Intel Software

THE PARALLEL UNIVERSE

Supercharging Python* for Open Data Science

Getting Your Python Code to Run Faster Using Intel® VTune™ Amplifier XE

Parallel Programming with Intel® MPI Library in Python

Issue 25 2016

CONTENTS

LETTER FROM THE EDITOR

James Reinders, an expert on parallel programming, is coauthor of the new ***Intel® Xeon Phi™ Processor High Performance Programming—Knights Landing Edition*** (June 2016), and coeditor of the recent ***High Performance Parallel Pearls Volumes One and Two*** (2014 and 2015). His earlier book credits include ***Multithreading for Visual Effects*** (2014), ***Intel® Xeon Phi™ Coprocessor High-Performance Programming*** (2013), ***Structured Parallel Programming*** (2012), ***Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*** (2007), and ***VTune™ Performance Analyzer Essentials*** (2005).

Democratization of HPC

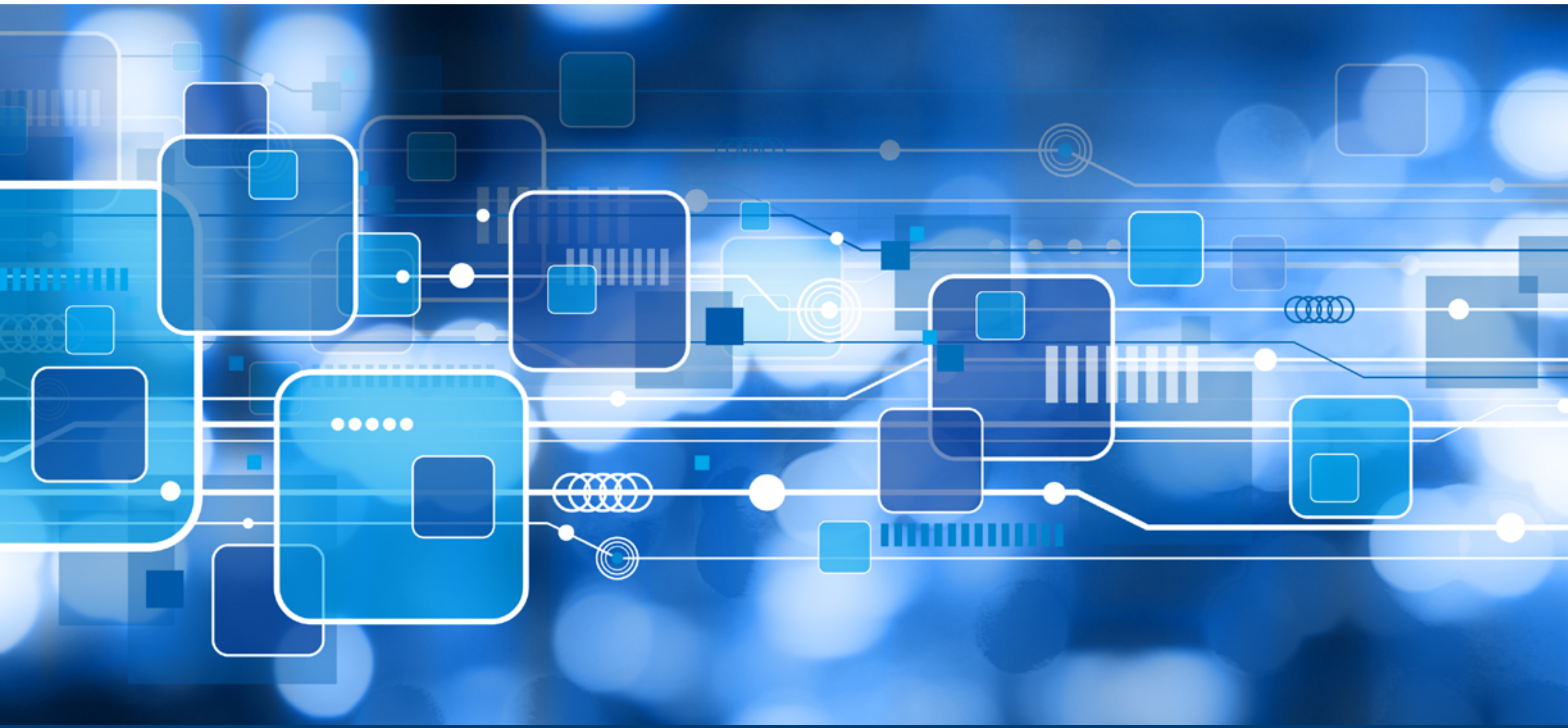
Python* is thriving, due in no small part to its simplicity, expressive syntax, and abundance of libraries. One area where this open source programming language is gaining a lot of traction is data analytics and machine learning. But how can developers merge the productivity benefits of Python with the performance that only parallel computing can bring?

Well, for starters, Intel has released the beta version of Intel® Distribution for Python, which offers faster performance from Python packages powered by Intel® Math Kernel Library. We've also added Python support in our most popular tools, such as the new version of Intel® VTune™ Amplifier (currently part of Intel® Parallel Studio XE 2017 Beta), which detects hidden performance hotspots in applications with Python code mixed with native C/C++ code and extensions.

This issue of The Parallel Universe walks us through some of these tools. Our feature article, **Supercharging Python* with Intel and Anaconda* for Open Data Science**, explores the leading open data science platform, Anaconda, that is accessible to beginner developers yet powerful enough for big data projects. **Getting Your Python* Code to Run Faster Using Intel® VTune™ Amplifier XE** is a how-to on profiling and optimizing Python code. And **Parallel Programming with Intel® MPI Library in Python*** shows how parallel programming is possible with Python and the dominant programming model used in HPC.

These efforts support not only Intel's immediate interest in helping developers tap the true potential of Intel® Xeon® and Xeon Phi™ products, but also our larger mission as a leader in the democratization of technical computing and HPC. Although we like to explore the most compelling use cases for parallel computing in *The Parallel Universe*, history tells us that today's supercomputer advances will be commonplace computing within a decade, and everyday computing in the decade after that. The trend of democratization of HPC is very exciting. As technologies mature, Intel is increasingly finding ways to make such high-performance systems more accessible. This in turn energizes more transformational science and engineering that utilize such computers. And that translates to a brighter future for all of us.

James Reinders
July 2016



SUPERCHARGING PYTHON* WITH INTEL AND ANACONDA* FOR OPEN DATA SCIENCE

The Technologies That Promise to Tackle Big Data Challenges

Travis Oliphant, *CEO and Founder*, Continuum Analytics

The most exciting areas of computing today are high-performance computing (HPC), big data, and data science. But, until recently, there was little to no bridge between them. Hardware advances have facilitated breakthroughs in HPC, but open data science (**ODS**) has relied primarily on process improvements from the thriving Python* and R* communities to push it forward.

Without optimization, high-level languages like Python lack the performance needed to analyze increasingly large data sets. Fortunately, **Continuum Analytics**—founders and creators of **Anaconda***—have found a way to marry development ease with state-of-the-art computation speed.

Anaconda is the leading **ODS** platform that includes a high-performance Python distribution linked with the **Intel® Math Kernel Library** (Intel® MKL). This offers packages and technology that are accessible to beginner Python developers and powerful enough to tackle big data projects. Anaconda offers support for advanced analytics, numerical computing, just-in-time compilation, profiling, parallelism, interactive visualization, collaboration, and other analytic needs.

By combining recent HPC advances with Python-based data science, the Anaconda community is driving the production of high-performing—yet low-cost and accessible—analytics technologies that promise to tackle big data challenges that mark the industry's future.

Accelerating Data Science Innovation with ODS and Python

The world of scientific computing and data science is dominated by two open source languages: Python and R. These languages pave a simple path for domain experts—such as quantitative scientists, data scientists, data engineers, and business analysts—to address their analytics needs without needing to be veteran programmers.

Python is particularly approachable. Taught in computer science classrooms around the world, it is a readable language with a thriving community. Python emphasizes adherence to a standard ("Pythonic") way of writing code that encourages clarity. As the "**Zen of Python**" poem states, "simple is better than complex."

Python is an ideal language for data science. And, Anaconda delivers a robust ODS platform that embraces and maximizes performance for Python, R, Hadoop* and an ever-growing list of open source technologies. Anaconda is proven to both scale up and scale out on numerous architectures—from Raspberry Pi* to high-performance clusters—and provides the flexibility to meet an enterprise's changing performance needs.

Also, Python has a "batteries-included" philosophy. The language has a sizeable standard library of packages to facilitate common programming tasks, and it integrates with hundreds of third-party libraries tailored to domains, ranging from policy modeling to chemistry. The Anaconda platform includes these standard packages and makes it simple to add packages to enable data science teams using Python to model almost anything one could imagine—without sacrificing performance or ease of use. Thanks to Anaconda, these packages are easy to set up in one's deployment environment, since the platform provides out-of-the-box functionality with a simple installation.

Python and Anaconda's blend of simplicity, breadth, and performance encourages domain experts and data scientists to experiment with and manipulate models—which drives innovation. As Intel chief evangelist James Reinders puts it, "If you don't think of programming as your primary job and you are a data scientist, chemist, or physicist, then programming's kind of a necessary evil. Something like Python is a more reasonable investment that delivers the performance necessary, so scientists are able to focus more on their science rather than programming."

Thankfully, domain experts have dozens of Python-based options to help them exploit the computational power of today's hardware for ODS. All of the technologies discussed below come standard with the Anaconda Enterprise subscription and interoperate effectively within it. Also, all are accessible to laymen programmers, including scientists, economists, and business leaders.

Python Compilation for Fast, Simple Code

One capability in Anaconda that domain experts will find useful is the Numba* just-in-time (JIT) compiler for Python that interoperates with NumPy*, the foundational analytics Python package on which all other Python analytics are based. Numba enables users to quickly mark sections of Python code with Python decorators (a syntactic convenience of the language), which are then compiled to machine code and executed rapidly.

Typical speedups range from two to 2,000 times faster Python execution by using the native machine code generated from the compiler instead of relying on interpreted Python. In addition to providing performance gains, Numba also reduces programmers' mental overhead because it does not require context switching. This means that Numba users can operate fully within the Python ecosystem without needing to rewrite performance-critical sections into a traditional compiled language such as C/C++ or Fortran. A great example of Anaconda's potential can be found in **TaxBrain***, a Web app that uses the **taxcalc*** package, a recent open source U.S. tax policy modeling project in which Anaconda and Numba optimized and simplified NumPy code.

Anaconda also helps data science teams increase computational performance through parallelization. By inserting a single-line decorator, users can instruct any function to run in as many threads as desired. For tasks easily separated into multiple threads, Numba offers programmers a surprisingly painless way to utilize the multiple cores in modern processors (including GPUs). For example, decorating a function with Numba to run simultaneously on eight cores will, by means of a one-line addition, deliver an eightfold speed boost.

It's these speed benefits that make Python an ideal language for quick development of algorithms and data science applications. According to Reinders, "Python offers a host of tools for lightning-fast, interactive development. Its **parallel computing** capabilities are unmatched and, when coupled with Anaconda and Intel MKL, Python can help developers reach new frontiers of machine learning and deep learning algorithms."

Anaconda is being actively developed to improve interoperability with other HPC tools. One initiative underway is to create an interface to the **Intel® Threading Building Blocks** (Intel® TBB) library. Such interoperability is a hallmark of the ODS movement.

Also available is an extreme performance distribution of Anaconda that uses the Intel® C++ Compiler to further boost performance for Intel®-based hardware and future-proofs Python scripts to ensure scaling will occur to utilize new Intel innovations.

Anaconda Coupled with Intel MKL Optimizes Key Analytics Packages in Python

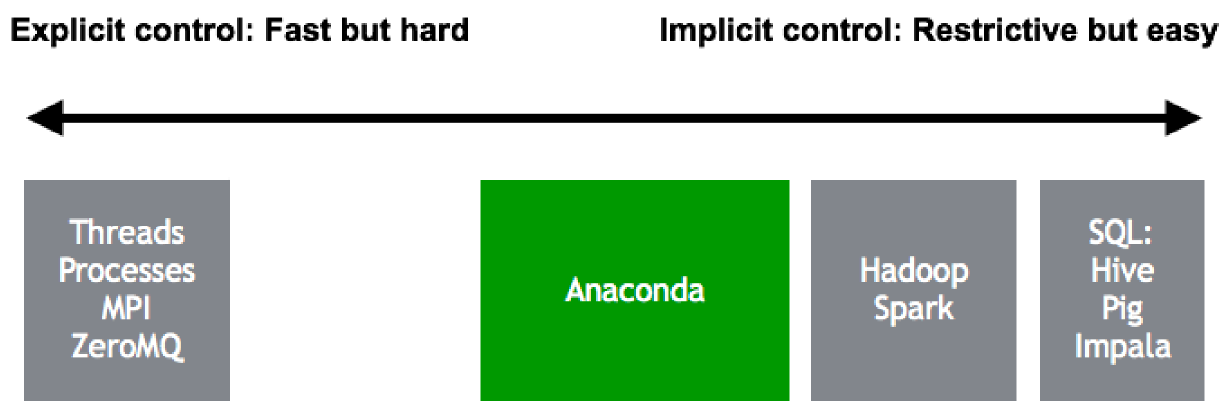
Anaconda bolsters Python's numerical computing performance by Intel MKL to deliver optimized, precompiled libraries for the fastest computations and throughput. This boosts linear algebra and fast Fourier transformation (FFT) operations underpinning the most popular Python numerical analysis libraries, including NumPy, SciPy*, scikit-learn*, and Numexpr*. The Intel MKL library speeds up numerical analysis from 10 to 100 times.

NumPy has a key feature that automatically maps a basic kernel function defined only for a small number of elements over a much larger array. This basic kernel is then executed many times in a typical NumPy function call. The basic kernels are usually built with compiled code (C, C++, or Fortran), which is what gives NumPy its raw speed. Anaconda with the Python compiler allows for these fundamental kernels to be written with Python and easily extends NumPy at a fundamental level using simple Python syntax. These kernels can also include calls to Intel MKL library functionality to provide even more capabilities.

Anaconda provides a number of Intel MKL-enhanced fundamental NumPy functions so that users get faster performance out of the box for commonly used functions, including trigonometry, square roots, and algebraic function calls. "Thanks to Intel MKL, today's NumPy and SciPy users can see application performance skyrocket without needing to change a single line of code," according to Reinders.

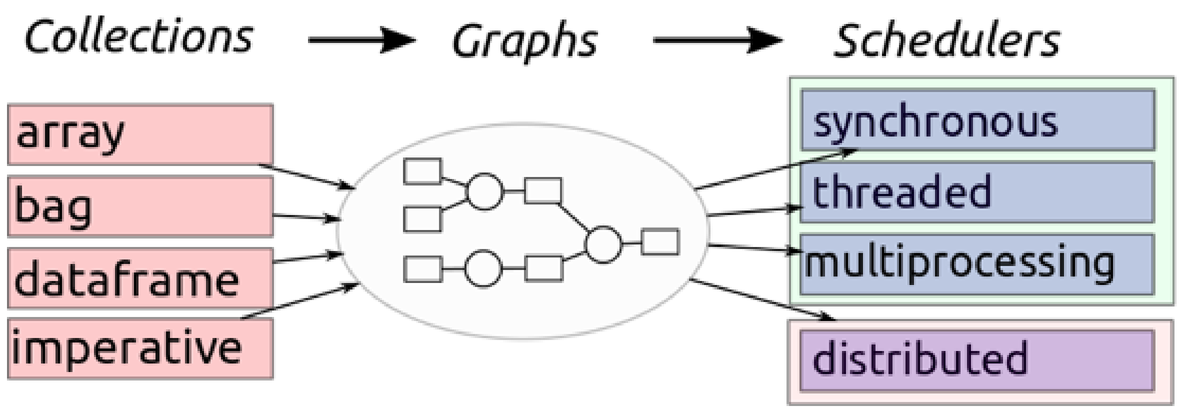
Powerful and Flexible Distributed Computing for ODS

To further increase performance, data science teams using Anaconda can take advantage of the robust and high-performance parallel computing framework that operates with low overhead and latency and minimal serialization for fast numerical analysis **(Figure 1)**. This enables flexible, distributed processing for analytic workloads across HPC and Hadoop clusters.


Explicit control: Fast but hard
Implicit control: Restrictive but easy
Threads
Processes
MPI
ZeroMQ
Anaconda
Hadoop
Spark
SQL:
Hive
Pig
Impala

1 Spectrum of parallelization

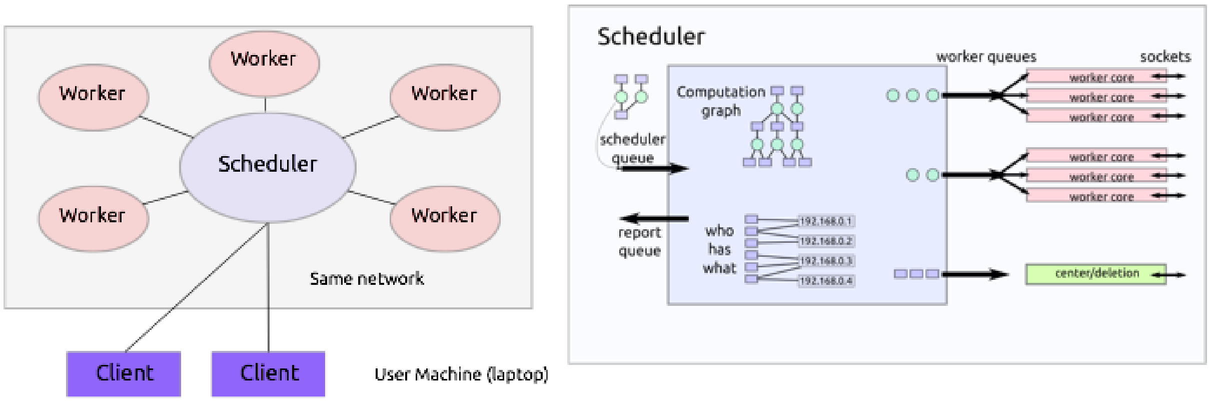
Anaconda gives data science teams a high-level interface to write familiar array-like or dataframe code for computations **(Figure 2)**. The new or existing Python scripts are then turned into directed acyclic graphs (DAGs) of tasks. These DAGs serve to break up large workloads into many smaller processing tasks. The DAG can then be executed by a built-in or proprietary task scheduler. Anaconda comes out of the box with schedulers for multithreaded, multiprocessing, or distributed processing across the nodes in a single machine or across the HPC or Hadoop cluster. The clusters can be cloud-based or bare metal clusters. The DAG allows for extremely quick processing of tens or hundreds of GBs of data using familiar array or dataframe operations on multicore CPUs.



2 Anaconda* parallel computing framework

Performance advantages come from array computation and automatic chunking of the tasks that are well suited to modern hardware. For example, Anaconda will break up a 10-million-row array into 10-million-row segments, resulting in a DAG with many calls to functions that each work on chunks of the large array. This general approach is very common in financial computing. Anaconda surfaces the technique to make it widespread. Data science teams have complete flexibility to revise a DAG because it is represented as a Python dictionary. One can modify chunking or insert calls as needed. The graph can then be sent to any of three built-in schedulers for multithreading, multiprocessing, or multinode distributed processing. If none of the preexisting schedulers are appropriate for an analysis, then Anaconda enables developers to create their own schedulers, either from scratch or by modifying a built-in reference.

Figure 3 shows an example with a distributed scheduler.



3 Distributed scheduler example

On a single machine, Anaconda intelligently streams data from disk or shared memory and uses all the cores of the modern CPU. On a cluster, Anaconda enables data science teams to rapidly prototype and deploy novel algorithms on hundreds of nodes. Significant speedups of 10 to 100 times over PySpark* have been demonstrated for text processing, statistics, machine learning, and image processing.

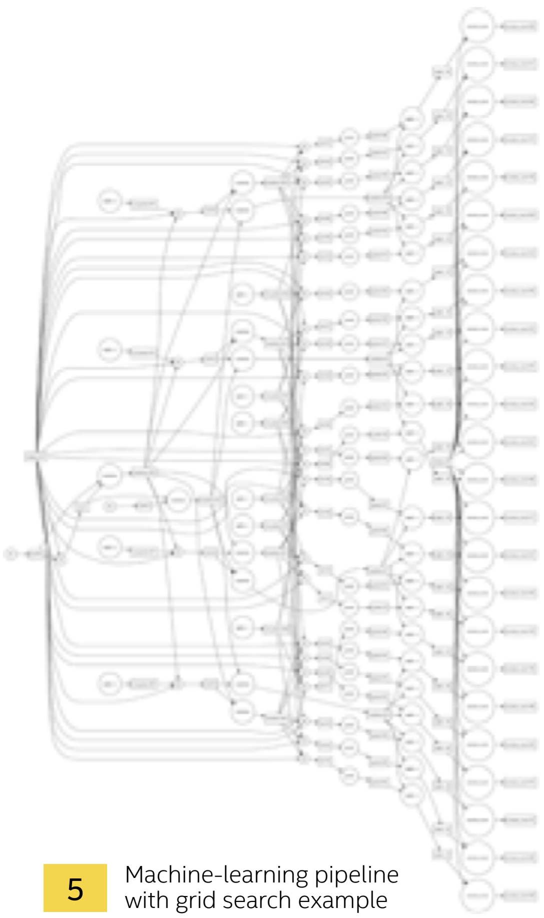
Figure 4 shows a machine-learning pipeline example.

4 Machine-learning pipeline example

A parallel computing framework is particularly suited to data science. It uses the same APIs as NumPy and the popular **pandas*** library. With the same API calls, code can run in parallel. For example, if one is using many-core Intel® Xeon Phi™ coprocessors with two parallel method calls, Anaconda can multiply analysis speeds by manyfold. Fortunately, with all of the high-level techniques available, Anaconda is simple to use and does not require parallelism expertise to benefit from it.

Figure 5 shows a machine-learning pipeline plus grid search example.

Data science teams can easily take advantage of progressive hardware advances with Anaconda by writing code once in the style of array computing. Anaconda can then parallelize this code to automatically maximize processing speeds across cores or nodes. Built with the "PyData" stack in mind, the Anaconda parallel computing framework delivers speed enhancements to packages designed for many disciplines including climate science (xarray), image processing (scikit-image), genomics (scikit-allel), and machine learning (scikit-learn).



5 Machine-learning pipeline with grid search example

Anaconda in-notebook, data, and code profilers and diagnostics can also be used to identify bottlenecks and points of contention, since data and workloads change over time. The DAGs can be modified to remediate bottlenecks. In this way, the initial DAG serves as a fast starting point and is flexible enough to be modified and enhanced over time to evolve with the ever-changing data and clusters. The parallel computing framework is resilient and elastic. The included resource schedulers (multithreaded, multiprocessing, and distributed processing) can all be integrated with third-party schedulers including Torque*, Slurm*, and YARN*. With this flexibility and performance, Anaconda fully maximizes infrastructure investments to accelerate time-to-value at the best price/performance ratio.

Open source software is on the rise because of its advantages over proprietary solutions.

Profiling Python with Intel® VTune™ Amplifier and Dask.diagnostics*

One of the best ways data science teams can address a performance problem is through profiling. Both Intel and Continuum Analytics offer capabilities to help profile Python code.

For function profiling, Python's standard library already contains cProfile* to gather function timing data. SnakeViz*, a browser-based graphical viewer for the output of cProfile's module, can then present a performance display to help programmers locate bottlenecks. Users can also perform memory and line profiling of Python using the heapy* and line_profile* packages.

Although these are useful tools for identifying performance bottlenecks, they are often insufficient to help programmers solve performance issues during large-case profiling of large-scale computations involving arrays or dataframes. Most profiling tools indicate how long a function takes to execute, but rarely provide adequate information about the arguments passed to that function. The profiler, for instance, may show the argument is a scalar, dictionary, or array, but not show the size, type, or shape of the array. High-quality profiling tools such as **Intel VTune Amplifier** reveal the details of how a particular function is called at line level in your Python and/or native code, including demonstrating which threads are waiting on a lock. For many big data applications, this becomes crucial to diagnosing performance hot spots or bottlenecks.

Anaconda provides a number of Intel® MKL-enhanced NumPy* functions so that users get faster performance.

This sort of profiling gets even harder for parallel computing in a cluster. It is important for the entire data science team to have the appropriate diagnostics and visibility to be able to assess the impact that performance bottlenecks may have on their code—especially as it scales across a cluster of machines.

Fortunately, profiling for parallel code on a cluster can also be carried out with Anaconda. Anaconda provides a number of features to assist with this notoriously tricky task. Scheduler callbacks allow inspection of distributed scheduler execution, and its progress bar shows the level of completion of a long-running computation. The profiler for the distributed scheduler provides information on the scheduling, resources, cache—such as start and finish times—memory, and CPU usage, as well as size metric and other values.

Price/Performance Ratio and HPC Data Science

Whichever software ecosystem a data science team chooses—whether proprietary or an ODS solution—costs matter. At one time, data storage was a limiting cost. But storage's price per terabyte has since dropped by many orders of magnitude.

That cost decrease enabled the rise of big data, but it hasn't rendered data processing costs irrelevant. Today's organizations gauge these costs with a price/performance ratio. Teams want a low ratio, enabling them to crunch through terabytes of cheaply stored data to produce useful results.

One way that data science teams can achieve a low price/performance ratio in their systems is to exploit ODS. This approach remains viable throughout multiple advances in hardware. As hardware processing power increases, performance capabilities increase alongside it. If companies can run existing software on newer hardware, then it reduces their overall costs and lowers the price/performance ratio. Purchasing new software whenever new hardware is released raises this ratio, because new software comes with licensing fees, training costs, and developer time needed to relearn systems or rewrite code.

ODS presents an advantage in price/performance. Many of the technologies described previously are open source: they are freely available, interoperable, and constantly updated by the open source community. Because these open source tools are not proprietary, they are not locked to vendor release schedules, licensing restrictions, and continued costs.

Some of these proprietary software costs come from purchasing and renewing licenses for analytics products. But costs can also come from expensive consultants and add-on packages. The costs inherent to proprietary software add up.

On the other hand, open source software can greatly reduce or eliminate many of these costs. Many ODS packages—including those referenced previously—are freely available and easily installed. Furthermore, they lack the continued licensing costs of their proprietary counterparts and rarely involve steep consultant fees, because the growing community of ODS enthusiasts provides free help.

To be clear, ODS is not a magic wand that makes all costs go away. Even with an ODS approach, software-as-a-service products that use open source packages have developer costs. And hardware costs will contribute to the total cost of organizations' analyses.

In many cases, a *purely* open source approach to analytics can also create unexpected challenges. Without support through an ODS vendor, software implementation costs can skyrocket because experts are needed to set up and maintain software. These experts can be hard to find and/or vet to be sure you are getting a quality solution.

The Speed of Change

This is a revolution going on across industries. Projects in the public sector, like TaxBrain*, are showcasing that ODS has a larger role to play in transforming business, government, and social good. Financial services are making use of ODS solutions for diverse teams composed of quants, traders, developers, and analysts.

In the business world, advances in interactive visualization through Anaconda* are establishing democracies in companies that give everyone simple and digestible data and high-impact business value to an easily consumable big data application.

All of these seemingly disparate factions are united by the common coinage of ODS, and individuals from business leaders to physicists and developers stand to benefit. Thanks to new tools that marry ODS to HPC, the pace of progress in understanding our world through data has never been quicker.

Disparate worlds are colliding—HPC, big data, and advanced analytics. For organizations to cost-effectively harness the untapped value in their data, they need to adopt open data science. Open source technology was not built for speed (never mind warp speed). However, the pace of business today requires the speed of light. Anaconda, supercharged by Intel® MKL, delivers high-performance analytics and makes it easy for data science teams to buckle up for the warp-speed journey.

Also, if an organization is managing its own ODS ecosystem, then it will bear the brunt of whatever change management it attempts. It can be difficult to transition data and employees from in-house software (sometimes cobbled together from free software by previous employees) to open source software—particularly if it is accompanied by hardware changes. With a field as innovative and individually tailored as data science, that's a bigger hidden cost than many managers realize.

Taken together, these hidden costs can mean that an idealized, zero-cost data science initiative can actually cost organizations more than expected. The middle road of a responsive ODS vendor that provides a thoroughly tested, fully compatible, and regularly updated ecosystem of software is likely the winner in terms of hassle and total cost of ownership.

The ODS Community and the Future of Data Science

At its core, ODS is not just about innovation and low costs. It is about community.

Python and ODS are excellent models for community-built software. And open source software advantages over proprietary solutions means open source is on the rise. Each year, new Python developers emerge from university programs. These individuals join online software forums like GitHub or StackOverflow. They submit patches to open source software or start their own open source projects. Because of ODS's clear benefits, data scientists and domain experts are learning Python and its suite of packages—including NumPy, pandas, matplotlib, dask, scikit-learn, and countless others.

Anaconda, the leading ODS platform, has opened the doors to an exhilarating new era where data science teams can focus on creating high value and impactful data science solutions without worrying about being boxed into a corner. Anaconda will easily exceed their current and evolving scaling requirements.

For More Information

Numba >

Anaconda Subscriptions >

Open Data Science >

Anaconda White Paper: The Journey to Open Data Science >

GETTING YOUR PYTHON* CODE TO RUN FASTER USING INTEL® VTUNE™ AMPLIFIER XE

The Tool Provides Line-Level Profiling Information with Very Low Overhead

Kevin O'Leary, *Software Technical Consulting Engineer*, Intel Corporation

In the latest version of Intel® VTune™ Amplifier XE 2017 (currently available free in the **Intel® Parallel Studio XE Beta** program), we now have the ability to analyze Python* scripts and find exactly where time is being spent in an application. In this article, we show how to use this new Intel VTune Amplifier Python functionality.

Why Do You Need Python Optimization?

- Python is used to power a wide range of software, including those where application performance matters. For example: Web server code, complex automation scripts (even build systems), and scientific calculations.
- Python allows you to quickly write some code that may not scale well, but you won't know it unless you give it enough workload to process.

How Do You Find Places in Code That Need Optimization?

There are several ways:

- **Code examination.** This is easiest in that you don't need any tools except a code editor. However, it is not the easiest one to execute—it assumes you know how certain code constructs perform, which might not be the case. Obviously, this won't work for an even moderately large codebase because there is just too much information to grasp.
- **Logging.** This is done by making a special version of source code augmented with timestamp logging. It involves looking at the logs and trying to find the part of your code that is slow. This analysis can be tedious and also involves changing your source.
- **Profiling.** Profiling is gathering some metrics on how your application works under certain workloads. In this article, we focus on CPU hotspot profiling (i.e., searching for and tuning functions that consume a lot of CPU cycles). There is also the possibility of profiling things like waiting on a lock, memory consumption, etc. For a typical Python application, the CPU is usually the culprit, though, and even some memory consumption points could be identified by finding CPU hotspots (because allocating and freeing memory, especially in stuff like big Python sequence objects, takes a significant amount of CPU usage).

You would need some type of tool to profile an application, but the productivity gains from using them is well worth it.

Overview of Existing Tools

Event-based tools collect data on certain events, like function entry/exit, class load/unload, etc. The built-in Python cProfile profiler (and sys.setprofile /sys.settrace API) are examples of event-based collections.

Instrumentation-based tools modify the target application to collect the data itself. It can be done manually, or can have support in a compiler or other tool, and it can be done on the fly.

Statistical tools collect data at regular intervals. Functions that are most executed should be at the top of a “samples by function” distribution. These tools also provide approximate results, and are much less intrusive (they don’t affect the target application too much, and are not affected much by frequent calls to small functions). The profiling overhead dependency on the workload is much less, too.

Profiling Python Code Using Intel VTune Amplifier XE

Intel VTune Amplifier XE now has the ability to profile Python code. It can give you line-level profiling information with very low overhead (about 1.1X–1.6X measured on the Grand Unified Python Benchmark). It is supported on both Windows* and Linux*. Some key features:

- Python 32- and 64-bit
- 2.7.x and 3.4.x branches—tested with 2.7.10 and 3.4.3
- Remote collection via SSH support
- Rich UI with multithreaded support, zoom and filter, and source drill-down

  Supported workflows:

  - Start application, wait for it to finish
  - Attach to application, profile for a bit, detach
- Allows profiling of code at all stages, including most Python code executed at initialization and shutdown of the interpreter (e.g., atexit-registered functions)

The steps to analyze Python code are as follows:

- Create an Intel VTune Amplifier project
- Run a basic **hotspot analysis**
- Interpret result data

Create a Project

To analyze your target with Intel VTune Amplifier, you need to create a project, which is a container for an analysis target configuration and data collection results.

a. Run the `amplxe-gui` script, launching the Intel VTune Amplifier GUI.

b. Click the ≡ menu button and select **New > Project ...** to create a new project. The **Create a Project** dialog box opens.

c. Specify the project name, `test_python`, that will be used as the project directory name.

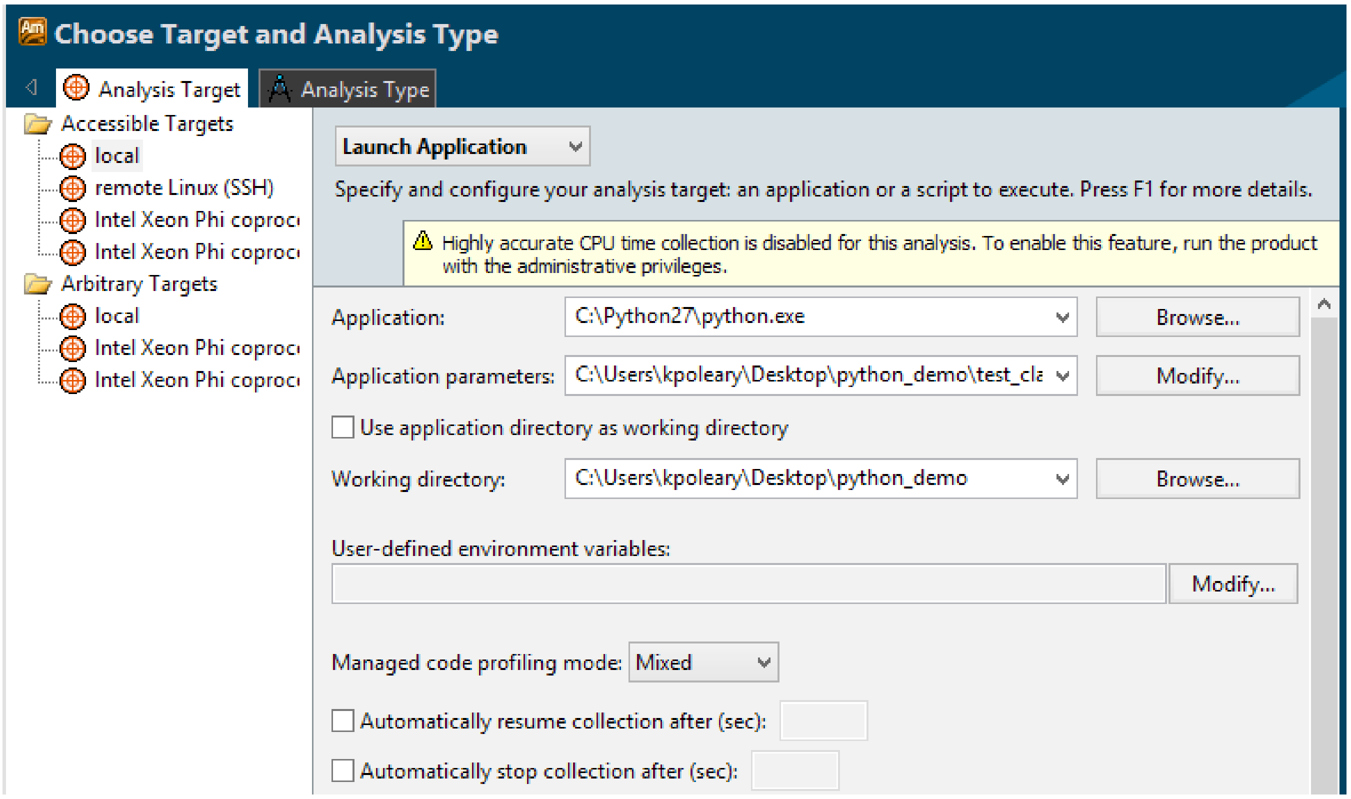
Intel VTune Amplifier creates the `test_python` project directory under the `$HOME/intel/ampl/projects` directory and opens the **Choose Target and Analysis Type** window with the **Analysis Target** tab active **(Figure 1)**.

a. From the left pane, select the **local** target system and from the right pane select the **Application to Launch** target type.

b. The configuration pane on the right is updated with the settings applicable to the selected target type.

c. Specify and configure your target as follows:

- For the **Application** field, browse to your Python executable
- For the **Application parameters** field, enter your Python script
- Specify the working directory for your program to run
- Use the **Managed code profiling mode** pulldown to specify **Mixed**



Specify and configure your analysis target: an application or a script to execute. Press F1 for more details.

Highly accurate CPU time collection is disabled for this analysis. To enable this feature, run the product with the administrative privileges.

Application: C:\Python27\python.exe

Application parameters: C:\Users\kpoleary\Desktop\python_demo\test_cla

Use application directory as working directory

Working directory: C:\Users\kpoleary\Desktop\python_demo

User-defined environment variables:

Managed code profiling mode: Mixed

Automatically resume collection after (sec):

Automatically stop collection after (sec):

1 Creating a project in Intel® VTune™ Amplifier

Run a Basic Hotspot Analysis

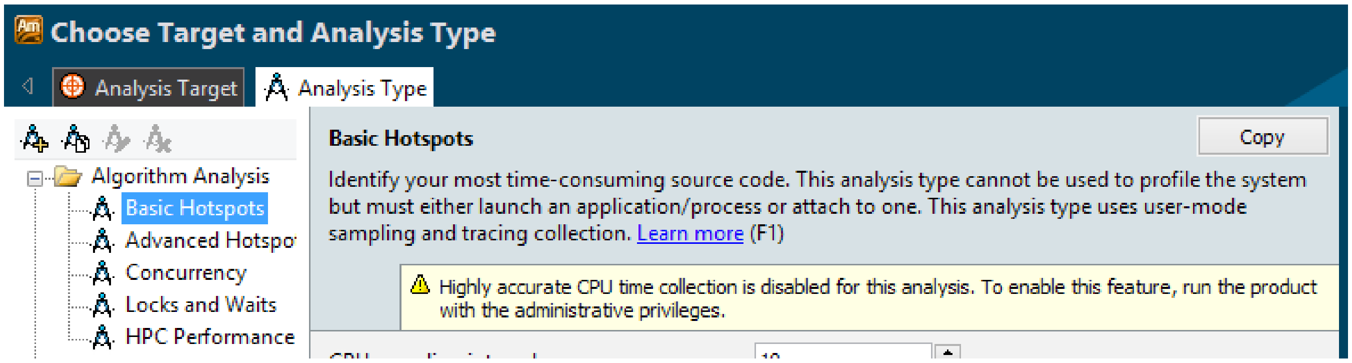
a. Click the **Choose Analysis** button on the right to switch to the **Analysis Type** tab.

b. In the **Choose Target and Analysis Type** window, switch to the **Analysis Type** tab.

c. From the analysis tree on the left, select **Algorithm Analysis > Basic Hotspots**.

d. The right pane is updated with the default options for the Hotspots analysis.

e. Click the **Start** button on the right command bar to run the analysis.



Basic Hotspots

Identify your most time-consuming source code. This analysis type cannot be used to profile the system but must either launch an application/process or attach to one. This analysis type uses user-mode sampling and tracing collection. Learn more (F1)

Highly accurate CPU time collection is disabled for this analysis. To enable this feature, run the product with the administrative privileges.

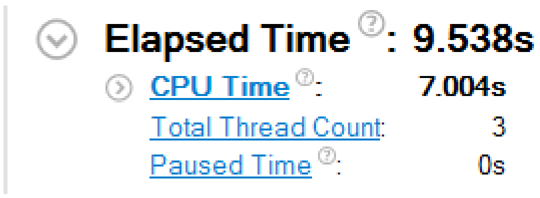
Interpret Result Data

When the sample application exits, Intel VTune Amplifier finalizes the results and opens the **Hotspots** by viewpoint where each window or pane is configured to display code regions that consumed a lot of CPU time. To interpret the data on the sample code performance, do the following:

- **Understand the basic performance metrics** provided by the Python Hotspots analysis.
- **Analyze the most time-consuming functions** and CPU usage.
- **Analyze performance per thread.**

Understand the Python Hotspots Metrics

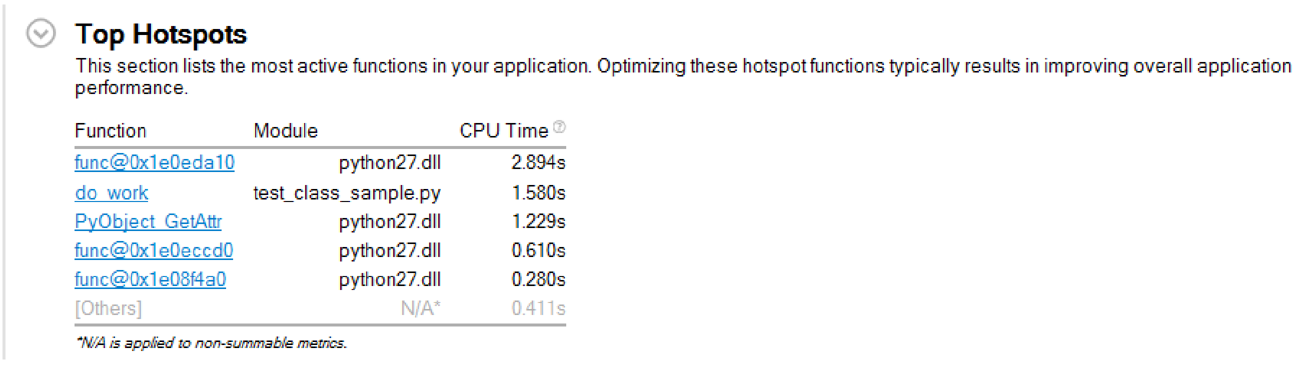
Start analysis with the **Summary** window. To interpret the data, hover over the question mark icons ⓘ to read the pop-up help and better understand what each performance metric means.



Elapsed Time:	9.538s
CPU Time:	7.004s
Total Thread Count:	3
Paused Time:	0s

Note that **CPU Time** for the sample application is equal to 18.988 seconds. It is the sum of CPU time for all application threads. **Total Thread Count** is 3, so the sample application is multithreaded.

The **Top Hotspots** section provides data on the most time-consuming functions (*hotspot functions*) sorted by CPU time spent on their execution.

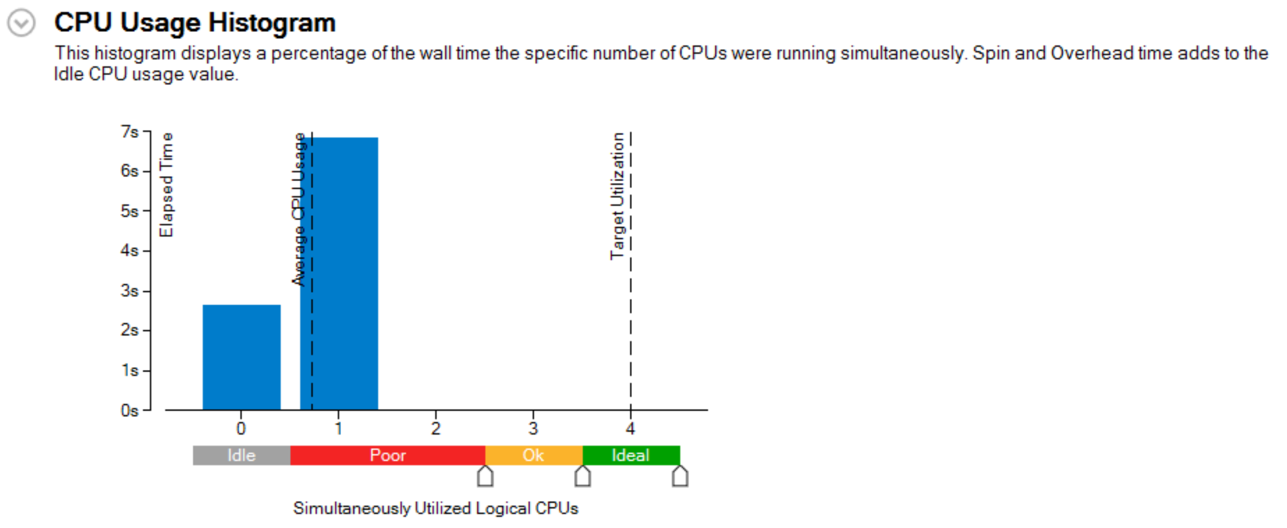


Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
func@0x1e0eda10	python27.dll	2.894s
do_work	test_class_sample.py	1.580s
PyObject_GetAttr	python27.dll	1.229s
func@0x1e0eccd0	python27.dll	0.610s
func@0x1e08f4a0	python27.dll	0.280s
[Others]	N/A*	0.411s

*N/A is applied to non-summable metrics.



CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

BLOG HIGHLIGHTS

Developer Access Program for Intel® Xeon Phi™ Processor (codenamed Knights Landing)

BY MICHAEL P. >

Intel is bringing to market, in anticipation of general availability of the Intel® Xeon Phi™ processor (codenamed Knights Landing), the Developer Access Program (DAP). DAP is an early access program for developers worldwide to purchase an Intel® Xeon Phi™ processor-based system. This is a stand-alone box that has a single bootable Knights Landing processor for developers to start developing codes, optimizing applications, and getting to see the performance.

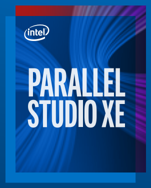
Read more >

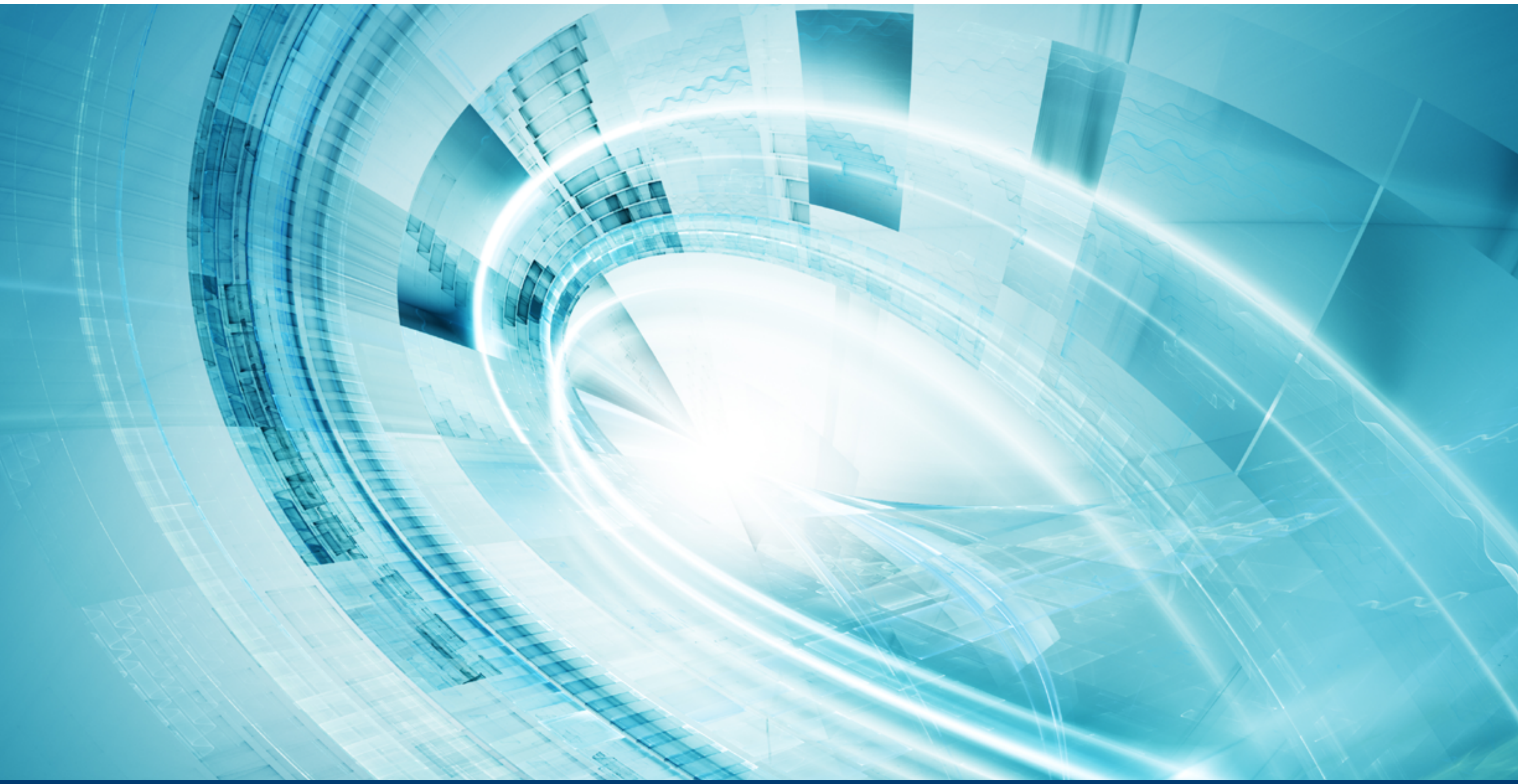
Running Mixed Mode Analysis

Python is an interpreting language and doesn't use a compiler to generate binary execution code. Cython* is an interpreting language, too (but C-extension), that can be built to native code. Intel VTune Amplifier XE 2017 beta can fully support reporting of hot functions both in Python code and Cython code.

Summary

Tuning can dramatically increase the performance of your code. Intel VTune Amplifier XE now has the ability to profile Python code. You can also analyze mixed Python/Cython code as well as pure Python. Intel VTune Amplifier has a powerful set of features that will allow you to quickly identify your performance bottlenecks.





PARALLEL PROGRAMMING WITH INTEL® MPI LIBRARY IN PYTHON*

Guidelines and Tools for Improving Performance

Artem Ryabov and Alexey Malhanov, *Software Development Engineers*, Intel Corporation

High-performance computing (HPC) is traditionally associated with software development using compiled languages such as Fortran and C. Nowadays, interpreted high-level languages like Python* have a wide range of modules, allowing for easy development of scientific and engineering applications related to modeling, simulation, and design. But an obvious question arises about the possibility of creating parallel applications in Python.

This article describes how parallel programming is possible with Python and the Message Passing Interface (MPI), the dominant programming model used in HPC on distributed memory systems. The main issues that arise on the interface between scripting languages and **MPI** often are related to performance.

We'll provide guidelines on developing a parallel Python application using **mpi4py** as a Python wrapper around the native MPI implementation (**Intel® MPI Library**). mpi4py is included in the Intel® Distribution for Python (starting with the **2017 Beta**). We also consider techniques to help improve the performance of MPI applications written with mpi4py and discuss how to use Intel tools such as **Intel® Trace Analyzer and Collector** to analyze parallel Python applications.

MPI for Python

MPI for Python (mpi4py) provides an object-oriented approach to MPI. The interface was designed to translate MPI syntax and semantics of the standard C++ MPI bindings to Python.

In general, mpi4py can communicate any built-in or user-defined Python object, mostly using Python object serialization facilities. The serialization may introduce significant overhead to memory and processor utilization, especially in the case of large buffer transfers. This is unacceptable. Fortunately, mpi4py supports direct communication of any object with a standard Python mechanism provided by some types (e.g., strings and numeric arrays). It provides access, on the C side, to a contiguous memory buffer (i.e., address and length) containing the relevant data. In conjunction with the capability of constructing user-defined MPI datatypes describing complicated memory layouts, this feature enables the implementation of many algorithms involving multidimensional numeric arrays directly in Python, with negligible overhead, and almost as fast as compiled Fortran, C, or C++ codes.

Usage

Figure 1 shows a simple "Hello World" example written with mpi4py.

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
name = MPI.Get_processor_name()
if rank == 0:
  print "Hello world: rank %d of %d running on %s" % (rank, size, name)
  for i in xrange(1, size):
    rank, size, name = comm.recv(source=i, tag=1)
    print "Hello world: rank %d of %d running on %s" % (rank, size, name)
else:
  comm.send((rank, size, name), dest=0, tag=1)
```

1 Sample application

In this simple example, `comm` is a communicator object. MPI functions `Get_size()`, `Get_rank()`, `send()`, and `recv()` are methods of this communicator object.

The **Figure 1** example can be run with the commands in **Figure 2**.

```
$ source <intel_mpi>/bin/mpivars.sh

$ source <intel_python>/bin/pythonvars.sh

$ mpirun -ppn 1 -n 2 -hosts node01,node02 python helloworld.py
Hello world: rank 0 of 2 running on node01
Hello world: rank 1 of 2 running on node02
```

2 Commands

The `mpivars.sh` script sets the environment for Intel MPI Library. `pythonvars.sh` sets necessary environment variables for Intel Distribution for Python. The `mpirun` utility is used to launch the MPI application.

Performance

As stated earlier, there are two types of communications in mpi4py:

- **Communications of generic Python objects based on serialization.** In mpi4py, they are lowercase methods (of a communicator object), like `send()`, `recv()`, and `bcast()`. An object to be sent is passed as a parameter to the communication call, and the received object is simply the return value.
- **Communications of buffer-like objects.** This corresponds to the method names starting with uppercase letters (e.g., `Send()`, `Recv()`, `Bcast()`, `Scatter()`, and `Gather()`). In general, buffer arguments to these calls must be specified explicitly by using a list/tuple, like `[data, MPI.DOUBLE]` or `[data, count, MPI.DOUBLE]` (the former uses byte-size data and the extent of the MPI data type to define the count).

Communications of generic Python objects may produce additional memory/CPU overhead due to object serialization. This may negatively impact performance. Communications of buffer-like objects have near-C speed, so performance impacts are minimal.

For better performance, the general recommendation is to use uppercase MPI communications for Python interface methods wherever possible.

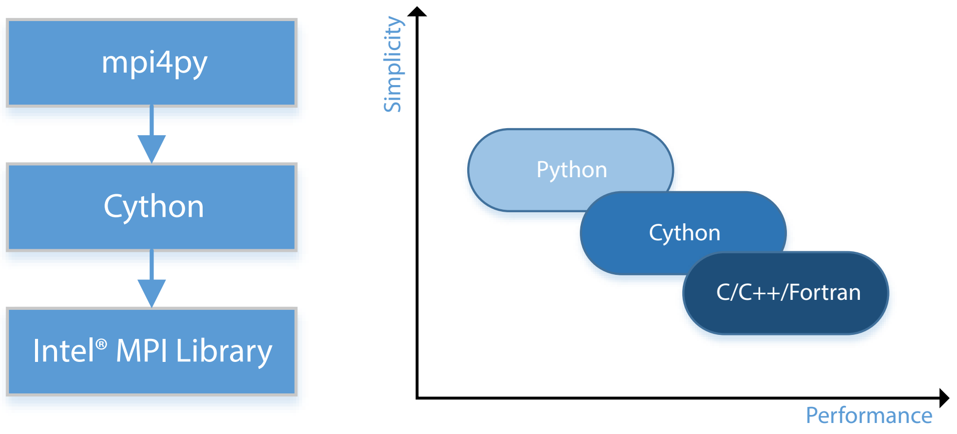
Another recommendation to improve the performance of MPI for Python with Intel MPI Library is to turn on the large message transfer mechanism for shared memory with the `I_MPI_SHM_LMT=shm` environment variable (see **Intel MPI Library documentation** for details).

mpi4py vs. Native MPI Implementation

An interesting exercise is comparing the mpi4py overhead to the native Intel MPI Library implementation.

Internally, mpi4py is implemented with the Cython* language. Cython is very similar to the Python language, but Cython supports C function calls and the declaration of variables and class attributes as C types. This allows the compiler to generate very efficient C code from Cython code. This makes Cython the ideal language for wrapping external C libraries and fast C modules that speed up the execution of Python code. So if you write your MPI code in Python, mpi4py will translate it to C calls with Cython.

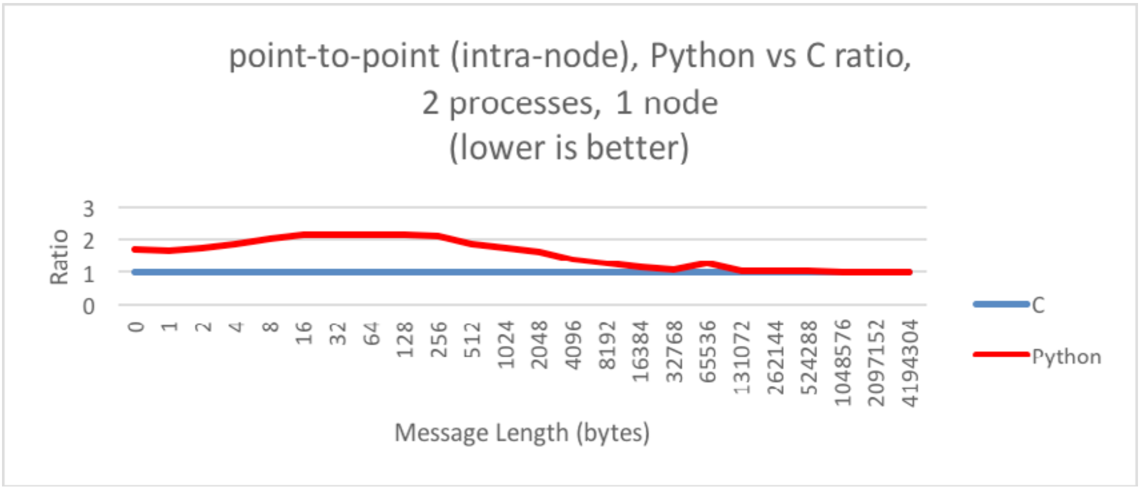
Schematically, it looks like **Figure 3**.

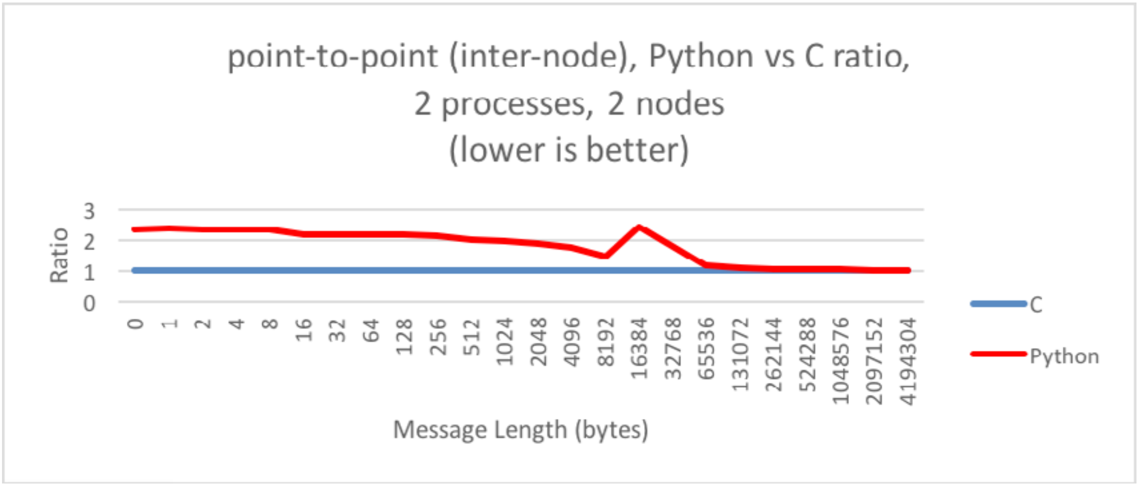


3 mpi4py schematic

Accelerate your machine learning Python* applications on the fastest processors. Intel® Distribution for Python* optimizes your code to speed up computational packages (NumPy, SciPy/scikit-learn) and more.

The corresponding performance experiments with some MPI point-to-point operations confirm this **(Figure 4)**. The similar picture is for the collective operations.





The **osu_latency** benchmark is used for point-to-point performance measurements.

Configuration: Hardware: Intel® Xeon® Processor E5-2699 v4 (2.20 GHz), 64GB RAM, Interconnect: Intel® Omni-Path Host Fabric Interface. Software: Red Hat* Enterprise Linux* 7.1, Intel® C++ Compiler 2017 Beta, Intel® MPI Library 2017 Beta, **OSU Micro-Benchmarks**. I_MPI_SHM_LMT=shm environment variable is used for each run.

4 Performance experiments

The analysis shows that the performance of MPI for Python is fairly good. mpi4py is slower than the native MPI implementation, up to 2.5x with small/medium message sizes, and has similar performance with large sizes.

Applications

The number of applications that use Python's flexibility and MPI's communication efficiencies has grown dramatically. One major project is **petsc4py**, which represents Python bindings for PETSc (Portable Extensible Toolkit for Scientific Computations). PETSc is a suite of data structures and routines for **parallel computing** solutions of scientific applications modeled by partial differential equations that support MPI.

MPI for Python is one of the main components of the ParaView multiplatform data analysis and visualization application, in case users need to implement filters in parallel. And finally, the **yt project** represents an integrated science environment for solving astrophysical problems that also utilizes MPI for Python.

Profiling with Intel Trace Analyzer and Collector

You can profile an MPI for Python application with Intel Trace Analyzer and Collector, as shown in **Figure 5**.

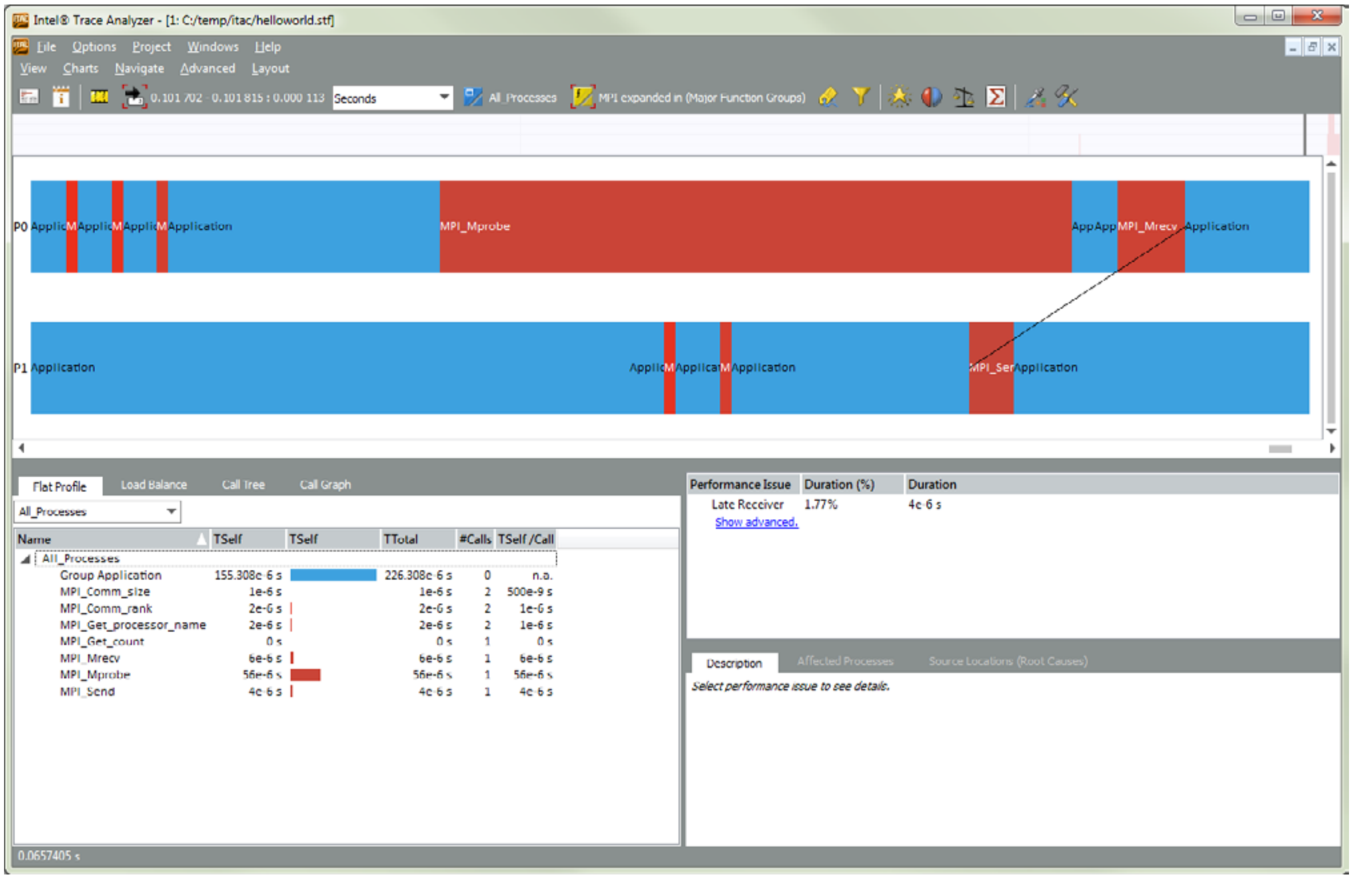
```
$ source <itac_dir>/bin/itacvars.sh
$ export VT_LOGFILE_NAME=helloworld.stf
$ export VT_LOGFILE_FORMAT=SINGLESTF
$ export LD_PRELOAD="<itac_dir>/intel64/slib/libVT.so <impi_dir>/intel64/lib/release_mt/libmpi.so"

$ mpirun -ppn 1 -n 2 -hosts node01,node02 python helloworld.py
Hello world: rank 0 of 2 running on node01
Hello world: rank 1 of 2 running on node02
[0] Intel(R) Trace Collector INFO: Writing tracefile helloworld.stf in <dirname>
```

5 Profiling with Intel® Trace Analyzer and Collector

The `VT_LOGFILE_NAME` and `VT_LOGFILE_FORMAT` variables are optional. They are used mainly for more convenient operation with the trace file. The commands in Figure 5 generate a trace file, `helloworld.stf`, and an auxiliary file, `helloworld.prot`. The trace file can be viewed with the `traceanalyzer` utility **(Figure 6)**.

```
$ traceanalyzer helloworld.stf &
```

6 traceanalyzer utility


Intel® Trace Analyzer - [1: C:/temp/itac/helloworld.stf]
File Options Project Windows Help
View Charts Navigate Advanced Layout
0.101 702 - 0.101 815 : 0.000 113
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0
Applic M Applic M Applic M Application
MPI_Mprobe
App App MPI_Mrecv Application
P1
Application
Applic M Applica M Application
MPI_Ser Application
Flat Profile
Load Balance
Call Tree
Call Graph
All_Processes
Description
Affected Processes
Source Locations (Root Causes)
0.0657405 s

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	155.308e-6 s		226.308e-6 s	0	n.a.
MPI_Comm_size	1e-6 s		1e-6 s	2	500e-9 s
MPI_Comm_rank	2e-6 s		2e-6 s	2	1e-6 s
MPI_Get_processor_name	2e-6 s		2e-6 s	2	1e-6 s
MPI_Get_count	0 s		0 s	1	0 s
MPI_Mrecv	6e-6 s		6e-6 s	1	6e-6 s
MPI_Mprobe	56e-6 s		56e-6 s	1	56e-6 s
MPI_Send	4e-6 s		4e-6 s	1	4e-6 s

Performance Issue	Duration (%)	Duration
Late Receiver	1.77%	4e-6 s

Show advanced.

Select performance issue to see details.

7 Profiling MPI for Python* applications with Intel® Trace Analyzer and Collector

Note that Intel Trace Analyzer and Collector reflects all low-level MPI calls, so some are not present directly in the original MPI for Python code, as shown in **Figure 7**. In this example, Python `recv()` produces `MPI_Mprobe()/MPI_Mrecv()`.

Resources

1. **Intel® MPI Library**
2. **Intel® Distribution for Python***
3. **Intel® Trace Analyzer and Collector**
4. **mpi4py and VTK**
5. **mpi4py official page**
6. **PETSc**
7. **yt project official page**

THE OTHER SIDE OF THE CHIP

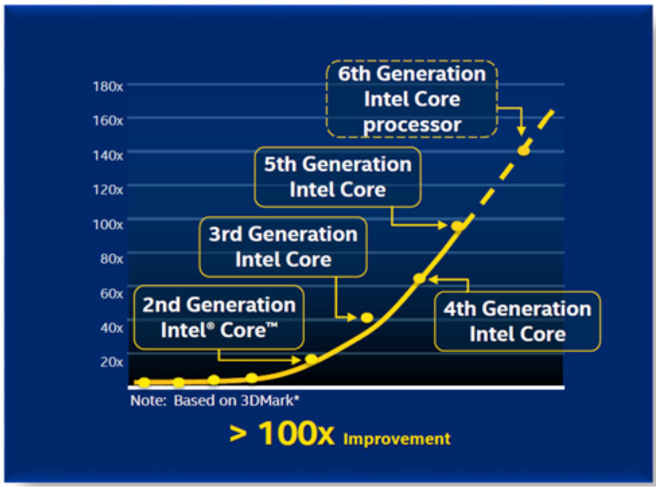
Using Intel® Processor Graphics for Compute with OpenCL™

Robert Ioffe, *Technical Consulting Engineer for OpenCL™*, Intel Corporation

Intel has been integrating processor graphics with its processors for the last five generations, starting with the introduction of the 2nd generation Intel® Core™ processor family in 2011. For the last four of those generations, Intel has provided OpenCL™ platform tools and drivers to access general-purpose compute capabilities of the Intel® Processor Graphics available in Intel® mobile, desktop, server, and embedded processors.

This article briefly outlines the 6th generation Intel Core processor's Gen9 processor graphics compute capabilities, explains how OpenCL takes advantage of those capabilities, and describes Intel's tools—such as Intel® SDK for OpenCL™ Applications and Intel® VTune™ Amplifier—that enable developers to create, debug, analyze, and optimize high-performance, heterogeneous applications, taking advantage of the compute capabilities of the whole platform. We'll also give a brief overview of the variety of resources available to aspiring processor graphics programmers on the OpenCL website.

Figure 1 shows the generational performance increases in Intel Processor Graphics.



1 Generational performance increases in Intel® Processor Graphics

BLOG HIGHLIGHTS

Understanding NUMA for 3D Isotropic Finite Difference (3DFD) Wave Equation Code

BY SUNNY G. >

This article demonstrates techniques that software developers can use to identify and fix NUMA-related performance issues in their applications using the latest Intel® software development tools. Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than the remote memory (memory local to another processor or memory shared between processors).

This article gives an overview of how the latest memory access feature of Intel® VTune™ Amplifier XE can be used to identify NUMA-related issues in the application. It extends the article published on Intel® Developer Zone (IDZ) related to development and performance comparison of Isotropic 3-dimensional finite difference application running on Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors. We also provide recommendations for source code modifications to achieve consistent high performance for your application in the NUMA environment.

Read more >

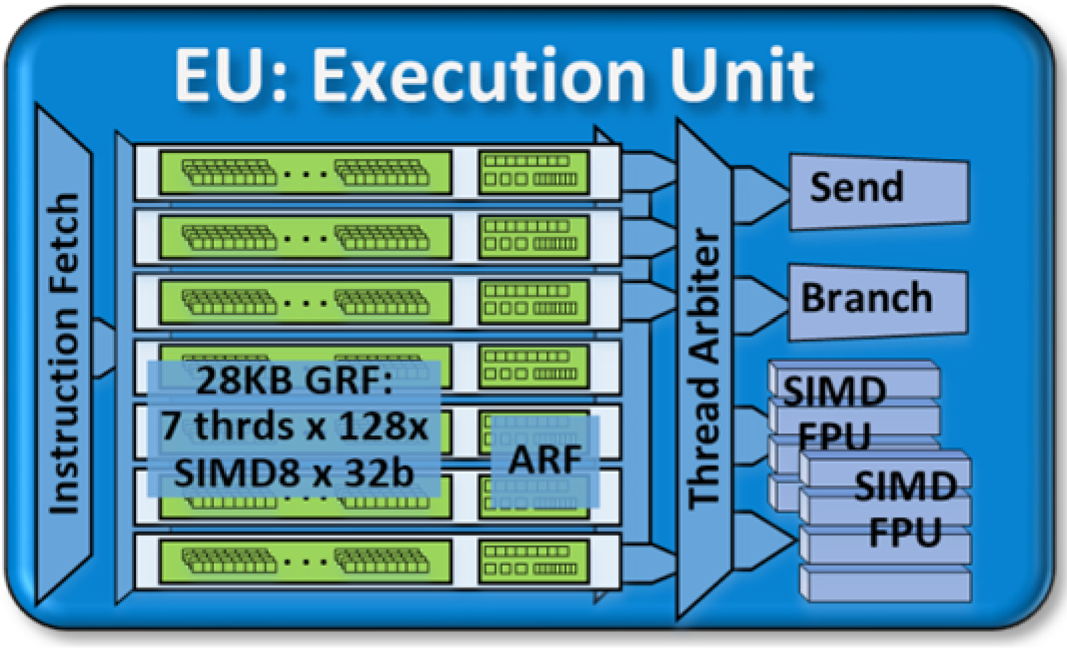
Compute Architecture of Intel® Processor Graphics Gen9

We'll start with a brief overview of the compute architecture of Intel® Processor Graphics Gen9, present in 6th generation Intel Core processors. (For an in-depth look, see Stephen Junkins' **white paper** with the same title as this section.)

The main building block of the Intel Processor Graphics Gen9 compute architecture is the execution unit (EU), which consists of seven hardware threads, each containing 128 32-byte-wide, general-purpose registers for a total of 4 KB of storage per hardware thread and 28 KB per EU **(Figure 2)**. Each EU has two **SIMD** FPUs capable of both floating-point and integer computations. Since each FPU is physically SIMD4 and capable of dispatching one multiply-add instruction per cycle, the peak theoretical performance of the processor graphics is calculated using the following formula:

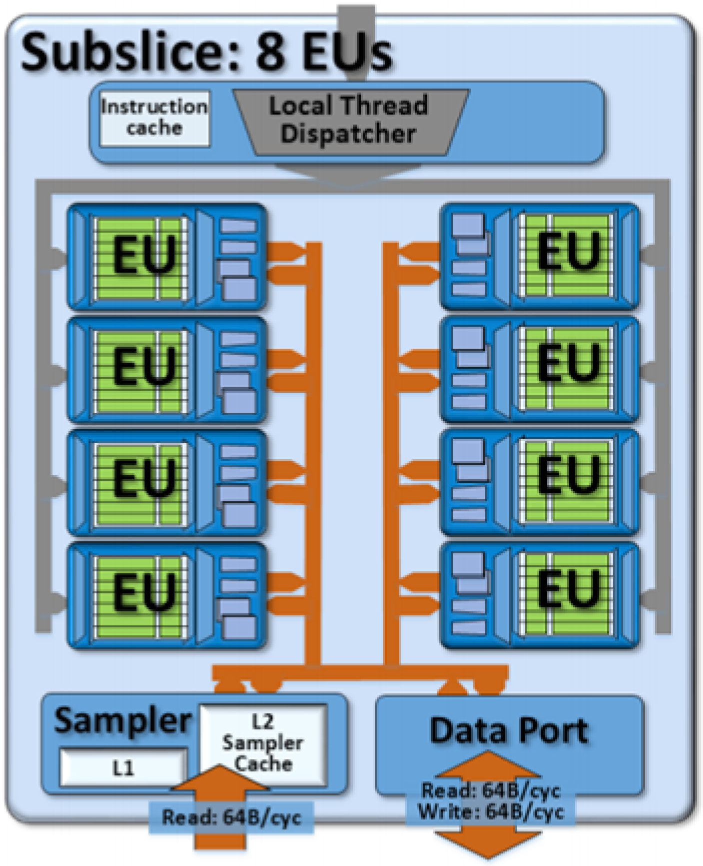
2 instructions × 4 (physical SIMD widths) × 2 FPUs per EU × number of EUs × processor graphics frequency

For example, Intel® Iris™ Pro Graphics P580 with 72 EUs is capable of running at maximum frequency of 1.1 GHz and is therefore capable of 1.267 peak single precision TFLOPs. Note: The FPU is physically SIMD4, but will appear as SIMD8, SIMD16, or SIMD32 to the programmer for convenience.



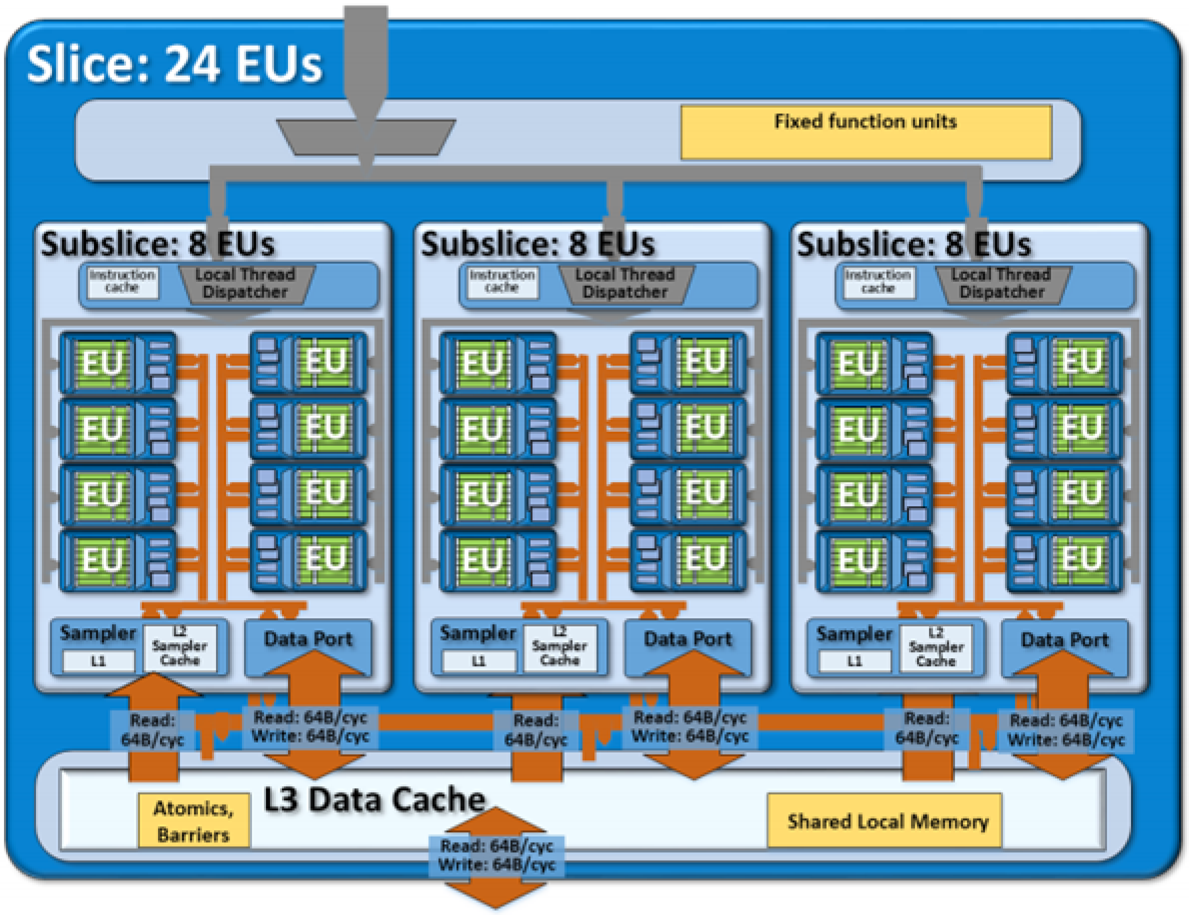
2 The execution unit (EU). Each Gen9 EU has seven threads. Each thread has 128 32-byte registers (GRF) and supporting architecture-specific register files (ARFs).

Eight EUs are grouped into a subslice **(Figure 3)**, which also has a local thread dispatcher unit with its own instruction cache, the sampler unit with its own L1 and L2 sampler caches for reading images, and a data port for reading and writing local and global memory and writing images.



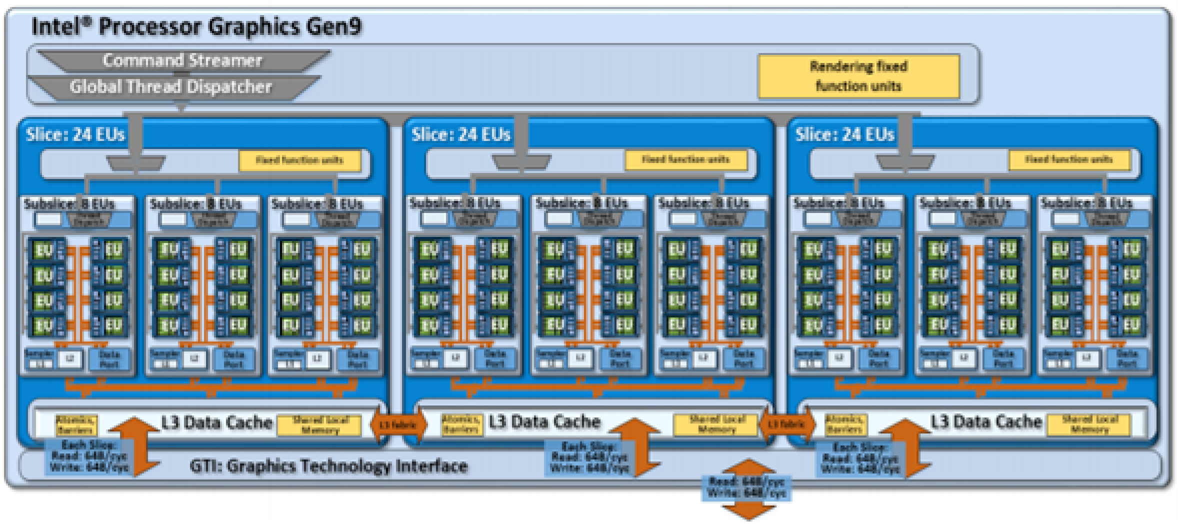
3 The Intel® Processor Graphics Gen9 subslice. Each subslice contains eight EUs and has its own sampler and data port.

Subslices are grouped into slices; typically three subslices form a slice **(Figure 4)**. Each slice houses a global thread dispatch unit; fixed-function hardware units such as a Video Motion Estimation (VME) unit and others; an L3 Data Cache of 512 KB per slice; shared local memory (SLM), which is highly banked, for sharing data among EU hardware threads within the same subslice of 64 KB per subslice; and barriers and atomics hardware for supporting barriers across groups of threads and for implementing atomic read-modify-write memory operations.



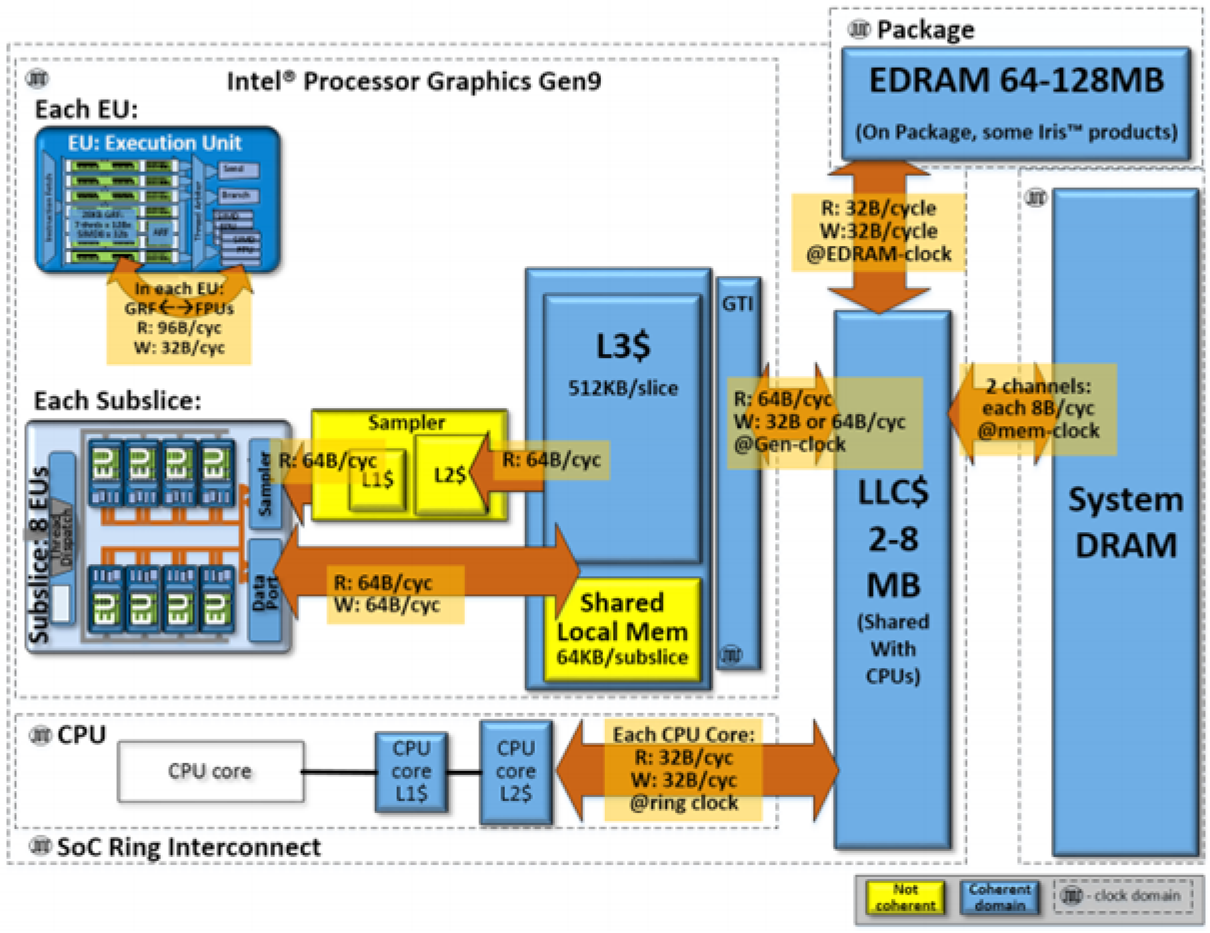
4 The Intel® Processor Graphics Gen9 slice contains three subslices for a total of 24 EUs. The slice adds an L3 data cache, shared local memory, atomics, barriers, and other fixed-function units.

One or more slices are grouped together for the creation of different product families. **Figure 5** shows one example of three slices grouped together.


Intel® Processor Graphics Gen9
Command Streamer
Global Thread Dispatcher
Rendering fixed function units
Slice: 24 EUs
Fixed function units
Subslice: 8 EUs
EU
L3 Data Cache
L3 fabric
GTI: Graphics Technology Interface
Each Slice: Read: 64B/cyc Write: 64B/cyc
Read: 64B/cyc Write: 64B/cyc

5 Intel® Iris™ Pro Graphics 580 has three slices with three subslices each, for a total of 72 EUs.

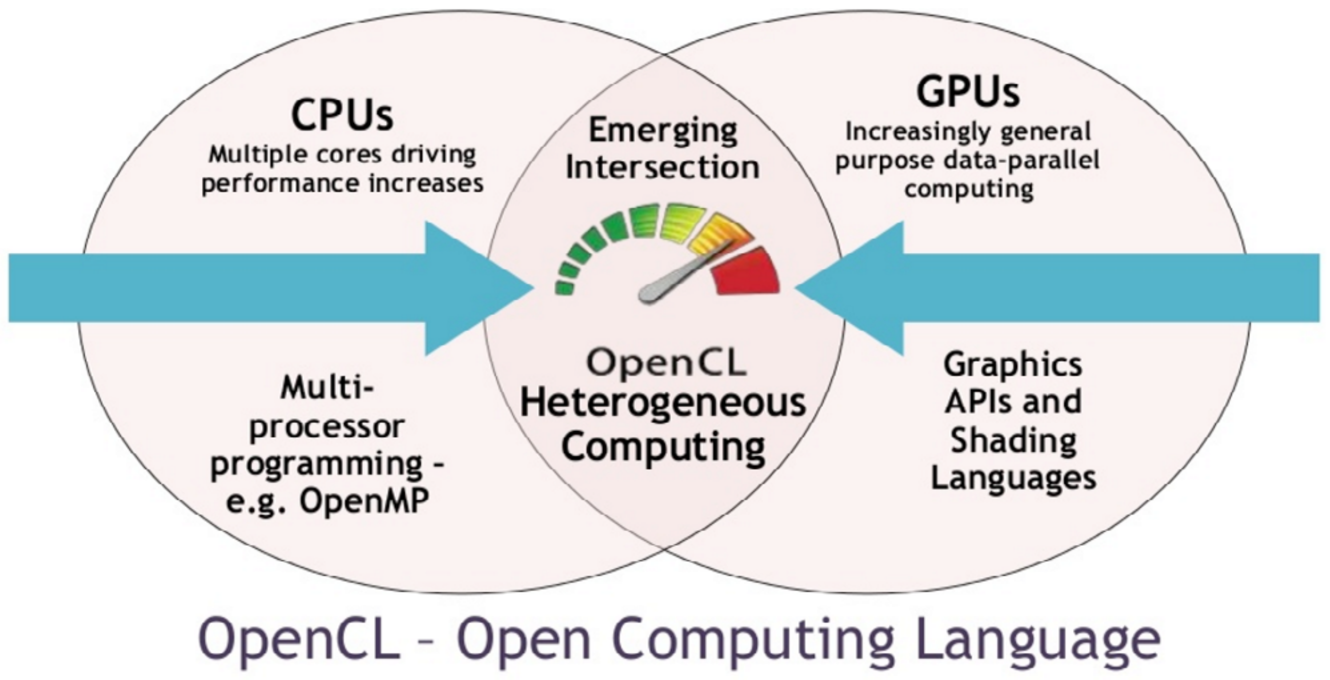
Intel® processors have a shared DRAM physical memory with the Intel Processor Graphics, which enables zero copy buffer transfers, meaning that no buffer copy is necessary. Note that not only System DRAM but the CPU L1 and L2 caches as well as the GPU's L3 Cache, a shared last level cache (LLC), and an optional EDRAM are coherent between the CPU and the Intel Processor Graphics **(Figure 6)**.



6 The SoC chip-level memory hierarchy and its theoretical peak bandwidths for the compute architecture of Intel® Processor Graphics Gen9

What Is OpenCL?

A great way to utilize Intel Processor Graphics for general purpose computation is to use OpenCL. OpenCL (Open Computing Language) is an open-standard, cross-platform, heterogeneous parallel programming that is supported for graphics on a wide range of Intel processors, from mobile devices all the way to server chips **(Figure 7)**. OpenCL is chaperoned by The Khronos Group, of which Intel is an active participant. Intel supports OpenCL on Windows*, Linux*, and Android*. Apple provides support for OpenCL in Apple desktop and laptop products.


CPUs
Multiple cores driving performance increases
Emerging Intersection
GPUs
Increasingly general purpose data-parallel computing
Multi-processor programming - e.g. OpenMP
OpenCL Heterogeneous Computing
Graphics APIs and Shading Languages
OpenCL - Open Computing Language

7 OpenCL™ is an open, royalty-free standard for portable, parallel programming of heterogeneous platforms.

OpenCL consists of a host-side API used to query and initialize available computing devices and a device-side programming language (typically OpenCL C, but soon OpenCL C++ as well) that expresses device-side computation. The bread and butter of OpenCL is the parallelization and **vectorization** of nested *for* loops of 1, 2, or 3 dimensions, called ND ranges. In addition, OpenCL provides access to fixed-function hardware such as samplers for dealing with image accesses or vendor-specific hardware such as video motion estimation (VME) units.

Typically, a programmer who wishes to parallelize and vectorize a *for* loop places the data that he or she wishes to operate on into buffers (or images), expresses the computation the body of the *for* loop as a kernel in OpenCL C, builds the kernel, binds parameters of the kernel to buffers, and then submits the bound kernel for execution into a command queue together with global and optionally local dimensions of the kernel. OpenCL runtime asynchronously submits the kernels in the command queue to the device. The kernel computation outputs reside in buffers or images and are available to the programmer after the kernel completes its execution.

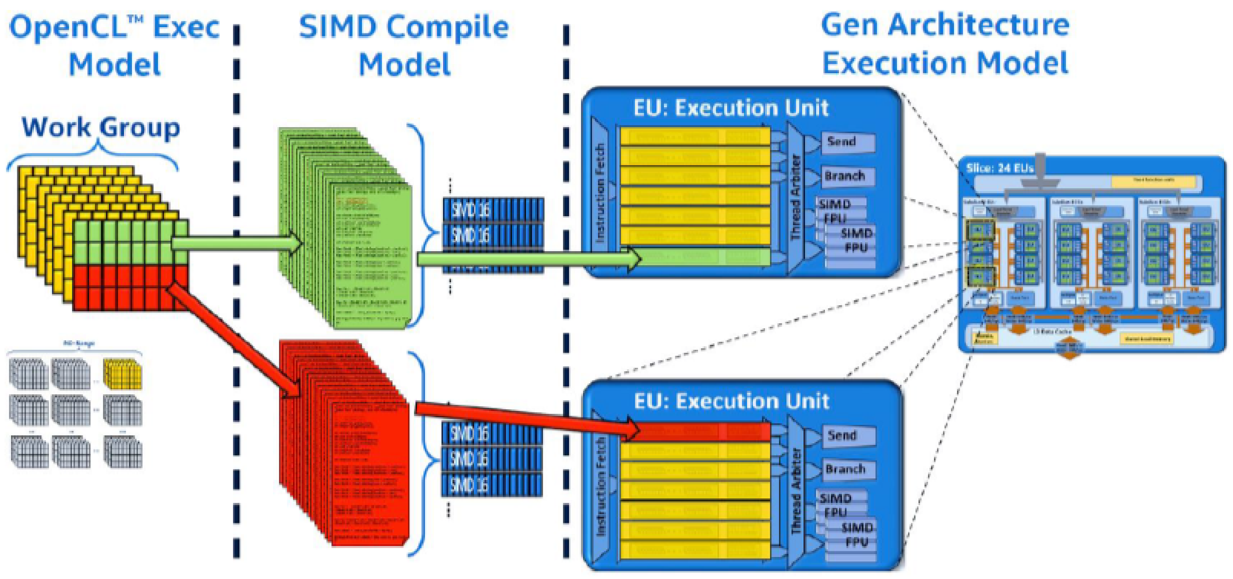
A single kernel execution is called a work item. A set of one or more consecutive work items, organized into a line or a rectangular 2D or 3D tile, form a workgroup. Work items within a group can synchronize using a barrier. A total work item of all workgroups for one kernel is called an NDRange.

OpenCL has a memory model that attempts to give explicit control over cache hierarchy to the programmer as opposed to hiding that hierarchy as is typical in regular C programming. Instead of one flat memory space, OpenCL has four distinct memory spaces: *private address space* (also called private memory) is a per work item memory; *local address space* (also called local memory) is a per workgroup memory, shared by all work items within a workgroup, but not available to the work items from other workgroups; *global address space* (also called global memory) is a device-side memory available to all work items of a kernel for both reading and writing; and *constant address space* (also called constant memory) is a device-side memory readable by all work items of a kernel but writable only from the host. OpenCL refers to the regular memory available on the CPU as a host memory. Memory management in OpenCL is very explicit: typically, it is the responsibility of the programmer to move the data from host to global to local to private memory and all the way back.

How OpenCL Maps to Intel Processor Graphics

Individual work items in OpenCL map to SIMD lanes of a hardware thread on Intel Processor Graphics. An OpenCL compiler may decide to compile a kernel SIMD32, SIMD16, or SIMD8 based on how much private memory the kernel requires. A compiler heuristic will choose a SIMD width that best maximizes the register footprint within a single hardware thread and avoids register spill/fill. Typically, short kernels that need less than 128 bytes of private memory will compile SIMD32, which means that 32 work items will fit on one hardware thread of the EU. Larger kernels that consume up to 256 bytes of private memory will typically compile SIMD16, or 16 work items per EU hardware thread. Very large kernels can consume up to 512 bytes of private memory and will be compiled with SIMD8, or eight work items per EU hardware thread. Private memory of work items maps to register space of an EU hardware thread and is therefore extremely fast.

Figure 8 shows an OpenCL execution model mapping to Intel Processor Graphics. Work items map to SIMD lanes of a thread—8, 16, or 32 work items per thread. A workgroup can span multiple threads and even multiple EUs.



8 OpenCL™ execution model mapping to Intel® Processor Graphics

Workgroups in OpenCL can fully map to just one hardware thread if they have 8, 16, or 32 work items, or they can span multiple EU threads within one EU, or even multiple EUs. The only limitation on the size of the workgroups is that if they are using local memory and therefore barriers, workgroups cannot span subslices.

Local memory in OpenCL maps to Shared Local Memory of Intel Processor Graphics, of which there are only 64 KB per subslice. L3 cache, LLC cache, optional EDRAM, and System DRAM map to OpenCL's global (and constant memory). Depending on the size of the OpenCL buffers and whether they fit into L3 cache (512 KB per slice), LLC cache (2–8 MB per slice), EDRAM (0, 64–128 MB), and System DRAM, the programmer could expect different bandwidths (see the memory diagram in **Figure 8**).

Note that since Shared Local Memory and L3 cache reside on the same chip and differ only in banking structure (16 banks for SLM from which the programmer can fetch four bytes from each bank, versus 64-byte cacheline fetches from L3 cache), only certain access patterns benefit local memory (in general, SLM and L3 cache have comparable latencies). For more details, see Intel's **OpenCL™ Developer Guide for Intel® Processor Graphics**.

When using discrete GPUs for computational purposes, the data typically needs to move from the CPU to the device and back. On Intel Processor Graphics, properly allocated buffers, aligned on a 4096-byte page boundary and sized in a multiple of cache line size (64 bytes), will be placed in a Shared Physical Memory and therefore will not be copied (zero copy). This feature is a very important optimization for OpenCL code running on Intel Processor Graphics.

Typically, OpenCL images are tiled in device memory, which always require a copy to the device, but support for the `cl_khr_image2d_from_buffer` extension in OpenCL 1.2, which became a core feature in OpenCL 2.0, allows the conversion of a properly allocated buffer to an image without requiring a copy.

There is support for sharing with graphics and media APIs, such as OpenGL* and **DirectX***, which enable us to avoid copying between the surfaces of these APIs and OpenCL buffers and allow postprocessing of rendered images with OpenCL and application of OpenCL filters to videos during decoding, transcoding, or encoding.

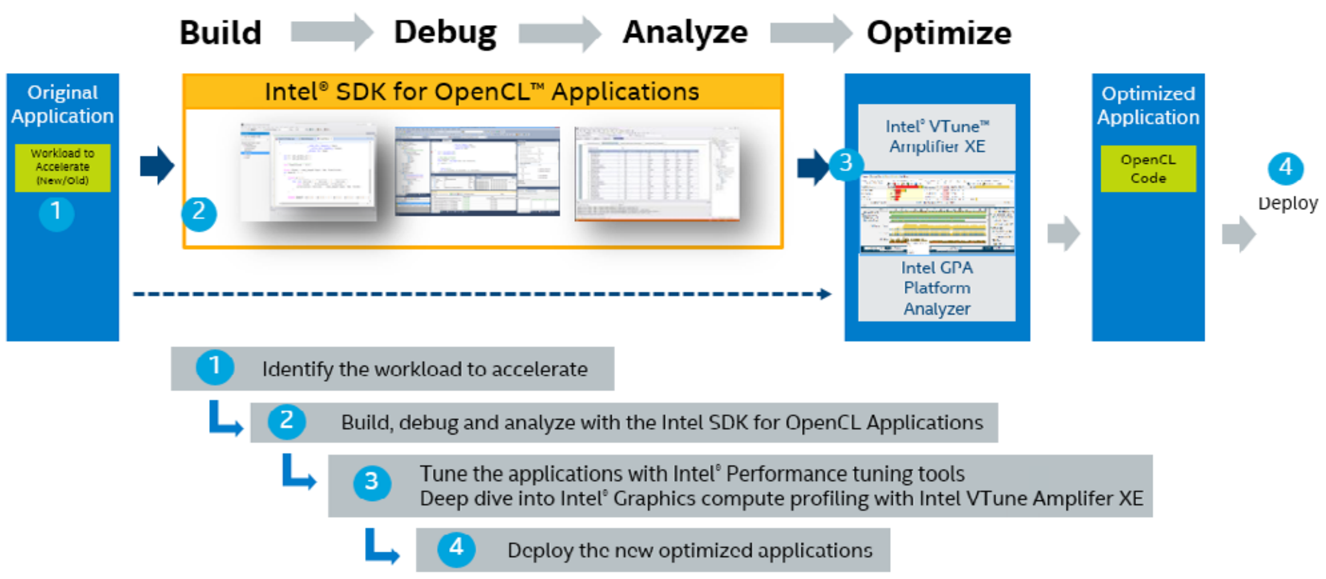
The SDK also provides a wizard-like guided performance profiling tool with hints and drill-down analysis.

There is support for a number of fixed-function extensions, such as **Video Motion Estimation** (VME) and advanced **VME**, which enables programmers to develop motion blur, object detection, and image stabilization algorithms with faster performance and lower power than purely CPU-based implementations.

OpenCL 1.2 is supported on 3rd through 6th generation Intel Core processors with Intel Processor Graphics. Additionally, OpenCL code can be run purely on the CPU. OpenCL 2.0, which introduced such powerful features as Shared Virtual Memory (SVM) that enables programmers to share the pointer containing data structures between the CPU and GPU, kernel self-enqueue that enables kernels enqueue other kernels directly on the device, generic address space that allows programmers to omit address spaces in function arguments, and many other powerful features, is supported on 5th and 6th generation Intel Core Processors with Intel Processor Graphics.

Intel Software Development Tools for OpenCL

Intel provides a variety of tools and samples that enables programmers to take full advantage of Intel Processor Graphics for their computational needs. The premier tool for OpenCL development is Intel SDK for OpenCL Applications that enables programmers to create, build, debug, analyze, and optimize their OpenCL applications within a single IDE **(Figure 9)**. Intel VTune Amplifier XE has a full set of OpenCL optimization capabilities. Intel® GPA Platform Analyzer supports basic OpenCL optimization as well.


Build
Debug
Analyze
Optimize
Original Application
Workload to Accelerate (New/Old)
1
Intel® SDK for OpenCL™ Applications
2
3
Intel® VTune™ Amplifier XE
Intel GPA Platform Analyzer
Optimized Application
OpenCL Code
4
Deploy
1 Identify the workload to accelerate
2 Build, debug and analyze with the Intel SDK for OpenCL Applications
3 Tune the applications with Intel® Performance tuning tools
Deep dive into Intel® Graphics compute profiling with Intel VTune Amplifer XE
4 Deploy the new optimized applications

9 For best results, use both Intel® SDK for OpenCL™ Applications and Intel® VTune™ Amplifier XE to create, build, debug, analyze, and optimize OpenCL™ applications targeting Intel® Processor Graphics.

OpenCL Drivers: Windows, Linux, Android, and OS X*

OpenCL support comes as part of the Intel® graphics driver on Windows. Even though Intel does not support the OpenCL driver on OS X*, Apple provides such a driver capable of running OpenCL code on Macs* with Intel processors that have Intel Processor Graphics. Ensure having the latest OpenCL driver on Windows by updating the graphics driver frequently to keep up-to-date with the latest optimization, features, and bug fixes. On OS X, follow Apple software update procedures. On Linux, the OpenCL driver is available as part of Intel® Media Server Studio or stand-alone drivers. On Android, the OpenCL driver is part of Intel SDK for OpenCL Applications. Android's OpenCL driver for the GPU is shipped with Android KitKat* devices based on Intel processors. For a full supported configurations matrix, please see the **Intel SDK for OpenCL Applications website**.

Intel SDK for OpenCL Applications

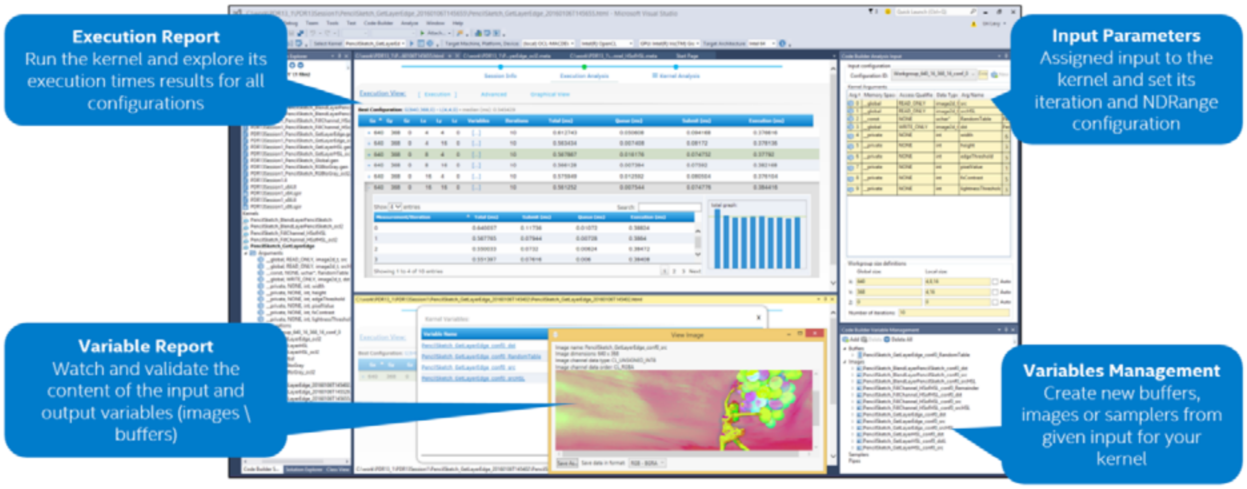
A good tool to start OpenCL development is Intel SDK for OpenCL Applications. The SDK provides a set of OpenCL wizards that enable quick creation of a full OpenCL application. Start exploration of OpenCL by developing kernels using the OpenCL™ Kernel Development tool. Start with the creation of a kernel development session using an empty file or an existing OpenCL file, use familiar code highlighting and code completion to assist writing or modifying a kernel, and build a kernel for either CPU or GPU devices. Then create buffers, images, and samplers; bind them to kernel arguments; and run the kernel with the specified global and local sizes for enough iterations to collect accurate kernel timing statistics. All this can be done directly from an easy-to-use GUI that is integrated into Microsoft Visual Studio* or Eclipse*. This also generates a number of artifacts such as SPIR portable binaries and GEN Assembly files to help with a deeper understanding of the kernels if wanted.

It is easy to find the best local size(s) for a kernel by running a design of experiments that goes through different kernel sizes. Another powerful way to explore and optimize kernels in an existing OpenCL application is to generate a kernel development session by running that application through a session generator, which captures all kernels and their parameters during one application pass.

There is a powerful way to run a deep performance analysis of kernels that will show the latency of various instructions and GPU occupancy numbers for a kernel. These metrics will help better optimize kernels.

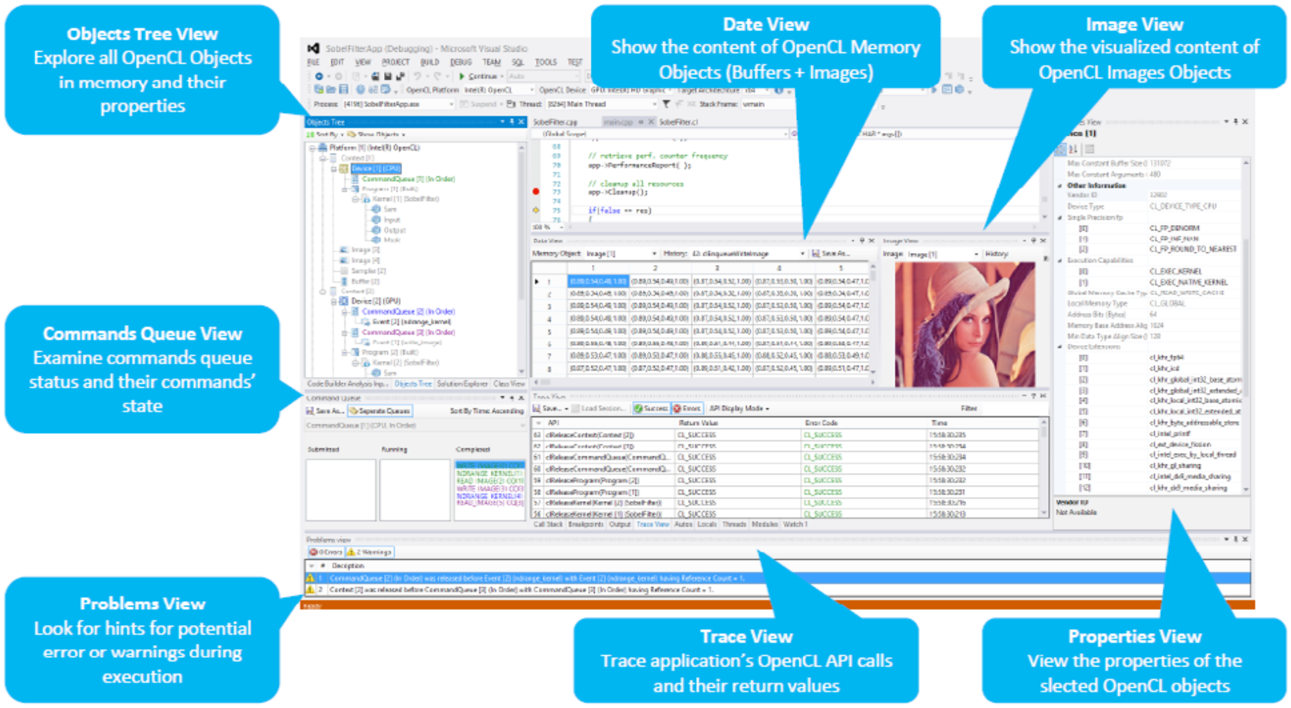
In addition, there is a capability to query platform information of local and remote (useful for developing kernels for an Android tablet or phone) devices right from an IDE. It reports supported device extensions, supported OpenCL versions, and image formats, among other things.

Another great feature is the OpenCL debugging capabilities for both the CPU and GPU that enable debugging a kernel one work item at a time (on a CPU) or one thread at a time (on a GPU, though it is necessary to separate host and target systems). It also provides all the conveniences of the modern debugger enhanced for the specifics of OpenCL, such as the ability to view the content of vector variables with types like float4 or uchar16 **(Figure 10)**.


Execution Report
Run the kernel and explore its execution times results for all configurations
Input Parameters
Assigned input to the kernel and set its iteration and NDRange configuration
Variable Report
Watch and validate the content of the input and output variables (images \ buffers)
Variables Management
Create new buffers, images or samplers from given input for your kernel

10 Execution of stand-alone kernels

The SDK also provides host-level debugger extension, which enables monitoring to understand the OpenCL environment of an application throughout execution, including API calls tracing, images and memory objects view, queues status views, and more.


Objects Tree View
Explore all OpenCL Objects in memory and their properties
Date View
Show the content of OpenCL Memory Objects (Buffers + Images)
Image View
Show the visualized content of OpenCL Images Objects
Commands Queue View
Examine commands queue status and their commands' state
Problems View
Look for hints for potential error or warnings during execution
Trace View
Trace application's OpenCL API calls and their return values
Properties View
View the properties of the slected OpenCL objects

11 OpenCL™ Code Builder is part of Intel® SDK for OpenCL™ Applications.

OpenCL Code Builder **(Figure 11)** integrates into Microsoft Visual Studio and provides powerful debugging capabilities for OpenCL code.

And finally, the SDK also provides a wizard-like guided performance profiling tool with hints and drill-down analysis, which allows collecting performance data from both the host side and the device side. The tool helps perform drill-down analysis from the host side by finding the bottlenecks in applications and identifying the most expensive kernels and API calls to the kernel level by performing deep-level analysis on the kernel latency and occupancy.

There is an extensive set of OpenCL 1.2 and OpenCL 2.0 **samples and tutorials** for Windows, Linux, and Android, ranging from very basic introductory samples all the way to advanced samples showcasing new features of OpenCL and Intel graphics.

A Quick Primer on Using the SDK

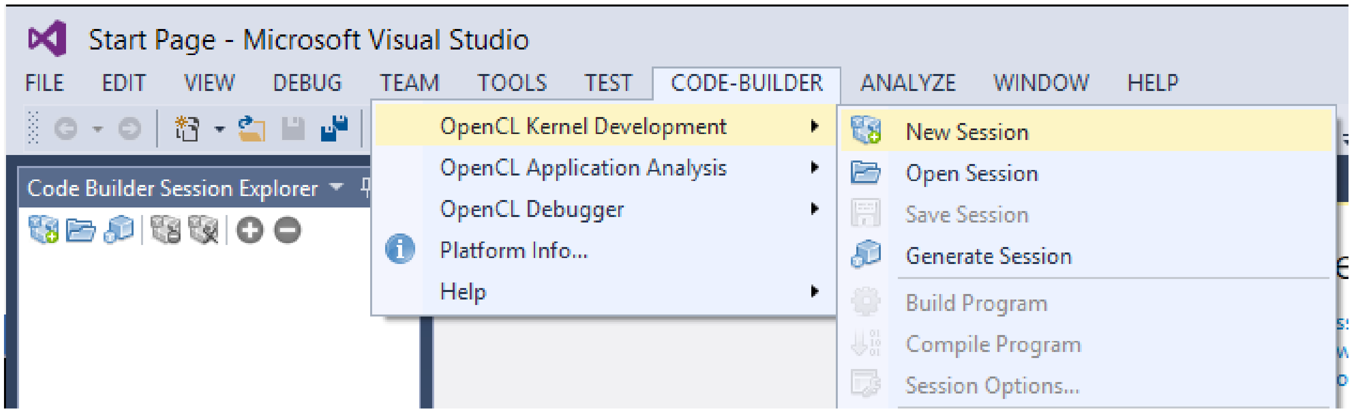
One of the key capabilities of the Intel SDK for OpenCL Applications is the ability to jump into OpenCL kernel development without developing the host side of the application. Let's say you want to develop an OpenCL version of the **Sierpiński carpet**. The iterative version of the formula, which decides how to color a particular pixel with coordinates (x, y), is given by:

```
#define BLACK 0
#define WHITE 255

uint sierpinski_color(int x, int y) {
 while ( x > 0 || y > 0 ) {
 if( x % 3 == 1 && y % 3 == 1)
 return BLACK;
 x /= 3;
 y /= 3;
 }
 return WHITE;
}
```

Wrap this function inside an OpenCL kernel, so once you figure out the pixel color, you can write it out to an image:

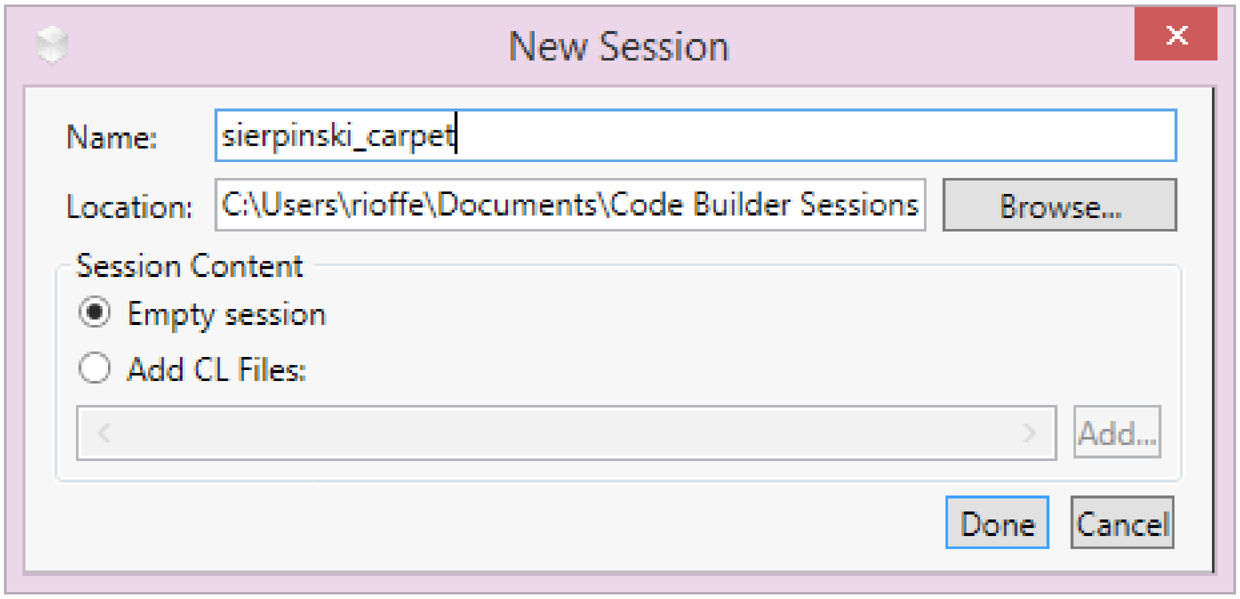
```
kernel void sierpinski(__write_only image2d_t out) {
   int x = get_global_id(0);
   int y = get_global_id(1);

   uint color = sierpinski_color(x, y);

   write_imageui(out, (int2)(x, y), (uint4)(color, color, color, 0));
}
```

Then, start Microsoft Visual Studio and under the CODE-BUILDER menu, select **OpenCL Kernel Development**, and click on **New Session (Figure 12)**.


Start Page - Microsoft Visual Studio
FILE EDIT VIEW DEBUG TEAM TOOLS TEST CODE-BUILDER ANALYZE WINDOW HELP
Code Builder Session Explorer
OpenCL Kernel Development
OpenCL Application Analysis
OpenCL Debugger
Platform Info...
Help
New Session
Open Session
Save Session
Generate Session
Build Program
Compile Program
Session Options...

12 OpenCL™ kernel development

In the **New Session** dialog box that pops up, enter **sierpinski_carpet** as a name and click the **Done** button **(Figure 13)**.

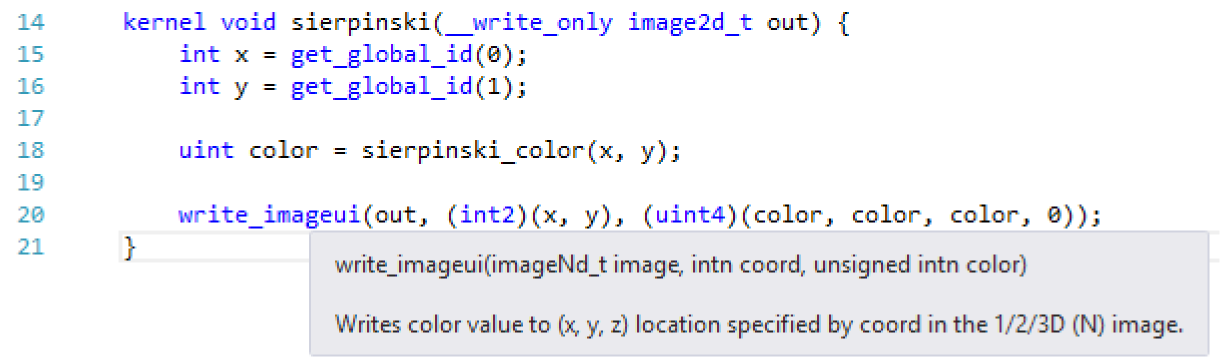


Name: sierpinski_carpet

Location: C:\Users\rioffe\Documents\Code Builder Sessions

13 Create a New Session

Double-click on the program.cl in the **Code Builder Session Explorer** and enter the OpenCL program in **Figure 13** (note that we enable syntax highlighting for OpenCL keywords, built-in vector types, and built-in functions, as well as pop-up help hints of the built-in functions).

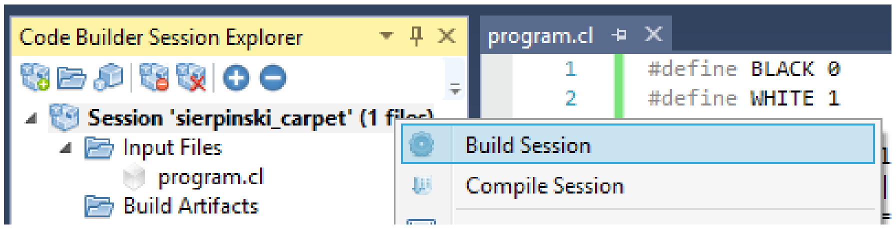


```
kernel void sierpinski(__write_only image2d_t out) {
    int x = get_global_id(0);
    int y = get_global_id(1);

    uint color = sierpinski_color(x, y);

    write_imageui(out, (int2)(x, y), (uint4)(color, color, color, 0));
}
```

write_imageui(imageNd_t image, intn coord, unsigned intn color)

Writes color value to (x, y, z) location specified by coord in the 1/2/3D (N) image.

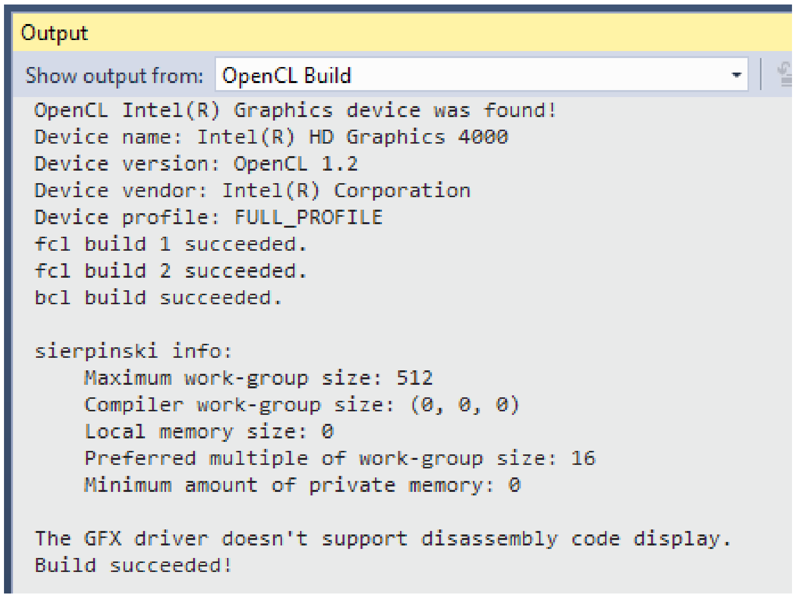
Save your OpenCL program, and then click on the **Session 'sierpinski_carpet'**, right-click, and in the pop-up window, select **Build Session (Figure 14)**.



```
#define BLACK 0
#define WHITE 1
```

14 Build session

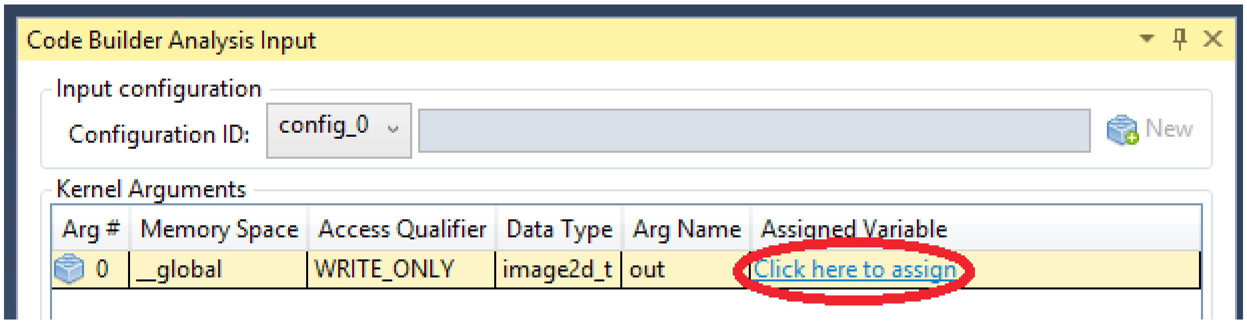
If the build was successful, you should see the following in the output window in **Figure 15**.



```
OpenCL Intel(R) Graphics device was found!
Device name: Intel(R) HD Graphics 4000
Device version: OpenCL 1.2
Device vendor: Intel(R) Corporation
Device profile: FULL_PROFILE
fcl build 1 succeeded.
fcl build 2 succeeded.
bcl build succeeded.

sierpinski info:
    Maximum work-group size: 512
    Compiler work-group size: (0, 0, 0)
    Local memory size: 0
    Preferred multiple of work-group size: 16
    Minimum amount of private memory: 0

The GFX driver doesn't support disassembly code display.
Build succeeded!
```

15 Successful build

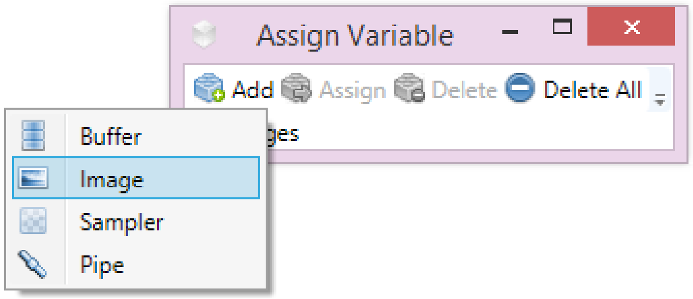
Now that we have built our kernel, we need to create the output image and specify a global size and the number of iterations for our kernel, so we can collect kernel run statistics. In the Code Builder Analysis Input, click on the **Click here to assign** link **(Figure 16)**.



Arg #	Memory Space	Access Qualifier	Data Type	Arg Name	Assigned Variable
0	__global	WRITE_ONLY	image2d_t	out	Click here to assign

16 Collect kernel run statistics

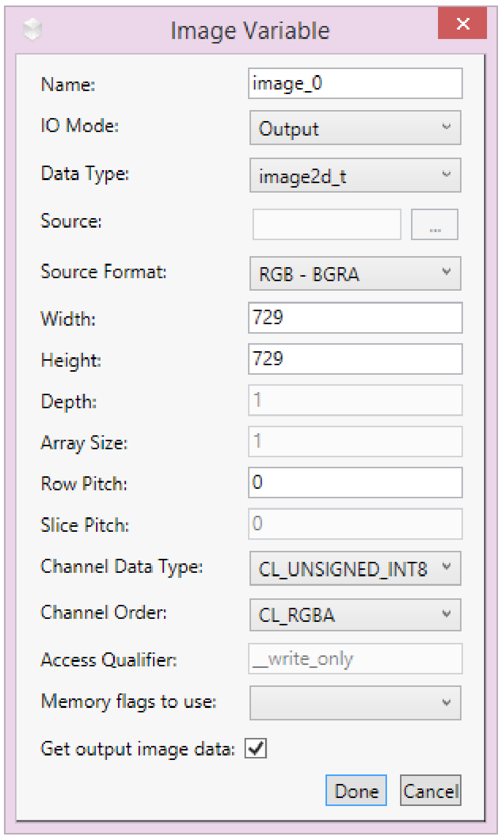
Click the **Add** button in the **Assign Variable** pop-up window and select **Image** from the menu **(Figure 17)**.



17 Select Image

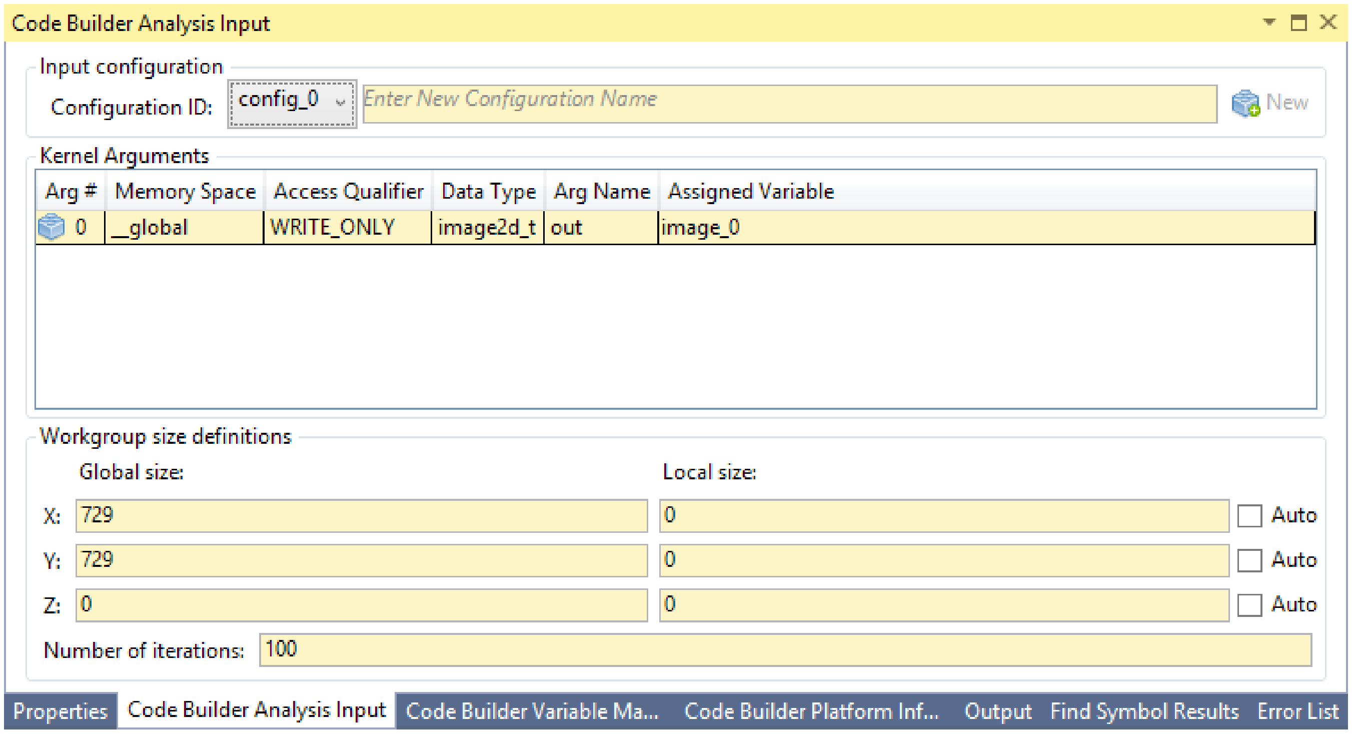
In the **Image Variable** dialog box, set **IO Mode as Output**, set **Width** and **Height** to **729**, set the **Channel Data Type** to **CL_UNSIGNED_INT8** and **Channel Order** to **CL_RGBA**, and click **Done (Figure 18)**.

Double-click **image_0 entry** to assign it to the out argument of the sierpinski kernel. Set **Global** size in the **Workgroup** size definitions section of the **Code Builder Analysis Input** tab to 729 for both X and Y and set the **Number of iterations** to 100 **(Figure 19)**.



Name:	image_0
IO Mode:	Output
Data Type:	image2d_t
Source:	
Source Format:	RGB - BGRA
Width:	729
Height:	729
Depth:	1
Array Size:	1
Row Pitch:	0
Slice Pitch:	0
Channel Data Type:	CL_UNSIGNED_INT8
Channel Order:	CL_RGBA
Access Qualifier:	_write_only
Memory flags to use:	
Get output image data:	✓

18 Image Variable


Code Builder Analysis Input

Input configuration

Configuration ID: config_0 | Enter New Configuration Name | New

Kernel Arguments

Arg #	Memory Space	Access Qualifier	Data Type	Arg Name	Assigned Variable
0	__global	WRITE_ONLY	image2d_t	out	image_0

Workgroup size definitions

	Global size:	Local size:	
X:	729	0	☐ Auto
Y:	729	0	☐ Auto
Z:	0	0	☐ Auto

Number of iterations: 100

Properties
Code Builder Analysis Input
Code Builder Variable Ma...
Code Builder Platform Inf...
Output
Find Symbol Results
Error List

19 Code Builder Analysis Input

BLOG HIGHLIGHTS

Unleash Parallel Performance of Python* Programs

BY ANTON MALAKHOV >

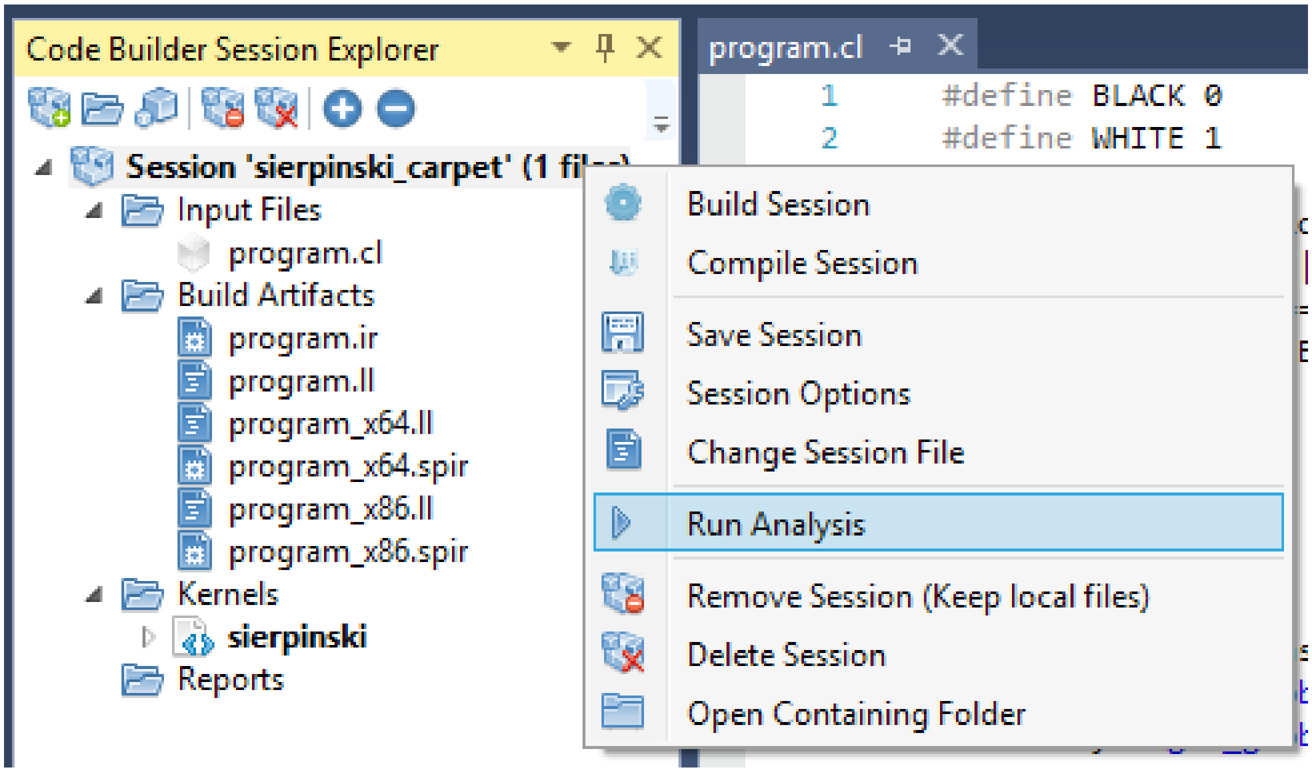
In the Beta release of Intel® Distribution for Python*, I am proud to introduce something new and unusual for the Python world. It is an experimental module which unlocks additional performance for multithreaded Python programs by enabling threading composability between two or more thread-enabled libraries.

Threading composability can accelerate programs by avoiding inefficient threads allocation (called oversubscription) when there are more software threads than available hardware resources.

The biggest improvement is achieved when a task pool like the ThreadPool from standard library or libraries like Dask or Joblib (used in multithreading mode) execute tasks calling compute-intensive functions of Numpy/Scipy/PyDAAL, which in turn are parallelized using Intel® MKL or/and Intel® Threading Building Blocks.

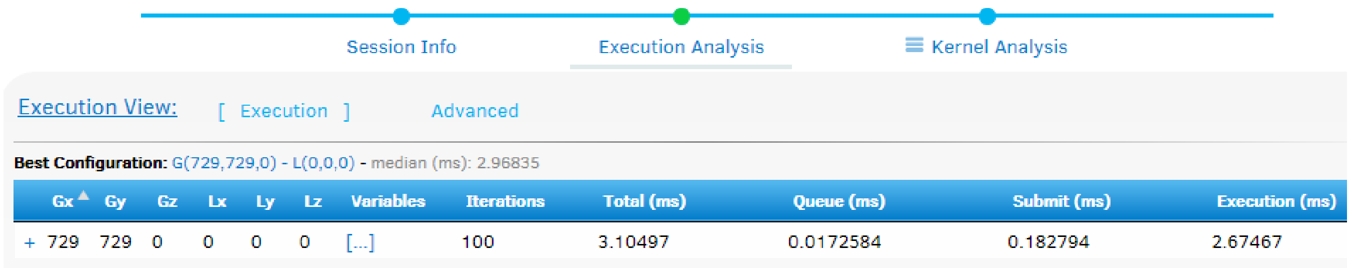
Read more >

Now, you are ready to run the kernel. Right-click on the **Session 'sierpinski_carpet'** in the **Code Builder Session Explorer** tab and select **Run Analysis (Figure 20)**.



20 Code Builder Session Explorer

You should see the Execution Analysis view once all 100 iterations of the kernel finish execution **(Figure 21)**.

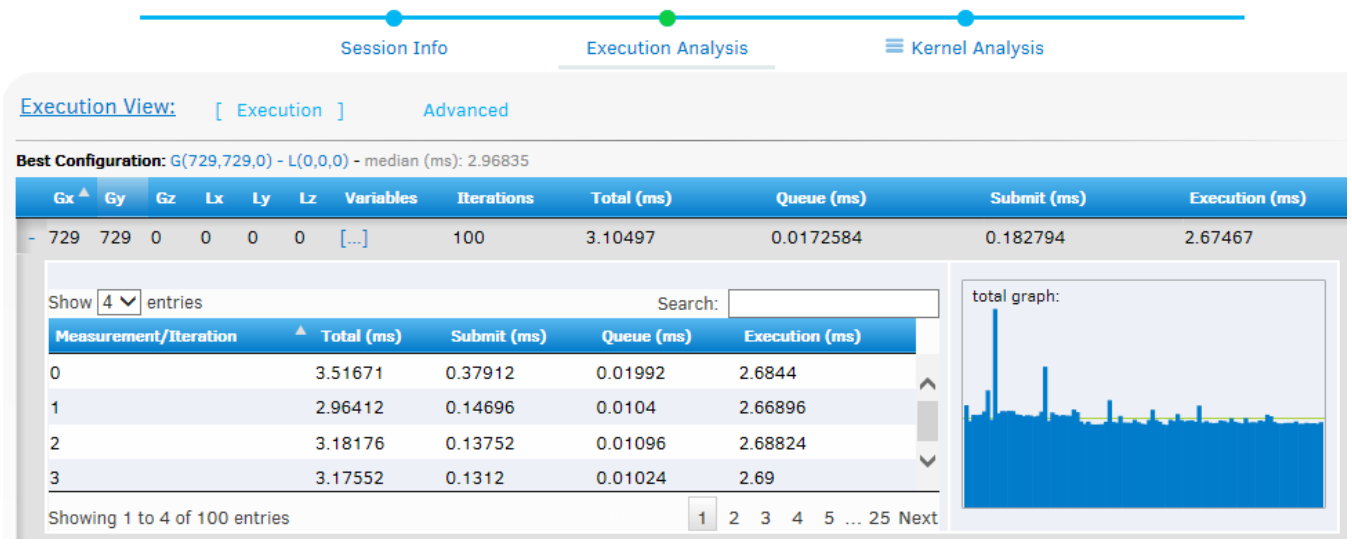


Best Configuration: G(729,729,0) - L(0,0,0) - median (ms): 2.96835

Gx	Gy	Gz	Lx	Ly	Lz	Variables	Iterations	Total (ms)	Queue (ms)	Submit (ms)	Execution (ms)
+ 729	729	0	0	0	0	[...]	100	3.10497	0.0172584	0.182794	2.67467

21 Execution Analysis view

Click on the **+** sign on the left side to view kernel execution statistics and a graph of individual kernel execution times **(Figure 22)**.



Best Configuration: G(729,729,0) - L(0,0,0) - median (ms): 2.96835

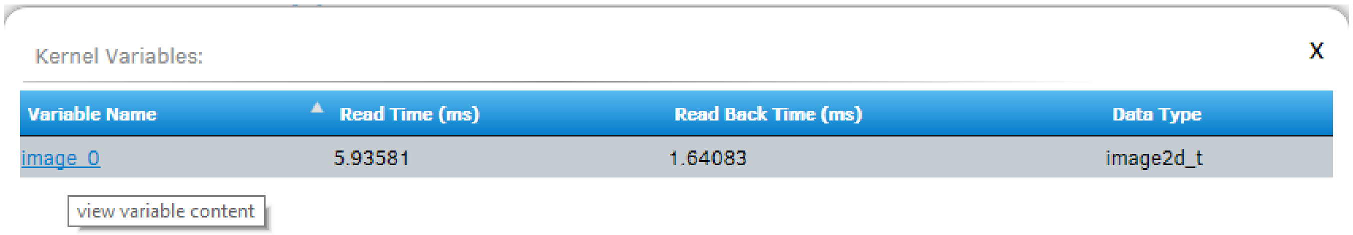
	Gx	Gy	Gz	Lx	Ly	Lz	Variables	Iterations	Total (ms)	Queue (ms)	Submit (ms)	Execution (ms)
-	729	729	0	0	0	0	[...]	100	3.10497	0.0172584	0.182794	2.67467

Measurement/Iteration	Total (ms)	Submit (ms)	Queue (ms)	Execution (ms)
0	3.51671	0.37912	0.01992	2.6844
1	2.96412	0.14696	0.0104	2.66896
2	3.18176	0.13752	0.01096	2.68824
3	3.17552	0.1312	0.01024	2.69

Showing 1 to 4 of 100 entries 1 2 3 4 5 … 25 Next

22 Execution View

Click on the **[...]** in the **Variables** column and, in the **Kernel Variables** pop-up window, click on **image_0** to view the results of running the sierpinski kernel **(Figure 23)**.


Kernel Variables:

Variable Name	Read Time (ms)	Read Back Time (ms)	Data Type
image_0	5.93581	1.64083	image2d_t

23 Kernel Variables

You should see the **View Image** window in **Figure 24**.

You can save this image in a variety of formats for future reference. In the upcoming version of the Intel SDK for OpenCL Applications, you will be able to generate a host-side code from the started kernel and have a full stand-alone OpenCL application. We have many more powerful features for both OpenCL kernel and OpenCL application development, analysis, and debugging. We encourage you to read our Development Guide and study our OpenCL samples (see the References at the end of this article).

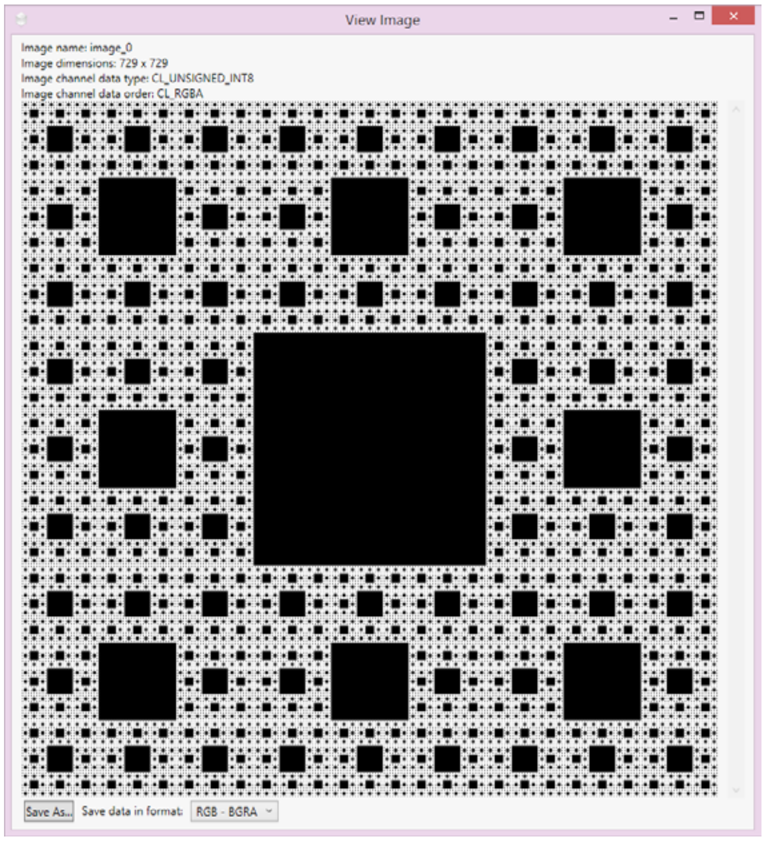


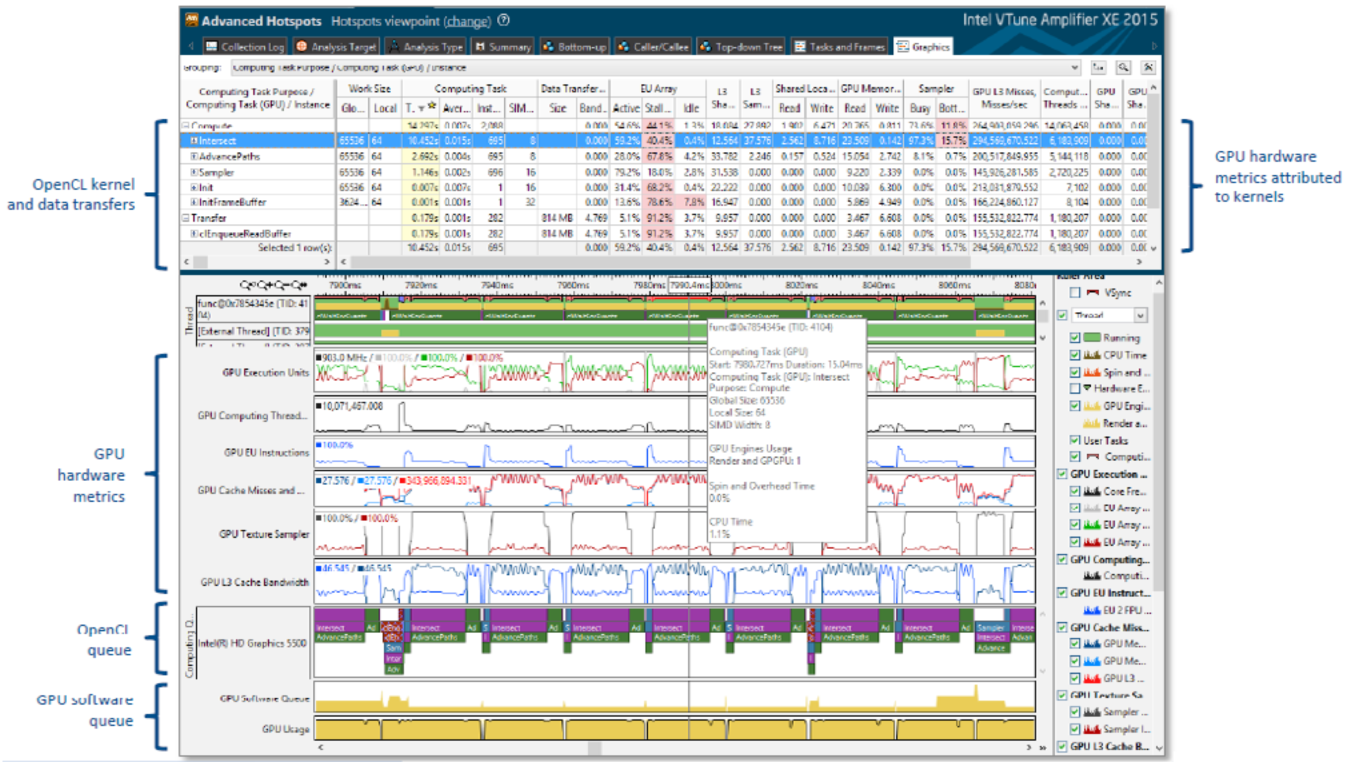
Image name: image_0
Image dimensions: 729 x 729
Image channel data type: CL_UNSIGNED_INT8
Image channel data order: CL_RGBA

24 View Image window

Intel VTune Amplifier: OpenCL Capabilities

Intel VTune Amplifier gives you the ability to profile applications that use Intel Processor Graphics for image rendering, video processing, and general-purpose computations **(Figure 25)**. For OpenCL applications, Intel VTune Amplifier gives the overall summary of a graphics device, the hottest OpenCL kernels, and host-side OpenCL calls as the best candidates for investing time in optimizing, performance and occupancy information for each kernel, the utilization of data buses and samplers, and other GPU hardware metrics. Intel VTune Amplifier also has a customizable timeline view with the ability to zoom in and out, to show different aspects of kernel activity.

In the 2017 version of Intel VTune Amplifier (available as a beta until release in late 2016), the ability to profile a kernel at the assembly level has been added. VTune nicely complements the tools available in Intel SDK for OpenCL Applications for a more in-depth view of kernel performance.



Computing Task Purpose / Computing Task (GPU) / Instance	Work Size		Computing Task				Data Transfer...		EU Array			L3 She...	L3 Sam...	Shared Loca...		GPU Memor...		Sampler		GPU L3 Misses, Misses/sec	Comput... Threads ...	GPU She...	GPU She...
	Glo...	Local	T...	Aver...	Inst...	SIM...	Size	Band...	Active	Stall...	Idle			Read	Write	Read	Write	Busy	Bott...				
⊟ Compute			14.297s	0.007s	2,088			0.000	54.6%	44.1%	1.3%	18.094	27.892	1.902	6.471	20.765	0.811	73.6%	11.8%	264,903,059,296	14,063,458	0.000	0.0
⊞ Intersect	65536	64	10.452s	0.015s	695	8		0.000	59.2%	40.4%	0.4%	12.564	37.576	2.562	8.716	23.509	0.142	97.3%	15.7%	294,569,670.522	6,183,909	0.000	0.0
⊞ AdvancePaths	65536	64	2.692s	0.004s	695	8		0.000	28.0%	67.8%	4.2%	33.782	2.246	0.157	0.524	15.054	2.742	8.1%	0.7%	200,517,849.955	5,144,118	0.000	0.0
⊞ Sampler	65536	64	1.146s	0.002s	696	16		0.000	79.2%	18.0%	2.8%	31.538	0.000	0.000	0.000	9.220	2.339	0.0%	0.0%	145,926,281.585	2,720,225	0.000	0.0
⊞ Init	65536	64	0.007s	0.007s	1	16		0.000	31.4%	68.2%	0.4%	22.222	0.000	0.000	0.000	10.039	6.300	0.0%	0.0%	213,031,879.552	7,102	0.000	0.0
⊞ InitFrameBuffer	3624...	64	0.001s	0.001s	1	32		0.000	13.6%	78.6%	7.8%	16.947	0.000	0.000	0.000	5.869	4.949	0.0%	0.0%	166,224,860.127	8,104	0.000	0.0
⊟ Transfer			0.179s	0.001s	282		814 MB	4.769	5.1%	91.2%	3.7%	9.957	0.000	0.000	0.000	3.467	6.608	0.0%	0.0%	155,532,822.774	1,180,207	0.000	0.0
⊞ clEnqueueReadBuffer			0.179s	0.001s	282		814 MB	4.769	5.1%	91.2%	3.7%	9.957	0.000	0.000	0.000	3.467	6.608	0.0%	0.0%	155,532,822.774	1,180,207	0.000	0.0
Selected 1 row(s):			10.452s	0.015s	695			0.000	59.2%	40.4%	0.4%	12.564	37.576	2.562	8.716	23.509	0.142	97.3%	15.7%	294,569,670.522	6,183,909	0.000	0.0

func@0x7854345e (TID: 4104)

Computing Task (GPU)
Start: 7980.727ms Duration: 15.04ms
Computing Task (GPU): Intersect
Purpose: Compute
Global Size: 65536
Local Size: 64
SIMD Width: 8

GPU Engines Usage
Render and GPGPU: 1

Spin and Overhead Time
0.0%

CPU Time
1.1%

903.0 MHz / 100.0% / 100.0% / 100.0%
10,071,467.008
100.0%
27.576 / 27.576 / 343,966,094.331
100.0% / 100.0%
46.545 / 46.545

25 The Graphics tab of Intel® VTune™ Amplifier XE shows the performance of OpenCL™ kernels running on Intel® Processor Graphics in a timeline view.

Conclusion

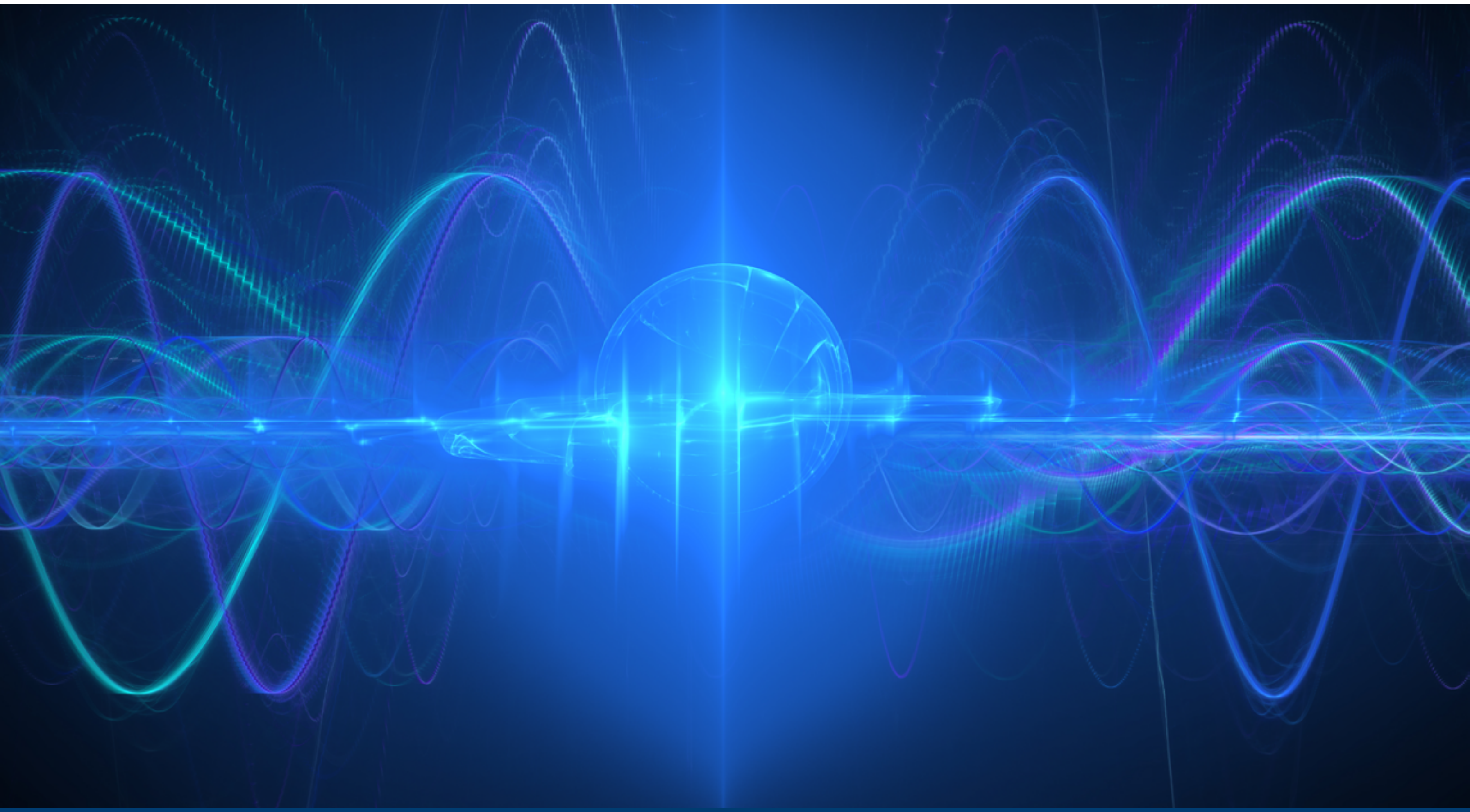
OpenCL is a powerful tool that allows programmers to exploit the full computational capabilities of Intel processors by utilizing CPU, GPU, fixed-function hardware and, in the future, FPGA parts of the chip. Intel offers OpenCL drivers, runtimes, and compilers—but in a whole suite of tools that allows the programmers to create, build, debug, analyze, and optimize their OpenCL applications for Intel Processor Graphics. Performance increases help fuel development of new and exciting capabilities in the areas of image and video processing, virtual and augmented reality, scientific computing and HPC, artificial intelligence, and robotics and machine learning (including deep learning).

References

1. **Intel® SDK for OpenCL™ Applications**
2. **Intel® OpenCL™ Forum**
3. **Intel® VTune™ Amplifier 2016**
4. **Intel® Graphics Performance Analyzers**
5. **Intel® Media Server Studio 2016**
6. **OpenCL™ Drivers and Runtimes for Intel® Architecture**
7. **Intel® Processor Graphics – Compute Architecture and Developer Guides**
8. **The Compute Architecture of Intel® Processor Graphics Gen9** by Stephen Junkins
9. **Introduction to Motion Estimation Extension for OpenCL™** by Maxim Shevtsov
10. **Introduction to Advance Motion Estimation Extension for OpenCL™** by Maxim Shevtsov and Dmitry Budnikov
11. **Intel® OpenCL™ Code Samples for Windows*, Linux*, Android***
12. **OpenCL™ Developer Guide for Intel® Processor Graphics**
13. **Developer Guide for Intel® SDK for OpenCL™ Applications**
14. **Latest Intel Drivers and Software**
15. **Using SPIR for Fun and Profit with Intel® OpenCL™ Code Builder** by Robert Ioffe
16. **Introduction to GEN Assembly** by Robert Ioffe
17. **Intel® VTune™ Amplifier XE: Getting Started with OpenCL™ Performance Analysis on Intel® HD Graphics** by Julia Fedorova

Acknowledgements

I would like to thank Stephen Junkins, Allen Hux, Jeff McAllister, Bethany Dalby, Uri Levy, Dan Petre, and James Reinders for reviewing this paper and providing valuable comments, edits, and suggestions. Most of the diagrams in this paper come from the excellent white paper **"The Compute Architecture of Intel® Processor Graphics Gen9"** by Stephen Junkins. It is worthwhile reading for anyone interested in a deeper understanding of the technology.



A RUNTIME-GENERATED FAST FOURIER TRANSFORM FOR INTEL® PROCESSOR GRAPHICS

Optimizing FFT without Increasing Complexity

Dan Petre, Adam T. Lake, and Allen Hux; *Graphics Software Engineers*; Intel Corporation

This article discusses techniques to optimize the fast Fourier transform (FFT) for Intel® Processor Graphics without using local memory or vendor extensions. The implementation uses a runtime code generator to support a wide variety of FFT dimensions. Performance depends on a deep understanding of how the OpenCL™ API maps to the underlying architecture.

When optimizing OpenCL applications, developers typically progress through a series of steps:

- Naïve implementation
- Attempt cross-work item collaboration via local memory
- Explore vendor-specific OpenCL extensions

There are downsides to these optimizations, however, including increased code complexity, which impairs extension and maintainability, and the need for a unique implementation for each architecture targeted by the application—a symptom for what is known as *reduced performance portability*.

What Is FFT?

FFT is a class of algorithms for the faster computation of the discrete Fourier transform (DFT), which itself is a way to compute the discrete time Fourier transform (DTFT). The DTFT is what we're really after, but the DTFT is an infinite series and can't be computed. While the DTFT requires an infinite number of input samples, the DFT requires a finite number:

$$X_k = \sum_{n=0}^{N} x_n e^{-j\frac{2\pi}{N}nk} \tag{1}$$

Where *k*=[0..N 1]. The DFT is a computationally expensive operation (O[n^2] complexity). This article focuses on the Cooley-Tukey (1965) radix-2 decomposition, which is a well-known and highly regular algorithm suitable for GPUs, and applies when N=2^n (the input signal has a length that is a power of two) and reduces the theoretical complexity to O(N $\log_2$N):

$$X_k = \sum_{n=0}^{N/2} x_{2n} e^{-j\frac{2\pi}{N/2}2nk} + \sum_{n=0}^{N/2} x_{2n+1} e^{-j\frac{2\pi}{N/2}(2n+1)k} \tag{2}$$

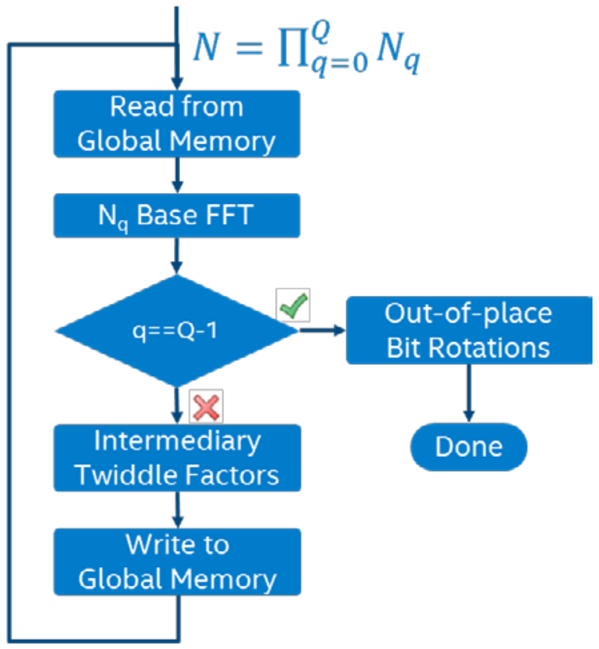
This is called radix-2 because the signal is broken in half at every stage of this recursive formula. There are at least two major considerations when implementing a fast DFT. The first is to improve the algorithm or, in other words, to reduce the total number of math operations necessary to perform the DFT as in the Cooley-Tukey decomposition shown previously. The second is to map the DFT algorithm efficiently to a physical processor in order to minimize the execution time.

genFFT Implementation

Our implementation differs from other general-purpose GPU implementations in several ways. It avoids local memory and barriers, the use of which can be a limiting factor on the total number of hardware threads active at any moment in time (thus reducing the memory latency hiding abilities of the machine). The FFT is broken down into smaller FFTs and, for each of these (called here "base FFTs"), the code generator produces efficient kernel code. Work items process unique sets of data independently from each other, reducing code complexity. Our FFT code generator can produce kernel code for:

- Any FFT length power of two
- Any FFT decomposition into base FFTs
- Any single instruction, multiple data (SIMD) size
- The size of the available register space
- Any global/local size configuration

Figure 1 shows a multikernel FFT execution flow.


$N = \prod_{q=0}^{Q} N_q$
Read from Global Memory
N_q Base FFT
q==Q-1
Out-of-place Bit Rotations
Done
Intermediary Twiddle Factors
Write to Global Memory

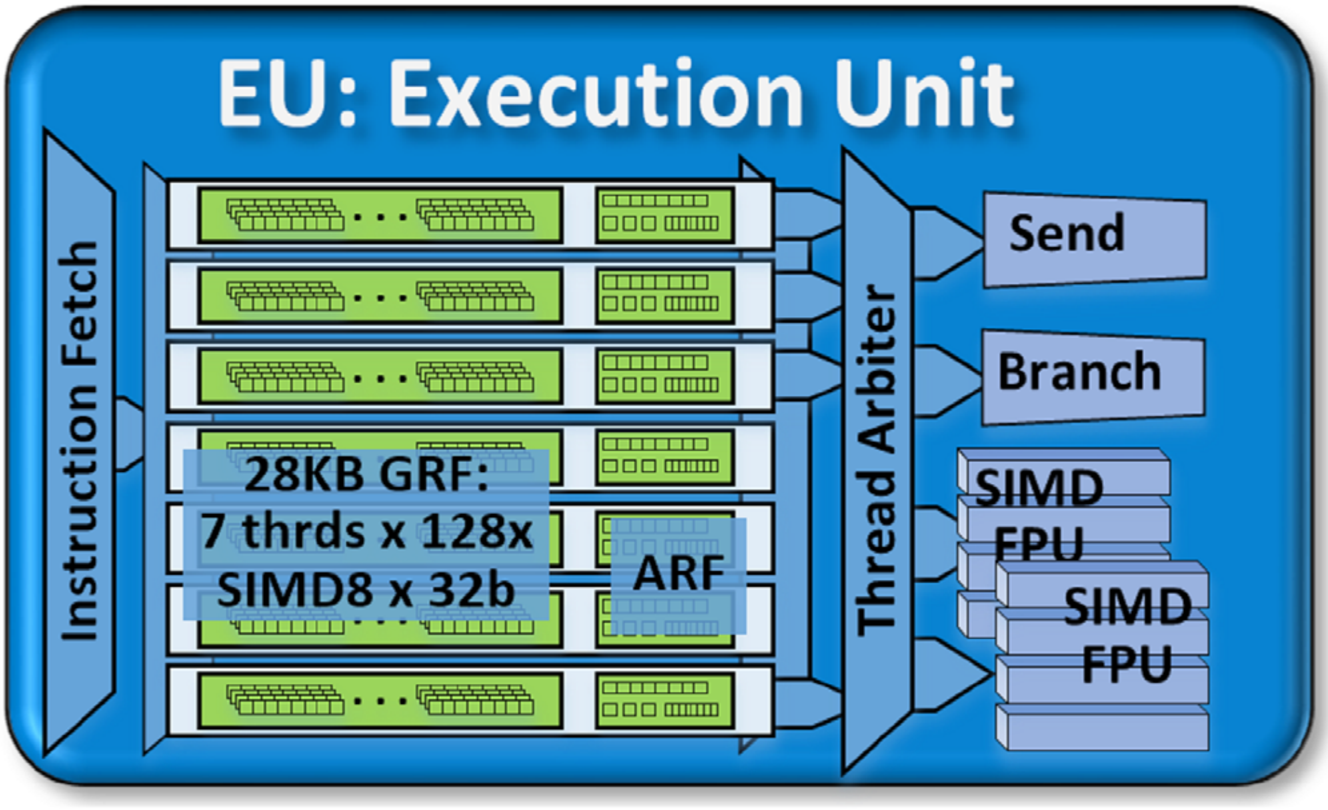
1 Multikernel FFT execution flow

Optimization efforts were focused on single dimensional (1D) input signals. The genFFT implementation is based on Cooley-Tukey. Performance analysis on the initial implementation showed that Cooley-Tukey is memory-bandwidth-limited on Intel Processor Graphics. For this reason, there was little motivation to investigate algorithms (Yavne, 1968; Duhamel and Hollmann, 1984; and Johnson and Frigo, 2007), which reduce the total number of math operations even further. Instead, the focus was switched to improving the efficiency of the memory operations on Intel Processor Graphics.

To accomplish this, it was necessary to address two questions:

1. How to make the most of the available register space, which is the fastest memory available on Intel Processor Graphics
2. How to reduce the cost of transferring the data between the global memory and registers

Figure 2 shows an Intel Processor Graphics execution unit (EU).



2 Intel® Processor Graphics Execution Unit (Junkins, 2015).

From architectural specifications, it is possible to compute the maximum length FFT that can be performed by a single work item without spilling from hardware registers. As shown in **Figure 2**, each hardware thread of each EU contains 128 32-byte-wide registers. Programs may execute 8, 16, or 32 work items per thread, one work item per SIMD lane, depending on the SIMD width chosen by the compiler. The register space is divided evenly among the work items in a hardware thread. If a kernel is compiled SIMD8, each work item will have access to the maximum 512 bytes (4KB/8). In single precision, 512 bytes correspond to 64 complex values. Thus, the maximum FFT length that can theoretically fit in the registers is 32 (computations require 32 single-precision complex values) because the compiler needs some register space to accommodate other program variables as well.

The OpenCL compiler ultimately chooses which private data resides in registers and which data spills to global memory. This means that the maximum FFT length that fits in the registers depends not only on the available register space, but also on the compiler's ability to promote the relevant data to the registers. Allocating in private memory only as much data as it can probably fit into registers gives the compiler a better chance of success. The code generator can be used to determine empirically what is the largest FFT length that fits in the registers. This is an important component to have before the next step, which is computation of FFT for signals that don't fit in the registers.

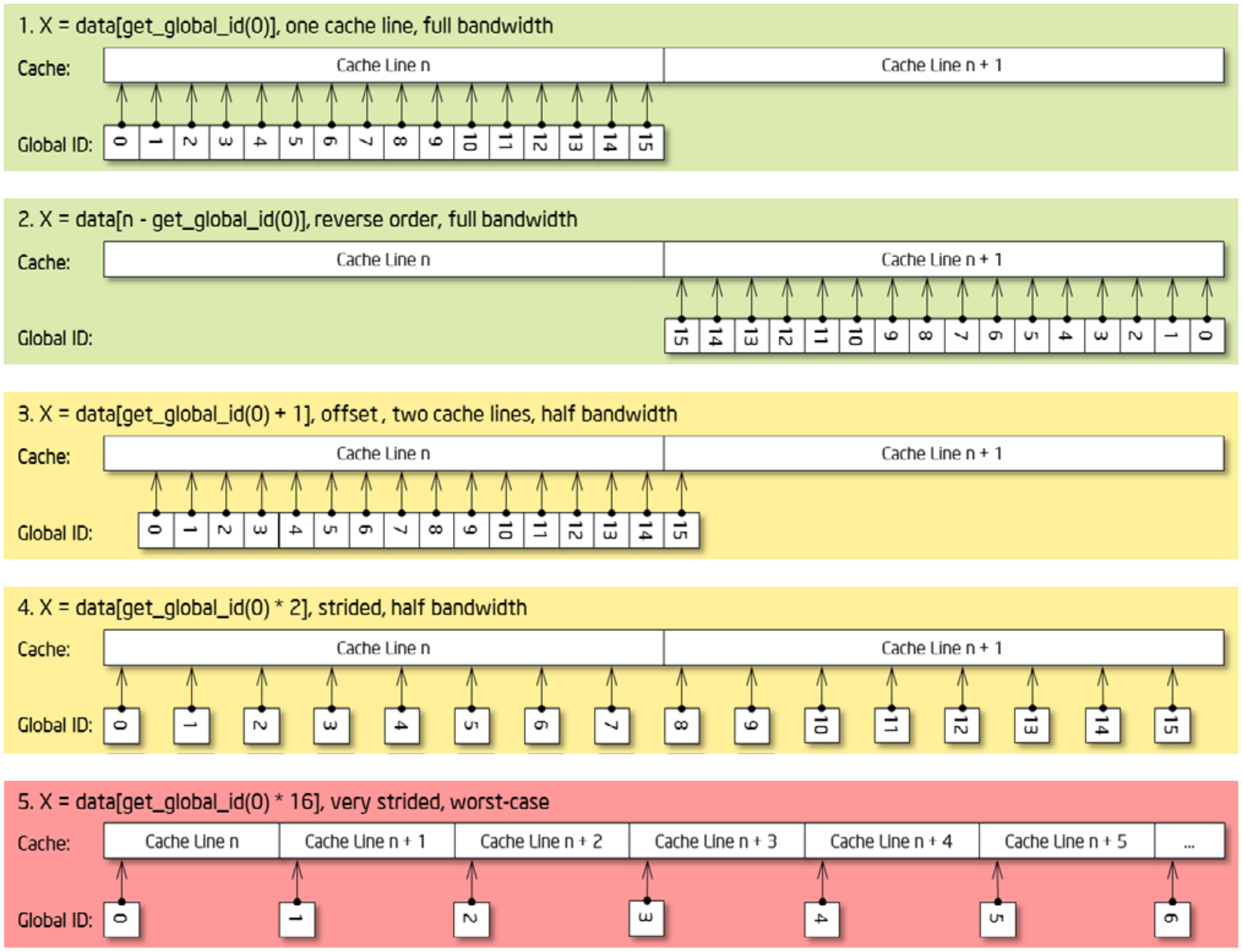
Performance depends on a deep understanding of how the OpenCL™ API maps to the underlying architecture.

The performance analysis of the first base FFT kernels showed that there was a high cost of transferring data between the global memory and registers. This overhead can be reduced by making sure that the input signal fits into the cache by reducing:

- The number of cache lines touched
- The number of memory address requests to the EU data port

Accessing 2 × 32 bit quantities at a time per work item in a column-major fashion has the effect that hardware threads access only multiples of cache lines from cache line-aligned addresses, thus reducing the cost of the data transfers.

Figure 3 shows cache access patterns to consider when transferring data between the global memory and registers.



3 Cache access patterns to consider when transferring data between the global memory and registers (Ashbaugh, 2013; Kunze, 2014)

Having implemented very efficient base FFTs (8, 16, and 32), it makes sense to attempt an implementation that decomposes any FFT length power-of-two into these base FFTs. Cooley Tukey makes this possible, since it can be generalized to any factor decomposition.

There are numerous decompositions into power-of-two factors. Our code generator will generate code for any such combination. The experimental data collected during the performance analysis of the base FFT kernels was used to design the following heuristic for selecting the best decomposition on Intel Processor Graphics. First, the number of factors must be minimized. This is accomplished by selecting factors as close as possible to the maximum FFT length that can be performed in registers. Second, the difference between the last two factors must be minimized. The code generator automatically chooses the decomposition, but it is ultimately the programmer's responsibility to provide the maximum FFT length that fits in the registers for the target architecture.

Input/Output Sorting

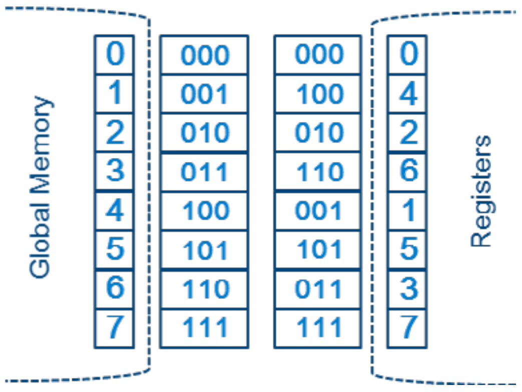
It is important to consider a few important details about how genFFT works:

- Input/output sorting
- Base FFT local twiddle factors
- Intermediary twiddle factors

The Cooley-Tukey algorithm requires a sorting step either on the input (Decimation in Time, or DIT) or on the output (Decimation in Frequency). Typically, this is done by applying bit reversal on the input/output indices when reading/writing the data.

In our DIT implementation, there's no global bit reversal on input. Instead, each base FFT performs bit reversal on its local input. This decision was based on the observation that the compiler is more likely to replace the bit reversal math with constant indices when the FFT fits in the registers.

Figure 4 shows the bit reversal radix-2 8-point FFT.



4 Bit reversal radix-2 8 point FFT

The distributed bit reversal in itself is quite inexpensive; however, the sorting is not complete at the end of a multikernel FFT pipeline. Additional bit rotations on the output indices must be performed out of place, reducing cache effectiveness (and thus performance).

It can be argued that the rotations at the end exhibit a more regular memory access pattern than a full-blown bit reversal. However, given our column major data layout in memory, and the fact that all work items in a hardware thread access the same data row index in lock-step, and that both approaches require out of place copies, there's practically no observable performance difference between a global bit reversal on input and our distributed bit reversal followed by rotations. If nothing else, our approach may have saved us a few math operations for larger FFTs that don't fit in the registers.

Figure 5 shows 16-point FFT indices for bit reversal, 2 × 8 rotation, and 4 × 4 rotation. Rotations are simpler and require less math operations to perform.

Sorted	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit Reversal	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
2x8 Rotation	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15
4x4 Rotation	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15

5 16-point FFT indices for bit reversal, 2 × 8 rotation, and 4 × 4 rotation

Base FFT Twiddle Factors

An N-point base FFT requires N/2 complex twiddle factors[1] (also known as "roots of unity" [Cooley-Tukey]) that can be difficult to store in registers. The twiddle factors can be computed either on the fly or provided in the form of a lookup table. On-the-fly computation is based on the use of *sin* and *cos* transcendental instructions, which operate in either high performance/low accuracy or low performance/high accuracy modes. The lookup table approach was used because, in this case, it offers both high performance (no additional math required) and high accuracy (the table is generated on the CPU in double precision). To minimize the register pressure, the size of the lookup table was reduced from N/2 to just N/4+1 single-precision values, a reduction made possible by the periodic nature of the *sin* and *cos* functions. The twiddle factors lookup table is stored in the kernel file in the form of a constant array.

```
constant float W[9] =
{
          +0.0f,
   +0.19509f,
   +0.38268f,
   +0.55557f,
   +0.70711f,
   +0.83147f,
   +0.92388f,
   +0.98078f,
          +1.0f,
};
```

6 Base FFT twiddle factors for radix-2 32-point FFT

Figure 6 shows base FFT twiddle factors for a radix-2 32-point FFT.

Intermediary Twiddle Factors

The intermediary twiddle factors, which are used to integrate the results of all the base FFTs into the larger FFT, are applied between each pair of base FFTs. Similar to the base FFT twiddle factors, they can be computed on the fly or provided as a lookup table. The lookup table approach, which balances performance and accuracy, was the choice here as well. Due to its larger size,[2] the intermediate twiddle factors lookup table is passed to the base FFT kernels as a kernel argument.

1. One of the first mentions we could find for "twiddle factors" is in Cochran and Cooley (1967) and it is attributed to Cooley, Lewis, and Welch.

2. In one experiment, we reduced the size of the intermediate lookup table from N single-precision complex values to N/4+1 single-precision real values and stored it in the kernel file, but the additional cost incurred by index calculations and dynamic array indexing led to a greater-than-10-percent performance loss.

Eight Point FFT Case Study

We can use snippets from the 8-point FFT OpenCL code to showcase some of the techniques described in this article. The kernel code is organized into these sections:

- Preprocessor constants
- Base FFT twiddle factors
- Common utilities
- Base FFT procedure code
- Kernel

Preprocessor Constants

Figure 7 shows a code snippet illustrating preprocessor constants. genFFT generates code for the global and local sizes required by a particular use case. This enables the compiler to unroll all the loops in the base FFT code and, in doing so, avoid any dynamic array indexing that could otherwise impact performance.

```
#define GLOBAL_SIZE_X 8
#define GLOBAL_SIZE_Y 8
#define GLOBAL_SIZE_Z 1

#define LOCAL_SIZE_X  8
#define LOCAL_SIZE_Y  1
#define LOCAL_SIZE_Z  1
```

7 Code snippet 1, preprocessor constants

Base FFT Twiddle Factors

Figure 8 shows code snippet 2, base FFT twiddle factors, with lookup tables for 8-, 16-, and 32-point FFT.

```
// 8 point base FFT LUT
constant float W[3] =
{
     +0.000000000000000f,


     +0.707106781186547f,


     +1.000000000000000f,
};
```

```
// 16 point base FFT LUT
constant float W[5] =
{
     +0.000000000000000f,

     +0.382683432365090f,

     +0.707106781186547f,

     +0.923879532511287f,

     +1.000000000000000f,
};
```

```
// 32 point base FFT LUT
constant float W[9] =
{
     +0.000000000000000f,
     +0.195090322016128f,
     +0.382683432365090f,
     +0.555570233019602f,
     +0.707106781186547f,
     +0.831469612302545f,
     +0.923879532511287f,
     +0.980785280403230f,
     +1.000000000000000f,
};
```

8 Code snippet 2, base FFT twiddle factors. Lookup tables for 8-, 16-, and 32-point FFT.

Each kernel gets its own base FFT twiddle factors lookup table. As you can see in figures 7 and 8, there are similarities between the lookup tables corresponding to different base FFT kernels. In other words, the N/2-point base FFT lookup table is included in the N-point FFT lookup table. When fusing together two or more base FFT kernels, a technique discussed later in this article, this property allows us to maintain a single lookup table.

Common Utilities

This section includes the following procedures:

- Bit reversal
- Complex multiplication
- N-point FFT butterfly procedure

Figure 9 shows the N-point FFT butterfly procedure.

```
void Butterfly(float2* x, int crtfft)
{
    // apply twiddle factors
    for (int i = 0; i < crtfft / 4; i++)
    {
        ComplexMul(&x[2 * crtfft / 4 + i], (float2)(+W[N / 4 - N / crtfft * i], -W[N / crtfft * i]));
        ComplexMul(&x[3 * crtfft / 4 + i], (float2)(-W[N / crtfft * i], -W[N / 4 - N / crtfft * i]));
    }

    // perform butterflies
    for (int i = 0; i < crtfft / 2; i++)
        FFT2(&x[i], &x[i + crtfft / 2]);
}
```

9 Code snippet 3, N-point FFT butterfly procedure

N in the butterfly procedure is replaced with a constant in the actual code, while `crtfft` represents the current radix-2 FFT stage and takes values between 2 and N. The procedure handles all the butterflies and applies all the twiddle factors corresponding to a particular FFT stage.

Base FFT Procedure Code

Figure 10 shows an 8-point base FFT procedure.

```
void FFT_8x1x1_stride_1(int gy, float2* x, global float2* cdata)
{
    int gx = get_global_id(0);

    global float2* crtcData = cdata + gy * GLOBAL_SIZE_X * 8 + gx;

    // READ signal
    for (int i = 0; i < 8; i++)
        x[BitReverse(i)] = crtcData[i * GLOBAL_SIZE_X];

    // perform FFT
    for (int crtfft = 2; crtfft <= 8; crtfft <<= 1)
        for (int offset = 0; offset < 8; offset += crtfft)
            Butterfly(x + offset, crtfft);

    // WRITE spectrum
    for (int i = 0; i < 8; i++)
        crtcData[i * GLOBAL_SIZE_X] = x[i];
}
```

10 Code snippet 4, 8-point base FFT procedure

The base FFT procedure reads the input signal, performs the FFT butterflies, and saves the result (the spectrum). Since performance analysis of the base FFT kernels showed that the execution time was the same when the computation was omitted, there was a strong indication that any additional reduction in the number of math operations required to perform a base FFT will not lead to any observable performance improvements.

Kernel

Figure 11 shows the 8-point base FFT kernel.

```
__kernel
__attribute__((reqd_work_group_size(LOCAL_SIZE_X, LOCAL_SIZE_Y, LOCAL_SIZE_Z)))
void FFT_8x1x1_kernel_stage_0_of_1(global float2* cdata)
{
    float2 x[8];

    FFT_8x1x1_stride_1(get_global_id(1), x, cdata);
}
```

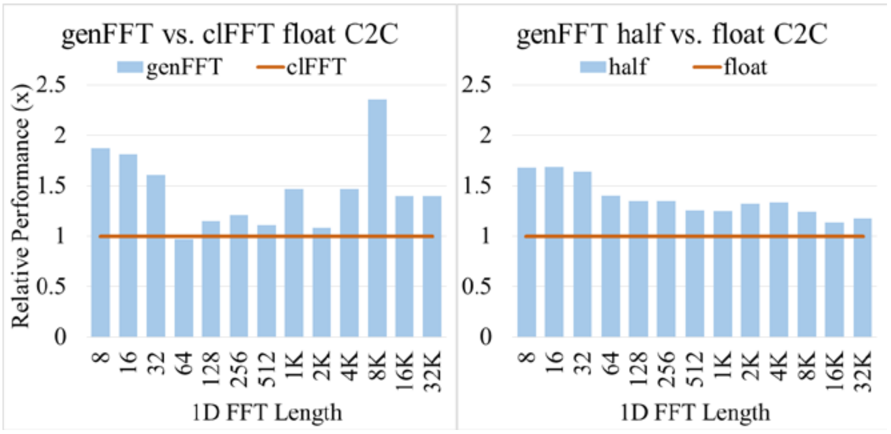
11 Code snippet 5, 8-point base FFT kernel

Results

genFFT performance was evaluated relative to clFFT on Intel® HD Graphics 530. clFFT is an excellent open source implementation that performs well on Intel Processor Graphics.

To reduce variability in our results due to thermal throttling, the frequency of Intel HD Graphics 530 was capped at 750 MHz. Performance analysis was conducted using a buffer large enough to fill the GPU with work, but small enough to fit in the last-level cache (LLC). **Figure 12** shows results for single-precision genFFT versus clFFT and for half-precision versus single-precision on genFFT.

As you can see in Figure 12, there are two important cases in which genFFT outperforms clFFT substantially. The first case corresponds to the base FFTs. In this case, the base FFT kernels outperform clFFT by more than 1.5X and achieve a memory throughput of up to 90 percent of the peak LLC memory bandwidth (GB/s). The second case likely corresponds to some inefficiency in the clFFT implementation and is a perfect example of the performance portability issues that can impact OpenCL applications.



12 Relative performance of genFFT versus clFFT on Intel® HD Graphics 530 capped at 750 MHz

The FFT generator can produce code for various combinations of input and output data types and supports:

- Single-precision on the input signal, the output spectrum, and computation
- Half-precision on the input signal, the output spectrum, and computation
- Signed and unsigned 16-bit and 32-bit integers on the input signal and single-precision on the output spectrum and computation

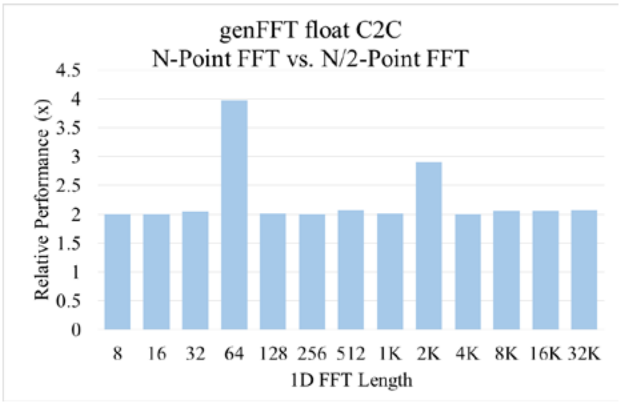
Of particular interest is the use of the half-precision data type (**Figure 12**). This offers additional performance gains in circumstances that can tolerate the accuracy loss induced by the half-precision data type.

There are a couple of cases where genFFT performance drops relative to clFFT:

1. 64-point FFT
2. 2k-point FFT

These correspond to the increase in the number of base FFT kernels in the pipeline. As mentioned earlier, one of the criteria to maximize genFFT performance was to minimize the number of factors that the global FFT length is decomposed. This is perhaps better illustrated in **Figure 13**, which shows the relative performance of N-point FFT versus N/2-point FFT. **Figure 13** shows two things. First, genFFT is memory bandwidth-constrained so the duration of the N-point FFT is twice that of the N/2-point FFT. In other words, the execution time is dominated by the cost of the memory transfers between the global memory and the registers, so twice as much data to move around means twice the execution time. There are two notable exceptions to this observation:

1. The 64-point FFT takes about four times the execution time required for 32-point FFT.
2. The 2k-point FFT takes about three times the execution time required by the 1k-point FFT.



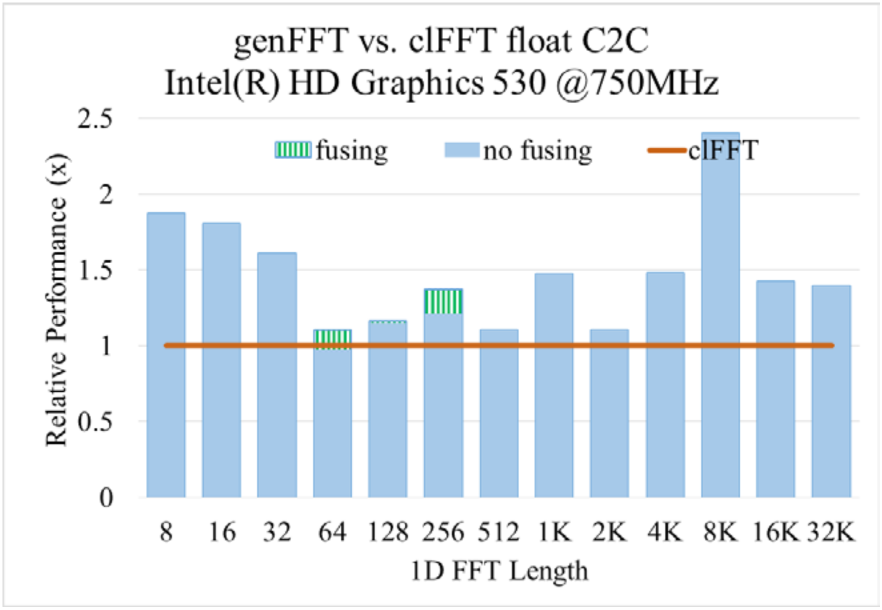
13 Relative performance of N-point FFT versus N/2-point FFT on Intel® HD Graphics 530 capped at 750 MHz

This is evidence of a penalty associated with enqueueing more kernels as lengths 64 to 1k require two kernels and lengths 2k to 32k require 3 kernels.

Figure 13 shows the relative performance of N-point FFT versus N/2-point FFT on Intel HD Graphics 530 capped at 750 MHz. **Figure 14** shows a kernel-fusing technique that can help mitigate the back-to-back kernel enqueue penalty.

Kernel Fusing

One potential solution for addressing the back-to-back kernel enqueue penalty observed previously is to fuse groups of two or more kernels into a single one. In the case of the 128-point FFT, the code generator breaks down the FFT into two stages, a 16-point FFT with stride 8 followed by an 8-point FFT.



14 Kernel fusing technique that can help mitigate the back-to-back kernel enqueue penalty on Intel® HD Graphics 530 capped at 750 MHz

```
__kernel
__attribute__((reqd_work_group_size(LOCAL_SIZE_X, LOCAL_SIZE_Y, LOCAL_SIZE_Z)))
void FFT_128x1x1_stage_0(global float2* cdata, global float2* spectrum, global float2* twiddles)
{
    float2 x[16];

    FFT_16x1x1_stride_8(get_global_id(1), x, cdata, spectrum, twiddles);
}
```

15 Code snippet 6, 16-point FFT stride 8, first stage toward the 128-point FFT

```
__kernel
__attribute__((reqd_work_group_size(LOCAL_SIZE_X, LOCAL_SIZE_Y, LOCAL_SIZE_Z)))
void FFT_128x1x1_stage_1(global float2* cdata, global float2* spectrum, global float2* twiddles)
{
    float2 x[8];

    FFT_8x1x1_stride_1(get_global_id(1), x, cdata, spectrum, twiddles);
}
```

16 Code snippet 7, 8-point FFT stride 1, second stage toward the 128-point FFT

The straightforward way to perform kernel fusing in this case is to call the 16-point FFT and the 8-point FFT procedures one after another inside the same kernel. To cover the whole input signal, twice as many 8-point FFT calls are needed as 16-point FFT calls. Since the output of each FFT stage is the input of the next stage, global memory must be made consistent between the FFT stages. In OpenCL, workgroup barriers can be used to guarantee global memory consistency across work items in the same workgroup.

Figure 17 shows a 128-point FFT fused kernel performing a 16-point FFT, followed by an 8-point FFT.

```
__kernel
__attribute__((reqd_work_group_size(LOCAL_SIZE_X, LOCAL_SIZE_Y, LOCAL_SIZE_Z)))
void FFT_128x1x1(global float2* cdata, global float2* spectrum, global float2* twiddles)
{
    float2 x[16];

    FFT_16x1x1_stride_8(get_global_id(1), x, cdata, spectrum, twiddles);

    barrier(CLK_GLOBAL_MEM_FENCE);

    FFT_8x1x1_stride_1(2 * get_global_id(1) + 0, x, cdata, spectrum, twiddles);

    FFT_8x1x1_stride_1(2 * get_global_id(1) + 1, x, cdata, spectrum, twiddles);
}
```

17 Code snippet 8, 128-point FFT fused kernel performing a 16-point FFT followed by an 8-point FFT

Using this approach, it is possible to fuse kernels for the 64, 128, 256, 512, and 1k FFTs. The performance benefits are proportional with the penalty of enqueueing back-to-back kernels on a particular architecture. Fusing kernels using a barrier leads to a more than 10 percent gain for the 64-point FFT and the 256-point FFT, which brings us slightly above clFFT for all the measured cases.

There are, however, drawbacks to this technique. Ever larger workgroups must be chosen to synchronize memory across more work items as FFT stages are fused.[3] This is avoidable if the work per work item is increased sufficiently (eight 16-point FFT calls followed by 16 8-point FFT calls in the case of the 128-point FFT). But experimental observations have shown a drop in performance relative to the barrier version.

3. In the 128-point FFT example, if the local size of the 16-point FFT stage before fusing was (8, 1, 1), the local size after fusing will need to be (8, 128/16, 1), which is equivalent to a workgroup size of 64. On Intel® HD Graphics 530, the maximum workgroup size is 256.

Conclusion

This article showcases an FFT implementation that avoids SLM, barriers, and vendor extensions and can execute under any combination of the following:

- Global size
- Local size
- SIMD size
- The size of the register

genFFT obtained performance gains in some cases. Others can be improved even more in the future.

Future Work

There's some evidence to support the fact that half-precision implementation performance can be further increased. This will require changes to better utilize the register space and to maximize the memory throughput of the transactions between the global memory and the registers. The penalty of enqueueing multiple back-to-back kernels also needs to be addressed. Finally, the scope of genFFT can be expanded to higher dimensions and arbitrary sizes.

Acknowledgments

We'd like to acknowledge the following for their valuable feedback while working on this project: Murali Sundaresan, Mike MacPherson, John Wiegert, Tim Bauer, Adam Herr, Robert Ioffe, Michal Mrozek, Jonathan Pearce, and Simon Finn.

References

1. Ashbaugh, Ben, 2013. **Taking Advantage of Intel® Graphics with OpenCL™**.
2. **clFFT**
3. Cochran, W.T., and Cooley, J.W. 1967. What Is the Fast Fourier Transform. IEEE Trans. Audio and Electroacoustics. AU-15 (June 1967), 44–55.
4. Cooley, J.W., and Tukey, J.W. 1965. An Algorithm for the Machine Computation of Complex Fourier Series. Mathematics of Computation. 19 (90). 297–301.
5. Duhamel, P., and Hollmann, H. 1984. Split-Radix FFT Algorithm, Electron. Lett., 20 (Jan 1984), 14–16.
6. Gaster, B., Kaeli, D. R., Howes, L., Mistry, P., and Schaa, D. 2011. Heterogeneous Computing with OpenCL™. Elsevier Science & Technology.
7. **Intel® Processor Graphics Gen8-Gen9 Developer Guides.** 2015.
8. Johnson, S. G., and Frigo, M. 2007. A Modified Split-Radix FFT with Fewer Arithmetic Operations IEEE Trans. Signal Process., 55(1), 111–119.
9. Junkins, Stephen. 2014. **The Compute Architecture of Intel® Processor Graphics Gen8.**
10. Junkins, Stephen. 2015. **The Compute Architecture of Intel® Processor Graphics Gen9.**
11. Khronos OpenCL Working Group. **The OpenCL™ Specification Version 1.2, 2.0.** 2015.
12. Kunze, Aaron, 2014. **Intel® Processor Graphics: Optimizing Computer Vision and More.**
13. Lloyd, D. B., Boyd, C., and Govindaraju, N. 2008. Fast Computation of General Fourier Transforms on GPUs. Microsoft. IEEE International Conference on Multimedia and Expo. (ICME 2008), 5–8.
14. Lyons, R. G. 2004. Understanding Digital Signal Processing, 3rd Ed., Prentice Hall Publishing, Upper Saddle River, NJ.
15. Petre, D., Lake, A., Hux, A. 2016. **OpenCL™ FFT Optimizations for Intel® Processor Graphics. IWOCL 2016.**
16. Yavne, R. 1968. An Economical Method for Calculating the Discrete Fourier Transform. Proc. AFIPS Fall Joint Comput. Conf., Washington DC, 1968, 33. 115–125.

INDIRECT CALLS AND VIRTUAL FUNCTIONS CALLS: VECTORIZATION WITH INTEL® C/C++ 17.0 COMPILERS

The Newest Intel C++ Compiler Introduces Support for Indirectly Calling a SIMD-Enabled Function in a Vectorized Fashion

Hideki Saito, *Principal Engineer*; Serge Preis, *Principal Engineer*; Sergey Kozhukhov, *Senior Software Engineer*; Xinmin Tian, *Principal Engineer*; Clark Nelson, *Principal Engineer*; Jennifer Yu, *Senior Software Engineer*; Sergey Maslov, *Senior Software Engineer*; and Udit Patidar, *Product Marketing Engineer*, Intel Corporation

The Parallel Universe Issue 22 elaborated the SIMD support features of the OpenMP* 4.0 specification. One of the big features of OpenMP 4.0 is the ability to vectorize a user-written function (called a SIMD-enabled function) and the ability to call vectorized SIMD-enabled functions from vectorized code. SIMD-enabled functions greatly improve the programmability of the underlying **SIMD** hardware by utilizing vector instructions on regular functions and subroutines to operate in a data parallel context (inside a vector loop or inside another SIMD-enabled function) on many elements simultaneously.

Previously, the mechanism of translating a scalar function call, written by the programmer, into a vectorized function call in the executable depended on the compile-time resolution of determining which scalar function was being called. Therefore, in many cases, they were limited to direct calls only.

Indirect function calls, such as those made through a function pointer or a virtual function call, are powerful aspects of modern programming languages such as C++. The main advantage of such a function pointer is to simplify the application code by providing a straightforward way to select a function to execute based on runtime values. Because of the nature of an indirect function call, the actual callee function is determined only at runtime. As a result, Intel® C++ Compilers up to version 16.0 serialized all indirect calls within the vectorized context, unless these calls are optimized into direct calls prior to vectorization. This caused a severe bottleneck in the execution of indirectly invoked SIMD-enabled functions, because the compiler could not vectorize these indirect calls and therefore could not take full advantage of the vector capabilities of the underlying SIMD-enabled hardware.

Indirect function calls are powerful aspects of modern programming languages such as C++.

Intel C++ Compiler 17.0, part of the upcoming Intel® Parallel Studio XE 2017 (available freely during the **beta program**), introduces support for indirectly calling a SIMD-enabled function in a vectorized fashion. We also introduce new syntax for declaring SIMD-enabled function pointers. A set of compatibility rules are enforced in order to enable scalar function to vector variant mapping without actually resolving the identity of the scalar function at compile time. The following is a short example of declaring and using an indirect call to a SIMD-enabled function.

You may associate several vector attributes with one SIMD-enabled function pointer, which reflects all the variants available for the target functions to be called through the pointer. Encountering an indirect call, the compiler matches the vector variants declared on the function pointer with the actual parameter kinds and chooses the best match. Consider **Figure 1** showing the declaration of vector function pointers and loops with indirect calls:

```
// pointer declaration

// universal but slowest definition matches the use in all three loops
__declspec(vector)

// matches the use in the first loop
__declspec(vector(linear(in1), linear(ref(in2)), uniform(mul)))

// matches the use in the second and the third loops
__declspec(vector(linear(ref(in2))))

// matches the use in the second loop
__declspec(vector(linear(ref(in2)), linear(mul)))

// matches the use in the third loop
__declspec(vector(linear(val(in2:2))))

int (*func)(int* in1, int& in2, int mul);
int *a, *b, mul, *c;
int *ndx, nn;
...
// loop examples
   for (int i = 0; i < nn; i++) {
       /*
        * In the loop, the first parameter is changed linearly,
        * the second reference is changed linearly too
        * the third parameter is not changed
        */
       c[i] = func(a + i, *(b + i), mul);
   }

   for (int i = 0; i < nn; i++) {
       /*
        * The value of the first parameter is unpredictable,
        * the second reference is changed linearly
        * the third parameter is changed linearly
        */
       c[i] = func(&a[ndx[i]], b[i], i + 1);

   }

   #pragma simd private(k)
   for (int i = 0; i < nn; i++) {

       /*
        * During vectorization, private variables are transformed into arrays:
        * k->k_vec[vector_length]
        */
       int k = i * 2;

       /*
        * The value of the first parameter is unpredictable,
        * the second reference and value can be considered linear
        * the third parameter has unpredictable value
        * (the __declspec(vector(linear(val(in2:2)))) value
        * will be chosen from the two matching variants)
        */
       c[i] = func(&a[ndx[i]], k, b[i]);
   }
```

1 Declaration of vector function pointers and loops with indirect calls

The invocation of a SIMD-enabled function directly or indirectly provides arrays wherever scalar arguments are specified as formal parameters. It is also possible to indirectly invoke a SIMD-enabled function within a parallel context. **Figure 2** shows two invocations that give instruction-level parallelism by having the compiler issue special vector instructions.

```
__declspec(vector)
float (**vf_ptr)(float, float);

//operates on the whole extent of the arrays a, b, c
a[:] = vf_ptr[:] (b[:],c[:]);

/*
 * use the full array notation construct to also specify n
 * as an extend and s as a stride
 */
a[0:n:s] = vf_ptr[0:n:s] (b[0:n:s],c[0:n:s]);
```

2 Code sample demonstrating indirect invocation of a SIMD-enabled function within a parallel context

Intel® C++ Compiler 17.0 introduces support for indirectly calling a SIMD-enabled function in a vectorized fashion.

Preliminary unit-test performance results are shown in **Figure 3**, using preproduction Intel C++ Compiler 17.0 and Intel® Xeon® processor E3-1240 v3. The unit test invokes function `FUNC()` in a loop and performs a sum reduction operation. The function may be a direct call (serialized or vectorized) or an indirect call (serialized or vectorized). The loop trip count NTIMES is set such that the test runs longer than one second in wall clock time. **Figure 4** shows the overhead of vectorizing an indirect call. Baseline is defined as directly invoking a function without using a SIMD-enabled function feature. First, let's look at the "direct, vector" results. With SIMD-enabled function feature, we expect approximately 4X performance, by computing four double-precision computation. With the TINY computation test mode, the overhead of serializing a function call is more visible than the SMALL/MEDIUM modes and, thus, the relative performance observed was 4X–5X.

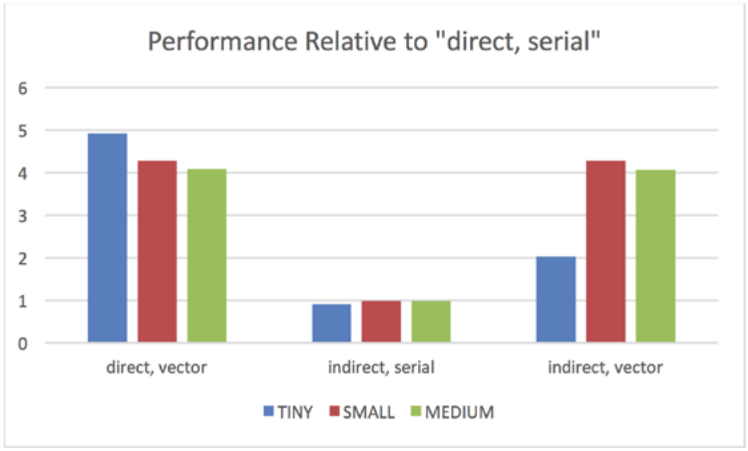
Next, the “indirect, serial” result shows that indirectly calling the function serially does not add much overhead (over directly calling the function). Finally, the “indirect, vector” result shows that the overhead of indirect vector function invocation is highly visible for the TINY mode, but it can be amortized if the amount of compute increases. This is the best-case scenario since all vector elements are invoking the same function.

```
double sum=0;
#pragma omp simd reduction(+:sum)
for (i=0;i<NTIMES;i++){
    sum += FUNC(i);
}
#ifdef SIMD_ENABLED_FUNC
#pragma omp declare simd processor(core_4th_gen_avx)
#endif
__declspec(noinline)
double unit_test0(double x){
    return COMPUTE(x);
}
```

3 Code sample used for unit test performance measurement of vectorized virtual functions in Intel® C++ Compiler 17.0. The code sample performs a sum reduction operation.

Test Mode	COMPUTE(x)	NTIMES
TINY	x+1	2,000,000,000
SMALL	exp(sin(x))	200,000,000
MEDIUM	log(exp(sin(x)))	200,000,000

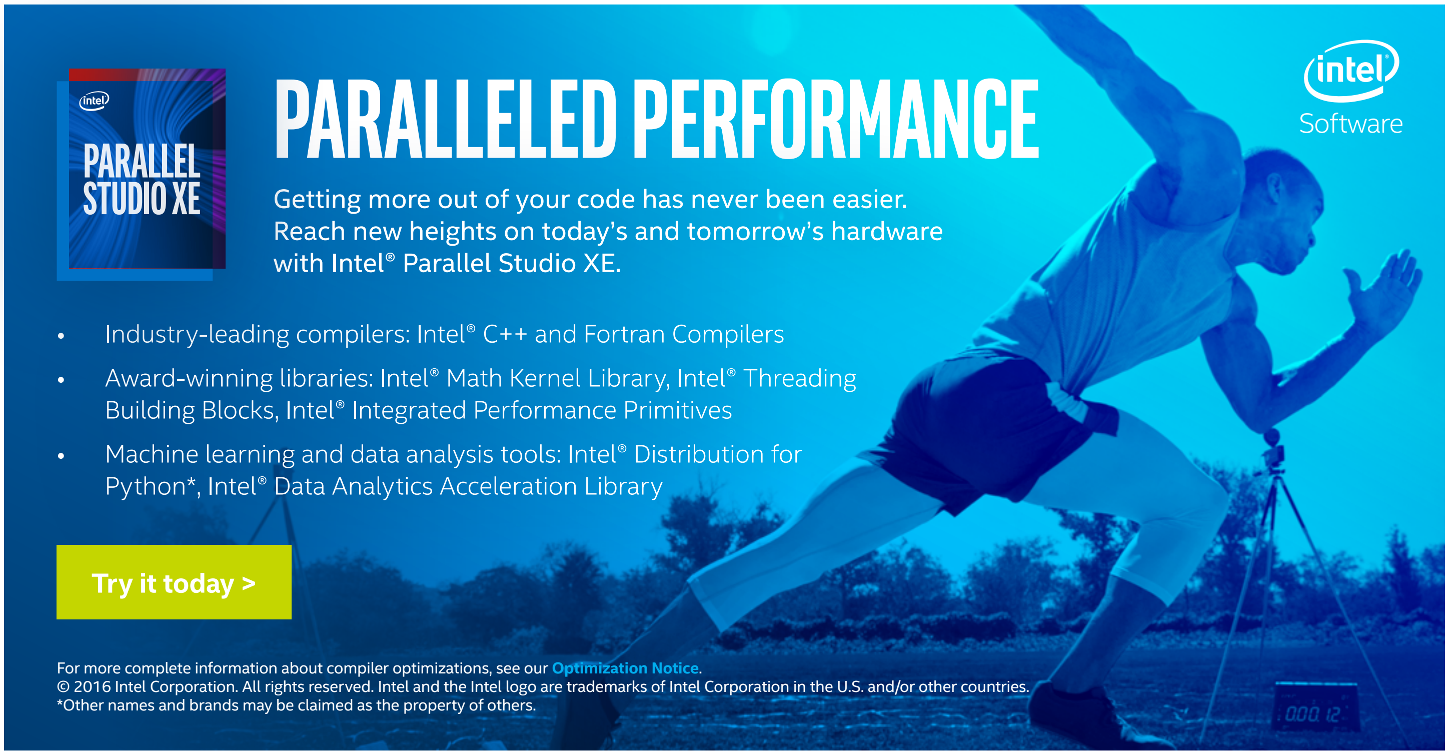
Label	FUNC	SIMD_ENABLED_FUNC	SIMD_ENABLED_FUNC_PTR
direct, serial	unit_test0	no	N/A
direct, vector	unit_test0	yes	N/A
indirect, serial	funcptr0[0]	no	N/A
indirect, vector	funcptr0[0]	yes	yes



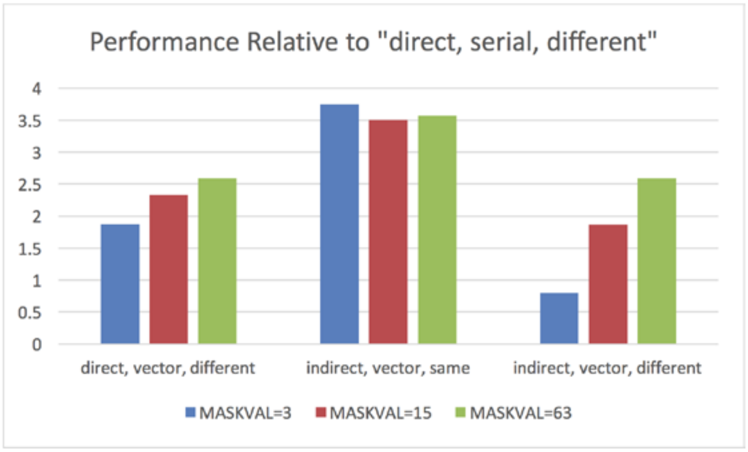
4 Indirect call vectorization overhead

Figure 5 shows the overhead of divergent calls. In this example, `FUNC()` is a direct call or an indirect call to either `unit_test0()` or `unit_test1()`, that is dependent on the loop index value. For this experiment, the TINY compute mode was used to highlight the overhead of divergence applied to the overhead of indirectly calling SIMD-enabled functions. For the vector length of four, the SIMD-enabled function call diverges every vector iteration with MASKVAL=3, every four vector iterations with MASKVAL=15, and every 16 vector iterations with MASKVAL=63. The middle set of bars labeled "indirect, vector, same" is interesting. Although it uses two different function pointer array elements, since the actual content of the function pointers is identical, the final call is not divergent, and the performance shows that the compiler takes advantage of it. Comparing the leftmost set of bars labeled "direct, vector, different" and the rightmost set of bars labeled "indirect, vector, different," the overhead of indirectly calling divergent calls reduces with the actual divergence.

Label	FUNC	SIMD_ENABLED_FUNC	SIMD_ENABLED_FUNC_PTR
direct, serial, different	((i & MASKVAL) ? unit_test0 : unit_test1)	no	N/A
direct vector, different	((i & MASKVAL) ? unit_test0 : unit_test1)	yes	N/A
indirect, vector, same	((i & MASKVAL) ? funcptr0[0] : funcptr0[1])	yes	yes
indirect, vector, different	((i & MASKVAL) ? funcptr1[0] : funcptr1[1])	yes	yes


PARALLELED PERFORMANCE
intel Software
intel PARALLEL STUDIO XE
Getting more out of your code has never been easier. Reach new heights on today's and tomorrow's hardware with Intel® Parallel Studio XE.
• Industry-leading compilers: Intel® C++ and Fortran Compilers
• Award-winning libraries: Intel® Math Kernel Library, Intel® Threading Building Blocks, Intel® Integrated Performance Primitives
• Machine learning and data analysis tools: Intel® Distribution for Python*, Intel® Data Analytics Acceleration Library
Try it today >
For more complete information about compiler optimizations, see our Optimization Notice.
© 2016 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.

The usage models of SIMD-enabled virtual function calls are similar to those of SIMD-enabled member function calls, and the performance characteristics are similar to the indirect call examples shown previously. We invite you to exercise this new feature of Intel C++ Compiler 17.0 if the compute-intensive hotspots of your code contain indirect calls, and they are legal to vectorize.



5 Overhead of divergent calls

Additional Resources

SIMD-Enabled Functions

SIMD-Enabled Function Pointers

OPTIMIZING AN ILLEGAL IMAGE FILTER SYSTEM

Tencent Doubles the Speed of Its Illegal Image Filter System Using a SIMD Instruction Set and Intel® Integrated Performance Primitives

Yueqiang Lu, *Application Engineer*; Ying Hu, *Technical Consulting Engineer*; and Huaqiang Wang, *Application Engineer*, Intel APAC R&D Ltd.

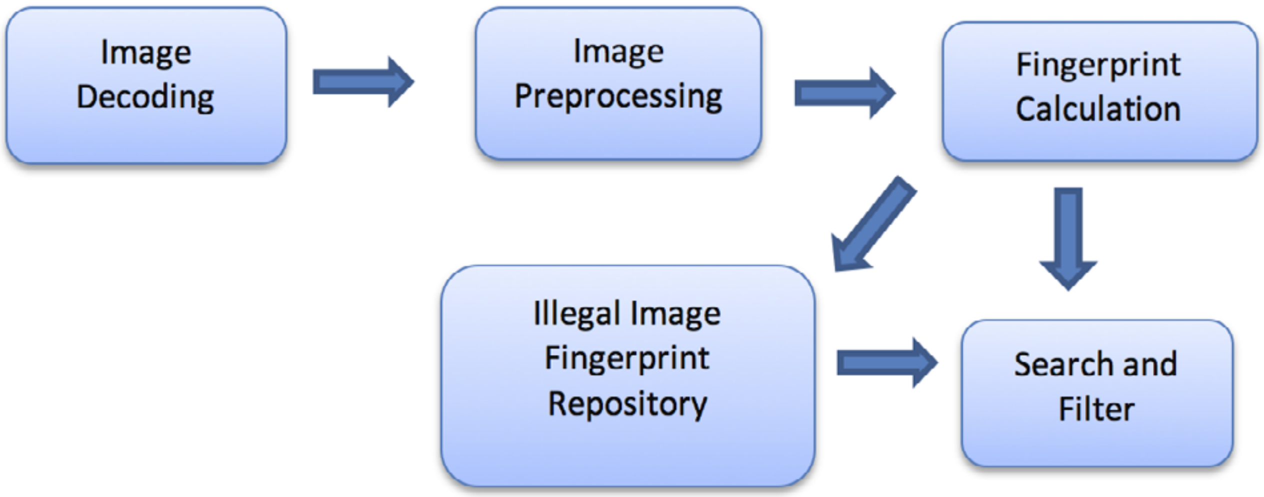
For a large Internet service provider like China's Tencent, being able to detect illegal images is important. Popular apps like WeChat*, QQ*, and QQ Album* are not just text-oriented, they also involve image generation and sharing apps. Every year, the volume of newly generated images reaches about 100 petabytes—even after image compression. Some users may try to upload illegal images (e.g., porn). They certainly don't tell the system that the image is illegal, so the system runs a check on each image to try to block them. This is a huge computing workload, with billions of images uploaded each day.

Technical Background

In our article in the last issue of Parallel Universe, **Optimizing Image Identification with Intel® Integrated Performance Primitives**, we discussed how Intel helped Tencent speed MD5 image identification by 2x. In this article, we'll explain how Intel has extended its optimization work to Tencent's illegal image filter system.

When a user uploads an image, the filter system decodes the image for preprocessing. The preprocessing uses a filter2D function to smooth and resize an image to one fixed size. This intermediate image is the input for fingerprint creation. The new fingerprint is compared to the seed fingerprint of an illegal image. If the image is illegal, it is blocked or deleted, and the fingerprint is added into the illegal image fingerprint repository.

Figure 1 shows how a fingerprint seed is added to the repository manually.

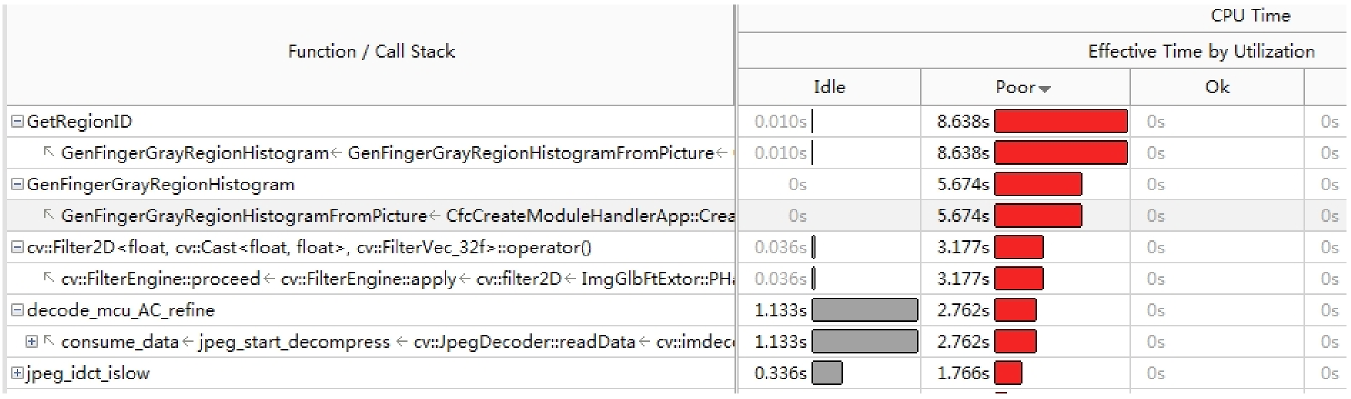


1 Illegal image filter system architecture

Optimization Overview

Working with Tencent engineers, Intel prepared the environment to tune the hotspot functions. Intel® VTune™ Amplifier is a powerful tool that can provide runtime information on code performance for developers creating serial and multithreaded applications. With this profiler information, a developer can analyze the algorithm choices and identify where and how the application can benefit from available hardware resources.

Using Intel VTune Amplifier to profile Tencent's illegal image filter system, we were able to locate the hotspot functions. As shown in **Figure 2**, we can conclude that GetRegionID, GetFingerGrayRegionHistogram, and cv::Filter2D are the top three hotspot functions.



Function / Call Stack	CPU Time			
	Effective Time by Utilization			
	Idle	Poor	Ok	
⊟GetRegionID	0.010s	8.638s	0s	0s
↖ GenFingerGrayRegionHistogram← GenFingerGrayRegionHistogramFromPicture←	0.010s	8.638s	0s	0s
⊟GenFingerGrayRegionHistogram	0s	5.674s	0s	0s
↖ GenFingerGrayRegionHistogramFromPicture← CfcCreateModuleHandlerApp::Crea	0s	5.674s	0s	0s
⊟cv::Filter2D<float, cv::Cast<float, float>, cv::FilterVec_32f>::operator()	0.036s	3.177s	0s	0s
↖ cv::FilterEngine::proceed← cv::FilterEngine::apply← cv::filter2D← ImgGlbFtExtor::PH	0.036s	3.177s	0s	0s
⊟decode_mcu_AC_refine	1.133s	2.762s	0s	0s
⊞ ↖ consume_data← jpeg_start_decompress ← cv::JpegDecoder::readData← cv::imdec	1.133s	2.762s	0s	0s
⊞jpeg_idct_islow	0.336s	1.766s	0s	0s

2 Illegal image filter system hotspot functions

We found that GetRegionID was called by GenFingerGrayRegionHistogram. The GetRegionID function included a cascaded clause ("if else"), which is hard to optimize. The GetFingerGrayRegionHistogram function can be reimplemented using **SIMD** instructions. And the filter2D function can be reimplemented by **Intel® Integrated Performance Primitives** (Intel® IPP). Both these functions achieved more than a 10x speedup. The whole system has more than doubled its speed, which helps Tencent double the performance of its system. Moreover, the latency reduction improves the usage experience when users share images and send them to friends. Since filter2D is also widely used in data processing and digital image processing, Intel's work with Tencent is a good example for other cloud service users.

Intel® Streaming SIMD Extensions and GetFingerGrayRegionHistogram Optimization

Intel introduced an instruction set extension with the Intel® Pentium® III processor called Intel® Streaming SIMD Extensions (Intel® SSE). This was a major extension that added floating-point operations over the earlier integer-only SIMD instruction set called MMX™. Since the original Intel SSE, SIMD instruction sets have been extended by wider vectors, new and extensible syntax, and rich functionality. The latest SIMD instruction, set **Intel® Advanced Vector Extensions** (Intel® AVX-512), can be found in the Intel® Core™ i7 processor.

Algorithms that significantly benefit from SIMD instructions include a wide range of applications such as image and audio/video processing, data transformation and compression, financial analytics, and 3D modeling and analysis.

The hardware that runs Tencent's illegal image filter system uses a mix of processors. Intel's optimization is based the common SSE2, which is supported by all the Tencent systems.

We investigated the top two functions: GetRegionID and GetFingerGrayRegionHistogram. GetRegionID was called by GenFingerGrayRegionHistogram. The GetRegionID function includes a cascaded clause of "if else," which, as stated earlier, is hard to optimize. But the function GetFingerGrayRegionHistogram has a big "for" loop for each image pixel, which has room for optimization. And there is a special algorithm logical when GetFingerGrayRegionHistogram calls GetRegionID. Intel rewrote part of the code to use the SIMD instruction to optimize the GetFingerGrayRegionHistogram function. **Figure 3** shows Simhash.cpp, which is the source file before optimization, and simhash11.cpp, the file after optimization.

Original Version Simhash.cpp	Optimized Version Using SIMD (simhash 11.cpp)

```
static std::string
GenFingerGrayRegionHistogram(const cv::Mat&
grayMat) {
   int histogram[9][4] = {0}; …
   for(int h = 0; h < grayMat.rows;h++)       {
   for(int w = 0 ; w < grayMat.cols; w ++)
   {  unsigned char v = grayMat.at<unsigned
char>(h, w);
int iRegionID = GetRegionID(w, h, grayMat.cols,
grayMat.rows, 3);
if(iRegionID < 0 || iRegionID >= 9)
continue;
  ...
}

static int GetRegionID(int cw, int ch, int
w, int h, int div, int skipDiv = 0/*skip
boundary*/) {
   int divWidth = w/div;
   if(0 == divWidth)           divWidth = 1;
   int wid = cw/divWidth;
   if(wid >= div)              wid = div - 1;
   int divHeight = h/div;
   if(0 == divHeight)          divHeight = 1;
   int hid = ch/divHeight;
   if(hid >= div)              hid = div - 1;
   if(wid < skipDiv || wid > div - 1 - skipDiv
         || hid < skipDiv || hid > div - 1 -
skipDiv)
   {    return -1;     }
          ...
}
```
	```
	static std::string
	GenFingerGrayRegionHistogram(const cv::Mat&
	grayMat) {
	   int histogram[9][4] = {0}; …
	   for(int h = 0; h < grayMat.rows; h++)      {
	unsigned char  const *p= grayMat.ptr<unsigned
	char>(h);
	int nextRegionCw = 0;
	   int regionSize;
	/*TODO: require more consideration for 3 in
	'grayMat.cols/3*3 if
	*  the region block schematic changed */
	for(int w = 0 ; w < grayMat.cols;
	w+=regionSize) {
	regionSize = GetNextRegionIDSize(w,h,grayMat.
	cols, grayMat.rows, 3, &iRegionID);
	…
	for(int i=0;i<(regionSize&(~0xf));i+=16)      {
	/* The region 'iRegionID fixed within this
	loop*/
	               __m128i xmm0; //tmp
	xmm0 = _mm_loadu_si128((__m128i*)(p+i+w));
	       …
	xmm1 = _mm_srli_epi16(xmm0, 6);
	xmm1 = _mm_and_si128(xmm1,xmm15);
	xmm2 = _mm_and_si128(xmm0,xmm14);
	...
	```

3 Before and after optimization

Intel tested the optimized code on an Intel® Xeon® processor E5-2680 at 2.70 GHz. The input image was a large JPEG (5,000 x 4,000 pixels). Tencent calls the function by a single thread, so we tested it on a single core. The result was a significant speedup **(Table 1)**.

Algorithm name	Latency (in second, smaller is better)	Gain
Tencent's original	0.457409	100%
SSE4.2 optimized	0.016346	2798%

Table 1. Performance data on Intel® Xeon® processor E5-2680 at 2.70 GHz

Intel® IPP and Filter2D Optimization

Intel SIMD capabilities change over time, with wider vectors, newer and more extensible syntax, and richer functionality. To reduce the effort needed on instruction-level optimization to support every processor release, Intel offers a storehouse of optimized code for all kinds of special computation tasks. The Intel IPP library is one of them. Intel IPP provides developers with ready-to-use, processor-optimized functions to accelerate image, signal, data processing, and cryptography computation tasks. It supplies rich functions for image processing, including an image filter, which was highly optimized using SIMD instructions.

Optimizing Filter2D

Filtering is a common image processing operation for edge detection, blurring, noise removal, and feature detection. General linear filters use a general rectangular kernel to filter an image. The kernel is a matrix of signed integers or single-precision real values. For each input pixel, the kernel is placed on the image in such a way that the fixed anchor cell (usually a geometric center) within the kernel coincides with the input pixel. Then it computes the accumulation of the kernel and the corresponding pixel. Intel IPP supports the variant filter operation **(Table 2)**.

IPP Filter Functionality	Notes
ippiFilter<Bilateral|Box|SumWindow>Border	Perform bilateral, box filter, sum pixel in window
ippiMedian Filter	Median filter
ippiGeneral Linear Filters	A user-specified filter
Separable Filters	FilterColumn or FilterRow function
Wiener Filters	Wiener filter
Fixed Filters	Gaussian, laplace, hipass and lowpass filter, Prewitt, Roberts and Scharr filter, etc.
Convolution	Image convolution

Table 2. Variant filter operation

For several border types (e.g., constant border or replicated border), Tencent used **OpenCV*** 2.4.10 to preprocess images. Using Intel VTune Amplifier XE, we found that the No. 3 hotspot is cv::Filter2D. It was convenient and easy to use the Intel IPP filter function to replace the OpenCV function.

The test file filter2D_demo.cpp is from OpenCV Version: 3.1:

- @brief Sample code that shows how to implement your own linear filters by using the filter2D function
- @author OpenCV team

We used the IPP filter function from Intel® IPP 9.0, update 1, as shown in **Figure 4**.

filter2D OpenCV* code

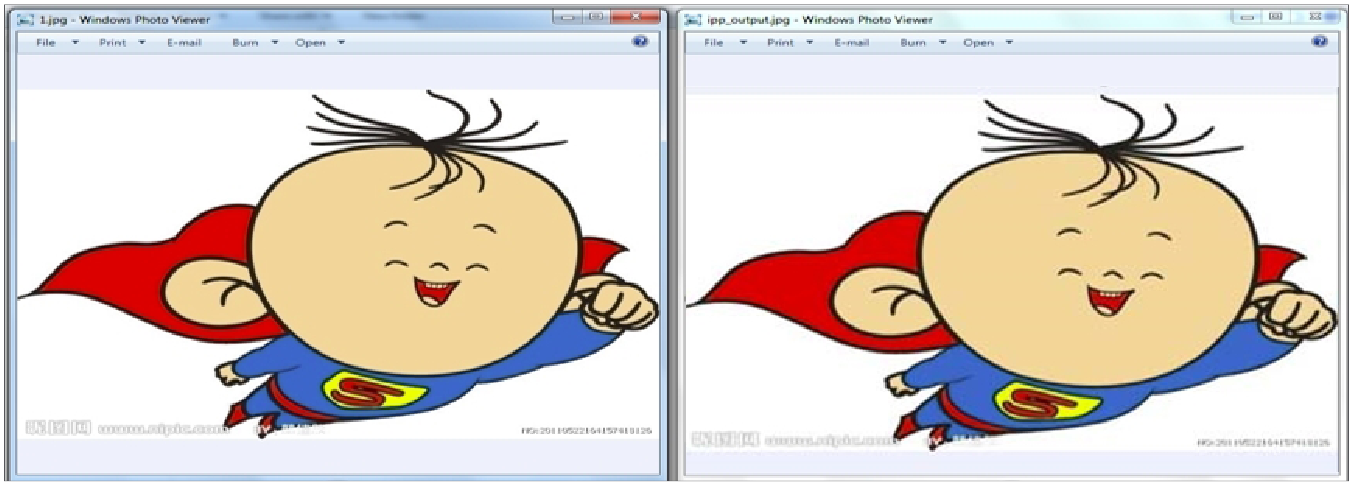
```
filter2D(src, dst, ddepth , kernel, anchor,
delta, BORDER_REPLICATE );
  /// Initialize arguments for the filter
  anchor = Point( -1, -1 );
  …
  /// Update kernel size for a normalized box
filter
  kernel_size = 3 + 2*( ind%5 );      // 3, 5,
7, 9, 11
  kernel = Mat::ones( kernel_size, kernel_size,
CV_32F )/ (float)(kernel_size*kernel_size);

  struct timeval start, finish;
  gettimeofday(&start, NULL);
  /// Apply filter
  filter2D(src, dst, ddepth , kernel, anchor,
delta, BORDER_REPLICATE );
  gettimeofday(&finish, NULL);
  double duration_1=(finish.tv_sec-start.tv_
sec)*1000+(finish.tv_usec-start.tv_usec)/1000;
  printf("opencv cost time: %f ms\n",
duration_1);

  imwrite("./opencv_output.jpg", dst);
}
```

IPP Filter2D

```
 ipp_filter2D(src, dst, ddepth , kernel,
anchor, delta, BORDER_REPLICATE );
typedef IppStatus (CV_STDCALL *
ippiFilterBorder)(const void * pSrc, int
srcStep, void * pDst, int dstStep, IppiSize
dstRoiSize,
        IppiBorderType border, const void *
borderValue,
        const IppiFilterBorderSpec* pSpec,
Ipp8u* pBuffer);

    int stype = src.type(), sdepth = CV_MAT_
DEPTH(stype), cn = CV_MAT_CN(stype), ktype =
kernel.type(), kdepth = CV_MAT_DEPTH(ktype);

    Point ippAnchor((kernel.cols-1)/2, (kernel.
rows-1)/2);
  int borderTypeNI = borderType & ~BORDER_
ISOLATED;
    IppiBorderType ippBorderType =
ippiGetBorderType(borderTypeNI);       /*
convert to IPP BorderType  */

  /* find Filter */
  ippiFilterBorder ippFunc =
     stype == CV_8UC1 ? (ippiFilterBorder)
ippiFilterBorder_8u_C1R :
    …
ippiFilterBorderInit_32f(pKerBuffer,
kernelSize, dataType, cn, ippRndFinancial,
spec);
status = ippFunc(src.data, (int)src.step, dst.
data, (int)dst.step, dstRoiSize, ippBorderType,
borderValue, spec, buffer);
```

4 Code before (left) and after (right) after accelerating the Filter2D by Intel® IPP

We used the same filter kernel to filter the original image, and then recompiled the test code and OpenCV code with GCC and benchmarked the result on Intel Xeon processor-based systems. The effects of the image filter are the same from Intel IPP code to OpenCV code. **Figure 5** shows the Intel IPP smooth image (right) compared to the original image.


1.jpg - Windows Photo Viewer
File Print E-mail Burn Open
ipp_output.jpg - Windows Photo Viewer
File Print E-mail Burn Open

5 Image before (left) and after filtering

The consumption times **(Table 3)** show that the ipp_filter2D take less time (about 9ms), while the OpenCV source code takes about 143ms. The IPP filter2D is 15x faster than the OpenCV plain code.

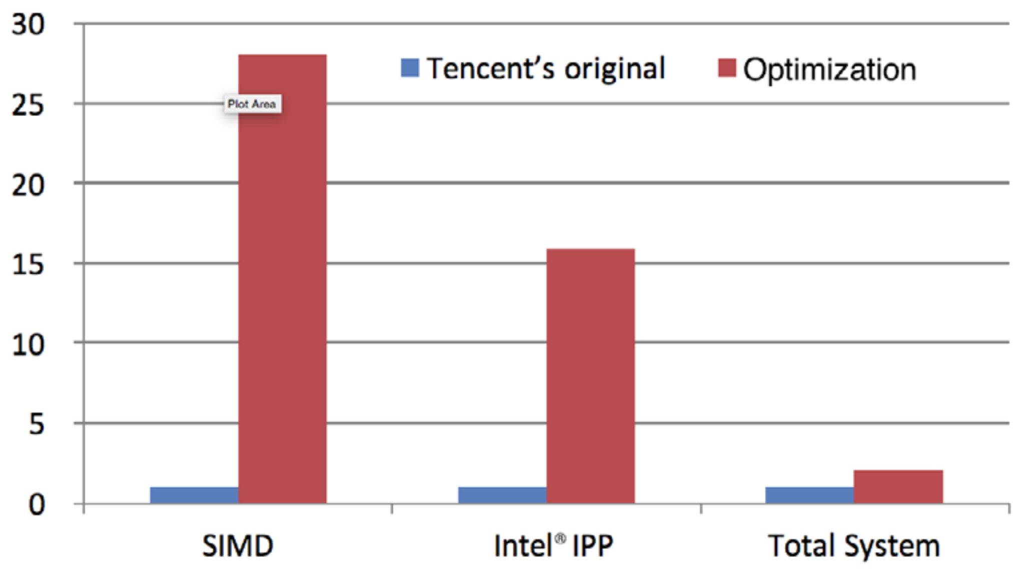
CPU	Program	Time (ms)
Intel® Xeon® Processor E5-2680 v3	ipp_filter2D	9ms
	OpenCV_filter2D	143ms

Table 3. Consumption times

Note that OpenCV has Intel IPP integrated during compilation; thus, some functions can call underlying Intel IPP functions automatically. In some OpenCV versions, the filter2D function can at least call the IPP filter2D. But, considering the overhead of function calls, we selected to call the IPP function directly.

Summary

With manual modifications to the algorithm, plus SSE instruction optimization and Intel IPP filter replacement, Tencent reports that the performance of its illegal image filter system has more than doubled compared to the previous implementation in its test machine. **Figure 6** shows the total results.



6 Performance of Tencent's illegal image filter system before and after optimization

Tencent is a leading Internet service provider with high-performance requirements for all kinds of computing tasks. Intel helped Tencent optimize the top three hotspot functions in its illegal image filter system. By manually implementing the fingerprint generation with SIMD instructions, and replacing the filter2D function with a call into the Intel IPP library, Tencent was able to speed up both hotspots by more than 10x. As a result, the entire illegal image filter system has more than doubled its speed. This will help Tencent double the capability of its systems. Moreover, the latency reduction improves the user experience for customers sharing and sending images.

Software

THE PARALLEL UNIVERSE