Input accessor
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                       [=](sycl::nd_item<1> index) [[intel::reqd_sub_group_size(8)]] [[intel::kernel_args_restrict]] {
            // compiler needs to hoist the loads
            size_t loc_id = index.get_local_id();
            for (size_t i = loc_id; i < mysize; i += 16) {
                sum_acc[i] = a_acc[i] + b_acc[i];
                sum_acc[i + 8] = a_acc[i + 8] + b_acc[i + 8];
            }
        });
    });
    q.wait();
}
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add5 completed on device - took "
           << (end - start).count() << " u-secs\n";
return ((end - start).count());
} // end VectorAdd5

7.6 Synchronization among Threads in a Kernel

There are a variety of ways in which the work-items in a kernel can synchronize to exchange data, update data, or cooperate with each other to accomplish a task in a specific order. These are:

**Accessor classes** Accessor classes specify acquisition and release of buffer and image data structures. Depending on where they are created and destroyed, the runtime generates appropriate data transfers and synchronization primitives.

**Atomic operations** DPC++ devices support a restricted subset of C++ atomics.

**Fences** Fence primitives are used to order loads and stores. Fences can have acquire semantics, release semantics, or both.

**Barriers** Barriers are used to synchronize sets of work-items within individual groups.

**Hierarchical parallel dispatch** In the hierarchical parallelism model of describing computations, synchronization within the work-group is made explicit through multiple instances of the parallel_for_work_item function call, rather than through the use of explicit work-group barrier operations.

**Device event** Events are used inside kernel functions to wait for asynchronous operations to complete.

In many cases, any of the preceding synchronization events can be used to achieve the same functionality, but with significant differences in efficiency and performance.
7.6.1 Atomic Operations

Atomics allow multiple work-items for any cross work-item communication via memory. DPC++ atomics are similar to C++ atomics and make the access to resources protected by atomics guaranteed to be executed as a single unit. The following factors affect the performance and legality of atomic operations:

- Data types
- Local vs global address space
- Host, shared and device allocated USM

Data types in atomic operations

The following kernel shows the implementation of a reduction operation in DPC++ where every work-item is updating a global accumulator atomically. The input data type of this addition and the vector on which this reduction operation is being applied is an integer. The performance of this kernel is reasonable compared to other techniques used for reduction, such as blocking.

Listing 42: /examples/atomics/atomics.cpp

```cpp
q.submit([&]<auto &h> {  
    sycl::accessor buf_acc(buf, h, sycl::read_only);  
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);  
    h.parallel_for(data_size, [=](auto index) {  
        size_t glob_id = index[0];  
        auto v = sycl::ext::oneapi::atomic_ref<
            int, sycl::ext::oneapi::memory_order::relaxed,
            sycl::ext::oneapi::memory_scope::device,
            sycl::access::address_space::global_space>(sum_acc[0]);  
        v.fetch_add(buf_acc[glob_id]);  
    });
});
```

If the data type of the vector is a float or a double as shown in the kernel below, the performance on certain accelerators is impaired due to lack of hardware support for float or double atomics. The following two kernels demonstrate how the time to execute an atomic add can vary drastically based on whether native atomics are supported.

Listing 43: /examples/atomics/test_atomic.cpp

```cpp
//
int VectorInt(sycl::queue &q, int iter) {
    VectorAllocator<int> alloc;
    AlignedVector<int> a(array_size, alloc);
    AlignedVector<int> b(array_size, alloc);

    InitializeArray<int>(a);
    InitializeArray<int>(b);
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
(continues on next page)
```
When using atomics, care must be taken to ensure that there is support in the hardware and that they can be executed efficiently. In Gen9 and Intel® Iris® Xe™ integrated graphics, there is no support for atomics on float or double data types and the performance of VectorDouble will be very poor. In future GPUs where the float and double atomics are supported in hardware, the performance of the above kernel will be much better.

Listing 44: /examples/atomics/test_atomic.cpp
auto v = sycl::ext::oneapi::atomic_ref<
    double, sycl::ext::oneapi::memory_order::relaxed,
    sycl::ext::oneapi::memory_scope::device,
    sycl::access::address_space::global_space>({a_acc[0]});
    
    v += b_acc[i];
    
};

q.wait();

auto end = std::chrono::steady_clock::now();
std::cout << "Vector Double completed on device - took "
        << (end - start).count() << " u-secs\n";
return ((end - start).count());

By analyzing these kernels using VTune Profiler, we can measure the impact of native atomic support. You can see that the VectorInt kernel is much faster than VectorDouble and VectorFloat.

VTune Profiler dynamic instruction analysis allows us to see the instruction counts vary dramatically when there is no support for native atomic.

Here is the assembly code for our VectorInt kernel.
Compared to the assembly code for VectorDouble, there are 33 million more GPU instructions required when we execute our VectorDouble kernel.

Fig. 18: VTune atomic int
The Intel Advisor tool has a recommendation pane that provides insights on how to improve the performance of GPU kernels.
One of the recommendations that Intel Advisor provides is “Inefficient atomics present”. When atomics are not natively supported in hardware, they are emulated. This can be detected and Intel Advisor gives advice on possible solutions.
Atomic operations on global and local address space

The standard C++ memory model assumes that applications execute on a single device with a single address space. Neither of these assumptions holds for DPC++ applications: different parts of the application execute on different devices (i.e., a host device and one or more accelerator devices); each device has multiple address spaces (i.e., private, local, and global); and the global address space of each device may or may not be disjoint (depending on USM support).

When using atomics in the global address space, again, care must be taken because global updates are much slower than local.

Listing 45: /examples/atomics/global_atomics_ref.cpp

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================================
#include <CL/sycl.hpp>
#include <iostream>
int main() {
    constexpr int N = 256 * 256;
    constexpr int M = 512;
    int total = 0;
    int *a = static_cast<int *>(malloc(sizeof(int) * N));
    for (int i = 0; i < N; i++)
        a[i] = 1;
    sycl::queue q({sycl::property::queue::enable_profiling()});
    sycl::buffer<int> buf(&total, 1);
    sycl::buffer<int> bufa(a, N);
    auto e = q.submit([&](sycl::handler &h) {
        sycl::accessor acc(buf, h);
        sycl::accessor acc_a(bufa, h, sycl::read_only);
        h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
            auto i = it.get_global_id();
            sycl::ext::oneapi::atomic_ref<int,
                sycl::ext::oneapi::memory_order_relaxed,
                sycl::ext::oneapi::memory_scope_device,
                sycl::access::address_space::global_space>
                atomic_op(acc[0]);
            atomic_op += acc_a[i];
        });
    });
    sycl::host_accessor h_a(buf);
    std::cout << "Reduction Sum : " << h_a[0] << "\n";
    std::cout << "Kernel Execution Time of Global Atomics Ref: "
        << e.get_profiling_info<sycl::info::event_profiling::command_end>() -
        e.get_profiling_info<sycl::info::event_profiling::command_start>()
        << "\n";
    return 0;
}

It is possible to refactor your code to use local memory space as the following example demonstrates.
Listing 46: /examples/atomics/local_atomics_ref.cpp

```cpp
// =============================================================================
// Copyright © 2022 Intel Corporation
// //
// // SPDX-License-Identifier: MIT
// // =============================================================================
#include <CL/sycl.hpp>
#include <iostream>
int main() {
    constexpr int N = 256 * 256;
    constexpr int M = 512;
    constexpr int NUM_WG = N / M;
    int total = 0;
    int *a = static_cast<int*>(malloc(sizeof(int) * N));
    for (int i = 0; i < N; i++)
        a[i] = 1;
    sycl::queue q({sycl::property::queue::enable_profiling()});
    sycl::buffer<int> global(&total, 1);
    sycl::buffer<int> bufa(a, N);
    auto e1 = q.submit([&](sycl::handler &h) {
        sycl::accessor b(global, h);
        sycl::accessor acc_a(bufa, h, sycl::read_only);
        auto acc = sycl::accessor<int, 1, sycl::access::mode::read_write,
                                  sycl::access::target::local>(NUM_WG, h);
        h.parallel_for(sycl::nd_range<1>(N, M), [=](auto it) {
            auto i = it.get_global_id(0);
            auto group_id = it.get_group(0);
            sycl::ext::oneapi::atomic_ref<int,
                            sycl::ext::oneapi::memory_order_relaxed,
                            sycl::ext::oneapi::memory_scope_device,
                            sycl::access::address_space::local_space>
                atomic_op(acc[group_id]);
            sycl::ext::oneapi::atomic_ref<int,
                            sycl::ext::oneapi::memory_order_relaxed,
                            sycl::ext::oneapi::memory_scope_device,
                            sycl::access::address_space::global_space>
                atomic_op_global(b[0]);
            atomic_op += acc_a[i];
            it.barrier(sycl::access::fence_space::local_space);
            if (it.get_local_id() == 0)
                atomic_op_global += acc[group_id];
        });
    });
    sycl::host_accessor h_global(global);
    std::cout << "Reduction Sum : " << h_global[0] << "\n";
    int total_time =
        (e1.get_profiling_info<sycl::info::event_profiling::command_end>() -
         e1.get_profiling_info<sycl::info::event_profiling::command_start>())
        / (e1.get_profiling_info<sycl::info::event_profiling::command_start>());
    std::cout << "Kernel Execution Time of Local Atomics : " << total_time
               << "\n";
    return 0;
}
```

(continues on next page)
Atomic operations on USM data

On discrete GPU,

- Atomic operations on host allocated USM (sycl::malloc_host) are not supported.
- Concurrent access from host and device to shared USM location (sycl::malloc_shared) is not supported.

We recommend using device allocated USM (sycl::malloc_device) memory for atomics and device algorithms with atomic operations.

7.6.2 Local Barriers vs Global Atomics

Atomics allow multiple work-items in the kernel to work on shared resources. Barriers allow synchronization among the work-items in a work-group. It is possible to achieve the functionality of global atomics through judicious use of kernel launches and local barriers. Depending on the architecture and the amount of data involved, one or the other can have better performance.

In the following example, we try to sum a relatively small number of elements in a vector. This task is can be achieved in different ways. The first kernel shown below does this using only one work-item which walks through all elements of the vector and sums them up.

Listing 47: examples/local-global-sync/atomics.cpp

```cpp
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(data_size, [=](auto index) {
        int glob_id = index[0];
        if (glob_id == 0) {
            int sum = 0;
            for (int i = 0; i < N; i++)
                sum += buf_acc[i];
            sum_acc[0] = sum;
        }
    });
});
```

In the kernel shown below, the same problem is solved using global atomics, where every work-item updates a global variable with the value it needs to accumulate. Although there is a lot of parallelism here, the contention on the global variable is quite high and in most cases its performance will not be very good.
In the following kernel, every work-item is responsible for accumulating multiple elements of the vector. This accumulation is done in parallel and then updated into an array that is shared among all work-items of the work-group. At this point all work-items of the work-group do a tree reduction using barriers to synchronize among themselves to reduce intermediate results in shared memory to the final result. This kernel explicitly created exactly one work-group and distributes the responsibility of all elements in the vector to the work-items in the work-group. Although it is not using the full capability of the machine in terms of the number of threads, sometimes this amount of parallelism is enough for small problem sizes.

The performance of these three kernels varies quite a bit among various platforms, and developers need to pick the technique that suits their application and hardware.
7.7 Considerations for Selecting Work-group Size

In DPC++ you can select the work-group size for `nd_range` kernels. The size of work-group has important implications for utilization of the compute resources, vector lanes, and communication among the work-items. The work-items in the same work-group may have access to hardware resources like shared memory and hardware synchronization capabilities that will allow them to run and communicate more efficiently than work-items across work-groups. So in general you should pick the maximum work-group size supported by the accelerator. The maximum work-group size can be queried by the call `device::get_info<cl::sycl::info::device::max_work_group_size>()`.

To illustrate the impact of the choice of work-group size, consider the following reduction kernel, which goes through a large vector to add all the elements in it. The function that runs the kernels takes in the work-group-size and sub-group-size as arguments, which lets you run experiments with different values. The performance difference can be seen from the timings reported when the kernel is called with different values for work-group size.

```
void reduction(sycl::queue &q, std::vector<int> &data, std::vector<int> &flush, int iter, int work_group_size) {
    const size_t data_size = data.size();
    const size_t flush_size = flush.size();
    int sum = 0;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    // int vec_size = q.get_device().get_info<sycl::info::device::native_vector_width_int>();
    int num_work_items = data_size / work_group_size;
    sycl::buffer<int> buf(data.data(), data_size, props);
    sycl::buffer<int> flush_buf(flush.data(), flush_size, props);
    sycl::buffer<int> sum_buf(&sum, 1, props);

    init_data(q, buf, data_size);

    double elapsed = 0;
    for (int i = 0; i < iter; i++) {
        q.submit([&](auto h) {
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(1, [=](auto index) { sum_acc[index] = 0; });
        });
        // flush the cache
        q.submit([&](auto h) {
            sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
        });

        Timer timer;
        // reductionMapToHWVector main begin
        q.submit([&](auto h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
        });
        // reductionMapToHWVector main end

        elapsed = timer.elapsed();
        if (elapsed > 0) {
            // do something with the elapsed time
        }
    }
}
```

(continues on next page)
In the code below, the above kernel is called with two different values: \(2^{\text{vec-size}}\) and the maximum possible work-group size supported by the accelerator. The performance of the kernel when work-group size is equal to \(2^{\text{vec-size}}\) will be lower than when the work-group size is the maximum possible value.

Listing 51: /examples/work-group-size/reduction-wg-size.cpp

```cpp
int vec_size = 16;
int work_group_size = vec_size;
reduction(q, data, extra, 16, work_group_size);
work_group_size =
    q.get_device().get_info<sycl::info::device::max_work_group_size>();
reduction(q, data, extra, 16, work_group_size);
```
In situations where there are no barriers nor atomics used, the work-group size will not impact the performance. To illustrate this, consider the following **vec_copy** kernel where there are no atomics or barriers.

**Listing 52: /examples/work-group-size/vec-copy.cpp**

```cpp
void vec_copy(sycl::queue &q, std::vector<int> &src, std::vector<int> &dst, 
              std::vector<int> &flush, int iter, int work_group_size) {
    const size_t data_size = src.size();
    const size_t flush_size = flush.size();

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    int num_work_items = data_size;
    double elapsed = 0;
    {
        sycl::buffer<int> src_buf(src.data(), data_size, props);
        sycl::buffer<int> dst_buf(dst.data(), data_size, props);
        sycl::buffer<int> flush_buf(flush.data(), flush_size, props);

        for (int i = 0; i < iter; i++) {
            // flush the cache
            q.submit([&](auto &h) {
                sycl::accessor flush_acc(flush_buf, h, sycl::write_only, sycl::no_init);
                h.parallel_for(flush_size, [=](auto index) { flush_acc[index] = 1; });
            });

            Timer timer;
            q.submit([&](auto &h) {
                sycl::accessor src_acc(src_buf, h, sycl::read_only);
                sycl::accessor dst_acc(dst_buf, h, sycl::write_only, sycl::no_init);

                h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
                               [=](sycl::nd_item<1> item)
                               {[[intel::reqd_sub_group_size(16)]] {
                                    int glob_id = item.get_global_id();
                                    dst_acc[glob_id] = src_acc[glob_id];
                                });
            });
            q.wait();
            elapsed += timer.Elapsed();
        }
        elapsed = elapsed / iter;
        std::string msg = "with work-group-size=" + std::to_string(work_group_size);
        check_result(elapsed, msg, dst);
    } // vec_copy end
```

In the code below, the above kernel is called with different work-group sizes. All the above calls to the kernel will have similar run times which indicates that there is no impact of work-group size on performance. The reason for this is that the threads created within a work-group and threads from different work-groups behave in a similar manner from the scheduling and resourcing point of view when there are no barriers nor shared memory in the work-groups.
In some accelerators, a minimum sub-group size is needed to obtain good performance due to the way in which threads are scheduled among the processing elements. In such a situation you may see a big performance difference when the number of sub-groups is less than the minimum. The call to the kernel on line 3 above has only one sub-group, while the call on line 5 has two sub-groups. There will be a significant performance difference in the timings for these two kernel invocations on an accelerator that performs scheduling of two sub-groups at a time.

### 7.7.1 Tuning Kernels with Local and Global Work-group Sizes in OpenMP Offload Mode

The approach of tuning kernel performance on accelerator devices as explained above for DPC++, is also applicable for implementations via OpenMP in offload mode. It is possible to customize an application kernel along with the use of OpenMP directives to make use of appropriate work-group sizes. However, this may require significant modifications to the code. The OpenMP implementation provides an option to custom tune kernels with the use of environment variables. The local and global work-group sizes for kernels in an app can be customized with the use of two environment variables – `OMP_THREAD_LIMIT` and `OMP_NUM_TEAMS` help in setting up the local work-group size (LWS) and global work-group size (GWS) as shown below:

```
LWS = OMP_THREAD_LIMIT
GWS = OMP_THREAD_LIMIT * OMP_NUM_TEAMS
```

With the help of following reduction kernel example, we show the use of LWS and GWS in tuning kernel performance on accelerator device.

#### Listing 54: /examples/OpenMP/23_omp_work_group/test_omp_work_group.cpp

```cpp
int N = 2048;

double* A = make_array(N, 0.8);
double* B = make_array(N, 0.65);
double* C = make_array(N*N, 2.5);

int i, j;
double val = 0.0;
```
```c
#pragma omp target map(to:A[0:N],B[0:N],C[0:N*N]) map(tofrom:val)
{
    #pragma omp teams distribute parallel for collapse(2) reduction(+ : val)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            val += C[i * N + j] * A[i] * B[j];
        }
    }

    printf("Reduced val[%f10.3]", val);

    free(A);
    free(B);
    free(C);
}
```

E.g. by choosing `OMP_THREAD_LIMIT = 1024` and `OMP_NUM_TEAMS = 120`, the LWS and GWS parameters are set to 1024 and 122880, respectively.

The figure above shows that the best performance for this kernel comes with \(\text{LWS} = 1024\) and \(\text{GWS} = 30720\) which corresponds to `OMP_THREAD_LIMIT = 1024` and `OMP_NUM_TEAMS = 30`. These environment variables will set the LWS and GWS values to a fixed numbers for all kernels offloaded via OpenMP. However, these environment variables will not affect the LWS and GWS used by highly tuned library kernels like OneMKL.
7.8 Reduction

Reduction is a common operation in parallel programming where an operator is applied to all elements of an array and a single result is produced. The reduction operator is associative and in some cases commutative. Some examples of reductions are summation, maximum, and minimum. A serial summation reduction is shown below:

**Listing 55: /examples/reduction/reduction.cpp**

```cpp
for (int it = 0; it < iter; it++) {
    sum = 0;
    for (size_t i = 0; i < data_size; ++i) {
        sum += data[i];
    }
}
```

The time complexity of reduction is linear with the number of elements. There are several ways this can be parallelized, and care must be taken to ensure that the amount of communication/synchronization is minimized between different processing elements. A naive way to parallelize this reduction is to use a global variable and let the threads update this variable using an atomic operation:

**Listing 56: /examples/reduction/reduction.cpp**

```cpp
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(data_size, [=](auto index) {
        size_t glob_id = index[0];
        auto v = sycl::atomic_ref<int, sycl::memory_order::relaxed,
                                sycl::memory_scope::device,
                                sycl::access::address_space::global_space>(
            sum_acc[0]);
        v.fetch_add(buf_acc[glob_id]);
    });
});
```

This kernel will perform poorly because the threads are atomically updating a single memory location and getting significant contention. A better approach is to split the array into small chunks, let each thread compute a local sum for each chunk, and then do a sequential/tree reduction of the local sums. The number of chunks will depend on the number of processing elements present in the platform. This can be queried using the get_info<info::device::max_compute_units>() function on the device object:

**Listing 57: /examples/reduction/reduction.cpp**

```cpp
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(num_processing_elements, [=](auto index) {
        size_t glob_id = index[0];
        size_t start = glob_id * BATCH;
        size_t end = (glob_id + 1) * BATCH;
```
This kernel will perform better than the kernel that atomically updates a shared memory location. However, it is still inefficient because the compiler is not able to vectorize the loop. One way to get the compiler to produce vector code is to modify the loop as shown below:

**Listing 58: /examples/reduction/reduction.cpp**

```cpp
#include <sycl/sycl.hpp>

int main() {

  std::vector<int> data(1000000);

  if (end > N)
    end = N;
  int sum = 0;
  for (size_t i = start; i < end; ++i)
    sum += buf_acc[i];
  accum_acc[glob_id] = sum;

  return 0;
}
```

The compiler can vectorize this code so the performance is better.

In the case of GPUs, a number of thread contexts are available per physical processor, referred to as Vector Engine (VE) or Execution Unit (EU) on the machine. So the above code where the number of threads is equal to the number of VEs does not utilize all the thread contexts. Even in the case of CPUs that have two hyperthreads per core, the code will not use all the thread contexts. In general, it is better to divide the work into enough workgroups to get full occupancy of all thread contexts. This allows the code to better tolerate long latency instructions. The following table shows the number of thread contexts available per processing element in different devices:

**Table 10: Number of thread contexts available by device**

<table>
<thead>
<tr>
<th>Device</th>
<th>VEs</th>
<th>Threads per VE</th>
<th>Total threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBL</td>
<td>24</td>
<td>7</td>
<td>24 × 7 = 168</td>
</tr>
<tr>
<td>TGL</td>
<td>96</td>
<td>7</td>
<td>96 × 7 = 672</td>
</tr>
</tbody>
</table>

The code below shows a kernel with enough threads to fully utilize available resources. Notice that there is no good way to query the number of available thread contexts from the device. So, depending on the device, you can scale the number of work-items you create for splitting the work among them.
One popular way of doing a reduction operation on GPUs is to create a number of work-groups and do a tree reduction in each work-group. In the kernel shown below, each work-item in the work-group participates in a reduction network to eventually sum up all the elements in that work-group. All the intermediate results from the work-groups are then summed up by doing a serial reduction (if this intermediate set of results is large enough then we can do fewer more round(s) of tree reductions). The tree reduction algorithm takes advantage of the very fast synchronization operations among the work-items in a work-group. The performance of this kernel is highly dependent on the efficiency of the kernel launches, because a large number of kernels are launched. Also, the kernel as written below is not very efficient because the number of threads doing actual work reduces exponentially each time through the loop.

Listing 59: /examples/reduction/reduction.cpp

```cpp
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(num_work_items, [=](auto index) {
        size_t glob_id = index[0];
        int sum = 0;
        for (size_t i = glob_id; i < data_size; i += num_work_items)
            sum += buf_acc[i];
        accum_acc[glob_id] = sum;
    });
});
```

Listing 60: /examples/reduction/reduction.cpp

```cpp
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    sycl::accessor<int, 1, sycl::access::mode::read_write,
                   sycl::access::target::local>
                   scratch(work_group_size, h);
    h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
                   [=](sycl::nd_item<1> item) {
        size_t global_id = item.get_global_id(0);
        int local_id = item.get_local_id(0);
        int group_id = item.get_group(0);

        if (global_id < data_size)
            scratch[local_id] = buf_acc[global_id];
        else
            scratch[local_id] = 0;

        // Do a tree reduction on items in work-group
        for (int i = work_group_size / 2; i > 0; i >>= 1) {
            item.barrier(sycl::access::fence_space::local_space);
            if (local_id < i)
                scratch[local_id] += scratch[local_id + i];
        }
    });
```
The single stage reduction is not very efficient since it will leave a lot work for the host. Adding one more stage will reduce the work on the host and improve performance quite a bit. It can be seen that in the kernel below the intermediate result computed in stage1 is used as input into stage2. This can be generalized to form a multi-stage reduction until the result is small enough so that it can be performed on the host.

**Listing 61: examples/reduction/reduction.cpp**

```cpp
q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum1_buf, h, sycl::write_only, sycl::no_init);
    sycl::accessor<int, 1, sycl::access::mode::read_write,
                   sycl::access::target::local>
                   scratch(work_group_size, h);

    h.parallel_for(sycl::nd_range<1>(num_work_items1, work_group_size),
                   [=](sycl::nd_item<1> item) {
    size_t global_id = item.get_global_id(0);
    int local_id = item.get_local_id(0);
    int group_id = item.get_group(0);

    if (global_id < data_size)
        scratch[local_id] = buf_acc[global_id];
    else
        scratch[local_id] = 0;

    // Do a tree reduction on items in work-group
    for (int i = work_group_size / 2; i > 0; i >>= 1) {
    item.barrier(sycl::access::fence_space::local_space);
    if (local_id < i)
        scratch[local_id] += scratch[local_id + i];
    }

    if (local_id == 0)
        accum_acc[group_id] = scratch[0];
    });

q.submit([&](auto &h) {
    sycl::accessor buf_acc(accum1_buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum2_buf, h, sycl::write_only, sycl::no_init);
    sycl::accessor<int, 1, sycl::access::mode::read_write,
                   sycl::access::target::local>
                   scratch(work_group_size, h);

    h.parallel_for(sycl::nd_range<1>(num_work_items2, work_group_size),
                   [=](sycl::nd_item<1> item) {
    size_t global_id = item.get_global_id(0);
```

(continues on next page)
int local_id = item.get_local_id(0);
int group_id = item.get_group(0);

if (global_id < static_cast<size_t>(num_work_items2))
    scratch[local_id] = buf_acc[global_id];
else
    scratch[local_id] = 0;

// Do a tree reduction on items in work-group
for (int i = work_group_size / 2; i > 0; i >>= 1) {
    item.barrier(sycl::access::fence_space::local_space);
    if (local_id < i)
        scratch[local_id] += scratch[local_id + i];
}
if (local_id == 0)
    accum_acc[group_id] = scratch[0];
}

DPC++ also supports built-in reduction operations, and you should use it where it is suitable because its implementation is fine tuned to the underlying architecture. The following kernel shows how to use the built-in reduction operator in the compiler.

Listing 62: /examples/reduction/reduction.cpp

q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor sum_acc(sum_buf, h, sycl::read_write);
    auto sumr =
        sycl::ext::oneapi::reduction(sum_acc, sycl::ext::oneapi::plus<>());
    h.parallel_for(sycl::nd_range<1>{data_size, 256}, sumr,
        [=](sycl::nd_item<1> item, auto &sumr_arg) {
            int glob_id = item.get_global_id(0);
            sumr_arg += buf_acc[glob_id];
        });
});

A further optimization is to block the accesses to the input vector and use the shared local memory to store the intermediate results. This kernel is shown below. In this kernel every work-item operates on a certain number of vector elements, and then one thread in the work-group reduces all these elements to one result by linearly going through the shared memory containing the intermediate results.

Listing 63: /examples/reduction/reduction.cpp

q.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::read_only);
    sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
    sycl::accessor<int, 1, sycl::access::mode::read_write,
        sycl::access::target::local>
    scratch(work_group_size, h);
});
The kernel below is similar to the one above except that tree reduction is used to reduce the intermediate results from all the work-items in a work-group. In most cases this does not seem to make a big difference in performance.

**Listing 64: /examples/reduction/reduction.cpp**
The kernel below uses the blocking technique and then the compiler reduction operator to do final reduction. This gives good performance on most of the platforms on which it was tested.

**Listing 65: /examples/reduction/reduction.cpp**

```cpp
q.submit([&](auto &h) {
  sycl::accessor buf_acc(buf, h, sycl::read_only);
  sycl::accessor sum_acc(sum_buf, h, sycl::read_write, sycl::no_init);
  auto sumr =
    sycl::ext::oneapi::reduction(sum_acc, sycl::ext::oneapi::plus<>());
  h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size}, sumr,
    [=](sycl::nd_item<1> item, auto &sumr_arg) {
      size_t glob_id = item.get_global_id(0);
      int offset = (glob_id >> log2workitems_per_block)
        << log2elements_per_block +
        (glob_id & mask);
      int sum = 0;
      for (int i = 0; i < elements_per_work_item; ++i)
        sum +=
          buf_acc[(i << log2workitems_per_block) + offset];
      sumr_arg += sum;
    });
});
```

This next kernel uses a completely different technique for accessing the memory. It uses sub-group loads to generate the intermediate result in a vector form. This intermediate result is then brought back to the host and the final reduction is performed there. In some cases it may be better to create another kernel to reduce this result in a single work-group, which lets you perform tree reduction through efficient barriers.

**Listing 66: /examples/reduction/reduction.cpp**

```cpp
q.submit([&](auto &h) {
  const sycl::accessor buf_acc(buf, h);
  sycl::accessor accm_acc(accm_buf, h, sycl::write_only, sycl::no_init);
  sycl::accessor<
sycl::vec<int, 8>, 1, sycl::access::mode::read_write,
  sycl::access::target::local>
    scratch(work_group_size, h);
  h.parallel_for(
    sycl::nd_range<1>{num_work_items, work_group_size},
    [=](sycl::nd_item<1> item) [[intel::reqd_sub_group_size(16)]] {
      size_t group_id = item.get_group(0);
      ...
    });
});
```
11 size_t loc_id = item.get_local_id(0);
12 sycl::ext::oneapi::sub_group sg = item.get_sub_group();
13 sycl::vec<int, 8> sum{0, 0, 0, 0, 0, 0, 0, 0};
14 using global_ptr =
15 sycl::multi_ptr<int, sycl::access::address_space::global_space>;
16 int base = (group_id * work_group_size +
17 sg.get_group_id()[0] * sg.get_local_range()[0]) *
18 elements_per_work_item;
19 for (int i = 0; i < elements_per_work_item / 8; ++i)
20 sum += sg.load<8>(global_ptr(&buf_acc[base + i * 128]));
21 scratch[loc_id] = sum;
22 for (int i = work_group_size / 2; i > 0; i >>= 1) {
23 item.barrier(sycl::access::fence_space::local_space);
24 if (loc_id < static_cast<size_t>(i))
25 scratch[loc_id] += scratch[loc_id + i];
26 }
27 if (loc_id == 0)
28 accum_acc[group_id] = scratch[0];
29 };

Different implementations of reduction operation are provided and discussed here, which may have different performance characteristics depending on the architecture of the accelerator. Another important thing to note is that the time it takes to bring the result of reduction to the host over the PCIe interface (for a discrete GPU) is almost same as actually doing the entire reduction on the device. This shows that one should avoid data transfers between host and device as much as possible or overlap the kernel execution with data transfers.

7.9 Kernel Launch

In DPC++, work is performed by enqueueing kernels into queues targeting specific devices. These kernels are submitted by the host to the device, executed by the device and results are sent back. The kernel submission by the host and the actual start of execution do not happen immediately - they are asynchronous and as such we have to keep track of the following timings associated with a kernel.

**Kernel submission start time**  This is the at which the host starts the process of submitting the kernel.

**Kernel submission end time**  This is the time at which the host finished submitting the kernel. The host performs multiple tasks like queuing the arguments, allocating resources in the runtime for the kernel to start execution on the device.

**Kernel launch time**  This is the time at which the kernel that was submitted by the host starts executing on the device. Note that this is not exactly same as the kernel submission end time. There is a lag between the submission end time and the kernel launch time, which depends on the availability of the device. It is possible for the host to queue up a number of kernels for execution before the kernels are actually launched for execution. More over, there are a few data transfers that need to happen before the actual kernel starts execution which is typically not accounted separately from kernel launch time.

**Kernel completion time**  This is the time at which the kernel finishes execution on the device. The current generation of devices are non-preemptive, which means that once a kernel starts, it has to complete its execution.
Tools like VTune™ Profiler (vtune), clIntercept, and zelIntercept provide a visual timeline for each of the above times for every kernel in the application.

The following simple example shows time being measured for the kernel execution. This will involve the kernel submission time on the host, the kernel execution time on the device, and any data transfer times (since there are no buffers or memory, this is usually zero in this case).

Listing 67: /examples/kernels/launch.cpp

```cpp
#include <CL/sycl.hpp>

class Timer {
    public:
        Timer();
        ~Timer();
        void start();
        void stop();
        double elapsed();
    private:
        double _start_time;
        double _stop_time;
};

void emptyKernel1(sycl::queue &q) {
    Timer timer;
    for (int i = 0; i < iters; ++i)
        q.parallel_for(1, [=](auto) {
            /* NOP */
        }).wait();
    std::cout << "emptyKernel1: Elapsed time: " << timer.elapsed() / iters
              << " sec\n";
} // end emptyKernel1
```

The same code without the wait at the end of the parallel_for measures the time it takes for the host to submit the kernel to the runtime.

Listing 68: /examples/kernels/launch.cpp

```cpp
void emptyKernel2(sycl::queue &q) {
    Timer timer;
    for (int i = 0; i < iters; ++i)
        q.parallel_for(1, [=](auto) {
            /* NOP */
        });
    std::cout << "emptyKernel2: Elapsed time: " << timer.elapsed() / iters
              << " sec\n";
```}

These overheads are highly dependent on the backend runtime being used and the processing power of the host.

One way to measure the actual kernel execution time on the device is to use the DPC++ built-in profiling API. The following code demonstrates usage of the DPC++ profiling API to profile kernel execution times. It also shows the kernel submission time. There is no way to programmatically measure the kernel launch time since it is dependent on the runtime and the device driver. Profiling tools can provide this information.

Listing 69: /examples/kernels/profiling-api.cpp

```cpp
#pragma once

#include <CL/sycl.hpp>

class Timer {

public:
```
(continues on next page)
Timer() : start_(std::chrono::steady_clock::now()) {}  

double Elapsed() {
    auto now = std::chrono::steady_clock::now();
    return std::chrono::duration_cast<Duration>(now - start_).count();
}

private:
    using Duration = std::chrono::duration<double>;
    std::chrono::steady_clock::time_point start_;  
};

int main() {
    Timer timer;
    sycl::queue q{sycl::property::queue::enable_profiling());
    auto evt = q.parallel_for(1000, [=](auto) {
        /* kernel statements here */
    });
    double t1 = timer.Elapsed();
    evt.wait();
    double t2 = timer.Elapsed();
    auto startK =
        evt.get_profiling_info<sycl::info::event_profiling::command_start>();
    auto endK =
        evt.get_profiling_info<sycl::info::event_profiling::command_end>();
    std::cout << "Kernel submission time: " << t1 << "secs\n";
    std::cout << "Kernel submission + execution time: " << t2 << "secs\n";
    std::cout << "Kernel execution time: "
        << ((double)(endK - startK)) / 1000000.0 << "secs\n";
    return 0;
}

The following picture shows the timeline of the execution for the above example. This picture is generated from running `clIntercept` to generate a trace file and using Chrome’s tracing to visualize the timeline. In this timeline there are two swim lanes, one for the host side and another for the device side. Notice that the only activity on the device side is the execution of the submitted kernel. A significant amount of work is done on the host side to get the kernel prepared for execution. In this case, since the kernel is very small, total execution time is dominated by the JIT compilation of the kernel, which is the block labeled `clBuildProgram` in the figure below.
The following picture is the zoomed in version to show the detail of the functions called on the host side to submit the kernel. Here the time is dominated by the `clEnqueueNDRangeKernel`. Also notice that there is a lag between the completion of kernel submission on the host and the actual launch of the kernel on the device.

**Fig. 23:** Functions called on host to submit the kernel

---

## 7.10 Executing Multiple Kernels on the Device at the Same Time

DPC++ has two kinds of queues that a programmer can create and use to submit kernels for execution.

- **in-order queues** where kernels are executed in the order they were submitted to the queue
- **out-of-order queues** where kernels can be executed in an arbitrary order (subject to the dependency constraints among them).

The choice to create an in-order or out-of-order queue is made at queue construction time through the property `sycl::property::queue::in_order()`. By default, when no property is specified, the queue is out-of-order.

In the following example, three kernels are submitted per iteration. Each of these kernels uses only one workgroup with 256 work-items. These kernels are created specifically with one group to ensure that they do not use the entire machine. This is done to illustrate the benefit of parallel kernel execution.
Listing 70: /examples/multiple-kernel-execution/kernels.cpp

```cpp
int multi_queue(sycl::queue &q, const IntArray &a, const IntArray &b) {
    IntArray s1, s2, s3;
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf1(s1);
    sycl::buffer sum_buf2(s2);
    sycl::buffer sum_buf3(s3);

    size_t num_groups = 1;
    size_t wg_size = 256;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q.submit([&](sycl::handler &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            sycl::accessor sum_acc(sum_buf1, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                [=](sycl::nd_item<1> index) {
                    size_t loc_id = index.get_local_id();
                    sum_acc[loc_id] = 0;
                    for (int j = 0; j < 1000; j++)
                        for (size_t i = loc_id; i < array_size; i += wg_size) {
                            sum_acc[loc_id] += a_acc[i] + b_acc[i];
                        }
                });
        });
        q.submit([&](sycl::handler &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            sycl::accessor sum_acc(sum_buf2, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                [=](sycl::nd_item<1> index) {
                    size_t loc_id = index.get_local_id();
                    sum_acc[loc_id] = 0;
                    for (int j = 0; j < 1000; j++)
                        for (size_t i = loc_id; i < array_size; i += wg_size) {
                            sum_acc[loc_id] += a_acc[i] + b_acc[i];
                        }
                });
        });
        q.submit([&](sycl::handler &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            sycl::accessor sum_acc(sum_buf3, h, sycl::write_only, sycl::no_init);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                [=](sycl::nd_item<1> index) {
```
size_t loc_id = index.get_local_id();
sum_acc[loc_id] = 0;
for (int j = 0; j < 1000; j++)
    for (size_t i = loc_id; i < array_size; i += wg_size) {
        sum_acc[loc_id] += a_acc[i] + b_acc[i];
    }
}
q.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "multi_queue completed on device - took "
       << (end - start).count() << " u-secs"
       ;
// check results
return ((end - start).count());
} // end multi_queue

In the case where the underlying queue is in-order, these kernels cannot be executed in parallel and have to be executed sequentially even though there are adequate resources in the machine and there are no dependencies among the kernels. This can be seen from the larger total execution time for all the kernels. The creation of the queue and the kernel submission is shown below.

Listing 71: /examples/multiple-kernel-execution/kernels.cpp

sycl::property_list q_prop{sycl::property::queue::in_order()};
std::cout << "In order queue: Jitting+Execution time\n";
sycl::queue q1(d_selector, q_prop);
multi_queue(q1, a, b);
usleep(500 * 1000);
std::cout << "In order queue: Execution time\n";
multi_queue(q1, a, b);

When the queue is out-of-order, the overall execution time is much lower, indicating that the machine is able to execute different kernels from the queue at the same time. The creation of the queue and the invocation of the kernel is shown below.

Listing 72: /examples/multiple-kernel-execution/kernels.cpp

sycl::queue q2(d_selector);
std::cout << "Out of order queue: Jitting+Execution time\n";
multi_queue(q2, a, b);
usleep(500 * 1000);
std::cout << "Out of order queue: Execution time\n";
multi_queue(q2, a, b);

In situations where kernels do not scale strongly and therefore cannot effectively utilize full machine compute resources, it is better to allocate only the required compute units through appropriate selection of workgroup/work-item values and try to execute multiple kernels at the same time.

The following timeline view shows the kernels being executed by in-order and out-of-order queues (this was collected using the onetrace tool available at https://github.com/intel/pti-gpu/tree/master/tools/onetrace). Here
one can clearly see that kernels submitted to the out-of-order queue are being executed in parallel. Another thing to notice is that not all three kernels are executed in parallel all the time. How many kernels are executed in parallel is affected by multiple factors such as the availability of hardware resources, the time gap between kernel submissions, etc.

![Fig. 24: Timeline for kernels executed with in-order and out-of-order queues](image)

It is also possible to statically partition a single device into sub-devices through the use of `create_sub_devices` function of `device_class`. This provides more control to the programmer for submitting kernels to an appropriate sub-device. However, the partition of a device into sub-devices is static, so the runtime will not be able to adapt to the dynamic load of an application because it does not have flexibility to move kernels from one sub-device to another.

### 7.11 Submitting Kernels to Multiple Queues

Queues provide a channel to submit kernels for execution on an accelerator. Queues also hold a context that describes the state of the device. This state includes the contents of buffers and any memory needed to execute the kernels. The runtime keeps track of the current device context and avoids unnecessary memory transfers between host and device. Therefore, it is better to submit and launch kernels from one context together, as opposed to interleaving the kernel submissions in different contexts.

The following example submits 30 independent kernels that use the same buffers as input to compute the result into different output buffers. All these kernels are completely independent and can potentially execute concurrently and out of order. The kernels are submitted to three queues, and the execution of each kernel will incur different costs depending on the how the queues are created.

**Listing 73: /examples/multiple-queue-submission/multi-queue-light-kernel.cpp**

```c++
int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3, 
              const IntArray &a, const IntArray &b) {

    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer<int> *sum_buf[3 * iter];

    for (size_t i = 0; i < (3 * iter); i++)
        sum_buf[i] = new sycl::buffer<int>(256);

    size_t num_groups = 1;
    size_t wg_size = 256;
```
auto start = std::chrono::steady_clock::now();
for (int i = 0; i < iter; i++) {
    q1.submit([&](auto &h) {
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) {
                size_t loc_id = index.get_local_id();
                sum_acc[loc_id] = 0;
                for (size_t i = loc_id; i < array_size; i += wg_size) {
                    sum_acc[loc_id] += a_acc[i] + b_acc[i];
                } // end::for
            } // end::parallel_for
        );
    });
    q2.submit([&](auto &h) {
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        auto sum_acc = sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) {
                size_t loc_id = index.get_local_id();
                sum_acc[loc_id] = 0;
                for (size_t i = loc_id; i < array_size; i += wg_size) {
                    sum_acc[loc_id] += a_acc[i] + b_acc[i];
                } // end::for
            } // end::parallel_for
        );
    });
    q3.submit([&](auto &h) {
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        auto sum_acc = sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);

        h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
            [=](sycl::nd_item<1> index) {
                size_t loc_id = index.get_local_id();
                sum_acc[loc_id] = 0;
                for (size_t i = loc_id; i < array_size; i += wg_size) {
                    sum_acc[loc_id] += a_acc[i] + b_acc[i];
                } // end::for
            } // end::parallel_for
        );
    });
}
q1.wait();
q2.wait();
q3.wait();
auto end = std::chrono::steady_clock::now();
std::cout << "Vector add completed on device - took " << (end - start).count() << " u-secs\n";
// check results
for (size_t i = 0; i < (3 * iter); i++)
    delete sum_buf[i];
return ((end - start).count());
} // end VectorAdd

Submitting the kernels to the same queue gives the best performance because all the kernels are able to just transfer the needed inputs once at the beginning and do all their computations.

**Listing 74**: /examples/multiple-queue-submission/multi-queue-light-kernel.cpp

```cpp
VectorAdd(q, q, q, a, b);
```

If the kernels are submitted to different queues that share the same context, the performance is similar to submitting it to one queue. The issue to note here is that when a kernel is submitted to a new queue with a different context, the JIT process compiles the kernel to the new device associated with the context. If this JIT compilation time is discounted, the actual execution of the kernels is similar.

**Listing 75**: /examples/multiple-queue-submission/multi-queue-light-kernel.cpp

```cpp
sycl::queue q1(d_selector);
sycl::queue q2(q1.get_context(), d_selector);
sycl::queue q3(q1.get_context(), d_selector);
VectorAdd(q1, q2, q3, a, b);
```

If the kernels are submitted to three different queues that have three different contexts, performance degrades because at kernel invocation, the runtime needs to transfer all input buffers to the accelerator every time. In addition, the kernels will be JITed for each of the contexts.

**Listing 76**: /examples/multiple-queue-submission/multi-queue-light-kernel.cpp

```cpp
sycl::queue q4(d_selector);
sycl::queue q5(d_selector);
sycl::queue q6(d_selector);
VectorAdd(q4, q5, q6, a, b);
```

If for some reason you need to use different queues, the problem can be alleviated by creating the queues with shared context. This will prevent the need to transfer the input buffers, but the memory footprint of the kernels will increase because all the output buffers have to be resident at the same time in the context, whereas earlier the same memory on the device could be used for the output buffers. Another thing to remember is the issue of memory-to-compute ratio in the kernels. In the example above, the compute requirement of the kernel is low so the overall execution is dominated by the memory transfers. When the compute is high, these transfers do not contribute much to the overall execution time.

This is illustrated in the example below, where the amount of computation in the kernel is increased a thousand-fold and so the runtime will be different.
```cpp
int VectorAdd(sycl::queue &q1, sycl::queue &q2, sycl::queue &q3,
               const IntArray &a, const IntArray &b) {
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer<int> *sum_buf = new sycl::buffer<int>(3 * iter);
    for (size_t i = 0; i < (3 * iter); i++)
        sum_buf[i] = new sycl::buffer<int>(256);

    size_t num_groups = 1;
    size_t wg_size = 256;
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        q1.submit([&](auto &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            auto sum_acc = sum_buf[3 * i]->get_access<sycl::access::mode::write>(h);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                           [=](sycl::nd_item<1> index) {
                size_t loc_id = index.get_local_id();
                sum_acc[loc_id] = 0;
                for (int j = 0; j < 1000; j++)
                    for (size_t i = loc_id; i < array_size; i += wg_size) {
                        sum_acc[loc_id] += a_acc[i] + b_acc[i];
                    }
            });
        });
        q2.submit([&](auto &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            auto sum_acc = sum_buf[3 * i + 1]->get_access<sycl::access::mode::write>(h);

            h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
                           [=](sycl::nd_item<1> index) {
                size_t loc_id = index.get_local_id();
                sum_acc[loc_id] = 0;
                for (int j = 0; j < 1000; j++)
                    for (size_t i = loc_id; i < array_size; i += wg_size) {
                        sum_acc[loc_id] += a_acc[i] + b_acc[i];
                    }
            });
        });
        q3.submit([&](auto &h) {
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            auto sum_acc = sum_buf[3 * i + 2]->get_access<sycl::access::mode::write>(h);
        });
    }
}
```
50 h.parallel_for(sycl::nd_range<1>(num_groups * wg_size, wg_size),
51 [=](sycl::nd_item<1> index) {
52 size_t loc_id = index.get_local_id();
53 sum_acc[loc_id] = 0;
54 for (int j = 0; j < 1000; j++)
55 for (size_t i = loc_id; i < array_size; i += wg_size) {
56 sum_acc[loc_id] += a_acc[i] + b_acc[i];
57 }
58 });
59 });
60 q1.wait();
61 q2.wait();
62 q3.wait();
63 auto end = std::chrono::steady_clock::now();
64 std::cout << "Vector add completed on device - took " << (end - start).count()
65 << " u-secs\n";
66 // check results
67 for (size_t i = 0; i < (3 * iter); i++)
68 delete sum_buf[i];
69 return ((end - start).count());
70 } // end VectorAdd

7.12 Avoid Redundant Queue Construction

To execute kernels on a device, the user must create a queue, which references an associated context, platform, and device. These may be chosen automatically, or specified by the user.

A context is constructed, either directly by the user or implicitly when creating a queue, to hold all the runtime information required by the SYCL runtime and the SYCL backend to operate on a device. When a queue is created with no context specified, a new context is implicitly constructed using the default constructor. In general, creating a new context is a heavy duty operation due to the need for JIT compiling the program every time a kernel is submitted to a queue with a new context. For good performance one should use as few contexts as possible in their application.

In the following example, a queue is created inside the loop and the kernel is submitted to this new queue. This will essentially invoke the JIT compiler for every iteration of the loop.

Listing 78: /examples/redundant-queues/queues.cpp

```cpp
int reductionMultipleQMultipleC(std::vector<int> &data, int iter) {
const size_t data_size = data.size();
int sum = 0;
int work_group_size = 512;
int num_work_groups = 1;
int num_work_items = work_group_size;
const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
...}
```
sycl::buffer<int> buf(data.data(), data_size, props);
sycl::buffer<int> sum_buf(sum, 1, props);

sycl::queue q1{sycl::default_selector{}, exception_handler};
// initialize data on the device
q1.submit([&](auto &h) {
    sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
});

double elapsed = 0;
for (int i = 0; i < iter; i++) {
    sycl::queue q2{sycl::default_selector{}, exception_handler};
    if (i == 0)
        std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
    // reductionMultipleQMultipleC main begin
    Timer timer;
    q2.submit([&](auto &h) {
        sycl::accessor buf_acc(buf, h, sycl::read_only);
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
        sycl::accessor<int, 1, sycl::access::mode::read_write,
                        sycl::access::target::local>
            scratch(work_group_size, h);
        h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
            [=](sycl::nd_item<1> item) {
                size_t loc_id = item.get_local_id(0);
                int sum = 0;
                for (int i = loc_id; i < data_size; i += num_work_items)
                    sum += buf_acc[i];
                scratch[loc_id] = sum;
                for (int i = work_group_size / 2; i > 0; i >>= 1) {
                    item.barrier(sycl::access::fence_space::local_space);
                    if (loc_id < i)
                        scratch[loc_id] += scratch[loc_id + i];
                }
                if (loc_id == 0)
                    sum_acc[0] = scratch[0];
            });
    });
    // reductionMultipleQMultipleC main end
    q2.wait();
    sycl::host_accessor h_acc(sum_buf);
    sum = h_acc[0];
    elapsed += timer.Elapsed();
}
elapsed = elapsed / iter;
if (sum == sum_expected)
    std::cout << "SUCCESS: Time reductionMultipleQMultipleC = " << elapsed << "s"
        << " sum = " << sum << "\n";
else
    std::cout << "ERROR: reductionMultipleQMultipleC Expected " << sum_expected
    << " but got " << sum << "\n";
return sum;
} // end reductionMultipleQMultipleC

The above program can be rewritten by moving the queue declaration outside the loop, which improves performance quite dramatically.

Listing 79: /examples/redundant-queues/queues.cpp

```cpp
int reductionSingleQ(std::vector<int> &data, int iter) {
    const size_t data_size = data.size();
    int sum = 0;

    int work_group_size = 512;
    int num_work_groups = 1;
    int num_work_items = work_group_size;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

    sycl::buffer<int> buf(data.data(), data_size, props);
    sycl::buffer<int> sum_buf(&sum, 1, props);
    sycl::queue q{sycl::default_selector{}, exception_handler};
    std::cout << q.get_device().get_info<sycl::info::device::name>() << "\n";

    // initialize data on the device
    q.submit([&](auto &h) {
        sycl::accessor buf_acc(buf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(data_size, [=](auto index) { buf_acc[index] = 1; });
    });

double elapsed = 0;
for (int i = 0; i < iter; i++) {
    // reductionIntBarrier main begin
    Timer timer;
    q.submit([&](auto &h) {
        sycl::accessor buf_acc(buf, h, sycl::read_only);
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
        sycl::accessor<int, 1, sycl::access::mode::read_write, 
                      sycl::access::target::local>
                      scratch(work_group_size, h);
        h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
            [=](sycl::nd_item<1> item) {
                size_t loc_id = item.get_local_id(0);
                int sum = 0;
                for (int i = loc_id; i < data_size; i += num_work_items)
                    sum += buf_acc[i];
                scratch[loc_id] = sum;
                for (int i = work_group_size / 2; i > 0; i >>= 1) {
                    item.barrier(sycl::access::fence_space::local_space);
```
```
In case you need to create multiple queues, try to share the contexts among the queues. This will improve the performance. The above kernel is rewritten as shown below where the new queues created inside the loop and the queue outside the loop share the context. In this case the performance is same as the one with one queue.

```cpp
// examples/redundant-queues/queues.cpp

Listing 80: /examples/redundant-queues/queues.cpp
```
```cpp
sycl::queue q2{q1.get_context(), sycl::default_selector{},
    exception_handler};
if (i == 0)
    std::cout << q2.get_device().get_info<sycl::info::device::name>() << "\n";
// reductionMultipleQSingleC main begin
Timer timer;
q2.submit([&](auto &h) {
    sycl::_accessor buf_acc(buf, h, sycl::read_only);
    sycl::_accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    sycl::accessor<int, 1, sycl::access::mode::read_write,
        sycl::access::target::local>
        scratch(work_group_size, h);
    h.parallel_for(sycl::nd_range<1>{num_work_items, work_group_size},
        [=](sycl::nd_item<1> item) {
            size_t loc_id = item.get_local_id(0);
            int sum = 0;
            for (int i = loc_id; i < data_size; i += num_work_items)
                sum += buf_acc[i];
            scratch[loc_id] = sum;
            for (int i = work_group_size / 2; i > 0; i >>= 1) {
                item.barrier(sycl::access::fence_space::local_space);
                if (loc_id < i)
                    scratch[loc_id] += scratch[loc_id + i];
                }
            if (loc_id == 0)
                sum_acc[0] = scratch[0];
        });
    });
// reductionMultipleQSingleC main end
q2.wait();
sycl::host_accessor h_acc(sum_buf);
sum = h_acc[0];
elapsed += timer.Elapsed();
}
elapsed = elapsed / iter;
if (sum == sum_expected)
    std::cout << "SUCCESS: Time reductionMultipleQSingleContext = " << elapsed
        << "s" << " sum = " << sum << "\n";
else
    std::cout << "ERROR: reductionMultipleQSingleContext Expected "
        << sum_expected << " but got " << sum << "\n";
return sum;
} // end reductionMultipleQSingleC
```
8.0 Using Libraries for GPU Offload

Several libraries are available with oneAPI toolkits that can simplify the programming process by providing specialized APIs for use in optimized applications. This section provides steps on using the libraries, including code samples, for application accelerations. Detailed information about each library, including the available APIs, is available in the main documentation for the specific library.

8.1 Using Performance Libraries

This section discusses using efficient functions from libraries like oneAPI Math Kernel Library (oneMKL) or oneAPI Deep Neural Network Library (oneDNN) instead of hand-coded alternatives. Unless you’re an expert studying a particular mathematical operation, it’s usually a bad idea to write your own version of that operation. For example, matrix multiplication is a common, straightforward mathematical operation:

\[ C_{m,n} = A_{m,k} \times B_{k,n} = \sum_{k} A_{m,k} \times B_{k,n} \]

It’s also easy to implement with just a few lines of code:

```cpp
// Multiply matrices A and B
for (m = 0; m < M; m++) {
    for (n = 0; n < N; n++) {
        C[m][n] = 0.0;
        for (k = 0; k < K; k++) {
            C[m][n] += A[m][k] * B[k][n];
        }
    }
} // End matrix multiplication
```

However, this naive implementation won’t give the best possible performance. Simple visual inspection of the inner loop shows non-contiguous memory access for matrix B. Cache reuse, and hence performance, will be poor.

It’s not difficult to port the naive algorithm to Data Parallel C++ (DPC++) to offload the matrix multiplication kernel to an accelerator. The following code initializes the queue to submit work to the default device and allocates space for the matrices in unified shared memory (USM):

```cpp
// Initialize SYCL queue
sycl::queue Q(sycl::default_selector{});
auto sycl_device = Q.get_device();
auto sycl_context = Q.get_context();
std::cout << "Running on: "
```

(continues on next page)
// Allocate matrices A, B, and C in USM
auto A = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
for (m = 0; m < M; m++)
    A[m] = sycl::malloc_shared<float>(K, sycl_device, sycl_context);

auto B = sycl::malloc_shared<float *>(K, sycl_device, sycl_context);
for (k = 0; k < K; k++)
    B[k] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);

auto C = sycl::malloc_shared<float *>(M, sycl_device, sycl_context);
for (m = 0; m < M; m++)
    C[m] = sycl::malloc_shared<float>(N, sycl_device, sycl_context);

// Initialize matrices A, B, and C

Data in USM can be moved between host and device memories by the DPC++ runtime. Explicit buffering is not required. To offload the computation to the default accelerator, it is converted to a DPC++ kernel and submitted to the queue:

Listing 83:
/examples/libraries-kernel/naive_matmul_sycl.cpp

Common, computationally demanding operations like matrix multiplication are well-studied. Experts have devised a number of algorithms that give better performance than naive implementations of the basic mathematical formulas. They also use tuning techniques like cache blocking and loop unrolling to achieve performance regardless of the shapes of matrices A and B.

oneMKL provides an optimized general matrix multiplication function (oneapi::mkl::blas::gemm) that gives high performance on the host processor or a variety of accelerator devices. The matrices are allocated in USM as before, and passed to the gemm function along with the device queue, matrix dimensions, and various other options:
As expected, the library function gives better performance and is more versatile than the naive implementations. For example, the library function can transpose one or both matrices before multiplication, if necessary.

### Table 11: Matrix A Dimensions (time in seconds)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>4000 x 4000</th>
<th>8000 x 2000</th>
<th>2000 x 8000</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive DPC++</td>
<td>19.2</td>
<td>38.1</td>
<td>9.8</td>
<td>Gen9</td>
</tr>
<tr>
<td>oneMKL gemm</td>
<td>0.9</td>
<td>1.3</td>
<td>0.8</td>
<td>Gen9</td>
</tr>
</tbody>
</table>

This simple example illustrates the separation of concerns between application developers and tuning experts. The former should rely on the latter to encapsulate common computations in highly-optimized libraries. The oneAPI specification defines many libraries to help create accelerated applications, e.g.:

- oneMKL for math operations
- oneDAL for data analytics and machine learning
- oneDNN for the development of deep learning frameworks
- oneVPL for video processing

Check whether your required operation is already available in a oneAPI library before creating your own implementation of it.

### 8.2 Using Standard Library Functions in DPC++ Kernels

Some, but not all, standard C++ functions can be called inside DPC++ kernels. See Chapter 18 (Libraries) of Data Parallel C++ for an overview of supported functions. A simple example is provided here to illustrate what happens when an unsupported function is called from a DPC++ kernel. The following program generates a sequence of random numbers using the `rand()` function:

Listing 85: /examples/libraries-stdlib/external_rand.cpp

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
```
The program can be compiled to execute the DPC++ kernel on host (i.e., the SYCL host selector), the CPU (i.e., cpu_selector), or GPU (i.e., gpu_selector) devices. It compiles without errors on all three devices, and runs correctly on the CPU, but fails when run on the GPU:

```
$ dpccpp -DHOST -std=c++17 -fsycl external_rand.cpp -o external_rand
$ ./external_rand
Running on: Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz
0.572586
0.691008
0.451763
0.793325
```


The failure occurs during Just-In-Time (JIT) compilation because of an undefined reference to `rand()`. Even though this function is declared SYCL_EXTERNAL, there's no SYCL equivalent to the rand() function on the GPU device.

Fortunately, the DPC++ library contains alternatives to many standard C++ functions, including those to generate random numbers. The following example shows equivalent functionality using the Intel® oneAPI DPC++ Library (oneDPL) and the Intel oneAPI Math Kernel Library (oneMKL):

```
#include <CL/sycl.hpp>
#include <iostream>
#include <oneapi/dpl/random>
#include <oneapi/mkl/rng.hpp>

int main(int argc, char **argv) {
    unsigned int N = (argc == 1) ? 20 : std::stoi(argv[1]);
    if (N < 20)
        N = 20;
    // Generate sequences of random numbers between [0.0, 1.0] using oneDPL and
    // oneMKL
```

```cpp
sycl::queue Q(sycl::gpu_selector{});
std::cout << "Running on: "
  << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

auto test1 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());
auto test2 = sycl::malloc_shared<float>(N, Q.get_device(), Q.get_context());

std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value

// oneDPL random number generator on GPU device
-clock_t start_time = clock(); // Start timer
Q.parallel_for(N, [=](auto idx) {
  oneapi::dpl::minstd_rand rng_engine(seed, idx); // Initialize RNG engine
  oneapi::dpl::uniform_real_distribution<float>
    rng_distribution; // Set RNG distribution
  test1[idx] = rng_distribution(rng_engine); // Generate RNG sequence
}).wait();

-clock_t end_time = clock(); // Stop timer
std::cout << "oneDPL took " << float(end_time - start_time) / CLOCKS_PER_SEC
  << " seconds to generate " << N
  << " uniformly distributed random numbers." << std::endl;

// oneMKL random number generator on GPU device
-start_time = clock(); // Start timer
oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine, set RNG distribution
oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
  rng_distribution(0.0, 1.0);
oneapi::mkl::rng::generate(rng_distribution, engine, N, test2)
  .wait(); // Generate RNG sequence
end_time = clock(); // Stop timer
std::cout << "oneMKL took " << float(end_time - start_time) / CLOCKS_PER_SEC
  << " seconds to generate " << N
  << " uniformly distributed random numbers." << std::endl;

// Show first ten random numbers from each method
std::cout << std::endl
  << "oneDPL"
  << "\t"
  << "oneMKL" << std::endl;
for (int i = 0; i < 10; i++)
  std::cout << test1[i] << " " << test2[i] << std::endl;

// Show last ten random numbers from each method
std::cout << "..." << std::endl;
for (size_t i = N - 10; i < N; i++)
  std::cout << test1[i] << " " << test2[i] << std::endl;
```

The necessary oneDPL and oneMKL functions are included in `<oneapi/dpl/random>` and `<oneapi/mkl/rng.hpp>`, respectively. The oneDPL and oneMKL examples perform the same sequence of operations: get a random number seed from the clock, initialize a random number engine, select the desired random number distribution, then generate the random numbers. The oneDPL code performs device offload explicitly using a DPC++ kernel. In the oneMKL code, the `mkl::rng` functions handle the device offload implicitly.

### 8.3 Efficiently Implementing Fourier Correlation Using oneAPI Math Kernel Library (oneMKL)

Now that straightforward use of oneMKL kernel functions has been covered, let's look at a more complex mathematical operation: cross-correlation. Cross-correlation has many applications, e.g.: measuring the similarity of two 1D signals, finding the best translation to overlay similar images, volumetric medical image segmentation, etc.

Consider the following simple signals, represented as vectors of ones and zeros:

| Signal 1: 0 0 0 0 0 1 1 0 |
| Signal 2: 0 0 1 1 0 0 0 0 |

The signals are treated as circularly shifted versions of each other, so shifting the second signal three elements relative to the first signal will give the maximum correlation score of two:

| Signal 1: 0 0 0 0 0 1 1 0 |
| Signal 2: 0 0 1 1 0 0 0 0 |

Correlation: \((1 \times 1) + (1 \times 1) = 2\)

Shifts of two or four elements give a correlation score of one. Any other shift gives a correlation score of zero. This is computed as follows:

\[
corr_\alpha = \sum_{i=0}^{N-1} sig1_i \times sig2_{i+\alpha}
\]

where \(N\) is the number of elements in the signal vectors and \(\alpha\) is the shift of \(sig2\) relative to \(sig1\).

Real signals contain more data (and noise) but the principle is the same whether you are aligning 1D signals, overlaying 2D images, or performing 3D volumetric image registration. The goal is to find the translation that maximizes correlation. However, the brute force summation shown above requires \(N\) multiplications and additions for every \(N\) shifts. In 1D, 2D, and 3D, the problem is \(O(N^2)\), \(O(N^3)\), and \(O(N^4)\), respectively.

The Fourier correlation algorithm is a much more efficient way to perform this computation because it takes advantage of the \(O(N\log N)\) of the Fourier transform:
where DFT is the discrete Fourier transform, IDFT is the inverse DFT, and CONJG is the complex conjugate. The Fourier correlation algorithm can be composed using oneMKL, which contains optimized forward and backward transforms and complex conjugate multiplication functions. Therefore, the entire computation can be performed on the accelerator device.

In many applications, only the final correlation result matters, so this is all that has to be transferred from the device back to the host.

In the following example, two artificial signals will be created on the device, transformed in-place, and then correlated. The host will retrieve the final result and report the optimal translation and correlation score. Conventional wisdom suggests that buffering would give the best performance because it provides explicit control over data movement between the host and the device.

To test this hypothesis, let’s generate two input signals:

```cpp
// Create buffers for signal data. This will only be used on the device.
 sycl::buffer<float> sig1_buf{N + 2};
sycl::buffer<float> sig2_buf{N + 2};

// Declare container to hold the correlation result (computed on the device,
// used on the host)
std::vector<float> corr(N + 2);
```

Random noise is often added to signals to prevent overfitting during neural network training, to add visual effects to images, or to improve the detectability of signals obtained from suboptimal detectors, etc. The buffers are initialized with random noise using a simple random number generator in oneMKL:

```cpp
// Open new scope to trigger update of correlation result
{
 sycl::buffer<float> corr_buf(corr);

// Initialize the input signals with artificial data
std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine

oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard> rng_distribution(-0.00005, 0.00005);

oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); // Noise
oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);
```

Notice that a new scope is opened and a buffer, `corr_buf`, is declared for the correlation result. When this buffer goes out of scope, `corr` will be updated on the host.

An artificial signal is placed at opposite ends of each buffer, similar to the trivial example above:
Now that the signals are ready, let's transform them using the DFT functions in oneMKL:

```cpp
// Initialize FFT descriptor
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
    oneapi::mkl::dft::domain::REAL>
transform_plan(N);
transform_plan.commit(Q);

// Perform forward transforms on real arrays
oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);
```

A single-precision, real-to-complex forward transform is committed to the SYCL queue, then an in-place DFT is performed on the data in both buffers. The result of $DFT(sig1)$ must now be multiplied by $CONJG(DFT(sig2))$. This could be done with a hand-coded kernel:

```cpp
Q.submit([&](sycl::handler &h) {
    sycl::accessor sig1_acc{sig1_buf, h, sycl::read_only};
    sycl::accessor sig2_acc{sig2_buf, h, sycl::read_only};
    sycl::accessor corr_acc{corr_buf, h, sycl::write_only};

    h.parallel_for<sycl::range<1>{N/2}, [=](auto idx) {
        corr_acc[idx*2+0] = sig1_acc[idx*2+0] * sig2_acc[idx*2+0] +
                            sig1_acc[idx*2+1] * sig2_acc[idx*2+1];
        corr_acc[idx*2+1] = sig1_acc[idx*2+1] * sig2_acc[idx*2+0] -
                            sig1_acc[idx*2+0] * sig2_acc[idx*2+1];
    });
}); // End signal initialization
```

However, this basic implementation is unlikely to give optimal cross-architecture performance. Fortunately, the oneMKL function, `oneapi::mkl::vm::mulbyconj`, can be used for this step. The `mulbyconj` function expects `std::complex<float>` input, but the buffers were initialized as the `float` data type. Even though they contain complex data after the forward transform, the buffers will have to be recast:
Listing 91: /examples/libraries-fcorr/fcorr_1d_buffers.cpp

```cpp
auto sig1_buf_cplx =
    sig1_buf.template reinterpret<
        std::complex<float>,
        1>(N + 2) / 2;
auto sig2_buf_cplx =
    sig2_buf.template reinterpret<
        std::complex<float>,
        1>(N + 2) / 2;
auto corr_buf_cplx =
    corr_buf.template reinterpret<
        std::complex<float>,
        1>(N + 2) / 2;
oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
    corr_buf_cplx);
```

The IDFT step completes the computation:

Listing 92: /examples/libraries-fcorr/fcorr_1d_buffers.cpp

```cpp
// Perform backward transform on complex correlation array
oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);
```

When the scope that was opened at the start of this example is closed, the buffer holding the correlation result goes out of scope, forcing an update of the host container. This is the only data transfer between the host and the device.

The complete Fourier correlation implementation using explicit buffering is included below:

Listing 93: /examples/libraries-fcorr/fcorr_1d_buffers.cpp

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
// ==============================================================
#include <CL/sycl.hpp>
#include <iostream>
#include <mkl.h>
#include <oneapi/mkl/dfti.hpp>
#include <oneapi/mkl/rng.hpp>
#include <oneapi/mkl/vm.hpp>

int main(int argc, char **argv) {
    unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);
    if ((N % 2) != 0)
        N++;
    if (N < 32)
        N = 32;
    // Initialize SYCL queue
    sycl::queue Q(sycl::default_selector{});
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
    // Create buffers for signal data. This will only be used on the device.
    sycl::buffer<float> sig1_buf(N + 2);

    // Perform forward transform on signal data
    oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf, sig2_buf);
    // Perform backward transform on complex correlation array
    oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);
    // Copy result back to host container
    sycl::event copy_result = Q.copy_from_host(corr_buf, sig1_buf);
    copy_result.wait();
    return 0;
}
```

(continues on next page)
sycl::buffer<float> sig2_buf(N + 2);

// Declare container to hold the correlation result (computed on the device, 
// used on the host)
std::vector<float> corr(N + 2);

// Open new scope to trigger update of correlation result
{
  sycl::buffer<float> corr_buf(corr);

  // Initialize the input signals with artificial data
  std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
  oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
    // Set RNG distribution
  oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
    rng_distribution(-0.00005, 0.00005);
  oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1_buf); // Noise
  oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2_buf);

  Q.submit([&](sycl::handler &h) {
    sycl::accessor sig1_acc{sig1_buf, h, sycl::write_only};
    sycl::accessor sig2_acc{sig2_buf, h, sycl::write_only};
    h.single_task<>() {
      sig1_acc[N - N / 4 - 1] = 1.0;
      sig1_acc[N - N / 4] = 1.0;
      sig1_acc[N - N / 4 + 1] = 1.0; // Signal
      sig2_acc[N / 4 - 1] = 1.0;
      sig2_acc[N / 4] = 1.0;
      sig2_acc[N / 4 + 1] = 1.0;
    });
  }); // End signal initialization

  clock_t start_time = clock(); // Start timer

  // Initialize FFT descriptor
  oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
    oneapi::mkl::dft::domain::REAL>
    transform_plan(N);
  transform_plan.commit(Q);

  // Perform forward transforms on real arrays
  oneapi::mkl::dft::compute_forward(transform_plan, sig1_buf);
  oneapi::mkl::dft::compute_forward(transform_plan, sig2_buf);

  // Compute: DFT(sig1) * CONJG(DFT(sig2))
  auto sig1_buf_cplx =
    sig1_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
  auto sig2_buf_cplx =
    sig2_buf.template reinterpret<std::complex<float>, 1>((N + 2) / 2);
  auto corr_buf_cplx =
corr_buf.template reinterpret<
std::complex<float>, 1>((N + 2) / 2);
oneapi::mkl::vm::mulbyconj(Q, N / 2, sig1_buf_cplx, sig2_buf_cplx,
    corr_buf_cplx);

// Perform backward transform on complex correlation array
oneapi::mkl::dft::compute_backward(transform_plan, corr_buf);

clock_t end_time = clock(); // Stop timer
std::cout << "The 1D correlation (N = " << N << ") took "
    << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
    << std::endl;

} // Buffer holding correlation result is now out of scope, forcing update of
// host container

// Find the shift that gives maximum correlation value
float max_corr = 0.0;
int shift = 0;
for (unsigned int idx = 0; idx < N; idx++) {
    if (corr[idx] > max_corr) {
        max_corr = corr[idx];
        shift = idx;
    }
}
int _N = static_cast<int>(N);
shift =
    (shift > _N / 2) ? shift - _N : shift; // Treat the signals as circularly
    // shifted versions of each other.
std::cout << "Shift the second signal " << shift
    << " elements relative to the first signal to get a maximum, "
    << "normalized correlation score of "
    << max_corr / N << "." << std::endl;

The Fourier correlation algorithm will now be reimplemented using Unified Shared Memory (USM) to compare
to explicit buffering. Only the differences in the two implementations will be highlighted. First, the signal and
correlation arrays are allocated in USM, then initialized with artificial data:

### Listing 94: /examples/libraries-fcorr/fcorr_1d_usm.cpp

// Initialize signal and correlation arrays
auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);

// Initialize input signals with artificial data
std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
    // Set RNG distribution
oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
    rng_distribution(-0.00005, 0.00005);
// Warning: These statements run on the device.
auto evt1 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // Noise
auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);
evt1.wait();
evt2.wait();

// Warning: These statements run on the host, so sig1 and sig2 will have to be
// updated on the device.
sig1[N - N / 4 - 1] = 1.0;
sig1[N - N / 4] = 1.0;
sig1[N - N / 4 + 1] = 1.0; // Signal
sig2[N / 4 - 1] = 1.0;
sig2[N / 4] = 1.0;
sig2[N / 4 + 1] = 1.0;

The rest of the implementation is largely the same except that pointers to USM are passed to the oneMKL functions instead of SYCL buffers:

Listing 95: /examples/libraries-fcorr/fcorr_1d_usm.cpp

// Perform forward transforms on real arrays
evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);

// Compute: DFT(sig1) * CONJG(DFT(sig2))
oneapi::mkl::vm::mulbyconj(
    Q, N / 2, reinterpret_cast<std::complex<float>*>(sig1),
    reinterpret_cast<std::complex<float>*>(sig2),
    reinterpret_cast<std::complex<float>*>(corr), {evt1, evt2})
    .wait();

// Perform backward transform on complex correlation array
oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();

It is also necessary to free the allocated memory:

Listing 96: /examples/libraries-fcorr/fcorr_1d_usm.cpp

sycl::free(sig1, sycl_context);
sycl::free(sig2, sycl_context);
sycl::free(corr, sycl_context);

The USM implementation has a more familiar syntax. It is also conceptually simpler because it relies on implicit data transfer handled by the DPC++ runtime. However, a programmer error hurts performance.

Notice the warning messages in the previous code snippets. The oneMKL random number generation engine is initialized on the device, so sig1 and sig2 are initialized with random noise on the device. Unfortunately, the code adding the artificial signal runs on the host, so the DPC++ runtime has to make the host and device data consistent. The signals used in Fourier correlation are usually large, especially in 3D imaging applications, so unnecessary data transfer degrades performance.
Updating the signal data directly on the device keeps the data consistent, thereby avoiding the unnecessary data transfer:

### Listing 97: /examples/libraries-fcorr/fcorr_1d_usm_fixed.cpp

```cpp
Q.single_task<>([
    sig1[N - N / 4 - 1] = 1.0;
    sig1[N - N / 4] = 1.0;
    sig1[N - N / 4 + 1] = 1.0; // Signal
    sig2[N / 4 - 1] = 1.0;
    sig2[N / 4] = 1.0;
    sig2[N / 4 + 1] = 1.0;
]()).wait();
```

The explicit buffering and USM implementations now have equivalent performance, indicating that the DPC++ runtime is good at avoiding unnecessary data transfers (provided the programmer pays attention to data consistency).

The complete Fourier correlation implementation in USM is included below:

### Listing 98: /examples/libraries-fcorr/fcorr_1d_usm_fixed.cpp

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//================================================================
#include <CL/sycl.hpp>
#include <iostream>
#include <mkl.h>
#include <oneapi/mkl/dfti.hpp>
#include <oneapi/mkl/rng.hpp>
#include <oneapi/mkl/vm.hpp>

int main(int argc, char **argv) {
    unsigned int N = (argc == 1) ? 32 : std::stoi(argv[1]);
    if ((N % 2) != 0)
        N++;
    if (N < 32)
        N = 32;

    // Initialize SYCL queue
    sycl::queue Q(sycl::default_selector{});
    auto sycl_device = Q.get_device();
    auto sycl_context = Q.get_context();
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    // Initialize signal and correlation arrays
    auto sig1 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
    auto sig2 = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
    auto corr = sycl::malloc_shared<float>(N + 2, sycl_device, sycl_context);
```

(continues on next page)
// Initialize input signals with artificial data
std::uint32_t seed = (unsigned)time(NULL); // Get RNG seed value
oneapi::mkl::rng::mcg31m1 engine(Q, seed); // Initialize RNG engine
oneapi::mkl::rng::uniform<float, oneapi::mkl::rng::uniform_method::standard>
    rng_distribution(-0.00005, 0.00005);
auto evt1 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig1); // Noise
auto evt2 = oneapi::mkl::rng::generate(rng_distribution, engine, N, sig2);

Q.single_task<>([=]() {
    sig1[N - N / 4 - 1] = 1.0;
    sig1[N - N / 4] = 1.0;
    sig1[N - N / 4 + 1] = 1.0; // Signal
    sig2[N / 4 - 1] = 1.0;
    sig2[N / 4] = 1.0;
    sig2[N / 4 + 1] = 1.0;
}).wait();

clock_t start_time = clock(); // Start timer

// Initialize FFT descriptor
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
    oneapi::mkl::dft::domain::REAL>
    transform_plan(N);
transform_plan.commit(Q);

// Perform forward transforms on real arrays
evt1 = oneapi::mkl::dft::compute_forward(transform_plan, sig1);
evt2 = oneapi::mkl::dft::compute_forward(transform_plan, sig2);

// Compute: DFT(sig1) * CONJG(DFT(sig2))
oneapi::mkl::vm::mulbyconj(Q, N / 2, reinterpret_cast<std::complex<float> *>(sig1),
    reinterpret_cast<std::complex<float> *>(sig2),
    reinterpret_cast<std::complex<float> *>(corr), {evt1, evt2}) .wait();

// Perform backward transform on complex correlation array
oneapi::mkl::dft::compute_backward(transform_plan, corr).wait();

clock_t end_time = clock(); // Stop timer
std::cout << "The 1D correlation (N = " << N << ") took " << float(end_time - start_time) / CLOCKS_PER_SEC << " seconds."
    << std::endl;

// Find the shift that gives maximum correlation value
float max_corr = 0.0;
int shift = 0;
for (unsigned int idx = 0; idx < N; idx++) {
    if (corr[idx] > max_corr) {
        max_corr = corr[idx];
        shift = idx;
    }
}

int _N = static_cast<int>(N);
shift =
    (shift > _N / 2) ? shift - _N : shift; // Treat the signals as circularly
    // shifted versions of each other.
std::cout << "Shift the second signal " << shift
    << " elements relative to the first signal to get a maximum, "
    << "normalized correlation score of "
    << max_corr / _N << "." << std::endl;

// Cleanup
sycl::free(sig1, sycl_context);
sycl::free(sig2, sycl_context);
sycl::free(corr, sycl_context);

Note that the final step of finding the location of the maximum correlation value is performed on the host. It would be better to do this computation on the device, especially when the input data is large. Fortunately, the maxloc reduction is a common parallel pattern that can be implemented using DPC++. This is left as an exercise for the reader, but Figure 14-11 of Data Parallel C++ provides a suitable example to help you get started.
9.0 Host/Device Memory, Buffer and USM

Accelerators have access to a rich memory hierarchy. Utilizing the right level in the hierarchy is critical to getting the best performance.

In this section we cover topics related to declaration, movement, and access to the memory hierarchy.

9.1 Performance Impact of USM and Buffers

SYCL offers several choices for managing memory on the device. This section discusses the performance tradeoffs, briefly introducing the concepts. For an in-depth explanation, see Data Parallel C++.

As with other language features, the specification defines the behavior but not the implementation, so performance characteristics can change between software versions and devices. This guide provide best practices.

Buffers. A buffer is a container for data that can be accessed from a device and the host. The SYCL runtime manages memory by providing APIs for allocating, reading, and writing memory. The runtime is responsible for moving data between host and device, and synchronizing access to the data.

Unified Shared Memory (USM). USM allows reading and writing of data with conventional pointers, in contrast to buffers where access to data is exclusively by API. USM has two commonly-used variants. Device allocations can only be accessed from the device and therefore require explicit movement of data between host and device. Shared allocations can be referenced from device or host, with the runtime automatically moving memory.

We illustrate the tradeoffs between choices by showing the same example program written with the three models. To highlight the issues, we use a program where a GPU and the host cooperatively compute, and therefore need to ship data back and forth.

We start by showing the serial computation below. Assume that we want to perform the loop at line 9 on the GPU and the loop on line 14 on the CPU. Both loops read and write the data array so data must move between host and GPU for each iteration of the loop in line 8.

Listing 99: /examples/usm/usm-buffer.cpp

```c++
void serial(int stride) {
   // Allocate and initialize data
   float *data = new float[data_size];
   init(data);
   timer it;

   for (int i = 0; i < time_steps; i++) {
      for (int j = 0; j < data_size; j++) {
         for (int k = 0; k < device_steps; k++)
            data[j] += 1.0;
      }

   for (int j = 0; j < data_size; j += stride)
```
data[j] += 1.0;
}
put_elapsed_time(it);
check(data);
delete[] data;
} // serial

9.1.1 Buffers

Below, we show the same computation using buffers to manage data. A buffer is created at line 3 and initialized by the init function. The init function is not shown. It accepts an accessor or a pointer. The parallel_for executes the kernel defined on line 13. The kernel uses the device_data accessor to read and write data in buffer_data.

Note that the code does not specify the location of data. An accessor indicates when and where the data is needed, and the SYCL runtime moves the data to the device (if necessary) and then launches the kernel. The host_accessor on line 21 indicates that the data will be read/written on the host. Since the kernel is also read/writing buffer_data, the host_accessor constructor waits for the kernel to complete and moves data to the host to perform the read/write on line 23. In the next iteration of the loop the accessor constructor on line 11 waits until the until the data is moved back to the device, which effectively delays launching the kernel.

```
void buffer_data(int stride) {
    // Allocate buffer, initialize on host
    sycl::buffer<float> buffer_data(data_size);
    init(sycl::host_accessor(buffer_data, sycl::write_only, sycl::no_init));

    timer it;
    for (int i = 0; i < time_steps; i++) {
        // Compute on device
        q.submit([&](auto &h) {
            sycl::accessor device_data(buffer_data, h);
            auto compute = [=](auto id) {
                for (int k = 0; k < device_steps; k++)
                    device_data[id] += 1.0;
            };
            h.parallel_for(data_size, compute);
        });

        // Compute on host
        sycl::host_accessor host_data(buffer_data);
        for (int i = 0; i < data_size; i += stride)
            host_data[i] += 1.0;
    }
    put_elapsed_time(it);
}
```
Performance Considerations

The data access on lines 15 and 23 appear to be simple array references, but they are implemented by the SYCL runtime with C++ operator overloading. The efficiency of accessor array references depends on the implementation. In practice, device code pays no overhead for overloading compared to direct memory references. The runtime does not know in advance which part of the buffer is accessed, so it must ensure all the data is on the device before the kernel begins. This is true today, but may change over time.

The same is not currently true for the host_accessor. The runtime does not move all the data to the host. The array references are implemented with more complex code and are significantly slower than native C++ array references. While it is acceptable to reference a small amount of data, computationally intensive algorithms using host_accessor pay a large performance penalty and should be avoided.

Another issue is concurrency. A host_accessor can block kernels that reference the same buffer from launching, even if the accessor is not actively being used to read/write data. Limit the scope that contains the host_accessor to the minimum possible. In this example, the host accessor on line 4 is destroyed after the init function returns and the host accessor on line 21 is destroyed at the end of each loop iteration.

9.1.2 Shared Allocations

Next we show the same algorithm implemented with shared allocations. Data is allocated on line 2. Accessors are not needed because USM-allocated data can be referenced with conventional allows pointers. Therefore, the array references on lines 10 and 15 can be implemented with simple indexing. The parallel_for on line 12 ends with a wait to ensure the kernel finishes before the host accesses data on line 15. Similar to buffers, the SYCL runtime ensures that all the data is resident on the device before launching a kernel. And like buffers, shared allocations are not copied to the host unless it is referenced. The first time the host references data, there is an operating system page fault, a page of data is copied from device to host, and execution continues. Subsequent references to data on the same page execute at full speed. When a kernel is launched, all of the host-resident pages are flushed back to the device.

Listing 101: /examples/usm/usm-buffer.cpp

```cpp
void shared_usm_data(int stride) {
    float *data = sycl::malloc_shared<float>(data_size, q);
    init(data);

    timer it;

    for (int i = 0; i < time_steps; i++) {
        auto compute = [=](auto id) {
            for (int k = 0; k < device_steps; k++)
                data[id] += 1.0;
        };
```
Performance Considerations

Compared to buffers, data references are simple pointers and perform well. However, servicing page faults to bring data to the host incurs overhead in addition to the cost of transferring data. The impact on the application depends on the reference pattern. Sparse random access has the highest overhead and linear scans through data have lower impact from page faults.

Since all synchronization is explicit and under programmer control, concurrency is not an issue for a well-designed program.

9.1.3 Device Allocations

The same program with device allocation can be found below. With device allocation, data can only be directly accessed on the device and must be explicitly copied to the host, as is done on line 21. All synchronization between device and host are explicit. Line 21 ends with a wait so the host code will not execute until the asynchronous copy finishes. The queue definition is not shown but uses an in-order queue so the memcpy on line 21 waits for the parallel_for on line 18 to complete.

Listing 102: /examples/usm/usm-buffer.cpp

```cpp
void device_usm_data(int stride) {
    // Allocate and initialize host data
    float *host_data = new float[data_size];
    init(host_data);

    // Allocate device data
    float *device_data = sycl::malloc_device<float>(data_size, q);
    timer it;

    for (int i = 0; i < time_steps; i++) {
        // Copy data to device and compute
        q.memcpy(device_data, host_data, sizeof(float) * data_size);
        auto compute = [=](auto id) {
            for (int k = 0; k < device_steps; k++)
                device_data[id] += 1.0;
        };
```

(continues on next page)
Performance Considerations

Both data movement and synchronization are explicit and under the full control of the programmer. Array references are array references on the host, so it has neither the page faults overhead of shared allocations, nor the overloading overhead associated with buffers. Shared allocations only transfer data that the host actually references, with a memory page granularity. In theory, device allocations allow on-demand movement of any granularity. In practice, fine-grained, asynchronous movement of data can be complex and most programmers simply move the entire data structure once. The requirement for explicit data movement and synchronization makes the code more complicated, but device allocations can provide the best performance.

9.2 Optimizing Memory Movement Between Host and Accelerator

Buffers can be created using properties to control how they are allocated. One such property is `use_host_ptr`. This informs the runtime that if possible, the host memory should be directly used by the buffer instead of a copy. This avoids the need to copy the content of the buffer back and forth between the host memory and the buffer memory, potentially saving time during buffer creation and destruction. To take another case, when the GPU and CPU have shared memory, it is possible to avoid copies of memory through sharing of pages. But for page sharing to be possible, the allocated memory needs to have some properties like being aligned on page boundary. In case of discrete devices, the benefit may not be realized because any memory operation by the accelerator will have to go across PCIe or some other slower interface than the memory of the accelerator.

The following code shows how to print the memory addresses on the host, inside the buffer, and on the accelerator device inside the kernel.

**Listing 103:**

```

int VectorAdd0(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b, 
               AlignedVector<int> &sum, int iter) {
    sycl::range num_items{a.size()};
    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

```
for (int i = 0; i < iter; i++) {
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer sum_buf(sum.data(), num_items, props);
    
    sycl::host_accessor a_host_acc(a_buf);
    std::cout << "add0: buff memory address = " << a_host_acc.get_pointer()
             << "\n";
    std::cout << "add0: address of vector a = " << a.data() << "\n";
}
q.submit([&](auto &h) {  
    // Input accessor
    sycl::accessor a_acc(a_buf, h, sycl::read_only);
    sycl::accessor b_acc(b_buf, h, sycl::read_only);
    // Output accessor
    sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    sycl::stream out(1024 * 1024, 1 * 128, h);
    h.parallel_for(num_items, [=](auto i) {
        if (i[0] == 0)
            out << "add0: dev addr = " << a_acc.get_pointer() << "\n";
        sum_acc[i] = a_acc[i] + b_acc[i];
    });
    });
q.wait();
return (0);
}

When this program is run, it can be seen that the addresses for all three (host, in the buffer, and on the accelerator) are the same when the property use_host_ptr is set for integrated GPU devices. But for discrete GPU devices, the buffer and device addresses will be different. Also note that in line 1, none of the incoming arguments are declared to be const. If these are declared const then during buffer creation they are copied and new memory is allocated instead of reusing the memory in the host vectors. The code snippet below demonstrates this. When this code is executed, we see that the addresses associated with the incoming vectors are different from the memory present in the buffer and also the memory present in the accelerator device.

Listing 104:
/examples/memory-movement/vec-buffer-host.cpp

int VectorAdd1(sycl::queue &q, const AlignedVector<int> &a,
               const AlignedVector<int> &b, AlignedVector<int> &sum, int iter) {
    sycl::range num_items{a.size()};
    
    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    
    for (int i = 0; i < iter; i++) {
        sycl::buffer a_buf(a, props);
        sycl::buffer b_buf(b, props);
        sycl::buffer sum_buf(sum.data(), num_items, props);
        
        sycl::host_accessor a_host_acc(a_buf);
        std::cout << "add0: buff memory address = " << a_host_acc.get_pointer()
                 << "\n";
        std::cout << "add0: address of vector a = " << a.data() << "\n";
    }
    q.submit([&](auto &h) {  
        // Input accessor
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
        sycl::stream out(1024 * 1024, 1 * 128, h);
        h.parallel_for(num_items, [=](auto i) {
            if (i[0] == 0)
                out << "add0: dev addr = " << a_acc.get_pointer() << "\n";
            sum_acc[i] = a_acc[i] + b_acc[i];
        });
        });
    q.wait();
    return (0);
}
The kernel `vectorAdd3` will not incur the cost of copying the memory contents from the buffer to the accelerator device because the `use_host_ptr` property is set while creating the buffers, and the buffers are aligned on a page boundary for an integrated GPU device. If memory pointed to by a buffer is not aligned on a page boundary, then new memory is allocated that aligns on a page boundary and the contents of the buffer are copied into that memory. This new memory from the buffer is then shared with the accelerator either by copying the contents from the buffer on host to the device (for accelerators that do not share any memory) or by using the page tables to avoid a physical copy of memory available on the device (for accelerators that share memory).

**Listing 105:**
/examples/memory-movement/vec-buffer-host.cpp

```cpp
int VectorAdd2(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &sum, int iter) {
    sycl::range num_items{a.size()};

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};

    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        sycl::buffer a_buf(a, props);
        sycl::buffer b_buf(b, props);
        sycl::buffer sum_buf(sum.data(), num_items, props);
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
```

(continues on next page)
The kernel below will incur the cost of copying memory contents between the host and buffer, and also from the buffer to the accelerator.

**Listing 106:**
/examples/memory-movement/vec-buffer-host.cpp

```cpp
int VectorAdd3(sycl::queue &q, const AlignedVector<int> &a,
               const AlignedVector<int> &b, AlignedVector<int> &sum, int iter) {
    sycl::range num_items{a.size()};
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        sycl::buffer a_buf(a);
        sycl::buffer b_buf(b);
        sycl::buffer sum_buf(sum.data(), num_items);
        auto e = q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(num_items,
                [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
        });
    } wait();
    auto end = std::chrono::steady_clock::now();
    std::cout << "Vector add3 completed on device - took " << (end - start).count() << " u-secs\n";
    return ((end - start).count());
}
```

Care must be taken to ensure that unnecessary copies are avoided during the creation of buffers and passing the memory from the buffers to the kernels. Even when the accelerator shares memory with the host, a few additional conditions must be satisfied to avoid these extra copies.
9.3 Avoid moving data back and forth between host and device

The cost of moving data between host and device is quite high, especially in the case of discrete accelerators. So it is very important to avoid data transfers between host and device as much as possible. In some situations it may be required to bring the data that was computed by a kernel on the accelerator to the host and do some operation on it and send it back to the device for further processing. In such situation we will end up paying for the cost of device to host transfer and then again host to device transfer.

Consider the following example, where one kernel produces data through some operation (in this case vector add) into a new vector. This vector is then transformed into another vector by applying a function on each value and then fed as input into another kernel for some additional computation. This form of computation is quite common and occurs in many domains where algorithms are iterative and output from one computation needs to be fed as input into another computation. One classic example is machine learning models which are structured as layers of computation and output of one layer is input to the next layer.

```
Listing 107: /examples/host-device-memory/mem-move.cpp

double myFunc1(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
    sycl::range num_items{a.size()};
    VectorAllocator<int> alloc;
    AlignedVector<int> sum(a.size(), alloc);

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer c_buf(b, props);
    sycl::buffer d_buf(b, props);
    sycl::buffer res_buf(res, props);
    sycl::buffer sum_buf(sum.data(), num_items, props);

    Timer timer;
    for (int i = 0; i < iter; i++) {
        // kernel
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(num_items,
                [=](auto id) { sum_acc[id] = a_acc[id] + b_acc[id]; });
        });

        {
            sycl::host_accessor h_acc(sum_buf);
            for (int j = 0; j < a.size(); j++)
                if (h_acc[j] > 10)
                    h_acc[j] = 1;
            else
```

(continues on next page)
Instead of bringing the data to the host and applying the function to the data and sending it back to the device in the second kernel, you can create a kernel to execute this function on the device itself. This has the advantage of avoiding the round trip of data from device to host. This technique is shown in the example below, which is functionally the same as the code before. We now introduce a third kernel \texttt{kernel3} that operates on the intermediate data in \texttt{accum_buf} in between \texttt{kernel1} and \texttt{kernel2}.

\textbf{Listing 108:} /examples/host-device-memory/mem-move.cpp

```cpp
double myFunc2(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
    sycl::range num_items{a.size()};
    VectorAllocator<int> alloc;
    AlignedVector<int> sum(a.size(), alloc);

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer c_buf(c, props);
    sycl::buffer d_buf(d, props);
    sycl::buffer res_buf(res, props);
    sycl::buffer sum_buf(sum.data(), num_items, props);

    Timer timer;
    for (int i = 0; i < iter; i++) {
        // kernel1
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor c_acc(c_buf, h, sycl::read_only);
            sycl::accessor d_acc(d_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(num_items, [=](auto id) {
                res_acc[id] = sum_acc[id] * c_acc[id] + d_acc[id];
            });
        });
        q.wait();
    }
    double elapsed = timer.Elapsed() / iter;
    return (elapsed);
} // end myFunc1
```

(continues on next page)
There are other ways to optimize this example. For instance, the clipping operation in kernel3 can be merged into the computation of kernel1 as shown below. This is kernel fusion and has the added advantage of not launching a third kernel. The DPCPP compiler cannot do this kind of optimization. In some specific domains like machine learning, there are graph compilers that operate on the ML models and fuse the operations, which has the same impact.

**Listing 109:** /examples/host-device-memory/mem-move.cpp

```cpp
double myFunc3(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
               AlignedVector<int> &c, AlignedVector<int> &d,
               AlignedVector<int> &res, int iter) {
    sycl::range num_items{a.size()};
    VectorAllocator<int> alloc;
    // ...
AlignedVector<int> sum(a.size(), alloc);

const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
sycl::buffer a_buf(a, props);
sycl::buffer b_buf(b, props);
sycl::buffer c_buf(b, props);
sycl::buffer d_buf(b, props);
sycl::buffer res_buf(res, props);
sycl::buffer sum_buf(sum.data(), num_items, props);

Timer timer;
for (int i = 0; i < iter; i++) {
    // kernel1
    q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(num_items, [=](auto i) {
            int t = a_acc[i] + b_acc[i];
            if (t > 10)
                sum_acc[i] = 1;
            else
                sum_acc[i] = 0;
        });
    });

    // kernel2
    q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor sum_acc(sum_buf, h, sycl::read_only);
        sycl::accessor c_acc(c_buf, h, sycl::read_only);
        sycl::accessor d_acc(d_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

        h.parallel_for(num_items, [=](auto i) {
            res_acc[i] = sum_acc[i] * c_acc[i] + d_acc[i];
        });
    });
    q.wait();
}

double elapsed = timer.Elapsed() / iter;
return (elapsed);
} // end myFunc3

We can take this kernel fusion one level further and fuse both kernel1 and kernel2 as shown in the code below. This gives very good performance since it avoids the intermediate accum_buf completely, saving memory in addition to launching an additional kernel. Most of the performance benefit in this case is due to improvement in locality of memory references.
### Listing 110: /examples/host-device-memory/mem-move.cpp

```cpp
double myFunc4(sycl::queue &q, AlignedVector<int> &a, AlignedVector<int> &b,
                AlignedVector<int> &c, AlignedVector<int> &d,
                AlignedVector<int> &res, int iter) {
    sycl::range num_items{a.size()};
    VectorAllocator<int> alloc;

    const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
    sycl::buffer a_buf(a, props);
    sycl::buffer b_buf(b, props);
    sycl::buffer c_buf(c, props);
    sycl::buffer d_buf(d, props);
    sycl::buffer res_buf(res, props);

    Timer timer;
    for (int i = 0; i < iter; i++) {
        // Kernel
        q.submit([&](auto &h) {
            // Input accessors
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            sycl::accessor c_acc(c_buf, h, sycl::read_only);
            sycl::accessor d_acc(d_buf, h, sycl::read_only);

            // Output accessor
            sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(num_items, [=](auto i) {
                int t = a_acc[i] + b_acc[i];
                if (t > 10)
                    res_acc[i] = c_acc[i] + d_acc[i];
                else
                    res_acc[i] = d_acc[i];
            });
        });
        q.wait();
    }
    double elapsed = timer.Elapsed() / iter;
    return (elapsed);
} // end myFunc4
```

### 9.4 Avoid Declaring Buffers in a Loop

When kernels are repeatedly launched inside a for-loop, you can prevent repeated allocation and freeing of a buffer by declaring the buffer outside the loop. Declaring a buffer inside the loop introduces repeated host-to-device and device-to-host memory copies.

In the following example, the kernel is repeatedly launched inside a for-loop. The buffer C is used as a temporary array, where it is used to hold values in an iteration, and the values assigned in one iteration are not used in any other iteration. Since the buffer C is declared inside the for-loop, it is allocated and freed in every loop iteration. In addition to the allocation and freeing of the buffer, the memory associated with the buffer is redundantly
transferred from host to device and device to host in each iteration.

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================================
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 25;
constexpr int STEPS = 100000;

int main() {
    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;

    // Create 2 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);

    Q.submit([&](auto &h) {
        // Create device accessors.
        // The property no_init lets the runtime know that the
        // previous contents of the buffer can be discarded.
        sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
        sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(N, [=](auto i) {
            aA[i] = 10;
            aB[i] = 20;
        });
    });

    for (int j = 0; j < STEPS; j++) {
        sycl::buffer<int> CBuf(&CData[0], N);
        Q.submit([&](auto &h) {
            // Create device accessors.
            sycl::accessor aA(ABuf, h);
            sycl::accessor aB(BBuf, h);
            sycl::accessor aC(CBuf, h);
            h.parallel_for(N, [=](auto i) {
                aC[i] = (aA[i] < aB[i]) ? -1 : 1;
                aA[i] += aC[i];
                aB[i] -= aC[i];
            });
        });
    } // end for
}
```

(continues on next page)
// Create host accessors.
const sycl::host_accessor haA(ABuf);
const sycl::host_accessor haB(BBuf);
printf("%d \%d\n", haA[N / 2], haB[N / 2]);
return 0;
}

A better approach would be to declare the buffer C before the for-loop, so that it is allocated and freed only once, resulting in improved performance by avoiding the redundant data transfers between host and device. The following kernel shows this change.

Listing 112: /examples/buffers/buf-kern2.cpp

//==============================================================
// Copyright © 2022 Intel Corporation
//
// SPDX-License-Identifier: MIT
//==============================================================
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 25;
constexpr int STEPS = 100000;

int main() {
  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  // Create 3 buffers, each holding N integers
  sycl::buffer<int> ABuf(&AData[0], N);
  sycl::buffer<int> BBuf(&BData[0], N);
  sycl::buffer<int> CBuf(&CData[0], N);

  Q.submit([&](auto &h) {
    // Create device accessors.
    // The property no_init lets the runtime know that the
    // previous contents of the buffer can be discarded.
    sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
    h.parallel_for(N, [=](auto i) {
      aA[i] = 10;
      aB[i] = 20;
    });
  });
for (int j = 0; j < STEPS; j++) {
    Q.submit([&](auto &h) {
        // Create device accessors.
        sycl::accessor aA(ABuf, h);
        sycl::accessor aB(BBuf, h);
        sycl::accessor aC(CBuf, h);
        h.parallel_for(N, [=](auto i) {
            aC[i] = (aA[i] < aB[i]) ? -1 : 1;
            aA[i] += aC[i];
            aB[i] -= aC[i];
        });
    });
} // end for

// Create host accessors.
const sycl::host_accessor haA(ABuf);
const sycl::host_accessor haB(BBuf);
printf("%d %d\n", haA[N / 2], haB[N / 2]);
return 0;
}

9.5 Buffer Accessor Modes

In DPC++, a buffer provides an abstract view of memory that can be accessed by the host or a device. A buffer cannot be accessed directly through the buffer object. Instead, we must create an accessor object that allows us to access the buffer’s data.

The access mode describes how we intend to use the memory associated with the accessor in the program. The accessor’s access modes are used by the runtime to create an execution order for the kernels and perform data movement. This will ensure that kernels are executed in an order intended by the programmer. Depending on the capabilities of the underlying hardware, the runtime can execute kernels concurrently if the dependencies do not give rise to dependency violations or race conditions.

For better performance, make sure that the access modes of accessors reflect the operations performed by the kernel. The compiler will flag an error when a write is done on an accessor which is declared as read_only. But the compiler does not change the declaration of an accessor form read_write to read if no write is done in the kernel.

The following example shows three kernels. The first kernel initializes the A, B, and C buffers, so we specify that the access modes for these buffers is write_only. The second kernel reads the A and B buffers, and reads and writes the C buffer, so we specify that the access mode for the A and B buffers is read_only, and the access mode for the C buffer is read_write.

The read_only access mode informs the runtime that the data needs to be available on the device before the kernel can begin executing, but the data need not be copied from the device to the host at the end of the computation.

If this second kernel were to use read_write for A and B instead of read_only, then the memory associated with A and B is copied from the device to the host at the end of kernel execution, even though the data has not
been modified by the device. Moreover, `read_write` creates unnecessary dependencies. If another kernel that reads A or B is submitted within this block, this new kernel cannot start until the second kernel has completed.

**Listing 113: /examples/buffer-accessors/kern1.cpp**

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================================
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;

int main() {
  int AData[N];
  int BData[N];
  int CData[N];

  sycl::queue Q;

  // Kernel1
  {
    // Create 3 buffers, each holding N integers
    sycl::buffer<int> ABuf(&AData[0], N);
    sycl::buffer<int> BBuf(&BData[0], N);
    sycl::buffer<int> CBuf(&CData[0], N);

    Q.submit([&](auto &h) {
      // Create device accessors.
      // The property no_init lets the runtime know that the
      // previous contents of the buffer can be discarded.
      sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
      sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

      h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
      });
    });

    // Kernel2
    {
      // Create 3 buffers, each holding N integers
      sycl::buffer<int> ABuf(&AData[0], N);
      sycl::buffer<int> BBuf(&BData[0], N);
      sycl::buffer<int> CBuf(&CData[0], N);
```

(continues on next page)
Specifying `read_only` accessor mode, instead of `read_write`, is especially useful when kernels are repeatedly launched inside a for-loop. If the access mode is `read_write`, the kernels launched will be serialized, because one kernel should finish its computation and the data should be ready before the next kernel can be launched. On the other hand, if the access mode is `read_only`, then the runtime can launch the kernels in parallel.

Note that the buffer declarations and kernels are launched inside a block. This will cause the buffers to go out of scope at the end of first kernel completion. This will trigger a copy of the contents from the device to the host. The second kernel is inside another block where new buffers are declared to the same memory and this will trigger a copy of this same memory again from the host to the device. This back-and-forth between host and device can be avoided by declaring the buffers once, ensuring that they are in scope during the lifetime of the memory pointed to by these buffers. A better way to write the code that avoids these unnecessary memory transfers is shown below.

```cpp
// Create 3 buffers, each holding N integers
```
sycl::buffer<int> ABuf(&AData[0], N);
sycl::buffer<int> BBuf(&BData[0], N);
sycl::buffer<int> CBuf(&CData[0], N);

// Kernel1
Q.submit([&](auto &h) {
    // Create device accessors.
    // The property no_init lets the runtime know that the
    // previous contents of the buffer can be discarded.
    sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
    sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

    h.parallel_for(N, [=](auto i) {
        aA[i] = 11;
        aB[i] = 22;
        aC[i] = 0;
    });
});

// Kernel2
Q.submit([&](auto &h) {
    // Create device sycl::accessors
    sycl::accessor aA(ABuf, h, sycl::read_only);
    sycl::accessor aB(BBuf, h, sycl::read_only);
    sycl::accessor aC(CBuf, h);
    h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
});

// The host accessor creation will ensure that a wait for kernel to finish
// is triggered and data from device to host is copied
sycl::host_accessor h_acc(CBuf);
for (int i = 0; i < N; i++) {
    printf("%d\n", h_acc[i]);
}
return 0;

The following example shows another way to run the same code with different scope blocking. In this case, there
will not be a copy of buffers from host to device at the end of kernel1 and from host to device at the beginning
of kernel2. The copy of all three buffers happens at the end of kernel2 when these buffers go out of scope.
```c
#include <stdio.h>

constexpr int N = 100;

int main() {
    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;

    {
        // Create 3 buffers, each holding N integers
        sycl::buffer<int> ABuf(&AData[0], N);
        sycl::buffer<int> BBuf(&BData[0], N);
        sycl::buffer<int> CBuf(&CData[0], N);

        // Kernel1
        Q.submit([&](auto &h) {
            // Create device accessors.
            // The property no_init lets the runtime know that the
            // previous contents of the buffer can be discarded.
            sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
            sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
            sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(N, [=](auto i) {
                aA[i] = 11;
                aB[i] = 22;
                aC[i] = 0;
            });
        });

        // Kernel2
        Q.submit([&](auto &h) {
            // Create device accessors
            sycl::accessor aA(ABuf, h, sycl::read_only);
            sycl::accessor aB(BBuf, h, sycl::read_only);
            sycl::accessor aC(CBuf, h);

            h.parallel_for(N, [=](auto i) {
                aC[i] += aA[i] + aB[i];
            });
        });

        // Since the buffers are going out of scope, they will have to be
        // copied back from device to host and this will require a wait for
        // all the kernels to finish and so no explicit wait is needed
        for (int i = 0; i < N; i++) {
            printf("%d\n", CData[i]);
        }
    }

    return 0;
}
```
There is another way to write the kernel where a copy of the read-only variable on the host can be accessed on the device as part of variable capture in the lambda function defining the kernel, as shown below. The issue with this is that for every kernel invocation the data associated with vectors AData and BData have to be copied to the device.

**Listing 116: /examples/buffer-accessors/kern4.cpp**

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================================
#include <CL/sycl.hpp>
#include <stdio.h>

constexpr int N = 100;
constexpr int iters = 100;

int main()
{
    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;
    sycl::buffer<int> CBuf(&CData[0], N);

    {
        // Create 2 buffers, each holding N integers
        sycl::buffer<int> ABuf(&AData[0], N);
        sycl::buffer<int> BBuf(&BData[0], N);

        // Kernel1
        Q.submit([&](auto &h) {
        // Create device accessors.
        // The property no_init lets the runtime know that the
        // previous contents of the buffer can be discarded.
            sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
            sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
            sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

            h.parallel_for(N, [=](auto i) {
                aA[i] = 11;
                aB[i] = 22;
                aC[i] = 0;
            });
        });
    }

    for (int it = 0; it < iters; it++) {
        // Kernel2
        Q.submit([&](auto &h) {
            // Create device accessors
            // ...
        });
    }
}
```

(continues on next page)
It is better to use a buffer and a read-only accessor to that buffer so that the vector is copied from host to device only once. In the following kernel, access to memory AData and BData is made through the ABuf and Bbuf on lines 38 and 39 and the declaration in lines 44 and 45 makes them read-only, which prevents them from being copied back to the host from the device when they go out of scope.

Listing 117: /examples/buffer-accessors/kern5.cpp

```cpp
//==========
// Copyright © 2022 Intel Corporation
//
// SPDX-License-Identifier: MIT
//=
#include <CL/sycl.hpp>
#include <stdio.h>
const expr int N = 100;
const expr int iters = 100;

int main() {
    int AData[N];
    int BData[N];
    int CData[N];

    sycl::queue Q;
    sycl::buffer<int> CBuf(&CData[0], N);

    {
        // Create 2 buffers, each holding N integers
        sycl::buffer<int> ABuf(&AData[0], N);
        sycl::buffer<int> BBuf(&BData[0], N);

        // Kernel1
        Q.submit([&](auto &h) {
            // Create device accessors.
            // The property no_init lets the runtime know that the
            // previous contents of the buffer can be discarded.
            sycl::accessor aA(ABuf, h, sycl::write_only, sycl::no_init);
            sycl::accessor aB(BBuf, h, sycl::read_only);

            // Accessors to be used in the kernel.
            sycl::accessor aC(CBuf, h);
            h.parallel_for(N, [=](auto i) { aC[i] += AData[i] + BData[i]; });
        });
    }
    sycl::host_accessor h_acc(CBuf);
    for (int i = 0; i < N; i++) {
        printf("%d\n", h_acc[i]);
    }
    return 0;
}
```
sycl::accessor aB(BBuf, h, sycl::write_only, sycl::no_init);
sycl::accessor aC(CBuf, h, sycl::write_only, sycl::no_init);

h.parallel_for(N, [=](auto i) {
aA[i] = 11;
aB[i] = 22;
aC[i] = 0;
});

sycl::buffer<int> ABuf(&AData[0], N);
sycl::buffer<int> BBuf(&BData[0], N);
for (int it = 0; it < iters; it++) {
    // Kernel2
    Q.submit([&](auto &h) {
        // Create device accessors
        sycl::accessor aA(ABuf, h, sycl::read_only);
sycl::accessor aB(BBuf, h, sycl::read_only);
        sycl::accessor aC(CBuf, h);
        h.parallel_for(N, [=](auto i) { aC[i] += aA[i] + aB[i]; });
    });
}

sycl::host_accessor h_acc(CBuf);
for (int i = 0; i < N; i++) {
    printf("%d\n", h_acc[i]);
}

return 0;
10.0 Host/Device Coordination

Significant computation and communication resources exist between the host and accelerator devices, and care must be taken to ensure that they are effectively utilized.

In this section, we cover topics related to the coordination of host and accelerator processing.

10.1 Asynchronous and Overlapping Data Transfers Between Host and Device

An accelerator is a separate device from the host CPU and is attached with some form of bus, like PCIe* or CXL*. This bus, depending on its type, has a certain bandwidth through which the host and devices can transfer data. An accelerator needs some data from host to do computation, and overall performance of the system is dependent on how quickly this transfer can happen.

10.1.1 Bandwidth Between Host and Accelerator

Most current accelerators are connected to the host system through PCIe. Different generations of PCIe have increased the bandwidth over time, as shown in the table below.

<table>
<thead>
<tr>
<th>PCIe Version</th>
<th>Transfer Rate</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.5 GT/s</td>
<td>0.250 GB/s</td>
</tr>
<tr>
<td>2.0</td>
<td>5.0 GT/s</td>
<td>0.500 GB/s</td>
</tr>
<tr>
<td>3.0</td>
<td>8.0 GT/s</td>
<td>0.985 GB/s</td>
</tr>
<tr>
<td>4.0</td>
<td>16.0 GT/s</td>
<td>1.969 GB/s</td>
</tr>
<tr>
<td>5.0</td>
<td>32.0 GT/s</td>
<td>3.938 GB/s</td>
</tr>
</tbody>
</table>

The local memory bandwidth of an accelerator is an order of magnitude higher than host-to-device bandwidth over a link like PCIe. For instance, HBM (High Bandwidth Memory) on modern GPUs can reach up to 900 GB/sec of bandwidth compared to an x16 PCIe, which can get 63 GB/s. So it is imperative to keep data in local memory and avoid data transfer from host to device or device to host as much as possible. This means that it is better to execute all the kernels on the accelerator to avoid data movement between accelerators or between host and accelerator even it means some kernels are not very efficiently executed on these accelerators.

Any intermediate data structures should be created and used on the device, as opposed to creating them on the host and moving them back and forth between host and accelerator. This is illustrated by the kernels shown here for reduction operations, where the intermediate results are created only on the device and never on the host. In kernel ComputeParallel1, a temporary accumulator is created on the host and all work-items put their intermediate results in it. This accumulator is brought back to the host and then further reduced (at line 37).
Listing 118: /examples/overlap-data-transfers/reduction.cpp

```cpp
float ComputeParallel1(sycl::queue &q, std::vector<float> &data) {
    const size_t data_size = data.size();
    float sum = 0;
    static float *accum = 0;

    if (data_size > 0) {
        const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
        int num_EUs =
            q.get_device().get_info<sycl::info::device::max_compute_units>();
        int vec_size =
            q.get_device()
            .get_info<sycl::info::device::native_vector_width_float>();
        int num_processing_elements = num_EUs * vec_size;
        int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
        sycl::buffer<float> buf(data.data(), data.size(), props);
        sycl::buffer<float> accum_buf(accum, num_processing_elements, props);

        if (!accum)
            accum = new float[num_processing_elements];

        q.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(num_processing_elements, [=](auto index) {
                size_t glob_id = index[0];
                size_t start = glob_id * BATCH;
                size_t end = (glob_id + 1) * BATCH;
                if (end > N)
                    end = N;
                float sum = 0.0;
                for (size_t i = start; i < end; i++)
                    sum += buf_acc[i];
                accum_acc[glob_id] = sum;
            });
        });
        q.wait();
        sycl::host_accessor h_acc(accum_buf);
        for (int i = 0; i < num_processing_elements; i++)
            sum += h_acc[i];
    }
    return sum;
}
```

An alternative approach is to keep this temporary accumulator on the accelerator and launch another kernel with only one work-item, which will perform this final reduction operation on the device as shown in the following `ComputeParallel2` kernel on line 36. Note that this kernel does not have much parallelism and so it is executed by just one work-item. On some platforms this might be better than transferring the data back to the host and doing the reduction there.
Listing 119: /examples/overlap-data-transfers/reduction.cpp

```cpp
float ComputeParallel2(sycl::queue &q, std::vector<float> &data) {
    const size_t data_size = data.size();
    float sum = 0;
    static float *accum = 0;

    if (data_size > 0) {
        const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
        int num_EUs = q.get_device().get_info<sycl::info::device::max_compute_units>();
        int vec_size = q.get_device().
            .get_info<sycl::info::device::native_vector_width_float>();
        int num_processing_elements = num_EUs * vec_size;
        int BATCH = (N + num_processing_elements - 1) / num_processing_elements;
        sycl::buffer<float> buf(data.data(), data.size(), props);
        sycl::buffer<float> accum_buf(accum, num_processing_elements, props);
        sycl::buffer<float> res_buf(&sum, 1, props);
        if (!accum)
            accum = new float[num_processing_elements];

        q.submit([&](auto &h) {
            sycl::accessor buf_acc(buf, h, sycl::read_only);
            sycl::accessor accum_acc(accum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(num_processing_elements, [=](auto index) {
                size_t glob_id = index[0];
                size_t start = glob_id * BATCH;
                size_t end = (glob_id + 1) * BATCH;
                if (end > N)
                    end = N;
                float sum = 0.0;
                for (size_t i = start; i < end; i++)
                    sum += buf_acc[i];
                accum_acc[glob_id] = sum;
            });
        });

        q.submit([&](auto &h) {
            sycl::accessor accum_acc(accum, h, sycl::read_only);
            sycl::accessor res_acc(res_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(1, [=](auto index) {
                res_acc[index] = 0;
                for (size_t i = 0; i < num_processing_elements; i++)
                    res_acc[index] += accum_acc[i];
            });
        });
    } // Buffers go out of scope and data gets transferred from device to host
    return sum;
} // end ComputeParallel2
```
10.1.2 Overlapping Data Transfer from Host to Device with Computation on Device

Some GPUs provide specialized engines for copying data from host to device. Effective utilization of them will ensure that the host-to-device data transfer can be overlapped with execution on the device. In the following example, a block of memory is divided into chunks and each chunk is transferred to the accelerator (line 57), processed (line 60), and the result (line 63) is brought back to the host. These chunks of three tasks are independent, so they can be processed in parallel depending on availability of hardware resources. In systems where there are copy engines that can be used to transfer data between host and device, we can see that the operations from different loop iterations can execute in parallel. The parallel execution can manifest in two ways:

- Between two memory copies, where one is executed by the GPU EUs and one by a copy engine, or both are executed by copy engines.
- Between a memory copy and a compute kernel, where the memory copy is executed by the copy engine and the compute kernel by the GPU EUs.

Listing 120: /examples/overlap-data-transfers/overlap.cpp

```cpp
#include <CL/sycl.hpp>

#define NITERS 10
#define KERNEL_ITERS 10000
#define NUM_CHUNKS 10
#define CHUNK_SIZE 1000000

class Timer {
public:
  Timer() : start_(std::chrono::steady_clock::now()) {}

  double Elapsed() {
    auto now = std::chrono::steady_clock::now();
    return std::chrono::duration_cast<Duration>(now - start_).count();
  }

private:
  using Duration = std::chrono::duration<double>;
  std::chrono::steady_clock::time_point start_;
};

int main() {
  const int num_chunks = NUM_CHUNKS;
  const int chunk_size = CHUNK_SIZE;
  const int iter = NITERS;

  sycl::queue q;

  // Allocate and initialize host data
```

(continues on next page)
float *host_data[num_chunks];
for (int c = 0; c < num_chunks; c++) {
    host_data[c] = sycl::malloc_host<float>(chunk_size, q);
    float val = c;
    for (int i = 0; i < chunk_size; i++)
        host_data[c][i] = val;
}  
std::cout << "Allocated host data"
;  
// Allocate and initialize device memory
float *device_data[num_chunks];
for (int c = 0; c < num_chunks; c++) {
    device_data[c] = sycl::malloc_device<float>(chunk_size, q);
    float val = 1000.0;
    q.fill<float>(device_data[c], val, chunk_size);
}
q.wait();
std::cout << "Allocated device data"
;  
Timer timer;
for (int it = 0; it < iter; it++) {
    for (int c = 0; c < num_chunks; c++) {
        auto add_one = [=](auto id) {
            for (int i = 0; i < KERNEL_ITERS; i++)
                device_data[c][id] += 1.0;
        };
        // Copy-in not dependent on previous event
        auto copy_in =
            q.memcpy(device_data[c], host_data[c], sizeof(float) * chunk_size);
        // Compute waits for copy_in
        auto compute = q.parallel_for(chunk_size, copy_in, add_one);
        auto cg = [=](auto &h) {
            h.depends_on(compute);
            h.memcpy(host_data[c], device_data[c], sizeof(float) * chunk_size);
        };
        // Copy out waits for compute
        auto copy_out = q.submit(cg);
    }
    q.wait();
}
auto elapsed = timer.Elapsed() / iter;
for (int c = 0; c < num_chunks; c++) {
    for (int i = 0; i < chunk_size; i++) {
        if (host_data[c][i] != (float)((c + KERNEL_ITERS * iter))) {
            std::cout << "Mismatch for chunk: " << c << " position: " << i
                << " expected: " << c + 10000 << " got: " << host_data[c][i]
                << "\n";
            break;
        }
    }
}  

In the timeline picture below, which is collected using `ze_tracer`, we can see that copy-ins from upcoming iterations overlap with the execution of compute kernel. Also, we see multiple copy-ins executing in parallel on multiple copy engines.

**Fig. 25**: `ze_tracer` plot showing copy-in overlap with execution of compute kernel

In the example above, we cannot have two kernels (even though they are independent) executing concurrently because we only have one GPU. (It is possible to partition the GPU into smaller chunks and execute different kernels concurrently on them.)
11.0 Using Multiple Heterogeneous Devices

Most accelerators reside in a server that has a significant amount of compute resources in it. For instance, a typical server can have up to eight sockets, with each socket containing over 50 cores. DPC++ provides the ability to treat the CPUs and the accelerators uniformly to distribute work among them. It is the responsibility of the programmer to ensure a balanced distribution of work among the heterogeneous compute resources in the platform.

11.1 Overlapping Compute on Various Accelerators in the Platform

DPC++ provides access to different kinds of devices through abstraction of device selectors. Queues can be created for each of the devices, and kernels can be submitted to them for execution. All kernel submits in DPC++ are non-blocking, which means that once the kernel is submitted to a queue for execution, the host does not wait for it to finish unless waiting on the queue is explicitly requested. This allows the host to do some work itself or initiate work on other devices while the kernel is executing on the accelerator.

The host CPU can be treated as an accelerator and the DPCPP can submit kernels to it for execution. This is completely independent and orthogonal to the job done by the host to orchestrate the kernel submission and creation. The underlying operating system manages the kernels submitted to the CPU accelerator as another process and uses the same openCL/Level0 runtime mechanisms to exchange information with the host device.

The following example shows a simple vector add operation that works on a single GPU device.

Listing 121: /examples/multiple-devices/overlap.cpp

```c
size_t VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b, IntArray &sum, int iter) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    auto start = std::chrono::steady_clock::now();
    for (int i = 0; i < iter; i++) {
        auto e = q.submit([&](auto &h) {
            // Input accessor
            sycl::accessor a_acc(a_buf, h, sycl::read_only);
            sycl::accessor b_acc(b_buf, h, sycl::read_only);
            // Output accessor
            sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
            h.parallel_for(num_items,
                [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
        });
    }
    q.wait();
}
```

(continues on next page)
In the following kernel the input vector is split into two parts and computation is done on two different accelerators (one CPU and one GPU) that can execute concurrently. Care must be taken to ensure that the kernels, in addition to being submitted, are actually launched on the devices to get this parallelism. The actual time that a kernel is launched can be substantially later than when it was submitted by the host. The implementation decides the time to launch the kernels based on some heuristics to maximize metrics like utilization, throughput, or latency. For instance, in the case of the OpenCL backend, on certain platforms one needs to explicitly issue a `clFlush` (as shown on line 41) on the queue to launch the kernels on the accelerators.

**Listing 122: /examples/multiple-devices/overlap.cpp**

```cpp
size_t VectorAdd2(sycl::queue &q1, sycl::queue &q2, const IntArray &a, const IntArray &b, IntArray &sum, int iter) {
    sycl::range num_items{a.size() / 2};

    auto start = std::chrono::steady_clock::now();
    {
        sycl::buffer a1_buf(a.data(), num_items);
        sycl::buffer b1_buf(b.data(), num_items);
        sycl::buffer sum1_buf(sum.data(), num_items);

        sycl::buffer a2_buf(a.data() + a.size() / 2, num_items);
        sycl::buffer b2_buf(b.data() + a.size() / 2, num_items);
        sycl::buffer sum2_buf(sum.data() + a.size() / 2, num_items);
        for (int i = 0; i < iter; i++) {
            q1.submit([&](sycl::handler &h) {
                // Input accessors
                sycl::accessor a_acc(a1_buf, h, sycl::read_only);
                sycl::accessor b_acc(b1_buf, h, sycl::read_only);
                // Output accessor
                sycl::accessor sum_acc(sum1_buf, h, sycl::write_only, sycl::no_init);

                h.parallel_for(num_items,
                    [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
            });
            // do the work on host
            q2.submit([&](sycl::handler &h) {
                // Input accessors
                sycl::accessor a_acc(a2_buf, h, sycl::read_only);
                sycl::accessor b_acc(b2_buf, h, sycl::read_only);
                // Output accessor
                sycl::accessor sum_acc(sum2_buf, h, sycl::write_only, sycl::no_init);

                h.parallel_for(num_items,
                    [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
            });
        }
    }
}
```

(continues on next page)
Checking the running time of the above two kernels, it can be seen that the application runs almost twice as fast as before since it has more hardware resources dedicated to solving the problem. In order to achieve good balance, you will have to split the work in proportion to the capability of the accelerator, instead of distributing it evenly as was done in the above example.
12.0 Compilation

oneAPI has multiple types of compilation. The main source to the application is compiled, and the offloaded kernels are compiled. For the kernels, this might be Ahead-Of-Time (AOT) or Just-In-Time (JIT).

In this section we cover topics related to this compilation and how it can impact the efficiency of the execution.

12.1 Just-In-Time Compilation in DPC++

The Intel® oneAPI DPC++ Compiler converts a DPC++ program into an intermediate language called SPIR-V and stores that in the binary produced by the compilation process. The advantage of producing this intermediate file instead of the binary is that this code can be run on any hardware platform by translating the SPIR-V code into the assembly code of the platform at runtime. This process of translating the intermediate code present in the binary is called JIT compilation (Just-In-Time compilation). JIT compilation can happen on demand at runtime. There are multiple ways in which this JIT compilation can be controlled. By default, all the SPIR-V code present in the binary is translated upfront at the beginning of the execution of the first offloaded kernel.

Listing 123: /examples/jitting/jit.cpp

```cpp
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================================
#include <CL/sycl.hpp>
#include <array>
#include <chrono>
#include <iostream>

// Array type and data size for this example.
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;

global void VectorAdd1(sycl::queue &q, const IntArray &a, const IntArray &b, IntArray &sum) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);

    auto e = q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
    });
}
```
h.parallel_for(num_items,
    [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
q.wait();
}

void VectorAdd2(sycl::queue &q, const IntArray &a, const IntArray &b,
    IntArray &sum) {
    sycl::range num_items{a.size()};
    sycl::buffer a_buf(a);
    sycl::buffer b_buf(b);
    sycl::buffer sum_buf(sum.data(), num_items);
    auto e = q.submit([&](auto &h) {
        // Input accessors
        sycl::accessor a_acc(a_buf, h, sycl::read_only);
        sycl::accessor b_acc(b_buf, h, sycl::read_only);
        // Output accessor
        sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::no_init);
        h.parallel_for(num_items,
            [=](auto i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
    });
    q.wait();
}

void InitializeArray(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++)
        a[i] = i;
}

int main() {
    sycl::default_selector d_selector;
    IntArray a, b, sum;
    InitializeArray(a);
    InitializeArray(b);
    sycl::queue q(d_selector, sycl::property::queue::enable_profiling{});
    std::cout << "Running on device: "
        << q.get_device().get_info<sycl::info::device::name>() << "\n";
    std::cout << "Vector size: " << a.size() << "\n";
    auto start = std::chrono::steady_clock::now();
    VectorAdd1(q, a, b, sum);
    auto end = std::chrono::steady_clock::now();
    std::cout << "Initial Vector add1 successfully completed on device - took "
        << (end - start).count() << " nano-secs\n";
    (continues on next page)
When the program above is compiled using the command below (assuming that the name of the source file is example.cpp):

```
dpcpp -O3 -o example example.cpp
```

and run, the output generated will show that the first call to VectorAdd1 takes much longer than the calls to other kernels in the program due to the cost of JIT compilation, which gets invoked when vectorAdd1 is executed for the first time.

The overhead of JIT compilation at runtime can be avoided by Ahead-Of-Time (AOT) compilation (it is enabled by appropriate switches on the compile-line). With AOT compile, the binary will contain the actual assembly code of the platform that was selected during compilation instead of the SPIR-V intermediate code. The advantage is that we do not need to JIT compile the code from SPIR-V to assembly during execution, which makes the code run faster. The disadvantage is that now the code cannot run anywhere other than the platform for which it was compiled.

The example above can be compiled on a Gen9 GPU using the following command with AOT code-generation:

```
dpcpp -O3 -o example example.cpp -fsycl-targets=spir64_gen-unknown-unknown-sycldevice -Xsycl --target-backend=spir64_gen-unknown-unknown-sycldevice -device skl
```

When this compiled program is run, it can be seen from the output that the time it takes to execute all the calls to the kernels takes the same amount of time, unlike before where the first kernel takes a lot more time because of JIT compilation.

If the application contains multiple kernels, one can force eager JIT compilation or lazy JIT compilation using compile-time switches. Eager JIT compilation will invoke the JITTter on all the kernels in the binary at the beginning of execution, while lazy JIT compilation will enable the JITTter only when the kernel is actually called during execution. In situations where certain kernels are not called, this has the advantage of not translating code that
is never actually executed, which avoids unnecessary JIT compilation. This mode can be enabled during compilation using the following option:

```bash
-fsycl-device-code-split=<value>
```

where `<value>` is

- **per_kernel**: generates code to do JIT compilation of a kernel only when it is called
- **per_source**: generates code to do JIT compilation of all kernels in the source file when any of the kernels in the source file are called
- **off**: performs eager JIT compilation of all kernels in the application
- **auto**: the default, the compiler will use its heuristic to select the best way of splitting device code for JIT compilation

If the above program is compiled with this option:

```bash
dpcpp -O3 -o example vec1.cpp vec2.cpp main.cpp -fsycl-device-code-split=per_kernel
```

and run, then from the timings of the kernel executions it can be seen that the first invocations of `VectorAdd1` and `VectorAdd2` take longer, while the second invocations will take less time because they do not pay the cost of JIT compilation.

In the example above, we can put `VectorAdd1` and `VectorAdd2` in separate files and compile them with and without the **per_source** option to see the impact on the execution times of the kernels. When compiled with

```bash
dpcpp -O3 -o example vec1.cpp vec2.cpp main.cpp -fsycl-device-code-split=per_source
```

and run, the execution times of the kernels will show that the JIT compilation cost is paid at the first kernel invocation, while the subsequent kernel invocations do not pay the cost of JIT compilation. But when the program is compiled with

```bash
dpcpp -O3 -o example vec1.cpp vec2.cpp main.cpp
```

and run, the execution times of the kernels will show that the JIT compilation cost is paid upfront at the first invocation of the kernel, and all subsequent kernels do not pay the cost of JIT compilation.

### 12.2 Specialization Constants

DPC++ has a feature called **specialization constants** that can explicitly trigger JIT compilation to generate code from the intermediate SPIR-V code based on the run-time values of these specialization constants. These JIT compilation actions are done during the execution of the program when the values of these constants are known. This is different from the JIT compilation, which is triggered based on the options provided to `-fsycl-device-code-split`.

In the example below, the call to `set_specialization_constant` binds the value returned by the call to function `get_value`, defined on line 10, to the SYCL kernel bundle. When the kernel bundle is initially compiled, this value is not known and so cannot be used for optimizations. At runtime, after function `get_value` is executed, the value is known, so it is used by command groups handler to trigger JIT compilation of the specialized kernel with this value.
Listing 124: /examples/jitting/spec-const1.cpp

//==============================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================
#include <CL/sycl.hpp>
#include <vector>

class specialized_kernel;

// const static identifier of specialization constant
const static sycl::specialization_id<float> value_id;

// Fetch a value at runtime.
float get_value() { return 10; };

int main() {
    sycl::queue queue;
    std::vector<float> vec(1);
    {
        sycl::buffer<float> buffer(vec.data(), vec.size());
        queue.submit([&](auto &cgh) {
            sycl::accessor acc(buffer, cgh, sycl::write_only, sycl::no_init);

            // Set value of specialization constant.
            cgh.template set_specialization_constant<value_id>(get_value());

            // Runtime builds the kernel with specialization constant
            // replaced by the literal value provided in the preceding
            // call of 'set_specialization_constant<value_id>'
            cgh.template single_task<specialized_kernel>(
                [=](sycl::kernel_handler kh) {
                    const float val = kh.get_specialization_constant<value_id>();
                    acc[0] = val;
                });
        });
    }
    queue.wait_and_throw();
    std::cout << vec[0] << std::endl;
    return 0;
}

The specialized kernel at line 24 will eventually become the code shown below:

cgh.single_task<specialized_kernel>(
    [=]() { acc[0] = 10; });

This JIT compilation also has an impact on the amount of time it takes to execute a kernel. This is illustrated by
the example below:

```
//==============================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================
#include <CL/sycl.hpp>
#include <chrono>
#include <vector>

class specialized_kernel;
class literal_kernel;

// const static identifier of specialization constant
const static sycl::specialization_id<float> value_id;

// Fetch a value at runtime.
float get_value() { return 10; };

int main() {
  sycl::queue queue;

  // Get kernel ID from kernel class qualifier
  sycl::kernel_id specialized_kernel_id =
      sycl::get_kernel_id<specialized_kernel>();

  // Construct kernel bundle with only specialized_kernel in the input state
  sycl::kernel_bundle kb_src =
      sycl::get_kernel_bundle<sycl::bundle_state::input>(
          queue.get_context(), {specialized_kernel_id});

  // set specialization constant value
  kb_src.set_specialization_constant<value_id>(get_value());

  auto start = std::chrono::steady_clock::now();
  // build the kernel bundle for the set value
  sycl::kernel_bundle kb_exe = sycl::build(kb_src);
  auto end = std::chrono::steady_clock::now();
  std::cout << "specialization took - " << (end - start).count() << " nano-secs\n";

  std::vector<float> vec{0, 0, 0, 0, 0};
  sycl::buffer<float> buffer1(vec.data(), vec.size());
  sycl::buffer<float> buffer2(vec.data(), vec.size());
  start = std::chrono::steady_clock::now();
  {
    queue.submit([&](auto &cgh) {
      sycl::accessor acc(buffer1, cgh, sycl::write_only, sycl::no_init);

      // use the precompiled kernel bundle in the executable state
      cgh.use_kernel_bundle(kb_exe);
    });
  }
  end = std::chrono::steady_clock::now();
  std::cout << "execute took - " << (end - start).count() << " nano-secs\n";

  std::vector<float> vec2{0, 0, 0, 0, 0};
  sycl::buffer<float> buffer3(vec2.data(), vec2.size());
  sycl::buffer<float> buffer4(vec2.data(), vec2.size());
  start = std::chrono::steady_clock::now();
  {
    queue.submit([&](auto &cgh) {
      sycl::accessor acc(buffer3, cgh, sycl::read_only, sycl::no_init);

      // use the precompiled kernel bundle in the executable state
      cgh.use_kernel_bundle(kb_exe);
    });
  }
  end = std::chrono::steady_clock::now();
  std::cout << "execute took - " << (end - start).count() << " nano-secs\n";

  std::vector<float> vec3{0, 0, 0, 0, 0};
  sycl::buffer<float> buffer5(vec3.data(), vec3.size());
  sycl::buffer<float> buffer6(vec3.data(), vec3.size());
  start = std::chrono::steady_clock::now();
  {
    queue.submit([&](auto &cgh) {
      sycl::accessor acc(buffer5, cgh, sycl::read_only, sycl::no_init);

      // use the precompiled kernel bundle in the executable state
      cgh.use_kernel_bundle(kb_exe);
    });
  }
  end = std::chrono::steady_clock::now();
  std::cout << "execute took - " << (end - start).count() << " nano-secs\n";
```

(continues on next page)
Looking at the runtimes reported by each of the timing messages, it can be seen that the initial translation of the kernel takes a long time, while the actual execution of the JIT-compiled kernel takes less time. The same kernel which had not been precompiled to the executable state takes longer because this kernel will have been JIT-compiled by the runtime before actually executing it.
13.0 Optimizing Media Pipelines

Media operations are ideal candidates for hardware acceleration because they are relatively large algorithms with well-defined inputs and outputs. Video processing hardware capabilities can be accessed via industry-standard frameworks, oneVPL, or low-level/operating system specific approaches like Video Acceleration API (VA-API) for Linux or Microsoft® DirectX® for Windows. Which path to choose depends on many factors. However, the basic principles like parallelization by multiple streams and maximizing data locality apply for all options.

The main differences between video processing and GPGPU work apply to all accelerator API options. Many typical GPGPU optimizations focus on optimizing how large grids of work are partitioned across multiple processing units. Hardware-accelerated media operations are implemented in silicon. They work in units of frames and usually work is partitioned by streams of frames.

Media optimization steps don’t match the GPGPU workflow described in other sections. However, they can be easily added before or after GPGPU work. Media steps will supply inputs to or take outputs from GPGPU steps. For example:

13.1 Media Engine Hardware

As described in Architecture section, Xe-Intel® Data Center GPU Flex Series and some other Intel GPUs contain media engine which provide fully-accelerated video decode, encode and processing capabilities. This is sometimes called Intel® Quick Sync Video. The media engine runs completely independent of compute engines (vector and matrix engines).
Several components can be used by applications:

- **MFX/Multi-format codec**: hardware decode and encode. Some configurations include two forms of encode. 1) motion estimation + bit packing and 2) full fixed function/low power
- **SFC/scaler and format conversion**: resize (primarily intended for downscaling), conversion between color formats such as NV12 and BGRA
- **Video Quality Engine**: multiple frame processing operations, such as denoise and deinterlace.

This hardware has its own instruction queue and clock, so fully fixed function work can be very low power if configured to use low power pathways. This can also leave the slice capabilities on the GPU free for other work.

### 13.1.1 Supported codecs

New codec capabilities are added with each new GPU hardware generation.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Media SDK GPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5th Generation Intel® Core™ (BOW)</td>
<td>D/E</td>
<td>D</td>
<td>D/I</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>6th Generation Intel® Core™ (Skylake)</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>Intel® Atom™ Processor E3900 series (APIL)</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
</tr>
<tr>
<td>6th Generation Intel® Core™ (Kaby Lake)</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
</tr>
<tr>
<td>7th Generation Intel® Core™ (Kaby Lake)</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
</tr>
<tr>
<td>Intel® Atom™ Processor X Series (E3L)</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
</tr>
<tr>
<td>oneVPL GPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel® Atom™ X5 (TGL/TLK/ADL)</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
</tr>
<tr>
<td>Intel® Atom™ X7-MAX (DXI)</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
</tr>
<tr>
<td>Intel® ARC</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
<td>D/E</td>
</tr>
</tbody>
</table>

Note: in this table two kinds of encode are represented.
Intel® Arc A-series and Intel® Server GPU (previously known as ATS-M) add AV1 encode. This cutting edge successor to VP9 adds additional encode control for tile, segmentation, film grain filtering, and other new features. These increase encode quality at a given bitrate or allow a decrease in bitrate to provide increased quality.

### 13.2 Media API Options for Hardware Acceleration

There are multiple ways to accelerate video processing on Intel® architecture (CPUs, GPUs). To choose the option that benefits you most, ensure your goals align with the tools you choose.
As shown above there are higher-level tools and lower-level tools. Do you need the extremely low-level control you can get with operating system specific tools like libva® or DirectX®? And do you have the extra time it takes to develop these low-level applications? Or is it more important to be able to easily port your code from Linux® to Windows® and save time by coding with higher level tools?

More details to help match the approach option to requirements are in the table below.

<table>
<thead>
<tr>
<th></th>
<th>oneVPL</th>
<th>Media frameworks (FFmpeg &amp; GStreamer)</th>
<th>Low-level/OS-specific solutions (Libva &amp; DXVA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>Elementary video stream processing with a limited set of frame processing operations</td>
<td>Full stack (network protocols, container support, audio support)</td>
<td>Working directly with the OS graphics stack</td>
</tr>
<tr>
<td>Level of control over hardware capabilities</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Portability</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

### 13.3 Media Pipeline Parallelism

For GPGPU, parallelism focuses on concerns like how the ND range is partitioned and related edge conditions. Multiple accelerators can work on this partitioned space, executing the same algorithm over the entire grid (SIMD). This is not the case for encode/decode.

Instead of analyzing the internal implementation details of an encoder or decoder to find opportunities for parallelism as it processes each frame, in most cases the entire operation would be treated as a black box. Decode
implementations for a codec are intended to be interchangeable, like substituting one box for another. Encode replacement is more complex, since effects of a broader range of parameters must be considered. However, the strategy is usually the same - swap the entire optimization to one best suited to the hardware instead of attempting to optimize hotspots/inner loops.

In theory, operations could parallelize by slice within frames:

<table>
<thead>
<tr>
<th>Slice0</th>
<th>Slice1</th>
<th>Slice2</th>
<th>Slice3</th>
<th>Slice4</th>
</tr>
</thead>
</table>

This is usually not practical. Since motion search cannot “see” across slice boundaries, overall compression quality is affected as the number of slices increase. Additional header bytes are required for slices as well.

Single streams can be processed asynchronously, but this is also not scalable. Dependencies between frames prevent parallelism. Turning off these dependencies reduces quality at a given bitrate. Increasing the number of frames in flight also increases latency.

<table>
<thead>
<tr>
<th>Frame0</th>
<th>Frame1</th>
<th>Frame2</th>
<th>...</th>
<th>FrameN</th>
</tr>
</thead>
</table>

For single stream optimization, Deep Link hyper encode may simplify development. Deep Link hyper encode can provide a performance boost when one or more discrete GPUs are available on a system where integrated/processor graphics is also available by automatically coordinating work between integrated and discrete GPUs. Single stream performance can be improved by utilizing the capabilities of dGPU and iGPU together.
The best way to scale efficiently while preserving quality and reducing latency is to process multiple streams simultaneously. (Note: for non-realtime processing even a single stream can be processed in parallel as segments since frames will not have dependencies across segment/GOP boundaries.)

Stream0

Stream1

Stream2

...

StreamN

This approach provides ideal “embarrassing” parallelism which scales across accelerators. There are no dependencies across streams, so each accelerator can process as quickly as possible without coordination. For the Hyper Encode case, it is usually faster to schedule separate streams on iGPU and dGPU.

From a oneAPI perspective, these properties greatly simplify interoperability with SYCL. Media operations generally will not run “inside” kernels, which means there are fewer concerns at the API or development level. Media operations will either provide data for a kernel (act as a source), or they will work as a sink on data provided by a
The main concern for performance is that the handoff between media operation and kernel implies synchronization and reduces opportunities to process asynchronously within a single stream. Processing multiple streams concurrently is the best workaround for this limitation.

### 13.3.1 Optimizing Media Operations

Since the algorithms are implemented in hardware, the main concerns with media development are data locality, synchronization, and providing a pipeline of work to keep the hardware busy.

**Data locality:** keep frames on the GPU, avoiding copying to the CPU unnecessarily. Since the media engine is connected to the GPU memory hierarchy, data can be shared locally between slice and media engine components. From a GPGPU perspective these operations work on local GPU data. Frames can be shared between this hardware and execution units with low latency/zero copy. This is especially important for discrete GPUs, since moving raw frames across a PCI bus can be expensive.

**Synchronization:** Because the multiple hardware units can function independently, they can work asynchronously. For best performance, the application should force synchronization with CPU as infrequently as possible. Design algorithms so that the accelerator can proceed as far as it can without interrupt.

**Keeping the hardware busy:** If the instruction queue is not kept full, the engine clock will go down. It can take a few milliseconds to ramp up to full clock speed again.

### 13.4 Media Pipeline Inter-operation and Memory Sharing

Media engine capabilities exposed in low-level OS-specific interfaces, such as

- VA-API (Video Acceleration API) for Linux OS
- Microsoft DirectX® Video Acceleration for Windows OS

as well as various high-level media frameworks built on top of low-level interfaces, such as

- oneVPL
- FFmpeg and libav
- Gstreamer

Each media framework defines own interfaces for device and context creation, memory allocation and task submission. Most frameworks also expose export/import interfaces to convert memory objects to/from other memory handles.

- high-level media frameworks (FFMpeg, Gstreamer) support conversion to/from low-level media handles (VA-API and DirectX surfaces)
- low-level media interfaces (VA-API, DirectX) support conversion to/ OS-specific general-purpose GPU memory handles such as DMA buffers on Linux and NT handles on Windows
- Level-zero support conversion between DMA buffers / NT handles and USM device pointers
Together these interfaces allow zero-copy memory sharing between media operations submitted via media frameworks and SYCL compute kernels submitted into SYCL queue, assuming the SYCL queue created on same GPU device as media framework and SYCL device uses Level-zero backend (not OpenCL backend).

Despite multiple stages of memory handles conversion (FFmpeg/GStreamer, VA-API/DirectX, DMA/NT, Level-Zero, SYCL), all converted memory handles refer to same physical memory block. Thus writing data into one memory handle makes the data available in all other memory handles, assuming proper synchronization between write and read operations.

Below is reference to interfaces used for zero-copy buffer sharing between media frameworks and SYCL

1. (Linux) VA-API to DMA-BUF
2. (Windows) DirectX to NT-Handle
3. DMA-BUF or NT-Handle to Level-zero

The memory pointer created by Level-zero from DMA-BUF or NT-Handle (#3 above) is USM device pointer only accessible by SYCL kernels running on same GPU device as used for media memory allocation and media operations. This USM pointer is not accessible from host and not accessible from SYCL kernels running on CPU or other XPU devices.

Example in next section demonstrates zero-copy buffer sharing between VA-API and SYCL using interfaces 1 and 3 from list above and synthetic video data (moving rectangle). For more advanced examples with FFmpeg/GStreamer video decode/encode on GPU media engine and SYCL kernels on GPU compute engines please refer to Intel® DL Streamer memory interoperability API (preview) and Intel® DL Streamer samples

13.4.1 VA-API and SYCL memory sharing example

The example

1. allocates shared VA-API surfaces and USM device pointers for NUM_FRAMES frames
2. submits VA-API calls to draw moving rectangle on frames
3. submits SYCL kernels to draw sub-rectangle inside rectangle created by VA-API on step 2
4. synchronize all frames and write RGB data into file

Output frames generated by this example look like picture below
The example supports Linux OS and requires installation of the following additional packages besides oneAPI packages (installation example via `apt` package manager on Ubuntu OS)

```bash
sudo apt install intel-level-zero-gpu level-zero-dev
sudo apt install intel-media-va-driver-non-free libva-dev libva-drm2
```

and requires linkage with Level-zero and VA-API libraries

```bash
dpcpp memory-sharing-with-media.cpp -lze_loader -lva -lva-drm
```

Example execution generates file `output.bgra` which could be directly played by some media players (ex, `ffplay`) or transcoded to compressed video format, for example using the following `ffmpeg` command:

```bash
ffmpeg -f rawvideo -pix_fmt bgra -s 320x240 -i output.bgra output.mp4
```

and then played by any media player, for example

```bash
ffplay output.mp4
```

**Listing 126: /examples/memory-sharing-with-media/memory-sharing-vaapi.cpp**

```cpp
#include <CL/sycl.hpp>

// SYCL oneAPI extension
#include <sycl/ext/oneapi/backend/level_zero.hpp>

// Level-zero
#include <level_zero/ze_api.h>

// VA-API
#include <va/va_drm.h>
#include <va/va_drmcommon.h>

#include <cstdio>
#include <fcntl.h>
#include <unistd.h>
#include <vector>

#define OUTPUT_FILE "output.bgra"
#define VAAPI_DEVICE "/dev/dri/renderD128"
#define FRAME_WIDTH 320
#define FRAME_HEIGHT 240
```

(continues on next page)
```c
#define RECT_WIDTH 160  
#define RECT_HEIGHT 160  
#define RECT_Y (FRAME_HEIGHT - RECT_HEIGHT) / 2  
#define NUM_FRAMES (FRAME_WIDTH - RECT_WIDTH)  
#define VA_FORMAT VA_FOURCC_BGRA  
#define RED 0xffff0000  
#define GREEN 0xff00ff00  
#define BLUE 0xff0000ff  
#define CHECK_STS(_FUNC)  
{  
    auto _sts = _FUNC;  
    if (_sts != 0) {  
        printf("Error %d calling " #_FUNC, (int)_sts);  
        return -1;  
    }  
}  

VASurfaceID alloc_va_surface(VADisplay va_display, int width, int height) {  
    VASurfaceID va_surface;  
    VASurfaceAttrib surface_attrib{};  
    surface_attrib.type = VASurfaceAttribPixelFormat;  
    surface_attrib.flags = VA_SURFACE_ATTRIB_SETTABLE;  
    surface_attrib.value.type = VAGenericValueTypeInteger;  
    surface_attrib.value.value.i = VA_FORMAT;  
    vaCreateSurfaces(va_display, VA_RT_FORMAT_RGB32, width, height, &va_surface,  
    1, &surface_attrib, 1);  
    return va_surface;  
}

int main() {  
    // Create SYCL queue on GPU device and Level-zero backend, and query  
    // Level-zero context and device  
    sycl::queue sycl_queue{sycl::ext::oneapi::filter_selector(  
        "level_zero")}; // { sycl::gpu_selector() }  
    auto ext_level_zero = sycl::backend::ext_oneapi_level_zero;  
    auto ze_context = sycl::get_native/ext_level_zero>(sycl_queue.get_context());  
    auto ze_device = sycl::get_native/ext_level_zero>(sycl_queue.get_device());  
    
    // Create VA-API context (VADisplay)  
    VADisplay va_display = vaGetDisplayDRM(open(VAAPI_DEVICE, O_RDWR));  
    if (!va_display) {  
        printf("Error creating VADisplay on device %s\n", VAAPI_DEVICE);  
        return -1;  
    }  
    int major = 0, minor = 0;  
    CHECK_STS(vaInitialize(va_display, &major, &minor));  
    
    // Create VA-API surfaces  
    (continues on next page)  
```

VASurfaceID surfaces[NUM_FRAMES];
for (int i = 0; i < NUM_FRAMES; i++) {
    surfaces[i] = alloc_va_surface(va_display, FRAME_WIDTH, FRAME_HEIGHT);
}

// Convert each VA-API surface into USM device pointer (zero-copy buffer
// sharing between VA-API and Level-zero)
void *device_ptr[NUM_FRAMES];
size_t stride;
for (int i = 0; i < NUM_FRAMES; i++) {
    // Export DMA-FD from VASurface
    VADRMPrimeSurfaceDescriptor prime_desc{};
    CHECK_STS(vaExportSurfaceHandle(va_display, surfaces[i],
        VA_SURFACE_ATTRIB_MEM_TYPE_DRM_PRIME_2,
        VA_EXPORT_SURFACE_READ_WRITE, &prime_desc));
    auto dma_fd = prime_desc.objects->fd;
    auto dma_size = prime_desc.objects->size;
    stride = prime_desc.layers[0].pitch[0] / sizeof(uint32_t);

    // Import DMA-FD into Level-zero device pointer
    ze_external_memory_import_fd_t import_fd = {
        ZE_STRUCTURE_TYPE_EXTERNAL_MEMORY_IMPORT_FD,
        nullptr, // pNext
        ZE_EXTERNAL_MEMORY_TYPE_FLAG_DMA_BUF, dma_fd};
    ze_device_mem_alloc_desc_t alloc_desc = {
        ZE_STRUCTURE_TYPE_DEVICE_MEM_ALLOC_DESC};
    alloc_desc.pNext = &import_fd;
    CHECK_STS(zeMemAllocDevice(ze_context, &alloc_desc, dma_size, 1, ze_device,
        &device_ptr[i]));

    // Close DMA-FD
    close(dma_fd);
}

// Create VA-API surface with size 1x1 and write GREEN pixel
VASurfaceID surface1x1 = alloc_va_surface(va_display, 1, 1);
VAImage va_image;
void *data = nullptr;
CHECK_STS(vaDeriveImage(va_display, surface1x1, &va_image));
CHECK_STS(vaMapBuffer(va_display, va_image.buf, &data));
*(uint32_t *)data = GREEN;
CHECK_STS(vaUnmapBuffer(va_display, va_image.buf));
CHECK_STS(vaDestroyImage(va_display, va_image.image_id));

// VA-API call to fill background with BLUE color and upscale 1x1 surface into
// moving GREEN rectangle
VACfg config;
VACtx context;
CHECK_STS(vaCreateConfig(va_display, VAProfileNone, VAEntrypointVideoProc, nullptr, 0, &config));
CHECK_STS(vaCreateContext(va_display, va_config_id, 0, 0, VA_PROGRESSIVE,
for (int i = 0; i < NUM_FRAMES; i++) {
    VAProcPipelineParameterBuffer param{};
    param.output_background_color = BLUE;
    param.surface = surface1x1;
    VARectangle output_region = {int16_t(i), RECT_Y, RECT_WIDTH, RECT_HEIGHT};
    param.output_region = &output_region;
    VABufferID param_buf;
    CHECK_STS(vaCreateBuffer(va_display, va_context_id,
        VAProcPipelineParameterBufferType, sizeof(param),
        1, &param, &param_buf));
    CHECK_STS(vaBeginPicture(va_display, va_context_id, surfaces[i]));
    CHECK_STS(vaRenderPicture(va_display, va_context_id, &param_buf, 1));
    CHECK_STS(vaEndPicture(va_display, va_context_id));
    CHECK_STS(vaDestroyBuffer(va_display, param_buf));
}

#if 0
    // Synchronization is optional on Linux OS as i915 KMD driver synchronizes
    // write/read commands submitted from Intel media and compute drivers
    for (int i = 0; i < NUM_FRAMES; i++) {
        CHECK_STS(vaSyncSurface(va_display, surfaces[i]));
    }
#endif

    // Submit SYCL kernels to write RED sub-rectangle inside GREEN rectangle
    std::vector<sycl::event> sycl_events(NUM_FRAMES);
    for (int i = 0; i < NUM_FRAMES; i++) {
        uint32_t *ptr = (uint32_t *)device_ptr[i] +
            (RECT_Y + RECT_HEIGHT / 4) * stride + (i + RECT_WIDTH / 4);
        sycl_events[i] = sycl_queue.parallel_for(
            sycl::range<2>(RECT_HEIGHT / 2, RECT_WIDTH / 2), [=](sycl::id<2> idx) {
                auto y = idx.get(0);
                auto x = idx.get(1);
                ptr[y * stride + x] = RED;
            });
    }

    // Synchronize all SYCL kernels
    sycl::event::wait(sycl_events);

    // Map VA-API surface to system memory and write to file
    FILE *file = fopen(OUTPUT_FILE, "wb");
    if (!file) {
        printf("Error creating file %s\n", OUTPUT_FILE);
        return -1;
    }
    for (int i = 0; i < NUM_FRAMES; i++) {
        CHECK_STS(vaDeriveImage(va_display, surfaces[i], &va_image));
        CHECK_STS(vaMapBuffer(va_display, va_image.buf, &data));
        fwrite(data, 1, FRAME_HEIGHT * FRAME_WIDTH * 4, file);
    }
13.5 DPCPP-Blur Example

For a working oneVPL example which ties several of these concepts together (currently only for a single stream), see dpcpp-blur. This sample shows memory interoperation between video APIs and oneVPL as the frame is input, manipulated and output using the following steps.

- Set up SYCL
- Set up a oneVPL session
- Initialize oneVPL VPP
- Loop through frames
  - Read the frame from a file
  - Run VPP resize/colorspace conversion on the GPU
  - Get access to the GPU surface, convert to USM
  - Run SYCL kernel (blur) on the GPU
  - Output the frame to a file

Find this sample here:
https://github.com/oneapi-src/oneVPL/tree/master/examples/interop/dpcpp-blur

In this example, you can see that the interaction between oneVPL and SYCL is at a frame level. oneVPL provides a frame then the SYCL kernel processes it. For the OS environment where zero-copy capabilities are enabled in L0 (Linux), the libva frame data is made available to SYCL as USM. Instead of copying the libva raw frame to a new USM surface, it is possible for the app to work with the frame on the GPU as a libva surface then start working with the same memory as if it were USM.

To keep this example simple there are many design simplifications which currently limit its ability to fully showcase the benefits of zero copy.

- Raw frames are read from disk and written to disk - this sets the overall frame rate
• VPP data is read in as system memory and converted to video memory
• The pipeline is synchronized at each frame

However, zero copy is the core concept which can be built into a high performance application.

Video streaming is prevalent in our world today. We stream meetings at work. We watch movies at home. We expect good quality. Taking advantage of this new media engine hardware gives you the option to stream faster, stream at higher quality and/or stream at lower power. This hardware solution is an important consideration for End-to-End performance in pipelines working with video data.
14.0 OpenMP Offloading Tuning Guide

Intel® LLVM-based C/C++ and Fortran compilers, icx, icpx, and ifx, support OpenMP offloading onto GPUs. When using OpenMP, the programmer inserts device directives in the code to direct the compiler to offload certain parts of the application onto the GPU. Offloading compute-intensive code can yield better performance.

This section covers various topics related to OpenMP offloading, and how to improve the performance of offloaded code.

14.1 OpenMP Directives

Intel® compilers, icx, icpx, and ifx support various OpenMP directives that control the offloading of computations and mapping of data onto a device. These include:

- target
- teams
- distribute
- target data
- target enter data
- target exit data
- target update
- declare target
- dispatch

The target construct specifies that a specific part of the code is to be executed on the device and how data is to be mapped to the device.

The teams construct creates a number of thread teams, where each team is composed of a master thread and a number of worker threads. If teams is specified without the num_teams clause, then the number of teams is implementation defined.

The distribute construct distributes iterations of a loop among the master threads of the teams, so each master thread executes a subset of the iterations.

The target data construct maps variables to a device data environment. Variables are mapped for the extent of the target data region, according to any map clauses.

The target enter data directive specifies that variables are mapped to a device. With this directive, the map-type specified in map clauses must be either to or alloc.

The target exit data directive specifies that variables are unmapped from the device. With this directive, the map-type specified in map clauses must be from, release, or delete.
The target update directive makes the values of variables on the device consistent with their original host variables, according to the specified motion clauses.

The declare target directive specifies that variables, functions (C, C++ and Fortran), and subroutines (Fortran) are mapped to a device.

The declare variant directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used.

The dispatch construct controls whether variant substitution occurs for a given function call.

The map clause determines how an original host variable is mapped to a corresponding variable on the device. Map-types include:

- to: The value of the original host variable is copied to the device on entry to the target region.
- from: The value of the variable on the device is copied from the device to the original host variable on exit from the target region.
- tofrom: The value of the original host variable is copied to the device on entry to the target region, and copied back to the host on exit from the target region.
- alloc: Allocate an uninitialized copy of the original host variable on the device (values are not copied from the host to the device).

Directives can be combined. For example, the following combined directives may be used:

- target teams
- target teams distribute
- target teams distribute parallel for
- target teams distribute parallel for simd

It is recommended that combined directives be used where possible because they allow the compiler and runtime to decide how to best partition the iterations of an offloaded loop for execution on the GPU.

14.2 OpenMP Execution Model

The OpenMP execution model has a single host device but multiple target devices. A device is a logical execution engine with its own local storage and data environment.

When executing on ATS or PVC, the entire GPU (which is composed of two tiles) can be considered as a device, or each tile can be considered as a device.

OpenMP starts executing on the host. When a host thread encounters a target construct, data is transferred from the host to the device (if specified by map clauses, for example), and code in the construct is offloaded onto the device. At the end of the target region, data is transferred from the device to the host (if so specified).

By default, the host thread that encounters the target construct waits for the target region to finish before proceeding further. nowait on a target construct specifies that the host thread does not need to wait for the target region to finish. In other words, the nowait clause allows the asynchronous execution of the target region.

Synchronizations between regions of the code executing asynchronously can be achieved via the taskwait directive, depend clauses, (implicit or explicit) barriers, or other synchronization mechanisms.
14.3 Terminology

In this chapter, OpenMP and DPC++ terminology is used interchangeably to describe the partitioning of iterations of an offloaded parallel loop.

As described in the “DPC++ Thread Hierarchy and Mapping” chapter, the iterations of a parallel loop (execution range) offloaded onto the GPU are divided into work-groups, sub-groups, and work-items. The ND-range represents the total execution range, which is divided into work-groups of equal size. A work-group is a 1-, 2-, or 3-dimensional set of work-items. Each work-group can be divided into sub-groups. A sub-group represents a short range of consecutive work-items that are processed together as a SIMD vector.

The following table shows how DPC++ concepts map to OpenMP and CUDA concepts.

<table>
<thead>
<tr>
<th>DPC++</th>
<th>OpenMP</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work-item</td>
<td>OpenMP thread or SIMD lane</td>
<td>CUDA thread</td>
</tr>
<tr>
<td>Work-group</td>
<td>Team</td>
<td>Thread block</td>
</tr>
<tr>
<td>Work-group size</td>
<td>Team size</td>
<td>Thread block size</td>
</tr>
<tr>
<td>Number of work-groups</td>
<td>Number of teams</td>
<td>Number of thread blocks</td>
</tr>
<tr>
<td>Sub-group</td>
<td>SIMD chunk (simdlen = 8, 16, 32)</td>
<td>Warp (size = 32)</td>
</tr>
<tr>
<td>Maximum number of work-items per work-group</td>
<td>Thread limit</td>
<td>Maximum number of of CUDA threads per thread block</td>
</tr>
</tbody>
</table>

14.4 Compiling and Running an OpenMP Application

Use the following compiler options to enable OpenMP offload onto Intel® GPUs. These options apply to both C/C++ and Fortran.

```
-fiopenmp -fopenmp-targets=spir64
```

By default the Intel® compiler converts the program into an intermediate language called SPIR-V and stores that in the binary produced by the compilation process. The code can be run on any hardware platform by translating the SPIR-V code into the assembly code of the platform at runtime. This process is called Just-In-Time (JIT) compilation.

To enable the output of the compiler optimization report, add the following options:

```
-qopt-report=3 -O3
```

**Note:**

- The -qopenmp compiler option is equivalent to -fiopenmp, and the two options can be used interchangeably.
14.4.1 Ahead-Of-Time (AOT) Compilation

For Ahead-Of-Time (AOT) compilation for ATS, you need to specify an additional compiler option (-xs), as shown below. This option applies to both C/C++ and Fortran.

```bash
-fiopenmp -fopenmp-targets=spir64_gen -Xs "-device ats"
```

14.4.2 OpenMP Runtime Routines

The following are some device-related runtime routines:

```c
omp_target_alloc
omp_target_free
omp_target_memcpy
```

The following runtime routines are supported by the Intel® compilers as Intel® extensions:

```c
omp_target_alloc_host
omp_target_alloc_device
omp_target_alloc_shared
```

omp_target_free can be called to free up the memory allocated using the above Intel® extensions.

For a listing of OpenMP features supported in the icx, icpx, and ifx compilers, see:

- OpenMP Features and Extensions Supported in Intel® oneAPI DPC++/C++ Compiler
- Fortran Language and OpenMP Features Implemented in Intel® Fortran Compiler (Beta)

14.4.3 Environment Variables

Below are some environment variables that are useful for debugging or improving the performance of programs.

For additional information on environment variables, see:

- Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference - Supported Environment Variables
- Intel® oneAPI Programming Guide - Debug Environment Variables
- LLVM/OpenMP Runtimes
- Debugging Variables for Level Zero Plugin

**LIBOMPTARGET_DEBUG=1**

Enables the display of debugging information from libomptarget.so.

**LIBOMPTARGET_DEVICES=<DeviceKind>**

Controls how sub-devices are exposed to users.
<DeviceKind> ::= DEVICE | SUBDEVICE | SUBSUBDEVICE |
                        device | subdevice | subsubdevice

DEVICE/device: Only top-level devices are reported as OpenMP devices, and subdevice clause is supported.

SUBDEVICE/subdevice: Only 1st-level sub-devices are reported as OpenMP devices, and subdevice clause is ignored.

SUBSUBDEVICE/subsubdevice: Only second-level sub-devices are reported as OpenMP devices, and subdevice clause is ignored. On Intel® GPU using Level Zero backend, limiting the subsubdevice to a single compute slice within a tile also requires setting additional GPU compute runtime environment variable CFESingleSliceDispatchCCSMode=1.

The default is equivalent to <DeviceKind>=device

LIBOMPTARGET_INFO=<Num>

Allows the user to request different types of runtime information from libomptarget. For details, see:
https://openmp.llvm.org/design/Runtimes.html#libomptarget-info

LIBOMPTARGET_LEVEL0_MEMORY_POOL=<Option>

Controls how reusable memory pool is configured.

<Option> ::= 0 | <PoolInfoList>
.PoolInfoList ::= <PoolInfo>[,<PoolInfoList>]
.PoolInfo ::= <MemType>[,<AllocMax>[,<Capacity>[,<PoolSize>]]]
.MemType ::= all | device | host | shared
.AllocMax ::= positive integer or empty, max allocation size in MB
.Capacity ::= positive integer or empty, number of allocations from a single block
.PoolSize ::= positive integer or empty, max pool size in MB

Pool is a list of memory blocks that can serve at least <Capacity> allocations of up to <AllocMax> size from a single block, with total size not exceeding <PoolSize>.

LIBOMPTARGET_LEVEL0_STAGING_BUFFER_SIZE=<Num>

Sets the staging buffer size to <Num> KB. Staging buffer is used to optimize copy operation between host and device when host memory is not Unified Shared Memory (USM). The staging buffer is only used for discrete devices. The default staging buffer size is 16 KB.

LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH=copy

Enables batching of commands for data transfer in a target region.

If there are map(to: ) clauses on a target construct, then this environment variable allows multiple data transfers from the host to the device to occur concurrently. Similarly, if there are map(from: ) clauses on the target construct, this environment variable allows multiple data transfers from the device to the host to occur concurrently. Note that map(tofrom: ) or map( ) would be split into map(to: ) and map(from: ).
LIBOMPTARGET_LEVEL_ZERO_USE_IMMEDIATE_COMMAND_LIST=<Bool>

Enables/disables using immediate command list for kernel submission.

| <Bool> := | 1 | T | t | 0 | F | f |

By default, using immediate command list is disabled.

LIBOMPTARGET_PLUGIN=<Name>

Designates the offload plugin name to use.

| <Name> := | LEVEL0 | OPENCL | X86_64 |
|          | level0 | opencl | x86_64 |

By default, the offload plugin is LEVEL0.

LIBOMPTARGET_PLUGIN_PROFILE=<Enable>[,<Unit>]

Enables basic plugin profiling and displays the result when the program finishes.

| <Enable> := | 1 | T |
| <Unit> := usec | unit_usec |

By default, plugin profiling is disabled.

If <Unit> is not specified, microsecond (usec) is the default unit.

LIBOMPTARGET_PROFILE=<FileName>

Allows libomptarget.so to generate time profile output similar to Clang’s -ftime-trace option.

OMP_TARGET_OFFLOAD=MANDATORY

Specifies that program execution is terminated if a device construct or device memory routine is encountered and the device is not available or is not supported by the implementation.

**Environment Variables to Control Implicit and Explicit Scaling**

To disable implicit scaling and use one GPU tile only, set: ZE_AFFINITY_MASK=0.0

To enable explicit scaling, set: LIBOMPTARGET_DEVICES=subdevice

For PVC, implicit scaling is on by default.
Environment Variables for DPC++

There are several SYCL_PI_LEVEL_ZERO environment variables that are useful for the development and debugging of DPC++ programs (not just OpenMP). They are documented at:

https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md

14.5 Offloading oneMKL Computations onto the GPU

The Intel® oneAPI Math Kernel Library (oneMKL) improves performance with math routines for software applications that solve large computational problems. oneMKL provides BLAS, Sparse BLAS, and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions, random number generation functions, and other functionality.

The oneMKL distribution includes an examples directory which contains examples of various calls to oneMKL routines.

For more information about the Intel oneAPI Math Kernel Library, see:

- Developer Reference for Intel® oneAPI Math Kernel Library - C
- Developer Reference for Intel® oneAPI Math Kernel Library – Fortran
- Introducing Batch GEMM Operations

14.5.1 Compile and Link Commands when Using oneMKL OpenMP Offload

The information given in this section is specific to Linux. For information specific to Windows, and for more details, refer to the Intel® oneAPI Math Kernel Library Link Line Advisor.

Notes:

- The link commands shown below will dynamically link to the oneMKL library.
- The Intel oneMKL LP64 libraries index arrays with the 32-bit integer type; whereas the Intel oneMKL ILP64 libraries use the 64-bit integer type (necessary for indexing large arrays, with more than $2^{31} - 1$ elements).

C/C++ (Linux)

The compile and link commands for a C/C++ program that uses OpenMP threading and calls oneMKL C/C++ API with 32-bit integers are as follows.

Compile:
```
icx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -c source.c
```

Link:
```
icx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fsycl -L${MKLROOT}/lib/intel64 -liomp5 -lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl source.o
```

If the program calls oneMKL C/C++ API with 64-bit integers, the compile and link commands are:
oneAPI GPU Optimization Guide

```bash
Compile:
icx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -DMKL_ILP64 -c source.c
Link:
icx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fsycl -L${MKLROOT}/lib/intel64 -liomp5 -lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl source.o
```

Fortran (Linux)

The compile and link commands for a Fortran program that uses OpenMP threading and calls oneMKL Fortran API with **32-bit integers** are as follows.

```bash
Compile:
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fpp -free -c source.f
Link:
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fsycl -L${MKLROOT}/lib/intel64 -liomp5 -lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl source.o
```

If the program calls oneMKL Fortran API with **64-bit integers**, the compile and link commands are:

```bash
Compile:
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -m64 -DMKL_ILP64 -i8 -fpp -free -c source.f
Link:
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fsycl -L${MKLROOT}/lib/intel64 -liomp5 -lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl source.o
```

### 14.5.2 OpenMP Directives to Offload oneMKL Computations

You can use OpenMP directives to offload oneMKL computations onto the GPU. There are two ways to do this.

One way involves using the Intel-specific OpenMP extension `target variant dispatch` directive. You would place the call to the oneMKL routine inside a target variant dispatch construct, as shown in the example below. In this example, arrays A, B, and C used in the multiplication are mapped to the device before the call to the oneMKL routine `cblas_dgemm`. The `use_device_ptr(A,B,C)` clause is used on the target variant dispatch directive to indicate that A, B, and C point to objects that have corresponding storage on the device. When `cblas_dgemm` is called, the corresponding device pointers for A, B, and C will be passed as arguments, and the device copies of A, B, and C will be used in the computation.

```
Listing 127: /examples/OpenMP/22_mkl_dispatch/dgemm_target_variant_dispatch_c.cpp

//===============================================
// Copyright © 2022 Intel Corporation
//
// SPDX-License-Identifier: MIT
//===============================================
// clang-format off
```
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define EPSILON 0.0001

int main()
{
    int64_t m, n, k;
    double alpha, beta;
    double sum;
    int64_t i, j, q;
    int fail;

    printf ("This example computes real matrix C=alpha*A*B+beta*C using 
    " Intel oneMKL function dgemm, where A, B, and C are matrices and 
    " alpha and beta are double precision scalars\n\n");

    m = 2000, k = 200, n = 1000;
    printf ("Initializing data for matrix multiplication C=A*B for matrix \n    
    A(\%li x \%li) and matrix B(\%li x \%li)\n\n", m, k, k, n);
    alpha = 1.0; beta = 0.0;

    printf ("Allocating memory for matrices aligned on 64-byte boundary for better \n    " performance \n\n");
    A = (double *)mkl_malloc( m * k * sizeof( double ), 64 );
    B = (double *)mkl_malloc( k * n * sizeof( double ), 64 );
    C = (double *)mkl_malloc( m * n * sizeof( double ), 64 );
    C_fl = (double *)mkl_malloc( m*n*sizeof( double ), 64 );

    if (A == NULL || B == NULL || C == NULL || C_fl == NULL) {
        printf("\nERROR: Cannot allocate memory for matrices. Exiting...
\n");
        return 1;
    }

    printf ("Intializing matrices \n\n");
    for (i = 0; i < (m*k); i++) {
        A[i] = (double)(i+1);
    }

    for (i = 0; i < (k*n); i++) {
        B[i] = (double)(-i-1);
    }

    for (i = 0; i < (m*n); i++) {
        C[i] = 0.0;
    }

    (continues on next page)
```
C_fl[i] = 0.0;
}

printf (" Computing matrix product using Intel oneMKL dgemm function via CBLAS interface \n\n -n");

#pragma omp target data map(to: A[0:m*k], B[0:k*n]) map(tofrom: C[0:m*n])
{
  #pragma omp target variant dispatch use_device_ptr(A, B, C)
  {
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    m, n, k, alpha, A, k, B, n, beta, C, n);
  }
}

printf ("\n Top left corner of matrix C: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C[j+i*n]);
    }
    printf ("\n");
}

printf (" Computing matrix product using for-loops \n");
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++) {
            sum += A[k*i+q] * B[n*q+j];
        }
        C_fl[n*i+j] = alpha * sum + beta * C_fl[n*i+j];
    }
}

printf ("\n Top left corner of matrix C_fl: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C_fl[j+i*n]);
    }
    printf ("\n");
}

printf (" Computing matrix product using for-loops \n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (fabs(C[i] - C_fl[i]) > EPSILON) {
        fail = 1;
        break;
    }
}

printf (" Computations completed. Verifying... \n\n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (fabs(C[i] - C_fl[i]) > EPSILON) {
        fail = 1;
        break;
    }
}
if (fail)
    printf ("\n **** FAIL **** \n");
else
    printf ("\n **** PASS **** \n");

printf ("\n Deallocating memory \n\n");
mkl_free(A);
mkl_free(B);
mkl_free(C);

return fail;
}

Another way to inform the compiler that oneMKL computations should be offloaded onto the GPU is by using the OpenMP 5.1 dispatch directive, as shown in the example below. In this example too, arrays A, B, and C are mapped to the device before the call to the oneMKL routine cblas_dgemm. When cblas_dgemm is called, the corresponding device pointers for A, B, and C will be passed as arguments, so the device copies of A, B, and C will be used in the computation.

The use_device_ptr clause is not needed on the dispatch directive. With OpenMP 5.1, the list of device pointers needed by the oneMKL routines is given in the oneMKL OpenMP offload header file, mkl_omp_offload.h, where the GPU variant function is declared. The user should carefully review the list of device pointers required in the oneMKL header file and make sure that the corresponding arrays are accessible from the device before calling the oneMKL routine.

Note that, depending on the version of the compiler you are using, you may need to add the compiler option -fopenmp-version=51 in order for the dispatch directive to be accepted.

Listing 128: /examples/OpenMP/22_mkl_dispatch/dgemm_dispatch_c.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define EPSILON 0.0001

int main()
{

```

int64_t m, n, k;
double alpha, beta;
double sum;
int64_t i, j, q;
int fail;

printf("
This example computes real matrix C=alpha*A*B+beta*C using
  " Intel oneMKL function dgemm, where A, B, and C are matrices and 
  " alpha and beta are double precision scalars
\n"
);

m = 2000, k = 200, n = 1000;
printf("
Initializing data for matrix multiplication C=A*B for matrix 
  " A(%li x %li) and matrix B(%li x %li)\n\n", m, k, k, n);
alpha = 1.0; beta = 0.0;

printf(" Allocating memory for matrices aligned on 64-byte boundary for better 
  " performance \n\n");
A = (double *)mkl_malloc( m * k * sizeof( double ), 64 );
B = (double *)mkl_malloc( k * n * sizeof( double ), 64 );
C = (double *)mkl_malloc( m * n * sizeof( double ), 64 );
C_fl = (double *)mkl_malloc( m*n*sizeof( double ), 64 );

if (A == NULL || B == NULL || C == NULL || C_fl == NULL) {
  printf("
ERROR: Cannot allocate memory for matrices. Exiting... \n\n");
  return 1;
}

printf(" Intializing matrices \n\n");
for (i = 0; i < (m*k); i++) {
  A[i] = (double)(i+1);
}
for (i = 0; i < (k*n); i++) {
  B[i] = (double)(-i-1);
}
for (i = 0; i < (m*n); i++) {
  C[i] = 0.0;
  C_fl[i] = 0.0;
}

printf(" Computing matrix product using Intel oneMKL dgemm function via CBLAS interface \n\n")
#if _OPENMP
#pragma omp target data map(to: A[0:m*k], B[0:k*n]) map(tofrom: C[0:m*n])
{ #pragma omp target variant dispatch use_device_ptr(A, B, C)
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
  m, n, k, alpha, A, k, B, n, beta, C, n);
}
#endif

(continues on next page)
printf ("\n Top left corner of matrix C: \n")
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C[j+i*n]);
    }
    printf ("\n");
}
printf (" Computing matrix product using for-loops \n")
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++) {
            sum += A[k*i+q] * B[n*q+j];
        }
        C_fl[n*i+j] = alpha * sum + beta * C_fl[n*i+j];
    }
}
printf ("\n Top left corner of matrix C_fl: \n")
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C_fl[j+i*n]);
    }
    printf ("\n");
}
printf (" Computations completed. Verifying... \n\n")
fail = 0;
for (i = 0; i < (m*n); i++) {
    if (fabs(C[i] - C_fl[i]) > EPSILON) {
        fail = 1;
        break;
    }
}
if (fail)
    printf ("\n **** FAIL **** \n");
else
    printf ("\n **** PASS **** \n");
printf ("\n Deallocating memory \n\n")
mkl_free(A);
mkl_free(B);
mkl_free(C);
return fail;
Notes:

- oneMKL routines expect the arrays/matrices to be on the device before the computation will be run. So the user has to map the data to the device, or allocate the data directly on the device, before calling a oneMKL routine.

- If a oneMKL routine is not called from a target variant dispatch (or dispatch) region, or if offload is disabled, then the oneMKL computations will be executed on the CPU.

- Only one call to a oneMKL routine can be issued from an OpenMP target variant dispatch (or dispatch) construct. If there are two consecutive calls to oneMKL routines, then the calls should be placed in separate target variant dispatch (or dispatch) constructs.

Fortran

When calling oneMKL routines from Fortran code, be sure to add the following include statement:

```fortran
include "mkl_omp_offload.f90"
```

Also, if calling oneMKL Fortran API with 32-bit integers, add the following module use statement:

```fortran
use onemkl_blas_omp_offload_lp64
```

On the other hand, if calling oneMKL Fortran API with 64-bit integers, add the following module use statement:

```fortran
use onemkl_blas_omp_offload_ilp64
```

The following Fortran example illustrates how DGEMM is called from a Fortran program, and the include and use statements mentioned above.

**Listing 129:**

```fortran
/examples/OpenMP/22_mkl_dispatch/dgemm_dispatch_ff
```

```fortran
!=============================================================================
! Copyright © 2022 Intel Corporation
!
! SPDX-License-Identifier: MIT
!=============================================================================
include "mkl_omp_offload.f90"

program DGEMM_MAIN

#if defined(MKL_ILP64)
  use onemkl_blas_omp_offload_ilp64
#else
  use onemkl_blas_omp_offload_lp64
#endif

use omp_lib
use iso_fortran_env
implicit none

integer, parameter :: m = 20
```

(continues on next page)
integer, parameter :: k = 5
integer, parameter :: n = 10
double precision a(m,k), b(k,n), c1(m,n), c2(m,n)
double precision alpha, beta
integer i, j

print*
print*, ' D G E M M EXAMPLE PROGRAM'

! Initialize
alpha = 1.025
beta = 0.75
do i = 1, m
  do j = 1, k
    a(i,j) = (i-1) - (0.25 * k)
  end do
end do
do i = 1, k
  do j = 1, n
    b(i,j) = -((i-1) + j)
  end do
end do
do i = 1, m
  do j = 1, n
    c1(i,j) = 0.2 + i - j
    c2(i,j) = 0.2 + i - j
  end do
end do

! Execute DGEMM on host.
call DGEMM('N','N',m,n,k,alpha,a,m,b,k,beta,c1,m)
print *
print *, 'c1 - After DGEMM host execution'
do i=1,m
  print 110, (c1(i,j), j=1,n)
end do
print*

! Execute DGEMM on device
$omp target data map(to: a, b) map(tofrom: c2)
!!$omp dispatch
    call DGEMM('N','N',m,n,k,alpha,a,m,b,k,beta,c2,m)
!!$omp end target data
    print *
    print *, 'c2 - After DGEMM device execution'
    do i=1,m
        print 110, (c2(i,j),j=1,n)
    end do
    print *
101 format(7x,'M=',i5,' N=',i5,' K=',i5)
102 format(7x,'ALPHA=',f10.2,' BETA=',f10.2)
110 format(7x,10(f10.2,2x))
end

To compile and link the above Fortran example with **32-bit integers**:

```bash
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fpp -free -c dgemm_example_f.f
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fsycl -L$\{MKLROOT\}/lib/intel64 -liomp5 -lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl dgemm_example_f.o
```

To compile and link the above Fortran example with **64-bit integers**:

```bash
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -m64 -DMKL_ILP64 -i8 -fpp -free -c dgemm_example_f.f
ifx -fiopenmp -fopenmp-targets=spir64 -qmkl=parallel -fsycl -L$\{MKLROOT\}/lib/intel64 -liomp5 -lsycl -lOpenCL -lstdc++ -lpthread -lm -ldl -lmkl_sycl dgemm_example_f.o
```

After generating the executable (a.out), from a C/C++ or Fortran program, you can run the executable under `ze_tracer` and look for the heading “Device Timing Results” in the generated trace. Below that heading we should see the oneMKL kernels listed. This way we confirm that oneMKL computations have been offloaded onto the GPU.

Example run command:

```bash
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 ze_tracer -h -d ./a.out
```

### 14.5.3 Batching of oneMKL GEMM Calls

The oneMKL library includes “batch” routines that allow the user to batch several oneMKL calls into a single oneMKL call. At runtime, oneMKL will intelligently execute all of the matrix operations to optimize overall performance.

For example, the `cblas_dgemm` routine computes a matrix-matrix product of two general matrices a and b, returning the result in a matrix c. The `cblas_dgemm` interface is shown below.
The `cblas_dgemm` routine is similar to the `cblas_dgemm` routine, but the `cblas_dgemm_batch` routine performs matrix-matrix operations on groups of matrices, processing a number of groups at once.

The `cblas_dgemm_batch` interface is shown below. Note that the interface resembles the `cblas_dgemm` interface. However, it involves passing matrix arguments as arrays of pointers to matrices, and passing parameters as arrays of parameters.

```c
void cblas_dgemm_batch (const CBLAS_LAYOUT layout,
const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array,
const MKL_INT* m_array,
const MKL_INT* n_array,
const MKL_INT* k_array,
const double* alpha_array,
const double* b_array,
const MKL_INT* lda_array,
const double* beta_array,
double**c_array,
const MKL_INT* ldc_array,
const MKL_INT group_count,
const MKL_INT* group_size);
```

The batch operation is defined as follows:

```c
idx = 0
for i = 0 .. group_count - 1
    alpha and beta in alpha_array[i] and beta_array[i]
    for j = 0 .. group_size[i] - 1
        a, b, and c matrices in a_array[idx], b_array[idx], and c_array[idx], respectively
        c := alpha*op(a)*op(b) + beta*c,
        idx = idx + 1
    end for
end for
```

where:

- `op(X)` is one of `op(X) = X`, or `op(X) = XT`, or `op(X) = XH`,
- `alpha` and `beta` are scalar elements of `alpha_array` and `beta_array`,
- `a`, `b`, and `c` are matrices such that for `m`, `n`, and `k` which are elements of `m_array`, `n_array`, and `k_array`:
  - `op(a)` is an `m`-by-`k` matrix,
  - `op(b)` is a `k`-by-`n` matrix,
  - `C` is an `m`-by-`n` matrix.
- `a`, `b`, and `c` represent matrices stored at addresses pointed to by `a_array`, `b_array`, and `c_array`, respectively.
  The number of entries in `a_array`, `b_array`, and `c_array` is `total_batch_count = the sum of all of the group_size entries.`
It is possible to batch the multiplications of different shapes and parameters by packaging them into groups, where each group consists of multiplications of matrices of the same shapes (same m, n, and k) and the same parameters.

The basic assumption for the batch API are that all operations in a batch (whether in the same group or different groups) are independent of one another. So oneMKL does not guarantee any particular ordering between operations in a batch, and will try to execute multiple operations in parallel.

In general, the larger you can make the batch size, the better. This allows oneMKL to better parallelize the operations and distribute the work across the GPU.

We illustrate how two calls to cblas_dgemm can be replaced with one call to cblas_dgemm_batch. The following example includes two calls to cblas_dgemm.

```
Listing 130: /examples/OpenMP/22_mkl_dispatch/dgemm_example_01.cpp
```
A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
  printf( "\n ERROR: Can't allocate memory for matrices. Aborting... \n\n"); return 1;
}

printf (" Intializing matrix data \n\n");
for (i = 0; i < (m*k); i++) {
  A1[i] = A2[i] = (double)(i+1);
}

for (i = 0; i < (k*n); i++) {
  B1[i] = B2[i] = (double)(-i-1);
}

for (i = 0; i < (m*n); i++) {
  C1[i] = C2[i] = 0.0;
  C1_fl[i] = C2_fl[i] = 0.0;
}

printf (" \nComputing matrix product using Intel MKL cblas_dgemm function \n");
t_start = omp_get_wtime();
#pragma omp target data \
  map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n]) \
  map(tofrom: C1[0:m*n], C2[0:m*n])
{
  #pragma omp dispatch
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
             m, n, k, alpha, A1, k, B1, n, beta, C1, n);
  #pragma omp dispatch
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
             m, n, k, alpha, A2, k, B2, n, beta, C2, n);
}
t_end = omp_get_wtime();
printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf("%12.5G", C1[j+i*n]);
    }
    printf("\n");
}

printf("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf("%12.5G", C2[j+i*n]);
    }
    printf("\n");
}

printf("\n Computing matrix product using for-loops \n");
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A1[k*i+q] * B1[n*q+j];
        C1_fl[n*i+j] = sum;
    }
}

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A2[k*i+q] * B2[n*q+j];
        C2_fl[n*i+j] = sum;
    }
}

printf("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf("%12.5G", C1_fl[j+i*n]);
    }
    printf("\n");
}

printf("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf("%12.5G", C2_fl[j+i*n]);
    }
    printf("\n");
}
printf ("\n Computations completed. Verifying... \n\n");

fail = 0;
for (i = 0; i < (m*n); i++) {
    if (! compare(C1[i], C1_fl[i]) || ! compare(C2[i], C2_fl[i])) {
        fail = 1;
        break;
    }
}

if (fail) {
    printf (" **** FAIL **** \n");
} else {
    printf(" time = %lf seconds\n", t_end - t_start);
    printf (" **** PASS **** \n");
}
mkl_free(A1);
mkl_free(B1);
mkl_free(C1);
mkl_free(C1_fl);
mkl_free(A2);
mkl_free(B2);
mkl_free(C2);
mkl_free(C2_fl);
return 0;
}

The two calls to cblas_dgemm in the above example can be batched together, resulting in one call to cblas_dgemm_batch, as shown in the following example. Note that the batch is composed of one group of size 2, since we have two matrix multiplications with the same set of parameters (layout, transa, transb, m, n, k, alpha, lda, ldb, beta, and ldc). total_batch_size in this case is 2.

**Listing 131:** /examples/OpenMP/22_mkl_dispatch/dgemm_batch_example_01.cpp

```c
// =============================================================
// clang-format off
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"
```
```c
#define min(x,y) (((x) < (y)) ? (x) : (y))
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // return true if x and y are the same
    return (fabs(x - y) <= epsilon);
}

int main()
{
    double *A1, *B1, *C1, *C1_fl;
    int m, n, k, i, j, q;
    double alpha, beta;
    double sum;
    int fail;
    double t_start, t_end;

    m = 2000, k = 200, n = 1000;
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n" " performance \n\n");
    A1 = (double *)mkl_malloc (m*k*sizeof(double), 64);
    B1 = (double *)mkl_malloc (k*n*sizeof(double), 64);
    C1 = (double *)mkl_malloc (m*n*sizeof(double), 64);
    C1_fl = (double *)mkl_malloc (m*n*sizeof(double), 64);

    if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
        A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
        printf (" \n ERROR: Can't allocate memory for matrices. Aborting... \n\n");
        return 1;
    }

    printf (" Intializing matrix data \n\n");
    for (i = 0; i < (m*k); i++) {
        A1[i] = A2[i] = (double)(i+1);
    }

    for (i = 0; i < (k*n); i++) {
        B1[i] = B2[i] = (double)(-i-1);
    }

    for (i = 0; i < (m*n); i++) {
        C1[i] = C2[i] = 0.0;
    }

    (continues on next page)
```
C1_fl[i] = C2_fl[i] = 0.0;
}

printf ("\nComputing matrix product using Intel MKL cblas_dgemm_batch function \n");

#define GRP_COUNT 1 // 1 group

MKL_INT group_count = GRP_COUNT;
MKL_INT group_sizes[GRP_COUNT] = {2}; // 8 matrix multiplications

CBLAS_TRANSPOSE transa_array[GRP_COUNT] = {CblasNoTrans};
CBLAS_TRANSPOSE transb_array[GRP_COUNT] = {CblasNoTrans};

MKL_INT m_array[GRP_COUNT] = {m};
MKL_INT n_array[GRP_COUNT] = {n};
MKL_INT k_array[GRP_COUNT] = {k};

MKL_INT lda_array[GRP_COUNT] = {k};
MKL_INT ldb_array[GRP_COUNT] = {n};
MKL_INT ldc_array[GRP_COUNT] = {n};

double alpha_array[GRP_COUNT] = {alpha};
double beta_array[GRP_COUNT] = {beta};

// Number of matrix multiplications = 2
double **a_array, **b_array, **c_array;
a_array = (double **)mkl_calloc(2, sizeof(double*), 64);
b_array = (double **)mkl_calloc(2, sizeof(double*), 64);
c_array = (double **)mkl_calloc(2, sizeof(double*), 64);

t_start = omp_get_wtime();

// Call cblas_dgemm_batch
#pragma omp target enter data \
map(to: A1[0:m*k], B1[0:k*n], C1[0:m*n]) \
map(to: A2[0:m*k], B2[0:k*n], C2[0:m*n])

#pragma omp target data use_device_ptr(A1, B1, C1, A2, B2, C2) \
{
    a_array[0] = A1, a_array[1] = A2;
b_array[0] = B1, b_array[1] = B2;
c_array[0] = C1, c_array[1] = C2;
}

#pragma omp target data \
map(to:a_array[0:2], b_array[0:2], c_array[0:2])
{
    #pragma omp dispatch
cblas_dgemm_batch ( 
        CblasRowMajor, 
        transa_array,
transb_array,
m_array,
n_array,
k_array,
alpha_array,
(const double **)a_array,
lda_array,
(const double **)b_array,
ldb_array,
beta_array,
c_array,
ldc_array,
group_count,
group_sizes);
} // end target data map

#pragma omp target exit data
  map(from: C1[0:m*n], C2[0:m*n])

  t_end = omp_get_wtime();

  printf ("\n Top left corner of matrix C1: \n");
  for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
      printf ("%12.5G", C1[j+i*n]);
    }
    printf ("\n");
  }

  printf ("\n Top left corner of matrix C2: \n");
  for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
      printf ("%12.5G", C2[j+i*n]);
    }
    printf ("\n");
  }

  printf ("\nComputing matrix product using for-loops \n");

  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      sum = 0.0;
      for (q = 0; q < k; q++)
        sum += A1[k*i+q] * B1[n*q+j];
      C1_fl[n*i+j] = sum;
    }
  }

  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      sum = 0.0;
      for (q = 0; q < k; q++)
        sum += A1[k*i+q] * B1[n*q+j];
      C1_fl[n*i+j] = sum;
    }
  }

(continues on next page)
for (q = 0; q < k; q++)
    sum += A2[k*i+q] * B2[n*q+j];
    C2_fl[n*i+j] = sum;
}

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<\text{min}(m,6); i++) {
    for (j=0; j<\text{min}(n,6); j++) {
        printf ("%12.5G", C1_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<\text{min}(m,6); i++) {
    for (j=0; j<\text{min}(n,6); j++) {
        printf ("%12.5G", C2_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Computations completed. Verifying... \n\n");

\text{fail} = 0;
for (i = 0; i < (m*n); i++) {
    if (!\text{compare}(C1[i], C1_fl[i]) || !\text{compare}(C2[i], C2_fl[i])) {
        \text{fail} = 1;
        \text{break};
    }
}

if (\text{fail}) {
    printf (" **** FAIL **** \n");
}
else {
    printf(" time = \%lf seconds\n", t\text{_end} - t\text{_start});
    printf (" **** PASS **** \n");
}

mkl\_free(A1);
mkl\_free(B1);
mkl\_free(C1);
mkl\_free(C1\_fl);
mkl\_free(A2);
mkl\_free(B2);
mkl\_free(C2);
mkl\_free(C2\_fl);

return 0;
The performance of the above two examples when running on the particular GPU used (1-tile only) was as follows:

<table>
<thead>
<tr>
<th>Example</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>dgemm_example_01_c.cpp (two calls to cblas_dgemm)</td>
<td>2.976183 seconds</td>
</tr>
<tr>
<td>dgemm_batch_example_01_c.cpp (one call to cblas_dgemm_batch)</td>
<td>1.881641 seconds</td>
</tr>
</tbody>
</table>

A more complex example of batching is shown below. In this example, we have a batch composed of 3 groups (GROUP_COUNT=3). The size of each group is a randomly chosen number between 1 and 10. Several parameters (layout, transA, transB, m, n, and k) are chosen randomly, but in each group the parameters are the same for all the multiplications. The total_batch_size is equal to the sum of all the group sizes.

Listing 132: /examples/OpenMP/22_mkl_dispatch/dgemm_batch_example_02.cpp

```c
#include <stdio.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"
#include "common.h"
#define GROUP_COUNT 3
int dnum = 0;
int main() {
    CBLAS_LAYOUT layout = (rand_int(0,1) == 0) ? CblasColMajor : CblasRowMajor;
    CBLAS_TRANSPOSE *transA, *transB;
    MKL_INT *m, *n, *k, *lda, *ldb, *ldc;
    double *alpha, *beta;
    MKL_INT *group_size, *sizea_array, *sizeb_array, *sizec_array, total_batch_size = 0, sizea, sizeb, sizec;
```
double **a_array, **b_array, **c_array, **c_ref_array;
double **a_array_dev, **b_array_dev, **c_array_dev;

transA = (CBLAS_TRANSPOSE *)mkl_malloc(GROUP_COUNT * sizeof(CBLAS_TRANSPOSE), 64);
transB = (CBLAS_TRANSPOSE *)mkl_malloc(GROUP_COUNT * sizeof(CBLAS_TRANSPOSE), 64);

m = (MKL_INT *)mkl_malloc(GROUP_COUNT * sizeof(MKL_INT), 64);
transB = (CblasTrans);
alpha = (double *)mkl_malloc(GROUP_COUNT * sizeof(double), 64);
alpha = rand_double_scalar();

if ((m == NULL) || (n == NULL) || (k == NULL) || (lda == NULL) || (ldb == NULL) || (ldc == NULL) ||
    (group_size == NULL) || (alpha == NULL) || (beta == NULL)) {
    printf("Cannot allocate input arrays\n");
    return 1;
}

MKL_INT i, j, p, idx;

for (i = 0; i < GROUP_COUNT; i++) {
    transA[i] = (rand_int(0,1) == 0) ? CblasNoTrans : CblasTrans;
    transB[i] = (rand_int(0,1) == 0) ? CblasNoTrans : CblasTrans;
    alpha[i] = rand_double_scalar();
    beta[i] = rand_double_scalar();
    m[i] = rand_int(1, 20);
    n[i] = rand_int(1, 20);
    k[i] = rand_int(1, 20);
    lda[i] = MAX(m[i], k[i]);
    ldb[i] = MAX(k[i], n[i]);
    ldc[i] = MAX(m[i], n[i]);
    group_size[i] = rand_int(1, 10);
    total_batch_size += group_size[i];
    
    #ifdef MKL_ILP64
        printf("Group %lld: layout = %s, transA = %s, transB = %s, m = %lld, n = %lld, k = %lld,\n            lda = %lld, ldb = %lld, ldc = %lld, alpha = %lf, beta = %lf, group_size = %lld\n", i, (layout == CblasColMajor) ? "Column Major" : "Row Major",
            (transB[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
            m[i], n[i], k[i], lda[i], ldb[i], ldc[i], alpha[i], beta[i], group_size[i]);
    #else
        printf("Group %d: layout = %s, transA = %s, transB = %s, m = %d, n = %d, k = %d, lda = %d, ldb = %d, ldc = %d, alpha = %lf, beta = %lf, group_size = %d\n", i, (layout == CblasColMajor) ? "Column Major" : "Row Major",
            (transB[i] == CblasNoTrans) ? "Non Transpose" : "Transpose",
            m[i], n[i], k[i], lda[i], ldb[i], ldc[i], alpha[i], beta[i], group_size[i]);
    #endif
}
m[i], n[i], k[i], lda[i], ldb[i], ldc[i], alpha[i], beta[i], group_size[i]);
#endif
}

sizea_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
sizeb_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
sizec_array = (MKL_INT *)mkl_malloc(sizeof(MKL_INT) * total_batch_size, 64);
a_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
b_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
c_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
a_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
b_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
c_array_dev = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);
c_ref_array = (double **)mkl_malloc(sizeof(double *) * total_batch_size, 64);

if ((sizea_array == NULL) || (sizeb_array == NULL) || (sizec_array == NULL) || (a_array == NULL) || (b_array == NULL) || (c_array == NULL) || (a_array_dev == NULL) || (b_array_dev == NULL) || (c_array_dev == NULL) || (c_ref_array == NULL)) {
    printf("Cannot allocate matrices and size arrays\n");
    return 1;
}

idx = 0;
for (i = 0; i < GROUP_COUNT; i++) {
    sizea = (((layout == CblasRowMajor) && (transA[i] == CblasTrans)) ||
             ((layout == CblasColMajor) && (transA[i] == CblasNoTrans))) ? lda[i] * k[i] :
             m[i] * lda[i];
    sizeb = (((layout == CblasRowMajor) && (transB[i] == CblasTrans)) ||
             ((layout == CblasColMajor) && (transB[i] == CblasNoTrans))) ? ldb[i] * n[i] :
             k[i] * ldb[i];
    sizec = (layout == CblasColMajor) ? ldc[i] * n[i] : ldc[i] * m[i];
    for (j = 0; j < group_size[i]; j++) {
        a_array[idx] = (double *)mkl_malloc(sizeof(double) * sizea, 64);
        a_array_dev[idx] = a_array[idx];
        sizea_array[idx] = sizea;
        if (a_array[idx] == NULL) {
            printf("cannot allocate a matrices\n");
            return 1;
        }
    }

    b_array[idx] = (double *)mkl_malloc(sizeof(double) * sizeb, 64);
    b_array_dev[idx] = b_array[idx];
    sizeb_array[idx] = sizeb;
    if (b_array[idx] == NULL) {
        printf("cannot allocate b matrices\n");
        return 1;
    }

    c_array[idx] = (double *)mkl_malloc(sizeof(double) * sizec, 64);
    c_array_dev[idx] = c_array[idx];
}
sizec_array[idx] = sizec;
if (c_array[idx] == NULL) {
    printf("cannot allocate c matrices\n");
    return 1;
}
c_ref_array[idx] = (double *)mkl_malloc(sizeof(double) * sizec, 64);
if (c_ref_array[idx] == NULL) {
    printf("cannot allocate c_ref matrices\n");
    return 1;
}
init_double_array(sizea, a_array[idx], 1);
init_double_array(sizeb, b_array[idx], 1);
init_double_array(sizec, c_array[idx], 1);
for (p = 0; p < sizec_array[idx]; p++) c_ref_array[idx][p] = c_array[idx][p];
idx++;
}

// run gemm_batch on host, use standard oneMKL interface
cblas_dgemm_batch(layout, transA, transB, m, n, k, alpha, (const double **) a_array, lda,
(const double **) b_array, ldb, beta, c_ref_array, ldc, GROUP_COUNT, group_size);
double *a, *b, *c;
for (i = 0; i < total_batch_size; i++) {
    a = a_array[i];
    b = b_array[i];
    c = c_array[i];
    #pragma omp target enter data map(to:a[0:sizea_array[i]],b[0:sizeb_array[i]],c[0:sizec_array[i]])
     #pragma omp target data use_device_ptr(a,b,c)
    {
        a_array_dev[i] = a;
        b_array_dev[i] = b;
        c_array_dev[i] = c;
    }
    #pragma omp target data map(to:a_array_dev[0:total_batch_size], \
        b_array_dev[0:total_batch_size], \
        c_array_dev[0:total_batch_size]) device(dnum)
    {
        #pragma omp dispatch
cblas_dgemm_batch(layout, transA, transB, m, n, k, alpha, (const double **) a_array_dev, 
        lda, (const double **) b_array_dev, ldb, beta, c_array_dev, ldc, GROUP_COUNT, group_size);
    }
    for (i = 0; i < total_batch_size; i++) {
        a = a_array[i];
        b = b_array[i];
c = c_array[i];
#pragma omp target exit data map(from:a[0:size_a_array[i]],b[0:size_b_array[i]],
 c[0:size_c_array[i]])
}

double computed, reference, diff;
MKL_INT l;
idx = 0;
for (p = 0; p < GROUP_COUNT; p++) {
   for (l = 0; l < group_size[p]; l++) {
      for (i = 0; i < m[p]; i++) {
         for (j = 0; j < n[p]; j++) {
            if (layout == CblasColMajor) {
               computed = c_array[idx][i + j * ldc[p]];
               reference = c_ref_array[idx][i + j * ldc[p]];
            }
            else {
               computed = c_array[idx][j + i * ldc[p]];
               reference = c_ref_array[idx][j + i * ldc[p]];
            }
            diff = computed - reference;
            if (diff > 0.0001) {
               #ifdef MKL_ILP64
               printf("Error in matrix %lld (group = %lld, matrix index in group = %lld) at index [%lld][%lld], computed = %lf, reference = %lf, difference = %lf\n", idx, p, l,
                   i, j, computed, reference, diff);
               #else
               printf("Error in matrix %d at index [%d][%d], computed = %lf, reference = %lf, difference = %lf\n", idx, i, j, computed, reference, diff);
               #endif
               free_double_matrices(a_array, total_batch_size);
               free_double_matrices(b_array, total_batch_size);
               free_double_matrices(c_array, total_batch_size);
               free_double_matrices(c_ref_array, total_batch_size);
               mkl_free(a_array);
               mkl_free(b_array);
               mkl_free(c_array);
               mkl_free(c_ref_array);
               mkl_free(a_array_dev);
               mkl_free(b_array_dev);
               mkl_free(c_array_dev);
               mkl_free(size_a_array);
               mkl_free(size_b_array);
               mkl_free(size_c_array);
               mkl_free(transA); mkl_free(transB);
               mkl_free(m); mkl_free(n); mkl_free(k);
               mkl_free(lda); mkl_free(ldb); mkl_free(ldc); mkl_free(group_size);
               mkl_free(alpha); mkl_free(beta);
               return 1;
            }
         }
      }
   }
}

(continues on next page)
14.5.4  Speeding Up Independent, Consecutive GEMM Calls

There are various ways to speed up the execution of consecutive GEMM calls that can be executed independently. One way is to batch the GEMM calls by calling the batch version of GEMM as shown above.

Another way is to enclose the calls to GEMM by an OpenMP parallel construct, so each OpenMP thread executing the parallel region dispatches one of the GEMM calls. This parallel approach is illustrated in the following example.
```c
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // returns true if x and y are the same
    return fabs(x - y) <= epsilon;
}

int main()
{
    double *A1, *B1, *C1, *C1_fl;
    int m, n, k, i, j, q;
    double alpha, beta;
    double sum;
    int fail;
    double t_start, t_end;

    m = 2000, k = 200, n = 1000;
    alpha = 1.0; beta = 0.0;

    printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n" " performance \n"");
    A1 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
    B1 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
    C1 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    C1_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    A2 = (double *)mkl_malloc (m*k*sizeof( double ), 64 );
    B2 = (double *)mkl_malloc (k*n*sizeof( double ), 64 );
    C2 = (double *)mkl_malloc (m*n*sizeof( double ), 64 );
    C2_fl = (double *)mkl_malloc (m*n*sizeof( double ), 64 );

    if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL ||
        A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
        printf("\nERROR: Can't allocate memory for matrices. Aborting... \n\n");
        return 1;
    }

    printf (" Intializing matrix data \n\n");
    for (i = 0; i < (m*k); i++) {
        A1[i] = A2[i] = (double)(i+1);
    }

    for (i = 0; i < (k*n); i++) {
        B1[i] = B2[i] = (double)(-i-1);
    }
```

(continues on next page)
for (i = 0; i < (m*n); i++) {
    C1[i] = C2[i] = 0.0;
    C1_fl[i] = C2_fl[i] = 0.0;
}

printf("\nComputing matrix product using Intel MKL cblas_dgemm function \n");

t_start = omp_get_wtime();

#pragma omp target data \
    map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n]) \
    map(tofrom: C1[0:m*n], C2[0:m*n])
{
    #pragma omp parallel num_threads(2)
    {
        int id = omp_get_thread_num();

        if (id == 0) {
            #pragma omp dispatch
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, A1, k, B1, n, beta, C1, n);
        } else if (id == 1) {
            #pragma omp dispatch
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            m, n, k, alpha, A2, k, B2, n, beta, C2, n);
        }
    }
}

t_end = omp_get_wtime();

printf("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf("%12.5G", C1[j+i*n]);
    }
    printf("\n");
}

printf("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf("%12.5G", C2[j+i*n]);
    }
    printf("\n");
}

printf("\nComputing matrix product using for-loops \n");
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A1[k*i+q] * B1[n*q+j];
        C1_fl[n*i+j] = sum;
    }
}

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        sum = 0.0;
        for (q = 0; q < k; q++)
            sum += A2[k*i+q] * B2[n*q+j];
        C2_fl[n*i+j] = sum;
    }
}

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C1_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
    for (j=0; j<min(n,6); j++) {
        printf ("%12.5G", C2_fl[j+i*n]);
    }
    printf ("\n");
}

printf ("\n Computations completed. Verifying... \n\n");
fail = 0;
for (i = 0; i < (m*n); i++) {
    if (!(compare(C1[i], C1_fl[i]) || !compare(C2[i], C2_fl[i]))) {
        fail = 1;
        break;
    }
}

if (fail) {
    printf (" **** FAIL **** \n");
} else {
    printf(" time = %lf seconds\n", t_end - t_start);
    printf (" **** PASS **** \n");
}
Yet another way to speed up the execution of independent, consecutive GEMM calls is to use the `nowait` clause on the `dispatch` construct so the host thread does not have to wait for a dispatched GEMM call to complete before dispatching the next one. After the last GEMM call, we insert an OpenMP `taskwait` directive to guarantee that all the dispatched MKL calls complete before the host thread proceeds any further. This `nowait` approach is illustrated in the following example.

**Listing 134**: /examples/OpenMP/22_mkl_dispatch/dgemm_example_03.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"

#define min(x,y) (((x) < (y)) ? (x) : (y))
#define epsilon 0.0000001f

bool compare(double x, double y)
{
    // returns true if x and y are the same
    return fabs(x - y) <= epsilon;
}

int main()
{
    double *A1, *B1, *C1, *C1_fl;
    int m, n, k, i, j, q;
    double alpha, beta;
```
double sum;
int fail;
double t_start, t_end;

m = 2000, k = 200, n = 1000;
alpha = 1.0; beta = 0.0;

printf (" Allocating memory for matrices aligned on 64-byte boundary for better \n" 
" performance \n\n");
A1 = (double *)mkl_malloc (m*k*sizeof(double), 64);
B1 = (double *)mkl_malloc (k*n*sizeof(double), 64);
C1 = (double *)mkl_malloc (m*n*sizeof(double), 64);
C1_fl = (double *)mkl_malloc (m*n*sizeof(double), 64);
A2 = (double *)mkl_malloc (m*k*sizeof(double), 64);
B2 = (double *)mkl_malloc (k*n*sizeof(double), 64);
C2 = (double *)mkl_malloc (m*n*sizeof(double), 64);
C2_fl = (double *)mkl_malloc (m*n*sizeof(double), 64);

if (A1 == NULL || B1 == NULL || C1 == NULL || C1_fl == NULL || 
A2 == NULL || B2 == NULL || C2 == NULL || C2_fl == NULL) {
    printf ("\n ERROR: Can't allocate memory for matrices. Aborting... \n\n");
    return 1;
}

printf (" Initializing matrix data \n\n");
for (i = 0; i < (m*k); i++) {
    A1[i] = A2[i] = (double)(i+1);
}

for (i = 0; i < (k*n); i++) {
    B1[i] = B2[i] = (double)(-i-1);
}

for (i = 0; i < (m*n); i++) {
    C1[i] = C2[i] = 0.0;
    C1_fl[i] = C2_fl[i] = 0.0;
}

printf ("Computing matrix product using Intel MKL cblas_dgemm function \n");
t_start = omp_get_wtime();

#pragma omp target data \
    map(to: A1[0:m*k], B1[0:k*n], A2[0:m*k], B2[0:k*n]) \
    map(tofrom: C1[0:m*n], C2[0:m*n])
{
    #pragma omp dispatch nowait
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, 
        m, n, k, alpha, A1, k, B1, n, beta, C1, n);
```c
#pragma omp dispatch nowait
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
               m, n, k, alpha, A2, k, B2, n, beta, C2, n);

#pragma omp taskwait
}
}
t_end = omp_get_wtime();

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
  for (j=0; j<min(n,6); j++) {
    printf ("%12.5G", C1[i+j*n]);
  }
  printf ("\n");
}

printf ("\n Top left corner of matrix C2: \n");
for (i=0; i<min(m,6); i++) {
  for (j=0; j<min(n,6); j++) {
    printf ("%12.5G", C2[i+j*n]);
  }
  printf ("\n");
}

printf ("\n Computing matrix product using for-loops \n");
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (q = 0; q < k; q++)
      sum += A1[k*i+q] * B1[n*q+j];
    C1_fl[n*i+j] = sum;
  }
}

for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (q = 0; q < k; q++)
      sum += A2[k*i+q] * B2[n*q+j];
    C2_fl[n*i+j] = sum;
  }
}

printf ("\n Top left corner of matrix C1: \n");
for (i=0; i<min(m,6); i++) {
  for (j=0; j<min(n,6); j++) {
    printf ("%12.5G", C1_fl[j+i*n]);
  }
  printf ("\n");
}
```

(continues on next page)
14.6 Tools to Analyze Performance of OpenMP Applications

There are various tools and mechanisms that are available that help in analyzing the performance of OpenMP programs and identifying bottlenecks.

**Intel® VTune™ Profiler.** Intel® Vtune Profiler can be used to analyze the performance of an application. It helps identify the most time-consuming (hot) functions in the application, whether the application is CPU- or GPU-bound, how effectively it offloads code to the GPU, and the best sections of code to optimize for sequential performance and for threaded performance, among other things. For more information about VTune Profiler, refer to the Intel® VTune™ Profiler User Guide.
Level Zero Tracer. The Level Zero Tracer (ze_tracer) is a host and device tracing tool for Level Zero backend with support for DPC++ and OpenMP GPU offload. For information about this tool, see the Level Zero Tracer section of this document.

When using ze_tracer with the -h and -d options, look at host- and device-side summaries at the end of the trace, under the headings “API Timing Results” and “Device Timing Results”, respectively.

Note that only explicit data transfers appear in the trace. Transfers of data allocated in Unified Shared Memory (USM) may not appear in the trace.

Note:

- ze_tracer is useful for confirming that offloading of oneMKL kernels has occurred. The environment variable OMP_TARGET_OFFLOAD=MANDATORY environment variable does not affect oneMKL, and therefore cannot be used to guarantee that offloading of oneMKL kernels has occurred. One way to check that offloading of oneMKL kernels (and other kernels) has occurred is to see which kernels are listed under “Device Timing Results” in the trace generated by ze_tracer.

SYCL_PI_TRACE=2 environment variable. The DPC++ Runtime Plugin Interface (PI) is an interface layer between the device-agnostic part of DPC++ runtime and the device-specific runtime layers which control execution on devices. Setting SYCL_PI_TRACE=2 provides a trace of all PI calls made with arguments and returned values. For more information, see the DPC++ Runtime Plugin Interface documentation.

LIBOMPTARGET_DEBUG=1 environment variable. LIBOMPTARGET_DEBUG controls whether or not debugging information from libomptarget.so will be displayed.

The debugging output provides useful information about things like ND-range partitioning of loop iterations, data transfers between host and device, memory usage, etc., as shown in the :Using More GPU Resources and :Minimizing Data Transfers and Memory Allocations sections of this document.

For more information about LIBOMPTARGET_DEBUG, see LLVM/OpenMP Runtimes.

LIBOMPTARGET_PLUGIN_PROFILE environment variable. LIBOMPTARGET_PROFILE allows libomptarget.so to generate time profile output. For more information, see LLVM/OpenMP Runtimes.

Dump of compiler-generated assembly for the device. You can dump the compiler-generated assembly by setting the following two environment variables before doing Just-In-Time (JIT) compilation (or before running the program in the case of Ahead-Of-Time (AOT) compilation).

```bash
export IGC_ShaderDumpEnable=1
export IGC_DumpToCustomDir=my_dump_dir
```

LLVM IR, assembly, and GenISA files will be dumped in the sub-directory named my_dump_dir (or any other name you choose). In this sub-directory, you will find a *.asm file for each kernel. The filename indicates the source line number on which the kernel occurs. The header of the file provides information about SIMD width, compiler options, as well as other information. Note that on ATS, ATS assembly will be generated; while on PVC, PVC assembly will be generated.

Also, in my_dump_dir, you will find an file named HardwareCaps.txt that provides information about the GPU, such as EU count, thread count, slice count, etc.

For more information about the Intel® Graphics Compiler and a listing of available flags (environment variables) to control the compilation, see Intel® Graphics Compiler for OpenCL™“Configuration Flags for Linux Release

For additional information about debugging and profiling, refer to the Debugging and Profiling section of this document.
14.7 OpenMP Offload Best Practices

In this chapter we present best practices for improving the performance of applications that offload onto the GPU. We organize the best practices into the following categories, which are described in the sections that follow:

14.7.1 Using More GPU Resources

The performance of offloaded code can be improved by using a larger number of work-items that can run in parallel, thus utilizing more GPU resources (filling up the GPU).

Note:

- ND-range partitioning of loop iterations is decided by compiler and runtime heuristics, and also depends on the GPU driver and the hardware configuration. So it can change over time. However, the methodology of figuring out the partitioning based on LIBOMPTARGET_DEBUG=1 output will remain the same.

**Collapse Clause**

One way to increase parallelism in a loop nest is to use the collapse clause to collapse two or more loops in the loop nest. Collapsing results in a larger number of iterations that can run in parallel, thus using more work-items on the GPU.

In the following example, a loop nest composed of four perfectly nested loops is offloaded onto the GPU. The parallel for directive indicates that the outermost loop (on line 52) is parallel. The number of iterations in the loop is BLOCKS, which is equal to 8.

**Listing 135:**
/examples/OpenMP/01_collapse/test_no_collapse.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
```

(continues on next page)
```c
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    {
    }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    /* map data to device */
    #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])

    start = omp_get_wtime();

    /* offload the kernel with no collapse clause */
    #pragma omp target teams distribute parallel for \ private(b, i, j, k, l)
    for (b = 0; b < BLOCKS; b++) {
        for (i = 0; i < P; i++) {
            for (j = 0; j < P; j++) {
                for (k = 0; k < P; k++) {
                    double ur = 0.;
                    double us = 0.;
                    double ut = 0.;

                    for (l = 0; l < P; l++) {
                        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                    }
                    w[IDX4(b, i, j, k)] = ur * us * ut;
                }
            }
        }
    }

    end = omp_get_wtime();
}
```
oneAPI GPU Optimization Guide

(continued from previous page)

```
#pragma omp target exit data map(from: w[0:SIZE])
/* print result */
printf("no-collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;
```

### Compilation command:
```
icx -fiopenmp -fopenmp-targets=spir64 test_noCollapse.cpp
```

### Run command:
```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 .a.out
```

Libomptarget.so debug information (emitted at runtime when the environment variable LIBOMPTARGET_DEBUG=1) shows the ND-range partitioning of loop iterations and how parallelism is increased by using the collapse clause. In the output, Lb and Ub refer to the parallel loop lower bound and upper bound, respectively, in each dimension of the partitioning.

Without the collapse clause, LIBOMPTARGET_DEBUG=1 output shows the following information about the target region on line 50.

#### Listing 136:
```
_launching target execution __omp_offloading_3d_9b5f515d__Z4main_l45 with
  pointer 0x0000000143d5d8 (index=1).
_target LEVEL0 RTL --> Executing a kernel 0x0000000143d5d8...
_target LEVEL0 RTL --> Assumed kernel SIMD width is 32
_target LEVEL0 RTL --> Preferred group size is multiple of 64
_target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 7, Stride = 1
_target LEVEL0 RTL --> Group sizes = {1, 1, 1}
_target LEVEL0 RTL --> Group counts = {8, 1, 1}
```

Note that without the collapse clause, the number of parallel loop iterations = 8, since the upper bound of the outermost loop (BLOCKS) = 8. In this case, we end up with 8 work-groups, with one work-item each (total work-group count = 8 x 1 x 1 = 8, and each work-group size = 1 x 1 x 1 = 1 work-item). The kernel is vectorized using SIMD 32, which means every 32 work-items in a work-group are combined into one sub-group. Since we have only one work-item per work-group, it follows that each work-group has only one sub-group where only one SIMD lane is active.

We can increase parallelism and hence the number of work-items used on the GPU by adding a collapse clause on the parallel for directive. We start by adding the collapse(2) clause, as shown in the following modified example.
Listing 137: /examples/OpenMP/01 Collapse/test collapse 2levels.cpp

```c
/* offload the kernel with collapse clause */
#pragma omp target teams distribute parallel for collapse(2) \
  private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
  for (i = 0; i < P; i++) {
    for (j = 0; j < P; j++) {
      double ur = 0.;
      double us = 0.;
      double ut = 0.;
      for (l = 0; l < P; l++) {
        ur += dx[IDXX(i, l)] * u[IDX4(b, l, j, k)];
        us += dx[IDXX(k, l)] * u[IDX4(b, i, l, k)];
        ut += dx[IDXX(j, l)] * u[IDX4(b, i, j, l)];
      }
      w[IDX4(b, i, j, k)] = ur * us * ut;
    }
  }
}
```

LIBOMPTARGET_DEBUG=1 output shows the following partitioning when collapse(2) is used.

Listing 138: /examples/OpenMP/01 Collapse/test collapse 2levels.debug

```
Libomp target --> Launching target execution __omp_offloading_3d_9b5f515f__Z4main_l45 with
  -> pointer 0x00000000017f45d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000017f45d8 ...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 1: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {1, 1, 1}
Target LEVEL0 RTL --> Group counts = {16, 8, 1}
```

Note that with collapse(2), the number of parallel loop iterations = BLOCKS x P = 8 x 16 = 128. In this case, we end up with 128 work-groups, and each work-group has 1 work-item (total work-group count = 16 x 8 x 1 = 128, and each work-group size = 1 x 1 = 1 work-item). The kernel is vectorized using SIMD32, which means every 32 work-items in a work-group are combined into one sub-group. Since we have only one work-item per work-group, it follows that each work-group has only one sub-group where only one SIMD lane is active.

On the other hand, if we use the collapse(3) clause, LIBOMPTARGET_DEBUG=1 output shows the following partitioning.
Listing 139: /examples/OpenMP/01_collapse/test-collapse_3levels.debug

Libomptarget --> Launching target execution __omp_offloading_3d_9b5f5160__Z4main_l45 with u
        --> pointer 0x000000001728d08 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x000000001728d08...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 1: Lb = 0, Ub = 15, Stride = 1
Target LEVEL0 RTL --> Level 2: Lb = 0, Ub = 7, Stride = 1
Target LEVEL0 RTL --> Group sizes = {8, 1, 1}
Target LEVEL0 RTL --> Group counts = {2, 16, 8}

With collapse(3), the number of resulting parallel loop iterations = BLOCKS x P x P = 8 x 16 x 16 = 2048. In this case, we end up with 256 work-groups, and each work-group has 8 work-items (total work-group count = 2 x 16 x 8 = 256, and each work-group size = 8 x 1 x 1 = 8 work-items). The kernel is vectorized using SIMD 32, which means every 32 work-items in a work-group are combined into one sub-group. Since we have only 8 work-items per work-group, it follows that we have only one sub-group where only 8 SIMD lanes are active.

If we were to use the collapse(4) clause, instead of collapse(3), LIBOMPTARGET_DEBUG=1 output shows the following partitioning.

Listing 140: /examples/OpenMP/01_collapse/test-collapse_4levels.debug

Target LEVEL0 RTL --> Executing a kernel 0x00000000aabad8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}

With collapse(4), the number of resulting parallel loop iterations = BLOCKS x P x P x P = 8 x 16 x 16 x 16 = 32768. In this case, we have 512 work-groups, and each work-group has 64 work-items (total work-group count = 512 x 1x1x1=512, and each work-group size = 64 x 1 x 1 = 64 work-items). The kernel is vectorized using SIMD 32, which means every 32 work-items are combined into one sub-group. It follows that each work-group has 2 sub-groups.

Using the collapse clause significantly reduces the runtime of the loop nest. The performance of the various versions when running on the particular GPU used (1-tile only) was as follows:

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no collapse</td>
<td>0.002430</td>
</tr>
<tr>
<td>collapse(2)</td>
<td>0.000839</td>
</tr>
<tr>
<td>collapse(3)</td>
<td>0.000321</td>
</tr>
<tr>
<td>collapse(4)</td>
<td>0.000325</td>
</tr>
</tbody>
</table>

The above timings show that adding the collapse(3) or collapse(4) clause gives a performance boost of about 7.5x. (0.000321 seconds versus 0.002430 seconds).

Notes:
- On the GPU, the `collapse` clause may not result in any actual loop collapsing at all, but the clause conveys to the compiler and runtime the degree of parallelism in the loop nest and is used in determine the ND-range partitioning.

- To take advantage of vector loads and stores, it is recommended that the innermost loop in a loop nest not be included in the collapsing so it can be vectorized. Best performance is achieved when the innermost loop has unit stride and its number of iterations is at least as large as the SIMD width.

### 14.7.2 Minimizing Data Transfers and Memory Allocations

When offloading computations onto the GPU, it is important to minimize data transfers between the host and the device, and reduce memory allocations on the device. There are various ways to achieve this, as described below.

**Use target enter data and target exit data Directives**

When variables are used by multiple target constructs, the `target enter data` and `target exit data` pair of directives can be used to minimize data transfers between host and device.

Place the `target enter data` directive before the first target construct to transfer data from host to device, and place the `target exit data` directive after the last target construct to transfer data from device to host.

Consider the following example where we have two target constructs (on lines 47 and 71), and each target construct reads arrays dx and u and and writes to array w.

**Listing 141:** /examples/OpenMP/03_target_enter_exit_data/test_no_target_enter_exit_data.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
```

(continues on next page)
double w[SIZE]; /* output */
double u[SIZE], dx[P * P]; /* input */
int b, i, j, k, l; /* loop counters */
double start, end; /* timers */

omp_set_default_device(0);

/* dummy target region, so as not to measure startup time. */
#pragma omp target
{
}

/* initialize input with random values */
srand(0);
for (int i = 0; i < SIZE; i++)
  u[i] = scaled_rand();
for (int i = 0; i < P * P; i++)
  dx[i] = scaled_rand();

start = omp_get_wtime();

/* offload kernel #1 */
#pragma omp target teams distribute parallel for collapse(4) \
  map(to: u[0:SIZE], dx[0:P * P]) map(from: w[0:SIZE]) \
  private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
  for (i = 0; i < P; i++) {
    for (j = 0; j < P; j++) {
      for (k = 0; k < P; k++) {
        double ur = 0.;
        double us = 0.;
        double ut = 0.;

        for (l = 0; l < P; l++) {
          ur += dx[idx2(i, l)] * u[idx4(b, l, j, k)];
          us += dx[idx2(k, l)] * u[idx4(b, i, l, k)];
          ut += dx[idx2(j, l)] * u[idx4(b, i, j, l)];
        }

        w[idx4(b, i, j, k)] = ur * us * ut;
      }
    }
  }
}

/* offload kernel #2 */
#pragma omp target teams distribute parallel for collapse(4) \
  map(to: u[0:SIZE], dx[0:P * P]) map(to: from: w[0:SIZE]) \
  private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
  for (i = 0; i < P; i++) {
    for (j = 0; j < P; j++) { /* remainder */
      ...
    }
  }
}
for (j = 0; j < P; j++) {
    for (k = 0; k < P; k++) {
        double ur = b + i + j - k;
        double us = b + i + j - k;
        double ut = b + i + j - k;

        for (l = 0; l < P; l++) {
            ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
            us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
            ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
        }

        w[IDX4(b, i, j, k)] += ur * us * ut;
    }
}
}

end = omp_get_wtime();

/* print result */
printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;

Compilation command:

icx -fiopenmp -fopenmp-targets=spir64 test_no_target_enter_exit_data.cpp

Run command:

OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 ./a.out

When the first target construct (on line 47) is encountered:

- Since arrays dx and u appear in a map clause with the to map-type, storage is allocated for the arrays on the device, and the values of dx and u on the host are copied to the corresponding arrays on the device.
- Since array w appears in a map clause with the from map-type, uninitialized storage is allocated for array w on the device.

At the end of the first target region:

- Since array w appears in a map clause with the from map-type, the values of array w on the device are copied to the original array w on the host.

When the second target construct (on line 71) is encountered:

- Since arrays dx, u, and w appear in a map clause with the to map-type, storage is allocated for arrays dx, u, and w on the device and the values of arrays dx, u, and w on the host are copied to the corresponding arrays on the device.
At the end of the second target region:

- Since array \( w \) appears in a map clause with the from map-type, the values of array \( w \) on the device are copied to the original array \( w \) on the host.

LIBOMPTARGET_DEBUG=1 output shows that both target regions (on lines 47 and 71) have the same data partitioning.

**Listing 142**: /examples/OpenMP/03_target_enter_exit_data/test_no_target_enter_exit_data.debug

Libomptarget --> Launching target execution _omp_offloading_3d_15ece5c8__Z4main_l42 with
  pointer 0x00000000024cb5d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000024cb5d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = \{64, 1, 1\}
Target LEVEL0 RTL --> Group counts = \{512, 1, 1\}

**Listing 143**: /examples/OpenMP/03_target_enter_exit_data/test_target_enter_exit_data.debug

Target LEVEL0 RTL --> Executing a kernel 0x0000000002b9c5e0...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = \{64, 1, 1\}
Target LEVEL0 RTL --> Group counts = \{512, 1, 1\}
Target LEVEL0 RTL --> Kernel Pointer argument 0 (value: 0xff00fffffffffee0000) was set successfully for device 0.

The amount of data transferred (for both target regions) can be seen in LIBOMPTARGET_DEBUG=1 output by grepping for "Libomptarget --> Moving":

```bash
$ grep "Libomptarget --> Moving" test_no_target_enter_exit_data.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007fff60f05030) -> (tgt:0xff00fffffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60ec5030) -> (tgt:0xff00fffffffffee0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff20000) -> (hst:0x00007fff60e85030)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60ec5030) -> (tgt:0xff00fffffffffee0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffff20000) -> (hst:0x00007fff60e85030)
Libomptarget --> Moving 262144 bytes (hst:0x00007fff60ec5030) -> (tgt:0xff00fffffffffee0000)
```

You can reduce the copying of data from host to device and vice versa by using the target enter data and target exit data directives as shown in this modified example.

**Listing 144**: /examples/OpenMP/03_target_enter_exit_data/test_target_enter_exit_data.cpp

```cpp
1  //==============================================================================
2  // Copyright © 2022 Intel Corporation
```
# Header License Information

// SPDX-License-Identifier: MIT

// clang-format off

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
  double w[SIZE];  /* output */
  double u[SIZE], dx[P * P]; /* input */
  int b, i, j, k, l; /* loop counters */
  double start, end; /* timers */

  omp_set_default_device(0);

  /* dummy target region, so as not to measure startup time. */
  #pragma omp target
  {
  }

  /* initialize input with random values */
  srand(0);
  for (int i = 0; i < SIZE; i++)
    u[i] = scaled_rand();
  for (int i = 0; i < P * P; i++)
    dx[i] = scaled_rand();
  start = omp_get_wtime();

  /* map data to device. alloc for w avoids map(tofrom: w[0:SIZE])
  on target by default. */
  #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P]) \ 
  map(alloc: w[0:SIZE])

  /* offload kernel #1 */
  #pragma omp target teams distribute parallel for collapse(4) \ 
  private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                double ur = 0.;
                double us = 0.;
                double ut = 0.;

                for (l = 0; l < P; l++) {
                    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }

                w[IDX4(b, i, j, k)] = ur * us * ut;
            }
        }
    }
}

/* offload kernel #2 */
#pragma omp target teams distribute parallel for collapse(4) \ 
    private(b, i, j, k, l)
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                double ur = b + i + j - k;
                double us = b + i + j - k;
                double ut = b + i + j - k;

                for (l = 0; l < P; l++) {
                    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }

                w[IDX4(b, i, j, k)] += ur * us * ut;
            }
        }
    }
}

#pragma omp target exit data map(from: w[0:SIZE])
end = omp_get_wtime();

/* print result */
printf("target region: w[0]=%lf time=%lf\n", w[0], end - start);
return 0;
In the modified example, when the target enter data directive (on line 48) is encountered:

- Since arrays dx and u appear in a map clause with the to map-type, storage is allocated for arrays dx and u on the device, and the values of arrays dx and u on the host are copied to the corresponding arrays on the device.

- Since array w appears in a map clause with the alloc map-type, uninitialized storage is allocated for array w on the device.

When the first target construct (on line 52) is encountered:

- The runtime checks whether storage corresponding to arrays dx, u, and w already exists on the device. Since it does, no data transfer occurs.

At the end of the first target region:

- The runtime will recognize that the storage for arrays dx, u, and w should remain on the device, and no copy back from the device to the host occurs.

When the second target construct (on line 75) is encountered:

- Again no data transfer from the host to the device occurs.

At the end of the second target region:

- The runtime will recognize that the storage for the arrays dx, u, and w should remain on the device, and no copy back from device to host will occur.

When the target exit data directive (on line 97) is encountered:

- Since array w appears in a map clause with the from map-type, the values of array w on the device are copied to the original array w on the host.

Using the target enter data and target exit data pair of directives reduced the runtime on the particular GPU used (1-tile only):

<table>
<thead>
<tr>
<th>No target enter/exit data version</th>
<th>0.001204 seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>target enter/exit data version</td>
<td>0.000934 seconds</td>
</tr>
</tbody>
</table>

LIBOMPTARGET_DEBUG=1 output shows that data partitioning is the same in both examples (with and without target enter data and target exit data).

**Listing 145**: /examples/OpenMP/03_target_enter_exit_data/test_target_enter_exit_data.debug

Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899939c0, Size=2048)...  
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffd899939c0, TgtPtrBegin=0xff00ffffffee0000, Size=2048, DynRefCount=2 (update suppressed), HoldRefCount=0  
Libomptarget --> Obtained target argument (Begin: 0xffffffff0000, Offset: 0) from host  
   pointer 0x00007ffd899939c0  
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899539c0, Size=262144)...  
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffd899539c0, TgtPtrBegin=0xff00ffffffef0000, Size=262144, DynRefCount=2 (update suppressed), HoldRefCount=0  
Libomptarget --> Obtained target argument (Begin: 0xffffffff0000, Offset: 0) from host  
   pointer 0x00007ffd899539c0  
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffd899139c0, Size=262144)...
The improvement in performance when using target enter data and target exit data came from the reduction of data transfers, where we now have the following three data transfers:

$ grep "Libomptarget --> Moving" test_target_enter_exit_data.debug
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd899539c0) -> (tgt:0xff00ffffffef0000)
Libomptarget --> Moving 2048 bytes (hst:0x00007ffd899939c0) -> (tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00fffffff30000) -> (hst:0x00007ffd899139c0)

Choose map-type Appropriately

For improved performance, it is important that the map-type for a mapped variable matches how the variable is used in the target construct.

In the following example, arrays u and dx are read only in the target construct, and array w is written to in the target construct. However, the map-types for all these variables is (inefficiently) specified to be tofrom.

Listing 147:
/examples/OpenMP/10_map/test_map_tofrom.cpp

(continues on next page)
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    start = omp_get_wtime();

    #pragma omp target teams distribute parallel for /
    private(b, i, j, k, l) \
    map(tofrom: u[0:SIZE], dx[0:P * P]) \
    map(tofrom: w[0:SIZE])
    for (int n = 0; n < SIZE; n++) {
        k = n - (n / P) * P;
        j = (n - k) / P;
        i = (n - (j * P + k)) / (P * P);
        b = n / (P * P * P);

        double ur = 0.;
        double us = 0.;
        double ut = 0.;

        for (l = 0; l < P; l++) {
            ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
            us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
            ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
        }

        w[IDX4(b, i, j, k)] = ur * us * ut;
    }

    end = omp_get_wtime();

    printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);
}

(continues on next page)
```c
    
return 0;
}
```

**Compilation command:**

```bash
icx -fiopenmp -fopenmp-targets=spir64 test_map_tofrom.cpp
```

**Run command:**

```bash
OMP_TARGET_OFFLOAD=MANDATORY ZE AFFINITY MASK=0.0 LIBOMPTARGET_DEBUG=1 ./a.out
```

For better performance, the map-type for `u` and `dx` should be `to`, and the map-type for `w` should be `from`, as shown in the following modified example.

**Listing 148:**

`/examples/OpenMP/10_map/test_map_to_or_from.cpp`

```c
#pragma omp target teams distribute parallel for \
private(b, i, j, k, l) \
map(to: u[0:SIZE], dx[0:P * P]) \
map(from: w[0:SIZE])
for (int n = 0; n < SIZE; n++) {
    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);
    double ur = 0.;
    double us = 0.;
    double ut = 0.;
    for (l = 0; l < P; l++) {
        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
    }
    w[IDX4(b, i, j, k)] = ur * us * ut;
}
```

Using more specific map-types (to or from, instead of to from), reduced the runtime on the particular GPU used (1-tile only):

```
to from map-types version : 0.001141 seconds
to or from map-types version : 0.000908 seconds
```

LIBOMPTARGET_DEBUG=1 output shows that there are unnecessary data transfers between the host and the device when the to from map-type is used for `u`, `dx`, and `w`. With to from, there are six transfers to copy the values of `u`, `dx`, and `w` from the host to the device and vice-versa:
With the more specific map-types (to or from), we see only three data transfers: two transfers to copy the values of u and dx from host to device, and one transfer to copy the values of w from device to host:

```
$ grep "Libomptarget --> Moving" test_map_to_or_from.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007fffc2258fd0) -> (tgt:0xff00fffffffe0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffc2218fd0) -> (tgt:0xff00ffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007fffc21d8fd0) -> (tgt:0xff00fffffff20000)
```

**Do Not Map Read-Only Scalar Variables**

The compiler will produce more efficient code if read-only scalar variables in a target construct are not mapped, but are listed in a firstprivate clause on the target construct or not listed in any clause at all. (Note that when a scalar variable is not listed in any clause on the target construct, it will be firstprivate by default.)

Listing a read-only scalar variable on a map(to:) clause causes unnecessary memory allocation on the device and copying of data from the host to the device. On the other hand, when a read-only scalar is specified to be firstprivate on the target construct, the variable is passed as argument when launching the kernel, and no memory allocation or copying for the variable is required.

In the following example, a loop nest is offloaded onto the GPU. In the target construct, the three scalar variables, s1, s2, and s3, are read-only and are listed in a map(to:) clause.

**Listing 149:**
/examples/OpenMP/05_scalars_fp/test_scalars_map.cpp

```cpp
// ==============================================================
// Copyright © 2022 Intel Corporation
// //
// // SPDX-License-Identifier: MIT
// // ==============================================================
// clang-format off
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
```
```c
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    double s1, s2, s3; /* scalars */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();
    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    /* initialize scalars */
    s1 = u[SIZE / 2];
    s2 = scaled_rand();
    s3 = 0.145;

    /* map data to device */
    #pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])

    start = omp_get_wtime();

    /* offload the kernel with collapse clause */
    #pragma omp target teams distribute parallel for collapse(4) \
        map(to: s1, s2, s3) private(b, i, j, k, l)
    for (b = 0; b < BLOCKS; b++) {
        for (i = 0; i < P; i++) {
            for (j = 0; j < P; j++) {
                for (k = 0; k < P; k++) {
                    double ur = 0.;
                    double us = 0.;
                    double ut = 0.;
                }
            }
        }
    }

    for (l = 0; l < P; l++) {
        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] + s1;
        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] - s2;
        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
    }

    end = omp_get_wtime();
}
```

Compilation command:

```
icx -fiopenmp -fopenmp-targets=spir64 test_scalars_map.cpp
```

Run command:

```
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 ./a.out
```

It is more efficient to list s1, s2, and s3 in a firstprivate clause on the target construct, as shown in the modified example below, or not list them in any clause at all.

**Listing 150:**

```
/* offload the kernel with collapse clause */
#pragma omp target teams distribute parallel for collapse(4) \ 
    firstprivate(s1, s2, s3) private(b, i, j, k, l) 
for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                double ur = 0.;
                double us = 0.;
                double ut = 0.;

                for (l = 0; l < P; l++) {
                    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)] * s1;
                    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)] * s2;
                    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)] * s3;
                }
                w[IDX4(b, i, j, k)] = ur * us * ut;
            }
        }
    }
    end = omp_get_wtime();

#pragma omp target exit data map(from: w[0:SIZE])
/* print result */
printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
return 0;
```
Using `firstprivate(s1, s2, s3)`, instead of `map(to:s1, s2, s3)`, reduced the runtime on the particular GPU used (1-tile only):

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(to:s1, s2, s3)</code> version</td>
<td>0.001324 seconds</td>
</tr>
<tr>
<td><code>firstprivate(s1, s2, s3)</code> version</td>
<td>0.000730 seconds</td>
</tr>
</tbody>
</table>

LIBOMPTARGET_DEBUG=1 output shows that data partitioning is the same in both examples (with `map(to:s1, s2, s3)` and with `firstprivate(to:s1, s2, s3)`.

**Listing 151:**

```shell
/examples/OpenMP/05_scalars_fp/test_scalars_map.debug
```

**Listing 152:**

```shell
/examples/OpenMP/05_scalars_fp/test_scalars_fp.debug
```

However, more device memory allocations and host-to-device data transfers occur when the `map(to:s1, s2, s3)` clause is used.

LIBOMPTARGET_DEBUG=1 output shows the following data about memory allocations on the device when `map(to:s1, s2, s3)` clause is used.

**Listing 153:**

```shell
/examples/OpenMP/05_scalars_fp/test_scalars_map.debug
```
Target LEVEL0 RTL --> -- Freed : 1179648, 262336
Target LEVEL0 RTL --> -- InUse : 0, 264192
Target LEVEL0 RTL --> -- PeakUse : 1179648, 526528
Target LEVEL0 RTL --> -- NumAllocs: 3, 6

Note that the memory allocated is 1,179,648 bytes, and the number of allocations (from the pool) is 6 – for the three arrays (dx, u, and w) and the three scalars (s1, s2, and s3).

In contrast, LIBOMPTARGET_DEBUG=1 output shows fewer memory allocations on the device when the firstprivate(s1, s2, s3) clause is used. The memory allocated is reduced from 1,179,648 to 1,114,112 bytes (a reduction of 64 kilobytes), and the number of allocations (from the pool) is reduced from 6 to 3, as shown below.

**Listing 154:**
/examples/OpenMP/05_scalars_fp/test_scalars_fp.debug

Target LEVEL0 RTL --> Memory usage for device memory, device 0x0000000001bab440
Target LEVEL0 RTL --> -- Allocator: Native, Pool
Target LEVEL0 RTL --> -- Requested: 1114112, 526336
Target LEVEL0 RTL --> -- Allocated: 1114112, 526336
Target LEVEL0 RTL --> -- Freed : 1114112, 264192
Target LEVEL0 RTL --> -- InUse : 0, 264192
Target LEVEL0 RTL --> -- PeakUse : 1114112, 526336
Target LEVEL0 RTL --> -- NumAllocs: 2, 3

In addition to more memory allocations, using the map(to: ) clause results in more data transfers from host to device. This can be seen by grepping for "Libomptarget --> Moving" in the LIBOMPTARGET_DEBUG=1 output:

```
$ grep "Libomptarget --> Moving" test_scalars_map.debug
Libomptarget --> Moving 262144 bytes (hst:0x000007ffdf5526760) -> (tgt:0xff00fffffff30000)
Libomptarget --> Moving 2048 bytes (hst:0x000007ffdf5566760) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffdf55670a0) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffdf556700a) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffdf5566760) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffdf5566760) -> (tgt:0xff00fffffff0000)

In contrast, when the firstprivate(to:s1, s2, s3) clause is used, LIBOMPTARGET_DEBUG=1 output shows:

```
$ grep "Libomptarget --> Moving" test_scalars_fp.debug
Libomptarget --> Moving 262144 bytes (hst:0x000007ffda809c4a0) -> (tgt:0xff00fffffff30000)
Libomptarget --> Moving 2048 bytes (hst:0x000007ffda809c4a0) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffda809c4a0) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffda809c4a0) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffda809c4a0) -> (tgt:0xff00fffffff0000)
Libomptarget --> Moving 8 bytes (hst:0x000007ffda809c4a0) -> (tgt:0xff00fffffff0000)
```

Note that in the example with map(to:s1, s2, s3) we have three additional data transfers, each moving 8 bytes. These transfers are for copying the values of s1, s2, and s3 from host to device.
Do Not Map Loop Bounds to Get Better ND-Range Partitioning

As mentioned above, the compiler will produce more efficient code if read-only scalar variables in a target construct are not mapped, but are listed in a first private clause on the target construct or not listed in any clause at all.

This is especially true when the scalars in question are parallel loop bounds in the target construct. If any of the loop bounds (lower bound, upper bound, or step) are mapped, then this will result in unnecessary memory allocation on the device and copying of data from host to device. Loop partitioning will also be affected, and may result in non-optimal ND-range partitioning that negatively impacts performance.

Consider the following example, where a parallel for loop is offloaded onto the GPU. The upper bound of the for loop is the scalar variable upper, which is mapped by the target construct (on line 53).

Listing 155: /examples/OpenMP/07_loop_bounds/test_loop_bounds_map.cpp

```c
// SPDX-License-Identifier: MIT
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)
int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    int b, i, j, k, l; /* loop counters */
    int upper;
    double start, end; /* timers */
    omp_set_default_device(0);
    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    {
    }
```

(continues on next page)
/* initialize input with random values */
srand(0);
for (int i = 0; i < SIZE; i++)
    u[i] = scaled_rand();

for (int i = 0; i < P * P; i++)
    dx[i] = scaled_rand();

upper = (int)dx[0] + SIZE;

/* map data to device */
#pragma omp target enter data map(to: u[0:SIZE], dx[0:P * P])

start = omp_get_wtime();

/* offload kernel */
#pragma omp target teams distribute parallel for private(b, i, j, k, l) \ 
    map(to: upper)
for (int n = 0; n < upper; n++) {
    double ur = 0.;
    double us = 0.;
    double ut = 0.;

    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);
    b = n / (P * P * P);

    for (l = 0; l < P; l++) {
        ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
        us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
        ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
    }

    w[IDX4(b, i, j, k)] = ur * us * ut;
}

end = omp_get_wtime();

/* map data from device */
#pragma omp target exit data map(from: w[0:SIZE])
printf("offload: w[0]=%lf time=%lf\n", w[0], end - start);

return 0;

Compilation command:

cxx -fiopenmp -fopenmp-targets=spir64 test_loop_bounds_map.cpp

Run command:
Since upper is mapped, the value of the variable upper on the host may be different from the value on the device. Because of this, when the target region is offloaded at runtime, the number of loop iterations in the offloaded loop is not known on the host. In this case, the runtime (libomptarget.so) will use device and kernel properties to choose ND-range partitioning that fills the whole GPU.

The compiler-generated code for the offloaded loop includes an additional innermost loop (per work-item) inside the offloaded loop. If the global size selected happens to be smaller than the actual number of loop iterations, each work-item will process multiple iterations of the original loop. If the global size selected is larger than the actual number of loop iterations, some of the work-items will not do any work. An if-condition inside the loop generated by the compiler will check this and skip the rest of the loop body.

For the above example (where upper is mapped), LIBOMPTARGET_DEBUG=1 shows the following ND-range partitioning.

**Listing 156: /examples/OpenMP/07_loop_bounds/test_loop_bounds_map.debug**

```
Libomptarget --> Launching target execution __omp_offloading_3d_1ff4bf1c__Z4main_l48 with, 
              --> pointer 0x00000000021175d8 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000021175d8...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Group sizes = {1024, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}
```

Note that in the above partitioning, the total number of work-items = 512 x 1024 = 524,288, which is larger than the actual number of loop iterations (32,767). So some of the work-items will not do any work.

Better ND-range partitioning is achieved if the number of loop iterations in the offloaded loop is known on the host. This allows the compiler and runtime to do an ND-range partitioning that matches the number of loop iterations.

To get this better partitioning, we use firstprivate(upper) instead of map(to:upper) on the target construct, as shown in the modified example below. This way, the compiler knows that the value of the variable upper on the host is the same as the value of the variable upper on the device.

**Listing 157: /examples/OpenMP/07_loop_bounds/test_loop_bounds_fp.cpp**

```
#pragma omp target teams distribute parallel for private(b, i, j, k, l) 
     firstprivate(upper)
for (int n = 0; n < upper; n++) {
    double ur = 0.;
    double us = 0.;
    double ut = 0.;
    k = n - (n / P) * P;
    j = (n - k) / P;
    i = (n - (j * P + k)) / (P * P);

(continues on next page)
```
\[
b = n / (P \times P \times P);
\]

```c
for (l = 0; l < P; l++) {
    ur += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
    us += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
    ut += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
}
```

w[IDX4(b, i, j, k)] = ur * us * ut;

For the modified example (where upper is firstprivate), LIBOMPTARGET_DEBUG=1 shows the following ND-range partitioning.

**Listing 158:** /examples/OpenMP/07_loop_bounds/test_loop_bounds_fp.debug

Libomptarget --> Launching target execution __omp_offloading_3d_1fed0edf__Z4main_l48 with pointer 0x00000000029b3d08 (index=1).
Target LEVEL0 RTL --> Executing a kernel 0x00000000029b3d08...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred group size is multiple of 64
Target LEVEL0 RTL --> Level 0: Lb = 0, Ub = 32767, Stride = 1
Target LEVEL0 RTL --> Group sizes = {64, 1, 1}
Target LEVEL0 RTL --> Group counts = {512, 1, 1}

Note that in the above partitioning, the total number of work-items = 512 \times 64 = 32,767, which exactly matches the actual number of loop iterations.

Using `firstprivate` instead of `map(to:)` reduced the runtime on the particular GPU used (1-tile only):  

<table>
<thead>
<tr>
<th>Version</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(to:)</code> version</td>
<td>0.000415 s</td>
</tr>
<tr>
<td><code>firstprivate</code> version</td>
<td>0.000307 s</td>
</tr>
</tbody>
</table>

**Allocate Memory Directly on the Device**

As mentioned above, the `map` clause determines how an original host variable is mapped to a corresponding variable on the device. However, the `map(to:)` clause may not be the most efficient way to allocate memory for a variable on the device.

In the following example, the variables `ur`, `us`, and `ut` are used as work (temporary) arrays in the computations on the device. The arrays are mapped to the device using `map(to:)` clauses (lines 51-53).

**Listing 159:** /examples/OpenMP/11_device_alloc/test_map_to.cpp

//==============================================
// Copyright © 2022 Intel Corporation
// (continues on next page)
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
    double w[SIZE]; /* output */
    double u[SIZE], dx[P * P]; /* input */
    double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
    int b, i, j, k, l; /* loop counters */
    double start, end; /* timers */

    omp_set_default_device(0);

    /* dummy target region, so as not to measure startup time. */
    #pragma omp target
    { ; }

    /* initialize input with random values */
    srand(0);
    for (int i = 0; i < SIZE; i++)
        u[i] = scaled_rand();

    for (int i = 0; i < P * P; i++)
        dx[i] = scaled_rand();

    start = omp_get_wtime();

    /* offload the kernel */
    #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) \
    map(to:u[0:SIZE],dx[0:P*P]) \
    map(from:w[0:SIZE]) \
    map(to:ur[0:SIZE]) \
    map(to:us[0:SIZE]) \
    map(to:ut[0:SIZE]) \
    private(b,i,j,k,l)
```

for (b = 0; b < BLOCKS; b++) {
    for (i = 0; i < P; i++) {
        for (j = 0; j < P; j++) {
            for (k = 0; k < P; k++) {
                w[IDX4(b, i, j, k)] = 0.;
                ur[IDX4(b, i, j, k)] = 0.;
                us[IDX4(b, i, j, k)] = 0.;
                ut[IDX4(b, i, j, k)] = 0.;

                for (l = 0; l < P; l++) {
                    ur[IDX4(b, i, j, k)] += dx[IDX2(i, l)] * u[IDX4(b, l, j, k)];
                    us[IDX4(b, i, j, k)] += dx[IDX2(k, l)] * u[IDX4(b, i, l, k)];
                    ut[IDX4(b, i, j, k)] += dx[IDX2(j, l)] * u[IDX4(b, i, j, l)];
                }

                w[IDX4(b, i, j, k)] = ur[IDX4(b, i, j, k)] * us[IDX4(b, i, j, k)] *
                ut[IDX4(b, i, j, k)];
            }
        }
    }
}
end = omp_get_wtime();
/* print result */
printf("collapse-clause: w[0]=%lf time=%lf\n", w[0], end - start);
return 0;
}

Compilation command:
icx -fiopenmp -fopenmp-targets=spir64 test_map_to.cpp

Run command:
OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 ./a.out

The amount of data transferred between host and device can be seen in LIBOMPTARGET_DEBUG=1 output
by grepping for "Libomptarget --> Moving". The output shows that the map (to: ) clauses for the arrays ur,
us, and ut cause the transfer of 262,144 bytes from host to device for each of the arrays:

$ grep "Libomptarget --> Moving" test_map_to.debug
Libomptarget --> Moving 262144 bytes (host:0x0000f0fff00000000) -> (target:0xff00000000000000)
Libomptarget --> Moving 262144 bytes (host:0x0000f0fff00000000) -> (target:0xff00000000000000)
Libomptarget --> Moving 262144 bytes (host:0x0000f0fff00000000) -> (target:0xff00000000000000)
Libomptarget --> Moving 262144 bytes (host:0x0000f0fff00000000) -> (target:0xff00000000000000)
Libomptarget --> Moving 262144 bytes (host:0x0000f0fff00000000) -> (target:0xff00000000000000)
Libomptarget --> Moving 262144 bytes (host:0x0000f0fff00000000) -> (target:0xff00000000000000)

These data transfers are wasteful because the arrays ur, us, and ut are simply used as temporary work arrays
on the device. A better approach would be to place the declarations of the arrays between the declare target
and end declare target directives. This indicates that the arrays are mapped to the device data environment, but no data transfers for these arrays occur unless the target update directive is used to manage the consistency of the arrays between host and device. This approach is illustrated in the following modified example.

**Listing 160:**
/examples/OpenMP/11_device_alloc/test_declare_target.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

#pragma omp declare target
double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
#pragma omp end declare target

int main(void) {
  double w[SIZE]; /* output */
  double u[SIZE], dx[P * P]; /* input */
  int b, i, j, k, l; /* loop counters */
  double start, end; /* timers */

  omp_set_default_device(0);

  /* dummy target region, so as not to measure startup time. */
  #pragma omp target
  {}

  /* initialize input with random values */
  srand(0);
  for (int i = 0; i < SIZE; i++)
    u[i] = scaled_rand();
  for (int i = 0; i < P * P; i++)
    dx[i] = scaled_rand();
```

(continues on next page)
In the above modified example, memory is allocated for arrays ur, us, and ut on the device, but no data transfers for these arrays take place. This is seen by grepping for "Libomptarget --> Moving" in LIBOMPTARGET_DEBUG=1 output. We no longer see the transfer of 262,144 bytes from host to device for each of the arrays:

```
$ grep "Libomptarget --> Moving" test_declare_target.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffe5564f000) -> (tgt:0xff00000000000000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffe5564f000) -> (tgt:0xff00000000000000)
```

An alternative approach for allocating memory on the device, without transferring any data between host and device, uses the map(alloc: ) clause instead of the map(to: ) clause, as shown below (lines 51-53).
# Listing 161:
/examples/OpenMP/11_device_alloc/test_map_alloc.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <math.h>
#include <omp.h>

#define P 16
#define BLOCKS 8
#define SIZE (BLOCKS * P * P * P)
#define MAX 100
#define scaled_rand() ((rand() % MAX) / (1.0 * MAX))
#define IDX2(i, j) (i * P + j)
#define IDX4(b, i, j, k) (b * P * P * P + i * P * P + j * P + k)

int main(void) {
  double w[SIZE]; /* output */
  double u[SIZE], dx[P * P]; /* input */
  double ur[SIZE], us[SIZE], ut[SIZE]; /* work arrays */
  int b, i, j, k, l; /* loop counters */
  double start, end; /* timers */

  omp_set_default_device(0);

  /* initialize input with random values */
  srand(0);
  for (int i = 0; i < SIZE; i++)
    u[i] = scaled_rand();
  for (int i = 0; i < P * P; i++)
    dx[i] = scaled_rand();

  start = omp_get_wtime();

  /* offload the kernel */
  #pragma omp target teams distribute parallel for simd simdlen(16) collapse(4) map(to:u[0:SIZE],dx[0:P*P]) 
```

(continues on next page)
In the above example, the `map(alloc: )` clauses for arrays `ur`, `us`, and `ut` cause memory to be allocated for `ur`, `us`, and `ut` on the device, and no data transfers occur – as in the declare target and end declare target case:

```
$ grep "Libomptarget --> Moving" test_map_alloc.debug
Libomptarget --> Moving 2048 bytes (hst:0x00007ffd46f256c0) -> (tgt:0xff00fffffffee0000)
Libomptarget --> Moving 262144 bytes (hst:0x00007ffd46ee56c0) -> (tgt:0xff00ffffffde0000)
Libomptarget --> Moving 262144 bytes (tgt:0xff00ffffffef0000) -> (hst:0x00007ffd46ea56c0)
```

The performance of the various versions when running on the particular GPU used (1-tile only) was as follows:

```
map(to: ) version : 0.001430 seconds
declare target / end declare target version : 0.000874 seconds
map(alloc: ) version : 0.000991 seconds
```
**14.7.3 Making Better Use of OpenMP Constructs**

**Reduce Synchronizations Using nowait**

If appropriate, use the nowait clause on the target construct to reduce synchronizations.

By default, there is an implicit barrier at the end of a target region, which ensures that the host thread that encountered the target construct cannot continue until the target region is complete.

Adding the nowait clause on the target construct eliminates this implicit barrier, so the host thread that encountered the target construct can continue even if the target region is not complete. This allows the target region to execute asynchronously on the device without requiring the host thread to idly wait for the target region to complete.

Consider the following example, which computes the product of two vectors, v1 and v2, in a parallel region (line 48). Half of the computations are performed on the host by the team of threads executing the parallel region. The other half of the computations are performed on the device. The master thread of the team launches a target region to do the computations on the device.

By default, the master thread of the team has to wait for the target region to complete before proceeding and participating in the computations (worksharing for loop) on the host.

**Listing 162: examples/OpenMP/04_target_nowait/test_target_no_nowait.cpp**

```c
// ==============================================================
// Copyright © 2022 Intel Corporation
// //
// SPDX-License-Identifier: MIT
// ==============================================================
// clang-format off
/
/*
 * This test is taken from OpenMP API 5.0.1 Examples (June 2020)
 * (4.13.2 nowait Clause on target Construct)
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define N 100000 // N must be even

void init(int n, float *v1, float *v2) {
  int i;
  for(i=0; i<n; i++){
    v1[i] = i * 0.25;
    v2[i] = i - 1.25;
  }
}
```

(continues on next page)
int main() {
    int i, n=N;
    float v1[N], v2[N], vxv[N];
    double start, end; // timers

    init(n, v1, v2);

    /* Dummy parallel and target regions, so as not to measure startup time. */
    #pragma omp parallel
    {
        #pragma omp master
        #pragma omp target
        {{}}
    }
    start=omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp master
        #pragma omp target teams distribute parallel for \
            map(to: v1[0:n/2]) \
            map(to: v2[0:n/2]) \
            map(from: vxv[0:n/2])
        for(i=0; i<n/2; i++){
            vxv[i] = v1[i]*v2[i];
        }
        /* Master thread will wait for target region to be completed 
           before proceeding beyond this point. */
        #pragma omp for
        for(i=n/2; i<n; i++) {
            vxv[i] = v1[i]*v2[i];
        }
        /* Implicit barrier at end of worksharing for. */
    }
    end=omp_get_wtime();

    printf("vxv[0]=%f, vxv[n-1]=%f, time=%lf\n", vxv[0], vxv[n-1], end-start);
    return 0;
}

Compilation command:
icx -fiopenmp -fopenmp-targets=spir64 test_target_no_nowait.cpp

Run command:
Performance could be improved if a nowait clause is specified on the target construct, so the master thread does not have to wait for the target region to complete and can proceed to work on the worksharing for loop. The target region is guaranteed to complete by the synchronization in the implicit barrier at the end of the worksharing for loop.

Listing 163: /examples/OpenMP/04_target_nowait/test_target_nowait.cpp

```c
// This test is taken from OpenMP API 5.0.1 Examples (June 2020)
// (4.13.2 nowait Clause on target Construct)

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define N 100000 // N must be even

void init(int n, float *v1, float *v2) {
    int i;

    for(i=0; i<n; i++){
        v1[i] = i * 0.25;
        v2[i] = i - 1.25;
    }
}

int main() {
    int i, n=N;
    float v1[N],v2[N],vxv[N];
    double start,end; // timers

    init(n, v1,v2);

    /* Dummy parallel and target (nowait) regions, so as not to measure startup time. */
    #pragma omp parallel
    {
        #pragma omp master
        #pragma omp target nowait
    }

    (continues on next page)```
The performance of the two versions when running on the particular ATS GPU used (1-tile only) was as follows:

```
no nowait version : 0.008220 seconds
nowait on target version : 0.002110 seconds
```

Fortran

The same nowait example shown above may be written in Fortran as follows.

```
!=============================================================
! Copyright © 2022 Intel Corporation
!
! SPDX-License-Identifier: MIT
!=============================================================

! This test is from OpenMP API 5.0.1 Examples (June 2020)
```

(continues on next page)
! (4.13.2 nowait Clause on target Construct)
!
subroutine init(n, v1, v2)
integer :: i, n
real :: v1(n), v2(n), vxv(n)
do i = 1, n
   v1(i) = i * 0.25
   v2(i) = i - 1.25
end do
end subroutine init

program test_target_nowait
use omp_lib
use iso_fortran_env
implicit none

integer, parameter :: NUM=100000 ! NUM must be even
real :: v1(NUM), v2(NUM), vxv(NUM)
integer :: n, i
real(kind=REAL64) :: start, end

n = NUM
call init(n, v1, v2)
!
! Dummy parallel and target (nowait) regions, so as not to measure
! startup time.
!$omp parallel
!$omp master
   !$omp target nowait
   !$omp end target
!$omp end master
!$omp end parallel
start=omp_get_wtime()
!
!$omp parallel
   !$omp master
      !$omp target teams distribute parallel do nowait &
         !$omp& map(to: v1(1:n/2)) &
         !$omp& map(to: v2(1:n/2)) &
         !$omp& map(from: vxv(1:n/2))
      do i = 1, n/2
         vxv(i) = v1(i)*v2(i)
      end do
   !$omp end master
!$omp end parallel

(continues on next page)
do i = n/2+1, n
  vxv(i) = v1(i)*v2(i)
end do

 !$omp end parallel
end=omp_get_wtime()

write(*,110) "vxv(1)=", vxv(1), ", vxv(n-1)=", vxv(n-1), ", time=", end-start
110 format (A, F10.6, A, F17.6, A, F10.6)
end program test_target_nowait

14.7.4 Memory Allocation

This section looks at various ways of allocating memory, and the types of allocations that are supported. A pointer on the host has the same size as a pointer on the device.

Host allocations are owned by the host and are intended to be allocated out of system memory. Host allocations are accessible by the host and all supported devices. Therefore, the same pointer to a host allocation may be used on the host and all supported devices. Host allocations are not expected to migrate between system memory and device-local memory. When a pointer to a host allocation is accessed on a device, data is typically sent over a bus, such as PCI-Express, that connects the device to the host.

Device allocations are owned by a specific device and are intended to be allocated out of device-local memory. Storage allocated can be read from and written to on that device, but is not directly accessible from the host or any other supported devices.

Shared allocations are accessible by the host and all supported devices. So the same pointer to a shared allocation may be used on the host and all supported devices, like in a host allocation. Shared allocations, however, are not owned by any particular device, but are intended to migrate between the host and one or more devices. This means that accesses on a device, after the migration has occurred, happen from much faster device-local memory instead of remotely accessing system memory though the higher-latency bus connection.

Shared-system allocations are a sub-class of shared allocations, where the memory is allocated by a system allocator (such as malloc or new) rather than by an allocation API (such as the OpenMP memory allocation API). Shared-system allocations have no associated device; they are inherently cross-device. Like other shared allocations, Shared-system allocations are intended to migrate between the host and supported devices, and the same pointer to a shared-system allocation may be used on the host and all supported devices.

Note:

- Currently, shared-system allocations are not supported on ATS and PVC systems. However, shared allocations where memory is allocated by an allocation API are supported on ATS and PVC.

The following table summarizes the characteristics of the various types of memory allocation.
Host allocations offer wide accessibility (can be accessed directly from the host and all supported devices), but have potentially high per-access costs because data is typically sent over a bus such as PCI Express*. Shared allocations also offer wide accessibility, but the per-access costs are potentially lower than host allocations, because data is migrated to the accessing device.

Device allocations have access limitations (cannot be accessed directly from the host or other supported devices), but offer higher performance because accesses are to device-local memory.

**OpenMP Runtime Routines for Memory Allocation**

Intel compilers support a number of OpenMP runtime routines for performing memory allocations. These routines are shown in the table below.

<table>
<thead>
<tr>
<th>OpenMP memory allocation routine</th>
<th>Intel extension?</th>
<th>Type of allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_target_alloc</td>
<td>No</td>
<td>Device</td>
</tr>
<tr>
<td>omp_target_alloc_device</td>
<td>Yes</td>
<td>Device</td>
</tr>
<tr>
<td>omp_target_alloc_host</td>
<td>Yes</td>
<td>Host</td>
</tr>
<tr>
<td>omp_target_alloc_shared</td>
<td>Yes</td>
<td>Shared</td>
</tr>
</tbody>
</table>

Note that the three routines `omp_target_alloc_device`, `omp_target_alloc_host`, and `omp_target_alloc_shared` are Intel extensions to the OpenMP specification.

The following examples use the above OpenMP memory allocation routines. Compare those to the ones using map clauses.

For more information about memory allocation, see:

- Data Parallel C++, by James Reinders et al.
- SYCL 2020 Specification
- oneAPI Level Zero Specification
- The DPC++ part of this guide
Using the **map** Clause

The first example uses map clauses to allocate memory on a device and copy data between the host and the device.

In the following example, arrays A, B, and C are allocated in system memory by calling the C/C++ standard library routine, malloc.

The target construct on line 58 is the main kernel that computes the values of array C on the device. The `map(to/from: C[0:length])` clause is specified on this target construct since the values of C need to be transferred from the host to the device before the computation, and from the device to the host at the end of the computation. The `map(to: A[0:length], B[0:length])` is specified for arrays A and B since the values of these arrays need to be transferred from the host to the device, and the device only reads these values. Under the covers, the map clauses cause storage for the arrays to be allocated on the device and data to be copied from the host to the device, and vice versa.

**Listing 165:** /examples/OpenMP/21_omp_target_alloc/test_target_map.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length 64*1024*1024

int main(void)
{
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays on the host using plain malloc()
    A = (double *) malloc(bytes);
    if (A == NULL){
        printf("ERROR: Cannot allocate space for A using plain malloc().\n");
        exit(1);
    }

    B = (double *) malloc(bytes);

    // (continues on next page)
```
if (B == NULL){
    printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
    exit(1);
}

C = (double *) malloc(bytes);
if (C == NULL){
    printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
    exit(1);
}

// Initialize the arrays
#pragma omp parallel for
for (size_t i=0; i<length; i++) {
    A[i] = 2.0;
    B[i] = 2.0;
    C[i] = 0.0;
}

// Perform the computation
nstream_time = omp_get_wtime();
for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for \ 
    map(to: A[0:length], B[0:length]) \ 
    map(tofrom: C[0:length])
    for (size_t i=0; i<length; i++) {
        C[i] += A[i] + scalar * B[i];
    }
}
nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results
double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}
double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}
free(A);
free(B);  
free(C);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
   printf("Failed Validation on output array\n"
   "     Expected checksum: %lf\n"
   "     Observed checksum: %lf\n"
   "ERROR: solution did not validate\n", cr, asum);
   return 1;
} else {
   printf("Solution validates\n");
   double avgtime = nstream_time/iterations;
   printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}
return 0;
}

Compilation command:

icx -fiopenmp -fopenmp-targets=spir64 test_target_map.cpp

Run command:

OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 ./a.out

The map clauses on the target construct inside the iterations loop cause data (values of A, B, C) to be transferred from the host to the device at the beginning of each target region, and cause data (values of C) to be transferred from the device to the host at the end of each target region. These data transfers incur a significant performance overhead. A better approach using map clauses would be to put the whole iterations loop inside a target data construct with the map clauses. This causes the transfers to occur once at the beginning of the iterations loop, and another time at the end of the iterations loop. The modified example using target data and map clauses is shown below.

Listing 166: examples/OpenMP/21_omp_target_alloc/test_target_map2.cpp

//===============================================
// Copyright © 2022 Intel Corporation
//
// SPDX-License-Identifier: MIT
//===============================================
// clang-format off
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

//===============================================
// Copyright © 2022 Intel Corporation
//
// SPDX-License-Identifier: MIT
//===============================================
// clang-format off
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

(continues on next page)
#define iterations 100
#define length 64*1024*1024

int main(void)
{
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays on the host using plain malloc()
    A = (double *) malloc(bytes);
    if (A == NULL){
        printf(" ERROR: Cannot allocate space for A using plain malloc().\n");
        exit(1);
    }
    B = (double *) malloc(bytes);
    if (B == NULL){
        printf(" ERROR: Cannot allocate space for B using plain malloc().\n");
        exit(1);
    }
    C = (double *) malloc(bytes);
    if (C == NULL){
        printf(" ERROR: Cannot allocate space for C using plain malloc().\n");
        exit(1);
    }

    // Initialize the arrays
    #pragma omp parallel for
    for (size_t i=0; i<length; i++) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    // Perform the computation
    nstream_time = omp_get_wtime();
    #pragma omp target data map(to: A[0:length], B[0:length])
    #pragma omp target data map(tofrom: C[0:length])
    {
        for (int iter = 0; iter<iterations; iter++) {
            #pragma omp target teams distribute parallel for
            for (size_t i=0; i<length; i++) {
                C[i] += A[i] + scalar * B[i];
            }
        }
    }
}(continues on next page)
nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}
free(A);
free(B);
free(C);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
            " Expected checksum: %lf\n"
            " Observed checksum: %lf\n"
            "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}
return 0;

omp_target_alloc

Next, the example above is modified to use device allocations instead of map clauses. Storage for arrays A, B, and C is directly allocated on the device by calling the OpenMP runtime routine omp_target_alloc. The routine takes two arguments: the number of bytes to allocate on the device, and the number of the device on which to allocate the storage. The routine returns a device pointer that references the device address of the storage allocated on the device. If the call to omp_target_alloc returns NULL, then this indicates that the allocation was not successful.
To access the allocated memory in a target construct, the device pointer returned by a call to `omp_target_alloc` is listed in an `is_device_ptr` clause on the target construct. This ensures that there is no data transfer before and after kernel execution since the kernel operates on data that is already on the device.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C on the device.

Listing 167: `/examples/OpenMP/21_omp_target_alloc/test_omp_target_alloc.cpp`

```c
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================================
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length 64*1024*1024

int main(void)
{
    int device_id = omp_get_default_device();
    size_t bytes = length*sizeof(double);
    double *__restrict A;
    double *__restrict B;
    double *__restrict C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays in device memory
    A = (double *) omp_target_alloc(bytes, device_id);
    if (A == NULL){
        printf(" ERROR: Cannot allocate space for A using omp_target_alloc().\n");
        exit(1);
    }
    B = (double *) omp_target_alloc(bytes, device_id);
    if (B == NULL){
        printf(" ERROR: Cannot allocate space for B using omp_target_alloc().\n");
        exit(1);
    }
    C = (double *) omp_target_alloc(bytes, device_id);
    if (C == NULL){
        printf(" ERROR: Cannot allocate space for C using omp_target_alloc().\n");
        exit(1);
    }
    (continues on next page)
```
// Initialize the arrays

#pragma omp target teams distribute parallel for \
     is_device_ptr(A,B,C)
for (size_t i=0; i<length; i++) {
    A[i] = 2.0;
    B[i] = 2.0;
    C[i] = 0.0;
}

// Perform the computation 'iterations' number of times

nstream_time = omp_get_wtime();
for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for \
     is_device_ptr(A,B,C)
    for (size_t i=0; i<length; i++) {
        C[i] += A[i] + scalar * B[i];
    }
}

nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp target teams distribute parallel for reduction(+:asum) \ 
     map(tofrom: asum) is_device_ptr(C)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}

omp_target_free(A, device_id);
omp_target_free(B, device_id);
omp_target_free(C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
           " Expected checksum: %lf\n"
           " Observed checksum: %lf\n"
"ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}

return 0;
}

Notes:

- When calling `omp_target_alloc`, the device number specified must be one of the supported devices, other than the host device. This will be the device on which storage will be allocated.

- Since the arrays A, B, and C are not accessible from the host, the initialization of the arrays, kernel execution, and summation of elements of C all need to be done inside OpenMP target regions.

- A device allocation can only be accessed by the device specified in the `omp_target_alloc` call, but may be copied to memory allocated on the host or other devices by calling `omp_target_memcpy`.

`omp_target_alloc_device`

The Intel extension `omp_target_alloc_device` is similar to `omp_target_alloc`. It is also called with two arguments: the number of bytes to allocate on the device, and the number of the device on which to allocate the storage. The routine returns a device pointer that references the device address of the storage allocated on the device. If the call to `omp_target_alloc_device` returns NULL, then this indicates that the allocation was not successful.

The above `omp_target_alloc` example can be rewritten using `omp_target_alloc_device` by simply replacing the call to `omp_target_alloc` with a call to `omp_target_alloc_device` as shown below.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C on the device.

Listing 168: /examples/OpenMP/21_omp_target_alloc/test_-omp_target_alloc_device.cpp

```
// Allocate arrays in device memory
A = (double *) omp_target_alloc_device(bytes, device_id);
if (A == NULL){
    printf(" ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
    exit(1);
}

B = (double *) omp_target_alloc_device(bytes, device_id);
if (B == NULL){
    printf(" ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
    exit(1);
}
```
```c
C = (double *) omp_target_alloc_device(bytes, device_id);
if (C == NULL){
    printf("ERROR: Cannot allocate space for C using omp_target_alloc_device().\n");
    exit(1);
}
```

Note:

- All of the above Notes that apply to `omp_target_alloc` also apply to `omp_target_alloc_device`.

**omp_target_alloc_host**

The above example can also be rewritten by doing a host allocation for A, B, and C. This allows the memory to be accessible to the host and all supported devices.

In the following modified example, the `omp_target_alloc_host` runtime routine (an Intel extension) is called to allocate storage for each of the arrays A, B, and C. The routine takes two arguments: the number of bytes to allocate, and a device number. The device number must be one of the supported devices, other than the host device. The routine returns a pointer to a storage location in host memory. If the call to `omp_target_alloc_host` returns NULL, this indicates that the allocation was not successful.

Note the directive requires `unified_address` is specified at the top of the program. This requires that the implementation guarantee that all devices accessible through OpenMP API routines and directives use a unified address space. In this address space, a pointer will always refer to the same location in memory from all devices, and the `is_device_ptr` clause is not necessary to obtain device addresses from device pointers for use inside target regions. When using Intel compilers, the `requires unified_address` directive is actually not needed, since unified address space is guaranteed by default. However, for portability the code includes the directive.

The pointer returned by a call to `omp_target_alloc_host` can be used to access the storage from the host and all supported devices. No map clauses and no `is_device_ptr` clauses are needed on a target construct to access the memory from a device since a unified address space is used.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C.

**Listing 169:** /examples/OpenMP/21_omp_target_alloc/test_-omp_target_alloc_host.cpp

```
1  //==============================================================================
2  // Copyright © 2022 Intel Corporation
3  //
4  // SPDX-License-Identifier: MIT
5  // =============================================================================
6  // clang-format off
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <stdint.h>
10 #include <math.h>
11
12 //==============================================================================
13 // Copyright © 2022 Intel Corporation
14 //
15 // SPDX-License-Identifier: MIT
16 // =============================================================================
17 // clang-format off
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <stdint.h>
21 #include <math.h>
```
#include <omp.h>

#pragma omp requires unified_address

#define iterations 100
#define length 64*1024*1024

int main(void)
{
  int device_id = omp_get_default_device();
  size_t bytes = length*sizeof(double);
  double *__restrict A;
  double *__restrict B;
  double *__restrict C;
  double scalar = 3.0;
  double nstream_time = 0.0;

  // Allocate arrays in host memory
  A = (double *) omp_target_alloc_host(bytes, device_id);
  if (A == NULL){
    printf(" ERROR: Cannot allocate space for A using omp_target_alloc_host().\n");
    exit(1);
  }

  B = (double *) omp_target_alloc_host(bytes, device_id);
  if (B == NULL){
    printf(" ERROR: Cannot allocate space for B using omp_target_alloc_host().\n");
    exit(1);
  }

  C = (double *) omp_target_alloc_host(bytes, device_id);
  if (C == NULL){
    printf(" ERROR: Cannot allocate space for C using omp_target_alloc_host().\n");
    exit(1);
  }

  // Initialize the arrays
  #pragma omp parallel for
  for (size_t i=0; i<length; i++) {
    A[i] = 2.0;
    B[i] = 2.0;
    C[i] = 0.0;
  }

  // Perform the computation
  nstream_time = omp_get_wtime();
  for (int iter = 0; iter<iterations; iter++) {
    #pragma omp target teams distribute parallel for
for (size_t i=0; i<length; i++) {
    C[i] += A[i] + scalar * B[i];
}

nstream_time = omp_get_wtime() - nstream_time;

// Validate and output results

double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}

double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(C[i]);
}

omp_target_free(A, device_id);
omp_target_free(B, device_id);
omp_target_free(C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
        "   Expected checksum: %lf\n"
        "   Observed checksum: %lf\n"
        "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}

return 0;

Notes:

- When calling omp_target_alloc_host, the device number specified must be one of the supported devices, other than the host device.

- Since the arrays A, B, and C are accessible from the host and device, the initialization of the arrays and summation of elements of C may be done either on the host (outside of a target construct) or on the device (inside a target construct).
• ATS and PVC do not support atomic operations (or algorithms that use atomic operations, such as some reductions) on host allocations (i.e., memory allocated via `omp_target_alloc_host`). Use atomic operations on memory allocated via `omp_target_alloc_device`, instead.

`omp_target_alloc_shared`

The above example is modified so that shared allocations are used instead of host allocations. The `omp_target_alloc_shared` runtime routine is called to allocate storage for each of arrays A, B, and C. The routine takes two arguments: the number of bytes to allocate on the device, and a device number. The device number must be one of the supported devices, other than the host device. The routine returns a pointer to a storage location in shared memory. If the call to `omp_target_alloc_shared` returns NULL, then this indicates that the allocation was not successful.

Note the `requires unified_address` directive is specified at the top of the program, for portability.

The pointer returned by a call to `omp_target_alloc_shared` can be used to access the storage from the host and all supported devices. No map clauses and `no is_device_ptr` clauses are needed on a target construct to access the memory from a device since a unified address space is used.

At the end of the program, the runtime routine `omp_target_free` is used to deallocate the storage for A, B, and C.

```c
// Allocate arrays in shared memory

A = (double *) omp_target_alloc_shared(bytes, device_id);
if (A == NULL){
    printf(" ERROR: Cannot allocate space for A using omp_target_alloc_shared().\n");
    exit(1);
}

B = (double *) omp_target_alloc_shared(bytes, device_id);
if (B == NULL){
    printf(" ERROR: Cannot allocate space for B using omp_target_alloc_shared().\n");
    exit(1);
}

C = (double *) omp_target_alloc_shared(bytes, device_id);
if (C == NULL){
    printf(" ERROR: Cannot allocate space for C using omp_target_alloc_shared().\n");
    exit(1);
}
```

**Notes:**

• When calling `omp_target_alloc_shared`, the device number specified must be one of the supported devices, other than the host device.

• Since the arrays are accessible from the host and device, the initialization and verification may be done either on the host or on the device (inside a `target` construct).

• Concurrent access from host and device to memory allocated via `omp_target_alloc_shared` is not supported.
omp_target_memcpy

The following example shows how the runtime routine omp_target_memcpy may be used to copy memory from host to device, and from device to host. First arrays h_A, h_B, and h_C are allocated in system memory using plain malloc, and then initialized. Corresponding arrays d_A, d_B, and d_C are allocated on the device using omp_target_alloc.

Before the start of the target construct on line 104, the values in h_A, h_B, and h_C are copied to d_A, d_B, and d_C by calling omp_target_memcpy. After the target region, new d_C values computed on the device are copied to h_C by calling omp_target_memcpy.

Listing 171: /examples/OpenMP/21_omp_target_alloc/test_omp_target_memcpy.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define iterations 100
#define length 64*1024*1024

int main(void)
{
    int device_id = omp_get_default_device();
    int host_id = omp_get_initial_device();
    size_t bytes = length*sizeof(double);
    double *__restrict h_A;
    double *__restrict h_B;
    double *__restrict h_C;
    double *__restrict d_A;
    double *__restrict d_B;
    double *__restrict d_C;
    double scalar = 3.0;
    double nstream_time = 0.0;

    // Allocate arrays h_A, h_B, and h_C on the host using plain malloc()
    h_A = (double *) malloc(bytes);
    if (h_A == NULL){
        printf(" ERROR: Cannot allocate space for h_A using plain malloc().\n");
        exit(1);
    }
    h_B = (double *) malloc(bytes);
    if (h_B == NULL){
```
printf(" ERROR: Cannot allocate space for h_B using plain malloc().\n");
exit(1);
}

h_C = (double *) malloc(bytes);
if (h_C == NULL){
    printf(" ERROR: Cannot allocate space for h_C using plain malloc().\n");
    exit(1);
}

// Allocate arrays d_A, d_B, and d_C on the device using omp_target Alloc()
d_A = (double *) omp_target Alloc(bytes, device_id);
if (d_A == NULL){
    printf(" ERROR: Cannot allocate space for d_A using omp_target Alloc().\n");
    exit(1);
}

d_B = (double *) omp_target Alloc(bytes, device_id);
if (d_B == NULL){
    printf(" ERROR: Cannot allocate space for d_B using omp_target Alloc().\n");
    exit(1);
}

d_C = (double *) omp_target Alloc(bytes, device_id);
if (d_C == NULL){
    printf(" ERROR: Cannot allocate space for d_C using omp_target Alloc().\n");
    exit(1);
}

// Initialize the arrays on the host
#pragma omp parallel for
for (size_t i=0; i<length; i++) {
    h_A[i] = 2.0;
    h_B[i] = 2.0;
    h_C[i] = 0.0;
}

// Call omp_target_memcpy() to copy values from host to device
int rc = 0;
rc = omp_target_memcpy(d_A, h_A, bytes, 0, 0, device_id, host_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
    exit(1);
}
rc = omp_target_memcpy(d_B, h_B, bytes, 0, 0, device_id, host_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(B) returned %d\n", rc);
exit(1);
}

rc = omp_target_memcpy(d_C, h_C, bytes, 0, 0, device_id, host_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(C) returned %d\n", rc);
    exit(1);
}

// Perform the computation
nstream_time = omp_get_wtime();
for (int iter = 0; iter<iterations; iter++) {
    //pragma omp target teams distribute parallel for \\is_device_ptr(d_A,d_B,d_C)
    for (size_t i=0; i<length; i++) {
        d_C[i] += d_A[i] + scalar * d_B[i];
    }
}
nstream_time = omp_get_wtime() - nstream_time;
// Call omp_target_memcpy() to copy values from device to host
rc = omp_target_memcpy(h_C, d_C, bytes, 0, 0, host_id, device_id);
if (rc) {
    printf("ERROR: omp_target_memcpy(A) returned %d\n", rc);
    exit(1);
}

// Validate and output results
double ar = 2.0;
double br = 2.0;
double cr = 0.0;
for (int iter = 0; iter<iterations; iter++) {
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }
}
double asum = 0.0;
#pragma omp parallel for reduction(+:asum)
for (size_t i=0; i<length; i++) {
    asum += fabs(h_C[i]);
}
free(h_A);
free(h_B);
free(h_C);
omp_target_free(d_A, device_id);
omp_target_free(d_B, device_id);
omp_target_free(d_C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
    " Expected checksum: %lf\n"
    " Observed checksum: %lf\n"
    "ERROR: solution did not validate\n", cr, asum);
    return 1;
} else {
    printf("Solution validates\n");
    double avgtime = nstream_time/iterations;
    printf("Checksum = %lf; Avg time (s): %lf\n", asum, avgtime);
}
return 0;

Performance Considerations

In the above examples (using the map clause, omp_target_alloc, omp_target_alloc_device, omp_target_alloc_host, omp_target_alloc_shared, omp_target_memcpy), the main kernel is the target construct that computes the values of array C. To get more accurate timings, this target construct is enclosed in a loop, so the offload happens iterations number of times (where iterations = 100). The average kernel time is computed by dividing the total time taken by the iterations loop by 100.

Listing 172: /examples/OpenMP/21_omp_target_alloc/test_omp_target_alloc.cpp

Listing 173: /examples/OpenMP/21_omp_target_alloc/test_omp_target_alloc.debug

LIBOMPTARGET_DEBUG=1 output shows that all the above examples have the same ND_range partitioning.
The following table shows the average times taken by the kernel in the various versions when running on the particular GPU used (1-tile only).

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>0.183604</td>
</tr>
<tr>
<td>map + target data</td>
<td>0.012757</td>
</tr>
<tr>
<td>omp_target_alloc</td>
<td>0.002501</td>
</tr>
<tr>
<td>omp_target_alloc_device</td>
<td>0.002499</td>
</tr>
<tr>
<td>omp_target_alloc_host</td>
<td>0.074412</td>
</tr>
<tr>
<td>omp_target_alloc_shared</td>
<td>0.012491</td>
</tr>
<tr>
<td>omp_target_memcpy</td>
<td>0.011072</td>
</tr>
</tbody>
</table>

The above performance numbers show that the map version is the slowest version (0.183604 seconds). This is because of the data transfers that occur at the beginning and end of each kernel launch. The main kernel is launched 100 times. At the beginning of each kernel launch, storage for arrays A, B and C is allocated on the device, and the values of these arrays are copied from the host to the device. At the end of the kernel, the values of array C are copied from the device to the host. Putting the whole iterations loop inside a target data construct with map clauses reduced the runtime to 0.012757 seconds, because the transfers occur once at the launch of the first kernel in the iterations loop, and again after the last kernel in that loop.

The omp_target_alloc and omp_target_alloc_device versions have the best performance (0.002501 and 0.002499 seconds, respectively). In these versions, storage for A, B, and C is allocated directly in device memory, so accesses on the device happen from device-local memory. This is a useful model for applications that use scratch arrays on the device side. These arrays never need to be accessed on the host. In such cases, the recommendation is to allocate the scratch arrays on the device and not worry about data transfers, as illustrated in this example.

The omp_target_alloc_shared version also performs well, but is somewhat slower (0.012491 seconds). In this version, storage for A, B, and C is allocated in shared memory. So the data can migrate between the host and the device. There is the overhead of migration but, after migration, accesses on the device happen from much faster device-local memory. In this version, the initialization of the arrays happens on the host. At the first kernel launch, the arrays are migrated to the device, and the kernels access the arrays locally on the device. Finally, before the host performs the reduction computation, the entire C array is migrated back to the host.

The omp_target_alloc_host version (0.074412 seconds) takes almost 6x more time than the omp_target_alloc_shared version. This is because data allocated in host memory does not migrate from the host to the device. When the kernel tries to access the data, the data is typically sent over a bus, such as PCI Express, that connects the device to the host. This is slower than accessing local device memory. If the device accesses only a small part of an array infrequently, then that array may be allocated in host memory using omp_target_alloc_host. However, if the array is accessed frequently on the device side, then it should be kept in device memory. Keeping the data in host memory and accessing it over the PCI will degrade performance.
Finally, a note regarding data transfers: The amount of data transferred in the map version can be seen in LIBOMPTARGET_DEBUG=1 output by grepping for "Libomptarget --> Moving". Notice that each launch of the main kernel yields the following data transfers:

```
$ grep "Libomptarget --> Moving" test_target_map.debug
Libomptarget --> Moving 536870912 bytes (hst:0x00007f1a5fc8b010) -> (tgt:0xff00000002000000)
Libomptarget --> Moving 536870912 bytes (hst:0x00007f1a9fc8d010) -> (tgt:0xff00000004020000)
Libomptarget --> Moving 536870912 bytes (hst:0x00007f1a7fc8c010) -> (tgt:0xff00000008040000)
Libomptarget --> Moving 536870912 bytes (tgt:0xff00000002000000) -> (hst:0x00007f1a5fc8b010)
```

On the other hand, data transfers in the omp_target_alloc... versions are handled by a lower layer of the run-time system. So grepping for "Libomptarget --> Moving" in LIBOMPTARGET_DEBUG=1 output for these versions will not show the data transfers that took place.

**Fortran**

The Fortran version of the example using target data and map clauses is shown below.

```
!=================================================================================
! Copyright © 2022 Intel Corporation
!
! SPDX-License-Identifier: MIT
!=================================================================================
program main
use iso_fortran_env
use omp_lib
implicit none

integer, parameter :: iterations=100
integer, parameter :: length=64*1024*1024
real(kind=REAL64), parameter :: epsilon=1.D-8
real(kind=REAL64), allocatable :: A(:)
real(kind=REAL64), allocatable :: B(:)
real(kind=REAL64), allocatable :: C(:)
real(kind=REAL64) :: scalar=3.0
real(kind=REAL64) :: ar, br, cr, asum
real(kind=REAL64) :: nstream_time, avgtime
integer :: err, i, iter

! Allocate arrays on the host using plain allocate
allocate( A(length), stat=err )
if (err .ne. 0) then
    print *, "Allocation of A returned ", err
    stop 1
endif
```

(continues on next page)
allocate( B(length), stat=err )
if (err .ne. 0) then
  print *, "Allocation of B returned ", err
  stop 1
endif

allocate( C(length), stat=err )
if (err .ne. 0) then
  print *, "Allocation of C returned ", err
  stop 1
endif

!
! Initialize the arrays
!
!$omp parallel do
do i = 1, length
  A(i) = 2.0
  B(i) = 2.0
  C(i) = 0.0
end do
!
! Perform the computation
!
nstream_time = omp_get_wtime()
!$omp target data  map(to: A, B) map(tofrom: C)
do iter = 1, iterations
  !$omp target teams distribute parallel do
do i = 1, length
    C(i) = C(i) + A(i) + scalar * B(i)
  end do
end do
!
!$omp end target data
nstream_time = omp_get_wtime() - nstream_time
!
! Validate and output results
ar = 2.0
br = 2.0
cr = 0.0
do iter = 1, iterations
do i = 1, length
  cr = cr + ar + scalar * br
end do
end do
asum = 0.0
!$omp parallel do reduction(+:asum)
do i = 1, length
   asum = asum + abs(C(i))
end do
if (abs(cr - asum)/asum > epsilon) then
   write(*,110) "Failed Validation on output array: Expected =", cr, ", Observed =", asum
else
   avgtime = nstream_time/iterations
   write(*,120) "Solution validates: Checksum =", asum, ", Avg time (s) =", avgtime
endif
110 format (A, F20.6, A, F20.6)
120 format (A, F20.6, A, F10.6)
deallocate(A)
deallocate(B)
deallocate(C)
end program main

The Fortran version of the example using omp_target_alloc_device is shown below. In this example, allocate directives, with the allocator omp_target_device_mem_alloc, are used to allocate arrays A, B, and C on the device. The use_device_addr(A, B, C) clause is used on the target data directive (line 37) to indicate that the arrays have device addresses, and these addresses should be used in the target region.

Listing 175: /examples/OpenMP/21_omp_target_alloc/test_-_omp_target_alloc_device_ff90

!===================================================================================================================
! Copyright © 2022 Intel Corporation
!
! SPDX-License-Identifier: MIT
!===================================================================================================================
program main
use iso_fortran_env
use omp_lib
implicit none
integer, parameter :: iterations=100
integer, parameter :: length=64*1024*1024
real(kind=REAL64), parameter :: epsilon=1.D-8
real(kind=REAL64), allocatable :: A(:)
real(kind=REAL64), allocatable :: B(:)
real(kind=REAL64), allocatable :: C(:)
real(kind=REAL64) :: scalar=3.0
real(kind=REAL64) :: ar, br, cr, asum
real(kind=REAL64) :: nstream_time, avgtime
integer :: i, iter
(continues on next page)
! Allocate arrays in device memory

!$omp allocate allocator(omp_target_device_mem_alloc)
allocate(A(length))

!$omp allocate allocator(omp_target_device_mem_alloc)
allocate(B(length))

!$omp allocate allocator(omp_target_device_mem_alloc)
allocate(C(length))

! Begin target data

!$omp target data use_device_addr(A, B, C)

! Initialize the arrays

!$omp target teams distribute parallel do
do i = 1, length
   A(i) = 2.0
   B(i) = 2.0
   C(i) = 0.0
end do

! Perform the computation

nstream_time = omp_get_wtime()
do iter = 1, iterations
   !$omp target teams distribute parallel do
do i = 1, length
      C(i) = C(i) + A(i) + scalar * B(i)
   end do
end do
nstream_time = omp_get_wtime() - nstream_time

! Validate and output results

ar = 2.0
br = 2.0
cr = 0.0
do iter = 1, iterations
do i = 1, length
   cr = cr + ar + scalar * br
end do
end do
```fortran
asum = 0.0
!$omp target teams distribute parallel do reduction(+:asum) &
!$omp map(tofrom: asum)
do i = 1, length
   asum = asum + abs(C(i))
end do
!
! End target data
$omp end target data
if (abs(cr - asum)/asum > epsilon) then
   write(*,110) "Failed Validation on output array: Expected =", cr, ", Observed =", asum
else
   avgtime = nstream_time/iterations
   write(*,120) "Solution validates: Checksum =", asum, ", Avg time (s) =", avgtime
endif
110 format (A, F20.6, A, F20.6)
120 format (A, F20.6, A, F10.6)
derialocate(A)
derialocate(B)
derialocate(C)
end program main
```

### 14.7.5 Clauses: is_device_ptr, use_device_ptr, has_device_addr, use_device_addr

The OpenMP clauses is_device_ptr, use_device_ptr, has_device_addr, and use_device_addr can be used to convey information about variables referenced in target, target data, or dispatch constructs. These clauses are described as follows.

**is_device_ptr**

The is_device_ptr clause appears on a target or dispatch directive. It indicates that the list items are device pointers. So each list item is privatized inside the construct and the new list item is initialized to the device address to which the original list item refers.

- In C, each list item should be of type pointer or array.
- In C++, each list item should be of type pointer, array, reference to pointer, or reference to array.
- In Fortran, each list item should be of type C_PTR.

The following C/C++ example illustrates the use of the is_device_ptr clause. The `omp_target_alloc_device` routine allocates memory on the device and returns a device pointer for that memory which is saved in the host variable `arr_device`. On the target directive, we use the `is_device_ptr(arr_device)` clause to indicate that `arr_device` points to device memory. So inside the target construct `arr_device` is privatized and initialized to the device address to which `arr_device` refers.
Listing 176: /examples/OpenMP/24_device_ptr_addr-_clauses/c_is_device_ptr_01.cpp

```c
//==============================================================
// Copyright © 2022 Intel Corporation
// SPDX-License-Identifier: MIT
//==============================================================
// clang-format off
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define N 100

int main(void)
{
    int *arr_host = NULL;
    int *arr_device = NULL;

    arr_host = (int *) malloc(N * sizeof(int));
    arr_device = (int *) omp_target_alloc_device(N * sizeof(int),
        omp_get_default_device());

    #pragma omp target is_device_ptr(arr_device) map(from: arr_host[0:N])
    {
        for (int i = 0; i < N; ++i) {
            arr_device[i] = i;
            arr_host[i] = arr_device[i];
        }
    }

    printf("%d, %d, %d \n", arr_host[0], arr_host[N/2], arr_host[N-1]);
}
```

use_device_ptr

The use_device_ptr clause appears on a target data directive. It indicates that each list item is a pointer to an object that has corresponding storage on the device or is accessible on the device.

If a list item is a pointer to an object that is mapped to the device, then references to the list item in the construct are converted to references to a device pointer that is local to the construct and that refers to the device address of the corresponding object.

If the list item does not point to a mapped object, it must contain a valid device address, and the list item references are converted to references to a local device pointer that refers to this device address.

Each list item must be a pointer for which the value is the address of an object that has corresponding storage in the device data environment or is accessible on the target device.

In C, each list item should be of type pointer or array.
In C++, each list item should be of type pointer, array, reference to pointer, or reference to array.

In Fortran, each list item should be of type C_PTR.

The following C/C++ example illustrates the use of the use_device_ptr clause. The omp_target_alloc_device routine is called three times to allocate memory on the device. The addresses of the memory allocated is saved in the pointer variables A, B, and C on the host. We use the use_device_ptr(A, B, C) clause on the target data directive to indicate that A, B, and C contain valid device addresses.

Listing 177: /examples/OpenMP/24_device_ptr_addr_clauses/c_use_device_ptr_01.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <omp.h>

#define length 65536

int main(void) {
    int device_id = omp_get_default_device();
    size_t bytes = length*sizeof(double);
    double * __restrict A;
    double * __restrict B;
    double * __restrict C;
    double scalar = 3.0;
    double ar;
    double br;
    double cr;
    double asum;

    // Allocate arrays in device memory
    A = (double *) omp_target_alloc_device(bytes, device_id);
    if (A == NULL){
        printf("ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
        exit(1);
    }

    B = (double *) omp_target_alloc_device(bytes, device_id);
    if (B == NULL){
        printf("ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
        exit(1);
    }

    // (continues on next page)
}
C = (double *) omp_target_alloc_device(bytes, device_id);
if (C == NULL){
    printf(" ERROR: Cannot allocate space for C using omp_target_alloc_device().\n");
    exit(1);
}

#pragma omp target data use_device_ptr(A,B,C)
{
    // Initialize the arrays

    #pragma omp target teams distribute parallel for
    for (size_t i=0; i<length; i++) {
        A[i] = 2.0;
        B[i] = 2.0;
        C[i] = 0.0;
    }

    // Perform the computation

    #pragma omp target teams distribute parallel for
    for (size_t i=0; i<length; i++) {
        C[i] += A[i] + scalar * B[i];
    }

    // Validate and output results

    ar = 2.0;
    br = 2.0;
    cr = 0.0;
    for (int i=0; i<length; i++) {
        cr += ar + scalar * br;
    }

    asum = 0.0;
    #pragma omp target teams distribute parallel for reduction(+:asum)
    for (size_t i=0; i<length; i++) {
        asum += fabs(C[i]);
    }
}

omp_target_free(A, device_id);
omp_target_free(B, device_id);
omp_target_free(C, device_id);

double epsilon=1.e-8;
if (fabs(cr - asum)/asum > epsilon) {
    printf("Failed Validation on output array\n"
        " Expected checksum: %lf\n"
        " Observed checksum: %lf\n"
        "ERROR: solution did not validate\n", cr, asum);
}
has_device_addr

The has_device_addr clause appears on a target directive. It indicates that the list items already have valid device addresses, and therefore may be directly accessed from the device.

Each list item must have a valid device address for the device data environment. It can be on any type, including an array section.

The has_device_addr clause is especially useful in Fortran, because it can be used with list items of any type (not just C_PTR) to indicate that the list items have device addresses.

The following Fortran example illustrates the use of the has_device_addr clause. In the example, the three arrays A, B, and C are allocated on the device. When the arrays are referenced in a target region, we use the has_device_addr(A, B, C) clause to indicate that A, B, and C already have device addresses.

Listing 178: /examples/OpenMP/24_device_ptr_addr-_clauses/f_has_device_addr_01.f90

```fortran
!=============================================================================
! Copyright © 2022 Intel Corporation
!
! SPDX-License-Identifier: MIT
!=============================================================================
program main
  use iso_fortran_env
  use omp_lib
  implicit none

  integer, parameter :: iterations=1000
  integer, parameter :: length=64*1024*1024
  real(kind=REAL64), parameter :: epsilon=1.D-8
  real(kind=REAL64), allocatable :: A(:)
  real(kind=REAL64), allocatable :: B(:)
  real(kind=REAL64), allocatable :: C(:)
  real(kind=REAL64) :: scalar=3.0
  real(kind=REAL64) :: ar, br, cr, asum
  real(kind=REAL64) :: nstream_time, avgtime
  integer :: i, iter

  ! Allocate arrays in device memory
  !$omp allocate allocator(omp_target_device_mem_alloc)
```
allocate(A(length))

!$omp allocate allocator(omp_target_device_mem_alloc)
allocate(B(length))

!$omp allocate allocator(omp_target_device_mem_alloc)
allocate(C(length))

! Initialize the arrays

!$omp target teams distribute parallel do has_device_addr(A, B, C)
do i = 1, length
  A(i) = 2.0
  B(i) = 2.0
  C(i) = 0.0
end do

! Perform the computation

nstream_time = omp_get_wtime()
do iter = 1, iterations
  !$omp target teams distribute parallel do has_device_addr(A, B, C)
do i = 1, length
    C(i) = C(i) + A(i) + scalar * B(i)
  end do
end do
nstream_time = omp_get_wtime() - nstream_time

! Validate and output results

ar = 2.0
br = 2.0
cr = 0.0
do iter = 1, iterations
  do i = 1, length
    cr = cr + ar + scalar * br
  end do
end do

asum = 0.0
!$omp target teams distribute parallel do reduction(+:asum) has_device_addr(C)
do i = 1, length
  asum = asum + abs(C(i))
end do
if (abs(cr - asum)/asum > epsilon) then
  print *, "Failed Validation on output array: ", "Expected =", cr, "Observed =", asum
else

(continues on next page)
avgtime = nstream_time/iterations
print *, "Solution validates:", "Checksum =", asum, "Avg time (s) =", avgtime
endif
deallocate(A)
deallocate(B)
deallocate(C)
end program main

use_device_addr

The use_device_addr clause appears on a target data directive. It indicates that each list item already has corresponding storage on the device or is accessible on the device.

If a list item is mapped, then references to the list item in the construct are converted to references to the corresponding list item. If a list item is not mapped, it is assumed to be accessible on the device.

A list item may be an array section.

Just like has_device_addr, the use_device_addr clause is especially useful in Fortran, because it can be used with list items of any type (not just C_PTR) to indicate that the list items have device addresses.

The following Fortran example illustrates the use of the use_device_addr clause. In the example, array_d is mapped to the device with the alloc map-type, so storage is allocated for array_d on the device and no data transfer between the host and the device occurs. We use the use_device_addr(array_d) clause on the target data directive to indicate that array_d has corresponding storage on the device.

Listing 179: /examples/OpenMP/24_device_ptr_addr-_clauses/f_use_device_addr_01.f90

1 !=============================================================
2 ! Copyright © 2022 Intel Corporation
3 !
4 ! SPDX-License-Identifier: MIT
5 !=============================================================
6 program target_use_device_addr
7 use omp_lib
8 use iso_fortran_env, only : real64
9 implicit none
10
11 integer, parameter :: N1 = 1024
12 real(kind=real64), parameter :: aval = real(42, real64)
13 real(kind=real64), allocatable :: array_d(:), array_h(:)
14 integer :: i,err
15
16 ! Allocate host data
17 allocate(array_h(N1))
18 !$omp target data map (from:array_h(1:N1)) map(alloc:array_d(1:N1))
```Fortran
!$omp target data use_device_addr(array_d)
!$omp target
  do i=1, N1
    array_d(i) = aval
    array_h(i) = array_d(i)
  end do
!$omp end target
!$omp end target data
!$omp end target data
!
! Check result
write (*,*) array_h(1), array_h(N1)
if (any(array_h /= aval)) then
  err = 1
else
  err = 0
end if

deallocate(array_h)
if (err == 1) then
  stop 1
else
  stop 0
end if
end program target_use_device_addr
```

The following table summarizes the properties of the clauses described in this section.

<table>
<thead>
<tr>
<th>Clause</th>
<th>On which directive</th>
<th>Type of list item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_device_ptr</td>
<td>target, dispatch</td>
<td>C/C++: Pointer, array, or reference Fortran: C_PTR</td>
<td>Indicates that list item is a device pointer (has valid device address).</td>
</tr>
<tr>
<td>use_device_ptr</td>
<td>target data</td>
<td>C/C++: Pointer, array, or reference Fortran: C_PTR</td>
<td>Indicates that list item is a pointer to an object that has corresponding storage on device or is accessible on device.</td>
</tr>
<tr>
<td>has_device_addr</td>
<td>target</td>
<td>Any type (may be array section)</td>
<td>Indicates that list item has a valid device address.</td>
</tr>
<tr>
<td>use_device_addr</td>
<td>target data</td>
<td>Any type (may be array section)</td>
<td>Indicates that list item has corresponding storage on device or is accessible on the device.</td>
</tr>
</tbody>
</table>
Note:

Used the following when collecting OpenMP performance numbers:

- 2-tile Intel® GPU
- One GPU tile only (no implicit or explicit scaling).
- Internal versions of the Intel® compilers, runtimes, and GPU driver
- Level-Zero plugin
- Introduced a dummy target construct at the beginning of a program, so as not to measure startup time.
- Just-In-Time (JIT) compilation mode.
15.0 Debugging and Profiling

Understanding the behavior of your system is critical to making informed decisions about optimization choices. Some tools like profilers, analyzers, or debuggers are full-featured. Other tools like interval timers, kernel timers, and print statements are lighter weight. But all of them serve an important purpose in the optimization process. This section covers topics related to these tools’ use for software optimization.

15.1 GPU Analysis with VTune™ Profiler

VTune™ Profiler is a performance analysis tool for serial and multi-threaded applications. It helps you analyze algorithm choices and identify where and how your application can benefit from available hardware resources. Use it to locate or determine:

- Sections of code that don’t effectively utilize available processor resources
- The best sections of code to optimize for both sequential and threaded performance
- Synchronization objects that affect the application performance
- Whether, where, and why your application spends time on input/output operations
- Whether your application is CPU-bound or GPU-bound and how effectively it offloads code to the GPU
- The performance impact of different synchronization methods, different numbers of threads, or different algorithms
- Thread activity and transitions
- Hardware-related issues in your code such as data sharing, cache misses, branch misprediction, and others
- Profiling a DPC++ application running on a GPU

The tool also has new features to support GPU analysis:

- GPU Offload Analysis (technical preview)
- GPU Compute/Media Hotspots Analysis (technical preview)

**GPU Offload Analysis (Preview)**

Use this analysis type to analyze code execution on the CPU and GPU cores of your platform, correlate CPU and GPU activity, and identify whether your application is GPU-bound or CPU-bound. The tool infrastructure automatically aligns clocks across all cores in the system so you can analyze some CPU-based workloads together with GPU-based workloads within a unified time domain. This analysis lets you:

- Identify how effectively your application uses DPC++ or OpenCL™ kernels.
- Analyze execution of Intel Media SDK tasks over time (for Linux targets only).
- Explore GPU usage and analyze a software queue for GPU engines at each moment in time.
GPU Compute/Media Hotspots Analysis (Preview)

Use this tool to analyze the most time-consuming GPU kernels, characterize GPU usage based on GPU hardware metrics, identify performance issues caused by memory latency or inefficient kernel algorithms, and analyze GPU instruction frequency for certain instruction types. The GPU Compute/Media Hotspots analysis lets you:

- Explore GPU kernels with high GPU utilization, estimate the efficiency of this utilization, and identify possible reasons for stalls or low occupancy.
- Explore the performance of your application per selected GPU metrics over time.
- Analyze the hottest DPC++ or OpenCL kernels for inefficient kernel code algorithms or incorrect work item configuration.
- Run GPU Offload Analysis on a DPC++ Application.

15.1.1 Using VTune Profiler to Analyze GPU Applications

1. Launch VTune Profiler and from the Welcome page, click **New Project**. The Create a Project dialog box opens.

2. Specify a project name and a location for your project and click **Create Project**. The Configure Analysis window opens.

3. Make sure the Local Host is selected in the WHERE pane.

4. In the WHAT pane, make sure the Launch Application target is selected and specify the matrix_multiply binary as an Application to profile.

5. In the HOW pane, select the GPU Offload analysis type from the Accelerators group.

6. Click **Start** to launch the analysis.

This is the least intrusive analysis for applications running on platforms with Intel Graphics as well as third-party GPUs supported by VTune Profiler.

15.1.2 Run Analysis from the Command Line

To run the analysis from the command line:

On Linux* OS:

1. Set VTune Profiler environment variables by sourcing the script:

```
source <install_dir>/env-vars.sh
```

2. Run the analysis command:

```
vtune -collect gpu-offload -- ./matrix.dpcpp
```

On Windows* OS:

1. Set VTune Profiler environment variables by running the batch file:
2. Run the analysis command:

```bash
export <install_dir>\env\vars.bat
```
Most applications may not present obvious situations as described above. A detailed analysis is important to understand all dependencies. For example, GPU engines that are responsible for video processing and rendering are loaded in turns. In this case, they are used in a serial manner. When the application code runs on the CPU, this can cause ineffective scheduling on the GPU, which can lead you to mistakenly interpret the application as GPU bound.

Identify the GPU execution phase based on the computing task reference and GPU Utilization metrics. Then, you can define the overhead for creating the task and placing it in a queue.
To investigate a computing task, switch to the Graphics window to examine the type of work (rendering or computation) running on the GPU per thread. Select the Computing Task grouping and use the table to study the performance characterization of your task. To further analyze your computing task, run the GPU Compute/Media Hotspots analysis type.
15.1.4 Run GPU Compute/Media Hotspots Analysis

To run the analysis:

1. In the Accelerators group, select the GPU Compute/Media Hotspots analysis type.
2. Configure analysis options as described in the previous section.
3. Click **Start** to run the analysis.
Run Analysis from the Command line

On Linux OS:
vtune -collect gpu-hotspots -- ./matrix.dpcpp

On Windows OS:

vtune.exe -collect gpu-hotspots -- matrix_multiply.exe

## 15.1.5 Analyze Your Compute Tasks

<table>
<thead>
<tr>
<th>Computing Task</th>
<th>L3 Bandwidth, GB/sec</th>
<th>Untyped Memory Bandwidth, GB/sec</th>
<th>Shared Local Memory Bandwidth, GB/sec</th>
<th>Typed Memory Bandwidth, GB/sec</th>
<th>GPU Memory Bandwidth, GB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix1&lt;float&gt;</td>
<td>77.621</td>
<td>52.275</td>
<td>25.346</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>clEnqueueReadBufferRect</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>[Outside any task]</td>
<td>0.000</td>
<td>0.015</td>
<td>0.000</td>
<td>0.000</td>
<td>0.14</td>
</tr>
</tbody>
</table>

**Fig. 31: Characterization profile**

The default analysis configuration invokes the Characterization profile with the Overview metric set. In addition to individual compute task characterization available through the GPU Offload analysis, VTune Profiler provides memory bandwidth metrics that are categorized by different levels of GPU memory hierarchy.

**Fig. 32: VTune Profiler memory bandwidth metrics**

You can analyze compute tasks at source code level too. For example, to count GPU clock cycles spent on a particular task or due to memory latency, use the Source Analysis option.
Use our matrix multiply example in DPC++:

```cpp
// Basic matrix multiply
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
               TYPE c[][NUM], TYPE t[][NUM]) {
    int i, j, k;

    // Declare a deviceQueue
    sycl::default_selector device;
    sycl::queue q(device, exception_handler);
    cout << "Running on " << q.get_device().get_info<sycl::info::device::name>()
         << "\n";
    // Declare a 2 dimensional range
    sycl::range<2> matrix_range{NUM, NUM};

    // Declare 3 buffers and Initialize them
    sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
    sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
    sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);
    // Submit our job to the queue
    q.submit([&](auto &h) {
        (continues on next page)
    });
}
```

Fig. 33: GPU Compute/Media Hotspots analysis, Source Analysis
// Declare 3 accessors to our buffers. The first 2 read and the last
// read_write
sycl::accessor accessorA(bufferA, h, sycl::read_only);
sycl::accessor accessorB(bufferB, h, sycl::read_only);
sycl::accessor accessorC(bufferC, h);

// Execute matrix multiply in parallel over our matrix_range
// ind is an index into this range
h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
  int k;
  for (k = 0; k < NUM; k++) {
    // Perform computation ind[0] is row, ind[1] is col
    accessorC[ind[0]][ind[1]] +=
      accessorA[ind[0]][k] * accessorB[k][ind[1]];
  }
});

Analyzing the GPU-offload report from the command-line gives detailed recommendations on how to optimize the application.

Elapsed Time: 2.805s
  GPU Utilization: 3.3%
  GPU utilization is low. Consider offloading more work to the GPU to increase overall application performance.

  GPU Utilization
  GPU Engine      Packet Type GPU Time GPU Utilization(%)  
  Render and GPGPU Unknown 0.091s 3.3%

Hottest GPU Computing Tasks
  Computing Task Total Time Execution Time % of Execution Instance Count
  -------------- ---------- -------------- ----------------- --------------
  Matrix1<float> 0.183s 0.086s 47.0% 1

Recommendations:
  GPU Utilization: 3.3%
  GPU utilization is low. Switch to the Graphics View for in-depth analysis of host activity. Poor GPU utilization can prevent the application from offloading effectively.
  Transfer Time: 0.097s
  Execution time on the device is less than memory transfer time. Make sure your offload schema is optimal. Use the Intel Advisor tool to get an insight into possible causes of inefficient offload.

We can also examine how efficiently our GPU kernel is running using GPU-hotspots. How often our execution units are stalled can be a good indication of GPU performance. Another important metric is whether we are L3 bandwidth bound. In our case VTune is indicating that our L3 bandwidth was high when VEs were stalled.
Elapsed Time: 1.849s
GPU Time: 0.090s

EU Array Stalled/Idle: 6.2%
GPU L3 Bandwidth Bound: 65.2%

L3 bandwidth was high when VEs were stalled or idle.
Consider improving cache reuse.

FPU Utilization: 76.4%

For more ways to optimize GPU performance using VTune Profiler, see Software Optimization for Intel® GPUs in the Intel® VTune™ Profiler Performance Analysis Cookbook and Optimize Applications for Intel® GPUs with Intel® VTune™ Profiler.

15.2 Intel® Advisor GPU Analysis

Intel Advisor has two features that can help you analyze the performance of your application running on a GPU:

- Offload Modeling identifies kernels in your CPU-based code and predicts their performance when run on a GPU. It also helps you explore different GPU configurations for GPUs that do not exist yet.
- GPU Roofline Insights helps you see how your application is performing when compared to the limitations of your GPU.

Prerequisites: To use Intel Advisor, first set up the Intel Advisor environment variables:

- On Linux*: source <install-dir>/advisor-vars.sh
- On Windows*: <install-dir>/advisor-vars.bat

The rest of this chapter covers the two features introduced above, and a detailed recipe on using GPU Roofline Insights to analyze and optimize memory-bound applications.

15.2.1 Identify Regions to Offload to GPU with Offload Modeling

The Offload Modeling feature, a part of Intel Advisor, can be used to:

- Identify the portions of a code that can profitably be offloaded to a GPU.
- Predict the code's performance if run on a GPU.
- Experiment with accelerator configuration parameters.

Offload Modeling produces upper-bound speedup estimates using a bounds-and-bottlenecks performance model. It takes measured x86 CPU metrics and application characteristics as an input and applies an analytical model to estimate execution time and characteristics on a target GPU.

You can run the Offload Modeling perspective from the Intel Advisor GUI by using the advisor command line interface, or by using the dedicated Python* scripts delivered with the Intel Advisor. This topic describes how to run Offload Modeling with the scripts. For detailed description of other ways to run the perspective, see the Intel Advisor User Guide.
To run Offload Modeling for a C++ Matrix Multiply application on Linux* OS:

1. Collect application performance metrics with `collect.py`:
   ```bash
   advisor-python $APM/collect.py ./advisor_project --config gen9_gt2 -- matrix_multiply
   ```

2. Model your application performance on a GPU with `analyze.py`:
   ```bash
   advisor-python $APM/analyze.py ./advisor_project --config gen9_gt2
   ```

Once you have run the performance modeling, you can open the results in the Intel Advisor GUI or see CSV metric reports and an interactive HTML report generated in the `advisor_project/e000/pp000/data.0`
For example, in the Summary section of the report, review the following:

- The original execution time on a CPU, the predicted execution time on a GPU accelerator, the number of offloaded regions, and the estimated speedup in the Program metrics pane. For Matrix Multiply, Intel Advisor reports a 4.4x potential speedup.

- What the offloads are bounded by. This pane reports the main limiting factors that prevent your application from achieving better performance on a target device. The Matrix Multiply application is 99% bounded by last level cache (LLC) cache bandwidth.

- Exact source lines of the **Top Offloaded** code regions that can benefit from offloading to the GPU and estimated performance of each code region. For Matrix Multiply, there is one code region recommended for offloading.

- Exact source lines of the **Top Non-Offloaded** code regions that are not recommended for offloading and specific reasons for it.

Go to the Offloaded Regions tab to view the detailed measured and estimated metrics for the code regions recommended for offloading. It also reports the estimated amount of data transferred for the code regions and the corresponding offload taxes.

Use the data in the report to decide what regions of your code to port to DPC++. For example, you can port the C++ Matrix Multiply application to DPC++ as follows:

```
// Basic matrix multiply
void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
               TYPE c[][NUM], TYPE t[][NUM]) {
    int i, j, k;
```

(continues on next page)
15.2.2 Run a GPU Roofline Analysis

To estimate actual performance of a GPU application against hardware-imposed ceilings, you can use the GPU Roofline Insights feature. Intel® Advisor can generate a roofline model for kernels running on Intel GPUs. The GPU Roofline model offers a very efficient way to characterize your kernels and visualize how far you are from ideal performance. For details about the GPU Roofline, see the Intel Advisor User Guide.

Prerequisites: It is recommended to run the GPU Roofline with root privileges on Linux® OS or as an administrator on Windows® OS.
**Linux OS Users**

If you do not have root permissions on Linux, configure your system to enable collecting GPU metrics for non-root users:

1. Add your username to the video group. To check if you are already in the video group:
   ```bash
groups | grep video
```

   If you are not part of the video group, add your username to it:
   ```bash
sudo usermod -a -G video <username>
```

   Set the value of the `dev.i915.perf_stream_paranoid` sysctl option to 0:
   ```bash
sysctl -w dev.i915.perf_stream_paranoid=0
```

2. Disable time limits to run the OpenCL kernel for a longer period:
   ```bash
sudo sh -c "echo N> /sys/module/i915/parameters/enable_hangcheck"
```

**All Users**

1. Make sure that your DPC++ code runs correctly on the GPU. To check which hardware you are running on, add the following to your DPC++ code and run it:
   ```cpp
sycl::default_selector selector;
sycl::queue queue(delector);
auto d = queue.get_device();
std::cout << "Running on : " << d.get_info<cl::sycl::info::device::name>() << std::endl;
```

2. Set up the Intel Advisor environment for Linux OS:
   ```bash
source <advisor_install_dir>/env/vars.sh
```
   and for Windows OS:
   ```bash
<install_dir>/advisor-vars.bat
```

To run the GPU Roofline analysis in the Intel Advisor CLI:

1. Run the Survey analysis with the `profile-gpu` option:
   ```bash
advisor -collect=survey --profile-gpu --project-dir=./advisor-project --search-dir src=r=./matrix_multiply -- matrix_multiply
```

2. Run the Trip Count and FLOP analysis with `--profile-gpu`:
   ```bash
advisor --collect=tripcounts --stacks --flop --profile-gpu --project-dir=./advisor-project -- --search-dir src=r=./matrix_multiply -- matrix_multiply
```
3. Open the generated GPU Roofline report in the Intel Advisor GUI. Review the following metrics for the DPC++ Matrix Multiply application:

- In the Summary tab, view top hotspots and the memory layout in the Top Hotspots pane.

<table>
<thead>
<tr>
<th>Compute Task</th>
<th>Elapsed Time</th>
<th>GFLOPS</th>
<th>GINTOPS</th>
<th>Work Size/Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix1_2&lt;float&gt;</td>
<td>0.11s</td>
<td>19.985</td>
<td>10.978</td>
<td>1024 x 1024/16 x 16</td>
</tr>
</tbody>
</table>

![Fig. 36: Top Hotspots pane](image)

See how efficiently your application uses execution units in the Performance Characteristics pane.

![Performance Characteristics pane](image)

In the GPU Roofline Regions tab, see the GPU Roofline chart and performance metrics.
The Matrix Multiply application gets 10.98 GFLOPS. It uses global memory and is not optimized for local (SLM) memory because the application uses a global accessor.

The application is far from the maximum bandwidth of the GTI, as represented by the red dot on the right.

The dot on the left represents the L3 bandwidth. As the chart shows, it is far from the L3 bandwidth maximum.

As the GPU Roofline chart suggests, several possible optimizations might result in more efficient memory usage:

- Use local memory (SLM).
- Use the cache blocking technique to better use SLM/L3 cache.

The following code is the optimized version of the Matrix Multiply application. In this version, we declare two tiles and define them as `sycl::access::target:local`. We also modify the kernel to process these tiles in some inner loops.

```
// Replaces accessorC reference with a local variable
void multiply1_1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
                 TYPE c[][NUM], TYPE t[][NUM]) {
    int i, j, k;
    // Declare a deviceQueue
```

(continues on next page)
```cpp
sycl::default_selector device;
sycl::queue q(device, exception_handler);
cout << "Running on " << q.get_device().get_info<sycl::info::device::name>() << "\n";

// Declare a 2 dimensional range
sycl::range<2> matrix_range{NUM, NUM};

// Declare 3 buffers and Initialize them
sycl::buffer<TYPE, 2> bufferA((TYPE *)a, matrix_range);
sycl::buffer<TYPE, 2> bufferB((TYPE *)b, matrix_range);
sycl::buffer<TYPE, 2> bufferC((TYPE *)c, matrix_range);

// Submit our job to the queue
q.submit([&](auto &h) {
    // Declare 3 accessors to our buffers. The first 2 read and the last read_write
    sycl::accessor accessorA(bufferA, h, sycl::read_only);
sycl::accessor accessorB(bufferB, h, sycl::read_only);
sycl::accessor accessorC(bufferC, h);

    // Execute matrix multiply in parallel over our matrix_range
    // ind is an index into this range
    h.parallel_for(matrix_range, [=](sycl::id<2> ind) {
        int k;
        TYPE acc = 0.0;
        for (k = 0; k < NUM; k++) {
            // Perform computation ind[0] is row, ind[1] is col
            acc += accessorA[ind[0]][k] * accessorB[k][ind[1]];
        }
        accessorC[ind[0]][ind[1]] = acc;
    });
}).wait_and_throw();

// Replaces accessorC reference with a local variable and adds matrix tiling
void multiply1_2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM],
                 TYPE c[][NUM], TYPE t[][NUM]) {
    int i, j, k;

    // Declare a deviceQueue
    sycl::default_selector device;
sycl::queue q(device, exception_handler);
cout << "Running on " << q.get_device().get_info<sycl::info::device::name>() << "\n";

    // Declare a 2 dimensional range
    sycl::range<2> matrix_range{NUM, NUM};
sycl::range<2> tile_range{MATRIXTILESIZE, MATRIXTILESIZE};

    // Declare 3 buffers and Initialize them
```
Save the optimized version as multiply_1_2 and rerun the GPU Roofline. As the new chart shows:

- The optimized application gets 19.985 GFLOPS.
The application uses global and SLM memory, which represents the 16x16 tile. This increases memory bandwidth.

![Fig. 39: GPU Roofline new chart](image)

### 15.2.3 Optimize Memory-bound Applications with GPU Roofline

This section explains how to identify performance problems of GPU applications and understand their reasons using the GPU Roofline Insights perspective of the Intel® Advisor.

When developing a GPU application with SYCL/DPC++ or OpenMP programming model, it is important to keep in mind kernel parallel execution. Usually, massive and uneven memory access is the main problem that limits the GPU performance. If the path between a memory level where global data is located and GPU execution units is complex, the GPU might be stalled and wait for data because of bandwidth limitations at different stages on the path. Understanding these stages and measuring data flow on the path is an essential part of performance optimization methodology. GPU Roofline Insights perspective can help you to analyze bottlenecks for each kernel and quickly find the data path stage that causes the problem. A typical method to optimize GPU execution algorithms is reconsidering data access parents.

### Memory Path in Intel® GPU Microarchitecture

Depending on a GPU generation, the compute architecture of Intel® Processor Graphics uses a system memory as a compute device memory, which is unified by sharing the same DRAM with the CPU, or a dedicated VRAM resided on a discrete GPU card.

On an integrated GPU, where DRAM is shared between CPU and GPU, global data can travel form system DRAM through last-level cache (LLC) to a graphics technology interface (GTI) on a GPU. If data is efficiently...
reused, it can stay in the L3 cache of a GPU where execution units can access it and fetch to an Xe Vector Engine (XVE) register file.

Assuming the fastest way to access data on a system with high bandwidth and low latency is accessing data from registers, the cache-aware Roofline model (CARM) of the Intel Advisor treats it as the most effective access with true, or pure payload, amount of data consumed by an algorithm. Let us call it algorithmic data. For example, for a naive implementation of the matrix multiplication algorithm, theoretically, the amount of data read for each matrix and used for calculations is:

\[ D \times M^3 \]

where
- \( D \) is a size of a matrix element, in Bytes
- \( M \) is a size of a square matrix

If the General Register File (GRF) could theoretically fit all matrix data, the data is transferred from DRAM to GPU only once. Otherwise, the data is transferred from DRAM to the L3 cache and further in parts ordered by memory address access defined by the algorithm. If the calculations reuse data, some portion of it can stay in cache for longer making access faster. Ideally, data used by the algorithm should fit to the L3 cache. In real life, the best situation is when data in the L3 cache is reused as much as possible, and then it is evicted to allow the next portion of data to be reused.

In some cases, data is evicted from L3 cache, but next calculations need it. This creates redundant cache traffic and adds more load to the data path bandwidth. A good indicator for such situation is the L3 cache miss metric.

As data is fetched from DRAM to the L3 cache by cache lines of 64 bytes, accessing data objects that are smaller than the cache line size or cross cache line boundaries creates excessive cache traffic because unnecessary data is yet to be fetched from DRAM. The worst-case scenario is accessing byte-size objects that are randomly resided in global memory spaces. In this case, each access brings extra 63 Bytes of unnecessary data to the L3 cache and the data path is loaded with transferring data overhead (as opposed to algorithmic data).

In addition to the L3 cache, there is shared local memory (SLM), which is a dedicated structure outside of the L3 cache (for the Gen9, it is a part of L3 physically but separated functionally) that supports the work-group local memory address space. SLM can significantly increase GPU performance, but as SLM size is limited, you should carefully select work-group size to leverage performance improvement.

Each stage has a bandwidth limitation. Usually, the further from VE, the lower the bandwidth (similar to CPU memory architecture). Depending on a particular data access pattern implemented by an algorithm, some stages can be a bottleneck for the data flow. A more complex algorithm can have a combination of bottlenecks as data access can be a combination of different patterns.

Intel Advisor GPU Roofline Insights perspective can identify bottlenecks at different stages of data transfer and map the bottlenecks to program source code helping you to focus on performance problems and optimization. In addition to source data provided by the GPU Roofline Insights, you can use other tools to identify data that creates a bottleneck.
**GPU Roofline Methodology**

Intel Advisor implements the Roofline analysis in two perspectives: **CPU/Memory Roofline Insights**, which can analyze performance of CPU applications, and **GPU Roofline Insights**, which can analyze performance of GPU applications. General methodology of a Roofline model focused on the CPU/Memory Roofline is explained in the resources listed in the Roofline Resources for Intel® Advisor Users. You are strongly recommended to learn the Roofline basics before continuing. This recipe explores features of the GPU Roofline Insights perspective for analyzing performance on Intel GPUs.

**Roofline Result Overview**

Measuring GPU performance with GPU Roofline Insights is quite straightforward (Using GPU Roofline):

1. Run the GPU Roofline Insights with your preferred method: from Intel Advisor GUI or Intel Advisor command line tool.

2. Open the analysis results and examine a GPU Roofline chart reported. It plots an application’s achieved performance and arithmetic intensity against the machine’s maximum achievable performance.

For example, for the matrix multiply application, the GPU Roofline chart filtered by GTI (memory) level has one red dot representing a GPU kernel.

In the chart, each dot represents a kernel plotted by its measured data and performance characteristics. They are a central point of analysis in two-dimensional coordinates: arithmetic intensity (X axis) and performance (Y axis). Dot location against these coordinates shows the relation of kernel’s performance and its algorithm data consumption to GPU hardware limitations including its maximum computing performance and data flow bandwidth. On the chart, the hardware limitations are shown as diagonal lines, or roofs. The kernel location can help you to figure out two main things:
- If there is room for improvement to speed up kernel performance on the current GPU
- What the kernel is bound by: compute, cache, or memory bandwidth, and what you can change in the algorithm implementation to go beyond those boundaries to increase performance

This recipe describes only cases for memory- or cache-bound applications.

**Kernel Location Calculation**

It is important to know why exactly a kernel dot is located at a certain place of the chart for the following reasons.

A kernel is an implementation of an algorithm and it performs a fixed number of compute operations (such as add, mul, mad) with fixed amount of data. For example, for the matrix multiply naive implementation, assuming data is directly brought from memory, the algorithm arithmetic intensity AI is calculated as:

$$AI = \frac{M^3}{3*M^2}$$

where:

- M is a size of a square matrix
- $M^3$ is the number of operations
- $3*M^2$ is the amount of read/write data

The algorithm performance P is calculated as:

$$P = \frac{M^3}{T}$$

where:

- T is time it takes for the operations to complete
- $M^3$ is the number of operations

These values AI and P define the location of the kernel dot on the graph.

Let us switch from theoretical calculations to a real-world case. Intel Advisor measures data at run time and is not aware of theoretical number of operations and amount of data the algorithm needs. Each kernel is isolated by an internal instrumenting tool and measured by API tracing. Measured performance $P'$ is calculated as:

$$P' = \frac{l'}{T'}$$

where:

- $l'$ is measured number of executed computing instructions
- $T'$ is measured time
Measuring data used in the algorithm is easy only at the stage when VEs fetch data from the GRF because computing instructions have specific data reference syntax, which helps to calculate the amount of bytes used by the kernel. However, this data may come from different sources in the memory hierarchy in the GPU microarchitecture, and the amount of data that goes through different stages can be different.

On the Roofline chart, the kernel dot can be split into multiple dots for different memory levels. The following sections describe each memory level in detail and how Intel Advisor plots them on the Roofline chart.

How do we understand which memory level limits the algorithm execution? The algorithm performance is measured as the number of instructions I executed during time T, and it requires data traffic $D_{XX}$ at each memory stage. Assuming the algorithm is memory bound, at some levels, the data flow should be close to hardware bandwidth, while at other levels, it can be less limited. To identify the most probable bottleneck of the algorithm implementation, you need to find out which dot is the closest to its corresponding memory level roof line. Note that data flows may have more than one bottleneck, and the distance between dots and their corresponding roof lines should be similar.

In the Intel Advisor, double-click a dot on the Roofline chart to quickly find the limiting roof with the shortest distance to the dot and identify the bottleneck. The tool also provides additional hints about memory limitations, but we will review them later.

Performance Optimization Scenarios using GPU Roofline

The Roofline chart does not directly indicate what you need to change in your code to make it run faster on a GPU (although it provides some code hints, or guidance), but it shows a memory locality pattern(s) that dominate in your algorithm implementation. By examining where the kernel dots are located on the chart in relation to memory levels, you can identify the memory stage that is too narrow for the data flow and is the bottleneck. With this information, you can modify the data pattern used in your algorithm and apply, for example, using data blocking to reuse cache, avoiding multiple unnecessary data reads.
The more experience you have, the better you can understand data patterns, but there are basic cases that we can examine. Although, real-life applications do not usually show extreme behavior, like purely bound to a certain roof, as they are affected by:

- Auxiliary data transferred between memory and VEs, such as indexes of work-group item calculations, memory addresses arithmetic, loop indexes
- Data caching being more complicated as it is affected by the auxiliary data
- VE thread scheduling affecting data management

Let us consider several real-life examples of different applications and their implementations similar the theoretical cases discussed earlier.

**Data Block Optimized for the Matrix Multiply Algorithm with no SLM Usage**

This implementation is a naive matrix multiply without data blocking and is similar to the optimized kernel and data flow optimization case.

Even though data is not organized in blocks in the source code, the compiler recognizes the pattern and optimizes access to matrix arrays. As a result, we have a high level of data reuse in cache, and kernel performance is limited by the L3 cache. The Roofline chart shows one dot corresponding to a single kernel in the application. Based on its location on the chart:

- The kernel is memory bound, and the corresponding dot it close to L3 Bandwidth roof.
- GTI data traffic is four times smaller than the L3 cache data traffic, which indicates high data reuse.
- CARM and L3 traffics are almost the same, which indirectly indicates 100% of cache line usage because cache lines are fully filled with algorithmic data.
To confirm the 100% of cache line usage, review the L3 Cache Line Utilization metric in the GPU pane grid, which is 100%. The grid also reports VE Threading Occupancy of 98.1%, which indicates good scheduling and data distribution among threads.

To understand the limitations for future kernel code optimization, review the following data reported by the Intel Advisor:

- The Roofline Guidance pane shows kernel limitation summary and provides estimation for possible performance speedup after optimization. For the matrix multiply kernel, the main limitation is the L3 cache bandwidth. The kernel can run 1.4x faster if it uses the L3 cache more effectively and reaches its maximum bandwidth with the same arithmetic intensity, but a better data access pattern.
The Memory Metrics pane can help you understand memory level impact, which is time spent in requests from different memory levels, in per cent to the total time. For this kernel, GTI has less impact that L3 cache, but it is still taking a big part of total kernel execution time and may become a bottleneck after the L3 bandwidth limits are eliminated, for example, using SLM.

Shares metric is a visual way of estimating data portions processed from different memory levels. In this case, L3 cache has 4x more data than GTI.

The OP/S and Bandwidth pane shows the number of measured operations per second and data traffic in relation to the bandwidth limitations. For this kernel, the summary reports the following data:

- The theoretical SLM bandwidth is almost 3x times higher than the L3 cache bandwidth, but the SLM is not used in this implementation. Blocking matrix arrays to use them as local shared data can eliminate the L3 cache bandwidth limits.

- The kernel performance is only 27% of theoretically possible peak performance for Int32 data. With better memory access implementation, we could potentially reach 3x performance increase for this kernel.
Following the recommendations from the previous Intel Advisor result, we split the matrix arrays into small blocks to implement matrix multiplication data blocking and put the data blocks to the SLM for faster data reuse on a Xe-core level.

For this optimized implementation with data blocking, the Roofline chart looks as follows:

The data distribution has changed from the previous result. Firstly, the execution is not limited to memory, but is compute bound, which is good for overall performance and further optimizations.

There are a couple things to note in the memory-level dots:

- SLM traffic is much bigger than L3 traffic. L3 traffic is not zero, which is expected as data blocks are read to L3 cache and then copied to SLM for reuse.
CARM data traffic is three times bigger than the SLM traffic. The reason is not clear from the result, but it is a known effect that happens due to VE data port buffering data brought from SLM and accessed sequentially. This effect is positive and implies data reuse on the memory level closest to VEs.

Let us review data in the GPU Detail pane to understand changes in performance for this algorithm implementation:

- As the OP/S and Bandwidth pane shows, the L3 and SLM bandwidth are far from their limits. The kernel performance has increased to 47% of its theoretical limit of integer operations per second (INTOPS).

  **OP/S AND BANDWIDTH**

<table>
<thead>
<tr>
<th>Bandwidth Type</th>
<th>Value</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GINTOPS</td>
<td>460.91</td>
<td>47% of 972.99</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>0</td>
<td>0% of 973.13</td>
</tr>
<tr>
<td>CARM Bandwidth</td>
<td>704.20</td>
<td>GB/sec</td>
</tr>
<tr>
<td>SLM Bandwidth</td>
<td>234.68</td>
<td>25% of 913.87</td>
</tr>
<tr>
<td>L3 Bandwidth</td>
<td>39.27</td>
<td>8% of 459.33</td>
</tr>
<tr>
<td>GTI (Memory) Bandwidth</td>
<td>19.93</td>
<td>11% of 166.40</td>
</tr>
</tbody>
</table>

- As the Roofline Guidance chart shows, the kernel performance is limited by the Int32 Vector operations, which are the operations that the compiler used to implement the code. The chart indicates that the kernel can be optimized to run 2.1x faster.

- As the Performance Characteristics pane shows, the VEs are stalled for 43.6% of execution cycles. As the algorithm is fully vectorized, there should be other reasons for the VE stalls. By optimizing the VE performance, you might get the 2.1x performance improvement indicated in the Roofline Guidance pane.
You can run the GPU Compute/Media Hotspots analysis of the Intel® VTune™ GPU Hotspot analysis to investigate reasons for the VE stalls further.

**Big Data Array with Data Reuse for a STREAM Benchmark**

The STEAM benchmark is a small application that brings a big chunk of data from memory and executes basic compute kernels: Copy, Scalar, Add, and Triad. The number of compute operations per kernel is small or equals to 0, so the kernels are expected to be memory bound. For this reason, we use it to define data bandwidth limits in a system.

After analyzing the benchmark with the GPU Roofline Insights on the Intel Processor Graphics code-named Tiger Lake, the Roofline chart shows four dots that correspond to the benchmark kernels. The dots are located on the memory-bound side of the chart below the DRAM bandwidth roof.

The Roofline Guidance chart shows that the kernels are GTI Bandwidth bound, not DRAM bound as the main Roofline chart suggests. The reason for it is that Intel Advisor cannot measure the bandwidth for data transferred between DRAM and XVE on integrated GPUs due to hardware limitations.
The Roofline Guidance suggests you improving cache locality to optimize performance and get better data reuse. This advice is also applicable to other cases when we test data bandwidth and compute performance is not a purpose for optimization.

**ROOFLINE GUIDANCE**

*This kernel is bounded by the GTI Bandwidth*

Improve cache locality. For example, optimize cache accesses by implementing cache blocking technique.

In the OP/S and Bandwidth pane, review the specific numbers for the achieved memory bandwidth. Notice that CARM, L3, and GTI stages has similar achieved bandwidth, so the bottleneck for this benchmark is the most distant memory interface.

**OP/S AND BANDWIDTH**

<table>
<thead>
<tr>
<th>Component</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GINTOPS</td>
<td>23.43</td>
<td>2% of 973,064 int32 GINTOPS</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>0</td>
<td>0% of 973,08 SP GFLOPS</td>
</tr>
<tr>
<td>CARM Bandwidth</td>
<td>39.45</td>
<td>GB/sec</td>
</tr>
<tr>
<td>SLM Bandwidth</td>
<td>0</td>
<td>0% of 913,86 GB/sec</td>
</tr>
<tr>
<td>L3 Bandwidth</td>
<td>39.38</td>
<td>8% of 446,41 GB/sec</td>
</tr>
<tr>
<td>GTI (Memory) Bandwidth</td>
<td>39.63</td>
<td>23% of 166,40 GB/sec</td>
</tr>
</tbody>
</table>

Note here that all stages CARM, L3, and GTI have the similar effective BW and all 3 memory components are roughly identical to each other for a given kernel. Having identical roofline components means that there is no reuse in the cache or register file and every attempt to fetch the data requires accessing all the way down to external memory, because no data is cached any time. This (equal CARM, L3 and External Memory roofline components) is a common indication of “streaming” pattern.

In given case, this also indicates that the most distant memory interface is the bottleneck for this benchmark. Slight difference in kernels BW which is still can be observed is due to Copy/Scale kernels have equal Reads/Writes, while Add/Triad kernels have twice more Reads then Writes, and Read BW is higher on the system.
To eliminate the hardware limitations of the Intel Processor Graphics code-named Tiger Lake that do not allow Intel Advisor to measure bandwidth between DRAM and VE, let us analyze the benchmark running on a discrete Intel® Iris Xe MAX graphics. The resulting Roofline chart shows four kernel dots below the DRAM bandwidth roof.

In the OP/S and Bandwidth pane, Intel Advisor now correctly identifies DRAM as the highest level of bottleneck.
As the OP/S and Bandwidth and Memory Metrics panes show, the DRAM data traffic is very close to its theoretical limit, and the stream benchmark really measures the practical limits of the data flow.
Partially Effective Data Access in a Stencil Data Pattern Benchmark

One of the most interesting cases is when data access is compact but in a very limited local range, while globally, the access is sparse. Such case is frequent in real-life applications, for example, in a stencil-based kernel computation where data in two axes, for example, X and Y, is accessed sequentially in the memory space, but data in Z axis is accessed in a big unit stride.

Let us analyze a 504.polbm applications from the SPEC ACCEL benchmark set running on the Intel Processor Graphics with Gen12 architecture. This benchmark application is written on C with OpenMP* offload to GPU. It works with double-precision numbers, but the Intel Processor Graphics with Gen12 architecture can only simulate the calculations with integer data. That is why we examine the Roofline chart for integer operations.

The GPU Roofline chart shows one dot that correspond to the benchmark kernel. The dot is located between memory and compute roofs, which means that if GPU parameters are changed (for example, if you run the analysis for a hardware with a higher memory bandwidth), the kernel might slightly move from memory bound to compute bound.

![GPU Roofline Chart]

As the Roofline Guidance pane shows, the kernel is limited by L3 cache bandwidth. Intel Advisor also detects low cache line utilization for the kernel, which is expected from a stencil-based kernel.
In general, to optimize data access in the stencil-based kernels, you need to apply different techniques that change data layout to use SLM for data locality and SIMD parallelism per data axis. However, you cannot change data layout for benchmarks, and all optimizations are done by the Graphics Compiler.

**Conclusion**

GPU Roofline Insights perspective of the Intel Advisor is a powerful tool that helps you investigate performance of kernel offloaded to Intel GPUs. It is easier to understand Roofline results for theoretical extreme cases. Such cases have only hardware limitations, so performance optimization strategy is clearer. However, real-life applications might have several limiting factors combined. The Roofline can help you address performance issues by identifying the most contributing factors. Once you eliminate one limitation, the analysis identifies the next factor you can address, until the performance is close to theoretical hardware limitations and optimization stops bringing improvements.

**15.3 Doing IO in the Kernel**

Print statement is the most fundamental capability needed for looking at the results of a program. In accelerators, printing is surprisingly hard and also fairly expensive in terms of overhead. DPC++ provides some capabilities to help make this task similar to standard I/O C/C++ programs, but there are some quirks you need to understand because of the way accelerators work. File I/O is not possible from DPC++ kernels.

SYCL* provides the `stream` class to let you print information to the console from within kernels, providing an easy way to debug simple issues without resorting to a debugger. The `stream` class provides functionality that is very similar to the C++ STL `ostream` class, and its usage is similar to the STL class. Below we describe how to use SYCL `stream` class to output information from within an enqueued kernel.

To use the class we must first instantiate it. The signature of the `stream` constructor is as follows:
The constructor takes three parameters:

- **BufferSize**: the total number of characters that may be printed over the entire kernel range
- **MaxStatementSize**: the maximum number of characters in any one call to the stream class
- **CGH**: reference to the `sycl::handler` parameter in the `sycl::queue::submit` call

Usage is very similar to that of the C++ STL `ostream std::cout` class. The message or data that needs to be printed is sent to the SYCL `stream` instance via the appropriate `operator<<` method. SYCL provides implementations for all the built-in data types (such as `int`, `char` and `float`) as well as some common classes (such as `sycl::nd_range` and `sycl::group`).

Here is an example usage of a SYCL `stream` instance:

```cpp
void out1() {
    constexpr int N = 16;
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 1024, cgh);
        cgh.parallel_for(N, [=](sycl::item<1> it) {
            int id = it[0];
            /* Send the identifier to a stream to be printed on the console */
            str << "ID= " << id << sycl::endl;
        });
    }).wait();
} // end out1
```

The use of `sycl::endl` is analogous to the use of the C++ STL `std::endl` stream reference—it serves to insert a new line as well as flush the stream.

Compiling and executing the above kernel gives the following output:

```
ID=0
ID=1
ID=2
ID=3
ID=4
ID=5
ID=6
ID=7
ID=8
ID=9
ID=10
ID=11
ID=12
ID=13
ID=14
ID=15
```
Care must be taken in choosing the appropriate **BufferSize** and **MaxStatementSize** parameters. Insufficient sizes may cause statements to either not be printed, or to be printed with less information than expected. Consider the following kernel:

```cpp
void out2() {
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 4, cgh);
        cgh.parallel_for(1, [=](sycl::item<>)) {
            str << "ABC" << sycl::endl; // Print statement 1
            str << "ABCDEFG" << sycl::endl; // Print statement 2
        };
    }).wait();
} // end out2
```

Compiling and running this kernel gives the following output:

```
ABC
```

The first statement was successfully printed out since the number of characters to be printed is 4 (including the newline introduced by `sycl::endl`) and the maximum statement size (as specified by the **MaxStatementSize** parameter to the `sycl::stream` constructor) is also 4. However, only the newline from the second statement is printed.

The following kernel shows the impact of increasing the allowed maximum character size:

```cpp
void out3() {
    sycl::queue q;
    q.submit([&](auto &cgh) {
        sycl::stream str(8192, 10, cgh);
        cgh.parallel_for(1, [=](sycl::item<>) {
            str << "ABC" << sycl::endl; // Print statement 1
            str << "ABCDEFG" << sycl::endl; // Print statement 2
        });
    }).wait();
} // end out3
```

Compiling and running the above kernel gives the expected output:

```
ABC
ABCDEFG
```

The examples above used simple kernels with a single work item. More realistic kernels will typically include multiple work items. In these cases, no guarantee is made as to the specific order of the statements printed to the console and you should expect statements from different work items to be interleaved. Consider the following kernel:
One run can produce the following output.

```
ID=0
ID=1
ID=2
ID=3
ID=4
ID=5
[snip]
ID=26
ID=27
ID=28
ID=29
ID=30
ID=31
```

When this program is run again, we might get the output in a totally different order, depending on the order the threads are executed.

```
ID=4
ID=5
ID=6
ID=7
ID=0
ID=1
[snip]
ID=14
ID=15
ID=28
ID=29
ID=30
ID=31
```

The output from `sycl::stream` is printed after the kernel has completed execution. In most cases this is of no consequence. However, should the kernel fault or throw an exception, no statement will be printed. To illustrate this, consider the following kernel, which raises an exception:
Compiling and executing the above code generates a segmentation fault due the write to a null pointer.

None of the print statements are actually printed to the console. Instead, you will see an error message about a segmentation fault. This is unlike traditional C/C++ streams.

### 15.4 Using the Timers

The standard C++ chrono library can be used for tracking times with varying degrees of precision in DPC++. The following example shows how to use the chrono timer class to time kernel execution from the host side.
auto sum_acc = sum_buf.get_access<h>());

h.parallel_for(num_items, [=](id<1> i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
}).wait();

auto t2 = std::chrono::steady_clock::now(); // Stop timing

return(std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count());
}

void InitializeArray(IntArray &a) {
    for (size_t i = 0; i < a.size(); i++) a[i] = i;
}

int main() {
    default_selector d_selector;

    IntArray a, b, sum;

    InitializeArray(a);
    InitializeArray(b);

    queue q(d_selector);

    std::cout << "Running on device: "
        << q.get_device().get_info<info::device::name>() << "\n";
    std::cout << "Vector size: " << a.size() << "\n";

    double t = VectorAdd(q, a, b, sum);

    std::cout << "Vector add successfully completed on device in " << t << " microseconds\n";
    return 0;
}

Note that this timing is purely from the host side. The actual execution of the kernel on the device may start much later, after the submission of the kernel by the host. DPC++ provides a profiling capability that let you keep track of the time it took to execute kernels.

#include <CL/sycl.hpp>
#include <array>
#include <iostream>
using namespace sycl;

// Array type and data size for this example.
constexpr size_t array_size = (1 << 16);
typedef std::array<int, array_size> IntArray;

double VectorAdd(queue &q, const IntArray &a, const IntArray &b, IntArray &sum) {
    range<1> num_items{a.size()};
}
buffer a_buf(a);
buffer b_buf(b);
buffer sum_buf(sum.data(), num_items);

event e = q.submit([&](handler &h) {
  // Input accessors
  auto a_acc = a_buf.get_access<access::mode::read>(h);
  auto b_acc = b_buf.get_access<access::mode::read>(h);

  // Output accessor
  auto sum_acc = sum_buf.get_access<access::mode::write>(h);

  h.parallel_for(num_items, [=](id<1> i) { sum_acc[i] = a_acc[i] + b_acc[i]; });
});
q.wait();
return e.template get_profiling_info<info::event_profiling::command_end>() -
    e.template get_profiling_info<info::event_profiling::command_start>();
}

void InitializeArray(IntArray &a) {
  for (size_t i = 0; i < a.size(); i++) a[i] = i;
}

int main() {
  default_selector d_selector;

  IntArray a, b, sum;

  InitializeArray(a);
  InitializeArray(b);

  queue q(d_selector, property::queue::enable_profiling{});

  std::cout << "Running on device: "
    << q.get_device().get_info<info::device::name>() << "\n";
  std::cout << "Vector size: " << a.size() << "\n";

  double t = VectorAdd(q, a, b, sum);

  std::cout << "Vector add successfully completed on device in " << t << " nanoseconds\n";
  return 0;
}

When these examples are run, it is quite possible that the time reported by chrono is much larger than the time reported by the DPC++ profiling class. This is because the DPC++ profiling does not include any data transfer times between the host and the offload device.
15.5 How to Use the Intercept Layer for OpenCL™ Applications

Linux* and OS X*: Linux OS X Build Status | Windows*: Windows Build Status

The Intercept Layer for OpenCL Applications is a tool that can intercept and modify OpenCL calls for debugging and performance analysis. Using the Intercept Layer for OpenCL Applications requires no application or driver modifications.

To operate, the Intercept Layer for OpenCL Applications masquerades as the OpenCL ICD loader (usually) or as an OpenCL implementation (rarely) and is loaded when the application intends to load the real OpenCL ICD loader. As part of the Intercept Layer for OpenCL Application's initialization, it loads the real OpenCL ICD loader and gets function pointers to the real OpenCL entry points. Then, whenever the application makes an OpenCL call, the call is intercepted and can be passed through to the real OpenCL with or without changes.

To access the OpenCL Intercept Layer repository:

```
git clone https://github.com/intel/opencl-intercept-layer
```

All controls are documented here:  https://github.com/intel/opencl-intercept-layer/blob/master/docs/controls.md

See intercept documentation for information about controls.

To run, use the following setup:

```
export CLI_OpenCLFileName=/opt/intel/inteloneapi/compiler/latest/linux/lib/libOpenCL.so.1
export LD_LIBRARY_PATH=/home/opencl-intercept-layer/build/intercept:$LD_LIBRARY_PATH
export SYCL_BE=PI_OPENCL
CLI_ReportToStderr=0 CLI_ReportToFile=1 CLI_HostPerformanceTiming=1 CLI_DevicePerformanceTiming=1 CLI_DumpDir=. ./matrix.dpcpp
```
This will generate a file called cli_intercept_report.txt. The file will include the following data and tables shown below.

- Total Enqueues: 2
- Total Time (ns): 1604325652

**Table 13:** Host Performance Timing Results

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Time (ns)</th>
<th>Time (%)</th>
<th>Average (ns)</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clBuildProgram</td>
<td>1</td>
<td>337069812</td>
<td>21.01%</td>
<td>337069812</td>
<td>337069812</td>
<td>337069812</td>
</tr>
<tr>
<td>clCreateBuffer</td>
<td>3</td>
<td>3393909</td>
<td>0.21%</td>
<td>1131303</td>
<td>140325</td>
<td>2036170</td>
</tr>
<tr>
<td>clCreateCommandQueue WithProperties</td>
<td>1</td>
<td>5221</td>
<td>0.00%</td>
<td>5221</td>
<td>5221</td>
<td>5221</td>
</tr>
<tr>
<td>clCreateContext</td>
<td>1</td>
<td>33639</td>
<td>0.00%</td>
<td>33639</td>
<td>33639</td>
<td>33639</td>
</tr>
<tr>
<td>clCreateKernel</td>
<td>1</td>
<td>11713</td>
<td>0.00%</td>
<td>11713</td>
<td>11713</td>
<td>11713</td>
</tr>
<tr>
<td>clCreateProgramWithIL</td>
<td>1</td>
<td>5221</td>
<td>0.00%</td>
<td>5221</td>
<td>5221</td>
<td>5221</td>
</tr>
<tr>
<td>clCreateContext</td>
<td>1</td>
<td>33639</td>
<td>0.00%</td>
<td>33639</td>
<td>33639</td>
<td>33639</td>
</tr>
<tr>
<td>clCreateKernel</td>
<td>1</td>
<td>11713</td>
<td>0.00%</td>
<td>11713</td>
<td>11713</td>
<td>11713</td>
</tr>
<tr>
<td>clCreateProgramWithIL</td>
<td>1</td>
<td>5221</td>
<td>0.00%</td>
<td>5221</td>
<td>5221</td>
<td>5221</td>
</tr>
<tr>
<td>clCreateContext</td>
<td>1</td>
<td>33639</td>
<td>0.00%</td>
<td>33639</td>
<td>33639</td>
<td>33639</td>
</tr>
<tr>
<td>clCreateKernel</td>
<td>1</td>
<td>11713</td>
<td>0.00%</td>
<td>11713</td>
<td>11713</td>
<td>11713</td>
</tr>
<tr>
<td>clCreateProgramWithIL</td>
<td>1</td>
<td>5221</td>
<td>0.00%</td>
<td>5221</td>
<td>5221</td>
<td>5221</td>
</tr>
<tr>
<td>clEnqueueNDRangeKernel (ZTS9Matrix1_2IfE)</td>
<td>3</td>
<td>3102488</td>
<td>0.19%</td>
<td>3102488</td>
<td>3102488</td>
<td>3102488</td>
</tr>
<tr>
<td>clEnqueueReadBufferRect</td>
<td>1</td>
<td>1099684</td>
<td>0.07%</td>
<td>1099684</td>
<td>1099684</td>
<td>1099684</td>
</tr>
<tr>
<td>clGetContextInfo</td>
<td>8</td>
<td>4720</td>
<td>0.00%</td>
<td>590</td>
<td>160</td>
<td>1997</td>
</tr>
<tr>
<td>clGetDeviceIDs</td>
<td>12</td>
<td>53004</td>
<td>0.00%</td>
<td>4417</td>
<td>504</td>
<td>14853</td>
</tr>
<tr>
<td>clGetDeviceInfo</td>
<td>30</td>
<td>85695</td>
<td>0.01%</td>
<td>2856</td>
<td>133</td>
<td>19920</td>
</tr>
<tr>
<td>clGetExtensionFunction AddressForPlatform</td>
<td>3</td>
<td>6446</td>
<td>0.00%</td>
<td>2148</td>
<td>1317</td>
<td>3687</td>
</tr>
<tr>
<td>clGetKernelInfo</td>
<td>2</td>
<td>716</td>
<td>0.00%</td>
<td>358</td>
<td>169</td>
<td>547</td>
</tr>
<tr>
<td>clGetPlatformIDs</td>
<td>2</td>
<td>1198290216</td>
<td>74.69%</td>
<td>599145108</td>
<td>715</td>
<td>1198289501</td>
</tr>
<tr>
<td>clGetPlatformInfo</td>
<td>12</td>
<td>22538</td>
<td>0.00%</td>
<td>1878</td>
<td>404</td>
<td>7326</td>
</tr>
<tr>
<td>clReleaseCommandQueue</td>
<td>1</td>
<td>1744</td>
<td>0.00%</td>
<td>1744</td>
<td>1744</td>
<td>1744</td>
</tr>
<tr>
<td>clReleaseContext</td>
<td>1</td>
<td>331</td>
<td>0.00%</td>
<td>331</td>
<td>331</td>
<td>331</td>
</tr>
<tr>
<td>clReleaseDevice</td>
<td>6</td>
<td>6365</td>
<td>0.00%</td>
<td>1060</td>
<td>491</td>
<td>1352</td>
</tr>
<tr>
<td>clReleaseEvent</td>
<td>2</td>
<td>2398</td>
<td>0.00%</td>
<td>1199</td>
<td>992</td>
<td>1406</td>
</tr>
<tr>
<td>clReleaseKernel</td>
<td>1</td>
<td>2733</td>
<td>0.00%</td>
<td>2733</td>
<td>2733</td>
<td>2733</td>
</tr>
<tr>
<td>clReleaseMemObject</td>
<td>3</td>
<td>45464</td>
<td>0.00%</td>
<td>15154</td>
<td>10828</td>
<td>22428</td>
</tr>
<tr>
<td>clReleaseProgram</td>
<td>1</td>
<td>51380</td>
<td>0.00%</td>
<td>51380</td>
<td>51380</td>
<td>51380</td>
</tr>
<tr>
<td>clRetainDevice</td>
<td>6</td>
<td>8680</td>
<td>0.00%</td>
<td>1446</td>
<td>832</td>
<td>2131</td>
</tr>
<tr>
<td>clSetKernelArg</td>
<td>20</td>
<td>6976</td>
<td>0.00%</td>
<td>348</td>
<td>180</td>
<td>1484</td>
</tr>
<tr>
<td>clSetKernelExecInfo</td>
<td>3</td>
<td>1588</td>
<td>0.00%</td>
<td>529</td>
<td>183</td>
<td>1149</td>
</tr>
<tr>
<td>clWaitForEvents</td>
<td>6</td>
<td>60864855</td>
<td>3.79%</td>
<td>10144142</td>
<td>928</td>
<td>60855555</td>
</tr>
</tbody>
</table>

**Table 14:** Device Performance Timing Results for Intel(R) Gen9 HD Graphics NEO (24CUs, 1200MHz)

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Time (ns)</th>
<th>Time (%)</th>
<th>Average (ns)</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ZTS9Matrix1_2IfE</td>
<td>1</td>
<td>58691515</td>
<td>99.98%</td>
<td>58691515</td>
<td>58691515</td>
<td>58691515</td>
</tr>
<tr>
<td>clEnqueueReadBufferRect</td>
<td>1</td>
<td>13390</td>
<td>0.02%</td>
<td>13390</td>
<td>13390</td>
<td>13390</td>
</tr>
</tbody>
</table>
The report includes detailed timing data on both your host and device.

### 15.6 Level Zero Tracer

This tool is an analogue of the Intercept Layer for OpenCL™ Applications designed to support Level Zero.

To access the Level Zero Tracer git:

```
git clone https://github.com/intel/pti-gpu
```

All steps to build and run these tools and all controls are documented in the Level Zero Tracer README:: https://github.com/intel/pti-gpu/blob/master/tools/ze_tracer/README.md

**See ze_tracer documentation for information about setup and controls.**

To run, use the following setup:

```
./ze_tracer [options] <target_application>
```

Example use case:

```
./ze_tracer -d -h --chrome-call-logging --chrome-device-timeline <target_application>
```

```
--chrome-call-logging --chrome-device-timeline will generate a file called zet_trace.<random number>.json. This file contains the dumps of timestamps for device activities per command queue to JSON format and can be opened in chrome://tracing browser tool.
```

```
-d -h is used to collect the duration for each device and host API call and provides the summary for the whole application in standard output.
```

**See ze_tracer documentation for more information.**
16.0 GPU Analysis with Intel® Graphics Performance Analyzers (Intel® GPA)

16.1 Introduction

16.1.1 Intel® Graphics Performance Analyzers (Intel® GPA)

Intel® GPA is a performance analysis tool suite for analysis of applications run on single and multi-CPU platforms as well as single and multi-GPU platforms. It offers detailed analysis of data both visually and via scripting.

Intel® GPA consists of 4 GUI tools and a command line tool.

- **GUI Tools**
  - Graphics Monitor - hub of Intel® GPA tools for selecting options and starting trace and frame captures.
  - System Analyzer - live analysis of CPU and GPU activity.
  - Graphics Trace Analyzer - deep analysis of CPU/GPU interactions during a capture of a few seconds of data.
  - Graphics Frame Analyzer - deep analysis of GPU activity in one or more frames.

- **Command Line Interface Tool**
  - Intel® GPA Framework - scriptable command line interface allowing the capture and analysis of extensive data from one or more frames.

This chapter focuses on the functionality of Graphics Frame Analyzer for the deep view it provides into GPU behavior.

(Note: Intel® GPA Framework can be used to capture the same data as it is the backend of Graphics Frame Analyzer. In addition Intel® GPA Framework can be used to automate profiling work.)

16.1.2 Graphics Frame Analyzer Features

Some of the useful features of Graphics Frame Analyzer are:

- Immediately see which frame of a set of frames takes longest.

- Use Advanced Profiling Mode to automatically calculate your application’s hottest bottlenecks based both on pipeline state and GPU metrics so that optimizing for one frame may optimize multiple parts of your application.

- Geometry - wire frame and solid:
  - in seconds you can see if you have drawn outside of your screen view.
  - View the geometry at any angle, dynamically.
Textures
- Visualize textures draw call by draw call.
- See if a draw call with a long duration is drawing an insignificant part of the frame.

Shaders
- See which shader code lines take the most time.
- See how many times each DXR (DirectX Raytracing) shader is called, as well as shader EU occupancy.

Render State Experiments - at the push of a button simplify textures or pixels or disable events to help locate the causes of bottlenecks, immediately seeing the changes in the metrics and other data.

Supported APIs
- Direct3D 11
- Direct3D 12 (including DirectX 12 Ultimate)
- Vulkan

16.2 Execution Unit Stall, Active and Throughput

Graphics Frame Analyzer is a powerful tool that can be used by novice and expert alike to take a quick look at frame duration, API calls, resource data and event data. Understanding more about the meaning of each data element levels you up making it easier to root cause performance issues.

16.2.1 Execution Stall, Execution Active, Execution Throughput

Knowing how to interpret the interrelationships of these 3 data elements can take you much further in your ability to understand the interworking of your applications with respect to the GPU(s).

16.2.2 EU, XVE (Xe Vector Engine), and XMX

As discussed in the Intel® Iris® Xe GPU Architecture section of this document, in Xe-LP and prior generations of Intel GPUs the EU - execution unit - is the compute unit of the GPU. In Xe-HPG and Xe-HPC we introduced the Xe-core as the compute unit. For these latter platforms each Xe-core consists of ALUs (arithmetic logic units) - 8 vector engines (XVE) and 8 matrix extensions (XMX).

In Graphics Frame Analyzer, if you are running on Xe-LP or earlier architecture, you will see EU Active, EU Stall and EU Throughput data labels. On newer architecture you will see XVE Active, XVE Stall and XVE Throughput data labels. Here we use Xe-LP as our reference architecture, thus we will refer to the EU. But understand that whether it is the EU or the XVE, the Stall/Active/Throughput relationships affect performance in the same ways.
16.2.3 Understanding these 3 data elements

First, let’s see what it looks like to drill down from the entire Xe-LP unit with its 96 EUs into a single EU. The General Register File (GRF) on each EU of this particular GPU holds 7 threads. Figure 1 shows the zoom in from the entire GPU to a single EU.

![Execution Unit (EU) Overview](image)

**Fig. 40:** Zooming in on a single EU

Let’s take a closer look at the details of the EU. In Figure 2, of the elements shown, we will focus primarily on the 7 thread slots of the GRF, addressing the importance of the thread interactions with the SEND unit and the 2 ALUs.
Now let’s look at a threading scenario. Figure 3 shows the contents of the GRF. We see that the GRF of this EU is loaded with only one thread in this instant. We see that single thread executing for some quantity of time. This means that one or both of the ALUs are invoked to execute instructions. At some point that thread needs to either read or write data, which activates the SEND unit. While the thread waits for the read/write to complete, the EU is stalled - it has a thread loaded but nothing to compute during that time.
Augmenting this scenario, in Figure 4 there is a second thread in the EU. If there is a second thread loaded into the GRF of this EU, then, at the time when the first thread invokes the SEND unit, instead of stalling execution, the EU begins executing the instructions of the second thread. When that second thread invokes a command requiring the SEND unit, the EU becomes stalled until the first thread is able to continue. Had there been a third thread in this EU or if the first SEND returned sooner, all latency may have been hidden, resulting in thread latency, but no stall during this time for this EU.
**Terminology**

For the following definitions the resulting data calculated

- is the average across all EUs;
- consists of both the full frame data and the data for the selected portion of the frame. The selection may be a single call or a set of calls, or even a set of frames.

**Idle**

EU Idle is the percentage of time when no thread is loaded.

**Active**

EU Active is the percentage of time when ALU0 or ALU1 were executing some instruction. It should be as high as possible; a low value is caused either by a lot of stalls or EUs being idle.
**Stall**

EU Stall is the percentage of time, when one or more threads are loaded but none of them are able to execute because they are waiting for a read or write.

**Thread Occupancy**

EU Thread Occupancy is the percentage of occupied GRF slots (threads loaded). This generally should be as high as possible. If the EU Thread Occupancy value is low, this indicates either a bottleneck in preceding HW blocks, such as vertex fetch or thread dispatch, or, for compute shaders it indicates a suboptimal SIMD-width or Shared Local Memory usage.

If only a single thread is executing on each of the 96 EUs, then 1 of 7 slots/EU is occupied, resulting in thread occupancy of $1/7 \approx 14\%$.

If 2 EUs have no threads loaded (they are idle) but the other 94 EUs have 6 threads loaded, we have occupancy $= (0 + 0 + 6 \times 94)/672 = 84\%$.

The Thread Occupancy values you will see in Graphics Frame Analyzer indicate the average occupancy across all EUs over the time selected. Though other hardware may have a different number of EUs or XVEs, the calculations are over all execution units. For example, below, on the left, you see a frame where over the entire frame duration of 6 ms, though thread occupancy fluctuated during that 6 ms, the average over that time for all 96 EUs is 77%. We can also look at thread occupancy over a part of the frame duration. Below, on the right, we select the 3 most time-consuming draw calls and see that during the 1.9 ms that it took to execute these bits of the application, thread occupancy is 85.3%.
16.3 Graphics Frame Analyzer

View this data in Intel(r) GPA's Graphics Frame Analyzer. For usage, see our 8 short videos and/or their corresponding articles. Video Series: An In-Depth Look at Graphics Frame Analyzer (intel.com)

In Graphics Frame Analyzer after opening a frame, you will see a view such as that in Figure 5. If you look at the data just after opening the frame, you will see data percentage values for the entire frame. That means the percentages averaged over all 96 EUs over the frame time for data such as Active, Stall and Throughput.
**Fig. 44:** Data values averaged across all EUs over the entire frame time.

You can then select a single draw call or a set of calls to see that same data recalculated for the part of the frame you have selected.
After making a selection, in this case calls 91, 94 and 95, the data will be recalculated to represent the data for only those calls.

Additionally, if you captured a stream, it will open in multi-frame view. From there you can select a single frame or multiple frames. If you select multiple frames the data calculated will be the aggregate of the data from all selected frames.

While it is important to understand how the GPU works and what the metrics mean for efficient profiling, you don’t need to analyze each draw call in your frame manually in order to understand the problem type. To help with this sort of analysis, Intel® GPA provides automatic hotspot analysis - Advanced Profiling Mode.

### 16.3.1 Hotspot Analysis

Now that we have some understanding of the EU architecture, let’s look at how that knowledge manifests in the profiler.

When you enable Advanced Profiling Mode Graphics, Graphics Frame Analyzer delineates bottlenecks by bottleneck type and pipeline state. This categorization provides the additional benefit of a fix for one issue often fixing not only the local issue, but rather an entire category of issues.
In Graphics Frame Analyzer enable Hotspot Analysis by clicking on the button on the top left of the tool - shown in Fig 7. The Bar Chart across the top then shows the bottlenecks, and the API Log in the left panel changes to show the bottleneck types. When you click on a bottleneck the metrics viewer will show more details about the bottleneck, with metrics descriptions and suggestions to alleviate the bottleneck.

![Image of Graphics Frame Analyzer showing bottleneck analysis](image)

Hotspot: L3 Cache

Characterization of an L3 Cache Hotspot

When the application has high thread occupancy, close to 90%, that is good. But if the high thread occupancy is coupled with stall, greater than 5-10%, you may have an L3 Cache bottleneck.

With a frame open in Graphics Frame Analyzer, look at the Metrics Viewer Panel on the right, enlarged in Fig. x. Occupancy is more than 90%, but there is still a stall in the EU, which means that EU threads are waiting for some data from memory.
Shader Profiler

For further analysis use the Shader Profiler to see per-instruction execution latencies. As you already know latency is not always equal to stall. However, an instruction with higher latency has a higher probability to cause a stall. And, therefore, when dealing with an EU Stall bottleneck, Shader Profiler gives a good approximation of what instructions most likely caused the stall.

Enable Shader Profiler

Access the shader profiler by doing the following. Click on any shader in the Resource List, in this case SH:17, to display the shader code in the Resource Panel. Then click the flame button at the top of the Resource Pane to see the shader code with the lines taking the most time annotated with the timings, toggle between execution count and duration (percentage of frame time consumed).

Map Shader Code to Assembly Code

For a potential L3 Cache bottleneck, you will also want to see the assembly code, where you will find the send commands from the Send Unit. Click the button in the Resource Panel above the shader code to see the mapping from the shader code to the assembly code.
Identify the Root Cause of the L3 Bottleneck

To find the root cause of an L3 Cache bottleneck, scroll through the assembly code, looking for the send instructions with the longest duration. Then identify which shader source portions caused them.

In the case of the application being profiled in Fig x, above, the CalcUnshadowedAmountPCF2x2 function which samples from ShadowMap and reads the constant buffer is cause of this bottleneck.

Hotspot: Shader Execution

Characterization of a Shader Execution Bottleneck

A Shader Execution bottleneck is characterized by very high thread occupancy and very low stall time. These are good. However, if the application reduced execution time, it is necessary to optimize the shader code itself.
Identify the Root Cause of the Shader Execution Bottleneck

For a shader execution bottleneck, it is necessary to analyze the hotspots in shader source code caused by arithmetic operations. Find these by toggling to Duration Mode in the shader profiler, then scroll through the code to find the lines of shader code that take exceedingly long. CalcLightingColor does calculations involving both simple and transcendental operations. Figure x shows that this function in this single shader consumes about 20% of the total frame time. In order to resolve this bottleneck this algorithm must be optimized.
Characterization of a Thread Dispatch Bottleneck

In this final example of hotspot analysis there is a sequence of draw calls which have a Thread Dispatch bottleneck. In this particular case we have a rather high stall rate (20%) and low thread occupancy (66%). As stated earlier, low occupancy may be caused by an insufficient number of threads loaded on the EU. Thus, instead of directly fixing stall time in shader code, it is necessary, instead, to increase the overall EU Occupancy.

Identify the Root Cause of the Thread Dispatch Bottleneck

Which is better, SIMD8 or SIMD16?

Open Shader Profiler, but in Execution Count mode rather than Duration mode which shows how many times each instruction was executed. In Fig x notice that the pixel shader has been compiled into both SIMD8 and SIMD16. Shader Profiler shows that each instruction in the SIMD8 version was executed 24,000 times, while instructions in SIMD16 were executed 16,000 times - a 1.5 times difference!

It is preferable to have more SIMD16 threads, as they perform twice as many operations per single dispatch, compared to SIMD8 threads. Why so many SIMD8 dispatches? And why should there be 2 SIMD versions for the Pixel Shader?
Examine the Geometry

The geometry for these draw calls is rather fine-grained. The observed anomaly is a result of how the GPU handles pixel shading. The shader compiler produced two SIMD versions, SIMD8 and SIMD16. This is required so that the pixel shader dispatcher can choose which one to execute based on the rasterization result. It is important to know that hardware does not shade pixels one at a time. Instead shading happens in groups. A single pixel shader hardware thread shades 16 pixels at a time for SIMD16. With SIMD16, if a primitive is rasterized into very few or just a single pixel, then the GPU will still shade 16 pixels and will discard all those which were unnecessary. Therefore, in order to discard less, the pixel shader dispatcher schedules SIMD8 for very small primitives. This is the case here. A large number of highly-detailed geometry (many small primitives) produced a huge number of SIMD8 invocations. As you may guess, in order to fix such a performance problem you need to use geometry LODs in your game.

Summary
<table>
<thead>
<tr>
<th>Bottleneck Type</th>
<th>Characterization</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 Cache</td>
<td>High occupancy</td>
</tr>
<tr>
<td></td>
<td>High stall</td>
</tr>
<tr>
<td>Shader Execution</td>
<td>High occupancy</td>
</tr>
<tr>
<td></td>
<td>Low stall</td>
</tr>
<tr>
<td>Thread Dispatch</td>
<td>High stall</td>
</tr>
<tr>
<td></td>
<td>Low occupancy</td>
</tr>
</tbody>
</table>

As shown above, different scenarios require different approaches. At times it is best to speed up CPU work to fully populate the GPU. Other times it is best to optimize shader code. And still others it might be best to change formats, dimensions or layouts of primitives. For each scenario, Graphics Frame Analyzer facilitates analysis of resources to assist developers to make informed decisions about how to optimize frame rate of their applications.

For more ways to optimize GPU performance using Intel® GPA, see Intel® GPA Use Cases as well as Deep Dives and Quick Tips.
17.0 Reference

For more information, see:

- Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference
- Intel® oneAPI Programming Guide
- Intel® Fortran Compiler Classic and Intel® Fortran Compiler Developer Guide and Reference
- Get Started with OpenMP Offload to GPU for the Intel® oneAPI DPC/C++ Compiler and Intel® Fortran Compiler
- OpenMP Features and Extensions Supported in Intel® oneAPI DPC++/C++ Compiler
- Fortran Language and OpenMP Features Implemented in Intel® Fortran Compiler (Beta)
- Developer Reference for Intel® oneAPI Math Kernel Library - C
- OpenMP API 5.2 Specification
- OpenMP API 5.1 Examples
- Data Parallel C++, by James Reinders et al
- SYCL 2020 Specification
- oneAPI Level Zero Specification
18.0 Terms and Conditions

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available security updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

*Other names and brands may be claimed as the property of others. © Intel Corporation.