INFERENCE OF ML MODELS ON INTEL GPUS WITH SYCL AND INTEL ONEAPI USING SOFIE

August 2023

AUTHOR(S):
Ioanna-Maria Panagou

University of Thessaly

SUPERVISOR(S):
Lorenzo Moneta
Sanjibang Sengupta

CERN openlab

PROJECT SPECIFICATION

TMVA provides a fast inference system that takes an ONNX model as input and produces compilation-ready standalone C++ scripts as output which can be compiled and executed on CPU architectures. The idea of this project is to extend this capability to generate from the TMVA SOFIE model representation code that can be run also on Intel GPUs using both SYCL and Intel OneAPI libraries. These will allow for a more efficient evaluation of these models on Intel accelerator hardware.

ABSTRACT

ROOT[2] is an open-source framework, born in CERN, used for high-scale data processing and analysis in High Energy Physics and beyond. ROOT provides a powerful and versatile toolkit that enables researchers to manipulate, visualize, and extract valuable insights from complex data generated by experiments and simulations.

Recently, machine learning has established itself as a valuable tool for researchers to analyze their data and draw conclusions in various scientific fields as well as HEP. ROOT offers native support for supervised learning techniques, such as multivariate classification and regression through the TMVA[13] ROOT library. Among others, the package includes neural networks, deep networks and multilayer perceptrons. TMVA also allows interoperability with commonly used machine learning libraries, such as Keras and Pytorch. Even though the above libraries provide functionality for inference, they only support their own models and are constrained by heavy dependencies. ONNXRuntime by Microsoft, which is based on the ONNX standard for describing deep learning models, can combat the issue of interopability, but its large dependencies constitute its use in HEP infeasible.

SOFIE[1], which stands for System for Optimized Fast Inference code Emit, is an extension of the TMVA module and was proposed as the inference engine that could tackle the issues described above. SOFIE can take as input a trained ML model in a Pytorch, Keras or ONNX format and create standalone C++ inference code, which is directly invokable from other C++ projects and has minimal dependencies (only on BLAS libraries). In addition, it allows full control over the inference code and can be compiled on the fly using Cling JIT.

The purpose of this project was to extend the SOFIE functionality, so that it would be able produce inference code in SYCL[11] that could run on Intel GPUs using Intel oneAPI libraries. Work has also been done in benchmarking the performance of multiple models on GPUs, as well as enhancing the test suit that verifies the correctness of the produced SOFIE code.

TABLE OF CONTENTS

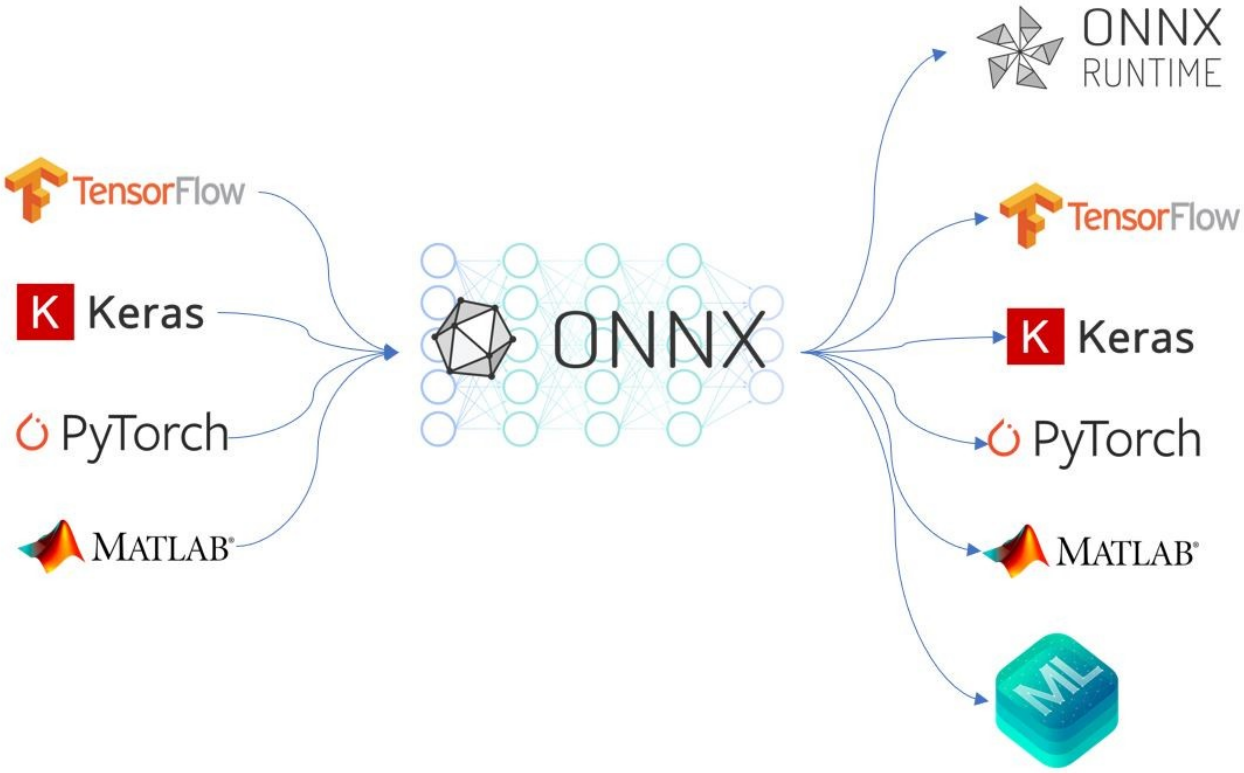

Figure 1: ONNX format allows for framework interoperability by providing a uniform format that acts as an intermediate between machine learning frameworks. This interoperability allows trained models to be easily deployed in different software/hardware platforms.

1 INTRODUCTION

A. OPEN NEURAL NETWORK EXCHANGE (ONNX)

ONNX[3], or Open Neural Network Exchange, is an open-source standard for representing deep learning models, developed by Facebook and Microsoft in order to make it easier for researchers and engineers to move models between different deep-learning frameworks and hardware platforms. Its main advantage is that it allows models to be easily exported from one framework, such as PyTorch[10], and imported into another framework, such as TensorFlow[8] (see fig. 1).

ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format (`.onnx`) to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers. An ONNX graph, as shown in fig. 2, is a directed graph, where the edges that connect the different operators represent the flow of data.

B. INTRODUCTION TO SOFIE

Before going into detail about our project, it is essential that the reader understands how SOFIE works. As seen in fig. 3, SOFIE takes as input a trained Machine Learning model in one of those 3 popular machine learning library formats: ONNX (`.onnx`), PyTorch, (`.pt`) or Keras (`.h5`). Then, the appropriate parser should be called by the user to parse the input model into an object of the `SOFIE::RModel` class. Internally, all input models are converted into their equivalent ONNX representation before they are transformed to an `RModel`. The `RModel` class is capable of storing the internal structure of the input model along with its learnable parameters.

The `RModel` class represents the input model as vector of `ROperators`. Those operators come in

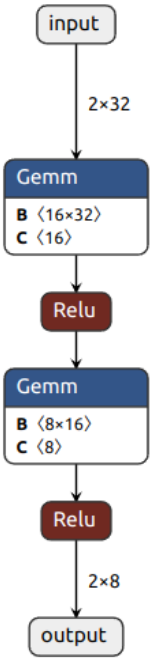

Figure 2: ONNX model visualization using Netron[9]

many flavors and have a 1-1 correspondence with the respective ONNX operators. Currently, about 30 out of the total ONNX Operators are supported by SOFIE.

After the `RModel` has been constructed, the code generation step takes place. The `Generate` function that is called on the model, internally calls the `Generate` functions of the operators that make up the model, as shown in fig. 4.

This step produces 2 outputs: a weight file in `.dat` file format (or in `.root` format, a functionality that has been added recently) that holds the model weights and parameters and a C++ header file (`.hxx`) that hardcodes the inference function. This header file can then be included in a plug-and-play fashion into any C++ project and has minimal dependencies. An example of this process can be seen in fig. 5.

C. SYCL

i. What is SYCL

SYCL is a single-source, high-level, C++ programming model that can target a wide range of heterogeneous platforms (CPUs, GPUs and FPGAs).

- **Single-source:** Unlike its predecessor, OpenCL, SYCL allows the code for the kernels that is going to be offloaded to a device to reside in the same source file as the host code. To produce the final executable, we need two compiler passes, one for the host and one for the device compiler. Both of them see the same SYCL API but interpret it differently. The device compiler (or SYCL compiler) identifies the kernel functions and creates a device IR for the requested ISA. The host compiler compiles the host code into a CPU object file, which is later linked with the device IR to form a single executable with both the CPU and GPU code (multi-compiler compilation model). The host and SYCL compiler can also be invoked by the same compiler driver (single-compiler compilation model). Both of these approaches are shown in fig. 6
- **High-level:** SYCL provides a number of high-level abstractions over boilerplate code, including platform/device selection, dependency management and much more.

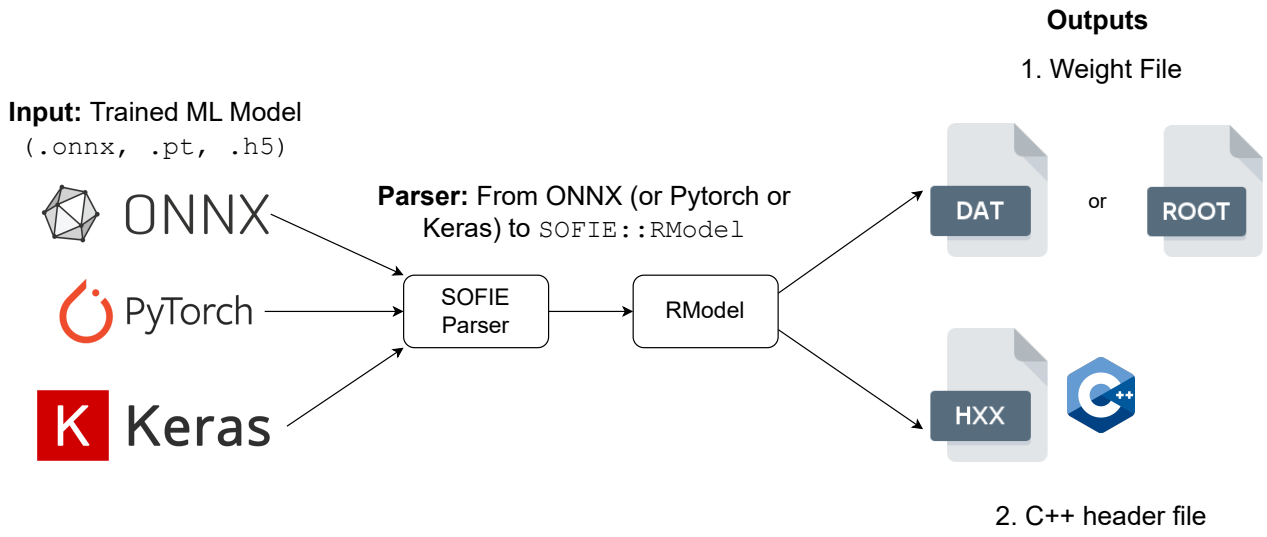

Figure 3: SOFIE Code Generation Flow

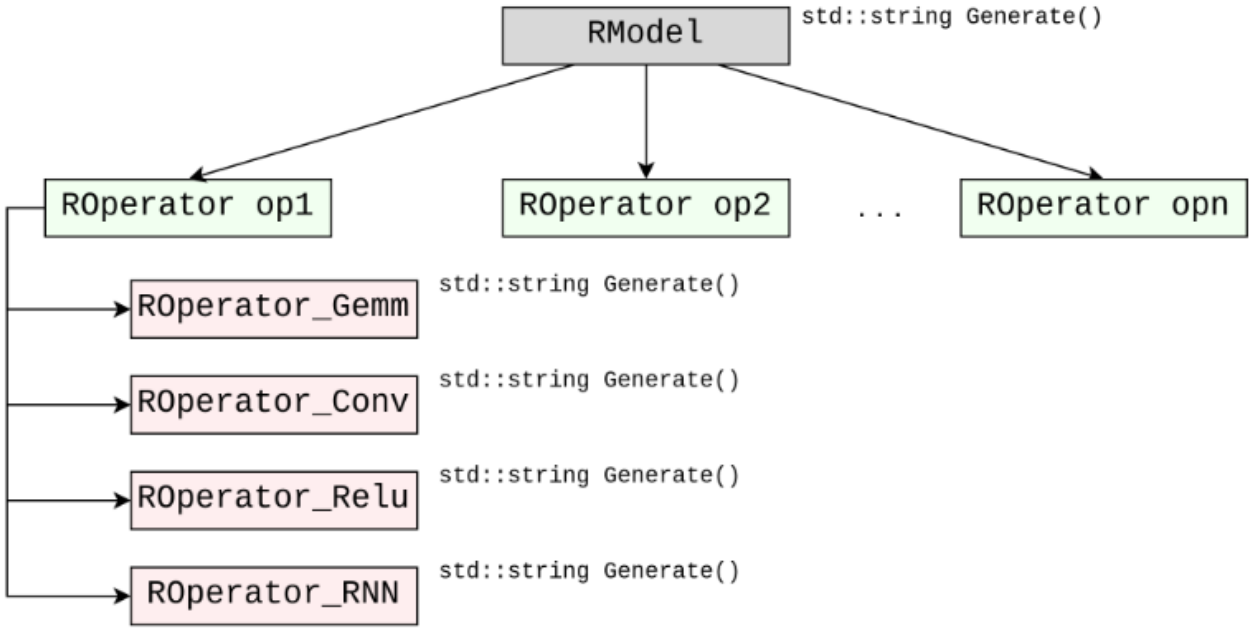

Figure 4: The RModel class holds a vector of ROperators. Each ONNX Operator corresponds to a different class that inherits from the ROperator interface.

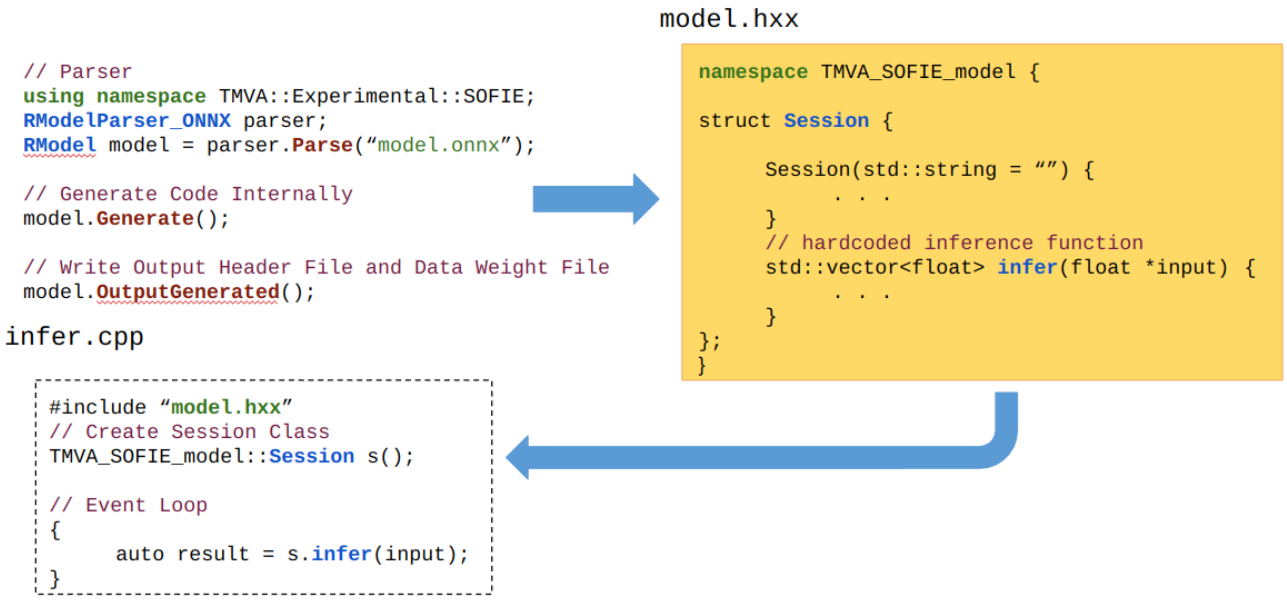

Figure 5: SOFIE C++ Code Generation Flow

Figure 6: Multi-compiler compilation model (left), Single-compiler compilation model (right) for SYCL

- **C++ programming model:** Perhaps the most important advantage of SYCL is that it allows programmers to write in standard C++ and doesn't rely on language extensions, pragmas or keywords like other languages do.
- **Targets:** SYCL can target a number of different backends, as shown in fig. 7. All SYCL implementations provide the same SYCL interface for both the host and the device code, as well as the SYCL runtime. The SYCL runtime is a library that schedules and executes work and calls down into a back-end interface in order to execute on a particular device. For our project, we used the Intel® oneAPI DPC++/C++ Compiler[7], since our primary target was Intel GPUs.

ii. SYCL Application Structure

In order to write a SYCL application similar to the one in 1, one must follow the steps below:

1. **Include the SYCL header**

   Including the SYCL header gives us access to the SYCL namespace.

2. **Setup host storage**

   This step includes setting up the vectors/arrays for the data we want to operate on.

3. **Initialize Device Selector**

   In order to operate on a device, we need to have some representation of it. A SYCL device selector is a function object, which describes a heuristic for scoring devices based on a custom configuration. The selector goes through all devices in the system and returns the one that scored

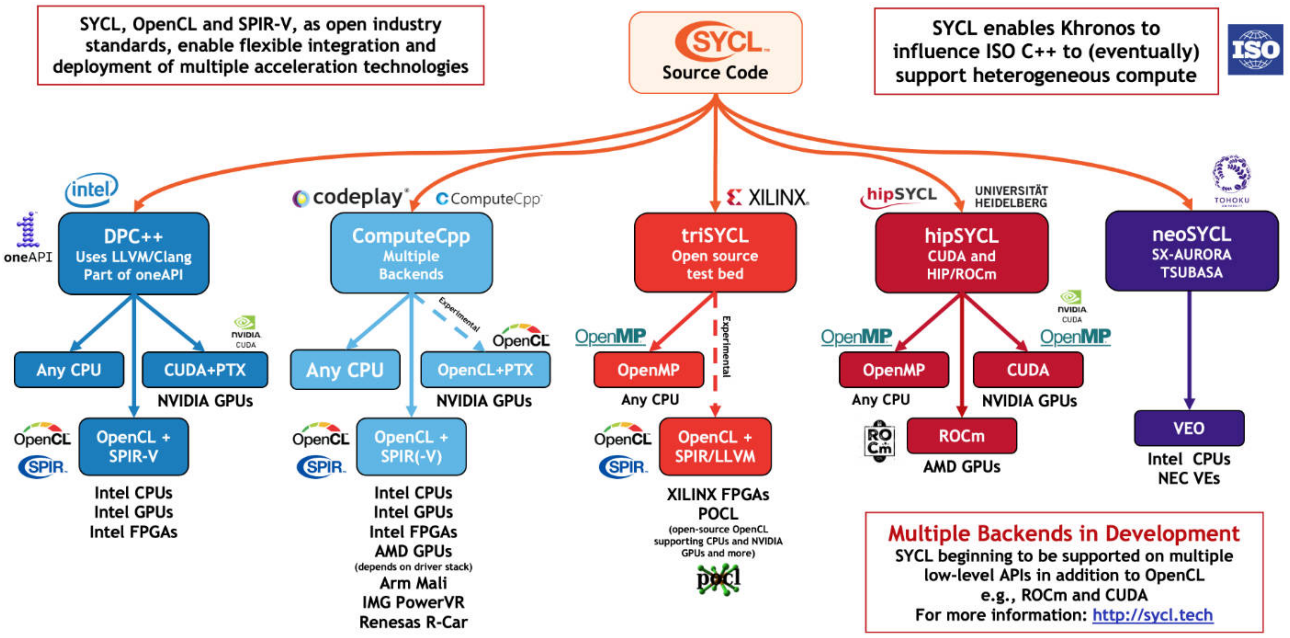

Figure 7: SYCL Implementations: compilers and backends

the highest depending on that heuristic. In the example, we use the `gpu_selector`, which selects a device of type GPU. One can define a function object that scores the devices with custom criteria, like platform or vendor.

4. **Initialize Queue**

   Each queue is associated with a chosen device and is used by the host CPU to communicate with the device, i.e. issue kernels and data transfers to and from the device.

5. **Setup Device Storage**

   In most systems, the host and the device do not share physical memory. The runtime needs to know which memory items are going to be shared between host and device. SYCL buffers exist for this purpose. To create a SYCL buffer, one must specify an element type and a dimensionality and initialize them with a pointer to the data and a range, which denotes the number of elements in the buffer. When passed a raw pointer, the buffer constructor takes full ownership of the memory it has been passed, which essentially means that we cannot use this memory as long as the buffer exists. Therefore, we declare the buffers in a new scope and after we exit this scope, the buffers are destroyed and the memory is returned to the user. Buffers are not associated with any particular queue, so they are capable of handling data transparently between multiple devices.

6. **Execute Kernel**

   In SYCL there are two models for managing data:

   - The buffer/accessor model
   - The USM (unified shared memory model)

   The chosen model affects how kernel functions are enqueued. In our project, we opted for the buffer/accessor model for reasons explained in section 2

   In the buffer/accessor model, commands must be enqueued via command groups. A command group represents a series of commands to be executed by a device, such as invoking kernel functions on a device, copying data to and from the device and waiting on other commands to

complete. A command group can be composed by calling the submit function on a queue. A handler is created and passed to the command group function and subsequently, the handler composes the command group. In our command group, we first setup accessors. In general, these objects define the inputs and outputs of a device-side operation. The accessors also provide access to various forms of memory. In this case, they allow us to access the memory owned by the buffers created earlier. An accessor is initialized with the buffer that points to the data we want to operate on, the associated command group handler and an accessor mode, which is used by the handler to handle the dependencies between kernels, as well as any additional properties we wish to add. In this case, the accessor mode for the input is `read_only` and for the output is `write_only`, with the additional property of `no_init`, which discards the original data of the buffer. We refer the reader to a. for a short description of the USM SYCL model.

In SYCL, kernel functions are executed by work items, which can be thought of as a thread of execution. Work items are collected together into work groups. SYCL kernels are invoked within an nd-range. An nd-range has a number of work groups and subsequently a number of work items. Kernel functions can be enqueued to execute over a range of work items using `parallel_for`. The `parallel_for` clause takes at a parameter a range which represents the iteration space, over which the kernel, which is described by a function object (in our case a lambda function), has to be executed over. With `parallel_for` you must also specify the `id` of the current-work item, which is essentially its position within the iteration space. See b. for more details.

```
#include <iostream>
// 1. Include SYCL Header
#include <CL/sycl.hpp>
namespace sycl = cl::sycl;

int main(int, char**) {
    // 2. Setup host storage
    std::vector<float> a = {1.0, 2.0, 3.0, 4.0};
    std::vector<float> b = {0.0, 0.0, 0.0, 0.0};
    auto length = a.size();

    // 3. Initialize device selector
    sycl::gpu_selector device_selector;

    // 4. Initialize queue
    sycl::queue queue(device_selector);

    { // begin scope
        // 5. Setup device storage
        sycl::buffer<float, 1> a_buf(a.data(), sycl::range<1>(length));
        sycl::buffer<float, 1> b_buf(b.data(), sycl::range<1>(length));

        // 6. Execute Kernel
        queue.submit([&] (sycl::handler& cgh) {
          // command group function
          auto a_acc = sycl::accessor(a_buf, cgh, sycl::write_only, sycl::no_init);
          auto b_acc = sycl::accessor(b_buf, cgh, sycl::read_only);

          cgh.parallel_for<class op>(sycl::range<1>(length), [=] (sycl::id<1> id) {
            // kernel code
```

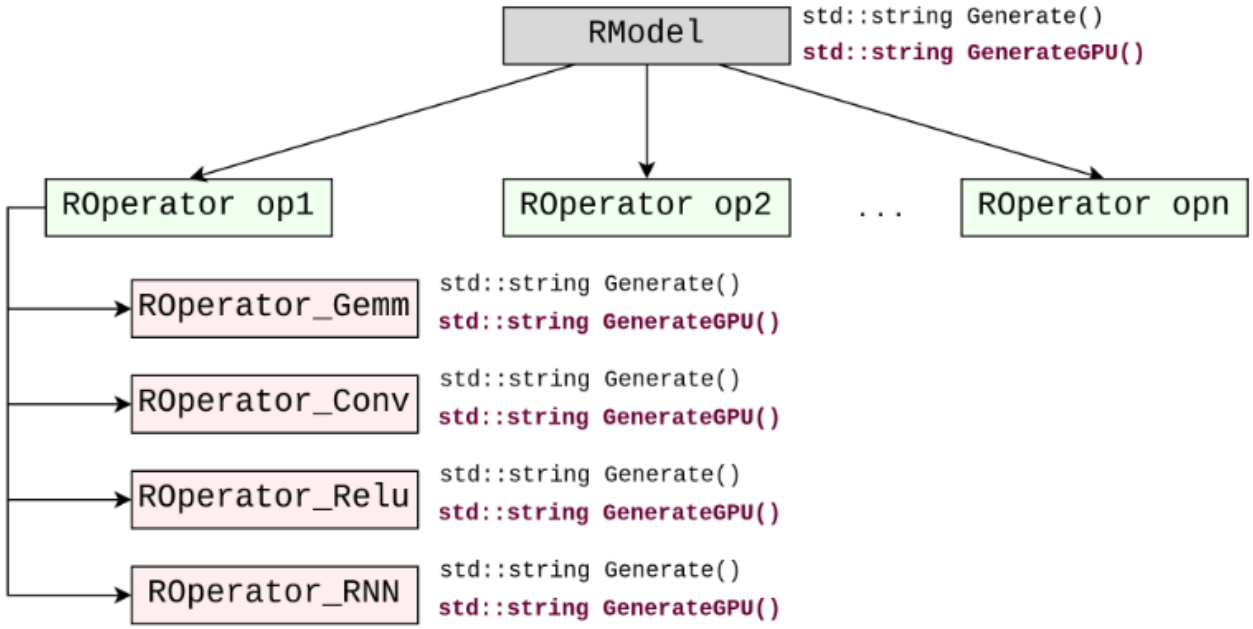

Figure 8: Modification of the RModel and ROperator classes for SYCL Inference Code Generation

```
            a_acc[id] = b_acc[id] * 2;
          });
        });
    }
    return 0;
}
```

Listing 1: An example of a SYCL application

2 IMPLEMENTATION

The process of adding to SOFIE the functionality to generate GPU SYCL code was pretty straightforward: the `RModel` class, as well as each operator class the inherited from `ROperator`, should be enhanced with a new `GenerateGPU` function that creates SYCL code instead of C++ code as shown in 8. The `GenerateGPU` member function of the `RModel` class is responsible for generating the code that handles device selection, queue initialization and setting up host and device storage, whereas the respective function of the `ROperator` interface produces SYCL code instead of C++ code as the `Generate` function did. An example of the latter is shown in 2.

```
// C++ Generated Code for ReLU activation
for (int id = 0; id < length ; id++){
    tensor_out[id] = ((tensor_in[id] > 0 )? tensor_in[id] : 0);
}

// SYCL Generated Kernel Code for ReLU activation
q.submit([&](cl::sycl::handler &cgh){
    auto acc_tensor_in = cl::sycl::accessor{buf_tensor_in, cgh, cl::sycl::read_only};
```

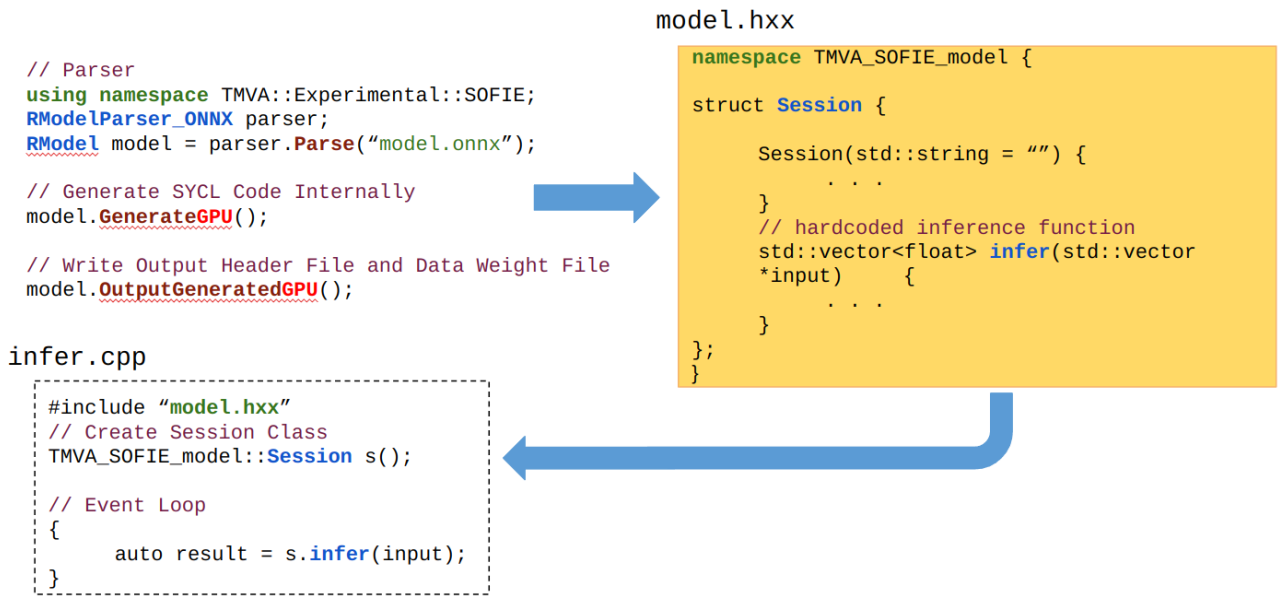

Figure 9: SOFIE Sycl Code Generation Flow

```
    auto acc_tensor_out = cl::sycl::accessor{buf_tensor_out, cgh, cl::sycl::write_only,
    cl::sycl::no_init};
    cgh.parallel_for<class op_relu>(cl::sycl::range<1>(length), [=](cl::sycl::id<1> id){
        acc_tensor_out[id] = cl::sycl::max(acc_tensor_in[id], 0.0f);
    });
});
```

Listing 2: C++ (top) and SYCL (bottom) generated code for ReLU activation.

The new code generation process is shown in fig. 9 and is practically the same as the one in fig. 5. The only change is in the infer function, which now takes as input an `std::vector<T>` instead of a pointer to `<T>` for implementation purposes.

A. BUFFER/ACCESSOR VS USM MODEL

The premise of SOFIE SYCL is that it would be able to support any type of Machine Learning Model. The buffer/accessor model guarantees consistency and avoids errors. The USM model gives us more fine grained control over data movement, but we have to manually establish the dependencies, which might lead to hidden bugs and wrong results. Therefore, we opted for the buffer/accessor model.

B. SPECIFYING WORK GROUP SIZE / ND-RANGE

We implemented the kernels in a way that no synchronization among the work-items is needed, so we did not need to explicitly specify the number of work-items in a work-group or the work-groups in an nd-range. Therefore, we just let the runtime choose those parameters for us.

3 PERFORMANCE CONSIDERATIONS

Although the transition to SYCL was a straightforward process, there were still details we had to consider in order to achieve the best possible performance[5]. This section highlights the most important ones.

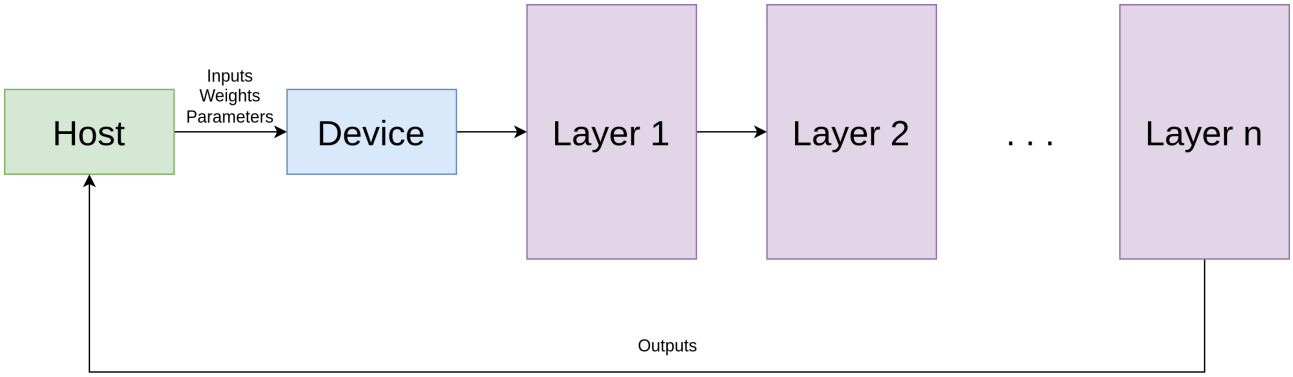

Figure 10: Abstract computation graph of an ML model for our implementation

1. **Avoid moving data back and forth between host and device**

   The cost of moving data between host and device is quite high, so it is very important to avoid data transfers between host and device whenever possible. Machine learning models are structured as layers of computation, where the output of one layer is input to the next one. Instead of transferring the data back for certain computations, we decided that it would be best that all layers are implemented on the GPU and that data is transferred from host to device only once in the beginning and from device to host only at the end of computation as shown in fig. 10. Keep in mind that those transfers in the buffer/accessor model are not explicit and are handled by the SYCL runtime, so it needs to be informed via accessor modes and buffer properties (explained below) of the data dependencies, so it can schedule transfers optimally.

2. **Buffer Accessor Modes and Properties**

   Accessor modes describe how we intend to use the memory associated with the accessor in the program. This information is used by the runtime to create an execution order for the kernels and perform data movement. It is therefore imperative that we specify the accessor modes in a way that coincides precisely with how the kernel actually uses this data. The available accessor modes are: `read_only`, `write_only` and `read_write`.

   - The `read_only` access mode informs the runtime that the data needs to be available on the device before the kernel can begin executing, but the data need not be copied from the device to the host at the end of the computation (when the associated buffer goes out of scope).
   - The `write_only` and `read_write` access modes inform the runtime that the data must be copied from the device to the host when the buffer goes out of scope.

   As explained before, moving data back and forth between host and device is costly. The accessor mode for the output of each layer has to be set to `write_only` (or `read_write` in some cases, such as accumulation). To avoid data transfers, we declare all the necessary buffers at the beginning of the scope and we don't close the scope until all kernels have been launched. In this way, we don't trigger copies back to the host in-between layers.

   In addition, we take advantage of the `set_final_data` buffer function. This function changes the destination the buffer will synchronize on destruction. Typically, the output that is inferred by the infer function is a different memory location than the output of the machine learning model, due to SOFIE semantics (see 3). In order to avoid the copy from the buffer to the temporary output memory location and then from that location to the result buffer, we instruct the runtime to directly copy the contents of the buffer to the result buffer on destruction (see 4). Furthermore,

for the input, initialized and intermediate tensors, we specify that the final data destination is `nullptr` which ensures that the data will remain on the device on destruction and will not be copied back to the host, since they are not needed.

```
namespace TMVA_SOFIE_Add{
struct Session {
std::vector<float> fTensor_2 = std::vector<float>(2);
float * tensor_2 = fTensor_2.data();

Session(std::string = "") {
}

std::vector<float> infer(float* tensor_onnxAdd0,float* tensor_onnxAdd1){

//------ Add
    for (size_t id = 0; id < 2 ; id++){
        tensor_2[id] = tensor_onnxAdd0[id] + tensor_onnxAdd1[id] ;
    }
   std::vector<float> ret (tensor_2, tensor_2 + 2);
   return ret;
}
};
} //TMVA_SOFIE_Add
```

Listing 3: C++ Generated code for vector addition. Notice that the contents of `tensor_2` that is the output of the add "kernel" are copied to vector `ret` which is the returned result of the `infer` function

```
// Create Queue
auto q = cl::sycl::queue{custom_gpu_selector, [=](cl::sycl::exception_list eL){
for (auto e:eL) {std::rethrow_exception(e);}}};
const sycl::property_list props = {sycl::property::buffer::use_host_ptr()};
{
auto buf_tensor_onnxAdd0 = cl::sycl::buffer{fTensor_onnxAdd0.data(),
cl::sycl::range{fTensor_onnxAdd0.size()}, props};
buf_tensor_onnxAdd0.set_final_data(nullptr);
auto buf_tensor_onnxAdd1 = cl::sycl::buffer{fTensor_onnxAdd1.data(),
cl::sycl::range{fTensor_onnxAdd1.size()}, props};
buf_tensor_onnxAdd1.set_final_data(nullptr);

auto buf_tensor_2 = cl::sycl::buffer{fTensor_2.data(),
cl::sycl::range{fTensor_2.size()}, props};

// change the final destination of the data held by buf_tensor_2
buf_tensor_2.set_final_data(ret.data());
buf_tensor_2.set_write_back(true);

//------ Add

q.submit([&](cl::sycl::handler& cgh){
    auto acc_tensor_onnxAdd0 = cl::sycl::accessor{buf_tensor_onnxAdd0, cgh,
    cl::sycl::read_only};
```

```
        auto acc_tensor_onnxAdd1 = cl::sycl::accessor{buf_tensor_onnxAdd1, cgh,
        cl::sycl::read_only};
        auto acc_tensor_2 = cl::sycl::accessor{buf_tensor_2, cgh,
        cl::sycl::write_only, cl::sycl::no_init};
        cgh.parallel_for<class op_0>(cl::sycl::range<1>(2), [=](cl::sycl::id<1> id){
            acc_tensor_2[id] = acc_tensor_onnxAdd0[id] + acc_tensor_onnxAdd1[id];
        });
    });

    q.wait_and_throw();
    }
    }
    catch (const cl::sycl::exception& e) {
        std::cout << "Exception caught: " << e.what() << "with OpenCL error code: "
        << e.code() << std::endl;
    }
    return ret;
    }
    };
    } //TMVA_SOFIE_Add
```

Listing 4: SYCL Generated code for vector addition. If we hadn't altered the final destination of the data held by `buf_tensor_2`, then they would be copied to vector `tensor_2` and then we would have to copy them to `ret`, but now they are directly coped to `ret`.

We also utilize the `sycl::no_init` property when creating the accessors that typically correspond to layer outputs, which lets the runtime know that the previous contents of the buffer can be discarded (usually accompanied by the `write_only` mode), so that no time is spent on initializing memory with "garbage" data.

Finally, when creating the buffers, we use the `use_host_ptr` property (see 4). This informs the runtime that if possible, the host memory should be directly used by the buffer instead of a copy. This avoids the need to copy the content of the buffer back and forth between the host memory and the buffer memory, potentially saving time during buffer creation and destruction.

3. **Using Libraries for GPU Offloading**

   Machine learning operators typically consist of standard math operations, such as matrix multiplication. Albeit simple, there is no need to write custom kernels for those operations, since they are well studied and very optimized libraries exist. For those, we made use of the oneAPI MKL[6] (Math Kernel Library) and more specifically BLAS routines, such as `copy`, `gemm`, `axpy` and `scal` (multiplication of a matrix with a scalar). The only downside is that we introduced additional library dependencies, which is a solid trade-off for performance.

4. **Reduction**

   Reduction is a common operation in parallel programming, where an operator is applied to all elements of an array and a single result is produced. A naive way to parallelize a reduction is to introduce a global variable and have all threads update it using an atomic operation. However, all threads then would access a single memory location, which would result in significant contention and poor performance. A better approach would be to split the array into chunks, let each thread compute part of the reduction and at the end have one thread do the final reduction in a sequential manner. This is a common and well studied approach in parallel computing. SYCL

Library	Supported GPU devices
Intel oneAPI Math Kernel Library	Intel GPU
portBLAS	Intel GPU, NVIDIA GPU, AMD GPU

Table 1: BLAS Library and Supported GPU devices

2020 introduced reduction variables, so we no longer need to write code by hand to handle a parallel reduction; SYCL handles it for use transparently. We refer the reader to [12] to learn more about this new feature.

5. **Kernel Fusion**

   For some subsequent operations, like clipping and activation, SOFIE C++ generated code used two subsequent loops. In our case, we fused the two subsequent loops into one kernel launch, potentially reducing execution time.

6. **Replacing Conditional Checks with Relational Functions**

   In GPUs, multiple work-items are packed into sub-groups. The work-items that belong to the same sub-group execute simultaneously on a SIMD processor. Given a SIMD width, maximizing SIMD lane utilization gives optimal instruction performance. If one or more lanes (work items) diverge, the thread executes both branch paths before the paths merge later, increasing the dynamic instruction count, which negatively affects performance and is widely known as branch divergence problem. To mitigate this problem, we replaced conditional checks with relational function wherever possible, to ensure that work-items do not execute different paths.

4 EXTENDING GPU MODEL SUPPORT

Although the initial target for our project was Intel GPUs with Intel oneAPI libraries, with minimal changes to our code, we can now support Intel, NVIDIA and AMD GPUs using portBLAS library for the BLAS routines. (see table 1)

5 BENCHMARKS

A. ROOTBENCH

The ROOTBench repository contains a set of benchmarks based on gbenchmark micro benchmarking infrastructure built on top of ROOT. Their primary goal is to provide stable performance metrics which can be monitored over time. An extension for benchmarking SOFIE models is also available. There is a number of .onnx models for which the header files with the inference function are generated during building the ROOTBench project. Then, we can run the `SOFIEInference` executable to derive the execution time per event in ms for each model. Table 2 provides a short description of each model. Visualizations for the more complex models are provided in c..

We added a new file `SOFIEGPUInference.cxx`, which is the same as the `SOFIEInference.cxx` file, but instantiates the SYCL versions of the models. The results of all benchmarks are listed in d.. The device specifications are listed in e..

B. GPU BENCHMARKS

We tested 3 different configurations: Intel GPU using MKL blas, Intel GPU using portBLAS and NVIDIA GPU using portBLAS. Figure 11 shows the time per event in milliseconds for a representative sample of benchmarks for each of those configurations.

Model	Description
Linear_{16, 32, 64}.onnx	10 FC layers followed by ReLU activation with hidden size = 50, input size = 100 and batch size = 16, 32, 64 respectively
Linear_event.onnx	10 FC layers followed by ReLU activation with hidden size = 50, input size = 100 and batch size = 1
Generator_{1, 64}.onnx	5 FC layers followed by Batch Normalization and ReLU activation with hidden sizes = {14, 20, 100, 500, 40500} and batch size = 1, 64 respectively
higgs_model_dense.onnx	6 FC layers followed by ReLU activation with hidden sizes = {7, 100, 100, 100, 100, 100} and batch size = 1
SimpleNN_Alice.onnx	3 FC layers followed by Leaky ReLU activation with hidden sizes = {16, 100, 50} and batch size = 1
SimpleNN_Alice.onnx	3 FC layers followed by Leaky ReLU activation with hidden sizes = {16, 100, 50} and batch size = 1
Conv_d100_L14_B{1, 32}.onnx	14 CONV2d layers followed by ReLU activation. Inputs dimensions = {B, 1, 100, 100}, Kernel dimensions = {D, 5, 5}, where B stands for batch size = {1, 32} depending on model, D is the kernel depth and for each layer it has a value of {2, 4, 8, 16, . . . , 128, 64, . . . , 2}. Padding for both height and width = 2 for every layer, so input height and width are preserved
Conv_d100_L14_B1.onnx	1 CONV2d layer followed by ReLU activation with input dimensions = {1, 1, 100, 100}, Kernel dimensions = {2, 5, 5} and Padding = {2, 2}
Conv3d_d32_L4_B1.onnx	5 CONV3d layers followed by ReLU activation. Inputs dimensions = {1, 1, 32, 32}, Kernel dimensions for the first 4 convolutions = {D, 5, 5, 5}, where D = {32, 8, 8, 8} and {4, 6, 6, 6} for the last layer. Padding = {1, 1, 1} for all layers

Table 2: Description of SOFIE ROOT benchmarks

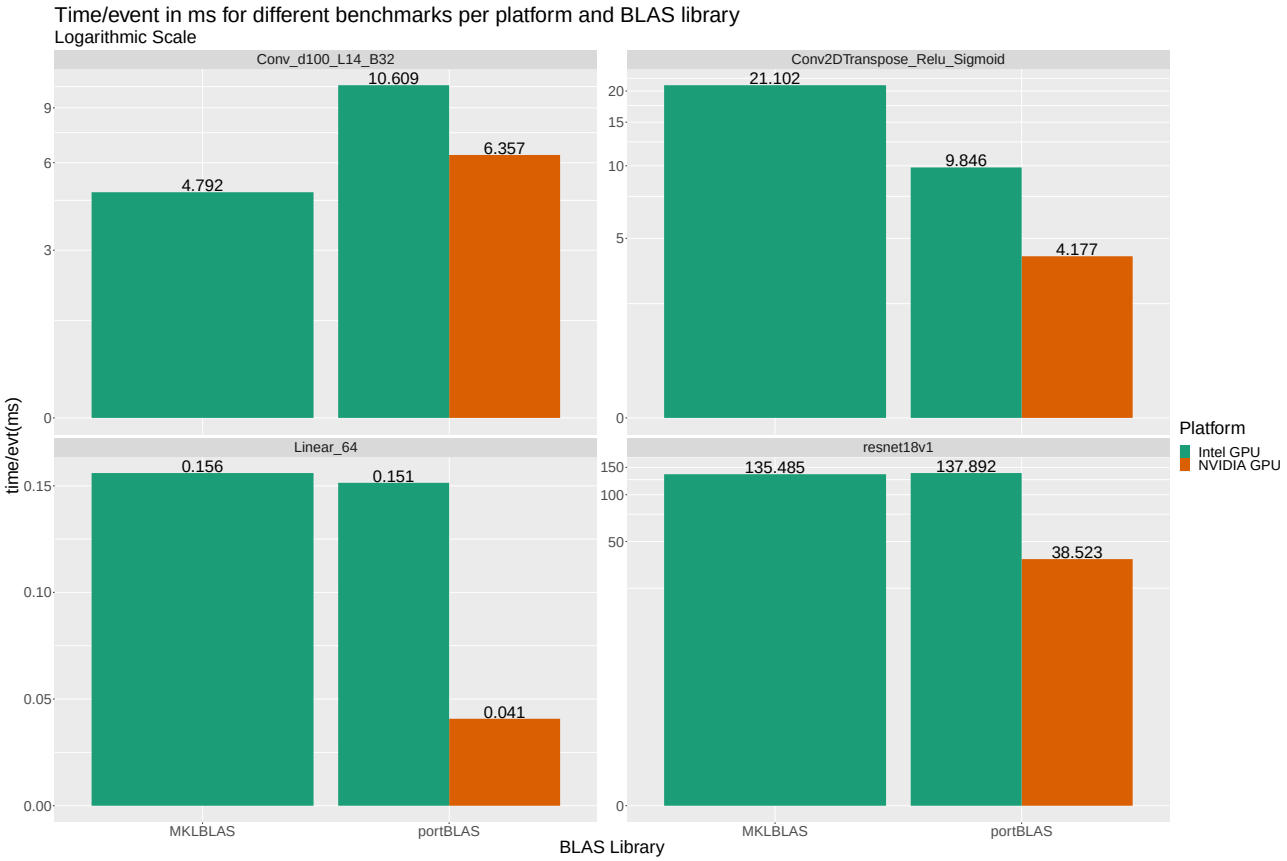

Figure 11: Time/event in ms for different benchmarks per GPU and BLAS library

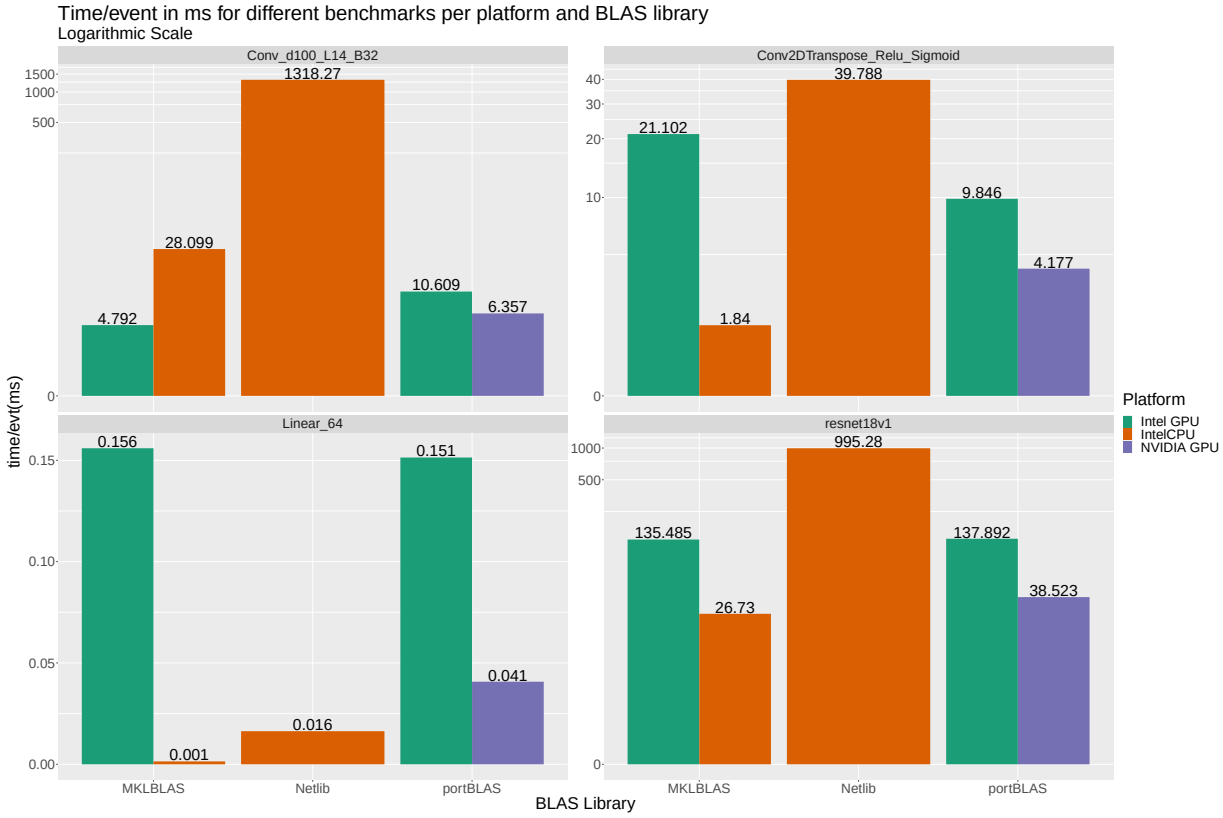

Figure 12: Time/event in ms for different benchmarks per platform and BLAS library

Since the Intel GPU and NVIDIA GPU are different devices with different specs (number of cores, memory bandwidth), there is no clear conclusion one could come to looking at the results. We could, however, say that the MKLBLAS library is better optimized compared to portBLAS, at least for Intel GPU devices. Even though at some cases, the performance with the portBLAS lib is better (Conv2DTranspose_Relu_Sigmoid and Linear_64), those networks have a small number of neurons and less layers, so they don't make much use of the BLAS libraries anyways and, hence, we cannot judge the performance difference of the BLAS libraries on them.

C. CPU VS GPU BENCHMARKS

For our next set of experiments, we compared our SOFIE SYCL implementation against the already existing SOFIE C++ Inference code, using both Netlib and MKLBLAS libraries (the version available for CPUs) on our available Intel CPU. From fig. 12, it is evident that for the convolutional models pictured and, in general, models with a lot of layers and computation, our SYCL GPU implementation is superior to the plain SOFIE C++ code. Also, performance significantly improves using the MKLBLAS lib compared with the Netlib for BLAS on the CPU. Lastly, for small models, such as Linear_64, GPU performance is much worse for all GPU configurations, which hints at the fact that the model is not large enough to take advantage of the GPU resources and that the overhead induced by data transfers cannot be compensated for by the computational power of the GPU.

D. ONNXRUNTIME CPU BENCHMARKS

ONNX also comes with its own runtime inference and training environment, called ONNXRUNTIME[4], which is well optimized for a number of different platforms, also called execution providers in ONNXRUNTIME terms. For our execution provider, we chose the Intel CPU we had at our disposal.

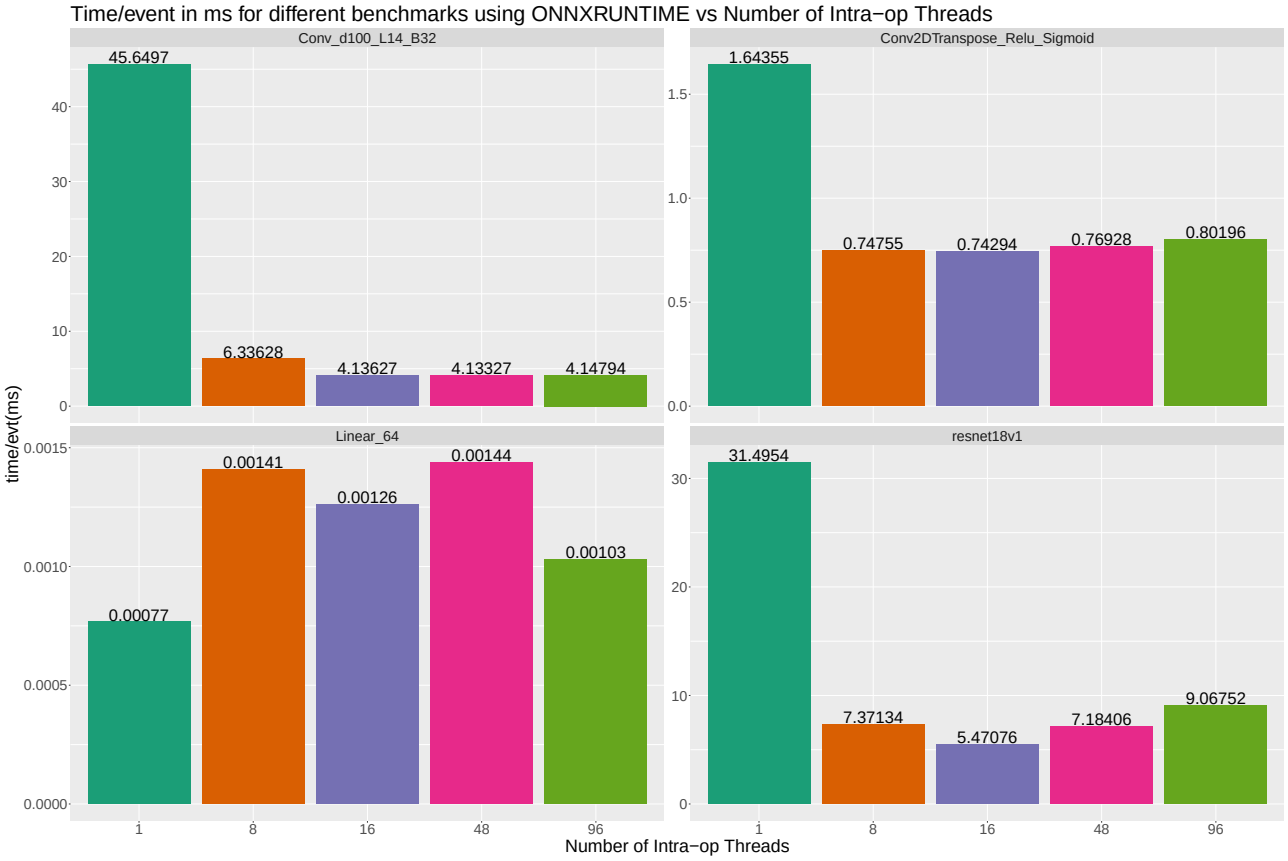

Figure 13: Time/event in ms using ONNXRUNTIME vs Number of intra-op threads

ONNXRUNTIME with a CPU provider can be configured to run with more that 1 threads. There is an option to increase the number of intra-op and inter-op threads. The intra-op threads are used to parallelize computation inside each operator and the inter-op threads are used for parallelism between operators. In both cases, a thread per physical core up to the number of user-specified threads will be created. Since most of our benchmarks are strictly sequential and there is no inter-op parallelism (except for some layers of the resnet), we didn't experiment with the number of inter-op threads, only with the number of intra-op threads from 1 to the maximum number of threads (96 in our case). The results of our experiments are shown in fig. 13.

Typically, multiple threads work better than just one thread, but increasing them beyond a certain point creates contention if there is not enough work to be done in parallel. In general, for our setup, a number of 16 intra-op thread works well for most benchmarks. Only exception is the Linear_64 model. As established in the previous section, the model is not large enough to take advantage of all the resources of a parallel GPU device and when it comes to CPU thread parallelism, it is no exception. In fact, it is much slower when deployed with more than 1 thread. So, a model like Linear_64, whose performance does not improve when running in parallel on a CPU, will probably be an unsuitable candidate for GPU offloading, as well.

E. COMPARISON BETWEEN INFERENCE ENGINES

Figure 14 gathers all results presented above in one plot, so it easy to compare between all the different inference engines. On average, every GPU implementation is faster than the SOFIE C++ code on the Intel GPU using Netlib and in some cases faster than the same code when using the MKL BLAS libraries. Again, that is not the case for Linear_64, where we can see that GPU devices perform very poorly. For Conv_d100_L14_B32 the Intel GPU-MKLBLAS library configuration beats all the other

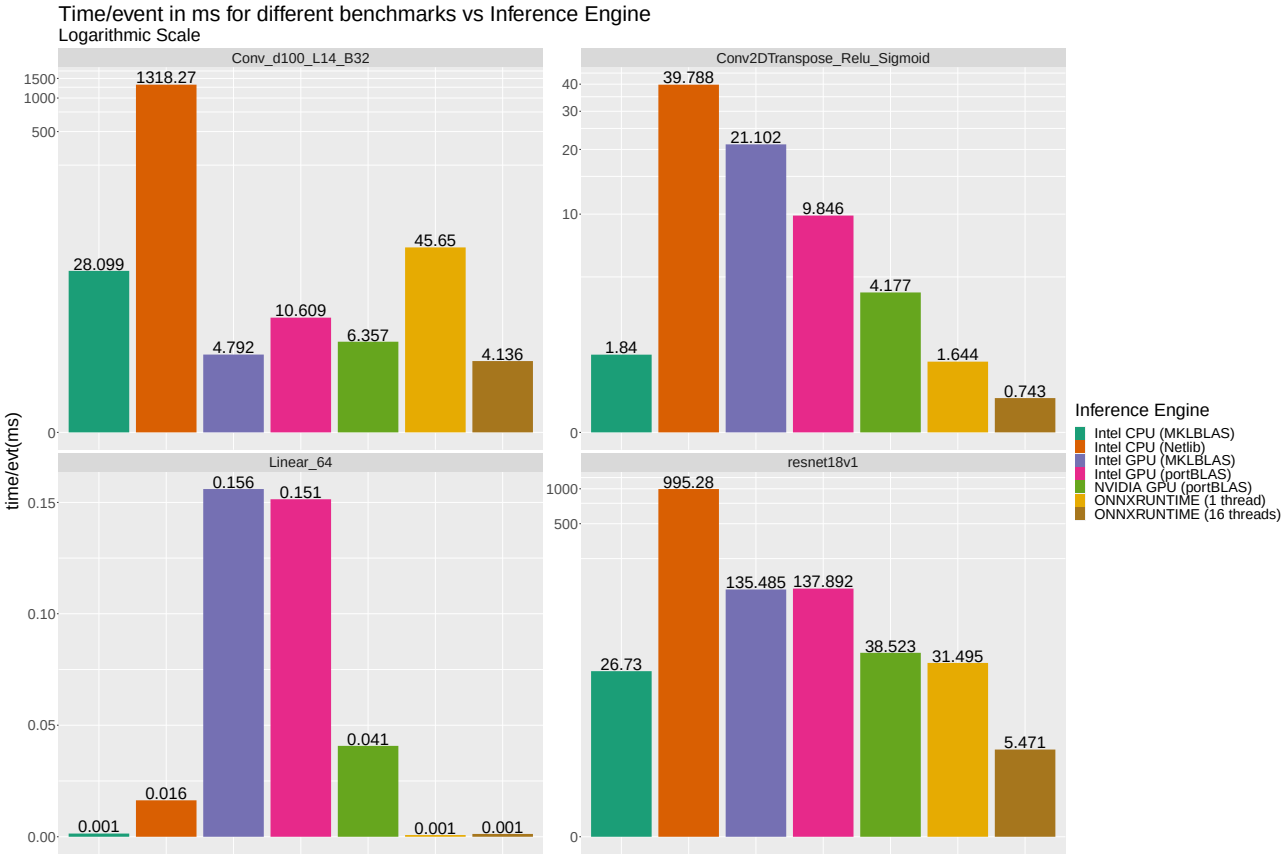

Figure 14: Comparison between different inference engines

configurations apart from the highly optimized ONNXRuntime engine with multiple threads.

Since our work comes as an extension to the existing SOFIE code, a comparison between them is necessary. Figure 15 shows the GPU speedup obtained from the (baseline) SOFIE C++ code. Numbers lower than 1 indicate worse performance. In almost all cases, again, except from Linear_64, GPU surpasses by far SOFIE C++ inference with Netlib and for large networks MKLBLAS. Keep in mind that SOFIE C++ inference uses only a single core and perhaps would benefit from exploiting parallelism (using OpenMP directives).

6 CONTRIBUTIONS

In this section, I will list some of the contributions I made to the public ROOT repository. At the time of writing this report, my pull requests have not been merged to the master branch, but work is currently being done to do so in the near future.

A. TMVA-SOFIE

My contribution to TMVA-SOFIE was three-fold. During my 2-month internship:

1. I added to SOFIE all the necessary functions needed to generate SYCL code in almost the exact same way that a user could generate C++ code before. This involved implementing the `GenerateGPU` function of the `RModel` and the `GenerateGPU` function for each of the operators that were already supported by SYCL. As an extension, I also provide to the user the option to use multiple BLAS libraries that target different GPUs.

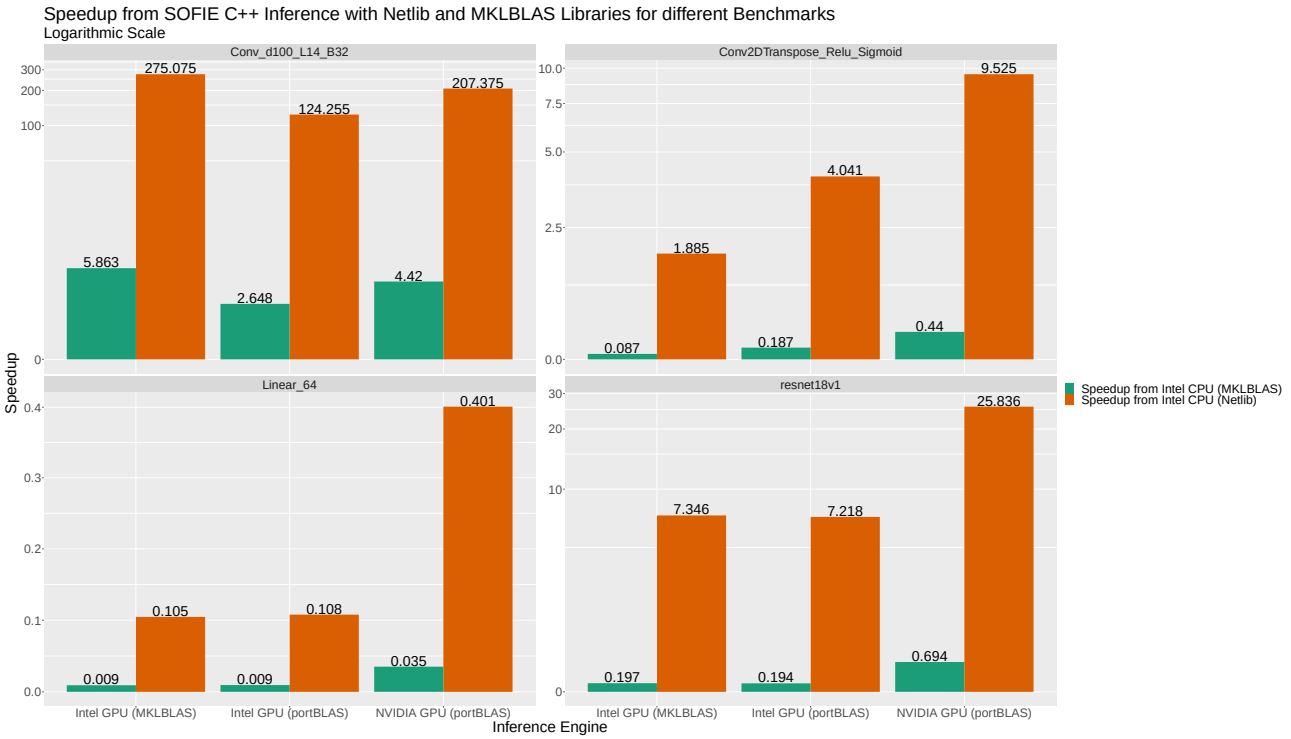

Figure 15: Speedup from existing SOFIE C++ Implementation

2. I added the implementation of the `ConvTranspose3d` Operator that was missing for both C++ and SYCL code generation.

3. I added tests that verify that the results of both C++ and SYCL SOFIE generated code are correct for the missing operators `BatchNormalization`, `Transpose`, `Slice` and `ConvTranpose3d`.

B. ROOTBENCH

My contribution to ROOTBench was that, based on the existing SOFIE benchmarking template, I added one for GPU Benchmarking and setup the whole project to work with the new SOFIE functionality.

7 FUTURE WORK

This project is still in an experimental stage, and, as shown from the benchmarking section, there is definitely room for optimization. Some of my proposed optimizations include:

- **Batch Normalization Folding:** A CONV/FC and a subsequent Batch Normalization layer could easily be merged into one layer by absorbing the BN parameters into the convolution/fully-connected network weights/biases, which would only require some preprocessing during code generation.

- **USM model:** In our current implementation, the runtime is responsible for data transfers and kernel scheduling taking into account the hints we have provided with the buffer accessor modes. Perhaps using the USM model could yield a better performance, since we would have fine-grained control over data transfers and kernel execution and could take decisions that the runtime wouldn't.

- **IntelDNN libraries:** In a next version of the project, someone could take advantage of the IntelDNN libraries, which are specialized for deep neural networks.
- **Manual control of work-group size and number of work groups:** Our implementation did not require work-item synchronization, so we didn't need to manually set the number of work groups and work-items per work group and just trusted the runtime to pick the optimal value. Potentially, manually setting the above parameters could lead to better performance.
- **In-place operations instead of intermediate buffers** SOFIE code uses intermediate buffers, when an operation (such as ReLU activation) could just be done in place, which unnecessarily wastes GPU memory.
- **Loop Unrolling:** Unrolling loops with pragmas or manually can limit the amount of control-instructions and loop variable increments/decrements and allow the compiler to move operations around in an optimal way thus potentially yielding better performance.

REFERENCES

[1] Sitong An et al. "SOFIE: C++ Code Generation for Fast Inference of Deep Learning Models in ROOT/TMVA". In: *Journal of Physics: Conference Series* 2438.1 (Feb. 2023), p. 012013. DOI: 10.1088/1742-6596/2438/1/012013. URL: https://dx.doi.org/10.1088/1742-6596/2438/1/012013.

[2] I. Antcheva et al. "ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization". In: *Computer Physics Communications* 182.6 (2011), pp. 1384–1385. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2011.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0010465511000701.

[3] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. 2019.

[4] ONNX Runtime developers. *ONNX Runtime*. https://onnxruntime.ai/. Version: x.y.z. 2021.

[5] *Intel oneAPI GPU Optimization Guide*. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/overview.html.

[6] *Intel oneAPI Math Kernel Library*. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-dpcpp/2023-1/introduction-to-the-intel-oneapi-math-kernel.html.

[7] *Intel® oneAPI DPC++/C++ Compiler*. https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-2/overview.html.

[8] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[9] *Netron: Visualizer for neural network, deep learning and machine learning models*. https://www.lutzroeder.com/ai.

[10] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[11] Ruyman Reyes et al. "Sycl 2020: More than meets the eye". In: *Proceedings of the International Workshop on OpenCL*. 2020, pp. 1–1.

[12] *SYCL 2020 Specification (revision 7) - Reduction*. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#sec:reduction.

[13] Jan Therhaag. "TMVA Toolkit for multivariate data analysis in ROOT". In: *PoS* ICHEP2010 (2010). Ed. by Bernard Pire et al., p. 510. DOI: 10.22323/1.120.0510.

8 APPENDIX

A. USM SYCL MODEL

The USM model allows a program to use C/C++ pointers for memory access. There are three ways to allocate memory in SYCL:

1. **Host:** The data is allocated on the host machine and stays there the whole time, but can be accessed from the device remotely through PCIe. High data access cost from the device.
2. **Device:** The data is allocated on the device and can only be accessed by that device only. Explicit data transfers are needed for the data to be accessible on the host or other devices. Fastest choice for kernel execution.
3. **Shared:** The allocated data can be accessed from both the host and the device. The runtime decides when the data migrates between host and device. No explicit copy is needed for the host and device associated with that memory allocation.

An example of the SYCL application in 1 in the USM model is shown below.

```
#include <iostream>
// 1. Include SYCL Header
#include <CL/sycl.hpp>
namespace sycl = cl::sycl;

int main(int, char**) {
    // 2. Setup host storage
    std::vector<float> a = {1.0, 2.0, 3.0, 4.0};
    std::vector<float> b = {0.0, 0.0, 0.0, 0.0};
    auto length = a.size();

    // 3. Initialize device selector
    sycl::gpu_selector device_selector;

    // 4. Initialize queue
    sycl::queue q(device_selector);

    // 5. Setup device storage
    auto a_dev = cl::sycl::malloc_device<float>(length, q);
    auto b_dev = cl::sycl::malloc_device<float>(length, q);

    // 6. Transfer input data to device
    auto e1 = q.memcpy(b_dev, b, sizeof(float) * length);

    // 7. Execute Kernel
    auto e2 = q.parallel_for<class op>(sycl::range<1>(length), {e1}, [=] (sycl::id<1> id) {
        a_acc[id] = b_acc[id] * 2;
    });

    // 8. Transfer output data to device
    auto e3 = q.memcpy(a.data(), a_dev, sizeof(float)*length, e2);
```

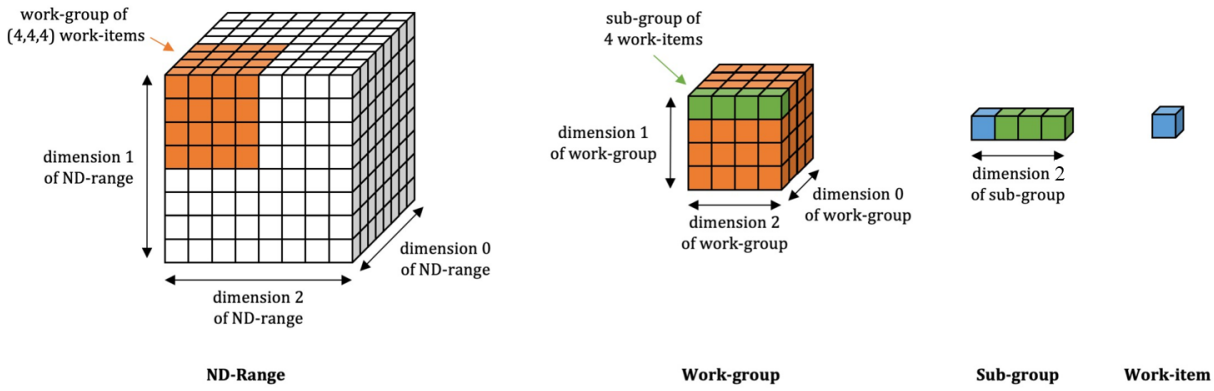

Figure 16: SYCL Thread execution model

```
    // 9. Call wait on the last event to make sure the data has returned to the host
    e3.wait();

    return 0;
}
```

Listing 5: An example of a SYCL application (USM model)

We can see that for the USM model we have to explicitly specify where the data will be allocated. Here we chose to allocate the data on the device. We no longer declare buffers that handle data movement transparently. After that, we also need to transfer the input data to the device using the `memcpy` operation. To guarantee that our data will have been transferred before executing the kernel we use events. Every operation we submit to the queue (`memcpy`, kernel execution) returns an event object, which can be used for synchronization. For example, copying vector `b` to the device returns an event `e1` that is then used in the dependency list of the kernel. This informs the runtime that event `e1` has to be completed before the kernel is executed.

B. SYCL EXECUTION MODEL

In SYCL kernel functions are executed by work-items. A work-item can be thought of as a thread of execution that can run on any type of processing element (PE). Work-items are collected together in work-groups. SYCL kernel functions are invoked within an nd-range. An nd-range has a number of work-groups and subsequently a number of work-items. A sub-group represents a short range of consecutive work-items that are processed together as a SIMD vector of length 8, 16, 32. Both the nd-range and the work-groups can be 1, 2 or 3-dimensional, as shown in 16. SYCL provides synchronization mechanisms for the work-items in the same work-group, but not across the entire nd-range. If the user does not need to synchronize the threads, then instead of specifying an nd-range, where the work-group size must also be specified, they can just use range instead of an nd-range. The runtime will take care of picking the (perhaps) optimal work-group size and nd-range.

C. VISUALIZATIONS FOR ROOTBENCH BENCHMARKS

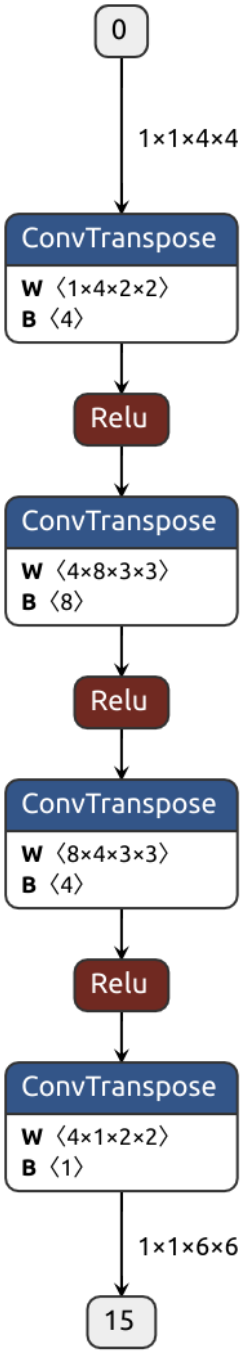

Figure 17: ConvTrans2dModel_B1

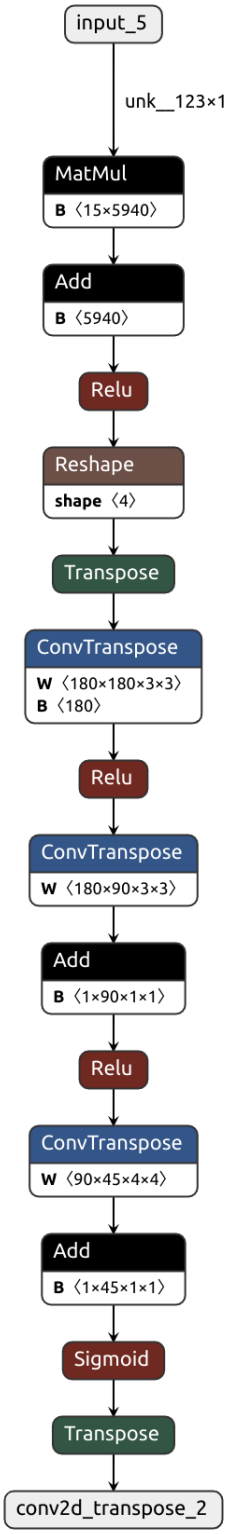

Figure 18: Conv2DTranspose_Relu_Sigmoid

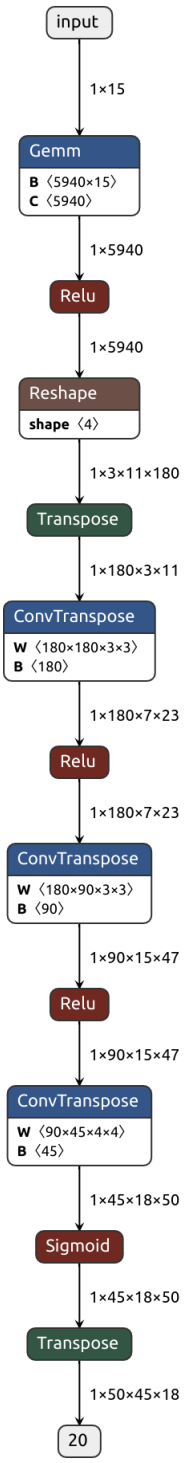

Figure 19: ConvTModel

D. BENCHMARK RESULTS

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
NVIDIA GPU	SOFIE-SYCL	portBLAS	Conv2DTranspose_Relu_Sigmoid	4.17739
NVIDIA GPU	SOFIE-SYCL	portBLAS	ConvTModel	4.17397
NVIDIA GPU	SOFIE-SYCL	portBLAS	ConvTrans2dModel	2.56897
NVIDIA GPU	SOFIE-SYCL	portBLAS	SimpleNN_Alice	2.1528
NVIDIA GPU	SOFIE-SYCL	portBLAS	Linear_16	0.162751
NVIDIA GPU	SOFIE-SYCL	portBLAS	Linear_32	0.081895
NVIDIA GPU	SOFIE-SYCL	portBLAS	Linear_64	0.0407422
NVIDIA GPU	SOFIE-SYCL	portBLAS	Linear_event	2.60755
NVIDIA GPU	SOFIE-SYCL	portBLAS	Generator_B1	5.03211
NVIDIA GPU	SOFIE-SYCL	portBLAS	Generator_B64	0.13425
NVIDIA GPU	SOFIE-SYCL	portBLAS	higgs_model_dense	2.34856
NVIDIA GPU	SOFIE-SYCL	portBLAS	Conv_d100_L14_B1	52.0865
NVIDIA GPU	SOFIE-SYCL	portBLAS	Conv_d100_L14_B32	6.35693
NVIDIA GPU	SOFIE-SYCL	portBLAS	Conv_d100_L1_B1	2.32522
NVIDIA GPU	SOFIE-SYCL	portBLAS	Conv_3d_d32_L4_B1	69.7217
NVIDIA GPU	SOFIE-SYCL	portBLAS	resnet18v1	38.5228

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
Intel GPU	SOFIE-SYCL	MKLBLAS	Conv2DTranspose_Relu_Sigmoid	21.1021
Intel GPU	SOFIE-SYCL	MKLBLAS	ConvTModel	13.1495
Intel GPU	SOFIE-SYCL	MKLBLAS	ConvTrans2dModel	11.5645
Intel GPU	SOFIE-SYCL	MKLBLAS	SimpleNN_Alice	4.33779
Intel GPU	SOFIE-SYCL	MKLBLAS	Linear_16	0.559951
Intel GPU	SOFIE-SYCL	MKLBLAS	Linear_32	0.294952
Intel GPU	SOFIE-SYCL	MKLBLAS	Linear_64	0.156017
Intel GPU	SOFIE-SYCL	MKLBLAS	Linear_event	8.12663
Intel GPU	SOFIE-SYCL	MKLBLAS	Generator_B1	14.6668
Intel GPU	SOFIE-SYCL	MKLBLAS	Generator_B64	0.337062
Intel GPU	SOFIE-SYCL	MKLBLAS	higgs_model_dense	5.37862
Intel GPU	SOFIE-SYCL	MKLBLAS	Conv_d100_L14_B1	80.6941
Intel GPU	SOFIE-SYCL	MKLBLAS	Conv_d100_L14_B32	4.7924
Intel GPU	SOFIE-SYCL	MKLBLAS	Conv_d100_L1_B1	5.26741
Intel GPU	SOFIE-SYCL	MKLBLAS	Conv_3d_d32_L4_B1	47.0783
Intel GPU	SOFIE-SYCL	MKLBLAS	resnet18v1	135.485

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
Intel GPU	SOFIE-SYCL	portBLAS	Conv2DTranspose_Relu_Sigmoid	9.84564
Intel GPU	SOFIE-SYCL	portBLAS	ConvTModel	9.98284
Intel GPU	SOFIE-SYCL	portBLAS	ConvTrans2dModel	7.33748
Intel GPU	SOFIE-SYCL	portBLAS	SimpleNN_Alice	4.29335
Intel GPU	SOFIE-SYCL	portBLAS	Linear_16	0.572602
Intel GPU	SOFIE-SYCL	portBLAS	Linear_32	0.281836
Intel GPU	SOFIE-SYCL	portBLAS	Linear_64	0.151426
Intel GPU	SOFIE-SYCL	portBLAS	Linear_event	8.3133
Intel GPU	SOFIE-SYCL	portBLAS	Generator_B1	13.0937
Intel GPU	SOFIE-SYCL	portBLAS	Generator_B64	0.369891
Intel GPU	SOFIE-SYCL	portBLAS	higgs_model_dense	5.41202
Intel GPU	SOFIE-SYCL	portBLAS	Conv_d100_L14_B1	64.5675
Intel GPU	SOFIE-SYCL	portBLAS	Conv_d100_L14_B32	10.6094
Intel GPU	SOFIE-SYCL	portBLAS	Conv_d100_L1_B1	5.58968
Intel GPU	SOFIE-SYCL	portBLAS	Conv_3d_d32_L4_B1	49.3109
Intel GPU	SOFIE-SYCL	portBLAS	resnet18v1	137.892

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
IntelCPU	ONNXRUNTIME (1 thread)	-	Conv2DTranspose_Relu_Sigmoid	1.64355
IntelCPU	ONNXRUNTIME (1 thread)	-	ConvTModel	1.69358
IntelCPU	ONNXRUNTIME (1 thread)	-	ConvTrans2dModel	0.00829917
IntelCPU	ONNXRUNTIME (1 thread)	-	SimpleNN_Alice	0.000428467
IntelCPU	ONNXRUNTIME (1 thread)	-	Linear_16	0.00112433
IntelCPU	ONNXRUNTIME (1 thread)	-	Linear_32	0.000900946
IntelCPU	ONNXRUNTIME (1 thread)	-	Linear_64	0.000766813
IntelCPU	ONNXRUNTIME (1 thread)	-	Linear_event	0.00829659
IntelCPU	ONNXRUNTIME (1 thread)	-	Generator_B1	0.479121
IntelCPU	ONNXRUNTIME (1 thread)	-	Generator_B64	0.0955
IntelCPU	ONNXRUNTIME (1 thread)	-	higgs_model_dense	0.00723121
IntelCPU	ONNXRUNTIME (1 thread)	-	Conv_d100_L14_B1	91.5335
IntelCPU	ONNXRUNTIME (1 thread)	-	Conv_d100_L14_B32	45.6497
IntelCPU	ONNXRUNTIME (1 thread)	-	Conv_d100_L1_B1	0.0837463
IntelCPU	ONNXRUNTIME (1 thread)	-	Conv_3d_d32_L4_B1	71.0397
IntelCPU	ONNXRUNTIME (1 thread)	-	resnet18v1	31.4954

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
IntelCPU	ONNXRUNTIME (8 threads)	-	Conv2DTranspose_Relu_Sigmoid	0.74755
IntelCPU	ONNXRUNTIME (8 threads)	-	ConvTModel	0.797615
IntelCPU	ONNXRUNTIME (8 threads)	-	ConvTrans2dModel	0.00858503
IntelCPU	ONNXRUNTIME (8 threads)	-	SimpleNN_Alice	0.000500897
IntelCPU	ONNXRUNTIME (8 threads)	-	Linear_16	0.00127948
IntelCPU	ONNXRUNTIME (8 threads)	-	Linear_32	0.00232777
IntelCPU	ONNXRUNTIME (8 threads)	-	Linear_64	0.00140783
IntelCPU	ONNXRUNTIME (8 threads)	-	Linear_event	0.00825026
IntelCPU	ONNXRUNTIME (8 threads)	-	Generator_B1	0.129205
IntelCPU	ONNXRUNTIME (8 threads)	-	Generator_B64	0.043
IntelCPU	ONNXRUNTIME (8 threads)	-	higgs_model_dense	0.00737023
IntelCPU	ONNXRUNTIME (8 threads)	-	Conv_d100_L14_B1	12.1657
IntelCPU	ONNXRUNTIME (8 threads)	-	Conv_d100_L14_B32	6.33628
IntelCPU	ONNXRUNTIME (8 threads)	-	Conv_d100_L1_B1	0.0185869
IntelCPU	ONNXRUNTIME (8 threads)	-	Conv_3d_d32_L4_B1	9.31684
IntelCPU	ONNXRUNTIME (8 threads)	-	resnet18v1	7.37134

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
IntelCPU	ONNXRUNTIME (16 threads)	-	Conv2DTranspose_Relu_Sigmoid	0.742937
IntelCPU	ONNXRUNTIME (16 threads)	-	ConvTModel	0.81772
IntelCPU	ONNXRUNTIME (16 threads)	-	ConvTrans2dModel	0.00851791
IntelCPU	ONNXRUNTIME (16 threads)	-	SimpleNN_Alice	0.000512725
IntelCPU	ONNXRUNTIME (16 threads)	-	Linear_16	0.00120839
IntelCPU	ONNXRUNTIME (16 threads)	-	Linear_32	0.00254811
IntelCPU	ONNXRUNTIME (16 threads)	-	Linear_64	0.00126199
IntelCPU	ONNXRUNTIME (16 threads)	-	Linear_event	0.00827918
IntelCPU	ONNXRUNTIME (16 threads)	-	Generator_B1	0.0653744
IntelCPU	ONNXRUNTIME (16 threads)	-	Generator_B64	0.0412656
IntelCPU	ONNXRUNTIME (16 threads)	-	higgs_model_dense	0.00742863
IntelCPU	ONNXRUNTIME (16 threads)	-	Conv_d100_L14_B1	6.97009
IntelCPU	ONNXRUNTIME (16 threads)	-	Conv_d100_L14_B32	4.13627
IntelCPU	ONNXRUNTIME (16 threads)	-	Conv_d100_L1_B1	0.0180515
IntelCPU	ONNXRUNTIME (16 threads)	-	Conv_3d_d32_L4_B1	5.15327
IntelCPU	ONNXRUNTIME (16 threads)	-	resnet18v1	5.47076

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
IntelCPU	ONNXRUNTIME (48 threads)	-	Conv2DTranspose_Relu_Sigmoid	0.769284
IntelCPU	ONNXRUNTIME (48 threads)	-	ConvTModel	0.835453
IntelCPU	ONNXRUNTIME (48 threads)	-	ConvTrans2dModel	0.00887564
IntelCPU	ONNXRUNTIME (48 threads)	-	SimpleNN_Alice	0.000526848
IntelCPU	ONNXRUNTIME (48 threads)	-	Linear_16	0.00128761
IntelCPU	ONNXRUNTIME (48 threads)	-	Linear_32	0.0025463
IntelCPU	ONNXRUNTIME (48 threads)	-	Linear_64	0.00143907
IntelCPU	ONNXRUNTIME (48 threads)	-	Linear_event	0.00862842
IntelCPU	ONNXRUNTIME (48 threads)	-	Generator_B1	0.0628781
IntelCPU	ONNXRUNTIME (48 threads)	-	Generator_B64	0.0542969
IntelCPU	ONNXRUNTIME (48 threads)	-	higgs_model_dense	0.00753104
IntelCPU	ONNXRUNTIME (48 threads)	-	Conv_d100_L14_B1	7.02565
IntelCPU	ONNXRUNTIME (48 threads)	-	Conv_d100_L14_B32	4.13327
IntelCPU	ONNXRUNTIME (48 threads)	-	Conv_d100_L1_B1	0.0178603
IntelCPU	ONNXRUNTIME (48 threads)	-	Conv_3d_d32_L4_B1	5.09109
IntelCPU	ONNXRUNTIME (48 threads)	-	resnet18v1	7.18406

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
IntelCPU	ONNXRUNTIME (96 threads)	-	Conv2DTranspose_Relu_Sigmoid	0.80196
IntelCPU	ONNXRUNTIME (96 threads)	-	ConvTModel	0.805271
IntelCPU	ONNXRUNTIME (96 threads)	-	ConvTrans2dModel	0.000879745
IntelCPU	ONNXRUNTIME (96 threads)	-	SimpleNN_Alice	0.000466415
IntelCPU	ONNXRUNTIME (96 threads)	-	Linear_16	0.00122956
IntelCPU	ONNXRUNTIME (96 threads)	-	Linear_32	0.00281769
IntelCPU	ONNXRUNTIME (96 threads)	-	Linear_64	0.00103089
IntelCPU	ONNXRUNTIME (96 threads)	-	Linear_event	0.0084097
IntelCPU	ONNXRUNTIME (96 threads)	-	Generator_B1	0.0621012
IntelCPU	ONNXRUNTIME (96 threads)	-	Generator_B64	0.05525
IntelCPU	ONNXRUNTIME (96 threads)	-	higgs_model_dense	0.00742874
IntelCPU	ONNXRUNTIME (96 threads)	-	Conv_d100_L14_B1	6.96241
IntelCPU	ONNXRUNTIME (96 threads)	-	Conv_d100_L14_B32	4.14794
IntelCPU	ONNXRUNTIME (96 threads)	-	Conv_d100_L1_B1	0.0178356
IntelCPU	ONNXRUNTIME (96 threads)	-	Conv_3d_d32_L4_B1	5.08413
IntelCPU	ONNXRUNTIME (96 threads)	-	resnet18v1	9.06752

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
IntelCPU	SOFIE C++ Inference	Netlib	Conv2DTranspose_Relu_Sigmoid	39.788
IntelCPU	SOFIE C++ Inference	Netlib	ConvTModel	39.9102
IntelCPU	SOFIE C++ Inference	Netlib	ConvTrans2dModel	0.00347443
IntelCPU	SOFIE C++ Inference	Netlib	SimpleNN_Alice	0.00331052
IntelCPU	SOFIE C++ Inference	Netlib	Linear_16	0.0164009
IntelCPU	SOFIE C++ Inference	Netlib	Linear_32	0.0163644
IntelCPU	SOFIE C++ Inference	Netlib	Linear_64	0.0163313
IntelCPU	SOFIE C++ Inference	Netlib	Linear_event	0.0166636
IntelCPU	SOFIE C++ Inference	Netlib	Generator_B1	2.27197
IntelCPU	SOFIE C++ Inference	Netlib	Generator_B64	2.09541
IntelCPU	SOFIE C++ Inference	Netlib	higgs_model_dense	0.0199256
IntelCPU	SOFIE C++ Inference	Netlib	Conv_d100_L14_B1	2696.81
IntelCPU	SOFIE C++ Inference	Netlib	Conv_d100_L14_B32	1318.27
IntelCPU	SOFIE C++ Inference	Netlib	Conv_d100_L1_B1	0.321535
IntelCPU	SOFIE C++ Inference	Netlib	Conv_3d_d32_L4_B1	564.539
IntelCPU	SOFIE C++ Inference	Netlib	resnet18v1	995.28

Platform	Inference Engine	BLAS Library	Benchmark	time/evt (ms)
IntelCPU	SOFIE C++ Inference	MKLBLAS	Conv2DTranspose_Relu_Sigmoid	1.8397
IntelCPU	SOFIE C++ Inference	MKLBLAS	ConvTModel	1.50859
IntelCPU	SOFIE C++ Inference	MKLBLAS	ConvTrans2dModel	0.00142627
IntelCPU	SOFIE C++ Inference	MKLBLAS	SimpleNN_Alice	0.000721473
IntelCPU	SOFIE C++ Inference	MKLBLAS	Linear_16	0.0012527
IntelCPU	SOFIE C++ Inference	MKLBLAS	Linear_32	0.00101778
IntelCPU	SOFIE C++ Inference	MKLBLAS	Linear_64	0.001431585
IntelCPU	SOFIE C++ Inference	MKLBLAS	Linear_event	0.00403714
IntelCPU	SOFIE C++ Inference	MKLBLAS	Generator_B1	0.325892
IntelCPU	SOFIE C++ Inference	MKLBLAS	Generator_B64	0.337125
IntelCPU	SOFIE C++ Inference	MKLBLAS	higgs_model_dense	0.00389043
IntelCPU	SOFIE C++ Inference	MKLBLAS	Conv_d100_L14_B1	64.2373
IntelCPU	SOFIE C++ Inference	MKLBLAS	Conv_d100_L14_B32	28.0985
IntelCPU	SOFIE C++ Inference	MKLBLAS	Conv_d100_L1_B1	0.268126
IntelCPU	SOFIE C++ Inference	MKLBLAS	Conv_3d_d32_L4_B1	69.9129
IntelCPU	SOFIE C++ Inference	MKLBLAS	resnet18v1	26.7298

E. DEVICE SPECIFICATIONS

Device	Intel GPU	NVIDIA GPU
Device Name	Intel Arctic Sound-P (2-tile)	NVIDIA GeForce RTX 3060
Subslices / Multiprocessors	30 (x2)	28
EUs per Subslice / CUDA Cores per MP	16	128
Number of Threads per EU	8	-
Total EU Count	480 (x2)	-
Total number of threads / CUDA Cores	3840 (x2)	3584
GPU Max Clock rate	1.4 GHz	1.78 GHz
Memory Clock	1400	1875 MHz
Memory Type	HBM2e	GDDR6
Memory Bus	2048-bit (x2)	192-bit
Memory Bandwidth	716.8GB/s (x2)	360GB/s
Global Memory	16GB (x2)	12053 MB

Table 3: NVIDIA and Intel GPU Specifications

Device	Intel CPU
Device Name	Intel Xeon Gold 6336Y CPU @ 2.40GHz
Architecture	x86_64
CPU(s)	96
Thread(s) per core	2
Core(s) per socket	24
Socket(s)	2
Frequency MHz	2400.00
CPU max MHz	3600.00
CPU min MHz	800.00
L1d	48KiB (x48)
L1i	32KiB (x48)
L2 Unified	1280KiB (x48)
L3 Unified	36864 KiB (x2)

Table 4: Intel CPU Specifications