Computer Vision for the Masses

Up Your Game: Practical Optimizations for Game Programmers

Understanding the Instruction Pipeline: The Key to Adaptability
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Letter from the Editor</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>Computer Vision: Coming Soon to a Browser Near You</strong></td>
<td>5</td>
</tr>
<tr>
<td>by Henry A. Gabb, Senior Principal Engineer, Intel Corporation</td>
<td></td>
</tr>
<tr>
<td><strong>Computer Vision for the Masses</strong></td>
<td>5</td>
</tr>
<tr>
<td>Bringing Computer Vision to the Open Web Platform*</td>
<td></td>
</tr>
<tr>
<td><strong>Up Your Game</strong></td>
<td>23</td>
</tr>
<tr>
<td>How to Optimize Your Game Development—No Matter What Your Role—Using Intel® Graphics Performance Analyzers</td>
<td></td>
</tr>
<tr>
<td><strong>Harp-DAAL for High-Performance Big Data Computing</strong></td>
<td>31</td>
</tr>
<tr>
<td>The Key to Simultaneously Boosting Productivity and Performance</td>
<td></td>
</tr>
<tr>
<td><strong>Understanding the Instruction Pipeline</strong></td>
<td>41</td>
</tr>
<tr>
<td>The Key to Adaptability in Modern Application Programming</td>
<td></td>
</tr>
<tr>
<td><em><em>Parallel CFD with the HiFUN</em> Solver on the Intel® Xeon® Scalable Processor</em>*</td>
<td>51</td>
</tr>
<tr>
<td>Maximizing HPC Platforms for Fast Numerical Simulations</td>
<td></td>
</tr>
<tr>
<td><em><em>Improving VASP</em> Materials Simulation Performance</em>*</td>
<td>61</td>
</tr>
<tr>
<td>Using the Latest Intel® Software Development Tools to Make More Efficient Use of Hardware</td>
<td></td>
</tr>
</tbody>
</table>

For more complete information about compiler optimizations, see our Optimization Notice.
Computer Vision: Coming Soon to a Browser Near You

My first exposure to the field of computer vision was Gary Bradski's OpenCV* article in Dr. Dobb's Journal. Gary was working in Intel Labs at the time. I'd just joined Intel a few months earlier and was surprised to learn that we were working in that area. OpenCV has come a long way since its first release 17 years ago. Among other things, it’s open-source and supported by OpenCV.org, a non-profit foundation. In this issue’s feature article, Computer Vision for the Masses, a group of collaborators from the University of California, Irvine, and Intel Labs describe their work to translate many of OpenCV's key capabilities to JavaScript* to do high-performance computer vision within the Open Web Platform*. They present compelling examples of common computer vision applications like edge detection, face recognition, background subtraction, and image recognition using deep neural networks running within a browser.

Continuing with graphics and video processing, Up Your Game shows how to optimize game performance using Intel® Graphics Performance Analyzers. We present several use cases to show how different members of a game development team—like artists, designers, gameplay programmers, and game engine programmers—can take advantage of these analysis tools.

The Google File System* and MapReduce* ushered in the era of large-scale, distributed-memory data analytics back in 2003. Since then, many options have become available to data analytics and machine learning at scale. Apache Spark* is my current favorite, but I’ve experimented with Dask*, HPAT (the High-Performance Analytics Toolkit project from Intel Labs), and one or two others. In Harp-DAAL for High-Performance Big Data Computing, Professor Judy Qiu of Indiana University introduces a new framework that takes advantage of the highly-optimized Intel® Data Analytics Acceleration Library. This article describes the framework and walks you through an example, complete with Java* code, showing how to do k-means clustering using Harp-DAAL.
From high-level abstractions, we move to the opposite extreme. **Understanding the Instruction Pipeline** offers a gentle, conceptual overview of the instruction pipeline and why it matters for performance on modern processors. We close this issue with two articles on boosting the performance of compute-intensive scientific applications: **Parallel CFD with the HiFUN* Solver on the Intel® Xeon® Scalable Processor** and **Improving VASP* Materials Simulation Performance**. The former presents a parallel performance tuning case study using a real fluid dynamics application. And the latter shows how to generate a concise summary of application performance using **Intel® VTune™ Amplifier** – Application Performance Snapshot, and describes an interesting new preview feature in **Intel® MPI Library**.

Future issues of *The Parallel Universe* will bring you articles on doing deep learning within the Apache Spark analytics framework, Java performance enhancements, profiling and tuning I/O performance, new features in Intel® Software Development Tools, and much more. Be sure to [subscribe](#) so you won’t miss a thing.

**Henry A. Gabb**

April 2018
The Web is the world’s most universal compute platform and the foundation for the digital economy. Since its birth in early 1990s, Web capabilities have been increasing in both quantity and quality. But in spite of all the progress, computer vision isn’t yet mainstream on the Web. The reasons include:

- The lack of sufficient performance of JavaScript*, the standard language of the Web
- The lack of camera support in the standard Web APIs
- The lack of comprehensive computer vision libraries

*COMpuTER VIsIOn FOR THE Masses
These problems are about to get solved—resulting in the potential for a more immersive and perceptual Web with transformational effects including online shopping, education, and entertainment, among others.

Over the last decade, the tremendous improvements in JavaScript performance, plus the recent emergence of WebAssembly*, close the Web performance gap with native computing. And the HTML5 WebRTC* API has brought camera support to the Open Web Platform*. Even so, a comprehensive library of computer vision algorithms for the Web was still lacking. This article outlines a solution for the last piece of the problem by bringing OpenCV* to the Open Web Platform.

OpenCV is the most popular computer vision library, with a comprehensive set of vision functions and a large developer community. It’s implemented in C++ and, up until now, was not available in Web browsers without the help of unpopular native plugins.

We’ll show how to leverage OpenCV efficiency, completeness, API maturity, and its community’s collective knowledge to bring hundreds of OpenCV functions to the Open Web Platform. It’s provided in a format that’s easy for JavaScript engines to optimize and has an API that’s easy for Web programmers to adopt and use to develop applications. On top of that, we’ll show how to port OpenCV parallel implementations that target single instruction, multiple data (SIMD) units and multiple processor cores to equivalent Web primitives—providing the high performance required for real-time and interactive use cases.

**The Open Web Platform**

The Open Web Platform is the most universal computing platform, with billions of connected devices. Its popularity in online commerce, entertainment, science, and education has grown exponentially—as has the amount of multimedia content on the Web. Despite this, computer vision processing on Web browsers hasn’t been a common practice. The lack of client-side vision processing is due to several limitations:

- A lack of standard Web APIs to access and transfer multimedia content
- Inferior JavaScript performance
- Lack of a comprehensive computer vision library to develop apps

The approach we outline here, along with other recent developments on the Web front, will address those limitations and empower the Web with proper computer vision capabilities.

**Adding Camera Support and Plugin-Free Multimedia Delivery**

HTML5 introduced several Web APIs to capture, transfer, and present multimedia content in browsers without the need for third-party plugins. One of these, Web Real-Time Communication* (WebRTC*), allows acquisition and peer-to-peer transportation of multimedia content and video elements to display videos.
Recently, the immersive Web with access to virtual reality (VR) and augmented reality (AR) content has begun delivering new, engaging user experiences.

**Improved JavaScript* Performance**

JavaScript is the dominant language of the Web. Because it’s a scripting language with dynamic typing, its performance is inferior to that of native languages such as C++. Multimedia processing often involves complex algorithms and massive amounts of computation. With client-side technologies such as just-in-time (JIT) compilation, and with the introduction of WebAssembly* (WASM*), a portable, binary format for the Web, Web clients can reach a near-native performance with JavaScript and handle more demanding tasks.

**A Comprehensive Computer Vision Library**

Although there are several computer vision libraries developed in native languages such C++, they can’t be used in browsers without relying on unpopular browser extensions, which pose security and portability issues. There have been a few efforts to develop computer vision libraries in JavaScript, but these are limited to select categories of vision functions. Expanding those efforts with new algorithms, and optimizing the implementation, are challenging tasks. Previous work lacked either functionality, performance, or portability.

As an alternative approach, we take advantage of an existing comprehensive computer vision library developed in C++ (i.e., OpenCV) and make it work on the Web. This approach works great on the Web for several reasons:

- **It provides** an expansive set of functions with optimized implementation.
- **It performs** more efficiently than normal JavaScript implementations, and performance will further improve through parallelism.
- **Developers can access** a large collection of existing resources such as tutorials and examples.

**OpenCV.js***

OpenCV* is the de facto library for computer vision development. It’s an open-source library that started at Intel Labs back in 2000. OpenCV is very comprehensive and has been implemented as a set of modules (Figure 1). It offers a large number of primitive kernels and vision applications, ranging from image processing, object detection, and tracking to machine learning and deep neural networks (DNN). OpenCV provides efficient implementations for parallel hardware such as multicore processors with vector units. We translate many OpenCV functionalities into JavaScript and refer to it as OpenCV.js.
OpenCV implemented as a collection of modules

Table 1 categorizes and lists the functions currently included in OpenCV.js. It omits several OpenCV modules, for two reasons:

1. **Not all of OpenCV’s offerings are suitable for the Web.** For instance, the high-level GUI and I/O module (highgui)—which provides functions to access media devices such as cameras and graphical user interfaces—is platform-dependent and can’t be compiled to the Web. Those functions, however, have alternatives using HTML5 primitives, which are provided by a JavaScript module (utils.js). This works, for instance, to access files hosted on the Web and media devices through getUserMedia and to display graphics using HTML Canvas.*

2. **Some of the OpenCV functions are only used in certain application domains that aren’t common in typical Web applications.** For instance, the camera calibration module (calib3d) is often used in robotics. To reduce the size of the generated library for general use cases, based on OpenCV community feedback, we have identified the least commonly used functions from OpenCV and excluded them from the JavaScript version of the library.

### Table 1. OpenCV.js provided functionalities

<table>
<thead>
<tr>
<th>Module</th>
<th>Provided Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core components</td>
<td>Image manipulation and basic arithmetic</td>
</tr>
<tr>
<td>Image processing</td>
<td>Numerous functions to process and analyze images</td>
</tr>
<tr>
<td>Image processing</td>
<td>Video processing algorithms such as tracking, background segmentation, and optical flow</td>
</tr>
<tr>
<td>Object detection</td>
<td>HAAR*- and HOG*-based cascade classifiers</td>
</tr>
<tr>
<td>DNN</td>
<td>Inference of trained Caffe*, Torch*, or TensorFlow* models</td>
</tr>
<tr>
<td>GUI features</td>
<td>Helper functions to access frames from HTML Canvas*, video elements, and cameras</td>
</tr>
</tbody>
</table>

Since there are still many functions that might be useful for special use cases, we’ve provided a way to build the library with user-selected functions.
Translating OpenCV to JavaScript* and WebAssembly*

The emergence of Emscripten*, an LLVM-based source-to-source compiler developed by Mozilla, has made it possible to port many programs and libraries developed in C++ to the Web. Originally, Emscripten targets a typed subset of JavaScript called asm.js that, because of its simplicity, allows JavaScript engines to perform extra levels of optimization. In fact, it’s even possible to compile asm.js functions before execution.

While performance is impressive, parsing and compiling large JavaScript files could become a bottleneck, especially for mobile devices with weaker processors. This was one of the main motivations for development of WASM3. WASM is a portable size- and load-time-efficient binary format designed as a target for Web compilation. We used Emscripten to compile OpenCV source code into both asm.js and WASM. They offer the same functionality and can be used interchangeably.

During compilation with Emscripten, the C++ high-level language information such as class and function identifiers are replaced with mangled names. Since it’s almost impossible to develop programs through mangled names, we provide binding information of different OpenCV entities such as functions and classes and expose them to JavaScript. This enables the library to have a similar interface to normal OpenCV, with which many programmers are already familiar. Since OpenCV is large, and grows continuously through new contributions, continuously updating the port by hand is impractical. So we developed a semi-automated approach that takes care of the tedious parts of the translation process while allowing expert insights that can enable high-quality, efficient code production.

Figure 2 lists the steps involved in converting OpenCV C++ code to JavaScript. First, OpenCV source code is configured to disable components and implementations that are platform-specific, or are not optimized for the Web. Next, information about classes and functions that should be exported to JavaScript will be extracted from OpenCV source code. We use a white list of OpenCV classes and functions that should be included in the final JavaScript build. It’s possible to update the list before building to include or exclude OpenCV modules and/or functions. For efficiency, binding information for the OpenCV core module, which includes the OpenCV main data structure (i.e., cv::Mat), is manually provided. By using the binding information and function white list, we generate a glue code that maps JavaScript symbols to C++ symbols and compiles it with Emscripten along with the rest of the OpenCV library into JavaScript. The output of this process will be a JavaScript file (OpenCV.js) that serves as the library interface along with a WASM or asm.js file that implements OpenCV functions. utils.js, which includes GUI, I/O, and utility functions, is also linked with OpenCV.js.
Generating OpenCV.js

Using OpenCV.js in Web Applications

Let's explore how to use OpenCV.js to develop Web applications. Figure 3 shows an overview of OpenCV.js and its interaction with Web applications. Web applications will use the OpenCV.js API to access the functions as listed in Table 1. While the vision functions from OpenCV are compiled either into WASM or asm.js, we have developed a JavaScript module that provides GUI features and media capture. OpenCV.js utilizes standard Web APIs, such as Web workers and SIMD.js, to achieve high performance, and Canvas and WebRTC* to provide media and GUI capabilities.

The OpenCV.js API is inspired by the OpenCV C++ API and shares many similarities with it. For instance, C++ functions are exported to JavaScript with the same name and signature. Function overloading and default parameters are also supported in the JavaScript version. This makes migration to JavaScript easier for users who are already familiar with OpenCV development in C++.

Although OpenCV C++ classes are ported to JavaScript objects with the same member functions and properties, basic data types are different between the two versions. Table 2 shows equivalent JavaScript data types for basic C++ data types. JavaScript engines use Garbage Collector (GC) to manage program memory. However, GC activity has a negative impact on performance so OpenCV.js uses static memory management. Programmers are responsible for freeing OpenCV.js objects when they are no longer being used. Since manual memory management is tedious, we've used JavaScript types for primitive OpenCV types such as cv::Point. All std::vector are translated into JavaScript arrays, except for vectors of cv::Mat. This is particularly helpful, since by removing the vector, it will remove all the cv::Mat elements.
OpenCV.js components and its interaction with applications and web APIs

Table 2. Exported JavaScript types for basic C++ types

<table>
<thead>
<tr>
<th>C++ Type</th>
<th>JavaScript Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical types (e.g., int and float)</td>
<td>Number</td>
</tr>
<tr>
<td>bool</td>
<td>Boolean</td>
</tr>
<tr>
<td>enum</td>
<td>Constant</td>
</tr>
<tr>
<td>primitive structures (e.g., cv::Point)</td>
<td>Value object</td>
</tr>
<tr>
<td>std::vector (of cv::Mats)</td>
<td>cv.Vector</td>
</tr>
<tr>
<td>std::vector (of primitive types)</td>
<td>Array</td>
</tr>
<tr>
<td>std::string</td>
<td>String</td>
</tr>
</tbody>
</table>

We’ll present several examples to demonstrate various computer vision tasks using OpenCV.js. All of these examples work on top of a simple HTML Web page. We only present the logic part of the programs that deal with the OpenCV.js API.

Figure 4 shows how to apply the Canny algorithm to find the edges in an image. Input images will be loaded from an HTML Canvas. For this purpose, we’ve provided a helper function that takes the Canvas name and returns a color image. Since the Canny algorithm works on grayscale images, we have to do the extra step at line 3 to invoke `cv.cvtColor` to convert the input image from color to grayscale. Finally, after getting the result of Canny algorithm, we can render the image in the output canvas (line 6). Figure 5 shows a snapshot of this program running inside a browser.
let src = cv.imread ('canvasInput');
let dst = new cv.Mat ();
cv.cvtColor (src, src, cv.COLOR_RGB2GRAY, 0);
// You can try more different parameters
cv.Canny (src, dst, 50, 100, 3, false);

How to apply the Canny algorithm to find the edges in an image

Rendering the image in line 6

The next example (Figures 6 and 7) uses Haar cascades to detect faces in an image. Since this algorithm works on grayscale images, the input image is converted at line 3. At line 7, we initialize a cascade classifier and load it with a model for detecting faces. Other models trained to detect different objects, such as cats and dogs, can be used as well. At line 9, we invoke detectMultiscale, which searches in multiple copies of input images scaled with different sizes. When finished, it returns a list of rectangles for possible faces in the image. At line 10, we iterate over those rectangles and use the cv.rectangle function to highlight that part of the image.
Face detection using cascade classifiers

We've seen how to process single images in Web applications using OpenCV.js. Processing video boils down to processing a sequence of individual frames. The next example (Figures 8 and 9) demonstrates:

- **How to capture** frames from a video element
- **How to subtract** background from input video using the MOG2 algorithm
- **How to display** the processed frame on an HTML Canvas

The `cv.VideoCapture` object provided by `utils.js` enables WebRTC to access and manage camera resources. This example assumes the input video contains 30 frames per second. So, at every 1/30 of a second, it invokes the `processVideo` function. This function:

- **Reads** the next video frame (line 18)
- **Applies** background extraction function (line 19)
- **Displays** the output foreground mask (line 20)

Finally, at line 23 the next invocation of the function is scheduled.
```javascript
let src = cv.imread ('canvasInput');
let gray = new cv.Mat();
cv.cvtColor (src, gray, cv.COLOR_RGBA2GRAY, 0);
let faces = new cv.RectVector ();
let faceCascade = new cv.CascadeClassifier ();
// load pre-trained classifiers
faceCascade.load ('haarcascade_frontalface_default.xml');
// detect faces
faceCascade.detectMultiScale (gray, faces);
for (let i = 0; i < faces.size(); ++i) {
  let roiGray = gray.roi (faces.get(i));
  let roiSrc = src.roi (faces.get(i));
  let point1 = new cv.Point (faces.get(i).x, faces.get(i).y);
  let point2 = new cv.Point (faces.get(i).x + faces.get(i).width,
                              faces.get(i).y + faces.get(i).height);
  cv.rectangle (src, point1, point2, [255, 0, 0, 255], 3);
  roiGray.delete (); roiSrc.delete ();
}
```

7 Subtracting background for different frames of input video using the MOG2 method

8 Rendering the image
Capturing a video frame and subtracting the background

The last example (Figures 10 and 11) demonstrates using a pre-trained DNN in Web applications. While in this example we use a DNN to recognize objects, they can also be specialized to do other recognition tasks, such as background segmentation. At line 2, the program reads a Caffe framework model of GoogleNet*. Other formats, such as Torch and TensorFlow, are also supported. In the next step, at line 6, we convert the input image into a blob that fits the network. Then, at lines 9 and 10, we forward the blob along the network and find the highest probability class. Figure 10 shows a snapshot of the program categorizing a sample image. (You can see more examples of OpenCV.js usage at https://docs.opencv.org.)
Best class: #812 space shuttle Probability: 0.9910982251167297 Elapsed time: 1174 ms

Object detection example using GoogLeNet model
let src = cv.imread('canvasInput');
let net = cv.readNetFromCaffe('bvlc_googlenet.prototxt', 'bvlc_googlenet.caffemodel');
if (net.empty())
    throw "Failed to read net";
let inputBlob = cv.blobFromImage(src, 1, new cv.Size(224, 224), new cv Scalar (104, 117, 123));
net.setInput(inputBlob, data);
let prob = net.forward("prob");
let minMax = cv.minMaxLoc(prob);
console.log("Best class: ");
console.log(keywords[minMax.maxLoc.x] + " Probability: " + minMax.maxVal);
prob.delete(); inputBlob.delete();
net.delete(); src.delete();

Using a pre-trained DNN in Web applications

Performance Evaluation
OpenCV.js brings a lot of computer vision capabilities to the Web. To demonstrate their performance, we have selected a number of vision benchmarks including primitive kernels and more sophisticated vision applications including:

- Canny's algorithm for edge detection
- Finding faces using Haar cascades
- Finding people using a histogram of gradients

We used a Firefox* browser running on an Intel® Core™ i7-3770 processor with 8GB of RAM with Ubuntu* 16.04 for our setup and ran experiments over sequences of videos.

Figure 12 shows the average speedup of both simple JavaScript kernels and vision applications compared to their native equivalent that use an OpenCV scalar build (i.e., not using parallelism). As you can see, the JavaScript performance is competitive. While we found WASM and asm.js performance to be close, the WASM version of the library is significantly faster to initialize and is more compact. Its size is about 5.3 MB compared to the asm.js version, which is 10.4 MB.
Performance evaluation of primitive kernels and vision applications running on Firefox
Making it Even Faster

Computer vision is computationally demanding. A lot of computations need to be performed on a massive number of pixels. For instance, each iteration of the baseline Canny, face, and people benchmarks takes on average 7 ms, 345 ms, and 323 ms, respectively, to process a single 800 x 600 resolution image. While it’s fast to compute Canny, face, and people detection, they are still very expensive and cannot be used in real-time, interactive use-cases.

Fortunately, computer vision algorithms are inherently parallel, and good algorithm design and an optimized implementation can lead to significant speedups on parallel hardware. OpenCV comes with parallel implementations of algorithms for different architectures. We take advantage of two methods that target multicore processors and SIMD units to make the JavaScript version faster. We’ve skipped GPU implementations at the moment due to their complexity. The upcoming WebGPU* API can potentially be used to accelerate OpenCV on GPUs.

SIMD.js

SIMD.js⁴,⁵ is a new Web API to expose processor vector capabilities to the Web. It is based on a common subset of the Intel SSE2 and ARM NEON* instruction sets that runs efficiently on both architectures. They define vector instructions that operate on 128-bit-wide vector registers, which can hold four integers, four single-precision floating-point numbers, or 16 characters. Figure 13 shows how vector registers can add four integers with one CPU instruction.

![Scalar versus SIMD for the addition of four integers](image)
SIMD is proven to be very effective in improving performance, especially for multimedia, graphics, and scientific applications.\textsuperscript{5, 6} In fact, many OpenCV functions, such as core routines, are already implemented using vector intrinsics. We've adapted the work done by Peter Jensen, Ivan Jibaja, Ningxin Hu, Dan Gohman, and John Mc-Cutchan\textsuperscript{7} to translate OpenCV vectorized implementations using SSE2 intrinsics into JavaScript with SIMD.js instructions. Inclusion of the SIMD.js implementation will not affect the library interface. Figure 14 shows the speedups that are obtained by SIMD.js on selected kernels and applications running on Firefox. Up to 8x speedup is obtained for primitive kernels. As expected, the speedup is higher for smaller data types. There are fewer vectorization opportunities in complex functions such as Canny, face, and people detection. Currently, SIMD.js can only be used in the asm.js context and is supported by the Firefox and Microsoft Edge\textsuperscript{*} browsers. SIMD in WebAssembly is currently planned to have the same specification as SIMD.js. Hence, similar performance numbers are expected.

![Performance improvement using SIMD.js](image)

**Figure 14** Performance improvement using SIMD.js

### Multithreading using Web Workers

JavaScript programs use Web workers for parallel processing of heavy computing tasks. Web workers communicate by message passing, which could incur significant cost, especially when passing large messages such as images. SharedArrayBuffer\textsuperscript{8} has recently been proposed as a storage system that can be shared between Web workers. They can use it to implement a shared-memory parallel programming model. OpenCV uses its parallel_for\_ framework to implement multithreaded versions of vision functions. The parallel_for\_ framework can target multiple multithreading models including POSIX\* threads (Pthreads). With recent Emscripten\* developments, it’s possible to translate the Pthreads
API into equivalent JavaScript using Web workers with shared array buffers. OpenCV.js with multithreading support will use a pool of Web workers and allocate a worker when a new thread is being spawned. Also, it exposes OpenCV APIs to dynamically adjust the concurrency such as changing number of concurrent threads (i.e., cv.SetNumThreads) to JavaScript.

To study the performance impact of using multiple Web workers, we measured the performance of three application benchmarks that did not gain significantly from SIMD vectorization using different numbers of workers (up to 8). The OpenCV load balancing algorithm divides the workload evenly among threads. As shown in Figure 15, on a processor with eight logical cores, we obtained a 3x to 4x speedup. Note that a similar trend is observed on native Pthreads implementations of the same functions.

![Graph showing speedup achieved using multiple Web workers](image)

**Figure 15** Speedup achieved using multiple Web workers

**Computer Vision for the Masses**

This work brings years of OpenCV development in computer vision processing to the Web with high efficiency. It provides a collection of carefully selected functions including image processing, object detection, video analysis, features extraction, and DNNs, among others. The results of our experiments show the framework’s high capability. Thanks to JavaScript portability, for the first time, a large collection of vision functions can be used not only on Web browsers but also on embedded devices and desktop applications. For instance, it provides computer vision for Node.js-based Internet of Things (IOT) devices and JavaScript desktop development frameworks such
as Electron*. Combined with the recent developments in Web platform, it's a more efficient way to make real new Web applications and experiences like emerging virtual and augmented reality. We've also provided a large collection of computer vision tutorials using OpenCV.js that we hope will be a good asset for education and research purposes.

The authors are grateful to Congxiang Pan, Gang Song, and Wenyao Gan for their contributions through the Google Summer of Code program, and would also like to thank the OpenCV founder, Dr. Gary Bradski, its chief architect, Vadim Pisarevsky, and Alex Alekin for their support and helpful feedback.

Learn More

We've developed extensive online resources to help developers and researchers learn more about OpenCV.js and computer vision in general:

- OpenCV.js documentation and tutorials
- OpenCV.js demos

OpenCV.js can also be used in Node.js-based environments. It's published on the Node Package Manager (NPM) at https://www.npmjs.com/package/opencv.js.

References

Performance optimization can mean the difference between a good game and a great game. And providing that smooth playing experience can be a daunting task. But Intel® Graphics Performance Analyzers (Intel® GPA) can help streamline the process, making it possible for everyone on your development team to lend a helping hand. After reading this article, you’ll be able to pinpoint exactly where you fit into the optimization process and see how the Intel GPA tool suite can help you reach your optimization goals, like higher fidelity or better framerate.

Where Do You Fit?
There are three main types of game developers:

1. Artists/designers
2. Gameplay programmers
3. Game engine programmers

For more complete information about compiler optimizations, see our Optimization Notice.
Each of these groups plays a vital role. And each interacts with the optimization process in a different way. Let's dive into some use cases to better understand the differences.

**Artists/Designers**
We'll begin with the artists who create the assets associated with characters and the environment within the game—the scenery, objects, surface textures, vehicles, etc.

To create the best viewing experience, artists typically develop these elements with the highest fidelity in mind (4K textures and more complex geometry), but they can be scaled back during optimization. Artists work to create textures and assets that allow for smooth gameplay on the target hardware. They also work hand-in-hand with gameplay programmers to execute design ideas relating to lighting, movement, and overall design execution.

When you're retargeting for a new minimum specification or optimizing due to inadequate framerate, you can run into issues like:

- **Texture bandwidth**: Textures that are too big or too high-resolution
- **Geometric complexity**: Polygons that are very small

Optimization tools like Intel GPA allow artists to self-check their work, ensuring that any texture or asset addition doesn't drop the framerate below your target or cause additional gameplay issues.

Using Intel GPA, artists can determine their overall impact on any given scene by capturing a frame with the Graphics Monitor and profiling that frame in Frame Analyzer. Frame Analyzer lets artists see the effect textures and assets make by extracting pertinent metrics associated with asset geometry—like viewing total rendered primitives—as well as selecting through the different mipmap (MIP) levels associated with a texture. Using the interactive experiments like “2*2 Texture” (Figure 1), artists can determine whether a texture or object has created a bottleneck that degrades game performance and could benefit from MIP level reduction (Figure 2) using a lower resolution of a texture, or using levels of detail (LODs), meaning lowering the level of detail of an object as it moves further away from the camera. By using Intel GPA as a validator for asset creation and performance impact, artists are better able to cross-reference gameplay with the asset additions—and are more equipped to maintain high framrates with high fidelity.
Using the 2*2 texture experiment from the bottom right corner (2) produces a 5% performance increase for the entire frame (1). This shows it might be beneficial to view the used textures for the selected draw call and consider using a lower MIP level for the textures, if they don't drastically impact the final scene quality.

Gameplay Programmers

Gameplay programmers combine the game-agnostic components created by the game engine programmers and the assets made by the artists to create the best gameplay experience possible. Their main optimization focus is on game functionality—user interface, character movement, controls, audio, etc.—while executing the artists' designs.
During optimization, these programmers focus their attention on efficient code performance to create a good base layer for assets. The more efficient the back-end code is, the better. When these programmers use Intel GPA, the main benefit they’ll see is in live editing with playback, CPU/GPU profiling, and experiments (although this list of benefits is not exhaustive by any means).

Intel GPA offers multiple ways to determine where and when to optimize code or to optimize the way in which objects are rendered (the number of objects and batching). One common area a gameplay programmer can optimize is the shaders—depending on how small the studio is and whether there are any dedicated game engine programmers on the team. Invoking the “Simple Pixel Shader” or “1*1 Scissor Rect” (Figure 3) within the Frame Analyzer, programmers can narrow down a bottleneck to either the pixel shader or the vertex shader. With live playback and editing, they can then edit the shader within the tool or import an optimized shader and see the outcome of those changes (Figure 4). Quickly identifying the dependencies of a costly shader is also incredibly easy with the Resource History tab, which lists all events that use the selected resource. Other common areas that gameplay programmers may pay attention to include:

To see any texture (both from input and output), select it from the resource viewer (1). MIP levels will appear in the left-hand side of the resource viewer, if there are different MIPs associated with that texture. From here, you can see how the texture will affect the scene and final render target.
By enabling the simple pixel shader experiment in the bottom left corner of Frame Analyzer, we see an increase of 6% (1) in overall frame performance. By identifying that a shader has room to be optimized, we can then go to the shader resource view to either import or modify a shader (Figure 2).

- **How particle effects** are rendered
- **The lighting** (reflection, number of lights, etc.)
- **How objects are placed** within a scene
- **How draw calls** are batched (dynamic batching versus static batching)

Another quick optimization is to look through the API log to determine if there are any duplicate draw calls, or in the bar chart to determine if there are duplicate passes like an additional depth pass. The tool gameplay programmers will initially use during optimization, however, is the Trace Analyzer tool. Capturing a trace during live analysis opens the door to profiling CPU and GPU utilization. Once open, a trace shows the CPU and GPU queue, frame timings, V-Sync, and process duration. Trace Analyzer is also customizable, allowing a user to change which metrics are collected during capture. By viewing where waits happen, how long an item is in the queue, and how workloads are distributed on the CPU and GPU, gameplay programmers can determine where optimizations are needed (CPU processes versus GPU processes) and where to focus their tuning efforts.
Selecting the shader view from the execution portion of the resource menu (2) allows for live editing and importing (3) of locally saved shaders. After editing a shader, Frame Analyzer will playback the frame using the new shader and display full frame performance impact in the top right corner of the window (1).

Here we’ve initiated the overdraw (dropdown 2) view of a render target. The lighter colored the area is, the more overdraw there is. If the objects creating the overdraw come from draw calls later in the scene, then Z-test is not implemented. To determine if a call comes later in the scene, turn on Scrub Mode (dropdown 1) and see if these sections disappear. Scrub Mode only shows the elements that have been drawn up to the currently selected draw call. Any elements that disappear when Scrub Mode is on will come from draw calls later in the scene.
Game Engine Programmers

Finally, we come to the game engine programmers, who create a tool that enables the gameplay programmers to complete their job. The game engine programmers create game-agnostic components that are the foundation on which the gameplay programmers create the game. This can include:

- **Implementing** how animations will happen
- **The physics** behind gameplay
- **Collision** detection
- **How pixels** will be rendered to the screen

As we touched on in the gameplay programmers section, without dedicated game engine programmers, the gameplay programmers would optimize the shaders. However, with the addition of game engine programmers, shader optimization (vertex, pixel, compute, etc.) falls to the game engine programmers. Using the overdraw view in Frame Analyzer, programmers can identify any Z-test- or depth-test-related optimizations. By entering the overdraw view (Figure 5) for a render target, a programmer can identify whether or not the engine is currently rejecting pixels that will be hidden by closer objects within the scene. If there’s significant overdraw in a scene, it might be useful to further investigate and determine if the depth test is being used within the engine. Adding Z-rejection could limit the number of times an unseen pixel is fully rendered, reducing the total number of draw calls for any given scene. Game engine programmers would also touch any other aspects of the rendering pipeline during optimization, as well as using Trace Analyzer to inspect CPU/GPU utilization and determine any optimizations needed on the CPU, such as optimizing the AI algorithm and physics computations. Game engine programmers can use every aspect of Intel GPA from the GPU pipeline view, which identifies GPU pipeline bottlenecks, to the buffer views.

To learn more on practical optimizations for game engine engineers using Intel GPA, go to [Practical Game Performance Analysis Using Intel® Graphics Performance Analyzers](http://software.intel.com/gpa) (though please note, this article is written using the old UI of Intel GPA), or watch [this video](#).

From Good Game to Great Game

Intel GPA isn’t for just one type of game developer—it offers features for every individual in the game developer team. Hopefully, this brief overview of roles and common pain points associated with the optimization process has made reaching a target framerate more achievable. We’ve only touched the surface in explaining the benefits Intel GPA can provide for each stage of the optimization process, providing some quick items to check for during optimization. To find more ways to optimize your application using Intel GPA, visit [software.intel.com/gpa](http://software.intel.com/gpa), where you’ll find a Getting Started Guide and a comprehensive help document for each supported operating system.
Get Intel’s powerful performance libraries to optimize your code and cut your development time. They’re free as part of Intel’s mission to support innovation and impressive performance on Intel® architecture.

FREE DOWNLOAD >
Large-scale data analytics is revolutionizing many business and scientific domains. And easy-to-use, scalable parallel techniques are vital to process big data and gain meaningful insights. This article introduces a novel high-performance computing (HPC)-cloud convergence framework named Harp-DAAL and demonstrates how the combination of big data (Hadoop*) and HPC techniques can simultaneously boost productivity and performance.

Harp* is a distributed Hadoop-based framework that orchestrates node synchronization¹. Harp uses Intel® Data Analytics Acceleration Library (Intel® DAAL)² for its highly-optimized kernels on Intel® Xeon® and Xeon Phi™ processor architectures. This way, the high-level API of big data tools can be combined with intra-node, fine-grained parallelism that’s optimized for HPC platforms.
We illustrate this framework in detail with K-means clustering, a compute-bound algorithm used in image clustering. We also show the broad applicability of Harp-DAAL by discussing the performance of three other big data algorithms:

1. Subgraph Counting by color coding
2. Matrix Factorization
3. Latent Dirichlet Allocation

These algorithms share characteristics such as load imbalance, irregular structure, and communication issues that create performance challenges.

The categories in Figure 1 illustrate a classification of data-intensive computation into five models that map into five distinct system architectures. It starts with Sequential, followed by centralized batch architectures corresponding exactly to the three forms of MapReduce: Map-Only, MapReduce, and Iterative MapReduce. Category five is the classic Message Passing Interface (MPI) model.

Harp brings Hadoop users the benefits of supporting all five classes of data-intensive computation, from naturally parallel to machine learning and simulation. It expands the applicability of Hadoop (with a Harp plugin) to more classes of big data applications, especially complex data analytics such as machine learning and graph analysis. The design consists of a modular software stack with native kernels (with Intel DAAL) to effectively utilize scale-up servers for machine learning and data analytics applications. Harp-DAAL shows how simulations and big data can use common programming environments with a runtime based on a rich set of collective operations and libraries.
Interfacing Harp and Intel DAAL

Intel DAAL provides a native C/C++ API but also provides interfaces to higher-level programming languages such as Java* and Python*. Harp is written in Java and extended from the Hadoop ecosystem, so Java was the natural choice to interface Harp and Intel DAAL.

In Harp-DAAL, data is stored in a hierarchical data structure called Harp-Table, which consists of tagged partitions. Each partition contains a partition ID (metadata) and a user-defined serializable Java object such as a primitive array. When doing communication, data is transferred among distributed cluster nodes via Harp collective communication operations. When doing local computation, data moves from Harp-Table (JVM heap memory) to Intel DAAL native kernels (off-JVM heap memory). Copying data between a Java object and C/C++ allocated memory space is unavoidable. Figure 2 illustrates two approaches to this data copy:

1. **Direct Bulk Copy**: If an Intel DAAL kernel allocates a continuous memory space for dense problems, Harp-DAAL will launch a bulk copy operation between a Harp-Table and a native memory address.

2. **Multithreading Irregular Copy**: If an Intel DAAL kernel uses an irregular and sparse data structure—which means the data will be stored in non-consecutive memory segments—Harp-DAAL performs a second copy operation using Java/OpenMP threads, where the threads transfer data segments concurrently.
Applying Harp-DAAL to Data Analytics

K-means is a widely-used and relatively simple clustering algorithm that provides a clear example of how to use Harp-DAAL. K-means uses cluster centers to model data and converges quickly via iterative refinement. K-means clustering was performed on a large image dataset from Flickr*, which includes 100 million images, each with 4,096 dimensional deep features extracted using a deep convolutional neural network model trained on ImageNet*. Data preprocessing includes format transformation and dimensionality reduction from 4,096 to 128 using Principal Component Analysis (PCA). (The Harp-DAAL tutorial contains a description of the data preparation.)*

Harp-DAAL provides modular Java functions for developers to customize the K-means algorithm as well as tuning parameters for end users. The programming model consists of map functions linked by collectives. The K-means example takes seven steps.

Step 1: Load Training Data (Feature Vectors) and Model Data (Cluster Centers)

Use this function to load training data from the Hadoop Distributed File System (HDFS):

```java
// create a pointArray
List<double[]> pointArrays = LoadTrainingData();
```

Similarly, create a Harp table object `cenTable` and load centers from HDFS. Since centers are requested by all the mappers, the master mapper will load and broadcast them to all other mappers. Different center initialization methods can be supported in this fashion:

```java
// create a table to hold cluster centers
Table<DoubleArray> cenTable = new Table<>(0, new DoubleArrPlus());

if (this.isMaster()) {
    createCenTable(cenTable);
    loadCentroids(cenTable);
}
// Bcast centers to other mappers
bcastCentroids(cenTable, this.getMasterID());
```
Step 2: Convert Training Data from Harp to Intel DAAL
The training data loaded from HDFS is stored in the Java heap memory. To invoke the Intel DAAL kernel, this step converts the data to an Intel DAAL NumericTable, which includes allocating the native memory for the NumericTable and copying data from pointArrays to trainingdata_daal.

```java
// convert training data from Harp to DAAL
NumericTable trainingdata_daal = convertTrainData(pointArrays);
```

Step 3: Create and Set Up an Intel DAAL K-means Kernel
Intel DAAL provides Java APIs to invoke native kernels for K-means on each node. It is called with:

- A specification of the input training data object
- The number of centers
- The number of threads to be used by the thread scheduler (Intel® Threading Building Blocks and Intel® Math Kernel Library)

```java
// create a DAAL K-means kernel object
DistributedStep1Local kmeansLocal = new DistributedStep1Local(daal_Context, Double.class, Method.defaultDense, this.numCentroids);
// set up input training data
kmeansLocal.input.set(InputId.data, trainingdata_daal);
// specify the number of threads used by DAAL kernel
Environment.setNumberOfThreads(numThreads);
// create cenTable on DAAL side
NumericTable cenTable_daal = createCenTableDAAL();
```

Step 4: Convert Center Format from Harp to Intel DAAL
The centers are stored in the Harp table cenTable for inter-process (mapper) communication. The centers are converted to Intel DAAL format at each iteration.

```java
// Convert center format from Harp to DAAL
convertCenTableHarpToDAAL(cenTable, cenTable_daal);
```
Step 5: Local Computation by Intel DAAL Kernel

Call Intel DAAL K-means kernels of local computation at each iteration.

```c
// specify cluster centers to DAAL kernel
kmeansLocal.input.set(InputId.inputCentroids, cenTable_daal);
// first step of local computation by using DAAL kernels to get a partial result
PartialResult pres = kmeansLocal.compute();
```

Step 6: Inter-Mapper Communication

Harp-DAAL K-means uses an AllReduce computation model where each mapper keeps a local copy of the whole model data (cluster centers). However, Harp provides different communication operations to synchronize model data among mappers:

- regroup & allgather (default)
- allreduce
- broadcast & reduce
- push & pull

In a regroup & allgather operation, it first combines the same center from different mappers and redistributes them to mappers by a specified order. After averaging the centers, an allgather operation makes every mapper get a complete copy of the averaged centers.

```c
comm_regroup_allgather(cenTable, pres);
```

In an allreduce operation, the centers are reduced and copied to every mapper. Then, on each mapper, an average operation is applied to the centers:

```c
comm_allreduce(cenTable, pres);
```

At the end of each iteration, call printTable to check the clustering result:

```c
// for iteration i, print the first ten centers of the first ten dimensions
printTable(cenTable, 10, 10, i);
```
Step 7: Release Memory and Store Cluster Centers

After all of the iterations, release the memory allocated for Intel DAAL and for the Harp table object. The center values are stored on HDFS as the output:

```java
// free memory and record time
cenTable_daal.freeDataMemory();
trainingdata_daal.freeDataMemory();
// Write out the cluster centers
if (this.isMaster()) {
    KMUtil.storeCentroids(this.conf, this.cenDir,
    cenTable, this.cenVecSize, "output");
}
cenTable.release();
```

Performance Results

The performance for Harp-DAAL is illustrated by the results for four applications with different algorithmic features:

1. **K-means**: A dense clustering algorithm with regular memory access
2. **MF-SGD (Matrix Factorization for Stochastic Gradient Descent)**: A dense recommendation algorithm with irregular memory access and large model data
3. **Subgraph Counting**: A sparse graph algorithm with irregular memory access
4. **Latent Dirichlet Allocation (LDA)**: A sparse, topic modeling algorithm with large model data and irregular memory access

The testbed has two clusters:

1. One with **Intel Xeon E5 2670 processors** and the InfiniBand Interconnect*
2. One with **Intel Xeon Phi 7250 processors** and the Intel® Omni-Path Interconnect

In **Figure 3**, Harp-DAAL achieves around a 30x speedup over Spark* for K-means on 30 nodes using its highly vectorized kernels from Intel Math Kernel Library, which is part of Intel DAAL. MF-SGD was run on up to 30 nodes and achieved a 3x speedup over NOMAD*, a state-of-the-art MPI C/C++ solution. The benefits come from Harp’s rotation collective operation that accelerates the communication of the big model data in the recommender system.
3 Performance comparison on three different important machine learning algorithms: K-means, MF-SGD, and Subgraph counting

Harp-DAAL subgraph counting on 16 Intel Xeon E5 processor-based nodes has a 1.5x to 4x speedup over MPI-Fascia for large subtemplates with billion-edged Twitter graph data. The performance improvement comes from node-level pipeline overlapping of computation and communication. Single-node thread concurrency improved by neighbor list partitioning of the graph vertex.

Figure 4 shows that Harp LDA achieves better convergence and speedup over other state-of-the-art MPI implementations such as LightLDA* and NomadLDA*5. This advantage comes from two optimizations for parallel efficiency:

1. Harp adopts the rotation computation model for inter-node communication of the latest model, and at the same time utilizes timer control to reduce the overhead of synchronization.
2. At the intra-node level, a dynamic scheduling mechanism is developed to mitigate load imbalance.

4 Performance of various LDA implementations on the Clueweb dataset (30 billion tokens, 5,000 topics)
The current Harp-DAAL system provides 13 distributed data analytics and machine learning algorithms leveraging the local computation kernels like K-means from the Intel DAAL 2018 release. In addition, Harp-DAAL is developing its own data-intensive kernels. This includes the large-scale subgraph counting algorithm given above, which can process a social network Twitter graph with billions of edges and subtemplates of 10 vertices in 15 minutes. The Harp-DAAL framework and machine learning algorithms are publicly accessible\(^5\) so you can download the software, explore the tutorials, and apply Harp-DAAL to other data-intensive applications.

References

2. Intel DAAL, [https://github.com/intel/daal](https://github.com/intel/daal)
Developers worldwide have upped the ante for application performance, scalability, and portability with Intel® Software Development Tools. And they’re sharing their stories to help you do the same.

EXPLORE >

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
Understanding the instruction pipeline, on at least a basic level, is as critical to achieving high efficiency in modern application programming as understanding color theory is to painting. It's a fundamental and ubiquitous concept. While sources vary on exact dates and definitions, instruction pipelining as we know it started gaining popularity at some point in the 1970s or 1980s and is omnipresent in modern machines.

Processing an instruction isn't instantaneous. There are several steps involved. While the exact details of implementation vary from machine to machine, conceptually it boils down to five main steps:
1. Fetching an instruction
2. Decoding it
3. Executing it
4. Accessing memory
5. Writing back the results

Without pipelining, each instruction is processed from start to finish before moving on to the next. If we assume that each of the five steps takes one cycle, then it would take 15 cycles to process three instructions (Figure 1).

Because each step is handled by a different section of hardware, modern processors improve efficiency by pipelining the instructions, allowing the various hardware sections to each process a different instruction simultaneously. For instance, in cycle 3 of Figure 2, the processor is fetching instruction C, decoding instruction B, and executing instruction A. All three instructions are completed by the end of cycle seven—eight cycles sooner than if they’d been processed sequentially.

We can compare this to washing a second load of laundry while your first load is in the dryer. While processing an instruction certainly involves more steps than doing laundry, we can still divide it into two sections:

- **The Front End**, the part of the CPU that fetches and decodes instructions
- **The Back End**, the part that executes and retires instructions
Of course, Figure 2 is an oversimplification of instruction pipelining. In reality, the number of steps in the pipeline varies among implementations, with each of the steps used in the example often being split into multiple substeps. However, this doesn’t affect conceptual understanding, so we’ll continue to use the simplified five-step model. Also, this simplified model doesn’t take into account superscalar design, which results in multiple pipelines per processor core because it duplicates functional units such as arithmetic logic units (ALUs) and fetches multiple instructions at once to keep the extra units busy.

The number of pipelines available is called the width. Figure 3 represents a two-wide design that fetches instructions A and B on the first cycle, and instructions C and D on the second cycle. The width is (theoretically) defined in terms of how many instructions can be issued each cycle, but this is somewhat complicated by the way pipelining is done with CISC designs such as the ever-popular x86.

Pipelining works best with RISC designs, those with a small number of simpler instructions that run quickly. The varying complexity and running times for more elaborate instructions like those found in x86 can make pipelining difficult for multiple reasons:
The solution to this problem was to break down these complex operations into smaller micro-operations, or \( \mu \)ops. For convenience, the \( \mu \) is often replaced with \( u \)—thus, the uop. The x86 instructions are therefore fetched and decoded, converted into uops, and then dispatched from a buffer to be executed and, ultimately, retired. This disconnect between x86 instructions being fetched and uops being dispatched makes it hard to define the width of a processor using this methodology, and this difficulty is exacerbated by the fact that pairs of uops can sometimes be fused together.

The difficulty of precisely defining the processor width in this scenario makes an abstraction appealing. Regardless of semantics or underlying hardware, there's ultimately a fixed number of uops that can be issued from the Front End per cycle, and a fixed number of uops that can be retired from the Back End per cycle. This is the number of pipeline slots available and, as a general rule of thumb, the magic number is usually four on modern Intel\textsuperscript{®} processors.

- **Slow instructions** can bog down the pipeline.
- **Complicated instructions** may be more likely to stall on data dependencies.
The concept of the pipeline slot is useful for application optimization because each slot can be classified into one of four categories on any given cycle based on what happens to the uop it contains (Figure 4). Each pipeline slot category is expected to fall within a particular percentage range for a well-tuned application of a given type (e.g., client, desktop, server, database, scientific). A tool like Intel® VTune™ Amplifier can help to measure the percentage of pipeline slots in an application that fall into each category, which can be compared to the expected ranges. If a category other than Retiring exceeds the expected range for the appropriate application type, it indicates the presence and nature of a performance bottleneck.

![Pipeline slot categorization flowchart](image)

Much has already been written on the technique of using these measurements for performance optimization, including the Intel VTune Amplifier tuning guides, and these methods are outside the scope of this article, so we won’t cover them here. (See the suggested readings at the end of this article for additional tuning advice.) Instead, we’ll focus on understanding what’s going on within the pipeline in these situations. For the sake of simplicity, our diagrams will have only a single pipeline.

We’ve already discussed the Retiring category. It represents normal functionality of the pipeline, with no stalls or interruptions. The Back-End-Bound and Front-End-Bound categories, on the other hand, both represent situations where instructions weren’t able to cross from the Front End to the Back End due to a stall. The stalls that cause Back-End-Bound and Front-End-Bound slots can have many root causes, including everything from cache misses to overly high demand for a particular type of execution unit. But the effects ultimately boil down to uops not leaving their current stages on schedule.
A Front-End-Bound slot occurs when an instruction fails to move from the Front End into the Back End despite the Back End being able to accommodate it. In Figure 5, instruction B takes an extra cycle to finish decoding, and remains in that stage on cycle 4 instead of passing into the Back End. This creates an empty space that propagates down the pipeline—known as a pipeline bubble—marked here with an exclamation point.

![Example of a Front-End-Bound slot](image)

A Back-End-Bound slot occurs when the Back End cannot take incoming uops (regardless of whether the Front End is capable of actually supplying them) (Figure 6). In this example, instruction B takes an extra cycle to execute and, because it is still occupying the Execute stage on cycle 5, instruction C can’t move into the Back End. This also results in a pipeline bubble.

![Example of a Back-End-Bound slot](image)
Note that the delay doesn’t have to occur in the Decode or Execute stages. In Figure 5, if B had taken an extra cycle to fetch, no instruction would have passed into the Decode stage on cycle 3, creating a bubble, so there would be no instruction to pass into the Back End on cycle 4. Likewise, in Figure 6, if instruction A had taken an extra cycle in the Memory stage, then B would have been incapable of moving out of the Execute stage on cycle 5, whether it was ready to or not. Therefore, it would remain where it was, blocking C from proceeding into the Back End.

The final category is Bad Speculation. This occurs whenever partially processed uops are cancelled before completion. The most common reason uops are cancelled is due to branch misprediction, though there are other causes (e.g., self-modifying code). A branch instruction must be processed to a certain point before it’s known whether the branch will be taken or not. Once again, the implementation details vary, but are conceptually similar. For the sake of demonstration, we’ll assume that we’ll know whether to take path X or path Y when the branch instruction reaches the end of the Execute stage (Figure 7). Branches are so common that it’s infeasible to incur a performance penalty every time one is encountered by waiting until it finishes executing to start loading the next instruction. Instead, elaborate algorithms predict which path the branch will take.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Br</td>
<td>X1</td>
<td>X2</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td>Br</td>
<td>X1</td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td>Br</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instructions from path X being loaded into the pipeline before the branch is resolved
The Parallel Universe

From here, there are two possible outcomes:

1. **The branch prediction is correct** (Figure 8) and things proceed as normal.
2. **The branch is mispredicted** (Figure 9), so the incorrect instructions are discarded, leaving bubbles in their place, and the correct instructions begin entering the pipeline.

The performance penalty is effectively the same as if the pipeline had simply waited for the execution of the branch to resolve before beginning to load instructions, but only occurs when the prediction algorithm is wrong rather than every time a branch is encountered. Because of this, there’s a constant effort to improve prediction algorithms.
Anyone with a performance analyzer can access Bad Speculation, Front-End-Bound, and Back-End-Bound slot counts for an application. But without understanding where those numbers come from or what they mean, they’re useful for little more than blindly following instructions from a guide, utterly dependent on the author’s recommendations. Understanding is the key to adaptability and, in the fluid world of software, it’s crucial to be able to respond to the unique needs of your own application—because some day, you’ll encounter a scenario that hasn’t been written about.

Learn More

- “Tuning Applications Using a Top-Down Microarchitecture Analysis Method.” Intel Developer Zone. Intel.
WHERE BRIGHT MINDS UNITE

Find, follow, and connect with world-class developers—plus, share your own projects and collaborate with experts in your areas of interest. It's all waiting for you when you join Intel® Developer Mesh.

JOIN >
Computational fluid dynamics (CFD) is a branch of science that deals with numerical solutions for equations governing fluid flow—with the help of high-speed computers. And, because they’re based on a mountain of data, it’s essential for CFD solutions to get every last bit of performance out of today’s high-performance computing (HPC) hardware platforms.

CFD uses the Navier-Stokes equations—non-linear, partial differential equations describing mass, momentum, and energy conservation for a fluid flow. In CFD, discretization is a technique to convert the Navier-Stokes equations into a set of algebraic equations. Due to the geometric complexity and complicated
flow physics associated with an industrial application, the typical size of the algebraic system varies from a few million to over a billion equations. That means realistic numerical simulations need to be carried out on large-scale HPC platforms to obtain design data in a timeframe short enough to impact the design cycle.

In this article, we'll explore the HiFUN® solver, proprietary software from S&I Engineering Solutions (SandI) Pvt. Ltd., as an example of a CFD application that can take full advantage of the architecture of massively parallel supercomputing platforms.

Reaching Scalable Performance

Three factors affect the performance of HPC applications:

1. Single-process performance
2. Load balance
3. Algorithmic scalability

To reach load balance, the discretized computational domain (also called the workload, grid, or mesh) should be divided so that the computational work assigned to every processor core is approximately the same. However, load balancing shouldn’t lead to excessive data communication among the distributed processors. These two requirements often conflict, so it’s necessary to strike a balance between them. Domain decomposition using METIS is a common way to achieve load balance in CFD simulations. Algorithmic scaling is another critical performance factor in parallel applications. Ideally, the performance of a parallel solver shouldn’t degrade as the number of processors increases.

The third factor limiting the parallel performance of a CFD application like the HiFUN solver is the gap between faster processor speed and slower memory access speed—which can lead to a decline in single-process performance. To overcome this problem, many computers use a memory hierarchy, where a small portion of the data required for immediate computations resides in memory, which has the fastest access, called cache. This means it’s important to fully utilize the data brought into the cache memory—which requires good data layout in the memory to ensure spatial locality of the grid data. Though the issue of single-process performance is partly addressed at the application level with ordering algorithms like Cuthill McKee, it’s still a chore to completely exploit process performance given the varied nature of computational operations involved in a CFD solver. In this context, the evolution of Intel® processor technology is boosting the single-process performance of CFD applications in a big way. The latest generation of Intel® Xeon® processors provides large-cache, high-memory bandwidth with an increased number of data channels between the processor and memory, as well as higher memory speed. For a CFD application, improved single-process performance naturally leads to improved single-node performance.
To explore how the advancements in processor technology translate into improved software performance, we evaluated the performance improvement of HiFUN on the Intel® Xeon® Scalable processor compared to its predecessors, assessing both the single-node and multi-node performance of HiFUN.

**HiFUN CFD Solver**

HiFUN is a state-of-the-art, general-purpose CFD solver that’s robust, fast, and accurate, providing aerodynamic design data in an attractive turnaround time. Its usefulness stems from its ability to handle the complex geometries and complicated flow physics in a typical industrial environment. Using unstructured data capable of handling arbitrary polyhedral volumes gives HiFUN the ability to simulate complex geometries with relative ease. Plus, the use of a matrix-free implicit procedure results in rapid convergence to steady state—making the solver both efficient and robust.

The accuracy of HiFUN has been amply demonstrated in various international CFD code evaluation exercises such as the AIAA Drag Prediction Workshops and AIAA High Lift Prediction Workshops. HiFUN has been successfully used in simulations for a wide range of flow problems, from low subsonic speeds to hypersonic speeds.

**Evaluating Parallel Performance**

To evaluate the parallel performance of the HiFUN solver, we need to consider these metrics:

1. **Ideal speedup**: The ratio of the number of cores used in a given run to a reference number of cores (i.e., the smallest number of cores used in the study).
2. **Actual speedup**: The ratio of time per iteration when the reference number of cores are used for a given computation to the time per iteration for a given number of cores.
3. **Parallel efficiency**: The ratio of actual speedup to ideal speedup.
4. **Machine performance parameter (MPP)**: The ratio of product of time per 100 iterations and the number of cores employed in a scalability run to the grid size in million volumes.

The first three parameters are well-known in parallel computing literature. The fourth parameter, MPP, provides a way to evaluate different computing platforms for a given CFD application. A computing platform with a smaller MPP value is expected to provide better computing performance compared to other platforms.

An important strength of the HiFUN solver is its ability to scale over several thousand processor cores. This was amply demonstrated in a joint Sandi-Intel study that showed HiFUN can scale over 10,000 Intel Xeon processor cores on the NASA Pleiades supercomputer. The configuration used for the study was the NASA...
Trap Wing (Figure 1). Figures 2 and 3 show the speedup and parallel efficiency curves, respectively, obtained using a grid of 63.5 million volumes. From Figure 2, we can see that HiFUN shows a near-ideal speedup for 4,096 processor cores. It’s also worth noting that for 7,168 processor cores on Pleiades, the parallel efficiency exhibited by HiFUN is about 88%. Also, even for 10,248 processor cores with a modest grid size of about 63.5 million volumes, HiFUN offers reasonable parallel efficiency of about 75%. This scalable parallel performance of HiFUN is a boon to designers because they can expect to have a turnaround time independent of the problem size.
The configuration for the study was the NASA Common Research Model (CRM) used in the 6th AIAA Drag Prediction Workshop. As shown in Figure 4, this configuration represents a transport aircraft with wing, fuselage, nacelle, and pylon. The free-stream Mach number, angle of attack, and free-stream Reynolds number, based on mean aerodynamic chord, are 0.85, approximately 2.6°, and 5 million, respectively. The workload for the analysis has approximately 5.1 million hexahedral volumes. The computed pressure distribution on the surface of NASA CRM is shown in Figure 4.
In this study, we chose two generations of Intel Xeon processors for comparing the single-node performance of the HiFUN solver (Table 1):

1. Intel Xeon E5-2697 v4 processor
2. Intel Xeon Scalable processor Gold 6148

Note that the Intel Xeon Scalable processor has 20 cores, while the Intel Xeon processor has 18 cores. Since the clock frequency of individual processor cores for both processors is nearly the same, based on just the increased numbers of cores, and assuming linear scalability, we expected the performance improvement would be about 11%. But, as shown in Figure 5, the HiFUN solver shows a performance improvement of about 22% on the Intel Xeon Scalable processor compared to the Intel Xeon processor. We can attribute this to:

- Extra processor cores
- Larger L2 cache
- Higher memory speeds
- Higher memory bandwidth from the increased number of memory channels available in the Intel Xeon Scalable processor.

Table 1. Processors used in the study

<table>
<thead>
<tr>
<th>Features</th>
<th>Intel Xeon Processor</th>
<th>Intel Xeon Scalable Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Xeon E5-2697 v4 processor</td>
<td>Intel Xeon Scalable processor Gold 6148</td>
</tr>
<tr>
<td>Cores per socket</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>Speed</td>
<td>2.3 GHz</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Cache (L2/L3)</td>
<td>256 KB/45 MB</td>
<td>1 MB/27 MB</td>
</tr>
<tr>
<td>Memory size</td>
<td>128 GB</td>
<td>192 GB</td>
</tr>
<tr>
<td>Memory speed</td>
<td>2,400 MHz</td>
<td>2,666 MHz</td>
</tr>
<tr>
<td>Memory channels</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 6 shows the MPP for both the processors. As expected, the Intel Xeon Scalable processor has lower MPP compared to the Intel Xeon processor—clearly establishing its superior computing performance. At this stage, we should note that the higher core density of the Intel Xeon Scalable processor leads to improved intra-node parallel performance and results in a compact parallel cluster for a given number of processor cores.
Performance of Intel Xeon Scalable processor compared to Intel Xeon processor

MPP for Intel Xeon Scalable processor compared to Intel Xeon processor
Figure 7 shows a comparison of ideal speedup and actual speedup curves obtained using the HiFUN solver. We can see that as the number of nodes increases, the actual speedup becomes increasingly higher compared to the ideal speedup. We can also see this from Figure 8, where the parallel efficiency curve shows super-linearity for a number of nodes greater than or equal to two. This is because during the scalability study, as the given problem is fragmented into a large number of parts, the reduced memory requirement per processor core means the cache utilization becomes better. But, in general, the performance gain from improved cache utilization is offset by the overhead associated with increased data transfer across the processor cores as the problem is fragmented into a large number of parts.

In our study, we can attribute the ability of the HiFUN solver to exploit the improvement in cache utilization—leading to the super-linear performance—to an excellent interconnect between the nodes and the optimized Intel® MPI Library.

![Speedup of HiFUN on a multi-node Intel processor-based cluster](image-url)
Maximizing HPC Platforms

To summarize, we've shown how the latest-generation Intel Xeon Scalable processor enhances the single-node performance of the HiFUN solver through the availability of large cache, higher core density per CPU, higher memory speed, and larger memory bandwidth through an increased number of memory channels. The higher core density improves intra-node parallel performance and should allow users to build more compact clusters for a given number of processor cores. Finally, we can attribute the super-linear performance of the HiFUN solver on HPC platforms based on the Intel Xeon Scalable processor to an excellent inter-node interconnect and the optimized Intel MPI Library.

References

Sharpen your technical skills. Get expert answers. Dive into new development areas. Explore it all with Intel’s free technical webinars—now available on demand.

CHOOSE YOUR TOPICS >
Materials modeling and design is a very important area of science and engineering. Synthetic materials are a part of modern life. Materials for our computers, cars, planes, roads, and even houses and food are the result of advances in materials science. High-performance computing (HPC) makes modern materials simulation and design possible. And Intel® Software Development Tools can help both developers and end users achieve maximum performance in their simulations.

VASP* is a one of the top applications for modeling quantum materials from first principles. It’s an HPC application designed with computing performance in mind. It uses OpenMP* to take advantage of all cores within a system, and MPI* to distribute computations across large HPC clusters.
Let's see how Intel Software Development Tools can help with performance profiling and tuning of a VASP workload.

**Benchmarking VASP**

To get initial information about VASP performance, we tested it on systems with Intel® Xeon® Scalable processors. (Note that VASP is copyright-protected software owned by the University of Vienna, Austria, represented by Professor Dr. Georg Kresse at the Faculty of Physics. You must have an appropriate license to use VASP.)

Our performance experiments use the VASP developer version that implements MPI/OpenMP hybrid parallelism (see Porting VASP from MPI to MPI+OpenMP [SIMD]) to simulate silicon—specifically, to simulate a vacancy in bulk silicon using HSE hybrid density functional theory.

We use this command to launch VASP:

```
$ mpiexec.hydra -genv OMP_NUM_THREADS ${NUM_OMP} -ppn ${PPN} -n ${NUM_MPI} ./vasp_std
```

In the command:

- `{NUM_OMP}` is the number of OpenMP threads.
- `{PPN}` is the number of MPI ranks on each node.
- `{NUM_MPI}` is the total number of MPI ranks in the current simulation.

For benchmarking, we will use output with LOOP+ time, which measures the execution time of the main computation without pre- and post-processing.

The initial results are for a dual-socket Intel® Xeon® Gold 6148 processor (2.4 GHz, 20 cores/socket, 40 cores/node), shown in Table 1.
The Parallel Universe

Table 1. Initial LOOP+ and MPI_Allreduce times for a cluster with Intel Xeon Gold processors

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP+ (Seconds)</td>
<td>194.01</td>
<td>115.77</td>
<td>75.52</td>
</tr>
<tr>
<td>MPI_Allreduce (Seconds)</td>
<td>25.78</td>
<td>5.44</td>
<td>24.81</td>
</tr>
</tbody>
</table>

Configuration: Hardware: Intel® Xeon® Gold 6148 processor @ 2.40GHz; 192 GB RAM. Interconnect: Intel® Omni-Path Host Fabric Interface (Intel® OP HFI) Adapter 100 Series [discrete]. Software: Red Hat Enterprise Linux® 7.3; IFS 10.2.0.0.158; Libfabric 1.3.0; Intel® MPI Library 2018 (I_MPI_FABRICS=shm:ofi); Intel® MPI Benchmarks 2018 (built with Intel® C++ Compiler XE 18.0.0 for Linux*). Benchmark Source: Intel Corporation.

We will use this data as our baseline performance. We use the MPI_Allreduce code path for this article. The default method based on MPI_Reduce can use the same approach.

Now let’s take a look at how to use Intel Software Development Tools to optimize application performance.

Performance Profiling with Application Performance Snapshot

A typical HPC application is a complex system composed of different programming and execution models. To achieve optimal performance on a modern cluster, the developer should consider many aspects of the system such as:

- MPI and OpenMP parallelism
- Memory access
- FPU utilization
- I/O

Poor performance in any of these aspects will degrade overall application performance. The Application Performance Snapshot (APS) in Intel® VTune™ Amplifier is a great feature to quickly summarize performance in these areas. It’s also freely available as a standalone utility.

From Getting Started with APS, the recommended way to launch MPI applications with APS is to run this command to collect data about your MPI application:

```
$ <mpi launcher> <mpi parameters> aps <my app> <app parameters>
```
where:

- `<mpi launcher>` is an MPI job launcher such as mpirun, srun, or aprun.
- `<mpi parameters>` are the MPI launcher parameters.

Note that:

- `apps` must be the last `<mpi launcher>` parameter.
- `<my app>` is the location of your application.
- `<app parameters>` are your application parameters.

APS launches the application and runs the data collection. When the analysis completes, an `aps_result_<date>` directory is created.

**Intel® MPI Library** also has two very convenient options for integrating external tools into the startup process:

1. `aps`
2. `gtool`

Let's take a look at both.

**–aps Option**

When you use this option, a new folder with statistics data is generated: `aps_result_<date>-<time>`. You can analyze the collected data with the `aps` utility. For example:

```
$ mpirun -aps -n 2 ./myApp
$ aps aps_result_20171231_235959
```

This option explicitly targets `aps` and suits our goals well.

**–gtool Option**

Use this option to launch tools such as Intel VTune Amplifier, Intel® Advisor, Valgrind*, and GNU* Debugger (GDB*) for the specified processes through the `mpiexec.hydra` and `mpirun` commands. An alternative to this option is the `I_MPI_GTOOL` environment variable.

The `–gtool` option is meant to simplify analysis of particular processes, relieving you of specifying the analysis tool's command line in each argument set (separated by colons ':'). Even though it's allowed to use `–gtool` within an argument set, don't use it in several sets at once and don't mix the two analysis methods (with `–gtool` and argument sets).
Syntax:

```
-gtool "<command line for tool 1>:<ranks set 1>[=launch mode 1][@arch 1];
    <command line for tool 2>:<ranks set 2>[=exclusive][@arch 2]; ...
    ;<command line for a tool n>:<ranks set n>[=exclusive][@arch n]"
<executable>
```

or:

```
$ mpirun -n <# of processes>
-gtool "<command line for tool 1>:<ranks set 1>[=launch mode 1][@arch 1]"
-gtool "<command line for a tool 2>:<ranks set 2>[=launch mode 2][@arch 2]"
...
-gtool "<command line for a tool n>:<ranks set n>[=launch mode 3][@arch n]"
<executable>
```

Table 2 shows the arguments.

### Table 2. Arguments

<table>
<thead>
<tr>
<th>&lt;arg&gt;</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;rank set&gt;</td>
<td>Specify the range of ranks that are involved in the tool execution. Separate ranks with a comma or use the '-' symbol for a set of contiguous ranks. To run the tool for all ranks, use the all argument. [Note: If you specify incorrect rank index, the corresponding warning is printed and the tool continues working for valid ranks.]</td>
</tr>
<tr>
<td>[=launch mode]</td>
<td>Specify the launch mode (optional). See below for the available values</td>
</tr>
<tr>
<td>[@arch]</td>
<td>Specify the architecture on which the tool runs (optional). For a given &lt;rank set&gt;, if you specify this argument, the tool is launched only for the processes residing on hosts with the specified architecture. This parameter is optional. For the available values of [@arch], see I_MPI_PLATFORM.</td>
</tr>
</tbody>
</table>

Note that rank sets cannot overlap for the same @arch parameter. A missing @arch parameter is also considered a different architecture. Thus, the following syntax is considered valid:

```
-gtool "gdb:0-3=attach;gdb:0-3=attach@hsw;/usr/bin/gdb:0-3=attach@knl"
```
Also, note that some tools cannot work together, or their simultaneous use may lead to incorrect results. Table 3 lists the parameter values for [launch mode).

Table 3. Parameter values for [launch mode]

<table>
<thead>
<tr>
<th>[launch mode]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exclusive</td>
<td>Specify this value to prevent the tool from launching for more than one rank per host.</td>
</tr>
<tr>
<td>attach</td>
<td>Specify this value to attach the tool from (-gtool) to the executable. If you use debuggers or other tools that can attach to a process in a debugger manner, you need to specify this value. This mode has been tested with debuggers only.</td>
</tr>
<tr>
<td>node-wide</td>
<td>Specify this value to apply the tool from (-gtool) to all ranks where the (&lt;rank\ set&gt;) resides or for all nodes in the case of all ranks. That is, the tool is applied to a higher level than the executable (to the pmi_proxy daemon).</td>
</tr>
</tbody>
</table>

This is a very powerful and complex control option. Intel MPI Library Developer Reference has a whole chapter dedicated to gtool options. We'll focus on those needed for our purposes. They can be controlled with an environment variable (I_MPI_GTOOL) in addition to command line options, so we can leave the launch command untouched. This is extremely useful when you have complex launch scripts that setup everything before the run.

Our desired analysis is simple, so we only need to set:

```
$ export I_MPI_GTOOL="aps:all"
```

and collect the profile for VASP. The profile has the name <name> and can be processed with:

```
$ aps -report=<name>
```

This will generate an HTML file with the results shown in Figure 1.
APS provides actionable information for finding performance bottlenecks. VASP is shown to be MPI bound (19.12% of the runtime was spent in MPI). We can look at the MPI Time section for more details. It shows that MPI Imbalance consumes 10.66% of the elapsed time. MPI Imbalance is defined as time spent in MPI that is unproductive, such as time spent waiting for other ranks to reach an MPI_Allreduce call. The Top 5 MPI functions section shows the most time-consuming functions. In this run, MPI_Allreduce was the top consumer, which suggests that it should be our first target for performance optimizations.

We’ll try using the latest Intel MPI Library 2019 with Intel® Performance Scaled Messaging 2 (PSM2) Multi-Endpoint (Multi-EP) technology to speedup MPI_Allreduce. Let’s begin.
Intel® MPI Library with Intel® Omni-Path Architecture

As you can see from the system specifications provided earlier, we’re using Intel® Omni-Path Architecture (Intel® OPA) hardware. For Intel MPI Library, the best way to take advantage of all Intel Omni-Path Architecture capabilities is to use the Open Fabrics Interface (OFI) for Intel MPI Library 2019. You can find Intel MPI Library 2019 Technical Preview bundled with Intel MPI Library 2018 Update 1 at:

<install_dir>/compilers_and_libraries_2018.1.163/linux/mpi_2019/

Both TMI and OFI support the provider PSM2 as the best option to use Intel OPA.

$ export I_MPI_FABRICS=shm:ofi
$ export I_MPI_OFI_PROVIDER=psm2

Intel Performance Scaled Messaging 2 (PSM2) is a successor of PSM and provides an API and library that targets very large clusters with huge numbers of MPI ranks. See New Features and Capabilities of PSM2 for a good overview.

Intel PSM2 Multi-Endpoint (Multi-EP) and Intel MPI Library 2019

A key concept for Intel PSM2 (and for PSM) is the endpoint. Intel PSM2 follows an endpoint communication model, where an endpoint is defined as an object (or handle) instantiated to support sending and receiving messages to other endpoints. (Learn more [here](#).) By default, every process can use only one endpoint. But with latest versions of Intel PSM2, this has changed, with Intel PSM2 Multi-Endpoint (Multi-EP) functionality allowing the use of endpoints with multithreaded applications. Intel MPI Library 2019 Technical Preview gives the developer the ability to effectively use Intel PSM2 Multi-Endpoint. For Multi-Endpoint support, the MPI standard threading model was extended with the `MPI_THREAD_SPLIT` programming model. It allows a program to effectively use multithreading with MPI to remove most synchronizations and increase performance.
As explained in the Intel® MPI Library Developer Reference for Linux® OS, an MPI_THREAD_SPLIT-compliant program must be at least a thread-compliant MPI program (supporting the MPI_THREAD_MULTIPLE threading level). In addition to that, the following rules apply:

- **Different threads of a process** must not use the same communicator concurrently.
- **Any request created in a thread** must not be accessed by other threads. That is, any non-blocking operation must be completed, checked for completion, or probed in the same thread.
- **Communication completion calls** that imply operation progress such as `MPI_Wait()` or `MPI_Test()` being called from a thread don’t guarantee progress in other threads.

Also, all threads should have an identifier `thread_id`, and communications between MPI ranks can be done only for threads with the same `thread_id`.

### Multi-EP in Collectives: MPI_Allreduce

Since VASP already has a hybrid MPI/OpenMP version, it can readily be adapted for MPI Multi-Endpoint. For each MPI rank, we create as many additional MPI communicators as the number of OpenMP threads with the `MPI_Comm_dup` routine. Then, each MPI buffer will be split between OpenMP threads as shown in **Figure 2**.

![Splitting the MPI buffers among threads](image)

**Figure 2** Splitting the MPI buffers among threads
To skip all code processing, let's use the PMPI_ interface and rewrite the MPI_Allreduce routine with the needed modifications:

```c
#include <stdio.h>
#include <mpi.h>
#include <omp.h>
#include <unistd.h>

int comm_flag=0;
int new_comm=0;
int first_comm=0;
int init_flag=0;
int mep_enable=0;
int mep_num_threads=1;
int mep_allreduce_threshold=1000;
int mep_bcast_threshold=1000;
int k_comm=0;

void mep_init() {
    if(init_flag == 0) {
        mep_enable=atoi(getenv("MEP_ENABLE"));
        mep_num_threads=atoi(getenv("MEP_NUM_THREADS"));
        mep_allreduce_threshold=atoi(getenv("MEP_ALLREDUCE_THRESHOLD"));
        mep_bcast_threshold=atoi(getenv("MEP_BCAST_THRESHOLD"));
    }
    init_flag=1;
}

int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) {
    int err;
    int count_thread;
    int rest=0;
    int num_threads=1;
    int thread_num=0;
    MPI_Aint lb,extent;
    size_t datatype_size;
    if (init_flag == 0) mep_init();
    MPI_Type_get_extent(datatype, &lb, &extent);
    datatype_size=extent;
    ```
// if MPI_Allreduce use MPI_IN_PLACE
if (&sendbuf[0] == MPI_IN_PLACE) {
    if(mep_enable == 1) {
        if(count >= mep_allreduce_threshold) {
            comm_duplicate(comm, mep_num_threads);
            count_thread=count/mep_num_threads;
            rest=count%mep_num_threads;

            #pragma omp parallel num_threads(mep_num_threads)
                private (num_threads, thread_num) {
                    num_threads=omp_get_num_threads();
                    thread_num=omp_get_thread_num();
                    if(thread_num == num_threads-1) {
                        err = PMPI_Allreduce(MPI_INPLACE,
                            recvbuf+thread_num*count_thread*datatype_size,
                            count_thread+rest,
                            datatype, op, comms.dup_comm[k_comm][thread_num]);
                    } else {
                        err = PMPI_Allreduce(MPI_INPLACE,
                            recvbuf+thread_num*count_thread*datatype_size,
                            count_thread,
                            datatype, op, comms.dup_comm[k_comm][thread_num]);
                    }
                }
            else { err = PMPI_Allreduce (MPI_INPLACE, recvbuf, count, datatype, op, comm); }
        } else { err = PMPI_Allreduce (MPI_INPLACE, recvbuf, count, datatype, op, comm); }
    } else { err = PMPI_Allreduce (MPI_INPLACE, recvbuf, count, datatype, op, comm); }
}
}

// else MPI_Allreduce not use MPI_IN_PLACE
else {
    if(mep_enable == 1) {
        if(count >= mep_allreduce_threshold) {
            comm_duplicate(comm, mep_num_threads);
            count_thread=count/mep_num_threads;
            rest=count%mep_num_threads;
            #pragma omp parallel num_threads(mep_num_threads)
                private (num_threads,thread_num) {
                    num_threads=omp_get_num_threads();
                    thread_num=omp_get_thread_num();
                    if(thread_num == num_threads-1) {
                        err = PMPI_Allreduce(sendbuf+thread_num*count_thread*datatype_size,
                            recvbuf+thread_num*count_thread*datatype_size,
                            count_thread+rest,
                            datatype, op, comms.dup_comm[k_comm][thread_num]);
                    } else {
                        err = PMPI_Allreduce(sendbuf+thread_num*count_thread*datatype_size,
                            recvbuf+thread_num*count_thread*datatype_size,
                            count_thread,
                            datatype, op, comms.dup_comm[k_comm][thread_num]);
                    }
                }
        }
    } else { err = PMPI_Allreduce (sendbuf+thread_num*count_thread*datatype_size,
                                recvbuf+thread_num*count_thread*datatype_size,
                                count_thread,
                                datatype, op, comms.dup_comm[k_comm][thread_num]);
    }
}
Note that `comm_duplicate(comm,mep_num_threads)` is a function that duplicates communicators. We will use it with `LD_PRELOAD` to replace the default `MPI_Allreduce`.

**Run with Multi-EP**

The modified command line is:

```bash
$ export I_MPI_THREAD_SPLIT=1
$ export I_MPI_THREAD_RUNTIME=openmp

$ export MEP_ALLREDUCE_THRESHOLD=1000
$ export MEP_ENABLE=1
$ export OMP_NUM_THREADS=2
$ export MEP_NUM_THREADS=2

$ LD_PRELOAD= lib_mep.so
$ mpiexec.hydra -ppn ${PPN} -n ${NUM_MPI} ./vasp_std
```

Performance results are shown in **Table 4**.
Table 4. Final LOOP+ and MPI_Allreduce times for cluster with Intel Xeon Gold processors (the best combination of OpenMP x MPI is reported), comparison with initial data (the best combination of OpenMP x MPI is reported)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP+, seconds</td>
<td>181.55</td>
<td>102.07</td>
<td>65.35</td>
</tr>
<tr>
<td>MPI_Allreduce, seconds</td>
<td>14.71</td>
<td>10.88</td>
<td>13.45</td>
</tr>
<tr>
<td>LOOP+ speedup</td>
<td>1.07</td>
<td>1.13</td>
<td>1.16</td>
</tr>
<tr>
<td>MPI_Allreduce speedup</td>
<td>1.75</td>
<td>2.34</td>
<td>1.84</td>
</tr>
</tbody>
</table>

**Improving Performance**

Using APS, we found that MPI_Allreduce is a performance “hotspot” in the application. With the new Intel MPI Library 2019 Technical Preview and Intel OPA, we were able to more efficiently utilize the system hardware to speedup MPI_Allreduce and improve the overall application. The maximum observed speedup, with only software modifications, was 2.34x for just MPI_Allreduce and 1.16x for LOOP+.

---

**BLOG HIGHLIGHTS**

**How Can Device Software Developers Move from Prototype to Product Faster?**

**BY DINVAR D. [INTEL CORPORATION]**

Developing embedded devices is becoming more and more complex as sensing, compute, memory and communications technologies rapidly evolve. Here, I’ll highlight some of the challenges of software development for devices, as teams cope with shrinking timelines, scarce resources, and rigorous requirements for quality and performance optimization.

In case you feel a bit overwhelmed, rest assured there is a solution with the strength of Intel’s edge-to-cloud ecosystem behind it. More on that, later....
THE PARALLEL UNIVERSE

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer, or learn more at www.intel.com.

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 information go to www.intel.com/performance.

Intel does not control or audit the design or implementation of third party benchmark data or Web sites referenced in this document. Intel encourages all of its customers to visit the referenced Web sites or others where similar performance benchmark data are reported and confirm whether the referenced benchmark data are accurate and reflect performance of systems available for purchase.

This document and the information given are for the convenience of Intel's customer base and are provided "AS IS" WITH NO WARRANTIES WHATSOEVER, EXPRESS OR IMPLIED, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS. Receipt or possession of this document does not grant any license to any of the intellectual property described, displayed, or contained herein. Intel® products are not intended for use in medical, lifesaving, life-sustaining, critical control, or safety systems, or in nuclear facility applications.

Copyright © 2018 Intel Corporation. All rights reserved. Intel, Xeon, Xeon Phi, VTune, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

* Other names and brands may be claimed as the property of others.

Printed in USA   0418/SS   Please Recycle