intel Software

THE PARALLEL UNIVERSE

Effectively Train and Execute Machine Learning and Deep Learning Projects on CPUs

Parallelism in Python* Using Numba*

Boosting the Performance of Graph Analytics Workloads

Issue 36 2019

CONTENTS

Letter from the Editor 3
Onward to Exascale
by Henry A. Gabb, Senior Principal Engineer, Intel Corporation

FEATURE
Effectively Train and Execute Machine Learning and Deep Learning Projects on CPUs 5
Meet the Intel-Optimized Frameworks that Make It Easier

Parallelism in Python* Using Numba* 17
It Just Takes a Bit of Practice and the Right Fundamentals

Boosting the Performance of Graph Analytics Workloads 23
Analyzing the Graph Benchmarks on Intel® Xeon® Processors

How Effective is Your Vectorization? 29
Gain Insights into How Well Your Application is Vectorized Using Intel® Advisor

Improving Performance using Vectorization for Particle-in-Cell Codes 37
A Practical Guide

Boost Performance for Hybrid Applications with Multiple Endpoints in Intel® MPI Library 53
Minimal Code Changes Can Help You on the March Toward the Exascale Era

Innovate System and IoT Apps 63
How to Debug, Analyze, and Build Applications More Efficiently Using Intel® System Studio

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

Sign up for future issues

LETTER FROM THE EDITOR

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

Onward to Exascale Computing

As you may know, I'm an old-school HPC guy—not by choice, but out of necessity. High-performance computing (HPC) has a double meaning, depending on who you're talking to. On the one hand, it simply means improving application performance in the generic sense. Some of my friends on the compiler team see a 5% speedup as HPC. And, in their world, they're right. In my world, HPC refers to computing on a grand scale—harnessing thousands of cores to get orders of magnitude speedups. (Think **TOP500**.)

That's why I'm understandably excited about the announcement last month of the Aurora* supercomputer Intel is collaborating on with Argonne National Laboratory. (See **U.S. Department of Energy and Intel to Deliver First Exascale Supercomputer** for the whole story.) Aurora is expected to deliver exaFLOPS performance (i.e., a quintillion, or 10^{18}, floating-point operations per second). Exascale systems will be essential for converged workflows, as we discussed in the **last issue** of *The Parallel Universe*.

Three articles in our current issue touch on optimizations that the push to exascale demands. The Princeton Plasma Physics Laboratory is doing the type of science that will take advantage of an exascale system. Their article, **Improving Performance by Vectorizing Particle-in-Cell Codes**, describes how they fine-tuned one of their critical algorithms. **How Effective Is Your Vectorization?** shows how to take advantage of the information provided by **Intel® Advisor**. **Boost Performance for Hybrid Applications with Multiple Endpoints in Intel® MPI Library** describes enhancements that improve the scalability of applications that combine message passing and multithreading.

That's enough about HPC. What else is in this issue? The feature article, **Effectively Train and Execute Machine Learning and Deep Learning Projects on CPUs**, describes the **Intel® Math Kernel Library for Deep Neural Networks** and how it's used to accelerate AI frameworks. We also have two other articles that data scientists should find interesting: **Parallelism in Python* Using Numba*** and **Boosting the Performance of Graph Analytics Workloads**. The

former provides practical advice on using the Numba compiler to significantly improve the performance of Python numerical kernels. The latter describes the analysis of the **GAP Benchmark Suite**, a common benchmark for graph analytics. Finally, we close this issue with a review of the analysis tools in **Intel® System Studio**: **Innovate System and IoT Apps.**

As always, don't forget to check out **Tech.Decoded**, Intel's knowledge hub for developers, for more on solutions for code modernization, visual computing, data center and cloud computing, data science, and systems and IoT development. And if you haven't already, be sure to **subscribe** to *The Parallel Universe* so you won't miss a thing.

Henry A. Gabb
April 2019

EFFECTIVELY TRAIN AND EXECUTE MACHINE LEARNING AND DEEP LEARNING PROJECTS ON CPUS

Meet the Intel-Optimized Frameworks that Make It Easier

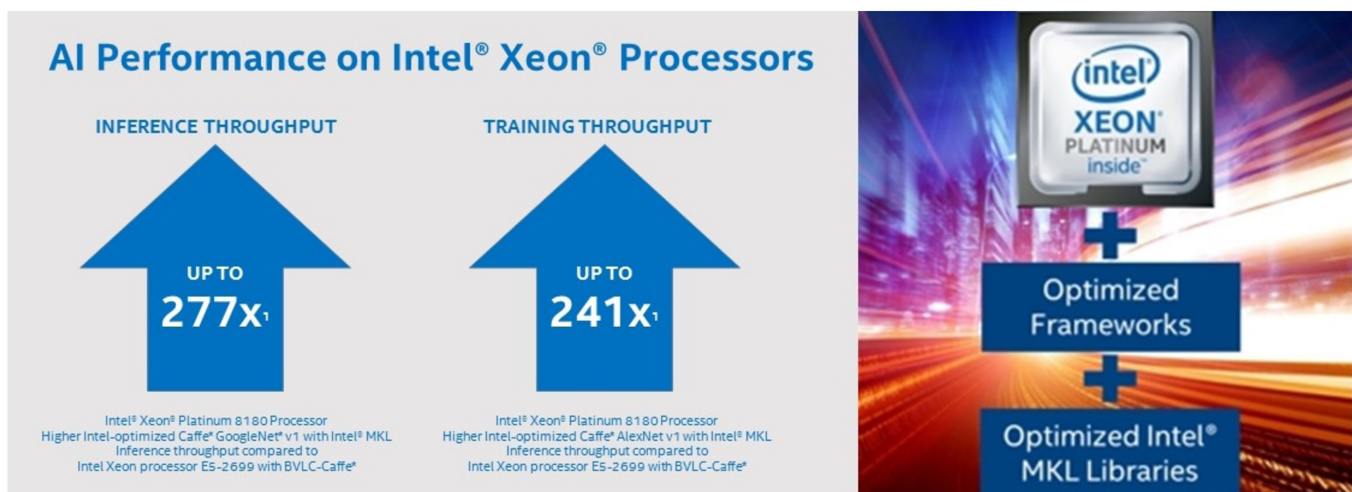
Nathan Greeneltch and Jing Xu, Software Technical Consulting Engineers, Intel Corporation

When you're developing AI applications, you need highly optimized deep learning models that enable an app to run wherever it's needed and on any kind of device—from the edge to the cloud. But optimizing deep learning models for higher performance on CPUs presents a number of challenges, like:

- Code refactoring to take advantage of modern vector instructions
- Use of all available cores
- Cache blocking
- Balanced use of prefetching
- And more

These challenges aren't significantly different from those you see when you're optimizing other performance-sensitive applications—and developers and data scientists can find a wealth of deep learning frameworks to help address them. Intel has developed a number of optimized deep learning primitives that you can use inside these popular deep learning frameworks to ensure you're implementing common building blocks efficiently through libraries like **Intel® Math Kernel Library (Intel® MKL)**.

In this article, we'll look at the performance of Intel's optimizations for frameworks like Caffe*, TensorFlow*, and MXNet*. We'll also introduce the type of accelerations available on these frameworks via the **Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN)** and show you how to acquire and/or build these framework packages with Intel's accelerations—so you can take advantage of accelerated CPU training and inference execution with no code changes (**Figures 1** and **2**).



1 **Deliver significant AI performance with hardware and software optimizations on Intel® Xeon® Scalable processors.**

2 **Boost your deep learning performance on Intel Xeon Scalable processors with Intel® Optimized TensorFlow and Intel MKL-DNN.**

Intel® Math Kernel Library for DNN

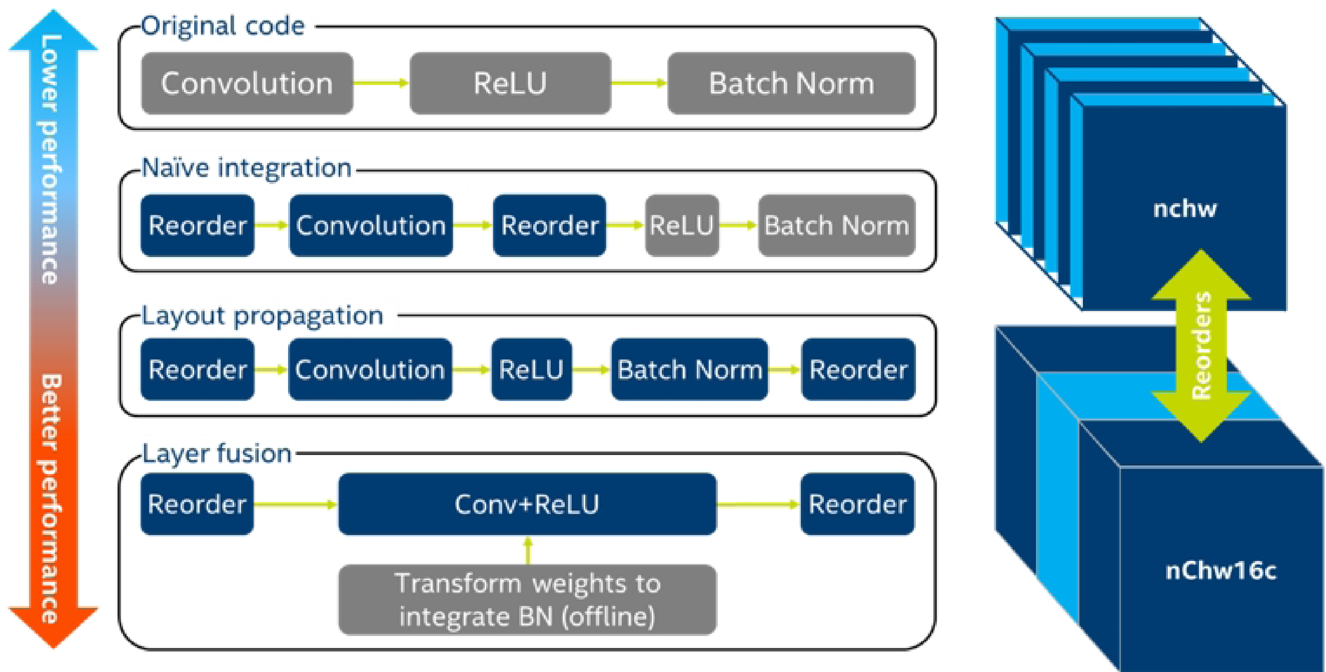
Intel MKL-DNN is an open-source performance library that accelerates deep learning applications and frameworks on Intel® architectures. Intel MKL-DNN contains vectorized and threaded building blocks that you can use to implement deep neural networks (DNN) with C and C++ interfaces (**Table 1**).

The performance benefit from Intel MKL-DNN primitives is tied directly to the level of integration to which the framework developers commit (**Figure 3**). There are reorder penalties for converting input data into Intel MKL-DNN preferred formats, so framework developers benefit from converting once and then staying in Intel MKL-DNN format for as much of the computation as possible.

Also, 2-in-1 and 3-in-1 fused versions of layer primitives are available if a framework developer wants to fully leverage the power of the library. The fused layers allow for Intel MKL-DNN math to run concurrently on downstream layers if the relevant upstream computations are completed for that piece of the data/image frame. A fused primitive will include compute-intensive operations along with bandwidth-limited ops.

Table 1. What's included in Intel MKL-DNN

Function	Features
Compute-intensive operations	• 1D, 2D and 3D spatial convolution and deconvolution • Inner product • General-purpose matrix-matrix multiplication • Recurrent neural network (RNN) cells
Memory-bound operations	• Pooling • Batch normalization • Local response normalization • Activation functions • Sum
Data manipulation	• Reorders/quantization • Concatenation
Primitive fusion	• Convolution with sum and activations
Data types	• fp32 • int8



3 **Performance versus level of integration and Intel MKL-DNN data format visualization**

Installing Intel MKL-DNN

Intel MKL-DNN is distributed in source code form under the Apache* License Version 2.0. See the **Readme** for up-to-date build instructions for Linux*, macOS*, and Windows*.

The `VTUNEROOT` flag is required for integration with **Intel® VTune™ Amplifier**. The Readme explains how to use this flag.

Installing Intel-Optimized Frameworks

Intel® Optimization for TensorFlow*

Current distribution channels are PIP, Anaconda, Docker, and build from source. See the **Intel® Optimization for TensorFlow* Installation Guide** for detailed instructions for all channels.

Anaconda – Linux:

```
conda install -c defaults tensorflow
```

Anaconda – Windows:

```
conda install tensorflow-mkl -c defaults
```

Intel® Optimization for Caffe*

Intel has a tutorial describing how to use **Intel® Optimization for Caffe*** to build Caffe optimized for Intel architecture, train deep network models using one or more compute nodes, and deploy networks.

```
(Ubuntu 16.04)
git clone https://github.com/intel/caffe.git
Open a Terminal window
sudo apt-get update
sudo apt-get install build-essential cmake git pkg-config
sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev
libhdf5-serial-dev protobuf-compiler
sudo apt-get install libatlas-base-dev
sudo apt-get install --no-install-recommends libboost-all-dev
sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev
sudo apt-get install libopencv-dev
Go to Caffe root directory.
cp Makefile.config.example Makefile.config
vi Makefile.config (add the red part)
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include /usr/include/hdf5/serial
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib /usr/lib/x86_64-
linux-gnu /usr/lib/x86_64-linux-gnu/hdf5/serial
make all -j4
```

Intel® Optimization for MXNet*

Intel has **a tutorial** explaining **Intel® Optimization for Apache* MXNet**.

```
git clone --recursive https://github.com/apache/incubator-mxnet.git
cd mxnet && make -j $(nproc) USE_OPENCV=1 USE_BLAS=mkl USE_MKLDNN=1
```

Performance Considerations and Runtime Settings

Now let's consider TensorFlow runtime settings for best performance—specifically, convolutional neural network (CNN) inference. The concepts can be applied to other frameworks accelerated with Intel MKL-DNN and other use cases. However, some empirical testing will be required. Where necessary, we'll give different recommendations for real-time inference (RTI) with batch size of 1 and maximum throughput (MxT) with tunable batch size.

Maximum Throughput versus Real-Time Inference

Deep learning inference is usually done with two different strategies, each with different performance measurements and recommendations:

- **Max Throughput (MxT)** looks to process as many images per second, passing in batches of size > 1. We can achieve the best performance by exercising all the physical cores on a socket. This solution is intuitive in that we simply load up the CPU with as much work as we can, and process as many images as we can, in a parallel and vectorized fashion.
- **Real-time Inference (RTI)** is an altogether different scenario where we want to process a single image as quickly as possible. Here, we aim to avoid penalties from excessive thread launching and orchestration between concurrent processes. The strategy is to confine and execute quickly.

Let's discuss some best-known methods (BKMs) for maximizing MxT and RTI performance.

TensorFlow Runtime Options Affecting Performance

These runtime options heavily affect TensorFlow performance. Understanding them will help you get the best performance out of Intel's optimizations. BKMs differ for MxT and RTI.

These runtime options heavily affect TensorFlow performance. Understanding them will help you get the best performance out of Intel's optimizations. BKMs differ for MxT and RTI. The runtime options are: `{intra|inter}_op_parallelism_threads` and `data layout`.

`{intra|inter}_op_parallelism_threads`

- **Recommended settings (MxT):** `intra_op_parallelism = #physical cores`
- **Recommended settings (RTI):** `intra_op_parallelism = #physical cores`
- **Recommended settings for `inter_op_parallelism`:** 2
- **Usage (shell):** `python script.py --num_intra_threads=cores --num_inter_threads=2 --mkl=True`

intra_op_parallelism_threads and inter_op_parallelism_threads are environment variables defined in tensorflow.ConfigProto. The ConfigProto is used for configuration when creating a session. These two environment variables control number of cores to use.

The intra_op_parallelism_threads environment variable controls parallelism inside an operation. For instance, if matrix multiplication or reduction is intended to be executed in several threads, this environment variable should be set. TensorFlow will schedule tasks in a thread pool which contains intra_op_parallelism_threads threads. OpenMP threads are bound to thread context as closely as possible on different cores. Setting this environment variable to the number of available physical cores is recommended.

The inter_op_parallelism_threads environment variable controls parallelism among independent operations. Since these operations are not relevant to each other, TensorFlow will try to run them concurrently in the thread pool, which contains inter_op_parallelism_threads threads. To minimize effects that will be brought to intra_op_parallelsim_threads threads, this environment variable is recommended to be set to the number of sockets where you want the code to run. For the Intel Optimization of TensorFlow, we recommend keeping the entire execution on a single socket.

Data Layout

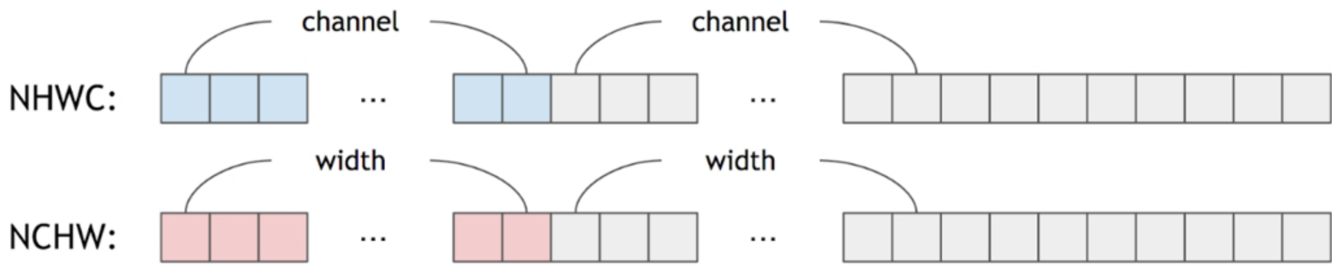
- **Recommended settings:** Data_format = NCHW
- **Usage (shell):**

```
python
script.py --num_intra_threads=cores --num_inter_threads=2 --mkl=True
data_format=NCHW
```

In modern Intel architectures, efficient use of cache and memory greatly impacts overall performance. A good memory access pattern minimizes the performance cost of accessing data in memory. To achieve this, it's important to consider how data is stored and accessed. This is usually referred as data layout. It describes how multidimensional arrays are stored linearly in the memory address space.

In most cases, data layout is represented by four letters for a two-dimensional image.

- **N:** Batch size, indicating number of images in a batch
- **C:** Channel, indicating number of channels in an image
- **W:** Width, indicating number of pixels in horizontal dimension of an image
- **H:** Height, indicating number of pixels in vertical dimension of an image



4 **Data format/layout: NHWC versus NCHW**

The order of these four letters indicates how pixel data are stored in 1-d memory space. For instance, NCHW indicates pixel data are stored in width-wise first, then height-wise, then channel-wise, and finally batch-wise (**Figure 4**). The data is then accessed from left to right with channels-first indexing. NCHW is the recommended data layout for Intel MKL-DNN because this is an efficient layout for the CPU. TensorFlow uses NHWC as the default data layout, but it also supports NCHW.

NUMA Controls Affecting Performance

- **Recommended settings:** `--cpunodebind=0 --membind=0`
- **Usage (shell):**

```
numactl --cpunodebind=0 --membind=0 python
script.py --num_intra_threads=cores --num_inter_threads=2 --mkl=True
data_format=NCHW
```

Running on a NUMA-enabled machine brings with it special considerations. NUMA, or non-uniform memory access, is a memory layout design used in data center machines meant to take advantage of locality of memory in multi-socket machines with multiple memory controllers and blocks. The Intel Optimization for TensorFlow runs best when confining both the execution and memory usage to a single NUMA node.

Intel MKL-DNN Technical Performance Considerations

The library takes advantage of SIMD instructions through vectorization, as well as multiple cores through multithreading. Vectorization effectively utilizes cache and the latest instruction sets. On modern Intel processors, a single core can perform up to two fused multiply and add (FMA) operations on 16 single-precision or 64 int8 numbers per cycle. Moreover, the technique of multi-threading helps in performing multiple independent operations simultaneously. Since deep learning tasks are often independent, getting available cores working in parallel is an obvious choice to boost performance.

To achieve the best possible CPU utilization, Intel MKL-DNN may use hardware-specific buffer layouts for compute-intensive operations, including convolution and inner product. All the other operations will run on the buffers in hardware-specific layouts or common layouts used by frameworks.

Intel MKL-DNN uses OpenMP to express parallelism. OpenMP is controlled by various environment variables: KMP_AFFINITY, KMP_BLOCKTIME, OMP_NUM_THREADS, and KMP_SETTINGS. These environment variables will be described in detail in the following sections. Changing the values of these environment variables affects performance of the framework, so we highly recommend that users tune them for their specific neural network model and platform.

KMP_AFFINITY

- **Recommended settings:**
  `KMP_AFFINITY=granularity=fine,verbose,compact,1,0`
- **Usage (shell):**
  ```
  numactl --cpunodebind=0 --membind=0 python
  script.py --num_intra_  threads=cores --num_inter_threads=2 --mkl=True
  data_format=NCHW --kmp_affinity=granularity=fine,verbose,compact,1,0
  ```

`KMP_AFFINITY` is used to restrict execution of certain threads to a subset of the physical processing units in a multiprocessor computer. Set this environment variable as follows:

```
KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]
```

- **`Modifier`** is a string consisting of a keyword and specifier.
- **`Type`** is a string indicating the thread affinity to use.
- **`Permute`** is a positive integer value that controls which levels are most significant when sorting the machine topology map. The value forces the mappings to make the specified number of most significant levels of the sort the least significant, and then inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.
- **`Offset`** is a positive integer value that indicates the starting position for thread assignment.

We'll use the recommended setting of `KMP_AFFINITY` as an example to explain basic content of this environment variable:

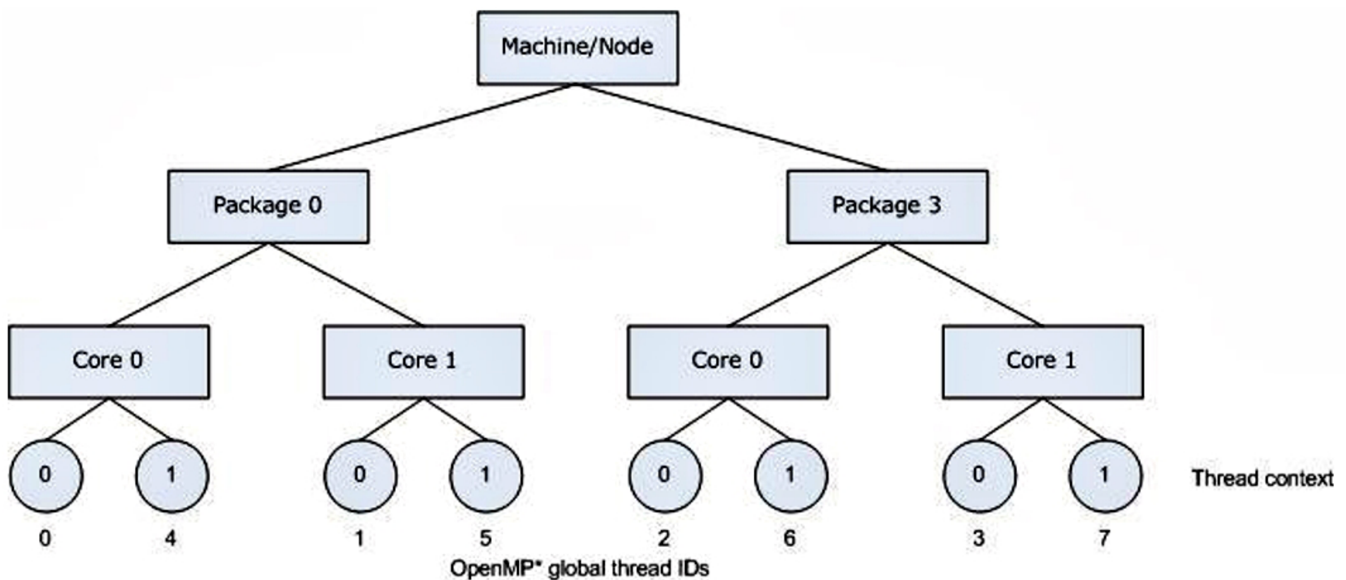
```
KMP_AFFINITY=granularity=fine,verbose,compact,1,0
```

The modifier is `granularity=fine,verbose`. The word `fine` causes each OpenMP thread to be bound to a single thread context, and `verbose` prints messages concerning the supported affinity, e.g.,

- The number of packages
- The number of cores in each package
- The number of thread contexts for each core
- OpenMP thread bindings to physical thread contexts

The word `compact` is the value of type, assigning the OpenMP thread `<n>+1` to a free thread context as close as possible to the thread context where the `<n>` OpenMP thread was placed.

Figure 5 shows the machine topology map when `KMP_AFFINITY` is set to these values. The OpenMP thread `<n>+1` is bound to a thread context as closely as possible to the OpenMP thread `<n>`, but on a different core. Once each core has been assigned an OpenMP thread, the subsequent OpenMP threads are assigned to the available cores in the same order, but they are assigned on different thread contexts.



5 **Machine topology map with the setting `KMP_AFFINITY=granularity=fine,compact,1,0`**

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

Sign up for future issues

The advantage of this setting is that consecutive threads are bound close together so that communication overhead, cache line invalidation overhead, and page thrashing are minimized. It's desirable to avoid binding multiple threads to the same core and leaving other cores not utilized. For more detailed description of `KMP_AFFINITY`, see the **Intel® C++ Compiler Developer Guide and Reference**.

KMP_BLOCKTIME

- **Recommended settings for CNN:** `KMP_BLOCKTIME=0`
- **Recommended settings for non-CNN:** `KMP_BLOCKTIME=1` (user should verify empirically)
- **Usage (shell):**

```
numactl --cpunodebind=0 --membind=0 python
script.py --num_intra_threads=cores --num_inter_threads=2 --mkl=True
data_format=NCHW --kmp_affinity=granularity=fine,verbose,compact,1,0
--kmp_blocktime=0( or 1)
```

This environment variable sets the time, in milliseconds, that a thread should wait after completing the execution of a parallel region before going to sleep. Default value is 200 ms.

After completing the execution of a parallel region, threads wait for new parallel work to become available. After a certain period of time has elapsed, they stop waiting and sleep. Sleeping allows the threads to be used, until more parallel work becomes available, by non-OpenMP threaded code that may execute between parallel regions, or by other applications. A small `KMP_BLOCKTIME` value may offer better overall performance if the application contains non-OpenMP threaded code that executes between parallel regions. A larger `KMP_BLOCKTIME` value may be more appropriate if threads are to be reserved solely for OpenMP execution, but may penalize other concurrently-running OpenMP or threaded applications. The suggested setting is 0 for CNN-based models.

OMP_NUM_THREADS

- **Recommended settings for CNN:** `OMP_NUM_THREADS = # physical cores`
- **Usage (shell):** `Export OMP_NUM_THREADS= # physical cores`

This environment variable sets the maximum number of threads to use in OpenMP parallel regions if no other value is specified in the application. The value can be a single integer, in which case each integer specifies the number of threads for a parallel region at each nesting level. The first position in the list represents the outermost parallel nesting level. The default value is the number of logical processors visible to the operating system on which the program is executed. The recommended value equals the number of physical cores.

KMP_SETTINGS

- **Usage (shell):** `Export KMP_SETTINGS=TRUE`

This environment variable enables (TRUE) or disables (FALSE) the printing of OpenMP runtime library environment settings during program execution.

Learn More

Start using Intel® optimized frameworks to accelerate your deep learning workloads on the CPU today. Check out our helpful resources on **www.intel.ai** and get support from the **Intel® AI Developer Forum**. Also, visit the new **Intel® AI Model Zoo** for solution-oriented resources for your accelerated TensorFlow* projects. Use these resources and you can have confidence that you're using your CPU resources to their fullest capability.

Configuration Note 1

INFERENCE using FP32 Batch Size Caffe GoogleNet v1 128 AlexNet 256.

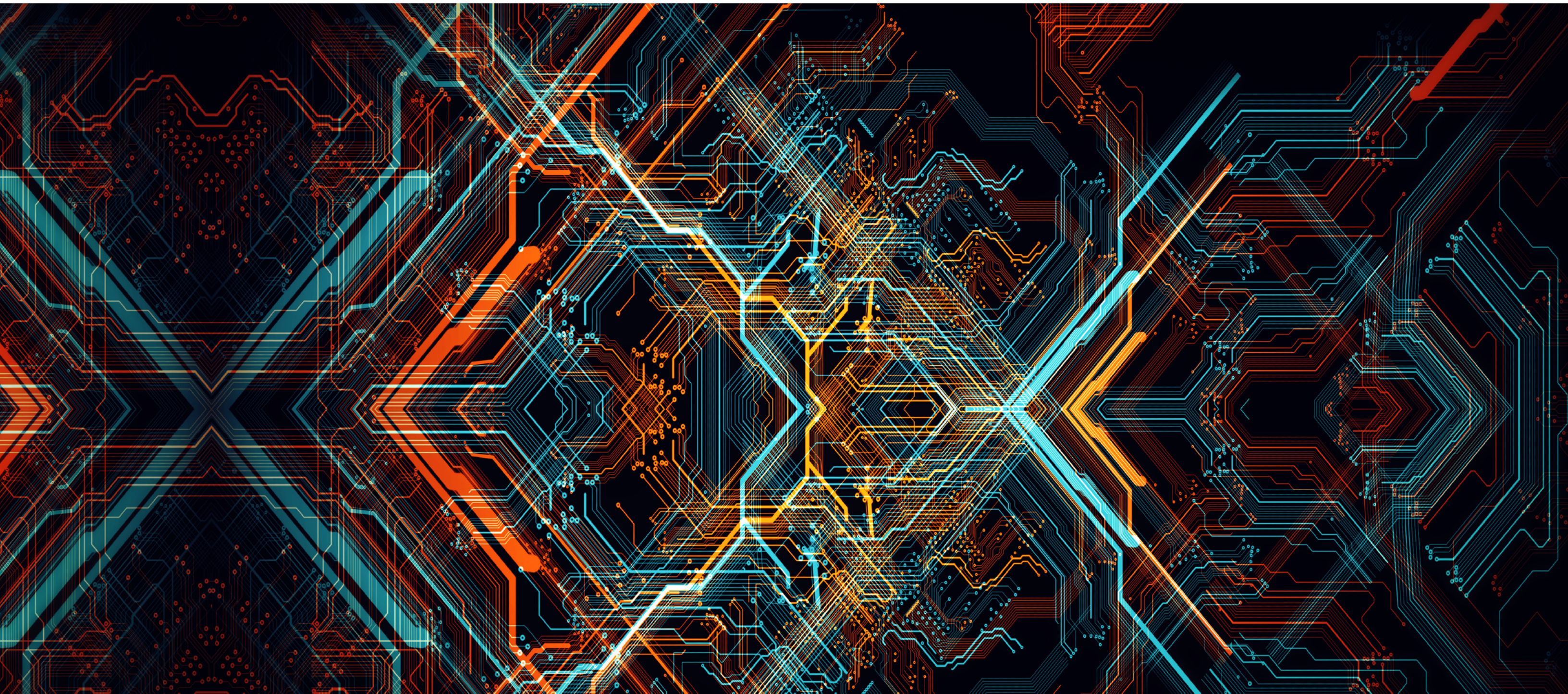
Configurations for Inference throughput: Tested by Intel as of 6/7/2018. :Platform :two-socket Intel® Xeon® Platinum processor, 8180 CPU @ 2.50GHz /28 cores HT ON , Turbo ON. Total Memory: 376.28GB (12slots /32 GB /2666 MHz), four instances of the framework, CentOS Linux*-7.3.1611-Core , SSD sda RS3WC080 HDD 744.1GB,sdb RS3WC080 HDD 1.5TB,sdc RS3WC080 HDD 5.5TB , Deep Learning Framework Caffe version: a3d5b022fe026e9092fc7abc7654b1162ab9940d. Topology:GoogleNet* v1 BIOS:SE5C620.86B.00.01.0004.071220170215 MKLDNN: version: 464c268e544bae26f9b85a2acb9122c766a4c396 NoDataLayer. Measured: 1449 imgs/sec vs Tested by Intel as of 06/15/2018 Platform: 2S Intel® Xeon® processor CPU E5-2699 v3 @ 2.30GHz (18 cores), HT enabled, turbo disabled, scaling governor set to "performance" via intel_pstate driver, 64GB DDR4-2133 ECC RAM. BIOS: SE5C610.86B.01.01.0024.021320181901, CentOS Linux-7.5.1804(Core) kernel 3.10.0-862.3.2.el7.x86_64, SSD sdb INTEL SSDSC2BW24 SSD 223.6GB. Framework BVLC-Caffe: **https://github.com/BVLC/caffe**, Inference & Training measured with "caffe time" command. For "ConvNet" topologies, dummy dataset was used. For other topologies, data was stored on local storage and cached in memory before training. BVLC Caffe (**http://github.com/BVLC/caffe**), revision 2a1c552b66f026c7508d390b526f2495ed3be594.

Configuration for training throughput: Tested by Intel as of 05/29/2018 Platform :2 socket Intel Xeon Platinum processor 8180 CPU @ 2.50GHz / 28 cores HT ON , Turbo ON Total Memory 376.28GB (12slots / 32 GB / 2666 MHz),4 instances of the framework, CentOS Linux-7.3.1611-Core , SSD sda RS3WC080 HDD 744.1GB,sdb RS3WC080 HDD 1.5TB,sdc RS3WC080 HDD 5.5TB , Deep Learning Framework Caffe version: a3d5b022fe026e9092fc7abc765b1162ab9940d Topology:alexnet BI OS:SE5C620.86B.00.01.0004.071220170215 MKLDNN: version: 464c268e544bae26f9b85a2acb9122c766a4c396 NoDataLayer. Measured: 1257 imgs/sec vs. Tested by Intel as of 06/15/2018 Platform: 2S Intel® Xeon® processor CPU E5-2699 v3 @ 2.30GHz (18 cores), HT enabled, turbo disabled, scaling governor set to "performance" via intel_pstate driver, 64GB DDR4-2133 ECC RAM. BIOS: SE5C610.86B.01.01.0024.021320181901, CentOS Linux-7.5.1804 (Core) kernel 3.10.0-862.3.2.el7.x86_64, SSD sdb INTEL SSDSC2BW24 SSD 223.6GB. Framework BVLC-Caffe: **https://github.com/BVLC/caffe**, Inference nd training measured with "caffe time" command. For "ConvNet" topologies, dummy dataset was used. For other topologies, data was stored on local storage and cached in memory before training. BVLC Caffe (**http://github.com/BVLC/caffe**), revision 2a1c552b66f026c7508d390b526f2495ed3be594

Configuration Note 2

System configuration: CPU Thread(s) per core: 2 Core(s) per socket: 28 socket(s): 2 NUMA node(s): 2 CPU family: 6 Model: 85 Model name: Intel Xeon Platinum processor 8180 CPU @ 2.50GHz HyperThreading: ON Turbo: ON Memory 376GB (12 x 32GB) 24 slots, 12 occupied 2666 MHz Disks Intel RS3WC080 x 3 (800GB, 1.6TB, 6TB) BIOS SE5C620.86B.00.01.0004. 070920180847 (microcode version 0x200004d) OS Centos Linux 7.4.1708 (Core) Kernel 3.10.0-693.11.6.el7.x86_64 TensorFlowSource: **https://github.com/tensorflow/tensorflow** commit: 6a0b536a779f485edc25f6a11335b5e640acc8ab MKLDNN version: 4e333787e0d66a1dca1218e99a891d493dbc8ef1 TensorFlow benchmarks: **https://github.com/tensorflow/benchmarks**

INTEL® MATH KERNEL LIBRARY
Fast Math Processing for Intel®-Based Systems
FREE DOWNLOAD



PARALLELISM IN PYTHON* USING NUMBA*

It Just Takes a Bit of Practice and the Right Fundamentals

David Liu, Software Technical Consulting Engineer, Intel Corporation

Obtaining parallelism in Python* has been a challenge for many developers. In **issue 35 of *The Parallel Universe***, we explored the basics of the Python language and the key ways to obtain parallelism. In this article, we'll explore how to achieve parallelism through Numba*.

There are three key ways to efficiently achieve parallelism in Python:

1. **Dispatch to your own native C code** through Python's `ctypes` or `cffi` (wrapping C code in Python).
2. **Rely on a library** that uses advanced native runtimes, such as NumPy or SciPy.
3. **Use a framework** that acts as an engine to generate native-speed code from Python or symbolic math expressions.

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

Sign up for future issues

All three methods escape the global interpreter lock (GIL), and do so in a way that's accepted within the Python community. The Numba framework falls under the third method, because it uses just-in-time (JIT) and low-level virtual machine (LLVM) compilation engines to create native-speed code.

The first requirement for using Numba is that your target code for JIT or LLVM compilation optimization must be enclosed inside a function. After the initial pass of the Python interpreter, which converts to bytecode, Numba will look for the decorator that targets a function for a Numba interpreter pass. Next, it will run the Numba interpreter to generate an intermediate representation (IR). Afterwards, it will generate a context for the target hardware, and then proceed to JIT or LLVM compilation. The Numba IR is changed from a stack machine representation to a register machine representation for better optimization at runtime. From there, the range of options and parallelism directives opens up.

In the following example, we're using pure Python to give Numba the best chance to optimize without having to specify directives:

```
import array
import random
from numba import jit
a = array.array('l', [random.randint(0,10) for x in range
(0,10000000)])

@jit(nopython=True, parallel=True)
def ssum(x):
    total = 0
    for items in x:
        total+=items
    return total

%timeit sum(a)
111 ms ± 861 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

%timeit ssum(a)
4.2 ms ± 108 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

# Nearly 26X faster!
```

Pure CPython bytecode is easier for the Numba interpreter to deal with compared to mixed CPython and NumPy code. The `@jit` decorator tells Numba to create the IR, and then a compiled variant, before running the function. Note the `nopython` attribute on the decorator. This means that we don't want to fall back to stock interpreter behavior if Numba fails to convert the code (more on this later). We used Python arrays instead of lists because they compile better to Numba. We also created a custom summation function because Python's standard sum has special iterator properties that won't compile in Numba.

The previous example works well for general Python. But what if your code requires the use of scientific or numerical packages like NumPy or SciPy? Take, for example, the following code that calculates a resistor-capacitor (RC) time constant for a circuit:

```
import numpy as np
test_voltages = np.random.rand(1,1000)*12
test_constants = np.random.rand(1,1000)

def filter_time_constant(voltage, time_constant):
  return voltage * (1-np.exp(1/time_constant))

%timeit filter_time_constant(test_voltages, test_constants)
11.2 µs ± 145 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In this case, we'll use the `@vectorize` decorator instead of `@jit` because of NumPy's implementation of `ufuncs`:

```
from numba import vectorize
@vectorize
def v_filter_time_constant(voltage, time_constant):
  return voltage * (1-np.exp(1/time_constant))

%timeit v_filter_time_constant(test_voltages, test_constants)
4.74 µs ± 46.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops
each)

# Over 2x faster!
```

When dealing with specialized frameworks such as NumPy and SciPy, Numba is not only dealing with Python, but also with a special type of primitive in the NumPy/SciPy stack called a `ufunc`, which normally means one would need to create a NumPy `ufunc` with C code—a difficult proposition. In this case, the `np.exp()` is a good candidate, since it's a transcendental function and can be targeted by the Intel® Compiler's Short Vector Math Library (SVML) in conjunction with Numba. Both `@vectorize` and `@guvectorize` can use Intel's SVML library and help with NumPy `ufuncs`.

While Numba does have good `ufunc` coverage, it's also important to understand that not every NumPy or SciPy codebase will optimize well in Numba. This is because some NumPy primitives are already highly optimized. For example, `numpy.dot()` uses the Basic Linear Algebra Subroutines (BLAS), an optimized C API for linear algebra. If the Numba interpreter is used, it will actually produce a slower function because it can't optimize the BLAS function any further. To use the `ufunc` optimally in Numba, we'd need to look for a stacked NumPy call, in which many operations to an array or vector are compounded. For example:

```
%timeit np.exp(np.arcsin(np.random.rand(1000)))
19.6 µs ± 85.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops
each)

@jit(nopython=True)
def test_func(size):
  np.exp(np.arcsin(np.random.rand(1000)))

%timeit test_func(1000)
16 µs ± 80.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

The Numba `@jit` performance is slightly better than the straight NumPy code because this computation has not one, but three, NumPy computations. Numba can analyze the `ufuncs` and detect the best vectorization and alignment better than NumPy itself can.

Another area to tweak Numba's compilation directives and performance is using the advanced compilation options. The main options used are `nopython`, `nogil`, `cache`, and `parallel`. With the `@jit` decorator, Numba attempts to choose the best method to optimize the code given to it. However, if the nature of the code is better known, you can directly specify a compilation directive.

The first option is `nopython`, which prevents the compilation from falling back to Python object mode. If the code is unable to convert, it will instead throw an error to the user. The second option is

nogil, which releases the GIL when not processing non-object code. This option assumes you've thought through multithreaded considerations such as consistency and race conditions. The cache option stores the compiled function in a file-based cache to avoid unnecessary compilation the next time Numba is invoked on the same function. The parallel directive is a CPU-tailored transformation to known reliable primitives such as arrays and NumPy computations. This option is a good first choice for kernels that do symbolic math.

Stricter function signatures improve the opportunities for Numba to optimize the code. Defining the expected datatype for each parameter in the signature gives the Numba interpreter the necessary information to find the best machine representation and memory alignment of the kernel. This is similar to providing static types for a C compiler. The following examples show how to provide type information to Numba:

```
@jit(int32(int32, int32))
# Expecting int32 values when being processed

@jit([(int64[:], int64, int64[:])]
# Expecting int64 arrays values when being processed

@vectorize([float64(float64, float64)])
# Expecting float64 values when being processed
```

In general, accessing parallelism in Python with Numba is about knowing a few fundamentals and modifying your workflow to take these methods into account while you're actively coding in Python. Here are the steps in the process:

1. **Ensure the abstraction of your core kernels is appropriate.** Numba requires the optimization target to be in a function. Unnecessarily complex code can cause the Numba compilation to fall back to object code.
2. **Look for places in your code where you see processing data in some form of a loop with a known datatype.** Examples would be a for-loop iterating over a list of integers, or an arithmetic computation that processes an array in pure Python.
3. **If you're using NumPy and SciPy, look at computations that can be stacked in a single statement and that are not BLAS or LAPACK functions.** These are prime candidates for using the ufunc optimization capabilities of Numba.

- **Experiment with Numba's compilation options.**
- **Determine the intended datatype signature of the function and core code.** If it's known (such as int8 or int32), then inform Numba about which input datatype parameters it should expect.

Achieving parallelism with Numba just takes a bit of practice and the right fundamentals. Getting both the performance advantages of stepping out of the GIL while having maintainable code is a testament to the Python community's hard work in the scientific computing space. Numba is one of the best tools to achieve performance and exploit parallelism so it should be in every Python developer's toolkit.

BLOG HIGHLIGHTS

Intel® Graphics Performance Analyzers 2019 R1 Release
GISELLE G., INTEL CORPORATION

Say hello to some of the latest features for the Intel® Graphics Performance Analyzers (Intel® GPA) tool suite.

- **Vulkan Support in Frame Analyzer Stream Capture:** Capture the lifetime of an application through Multiframe Stream capture. Use keyframe functionality to create "save points" that allow for quicker playback time during profiling and can be used to signify rendering anomalies at capture.
- **Edit Shaders:** Quickly iterate over shader code changes, without leaving the application or recompiling code.

Read more >

BOOSTING THE PERFORMANCE OF GRAPH ANALYTICS WORKLOADS

Analyzing the Graph Benchmarks on Intel® Xeon® Processors

Stijn Eyerman, Wim Heirman, and Kristof Du Bois, Research Scientists, and Joshua B. Fryman and Ibrahim Hur, Principal Engineers, Intel Corporation

A graph is an intuitive way of representing a big data set and the relationships between its elements. Each vertex represents an element, and edges connect related elements. Representing data sets as a graph means you can build on the rich history of graph theory. And there are a variety of algorithms to extract useful information from a graph. In this article, we'll explore the implementation characteristics of basic graph analysis algorithms and how they perform on **Intel® Xeon™ processors**.

Graphs and Graph Analytics

A graph is a structured representation of a data set with relationships between elements. Each element is represented as a vertex, and relationships between elements are shown as an edge between two vertices. Both vertices and edges can have attributes representing either the characteristics of the element or the relationship. The vertices to which a vertex is connected through edges are called its neighbors, and the number of neighbors is called the degree of a vertex.

Graph analysis applications extract characteristics from a graph (or multiple graphs) that provide useful information, e.g.:

- Vertices with specific features
- The shortest path between vertices
- Vertex clusters
- Higher-order relationships among vertices

With the growing availability of big data sets and the need to extract useful information from them, graph analytics applications are becoming an important workload, both in the data center and at the edge.

For this study, we'll use the GAP Benchmark Suite[1], a set of common graph algorithms implemented in C++ and optimized by researchers at the University of California at Berkeley for performance on shared-memory multicore processors (**Table 1**). We evaluate GAP performance on an Intel Xeon processor-based server and investigate opportunities to further improve performance.

Table 1. Gap Benchmark Suite overview

Algorithm	Abbreviation	What it Does
PageRank*	pr	Calculates the popularity of a vertex by aggregating the popularity of its neighbors
Triangle counting	tc	Counts the number of triangles (three vertices that are fully connected)
Connected components	cc	Splits the graph into subgraphs with no edges between them
Breadth-first search	bfs	Walks through the graph in breadth-first order
Single-source shortest path	sssp	Calculates the shortest path from one vertex to all others
Betweenness centrality	bc	Calculates the centrality of a vertex, determined by how many shortest paths go through it

Characteristics of Graph Algorithms

Graph algorithms pose challenging behavior for conventional processor architectures. A typical operation is to fetch the attributes of all neighbors of a vertex. The list of neighbors, determined by the topology of the graph, is usually irregular. This leads to sparse memory accesses—accessing individual elements scattered on a large data structure. Sparse memory accesses have no locality, leading to poor cache utilization.

Fetching the attributes of the neighbors means using indirect accesses. For example, if `N` is the array of neighbors of a vertex, and `A` the array containing the attributes, the attribute of neighbor i is accessed by `A[N[i]]`. This pattern is difficult to predict and to vectorize, which leads to an underutilization of the available compute and memory resources.

On the other hand, graph algorithms generally have a lot of parallelism. The set of algorithms in the GAP suite fall into two categories in terms of parallelism. The first category consists of algorithms that operate on all vertices concurrently (`pr`, `tc`, and `cc`). They have abundant parallelism and can be executed across many threads. Their parallelism is only limited by the size of the graph.

The second category is front-based algorithms where, at each iteration, a subset of vertices is analyzed (the current front) and a new front is defined to be processed in the next iteration (`bfs`, `sssp`, and `bc`). These algorithms usually start with a front containing a single vertex. The next front consists of its neighbors, then the neighbors of these neighbors, and so on. In the first iterations, the size of the front (and thus the parallelism that can be exploited) is limited. Also, each iteration ends with a global barrier, which creates additional synchronization overhead. These algorithms scale worse with increasing thread count, especially on smaller graphs.

Running Graph Algorithms on Intel Xeon Processors

Despite the challenging behavior of graph algorithms, there are ways to increase the efficiency of running these applications on a multi-core Intel Xeon processor-based server.

Vectorization

Using vector memory instructions can increase the performance of a graph algorithm by increasing the number of parallel load operations, which hides part of their latency. Specifically, you can use the vector gather instruction (AVX2* and AVX-512*) to perform multiple indirect loads in one instruction. However, the compiler isn't always able to detect these indirect access patterns, or it can decide to not vectorize based on its heuristics. Therefore, it might be useful to add `#pragma` vector `always` to force the compiler to vectorize and/or to rewrite the code to make the indirect access pattern more apparent to the compiler.

Figure 1 gives an example for cc. The original code on the left did not generate vector gather instructions, while the code on the right did. This led to a speedup of 5x for cc on Intel Xeon processors.

```
for (NodeID v : g.out_neigh(u)) {
  NodeID comp_v = comp[v];
  if ((comp_u < comp_v) &&
      (comp_v == comp[comp_v])) {
    change = true;
    comp[comp_v] = comp_u;
  }
}
```

```
NodeID *a = g.out_neigh(u).begin();
#pragma vector always
for (int i=0; i<n; i++){
  NodeID comp_v = comp[a[i]];
  if ((comp_u < comp_v) &&
      (comp_v == comp[comp_v])) {
    change = true;
    comp[comp_v] = comp_u;
  }
}
```

1 Inner loop of connected components. On the left is the original code, which didn't generate vector gather instructions. On the right is the altered code, which makes the indirect pattern clearer and forces the compiler to vectorize.

It might also be useful to look at other vectorization opportunities, and to rewrite the code so that they can be exploited (e.g., using intrinsics). For example, in tc, we need to count the number of matches between the elements of two neighbor lists. Katsov describes an algorithm to speed up the matching algorithm with SSE instructions[2]. We adapted this algorithm to AVX-512 and included this in the tc benchmark, leading to a performance increase of 2.5x (code not included for brevity).[3]

Parallelism

The GAP benchmarks are parallelized using OpenMP*. As discussed before, there are two categories of parallelism: vertex- and front-based. For the vertex-parallel algorithms (pr, cc, and tc), it's important to use dynamic scheduling in the OpenMP parallel for-loops because the processing time of a vertex depends on its neighbor count, which can differ significantly across vertices. With static scheduling, threads that are assigned vertices with many neighbors execute longer than other threads—leading to load imbalance. To reduce the scheduling overhead while still maintaining enough scheduling flexibility, set chunk size to 16 or 32.

The front-based algorithms (bfs, sssp, and bc) are harder to parallelize, which means they don't use the full capacity of the processor (fewer threads than cores). The current front contains many fewer vertices than the full graph, and the next front can only be processed when the current front is finished. To fully exploit the increasing core count of Intel Xeon processor-based servers, these algorithms need to be revised to increase their parallelism.

An example of this is already implemented in `bfs`. Instead of looking for neighbors of the current front and checking whether they've already been visited (forward algorithm), all non-visited vertices are considered, and it is checked whether they are a neighbor of a vertex in the current front (backward algorithm). Because there are more non-visited vertices than vertices in the front, there's more parallelism to exploit. The downside is that the backward algorithm sometimes does unnecessary work when most of the non-visited vertices are not neighbors of the current front.

At each step of the algorithm, we can choose between the two methods. This choice is currently done using the characteristics of the current front and the remaining vertices, but it should also include the available parallelism (core or thread count) to better exploit the capacity of the processor.

Caches and Input Graphs

Graph workloads generally don't generate cache-friendly access patterns. The one exception is when the graph, or the most accessed data structure of the graph (e.g., the attributes of vertices), fits in the last-level cache (which is up to 38 MB per socket on high-end Intel Xeon processors). From our experiments, we notice that performance (expressed in GTEPS, or giga traversed edges per second) decreases with increasing graph size, since less and less of the data fits into the cache. Because of the nature of graphs and graph algorithms, methods to improve cache locality either don't work well or take too much time to reorganize the graph, often more than the algorithm itself.

Distributed Graph Processing

The GAP benchmarks are designed for single-node execution only (using OpenMP parallelization). However, based on the insights from our study, we briefly discuss the impact on distributed graph processing (i.e., using multiple nodes). For high-performance multinode execution, it's crucial to minimize the communication and maximize local data and computation. This is challenging for graph applications because of the irregular and non-localized access pattern. Partitioning a graph to minimize the number of edges between partitions is an NP-complete problem in itself, and often leads to more compute time than the algorithm itself. Therefore, when you're deploying a graph analysis algorithm on multiple nodes, the nodes should be connected by a high-bandwidth, low-latency network such as Intel® Omni-Path Architecture to deal with the unavoidably high amount of communication between the nodes.

Boosting Performance through Analysis

Graph applications form a challenging workload for current processors because of their memory intensiveness and irregularity. By carefully crafting their implementation and exploiting vector units and thread parallelism, we can increase performance significantly. However, more investigation is needed, including redesigning the algorithms to fully exploit the capabilities of an Intel Xeon processor-based server, especially when moving to distributed processing.

References

1. S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP Benchmark Suite," 2015. **http://gap.cs.berkeley.edu/benchmark.html**
2. I. Katsov, "Fast Intersection of Sorted Lists Using SSE Instructions," 2012. **https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/**
3. S. Eyerman et al., "Many-Core Graph Workload Analysis," 2018. **https://dl.acm.org/citation.cfm?id=3291686.**

BLOG HIGHLIGHTS

Improved Parallelization, Extended Deep Learning Capabilities in Intel® Distribution of OpenVINO™ Toolkit

SHUBHA R., INTEL CORPORATION

The latest release of Intel® Distribution of OpenVINO™ toolkit 2019 (which stands for open visual inference and neural network optimization) unveils new features that improve parallelization, extend deep learning capabilities, and provides support for macOS*. Get a quick view of the major new enhancements. Then learn more about new parallelization capabilities that deliver optimal performance for multi-network scenarios.

New Features in 2019 R1

- Supports 2nd generation Intel® Xeon® Processors (codenamed Cascade Lake) and provides performance speedup for inference through Intel® Deep Learning Boost (VNNI instruction set).

Read more >

HOW EFFECTIVE IS YOUR VECTORIZATION?

Gain Insights into How Well Your Application is Vectorized Using Intel® Advisor

Kevin O'Leary, Technical Consulting Engineer, Intel Corporation

Determining how well your application is vectorized is crucial to getting the best performance on your system. In this article, we'll show how to pinpoint vectorization issues, see how well you're using your hardware, and optimize performance using **Intel® Advisor**, which is available in a **free, standalone version** and as part of both **Intel® Parallel Studio XE** and **Intel® System Studio**.

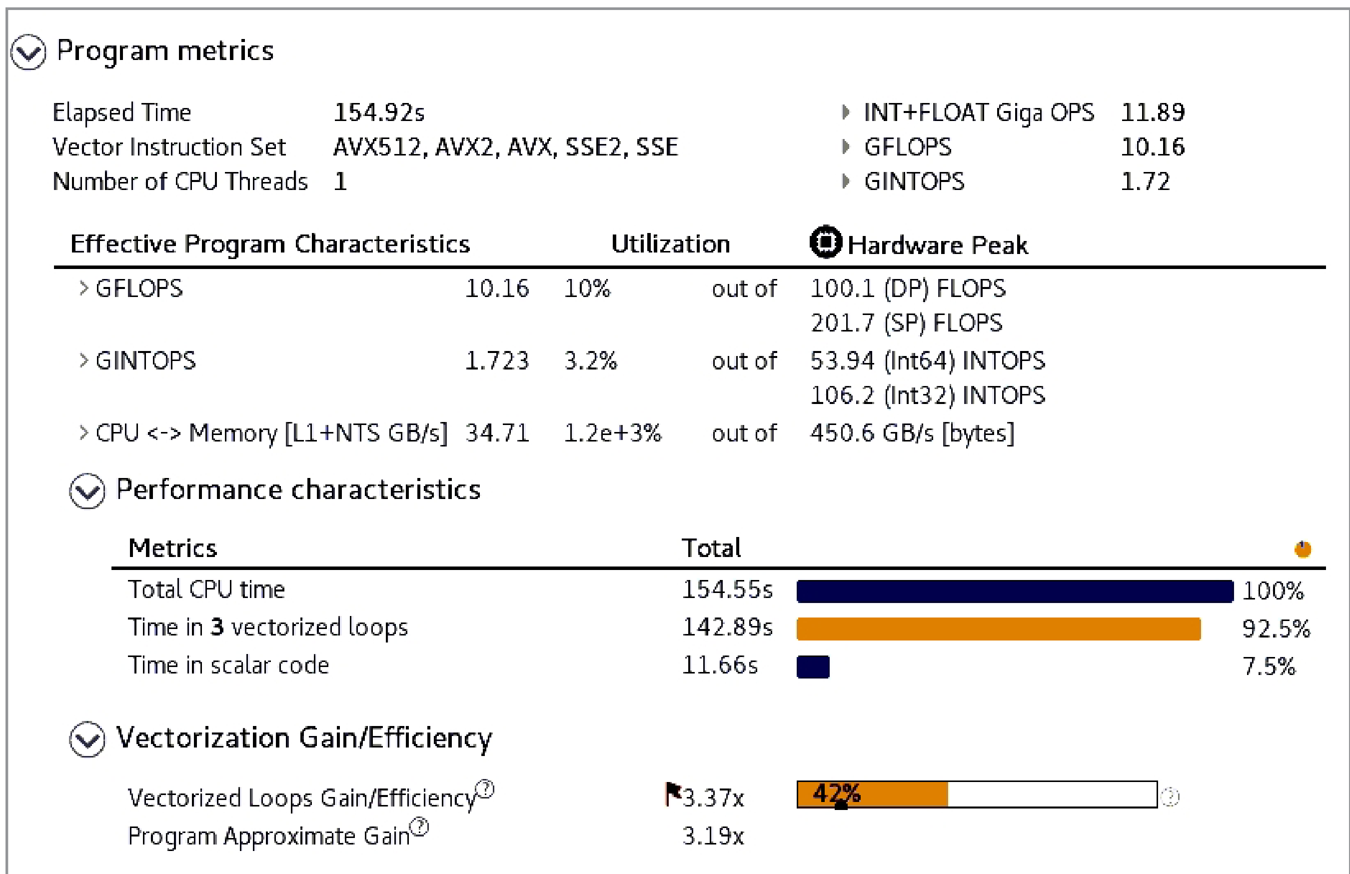
Intel Advisor helps you to see:

- Which loops are vectorized
- Data types, vector widths, and instruction sets (e.g., AVX-512, AVX2)

- How many floating-point and/or integer operations are executed
- How many instructions were devoted to computation and how many to memory operations
- Your register utilization
- How to improve your vectorization
- And much more

Getting Great Performance

To get top performance out of your application, you need information on how well you're using all the resources of the system. Intel Advisor's new and improved summary view (**Figure 1**) gives you an indication of how well the application is performing as a whole.



1 **Intel Advisor summary view**

You can see the vectorization instruction sets used and some useful performance metrics. This view now includes a program characteristics section, which compares your relative performance to the peak performance obtainable on your system. In **Figure 1**, notice that the application is using several different instruction sets—something we should investigate. Also notice that the program is getting vectorization efficiency of just 42%. Where did we lose 58% of our efficiency? We can drill down to investigate.

Drilling Down

You can get more detail in the survey and roofline tab (**Figure 2**). The survey view gives details on a loop-by-loop basis. Focus on the loops where you're spending the most time, and try to get these loops to vectorize as efficiently as possible. Intel Advisor highlights whether the loop is vectorized and its efficiency. If the compiler wasn't able to vectorize the loop, Intel Advisor can tell you why. The performance issues column can give you clues as to why efficiency is poor.

Summary | Survey & Roofline | Refinement Reports

ROOFLINE

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?	Vectorized Loops	
		Self Time	Total Time			Vecto ...	Efficiency
[loop in matvec at Multiply.c:69]	**1 Ineffective pe...**	**119.010s**	**119.010s**	**Vectorized (Body ...**	**1 vectorization possibl...**	**AVX5 ...**	**16%**
[loop in matvec at Multiply.c:60]	1 Misaligned lo ...	29.500s	29.500s	Vectorized (Body)		AVX2	~100%
[loop in matvec at Multiply.c:82]	1 Misaligned lo ...	22.610s	22.610s	Vectorized (Body)		AVX2	71%
[loop in matvec at Multiply.c:49]		11.720s	182.840s	Scalar Versions	1 inner loop was already ..		

2 Survey and rooftop tab

Instruction Set Analysis

Instruction set analysis (**Figure 3**) takes a deep dive into what the compiler did to vectorize your code. It shows the:

- Vectorization instruction set used
- Vector widths
- Data type being operated on

The traits column generally indicates the memory manipulation the compiler had to do to fit your data structure into a vector. These memory manipulations can be indicators of poor efficiency.

Function Call Sites and Loops	Traits	Data Ty...	Numb...	Vector Widths	Instruction Sets
	Instruction Set Analysis				
[loop in matvec at Multiply.c:69]	**FMA; Inserts; Unpacks**	**Float32**	**12; 2**	**128/256**	**AVX; AVX512F_256**
[loop in matvec at Multiply.c:60]	FMA	**Float32**		256	AVX; FMA
[loop in matvec at Multiply.c:82]	FMA	**Float32**		256	AVX; FMA
[loop in matvec at Multiply.c:49]	Extracts; Shuffles	Float32; ...		128/256	AVX; AVX2; AVX512F_256
matvec	Extracts; FMA; Inserts; Shu...	Float32; ...		128/256	AVX; AVX2; AVX512F_256; FMA

3 Instruction set analysis

In our example application, the main loop is using **Intel® AVX-512**, but the vector widths are only 128 and 256. Also, Intel Advisor gives you a warning message if your application seems to be underperforming, and offers tuning advice (**Figure 4**).

⚠ **Your application might be underperforming**

Your application might be underperforming due to disabled zmm registers by default. To define a level of zmm registers, use the **-qopt-zmm-usage** option. Refer to Compiler Guide for more information.

4 Warning message

Recompiling to enable the ZMM registers yields the instruction set analysis in **Figure 5**. Most of our loops now use the complete 512 bytes of the vector registers. In our example, using the ZMM registers improved performance. However, this isn't always the case. It's application-specific.

Function Call Sites and Loops	Traits	Data Ty...	Vector Widths	Instruction Sets
	Instruction Set Analysis			
[loop in matvec at Multiply.c:69]	FMA; Inserts; Unpacks	Float32	128/256	AVX; AVX512F_128; AVX512F_256
[loop in matvec at Multiply.c:49]	Extracts; FMA; Gathers; Ma...	Float32; ...	128/256/512	AVX; AVX512F_256; AVX512F_512
[loop in matvec at Multiply.c:60]	FMA	**Float32**	512	AVX512F_512
[loop in matvec at Multiply.c:82]	FMA	**Float32**; I..	512	AVX512F_512
[loop in main at Driver.c:155]	Type Conversions	Float32; ...		

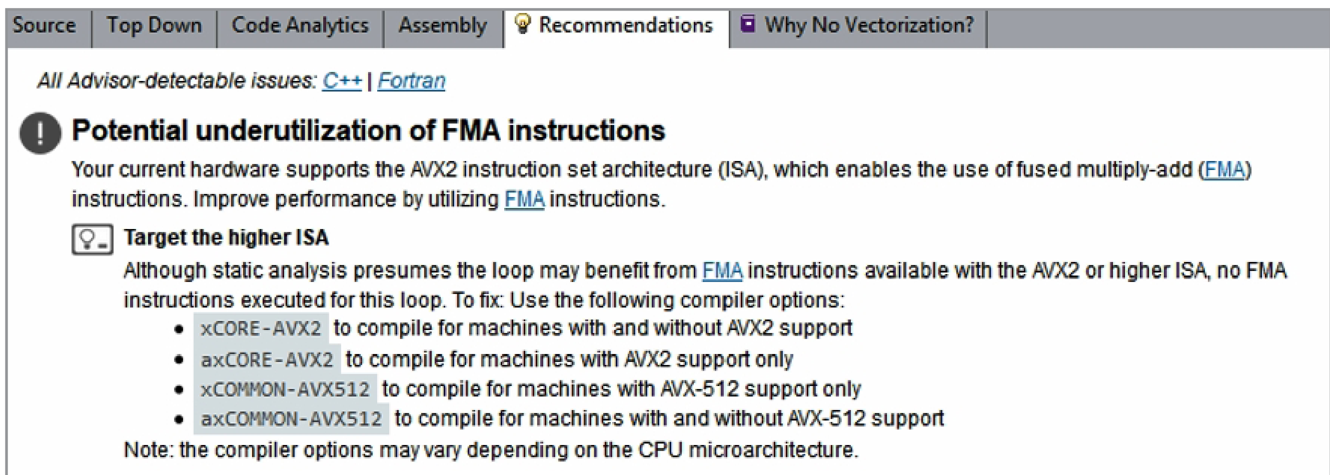
5 Instruction set analysis

Using the Middle Part of the Intel Advisor GUI

The tabs in the middle of the Intel Advisor GUI contain a wealth of program information (**Figure 6**).

6 Intel Advisor GUI tabs

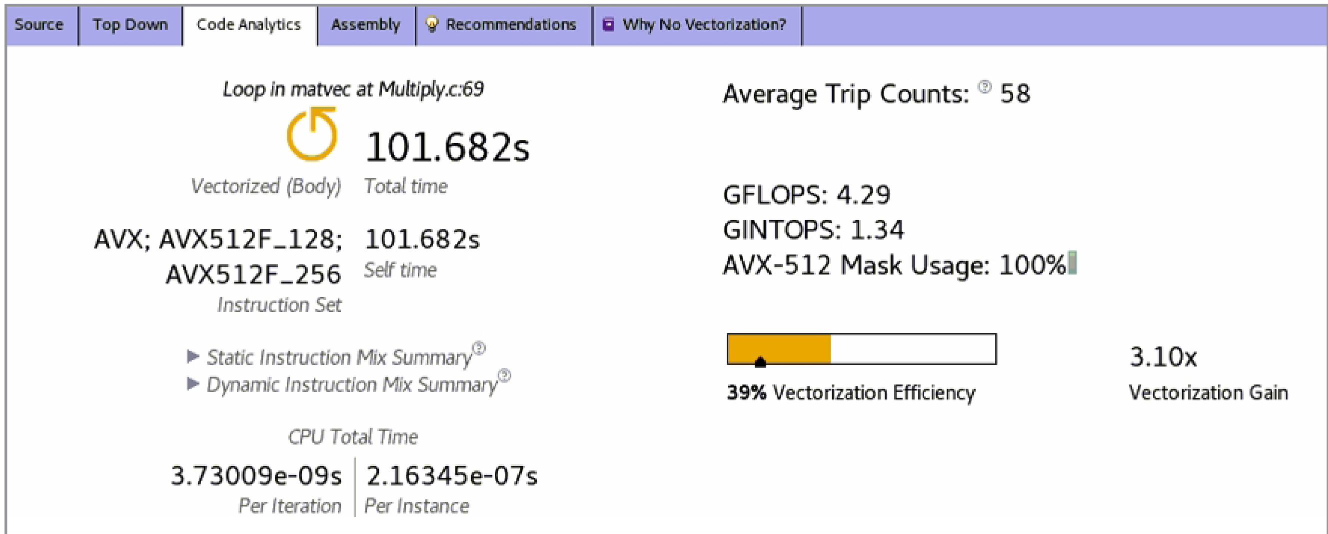
The recommendations tab is a great way to get tips to improve performance (**Figure 7**). For instance, if a loop didn't vectorize, the vectorization tab can tell you why, along with providing code examples showing how to fix the issue.



7 Intel Advisor recommendations tab

Code Analytics

The code analytics tab (**Figure 8**) gives details about what's happening in a loop. You can see your performance at a high level or get statistics for all operations and an instruction mix summary.



8 **Intel Advisor code analytics tab**

Statistics for All Operations

You can get statistics for all operations, including floating-point (FLOPS), integer (INTOP), or mixed (INT+FLOAT) operations (**Figure 9**). This gives you a detailed view of some key performance metrics, showing how many instructions are executing per second. This view also gives you metrics on how well you're using the memory hierarchy in this loop.

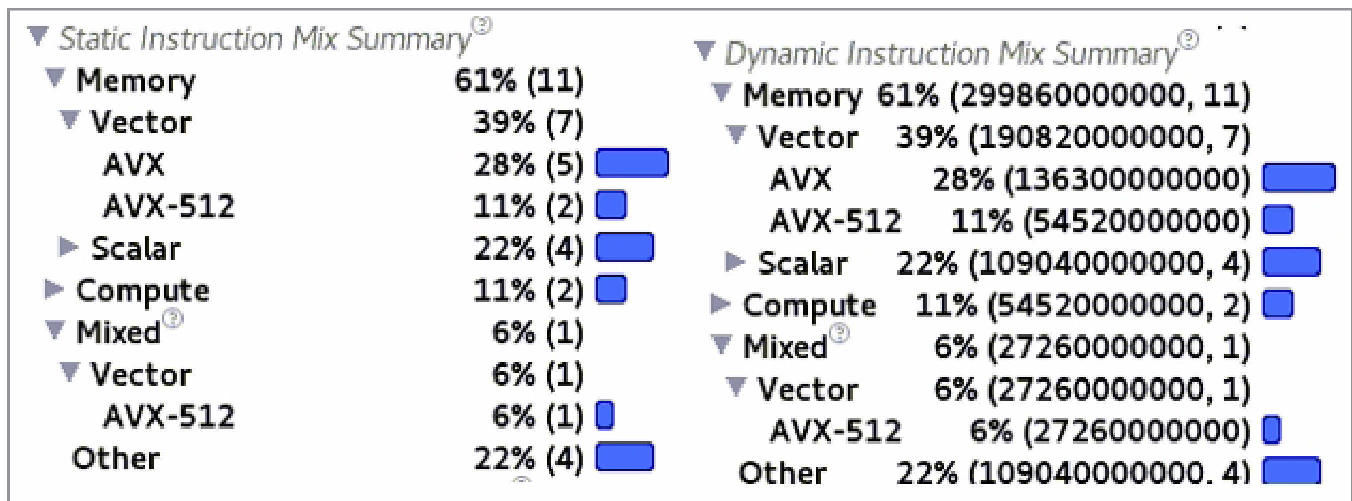
Statistics for All Operations
And Data Transfers — Self / Total

	Per loop	Per Iteration	Per Instance
GFLOP	4.36e+02	1.60e-08	9.28e-07
GFLOPS		4.29	
AI		0.25	
Mask Utilization		100%	
GINTOP	1.36e+02	5.00e-09	2.90e-07
GINTOPS		1.34	
INT AI		0.08	
Mask Utilization		-	
INT+FLOAT OP	5.72e+02	2.10e-08	1.22e-06
INT+FLOAT OPS		5.63	
INT+FLOAT AI		0.33	
Mask Utilization		100%	
L1 Gb	1.74e+03	6.40e-08	3.71e-06
L1 Gb/s		17.16	
Elapsed Time	1.02e+02s	3.73e-09s	2.16e-07s

9 **Statistics for all operations**

How Many Operations Are You Executing?

What are the types of instructions in your loop? Are they compute- or memory-based? Intel Advisor can answer these questions, and give you both the static and dynamic instruction count, with the static instruction mix summary (**Figure 10**). You get the percentage of each instruction you're executing, so you can see if you're really using the newest instructions where you should be.



10 **Static instruction mix summary**

Optimizing Vectorization

It's crucial to optimize the vectorization of your program. Understanding how well your program is vectorized by using a tool like Intel Advisor can help you make sure you're getting the most out of your hardware.

Related Articles

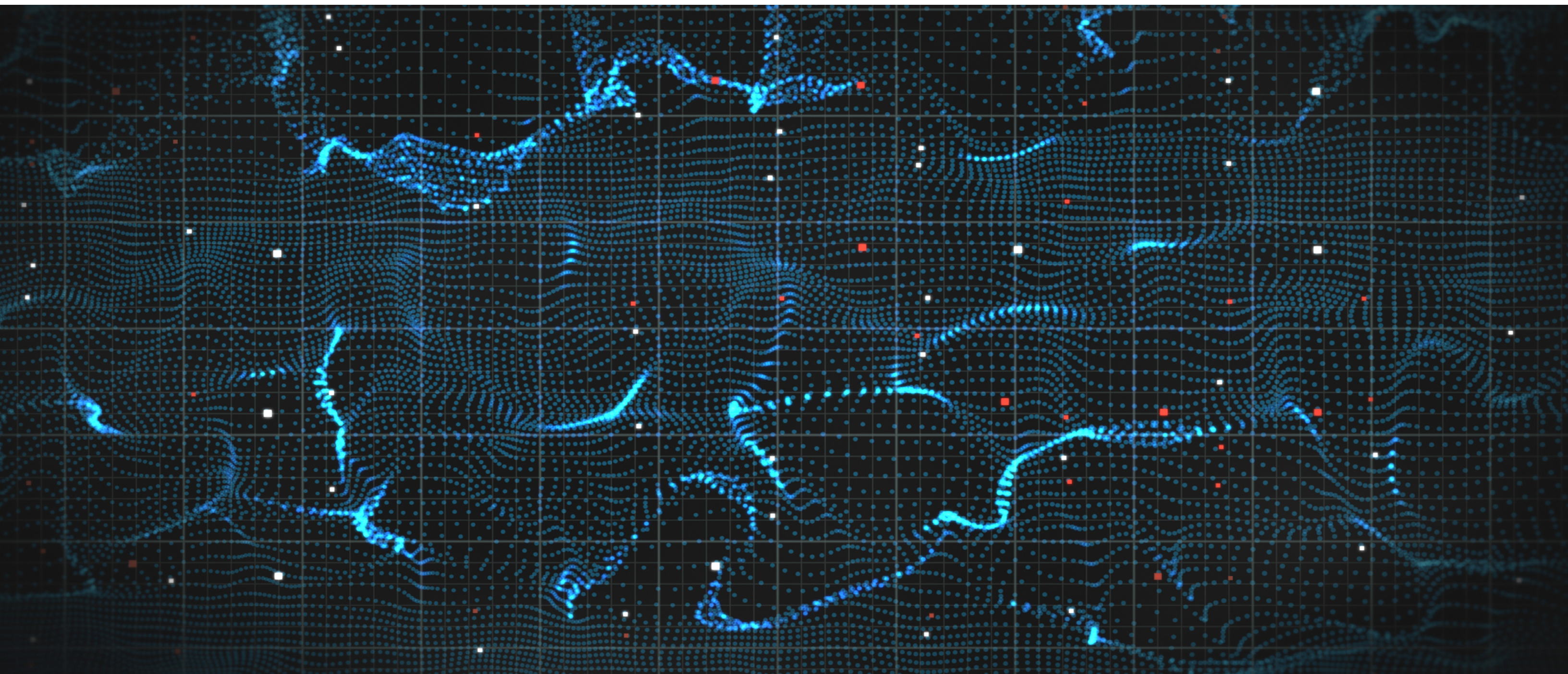
- **Intel Advisor Roofline**
- **Intel Advisor Integer Roofline**
- **Intel Advisor Integrated Roofline**

TEACH YOUR CODE TO BE SMARTER

Download free Intel® Performance Libraries and start creating better, more reliable, and faster applications now.

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



IMPROVING PERFORMANCE BY VECTORIZING PARTICLE-IN-CELL CODES

A Practical Guide

Bei Wang, HPC Software Engineer, Princeton University; Carlos Rosales-Fernandez, Software Technical Consulting Engineer, Intel Corporation; and William Tang, Professor, Princeton Plasma Physics Laboratory

The basic particle method is a well-established approach to simulating the behavior of charged particles interacting with each other through pairwise electromagnetic forces. At each step, the particle properties are updated according to these calculated forces. For applications on powerful modern supercomputers with deep cache hierarchies, a pure particle method is efficient with respect to both locality and arithmetic intensity (compute-bound).

Unfortunately, the $O(N^2)$ complexity makes a particle method impractical for plasma simulations using millions of particles per process. Instead of calculating $O(N^2)$ forces, the particle-in-cell (PIC) method uses a grid as the medium to calculate long-range electromagnetic forces. This reduces the complexity from

O(N^2) to O(N + M log M), where M is the number of grid points, generally much smaller than N. However, achieving high parallel and architectural efficiency is a significant challenge for PIC methods due to the gather/scatter nature of the algorithm.

Attaining performance becomes even more complex as HPC technology moves to the era of multi- and many-core architectures with increased thread and vector parallelism on shared memory processors. A deep understanding of how to improve the associated scalability will have a wide-ranging influence on numerous physical applications that use particle-mesh algorithms—including molecular dynamics, cosmology, accelerator physics, and plasma physics.

This article is a practical guide to improving performance by enabling vectorization for PIC codes.

Optimization for PIC Codes

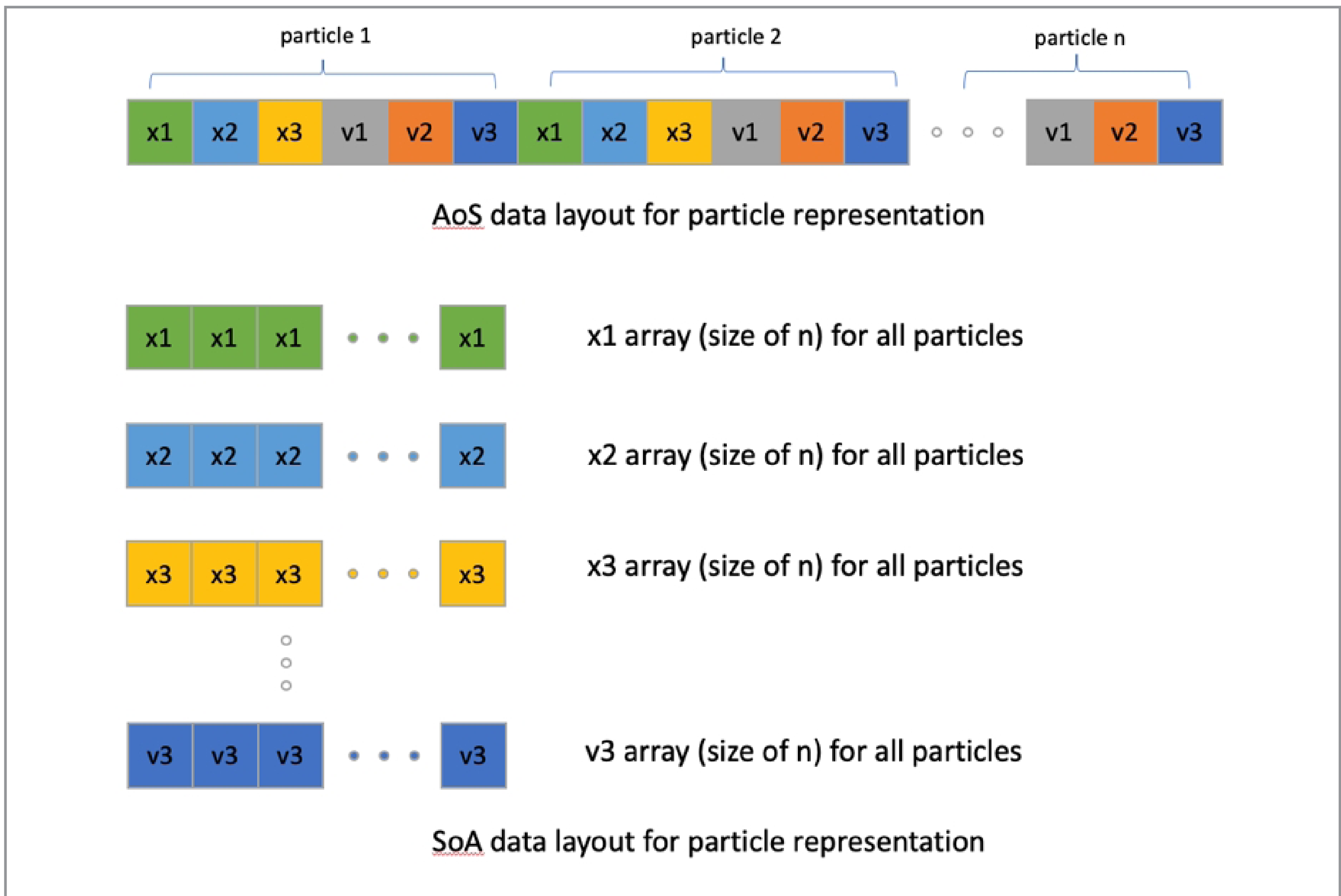
The code example we use for this demonstration is the particle class in Athena++, an astrophysical magnetohydrodynamics (MHD) code written in C++ [1]. The particle class encapsulates basic data structures and functions in PIC methods. Particle properties are represented by phase space position—that is, physical space position (`x1,x2,x3`) and velocity (`v1,v2,v3`). The class functions implement three essential particle-based operations in PIC methods:

1. **Deposit:** Charge deposition from particles onto grid
2. **Move:** Interpolation of grid-based fields onto particles and updating of particle properties using the fields
3. **Shift:** Move particles among processes in the distributed environment. Usually, deposit and move take 80 to 90% of the total computational time and are the focus for optimizations.

Data Layout and Alignment

Particle properties can be stored as array-of-structures (AoS) or structure-of-arrays (SoA) data layout (**Figure 1**).

AoS helps to pack and unpack particles for shift, but doesn't help to enable vectorization for `deposit` and `move` with stride-one memory access. Since `deposit` and `move` are the hotspots of the code, we choose SoA data layout for particle representation. Specifically, we allocate the memory with alignment using the `posix_memalign()` function (**Figure 2**).



1 Particle representation using AoS and SoA layout

```
posix_memalign((void **)&x1, CACHELINE_BYTES, max_prtl*sizeof(Real));
posix_memalign((void **)&x2, CACHELINE_BYTES, max_prtl*sizeof(Real));
posix_memalign((void **)&x3, CACHELINE_BYTES, max_prtl*sizeof(Real));
posix_memalign((void **)&v1, CACHELINE_BYTES, max_prtl*sizeof(Real));
posix_memalign((void **)&v2, CACHELINE_BYTES, max_prtl*sizeof(Real));
posix_memalign((void **)&v3, CACHELINE_BYTES, max_prtl*sizeof(Real));
```

2 Allocating the memory with alignment using the `posix_memalign()` function

We start the optimization by checking the vectorization report of the original code for `deposit.cpp` and `move.cpp` (**Figure 3**).

```
67     for (long p=0; p<nparticle; p++)
68     {
69       Real x1tmp = x1p[p];
70       Real x2tmp = x2p[p];
71       Real x3tmp = x3p[p];
72       Real v1tmp = v1p[p];
73       Real v2tmp = v2p[p];
74       Real v3tmp = v3p[p];
75
76       Real a = (x1tmp - x1s) * dx1 + isg;
77       int ig = (int)(a);
78       int is = ig - 1;
79       Real d = a - ig;
80       Real wei1[3];
81       wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
82       wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
83       wei1[2] = 0.5 * d * d;
84
85       a = (x2tmp - x2s) * dx2 + jsg;
86       ig = (int)(a);
87       int js = ig - 1;
88       d = a - ig;
89       Real wei2[3];
90       wei2[0] = 0.5 * (1.0 - d) * (1.0 - d);
91       wei2[1] = 0.75 - (d - 0.5) * (d - 0.5);
92       wei2[2] = 0.5 * d * d;
93
94       a = (x3tmp - x3s) * dx3 + ksg;
95       ig = (int)(a);
96       int ks = ig - 1;
97       d = a - ig;
98       Real wei3[3];
99       wei3[0] = 0.5 * (1.0 - d) * (1.0 - d);
100      wei3[1] = 0.75 - (d - 0.5) * (d - 0.5);
101      wei3[2] = 0.5 * d * d;
102
103
104      for (int k0=0; k0<=2; k0++){
105        for (int j0=0; j0<=2; j0++){
106          for (int i0=0; i0<=2; i0++){
107            Real w = wei3[k0] * wei2[j0] * wei1[i0];
108            //#pragma omp ordered simd
109            mcoup(IDN,ks+k0,js+j0,is+i0) += w;
110            //#pragma omp ordered simd
111            mcoup(IM1,ks+k0,js+j0,is+i0) += w * v1tmp;
112            //#pragma omp ordered simd
113            mcoup(IM2,ks+k0,js+j0,is+i0) += w * v2tmp;
114            //#pragma omp ordered simd
115            mcoup(IM3,ks+k0,js+j0,is+i0) += w * v3tmp;
116      }}}
117    }
```

```
76     for (long p=0; p<nparticle; p++)
77     {
78       Real x1tmp = x1p[p];
79       Real x2tmp = x2p[p];
80       Real x3tmp = x3p[p];
81       Real v1tmp = v1p[p];
82       Real v2tmp = v2p[p];
83       Real v3tmp = v3p[p];
84
85       Real x1n = x1tmp + v1tmp * halfdt;
86       Real x2n = x2tmp + v2tmp * halfdt;
87       Real x3n = x3tmp + v3tmp * halfdt;
88
89       Real a = (x1tmp - x1s) * dx1 + isg;
90       int ig = (int)(a);
91       int is = ig - 1;
92       Real d = a - ig;
93       Real wei1[3];
94       wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
95       wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
96       wei1[2] = 0.5 * d * d;
97
98       a = (x2tmp - x2s) * dx2 + jsg;
99       ig = (int)(a);
100      int js = ig - 1;
101      d = a - ig;
102      Real wei2[3];
103      wei2[0] = 0.5 * (1.0 - d) * (1.0 - d);
104      wei2[1] = 0.75 - (d - 0.5) * (d - 0.5);
105      wei2[2] = 0.5 * d * d;
106
107      a = (x3tmp - x3s) * dx3 + ksg;
108      ig = (int)(a);
109      int ks = ig - 1;
110      d = a - ig;
111      Real wei3[3];
112      wei3[0] = 0.5 * (1.0 - d) * (1.0 - d);
113      wei3[1] = 0.75 - (d - 0.5) * (d - 0.5);
114      wei3[2] = 0.5 * d * d;
115
116      Real bfld1 = 0.0; Real bfld2 = 0.0; Real bfld3 = 0.0;
117      Real efld1 = 0.0; Real efld2 = 0.0; Real efld3 = 0.0;
118      Real totwei=0.0; Real eb=0.0;
119      for (int k0=0; k0<=2; k0++){
120        for (int j0=0; j0<=2; j0++){
121          for (int i0=0; i0<=2; i0++){
122            Real w = wei3[k0] * wei2[j0] * wei1[i0];
123            totwei += w;
124            bfld1 += w * fcoup(IB1,ks+k0,js+j0,is+i0);
125            bfld2 += w * fcoup(IB2,ks+k0,js+j0,is+i0);
126            bfld3 += w * fcoup(IB3,ks+k0,js+j0,is+i0);
127            efld1 += w * fcoup(IE1,ks+k0,js+j0,is+i0);
128            efld2 += w * fcoup(IE2,ks+k0,js+j0,is+i0);
129            efld3 += w * fcoup(IE3,ks+k0,js+j0,is+i0);
130            eb    += w * fcoup(IEB,ks+k0,js+j0,is+i0);
131          }
132        }
133      }
135      Real bsq   = std::max(bfld1 * bfld1 + bfld2 * bfld2 + bfld3 * bfld3, TINY_NUMBER);
136      Real edotb = efld1 * bfld1 + efld2 * bfld2 + efld3 * bfld3;
137      Real diff = (eb*totwei - edotb)/bsq;
138      efld1 += diff*bfld1;
139      efld2 += diff*bfld2;
140      efld3 += diff*bfld3;
141      totwei = qom/totwei * halfdt;
142
143      bfld1 *= totwei; bfld2 *= totwei; bfld3 *= totwei;
144      efld1 *= totwei; efld2 *= totwei; efld3 *= totwei;
145
146      Real v1n = v1tmp + efld1;
147      Real v2n = v2tmp + efld2;
148      Real v3n = v3tmp + efld3;
149
150      Real vp1 = v1n + v2n * bfld3 - v3n * bfld2;
151      Real vp2 = v2n + v3n * bfld1 - v1n * bfld3;
152      Real vp3 = v3n + v1n * bfld2 - v2n * bfld1;
153
154      Real b = 1.0 + bsq * totwei * totwei; // 1 -> 1.0 avoid type conversion
155      b = 2.0 / b; // reduce the divident call from 3 to 1
156      Real s1 = bfld1*b;
157      Real s2 = bfld2*b;
158      Real s3 = bfld3*b;
159
160      v1tmp = v1n + efld1 + vp2 * s3 - vp3 * s2;
161      v2tmp = v2n + efld2 + vp3 * s1 - vp1 * s3;
162      v3tmp = v3n + efld3 + vp1 * s2 - vp2 * s1;
163      x1p[p] = x1n + v1tmp * halfdt;
164      x2p[p] = x2n + v2tmp * halfdt;
165      x3p[p] = x3n + v3tmp * halfdt;
166      v1p[p] = v1tmp;
167      v2p[p] = v2tmp;
168      v3p[p] = v3tmp;
169    }
```

3 **Original code for `deposit.cpp` (left) and `move.cpp` (right)**

We compile the code with the Intel® C++ Compiler 19.0 and the following options: `-O3 -g -qopt -report5 -xCORE-AVX512 -qopenmp simd`. By default, the compiler will try to vectorize the innermost loop for both `deposit` (L106) and `move` (L121). From the vectorization report, we see that the compiler fails to do so because of a potential data dependency in `deposit.cpp` (**Figure 4**) and efficiency reasons in `move.cpp` (**Figure 5**).

```
LOOP BEGIN at src/particle/deposit.cpp(67,3)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed ANTI dependence between x1p[p] (69:18) and mcoup.pdata_[?-1+i0+mcoup
   remark #15346: vector dependence: assumed FLOW dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+j0+mcoup

   LOOP BEGIN at src/particle/deposit.cpp(104,5)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
      remark #15346: vector dependence: assumed OUTPUT dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+j0+
      remark #15346: vector dependence: assumed OUTPUT dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+j0+

      LOOP BEGIN at src/particle/deposit.cpp(105,7)
         remark #15344: loop was not vectorized: vector dependence prevents vectorization
         remark #15346: vector dependence: assumed OUTPUT dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+
         remark #15346: vector dependence: assumed OUTPUT dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1+

         LOOP BEGIN at src/particle/deposit.cpp(106,9)
            remark #15344: loop was not vectorized: vector dependence prevents vectorization
            remark #15346: vector dependence: assumed ANTI dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1
            remark #15346: vector dependence: assumed FLOW dependence between mcoup.pdata_[?-1+i0+mcoup.nx1_*(?-1
```

4 **Vectorization report for `deposit.cpp`**

```
LOOP BEGIN at src/particle/move.cpp(76,3)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at src/particle/move.cpp(119,5)
      remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [ src/particle/move.cpp(120,7) ]

      LOOP BEGIN at src/particle/move.cpp(120,7)
         remark #15541: outer loop was not auto-vectorized: consider using SIMD directive   [ src/particle/move.cpp(121,9)

         LOOP BEGIN at src/particle/move.cpp(121,9)
            remark #15389: vectorization support: reference wei1[i0] has unaligned access   [ src/particle/move.cpp(122,42)
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0))] h
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcou
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcou
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcou
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcou
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcou
            remark #15389: vectorization support: reference fcoup.pdata_[?-1+i0+fcoup.nx1_*(?-1+j0+fcoup.nx2_*(ig-1+k0+fcou
            remark #15381: vectorization support: unaligned access used inside loop body
            remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector always directi
            remark #15305: vectorization support: vector length 4
            remark #15427: loop was completely unrolled
            remark #15309: vectorization support: normalized vectorization overhead 1.832
            remark #15456: masked unaligned unit stride loads: 8
            remark #15475: --- begin vector cost summary ---
            remark #15476: scalar cost: 61
            remark #15477: vector cost: 25.250
            remark #15478: estimated potential speedup: 0.630
            remark #15488: --- end vector cost summary ---
         LOOP END
```

5 **Vectorization report for `move.cpp`**

Explicit SIMD via OpenMP Directives

For both `deposit` and `move`, the loop count for the outermost loop (e.g., `nparticle` loop) is large, while the loop count for the innermost loop is small. Therefore, the outermost loop is a more suitable candidate for vectorization. Our first attempt is to enable vectorization for the outermost loop in both `deposit` and `move` using OpenMP* SIMD directives. For both kernels, there's no data dependency with respect to pointer aliasing. In considering vectorization for `deposit` in the outermost loop, a potential data dependency will appear when two particles within a single vector length try to write on the same memory location for the grid-based array (e.g., `mcoup`). When compiling for AVX2, it's important to use `#pragma omp ordered simd` such that the loop can be safely vectorized. Since AVX512* provides conflict detection instructions (`vpconflict`), this is no longer necessary for AVX512. Also, we use `aligned` and `simdlen` clauses in the OpenMP SIMD directives to have the compiler generate more efficient vector instructions, e.g.:

```
#pragma omp simd aligned(x1p,x2p,x3p,v1p,v2p,v3p:64) simdlen(SIMD_WIDTH)
  for (long p=0; p<nparticle; p++)
  {
    Real x1tmp = x1p[p];
    Real x2tmp = x2p[p];
    Real x3tmp = x3p[p];
    Real v1tmp = v1p[p];
    Real v2tmp = v2p[p];
    Real v3tmp = v3p[p];
```

Note the current Intel C++ Compiler only allows a local variable name in the `aligned` clause list. Using a class data member name directly will cause a compiler error. Our solution here is to define local variables and assign class data member variables to local variables, e.g.:

```
Real *__restrict__ x1p = x1;
Real *__restrict__ x2p = x2;
Real *__restrict__ x3p = x3;
Real *__restrict__ v1p = v1;
Real *__restrict__ v2p = v2;
Real *__restrict__ v3p = v3;
```

When we check the vectorization report after adding the OpenMP SIMD directives, we see that the compiler has now successfully vectorized the outermost loop with aligned access to the particle data (**Figure 6**). Unfortunately, the vectorized version is quite inefficient and doesn't give a speedup over the scalar version due to the gather/scatter nature of the kernels. For example, the estimated potential speedup is only 0.55 and 0.67 for `deposit` and `move`, respectively.

```
LOOP BEGIN at src/particle/deposit.cpp(67,3)
   remark #15388: vectorization support: reference x1p[p] has aligned access
   remark #15388: vectorization support: reference x2p[p] has aligned access
   remark #15388: vectorization support: reference x3p[p] has aligned access
   remark #15388: vectorization support: reference v1p[p] has aligned access
   remark #15388: vectorization support: reference v2p[p] has aligned access
   remark #15388: vectorization support: reference v3p[p] has aligned access
  remark #15305: vectorization support: vector length 8
  remark #15309: vectorization support: normalized vectorization overhead 0.097
  remark #26012: vectorization support: data layout of a private variable wei1 was optimized, converted to SoA
  remark #26012: vectorization support: data layout of a private variable wei2 was optimized, converted to SoA
  remark #26012: vectorization support: data layout of a private variable wei3 was optimized, converted to SoA
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 6
  remark #15450: unmasked unaligned unit stride loads: 27
  remark #15451: unmasked unaligned unit stride stores: 9
  remark #15462: unmasked indexed (or gather) loads: 36
  remark #15463: unmasked indexed (or scatter) stores: 36
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 552
  remark #15477: vector cost: 960.500
  remark #15478: estimated potential speedup: 0.550
  remark #15487: type converts: 6
  remark #15488: --- end vector cost summary ---
```

6 Vectorization report for loops in `deposit` (after adding OpenMP SIMD directives)

Strip Mining

Strip mining is a common technique to help the compiler automatically vectorize code by exposing data parallelism[2]. The idea is to transform a single loop into a nested loop where the outer loop strides through a strip and the inner loop strides all iterations within the strip (**Figure 7**).

Usually, the strip size is a multiple of the vector length. In some computationally-intensive loops, the strip mining technique alone can lead to a significant performance boost. For `deposit` and `move`, replacing the particle loop with two nested loops doesn't help with performance due to the gather/scatter. After applying strip mining, for example, the estimated potential speedup is still only 0.52 and 0.67 for `deposit` and `move`, respectively (**Figure 8**).

```
  for (long p=0; p<nparticle; p+=SIMD_WIDTH)
    {
      int remains = nparticle - p;
      Real *__restrict__ x1p = x1+p;
      Real *__restrict__ x2p = x2+p;
      Real *__restrict__ x3p = x3+p;
      Real *__restrict__ v1p = v1+p;
      Real *__restrict__ v2p = v2+p;
      Real *__restrict__ v3p = v3+p;

#pragma omp simd aligned(x1p,x2p,x3p,v1p,v2p,v3p:64) simdlen(SIMD_WIDTH)
      for (int pp=0; pp<std::min(SIMD_WIDTH, remains); pp++) {

        Real x1tmp = x1p[pp];
        Real x2tmp = x2p[pp];
        Real x3tmp = x3p[pp];
        Real v1tmp = v1p[pp];
        Real v2tmp = v2p[pp];
        Real v3tmp = v3p[pp];
```

7 Strip mining for `particle` loop

```
remark #15305: vectorization support: vector length 8
remark #15309: vectorization support: normalized vectorization overhead 0.096
remark #26012: vectorization support: data layout of a private variable wei1 was optimized, converted to SoA
remark #26012: vectorization support: data layout of a private variable wei2 was optimized, converted to SoA
remark #26012: vectorization support: data layout of a private variable wei3 was optimized, converted to SoA
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 6
remark #15462: unmasked indexed (or gather) loads: 36
remark #15463: unmasked indexed (or scatter) stores: 36
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 515
remark #15477: vector cost: 942.500
remark #15478: estimated potential speedup: 0.520
remark #15487: type converts: 6
remark #15488: --- end vector cost summary ---
remark #15305: vectorization support: vector length 8
remark #15309: vectorization support: normalized vectorization overhead 0.079
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 6
remark #15449: unmasked aligned unit stride stores: 6
remark #15462: unmasked indexed (or gather) loads: 189
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 2089
remark #15477: vector cost: 2959.870
remark #15478: estimated potential speedup: 0.670
remark #15486: divides: 3
remark #15487: type converts: 6
remark #15488: --- end vector cost summary ---
```

8 Vectorization report for `deposit` and `move` after applying strip mining

However, strip mining can be a powerful technique when you use it as a basis for other techniques, such as loop fission.

Loop Fission for `move`

The `move` loop includes two operations:

1. **The interpolation** of grid-based fields onto particles (gather)
2. **Updating** of particle properties according to the interpolated fields

While the former won't benefit from vector instructions, the latter will. The idea here is to use loop fission to separate the non-vectorizable operation from the vectorizable one to improve performance. At the same time, by applying strip mining with loop fission, we can significantly reduce the storage requirement for passing local variables from the first loop to the second. **Figure 9** shows the new move implementation.

The vectorization report shows that although there is no potential speedup for the first loop, the second achieves a 6.59 estimated potential speedup (**Figure 10**).

Vectorizable Charge Deposition Algorithm for `deposit`

Compared with `move`, `deposit` is more challenging to optimize. For the scatter operation, the performance challenges involve not only random memory accesses and potential data conflicts, but also increased pressure on the memory subsystem. Also, unlike `move`, which also includes many independent computationally-intensive operations (i.e., updating particles properties), `deposit` is mostly memory operations. A key optimization here is to reduce (or even avoid) data conflicts and regularize memory accesses. Motivated by the portable SIMD charge deposition algorithm in the PICSAR* code[3], we implemented a vectorizable charge deposition algorithm for `deposit` (**Figure 11**).

```
50     Real bfld1v[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
51     Real bfld2v[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
52     Real bfld3v[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
53     Real efld1v[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
54     Real efld2v[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
55     Real efld3v[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
56     Real ebv[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
57     Real totweiv[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
78     for (long p=0; p<nparticle; p+=SIMD_WIDTH)
79       {
80         int remains = nparticle - p;
81         Real *__restrict__ x1p = x1+p;
82         Real *__restrict__ x2p = x2+p;
83         Real *__restrict__ x3p = x3+p;
84         Real *__restrict__ v1p = v1+p;
85         Real *__restrict__ v2p = v2+p;
86         Real *__restrict__ v3p = v3+p;
87
88   #pragma omp simd aligned(x1p,x2p,x3p,v1p,v2p,v3p:64) simdlen(SIMD_WIDTH)
89         for (int pp=0; pp<std::min(SIMD_WIDTH,remains); pp++) {
90
91           Real x1tmp = x1p[pp];
92           Real x2tmp = x2p[pp];
93           Real x3tmp = x3p[pp];
94           Real v1tmp = v1p[pp];
95           Real v2tmp = v2p[pp];
96           Real v3tmp = v3p[pp];
97
98           Real x1n = x1tmp + v1tmp * halfdt;
99           Real x2n = x2tmp + v2tmp * halfdt;
100          Real x3n = x3tmp + v3tmp * halfdt;
101
102          Real a = (x1tmp - x1s) * dx1 + isg;
103          int ig = (int)(a);
104          int is = ig - 1;
105          Real d = a - ig;
106          Real wei1[3];
107          wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
108          wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
109          wei1[2] = 0.5 * d * d;
```

9 New `move` implementation with strip mining and loop fission (part 1)

```
129         bfld1v[pp] = 0.0; bfld2v[pp] = 0.0; bfld3v[pp] = 0.0;
130         efld1v[pp] = 0.0; efld2v[pp] = 0.0; efld3v[pp] = 0.0;
131         ebv[pp] = 0.0; totweiv[pp] = 0.0;
132         for (int k0=0; k0<=2; k0++){
133           for (int j0=0; j0<=2; j0++){
134             for (int i0=0; i0<=2; i0++){
135               Real w = wei3[k0] * wei2[j0] * wei1[i0];
136               totweiv[pp] += w;
137               bfld1v[pp] += w * fcoup(IB1,ks+k0,js+j0,is+i0);
138               bfld2v[pp] += w * fcoup(IB2,ks+k0,js+j0,is+i0);
139               bfld3v[pp] += w * fcoup(IB3,ks+k0,js+j0,is+i0);
140               efld1v[pp] += w * fcoup(IE1,ks+k0,js+j0,is+i0);
141               efld2v[pp] += w * fcoup(IE2,ks+k0,js+j0,is+i0);
142               efld3v[pp] += w * fcoup(IE3,ks+k0,js+j0,is+i0);
143               ebv[pp]    += w * fcoup(IEB,ks+k0,js+j0,is+i0);
144             }
145           }
146         }
147       }
148
149 #pragma omp simd aligned(x1p,x2p,x3p,v1p,v2p,v3p:64) simdlen(SIMD_WIDTH)
150       for (int pp=0; pp<std::min(SIMD_WIDTH,remains); pp++) {
151         Real x1tmp = x1p[pp];
152         Real x2tmp = x2p[pp];
153         Real x3tmp = x3p[pp];
154         Real v1tmp = v1p[pp];
155         Real v2tmp = v2p[pp];
156         Real v3tmp = v3p[pp];
157
158         Real x1n = x1tmp + v1tmp * halfdt;
159         Real x2n = x2tmp + v2tmp * halfdt;
160         Real x3n = x3tmp + v3tmp * halfdt;
161
162         Real bsq   = std::max(bfld1v[pp] * bfld1v[pp] + bfld2v[pp] * bfld2v[pp] + bfld3v[pp] * bfld3v[pp], TINY_NUMBER);
...
187         v1tmp = v1n + efld1v[pp] + vp2 * s3 - vp3 * s2;
188         v2tmp = v2n + efld2v[pp] + vp3 * s1 - vp1 * s3;
189         v3tmp = v3n + efld3v[pp] + vp1 * s2 - vp2 * s1;
190         x1p[pp] = x1n + v1tmp * halfdt;
191         x2p[pp] = x2n + v2tmp * halfdt;
192         x3p[pp] = x3n + v3tmp * halfdt;
193         v1p[pp] = v1tmp;
194         v2p[pp] = v2tmp;
195         v3p[pp] = v3tmp;
196       }
197   }
```

9 **New `move` implementation with strip mining and loop fission (part 2)**

```
remark #15305: vectorization support: vector length 8
remark #15309: vectorization support: normalized vectorization overhead 0.055
remark #26012: vectorization support: data layout of a private variable wei1 was optimized, converted to SoA
remark #26012: vectorization support: data layout of a private variable wei2 was optimized, converted to SoA
remark #26012: vectorization support: data layout of a private variable wei3 was optimized, converted to SoA
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 193
remark #15449: unmasked aligned unit stride stores: 224
remark #15450: unmasked unaligned unit stride loads: 81
remark #15451: unmasked unaligned unit stride stores: 9
remark #15462: unmasked indexed (or gather) loads: 189
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 2182
remark #15477: vector cost: 2890.250
remark #15478: estimated potential speedup: 0.710
remark #15487: type converts: 6
remark #15488: --- end vector cost summary ---

remark #15305: vectorization support: vector length 8
remark #15309: vectorization support: normalized vectorization overhead 0.147
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 49
remark #15449: unmasked aligned unit stride stores: 16
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 367
remark #15477: vector cost: 48.500
remark #15478: estimated potential speedup: 6.590
remark #15486: divides: 3
remark #15488: --- end vector cost summary ---
```

10 Vectorization report for the revised `move` code

First, the outermost particle loop is changed to two nested loops using strip mining. In the first nested loop (L69-L133), the particle properties are deposited on a set of local arrays with stride-one memory access, so the compiler should have no trouble generating highly vectorized machine code.

In the second nested loop (L135-L159), the information saved in the local arrays is transferred to an extended global grid array (L163-L205), where each grid point is associated with particle deposition on surrounding grid points (i.e., a 27-point stencil). The price we pay here is the extra storage required for the extended global array. The storage can be reduced if we consider the 27-point stencil as three 9-point stencils.

The grid-based values in the extended global array need to be merged at the end. Though it will be difficult to vectorize the merging operation, the cost isn't significant, since the loop is over grid indices at least an order of magnitude smaller than the particle one.

```
63    Real wei1x2v[9][SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
64    Real wei3v[3][SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
65    int ind1[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
66    int ind2[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
67    int ind3[SIMD_WIDTH] __attribute__((aligned(CACHELINE_BYTES)));
68
69    for (int i=0; i<8*4*ncells1g*ncells2g*ncells3g; i++){
70      mcoupv[i] = 0.0;
71    }
72
73    time = clock();
74
75    for (long p=0; p<nparticle; p+=SIMD_WIDTH)
76      {
77        int remains = nparticle - p;
78        Real *__restrict__ x1p = x1+p;
79        Real *__restrict__ x2p = x2+p;
80        Real *__restrict__ x3p = x3+p;
81        Real *__restrict__ v1p = v1+p;
82        Real *__restrict__ v2p = v2+p;
83        Real *__restrict__ v3p = v3+p;
84
85  #pragma omp simd aligned(x1p,x2p,x3p,v1p,v2p,v3p:64) simdlen(SIMD_WIDTH)
86        for (int pp=0; pp<std::min(SIMD_WIDTH, remains); pp++) {
87
88          Real x1tmp = x1p[pp];
89          Real x2tmp = x2p[pp];
90          Real x3tmp = x3p[pp];
91          Real v1tmp = v1p[pp];
92          Real v2tmp = v2p[pp];
93          Real v3tmp = v3p[pp];
94
95          Real a = (x1tmp - x1s) * dx1 + isg;
96          int ig = (int)(a);
97          int is = ig - 1;
98          Real d = a - ig;
99          Real wei1[3];
100         wei1[0] = 0.5 * (1.0 - d) * (1.0 - d);
101         wei1[1] = 0.75 - (d - 0.5) * (d - 0.5);
102         wei1[2] = 0.5 * d * d;
103         ind1[pp] = is - isg + NGHOST;
...
124           wei1x2v[0][pp] = wei1[0]*wei2[0];
125           wei1x2v[1][pp] = wei1[1]*wei2[0];
126           wei1x2v[2][pp] = wei1[2]*wei2[0];
127           wei1x2v[3][pp] = wei1[0]*wei2[1];
128           wei1x2v[4][pp] = wei1[1]*wei2[1];
129           wei1x2v[5][pp] = wei1[2]*wei2[1];
130           wei1x2v[6][pp] = wei1[0]*wei2[2];
131           wei1x2v[7][pp] = wei1[1]*wei2[2];
132           wei1x2v[8][pp] = wei1[2]*wei2[2];
133         }
134
135         for (int pp=0; pp<std::min(SIMD_WIDTH,remains); pp++) {
136           Real v1tmp = v1p[pp];
137           Real v2tmp = v2p[pp];
138           Real v3tmp = v3p[pp];
139           for (int k=0; k<=2; k++){
140             int index = ind1[pp] + ind2[pp]*ncells1g + (ind3[pp]+k)*ncells1g*ncells2g;
141             Real *mcoupvp = mcoupv+index*32;
142             Real wei3tmp = wei3v[k][pp];
143 #pragma omp simd aligned(mcoupvp) simdlen(SIMD_WIDTH)
144             for (int ij=0; ij<8; ij++){
145               Real weight = wei3tmp*wei1x2v[ij][pp];
146
147               mcoupvp[ij] += weight;
148               mcoupvp[8+ij] += weight*v1tmp;
149               mcoupvp[16+ij] += weight*v2tmp;
150               mcoupvp[24+ij] += weight*v3tmp;
151             }
152             Real weight8 = wei3tmp*wei1x2v[8][pp];
153             int index8 = 32*(index+1+ncells1g);
154             mcoupv[index8] += weight8;
155             mcoupv[index8+8] += weight8*v1tmp;
156             mcoupv[index8+16] += weight8*v2tmp;
157             mcoupv[index8+24] += weight8*v3tmp;
158           }
159         }
160
161       }
...
163     for (int k=pmy_block->ks-NGHOST; k<=pmy_block->ke+NGHOST; k++) {
164       for (int j=pmy_block->js-NGHOST; j<=pmy_block->je+NGHOST-2; j++) {
165 #pragma omp simd simdlen(SIMD_WIDTH)
166         for (int i=pmy_block->is-NGHOST; i<=pmy_block->ie+NGHOST-2; i++) {
167             int index = 32*((i-isg+NGHOST)+(j-jsg+NGHOST)*ncells1g+(k-ksg+NGHOST)*ncells1g*ncells2g);
168           mcoup(IDN,k,j,i) += mcoupv[index];
169           mcoup(IDN,k,j,i+1) += mcoupv[index+1];
...
201           mcoup(IM3,k,j+2,i) += mcoupv[index+30];
202           mcoup(IM3,k,j+2,i+1) += mcoupv[index+31];
203         }
204       }
205     }
```

11 **Vectorizable charge deposition algorithm for `deposit`**

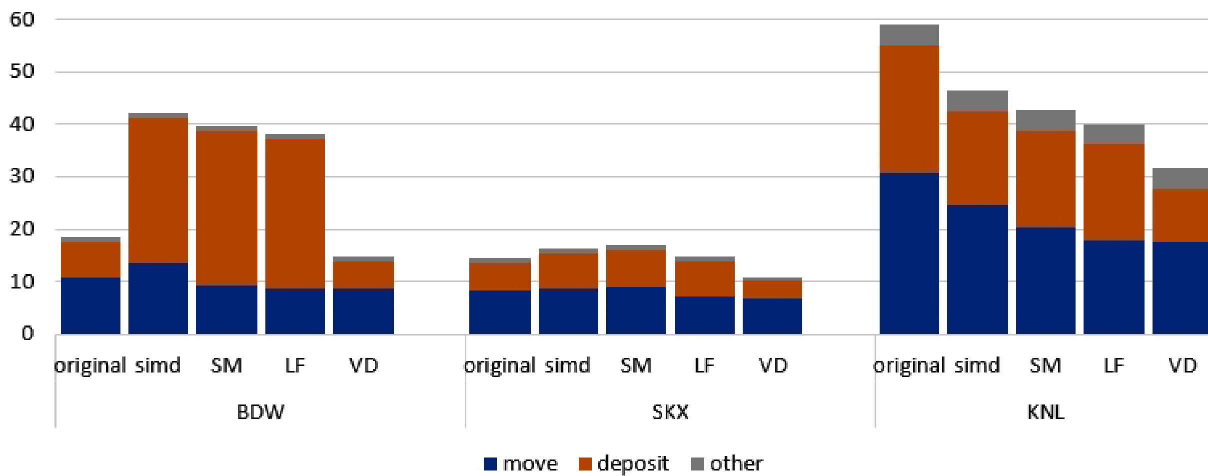
Figure 12 shows the vectorization report for the new implementation. The first and second loops now achieve 5.72 and 5.69 estimated potential speedup, respectively.

Performance Evaluation on Intel Architectures

Performance was evaluated on a 10x10x10 test grid with 100 particles per cell. We ran the simulations for 1,001 timesteps and measured the total wall clock time. We measured performance on single-core Intel® Xeon® and Intel® Xeon Phi™ processors: “BDW” (Intel® Xeon® E5-2680 v4 processor, 2.4GHz, 2 sockets, 14 cores), “SKX” (Intel® Xeon® Gold 6148 processor, 2.40GHz, 2 sockets, 20 cores), and “KNL” (Intel® Xeon Phi™ 7250 processor, 1.4GHz, 1 sockets, 68 cores) (**Figure 13**).

```
LOOP BEGIN at src/particle/deposit.cpp(86,62)
   remark #15388: vectorization support: reference x1p[pp] has aligned access   [ src/particle/deposit.cpp(88,15) ]
   remark #15388: vectorization support: reference x2p[pp] has aligned access   [ src/particle/deposit.cpp(89,15) ]
   remark #15388: vectorization support: reference x3p[pp] has aligned access   [ src/particle/deposit.cpp(90,15) ]
   remark #15388: vectorization support: reference ind1[pp] has aligned access   [ src/particle/deposit.cpp(103,2) ]
   remark #15388: vectorization support: reference ind2[pp] has aligned access   [ src/particle/deposit.cpp(113,2) ]
   remark #15388: vectorization support: reference wei3v[0][pp] has aligned access   [ src/particle/deposit.cpp(119,2) ]
   remark #15388: vectorization support: reference wei3v[1][pp] has aligned access   [ src/particle/deposit.cpp(120,2) ]
   remark #15388: vectorization support: reference wei3v[2][pp] has aligned access   [ src/particle/deposit.cpp(121,2) ]
   remark #15388: vectorization support: reference ind3[pp] has aligned access   [ src/particle/deposit.cpp(122,2) ]
   remark #15388: vectorization support: reference wei1x2v[0][pp] has aligned access   [ src/particle/deposit.cpp(124,2) ]
   remark #15388: vectorization support: reference wei1x2v[1][pp] has aligned access   [ src/particle/deposit.cpp(125,2) ]
   remark #15388: vectorization support: reference wei1x2v[2][pp] has aligned access   [ src/particle/deposit.cpp(126,2) ]
   remark #15388: vectorization support: reference wei1x2v[3][pp] has aligned access   [ src/particle/deposit.cpp(127,2) ]
   remark #15388: vectorization support: reference wei1x2v[4][pp] has aligned access   [ src/particle/deposit.cpp(128,2) ]
   remark #15388: vectorization support: reference wei1x2v[5][pp] has aligned access   [ src/particle/deposit.cpp(129,2) ]
   remark #15388: vectorization support: reference wei1x2v[6][pp] has aligned access   [ src/particle/deposit.cpp(130,2) ]
   remark #15388: vectorization support: reference wei1x2v[7][pp] has aligned access   [ src/particle/deposit.cpp(131,2) ]
   remark #15388: vectorization support: reference wei1x2v[8][pp] has aligned access   [ src/particle/deposit.cpp(132,2) ]
   remark #15305: vectorization support: vector length 8
   remark #15309: vectorization support: normalized vectorization overhead 0.229
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 3
   remark #15449: unmasked aligned unit stride stores: 15
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 219
   remark #15477: vector cost: 31.120
   remark #15478: estimated potential speedup: 5.720
   remark #15487: type converts: 12
   remark #15488: --- end vector cost summary ---
LOOP END
LOOP BEGIN at src/particle/deposit.cpp(144,15)
   remark #15388: vectorization support: reference mcoupvp[ij] has aligned access   [ src/particle/deposit.cpp(147,6) ]
   remark #15388: vectorization support: reference mcoupvp[ij] has aligned access   [ src/particle/deposit.cpp(147,6) ]
   remark #15388: vectorization support: reference mcoupvp[ij+8] has aligned access   [ src/particle/deposit.cpp(148,6) ]
   remark #15388: vectorization support: reference mcoupvp[ij+8] has aligned access   [ src/particle/deposit.cpp(148,6) ]
   remark #15388: vectorization support: reference mcoupvp[ij+16] has aligned access   [ src/particle/deposit.cpp(149,6) ]
   remark #15388: vectorization support: reference mcoupvp[ij+16] has aligned access   [ src/particle/deposit.cpp(149,6) ]
   remark #15388: vectorization support: reference mcoupvp[ij+24] has aligned access   [ src/particle/deposit.cpp(150,6) ]
   remark #15388: vectorization support: reference mcoupvp[ij+24] has aligned access   [ src/particle/deposit.cpp(150,6) ]
   remark #15415: vectorization support: non-unit strided load was generated for the variable <wei1x2v[ij][pp]>, stride is 8
   remark #15305: vectorization support: vector length 8
   remark #15427: loop was completely unrolled
   remark #15309: vectorization support: normalized vectorization overhead 0.182
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 4
   remark #15449: unmasked aligned unit stride stores: 4
   remark #15452: unmasked strided loads: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 37
   remark #15477: vector cost: 5.500
   remark #15478: estimated potential speedup: 5.690
   remark #15488: --- end vector cost summary ---
LOOP END
```

12 **Vectorization report for `deposit`**



13 Performance comparison of various optimization techniques on Intel "BDW," "SKX," and "KNL" processors. The optimization techniques include SIMD, strip mining (SM), loop fission for move (LF), and vectorizable charge deposition (VD). For each case, the reported number is the averaged time of multiple runs on a dedicated system.

We can see that the actual performance is consistent with the estimated potential speedup from the compiler report. On BDW, we see significant performance overhead in using SIMD for `deposit`. This is due to the `#pragma omp ordered simd` clause ensuring the right order of writing to the grid array when compiling with AVX2 instructions. Overall, we see that strip mining, combined with other techniques, has led to 1.3x to 1.9x performance boost on Intel® processors.

Acknowledgements

This work was supported by the Intel® Parallel Computing Center (IPCC) program. We would like to thank Jason Sewall at Intel for providing strong technical support for this work. We also would like to thank Lev Arzamasskiy and Matthew Kunz in the Princeton University Department of Astrophysical Sciences for sharing their source code.

References

1. **https://princetonuniversity.github.io/athena/**
2. Andrey Vladimirov "Optimization Techniques for the Intel MIC Architecture, part 2 of 3: Strip-Mining for Vectorization," Colfax International, June 26, 2015.
3. **https://picsar.net/features/optimized-morse-nielson-deposition/**

CODE YOUR VISION

Accelerate your AI from edge to cloud. Intel® Distribution of OpenVINO™ toolkit speeds up computer vision workloads, streamlines deep learning deployments, and enables easy heterogeneous execution across Intel® platforms.

FREE DOWNLOAD >

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

Intel, the Intel logo, and OpenVINO 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

BOOST PERFORMANCE FOR HYBRID APPLICATIONS WITH MULTIPLE ENDPOINTS IN INTEL® MPI LIBRARY

Minimal Code Changes Can Help You on the March Toward the Exascale Era

Rama Kishan Malladi, Graphics Performance Modeling Engineer, and Dr. Amarpal Singh Kapoor, Technical Consulting Engineer, Intel Corporation

Since the mid-1990s, MPI* has been the *de facto* standard for message passing in distributed-memory, high-performance computing (HPC) applications. With the advent of highly parallel multi-core processors, MPI also found its place for doing message passing within shared-memory systems.

Because there's an overhead associated with using the MPI library, pure MPI applications running on a group of multi-core systems experienced more overhead than was strictly necessary. A workaround for this was to use hybrid parallelism, where MPI applications also used multithreading (e.g., with Pthreads* or OpenMP*) to reduce the number of MPI ranks per node.

To support hybrid parallelism, MPI version 2.1 introduced the `MPI_INIT_THREAD` function to initialize the thread environment with a user-specified level of thread support. Four levels of thread support are available:

- `MPI_THREAD_SINGLE`
- `MPI_THREAD_FUNNELED`
- `MPI_THREAD_SERIALIZED`
- `MPI_THREAD_MULTIPLE`

The first three levels are more restrictive and don't allow threads make concurrent MPI calls. Although the `MPI_THREAD_MULTIPLE` support level places no restrictions, it wasn't widely used due to performance loss arising from internal synchronization among threads.

One of the many enhancements in the **Intel® MPI Library 2019** is scalable endpoints/multi-EP which, with some functionality limitations in the `MPI_THREAD_MULTIPLE` support level, results in better performance for hybrid applications[1]. Multi-EP's novelty lies in its ability to allow multiple threads to be simultaneously active in the MPI runtime without requiring extra synchronization. This allows a single MPI rank using multiple threads to saturate the network bandwidth—eliminating the need for multiple ranks per node.

This article will introduce you to multi-EP, demonstrate the use of multi-EP feature in Intel MPI library, and show the potential performance gains using a simple benchmark and a real-world application.

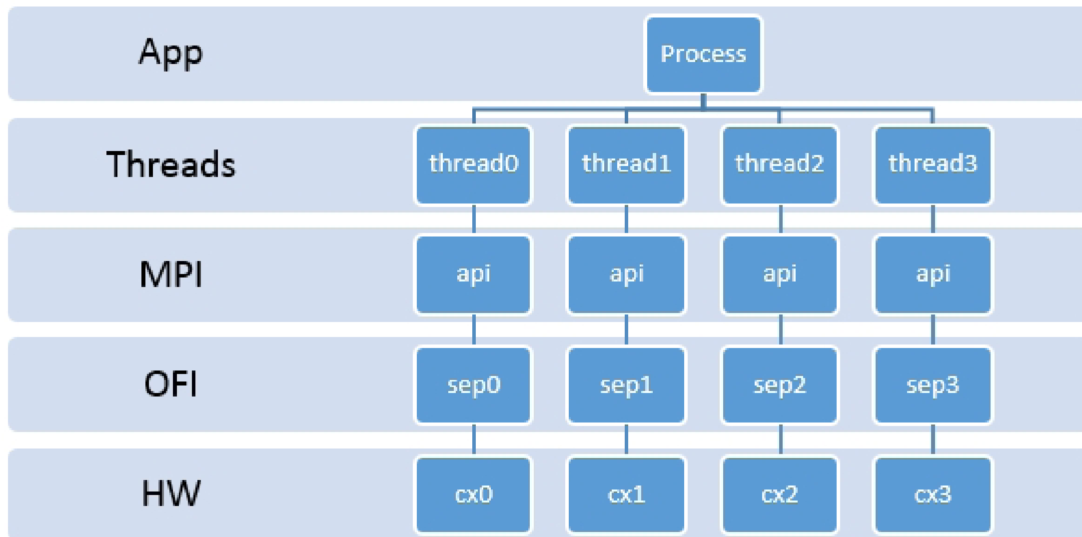
Multiple Endpoints

An MPI endpoint is a set of resources that support the independent execution of MPI communications[2]. An endpoint corresponds to a rank in an MPI communicator. In a hybrid setting, multiple threads may be attached to an endpoint, which lets them communicate using the corresponding endpoint's resources. This generates multiple streams of data for communication, which can be transmitted over the communication medium in parallel with independent hardware context. The result is lockless data transmission from the application layer to the threads and the MPI, OFI, and hardware layers (**Figure 1**).

How to Use Multi-EP in Intel MPI Library 2019

Let's consider the code and environment changes necessary to use multi-EP. We'll assume a thread-compliant MPI code already exists. Since this is a non-standard MPI feature, you must set an environment variable to activate this functionality:

```
$ export I_MPI_THREAD_SPLIT=1
```



1 **Lockless transmission of messages from the application to the interconnect hardware**

Multi-EP is only supported with the `release_mt` and `debug_mt` configurations of the Intel MPI Library, which don't have the global lock. Use one of these methods to select the correct configuration:

```
$ source <installdir>/intel64/bin/mpivars.sh release_mt
```

or

```
$ export I_MPI_LIBRARY_KIND=release_mt
```

Enable multi-EP support within the PSM2 library, which is disabled by default. Also, specify the threading runtime to be used (the OpenMP runtime has special support for multi-EP; however, any other threading runtime may be used). Here's how to do it:

```
$ export PSM2_MULTI_EP=1
$ export I_MPI_THREAD_RUNTIME=openmp
```

Request the `MPI_THREAD_MULTIPLE` level of support via the `MPI_INIT_THREAD` function inside the user code. Duplicate the `MPI_COMM_WORLD` communicator as many times as the number of threads per rank. This step ensures lockless transmission of messages. Use the new communicator in the subsequent MPI function calls.

Here's the environment variable that controls the maximum number concurrent threads per MPI rank:

```
$ export I_MPI_THREAD_MAX=n
```

For the OpenMP runtime, you can use the OMP_NUM_THREADS environment variable instead of I_MPI_THREAD_MAX.

Partition the data being handled by a rank so that each thread originating from that rank has ownership over a certain subset of data (owned by that rank) and acts only upon that data (input and output). Consequently, the number of MPI function calls increases, but the amount of data being transmitted in each function call decreases, proportionately to the number of threads per rank.

Note that there are some limitations when you use multi-EP compared to what's allowed with the MPI_THREAD_MULTIPLE support level (for details, see the **online documentation**).

Benchmark Application

We used the MPI_ALLREDUCE blocking collective function here to reduce (sum) an array of 2,097,152 integers across multiple nodes using the pure MPI approach as well as the hybrid approach with multi-EP (**Figures 2** and **3**). OpenMP is used in the hybrid approach. The reduction operation was repeated 100 times to get time estimates that were free of noise. The application was written in Fortran* and compiled with the 2019.0 versions of the **Intel® Fortran Compiler** and Intel® MPI Library.

```
!  Reduce data
Starttime=MPI_WTIME()
do I = 1,niter
   Call MPI_ALLREDUCE(mydata(;,1),myres,n,MPI_INTEGER,MPI_SUM, &
                       MPI_COMM_WORLD,ierr)

enddo  !i
endtime=MPI_WTIME()
write(*,*)"Time for MPI_ALLREDUCE=",endtime-starttime
```

2 **MPI_ALLREDUCE using pure MPI**

```
!Duplicate communicators for each thread
allocate (NEWCOM(num_threads))
do i = 1,num_threads
    call MPI_COMM(MPI_COMM_WORLD,NEWCOMM(i),ierr)
enddo !1

!Reduce
starttime=MPI_WTIME()
!$OMP PARALLEL PRIVATE(l1,l2,tid)
tid = OMP_GET_THREAD_NUM()
l1 = offset(tid+1,1)
l2 = offset(tid+1,2)
do I = 1,niter
   call MPI_ALLREDUCE(mydata(l1:l2,1),myres(l1:l2,1),chunk, &
   MPI_INTEGER,MPI_SUM,NEWCOMM(tid+1),ierr)
enddo !i
!$OMP END PARALLEL
Endtime=MPI_WTIME()
Write(*,*)"Time for MPI_ALLREDUCE=",endtime-starttime
```

3 **MPI_ALLREDUCE with multi-EP**

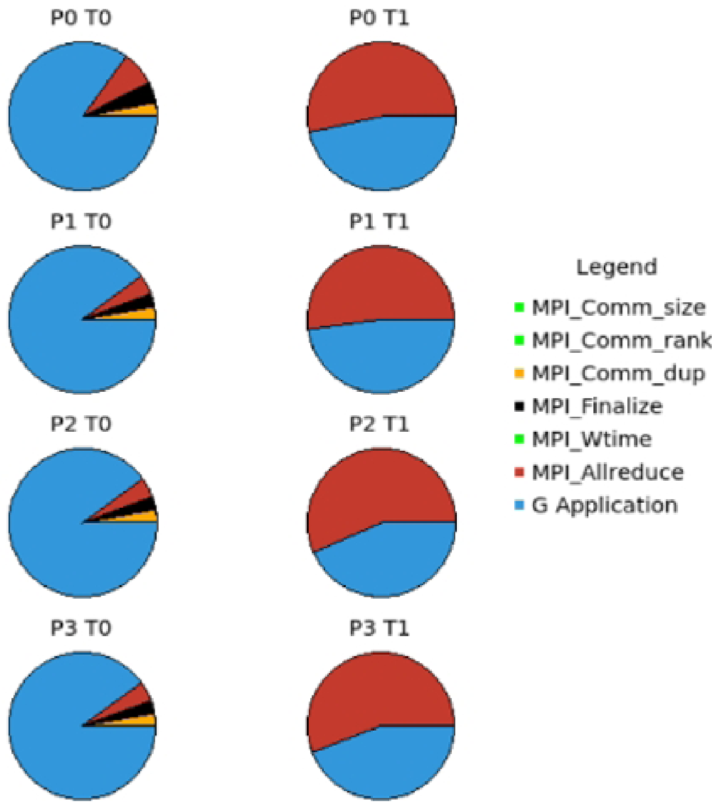
Figure 3 shows the difference from using `MPI_ALLREDUCE` for the multi-EP case. As mentioned before, `MPI_COMM_WORLD` is duplicated as many times as required and the reduction call is made inside a parallel region with clear thread-based ownership of `mydata` and `myre`. The offset is a matrix variable containing the start and end ownership indices for every thread. Also, note that the number of `MPI_ALLREDUCE` calls in the pure MPI case is (`niter × MPI ranks`), while in the multi-EP case it's `(niter × MPI ranks` × `threads per rank`).

The MPI environment was according to the steps mentioned previously and the application was launched with this command:

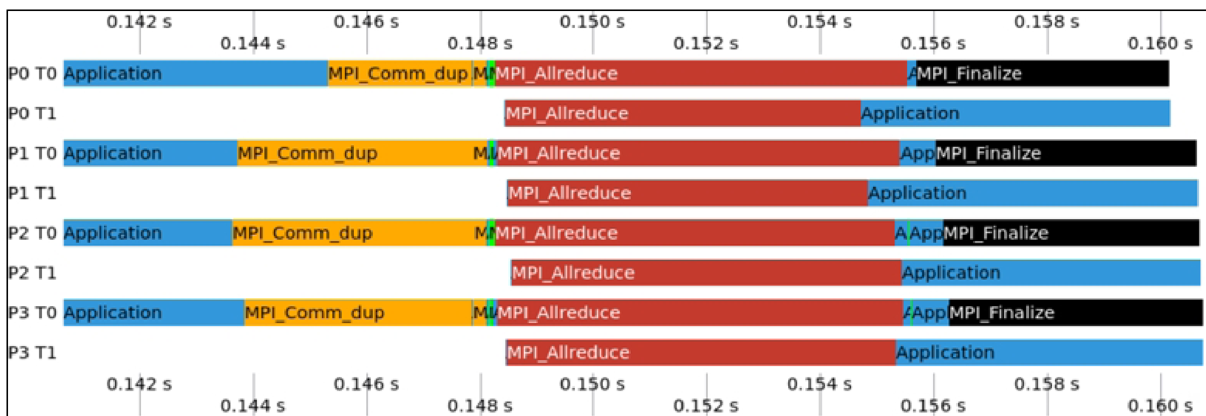
```
$ mpiexec.hydra -n -ppn 1 -f hostfile -prepend-rank -genv OMP_NUM_THREADS NT ./multiEP
```

We use **Intel® Trace Analyzer and Collector** to profile this application and verify the rank and thread decomposition (for details, see the **online documentation**).

For the profiles shown in **Figures 4** and **5**, we use only four nodes, with one rank per node and two threads per rank. The number of repetitions was set to one to make it easier to view the profiles in a static image.



4 **Load balance with four nodes, one rank per node, and two threads per rank**

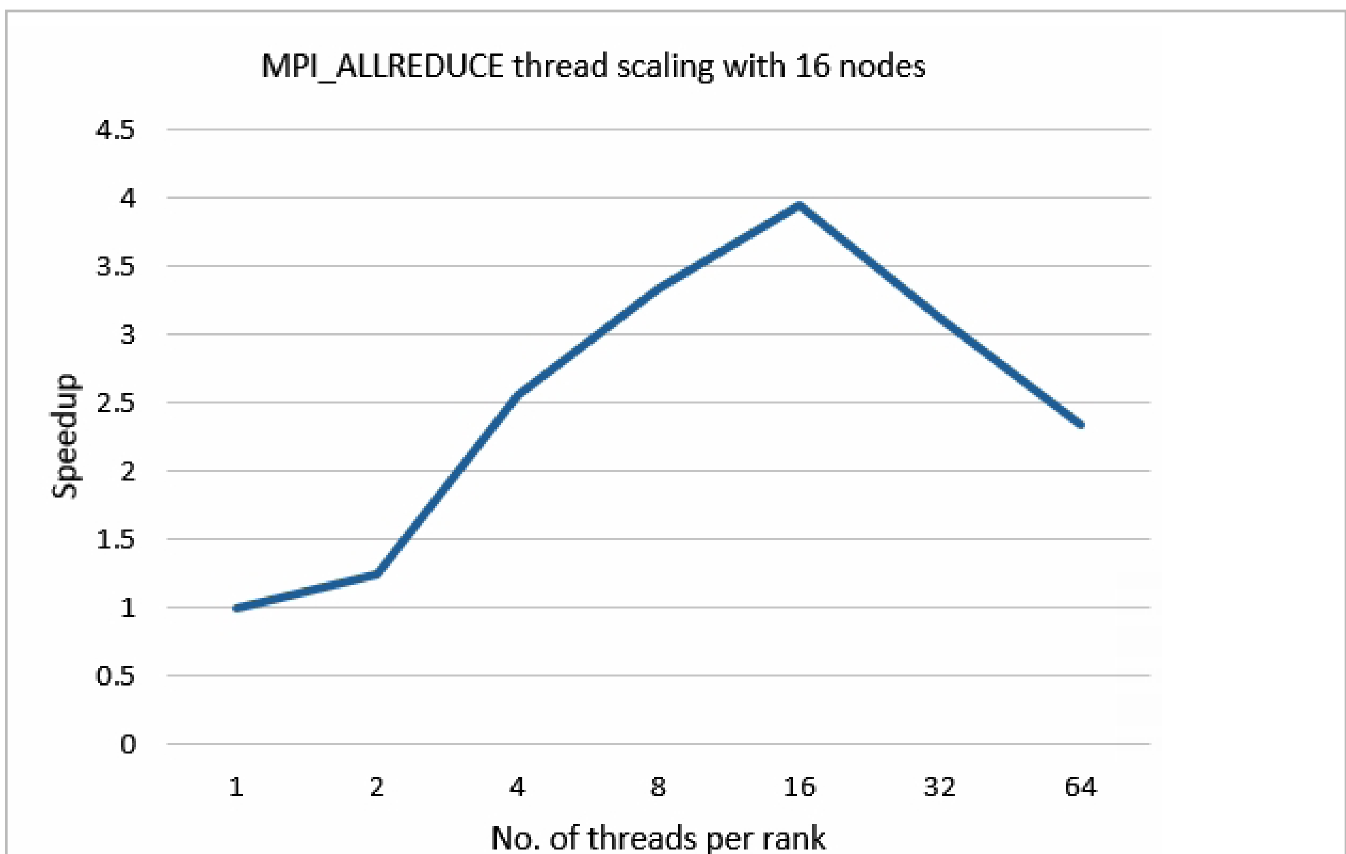


5 **Event timeline showing function invocations in every rank and its threads as a function of time**

Figure 4 shows that every rank did spawn two threads (T0 and T1), as requested. For the small problem being run here, there's significant serial execution (shown in blue). Also, the load balance across threads of the same rank isn't very good. However, the amount of work being done by the first thread of each rank seems to be relatively balanced. We see the same for the second thread of each rank.

Figure 5 shows the timeline along the horizontal and the ranks (with threads) along the vertical. The application starts off running serial code sections in every rank, before duplicating communicators for each thread, and finally invoking `MPI_ALLREDUCE` across all ranks and their threads. Each rank then exits by calling `MPI_FINALIZE` outside the parallel region.

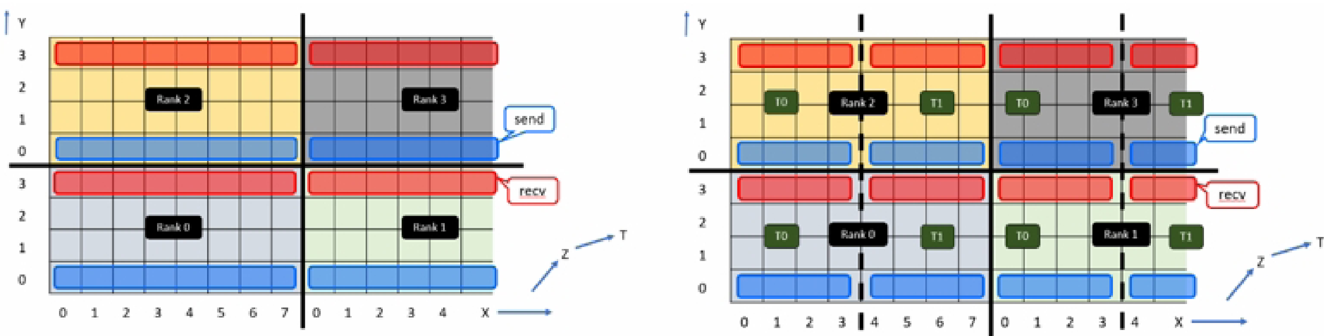
Figure 6 shows the speedup in `MPI_ALLREDUCE` as a function of thread count for a distributed run with 16 nodes and one rank per node. Peak performance occurs with 16 threads, indicating this as the optimal thread count for the system under test (a cluster of **Intel® Xeon Phi™ 7250F processors** connected via Intel® Omni-Path).



6 **Thread scaling with multi-EP**

Multi-EP Use in a Quantum Chromodynamics (QCD) Code

Using multi-EP in real-world applications is quite simple. **Figure 7** shows an example of a QCD application with a halo exchange of boundary data between the nodes (ranks). The communication pattern in the QCD code (Wilson-Dslash operator of a CG solver) is the nearest-neighbor point-to-point (`send-recv`) exchange in the X, Y, Z, and T directions. The figure shows the message exchange in the Y direction, with the multi-EP implementation using two threads per MPI rank. The lattice is partitioned into four ranks, shown as ranks 0-3, and the communication between these ranks is shown in blue and red corresponding to send and receive of halo exchange boundary data (**Figure 8**). The multi-EP version of the code using eight threads is shown in **Figure 9**. A pictorial representation of multi-EP, shown in **Figure 10**, demonstrates using two threads per rank for multi-EP MPI message passing. The threads, shown as T0 and T1, partition the MPI send and receive and perform this communication in parallel, increasing the network bandwidth utilization.



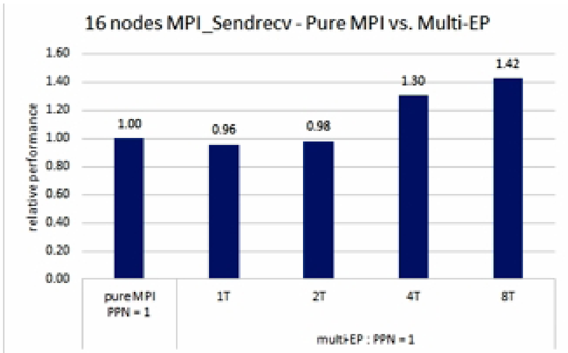
7 **Y direction communication without and with multi-EP on a 16*8*Z*T QCD lattice**

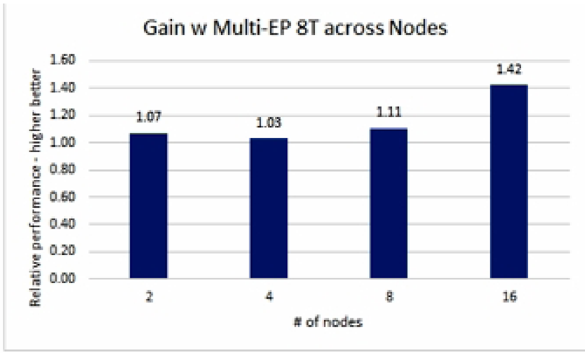
```
sz_y = SZY
tstart = MPI_wtime()
call MPI_Sendrecv(send_yu,sz_y,MPI_DOUBLE_COMPLEX,node_dn(2),
+                     2,recv_yu,sz_y,MPI_DOUBLE_COMPLEX,node_up(2),
+                     2,MPI_COMM_WORLD,iStatus,ierr)
call MPI_Sendrecv(send_yd,sz_y,MPI_DOUBLE_COMPLEX,node_up(2),
+                3,recv_yd,sz_y,MPI_DOUBLE_COMPLEX,node_dn(2),
+                     3,MPI_COMM_WORLD,iStatus,ierr)
Tend = MPI_Wtime()
If(nodeid.eq.0) write(*,*)'Time taken for devault MPI send-recv', tend-tstart
```

8 **Y direction (up and down) communication without multi-EP using eight threads**

```
      sz_y = szy/8
      tstart  = MPI_Wtime()
!$OMP parallel do default(shared) num_threads(8) private(ls,le)
      do iy=1,8
      ls = (iy-1)*sz_y+1
      fle = (iy )*sz_y
      call MPI_Sendrecv(send_yu(ls:le),sz_y,MPI_DOUBLE_COMPLEX,node_dn(2),
+                    2,recv_yu(ls:le),sz_y,MPI_DOUBLE_COMPLEX,node_up(2),
+                    2,comm(iy),iStatus,ierr)
call MPI_Sendrecv(send_yd(ls:le)sz_y,MPI_DOUBLE_COMPLEX,node_up(2),
+                    3,recv_yd(ls:le),sz_y,MPI_DOUBLE_COMPLEX,node_dn(2),
+                    3,comm(iy),iStatus,ierr)
enddo
tend - MPI_Wtime()
if(nodeid.eq.0) write(*,*)'time taken for multii-EP send-recv', tend-tstart
```

9 **Y direction (up and down) communication with multi-EP using eight threads**





10 **Relative performance improvement using multi-EP across different thread counts and node scaling**

The performance gain using multi-EP across various threads and node counts is shown in **Figure 10**. The experiments/runs were done to study the performance gains with multi-EP using 1 to 8 threads and on node counts of 2 to 16 for a lattice size of 64*64*64*16. The system under test was a cluster of Intel® Xeon Phi™ 7250F processor nodes connected via Intel® Omni-Path interconnect.

Better Performance for Hybrid Applications

The multi-EP optimizations in Intel MPI Library 2019 result in better performance for hybrid applications. Using this new feature requires code changes and environment-related settings, but as we've shown, the changes are minimal and don't alter the overall program structure. As we march toward the exascale era, features like multi-EP will become necessary to achieve the best performance at scale from hybrid architectures.

Get Started

- **Learn more**
- **Download Intel MPI Library**

References

1. Intel MPI Library Developer Guide for Linux OS. **https://software.intel.com/en-us/mpi-developer-guide-linux-multiple-endpoints-support**
2. Enabling MPI Interoperability Through Flexible Communication Endpoints. James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. EuroMPI 2013.
3. Intel Trace Analyzer and Collector. **https://software.intel.com/en-us/trace-analyzer**
4. Intel® Omni-Path Architecture Multiple Endpoints. James Erwin, Edward Mascarenhas, and Kevin Pine - Intel. IXPUG September 2018. **https://www.ixpug.org/resources/download/intel-omni-path-architecture-multiple-endpoints**
5. Multiple Endpoints for Improved MPI Performance on a Lattice QCD Code - Larry Meadows, Ken-Ichi Ishikawa, Taisuke Boku, Masashi Horikoshi. HPC Asia 2018 WS, January 31, 2018, Chiyoda, Tokyo, Japan.

INTEL® MPI LIBRARY
Flexible, Efficient, and Scalable Cluster Messaging
FREE DOWNLOAD

INNOVATE SYSTEM AND IOT APPS

How to Debug, Analyze, and Build Applications More Efficiently using Intel® System Studio

Ramya Chandrasekaran and Thorsten Moeller, Product Marketing Engineers, Intel Corporation

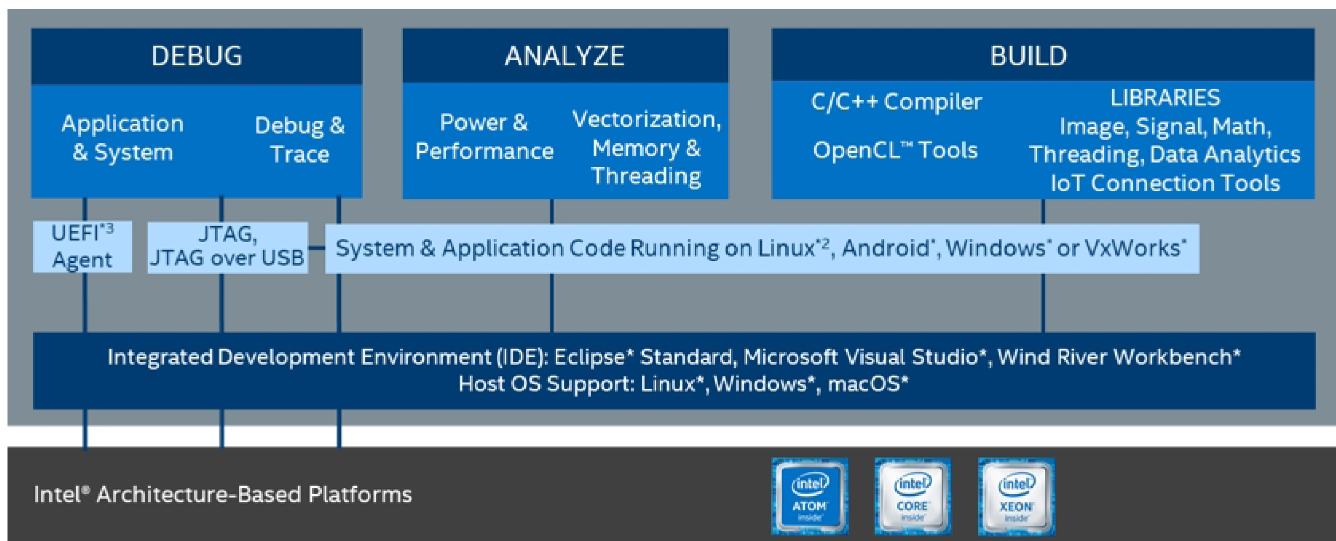
Embedded software development has evolved dramatically over the last few years as applications like IoT, automotive, retail, healthcare, and industrial have grown exponentially. Software development challenges have grown too, as teams cope with shrinking timelines, scarce resources, and rigorous requirements for quality and performance optimization.

One way to help meet these challenges is **Intel® System Studio**, which is specifically designed—and constantly updated—to help address the complexity. This easy-to-use, comprehensive, cross-platform tool suite simplifies system and IoT application development and helps developers quickly and efficiently move from prototype to product. It includes optimizing compilers, highly tuned libraries, analyzers, debuggers, and code wizards and samples that make it easy to get started.

The goal of Intel System Studio is to enhance:

- **Efficiency.** Boost performance and power efficiency for diverse workloads across Intel® processors (CPU and GPU/Intel® Processor Graphics).
- **Productivity.** Find code issues fast and move from prototype to product faster. Integrate smart capabilities with access to 400+ sensors.
- **System reliability.** Improve system bring-up, use deep platform insight and sophisticated trace capabilities.

We can break the tools into three broad categories: build, analyze, and debug (**Figure 1**).



1 **Intel System Studio includes tools to build, analyze, and debug embedded code.**

Tools to Build

Building your project is the first step to go from prototype to production. Intel System Studio's building tools include:

- **Intel® C++ Compiler:** Plugging right into popular development environments like Eclipse* and Microsoft Visual Studio*, Intel® C++ Compilers are source- and binary-compatible with other compilers, such as Visual C++* for Windows* and GNU* Compiler Collection (GCC) for Linux*, macOS*, and Android*.
- **Intel® Math Kernel Library:** Speed computations through highly-optimized, threaded, and vectorized math functions. Provides key functionality for dense and sparse linear algebra (BLAS, LAPACK, PARDISO), FFTs, vector math, summary statistics, splines, and more. It also dispatches optimized code for each processor automatically without the need to branch code.

- **Intel® Data Analytics Acceleration Library:** Helps applications deliver better predictions faster. This tool optimizes data ingestion and analytics together for the highest performance. Supports offline, streaming, and distributed usage models to meet a range of application needs. It splits analytics workloads between edge devices and cloud to optimize overall application throughput.
- **Intel® Integrated Performance Primitives:** Accelerates signal and image processing, data processing, and cryptography tasks. Multi-core, multi-OS, and multi-platform-ready. Easy-to-use, production-ready APIs improve application development and performance.
- **Threading Building Blocks:** Use threading techniques to leverage multicore performance and heterogeneous computing for C++. Parallelize computationally-intensive work across CPUs and GPUs to deliver better solutions.
- **OpenCL™ tools:** Intel is a strong supporter of OpenCL software technology. The **Intel® SDK for OpenCL™ Applications** is a comprehensive development environment for Intel® platforms. It supports offloading compute-intensive parallel workloads to Intel® Graphics Technology using an advanced compiler for OpenCL kernels, runtime debugger, and code performance analyzer.
- **IoT connection tools:** The IoT connection tools are a collection of libraries essential to any IoT solution developer. The tools take advantage of tight integration with the IDE interface and project templates. They provide standardized, open-sourced abstraction libraries and tools. And they include 400+ sensor libraries and advanced cloud connectors for easy and seamless device programming.

Tools to Analyze

Intel System Studio offers several analysis tools to fit your specific use case. To resolve your performance issues, you may end up using more than one analysis tool to pinpoint problems and optimize your system.

- **Intel® VTune™ Amplifier:** Offers a range of analysis types to fit multiple use cases on local or remote systems. Examine hotspots in your code, discover memory issues, and optimize threading across multiple CPU cores. It helps accurately profile C, C++, Java*, Python*, Go*, or any mix and optimize CPU/GPU, threading, memory, cache, storage, and more.
- **Intel® Inspector:** This tool helps you find and debug memory leaks, corruption, data races, and deadlocks.
- **Energy Analysis and Intel® SoC Watch:** These tools collect metrics you can use to analyze power consumption and identify system behaviors that waste energy. View the results as CSV files or graphically with Intel VTune Amplifier.
- **Intel® Advisor:** Helps identify areas where your application can benefit from vectorization and threading. It also analyzes memory patterns and helps quickly prototype threading.
- **Intel® Graphics Performance Analyzers:** Provide insightful analysis of real-time hardware metrics. Spot difficult CPU/GPU interaction issues.

Tools to Debug

As systems and IoT devices have become more sophisticated, they've also become more complicated to implement and maintain. Code bases—including BIOS/UEFI, firmware, operating system layers, and device drivers—have grown so large and complex that just navigating through instruction trace data to find the root cause of a bug is difficult and time-consuming.

Intel® System Debugger can help build reliable code. In early stages of development, it provides system debug and system trace capabilities to help build a stable platform. This full-featured, source-line debugger enables deep analysis of BIOS and UEFI* code, system-on-chip (SoC) peripheral registers, operating system kernels, and device drivers with full operating system awareness. It also provides system trace capabilities to capture, view, and correlate trace information for software, firmware, and hardware components. You can collect, export, and analyze trace data collected by Intel® Trace Hub from sources like BIOS, Intel® Management Engine (Intel® ME), Intel® Converged Security and Management Engine (Intel® CSME), Architecture Event Trace (AET), Microsoft Event Tracing for Windows* (ETW*), and more. These events can be extracted via several mechanisms, including JTAG, system memory, or through a USB connection. Analyzing Intel Trace Hub events from multiple platform sources helps you determine root causes of bugs quickly.

The Intel® Debug Extensions for WinDbg*, another feature of Intel System Studio, is the first solution capable of debugging features such as Winload, HAL initialization, Bitlocker, or system recovery directly on the target. You no longer need to reproduce sleep or hibernation issues with the WinDbg* kernel agent enabled. The only requirement is a functional JTAG connection (e.g., over USB/DCI) and the debug tool can connect at any time.

The debugger enables low-cost, closed-chassis debug on production systems/devices and a debug engineer will gain direct access to failing production system/device where it was previously impossible.

Innovate System and IoT Apps

Intel System Studio helps you deal with squeezed timelines, limited resources, and rigorous optimization needs during all software development stages—from early system bring-up to application optimization and deployment. Solutions that benefit from using Intel System Studio include digital security and surveillance, 5G, networking, industrial, data storage, healthcare, retail, smart homes/cities/buildings, automotive, office automation, and more.

Get Started

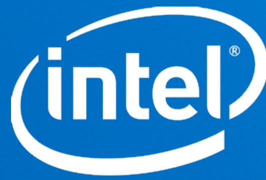
- **Get a free download of Intel System Studio 2019**
- **Learn more about Intel System Studio**

SUPERCHARGE PYTHON* PERFORMANCE

Get your hands on
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



THE PARALLEL UNIVERSE

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/benchmarks.

Performance results are based on testing as of October 1, 2018, and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure.

For more information regarding performance and optimization choices in Intel® Software Development Products, see our Optimization Notice: https://software.intel.com/articles/optimization-notice#opt

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 retaile. No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Copyright © 2019 Intel Corporation. All rights reserved. Intel, Xeon, Xeon Phi, VTune, 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. Printed in USA 0419/SS Please Recycle