Towards Cross-Platform Performance Portability of DNN Models using SYCL

Mehdi Goli
Codeplay Research
mehdi.goli@codeplay.com

Kumudha Narasimhan,
Ruyman Reyes,
Ben Tracy,
Daniel Soutar,
Svetlozar Georgiev
Codeplay
name.surname@codeplay.com

Evarist M Fomenko,
Eugene Chereshnev
Intel Corporation
name.surname@intel.com

***Abstract*—The incoming deployment of Exascale platforms with a myriad of different architectures and co-processors have prompted the need to provide a software ecosystem based on open standards that can simplify maintaining HPC applications on different hardware. Applications written for a particular platform should be portable to a different one, ensuring performance is as close to the peak as possible. However, it is not expected that key performance routines on relevant HPC applications will be performance portable as is, especially for common building blocks such as BLAS or DNN. The oneAPI the initiative aims to tackle this problem by combining a programming model, SYCL, with a set of interfaces for common building blocks that can be optimized for different hardware vendors. In particular, oneAPI includes the oneDNN performance library, which contains building blocks for deep learning applications and frameworks. By using the SYCL programming model, it can integrate easily with existing SYCL and C++ applications, sharing data and executing collaboratively on devices with the rest of the application. In this paper, we introduce a cuDNN backend for oneDNN, which allows running oneAPI applications on NVIDIA hardware taking advantage of existing building blocks from the CUDA ecosystem. We implement relevant neural networks (ResNet-50 and VGG-16) on native CUDA and also a version of oneAPI with a CUDA backend, and demonstrate that performance portability can be achieved by leveraging existing building blocks for the target hardware.**

***Index Terms*—Cross-performance Portability, Deep Neural Network, cuDNN, `oneDNN`, SYCL, High Performance Computing**

I. Introduction

As the High Performance Computing landscape moves towards the Exascale era, an increasing number of low level APIs, libraries and frameworks are being integrated to accelerate existing code bases. With the combination of HPC and AI disciplines using the same systems, the number of devices, libraries and interfaces can only increase. For example, the Aurora super computer at Argonne national labs [1] is one of the examples of Exascale systems being developed for accelerating the convergence of High performance computing and Artificial intelligence. These Exascale computers come with a support for different architectures and co-processors to achieve exa (10^{18}) operations per second. Thus, applications written for one platform should be portable to a different one without compromising on their performance. The Aurora HPC system will use an Intel-provided accelerator, but the majority of the existing HPC applications have traditionally relied on NVIDIA-based accelerators. Other systems being provisioned for Exascale programs across the world are using different sets of accelerators or combinations of chips (e.g. ARM and AMD among others).

Most of the high-performance applications depend on some common building blocks which usually use highly optimised libraries which target a particular architecture to obtain the performance required. Some of the common examples of these building blocks include any type of BLAS [2] or deep neural network libraries. Since these applications are tied to specific library interfaces, obtaining optimal performance on different architectures out of the box sometimes includes a significant effort on tuning those building blocks, or finding existing libraries that close that gap.

The oneAPI [3] initiative was introduced to tackle this problem by combining a programming model, SYCL [4], with a set of interfaces for common building blocks that can be optimised for (or by) different vendors to their target platforms. By including multiple libraries covering a large variety of building blocks together with a standard programming model, the oneAPI industry standard provides a platform for applications to be written against a single unified interface that can be implemented on different platforms.

`oneDNN` [5] is one of the open-source APIs part of oneAPI which provides an interface to interact with deep neural network operations. `oneDNN` interfaces with the SYCL programming model, helping developers integrate with existing C++ and SYCL applications, and with the rest of the oneAPI building blocks. Data can be passed across a variety of devices supported by oneAPI implementations and processed by `oneDNN` in the same or different devices without needing write different memory handling operations.

In this paper, we describe the integration of NVIDIA support into `oneDNN` by interfacing with cuDNN [6], a library with similar functionality available in the CUDA proprietary ecosystem. This allows running oneAPI applications on NVIDIA hardware whilst taking advantage of the existing

978-0-7381-1088-2/20/$31.00 ©2020 IEEE
DOI 10.1109/P3HPC51967.2020.00008

optimised libraries from the CUDA ecosystem. We implement relevant neural networks (ResNet-50 and VGG-16) using both native CUDA and `oneDNN` CUDA backend, and demonstrate that performance portability can be achieved by leveraging existing building blocks for the target hardware without much loss of performance.

The rest of the paper is organised as follows – Section II explains the related work in the area of deep neural networks and heterogeneous computing support. It also motivates the need for this work. Section IV discusses our approach to adding CUDA backend support for `oneDNN`. SectionV discusses the evaluation methodology used and performance results obtained. We finalize with some conclusions and future work opportunities in Section VI.

A. Contribution

This work contributes an open-source `oneDNN` backend for NVIDIA platform, describes its design and implementation, and provides a detailed mapping of the functionality between both libraries. Furthermore, extensions required to the SYCL programming model and the different implementations are described and justified.

The NVIDIA support for `oneDNN` is enabled by integrating the cuDNN library via SYCL. In particular

- we have introduced support for the SYCL interop task in `oneDNN` backend. The proposed SYCL interoperability task can be used as a generic API for enabling different vendor-specified DNN libraries in `oneDNN`.
- we have translated the `oneDNN` routines descriptor to cuDNN routines descriptor explicitly, to enable separating model construction from model execution.
- we have embedded the cuDNN execution model to the `oneDNN` execution model to seamlessly execute `oneDNN` routines on NVIDIA devices that enables cross-performance portability of the existing code.
- we have mapped all supported cuDNN routines to the `oneDNN` equivalent ones using the proposed SYCL interoperability task.

The performance evaluation details results that demonstrate the suitability of the approach.

II. Related Work

Deep neural network applications rely on three different approaches to achieve optimal performance. (a) use vendor-specific libraries written specifically for an architecture, (b) convert neural networks into computation graphs and use graph-based compilers to tune and generate the kernels and (c) hard code a neural network node or import different lower-level libraries to obtain performance.

There are several researches on the creation of abstraction to support portability across heterogeneous devices on various application domains. BLAS [7] has provided a general specification as a de-facto standard low-level routines for linear algebra libraries for both C and Fortran, allowing vendors to develop high-level optimized implementations yielding substantial performance portability on their particular hardware. However, with the emerge of GPGPU with dedicated memory space, an extra abstraction layer is required to conceal the memory creation and data transfer for supporting the portability of application across heterogeneous architectures. The Kokkos [8] and RAJA [9] expose a set of parallel pattern for expressing high-level abstractions which can be mapped onto a device at runtime to achieve performance portability across various architectures, which alleviates the difficulty of writing specialized code for each system. Intel provides MKL [10] for its linear algebra subroutine and MKL-DNN [11] for accelerating deep neural network routines for multi-core and many-core systems. Recently, MKL-DNN has also added support for obtaining high performance on Intel GPUs. ARM Compute Library [12] provides a set of optimized functions for linear algebra and machine learning for ARM devices. Similarly, cuBLAS [13] and cuDNN [6] provide optimized implementations for basic linear algebra subroutines and deep neural network routines, respectively for NVIDIA devices. AMD has a similar library called MIOpen [14], to provide highly optimised matrix multiplication and GEMM routines on AMD GPUs. Each of these libraries is optimized specifically for their target devices, and therefore, do not provide performance portability across platforms, and sometimes they are not even portable across the same vendor hardware generations.

Several projects like Glow [15], nGraph [16], MLIR [17] and Tensor Comprehensions [18] use a compiler-based approach to obtain optimised code for a specific hardware. The neural network model is represented as a graph and is lowered into one or more intermediate representations before generating an optimised kernel for a specific architecture. However, some of these frameworks miss out on utilising the highly optimised architecture-specific deep neural network libraries for architectures when they are available.

Frameworks like Caffe [19], TensorFlow [20], PyTorch [21] and TinyNN [22] provide developers with a simple way of integrating their neural network models and running them on specific architectures. Internally they also rely on either generating code of the model either using one of the vendor-specific libraries or using a graph compiler to optimise the model and generate their own kernel for the given hardware.

A. Research Gap

There are many libraries and frameworks highly optimized for different hardware platforms and architectures. Although there are several approaches towards creating device-agnostic deep neural network frameworks, performance portability across heterogeneous platforms remains a challenging issue.

This can be attributed to the fact that there is no common language to abstract out the memory models and the execution models from various heterogeneous devices. In other words, there is no glue code that can integrate different parts of the code written for different hardware/ architectures.

Adopting SYCL [4] as the unifying programming model across different hardware can be considered as a viable approach to develop a performance portable library targeting various hardware architectures while sharing the same library

Listing 1 construction and execution of `binary` primitive in `oneDNN`

```
void binary_op(void* a, void* b, void * c) {
    // 1) construction
    auto eng =
        engine(engine::kind::dnnl_any_engine, 0);
    auto dims = memory::dims {8, 7, 6, 5};
    auto dt = memory::data_type::f32;
    // memory description
    auto desc_A = memory::desc(dims, dt, tag::nchw);
    auto desc_B = memory::desc(dims, dt, tag::nchw);
    auto desc_C = memory::desc(dims, dt, tag::nchw);
    // regular operation description ctor
    auto op_desc
            = binary::desc(algorithm::binary_add,
                           desc_A, desc_B, desc_C);
    // primitive descriotion
    auto pd = binary::primitive_desc(op_desc, eng);
    // regular primitive ctor
    auto prim = binary(pd);

   // 2) execution
    auto mem_A = memory(desc_A, eng, a);
    auto mem_B = memory(desc_B, eng, b);
    auto mem_C = memory(desc_C, eng, c);

    auto strm = stream(eng);
    prim.execute(strm,
            {{DNNL_ARG_SRC_0, mem_A},
             {DNNL_ARG_SRC_1, mem_B},
             {DNNL_ARG_DST, mem_C}});
    strm.wait();
}
```

interface. Thus, vendors can use a common unifying interface (SYCL), to "glue-in" their optimised hardware-specific libraries for current and next generation of processors and accelerators. This abstraction is also very beneficial from the point of view of the developer, as there is little to no difference in performance even when the program is run across different architectures.

III. PRELIMINARIES

A. oneDNN

The `oneDNN` interface is based around the following key concepts: Primitives, Engines, Streams, and Memory Objects. It is designed to enable executing one or several primitives to process data in one or several memory objects [5]. Running a `oneDNN` based neural network is divided into two stages, the first step is constructing the primitives used in the model based on the description for a defined Engine, and then, given the input data provided in the form of Memory Objects, execute the model on a Stream associated with the engine. The neural network is described using the `oneDNN` C++ API. Listing 1 shows the construction and execution model for binary operation in `oneDNN`.

The description contains placeholders for data that will be added later when executing the graph in the second stage. Once a neural network has been defined, it can be executed several times with different input data.

B. cuDNN

The cuDNN interface model is composed of the following concepts: Operation description, operation execution, a handle object associated with a CUDA context for a selected device to orchestrate the resource allocations and the kernels execution itself [6].

Listing 2 shows the construction and execution model for binary operation in cuDNN. Similar to `oneDNN`, cuDNN separates the model construction from the model execution. In both `oneDNN` and cuDNN, the memory and operation descriptions are independent of the selected device. However, cuDNN provides C API for constructing and executing neural network operations. Unlike `oneDNN`, the construction of device, context, and stream in cuDNN are abstracted by the `handle` object.

A primitive is a functor object for encapsulating a particular computation such as forward convolution, backward LSTM computations, or a data transformation operation and can represent more complex fused computations such as a forward convolution followed by a ReLU. The most important difference between a primitive and a pure function is that a primitive can store state. `oneDNN` primitive description is bound to an Engine for a given context associated with a device. Therefore a model construction in `oneDNN` will be attached to the chosen engine. However, the same operation description and memory descriptions can be reused for constructing different primitives and memory objects respectively. In cuDNN, the primitive descriptions are not bound to a context, thus, the primitive and memory descriptions are bound to a context associated with the memory and primitive objects at the execution step via the cuDNN handle.

C. SYCL

SYCL is an open-standard programming model for accelerators that facilitates programming heterogeneous platforms using standard C++. In SYCL, a kernel is compiled statically and both the host and the accelerator code are in the same file, as opposed to OpenCL where the kernel source is built and load into the application at runtime. Developers program at a higher level than the native acceleration API, but always have access to lower-level code through integration with the native acceleration API using the different interoperability interfaces available. SYCL programming model executes kernels in the different accelerator devices available in the system by submitting *command groups* to SYCL queues. When submitting *command groups* to any queue, the data dependency access is tracked and a dependency graph to orchestrate execution is created. The runtime of the SYCL implementation is responsible for executing the different kernels on any device available in the system in the correct order based on a well-defined set of rules for dependency checking.

Listing 2 construction and execution of `binary` primitive in cuDNN

```
void binary(void *a, void *b, void *c) {
    // 1) construction
    int ndims = 4;
    int dims[ndims] = {8, 7, 6, 5};
    int strides[ndims] = {210, 30, 5, 1};
    float alpha = 1;
    float beta = 0;
    cudnnDataType_t dt =
        cudnnDataType_t::CUDNN_DATA_FLOAT;
    cudnnOpTensorDescriptor_t op_desc;

    // memory creation
    cudnnTensorDescriptor_t desc_A, desc_B, desc_C;
    // desc_A
    cudnnCreateTensorDescriptor(&desc_A);
    cudnnSetTensorNdDescriptor(desc_A, dt,
                ndims, dims, strides);
    // desc_B
    cudnnCreateTensorDescriptor(&desc_B);
    cudnnSetTensorNdDescriptor(desc_B, dt,
                ndims, dims, strides);
    // desc_C
    cudnnCreateTensorDescriptor(&desc_C);
    cudnnSetTensorNdDescriptor(desc_C, dt,
                ndims, dims, strides);

    // create and set the oeration descriptor
    cudnnCreateOpTensorDescriptor(&op_desc);
    cudnnSetOpTensorDescriptor(op_desc,
            cudnnOpTensorOp_t::CUDNN_OP_TENSOR_ADD,
            cudnnDataType_t::CUDNN_DATA_FLOAT,
            cudnnNanPropagation_t::CUDNN_PROPAGATE_NAN);

    //2) execution

    // create handle
    cudnnHandle_t handle;
    cudnnCreate(&handle);

    // execute model
    cudnnOpTensor(handle, op_desc, &alpha, desc_A, a,
            &beta, desc_B, b, &beta,
            desc_C, c);

    // relese the resources
    cudnnDestroyOpTensorDescriptor(op_desc);
    cudnnDestroyTensorDescriptor(desc_A);
    cudnnDestroyTensorDescriptor(desc_C);
    cudnnDestroyTensorDescriptor(desc_B);
    cudnnDestroy(&handle);
}
```

The original SYCL 1.2.1 version of the standard is based on OpenCL support, and offered only interoperability for said low-level API. The latest revision of SYCL 2020 includes support for programming models beyond OpenCL, such as CUDA, by offering a generalized backend model with a template-based API. This work, however, predates the SYCL 2020 standard and is therefore based on the SYCL 1.2.1 revision with a number of extensions that allow using CUDA support.

Interoperability with existing libraries

SYCL provides an interface that facilitates the SYCL runtime interaction with native objects for the supported backends [23]. A SYCL application is structured in three scopes: application scope, command group scope, and kernel scope. The kernel scope specifies a single kernel function to interface with native objects and is executed on the device. The SYCL interoperability with existing native objects is supported by either *host_task* or *interop_task* interfaces inside the command group scope. The command group scope specifies a unit of work that is comprised of a kernel function and accessors. The application scope specifies all other code outside of a command group scope. These three scopes are used to control the application flow and the construction and lifetimes of the various objects used within SYCL.

When using the *interop_task* interface, the SYCL runtime injects a task into the DAG that will execute on the host, but ensures dependencies are satisfied on the device. This allows the code on kernel scope to be written as-if working directly on the low-level API on the host, but producing side-effects on the device, such as existing API calls or libraries. The *interop_task* is an extension to the SYCL programming model contributed by Codeplay [24]. Figure 1 illustrates an example of a SYCL DAG that executes tasks concurrently on the CPU and the GPU and calls native CUDA libraries based on data dependencies.

The Interop task is based on Codeplay's extensions that enabled calling the OpenCL routines from the SYCL command group, and has now been integrated into the next version of the specification [24].

On the other hand, the *host_task* is a more complete mechanism, core part of SYCL 2020, that allows fine-grained control over dependencies, allowing mixing dependencies on the host and the device in the same command group. Feedback from this work was used on the creation of the *host_task* interface and helped its integration in the standard.

IV. Adding NVIDIA GPU support for oneDNN

In this work, we introduce CUDA support for the pre-existing SYCL backend of `oneDNN` by leveraging the existing building blocks in the CUDA ecosystem. In particular, we (1) introduce support for the SYCL interop task in the backend and (2) map the different DNN routines to the cuDNN equivalent ones.

For (1), we worked with the DPC++ and Codeplay's compiler teams to design and implement an extension to the SYCL

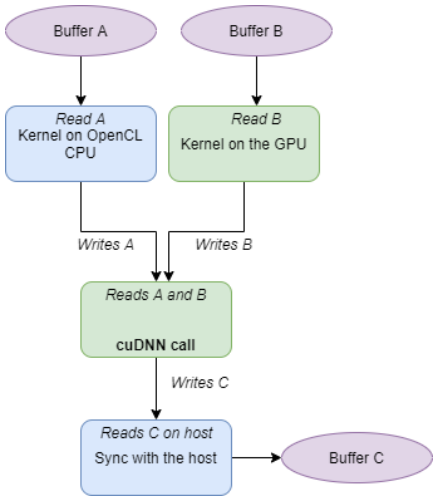

Fig. 1. Dependency graph formed by executing two kernels and then using the *interop_task* to execute a call to cuDNN when the kernels have completed. Blue represents operations executed by the host CPU and green on an NVIDIA GPU. The SYCL runtime uses the dependency information provided in the command groups to execute the kernels and library calls in the correct order.

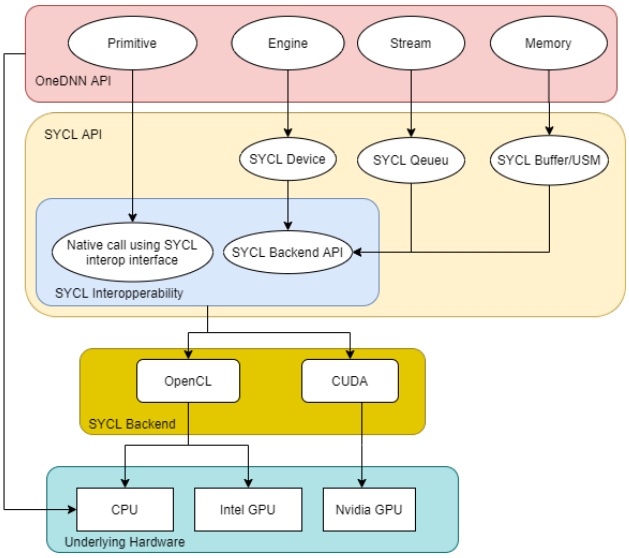

Fig. 2. Architectural view of mapping `oneDNN` to SYCL

specification that allows calling native CUDA code from inside a SYCL command group.

To accomplish (2), a deep analysis of the functionality of both libraries was required. Unlike other performance building blocks from other domain areas, there is no standard, or de-facto standard, for interfacing with DNN libraries. This means that, although the functionality between libraries can be similar, the actual interfaces or the acceptable input parameters can vary greatly.

A. SYCL backend for `oneDNN`

Figure 2 represents the mapping architecture of `oneDNN` for SYCL for CUDA and OpenCL backends. The `Engine` class wraps one of the devices supported by the SYCL backend. When using the DPC++ compiler from Intel [25], this can be host, OpenCL or CUDA devices.

The `oneDNN` architecture relies on the following components:

- *Engine* represents an abstraction of the computational device. The engine wraps one of the computational devices supported by SYCL backend. In the current version of the oneAPI Data Parallel C++ compiler [26] this can be either host, OpenCL, or CUDA. An engine also encapsulates a context associated with the device. The context is used to create a stream that launches a primitive execution.
- *Primitive* encapsulates a single unit of DNN computation, such as a convolution or pooling operations. Each operation is at least one SYCL command group, calling either an OpenCL or SYCL kernel in those backends natively implemented in `oneDNN` or the SYCL interoperability for those backends where `oneDNN` interacts with an existing library, such as the cuDNN example presented here.
- *Stream* encapsulates the execution context tied to a particular engine, orchestrating the execution of different Primitives on the associated devices. Each Stream runs in a particular Engine which holds the execution context for a device.
- *Memory* represents the memory allocated for a specific engine. In the SYCL backend, a `oneDNN` memory object maps to a SYCL buffer or a USM allocation. When the SYCL buffer is used, the data dependencies among the primitives are handled directly by the SYCL graph scheduler, enabling asynchronous kernel execution. On the other hand, when Memory is allocated using a USM allocation, the Memory object stores all the events related to the execution of the command groups.

The SYCL device, context, queue, and memory allocations, are automatically mapped to the underlying device, context, stream, and memory of the underlying SYCL device. The interaction with device-native objects (e.g. OpenCL memory objects) is handled by the SYCL interoperability interface.

B. Enabling CUDA on the `oneDNN` SYCL backend

Integrating CUDA on the `oneDNN` SYCL backend relies mostly on the existing SYCL with CUDA support of the underlying SYCL implementation. In this work, we rely on the DPC++ CUDA backend contributed by the authors, which uses the SYCL 1.2.1 interface with selective SYCL-2020 features to expose the CUDA interoperability objects, for example, for a SYCL queue we can obtain the underlying native CUDA stream object. This allows us to interact with the native CUDA libraries, cuDNN in this case.

Comparing the listing 1 with listing 2, shows that a neural network operation in both cuDNN and `oneDNN` can be divided into three sections: operation description, operation execution, and operation scheduler. Therefore, enabling the

CUDA backend for oneDNN requires mapping of the above three mentioned operations from oneDNN to cuDNN.

1) Mapping oneDNN description to cuDNN description: In cuDNN, the description of the graph is also separated from the execution as in oneDNN.

At the construction step, the oneDNN primitive descriptions are translated to cuDNN descriptions. The separation of primitive construction from primitive execution eliminates the overhead of neural network model construction as a model can be built once but executed several times. Algorithm 1 represents the construction of cuDNN operations from oneDNN primitive descriptions. In this process, the cuDNN equivalent types, tensor descriptions, and an operation description are created from those provided in oneDNN.

2) Mapping oneDNN Engine to cuDNN handle: cuDNN requires *handle* to set the device and the context associated with the device, allocate temporary resources, create stream for launching (sub-)kernels and track the (sub-)kernels dependencies. However, in oneDNN setting the device and its associated context along with the resource allocation are handled by the engine while launching and tracking the kernel dependencies are handled by the stream. The stream creation in oneDNN is a light-weight operation since it is bounded to the context associated with the Engine. However, the creation of an engine is a costly operation as it requires to construct a context for a given device. The creation of a cuDNN handle is costly [27]. To avoid performance overhead, one handle per thread and per context should be created. Given that, for each context, one instance of cuDNN *handle* is allocated, the cuDNN *handle* is encapsulated inside the oneDNN *Engine* construction. Since there is one engine per each stream, different primitives submitted to different streams with the same context use the same cuDNN *handle*.

Algorithm 1 meta-data translation: oneDNN $\rightarrow$ cuDNN

function NATIVE_DESC($Desc$)
  Initialize $cudnnDesc$
  // Converting all types used in a primitve
  for all $i \in size(Desc.Types)$ **do**
    $cudnnDesc.Types[i] \leftarrow Desc.Types[i]$
  end for
  //create cuDNN tensor descriptor from oneDNN tensor descriptor
  for all $i \in size(Desc.Tensors)$ **do**
    $cudnnDesc.Tensors[i] \leftarrow Desc.Tensors[i]$
  end for
  //create cuDNN operation descriptor from oneDNN primitive descriptor
  $cudnnDesc.Op \leftarrow Desc.primitive$
  //return cuDNN meta-data
  return $cudnnDesc$
end function

3) Mapping cuDNN execution to SYCL Execution Model: Algorithm 2 shows the mapping of cuDNN backend to the three different scopes of SYCL execution model. Interfacing with the native cuDNN objects is only valid inside the *interop_task*. The CUDA context used for creating a cuDNN handle must be the active CUDA runtime context when executing the cuDNN function. Therefore, the $ScopeContextHandler$ is called before cuDNN function invocation to guarantee the context activation of the cuDNN *handle*.

Algorithm 2 cuDNN invocation from SYCL

Kernel Scope: Interfacing with the cuDNN native objects

function NATIVE_OP($ih, accs, descs, engine$)
  // setting CUDA active context to cudnn context
  $ScopeContextHandler(engine)$
  // retrieving native CUDA memory from sycl accessor
  for all $i \in size(accs)$ **do**
    $cuMems[i] \coloneqq get\_native < void* > (ih, accs[i])$
  end for
  // retrieving the cuDNN handle
  $h \coloneqq get\_handle(engine)$
  // call to cudnn function
  $status \coloneqq cudnn\_func(h, descs, cuMems)$
  return $status$
end function

Command Group Scope: extracting the accessors to invoke interoperability with cuDNN

function INTEROP_CALL($cgh, buffs, opDesc, engine$)
  // extracting sycl accessor from sycl buffers
  for all $i \in size(buffs)$ **do**
    $accs[i] \coloneqq buffs[i].get\_access(cgh)$
  end for
  // invoking cuda kernel
  cgh.interop_task([](interop_handler ih){
    NATIVE_OP($ih, accs, descs, engine$))
  });
end function

Application Scope: specialization of oneDNN execution function

function EXECUTE($mems, descs, stream$)
  // retrieving SYCL queue from oneDNN Stream
  $sycl\_queue \coloneqq stream.get\_queue()$
  // retrieving the engine containing cudnn handle
  $engine \coloneqq stream.get\_engine()$
  // retrieving sycl buffers from oneDNN memory
  for all $i \in size(mems)$ **do**
    $buffs[i] \coloneqq get\_buffer(mems[i])$
  end for
  // submitting interop task to sycl queue
  sycl_queue.submit([](handler &cgh){
    INTEROP_CALL($cgh, buffs, descs, engine$))
  });
end function

V. Performance Evaluation

A. Benchmark Models

To demonstrate the performance of the proposed model, in this work we present results with the following popular neural network models:

VGG-16 is a network introduced in 2014. It consists of 16 layers of 3x3 convolutions interleaved with max pool layers [28].

ResNet-50 is a 50 layer deep network introduced in 2015. It consists of number of blocks each consisting of - 1x1 convolution,3x3 convolution and 1x1 convolution, in that order [29].

The above model is chosen to cover cross-performance portability of `oneDNN` across variety of device ranges. ResNet-50 has 23 million parameters and requires less storage space in comparison with VGG-16 with 138 million parameters. This feature makes ResNet-50 more suitable for low-end and embedded GPU, in comparison with VGG-16 which is more suitable for high-end GPU.

B. Hardware setup

The performance evaluation is carried out on a range of hardware to show platform portability of `oneDNN` across various types of devices.

Titan RTX: a high-end NVIDIA Turing architecture with a boost clock frequency of $1770MHz$. The CPU used to launch the kernels is Intel(R) Core (TM) i7-6700K CPU @ 4.00GHz. The CUDA version used is 10.2 and the cuDNN library version used is 7.6.5.

GeForce GTX 1050 Ti: a low-end NVIDIA pascal architecture with a boost clock frequency of $1392MHz$. The CPU used to launch the kernels is Intel(R) Core (TM) i7-9700K CPU @ 3.6GHz. The CUDA version used is 10.2 and the cuDNN library version used is 7.6.5.

Intel HD Graphics 630 integrated GPU: an integrated GPU with a base frequency of 350 MHz and maximum dynamic frequency of 1.2GHz. Intel i7-9700k CPU was used to launch the kernels.

NVIDIA's backend targets the CUDA Driver API to guarantee low-level control of the platform.

C. Performance comparison with cuDNN on TitanRTX

We compare the performance of `oneDNN`'s NVIDIA backend with cuDNN benchmarks which have been implemented from scratch using the CUDA Runtime API.

Figure3 and Figure4 shows the performance of `oneDNN`'s NVIDIA backend with cuDNN. The $x-axis$ shows the different batch sizes used for the performance evaluation and $y-axis$ shows the number of images processed by the network in a second. For VGG-16(Fig3), the performance of `oneDNN`'s NVIDIA backend is within 1% ($0.2\%-0.7\%$) of direct cuDNN for batch sizes greater than 8. However for smaller batch sizes it is between $2\% - 5\%$. Table I, shows that the difference in the execution time for small sizes is less than $0.5ms$. Thus, `oneDNN`'s NVIDIA backend performs well for VGG-16 type networks.

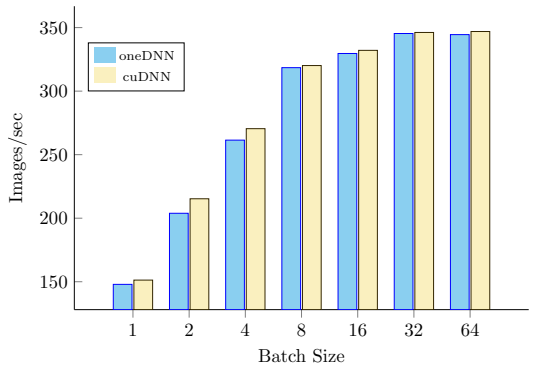

Fig. 3. Performance comparison between `oneDNN` and cuDNN for VGG-16 model with different batch sizes on NVIDIA GPU TITAN RTX

TABLE I
Execution time of VGG-16 model in milliseconds on NVIDIA GPU Titan RTX

Batch Size	oneDNN	cuDNN
1	6.76	6.61
2	9.81	9.29
4	15.3	14.79

Figure 4 compares `oneDNN`'s NVIDIA backend with direct cuDNN for the ResNet-50 model. There is a performance drop of about 27% for batch size 1 and 1% for batch size 64. To understand the drop in performance seen by the ResNet-50 model, the average execution time of the network is provided in Table II. As shown in the table, there is a difference of about $1.8ms$ between `oneDNN`'s NVIDIA backend and the direct cuDNN implementation irrespective of the batch size used to run the network.

The average performance of each layer/operation inside the network is shown in Figure5. The $x-axis$ shows the different primitives (layers) and $y-axis$ shows the average execution time in milliseconds. A batch size of 16 was used in this comparison. As stated in the Figure5, the compute bound operations have lower performance drop as compared

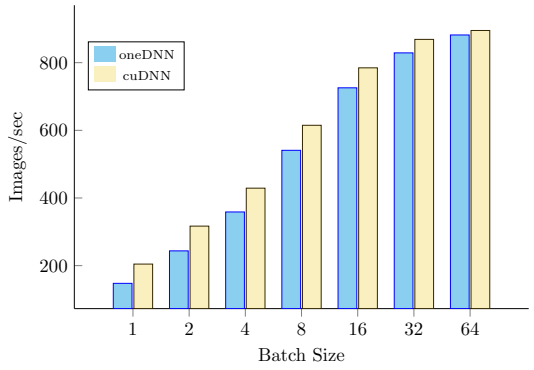

Fig. 4. Performance comparison between `oneDNN` and cuDNN for ResNet-50 model with different batch sizes on NVIDIA GPU TITAN RTX

TABLE II
EXECUTION TIME OF RESNET-50 MODEL IN MILLISECONDS ON NVIDIA GPU TITAN RTX

Batch Size	oneDNN	cuDNN
1	6.77159	4.88737
2	8.20827	6.31255
4	11.152	9.32324
8	14.7896	13.0104
16	22.0494	20.3903
32	38.605	36.8317
64	72.5731	71.4991

TABLE III
NVPROF RESULTS FOR SOFTMAX OPERATION USING ONEDNN WITH MB=16

Type	Avg time per iter	No. iters	Name
GPU activities:	4.852us	15001	softmax_fw_kernel
API calls:	5.2140us	15001	cudaLaunchKernel
	3.12us	45004	cuEventSynchronize

to bandwidth bound operations. In particular, the maximum performance difference is seen in softmax ($> 50\%$).

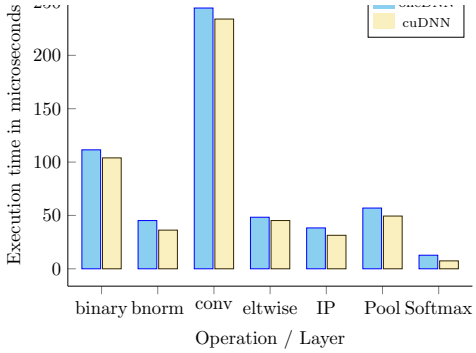

Fig. 5. Average execution time comparison for different layers in ResNet-50 on NVIDIA GPU TITAN RTX

Further, the *nvprof* on the softmax layer for both oneDNN and direct cuDNN implementations shows the the cost of kernel execution is comparable with the cost of launching application. Table III and Table IV state that the kernel execution time is almost the same in both cases. However, the API functions show that there are 3 times more calls to *cuEventSynchronize* for oneDNN than for the direct cuDNN implementation. This is due to the extra barrier required by the interop-task implementation, which adds an additional extra synchronization point. Moreover, the kernel launch time is higher than that of cuDNN. This is due to the difference between calling cuDNN via CUDA rutime versus running

TABLE IV
NVPROF RESULTS FOR SOFTMAX OPERATION USING CUDNN WITH MB=16

Type	Avg time per iter	No. iters	Name
GPU activities:	4.8390us	15001	softmax_fw_kernel
API calls:	3.9300us	15001	cudaLaunchKernel
	5.8400us	15001	cuEventSynchronize

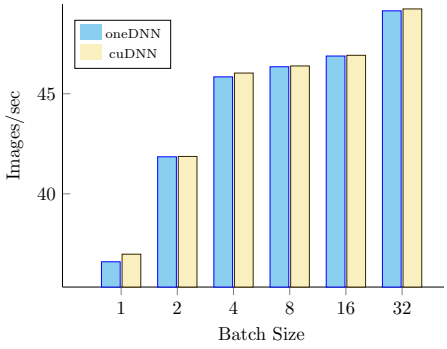

Fig. 6. Performance comparison between oneDNN and cuDNN for VGG-16 model with different batch sizes on NVIDIA GPU 1050

cuDNN via SYCL runtime using the CUDA driver API. An empty interop task for a softmax primitive in oneDNN is executed to measure the cost of launching and synchronizing a cuDNN function via SYCL interoperability task. The result shows about $0.75ms$ on average. The current implementation of the interop task forces the generation of a barrier at the end of each call to ensure program correctness which is unnecessary and costly in for the CUDA backend. By switching to the aforementioned new host task [30], the replacement for the interop task already present on the SYCL 2020 provisional specification, a further performance improvement is expected given the removal of this implicit barrier. To highlight the importance of an efficient and asynchronous implementation of host task, the asynchronous barrier dispatched automatically after each interop task submission is removed. This barrier is required to ensure the execution of any kernel after the primitive waits for all operations that have been enqueued. However, in our benchmark, there is always a dependency injection to the next operation or a barrier, hence, the interop task barrier is redundant. Running the empty softmax primitive without the extra synchronization for oneDNN takes an average time of $0.396142ms$, which results in 40% further performance improvement. This cumulatively adds up for the different primitives in the network and causes the constant overhead for ResNet-50 as seen in Table II. This result represents the maximum achievable performance when using an efficient implementation of the host task.

To summarise, the overhead of using SYCL to call native library functions is almost negligible.

D. Performance comparison with cuDNN on GTX 1050 Ti

Similar to the previous sub-section, we compare the performance of oneDNN's NVIDIA backend with cuDNN benchmarks which have been implemented from scratch using the CUDA Runtime API.

Figure6 and Figure7 shows the performance of oneDNN's NVIDIA backend with cuDNN. The $x - axis$ shows the different batch sizes used for the performance evaluation and $y-axis$ shows the number of images processed by the network

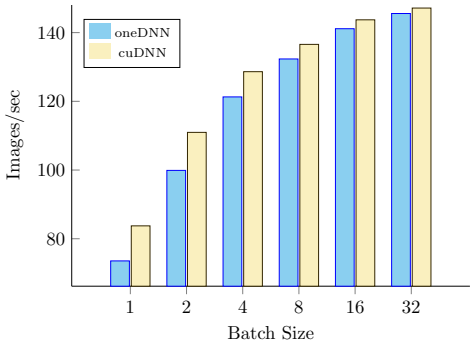

Fig. 7. Performance comparison between oneDNN and cuDNN for ResNet-50 model with different batch sizes on Nivida GPU 1050

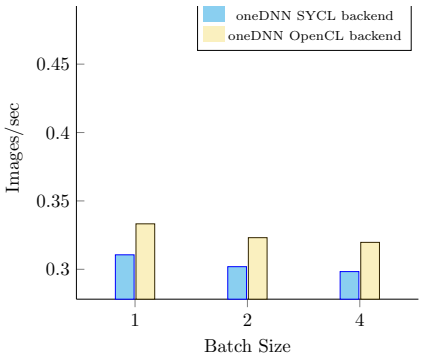

Fig. 8. Performance comparison between oneDNN and openCL for ResNet-50 model with different batch sizes on Intel GPU

in a second. Similar to the performance on Titan, for VGG-16(Fig6), the performance of oneDNN's NVIDIA backend is within 1% $(0.04\% - 1\%)$ of direct cuDNN for all the batch sizes.

Figure 7 compares oneDNN's NVIDIA backend with direct cuDNN for the ResNet-50 model. The performance drop seen is similar to that discussed in the previous sub-section.

This experimental analysis shows that using SYCL to achieve performance and portability works (scales) both for high-end GPU like Titan RTX and desktop range GPUs like GTX 1050.

E. Performance comparison on Intel GPU

To illustrate the flexibility of the oneDNN library (and, in general, of the oneAPI approach of re-using building blocks), Figure 8 shows the performance of using oneDNN SYCL backend compared to using oneDNN OpenCL running on Intel GPU. Since the hardware used was an integrated Intel GPU, only the results for small batch sizes could be collected. As stated in Figure 8, there is minimal overhead even when running on Intel GPUs (about 7%). Note that no changes are required to the code, only specifying which device is going to run the network by passing the device type as an input parameter to the executable is enough to make the code run on Intel GPUs.

VI. Conclusion and Future Works

This paper enabled the integration of vendor specified cuDNN library as a backend of oneDNN framework, using SYCL as a glue for execution coordination and orchestration of resource allocation. The performance portability of integrated cuDNN backend has been evaluated by comparing the execution time of VGG-16 and ResNet-50 models written in oneDNN versus those written in plain cuDNN functions. The evaluation demonstrated that using SYCL backend via oneDNN is a viable option for seamless cross-platform performance portability, despite the early stage of the work at the time of writing.

In the current implementation, the SYCL interop_task is used for interoperability with cuDNN. The performance analysis shows that replacing interop_task with the newly introduced host_task model in SYCL can up to 40% improve the execution overhead of run-time kernel scheduling due to using non-blocking multi-threaded launching of kernels. Integration of SYCL interoperability with host_task remains as a future work for oneDNN backend.

Moreover, there are still primitives in oneDNN for which the cuDNN backend has not been integrated. An example of such primitives is the recurrent neural network(RNN), where the implementation of cuDNN is different from that of in oneDNN. As future work, we are planning to implement a SYCL kernel that can be compiled and run on all SYCL supported backends to match the missing functionality.

Lastly, supporting the tensor core for compute-intensive operations and the decision on when to use it remains as a future work. NVIDIA does not enable it by default, given it can affect the model accuracy. This difference in numerical result is due to the different sequencing of the floating-point operations in the tensor core module versus the sequential executing of floating-point operations. How this can be exposed in the oneDNN interface is still an open question.

In the context of oneAPI CUDA implementation, there are plenty opportunities to introduce performance-portable support for CUDA, such as communication libraries or linear algebra interfaces. Future work will address those areas of support.

Integrating NVIDIA support is an initial step towards achieving cross performance portability using SYCL API, enabling the feasibility of integrating SYCL support for other vendor specified frameworks to the oneAPI open-source echosystem. In particular, integration of the AMD backend to oneAPI echosystem can significantly extend the cross performance portability for a large volume of high-performance computing applications. Supporting oneDNN on AMD devices requires the integration of HIP on the DPCPP compiler. Once added, hipDNN integration into oneDNN would be feasible future work.

Acknowledgments

This work would not have been possible without the help of a large number of engineers from Intel and Codeplay. The

DPC++ team at Intel and the DPC++ CUDA team at Codeplay provided support and feedback during the implementation of the DNN backend and the performance analysis.

This work, alongside a broader effort from Codeplay and Intel engineers to bring CUDA support to the open-source DPC++, provided significant amount of feedback and implementation experience to the SYCL 2020 revision of the standard. We also want to thank the members of the SYCL Working Group for their feedback on our extensions and the progress on the SYCL 2020 provisional.

REFERENCES

[1] "Aurora, argonne leadership computing facility," https://www.alcf.anl.gov/aurora, accessed: 2020-08-31.

[2] "Netlib blas: Basic linear algebra subprograms," http://www.netlib.org/blas/, accessed: 2020-08-31.

[3] "The oneapi specification," https://www.oneapi.com/, accessed: 2020-08-31.

[4] "Sycl: C++ single-source heterogeneous programming for opencl," https://www.khronos.org/sycl/, accessed: 2019-03-11.

[5] "Oneapi deep neural network library (onednn)," https://01.org/onednn, accessed: 2020-08-31.

[6] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 2014.

[7] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[9] R. D. Hornung and J. A. Keasler, "The raja portability layer: overview and status," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[10] "Intel® math kernel library," https://intel.ly/32eX1eu, accessed: 2020-08-31.

[11] "Intel® math kernel library for deep learning networks," https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-dnn-part-1-library-overview-and-installation.html, accessed: 2020-10-05.

[12] "The arm computer vision and machine learning library," https://github.com/ARM-software/ComputeLibrary/, accessed: 2020-08-31.

[13] "Nvidia cublas: Dense linear algebra on gpus," https://developer.nvidia.com/cublas, accessed: 2020-08-31.

[14] "Miopen: Amd's machine intelligence library," https://github.com/ROCmSoftwarePlatform/MIOpen, accessed: 2020-08-31.

[15] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2018.

[16] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," 2018.

[17] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," 2020.

[18] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," 2018.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 675–678. [Online]. Available: https://doi.org/10.1145/2647868.2654889

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8026–8037. [Online]. Available: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[22] "Tinynn," https://github.com/borgwang/tinynn, accessed: 2020-07-23.

[23] "Sycl: C++ single-source heterogeneous programming for opencl," https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf, accessed: 2020-07-23.

[24] "interop_task: Improving sycl opencl interoperability," https://github.com/codeplaysoftware/standards-proposals/blob/master/interop_task/interop_task.md, accessed: 2020-08-31.

[25] "Intel llvm," https://github.com/intel/llvm, accessed: 2020-07-23.

[26] "Intel llvm," https://bit.ly/3heBsPu, accessed: 2020-07-23.

[27] "Nvidia cudnn documentation," https://bit.ly/35nByC5, accessed: 2020-07-23.

[28] A. Z. Karen Simonyan, "Very deep convolutional networks for large-scale image recognition," 2014.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[30] "Sycl 2020 provisional," https://www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf, accessed: 2020-07-23.

APPENDIX

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [CROSS-PLATFORM PERFORMANCE PORTABILITY OF DNN MODELS USING SYCL]

A. Abstract

The incoming deployment of Exascale platforms with a myriad of different architectures and co-processors have prompted the need to provide a software ecosystem based on open standards that can simplify maintaining HPC applications on different hardware. Applications written for a particular platform should be portable to a different one, ensuring performance is as close to the peak as possible. However, it is not expected that key performance routines on relevant HPC applications will be performance portable as is, specially for common building blocks such as BLAS or DNN. The oneAPI initiative aims to tackle this problem by combining a programming model, SYCL, with a set of interfaces for common building blocks that can be optimised for different hardware vendors. In particular, oneAPI includes the oneDNN performance library, which contains building blocks for deep learning applications and frameworks. By using the SYCL programming model, it can integrate easily with existing SYCL and C++ applications, sharing data and executing collaboratively on devices with the rest of the application. In this paper, we introduce a cuDNN backend for oneDNN, which allows running oneAPI applications on NVIDIA hardware taking advantage of existing building blocks from the CUDA ecosystem. We implement relevant neural networks (ResNet-50 and VGG-16) on native CUDA and also a version of oneAPI with a CUDA backend, and demonstrate that performance portability can be achieved by leveraging existing building blocks for the target hardware.

B. Description

1) Check-list (artifact meta information):

- **Program: SYCL and C++**
- **Compilation: Intel DPCPP and cuDNN** 7.6.5
- **Run-time environment: Ubuntu 18.04.4 LTS**
- **Hardware: Nvidia GPU Titan RTX, GeForce GTX 1050 Ti, Intel HD Graphics 630 integrated GPU with Intel i7-9700k CPU**
- **Publicly available?: Yes**

2) How software can be obtained (if available): The `oneDNN`'s Nvidia backend implementation can be found at https://github.com/oneapi-src/oneDNN/tree/dev-v2-nvidia-support. The models' VGG-16 and ResNet-50, used in the paper, are publicly available at https://github.com/codeplaysoftware/oneDNN/tree/dev-v2-nvidia-support. The cuDNN benchmarks are currently not open source, but a binary of them can be provided upon request.

3) Hardware dependencies:

- Nvidia GPU Titan RTX with Intel i7-6700k CPU @ 4.00GHz (32K L1 cache, 256K L2 cache and 8192K L3cache)
- GeForce GTX 1050 Ti with Intel i7-9700k CPU @ 3.60GHz (32K L1 cache, 256K L2 cache and 12M L3cache)
- Intel HD Graphics 630 integrated GPU with Intel i7-9700k CPU @ 3.60GHz (32K L1 cache, 256K L2 cache and 12M L3cache)

4) Software dependencies:

- cuDNN library version used is 7.6.5
- DPCPP version available at https://bit.ly/3heBsPu

5) Datasets: Trained weights for VGG-16 and ResNet-50 are available as *resnet50_transposed_param_files* and *vgg16_transposed_param_files* at https://github.com/mehdi-goli/oneDNN/tree/codeplay-onednn-nvidia-support

C. Installation

Please follow the README file at https://github.com/codeplaysoftware/oneDNN/blob/dev-v2-nvidia-support/examples/models/README.md

D. Experiment workflow

We ran models on a machine with three different hardware setup as in B3. The experimental setup and flow is further described in detail in the paper. Please follow the instruction in section C to prepare the environment.

E. Evaluation and expected result

The expected results include the computation time and exclude the data transfer and compilation time. We ran benchmarks multiple times and used average execution times. Multiple individual runs confirmed the accuracy of the results.