FPGA Programming with the OpenCL™ Platform

Accelerating the Eigen* Math Library for Automated Driving Workloads

Speeding Algebra Computations with the Intel® Math Kernel Library Vectorized Compact Matrix Functions
CONTENTS

Letter from the Editor
Happy New Year, Happy Parallel Computing
   by Henry A. Gabb, Senior Principal Engineer, Intel Corporation

FPGA Programming with the OpenCL™ Platform
Knowing How to Program an FPGA is a Skill you Need—and Here’s How to Start

Accelerating the Eigen* Math Library for Automated Driving Workloads
Meeting the Need for Speed with Intel® Math Kernel Library

Speeding Algebra Computations with the Intel® Math Kernel Library
Vectorized Compact Matrix Functions
Maximizing the Performance Benefits of the Compact Data Layout

Boosting Java* Performance in Big Data Applications
How New Enhancements Enable Faster and Better Numerical Computing

Gaining Performance Insights Using the Intel® Advisor Python* API
Getting Good Data to Make Code Tuning Decisions

Welcome to the Intel® AI Academy
AI Education for All
Happy New Year, Happy Parallel Computing

Welcome to our first issue of 2018. I thought about making some predictions about hardware and software trends, but I’m more of a fast follower than a computing visionary. After all, my academic background is biotechnology and data science—not computer science. In the last issue, I made a not-so-bold prediction about the heterogeneous parallel computing future. I’ve been worrying about heterogeneous parallelism since 2009, and real visionaries were worrying about it years before that. It’s not a prediction when there’s evidence of a trend all around you, but FPGAs are the next phase in this evolution toward heterogeneity.

It won’t be long before FPGAs are as ubiquitous as multicore processors—but few of us can effectively program them. So we welcome back James Reinders, the founding editor of The Parallel Universe, to tell us more about FPGA programming. In the last issue, we discussed FPGA programming from a software development perspective. This time, James and Tom Hill from Intel’s Programmable Logic Group give a detailed, practical guide to help you get started with FPGA Programming with the OpenCL™ Platform. The OpenCL (Open Computing Language) platform is an “open standard for parallel programming of heterogeneous systems.” For many of us, our first foray into FPGA programming is likely to be with the OpenCL platform.

In 2018, we’ll continue to cover programming tools and programming models. Two articles in this issue discuss new features in Intel® Software Development Tools: Speeding Algebra Computations with the Intel® Math Kernel Library Vectorized Compact Matrix Functions and Gaining Performance Insights Using the Intel® Advisor Python API. The former describes a new data layout designed to improve the performance of applications that compute over large groups of small matrices. The latter explores a new API to directly access the Intel Advisor database to do custom analytics or create custom visualizations of your application’s performance.
Java* is one of the most popular programming languages in the world, but we don’t often discuss it in this magazine. That’s going to change in 2018. Intel® Parallel Studio XE is improving its Java tuning support and the Java JVM is improving its support for vector computation. Boosting Java Performance in Big Data Applications describes the latter enhancements.

Artificial intelligence (AI) isn’t going away in 2018. Autonomous driving is becoming a reality because of advances in AI, but it requires high-performance computing. Accelerating the Eigen* Math Library for Automated Driving Workloads shows how to improve the performance of an important computational kernel. Finally, Welcome to the Intel® AI Academy gives an overview of Intel’s new, comprehensive program for AI education, tools, and technology.

Future issues of The Parallel Universe will bring you articles on a wide range of topics, including the effective use of new non-volatile memory, tuning code for non-uniform memory access (NUMA) architectures, best practices for productivity languages like Python* and R (maybe even some articles about Go* and Julia*), and much more.

We’re looking forward to another year of exploring the exciting future of software development with you.

Henry A. Gabb
January 2018
Why FPGAs?

Field-programmable gate arrays (FPGAs) are exciting because they offer high performance with low latency and power efficiency. These benefits are realized through their massive parallel capabilities coupled with reconfigurability. An FPGA provides a reconfigurable sea of gates on which anyone can:

- **Design** a custom hardware accelerator
- **Deploy** it for a single application
- **Quickly reconfigure** the device as a new accelerator for a different application
This article explains how you can make your own designs and use them to accelerate your applications. Hardware engineers have long used FPGAs in place of ASICs in a variety of applications. Historically, configuring them (programming) has been done with high-level definition languages like Verilog* or VHDL*. These design methodologies are familiar and useful for hardware engineers but completely foreign to software developers.

New tools centered on the OpenCL™ platform bridge this gap to bring the benefits of FPGA hardware platforms to software developers. Using these new tools, we can view FPGAs as highly configurable devices that can transform custom algorithms, written in a C-like syntax, into fast and power-efficient hardware. The flexibility and performance of these solutions are significant, thanks to the latest generation of FPGA devices.

**Massive Parallelism**

An Intel® Stratix® 10 FPGA can offer as much as 10 TFLOPs per second of single-precision performance. Also, it's not limited to standard number formats. For instance, implementing a 2-bit deep neural network is not only possible, it's easy thanks to Intel's upcoming Deep Learning Accelerator configurable workload that was demoed at Supercomputing in November 2017.

Intel FPGAs offer massive amounts of parallelism, with hardware flexibility to reduce latency (Figure 1). Data movement is optimized because the FPGAs are naturally programmed in a dataflow—or pipelined—way, leading to reduced data transfers between the host CPU and FPGAs.

**Sounds Great…So What’s the Catch?**

Today, nothing prevents C programmers from following along and trying their hands at FPGA programming. As with all programming specialties, there's a learning curve when programming FPGAs using the OpenCL platform, including new development environments and new methods. Fortunately, both are addressed through online training and OpenCL platform coding guides that make FPGA programming very approachable.

**FPGA Development Environments**

OpenCL platform software development environments are based on Eclipse* or Visual Studio*, so they're already familiar to C programmers. In this article, we'll describe ways to write a first program in the OpenCL platform and run it on an FPGA.
OpenCL platform coming of age for FPGAs: Competitive results using highly productive OpenCL platform FPGA programming versus three other traditional, but less productive, approaches. (See “Gzip on a Chip” in the Learn More and Explore section at the end of this article.)

FPGA Optimizations

For any new device and programming method, there are tips and tricks to achieve top performance. These optimizations may range from familiar approaches, such as data structure organization and clean thinking on algorithms, to the most effective ways to express pipelining with the OpenCL platform. In less common situations, we can use hardware description language programming (Verilog* or VHDL*) similar to how we might use inline assembly for special needs on a CPU. FPGA optimization efforts end up feeling different—but no better or worse—than optimizing for a CPU or a GPU.

Similar to CPUs or GPUs, FPGA programmers can use libraries written for FPGAs (which hardware engineers commonly refer to as IP blocks). (The Intel® FPGA SDK for OpenCL™ Best Practices Guide is an excellent follow-on to this article for exploring optimization for FPGAs in more depth.)
Three Approaches to Trying Out FPGA Programming Now

We'll detail three ways to try out FPGA programming:

- **Emulation.** We can develop and debug using ways to try out FPGA programming without having to purchase FPGA hardware. This is always the recommended first step for developing and debugging a program. The SDK runs on Windows* or Linux*.

- **Offline compile.** After debugging our emulated compilation, we can use the OpenCL platform environment, without an FPGA, to do a compile that will generate the FPGA programming file. This can provide accurate area and performance reports for the OpenCL platform kernel, helping us refine our code before actually running it on an FPGA.

- **Cloud.** We can use tools and FPGAs on Nimbix® systems in the cloud, from any browser on any system, to do all development from emulation through running on an FPGA. The current configurations include CPU-based machines with just the Intel® tools, as well as machines with the tools and an Arria® 10 FPGA. Cost starts at $0.36 per hour for program development and $3.00 per hour for using an FPGA-equipped machine. Nimbix systems run Linux.

We could also use an FPGA development board as our target platform. This gives access to the full power of the most advanced FPGAs. The cost for a single PCIe card is approximately $5,000 if equipped with an Intel® Arria® 10 FPGA or $11,000 if equipped with an Intel Stratix 10 FPGA. Technically, we could try an Intel® Cyclone®-based FPGA development board with the OpenCL platform. The DE1-SoC Board might seem tempting, and it’s only $250. Unfortunately, as shown in Table 1, the part is generally uninteresting for computational acceleration. To experiment with the OpenCL platform on an FPGA for computational acceleration for under $250, we recommend a combination of options 1, 2, and 3 above. Despite our recommendation, we’ve used the Intel OpenCL SDK with the DE1-SoC Board and it does work quite well. It simply lacks compute power.

Table 1. Some of the Intel FPGA Platforms with OpenCL platform support

<table>
<thead>
<tr>
<th>Platform</th>
<th>Part Number</th>
<th>Max. Logic Elements (Millions)</th>
<th>Max. Embedded Memory (Mb)</th>
<th>Max. Peak TFLOP/s (SP, IEEE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclone V SoC development board</td>
<td>DE1-SoC Board (features a Cyclone V - 5CSEMA5F31C6N)</td>
<td>0.085</td>
<td>3.9</td>
<td>Under 0.001</td>
</tr>
<tr>
<td>Arria® 10 Dev kit</td>
<td>DK-DEV-10AX1155-A</td>
<td>1.1</td>
<td>53</td>
<td>1.4</td>
</tr>
<tr>
<td>Stratix® 10 Dev Kit</td>
<td>DK-DEV-1SGX-H-0A</td>
<td>5.5</td>
<td>229</td>
<td>9.2</td>
</tr>
<tr>
<td>Nimbix Cloud</td>
<td>Uses a BittWare A10PL4 board with an Intel Arria 10 model GX 1150.</td>
<td>1.1</td>
<td>53</td>
<td>1.4</td>
</tr>
</tbody>
</table>
The Intel Arria and Stratix families are aimed at application acceleration, while the Intel Cyclone family is aimed at low-power embedded usage. The OpenCL platform is well-supported on both Intel Arria and Intel Stratix. Plus, there are some Intel Cyclone V SoC boards supported. No matter what board we choose, we'll need a board support package (BSP), which the OpenCL SDK uses to properly target a specific system configuration.

**Why the OpenCL Platform?**

The OpenCL platform is a standard that extends C with a programming model for accelerating algorithms on heterogeneous systems. Central to the OpenCL platform, like CUDA®, is the concept of a kernel running on a device (hardware accelerator). Kernels are written in a subset of C99®. Unlike CUDA, the OpenCL platform is a portable, open, royalty-free standard designed to support virtually any compute device including CPUs, GPUs, and FPGAs.

The OpenCL platform also includes an application program interface (API) for the host (CPU) to communicate with the device (an FPGA in our case), traditionally over PCIe, or for one kernel to communicate with another without host interaction. In addition to this, Intel provides an I/O Channel API to stream data into a kernel directly from a streaming I/O interface such as 10Gb Ethernet®.

In the OpenCL platform programming model, the user schedules tasks to command queues. Each device has at least one command queue. The OpenCL platform runtime then breaks the data-parallel tasks into pieces and sends them to the processing elements in the device. This is the way a host communicates with any device. It’s up to the individual device vendors to abstract away the vendor-specific implementation. The Intel® FPGA SDK for OpenCL does this and conforms to the OpenCL 1.2 standard.

**The Intel® FPGA SDK for OpenCL**

The Intel FPGA SDK for OpenCL allows a user to abstract away the traditional hardware FPGA development flow for much faster and higher-level software development. We can emulate OpenCL platform accelerator code on an x86-based host in seconds, and get a detailed optimization report with specific algorithm pipeline dependency information. This allows us to debug our program before we wait for the longer compile time needed to generate the final code to run on an FPGA.
The Intel FPGA SDK for OpenCL facilitates development by abstracting away the complexities of FPGA design, allowing software programmers to write hardware-accelerated kernel functions in OpenCL C*, an ANSI C-based language with additional OpenCL platform constructs.

As part of the SDK, Intel provides a suite of tools to further resemble the fast development flow of software programmers, including:

- **An emulator** to step through the code on an x86 CPU and ensure that it's functionally correct
- **A detailed optimization report** to help you understand the load and store inner loop dependencies
- **A profiler** that provides performance insight into the kernel to ensure proper memory coalescence and stall-free hardware pipelines
- **An OpenCL platform compiler** capable of performing over 300 optimizations on the kernel code and producing the entire FPGA image in one step

The Intel FPGA SDK for OpenCL supports a variety of host CPUs, including Intel® Xeon® processors, the embedded ARM Cortex-A9* processor cores in SoCs, and IBM Power Systems* processors. It supports scalable solutions on multiple FPGAs and multiple boards, as well as a variety of memory targets such as DDR SDRAM for sequential memory accesses, QDR SRAM for random memory accesses, or internal FPGA memory for low-latency memory access. Half-precision, as well as single- and double-precision, floating-point computation is also supported.

**Trying FPGA Programming**

Let’s explore the three methods for trying FPGA programming.

**Emulation**

The emulation method doesn't require an FPGA. When writing an OpenCL program for an FPGA, emulation is a crucial first step that allows us to efficiently debug OpenCL platform code. Because compiling for an FPGA is much slower, and debug options are more limited, it's highly preferable to debug as much as possible using emulation before moving to an FPGA.

Let’s consider a simple vector add to illustrate these three methods for trying out FPGA programming. We downloaded and installed the Intel FPGA SDK for OpenCL, and we downloaded the vector add example from the Intel FPGA software design center. After we download and unpack the .zip file for Windows, or the .tar file for Linux, there is a README.html with additional instructions. The SDK and the example code...
are free to download, install, and use. We were able to do this on Windows and Linux—plus, we found the
OpenCL SDK to be very effective on a MacOS* system running Linux inside a VBox*.

It’s important to avoid spaces in the path to our code. We ran into several issues which seemed to occur
because the tools were not prepared to handle spaces in the path. There are mentions of this in some of the
documentation for the FPGA tools, but not in places we’d be likely to read if we were only interested in the
OpenCL platform. (Work is underway to remove these limitations in the future.)

There are three steps in the emulation method:

1. Build the FPGA code.
2. Build the host code.
3. Run the program.

**Build the FPGA Code**
The FPGA code consists of kernels written in the OpenCL platform. We specify that we want to use
emulation with the `-march=emulator` option on the Altera OpenCL* compiler (aoc*). If we forget this
option, the compile will take several hours because it will do the full synthesis, place, and route for the
FPGA. With `-march=emulator`, the compile takes mere seconds and simply creates x86 code so we can
run our program without an FPGA. Since our goal is emulation, we’ll use this option. From the `vector_add`
directory, we followed the README.html directions and used the command:

```
aoc -march=emulator device/vector_add.cl -o bin/vector_add.aocx
```

**Build the Host Code**
The host code is written in C with calls to the OpenCL API to setup the FPGA, load kernels, and use them on
the FPGA. On Windows, the instructions have us use Microsoft Visual Studio* (the free Community version
worked for us). On Linux, a Makefile lets us do a simple “make” which uses the GNU C++ compiler.

**Run the Program**
Here, it’s important to follow the README.html instructions to set CL_CONTEXT_EMULATORDEVICE_INTELFPAGA=1
so the host program expects to run emulated FPGA code. The instructions are good, but we offer two
additional tips.
First, before doing any steps, initialize the OpenCL SDK variables:

- **Windows:** `%INTELFPGAOCLSDKROOT%\init_opencl.bat` (on our system, this was `C:\intelFPGA\17.1\hld\init_opencl.bat`).
- **Linux:** `$INTELFPGA_ROOTDIR/hld/init_opencl.sh` (on our system, this was `/opt/intelFPGA_pr/17.1/hld/init_opencl.sh`).

On Windows, also initialize the Visual Studio variables to make sure everything is done consistently for 64 bits. (We had compile failures on our system when we failed to do this.)

- Find and run `vcvars64.bat` (on our system, this was `C:\Program Files (x86)\Microsoft Visual Studio\Shared\14.0\VC\bin\amd64\vcvars64.bat`).

### Offline Compile

The OpenCL platform environment can be used to perform a compile that will generate the FPGA programming file and reports. These reports are created automatically by the `aoc` command. There is also a `-fast-compile` option which creates the programming file output and the report much faster, with some loss of performance (i.e., 10 to 20 percent) for the FPGA. These reports are useful in optimizing a kernel, and they are statically created during compilation—meaning there’s no need to actually run the program on an FPGA to get this output. However, we do need to compile for a specific FPGA (hence the `-march=emulator` option will not produce reports). From the `vector_add` directory, we used the same command as we did when building the FPGA code:

```bash
aoc -march=emulator device/vector_add.cl -o bin/vector_add.aocx
```

Within seconds, the compiler prints out a utilization report:
These resource utilizations give us an idea of how well our OpenCL platform-based program is using the parallelism available on our FPGA. This can be valuable tuning information and it’s available without having to wait for a long compilation. If we use too many resources, the program might not fit on a particular FPGA. But, even more commonly, if we’re using too few, we may have opportunities to tune our code to get even more performance. It’s not hard to understand that if we’re using only 10 percent of the devices (e.g., the floating-point capabilities), we’re not getting maximum performance.

More detailed reports are available after the full compile is done. These reports are more accurate, so they’re important for fine tuning, but they take longer to create. This is where the fast compile option is very helpful, even without an FPGA, because the reports come out faster (i.e., in a quarter of the time). (The Intel® FPGA SDK for OpenCL™ Best Practices Guide from Intel does a good job of explaining how to use the reports.)

Multiple results, which previously had to be done manually, are now done automatically by the compiler and assembled so that we can view them in a browser. We start by finding the report.html file created during compilation (Figure 2). (The previous utilization report shows where the file is located.)
The resulting page has multiple reports we can browse through. Figure 3 shows a summary for the vector add example. It’s interesting that this detailed report, which came after nearly an hour of compiling, confirms the utilization estimates reported in the first few seconds of the compilation (9 percent ALUTs used, 9 percent dedicated logic registers, 9 percent of the memory blocks, and none of the DSPs). Figure 4 shows an area analysis report, which allows us to drill down into the program to understand the relationship of our source code and resource utilization.
Area analysis of system report from compilation using `-fast-compile`

A final useful tip: the "aocl env aocxfilename.aocx" command can give interesting information detailing how the aocx file was compiled (more details than shown in the report.html summary).
Cloud
The cloud method doesn't require us to have our own FPGAs, nor do we have to download or setup any software. Instead, we setup an account with Nimbix and pay based on usage. We had success using this approach from every browser we tried, on MacOS, Windows, and Linux hosts. With rates starting at $0.36 per hour, prorated to the second, we didn't run up significant charges to try out the OpenCL platform and saved a tremendous amount of time by not having to do hardware or software setup. In all, we spent less than $5 to try the vector add example on Nimbix.

Unlike other FPGA-in-the-cloud-systems, Nimbix and Intel did not create special versions of their tools to enable cloud work. This means we can use the same free tools from methods 1 and 2 to build binaries that run in the cloud if we want to avoid fees for building the FPGA code. It might feel more natural to develop, debug, and build locally until we're ready to run on an FPGA. Likewise, the cloud has the latest tools, with the bonus that compilations will also work on non-cloud systems if we have the same board as those used in the cloud.

Because the Nimbix folks set up everything in advance, there's no need to install the OpenCL platform tools—or even to source the init_opencl script. But we do need to get a Nimbix account, activate an instance, and download the example code before we can follow the instructions to run the example.

Here are the steps we took to utilize the Nimbix cloud:

- Sign up for an account using this link. Fill out only the customer and billing Information. Do not select PushToCompute. The total should be $0.00 when we submit.
- Open up the Quick Start Guide for Intel FPGA Development Tools on the Nimbix Cloud published by Intel. This contains a lot of useful information, but also has a lot of things we don't need to know to use the OpenCL platform. We recommend reading it and making a special note of the fact that we get storage at /data on the machine, which can be read and written without activating a machine (see the section on Persistent Storage and Data Transference).

We recommend starting by following the emulation instructions presented earlier in this article, but do it in the cloud. Here's how:
Visit https://www.nimbix.net, click “Login,” and log onto the system. Click “Compute” (menu on the right side), and enter “Quartus” in the search field to narrow to the machines to consider using. Select the latest Quartus version available (17.1 for us) and “PAYG”.

Next, select “Desktop,” then select the CPU/memory desired. For the example, two CPUs worked fine for emulation compiles and runs. We recommend the eight-CPU machine, primarily because of the extra memory, to do a non-emulation build.

When the machine is ready, click to connect, which gives you access to use the machine. Please note that charges continue until you terminate the machine (which you can do by clicking the “N” in the lower left corner, selecting “Log Out,” and then selecting “Shutdown”). Note that disconnecting a terminal window does not terminate the machine running, so charges will continue. We prefer to kill the machine from the dashboard (https://www.nimbix.net) because it shows if anything is active and verifies that all activity has stopped.

In the machine, right-click to bring up a menu and select to open a terminal window.

We recommend working in the /data directory so that work is preserved. Therefore, we start in the terminal window with the command:

```
cd /data
```

Visit the OpenCL Vector Addition Design Example webpage and find the link for the tar file (.tgz). We like to use `wget` to fetch it:

```
```

Untar it:

```
tar xvf exm_opencl_vector_add_x64_linux.tgz
```

README.html explains the rest. We need only three simple commands to build the FPGA code, build the host code, and then run it (these compilations and run collectively take well under half a minute):

```
cd vector_add
```

```
aoc -march=emulator device/vector_add.cl -o bin/vector_add.aocx
make
```
Using the `-fast-compile` option, the 17.1 compiler built the aocx file in under an hour, using this command:

```
aocl -fast-compile device/vector_add.cl -o bin/vector_add.aocx
```

- **Run.** We need to run on a machine equipped with an FPGA. Since Nimbix uses the BittWare® A10PL4 boards with Intel Arria 10 FPGAs, we start by selecting “Application” with the A10PL4 name in it. We need to be sure that the machine type selected actually has access to an A10PL4 board. To do this, select a machine type in the pulldown that is not simply “[CPU only]” but also includes A10PL4 (Figure 5). Access to at least one such FPGA-enabled machine should be available.

5 Example of selecting an FPGA-enabled machine from the pulldown menu of machine types
- Run the application using the command (without setting `CL_CONTEXT_EMULATOR_DEVICE_ALTERA`):

  ```
  bin/host -n=10000
  ```

Congratulations should be in order again. A successful run should produce something like this:

```
Initializing OpenCL
Platform: Intel(R) FPGA SDK for OpenCL(TM)
Using 1 device(s)
a10pl4_dd4gb_gx115 : A10PL4 (acla10pl40)
Using AOCX: vector_add.aocx
Reprogramming device [0] with handle 1
Launching for device 0 (10000 elements)
Time: 0.409 ms
Kernel time (device 0): 0.062 ms
Verification: PASS
```

The `aocl diagnose` command is useful to query the available card(s) in a system:

```
aocl diagnose
aocl diagnose: Running diagnose from /opt/intelFPGA_pro/17.0/hld/board/a10pl4/linux64/libexec
------------------------- acl0 -------------------------
Vendor: BittWare Inc

Phys Dev Name Status Information
acla10pl40 Passed A10PL4 (acla10pl40)
PCle dev_id = 2494, bus:slot.func = 02:00.00, Gen3 x8
FPGA temperature = 52.6523 degrees C.
DIAGNOSTIC_PASSED
```

For more complete information about compiler optimizations, see our Optimization Notice.
The new `-fast-compile` option offers substantially faster compiles when using a 17.1 build from December 2017 or later. This is a unique Intel advantage for the dreaded slow compilation of FPGAs. The `-fast-compile` option results in a slightly less efficient FPGA program (normally 10 to 20 percent reduction) in exchange for much faster compiles (often four to five times faster). For designs that consist of multiple kernels, an incremental compile option is available that will automatically identify the kernels that have changed and only recompile the changes.

**The OpenCL Platform: High-Performance, Open, and Well Supported**

We've barely scratched the surface of showing how powerful OpenCL programming is for FPGAs. For instance, the Intel OpenCL platform supports hooks for profiling actual runs on the FPGA and studying the results. Also, the Intel OpenCL platform has extensive support for FPGA performance, including host and I/O channels. Channels are an extension to the OpenCL language, created by Intel to support data streaming directly from the I/O or host to the FPGA kernels.

We hope this article gets you started exploring the new world of programming FPGAs with the OpenCL platform. There are many reports, tools, libraries, and extensions on the Intel and Altera websites, including free training resources.

**Learn More and Explore**

- Intel® FPGA SDK for OpenCL™
- Documentation for Intel® FPGA SDK for OpenCL™
- The Intel® FPGA SDK for OpenCL™ Best Practices Guide is an excellent place to learn more about how to think about FPGAs when writing in the OpenCL platform, and it gives good information on reports and what they mean.
- Quick Start Guide for Intel FPGA Development Tools on the Nimbix Cloud
- Intel FPGA Software Design Center is where to download code examples.
- Intel FPGA platforms has links to accelerator boards.
- Official OpenCL platform standards information
- Intel Altera product information on the Intel Cyclone 10 LP, Arria 10, and Stratix 10 families
- Paper evaluating gzip acceleration via OpenCL + FPGA from the International Workshop on OpenCL 2014: Gzip on a Chip: High Performance Lossless Data Compression on FPGAs using OpenCL
Automated driving workloads include several matrix operations at their core. Sensor fusion and localization algorithms—such as different versions of the Kalman filter—are critical components in the automated driving software pipeline. The Intel® Math Kernel Library (Intel® MKL) is a powerhouse of tuned subprograms for numerous math operations, including a fast DGEMM. The automated driving developer community typically uses Eigen®, a C++ math library, for matrix operations. In addition to Intel MKL, LIBXSMM®, a highly-tuned library for high-performance matrix-matrix multiplications, shows potential to speed up matrix operations. In this article, we investigate and improve the performance of native Eigen on matrix multiplication benchmarks and the extended Kalman filter (EKF) by using Intel MKL and LIBXSMM with GNU® and Intel® compilers on the Intel® Xeon® processor.
The Need for Speed in Kalman Filtering

The automated driving pipeline includes a series of computational blocks, starting with perception, which acquires information on the driving environment from sensors such as cameras, RADARs and LIDARs, sensor fusion and localization, path planning, and finally actuation of vehicle controls such as steering angle and throttle. Performance optimizations across the entire software pipeline are crucial for meeting strict end-to-end latency requirements. Each component of the pipeline is typically assigned a tight latency budget that needs to be met almost 100 percent of the time. In this study, we focus on speeding up the EKF, an important component of sensor fusion and localization.

Extended Kalman Filter Algorithm

The EKF is a simple—yet extremely powerful—algorithm that makes predictions about the state of the vehicle (e.g., Cartesian position coordinates, velocities, yaw angle). The EKF has two consecutive steps over several iterations:

- **The prediction step** estimates values of current variables and their uncertainties based on motion models, including changes in values over time.
- **The update step** occurs when the next set of measurements is received from the sensors. This phase updates the predicted estimates based on one important factor—the weighted average of the predicted estimate and the estimate from the current measurement. Higher weights imply lower uncertainty.

In particular, this algorithm predicts the position of the vehicle \((p_x, p_y)\) and its velocity \((v_x, v_y)\) from noisy LIDAR and RADAR sensor measurements. The coupled estimate of the vehicle’s position from fusing both RADAR and LIDAR has higher accuracy than using noisy LIDAR and RADAR by themselves. LIDAR measurements that localize an object are defined in Cartesian coordinate form—\((p_x, p_y)\). RADAR measurements are typically in polar coordinate form and can be converted to Cartesian coordinates, forming measurements that are at a lower resolution than those from LIDAR.

Table 1 shows the vectors and matrices the EKF uses to represent different states and estimates.
Table 1. Matrices and vectors used by EKF

<table>
<thead>
<tr>
<th>Matrix/Vector</th>
<th>Purpose</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Control vector</td>
<td>4x1</td>
</tr>
<tr>
<td>X</td>
<td>State vector</td>
<td>4x1</td>
</tr>
<tr>
<td>Z</td>
<td>Observation vector</td>
<td>2x1 or 3x1</td>
</tr>
<tr>
<td>F</td>
<td>State transition matrix</td>
<td>4x4</td>
</tr>
<tr>
<td>P</td>
<td>Predicted covariance estimate</td>
<td>4x4</td>
</tr>
<tr>
<td>Q</td>
<td>Process (Gaussian) noise covariance matrix</td>
<td>2x3, 2x2, 3x3, 3x2</td>
</tr>
<tr>
<td>R</td>
<td>Measurement (Gaussian) noise covariance matrix</td>
<td>2x3, 2x2, 3x3, 3x2</td>
</tr>
<tr>
<td>H</td>
<td>Observation matrix</td>
<td>2x4 or 3x4</td>
</tr>
</tbody>
</table>

Predict

\[ x' = F \ast x + u \quad \text{Predicted state estimate} \]
\[ P' = F \ast P \ast F^T + Q \quad \text{Predicted covariance estimate} \]

Measurement Update

\[ y = z - H' \ast x \quad \text{Innovation or measurement residual} \]
\[ S = H \ast P' \ast H^T + R \quad \text{Innovation (or residual) covariance} \]
\[ K = P' \ast H^T \ast S^{-1} \quad \text{Near-optimal Kalman gain} \]
\[ x = x' + K \ast y \quad \text{Updated state estimate} \]
\[ P = (I - K \ast H) \ast P' \quad \text{Updated covariance estimate} \]

Important Math Libraries

Intel MKL provides highly optimized, threaded, and vectorized math functions that maximize performance on Intel® processor architectures. It is compatible across many different compilers, languages, operating systems, linking, and threading models. Important for our purposes, it provides a highly-tuned DGEMM function for matrix-matrix multiplication. To eliminate overhead from additional error checking for DGEMM on small matrices, Intel MKL provides the –DMKL_DIRECT_CALL compiler flag to guarantee that the fastest code path is used at runtime.

Eigen is an open-source, easy-to-use C++ library that provides operations ranging from matrix math to geometry algorithms. It enables vectorization across different levels of SSE and AVX. Eigen can take advantage of Intel MKL through the –DEIGEN_USE_MKL_ALL flag.

For more complete information about compiler optimizations, see our Optimization Notice.
LIBXSMM is an open-source, high-performance library tuned for fast matrix-matrix multiplication on very small matrix sizes\textsuperscript{2,3}. LIBXSMM generates just-in-time (JIT) code for small matrix-matrix multiplication kernels for various instruction sets including SSE, AVX, AVX2, and AVX512. LIBXSMM is best suited for matrices where $(M \times N \times K)^{1/3}$ is less than 80. LIBXSMM provides high performance through its modular design—specifically, a separate frontend (high-level language and routine selection) and backend for xGEMM code generation\textsuperscript{2}. LIBXSMM provides a simple interface to call S/DGEMM to integrate into an application with very little effort. Figures 1, 2, and 3 show three modes in which LIBXSMM can be used for matrix multiplications.

During installation, LIBXSMM can be built explicitly for:

- Particular M, N, and K values
- Leading dimension values that differ from M, N, and K values
- Specific values of $\alpha$ and $\beta$

```c
void libxsmm smm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm dmm(int m, int n, int k, const double* a, const double* b, double* c);
```

1. **Automatically dispatched matrix multiplication API in LIBXSMM\textsuperscript{2}**

```c
void libxsmm simm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm dimm(int m, int n, int k, const double* a, const double* b, double* c);
```

2. **Non-dispatched matrix multiplication API in LIBXSMM\textsuperscript{2}**

```c
void libxsmm sblasmm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm dblasmm(int m, int n, int k, const double* a, const double* b, double* c);
```

3. **LIBXSMM API for matrix multiplication using BLAS\textsuperscript{3}**

```c
void libxsmm sblasmm(int m, int n, int k, const float* a, const float* b, float* c);
void libxsmm dblasmm(int m, int n, int k, const double* a, const double* b, double* c);
```
Enabling Eigen with Intel® MKL and LIBXSMM

In its original form, Eigen does not use Intel MKL for small matrix multiplication (specifically, when M+N+K is less than 20). To allow Eigen to call the DGEMM function in Intel MKL, we modify the Eigen source code to eliminate the M+N+K<20 heuristic and permit calls to Intel MKL DGEMM for all matrix sizes.

To enable LIBXSMM in Eigen, we replace Eigen’s native matrix-matrix multiplication implementation with a call to `libxsmm_dgemm`.

Experiment Setup

We examine the performance of two workloads that use Eigen:

1. **A simple DGEMM benchmark** that implements DGEMM on a set of square, double-precision matrices
2. **An implementation of EKF** that works on synthetically generated RADAR and LIDAR data

We use native Eigen, Eigen with Intel MKL, and Eigen with LIBXSMM in these experiments. All benchmarks are executed in serial.

Table 2 details our library and compiler versions and hardware specifications.

| Table 2. Library, compiler, and hardware specifications |
|---------------------------------|-----------------|
| **Cores per socket**            | 20              |
| **Sockets**                     | 2               |
| **L1 cache**                    | 32K             |
| **L2 cache**                    | 1,024K          |
| **L3 cache**                    | 28,160K         |
| **Clock**                       | 2.4GHz (3.7 GHz Turbo) |
| **Intel® C++ Compiler version** | 17.0.4 20170411 |
| **GNU G++ compiler version**    | 7.2             |
| **Eigen version**               | 3.3.3           |
| **Intel MKL version**           | Intel MKL 2017 Update 3 |
Benchmarking DGEMM on Intel® Xeon® Processor

In this DGEMM benchmark, our figure of merit is the improvement in performance (gigaflops/second) over native Eigen with g++. With the exception of matrix sizes 2 and 4, both Eigen with Intel MKL and Eigen with LIBXSMM provide a speedup over native Eigen across all classes of matrices. It is interesting to note that native Eigen has the lowest performance, regardless of whether it is compiled with GNU or Intel® compilers (Figures 4 and 5). In terms of performance improvement, the overall trend is that:

- **Eigen+LIBXSMM** produces the highest performance across all matrices (excluding matrix sizes 2 and 4).
- **Eigen+LIBXSMM with g++** produces the highest speedup for matrices of sizes less than size 13.
- **Eigen+LIBXSMM with ICPC** produces the highest speedup across all g++ and ICPC variants for matrix sizes greater than 13.

---

4  Speedup over native Eigen with Intel MKL and LIBXSMM using g++ 7.2

5  Speedup over native Eigen with Intel MKL and LIBXSMM using the Intel® C++ Compiler
Evaluating and Speeding Up the Extended Kalman Filter

We evaluate EKF by using native Eigen, Eigen with Intel MKL, and Eigen with LIBXSMM. From our earlier DGEMM benchmarking, we see that g++ provides higher performance for matrix sizes less than 13. Since EKF works on smaller matrices, we evaluate speedup in EKF using g++. Our baseline for evaluating speedup is EKF that uses native Eigen. Our figure of merit is the median time to predict and update each sensor measurement (a total of 10,000 sensor measurements were processed). As shown in Figure 6, incorporating Intel MKL or LIBXSMM can produce a speedup of approximately 1.2X in EKF.

![Speedup in Extended Kalman filter from using Eigen with Intel MKL and Eigen with LIBXSMM](image)
Improving Performance
In this article, we concentrated on speeding up the performance of EKF, a common automated driving workload used for sensor fusion and localization. We investigate this performance improvement on the Intel Xeon processor in two ways:

- Speeding up matrix-matrix multiplication kernel in native Eigen from using Intel MKL and LIBXSMM
- Improving performance of the EKF workload

We show a maximum speedup of 3.1X over native Eigen from using Eigen+LIBXSMM with the Intel C++ compiler. We improved EKF performance by using Intel MKL and LIBXSMM to produce a speedup of 1.2X.

References

1. Eigen: http://eigen.tuxfamily.org
3. LIBXSMM code repository: https://github.com/hfp/libxsmm
Many high-performance computing applications depend on matrix computations performed on large groups of very small matrices. The latest version of Intel® Math Kernel Library (Intel® MKL) provides new compact functions that include vectorization-based optimizations for problems of this type.

Intel MKL Compact functions rely on true SIMD (single instruction, multiple data) matrix computations, in which subgroups of matrices are operated on by kernels that abstractly appear as scalar kernels, while registers are filled by cross-matrix vectorization. These functions provide significant performance benefits compared to batched techniques, which exploit multithreading but rely on kernels written for standard data formats.
The Parallel Universe

(Find more detailed information about Intel MKL Batch general matrix-matrix multiplication here.) Besides the performance benefits that result from efficient vectorization, Intel MKL Compact functions can benefit from the same parallelization principles as batched techniques because calculations on subgroups can be threaded, offering multiple levels of parallelism to the user.

The latest version of Intel MKL has added six compact functions:

1. General matrix-matrix multiply
2. Triangular matrix equation solve
3. LU factorization (without pivoting)
4. Inverse calculation (from LU without pivoting)
5. Cholesky factorization
6. QR factorization

For ease of use, Intel has also added service functions to facilitate the packing and unpacking of groups of matrices into this format. (See the Intel MKL Developer Reference for more detailed information about the Compact API in Intel MKL 2018.)

**Compact Format**

Compact functions operate on matrices that are packed into a contiguous segment of memory in an interleaved data layout, called Compact format. In Compact format, matrices are organized into packs of length V, where V is related to the register length of the underlying architecture and the size of the matrix elements. Each pack is a 3D tensor, with the matrix index incrementing fastest. These Compact packs are then loaded into registers and operated on using SIMD instructions.

*Figure 1* shows the packing of a set of four 3x3, real-precision matrices. The pack length for this example is V=2, resulting in two compact packs.
For complex-precision matrices, the real and imaginary parts are packed separately. Real and imaginary packs alternate in memory; thus, the number of packs for a problem involving complex matrices is twice the number of packs for the same problem involving real matrices. Most arithmetic operations on complex elements can be expressed with register type variables without extra operations (i.e., shuffle operations), making this packing format suitable for complex precision Compact functions.

The pack length V is the quotient of the SIMD vector bit length and the bit size of a matrix element (or the bit size of the corresponding real type, in the case of complex precisions). For example, for double-precision matrices on an architecture with a SIMD vector length of 256 bits, V=4, allowing Compact kernels to operate on four matrices simultaneously through vectorized instructions.
Compact kernels operate on separate packs. If the number of matrices to be packed is not evenly divisible by the pack length, the last pack will only be partially filled, and cannot be processed by intrinsic-based kernels. For optimal performance, the last pack should be padded with additional data so that it’s fully packed. To avoid possible numerical issues such as division by zero, the identity matrix is used as padding, since the identity matrix will not produce numerical errors when operated on by any of the Intel MKL Compact functions. Figure 2 illustrates the padding process for three matrices when the pack length is equal to 2.

<table>
<thead>
<tr>
<th>A111</th>
<th>A112</th>
<th>A113</th>
</tr>
</thead>
<tbody>
<tr>
<td>A121</td>
<td>A122</td>
<td>A123</td>
</tr>
<tr>
<td>A131</td>
<td>A132</td>
<td>A133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A211</th>
<th>A212</th>
<th>A213</th>
</tr>
</thead>
<tbody>
<tr>
<td>A221</td>
<td>A222</td>
<td>A223</td>
</tr>
<tr>
<td>A231</td>
<td>A232</td>
<td>A233</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A311</th>
<th>A312</th>
<th>A313</th>
</tr>
</thead>
<tbody>
<tr>
<td>A321</td>
<td>A322</td>
<td>A323</td>
</tr>
<tr>
<td>A331</td>
<td>A332</td>
<td>A333</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 2** Padding with identity

<table>
<thead>
<tr>
<th>A111</th>
<th>A112</th>
<th>A113</th>
</tr>
</thead>
<tbody>
<tr>
<td>A211</td>
<td>A212</td>
<td>A213</td>
</tr>
<tr>
<td>A311</td>
<td>A312</td>
<td>A313</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A113</th>
<th>A113</th>
<th>A113</th>
</tr>
</thead>
<tbody>
<tr>
<td>A123</td>
<td>A123</td>
<td>A123</td>
</tr>
<tr>
<td>A223</td>
<td>A223</td>
<td>A223</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A323</th>
<th>A323</th>
<th>A323</th>
</tr>
</thead>
<tbody>
<tr>
<td>A333</td>
<td>A333</td>
<td>A333</td>
</tr>
</tbody>
</table>

**Figure 3** shows a simple, Compact version of a matrix-matrix multiplication performing the operation $C = A \times B$ for a set of four 3x3, real-precision matrices. Generic (or batched) functions require four matrix-matrix multiplications to be performed for a problem of this type.

Assuming that the matrices have been packed into Compact format using a pack length of $V=2$, the Compact version of this problem involves two matrix-matrix multiplications, as illustrated in Figure 4.

The elements of the matrices involved in these two multiplications are vectors of length $V$, which are loaded into registers and operated on as if they were a scalar element in an ordinary matrix-matrix multiplication.
Compact API

Before calling a Compact function, the input data must be packed in Compact format. After execution, the output data should be unpacked from this Compact format, unless another Compact function will be called immediately following the first. Figure 5 demonstrates the standard call flow when using Intel MKL Compact functions (replace ? with the desired single-letter precision designation: s, d, c, z).
Intel MKL provides service functions that facilitate the process of packing and unpacking matrices into and out of Compact format. Two service functions, `mkl_get_format_compact` and `mkl_?get_size_compact`, will calculate the optimal format for the underlying architecture and the size of the necessary Compact buffer to store the Compact arrays. Users will pass the returned values of these functions as parameters for the packing and unpacking functions `mkl_?gepack_compact` and `mkl_?geunpack_compact`.

Packing and unpacking matrices to and from Compact format adds additional overhead to calculations. Compact functions offer the greatest performance benefit when the user calls multiple Compact functions on the packed matrices, packing the matrices into Compact format before the first Compact function call and unpacking them after the last.

Intel MKL Compact functions are optimized for 128-, 256-, and 512-bit SIMD registers. Compact packs are processed individually by Compact kernels. There are two types of Compact kernels: reference and intrinsic-based. Reference kernels are implemented using C, without specific CPU optimizations, whereas intrinsic-based kernels are optimized for a specific instruction set based on vector intrinsics. In Intel MKL Compact functions, reference kernels are called in two cases:

1. **The architecture** does not support SSE2 (Intel® Streaming SIMD Extensions 2) or later.
2. **The native format** for the architecture is not compatible with the format of packed matrices.

In both cases, Compact functions will perform correct computations, but performance of the reference kernels may be lower than what one would expect from intrinsic-based kernels.
Calculations on separate Compact packs are inherently independent. Thus, Compact functions are easily threaded. Intel MKL Compact functions implement this naïve form of parallelism behind the scenes, offering improved performance results when calling Compact functions for multiple threads. Figure 6 compares two approaches for calculating the matrix inverse of many matrices. On the left, calculations are parallelized through an OpenMP* loop over all matrices. The code on the right takes advantage of vectorization and parallelism within the Compact API.

```c
/* OpenMP loop */
#pragma omp parallel for
for (i = 0; i < number_of_matrices; i++) {
    /* Perform LU Factorization */
    call mkl_dgetrfnp
    /* Perform Inverse Calculation */
    call mkl_dgetrinp
}
/* Pack from pointer-to-pointer to Compact format */
call mkl_dgepack_compact
/* Perform Compact LU Factorization */
call mkl_dgetrfnp_compact
/* Perform Compact Inverse Calculation */
call mkl_dgetrinp_compact
/* Unpack from Compact to pointer-to-pointer format */
call mkl_dgeunpack_compact
```

### Matrix inverse calculation: OpenMP parallel loop over all matrices (left) and using Compact functions (right)

## Applications

There are many potential applications for Compact BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) functions. Computer vision requires dense linear algebra operations on large groups of very small matrices. For example, anomaly detection in images requires simultaneously solving thousands of dense linear systems using their Cholesky factorizations. These factorizations are independent, and thus are strong candidates for speedup with Compact functions.

Partial differential equation (PDE)-based simulations use a discretization over a mesh that is represented using a block sparse matrix, where each block is a mesh entity (such as a node, edge, face, or cell center). For example, a 3D compressible fluid dynamics model using the ideal gas model uses 5x5 blocks. In a typical PDE-based application, an iterative linear solver then performs a sequence of matrix-vector and matrix-matrix products. Operations on and within the matrices can be performed in parallel. In Designing Vector-Friendly Compact BLAS and LAPACK Kernels (Kim et al., 2017), the authors demonstrate up to 6X application speedup of a linear solver for compressible fluid dynamics simulation using Compact matrix-matrix multiplication, triangular solve, and LU factorization.
Performance Results

Figures 7 and 8 show the performance improvement for general matrix-matrix multiplication (GEMM) and non-pivoting LU factorization of a general matrix (GETRFNP). The results are measured against calls to the generic MKL functions.

7 512 Small matrix multiplications using Intel MKL mkl_dgemm_compact

8 512 small LU factorizations using Intel MKL mkl_dgetrfnp_compact
The packing and unpacking functions add additional overhead, which is mitigated by calling multiple Compact functions between the calls to pack and unpack. A typical use case is to calculate the inverse from a non-pivoting LU factorization. As shown in Figure 9, even when including overhead in performance the Intel MKL Compact functions still provide consistently good speedup, with some sizes and numbers of matrices demonstrating up to 4X speedup compared to calls to the generic Intel MKL functions.

![Figure 9 showing speedup comparison](image)

Configuration info - Version: Intel® Math Kernel Library (Intel® MKL) 2018; Hardware: Intel® Xeon® Platinum Processor 8180; 2 Twenty-eight-core CPU (2.5 GHz), 96 GB of RAM; Operating System: Windows 7.6.64

**9. LU factorization + inverse calculation (without pivoting) for 128, 512, and 2,048 small matrices, including overhead**

### Speeding Algebra Computations

Storing data in non-standard layouts that allow for cross-matrix vectorization can provide a significant speedup in BLAS and LAPACK functions for small-sized matrices. The new Compact functions in Intel MKL 2018 support and maximize the performance benefits of the Compact data layout.

---

INNOVATE
System & IoT Applications

Intel® System Studio 2018 Has Landed

Super-charge development for your IoT devices and applications with the new version of this all-in-one tool suite that simplifies system bring-up and maximizes performance on Intel® platforms. Check out a universe of new features including support for Java* and Intel Atom® platforms, libraries that accelerate data processing at the edge, access to cloud connectors and 400+ sensors, an enhanced debugger, and trace automation.

Free 90-Day Commercial License >

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel, Atom, and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
© 2018 Intel Corporation

#PurePerformance
Since its introduction more than two decades ago, Java® has become a popular enterprise language. Embraced initially for its write-once, run-anywhere property, its usage increased due to the relative ease of building new applications by integrating existing Java projects. The growth of several open-source Java application stacks such as Apache Hadoop® has added to this growth. Fundamentally, developers view Java as enhancing their productivity.

Java programmers rely on the Java Virtual Machine® (JVM®) to deliver on the write-once, run-anywhere promise. The JVM also delivers high performance. By contrast, C and C++ developers rely on the compiler. For these languages, compilation is not in the critical path to runtime performance and happens relatively rarely,
so the compiler can do deep analysis and provide esoteric optimizations. The JVM, on the other hand, uses a just-in-time (JIT) compilation model. Compilation is dynamic and happens during every run, so it needs to be lightweight—yet as sophisticated as possible. A lot of work has gone into optimizing the JIT compiler over the years, and for general business processing, the performance of Java has been close to that provided by C or C++.

Today’s business applications—for example, analytics and deep learning—require extensive numerical computing. These use cases all consume enormous amounts of data while doing large amounts of matrix math. Java applications are a natural fit for marshalling and delivering the data. It’s not an exaggeration to say that big data runs on the JVM. However, the heavy use of floating-point math is more naturally in the domain of hand-tuned kernels written in C or assembly. Improving the Java ecosystem to handle the numerical computing will allow developers to be more productive.

Big data applications, distributed deep learning programs, and artificial intelligence solutions can run directly on top of existing Apache Spark* or Hadoop clusters, and can benefit from efficient scale-out. Machine learning algorithms running on top of the Spark framework, and the current revolution of data science on the JVM, push the need for enhanced single instruction, multiple data (SIMD) support in Java. SIMD support would open up ways to explore new opportunities in areas like high-performance computing (HPC), linear algebra-based machine learning (ML) algorithms, deep learning (DL), and artificial intelligence (AI).

To illustrate, let’s consider the new Math FMA API available in Open JDK9* and explore its Java performance in a few ML algorithms on the Intel® Advanced Vector Instructions (Intel® AVX)-enabled Intel® Xeon Phi™ processors.

**Fused Multiply Add (FMA) Operations**

Modern Intel® processors include Intel AVX instructions for performing SIMD operations, including FMA operations \((A = A \cdot B + C)\). FMA is highly valuable for linear algebra-based ML algorithms, deep learning and neural networks (dot product, matrix multiplication), financial and statistical computation models, and polynomial evaluations. FMA instructions perform vectored \(a \cdot b + c\) operations on IEEE-754-2008 floating-point values, where the \(a \cdot b\) multiplications are performed with infinite precision. The final results of the addition are rounded to produce the desired precision. The FMA instructions have been available in second-generation and later Intel® Core™ processors. To leverage them in SIMD operations, the JVM JIT compiler needs to map FMA operations written in Java to further Intel AVX FMA extensions, if available, on the underlying CPU platform.
FMA API Available in Java 9

Open JDK9 provides the FMA API for Java developers as part of the java.lang.math package. The open JDK9 release includes compiler intrinsics for Intel AVX FMA extensions, which map FMA Java routines to CPU instructions directly on modern CPUs (e.g., Intel Xeon Phi and Intel® Xeon® Platinum 8180 processors). This doesn't require any additional work from the developer. However, Java algorithms designed to use FMA instructions should take into consideration that a non-FMA sequence of packed floating-point multiply and add instructions likely will produce slightly different results compared to FMA. When formulating convergence criteria, it's important to factor in the difference in the precision of intermediate results to avoid surprises in the final results. In Java, when $a*b+c$ is evaluated as a regular floating-point expression, two rounding errors are involved: the first for the multiply operation, the second for the addition operation.

In Java, FMA operations are supported via the Math FMA API. The FMA routine returns the FMA of the three arguments. It returns the exact product of the first two arguments, summed with the third argument, and then rounded once to the nearest double-precision value. The FMA operation is performed using the java.math.BigDecimal class. Infinity and NaN arithmetic input values aren't supported by BigDecimal; these inputs are handled with two roundings for computation of correct results.

The FMA API takes floating-point inputs $a$, $b$, and $c$ and returns floating-point type. It supports both single- and double-precision. The FMA computation is performed in double-precision (discussed later). The implementation first screens for and handles non-finite input values whose arithmetic is not supported by BigDecimal. If all inputs are within finite range, the following expression computes the product of floating-point inputs $a$ and $b$ by explicitly casting them into BigDecimal objects:

```java
BigDecimal product = (new BigDecimal (a)).multiply (new BigDecimal (b));
```

In special cases, where a third floating-point input $c$ is zero, the API carefully handles the sign of a zero in case the product ($a*b$) is also zero. The sign for the zero final result is computed by the following floating-point expression:

```java
if (a == 0.0 || b == 0.0) {
    return a * b + c;
}
```
In cases where \( c \) is zero and the product is nonzero, the `doubleValue()` of the `BigDecimal` product is returned:

```java
else {
    return product.doubleValue();
}
```

For all nonzero inputs \( a, b, \) and \( c \):

```java
else {
    return product.add(new BigDecimal(c)).doubleValue();
}
```

The following example shows the \( A[i] = A[i] + C \times B[i] \) operation using the FMA API available in Open JDK9:

```java
for (int i = 0; i < A.length && i < B.length; i++)
```

FMA for both single-precision and double-precision floats is computed in the double format, and then explicitly stored as a float or double to match the return type. Since double has more than twice the precision of the float format, the multiplication of \( a \times b \) is exact. Addition of \( c \) to the product incurs one rounding error. Moreover, since double has more than \((2p+2)\) precision bits compared to the \( p \) bits of float, the two roundings of \( a \times b + c \), first to double and then secondarily to float, are equivalent to rounding the intermediate result directly to float. The FMA method calls are further mapped to CPU FMA instructions (if available) for higher performance.

When you're running your Java applications on the latest Open JDK 9 release and source builds, the JVM enables hardware-based FMA intrinsics for hardware where FMA instructions are available (on Intel processors beginning with second-generation Intel Core processors). FMA intrinsics are generated for the `java.lang.Math.fma(a, b, c)` methods in JDK9, which calculate value of \( a \times b + c \) expressions.

The JVM option `-XX:+PrintIntrinsics` can be used to confirm the FMA intrinsification:

```java
@ 6 java.lang.Math::fma (12 bytes) (intrinsic)
```
FMA Performance on BLAS Machine Learning Algorithms

In Java programs, computational kernels can be implemented using the Math FMA, which can further leverage FMA hardware extensions on modern CPUs. On the latest Open JDK 10 source builds, BLAS I DDOT (Dot Product) performance can improve up to 3.5X on Intel Xeon Phi processors using Math.fma. The JVM JIT compiler intrinsifies the FMA method calls into hardware instructions, which are further vectorized via auto-vectorization and super-word optimizations into SIMD operations. BLAS-I DAXPY performance can improve up to 2.3X on Intel Xeon Phi processors using the Math FMA on the latest Open JDK source builds.

The original Java DAXPY implementation is shown below:

```java
int _r = n % 4;
int _n = n - _r;
for (i = 0; i < _n && i < dx.length; i+=4) {
    dy[i     +dy_off] = dy[i     +dy_off] + da*dx[i     +dx_off];
    dy[i+1 +dy_off] = dy[i+1 +dy_off] + da*dx[i+1 +dx_off];
    dy[i+2 +dy_off] = dy[i+2 +dy_off] + da*dx[i+2 +dx_off];
    dy[i+3 +dy_off] = dy[i+3 +dy_off] + da*dx[i+3 +dx_off];
}
for (i = _n; i < n; i++)
    dy[i +dy_off] = dy[i +dy_off] + da*dx[i +dx_off];
```

Here is the Math.fma implementation of the DAXPY algorithm:

```java
for (i = 0; i < dx.length; i++)
    dy[i] = Math.fma (da, dx[i], dy[i]);
```

The machine code generated by the JVM JIT compiler is shown below. We see that FMA vector SIMD instructions are generated for the Java algorithm and mapped into the SIMD vfmadd231pd instructions on the Intel Xeon Phi processor for the latest Open JDK source build.

```assembly
vmovdqu64 0x10(%rbx,%r13,8),%zmm2{%k1}{z}
vfmadd231pd 0x10(%r10,%r13,8),%zmm1,%zmm2{%k1}{z}
;*invokestatic fma {reexecute=0 rethrow=0
return_oop=0}
; - SmallDoubleDot::daxpy@146 (line 82
```

Applications of the Math FMA Interface

The Math FMA application interface is very useful in programs that perform basic linear algebra computations. Dense DGEMM (double-precision matrix multiplication) innermost compute kernels, for instance, can be rewritten using Math.fma operations as follows:
Sparse matrix calculations can be rewritten as follows:

```java
for (int steps = 0; steps < NUM_STEPS; steps++) {
    for (int l = 0; l < ALPHA; l++) {
        double Total = 0.0;
        int rowBegin = Rows[l];
        int rowEnd = Rows[l+1];
        for (int j=rowBegin; j<rowEnd; j++) {
            Total = Math.fma (A[Cols[j],Value[j],Total);
        }
        y[l] = Total;
    }
}
```

In Binomial Option Pricing, a common financial algorithm, the operation

\[ \text{stepsArray}[k] = p_d \text{Byr} \times \text{stepsArray}[k+1] + p_u \text{Byr} \times \text{stepsArray}[k] \]

can be written using FMA:

```java
void BinomialOptions (double[] stepsArray, int STEPS_CACHE_SIZE,
                     double vsdt, double x, double s, int numSteps,
                     int NUM_STEPS_ROUND, double p_dByr,
                     double p_uByr) {
    for (int j = 0; j < STEPS_CACHE_SIZE; j++) {
        double profit = s * Math.exp (vsdt * (2.0D * j - numSteps)) - x;
        stepsArray[j] = profit > 0.0D? profit: 0.0D;
    }
    for (int j = 0; j < numSteps; j++) {
        for (int k = 0; k < NUM_STEPS_ROUND; ++k) {
            stepsArray[k] = Math.fma (p_uByr,
                                     stepsArray[k],
                                     (p_dByr * stepsArray[k+1]));
        }
    }
}
```
Boosting Java Performance

New enhancements to Java enable faster and better numerical computing. The JVM has been improved to use the Intel AVX FMA instructions in the implementation of the Math.fma(). This results in significant performance improvements of matrix multiplications, the basic workhorse of HPC and AI applications.

---

Java and Intel® Technology: Building the Future
BY MICHAEL G., INTEL CORPORATION

In my keynote address at JavaOne 2017 on Oct 2nd, I talked about building the future of data with Java. The growth in daily data generation continues, but the Internet of Things (IoT) soon will outpace the connections and data generated by people.

Smart, connected devices and sensors are joining the Internet at a rapid pace, and they will generate tremendous amounts of data. By itself, data is not so interesting; the critical idea is to rapidly extract insight and intelligence from data.

The speed of insight has become a new competitive advantage for companies—it boosts their business, helping to reduce costs, better understand customers, create new innovations...

Read more >
Discover limitless possibilities with amazing code.
Speed up performance with vectorization and threading,
powerful new analysis and profiling tools, and much more with Intel® Parallel Studio XE.

Try it today >

#PurePerformance
GAINING PERFORMANCE INSIGHTS USING THE INTEL® ADVISOR PYTHON* API

Getting Good Data to Make Code Tuning Decisions

Kevin O'Leary, Technical Consulting Engineer, and Egor Kazachkov, Senior Software Developer, Intel Corporation

Good design decisions are based on good data:

- What loops should be threaded and vectorized first?
- Is the performance gain worth the effort?
- Will the threading performance scale with higher core counts?
- Does this loop have a dependency that prevents vectorization?
- What are the trip counts and memory access patterns?
- Have you vectorized efficiently with the latest Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions? Or are you using older SIMD instructions?
Intel® Advisor is a dynamic analysis tool that’s part of Intel® Parallel Studio XE, Intel’s comprehensive tool suite for building and modernizing code. Intel Advisor answers these questions—and many more. You can collect insightful program metrics on the vectorization and memory profile of your application. And, besides providing tailored reports using the GUI and command line, Intel Advisor now gives you the added flexibility to mine a collected database and create powerful new reports using Python*.

When you run Intel Advisor, it stores all the data it collects in a proprietary database that you can now access using a Python API. This provides a flexible way to generate customized reports on program metrics. This article will describe how to use this new functionality.

**Getting Started**

To get started, you need to setup the Intel Advisor environment. (For this article, we ran all the scripts on Linux*, but the Intel Advisor Python API also supports Windows*.)

```bash
source advixe-vars.sh
```

Next, to set up the Intel Advisor data, you need to run some collections. Some of the program metrics require additional analysis such as tripcounts, memory access patterns, and dependencies.

```bash
advixe-cl --collect survey --project-dir ./your_project -- <your-executable-with-parameters>
advixe-cl --collect tripcounts -flops-and-masks -callstack-flops --project-dir ./your_project -- <your-executable-with-parameters>
```

To run a map or dependencies collection, you need to specify the loops that you want to analyze. You can find this information using the Intel Advisor GUI or by doing a command-line report.

```bash
advixe-cl --collect map -mark-up-list=1,2,3,4 --project-dir ./your_project -- <your-executable-with-parameters>
advixe-cl --collect dependencies -mark-up-list=1,2,3,4 --project-dir ./your_project -- <your-executable-with-parameters>
```

Finally, you will need to copy the Intel Advisor reference examples to a test area.

```bash
cp -r /opt/intel/advisor_2018/pythonapi/examples .
```

Note that all the scripts we ran for this article use the Python that currently ships with Intel Advisor on Linux. The standard distributions of Python should also work just as well.
Using the Intel Advisor Python API

The reference examples we’ve provided are just small set of the reporting that's possible using this flexible way to access your program data. You could use the `columns.py` example to get a list of available data fields. For example, you could see the metrics in Table 1 after running a basic survey collection.

Table 1. Sample survey metrics

<table>
<thead>
<tr>
<th>Capability</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler version</td>
<td>compiler_version</td>
</tr>
<tr>
<td>Vectorization status</td>
<td>is_vectorized</td>
</tr>
<tr>
<td>Mangled function or loop</td>
<td>mangled_name</td>
</tr>
<tr>
<td>Data types provided by binary static analysis</td>
<td>data_types</td>
</tr>
<tr>
<td>Loop unroll factor applied by compiler</td>
<td>unroll_factor</td>
</tr>
</tbody>
</table>

Intel Advisor Python API in Action

Let's walk through a simple example that shows how to collect some powerful metrics using the Intel Advisor Python API. The first step is to import the Intel Advisor library package.

```python
import advisor
```

You then need to open the Intel Advisor project that contains the result you've collected.

```python
project = advisor.open_project(sys.argv[1])
```
You also have the option of creating a project and running collections. (In the example below, we're just doing an open_project.) In this example, we access data from the memory access pattern (MAP) collection. We do this using the following line of code:

```python
data = project.load(advisor.MAP)
```

Once we've loaded this data, we can loop through the table and gather cache utilization statistics. We then print out the data we've collected:

```python
import sys
try:
    # First import the Advisor library
    import advisor
except ImportError:
    sys.exit(1)
# Open your Advisor project
    project = advisor.open_project(sys.argv[1])
# Load the Memory Access Pattern (MAP) data
    data = project.load(advisor.MAP)
# Loop through the MAP data and gather information about cache utilization
for site in data.map:
    site_id = site['site_id']
    cachesim = data.get_cachesim_info(site_id)
    print(indent * 2 + 'Average utilization'.ljust(width) + ' = {:.2f}%'.format(cachesim.utilization))
```

## Intel Advisor Python API Advanced Topics

The examples provided as part of the Intel Advisor Python API give you a blueprint for writing your own scripts. **Table 2** shows some of these advanced capabilities.
Table 2. Intel Advisor Python API advanced capabilities

<table>
<thead>
<tr>
<th>Capability</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an Intel Advisor project</td>
<td><code>compiler_version project = advisor.create_project(project_dir)</code></td>
</tr>
<tr>
<td>Run an Intel Advisor survey collection</td>
<td><code>data = project.collect(advisor.SURVEY)</code></td>
</tr>
<tr>
<td>Run an Intel Advisor tripcounts collection</td>
<td><code>data = project.collect(advisor.TRIPCOUNTS)</code></td>
</tr>
<tr>
<td>Run an Intel Advisor tripcounts collection and also collect FLOPS data</td>
<td><code>data = project.collect(advisor.TRIPCOUNTS, collection_args=['flop'])</code></td>
</tr>
<tr>
<td>Run an Intel Advisor tripcounts collection but only collect FLOPS and not tripcounts</td>
<td><code>data = project.collect(advisor.TRIPCOUNTS, collection_args=['flop', 'no-trip-counts'])</code></td>
</tr>
<tr>
<td>Run an Intel Advisor Roofline collection</td>
<td><code>data = project.collect(advisor.ROOFLINE)</code></td>
</tr>
<tr>
<td>Run an Intel Advisor memory access pattern (MAP) collection</td>
<td><code>data = project.collect(advisor.MAP)</code></td>
</tr>
<tr>
<td>Run an Intel Advisor dependencies collection</td>
<td><code>data = project.collect(advisor.DEPENDENCIES)</code></td>
</tr>
</tbody>
</table>

Here are some highlights of our various examples. We are constantly adding to the list of examples.

Generate a combined report showing all data collected:

```python
project = advisor.create_project(project_dir)
```

Generate an html report:

```bash
advixe-python to_html.py ./your_project
```
You can generate a roofline HTML chart (Figure 1) with this code:

```
python roofline.py ./your_project
```

You must run the `roofline.py` script with an external Python command and not `advixe-python`. It currently only runs on Linux. It also requires the additional libraries numpy, pandas, and matplotlib to be installed. Use this code to generate cache simulation statistics:

```
advixe-python cache.py ./your_project
```
You can see the results we obtained from the cache model in Table 3.

**Table 3. Cache model results**

<table>
<thead>
<tr>
<th>Capability</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes</td>
<td>46</td>
</tr>
<tr>
<td>Reads</td>
<td>92</td>
</tr>
<tr>
<td>Read misses</td>
<td>50</td>
</tr>
<tr>
<td>Average evicted cache line utilization</td>
<td>6.25%</td>
</tr>
<tr>
<td>Evicted cache line bytes used</td>
<td>4</td>
</tr>
<tr>
<td>Evicted lines</td>
<td>48</td>
</tr>
</tbody>
</table>

**Case Study: Vectorization Comparison**

In this case study, we create a Python script that can compare the vectorization of a given loop when compiled with different compiler options.

**Step 1: Compile Code with Different Optimization Flags**

First, compile the app with different options. In this example, we use the Intel® C++ Compiler (but Intel Advisor works at the binary level, so any compiler should work). In the first case, we are compiling without optimization using the compiler option `-O0`. The second case uses full optimization `-O3`.

```bash
icc loops1.cpp -O0 -g -debug inline-debug-info -qopt-report=5 -ipo -o loops1-no-opt
icc loops1.cpp -O3 -g -debug inline-debug-info -qopt-report=5 -ipo -o loops1
```
Step 2: The Python Code

The script is very simple. First, get some arguments from the command-line. If they are being passed an Intel Advisor project, then use the data contained in the project. Otherwise, do an Intel Advisor survey run. Once the survey runs complete, decode the assembly for the loops and print the instructions of the two loops side-by-side. The main function in our Python code is named `get_formatted_asm`. This function is able to access the Intel Advisor database and decode the assembly for our loops. It can also check whether the assembly code is using vector instructions, as well as how fast the loop executed.

```python
import sys
import itertools
import advisor

# second form allows collecting data before analysis
# first form just analyses already collected data
if len(sys.argv) < 3 or len(sys.argv) > 6:
    print('''
Usage:
advixe-python {} path_to_project_dir loop1 [loop2]
Or:
advixe-python {} path_to_project_dir loop1 [loop2] executable1 executable2
'''.format(__file__, __file__))
    sys.exit(1)

project_dir = sys.argv[1]
project_dir1 = project_dir + ".1"
project_dir2 = project_dir + ".2"
loop1 = sys.argv[2]
# if we have an odd number of arguments (including script name) then loop2 is the same as loop1
loop2 = sys.argv[3] if len(sys.argv)%2 == 0 else loop1
binary1 = ''
binary2 = ''

# in the second form two last args are executables to run
if 4 < len(sys.argv) < 7:
    binary1 = sys.argv[-2]
    binary2 = sys.argv[-1]
# try open or create project, run collection if needed:
# returns formatted asm listing with vectorized instructions marked with "VEC " for given loop
def get_formatted_asm(project_dir, binary, loop):
    asm = []
    try:
        project = advisor.open_project(project_dir)
        except:
            project = advisor.create_project(project_dir)
    if binary:
        project.collect(advisor.SURVEY, binary)
    data = project.load(advisor.SURVEY)
    for entry in data.bottomup:
        if loop in entry["function_call_sites_and_loops"]:  
            asm += ["{:54.54} " format(entry["function_call_sites_and_loops"]) ,  "{:54.54} " format("Self time: " + entry["self_time"]),  "{:54.54} " format("Self time: " + entry["self_time"])],
        isVectorized = "VEC " if "VECTORIZED" in instruction["instruction_type"] else ""
        if Vectorized, instruction["asm"]):
            asm.append("{:4.4}{:50.50} " format(isVectorized, instruction["asm"]))
        asm.append(""
        return asm

asm1 = get_formatted_asm(project_dir1, binary1, loop1)
asm2 = get_formatted_asm(project_dir2, binary2, loop2)

# print alongside asm listings for comparison
for (a1,a2) in itertools.zip_longest(asm1, asm2, fillvalue = ' '*40):
    print('{}{}'.format(a1,a2))
```

For more complete information about compiler optimizations, see our Optimization Notice.
Step 3: Run the Python Script

```plaintext
advixe-python compare_asm.py /home/work/projects/loops-compare loops1.cpp:34 /home/work/tests/loops/loops1-no-opt /home/work/tests/loops/loops1

[loop in main at loops1.cpp:34]
Self time: 45.5463

Block 1
movl  -0xbc(%rbp), %eax
movsx %eax, %rax
imul $0x8, %rax, %rax
addq  -0x88(%rbp), %rax
movl  -0xac(%rbp), %edx
imull  -0xac(%rbp), %edx
mov $0x1, %ecx
addl  -0xac(%rbp), %ecx
movq  %rax, -0x28(%rbp)
mov %edx, %eax
cdq
idiv %ecx
cvttsi2sd %eax, %xmm0
movl  -0xc0(%rbp), %eax
movsdq  0x555(%rip), %xmm2
divsd %xmm2, %xmm1
movq  -0x28(%rbp), %rax
movsdq  (%rax), %xmm1
subsd %xmm0, %xmm1
movl  -0xbc(%rbp), %eax
movsx %eax, %rax
imul $0x8, %rax, %rax
addq  -0x88(%rbp), %rax
movsdq  %xmm1, (%rax)
mov  $0x1, %eax
addl  -0xac(%rbp), %eax
mov  %eax, -0xac(%rbp)
movl  -0xac(%rbp), %eax
cmp $0x64, %eax
jl 0x401020 <Block 1>

[loop in main at loops1.cpp:34]
Self time: 4.62404

Block 1
VEC movdqa %xmm11, %xmm2
VEC movdqa %xmm11, %xmm0
VEC psrlq $0x20, %xmm2
VEC movdqa %xmm10, %xmm1
VEC pmuludq %xmm2, %xmm2
VEC pmuludq %xmm11, %xmm0
VEC psllq $0x20, %xmm2
VEC pand %xmm13, %xmm0
VEC por %xmm2, %xmm0
callq  0x401770 <__svml_idiv4>

Block 2
VEC cvtdq2pd %xmm0, %xmm2
VEC punpckhqdq %xmm0, %xmm0
add $0x4, %r15b
VEC cvtdq2pd %xmm0, %xmm3
VEC addpd %xmm14, %xmm2
VEC addpd %xmm14, %xmm3
VEC subpd %xmm2, %xmm12
VEC subpd %xmm3, %xmm8
VEC paddd %xmm15, %xmm11
VEC paddd %xmm15, %xmm10
cmp $0x64, %r15b
jb 0x401397 <Block 1>
```

For more complete information about compiler optimizations, see our Optimization Notice.
Step 4: Recompile with AVX2 Vectorization

Now let's try a further optimization. Since our processor supports the AVX2 instruction set, we are going to tell the compiler to generate AVX2. (You should note that this generally not what the compiler with generate by default.)

```
icc loops1.cpp -O3 -xCORE-AVX2 -g -debug inline-debug-info -qopt-report=5 -ipo -o loops1-avx2
```

---

Step 5: Rerun the Comparison

```
advixe-python compare_asm.py /home/work/projects/loops-compare-opt loops1.cpp:34
/home/work/tests/loops/loops1 /home/work/tests/loops/loops1-avx2

[loop in main at loops1.cpp:34]
Self time: 4.81401

Block 1
VEC movdqa %xmm11, %xmm2
VEC movdqa %xmm11, %xmm0
VEC psrlq $0x20, %xmm2
VEC movdqa %xmm10, %xmm1
VEC pmuludq %xmm2, %xmm2
VEC pmuludq %xmm11, %xmm0
VEC psllq $0x20, %xmm2
VEC pand %xmm13, %xmm0
VEC por %xmm2, %xmm0
  callq 0x401770 <__svml_idiv4>

Block 2
VEC cvtdq2pd %xmm0, %xmm0
VEC punpckhqdq %xmm0, %xmm0
  add $0x4, %r15b
VEC cvtdq2pd %xmm0, %xmm3
VEC addpd %xmm14, %xmm2
VEC addpd %xmm14, %xmm3
VEC subpd %xmm2, %xmm12
VEC subpd %xmm3, %xmm8
VEC padd %xmm15, %xmm11
VEC padd %xmm15, %xmm10
  cmp $0x64, %r15b
  jb 0x401397 <Block 1>
```

```
[loop in main at loops1.cpp:34]
Self time: 1.97998

Block 1
VEC vpmulld %ymm15, %ymm15, %ymm0
VEC vmovdqa %ymm14, %ymm1
  callq 0x4018f0 <__svml_idiv8>

Block 2
  add $0x8, %r15b
VEC vextractil28 %0x1, %ymm0, %xmm2
VEC vcvtqd2pd %ymm0, %ymm3
VEC vpaddd %ymm13, %ymm15, %ymm15
VEC vcvtqd2pd %ymm2, %ymm5
VEC vpaddd %ymm13, %ymm15, %ymm14
VEC vaddpd %ymm3, %ymm10, %ymm4
VEC vaddpd %ymm5, %ymm10, %ymm6
VEC vsubpd %ymm4, %ymm8, %ymm8
VEC vsubpd %ymm6, %ymm9, %ymm9
  cmp $0x60, %r15b
  jb 0x4015a9 <Block 1>
```
You can see that the assembly code now uses YMM registers instead of XMM, doubling the vector length and giving a 2X speedup.

Results
The gains we made by optimizing and by using the latest vectorization instruction set were significant:

- No optimization of `-O0`: 45.148 seconds
- Optimizing `-O3`: 4.403 seconds
- Optimizing and AVX2 `-O3 -AVX2`: 2.056 seconds

Maximizing System Performance
On modern processors, it’s crucial to both vectorize and thread software to realize the full performance potential of the processor. The new Intel Advisor Python API in Intel Parallel Studio XE provides a powerful way to generate program statistics and reports that can help you get the most performance out of your system. The examples we outlined in this article illustrate the power of this new interface. Based on your specific needs, you can tailor and extend these examples. Intel is actively gathering feedback on the Intel Advisor Python API. If you’ve tried it and found it useful, or would like to provide feedback, send email to vector_advisor@intel.com.
Sharpen your technical skills. Get expert answers. Dive into new development areas. Explore it all with Intel’s free technical webinars—now available on demand.
With the rise of artificial intelligence (AI), it's no surprise that Intel has a clear vision and approach to engaging with those at the cutting edge of innovation—developers and academics. Whether you're just starting out or already an expert, the Intel® AI Academy provides learning materials, tools, and technology to help shape, create, build, and develop the future of AI.

Intel’s AI Story
Intel has a full stack of end-to-end AI products for building AI solutions across all levels—from consumer to enterprise scale. This includes a machine-learning- and deep-learning-specific hardware portfolio (Figure 1), and also a completely optimized software stack to deliver game-changing AI applications and solutions.
At the library level, we’ve optimized primitive functions that are used across a wide array of machine and deep learning frameworks and solutions, including the Intel® Math Kernel Library (Intel® MKL), Intel® Data Analytics Acceleration Library (Intel® DAAL), and Intel® Distribution for Python®.

Then, at the framework level, we’ve committed to optimizing the most popular analytics, machine, and deep learning frameworks—enabling developers to use their choice of frameworks.

In terms of tools, we offer the Intel® Deep Learning SDK to accelerate deep learning training and deployment, the Intel® Saffron™ natural intelligence platform for reasoning systems, and are a key contributor to the open-source Trusted Analytics Platform® for classic machine learning and data analytics.

Welcome to the Intel® AI Academy

The Intel AI Academy is a membership program designed to give developers, data scientists, students, and educators the tools they need to get hands-on with Intel’s technology, and to experiment and design solutions for the future of AI—taking advantage of all that Intel has to offer, from hardware to platforms and experiences.
The program is based around four key pillars:

1. Learning
2. Developing
3. Sharing
4. Teaching

Academy members can stay on top of the latest developments in the AI space with learning materials and tools, run their own solutions using Intel® Cloud Technology, and get feedback and support from peers and experts on their AI projects.

We believe access to technology and community are key to driving innovation, and the AI Academy offers just that. Figure 2 shows the Academy's key offerings.
The Academy includes:

- **Online training and tutorials** from both Intel and our partners
- **Curated** learning paths
- **Technical content** from AI Academy community experts
- **Access** to the *Intel® AI DevCloud*

If you sign up to join the Academy, you can request free access to the Intel AI DevCloud, powered by **Intel® Xeon® Scalable processors**, to help you with your machine learning and deep learning training and inference computing needs while leveraging optimized frameworks.

**The Community: Developers, Innovators, and Student Ambassadors**

Besides extensive training and development resources, the Intel AI Academy offers a robust community of developers, data scientists, students, and academics working on innovative and exciting AI projects. In working with such a diverse group of individuals, Intel has had the opportunity to seed and facilitate innovation across verticals, industries, and focus areas—with traditional developers, but also with Intel’s own innovators and student ambassadors.

**NASA Frontier Development Lab**

Intel supported and mentored researchers on the NASA Frontier Development Lab (FDL) (**Figure 3**) team, which used the *Intel® Nervana™* platform’s deep learning technology to tackle the challenge of building detailed maps of the lunar poles. The team demonstrated that deep learning could achieve the same results as a human expert with vastly improved speeds, suggesting that detailed maps of all rocky objects in the solar system could be automated using this process. The NASA FDL team used Intel Nervana Cloud access and neon framework to achieve these insights. Learn more about Intel and NASA FDL’s partnership [here](#).
Intel® Software Innovator Peter Ma has been working on using artificial intelligence to identify cancerous skin moles (Figure 4). Using the Intel® Movidius™ Neural Compute Stick, the platform currently uses 8,000 image variables to determine one of four possibilities:

1. Nothing
2. A mole
3. Melanoma
4. Some other type of cancer

Users first go to the website, upload a concerning mole or mark on their skin, and then get a result within seconds. If the AI thinks what you’ve got may be cancerous, it recommends you see a doctor for further testing. As more images are acquired, the model becomes smarter and prediction accuracy increases.

Learn more about Intel Software Innovator Peter Ma and Doctor Hazel here.
Deep Learning and Cryptocurrencies

Intel Student Ambassador Teju Tadi is looking at a new potential use case of deep learning as it relates to developing a cryptocurrency trader sentiment detector. Building on the concept of sentiment analysis of news headlines to determine price movements in the stock market, this use case takes it to the relatively new cryptocurrency markets using recursive neural tensor networks (RNTN).

Read more about Intel Student Ambassador Teju Tadi’s cryptocurrency project here.

Are You a Student or Professor?

On the academic side, the Intel AI Academy has features and benefits specifically designed for students and professors to get hands-on experience.

Intel AI Academy for Students

The Intel AI Academy for Students offers a myriad of opportunities and touchpoints for students to engage directly with Intel. Benefits include:

- **Live and interactive** training on the latest optimized frameworks and tools
- **Free** training courses
- **Opportunities** to join data science contests and hackathons
- **The opportunity** to become a Student Ambassador

Intel Student Ambassador Program

This program facilitates an exclusive group of graduate students from top institutions around the world to further their research and showcase their expertise, inspiration, and innovation using Intel® architecture. The program offers public recognition for students, both online and at industry events, plus formal affiliation with Intel and access to engineers and subject matter experts for mentorship and support.

Intel AI Academy for Professors

Finally, the Academy offers numerous benefits to professors and academics in the form of tools to easily engage students in AI. Professors can get access to expert-led lessons and teaching materials, videos, coursework, and exclusive remote access to resources like the latest libraries, frameworks, tools, and environments. This includes extended access to Intel computing resources for learning and education. Also, the Intel AI Academy for Professors offers opportunities to create instructional content in collaboration with Intel for those interested in pursuing grants and industry engagements.
Enter the World of AI: Join the Intel AI Academy Today

Whether you want to learn, start developing, or share your work, you can do it as part of the Intel AI Academy. Here’s where to start your journey:

- **Intel® AI DevCloud.** There’s no better way to enter a new arena than to jump in and get your hands dirty. [Request access](#) to our AI DevCloud to sandbox, develop, test, and optimize your AI solutions.
- **Student Kits.** Delve into AI theory and learn with hands-on exercises as part of our free, self-guided AI Student Kits. Lessons explore tools and optimized libraries that take advantage of Intel® processors in personal computers and server workstations. Get started with [Machine Learning 101 or Deep Learning 101 today](#).
- **Intel® Developer Mesh.** [Check out our online developer community](#) to browse featured projects, contribute ideas, or share your own work, as well as to follow your favorite development topics, including the latest trends, tutorials, and support.
THE PARALLEL UNIVERSE