Choose the Best Accelerated Technology

# Intel Performance optimizations for Deep Learning

Adel Chaibi – AI TSS
adel.chaibi@intel.com
November 1 2022

intel.

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.

# Agenda

- Data precision
- Optimized DL frameworks
  - oneDNN
  - Tensorflow
  - PyTorch
  - Intel Extension for PyTorch

# Data Precision

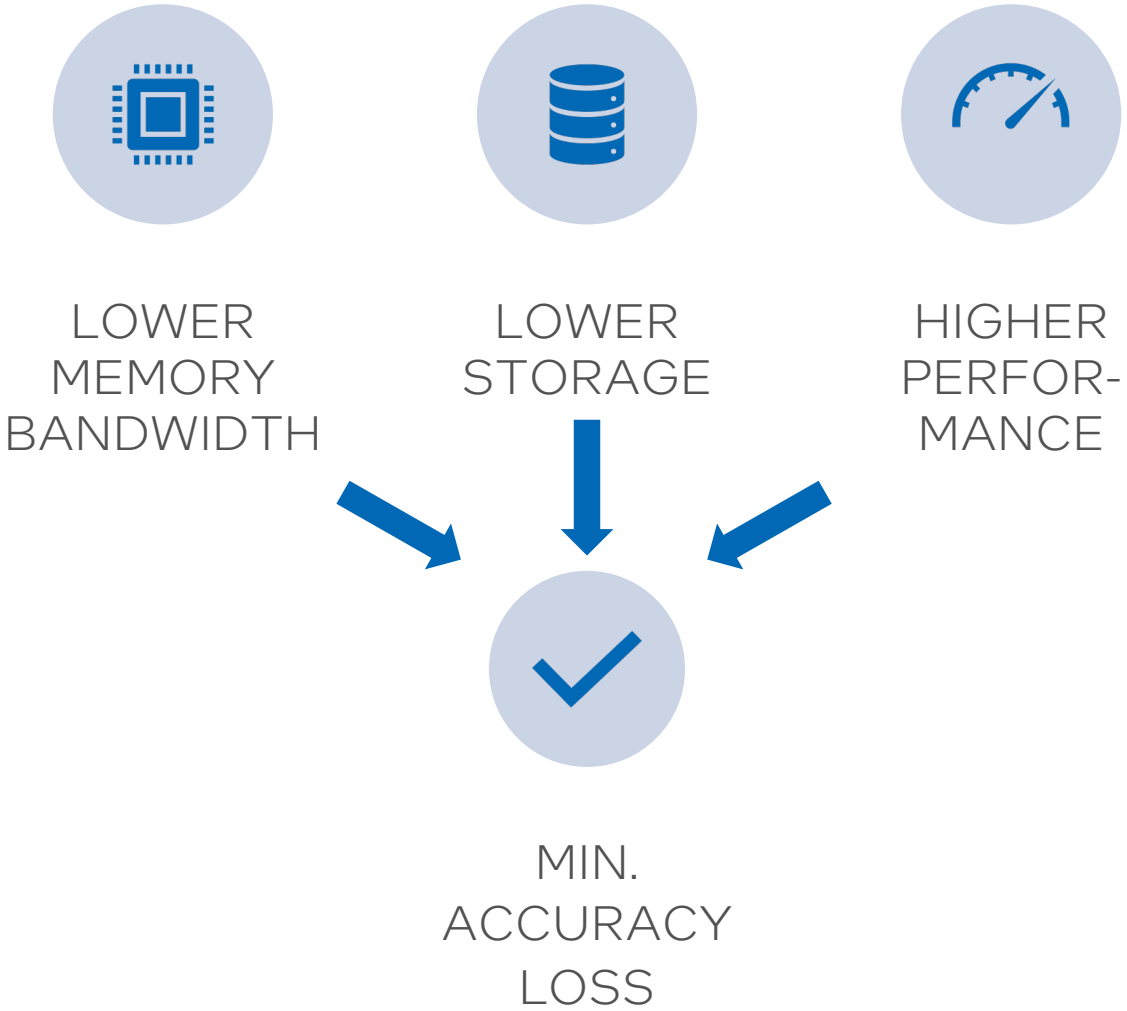
- Data precision:

  Number of bits used to store numerical values in memory

- Commonly found types of precision in Deep Learning:

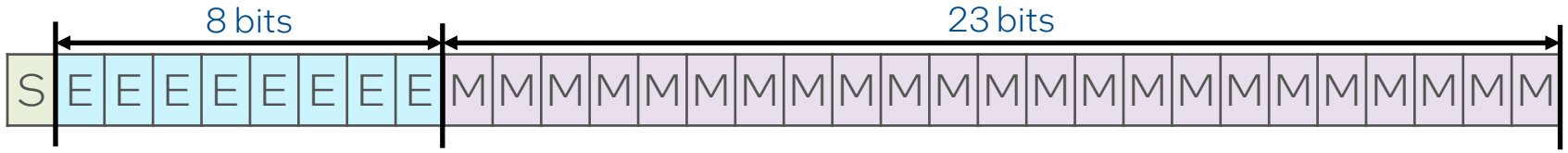
| INT8 | BF16 - FP16 | TF32 | FP32 |
|---|---|---|---|
| 8 bits | 16 bits | 19 bits | 32 bits |

# Lower Precision – Summary



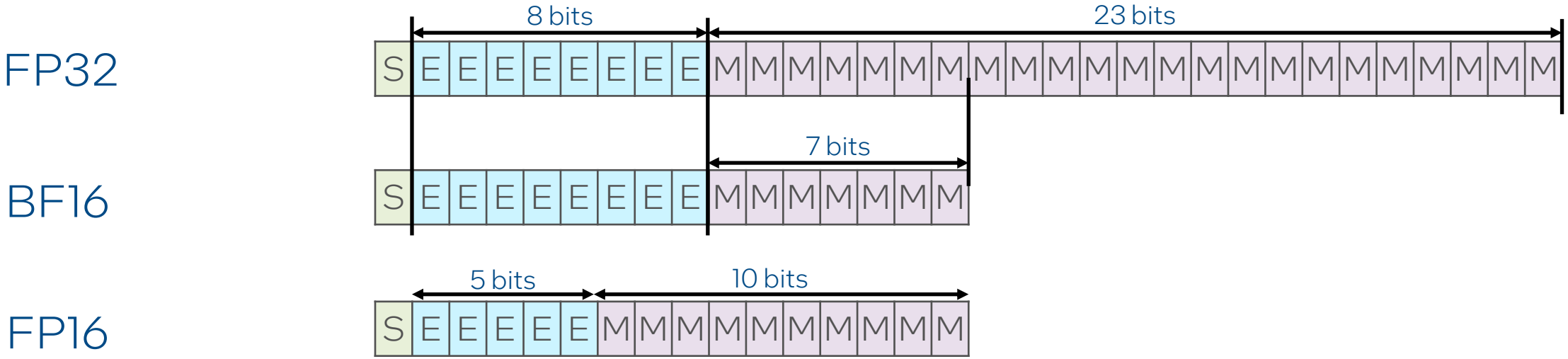
# Floating Point – Precision -32 bits

FP32




- FP32: The standard type for all neural network computations

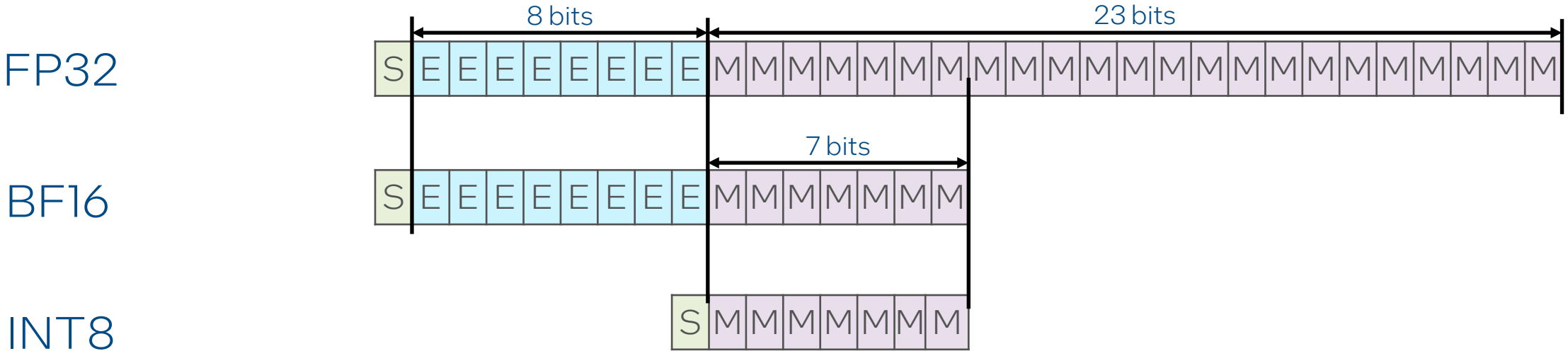
# Floating Point – Precision – 16bits




- BF16: Efficient replacement for FP32 in training and inference

- Benefit of BF16
  - Performance 2x up
  - Comparable accuracy loss against fp32
  - No loss scaling, compared to fp16
  - Can be used for training (mixed-precision training)

# Floating Point – Precision – 8 bits




- INT8: Significant speed-up in inference with small loss in accuracy
- Not suitable for training but recommended for inference when a small loss of accuracy is accepted.
- Intel Hardware takes great advantage of INT8 and BF16 precision

# Intel® Xeon® Scalable Processors

## The Only Data Center CPU with Built-in AI Acceleration

Intel Advanced Vector Extensions 512

Intel Deep Learning Boost (Intel DL Boost)

Intel Optane Persistent Memory

### Shipping

**Cascade Lake**

New Intel DL Boost (VNNI)
New memory storage hierarchy

**Cooper Lake**

Intel DL Boost (BFLOAT16)

### April 2021

**Ice Lake**

Intel DL Boost (VNNI) and new Intel Software Guard Extensions (Intel® SGX) that enable new AI use cases like federated learning

### 2022

**Sapphire Rapids**

Intel Advanced Matrix Extensions (AMX) extends built-in AI acceleration capabilities on Xeon Scalable

**Leadership performance**

# Optimized Deep Learning FW

TensorFlow

PyTorch

# Intel's oneAPI Ecosystem

## Built on Intel's Rich Heritage of CPU Tools Expanded to XPUs

### oneAPI

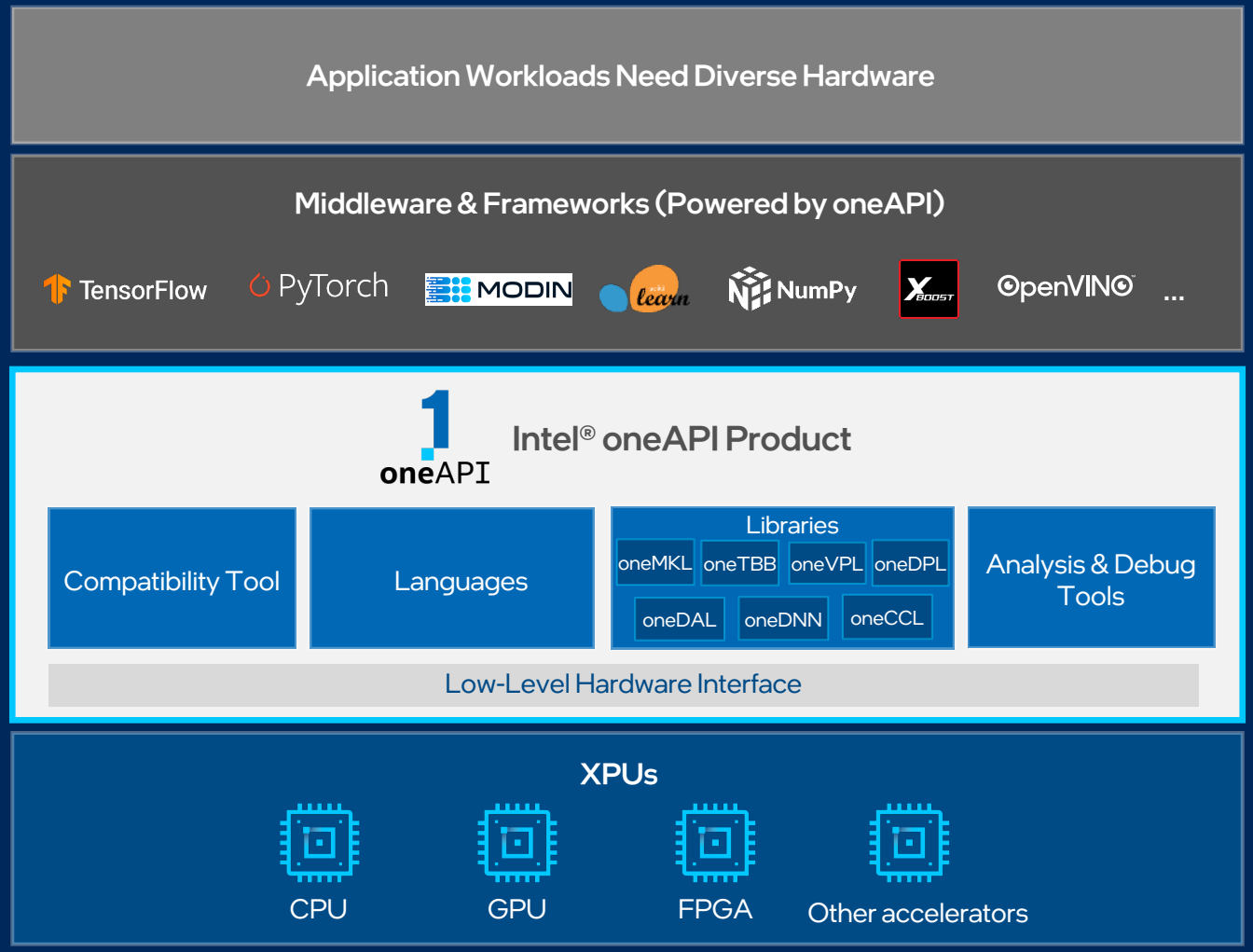
A cross-architecture language based on C++ and SYCL standards

Powerful libraries designed for acceleration of domain-specific functions

A complete set of advanced compilers, libraries, and porting, analysis and debugger tools

### Powered by oneAPI

Frameworks and middleware that are built using one or more of the oneAPI industry specification elements, the DPC++ language, and libraries listed on oneapi.com.




Available Now



Visit software.intel.com/oneapi for more details
Some capabilities may differ per architecture and custom-tuning will still be required. Other accelerators to be supported in the future.

oneAPI

# Intel® oneAPI **Base** Toolkit

Native Code Developers

A core set of high-performance tools for building C++, Data Parallel C++ applications & oneAPI library-based applications

# Add-on **Domain-specific** Toolkits

Specialized Workloads

### Intel® oneAPI Tools for HPC

Deliver fast Fortran, OpenMP & MPI applications that scale

### Intel® oneAPI Tools for IoT

Build efficient, reliable solutions that run at network's edge

### Intel® oneAPI Rendering Toolkit

Create performant, high-fidelity visualization applications

# Toolkits **powered by oneAPI**

Data Scientists & AI Developers

### Intel® AI Analytics Toolkit

Accelerate machine learning & data science pipelines with optimized DL frameworks & high-performing Python libraries

### Intel® Distribution of OpenVINO™ Toolkit

Deploy high performance inference & applications from edge to cloud

# Intel® AI Analytics Toolkit

## Powered by oneAPI

Accelerate end-to-end AI and data analytics pipelines with libraries optimized for Intel® architectures

### Who Uses It?

Data scientists, AI researchers, ML and DL developers, AI application developers

### Top Features/Benefits

- Deep learning performance for training and inference with Intel optimized DL frameworks and tools
- Drop-in acceleration for data analytics and machine learning workflows with compute-intensive Python packages

Learn More: software.intel.com/oneapi/ai-kit

**Deep Learning**

- Intel® Optimization for TensorFlow
- Intel® Optimization for PyTorch
- Intel® Neural Compressor
- Model Zoo for Intel® Architecture

**Data Analytics & Machine Learning**

Accelerated Data Frames

- Intel® Distribution of Modin
- OmniSci Backend

Intel® Distribution for Python

- XGBoost
- Scikit-learn
- Daal-4Py
- NumPy
- SciPy
- Pandas

**Samples and End2End Workloads**

CPU GPU

Supported Hardware Architechures[1]

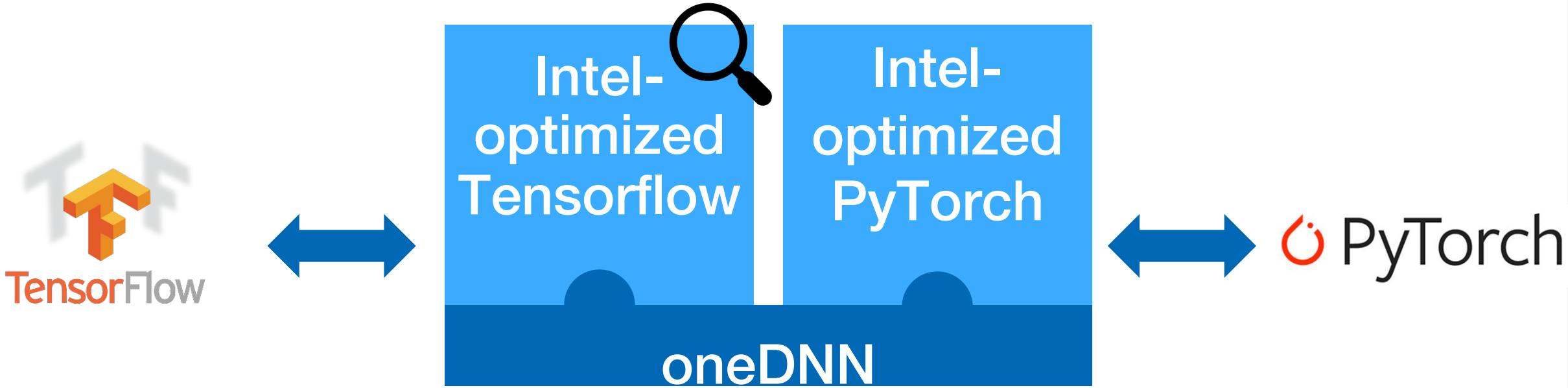
Hardware support varies by individual tool. Architecture support will be expanded over time.
Other names and brands may be claimed as the property of others.

**Get the Toolkit HERE or via these locations**

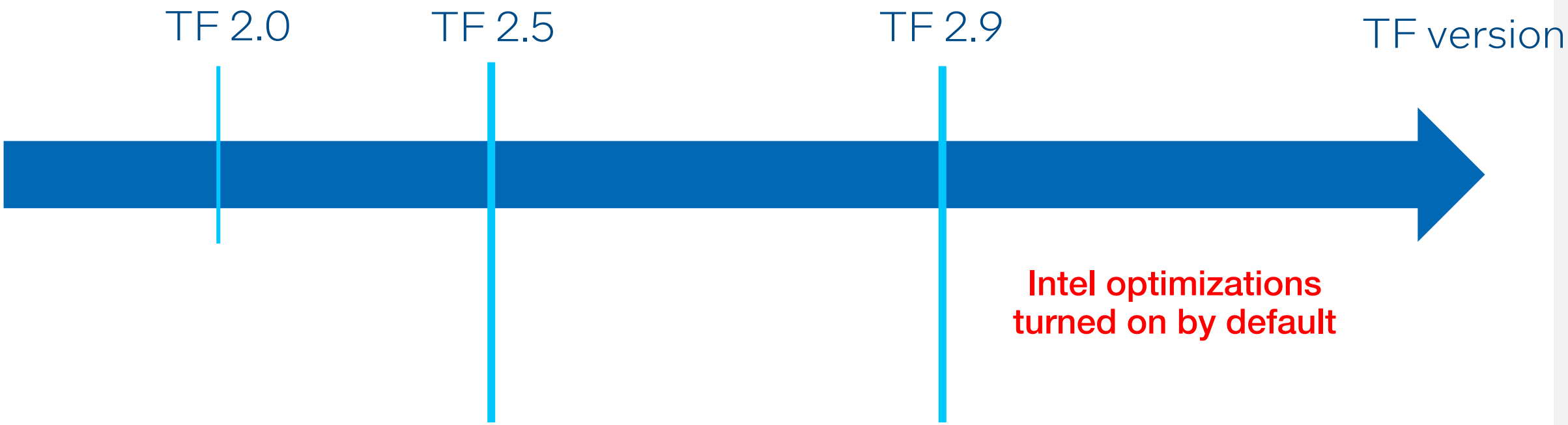
- Intel Installer
- Docker
- Apt, Yum
- Conda
- Intel® DevCloud

# Intel-optimized Deep Learning Frameworks

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



# Tensorflow timeline of Intel optimizations



# How to get the optimized frameworks

- In the Intel AI Analytics toolkit

No need to call the flag for Tensorflow

- Through the framework pip/conda wheel:

| | | | | |
|---|---|---|---|---|
| PyTorch Build | Stable (1.11.0) | Preview (Nightly) | LTS (1.8.2) | |
| Your OS | Linux | Mac | Windows | |
| Package | Conda | Pip | LibTorch | Source |
| Language | Python | C++ / Java | | |
| Compute Platform | CUDA 10.2 | CUDA 11.3 | ~~ROCm 4.5.2 (beta)~~ | CPU |
| Run this Command: | `pip3 install torch torchvision torchaudio` | | | |

TensorFlow > Install

Was this helpful?

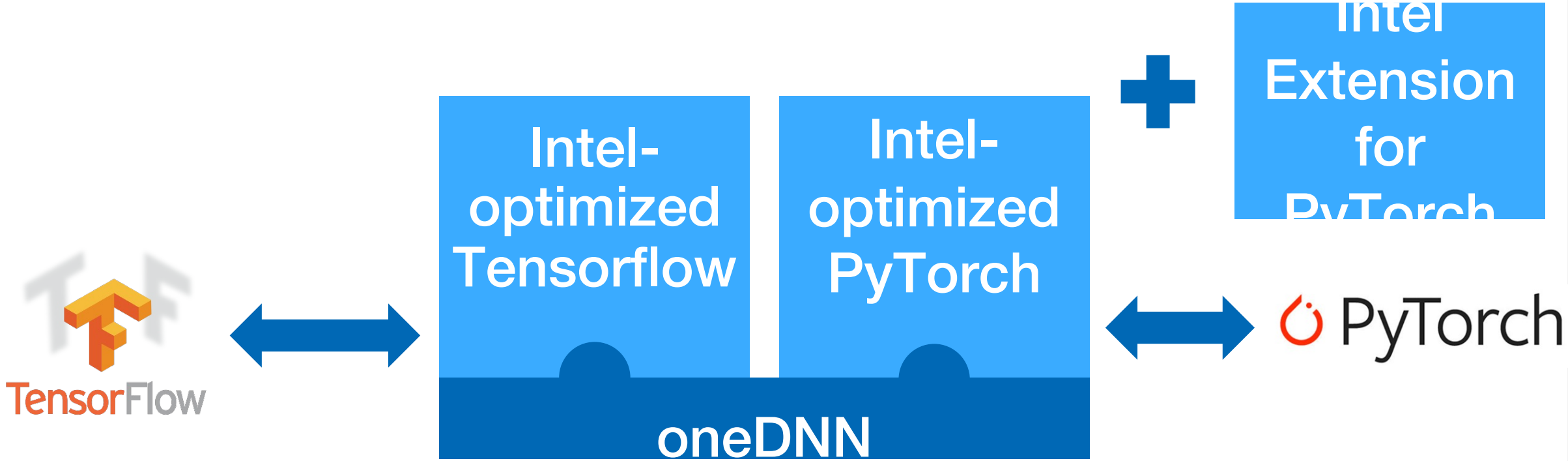
## Install TensorFlow with pip

TensorFlow 2 packages are available

- `tensorflow` —Latest stable release with CPU and GPU support *(Ubuntu and Windows)*
- `tf-nightly` —Preview build *(unstable)*. Ubuntu and Windows include GPU support.

# Intel-optimized Deep Learning Frameworks

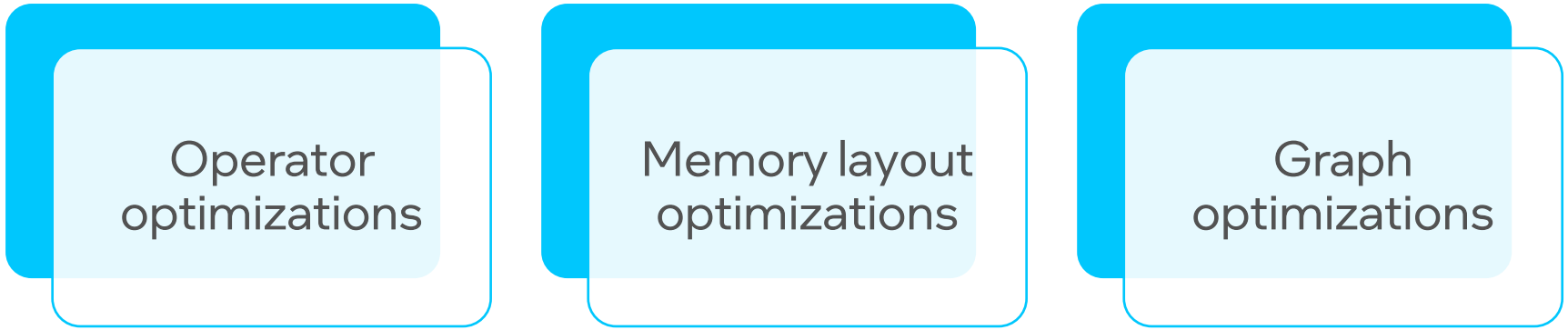
- Intel Extension for PyTorch is an additional module for functions not supported in standard PyTorch (such as mixed precision and dGPU support)
- As they offer more aggressive optimizations, they offer bigger speed-up for training and inference



# Optimizations

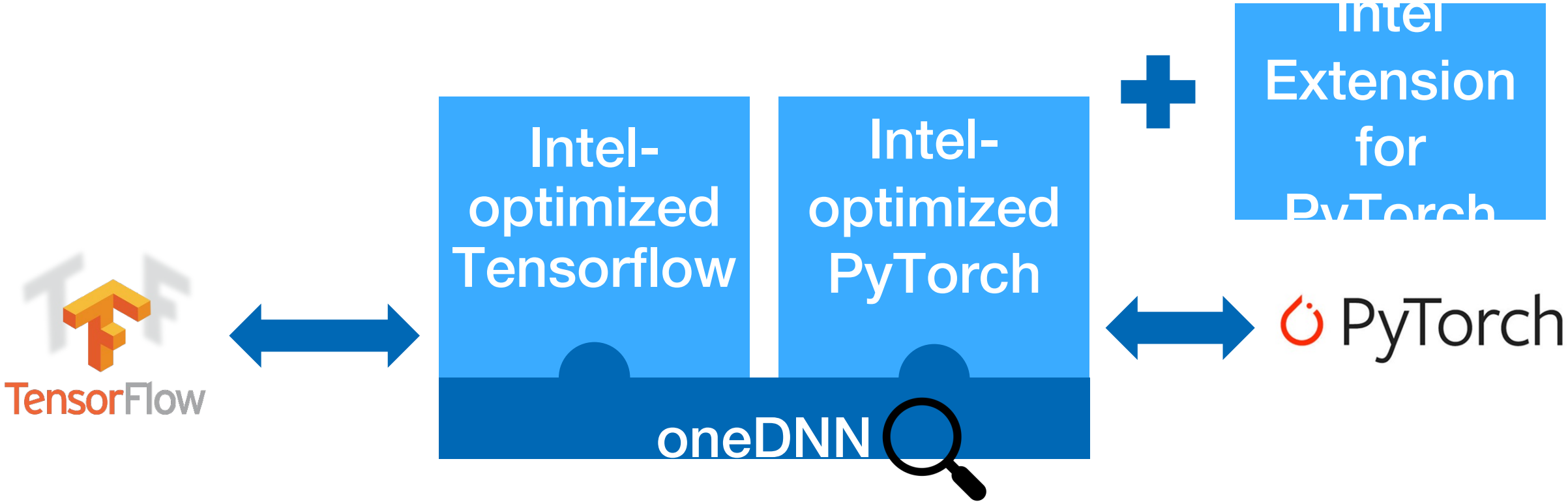
Same type of optimizations at two different levels:

1) In Intel Extension for PyTorch
2) in oneDNN



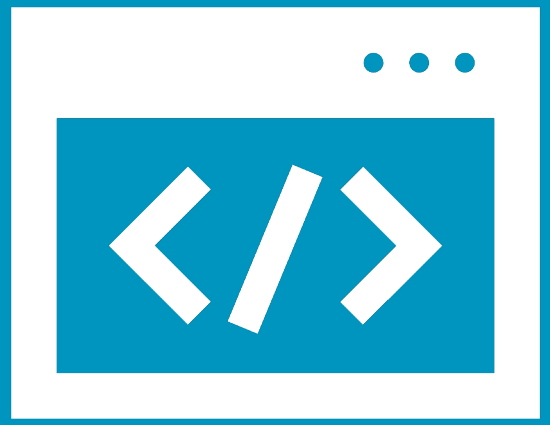

Intel Extension for PyTorch optimizations extends the oneDNN optimizations

# Intel-optimized Deep Learning Frameworks



# oneDNN

Intel® oneAPI Deep Neural Network Library



# Intel® oneAPI Deep Neural Network Library (oneDNN)

- An open-source cross-platform performance library for deep learning applications
  - Helps developers create high performance deep learning frameworks
  - Abstracts out instruction set and other complexities of performance optimizations
  - Open source for community contributions

- Supported data precision
  - **Training**: float32, bfloat16
  - **Inference**: float32, bfloat16, float16, and int8

- Runs on Intel CPU and GPU

Operator optimizations

Replace default kernels by highly-optimized kernels (using Intel® oneDNN)

Memory layout optimizations

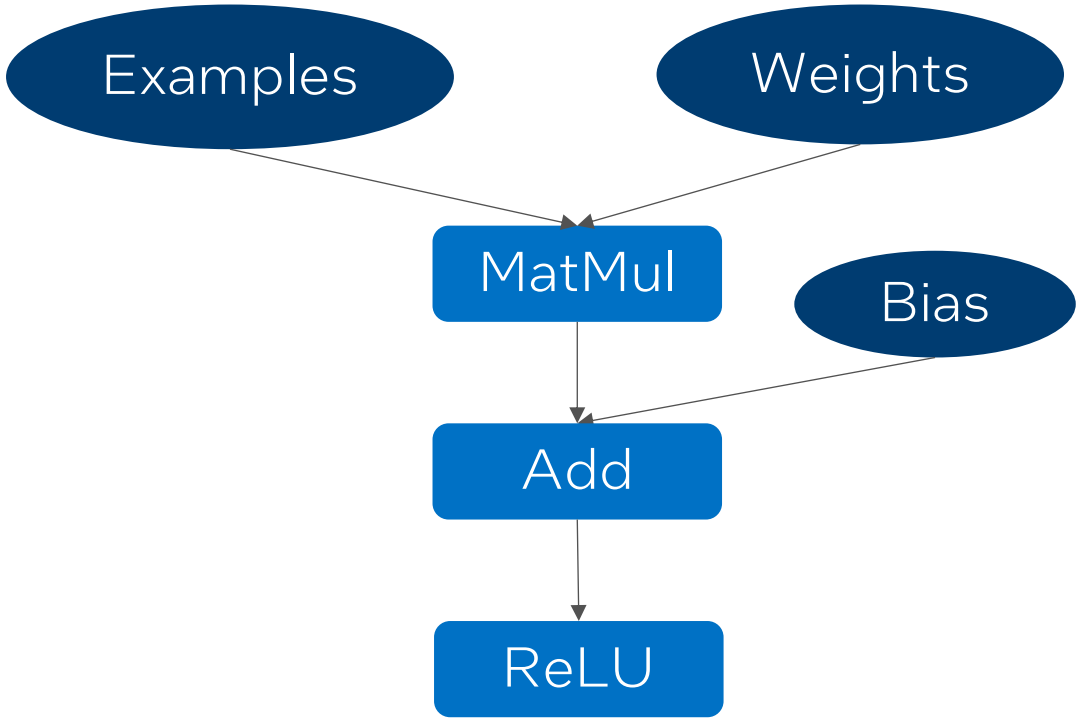
Set optimal layout for each kernel, while minimizing memory changes in between kernels

Graph optimizations

Fusion, Layout Propagation

# Operator optimizations

In TensorFlow, computation graph is a data-flow graph.



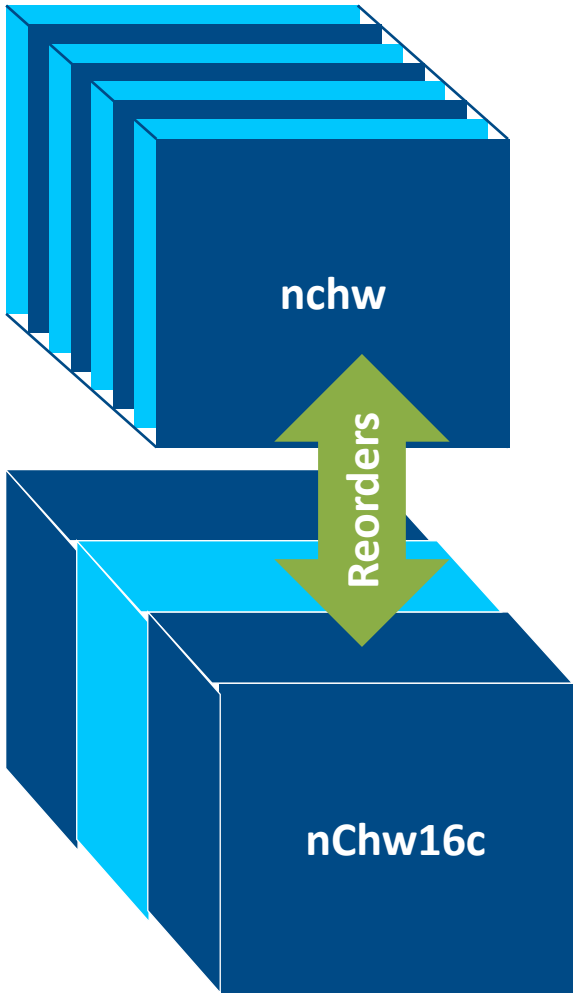
# Operator optimizations

- Replace default kernels by highly-optimized kernels (using Intel® oneDNN)
- Adapt to available instruction sets (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<br>2D Winograd convolution |
| Inner Product | 2D/3D Inner Production |
| Pooling | 2D/3D Maximum<br>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;<br>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 |

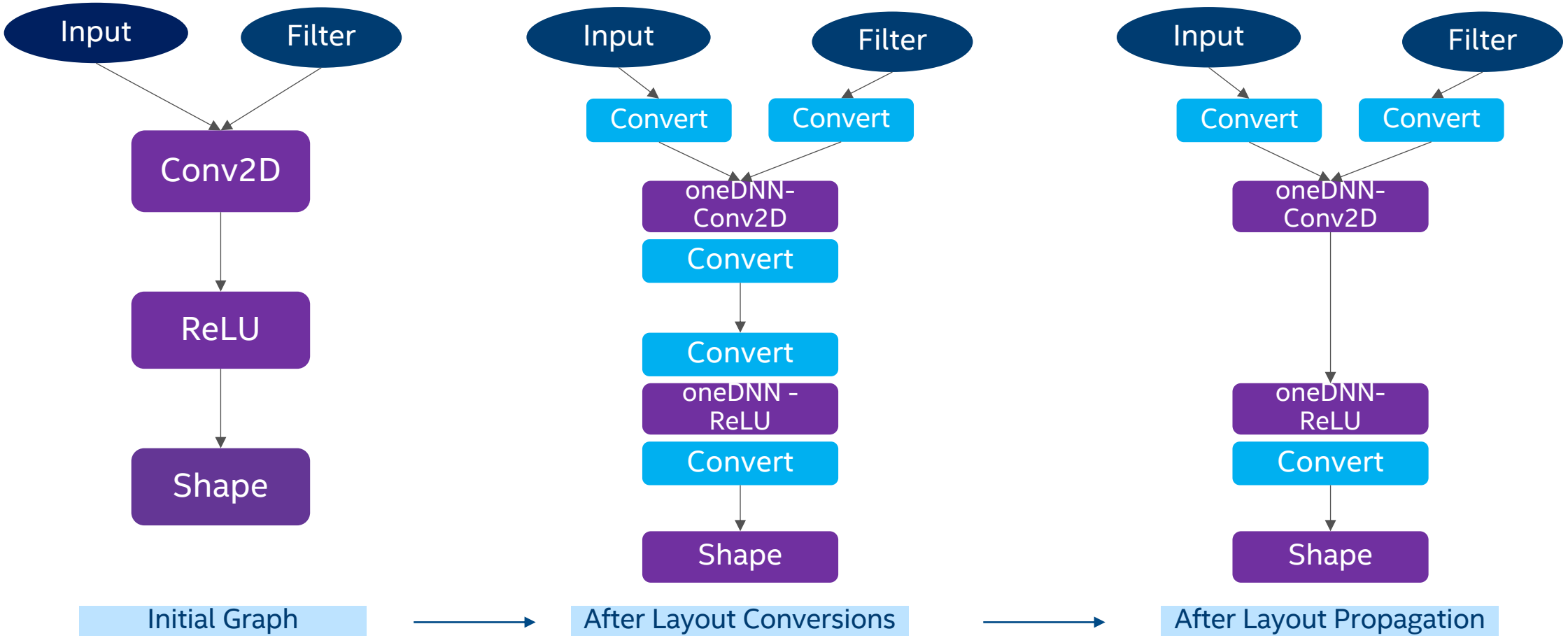
# Memory layouts optimization

- 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



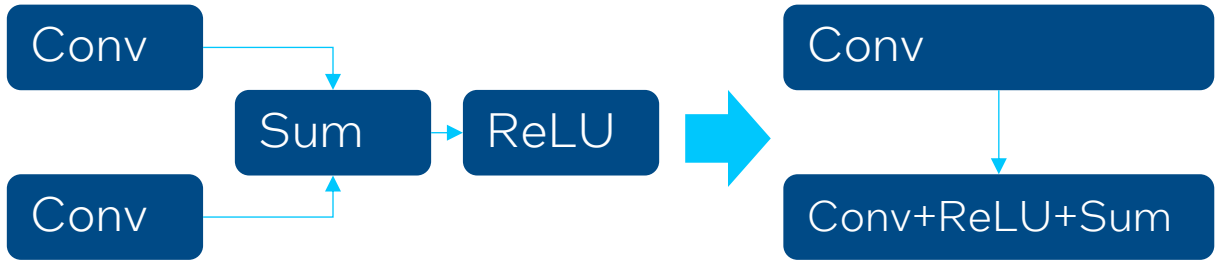

More details: https://oneapi-src.github.io/oneDNN/understanding_memory_formats.html

# Graph optimizations: layout propagation

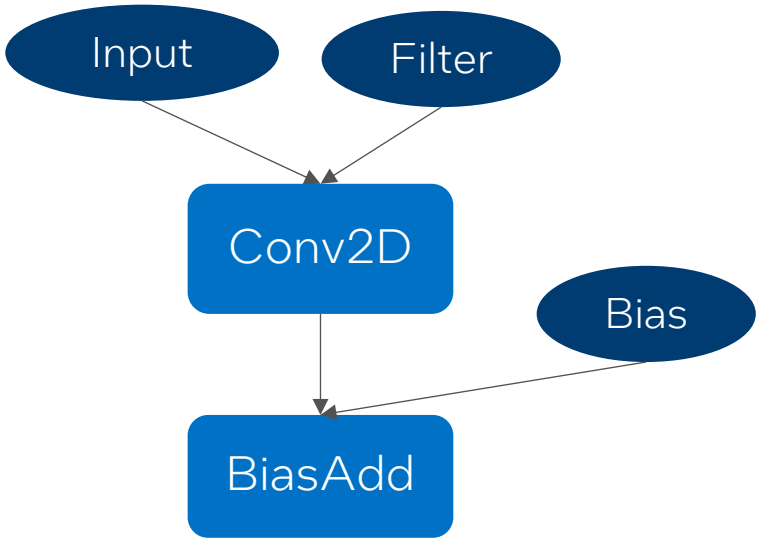


# Fusing computations

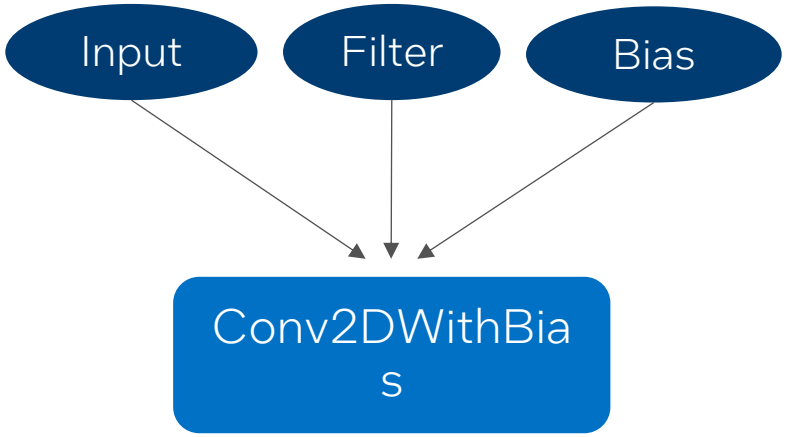
- On Intel processors a high percentage of time is typically spent in bandwidth-limited ops such activation functions
  - ~40% of ResNet-50, even higher for inference
- The solution is to fuse BW-limited ops with convolutions or one with another to reduce the number of memory accesses
  - We fuse patterns: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



# Graph optimizations: fusion

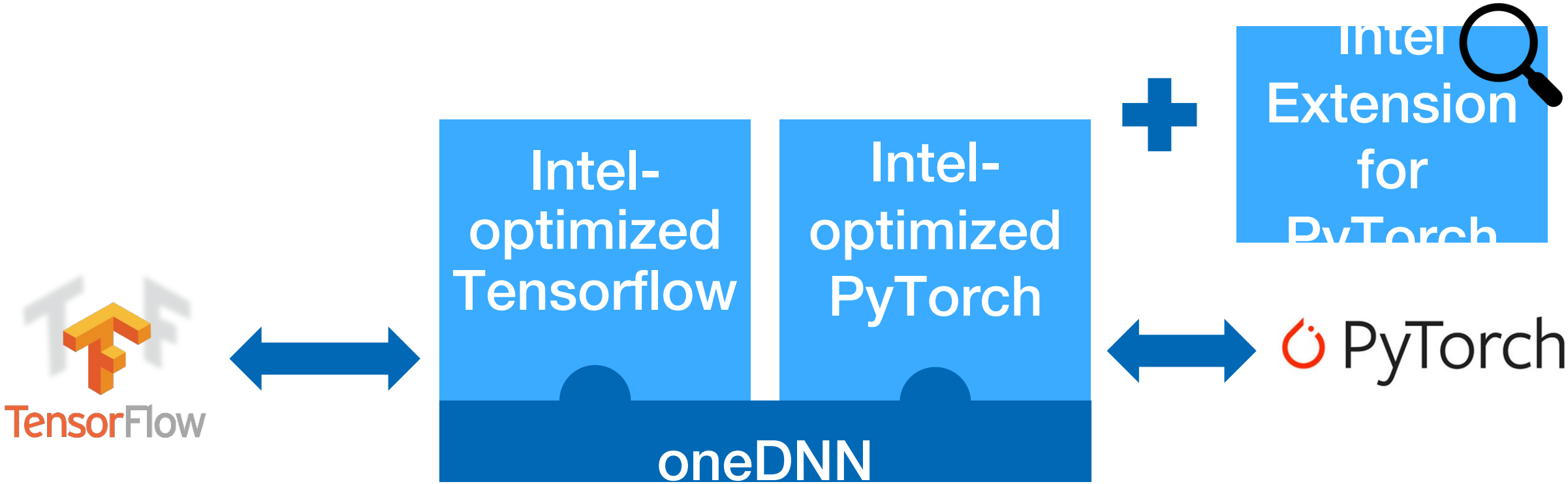



Before Merge




After Merge

# Intel-optimized Deep Learning Frameworks



# 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

# Python – Imperative Mode

- FP32

```python
import torch
import torchvision.models as models

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

import intel_extension_for_pytorch as ipex
model = model.to(memory_format=torch.channels_last)
model = ipex.optimize(model)
data = data.to(memory_format=torch.channels_last)

with torch.no_grad():
  model(data)
```

- BFloat16

```python
import torch
from transformers import BertModel

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])

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

with torch.no_grad():
  with torch.cpu.amp.autocast():
    model(data)
```

https://intel.github.io/intel-extension-for-pytorch/1.11.0/tutorials/examples.html

# Python – TorchScript Mode

- FP32

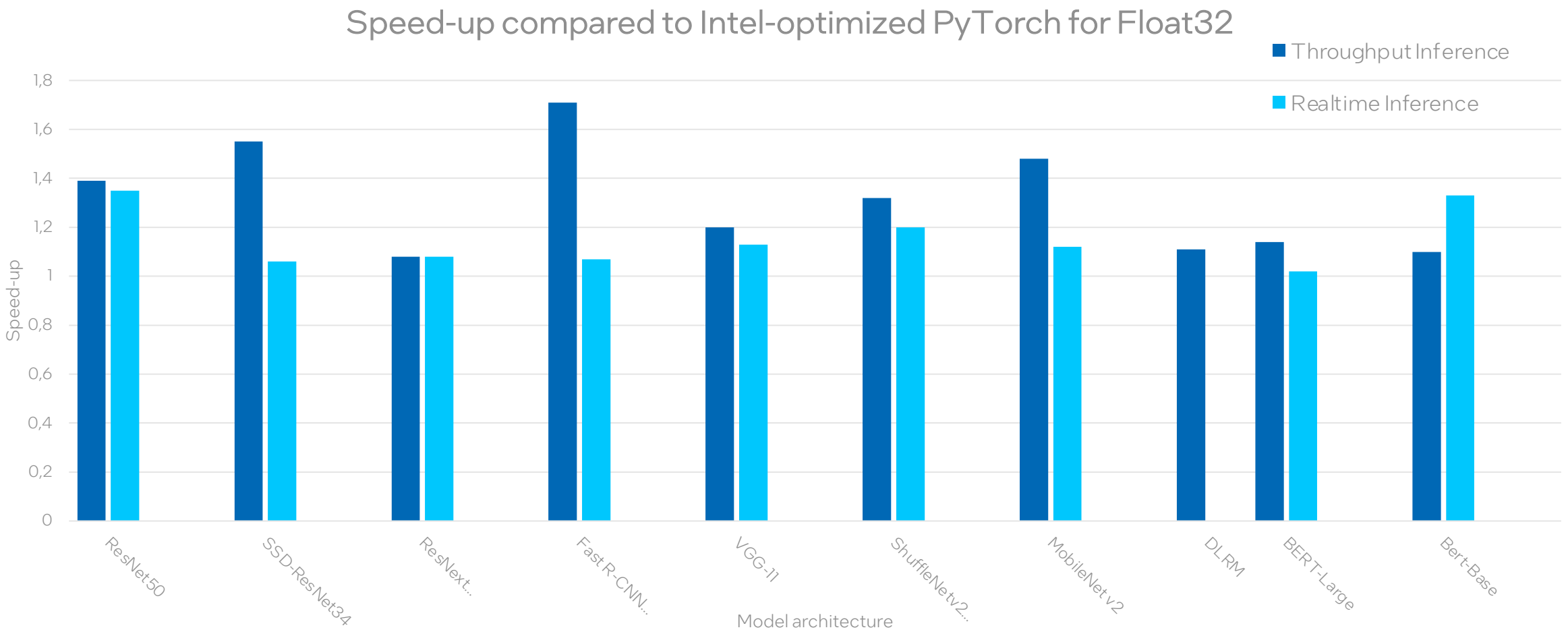
```python
import torch
from transformers import BertModel

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])

import intel_extension_for_pytorch as ipex
model = ipex.optimize(model)

with torch.no_grad():
  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)
```

- BFloat16

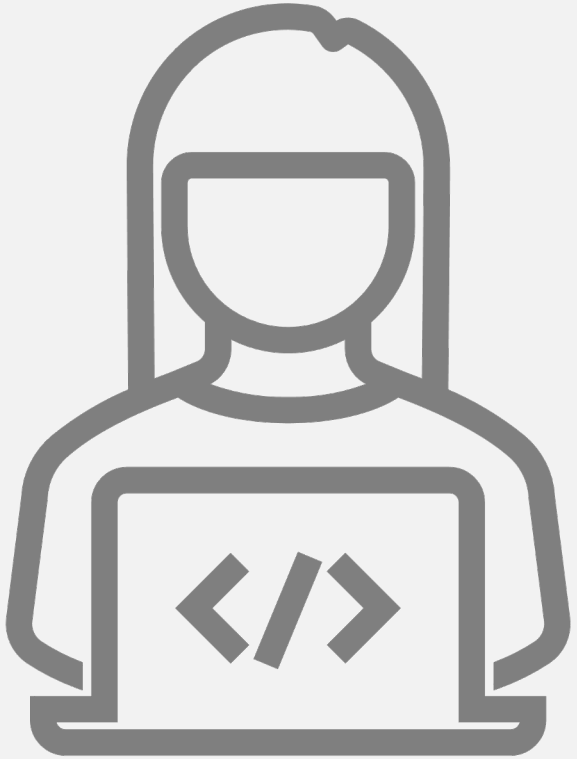
```python
import torch
import torchvision.models as models

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

import intel_extension_for_pytorch as ipex
model = model.to(memory_format=torch.channels_last)
model = ipex.optimize(model, dtype=torch.bfloat16)
data = data.to(memory_format=torch.channels_last)

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

    model(data)
```

https://intel.github.io/intel-extension-for-pytorch/1.11.0/tutorials/examples.html

# Intel Extension for PyTorch benchmark




Refer to https://intelai.github.io/intel-extension-for-pytorch/tutorials/performance.html for configuration details



# How to get the Intel Extension for PyTorch

- **pip wheel:**

python -m pip install intel_extension_for_pytorch

Operator optimizations

Memory/data layout optimizations

Graph optimizations

Optimal Optimizer

Mixed Precision

# Fusing computations

- Intel Extension for PyTorch in JIT/Torchscript mode can fuse:
  - Multi-head-attention fusion, Concat Linear, Linear+Add, Linear+Gelu, Add+LayerNorm fusion and 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

Operator optimizations

Memory/data layout optimizations

Graph optimizations

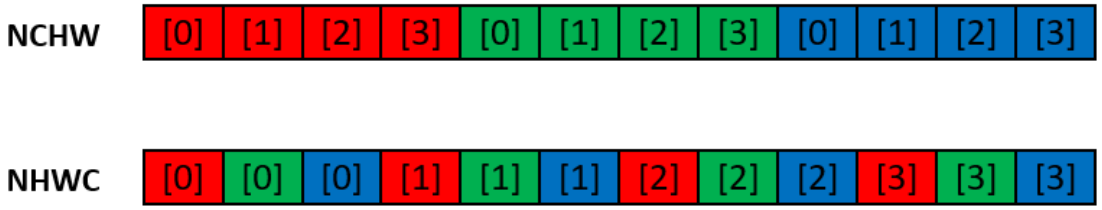
Optimal Optimizer

Mixed Precision

# Data Layout optimization

# Data Layouts in PyTorch

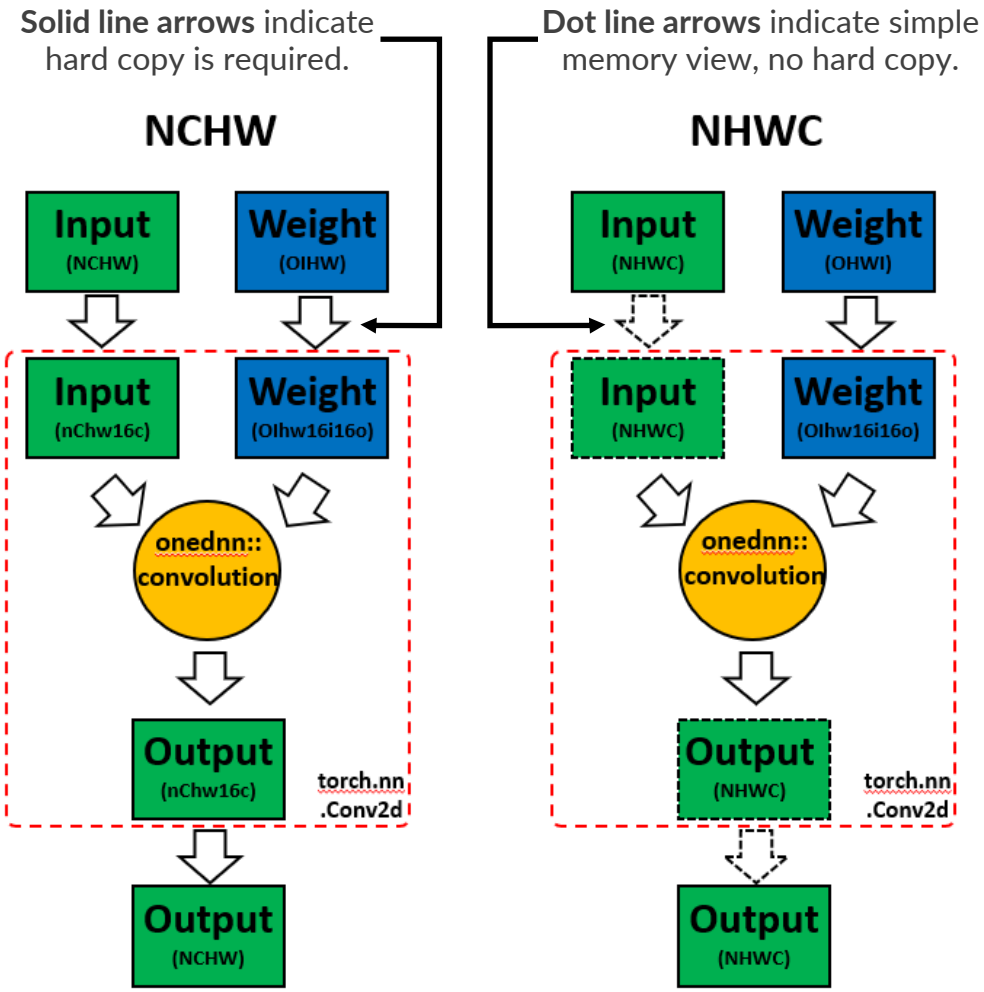
- Used in Vision workloads
- NCHW
  - Default format
  - *torch.contiguous_format*
- NHWC
  - A working-in-progress feature of PyTorch
  - *torch.channels_last*
  - NHWC format yields higher performance




```
## NB: internally blocked format will still be used.
##   aka. we do 'reorder' for 'input', 'weight' and 'output',
##   and believe me this is expensive, roughly 50% perf loss...
input = torch.randn(1, 10, 32, 32)
model = torch.nn.Conv2d(10, 20, 1, 1)
output = model(input)
```

```
input = torch.randn(1, 10, 32, 32)
model = torch.nn.Conv2d(10, 20, 1, 1)
## NB: convert to Channels Last memory format.
##   oneDNN supports NHWC for feature maps (input, output),
##   but weight still needs to be of blocked format.
##   Still we can save reorders for feature maps.
input = input.to(memory_format=torch.channels_last)
model = model.to(memory_format=torch.channels_last)
output = model(input)
```

# Benefit of NHWC in Intel® Extension for PyTorch*



Operator optimizations

Memory/data layout optimizations

Graph optimizations

Optimal Optimizer

Mixed Precision

# Auto Mixed Precision (AMP)

# Python – Imperative Mode

- FP32

```python
import torch
import torchvision.models as models

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

import intel_extension_for_pytorch as ipex
model = model.to(memory_format=torch.channels_last)
model = ipex.optimize(model)
data = data.to(memory_format=torch.channels_last)

with torch.no_grad():
  model(data)
```

- BFloat16

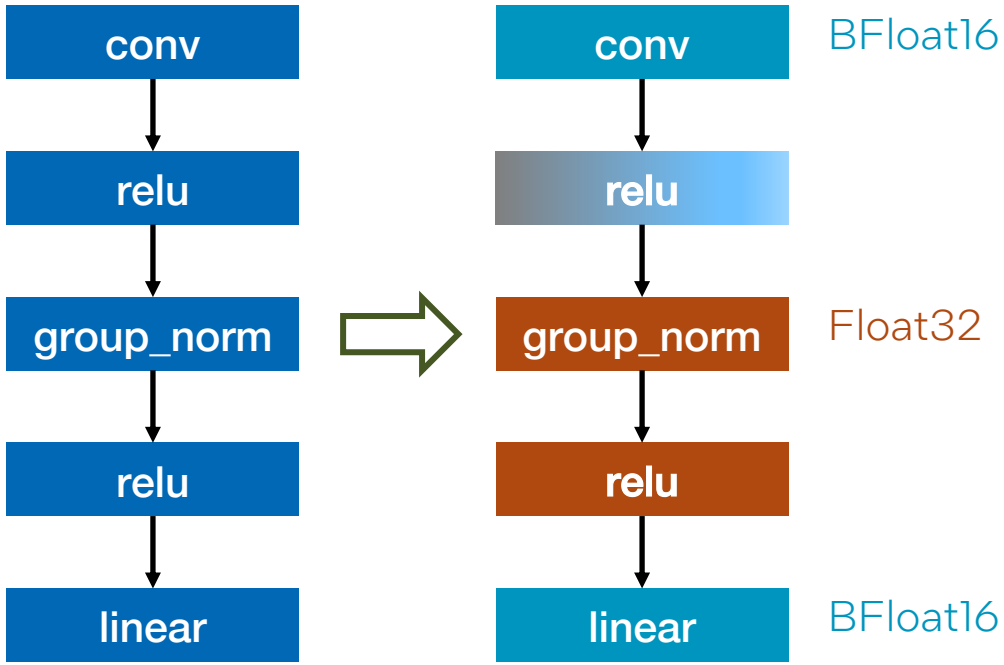
```python
import torch
from transformers import BertModel

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])

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

with torch.no_grad():
  with torch.cpu.amp.autocast():
    model(data)
```

https://intel.github.io/intel-extension-for-pytorch/1.11.0/tutorials/examples.html

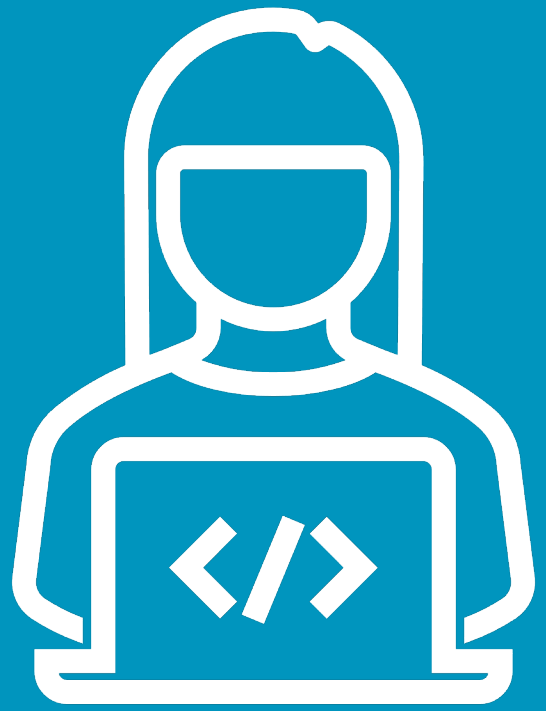
# Auto Mixed Precision (AMP)

```python
import intel_extension_for_pytorch as ipex
model = ipex.optimize(model, dtype=torch.bfloat16)

with torch.no_grad():
  with torch.cpu.amp.autocast():
    model(data)
```

- 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



# Demo with Intel Extension for PyTorch



## Notices and 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.
- You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.
- The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.
- No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that code included in this document is licensed subject to the Zero-Clause BSD open source license (OBSD), http://opensource.org/licenses/0BSD.
- No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.
- © 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.

intel®