Choose the Best Accelerated Technology

Generative AI Powered by Intel

Akash Dhamasia – AI Software Solutions Engineer
akash.dhamasia@intel.com
July 22nd 2024

All information provided in this deck is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.
No product or component can be absolutely secure.

Your costs and results may vary.

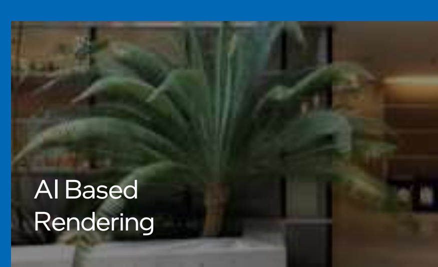
Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

© Intel Corporation. Intel, the Intel logo, Xeon, Core, VTune, OpenVINO, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.
Other names and brands may be claimed as the property of others.

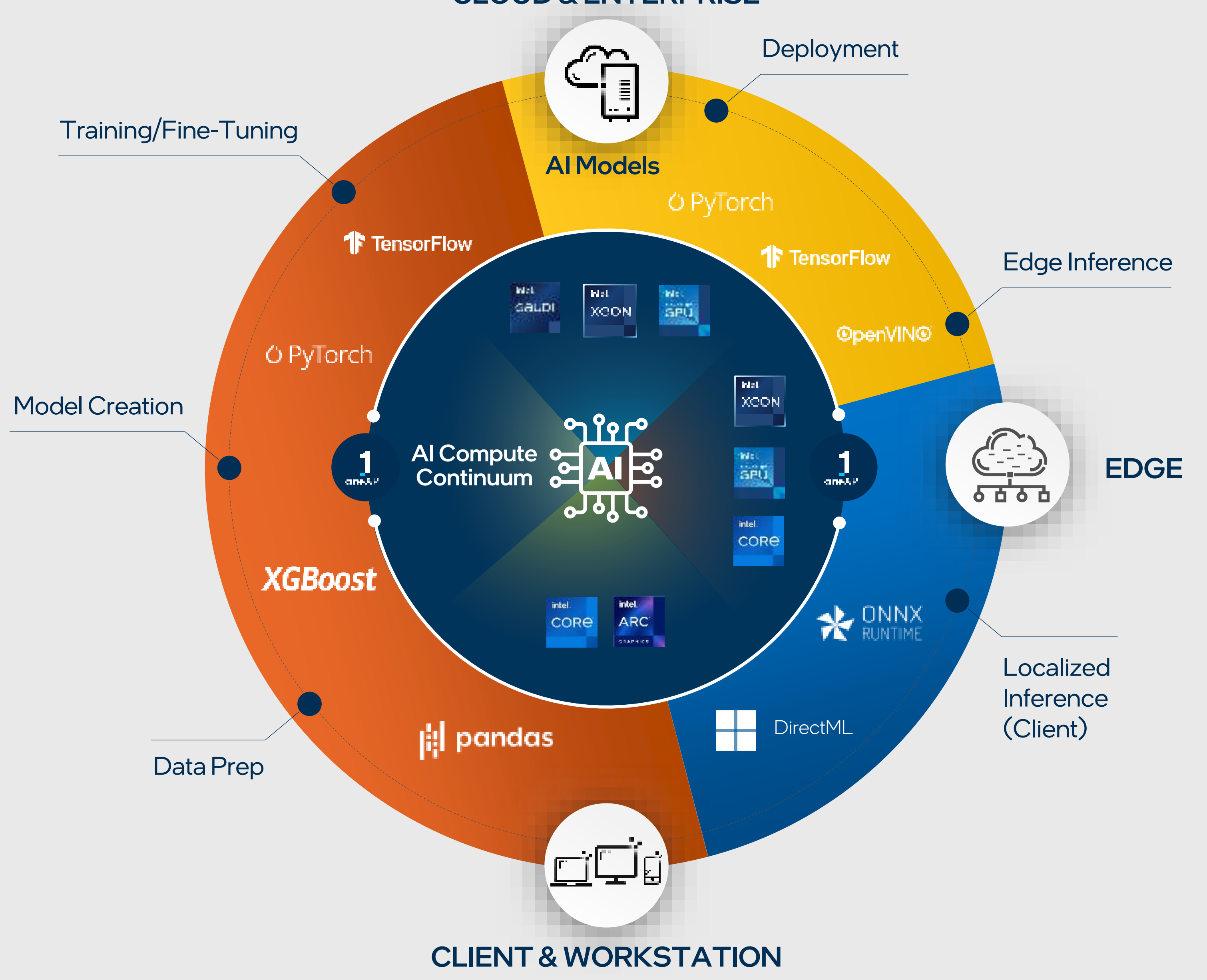
Agenda

- Intel® AI stack
- GenAI @ Intel®
  - Intel® Optimization for PyTorch/TF
    - Intel® Extension of PyTorch
    - Intel® Extension of Tensorflow
  - Intel® Neural Compressor
  - Intel® Extension for Transformers
  - Distributed training @ Intel®
    - DistributedDataParallel (DDP)
    - Horovod
    - FSDP (Fully Shared Data Parallel)
    - DeepSpeed
- Performance
- Conclusion



AI is transforming how we live everyday

AI Continuum



Intel® AI Portfolio

Open Software Environment	oneAPI, OpenVINO, PyTorch, XGBoost, MODIN, TensorFlow, ONNX RUNTIME, deepspeed	
Deep Learning Acceleration	intel GAUDI — Dedicated Deep Learning Training and Inference	
General Acceleration	intel GPU Flex Series — Cloud Gaming, VDI, Media Analytics, Real-Time Dense Video	intel GPU Max Series — Parallel Compute, HPC, AI for HPC
General Purpose	intel Xeon, intel Xeon Max Series — Real-Time, Medium Throughput, Low Latency, and Sparse Inference	intel Xeon, intel Xeon Max Series, intel Xeon W — Medium to Small Scale Training and Fine Tuning
	intel Core, intel Xeon D, intel GPU Flex Series — Edge and Network AI Inference	intel Core, intel Arc Graphics — Inference on Client

Intel AI Software Portfolio

Engineer Data → **Create Models** → **Optimize & Deploy**

Intel-optimized open-source AI tools and frameworks

Data Analytics at Scale: MODIN, SciPy, pandas, NumPy, Apache Spark, Numba

Machine & Deep Learning Frameworks, Optimization and Deployment Tools: dmlc XGBoost, PyTorch, ONNX Runtime, scikit-learn, TensorFlow, DirectML, SigOpt AutoML, intel Neural Compressor

Intel distribution

OpenVINO
Write Once
Deploy Anywhere

oneAPI

- Intel® oneAPI Deep Neural Network Library
- Intel® oneAPI Collective Communications Library
- Intel® oneAPI Math Kernel Library
- Intel® oneAPI Data Analytics Library

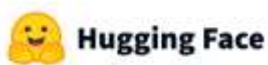
Open, cross-architecture programming model for CPUs, GPUs, and other accelerators

CLOUD & ENTERPRISE | CLIENT & WORKSTATION | EDGE

Intel® Tiber™ Developer Cloud
cloud.intel.com
Try the latest Intel tools and hardware, and access optimized AI Models

Partner

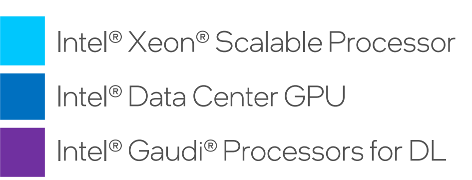
Intel® Tiber™ AI Studio
Full stack ML operating system



Intel optimizations and fine-tuning recipes, optimized inference models, and model serving

Note: components at each layer of the stack are optimized for targeted components at other layers based on expected AI usage models, and not every component is utilized by the solutions in the rightmost column

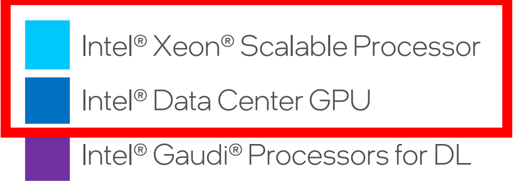
Intel AI Software by Platform



Category	Software	Open Source	Optimizations Upstreamed*	Intel Extension**	Intel Distribution	Intel Tool / Kit
Orchestration	Cnvrg.io	No				Xeon, GPU, Gaudi
Toolkits	BigDL	Yes				Xeon
	OpenVINO	Yes			Xeon, GPU	Xeon, GPU
Optimization	Neural Compressor	Yes				Xeon, GPU
	SigOpt	Yes				Xeon, GPU, Gaudi
DL Frameworks	TensorFlow	Yes	Xeon	Xeon, GPU	Gaudi	
	PyTorch	Yes	Xeon	Xeon, GPU	Gaudi	
	ONNX	Yes	Xeon			
	PDPD	Yes	Xeon			
	DeepSpeed	Yes		GPU		
	OpenFL	Yes	Xeon			
ML Frameworks	XGBoost	Yes	Xeon			
	Scikit-Learn	Yes		Xeon, GPU		
	CatBoost	Yes	Xeon			
	LightGBM	Yes	Xeon			
Data Preprocessing	Modin (for Pandas)	Yes	Xeon		Xeon	
	Intel® Distribution for Python	Yes			Xeon, GPU	
	Spark	Yes	Xeon	Xeon		
AI Compilers	Triton	Yes		GPU		
	OpenXLA	Yes		GPU		

* Intel strives to upstream as many optimizations for as many hardware targets as soon as possible
** Access more Intel optimizations and target hardware support through API-compliant extensions

Intel AI Software by Platform



Category	Software	Open Source	Optimizations Upstreamed*	Intel Extension**	Intel Distribution	Intel Tool / Kit
Orchestration	Cnvrg.io	No				Xeon, GPU, Gaudi
Toolkits	BigDL	Yes				Xeon
	OpenVINO	Yes			Xeon, GPU	Xeon, GPU
Optimization	Neural Compressor	Yes				Xeon, GPU
	SigOpt	Yes				Xeon, GPU, Gaudi
DL Frameworks	TensorFlow	Yes	Xeon	Xeon, GPU	Gaudi	
	PyTorch	Yes	Xeon	Xeon, GPU	Gaudi	
	ONNX	Yes	Xeon			
	PDPD	Yes	Xeon			
	DeepSpeed	Yes		GPU		
	OpenFL	Yes	Xeon			
ML Frameworks	XGBoost	Yes	Xeon			
	Scikit-Learn	Yes		Xeon, GPU		
	CatBoost	Yes	Xeon			
	LightGBM	Yes	Xeon			
Data Preprocessing	Modin (for Pandas)	Yes	Xeon		Xeon	
	Intel® Distribution for Python	Yes			Xeon, GPU	
	Spark	Yes	Xeon	Xeon		
AI Compilers	Triton	Yes		GPU		
	OpenXLA	Yes		GPU		

* Intel strives to upstream as many optimizations for as many hardware targets as soon as possible
** Access more Intel optimizations and target hardware support through API-compliant extensions

Generative AI Powered by Intel

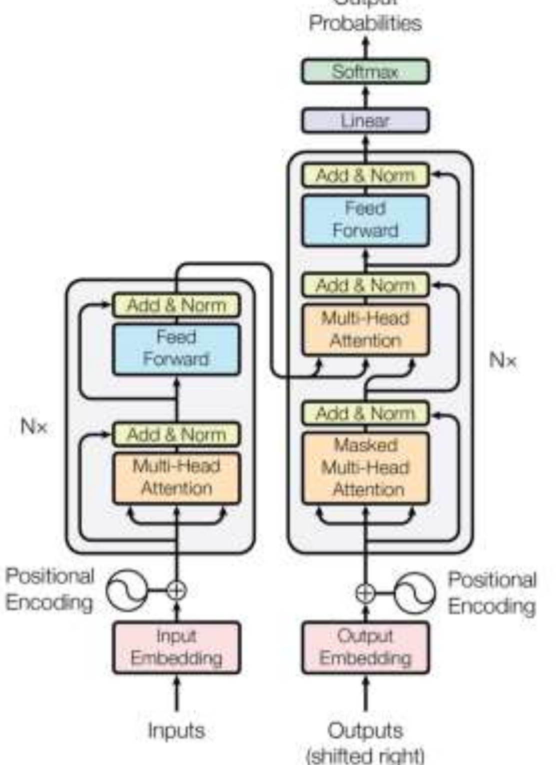
Generative AI

- End of 2022, ChatGPT was released, and the generative AI (genAI) craze started!
- Generative AI is the ability for the AI model to create contents (text, image , music, code …)

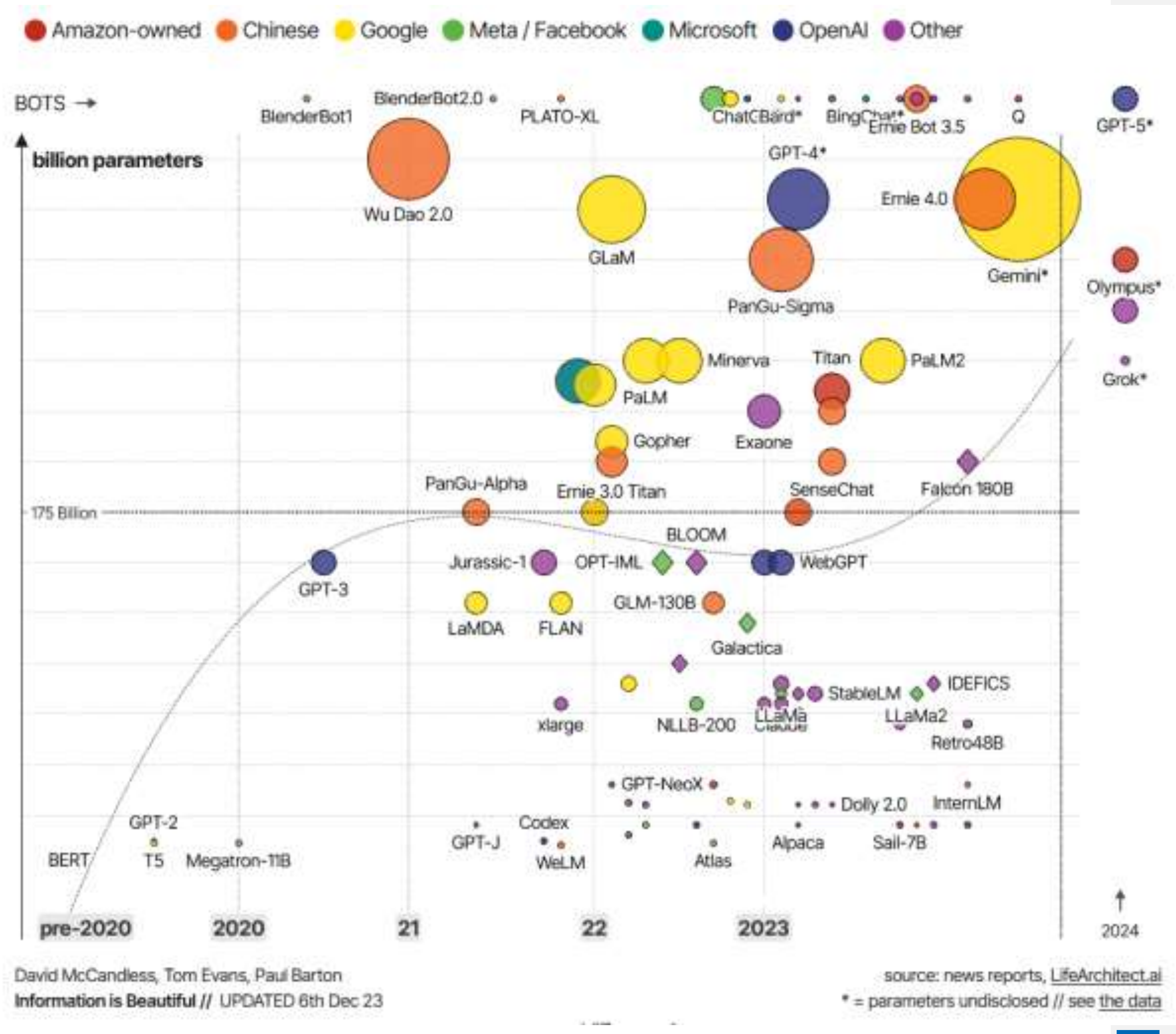
Image fully generated by Stable Diffusion SDXL, a text-to-image AI

Transformers

- Transformer architecture is the base for NLP and genAI (e.g., BERT, LLM, …)
  - Composed of 2 building blocks: encoder and decoder



Model keeps growing: Starting with Bert was 0.3B in 2018



But Recent trend is to scale down: Models are trained on better quality (and smaller) dataset

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Intel's Optimized SW Stack for Generative AI

Frameworks

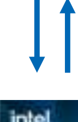
Intel® Extension for PyTorch

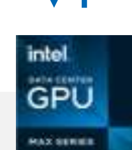
Intel® Extension for Tensorflow

Tools

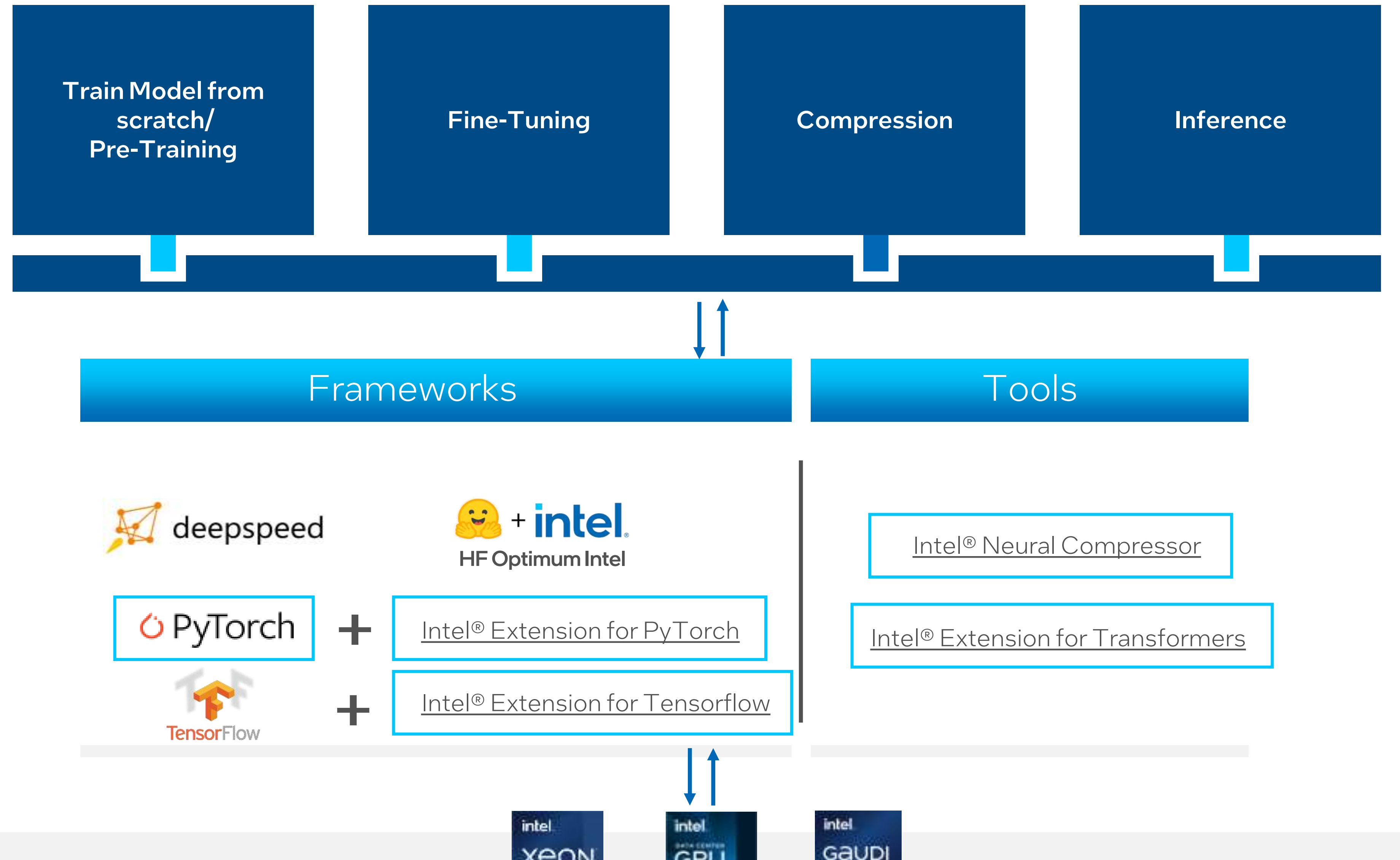
Intel® Neural Compressor

Intel® Extension for Transformers

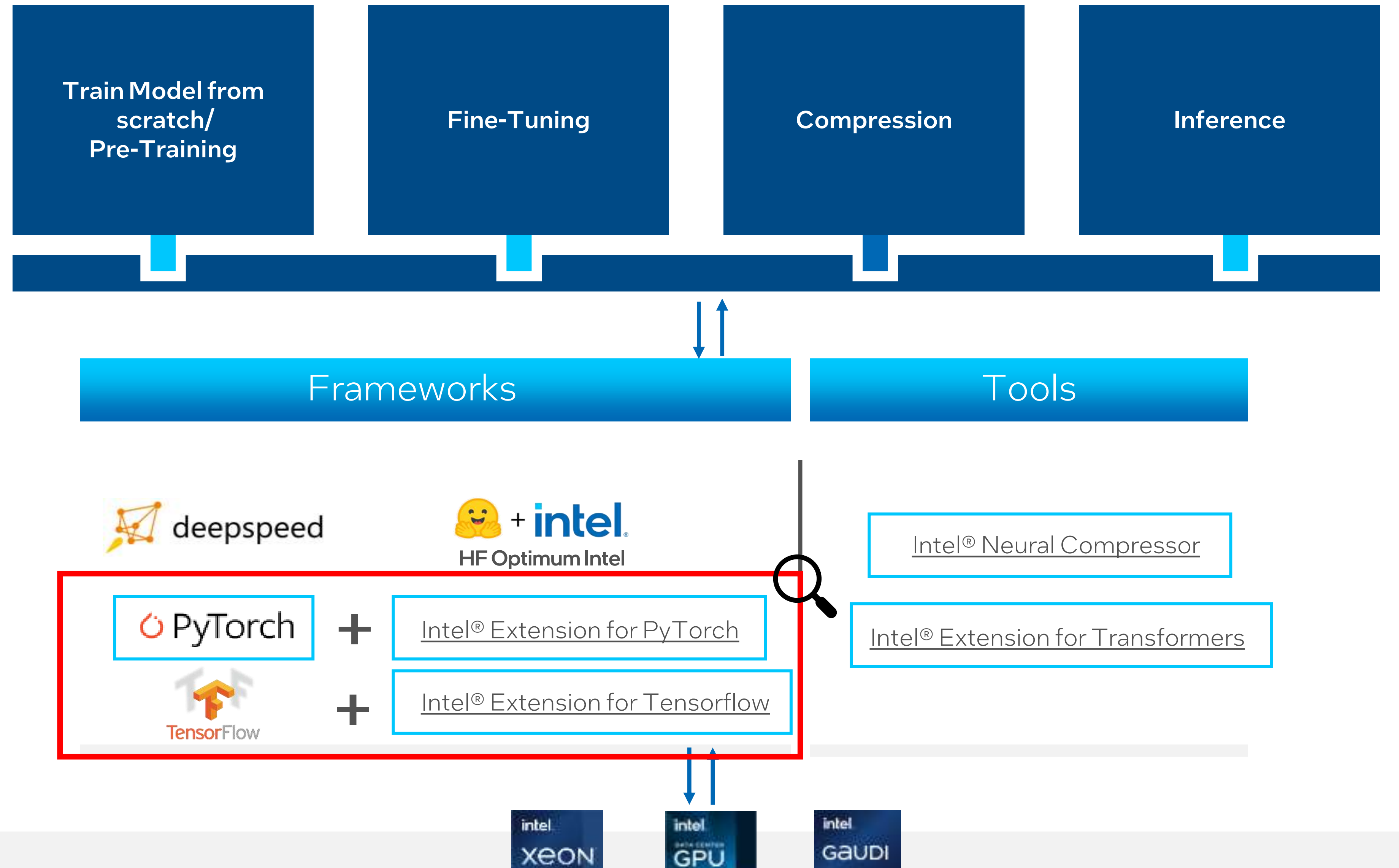




GenAI Deep Learning Funnel Pipeline



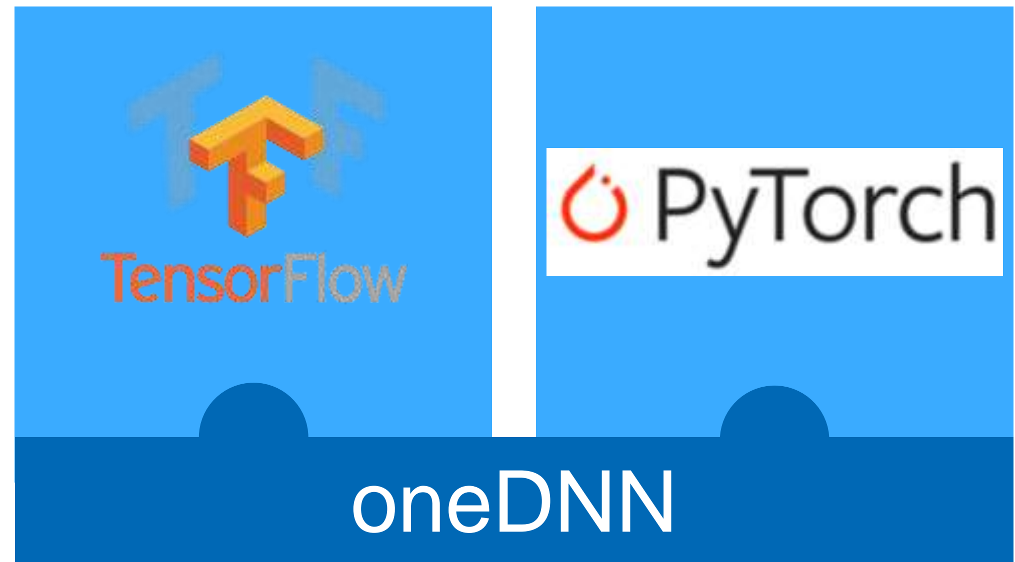
GenAI Deep Learning Funnel Pipeline



Intel®-Optimized Deep Learning Frameworks – Introduction

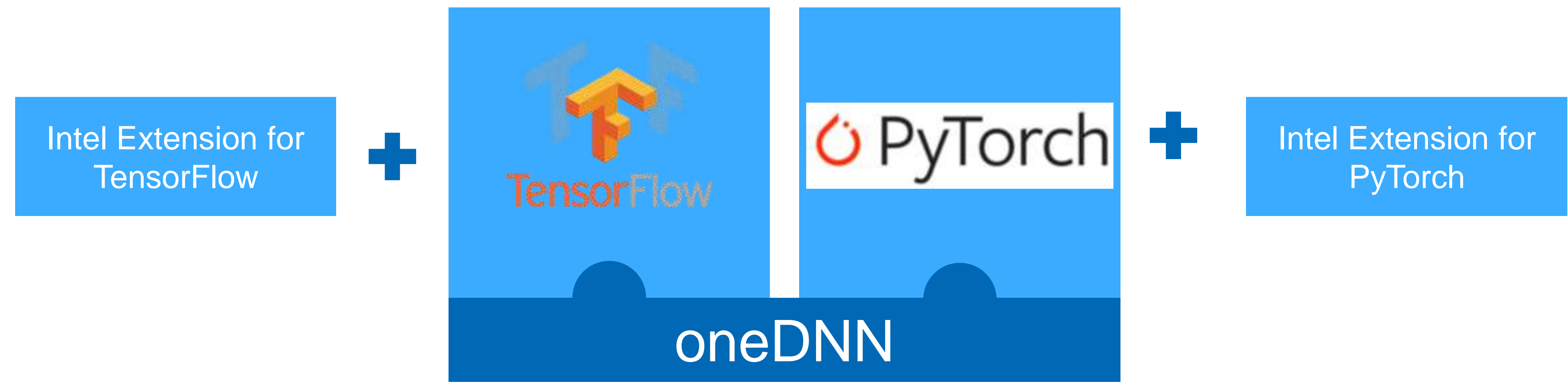
Intel®-Optimized Deep Learning Frameworks

- Intel®-optimized DL frameworks are drop-in replacement,
  - **No front code change for the user**
- Optimizations are up-streamed automatically (TF) or on a regular basis (PyTorch) to stock frameworks

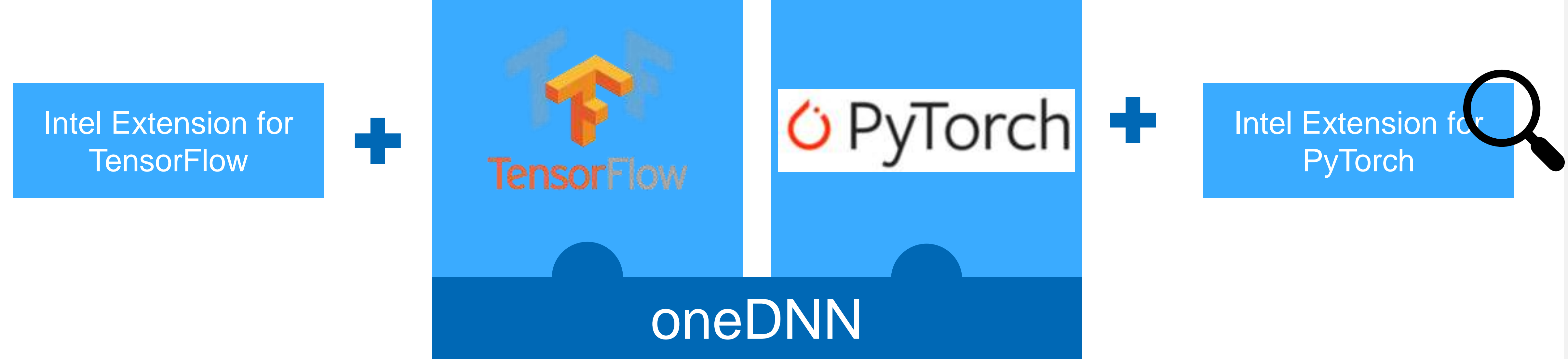


Intel®-Optimized Deep Learning Frameworks

- Intel® Extension for PyTorch and TensorFlow are additional modules for functions not supported in standard frameworks (such as mixed precision and dGPU support).
- As they offer more aggressive optimizations, they offer bigger speed-ups for training and inference.



Intel®-Optimized Deep Learning Frameworks



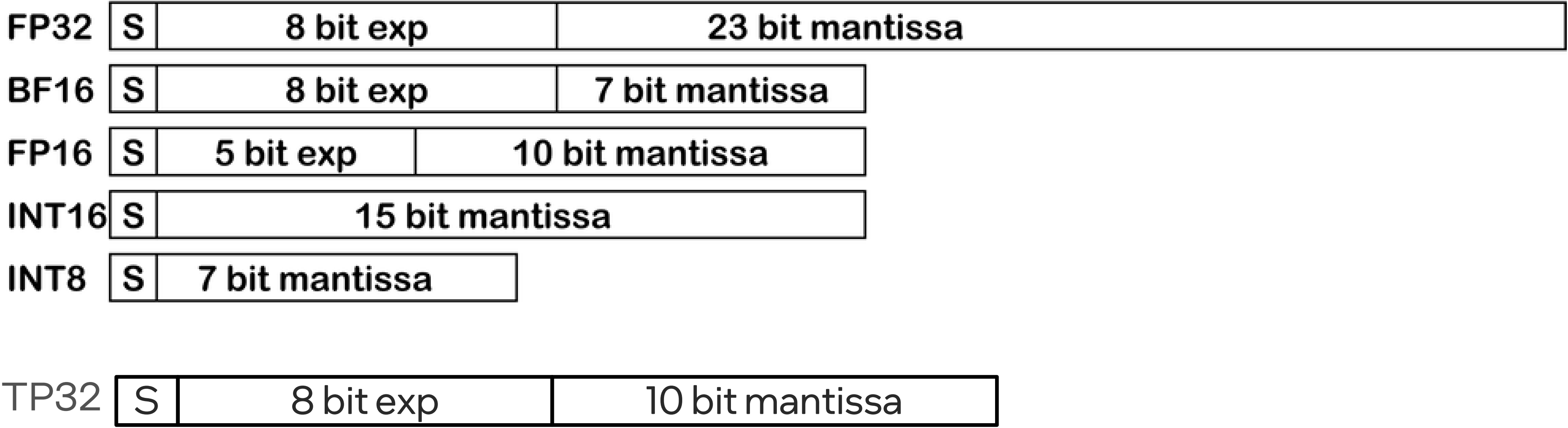
Data Precision

Data Precision

- Data precision:

  Number of bits used to store numerical values in memory

- Commonly found types of precision in Deep Learning:



Ref. https://www.intel.com/content/www/us/en/developer/videos/how-to-use-new-built-in-acceleration-engines.html#gs.wige3o

INT8/BF16 on Artificial intelligence/Machine Learning

- F32 is the default datatypes used in AI/ML for inference, which has a high memory footprint and higher latency.

- Low-precision models are faster in computation. To optimize and support these:
  - HW needs special features/instructions
  - Intel provide those in the form of Intel AMX/Intel XMX.

- SYCL Joint Matrix is the coding abstraction to invoke Intel AMX/Intel XMX, which ensures portability and performance of the code

Ref. https://www.intel.com/content/www/us/en/developer/videos/how-to-use-new-built-in-acceleration-engines.html#gs.wige3o

Introduction to Intel® Advanced Matrix Extension and Intel® Xe Matrix Extensions

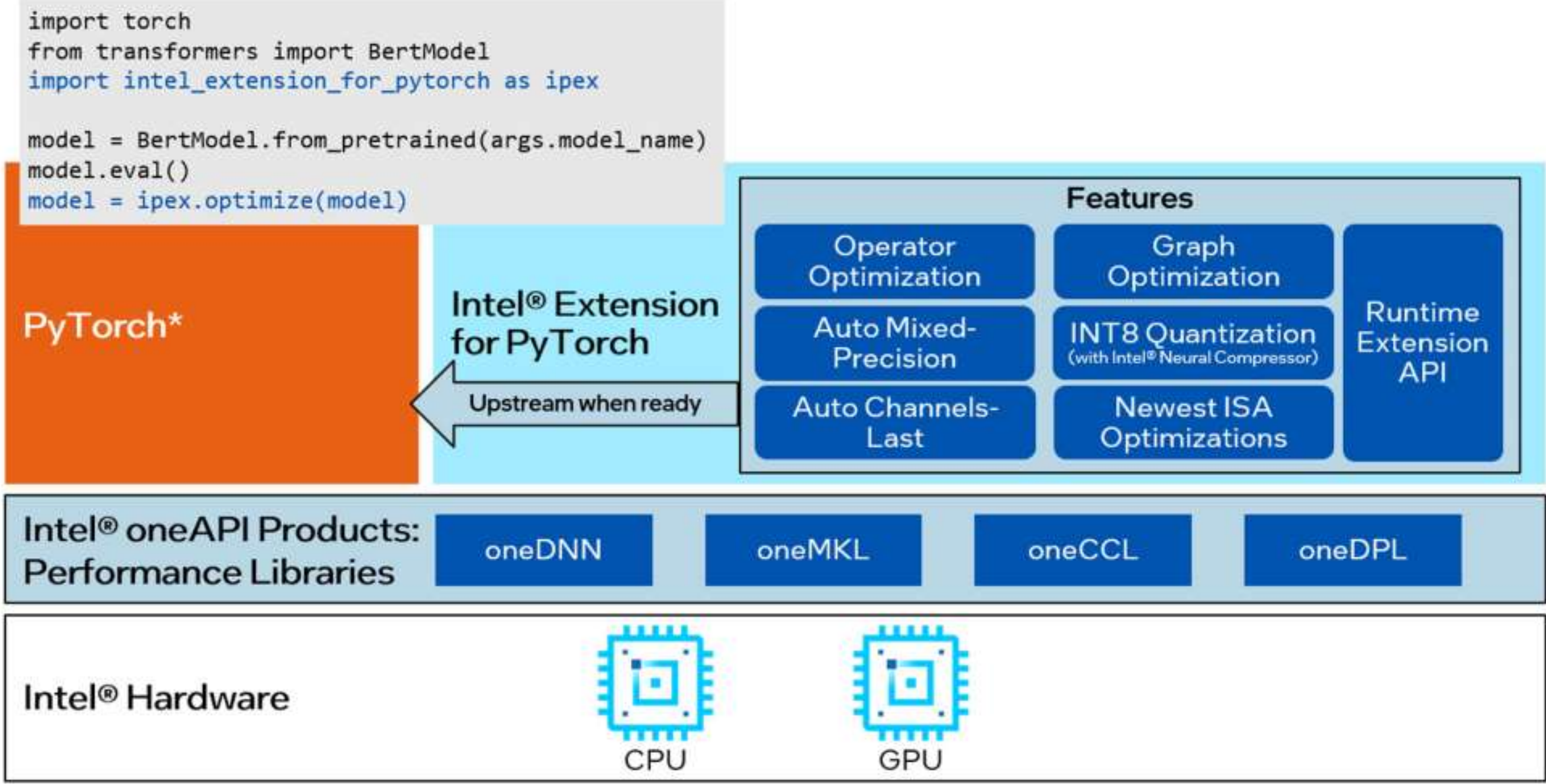
Instruction Set	**Hardware support**	Description
Intel® AMX	Intel® Xeon 4th Generation Scalable CPUs (Formerly code-named Sapphire Rapids)	Intel® Advanced Matrix Extension are extensions to the x86 instruction set architecture (ISA) for microprocessors using 2-dimensional registers called tiles upon which accelerators can perform operations. Supports INT8/BF16
Intel® XMX	Intel® Data Center GPU Max (Formerly code-named Ponte Vecchio) or Intel® Data Center GPU Flex Series	Intel® Xe Matrix Extensions also known as DPAS specializes in executing dot product and accumulate instructions on 2D systolic arrays Supports U8,S8,U4,S4,U2,S2, INT8 FP16, BF16, TF32

Both these Instruction Sets require Intel® oneAPI Base Toolkit 2023.0.0 and above for compilation

Ref. https://www.intel.com/content/www/us/en/developer/videos/how-to-use-new-built-in-acceleration-engines.html#gs.wige3o

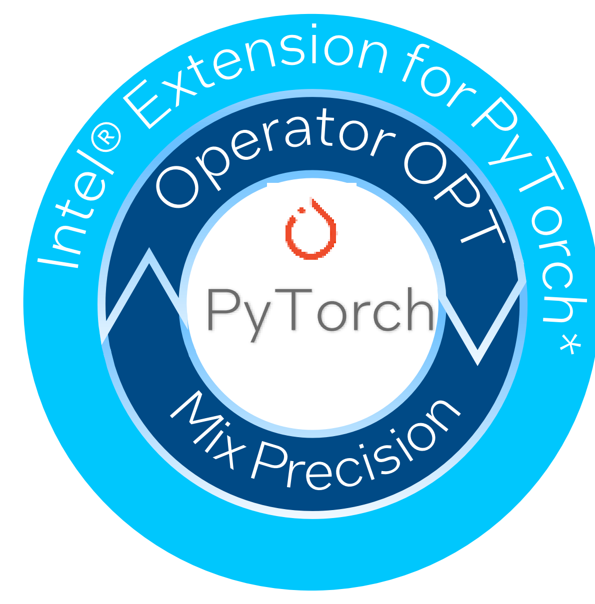
Intel® Extension for PyTorch

PyTorch* Optimizations from Intel



Intel® Extension for PyTorch* (IPEX)

- Buffer the PRs for stock PyTorch
- Provide users with the up-to-date Intel software/hardware features
- Streamline the work to integrate oneDNN
- Unify user experiences on Intel CPU and GPU



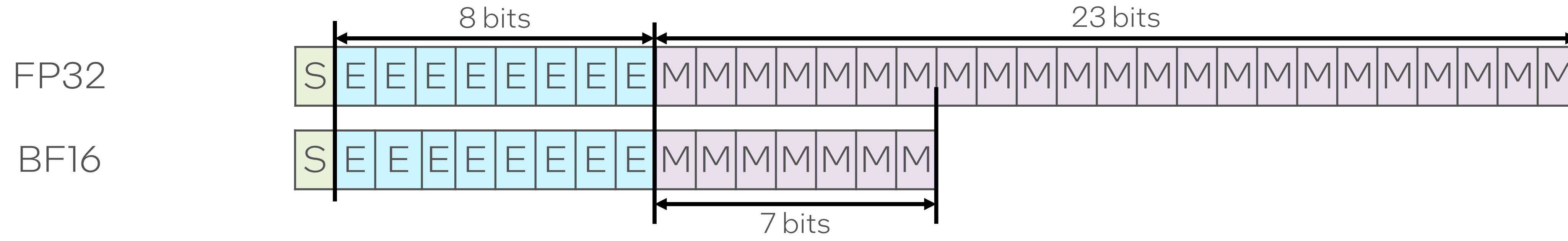
Major Optimization Methodologies

- General performance optimization and Intel new feature enabling in PyTorch upstream
- Additional performance boost and early adoption of aggressive optimizations through Intel® Extension for PyTorch*

Operator Optimization	Graph Optimization	Runtime Extension
▪ Vectorization ▪ Parallelization ▪ Memory Layout ▪ Low Precision	▪ Operator fusion ▪ Constant folding	▪ Thread affinity ▪ Memory allocation

Building and Deploying with BF16

Low-precision Optimization – BF16



BF16 has the same range as FP32 but less precision due to 16 less mantissa bits. Running with 16 bits can give significant performance speedup.

https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf

Inference w/AMX BF16 on Intel® Extension for PyTorch (CPU)

Resnet50

```
import torch
import torchvision.models as models

model = models.resnet50(weights='ResNet50_Weights.DEFAULT')
model.eval()
data = torch.rand(1, 3, 224, 224)

#################### code changes ####################
import intel_extension_for_pytorch as ipex
model = ipex.optimize(model, dtype=torch.bfloat16)
######################################################

with torch.no_grad(), torch.cpu.amp.autocast():
  model = torch.jit.trace(model, torch.rand(1, 3, 224, 224))
  model = torch.jit.freeze(model)

  model(data)
```

BERT

```
import torch
from transformers import BertModel

model = BertModel.from_pretrained("bert-base-uncased")
model.eval()

vocab_size = model.config.vocab_size
batch_size = 1
seq_length = 512
data = torch.randint(vocab_size, size=[batch_size, seq_length])

#################### code changes ####################
import intel_extension_for_pytorch as ipex
model = ipex.optimize(model, dtype=torch.bfloat16)
######################################################

with torch.no_grad(), torch.cpu.amp.autocast():
  d = torch.randint(vocab_size, size=[batch_size, seq_length])
  model = torch.jit.trace(model, (d,), check_trace=False, strict=False)
  model = torch.jit.freeze(model)

  model(data)
```

Training w/AMP on Intel® Extension for PyTorch (GPU)

```
import torch
import torchvision
############# code changes ###############
import intel_extension_for_pytorch as ipex
############# code changes ###############

LR = 0.001
DOWNLOAD = True
DATA = 'datasets/cifar10/'

transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224, 224)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
train_dataset = torchvision.datasets.CIFAR10(
        root=DATA,
        train=True,
        transform=transform,
        download=DOWNLOAD,
)
train_loader = torch.utils.data.DataLoader(
        dataset=train_dataset,
        batch_size=128
)

model = torchvision.models.resnet50()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = LR, momentum=0.9)
model.train()
```

*The .to("xpu") is needed for GPU only
**Use torch.cpu.amp.autocast() for CPU
***Channels last format is automatic

```
##################################### code changes ################################
model = model.to("xpu")
criterion = criterion.to("xpu")
model, optimizer = ipex.optimize(model, optimizer=optimizer, dtype=torch.bfloat16)
##################################### code changes ################################

for batch_idx, (data, target) in enumerate(train_loader):
    optimizer.zero_grad()
    ######################### code changes #########################
    data = data.to("xpu")
    target = target.to("xpu")
    with torch.xpu.amp.autocast(enabled=True, dtype=torch.bfloat16):
    ######################### code changes #########################
        output = model(data)
        loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    print(batch_idx)
torch.save({
     'model_state_dict': model.state_dict(),
     'optimizer_state_dict': optimizer.state_dict(),
     }, 'checkpoint.pth')
```

[Ref: https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/examples.html](https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/examples.html)

Inference w/AMP on Intel® Extension for PyTorch (GPU)

*The .to("xpu") is needed for GPU only
**Use torch.cpu.amp.autocast() for CPU
***Channels last format is automatic

Resnet50

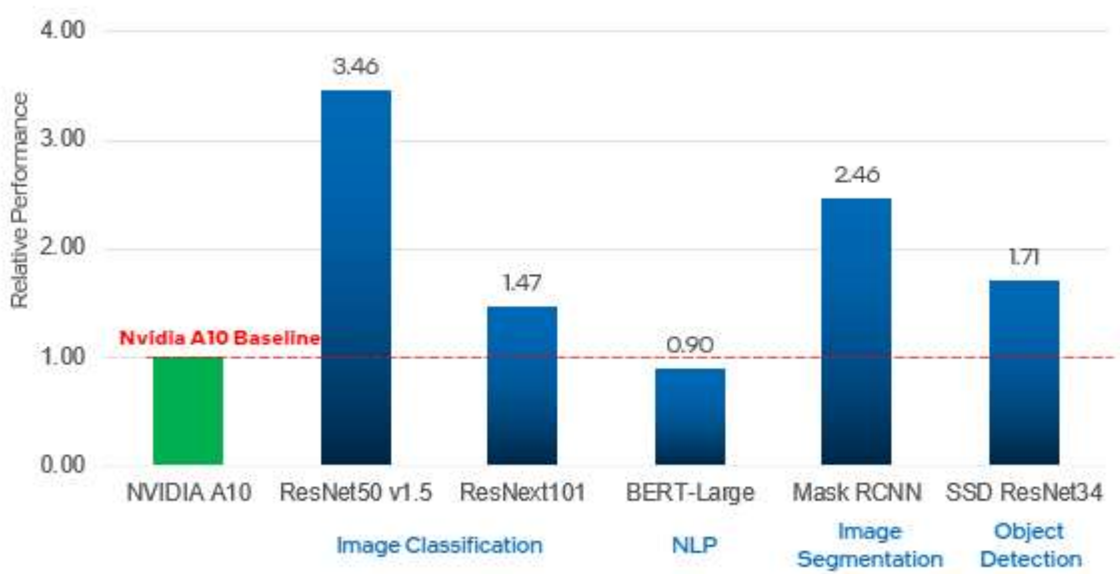
```
import torch
import torchvision.models as models
############# code changes ################
import intel_extension_for_pytorch as ipex
############# code changes ################

model = models.resnet50(pretrained=True)
model.eval()
data = torch.rand(1, 3, 224, 224)

#################### code changes #################
model = model.to("xpu")
data = data.to("xpu")
model = ipex.optimize(model, dtype=torch.bfloat16)
#################### code changes #################

with torch.no_grad():
    d = torch.rand(1, 3, 224, 224)
    ############################# code changes #####################
    d = d.to("xpu")
    with torch.xpu.amp.autocast(enabled=True, dtype=torch.bfloat16):
    ############################# code changes #####################
        model = torch.jit.trace(model, d)
        model = torch.jit.freeze(model)
        model(data)
```

BERT

```
import torch
from transformers import BertModel
############# code changes ###############
import intel_extension_for_pytorch as ipex
############# code changes ###############

model = BertModel.from_pretrained(args.model_name)
model.eval()

vocab_size = model.config.vocab_size
batch_size = 1
seq_length = 512
data = torch.randint(vocab_size, size=[batch_size, seq_length])

#################### code changes #################
model = model.to("xpu")
data = data.to("xpu")
model = ipex.optimize(model, dtype=torch.bfloat16)
#################### code changes #################

with torch.no_grad():
    d = torch.randint(vocab_size, size=[batch_size, seq_length])
    ############################# code changes #####################
    d = d.to("xpu")
    with torch.xpu.amp.autocast(enabled=True, dtype=torch.bfloat16):
    ############################# code changes #####################
        model = torch.jit.trace(model, (d,), strict=False)
        model = torch.jit.freeze(model)

        model(data)
```

Ref: https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/examples.html

Intel Extension for PyTorch Performance



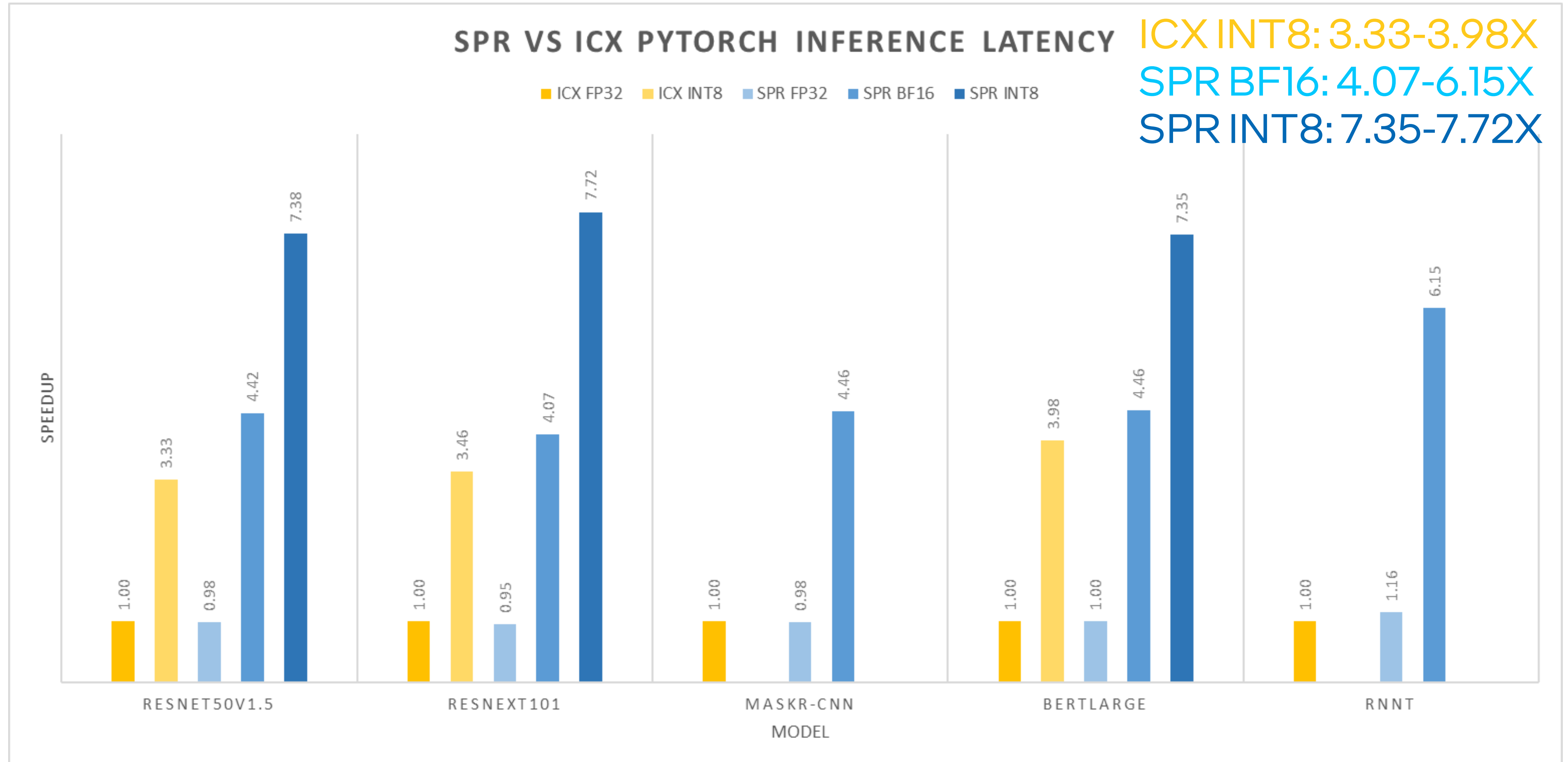
1.8x higher average* BF16/FP16 inference performance vs Nvidia A10 GPU[3]

Benchmark data for the Intel® 4th Gen Xeon Scalable Processors can be found here.

Ref: https://venturebeat.com/ai/unlocking-generative-ai-with-ubiquitous-hardware-and-open-software/

PyTorch Benchmark: SPR vs ICX Inference (Batch Size = 1)

Inference latency speedup: the higher the better



Benchmark data for the Intel® 4th Gen Xeon Scalable Processors can be found here.
Also check Appendix for test configurations.

LLM Optimizations with IPEX (Intel® Extension for PyTorch)

How to apply LLM optimizations with IPEX?

CPU:

`ipex.llm.optimize`(*model, optimizer=None, dtype=torch.float32, inplace=False, device='cpu', quantization_config=None, qconfig_summary_file=None, low_precision_checkpoint=None, sample_inputs=None, deployment_mode=True*)

Apply optimizations at Python frontend to the given transformers model (nn.Module). This API focus on transformers models, especially for generation tasks inference. Well supported model family: Llama, GPT-J, GPT-Neox, OPT, Falcon, Bloom, CodeGen, Baichuan, ChatGLM, GPTBigCode, T5, Mistral, MPT.

GPU:

`ipex.optimize_transformers`(*model, optimizer=None, dtype=torch.float32, inplace=False, device='cpu', quantization_config=None, qconfig_summary_file=None, low_precision_checkpoint=None, sample_inputs=None, deployment_mode=True*)

Apply optimizations at Python frontend to the given transformers model (nn.Module). This API focus on transformers models, especially for generation tasks inference. Well supported model family: Llama, GPT-J, GPT-Neox, OPT, Falcon.

Examples

- Examples:
  CPU - https://github.com/intel/intel-extension-for-pytorch/tree/v2.3.0%2Bcpu-rc0/examples/cpu/inference/python/llm

  GPU - https://github.com/intel/intel-extension-for-pytorch/tree/xpu-main/examples/gpu/inference/python/llm

  A page dedicated to running LLMs with IPEX

- Several ways to set up environment:
  -Docker based
  - Conda Based
  -Pre-built Wheels
  -Build from Source
- Scripts included that set the appropriate environment variables for best performance

```
# Activate environment variables
source ./tools/env_activate.sh
```

Verified Models: Single Instance

CPU:

MODEL FAMILY	MODEL NAME (Huggingface hub)	FP32	BF16	Static quantization INT8	Weight only quantization INT8	Weight only quantization INT4
LLAMA	meta-llama/Llama-2-7b-hf	🟩	🟩	🟩	🟩	🟧
LLAMA	meta-llama/Llama-2-13b-hf	🟩	🟩	🟩	🟩	🟧
LLAMA	meta-llama/Llama-2-70b-hf	🟩	🟩	🟩	🟩	🟧
GPT-J	EleutherAI/gpt-j-6b	🟩	🟩	🟩	🟩	🟩
GPT-NEOX	EleutherAI/gpt-neox-20b	🟩	🟧	🟧	🟩	🟧
DOLLY	databricks/dolly-v2-12b	🟩	🟧	🟧	🟩	🟧
FALCON	tiiuae/falcon-40b	🟩	🟩	🟩	🟩	🟩
OPT	facebook/opt-30b	🟩	🟩	🟩	🟩	🟧
OPT	facebook/opt-1.3b	🟩	🟩	🟩	🟩	🟧
Bloom	bigscience/bloom-1b7	🟩	🟧	🟩	🟩	🟧
CodeGen	Salesforce/codegen-2B-multi	🟩	🟩	🟧	🟩	🟩
Baichuan	baichuan-inc/Baichuan2-7B-Chat	🟩	🟩	🟩	🟩	
Baichuan	baichuan-inc/Baichuan2-13B-Chat	🟩	🟩	🟩	🟩	
Baichuan	baichuan-inc/Baichuan-13B-Chat	🟩	🟧	🟩	🟩	
ChatGLM	THUDM/chatglm3-6b	🟩	🟩	🟧	🟩	
ChatGLM	THUDM/chatglm2-6b	🟩	🟩	🟧	🟩	
GPTBigCode	bigcode/starcoder	🟩	🟩	🟧	🟩	🟧
T5	google/flan-t5-xl	🟩	🟩	🟧	🟩	
Mistral	mistralai/Mistral-7B-v0.1	🟩	🟩	🟧	🟩	🟧
MPT	mosaicml/mpt-7b	🟩	🟩	🟧	🟩	🟩
Mixtral	mistralai/Mixtral-8x7B-v0.1	🟩	🟩		🟩	
Stablelm	stabilityai/stablelm-2-1_6b	🟩	🟩		🟧	
Qwen	Qwen/Qwen-7B-Chat	🟩	🟩		🟩	

GPU:

MODEL FAMILY	Verified < MODEL ID > (Huggingface hub)	FP16	Weight only quantization INT4	Optimized on Intel® Data Center GPU Max Series (1550/1100)	Optimized on Intel® Arc™ A-Series Graphics (A770)
Llama 2	"meta-llama/Llama-2-7b-hf", "meta-llama/Llama-2-13b-hf", "meta-llama/Llama-2-70b-hf"	✅	✅	✅	✅
GPT-J	"EleutherAI/gpt-j-6b"	✅	✅	✅	✅
Qwen	"Qwen/Qwen-7B"	✅	✅	✅	✅
OPT	"facebook/opt-6.7b", "facebook/opt-30b"	✅	❎	✅	❎
Bloom	"bigscience/bloom-7b1", "bigscience/bloom"	✅	❎	✅	❎
ChatGLM3-6B	"THUDM/chatglm3-6b"	✅	❎	✅	❎
Baichuan2-13B	"baichuan-inc/Baichuan2-13B-Chat"	✅	❎	✅	❎

CPU - https://github.com/intel/intel-extension-for-pytorch/tree/v2.3.0%2Bcpu-rc0/examples/cpu/inference/python/llm

GPU - https://github.com/intel/intel-extension-for-pytorch/tree/xpu-main/examples/gpu/inference/python/llm

Deploying with INT8

Low-precision Optimization – INT8

What is Quantization?

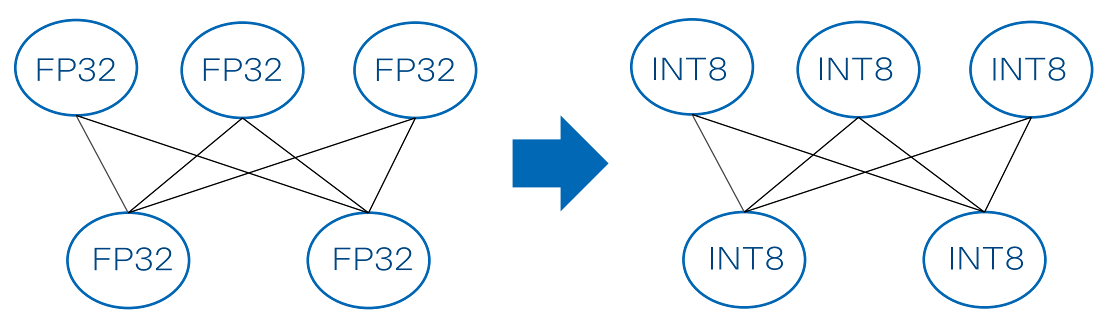
- Systematic reduction of the precision of all or several layers within the model.

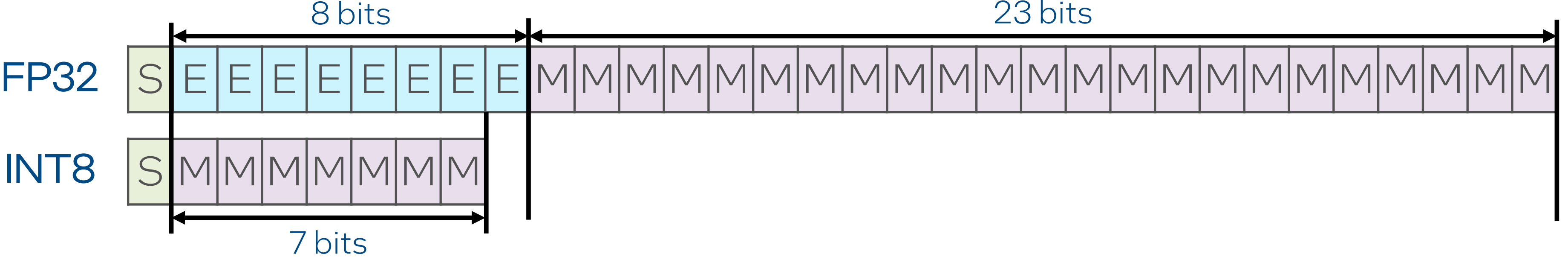
Why Quantization?

- Reduces model size. Uses less memory storage and bandwidth.
- Allows for faster inference.
- All with minimal accuracy loss.

How to Quantize?

- PyTorch quantization
- **IPEX quantization (with or w/o INC integration)**
- Intel Neural Compressor (INC)





Static vs Dynamic Quantization

Static (Preferred)

- Quantizes weights and activations of model
- Fuses activations into preceding layers
- **Requires calibration dataset** to determine optimal quantization parameters for activations
- Used when both memory bandwidth and compute savings are important
- Only works on inputs with fixed sizes; not all models are traceable; typically used for CNNs

Dynamic

- Weights are quantized ahead of time, but activations are quantized during inference
- Used when model execution time is dominated more by memory bandwidth than compute
- Can work on inputs with variable sizes; typically used for LSTM and Transformer models with small batch size

Quantization Workflow and API

Static Quantization

1. Import `intel_extension_for_pytorch` as `ipex`.
2. Import `prepare` and `convert` from `intel_extension_for_pytorch.quantization`.
3. Instantiate a config object from `torch.ao.quantization.QConfig` to save configuration data during calibration.
4. Prepare model for calibration.
5. Perform calibration against dataset.
6. Invoke `ipex.quantization.convert` function to apply the calibration configure object to the fp32 model object to get an INT8 model.
7. Save the INT8 model into a `pt` file.

```python
import os
import torch
#################### code changes ####################
import intel_extension_for_pytorch as ipex
from intel_extension_for_pytorch.quantization import prepare, convert
######################################################

model = Model()
model.eval()
data = torch.rand(<shape>)

qconfig = ipex.quantization.default_static_qconfig
# Alternatively, define your own qconfig:
#from torch.ao.quantization import MinMaxObserver, PerChannelMinMaxObserver, QConfig
#qconfig = QConfig(activation=MinMaxObserver.with_args(qscheme=torch.per_tensor_affine, dtype=torch.quint8),
#        weight=PerChannelMinMaxObserver.with_args(dtype=torch.qint8, qscheme=torch.per_channel_symmetric))
prepared_model = prepare(model, qconfig, example_inputs=data, inplace=False)

for d in calibration_data_loader():
  prepared_model(d)

converted_model = convert(prepared_model)
with torch.no_grad():
  traced_model = torch.jit.trace(converted_model, data)
  traced_model = torch.jit.freeze(traced_model)

traced_model.save("quantized_model.pt")
```

Dynamic Quantization

1. Import `intel_extension_for_pytorch` as `ipex`.
2. Import `prepare` and `convert` from `intel_extension_for_pytorch.quantization`.
3. Instantiate a config object from `torch.ao.quantization.QConfig` to save configuration data during calibration.
4. Prepare model for quantization.
5. Convert the model.
6. Run inference to perform dynamic quantization.
7. Save the INT8 model into a `pt` file.

```python
import os
import torch
#################### code changes ####################
import intel_extension_for_pytorch as ipex
from intel_extension_for_pytorch.quantization import prepare, convert
######################################################

model = Model()
model.eval()
data = torch.rand(<shape>)

dynamic_qconfig = ipex.quantization.default_dynamic_qconfig
# Alternatively, define your own qconfig:
#from torch.ao.quantization import MinMaxObserver, PlaceholderObserver, QConfig
#qconfig = QConfig(
#        activation = PlaceholderObserver.with_args(dtype=torch.float, compute_dtype=torch.quint8),
#        weight = PerChannelMinMaxObserver.with_args(dtype=torch.qint8, qscheme=torch.per_channel_symmetric))
prepared_model = prepare(model, qconfig, example_inputs=data)

converted_model = convert(prepared_model)
with torch.no_grad():
  traced_model = torch.jit.trace(converted_model, data)
  traced_model = torch.jit.freeze(traced_model)

traced_model.save("quantized_model.pt")
```

TorchScript and torch.compile()

TorchScript

- Converts PyTorch model into a graph for faster execution
- torch.jit.trace() traces and records all operations in the computational graph; requires a sample input
- torch.jit.script() parses the Python source code of the model and compiles the code into a graph; sample input not required

Resnet50

```
import torch
import torchvision.models as models

model = models.resnet50(weights='ResNet50_Weights.DEFAULT')
model.eval()
data = torch.rand(1, 3, 224, 224)

#################### code changes ####################
import intel_extension_for_pytorch as ipex
model = ipex.optimize(model, dtype=torch.bfloat16)
######################################################

with torch.no_grad(), torch.cpu.amp.autocast():
  model = torch.jit.trace(model, torch.rand(1, 3, 224, 224))
  model = torch.jit.freeze(model)

  model(data)
```

torch.compile() – in BETA

- Makes PyTorch code run faster by just-in-time (JIT)-compiling PyTorch code into optimized kernels

Verifying That AMX Is Used

How to Check If AMX Is Enabled

- On bash terminal, enter the following command:
  - *cat /proc/cpuinfo*
- Check the "flags" section for amx_bf16, amx_int8
- Alternatively, you can use:
  - *lscpu | grep amx*
- If you do not see them, upgrade to Linux kernel 5.17 and above

```
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse s
se2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpu
id aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 s
se4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cat_
l2 cdp_l3 invpcid_single intel_ppin cdp_l2 ssbd mba ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid ept_
ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflus
hopt clwb intel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_
mbm_local split_lock_detect avx_vnni avx512_bf16 wbnoinvd dtherm ida arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req hfi
 avx512vbmi umip pku ospke waitpkg avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg tme avx512_vpopcntdq la57 rdp
id bus_lock_detect cldemote movdiri movdir64b enqcmd fsrm uintr avx512_vp2intersect md_clear serialize tsxldtrk pconfig arch
_lbr amx_bf16 avx512_fp16 amx_tile amx_int8 flush_l1d arch_capabilities
```

How to Check AMX Is Actually Used

- Generate oneDNN Verbose logs using guide and parser
- To enable verbosity, set environment variables:
  - export DNNL_VERBOSE=1
  - export DNNL_VERBOSE_TIMESTAMP=1
- Set a Python breakpoint RIGHT AFTER one iteration of training/inference

oneDNN Verbose Sample Output

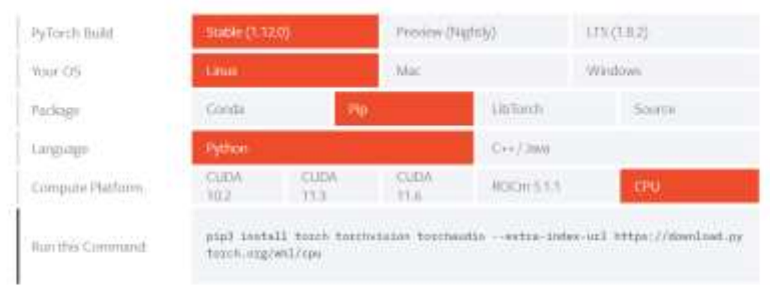
Sample oneDNN Verbose Output

```
onednn_verbose,info,oneDNN v2.6.0 (commit 52b5f107dd9cf10910aaa19cb47f3abf9b349815)
onednn_verbose,info,cpu,runtime:OpenMP,nthr:32
onednn_verbose,info,cpu,isa:Intel AVX-512 with Intel DL Boost
onednn_verbose,info,gpu,runtime:none
onednn_verbose,info,prim_template:timestamp,operation,engine,primitive,implementation,prop_kind,memory_descriptors,attributes,auxiliary,problem_desc,exec_time
onednn_verbose,1678917979730.501953,exec,cpu,reorder,jit:uni,undef,src_f32::blocked:abcd:f0 dst_f32:p:blocked:Acdb16a:f0,attr-scratchpad:user ,,1x1x1x37,0.00292969
onednn_verbose,1678917979730.888916,exec,cpu,convolution,jit:avx512_core,forward_training,src_f32::blocked:abcd:f0 wei_f32:p:blocked:Acdb16a:f0 bia_undef::undef::f0 dst_f
onednn_verbose,1678917979732.105957,exec,cpu,reorder,jit:uni,undef,src_f32:p:blocked:aBcd16b:f0 dst_f32::blocked:abcd:f0,attr-scratchpad:user ,,1x1x1x48000,0.0649414
onednn_verbose,1678917980009.694092,exec,cpu,reorder,jit:uni,undef,src_f32::blocked:abc:f0 dst_f32::blocked:acb:f0,attr-scratchpad:user ,,1x60x305,0.00878906
onednn_verbose,1678917980011.387939,exec,cpu,convolution,brgconv:avx512_core,forward_training,src_f32::blocked:acb:f0 wei_f32::blocked:Acb32a:f0 bia_f32::blocked:a:f0 dst_
onednn_verbose,1678917980012.134033,exec,cpu,reorder,jit:uni,undef,src_f32::blocked:abc:f0 dst_f32::blocked:acb:f0,attr-scratchpad:user ,,1x1024x301,0.278076
onednn_verbose,1678917980012.912109,exec,cpu,reorder,simple:any,undef,src_f32:p:blocked:Acb48a:f0 dst_f32::blocked:Acb64a:f0,attr-scratchpad:user ,,1024x1024x1,3.31201
```

- Note the ISA. For AMX, you should see the following:
  - Intel AMX with bfloat16 and 8-bit integer support
- Check for AMX in the primitive implementation:

```
onednn_verbose,1673049613345.454102,exec,cpu,convolution,brgconv:avx512_core_amx_bf16,forward_training,src_bf16::blocked:acdb:f0 wei_
onednn_verbose,1673049613348.691895,exec,cpu,convolution,brgconv_1x1:avx512_core_amx_bf16,forward_training,src_bf16::blocked:acdb:f0
onednn_verbose,1673049613353.259033,exec,cpu,convolution,brgconv_1x1:avx512_core_amx_bf16,forward_training,src_bf16::blocked:acdb:f0
onednn_verbose,1673049613364.104980,exec,cpu,convolution,brgconv_1x1:avx512_core_amx_bf16,forward_training,src_bf16::blocked:acdb:f0
```

How to get the Intel Extension for PyTorch



Note: Intel® Extension for PyTorch* has PyTorch version requirement. Check the mapping table here.

- pip wheel - CPU:

```
python -m pip install torch==2.3.0 torchvision==0.18.0 torchaudio==2.3.0 --index-url https://download.pytorch.org/whl/cpu
python -m pip install intel-extension-for-pytorch
python -m pip install oneccl_bind_pt --extra-index-url https://pytorch-extension.intel.com/release-whl/stable/cpu/us/
```

- pip wheel – GPU:

```
python -m pip install torch==2.1.0.post2 torchvision==0.16.0.post2 torchaudio==2.1.0.post2 intel-extension-for-pytorch==2.1.30.post0
oneccl_bind_pt==2.1.300+xpu --extra-index-url https://pytorch-extension.intel.com/release-whl/stable/xpu/us/
```

PyTorch AMX Training/Inference Code Samples

Training

GitHub: https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/Features-and-Functionality/IntelPyTorch_TrainingOptimizations_AMX_BF16

Trains a ResNet50 model with Intel Extension for PyTorch and shows performance speedup with AMX BF16

Inference

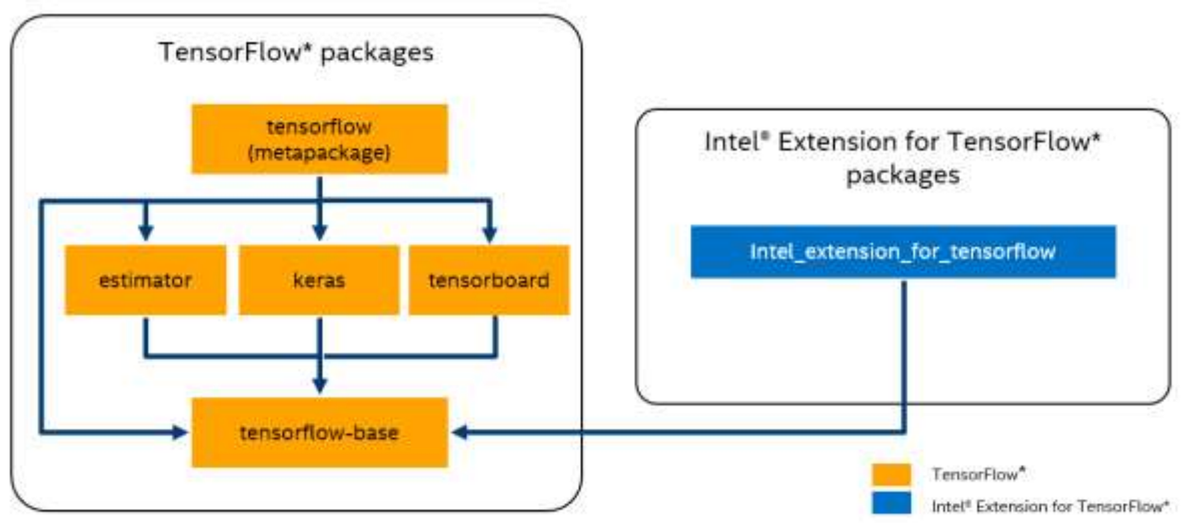
GitHub: https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/Features-and-Functionality/IntelPyTorch_InferenceOptimizations_AMX_BF16_INT8

Performs inference on ResNet50 and BERT with Intel Extension for PyTorch and shows performance speedup with AMX BF16 and INT8 over VNNI INT8

Intel® Extension for TensorFlow

Intel® Extension for TensorFlow* (ITEX)

- Provide users with the up-to-date Intel software/hardware features
- Streamline the work to integrate oneDNN
- Unify user experiences on Intel CPU and GPU



Intel® Extension for TensorFlow* PyPI packages and dependencies

How to use Intel® Extension for TensorFlow* - FP32

No code changes, the default backend will be Intel GPU after installing intel-extension-for-tensorflow[xpu]

OR

```
import intel_extension_for_tensorflow as itex

#CPU, GPU or AUTO
backend = "GPU"
itex.set_backend(backend)
```

Advanced Auto Mixed Precision - Environment Variable

- export ITEX_AUTO_MIXED_PRECISION=1
- export ITEX_AUTO_MIXED_PRECISION_DATA_TYPE="BFLOAT16" (or "FLOAT16“)

BF16 API

1. Train with BF16 with AVX-512

```
# BF16 without AMX
os.environ["ONEDNN_MAX_CPU_ISA"] = "AVX512_BF16"
tf.config.optimizer.set_experimental_options({'auto_mixed_precision_onednn_bfloat16':True})

transformer_layer = transformers.TFDistilBertModel.from_pretrained('distilbert-base-uncased')
tokenizer = transformers.DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
model = build_model(transformer_layer, max_len=160)

# fine tune model according to disaster tweets dataset
if is_tune_model:
    train_input = bert_encode(train.text.values, tokenizer, max_len=160)
    train_labels = train.target.values
    start_time = time.time()
    train_history = model.fit(train_input, train_labels, validation_split=0.2, epochs=1, batch_size=16)
    end_time = time.time()
  # save model weights so we don't have to fine tune it every time
    os.makedirs(save_weights_dir, exist_ok=True)
    model.save_weights(save_weights_dir + "/bf16_model_weights.h5")
```

Turned on by default after TF 2.11

2. Train with BF16 with AMX

```
# BF16 without AMX
os.environ["ONEDNN_MAX_CPU_ISA"] = "AVX512_BF16"
```

```
# BF16 with AMX
os.environ["ONEDNN_MAX_CPU_ISA"] = "AMX_BF16"
```

BF16 API (cont.)

3. Inference with BF16 without AMX

```
# Reload the model as the bf16 model with AVX512 to compare inference time
os.environ["ONEDNN_MAX_CPU_ISA"] = "AVX512_BF16"
tf.config.optimizer.set_experimental_options({'auto_mixed_precision_onednn_bfloat16':True})
bf16_model_noAmx = tf.keras.models.load_model('models/my_saved_model_fp32')

bf16_model_noAmx_export_path = "models/my_saved_model_bf16_noAmx"
bf16_model_noAmx.save(bf16_model_noAmx_export_path)
```

4. Inference with BF16 with AMX

```
# Reload the model as the bf16 model with AMX to compare inference time
os.environ["ONEDNN_MAX_CPU_ISA"] = "AMX_BF16"
tf.config.optimizer.set_experimental_options({'auto_mixed_precision_onednn_bfloat16':True})
bf16_model_withAmx = tf.keras.models.load_model('models/my_saved_model_fp32')

bf16_model_withAmx_export_path = "models/my_saved_model_bf16_with_amx"
bf16_model_withAmx.save(bf16_model_withAmx_export_path)
```

How to get the Intel® Extension for TensorFlow*

- pip wheel - GPU:

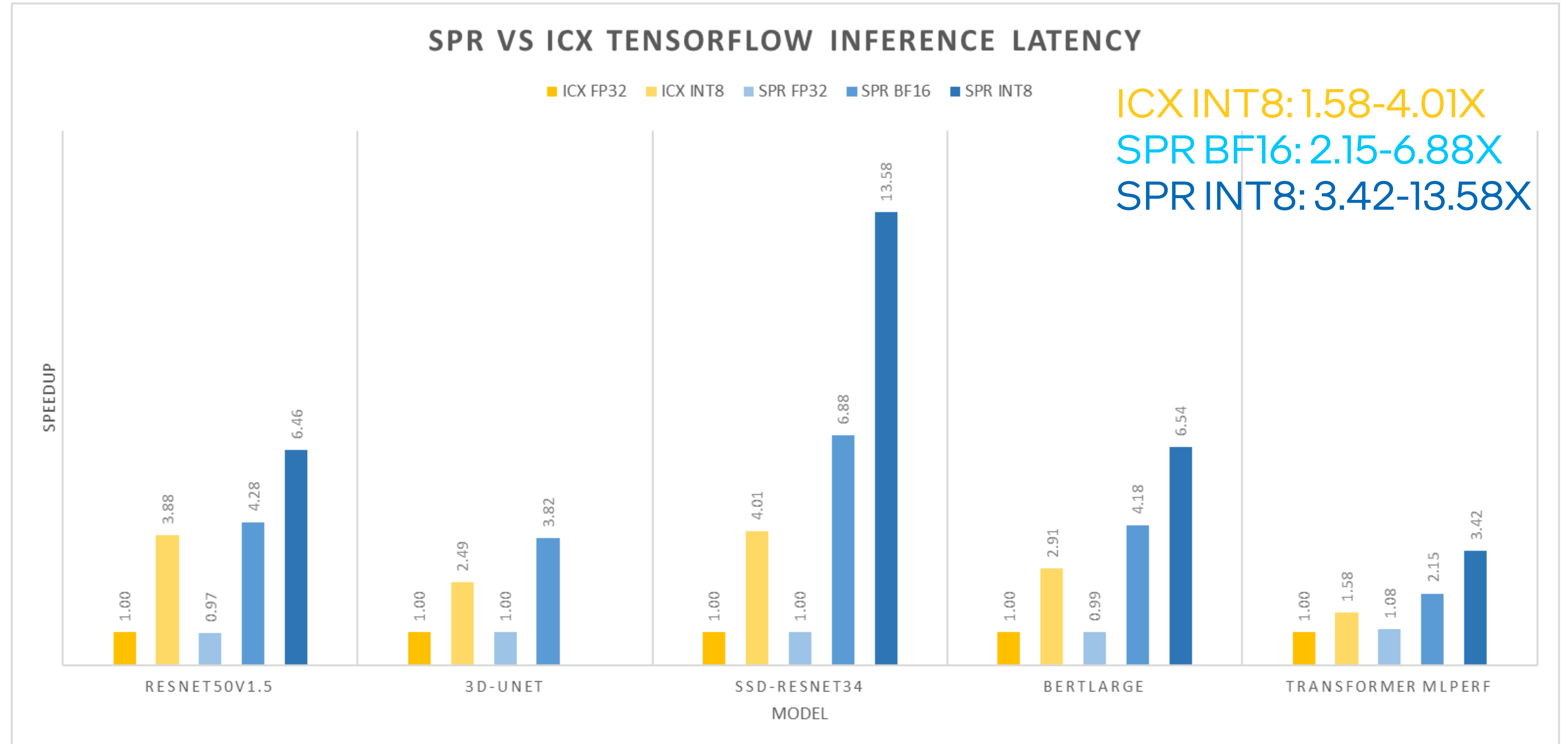
pip install --upgrade intel-extension-for-tensorflow[xpu]

- pip wheel - CPU (experimental)

pip install --upgrade intel-extension-for-tensorflow[cpu]

TensorFlow Benchmark: SPR vs ICX Inference (Batch Size = 1)

Inference latency speedup: the higher the better



Benchmark data for the Intel® 4th Gen Xeon Scalable Processors can be found here.
Also check Appendix for test configurations.

TensorFlow AMX Training/Inference Code Samples

- Training
  - GitHub: https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/Features-and-Functionality/IntelTensorFlow_AMX_BF16_Training
  - Trains a DistilBERT model using Intel Optimization for TensorFlow and shows performance speedup with AMX BF16
- Inference
  - GitHub: https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/Features-and-Functionality/IntelTensorFlow_AMX_BF16_Inference
  - Performs inference on ResNet50v1.5 with Intel Optimization for TensorFlow and shows performance speedup with AMX BF16

Optimizations under IPEX & ITEX

Operator optimizations

Memory/data layout optimizations

Graph optimizations

Mixed Precision

Operator Optimizations

- Replace default kernels by highly-optimized kernels (using Intel® oneDNN)
- Adapt to available instruction sets (AMX, AVX-512, AVX2, VNNI)
- Adapt to required precision:
  - **Training**: FP32, BF16
  - **Inference**: FP32, BF16, FP16, and INT8

	Intel® oneDNN
Convolution	2D/3D Direct Convolution/Deconvolution, Depthwise separable convolution 2D Winograd convolution
Inner Product	2D/3D Inner Production
Pooling	2D/3D Maximum 2D/3D Average (include/exclude padding)
Normalization	2D/3D LRN across/within channel, 2D/3D Batch normalization
Eltwise (Loss/activation)	ReLU(bounded/soft), ELU, Tanh; Softmax, Logistic, linear; square, sqrt, abs, exp, gelu, swish
Data manipulation	Reorder, sum, concat, View
RNN cell	RNN cell, LSTM cell, GRU cell
Fused primitive	Conv+ReLU+sum, BatchNorm+ReLU
Data type	f32, bfloat16, s8, u8

Linear Operator Optimization for LLMs

- Optimization of Linear GEMM Kernels in LLM Inference:
- CPU Optimizations:
  - Utilizes Intel® oneDNN and customized linear kernels for efficient weight-only quantization.
  - Employs specific block formats to maximize hardware resource utilization.
- GPU Optimizations:
  - Incorporates Intel® oneDNN and Intel® Xe Templates for Linear Algebra (XeLTA) to enhance performance.
  - Customized linear kernels for weight-only quantization streamline GPU computations.
- Common Strategies:
  - Both CPU and GPU optimizations focus on accelerating linear GEMM operations critical for LLM inference.
  - Targeted optimizations to meet the specific demands of memory-bound linear weight computations in LLMs.

Operator optimizations

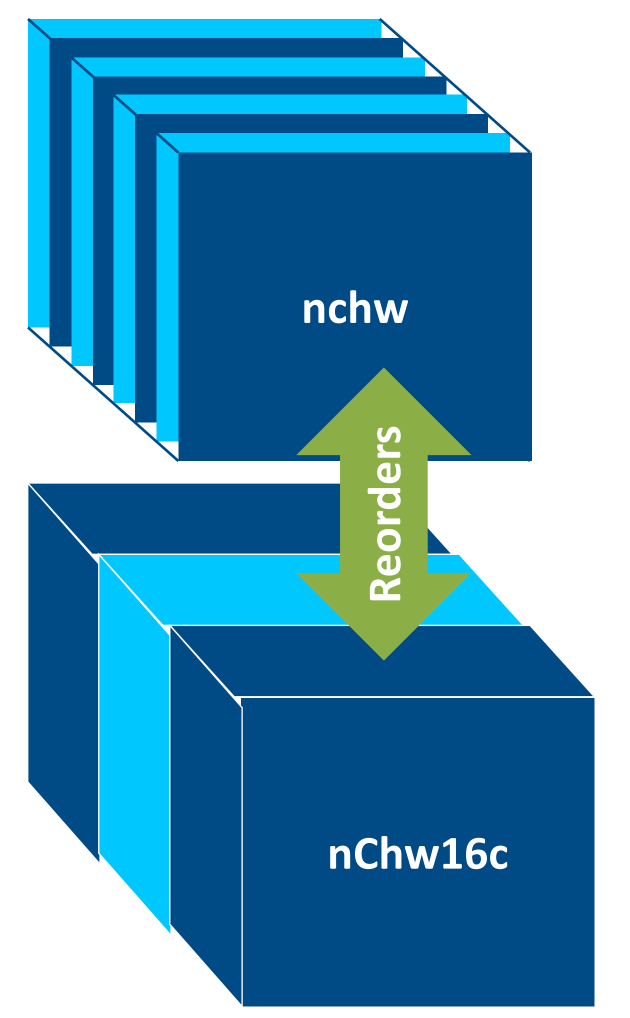
Memory/data layout optimizations

Graph optimizations

Mixed Precision

Memory Layouts Optimization

- Most popular memory layouts for image recognition are **NHWC** and **NCHW**
  - Challenging for Intel processors both for vectorization or for memory accesses
- Intel oneDNN convolutions use blocked layouts
  - Most popular oneDNN data format is nChw16c on AVX512+ systems and nChw8c on SSE4.1+ systems



More details: https://oneapi-src.github.io/oneDNN/dev_guide_understanding_memory_formats.html

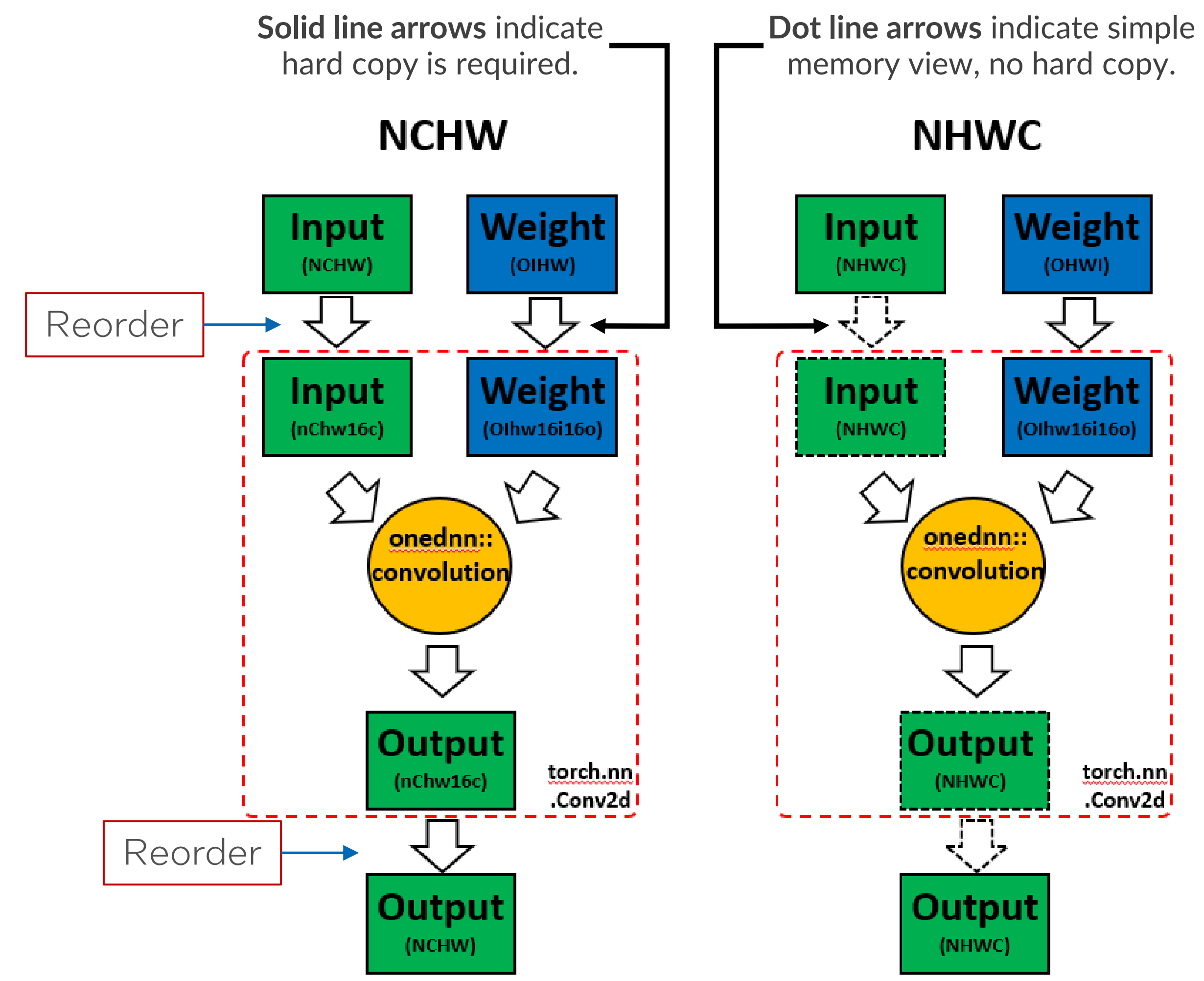
Data Layouts in PyTorch

- Used in Vision workloads
- NCHW
  - Default format
  - *torch.contiguous_format*
- NHWC
  - *torch.channels_last*
  - NHWC format yields higher performance with IPEX

NCHW	[0]	[1]	[2]	[3]	[0]	[1]	[2]	[3]	[0]	[1]	[2]	[3]
NHWC	[0]	[0]	[0]	[1]	[1]	[1]	[2]	[2]	[2]	[3]	[3]	[3]

Channels last conversion is now applied **automatically** with IPEX
Users do not have to explicitly convert input and weight for CV models.

Benefit of NHWC in IPEX



Operator optimizations

Memory/data layout optimizations

Graph optimizations

Mixed Precision

Graph Optimizations: Layout Propagation

Input → Conv2D; Filter → Conv2D; Conv2D → ReLU → Shape

Initial Graph

Input → Convert; Filter → Convert; Convert → oneDNN-Conv2D; Convert → oneDNN-Conv2D; oneDNN-Conv2D → Convert → Convert → oneDNN - ReLU → Convert → Shape

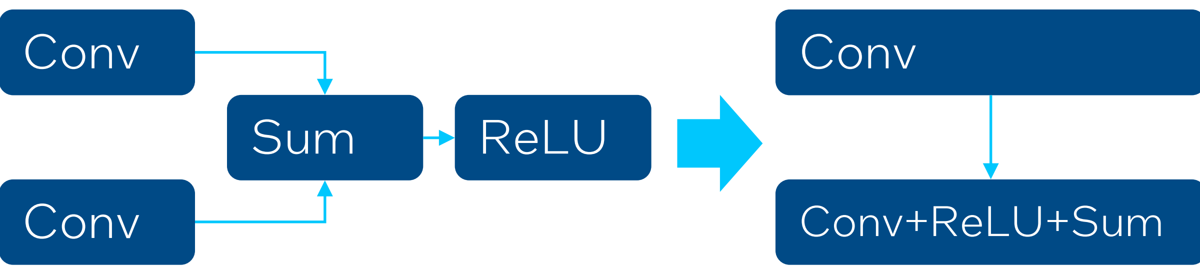
After Layout Conversions

Input → Convert; Filter → Convert; Convert → oneDNN-Conv2D; Convert → oneDNN-Conv2D; oneDNN-Conv2D → oneDNN-ReLU → Convert → Shape

After Layout Propagation

Fusing Computations

- On Intel processors a high percentage of time is typically spent in bandwidth-limited ops such activation functions
  - ~40% of ResNet-50, even higher for inference
- The solution is to fuse BW-limited ops with convolutions or one with another to reduce the number of memory accesses
  - We fuse patterns: Conv+ReLU+Sum, BatchNorm+ReLU, etc…



Graph Optimizations: Fusion

Fusing Computations in IPEX

- Intel® Extension for PyTorch in JIT/Torchscript mode can fuse:
  - Multi-head-attention fusion, Conv(2, 3)D+SUM+ReLU, Conv(2, 3)D + Sigmoid, Concat Linear, Linear+Add, Linear+Gelu, Add+LayerNorm fusion, etc.
- Hugging Face reports that ~70% of most popular NLP tasks in question-answering, text-classification, and token-classification can get performance benefits with such fusion patterns [1]
  - For both Float32 precision and BFloat16 Mixed precision

[1] https://huggingface.co/docs/transformers/perf_infer_cpu

Fusing Computations in LLMs

- Operator Fusion Strategy:
  - Reduces memory footprint on CPUs and decreases memory access and kernel launches on GPUs.
- Specific Fusion Techniques:
  - **Linear Post Ops Fusion**: Combines linear operations with activation functions for improved efficiency.
- Customized Operators for Performance:
  - **Examples**:
    - **Rotary Position Embedding (RoPE)**: Enhances positional calculations.
    - **Root Mean Square Layer Normalization (RMSNorm)**: Streamlines normalization processes.
  - **Available for Both CPU and GPU**: Tailored to exploit the architectural advantages of both platforms.

Operator optimizations

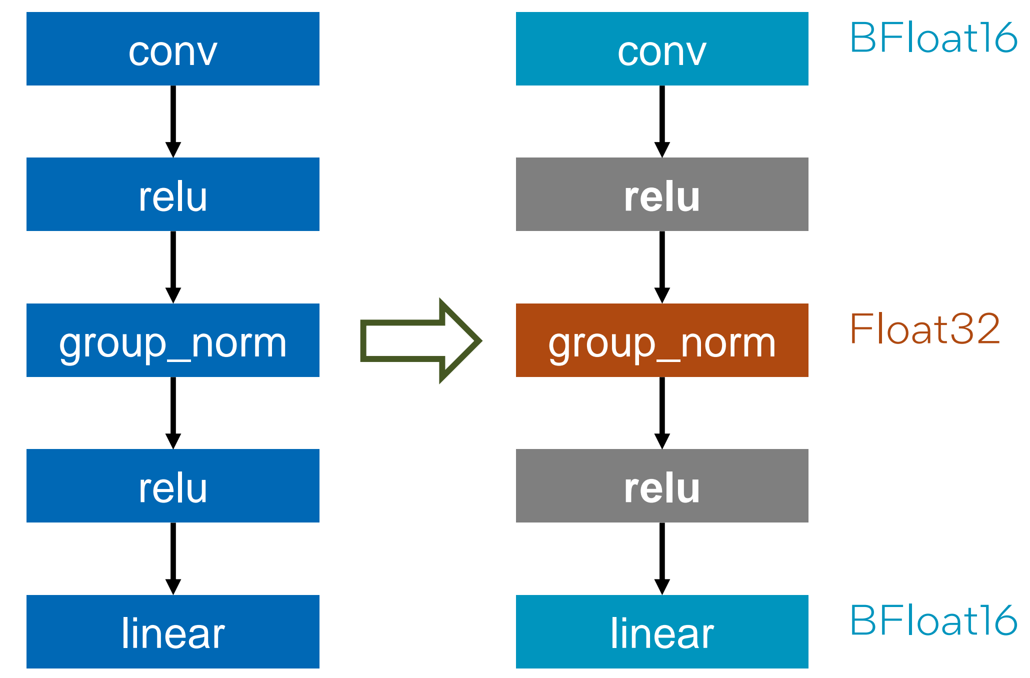
Memory/data layout optimizations

Graph optimizations

Mixed Precision

Auto Mixed Precision (AMP)

- 3 Categories of operators
  - lower_precision_fp
    - Computation bound operators that could get performance boost with BFloat16.
    - E.g.: conv, linear
  - Fallthrough
    - Operators that runs with both Float32 and BFloat16 but might not get performance boost with BFloat16.
    - E.g.: relu, max_pool2d
  - FP32
    - Operators that are not enabled with BFloat16 support yet. Inputs of them are casted into float32 before execution.
    - E.g.: max_pool3d, group_norm



Profiling tools

CPU – PyTorch* Profiler

Measure time and memory consumption

- Use built-in PyTorch profiler API to gain information about operator overhead

Example Use

```
model = models.resnet50(weights='ResNet50_Weights.DEFAULT')
model.eval()
data = torch.rand(1, 3, 224, 224)

model = ipex.optimize(model)

with profile(activities=[ProfilerActivity.CPU], record_shapes=True) as prof:
    with record_function("model_inference"):
        model(data)

print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

Direct Output

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
torch_ipex::convolution_forward	1.77%	1.021ms	158.19%	91.322ms	861.528us	106
model_inference	11.47%	6.623ms	100.00%	57.730ms	57.730ms	1
IPEXConvolutionOp::forward	-4.36%	-2518.000us	78.72%	45.447ms	857.491us	53
IPEXConvolutionOp::_forward	5.08%	2.935ms	78.34%	45.226ms	853.321us	53
torch_ipex::convolution_forward_impl	76.89%	44.387ms	77.72%	44.867ms	846.547us	53
aten::relu_	1.24%	714.000us	3.66%	2.111ms	43.082us	49
aten::add	2.50%	1.445ms	2.50%	1.445ms	90.312us	16
aten::clamp_min_	2.42%	1.397ms	2.42%	1.397ms	28.510us	49
aten::select	0.80%	462.000us	0.91%	524.000us	9.704us	54
aten::empty	0.73%	422.000us	0.73%	422.000us	3.836us	110

GPU – Legacy Profiler Tool

Experimental

- Extension of PyTorch* legacy profiler for profiling operators' overhead on XPU devices
- Users can get the information in many fields of the run models or code scripts
- Export to Chrome Trace

Example Use

```python
# import all necessary libraries
import torch
import intel_extension_for_pytorch

# these lines won't be profiled before enabling profiler tool
input_tensor = torch.randn(1024, dtype=torch.float32, device='xpu:0')

# enable legacy profiler tool with a `with` statement
with torch.autograd.profiler_legacy.profile(use_xpu=True) as prof:
    # do what you want to profile here after the `with` statement with proper indent
    output_tensor_1 = torch.nonzero(input_tensor)
    output_tensor_2 = torch.unique(input_tensor)

# print the result table formatted by the legacy profiler tool as your wish
print(prof.key_averages().table())
```

Direct Output

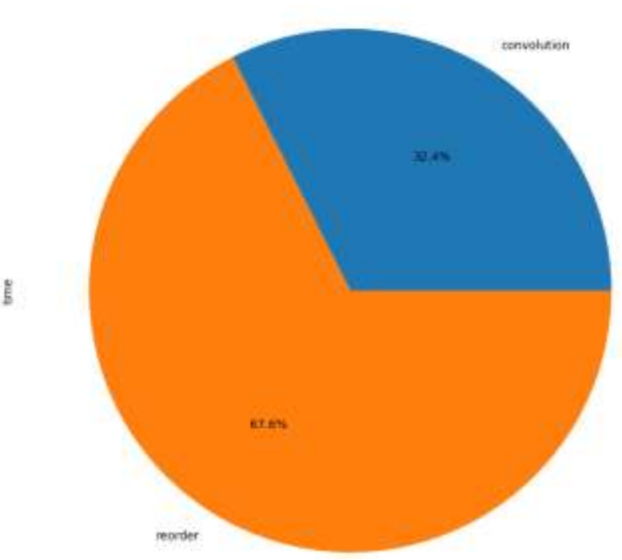
Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self XPU	Self XPU %	XPU total	XPU time avg	# of Calls
aten::nonzero	1.29%	13.910ms	51.11%	550.374ms	550.374ms	26.240us	15.08%	59.840us	59.840us	1
aten::empty	0.02%	174.518us	0.02%	174.518us	13.424us	0.000us	0.00%	0.000us	0.000us	13
copy_if_xpu	43.79%	471.609ms	49.81%	536.371ms	536.371ms	15.040us	8.64%	33.600us	33.600us	1
inclusive_scan_xpu	5.41%	58.230ms	5.41%	58.230ms	58.230ms	15.360us	8.78%	15.360us	15.360us	1
aten::select	0.06%	604.505us	0.14%	1.515ms	505.005us	0.000us	0.00%	0.000us	0.000us	3
aten::as_strided	0.08%	910.659us	0.08%	910.659us	303.553us	0.000us	0.00%	0.000us	0.000us	3
aten::item	0.18%	1.940ms	0.53%	5.757ms	1.919ms	0.000us	0.00%	0.000us	2.933us	3
aten::_local_scalar_dense	0.35%	3.816ms	0.35%	3.816ms	1.272ms	0.000us	5.03%	0.000us	2.933us	3
aten::resize_	0.00%	51.493us	0.00%	51.493us	25.747us	0.000us	0.00%	0.000us	0.000us	2
aten::_unique2	0.78%	8.387ms	48.89%	526.406ms	526.406ms	0.000us	0.00%	115.040us	115.040us	1
aten::clone	0.15%	1.591ms	20.55%	221.348ms	221.348ms	0.000us	0.00%	5.440us	5.440us	1
aten::empty_strided	0.00%	58.469us	0.00%	58.469us	58.469us	0.000us	0.00%	0.000us	0.000us	1
aten::copy_	20.40%	219.708ms	20.40%	219.708ms	219.708ms	5.440us	3.11%	5.440us	5.440us	1
aten::reshape	0.00%	21.341us	0.17%	1.855ms	1.855ms	0.000us	0.00%	0.000us	0.000us	1
aten::_reshape_alias	0.17%	1.833ms	0.17%	1.833ms	1.833ms	0.000us	0.00%	0.000us	0.000us	1
iota_xpu	0.04%	389.862us	0.04%	389.862us	389.862us	5.440us	3.11%	5.440us	5.440us	1
pstl::sort	14.04%	151.192ms	14.04%	151.192ms	151.192ms	56.800us	32.40%	56.800us	56.800us	1
unique_kernel_xpu	13.20%	142.120ms	13.30%	143.242ms	143.242ms	25.760us	14.73%	47.360us	47.360us	1
exclusive_scan_xpu	0.03%	299.959us	0.03%	299.959us	299.959us	16.000us	9.15%	16.000us	16.000us	1

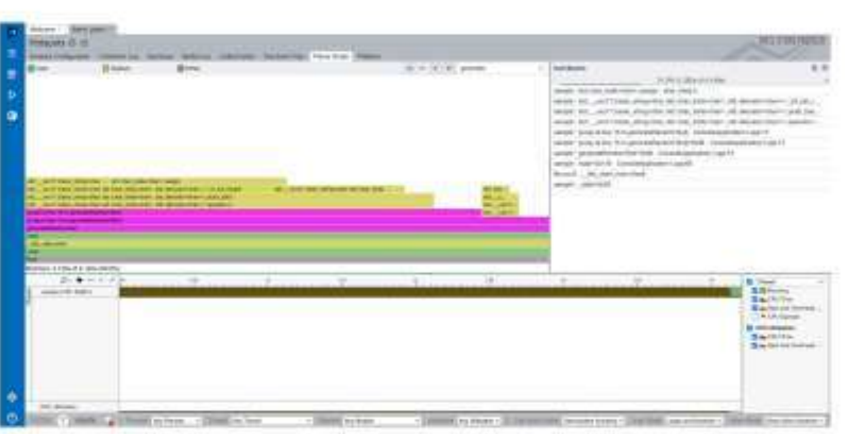
Self CPU time total: 1.077s
Self XPU time total: 174.880us

Profilers

- Built-in PyTorch profiler
- You can profile your application via oneDNN verbose logs.
  - DNN_VERBOSE=1 python application.py
  - You can also use profile_utils.py script to parse oneDNN verbose logs.
  - Code sample on oneDNN profiling can be found here: https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDNN/tutorials/profiling

- Another famous profiling tool is VTune from Intel which provides very deep hardware information and show them in easier way on how to **optimize the performance.** You can easily **find the hotspots** using VTune. (most costly functions)





Intel® XPU Manager

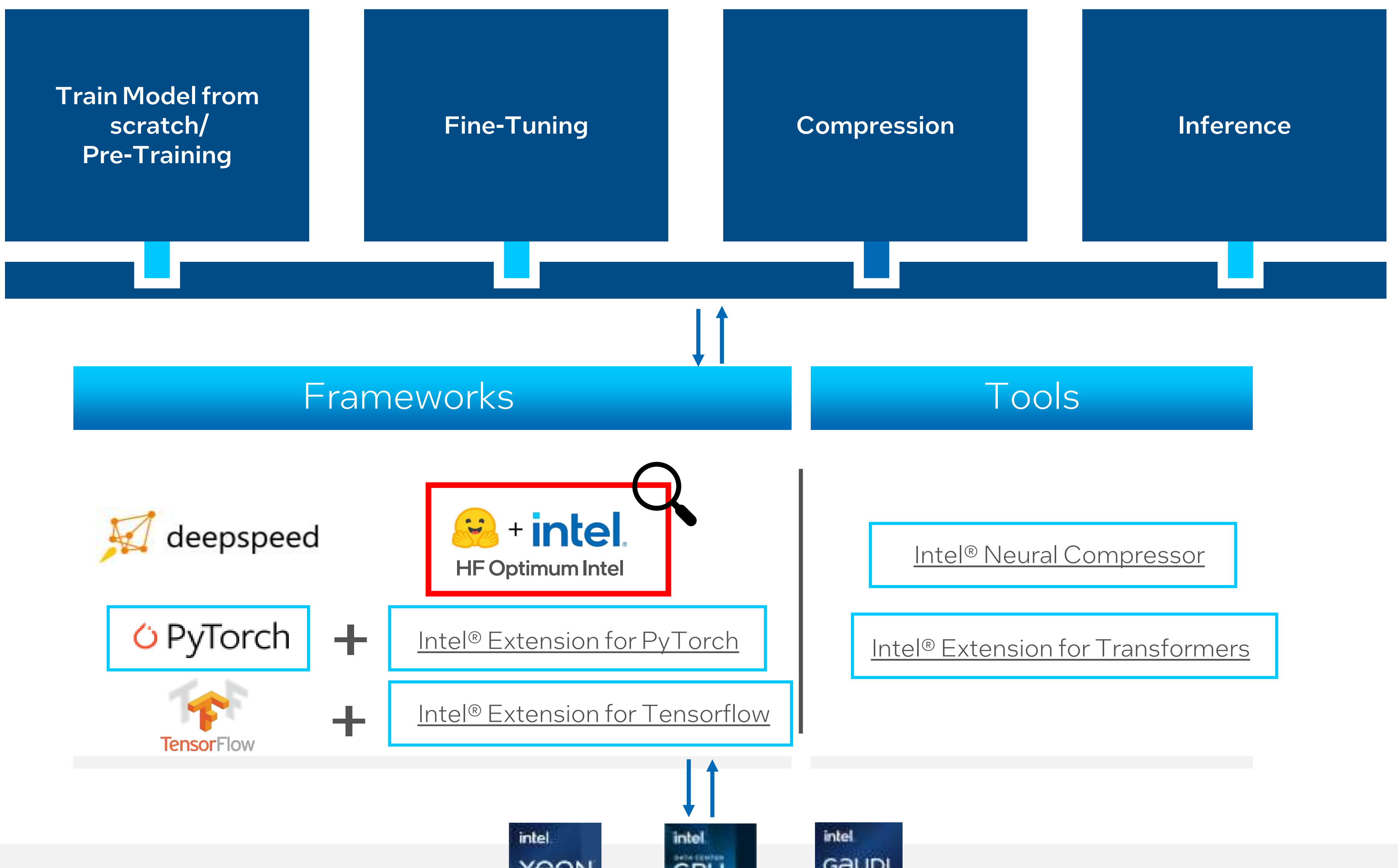
- Intel® XPU Manager is a free and open-source tool for monitoring and managing Intel data center GPUs.
- XPU Manager can be used standalone through its command line interface (CLI) to manage GPUs locally, or through its RESTful APIs to manage GPUs remotely.
- Can be downloaded through binary packages or docker image.
- Important Links:
  - https://github.com/intel/xpumanager
  - https://www.intel.com/content/www/us/en/software/xpu-manager.html
- Please note:
  - If you want to use XPU Manager, please uninstall XPU-SMI (comes default through XPU drivers, subset of XPU manager) and install XPU Manager

Recipe for Intel® Optimizations with IPEX

Easy Recipe for Intel® Optimizations with IPEX

- Add IPEX
- Add some Warmup steps for oneDNN initialization
- Utilize AMX or XMX instruction sets with efficient bfloat16 data type
- Utilize graph mode with TorchScript
- Quantize model to INT8
- Runtime optimizations with Performance Tuning Guide in case of cpu
- Use Advanced configuration in case of xpu.
- Distributed training with oneCCL/ DDP/Horovod/FSDP/DeepSpeed.
- Profile with oneDNN verbose / Pytorch Profiler / VTune for further analysis.

GenAI Deep Learning Funnel Pipeline



Dolly	LLAMA2	MPT	LDM3D	Whisper	*Hundreds of thousands more…*

Intel Optimized Hugging Face Libraries & Tools

Transformers	Diffusers	Accelerate	PEFT	Optimum
Fine Tuning for NLP,CV	Generative Use Cases	Fine Tuning at Scale	Efficient Fine Tuning	Performance Optimization

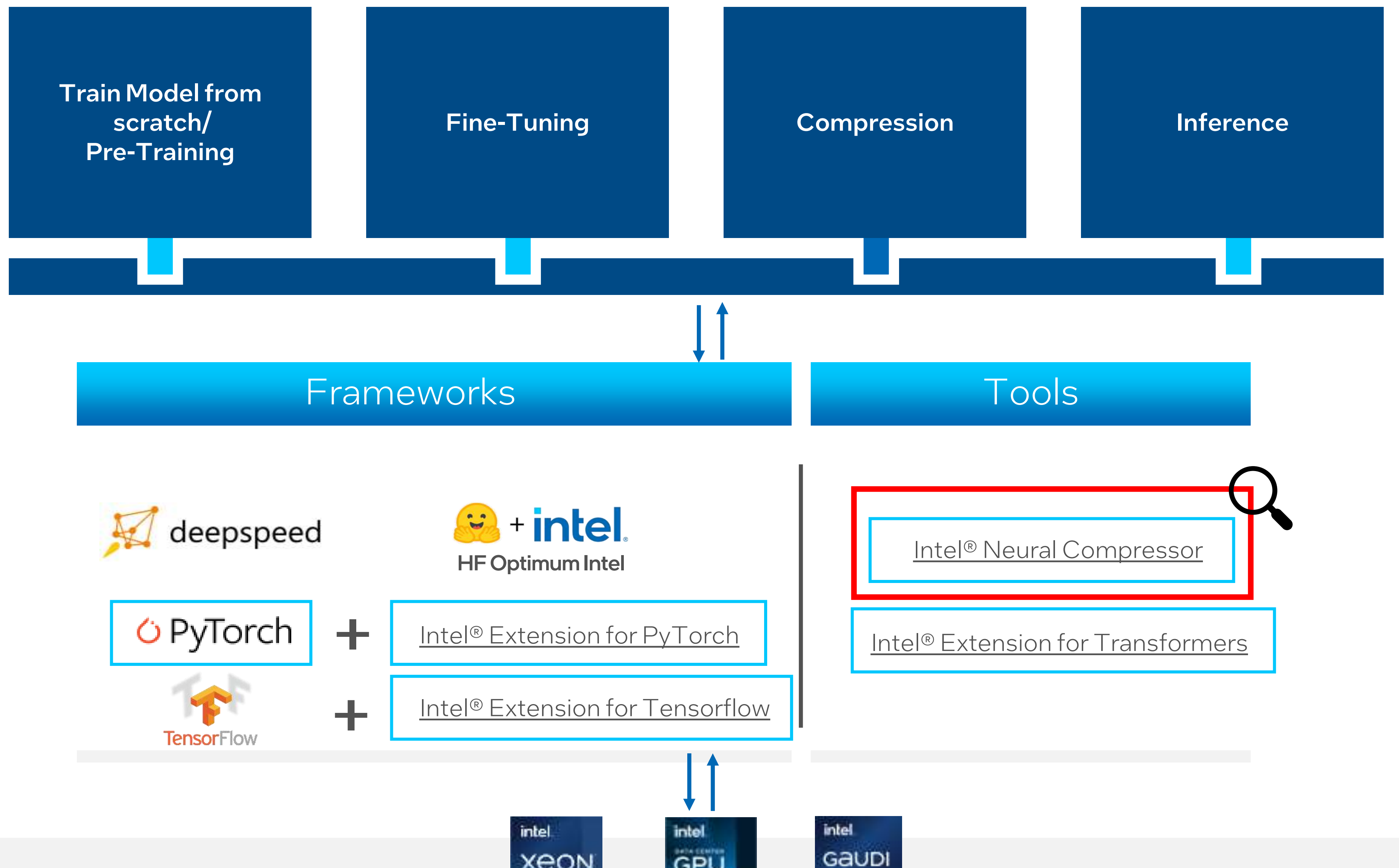
Foundational Stack

OpenVINO

Fine Tuning workflows on Hugging Face Platform optimized OOB for Intel products

https://huggingface.co/Intel

GenAI Deep Learning Funnel Pipeline



Intel® Neural Compressor

Intel® Neural Compressor

Intel® Neural Compressor is designed to use automatic accuracy-aware tuning strategies to help user **easily & quickly** find out the best optimization methods.

- Reduce accuracy loss:

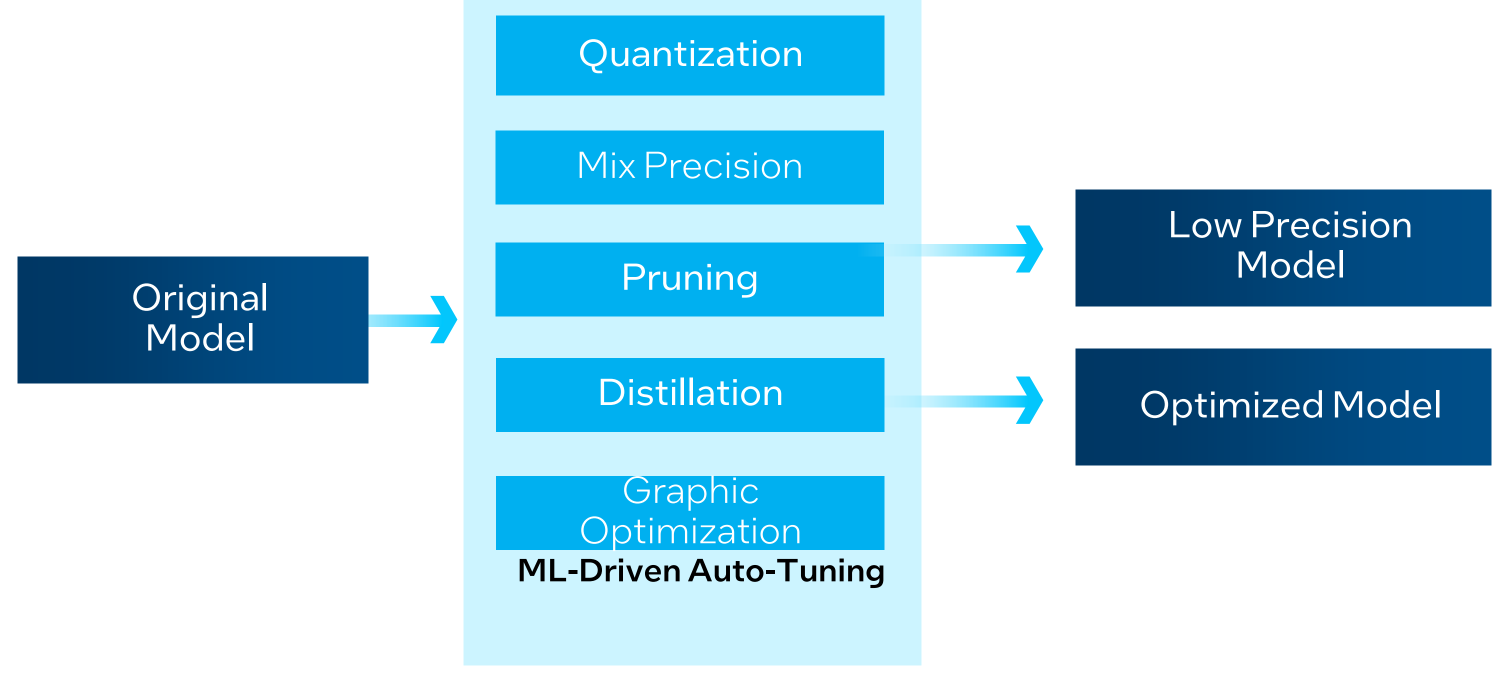
Automatic accuracy-driven quantization strategies

- Collaborate with:

Cloud marketplaces such as Google Cloud Platform, Amazon Web Services, and Azure

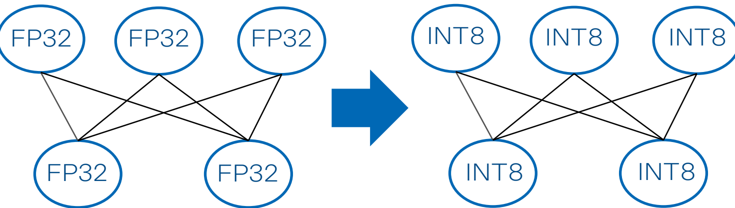
Software platforms such as Alibaba Cloud, Tencent TACO and Microsoft Olive

Open AI ecosystem such as Hugging Face, PyTorch, ONNX, ONNX Runtime, and Lightning AI

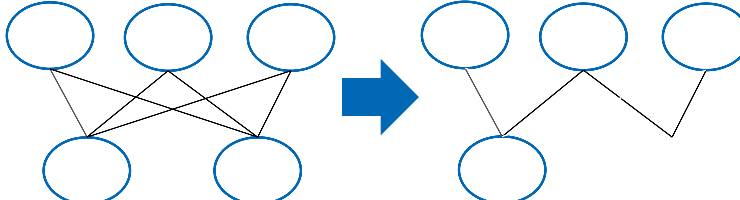


Deep Learning Inference Optimization

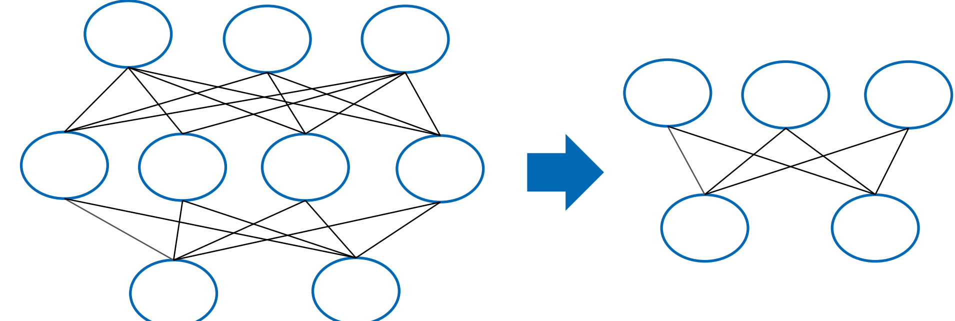
Quantization



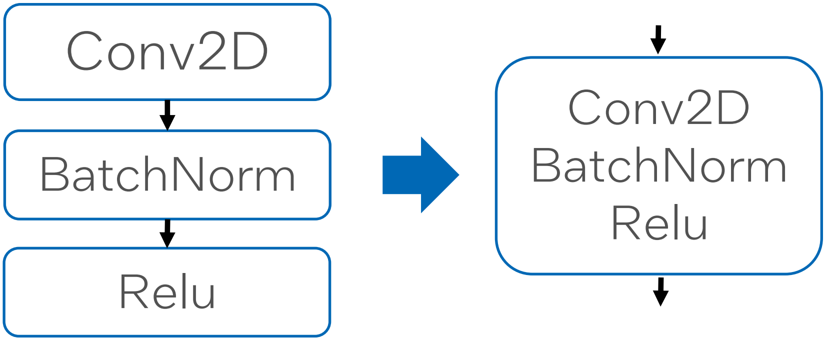
Pruning



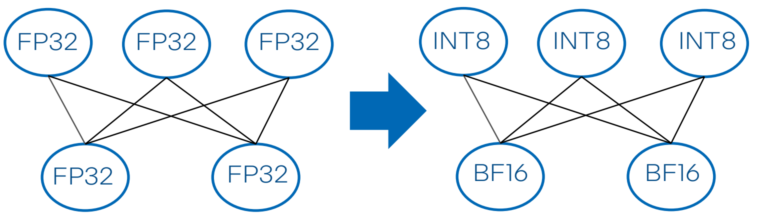
Knowledge Distillation



Graph Optimization



Mixed Precision Graph Optimization



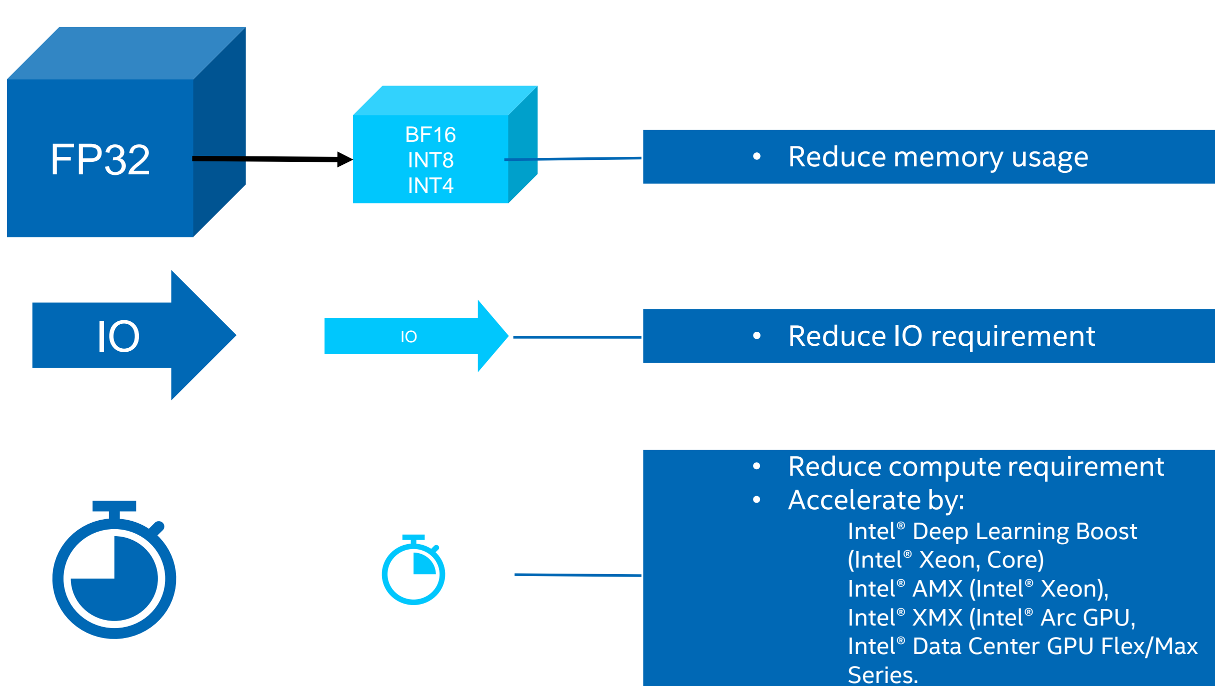
INC use automatic accuracy-driven tuning strategies to help user **easily & quickly** find out the best optimization methods above.

Quantization for LLM and GenAI

- Support Popular LLMs:

Bloom-176B, OPT-6.7B, Stable Diffusion, GPT-J, BERT-Large from popular model hubs such as Hugging Face, Torch Vision, and ONNX Model Zoo

- HuggingFace style API

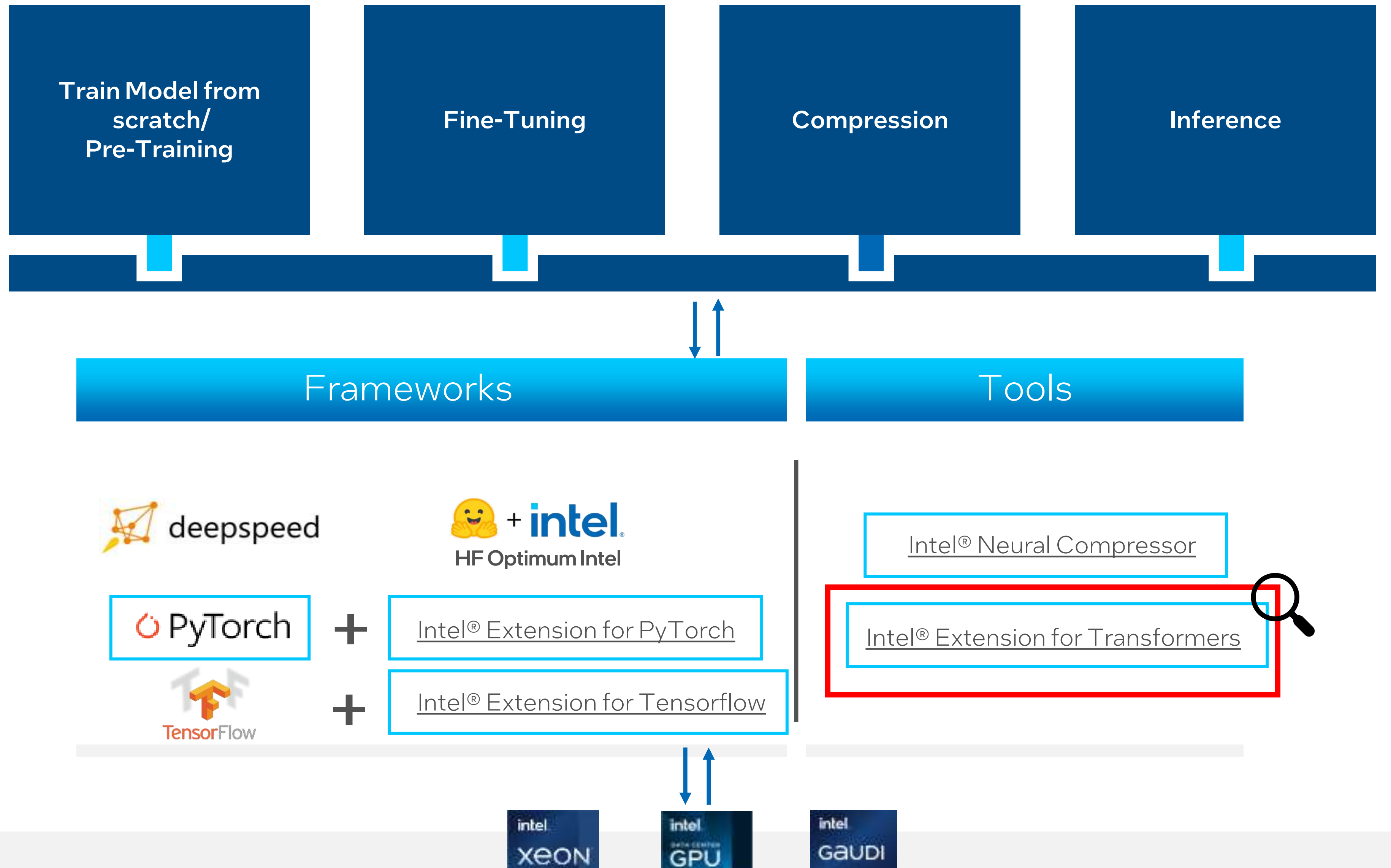


Getting Intel® Neural Compressor

- Pip Installation:
  # Install 2.X API + Framework extension API + PyTorch dependency
  - **pip install neural-compressor[pt]**

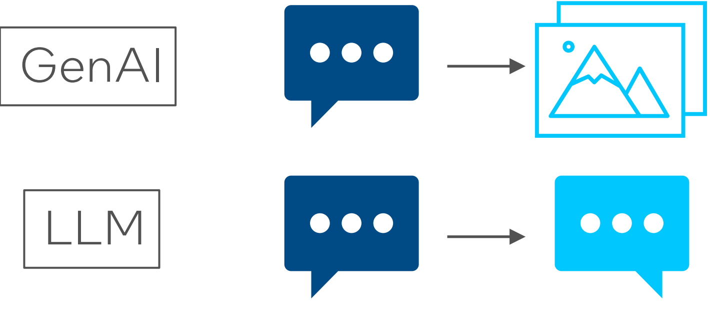
  # Install 2.X API + Framework extension API + TensorFlow dependency
  - **pip install neural-compressor[tf]**
- Intel® Neural Compressor is included in the **Intel® AI Analytics Toolkit (AI Kit):**
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/ai-analytics-toolkit-download.html?operatingsystem=linux
- Download the Stand-Alone Version:
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/neural-compressor.html
- Use Intel® Developer Cloud:
  - https://www.intel.com/content/www/us/en/secure/forms/devcloud/enrollment.html?tgt=www.intel.com/content/www/us/en/secure/forms/devcloud-enrollment/account-provisioning.html

GenAI Deep Learning Funnel Pipeline



Intel® Extension for Transformers

Intel® Extension for Transformers



Transformers style API

Intel® Extension for Transformers

- Seamless user experience of model compressions on Transformer-based models by extending Hugging Face transformers APIs
- Advanced software optimizations and unique compression-aware runtime
- NeuralChat, a customizable chatbot framework to create your own chatbot within minutes by leveraging a rich set of plugins
- Inference of Large Language Model (LLM) in pure C/C++ with weight-only quantization kernels

Ref: https://github.com/intel/intel-extension-for-transformers

Intel® Extension for Transformers Features

Hugging Face Transformers

Model Compression	Neural Speed*	Neural Chat
	*Separate installation starting v1.3.1	
• LLM Compression • General Compression	• Inference of Large Language Model (LLM) in pure C/C++ (llama.cpp inspired) • Streaming LLM • Tensor parallelism	• Framework for customizable chatbot • OpenAI-compatible RESTful API • LangChain extension API

Installation & Validated Configurations

```
pip install intel-extension-for-transformers
```

*With requirement.txt for specific use-cases and features

OS: Ubuntu 20.04/22.04, Centos 8.

Hardware

Hardware	Fine-Tuning		Inference	
	Full	PEFT	8-bit	4-bit
Intel Gaudi2	✔	✔	WIP (FP8)	-
Intel Xeon Scalable Processors	✔	✔	✔ (INT8, FP8)	✔ (INT4, FP4, NF4)
Intel Xeon CPU Max Series	✔	✔	✔ (INT8, FP8)	✔ (INT4, FP4, NF4)
Intel Data Center GPU Max Series	WIP	WIP	WIP (INT8)	✔ (INT4)
Intel Arc A-Series	-	-	WIP (INT8)	✔ (INT4)
Intel Core Processors	-	✔	✔ (INT8, FP8)	✔ (INT4, FP4, NF4)

In the table above, "-" means not applicable or not started yet.

Software

Software	Fine-Tuning		Inference	
	Full	PEFT	8-bit	4-bit
PyTorch	2.0.1+cpu, 2.0.1a0 (gpu)	2.0.1+cpu, 2.0.1a0 (gpu)	2.1.0+cpu, 2.0.1a0 (gpu)	2.1.0+cpu, 2.0.1a0 (gpu)
Intel® Extension for PyTorch	2.1.0+cpu, 2.0.110+xpu	2.1.0+cpu, 2.0.110+xpu	2.1.0+cpu, 2.0.110+xpu	2.1.0+cpu, 2.0.110+xpu
Transformers	4.35.2(CPU), 4.31.0 (Intel GPU)	4.35.2(CPU), 4.31.0 (Intel GPU)	4.35.2(CPU), 4.31.0 (Intel GPU)	4.35.2(CPU), 4.31.0 (Intel GPU)
Synapse AI	1.13.0	1.13.0	1.13.0	1.13.0
Gaudi2 driver	1.13.0-ee32e42	1.13.0-ee32e42	1.13.0-ee32e42	1.13.0-ee32e42
intel-level-zero-gpu	1.3.26918.50-736~22.04	1.3.26918.50-736~22.04	1.3.26918.50-736~22.04	1.3.26918.50-736~22.04

Intel® Extension for Transformers

Supported LLM(Large language Model) Model List

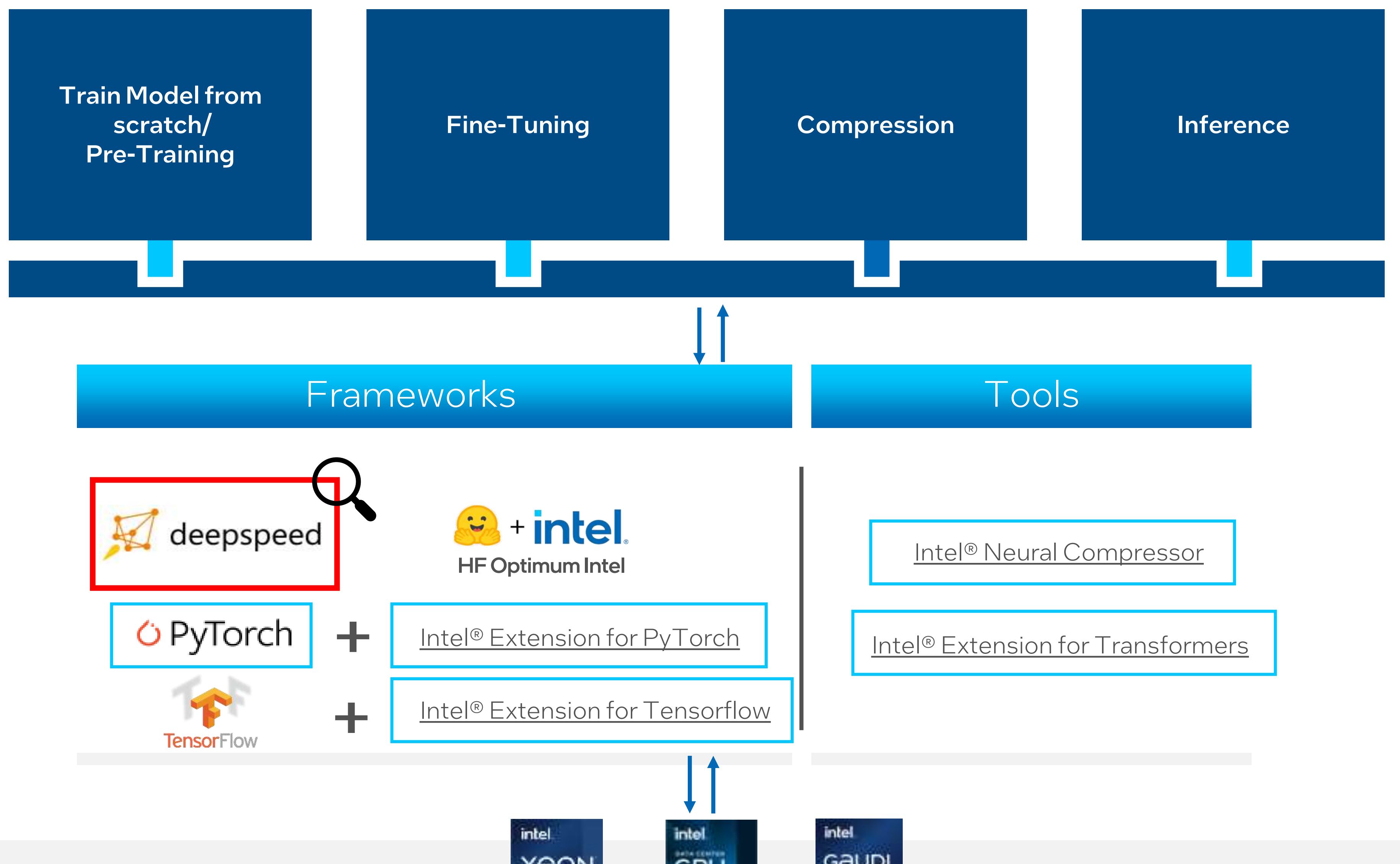
Type of GenAI & LLM Models			
Stable Diffusion	LLAMA3	Baichuan2-13B	OPT
BLOOM-176B	LLAMA2	GPT-NEOX	Dolly-v2-3B
Qwen-7B	LLAMA	MPT	Falcon
Qwen-14B	T5	FALCON	GPT-J-6B
ChatGLM2-6B	Flan-T5	BLOOM-7B	GPT-NEOX
ChatGLM4-6B	…		

Ref: https://github.com/intel/intel-extension-for-transformers

Intel Extension for Transformers

- Intel Extension for Transformers (ITREX): Built on top of INC ecosystem and Hugging Face
- Its target is the democratization of NLP and Transformers for both training/fine-tuning and inference
- Brings compression and model optimizations in a high-level HF – like API
- Staging area for all Intel's transformer feature enhancements:
  - Upstream to HF as much as possible (Transformers + Optimum)
  - Intel's differentiation remains, e.g., NAS, MoE, dynamic model, etc., and is ready for future upstream

GenAI Deep Learning Funnel Pipeline



Choose the Best Accelerated Technology

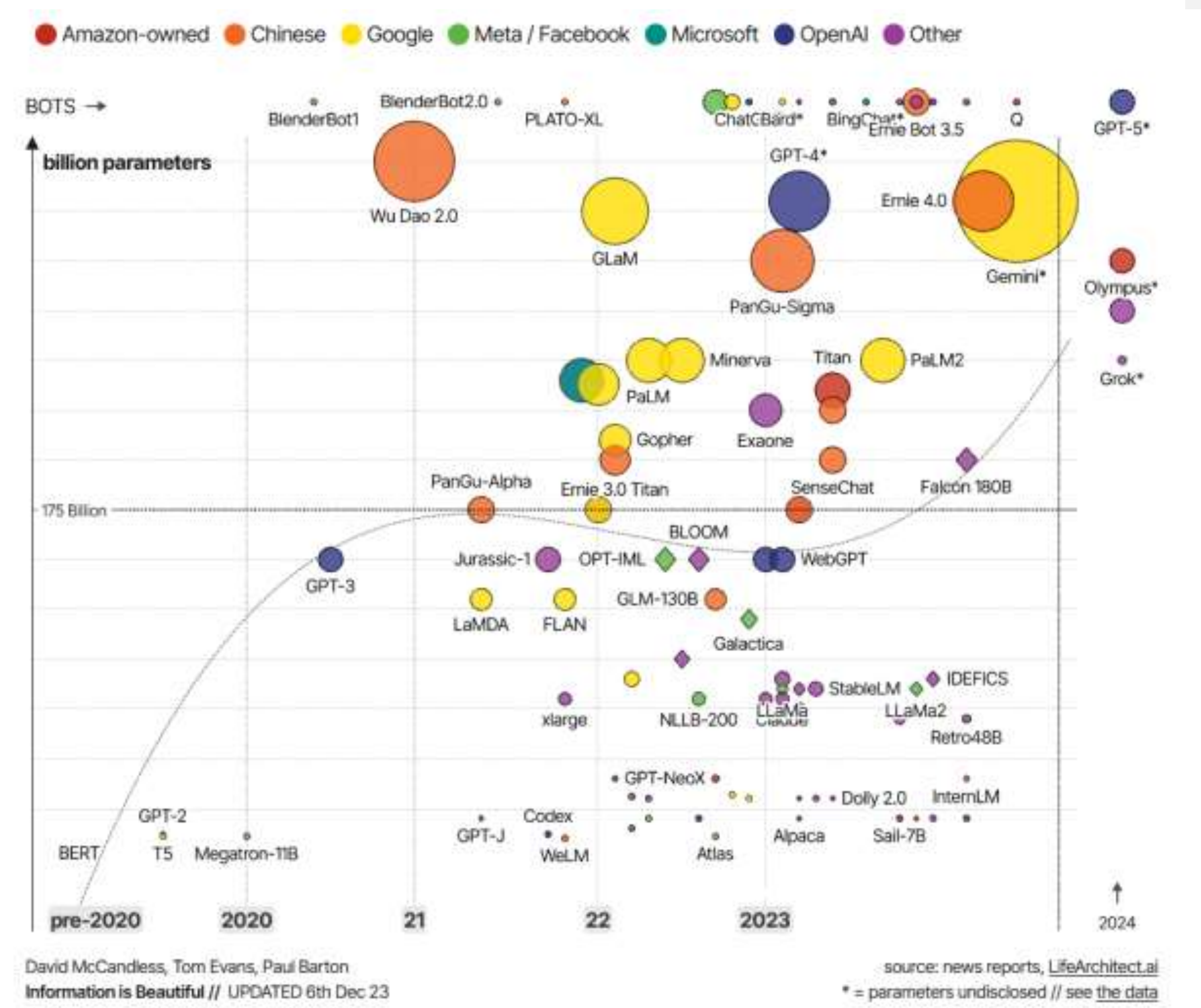
Distributed Training @ Intel Architecture

Akash Dhamasia – AI Software Solutions Engineer
akash.dhamasia@intel.com
July 22nd 2024

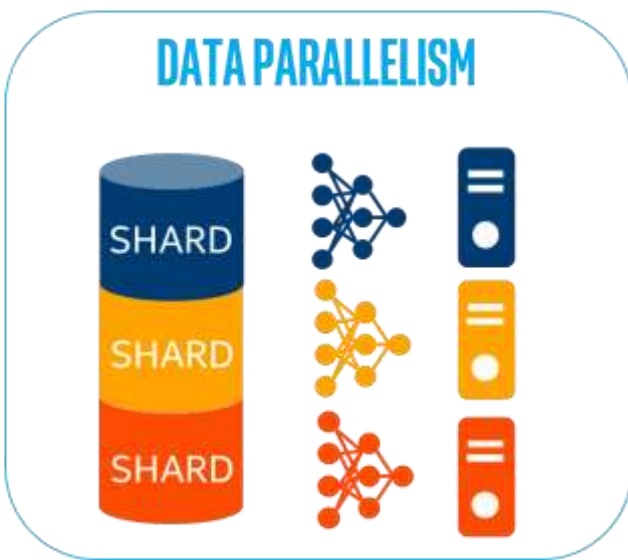
All information provided in this deck is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Why Distributed Training?

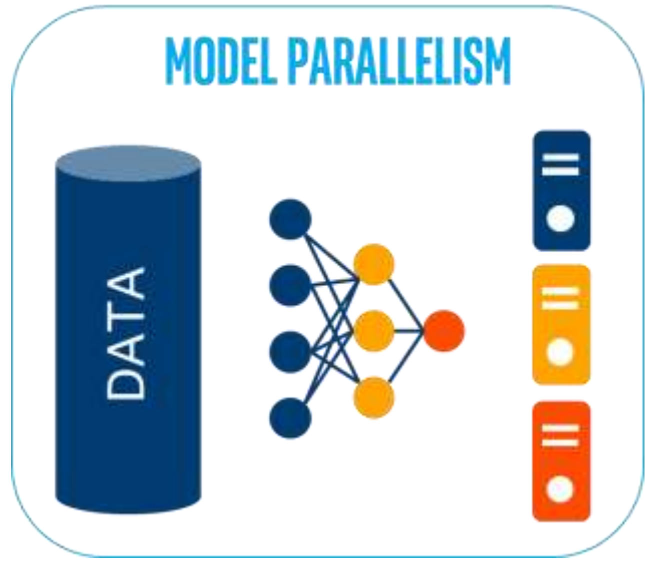
- Increases the amount of compute
- Helps train model faster.
- with Increasing Model size & Dataset size, it makes sense to divide them to do computation parallelly and faster, Also not possible to fit big model on single GPU.



Neural Network Parallelism



Data is processed in increments of N. Work on minibatch samples and distributed among the available resources.



The work is divided according to a split of the model. The sample minibatch is copied to all processors which compute part of the DNN.

- Distributed Training Methods
  - Data Parallel
  - Model Parallel
  - Data + Model Parallel
- Types of Multi-worker communication
- NCCL
- MPI
- CCL

source: https://arxiv.org/pdf/1802.09941.pdf

Intel® oneAPI Collective Communications Library (oneCCL)

- enables developers and researchers to quickly train DL models
- optimizes communication patterns to distribute model training across multiple nodes
- designed for easy integration into deep learning frameworks, whether they are implemented them from scratch or customizing existing ones
- DistributedDataParallel (DDP) with Intel® oneCCL
  - E.g mpirun -n 2 -l python Example_DDP.py
  - Important links:
    - https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/features/DDP.html
    - https://github.com/oneapi-src/oneCCL
- Horovod with Intel® oneCCL & PyTorch
  - E.g horovodrun -np 2 python Example_horovod.py
  - Or e.g mpirun -np 2 python Example_horovod.py
  - Important links:
    - https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/features/horovod.html
    - https://github.com/intel/intel-optimization-for-horovod
- Fully Sharded Data Parallel (FSDP)
- DeepSpeed
  - Deep learning optimization software suite that enables scale and speed for Deep Learning Training and inference of models with billions or trillions of parameters
  - Example to train GPT 3.6B, 20B, 175B
    - https://github.com/intel/intel-extension-for-deepspeed/tree/main/examples

DistributedDataParallel (DDP)

- DDP is a PyTorch module that implements multi-process data parallelism across multiple GPUs and machines.
- With DDP, the model is replicated on every process, and each model replica is fed a different set of input data samples.
- **To run DDP optimized for Intel hardware, we use Intel® oneCCL Bindings for Pytorch***
- Important links:
  - https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/features/DDP.html
  - https://github.com/oneapi-src/oneCCL

```python
import os
import torch
import torch.distributed as dist
import torchvision
import oneccl_bindings_for_pytorch as torch_ccl
import intel_extension_for_pytorch as ipex

LR = 0.001
DOWNLOAD = True
DATA = 'datasets/cifar10/'

transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224, 224)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
train_dataset = torchvision.datasets.CIFAR10(
        root=DATA,
        train=True,
        transform=transform,
        download=DOWNLOAD,
)
train_loader = torch.utils.data.DataLoader(
        dataset=train_dataset,
        batch_size=128
)

os.environ['MASTER_ADDR'] = '127.0.0.1'
os.environ['MASTER_PORT'] = '29500'
os.environ['RANK'] = os.environ.get('PMI_RANK', 0)
os.environ['WORLD_SIZE'] = os.environ.get('PMI_SIZE', 1)
dist.init_process_group(
backend='ccl',
init_method='env://'
)

model = torchvision.models.resnet50()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = LR, momentum=0.9)
model.train()
model, optimizer = ipex.optimize(model, optimizer=optimizer)

model = torch.nn.parallel.DistributedDataParallel(model)

for batch_idx, (data, target) in enumerate(train_loader):
    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    print('batch_id: {}'.format(batch_idx))
torch.save({
     'model_state_dict': model.state_dict(),
     'optimizer_state_dict': optimizer.state_dict(),
     }, 'checkpoint.pth')
```

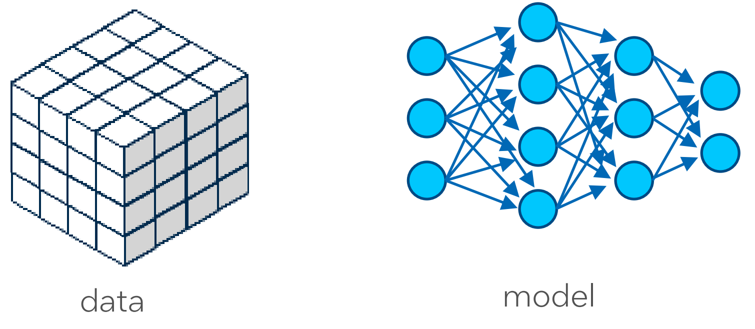
sdp@4pvc-gpu: ~/distributed_ipex

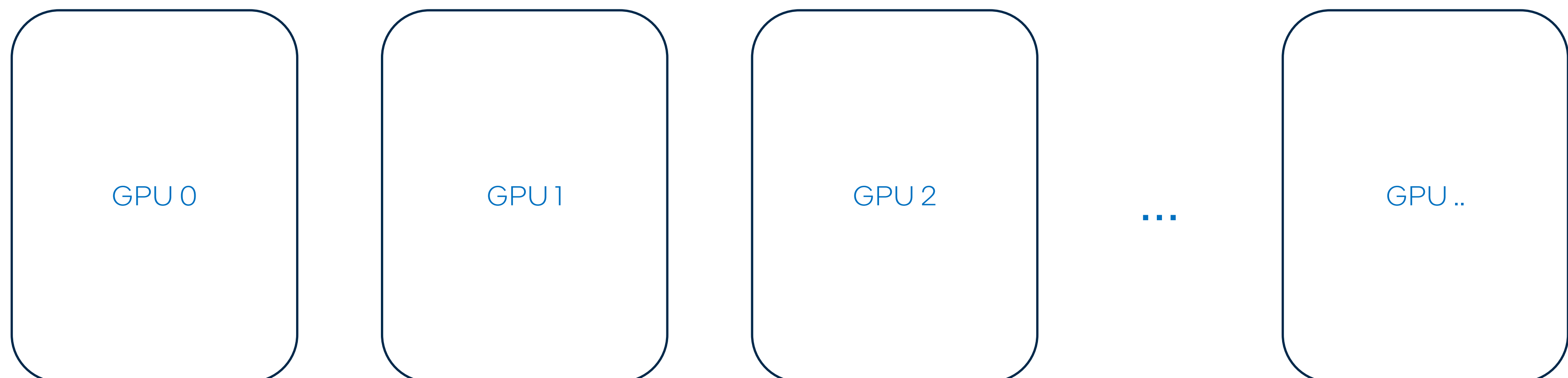
```
Epoch: [0][   51/10010] Time  0.108 ( 0.226)    Data  0.009 ( 0.016)    Loss 7.0634e+00 (7.0062e+00)
Epoch: [0][   61/10010] Time  0.108 ( 0.207)    Data  0.009 ( 0.015)    Loss 6.9980e+00 (7.0021e+00)
Epoch: [0][   71/10010] Time  0.109 ( 0.193)    Data  0.009 ( 0.014)    Loss 7.0117e+00 (6.9989e+00)
Epoch: [0][   81/10010] Time  0.108 ( 0.182)    Data  0.009 ( 0.014)    Loss 7.0338e+00 (6.9965e+00)
Epoch: [0][   91/10010] Time  0.108 ( 0.174)    Data  0.009 ( 0.013)    Loss 6.9748e+00 (6.9943e+00)
Epoch: [0][  101/10010] Time  0.108 ( 0.168)    Data  0.010 ( 0.013)    Loss 6.9530e+00 (6.9917e+00)
Epoch: [0][  111/10010] Time  0.108 ( 0.162)    Data  0.009 ( 0.012)    Loss 6.9941e+00 (6.9897e+00)
```

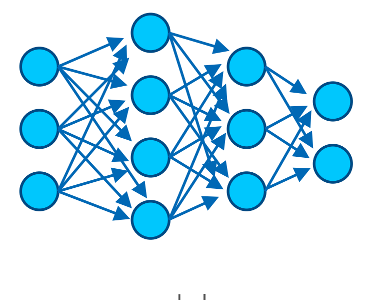
sdp@4pvc-gpu: ~

```
05:59:53.000,    0, 90.88, 289.64,   775
05:59:53.000,    1, 0.00, 37.34,     0
05:59:53.000,    2, 0.00, 36.63,     0
05:59:53.000,    3, 0.00, 29.39,     0
```

Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

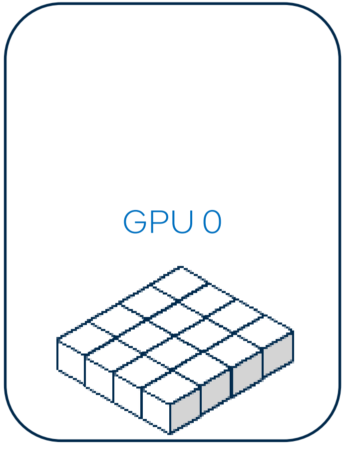

data
model

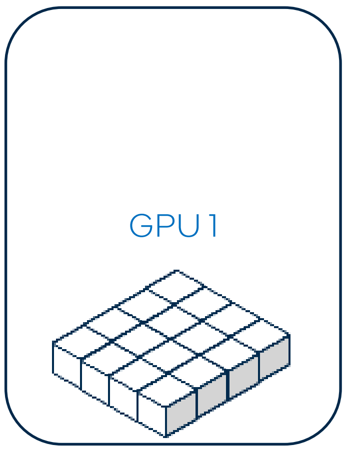

GPU 0
GPU 1
GPU 2
...
GPU ..

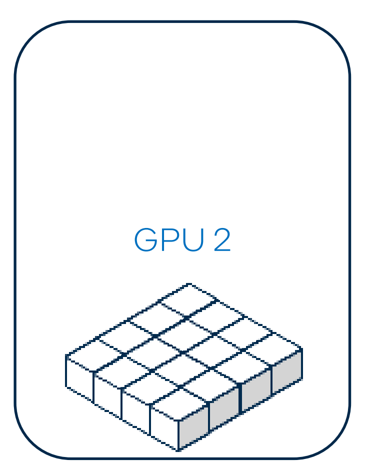


data

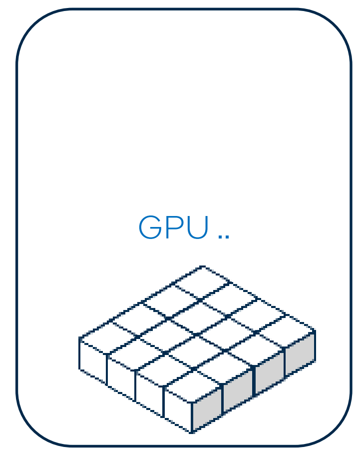
model



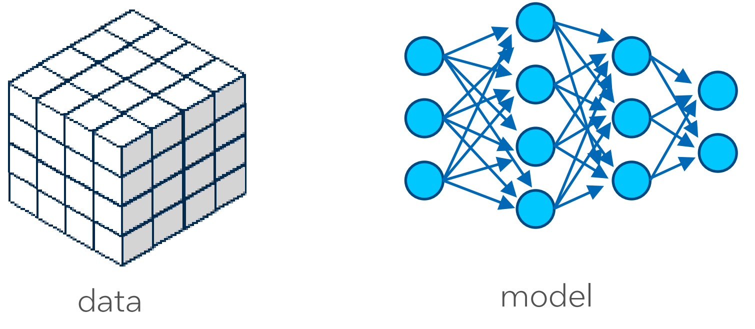




...



Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg



GPU 0

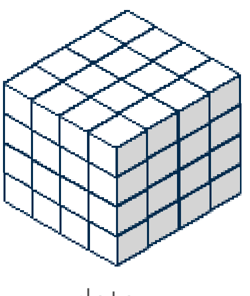
GPU 1

GPU 2

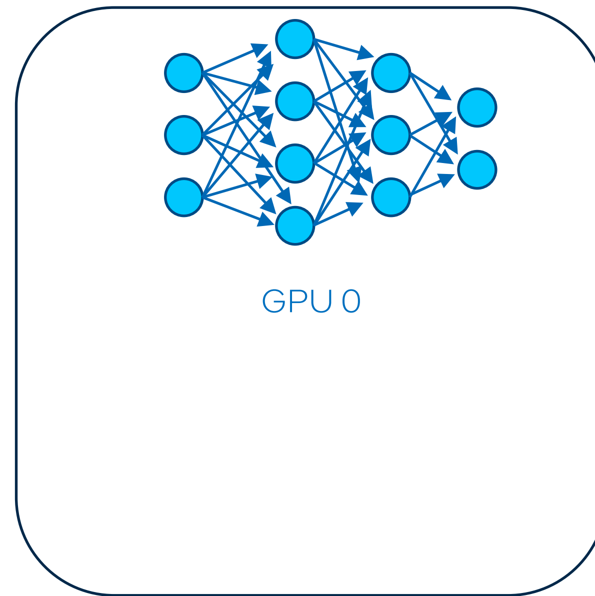
...

GPU ..

Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg


data

model



Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

data

model

GPU 0

Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

data

model

GPU 0

Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

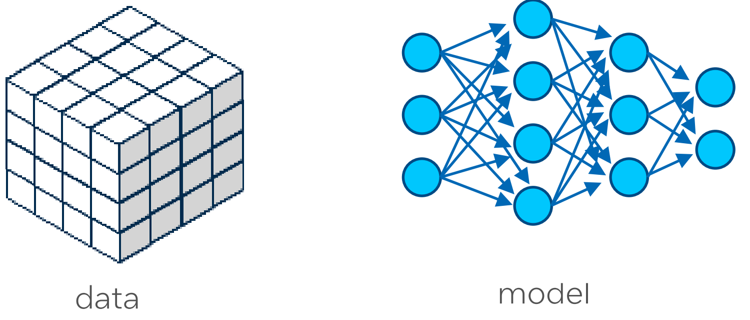
data

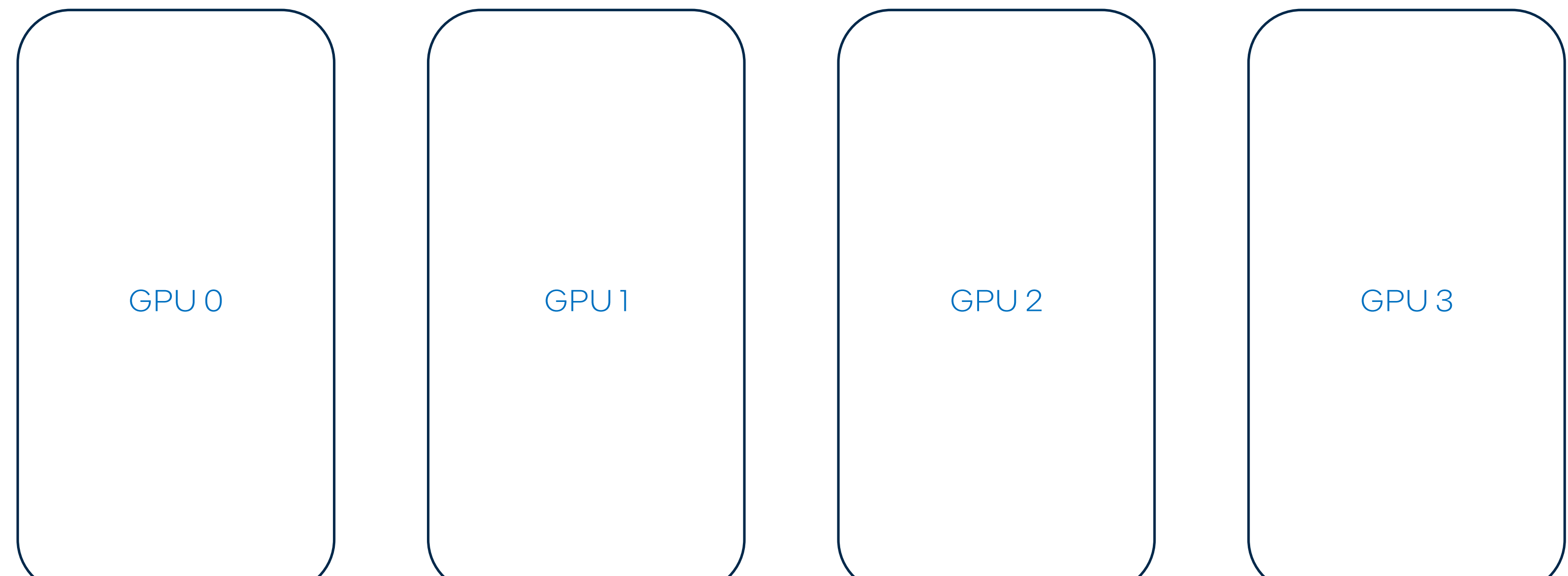
model

GPU 0

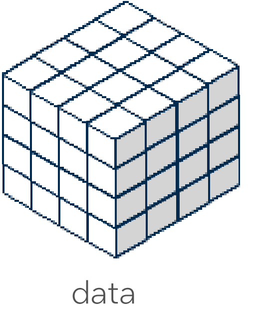
optimizer

Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

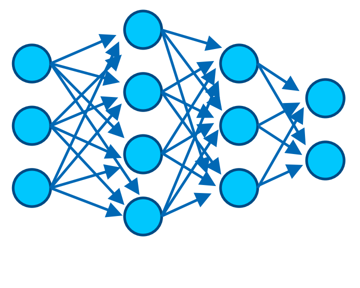




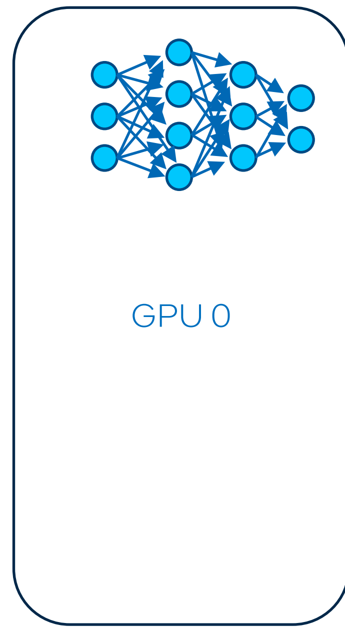
Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

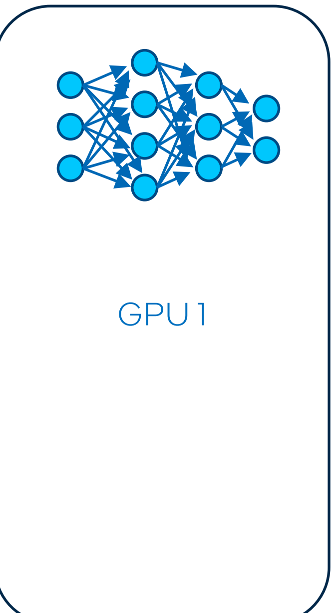


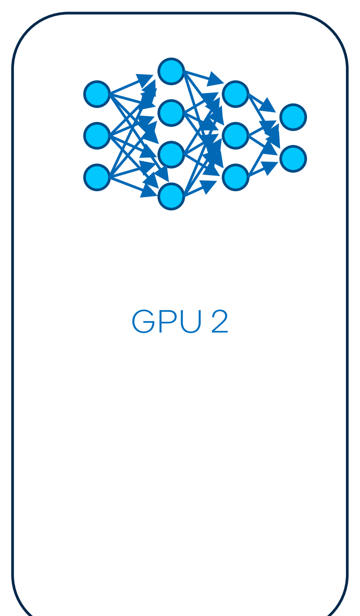
data

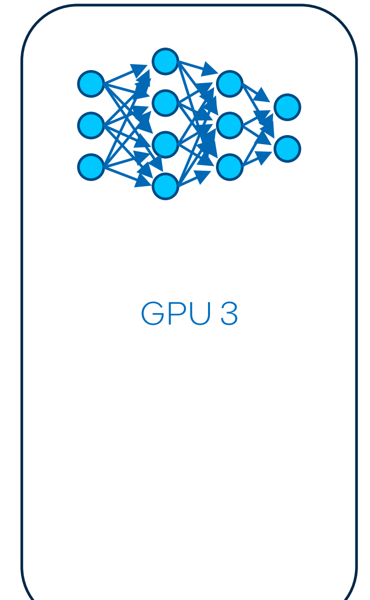


model



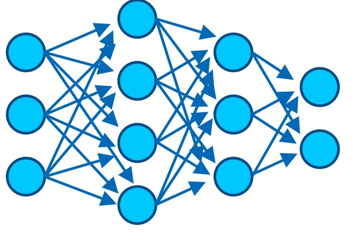






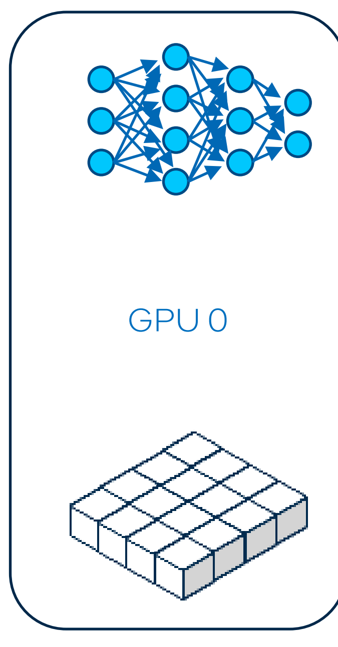
Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

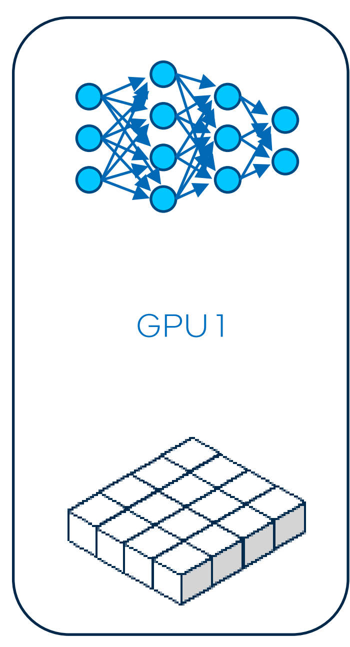
```
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
```

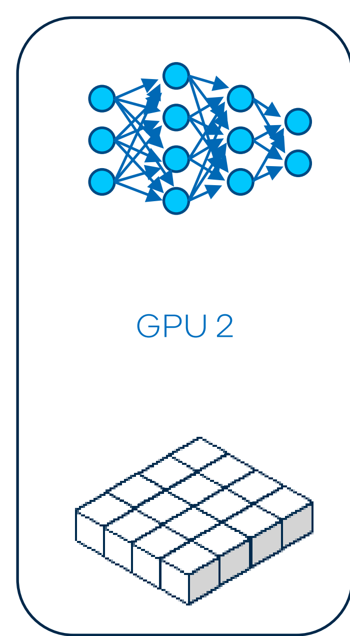


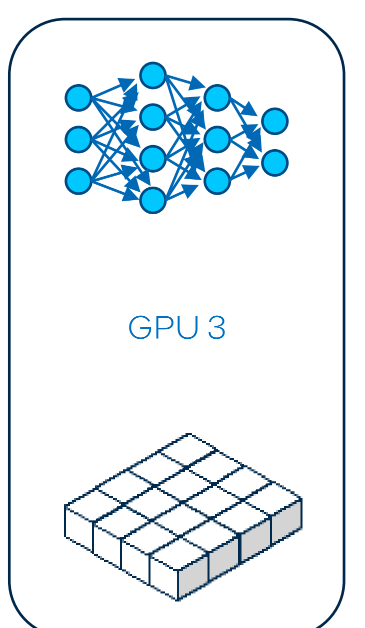
data

model

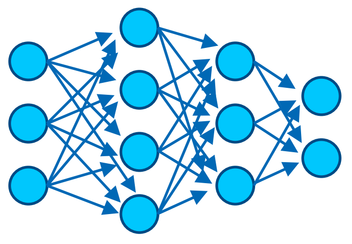








Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg



data

model

GPU 0

GPU 1

GPU 2

GPU 3

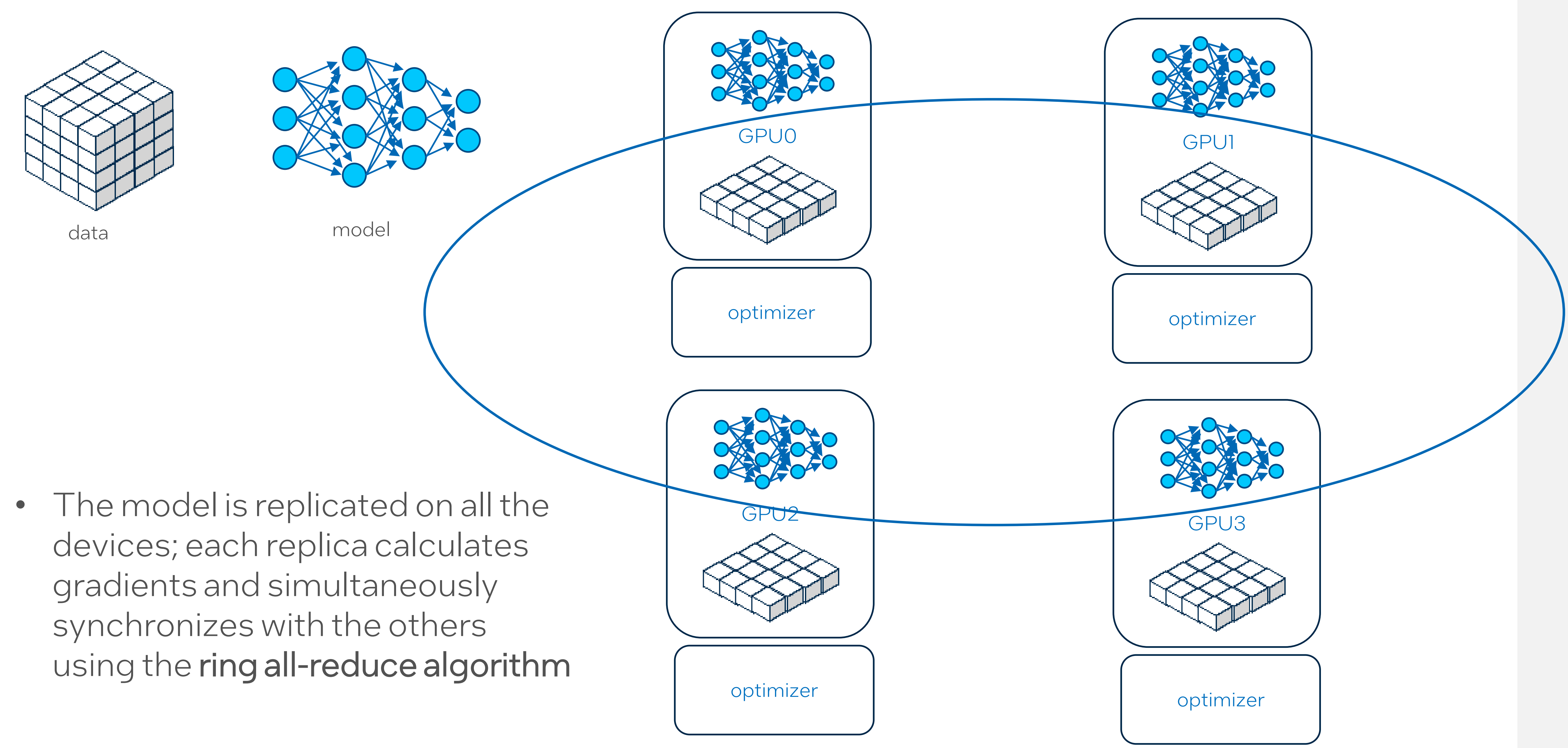
optimizer

optimizer

optimizer

optimizer

Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg



- The model is replicated on all the devices; each replica calculates gradients and simultaneously synchronizes with the others using the **ring all-reduce algorithm**

Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

GPU0

optimizer

GPU1

optimizer

GPU2

optimizer

GPU3

optimizer

oneCCL

GPU0

optimizer

GPU1

optimizer

GPU2

optimizer

GPU3

optimizer

oneCCL
Bindings for
PyTorch

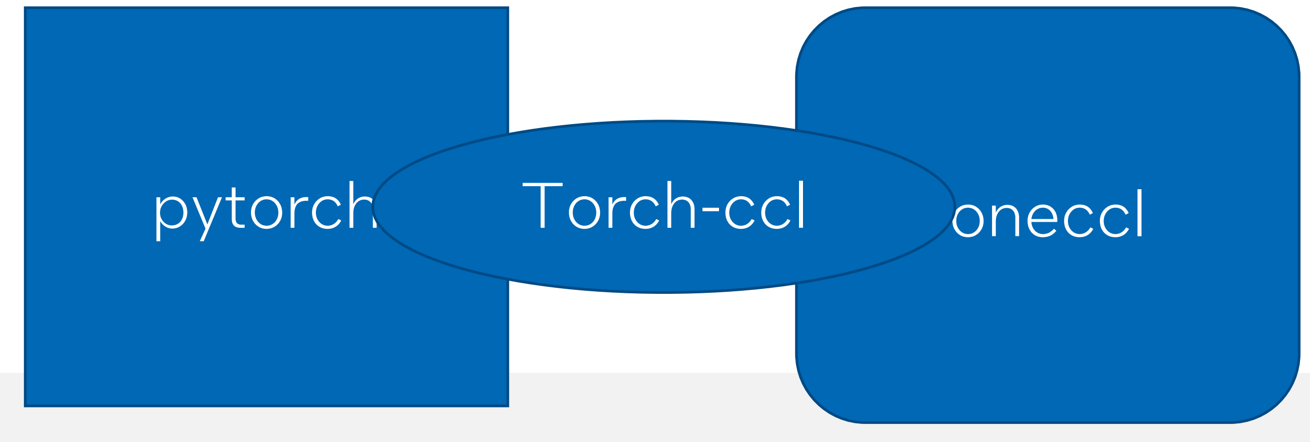
Credits: https://www.youtube.com/watch?v=3A8AVsNNHOg

Intel® oneCCL Bindings for Pytorch(Torch-CCL)

- Holds PyTorch bindings for the Intel® oneAPI Collective Communications Library (oneCCL).

- Github repository maintained by Intel
  - https://github.com/intel/torch-ccl

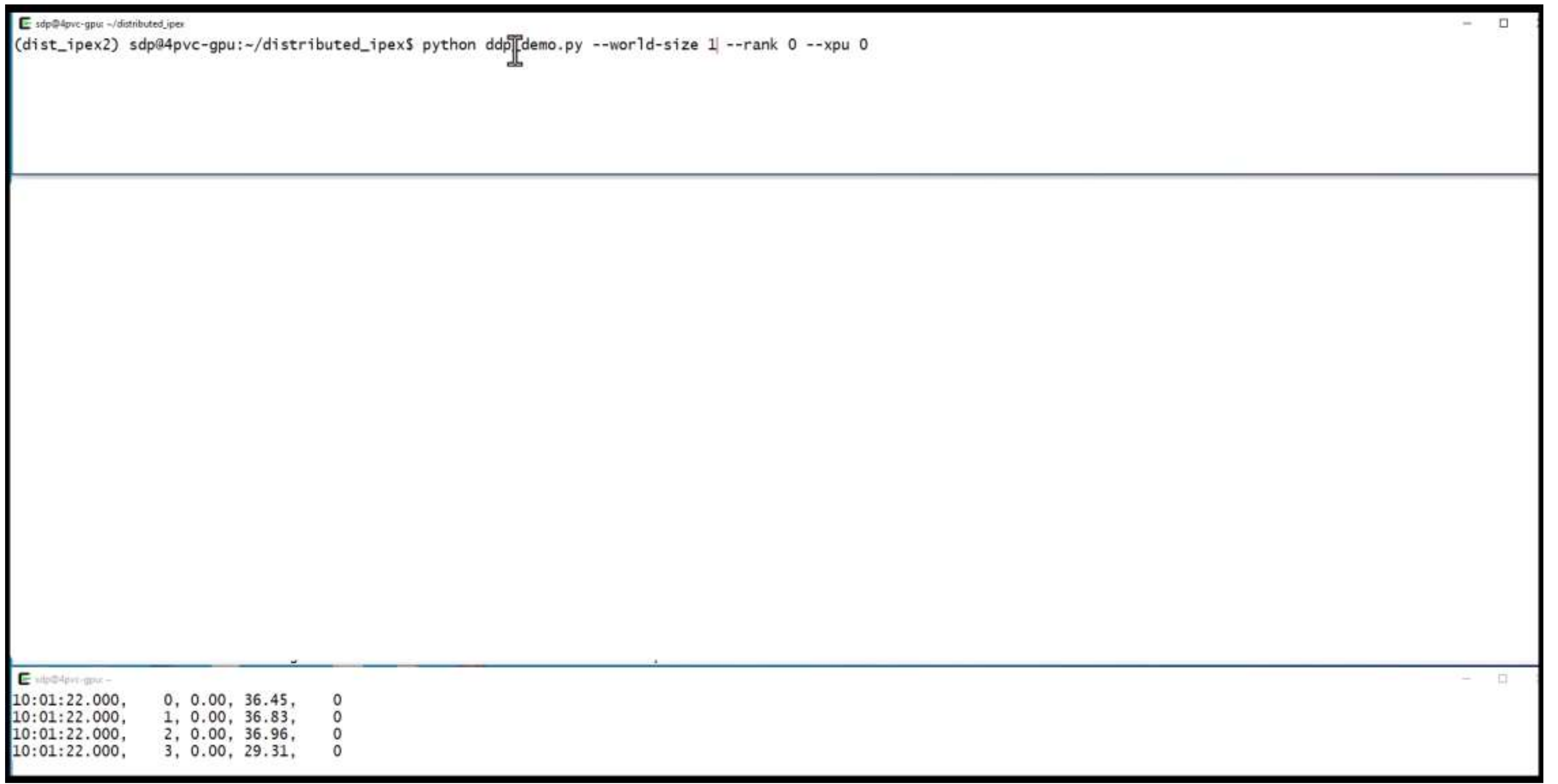
- Can be easily installed through prebuilt wheel:
  - python -m pip install oneccl_bind_pt --extra-index-url https://pytorch-extension.intel.com/release-whl/stable/xpu/us/



```python
87  def main_worker(ngpus_per_node, args):
88      # rank, local_rank setup
89      if args.distributed:
90          if args.rank == -1:
91              args.rank = int(os.environ["RANK"])
92          if args.multiprocessing_distributed:
93              # For multiprocessing distributed training, rank needs to be the
94              # global rank among all the processes
95              args.rank = args.rank * ngpus_per_node + args.xpu
96          init_method = 'tcp://' + args.dist_url + ':' + args.dist_port
97          dist.init_process_group(backend='ccl', init_method=init_method,
98                                  world_size=args.world_size, rank=args.rank)
99
```

```python
118     if args.distributed:
119         print("Generating DDP model for {}".format(args.xpu))
120         model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.xpu])
```

```
124     train_dataset = datasets.FakeData(1281167, (3, 224, 224), 1000, transforms.ToTensor())
125
126     train_sampler = None
127     if args.distributed:
128         train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
```


sdp@4pvc-gpu: ~/distributed_ipex
(dist_ipex2) sdp@4pvc-gpu:~/distributed_ipex$ python ddp_demo.py --world-size 1 --rank 0 --xpu 0
10:01:22.000, 0, 0.00, 36.45, 0
10:01:22.000, 1, 0.00, 36.83, 0
10:01:22.000, 2, 0.00, 36.96, 0
10:01:22.000, 3, 0.00, 29.31, 0

Initialization Function of DistributedDataParallel

- TCP initialization
  - IP address and port of rank 0 node is required.
  - init_method='tcp://10.1.1.20:23456'
- Shared file-system initialization
  - makes use of a file system that is shared and visible from all machines in a group.
  - init_method='file:///mnt/nfs/sharedfile'

- Environment variable initialization
  - Default method
  - init_method='env://'
  - MASTER_PORT - required; has to be a free port on machine with rank 0
  - MASTER_ADDR - required (except for rank 0); address of rank 0 node
  - WORLD_SIZE - required; can be set either here, or in a call to init function
  - RANK - required; can be set either here, or in a call to init function

Quick DDP Recipe @Intel®

Only 3-5 changes needed from general torch DDP code

1. import torch_ccl & DDP

2. Access PMI_* environment variables

3. Set backend to ‘ccl’

4. Use Distributed dataset sampler

e.g import torch.utils.data.distributed train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)

5. Pass model to DDP

https://github.com/intel/optimized-models/tree/master/pytorch/distributed

```python
import os
import torch
import torch.distributed as dist
import torchvision
import oneccl_bindings_for_pytorch as torch_ccl
import intel_extension_for_pytorch as ipex

LR = 0.001
DOWNLOAD = True
DATA = 'datasets/cifar10/'

transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224, 224)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
train_dataset = torchvision.datasets.CIFAR10(
        root=DATA,
        train=True,
        transform=transform,
        download=DOWNLOAD,
)
train_loader = torch.utils.data.DataLoader(
        dataset=train_dataset,
        batch_size=128
)

os.environ['MASTER_ADDR'] = '127.0.0.1'
os.environ['MASTER_PORT'] = '29500'
os.environ['RANK'] = os.environ.get('PMI_RANK', 0)
os.environ['WORLD_SIZE'] = os.environ.get('PMI_SIZE', 1)
dist.init_process_group(
backend='ccl',
init_method='env://'
)

model = torchvision.models.resnet50()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = LR, momentum=0.9)
model.train()
model, optimizer = ipex.optimize(model, optimizer=optimizer)

model = torch.nn.parallel.DistributedDataParallel(model)

for batch_idx, (data, target) in enumerate(train_loader):
    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    print('batch_id: {}'.format(batch_idx))
torch.save({
     'model_state_dict': model.state_dict(),
     'optimizer_state_dict': optimizer.state_dict(),
     }, 'checkpoint.pth')
```

Usage for Distributed Training with DDP

- 4 root devices, 4 GPUs
- 8 ranks and two ranks per GPU
- E.g mpirun -n 8 -l python Example_DDP.py

```
(base) ac.louie.tsai@florentia05:~> sycl-ls
Warning: SYCL_DEVICE_FILTER environment variable is set to level_zero.
To see the correct device id, please unset SYCL_DEVICE_FILTER.

[ext_oneapi_level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) Graphics [0x0bd5] 1.3 [1.3.23937]
[ext_oneapi_level_zero:gpu:1] Intel(R) Level-Zero, Intel(R) Graphics [0x0bd5] 1.3 [1.3.23937]
[ext_oneapi_level_zero:gpu:2] Intel(R) Level-Zero, Intel(R) Graphics [0x0bd5] 1.3 [1.3.23937]
[ext_oneapi_level_zero:gpu:3] Intel(R) Level-Zero, Intel(R) Graphics [0x0bd5] 1.3 [1.3.23937]
```

- Monitor XPU usage using Intel® XPU manager:
  - https://www.intel.com/content/www/us/en/software/xpu
- xpumcli dump –d 0 –m 0,1,2,3,4,5

```
Timestamp, DeviceId, GPU Utilization (%), GPU Power (W), GPU Frequency (MHz), GPU Core Temperature
(Celsius Degree), GPU Memory Temperature (Celsius Degree), GPU Energy Consumed (J)

08:04:16.000,   0, 53.35, 234.08, 0.00,    ,    , 2018647.97
08:04:17.000,   0, 65.83, 341.15, 1600.00,    ,    , 2018956.02
08:04:18.000,   0, 92.52, 375.21, 900.00,    ,    , 2019332.25
08:04:19.000,   0, 92.54, 384.55, 1500.00,    ,    , 2019715.47
08:04:20.000,   0, 94.21, 387.95, 975.00,    ,    , 2020105.06
08:04:21.000,   0, 93.25, 386.10, 1600.00,    ,    , 2020491.66
08:04:22.000,   0, 94.21, 391.84, 800.00,    ,    , 2020881.66
```

```
[6] ZE_AFFINITY_MASK=0,1,2,3
[6] Iterations: 5. Warmup runs: 2
[6] Running on device: IntelGPU6
[6] Running on torch: 1.10.0a0+git90332b4
[6] ModelType: resnet50, Kernels: DPCPP Input shape: 16x3x224x224
[6] Converting model to DDP & syncing...
[6] Starting warmup runs...
[6] Starting benchmark runs...
[6]       total:    459.63ms (458.7-460.4) +-1.15, 34.81 (imgs/s)
[6] csv,resnet50,16,0,34.81,1.10.0a0+git90332b4,IntelGPU6,2,5
[0] ZE_AFFINITY_MASK=0,1,2,3
[0] Iterations: 5. Warmup runs: 2
[0] Running on device: IntelGPU0
[0] Running on torch: 1.10.0a0+git90332b4
[0] ModelType: resnet50, Kernels: DPCPP Input shape: 16x3x224x224
[0] Converting model to DDP & syncing...
[0] Starting warmup runs...
[0] Starting benchmark runs...
[0]       total:    459.24ms (458.3-460.4) +-1.49, 34.84 (imgs/s)
[0] csv,resnet50,16,0,34.84,1.10.0a0+git90332b4,IntelGPU0,2,5
[0] Total img/sec on 8 IntelGPU(s): 278.7213323030451З
[3] ZE_AFFINITY_MASK=0,1,2,3
[3] Iterations: 5. Warmup runs: 2
[3] Running on device: IntelGPU3
[3] Running on torch: 1.10.0a0+git90332b4
[3] ModelType: resnet50, Kernels: DPCPP Input shape: 16x3x224x224
[3] Converting model to DDP & syncing...
[3] Starting warmup runs...
[3] Starting benchmark runs...
[3]       total:    458.77ms (457.9-459.7) +-1.42, 34.88 (imgs/s)
[3] csv,resnet50,16,0,34.88,1.10.0a0+git90332b4,IntelGPU3,2,5
[4] ZE_AFFINITY_MASK=0,1,2,3
[4] Iterations: 5. Warmup runs: 2
[4] Running on device: IntelGPU4
[4] Running on torch: 1.10.0a0+git90332b4
[4] ModelType: resnet50, Kernels: DPCPP Input shape: 16x3x224x224
[4] Converting model to DDP & syncing...
[4] Starting warmup runs...
[4] Starting benchmark runs...
[4]       total:    459.09ms (458.4-460.0) +-1.22, 34.85 (imgs/s)
[4] csv,resnet50,16,0,34.85,1.10.0a0+git90332b4,IntelGPU4,2,5
[2] ZE_AFFINITY_MASK=0,1,2,3
[2] Iterations: 5. Warmup runs: 2
[2] Running on device: IntelGPU2
[2] Running on torch: 1.10.0a0+git90332b4
[2] ModelType: resnet50, Kernels: DPCPP Input shape: 16x3x224x224
[2] Converting model to DDP & syncing...
[2] Starting warmup runs...
[2] Starting benchmark runs...
[2]       total:    459.20ms (458.7-460.1) +-1.00, 34.84 (imgs/s)
[2] csv,resnet50,16,0,34.84,1.10.0a0+git90332b4,IntelGPU2,2,5
[1] ZE_AFFINITY_MASK=0,1,2,3
[1] Iterations: 5. Warmup runs: 2
[1] Running on device: IntelGPU1
[1] Running on torch: 1.10.0a0+git90332b4
[1] ModelType: resnet50, Kernels: DPCPP Input shape: 16x3x224x224
[1] Converting model to DDP & syncing...
[1] Starting warmup runs...
[1] Starting benchmark runs...
[1]       total:    459.00ms (458.1-460.4) +-1.91, 34.86 (imgs/s)
[1] csv,resnet50,16,0,34.86,1.10.0a0+git90332b4,IntelGPU1,2,5
[5] ZE_AFFINITY_MASK=0,1,2,3
[5] Iterations: 5. Warmup runs: 2
[5] Running on device: IntelGPU5
[5] Running on torch: 1.10.0a0+git90332b4
[5] ModelType: resnet50, Kernels: DPCPP Input shape: 16x3x224x224
[5] Converting model to DDP & syncing...
[5] Starting warmup runs...
[5] Starting benchmark runs...
[5]       total:    458.69ms (457.9-460.5) +-1.88, 34.88 (imgs/s)
[5] csv,resnet50,16,0,34.88,1.10.0a0+git90332b4,IntelGPU5,2,5
[7] ZE_AFFINITY_MASK=0,1,2,3
[7] Iterations: 5. Warmup runs: 2
[7] Running on device: IntelGPU7
[7] Running on torch: 1.10.0a0+git90332b4
```

Horovod

- Horovod is a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet.
- Horovod can be easily installed through:
  - python -m pip install intel-optimization-for-horovod
- Important links:
  - https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/features/horovod.html
  - https://intel.github.io/intel-extension-for-tensorflow/latest/examples/train_horovod/mnist/README.html?highlight=horovod
  - https://github.com/intel/intel-optimization-for-horovod
  - https://horovod.readthedocs.io/en/latest/oneccl_include.html#advanced-settings

```python
import torch
import intel_extension_for_pytorch
import horovod.torch as hvd

# Initialize Horovod
hvd.init()

# Pin GPU to be used to process local rank (one GPU per process)
devid = hvd.local_rank()
torch.xpu.set_device(devid)
device = "xpu:{}".format(devid)

# Define dataset...
train_dataset = ...

# Partition dataset among workers using DistributedSampler
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset, num_replicas=hvd.size(), rank=hvd.rank())

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., sampler=train_sampler)

# Build model...
model = ...
model.to(device)

optimizer = optim.SGD(model.parameters())

# Add Horovod Distributed Optimizer
optimizer = hvd.DistributedOptimizer(optimizer, named_parameters=model.named_parameters())

# Broadcast parameters from rank 0 to all other processes.
hvd.broadcast_parameters(model.state_dict(), root_rank=0)

for epoch in range(100):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{}]\tLoss: {}'.format(
                epoch, batch_idx * len(data), len(train_sampler), loss.item()))
```

Fully Sharded Data Parallel (FSDP)

- Fully Sharded Data Parallel (FSDP) is a PyTorch module that provides solution for large Model training.

- FSDP shards model parameters, optimizer states and gradients across DDP ranks to reduce the GPU memory footprint used in training, unlike DDP, where each process/worker maintains a replica of the model,

- Important links:
  - https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/features/FSDP.html
  - https://pytorch.org/tutorials/intermediate/FSDP_tutorial.html

<u>Some Additions on top of DDP:</u>

```
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP

model = FSDP(model, device_id="xpu:{}".format(rank)
```

DeepSpeed

DeepSpeed – Introduction

- Deep learning optimization software suite for PyTorch that enables scale and speed for Deep Learning training and inference

⇒ Train/inference models with billions or trillions of parameters

⇒ Efficiently scale to thousands of computing units

⇒ Train/inference on GPU system with limited GPU memory

⇒ Low latency and high throughput for inference

DeepSpeed – Inference

Tensor Parallelism

- The reason to run Transformer based model inference with DeepSpeed on multiple device is to get better inference latency through Tensor Parallelism
- Tensor Parallelism parallelize Tensor operations in LLMs between multiple workers, so each worker does less tensor operation; results in less inference latency time
- DeepSpeed offers Tensor Parallelism with two different technologies: AutoTP and Kernel Injection

Simple DeepSpeed Example (Inference)

```python
# deepspeed engine
ds_local_rank = int(os.getenv('LOCAL_RANK', '0'))
ds_world_size = int(os.getenv('WORLD_SIZE', '0'))
generate_kwargs['TP_number'] = ds_world_size
if args.dtype == 'int8':
    amp_enabled = False
    amp_dtype = torch.int8
if ds_world_size == 0:
    ds_world_size = 1
engine = deepspeed.init_inference(model=model, mp_size=ds_world_size, dtype=amp_dtype,
                                  replace_method="auto", replace_with_kernel_inject=replace_with_kernel_inject)
model = engine.module
get_memory_usage("DeepSpeed", args)
# to ipex
if args.ipex:
    import intel_extension_for_pytorch as ipex
    try: ipex._C.disable_jit_linear_repack()
    except: pass
    model = ipex.optimize(model.eval(), dtype=amp_dtype, inplace=True)
    get_memory_usage("IPEX", args)
```

DeepSpeed Inference API

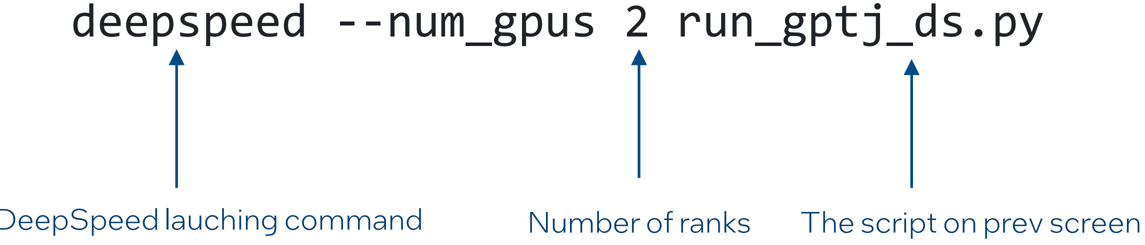
PyTorch model

Tensor Parallel size

DeepSpeed model

True: use kernel injection
False: use AutoTP

Simple DeepSpeed Example (Inference) – Single Node



deepspeed ~~--num_gpus 2~~ run_gptj_ds.py

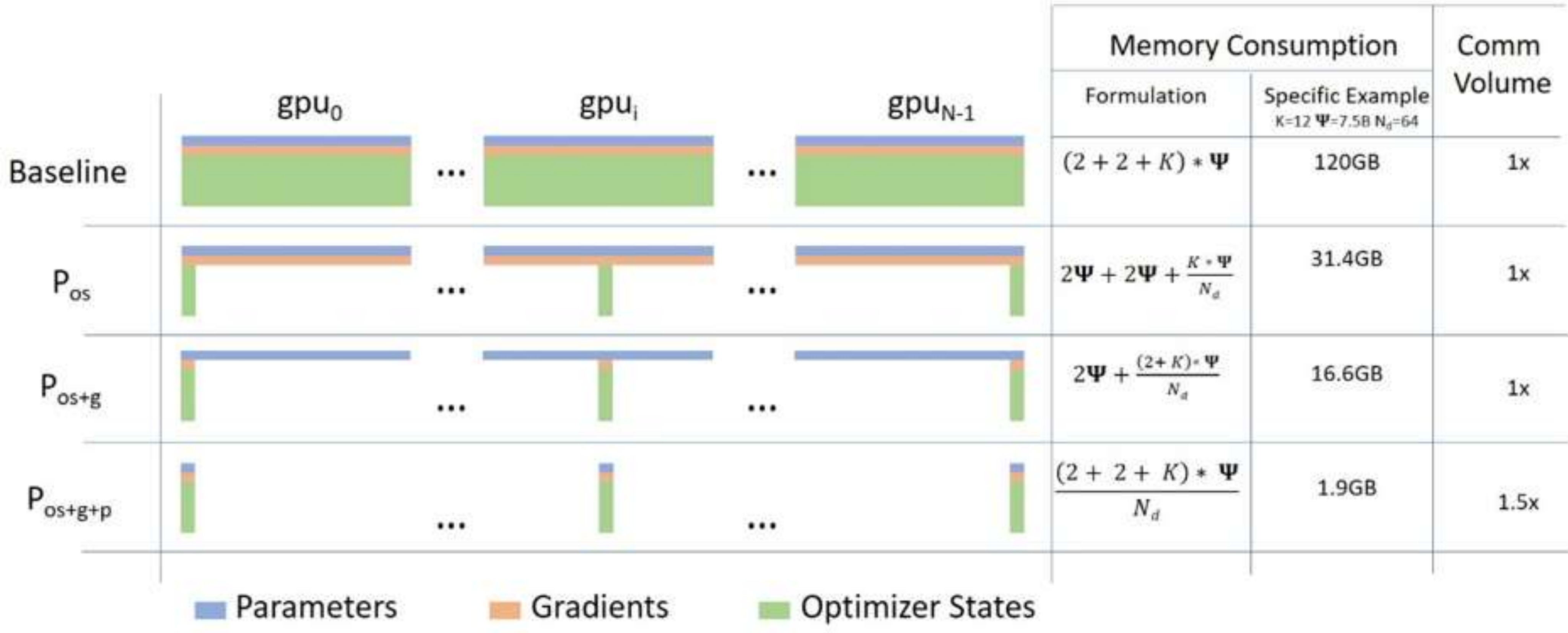
Auto detect
number of ranks

Simple DeepSpeed Example (Inference) – Summary

- A PyTorch model will be converted to a DeepSpeed model through DeepSpeed *init_inference()* interface
- Converted DeepSpeed model can be further optimized with framework optimizations, i.e., *ipex.optimize()*
- Framework optimization should not go before DeepSpeed *init_inference()*, otherwise DeepSpeed optimizations will be blocked (cannot recognize optimized model)
- DeepSpeed model is executed with *deepspeed* command, which would launch multiple workers with multiprocess launcher (single node) or mpich/impi launcher (multi node)

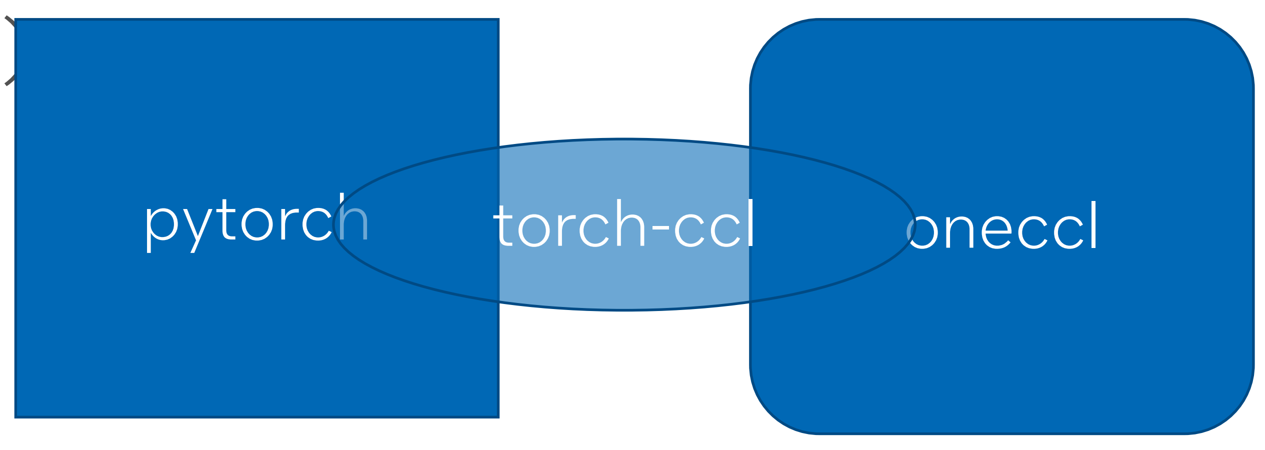
DeepSpeed – Training

DeepSpeed Training Technology – ZeRO Stage 1/2/3



DeepSpeed @ Intel

- PyTorch 2.1
- Intel Extension for PyTorch 2.1
- DeepSpeed
- Intel Extension for DeepSpeed / Intel Extension for PyTorch DeepSpeed
- oneCCL Bindings (torch-ccl)



- oneAPI 2024

Verified Models: Distributed

CPU:

MODEL FAMILY	MODEL NAME (Huggingface hub)	BF16	Weight only quantization INT8
LLAMA	meta-llama/Llama-2-7b-hf	🟩	🟩
LLAMA	meta-llama/Llama-2-13b-hf	🟩	🟩
LLAMA	meta-llama/Llama-2-70b-hf	🟩	🟩
LLAMA	meta-llama/Meta-Llama-3-8B	🟩	🟩
LLAMA	meta-llama/Meta-Llama-3-70B	🟩	🟩
GPT-J	EleutherAI/gpt-j-6b	🟧	🟩
GPT-NEOX	EleutherAI/gpt-neox-20b	🟧	🟩
DOLLY	databricks/dolly-v2-12b	🟧	🟩
FALCON	tiiuae/falcon-40b	🟩	🟩
OPT	facebook/opt-30b	🟧	🟩
OPT	facebook/opt-1.3b	🟩	🟩
Bloom	bigscience/bloom-1b7	🟧	🟩
CodeGen	Salesforce/codegen-2B-multi	🟩	🟩
Baichuan	baichuan-inc/Baichuan2-7B-Chat	🟩	🟩
Baichuan	baichuan-inc/Baichuan2-13B-Chat	🟩	🟩
Baichuan	baichuan-inc/Baichuan-13B-Chat	🟧	🟩
GPTBigCode	bigcode/starcoder	🟩	🟩
T5	google/flan-t5-xl	🟩	🟩
Mistral	mistralai/Mistral-7B-v0.1	🟩	🟩
Mistral	mistralai/Mixtral-8x7B-v0.1	🟩	🟩
MPT	mosaicml/mpt-7b	🟩	🟩
Stablelm	stabilityai/stablelm-2-1_6b	🟩	🟩
Qwen	Qwen/Qwen-7B-Chat	🟩	🟩
GIT	microsoft/git-base	🟩	🟩
Phi	microsoft/phi-2	🟩	🟩
Phi	microsoft/Phi-3-mini-4k-instruct	🟩	🟩
Phi	microsoft/Phi-3-mini-128k-instruct	🟩	🟩
Phi	microsoft/Phi-3-medium-4k-instruct	🟩	🟩
Phi	microsoft/Phi-3-medium-128k-instruct	🟩	🟩

GPU:

MODEL FAMILY	Verified < MODEL ID > (Huggingface hub)	FP16	Weight only quantization INT4	Optimized on Intel® Data Center GPU Max Series (1550/1100)	Optimized on Intel® Arc™ A-Series Graphics (A770)
Llama 2	"meta-llama/Llama-2-7b-hf", "meta-llama/Llama-2-13b-hf", "meta-llama/Llama-2-70b-hf"	✅	✅	✅	✅
GPT-J	"EleutherAI/gpt-j-6b"	✅	✅	✅	✅
Qwen	"Qwen/Qwen-7B"	✅	✅	✅	✅
OPT	"facebook/opt-6.7b", "facebook/opt-30b"	✅	❎	✅	❎
Bloom	"bigscience/bloom-7b1", "bigscience/bloom"	✅	❎	✅	❎
ChatGLM3-6B	"THUDM/chatglm3-6b"	✅	❎	✅	❎
Baichuan2-13B	"baichuan-inc/Baichuan2-13B-Chat"	✅	❎	✅	❎

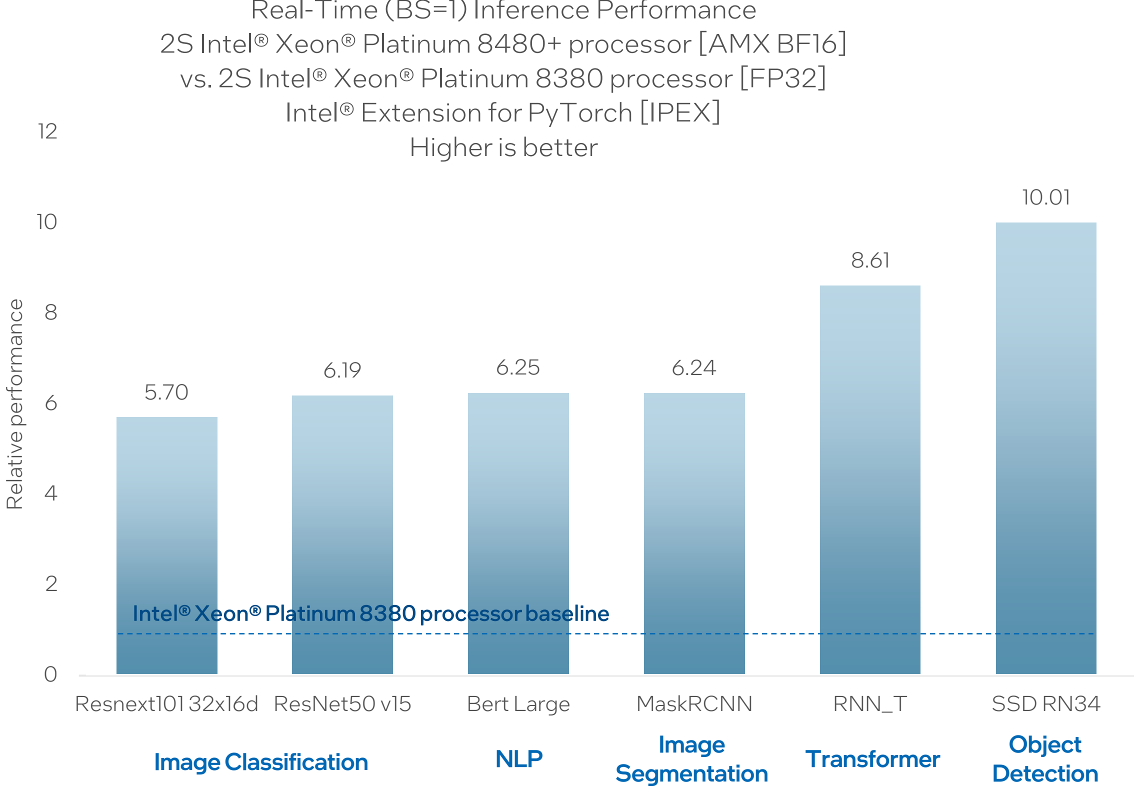
CPU - https://github.com/intel/intel-extension-for-pytorch/tree/v2.3.0%2Bcpu-rc0/examples/cpu/inference/python/llm

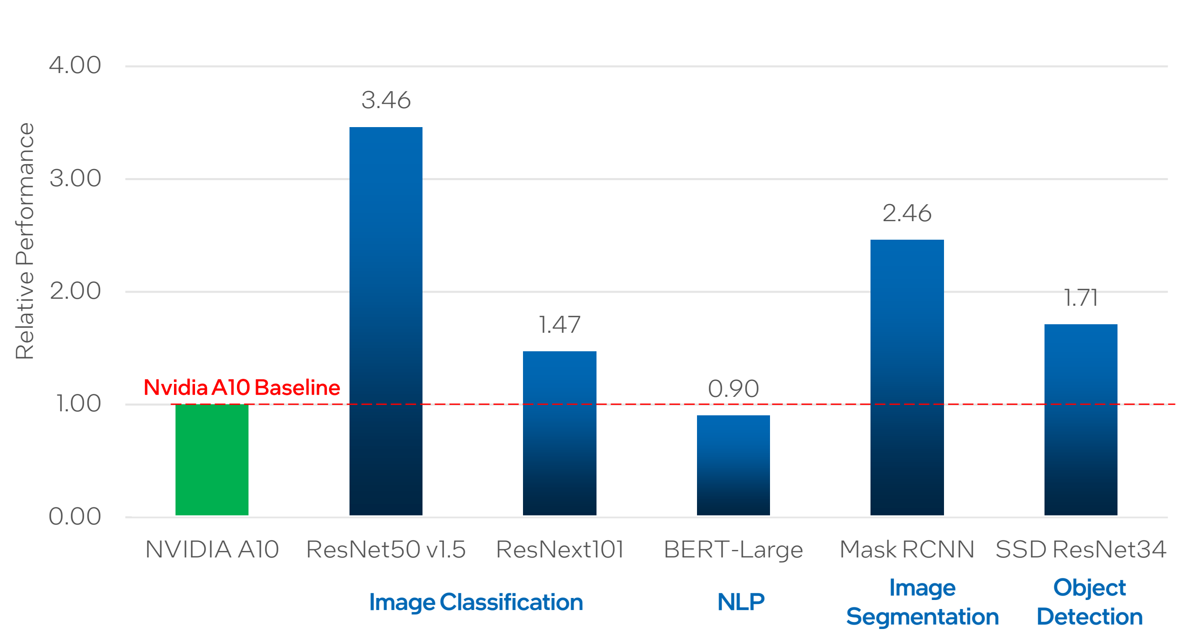
GPU - https://github.com/intel/intel-extension-for-pytorch/tree/xpu-main/examples/gpu/inference/python/llm

Performance

4th Gen Intel® Xeon® (SPR) & Intel® Data Center GPU Max 1550 (PVC)

Benchmarks: Inference Performance





Up to 10x higher gen-to-gen performance
Up to 7.7x higher gen-to-gen perf/watt[1]

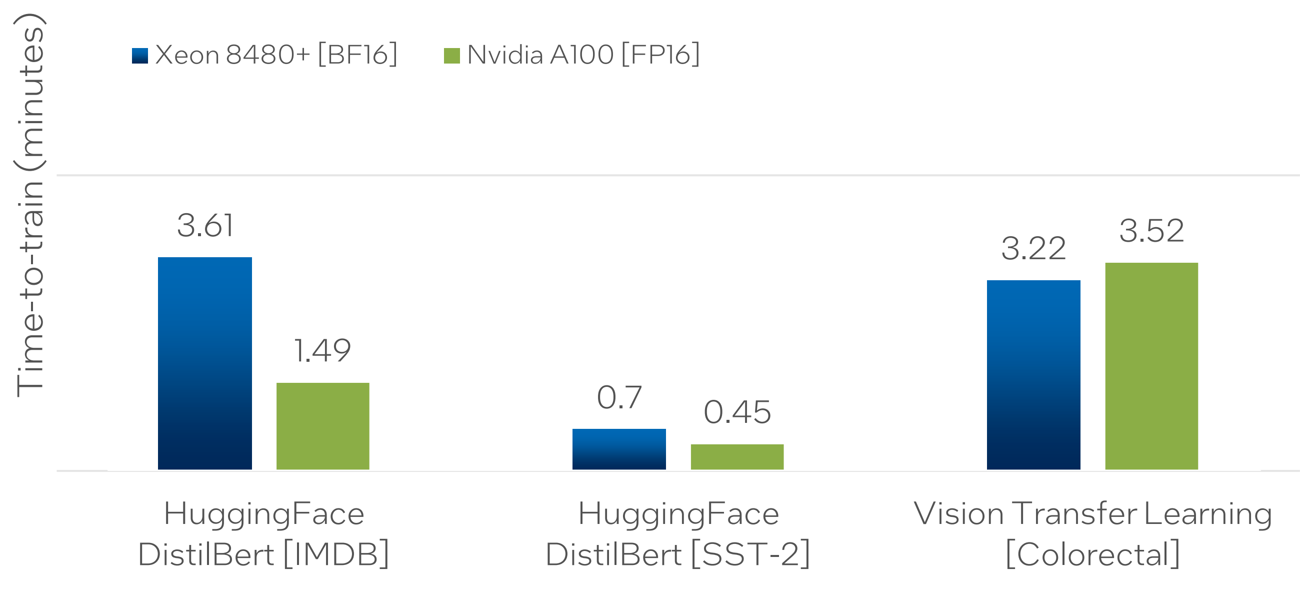
1.8x higher average* BF16/FP16 inference performance vs Nvidia A10 GPU[2]

1 See [A17, A33] at intel.com/processorclaims: 4th Gen Intel Xeon Scalable processors. Results may vary.
2 See [A218] at intel.com/processorclaims: 4th Gen Intel Xeon Scalable processors. Results may vary.

Real Workloads: Train With Fine Tuning in Less than 4 Minutes

Fine tuning time-to-train performance
Intel® Xeon® Platinum 8480+ processor
vs. Nvidia A100 GPU

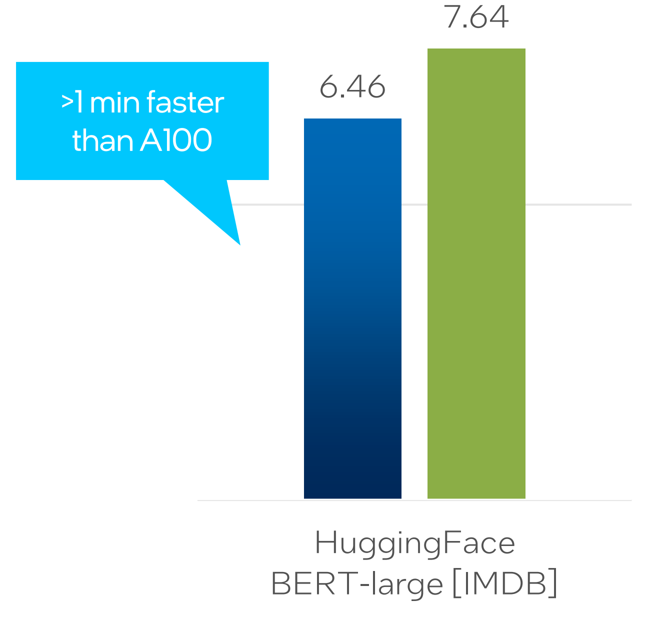
Lower is better



Xeon 8480+ [Intel® Extension for PyTorch
NVIDIA A100 [Stock PyTorch]

Xeon 8480+ [Intel® Optimization for TensorFlow]
NVIDIA A100 [Stock TensorFlow]

In the lab: Intel optimizations to shorten TTT for large natural language models

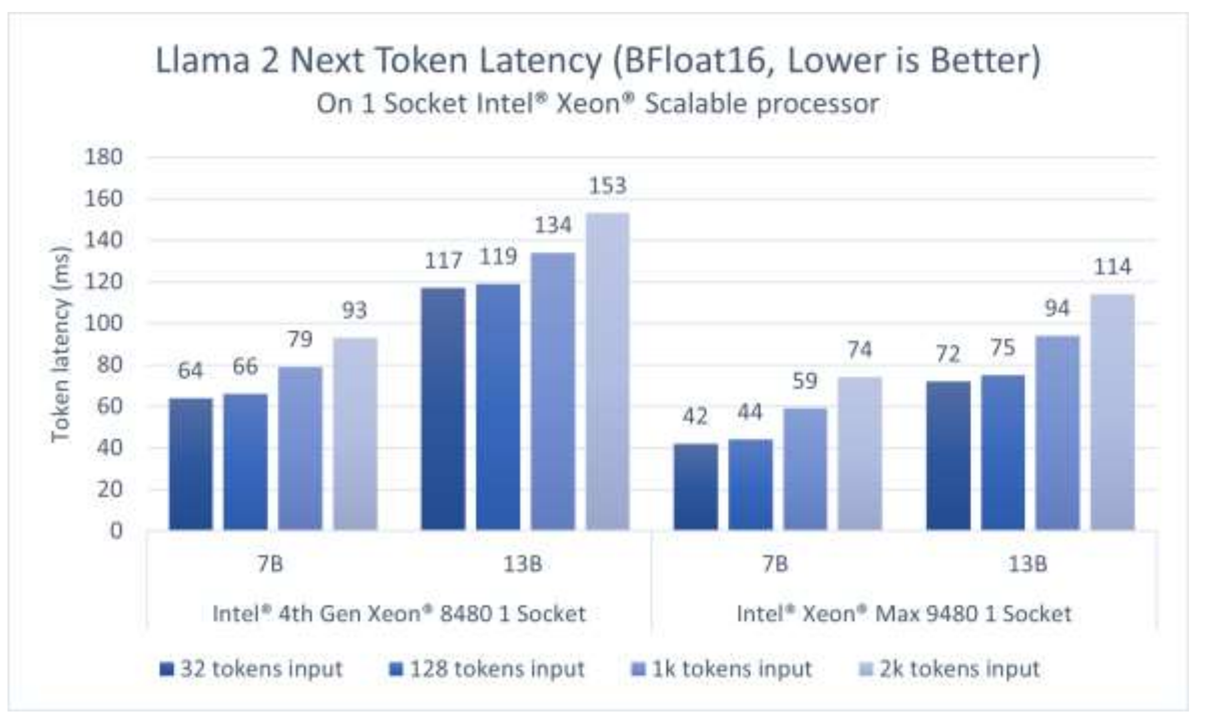


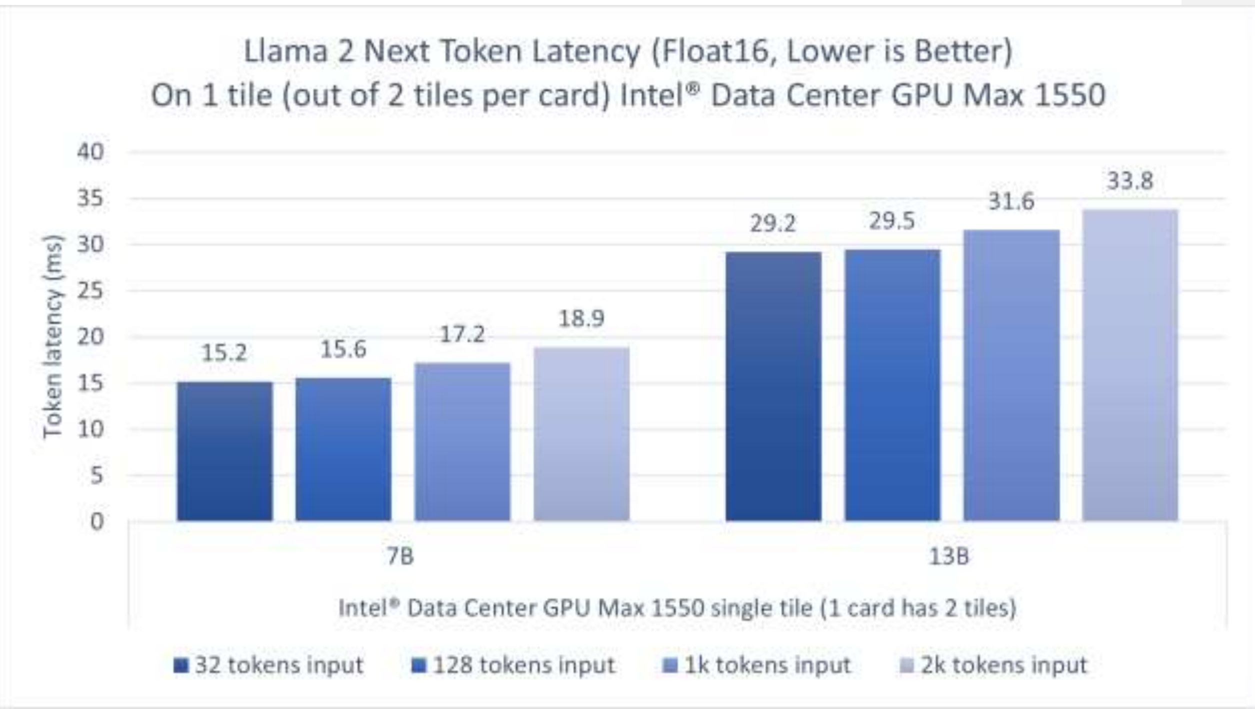
Intel® Tensor Processing Primitives (TPP) Extension for PyTorch

NVIDIA A100 [Stock PyTorch]

See backup for workloads and configurations. Results may vary.

Llama 2 Inference Performance



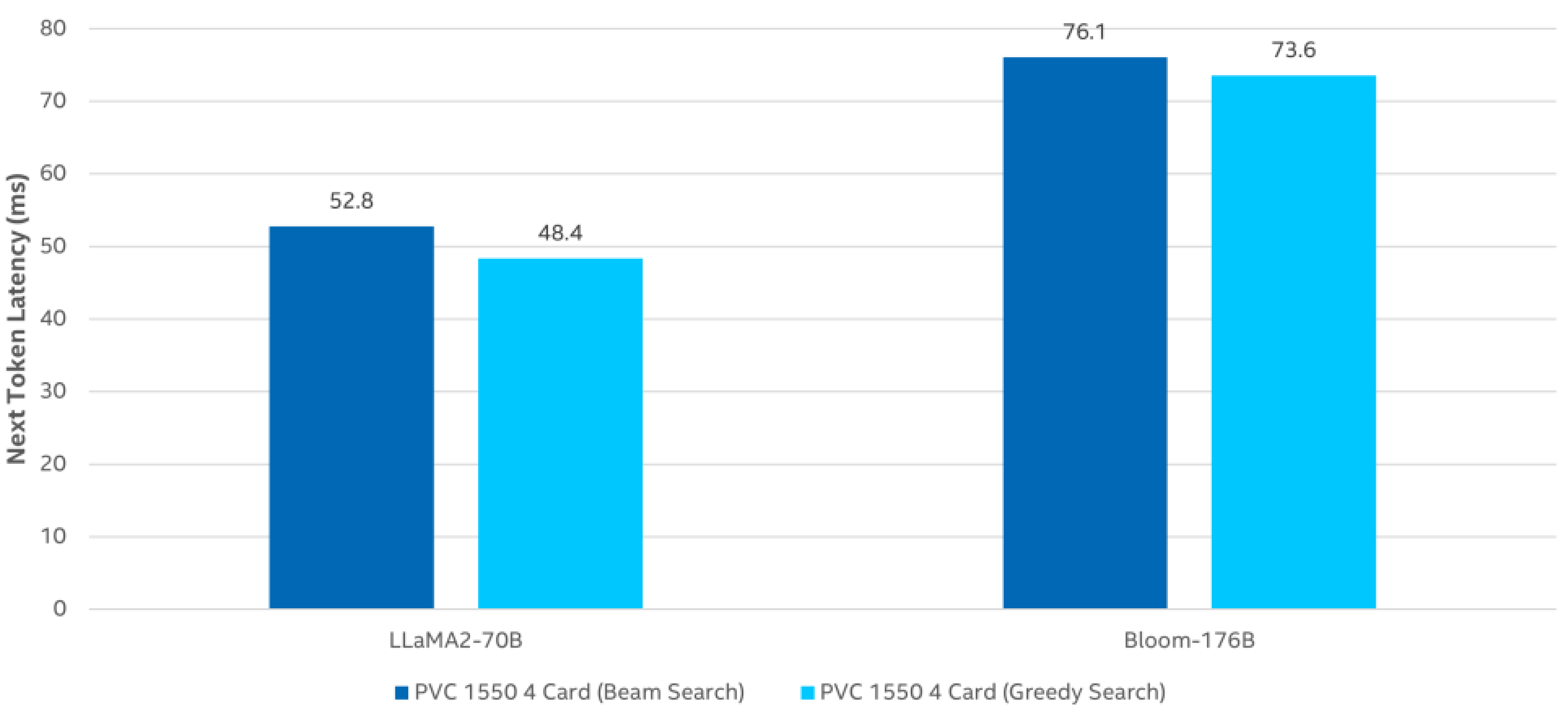


One 4th Gen Xeon socket delivers latencies under 100ms with 7 billon parameter and 13 billon parameter size of models. Users can run 2 parallel instances, one on each socket, for higher throughput and to serve clients independently

Intel® Data Center GPU Max 1550: Users can run 2 parallel instances, one on each tile, for higher throughput and to serve clients independently.

Ref: https://www.intel.com/content/www/us/en/developer/articles/news/llama2.html

Llama 2 – 70B & Bloom-175B Inference Performance



Ref: https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/performance.html

Conclusion

Key Takeaways & Call to Action

- 4th Gen Intel® Xeon®(SPR) & Intel® Data Center GPU Max Series 1550(PVC) Enhances DL Workloads on PyTorch and TensorFlow and are accelerated by AMX & XMX instruction set respectively.
- Minimal code changes are needed in PyTorch and TensorFlow to take advantage of AMX & XMX and lower precision datatypes
- Intel provides a plethora of AI software tools to optimize GenAI/LLM AI workloads.
- Many Code samples are available to get started.

Important Links:

Intel® oneAPI Toolkits

Intel Extension for PyTorch

Intel® Extension for TensorFlow*

Intel Extension for Transformers

VTune Profiler

Getting Started Samples

Model Zoo for Intel® Architecture GitHub

Intel oneAPI Powered AI Reference Kit

OPEA [Open Platform for Enterprise AI]

Intel® Tiber™ Developer Cloud

Intel AI Tools Selector

Bring AI Everywhere

An Entire Ecosystem Built for Artificial Intelligence

From hardware to model support and advanced AI dev tools, Intel has everything you need to successfully build your AI solutions and get started today.

Solution Brief

Hardware for the Whole Pipeline

Intel has CPUs, GPUs, and accelerators tailor-made for AI workloads, wherever you are in the project lifecycle.

Data Prep | Train & Tune | Inference

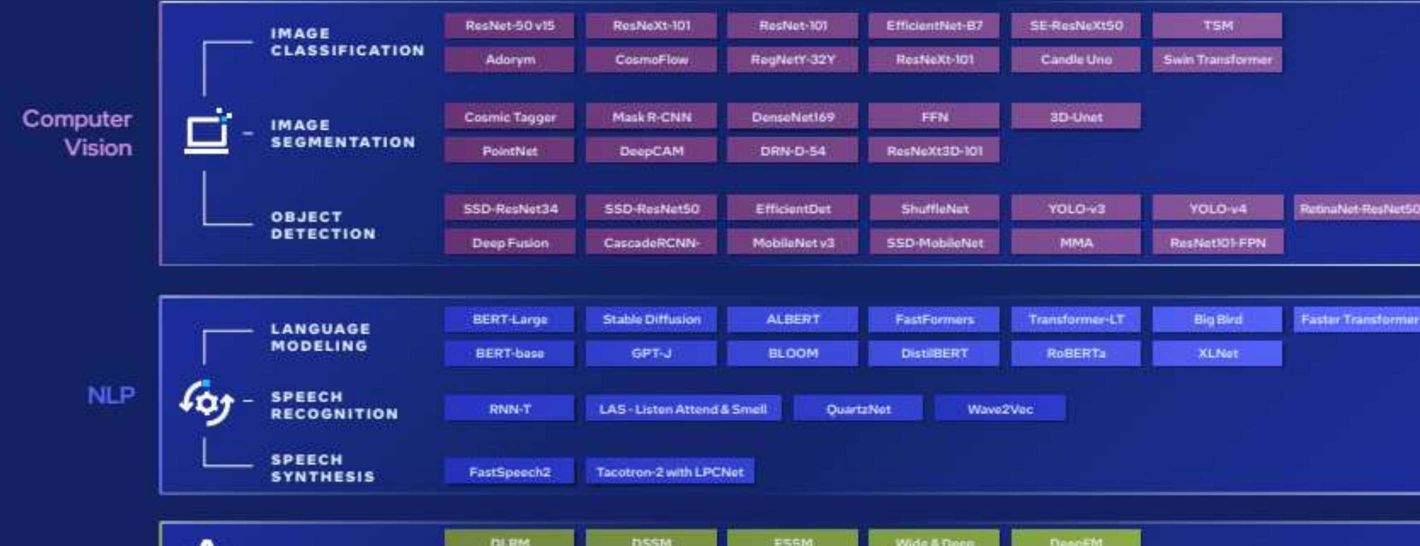
Intel's AI GPU Building Block

Intel® Data Center GPU Max Series Core Features:

- Xe cores
- Media engines and VRAM Xe cores for parallel processing
- Intel® XMX cores for DL and neural networks
- Data Types / Precision: FP64, FP32, BF16, Int8, TF32, and Int4
- Memory size: L2 cache
- Memory bandwidth: HBM2e-126, 48
- Xe link for fast GPU-to-GPU connectivity

Supporting 50+ AI Models

Intel® Data Center GPU Max Series already supports the industry's most popular computer vision, natural language processing, and recommendation models, with additional model support planned for the future.



Solution Brief

AI Tools for **Every Task**

Accelerate end-to-end data science and analytics pipelines with free tools from Intel!

Data Analytics

Intel® Distribution of Modin

- Accelerate your pandas workflows and scale data preprocessing across multi-nodes.

Inference Optimization

Intel® Neural Compressor

- Reduce model size and speed up inference for deployment on CPUs or GPUs.

Intel® AI Reference Models

- Pre-trained, Intel-optimized models.

Deep Learning

PyTorch Optimizations from Intel

- Intel releases its newest optimizations and features in Intel® Extension for PyTorch* before upstreaming them into open source PyTorch.

Tensorflow Optimizations from Intel

- TensorFlow* has been directly optimized for Intel architecture using the primitives of Intel® oneAPI Deep Neural Network Library (oneDNN) to maximize performance.

Intel® AI Reference Models

- Pre-trained, Intel-optimized models.

cnvrg.io

- A full-service machine learning operating system that enables you to manage all your AI projects from one place. cnvrg.io is an optional component and requires a separate license.

Machine Learning

Intel® Extension for Scikit-learn

- Accelerate scikit-learn applications on Intel® CPUs and GPUs across single- and multi-nodes.

Intel® Optimization for XGBoost

- Significantly speed up model training and improve accuracy for better predictions.

cnvrg.io

- A full-service machine learning operating system that enables you to manage all your AI projects from one place. cnvrg.io is an optional component and requires a separate license.

4

Get Started Now with Intel® Developer Cloud

You don't need to wait to start developing AI models with Intel® hardware and software. Register for Intel® Developer Cloud today to learn, prototype, and test applications and workloads on a cluster of the latest Intel® technology. Begin on your laptop now and scale when the time is right.

- Intel® Xeon® Processors ›
- Intel® Data Center GPU Max Series ›
- Intel® Developer Cloud ›
- AI Software Solutions with Intel ›
- What is oneAPI? ›
- What is OpenVINO™ Toolkit? ›

HARDWARE CATALOG

Access the latest Intel® Data Center GPU Max Series instances, pre-installed with relevant toolkits:*

Request access here:

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Thank you for your attention!

Appendix

PyTorch Benchmarking Configurations

4th Generation Intel® Xeon® Scalable Processors

Hardware and software configuration (measured October 24, 2022):

- **Deep Learning config:**
  - Hardware configuration for Intel® Xeon® Platinum 8480+ processor (formerly code named Sapphire Rapids): 2 sockets, 56 cores, 350 watts, 16 x 64 GB DDR5 4800 memory, BIOS version EGSDCRB1.SYS.8901.P01.2209200243, operating system: CentOS* Stream 8, using Intel® Advanced Matrix Extensions (Intel® AMX) int8 and bf16 with Intel® oneAPI Deep Neural Network Library (oneDNN) v2.7 optimized kernels integrated into Intel® Extension for PyTorch* v1.13, Intel® Extension for TensorFlow* v2.12, and Intel® Distribution of OpenVINO™ toolkit v2022.3. Measurements may vary.
  - Wall power refers to platform power consumption.
  - If the dataset is not listed, a synthetic dataset was used to measure performance. Accuracy (if listed) was validated with the specified dataset.
- **Transfer Learning config:**
  - Hardware configuration for Intel® Xeon® Platinum 8480+ processor (formerly code named Sapphire Rapids): Use DLSA single node fine tuning, Vision Transfer Learning using single node, 56 cores, 350 watts, 16 x 64 GB DDR5 4800 memory, BIOS version EGSDREL1.SYS.8612.P03.2208120629, operating system: Ubuntu 22.04.1 LT, using Intel® Advanced Matrix Extensions (Intel® AMX) int8 and bf16 with Intel® oneAPI Deep Neural Network Library (oneDNN) v2.6 optimized kernels integrated into Intel® Extension for PyTorch* v1.12, and Intel® oneAPI Collective Communications Library v2021.5.2. Measurements and some software configurations may vary.

3rd Generation Intel® Xeon® Scalable Processors

Hardware and software configuration (measured October 24, 2022):

- Hardware configuration for Intel® Xeon® Platinum 8380 processor (formerly code named Ice Lake): 2 sockets, 40 cores, 270 watts, 16 x 64 GB DDR5 3200 memory, BIOS version SE5C620.86B.01.01.0005.2202160810, operating system: Ubuntu 22.04.1 LTS, int8 with Intel® oneAPI Deep Neural Network Library (oneDNN) v2.6.0 optimized kernels integrated into Intel® Extension for PyTorch* v1.12, Intel® Extension for TensorFlow* v2.10, and Intel® oneAPI Data Analytics Library (oneDAL) 2021.2 optimized kernels integrated into Intel® Extension for Scikit-learn* v2021.2. XGBoost v1.6.2, Intel® Distribution of Modin* v0.16.2, Intel oneAPI Math Kernel Library (oneMKL) v2022.2, and Intel® Distribution of OpenVINO™ toolkit v2022.3. Measurements may vary.
- If the dataset is not listed, a synthetic dataset was used to measure performance. Accuracy (if listed) was validated with the specified dataset.

*All performance numbers are acquired running with 1 instance of 4 cores per socket

PyTorch/TensorFlow Benchmarking Configurations

5th Generation Intel® Xeon® Scalable Processors

Hardware and software configuration (measured October 24, 2023):

- **Deep Learning configuration:**
  - Hardware configuration for Intel® Xeon® Platinum 8592+ processor (code named Emerald Rapids): 2 sockets for inference, 1 socket for training, 64 cores, 350 watts, 1024GB 16 x 64GB DDR5 5600 MT/s memory, operating system CentOS* Stream 9. Using Intel® Advanced Matrix Extensions (Intel® AMX) int8 and bf16 with Intel® oneAPI Deep Neural Network Library (oneDNN) optimized kernels integrated into Intel® Extension for PyTorch*, Intel® Extension for TensorFlow*, and Intel® Distribution of OpenVINO™ toolkit. Measurements may vary. If the dataset is not listed, a synthetic dataset was used to measure performance.
- **Transfer Learning configuration:**
  - Hardware configuration for Intel® Xeon® Platinum 8592+ processor (code named Emerald Rapids): 2 sockets, 64 cores, 350 watts, 16 x 64 GB DDR5 5600 memory, BIOS version 3B05.TEL4P1, operating system: CentOS stream 8, using Intel® Advanced Matrix Extensions (Intel® AMX) int8 and bf16 with Intel® oneAPI Deep Neural Network Library (oneDNN) v2.6.0 optimized kernels integrated into Intel® Extension for PyTorch* v2.0.1, Intel® Extension for TensorFlow* v2.14, and Intel® oneAPI Data Analytics Library (oneDAL) 2023.1 optimized kernels integrated into Intel® Extension for Scikit-learn* v2023.1. Intel® Distribution of Modin* v2.1.1, and Intel oneAPI Math Kernel Library (oneMKL) v2023.1. Measurements may vary.