Supply Chain Optimization at Enterprise Scale

Optimizing Transformer Model Inference on Intel® Processors

Optimize Utility Maintenance Prediction for Better Service
## Contents

### LETTER FROM THE EDITOR

#### Supply Chain Optimization at Enterprise Scale
Leveraging Open-Source AI Technologies with Red Hat OpenShift* Data Science and Intel® Architecture

#### Optimizing Transformer Model Inference on Intel® Processors
Fusing Operations to Take Advantage of the oneAPI Deep Neural Network (oneDNN) Library

#### Optimize Utility Maintenance Prediction for Better Service
Accelerate Machine Learning with the Intel® AI Analytics Toolkit

#### Accelerating Artificial Intelligence with Intel® End-to-End AI Optimization Kit
Democratizing AI with Intel-Optimized Software

#### Solving Heterogeneous Programming Challenges with Fortran and OpenMP*
Expressing Heterogeneous Parallelism Using Open, Standard Programming Languages

#### Solving Linear Systems Using oneMKL and OpenMP* Target Offloading
Accelerate Linear Algebra Computations with Standard, Open Approaches

#### Device Discovery with SYCL*
Learning about the Accelerators in Your System

#### Zero in on Level Zero
An Open, Backend Approach to Compute Anywhere
Cross-Vendor Heterogeneous Computing

My teammates have been busy lately experimenting with different aspects of oneAPI, all of which demonstrate its promise. Tony Mongkolsmai asked, “Do I really need to learn CUDA*, ROCm*, and SYCL* to program for NVIDIA*, AMD*, and Intel® accelerators?” Then he proceeds to answer this question by showing how to compile and run the same SYCL code on GPUs from all three vendors, even running on Intel® and Nvidia GPUs at the same time. Next up, our editor emeritus, James Reinders, introduces a collection of resources to Migrate from CUDA* to C++ with SYCL*, complete with code examples and self-guided tutorials to help free your code from the constraints of vendor-specific tools and accelerators. Finally, Guy Tamir continues to release videos for his oneAPI Basics Training Series, which cover all aspects of oneAPI, from direct programming with SYCL, to API-based programming with component libraries like the oneAPI Deep Neural Network Library (oneDNN). Guy also covers data science applications from a oneAPI perspective.

Speaking of data science, this issue is full of AI-related content, starting with our feature article: Supply Chain Optimization at Enterprise Scale. This article was coauthored by Ted Jones and Karl Eklund from Red Hat, and Karol Brejna and Piotr Grabusznyski from Intel. It describes how to leverage open-source AI technologies using the Red Hat OpenShift* Data Science platform with Intel-optimized software. This is followed by Optimizing Transformer Model Inference on Intel® Processors, which describes several optimizations to Google’s popular Transformer model for natural language processing. Next, Optimize Utility Maintenance Prediction for Better Service describes one of many practical AI reference kits developed in collaboration with Accenture. Accelerating Artificial Intelligence with Intel® End-to-End AI Optimization Kit shows how Intel-optimized software is democratizing AI.

From data science, we turn our attention to heterogeneous parallel programming using Fortran and the OpenMP* target offload API. Solving Heterogeneous Programming Challenges with Fortran and OpenMP* describes several compiler directives to offload parallel loops to an accelerator and move data between host and device memory, and then closes with a brief overview of standard Fortran DO CONCURRENT loops. Solving Linear Systems Using oneMKL and OpenMP* Target Offloading shows how to dispatch oneMKL functions to an accelerator. The latter is a follow-up to my previous article Accelerating LU Factorization Using Fortran, oneMKL, and OpenMP* in The Parallel Universe (Issue 51),
but this time we go from a conceptual analysis of host-device data transfer to direct measurement and analysis of the performance advantage of minimizing data movement.

Finally, we close this issue with two oneAPI articles: Device Discovery with SYCL* and Zero in on Level Zero. In the former, John Pennycook and I show how to use the SYCL device discovery API to determine what accelerators are available in a system and to query their characteristics. The latter describes the Level Zero hardware abstraction layer that makes oneAPI so powerful.

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
April 2023
Code for the Future.
Grow beyond proprietary boundaries.

Expand your code’s reach with a single, open programming model that supports multiple languages to deliver heterogeneous computing performance.

Rooted in open standards, oneAPI offers cross-architecture libraries, compilers and tools that open your code to more hardware choices—for unparalleled performance.

Discover oneAPI ➔
Supply Chain Optimization at Enterprise Scale

Leveraging Open-Source AI Technologies with Red Hat OpenShift* Data Science and Intel® Architecture

Ted Jones, Senior Solution Architect, and Karl Eklund, Principal Architect, Red Hat
Karol Brejna, Cloud Solutions Architect, and Piotr Grabuszynski, Cloud Solutions Engineer, Intel Corporation

Broken supply chains, restrictions related to COVID-19, and unfavorable economic conditions are just some of the challenges retailers are facing. At each stage of the business process, entrepreneurs must make decisions that determine the revenue, competitiveness, or future path of the company. Developing the best solutions is time-consuming, demanding, and error-prone.

Making decisions should be supported by historical data. However, it is difficult for a human to analyze huge amounts of data from thousands of orders and deliveries. The decision-making process should, therefore, be supported or completely replaced by automatic mechanisms using machine learning (ML). ML is a process where a machine analyzes and gets better (learns) with the more data it sees. Millions of entries are not a problem. Based on the relationships found between the data, a mathematical model is created, which is later used to predict behavior, results, prices, delivery times, etc.
Solving a problem using ML requires looking at it from a different angle. For the company’s processes to be supported by artificial intelligence algorithms, appropriate models must first be prepared. This, however, requires the use of appropriate tools. That’s why Intel and Red Hat are constantly working on new solutions to facilitate ML-related processes. Products such as Red Hat OpenShift* Data Science improve data transport, data mining, training, process automation, and model deployment. At the same time, Intel offers optimized hardware and software such as Intel® Extension for Scikit-learn* (a part of Intel® AI Analytics Toolkit) to improve computational efficiency.

**Problem Statement**

One of the growing problems is delivery date estimation. Supply shortages can increase costs while reducing earnings. That is why it is important to control the processes and manage the risks associated with them. In the example presented below, we show how to use historical data to predict whether a delivery will be delayed. A high probability of a delay may, for example, result in a decision to launch additional stock from the warehouse, use a substitute part, or even change the supplier.

**Approach Description**

Late delivery prediction is a type of classification problem. In this case, the goal is to predict whether delivery will be late or on time based on a set of inputs, such as past delivery history or traffic data. Those features are used to train a model. Once trained, it can be used to make predictions on new, previously unseen data. This approach allows using one simple, fast model that “scans” all orders for potential delays. Risky orders can be subjected to further analysis (e.g., using other algorithms that indicate a potential reason for the delay).

**Technologies**

**Red Hat OpenShift Data Science**

Red Hat OpenShift Data Science is a service for data scientists and programmers of intelligent applications, available as a self-managed or managed cloud platform. It offers a fully supported environment where ML models may be developed, trained, and tested quickly before being deployed in a real-world setting. Teams can deploy ML models in production on containers—whether on-premises, in the public cloud, in the data center, or at the edge—thanks to the ease with which they can be exported from Red Hat OpenShift Data Science to other platforms.

There are many benefits to using Red Hat OpenShift Data Science for ML. The platform includes a wide range of commercially available partners and open-source tools and frameworks—such as Jupyter Notebooks*, TensorFlow*, PyTorch*, and scikit-learn*—for data scientists to use in their workflows. Red Hat OpenShift Data Science provides a secure and scalable environment.
Intel® AI Analytics Toolkit

Intel AI Analytics Toolkit provides a set of Intel-optimized AI software, libraries, and frameworks, such as XGBoost®, scikit-learn, TensorFlow, and PyTorch. It helps to accelerate end-to-end data science and analytics pipelines on Intel® architectures. The toolkit is intended to speed up and simplify the creation of AI applications. It also supports model deployment across multiple platforms.

The Synergy

To develop the example shown below, we used Intel AI Analytics Toolkit and the Jupyter Notebook kernel built into Red Hat OpenShift Data Science. This is a simple and elegant way to provide a stable development environment for data science and analytics.

Reference Implementation

The process of creating a ML solution typically involves several steps:

- Define the problem
- Data acquisition
- Prepare your environment and choose tools
- Data preprocessing and feature engineering
- Preparing the training data
- Choosing the best ML algorithm
- Training
- Model validation
- Model distribution and deployment

Define the Problem

First, decide what type of problem you want to resolve. The likelihood of finding a solution decreases with the problem's complexity. Although this isn't a general rule, we should always start by trying to simplify the issue. In the example, we need a simple solution that indicates potential delivery issues before the delivery even starts.

Data Acquisition

When you know what kind of answer you need, look for the data that might contain it. Sometimes, it requires looking into multiple databases or files and joining them together into one, unified set. (All examples in this article are based on publicly available, open-source data from the Kaggle® platform.) In the following example, we assume that the company has a CSV file with historical records that contains the following features:

© Intel Corporation, Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.
• Order ID
• Order Date
• Delivery Date
• Shipping Mode
• Ordering Party (ID, name)
• Customer Segment (Consumer, Corporate...)
• City, State, and Country of delivery
• Market Symbol (US, APAC...)
• Region
• Product (ID, category)
• Order (price, quantity, discounts, delivery costs)
• Order Priority

Figure 1 shows a slice of the data.

There’s also a Service Level Agreement (SLA) signed with a shipping company. The delivery time should be in line with the provisions contained therein:

• Same-day delivery for packages with a “Ship Mode” equal to “Same Day”
• Next-day delivery for “First Class”
• Three-day delivery for “Second Class”
• Five-day delivery for “Standard Class”
Prepare Your Environment and Choose Tools

Use tools supporting the development process, such as Jupyter Notebooks, to facilitate work on the project. The Red Hat OpenShift Data Science UI allows launching such an environment quickly (Figure 2). Intel-optimized software is already integrated. Therefore, creating and managing an environment will be easy.

Choose an AI framework for the solution. Scikit-learn, a popular open-source ML library, provides a wide range of algorithms for supervised and unsupervised training. The Intel Extension for Scikit-learn improves its performance on Intel® hardware (see Intel Gives scikit-learn the Performance Boost Data Scientists Need). Enabling it inside an Intel AI Analytics Toolkit container is as easy as adding two lines of code:

```python
from sklearnex import patch_sklearn
patch_sklearn()
```

Our example is based on Red Hat OpenShift Data Science with Intel Extension for Scikit-learn libraries installed. The architecture of the proposed solution is presented in Figure 3.
Data Preprocessing and Feature Engineering

Having all data collected, decide how to handle them. Preprocessing involves cleaning and formatting the dataset. This may include removing missing values and inconsistencies, dealing with outliers, and scaling the set. Then, decide what features to use during training. Feature engineering involves the selection and transformation of the data into new features. The point is to better represent patterns and relations in the data. In our example, there’s no straightforward answer to our problem. However, knowing the order date, delivery date, shipping mode, and SLA conditions, we can calculate the delay.

First, we read the data and store them in the pandas DataFrame format:

```python
df = pd.read_csv(DATA_PATH, encoding='latin-1', parse_dates=True)
```

We need to convert the date to a more convenient format, then calculate the delivery time in days:

```python
df['Order Date'] = pd.to_datetime(df['Order Date'], format='%d-%m-%Y')
df['Ship Date'] = pd.to_datetime(df['Ship Date'], format='%d-%m-%Y')
df['Delivery Time'] = (df['Ship Date'] - df['Order Date']).dt.days
```

Having this data and knowing the SLA, we can calculate the delay. We create a new column with the labels: delayed (1) or on time (0):
The “Delay” value is the answer. In this case, there are 31,741 deliveries on time and 19,549 delayed deliveries. The “Delay” column should be used as the target (i.e., the value to predict) during training.

Preparing Data for Training

Previous processes focused on building a collection that contains the necessary features. More operations—like data balancing, categorization, or shuffling—might still be needed to produce a valid model. Then we need to divide the data into training and test sets. In our example, the ratio of delayed to non-delayed deliveries is about 2:3, which is acceptable. However, we must categorize the data:

```python
df['SLA'] = df['Ship Mode'].map({'Same Day': 0, 'First Class': 1, 'Second Class': 3, 'Standard Class': 5})
df['Delay in Days'] = (df['Delivery Time'] - df['SLA']).map(lambda d: d if d > 0 else 0)
df['Delay'] = df['Delay in Days'].map(lambda d: 1 if d > 0 else 0)
df['Delay'].value_counts()
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31741</td>
</tr>
<tr>
<td>1</td>
<td>19549</td>
</tr>
</tbody>
</table>

Name: Delay, dtype: int64

Now, we randomly split the data into training and testing sets.
Choosing the Best Algorithm

Depending on the problem, a supervised or unsupervised algorithm can be used. It is important to consider the type of data, goal, performance, and accuracy while making this decision. To evaluate and compare algorithms, we can use known metrics, like prediction accuracy or Area Under Curve (AUC). AUC is calculated as the area under the plot of true positive vs. false positive predictions. Therefore, it is always a number between 0 and 1 (higher is better).

We'll do a quick check to see which algorithm is best for our application. We train and evaluate models from six different classifiers: Gaussian Naive Bayes, Random Forest, Light Gradient Boosting Machine, Decision Tree, Extreme Gradient Boosting, and KNeighbors. For each algorithm, we calculate the mean AUC and time across 20 iterations (Figure 4). The best model for our data is Random Forest, so that's what we'll use.

```python
m = np.random.rand(len(pds)) < 0.8
ds_train = pds[m]
ds_test = pds[~m]

from statistics import mean

NUMBER_OF_ITERATIONS = 20
time_results = {}
roc_auc_results = {}

for _ in range(NUMBER_OF_ITERATIONS):
    models = [
        ('NB', GaussianNB()),
        ('RF', RandomForestClassifier()),
        ('lgbm', lgb.LGBMClassifier()),
        ('Dtree', DecisionTreeClassifier()),
        ('XGB', xgb.XGBClassifier()),
        ('KN', KNeighborsClassifier()),
    ]
    for name, model in models:
        start_time = time()
        kf = KFold(shuffle=True, random_state=7919)
        cv_result = cross_val_score(model, ds_train_features, ds_train_target, cv=kf,
                                    scoring='roc_auc')
        stop_time = time()
        mean_cv_results = mean(cv_result)
        print(name, cv_result, mean_cv_results)
        total_t = stop_time - start_time
        try:
            time_results[name].append(total_t)
        except KeyError:
            time_results[name] = [total_t]
        try:
            roc_auc_results[name].append(mean_cv_results)
        except KeyError:
            roc_auc_results[name] = [mean_cv_results]
```
Training
ML training finds patterns in the data that map the input features to the target. The resulting model contains those dependencies, so it can try to predict future values. While developing the solution, depending on the selected algorithm and library, we can change the settings of a training process. Tuning those settings may improve or worsen the model, so it is worth experimenting with different values. In our example, we set the class_weight parameter to compensate for the slight imbalance in the data. Besides that, training with the default parameters gave the best results.

```python
crf = RandomForestClassifier(
    class_weight={0:2, 1:3}
)
crf.fit(ds_train_features, ds_train_target)
```

Validating the Final Model
Validating a ML model is the process of evaluating its accuracy on previously unseen data. It is a crucial step because it enables data scientists to find and fix problems before deploying the model in a real-world setting. Too low or very high accuracy may indicate potential issues, such as too little data, lack of important features, data leakage, or under- or over-fitting. For the basic evaluation, you can use the accuracy and AUC metrics mentioned above. The code we use to start the validation process and collect data is provided below:

```python
ds_test_features = ds_test[FEATURES]
ds_test_target = ds_test[TARGET]

preds = crf.predict(ds_test_features)
preds_prob = crf.predict_proba(ds_test_features)[:, 1]

acc_score = accuracy_score(ds_test_target, preds)
auc_score = roc_auc_score(ds_test_target, preds_prob)

print(f'acc: {acc_score} | auc: {auc_score}')
```
We achieved an accuracy of about 86%, and the AUC is above 0.93. Your result might vary slightly because of a random factor in the data preparation phase (e.g., shuffling).

We do a sanity check by calculating the baseline for our solution. We simulate a model that answers "0" every time and calculate its accuracy.

```python
accuracy_score(ds_test_target, np.zeros_like(ds_test_target)), roc_auc_score(ds_test_target, np.zeros_like(ds_test_target))
```

(0.6153169873986519, 0.5)

The base accuracy is about 62%, so we observe significant improvement compared to the baseline.

We can use charts to visualize the results. One of them is a receiver operating characteristic (ROC) curve. **Figure 5** presents the ROC curve for our experiment. As you might have guessed, the area under this curve is the AUC value.

```python
%matplotlib inline

fpr, tpr, thresholds = roc_curve(ds_test_target, preds_prob)
plt.plot(fpr, tpr)
plt.show()
```
We can also visualize what features cause the biggest output changes (Figure 6).

```python
importances = rfc.feature_importances_
std = np.std([tree.feature_importances_ for tree in rfc.estimators_], axis=0)
forest_importances = pd.Series(importances, index=FEATURES)
fig, ax = plt.subplots()
forest_importances.plot.bar(yerr=std, ax=ax)
fig.tight_layout()
```
Running Inference: Simple Usage

For a quick sanity check, try to use the model inside the same Jupyter Notebook. We choose one row from the validation data, and prepare a sample:

```plaintext
INPUT_DATA = {
    'Customer ID': 'AA-00000',
    'Order Date': '12-07-2022',
    # ...
    'Quantity': '12',
    'Order Priority': 'Medium',
}
```

To use our model, we need to transform input data the same way we did for the training phase. It’s important that we use the same categories that were used during training! For example, if “New York” was mapped to 129, we need to use the same number now.
The Parallel Universe

```python
INPUT_DATA['Postal Code'] = float(INPUT_DATA['Postal Code'])

input_d = pd.DataFrame(data={k: [v] for k, v in INPUT_DATA.items()})

udf = pd.DataFrame()

# SLA
udf['SLA'] = input_d['Ship Mode'].map({'Same Day': 0,
                                       'First Class': 1,
                                       'Second Class': 3,
                                       'Standard Class': 5,})

# Month
udf['Month'] = pd.to_datetime(input_d['Order Date'], format='%d-%m-%Y').dt.month

# Ship Mode-cat
for c in CATEGORIZE:
    try:
        udf[f'{c}-cat'] = pd_categories[c].categories.get_loc(INPUT_DATA[c])
    except KeyError:
        udf[f'{c}-cat'] = -1

udf.head()
```

Run a prediction:

```python
if_delay = not rfc.predict(udf)[0]
proba_of_delay = rfc.predict_proba(udf)[0,1][0]
proba_of_delay = (1 - proba_of_delay if if_delay else proba_of_delay) * 100

print('We expect ' + ('no' if if_delay else '') + f' delay (with probability ~{proba_of_delay:.0f}%)
```

We expect no delay (with probability ~93%)
Concluding Remarks

Entrepreneurs nowadays face difficulties such as broken supply chains, COVID-19-related constraints, and adverse economic conditions. Retailers must make choices at every level of the business process that will affect revenue, competitiveness, or future course of the company’s development. ML can significantly simplify the decision-making process and improve its accuracy. The example presented in this article shows how to detect delays in retail deliveries.

Using techniques and algorithms, based on historical data, companies can create solutions that warn of potential delivery delays. This approach allows us to scan all deliveries quickly and with high accuracy. Problematic orders might be analyzed further. For example, you could try to answer questions such as how long the delay will take, what factors will affect it the most, or which supplier will be able to provide parts or services on time. In further analysis, you can use different algorithms depending on the product, category, or data provided by the supplier and produce multiple models to increase prediction accuracy.

Red Hat OpenShift Data Science and Intel Extension for Scikit-learn are a powerful combination that can be used to improve data science and analytics processes. Thanks to the Jupyter Notebook built into Red Hat OpenShift Data Science with a kernel containing Intel-optimized libraries, preparation of the environment is quick and hassle-free.
The Transformer model is one of the most popular models in natural language processing (NLP) (Figure 1). Since its publication by Google in 2017, it has been adopted by many other NLP models, and even extended to non-NLP models. Language translation is a common use of Transformer, largely replacing the LSTM models that were used previously, because of its better accuracy and parallelism. The BERT model is based on Transformer, so optimizing Transformer can improve many NLP and non-NLP models.
Starting with the Transformer training and evaluation code from MLCommons (v0.5), Intel has optimized the model for inference on Intel® Xeon® Scalable processors. These optimizations achieve significant performance improvement for both inference throughput and latency. The optimized model has been added to the Model Zoo for Intel® Architecture.

Training Models

Before attempting to optimize for inference, we must first get a trained model. We started with the MLPerf* Transformer model (v0.5), trained to the accuracy suggested by MLPerf. We obtained checkpoints for the model, which can be further trained for slightly better accuracy, if needed. Although inference can be run from the trained checkpoints, it is usually a suboptimal way to deploy the model because the size of the checkpoints is large because they contain training nodes, which are not needed for inference. Running inference from checkpoints also adds overhead from weight loading.

To optimize inference performance, instead of using the trained checkpoints, we freeze the trained model checkpoints into a graph that only contains the inference graph and the model weights. As a result, the MLPerf Transformer model size is reduced to around 800 MB, compared to more than 1 GB with checkpoints. More importantly, the frozen graph is necessary if users want to quantize the model to int8 for an additional performance boost. The script to freeze the graph is available in the Model Zoo for Intel Architecture.
In the process of freezing the graph, the graph has gone through a few optimizations, like constants folding, identity node removal, and so on. After freezing the graph, we obtained a protobuf (.pb) file with known input and output nodes. It is quite simple to run the inference from the frozen graph. Users just need to feed input data into the input nodes of the model. After execution, the results are obtained from output nodes of the model. More importantly, if users have ideas to further optimize the model later, they can refreeze the model with the optimized model code without retraining, as long as the weights of the model are unchanged.

The Transformer model has two major modules: an encoder and a decoder. Each module has multiple layers: feed forward network (FFN), encoder-decoder attention, beam search, ops fusion, embedding, self-attention, etc. We will demonstrate the optimizations that we’ve done for several of these layers.

Padding/Unpadding Optimization in the FFN Layer

The most important optimization that we’ve done is to remove the padding/unpadding algorithm from the FFN layer. The input to this layer is a 3D padded tensor with shape \([\text{batch}\_\text{size}, \text{input}\_\text{length}, \text{Hidden}\_\text{size}]\). The model has paddings in the second dimension, as shown in Figure 2, where blank blocks denote the padding value, which is zero. The model finds and removes the paddings before feeding the tensor into the dense layer and reshapes the tensor to 2D with shape \([\text{batch}\_\text{size}, \text{number}\_\text{tokens}\*\text{Hidden}\_\text{size}]\), as shown in Figure 3. There is no reason to compute the values at the paddings because they will still be zero. However, after the dense layer, the model needs to put the padding back into the output tensor and restore the original shape (because it will be the input tensor for the next layer).

![Figure 2. Data layout of the FFN input tensor](image1)

![Figure 3. Optimized data layout of the FFN input tensor](image2)
The padding/unpadding algorithm tries to reduce the amount of computation by using more memory operations. This can improve performance if the computations are slower than memory operations. However, in modern Intel Xeon Scalable processors, computation is faster than the memory operations for this algorithm. Therefore, removing padding/unpadding from the FFN layer improves performance.

**Encoder-Decoder Attention Cache Optimization**

In the Transformer model, when the input tensor goes through the encoder module, the output is an encoder memory tensor, which will be used in the decoder module to compute encoder-decoder attention (Figure 4).

![Figure 4. Encoder-decoder attention](image)

Attention is the most important part of the Transformer model. It is computed with the following formula, where Q, K, and V are the tensors after dense layers:

$$Attention(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Q is the result of the dense layer with the decoder’s input. K and V are the results of the dense layers with the input of the encoder memory tensor. In every step of the decoder iterations, the value of the input tensor of the decoder changes, but the values of the memory tensor from the encoder stay the same. That means the K and V tensors stay the same in every step of the decoder, but the original Transformer model recomputes these tensors.

Obviously, this redundant computation should be eliminated. To remove it, we added a mechanism to cache K and V. We only compute the K and V tensors at the beginning of encoder-decoder attention, then cache the values to reuse in the remaining decoder iterations.
Beam Search Cache Optimization

Beam search is another time-consuming Transformer module. Profiling shows that the biggest bottleneck in this module is the Gather_ND operation, which collects tensor values that satisfy certain conditions (Figure 5). The random-access memory pattern makes this operation hard to vectorize and parallelize.

![Figure 5. Gather_ND pattern](image)

However, we noticed that the values we need to collect in the model are vectors in a multidimension tensor. The TensorFlow* framework includes a Gather_V2 operation that collects vectors from a tensor (Figure 6). This operation is easier to vectorize and parallelize, but the results should be the same. Replacing Gather_ND with Gather_V2 in the beam search algorithm improves performance.

![Figure 6. Gather_V2 pattern](image)

Ops Fusion Optimization

Ops fusion is a common optimization in deep learning models. In Transformer, we noticed that MatMul, Reshape, BiasAdd, and element-wise operations are hotspots in the FFN layer. Intel optimizations in TensorFlow fuse certain patterns of these operations into a new operation. However, the pattern of operations in the Transformer model (Figure 7) don’t match the required fusion pattern (Figure 8). After investigating the model source code, we found that we can move the Reshape operation before MatMul without causing any errors or increasing computation or memory usage. This can be done by changing just a few lines in the source code.
When we run inference with the regenerated model, the Intel® Optimization for TensorFlow* runtime automatically fuses operations to a new operation named MklFusedMatMul (Figure 9), which calls the highly optimized oneAPI Deep Neural Network Library (oneDNN). This reduces memory operations and achieves much better performance than the original model.

We noticed that the performance hotspots for inference in the Transformer attention layer are Mul, BatchMatMul, AddV2, and Softmax (Figure 10). Mul, AddV2, and Softmax are element-wise, memory-bound operations.

© Intel Corporation, Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others. For more complete information about compiler optimizations, see our Optimization Notice.
The Intel optimizations in TensorFlow fuse the BatchMatMul, Mul, and AddV2 sequential operations. Once again, rearranging the pattern (as shown in Figure 11) exposes an optimization opportunity (Figure 12). These fusions are important to the model—not only to compute more efficiently but also reduce memory traffic, which is especially important for memory-bound operations.

The next optimization is to fuse layer normalization in the Transformer model. Layer normalization is used in every layer of the encoder and decoder modules. These operations are either memory movement or element-wise operations. They are memory-bound, performance bottlenecks, so to reduce memory traffic for layer normalization, we implemented the layer normalization fusion (Figures 13 and 14).

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others. For more complete information about compiler optimizations, see our Optimization Notice.
Concluding Remarks

Transformer is a powerful but complicated deep learning model. The Transformer inference model in the MLPerf repository does not perform as well as the Intel-optimized version that takes advantage of the oneDNN library. The optimizations described above are based on the default FP32 model, which gave us the baseline inference performance. They significantly improved performance without sacrificing model accuracy. The model can be further optimized by converting to bfloat16 and int8 on supporting hardware. For example, the 4th Generation Intel Xeon processor supports both bfloat16 (BF16) and 8-bit quantization (int8) using Advanced Matrix Extension (AMX) instructions:

- Accelerating AI Performance on 3rd Gen Intel Xeon Scalable Processors with TensorFlow and Bfloat16
- Optimizing TensorFlow for 4th Gen Intel Xeon Processors

With converted models using these reduced precision supports, we can achieve even better performance. For more information, please refer to the Model Zoo for Intel Architecture. We have published all trained checkpoints, frozen graphs, and source code in the repository for your reference and use.
The Parallel Universe

Optimize Utility Maintenance Prediction for Better Service

Accelerate Machine Learning with the Intel® AI Analytics Toolkit

Kelli Belcher, AI Solutions Engineer, Intel Corporation

With increasing energy demand globally, the need for accurate and efficient monitoring of utility assets is crucial in being able to provide reliable services and avoid costly and unexpected downtimes. Current manual problem identifications are less than 50% accurate and the costs of maintenance and replacement of utility poles are over $10 billion. (Source: Utility Poles: Maintenance or Replacement. Utility Partners of America. August 3, 2020.)

In collaboration with Accenture, Intel has developed a Predictive Asset Analytics Reference Kit, designed to predict the health of utility assets and the probability of failure in order to proactively maintain assets, improve system reliability, and avoid outages, downtime, and operational costs. Included in this reference kit are the training data; an open source, trained, predictive asset analytics model; libraries; user guides; and oneAPI components to optimize the training cycles, prediction throughput, and accuracy.
To demonstrate how to implement this reference kit, the dataset in this tutorial was generated following the steps in the User Guide. The workflow is shown in Figure 1. It consists of 34 features on the overall health of the utility, including the age of the pole, maintenance history, outage records, and geospatial data. Our target variable is a binary indicator of whether or not the utility pole is failing.

A key benefit of this reference kit is the ability to optimize model training and inference across a heterogeneous XPU architecture with little or no code changes. This is enabled by the Intel® AI Analytics Toolkit (AI Kit) and the Intel® oneAPI Data Analytics Library (oneDAL). The main libraries from the AI Kit that we'll be working with in this guide are the Intel® Distribution of Modin*, Intel® Extension for Scikit-learn*, and Intel® Optimization for XGBoost*, all of which can be downloaded as part of the AI Kit or as standalone libraries. In addition, daal4py from oneDAL is used to speed up the inference time of the XGBoost* model.

Data Processing

To get started, we'll use Modin* to process and explore the data. Modin is a distributed DataFrame library designed to scale your pandas workflow with the size of your dataset, supporting datasets that range from 1 MB to 1 TB+. With pandas, only one core is used at a time. Modin’s Dask engine can take advantage of all available CPU cores, which allows you to work with very large datasets at much faster speeds.

Figure 2 shows Modin (left) and pandas (right) performing the same pandas operations on a 2 GB dataset. The only difference between the two notebook examples is the import statement.
To use Modin with the Dask engine, you can first import the drop-in replacement for pandas and then initialize the Dask execution environment in the engine call statement, shown in the following code cell:

```python
import modin.pandas as pd
from modin.config import Engine
Engine.put("dask")
```

**Exploratory Data Analysis**

Once Modin's Dask engine has been initialized, you can continue processing and exploring the data using the same pandas functions that will be parallelized across the available CPU cores. About 40% of the utility poles in our data have been identified as failing, shown as State 1 in Figure 3. Given the slight imbalance in the target distribution, we will use stratified sampling during cross-validation. For a further exploration of the data, please refer to this notebook.

![Figure 3. Distribution of the target variable: Asset Label](image)

```
```

© Intel Corporation, Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others. For more complete information about compiler optimizations, see our Optimization Notice.
Building the Solution

To predict the probability of a utility pole failure, several machine learning algorithms were considered. Due to the nonlinearity in the data, two nonparametric models were selected: a Support Vector Classifier (SVC) and XGBoost. Both are highly performant classifiers that are able to capture nonlinear relationships between the features and the target variable. However, they can also take a long time to tune hyperparameters and train models for increasingly large industrial datasets. This is where the AI Analytics Toolkit comes in.

Analysis with a Support Vector Classifier

To optimize the training and inference of the SVC model, we will use the accelerations provided by the Intel Extension for Scikit-learn. The Intel Extension for Scikit-learn reduces algorithm run time through the use of vector instructions, threading, and memory optimizations for both Intel® CPUs and GPUs. It supports many estimators and functions within the scikit-learn* library, including support vector machines, $k$-nearest neighbors, and random forests.

There are a few different ways to implement this extension. To dynamically patch all supported algorithms globally, all you need to do is call the function shown in the code cell below:

```python
from sklearnex import patch_sklearn
patch_sklearn()
```

The patch will replace supported stock scikit-learn algorithms with their optimized versions, and you can continue to import and use the same AI packages and scikit-learn libraries without any other changes to your code. To learn more about applying this extension, please see the Developer Guide.

Using stratified 3-fold cross-validation to tune the SVC, the best hyperparameters found were a radial basis function kernel with a regularization parameter of 10. On the out-of-sample test set, the SVC achieved an Area Under the ROC Curve (AUC) of 0.919 and an Average Precision (AUPRC) score of 0.902 (Figure 4).
Analysis with the Intel® Optimization for XGBoost®

To see if further improvement on the performance of the SVC could be achieved, an XGBoost model was tuned following the same steps above using stratified 3-fold cross-validation. During development of the reference kit, XGBoost v1.4.3 was used. XGBoost v0.81 and later contains the optimizations that Intel has been directly upstreaming into the package. The following graphs show the training (Figure 5) and prediction (Figure 6) time improvements that XGBoost v1.4.3 provides when compared to stock XGBoost v0.81 with default hyperparameters on this dataset.
For further improved performance on prediction time, the tuned XGBoost model can be converted to a daal4py model. Daal4py is the Python API for oneDAL. It uses Intel AVX-512 vectorization to maximize gradient boosting performance on Intel Xeon processors (Figure 7).

Figure 6. Intel optimizations available in XGBoost* v1.4.3 provide faster prediction ranging between 1.62x and 1.70x better performance than stock XGBoost v0.81.

Figure 7. Prediction time speedup from Intel optimizations available in XGBoost* v1.4.3 and daal4py v2021.6.0 vs. stock XGBoost v0.81
With tuned hyperparameters, Intel Optimization for XGBoost offers faster prediction times, ranging between 2.14x and 2.22x speedup. Daal4py offers an additional improvement in prediction time, ranging between 2.94x to 3.75x speedup, compared to stock XGBoost v0.81. Converting a tuned XGBoost model to daal4py can be done very easily with the following lines of code:

```python
import daal4py as d4p
daal_model = d4p.get_gbt_model_from_xgboost(xgb.get_booster())
```

Then, you can send the trained model along with the input data into daal4py’s prediction function to calculate the probabilities on the test set:

```python
daal_prob = d4p.gbt_classification_prediction(nClasses = 2, resultsToEvaluate = "computeClassLabels|computeClassProbabilities", fptype = 'float').compute(X_test, daal_model).probabilities
```

**Figure 8** shows the prediction performance attained by the XGBoost model using daal4py accelerations on the out-of-sample test set. XGBoost was able to outperform the SVC with an AUC of 0.9401 and an Average Precision score of 0.937.

**Conclusion**

Predictive maintenance solutions of huge scale typically require acceleration in training and inference. Using the optimizations available in the Intel AI Analytics Toolkit, we developed an efficient, end-to-end probabilistic classification tool to predict the likelihood of a utility pole failure. Faster training and inference results in less compute time, higher productivity, and fewer costs to produce predictive utility health forecasts for hundreds of thousands of assets. This reference provides a performance-optimized guide to the prediction of asset maintenance for utility customers that can easily be scaled across similar industries and use-cases.
Artificial intelligence (AI) has revolutionized virtually every industry from healthcare, retail, manufacturing, etc. However, most of today’s AI solutions are expensive and limited to a small set of data scientists. This is due to multiple factors. First, modern end-to-end AI pipelines are complex. They require multiple stages, like data processing, feature engineering, model development, model deployment, and maintenance. The iterative nature of these stages makes the process time-consuming. Second, deep expertise is often required to develop AI solutions. This creates an entry barrier for novice and citizen data scientists. Third, people tend to develop larger and deeper models to get better accuracy. These “over-parameterized” models lead to significant computational demands, which hinders deployment in resource-constrained environments.
We developed Intel® End-to-End AI Optimization Kit to make the end-to-end AI pipeline faster, simpler, and more accessible, broadening AI access to everyone, everywhere. It is a composable toolkit for end-to-end AI optimization to deliver high performance, lightweight models efficiently on commodity hardware. The toolkit is built on a set of Intel-optimized frameworks, such as Intel® Extension for PyTorch* (IPEX), Intel® Extension for TensorFlow* (ITEX), and Intel® AI Analytics Toolkit (AI Kit). It also integrates SigOpt for hyperparameter optimization. Intel End-to-End AI Optimization Kit also provides unique components and features for data preparation, model optimization, and model construction.

It improves the scale-up and scale-out efficiency of end-to-end AI pipelines to make “overnight training” for complex DL models possible. It delivers lighter DL models that have higher inference throughput lower resource requirements. It also makes end-to-end AI simpler. It automates the pipeline with click-to-run workflows and SigOpt AutoML, abstracts the complex APIs for data processing and feature engineering, simplifies distributed training, and can be easily integrated with existing or third-party machine learning (ML) solutions or platforms. It brings complex, compute-intensive deep learning models to commodity hardware, delivers built-in optimized models through parameterized models generated by smart democratization advisor (SDA), and domain-specific compact neural networks constructed with neural architecture search (NAS) technology. All of this makes AI more accessible to citizen data scientists.

Architecture

Popular models (e.g., RecSys, CV, NLP, ASR, and RL) from various domains are the input to Intel End-to-End AI Optimization Kit (Figure 1). These stock models are heavy, slow to train, and complex to tune and optimize. Depending on model type, Intel End-to-End AI Optimization Kit will optimize the models with either model advisor or neural network constructor. The expected outputs are optimized models that require only one-tenth the FLOPS (Floating-point Operations Per Second) and training time of the stock model with the same or minimal loss of accuracy.

![Figure 1. Architecture and workflow of Intel® End-to-End AI Optimization Kit](image)
RecDP

RecDP is a parallel data processing and feature engineering library built on PySpark* and extensible to other data processing tools, like Modin*. Its key features and functions are as follows:

- A tabular dataset processing toolkit
- Abstract APIs to hide Spark* programming complexity
- Optimized performance through adaptive query plan and strategy
- Support for common feature engineering functions like target and count encoding
- Easy integration into third-party solutions

RecDP uses “lazy execution” for better performance. It fuses operators and leverages the collection of data statistics to avoid unnecessary passes through the dataset, which is critical when handling large datasets. RecDP can also leverage the native columnar SQL engine capabilities provided by Optimized Analytics Package for Spark* Platform to improve performance.

Smart Democratization Advisor (SDA)

SDA is a user-guided tool to facilitate automation. It provides built-in intelligence through parameterized models and leverages SigOpt for hyperparameter optimization (HPO) and built-in optimized models (e.g., RecSys, CV, NLP, ASR, and RL). It also converts manual model tuning and optimization to assist AutoML and AutoHPO.

Neural Network Constructor

Neural network constructor is based on neural architecture search technology. Using a predefined supernet, it constructs neural network structures directly for a given domain. Its key features and functions are as follows:

- Multi-model support, such as models from the CV, NLP, and ASR domains.
- Uses a unified, transformer-based supernet.
- Hardware-aware NAS uses metrics, like FLOPS and latency, as thresholds to determine the model architecture and model size.
- Train-free NAS uses a zero-cost proxy metric rather than training accuracy for candidate evaluation. It takes multiple network’s characteristics into consideration, such as trainability, expressivity, diversity, and saliency.
- Leverages model adapter to deploy the model in the user’s production environment. Model adapter is a transfer learning-based component that provides fine tuning, knowledge distillation, and domain adaption features.

Example

Here is an example of how the toolkit works on the DL Recommendation Model (DLRM), including environment setup, data processing, built-in model advisor with patched code, and a one-line command to kick off the entire optimization process.
The Parallel Universe

Step 1: Environment Setup
Intel End-to-End AI Optimization Kit is built on top of AI Kit, so the software necessary to run the pipeline is already accessible:

```bash
# DockerFile
FROM docker.io/intel/oneapi-akit
ENV http_proxy=http://proxy-prc.intel.com:912
ENV https_proxy=http://proxy-prc.intel.com:912

# SigOpt
RUN python -m pip install sigopt==7.5.0 --ignore-installed

# PyTorch conda
RUN conda activate pytorch
RUN python -m pip install prefetch_generator tensorboardX onnx tqdm lark-parser

# Intel Extension for PyTorch
RUN python -m pip install intel_extension_for_pytorch==1.10.0 -f https://software.intel.com/ipex-whl-stable psutil

RUN mkdir -p /home/vmagent/app
WORKDIR /home/vmagent/app
```

Step 2: Parallel Data Processing with RecDP
The next step is to use RecDP for simplified data processing. In this example, two operators, Categorify() and FillNA(), are chained together, and Spark lazy execution is used to reduce unnecessary passes through the data:

```python
from pyspark.sql import *
from pyspark import *
from pyspark.sql.types import *
from pyrecdp.data_processor import *
from pyrecdp.encoder import *
from pyrecdp.utils import *
import numpy as np

path_prefix = "file://"
current_path = "/home/vmagent/app/dataset/demo/processed/"
csv_file = "/home/vmagent/app/dataset/demo/criteo_mini.txt"

# 1. Start spark and initialize data processor

t0 = timer()
spark = SparkSession.builder.master('local[*]')
   .config('spark.driver.memory','100G')
   .appName("DLRM").getOrCreate()

schema = StructType([StructField(f'_i{i}', IntegerType()) for i in range(0, 14)])
df = spark.read.option('sep', '\t').option("mode", "DROPMALFORMED")
   .schema(schema).csv(path_prefix + csv_file)
proc = DataProcessor(spark, path_prefix, current_path=current_path, spark_mode='local')

# 2. Process data with RecDP

CAT_COLS = list(range(14, 40))
to_categorify_cols = [f'c{i}d' for i in CAT_COLS]
op_categorify = Categorify(to_categorify_cols)
op_fillna_for_categorified = FillNA(to_categorify_cols, 0)
proc.append_ops([op_categorify, op_fillna_for_categorified])
df = proc.transform(df, name='dlrm')

t1 = timer()

print(f"Total process time is {{{t1 - t0}}} secs")
```
Step 3: AutoML with Smart Democratization Advisor

SDA converts the manual optimizations to assist AutoML. It provides predefined parameters for built-in models, which significantly reduces the time for AutoML:

```python
model_info = dict()

# Config for model
model_info["score_metrics"] = [{"accuracy", "maximize"}, {"training_time", "minimize"}]
model_info["execute_cmd_base"] = "python launch.py"
model_info["result_file_name"] = "best_auc.txt"

# Config for SigOpt
model_info["experiment_name"] = "dlrm"
model_info["sigopt_config"] = [
    {'name': 'learning_rate', 'bounds': {'min': 5, 'max': 50}, 'type': 'int'},
    {'name': 'lamb_lr', 'bounds': {'min': 5, 'max': 50}, 'type': 'int'},
    {'name': 'warmup_steps', 'bounds': {'min': 2000, 'max': 4500}, 'type': 'int'},
    {'name': 'decay_start_steps', 'bounds': {'min': 4501, 'max': 9000}, 'type': 'int'},
    {'name': 'num_decay_steps', 'bounds': {'min': 5000, 'max': 15000}, 'type': 'int'},
    {'name': 'sparse_feature_size', 'grid': [128, 64, 16], 'type': 'int'},
    {'name': 'mlp_top_size', 'bounds': {'min': 0, 'max': 7}, 'type': 'int'},
    {'name': 'mlp_bot_size', 'bounds': {'min': 0, 'max': 3}, 'type': 'int'}
]
model_info["observation_budget"] = 1
```

Besides some configurable parameters, there are cases that need code-level optimization. For all built-in modes, the kit provides optimized models with patched code. Here is an example using IPEX and BF16, as well as the optimizer to improve model convergence on multiple CPU nodes:

```python
# Framework optimization. Use IPEX & BF16

# Embedding Optimization
- m_curr = (m[i] if self.max_emb_dim > 0 else m)
- EE = nn.EmbeddingBag(n, m_curr, mode="sum", sparse=True)
- W = np.random.uniform(
    low=-np.sqrt(1 / n), high=np.sqrt(1 / n), size=(n, m_curr)
).astype(np.float32)
- # democratized, use two dimension, sparse and dense
+ W = np.random.uniform(
    low=-np.sqrt(1 / n), high=np.sqrt(1 / n), size=(n, m)
).astype(np.float32)

# Optimizer Optimization
- optimizer = torch.optim.SGD(dlrm.parameters(), lr=learning_rate) ...
```

Step 4: End-to-End Model Optimization

Finally, we kick off the end-to-end model optimization with just a few lines of codes:

```python
import sys
from e2eAIOK import SDA
sda = SDA(model="dlrm", settings=setting)
sda.launch()
```
Intel End-to-End AI Optimization Kit provides more click-to-run recipes for popular models, including:

- RecSys **DLRM, DIEN, WnD**
- Automatic speech recognition (ASR) **RNNT**
- Compute vision (CV) **RESNET**
- Natural language processing (NLP) **BERT**
- Reinforcement learning (RL) **MiniGO**

**Performance**

The tests were conducted on a four-node cluster (**Table 1**). Each node was equipped with two Intel® Xeon® Platinum 6240 processors and 384 GB memory. The nodes were connected through 25 GB Ethernet. One 1 TB HDD SSD was used as a data drive.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Date</td>
<td>12/2021</td>
</tr>
<tr>
<td>Platform</td>
<td>S2600WFT</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel® Xeon® Gold 6240</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>4</td>
</tr>
<tr>
<td>CPU per node</td>
<td>18core/socket, 2 sockets, 2 threads/core</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR4 dual rank 384G, 12 slots / 32GB / 2666</td>
</tr>
<tr>
<td>Storage</td>
<td>1x 400GB Intel® SSD (SSDSC2BA400G3) OS Drive</td>
</tr>
<tr>
<td></td>
<td>1TB HDD for data storage</td>
</tr>
<tr>
<td>Network</td>
<td>1x Intel® X722, 1x Intel XXV710</td>
</tr>
<tr>
<td>Microcode</td>
<td>0x500002C</td>
</tr>
<tr>
<td>BIOS version</td>
<td>SE5C620.86B.0X.02.0094.102720191711</td>
</tr>
<tr>
<td>OS/Hypervisor/SW</td>
<td>Fedora* 29</td>
</tr>
<tr>
<td></td>
<td>5.3.11-100.fc29.x86_64</td>
</tr>
</tbody>
</table>

**Table 1. System configuration**

We compared the stock and optimized models’ performance on three popular RecSys models (**Table 2**). It delivered significant speedups for ETL, training, and inference for all three workloads (**Figure 2**).
<table>
<thead>
<tr>
<th>Workloads</th>
<th>DLRM</th>
<th>WnD</th>
<th>DIEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>PyTorch *</td>
<td>TensorFlow *</td>
<td>TensorFlow</td>
</tr>
<tr>
<td>Libraries</td>
<td>oneMKL, oneCCL</td>
<td>oneMKL, oneCCL</td>
<td>oneMKL, oneCCL</td>
</tr>
<tr>
<td>Dataset (size, shape)</td>
<td>Criteo</td>
<td>outbrain</td>
<td>categoryFiles</td>
</tr>
<tr>
<td>Precision (FP32, INT8., BF16)</td>
<td>BF16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>KMP AFFINITY</td>
<td>granularity=fine,compact,1,0</td>
<td>granularity=fine,compact,1,0</td>
<td>granularity=fine,compact,1,0</td>
</tr>
<tr>
<td>NUMACTL</td>
<td>Socket binding</td>
<td>Socket binding</td>
<td>Socket binding</td>
</tr>
<tr>
<td>OMP_NUM_THREADS</td>
<td>20</td>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2. Test configuration

![E2E RecSys democratization Performance](image)

Figure 2. RecSYS speedup with Intel® End-to-End AI Optimization Kit. Baseline configuration: one node with two Intel® Xeon® Gold 6240 processors (18 cores), HT On, Turbo ON, 384 GB (12 slots/32GB/2666 MHz) memory, BIOS SE5C620.86B.0X.02.0094.10272019171 (ucode:0x500002C), Fedora* 29, 5.3.11-100.fc29.x86_64, PyTorch*, TensorFlow*, Spark*. Optimized configuration: same hardware and software configuration except four nodes, and Intel® Extension for PyTorch* and Intel® Extension for TensorFlow*, Horovod*, and modified DLRM, WnD, and DIEN workloads.
Call to Action

Intel End-to-End AI Optimization Kit is a composable toolkit that delivers high performance and lightweight models for commodity hardware, which helps to democratize AI. It leverages several key optimizations:

- Parallel data processing with RecDP
- Intel-optimized training frameworks
- Lighter models with fewer layers and reduced communication overhead on distributed-memory parallel computers
- Optimizer tuning (DLRM) to converge faster with larger batch size
- Feature engineering (embedding table and encoding) optimizations

It gives good results on a wide range of popular models. If you want to use it for your own project, please visit https://github.com/intel/e2eAIOK/.
Rapid technology innovation is driving a new era of heterogeneous computing. Hardware diversity and growing computational demands require programming models that can exploit heterogeneous parallelism. These models must also be open and portable so that programs can run on hardware from different vendors. Though decades old, Fortran is still an active and important programming language in science and engineering. Likewise, OpenMP*, the open standard for compiler-directed parallelism released in 1997, has evolved to support heterogeneity. It now includes directives to offload computation to an accelerator device and to move data between disjoint memories. The concepts of host and device memory, and other more subtle memory types, like texture/surface and constant memory, are exposed to developers through OpenMP directives.
Offloading tasks to accelerators can make some computations more efficient. For example, highly data parallel computations can take advantage of the many processing elements in a GPU. This article will show how Fortran + OpenMP solves the three main heterogeneous computing challenges: offloading computation to an accelerator, managing disjoint memories, and calling existing APIs on the target device.

**Offloading Computation to an Accelerator**

Let’s start with an example. Figure 1 shows how the OpenMP target, teams, and distribute parallel do constructs execute a nested loop. The target construct creates a parallel region on the target device. The teams construct creates a league of teams (i.e., groups of threads). In the example, the number of teams is less than or equal to the num_blocks parameter. Each team has a number of threads less than or equal to the variable block_threads. The primary thread of each team executes the code in the teams region. The iterations in the outer loop are distributed among the primary threads of each team. When a team’s primary thread encounters the distribute parallel do construct, the other threads in its team are activated. The team executes the parallel region and then workshares the execution of the inner loop. This is shown schematically in Figure 2.
program target_teams_distribute
  external saxpy

  integer, parameter :: n = 2048, num_blocks = 64
  real, allocatable :: A(:,), B(:,), C(:,)
  real :: d_sum = 0.0
  integer :: i, block_size = n / num_blocks
  integer :: block_threads = 128

  allocate(A(n), B(n), C(n))
  A = 1.0
  B = 2.0
  C = 0.0

  call saxpy(A, B, C, n, block_size, num_blocks, block_threads)

  do i = 1, n
    d_sum = d_sum + C(i)
  enddo

  print '("sum = 2048 x 2 saxpy sum:"(f))', d_sum
  deallocate(A, B, C)
end program target_teams_distribute

subroutine saxpy(B, C, D, n, block_size, num_teams, block_threads)
  real :: B(n), C(n), D(n)
  integer :: n, block_size, num_teams, block_threads, i, i0

  !$omp target map(to: B, C) map(tofrom: D)
  !$omp teams num_teams(num_teams) thread_limit(block_threads)
  do i0 = 1, n, block_size
    !$omp distribute parallel do
      do i = i0, min(i0 + block_size - 1, n)
        D(i) = D(i) + B(i) * C(i)
      enddo
    enddo
  !$omp end teams
  !$omp end target
end subroutine

Figure 1. Offloading a nested loop to an accelerator using OpenMP* directives (shown in blue)

Figure 2. Conceptual diagram of the OpenMP* target, teams, and distribute parallel do regions
Host-Device Data Transfer

Now let’s turn our attention to memory management and data movement between the host and the device. OpenMP provides two approaches. The first uses the `data` construct to map data between disjoint memories. In Figure 1, for example, the `map(to: B, C)` and `map(from: D)` clauses on the target directive copy arrays B, C, and D to the device and retrieve the final values in D from the device. The second approach calls the device memory allocator, an OpenMP runtime library routine. This article will not cover the latter approach.

In Figure 3, the `target data` construct creates a new device data environment (also called the target data region) and maps arrays A, B, and C to it. The target data region encloses two target regions. The first one creates a new device data environment, which inherits A, B, and C from the enclosing device data environment according to the `map(to: A, B)` and `map(from: C)` data motion clauses. The host waits for the first target region to complete, then assigns new values to A and B in the data environment. The `target update` construct updates A and B in the device data environment. When the second target region finishes, the result in C is copied from the device to the host memory upon exiting the device data environment. This is all shown schematically in Figure 4.
program target_data_update
  integer :: i, n = 2048
  real, allocatable :: A(:), B(:), C(:)
  real :: d_sum = 0.0
  allocate(A(n), B(n), C(n))
  A = 1.0
  B = 2.0
  C = 0.0

  !$omp target data map(to: A, B) map(from: C)
  !$omp target
  !$omp parallel do
  do i = 1, n
    C(i) = A(i) * B(i)
  enddo
  !$omp end target
  A = 2.0
  B = 4.0
  !$omp target update to(A, B)
  !$omp target
  !$omp parallel do
  do i = 1, n
    C(i) = C(i) + A(i) * B(i)
  enddo
  !$omp end target
  !$omp end target data
  do i = 1, n
    d_sum = d_sum + C(i)
  enddo
  print '("sum = 2048 x (2 + 8) sum:"(f))', d_sum
  deallocate(A, B, C)
end program target_data_update

Figure 3. Creating a device data environment.

Figure 4. Host-device data transfer for the OpenMP* program shown in Figure 3. Each arrowhead indicates data movement between the host and device memories.
The command to compile the previous example programs using the Intel® Fortran Compiler and OpenMP target offload on Linux* is:

```
$ ifx -xhost -qopenmp -fopenmp-targets=spir64 source_file.f90
```

### Using Existing APIs from OpenMP Target Regions

Calling external functions from OpenMP target regions is covered in [Accelerating LU Factorization Using Fortran, oneMKL, and OpenMP*](https://software.intel.com). In a nutshell, the dispatch directive tells the compiler to output conditional dispatch code around the associated subroutine or function call:

```
!$omp target data
!$omp dispatch
  call external_subroutine_on_device
!$omp end target data
```

If the target device is available, the variant version of the structured block is called on the device.

### Intel Fortran Support

The Intel® Fortran Compiler (ifx) is a new compiler based on the Intel Fortran Compiler Classic (ifort) frontend and runtime libraries, but it uses the LLVM (Low Level Virtual Machine) backend. See the [Intel Fortran Compiler Classic and Intel Fortran Compiler Developer Guide and Reference](https://software.intel.com) for more information. It is binary (.o/.obj) and module (.mod) compatible, supports the latest Fortran standards (95, 2003, 2018) and heterogeneous computing via OpenMP (v5.0 and v5.1). Another approach to heterogeneous parallelism with Fortran is the standard `do concurrent` loop:

```fortran
program test_auto_offload
  integer, parameter :: N = 100000
  real :: a(N), b(N), c(N), sumc
  a = 1.0
  b = 2.0
  c = 0.0
  sumc = 0.0
  call add_vec
  do i = 1, N
    sumc = sumc + c(i)
  endo
  print *, ' sumc = 300,000 = ', sumc
contains
  subroutine add_vec
    do concurrent (i = 1:N)
      c(i) = a(i) + b(i)
    enddo
  end subroutine add_vec
end program test_auto_offload
```
Compile this code as follows and the OpenMP runtime library will generate device kernel code:

\[ $\text{ifx -xhost -openmp -openmp-targets=spir64 -openmp-target-do-concurrent source_file.f90} \]

The `-openmp-target-do-concurrent` flag instructs the compiler to generate device kernel for the `do concurrent` loop automatically.

The OpenMP runtime can provide a profile of kernel activity by setting the following environment variable:

\[ $\text{export LIBOMPTARGET_PLUGIN_PROFILE=T} \]

Running the executable will give output

Look for the subroutine name “add vec” in the output when the program is executed, e.g.:

\[ \text{Kernel 0 : __omp_offloading_3b_dd004710_test_auto_offload_IP_add_vec_110} \]

The Fortran language committee is working on a proposal to add reductions to `do concurrent` in the 2023 standard, i.e.:

\[ \text{do concurrent(i = 1:N) reduce(+:sum)} \]

**Closing Thoughts**

We've given an overview of heterogeneous parallel programming using Fortran and OpenMP. As we've seen in the code examples above, OpenMP is a descriptive approach to express parallelism that is generally noninvasive. In other words, the underlying sequential program is still intact if the OpenMP directives are not enabled. The code still works on homogeneous platforms without accelerator devices. Fortran + OpenMP is a powerful, open, and standard approach to heterogeneous parallelism.
Break Free of Code Boundaries

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

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

Learn
Get hands-on experience with SYCL 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 >
The previous article, *Accelerating LU Factorization Using Fortran, oneMKL, and OpenMP*, showed how to offload LU factorization and subsequent inversion to an accelerator. It gave some tips on reducing overhead by consolidating OpenMP offload regions and minimizing host-device data transfers but stopped short of solving a system of linear equations. This was left as an exercise for the reader, but we’re going to do it now to illustrate some techniques from the excellent *Minimizing Data Transfers and Memory Allocations* chapter in the *oneAPI GPU Optimization Guide*.

We’re going to show how to solve a group of linear systems using the batched LU factorization and solver functions in oneMKL, offloaded to an accelerator using OpenMP. We don’t use inverted matrix factors much in practice, so we’re going to drop that step in this demonstration and go from factorization to solution. Because the previous example codes are written in Fortran, and because we like it, we’re going to continue using Fortran here.
Let’s start with a couple of small linear systems to confirm that we’re loading and solving them correctly. We found some suitable examples in an old linear algebra textbook (Figure 1). The matrices are square so we can solve them using LU factorization.

\[
\begin{bmatrix}
2 & 4 & -4 \\
1 & -4 & 3 \\
-6 & -9 & 10
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} =
\begin{bmatrix}
12 \\
-21 \\
-24
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & 0 & 3 \\
2 & 4 & -1 \\
6 & 5 & 5
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} =
\begin{bmatrix}
6 \\
-10 \\
-7
\end{bmatrix}
\]

\[(x_1, x_2, x_3) = (-4, 2, -3)\]  \( (x_1, x_2, x_3) = (-2, -1, 2) \]

**Figure 1.** We’ll solve these two linear systems as a batch with oneMKL and OpenMP* target offload. Small problems like these are fine for testing, but far too small to merit acceleration.

The demo program for this article is similar to the one in the previous article. It begins by including the Fortran interfaces for OpenMP offload support, then it decides whether to use 32- or 64-bit integer type (Figure 2). (See Using the ILP64 Interface vs. LP64 Interface for more information.) This decision will be made at compile-time (see the compiler command below). The test problem in Figure 1 consists of two, 3x3 linear systems, each with one right-hand side (RHS). The `batch_size`, `n`, `nrhs`, and `stride` variables are set accordingly, and the `a`, `b`, and `ipiv` arrays are allocated to hold the batched systems. The last four statements load the two matrices and their right-hand sides into `a` and `b`, respectively.
include "mkl_omp_offload.f90"

program solve_batched_linear_systems
  ! Decide whether to use 32- or 64-bit integer type
  #if defined(MKL_ILP64)
    use onemkl_lapack_omp_offload_ilp64 ! 64-bit
  #else
    use onemkl_lapack_omp_offload_lp64 ! 32-bit
  #endif

  integer, parameter :: n = 3, batch_size = 2, nrhs = 1
  integer :: lda, stride_a, stride_ipiv
  integer :: ldb, stride_b
  real (kind=8), allocatable :: a(:,,:), b(:,:)
  integer, allocatable :: ipiv(:,,:), info(:)

  integer allocstat
  character (len = 132) :: allocmsg

  lda = n
  stride_a = n * lda
  stride_ipiv = n
  ldb = n
  stride_b = n * nrhs

  ! Allocate required memory
  allocate (a(stride_a, batch_size), b(n, batch_size*nrhs), &
    ipiv(stride_ipiv, batch_size), &
    info(batch_size), stat = allocstat, errmsg = allocmsg)
  if (allocstat > 0) stop trim(allocmsg)

  ! Initialize the matrices. Remember that Fortran is a column-major language.
  a(:,1) = (/2.0, 1.0, -6.0, 4.0, -4.0, -9.0, -4.0, 3.0, 10.0/)
  a(:,2) = (/0.0, 2.0, 6.0, 0.0, 4.0, 5.0, 3.0, -1.0, 5.0/)
  b(:,1) = (/12.0, -21.0, -24.0/)
  ! x = (-4, 2, -3)
  b(:,2) = (/6.0, -10.0, -7.0/)
  ! x = (-2, -1, 2)

Figure 2. Setting up the example program. The oneMKL headers and modules that provide OpenMP* target offload support are highlighted in blue.

We are now ready to begin the computation. Let’s start with a basic implementation that uses two OpenMP target regions to dispatch the LU factorizations and the solutions to the linear systems (Figure 3). We compile and run the example as follows to verify that it’s giving correct results:
The target construct transfers control to the accelerator device. The first region transfers the input matrices \([\text{map(tofrom:a)}]\) to the device memory; dispatches the in-place LU factorization (\text{dgetrf\_batch\_strided}) to the device; and retrieves the LU-factored matrices (now stored in \(a\)), pivot indices \([\text{map(from:ipiv)}]\), and status \([\text{map(from:info)}]\) from device memory. If factorization succeeded, the program proceeds to the next OpenMP region. The LU-factored matrices and pivot indices are transferred back to the device memory \([\text{map(to:a)}\) and \(\text{map(to:ipiv)}\)] and the linear systems are solved on the device (\text{dgetrs\_batch\_strided}). The RHS are overwritten with the solution vectors and retrieved from the device memory \([\text{map(tofrom:b)}]\) along with the status of the computation \([\text{map(from:info)}]\). The host-device data transfers are shown schematically in Figure 4.
We don’t have to rely on a conceptual understanding of host-device data transfer, however. We can directly check data movement by requesting debugging information from the OpenMP runtime library, as explained in Minimizing Data Transfers and Memory Allocations. The test problem is too small to warrant acceleration, so we'll increase the problem size to something more interesting. Let's solve (in double precision) a batch of eight 8,000 x 8,000 linear systems, each with one RHS:

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./lu_solve_ex1 8000 8 1 1 >& lu_solve_ex1.out
$ grep Moving lu_solve_ex1.out
```

![Figure 4. Host-device data transfer for the two OpenMP* target offload regions shown in Figure 3. Each arrowhead indicates data movement between the host and device memories.](image)

We’ve highlighted the arrays that are explicitly transferred using OpenMP `map` constructs and added a space to delineate the two OpenMP target regions. The unhighlighted data movements are the **dope vectors** of the arrays being mapped to the target device. We can ignore them because they’re generally small.

In the first target region, array \(a\) (4,096,000,000 bytes) is transferred \([\text{map (tofrom:a)}]\) from host to target (hst→tgt) and target to host (tgt→hst). It is transferred \([\text{map (to:a)}]\) back to the target device in...
The second target region. The pivots \(\text{ipiv}, 512,000\) bytes are computed on the device in the first target region, retrieved \([\text{map(from:ipiv)}, \text{tgt\rightarrow hst}]\), then transferred back to the device \([\text{map(to:ipiv)}, \text{hst\rightarrow tgt}]\) in the second target region to compute the solutions. The RHS stored in array \(b\) \((512,000\) bytes) are transferred to the device, overwritten with the solution vectors, then transferred back to the host \([\text{map(tofrom:b}]\). The status array, \(\text{info}\) \((64\) bytes), is retrieved from the device \([\text{map(from:info)}, \text{tgt\rightarrow hst}]\) at the end of both target regions.

Moving data between disjoint memories takes time and energy, so we need to pay close attention to the highlighted transfers \((12,290,048,128\) bytes in total). With this in mind, let’s see how to improve our initial implementation.

The previous article on LU factorization discussed the disadvantages of using two OpenMP target regions when one will suffice. First, transferring the control flow to the target device incurs overhead. This happens twice in the initial implementation (Figure 3). Second, redundant host-device data transfer is required (Figure 4). When solving the linear systems, we only need to copy the input arrays and RHS to the device and retrieve the solutions from the device. We also need to retrieve the status arrays, but these are relatively small. The pivots are only used by the device, so we’ll simply allocate the \(\text{ipiv}\) array directly in device memory \([\text{map(alloc:ipiv(1:stride_ipiv, 1:batch_size))}]\). Fusing the two OpenMP target regions results in cleaner, more concise code (Figure 5) that requires less data movement (Figure 6).

```c
!$omp target data map(to:a) map(tofrom:b)
   &
   map(alloc:ipiv(1:stride_ipiv, 1:batch_size)) &
   map(from:info_rf, info_rs)

!$omp dispatch
call dgetrf_batch_strided(n, n, a, lda, stride_a, &
   ipiv, stride_ipiv, batch_size, info_rf)

!$omp dispatch
call dgetrs_batch_strided('N', n, nrhs, a, lda, stride_a, &
   ipiv, stride_ipiv, &
   b, ldb, stride_b, batch_size, info_rs)

!$omp end target data
if (any(info_rf .ne. 0)) then
   print *, 'Error: getrf_batch_strided returned with errors.'
Elseif (any(info_rs .ne. 0)) then
   print *, 'Error: getrs_batch_strided returned with errors.'
Else
   print *, 'Computation executed successfully'
endif
```

Figure 5. Using a single OpenMP* target region to solve batched linear systems with oneMKL. OpenMP directives are highlighted in blue. LAPACK calls are highlighted in green.
The debug information confirms this. Once again, we’ve highlighted the arrays that are explicitly transferred using OpenMP map constructs:

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Libomptarget --&gt; Moving 88 bytes (hst:0x00007ffe443ba9c8)</code></td>
<td><code>tgt:0x0000000000155008</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 64 bytes (hst:0x00007ffe443bad68)</code></td>
<td><code>tgt:0x0000000000155088</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 64 bytes (hst:0x00007ffe443bad18)</code></td>
<td><code>tgt:0x0000000000155108</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 512000 bytes (hst:0x00007ffe443bad18)</code></td>
<td><code>tgt:0x0000000000155108</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 88 bytes (hst:0x00007ffe443bae38)</code></td>
<td><code>tgt:0x0000000000155188</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 4096000000 bytes (hst:0x00007ffe443bae38)</code></td>
<td><code>tgt:0x0000000000155188</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 512000 bytes (hst:0x00007ffe443bad18)</code></td>
<td><code>tgt:0x0000000000155208</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 64 bytes (hst:0x00007ffe443bad68)</code></td>
<td><code>tgt:0x0000000000155208</code></td>
</tr>
<tr>
<td><code>Libomptarget --&gt; Moving 64 bytes (hst:0x00007ffe443bad18)</code></td>
<td><code>tgt:0x0000000000155208</code></td>
</tr>
</tbody>
</table>

If we ignore the dope vectors, the single OpenMP target region in Figure 5 requires significantly less host-device data transfer (4,097,024,128 bytes) than the two target regions in the initial implementation (12,290,048,128 bytes) (Figure 3).

Minimizing host-device data transfer is a first-order concern in heterogeneous parallel computing. The code in the second example (Figure 5) significantly outperforms the original code (Figure 3), especially as the linear systems get larger (Table 1).
<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>CPU Time</th>
<th>GPU time (Example 1)</th>
<th>GPU time (Example 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000 x 1,000</td>
<td>0.13</td>
<td>1.72</td>
<td>1.70</td>
</tr>
<tr>
<td>2,000 x 2,000</td>
<td>0.60</td>
<td>0.84</td>
<td>0.72</td>
</tr>
<tr>
<td>4,000 x 4,000</td>
<td>3.90</td>
<td>2.51</td>
<td>2.10</td>
</tr>
<tr>
<td>8,000 x 8,000</td>
<td>19.15</td>
<td>5.98</td>
<td>4.55</td>
</tr>
<tr>
<td>12,000 x 12,000</td>
<td>63.11</td>
<td>12.55</td>
<td>9.51</td>
</tr>
<tr>
<td>16,000 x 16,000</td>
<td>139.45</td>
<td>21.33</td>
<td>15.71</td>
</tr>
</tbody>
</table>

Table 1. Timing the solution of eight batched linear systems of varying matrix sizes on a Linux* (Ubuntu* 20.04 x64, 5.15.47 kernel) system with two 2.0 GHz 4th Gen Intel® Xeon® Platinum 8480+ processors (CPU), an Intel® Data Center GPU Max 1550 (GPU), and 528 GB memory. All times are in seconds. GPU tests used only one tile. Each linear system had one RHS. Each experiment was run five times. The first run was discarded because it includes the just-in-time compilation overhead for the oneMKL functions. The reported time is the sum of the remaining four runs.

The example programs used in this article are available at this repository. You can experiment with OpenMP accelerator offload on the free Intel® Developer Cloud, which has the latest Intel hardware and software.

The OpenMP target construct has many more options to give the programmer fine control over accelerator offload. We encourage you to review the OpenMP specification and example codes, and to check out the Intro to GPU Programming with the OpenMP API and Minimizing Data Transfers and Memory Allocations tutorials.
Device discovery is an important aspect of SYCL* or any cross-architecture, heterogeneous parallel programming approach. My previous oneAPI articles focused on using SYCL and the oneMKL and oneDPL libraries to offload computations to an accelerator device; in other words, to control where the code executes. This article focuses on device discovery because writing portable code for heterogeneous systems requires the ability to query the system for information about the available hardware. For example, if we hardcoded the SYCL device selector to use a GPU, but there’s no GPU in the system, the following statement will fail:

```c
   sycl::queue Q(sycl::gpu_selector_v);
   ...
   terminate called after throwing an instance of 'sycl::_V1::runtime_error'
   Aborted
```
The code isn’t portable to systems without a GPU. Instantiating the SYCL queue with the default selector instead of the GPU selector is guaranteed to work, but we lose control over where the queue submits work. The SYCL runtime chooses the device, e.g.:

```cpp
...  sycl::queue Q(sycl::default_selector_v);
  std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;
...
```

Running on: **Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz**

To write more robust heterogeneous parallel programs, let’s take a closer look at SYCL device discovery to answer the following questions:

- What accelerator devices are available?
- What device is a SYCL queue using?
- What device is a oneDPL execution policy using?

## Robust Device Discovery

Let’s run some examples on the **Intel® DevCloud for oneAPI** because it has a variety of Intel® hardware options and the latest Intel® oneAPI toolkits are already installed. The hardware is refreshed periodically, but the following compute nodes are available at the time of writing (December 2, 2022):

```
$ pbsnodes | grep properties | sort | uniq -c | sort -nr
```

```
79      properties = xeon,skl,gold6128,ram192gb,net1gbe,jupyter,batch
78      properties = xeon,cfl,e-2176g,ram64gb,net1gbe,gpu,gen9
26      properties = xeon,skl,gold6128,ram192gb,net1gbe,jupyter,batch,fpga_compile
25      properties = core,tgl,i9-11900kb,ram32gb,net1gbe,gpu,gen11
12      properties = xeon,skl,ram384gb,net1gbe,renderkit
12      properties = xeon,skl,gold6128,ram192gb,net1gbe,fpga_runtime,fpga,arria10
  6      properties = xeon,icx,gold6348,ramgb,net1gbe,jupyter,batch
  4      properties = xeon,icx,plat8380,ram2tb,net1gbe,batch
  4      properties = xeon,clx,ram192gb,net1gbe,batch,extended,fpga,stratix10,fpga_runtime
```

As you can see, we have plenty of CPU, GPU, and FPGA options. (Users who have nondisclosure agreements with Intel can access prerelease hardware in the NDA partition of the DevCloud.) Let’s request a node and see what devices are available:

```
$ qsub -I -l nodes=1:gen11:ppn=2
```

This command requests interactive access to single node with Intel® Processor Graphics Gen11. SYCL provides several built-in selectors, in addition to the two you’ve already seen: `default_selector_v`, `gpu_selector_v`, `cpu_selector_v`, and `accelerator_selector_v`. Note that Intel also provides `fpga_selector` and `fpga_emulator_selector` extensions for FPGA development. They are in the `sycl/ext/intel/fpga_device_selector.hpp` header. See the chapter on **FPGA Flow** in the **Intel® oneAPI Programming Guide** for more information about using SYCL on FPGAs.
The built-in selectors are mainly for convenience, but they can be robust when combined with exception handling, e.g.:

```cpp
sycl::device d;
try {
    d = sycl::device(sycl::gpu_selector_v);
} catch (sycl::exception const &e) {
    d = sycl::device(sycl::cpu_selector_v);
}
```

However, the SYCL runtime is still selecting the device. We may want more control, especially if multiple devices are available.

The following program lists the platforms and devices that are available in our compute node (note that you can also get this information using the `sycl-1s` command-line utility):

```cpp
#include <sycl/sycl.hpp>

int main()
{
    for (auto platform : sycl::platform::get_platforms())
    {
        std::cout << "Platform: " << platform.get_info<sycl::info::platform::name>() << std::endl;
        for (auto device : platform.get_devices())
        {
            std::cout << "\tDevice: " << device.get_info<sycl::info::device::name>() << std::endl;
        }
    }
}
```

$ icpx -fsycl show_platforms.cpp -o show_platforms
$ ./show_platforms

Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
Device: Intel(R) FPGA Emulation Device
Platform: Intel(R) OpenCL
Device: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
Platform: Intel(R) OpenCL HD Graphics
Device: Intel(R) UHD Graphics [0x9a60]
Platform: Intel(R) Level-Zero
Device: Intel(R) UHD Graphics [0x9a60]

SYCL platforms are based on the OpenCL™ platform model, in which a host is connected to accelerator devices. This is apparent in the example output above. This system has an OpenCL platform and a oneAPI Level Zero platform. Each platform has a device where a SYCL program can submit work. We have two GPU platforms, depending on whether we want to use the OpenCL or oneAPI Level Zero backend. We also have CPU and FPGA emulation platforms. This information allows us to create queue to submit work to either of these devices, e.g.:
The Parallel Universe

```cpp
#include <sycl/sycl.hpp>

int main()
{
  auto platforms = sycl::platform::get_platforms();

  sycl::queue Q1(platforms[1].get_devices()[0]);
  sycl::queue Q2(platforms[3].get_devices()[0]);

  std::cout << "Q1 mapped to "
            << Q1.get_device().get_info<sycl::info::device::name>()
            << std::endl;

  std::cout << "Q2 mapped to "
            << Q2.get_device().get_info<sycl::info::device::name>()
            << std::endl;
}
```

$ ipcx -fsycl map_queues.cpp -o map_queues
$ ./map_queues

Q1 mapped to 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
Q2 mapped to Intel(R) UHD Graphics [0x9a60]

Note that the devices are hard-coded in the previous example, so remember to update the platform indices if you try this program on one of your systems.

Querying the SYCL Queue and Device

Queue creation is visible in the previous example codes, but this may not always be the case. For example, a SYCL queue is typically passed to oneAPI library functions. Therefore, it may be necessary to query the queue for information. What's the target device that the queue is mapped to? What's the backend API for the device? Is it an in-order queue (i.e., kernels must be executed in the order that they were submitted)? What are the vector widths or maximum work-item dimensions of the target device?

A library developer may use this information to select an optimal code path. Consequently, the SYCL `queue` class provides several member functions to query information: `get_backend()`, `get_context()`, `get_device()`, `is_in_order()`, etc. Likewise, the SYCL `device` class provides member functions to query device characteristics [e.g., the `is_cpu()`, `is_gpu()`, and `get_info()` functions]. The `get_info()` function in particular can be used to gather detailed information about the target device: vendor, vector widths, maximum work-item and image dimensions, memory characteristics, etc. The SYCL 2020 Specification contains the complete lists of `device information descriptors` and `device aspects` that can be queried.

Using such information to optimize code is beyond the scope of this article, but it will be the subject of a future article.
Custom Selectors

Each of the device selectors we've seen so far has been a built-in selector provided by the SYCL implementation. Behind the scenes, each device selector is implemented as a C++ callable that accepts a device and returns a score. The SYCL implementation calls the device selector to score every available device in the system, finally selecting the device with the highest score.

We can write our own custom device selectors by writing callables of the same form. As a simple first example, we can write a device selector with the same behavior as the built-in CPU selector by assigning a positive score to all CPU devices and a negative score to all other devices:

```cpp
auto my_cpu_selector = [](const sycl::device& d) {
    if (d.is_cpu()) {
        return 1;
    } else {
        return -1;
    }
};
sycl::queue Q(my_cpu_selector);
```

Running on: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz

Because a device selector is just a function, we're free to use any properties of a device (e.g., aspects or device information descriptors) in conjunction with other variables in our program (e.g., command-line arguments) to score devices. This is very powerful, giving us complete control over how devices are scored and selected, and allowing us to ensure that the selected device meets our application's requirements. The example below gives a taste of what's possible, showing a device selector that ignores devices that do not support double-precision floating-point arithmetic, and which can be configured to prefer GPUs via a Boolean variable:

```cpp
bool prefer_gpus = true; // e.g., from command-line or configuration file
auto my_selector = [=](const sycl::device& d) {
    // Ignore devices that do not support double-precision
    if (!d.has(sycl::aspect::fp64)) {
        return -1;
    }
    // Optionally prefer GPUs
    if (prefer_gpus && d.is_gpu()) {
        return 1;
    } else {
        return 0;
    }
};
sycl::queue Q(my_selector);
```

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others. For more complete information about compiler optimizations, see our Optimization Notice.
SYCL recently added the `aspect_selector` function to help select devices that meet the programmer's requirements. For example, the following statement selects a GPU device that supports half-precision while excluding emulated, fixed-function devices:

```cpp
auto dev = sycl::device{sycl::aspect_selector{
    std::vector{sycl::aspect::fp16, sycl::aspect::gpu}, // allowed aspects
    std::vector{sycl::aspect::custom, sycl::aspect::emulated} // disallowed aspects
}};
```

At the time of writing, `aspect_selector` is not yet supported by the Intel® oneAPI DPC++/C++ Compiler, but it should be available soon.

### Changing the oneDPL Execution Policy

The article, *The Maxloc Reduction in oneAPI* (The Parallel Universe, Issue 48), showed how oneDPL uses the execution policy to offload functions to accelerators. The code examples simply used the `oneapi::dpl::execution::dpcpp_default` policy, so let's see how to use SYCL queues to modify the execution policy to explicitly control where oneDPL functions run:

```cpp
#include <oneapi/dpl/execution>

int main()
{
    sycl::queue Q1(sycl::gpu_selector_v);
    auto gpu_policy = oneapi::dpl::execution::make_device_policy(Q1);
    std::cout << "GPU execution policy runs oneDPL functions on "
    << gpu_policy.queue().get_device().get_info<sycl::info::device::name>()
    << std::endl;

    sycl::queue Q2(sycl::cpu_selector_v);
    auto cpu_policy = oneapi::dpl::execution::make_device_policy(Q2);
    std::cout << "CPU execution policy runs oneDPL functions on "
    << cpu_policy.queue().get_device().get_info<sycl::info::device::name>()
    << std::endl;
}
```

```
$ icpx -fsycl onedpl_policy_example.cpp -o onedpl_example
$ ./onedpl_example
GPU policy runs oneDPL functions on Intel(R) UHD Graphics [0x9a60]
CPU policy runs oneDPL functions on 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
```

The previous program creates queues using the built-in CPU and GPU selectors, then uses these queues to set the oneDPL execution policy. We could just as easily have queried the platforms and devices and instantiated the queues as shown previously:
... auto platforms = sycl::platform::get Platforms();

sycl::queue Q1(platforms[3].get_devices()[0]);
auto gpu_policy = oneapi::dpl::execution::make_device_policy(Q1);

sycl::queue Q2(platforms[1].get_devices()[0]);
auto cpu_policy = oneapi::dpl::execution::make_device_policy(Q2);
...

Once again, the devices are hardcoded in the previous example so remember to update the platform indices for your system.

We've only scratched the surface of what SYCL provides for device discovery, and how programs can use platform and device information. Expect to see more about this in future issues of The Parallel Universe, especially as multi-device systems become more prevalent, and programmers begin targeting algorithms to specific devices.

Additional Resources

- **Intel® Developer Cloud** – This is a free, one-stop-shop for experimenting with oneAPI on a variety of Intel hardware.
- **SYCL™ 2020 Reference Guide** – Short of the SYCL™ 2020 Specification itself, this reference is the source for easily digestible information. The Device selection, Platform, Context, and Device class descriptions figure highly in this article.
- **Chapter 12: Device Information** of Data Parallel C++: Mastering DPC++ for Programming Heterogeneous Systems using C++ and SYCL provides a good overview of device discovery, though the syntax for some code examples is now out-of-date.
- The oneAPI-samples repository contains hundreds of code samples, illustrating programming with SYCL and oneAPI libraries.
- The Intel® oneAPI Programming Guide gives a basic overview of the Intel tools for oneAPI.
- The oneAPI GPU Optimization Guide is a living document of coding advice for best performance using oneAPI.
This article provides an overview of the application interface, philosophy, purpose, and vision of Level Zero. We will look at the basic architecture of Level Zero and its benefits for low-level access control to compute unit resources. It can be used with language extensions like OpenMP* and SYCL*. We will briefly highlight how the interaction between Level Zero and the SYCL C++ language extension abstraction layer is exposed to the application software developer.

Level Zero is designed as a low-level API to configure and manage access to any number of offload devices. In doing so, it also provides the abstraction layer that enables C++ standard-compliant heterogeneous computation without interfering with the program flow. This makes code portable across different runtime environments. Being aware of the Level Zero API backend allows you to go beyond the abstraction of the SYCL or OpenMP language extension, thus increasing your level of control.

The Level Zero interface is part of the oneAPI Specification. It complements the API-based and direct
programming models of oneAPI with bare-metal access to CPUs, GPUs, and accelerators. Intel's reference implementation targeting Intel® GPUs as part of the Intel® oneAPI Base Toolkit and its usage with the Intel® oneAPI DPC++/C++ Compiler are also well documented. After reading this article, you should have the resources at your fingertips to dive deeper and get started using a Level Zero runtime or contemplating your own runtime.

Unlock Heterogeneous Computing

Intel's first implementation of Level Zero targets Intel GPUs. However, the vision and potential of Level Zero goes far beyond that. It has the potential to create a tailored abstraction for specific device requirements. It can be adapted to support broader sets of language features such as function pointers, memory, and I/O. The API is designed to work across a variety of compute devices, including CPUs, GPUs, Field Programmable Gate Arrays (FPGAs), and other accelerator architectures.

Level Zero can coexist with other low-level APIs such as OpenCL* and Vulkan*. However, it is intended to evolve independently to permit the high-level oneAPI and SYCL developer experience to stay hardware agnostic and as architecture independent and flexible as possible. It also provides explicit controls that higher-level runtime APIs and libraries taking advantage of SYCL may desire. Level Zero is fully open-source, with its specification, source repository, and Intel GPU compute runtime implementation easily accessible on GitHub.

In short, Level Zero unlocks the vision of heterogeneous compute and offers the flexible open backend to make true choice of offload compute a reality. It provides the ability to explicitly control system-level interfaces through capabilities like:

- Device discovery
- Memory allocation
- Peer-to-peer communication
- Interprocess sharing
- Kernel submission
- Asynchronous execution and scheduling
- Synchronization primitives
- Metrics reporting
- System management

Let's start with the key concepts of Level Zero.

Level Zero Basics

Level Zero sits below the application layer. It can be adopted as an abstraction interface between a C++ application and target device properties (Figure 1). These could include CPUs as well as other compute devices. In doing so, it enables the developer to interact seamlessly with shared device resources and to
dispatch workloads to a specific Level Zero driver supported device. The driver adds a supported device to an available device list, and any SYCL queue can be mapped to and submit work to that device. If we do not need to access specific device properties, or resources shared between multiple Level Zero devices, those devices thus behave exactly like any other device using a different SYCL backend API.

The real strength of Level Zero lies in its low-level control and support for device-specific memory sharing or synchronization context objects. This not only adds more transparency for a device, but it also gives the Level Zero API additional configurability for heterogeneous offload target devices.

The sequential flow of Level Zero device detection and selection is as follows.

### Level Zero Loader

Accessing an offload device or accelerator starts with the Level Zero Loader. It discovers Level Zero drivers for devices in the system. The loader project also contains the [Level Zero headers and libraries](#) that allow us to build and interact with Level Zero implementations. Driver initialization and discovery are illustrated in the following code example:

```c
// Initialize the driver
zeInit(0);

// Discover all the driver instances
uint32_t driverCount = 0;
zeDriverGet(&driverCount, nullptr);

ze_driver_handle_t* allDrivers = allocate(driverCount * sizeof(ze_driver_handle_t));
zeDriverGet(&driverCount, allDrivers);

// Find a driver instance with a GPU device
ze_driver_handle_t hDriver = nullptr;
ze_device_handle_t hDevice = nullptr;
for(i = 0; i < driverCount; ++i) {
    uint32_t deviceCount = 0;
zeDeviceGet(allDrivers[i], &deviceCount, nullptr);

    ze_device_handle_t* allDevices = allocate(deviceCount * sizeof(ze_device_handle_t));
zeDeviceGet(allDevices, &deviceCount, allDevices);

    for(d = 0; d < deviceCount; ++d) {
    ...
}
```

![Figure 1. Level Zero interface in the context of the oneAPI backend architecture](#)
This is followed by the creation of a context object for the purpose of managing memory, command queues, modules, synchronization objects, and such. The use of contexts becomes especially important if managing system resources that could be shared by multiple devices. A simple example for the scenario of shared memory is given below:

```c
ze_device_properties_t device_properties {}; device_properties.stype = ZE_STRUCTURE_TYPE_DEVICE_PROPERTIES; zeDeviceGetProperties(allDevices[d], &device_properties);
if(ZE_DEVICE_TYPE_GPU == device_properties.type)
{
    hDriver = allDrivers[i];
    hDevice = allDevices[d];
    break;
}
free(allDevices);
if(nullptr != hDriver)
{
    break;
}
free(allDrivers);
if(nullptr == hDevice)
    return; // no GPU devices found
```

// Create context(s)
zeContextCreate(hDriver, &ctxtDesc, &hContextA);
zeContextCreate(hDriver, &ctxtDesc, &hContextB);
zeMemAllocHost(hContextA, &desc, 80, 0, &ptrA);
zeMemAllocHost(hContextB, &desc, 88, 0, &ptrB);

### Scheduling Model

Any commands will be scheduled and dispatched to Level Zero devices as depicted in the scheduling diagram in Figure 2.
Commands are appended to command lists that represent a sequence of commands to be executed on the offload compute unit or accelerator. A command list can be recycled by resetting the list, without needing to create it again. It can be reused by submitting the same sequence of commands several times, without needing to reappend commands.

Command lists are then submitted to a command queue for execution. A queue is a logical object associated to a physical input stream in the device, which can be configured as synchronous or asynchronous and can be organized in queue groups. This scheduling model translates into the source code flow shown in Figure 3.

Figure 2. Level Zero scheduling model

Figure 3. Level Zero scheduling execution flow
Immediate Command Lists

Command list handling can also be optimized to help manage latency. High priority tasks requiring guaranteed response times can be handled via low-latency immediate command lists. This is a special type of command list dedicated to very low-latency submission usage models.

The command list and its implicit command queue are created using a command queue descriptor. Commands appended into an immediate command list are immediately executed on the device. Commands appended into an immediate command list may execute synchronously by blocking until the command is complete. An immediate command list does not need to be closed or reset after completion. However, usage will be honored, and expected behaviors will be followed. The following pseudo-code demonstrates a basic sequence for creation and usage of immediate command lists:

```c
// Create an immediate command list
ze_command_queue_desc_t commandQueueDesc = {
    ZE_STRUCTURE_TYPE_COMMAND_QUEUE_DESC,
    nullptr,
    computeQueueGroupOrdinal,
    0, // index
    0, // flags
    ZE_COMMAND_QUEUE_MODE_DEFAULT,
    ZE_COMMAND_QUEUE_PRIORITY_NORMAL
};
ze_command_list_handle_t hCommandList;
zeCommandListCreateImmediate(hContext, hDevice, &commandQueueDesc, &hCommandList);

// Immediately submit a kernel to the device
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0, nullptr);
...
```

Imagine the wealth of closed-feedback-loop use cases where immediate action may be required upon determining an industrial system error condition. The concept of immediate command lists introduces GPU offload compute to use cases that require guaranteed response times.

Application Developer Benefits

Now that we’ve covered the key design principles of Level Zero, let’s have a look at how application developers can interact with it from their SYCL-based applications.

Device Selection

Any device with a Level Zero driver implementation can be initialized and used by the application developer. Intel’s Level Zero implementation is available on GitHub. It can be used as a reference for other devices that want to take advantage of oneAPI cross-architecture heterogeneous compute support.

To select a specific offload device, the SYCL_DEVICE_FILTER environment variable can be used. Using it affects all the device query functions and device selectors. To check on the availability of devices for
use with SYCL on a running system, simply use the `sycl-ls` command, e.g.:

```sh
$ sycl-ls
 [opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R) FPGA Emulation Device 1.2 [2022.13.3.0.16_160000]
 [opencl:cpu:1] Intel(R) OpenCL, 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz 3.0 [2022.13.3.0.16_160000]
 [opencl:gpu:2] Intel(R) OpenCL HD Graphics, Intel(R) Iris(R) Xe Graphics 3.0 [31.0.101.3358]
 [ext_oneapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) Iris(R) Xe Graphics 1.3 [1.3.23828]
 [host:host:0] SYCL host platform, SYCL host device 1.2 [1.2]
```

We can think about device properties as follows. Driver objects represent a collection of physical devices in the system. More than one driver may be available. For example, one driver may support accelerators from one vendor, and another driver may support an accelerator from a different vendor. Context objects represent device or system resources. Their primary purpose is creation and management of resources that may be used by multiple devices. Device objects represent a physical device in the system. The device discovery API is used to enumerate devices in system. The `zeDeviceGet()` function is used to query the number of Level Zero devices supported by a driver and obtain any devices objects, which are read-only global constructs. Every device has a 16-byte universal unique globally identifier (UUID) assigned to it. A device handle is used during creation and management of resources that are specific to a device.

If we want to take advantage of the additional capabilities of Level Zero, we first want to enable interoperability between SYCL-enabled C++ code and the Level Zero API. If we intend to directly interact with Level Zero device-specific context objects from within a C++ application, the following header files need to be included in our source code in the order shown:

```cpp
#include "level_zero/ze_api.h"
#include "sycl/backend/level_zero.hpp"
```

The Level Zero backend is added to the `sycl::backend` enumeration with:

```cpp
eenum class backend {
    // ...
    ext_oneapi_level_zero,
    // ...
};
```

This way, you can use the `sycl::get-native` API from the SYCL namespace to request Level Zero data structures underlying SYCL objects.

```cpp
template <backend BackendName, class SyclObjectT>
auto get_native(const SyclObjectT &Ob)
    -> backend_return_t<BackendName, SyclObjectT>
ext_oneapi_level_zero,
```
Please consult the Intel® oneAPI Level Zero Backend Specification for more details. A given SYCL queue will attach to an available device on the system:

```cpp
try {
    vector<device> sub_devices = ...;
    for (auto &d : sub_devices)
    {
        // Each queue is in its own context, no data sharing across them.
        auto q = queue(d);
        q.submit(
            [d](handler & cgh) {...});
    }
}
```

The device is then used and has an execution queue assigned to it like any other SYCL device. Where it all gets interesting is when we want to access device-specific resources, or resources that are shared between host and offload execution devices. Let’s take unified shared memory (USM) as an example for accessing device-specific resources. The outline below reflects how Intel implemented USM, but identical functionality could be implemented for other Level Zero libraries.

**Unified Shared Memory**

Memory is visible to the upper-level software stack as unified memory with a single virtual address space covering both CPU and GPU. Linear, unformatted memory allocations are represented as pointers in the host application. A pointer on the host has the same size as a pointer on the device. There are three ways to allocate memory using the SYCL namespace:

1. `sycl::malloc_device`
   - Allocation can only be accessed by the specified device, but not by the host or other devices in the context.
   - The data always stays on the device and is the fastest available for kernel execution.
   - Explicit copy is needed for transferring data to the host or other devices in the context.

2. `sycl::malloc_host`
   - Allocation can be accessed by the host and any other device in the context.
   - The data always stays on the host and is accessed via Peripheral Component Interconnect (PCI) from the devices.
   - No explicit copy is needed for synchronizing the data with the host or devices.

3. `sycl::malloc_shared`
   - Allocation can only be accessed by the host and the specified device.
   - The data can migrate (operated by the Level Zero driver) between the host and the device for faster access.
   - No explicit copy is necessary for synchronizing between the host and the device, but it is needed for other devices in the context.
The equivalent lower-level Level Zero calls invoked this way are `zeMemAllocDevice`, `zeMemAllocHost`, and `zeMemAlloc_shared`, respectively. If you'd like to delve a bit deeper into the implementation behind this, please check out the Level Zero Core API Specification.

USM is only one example of the advanced device awareness that Level Zero provides. More details can be found in the Level Zero Core Programming Guide.

**Summary and Next Steps**

Level Zero provides a rich set of interfaces to schedule work and manage memory on compute units and accelerators. It provides all the services for loading and executing programs, allocating memory, and managing heterogeneous workloads. It does so with an open interface that can be customized to your specific hardware configuration, while maintaining the abstraction and flexibility required for workloads to run on less specialized setups.

Objects defined in Level Zero, such as command queues and command lists, allow for low-level control of the underlying hardware. With these and available optimization techniques, high-level programming languages and applications may execute workloads with close-to-metal latencies for higher performance.

In conjunction with SYCL, it can be seamlessly used and accessed using C++. Furthermore, we are currently leveraging our experiences with Python*, Julia*, and Java* to provide better language runtime support in Level Zero across languages. We would like to invite the open source community and the industry as a whole to contribute to Level Zero to make it an even more versatile and powerful interface for multi-architectural choice.

**Additional Resources**

- Intel® Graphics Compute Runtime for oneAPI Level Zero and OpenCL Driver
- Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference: Intel oneAPI Level Zero
- oneAPI Level Zero Specification
- oneAPI Level Zero Headers, Loader, and Validation Layer