TMVA SOFIE: GPU Support for Machine Learning Inference

GSoC 2025 Project Proposal

Organization: CERN HSF

Mentors:	Dr. Lorenzo Moneta	lorenzo.moneta@cern.ch
	Sanjiban Sengupta	sanjiban.sengupta@cern.ch

Contact information:

Full name:	Prasanna Sanjay Kasar		
Degree:	Bachelor of Technology in Computer Engineering (B. Tech)		
University:	Veermata Jijabai Technological Institute (VJTI), Mumbai		
Location:	Mumbai, Maharashtra, India		
Time Zone:	Indian Standard Time (UTC+5.30)		
Contact:	**Email**	:	prasanna.kasar06@gmail.com
	GitHub	:	PrasannaKasar
	LinkedIn	:	Prasanna Kasar
Resume:	Link		

Introduction

Hi, I'm Prasanna Kasar, and I'm currently in my second year of BTech in Computer Engineering at Veermata Jijabai Technological Institute, Mumbai.

I've developed several deep learning projects using Python, focusing on technologies like Transformers, CNNs, and RNNs, with frameworks such as PyTorch and TensorFlow. Here are a couple of my recent projects:

1. **Text-to-Speech:** Created a **Transformer-based Text-to-Speech** system, where I built a modular pipeline to convert raw text into **natural-sounding speech**. I used the Transformer's **Attention mechanism** in the front end to convert text to **mel spectrograms** and implemented the **Griffin-Lim algorithm** in the back end to reconstruct speech from the predicted **mel spectrograms**.

2. **Image Captioning:** Developed an **Image Captioning model** using **Inception v3** and **LSTM** on the **MS COCO dataset** to generate contextually relevant captions from images. Utilized **CNN** to extract **visual features** in the image and **LSTM** to process these features to generate coherent and descriptive captions.

I have a strong understanding of programming languages like Python, C/C++, and Java, and I'm also familiar with JavaScript. I've worked with C to solve various tasks and challenges given to me for this project. I am interested in **GPU programming** and am actively learning about it.

Past Open Source Experience

I began my open-source journey with **Project X** at university, which closely resembles the Google Summer of Code (GSoC) process. I worked on a Text-to-Speech system, gaining experience with libraries like **PyTorch**, **Librosa,** and **NumPy,** and learned how to use **Git** and **GitHub** effectively.

After completing these projects, I participated in **HacktoberFest 2024**, merging all four pull requests before the deadline.

During this year's GSoC, I encountered challenges while building **ROOT** on Windows, so I switched to **Linux**. This made the process easier and provided me with greater control over the system.

Time Commitment

I can dedicate 7 to 8 hours per day to the project, which would amount to 49-52 hours per week. I am open to increasing this duration if needed. As I do not have any strict commitments during this summer, I can easily adjust my schedule and work for longer hours if required. I am flexible with working hours and can adapt them according to my mentor's time zone. My summer vacation is from May 31 to July 31, so a significant portion of the work can be completed during this period.

Pre-GSoC Evaluation Tasks

I have submitted the **preliminary tasks**, as well as the **project-specific tasks,** to the Project Mentors Dr. Lorenzo Moneta and Sanjiban Sengupta.

GitHub **personal development branch** of ROOT: Link

Preliminary task document: Link

GPU Support for Machine Learning Inference task document: Link

Project Description

The **SOFIE (System for Optimized Fast Inference Code Emit)** project is an initiative within the **TMVA (Toolkit for Multivariate Data Analysis)** framework in ROOT, which aims to enhance the efficiency and speed of inference on Machine Learning Models. SOFIE converts ML models trained in different frameworks such as ONNX, PyTorch, and Tensorflow, converting them into an **Intermediate Representation (IR)**. This IR allows SOFIE to generate optimized C++ functions for fast and effective inference of neural networks and subsequently convert them into C++ header files, which can be used in plug-and-go style for inference.

To make this inference even faster, we can leverage the use of **GPU** for the inference. The goal of this project is to explore various GPU stacks, their integration with SOFIE, and implement inference using those tasks.

Project Milestones

1. Analyzing various **GPU stacks** (CUDA, ROCm, ALPAKA, etc.) in terms of their features such as **Parallel computing capabilities**, **Memory optimization**, **Execution efficiency**, etc. Determine their alignment with **SOFIE**.
2. Select the most suitable GPU stack by considering the points mentioned above. Implement **GPU** Inference for SOFIE in the chosen stack for various ONNX operators (Activation functions like ReLU, Tanh, Softmax, and operators such as RNN, LSTM, Convolution, Batchnorm, etc.)
3. **Benchmarking**: Evaluate the performance of the new **GPU** functionality by measuring **memory usage**, **execution time** on models with different operators. Compare the results with other frameworks (such as TensorFlow or PyTorch) across various **GPU** devices (NVIDIA, Intel, AMD) and against **CPU** performance.

Overview on GPU Programming

GPUs (Graphics Processing Units) are hardware designed for parallel execution, processing many tasks simultaneously. Unlike CPUs, which process one instruction at a time, GPUs use **SIMD (Single Instruction Multiple Data)** to handle multiple data pieces in parallel, speeding up tasks like vector operations. GPUs can outperform CPUs by 10x or 100x, reducing processing times from hours to minutes.

By integrating **GPUs** with **SOFIE**, inference execution can be significantly improved, offering faster and more efficient inference.

Explore GPU Stacks

There are multiple GPU programming frameworks available for parallel computing, such as CUDA, OpenCL, SYCL, etc. They can be categorized into 2 choices: 1. **Vendor Specific** and 2. **Cross-platform**.

Vendor-specific stacks can only run on hardware supported by the manufacturer. Example: **CUDA** is developed by **NVIDIA**; it only runs on NVIDIA GPUs. Similarly, **ROCm** is managed by **AMD** it only works on their native devices, such as AMD Radeon.

Cross-platform frameworks, such as **OpenCL**, **SYCL**, **ALPAKA**, and others, support multiple manufacturers and do not depend on a single set of GPU devices. Example:

OpenCL. OpenCL is a heterogeneous GPU stack that can run on almost all types of GPUs.

Integrating the cross-platform GPU stack with SOFIE will avoid building support for every individual GPU stack.

Project Implementation Approach

We can implement **GPU-based** inference for **SOFIE** using one of the **GPU stacks** mentioned above. There's already a SYCL implementation present for SOFIE. We can use it as inspiration to modify SOFIE's parsing and code generation algorithms for GPU-based inference.

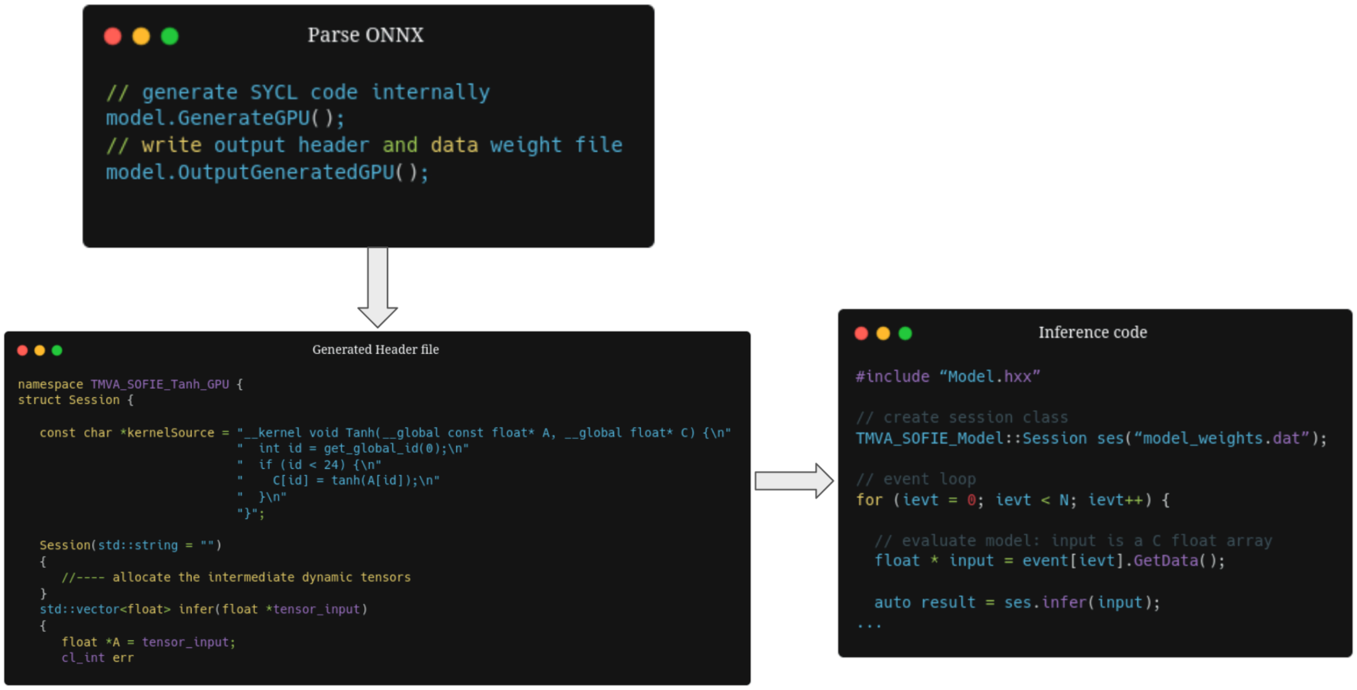
My understanding of the project, while building support for GPU inference in SOFIE, is as follows:

Modify files in SOFIE for GPU integration

1. `RModel.cxx`:

RModel.cxx is one of the crucial files responsible for the generation of the inference code. It generates the code by adding output strings to the **fGC**. Different functions handle tasks like outputting header files, memory allocation for intermediate tensors, generating session code, etc. All of these functions are used inside the primary function, which generates inference code for respective operators called **'Generate. '**

To implement inference on GPU, we can create another function similar to `Generate`, but with a different name, such as `GenerateGPU` or `GenerateGPUOpenCL`.

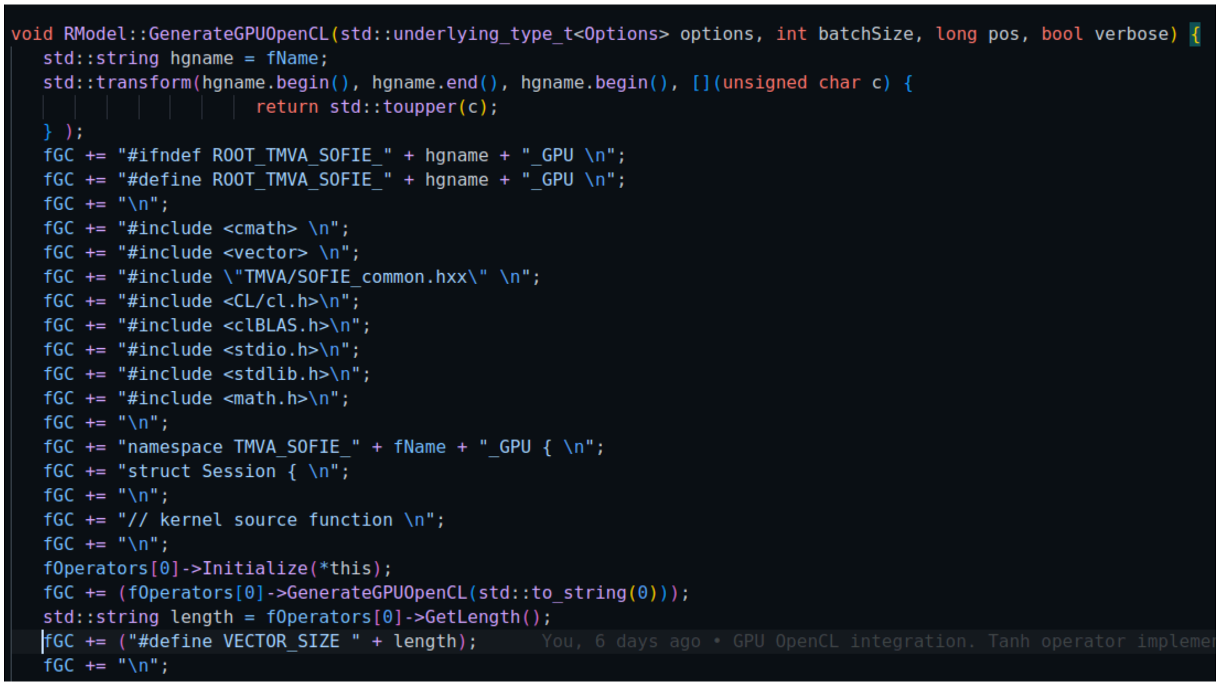


```
// generate SYCL code internally
model.GenerateGPU();
// write output header and data weight file
model.OutputGeneratedGPU();
```

```
namespace TMVA_SOFIE_Tanh_GPU {
struct Session {

   const char *kernelSource = "__kernel void Tanh(__global const float* A, __global float* C) {\n"
                              "  int id = get_global_id(0);\n"
                              "  if (id < 24) {\n"
                              "    C[id] = tanh(A[id]);\n"
                              "  }\n"
                              "}";

   Session(std::string = "")
   {
      //---- allocate the intermediate dynamic tensors
   }
   std::vector<float> infer(float *tensor_input)
   {
      float *A = tensor_input;
      cl_int err
```

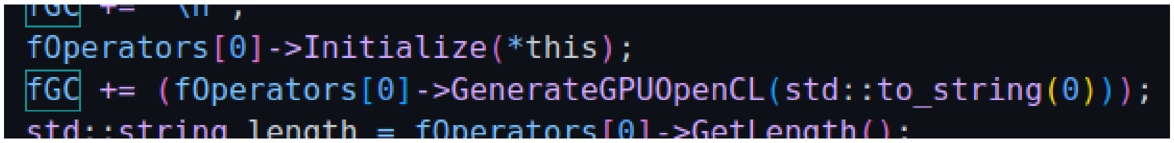
```
#include “Model.hxx”

// create session class
TMVA_SOFIE_Model::Session ses(“model_weights.dat”);

// event loop
for (ievt = 0; ievt < N; ievt++) {

  // evaluate model: input is a C float array
  float * input = event[ievt].GetData();

  auto result = ses.infer(input);
...
```

This function would serve the same purpose: generating optimized inference code, but for the **GPU**. It will use many of the functions used in the `Generate` function as they are only associated with generating header files, allocating memory for intermediate tensors, session code, etc., which are still essential for inference on GPU. While generating the header files, we need to include a few header files associated with the respective GPU stacks. Such as ‘`cuda.h`’ for CUDA, ‘`CL/cl.h`’ for OpenCL, etc.



```
void RModel::GenerateGPUOpenCL(std::underlying_type_t<Options> options, int batchSize, long pos, bool verbose) {
   std::string hgname = fName;
   std::transform(hgname.begin(), hgname.end(), hgname.begin(), [](unsigned char c) {
                     return std::toupper(c);
   } );
   fGC += "#ifndef ROOT_TMVA_SOFIE_" + hgname + "_GPU \n";
   fGC += "#define ROOT_TMVA_SOFIE_" + hgname + "_GPU \n";
   fGC += "\n";
   fGC += "#include <cmath> \n";
   fGC += "#include <vector> \n";
   fGC += "#include \"TMVA/SOFIE_common.hxx\" \n";
   fGC += "#include <CL/cl.h>\n";
   fGC += "#include <clBLAS.h>\n";
   fGC += "#include <stdio.h>\n";
   fGC += "#include <stdlib.h>\n";
   fGC += "#include <math.h>\n";
   fGC += "\n";
   fGC += "namespace TMVA_SOFIE_" + fName + "_GPU { \n";
   fGC += "struct Session { \n";
   fGC += "\n";
   fGC += "// kernel source function \n";
   fGC += "\n";
   fOperators[0]->Initialize(*this);
   fGC += (fOperators[0]->GenerateGPUOpenCL(std::to_string(0)));
   std::string length = fOperators[0]->GetLength();
   fGC += ("#define VECTOR_SIZE " + length);      You, 6 days ago • GPU OpenCL integration. Tanh operator implemen
   fGC += "\n";
```

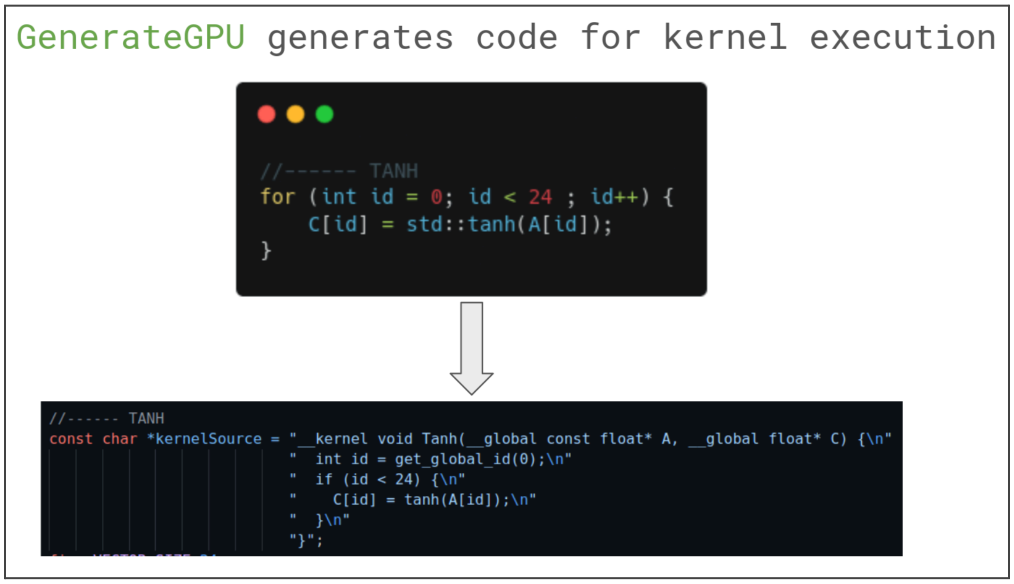
(Example of GenerateGPU function on OpenCL)



```
fGC += "\n";
fOperators[0]->Initialize(*this);
fGC += (fOperators[0]->GenerateGPUOpenCL(std::to_string(0)));
std::string length = fOperators[0]->GetLength();
```

(Getting operator specific code in GenerateGPUOpenCL function)

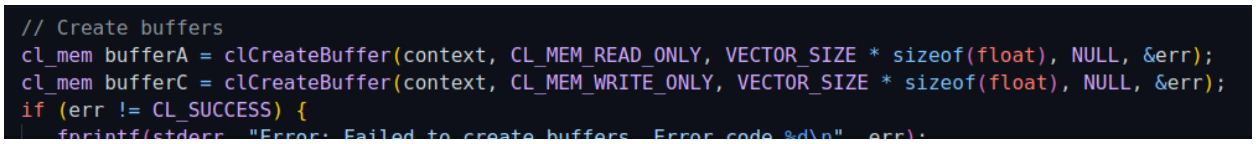
The inference function, however, won’t be the same as the Generate function. The steps to compile and execute the code on the GPU are quite different. We need to move the input tensors from **CPU** to **GPU**, execute the kernel function, move the calculated results back to CPU, and then return it. The `GenerateGPU` function will provide the template for all of this.


GenerateGPU generates code for kernel execution

```
//------ TANH
for (int id = 0; id < 24 ; id++) {
    C[id] = std::tanh(A[id]);
}
```

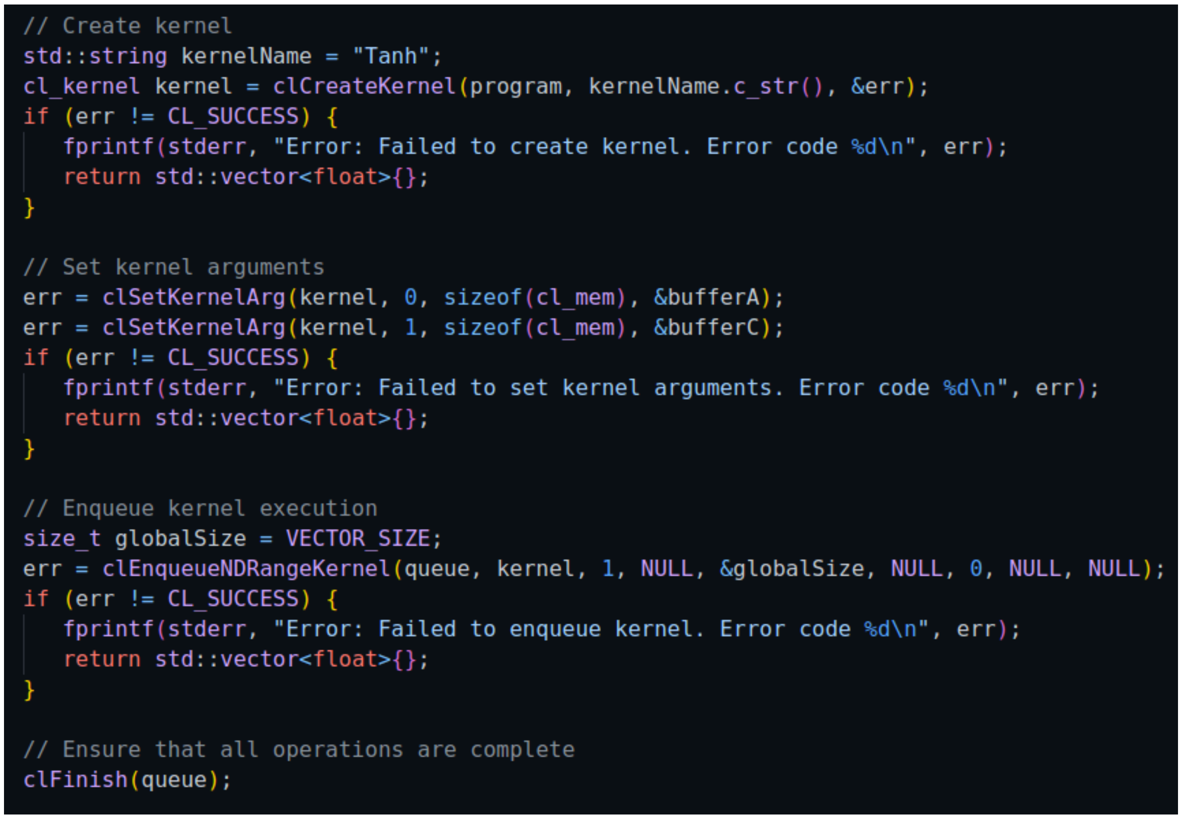
```
//------ TANH
const char *kernelSource = "__kernel void Tanh(__global const float* A, __global float* C) {\n"
                           "  int id = get_global_id(0);\n"
                           "  if (id < 24) {\n"
                           "    C[id] = tanh(A[id]);\n"
                           "  }\n"
                           "}";
```

(Example: Tanh kernel function)



```
// Create buffers
cl_mem bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_SIZE * sizeof(float), NULL, &err);
cl_mem bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, VECTOR_SIZE * sizeof(float), NULL, &err);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Error: Failed to create buffers. Error code %d\n", err);
```

(Creating buffers for input and output tensors in OpenCL)



```
// Create kernel
std::string kernelName = "Tanh";
cl_kernel kernel = clCreateKernel(program, kernelName.c_str(), &err);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Error: Failed to create kernel. Error code %d\n", err);
    return std::vector<float>{};
}

// Set kernel arguments
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufferA);
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufferC);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Error: Failed to set kernel arguments. Error code %d\n", err);
    return std::vector<float>{};
}

// Enqueue kernel execution
size_t globalSize = VECTOR_SIZE;
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, NULL, 0, NULL, NULL);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Error: Failed to enqueue kernel. Error code %d\n", err);
    return std::vector<float>{};
}

// Ensure that all operations are complete
clFinish(queue);
```

(Moving input and output tensors from CPU to GPU and performing kernel execution)

Now, for each operator, we can write its kernel function. But in SOFIE, during inference, the input tensors are flattened before passing them. So, writing kernel functions for complex operators such as Convolution, GEMM (General Matrix Multiplication), etc, would be quite challenging. So, instead, we can use the **BLAS** library for the respective GPU stack.

BLAS

The **BLAS** (Basic Linear Algebra Subprograms) library is a set of low-level routines that provide standard building blocks for performing basic vector and matrix operations such as:

- Vector addition and scalar multiplication
- Matrix-vector multiplication
- Matrix-matrix multiplication
- Solving systems of linear equations
- Computing dot products

BLAS is a library in C, and it alone cannot be used to compile and execute the code on the GPU. For this, each of the GPU stacks has its version of BLAS libraries, which are optimized and are meant to be worked for their GPU.
The libraries associated with each GPU stack are:

- **CUDA**: `cuBLAS`
- **ROCm**: `rocBLAS`
- **OneAPI**: `oneMKL`
- **OpenCL**: `clBLAS`
- **SYCL**: `portBLAS` and `oneMKL`

portBLAS: portBLAS is an open-source BLAS library written using **SYCL**, designed to work across various **GPUs**. It supports **Intel**, **NVIDIA**, and **AMD** GPUs. It's a **cross-platform** library.

oneMKL: oneMKL provides optimized math functions for **Intel hardware** and has **SYCL** interfaces that can target other hardware (like NVIDIA and AMD GPUs)**.**

Which one to choose for BLAS inside SYCL?

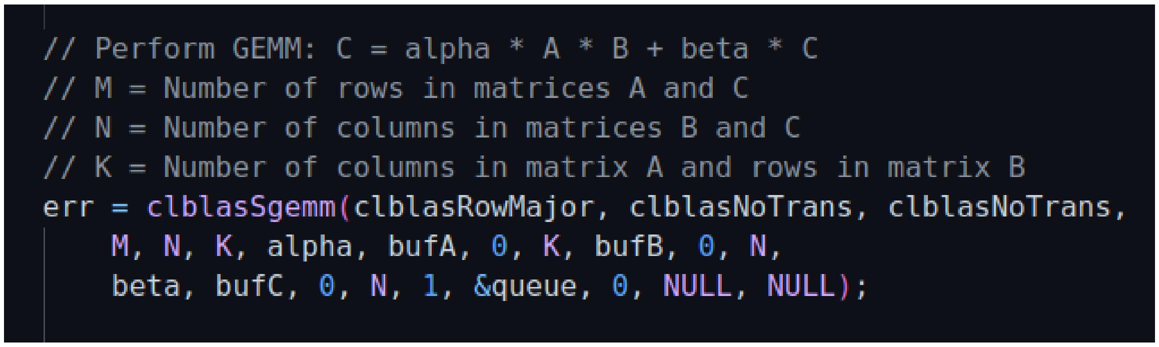
If we're primarily working with **Intel hardware** (CPUs and GPUs), **oneMKL** is the best choice because it's optimized for Intel architectures. It provides optimized implementations of BLAS and other mathematical operations.

If we need a **vendor-agnostic** BLAS library that can run on Intel, AMD, and NVIDIA GPUs, we should use **portBLAS**.

- **ALPAKA**: No dedicated BLAS, but can use libraries like `cuBLAS`, `rocBLAS`, or OpenCL-based BLAS through integration.

Using these libraries, we can perform complex operations such as Convolution, GEMM, Batch Normalization, Layer Normalization, etc.

Operators that require their kernel functions include **Activation functions** like ReLU, Leaky ReLU, Tanh, Sin, Cos, ELU, SELU, Softmax, etc.



```
// Perform GEMM: C = alpha * A * B + beta * C
// M = Number of rows in matrices A and C
// N = Number of columns in matrices B and C
// K = Number of columns in matrix A and rows in matrix B
err = clblasSgemm(clblasRowMajor, clblasNoTrans, clblasNoTrans,
    M, N, K, alpha, bufA, 0, K, bufB, 0, N,
    beta, bufC, 0, N, 1, &queue, 0, NULL, NULL);
```

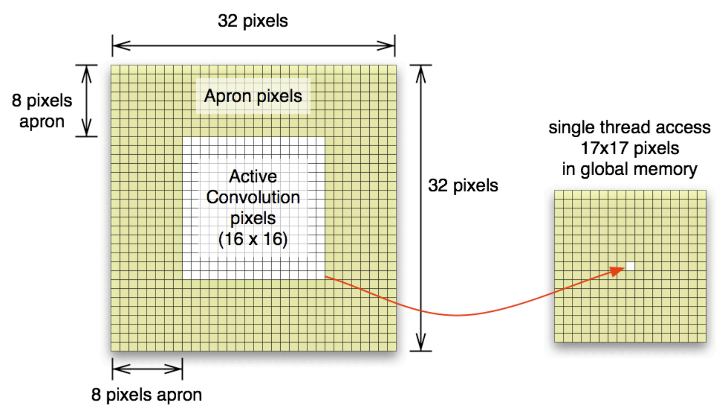
(Example: GEMM operator using clBLAS)

Operators that can be built using BLAS: Convolution, GEMM, Batch Normalization, Layer Normalization, etc

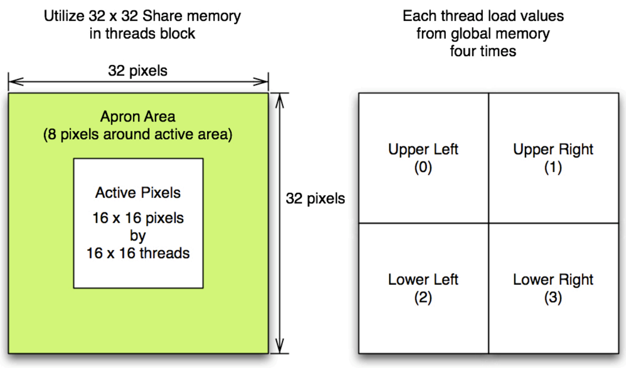
GPU kernel testing with custom libraries

There isn't a single correct way to write GPU kernels for operators. We can optimize naive approaches.

For example, there are multiple ways to write a Convolutional operator


32 pixels
8 pixels apron
Apron pixels
Active Convolution pixels (16 x 16)
32 pixels
single thread access 17x17 pixels in global memory
8 pixels apron

1. Naive approach (use global memory to send data to device and each thread accesses this to compute convolution kernel)


Utilize 32 x 32 Share memory in threads block
32 pixels
Apron Area (8 pixels around active area)
Active Pixels
16 x 16 pixels by 16 x 16 threads
32 pixels
Each thread load values from global memory four times
Upper Left (0)
Upper Right (1)
Lower Left (2)
Lower Right (3)

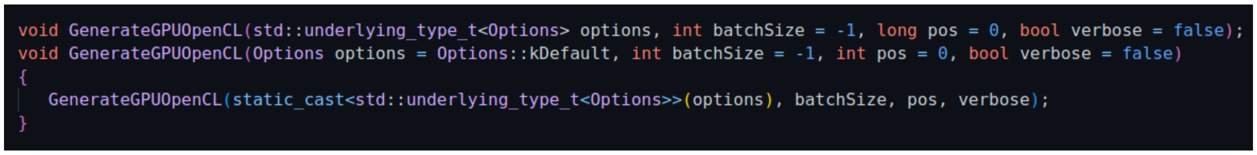
2.Optimized approach (each thread in block loads 4 values from global memory. So, total shared memory size is 4 times bigger)

Instead of solely relying on **BLAS** libraries, we can use a hybrid of functions from **BLAS** library and **GPU kernels**. This could take the form of specialized custom libraries specifically designed for that particular operator.

By testing the performance of these variants, we can determine the optimal approach for our specific operator.

In summary, the `GenerateGPU` function will fetch the respective inference code for each operator, create memory buffers, move the tensor from CPU to GPU, provide commands to GPU for the execution of the operation, move the calculated result back to CPU, and return the result. Another function, called `OutputGenerated,` writes the output header and data weight file. We need to create a similar function, say `OutputGeneratedGPU,` which would have similar functionality as `OutputGenerated,` but it would be for GPU.

2. `RModel.hxx`

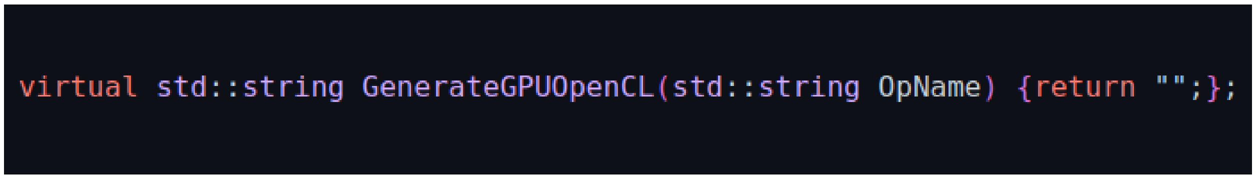


```
void GenerateGPUOpenCL(std::underlying_type_t<Options> options, int batchSize = -1, long pos = 0, bool verbose = false);
void GenerateGPUOpenCL(Options options = Options::kDefault, int batchSize = -1, int pos = 0, bool verbose = false)
{
   GenerateGPUOpenCL(static_cast<std::underlying_type_t<Options>>(options), batchSize, pos, verbose);
}
```

(Defining GenerateGPU function in RModel.hxx)

`RModel.hxx` contains the list of functions used in `RModel.cxx`. Here, all the functions are defined in public, protected, or private to generate the inference code. We need to define the function `GenerateGPU` in this class as public, so it can be used by objects of `RModel.cxx` while generating inference.

3. `ROperator.hxx`



```
virtual std::string GenerateGPUOpenCL(std::string OpName) {return "";};
```

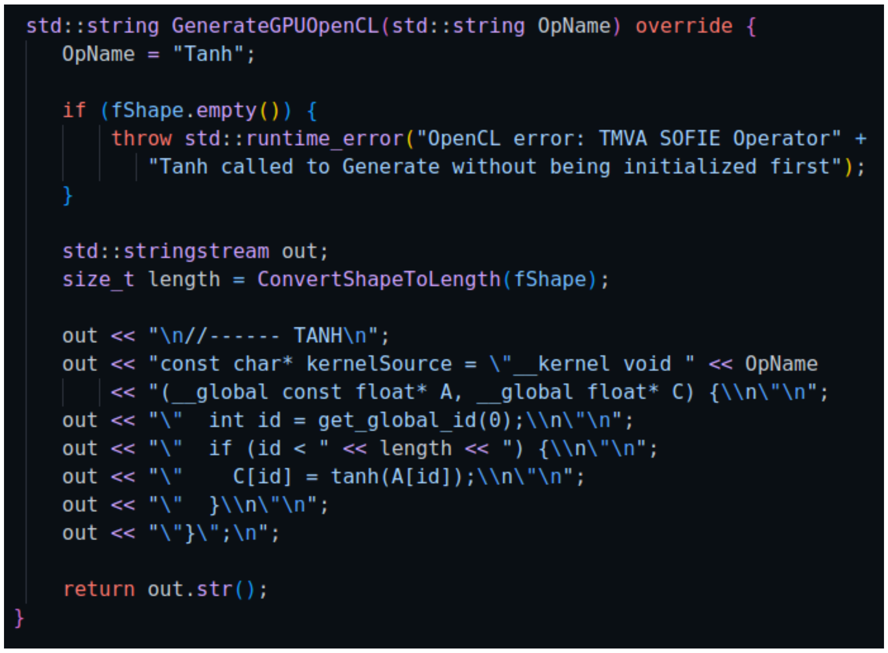
(Defining GenerateGPU function as virtual in ROperator.hxx class)

`ROperator.hxx` contains the declaration of all the functions used by the **ONXX** operators. We will be using the `GenerateGPU` function, which will return the operator-specific code (kernel function or BLAS code).

`GenerateGPU` function would be defined as a **virtual** function inside `ROperator.hxx,` and they would return an empty string. This is because a few operators might not need these functions, and while building ROOT, it might cause errors such as `GenerateGPU` not found.

`The GenerateGPU` function would be overwritten by the ROperators.

4. `ROperator_[op name].hxx`



```
std::string GenerateGPUOpenCL(std::string OpName) override {
   OpName = "Tanh";

   if (fShape.empty()) {
      throw std::runtime_error("OpenCL error: TMVA SOFIE Operator" +
         "Tanh called to Generate without being initialized first");
   }

   std::stringstream out;
   size_t length = ConvertShapeToLength(fShape);

   out << "\n//------ TANH\n";
   out << "const char* kernelSource = \"__kernel void " << OpName
      << "(__global const float* A, __global float* C) {\\n\"\n";
   out << "\"  int id = get_global_id(0);\\n\"\n";
   out << "\"  if (id < " << length << ") {\\n\"\n";
   out << "\"    C[id] = tanh(A[id]);\\n\"\n";
   out << "\"  }\\n\"\n";
   out << "\"}\";\n";

   return out.str();
}
```

(Example: GenerateGPU function for Tanh operator)

All of the **ONNX** operators will have a `GenerateGPU` function, similar to the `Generate` function, which will override the `GenerateGPU` function in the `ROperator.hxx`.

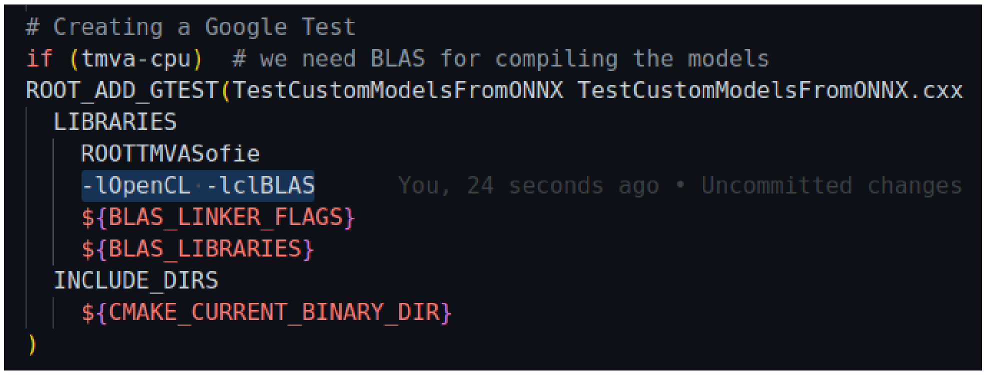
The GenerateGPU function would be unique for each operator. Few will return a kernel function with their defined own operation, and few will involve using BLAS functions.

5. `CMakeLists.txt`

For each of the GPU stacks, to execute the code on GPU, it is usually written in `C/C++`. However, to compile it with `g++`, we need to specify a certain flag while compiling. This flag varies with GPU stack. For example, **OpenCL** uses `-lOpenCL`. Also, while using respective **BLAS** libraries, we need to specify the BLAS flag as well. For example, **OpenCL** uses `-clBLAS`

Required Flags for different GPU stacks while compiling:

- **CUDA**: `-lcuda -lcudart -lcublas`
- **ROCm**: `-lamdhip64 -lrocblas`
- **OneAPI**: `-lSYCL -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5` (or `-lmkl_rt` for runtime linking)
- **OpenCL**: `-lOpenCL -lclBLAS`
- **SYCL**: `-fsycl -lSYCL -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5`
- **ALPAKA** (using CUDA as backend): `-lAlpaka_cuda -lcublas`
- **ALPAKA** (using OpenMP as backend): `-lAlpaka_openmp -lopenblas`



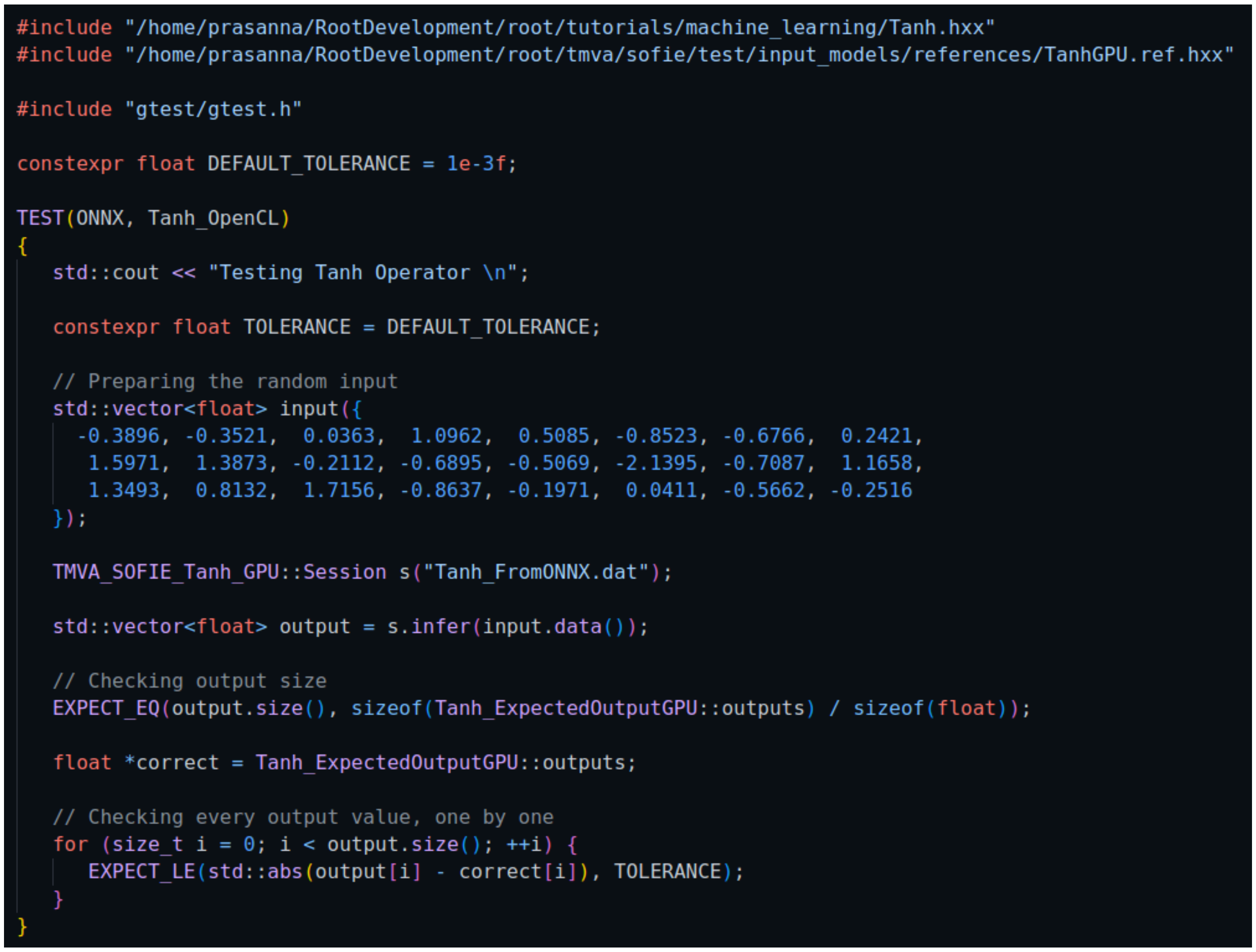
```
# Creating a Google Test
if (tmva-cpu)  # we need BLAS for compiling the models
ROOT_ADD_GTEST(TestCustomModelsFromONNX TestCustomModelsFromONNX.cxx
  LIBRARIES
    ROOTTMVASofie
    -lOpenCL -lclBLAS      You, 24 seconds ago • Uncommitted changes
    ${BLAS_LINKER_FLAGS}
    ${BLAS_LIBRARIES}
  INCLUDE_DIRS
    ${CMAKE_CURRENT_BINARY_DIR}
)
```

(Example: Adding -lOpenCL flag and -clBLAS flag in CMakeLists.txt for testing)

By modifying these files and adjusting `CMakeLists.txt` according to the chosen GPU stack, we can implement GPU-based functionality in SOFIE.

Write tests for Operators implemented on GPU

After implementing GPU inference support, we should test each operator to validate its functionality. SOFIE already has tests for this written under `tmva/sofie/tests` directory in the file `TestCustomModelsFromONNX.cxx`, which can be used to write similar tests for GPU implementation.



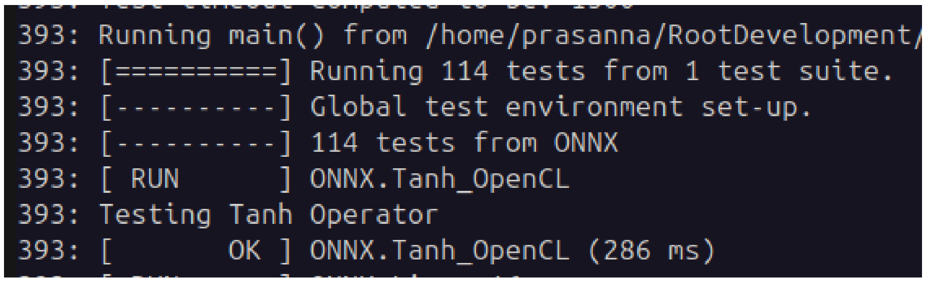
```cpp
#include "/home/prasanna/RootDevelopment/root/tutorials/machine_learning/Tanh.hxx"
#include "/home/prasanna/RootDevelopment/root/tmva/sofie/test/input_models/references/TanhGPU.ref.hxx"

#include "gtest/gtest.h"

constexpr float DEFAULT_TOLERANCE = 1e-3f;

TEST(ONNX, Tanh_OpenCL)
{
   std::cout << "Testing Tanh Operator \n";

   constexpr float TOLERANCE = DEFAULT_TOLERANCE;

   // Preparing the random input
   std::vector<float> input({
     -0.3896, -0.3521,  0.0363,  1.0962,  0.5085, -0.8523, -0.6766,  0.2421,
      1.5971,  1.3873, -0.2112, -0.6895, -0.5069, -2.1395, -0.7087,  1.1658,
      1.3493,  0.8132,  1.7156, -0.8637, -0.1971,  0.0411, -0.5662, -0.2516
   });

   TMVA_SOFIE_Tanh_GPU::Session s("Tanh_FromONNX.dat");

   std::vector<float> output = s.infer(input.data());

   // Checking output size
   EXPECT_EQ(output.size(), sizeof(Tanh_ExpectedOutputGPU::outputs) / sizeof(float));

   float *correct = Tanh_ExpectedOutputGPU::outputs;

   // Checking every output value, one by one
   for (size_t i = 0; i < output.size(); ++i) {
      EXPECT_LE(std::abs(output[i] - correct[i]), TOLERANCE);
   }
}
```

(Example: Unit test for Tanh operator implemented on OpenCL)

Steps to write unit tests:

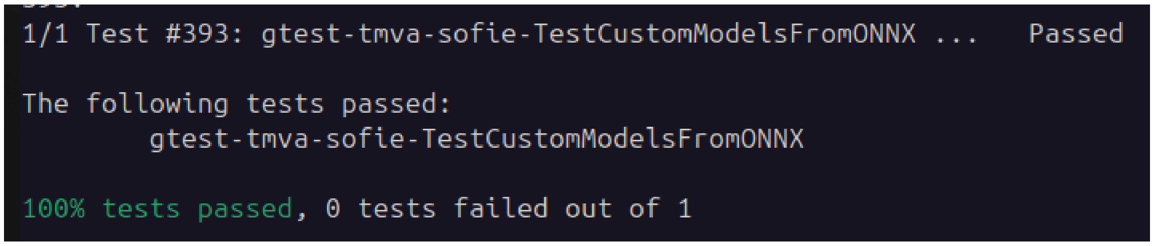
1. Parse the models/operators from `tmva/sofie/tests/input_models` to `RModelParser_ONNX` and generate its equivalent code, which can be run on GPU using the `GenerateGPU` function. This will generate a header file with the same name as the input model with the extension `.hxx`.
2. Create a new file with the name `TestCustomModelsFromONNXGPU.hxx` and include the header file in it.

3. Each of these models has a `.ref.hxx` file, which contains an expected value for the answer when passed through the operator. Include these files as well in the test file.
4. Include `gtest/gtest.h` to perform tests on the operators.
5. Write tests with function `TEST` & input parameters as `(ONNX, [model_name])` and define flattened input tensor. Create a session for that particular operator and pass the input tensor to the infer function.
6. Validate the results with `[operator_name].ref.hxx`



```
393: Running main() from /home/prasanna/RootDevelopment/
393: [==========] Running 114 tests from 1 test suite.
393: [----------] Global test environment set-up.
393: [----------] 114 tests from ONNX
393: [ RUN      ] ONNX.Tanh_OpenCL
393: Testing Tanh Operator
393: [       OK ] ONNX.Tanh_OpenCL (286 ms)
```

(Running tests for Tanh operator on OpenCL)



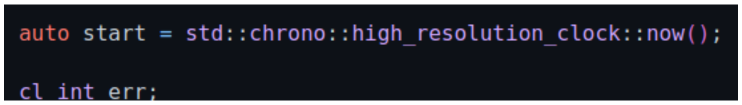
```
1/1 Test #393: gtest-tmva-sofie-TestCustomModelsFromONNX ...   Passed

The following tests passed:
        gtest-tmva-sofie-TestCustomModelsFromONNX

100% tests passed, 0 tests failed out of 1
```

(Expected result for the tests)

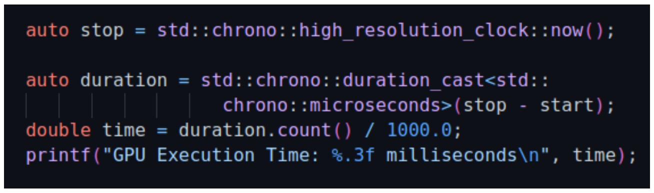
Benchmarking the developed GPU support for ML inference

After implementing GPU support for ML inference, we should benchmark its performance against **CPU-based inference** and across various GPU stacks. To conduct the benchmarking, we would test different GPU stacks (OpenCL, SYCL, CUDA, etc.) on various **GPUs** (Intel, NVIDIA, AMD), along with the **ONNX runtime**. The benchmarking will involve testing multiple **ML models**, including **ResNet**, **Inception**, **VGG**, and sequence-to-sequence models like **RNN**, **LSTM**, and **Transformer**, on the GPU implementation. This includes a variety of GPU stacks, reduces GPU performance dependency, and tests it on different ML models.

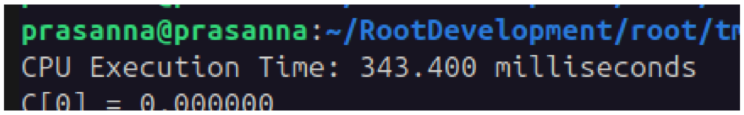
This would show which GPU stack performed better than others, and we can focus on the best, which suits SOFIE.



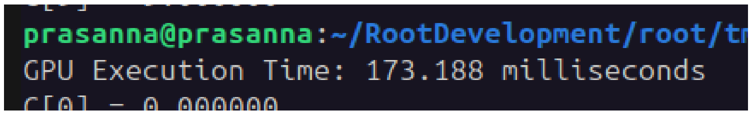
```
auto start = std::chrono::high_resolution_clock::now();

cl_int err;
```



```
auto stop = std::chrono::high_resolution_clock::now();

auto duration = std::chrono::duration_cast<std::
                chrono::microseconds>(stop - start);
double time = duration.count() / 1000.0;
printf("GPU Execution Time: %.3f milliseconds\n", time);
```



```
prasanna@prasanna:~/RootDevelopment/root/tr
CPU Execution Time: 343.400 milliseconds
C[0] = 0.000000
```

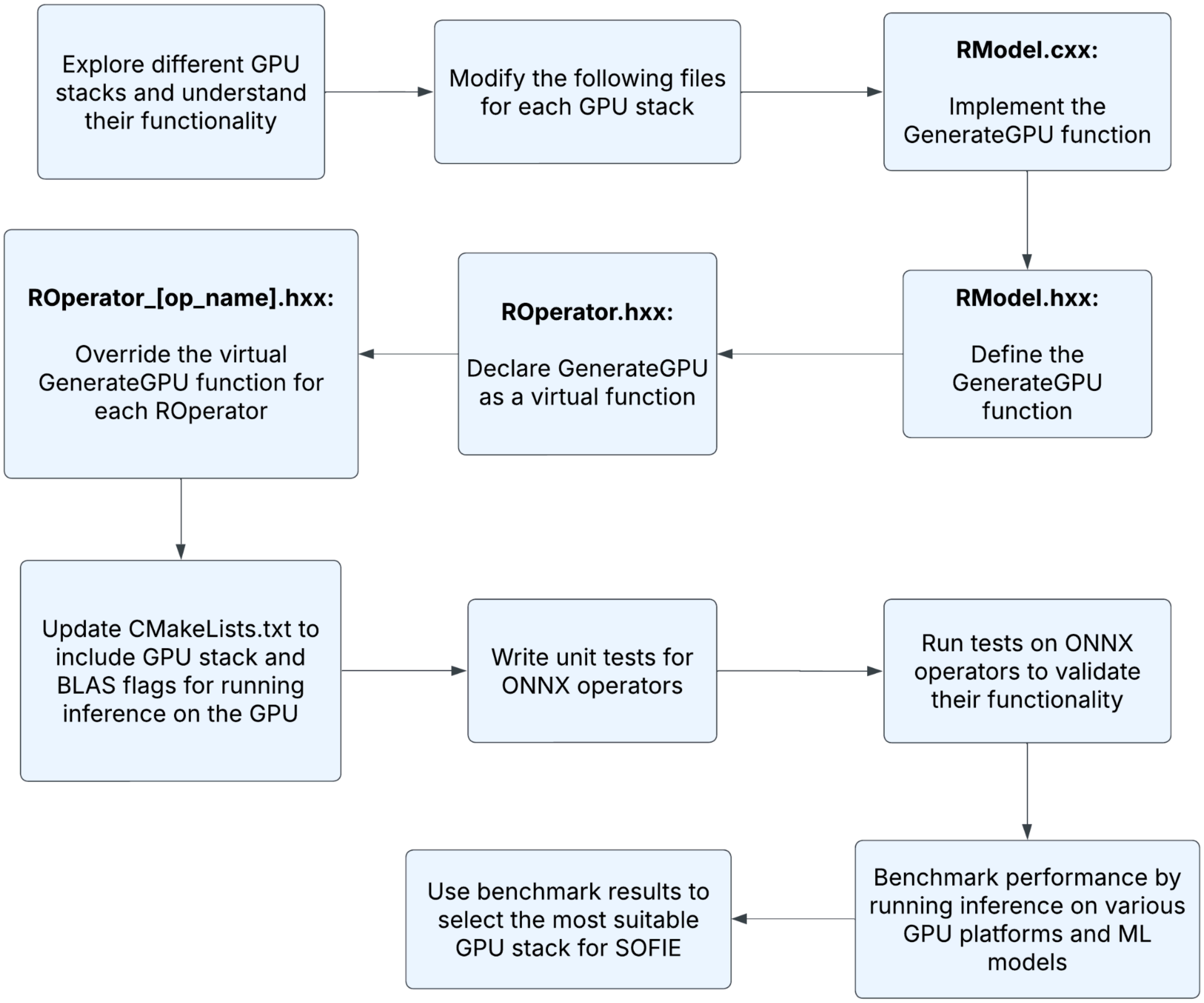


```
prasanna@prasanna:~/RootDevelopment/root/tr
GPU Execution Time: 173.188 milliseconds
C[0] = 0.000000
```

(Example: Measure execution time on GPU using OpenCL and on CPU)

After benchmarking and performance evaluation, we will select the most appropriate GPU stack for SOFIE. The ideal choice will be compatible across a wide range of devices, support heterogeneous platforms (CPU, GPU, FPGA), offer readily available documentation, and have an active community for support.

Steps for GPU Integration



Explore different GPU stacks and understand their functionality
Modify the following files for each GPU stack
RModel.cxx: Implement the GenerateGPU function
RModel.hxx: Define the GenerateGPU function
ROperator.hxx: Declare GenerateGPU as a virtual function
ROperator_[op_name].hxx: Override the virtual GenerateGPU function for each ROperator
Update CMakeLists.txt to include GPU stack and BLAS flags for running inference on the GPU
Write unit tests for ONNX operators
Run tests on ONNX operators to validate their functionality
Benchmark performance by running inference on various GPU platforms and ML models
Use benchmark results to select the most suitable GPU stack for SOFIE

Which GPU stack would work best with SOFIE?

In my opinion, **SYCL** and **ALPAKA** would align best with **SOFIE** after considering all the GPU stack choices.

Why would SYCL and ALPAKA be the best choices for GPU support?

While analyzing various GPU stacks and determining their alignment with SOFIE, we considered a variety of options, including both **vendor-specific** and **cross-platform** stacks. However, heterogeneous platforms such as **OpenCL**, **SYCL**, and **ALPAKA** offer broader compatibility than vendor-specific ones and eliminate building GPU support for every other GPU stack. So, it's best to consider a heterogeneous platform for building GPU support for SOFIE. That eliminates GPU stacks like **CUDA** and **ROCm**, ultimately leaving OpenCL, SYCL, and ALPAKA.

SYCL is a higher-level abstraction built on top of OpenCL to enable more modern C++ style programming. It also reduces code redundancy since OpenCL requires manual management of memory and kernel dispatching.

Due to the elimination of these GPU stacks, we turn to **SYCL** and **ALPAKA**, both of which offer the **flexibility and cross-platform compatibility** needed for this project.

SYCL Implementation in SOFIE

After comparing SYCL and ALPAKA in terms of their alignment with SOFIE and their performance, we can choose the most suitable GPU stack for the project. Since SOFIE already has a SYCL implementation, I believe continuing with **SYCL** is the most practical choice for this project.

Missing operators in SYCL implementation

The current SYCL implementation is almost complete, but it is missing 2 key operators: **GRU** and **LSTM**. To complete the implementation, we need to add these operators in SYCL.

Throughout this project, I will also explore **ALPAKA** and other GPU stacks, evaluating their compatibility with SOFIE. However, my primary focus will be on extending the **SYCL** implementation by implementing the **GRU** and **LSTM** operators.

Project Timelines

Timeline	Tasks
Up until May 8: Proposal Acceptance or Rejection	➔ Become familiar with the codebase. ➔ Understand the basics of GPU programming, which is crucial for building GPU support for SOFIE. ➔ Discuss any doubts related to implementation with your mentor. ➔ Establish a proper project structure to ensure smooth development. **Deliverables:** Gain solid knowledge of GPU programming. Have a clear understanding of the project's requirements and objectives, with no remaining doubts.
May 8 - June 1: Community Bonding Period	➔ Set up environments for all the GPU stacks. ➔ Begin gaining knowledge about the various GPU stacks, their similarities and differences. ➔ Take inspiration from the SYCL implementation for further development. **Deliverables:** Gain in-depth knowledge of all the GPU stacks and their functionalities.
Week 1, 2: Official Coding Begins	➔ Study different GPU stacks (CUDA, OpenCL, SYCL, ALPAKA, etc.) in terms of their features such as Parallel computing capabilities, hardware acceleration, memory management, performance, and scalability. ➔ Discuss with mentors about their alignment with SOFIE. **Deliverables:** Finalize the most suitable GPU stack for SOFIE by exploring them.

Week 3	➔ Research about chosen GPU stack and its BLAS library. ➔ Examine which files needs to be changed to integrate GPU support within SOFIE. **Deliverables:** Get clear understanding of implementation of GPU support for selected stack. Have in-depth knowledge about its related BLAS libraries.
Week 4, 5	➔ Explore necessary files in SOFIE which convert Intermediate Representation (RModel) to C++ header file for the generation of inference code. ➔ Look into the existing SYCL implementation in SOFIE as a reference to implement GPU support for the chosen stack. **Deliverables:** Create `GenerateGPU` function for inference code generation. Modify required files in SOFIE to support inference on GPU.
Week 6, 7	➔ Select 1-2 ONNX operators to start with (e.g., basic activation functions or simple operations) and implement the `GenerateGPU` function for those operators in the corresponding `ROperator_[op_name].hxx` file to generate GPU kernel code. ➔ Replicate existing unit tests for the chosen operators and verify the correctness of the GPU implementation on different types of inputs (e.g., long inputs, varying data types, batch sizes, etc.). ➔ Explore benchmarking methods to assess the performance improvements brought by GPU support for these operators. ➔ Prepare documentation of progress made so far for mid-term evaluation **Deliverables:** A fully functioning GenerateGPU function that generates kernel code for 1-2 ONNX operators. Verified unit tests to ensure GPU functionality for the implemented operators. Conduct performance benchmarking on the built operators.

Week 8, 9	➔ Implement GPU Support for More ONNX Operators, starting with all activation functions (e.g., ReLU, Sigmoid, Tanh, etc.). ➔ Replicate Unit Tests for the New Operators. ➔ Perform Benchmarking on the Generated Kernel Functions. **Deliverables:** Implementation and Verification of GPU Support for Activation Functions. Benchmarking results for the newly implemented GPU operators.
Week 10, 11	➔ Implement Complex ONNX Operators (e.g., RNN, Convolution) ➔ Adapt existing unit tests for the chosen operators ➔ Conduct detailed benchmarking to evaluate the performance of GPU-accelerated versions of the complex operators. ➔ Ensure all mentor requirements are fulfilled. If additional operators are required, extend the project deadline to implement support for them. **Deliverables:** Create and perform unit tests on implemented operators to validate generate inference code for each operator. Conduct performance benchmarking on the built operators.
Week 12	➔ Focus on documentation and prepare a blog post. ➔ Summarize all work in a blog post and submit it for evaluation.

Why me and my Motivation?

I recently started exploring the domain of GPU programming by writing kernels for OpenCL, and it has truly intrigued me. I always wanted to learn about Low-level programming, and GPU programming has given me a head start on my journey.

I found SOFIE's method of converting trained ML models into C++ header files for fast inference to be particularly impressive. My goal is to enhance this process further by integrating GPU support to make the inference even faster.

I have always aspired to work for CERN and would be highly grateful to receive the opportunity to contribute as a Google Summer of Code student developer. This is a great opportunity and a head start for achieving success in my life.

References

OpenCL:

Official Documentation by Khronos Group

Beginner-friendly guide to OpenCL

Overview of SOFIE with SYCL

Presentation

SYCL

SYCL implementation of SOFIE

Accelerating Machine Learning Inference on GPUs with SYCL