Advancing Artificial Intelligence on Apache Spark* with BigDL

Why WebAssembly* is the Future of Computing on the Web

Code Modernization in Action: Threading, Memory, and Vectorization Optimizations
CONTENTS

Letter from the Editor  3
What's the Big Deal about BigDL?
by Henry A. Gabb, Senior Principal Engineer, Intel Corporation

Advancing Artificial Intelligence on Apache Spark* with BigDL  5
Features, Use-Cases, and the Future

Why WebAssembly* Is the Future of Computing on the Web  19
The History and New Direction of Processing in the Browser

Code Modernization in Action: Threading, Memory, and Vectorization Optimizations  29
Using the Latest Intel® Software Development Tools to Make More Efficient Use of Hardware

In-Persistent Memory Computing with Java*  43
The Key to Adaptability in Modern Application Programming

Faster Gradient-Boosting Decision Trees  55
How to Lift Machine Learning Performance Using Intel® Data Analytics Acceleration Library

Hiding Communication Latency Using MPI-3* Non-Blocking Collectives  63
Improving HPC Performance by Overlapping Communication and Computation
What's Big Deal about BigDL?

When I started using Apache Spark\textsuperscript{*} for large-scale data analytics, MLlib\textsuperscript{*} was the only option to do machine learning within this framework. MLlib is a good—but limited—package. BigDL brought deep learning to Spark, so when it was first released back in December 2016, I asked the developers for an overview: \textit{BigDL: Optimized Deep Learning on Apache Spark} (\textit{The Parallel Universe}, Issue 28). In this issue’s feature article, we’re checking in with them a year and a half later to see what’s new: \textit{Advancing Artificial Intelligence on Apache Spark with BigDL}.

Imagine being able to develop applications written in your favorite programming language, deploy them on the Web, and achieve near-native performance. In our last issue, we talked about OpenCV.js\textsuperscript{*}, a new technology to do sophisticated computer vision computations within a Web browser (see \textit{Computer Vision for the Masses} in \textit{The Parallel Universe}, Issue 32). In this issue, \textit{Why WebAssembly\textsuperscript{*} Is the Future of Computing on the Web} describes another new technology to run complex computations inside the browser.

I once evaluated a programming tool that gathered reams of performance data for my application and displayed it in a concise way. At first, I was enthralled by the amount of data at my disposal and the colorful GUI. But when the novelty wore off, I realized that none of the data was actionable—or otherwise helpful in tuning my application. \textit{Code Modernization in Action} demonstrates step-by-step how to turn Intel\textsuperscript{®} Parallel Studio XE analyses into code optimizations.

Non-volatile memory is becoming an increasingly important hardware technology. \textit{In-Persistent Memory Computing with Java\textsuperscript{*}} describes libraries that allow Java applications to use this technology. Expect more articles on non-volatile memory in future issues of \textit{The Parallel Universe}.

\textit{Faster Gradient-Boosting Decision Trees} describes one of the many new features and enhancements in the latest release of Intel\textsuperscript{®} Data Analytics Acceleration Library.
The Intel® MPI Library is also being continuously optimized to take advantage of new hardware and changes to the MPI standard. Always overlap communication and computation to hide latency in MPI has been the conventional wisdom for as long as I can remember. But demonstrating the performance benefit of communication-computation overlap in real applications has been elusive. So when non-blocking collectives were added to the MPI standard a few years ago, I didn’t pay much attention. *Hiding Communication Latency Using MPI-3 Non-Blocking Collectives* is causing me to rethink these new functions, especially if I have to harness a large number of compute nodes to process a large dataset.

Future issues of *The Parallel Universe* will bring you articles on using FPGAs for deep learning, threading in Python®, new approaches to large-scale distributed data analytics, new features in Intel® software tools, and much more. Be sure to *subscribe* so you won’t miss a thing.

*Henry A. Gabb*

July 2018
ADVANCING ARTIFICIAL INTELLIGENCE ON APACHE SPARK* WITH BIGDL
Features, Use-cases, and the Future

Jason Dai, Senior Principal Engineer, and Radhika Rangarajan, Engineering Manager, Intel Corporation

BigDL has evolved into a vibrant open-source project since Intel introduced it in December of 2016. In this article, we give details on its implementation, describe some real-world use-cases, and provide a glimpse into the new end-to-end analytics plus artificial intelligence (AI) pipelines (the Analytics Zoo platform) being built on top of Apache Spark* and BigDL. [Editor’s note: We ran the article BigDL: Optimized Deep Learning on Apache Spark in The Parallel Universe, Issue 28, describing the first version of this framework. Much has changed since then.]
Why BigDL?

Demand is growing for organizations to apply deep learning technologies—computer vision, natural language processing, generative adversarial networks—in their big data platforms and analytics pipelines. To support this convergence of deep learning and big data analytics, Intel open-sourced the BigDL distributed deep learning framework back in December of 2016.

BigDL is implemented as an Apache Spark library that lets users write large-scale deep learning applications (including model training and inference) as standard Spark programs that can run directly on existing big data (Hadoop* or Spark) clusters.

BigDL provides comprehensive support for deep learning technologies:

- Neural network operations
- Layers
- Losses
- Optimizers

In particular, users can run existing models defined in other frameworks (i.e., TensorFlow*, Keras*, Caffe*, and Torch*) directly on Spark in a distributed way. BigDL also provides seamless integration of deep learning technologies into the big data ecosystem. A BigDL program can not only interact directly with Spark components (e.g., DataFrames, Spark Streaming, ML Pipelines), it can also run in a variety of big data frameworks (e.g., Apache Storm*, Apache Flink*, Apache Kafka*).

Writing BigDL Applications

Since BigDL is implemented in Apache Spark, users can easily do end-to-end data analytics, as illustrated in the five-step sequence below.

**Step 1:** Read text data (articles and associated labels) into a Spark resilient distributed dataset (RDD).

```python
spark = SparkContext(appName="text_classifier", ...) 
texts_rdd = spark.textFile("hdfs://...")
words_rdd = texts_rdd.map(lambda text, label:
(w for w in to_words(text)], label))
```
Step 2: Transform loaded data into an RDD of training samples using RDD transformations.

```python
w2v = news20.get_glove_w2v(dim=...)
vector_rdd = words_rdd.map(lambda word_list, label
    ([to_vec(w, w2v) for w in word_list], label))
sample_rdd = vector_rdd.map(lambda vector_list, label:
    to_sample(vector_list, label))
```

Step 3: Construct a neural network model.

```python
model = Sequential().add(Recurrent().add(LSTM(...)))
    .add(Linear(...))
    .add(LogSoftMax())
```

Step 4: Train the model (with the specified loss and optimization method).

```python
loss = ClassNLLCriterion()
optim_method = Adagrad()
optimizer = Optimizer(model=model, training_rdd=sample_rdd,
        criterion=loss, optim_method=optim_method, ...)
optimizer.set_train_summary(summary=TrainSummary(...))
trained_model = optimizer.optimize()
```

Step 5: Predict results using existing models (pretrained in BigDL, TensorFlow, Keras, Caffe, or Torch).

```python
test_rdd = ...
prediction_rdd = trained_model.predict(test_rdd)
```

Distributed Training

BigDL provides highly scalable, distributed training using synchronous mini-batch stochastic gradient descent and data parallelism. It implements a parameter server (PS) architecture (through an all-reduce operation) directly on top of the BlockManager inside Spark. After each task computes its local gradients, instead of sending them back to the driver, it locally aggregates the gradients from all partitions within a single worker. Then the aggregated gradient on each node is sliced into chunks that are exchanged among all the cluster nodes. Each node is responsible for a specific chunk, which in essence implements a PS architecture in BigDL for parameter synchronization. Each node retrieves and aggregates gradients for the slice of the model...
that it’s responsible for computing. After the pairwise exchange completes, each node has its own portion of aggregated gradients, which it uses to update its own portion of weights. Then the exchange happens again to synchronize the updated weights. At the end of this procedure, each node has a copy of the updated weights. As a result, BigDL can perform highly scalable, distributed training of deep learning models (Figure 1).

Model Quantization

Quantization refers to technologies that store numbers and perform calculations in more compact, lower-precision form. BigDL takes advantage of this type of low-precision computing to quantize existing models for optimized inference. BigDL first loads existing models, then quantizes the parameters of selected layers (e.g., spatial convolution) into 8-bit integer form to produce a quantized model (Figure 2).
During model inference, each quantized layer:

- **Quantizes** the 32-bit floating point input to 8-bit integer
- **Performs** the 8-bit calculations (such as GEMM) using the quantized parameters and data
- **Dequantizes** the results to 32-bit floating point

Many of these operations can be fused in the implementation, so the quantization and dequantization operations in each small, local quantization window (a small sub-block, such as a patch or kernel in convolution, of the parameters or input data). As a result, BigDL can use very low-bit integers (i.e., 8-bit) in model quantization with minimal loss in accuracy (less than 0.1%), a 4x reduction in model size, and up to a 2x speedup in inference (Figure 3).

\[
\text{Math.round}(1.0 \times \text{value} / \text{Math.max(Math.abs(max), Math.abs(min))} \times \text{Byte.MaxValue}).\text{toByte}
\]

2 **Equation for quantizing 32-bit floating point to 8-bit integer**

Model quantization results (accuracy, inference speed, and model size) for SSD, VGG16, and VGG19 using BigDL 0.3.0 and Amazon Web Services* EC2 C5.18x large instances²
Local Execution

Besides being a standard Spark program, BigDL also provides support for model training and inference on a local Java* virtual machine (JVM) without Spark. This helps improve efficiency when running BigDL on a single node because there’s no overhead from parameter synchronization or task scheduling. More importantly, it becomes easier to directly integrate BigDL models (for either inference or fine tuning) with various JVM-based big data frameworks (e.g., Apache Storm, Flink, or Kafka).

Use-Case and Applications

In this section, we describe three typical BigDL use-cases:

1. Model inference
2. Distributed training
3. Transfer learning

Image Feature Extraction

JD.com is one of the largest online retailers in the world. It’s built an end-to-end object detection and image feature extraction pipeline on top of BigDL and Spark (Figure 4). The pipeline first reads hundreds of millions of pictures from a distributed database into Spark (as an RDD of pictures), and then preprocesses the RDD (including resizing, normalization, and batching) in a distributed fashion using Spark. After that, it uses BigDL to load an SSD model (pretrained in Caffe) for large-scale, distributed object detection on Spark. This generates the coordinates and scores for the detected objects in each of the pictures. It then generates the target images (by keeping the highest-scoring object as the target and cropping the original picture based on the coordinates of the target) and further preprocesses the RDD of target images (including resizing and batching). Finally, it uses BigDL to load a DeepBit model (again, pretrained in Caffe) for distributed feature extraction of the target images to generate the corresponding features. Finally, it stores the results (i.e., an RDD of extracted object features) in the Hadoop Distributed File System (HDFS).

The entire data analytics and deep learning pipeline—including data loading, partitioning, preprocessing, model inference, and storing the results—is easy to implement under a unified programming paradigm using Spark and BigDL. The end-to-end pipeline also delivers a speedup of approximately 3.83x compared to running the same solution on a GPU cluster (Figure 5), as reported by JD.³
End-to-end object detection and image feature extraction pipeline (using SSD and DeepBit models) on top of Spark and BigDL.

Throughput of GPU clusters and Intel Xeon processor-based clusters for the image feature extraction pipeline benchmarked by JD. The GPU throughput is tested on 20 NVIDIA Tesla* K40 cards, and the Intel Xeon processor throughput is tested on 1,200 logical cores (where each dual-socket Intel Xeon processor E5-2650 v4-based server runs 50 logical cores).
Precipitation Nowcasting

Cray has integrated BigDL into its Urika-XC* analytics software suite and built an end-to-end precipitation nowcasting (predicting short-term precipitation) workflow that includes data preparation, model training, and inference (Figure 6). First, the application reads more than a terabyte of raw radar scan data into Spark as an RDD of radar images and then converts it into an RDD of NumPy* ndarrays. It then trains a sequence-to-sequence model using a sequence of images leading up to the current time as the input with a sequence of predicted images, arbitrarily far in the future, as the output. After the model is trained, it can be used for predictions such as precipitation patterns for the next hour (Figure 7).

6 End-to-end precipitation nowcasting workflow on Spark and BigDL

7 Predicting precipitation patterns for the next hour using Spark and BigDL
Image Similarity-Based Recommendations

**MLSLListings Inc.** is a large real estate multiple listing service (MLS) that’s been building an image-similarity-based house recommendation system on Spark and BigDL. The end-to-end workflow is implemented through transfer learning (including feature extractions and fine-tuning) to compute both the semantic and visual similarity of house photos (Figure 8). To compute the semantic similarity for the photos, the system fine-tunes the Inception v1 model pretrained on the Places dataset⁴ to build three new classifiers that determine whether the photo shows the house front exterior, the house style, and the number of stories.

It first loads three pretrained Inception v1 models and then appends two new layers (a fully-connected layer followed by a Softmax* layer) to each model to train the new classifiers. After training, these classifiers can be used to tag (or label) each house listing photo. To compute the visual similarity, the system uses the VGG-16 model pretrained on the Places dataset to extract the image feature for each house listing photo. This is then combined with the tags generated by the classifiers and stored as a distributed table. At model serving time, the user can select a house listing photo and have the system recommend listings with similar visual characteristics (Figure 9).
What's Next for BigDL?

While BigDL is integrated into Spark and extends its capabilities to address the challenges of big data application developers, will a library alone be enough to simplify and accelerate the deployment of ML/DL workloads on production clusters? Based on the community feedback and customer implementation, Intel has built Analytics Zoo, an end-to-end analytics plus AI platform that makes it easy to build Spark and BigDL applications by providing high-level pipeline APIs and built-in deep learning models, with reference use-cases.

High-Level Pipeline APIs

Analytics Zoo provides a set of easy-to-use, high-level pipeline APIs on top of Spark and BigDL, including:

- **nnframes**: Native deep learning support in Spark DataFrames and ML Pipelines
- **autograd**: Building custom layer/loss using auto-differentiation operations
- **Transfer learning**: Customizing pretrained models for feature extraction or fine tuning
Using these high-level pipeline APIs, users can easily build complex deep learning pipelines in just a few lines of code.

**Step 1:** Initialize `nnContext` and load images into DataFrames using `NNImageReader`.

```python
from zoo.common.nncontext import *
from zoo.pipeline.nnframes import *
sc = init_nncontext ()
imageDF = NNImageReader.readImages(image_path, sc)
```

**Step 2:** Process data using DataFrames transformations.

```python
getName = udf(lambda row: ...)
getLabel = udf(lambda name: ...)
df = imageDF.withColumn("name", getName(col("image"))) \
    .withColumn("label", getLabel(col('name')))
```

**Step 3:** Process images using built-in feature engineering operations.

```python
from zoo.feature.image import *
transformer = RowToImageFeature() \ 
    -> ImageResize(64, 64) \ 
    -> ImageChannelNormalize(123.0, 117.0, 104.0) \ 
    -> ImageMatToTensor() \ 
    -> ImageFeatureToTensor()
```

**Step 4:** Load an existing model (in this case, pre-trained in Caffe), remove the last few layers, and freeze the first few layers.

```python
from zoo.pipeline.api.net import *
full_model = Net.load_caffe(def_path, model_path)

# Remove layers after pool5/drop_7x7_s1
model = full_model.new_graph(["pool5/drop_7x7_s1"])

# freeze layers from input to pool4/3x3_s2 inclusive
model.freeze_up_to(["pool4/3x3_s2"])
```
Step 5: Add a few new layers using a Keras-style API and a custom Lambda layer.

```python
from zoo.pipeline.api.autograd import *
from zoo.pipeline.api.keras.layers import *
from zoo.pipeline.api.keras.models import *

def add_one_func(x):
    return x + 1.0

input = Input(name="input", shape=(3, 224, 224))
inception = model.to_keras()(input)
flatten = Flatten()(inception)
lambda = Lambda(function=add_one_func)(flatten)
logits = Dense(2)(lambda)
newModel = Model(input, logits)
```

Step 6: Train the model using Spark ML Pipelines.

```python
cls = NNClassifier(model, CrossEntropyCriterion(), transformer) \
    .setLearningRate(0.003).setBatchSize(40) \
    .setMaxEpoch(1).setFeaturesCol("image") \
    .setCachingSample(False)
nnModel = cls.fit(df)
```

Built-In Deep Learning Models

Analytics Zoo provides several built-in deep learning models you can use for a variety of problem types such as object detection, image classification, text classification, and recommendation.

- **Object detection**: Using the Object Detection API (including a set of pretrained detection models such as SSD and Faster-RCNN), you can easily build object detection applications (e.g., localizing and identifying multiple objects in images and videos).
- **Image classification**: Using the Image Classification API (including a set of pretrained detection models such as VGG, Inception, ResNet, and MobileNet), you can easily build image classification applications.
- **Text classification**: The Text Classification API provides a set of pre-defined models (e.g., CNN, LSTM) for text classification applications.
- **Recommendation**: The Recommendation API provides a set of pre-defined models (e.g., Neural Collaborative Filtering, Wide and Deep Learning) to create recommendation engines.
You can easily use the Analytics Zoo Object Detection API for out-of-box inference on a large set of images.

**Step 1:** Download object detection models in Analytics Zoo from the Detection Model Zoo. A collection of detection models (pretrained on the PSCAL VOC and COCO datasets) is available.

**Step 2:** Load the image data and object detection model.

```python
from zoo.common.nncontext import get_nncontext
from zoo.models.image.objectdetection import *

spark = init_nncontext()
image_set = ImageSet.read(img_path, spark)
```

**Step 3:** Use the Object Detection API for off-the-shell inference and visualization.

```python
output = model.predict_image_set(image_set)

visualizer = Visualizer(model.get_config().label_map(), 
    encoding="jpg")
visualized = visualizer(output).get_image(to_chw=False) 
    .collect()

for img_id in range(len(visualized)):
    cv2.imwrite(output_path + '/' + str(img_id) + '.jpg', 
        visualized[img_id])
```

**The Big Deal about BigDL**

BigDL is a work in progress, but initial experience and feedback are encouraging. Since its initial release in December of 2016, it’s received over 2,400 stars on GitHub* and enabled many users to build new analytics and deep learning applications on top of Hadoop and/or Spark clusters. The new Analytics Zoo project will make it even easier to build BigDL applications by providing an end-to-end analytics plus AI platform on top of Apache Spark and BigDL.

**References**

2. Jason (Jinquan) Dai, et al. "Leveraging Low Precision and Quantization for Deep Learning Using the Amazon EC2 C5 Instance and BigDL."
Welcome to Tech.Decoded, the New Knowledge Hub for Developers

You’ll find an always-growing library of information curated to help you get the most out of modern hardware. Boost your competitive edge. And get to market faster.

- Code Modernization
- Visual Computing
- Data Center and Cloud Computing
- Data Science
- Systems and IoT

Get Expert Insights
Watch forecasters and visionaries explore today’s tech landscape and identify the hottest trends.

Dig Deeper
Learn how to get every last ounce of performance from your code with on-demand webinars covering today’s most important strategies, practices, and tools.

Put it All to Work in your Code
Use short videos and articles to understand the how-to’s of key programming tasks using specific development tools.

EXPLORE TECH.DECODED NOW >

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.
© Intel Corporation
The Parallel Universe

The Parallel Universe

Rich Winterton, Deepti Aggarwal, and Tuyet-Trang (Snow) Lam Piel, Software Engineers, andBrittney Coons and Nathan Johns, Software Engineer Interns, Intel Corporation

In 1983, the Advanced Research Projects Agency Network (ARPANET), a project of the U.S. Defense Department, adopted TCP/IP networking protocols—and the Internet was created. At first, the Internet was mostly used for data and file sharing. In 1992, it was opened up to commercial use. And since then, its capabilities have increased, and use-cases have diversified, to make it arguably the world's most important technology.

The browser is a software application that delivers content to the client's computer. At first, the browser was simply an application to access data over the World Wide Web. Over time, the browser came to be used
for more complex Web applications—such as gaming applications and hardware codecs—which increased computation and performance requirements.

As different browsers competed for market share, synthetic benchmarks were created to show which browser was fastest. These benchmarks included Apple's Sunspider*, later replaced by JetStream*², Mozilla's Kraken*³, and Google's Octane* 1.0 (later replaced by Octane 2.04). These benchmarks were designed not only to compare the browser speed, but also to show off and improve the code generated by the JavaScript* engines. Many synthetic browser benchmarks stressed JavaScript performance, and JavaScript adoption increased during this time as hardware and software improved.

Meanwhile, browser plug-ins like Adobe Flash* and Google's Native Client* (NaCl*) were introduced to improve media and gaming application performance. Plug-ins weren't part of Web standards, so it was hard to accelerate video on many of the platforms. It was a challenging task to enable implementations, such as Adobe Flash Stage Video*, across different platforms in different browsers. Engineers from YouTube and Intel created the Stats for Nerds interface in YouTube to help hardware vendors identify and resolve hardware and software video decode performance issues. Around this same time, the sandboxing technique of putting the graphics processing unit (GPU) into a separate process enabled browser vendors to use the GPU to render webpages. This allowed secure hardware compositing and rendering on most platforms, and also prevented rogue websites from locking the browser—or worse, the system.

In 2015, browser vendors began to pay more attention to how browsers were actually being used instead of how they performed on synthetic benchmarks. Figure 1 shows the percentage of time spent at different stages when running common websites and benchmarks. We would expect most major browsers to have similar component breakdowns. Figure 1 shows significantly less time spent in the JavaScript's just-in-time (JIT) code in websites like YouTube.com, Wikipedia.org, Amazon.com, and Intel.com compared to the Kraken and Octane synthetic benchmarks. Another issue with these benchmarks is that browser JavaScript engines used knowledge of the benchmark code to start gaming the performance results. In 2014, Apple released a more realistic benchmark called Speedometer*, which imitated the functionality of real websites. This similarity is apparent in Figure 1.
Compilation of JavaScript code consumes a lot of time, memory, and power. With growing demand for use of browser components and browsers as a framework for Web applications, vendors soon realized it was time for a change to be able to take performance to the next level. They began changing the browser’s design for the typical website and future Web application models. Google and Mozilla were early adopters of these changes, introducing asm.js and NativeClient (NaCl), two technologies that led to the development and acceptance of the need for a new Web technology. Browser vendors quickly realized it was challenging to develop a new Web technology that would be universally accepted. Several browser vendors, including Apple, Google, and Mozilla, came together to solve this problem, which led to the development of WebAssembly (WASM).

This new technology was designed to compile languages typically used by native or managed code—instead, compiling the code to a new type of bytecode. The bytecode generated by WASM is a portable stack machine designed to provide near-native compilation performance for general applications. With a portable infrastructure and good performance, WASM enabled Web applications to be more like native applications, with the ability to run well and to run when not connected to the Internet.

Progressive Web applications (PWAs) are rapidly gaining momentum. They're designed to first look locally for the needed resources, and then to use these resources if they're available. When the application connects to the Internet, it checks to see if the local resources are up to date. If they are, then the application is prepared for offline use. If there are new or updated resources, then it updates the local database. This functionality didn't exist in Web standards until November 2017, when W3C published the draft specification for service workers. Service workers enable PWAs to use or update cached assets before getting data from the network, a technique known as "offline first."

<table>
<thead>
<tr>
<th></th>
<th>JavaScript</th>
<th>Blink C++</th>
<th>V8 C++</th>
<th>Parse</th>
<th>IC</th>
<th>GC Total</th>
<th>Compile</th>
<th>Optimize</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>YouTube.com</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wikipedia.org (United States)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon.com</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel.com</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speedometer 2.0 Benchmark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kraken 1.1 Benchmark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Octane 2.0 Benchmark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 These tests were run on an Intel® Core™ i7-6770HQ 2.60GHz processor using Google Chrome® on Ubuntu® 16.04. The recorded percentages correspond to the percent of time spent in each region.

For more complete information about compiler optimizations, see our Optimization Notice.
The Parallel Universe

PWAs can be functionally similar to most native applications, but they have the advantage of being smaller, with little to no explicit installation or update required by the user. Some of the most popular PWAs are Gmail*, Twitter*, and Google Maps.* Typically, only one in four applications will still be installed 24 hours after installation. After 90 days, fewer than 5% will still be installed. PWAs have no noticeable install or update, so retention is significantly improved and their value can be measured by user activity and engagement. For example, Twitter reports that as a mobile PWA, it has seen a 65% increase in content viewed and a 75% increase in user engagement (Tweets sent), with less than a 3% bounce rate (leaving from the application after the first page). Another example is AliExpress*, an online retail service in China, which improved its conversion rate (a ratio of visits to purchases) by over 2x. These kinds of improvements significantly increase revenue. As a result, many businesses are transitioning to PWAs and Forbes predicts that “PWAs will replace native mobile applications.”

WASM is key to addressing the performance needs of current and future Web applications that can increase performance of computationally intensive tasks up to 3x compared to JavaScript. This performance boost will greatly benefit PWAs.

Moving Web Computing to the Browser

To see how Web computing is moving to the browser, consider the still image edge detection example in Figure 2, which explores a simple edge detection using the Sobel* filter algorithm and shows a substantial difference between the performance of JavaScript and WASM with a native code implementation as a baseline. This algorithm takes an input image (on the left), calculates relative edges, and displays them on a spectrum of black to white depending on the intensity of the edge (on the right). This occurs as the image is either imported as grayscale or converted to grayscale. The algorithm iterates through the original image pixel by pixel, compares each pixel to its surrounding pixels, and calculates a new pixel for the Sobel image (Figure 3).
3 Sobel edge detection algorithm

As Figure 3 shows, matrix multiplication occurs between the original image data and each of the two filters. The Hx filter calculates the relative change in pixel data in the x direction (\(G_x\)), while the Hy filter calculates the relative change in pixel data in the y direction (\(G_y\)). Both filters are necessary to get an accurate representation of the edges found in the original image (Figure 4).

4 Images created using either the Hx or Hy portion of Sobel filter, but not both
You can see C++ and JavaScript implementations of the Sobel edge detection algorithm in Figures 5 and 6, respectively. The method to read the input data had to be changed from reading a local file to getting the data from the HTML canvas. Other than that, very few changes were needed. The JavaScript code takes an input image and converts it to RGBA image data so that it can be passed into the Sobel filter function. The data is then returned to the browser to be displayed on the canvas.

The C++ and JavaScript implementations are nearly identical, which allows for reasonable performance comparisons (Figure 7). The native C++ implementation performs about 6x better than the JavaScript implementation, so it's clear that complex computations like edge detection require better computing capabilities. Obviously, better computing capabilities make PWA technology a viable option for more applications. To compare the performance of natively compiled C++, JavaScript-generated code, and WASM, we compiled the Sobel C++ source code to WASM using Emscripten*16 and ran it within the browser. As you can see, WASM significantly closes the performance gap between the native binary and the JavaScript code (Figure 8).

```c++
float magnitudeValue(unsigned char **imageData, int row, int col, int imageHeight, int imageWidth) {
  // accumulation 1
  if ((row == 0) && (col == 0) && (row != imageHeight - 1) && (col != imageWidth - 1)) { // Center of data
    accumulation1_1 = (float)((h[0][0] * imageData[row - 1][col - 1]) + (h[1][0] * imageData[row - 1][col]) + (h[0][1] * imageData[row - 1][col + 1]));
    accumulation1_2 = (float)((h[0][0] * imageData[row][col - 1]) + (h[1][0] * imageData[row][col]) + (h[0][1] * imageData[row][col + 1]));
    accumulation1_3 = (float)((h[0][0] * imageData[row + 1][col - 1]) + (h[1][0] * imageData[row + 1][col]) + (h[0][1] * imageData[row + 1][col + 1]));
  } // Repeat with h[y] to calculate accumulations
  accumulation1 = accumulation1_1 + accumulation1_2 + accumulation1_3;
  return magnitude;
}

unsigned char** sobelFilter(unsigned char** imageData, int imageHeight, int imageWidth) {
  unsigned char** outputData = new unsigned char*[imageHeight];
  for (int row = 0; row < imageHeight; row++) {
    outputData[row] = new unsigned char[imageWidth];
    for (int col = 0; col < imageWidth; col++) {
      mag = sqrt(magnitudeValue(imageData, row, col, imageHeight, imageWidth));
      mag = (mag > 255) ? 255 : mag;
      outputData[row][col] = mag;
    }
  }
  return outputData;
}
```

C++ code for the Sobel filter algorithm (complete source code is available on GitHub15)
The Parallel Universe

JavaScript code for the Sobel filter algorithm (complete source code is available on GitHub)

Performance of Sobel filtering on a 4,160 x 3,160 pixel image running on an Intel Core i7-6770HQ 2.60GHz processor with Ubuntu 16.04 (blue bars) and an Intel Core i5-7300U 2.60GHz processor with Windows* 10 (orange bars). In JavaScript, performance.now() was used for timing. In C++, a high-precision clock was used for timing. JavaScript tests were run using Google Chrome.
The Parallel Universe

What the Future May Hold

The future is anyone’s guess, but the trends are getting clearer:

- **PWAs** will replace many mobile and native applications.
- **Web technologies** are the simplest and best way to send and receive data.
- **Machine learning and artificial intelligence** are used by home and assistant devices and applications—not only to make the application more customizable and relevant to the user, but also to improve PWA implementation itself.

This makes client and edge computing significantly more important.

Fueling this need are several near-term enhancements to WebAssembly such as parallel and JIT compilation of WASM code. This technique takes advantage of multi-core platforms and significantly improves the first-time or update runtime of an application being compiled by the client. Memory management is improving by moving to a parallel implementation of garbage collection within the JavaScript/WASM memory management software. Native implementations of applications are still the best-performing option if complex computation is required, because code is compiled non-real-time and can take advantage of auto-vectorizing and hand-customized SIMD much more easily. But this advantage will be minimized in the near future when WASM adds auto-vectorization by simply compiling and linking the C/C++ code with the LLVM WASM framework.

---

Performance of Sobel filtering on a 4,160 x 3,160 pixel image running on an Intel Core i7-6770HQ 2.60GHz processor with Ubuntu 16.04 (blue bars) and an Intel Core i5-7300U 2.60GHz processor with Windows 10 (orange bars)
Addressing Tomorrow’s Demands

The Web browser is the dominant application of the Internet revolution. In the beginning, the browser was just an application to provide content to the client. However, the growing demand for performance, and new use-cases, have changed the browser tremendously. New benchmarks and languages were introduced to address power and performance problems, but it quickly became clear that meeting the demands of current and future Web applications would require better performance.

WASM is a new technology to address these performance demands. WASM was designed to compile languages that are typically used by native or managed code. It generates portable bytecode to provide near-native compilation performance for general applications.

To show the benefits of WASM, we implemented a Sobel filtering algorithm in C++, JavaScript, and WASM. As expected, WASM, without any SIMD capabilities, was able to deliver 3x to 4x better performance over general JavaScript. We believe that WASM, especially after the introduction of new capabilities such as SIMD auto-vectorization, will be a key technology to satisfy the performance demands of current and future PWAs.

References

2. JetStream Benchmark
3. Kraken Benchmark
4. Octane Benchmark
5. Asm.js
6. Native Client (NaCl)
7. WebAssembly
8. Progressive Web Application (PWA)
9. W3C Specifications for Service Workers
15. Sample Source Code
16. Setting Up Emscripten
UNLEASH A FASTER PYTHON ON YOUR DATA

Supercharge your applications with Intel® Distribution for Python*

FREE DOWNLOAD >

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
High-performance computing (HPC) systems are a complex combination of hardware components. And hardware vendors are constantly working to increase speed, reduce latency, add cores, and make vectors longer to improve the performance of compute-intensive applications. But simply running an application on a system with better components doesn't guarantee those components will be used efficiently. Getting the expected performance boost may still take code modifications.

Fortunately, Intel® Parallel Studio XE has tools that cover all aspects of code tuning on Intel® hardware. Sometimes it's hard to know which tool to start with, so the Intel® VTune™ Amplifier Application Performance
Snapshot (APS) is a good way to get a quick summary of your application’s performance characteristics—particularly the bottlenecks limiting performance. You can use the snapshot to select the right Intel Parallel Studio XE tool to tune cluster-, node-, and/or core-level performance (Figure 1).

Iso3DFD*: A Wave Propagation Kernel and its Performance Measurement

Let’s look at how the tools help to find bottlenecks and improve the performance of an example code. Iso3DFD implements the isotropic acoustic wave equation:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \nabla^2 p$$

where $\nabla^2$ is the Laplace operator, $p$ is the pressure field, and $c$ is the velocity field. Finite differences can be used to express $p_{t+1}$ as a function of $p_t$ and $p_{t-1}$. Implementing a finite difference propagation kernel can be done with a stencil pattern. In three dimensions, a stencil looks like a 3D cross that we move across the pressure field. In other words, to update $p_{t+1}[x,y,z]$, we need to access all the neighbors of $p_t[x,y,z]$ in 3D. Figure 2 demonstrates the stencil pattern in 2D.
A 2D stencil pattern

In a real-world application, physicists usually implement a specific propagation methodology for corner cases. These corner cases are called boundary conditions. To keep the code simple in this example, we don’t propagate the wave on the borders. The straightforward implementation in Figure 3 computes a single iteration of the stencil.

```c
void iso_3dfd_it(float *ptr_next, float *ptr_prev, float *ptr_vel, float *coeff, const int n1, const int n2, const int n3,
                const int num_threads, const int n1_block, const int n2_block, const int n3_block)
{
  int dim1n2 = n1*n2;
  for(int ix=0; ix<n1; ix++)
  {
    for(int iy=0; iy<n2; iy++)
    {
      for(int iz=0; iz<n3; iz++)
      {
        if( ix>=HALF_LENGTH && ix<n1-HALF_LENGTH &&
            iy>=HALF_LENGTH && iy<n2-HALF_LENGTH &&
            iz>=HALF_LENGTH && iz<n3-HALF_LENGTH )
        {
          int offset = ix*dim1n2 + iy*n1 + ix;
          float value = 0.0;
          value += ptr_prev[offset]*coeff[0];
          for(int ir=1; ir<=HALF_LENGTH; ir++)
          {
            int offset = (ptr_prev[offset + ir] + ptr_prev[offset - ir]); // horizontal
            value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr_prev[offset - ir*n1]); // vertical
            value += coeff[ir] * (ptr_prev[offset + ir*dim1n2] + ptr_prev[offset - ir*dim1n2]); // in front / behind
          }
          ptr_next[offset] = 2.0f*ptr_prev[offset] - ptr_next[offset] + value*ptr_vel[offset];
        }
      }
    }
  }
}
```

Computing a single iteration of the stencil
Since this article will focus on single-node performance, the example code will not use MPI to implement cluster-level parallelism.

To establish baseline performance, we benchmarked the initial implementation using a 512 x 400 x 400 grid running on a dual-socket Intel® Xeon® Gold 6152 processor (2.1 GHz, 22 cores/socket) system (Red Hat Enterprise Linux Server* 7.4). Use this command to set up the Intel Parallel Studio XE environment:

```
> source <Parallel_Studio_install_dir>/psxevars.sh
```

Let's use Intel VTune Amplifier's Application Performance Snapshot to quickly find tuning opportunities in the initial Iso3DFD implementation. This command invokes APS:

```
> aps ./Iso3DFD
```

Figure 4 shows the APS report that appears in a Web browser. Of course, the initial single-threaded version of the application only uses one of the 44 physical cores on the system. This was recognized by APS as the major performance opportunity, so let's parallelize the code.

![APS report for the original single-threaded code](image)

For more complete information about compiler optimizations, see our Optimization Notice.
Optimization Step 1: Introducing Thread-Level Parallelism with OpenMP*

There are no dependencies in the loops, so let’s do a straightforward parallelization using the OpenMP* “parallel for” pragma on the outermost loop because it has the most work per iteration (Figure 5).

```c
void do_work(float *next, float *prev, float *vel, float *coeff, const int n1, const int n2, const int n3,
    const int num_threads, const int n1_block, const int n2_block, const int n3_block) {
    int dimx = n1*n2;
    for(int n=0; n<n1; n++) {
        for(int m=0; m<n2; m++) {
            x(n) = HALF_LENGTH + x(n1-n1_HALF_LENGTH) +
                y(m) = HALF_LENGTH + y(n2-n2_HALF_LENGTH) +
                z(n,m) = HALF_LENGTH + z(n1-n1_HALF_LENGTH)
        }
    }
}
```

Doing a quick check with APS shows that elapsed time has significantly improved (Figure 6). The snapshot reports some load imbalance in the OpenMP, but a speedup of approximately 38x is a good start. The more pressing performance issue is that the application is memory-bound. We need to improve memory access to remove this bottleneck.
Optimization Step 2: Using Loop Interchange to Improve Memory Access

Memory stalls due to suboptimal memory access are now the major bottleneck in our application, so let’s run an Intel VTune Amplifier Memory Access Analysis, as advised in the APS hints (Figure 6):

>`amplxe-cl -collect memory-access ./Iso3DFD`

The analysis results are shown in Figure 7. The application is DRAM-bound, but memory bandwidth consumption is far from its limit (i.e., the DRAM Bandwidth Bound value is small). This is typical for applications that access memory with long stride. The Intel® Advisor Check Memory Access Patterns analysis can help us to detect which lines of code are causing the problem (Figure 8).

![Intel VTune Amplifier Memory Access Analysis summary](image-url)
Intel Advisor Memory Access Pattern Analysis

A stride of 204,800 is exactly the size of our first two dimensions (512 x 400). This means that iteration by array elements in the innermost loop is not using the processor cache efficiently. We can fix this by changing the loop order (Figure 9).

Changing the loop order

With this change, access to array elements in the innermost loop will be contiguous in memory, which should increase cache reuse. Let’s recompile and get another performance snapshot with APS (Figure 10). According to the report, we got a speedup of approximately 4.7x by optimizing memory access. The percentage of memory stalls is significantly lower and the application is no longer DRAM-bound. However, the OpenMP Imbalance reported previously in Figure 6, and again in Figure 10, is now the main performance problem.
### Optimization Step 3: Improving Load Balance with OpenMP Dynamic Scheduling

Even though we still have memory stalls, APS draws our attention to the OpenMP Imbalance, which accounts for 30% of elapsed time. OpenMP does static scheduling by default. This is generally an effective, low-overhead scheduling protocol. However, it can sometimes cause load imbalance, so let’s try dynamic scheduling (**Figure 11**) instead.

```c
2  const int num_threads, const int n1_Tblock, const int n2_Tblock,
3     int dimn1n2 = n1*n2;
4  #pragma omp parallel schedule(dynamic) for default(shared)
5  for(int iz=0; iz<n3; iz++) {
6      for(int iy=0; iy<n2; iy++) {
7          for(int ix=0; ix<n1; ix++) {
8              if( ix>=HALF_LENGTH && ix<(n1-HALF_LENGTH) &&
```

**OpenMP dynamic scheduling**
This significantly relieves the load imbalance (Figure 12).

![APS report after applying dynamic scheduling](image)

**Optimization Step 4: Improving Memory Access Using Cache Blocking**

Since cache stalls are still significant (Figure 12), we can try to optimize data reuse in the cache by implementing a well-known optimization called cache blocking (Figure 13).
13 Cache blocking

Please note that to feed OpenMP threads with parallel work, we added the `collapse(3)` clause to our `omp parallel for` pragma so that the three loops arranged for blocking will be collapsed into one large iteration space. The new APS shows a 1.3x improvement with fewer memory stalls and better OpenMP load balance as a result of cache blocking (Figure 14).

### BLOG HIGHLIGHTS

**Opening Doors to Computer Vision & Deep Learning for Developers**

By Charlotte Dryden, Intel Corporation

At the Embedded Vision Summit in Santa Clara this week, I’m excited to meet with fellow innovators and leaders—all working toward the space of bringing data visualization to life.

Computer vision is the science and technology used by machines to see as a human eye would see. This includes all the hardware and software for capturing, processing, analyzing, and understanding digital images....
APS report after cache blocking

Optimization Step 5: Introducing Vectorization

From the APS report, we can see that our compute kernel contains only scalar floating-point instructions (the “% of Scalar FP Instr.” value is 100%). This means that the code was not automatically vectorized by the compiler. Let’s use Intel Advisor to explore opportunities to vectorize the code:

> advixe-cl –collect survey ./Iso3DFD

Opening the report in the GUI and looking at the Survey and Roofline tab, we can see that the tool advises us to apply a SIMD directive to help the compiler vectorize the highlighted loop (Figure 15).
Intel Advisor points to an opportunity for loop vectorization

The OpenMP standard recently added the `omp simd` pragma to help compilers do automatic vectorization when they can't determine if loop iterations are independent. So let's insert the pragma and see if it helps (Figure 16).

```c
for(int iiz=bz; iiz<iEnd; iiz++) {
    for(int iiy=by; iiy<iyEnd; iiy++) {
        float* ptr_next = ptr_next_base + iiz*dimin2 + iiy*n1 + bx;
        float* ptr_prev = ptr_prev_base + iiz*dimin2 + iiy*n1 + bx;
        float* ptr_vel = ptr_vel_base + iiz*dimin2 + iiy*n1 + bx;
        #pragma omp simd
        for(int ix=0; ix<iEnd; ix++) {
            float value = 0.0;
            value += ptr_prev[ix]*coeff[iy];
            for(int ir=1; ir<HALF_LENGTH; ir++) {
                value += coeff[ir] * (ptr_prev[ix + ir] + ptr_prev[ix - ir]);
                value += coeff[ir] * (ptr_prev[ix + i*n1] + ptr_prev[ix - i*n1]);
                value += coeff[ir] * (ptr_prev[ix + ir*dimin2] + ptr_prev[ix - ir*dimin2]);
            }
            ptr_next[iiz] = 2.0f* ptr_prev[ix] - ptr_next[ix] + value*ptr_vel[ix];
        }
    }
}
```

Inserting the `omp simd` pragma
Recompile the application with the `–xHost` compiler option to use the proper vector instruction set. The APS report now shows a 2x performance improvement (Figure 17). Also, we can see that the vector capacity usage is at 50%. As we improved vectorization efficiency, however, we see that there's now high pressure on the memory. We need to bring in much more data than before to feed the vector units, which is why we see memory stalls again.
Threading, Memory, and Vectorization Optimizations

We could try further optimizations (e.g., using the -qopt-zmm-usage=high compiler option), but we've gone from a baseline performance of 1,767 seconds down to 3 seconds with just a few basic code modifications and effective use of Intel Parallel Studio XE, particularly APS. APS was helpful to quickly check optimization progress and identify tuning opportunities. Intel VTune Amplifier's Memory Access analysis and Intel Advisor's Survey and Memory Access Patterns helped with deeper analysis of particular performance aspects.

Learn More

- Intel Parallel Studio XE
- Intel Advisor
- Intel VTune Amplifier
- Intel VTune Amplifier Application Performance Snapshot

Using Intel® Threading Building Blocks in Universal Windows*
Platform Applications
BY NIKITA P., INTEL CORPORATION

The Intel® Threading Building Blocks (Intel® TBB) library provides a set of algorithms to enable parallelism in C++ applications. It is highly portable and supports multiple platforms, including the full spectrum of Windows* devices based on Intel® architecture.

The Intel TBB 2018 release added support for the Universal Windows Platform* (UWP*)—an application platform for the Windows 10 ecosystem that allows developers to create and run apps on all kinds of Windows devices—PC, tablet, phone, Xbox*, HoloLens*, Surface Hub*, and even IoT....
Despite the evolution of computers since the 1960s, hard disks have consistently remained the most conventional and viable way to store large amounts of data. But although they offer large capacity and durability, they have several shortcomings. Hard disks have low bandwidth and high latency, increasing the amount of time the processor has to wait for data to be transferred from disk to DRAM. Also, hard disks store data as a stream of bytes, leading to additional overhead from serialization and deserialization. As a result, disk IO presents a major challenge for many resource-intensive software applications that require data persistence, such as a database.

*IN-PERSISTENT MEMORY COMPUTING WITH JAVA*

How Open-Source Libraries Make it Easy to Integrate Persistent Memory into your Applications

_Eric Kaczmarek, Software Engineering Manager, and Preetika Tyagi, Software Engineer, Intel Corporation_
The ideal solution would be to store the data in memory, but DRAM can't provide sufficient capacity and durability at an affordable cost. Persistent memory answers this challenge by combining the best of both worlds (Figure 1).

![Diagram showing storage versus memory and persistent memory]

**What is Persistent Memory?**
Persistent memory offers memory-like performance while providing durability and storage capacity in the terabyte range. The data remains fully persistent across machine reboots, and also allows direct user-mode access, thereby eliminating the kernel/IO from the data path.

Developers program persistent memory like regular memory, but it persists like storage. The **Persistent Memory Development Kit (PMDK)** is an open-source C/C++ library that can be used to write applications that take advantage of persistent memory. **The Persistent Collections for Java* (PCJ) library** extends its scope to Java-based applications such as Apache Cassandra*, Apache Spark*, and Apache Ignite*, to name a few.
Persistent Collections for Java (PCJ): A Library for Persistent Memory Java Programming

The PCJ library is an open-source pilot project that enables Java developers to design or retrofit their applications around persistent memory without having to worry about disk IO limitations. Figure 2 shows the implementation stack. The PCJ library offers persistent classes that can be used to create persistent objects similar to regular Java objects. These objects are stored on the persistent heap and persist across JVM sessions and machine reboots. These persistent objects have a reachability based lifetime and are not garbage-collected until they become unreachable. Since data is stored directly in an object layout form within persistent memory, no serialization or deserialization is necessary. The PCJ library also exposes APIs for defining customized persistent classes similar to that of regular Java classes. Figure 3 shows a non-exhaustive list of built-in persistent classes in the PCJ library.

![PCJ implementation stack](image)
### Persistent classes

**Primitives types** (as field and array element values, no separate class)

**Boxed primitives** (e.g., PersistentLong)

**PersistentString**

**PersistentByteBuffer**

**PersistentAtomicReference**<T extends AnyPersistent>

**Primitive arrays** (e.g., PersistentByteArray, mutable and immutable)

**PersistentArray**<E extends AnyPersistent> (mutable and immutable)

**PersistentTuple**<T1 extends AnyPersistent, ...> (mutable and immutable)

**PersistentArrayList**<E extends AnyPersistent>

**PersistentHashMap**<K extends AnyPersistent, V extends AnyPersistent>

**PersistentLinkedLIst**<E extends AnyPersistent>

**PersistentLinkedQueue**<E extends AnyPersistent>

**PersistentSkippingLIstMap**<K extends AnyPersistent, V extends AnyPersistent>

**PersistentSkipTreeMap**<K extends AnyPersistent, V extends AnyPersistent>

**PersistentSkipHashMap**<K extends AnyPersistent, V extends AnyPersistent>

**ObjectDirectory** is an indefinitely reachable root map of <String, T extends AnyPersistent>

---

**Programming with PCJ**

**Figure 4** shows a code snippet demonstrating how to use the PCJ library to allocate and store data in persistent memory.
4 Data persistence after reboot

Figure 5 shows a regular Java class, followed by its persistent version (Figure 6).

```java
public final class Employee {
    private final long id;
    private String name;

    public Employee(long id, String name) {
        this.id = id;
        setName(name);
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int hashCode() {
        return Long.hashCode(getId());
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Employee)) return false;
        Employee emp = (Employee) obj;
        return emp.getId() == getId() && emp.getName().equals(getName());
    }

    public String toString() {
        return String.format("Employee (%d,%s)", getId(), getName());
    }
}
```

5 Regular Java class
6 Persistent class

The PCJ library also provides support for transactions. All methods in the PCJ library are transactional—changes to persistent memory happen completely or not at all. If you want to expand the transactional nature to multiple method calls, put them in a transaction (Figure 7).

The PCJ library enables Java developers to write high-performance applications that manipulate large amounts of persistent data in a natural way. While new applications can include PCJ functionality in the design process, existing applications must be retrofitted to use persistent memory.
Transactions with the PCJ

Low-Level Persistent Library (LLPL)

The PCJ library also offers separate low-level access to arbitrary regions of persistent memory, giving Java developers greater flexibility to create their own data abstractions. Figure 8 shows the implementation stack of LLPL. A heap API offers allocation and deallocation of MemoryRegions. The MemoryRegion interface provides setters and getters to access persistent memory. Three kinds of MemoryRegions are provided:

1. **MemoryRegion<Raw>:** Suitable for volatile use or caller-provided data consistency.
2. **MemoryRegion<Flushable>:** Includes fail-safe `flush()` and `isFlushed()` methods.
3. **MemoryRegion<Transactionable>:** Writes are transactional.

LLPL lets developers choose whichever implementation best fits their requirements and also retrofit their existing applications at a low level.

In-Persistent-Memory Enabled Apache Cassandra

From Wikipedia: "Apache Cassandra is a free and open-source distributed NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure." It’s used by several major Web services (Figure 9).
8 LLPL implementation stack

9 Apache Cassandra® users

For more complete information about compiler optimizations, see our Optimization Notice.
Currently, Cassandra uses HD/SSD for data storage. To improve performance, it employs several optimization mechanisms including:

- Caches
- Index/offsets
- Data summary
- Bloom filter

The primary goal of these optimizations is to reduce the number of disk IOs, since a large number of disk IOs can significantly hamper throughput and increase latency under a heavy load. Persistent memory eliminates the disk storage requirement. Intel developed a proof-of-concept version of Cassandra to demonstrate how the overall software stack can be greatly simplified by using the PCJ library. With this library, data can be stored in an object layout format directly in persistent memory.

The existing underlying storage mechanism in Cassandra is based on the log-structured merge-tree (LSM tree) data structure that employs multiple levels to organize the data in the form of key-value pairs. The data at each level is sorted on keys and migrated across these levels using merge sort. Each key has a row associated with it. **Figure 10** demonstrates a simple version of an LSM tree with two layers in the context of Cassandra. The Memtable (an in-memory buffer to hold the data until it’s full and eventually gets flushed to disk) at level-0 is stored in the DRAM, where the data is sorted by keys. Once the Memtable has reached its configurable size threshold, the data is flushed to disk in the form of sorted string tables (SSTables, an on-disk immutable file that gets generated from the Memtable flush) at level-1.
SSTables are immutable by nature, so multiple SSTables can have different versions of a row as a result of multiple writes to the same key. The write operations are efficient in this scheme, since the client is returned a response as soon as the data is written to the Memtable without the need to wait for flushing to the on-disk level-1 storage. However, the gradual increase in the number of SSTables holding multiple versions of the same row can lead to high read latencies, since multiple on-disk SSTables will need to be read to generate the latest, accurate response for a client. Also, it may lead to inefficient disk utilization, since the stale pieces of data continue to reside on the disk due to immutability.

To mitigate the eventual drop in read performance, and to reclaim disk space, Cassandra uses a background process known as Compaction. The primary job of Compaction is to merge one or more SSTables into a single new one (Figure 11). It merges keys by combining columns and retaining the latest copy of the data. It also discards tombstones (a special value written for a key to indicate that it has been deleted), which reduces the overall size of the data in the new SSTable. By recurrently running Compaction, both the read latency and disk capacity are regulated in Cassandra. However, Compaction can still be a CPU-intensive task that can often result in unpredictable latency spikes.
The persistent-memory-enabled Cassandra prototype developed using the PCJ library allows a mutable data structure (PMTable) to be stored in persistent memory. In contrast to the serialized on-disk SSTables, the mutable PMTables can store data directly in the form of objects and fields (Figure 12). This provides several benefits. Because PMTables are mutable, data is written or updated in place and only the latest version of a row is retained—so the latency to read the data remains consistent. Since the data is stored in an object layout format, there is no serialization and deserialization overhead. The keys can be used to look up the data in the PMTable that resides in persistent memory without any disk IO overhead. This scheme also removes the overhead of Compaction, since no data merging is required due to in-place updates. Disk-based optimization features (i.e., bloom filter, index, caches, etc.) are no longer required—which simplifies the read path and delivers more predictable outlier latencies. The current prototype is limited in terms of the features it offers. Intel is working with the Apache community to open-source this prototype and make it fully functional.

In short, the open-source PCJ library provides easy-to-use persistent classes that allow Java developers to integrate persistent memory into their applications.

References and Open-Source Projects

- Persistent Collections for Java (PCJ)
- Persistent Memory Development Kit (PMDK)
- Apache Cassandra
- Introduction to Programming with Persistent Memory from Intel
START CODING YOUR VISION

Quickly and easily integrate computer vision deep learning inference into your apps—from smart cameras and video surveillance to robotics, transportation, and more. And fast-track your workloads from edge to cloud with the new OpenVINO™ toolkit.

FREE DOWNLOAD >
Machine learning (ML) has grown explosively over the last decade. And alongside specific domains such as images or speech recognition, where deep learning thrives, one of the most popular techniques for solving a wide range of real-world ML problems—like regression and classification issues—is gradient boosting. Recently, boosting has been used for page ranking for commercial Web search engines and other winning ML challenge solutions.

To make it easier, Intel® Data Analytics Acceleration Library (Intel® DAAL) has optimized a collection of practical ML algorithms for Intel® processors. The latest version contains gradient-boosted tree classification
and regression algorithms using the stochastic gradient boosting technique. The implementation enhances performance and functionality through multithreading and vectorization.

This article describes, in a nutshell, the gradient boosted trees algorithm and explains why it’s so widely used by ML practitioners. Compared to the popular XGboost* library, Intel DAAL gradient boosting can achieve up to 6.5x better training performance on the same datasets.

**Gradient Boosting and Decision Tree**
Gradient boosting is a powerful ML supervised algorithm used to achieve state-of-the-art accuracy on a variety of tasks like regression, classification, and ranking. The generalized gradient boosting algorithm, introduced by Jerome Friedman¹, uses ensemble or committee methods to reduce the variance of an estimated prediction function. Gradient boosting is typically used with decision trees as base learners. Decision trees are simple, powerful analytical models used in a variety of analytic solutions such as segmentation, regression, and classification. [Editor’s note: You can learn about the advantages and construction and scoring of decision tree models, as well as some of the limitations and ways to overcome them, here.] Ensemble learning is one of the advanced methods to manage the problem of sampling and overfitting errors in decision trees. Like random decision forests, another popular tree ensemble model is gradient-boosted trees. It’s been implemented in many ML software packages including scikit-learn*, Xgboost*, LightGBM*, GBM*, H20*, Spark MLlib*, and OpenCV.*

**Gradient-Boosted Decision Tree Algorithm Details**
As we mentioned above, the gradient-boosted decision trees (GBDT) classification and regression algorithms are an ensemble processing of regression (decision) trees built using the stochastic gradient boosting technique.³

Given \( n \) feature vectors \( X = \{ x_1 = (x_{11}, \ldots, x_{1p}), \ldots, x_n = (x_{n1}, \ldots, x_{np}) \} \) of \( n \) \( p \)-dimensional feature vectors and \( n \) responses \( Y = \{ y_1, \ldots, y_n \} \), the learning process of the algorithm is to build a gradient-boosted trees classification or regression model based on the feature and response data set, and then use the classification and regression model to classify/predict new incoming samples.

The tree ensemble model uses \( M \) additive functions to predict the output \( \hat{y}(x) = f(x) = \sum_{k=1}^m f_k(x) \) \( f_k \in F \) where \( F = \{ f(x) = w(q(x)), q: R^p \to T, w \in T \} \) is the space of regression trees, \( T \) is the number of leaves in the tree, \( w \) is a leaf weights vector, and \( w \) is a score on \( i\)-th leaf. Also, \( q(x) \) represents the structure of each tree that maps an observation to the corresponding leaf index.

For more complete information about compiler optimizations, see our Optimization Notice.
The training procedure is an iterative functional gradient descent algorithm, which minimizes the objective function by iteratively choosing a function (regression tree) that points in the negative gradient direction. The objective function is:

$$L(f) = \sum_{i=1}^{n} l(y_i, f(x_i)) + \sum_{k=0}^{M} \Omega(f_k)$$

where $l(f)$ is twice differentiable convex loss function and $\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|$ is a regularization term that penalizes the complexity of the model defined by the number of leaves $T$ and the L2 norm of the weights $\|w\|$ for each tree. $\gamma$ and $\lambda$ are regularization parameters.

The GBDT algorithm principle has been well explained. The main processing effort is in building the tree. **Figure 1** shows the additive tree structure to determine if a person likes computer games, where the tree's leaf (terminal) nodes represent the corresponding response values (i.e., the prediction result from the tree). (Find more details [here](#).)

1 Tree ensembles[^6]
Training Flow
Let $S = (X, Y)$ be the set of observations. Given the training parameters, such as the number of iterations $M$, loss function $l(f)$, regression tree training parameters, regularization parameters $\gamma$ and $\lambda$, shrinkage (learning rate) parameter $\theta$, the algorithm does the following:

- Find an initial guess $(\hat{y}_i)^{(0)}$, $i = 1, \ldots, n$
- For $k = 1, \ldots, M$:
  - Update $g_i$ and $h_i$, $i = 1, \ldots, n$
  - Grow a regression tree $t_i \in F$ that minimizes objective function
    $$-\frac{1}{2} \sum_{j=1}^{T} \frac{G^2_j}{H_j + \lambda} + \gamma T$$
    where $G_j = \sum_{i \in C_l} g_j$, $H_j = \sum_{i \in C_l} h_j$, $I_j = \{i \mid q(x_i) = j\}$, $j = 1, \ldots, T$
  - Assign an optimal weight $w_j^* = \frac{g_j}{h_j + \lambda}$ to the leaf $j = 1, \ldots, T$
  - Apply shrinkage parameter $\theta$ to the tree leaves and add the tree to the model
  - Update $(\hat{y}_i)^{(k)}$

The algorithm for growing the tree:
- Generate a bootstrap training set if required (stochastic gradient boosting).
- Start from the tree with depth 0.
- For each leaf node in the tree:
  - Choose a subset of features for split finding if required (stochastic gradient boosting)
  - Find the best split that maximizes gain:
    $$\frac{G^2_L}{H_L + \lambda} + \frac{G^2_R}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma$$
- Stop when the termination criteria are met.

Prediction Flow
Once the trees are built in the training phase, we can use them to do predictions for a given set of queries. The classification algorithm computes the sum of responses of all the trees for each class. The class with the maximal response value (highest class probability) determines the prediction. The regression algorithm returns the aggregated result as the final result of given sample.

The GBDT Implementation in Intel® DAAL
The GBDT algorithm is computationally expensive, especially with continuous features and large datasets, but Intel DAAL provides a highly-tuned implementations for classification and regression. For maximum performance, it uses vectorization and multiple levels of parallelization in tree construction and prediction.
The training algorithms implemented in Intel DAAL support two kinds of split calculation modes to build a tree:

1. **Exact**: All possible split values are examined when searching for the best split for a feature.
2. **Inexact**: Continuous features are bucketed into discrete bins and the possible splits are restricted by the bucket borders only. The bucketing allows a reduction in the number of splits to compute on each step, thus making computation faster.

The sample code in **Figure 2** shows how to train a GBDT model with Intel DAAL.

```python
trainDatasetFileName = 'df_classification_train.csv'
nFeatures = 30
nClasses = 6

# Gradient boosted trees parameters
maxIterations = 50
minObservationsInLeafNode = 1

# Model object for the gradient boosted trees classification algorithm
model = None
predictionResult = None
testGroundTruth = None

def trainModel():
    # Initialize FileDataSource<CSVFeatureManager> to retrieve the input data from a .csv file
    trainDataSource = FileDataSource(
        trainDatasetFileName,
        DataSourceIface.notAllocateNumericTable,
        DataSourceIface.doDictionaryFromContext)

    global model

    # Create Numeric Tables for training data and labels
    trainData = HomogenNumericTable(nFeatures, 0, NumericTableIface.notAllocate)
    trainGroundTruth = HomogenNumericTable(1, 0, NumericTableIface.notAllocate)
    mergedData = MergedNumericTable(trainData, trainGroundTruth)

    # Retrieve the data from the input file
    trainDataSource.loadDataBlock(mergedData)

    # Create an algorithm object to train the gradient boosted trees classification model
    algorithm = training.Batch(nClasses)
    algorithm.parameter().maxIterations = maxIterations
    algorithm.parameter().minObservationsInLeafNode = minObservationsInLeafNode
    algorithm.parameter().featuresPerNode = nFeatures

    # Pass the training data set and dependent values to the algorithm
    algorithm.input.set(classifier.training.data, trainData)
    algorithm.input.set(classifier.training.labels, trainGroundTruth)

    # Train the model and retrieve the results of the training algorithm
    trainingResult = algorithm.compute()
    model = trainingResult.get(classifier.training.model)

# Training a GBDT model
```
For training parameters like splitMethod, maxIterations, maxTreeDepth, shrinkage, and their default values, see **Usage Model: Training and Prediction**.

The sample code in **Figure 3** demonstrates how to use the trained GBDT model to make predictions.

```python
testDatasetFileName = 'df_classification_test.csv'

def testModel():
    global testGroundTruth, predictionResult
    # Initialize FileDataSource<CSVFeatureManager> to retrieve the test data from a .csv file
    testDataSource = FileDataSource(
        testDatasetFileName,
        DataSourceIface.notAllocateNumericTable,
        DataSourceIface.doDictionaryFromContext)

    # Create Numeric Tables for testing data and labels
    testData = HomogenNumericTable(nFeatures, 0, NumericTableIface.notAllocate)
    testGroundTruth = HomogenNumericTable(1, 0, NumericTableIface.notAllocate)
    mergedData = MergedNumericTable(testData, testGroundTruth)

    # Retrieve the data from input file
    testDataSource.loadDataBlock(mergedData)

    # Create algorithm objects for gradient boosted trees classification prediction
    algorithm = prediction.Batch(nClasses)

    # Pass the testing data set and trained model to the algorithm
    algorithm.input.setTable(classifier.prediction.data, testData)
    algorithm.input.setModel(classifier.prediction.model, model)

    # Compute prediction results and retrieve algorithm results
    # (Result class from classifier.prediction)
    predictionResult = algorithm.compute()
```

**Using the trained GBDT model to make predictions**

**Performance Considerations and Results**

**XGBoost** is a well-known library designed and optimized for tree boosting. It's been implemented in various ML software packages and provides native interfaces for C++, R, Python*, Julia*, and Java* users. We compared Intel DAAL and XGBoost. Note that this performance comparison represents a snapshot in time. Both packages are being updated. **Table 1** shows the performance advantage of Intel DAAL 2019 beta versus XGBoost 1.6. The results were measured for classification on four different data sets: Higgs, Mnist, Letter, and Isolet from USI ML repository.
The Parallel Universe

- **HIGGS**: Two classification problems to distinguish between a signal process which produces Higgs bosons and a background process which does not. It has 1M samples, each with 28 features.
- **Letter**: A database of character image features. The goal is to identify the 26 English capital letters. It has 16,000 samples, each with 16 features.
- **Isolet**: A simple classification task to predict which letter name was spoken. It has 7,797 samples, each with 617 features.
- **MNIST**: A standard classification task to identify the handwritten numbers. The training set consists of 60,000 images and the test set consists of 10,000 images of handwritten numbers. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in fixed-size images.

The test uses 50 trees with default Intel DAAL parameters (inexact model) and XGBoost (approximate model). The results show that Intel DAAL is, on average, 2.8x faster and up to 6.5x faster for training, and about 1.2x faster for inference.

### Table 1. Comparison of Intel DAAL and XGBoost

<table>
<thead>
<tr>
<th></th>
<th>HIGGS</th>
<th>MNIST</th>
<th>Letter</th>
<th>Isolet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Training(s)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XGBoost</td>
<td>22.6</td>
<td>119.5</td>
<td>3.9</td>
<td>15</td>
</tr>
<tr>
<td>Intel DAAL</td>
<td>10.5</td>
<td>95.7</td>
<td>0.6</td>
<td>9.98</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.15</td>
<td>1.25</td>
<td>6.5</td>
<td>1.5</td>
</tr>
<tr>
<td><strong>Inference (ms)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XGBoost</td>
<td>161</td>
<td>29.7</td>
<td>26.7</td>
<td>24</td>
</tr>
<tr>
<td>Intel DAAL</td>
<td>150</td>
<td>25.7</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.07</td>
<td>1.16</td>
<td>0.86</td>
<td>1.6</td>
</tr>
</tbody>
</table>

Configuration: Intel® Xeon® Silver 4110 processor, 2x8 cores, 2.10 GHz, 32 GB RAM, OS Ubuntu, 16.04.4 LTS; Intel DAAL 2019 beta.

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

Optimization Notice: 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

For more complete information about compiler optimizations, see our Optimization Notice.
Boosting Performance

Intel DAAL contains a collection of ML algorithms optimized for Intel® processors. It recently added gradient-boosted trees classification and regression algorithms to its repertoire. Compared to the popular XGBoost library, Intel DAAL gradient boosting achieved better performance for both model training and inference.

References

2. Kaggle: The Home of Data Science and Machine Learning
4. XGBoost on GitHub
5. Intel® Data Analytics Acceleration Library Decision Forest
An indicator of a well-written HPC application is its ability to scale linearly—or near-linearly—across a large number of discrete nodes. And linear scalability/speedup of applications largely depends on minimizing communication overhead.

One way to minimize communication overhead is by hiding it. Non-blocking communication functions have always been part of the Message Passing Interface* (MPI*) to let developers overlap communication with computation—thus hiding communication latency. Basic interprocess communication in MPI (e.g., MPI_Send, MPI_Recv) is blocking in nature (i.e., calls don’t return until the communication has completed). Non-blocking
functions (e.g., MPI_Isend, MPI_Irecv), on the other hand, return immediately—allowing computation to continue while communication completes in the background.

This article focuses on the relatively new non-blocking collective (NBC) communication functions (e.g., MPI_Iallreduce, MPI_Ibcast) in the MPI-3* standard. Despite the potential goodness of NBC functions, they’re not yet widely used in MPI applications. We’ll try to make them more approachable and show how to use them to improve performance by overlapping communication and computation.

### Understanding Non-Blocking Communication

Using NBC in an MPI code involves more than just replacing a blocking collective function (e.g., replacing MPI_Bcast with its non-blocking counterpart, MPI_Ibcast). In fact, blindly making this change might actually degrade performance due to the additional overhead associated with checking for completion of the communication. Besides replacing MPI_Bcast with MPI_Ibcast, it’s also necessary to carefully examine the code and identify computation that can be executed while the NBC is in progress. The goal is to ensure that the data being computed is independent of the data being communicated. Otherwise, parallel correctness is compromised (Figure 1).

![Example of NBC communication](image)

1. PSEUDO CODE

   ```
   1 ! PSEUDO CODE
   2 MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
   3 ! Overlapping independent compute section
   4 x = a*a + 2*a*b + b*b
   5 y = a*a - 2*a*b + b*b
   6 MPI_WAIT(REQUEST, STATUS, IERROR)
   7 ! More code
   8 z = (a+b)*(a-b)
   ```

   ![NBC Call Returns Immediately (Without Completion) and Execution Continues](image)
   
   ![This Call Returns Upon Completion of the NBC Call in Line 2](image)
   
   ![Execution Continues as Usual](image)

Ideally, it's also important to consider the execution time of the code in the overlapping compute section. The execution time for this section should be equal to or greater than the time required for the encompassing communication (MPI_Ibcast in this case) to justify the additional complexity of non-blocking communication.

The MPI implementation used here is Intel® MPI Library 2018 Update 2 (Intel® MPI), which is fully compliant with the latest MPI 3.1 standard. This library is available standalone or as part of Intel® Parallel Studio XE Cluster Edition.
Meeting the Challenges Associated with Non-Blocking Communication

The semantics of blocking communication ensure completion of the associated data transfer. Non-blocking calls, however, use helper functions like `MPI_Wait` or `MPI_Test` to check for completion. The fact that the non-blocking semantics don't ensure completion presents an opportunity to overlap computation and communication. However, the use of helper functions results in an increase in the number of instructions on the critical path due to cascading overheads at several levels.

Figure 2 shows the software stack used in Intel MPI. CH3 in the abstract device interface layer represents Intel MPI. To put the data to be transferred onto the wire (i.e., the communication medium), several software layers (Netmod and Provider) must be traversed before arriving at the hardware layer.

![Intel® MPI Library 2018 software stack](image)

All the auxiliary instructions that run after invoking a communication call to put data on the wire are referred to as the critical path for the sender. `MPI_Wait` or `MPI_Test` calls originating from the abstract device interface layer complete with the help of additional calls in the Netmod and Provider layer. This is why non-blocking communication is more expensive than blocking communication. However, this expense can be offset by the performance gains from doing computation while communication is in progress.

Another overhead associated with non-blocking communication comes from making it asynchronous. Although asynchronous progress improves communication-computation overlap, it requires an additional thread per MPI rank. This thread consumes CPU cycles and, ideally, must be pinned to an exclusive core. However, exclusive thread pinning for each rank results in half of the cores being assigned just to accelerate the progress of non-blocking MPI calls. Therefore, through careful experimentation, we must select a certain number of cores per node to be assigned for asynchronous progress without causing a considerable compute penalty.

For more complete information about compiler optimizations, see our Optimization Notice.
In Intel MPI, the following environment variables are available to enable asynchronous progress and to pin threads to cores:

```bash
$ export I_MPI_ASYNC_PROGRESS=1
$ export I_MPI_ASYNC_PROGRESS_PIN=<CPU list>
```

Another tuning parameter to be considered is the spin count, which controls the aggressiveness of these threads. This can be controlled using:

```bash
$ export I_MPI_SPIN_COUNT=<scount>
```

The asynchronous approach requires careful consideration and usage. Some alternatives leverage the imbalance in the application to schedule threads on idle cores. A more elegant alternative would be for the network hardware to enable asynchronous progress. When such hardware features become available, it will be much easier to use asynchronous progress to enhance performance.

**Lattice Quantum Chromodynamics (QCD)**

Now let's explore the impact of non-blocking communication on a QCD application. In QCD, space-time is discretized on a four-dimensional hypercube lattice, with fermion (quark) fields ascribed to the lattice sites and gauge (gluon) fields ascribed to the links between sites. The Wilson-Dslash operator is used as the gauge covariant derivative in a variety of lattice Dirac Fermion operators. This operator is like a four-dimensional, nearest-neighbor stencil (a nine-point stencil in 4D), with the added elaboration that the spinors on the sites and the gauges ascribed to the links are represented in component complex matrices. A large proportion of time in QCD applications is spent solving linear systems with these operators, typically using sparse iterative solvers such as conjugate gradient (CG).

The communication pattern for the Wilson-Dslash operator is nearest-neighbor point-to-point (send-receive) exchange. This computation and communication is implemented in the fdopr function (not shown) in Figure 3. The CG solver code has two such invocations to the fdopr function for computing the "red" and "black" data (red-black algorithm). Typical multi-node implementations overlap computation of the Wilson-Dslash operator with the communication of the faces using non-blocking MPI functions (MPI_Isend, MPI_Irecv). The CG solver code also uses the collective communication function, MPI_Allreduce, to compute the error residue of the iterative solver.
The performance of the CG solver is expected to be lower as compared to Wilson-Dslash due to MPI_Allreduce collective operations in the CG solver loop. This is an opportunity to experiment with non-blocking collective communication, where the collective communication after the red computation is overlapped with the black for every iteration of the solver (Figure 4). MPI_Wait is necessary to verify completion of the non-blocking communication. (The optional call to optional MPI_Test is to test for completion of a pending request, and to initiate progress of the non-blocking communication if it hasn’t already begun.) The asynchronous communication is overlapped with the computation (and communication) in function fdopr.

3 Baseline code using MPI_Allreduce (blocking communication)

```fortran
  tstart1 = MPI_Wtime()
call MPI_Allreduce(alphad,ret,1,MPI_DOUBLE_PRECISION,
+                  MPI_SUM, MPI_COMM_WORLD, ierr)
call fdopr(atap,p,ap,0,1)
tend1 = MPI_Wtime()
```

4 Optimized code using MPI_Iallreduce (non-blocking communication)

The performance improvement from overlapping communication and computation using NBC is shown in Figure 5. The application was run on a cluster of dual Intel® Xeon® Gold 6148 processor nodes connected by the Intel® Omni-Path Fabric (Intel® OP Fabric). The improvement is higher for larger node counts, which is expected because the MPI_Allreduce time also increases with node count. Switching to MPI_Iallreduce hides some of this communication latency.
Improving Performance

NBC in Intel MPI provides opportunities to improve performance by overlapping communication and computation. It's important to carefully consider the code segments that lend themselves to this optimization.

References


GET YOUR GAME ON

Analyze hotspots, capture multiple back-to-back frames, isolate performance bottlenecks—and take your game over the top with Intel® Graphics Performance Analyzers.

FREE DOWNLOAD >
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, V-Tune, OpenVINO, 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.