Getting Started with Habana® Gaudi® for Deep Learning Training

Cost Matters: On the Importance of Cost-Aware Hyperparameter Optimization

Speeding Up the Databricks Runtime for Machine Learning
Contents

Letter from the Editor 3

FEATURE

Getting Started with Habana® Gaudi® for Deep Learning Training 5
A New Hardware and Software Stack for Deep Learning

Speeding Up the Databricks Runtime for Machine Learning 14
Intel-Optimized Artificial Intelligence in the Cloud

A Novel Scale-Out Training Solution for Deep Learning Recommender Systems 21
Demonstrating Better Parallel Scaling in the MLPerf Benchmark

Cost Matters: On the Importance of Cost-Aware Hyperparameter Optimization 32
Accounting for the Cost Metric that Actually Matters

Katana's High-Performance Graph Analytics Library 37
Python Programmers Have a New Option for Graph Analytics

Accelerate R Code with Intel® oneAPI Math Kernel Library 45
Improve R Performance without Modifying Code

Optimizing the Maxloc Operation Using Intel® AVX-512 Instructions 49
A Guide to Vectorizing this Common Reduction Operation

For more complete information about compiler optimizations, see our Optimization Notice.
Let’s Talk about High-Performance Data Analytics

Recent issues of *The Parallel Universe* have emphasized oneAPI; namely, DPC++ and the component libraries like oneMKL. This issue focuses on data science; in particular, training machine learning and deep learning models. Our feature article, *Getting Started with Habana Gaudi for Deep Learning Training*, describes the Gaudi HPU (Habana Processor Unit) architecture and shows you how to use it. *Speeding Up the Databricks Runtime for Machine Learning* discusses Intel optimizations for doing artificial intelligence in the cloud. *A Novel Scale-Out Training Solution for Deep Learning Recommender Systems* presents the results of a recent collaboration with Facebook to improve the scalability of training. Finally, *Cost Matters: On the Importance of Cost-Aware Hyperparameter Optimization* presents the results of a recent collaboration with Facebook and Amazon to improve hyperparameter tuning.

From there, we look at another important part of the end-to-end data analytics pipeline: graph analytics. Intel has a long history in graph processing research and has active collaborations with many of the top practitioners, e.g.: the GraphBLAS specification, the LDBC Graphalytics benchmark, comprehensive graph analytics analyses, and the PIUMA architecture for efficient and scalable graph analysis. Data scientists have a great package, NetworkX, for graph and network analysis, but it’s not known for performance. Fortunately, our friends at Katana Graph just released a high-performance, parallel graph analytics library for Python programmers. *Katana's High-Performance Graph Analytics Library* offers an alternative for compute-intensive operations on extremely large graphs.

The R programming language is popular with data scientists and statisticians, but like NetworkX, it’s not known for performance. *Accelerate R Code with Intel® oneAPI Math Kernel Library* shows you how to improve the performance simply by linking the R programming environment to oneMKL. No code changes are required.
We close this issue with a follow-up to a previous article on vectorization: Optimization of Scan Operations Using Explicit Vectorization. Optimizing the Maxloc Operation Using AVX-512 Vector Instructions is another how-to guide to using vector intrinsics to accelerate common kernels; in this case, the maxloc reduction.

As always, don’t forget to check out Tech.Decoded for more information on Intel solutions for code modernization, visual computing, data center and cloud computing, data science, systems and IoT development, and heterogeneous parallel programming with oneAPI.

Henry A. Gabb
October 2021
Introduction

Demand for high-performance deep learning (DL) training is accelerating with the growing number of applications and services based on image and gesture recognition in videos, speech recognition, natural language processing, recommendation systems, and more. The Habana® Gaudi® AI processor is designed to maximize training throughput and efficiency, while providing developers with optimized software and tools that scale to many workloads and systems. Habana Gaudi software was developed with the end user in mind, providing versatility and ease of programming to address the unique needs of users’ proprietary models, while allowing for a simple and seamless transition of their existing models to Gaudi. We want to get you started with Habana’s Gaudi HPU (Habana processor unit). There are many new and existing users who may have experience running workloads on a GPU or CPU, and the goal of this article is to take
you through the process of migrating your existing AI workloads and models over to the Gaudi HPU. We'll introduce you to the basics of the hardware architecture, the software stack, and the tools needed to get your models running on Gaudi.

**Gaudi Architecture and Habana SynapseAI® Software Suite**

We'll start with an overview of the basic architecture. Gaudi is designed from the ground up to accelerate DL training workloads. Its heterogeneous architecture comprises a cluster of fully programmable Tensor Processing Cores (TPC) along with its associated development tools and libraries, plus a configurable Matrix Math engine.

The TPC is a VLIW SIMD processor with instruction set and hardware tailored for training workloads. It is programmable, providing the user with maximum flexibility to innovate, coupled with many workload-oriented features, such as:

- GEMM operation acceleration
- Tensor addressing
- Latency hiding capabilities
- Random number generation
- Advanced implementation of special functions

The TPC natively supports the following data types: FP32, BF16, INT32, INT16, INT8, UINT32, UINT16, and UINT8.

The Gaudi memory architecture includes on-die SRAM and local memories in each TPC. In addition, the chip package integrates four HBM devices, providing 32 GB of capacity and 1 TB/s bandwidth. The PCIe interface provides a host interface and supports both generation 3.0 and 4.0 modes.

Gaudi is the first DL training processor that has integrated RDMA over Converged Ethernet (RoCE v2) engines on-chip. With bi-directional throughput of up to 2 Tb/s, these engines play a critical role in the inter-processor communication needed during the training process. This native integration of RoCE allows customers to use the same scaling technology, both inside the server and rack (scale-up), as well as to scale across racks (scale-out). These can be connected directly between Gaudi processors, or through any number of standard Ethernet switches. **Figure 1** shows the hardware block diagram:
Designed to facilitate high-performance DL training on Habana’s Gaudi accelerators, the SynapseAI software suite enables efficient mapping of neural network topologies onto Gaudi hardware. The software stack includes Habana’s graph compiler and runtime, TPC kernel library, firmware and drivers, and developer tools such as the TPC SDK for custom kernel development and SynapseAI Profiler. SynapseAI is integrated with popular frameworks, TensorFlow and PyTorch, and performance-optimized for Gaudi. Figure 2 shows the components of the SynapseAI software suite. To easily integrate the SynapseAI software into your working environment, Habana provides a set of Docker images for TensorFlow and PyTorch that includes all the ingredients needed to create the environment to run your models. We’ll explore how to integrate these libraries into your models.
Setting up the Environment

In this section, we address how to set up the environment, and then add the simple steps needed to ensure that the Gaudi HPU is recognized by the framework and can start to execute ops in your model. In addition to the Docker images, Habana also has a set of reference models and examples in our Model-References GitHub repository that can be used as guides to add the proper components to your models.

The first step to getting started is to ensure you have a full build environment, which includes the Habana driver, SynapseAI software stack, and framework. In most cases, the best way to ensure that this full environment is created is to use the pre-built Docker images provided by Habana, which are available in our Software Vault. These Docker images contain both single-node and scale-out binaries and do not require additional installation steps. If you are using Gaudi in a cloud-based environment, be sure to select a Habana Gaudi image provided by your cloud service provider that contains the full driver and associated framework. For those who need to perform an installation of the full driver and SynapseAI software stack independently (as an on-premise installation), you can refer to the dedicated Setup and Install GitHub repository for detailed instructions. The second step is to load the Habana libraries and target the Gaudi HPU device. We’ll explain how to do this for both TensorFlow and PyTorch in the next section.
Getting Started with TensorFlow

For TensorFlow, Habana integrates the TensorFlow framework with SynapseAI in a plug-in using `tf.load_library` and `tf.load_op_library`, calling library modules and custom ops/kernels. The framework integration includes three main components:

- SynapseAI helpers
- Device
- Graph passes

The TensorFlow framework controls most of the objects required for graph build or graph execution. SynapseAI allows users to create, compile, and launch graphs on the device. The Graph passes library optimizes the TensorFlow graph with operations of Pattern Matching, Marking, Segmentation, and Encapsulation (PAMSEN). It is designed to manipulate the TensorFlow graph to fully utilize Gaudi’s hardware resources. Given a collection of graph nodes that have implementations for Gaudi, PAMSEN tries to merge as many graph nodes as possible while maintaining graph correctness. By preserving graph semantics and automatically discovering subgraphs that can be fused into one entity, PAMSEN delivers performance that should be on par with (or better than) native TensorFlow.

To prepare your model, you must load the Habana module libraries. Call `load_habana_module()` located under `library_loader.py`. This function loads the Habana libraries needed to use Gaudi HPU at the TensorFlow level:

```python
import tensorflow as tf
from habana_frameworks.tensorflow import load_habana_module
load_habana_module()
```

There are some specific requirements when running legacy (TF1.x) models, and the requirements differ when using Horovod for multicard and multinode training. If you are running TF2.x models on a single card, the last statement is not needed.

Once loaded, the Gaudi HPU is registered in TensorFlow and prioritized over CPU. This means that when a given op is available for both CPU and the Gaudi HPU, the op is assigned to the Gaudi HPU. After the initial model migration is completed using the steps above, you can start to look at integration of Habana ops and custom TensorFlow ops as defined in the `habana_ops` object. It can be imported with the command:

```python
from habana_frameworks.tensorflow import habana_ops
```

It should only be used after `load_habana_module()` is called. The custom ops are used for pattern matching to standard TensorFlow ops. When the model is ported to the Gaudi HPU, the software stack decides which ops are placed on the CPU and which are placed on the Gaudi HPU. The optimization pass automatically places unsupported ops on the CPU.

Once these steps are completed in your model, you will be able to run it on a Gaudi HPU instance. Below is a simple example showing how an MNIST model is enabled for Gaudi. (The minimal changes required are...
highlighted in bold):

```python
import tensorflow as tf
from TensorFlow.common.library_loader import load_habana_module
load_habana_module()
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = tf.keras.models.Sequential(
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(10),
)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=128)
model.evaluate(x_test, y_test)
```

### Getting Started with PyTorch

The PyTorch Habana bridge interfaces between the framework and SynapseAI software stack to drive the execution of deep learning models on the Habana Gaudi device (Figure 3). The installation package provided by Habana comes with modifications on top of the standard PyTorch release and are included in the Docker images provided by Habana. The customized framework from this installed package needs to be used to integrate PyTorch with the Habana bridge. In this case, you will modify the PyTorch deep learning model training scripts by loading the PyTorch Habana plugin library and

```python
import habana_frameworks.torch.core
```
module to integrate with Habana Bridge.

Execution of PyTorch models on the Gaudi HPU has two main modes that are supported by Habana PyTorch:

- **Eager mode** – op-by-op execution, as defined in standard PyTorch eager mode scripts.
- **Lazy mode** – deferred execution of graphs, comprising ops delivered from script op-by-op, like eager mode. It gives the eager mode experience with performance on Gaudi.

![Figure 3. PyTorch Habana Full Stack Architecture](image)

For more complete information about compiler optimizations, see our Optimization Notice.
In general, it is recommended that initial model development should be run in eager mode to establish functionality, then run in lazy mode for best performance. This can be selected by setting a runtime flag (as shown in **Step 5** below).

The following set of code additions are required to run a model on Habana. The following steps cover eager and lazy modes of execution.

1. Load the Habana PyTorch plugin library, `libhabana_pytorch_plugin.so`:
   ```python
   import torch
   from habana_frameworks.torch.utils.library_loader import load_habana_module
   load_habana_module()
   ```

2. Target the Gaudi HPU device:
   ```python
   device = torch.device("hpu")
   ```

3. Move the model to the device:
   ```python
   model.to(device)
   ```

4. Import the Habana Torch Library:
   ```python
   import habana_frameworks.torch.core as htcore
   ```

5. Enable lazy execution mode by setting the environment variable shown below. Do not set this environment variable if you want to execute your code in eager mode:
   ```python
   os.environ["PT_HPU_LAZY_MODE"] = "1"
   ```

6. In lazy mode, execution is triggered wherever data is read back to the host from the Habana device. For example, execution is triggered if you are running a topology and getting loss value into the host from the device with `loss.item()`. Adding a `mark_step()` in the code is another way to trigger execution:
   ```python
   htcore.mark_step()
   ```

The placement of `mark_step()` is required at the following points in a training script:

- Right after `optimizer.step()` to cleanly demarcate training iterations,
- Between `loss.backward` and `optimizer.step()` if the optimizer being used is a Habana custom optimizer.

When the model is ported to run on Gaudi HPU, the software stack decides which ops are placed on CPU and which are placed on the HPU. This decision is based on whether the op is registered with PyTorch with HPU as the backend. Execution of an op automatically falls back to the host CPU if the op is not registered with its backend as an HPU.
Habana provides its own implementation of some complex PyTorch ops customized for Habana devices. In a given model, replacing these complex ops with custom Habana versions will give better performance.

The following is a list of custom optimizers currently supported on Habana devices:

- FusedAdagrad – refer to `torch.optim.Adagrad`
- FusedAdamW – refer to `torch.optim.AdamW`
- FusedLamb – refer to LAMB optimizer paper
- FusedSGD – refer to `torch.optim.SGD`

**Conclusion**

The goal of this article was to take you through the process of migrating your existing deep learning models over to Gaudi and show the basic steps to get your model ready to run. Please keep watching the main website, Habana.ai, for product launch announcements and sign up to our interest list to get notified of upcoming activities.

**Useful Resources**

- [Developer site](#) is the best place to get started. This will have pointers to our models, documentation, and other resources.
- [Habana Documentation](#) has all the detailed collateral, especially the [Migration Guide](#).
- [Habana Vault](#) is the place to download the latest drivers and Docker images.
- [User Forum](#) is the place for community discussion of all things related
Break Free of Code Boundaries

Experience the power of cross-architecture programming in the Intel® DevCloud for oneAPI.

Demo
Run our Mandelbrot demo on different architectures to see cross-architecture performance for yourself.

Learn
Get hands-on experience with Data Parallel C++ with 25 Jupyter notebooks loaded with code samples.

Develop
Plan and test future-ready applications on the latest Intel CPUs, GPUs, and FPGAs.

GET STARTED NOW >

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

1.1 Scikit-learn

Scikit-learn is a popular open-source machine learning (ML) library for the Python programming language. It features various classification, regression, and clustering algorithms, including support vector machines, random forests, gradient boosting, k-means, and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

The Intel® Extension for Scikit-learn, made available through Intel® oneAPI AI Analytics Toolkit, boosts ML performance and gives data scientists time back to focus on their models. Intel has invested in optimizing performance of Python itself, with the Intel® Distribution for Python, and has optimized key data science libraries used with scikit-learn, such as XGBoost, NumPy, and SciPy. This article gives more information on...
installing and using these extensions.

1.2 **TensorFlow**

TensorFlow is another popular open-source framework for developing end-to-end ML and deep learning (DL) applications. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that let researchers easily build and deploy applications.

To take full advantage of the performance available in Intel® processors, TensorFlow has been optimized using Intel® oneAPI Deep Neural Network Library (oneDNN) primitives. For more information on the optimizations as well as performance data, refer to [TensorFlow Optimizations on Modern Intel® Architecture](#).

2 **Databricks Runtime for Machine Learning**

Databricks is a unified data analytics platform for data engineering, ML, and collaborative data science. It offers comprehensive environments for developing data-intensive applications. [Databricks Runtime for Machine Learning](#) is an integrated end-to-end environment, incorporating managed services for experiment tracking, model training, feature development and management, and feature and model serving. It includes the most popular ML/DL libraries, such as TensorFlow, PyTorch, Keras, and XGBoost, and also includes libraries required for distributed training, such as Horovod.

Databricks has been integrated with Amazon Web Services, Microsoft Azure, and Google Cloud Platform service. These cloud service providers bring great convenience to manage production infrastructure and run production workloads. Though cloud services are not free, there are opportunities to reduce the cost for ownership by utilizing the optimized libraries. In this article, we will use Databricks on Azure to demonstrate the solution and the performance results we achieved.

3 **Intel-Optimized ML Libraries on Azure Databricks**

Databricks Runtime for ML includes the stock versions of scikit-learn and TensorFlow. To boost performance, however, we will replace them with Intel-optimized versions. Databricks provides [initialization scripts](#) to facilitate customization. They run during the startup of each cluster node. We developed two initialization scripts to incorporate the Intel-optimized versions of scikit-learn and TensorFlow, depending on whether you want to install the statically patched version or not:

1. `init_intel_optimized_ml.sh` installs statically patched scikit-learn and TensorFlow in the runtime environment.

2. `init_intel_optimized_ml_ex.sh` installs Intel Extension for Scikit-Learn and TensorFlow in the runtime environment.

The following instructions show how to create a cluster. First, copy the initialization script to DBFS:
1. Download either `init_intel_optimized_ml.sh` or `init_intel_optimized_ml_ex.sh` to a local folder.

2. Click the Data icon in the left sidebar.

3. Click the DBFS button and then Upload button at the top.

4. Select a target directory, for example, FileStore, in the Upload Data to DBFS dialog.

5. Browse to the local file previously downloaded in the local folder to upload in the Files box.

Next, launch the Databricks cluster using the uploaded initialization script:

1. Click the Advanced Options toggle on the cluster configuration page.

2. Click the Init Scripts tab at the bottom right.

3. Select the DBFS destination type in the Destination drop-down menu.

4. Specify the path to the previously uploaded initialization script:
   `dbfs:/FileStore/init_intel_optimized_ml.sh` or `dbfs:/FileStore/init_intel_optimized_ml_ex.sh`

5. Click Add.
Refer to [Intel Optimized ML for Databricks](#) for more detailed information.

## 4 Performance Measurements

### 4.1 Scikit-learn Training and Prediction Performance
We used Databricks Runtime Version 7.6 ML for the following benchmarks. We use `scikit-learn_bench` to compare the performance of common scikit-learn algorithms with and without the Intel optimizations. For convenience, the `benchmark_sklearn.ipynb` notebook is provided to run `scikit-learn_bench` on Databricks Cloud.

We compared training and prediction performance by creating one single-node Databricks cluster with the stock library and another with the Intel-optimized version. Both clusters used Standard_F16s_v2 instance type.

The benchmark notebook was run on both clusters. For each algorithm, we set multiple configurations to get accurate training and prediction performance data, and below (Table 1) shows the performance data of one configuration for each algorithm.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Input Config</th>
<th>Training Time (seconds)</th>
<th>Prediction Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Stock scikit-learn (baseline)</td>
<td>Intel Extension for Scikit-learn</td>
</tr>
<tr>
<td>kmeans</td>
<td>config1</td>
<td>517.13</td>
<td>24.41</td>
</tr>
<tr>
<td>ridge_regression</td>
<td>config1</td>
<td>1.22</td>
<td>0.11</td>
</tr>
<tr>
<td>linear_regression</td>
<td>config1</td>
<td>3.1</td>
<td>0.11</td>
</tr>
<tr>
<td>logistic_regression</td>
<td>config3</td>
<td>87.5</td>
<td>5.94</td>
</tr>
<tr>
<td>svm</td>
<td>config2</td>
<td>271.58</td>
<td>12.24</td>
</tr>
<tr>
<td>kd_tree_knn_classification</td>
<td>config4</td>
<td>0.84</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 1. Comparing training and prediction performance (all times in seconds)

For each algorithm, the Intel-optimized version of scikit-learn greatly improved training and prediction performance. For some algorithms, like svm and brute_knn, it achieved order of magnitude speed-up (Figure 1).
4.2 TensorFlow Training and Prediction Performance

BERT, or Bidirectional Encoder Representations from Transformers, is a new method of pre-training language representations that obtains state-of-the-art results on a wide range of natural language processing tasks. Model Zoo contains links to pre-trained models, sample scripts, best practices, and step-by-step tutorials for many popular open-source machine learning models optimized by Intel to run on Intel® Xeon® Scalable processors.

We used Model Zoo to run the BERT-Large model on SQuADv1.1 datasets to compare the performance of TensorFlow with and without our optimizations. Once again, we provide a notebook (benchmark_tensorflow_bertlarge.ipynb) to run the benchmark on the Databricks Cloud. Refer to Run Performance Comparison Benchmarks for more details.

We used Databricks Cloud Single Node with Standard_F32s_v2, Standard_F64s_v2, and Standard_F72s_v2 instance types for the TensorFlow performance evaluation. For each instance type, we compared the inference and training performance between stock TensorFlow and the Intel-optimized TensorFlow. The latter delivers 2.09x, 1.95x, and 2.18x inference performance on Databricks Runtime for ML with Standard_F32s_v2, Standard_F64s_v2, and Standard_F72s_v2 instances, respectively (Figure 2). For training, the Intel-optimized TensorFlow delivers 1.76x, 1.70x, and 1.77x training performance on Standard_F32s_v2, Standard_F64s_v2, and Standard_F72s_v2 instances, respectively (Figure 3).
The Parallel Universe

5 Concluding Remarks

The Intel-optimized versions of scikit-learn and TensorFlow deliver significant improvements in training and inference performance on Intel XPU. We demonstrated that replacing the stock scikit-learn and TensorFlow included in Databricks Runtime for Machine Learning with the optimized versions can improve performance and, hence, reduce cost for customers.
A Novel Scale-Out Training Solution for Deep Learning Recommender Systems

Demonstrating Better Parallel Scaling in the MLPerf Benchmark

Liangang Zhang, Machine Learning Engineer; Guokai Ma, Machine Learning Engineer; Roger Feng, Deep Learning Software Engineer; Fan Zhao, Deep Learning Software Engineer; and Ke Ding, Principal AI Engineer, Intel Corporation

DLRM (Deep Learning Recommendation Model) is a deep learning-based model for recommendations introduced by Facebook. It’s a state-of-the-art model and part of the MLPerf training benchmark. DLRM poses unique challenges on single- and multi-socket distributed training because of the need to balance compute-, memory-, and I/O-bound operations. To tackle this challenge, we implemented an efficient scale-out solution for DLRM training on an Intel® Xeon® processor cluster using data and model parallelization, new hybrid splitSGD + LAMB optimizers, efficient hyperparameter tuning for model convergence with much larger global batch size, and a novel data loader to support scale-up and scale-out. The MLPerf v1.0 training results demonstrate that we can train a DLRM with 64 Intel Xeon 8376H processors in 15 minutes, a 3x improvement over our MLPerf v0.7 submission (which could only scale to 16 Intel Xeon 8380 processors). In this article, we describe the optimizations that were applied to achieve this performance improvement.
Vertical Split Embedding Table-Based Hybrid Parallelism

We use a hybrid parallel solution to improve the scalability of DLRM MLPerf training, and then use a vertical split embedding table to improve scaling even further (Figure 1).

![Figure 1. DLRM model structure. Dense features are the input of the bottom MLP (multilayer perceptron) layer, and sparse features are inputs for the embedding table. The outputs from the bottom MLP and embedding are the input to the top MLP layer.]

There are 26 embedding tables in MLPerf DLRM training. The table entry numbers are 40M, 40M, 40M, 40M, 40790948, 3067956, 590152, 405282, 39060, 20265, 17295, 12973, 11938, 7424, 7122, 2209, 1543, 976, 155, 108, 63, 36, 14, 10, 4, and 3. To model the semantics for every entry, 128 Float32/Bfloat16 numbers are used. A straightforward way to address an embedding table is to use data parallelism via sparse all-reduce. Model weights need to be replicated across model instances. With data parallelism, all embedding tables should be replicated, so 26 embedding tables require more than 100 GB memory for a single model instance.

To reduce communication overhead and the memory requirement on each device, we use a hybrid-parallel distributed training solution (Figures 2 and 3). The embedding table is divided into smaller tables that use dense gradients and large embedding tables that use sparse gradients. In the case of MLPerf DLRM, an embedding table is treated as a small table if an entry number is less than 2048; otherwise, it is treated as a large table. We get 10 small and 16 large embedding tables. For model parallelism, model instances would hold a local copy of part of the large embedding tables. For example, if we use eight sockets and one
instance per socket, then every instance will hold two large embedding tables. For 16 sockets, every instance will only hold one large embedding table. Instead of lookup embedding tables with indices in a local batch, each model instance lookup local embedding table with indices in a global batch. After the lookup operation, model instances not only have lookup entries of their own local batches, but also have lookup entries of other instances’ batches. All-to-all collective communication is used to exchange embedding information between ranks. For data-parallelism, bottom MLP, top MLP, and 10 small embedding tables are replicated in every model instance, and all-reduce collective communication is used to average the gradient between ranks.

Figure 2. Illustration of hybrid parallelism between four model instances with four embedding tables. The colored blocks indicate different embedding tables (gbs: global batch size). Lookup entries from the same embedding table are dispersed to different instances.
The limitation of this approach is that the number of instances cannot exceed the number of large embedding tables. For MLPerf DLRM, we only have 16 large embedding tables, which means that we cannot scale to more than 16 instances. To improve scaling, we use vertical split embedding-based model parallelism (Figure 4). In this method, large embedding tables are vertically split into multiple embedding tables with the same entry number as the original. Each table has a subset of the columns in the original table. We then let each model instance hold one of the split tables and use all-to-all communication.

Suppose \( p \) (the number of ranks) is divisible by \( N \) (the number of embedding tables in the model) and the group number is \( g=p/N \). We divide each embedding table into \( g \) tables. There are \( g*N=p \) embedding tables after vertical splitting. We put one embedding table on each model instance, lookup each table with global batch, and then use all-to-all to transpose the lookup entries among instances. After that, we concatenate entries belonging to the same original embedding table, and then go through the upper layers as in the data parallel approach.

Figure 3. Turning small embedding table gradients into dense gradients. All-reduce is used to synchronize the gradients for data parallelism, and all-to-all is used to exchange embedding information for model parallelism.
The advantages of our vertical split embedding-based hybrid parallel approach are as follows:

1. Compared to sparse all-reduce data parallelism, by treating small tables as dense and splitting large sparse tables vertically, we reduce the communication overhead of models with embedding tables. This reduces TTT (time-to-train) and allows more efficient scaling of multiple model instances.

2. DLRM training is a memory-bound workload because of the large embedding tables. In our solution, there is only local copy to the subset of columns of the large embedding table, which reduces the memory requirement. For example, 26 embedding tables require 100+ GB memory when a single node is used for training. When we scale to 64 ranks using the solution described above, there is only a subset of one large table on every rank. With vertical split embedding, the feature size is 32 instead of 128 for single node training. So, we only need about 6 GB for 10 small embedding tables and a large embedding table with a subset of columns of the original large embedding table. Therefore, the vertical split embedding optimization is also a general solution to train workload with oversized embedding tables.

Figure 4. A model with two embedding tables training on four instances. Each table is split vertically into two tables (V: vector length of each row in embedding table). The lookup entries are transposed among all instances, and entries belonging to the same table (same color) are concatenated. This allows us to train a model with two embedding tables on four ranks, which wasn’t previously possible.
Large Batch Size DLRM BFloat16 Training with Split-LAMB Optimizers

To get better scaling efficiency, a layer-wise adaptive large batch optimization technique, called LAMB, is used to enable large batch-size training. We also use a split version of LAMB and SGD to leverage the BFloat16 Intel® DL Boost instruction. This allows scaling to 64 sockets and higher to reduce the TTT for DLRM training. A 32K global batch size is common for DLRM. When we scale to more ranks, the local batch size will be exceedingly small, which means that the local workload cannot saturate the processors. In this case, there is no opportunity to overlap between communication and computation. Therefore, we need to use a larger global batch size when scaling to more ranks.

SGD (stochastic gradient descent) is the default optimizer in the reference code of DLRM. It works well and converges in 0.75 epochs with 64K global batch size, but fails to converge at larger batch size (i.e., 256K). In our solution, we use LAMB (an Adam-based optimizer) to enable 256K global batch size training and achieve convergence in 0.8 epochs for DLRM training. LAMB stores the first- and second-order moments for every weight. Compared to the naive SGD optimizer, LAMB needs 3x the memory footprint. DLRM is memory-bound because of the large embedding tables and because the gradients of the large embedding tables are sparse. To reduce the memory footprint, the LAMB optimizer is only used in the data parallel part of the computation. The sparse embedding table still uses the SGD optimizer.

We use the Intel DL Boost BFloat16 instruction to speed up DLRM training. Master weight is often used to maintain training accuracy with BFloat16. Master weight is a copy of Float32 weight stored in the optimizer to update weight, while a BFloat16 weight converted from master weight is also needed to forward and backward pass. It needs about 1.5x higher memory footprint compared to Float32 training and aggravates the memory bound nature of DLRM. In this part, we use the split optimizer to reduce the memory footprint of BFloat16 training (Figure 5). All inputs parameters in Scope I work with BFloat16 (which are truncated from corresponding Float32 parameters) at forward and backward training stages, and then are fed into BFloat16 operators (InnerProduct, EmbeddingBag) to leverage Intel DL Boost BFloat16. When running into parameter-update stage (SGD optimizer scope), it will pack BFloat16 data in Scope I, with another bottom half data in Scope II (which also exist with BFloat16) into full precision Float32 parameters and do normal calculation in Float32. After each update, it will split Float32 data back to separate BFloat16 representation in Scopes I and II. Hence, the Split-SGD doesn’t involve additional memory overhead for every weight. For Split-LAMB, we use the same method to pack and unpack weight and keep momentums in Float32.
Hyperparameter Optimization Powered by SigOpt

With this combination of innovative data and model parallelization, BFloat16 optimization, and new hybrid split-SGD and LAMB optimizers, it is crucial to run hyperparameter optimization to maximize performance against our metrics. Hyperparameter optimization can be time- and resource-intensive using traditional methods like grid search or random search. We apply a much more sample-efficient search method designed to find the optimal hyperparameters.

We used SigOpt, a leading experimentation platform that combines run tracking with scalable hyperparameter optimization for any type of model (e.g., deep learning, machine learning, high performance computing, and simulation). Acquired by Intel in October 2020, SigOpt allows for the use of any hyperparameter optimization (HPO) method with its scheduler (e.g., random search, grid search, and Bayesian optimization), but also offers a proprietary optimizer that combines the best attributes of a variety of Bayesian and global optimization algorithms. We found SigOpt's optimizer performed best for our purposes.

DLRM training converges in a few iterations and reaches the 0.8025 AUC (area under curve) threshold for 256K global batch size (Figure 6). This exceeds the 0.75 AUC achieved for 32K global batch size. We can see that SigOpt quickly finds the hyperparameter set that meets the threshold and continues to improve beyond our threshold. SigOpt provides a variety of out-of-the-box visualizations, charts, plots, comparisons, and tables in a web dashboard. The SigOpt parameter importance analysis shows the critical parameters for our experiment (Figure 7).
Figure 6. DLRM experiment improvement on AUC score

Figure 7. Parameter importance for DLRM
A Novel Model Parallel Friendly Data Loader

So far, we have described the vertical split embedding table to scale to more ranks and reduce memory footprint. A large global batch size with LAMB can also help to get better scaling efficiency, if everything goes well. A large embedding table needs to lookup global batch size entries with model parallelism, which means that we also need to read global batch size inputs from disk. Therefore, I/O is a potential bottleneck on multi-socket systems. A novel model parallel data loader is used to reduce this overhead. The loader will only read the local batch size inputs, which is a fraction of global batch size inputs, and use all-to-all communication to get global batch size inputs.

A terabyte dataset is used to train the MLPerf DLRM model. The data is row-major and contiguous in memory. There are 40 elements per sample (one label, 13 numerical features, and 26 categorical features) and every element uses four bytes. Numerical features work as inputs of bottom MLP (data parallelism), and categorical features work as inputs for embedding. (Small tables work with data parallelism, large tables work with model parallelism.) For a single-instance case, we need to read local batch size (LBS) samples for every iteration. If we just use the data parallel scale-out solution, every instance just reads LBS samples per iteration, but when we use the hybrid parallel approach, there is only local copy to part of the large embedding tables for every instance. So, global batch size (GBS) embedding indices are needed for large embedding tables in the current instance, and inputs data for every instance.

For LBS inputs, there are 26 categorical features for every sample, which means that the current instance not only reads LBS categorical features for its own large tables, but also LBS categorical features for other model instances. So, a naïve idea is that every instance only reads LBS samples per iteration and uses all-to-all communication to get GBS categorical features for large embedding tables (Figure 8). The inputs for every instance here mean the LBS categorical features for the large embedding tables in the first instance and all rank-0 will be integrated into GBS categorical features. Rank-1 means the LBS categorical features for the large embedding tables in the second instance, and so on. Before looking up embedding tables, all-to-all communication is used to collect GBS categorical features for every instance. If we use $N$ instances, we can save $(N-1)/N$ I/O bandwidth compared to the naïve data loader.
Results

The DLRM scale-out solution is implemented using PyTorch, Intel® Extension for PyTorch (IPEX), and Intel® oneAPI Collective Communications Library (oneCCL). We submit both closed division and open division MLPerf benchmarks (Figure 9). The closed division results (retrieved June 30, 2021) show that it takes about two hours to get convergence if we only use four sockets with 32K GBS on an Intel Xeon Platinum 8380H processor. When we used LAMB to enable 256K GBS and vertical split embedding table to scale to 64 sockets on an Intel Xeon Platinum 8376H processor, it only took 15 minutes to converge (open division results, retrieved June 30, 2021).
Summary

For DLRM MLPerf training, we provide a complete scale-out training solution to resolve the challenge of balancing a mixture of compute-, memory-, memory capacity-, and I/O-bound work. First, we use hybrid parallel to reduce communication cost and memory consumption. Vertical split embedding tables not only helped us to scale to more ranks, it is also a fantastic solution to train any other workloads with oversized embedding tables. Second, a LAMB optimizer enables large batch size training to get better scaling efficiency. At the same time, split optimizer is also the overwhelming choice to leverage Intel DL Boost BFloat16 instruction for training. Finally, a novel model parallel data loader reduces the I/O bandwidth requirement. With this solution, you can train the DLRM MLPerf model in 15 minutes (or less if more sockets are used). At the same time, most techniques used in this article can be generalized to other distributed training approaches.

Figure 9. DLRM training performance for different numbers of sockets. We use one model instance (rank)/socket. The 4-, 8-, and 16-socket results were submitted to the MLPerf closed division. The 32- and 64-socket results were submitted to the open division.
We are excited to present our research on a nonmyopic approach to cost-constrained Bayesian optimization, which was recently published by the Conference on Uncertainty in Artificial Intelligence. This work was done in collaboration with scientists from Facebook and Amazon. In this summary, we'll discuss the motivating factors behind this work.

The Default State of Hyperparameter Optimization: Measuring Progress by Iterations

Nearly all practical hyperparameter optimization packages attempt to determine optimal hyperparameters within a certain number of iterations. Let's see how a user might run HyperOpt, Optuna, SKOpt, and SigOpt with a budget of 100 iterations:

For more complete information about compiler optimizations, see our Optimization Notice.
Most optimizers have essentially the same interface: Plug in the objective you want to maximize/minimize and number of iterations you want to run. Most people take this interface for granted, but is it really the best way to optimize the objective? We argue that by asking for iteration count when we really care about something like cumulative training time, we are leaving performance on the table.

### The Challenge: Varying Costs of Hyperparameter Evaluation

Measuring optimization progress by iterations is reasonable if each evaluation takes the same amount of time; however, this is often not the case in hyperparameter optimization (HPO). The training time associated with one hyperparameter configuration may differ significantly from another. We confirm this in the study cited above, where we considered five of the most commonly used machine learning models:

- k-nearest neighbor (KNN)
- Multilayer perceptron (MLP)
- Support vector machine (SVM)
- Decision tree (DT)
- Random forest (RF)

Taken together, these models comprise the majority of general models that data scientists use. Note that deep learning models have been omitted, though our results hold for those as well.
We trained our five models on the commonly benchmarked OpenML w2a dataset using 5,000 randomly chosen hyperparameter configurations drawn from a standard search space. We then plotted a distribution of the resulting runtimes for each model (Figure 1). As you can see, the training times of each model vary greatly, often by an order of magnitude or more. This is because, for each model, a few hyperparameters heavily influence not only model performance, but also training time (e.g., the layer sizes in a neural network or the number of trees in a forest). In fact, we have found that in nearly all practical applications, evaluation costs vary significantly in different regions of the search space.

Consequently, the cumulative training time spent tuning these hyperparameters is not directly proportional to the number of iterations. In fact, it is entirely possible, according to the histogram above, for one optimizer to evaluate one hyperparameter configuration, for another optimizer to evaluate 100 points, and for both to take an equal amount of time doing so.

The Challenge: Handling a Heterogeneous Cost Landscape

We developed an optimizer that accounts for the varying cost landscape of the objective function and makes intelligent decisions about where to evaluate next. This is known as a cost-aware optimization routine. Instead of budgeting optimization by iteration count, our optimization routine takes into account an optimization budget measured by a cost metric, such as time. For example, instead of asking an optimization routine to compute the optimal hyperparameters in 100 iterations, we can instead ask it to compute the optimal hyperparameters in 100 minutes of training time. We might invoke such an optimization routine in the following way:

\[
\text{optimizer.minimize(func=objective, num\_minutes=100)}
\]

This is an important research problem that the hyperparameter optimization community is actively prioritizing because, in practice, the user often has a cost budget that is expressed in money, time, or some other heterogeneous metric that constrains hyperparameter optimization.

For more complete information about compiler optimizations, see our Optimization Notice.
On the Benefit of Being Cost-Aware

We developed a cost-aware optimization algorithm that we'll call, "cost-aware Bayesian optimization (CA-BO)." To keep things simple, let's consider an example comparing our CA-BO method to Optuna, a great open-source tool from Preferred Networks. Both optimizers are asked to maximize the binary classification accuracy of an XGBoost model in 100 optimization iterations. We see that as iteration continues, Optuna gradually outperforms CA-BO (Figure 2). The performance gap in this case is quite convincing. What might be happening? We would hope CA-BO at least performs as well as Optuna.

This gap exists because CA-BO is cost-aware. It recognizes that the evaluation cost varies and makes decisions accordingly. Let's examine optimization performance when we account for varying evaluation cost, by replacing the x-axis with cumulative training time instead of iterations (Figure 3). We see that our CA-BO performs much better. Optuna eventually outperforms CA-BO, but only after CA-BO finishes running within its time budget (and it takes much longer to achieve comparable accuracy at nearly all times during optimization).
Side-by-side comparison highlights the stark difference between the cost-aware and cost-unaware optimization routines (Figure 4). On the right, we see that CA-BO finds superior hyperparameters within 200 minutes. On the left, we see that it finds worse hyperparameters within 100 iterations, which takes the cost-unaware approach more than 500 minutes to complete.

![Figure 4. Side-by-side comparison of cost-aware and cost-unaware optimizer performance](image)

The picture on the left is the current state of HPO. The picture on the right is what developers trying to move models to production actually care about: a method that makes intelligent decisions based on the metric they care about.

**Conclusions**

CA-BO is a rapidly evolving class of algorithms for HPO. For more information about CA-BO, see our paper, which contains the technical details of our approach. For more information on other optimization techniques and applications, check out the SigOpt research page. Techniques like this are built into the SigOpt optimizer, one of the pillars of the SigOpt Intelligent Experimentation Platform. You can try out the optimizer and the rest of the platform for free by signing up today.
According to Gartner, Inc., graph processing is one of the top 10 data analytics trends for 2021. It is an emerging application area, as well as a necessary tool for data scientists working with linked datasets (e.g., social, telecommunication, and financial networks; web traffic; and biochemical pathways). Graphs in practical applications tend to be large, and they’re getting larger. For example, social networks today can have billions of nodes and edges, so high-performance parallel computing is essential.

To this end, Katana Graph, in collaboration with Intel, has designed a high-performance, easy-to-use graph analytics Python library with (a) highly optimized, parallel implementations of important graph analytics algorithms; (b) a high-level Python interface to write custom parallel algorithms on top of the underlying C++ graph engine; (c) interoperability with pandas, scikit-learn, and Apache Arrow, and tools and libraries in the Intel AI software stack; (d) comprehensive support for extraction, transformation, and loading (ETL) from various formats; and (e) a Metagraph plugin.
This article will cover what's in the library, how to get access to the library, usage examples, and benchmark data to highlight performance.

**Graph Analytics Algorithms in the Library**

The key algorithms that are commonly used in graph-processing pipelines come prepackaged in the Katana library. The algorithms that are currently available are listed below:

- **Breadth-first search**: Returns an oriented tree constructed from a breadth-first search starting at a source node
- **Single-source shortest path**: Computes the shortest paths to all the nodes starting from a source node
- **Connected components**: Finds the components (i.e., groups of nodes) of the graph that are connected internally, but not connected to other components
- **PageRank**: Computes the ranking of nodes in the graph based on the structure of incoming links
- **Betweenness centrality**: Computes the centrality of nodes in the graph based on the number of shortest paths that pass through each node
- **Triangle counting**: Counts the number of triangles in a graph
- **Louvain community detection**: Computes the communities of the graph that maximizes the modularity using the Louvain heuristics
- **Subgraph extraction**: Extracts the induced subgraph of the graph
- **Jaccard similarity**: Computes the Jaccard coefficient of a given node to every other node in the graph
- **Community detection using label propagation**: Computes the communities in the graph using a label propagation algorithm
- **Local clustering coefficient functions**: Measures the degree to which nodes in a graph tend to cluster together
- **K-Truss**: Finds the maximal induced subgraph of the graph that contains at least three vertices where every edge is incident to at least K - 2 triangles
- **K-Core**: Finds the maximal subgraph that contains nodes of degree K or more

More algorithms are being added to the library, and it's easy for users to add their own algorithms, as we'll demonstrate below.

**Getting the Katana Graph Library**

Katana Graph's analytics library is open-source and freely available under the [3-Clause BSD license](https://opensource.org/licenses/BSD-3-Clause). It is available on [GitHub](https://github.com/katanagraph/katana) or easily installed from Anaconda.org:

```
$ conda install -c katanagraph/label/dev -c conda-forge katana-python
```
Using the Katana Graph Library

Katana's Python library supports ETL from various formats, such as adjacency matrices, pandas DataFrames, NumPy arrays, edge lists, GraphML, NetworkX, etc. A few examples are shown below:

```python
import numpy as np
import pandas
from katana.local import Graph
from katana.local.import_data import (
    from_adjacency_matrix,
    from_edge_list_arrays,
    from_edge_list_dataframe,
    from_edge_list_matrix,
    from_graphml)

Input from an Adjacency Matrix

katana_graph = from_adjacency_matrix(
    np.array([[0, 1, 0], [0, 0, 2], [3, 0, 0]]))

Input from an Edge List

katana_graph = from_edge_list_arrays(
    np.array([0, 1, 10]), np.array([1, 2, 0]),
    prop = np.array([1, 2, 3]))

Input from a Pandas DataFrame

katana_graph = from_edge_list_dataframe(
    pandas.DataFrame(dict(source=[0, 1, 10],
                         destination=[1, 2, 0],
                         prop = [1, 2, 3])))

Input from GraphML

katana_graph = from_graphml(input_file)
```
Executing a Graph Analytics Algorithm

The following example computes the betweenness centrality of an input graph:

```
import katana.local
from katana.example_utils import get_input
from katana.property_graph import PropertyGraph
from katana.analytics import betweenness_centrality,
                           BetweennessCentralityPlan,
                           BetweennessCentralityStatistics

katana.local.initialize()

property_name = "betweenness_centrality"
betweenness_centrality(katana_graph, property_name, 16,
                       BetweennessCentralityPlan.outer())
stats = BetweennessCentralityStatistics(g, property_name)

print("Min Centrality:", stats.min_centrality)
print("Max Centrality:", stats.max_centrality)
print("Average Centrality:", stats.average_centrality)
```

Katana's Python library is interoperable with pandas, scikit-learn, and Apache Arrow.

Along with the prepackaged routines listed previously, data scientists can also write their own graph algorithms using an easy Python interface that exposes Katana Graph's optimized C++ engine and its concurrent data structures and parallel loop constructs. The Katana Graph library already contains a breadth-first search implementation, but the following example illustrates how easy it is to implement such algorithms using the API:

```
def bfs(graph: Graph, source):
    """
    Compute the BFS distance to all nodes from source.
    The algorithm in bulk-synchronous level by level.
    :param graph: The input graph.
    :param source: The source node for the traversal.
    :return: An array of distances, indexed by node ID.
    """
    next_level_number = 0

    # The work lists for the current and next levels using a Katana concurrent data structure.
    curr_level_worklist = InsertBag[np.uint32]()
    next_level_worklist = InsertBag[np.uint32]()

    # Create and initialize the distance array.
    # source is 0, everywhere else is INFINITY
```
distance = np.empty((len(graph),), dtype=np.uint32)
distance[:] = INFINITY
distance[source] = 0

# Start processing with just the source node.
next_level_worklist.push(source)

# Execute until the worklist is empty.
while not next_level_worklist.empty():
    # Swap the current and next work lists
    curr_level_worklist, next_level_worklist = next_level_worklist, curr_level_worklist

    # Clear the worklist for the next level.
    next_level_worklist.clear()
    next_level_number += 1

    # Process the current worklist in parallel by applying
    # bfs_operator for each element of the worklist.
    do_all(
        curr_level_worklist,
        # The call here binds the initial arguments of bfs_operator.
        bfs_operator(graph, next_level_worklist, next_level_number, distance)
    )

return distance

# This function is marked as a Katana operator, meaning that it will
# be compiled to native code and prepared for use with Katana do_all.
@do_all_operator()
def bfs_operator(graph: Graph, next_level_worklist, next_level_number, distance, node_id):
    """
The operator called for each node in the work list.

    The initial four arguments are provided by bfs above.
    node_id is taken from the worklist and passed to this
    function by do_all.

    :param next_level_worklist: The work list to add next nodes to.
    :param next_level_number: The level to assign to nodes we find.
    :param distance: The distance array to fill with data.
    :param node_id: The node we are processing.
    :return:
    """
    # Iterate over the out edges of our node
    for edge_id in graph.edges(node_id):
        # Get the destination of the edge
        dst = graph.get_edge_dest(edge_id)
# If the destination has not yet been reached, set its level
# and add it to the work list so its out edges can be processed
# in the next level.
if distance[dst] == INFINITY:
    distance[dst] = next_level_number
    next_level_worklist.push(dst)
# There is a race here, but it's safe. If multiple calls to
# operator add the same destination, they will all set the
# same level. It will create more work because the node will
# be processed more than once in the next level, but it avoids
# atomic operations so it can still be a win in low-degree graphs.

## Metagraph Support

Katana Graph's Python analytics library will be available via a Metagraph plugin. Metagraph provides a consistent Python entry point into graph analysis. One can write a graph workflow using a standard API, and then have that dispatch to compatible graph libraries that plug into Metagraph. Now, the open-source graph community will be able to directly use Katana Graph's high-performance applications. The Metagraph plugin comes in an Anaconda package that can be installed and invoked as follows:

```bash
$ conda create -n metagraph-test -c conda-forge \
    -c katanagraph/label/dev \
    -c metagraph metagraph-katana
```

import metagraph as mg
bfs = mg.algos.traversal.bfs_iter(katana_graph, <start node>)

## How Fast Is the Katana Graph Library?

The Katana library has been extensively benchmarked against other graph analytics frameworks and has consistently shown equivalent or better performance for the GAP Benchmark Suite. Table 1 shows the Katana Graph's performance relative to the GAP reference implementations for various graphs from diverse domains.
### The Parallel Universe

The Katana Graph library has also been shown to perform well on extremely large graphs, such as Clueweb\(^3\) and WDC\(^4\) (at 42 and 128 billion edges, respectively, these are some of the largest publicly available graphs) on recent byte-addressable memory technologies such as Intel\(^\circledR\) Optane\(^\text{TM}\) DC persistent memory\(^5,6\) (Figure 1).

---

### Table 1: Measuring Katana Graph performance using the GAP Benchmark Suite.

This data is taken from Azad et al. (2020). System: dual-socket 2.0 GHz Intel\(^\circledR\) Xeon\(^\circledR\) Platinum 8153 processor (64 logical cores) and 384 GB DDR4 memory. For more complete information about performance and benchmark results, visit [www.intel.com/benchmarks](http://www.intel.com/benchmarks).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Real Graphs</th>
<th></th>
<th>Synthetic Graphs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Web</td>
<td>Twitter</td>
<td>Road</td>
<td>Kron</td>
</tr>
<tr>
<td>BFS</td>
<td>1.6</td>
<td>0.7</td>
<td>1.0</td>
<td>1.3</td>
</tr>
<tr>
<td>SSSP</td>
<td>1.1</td>
<td>1.0</td>
<td>0.8</td>
<td>1.3</td>
</tr>
<tr>
<td>CC</td>
<td>1.4</td>
<td>1.1</td>
<td>1.5</td>
<td>1.1</td>
</tr>
<tr>
<td>PageRank</td>
<td>1.0</td>
<td>1.3</td>
<td>1.5</td>
<td>0.9</td>
</tr>
<tr>
<td>BC</td>
<td>1.3</td>
<td>0.6</td>
<td>0.7</td>
<td>0.8</td>
</tr>
<tr>
<td>TC</td>
<td>1.6</td>
<td>1.4</td>
<td>1.0</td>
<td>1.3</td>
</tr>
</tbody>
</table>

---

**Figure 1.** Katana Graph BFS performance on extremely large graphs. It compares the performance of a single Intel Optane memory-based node to clusters with multiple nodes. Each TACC Stampede Skylake cluster node has two 2.1 GHz Intel Xeon Platinum 8160 processors and 192 GB DDR4 memory. The Cascade Lake servers have two 2.2 GHz 2nd Generation Intel Xeon Scalable processors with 6 TB Intel Optane PMM and 384 GB DDR4 DRAM. The Ice Lake servers have two 2.2 GHz Intel Xeon Platinum 8352Y processors with 8 TB Intel Optane PMM and 1 TB DDR4 DRAM. This chart was compiled with data from references [5] and [6]. For more complete information about performance and benchmark results, visit [www.intel.com/benchmarks](http://www.intel.com/benchmarks).
Where Can I Learn More?

I hope you’re convinced that the Katana Graph library is a versatile, high-performance option for graph analytics. You can learn more about the library, ask questions, post feature requests, etc. at the GitHub site.

References


The R statistical computing package is used in many areas, like IT, finance, e-commerce, healthcare, manufacturing, and so on. This article will show how to boost the performance of R using the Intel® oneAPI Math Kernel Library (oneMKL), but first, let’s explore what R is all about.

R is an open-source programming environment for statistics and data analytics. It is also a domain-specific programming language for statistics (see www.r-project.org for more information). Many R users don’t realize that they can easily boost the performance of their computations significantly by linking to a high-performance math library like oneMKL.

oneMKL contains highly optimized, threaded, and vectorized functions for common mathematical operations that are used for scientific, engineering, and financial applications (Figure 1). It covers dense
and sparse linear algebra (BLAS, LAPACK, PARDISO), fast Fourier transforms, vector math, summary statistics, splines, and much more. It runs optimized code for a given processor automatically without the need to branch code. It is also optimized for single-core vectorization and cache utilization. Finally, it automatically uses parallelism for multi-core CPUs and GPUs, and scales some computations from single systems to clusters.

Figure 1. Mathematical domains covered by oneMKL

oneMKL is part of the Intel® oneAPI Base Toolkit. The Intel® oneAPI HPC Toolkit is also required to link to R. Both packages can be freely downloaded:

- Get the Intel oneAPI Base Toolkit
- Get the Intel oneAPI HPC Toolkit

Linking R to oneMKL confers significant performance advantages, as we'll see below, without requiring developers to change their R code. oneMKL is one layer underneath the R application. It works with the R engine to use appropriate oneMKL functions to improve performance. The oneMKL functions will automatically take advantage of hardware features in Intel® processors like Intel® Advanced Vector Extensions 512 (Intel® AVX-512), Intel Advanced Vector Extensions 2 (Intel AVX2), and Intel Advanced Vector Extensions (Intel AVX). oneMKL is designed to allow developers to focus on their applications without worrying about the underlying hardware. For example, a oneMKL application created on a system that supports Intel AVX will also take advantage of Intel AVX-512 if moved to a system that supports the later extensions.
Linking R to oneMKL is straightforward (see Quick Linking Intel® MKL BLAS, LAPACK to R). (Note that the following instructions are for Linux.) Linking R to oneMKL will redirect appropriate R functions to optimized oneMKL functions; however, it is important to set certain oneMKL environment variables to work with R:

$ export MKL_INTERFACE_LAYER=GNU,LP64
$ export MKL_THREADING_LAYER=GNU

These environment variables set the MKL interface and threading layer to GNU and LP64. To verify that R is linked to oneMKL, run `sessionInfo()` from the R command prompt (Figures 2 and 3).

**Figure 2. Output from sessionInfo() showing R without oneMKL**

**Figure 3. Output from sessionInfo() showing R using oneMKL**
In general, R users don’t have to do anything more than simply link to oneMKL; however, there are a few things they can do to help:

- Although R is single-threaded, oneMKL can run in either single- or multi-threaded mode. oneMKL uses multiple threads by default (i.e., the environment variable MKL_DYNAMIC=TRUE). This is usually best, provided there’s enough work to justify thread creation. In other words, oneMKL works best for large datasets.
- It is important not to oversubscribe the system (i.e., using more threads than available processors). The common practice is to set the number of threads equal to the number of cores in the system.
- This doesn’t always give best performance if there isn’t enough work to saturate available resources, however. In this case, manually setting the number of threads can give better performance (e.g., setting the environment variables MKL_DYNAMIC=FALSE and MKL_NUM_THREADS=4).
- Disable hyperthreading in the BIOS.
- Set the environment variable MKL_VERBOSE=1 to see which oneMKL functions are being called and how many threads are being used.

The R Benchmarks (v2.5) are used to measure the performance improvement from linking R to oneMKL. The benchmark takes 27.9 seconds when R is not linked to oneMKL:

The benchmark takes 2.8 seconds when R is linked to oneMKL:

This is a 9.8x speedup just by linking R to oneMKL. No R code modifications were needed.
Finding the index of a target element (minimum, maximum, or their absolute values) is used in many applications. In this article, I use the Intel® Advanced Vector Extensions 512 (Intel AVX-512) compiler intrinsics to accelerate this operation (henceforth referred to as “Maxloc”) and benchmark the performance on Intel® Xeon® Scalable processors. Intel AVX-512 provides a broad set of instructions that facilitates single instruction, multiple data (SIMD) execution. It is available on all Intel Xeon Scalable processors and uses 512-bit vector registers to operate on wider execution units for maximum efficiency. By careful application of Intel AVX-512, the number of instructions and comparisons needed to identify the index of the target element is minimized, resulting in a significant speed-up.

Maxloc is a common search operation performed on arrays (e.g., the NumPy argmax function, TensorFlow
GlobalMaxPool1D, and oneMKL amax functions). I will be using the Intel AVX-512 SIMD API (popularly known as vector intrinsics) provided by the Intel® C/C++ Compiler Classic. (Readers are also referred to my previous article, Optimization of Scan Operations Using Explicit Vectorization.) The following sections will illustrate the implementation of Intel AVX-512 Maxloc using Intel AVX-512 vector intrinsics.

Baseline Implementation

Code listing 1 shows a baseline implementation consisting of a single pass, scalar version of Maxloc. As can be seen from the code, a vector containing n elements requires n comparison operations. We visit the input vector only once, and store the index location corresponding to the maximum value. The instructions generated in the baseline implementation are shown in Table 2 and will be discussed in the performance evaluation section.

Intel AVX-512 SIMD Implementation

Table 1 lists the Intel AVX-512 SIMD instructions used in my approach. Figure 1 shows a visual representation of the operations performed by vmaxpd, vcmpps, and vblendmps instructions for a sample register state.

<table>
<thead>
<tr>
<th>Op Name</th>
<th>Instruction</th>
<th>AVX512 Vector Intrinsics API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD Load</td>
<td>vmovups</td>
<td>_m512d_mm512_loadu_ps</td>
<td>Load 512-bits (composed of 16 packed single-precision (32-bit) floating-point elements) from memory into dst.</td>
</tr>
<tr>
<td>SIMD Max</td>
<td>vmaxaps</td>
<td>_m512d_mm512_max_ps</td>
<td>Compare packed single-precision (32-bit) floating-point elements in a and b, and store packed maximum values in dst.</td>
</tr>
<tr>
<td>SIMD Compare</td>
<td>vcmpps</td>
<td>_mm512_cmpps</td>
<td>Compare packed single-precision (32-bit) floating-point elements in a and b based on the comparison operand specified by imm8, and store the results in mask vector k.</td>
</tr>
<tr>
<td>SIMD Blend</td>
<td>vblendmps</td>
<td>_m512_mm512_mask_blend_ps</td>
<td>Blend packed single-precision (32-bit) floating-point elements from a and b using control mask k, and store the results in dst.</td>
</tr>
<tr>
<td>SIMD Broadcast</td>
<td>wpbroadcastd</td>
<td>_m512_mm512_mask_set1_epi32</td>
<td>Broadcast 32-bit integer a to all elements of dst using wmmask k (elements are copied from src when the corresponding mask bit is not set).</td>
</tr>
<tr>
<td>SIMD Extract</td>
<td>wextractf32x8</td>
<td>_m526_mm512_extractf32x8_ps</td>
<td>Extract 256 bits (composed of 8 packed single-precision (32-bit) floating-point elements) from a, selected with imm8, and store the result in dst.</td>
</tr>
</tbody>
</table>

Table 1. Intel AVX-512 SIMD instructions used in Maxloc
For Maxloc, I unroll the loop and apply a series of \texttt{vmaxpd} instructions to obtain a sequence of 16 FP32 (32-bit floating-point) maximum values, represented as 512-bit Intel AVX-512 registers. In addition to that, I keep track of the indices of the maximum values as the input data is processed. This is achieved by doing an additional comparison operation (between the current and previous iteration maximum values) and using the results to blend the latest maximum values and their corresponding unrolled block ID. The key steps of the algorithm are as follows:

1. **Main block:**
   a. Each iteration loads 64 input elements into four Intel AVX-512 vector registers and does pairwise comparison to identify the maximum value in each quadruplet to form a current max register representing the max values in offsets of 16.
   b. The previously populated max values are compared to form a 16-bit mask (with each bit representing a FP32 element in an Intel AVX-512 register). The mask bit is set if the quadruplet maximum value found in the current iteration is greater than the previous quadruplet max value.
   c. Once this mask is calculated, it is used to blend the values from both the current and previous max Intel AVX-512 registers to form the new quadruplet of max values in the register for use in future iterations.
   d. To track the indices, the same mask is reused to set the current loop index (unrolled block ID) to a register of indices representing the block at which the corresponding maximum value is located.
   e. The previous steps form the main loop and are repeated for all input elements to find the 16 max values among quadruplets (at indices \{0,16,32,48\}, \{1,17,33,49\}, ...) and their corresponding loop/block IDs \{0, 64, 128, 192, ...\).
2. **Reduction and target block:** After the main loop, a single reduction on an Intel AVX-512 register finds the maximum among the 16 values from the main block. The maximum value found from the reduction step is used to identify (in the form of a 16-bit mask) where among the 16 locations in the Intel AVX-512 register the maximum value is encountered. Each bit of the 16-bit result mask represents a selection of int32 elements into the Intel AVX-512 block-indices register. This int32 value represents the iteration ID where the maximum value is encountered. Now, depending on whether the maximum value is encountered at one or more locations, the following two methods identify the first block containing the maximum value.

3. **Single instance:** In this case, the maximum value is found only once in the dataset, which implies a single set bit in the 16b result mask. This identifies the corresponding int32 value from the Intel AVX-512 block-indices register. For example, when the mask is 0000 0000 1000 0000, the value at the seventh int32 element in the Intel AVX-512 indices register represents the block ID where the maximum value is found.

   **Offset into block:** One of the unique features of this implementation is that we can use the inherent pattern of the \texttt{vmaxps} instruction to identify the target index without reading the whole block (size of N\_UNROLL). Let’s denote Blk\_id as the int32 value from the Intel AVX-512 indices register corresponding to the 1-bit set in the 16-bit result mask, M\_x as the bit location (0 to 15) where 1-bit is set in 16-bit result mask. From Figure 2, we observe that \texttt{vmaxps} compares values that are at offsets of 16 elements. Hence, we can conclude that index of maximum value has occurred at one of the following four locations: (Blk\_id + M\_x), (Blk\_id + M\_x + 16), (Blk\_id + M\_x + 32), (Blk\_id + M\_x + 48) when N\_UNROLL is 64.

4. **Duplicates:** If the number of 1-bits in the result mask is greater than one, then the maximum value is encountered at more than one location in the dataset. So, we find the minimum of all the int32 values corresponding to the 1-bits set in the mask from the Intel AVX-512 indices register. We again compare the minimum index against the Intel AVX-512 indices register to know if the maximum value is encountered more than once within an unrolled block. If true, we read the whole block in chunks of 16 elements to identify the first index where the maximum value is encountered. Otherwise, we use the approach described in Step 3.

**Figures 2 and 3** show the visual representation of main block computations and reduction sequence on Intel AVX-512 registers. **Code listing 2** shows the implementation of this algorithm.

![Figure 2. Visual representation of the main block of AVX-512 + Index Tracking implementation of Maxloc](image)
Figure 3. Instruction sequence to reduce Intel AVX-512/ZMM register to find the maximum value

Code listing 2. Implementation of Maxloc using Intel AVX-512
Performance Evaluation

The Intel AVX-512 Maxloc implementation is compared to the baseline implementation shown in Code listing 1 using the following experimental protocol:

- GCC (8.4.1) and Clang (11.0.0) are used with the -O3 -march=icelake-server -mprefer-vector-width=512 flags to compile the baseline implementation. ICC (v19.1.3.304) is used to compile the AVX-512 implementation.
- The same input data is used in all benchmarks. The input data consists of values in ascending order (i.e., the maximum value is in the last index location). No performance differences were observed when random values were used.
- The input size is varied from 1,024 to 4,194,304 elements (16 MB with FP32 elements). This allows us to evaluate performance when the data fits into different cache hierarchies (L1D, L2, and L3).
- An Intel Xeon Platinum 8368 processor (3rd Gen Intel Xeon Scalable processor) is used for benchmarking. It has 38 cores per socket, 48 KB L1D cache, 1280 KB L2, and 57 MB L3 per socket.
- Performance is measured for one thread pinned to a single core. The average time elapsed for 100 iterations of each problem size is reported. Memory is allocated on 64B aligned boundaries.
- In the AVX-512 implementation, the main block is explicitly unrolled by 128 elements.

Table 2 shows the instruction sequence generated by GCC, Clang, and explicit Intel AVX-512 implementation compiled with ICC (main block only due to space constraints). Figure 4 shows the performance of Maxloc with GCC, Clang, and explicit Intel AVX-512 methods. Figure 5 shows the speed-up of the AVX-512 implementation over GCC. From the performance charts, we can make the following observations:

1. In the baseline implementation, performance is determined by the automatic vectorization ability of the compilers (GCC, Clang). As observed in the disassembly of object code generated in Table 2, neither GCC nor Clang is able to vectorize the computations, instead relying on scalar instructions (vmovss, vucomiss, vmaxss, cmov) that operate on a single FP32 element; hence, the "ss" suffix (stands for "single, single-precision") in the instruction name. Interestingly, even though Clang unrolls the loop by eight elements, it still processes the eight elements in serial order by emitting the same single, scalar instructions as GCC.

2. In contrast, the Intel AVX-512 implementation works on 16 FP32 elements stored in 512-bit wide ZMM registers using Intel AVX-512 instructions (as seen in the third column of Table 2). The Intel AVX-512 instructions map to the intrinsics in Code listing 2.

3. The Intel AVX-512 implementation of Maxloc delivers superior performance (Figure 5). The average speed-up for sizes that fit in L1, L2, and L3 caches are 18x, 40x, and 13x, respectively. As we hit L3 cache, the latency to load data into registers increases, so performance gains from Intel AVX-512 vectorization diminishes. For GCC and Clang, the lack of SIMD is reflected in the poor performance for all the problem sizes across caches. Even though Clang does unroll by eight elements, its performance is identical to GCC.
<table>
<thead>
<tr>
<th>Baseline with GCC</th>
<th>Baseline with Clang</th>
<th>AVX512 with ICC</th>
</tr>
</thead>
<tbody>
<tr>
<td>20: mov %rcx,%rdx</td>
<td>42: vmovd %m128d,%m128d</td>
<td>20: vmovups %m128d,%m128d</td>
</tr>
<tr>
<td>23: vmoveq %rsi,%rdx,%xmm1</td>
<td>45: vmovq %m128d,%mm128d,%mm128d</td>
<td>26: vmovups %m64,%mm0</td>
</tr>
<tr>
<td>24: vcmpeq %xmm0,%xmm1</td>
<td>47: vcmovq %mm0,%mm1</td>
<td>2d: vmovups %m64(%rsi),%mm3</td>
</tr>
<tr>
<td>25: vcomiss %xmm0,%xmm1</td>
<td>49: vmovaps %m128d,%mm0</td>
<td>2f: vmovups %m64(%rsi),%mm4</td>
</tr>
<tr>
<td>26: lea 0x1(%rdx),%rcx</td>
<td>51: vcomiss %mm0,%mm1</td>
<td>34: vmovups %m128d,%mm5</td>
</tr>
<tr>
<td>27: vcmov %rcx,%eax</td>
<td>52: vcomiss %mm0,%mm1</td>
<td>3b: vmovaps %m128d,%mm6</td>
</tr>
<tr>
<td>30: vcmov %xmm0,%xmm1</td>
<td>53: vcmovd %m128d,%m128d</td>
<td>42: vmovaps %m128d,%mm6</td>
</tr>
<tr>
<td>34: cmp %rcx,%edi</td>
<td>54: cmp %m128d,%m128d</td>
<td>49: vmovaps %m128d,%mm7</td>
</tr>
<tr>
<td>37: jne 20</td>
<td>56: cmp %m128d,%m128d</td>
<td>50: vmovaps %m128d,%mm7</td>
</tr>
<tr>
<td>3a: jne 20</td>
<td>57: cmp %m128d,%m128d</td>
<td>57: vmovaps %m128d,%mm8</td>
</tr>
<tr>
<td>3b: jne 20</td>
<td>59: cmp %m128d,%m128d</td>
<td>5d: vmovaps %m128d,%mm9</td>
</tr>
<tr>
<td>3c: jmp 20</td>
<td>63: cmp %m128d,%m128d</td>
<td>63: vmovaps %m128d,%mm10</td>
</tr>
<tr>
<td>3e: jle 20</td>
<td>69: cmovpaps %m128d,%m128d</td>
<td>69: add %m128d,%m128d</td>
</tr>
<tr>
<td>3f: jne 20</td>
<td>70: vcmppq %m128d,%m128d</td>
<td>77: vpbroadcast %eax,%mm1</td>
</tr>
<tr>
<td>40: cmp %rcx,%eax</td>
<td>7d: add %m128d,%m128d</td>
<td></td>
</tr>
<tr>
<td>4a: cmovb %m128d,%m128d</td>
<td>82: vblendvps %m128d,%m128d,%m128d</td>
<td></td>
</tr>
<tr>
<td>4b: cmp %m128d,%m128d</td>
<td>8a: jle 20 &lt;max_idx_tracking+0x20&gt;</td>
<td></td>
</tr>
<tr>
<td>4c: cmovbe %m128d,%m128d</td>
<td>8d: jle 20 &lt;max_idx_tracking+0x20&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Disassembly of Maxloc with GCC, Clang, and ICC
Conclusion

These results demonstrate the performance advantage of Intel AVX-512 instructions when applied to the Maxloc operation. Explicit AVX-512 vectorization delivers superior performance to GCC and Clang. I hope this demonstration will motivate you to explore Intel AVX-512 opportunities in your compute-intensive code.