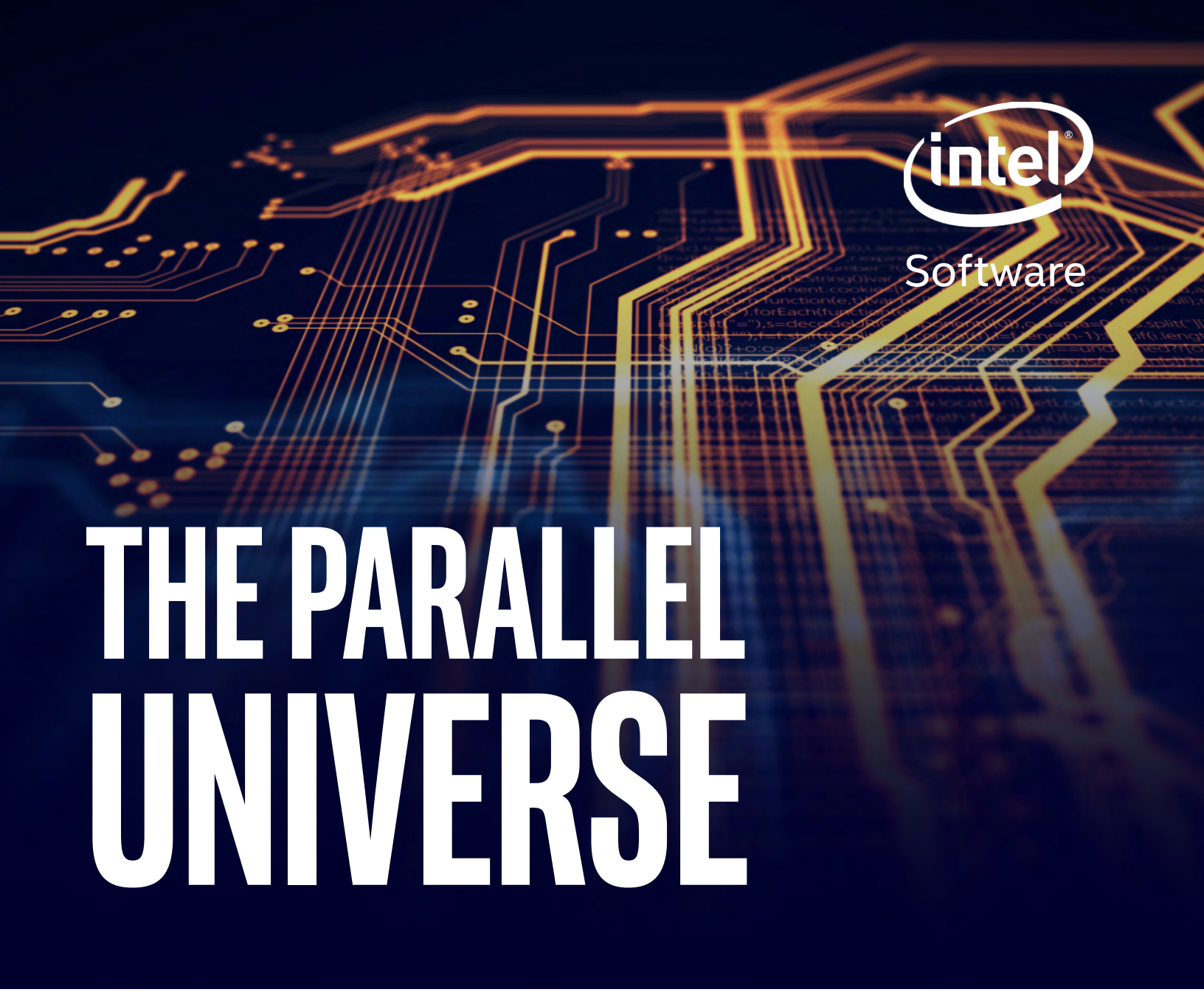


Software

THE PARALLEL UNIVERSE

Driving Code Performance with Intel® Advisor's Flow Graph Analyzer

Modernize your Code with Intel® Parallel Studio XE

Enabling FPGAs for Software Developers

Issue 30 2017

CONTENTS

LETTER FROM THE EDITOR

Henry A. Gabb, Senior Principal Engineer at Intel Corporation, **is a long-time high-performance and parallel computing practitioner and has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and was program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.**

Meet Intel® Parallel Studio XE 2018

Intel Parallel Studio XE 2018 is the latest version of Intel's comprehensive tool suite for modernizing software on Intel® architectures. In honor of its release, we've included several articles on Intel Parallel Studio XE components in this issue. **Modernize Your Code for Performance, Portability, and Scalability** gives a high-level overview of the many new features and capabilities in this tool suite. (You can also learn more in **my blog** on the release of Intel Parallel Studio XE 2018.) **Dealing with Outliers** shows how to detect fraud in a real-world dataset of credit card transactions, using the **Intel® Data Analytics Acceleration Library** to achieve high accuracy at very high performance.

Intel Parallel Studio XE has always supported OpenMP*. The latest release supports OpenMP 4.5 and many features of the 5.0 draft specification. We close out our celebration of OpenMP's 20th birthday with a final guest editorial, this one from Barbara Chapman, Professor at Stony Brook University and Director of Computer Science and Mathematics at Brookhaven National Laboratory. In **Welcome to the Adult World, OpenMP**, Barbara discusses the early success of OpenMP and why it's likely to remain a vital parallel programming model for years to come.

The **Intel® HPC Developer Conference** (November 11-12 in Denver, CO) and **SC17** (November 12-17, also in Denver) are just around the corner, so this issue contains three articles devoted to HPC. We explore **Intel® Cluster Checker** in **Is Your Cluster Healthy?** This component of Intel Parallel Studio encapsulates many best-known methods and system diagnostics to keep clusters operating efficiently. Several BIOS options can affect application performance, but it's difficult to change these options on demand in a production cluster environment. Learn about a technique to enable on-demand BIOS configuration changes in **Optimizing HPC Clusters**. And **Effectively Using Your Whole Cluster** presents a case study of HPC application tuning using several of the tools in Intel Parallel Studio XE.

The Heterogeneous Parallel Computing Future

The future is heterogeneous. (Actually, CPUs and GPUs have existed within the same system—and even on the same processor die—for many years now, so heterogeneous computing is already here.) Just as multicore processors have made parallelism ubiquitous, it won't be long before CPU, GPU, FPGA, ASIC, etc. coexisting within the same system makes heterogeneous parallelism ubiquitous, too. I used to worry about the heterogeneous future, but new parallel programming models will make it easier to map computations to the most efficient processor architecture. The **Intel® Threading Building Blocks** Flow Graph API is one such approach.

This API has already been covered in *The Parallel Universe* (see "Heterogeneous Programming with Intel® Threading Building Blocks" in our **special issue**), so I won't discuss it here, but this issue's feature article, **Driving Performance with Intel® Advisor's Flow Graph Analyzer**, gives an in-depth look at the Flow Graph Analyzer technology preview feature in Intel Parallel Studio XE. We use an autonomous driving application to illustrate the flow graph computation and analysis.

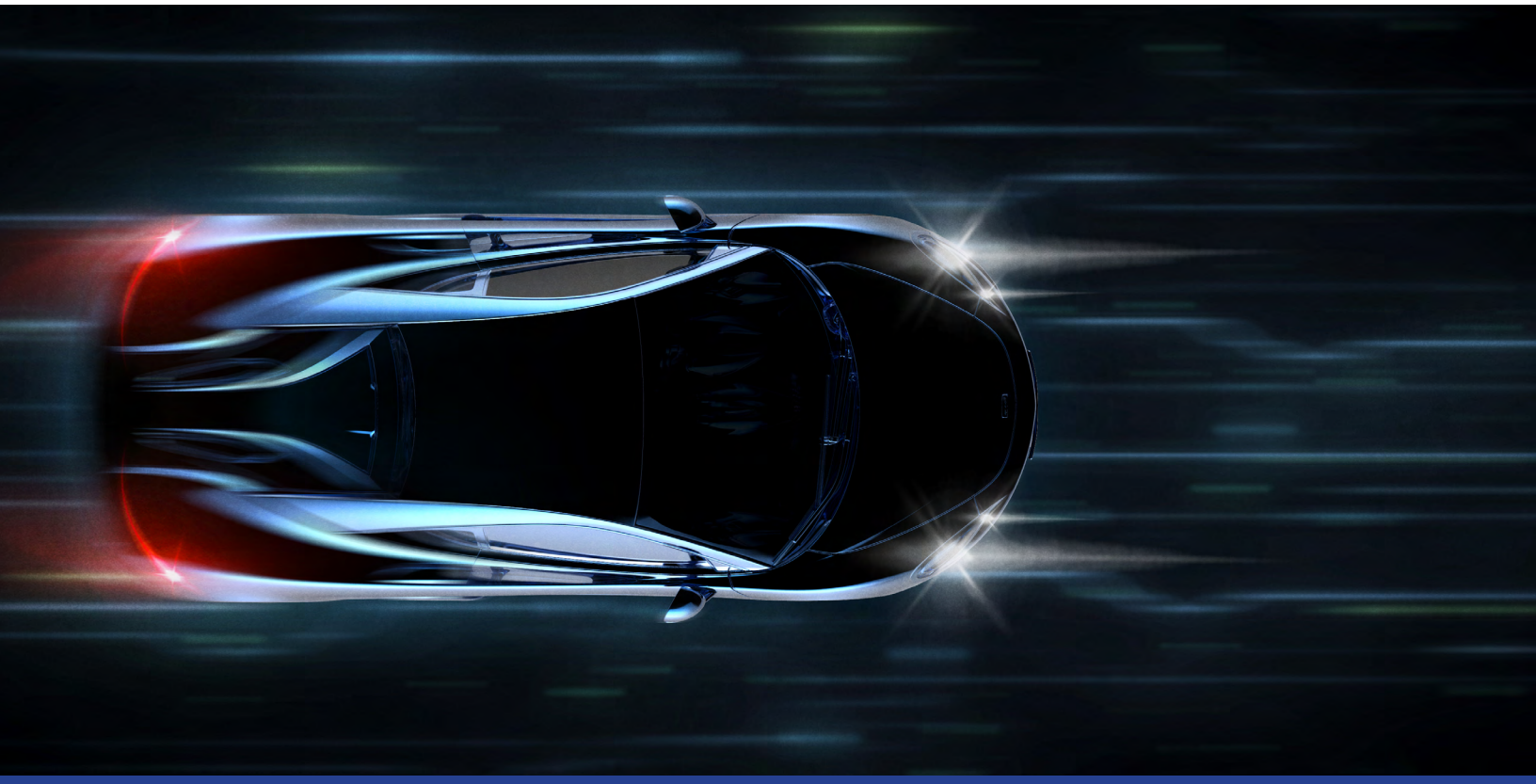
The Parallel Universe welcomes back its founding editor, James Reinders, to continue the theme of heterogeneity. In **Enabling FPGAs for Software Developers**, James and Bernhard Friebe discuss FPGA programming from a software, rather than a hardware, development perspective. Look for a follow-up article on FPGA programming in our next issue.

Finally, we close out this issue with a detailed look at what's new in the AVX-512 instruction set architecture. **Tuning for Success with the Latest SIMD* Extensions and Intel® Advanced Vector Extensions 512** discusses best practices in performance tuning with the new SIMD language extensions for AVX-512 and the latest support in the Intel compilers for the **Intel® Xeon® Scalable processors**.

Coming Attractions

We're working on a wide range of topics for future issues of *The Parallel Universe*, including FPGA programming, Java* performance tuning, new Intel Parallel Studio features, and much more. Be sure to **subscribe** so you won't miss a thing.

Henry A. Gabb
October 2017



DRIVING CODE PERFORMANCE WITH INTEL® ADVISOR'S FLOW GRAPH ANALYZER

Optimizing Performance for an Autonomous Driving Application

Vasanth Tovinkere, Architect, Intel® Flow Graph Analyzer; Pablo Reble, Software Engineer; Farshad Akhbari, Perceptual Computing Technical Lead; and Palanivel Guruvareddiar, Perceptual Computing Software Architect; Intel Corporation

The advanced driver assistance systems (ADAS) and autonomous driving technologies deployed in modern cars rely on environmental awareness through sensors such as radar, LIDAR, and cameras. The underlying machine learning or deep learning algorithms that provide this awareness are compute-intensive—and become even more demanding as the sensor resolution grows. For example, detection and classification of objects in a video stream requires inspection of every frame for objects of interest at all possible aspect ratios and sizes. A real-time object detection system also demands a lot of compute resources. And the demand only increases as we approach near-perfect systems.

These systems often involve a process of proposal generation (i.e., they propose and then analyze potential regions of interest in an image). Fortunately, proposal generation is repetitive and can be parallelized to improve performance. The expressed parallelism is scalable, since proposal generation can include the entire image as a single proposal all the way down to each pixel in a frame as a separate proposal. Since there are no dependencies among the processing of proposals, they can be processed in batches as system resources allow, and as they become available (i.e., in a pipeline fashion).

In this article, we explain how to design and analyze the parallelism in such an application using **Intel® Advisor**'s Flow Graph Analyzer (FGA), a tool from Intel that supports parallel applications using the **Intel® Threading Building Blocks (Intel® TBB)** flow graph interface. Intel TBB is a widely used C++ template library that enables developers to easily create parallel applications to take advantage of multicore architectures and heterogeneous systems. The flow graph interface was introduced in Intel TBB in 2011 to exploit parallelism at higher levels by providing efficient implementations of dependency graphs and data flow algorithms. [**Editor's note***:* "Intel® Threading Building Blocks Celebrates 10 Years," in ***The Parallel Universe* special issue**, includes several interesting articles on the evolution and future directions of the Intel TBB API.]

Even though Intel TBB makes it easier to express parallelism, ADAS and autonomous driving applications are complicated and may contain many interconnected algorithms and layers of parallelism—making them challenging to both design and tune. FGA, now available as a technology preview feature in Intel Advisor, makes this task more manageable. We'll discuss how to use FGA's features as we go through the analysis of an **OpenCV***-based autonomous driving example that is implemented using the Intel TBB flow graph API.

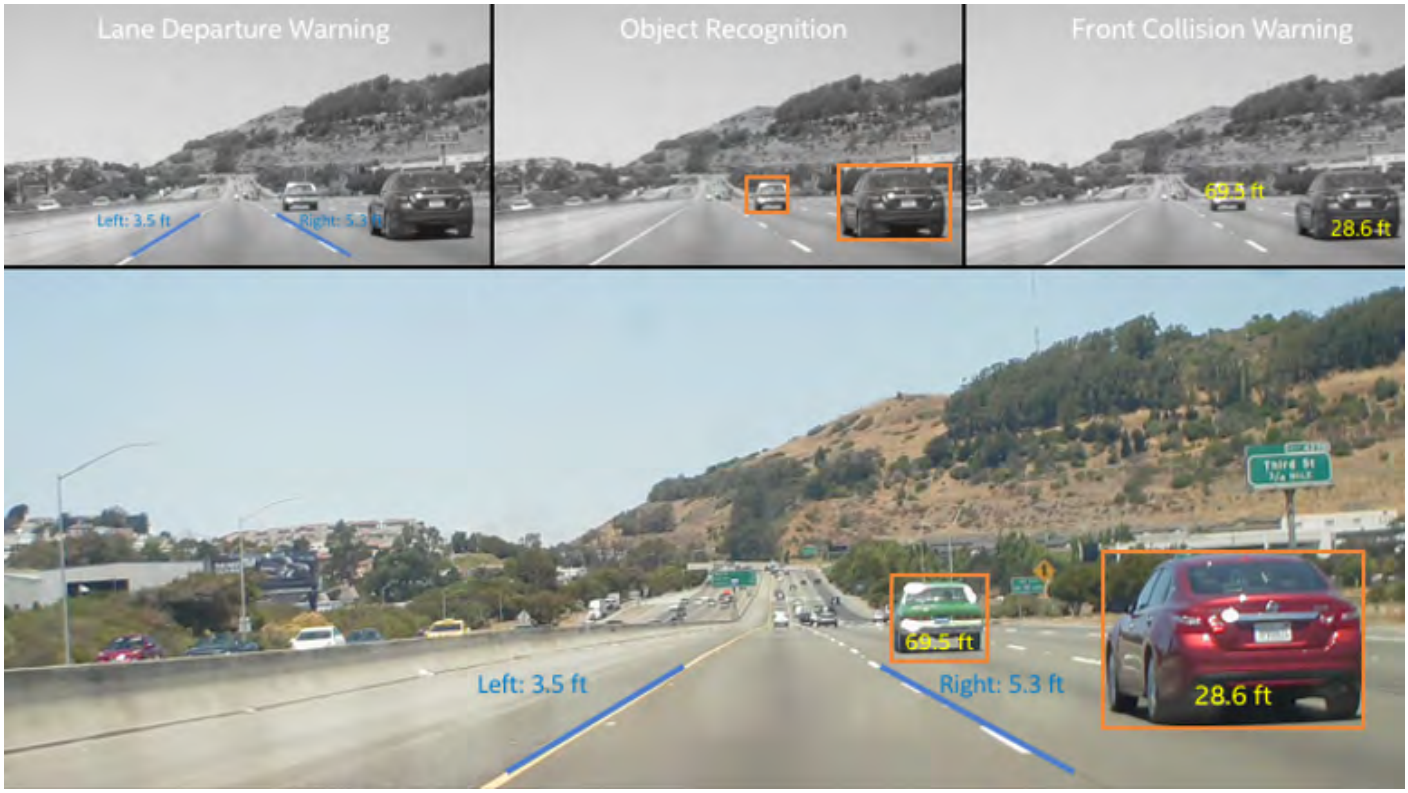
The Advanced Driver Assist Example

We'll use a demo framework for an ADAS as a running example. Our example framework supports two modes of operation:

1. **Online mode**, where the framework receives a live feed from the camera sensor mounted on the rearview mirror and captures data from the front of the car.
2. **Offline mode**, where recorded frames are read from a file to simulate the camera sensor functionality.

Incoming compressed camera frames are decoded and forwarded to a video input processing module that performs post-processing of the frames to ensure they are ready to be consumed by the computer vision algorithms.

The post-processed data is then broadcast to the registered computer vision use-cases in the framework (e.g., object recognition, lane departure warning, front collision warning). These use-cases can be executed independently (in parallel) on the same frame. And many of them internally use a process of proposal generation that allows for parallelization at the use-case level as well. Often, the use-cases are implemented using OpenCV* algorithms that already contain loop-level parallelism. We therefore have a framework with multiple levels of parallelism. The framework is extensible, since it allows new algorithms to be registered through plugins. The computer vision algorithms operate on the decompressed frames and produce metadata associated with detected objects/lanes and alerts for lane departure and front collision warnings. A combined module fuses all the information together and overlays it on top of the decompressed frame, which is then rendered for display, as shown in the colored highlights of **Figure 1**. The smaller grayscale images at the top highlight the output of each use-case.



1 **ADAS application example**

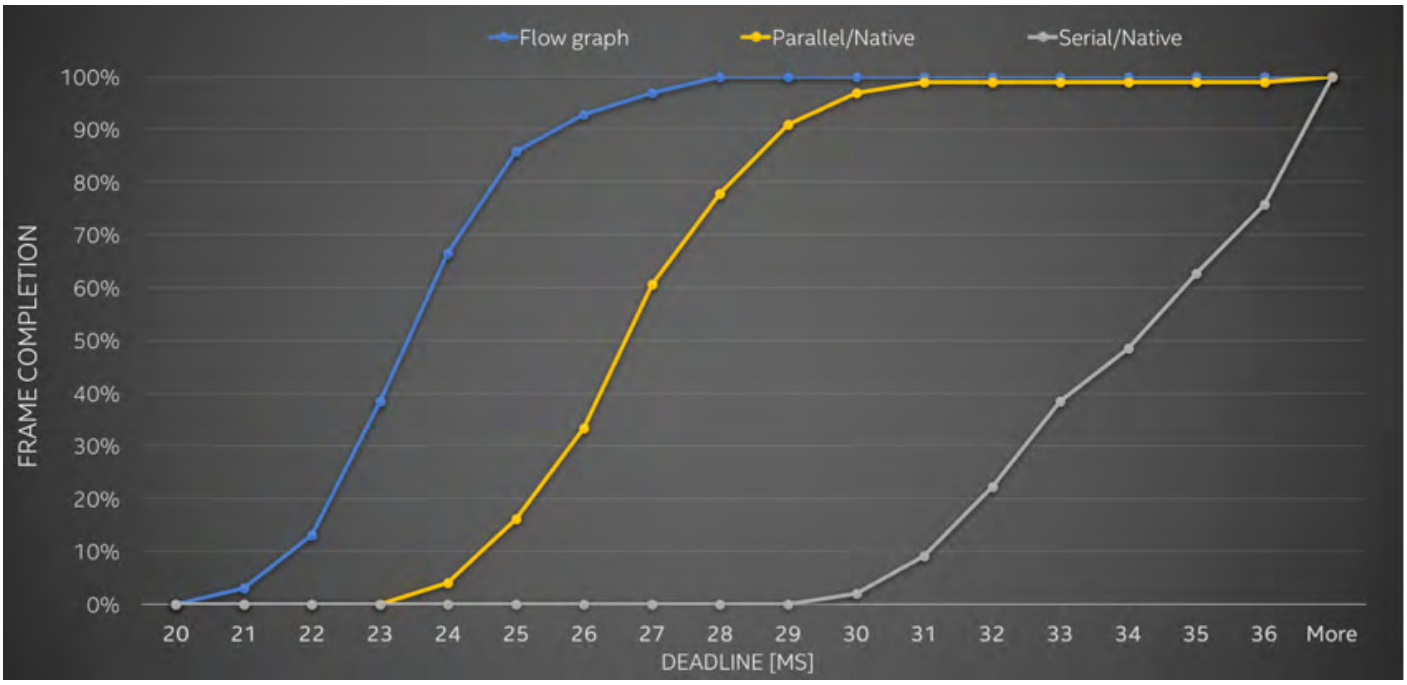
Evolution of the Parallel Framework

In general, a key performance indicator for a video processing application is the frame completion rate, which means how many frames are completed if there is a fixed time budget for completing each frame. If the processing of a frame exceeds the budget, it's dropped. So a completion rate of 70 percent for a 24 ms deadline means that 70 percent of frames can be completed given this deadline, while 30 percent must be dropped.

We examined three implementations during our optimization of this framework:

1. The original serial/native implementation
2. A parallel/native implementation
3. An Intel TBB flow graph implementation

Figure 2 shows the frame completion rates for these three implementations, demonstrating that shorter deadlines become feasible as optimizations are applied.



2 **Comparison of frame completion rates**

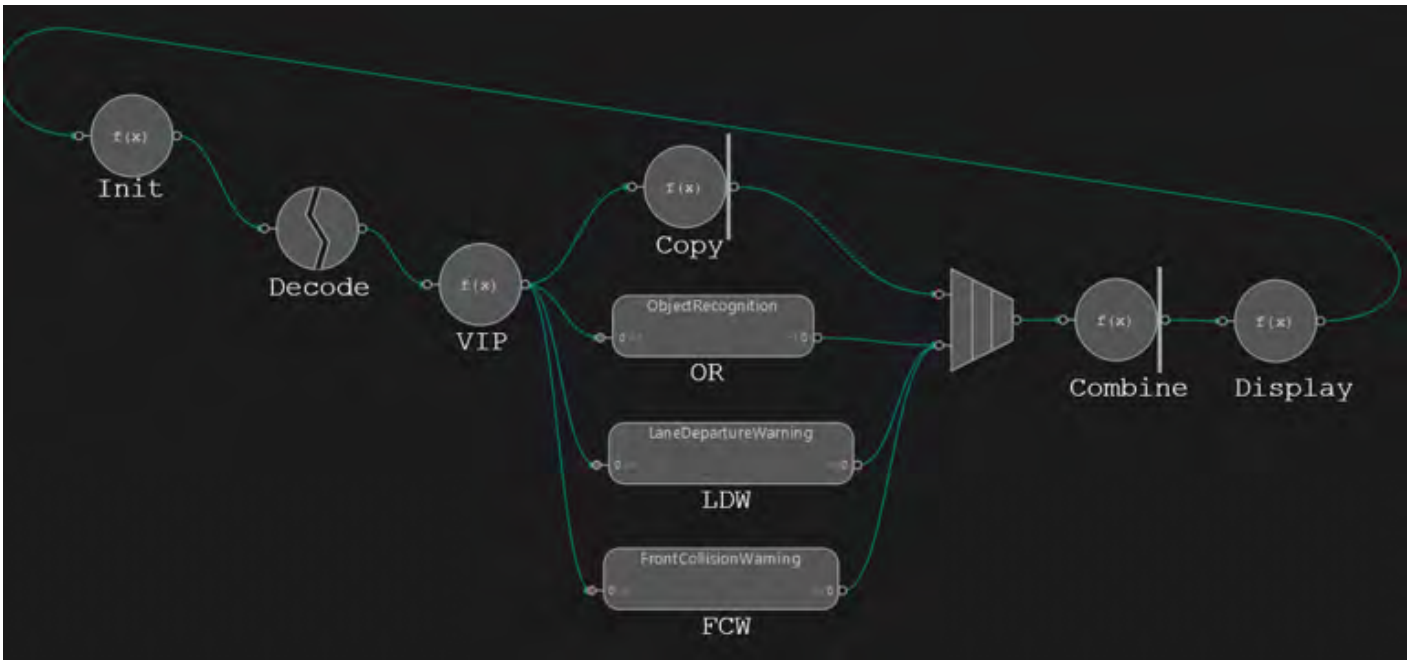
The serial/native implementation executes the use-cases in a serial fashion on a multi-core processor. Many of the use-cases are implemented using OpenCV, and even though they may contain nested, loop–level parallelism, this approach results in poor utilization of the cores in the system. The exposed parallelism of the individual algorithms is limited, which directly impacts the scalability of the application.

The parallel-native implementation was task-based, allowing concurrent execution of algorithm modules using standard threads. However, this causes oversubscription because threads are being created for the tasks in addition to the Intel TBB threads created within OpenCV. To have a truly flexible and extensible framework that avoids oversubscription, the flow graph implementation uses Intel TBB at both levels.

Figure 3 shows the advanced driver assist framework implemented as an Intel TBB data flow graph. Execution starts with a standard flow graph function node (Init) which encapsulates an initial setup routine that is applied to each frame. Subsequently, interaction with an external activity, the actual decoding of each frame, is expressed through an asynchronous node (Decode). The video input processing (VIP) node broadcasts the input image to each use-case (i.e., object recognition, lane departure warning, front collision warning) and a multifunction node (Copy). This node performs two actions:

1. **Stores** the unmodified input image for a later merge with the output of the use-cases.
2. **Generates** a token for each use-case.

The next node generates a tuple of this token and the output buffer, which is subsequently consumed by the multifunction Combine node. The highlights in **Figure 1** are outputs from the different use-cases represented



3 **Graph depicting the autonomous driving example framework**

by flow graph composite nodes in the center of **Figure 3**. In our example, object recognition (OR), lane departure warning (LDW), and front collision warning (FCW) are implemented using OpenCV. The Display node in our graph displays the output image. [**Editor's note***:* The full graph node attributes are beyond the scope of this article. See the **Intel Threading Building Blocks Flow Graph** documentation for API details.]

The major advantage of our data flow design is that more parallelism can be expressed, which improves performance. Also, our implementation doesn't require any modification of the existing use-cases, since they can be represented as single grouped function of nodes that just invoke their process_frame() function. That way, the flow graph implementation is backwards-compatible to all existing framework plugins.

Next, we'll analyze the performance of this flow graph using the features available in the Intel FGA tool to understand more about its behavior and discover that while it currently outperforms previous implementations, there is still room for improvement.

Overview of Intel FGA

Figure 4 shows Intel FGA visualizing a graph and its associated performance trace data. The tool has five viewing areas:

4 **Intel FGA showing a captured graph and associated trace data**

1. **Menu/Toolbar area**: Exposes the operations for designing, manipulating, visualizing, and analyzing graphs.
2. **Index area**, divided into three categories. Designer mode shows available node types for building new graphs. Hierarchical View represents the graph topology as a hierarchical tree. And Analysis mode (**Figure 4**) represents graph performance as squares in a tree map. (In this article, we focus on the Analysis mode.)
3. **Canvas area**: This is the primary graph viewing area where graphs can be constructed or displayed. This area supports common editing operations to facilitate the design of new graphs.
4. **Reports area**: This houses all basic reports such as graph and node properties and reports that are generated by analytics, such as semantic rule checks and critical path algorithms.
5. **Chart area**: This area displays performance statistics and execution trace timeline charts for graphs that have associated trace data. This data is automatically captured by the trace collector, which can be invoked from the GUI or as a command-line utility.

In Intel FGA, the viewing areas and modes can be used to create two main workflows for design and analysis:

1. **The design workflow** uses a drag-and-drop paradigm for interactively constructing Intel TBB flow graphs.
2. **The analysis workflow** captures performance data from Intel flow graph applications.

The tool provides the ability to visualize and interact with the performance data. We use this analysis workflow to examine the performance data collected for our ADAS example.

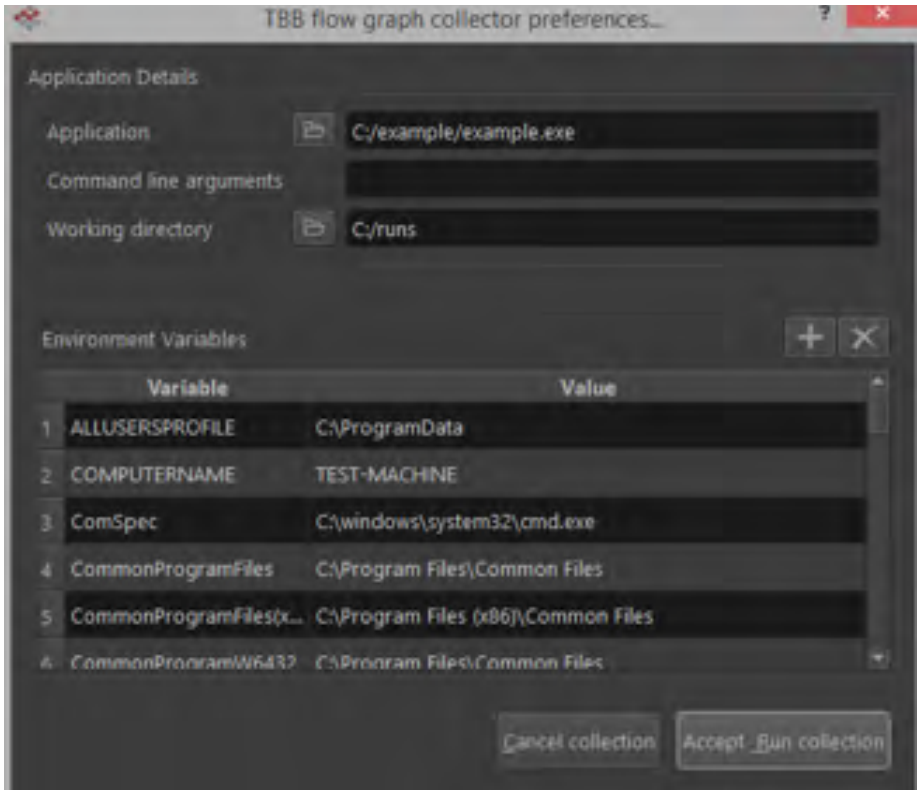
Performance Analysis of the ADAS Example

To optimize the performance of our example, we first need to identify the bottlenecks. Intel FGA allows developers to capture the graph topology and task execution traces from running Intel TBB flow graph applications. **Figure 5** shows the trace collection dialog window that allows users to specify the application to trace. When data collection is finished, two files are generated:

1. **GraphML file** describing the topology of the graph that was just executed
2. **TraceML file** with the task execution traces from the graph

Finding the Most Important Parts of the Application

A common first step in analyzing application performance is identifying where the majority of the time is spent in an application. Intel FGA has two features to assist with this step.

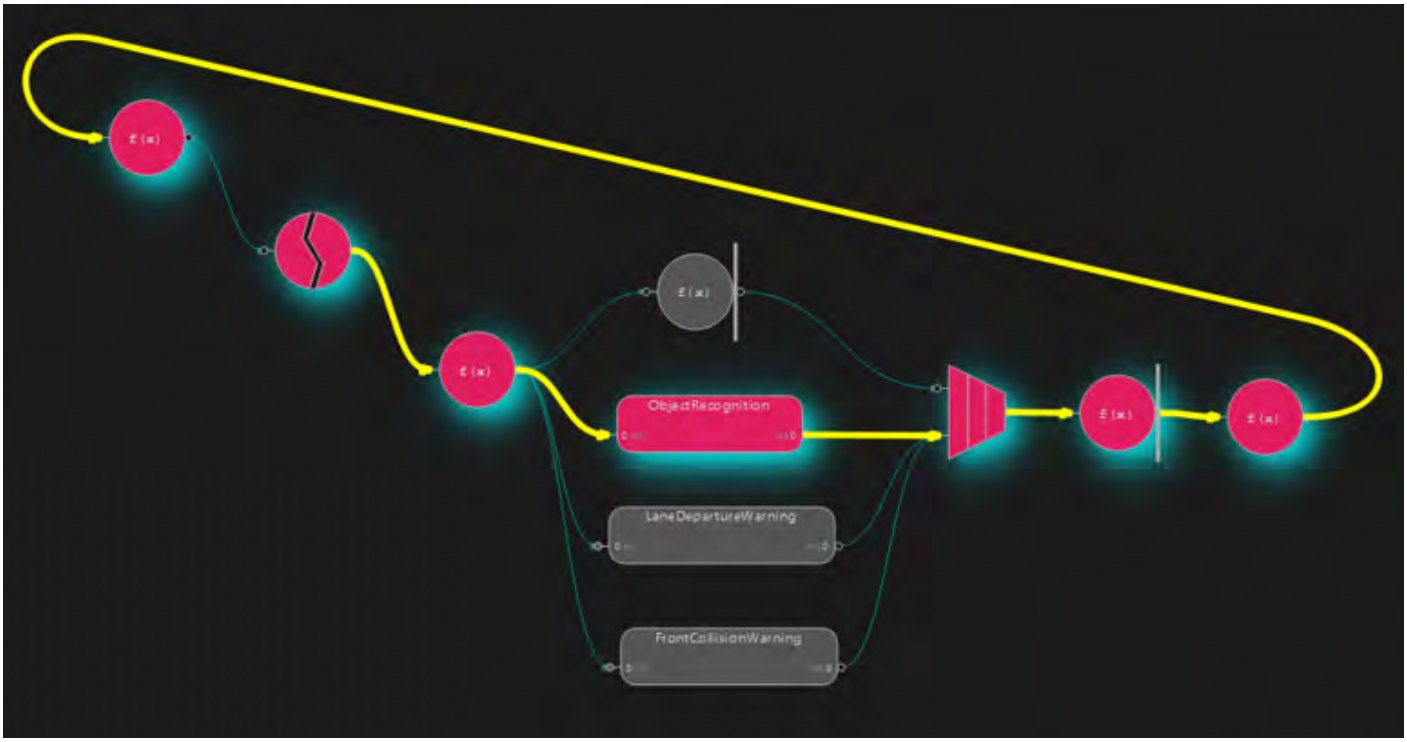


5 **Intel FGA trace collection dialog**

First, the tree-map view shows the total CPU time represented as a large rectangle that is subdivided into smaller rectangles representing the nodes in the graph. The area of the nodes is proportional to the total CPU time consumed by the node. Colors reflect the concurrency observed when the node was executing (red means low concurrency). **Figure 6** shows the tree-map view of our ADAS example. From this figure, we can conclude that the use-cases require different amounts of time to compute and that OR dominates the CPU time of the framework. The differences in execution times for the use-cases supports our decision to allow each use-case to send its results to the overlay node independently. Any kind of barrier might create a load imbalance.

Intel FGA can also calculate the critical path through an application and project it onto the flow graph's topology. This second view is shown in **Figure 7**. The critical path for our ADAS example highlights the key set of nodes to consider for optimization. From the combination of tree map and critical path views, we now know that not only is OR the most time-consuming node individually, it also lies on the most important path through the application. The performance of the graph is therefore limited by the performance of OR, so it should be our highest priority for optimization. Sometimes, it may be necessary to determine which algorithms are candidates for offloading to accelerators to meet a strict performance specification.

6 **Tree-map view**



7 **Screenshot showing the critical path of a graph**

Analyzing the Performance Characteristics of the Bottleneck

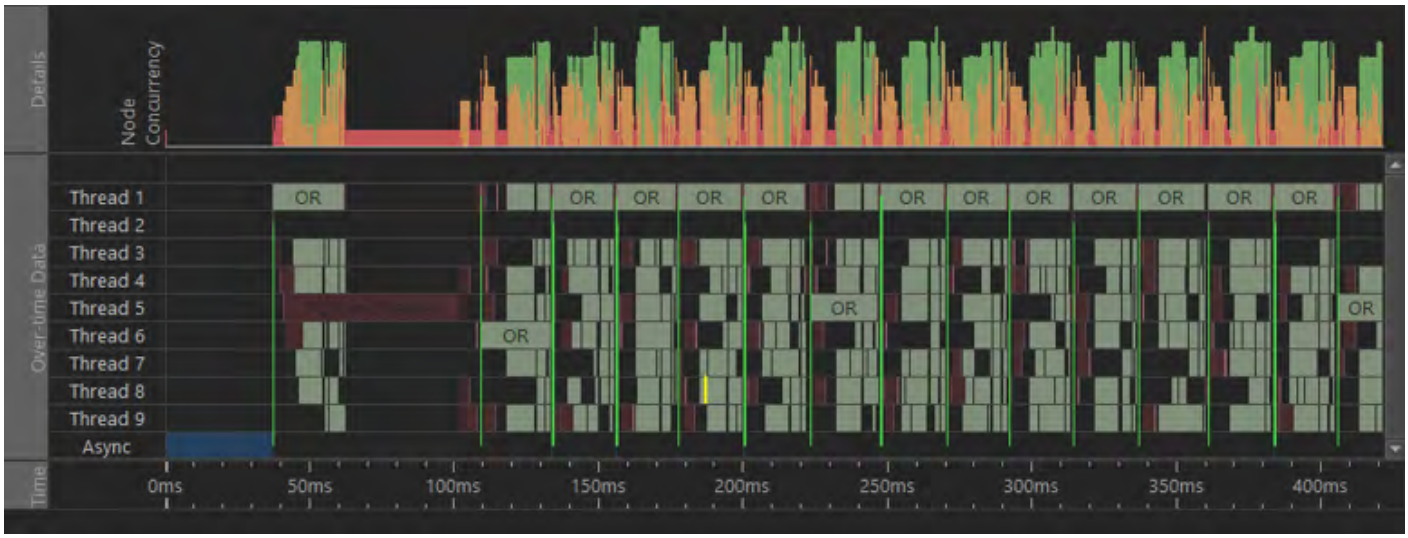
Now that we've identified that OR is the most important algorithm, we can further inspect its performance by looking at the timeline views in Intel FGA. The over-time task data grouped by thread provides the raw view of the collected traces. In this chart, we can see tasks executed by each thread and their duration. **Figure 8** shows the zoomed-in view of the data for one frame of processing and the legend describing the colors. The tasks are colored based on the type of task. Graph tasks are colored based on the task duration, with a lighter color used to highlight tasks that are small relative to the cost of scheduling a task. Nested parallel tasks and asynchronous tasks are shown in blue.

8 **Over-time trace charts showing one frame of data and the associated color legend**

The graph task colors are quantized using a gradient that ranges from 1µs to 1ms so developers can identify tasks (hence nodes) that have execution times that are on the lower end and may be affecting performance scaling. Selecting a task in the timeline highlights the node in the graph on the canvas. This chart allows us to focus on the tasks that are too small to be spawned on a thread and enables us to debug the performance of the graph when we see low concurrency. Similar information is available for the tasks spawned from nested parallel algorithms.

From **Figure 8**, we see that while threads are busy at times during OR, there is still room for improvement. The blue tasks indicate that there is nested parallelism that uses Intel TBB, but the black regions indicate times during which the cores are idle. Since OR is the dominant computation in our example, we may want to see which tasks correspond to it. All tasks are shown in **Figure 8**. If we want to focus only on the tasks that fall on the critical path, we can use selection-based highlighting to show tasks on the critical path.

Figure 9 shows the timeline view when only tasks related to the critical path are highlighted. Note that displaying nested tasks is currently an experimental feature of Intel FGA that requires manual instrumentation. From this figure, we can see that the majority of the tasks in the timeline can be mapped to the OR node. The core idle times, indicated by the black regions, can be addressed by shrinking the time taken to compute OR, thereby eliminating the idle time or by scheduling additional algorithms that could execute in parallel and keep the idle cores busy.



9 **Critical path shown on the execution traces**

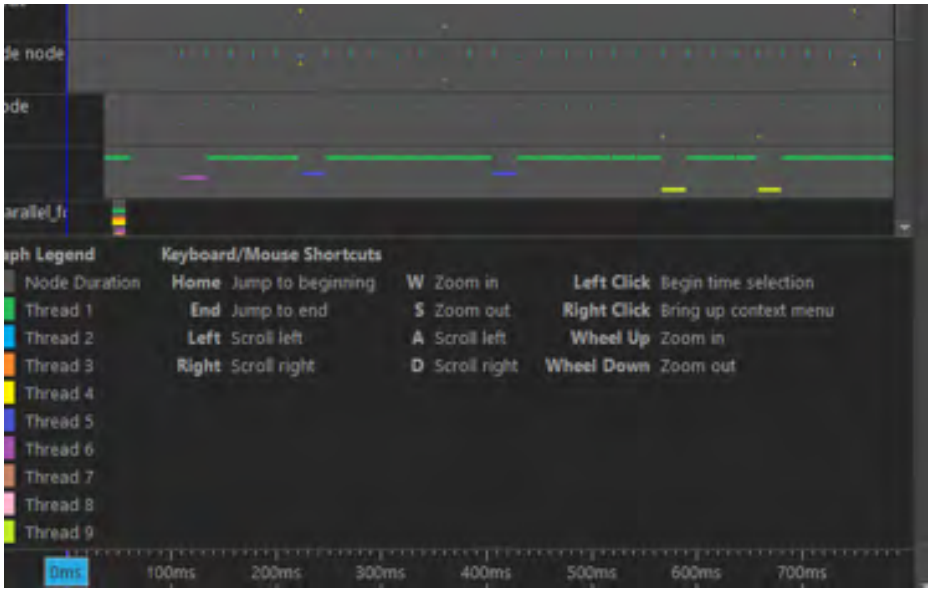
Let's inspect the OR node execution traces for the frame shown in **Figure 9**. Zooming into this frame reveals four nested parallel algorithms executed by the OR node in this frame (**Figure 10**). Additional information is presented for the parallel algorithms to help us determine if these algorithms are executing efficiently. The Statistics view has per-node data presented as a table that shows the performance statistics of each node at the graph level. In our analysis of the OR node, we're more interested in the algorithms that are executed in each node. **Figure 10** shows the per-algorithm data for the four parallel algorithms that were executed in our selected frame. We can see that the efficiencies of OR::tbb149 and OR::tbb213 are reasonably high and require less performance tuning. OR::tbb261 and OR::tbb284, on the other hand, exhibit poor efficiency, possibly because of the small number of tasks executed by these algorithms. This could result in load imbalances that reduce parallel efficiency and scalability.

Graph | Algorithms

Severity	For	Efficiency(%)	Task Count	Duration (ms)	CPU Time(%)	Other Time(%)	Task Size (min)	Task Size(max)	Active Elsewhere(%)
tbb_parallel_for(tbb1474)	Results(7)	83.87	103		83.87	16.13	0.263844	0.819408	1.70
tbb_parallel_for(tbb149)	Results(3)	94.54	46		94.54	5.46	0.317937	0.959297	3.70
	[Worker Thread]	100.00	19	9.485726	100.00	0.00	0.317937	0.830145	11.11
	[Worker Thread]	94.87	14	9.938140	94.87	5.13	0.509535	0.959297	0.00
	[Worker Thread]	88.75	13	9.341695	88.75	11.25	0.436504	0.940199	0.00
tbb_parallel_for(tbb213)	Results(4)	86.21	30		86.21	13.79	0.211302	0.737325	5.41
	[Worker Thread]	100.00	9	2.931293	100.00	0.00	0.211302	0.409358	21.63
	[Worker Thread]	86.06	9	3.085395	86.06	13.94	0.229712	0.485408	0.00
	[Worker Thread]	75.83	7	2.723886	75.83	24.17	0.277538	0.519274	0.00
	[Worker Thread]	82.95	5	2.914032	82.95	17.05	0.481382	0.737325	0.00
tbb_parallel_for(tbb261)	Results(3)	71.74	10		71.74	28.26	0.105216	0.241558	13.11
	[Worker Thread]	100.00	4	0.670851	100.00	0.00	0.105216	0.192787	39.33
	[Worker Thread]	47.35	2	0.497640	47.35	52.65	0.222739	0.232367	0.00
	[Worker Thread]	67.86	4	0.714158	67.86	32.14	0.109674	0.241558	0.00
tbb_parallel_for(tbb284)	Results(4)	48.84	6		48.84	51.16	0.067957	0.119980	12.71
	[Worker Thread]	100.00	2	0.194970	100.00	0.00	0.067957	0.119980	50.83
	[Worker Thread]	22.24	1	0.084998	22.24	77.76	0.084998	0.084998	0.00
	[Worker Thread]	24.74	1	0.094556	24.74	75.26	0.094556	0.094556	0.00
	[Worker Thread]	48.39	2	0.226341	48.39	51.61	0.082364	0.102569	0.00

10 **Graph statistics table showing algorithm statistics for OR::tbb149, OR::tbb213, OR::tbb261, and OR::tbb284**

Since Intel TBB uses work-stealing to schedule tasks, we may also be interested in seeing which threads execute particular nodes. It is easy to see this in Intel FGA by switching to a per-node view of the timeline (**Figure 11**). We can now see that the OR node is often executed by the same thread (colored green), but that at times other threads execute it.



11 **Tasks grouped by nodes**

In addition to the timeline, algorithm statistics, and graph topology views, we can also view per-node data in tabular form (**Figure 12**). This allows developers to sort the data and determine the best regions for optimization. Selecting a row will highlight the corresponding node in the graph view.

Graph | Algorithms

Severity	For	Node Name	Instance Count	In-degree	Out-degree	Total Time (ms)	Avg. Task Duration (ms)	Std. Dev.	Average Concurrency
g1	Results(14)								
g1::n5	Results(3)								
g1::n5::n8	Results(3)								
	Node	g1::n5::n8::n10	153	1	0	3.249428	0.021238	0.01	3.60
	Node	processFCWFrame	153	1	1	547.956726	3.581417	0.71	3.81
	Node	initFCWFrame	51	0	1	11.072482	0.217107	0.20	2.20
	Node	FCW init frame	51	0	1	143.948059	2.822511	8.20	2.73
	Node	g1::n5::n7	51	1	0	1.310975	0.025705	0.01	3.13
g1::n0	Results(4)								
	Node	LDW: Init Frame	51	0	2	10.974422	0.215185	0.11	3.86
	Node	LDW: Process Left	51	1	1	56.322594	1.104365	0.72	3.79
	Node	LDW: Process Right	51	1	1	41.114304	0.806163	0.65	3.67
	Node	LDW: Output Frame	51	2	0	0.991448	0.019440	0.01	3.62
	Node	Init node	52	1	1	1.019682	0.019609	0.01	1.00
	Node	Decode node	102	1	1	7.502239	0.073551	0.20	1.82
	Node	VIP node	51	1	4	2.115185	0.041474	0.02	1.30
	Node	g1::n15	51	1	1	1.561082	0.030609	0.03	3.85
	Node	g1::n16	51	1	1	1.055234	0.020691	0.01	3.66
	Node	g1::n17	51	1	1	1.357667	0.026621	0.01	3.90
	Node	g1::n18	51	1	1	1.276426	0.025028	0.02	3.51
	Node	OR	9792	1	1	6320.552246	0.645481	1.78	5.41
	Node	Copy to node	51	1	1	19.666084	0.385610	0.18	3.85
	Node	Display node	51	1	1	48.786430	0.956597	0.27	1.00
	Node	g1::n22	0	4	1	0.000000	0.000000	0.00	0.00
	Node	Draw overlay node	153	1	1	12.802725	0.083678	0.06	3.03

12 **Graph Statistics table showing per-node statistics**

Conclusions

Intel FGA gives developers a comprehensive set of tools to examine, debug, and analyze Intel TBB flow graphs. Applying it to our ADAS example application, we were able to quickly identify the most important nodes for optimization. We were also able to determine that while there was some use of nested parallelism in the OR use-case, there are still idle cores that we may be able to use to find additional parallelism.

intel Software

THE ULTIMATE WORKHORSE

As one of nature's most extraordinary athletes, a hummingbird beats its wings at an amazing 200 times per second. Now you can deliver outstanding performance of your code to achieve 187x faster code[1] through vectorization, thread-safe algorithms, industry-leading compilers and performance-tuning tools. Let your app soar.

intel PARALLEL STUDIO XE

Try it today:
software.intel.com/intel-parallel-studio-xe

#PurePerformance

[1]Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit intel.com/performance.

See software.intel.com/en-us/intel-parallel-studio-xe/details#configurations for more details.

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.

WELCOME TO THE ADULT WORLD, OPENMP*

After 20 Years, It's More Relevant than Ever

Barbara Chapman, Professor, Stony Brook University, and Director of Computer Science and Mathematics, Brookhaven National Laboratory

OpenMP* entered the world in 1997, at a time when:

- Fortran* was still the clear language of choice for technical computing.
- The term multicore had not entered the vernacular.
- Computer vendors were beginning to market desktop systems configured with two or more processors that shared memory (SMPs).

Many in the field expected that SMPs would remain modestly parallel systems, with perhaps up to four processors, as a result of technology limitations exposed by experience with shared-memory platforms in the 1980s. Yet large distributed, shared-memory (DSM) machines were available, software DSM was

actively being researched, and the Tera MTA* system demonstrated another form of massive multithreading.

Together with Mats Brorsson (who was then at the University of Lund), I organized the first European Workshop on OpenMP (EWOMP) in 1999. Although the standard had been around for less than two years at the time, it attracted plenty of papers and a lively audience. A large fraction of the participants were application developers who were already using OpenMP for scientific research on large DSM platforms and reporting, in some cases, astonishingly good results. Experimentation with hybrid programming that combined MPI and OpenMP was already a topic of interest. Attendees demonstrated a good deal of enthusiasm about the interface. They enjoyed both the simplicity of creating OpenMP programs and the relative ease with which the behavior of the resulting parallel code could be understood.

More workshops were initiated that attracted researchers and scientific application developers who had a specific interest in OpenMP. Participants from industry presented a variety of application codes, ranging from databases through engineering product design. At the North American Workshop on OpenMP Applications and Tools (WOMPAT), the first of which took place at the San Diego Supercomputing Center in 2000, the program included a mix of presentations on application experiences, proposals for extensions (including features for data placement on NUMA systems developed by Compaq), implementation experience, and performance modeling. The Workshop on OpenMP Experiences and Implementations (WOMPEI) was organized in Japan, where the active research and development on OpenMP included serious efforts to translate OpenMP to clusters.

Overall, it proved difficult to attract application developers from business and industry to the workshops. In the words of one major user, whose company's hardware procurement decisions were largely dictated by the requirements of their large-scale OpenMP programs, he didn't need to go to a conference on OpenMP because "it just works." Some users in industry were satisfied with a modest to moderate application speedup, potentially accomplished by parallelizing a piece of an overall computation. Computer scientists, in contrast, were discovering more of the challenges associated with shared memory in practice and proposing parallel program representations in the compiler, developing more efficient translations, and devising enhanced runtime techniques. In addition, they created benchmarks that helped application developers better understand the overheads incurred by different features, as well as track the quality of individual implementations.

In 2005, the various regional workshops were merged into the International Workshop on OpenMP (IWOMP), which has since then brought together researchers and members of the language committees for several days each year to look at the latest ideas for features, implementations, and tools. IWOMP

contributions reflect the entire range of topics that influence the evolution of the specification: experience reports that highlight the good and the not-so-good, proposals for new features and implementation techniques, explorations of alternative means to overcome open challenges, consideration of new hardware technology, strategies for improving the performance of existing constructs, and much more.

While the initial version of the specification was produced via a careful evaluation of prior experience with shared-memory directives, today's demands call for a more comprehensive approach. Fortunately, scientific application developers and computer scientists have been actively exploring OpenMP enhancements ever since it was announced. Their work has been essential for the continued evolution of OpenMP. Yet in the early days, it was hard for researchers to directly contribute to the work of improving the specification. ARB vendors wanted to be absolutely certain that those involved in making key decisions had the best interests of the standard at heart and didn't provide inexpensive membership options for academia. This was partly a response to the commercial failure of the High Performance Fortran* (HPF*) standards effort, in which the drawbacks of standard design via broad community participation had become all too apparent.

To enable participation by researchers who were keen to contribute their time and expertise, and with strong encouragement by existing ARB members, I founded an organization called cOMPunity that subsequently joined the ARB. The initial members of cOMPunity included Mark Bull from EPCC; Eduard Ayguade from CEPBA, Barcelona; Dieter an Mey at the RWTH Aachen; Mitsuhisa Sato from the University of Tsukuba; and Rudi Eigenmann from Purdue University. One of the notable things about this group, other than the fact that most of them are still involved in some way, is the geographic diversity of their affiliations. Fortunately, it's now very easy for research organizations to join the ARB directly, and those with more than a passing interest in contributing to the standard can gain direct membership. The OpenMP experience has, moreover, demonstrated that researchers, too, can have a long-term interest in—and positive impact on—industry standards.

The computer systems on which OpenMP is deployed have undergone significant change since 1997. From a very small number of shared-memory processors, through the multicore revolution and the corresponding need for explicit parallelism in many applications, we are now in an era of heterogeneous many-core computing. The diversity in the kinds of cores configured on today's computing systems, the differences in their capabilities, as well as in the nature of their resource sharing and connectivity, have led to a large variety in the platforms that may be targeted. The core count continues to increase; the memory system is undergoing major change and diversification as the vendors attempt to overcome size, latency, and bandwidth limitations. There has also been change over the last 20 years in the nature of the applications using OpenMP. From a time when parallelism was primarily exploited in scientific codes, invariably written in Fortran, we have evolved into a period where an increasing variety of applications are

parallelized, and where they may be written in C, C++, Fortran, or some combination thereof. OpenMP has adapted to all of these changes.

From the outset, OpenMP developers chose to take on the challenge of remaining relevant in a rapidly changing world—and the OpenMP specification was never static. Under able and forward-looking leadership, it continuously evolved to fit in with its environment, from the addition of bindings for C and C++, via a much more careful specification, up to features that support the most demanding applications and systems today. The language committee has, moreover, been adept at identifying and integrating the best new ideas for parallel application development from a variety of sources.

From the outset, OpenMP developers chose to take on the challenge of remaining relevant in a rapidly changing world—and the OpenMP specification was never static.

One of the biggest advantages of OpenMP from the application developer's perspective is its portability. Once its features are understood, an application can be adapted for deployment on a variety of vendor platforms. Yet it's taken an enormous effort to ensure that OpenMP remains suitable for application needs across the different parallel computing systems available today. The language committee has tackled the difficult problem of facilitating execution across heterogeneous cores without shared memory, and even modified its mission statement accordingly. It's added features to support the use of SIMD components. It's enhanced tasking features to increase their expressivity and enable them to address a wider variety of coding problems. Features are desirable to help applications efficiently utilize new kinds of memory and to increase data locality in their codes. The language committee is already looking into it.

Large-scale application developers—who have traditionally been willing to invest significant effort to get their codes to run with very high efficiency on the largest computers on the planet—are beginning to voice a new demand for performance portability. Underlying this term is a growing expectation that their application code should execute well, with little to no modification, across a diverse range of parallel computing systems. This is a very reasonable demand in view of the cost of moving a code from one system to another. Yet OpenMP was conceived in an age where transparency of behavior was considered vital and many of its users continue to place a premium on this characteristic. If OpenMP is to be useful for performance-portable parallel application creation as well as for prescriptive parallel programming, a non-trivial effort will be required.

The single most amazing thing about OpenMP is that after 20 years, it's still relevant. Indeed, it's used more extensively than ever. And with implementations on all major platforms, it's broadly available. More user representatives than ever before participate in the committee meetings. Application developers are more vociferous, with specific and sometimes highly demanding requirements. Major agencies such as the U.S. Department of Energy, in its Exascale Computing Project, support efforts to ensure that OpenMP is able to meet their requirements at scale. The work to evolve the specification is driven by an active team of vendor representatives, along with participants from research laboratories and universities who have committed themselves for the long haul. For a compiler-based programming interface, this is huge.

Still, much remains to be done. Despite the complexity involved, high-quality implementations must be developed and made available faster. We need to help application developers transition codes written using other programming interfaces to OpenMP. We need more tools to help those who are creating high-performing, power-efficient code at scale. We must continue to support those who primarily want ease of use and maintenance benefits. With the growth in scope and complexity, there's a need for expanded training programs targeting users with different levels of expertise. With the growing sophistication of OpenMP application development, only a vibrant community of researchers, developers, trainers, and users can satisfy these demands. Fortunately, OpenMP has such a community.


intel
Software
FUTURE FORWARD
FALL TECHNICAL WEBINARS
Sharpen your skills.
Get expert answers.
Dive into new development areas.
October 25
9 a.m. PDT
ACCELERATING LOSSLESS DATA COMPRESSION CODE FOR CLOUD AND EDGE APPLICATIONS
Discover how Intel® IPP's highly optimized lossless data compression functions can improve system performance.
November 1
9 a.m. PDT
PARALLEL PROGRAMMING STANDARDS UPDATE: MPI*, OPENMP*, AND INTEL® TBB
Get an overview of these long-lived parallel programming workhorses, including their strengths and the types of problems each can help you solve.
November 8
9 a.m. PDT
BETTER, FASTER, AND MORE SCALABLE: THE MARCH TO EXASCALE
In 2018, Intel® MPI Library will upgrade to next-generation MPICH*—the exascale source used by Argonne National Labs. See why it matters.
REGISTER NOW >
BROWSE ARCHIVE >
For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation

ENABLING FPGAS FOR SOFTWARE DEVELOPERS

Boosting Efficiency and Performance for Automotive, Networking, and Cloud Computing

Bernhard Friebe, Senior Director of FPGA Software Solutions Marketing, Intel Corporation, and James Reinders, HPC Enthusiast

There couldn't be a better time to examine field programmable gate arrays (FPGAs) (**Figure 1**). A new era in computing is emerging thanks to the new programmability of FPGAs. With the onslaught of data in the world, there's an incredible need for the power-efficient computing available with custom silicon designs—but with the flexibility available with the flexibility of FPGAs.

In this article, we share our thoughts on what it takes to truly enable FPGAs for software developers across all fields—including three that we explicitly discuss:

1. Automotive
2. Networking (e.g., 5G)
3. End-to-end cloud computing (e.g., the data center)

1 **Intel® Stratix® 10 FPGA**

We dive a little deeper into enabling the data center by discussing the acceleration stack and the open programmable acceleration engine (OPAE). We don't dive into any particular domain, such as machine learning, which a complete solutions stack enables through encapsulation of FPGA IP (think library routines implemented to use an FPGA). However, we do provide some links to explain more about the new world of accessible FPGA programming that Intel is leading.

What's an FPGA?

An FPGA is essentially a blank slate waiting for us to draw a circuit design onto it. We do this by writing a description for the digital circuit we want, compiling it (FPGA developers call this synthesizing) into a configuration file (called a bitfile) and loading it into the FPGA. Once loaded, an FPGA behaves like the digital circuit we designed.

FPGAs are never permanently programmed. They don't remember their program (bitfile/bitstream) when powered off. In most systems, the FPGA is loaded at power-up—either from firmware on the board with the FPGA, or programmatically by the host processor. The FPGA can be reloaded any time we want to change it.

The highly parallel architecture of FPGAs means that running computations on an FPGA will give better performance, lower power, lower latency, and higher throughput than software. Imagine being able to configure a hardware realization of a function and use that in our program. As we'll discuss, the hottest topic in the FPGA world is making FPGA programming more accessible to software developers. Intel is leading the way to making this happen.

The Data Explosion

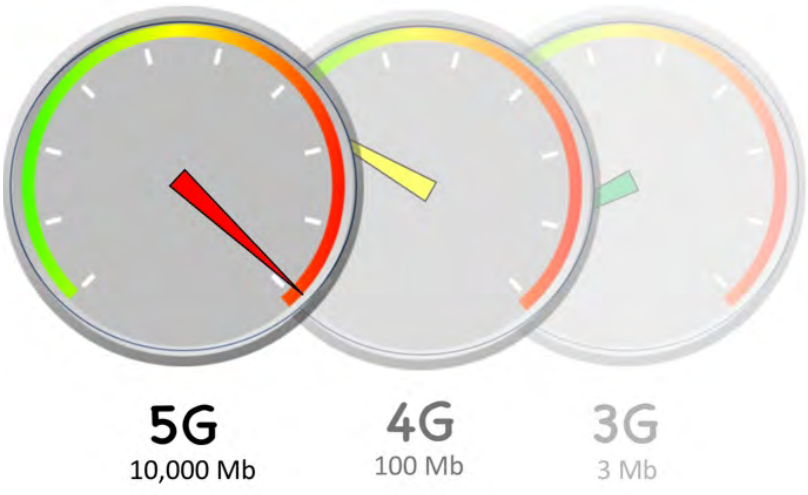
Three examples where Intel® FPGAs are key to delivering faster speeds, ultra-low latency, power-efficiency, and enormous flexibility are autonomous and assisted driving, accelerated networking (including 5G and software-defined networking, or SDN), and cloud computing (including high-performance deep neural network, or DNN inference). In all three usages, big data is part of why the parallel processing architecture of FPGAs is so critical.

Automated Vehicles

Automated vehicles, including those offering autonomous driving, are possible because of a combination of computation in the car, the networking, and the data center. Automated vehicles generate and use huge amounts of data to navigate safely. It takes about 1GB/second of real-time processing, while the time from sensing to reacting must be less than one second for safety reasons. For example, Intel processing power is part of the 2018 **Audi A8* Autonomous Driving System**. In that design, Intel FPGAs handle the object fusion, map fusion, parking, pre-crash, processing, and functional safety aspects of the self-driving car.

Networking and 5G

Networks are expanding rapidly, with 1000X increases in bandwidth, 100X increases in devices, and requirements for 1ms end-to-end round trips. FPGAs are critical in meeting these demands. With devices supporting 100Gb/second just gaining adoption, IEEE 802.3's 400 Gb/s Ethernet Study Group (started in 2013) is expected to result in a standard this year for 400Gb/second. With efforts well underway to standardize 5G, we are nearing a point where data carried on wireless networks will exceed that of purely wire-based networks (**Figure 2**).



2 **The move to 5G**

Cloud Computing

IDC has said that the total data in the world was about 4.4 zettabytes in 2013, 8.6 zettabytes in 2015, and will grow to 44 zettabytes by 2020, when it has been predicted that there will be as many as 50 billion devices connected to the Internet. Cloud computing has demands in the data center, on the edge, and everywhere in between. It demands scale, throughput, performance/watt efficiency, flexibility, and low latency. Again, FPGAs are critical in meeting these demands.

The Computing Imperative: More Parallelism, Less Power, Greater Flexibility

To make sense of this flood of data, we need to automate decision-making, have real-time insights into what connected devices are telling us, and present interactive and intuitive user connections to this data. Without accelerated computing, scale-out of many applications (e.g., artificial intelligence) will prove impractical. New computing requirements demand more parallelism, lower power consumption, and a degree of flexibility never before seen in accelerators. To meet this need, hardware platforms—from the edge to the cloud—have been evolving to include mixtures of CPUs and accelerators. FPGAs play a critical role in the trend toward heterogeneous computing platforms that are highly parallel, power-efficient, and reprogrammable. In short, FPGAs enable hardware performance with the programmability of software. However, the FPGA programming model has typically been hardware-centric. As FPGAs become a standard component of the computing environment, with users expecting the hardware to be software-defined, they must be accessible not just by hardware developers, but by software developers.

FPGA Programming for a Software World

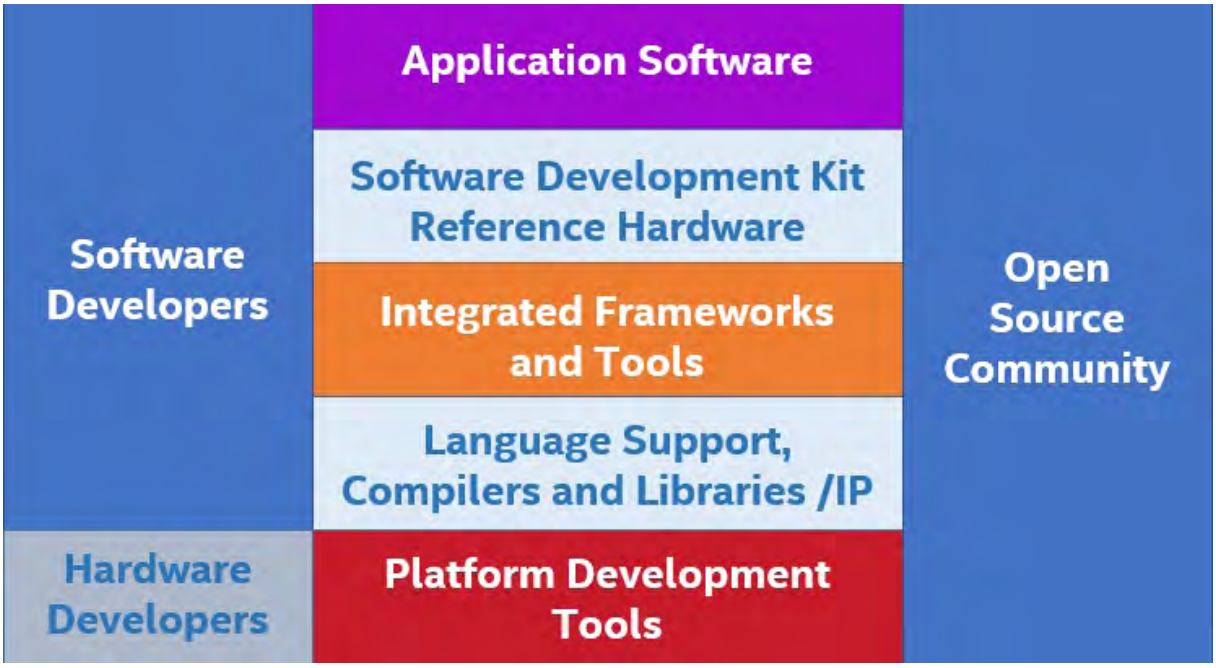
FPGAs have been around for years to solve hardware design problems. Their programmability was done exclusively in terms familiar to hardware designers instead of via any programming language designed for software development. New FPGA designs aimed at supporting software development instead of just hardware replacement designs, coupled with new software development tools, make FPGA programming worth a serious look by software developers.

This shift reminds us of the development of GPU programming methods, but with the advantage that FPGAs are not predesigned with a particular use in mind (i.e., graphics processing). This flexibility ushers in a new era in computing like none before it.

Completing the Solution Stack for FPGAs

Pioneers in using FPGAs as programmable accelerators have come a long way with very few tools to help them. They have lived with a foot in the world of hardware design as they programmed using a hardware description language (HDL) such as Verilog* or VHDL*. A few years ago, we surveyed people interested in FPGA development. A telling comment we heard was "There's a community of FPGA developers out there; the [various FPGA] forums are both pretty active; and there's a lot of literature available. That being said, it's a community of mostly hardware engineers—and a software developer might have trouble having their questions answered." While this is changing, the comment is still very important because fewer than one in 10 people surveyed felt comfortable programming in VHDL or Verilog despite expressing an interest in FPGAs. In the same group, over 90 percent of software developers knew C, C++, or OpenCL*, and over 75 percent said they would prefer to depend on libraries to exploit FPGA functionality. Therefore, it makes sense that Intel's next generation of FPGA tools focus on C, C++, OpenCL, and libraries.

The software stack continues to rely on hardware flows as always, but the complete solutions stack (**Figure 3**) provides programming layers that give efficient but software-friendly interfaces. These new methods don't abandon traditional FPGA tools—which remain available, and have improvements of their own. With a complete solutions stack for FPGAs, however, programming mirrors what we find in the CPU world—deep architectural expertise goes into compilers and libraries, which are accessible to programmers whose expertise is in other areas (i.e., science and engineering disciplines other than computer architecture). There is a separation of concerns between application developers using FPGAs via FPGA IP and those who develop the FPGA IP. A complete solutions stack makes this new way of thinking about FPGA programming possible.

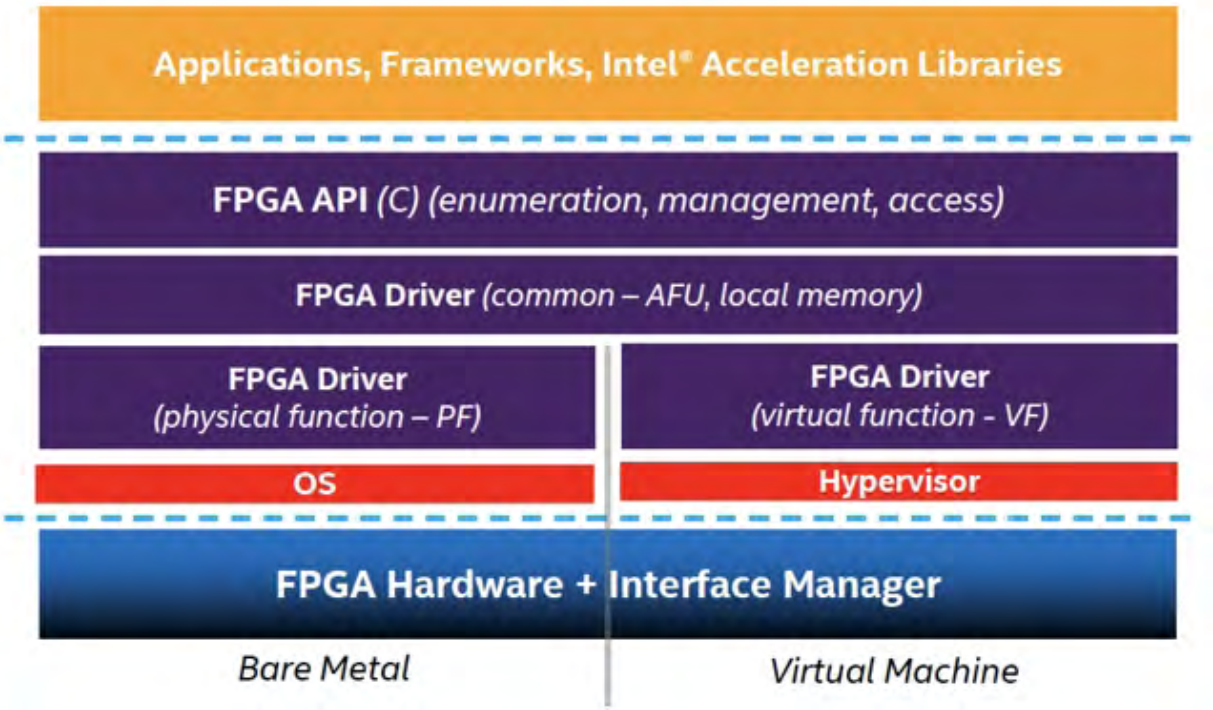


3 **Complete solutions stack empowers both software and hardware developers**

Acceleration Stack for Intel® Xeon® Processors with FPGAs

Intel is investing in FPGA design, programming tools, and libraries to bring about this complete solutions stack for FPGAs. We will discuss how this is happening for cloud computing and particularly within the data center.

The acceleration stack for Intel® Xeon® processors with FPGAs (**Figure 4**) is a robust collection of software, firmware, and tools designed and distributed by Intel to make it easier to develop and deploy Intel FPGAs for workload optimization in the data center. It provides optimized and simplified hardware interfaces and software APIs so that software developers can focus on the unique value-add of their own solutions. Unprecedented code reuse is now possible with our common developer interface for Intel FPGAs. For even faster time to market, system-optimized reference libraries are provided for some domains, making it possible for application developers with little to no prior FPGA experience to use Intel FPGAs to supercharge performance.



4 **Abstracting and managing FPGAs**

An Elegant Solution for using FPGAs in a Software World

Intel aims to make FPGAs accessible through ordinary library calls. Early FPGA users successfully pursued similar approaches, but without portable standards. Consider a standard framework (**Figure 5**) that gives:

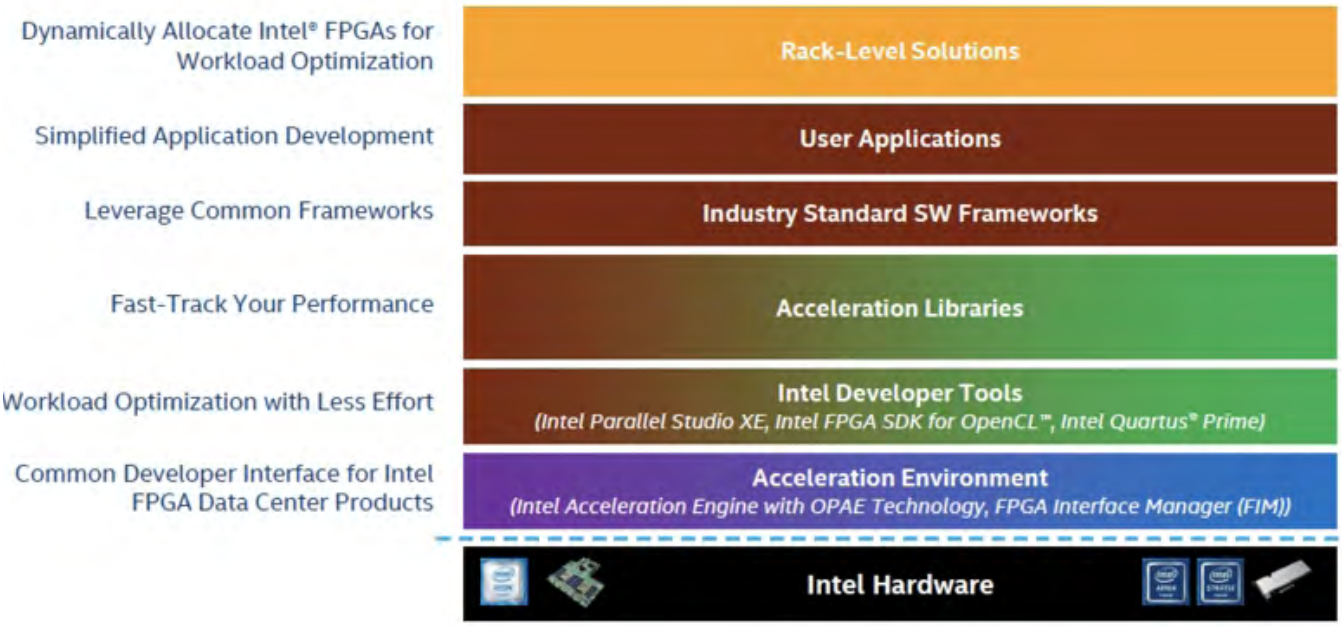
- **Application developers** a standard framework or SDK to indicate required functionality, and then provides FPGA-accelerated versions of that functionality. These developers continue to program in C, C++, Fortran, or some other high-level language with a C interface (e.g., Python*). This allows users to tap into the broader community of open source and commercial tools developers.
- **Systems** a standard way to access and use FPGAs they deploy. Data center operators, for instance, can use the open programmable acceleration engine (OPAE) in their acceleration stack to manage FPGAs in a data center environment.
- **FPGA code developers** a standard way to package and share accelerated routines. These developers will use OpenCL or some other method (e.g., an HDL/RTL such as VHDL or Verilog) to precisely tune and export their FPGA routines.

System owners use the framework to allow applications to load/unload the FPGA packages they need. Application developers use the framework to request and use FPGA packages, while still doing all programming in a high-level language. FPGA programmers can use low-level languages to create FPGA modules and package them for use by application developers.

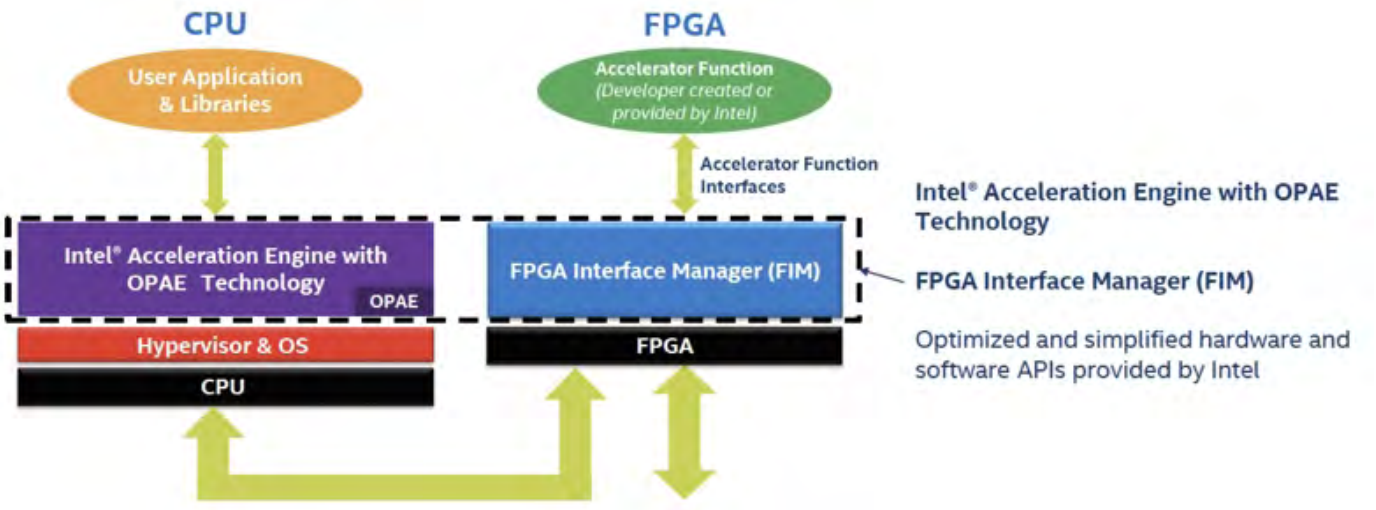
Open Programmable Acceleration Engine (OPAE)

To help realize this acceleration stack in data centers, Intel helped create the Open Programmable Acceleration Engine (OPAE) (**Figure 6**). OPAE runs on the processor and handles all the details of the FPGA reconfiguration process. The OPAE also provides libraries, drivers, and sample programs that can be used to develop routines for the FPGA. It offers consistency across product generations by abstracting hardware-specific FPGA resource details. It's designed for minimal software overhead and latency and offers a lightweight user-space library (libfpga). OPAE supports both virtual machines and bare-metal platforms.

The OPAE C API allows software systems to interface with FPGAs. The API provides device- or platform-specific extensions to model specific features of target architectures. For example, Intel has created a platform-specific API extension to expose a low-latency notification mechanism over the coherent memory interconnect of the Intel Xeon processor with Integrated FPGA, which is included as part of the Intel FPGA IP library.



5 **Intel® Xeon® processor acceleration stack for FPGAs**



6 **Acceleration environment**

A key capability of Intel FPGAs is the ability to dynamically reconfigure a portion of an FPGA while the remaining design continues to function. This allows us to reconfigure regions of the FPGA at runtime to implement different functionality as needed. The OPAE takes advantage of partial reconfigurability via distinct bitstreams. This is FPGA lingo for compiled FPGA programs. The FPGA interface manager (FIM, nicknamed blue bitstream) contains the logic to support FPGA accelerators, including the PCIe* IP core, the CCI-P* fabric, the onboard memory interface, and the management engine. The accelerator functional units (AFUs, nicknamed green bitstream) are the compiled versions of our custom functionality. An AFU is an accelerated computational routine, implemented in FPGA logic, which OPAE offloads to an Intel FPGA to improve performance. OPAE supports multiple slots for AFUs on the same FPGA. This makes it reasonable for us to think of AFUs as libraries of FPGA-accelerated functions that applications can load and use, with the ability to have multiple libraries (AFUs) active at the same time.

Intel also supports an AFU simulation environment (ASE), a code development and simulation tool suite available in the Intel® QuickAssist Accelerator Abstraction Layer Software Development Kit. It allows testing of OPAE-enabled software applications against AFU simulations. It aims to provide a consistent transaction-level hardware interface and software API that allows users to develop and debug production-quality AFU and host applications that can then be run on the real FPGA system without modifications.

FPGA IP Libraries: Acceleration Libraries

We mentioned earlier that there was plenty of opportunity for FPGA experts to continue using RTL/HDL to write highly-optimized code. However, what's new for FPGAs is the ability to package such expertise in a standard way, essentially as a library, to be used by application developers who are not FPGA experts. We refer to such libraries as FPGA IP libraries, and the opportunities for writing them seem endless. FPGA IP libraries could help applications in diverse fields like machine learning, genomics, data analytics, networking, autonomous driving, beam forming, or anything else that can benefit by having an FPGA do it in hardware to give us enormous parallelism, low latency, and flexible and power-efficient capabilities. Not surprisingly, Intel has taken the initiative to write an FPGA IP library to help with the basic—but important—functionality to use FPGAs. This library helps bootstrap the rest of us and serves as an example of an FPGA IP library and how it easily integrates into the complete solution stack.

Intel FPGA IP Library

The Intel® FPGA IP Library is a lightweight user-space library that provides abstraction for Intel FPGA IP resources in a compute environment. Built on top of the driver stack that supports an Intel FPGA IP device, the library abstracts away hardware- and operating system (OS)-specific details and exposes the underlying Intel FPGA IP resources as a set of features accessible from within software programs running on the host.

These features include the acceleration logic preconfigured on the device, as well as functions to manage and reconfigure the device. The library enables user applications to transparently and seamlessly leverage Intel FPGA IP-based acceleration.

By providing a unified C API, the Intel FPGA IP library supports different kinds of integration and deployment models, ranging from single-node systems with one or more Intel FPGA IP devices to large-scale deployments in a data center. A simple use-case, for example, is for an application running on a system with an Intel FPGA IP PCIe device to easily use the Intel FPGA IP to accelerate certain algorithms. At the other end of the spectrum, resource management and orchestration services in a data center can use this API to discover and select Intel FPGA IP resources and then divide them up to be used by workloads with acceleration needs.

Intel FPGA SDK for OpenCL

For FPGA developers who want to create custom accelerator functions to run on Intel FPGAs, the acceleration stack provides the Intel® FPGA SDK for Open Computing Language (OpenCL). OpenCL is an industry standard, C-based programming language that allows users to abstract away the traditional hardware FPGA development flow and use a faster, higher-level software development flow. With the Intel FPGA SDK for OpenCL, you develop FPGA designs in C using a high-level software flow. We can emulate OpenCL C accelerator code on an x86-based host in seconds, get a detailed optimization report with specific algorithm pipeline dependency information, or prototype the accelerator kernel on a virtual FPGA fabric in minutes.

Summary

Intel is leading the FPGA world with highly programmable FPGA solutions. The time is right to examine the programming aspects of FPGAs. Moore's Law has given FPGA designers a lot to work with, and the designers, in turn, have used those transistors to add features with software programmability in mind. OpenCL has given a boost for connecting FPGA developers with application needs. And now Intel has released a framework to help system owners, application developers, and FPGA programmers interact in a standard way—a real advantage when using Intel FPGAs through a common developer interface, tools, and IP that make it easier to leverage FPGAs and reuse code.

Explore This Topic in Depth

- Intel landing page for FPGA programming
- Intel information on the Intel Xeon processor acceleration stack for FPGAs
- Open Programmable Acceleration Engine (OPAE)
- Overview paper: The Open Programmable Acceleration Engine (OPAE)
- Intel® FPGA SDK for Open Computing Language* (OpenCL*)
- Intel FPGAs for AI
- Intel® FPGA product overviews
- Talk: Inside Microsoft's FPGA-Based Configurable Cloud by Mark Russinovich, CTO, Azure
- Intel's Autonomous Driving Press Kit (lots of links)
- Forbes Insights: The Coming Data Avalanche – and How We'll Handle It
- Accelerating the Future: The Economic Impact of the Emerging Passenger Economy
- FPGA for Dummies and other introductory materials
- Accelerate FPGA development with C++ using Intel® HLS Compiler

BLOG HIGHLIGHTS

Intel® Computer Vision SDK—A Brief Overview

BY MARTIN K., INTEL CORPORATION

Earlier this month the Intel® Computer Vision SDK Beta was released. This SDK allows developers to make their computer vision applications more accurate and faster. This week, I had the pleasure of sitting down with Tudor Panu in order to talk to him about what the Intel CV SDK is, a demo showing off some capabilities, and how developers can leverage the SDK for their computer vision projects. You can watch the full discussion here.

In this blog, I would like to share what I learned about the Intel CV SDK and share some resources to help get you started in developing computer vision applications.

Read more >

MODERNIZE YOUR CODE FOR PERFORMANCE, PORTABILITY, AND SCALABILITY

What's New in Intel® Parallel Studio XE

Jackson Marusarz, Technical Consulting Engineer, Intel Corporation

Whether you're working on HPC clusters, remote clouds, local workstations, or anything in between, **Intel® Parallel Studio XE** is a workhorse—with compilers, libraries, and tools to help you improve your productivity and application performance. And Intel continues to innovate with the latest release of Intel Parallel Studio XE. Besides adding support for the newest hardware and programming language standards, several new features address growing technologies and new environments.

Hardware

Tools in Intel Parallel Studio XE have support for the latest hardware including **Intel® Xeon Phi™ processors** and the **Intel® Xeon® Scalable processor family**. Taking advantage of the 60-plus cores (200-plus threads) on Intel Xeon Phi processors and the latest **Intel® AVX-512** vectorization instructions on these platforms moves your application performance into the future—and this latest tool suite helps you get there.

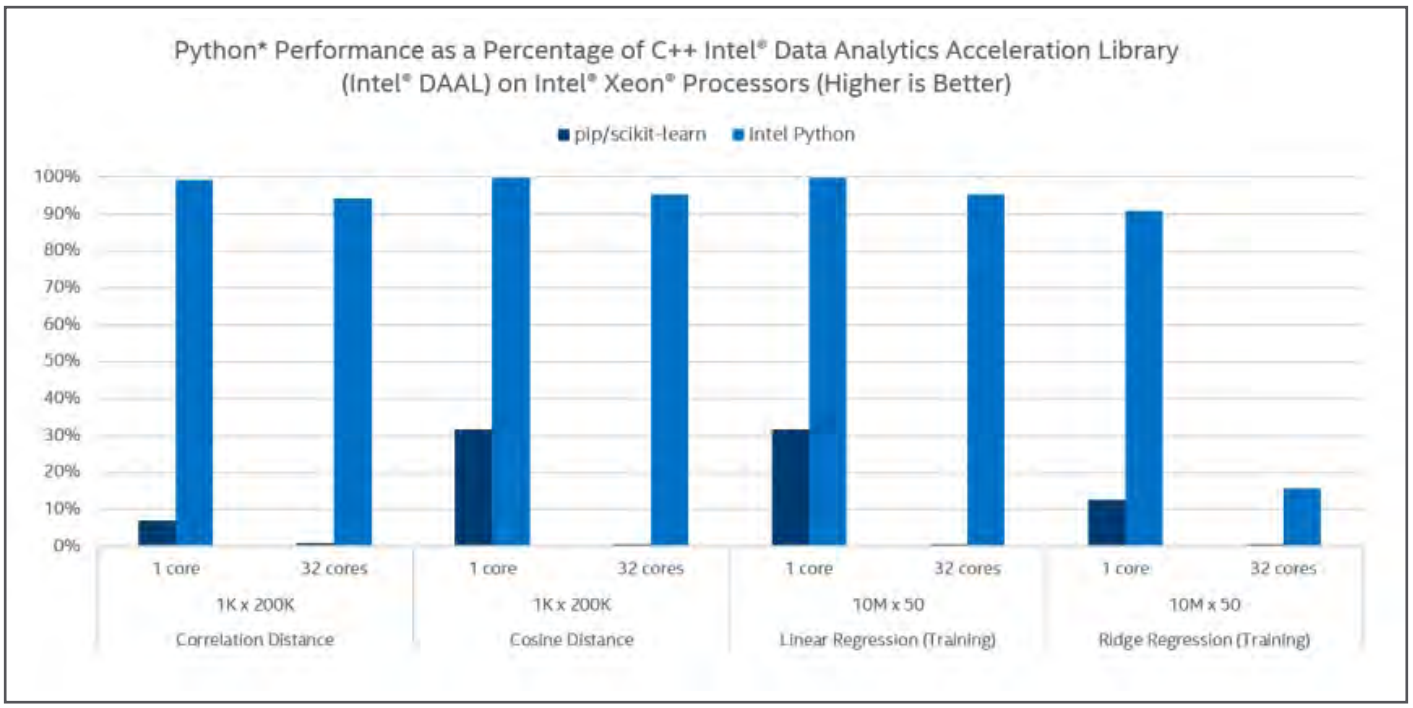
Compilers

The **Intel® C/C++** and **Fortran*** compilers have updates for many of the latest standards without sacrificing performance. The Intel C/C++ compiler has full support for C11 and C++14, as well as initial C++ 17 support. The Fortran compiler has full Fortran 2008 and initial Fortran 2015 support. Also, to help take advantage of growing core counts and vector registers, this release adds Parallel Standard Template Library (Parallel STL) for parallel and vector execution of the C++ STL and initial OpenMP* 5.0 draft support. Using the Intel compiler is often as easy as switching a few variables in a Makefile* or integrating with your environment like Microsoft Visual Studio* or Xcode*. **Try out a free evaluation** to see if you can boost your application's performance.

High-Performance Python*

The latest release of the **Intel® Distribution for Python*** improves the speed of many common libraries and algorithms, especially in data analytics (**Figure 1**). The new optimizations in the SciPy* library and NumPy* package provide speedups for scikit-learn*, one of the most popular machine learning packages. The Intel Distribution for Python also leverages the **Intel® Data Analytics Acceleration Library (Intel® DAAL)** through the pyDAAL interface. For some algorithms in scikit-learn, speedups as high as 140X have been achieved.

This release also includes an **OpenCV*** package accelerated with **Intel® Integrated Performance Primitives**. OpenCV is a popular library used for computer vision across many disciplines.



1 **Python performance as a percentage of C++/Intel DAAL on Intel Xeon processors (higher is better)**

Performance Libraries

Intel Parallel Studio XE comes with several libraries designed to ease the burden of creating high-performance software. This latest release builds on that motivation with improved algorithms and usability across the board.

Intel® Math Kernel Library (Intel® MKL)

Intel® MKL implements some of the most common mathematical routines in highly optimized, hand-tuned versions to take advantage of the latest processor features. Several improvements have been made targeting batched and compact operations, allowing users to run large groups of linear algebra computations more efficiently. This is done using the Batch and Compact APIs. [**Editor's note***:* The Batch API is discussed in "**Introducing Batch GEMM Operations**" on **Intel® Developer Zone**. The Packed API was discussed in "Reducing Packing Overhead in Matrix-Matrix Multiplication" in ***The Parallel Universe* Issue 27**.] Intel MKL takes care of the grouping and parallelization behind the scenes. Also, this release adds 24 new vector math functions, providing a wider range of highly optimized routines to choose from.

Intel® Integrated Performance Primitives (Intel® IPP)

Intel® IPP provides performance-optimized, low-level building blocks for image, signal, and data processing (data compression/decompression and cryptography) applications. The latest version of Intel IPP has again boosted many performance-sensitive algorithms, including the addition of SSE4.2 and AVX2 vector instructions for LZO (Lempel–Ziv–Oberhumer) data compression. Also, the previous dependency the cryptography package had on the main library has been removed, making it easier to take advantage of the powerful cryptography algorithms included in Intel IPP.

Intel® Threading Building Blocks (Intel® TBB) and Parallel STL

Developers looking to parallelize their C++ applications have known about Intel® TBB for years. The open-source library has been improved upon again and again, and this year Intel TBB is being used in the Intel implementation of the Parallel STL.

Analysis Tools

Intel Parallel Studio XE Professional and Cluster editions include several tools to help analyze, tune, and debug applications in increasingly complex hardware and software ecosystems.

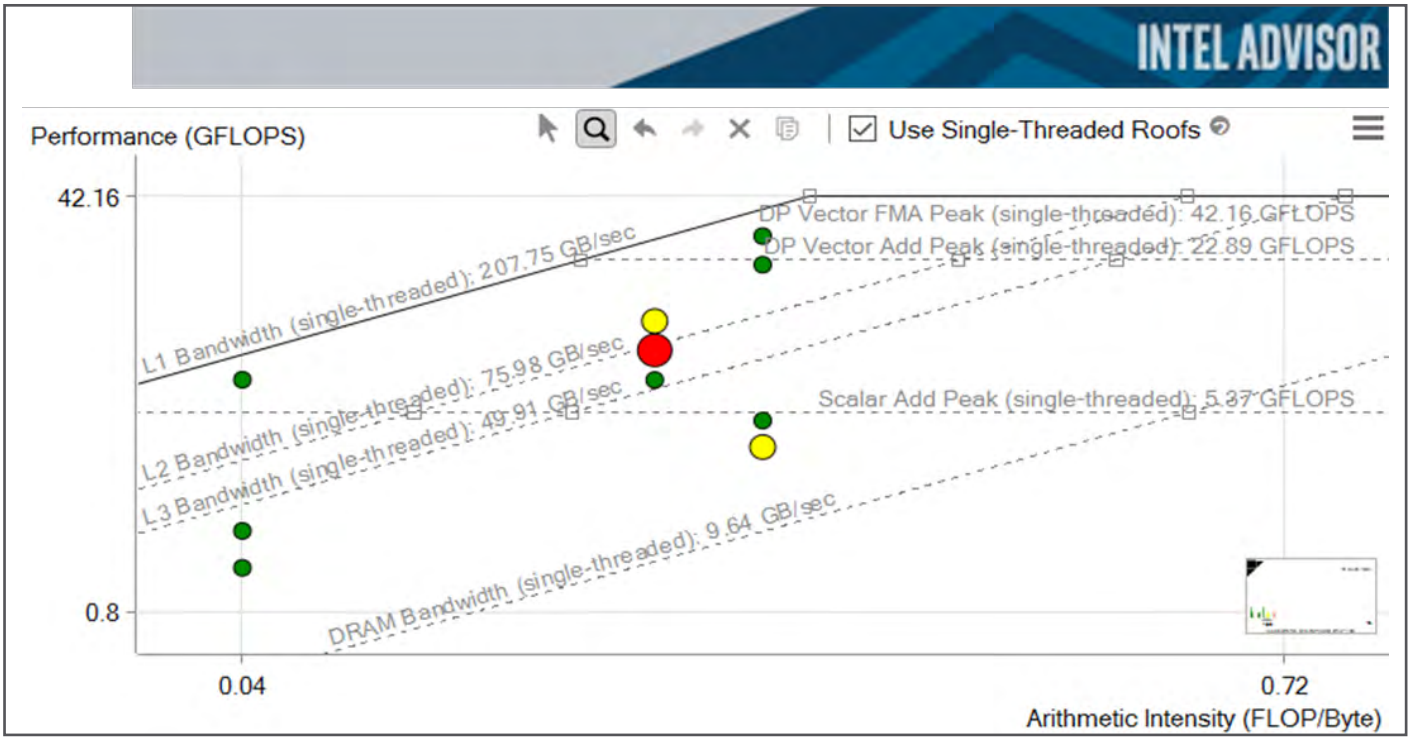
Intel® Advisor

Whether you're looking to add performance through parallelism or vectorization, Intel® Advisor can help. Which functions and loops should I target for optimizations? What's preventing the compiler from vectorizing my code? How much will my performance improve if I make the recommended changes? Intel Advisor is designed to answer questions like these. The 2018 version of Intel Advisor includes the cutting-edge Roofline Analysis feature to pinpoint memory bottlenecks—specifically, loops suffering from poor vectorization or memory locality (**Figure 2**).

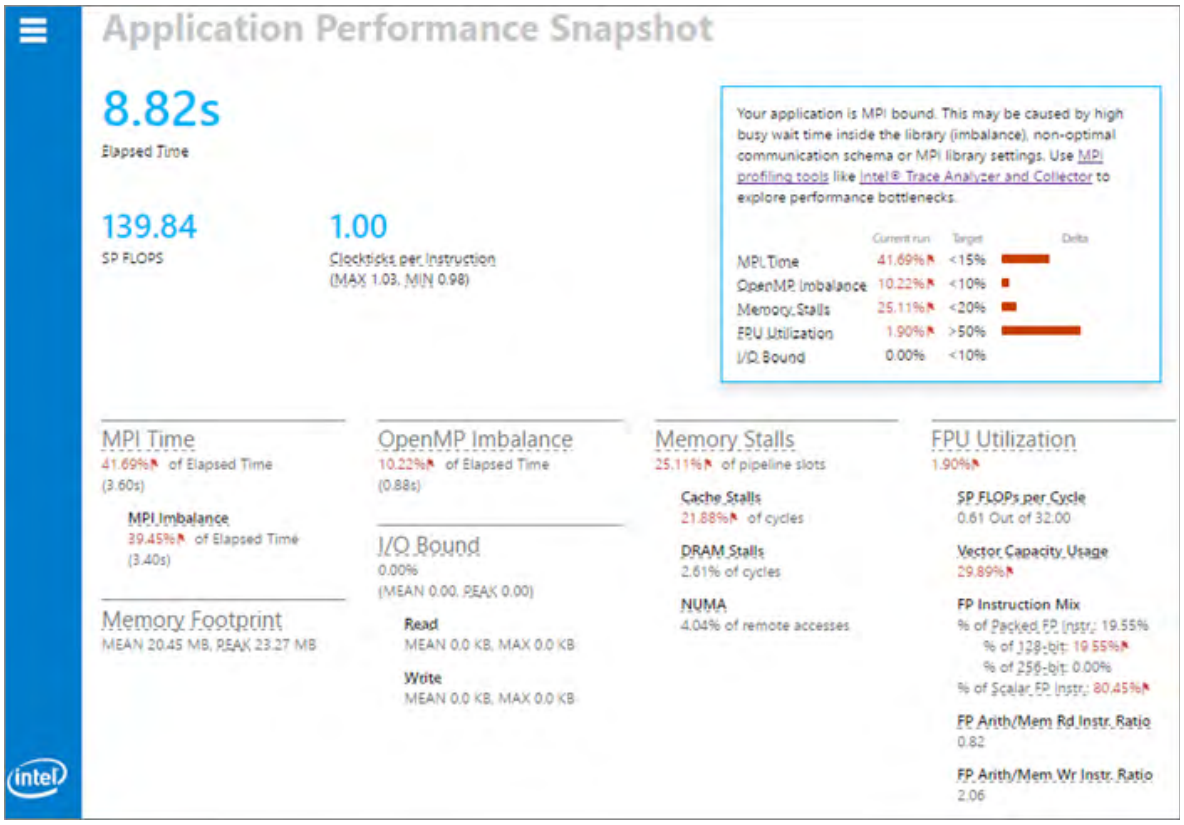
Use Intel Advisor to easily determine where and how your code can benefit from advances in the latest technologies like AVX-512 and the highly-parallel Intel Xeon and Xeon Phi architectures.

Intel® VTune™ Amplifier—Now including Performance Snapshot

Intel® VTune™ Amplifier is a powerful analysis tool with built-in expert guidance to help you understand and boost your application's performance. And now it includes Performance Snapshot (**Figure 3**), a quick and easy-to-use script that provides a high-level view of an application's use of available hardware (CPU, FPU, and memory). Whether you're parallelizing with MPI, OpenMP, or both, time spent inside these libraries or time spent waiting for parallel work to complete can quickly degrade performance.



2 **Intel® Advisor Roofline Analysis**



3 **Intel® VTune™ Amplifier Performance Snapshot**

Additionally, letting floating-point units sit idle or stalling CPUs while they wait for memory accesses can leave lot of performance on the table. Use Performance Snapshot to reclaim these lost cycles.

Intel VTune Amplifier is also adding features for users taking advantage of new technologies like containerization and cloud computing. It's now possible to profile applications running in Docker* and Mesos* containers or attach an Intel VTune Amplifier performance analysis to a running Java* service or daemon.

Intel® Inspector

Optimizing applications for performance is as important as ever, but it's all for naught if an application doesn't run correctly. Intel Inspector automatically checks your application for threading and memory errors as it runs. These hard-to-diagnose issues may not be detected through standard tests that rely on incorrect results. However, the algorithms in Intel Inspector can detect issues that may cause a problem in the future, like memory leaks and non-deterministic race conditions. We've added more advanced locking models in this release and, because Intel Inspector doesn't require any special recompilation, it just takes a few clicks to get a profile started and see what issues may be hiding in your code.

Cluster Tools

Cluster computing used to be confined to a limited audience that needed remote access to large, often strictly managed clusters. Now, with the ubiquity of cloud computing resources and processors like the Intel Xeon Phi processor—with enough cores to act like a single-node cluster—cluster computing is expanding its reach. Intel has always been in this field, and the tools in the Cluster Edition of Intel Parallel Studio XE reflect that experience.

The latest version of **Intel® MPI Library** includes significant optimizations to application startup and finalization, enhancing productivity and scalability.

Intel® Trace Analyzer and Collector now supports OpenSHMEM*, which has growing interest in the partitioned global address space (PGAS) community.

A recent addition to the suite of tools is **Intel® Cluster Checker**, which uses a built-in expert system to diagnose cluster issues and propose actionable remedies. [**Editor's note***:* Intel Cluster Checker is discussed in greater detail in **Is Your Cluster Healthy?** in this issue.]

Intel Cluster Checker performs tests to identify issues with cluster functionality, performance, and uniformity, including on the latest Intel® hardware and software. The tool is useful for system administrators who frequently check cluster status. Now the API gives developers access to these diagnostics and can provide analyses specific to their needs.

Hardware and Software are Evolving

Creating applications to meet the demands of today's ecosystem is hard enough. Add to that the requirement of seamlessly adapting to future platforms and environments and keeping up becomes a huge challenge. Adopting tools like those in Intel Parallel Studio XE makes it easier and more efficient to design scalable, high-performance software. Try it out for free today and prepare your code for the platforms of tomorrow.

BLOG HIGHLIGHTS

Create High-Performance, Scalable, and Portable Parallel Code with New Intel® Parallel Studio XE 2018

BY HENRY GABB, INTEL CORPORATION

Intel® Parallel Studio XE is our flagship product for software development, debugging, and tuning on Intel® processor architectures for HPC, enterprise, and cloud computing. It is a comprehensive tool suite that contains everything from compilers and high-performance math libraries all the way to debuggers and profilers for large-scale cluster applications. These tools enable developers to exploit the full performance potential of Intel® processors. Intel Parallel Studio XE is designed to help devs create high-performance, scalable, reliable parallel code—faster.

The latest release, Intel Parallel Studio XE 2018, contains many new and interesting features[1]. Let's start with parallelism. It's in the product name, after all. Software development and...

Read more >

DEALING WITH OUTLIERS

How to Find Fraudulent Transactions in a Real-World Dataset

Oleg Kremnyov, QA Engineer; Mikhail Averbukh, Software Engineer; and Ivan Kuzmin, Software Engineering Manager; Intel Corporation

One challenging—but also very important—task in data analytics is dealing with outliers. We generally define outliers as samples or events that are inconsistent with the rest of data population[1]. An outlier often contains useful information about abnormal characteristics of the systems and entities that impact the data generation process[2].

Common applications for outlier detection algorithms include:

- Intrusion detection systems
- Credit card fraud
- Interesting sensor events
- Medical diagnosis

In this article, we'll focus on one of the most common applications of outlier detection—credit card fraud. With some simple outlier detection approaches, it's possible to find 75 to 85 percent of fraudulent transactions—with false alarms less than 1 percent of the time—on a real-world dataset.

Defining the Approaches

There are two basic approaches to detecting outliers:

1. **Supervised**, where we use fraudulent and non-fraudulent examples to predict the class of a new observation
2. **Unsupervised**, where we don't have labeled examples and detect fraudulent examples as outliers or abnormalities

Breaking it down further, there are three basic supervised approaches:

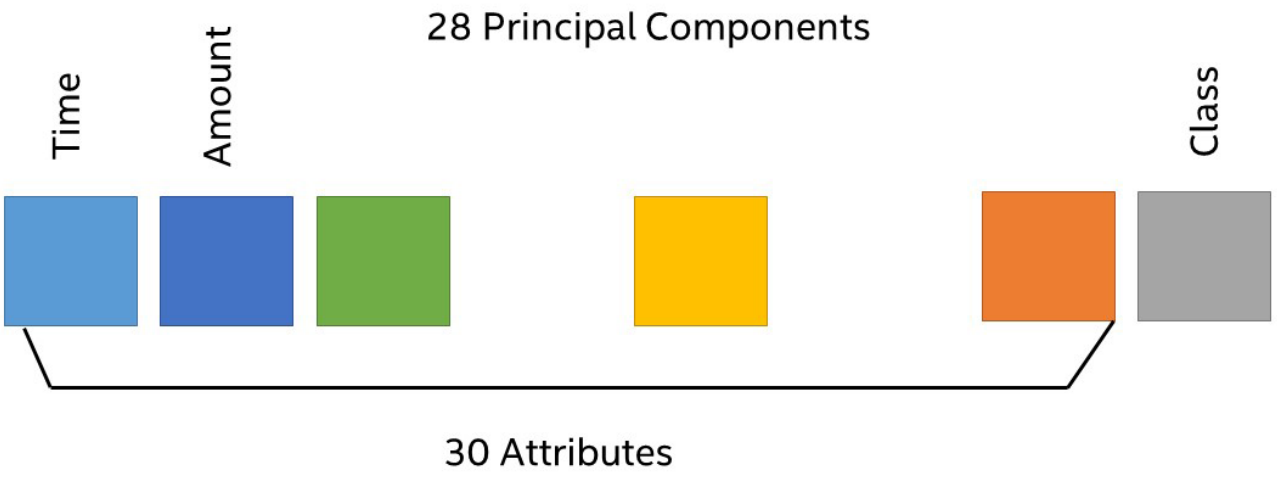
1. Neural networks
2. SVM
3. Logistic regression

Unsupervised approaches include:

1. Statistical-based techniques like BACON* outlier detection
2. Multivariate outlier detection
3. Clustering-based techniques[3, 4] like k-means, expectation-maximization (EM), and DBSCAN*[5], an algorithm that detects noise in data that can be treated as outliers[6].

We'll compare these techniques and evaluate each methodology based on certain design criteria. Accuracy metrics will include both detection and false positive rates[7].

It's important to remember that fraud detection is about more than capturing fraudulent events. It also requires capturing these activities as quickly as possible—so an algorithm's performance is an important metric[8]. In our test, we'll use some popular machine learning libraries, including **Intel® Data Analytics Acceleration Library (Intel® DAAL)**, that implement algorithms for outlier detection to compare results for both accuracy and performance.



1 **Dataset attributes**

Dataset Description

We used a dataset[9] from Kaggle*, a platform for predictive modeling and analytics competitions in which companies and researchers post their data and statisticians and data miners from all over the world compete to produce the best models[10]. We chose the most popular real-world dataset on credit card fraud detection from Kaggle[11]. It contains 31 numerical input variables (**Figure 1**).

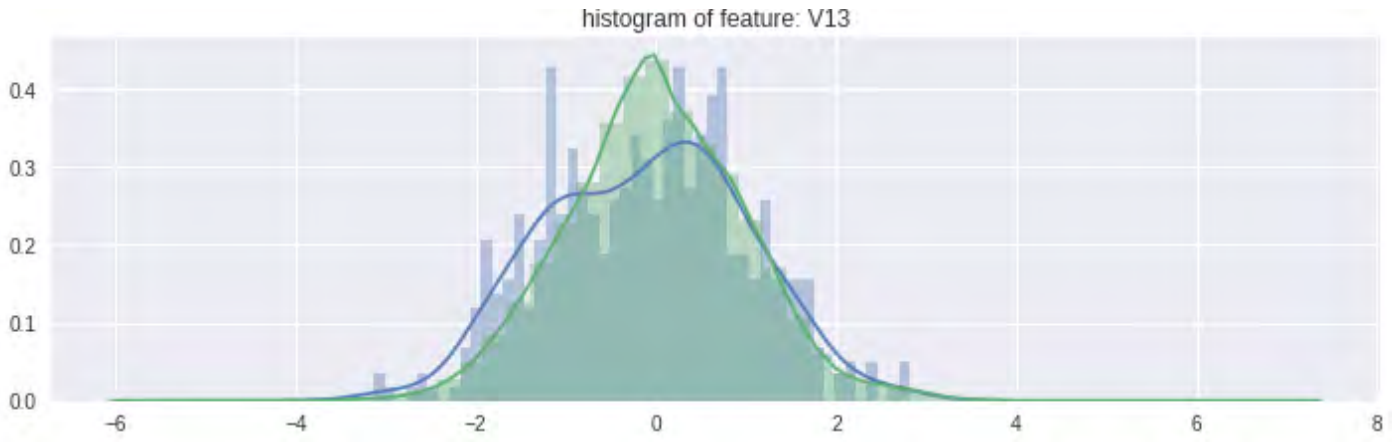
The time attribute represents seconds elapsed between each transaction and the first transaction in the dataset. Amount represents the transaction amount, as the name implies. The remaining 28 features are the principal components obtained with PCA. The original features are not provided due to confidentiality issues.

Class is the response variable. It equals 1 in case of fraud and 0 otherwise. The dataset is highly unbalanced because fraud transactions are far less common than normal ones. Frauds represent only 0.172 percent of all transactions (492 frauds out of 284,807 transactions).

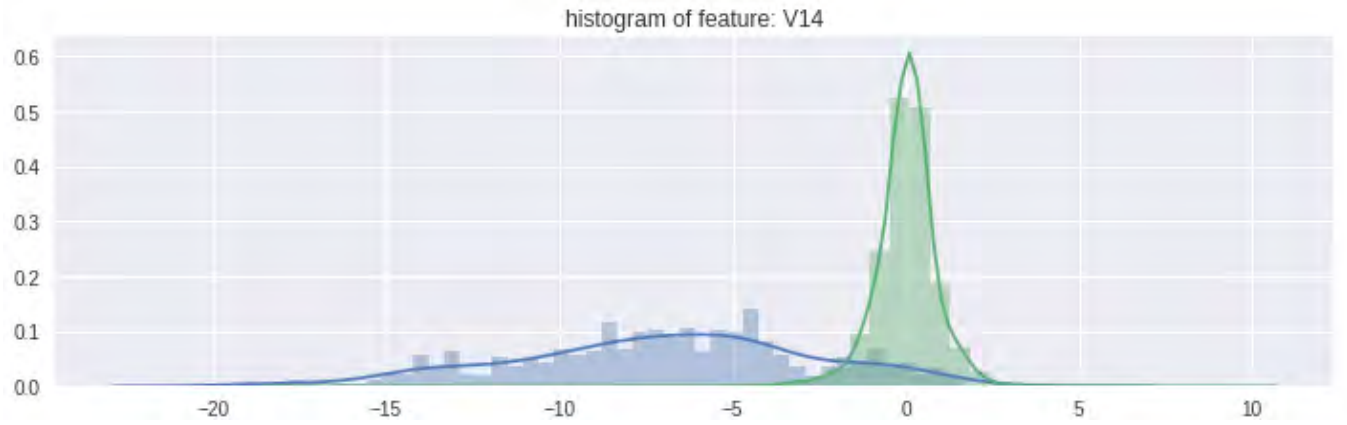
Let's take a look at conditional distributions of some of principal components (**Figures 2** and **3**). Principal components are not correlated, so dropping those with similar conditional distribution for fraudulent and normal examples could help us build more accurate models. We'll use a point-biserial correlation coefficient[12] to measure the relationship between each principal component and target variable (**Table 1**). We'll use this information later for feature engineering.

Credit Card Fraud Detection Approaches

Now we'll apply different outlier detection techniques to the data, evaluate their accuracy and performance, and compare them. For credit card fraud detection, we'll use both supervised and unsupervised methods.



2 **Attribute with similar distribution between two types of transactions**



3 **Attribute with different distribution between two types of transactions**

Table 1. Point-biserial correlation coefficients for principal components

Variable	Coefficient	Variable	Coefficient	Variable	Coefficient	Variable	Coefficient
V22	0.000805	V28	0.009536	V2	0.091289	V7	0.187257
V23	0.002685	V27	0.017580	V5	0.094974	V3	0.192961
V25	0.003308	V8	0.019875	V9	0.097733	V16	0.196539
V15	0.004223	V20	0.020090	V1	0.101347	V10	0.216883
V26	0.004455	V19	0.034783	V18	0.111485	V12	0.260593
V13	0.004570	V21	0.040413	V4	0.133447	V14	0.302544
V24	0.007221	V6	0.043643	V11	0.154876	V17	0.326481

Unsupervised Approaches

Statistical approaches were the earliest algorithms used for outlier detection[13]. We'll start from outlier-detection-specific algorithms—multivariate outlier detection and BACON* outlier detection. A great advantage of these methods is that they are so computationally efficient that it's easy to apply them to large datasets[14]. We achieved the best results with six principal components with a high point-biserial correlation coefficient (V11, V12, V14, V16, V17, V18) as features. We used implementations from Intel DAAL and the R* package robustX*. Multivariate outlier detection gave better results for accuracy than BACON outlier detection (**Table 2**).

Table 2. Statistical-based outlier detection approaches

Algorithm	Detection Rate, %	False Positive Rate, %	Perfor-mance, msec
Multivariate outlier detection, Intel DAAL (threshold = 8.0)	83.54	0.20	16.16
Multivariate outlier detection, Intel DAAL (threshold = 7.0)	84.55	0.41	17.29
BACON outlier detection, Intel DAAL (alpha = 1e-9, initMethod = baconMahalanobis)	84.55	0.70	82.25
BACON outlier detection, R (alpha = 1e-9, initMethod = baconMahalanobis)	84.73	1.80	1,860
BACON outlier detection, Intel DAAL (alpha = 1e-9, initMethod = baconMedian)	84.55	0.70	78.14
ON outlier detection, R (alpha = 1e-9, initMethod = baconMedian)	84.55	1.80	808

Clustering and outlier detection have a complementary relationship. In clustering, the goal is to partition the points into dense subsets. In outlier detection, the goal is to identify points that don't seem to fit naturally into these dense subsets[2]. Some clustering algorithms, like DBSCAN*, provide sub-products, which can be considered outliers.

There are many clustering-based approaches to outlier detection[2,3,4,15]. We'll apply some of them in our tests. Clustering is inherent in credit card fraud because perpetrators usually produce a group of fraudulent transactions[16].

There are two types of outliers, local and global. Different clustering-based algorithms can detect both types. The attribute values of a global outlier are outlying with respect to the values taken by the majority of the data points. The attribute values of a local outlier are extreme when compared to those of its neighbors[17].
To detect global outliers, we can perform clustering on a dataset and define some clusters as outliers[18] (**Table 3**). We'll perform clustering with k-means and EM, treating small clusters as outliers[19]. Values of k and n_components were obtained by grid search from range [2, 3, ..., 50]. We used the same initial centroids for k-means for Intel DAAL and Scikit-learn*, obtaining them with k-means++ initialization using the implementation in Intel DAAL. We achieved the best results using 17 principal components with high point-biserial correlation coefficients as features. With these methods, we achieved a relatively low false positive rate.

Table 3. Clustering-based methods for global outlier detection

Algorithm	Detection Rate, %	False Positive Rate, %	Performance, msec
k-means, Intel DAAL (k=20, 10 iterations), detection of clusters of outliers	76.42	0.02	30.15
k-means, Scikit-learn (k=20, 10 iterations), detection of clusters of outliers	76.42	0.02	1,975.49
EM-GMM, Scikit-learn (n_components=7), detection of clusters of outliers	81.10	0.09	73,490.88

We'll also apply clustering techniques that can detect local outliers[20]. A naïve approach is to cluster the transactions using the k-means algorithm[21] and treat points far from cluster centers as outliers (**Table 4**).

We obtained a good rate of detections. However, the number of false alarms is high. The reason is that the k-means algorithm is extremely sensitive to outliers, which have a large impact on cluster configuration. As a result, data points that should be declared as outliers are masked by the clustering. Thus, we need to use a robust version of k-means that can gracefully handle the presence of outliers—k,l-means[15] (**Table 5**). Like k-means, this algorithm requires initial centroids. We'll compute them with different initialization algorithms available in Intel DAAL—random, k-means++[22], and k-means||[23]—and compare the results.

Table 4. K-means-based method for local outliers detection

Algorithm	Detection Rate, %	False Positive Rate, %	Perfor-mance, msec
k-means, Intel DAAL (k=7, 10 iterations), detection of local outliers	80.08	0.94	26.94
k-means, Scikit-learn (k=7, 10 iterations), detection of local outliers	80.08	0.94	1,210.25

Table 5. K,l-means algorithm results

Algorithm	Detection Rate, %	False Positive Rate, %	Perfor-mance, msec
k,l-means(k=10, l=800, 5 iterations, random init method)	81.71	0.14	165.15
k,l-means(k=10, l=1000, 5 iterations), random init method)	83.13	0.21	167.05
k,l-means(k=10, l=2000, 5 iterations), random init method)	85.16	0.56	169.03
k,l-means(k=10, l=2000, 5 iterations), k-means++ init method)	84.95	0.55	169.46
k,l-means(k=10, l=2000, 5 iterations), k-means|| init method)	81.30	0.55	179.84

We can see that this approach provides better accuracy than naïve k-means. Results are different for different initializations. The random method proved to be the best for this problem.

We'll also use a density-based clustering algorithm that can detect noise in data (**Table 6**). A point is detected as noise if it differs significantly from its neighbors. A natural assumption is to treat noise data as local outliers. We used the same six features as for statistical methods. Values of parameters were obtained by grid search. We used the R package dbscan*. We reached a good level of detected frauds with a moderate level of false alarms. However, the performance is significantly worse compared to previous approaches.

Table 6. Dbscan for outlier detection

Algorithm	Detection Rate, %	False Positive Rate, %	Perfor-mance, msec
DBSCAN, R (eps = 1, minPts = 8)	82.52	0.43	181,214
DBSCAN, R (eps = 0.9, minPts = 8)	84.35	0.64	121,348

Supervised Approaches

Supervised learning methods require training data that contain examples of each class—in our case, normal and fraudulent transactions[2]. We'll divide our dataset into training and evaluation sets, with 80 percent of the data used for training and 20 percent used for model evaluation.

We start with the neural network (**Table 7**) proposed in the Kaggle kernel[24]. A simple multi-layer perceptron with three hidden layers was used by the kernel author. Multi-layered perceptrons (MLPs) are widely used neural networks for classification as well as outlier detection[25].

Table 7. Neural network

Algorithm	Detection Rate, %	False Positive Rate, %	Performance, msec (Training)	Performance, msec (Prediction)
Neural network, TensorFlow	82.93	0.10	48,005.41	7.62

Another supervised learning algorithm used in outlier detection is logistic regression[16] (**Tables 8** and **9**). We'll use 17 attributes with the highest point-biserial correlation coefficients (**Table 1**) and the Amount feature, which was pre-processed with Z-score normalization. From the histograms[24], we can see that fraud and normal observation are linearly separable due to the use of logistic regression. Our dataset is unbalanced, so use of random undersampling could help to build a model that detects frauds more accurately. This omits many non-fraudulent examples from training, so we'll use a voting classifier[26], with each estimator trained on a randomly under-sampled training dataset. (The regularization parameter, C, was set to one after experimentation with a wide range of values.) With this technique, we improved the detection rate by 3 percent with almost the same level of false alarms.

Table 8. Logistic regression

Algorithm	Detection Rate, %	False Positive Rate, %	Performance, msec (Training)	Performance, msec (Prediction)
Logistic regression, Scikit-learn (1% of samples in train data are fraudulent)	80.61	0.02	323.69	1.45
Logistic regression, Scikit-learn (9% of samples in training data are fraudulent)	83.67	0.02	20.88	1.51

Table 9. Ensemble of logistic regression models

Algorithm	Detection Rate, %	False Positive Rate, %	Performance, msec (Training)	Performance, msec (Prediction)
Logistic regression, Scikit-learn (9% of samples in train data are fraudulent, ensemble of 100 classifiers)	85.71	0.19	3,530.13	131.72
Logistic regression, Scikit-learn (9% of samples in train data are fraudulent, ensemble of 10 classifiers)	86.73	0.18	335.81	12.22

Let's see how support vector machines (SVM), another supervised learning algorithm which can also be used for outlier detection[16], works on the same data. We run SVM with 100,000 iterations, a linear kernel, and C=1. (These parameters were obtained with a grid search.) The same features were used as for logistic regression. Use of SVM ensembles improves the detection rate significantly while giving a low false alarm rate (**Table 10**).

Evaluation of the Different Anomaly Detection Approaches

There are a variety of unsupervised approaches for fraud detection. We applied a few of them in this article. Outlier detection-specific algorithms, k,l-means and DBSCAN, achieved the best detection rates, reaching 85 percent of frauds detected, while use of k-means to detect clusters of fraudulent transactions gives the lowest number of false alarms. The sooner fraudulent transactions are detected, the more losses can be avoided by stopping transactions made with fraudulent credit cards[16]—making the performance of the algorithm another key metric. The final choice of fraud detection approach will depend on many factors, including the cost of the fraudulent behavior detected and the cost associated with stopping it[7]. Unsupervised methods can be useful in situations where there is no prior knowledge of classes of observations.

Table 10. SVM

Algorithm	Detection Rate, %	False Positive Rate, %	Performance, msec (Training)	Performance, msec (Prediction)
SVM, Scikit-learn (9% of samples in train data are fraudulent)	83.67	0.04	250.00	174.40
SVM, Intel DAAL (9% of samples in train data are fraudulent)	76.53	0.01	246.68	12.23
SVM, Scikit-learn (9% of samples in train data are fraudulent, ensemble of 10 classifiers)	91.83	0.04	2,781.41	1,792.12
SVM, Intel DAAL (9% of samples in train data are fraudulent, ensemble of 10 classifiers)	91.83	0.04	2,662.94	119.24
SVM, Scikit-learn (9% of samples in train data are fraudulent, ensemble of 100 classifiers)	91.83	0.03	28,508.28	21,806.03
SVM, Intel DAAL (9% of samples in train data are fraudulent, ensemble of 100 classifiers)	91.83	0.03	27,718.29	1,191.73

When we have prior knowledge of classes of observations, we can use supervised learning algorithms. We used techniques for unbalanced classification, like undersampling, to help increase the number of frauds detected. Ensembles of SVM classifiers give the highest detection rates, while random forests have the lowest false alarm rates. Although supervised methods like SVM gave us significantly better accuracy compared to unsupervised ones, use of them can pose problems:

- **Accurately labeled training data** might be prohibitively expensive to obtain[25] and a data scientist has to be confident about the true classes of the training data used to build the models.
- **The time to train a model** can be high depending on the number of samples and the number of features per sample.
- **Unlike unsupervised approaches,** supervised learning cannot detect new types of fraud without retraining[16,27].

As we've shown, choosing the right machine learning algorithm for a given application is a non-trivial problem that requires a lot of thought.

References

1. Victoria Hodge, “Outlier and Anomaly Detection: A Survey of Outlier and Anomaly Detection Methods.”
2. Charu C. Aggarwal, “Outlier Analysis,” Second Edition.
3. Vaishali, “Fraud Detection in Credit Card by Clustering Approach.”
4. Yogesh Bharat Sonawane, Akshay Suresh Gadgil, Aniket Eknath More, Niranjan Kamalakar Jathar, “Credit Card Fraud Detection Using Clustering Based Approach.”
5. Ester, M., H. P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise” in Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996.
6. Samson Kiware, B.A, “Detection of Outliers in Time Series Data.”
7. Elham Hormozi , Hadi Hormozi, Mohammad Kazem Akbari, Morteza Sargolzaei Javan, “Accuracy Evaluation of a Credit Card Fraud Detection System on Hadoop MapReduce.”
8. Shraddha Ramesh Bhagwat, Vaishali Londhe, “A Review of Various Credit Card Detection Techniques.”
9. Credit card fraud detection dataset.
10. Overview of Kaggle on Wikipedia.
11. Credit card fraud datasets on Kaggle.
12. Point-biserial correlation coefficient on Wikipedia.
13. Victoria J. Hodge and Jim Austin, “A Survey of Outlier Detection Methodologies.”
14. Nedret Billor, Ali S. Hadi, Paul F. Velleman, “BACON: Blocked Adaptive Computationally Efficient Outlier Nominators.”
15. Sanjay Chawla, Aristides Gionis, “k-means: A unified Approach to Clustering and Outlier Detection.”
16. Sanjeev Jha, Montserrat Guillen, and J. Christopher Westland, “Employing Transaction Aggregation Strategy to Detect Credit Card Fraud.”
17. Marie Ernst and Gentiane Haesbroeck, “Detection of Local and Global Outliers in Spatial Data.”
18. Bin-mei Liang, “A Hierarchical Clustering Based Global Outlier Detection Method.”
19. Yuan Wang, Xiaochun Wang, Xia Li Wang, “A Spectral Clustering Based Outlier Detection Technique.”
20. Zengyou He, Xiaofei Xu, Shengchun Deng, “Discovering Cluster-Based Local Outliers.”
21. Stuart P Lloyd, “Least Squares Quantization in PCM,” IEEE Transactions on Information Theory 1982, 28 (2): 1982pp: 129–137.
22. Arthur, D. and Vassilvitskii, S, “k-means++: The Advantages of Careful Seeding,” Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007, pp. 1027-1035.
23. B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, “Scalable K-means++,” Proceedings of the VLDB Endowment, 2012.

References (continued)

24. Kaggle kernel, use of neural network for fraud detection.
25. Varun Chandola, Arindam Banner Jee, and Vipin Kumar, “Outlier Detection : A Survey.”
26. Gareth James, “Majority Vote Classifiers: Theory and Applications.”
27. Bolton, R.J. and Hand, D.J., “Unsupervised Profiling Methods for Fraud Detection” in Conference on Credit Scoring and cCedit Control, Edinburgh, 2001.

Configuration Information

Hardware

- Processor: Intel® Xeon® Platinum 8168 processor @ 2.70GHz
- Core(s) per socket:24
- Socket(s): 2
- Memory: 63 GB

Software

- Intel® Data Analytics Acceleration Library 2018 Gold
- R version 3.4.1
- Scikit-learn version 0.19.1
- robustX package 1.2-2
- dbscan package 1.1-1

TUNING FOR SUCCESS WITH THE LATEST SIMD EXTENSIONS AND INTEL® ADVANCED VECTOR EXTENSIONS 512

Best Practices for Taking Advantage of the Latest Architectural Features

Xinmin Tian, Senior Principal Engineer; Hideki Saito, Principal Engineer; Sergey Kozhukhov, Senior Staff Engineer; and Nikolay Panchenko, Staff Engineer; Intel Compiler and Language Lab, Intel Corporation

Intel® Advanced Vector Extensions 512 (Intel® AVX-512), the latest x86 vector instruction set, has up to two fused-multiply add units plus other optimizations. It can accelerate the performance of workloads such as:

- Scientific simulations
- Financial analytics
- Artificial intelligence
- 3D modeling and simulation
- Image and audio/video processing
- Cryptography
- Data compression/decompression

In this article, we'll give a brief overview of the Intel AVX-512 Instruction Set Architecture (ISA) and describe what's new in the Intel 18.0 compilers for **Intel® Xeon® Scalable processors**. Next, we present several new `simd` language extensions for Intel AVX-512 support in Intel's latest compilers. Finally, we share our best practices in performance tuning to achieve optimal performance with the Intel AVX-512 ISA.

What's New

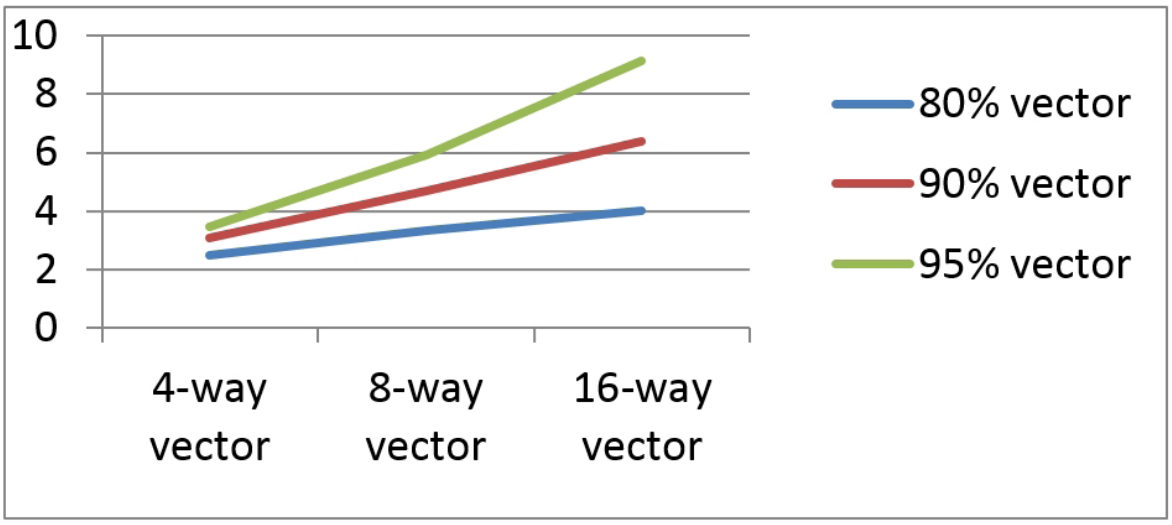
Intel Xeon Scalable processors introduce several new variations of Intel AVX-512 instruction support. The main Intel AVX-512 ISA performance features on Intel Xeon Scalable processor-based servers compared to previous-generation Intel AVX2 are:

- Intel AVX-512 foundation:
  - 512-bit vector width
  - 32 512-bit long vector registers
  - Data expand and compress instructions
  - Ternary logic instruction
  - Eight new 64-bit long mask registers
  - Two source cross-lane permute instructions
  - Scatter instructions
  - Embedded broadcast/rounding
  - Transcendental support
- Intel AVX-512 double- and quad-word Instructions (DQ): QWORD support
- Intel AVX-512 byte and word instructions (BW): Byte and Word support
- Intel AVX-512 Vector Length Extensions (VL): Vector length orthogonality
- Intel AVX-512 Conflict Detection Instructions (CDI): Vconflict instruction

These different AVX-512 features are meant to be supported directly by the hardware and enabled therein. In this article, we'll focus on `simd` language extensions, compiler support, and tuning for the AVX-512-F, AVX-512-BW, AVX-512-CD, AVX-512-DQ, and AVX-512-VL features in Intel Xeon Scalable processors.

Tuning Challenges for Compilers and Programmers

Figure 1 shows the projected speedup for a theoretical application where a perfect vector speedup can be achieved for large portions of the code. In the era of four-way vectors such as in 32-bit integers/floats computed in XMM registers, having the application 80 percent vectorized results in a 2.5x speedup. Incremental benefits from the compiler and the programmers working harder to vectorize 90 or 95 percent of the code are limited.



1 **Ideal vector speedup for a theoretical application**

However, increasing to 8-way and especially 16-way vectors can significantly boost the performance of heavily-vectorized applications. **Figure 1** clearly indicates the potential performance improvements when compilers and programmers squeeze more vector parallelism from applications to enjoy the benefits of the YMM and ZMM vectors in the Intel Xeon Scalable processors. Besides widening the vector registers, the AVX-512 ISA extension comes with a variety of architectural enhancements to help vectorize many more types of code patterns.

AVX-512F, AVX-512VL, and AVX-512BW

AVX-512F, AVX-512VL, and AVX-512BW include natural extensions of AVX and AVX2 for 32 registers and masking support. Together, they cover the largest portion of the Intel AVX-512 family of extensions. AVX-512F extends vector register size to 512-bit for 32-bit and 64-bit integer and floating-point element data. It also:

- **Increases** the number of architectural vector registers from 16 to 32
- **Introduces** dedicated mask registers
- **Adds** masking support for non-load/store operations

AVX-512VL extends AVX-512F so that support for 32 registers and masking can be applied to 128- and 256-bit vectors. Intel AVX-512BW extends Intel AVX-512F so that 32 registers and masking can be applied to 8- and 16-bit integer element data.

Table 1 uses vector add instructions to illustrate the relationship between the element data type (b/w/d/q/ps/pd), the instruction features (xmm/ymm/zmm, masked or no-mask, and register number 0-15/16-31),

Table 1. Relationship between element data type and instruction features (xmm/ymm/zmm, masked/no-mask, etc.)

	(v)paddb	(v)paddw	(v)paddw	(v)paddq	(v)addps	(v)addpd
xmm (0-15) no-mask	SSE2	SSE2	SSE2	SSE2	SSE	SSE2
xmm (16-31) no-mask	AVX-512VL + AVX-512BW	AVX-512VL + AVX-512BW	SSE2	AVX-512VL	AVX-512VL	AVX-512VL
xmm masked	AVX-512VL + AVX-512BW	AVX-512VL + AVX-512BW	AVX-512VL	AVX-512VL	AVX-512VL	AVX-512VL
ymm (0-15) no-mask	AVX2	AVX2	AVX2	AVX2	AVX	AVX
ymm (16-31) no-mask	AVX-512VL + AVX-512BW	AVX-512VL + AVX-512BW	AVX-512VL	AVX-512VL	AVX-512VL	AVX-512VL
ymm masked	AVX-512VL + AVX-512BW	AVX-512VL + AVX-512BW	AVX-512VL	AVX-512VL	AVX-512VL	AVX-512VL
Zmm (0-31) masked/ no-mask	AVX-512BW	AVX-512BW	AVX-512F	AVX-512F	AVX-512F	AVX-512F

and the minimum required ISA extensions on an Intel64 platform. For example, to add a vector of bytes (vpaddb) on the XMM16 register, support for both AVX-512VL and AVX-512BW are required whether the operation uses a mask or not. By providing support for all of AVX-512F, AVX-512VL, and AVX-512BW, Intel Xeon Scalable processors allow the programmers to utilize all 32 available registers and masking capability to all register sizes (XMM, YMM, or ZMM) and all data element types (byte, word, dword, qword, float, and double) on most of the operations.

Beyond natural extensions of AVX and AVX2, AVX-512F includes:

- **Data compress (vcompress) operations** that read elements from an input buffer on indices specified by mask register 1's bits. The elements, which have been read, are then written to the destination buffer. If the number of elements is less than the destination register size, the rest of the space is filled with zeroes.
- **Data expand (vexpand) operations** that read elements from the source array (register) and put them in the destination register on the places indicated by enabled bits in the mask register. If the number of enabled bits is less than destination register size, the extra values are ignored.

AVX-512CDI

Intel AVX-512CDI is a set of instructions that, together with Intel AVX-512F, enable efficient vectorization of loops with *possible* vector dependences (i.e., conflicts) through memory. The most important

instruction is VPCONFLICT, which performs horizontal comparisons of elements within a single vector register. In particular, VPCONFLICT compares each element of a vector register with all previous elements in that register, then outputs the results of all of the comparisons. Other instructions in Intel AVX-512CDI allow for efficient manipulation of the comparison results. We can use VPCONFLICT in different ways to help us vectorize loops. The simplest is to check if there are any duplicate indices in a given `simd` register. If not, we can safely use `simd` instructions to compute all elements simultaneously. If so, we can execute a scalar loop for that group of elements. Branching to a scalar version of the loop on any duplicate indices can work well if duplicates are extremely rare. However, if the chance of getting even one duplicate in a given iteration of the vectorized loop is large enough, then we would prefer to use `simd` as much as possible, to exploit as much parallelism as we can.

What's New in Compiler 18.0 for AVX-512 Tuning

To achieve optimal performance on Intel Xeon Scalable processors, applications should be compiled with the processor-specific option `-xCORE-AVX512` (`/QxCORE-AVX512` on Windows*). Such an executable will not run on non-Intel® processors or on Intel processors that support only older instruction sets. Intel® Compilers 18.0 provide a new option `-qopt-zmm-usage=[high|low]` for users to tune ZMM code generation:

- **Option value low** provides a smooth transition experience from the AVX2 ISA to AVX-512 ISA on Skylake (SKX) targets, such as for enterprise applications. Programmers are encouraged to tune for ZMM instruction usage via explicit vector syntax such as `#pragma omp simd simdlen()`.
- **Option value high** is recommended for applications that are bounded by vector computation (e.g., HPC applictions) in order to achieve more computations per instruction through wider vectors.

The default value is low for SKX-family compilation targets (e.g. `-xCORE-AVX512`). Intel compilers also provide a way to generate a fat binary that supports multiple instruction sets by using the *`-axtarget`* option. For example, if the application is compiled with `-axCORE-AVX512,CORE-AVX2` the compiler can generate specialized code for AVX-512 and AVX2 targets, while also generating a default code path that will run on any Intel, or compatible non-Intel, processor that supports at least SSE2. At runtime, the application automatically detects whether it is running on an Intel processor. If so, it selects the most appropriate code path for Intel processors; if not, the default code path is selected.

It's important to note that irrespective of the options used, compiler may insert calls to specialized library routines (such as optimized versions of memset/memcpy) that will dispatch to the appropriate code path at runtime based on processor detection.

New `simd` Extensions for AVX-512

Intel Compilers pioneered explicit vectorization of C/C++ and Fortran* application programs and led the standardization effort in OpenMP* 4.0 and 4.5 [3, 4, 5, 7, 8]. What follows are the results of our continued innovation on extending the power of explicit vectorization. We proposed them for future OpenMP specifications and have implemented them in Intel Compilers 18.0 as a lead vehicle for OpenMP standardization[3, 8].

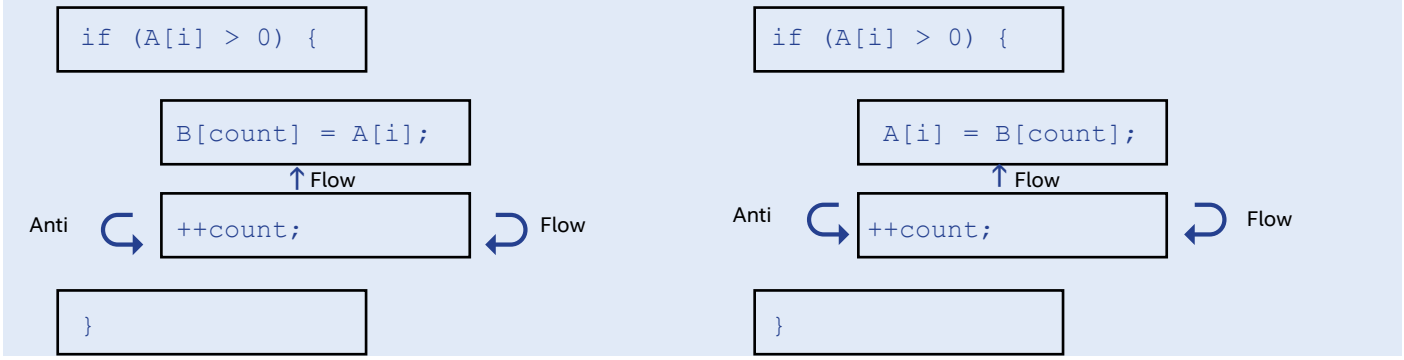
Compress and Expand

Figure 2 shows compress and expand idioms. The compress and expand terms come from the code semantics. For instance, the compress pattern compresses all positive elements from the array `A[]` and stores them consecutively in the array `B[]`. It's known that both patterns contain loop-carried dependencies (**Figures 3** and **4**) and can't be parallelized.

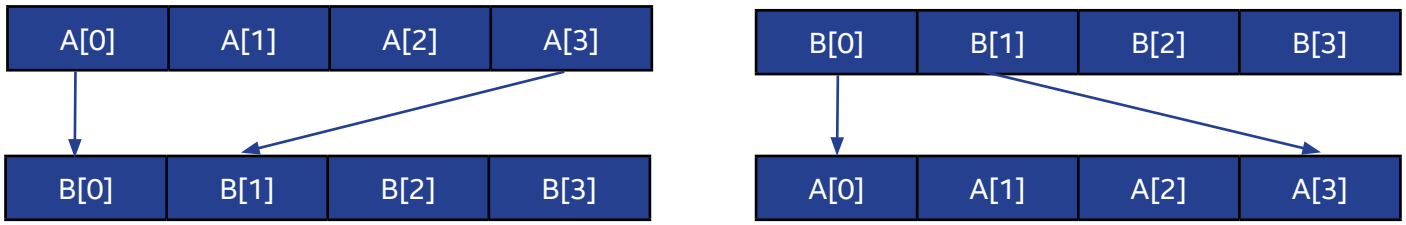
```
// A is compressed into B
int count = 0;
for (int i = 0; i < n; ++i) {
    if (A[i] > 0) {
        B[count] = A[i];
        ++count;
    }
}
```

```
// A is expanded into B
int count = 0;
for (int i = 0; i < n; ++i) {
    if (A[i] > 0) {
        B[i] = A[count];
        ++count;
    }
}
```

2 **Compress and expand idioms**



3 **Data dependence graph for compress and expand patterns**



4 **Compress and expand execution**

However, vectorizing compress and expand idioms is possible with special processing of the dependencies. AVX-512 ISA adds `v[p]compress` and `v[p]expand` instructions that helps effectively vectorize these codes. **Figure 5** shows the generated `simd` code for the compress idiom using the new AVX-512 instruction `vcompressps`.

```
..B1.14:
vmovups (%rdi,%r11,4), %zmm2
vcmpps $6, %zmm0, %zmm2, %k1
kmovw %k1, %edx
testl %edx, %edx
je ..B1.16
..B1.15:
popcnt %edx, %edx
movl $65535, %r10d
vcompressps %zmm2, %zmm1{%k1}
movslq %ebx, %rbx
```

```
shlx %edx, %r10d, %r12d
notl %r12d
kmovw %r12d, %k1
vmovups %zmm1, (%rsi,%rbx,4){%k1}
addl %edx, %ebx
..B1.16:
addq $16, %r11
cmpq %r9, %r11
jb ..B1.14
```

5 **Compress idiom vectorization with AVX-512**

Automatic idiom recognition and vectorization of compress and expand idioms have been implemented in the compiler for simple cases without a guarantee for complicated cases. To provide a vectorization guarantee for these idioms, we proposed and implemented simple language extensions to the OpenMP 4.5 `simd` specification to express the compress and expand idioms. **Figure 6** shows the new clause `monotonic` added to the OpenMP `ordered simd` construct.

```
int j = 0;
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (A[i]>0) {
#pragma omp ordered simd monotonic(j:1)
        {
            B[j++] = A[i];
        }
}
    }
```

```
int j = 0;
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (A[i]>0) {
#pragma omp ordered simd monotonic(j:1)
        {
            A[i] = B[j++];
        }
}
    }
```

6 **Block-based syntaxes for compress and expand patterns**

The `monotonic` clause and its associated `ordered simd` code block impose the following semantics and rules:

- **When a vector iteration reaches the structured block of code, it evaluates step.** This must yield an integral or pointer value that is invariant with respect to the `simd` loop. The `simd` execution mask, under which the structured block of code is executed, is computed as well. Private copies of each monotonic list item are created for the structured block of code and initialized to `item, item + step, item + 2 * step, …,` for each executing (with mask==T) `simd` element (from lower iteration index to higher). At the end of the structured block of code execution, a uniform copy of item is updated as `item + popcount(mask) * step` and private copies of list items become undefined.
- **The use of the monotonic list item outside of the associated ordered `simd` construct is disallowed (Figure 7).** Multiple usages of the same list item in more than one ordered `simd` construct is also disallowed.

```
int j = 0;
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (A[i]>0) {
        #pragma omp ordered simd monotonic(j:1)
        {
            ++j;
        }
        B[j] = A[i];
    }
}
```

7 **Disallowed usage of a monotonic item outside of the block**

- **Ordered `simd` construct with monotonic clause** may be implemented as ordered `simd` construct without the monotonic clause.

Histogram

Histogram (**Figure 8**) is a well-known idiom. The semantics of the histogram code counts equal elements of the array `B[]` and stores results in the array `A[]`. Obviously, depending on the content of the array `B[]`, this loop may or may not have loop-carried dependency.

```
for (int i = 0; i < n; ++i) { ++A[B[i]]; }
```

8 **A simple histogram idiom**

Essentially, there are three cases:

1. **If the array `B[]` has unique values**, there is no loop-carried dependency.
2. **If all values of the array `B[]` are equal**, this statement represents reduction of a scalar value.
3. **If some values are repeated in the array `B[]`**, this loop has loop-carried dependency.

At compile time, if the compiler can't prove whether the values of `B[]` may or may not introduce data dependencies, the compiler behaves pessimistically and assumes loop-carried dependency. Programmers can help the compiler for case 1 by using `#pragma ivpdep`. For case 2, programmers may be asked to do manual transformation. For case 3, depending on the target architecture, vectorization can be implemented in the following ways with explicit vector annotations:

- **Array reduction implementation** (use `simd` construct and reduction over array `A[]`)
- **Serial implementation** (use ordered `simd` construct for the self-update statement)
- **Gather-update-scatter implementation** and repeat for equal `B[]` values

For the gather-update-scatter implementation, AVX-512 provides AVX-512CDI, which contains the `v[p]conflict` instruction that allows efficient computation of overlapping indices within a vector of indices. To effectively leverage AVX-512CDI with a programmer guarantee to handle the histogram idiom, a new clause overlap is proposed and implemented in the Intel compilers as shown in **Figure 9**.

```
#pragma omp simd
for (int i = 0; i < n; i++) {
    #pragma omp ordered simd overlap(B[i])
    {
        A[B[i]]++;
    }
}
```

9 **Usage example of the overlap clause on the histogram idiom**

This new clause has the following language semantics:

- **When execution reaches the ordered `simd` construct**, it evaluates the values of overlap expression `B[i]` as an r-value, which must yield an integral or pointer value. The implementation must ensure that if two iterations of the enclosing `simd` loop compute the same values, the iteration with the lower logical iteration number must complete the execution of ordered `simd` code block before the other iteration starts execution of the same code block.
- **Total ordering is not guaranteed by the compiler.** It is the programmer's responsibility to check whether the partial ordering is sufficient for any possible aliasing between stores to `A[]` and any loads of `A[]` and `B[]`.
- **The ordered `simd` block specifies the region where overlapping should be checked.** Thus, for statements outside of the ordered `simd` overlap block, the overlap check will not be performed.
- **The ordered `simd` construct with overlap clause** may be implemented as an ordered `simd` construct without the overlap clause.

Conditional Lastprivate

A common case in many programs is that a loop produces the value of its last iteration where the assignment statement is actually executed (i.e., produces a write to a scalar). The value of the scalar will be used after the loop (**Figure 10**).

```
int t = 0;
#pragma omp simd lastprivate(t)
for (int i = 0; i < n; i++) {
    t = A[i];
}
```

```
int t = 0;
for (int i = 0; i < n; i++) {
    if (A[i] > 0) {
        t = A[i];
    }
}
```

10 **The code on the left gets the last element of `A[i]`, while the code on the right gets the last positive element of `A[i]`.**

Both code snippets in **Figure 10** can be vectorized, but only the code on the left has explicit support in OpenMP through the `lastprivate` clause. The `lastprivate` clause can't be used for the code on the right because of its semantics. The scalar `t` gets the value from the last executed iteration of the loop; thus, the `t` is undefined if the condition `A[n-1]>0` is not true. To address this issue, a new modifier, conditional, is proposed and added to the clause `lastprivate` (**Figure 11**).

```
int t = 0;
#pragma omp simd lastprivate(conditional:t)
for (int i = 0; i < n; i++) {
    if (A[i] > 0) {  t = A[i]; }
}
```

11 **A `lastprivate` example of supporting conditionally assigned scalars**

Here are the language semantics of the new `conditional` modifier:

- **A list item that appears in a `conditional lastprivate` clause** is subject to the private clause semantics. At the end of the `simd` construct, the original list item gets the value as if the lexically last conditional assignment happened during scalar execution of the loop.
- **The item should not be used** if its use happens prior to its definition or it's outside of the definition's scope.

Loops with Early Exits

The existing `simd` vectorization in OpenMP 4.5 does not support loops with more than one exit. For example, a common usage such as finding an index of some elements in an array (**Figure 12**) can't be explicitly vectorized properly with the existing `simd` construct.

```
for (int i = 0; i < n; ++i) {
    if (A[i] == B[i]) {
        j = i; break;
    }
}
// use of j.
```

```
int j = 0;
#pragma omp simd early_exit lastprivate(conditional:j)
for (int i = 0; i < n; i++) {
    if (A[i] == B[i]) { j = i; break; }
}
// use of j.
```

12 **A loop with two exits that is searching for the first equivalent elements of the arrays `A[]` and `B[]`**

To address this issue, the Intel Compilers have introduced a new loop-level clause, `early_exit`, as shown in the right box of **Figure 12**, with the following semantics:

- Each operation before the last lexical early exit of the loop may be executed as if early exit were not triggered within the `simd` chunk.
- After the last lexical early exit of the loop, all operations are executed as if the last iteration of the loop was found.
- The last value for `linear` and `conditional lastprivate` is preserved with respect to scalar execution.
- The last value for reductions is computed as if the last iteration in the last `simd` chunk was executed upon exiting the loop.
- The shared memory state may not be preserved with regard to scalar execution.
- Exceptions are not allowed.
- If the innermost loop has an exit out of the `simd` construction, execution of the innermost loop is similar to if it had been completely unrolled.

Performance Results

Tables 2 through **4** provide performance results of micro-kernel programs for the `monotonic`, `overlap`, and `early_exit` extensions discussed previously. Performance measurement is done on an Intel® system with preproduction Intel Xeon Scalable processors running at 2.1GHz, in a 2-socket configuration with 24x 2666MHz DIMMs. Actual performance on specific configurations can vary. The loops are the examples shown in previous sections. Loop trip count n is 109, data types are all 32-bit integers, vector length is 16, and the compilation command line is `icc -xCORE-AVX512`. Baseline is scalar execution of the same set of loops.

Table 2 shows the normalized speedup of vectorized compress and expand idioms. In the "all false" column, compress/expand conditions are all false. Compressing stores or expanded loads are completely skipped. Vector evaluation of the condition, and the reduced number of iterations from vectorization, still lead to 1.17x to 1.19x speedup. For the "all true" column, compress/expand conditions are true; thus, all 16 elements are stored/loaded.

Table 2. Normalized speedup of compress/expand versus scalar execution. Each column has a different number of true conditions.

	All False	2 Elements	4 Elements	8 Elements	15 Elements	All True
Compress	1.20x	1.21x	1.25x	1.32x	1.81x	4.09x
Expand	1.20x	1.18x	1.18x	1.13x	1.35x	3.98x

In the middle columns, 2, 4, 8, or 15 elements of the 16-way vector are compress stored or expand loaded. This table indicates that regardless of the actual number of elements compressed/expanded, the vectorization of these idiomatic patterns is favorable compared to scalar execution. We ran the same experiment using aligned and misaligned compressed-to/expanded-from arrays. As we expected, trending was very similar between aligned and misaligned cases, except for the "all true" extreme, where the aligned cases have better cache line access characteristics.

Table 3 shows the normalized speedup of the vectorized histogram example. On the left, there is no overlap case, which is running 23 percent slower than scalar execution. Consider finding alternatives where the programmer can assert no-overlaps and thus avoid using overlap extension. Sometimes, choosing a shorter vector length may accomplish that. If conflict resolution is required for vector execution to make sense, other parts of the loop need to be highly profitable to vectorize. It's interesting to note that the eight repeats and full overlap (16 repeats) cases look more favorable than the cases with fewer repeats. This is likely due to register-based repeats in the vector execution catching up the speed of on-memory dependency over scalar execution. For more information on conflict performance, see reference 9 at the end of this article.

Table 3. Normalized speedup of Histogram versus scalar execution. Each column has a different number of conflicts within one vector iteration.

	No Overlap	2 Repeats	4 Repeats	8 Repeats	Full Overlap
Histogram	0.77x	0.65x	0.66x	0.82x	1.3x

Table 4 shows the normalized speedup of a vectorized search loop example. From left to right, the loop index value at which the match is found increases from 0 to 5,000. A few vector iterations are needed to achieve visible speedups from vectorization of the search loop.

Table 4. Normalized speedup of Search relative to scalar execution

Match At	i=0	i=5	i=10	i=50	i=100	i=500	i=1,000	i=5,000
Vector Iteration	1	1	1	4	9	32	63	313
Search Loop	1.0x	0.99x	1.02x	1.31x	1.54x	3.04x	4.12x	4.46x

It's important to note that standardization for the explicit vectorization techniques we've discussed is still in progress. Syntax, semantics, and performance characteristics may be subject to change in future implementations. Some features may remain as Intel-specific extensions.

Best Practices for AVX-512 Tuning

Setting the Baseline

All tuning work should start by setting an appropriate performance baseline. Tuning for Intel Xeon Scalable processors is no different. Let's assume that the programmer already has an application reasonably tuned for Intel Xeon processors using the `-xCORE-AVX2` flag of Intel Compilers 17.0. We recommend starting from the following three binaries using Intel Compilers 18.0:

1. Build the binary with `-xCORE-AVX2` (`/QxCORE-AVX2` for Windows)
2. Build the binary by replacing `-xCORE-AVX2` with `-xCORE-AVX512` (`/QxCORE-AVX512` for Windows)
3. Build the binary by replacing `-xCORE-AVX2` with `-xCORE-AVX512 -qopt-zmm-usage=high` (`/QxCORE-AVX512 /Qopt-zmm-usage=high` for Windows)

Although we expect overall performance of binary-B and/or binary-C to be on-par or better than binary-A, it's still best to double-check.

Read Optimization Reports for the Hotspots

Intel compilers produce an optimization report when compilation is performed with `-qopt-report` (`/Qopt-report` for Windows). Before trying to change the behavior of the compiler, it's best to understand what the compiler has done and what it knows already. Article 1 in the References section at the end of this article is a good introduction to using the optimization report. [**Editor's note**: "Vectorization Opportunities for Improved Performance with AVX-512" in ***The Parallel Universe* Issue 27** also has a good and more current overview of compiler reports.]

Compare Hot Spot Performance

Responses to the different compilation flag combinations can vary from one hotspot to another. If one hotspot runs fast with binary-B and another hotspot runs fast with binary-C, learning what the compiler did for those cases greatly helps in achieving the best for both.

Fine Tuning for ZMM Usage

For binary-B and binary-C, with the OpenMP `simd` construct, vector length can be explicitly controlled using the `simdlen` clause. Try using a larger or smaller vector length on a few hot spots to see how application performance responds. You may need to combine one or more tuning techniques.

Alignment

When you're using Intel AVX-512 vector load/store instructions, it's recommended to align your data to 64 bytes for optimal performance[6], since every load/store is a cache-line split whenever a 64-byte Intel AVX-

512 unaligned load/store is performed, given the cache line is 64 bytes. This is twice as much as the cache-line split rate of Intel® AVX2 code that uses 32-byte registers. A high rate of cache-line split in memory-intensive code may cause a 20 to 30 percent performance degradation. Consider the following struct:

```
struct RGB_SOA {
__declspec(align(64)) float Red[16];
__declspec(align(64)) float Green[16];
__declspec(align(64)) float Blue[16];
}
```

The memory allocated for the struct is aligned to 64 bytes, if you use this struct as follows:

```
RGB_SOA rgb;
```

However, if a dynamic memory allocation is used as follows, `the __declspec` annotation is ignored and the 64-byte memory alignment is not guaranteed:

```
RGB_SOA* rgbPtr = new RGB_SOA();
```

In this case, you should use dynamic aligned memory allocation and/or redefine the operator new. For AVX-512, align data to 64 bytes when possible using the following approaches:

- Use the `_mm_malloc intrinsic` with the Intel® Compiler, or `_aligned_malloc` of the Microsoft* Compiler for dynamic data alignment—e.g.: `DataBuf = (float *)_mm_malloc (1024 * 1024 * sizeof(float), 64);`
- `Use __declspec(align(64))` for static data alignment—e.g.: `_declspec(align(64)) float DataBuf[1024*1024];`

Data allocation and uses may happen in different subroutines/files. Thus, the compiler working on the data usage site often does not know that the alignment optimization has been made at the allocation site. `__assume_aligned()/ASSUME_ALIGNED` is a common assertion to indicate alignment at the data usage site. (See article 2 in the References section of this article and the applicable Intel Compiler Developer Guide and Reference for details.)

To Peel or Not to Peel?

The vectorizer often produces three versions of the same loop:

1. One processing the beginning of the loop (called the peel loop)
2. One for the main vector loop
3. One processing the end of the loop (called the remainder loop)

The peel loop is generated for alignment optimization purposes, but this optimization is a double-edged sword. If the loop trip count is small, it may lower the number of data elements processed by main vector loop. If the compiler chooses a wrong array to peel for, it can worsen the data alignment of the loop execution. The compiler flags `-qopt-dynamic-align=F` (`/Qopt-dynamic-align=F` for Windows) can be used to suppress the loop peeling optimization. The pragma `vector unaligned` can be used on a per-loop basis. The pragma `vector aligned` is even better if data is all aligned when entering the loop.

Vectorize Peel/Remainder and Short-Trip-Count Loops

The AVX-512 masking allows creation of a vectorized peel and remainder loops and vectorization of the short-trip-count loops. However, the vectorized versions of those special loops may not always result in performance gains, since most of the instructions need to be masked. The main reasons for possible performance problems are restrictions for store forwarding and possible stalls on merge-masked operations, when operation depends on others. In the compiler, the store forwarding issue can be addressed by thorough analysis of data flow and avoiding masking whenever possible. When the store-forwarding issue becomes inevitable, the unmasked version of the loop (e.g., scalar or vectorized loop with a shorter vector length) is preferred. You can avoid stalls on merge-masked operations by trying to use zero-masking operations. The following pragmas make it possible to provide hints to alter the compiler behaviors:

- For remainder loops:
  - `#pragma vector novecremainder`, to not vectorize remainder loops
  - `#pragma vector vecremainder`, to vectorize remainder loop depending on compiler cost model
  - `#pragma vector always vecremainder`, to vectorize remainder loop always
- For short-trip-count loops:
  - `#pragma vector always`, to enforce vectorization
  - `#pragma novector`, to disable vectorization

Gather and Scatter Optimization

The gather and scatter instructions allow us to vectorize more loops. However, the vectorized code may or may not get performance benefits. The performance of loops that contain gather/scatter code depends on the ratio between the number of calculations and the number of data loads/stores and on selecting an optimal vector length for reducing the latency of gather and scatter instructions. A shorter vector length implies less latency of gather/scatter code. Often, a couple of simple optimizations at the source code level can help reduce the number of gather and scatter instructions.

Figure 13 shows the two simple manual gather/scatter optimizations while the compiler is trying to automatically perform these optimizations. The first optimization is to reduce number of gathers/scatters when the results of two gathers/scatters are blended as shown on the left. In this case, blending the indices makes it possible to have one gather instead of two as shown on the right. Note that we can apply the same process to scatter optimization.

The second gather/scatter optimization is an opposite transformation (**Figure 14**). In this case, a gather is performed with indices resulting from blend of unit-stride linear indices as shown on the left. To improve performance, it's better to perform two unit-stride loads and then do the blend as shown on the right.

```
// Two gathers in the loop
float *a, sum = 0; int *b, *c;
… …
for (int i; i < n; i++) {
  if ( pred(x[i]))
      sum += a[b[i]]; // gather
  else
     sum += a[c[i]]; // gather
}
```

```
// Optimized with one gather in the loop
for (int i; i < n; i++) {
  int t;
  if ( pred(x[i]))
     t = b[i];
  else
     t = c[i];
  sum += a[t]; // one gather remain
}
```

13 **Reducing gathers in the loop**

```
// One gather in the loop
float *a, sum = 0; int a;
for (int i; i < n; i++) {
  int t;
  if (pred(x[i])) {
      t = i + b;
  }
  else {
    t = i;
  }
  sum += a[t]; // gather
}
```

```
// Replace gather with unit-stride loads + blending
for (int i; i < n; i++) {
  float s;
  if ( pred(x[i])) {
      s = a[i + b];  // unit stride vector load
  }
  else {
      s = a[i];      // unit stride vector load
  }
  sum += s;
}
```

14 **Reducing gathers with two unit-stride loads + blend in the loop**

Execution Latency Improvement

For loop vectorization, a known issue for compilers is that the loop trip count is unknown at compile time. However, programmers can often predict the trip count and provide a hint to the compiler using `#pragma loop count`. In some cases, the loop trip count can be predicted approximately based on `#pragma unroll` information as well. Preferably, the `unroll` pragma should be used on loops with small enough bodies to decrease number of loop iterations and to increase loop iteration independency, so they can be executed in parallel on out-of-order Intel® architectures to hide execution latency. Loop unrolling is very helpful for computation-bounded loops (i.e., when the computation takes more time than memory accesses). For example, the loop shown in **Figure 15** will execute about 45 percent faster when the unroll-factor (UF) is set to 8 compared to a UF equal to 0. The measurement was done for n=1,000.

```
float *a,*b, *c;
#pragma unroll(8)
#pragma omp simd
for (i= 0; i < n; i++) {
  if (a[i] > c[i]) sum += b[i] * c[i];
}
```

15 **Reducing latency using `simd + unroll(8)`**

Summary and Future Work

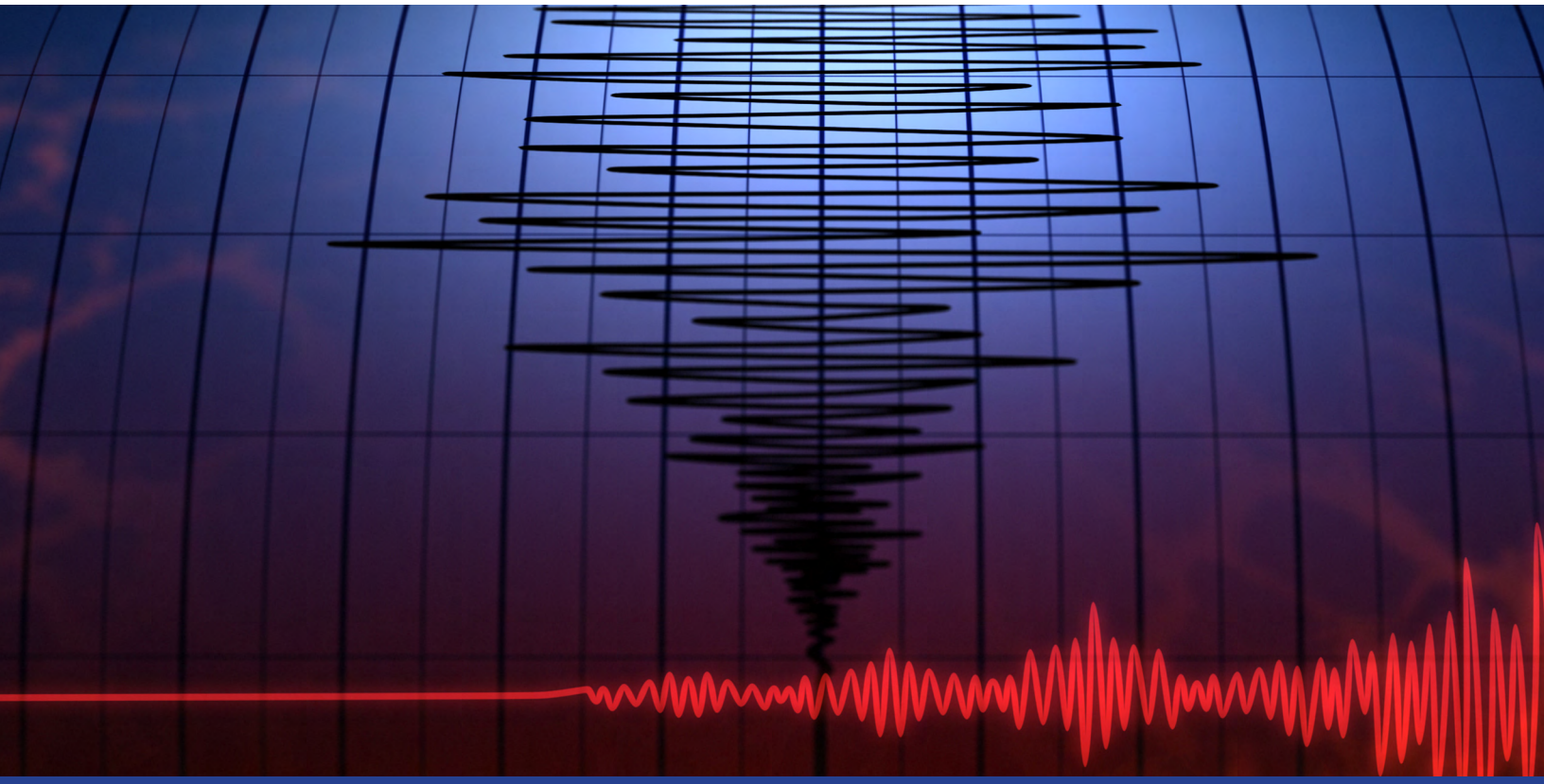
We've looked at several new `simd` language extensions for Intel AVX-512 support in Intel Compilers 18.0 for Intel Xeon Scalable processors. We shared and discussed a set of performance optimization and tuning practices for achieving optimal performance with AVX-512. We provided performance results based on microkernels to demonstrate the effectiveness of these new `simd` extensions and tuning practices. In the future, there are several new `simd` extensions for C/C++ under consideration for taking full advantage of AVX-512 through OpenMP and the C++ Parallel STL, including inclusive-scan, exclusive-scan, range-based C++ loops, and lambda expression support.

Learn More

- **Intel Advanced Vector Extensions 512 >**
- **Intel Xeon Scalable processors >**

References

1. M. Corden. "Getting the Most out of your Intel® Compiler with the New Optimization Reports," Intel Developer Zone, 2014.
2. R. Krishnaiyer. "Data Alignment to Assist Vectorization," Intel Developer Zone, 2015.
3. H. Saito, S. Preis, N. Panchenko, and X. Tian. "Reducing the Functionality Gap between Auto-Vectorization and Explicit Vectorization." In Proceedings of the International Workshop on OpenMP (IWOMP), LNCS9903, pp. 173-186, Springer, 2016.
4. H. Saito, S. Preis, A. Cherkasov, and X. Tian. "Obtaining the Last Values of Conditionally Assigned Privates," OpenMPCon Developer Conference, 2016.
5. H. Saito, "Extending LoopVectorizer: OpenMP4.5 SIMD and Outer Loop Auto-Vectorization," presentation at LLVM Developer Conference, 2016.
6. X. Tian. H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A. Duran. "Putting Vector Programming to Work with OpenMP* SIMD," *The Parallel Universe* magazine, Issue 22, September 2015.
7. X. Tian, R. Geva, B. Valentine. "Unleash the Power of AVX-512 through Architecture, Compiler and Code Modernization," ACM Parallel Architecture and Compiler Technology, September 11-15, 2016, Haifa, Israel.
8. X. Tian, H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A. G. Cherkasov, C. Nelson, N. Panchenko, R. Geva. Compiling C/C++ SIMD Extensions for Function and Loop Vectorizaion on Multicore-SIMD Processors. IEEE IPDPS Workshops 2012: 2349-23
9. Intel Corporation. "Conflict Detection" In Section 13.16 of Intel® 64 and IA-32 Architectures Optimization Reference Manual, July 2017.



EFFECTIVELY USING YOUR WHOLE CLUSTER

Optimizing SPECFEM3D_GLOBE* Performance on Intel® Architecture

Rama Kishan Malladi, Technical Marketing Engineer, Intel Corporation

Intel is a leading provider of both hardware and software for datacenter users—such as the latest **Intel® Xeon®** and **Intel® Xeon Phi™** processors. However, many high-performance computing (HPC) applications don't make full use of the processors' advanced capabilities. In this article, we'll provide a step-by-step methodology to improve the performance of **SPECFEM3D_GLOBE***, a software package that simulates three-dimensional global and regional seismic wave propagation and performs full waveform imaging (FWI) or adjoint tomography based on the spectral element method (SEM). All SPECFEM3D_GLOBE software is written in Fortran* 2003 with full portability in mind, and conforms strictly to the 2003 standard. The package uses the Message Passing Interface (MPI) to express distributed-memory parallelism. Recently, OpenMP* shared-memory parallel constructs were introduced in the solver source code.

Follow these steps to quickly get started building and running this code:

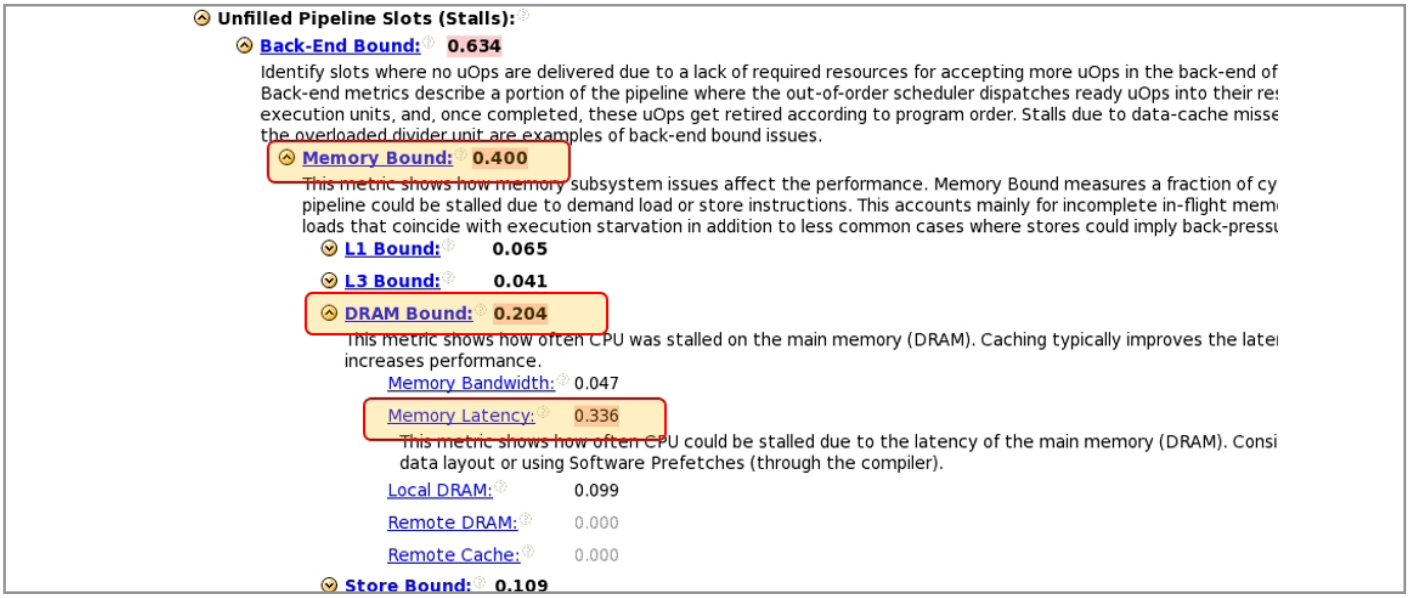
```
$ cd specfem3d_globe
$ ./configure FC=ifort MPIFC=mpiifort CC=icc CXX=icpc FCFLAGS="-O3 -xMIC-AVX512 -qopenmp"
$ cp EXAMPLES/small_benchmark_run_to_test_more_complex_Earth/Par_file DATA
$ make -j 8 xcreate_header_file xmeshfem3D xspecfem3D
$ cd EXAMPLES/small_benchmark_run_to_test_more_complex_Earth
$ ./run_this_example.sh
```

Notes

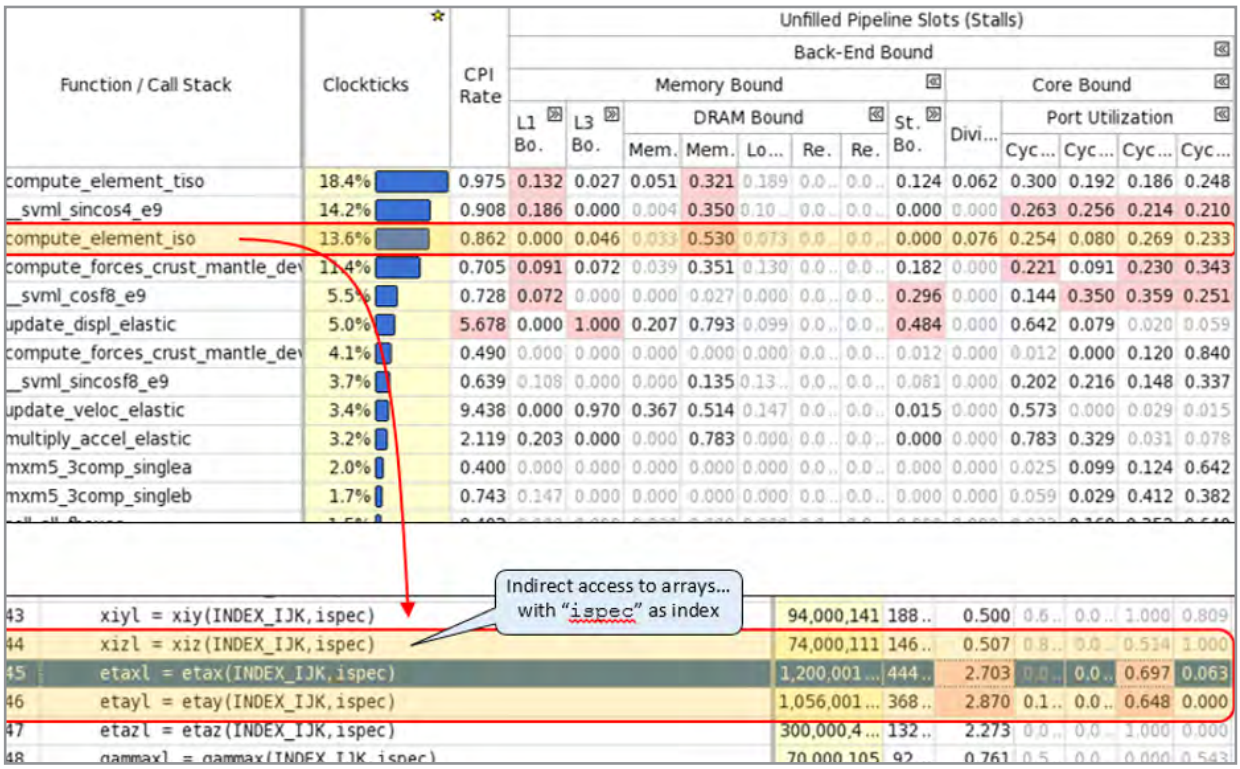
- Download SPECFEM3D_GLOBE from https://geodynamics.org/cig/software/specfem3d_globe/.
- The test benchmark being executed is small_benchmark_run_to_test_more_complex_Earth in the EXAMPLES directory.
- The Intel® compilers and MPI library are used for this build. Hence, the 'icc', 'icpc', 'ifort', and 'mpiifort' configuration options.
- The build/run machine contains Intel Xeon Phi (Knights Landing) processors so the compiler flag '-xMIC-AVX512' is used.
- The '-qopenmp' flag builds the package with OpenMP threads enabled.
- Performance of the solver code is measured by the time taken to simulate 'n'"time-steps for a given mesh volume.
- The steps to edit model resolution, parameters, and number of MPI processes are available in the SPECFEM3D_GLOBE documentation.

Assuming that the solver ran successfully, we need to know if it executed efficiently, making the best use of the resources available. A profile of the runtime execution will give us a sneak peek at the time spent in various subroutines of the solver code. We can collect such a profile across multiple cluster nodes to monitor MPI behavior plus each node's execution statistics. Tools like Intel® MPI Performance Snapshot and **Intel® Trace Analyzer and Collector** have features to collect an MPI execution profile. In addition, single-node profiles can be collected using **Intel® VTune™ Amplifier** and **Intel® Advisor**. SPECFEM3D_GLOBE has excellent MPI scalability with its asynchronous MPI communication/computation overlap, so our focus will be on per-node application profiling and optimization.

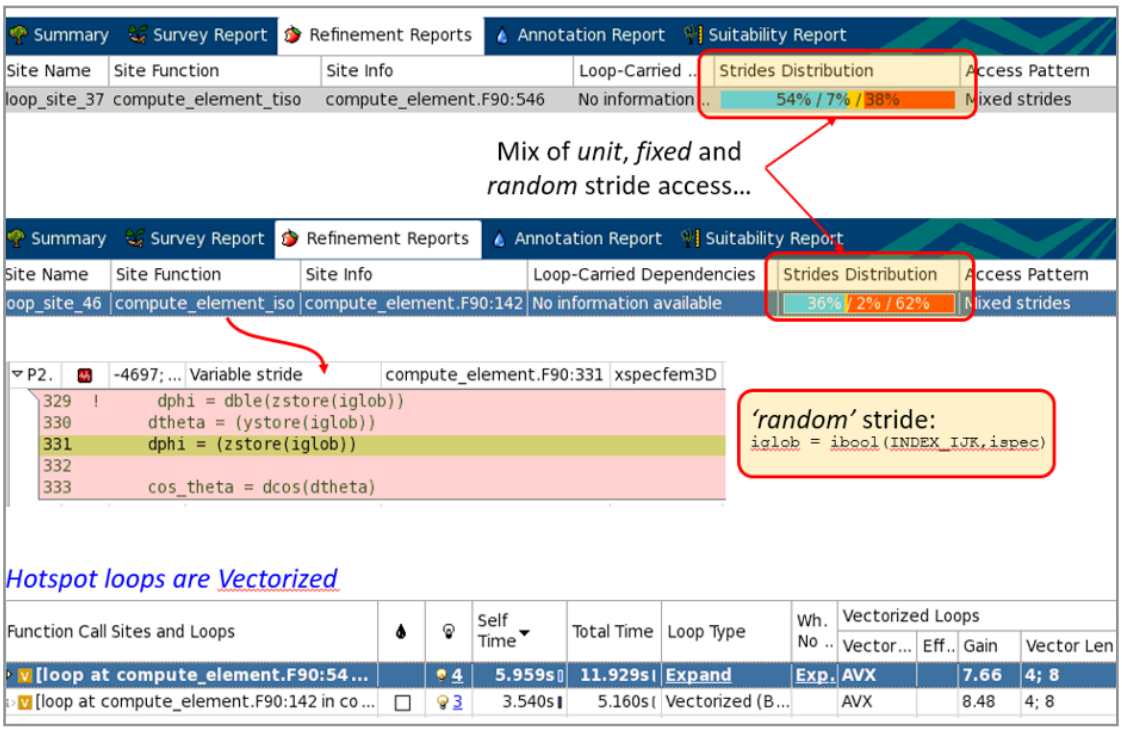
A general exploration profile collected using Intel VTune Amplifier shows that this code is backend-bound with further classification showing it has a memory (DRAM) latency issue. The summary view of general exploration is shown in **Figure 1**. Drilling down to hotspots and then the source code gives us the profile shown in **Figure 2**. The `etax` arrays, accessed in the `compute_element_iso` subroutine, have a dimension of 125 x N. The `INDEX_IJK` increments from 1 to 125 and the index `ispec` is arbitrary—hence, an indirect access. Tools such as Intel Advisor give you insight into the randomness of this access and also vectorization of the code (compiler generated). **Figure 3** shows such a profile.



1 **Intel® VTune™ Amplifier general exploration**



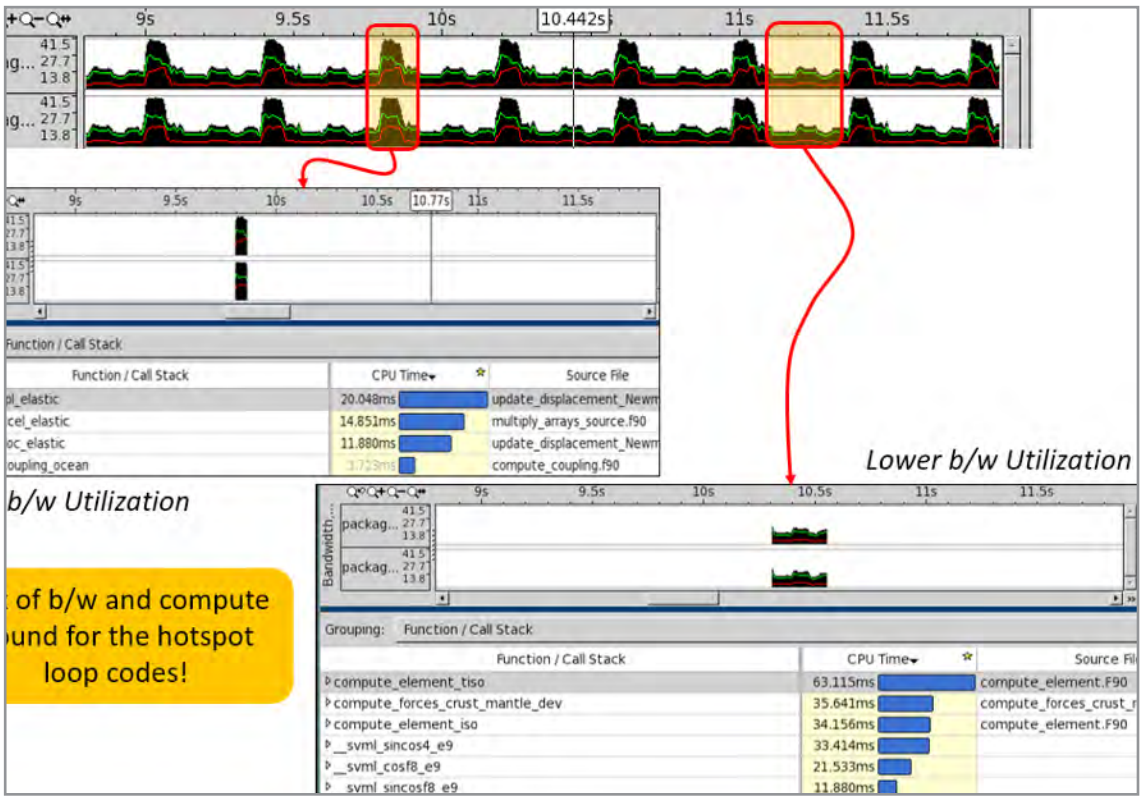
2 **Intel VTune Amplifier drill-down showing hotspots, source code**



3 **Intel Advisor: Unit/random stride distribution, source drill-down, vector intensity/code-generated**

Intel Advisor shows information about memory access strides—including whether they are unit, random, or fixed stride—and their distribution. Also, it gives information about compiler code generation and the vector length used for a loop execution, the instruction set, and vectorization gain. This profile helps determine if an application benefits from the vector features and the newer instructions in the processor. Sometimes, a loop may be vectorized by the compiler to utilize the full vector width of the processor (e.g., AVX-512*) but a corresponding speedup in the loop execution isn't observed. One reason could be that it is bandwidth-limited. Intel VTune Amplifier allows for such a profiling (**Figure 4**). It's important to understand that optimizations to improve vector performance of loops that are bandwidth-bound (high utilization) will be less fruitful. The low-bandwidth utilization loops and code regions could be executing non-vector instructions and/or suffering from memory access latency issues. This needs to be fixed.

After analyzing the execution profile of SPECFEM3D_GLOBE, we attempted some code changes to improve performance on Intel® processors.



4 **Intel VTune Amplifier memory bandwidth analysis**

Mitigate Memory Access Latency Issues

An indirect (random) access was transformed into a unit-stride access. Much of the mesh data in the SPECFEM3D_GLOBE solver is invariant over time/solver steps. Hence, it is a valid transformation to copy data and make it a linear access.

Original Code

```
subroutine compute_element_tiso(ispec,
...
    c13 = 0.125_CUSTOM_REAL*cosphisq*(rhovphsq + six_eta_aniso*rhovphsq \
...
```

Initialization/Startup Code

```
prepare_timerun (...)
...
    do ispec_p = 1,num_elements
      ele_num = ele_num + 1
...
          ia_c1store(idx6+3,tiso_ele_num) = c13
```

Optimized Code

```
subroutine compute_element_tiso(ispec, ele_num, &
...
    c13 = ia_c1store(idx7+3,tiso_ele_num)
...
```

This code modification copies all nine arrays—'`xix`,' '`xiy … gammzl`'—into one array, '`ia_arr`'. This will help alleviate bandwidth pressure for these accesses (because only one array access needs to be done (instead of nine) but is less vector-friendly (because the elements of each array aren't contiguous). For cases where bandwidth pressure isn't much of a concern, a user can create nine arrays corresponding to the original code and get vectorization benefit. The best option must be determined by experiment.

Compiler Vectorization/Loop Fission

The compute loops '`iso`' and '`tiso`' are huge. The compiler is unable to vectorize these loops. So, a manual loop fission was done. A similar effect can be realized by using '`!DIR$ DISTRIBUTE POINT`' syntax supported by Intel® compilers for loop distribution/fission.

Data Alignment/Padding

The compute loops '`iso`' and '`tiso`' are invoked for each element in the mesh and are invoked from either an MPI or a thread region. These loops have trip count of 125. Since the arrays accessed in the loop have the dimensions 125 x N, another optimization applied to this code was to make it aligned to a 2n boundary. A padding of three elements was applied to make it a 128 x N array.

Original Code

```
subroutine compute_element_tiso(ispec,
...
    c13 = 0.125_CUSTOM_REAL*cosphisq*(rhovphsq + six_eta_aniso*rhovphsq \
...
```

```
prepare_timerun (...)
...
    do ispec_p = 1,num_elements
      ele_num = ele_num + 1
...
          ia_c1store(idx6+3,tiso_ele_num) = c13

-----------------------------------------------------------------
```

Optimized Code

```
subroutine compute_element_tiso(ispec, ele_num, &
...
    c13 = ia_c1store(idx7+3,tiso_ele_num)
...
```

Replace Redundant Computation with Lookup Tables

The 'tiso' loop has some computations that invoke transcendental functions. These computations are invariant with solver execution time-steps. The same can be replaced with lookup tables.

IVDEP or SIMD Directives

Some hotspots in the SPECFEM3D_GLOBE solver are nested loops with trip counts 5 x 5 and 5 x 25. These are 'm x m' matrix-matrix multiplications. The compiler optimization reports (`use -qopt-report flag`) indicated that not all these loops were vectorized. Using IVDEP or SIMD directives helped the compiler to generate vector code for these loops.

In conclusion, some simple code changes (and data transformations) improved the performance of SPECFEM3D_GLOBE solver by approximately 2.1X on an Intel Xeon Phi processor-based system. There's room for further optimization—and it's being explored.

Configuration and Tools Used

- Intel Parallel Studio XE 2017
- Self-boot system with Intel Xeon Phi 7250 processor, 96GB DDR memory
- MCDRAM on Intel Xeon Phi processor configured to FLAT mode. Mesh interconnect in QUAD mode.
- OS version: CentOS* Linux* release 7.3.1611 (kernel 3.10.0-514.10.2.el7.x86_64, glibc 2.17-157. el7_3.1.x86_64)

References

- # SPECFEM3D_GLOBE: Komatitsch and Tromp 1999; Komatitsch and Vilotte (1998): https://geodynamics.org/cig/software/specfem3d_globe/
- Intel Software tools/manuals: https://software.intel.com/en-us/intel-parallel-studio-xe
- Intel® 64 and IA-32 Architectures Software Developer Manuals: https://software.intel.com/en-us/articles/intel-sdm
- Ahmad Yasin. “A Top-Down Method for Performance Analysis and Counters Architecture.” IEEE Xplore: 26 June 2014. Electronic ISBN: 978-1-4799-3606-9.

IS YOUR CLUSTER HEALTHY?

Must-Have Cluster Diagnostics in Intel® Cluster Checker

Brock A. Taylor, HPC Solution Architect, Intel Corporation

Intel® Cluster Checker is a powerful tool for quickly identifying and solving issues in high-performance computing (HPC) clusters. Subtle and sometimes simple issues on a system can impact cluster performance and blunt the efforts of fine-tuning and parallelizing an application. Often, the first signs of a system issue appear when applications run too slowly—or simply stop running altogether. Intel Cluster Checker provides a methodical way to help quickly determine if the underlying reason an application is experiencing problems is actually a problem with the cluster.

Cluster Systems Expertise in a Tool

Intel Cluster Checker captures best-known methods and system diagnostics in a single tool. Introduced in 2007 as part of the **Intel® Cluster Ready Program**, a key goal was to provide a tool that would assist a broad range of people who design, deploy, and manage clusters.

Building and managing HPC clusters is far more complex than managing single systems. By their nature, the most powerful systems in the world, the **TOP500**, are custom-built systems with dedicated staff to operate them. Many of these environments grew organically over years, maintained by solution architects with deep knowledge in this space.

The problem is that the approach of large HPC data centers doesn't necessarily scale down to smaller systems. The high level of expertise required can intimidate small and medium businesses. Even larger enterprises considering moving to HPC clusters must weigh the time and effort it takes to ramp capabilities. The learning curve can look like a mountain too steep to climb—even though the return on investment for using HPC is substantial. [**Editor's note:** According to a study by Hyperion Research, every dollar invested in HPC yields $551 of revenue growth and $52 of profit. (Sources: **IDC Economic Models Linking HPC and ROI** and **Hyperion (IDC) Paints a Bullish Picture of HPC Future**)] Intel Cluster Checker provides expertise in a tool to lower the intimidation factor for people ramping HPC capabilities.

Intel Cluster Checker works like a clinical system. It looks for signs that an issue exists and then examines the signs holistically to diagnose issues—and potentially even suggest remedies. Data providers encapsulate common diagnostic tools and functions, and the tool uses these providers to collect information about the cluster. A rules-based expert system then analyzes the information to produce signs that may indicate issues. A combination of different signs can lead to a diagnosis, and the tool can often suggest a remedy. In this way, Intel Cluster Checker models an expert analysis of cluster functionality and makes it easier to resolve issues quickly.

What Does It Check?

Intel Cluster Checker includes a broad range of data providers and rules that target common issues that cause system failures or performance degradations. At a high level, Intel Cluster Checker looks at elements of individual nodes and basic functionality, stepping up cluster-wide functionality. It's not feasible to list everything it checks, but here are some examples:

- **It checks** if the user running the tool has SSH keys set properly for executing message-passing interface (MPI) parallel applications.
- **It verifies** that the firmware version of the add-in network card is the same on each node of the system, examining library version and software consistency across the cluster.
- **It finds** differences in processor steppings, memory, and hardware components.
- **It uncovers** differences in configurations of both hardware and software components.

It also uses some common benchmarks to try to gauge how actual performance compares to expected performance. These functions are valuable at deployment time to declare a system ready for use. They also play a role in maintaining overall system health. There are hundreds of aspects of a cluster examined today, and the list of available checks keeps growing with update releases of the tool.

Over the operational lifetime of a cluster, subtle changes may be introduced with replacement parts, expansion of nodes, and reconfigurations of software or hardware. For example, a replacement network card could be plugged into a different PCIe* slot than before. New nodes added to the cluster could have a different Intel® Xeon® processor than the other nodes. Someone could have accidentally skipped updating the new BIOS settings on one of the nodes.

Over the operational lifetime of a cluster, subtle changes may be introduced with replacement parts, expansion of nodes, and reconfigurations of software or hardware.

Intel Cluster Checker helps find these issues and calls attention to them. None of them may actually be a problem for a particular system or application, but the tool highlights items to examine. Using Intel Cluster Checker can make running clusters less daunting for those who don't have deep knowledge of cluster administration and management, and it can augment the toolset for those who do.

In addition to cluster health, Intel Cluster Checker can also verify that a cluster provides the application compatibility described in the **Intel® Scalable System Framework (Intel® SSF) reference architecture**. The Intel SSF reference architecture describes system requirements that define a minimum level of system characteristics. Some of these characteristics include elements of the system software for Linux*-based clusters as well as minimum requirements for system hardware. Clusters that comply with the specification provide a common platform interface that application developers can target. Applications that build on this common layer execute on any system that complies with the reference architecture. This pairing of applications and systems enables interoperability that also simplifies the ramp to using HPC clusters.

Extending and Embedding Functionality

The technologies and components that comprise clusters are constantly evolving—which increases the potential for new types of problems. Because of this, extensibility is a key feature for Intel Cluster Checker

to keep pace with the scope of issues users face. Once a particular type of problem is known, capturing and adding the mechanisms to detect and resolve these new issues makes checking for them routine. Users can even create their own data providers and checks and include them in the same fashion. Intel Cluster Checker 2018 provides the capability to group data collection and analysis functions into frameworks. These frameworks allow flexibility in how the tool operates and provide a quick way to drop in new checks to extend capabilities.

Application developers can also embed Intel Cluster Checker functionality directly into their applications using an API to control data collection and analysis. Embedding functionality provides a range of options that developers can take advantage of, similar to running Intel Cluster Checker from the command line. Applications can check general health or compliance with the Intel SSF architecture. It also means developers can add customized rules that look for aspects of the system that are specific to the application's needs. It provides a programmatic mechanism to perform system checking and debugging from the application's point of view. The application can find underlying issues on a cluster and inform the user of potential problems. Examples of using the API are included in the online documentation for the tool.

Focus on Productivity

Variations in configurations, a mix of hardware and software components, or the state of system health can all manifest as problems for an HPC application. Using Intel Cluster Checker helps identify when systems are in a known, healthy state. That promotes a better out-of-the-box application experience for users. If problems do exist, the tool can quickly direct users to potential resolutions. Ultimately, this lowers the expertise barrier of running HPC clusters and opens doors for more users running cluster applications to achieve bigger and better results.

Intel Cluster Checker is currently available as part of **Intel® Parallel Studio XE** Cluster Edition. It's also provided on systems using **Intel® HPC Orchestrator** and may be included in solutions that comply with the Intel SSF reference architectures for classic HPC clusters.

Learn more about Intel Cluster Checker >

OPTIMIZING HPC CLUSTERS

Enabling On-Demand BIOS Configuration Changes in HPC Clusters

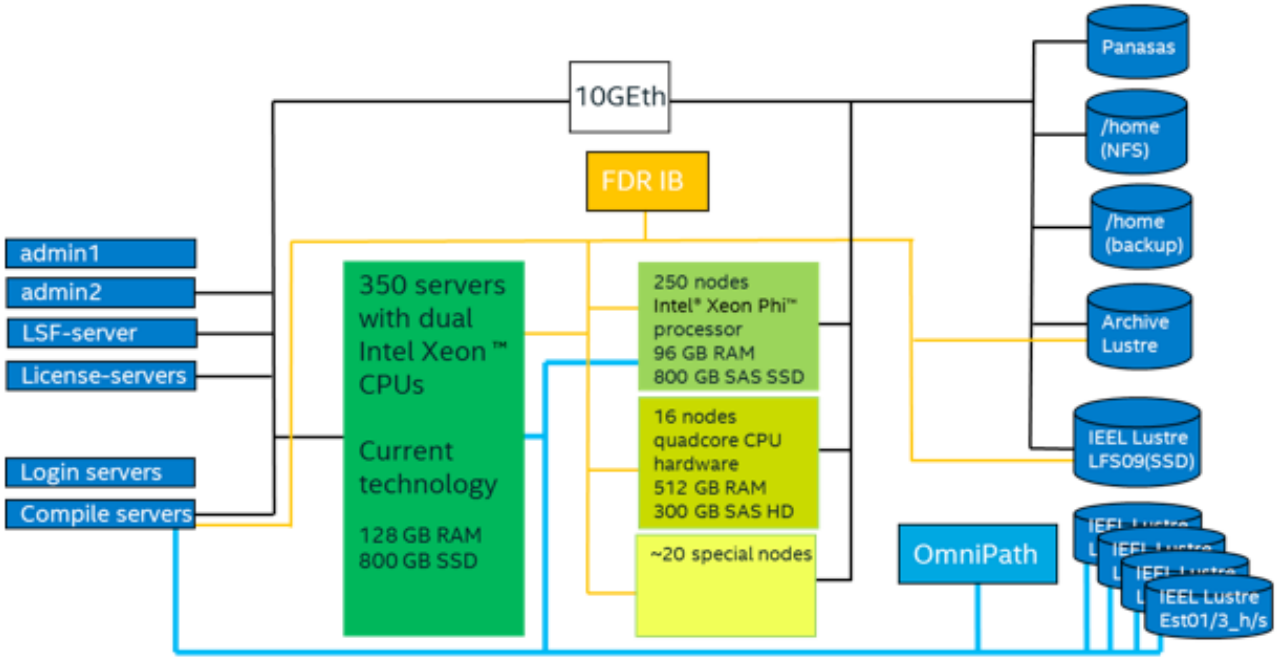
Michael Hebenstreit, Data Center Engineer, Intel Corporation

Since around 2000, most high-performance computing (HPC) systems have been set up as clusters based on commodity x86 hardware. These clusters consist of one- or two-socket servers to perform the actual computations, plus storage systems and administrative nodes.

Modern **Intel® Xeon® processor**-based systems, as well as the Linux* kernel, provide many ways to optimize both hardware and operating system (OS) for a specific application. It's easy to do if your cluster is only used for a specific workflow—but for more complex usages, it's beyond the ability of most cluster managers.

There is a way to perform complex optimizations on a per-job basis. Intel tested this idea in its benchmarking data center, which has approximately 500 compute nodes. The cluster, known as Endeavor,

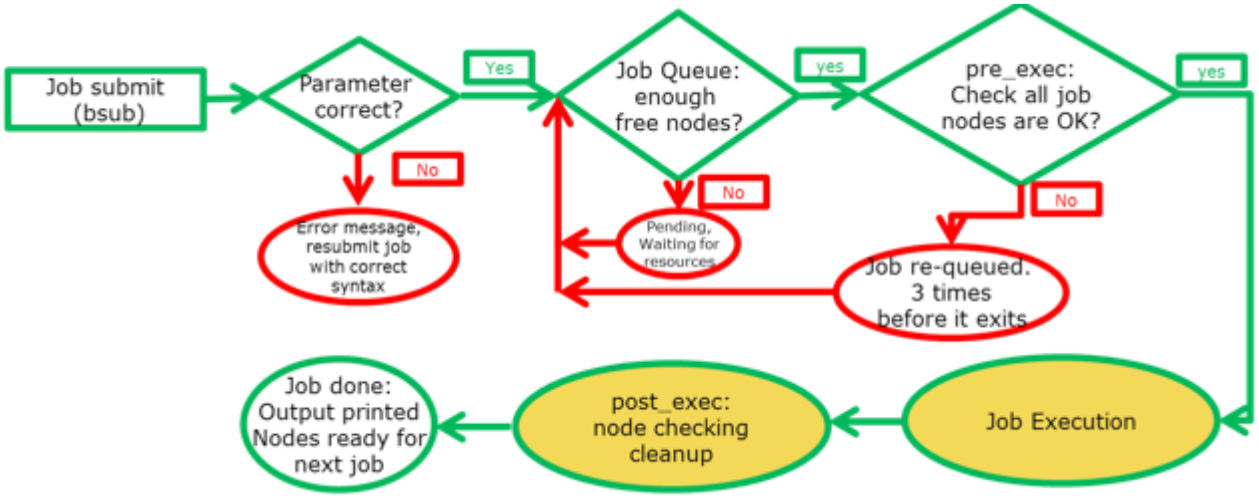
is rebuilt on a regular basis with the latest hardware and has been listed among the **TOP500 SuperComputer Sites** since 2006. **Figure 1** shows the layout of Endeavor.



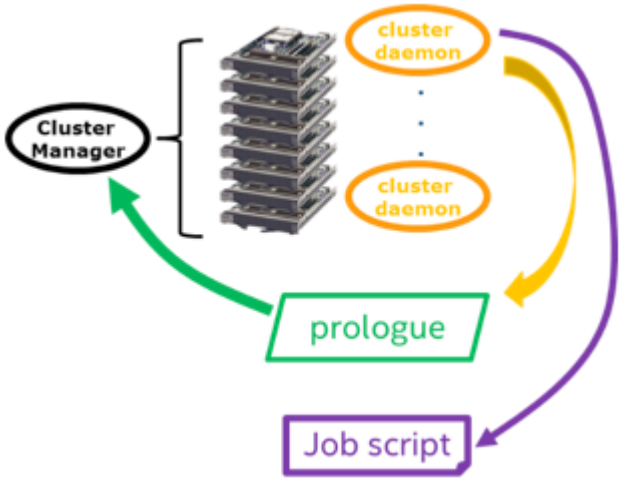
1 **Layout of Intel's Endeavor cluster**

Users typically connect to one of several login nodes, package up their workloads in the form of job scripts, and submit those jobs to a cluster manager. The cluster manager (e.g., Altair PBS Pro*, Bright Cluster Manager*, IBM LSF*, or Slurm*) is a scheduling tool trying to allocate the cluster resources as efficiently as possible. For the cluster manager, each job is simply a request to use X compute nodes for a Y amount of time.

When a user submits a job, the system checks to see if all necessary parameters are within sensible limits, and then waits until enough resources of the required type become free (**Figure 2**). Once the cluster manager can assign enough nodes, it runs a special program called the prologue (**Figure 3**). This program is usually executed on the first node (also called the headnode) assigned to the job. The purpose of the prologue varies, but it might be used to check that all nodes assigned to a job are in good health. Once the prologue successfully completes, the cluster manager starts the actual job on the nodes. In most cases, this is a shell script executed on the headnode. Once this script terminates in any way, the cluster manager cleans up the nodes, running an epilogue program and preparing the nodes for the next job.



2 **LSF job flow chart on Intel's Endeavor cluster**



3 **Prologue program**

This workflow, implemented by most cluster managers, faces a problem when it becomes necessary to reboot nodes as part of the prologue process. Rebooting might be necessary because:

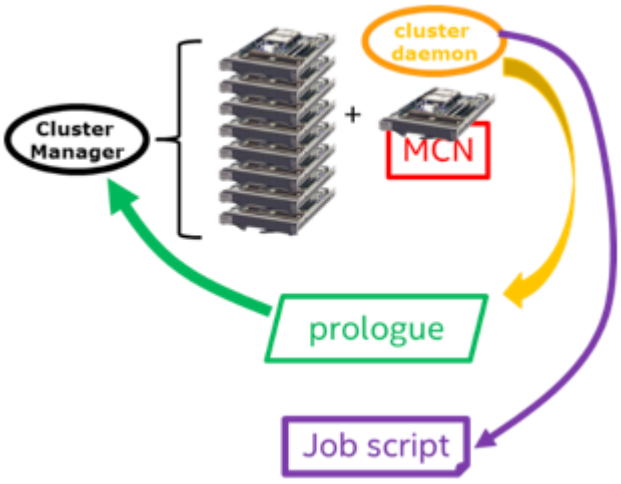
- The **Intel® Xeon Phi™ processor** is equipped with fast on-socket MCDRAM memory, usable either as a standard memory block or as a fourth-level cache. Using it as a cache will speed up programs automatically, but using it as standard memory might be even faster. Switching between those modes requires changing a BIOS option and rebooting the system.

- The **Intel® Xeon® processor** uses a mesh to communicate among the cores, cache, memory, and the PCIe controller. The BIOS Option Sub Numa Cluster allows a user to reconfigure the CPU and split it into virtual sockets.
- Modern Linux* kernels know the concept of NO_HZ cores. Normally, cores are interrupted every 100 to 1,000 ms to do kernel work and task switching. For typical HPC workloads, this behavior is counter-productive. With the NO_HZ parameter, one can configure the kernel to schedule only one interrupt per second. This decreases OS noise, increases scalability, and can improve performance but requires kernel changes and a system reboot.

With the standard approach of prologue scripts, the requirement of a reboot leads to a dilemma. The moment the headnode reboots, the cluster manager will assume the prologue has terminated, stop preprocessing of the job, and then reschedule it. A new set of nodes will be allocated for the job, a new headnode will be selected, and the prologue will execute—with exactly the same results.

Finding a Solution

Intel's HPC benchmarking cluster, Endeavor, currently uses IBM LSF*. The only simple and portable solution we could find was to add a master control node (MCN) to each job (**Figure 4**). This MCN would automatically become the headnode of a job. It would execute the prologue, use syscfg to make any changes to the BIOS configuration, and reboot the compute nodes using the Intelligent Platform Management Interface (IPMI). It would then wait for the nodes to come back up, check that everything is correct, and finish the prologue. If the prologue completes successfully, the job will start.



4 **Master control node**

Users would now see a small difference. Normally, they would request a number of nodes of the same type including the headnode. But for a reconfig-job, the headnode will be of a different type. The user now has two options:

1. **Login to the first node in the nodelist**, then continue to work as usual.
2. **Use an option in modern MPI versions** to exclude the headnode from the list of nodes used for job processing.

The latter approach has some advantages:

- Typically, the headnode of an MPI program has to start up additional processes—for example, an ssh process for every MPI process in the program. Since this additional load remains on the MCN, all compute nodes will have an identical system load.
- Specific to Intel Xeon Phi processor-based clusters, if the MCN is a standard Intel Xeon processor-based server, startup scripts and MPI initialization will process faster than on an Intel Xeon Phi processor-based system.

Implementation

On Endeavor, the prologue and epilogue not only check node health. The system also allows users to change parameters requiring root privileges. Since most of the code for the prologue and epilogue is identical, we use the same script in both cases, switching codepaths when necessary.

Prerequisites

Job Submission

We wanted to stay close to standard LSF syntax, so we wrote a small wrapper script around `bsub`, extending the command with a `-l` option. All special requests are translated into unique environment variables, since LSF transfers the environment of the user not only into the job, but also to the prologue and epilogue.

If necessary, the wrapper would automatically extend the resource requirements to include the control nodes:

```
$ bsub -R '2*{select[ekf]span[ptile=1]}' -l KNL_MEMMODE=1 run.sh
Warning '-l KNL_MEMMODE=1' will reboot compute nodes
Resource_List_KNL_MEMMODE=1
bsub.orig -R '1*{select[rebootctrl]} + 2*{select[ekf] span[ptile=1]}'  run.sh
…
```

The user requests two nodes of the type ekf. The script sets the corresponding environment variable. The selection string is expanded to include the control node type. With those two changes, the original LSF binary (renamed `bsub.orig`) is called.

Remotely Booting Nodes

To reboot nodes via the network, we use the IPMI.

Booting Cluster Nodes via Preboot Execution Environment (PXE)

PXE is already used in most clusters. For reconfiguration purposes, we made use of the way the pxelinux.0 binary queries the tftp server for boot configuration files (**Figure 5**).

```
Intel(R) Boot Agent XE v2.3.41
Copyright (C) 1997-2015, Intel Corporation

CLIENT MAC ADDR: 00 1E 67 94 A0 8F  GUID: FFFFFFFF FFFF FFFF FFFF FFFFFFFFFFFF
CLIENT IP: 36.101.34.10  MASK: 255.255.0.0  DHCP IP: 36.101.201.4
GATEWAY IP: 36.101.255.1

PXELINUX 4.02 0x53441223  Copyright (C) 1994-2010 H. Peter Anvin et al
!PXE entry point found (we hope) at 9580:0106 via plan A
UNDI code segment at 9580 len 5850
UNDI data segment at 8E3C len 7440
Getting cached packet  01 02 03
My IP address seems to be 2465220A 36.101.34.10
ip=36.101.34.10:36.101.201.4:36.101.255.1:255.255.0.0
BOOTIF=01-00-1e-67-94-a0-8f
TFTP prefix:
Trying to load: pxelinux.cfg/2465220A                              ok
boot:
Loading vmlinuz-3.10.0-514.6.2.0.1.el7.x86_64.knl1.....
Loading initramfs-3.10.0-514.6.2.0.1.el7.x86_64.knl1.img....._
```

5 **Preboot execution environment**

The first query is for a file named after the MAC address (01-00-1e-67-94-a0-8f). The second query is for a file named after the IP address assigned by the DHCP server coded in hexadecimal (2465220A). We use the second query for default boots and can therefore—temporarily—create a suitable file of the first type to override boot configurations for a specific job.

For this to work, we depend on a systematic PXE configuration. For each node a link nodename points to its IP address coded in hexadecimal. This second file is again a link pointing to the default configuration.

```
emhtest329 -> 2465220A
2465220A  -> ol7u3_sda6
```

We allow users only specific combinations prepared as special1, special2, and so on. The PXE boot directory contains next to the default configuration:

```
ol7u3_sda6
```

Files like:

```
ol7u3_sda6-k229sp0
ol7u3_sda6-k514sp1
ol7u3_sda6-k514sp2
…
```

Each represents a boot configuration. In this case, k229 and k514 indicate different kernel versions, `sp1,2,3…` all have special kernel options (e.g., `NOHZ_full`). If the user requests a specific configuration, the corresponding file for this node has to be present. So, in our example, the node `emhtest329` could be rebooted into the configuration `k515sp2` because the file `ol7u3_sda6-k514sp2` is present. But asking for `k514x5` would fail.

Changing BIOS Options

Intel provides the syscfg utility for Intel-manufactured motherboards, which allows reading and modifying BIOS parameters from Linux. Not all OEMs provide similar tools.

Integration Into the Cluster Manager

The integration into LSF is now straightforward. The prologue is automatically executed by LSF on the master control node. Early in the prologue, before any other checking or setup is done, the reconfiguration script needs to be executed on all compute nodes of the job (not on the control node itself). If a node declares that a reboot is needed, the prologue can use IPMI to reset it. It then waits until the reboot is complete. A maximum wait time ensures that nodes failing to boot will not wreck this scheme. During epilogue, the similar jobflow reestablishes node settings to their default values and, if necessary, reboots the nodes.

Anatomy of the Reconfiguration Script

The script is executed by the prologue on each node and reacts to a number of environment variables:

```
Prologue                          # 1 if prologue is executed

# user requirements from bsub
Resource_List_KNL_MEMMODE
Resource_List_KNL_CLUSTERMODE
Resource_List_Sub_NUMA_Cluster
Resource_List_SPECIAL_KERNEL
```

Intialization shows where the critical syscfg binary is and where the files are stored. We track if the node needs a reboot in the REBOOT variable:

```
SYSCFG=/usr/local/bin/syscfg
SAFEDIR=/var/lib/icsmoke3/safe
CURRENTDIR=/var/lib/icsmoke3/current
PXEDIR=/admin/tftpboot/3.0/pxelinux/pxelinux.cfg
REBOOT=no
HOSTNAME=`hostname`
```

The helper function helps when output from the syscfg command is not always directly usable as input. The `sed` command below will transform a line from syscfg.INI in the form:

```
Cluster Mode=Quadrant;Options: All2All=00: SNC-2=01: SNC-4=02: Hemisphere=03:
Quadrant=04: Auto=05
```

into its associated numerical value. It requires the variable `$I` to be set correctly.

```
convert_syscfg()
{
 echo "$1" | sed -e "s,${I}=Cache.*,0," -e "s,${I}=Flat.*,1," -e
"s,${I}=All2All.*,0," -e "s,${I}=SNC-2.*,1," -e "s,${I}=SNC-
4.*,2," -e "s,${I}=Hemisphere.*,3," -e "s,${I}=Quadrant.*,4," -e
"s,${I}=Disabled.*,0," -e "s,${I}=Enabled.*,1,"
}
```

Dump the current BIOS configuration to get all current settings:

```
cd $CURRENTDIR
/bin/rm syscfg.INI
$SYSCFG /s INI
```

After a job completes, the epilogue should run on all nodes.

```
if [ "$prologue" != "1" ]
then
```

We first check if a special PXE configuration already exists. If so, it should be removed. If it's not possible to remove the file, the script will fail with an error:

```
 # this is the PXE link used to boot to special kernel
 ADDR="01-`sed -e 's,:,-,g' /sys/class/net/eth0/address`"
 ADDRFILE="/admin/tftpboot/3.0/pxelinux/pxelinux.cfg/$ADDR"
 if [ -e "$ADDRFILE" ]
 then
  /bin/rm $ADDRFILE
  sleep 1  # wait for the cluster file system to catch up
  if [ -e "$ADDRFILE" ]
  then
   badmin hclose -C "wrong bootimage, fix $ADDRFILE" $HOSTNAME
   REBOOT=error
   exit 1
  fi
```

Check current values for BIOS options against expected values in `$SAFEDIR/syscfg.INI` and correct any differences. If there are any differences, set the variable `$REBOOT=yes`.

```
for I in “Memory Mode” “Cluster Mode” “Sub_NUMA Cluster” “IMC Interleaving”
 do
  CURRENT=`egrep “$I” $ CURRENTDIR/syscfg.INI`
  SAFE=`egrep “$I” $ SAFEDIR/syscfg.INI`
  if [ “$CURRENT” != “$SAFE” ]
  then
   VAL=`convert_syscfg “$SAFE”`
   $SYSCFG /bcs “” “$I”  “$VAL”
   REBOOT=yes
  fi
 done
```

Our kernel command lines contain a hint to whether the node is running the default kernel or anything special in need of a reboot.

```
grep -q “CRTBOOT=default” /proc/cmdline || REBOOT=yes
```

The rest of the script is only processed during prologue:

```
else
```

For every BIOS option, we check if the user supplied allowable values. Then the current configuration is compared against the one requested by the user. If a configuration change is necessary, we use syscfg to set the new value, and set the $REBOOT variable to indicate that rebooting is necessary, e.g.:

```
  case “$Resource_List_KNL_MEMMODE”   in 0|1)
   I=”Memory Mode”
   CURRENT=`egrep “$I” $CURRENTDIR/syscfg.INI`
   SAFE=`egrep “$I” $SAFEDIR/syscfg.INI`
   VAL=`convert_syscfg “$CURRENT”`
   test -n “$SAFE” && test “$VAL” != “$Resource_List_KNL_MEMMODE”
 && { $SYSCFG /bcs “” “$I”  “$Resource_List_KNL_MEMMODE” ;
REBOOT=yes; }
  esac
```

For logging purposes, we display the current values of all settings:

```
      # display current settigns
      $SYSCFG /d BIOSSETTINGS “Memory Mode”
      $SYSCFG /d BIOSSETTINGS “Cluster Mode”
      $SYSCFG /d BIOSSETTINGS “Sub_NUMA Cluster”
```

To configure a different kernel, the script first locates the standard PXE-config file for this node in $PXEDIR. That name is expanded by $Resource_List_SPECIAL_KERNEL. If the requested configuration file exists, a link with the name 01-MACADDRESS is created, and will take precedence on the next boot. The $REBOOT variable is set to “yes.”

```
 if [ -n “$Resource_List_SPECIAL_KERNEL” ]
 then
  echo “configuring for Kernel $Resource_List_SPECIAL_KERNEL”
  if ! `grep -q “CRTBOOT=${Resource_List_SPECIAL_KERNEL}” /proc/cmdline`
  then
   DEFAULT=`readlink -f $PXEDIR/$HOSTNAME`
   DIR=`dirname $DEFAULT`
   BASE=`basename $DEFAULT`
   if [ -e “$DIR/${BASE}-${Resource_List_SPECIAL_KERNEL}” ]
   then
    cd $DIR
    ln -s ${BASE}-${Resource_List_SPECIAL_KERNEL} “$ADDR”
    echo “created $DIR/$ADDR”
    ls -l “$DIR/$ADDR”
    REBOOT=yes
   else
    echo “can not set kernel to $DIR/${BASE}-${Resource_List_SPECIAL_KERNEL}”
   fi
  fi
 fi
fi
```

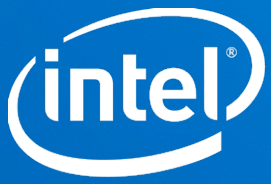
The script ends, producing an output of either "REBOOT=yes" or "REBOOT=no".

```
echo "REBOOT=$REBOOT"
```

The prologue script running on the control node will parse the output and, depending on this output, issue a reboot sequence via IPMI.

Optimizing Your Cluster

Modern Intel Xeon processor-based systems, as well as the Linux kernel, provide many options for optimizing the hardware and OS for a specific application. We've outlined a way to perform complex optimizations on a per-job basis. There's a price to pay in the form of added complexity and job startup times, but for Intel's HPC benchmarking cluster, Endeavor, this feature became a very important way to boost performance over the last year. Your gains might be even higher.



Software

THE PARALLEL UNIVERSE

Printed in USA 1017/SS Please Recycle