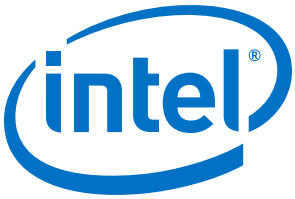


AN 831: Intel® FPGA SDK for OpenCL™

Host Pipelined Multithread

AN-831 | 2017.11.20
Latest document on the web: PDF | HTML

Contents

1 Intel® FPGA SDK for OpenCL™ Host Pipelined Multithread

In this document, a new pipelined framework for high throughput design is proposed. This framework is optimal for processing large input data through algorithms, where data dependency forces sequential execution of all stages or tasks of the algorithm.

1.1 Introduction

The basic idea of this pipelined framework is to build a pipelined architecture and accelerate the process for a large set of input data.

In this architecture, each task is a consumer of the previous task and producer for the next task or stage in the pipeline. Due to the dependency between tasks, it is impossible to run tasks in parallel for a single input data and improve the performance.

However, if each task works on a separate input data, there would be no data dependency and all tasks can run concurrently. Elimination of dependency creates a pipeline of several tasks (that is, steps of the original algorithm) and therefore, can significantly improve the performance of the whole system, especially when processing large amount of input data.

The following figures illustrate the performance of such an algorithm, which includes three sequential tasks without and with pipelined architecture, respectively.

Figure 1. Performance of the Original Algorithm for Multiple Input Data

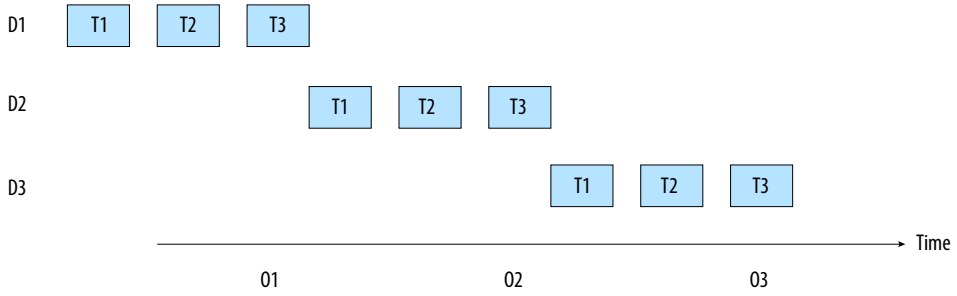
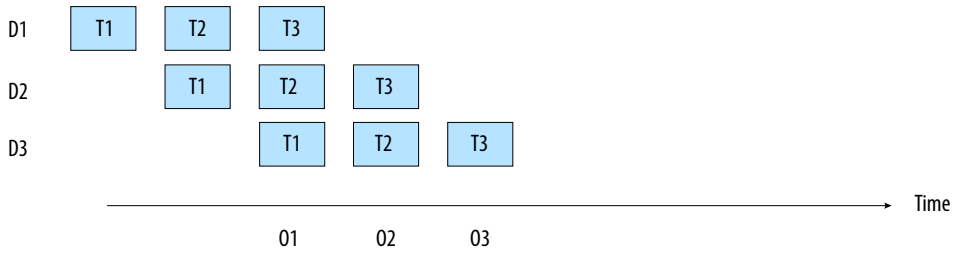


Figure 2. Performance of the Pipelined Algorithm for Multiple Input Data



One of the best applications of this framework is in heterogeneous platforms where high-throughput hardware or platform is used to accelerate the most time consuming part of the application. Remaining parts of the algorithm must run in a sequential order on other platforms such as CPU, to either prepare the input data for the accelerated task or to use the output of that task to prepare the final output. In this scenario, although the performance of the algorithm is partially accelerated, the total system throughput is much lower because of the sequential nature of the original algorithm.

Intel has applied this framework to its data compression reference design. The data compression algorithm is based on the producer-consumer model and has several sequential steps. The most demanding task called *Deflate* is accelerated using FPGA. Yet there are few tasks or steps that must be performed before and after the Deflate process on the CPU, leading to high degradation in the total system throughput. However, by using the proposed pipelined framework, Intel was able to achieve a very high system-throughput (close to the throughput of the Deflate task on FPGA) for multiple input files.

For more information about the proposed pipelining framework, refer to *Pipelining Framework for High Throughput Design*. For information about relevant CPU optimization techniques that can improve this pipelined framework further, refer to *Optimization Techniques for CPU Tasks*. For information about Intel's data compression algorithm, refer to *Design Example: Data Compression Algorithm*.

Related Links

- Pipelining Framework for High Throughput Design on page 4
- Optimization Techniques for CPU Tasks on page 6
- Design Example: Data Compression Algorithm on page 8

1.2 Pipelining Framework for High Throughput Design

Multithreading is a great technique to achieve higher throughput by running multiple tasks concurrently. The framework Intel has built uses both threads and a producer-consumer architecture to optimize the tasks.

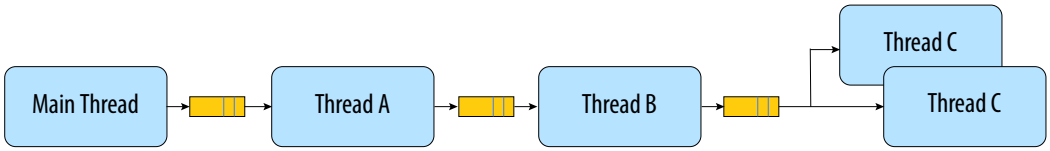
A given algorithm is partitioned into tasks that process data completely before the next task starts. Each task is modeled as a producer and/or a consumer that consumes data produced by the previous task and produces data for the next task. Each task waits for data from the previous task, in a loop on a different thread that runs on separate cores. With continuous streaming of data, the framework forms a pipelined architecture. After the pipeline is filled-up at all stages, output data is generated at the speed of the slowest task in the pipeline. In general, if there are n

tasks in the algorithm or pipeline and all tasks are taking about `ts` time to produce their output, the process time for each input reduces to `ts` instead of `n x ts`. This improvement is presented in Figure 1 on page 3 and Figure 2 on page 4.

Queuing is the best way to transfer work data between these threads or tasks (steps of the pipeline). Main thread might enqueue the incoming inputs or job-items in the input queue for the next task or thread. This thread dequeues and consumes job-items from the front of the input queue as soon as it has the ability to process a new job. Then, it generates a new job-item and pushes it into the input queue for the next task or thread, as illustrated in the following figure.

Note: As observed in the following figure, it is possible to have multiple threads consuming from the same queue in case a task performed by the thread cannot keep up with the input queue speed. In the figure, two Thread Cs are started to match the speed of other stages of the pipeline.

Figure 3. Producer-Consumer Pipelining Framework



The important features of this design are:

- Synchronizing process to access the same queue by a producer and a consumer.
- Designing threads properly to avoid starving or overflowing queues and to maintain efficient performance.

Synchronization

Since each queue is accessed by both producer and consumer tasks, you must protect the queue as a critical area. Mutex lock (`pthread_mutex_t`) is used to control access to the queue (both for enqueuing and dequeuing data from the queue). The enqueue or push function must ensure that the queue is not full and obtain the lock before writing into it. The dequeue or pop function must also lock the queue before reading the front element from the queue, in case the queue is not empty. To efficiently check the status of the queue, a conditional (`pthread_condition_t`) variable is used. If a queue is empty, instead of busy looping, wait for the `pthread_condition_t` variable to complete. The variable releases the lock on the queue allowing another process (producer) to access the queue and push into it. Eventually, the producer sends signal to the consumer after pushing the new item and the consumer thread (waiting on `pthread_condition_t` variable) return to lock the queue, pop an item from it, release the lock, and continue its process.

This implementation minimizes the synchronization process because all the required synchronization is implemented as part of the push and pop functions to or from the queue.

Avoiding Overflow and Starvation

It is important to ensure that the throughput of all pipeline steps (all threads) are in the same range, and the producing and consuming rates are similar. This avoids overflow and starvation of the queues. Hence, the throughput of all pipeline steps must be calculated, and each thread or task is optimized in case its throughput is not in the expected range.

It is possible to have more than one consumer thread in case the process of that stage is too time consuming, as illustrated in Figure 3 on page 5.

Related Links

- Introduction on page 3
- Fine Tuning the Framework Design on page 12

1.3 Optimization Techniques for CPU Tasks

In this section, you learn how to bind or unbind a process or a thread to a specific core or to a range of cores or CPUs, and use cache optimization techniques to optimize cache and improve performance of processors.

Related Links

- Introduction on page 3
- Fine Tuning the Framework Design on page 12

1.3.1 Processor Affinity or CPU Pinning

A preemptive multitasking operating system consistently reschedules jobs on a multiple core processor for optimal system performance.

Each time a job or a thread is preempted and another job or thread is scheduled, the operating system might assign a new job or thread to a core that is free. Due to this reason, the core to which a given thread or process is assigned to can be different each time. Each time a thread is assigned to a core, the processor must copy a thread's relevant data and instructions into its cache, if the data is not already in the cache. This means that the cache must be updated each time a thread is rescheduled on a new core.

Processor affinity or CPU pinning enables applications to bind or unbind a process or a thread to a specific core or to a range of cores or CPUs. The operating system ensures that a given thread executes only on the assigned core(s) or CPU(s) each time it is scheduled, if it was pinned to a core.

Processor affinity takes advantage of the fact that the remnants of a process execution remains valid when the same process or thread is scheduled a second time on the same processor. So, the cache may still be valid when a thread execution is re-scheduled after being preempted. This helps scale the performance on multiple core processor architectures that share the same global memory and have local caches (UMA Architecture).

Operating systems support CPU affinity through APIs. The *NX platform has a scheduler library that provides APIs for CPU affinity. The following code snippet when called within a thread sets its affinity to `cpu_id`:

```
cpu_set_t cpu_set;
CPU_ZERO (&cpu_set);
CPU_SET (cpu_id, &cpu_set);
sched_setaffinity (0, sizeof(cpu_set_t), &cpu_set);
```

Related Links

Processor Affinity in GZIP on page 14

1.3.2 Cache Optimizations

Use data prefetching and cache coalescing techniques to optimize cache and improve performance of processors.

Prefetching

Prefetching data or instructions from a slower global memory to the local cache can improve performance of algorithms on modern processors. Prefetching data works best when you access your data sequentially rather than from random locations in the memory. Blocks of data are copied one time from the global memory to cache, thus minimizing cache misses and frequent data fetch to the cache. For example, consider adding one million numbers from an array in the global memory, as shown in the following code snippet:

```
Sum = 0;
for (int i = 0; i < 1000000; ++i) {
    Sum += a[i];
}
```

For each data access, CPU verifies if the data is available in its cache and if not, it loads the data from the global memory to its local cache and adds the data to the `Sum`. Changing the above code to prefetch data with additional directives reduces cache misses. This can improve the performance of the code many folds. *NX platforms have built-in APIs for prefetching.

```
void __builtin_prefetch (const void *addr,...)
```

The value of `addr` is the address of the memory to prefetch. It has two optional arguments:

- `rw`: This argument is a compile-time constant (0 or 1).
  - A value of 0 (default) means that the prefetch is preparing for a read.
  - A value of 1 means that the prefetch is preparing for a write to the memory address.
- `locality`: This argument is also a compile-time constant integer between zero and three.
  - A value of 3 (default) means that the data has a high degree of temporal locality and should be left in all levels of cache possible.
  - Values of 1 and 2 mean that a low or moderate degree of temporal locality.
  - A value of 0 means that data has no temporal locality, so it need not be left in the cache after the access.

Cache Coalescing

CPUs load 64-bytes of data at a time from the cache to registers, update the data and write it back to the cache. If consecutive memory locations are to be updated in consecutive cycles, CPU adds barriers to prevent wrong updates, that is, the CPU ensures that the previous update is completed before the next data is loaded. This can cause performance bottlenecks in computations when CPUs attempt to vectorize and perform multiple operations in parallel. Hence, in such scenarios where the processor has to update a sequence of locations in the memory, a different type of cache optimization technique, Cache coalescing, can be used.

Cache coalescing optimizes memory access so that the processors can do multiple operations in parallel. For example, consider the following frequency counter:

```
for (int i = 0; i < length; ++i) {
     freq_counter[[data[i]]++;
 }
```

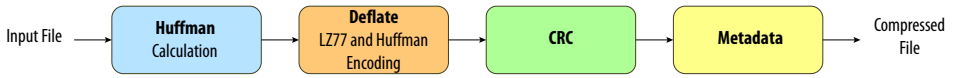
The statement `freq_counter[data[ch]]++` is composed of three instruction executions at the processor level: load, increment or update, and store. The simplest form is to interleave memory operations or to leave extra padding of memory between adjacent locations that must be updated.

1.4 Design Example: Data Compression Algorithm

In this section, an example design of the data compression algorithm is presented to show how it influences the total system performance.

The following figure illustrates sequential tasks of the data compression algorithm. Most of these tasks must be executed in a sequential order for each input file.

Figure 4. Data Compression Algorithm Sequential Tasks



- **Huffman** calculation finds the frequency table of all symbols in the input file and generates the Huffman code.
- **Deflate** including LZ77 and Huffman encoding algorithms generate the compressed data stream.

  Attention: This is the most time consuming task of the sequence.

  — LZ77 engine searches the input file to find repetitions included in the file and replaces them with a pointer. This pointer includes distance to the location of previous occurrence of that string and length that matches.

  — Huffman encoding compresses the file furthermore by using the Huffman code.
- **CRC** is calculated for the input file and later added at the end of the compressed file for the purpose of error detection. This is the only task that can happen in parallel to Deflate task.
- Output generation called **Metadata** is the last task in the sequence that puts the output stream together, as defined in RFC 1951.

As illustrated in Figure 4 on page 8, the total time and latency required to generate the compressed output from any input file is the summation of processing time of each individual task.

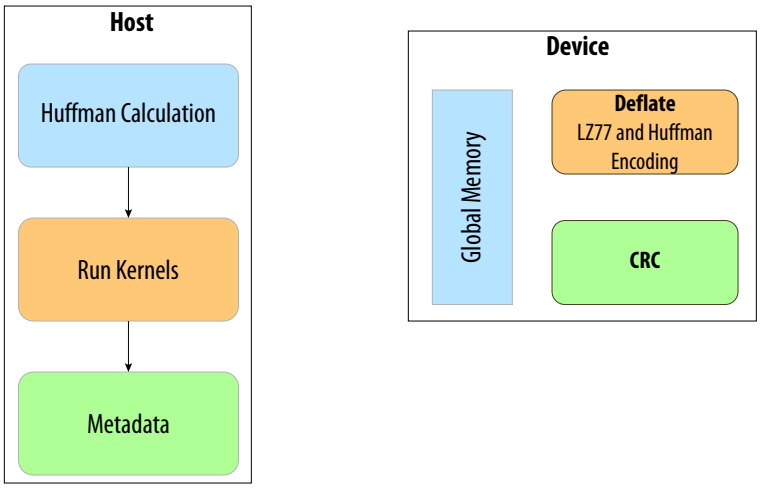
To accelerate this algorithm, FPGA platform is used. OpenCL* is a very powerful tool that makes implementation on a hardware much faster when compared to the RTL design, especially for the software programmers. Using Intel FPGA SDK for OpenCL for this purpose helps in optimizing the Deflate algorithm (the most time consuming task) for FPGA platform and achieving a very high throughput of more than 3 GB/s for input files larger than 4 MB.

CRC is also offloaded into FPGA with the same throughput as Deflate task. The CRC task can run in parallel to the Deflate task on the FPGA hardware since it requires only the input file for calculating the checksum.

Notice: Development and optimizations on the FPGA hardware is not covered in this document.

The following figure illustrates how the tasks are arranged on a CPU and an FPGA:

Figure 5. Arrangement of Tasks on CPU and FPGA



After adding the Huffman calculation and Metadata process to the CPU as a pre-process or a post-process for deflate algorithm, a huge reduction was observed in the total system performance. By using the pipelining framework along with the host channel streaming feature to pass data between the host and device, total system performance improved back to the same throughput of the Deflate algorithm in situations where multiple input files were used for compression. This new proposed system architecture also minimizes latency.

For more information about the host channel streaming feature provided by the Intel FPGA SDK for OpenCL, refer to the *Host Channel Streaming Feature* section. For information about applying the pipelining framework to the data compression algorithm, refer to the *Pipelining Framework Details*. For information about CPU optimization required for fine tuning the framework, refer to *Fine Tuning the Framework Design* and for final results, refer to *Performance Evaluation*.

Related Links

- Introduction on page 3
- Host Channel Streaming Feature on page 10
- Pipelining Framework Details on page 10
- Fine Tuning the Framework Design on page 12
- Performance Evaluation on page 14

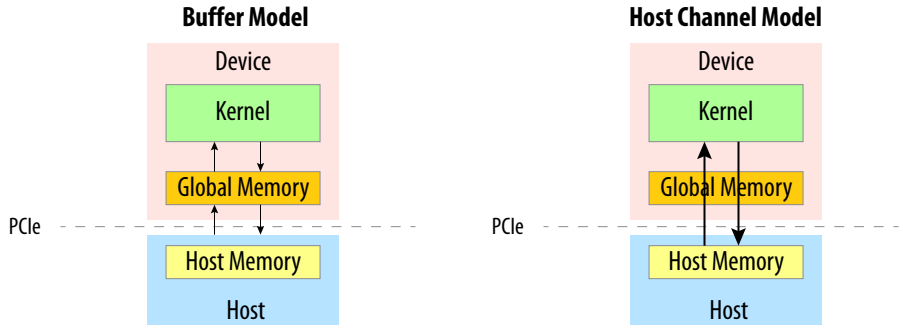
1.4.1 Host Channel Streaming Feature

Communication between a host and a device is usually a time consuming, especially for large sized data.

Input data is transferred from the host memory into the global memory on the device. To use the data, kernels access this global memory and copy the required data into their local memory. This process of transferring data adds a huge latency since the whole data must be transferred from the host to global memory before starting any process on the device. The same latency is observed for the output data since the whole output must be generated first, saved into the global memory and then, transferred into the host memory.

The following figure illustrates the host channel streaming feature and how it provides a lower overhead by eliminating the need to access global memory. When you implement this feature, kernel is launched only once into the device and used for multiple input files, as they are streamed directly into the device from the host.

Figure 6. Host Channel vs Memory-Based Data Communication



Intel FPGA SDK for OpenCL supports streaming data between a host and a device to eliminate the global memory access in both directions and directly pass data to the local memory through host channels, and also read back from the device. This reduces latency and enables platform (CPU or FPGA) to start processing the streamed data.

Related Links

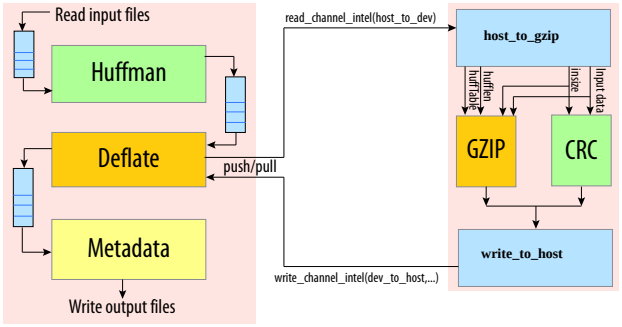
Design Example: Data Compression Algorithm on page 8

1.4.2 Pipelining Framework Details

This section provides a detailed explanation of the data compression tasks in the pipelined framework architecture.

As illustrated in the following figure, several threads were generated to process the data compression algorithm. The main thread handles the input files and enqueues them into the input queue of the first task or thread, and also starts the rest of the threads. There are three more threads that follow the same provider-consumer model and communicate through specific queues, which are protected by lock and have efficient push and pop functions for any access, as explained in Pipelining Framework for High Throughput Design on page 4:

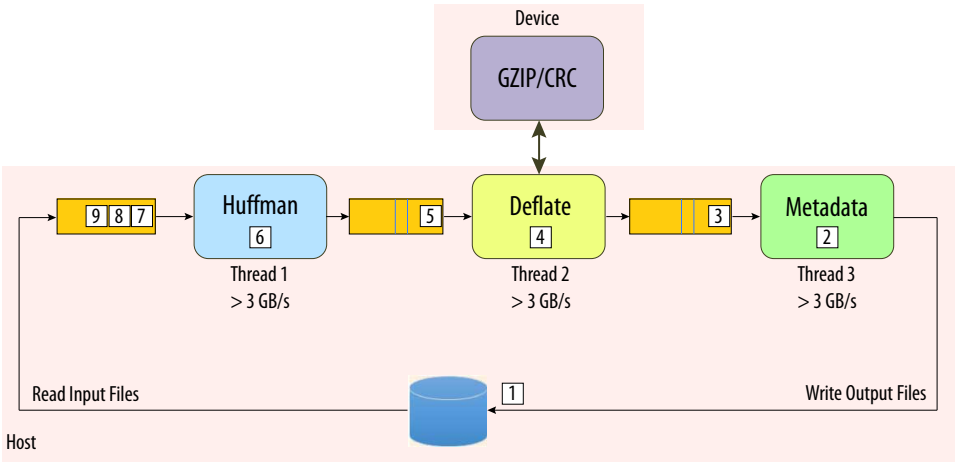
Figure 7. Data Compression Tasks in the Pipelined Framework Architecture



- **Huffman** calculation thread calculates the frequency table of the input file and generates the Huffman code. This code must be passed to the Deflate and CRC thread.
- **Deflate** and **CRC** thread starts executing Deflate and CRC kernels on the device and feeds them with the required input, including the input file and Huffman code. Simultaneously, this thread reads back all the generated outputs from the device. Communication between the host and device on this thread is done in two different threads using the host channel streaming feature. One thread streams input file and Huffman code into the device. Second thread reads back the generated compressed file and the CRC value. The received results are passed to the next thread through the connecting queue.
- **Metadata** thread is executed on the CPU. This thread reads the compressed data by popping them from the input queue and processing them to support RFC 1951 standard. Metadata thread generates the final compressed file.

Although execution of each of these tasks depends on the data generated by the previous thread, they can still run concurrently on different input files as illustrated in the following figure.

Figure 8. Multiple Input File Process in the Designed Pipelining Framework



Each thread pops an input job or item from the input queue, processes and generates data required for the next thread, and pushes the output into its output queue. Therefore, while Huffman thread is calculating the Huffman code for the third input file, Deflate and CRC thread is processing the second input file, and Metadata thread is preparing the output results of the first input file simultaneously.

Related Links

Design Example: Data Compression Algorithm on page 8

1.4.3 Fine Tuning the Framework Design

In this section, you learn how to optimize threads and fine tune your framework design, to have an efficient system.

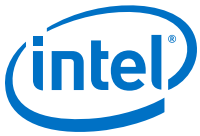
As discussed earlier in the *Pipelining Framework for High Throughput Design* topic, to have an efficient system, it is important to design the pipeline with a similar throughput for all steps of the data compression algorithm.

- Deflate and CRC thread is executed on FPGA with a throughput higher than 3 GB/s. Therefore, ensure that all other threads are faster than this step to have the system throughput similar to the device throughput.
- Metadata thread is optimized by benefiting from multithreading and processing on larger chunks of data. This step also achieves throughput higher than 3 GB/s.
- Huffman calculation thread that generates the Huffman code must calculate the frequency table based on each input file. This is a time consuming task, especially for very large files (in the order of N). To accelerate this process, use optimizations discussed in *Optimization Techniques for CPU Tasks*.

For more details about how optimization techniques are applied to Huffman frequency table calculation, refer to *Huffman Code Generation*.

Related Links

- Design Example: Data Compression Algorithm on page 8
- Pipelining Framework for High Throughput Design on page 4
- Optimization Techniques for CPU Tasks on page 6



- Huffman Code (Tree) Generation on page 13

1.4.3.1 Huffman Code (Tree) Generation

With the existing FPGA software solutions, one of the main bottlenecks is the Huffman code or tree generation. The Huffman code generation first computes the frequency of each character in the input data, and then uses this frequency to construct the Huffman tree and related codes corresponding to each character.

It is observed that the frequency computation is the major bottleneck but not the tree generation from the codes. Following is the code snippet that computes the frequency:

```
unsigned int freq[256]
    for (int i = 0; i < length; ++i) {
        freq[data[i]]++;
    }
```

The existing implementations with the Huffman frequency table computed on a CPU had a maximum throughput of up to 4.5 GB/s and the worst case was up to 1.2 GB/s with multiple threads, when data was homogeneous. The following two aspects of computing Huffman data frequency table is considered:

- Improving worst case scenarios
- Improving threads

Worst Case Scenarios

With homogeneous data, throughput of the frequency computation is always one fourth of the best case. It is noticed that each frequency computation involves a load, add, and store operation (`freq[ch]++`). This means that if the adjacent characters in the data stream are similar or closer in the ASCII table (for example, A and B), the compiler and CPU cannot vectorize and pipeline it efficiently. Vectorization does multiple load, add, and store in parallel. This is because CPUs access caches in blocks (usually 64-bytes), and if same blocks are updated in adjacent instructions, compiler and CPU have to add barriers to make updates on adjacent locations sequentially. For more information, refer to the Prefetching section in Cache Optimizations on page 7 topic.

You can solve this issue by adding a padding between the frequency table entries for each character. In case of the Huffman frequency counter, the loop was modified and updated to have 16 different tables. The data is accumulated after the loop ends.

```
unsigned int freq[256]

unsigned int freq_tab[16][256];
for (int i = 0; i < length/16; ++i) {
   for (int j = 0; j < 16; ++j) {
      freq_tab[j][data[i]]++;
   }
}

for (int i = 0; i < 16; ++i) {
   for (int j = 0; j < 256; ++j) {
      freq[j] += freq_tab[i][j];
   }
}
```

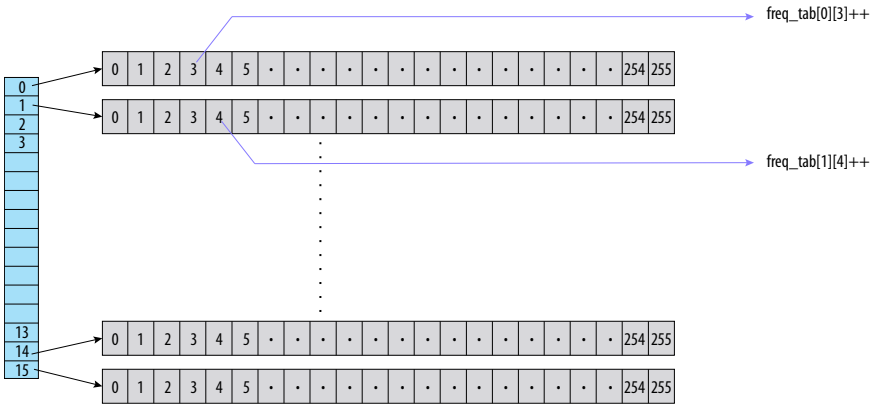
With the above distributed table update, the frequency computation throughput became identical for all cases.

Threads in Huffman Frequency Counter

Data is divided into multiple blocks and the frequency table of each block is computed independently in separate threads. The frequency tables are then merged at the end to generate the final table.

While experimenting, it was observed that as the number of threads increased, the frequency computation did not scale. This issue can be improved with CPU affinity of each thread by assigning threads to specific CPUs, to keep caches valid for longer than usual. By setting CPU affinity to threads, you can improve the best performance of the frequency computation (up to 15 GB/s).

Figure 9. Update 16 Tables for 16 Adjacent Chars in the Stream



Related Links

Fine Tuning the Framework Design on page 12

1.4.3.2 Processor Affinity in GZIP

In a high-throughput design, Processor Affinity enables assigning each thread loop to a specific core.

As mentioned earlier, Huffman code generation, kernel execution, and metadata generation, all execute a thread loop that reads or consumes data object from an input queue and delivers the processed data or job item to an output queue that acts as an input queue for the next stage in the pipeline. Along with these thread loops, the frequency counter in Huffman code generation uses up to three threads internally to obtain high-throughput. These threads are also assigned to specific cores for optimal performance, as mentioned earlier.

Related Links

Processor Affinity or CPU Pinning on page 6

1.4.4 Performance Evaluation

Intel has tested this framework on different benchmarks and by providing multiple input files. It was observed that the system throughput is same as the kernel throughput or even better.

The kernel throughput is greater than 3 GB/s for input files larger than 130 MB. This result determines that the design is very competitive for RTL design, and more powerful when compared to the CPU alone.

Related Links

Design Example: Data Compression Algorithm on page 8

1.5 Document Revision History

Table 1. Document Revision History of the Intel FPGA SDK for OpenCL Host Pipelined Multithread

Date	Version	Changes
November 2017		Initial release.