# Leveraging Intel® oneDNN for AI Workload

## oneAPI – 가속 컴퓨팅을 개발하기 위한 스마트한 방식

2022. 01. 21.

MOASYS

# oneAPI Smart Development Series (2021)

1. Introduction to Intel oneAPI for HPC and AI-DL
   - https://www.allshowtv.com/detail.html?idx=474

2. Benchmarking the Performance of oneAPI on Heterogeneous Computing Platforms
   - https://www.allshowtv.com/detail.html?idx=660

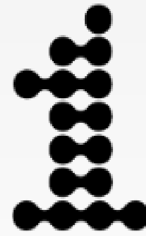
3. Optimization and GPU Offloading Workflow with Intel oneAPI
   - https://www.allshowtv.com/detail.html?idx=826

4. Leveraging Intel® oneDNN for AI Workload
   - TBD

# Contents


- oneAPI:
  - oneAPI Overview
  - Intel GPU Architecture Overview
    - Gen9
    - Gen11
    - Ponte Vecchio
- oneDNN:
  - Intel optimizations for TensorFlow
  - Intel optimizations for PyTorch
  - oneDNN primitives in action
    - Sigmoid activation
    - Edge detection convolution
- oneAPI new features (2022)
- Conclusion

# Overview of oneAPI for Deep Learning



oneAPI

Open, Standards-Based Unified Software Stack

Freedom from proprietary programming models

Full performance from the hardware

Piece of mind for developers

CPU & XPU - Optimized Stack

Applications & Services

Middleware, Frameworks & Runtimes

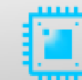
TensorFlow, PyTorch, mxnet, NumPy, XGBoost, OpenVINO ...

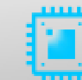
**Low-level Libraries**: oneMKL, oneDNN, oneDAL, oneVPL, oneTBB, oneCCL, oneDPL, Other Libraries

**Languages**: DPC++, Other Languages

Hardware Abstraction Layer: Level Zero

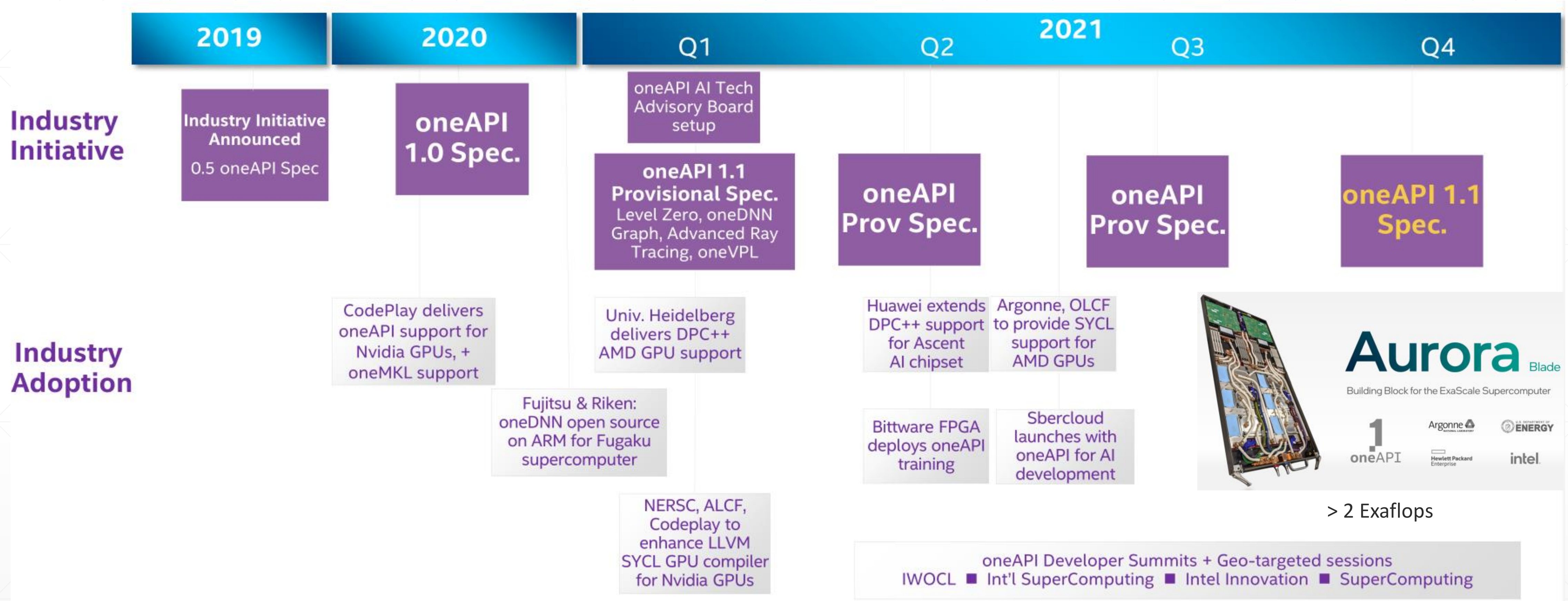
Compute Hardware:


CPU


GPU

- Support of popular deep learning frameworks: TensorFlow, Pytorch and mxnet
- Continuously evolving specifications for deep learning workload and ray-tracing libraries

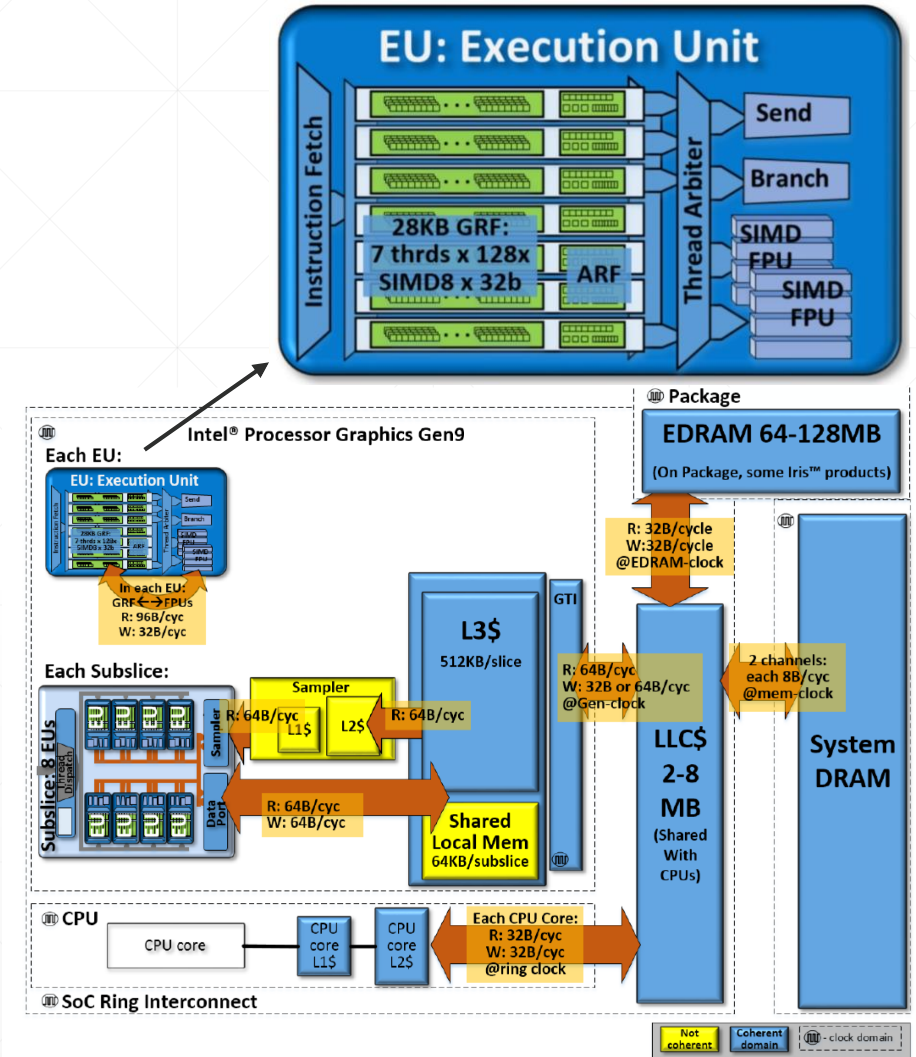
# oneAPI Industry Initiative Progress




- Cross-Vendor implementations:
  - ARM CPU: Fujitsu & Riken (Japan)
  - NVIDIA GPU: CodePlay (USA), NERSC (USA), Argonne National Lab (USA)
  - AMD GPU: Heidelberg Computing Center (Germany), Argonne National Lab (USA), Oak Ridge National Lab(USA)

# Intel GPU Architecture: Gen9

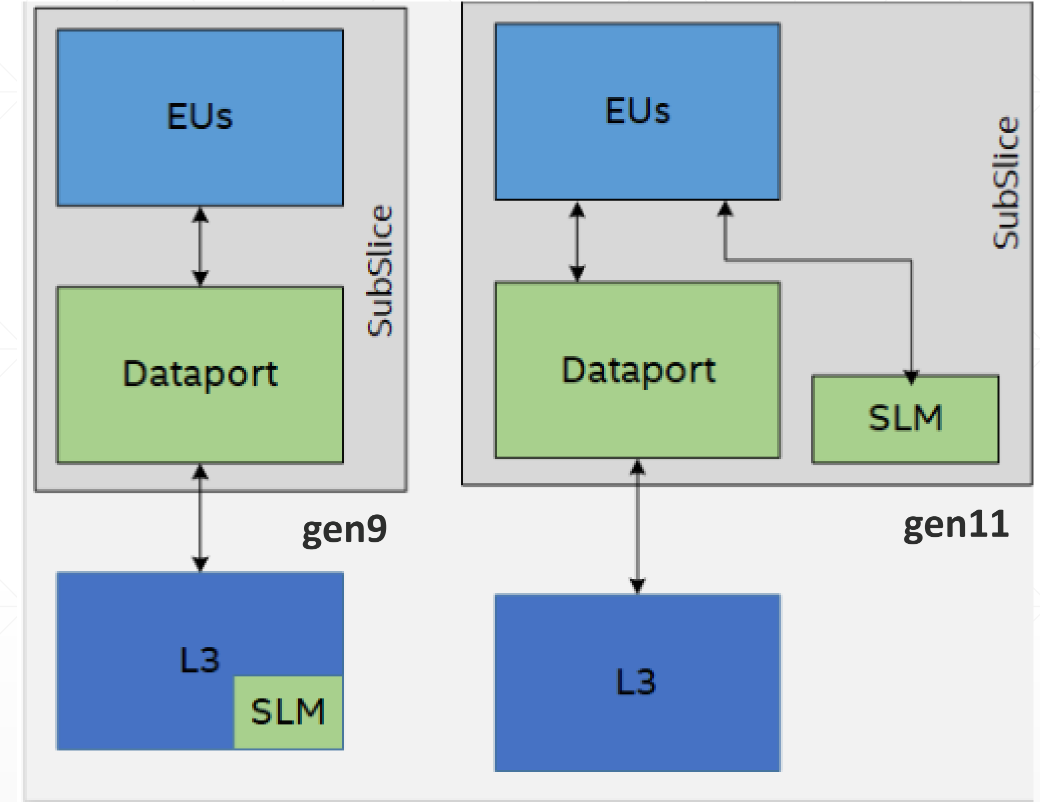
- SIMD Floating Point Unit (FPU):
  - 2 per EU
  - Simultaneous multiply and add (MAD) instruction per cycle
  - Half-precision floating point support for machine learning
- Execution Unit (EU):
  - Each EU has 7 hardware threads.
  - Each thread has 128 general purpose registers of 32 bytes
  - Each register corresponds to a SIMD8 of 32 bit, i.e. 4 bytes
- Sub-slice:
  - Arrays of 8 EU
  - Data port to slice's L3 cache and shared local memory
- Slice:
  - Arrays of 3 sub-slices
  - A sampler unit: read-only memory for images and textures
    - L1/L2 cache
  - Shared access to Last-level cache (LLC) of the host CPU




*Intel® Processor Graphics Gen9 Architecture Whitepaper*

# Intel GPU Architecture: Gen11

| Key Peak Metrics | Gen9 GT2 | Gen11 GT2 |
|---|---|---|
| **Slice Attribute** | | |
| # of Slices | 1 | 1 |
| # of Sub-Slices | 3 | 8 |
| # of Cores (EUs) | 24 (3x8) | 64 (8x8) |
| Single Precision FLOPs per Clock (MAD) | 384 | 1024 |
| Half Precision FLOPs per Clock (MAD) | 768 | 2048 |
| Register File Total | 672KB(=3x224KB) | 1792KB(=8x224KB) |
| # of Samplers | 3 | 8 |
| Point/Bilinear Texel's/Clock (32bpt) | 12 | 32 |
| Point/Bilinear Texel's/Clock (64bpt) | 12 | 32 |
| Shared Local Memory Total | 192KB(=3 x 64KB)* | 512KB(=8 x 64KB) |
| **Slice-Common Attributes** | | |
| Pixels/Clock (RGBA8) wo. Alpha Blend | 8 | 16 |
| Pixels/Clock (RGBA8) w. Alpha Blend | 8 | 16 |
| HiZ Zixel's/Clock | 64 | 128 |
| L3$ Cache | 768 KB | 3072 KB |
| **Geometry Attributes** | | |
| Primitive / Clock (backface Cull – strips) | 1 | 1 |
| Primitive / Clock (backface Cull – lists) | 0.67 | 0.67 |
| **Global Attributes** | | |
| GTI Bandwidth (Bytes/Clock) | R: 64<br>W: 32 | R: 64<br>W: 64 |
| LLC Configuration | 2-8MB | TBD |
| DRAM Configuration | 2x64 LPDDR3/DDR4 | 4x32 LPDDR4/DDR4 |




*Intel® Processor Graphics Gen11 Architecture Whitepaper*

- Gen11 consists of 64 EUs which increases the core computability to 2.87x over Gen9.
- EU has direct access to shared local memory (SLM):
  - SLM traffic does not interfere with L3 traffic, providing lower latency than Gen9
  - All kernel instances within a workgroup have access to 64KB SLM.

# Intel Architecture Day 2021: Ponte Vecchio

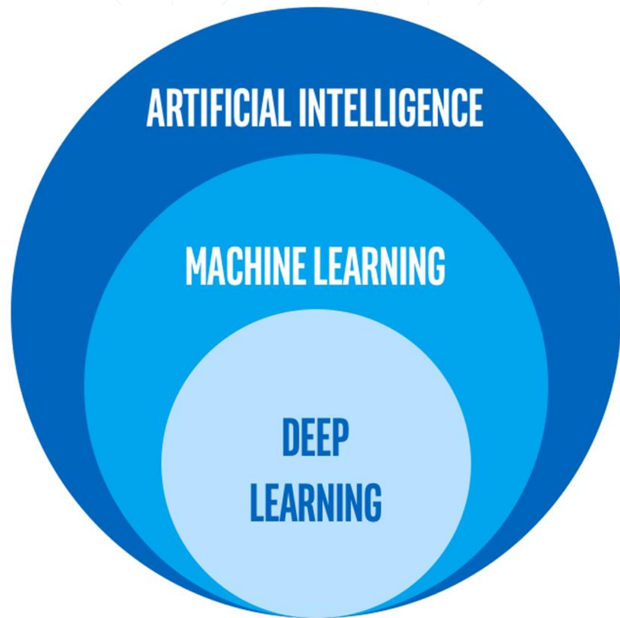
Xe-core
Compute Building Block of Xe HPC-based GPUs
8 Vector Engines, 512 bit per engine
8 Matrix Engines, 4096 bit per engine
Load / Store 512 B/CLK
Cache L1$/ SLM (512KB), I$

Xe HPC Stack
Up to
4 Slices
64 Xe - cores
64 Ray Tracing Units
4 Hardware Contexts
L2 Cache
4 HBM2e controllers
1 Media Engine
8 Xe Links

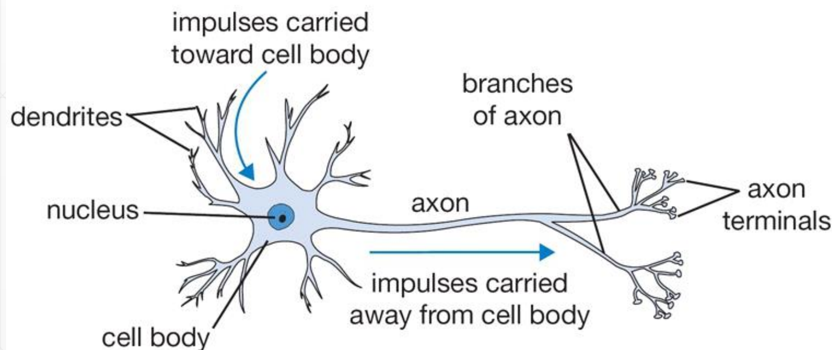
Ponte Vecchio Execution Progress
A0 Silicon Current Status
> 45 TFLOPS FP32 Throughput
> 5 TBps Memory Fabric Bandwidth
> 2 TBps Connectivity Bandwidth

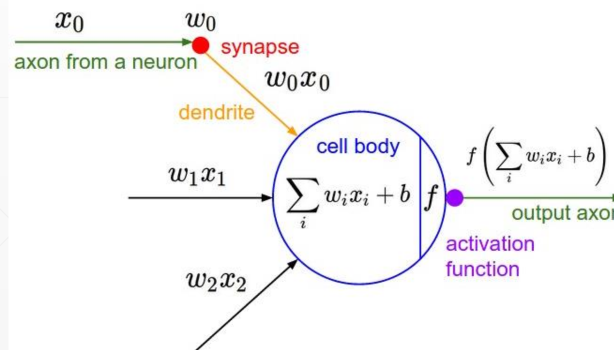
- Xe-core is the new sub-slice design for Xe-HPC architecture
- Each Xe-HPC GPU consist of
  - 4 slices, i.e. 64 Xe-cores in total
  - High bandwidth memory (HBM) interface with next gen Xeon processors-Sapphire Rapids (7 nm)
  - High-speed coherent fabric between multiple-GPUs provided by Xe links
  - Expected performance of FP32: 45 TFLOPS (V100: 15.7 TFLOPS, A100: 19.5 TFLOPS)

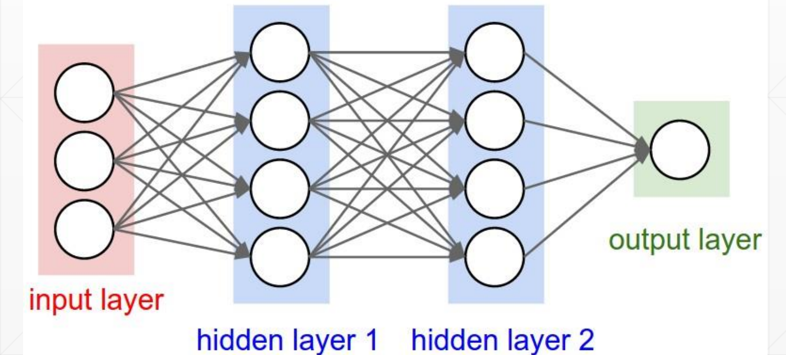
# From Artificial Intelligence to Machine Learning



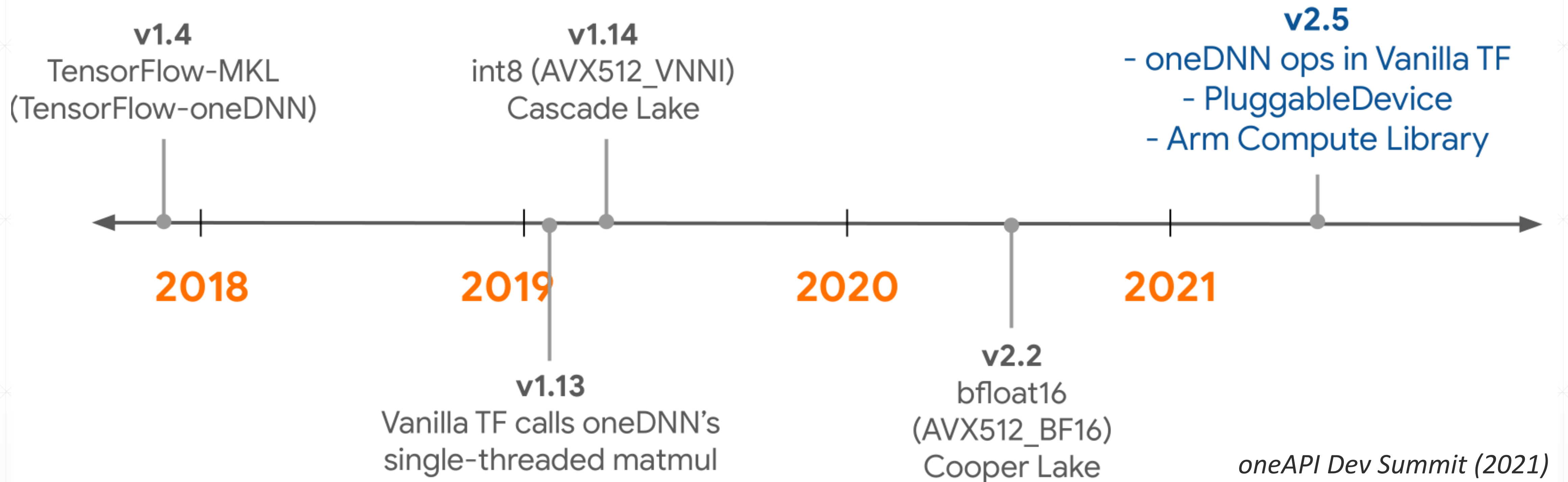

- Inspired by biological neural system:
  - Interconnected *neurons* communicated via electrical pulses
  - Dendrites: where input signal is received
  - Nucleus: where information is processed
  - Axons: where output signal is transported
- Mathematical model of a neuron:
  - $x_i$: input
  - $w_i$: weight associated with $x_i$
  - $b$: bias parameter
  - $f$: non-linear activation function
  - There can be zero or more layers between input and output layers



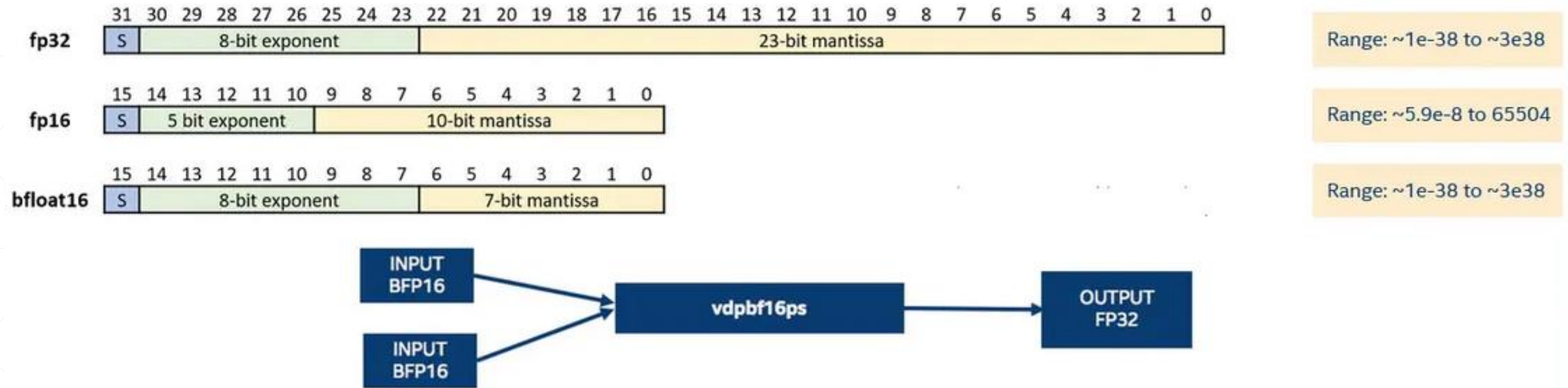



# Intel Optimized AI Framework: TensorFlow-oneDNN



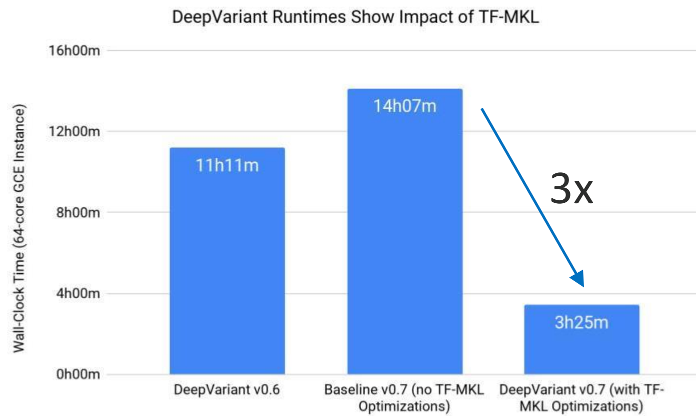

- Partnership between Intel and Google (oneDNN is formerly known as MKL-DNN)
  - Optimizations for 2nd and 3rd generation Intel® Xeon Processor
    - AVX512_VNNI: vectorized neural network instruction for low precision inference
    - AVX512_BF16: vectorized Brain FP16 for low precision training
  - Usage of oneDNN's optimized kernels for computational intensive operations (since v2.5)
  - Environment variable TF_ENABLE_ONEDNN_OPTS=1.

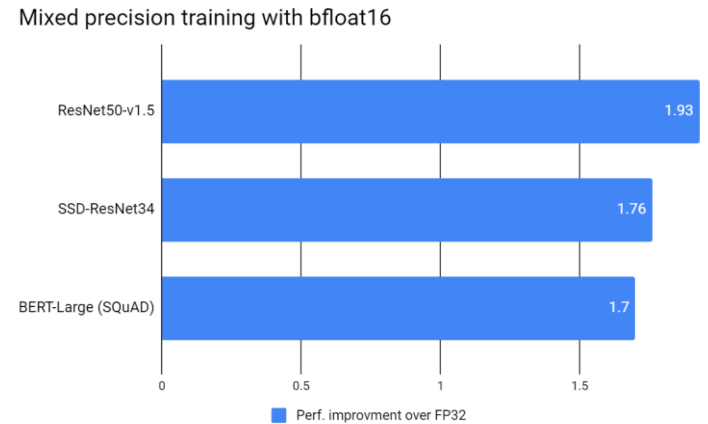
# Intel Deep Learning Boost Technology

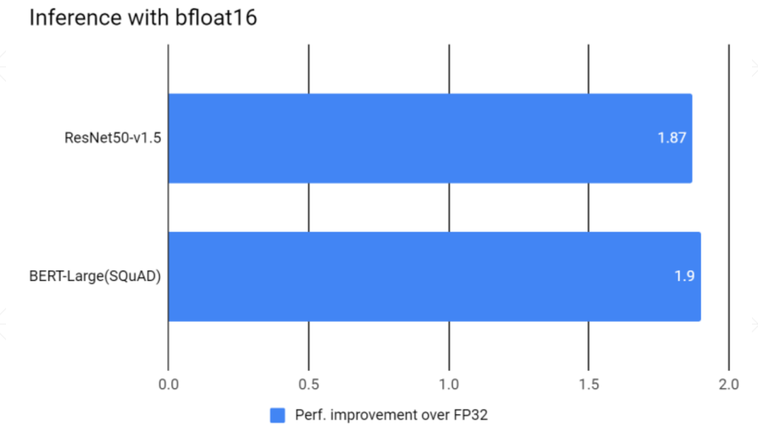



- BFP16 (Brain Float16)
  - New data type for improving the training speed and memory footprint of DNN training workload
  - Same range as FP32 (8-bit mantissa) but with reduced precision (7-bit mantissa)
  - Supported in TensorFlow, PyTorch via oneDNN
- AVX512_BF16 ( 3rd Gen Xeon and Xe-HPC)
  - VDPBF16PS: SIMD dot-product of a BF16 pairs and accumulate results into one FP32 register
  - VCVTNE2PS2BF16: convert 2 x FP32 register into one BF16 register
  - 2x training throughput is expected

# Intel Deep Learning Optimizations in TensorFlow




https://blog.tensorflow.org/2020/06/accelerating-ai-performance-on-3rd-gen-processors-with-tensorflow-bfloat16.html

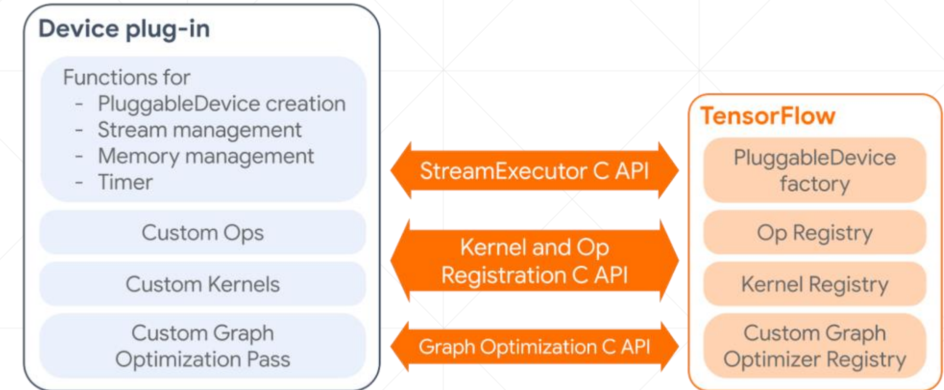
- Google Brain (2019)
  - DeepVariant: an open-source DNA sequencing analysis based on deep neural network
  - 3x reduction of running time using TF-MKL (oneDNN)
- TensorFlow's published benchmark with BF16 (2020)
  - Resnet50-v1.5: image classification model
  - SSD-ResNet34: object detection model
  - BERT-Large: text reading comprehension
  - Efficiency with respect to FP32:
    - Training: (1.7 ~ 1.9x)
    - Inference: (1.9x)

# TensorFlow Pluggable Device: Accelerator Support

```
>>> import tensorflow as tf
>>> tf.config.list_physical_devices()
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU'),
 PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU'),
 PhysicalDevice(name='/physical_device:XPU:0', device_type='XPU')]

>>> a = tf.random.normal(shape=[5], dtype=tf.float32)  # Runs on CPU
>>> b = tf.nn.relu(a)                                  # Runs on NVIDIA GPU

>>> with tf.device("/XPU:0"):                          # Device selection
...   c = tf.nn.relu(a)                                # Runs on Intel XPU
```




- Porting TensorFlow to DPC++/SYCL is a tremendously tasks.
  - In contrast, CUDA port of TensorFlow is matured and well-tested
- *Pluggable device* is proposed by Intel DL Team:
  - Communication between TensorFlow binary and devices via proper C API
  - Utilization of Intel XPU (CPU, GPU, FPGA) *without* changing source code
  - Default device selection based on a heuristically determined priority
  - Short development and easy maintenance

Scenario 2: Single PluggableDevice registered as a new device type, e.g., "XPU"

- Seamless integrations of any type of accelerator regardless of vendors and programming models
  - AMD GPUs, ARM CPUs, etc

# Intel Optimized AI Framework: PyTorch-oneDNN

```
import torch
# Step 1: Register IPEX optimizations
import intel_pytorch_extension as ipex
from my_models import SomeModel

# Step 2: Enable BF16 auto-mixed-precision
ipex.enable_auto_mixed_precision(mixed_dtype = torch.bfloat16)

# Step 3: Enable IPEX optimizations
model = SomeModel().to(ipex.DEVICE).eval()
model = torch.jit.script(model)
out   = mode(input.to(ipex.DEVICE))
```

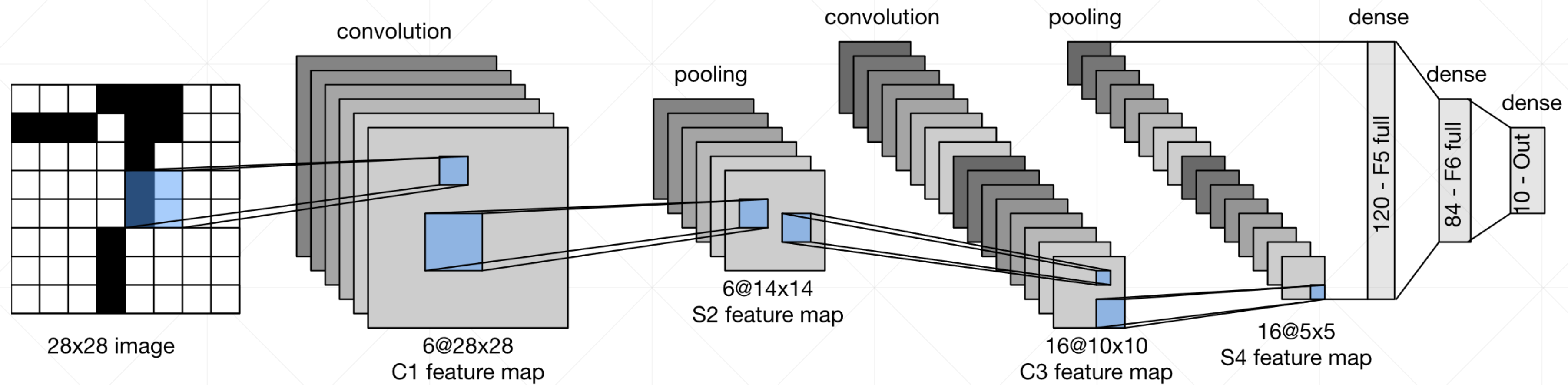
https://medium.com/pytorch/accelerate-pytorch-with-ipex-and-onednn-using-intel-bf16-technology-dca5b8e6b58f

| Deep Learning Training Model | # Model Instances / # Cores (on Xeon 3rd Gen Scalable Xeon processor) | BF16 vs. FP32 Speedup Ratio |
|---|---|---|
| DLRM | 1/28 | 1.55 |
| BERT-Large | 1/28 | 1.81 |
| ResNext-101-32x4d | 1/28 | 2.42 |

- Intel Extension for PyTorch (IPEX) proposed by Intel DL Team
  - Implementation of BF16 data type and drop-in optimized kernels from oneDNN
  - Easy-to-use python API:
    - Before IPEX, users most manually perform FP32 <-> BF16 type and layout conversion
    - *enable_auto_mixed_precision()*: automatic BF16 conversion
    - *to(ipex.DEVICE)*: model with vectorized AVX512_BF16/AVX512_VNNI instructions on Intel devices

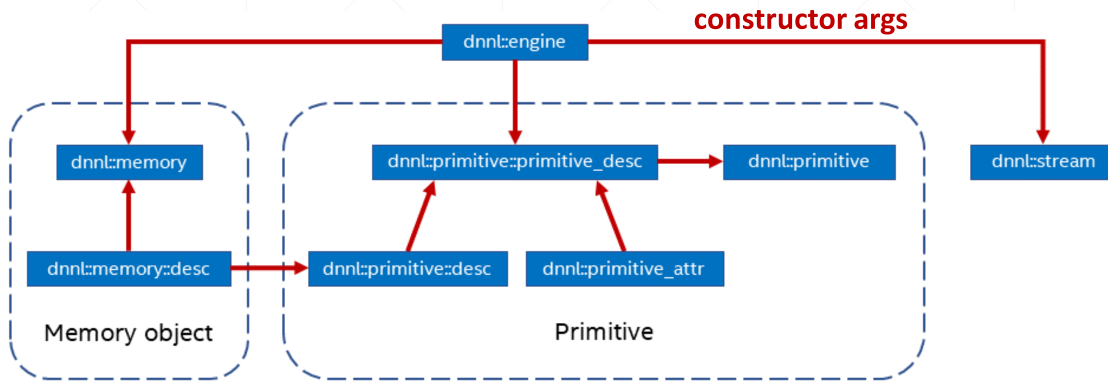
- PyTorch's published benchmark with BF16 (2021)
  - Deep Learning Recommended Model (DLRM)
  - Speed up w.r.t to FP32
    - Training: 1.5 ~ 2.4x

# Introduction to oneDNN




- oneAPI Deep Neural Network Library:
  - Open-source cross-platform performance library of deep learning (DL): https://github.com/oneapi-src/oneDNN
  - Low-levels primitives for for DL applications: convolutions, pooling, activation etc
  - Optimized for Intel® processors, Intel Gen9/Gen11 graphics and Xe graphics architectures
  - Experimental support for following architectures:
    - ARM 64-bit architecture (AArch64), NVIDIA GPU, OpenPOWER (PPC64), IBMz* (s390x), RSIC-V
  - Application enabled with oneDNN:
    - TensorFlow, PyTorch, MXNet
    - openVINO Toolkit (Intel), FlashLight (Facebook), ONNX Runtime (Microsoft), Matlab Deep Learning Toolbox

# Introduction to oneDNN: Basic Concepts




```
# Select GPU device
dnnl::engine engine(engine::kind::gpu, 0);

# Initialize a execution stream for GPU device
dnnl::stream stream(engine);
```

- Engine:
  - dnnl::engine: encapsulation of a computation device, CPU or GPU
- Memory objects:
  - dnnl:memory::desc: input/output tensor's dimension, data type (ex FP32) and memory format (ex NCHW)
  - dnnl::memory: handle of memory allocated on dnnl::engine
- Primitive:
  - dnnl::primitive_attr: quantization(INT8), floating-point math mode(FP32 -> FP16/BF16)
  - dnnl::primitive::desc: forward/backward, algorithm (GEMM/FFT/Winograd), padding, dilation etc
  - dnnl::primitive::primitive_desc: primitive generated for dnnl::engine
- Stream:
  - An object encapsulating to the context of execution on dnnl::engine (ex OpenCL command queue)

# Convert cuDNN to oneDNN: Sigmoid Activation (1/2)

```
// device selection
cudaSetDevice(0);
cudnnHandle_t handle;
cudnnCreate(&handle);

// tensor dimension
const int N = 1, C = 1, H = 1, W = 7;

// host allocation
std::vector<float> src(N*C*H*W);
std::vector<float> dst(N*C*H*W);

// input tensor descriptor
cudnnTensorDescriptor_t src_d;
cudnnCreateTensorDescriptor(&src_d);
cudnnSetTensor4dDescriptor(
    src_d,
    CUDNN_TENSOR_NCHW,
    CUDNN_DATA_FLOAT,
    N, C, H, W);

// device memory allocation
float *ds, *dd;
cudaMalloc(&ds, src.size()*sizeof(float));
cudaMalloc(&dd, src.size()*sizeof(float));

// copy src tensor to device memory
cudaMemcpy(ds, src.data(), src.size()*sizeof(float), cudaMemcpyHostToDevice);
```

```
// device selection
dnnl::engine engine(engine::kind::gpu, 0);
dnnl::stream stream(engine);


// tensor dimension
const int N = 1, C = 1, H = 1, W = 7;

// host allocation
std::vector<float> src(N*C*H*W);
std::vector<float> dst(N*C*H*W);

// input tensor descriptor
auto src_d = dnnl::memory::desc(
    {N,C,H,W},
     dnnl::memory::data_type::f32,
     dnnl::memory::format_tag::nchw);

// device memory allocation
auto src_mem = dnnl::sycl_interop::make_memory(
    src_d, engine,
    dnnl::sycl_interop::memory_kind::buffer);

// copy src tensor to device memory
write_to_dnnl_memory(src.data(), src_mem);
```

```
cudnnStatus_t cudnnSetTensor4dDescriptor(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnTensorFormat_t     format,
    cudnnDataType_t         dataType,
    int                     n,
    int                     c,
    int                     h,
    int                     w)
```

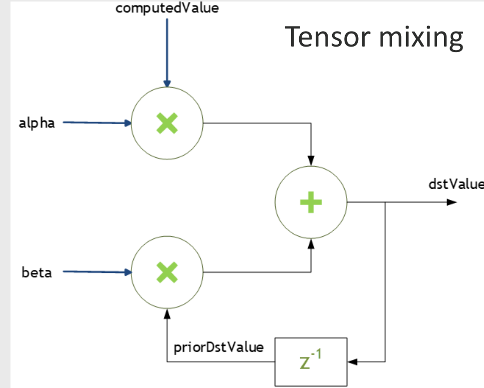
- CuDNN:
  - cudnnCreateTensorDescriptor()
  - cudnnSetTensor4Descriptor()
- oneDNN:
  - dnnl::memory::desc()

# Convert cuDNN to oneDNN: Sigmoid Activation (2/2)

```
// create activation descriptor
cudnnActivationDescriptor_t sigmoid_d;
cudnnCreateActivationDescriptor(&sigmoid_d);
cudnnSetActivationDescriptor(
    sigmoid_d,
    CUDNN_ACTIVATION_SIGMOID,
    CUDNN_NOT_PROPAGATE_NAN,
    0.0f);

// activation
cudnnActivationForward(
    handle,
    sigmoid_d,
    &alpha,
    src_d,
    ds,
    &beta,
    dst_d,
    dd);

// copy data to host
cudaMemcpy(dst.data(), dd, dst.size()*sizeof(float), cudaMemcpyDeviceToHost);
```




```
cudnnStatus_t cudnnSetActivationDescriptor(
    cudnnActivationDescriptor_t         activationDesc,
    cudnnActivationMode_t               mode,
    cudnnNanPropagation_t               reluNanOpt,
    double                              coef) (not required for sigmoid)
```

```
// operation descriptor
auto eltwise_d = dnnl::eltwise_forward::desc(
    prop_kind::forward_training,
    algorithm::eltwise_logistic,
    src_d,
    alpha, beta);

// sigmoid descriptor
auto eltwise_p = dnnl::eltwise_forward::primitive_desc(eltwise_d, engine);

// sigmoid object
auto sigmoid = dnnl::eltwise_forward(eltwise_p);

// activation
sigmoid.execute(
    stream,
    {
        {DNNL_ARG_SRC, src_mem},
        {DNNL_ARG_DST, dst_mem}
    }
);

// copy data to host
read_from_dnnl_memory(dst.data(), dst_mem);
```

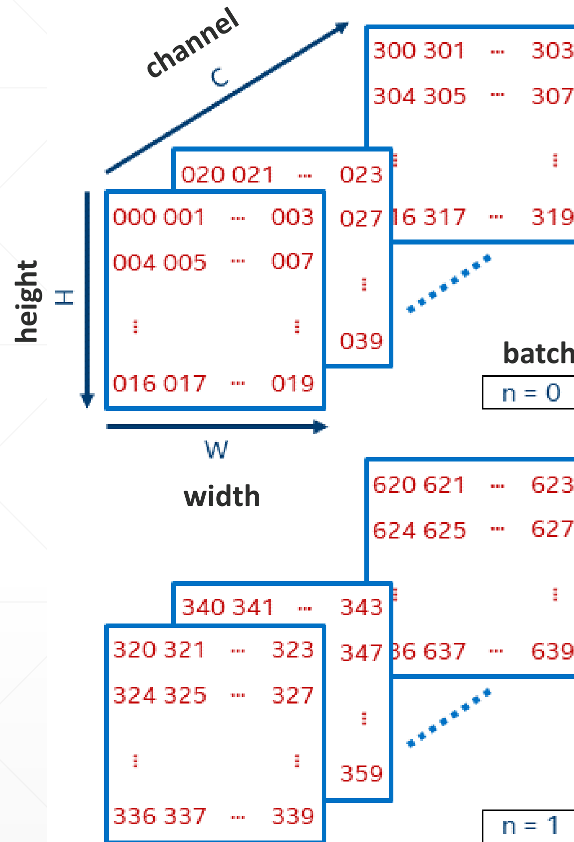
- CuDNN:
  - cudnnCreateActivationDescriptor()
  - cudnnSetActivationDescriptor()
- oneDNN:
  - dnnl::eltwise_forward::desc()
  - dnnl::eltwise_forward::primitive_desc()

# oneDNN: Memory Object Creation

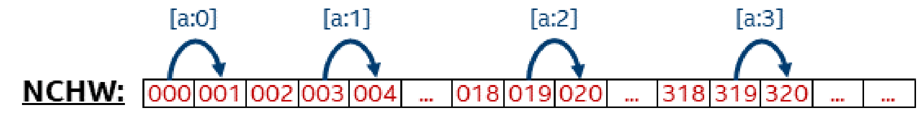
```
dnnl::mem::desc(
    const dims&     adims,
    data_type       adata_type,
    format_tag      aformat_tag,
    bool            allow_empty = false
);
```

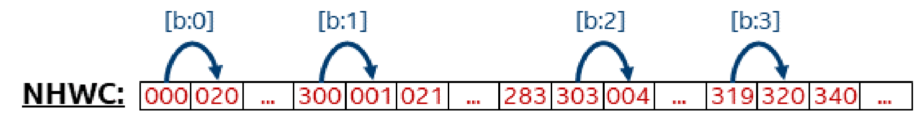
| Type | CPU | GPU |
|---|---|---|
| f32 | SSE4.1 | Gen9/11 |
| s8,u8 | AVX2 | Xe-LP |
| bf16 | Xeon® Gen2 | Xe-HPC |
| f16 | - | Gen9/11 |

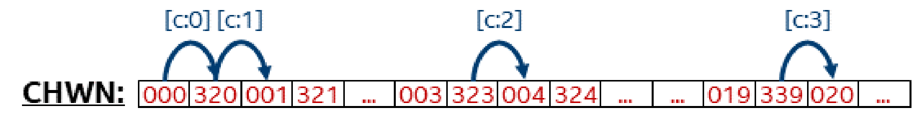
| Format | oneDNN |
|---|---|
| NCHW | memory::format_tag::nchw |
| NHWC | memory::format_tag::nhwc |
| CHWN | memory::format_tag::chwn |
| OIHW | memory::format_tag::iohw |




Physical data layout
NCHW, NHWC, and CHWN layouts



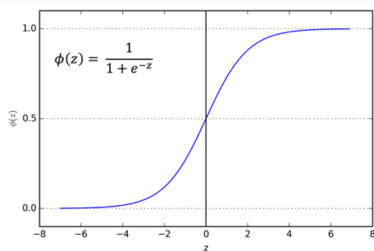




- Consistent layout definition between cuDNN and oneDNN
  - dnnl::data_type::f32 -> CUDNN_DATA_FLOAT
  - dnnl::memory::format_tag::nchw -> CUDNN_TENSOR_NCHW
- For NVIDIA device, use dnnl::sycl_interop::make_memory() to create buffer memory (USM not yet supported)

# oneDNN: Element-wise Activation Function

Primitive classes

```
enum kind
{
    undef
    reorder
    shuffle
    concat
    sum
    convolution
    deconvolution
    eltwise
    softmax
    pooling
    lrn
    batch_normalization
    layer_normalization
    inner_product
    rnn
    binary
    logsoftmax
    matmul resampling
    pooling_v2
    reduction
    prelu
};
```




Constructors & Methods

```
dnnl::eltwise_forward::desc(
    prop_kind               prop_kind,
    algorithm               algorithm,
    const memory::desc&     data_desc,
    float                   alpha = 0,
    float                   beta  = 0
);
```

Description of primitives, tied to data

```
dnnl::eltwise_forward::primitive_desc(
     const desc&       desc,
     const engine&     engine,
     bool              allow_empty = false
);
```

Description of primitives, tied to engine

```
dnnl::eltwise_forward(
     const primitive_desc&     desc,
);
```

Primitive objects with desired properties

```
dnnl::primitive::execute(
     const stream&                               astream,
     const std::unordered_map<int, emory>&       args
);
```

Activation via execute()

Propagation

```
enum prop_kind
{
    undef
    forward_training
    forward_inference
    forward_scoring
    backward_data
    backward_weights
    backward_bias
};
```

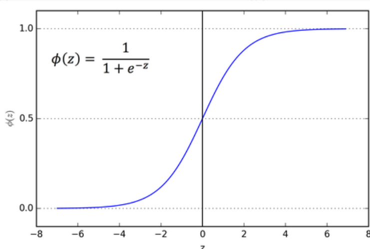
Algorithm

```
enum algorithm
{
    eltwise_relu
    eltwise_tanh
    eltwise_elu
    eltwise_square
    eltwise_abs
    eltwise_sqrt
    eltwise_swish
    eltwise_linear
    eltwise_bounded_relu
    eltwise_soft_relu
    eltwise_logsigmoid
    eltwise_mish
    eltwise_logistic
    eltwise_exp
    eltwise_gelu
    eltwise_gelu_tanh
    eltwise_gelu_erf
    eltwise_log
    eltwise_clip
    eltwise_clip_v2
    eltwise_pow
    eltwise_round
    eltwise_hardswish
    eltwise_relu_use_dst_for_bwd
    eltwise_tanh_use_dst_for_bwd
    eltwise_elu_use_dst_for_bwd
    eltwise_sqrt_use_dst_for_bwd
    eltwise_logistic_use_dst_for_bwd
    eltwise_exp_use_dst_for_bwd
    eltwise_clip_v2_use_dst_for_bwd
}
```

# oneDNN: Element-wise Activation Function

**Primitive classes**

```
enum kind
{
    undef
    reorder
    shuffle
    concat
    sum
    convolution
    deconvolution
    eltwise
    softmax
    pooling
    lrn
    batch_normalization
    layer_normalization
    inner_product
    rnn
    binary
    logsoftmax
    matmul resampling
    pooling_v2
    reduction
    prelu
};
```




**Constructors & Methods**

```
dnnl::eltwise_forward::desc(
    prop_kind               prop_kind,
    algorithm               algorithm,
    const memory::desc&     data_desc,
    float                   alpha = 0,
    float                   beta  = 0
);
```

Description of primitives, tied to data

```
dnnl::eltwise_forward::primitive_desc(
    const desc&        desc,
    const engine&      engine,
    bool               allow_empty = false
);
```

Description of primitives, tied to engine

```
dnnl::eltwise_forward(
    const primitive_desc&     desc,
);
```

Primitive objects with desired properties

```
dnnl::primitive::execute(
    const stream&                                astream,
    const std::unordered_map<int, memory>&       args
);
```

Activation via execute()

**Implementation**

```
auto eltwise_d = dnnl::eltwise_forward::desc(
    prop_kind::forward_training,
    algorithm::eltwise_logistic,
    src_d,
    alpha,
    beta
);

auto eltwise_p = dnnl::eltwise_forward::primitive_desc(
    eltwise_d,
    engine
);

auto sigmoid = dnnl::eltwise_forward(
    eltwise_p
);

sigmoid.execute(
    stream,
    {
        {DNNL_ARG_SRC, src_mem},
        {DNNL_ARG_DST, dst_mem}
    }
);
```

# Convert cuDNN to oneDNN: Edge Detection Convolution (1/3)

```
// device selection
// read and convert image to matrix via OpenCV
// src/dst tensor descriptor
...
// Laplacian kernel
const float laplacian[3][3] = { {1, 1, 1}, {1, -8, 1}, {1, 1, 1} };

// 4d conv tensor
float filter[3][3][3][3];

// copy laplacian to 4d filter tensor
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 3; k++)
            for (int m = 0; m < 3; m++)
                filter[i][j][k][m] = laplacian[k][m];

// create filter descriptor
cudnnFilterDescriptor_t filter_d;
cudnnCreateFilterDescriptor(&filter_d);
cudnnSetFilter4dDescriptor(
    filter_d,
    CUDNN_DATA_FLOAT,
    CUDNN_TENSOR_NCHW,
    3, 3, 3, 3);
```

```
cudnnStatus_t cudnnSetFilter4dDescriptor(
    cudnnFilterDescriptor_t    filterDesc,
    cudnnDataType_t            dataType,
    cudnnTensorFormat_t        format,
    int                        k,
    int                        c,
    int                        h,
    int                        w)
```

```
// device selection
// read and convert image to matrix via OpenCV
// src/dst tensor descriptor
...
// Laplacian kernel
const float laplacian[3][3] = { {1, 1, 1}, {1, -8, 1}, {1, 1, 1} };

// 4d conv tensor
float filter[3][3][3][3];

// copy laplacian to 4d filter tensor
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 3; k++)
            for (int m = 0; m < 3; m++)
                filter[i][j][k][m] = laplacian[k][m];

// create filter descriptor
auto filter_d = dnnl::memory::desc(
    {3,3,3,3},
     memory::data_type::f32,
     memory::format_tag::oihw);
```

- CuDNN:
  - cudnnCreateFilterDescriptor()
  - cudnnSetFilter4Descriptor()
- oneDNN:
  - dnnl::memory::desc()

# Convert cuDNN to oneDNN: Edge Detection Convolution (2/3)

```
 // create convolution descriptor
cudnnConvolutionDescriptor_t conv_d;
cudnnCreateConvolutionDescriptor(&conv_d);
cudnnSetConvolution2dDescriptor(
    conv_d,
    1, 1,
    1, 1,
    1, 1,
    CUDNN_CONVOLUTION,
    CUDNN_DATA_FLOAT
);

// creat convolution algorithm
cudnnConvolutionFwdAlgo_t conv_algo;
cudnnGetConvolutionForwardAlgorithm(
    handle,
    src_d,
    filter_d,
    conv_d,
    dst_d,
    CUDNN_CONVOLUTION_FWD_PREFER_FASTEST,
    0,
    &conv_algo
);
```

```
cudnnStatus_t cudnnSetConvolution2dDescriptor(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             pad_h,
    int                             pad_w,
    int                             u,
    int                             v,
    int                             dilation_h,
    int                             dilation_w,
    cudnnConvolutionMode_t          mode,
    cudnnDataType_t                 computeType)
```

```
// create convolution descriptor
auto conv_d = dnnl::convolution_forward::desc(
    prop_kind::forward_training,
    algorithm::convolution_direct,
    src_d,
    filter_d,
    bias_d,
    dst_d,
    {1,1},
    {1,1},
    {1,1}
);

// create convolution primitive descriptor
auto conv_p = dnnl::convolution_forward::primitive_desc(
    conv_d,
    engine
);


// create convolution object
auto conv = dnnl::convolution_forward(
   conv_p
);
```

- CuDNN:
  - cudnnCreateConvolutionDescriptor()
  - cudnnSetConvolution2Descriptor()
  - cudnnGetConvolutionForwardAlgorithm()
- oneDNN:
  - dnnl::convolution_forward::(desc|primitive_desc)

# Convert cuDNN to oneDNN: Edge Detection Convolution (3/3)

```
// get workspace size
size_t ws_size;
cudnnGetConvolutionForwardWorkspaceSize(
    handle,
    src_d,
    filter_d,
    conv_d,
    dst_d,
    conv_algo,
    &ws_size
);

// convolution
cudnnConvolutionForward(
    handle,
    &alpha,
    src_d,
    ds,
    filter_d,
    df,
    conv_d,
    conv_algo,
    dw,
    ws_size,
    &beta,
    dst_d,
    dd
);
```

```
// get workspace size
auto workspace_mem = sycl_interop::make_memory(
    conv_p.workspace_desc(),
    engine,
    sycl_interop::memory_kind::buffer
);

// perform convolution
conv.execute(
    stream,
    {
        {DNNL_ARG_SRC,       src_mem},
        {DNNL_ARG_WEIGHTS,   filter_mem},
        {DNNL_ARG_BIAS,      bias_mem},
        {DNNL_ARG_DST,       dst_mem},
        {DNNL_ARG_WORKSPACE, workspace_mem}
    }
);
```

- cuDNN:
  - cudnnGetConvolutionForwardWorkspaceSize(): memory required for convolution
  - workspace contains information required for backward propagation

# oneDNN: Convolution

| Primitive classes | Constructors & Methods | Propagation | Algorithm |
|---|---|---|---|

**Primitive classes**

```
enum kind
{
    undef
    reorder
    shuffle
    concat
    sum
    convolution
    deconvolution
    eltwise
    softmax
    pooling
    lrn
    batch_normalization
    layer_normalization
    inner_product
    rnn
    binary
    logsoftmax
    matmul resampling
    pooling_v2
    reduction
    prelu
};
```

**Constructors & Methods**

```
dnnl::convolution_forward::desc(
    prop_kind              prop_kind,
    algorithm              algorithm,
    const memory::desc&    src_desc,
    const memory::desc&    weights_desc,
    const memory::desc&    bias_desc,
    const memory::desc&    dst_desc,
    const memory::dims&    strides,
    const memory::dims&    padding_l,
    const memory::dims&    padding_r
);
```

```
dnnl::convolution_forward::primitive_desc(
    const desc&      desc,
    const engine&    engine,
    bool             allow_empty = false
);
```

```
dnnl::convolution_forward(
    const primitive_desc&    desc,
);
```

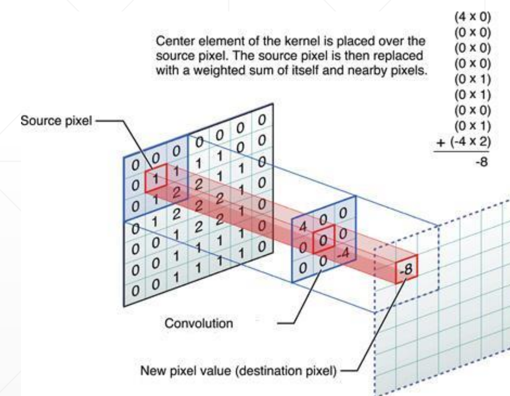
```
dnnl::primitive::execute(
    const stream&                              astream,
    const std::unordered_map<int, emory>&      args
);
```

**Propagation**

```
enum prop_kind
{
    undef
    forward_training
    forward_inference
    forward_scoring
    backward_data
    backward_weights
    backward_bias
};
```

**Algorithm**

```
enum algorithm
{
    convution_auto
    convolution_direct
    convolution_winograd
}
```




- src: N x IC x IH x IW
- filter: OC x IC x KH x KW
- dst: N x OC x OH x OW
- bias: OC
- padding (L): $PD_L$, $PH_L$, $PW_L$
- padding (R): $PD_R$, $PH_R$, $PW_R$
- stride: SD, SH, SW

$$\mathrm{dst}(n, oc, oh, ow) = \mathrm{bias}(oc) + \sum_{ic=0}^{IC-1}\sum_{kh=0}^{KH-1}\sum_{kw=0}^{KW-1} \mathrm{src}(n, ic, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L) \cdot \mathrm{weights}(oc, ic, kh, kw).$$

Here:

- $OH = \left\lfloor \frac{IH - KH + PH_L + PH_R}{SH} \right\rfloor + 1,$
- $OW = \left\lfloor \frac{IW - KW + PW_L + PW_R}{SW} \right\rfloor + 1.$

# oneDNN: Convolution

Primitive classes

```
enum kind
{
    undef
    reorder
    shuffle
    concat
    sum
    convolution
    deconvolution
    eltwise
    softmax
    pooling
    lrn
    batch_normalization
    layer_normalization
    inner_product
    rnn
    binary
    logsoftmax
    matmul resampling
    pooling_v2
    reduction
    prelu
};
```

Constructors & Methods

```
dnnl::convolution_forward::desc(
    prop_kind              prop_kind,
    algorithm              algorithm,
    const memory::desc&    src_desc,
    const memory::desc&    weights_desc,
    const memory::desc&    bias_desc,
    const memory::desc&    dst_desc,
    const memory::dims&    strides,
    const memory::dims&    padding_l,
    const memory::dims&    padding_r
);
```

```
dnnl::convolution_forward::primitive_desc(
    const desc&      desc,
    const engine&    engine,
    bool             allow_empty = false
);
```

```
dnnl::convolution_forward(
    const primitive_desc&    desc,
);
```

```
dnnl::primitive::execute(
    const stream&                                astream,
    const std::unordered_map<int, emory>&        args
);
```

Implementation

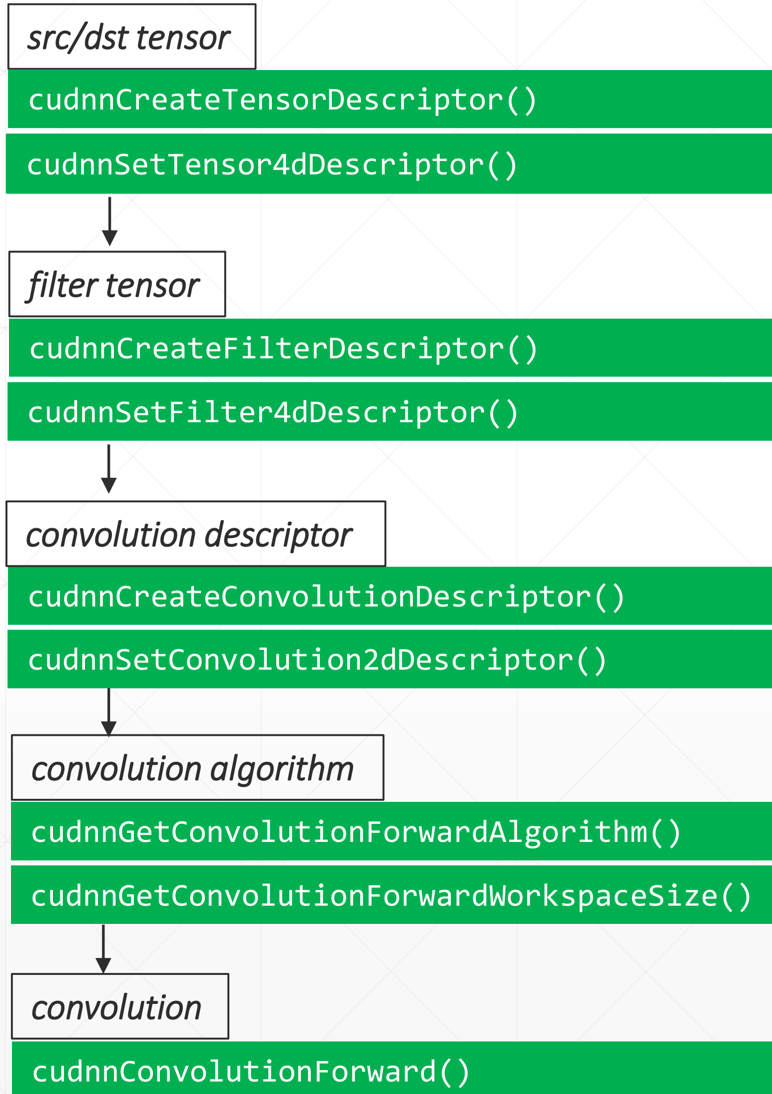
```
auto conv_d = dnnl::convolution_forward::desc(
    prop_kind::forward,
    algorithm::convolution_direct,
    src_d,
    filter_d,
    bias_d,
    dst_d,
    {1,1},
    {1,1},
    {1,1}
);
```

```
auto conv_p = dnnl::convolution_forward::primitive_desc(
    conv_d,
    engine
);
```

```
auto conv = dnnl::convolution_forward(
   conv_p
);
```

```
conv.execute(
        stream,
        {
            {DNNL_ARG_SRC,        src_mem},
            {DNNL_ARG_WEIGHTS,    filter_mem},
            {DNNL_ARG_BIAS,       bias_mem},
            {DNNL_ARG_DST,        dst_mem},
        }
    );
```

# cuDNN vs oneDNN: Convolution Work Flow Comparison




memory::desc()

sycl_interop::make_memory()

memory::desc()

sycl_interop::make_memory()

convolution_forward::desc()

convolution_forward::primitive_desc()

convolution_forward()

sycl_interop::make_memory()

execute()

# oneDNN: Primitives Benchmark

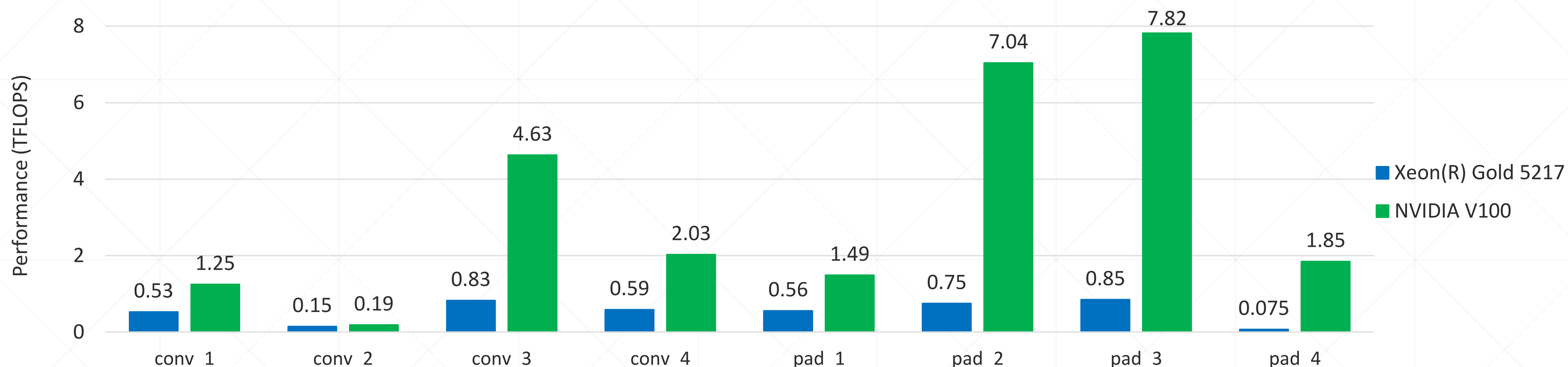
- oneDNN provides internal benchmark tool – *benchdnn*:
  - Measurement of performance of primitive operations on different hardwares
  - Which hardware is best for my workload ?
- Benchmark of convolution primitives

```
benchdnn                          \
    --conv                        \ # Convolution primitive
    --engine=gpu:0                \ # Use first GPU device
    --dir=FWB_B                   \ # Forward training with bias
    --cfg=f32                     \ # Single precision (FP32)
    --mode=p                      \ # Performance measurement mode
    --alg=direct                  \ # Direct algorithm
    --batch=inputs/conv/shapes_3d \ # input file
    --perf-template=csv           \ # output in csv format
    > conv_perf.dat
```

| | |
|---|---|
| FWD_B | dnnl_forward_training w/ bias |
| FWD_D | dnnl_forward_training w/o bias |
| FWD_I | dnnl_forward_inference |
| BWD_D | dnnl_backward_data |
| BWD_WB | dnnl_backward_weights w/ bias |
| BWD_W | dnnl_backward_weights w/o bias |
| BWD_DW | dnnl_backward |

  - Return status of *benchdnn*:
    - *PASSED:* test passed the validation
    - *SKIPPED:* test was not run and a reason is reported.
    - *FAILED:* test did not pass the validation with respect to reference result
    - *LISTED:* test was initialized but primitive descriptor was not created in this case, i.e dry run
    - *UNIMPLEMENTED:* test corresponds to unimplemented feature and treated as *FAILED*

# oneDNN Convolution Benchmarks




- Problem descriptors:
  - ic, io: input and output channel
  - id, ih, iw: input depth, height and width
  - kd, kh, kw: kernel depth, height and width
  - pd, ph, pw: front, top and left padding
  - n: descriptor name
  - _: optional delimiter between entries for readability.
- Limitations of NVIDIA backend:
  - bf16 is *not* yet implemented
  - https://github.com/oneapi-src/oneDNN/blob/master/src/gpu/nvidia/README.md

```
# inputs/conv/shapes_3d
ic16oc16_ih13kh3ph1_iw50kw3pw1_id10kd3pd1_n"3d_conv_pad:1"
ic64oc64_ih10kh3ph1_iw20kw3pw1_id15kd3pd1_n"3d_conv_pad:2"
ic256oc256_ih7kh3ph1_iw9kw3pw1_id11kd3pd1_n"3d_conv_pad:3"
ic256oc256_ih7kh1ph1_iw9kw1pw1_id11kd1pd1_n"3d_conv_pad:4"
ic16oc16_ih13kh3ph0_iw50kw3pw0_id10kd3pd0_n"3d_conv:1"
ic16oc16_ih13kh1ph0_iw50kw1pw0_id10kd1pd0_n"3d_conv:2"
ic256oc256_ih7kh3ph0_iw9kw3pw0_id11kd3pd0_n"3d_conv:3"
ic256oc256_ih7kh1ph0_iw9kw1pw0_id11kd1pd0_n"3d_conv:4“

# conv_perf.dat
tests:8 passed:8 skipped:0 mistrusted:0 unimplemented:0 failed:0 listed:0
total perf: min(ms):1.54983 avg(ms):1.55749
```

# oneAPI 2022 Update

- Compiler features:
  - Improved performance on 3rd Gen Intel® Xeon® Scalable processors (*Ice Lake*)
  - Implementation of more SYCL 2020 features for hardware accelerator:
    - https://www.intel.com/content/www/us/en/developer/articles/technical/sycl-2020-features-dpc-language-oneapi-c.html
  - New OpenMP 5.0 & 5.1 features for C and C++ applications
  - Support for Microsoft Visual Studio 2022 and expanded support for Microsoft Visual Code

- Library features:
  - Support of Random Number Generator (RNG) on Intel® GPUs via oneMKL
  - Support for video processing on Intel® Iris® Xe / Xe MAX via oneAPI Video Processing Library (oneVPL)
  - Support for multi-GPU programming via oneAPI Collective Communication Library (oneCCL)
  - Data Parallel Python for high-level development of accelerated computing on CPUs and GPUs

# oneAPI 2022 Update

- Intel® VTUNE Profiler features:
  - Support for 3rd Gen Intel® Xeon® Scalable processors
  - Analysis of under utilization of EU hardware threads on GPU
  - Analysis of bottleneck of data transfer between CPU and GPU
  - Parallel I/O analysis and latency optimization
  - Visualization of hot code paths using Flame Graph
  - Analysis of power consumption via CPU Throttle Analysis

- Intel® Advisor features:
  - GPU-to-GPU modeling to estimate performance gain
  - GPU optimization recommendations from roofline analysis
  - Enhanced source view analysis for Offload Modelling and GPU Roofline capabilities

# Conclusion

- Intel® DL boost and oneDNN provides best performance on Intel hardwares
  - BF16 data type support on popular deep learning frameworks:
  - TensorFlow: pluggable device
  - PyTorch: IPEX
- oneDNN for AI workload on multi-architectures
  - Porting from cuDNN to oneDNN is straightforward
  - Demonstration of sigmoid activation and convolution primitives
- oneAPI continues to expand in 2022:
  - oneVPL
  - oneCCL
  - Expansion and implementation of new features from SYCL standards
  - Improved GPU code profiling and analysis with Intel® VTune and Intel® Advisor
- Code migration, porting and DL consultant service: sales@moasys.com