intel software

THE PARALLEL UNIVERSE

Maintaining Performant AI in Production

Deep Learning Model Optimizations Made Easy (or at Least Easier)

The Habana Gaudi2* Processor for Deep Learning

Issue 50 2022

Contents

Sign up for future issues

Letter from the Editor

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

Just Got Back from Intel® Innovation

The Intel® Developer Forum was a big part of my early career at Intel, so I was disappointed when it was discontinued. As I write this, Intel® Innovation 2022 has just wrapped up. It reminded me a lot of the Intel Developer Forum. There were plenty of big announcements, technical sessions, breakouts, and catching up with colleagues, collaborators, and customers. The link above has the full agenda, speaker info, and content, so I won't try to recap the entire conference in a few paragraphs. The Day 1 and Day 2 highlights cover the key announcements, such as:

- The Intel® Developer Cloud will make new and future hardware platforms, like 4th Gen Intel® Xeon® Scalable processors and Intel® Data Center GPUs, available for prelaunch development and testing.
- The new Intel® Geti™ platform will enable enterprises to quickly develop and deploy computer vision AI.
- Intel previewed future high-volume, system-in-package capabilities that will enable pluggable co-package photonics for a variety of applications.
- The open oneAPI specification will now be managed by Codeplay, an Intel subsidiary.
- Intel released three new AI reference kits focused on healthcare use-cases.

Much of this issue focuses on sustainable AI, model optimization, and deep learning performance. Our feature article, **Maintaining Performant AI in Production**, covers MLOps, an often-overlooked component of the AI workflow. It describes how to build an MLOps environment using the Intel® AI Analytics Toolkit, MLflow*, and AWS*. Sustainable AI is becoming an important topic as the use of AI and the size of models increases. This is discussed in our second article, **Deep Learning Model Optimizations Made Easy (or at Least Easier)**. Along these same lines, **The Habana Gaudi2* Processor for Deep Learning** describes improvements to this already efficient architecture with impressive MLPerf benchmark results to back it up. **PyTorch* Inference Acceleration with Intel® Neural Compressor** describes a new, open-source Python* library for model compression that reduces the model size and increases the speed of deep learning inference on CPUs or GPUs. Finally, **Accelerating PyTorch with Intel® Extension for PyTorch** describes our open-source extension to boost performance.

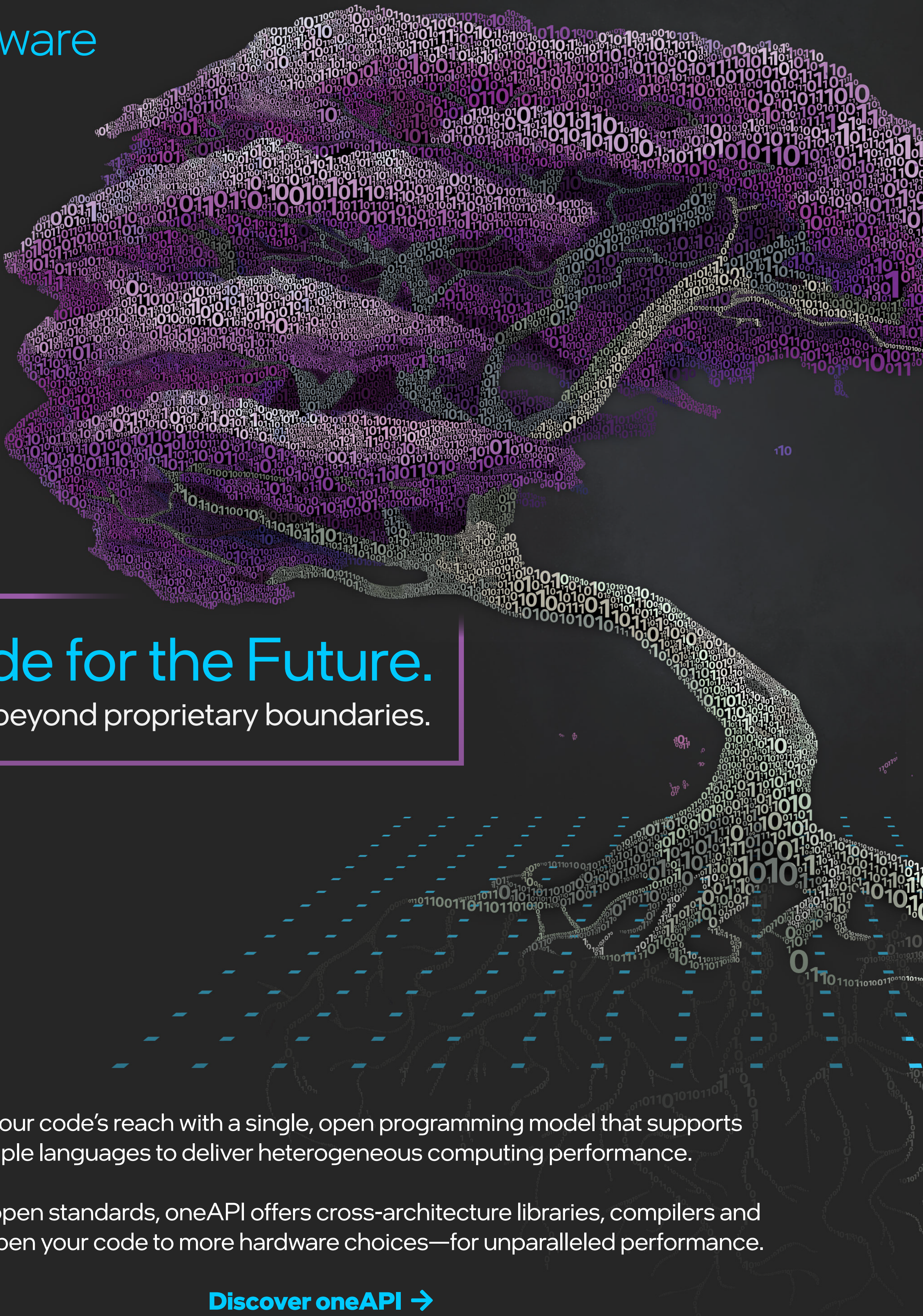
Sign up for future issues

From there, we turn our attention to heterogeneous computing using Python. In **Accelerating Python Today**, Editor Emeritus James Reinders helps us get ready for the Cambrian explosion in accelerator architectures.

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

Henry A. Gabb
October 2022

intel. software



Code for the Future.

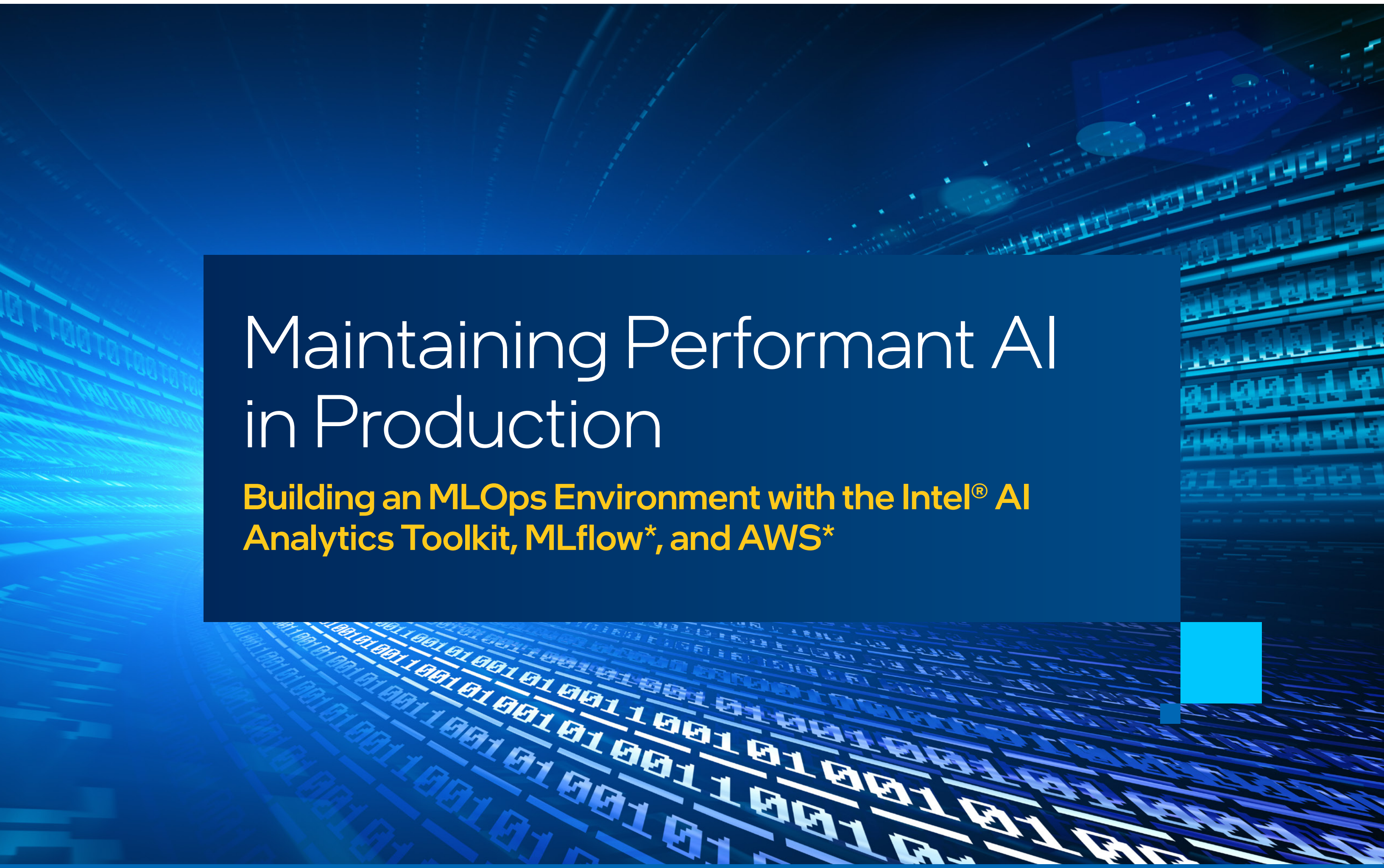
Grow beyond proprietary boundaries.

oneAPI

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 →



Maintaining Performant AI in Production

Building an MLOps Environment with the Intel® AI Analytics Toolkit, MLflow*, and AWS*

Eduardo Alvarez, Senior AI Solutions Engineer, Intel Corporation

When strictly speaking about taking machine learning (ML) applications from R&D to production, MLOps is a critical component often overlooked. The solutions to this challenge are often best addressed on a case-by-case basis. In this article, I offer a general blueprint for building an MLOps environment in the cloud with open-source tools to ensure that we develop and deploy performant models (**Figure 1**). As the backbone for our ML component, I will leverage the Intel® AI Analytics Toolkit, specifically the Intel® Distribution for Python* and the Intel® Extension for PyTorch*, to drive higher performance during training and inference. After reading this article, you will have at your disposal the mechanisms to perform fast model experimentation, model management, and serving, all in a serverless cloud environment.

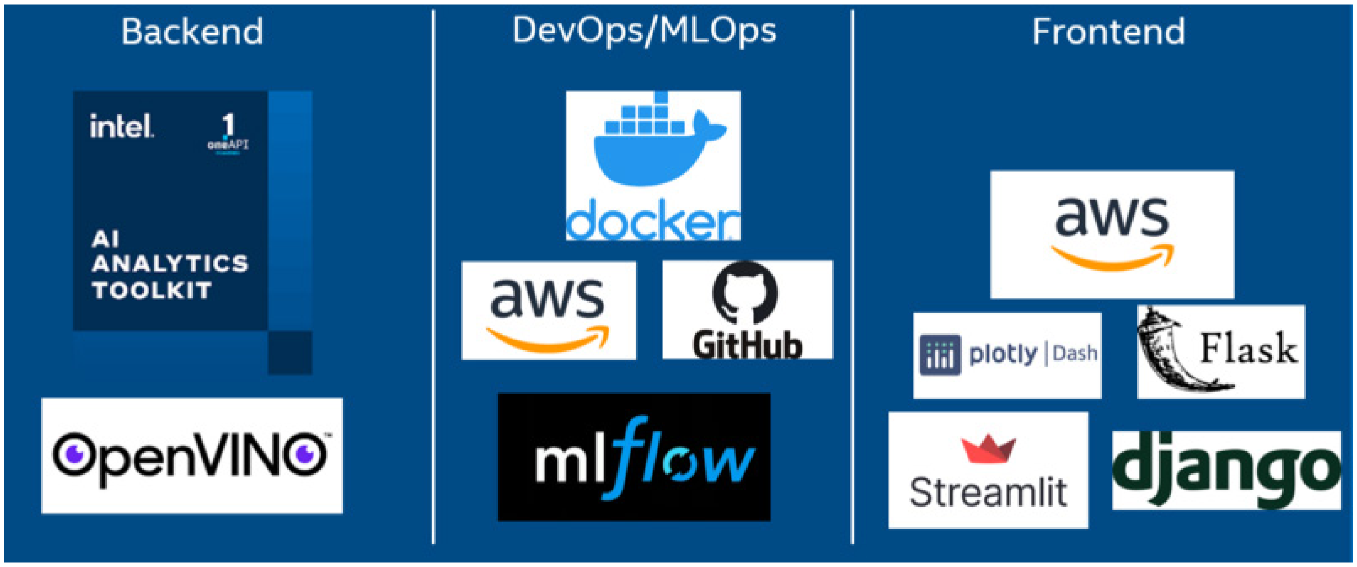


Figure 1. Image showing the various tools that play a role in the server-side, operational, and client-side components of a solution that utilizes this MLOps blueprint. We will not be focusing on Docker*, GitHub*, or any of the frontend components in this article.

MLOps—How Models Reach Escape Velocity

MLOps marries ML with the agility and resilience of DevOps—tying ML assets to your CI/CD pipelines for stable deployment into production environments. This creates a unified release process that addresses model freshness and drifts concerns. Without a proper MLOps pipeline, ML application engineers cannot deliver high-quality ML assets to the business unit. It becomes just one big science experiment inside of R&D.

The main reason there is a lack of investment in MLOps stems from a poor understanding of the impacts of model/data drift and how they can affect your application in production. Model drift refers to the degradation of model performance due to changes in data and relationships between input and output variables. Data drift is a type of model drift where the properties of the independent variables change. Examples of data drift include changes in the data due to seasonality, changes in consumer preferences, the addition of new products, etc. A well-established MLOps pipeline can mitigate these effects by deploying the most relevant models to your data environment. At the end of the day, the goal is to always have the most performant model in our production environment.

MLOps Solution—MLflow* ML Lifecycle Management Open Source Tool

MLflow* is an open-source ML lifecycle management platform (**Figure 2**). MLflow works with any ML library that runs in the cloud. It's easily scalable for big data workflows. MLflow is composed of four parts:

- **MLflow Tracking** allows you to record and query code, data, configurations, and results of the model training process.
- **MLflow Projects** allows you to package and redeploy your data science code to enable reproducibility on any platform.
- **MLflow Models** is an API for easy model deployment into various environments.
- **MLflow Model Registry** allows you to store, annotate, and manage models in a central repository.

I will cover how to set up a password-protected MLflow server for your entire team.

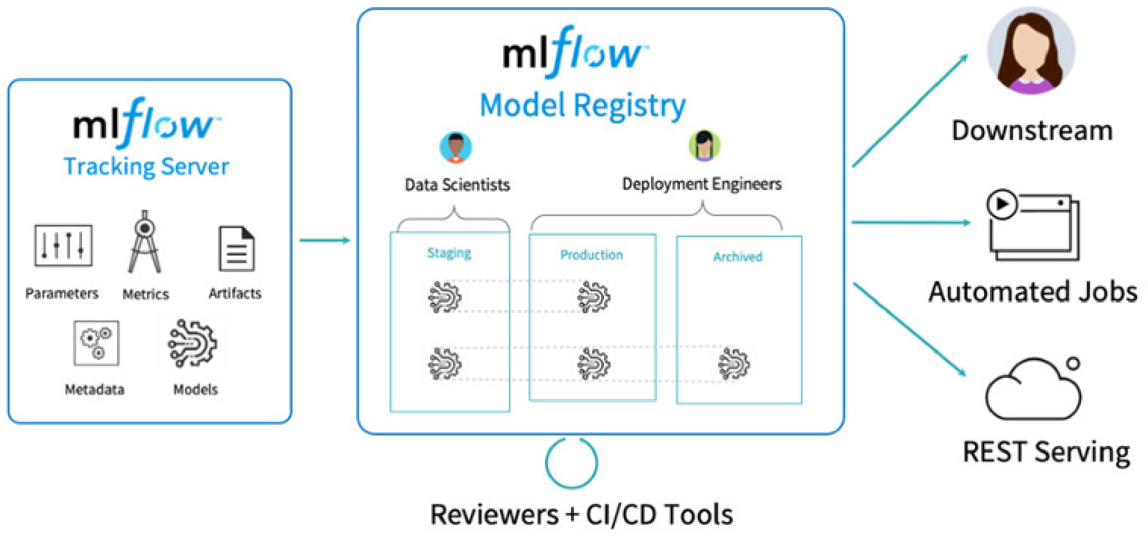


Figure 2. MLflow* tracking and registry components. (Image courtesy of Databricks.)

MLflow Server—AWS* Solution Infrastructure Design

The cloud component of our MLOps environment is very straightforward. I require the infrastructure to train, track, register and deploy ML models. The diagram below depicts our cloud solution, which consists of a local host (or another remote host) that uses various MLflow APIs to communicate with the resources in a remote host (**Figure 3**). The remote host is responsible for hosting our tracking server and communicating with our database and object storage.

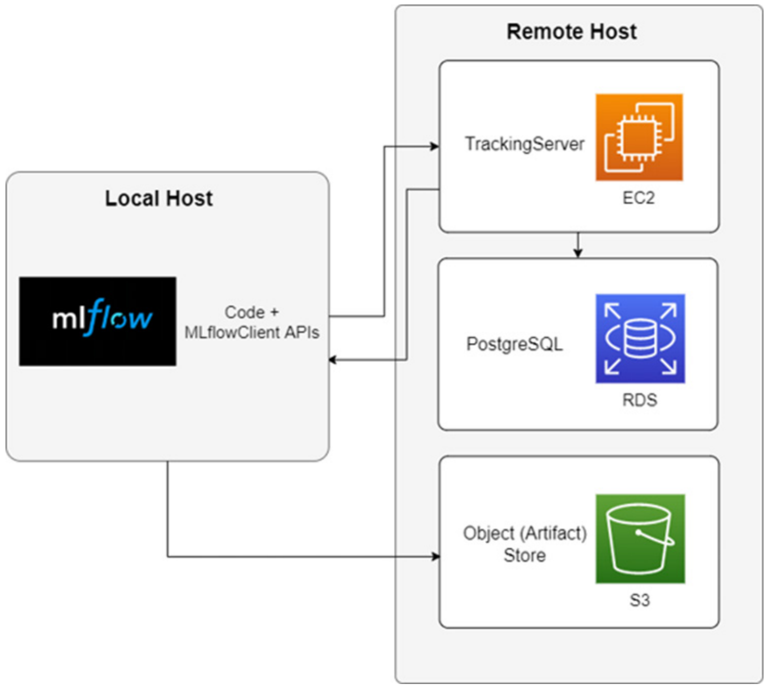


Figure 3. AWS* solution architecture depicting the components that reside on the remote and local hosts.

This article includes step-by-step instructions to set up this architecture. You will find that I leverage free-tier AWS services. This is not recommended if you are establishing a production-level environment.

Table 1 shows the resources used to set up the product demo. You are welcome to use this as a blueprint and adjust it to your needs. The selected instances are all 3rd Gen Intel® Xeon® Scalable processors with varying vCPU and GiB capacities.

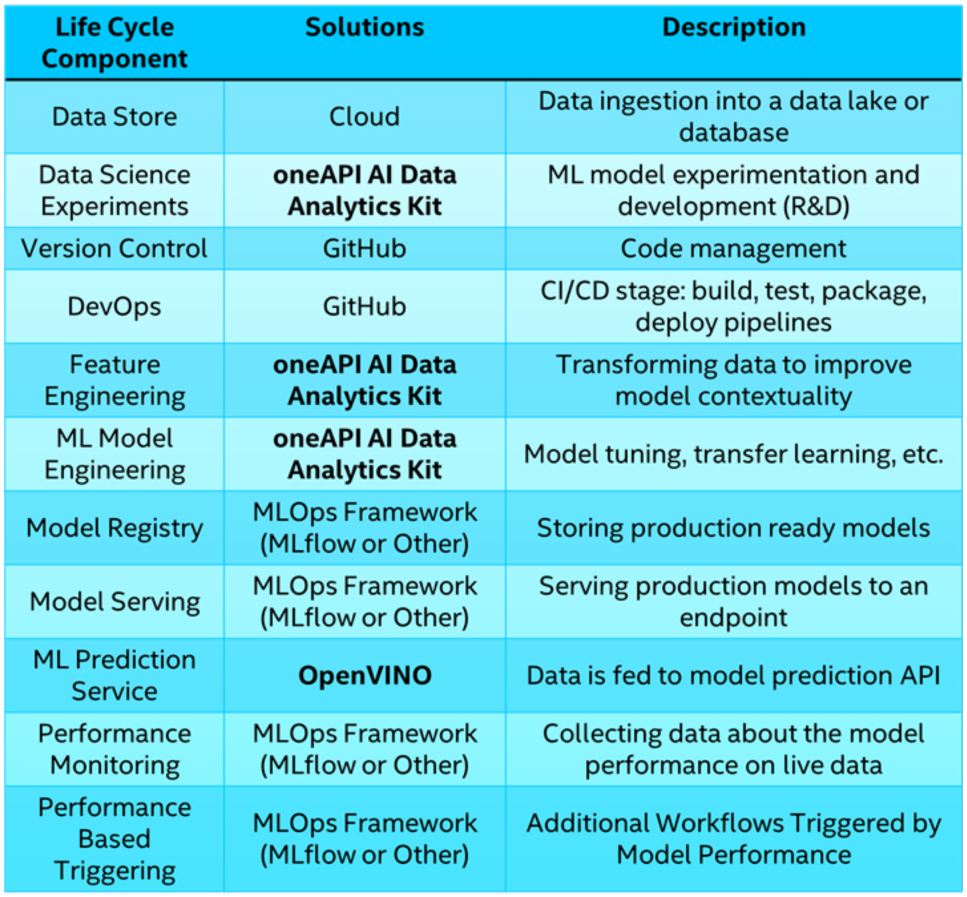
AWS Service	Purpose	Resource Requirements
EC2	Host MLFlow Sever	m6i.xlarge (4 vCPU and 16 GiB Memory) and 8GB Block Storage
EC2	Host Jupyter Server	m6i.4xlarge (16 vCPU and 64 GiB Memory) and 256GB Block Storage
RDS	MLFlow Backend Store	Any suitable DB instance and at least 100GB of Storage
S3	MLFlow Object Store	Default Configuration (establish backup frequency as needed)

Table 1. The AWS* services and resource requirements for various components of this blueprint that have been tested in a general-purpose environment.

Machine Learning Frameworks — Intel® AI Analytics Toolkit

Our MLOps environment requires a set of performant ML frameworks to build our models. The Intel AI Analytics Toolkit provides Python tools and frameworks built using oneAPI libraries for low-level compute optimizations. By leveraging this toolkit, I can underpin our MLOps environment with the ability to deliver high-performance, deep learning training on Intel® XPUs and integrate fast inference into our workflow.

Table 2 illustrates an end-to-end ML lifecycle, and how each component is addressed in our MLOps blueprint. Components of this workflow where the Intel AI Analytics Toolkit help boost performance and drive down compute costs are highlighted in bold.

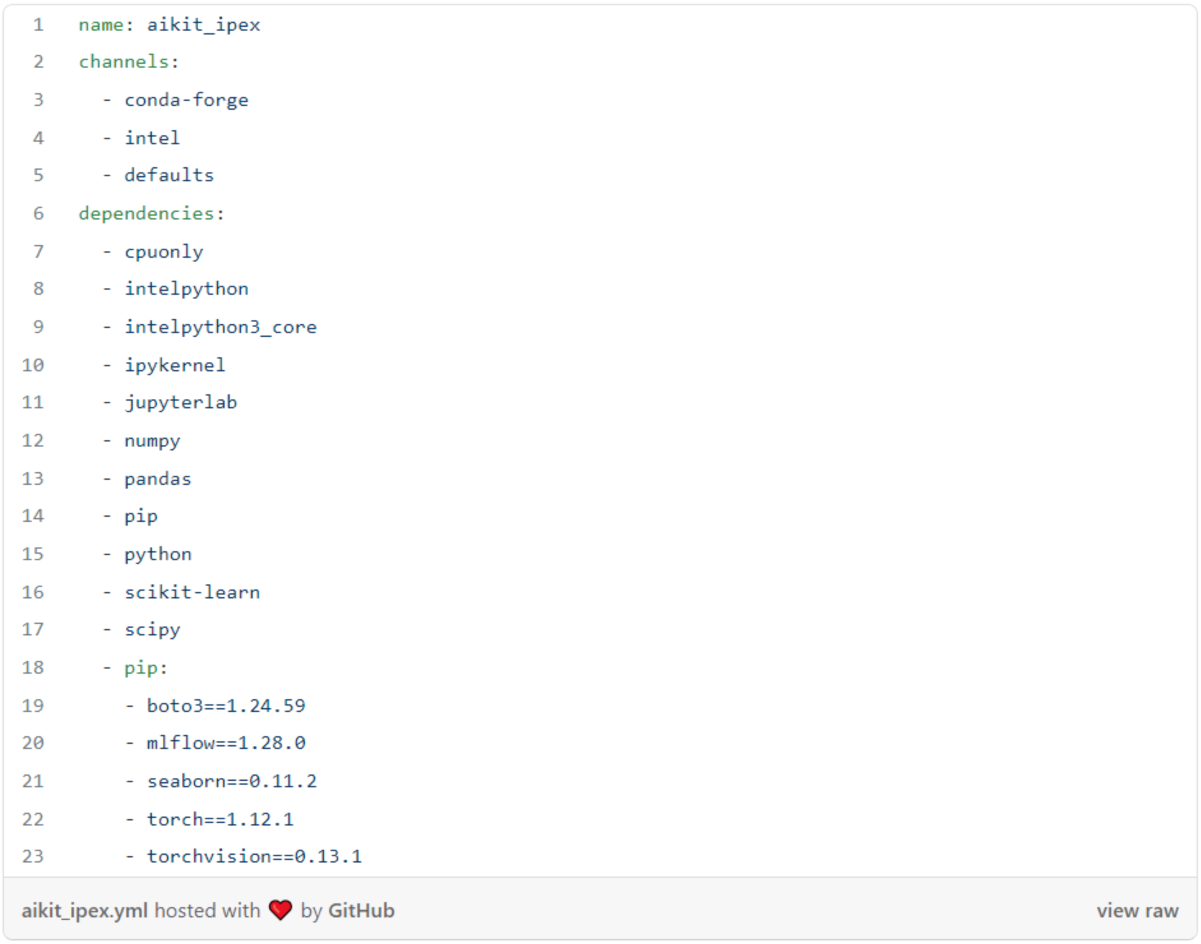


Life Cycle Component	Solutions	Description
Data Store	Cloud	Data ingestion into a data lake or database
Data Science Experiments	**oneAPI AI Data Analytics Kit**	ML model experimentation and development (R&D)
Version Control	GitHub	Code management
DevOps	GitHub	CI/CD stage: build, test, package, deploy pipelines
Feature Engineering	**oneAPI AI Data Analytics Kit**	Transforming data to improve model contextuality
ML Model Engineering	**oneAPI AI Data Analytics Kit**	Model tuning, transfer learning, etc.
Model Registry	MLOps Framework (MLflow or Other)	Storing production ready models
Model Serving	MLOps Framework (MLflow or Other)	Serving production models to an endpoint
ML Prediction Service	**OpenVINO**	Data is fed to model prediction API
Performance Monitoring	MLOps Framework (MLflow or Other)	Collecting data about the model performance on live data
Performance Based Triggering	MLOps Framework (MLflow or Other)	Additional Workflows Triggered by Model Performance

Table 2. End-to-end ML lifecycle solutions.

This article is not about setting up conda environments, but I provide an environment configuration script (`aikit_ipex.yml`) below for your benefit. You can create the appropriate conda environment using the following command:

```
conda env create -f aikit_ipex.yml
```



```yaml
name: aikit_ipex
channels:
  - conda-forge
  - intel
  - defaults
dependencies:
  - cpuonly
  - intelpython
  - intelpython3_core
  - ipykernel
  - jupyterlab
  - numpy
  - pandas
  - pip
  - python
  - scikit-learn
  - scipy
  - pip:
    - boto3==1.24.59
    - mlflow==1.28.0
    - seaborn==0.11.2
    - torch==1.12.1
    - torchvision==0.13.1
```

Setting up the MLflow Remote Host

Set Up the Host Machine with AWS EC2

1. Go to your AWS management console and launch a new EC2 instance. I selected an Amazon Linux OS*, but you're welcome to use another Linux and Windows OS* from the list.
2. Create a new key pair if you don't already have one. You will need this to SSH into your instance and configure various aspects of your server.
3. Create a new security group and allow SSH, HTTPS, and HTTP traffic from Anywhere (0.0.0.0/0). For maximum security, however, it is recommended that you safelist specific IPs that should have access to the server instead of having a connection that is open to the entire internet.
4. The EC2 storage volume for your server doesn't need to be any bigger than 8–16 GB unless you intend to use this instance for other purposes. My recommendation would be that you leave this instance as a dedicated MLflow server.

Set Up the S3 Object Store

Your S3 bucket can be used to store model artifacts, environment configurations, code versions, and data versions. It contains all the vital organs of your MLflow ML management pipeline.

1. Create a new bucket from the AWS management console.
2. Enable ACLs and leave all public access blocked. ACLs will allow other AWS accounts with proper credentials to access the objects in the bucket.

Set Up the AWS RDS Postgres Database

This component of the MLflow workflow will be in charge of storing runs, parameters, metrics, tags, notes, paths to object stores, and other metadata.

1. Create a database from the AWS management console. I will be working with a PostgreSQL database. Select the Free Tier.
2. Give your database a name, assign a master username, and provide a password for this account. You will be using the information to launch your MLflow server.
3. Select an instance type. Depending on how much traffic you intend to feed through the MLflow server (i.e., how many times you will be querying models and data), you might want to provide a beefier instance.
4. You will also need to specify the storage. Again, this will depend on how much metadata you need to track. Remember, models and artifacts will not be stored here, so don't expect to need excessive space.
5. Public access is essential to allow others outside your virtual private cluster to write/read the database. Next, you can specify the safelist IPs using the security group.
6. You must create a security group with an inbound rule that allows all TCP traffic from anywhere (0.0.0.0/0) or specific IPs.
7. Launch RDS.

Install the AWS CLI and Access Instance with SSH

AWS CLI is a tool that allows you to control your AWS resources from the command line. Follow this link to install the CLI from the AWS website. Upon installing the AWS CLI, you can configure your AWS credentials to your machine to permit you to write to the S3 object store. You will need this when incorporating MLflow into scripts, notebooks, or APIs.

1. From your EC2 instance Dashboard, select your MLflow EC2 instance and click **Connect**.
2. Navigate to the **SSH** client tab and copy the SSH example to your clipboard from the bottom of the prompt.
3. Paste the SSH command into your prompt. I'm using Git Bash from the folder where I have stored my .pem file.
4. Install the following dependencies:

```
sudo pip3 install mlflow[extras] psycopg2-binary boto3

sudo yum install httpd-tools

sudo amazon-Linux-extras install nginx1
```

5. Create a nginx user and set a password:

```
sudo htpasswd -c /etc/nginx/.htpasswd testuser
```

6. Finally, I will configure the nginx reverse proxy to port 5000:

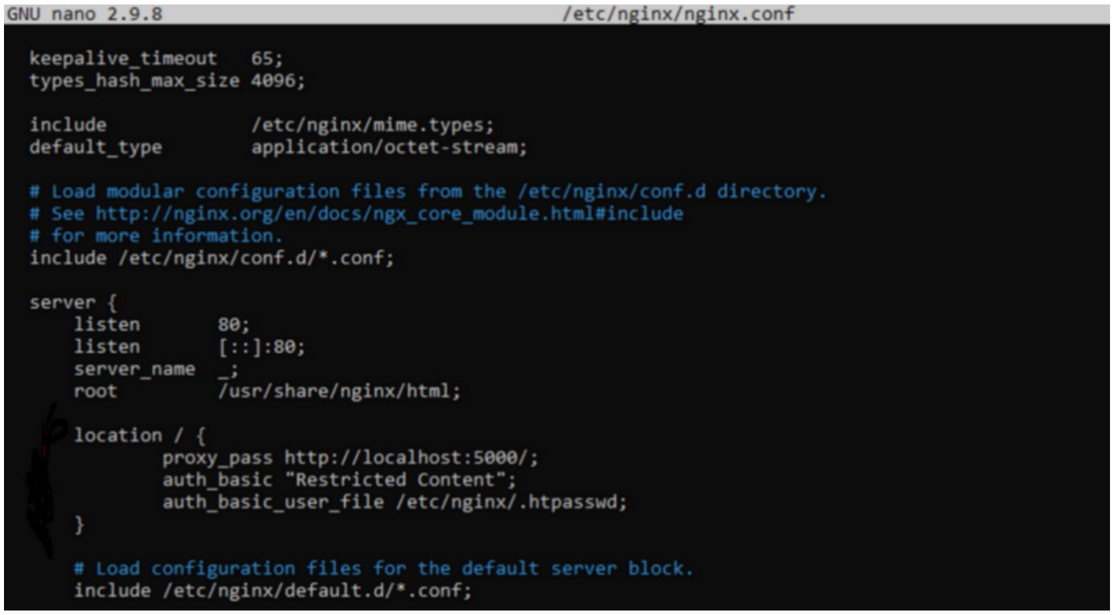
```
sudo nano /etc/nginx/nginx.conf
```

Using nano, I can edit the nginx config file to add the following information about the reverse proxy:

```
location / {
proxy_ass http://localhost:5000/;
auth_basic “Restricted Content”;
auth_basic_user_file /etc/nginx/.htpasswd;
}
```

add2nginx.txt hosted with ❤ by GitHub view raw

Your final script should look something like the console nano printout in **Figure 4**.



```
GNU nano 2.9.8                                  /etc/nginx/nginx.conf

  keepalive_timeout   65;
  types_hash_max_size 4096;

  include             /etc/nginx/mime.types;
  default_type        application/octet-stream;

  # Load modular configuration files from the /etc/nginx/conf.d directory.
  # See http://nginx.org/en/docs/ngx_core_module.html#include
  # for more information.
  include /etc/nginx/conf.d/*.conf;

  server {
      listen       80;
      listen       [::]:80;
      server_name  _;
      root         /usr/share/nginx/html;

      location / {
              proxy_pass http://localhost:5000/;
              auth_basic "Restricted Content";
              auth_basic_user_file /etc/nginx/.htpasswd;
      }

      # Load configuration files for the default server block.
      include /etc/nginx/default.d/*.conf;
```

Figure 4. nano print of our proxy server nginx.conf after adding the server location information.

Start the nginx and MLflow Servers

Now you can start your nginx reverse proxy server and run your MLflow remote servers. This part requires you to retrieve some information from the AWS management console.

1. Start our nginx server:

   ```
   sudo service nginx start
   ```

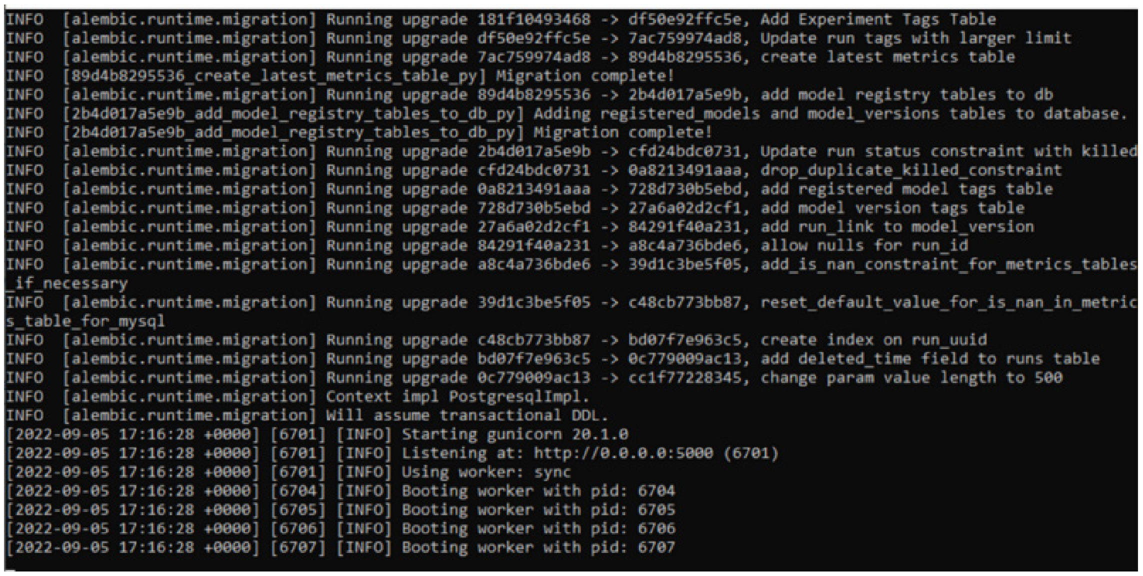
2. Start your MLflow server and configure all components of your object storage (S3) and backend storage (RDS):

   ```
   mlflow server --backend-store-uri
   postgresql://MASTERUSERNAME:YOURPASSWORD@YOUR-DATABASE-
   ENDPOINT:5432/postgres --default-artifact-root s3://BUCKETNAME --
   host 0.0.0.0
   ```

Where can you find this info?

- MASTERUSERNAME — The username that you set for your PostgreSQL RDS DB.
- YOURPASSWORD — The password you set for your PostgreSQL RDS DB.
- YOUR-DATABASE-ENDPOINT — This can be found in your RDS DB information within the AWS management console.
- BUCKETNAME — The name of your S3 bucket.

Once executing this command, you should get a massive printout and information about your worker and pid IDs:



```
INFO  [alembic.runtime.migration] Running upgrade 181f10493468 -> df50e92ffc5e, Add Experiment Tags Table
INFO  [alembic.runtime.migration] Running upgrade df50e92ffc5e -> 7ac759974ad8, Update run tags with larger limit
INFO  [alembic.runtime.migration] Running upgrade 7ac759974ad8 -> 89d4b8295536, create latest metrics table
INFO  [89d4b8295536_create_latest_metrics_table_py] Migration complete!
INFO  [alembic.runtime.migration] Running upgrade 89d4b8295536 -> 2b4d017a5e9b, add model registry tables to db
INFO  [2b4d017a5e9b_add_model_registry_tables_to_db_py] Adding registered_models and model_versions tables to database.
INFO  [2b4d017a5e9b_add_model_registry_tables_to_db_py] Migration complete!
INFO  [alembic.runtime.migration] Running upgrade 2b4d017a5e9b -> cfd24bdc0731, Update run status constraint with killed
INFO  [alembic.runtime.migration] Running upgrade cfd24bdc0731 -> 0a8213491aaa, drop_duplicate_killed_constraint
INFO  [alembic.runtime.migration] Running upgrade 0a8213491aaa -> 728d730b5ebd, add registered model tags table
INFO  [alembic.runtime.migration] Running upgrade 728d730b5ebd -> 27a6a02d2cf1, add model version tags table
INFO  [alembic.runtime.migration] Running upgrade 27a6a02d2cf1 -> 84291f40a231, add run_link to model_version
INFO  [alembic.runtime.migration] Running upgrade 84291f40a231 -> a8c4a736bde6, allow nulls for run_id
INFO  [alembic.runtime.migration] Running upgrade a8c4a736bde6 -> 39d1c3be5f05, add_is_nan_constraint_for_metrics_tables
_if_necessary
INFO  [alembic.runtime.migration] Running upgrade 39d1c3be5f05 -> c48cb773bb87, reset_default_value_for_is_nan_in_metric
s_table_for_mysql
INFO  [alembic.runtime.migration] Running upgrade c48cb773bb87 -> bd07f7e963c5, create index on run_uuid
INFO  [alembic.runtime.migration] Running upgrade bd07f7e963c5 -> 0c779009ac13, add deleted_time field to runs table
INFO  [alembic.runtime.migration] Running upgrade 0c779009ac13 -> cc1f77228345, change param value length to 500
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
[2022-09-05 17:16:28 +0000] [6701] [INFO] Starting gunicorn 20.1.0
[2022-09-05 17:16:28 +0000] [6701] [INFO] Listening at: http://0.0.0.0:5000 (6701)
[2022-09-05 17:16:28 +0000] [6701] [INFO] Using worker: sync
[2022-09-05 17:16:28 +0000] [6704] [INFO] Booting worker with pid: 6704
[2022-09-05 17:16:28 +0000] [6705] [INFO] Booting worker with pid: 6705
[2022-09-05 17:16:28 +0000] [6706] [INFO] Booting worker with pid: 6706
[2022-09-05 17:16:28 +0000] [6707] [INFO] Booting worker with pid: 6707
```

Shut Down Your MLflow Server or Run in the Background

If you would like to shut down your MLflow server at any time, you can use the following command:

```
sudo fuser -k <port>/tcp
```

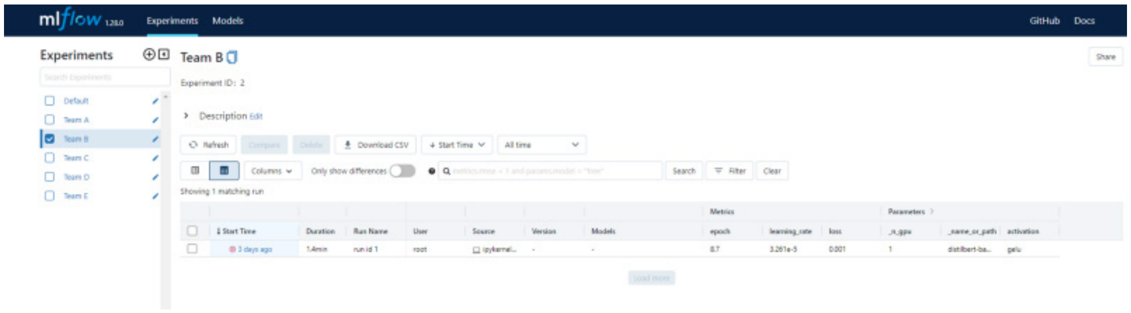
If you want to run it in the background so that you can close the terminal and continue running the server, add `nohup` and & to your MLflow server launch command:

```
nohup mlflow server --backend-store-uri
postgresql://MASTERUSERNAME:YOURPASSWORD@YOUR-DATABASE-
ENDPOINT:5432/postgres --default-artifact-root s3://BUCKETNAME --
host 0.0.0.0 &
```

Access the MLflow UI from Your Browser

Now that you have set up your entire MLflow server infrastructure, you can start interacting with it from the command line, scripts, etc. To access the MLflow UI, you need your EC2 instance's IPV4 Public IP address.

1. Copy and paste your IPV4 Public IP. (This can be found inside your EC2 management console.) Ensure that you add http:// before the IP address.
2. You will be prompted to enter your nginx username and password.
3. Voilà! You can now access and manage your experiments and models from the MLflow UI:



	Start Time	Duration	Run Name	User	Source	Version	Models	epoch	learning_rate	loss	_n_gpu	_name_or_path	activation
	3 days ago	1.4min	run id 1	root	ipykernel...	-	-	8.7	3.261e-5	0.001	1	distilbert-ba...	gelu

Adding MLflow to Your Code and Intel® AI Reference Kit Example

MLflow Tracking Server

The first thing you need to do is to add MLflow credentials and configure the tracking URI inside of your development environment:

```python
import mlflow

# setting up tracking server uri
# this is your remote server EC2 instance IPV4 Public IP
tracking_uri = ''
client = mlflow.tracking.MlflowClient(tracking_uri=tracking_uri)

# nginx credentials
# these are the same credentials that you created during your nginx config step
os.environ['MLFLOW_TRACKING_USERNAME'] = 'testuser'
os.environ['MLFLOW_TRACKING_PASSWORD'] = 'password'
os.environ['MLFLOW_TRACKING_URI'] = tracking_uri
```

mlflow_config.py hosted with ❤ by GitHub view raw

Now let's train a model to classify images using a VGG16 model (please visit the Intel® AI Reference Kit visual quality inspection repository to see the rest of the source code):

```python
# mlflow training stuff
experiment_name = 'Team A'
mlflow.set_experiment(experiment_name) # creation of an new experiment
mlflow_artifact_path='artifacts' # this is the folder name within your S3 bucket where model will
run_id = 'run 1' # name for current mlflow training run


# initializing an mlflow run
with mlflow.start_run(run_name=run_id) as run:

    # Intitalization of DL architecture along with optimizer and loss function
    model = CustomVGG()
    class_weight = torch.tensor(class_weight).type(torch.FloatTensor).to(DEVICE)
    criterion = nn.CrossEntropyLoss(weight=class_weight)
    optimizer = optim.Adam(model.parameters(), lr=LR)

    # intel extension for PyTorch Optimization
    model, optimizer = ipex.optimize(model=model, optimizer=optimizer, dtype=torch.float32)


    # Training section
    start_time = time.time()
    trained_model = train(data_loader=train_loader, model=model, optimizer=optimizer, criterion=cr
        device=DEVICE, target_accuracy=TARGET_TRAINING_ACCURACY, data_aug=data_aug)
    train_time = time.time()-start_time

    mlflow.pytorch.log_model(pytorch_model=trained_model, artifact_path=mlflow_artifact_path)
    mlflow.log_metrics('Training Time', train_time)

    latest_run_id = run.info.run_id

# ending the mlflow run
mlflow.end_run()
```

MLflow Model Registry

After performing ML experimentation and defining a few models worth putting into production, you can register them and stage them for deployment. In the block below, I use the model's run ID and the experiment name to add a model to the MLflow Model Registry. I then move this model into a "Production" stage, load it, and use it as a prediction service for identifying pills with anomalies:

```python
import mlflow.pyfunc

# adding a model to the model registry
model_uri=f"runs:/{current_run_id}/{mlflow_artifact_path}"
name='Team A'
await_registration_for=5
mlflow.register_model(model_uri, name, await_registration_for)

# move a model to archived, staging, and production
client.transition_model_version_stage(name="Team A", version=8, stage="Production")

# Loading latest production version of a model
model_name = 'Team A'
stage = 'Production'
model_uri= f"models:/{model_name}/{stage}"
mlflow_model = mlflow.pytorch.load_model(model_uri)

# predicting with latest production version of a model
predict_localize(mlflow_model, test_loader, DEVICE, thres=HEATMAP_THRESH, n_samples=3, show_heatma
```

mlflow_model_register&serving.py hosted with ❤ by GitHub view raw

The model I deployed into production is working great (**Figure 5**)!

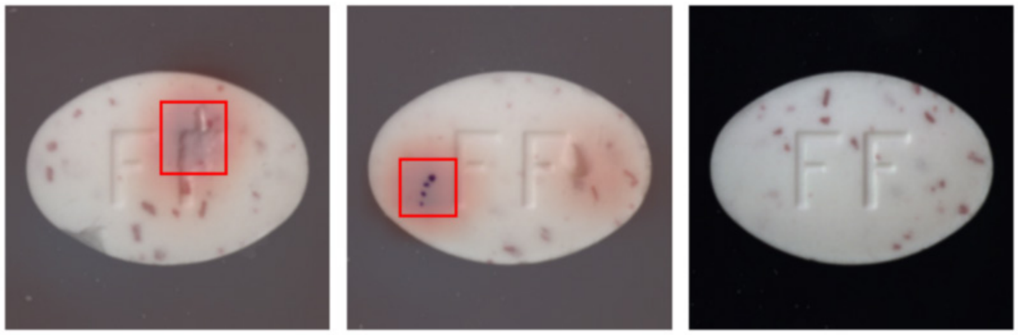


Figure 5. Model deployed to identify anomalous (the first two images) and normal (the image on the right) pills.

Conclusion

This article explored a general cloud-based MLOps environment that leverages the Intel AI Analytics Toolkit to compose an accelerated, end-to-end ML pipeline. This pipeline is a significant first step in ensuring that you always serve the most performant models in your production environment. The intent of this article is not to propose a cookie-cutter environment. As the reader and engineer implementing the pipeline, it is essential to recognize where changes need to be made to consider your compute, security, and general application requirements.

Deep Learning Model Optimizations Made Easy (or at Least Easier)

Sustainable AI, One Model Optimization at a Time

Tony Mongkolsmai, Technical Evangelist, Intel Corporation

Deep learning AI models have grown immensely in the last decade, and along with this rapid growth is an explosion in compute resource requirements. Every larger model requires more computational resources and more movement of bits, both in and out of various memory hierarchies and across systems.

Sustainable AI and Why Deep Learning Optimizations Matter to Me

In January 2020, *Wired* published this piece, AI Can Do Great Things – if It Doesn't Burn the Planet. More recently, *MIT Technology Review* penned an article, These Simple Changes Can Make AI Research More Energy Efficient, about how the Allen Institute for AI, Microsoft, Hugging Face, and several universities partnered to understand how to reduce emissions by running workloads based on when renewable energy is available. I've spent some time thinking about sustainable AI and discussed a few software/hardware alternatives to traditional, deep learning neural networks in a previous article on AI Emerging Technologies to Watch. Although I didn't frame that article around sustainability, all of those technologies have a chance to solve similar problems as deep learning models in specific domains, while significantly reducing the amount of compute power used to arrive at those solutions.

The wonderful thing about optimization of models is that it not only increases performance but also reduces cost and the amount of energy used. By leveraging some of the techniques below, we get the wonderful intersection of solving interesting problems faster, cheaper, and in a more sustainable way.

Common Deep Learning Optimizations

Knowledge Distillation

As the name suggests, the goal of knowledge distillation is to take functionality from one model and move it into another. By leveraging a model that is already a working solution to a problem, we can create a similar, less complex model that can perform the same task. Obviously, the smaller model must perform with similar accuracy to be a successful distillation. In many recent publications on the topic, a teacher/student analogy is used to describe how knowledge distillation learning models work. There are three different ways that the larger teacher model is used to help train the smaller student model: response-, feature-, and relation-based knowledge (**Figure 1**).

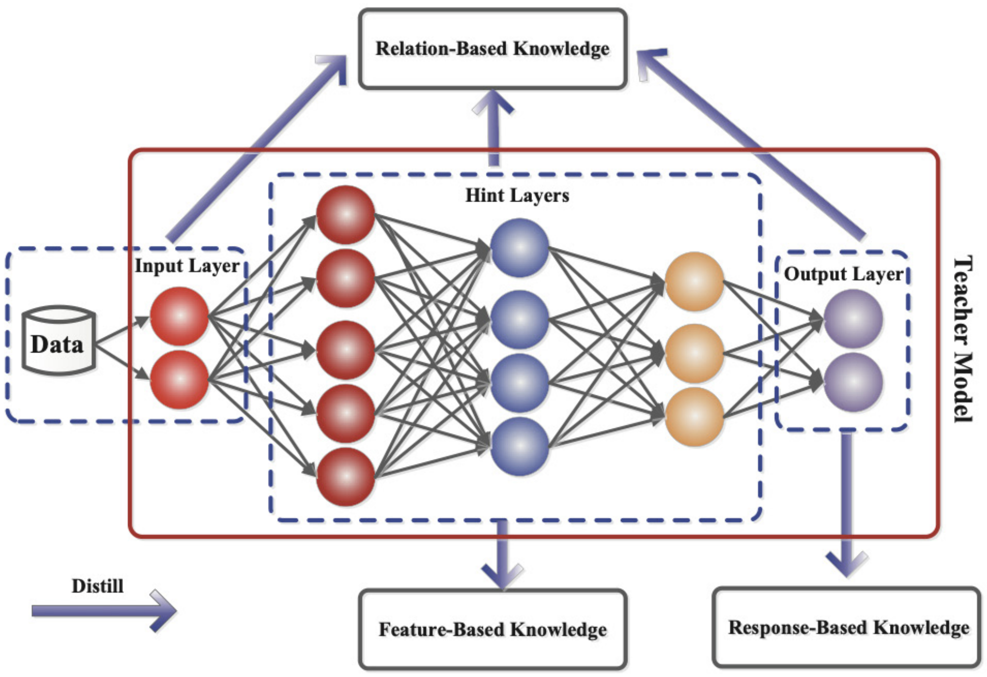


Figure 1. Where does our knowledge come from? (Source: Knowledge Distillation: A Survey)

Response-based knowledge helps train the student model by looking at the output of the teacher model. This is probably the most common-sense way to create a smaller model. We take the larger model output and try to get the same output behavior on our smaller model based on the same or similar input.

Feature-based knowledge helps train the student model by attempting to have the intermediate layers mimic the behavior of the teacher model. This can be difficult because it is not always easy to capture the intermediate feature activations of the model. However, a variety of work has been done in this area to capture the behavior of the intermediate features, which has made such feature-based knowledge distillation possible.

Relation-based knowledge transfer is based on the idea that in the teacher network, the outputs of significantly different parts of the network may work together to help drive the output. This is a little less intuitive to define an algorithm to help train, but the basic idea is to take various groups of nodes, commonly known as feature maps, and train the student nodes to provide similar output as the feature maps in the parent.

Through a combination of these three techniques, it has been shown that some very large models can be migrated to smaller representations. Probably the most well-known of these is DistilBERT, which is able to keep "97% of its language understanding versus BERT while having a 40% smaller model and being 60% faster."

Quantization

Perhaps the most well-known type of deep learning optimization is quantization. Quantization involves taking a model trained using higher precision number formats, like 32- or 64-bit floating point representations, and reproducing the functionality with a neural network that uses lower precision number formats, typically an 8-bit integer (INT8). There are a few approaches to quantization. One can perform quantization after the initial model is trained. The subsequent INT8 model can then be computed by scaling the weights within the original model to generate a new model. This has the benefit of being able to run against existing models that you are trying to optimize after fine-tuning them. Another option is to include quantization techniques as part of the initial training process. This process often creates an INT8 model with greater accuracy versus the post-trained, computed INT8 model method, but at the cost of upfront complexity when creating your model training system.

In both of these cases, the result of using an INT8 representation provides significant savings in terms of model size, which translates into lower memory and compute requirements. Often this can be done with little or no loss of accuracy as documented on the official TensorFlow Quantization Aware Training site.

Making Optimizations Easier

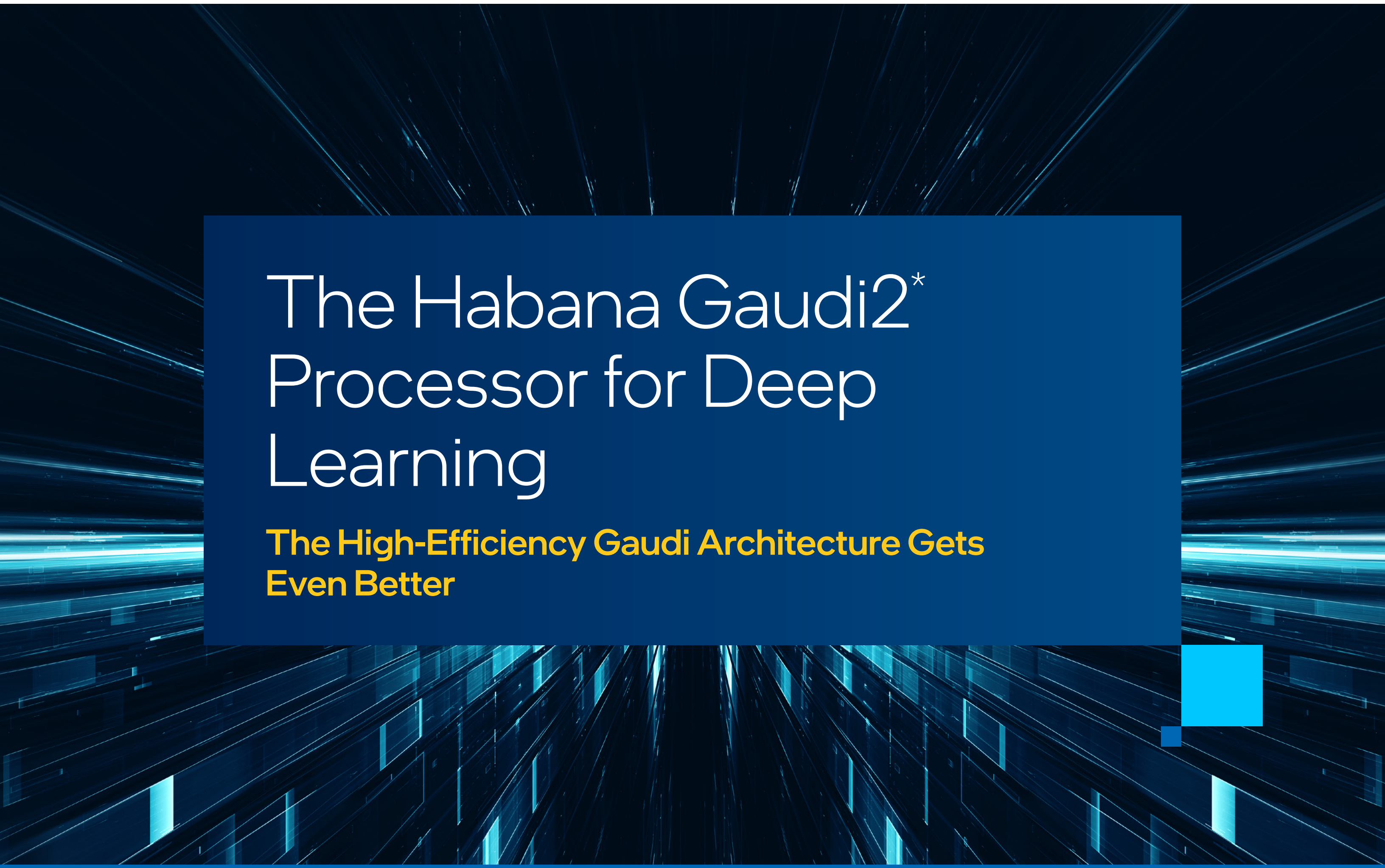
As one might imagine, these simple descriptions of how to create smaller, but still efficient, models require a variety of complex real-world solutions to properly execute them. There are a significant number of research papers devoted to these topics, and a significant amount of research has gone into approaches that can generalize these solutions. Both TensorFlow* and PyTorch* provide some quantization APIs to simplify the quantization process. Keras has a nice TensorFlow example at Knowledge Distillation. For PyTorch, there's a nice Introduction to PyTorch Model Compression Through Teacher-Student Knowledge Distillation, although the example code is a little bit older.

As you can imagine, combining these techniques to generate an optimized model is not always a straightforward task. To help provide a simplified workflow for model optimization, Intel recently released the Intel® Neural Compressor as part of the Intel® AI Analytics Toolkit. This open-source, Python library for CPU and GPU deployment simplifies and automates a significant amount of the setup and process around performing these optimizations. Because it supports TensorFlow, PyTorch, MXNet*, and ONNX, this library should be able to help quickly migrate many larger models into smaller, more optimized models that require fewer hardware resources. *[Editor's note: For more information on how you can leverage this library in PyTorch, check out "**PyTorch Inference Acceleration with Intel Neural Compressor**" in this issue of The Parallel Universe.]*

There are other alternatives as well, depending on your use-case and what frameworks you are already using. For example, if you happen to be using something like OpenVINO™, you can leverage the framework's associated solutions: Neural Network Compression Framework and Post-training Optimization Tool. Obviously, your best option is to use a tool that is tied to whatever framework or SDKs you are already using.

Conclusion

Deep learning models are a vital component of solutions across a large number of industries. As this trend continues, model compression and optimization are critical to reducing the size of models to enable them to run faster and more efficiently than before. These techniques provide a scalar reduction in the amount of energy used but, at their core, the end solution is still a neural network. As a community, it is both an incredible challenge and an imperative that we find more ways to reduce energy usage while simultaneously driving innovation. Looking to the future, I am hopeful to see if and how the paradigms shift to enable us to continue to leverage AI, but with an exponential reduction in compute and energy usage.



The Habana Gaudi2* Processor for Deep Learning

The High-Efficiency Gaudi Architecture Gets Even Better

Chen Levkovich, Principal Software Product Manager, Habana Labs, an Intel company

At Intel Vision 2022 last May, Habana launched its second-generation deep learning processor, Gaudi2* (**Figure 1**), which significantly increases training performance. It builds on the high-efficiency, first-generation Gaudi architecture to deliver up to 40% better price-to-performance on AWS* EC2 DL1 cloud instances and on-premises in the Supermicro Gaudi AI Training Server. It shrinks the process from 16nm to 7nm, increases the number of AI-customized Tensor Processor Cores from 8 to 24, adds FP8 support, and integrates a media compression engine. Gaudi2's in-package memory has tripled to 96 GB of HBM2e at 2.45 TB/s bandwidth. These advances give higher throughput compared to the NVIDIA* A100 80G on popular computer vision and natural language processing models (**Figures 2 and 3**).

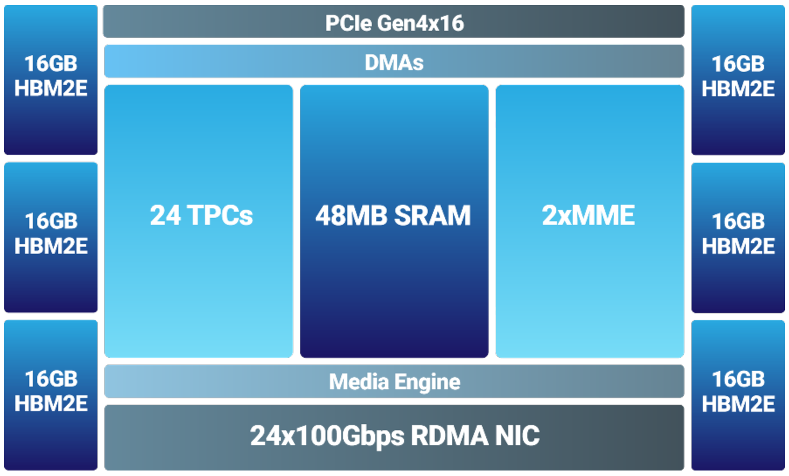


Figure 1. Gaudi2* architecture.

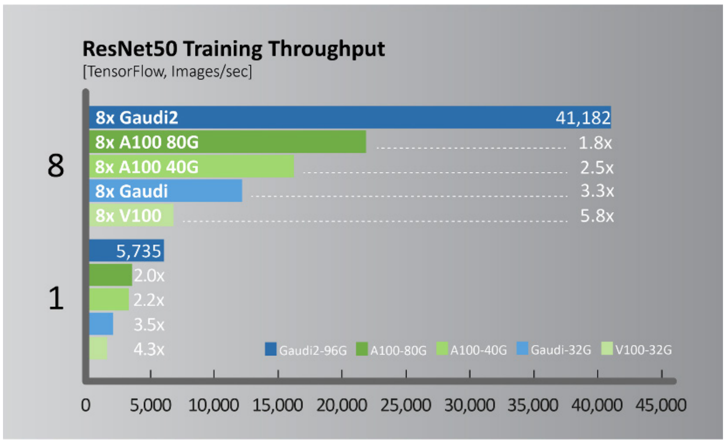
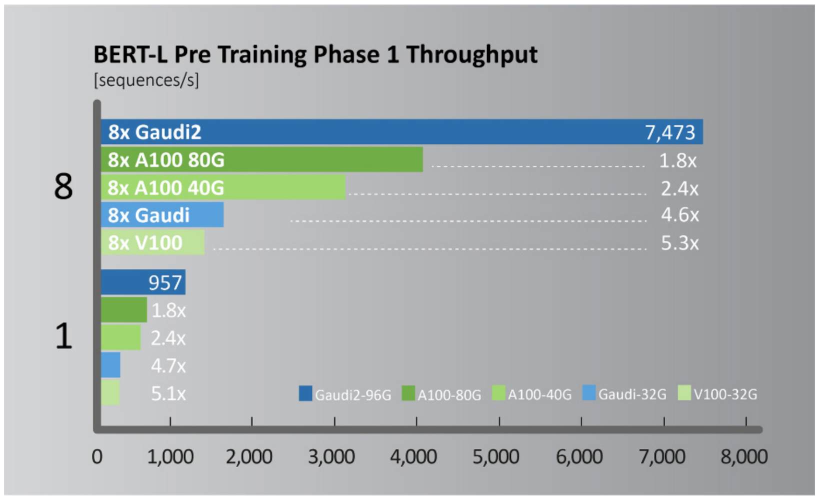


Figure 2. ResNet50 training throughput comparisons. Test configuration: https://github.com/HabanaAI/Model-References/tree/master/TensorFlow/computer_vision/Resnets/resnet_keras. Habana SynapseAI* Container: https://vault.habana.ai/ui/repos/tree/General/gaudi-docker/1.6.0/ubuntu20.04/habanalabs/tensorflow-installer-tf-cpu-2.9.1. Habana Gaudi* Performance: https://developer.habana.ai/resources/habana-training-models. Results may vary. NVIDIA* A100/V100 performance source: https://ngc.nvidia.com/catalog/resources/nvidia:resnet_50_v1_5_for_tensorflow/performance, results published for DGX A100-40G and DGX V100-32G.



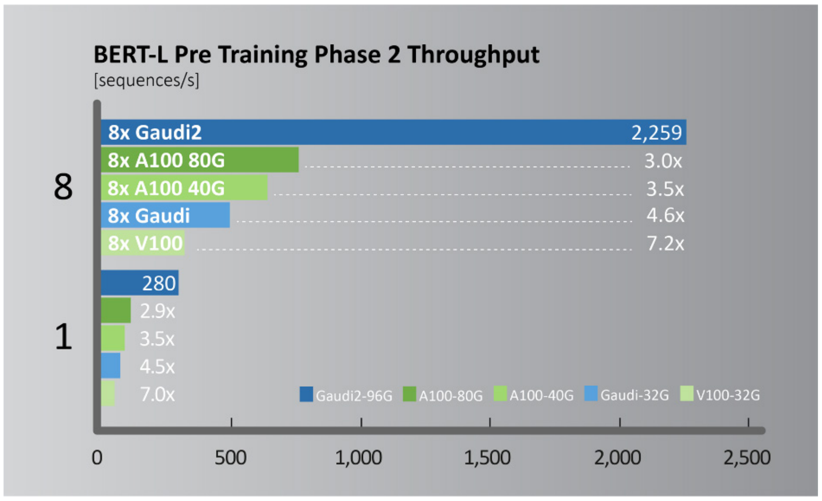


Figure 3. BERT-L pretraining phase 1 and 2 throughput comparisons. Test configuration: A100-80GB: Measured by Habana on Azure* instance Standard_ND96amsr_A100_v4 using single A100-80GB with TF docker 22.03-tf2-py3 from NGC (Phase-1: Seq len=128, BS=312, accu steps=256; Phase-2: seq len=512, BS=40, accu steps=768). A100-40GB: Measured by Habana on DGX-A100 using single A100-40GB with TF docker 22.03-tf2-py3 from NGC (Phase-1: Seq len=128, BS=64, accu steps=1024; Phase-2: seq len=512, BS=16, accu steps=2048). V100-32GB: Measured by Habana on p3dn.24xlarge using single V100-32GB with TF docker 21.12-tf2-py3 from NGC (Phase-1: Seq len=128, BS=64, accu steps=1024; Phase-2: seq len=512, BS=8, accu steps=4096). Gaudi2*: Measured by Habana on Gaudi2-HLS system using single Gaudi2 with SynapseAI TF docker 1.5.0 (Phase-1: Seq len=128, BS=64, accu steps=1024; Phase-2: seq len=512, BS=16, accu steps=2048). Results may vary.

Habana has made it cost-effective and easy for customers to scale out training capacity with the integration of 24 100-gigabit RDMA over Converged Ethernet (RoCE2) ports on every Gaudi2, an increase from ten ports on the first-generation Gaudi. Twenty-one ports on every Gaudi2 are dedicated to connecting to the other seven processors in an all-to-all, non-blocking configuration within the server (**Figure 4**). Three of the ports on every processor are dedicated to scale out, providing 2.4 terabits of networking throughput in the 8-card Gaudi server, the HLS-Gaudi2. To simplify customers' system design, Habana also offers an 8-Gaudi2 baseboard. With the integration of RoCE on chip, customers can easily scale and configure Gaudi2 systems to suit their deep learning cluster requirements, from one to 1,000s of Gaudi2s. With system implementation on industry-standard Ethernet, Gaudi2 enables customers to choose from a wide array of Ethernet switching and networking equipment, enabling added cost-savings. And the on-chip integration of the networking interface controller ports lowers component count and total system cost.

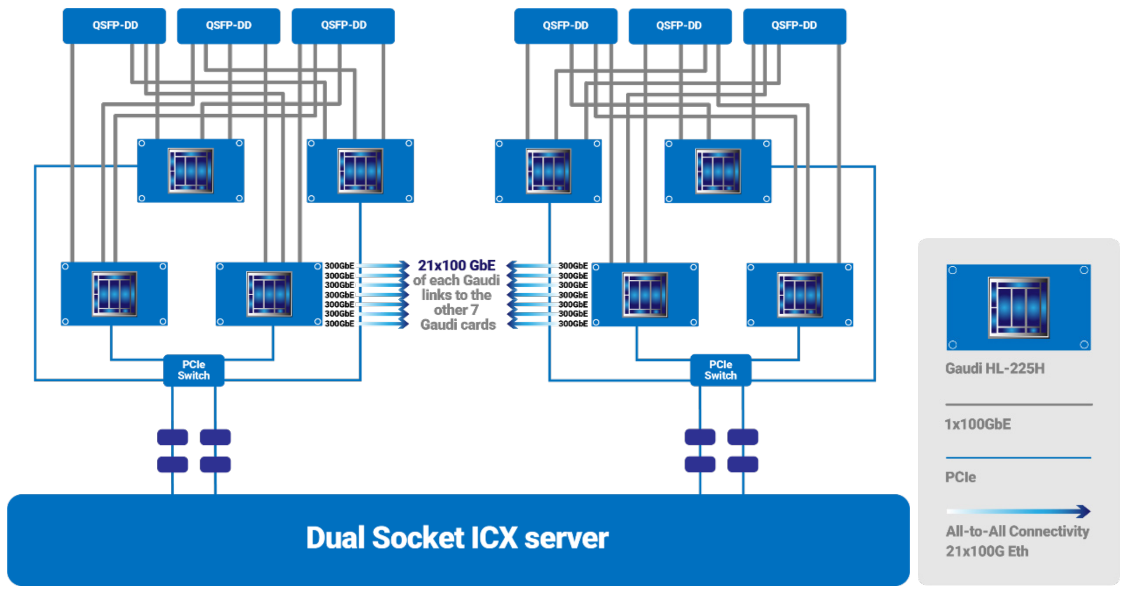


Figure 4. Gaudi2* network configuration.

Gaudi2, like its predecessor, supports developers with the Habana SynapseAI* Software Suite, optimized for deep learning model development and to ease migration from GPU-based models to Gaudi hardware (**Figure 5**). It integrates TensorFlow and PyTorch* frameworks and 50+ computer vision and natural language processing reference models. Developers are provided documentation and tools, how-to content, a community forum, and reference models and model roadmap on the Habana GitHub repository. Getting started with model migration is as easy as adding two lines of code. For expert users programming their own kernels, Habana offers the full toolkit. SynapseAI supports training models on Gaudi2 and inferencing them on any target, including Intel® Xeon® processors, Habana Greco*, or Gaudi2 itself. SynapseAI is also integrated with ecosystem partners, such as Hugging Face with transformer model repositories and tools, Grid.ai Pytorch Lightning*, and cnvrg.io MLOps software.

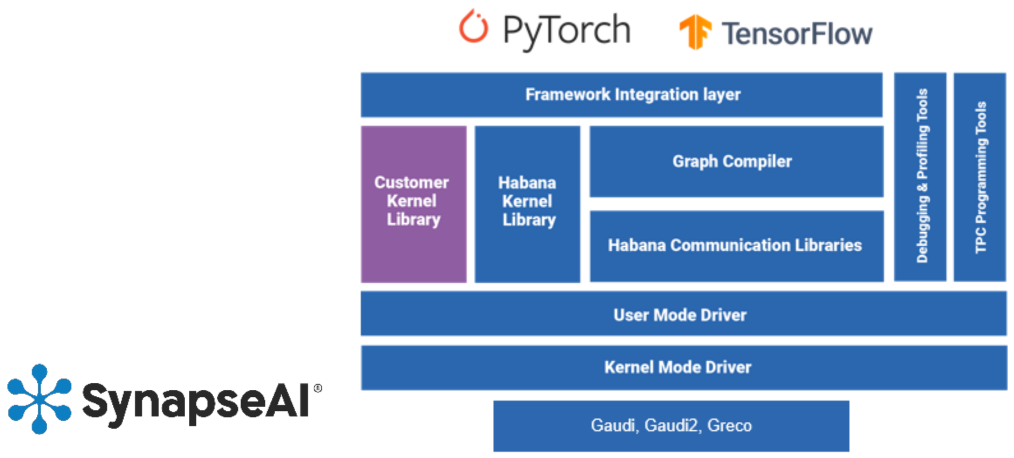


Figure 5. Simplified model building and migration with the Habana SynapseAI* Software Suite.

Ten days after the launch of Gaudi2, the Habana team submitted performance results for June publication in the MLPerf industry benchmark. The Gaudi2 results show dramatic advancements in time-to-train, resulting in Habana's May 2022 MLPerf submission outperforming NVIDIA's A100-80G submission for 8-card server for both the vision (ResNet-50) and language (BERT) models (**Figures 6 and 7**). For ResNet-50, Gaudi2 achieved a significant reduction in time-to-train of 36% vs. NVIDIA's submission for A100-80GB and 45% reduction compared to Dell's submission cited for an A100-40GB 8-accelerator server that was submitted for both ResNet-50 and BERT results. Compared to its first-generation Gaudi, Gaudi2 achieves 3x speed-up in training throughput for ResNet-50 and 4.7x for BERT. These advances can be attributed the numerous hardware and software advances over Gaudi.

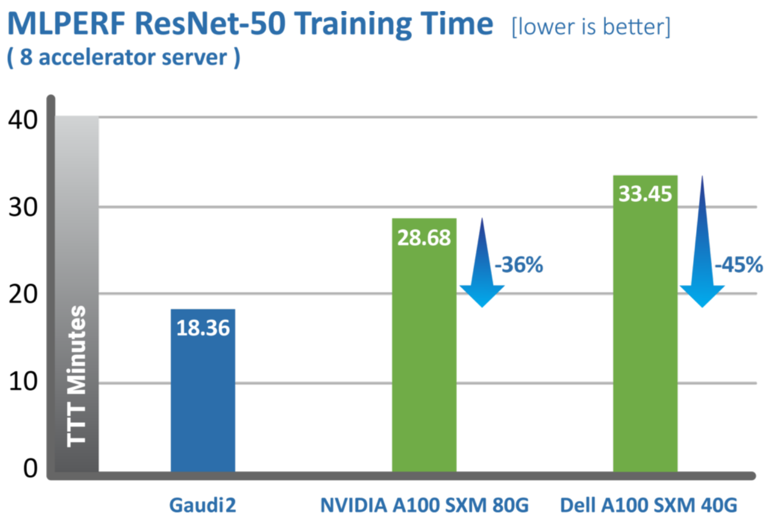


Figure 6. MLPerf results for ResNet-50 training.
Source: https://mlcommons.org/en/training-normal-20 (June 2022).

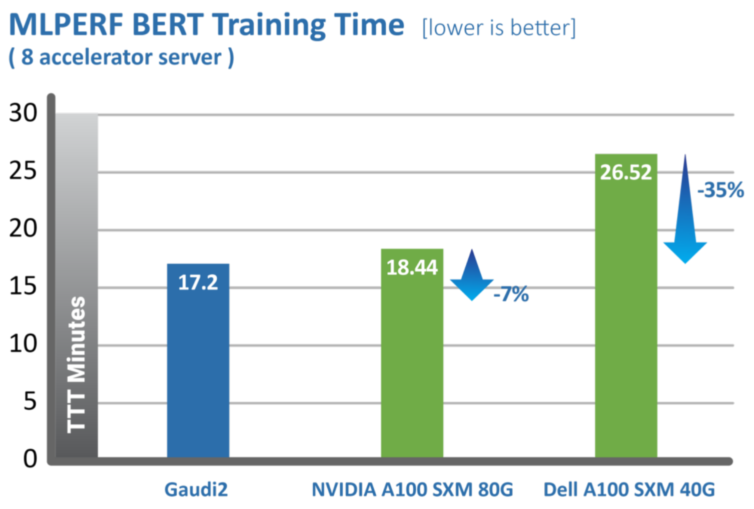


Figure 7. MLPerf results for BERT training.
Source: https://mlcommons.org/en/training-normal-20 (June 2022).

The performance of both generations of Gaudi processors is achieved without special software manipulations that differ from Habana's commercial software stack available to its customers, out of the box. As a result, customers can expect to achieve MLPerf-comparable results in their own Gaudi or Gaudi2 systems using Habana's commercially available software. Both generations of Gaudi were designed at to deliver exceptional deep learning efficiency, so Habana can provide customers with excellent performance while maintaining very competitive pricing.

For more information about Habana Gaudi and Gaudi2, please visit: https://habana.ai/training.

PyTorch* Inference Acceleration with Intel® Neural Compressor

Speed Up AI Inference without Sacrificing Accuracy

Feng Tian, AI Technical Lead, Haihao Shen, AI Architect, Huma Abidi, AI Software Engineering Manager, and Chandan Damannagari, Director of AI Software Product Marketing, Intel Corporation

Intel® Neural Compressor is an open-source Python* library for model compression that reduces the model size and increases the speed of deep learning (DL) inference on CPUs or GPUs (**Figure 1**). It provides unified interfaces across multiple DL frameworks for popular network compression technologies, such as quantization, pruning, and knowledge distillation. This tool supports automatic accuracy-driven tuning strategies to help the user quickly find the best quantized model. It also implements different weight pruning algorithms to generate pruned models using a predefined sparsity goal and supports knowledge distillation from the teacher model to the student model. Intel Neural Compressor provides APIs for a range of frameworks including TensorFlow*, PyTorch*, and MXNet* in addition to ONNX* runtime for greater interoperability across frameworks. We will focus on the benefits of using the tool with a PyTorch model.

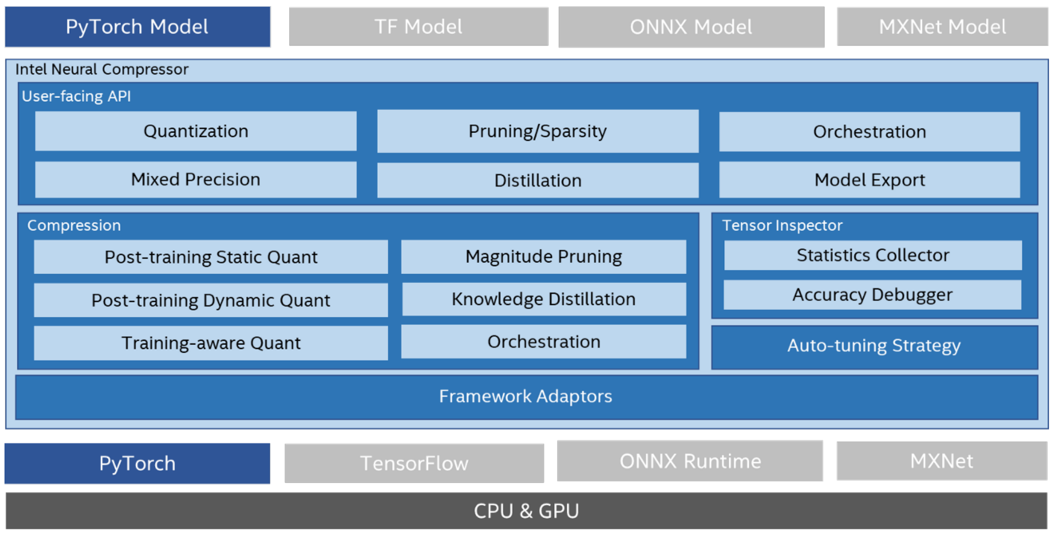


Figure 1. Intel® Neural Compressor.
***Other names and brands may be claimed as the property of others.**

Intel Neural Compressor extends PyTorch quantization by providing advanced recipes for quantization and automatic mixed precision, and accuracy-aware tuning. It takes a PyTorch model as input and yields an optimal model. The quantization capability is built on the standard PyTorch quantization API and makes its own modifications to support fine-grained quantization granularity from the model level to the operator level. This approach gives better accuracy without additional hand-tuning.

It further extends the PyTorch automatic mixed precision feature on 3rd Gen Intel® Xeon® Scalable processors with support for INT8 in addition to BF16 and FP32. It first converts all the quantizable operators from FP32 to INT8, and then converts the remaining FP32 operators to BF16, if BF16 kernels are supported on PyTorch and accelerated by the underlying hardware (**Figure 2**).

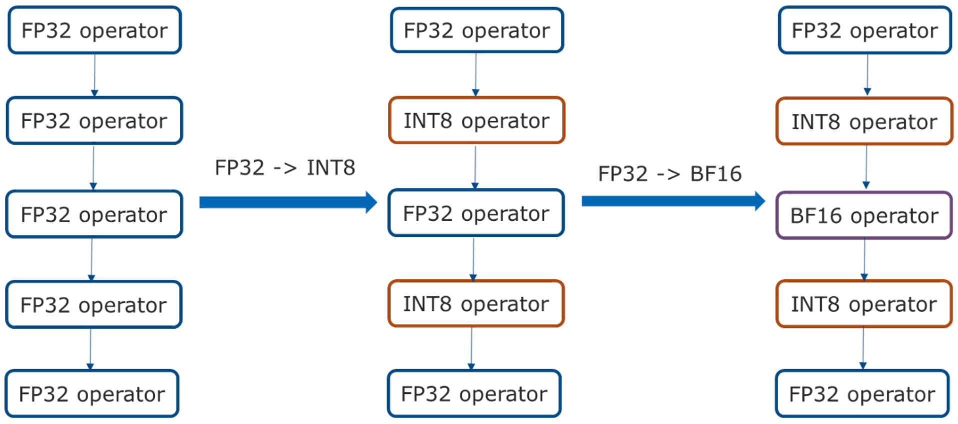


Figure 2. Automatic mixed precision.

Intel Neural Compressor also supports an automatic accuracy-aware tuning mechanism for better quantization productivity. It first queries the framework for the quantization capabilities, such as quantization granularity (per_tensor or per_channel), quantization scheme (symmetric or asymmetric), quantization data type (u8 or s8), and calibration approach (min-max or KL divergence) (**Figure 3**). Then it queries the supported data types for each operator. With these queried capabilities, the tool generates a whole tuning space of different sets of quantization configurations and starts the tuning iterations. For each set of quantization configurations, it performs calibration, quantization, and evaluation. Once the evaluation meets the accuracy goal, the tool terminates the tuning process and produces a quantized model.

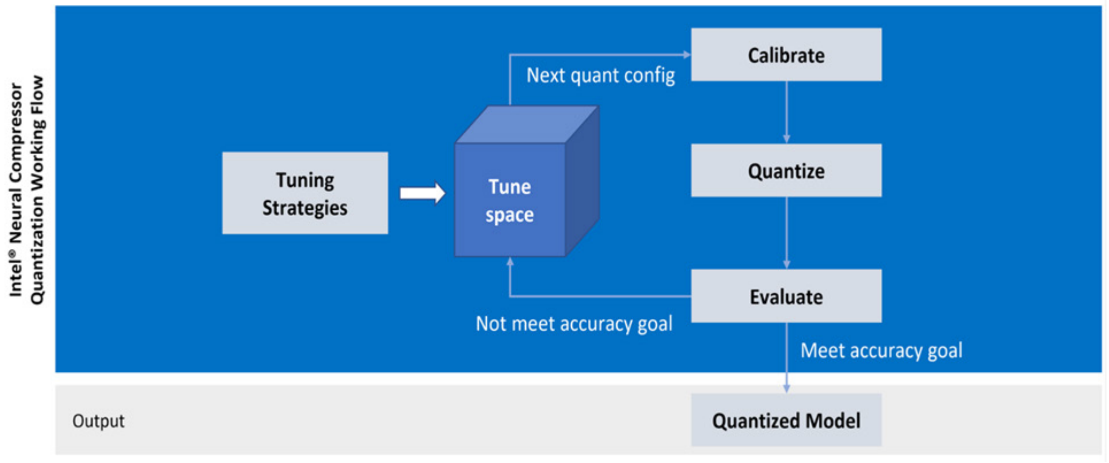


Figure 3. Automatic accuracy-aware tuning.

Pruning is mainly focused on unstructured and structured weight pruning and filter pruning. Unstructured pruning uses a magnitude algorithm to prune weights during training when their magnitude is below a predefined threshold. Structured pruning implements experimental tile-wise sparsity kernels to boost the performance of the sparsity model. Filter pruning implements a gradient-sensitivity algorithm that prunes the head, intermediate layers, and hidden states in the model according to the importance score calculated by the gradient.

Intel Neural Compressor also implements a knowledge distillation algorithm to transfer knowledge from a large "teacher" model to a smaller "student" model without loss of validity (**Figure 4**). The same input is fed to both models, and the student model learns by comparing its results to both the teacher and the ground-truth label.

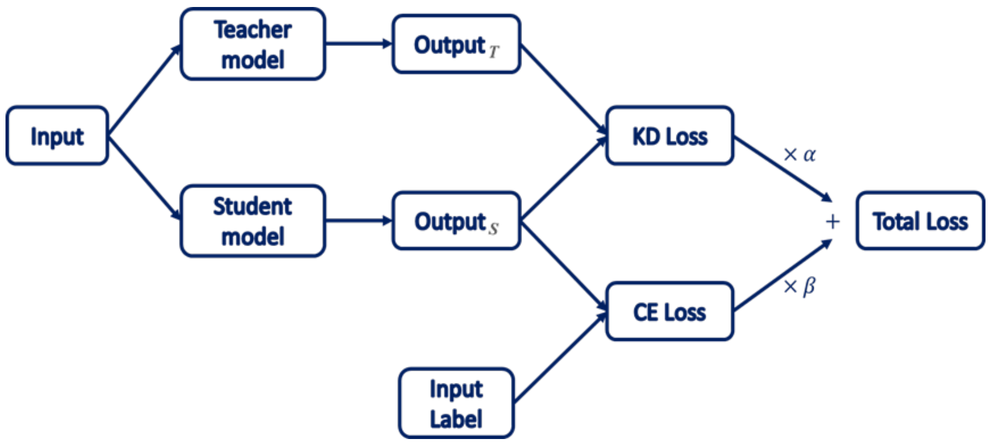


Figure 4. Knowledge distillation algorithm.

The following example shows how to quantize a natural language processing model with Intel Neural Compressor:

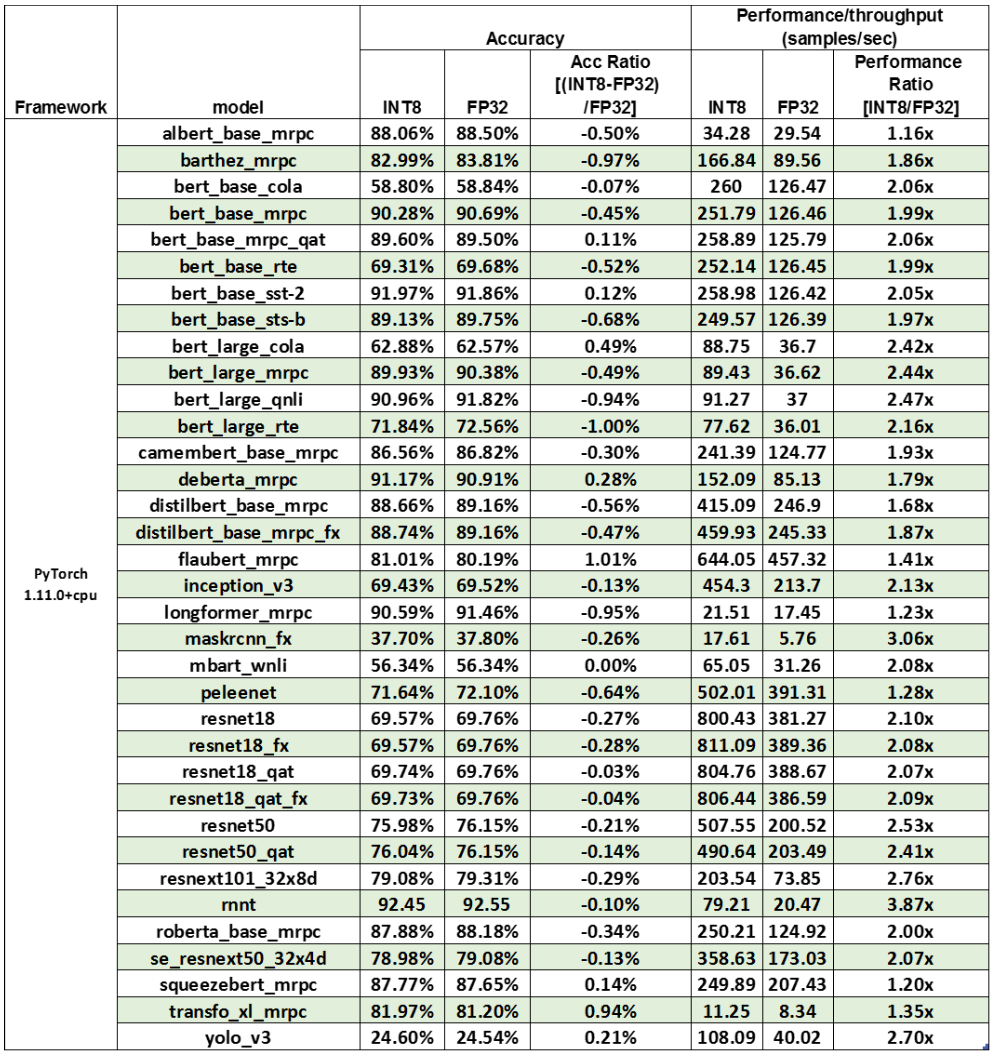
```
# config.yaml
model:
  name: distilbert
  framework: pytorch_fx
tuning:
  accuracy_criterion:
    relative: 0.01

# main.py
import torch
import numpy as np
from transformers import (
    AutoModelForSequenceClassification,
    AutoTokenizer
)

model_name = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
)
```

```
# Calibration dataloader
class CalibDataLoader(object):
    def __init__(self):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.sequence = "Shanghai is a beautiful city!"
        self.encoded_input = self.tokenizer(
            self.sequence,
            return_tensors='pt'
        )
        self.label = 1 # negative sentence: 0; positive sentence: 1
        self.batch_size = 1

    def __iter__(self):
        yield self.encoded_input, self.label

# Evaluation function
def eval_func(model):
    output = model(**calib_dataloader.encoded_input)
    print("Output: ", output.logits.detach().numpy())
    emotion_type = np.argmax(output.logits.detach().numpy())
    return 1 if emotion_type == calib_dataloader.label else 0

# Enable quantization
from neural_compressor.experimental import Quantization
quantizer = Quantization('./config.yaml')
quantizer.model = model
quantizer.calib_dataloader = CalibDataLoader()
quantizer.eval_func = eval_func
q_model = quantizer.fit()
```

Note that the generated mixed-precision model may vary, depending on the capabilities of the low precision kernels and the underlying hardware (e.g., INT8/BF16/FP32 mixed-precision model on 3rd Gen Intel Xeon Scalable processors).

Performance Results

Intel Neural Compressor has validated 400+ examples with a performance speedup geomean of 2.2x on an Intel Xeon Platinum 8380 Processor with minimal accuracy loss (e.g., Table 1). More details for validated models are available here.



Framework	model	Accuracy			Performance/throughput (samples/sec)		
		INT8	FP32	Acc Ratio [(INT8-FP32) /FP32]	INT8	FP32	Performance Ratio [INT8/FP32]
PyTorch 1.11.0+cpu	albert_base_mrpc	88.06%	88.50%	-0.50%	34.28	29.54	1.16x
	barthez_mrpc	82.99%	83.81%	-0.97%	166.84	89.56	1.86x
	bert_base_cola	58.80%	58.84%	-0.07%	260	126.47	2.06x
	bert_base_mrpc	90.28%	90.69%	-0.45%	251.79	126.46	1.99x
	bert_base_mrpc_qat	89.60%	89.50%	0.11%	258.89	125.79	2.06x
	bert_base_rte	69.31%	69.68%	-0.52%	252.14	126.45	1.99x
	bert_base_sst-2	91.97%	91.86%	0.12%	258.98	126.42	2.05x
	bert_base_sts-b	89.13%	89.75%	-0.68%	249.57	126.39	1.97x
	bert_large_cola	62.88%	62.57%	0.49%	88.75	36.7	2.42x
	bert_large_mrpc	89.93%	90.38%	-0.49%	89.43	36.62	2.44x
	bert_large_qnli	90.96%	91.82%	-0.94%	91.27	37	2.47x
	bert_large_rte	71.84%	72.56%	-1.00%	77.62	36.01	2.16x
	camembert_base_mrpc	86.56%	86.82%	-0.30%	241.39	124.77	1.93x
	deberta_mrpc	91.17%	90.91%	0.28%	152.09	85.13	1.79x
	distilbert_base_mrpc	88.66%	89.16%	-0.56%	415.09	246.9	1.68x
	distilbert_base_mrpc_fx	88.74%	89.16%	-0.47%	459.93	245.33	1.87x
	flaubert_mrpc	81.01%	80.19%	1.01%	644.05	457.32	1.41x
	inception_v3	69.43%	69.52%	-0.13%	454.3	213.7	2.13x
	longformer_mrpc	90.59%	91.46%	-0.95%	21.51	17.45	1.23x
	maskrcnn_fx	37.70%	37.80%	-0.26%	17.61	5.76	3.06x
	mbart_wnli	56.34%	56.34%	0.00%	65.05	31.26	2.08x
	peleenet	71.64%	72.10%	-0.64%	502.01	391.31	1.28x
	resnet18	69.57%	69.76%	-0.27%	800.43	381.27	2.10x
	resnet18_fx	69.57%	69.76%	-0.28%	811.09	389.36	2.08x
	resnet18_qat	69.74%	69.76%	-0.03%	804.76	388.67	2.07x
	resnet18_qat_fx	69.73%	69.76%	-0.04%	806.44	386.59	2.09x
	resnet50	75.98%	76.15%	-0.21%	507.55	200.52	2.53x
	resnet50_qat	76.04%	76.15%	-0.14%	490.64	203.49	2.41x
	resnext101_32x8d	79.08%	79.31%	-0.29%	203.54	73.85	2.76x
	rnnt	92.45	92.55	-0.10%	79.21	20.47	3.87x
	roberta_base_mrpc	87.88%	88.18%	-0.34%	250.21	124.92	2.00x
	se_resnext50_32x4d	78.98%	79.08%	-0.13%	358.63	173.03	2.07x
	squeezebert_mrpc	87.77%	87.65%	0.14%	249.89	207.43	1.20x
	transfo_xl_mrpc	81.97%	81.20%	0.94%	11.25	8.34	1.35x
	yolo_v3	24.60%	24.54%	0.21%	108.09	40.02	2.70x

Table 1. Performance results for Intel® Neural Compressor. Configuration: Tested by Intel as of 6/10/2022: Processor: 2S Intel® Xeon® Platinum 8380 CPU @ 2.30GHz, 40-core/80-thread, Turbo Boost on, Hyper-Threading on; Memory: 256GB (16x16GB DDR4 3200MT/s); storage: Intel® SSD *1; NIC: 2x Ethernet Controller 10G X550T; BIOS: SE5C6200.86B.0022. D64.2105220049(ucode:0xd0002b1); OS: Ubuntu 20.04.1 LTS; Kernel: 5.4.0–42-generic; Batch Size: 1; Core per Instance: 4.

The vision of Intel Neural Compressor is to improve productivity and reduce accuracy loss using an auto-tuning mechanism and an easy-to-use API that can be applied to popular neural network compression approaches. We are continuously improving this tool by adding more compression recipes and combining those techniques to produce optimal models. We invite users to try Intel Neural Compressor and provide feedback and contributions via the GitHub repo.

Accelerating PyTorch* with Intel® Extension for PyTorch

An Open-Source Extension to Boost PyTorch Performance

Fan Zhao, Engineering Manager, Jiong Gong, Principal Engineer, and Eikan Wang, AI Software Technical Lead, Intel Corporation

Intel engineers work with the PyTorch* open-source community to improve deep learning (DL) training and inference performance. Intel® Extension for PyTorch is an open-source extension that optimizes DL performance on Intel® processors. Many of the optimizations will eventually be included in future PyTorch mainline releases, but the extension allows PyTorch users to get up-to-date features and optimizations more quickly. In addition to CPUs, Intel Extension for PyTorch will also include support for Intel® GPUs in the near future.

Intel Extension for PyTorch optimizes both imperative mode and graph mode (**Figure 1**). The optimizations cover PyTorch operators, graph, and runtime. Optimized operators and kernels are registered through the PyTorch dispatching mechanism. During execution, Intel Extension for PyTorch overrides a subset of ATen operators with their optimized counterparts and offers an extra set of custom operators and optimizers for popular use-cases. In graph mode, additional graph optimization passes are applied to maximize the performance. Runtime optimizations are encapsulated in the runtime extension module, which provides a couple of PyTorch frontend APIs for users to get finer-grained control of the thread runtime.

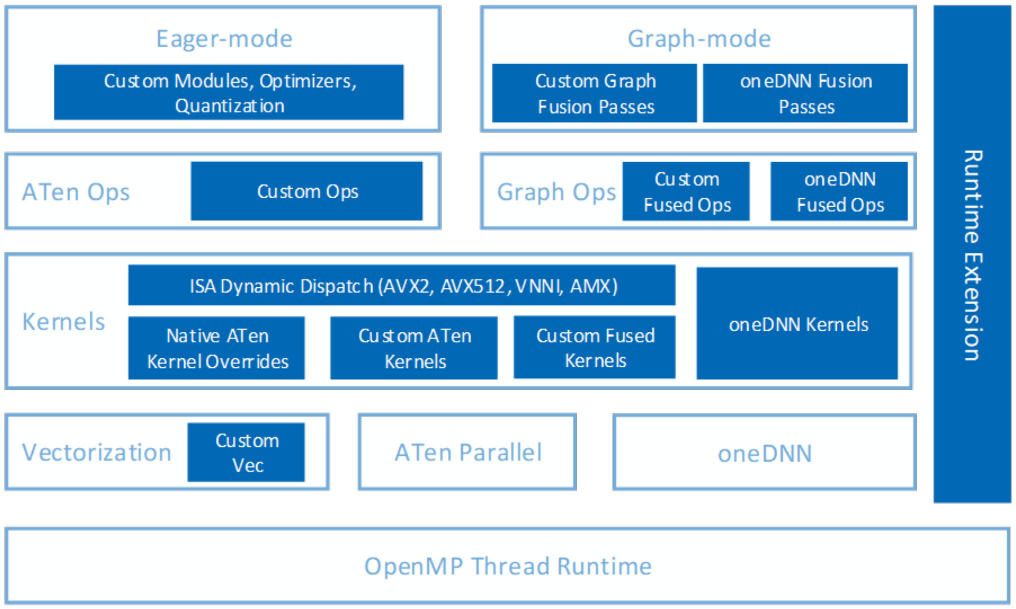


Figure 1. Intel® Extension for PyTorch*.

A Peek at the Optimizations

Memory layout is a fundamental optimization for vision-related operators. Using the right memory format for input tensors can significantly improve the performance of PyTorch models. “Channels last memory format” is generally beneficial for multiple hardware backends:

- (Beta) Channels Last Memory Format in PyTorch
- Efficient PyTorch: Tensor Memory Format Matters
- Understanding Memory Formats

This holds true for Intel processors. With Intel Extension for PyTorch, we recommend using the “channels last” memory format, i.e.:

```
model = model.to(memory_format=torch.channels_last)

input = input.to(memory_format=torch.channels_last)
```

The oneAPI Deep Neural Network Library (oneDNN) introduces blocked memory layout for weights to achieve better vectorization and cache reuse. To avoid runtime conversion, we convert weights to predefined optimal block format prior to the execution of oneDNN operators. This technique is called weight prepacking, and it's enabled for both inference and training when users call the `ipex.optimize` frontend API provided by the extension.

Intel Extension for PyTorch provides several customized operators to accelerate popular topologies, including fused interaction and merged embedding bag, which are used for recommendation models like DLRM, ROIAlign and FrozenBatchNorm for object detection workloads.

Optimizers play an important role in training performance, so we provide highly tuned fused and split optimizers in Intel Extension for PyTorch. We provide the fused kernels for Lamb, Adagrad, and SGD through the ipex.optimize frontend so users won't need to change their model code. The kernels fuse the chain of memory-bound operators on model parameters and their gradients in the weight update step so that the data can reside in cache without being loaded from memory again. We are working to provide more fused optimizers in the upcoming extension releases.

BF16 mixed precision training offers a significant performance boost through accelerated computation, reduced memory bandwidth pressure, and reduced memory consumption. However, weight updates would become too small for accumulation in late stages of training. A common practice is to keep a master copy of weights in FP32, which doubles the memory requirement. The added memory usage burdens workloads that require many weights like recommendation models, so we apply a "split" optimization for BF16 training. We split FP32 parameters into top and bottom halves. The top half is the first 16 bits, which can be viewed exactly as a BF16 number. The bottom half is the last 16 bits, which are kept preserve accuracy. When performing forward and backward propagations, the top half benefits from native BF16 support on Intel CPUs. While performing parameter updates, we concatenate the top and bottom halves to recover the parameters back to FP32, thus avoiding accuracy loss.

Deep learning practitioners have demonstrated the effectiveness of lower numerical precision. Using 16-bit multipliers with 32-bit accumulators improves training and inference performance without compromising accuracy. Even using 8-bit multipliers with 32-bit accumulators is effective for some inference workloads. Lower precision improves performance in two ways: The additional multiply-accumulate throughput boosts compute-bound operations, and the smaller footprint boosts memory bandwidth-bound operations by reducing memory transactions in the memory hierarchy.

Intel introduced native BF16 support in 3rd Gen Intel® Xeon® Scalable processors with BF16→ FP32 fused multiply-add (FMA) and FP32→BF16 conversion Intel® Advanced Vector Extensions-512 (Intel® AVX-512) instructions that double the theoretical compute throughput over FP32 FMAs. BF16 will be further accelerated by the Intel® Advanced Matrix Extensions (Intel® AMX) instruction set in the next generation of Intel Xeon Scalable processors.

Quantization refers to information compression in deep networks by reducing the numerical precision of its weights and/or activations. By converting the parameter information from FP32 to INT8, the model gets smaller and leads to significant savings in memory and compute requirements. Intel introduced the AVX-512 VNNI instruction set extension in 2nd Gen Intel Xeon Scalable processors. It gives faster computation of INT8 data and results in higher throughput. PyTorch offers a few different approaches to quantize models. (See Practical Quantization in PyTorch.)

Graph optimizations like operator fusion maximizes the performance of the underlying kernel implementations by optimizing the overall computation and memory bandwidth. Intel Extension for PyTorch applies operator fusion passes based on the TorchScript IR, powered by the fusion ability in oneDNN and the specialized fused kernels in the extension. The whole optimization is fully transparent to users. Constant-folding is a compile-time graph optimization that replaces operators that have constant inputs with precomputed constant nodes. Convolution+BatchNorm folding for inference gives nonnegligible performance benefits for many models. Users get this benefit from the ipex.optimize frontend API. It's worth noting that we are working with the PyTorch community to get the fusion capability better composed with PyTorch NNC (Neural Network Compiler) to get the best of both.

Examples

Intel Extension for PyTorch can be loaded as a module for Python programs or linked as a library for C++ programs. Users can get all benefits with minimal code changes. A few examples are included below, but more can be found in our tutorials.

BF16 Training

```
...
import torch
...
model = Model()
model = model.to(memory_format=torch.channels_last)
criterion = ...
optimizer = ...
model.train()
#################### code changes ####################
import intel_extension_for_pytorch as ipex
model, optimizer = ipex.optimize(model, optimizer=optimizer, dtype=torch.bfloat16)
######################################################
...
with torch.cpu.amp.autocast():
    # Setting memory_format to torch.channels_last could improve performance
    # with 4D input data.
    data = data.to(memory_format=torch.channels_last)
    optimizer.zero_grad()
    output = model(data)
    loss = ...
    loss.backward()
...
```

BF16 Inference

```
...
import torch
...
model = Model()
model = model.to(memory_format=torch.channels_last)
model.eval()
#################### code changes ####################
import intel_extension_for_pytorch as ipex
model = ipex.optimize(model, dtype=torch.bfloat16)
######################################################
...
with torch.cpu.amp.autocast(),torch.no_grad():
    # Setting memory_format to torch.channels_last could improve performance
    # with 4D input data.
    data = data.to(memory_format=torch.channels_last)
    model = torch.jit.trace(model, data)
    model = torch.jit.freeze(model)
with torch.no_grad():
    output = model(data)
...
```

INT8 Inference – Calibration

```
import os
import torch
model = Model()
model.eval()
data = torch.rand(<shape>)
# Applying torch.fx.experimental.optimization.fuse against model performs
# conv-batchnorm folding for better performance.
import torch.fx.experimental.optimization as optimization
model = optimization.fuse(model, inplace=True)
#################### code changes ####################
import intel_extension_for_pytorch as ipex
conf = ipex.quantization.QuantConf(qscheme=torch.per_tensor_affine)
for d in calibration_data_loader():
    # conf will be updated with observed statistics during calibrating with the dataset
    with ipex.quantization.calibrate(conf):
        model(d)
conf.save('int8_conf.json', default_recipe=True)
with torch.no_grad():
    model = ipex.quantization.convert(model, conf, torch.rand(<shape>))
######################################################
model.save('quantization_model.pt')
```

INT8 Inference – Deployment

```
import torch
#################### code changes ####################
import intel_extension_for_pytorch as ipex
######################################################
model = torch.jit.load('quantization_model.pt')
model.eval()
data = torch.rand(<shape>)
with torch.no_grad():
    model(data)
```

Performance

The potential performance improvements using Intel Extension for PyTorch are shown in **Figure 2** and **Figure 3**. Benchmarking was done on 2.3 GHz Intel Xeon Platinum 8380 processors. (See the measurement details for more information about the hardware and software configuration.) Offline refers to running single-instance inference with large batch using all cores of a socket (**Figure 2**). Realtime refers to running multi-instance, single batch inference with four cores per instance.

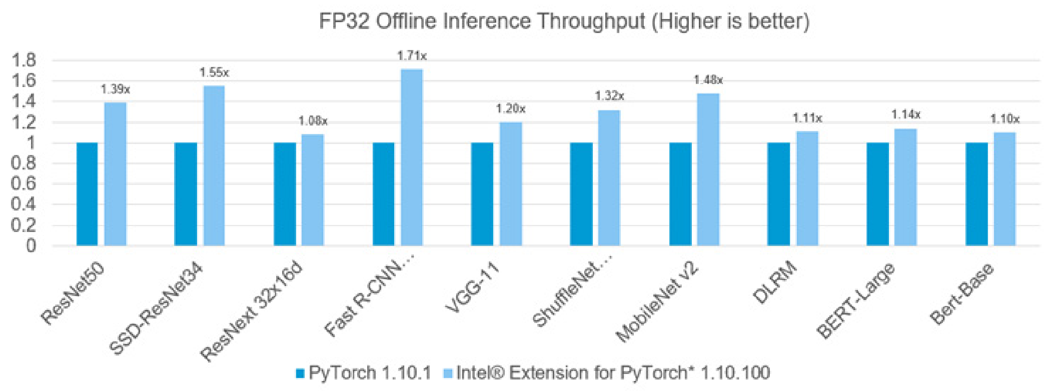


Figure 2. Performance improvement for offline inference using Intel® Extension for PyTorch*.

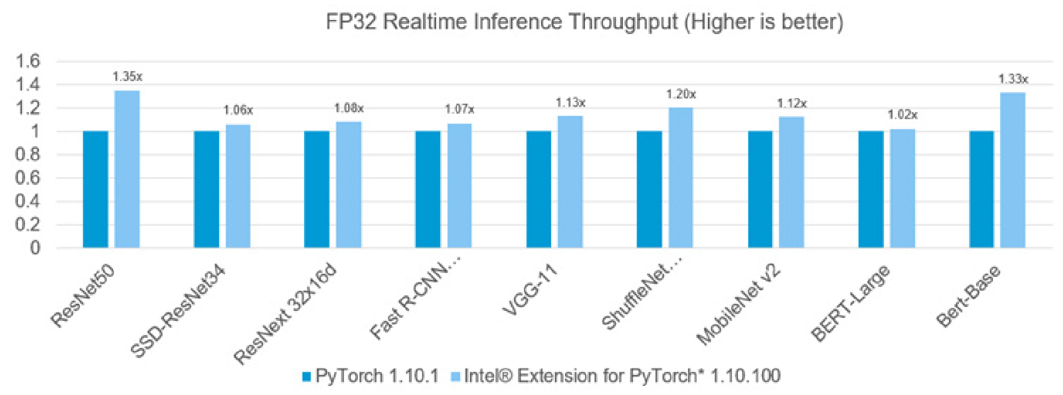


Figure 3. Performance improvement for inference using Intel® Extension for PyTorch*.

Future Work

The intention of Intel Extension for PyTorch is to quickly bring PyTorch users additional performance on Intel processors. We will upstream most of the optimizations to the mainline PyTorch while continuously experimenting with new features and optimizations for the latest Intel hardware. We encourage users to try the open-source project and provide feedback in the GitHub repository.

intel® software

Break Free of Code Boundaries

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

Demo

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

Learn

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

Develop

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

GET STARTED NOW >

Accelerating Python* Today

Getting Ready for the Cambrian Explosion in Accelerator Architectures

James Reinders, oneAPI Evangelist and Editor Emeritus of The Parallel Universe, Intel Corporation

Python* continually surprises many with how versatile and performant it can be. I'm a dyed-in-the-wool C and Fortran programmer, with substantial C++ smarts too, because they allow me to get high performance. Python offers this too, with a convenience that separates it from the previously mentioned languages. So, I'm a Python fan, too.

Python can deliver performance because it has key libraries that are well optimized, and there is support for just-in-time compilation (at run time) for key code that was not precompiled. However, my Python code tends to slow when I reach for larger data sets or more complex algorithms. In this article, we'll review:

1. Why the “Heterogeneous Future” is so important to think about
2. Two key challenges we need to solve in an open solution
3. Parallel execution to use available CPU performance better
4. Using an accelerator to push performance even higher

Step 3 alone offered a 12x speedup, and step 4 offers even more when an accelerator is available. These easy-to-use techniques can be invaluable for a Python programmer when a performance boost is needed. The techniques shared here let us forge ahead without waiting too long for results.

Thinking about the “Heterogeneous Future”

While understanding heterogeneity is not critical if all we want to do is make Python code run faster, it is worth highlighting a substantial shift that is happening in computing now. Computers have become faster every year. At first, clever and more complex architectures drove these performance gains. Then, from about 1989–2006, the increasing clock frequency was the key driver. Suddenly in 2006, raising clock rates no longer made sense and architectural changes were once again needed to increase performance.

Multicore processors offered more performance by increasing the number of (homogeneous) cores in a processor. Unlike rising clock rates, getting additional performance from multicore required software to change to harness the new performance. Herb Sutter’s classic article, “The Free Lunch Is Over,” highlighted the need for concurrency. While necessary, this shift complicated our lives as software developers.

Next, accelerators emerged to augment CPU computations with computations on a specialized device. The most successful of these so far has been the GPU. GPUs were originally introduced to offload graphics processing on its way to a computer’s display. Several programming models emerged to harness this additional computing power, but instead of sending the results to the display, they were sent back to the program running on the CPU. The most successful model so far has been CUDA* for NVIDIA GPUs. Today, the (heterogeneous) processors in a single system are no longer equivalent. Yet, all popular programming languages generally assume a single compute device, so the term “offload” is used when we select part of our code to run on a different compute device.

A few years ago, two industry legends, John Hennessey and David Patterson, announced that we were entering “A New Golden Age for Computer Architecture.” Heterogeneous computing is exploding thanks to the emergence of many ideas for domain-specific processors. Some will succeed and many will fail, but computing is forever changed because it is no longer about doing all computations on a single device.

Two Key Challenges Solved by One Good Solution

While CUDA is a popular option today, it is limited to NVIDIA GPUs. However, we need open solutions to handle the wave of new accelerator architectures that are arriving from multiple vendors. Programs running on heterogeneous platforms need a way to discover what devices are available at runtime. They also need a way to offload computation to those devices.

CUDA ignores device discovery by assuming that only NVIDIA GPUs are available. Python users can choose CuPy to leverage GPUs using CUDA (NVIDIA) or ROCm* (AMD); but, while CuPy is a solid option, it doesn't improve CPU performance or generalize to other vendors or architectures. We would do better with a programming solution that is portable to multiple vendors and can support new hardware innovations. However, before we get too excited about accelerator offload, let's be sure we are getting the most out of the host CPU, because once we understand how to get parallelism and compiled code, we will also be better positioned to use the parallelism in an accelerator as well.

Numba is an open-source, NumPy-aware optimizing (just-in-time) compiler for Python, developed by Anaconda. Under the covers, it uses the LLVM compiler to generate machine code from Python bytecode. Numba can compile a large subset of numerically focused Python, including many NumPy functions. Numba also has support for automatic parallelization of loops, generation of GPU-accelerated code, and creation of universal functions (ufuncs) and C callbacks.

The Numba auto-parallelizer was contributed by Intel. It can be enabled by setting the `parallel=True` option in the `@numba.jit`. The auto-parallelizer analyzes data-parallel code regions in the compiled function and schedules them for parallel execution. There are two types of operations that Numba can automatically parallelize:

1. Implicitly data-parallel regions, such as NumPy array expressions, NumPy ufuncs, NumPy reduction functions
2. Explicitly data-parallel loops that are specified using the numba.prange expression

For example, consider the following simple Python loop:

```
def f1(a,b,c,N):
    for i in range(N):
        c[i] = a[i] + b[i]
```

We can make it explicitly parallel by changing the serial range (`range`) to a parallel range (`prange`) and adding a `njit` directive (`njit` = Numba JIT = compile a parallel version):

```
@njit(parallel=True)
def add(a,b,c,N):
    for i in prange(N):
        c[i] = a[i] + b[i]
```

Run time improved from 24.3 seconds to 1.9 seconds when I ran it, but results can vary depending on the system. To try it, clone the oneAPI-samples repository (`git clone https://github.com/oneapi-src/oneAPI-samples`) and open the `AI-and-Analytics/Jupyter/Numba_DPPY_Essentials_training/Welcome.ipynb` notebook. An easy way to do this is by getting a free account on the Intel® DevCloud for oneAPI.

Using an Accelerator to Push Performance Even Higher

An accelerator can be highly effective when an application has sufficient work to merit the overhead of offloading. The first step is to compile select computations (a kernel) so it can be offloaded. Extending the prior example, we use Numba data-parallel extensions (numba-dpex) to designate an offload kernel. (For more details, see Jupyter notebook training.)

```
@dppy.kernel
def add(a, b, c):
    i = dppy.get_global_id(0)
    c[i] = a[i] + b[i]
```

The kernel code is compiled and parallelized, like it was previously using `@njit` to get ready for running on the CPU, but this time it is ready for offload to a device. It is compiled into an intermediate language (SPIR-V) that the runtime maps to a device when it is submitted for execution. This gives us a vendor-agnostic solution for accelerator offload.

The array arguments to the kernel can be NumPy arrays or Unified Shared Memory (USM) arrays (an array type explicitly placed in Unified Shared Memory) depending on what we feel fits our programming needs best. Our choice will affect how we set up the data and invoke the kernels.

Next, we'll take advantage of a C++ solution for open multivendor, multiarchitecture programming called SYCL*, using the open source data-parallel control (dpctl: C and Python bindings for SYCL). (See GitHub docs and Interfacing SYCL and Python for XPU Programming for more information.) These enable Python programs to access SYCL devices, queues, and memory resources and execute Python array/tensor operations. This avoids reinventing solutions, reduces how much we must learn, and allows a high level of compatibility as well.

Connecting to a device is as simple as:

```
device = dpctl.select_default_device()
print("Using device ...")
device.print_device_info()
```

The default device can be set with an environment variable `SYCL_DEVICE_FILTER` if we want to control device selection without changing this simple program. The `dpctl` library also supports programmatic controls to review and select an available device based on hardware properties.

The kernel can be invoked (offloaded and run) on the device with a couple lines of Python code:

```
with dpctl.device_context(device):
    dpar_add[global_size,dppy.DEFAULT_LOCAL_SIZE](a,b,c)
```

Our use of `device_context` has the runtime do all the necessary data copies (our data was still in standard NumPy arrays) to make it all work. The `dpctl` library also supports the ability to allocate and manage USM memory for devices explicitly. That could be valuable when we get deep into optimization, but the simplicity of letting the runtime handle it for standard NumPy arrays is hard to beat.

Asynchronous vs. Synchronous

Python coding style is easily supported by the synchronous mechanisms shown above. Asynchronous capabilities, and their advantages (reducing or hiding latencies in data movement and kernel invocations), are also available if we're willing to change our Python code a little. See the example code at dpctl gemv example to learn more about asynchronous execution.

What about CuPy?

CuPy is a reimplementation of a large subset of NumPy. The CuPy array library acts as a drop-in replacement to run existing NumPy/SciPy code on NVIDIA CUDA or AMD ROCm platforms. However, the massive programming effort required to reimplement CuPy for new platforms is a considerable barrier to multivendor support from the same Python program, so it does not address the two key challenges mentioned earlier. For device selection, CuPy requires a CUDA-enabled GPU device. For memory, it offers little direct control over memory, though it does automatically perform memory pooling to reduce the number of calls to `cudaMalloc`. When offloading kernels, it offers no control over device selection and will fail if no CUDA-enabled GPU is available. We can get better portability for our application when we use a better solution that addresses the challenges of heterogeneity.

What about scikit-learn?

Python programming in general is well suited for *compute-follows-data*, and using enabled routines is beautifully simple. The `dpctl` library supports a tensor array type that we connect with a specific device. In our program, if we cast our data to a device tensor [e.g., `dpctl.tensor.asarray(data, device="gpu:0")`] it will be associated with and placed on the device. Using a patched version of scikit-learn that recognizes these device tensors, the patched scikit-learn methods that involve such a tensor are automatically computed on the device.

It is an excellent use of dynamic typing in Python to sense where the data is located and direct the computation to be done where the data resides. Our Python code changes very little, the only changes are where we recast our tensors to a device tensor. Based on feedback from users thus far, we expect *compute-follows-data* methods to be the most popular models for Python users.

Open, Multivendor, Multiarchitecture – Learning Together

Python can be an instrument to embrace the power of hardware diversity and harness the impending Cambrian Explosion in accelerators. Numba data-parallel Python combined with `dpctl` and *compute-follows-data* patched scikit-learn are worth considering because they are vendor and architecture agnostic.

While Numba offers great support for NumPy, we can consider what more can be done for SciPy and other Python needs in the future. The fragmentation of array APIs in Python has generated interest in array-API standardization for Python (read a nice summary) because of the desire to share workloads with devices other than the CPU. A standard array API goes a long way in helping efforts like Numba and `dpctl` increase their scope and impact. NumPy and CuPy have embraced array-API, and both `dpctl` and PyTorch* are working to adopt it. As more libraries go in this direction, the task of supporting heterogeneous computing (accelerators of all types) becomes more tractable.

Simply using `dpctl.device_context` is not sufficient in more sophisticated Python codes with multiple threads or asynchronous tasks. (See the GitHub issue.) It is likely better to pursue a *compute-follows-data* policy, at least in more complex threaded Python code. It may become the preferred option over the `device_context` style of programming.

There is a lot of opportunity for us to all contribute and refine ways to accelerate Python together. It's all open source and works quite well today.

Learn More

For learning, there is nothing better than jumping in and trying it out yourself. Here are some suggestions for online resources to help.

For Numba and dpctl, there is a 90-minute video talk covering these concepts: Data Parallel Essentials for Python.

Losing your Loops Fast Numerical Computing with NumPy by Jake VanderPlas (author of the *Python Data Science Handbook*) is a delightfully useful video on how to use NumPy effectively.

The heterogeneous Python capabilities described in this article are all open source, and come prebuilt in the Intel® oneAPI Base and Intel® AI Analytics toolkits. A SYCL-enabled NumPy is hosted on GitHub. Numba compiler extensions for kernel programming and automatic offload capabilities are also hosted on GitHub. The open source data-parallel controls (dpctl: C and Python bindings for SYCL) has GitHub docs and a paper, Interfacing SYCL and Python for XPU Programming. These enable Python programs to access SYCL devices, queues, memory and execute Python array/tensor operations using SYCL resources.

Exceptions are indeed supported, including asynchronous errors from device code. Async errors will be intercepted once they are rethrown as synchronous exceptions by async error handler functions. This behavior is courtesy of Python extensions generators and the community documentation explains it well in Cython and Pybind11.

intel software

THE PARALLEL UNIVERSE