Received: 17 January 2021 | Revised: 8 February 2022 | Accepted: 10 February 2022

DOI: 10.1002/cpe.6917

RESEARCH ARTICLE

WILEY

Applying Intel's oneAPI to a machine learning case study

Pablo Antonio Martínez[1] | **Biagio Peccerillo**[2] | **Sandro Bartolini**[2] | **José Manuel García**[1] | **Gregorio Bernabé**[1]

[1]Computer Engineering Department, University of Murcia, Murcia, Spain

[2]Department of Information Engineering and Mathematics, University of Siena, Siena, Italy

Correspondence
Pablo Antonio Martínez, Computer Engineering Department, University of Murcia, Murcia, Spain.
Email: pabloantonio.martinezs@um.es

Funding information
European Regional Development Fund; MCIN/AEI/10.13039/501100011033, Grant/Award Number: RTI2018-098156-B-C53

Abstract

Different technologies and approaches exist to work around the performance portability problem. Companies and academia work together to find a way to preserve performance across heterogeneous hardware using a unified language, one language to rule them all. Intel's oneAPI appears with this idea in mind. In this article, we try the new Intel solution to approach heterogeneous programming, choosing machine learning as our case study. More precisely, we choose Caffe, a machine learning framework that was created six years ago. Nevertheless, how would it be to make Caffe again from the beginning, using a fresh new technology like oneAPI? In terms of not only the ease of programming-because only one source code would be needed to deploy Caffe to CPUs, GPUs, FPGAs, and accelerators (platforms that oneAPI currently supports)-but also performance, where oneAPI may be capable of taking advantage of specific hardware automatically. Is Intel's oneAPI ready to take the leap?

KEYWORDS

heterogeneous computing, high performance computing, performance portability, machine learning

1 | INTRODUCTION

Computer architecture is a field that is constantly evolving. Many years of development has led to the adoption of heterogeneous architectures and specialized hardware to solve certain kinds of problems. This approach, however, has many different trade-offs.

First, the performance of an application can be enhanced by using accelerators: hardware explicitly designed to work in a given area. For example, Google's tensor processing unit (TPU)[1] is designed to speed up machine learning workloads. However, we can find less specialized accelerators, as is the case of graphics processing units (GPUs) or field programmable gate arrays (FPGAs). Another benefit of this specialization is energy efficiency. Even though accelerators offer significant performance improvements, they are simpler pieces of hardware that can achieve much better energy efficiency.[1,2]

On the other hand, each kind of accelerator needs a different environment, which implies the usage of different programming languages and/or models. Some examples are CUDA[3](for NVIDIA GPUs), hardware description languages (HDLs) (for FPGAs),[4] and domain-specific languages (DSLs)[5] for various kinds of accelerators. Since we need different approaches to deploy software on each, we introduce much complexity in software development. Today, to develop an application to work with CPUs and NVIDIA GPUs, we need to write two different versions of the same algorithm, one for each device. If we are willing to add FPGAs, we would need to add a new version. Another scenario is the case of the usage of different accelerators. If we have a wide variety of accelerators designed for machine learning, we need to use different DSLs to take advantage of them.

Although the hardware evolves quickly, the software struggles to keep up with this evolution. There are many different software solutions to support this wide variety of hardware. However, none of them has set a standard. There is no consensus because using these technologies still

presents some caveats. The most critical problems are the complexity needed to develop these solutions and the performance loss against native code. The existence of a unified language to program heterogeneous hardware would provide uncountable benefits: one single source code to program all devices and the possibility to distribute a workload among radically different devices to exploit their hardware capabilities. This new area is called performance portability.[6] It is said to open a new horizon in the computer architecture field, to the point where some researchers believe this will ultimately lead to "a new golden age for computer architecture".[7] In the last year, we have witnessed an increasing interest in this topic. Academia and companies are pursuing the same target. For example, Xilinx is working on triSYCL[8] and Intel recently launched oneAPI.[9,10]

This article presents a study of Intel's approach to heterogeneous computing, oneAPI, applied to a machine learning case study, the Caffe framework. We port two different layers in Caffe using oneAPI (softmax and convolution layer) to evaluate the strengths and weaknesses of the constantly evolving oneAPI library. In the case of the softmax layer, we implement it directly in oneAPI (compiling the code using the oneAPI compiler `dpcpp`). For the convolution layer, we implement it using the oneAPI library specialized for deep learning, oneDNN. Both layers are then evaluated on CPU and GPU. This article makes the following contributions:

- A oneAPI port of the Caffe's softmax layer for the feedforward phase, implemented using direct oneAPI programming.
- A oneAPI port of the Caffe's convolution layer for the feedforward phase, implemented using oneDNN.
- A performance study between the original Caffe layers and our oneAPI port, evaluating the viability of a single source approach (oneAPI Caffe port) against a native implementation (original Caffe).

The rest of the article is organized as follows. In Section 2 we discuss the background in this area and compare some of the most relevant approaches. In Section 3 we show our article. First, we show the up-to-date limitations and strengths of oneAPI. Then, we present our approach to design a softmax and a convolution layer using oneAPI. Our experimental methodology and the performance evaluation of the oneAPI implementation are shown in Section 4. Finally, Section 5 concludes the article.

2 | BACKGROUND AND RELATED WORK

2.1 | Approaches comparison

Nowadays, systems on chips (SoCs) are very popular. These kinds of integrated circuits have many heterogeneous components in a single chip, such as a CPU, a GPU and one or more accelerators, the most common being those for machine learning or video processing. Some examples of these kinds of SoCs are Kirin 970,[11] Qualcomm Snapdragon 855[12] and the new Apple M1.[13] Because each accelerator is unique, a different language or SDK is needed to use each of them. Kirin 970 uses HiAI[11] and Snapdragon 855 uses Neural Processing SDK.[14] Exploring different SoCs teaches us that the variety of platforms and languages is very large.

Many different languages and frameworks try to deliver high performance while providing a single source code for programming heterogeneous devices. One of the most well-known and stable solutions for the performance portability problem is the SYCL standard.[15] SYCL defines a standard that must be followed by specific implementations. ComputeCpp,[16] oneAPI,[9] and triSYCL[8] are some of the implementations of SYCL. In addition to the SYCL standard, each implementation may add more features. For example, oneAPI includes interesting features[17] like USM (unified shared memory) memory model or the concept of sub items. It makes sense to add USM in oneAPI because one of the things that oneAPI offers is the possibility of targeting Intel integrated GPUs, which use the same address space as the CPU itself. Sub items are an interesting contribution too because they allow automatic vectorization on Intel CPUs. Furthermore, if a SYCL application does not use any vendor-specific features, the code is compatible with all SYCL implementations. However, each of them implements the standard differently, so performance mismatches are expected.

In the case of oneAPI, the implementation of SYCL is based on LLVM.[18] It is a compilation strategy that uses a low-level virtual instruction set with rich type information as a common code representation for all phases of compilation. Therefore, oneAPI follows the path of other remarkable approaches that use LLVM compiler infrastructure, like HPVM,[19] which is based on LLVM and the idea of virtual ISA. It uses a hierarchical dataflow graph to represent computation tasks and the relations between them. MLIR[20] is a compiler infrastructure also based on LLVM, which is intended to be used in heterogeneous systems. One of the targets of MLIR is to be used in machine learning (Tensorflow[21] is currently using it). Another strategy is to build a solution targeted to a specific area. This is the case of TVM,[22] a compiler that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware backends. An interesting detail of TVM is the application of machine learning to its optimization. Using machine learning, they aim to find optimal operator implementations for each layer of a deep learning model. In their experiments, they expose a notable speedup with this technique against cuDNN.[22]

A very different idea is the concept of ALP (accelerator-level parallelism).[23] It represents an approach that generalizes any kind of accelerator thanks to an abstraction of instructions that can be translated to the ISA of a CPU but also to the instructions of any accelerator.

PHAST[24,25] is a high-level library for programming both multicore systems and NVIDIA GPUs. PHAST code can be written once and targeted to different devices via a single macro at compile-time. This macro will generate either CPU or NVIDIA GPU executables. The idea to provide such functionality is to develop different backends for each library call for all the hardware devices that the library supports. PHAST has been studied and compared to other low-level and high-level approaches, from both performance and productivity points of view. It has been demonstrated to be a more productive approach in terms of three different complexity metrics and also to provide high performance.[25]

2.2 | Machine learning as a case study

We decided to use machine learning as a case study since nowadays there is a lot of work around this topic. Not only in the improvements in machine learning techniques but also in the performance and the performance portability of machine learning applications. As we mentioned, TVM aims to provide a portable solution for machine learning, and there is a high count of works focused on improving the performance of machine learning inference[26] and training workloads. Moreover, it is also an attractive topic due to the growing interest in its applications and its significant computing power requirements.

Pytorch,[27] Tensorflow,[21] Keras,[28] and Theano[29] are some of the most popular machine learning frameworks. Caffe[30] used to be, but its popularity started to slow down years ago. However, it fits our needs since it is an open source[*], easy to extend and modify, well studied framework. It is designed in C++ and CUDA, supporting CPU and NVIDIA GPUs. Its design holds no secrets, as the majority of the code belongs to each of the layers that the framework supports. Even though it does not represent today's trend, it is a perfect fit for our study: it is written in C++, and oneAPI is intended to work with C++. Moreover, many libraries and frameworks offer machine learning acceleration. cuDNN provides such functionality for CUDA programming and Intel's MKL-DNN does the same for Intel CPUs. However, MKL-DNN disappeared with the birth of oneAPI, changing its name to oneDNN.

2.3 | Current state of oneAPI

This section is divided in two; the state of `dpcpp` (the oneAPI compiler) and the state of oneDNN (optimized oneAPI library for deep learning):

a oneAPI can be downloaded from Intel's webpage[†]. The oneAPI toolkit provides libraries and tools to deploy a single-source application to many heterogeneous architectures. The key component that allows this integration is the `dpcpp` compiler (Data Parallel C++ compiler). At the time of writing, oneAPI's latest official release (`2021.1-beta09`) provides a `dpcpp` compiler that supports Intel CPUs, some Intel GPUs (integrated graphics) and Intel FPGAs, but not NVIDIA GPUs. However, the source code of the `dpcpp` compiler does have support for NVIDIA GPUs[‡]. This enhancement is quite new,[31] so it is expected to be less mature than the rest of the oneAPI standard. Since this is an experimental feature, it is disabled in the official builds (Intel may include support in a future official release of oneAPI), but compiling `dpcpp` from source allows to build it with support for NVIDIA GPUs (AMD GPUs support is available too). To do so, the flag `--cuda` has to be used when running the `configure.py` script.

b In oneDNN we find a similar scenario. Even with a `dpcpp` with support for NVIDIA, the official build of oneDNN (version 1.7) does not support NVIDIA GPUs. This happens because it does not interoperate with `dpcpp`, so it is unable to use the `dpcpp` support for heterogeneous architectures. However, oneDNN v2.0 Beta can interoperate with `dpcpp`, but because it has no official release yet, to enable CUDA we must compile it from source. To enable CUDA support, several CMake flags such as `-DDNNL_GPU_VENDOR=NVIDIA`, have to be set to specify the path of CUDA and cuDNN installations.

To sum up, both `dpcpp` and oneDNN offer support for NVIDIA GPUs, but none of them has enabled such a functionality in the official builds. To enable CUDA support, compiling from source is needed. Furthermore, we show in this article that it is in a very early stage, and a lot of features are not supported yet.

3 | PORTING CAFFE TO ONEAPI

Caffe framework is composed of many different layers and some functionality to manage the framework itself, allowing for the creation of networks, management of layer's memory, and so forth. In this article, we port and evaluate the opportunities, pros, and cons of using the new Intel's oneAPI in a machine learning scenario. We focus on convolutional neural networks, which are the kind of networks supported in Caffe.

We have decided to port two different layers from Caffe. On the one hand, the softmax layer is a simple, very common layer, useful as a simple example to compare a native implementation (Caffe in CPU and Caffe in GPU) against a hardware-agnostic one (oneAPI). On the other hand, the

convolution layer exhibits weakness and opportunities inside the main component of a convolutional neural network, where performance is crucial. Without losing generality, we decide to concentrate on the feedforward phase. It is the heart of convolutional neural networks in inference mode, and its algorithms have features and access patterns similar to those used in backpropagation.[32] For these reasons results emerging from this study may well be valid also for most parts of backpropagation too.

To design the oneAPI version of a given layer, we first need to isolate a layer from the Caffe framework. We have to remove all the dependencies of a given layer from the framework. It allows us to run just this layer, filling it with arbitrary data and content of various sizes. Therefore, in the oneAPI code, we must respect the data layout of Caffe, because we are going to receive the data in the same way as the Caffe original layer. This means that both the input and output of the layer are 4D tensors in the NCHW layout. In the case of feedforward, Caffe represents the input as `bottom` and the output as `top`. This way, we can check the correctness of our implementation of the layer by comparing the output of the original Caffe code with our oneAPI implementation.

3.1 | Softmax layer (feedforward)

3.1.1 | Original Caffe version

The softmax layer essentially applies the softmax function. This function takes a vector of N dimensions as input and outputs a vector of the same number of dimensions. At each dimension, the output contains all the values in the input normalized in the [0, 1] range. The total sum must be 1 so that they can be interpreted as probabilities. Given a vector z with size i, and σ as the softmax function, it is computed as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}.$$

This layer is usually the last one in neural networks since it computes the probabilities of the classifier, that is, the probability of an image belonging to a class. Following the formula, the Caffe code computes softmax in four differentiated steps:

1. The max value of the first row of the cube is subtracted from all the elements. This procedure guarantees that no numerical issues arise.
2. The exponentiation of all the elements is done (e^{z_i}).
3. All the elements for a given classification are accumulated ($\sum_{j=1}^{K} e^{z_j}$). To do so, the cube is iterated by columns.
4. All the elements are divided by the accumulated value (again, this is done by columns).

Figure 1 shows the data layout that is used by the softmax layer in the original Caffe version.

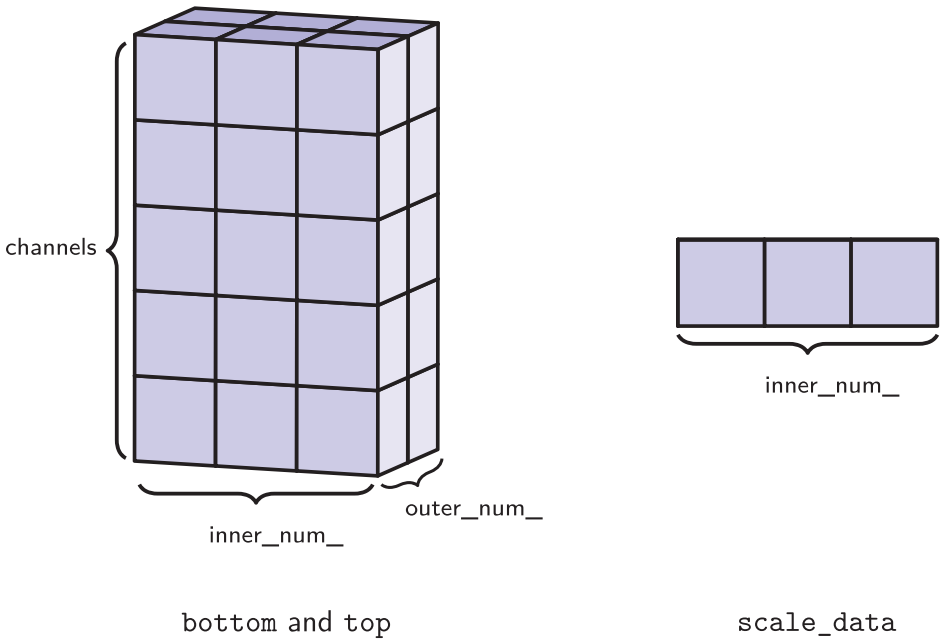

FIGURE 1 Data organization in Caffe's original softmax layer

In addition to `bottom` and `top` cubes, `scale_data` is an additional 1D vector that Caffe uses to store temporal data. Figure 1 shows `bottom` and `top` as a three-dimension structure, instead of four. This happens because the last dimension stores more data, so the full input contains many 3D cubes as shown in the figure. From an implementation standpoint, the main weakness of the Caffe implementation in CPU is the code being sequential, and as such, it is unable to benefit from multicores' parallel hardware.

3.1.2 | oneAPI version

Before the softmax is computed, oneAPI's environment has to be initialized (the same happens in the convolution layer). The softmax layer in oneAPI receives the same parameters as Caffe plus one additional parameter: the queue of the device. In the initialization, a device is selected, and a queue for such a device is created and passed to the layer. Note that this allows for generating a single executable that can be run on multiple hardware platforms. The device selected by oneAPI can be controlled:

- Before program execution: using environment variables;
- At runtime: relying on hardware detection and selecting one of the available devices.

The source code to compute the softmax layer in oneAPI is divided into four parts for clarity, which are:

1. Definition of the buffers that store the data to be computed by the softmax layer. The definition code is shown in Listing 1. SYCL buffers represent data that can exist on the host and/or any other device. This means that if the code is running on the host, no data copy is needed from the Caffe data (first argument of buffer constructor) to the buffer. If the device is not the host (e.g., a GPU with its memory), a data copy is done implicitly. Work items and work groups are defined in this section of the code. SYCL provides the possibility to express the kernel in the SIMT execution model.[33] Therefore, we have to control certain parameters in the execution of the kernel. This is the same idea of how a kernel works in CUDA. In the context of SYCL, (as happens in CUDA) these two values control the execution of SYCL kernels. Thus, in CUDA terminology work groups are equivalent to blocks and work items are equivalent to threads. In the case of softmax, we set the size of work items to the minimum of 32 or the total size of the input (see Listing 1). We adjust the work groups so that all work items will be working with a single data. Experienced programmers in CUDA will find this approach straightforward since oneAPI and CUDA share the SIMT execution model.

```
template <>
void SoftmaxLayer<float>::Forward_cpu(
    vector<Blob<float>*>& bottom,
    vector<Blob<float>*>& top,
    sycl::queue queue) {
  ...
  int work_items = min(32, t_size);
  int work_groups = (t_size + work_items - 1) / work_items;

  sycl::buffer<float,1> buf_t(top_ptr, sycl::range<1>(t_size));
  sycl::buffer<float,1> buf_b(bot_ptr, sycl::range<1>(b_size));
  sycl::buffer<float,1> buf_s(scale_ptr, sycl::range<1>(s_size));
```

Listing 1: Init code for softmax layer

2. Computation of the exponentials (step 2 mentioned in Section 3.1.1). In SYCL, the work is described as a command group. In Listing 2, a command group handler (variable `cgh`) is passed to the kernel. A command group encapsulates a kernel with its dependencies, and it is processed as a single entity atomically by the SYCL runtime once submitted to a queue. Inside the exponential kernel, we request access to write in the top buffer (`buf_t`, the output) and read from the bottom buffer (`buf_b`, the input). Finally, the kernel is submitted. Inside the kernel each thread will compute the exponential of one element of the input and store it in the same position on the output.

```
// 1. EXPONENTIAL
queue.submit([&] (sycl::handler& cgh) {
  auto t = buf_t.get_access<sycl::access::mode::discard_write>(cgh);
  auto b = buf_b.get_access<sycl::access::mode::read>(cgh);

  cgh.parallel_for<class SoftmaxExp>(sycl::nd_range<1>(work_groups * work_items, work_items),
 [=] (sycl::nd_item<1> item) {
    size_t local_id = item.get_local_linear_id();
    size_t global_id = item.get_global_linear_id();
```

```
    t[global_id] = sycl::exp(b[global_id]);
  });
});

queue.wait();
```

Listing 2: Exponential computation

3. Accumulation of the values previously computed with the exponential (step 3 mentioned in Section 3.1.1). This kernel, shown in Listing 3, is a little more elaborate because the access to the structure where data is stored is complex (due to the data layout adopted by Caffe). Fortunately, because SYCL kernels can be expressed as SIMT, we copied the code to compute this part directly from the source code of Caffe in GPU (source file `softmax_layer.cu`[§]).

   Memory is another concept in SYCL that shares similarities with CUDA. In SYCL, there is also a difference between local and global memory. In the kernel shown in Listing 1, all the accesses were done in global memory. There was no other choice since we need to modify the entire output structure. However, in the case of this kernel, we can do almost all the computation in local memory. While this will likely have no impact on the CPU, GPUs, and other accelerators will probably benefit from this optimization. To access global memory, SYCL provides a mechanism to allocate a chunk of local memory. In this kernel, however, we do not make use of such a mechanism. Instead, we use local memory implicitly. Local variables are stored in local memory, so `sum` accumulation is done in local memory although read access to `t` variable has to be done from global memory. After `sum` is computed, we store it in the `scale` global memory structure (see Figure 1). We have to reassign the work item and work group variables (see the first two lines in Listing 3) because the length of the data to work with is different from the previous kernel.

```
work_items = min(32, s_size);
work_groups = (s_size + work_items - 1) / work_items;
...

// 2. SCALE
queue.submit([&] (sycl::handler& cgh) {
  auto t = buf_t.get_access<sycl::access::mode::read>(cgh);
  auto sss = buf_s.get_access<sycl::access::mode::write>(cgh);

  cgh.parallel_for<class SoftmaxScale>(sycl::nd_range<1>(work_groups * work_items, work_items),
 [=] (sycl::nd_item<1> item) {
    size_t local_id = item.get_local_linear_id();
    size_t global_id = item.get_global_linear_id();
    size_t spatial_dim = height * width;
    size_t n = global_id / spatial_dim;
    size_t s = global_id

    float sum = 0.0f;
    for (int c = 0; c < channels; c++) {
      sum += t[(n * channels + c) * spatial_dim + s];
    }

    sss[global_id] = sum;
  });
});

queue.wait();
```

Listing 3: Accumulate

4. The division of all the exponentiated values (step 4 mentioned in Section 3.1.1). These values are stored in `top` data structure, represented by the `t` variable in Listing 4. This structure has to be divided by the sum of the exponentiated values (stored in `sss`). Again, the code can be easily adapted from the CUDA Caffe version and we also have to reassign the work item and work group variables.

```
// 3. DIVISION
work_items = min(32, t_size);
work_groups = (t_size + work_items - 1) / work_items;

queue.submit([&] (sycl::handler& cgh) {
  auto t = buf_t.get_access<sycl::access::mode::write>(cgh);
  auto sss = buf_s.get_access<sycl::access::mode::read>(cgh);
```

```
  cgh.parallel_for<class SoftmaxDivide>(sycl::nd_range<1>(work_groups * work_items, work_items),
 [=] (sycl::nd_item<1> item) {
    // variable position computations ...

    t[global_id] = t[global_id] / sss[n * spatial_dim + s];
  });
});

queue.wait();
}
```

Listing 4: Division

However, the implementation of this kernel has a little room for improvement, but SYCL and oneAPI do not provide an easy approach to accomplish that. The problem is the usage of local memory. In the `SoftmaxScale` kernel, the variable `sum` contains the value in local memory, which has to be divided. However, we had to store it in global memory to be able to read it in the `SoftmaxDivide` kernel. The best approach would have been to compute the sum in local memory and then compute the division using local memory too, but there is no way to communicate this data between different kernels (`sum` is unavailable from `SoftmaxDivide` kernel). A tricky solution could be to calculate the division in the same kernel (the `SoftmaxScale` kernel). This is not a good idea due to the different grains of parallelism of the two kernels: `SoftmaxScale` kernel needs a small number of work groups, while `SoftmaxDivide` kernel is capable of running a much bigger amount of work groups.

3.1.3 | oneAPI support

We were able to build `dpcpp` compiler from source successfully, even though we encountered unexpected issues[¶]. We enabled CUDA support in the compilation process in order to deploy the layer to NVIDIA devices too. In the end, our custom oneAPI softmax layer builds and runs successfully on both CPU and GPU.

3.2 | Convolution layer (feedforward)

3.2.1 | Original Caffe version

Caffe implements the convolution using general matrix multiplication (`GeMM`). This approach is adopted by many deep learning frameworks due to performance reasons. To compute the convolution using matrix multiplication, the first thing to do is to transform the input using a method called "image to columns" (`im2col`). Caffe relies on different optimized libraries to do the convolution: openBLAS, ATLAS or MKL in the CPU and cuBLAS in the GPU. Depending on the selected backend, the performance may vary. Some backends do not even provide parallelization in CPU (e.g., openBLAS), in which case convolution will run sequentially. If MKL is used instead, parallelization and other low-level optimizations (like vectorization) are applied. Furthermore, Intel made its version of Caffe, which is also based on the same `im2col`and `GeMM` idea, but it is implemented using OpenMP. Thus, we will compare the oneAPI convolution layer against two different approaches: the isolated convolution layer from BVLC Caffe[#] and the one from Intel Caffe[‖]. While the `im2col+GeMM` method is a typical way of implementing the convolution, it is not the only one. Winograd convolution[34] and direct convolution[35] are other efficient competitive approaches. Both of them are available in the oneDNN library, which we adopt to compute the convolution using oneAPI. Another important fact to take into account is that Caffe uses an NCHW layout for the input. During the original Caffe layer exploration, we found three weaknesses that can be exploited to improve the performance. First, Caffe uses its own implementation of `im2col`, which is potentially slower than an optimized one. Second, Caffe expects the selected backend to be parallelized, which is not always true. Third, Caffe implementation has a lot of code that handles manually the `im2col` execution and other aspects of the convolution. This handcrafted code is also potentially slower than an optimized one, like the one in the oneDNN library.

3.2.2 | oneAPI version

To implement convolution in oneAPI, we use oneDNN. This library allows the acceleration of machine learning workloads and is integrated inside the oneAPI ecosystem. As we mentioned in Section 2, oneDNN was renamed from MKL-DNN, and in the future, it will work with `dpcpp` to deploy the workloads to the same platforms as those supported by `dpcpp`.

The initialization of oneDNN is similar to the one explained in Section 3.1.1. The only difference is that, instead of a device queue, the layer receives the kind of engine to be used. oneDNN engines are an abstraction of a computational device (CPU, GPU, etc). With the engine kind, we create an engine that will be used to compute the convolution. The engine creation is shown in Listing 5.

```
template <>
void ConvolutionLayer<float>::Forward_cpu(
    vector<Blob<float>*>& bottom,
    vector<Blob<float>*>& top,
    engine::kind engine_kind) {
  dnnl::engine engine(engine_kind, 0);
  dnnl::stream engine_stream(engine);
```

Listing 5: Engine initialization in convolution layer

The goal of the initialization code is to translate from the Caffe idiom to oneDNN. After the engine creation, we populate variables Listing 6 to be used in oneDNN calls using Caffe data. With `bottom[0]->shape()` and `top[0]->shape()`, we retrieve the shape of the tensor that represents the input (`bottom`) and the output (`top`), which are stored in a 4D vector. With that information, we can initialize the convolution values (batch size, number of weights, size of the weights, etc).

```
std::vector<int> input_shape = bottom[0]->shape();
std::vector<int> output_shape = top[0]->shape();
...

const memory::dim N  = input_shape[0], // batch size
const memory::dim IC = input_shape[1], // input channels
...

const memory::dim OC = output_shape[1], // output channels
const memory::dim KH = weights_shape[2], // weights height
const memory::dim KW = weights_shape[3], // weights width
...
```

Listing 6: Convolution data size initialization

To interact with memory, oneDNN has two abstractions: the memory description and the memory object itself. In Listing 7, we show the creation of the memory descriptor of the input data, represented by `conv_src_md`.

```
memory::dims src_dims = {N, IC, IH, IW};
...
auto conv_src_md = memory::desc(src_dims, dt::f32, tag::any);
```

Listing 7: Input memory description creation

After the creation of the memory description, we create the memory object (see Listing 8), which describes not only the data layout (NCHW in this case) but also the engine to be used and the dimensions of the tensor. Because the memory object also contains the data to be processed, we must copy the contents of the input to the memory object. The input (given by the Caffe layer) is stored in `blob`structure. If the code is running in a CPU, a simple `memcpy` is performed. In the case of a GPU or other device, the same idea is applied using the appropriate functions. The data copy is wrapped inside the function `write_to_dnnl_memory`.

```
auto conv_src_mem = memory({src_dims, dt::f32, tag::nchw}, engine);
write_to_dnnl_memory((void *) bottom[0]->cpu_data(), conv_src_md);
```

Listing 8: Input memory creation

After creating the memory object and descriptors for the output, bias and weights, we can create the forward convolution description Listing 9, which is needed to run the convolution. In this description, we specify which convolution algorithm we want. In this case, we are using direct convolution (`algorithm::convolution_direct`), but the Winograd convolution is available too in oneDNN (see https://oneapi-src.github.io/oneDNN/group__dnnl__api__attributes.html#ga00377dd4982333e42e8ae1d09a309640). With the forward convolution description, we can create the oneDNN primitive, which we named `conv_prim`.

```
auto conv_desc = convolution_forward::desc(prop_kind::forward_training,
                                           algorithm::convolution_direct,
                                           conv_src_md,
                                           conv_weights_md,
                                           user_bias_md,
                                           conv_dst_md,
                                           strides_dims,
                                           padding_dims_l,
                                           padding_dims_r);
// Create primitive descriptor.
auto conv_pd  = convolution_forward::primitive_desc(conv_desc, engine);
...
// Create the primitive.
auto conv_prim = convolution_forward(conv_pd);
```

Listing 9: Fordward convolution description creation

To link the memory descriptor with its memory objects, we use the convolution arguments, whose creation is shown in Listing 10.

```
std::unordered_map<int, memory> conv_args;
conv_args.insert({DNNL_ARG_SRC, conv_src_mem});
conv_args.insert({DNNL_ARG_WEIGHTS, conv_weights_mem});
conv_args.insert({DNNL_ARG_BIAS, user_bias_mem});
conv_args.insert({DNNL_ARG_DST, conv_dst_mem});
```

Listing 10: Convolution arguments creation

To actually run the convolution, we need the oneDNN primitive, the oneDNN engine and the convolution arguments (Listing 11).

```
conv_prim.execute(engine_stream, conv_args);
```

Listing 11: Convolution execution

After the convolution is computed, we have to read the data from the appropriate memory object and copy it back to the Caffe structure. We encapsulate the data copy inside the function `read_from_dnnl_memory` (Listing 12).

```
read_from_dnnl_memory(top[0]->mutable_cpu_data(), user_dst_mem);
```

Listing 12: Data copy from oneDNN memory to Caffe structures

3.2.3 | oneAPI support

We were able to build oneDNN from the source (again, with some issues[**]), enabling CUDA support. Nonetheless, we were not able to run the convolution layer on the GPU. We built an NVIDIA compatible oneDNN library, but at the moment, our implementation is incompatible with oneDNN in NVIDIA GPUs. The underlying implementation of the convolution uses a USM memory (unified shared memory) approach, while the CUDA backend currently only supports a buffer-based model[††]. Therefore, we were able to build the code for CPU and GPU, but only the CPU version ran successfully.

4 | EVALUATION

In this section, we show an evaluation study of the ported layers to oneAPI in this research: softmax and convolution. We compare our single-source implementation against the original Caffe framework, which is implemented using two different codebases, one for CPU and one for GPU.

4.1 | Test bed

The evaluation platform was equipped with the following hardware and software:

- CPU: 2 x Intel Xeon Gold 6238 CPU (22 cores each, 44 cores in total).
- GPU: NVIDIA RTX 2080Ti (4352 CUDA cores, 11 GB GDDR6).
- OS: CentOS Linux 8.2 (4.18.0-193.14.2.el8_2.x86_64 kernel).
- NVIDIA driver: 450.51.06.
- CUDA: 11.0.
- `dpcpp`: Built from source: Commit 633691327b3f2d6a2a7a8321244682e91db78e7d[‡‡].
- oneDNN: v2.0-beta10 (dnnl_lnx_1.96.0_cpu_iomp.tgz)[§§].
- TBB: `tbb-2021.1-beta08`.
- BVLC Caffe isolated layers: Commit 99bd99795dcdf0b1d3086a8d67ab1782a8a08383[¶¶].
- Intel Caffe isolate layer: Commit 3f494b442ee3f9d17a07b09ecbd5fa2bbda00836[##].
- MKL: 2021.1 Beta Update 9.
- openBLAS: 0.3.3.

For the values in the tables in Sections 4.2 and 4.3, each experiment is repeated five times (for each version). The values shown are the average over these five independent runs. Table 1 lists the compilers used for each layer.

4.2 | Softmax

To evaluate a layer, we design a set of inputs to gather information about the performance of each implementation. In each layer, we create a tensor with the proper dimensions and then run the layer with the input, measuring the execution time. The inputs used in the softmax layer are detailed in Table 2.

The dimensions of the first input are the same as the dimensions of the softmax input in a LeNet layer running MNIST. Because a common workload is to run the net for 1000 iterations, we also used 1000 iterations for input 1. We intend to create a realistic input. However, because MNIST is a small data set, we design input 5, which is similar to bigger workloads. We are interested in how the layer reacts to a different number of iterations, so we create input 4 from input 5. Finally, inputs 2 and 3 illustrate the behavior of the layer in a large instance.

TABLE 1 Compilers used for each version of the layers

(a) Softmax			(b) Convolution		
Version	**CPU**	**GPU**	**Version**	**CPU**	**GPU**
Caffe (BVLC)	g++ 8.3.1	CUDA nvcc 11.0.221	Caffe (Intel)	icpc 19.1.2	-
oneAPI	clang 12.0.0	clang 12.0.0	Caffe (BVLC)	g++ 8.3.1	-
			oneAPI	clang 12.0.0	-

TABLE 2 Different inputs for isolated softmax layer

Input	Iterations	outer_num	inner_num	Channels
Input 1	1000	2	6	10
Input 2	1	2000	600	100
Input 3	1000	2000	600	100
Input 4	1	200	600	100
Input 5	1000	200	600	100

Table 3 depicts the execution times for each version of the softmax layer. Because original Caffe allows us to select the math library backend (which remarkably impacts the execution time), we decide to build a version using openBLAS (which we observed to run sequentially) and one version using MKL (which we observed to run in parallel). For the MNIST-like input, Caffe with openBLAS is the winner, and oneAPI is the loser by far. The input is tiny and the number of iterations is large, so the sequential version of softmax wins against the parallel, which is incapable of benefitting from parallelism. A good point to explain the oneAPI loss is because of the overhead of the library, which is quite large compared with the actual work that needed to be done. However, oneAPI completely outperforms Caffe in bigger instances. In input 3, oneAPI achieves 5.8× and 9.0× speedup against Caffe, while the gain in input 5 remains the same. At a first glance, it does not make any sense. Figure 2 sheds a bit of light on this issue.

In this plot, we evaluate the execution time from one input to another. From input 2 to 3 we can see that the workload should be 1000× larger since the input is the same, but the layer runs a thousand times. Caffe with sequential math library evolution is close to 1000×, reaching around 900×. However, the parallel version using MKL is around 600×, which means that one single iteration is more expensive than one of the 1000 iterations. Finally, oneAPI drops to 200×. In the transition from input 4 to 5, the difference is even more dramatic, with a growth of 50× in the execution time. This explains the gain of oneAPI over the other versions. We found that this situation is caused by the overhead of oneAPI. For example, in input 5, oneAPI takes almost 7.4 s. We found that the first iteration of oneAPI takes 0.150 s, while the remaining 999 iterations take ∼0.007 s. Therefore, $0.150 + 0.007 * 999 = 7.143 \approx 7.395$. In other words, oneAPI initialization is heavy, which causes poor performance when the network is run for one iteration. However, the execution time of an iteration itself is great, so the performance in a real world scenario, running for 1000 iterations, is much better than the one offered by the original Caffe.

Table 4 shows the results for the softmax layer in GPU. In this case, oneAPI loses against Caffe with all the considered inputs. It happens even though the softmax layer was written in a similar way as CUDA kernels. The oneAPI version is slower for small inputs (like inputs 1 and 2) but also

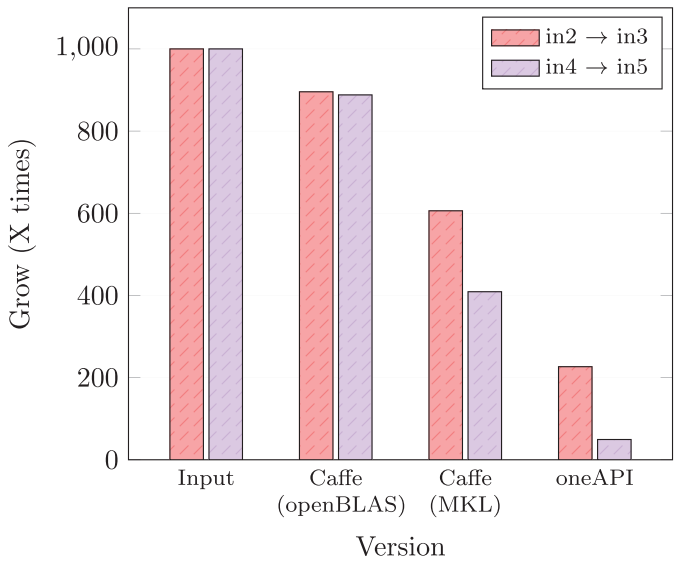

FIGURE 2 Execution time evolution from 1 to 1000 iterations in each version

TABLE 3 Execution times in seconds for isolated softmax layer (CPU)

Version	Input 1	Input 2	Input 3	Input 4	Input 5
Caffe (openBLAS)	0.001	0.709	635.000	0.071	63.051
Caffe (MKL)	0.004	0.670	406.000	0.100	40.900
oneAPI	0.474	0.309	70.000	0.150	7.395

TABLE 4 Execution times in seconds for isolated softmax layer (GPU)

Version	Input 1	Input 2	Input 3	Input 4	Input 5
Caffe (BVLC)	0.015	0.041	7.842	0.004	0.821
oneAPI	0.336	0.397	300.000	0.040	31.83

for big ones (like input 3), where the difference is even more dramatic. These results highlight that oneAPI support for NVIDIA GPUs is still in a very experimental stage and is thus not mature enough to be used in a real-world application.

4.3 | Convolution

Table 5 shows the different inputs used for evaluating the convolution layer. As we did in the softmax layer, we chose the input sizes based on real world datasets:

- Input 1 represents the input size of the MNIST dataset (gray-scale 28 × 28 images) with 5 × 5 filters.
- Input 3 represents the input size of the CIFAR-10 dataset (32 × 32 RGB images) with 5 × 5 filters.
- Input 5 represents the input size of the Imagenet dataset (227 × 227 RGB images) with 11 × 11 filters.

The rest of the inputs (2, 4, and 6) represent the same dataset but with 1000 iterations, instead of one. This differentiation helps to understand the behavior of the layer when it is run a single time and when it is run many times, emulating that the layer is inside a real network. To find the input sizes, we checked Vivienne Sze's article,[32] and we ran Caffe using different datasets.

In the evaluation of the convolution layer we also included the Intel Caffe version, so we can make a fairer comparison against oneAPI. To create the isolated layer from the Intel version, we took the `conv_layer.cpp`[III]. There are two other additional implementations of the layer, but we decided to compare this one for consistency. One of the other implementations used MKL-DNN (oneDNN now), so we should expect a similar performance. A detail to take into account when comparing the Intel convolution layer against the original BVLC one is the compiler used. Caffe BVLC uses g++ compiler by default, while Intel implementation uses icpc (Intel compiler). To be fair in the isolation layer phase, we keep the compiler in both cases, so we build each of three versions of layers (Caffe BVLC, Caffe Intel, and oneAPI) with a different compiler. See Table 1 for more details.

Execution times of the different convolution layers are presented in Table 6. In the MNIST dataset (inputs 1 and 2), we can see that Caffe BVLC is far from the other two versions. oneAPI's version is close but a bit far from Intel's version when running for 1000 iterations. This situation is reversed in the case of the CIFAR-10 dataset (inputs 3 and 4) because oneAPI achieves slightly better results than the Intel Caffe version. Moreover, BVLC Caffe competes in a lower league because the difference, in this case, is even greater. In the Imagenet dataset, the biggest one (inputs 5 and 6), oneAPI is much faster than the Intel version of Caffe, and both of them are much faster than the BVLC Caffe. Compared to the Intel optimized version of Caffe, the oneAPI implementation achieves 1.69× and 2.73× speedup in inputs 4 and 6, respectively. We can see that oneAPI is faster than the Intel version when the input size is big enough, and we believe this may be due to two different facts. First, the overhead in oneAPI may be harmful when computing convolution. Second, the algorithm used in both versions is not the same (matrix multiplication in the case of Intel Caffe and direct convolution in the case of oneAPI).

TABLE 5 Different inputs for isolated convolution layer

Input	Iterations	Image size	Number of filters	Filters size	Batches
Input 1	1	28 × 28 × 1	20	5 × 5	100
Input 2	1000	28 × 28 × 1	20	5 × 5	100
Input 3	1	32 × 32 × 3	32	5 × 5	100
Input 4	1000	32 × 32 × 3	32	5 × 5	100
Input 5	1	227 × 227 × 3	96	11 × 11	100
Input 6	1000	227 × 227 × 3	96	11 × 11	100

TABLE 6 Execution times in seconds for isolated convolution layer (CPU)

Version	Input 1	Input 2	Input 3	Input 4	Input 5	Input 6
Caffe (BVLC) MKL	0.058	3.872	0.069	11.971	2.536	1970.1
Caffe (Intel)	0.085	0.734	0.087	1.462	0.924	300.5
oneAPI	0.063	1.122	0.053	0.862	0.267	109.8

As mentioned in Section 3, our convolution implementation is not supported in GPU with the current oneDNN version, so we were unable to carry out a benchmark on the GPU. Furthermore, we used the direct convolution algorithm in oneDNN. We tried Winograd convolution too, but the program crashed with an exception (`dnnl::error`: could not create a primitive descriptor iterator).

Finally, we believe that the solid performance oneAPI demonstrated in both benchmarks is strongly influenced by the hardware platform used. Our dual-socket Xeon Gold has 44 cores and 88 threads, which is a considerable level of parallelism. oneAPI seems to be well optimized for Intel parallel architectures and benefit very well from a large number of cores.

5 | CONCLUSIONS

Heterogeneous computing is the answer the community adopted to face the slowdown in performance improvements in CPUs. At first glance, heterogeneity seems like a win–win, promising improved performance and lower energy consumption concerning classic CPU-based solutions. The problem arises when the programmer needs to face this heterogeneity. Heterogeneous programming is much harder than traditional programming, so unified heterogeneous languages like oneAPI are so important.

In this article, we have evaluated oneAPI, the Intel proposal for heterogeneous computing. We decided to apply oneAPI in machine learning because it is one of the most relevant workloads nowadays. In this process, we found things that worked well in oneAPI and other aspects that still need significant improvements, which however appear feasible in the near future.

Regarding oneAPI's strong points, we want to emphasize two: the performance in CPU and the programming interface. In evaluating oneAPI, we found a very mature CPU backend that can challenge and even outperform native alternatives. We believe this is even more important given that we did the evaluation running on a server CPU (Intel Xeon Gold), which highlights the excellent performance that oneAPI can achieve in CPUs. In the programmability aspect, oneAPI may be difficult because the programmer needs to learn new concepts related to the SYCL environment. However, programmers familiar with CUDA or OpenCL interfaces may learn them quickly because of their similarities with SYCL. Furthermore, using the oneDNN library was straightforward.

On the other hand, two of the weakest points in oneAPI we have found are the build process and the support for NVIDIA GPUs. At the moment, oneAPI official builds support Intel CPUs, some Intel integrated GPUs, and Intel FPGAs, but not NVIDIA GPUs. Even so, it is possible to build the `dpcpp` compiler enabling CUDA support to use NVIDIA GPUs. We struggled a bit in the process due to some issues encountered[***]. As we have shown, this support is very experimental at the moment. The performance delivered by the GPU backend in oneAPI makes it currently unusable in practice. Even worse is the case of oneDNN because we were unable to run it in GPU since official builds do not work together with `dpcpp`, so it does not interoperate with GPUs. Future versions of the library (oneDNN v2.0 Beta) will achieve this goal. We tried building the library (the v2.0 Beta version) to get the latest functionality (suffering other problems in the process[†††]). Unfortunately, once we compiled oneDNN v2.0, we found that its support for NVIDIA GPUs is quite limited; it only works with a buffered model but not with USM[‡‡‡], which is the model we use in our convolution implementation.

Overall, it is positive to see the emergence of libraries, like oneAPI, for effective programming of heterogeneous and parallel architectures due to their strategic role in the current more and more intense evolution of computing toward accelerated architectures. Therefore it is equally important to evaluate the strengths and weaknesses of these kinds of libraries, especially in critical application domains, to promote their improvement.

ACKNOWLEDGMENT

This work was funded by Grant RTI2018-098156-B-C53 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe."

DATA AVAILABILITY STATEMENT

Data available in article supplementary material.

ENDNOTES

[*] Available on https://github.com/BVLC/caffe.
[†] Available on https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit.html.
[‡] Available on https://github.com/intel/llvm.
[§] Available on https://github.com/BVLC/caffe/blob/master/src/caffe/layers/softmax_layer.cu.
[¶] See our issue in the oneAPI GitHub repository for more details: https://github.com/intel/llvm/issues/2696.
[#] Available on: https://github.com/BVLC/caffe.
[‖] Available on https://github.com/intel/caffe.
[**] See our issue in the oneDNN GitHub repository for more details: https://github.com/oneapi-src/oneDNN/issues/885.
[††] See our issue in the oneDNN GitHub repository for more details: https://github.com/oneapi-src/oneDNN/issues/888.
[‡‡] Available on https://github.com/intel/llvm.
[§§] Available on https://github.com/oneapi-src/oneDNN/releases.
[¶¶] Available on https://github.com/BVLC/caffe.

[##]Available on https://github.com/intel/caffe.
[|||]Available on https://github.com/intel/caffe/blob/master/src/caffe/layers/conv_layer.cpp.
[***]See our issue in the oneAPI GitHub repository for more details: https://github.com/intel/llvm/issues/2696.
[†††]See our issue in the oneDNN GitHub repository for more details: https://github.com/oneapi-src/oneDNN/issues/885.
[‡‡‡]See our issue in the oneDNN GitHub repository for more details: https://github.com/oneapi-src/oneDNN/issues/888.

ORCID

Pablo Antonio Martínez https://orcid.org/0000-0002-4391-2451
Biagio Peccerillo https://orcid.org/0000-0002-4998-0092
Sandro Bartolini https://orcid.org/0000-0002-7975-3632
José Manuel García https://orcid.org/0000-0002-6388-2835
Gregorio Bernabé https://orcid.org/0000-0002-7265-3508

REFERENCES

1. Jouppi NP, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit. *ISCA '17*. 2017:1-12.
2. Dally WJ, Turakhia Y, Han S. Domain-specific hardware accelerators. *Commun ACM*. 2020;63(7):48-57. doi:10.1145/3361682
3. NVIDIA. CUDA C programming guide. [Online]; 2021.
4. Nane R, Sima VM, Pilato C, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans Comput Aided Des Integr Circuits Syst*. 2016;35(10):1591-1604. doi:10.1109/TCAD.2015.2513673
5. Kosar T, Bohra S, Mernik M. Domain-specific languages: a systematic mapping study. *Inf Softw Technol*. 2016;71:77-91. doi:10.1016/j.infsof.2015.11.001
6. Neely J. DOE centers of excellence performance portability meeting. Technical report LLNL-TR-700962, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States); 2016.
7. Hennessy JL, Patterson DA. A new golden age for computer architecture. *Commun ACM*. 2019;62(2):48-60. doi:10.1145/3282307
8. Xilinx. triSYCL. [Online]; 2021. Accessed November 17, 2020. https://github.com/triSYCL/triSYCL
9. Intel. oneAPI. [Online]; 2021. Accessed November 17, 2020. https://github.com/intel/llvm
10. Intel. oneAPI specification. [Online]; 2020. Accessed November 17, 2020. https://spec.oneapi.com/versions/latest/oneAPI-spec.pdf
11. AnandTech. HiSilicon Kirin 970 - Android SoC power and performance overview. [Online]; 2018. Accessed November 17, 2020. https://www.anandtech.com/show/12195/hisilicon-kirin-970-power-performance-overview/6
12. Qualcomm. Snapdragon 855+ mobile platform. [Online]; 2020. Accessed November 17, 2020. https://www.qualcomm.com/products/snapdragon-855-plus-mobile-platform
13. AnandTech. Apple announces the apple silicon M1: ditching x86 - what to expect, based on A14. [Online]; 2020. Accessed November 17, 2020. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive
14. Qualcomm. Qualcomm neural processing SDK for AI. [Online]; 2020. https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk
15. Khronos OpenCL Working Group. SYCL provisional specification, version 1.2.1. [Online]; 2019. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
16. Software C. ComputeCpp. [Online]; 2020. Accessed November 17, 2020. https://github.com/codeplaysoftware/computecpp-sdk
17. Intel. oneAPI extensions. [Online]; 2021. Accessed January 12, 2021. https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions
18. Chris L, Vikram A. The LLVM instruction set and compilation strategy. Technical report UIUCDCS-R-2002-2292, CS Department, University of Illinois at Urbana-Champaign; : 2002.
19. Kotsifakou M, Srivastava P, Sinclair MD, Komuravelli R, Adve V, Adve S. HPVM: heterogeneous parallel virtual machine. *SIGPLAN Not*. 2018;53(1):68-80. doi:10.1145/3200691.3178493
20. Lattner C, Amini M, Bondhugula U, et al. MLIR: a compiler infrastructure for the end of Moore's law. Proceedings of the Compilers for Machine Learning Workshop at CGO; 2020.
21. Abadi M, Barham P, Chen J, et al. TensorFlow: system for large-scale machine learning. Google Brain; 2016:265-283.
22. Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning; 2018:578-594; USENIX Association, Carlsbad, CA.
23. Hill MD, Reddi VJ. Accelerator-level Parallelism; 2020. arXiv preprint arXiv:1907.02064.
24. Peccerillo B, Bartolini S. Task-DAG support in single-source PHAST library: enabling flexible assignment of tasks to CPUS and GPUS in heterogeneous architectures. *PMAM'19*. 2019:91-100.
25. Peccerillo B, Bartolini S. PHAST - a portable high-level modern C++ programming library for GPUs and multi-cores. *IEEE Trans Parallel Distrib Syst*. 2019;30(1):174-189. doi:10.1109/TPDS.2018.2855182
26. Wang YE, Wu CJ, Wang X, Hazelwood K, Brooks D. Exploiting parallelism opportunities with deep learning frameworks. *ACM Trans Archit Code Optim*. 2021;18(1). doi:10.1145/3431388
27. Paszke A, Gross S, Massa F, et al. PyTorch: an imperative style, high-performance deep learning library arXiv; 2019.
28. Chollet F. Keras; 2015. https://github.com/fchollet/keras
29. Team TTD,Al-Rfou R, Alain G, et al. Theano: a python framework for fast computation of mathematical expressions. arXiv; 2016.
30. Jia Y, Shelhamer E, Donahue J, et al. Caffe: convolutional architecture for fast feature embedding; 2014. arXiv preprint arXiv:1408.5093.
31. Goli M, Narasimhan K, Reyes R, et al. Towards cross-platform performance portability of DNN models using SYCL. Proceedings of the 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC); 2020:25-35
32. Sze V, Chen YH, Yang TJ, Emer JS. Efficient processing of deep neural networks: a tutorial and survey. *Proc IEEE*. 2017;105(12):2295-2329. doi:10.1109/JPROC.2017.2761740
33. Codeplay. SYCL for CUDA developers - execution model. [Online]; 2020. Accessed November 18, 2020. https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/execution-model

34. Lavin A, Gray S. Fast algorithms for convolutional neural networks. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2016:4013-4021; IEEE
35. Zhang J, Franchetti F, Low TM. High performance zero-memory overhead direct convolutions. Proceedings of Machine Learning Research; 2018:5776-5785; PMLR, Stockholmsmässan, Stockholm Sweden.

How to cite this article: Martínez PA, Peccerillo B, Bartolini S, García JM, Bernabé G. Applying Intel's oneAPI to a machine learning case study. *Concurrency Computat Pract Exper*. 2022;34(13):e6917. doi: 10.1002/cpe.6917