PKMLC

Colin McGann

Team Number 11

April 28, 2024

Final Project Report for
HCI 575: Computational Perception

Iowa State University
Spring 2024

ABSTRACT

This project focuses on discovering possible speed improvements to machine learning algorithms that can be made by supplying neural network parameters during compile time. This will be done by expanding loops at compile time and reducing the amount of GPU processes that must be executed to complete a training cycle. These optimizations are generally impossible due to network parameters being supplied during the program's runtime. This problem is an implicit issue with supplying the neural network tools as a binary library. This is why my tool will allow the generation of source code tailored explicitly to the neural network requested to be generated. This tool will use configuration files to allow easy modification of neural networks. A tool similar to this is not currently available, so I believe there is a space for my project in the machine learning community.

1 Introduction

With neural networks becoming more prevalent in modern technology, the need for faster training systems has become more desired. A large amount of network training has begun being done on general-purpose graphics processing units (GPGPUs) [2]. The most popular manufacturer of GPGPUs currently is Nvidia [3]; these will be the GPGPUs this paper will focus on.

Nvidia uses a SIMD parallel control mechanism for their GPGPUs. This means that each Streaming Multiprocessor (SM) may only execute a single instruction for all threads running on the SM [Figure 1] [1, ch. 34.1]. This means the processor can suffer significant speed losses from divergent branches as it is not feasible to have a branch prediction system.

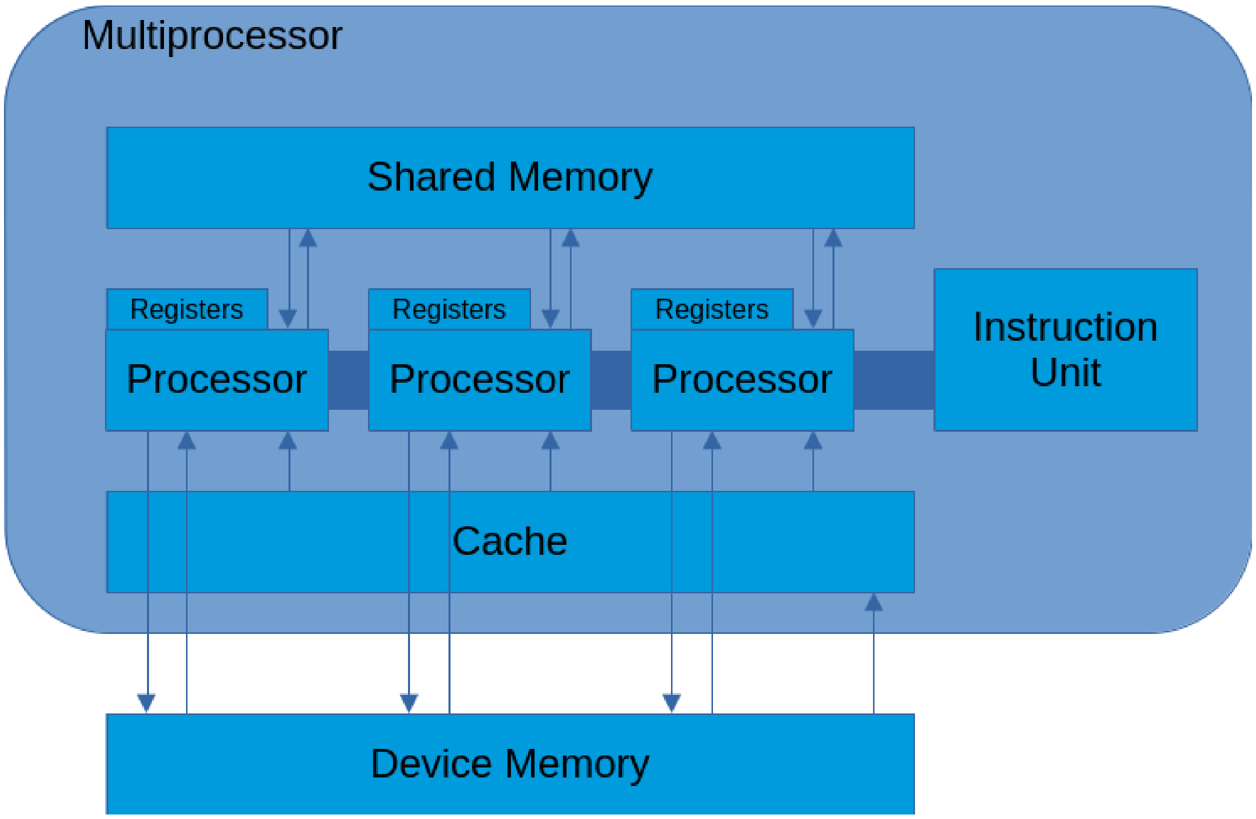
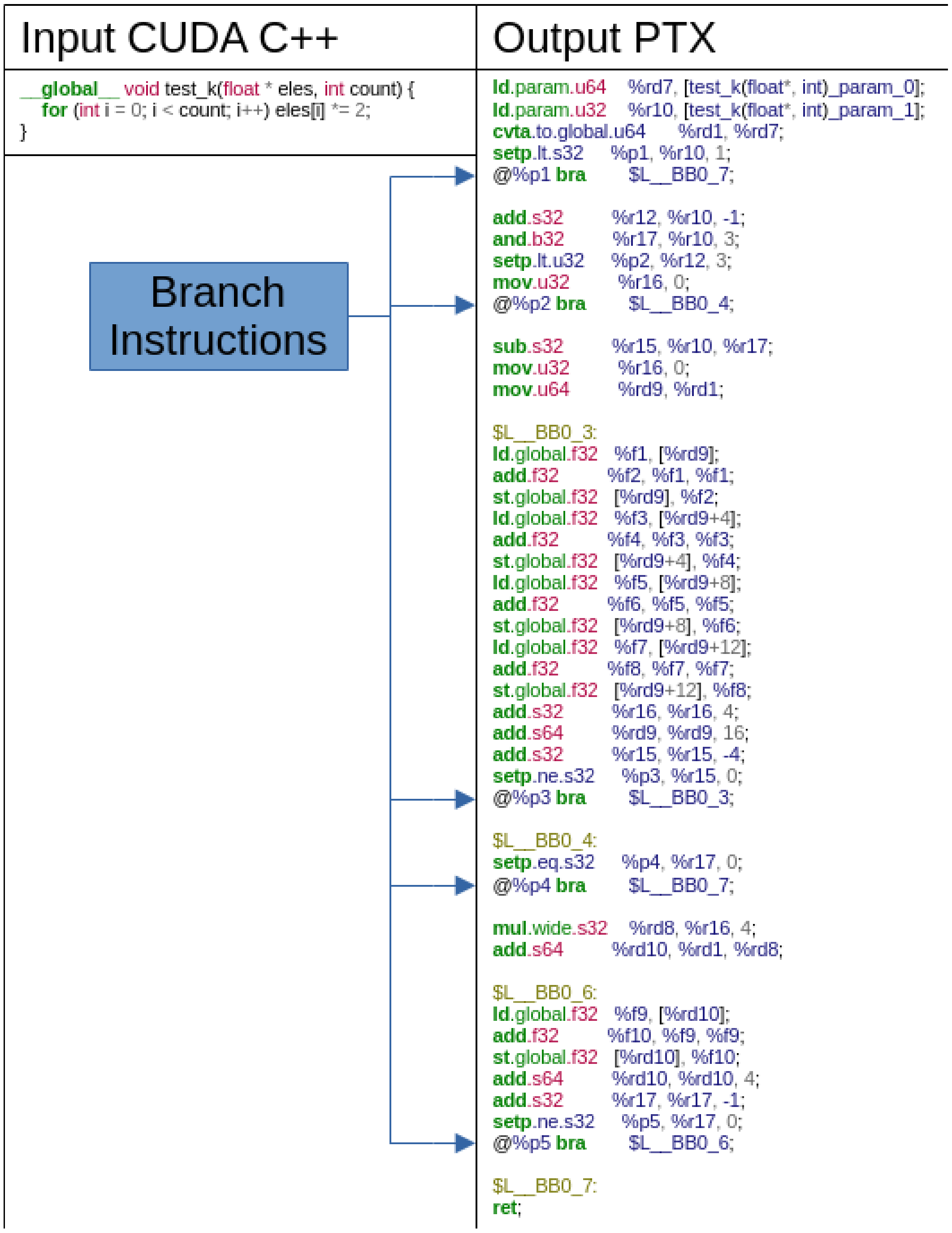


Figure 1: High-level diagram of a GPGPU, each processor must execute the same instruction at the same time.

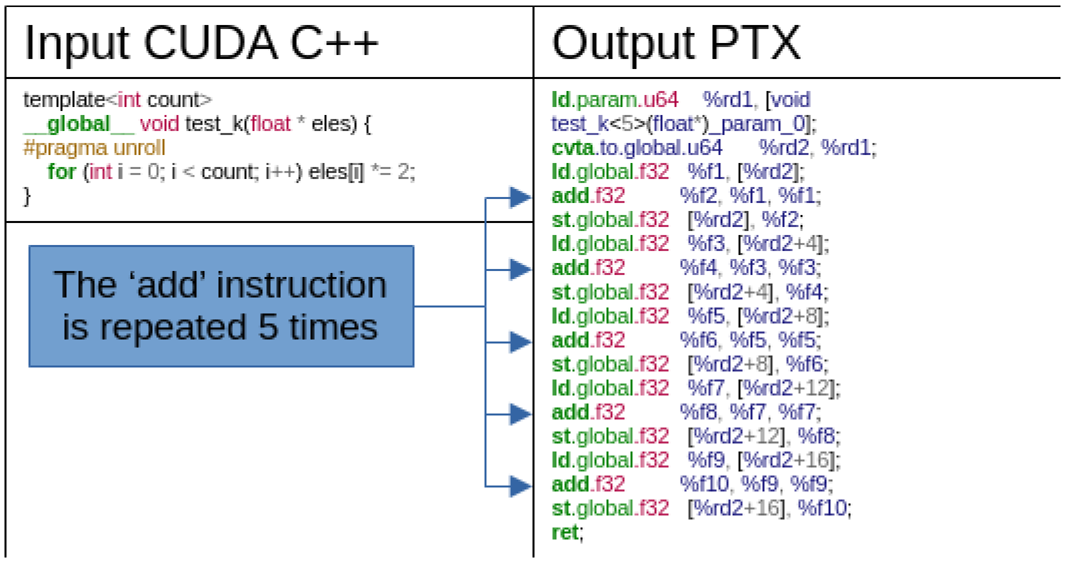
[Appendix A] A way to reduce the amount of branch-related issues is to remove the branch instructions altogether. This can be done by evaluating all branches at compile time. When a conditional is known a compile time the false path may be removed along with the branch instruction before the code is executed. This is applicable to looping code segments as well. If a loop has a number of iterations that can be determined at compile time, it may be expanded [Figure 2] [Figure 3]. Expanding a loop will copy the contents equal to the number of iterations the loop is expected to execute.



```
__global__ void test_k(float * eles, int count) {
    for (int i = 0; i < count; i++) eles[i] *= 2;
}
```

```
ld.param.u64    %rd7, [test_k(float*, int)_param_0];
ld.param.u32    %r10, [test_k(float*, int)_param_1];
cvta.to.global.u64      %rd1, %rd7;
setp.lt.s32     %p1, %r10, 1;
@%p1 bra        $L__BB0_7;

add.s32         %r12, %r10, -1;
and.b32         %r17, %r10, 3;
setp.lt.u32     %p2, %r12, 3;
mov.u32          %r16, 0;
@%p2 bra        $L__BB0_4;

sub.s32         %r15, %r10, %r17;
mov.u32          %r16, 0;
mov.u64          %rd9, %rd1;

$L__BB0_3:
ld.global.f32   %f1, [%rd9];
add.f32         %f2, %f1, %f1;
st.global.f32   [%rd9], %f2;
ld.global.f32   %f3, [%rd9+4];
add.f32         %f4, %f3, %f3;
st.global.f32   [%rd9+4], %f4;
ld.global.f32   %f5, [%rd9+8];
add.f32         %f6, %f5, %f5;
st.global.f32   [%rd9+8], %f6;
ld.global.f32   %f7, [%rd9+12];
add.f32         %f8, %f7, %f7;
st.global.f32   [%rd9+12], %f8;
add.s32         %r16, %r16, 4;
add.s64         %rd9, %rd9, 16;
add.s32         %r15, %r15, -4;
setp.ne.s32     %p3, %r15, 0;
@%p3 bra        $L__BB0_3;

$L__BB0_4:
setp.eq.s32     %p4, %r17, 0;
@%p4 bra        $L__BB0_7;

mul.wide.s32    %rd8, %r16, 4;
add.s64         %rd10, %rd1, %rd8;

$L__BB0_6:
ld.global.f32   %f9, [%rd10];
add.f32         %f10, %f9, %f9;
st.global.f32   [%rd10], %f10;
add.s64         %rd10, %rd10, 4;
add.s32         %r17, %r17, -1;
setp.ne.s32     %p5, %r17, 0;
@%p5 bra        $L__BB0_6;

$L__BB0_7:
ret;
```

Figure 2: PTX generated by code with a runtime loop.



```
template<int count>
__global__ void test_k(float * eles) {
#pragma unroll
    for (int i = 0; i < count; i++) eles[i] *= 2;
}
```

```
ld.param.u64    %rd1, [void
test_k<5>(float*)_param_0];
cvta.to.global.u64     %rd2, %rd1;
ld.global.f32   %f1, [%rd2];
add.f32        %f2, %f1, %f1;
st.global.f32   [%rd2], %f2;
ld.global.f32   %f3, [%rd2+4];
add.f32        %f4, %f3, %f3;
st.global.f32   [%rd2+4], %f4;
ld.global.f32   %f5, [%rd2+8];
add.f32        %f6, %f5, %f5;
st.global.f32   [%rd2+8], %f6;
ld.global.f32   %f7, [%rd2+12];
add.f32        %f8, %f7, %f7;
st.global.f32   [%rd2+12], %f8;
ld.global.f32   %f9, [%rd2+16];
add.f32        %f10, %f9, %f9;
st.global.f32   [%rd2+16], %f10;
ret;
```

Figure 3: PTX generated by code with a iteration count known at compile time. This example is instantiated with count set to 5

GPGPUs may also have somewhat significant performance draws from starting a GPU program [4, Table 1]. This may be especially prevalent for neural networks where the program may leave and reenter the GPU multiple times before completing a single forward and backward pass [Figure 4]. I will solve this by combining all forward and backward pass code segments into a single GPU program launch during compile time.

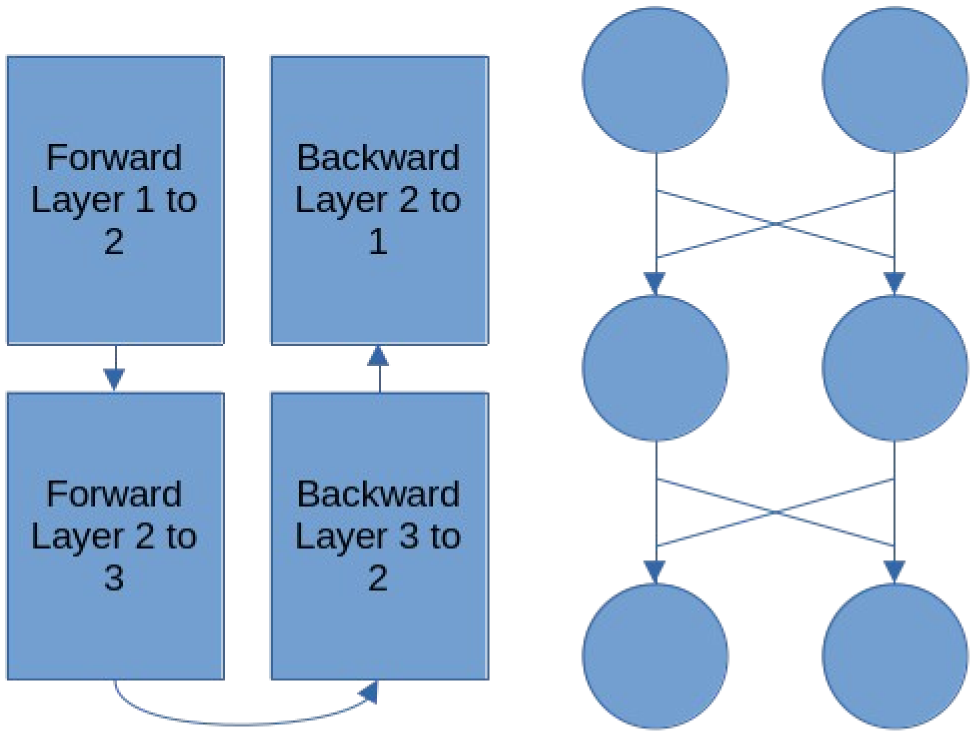


Figure 4: A diagram representing a forward and backward pass through a simple neural network. Each block represents a separate GPU program launch.

Expanding conditional loops will allow the complete removal of branch conditions. My project plans to apply this concept to neural network learning algorithms on a larger scale. This will be done by creating a program to convert a set of configuration files into a CUDA C++ source and header file. This will then be able to be compiled into a GPU-capable program and used to train neural networks.

2 Related Work

I have not been able to find any projects available that exactly replicate what my project seeks to perform. I believe this is because most users requiring the most optimal neural network algorithms may prefer to write their own programs from scratch, where supplying compile-time network parameters may be done directly for the specific network implementation. I still believe that there is a place for my tool in the world. My tool will simplify the process of implementing the neural network and allow for the customization of network parameters easily, which may not be possible for custom-written software.

I have taken some inspiration from [5] for the details of implementing a multi-layer perceptron(MLP) neural network on a Nvidia GPGPU. [5] describes a neural network implementation on a GPGPU. The primary difference between this work and mine is the use of compile-time network parameters. This should allow for performance improvements through branch condition removal and reduction of GPGPU program launches. The implementation of the MLP shown in [5] will be mostly identical to the algorithm used in my project.

I have used [6] for inspiration on the backpropagation learning algorithm implementation in my project. This is a critical portion of my project as I am interested in improving network training speeds using backpropagation. The primary difference between my project and [6] will be the availability of network parameters at compile time and the ability to unroll conditional loops at compile time. The general implementation of the algorithms will be similar, however.

3 Experimental Platform

All experimental results will be obtained from a server with:

- 12th Gen Intel(R) Core(TM) i7-12700K x20
- NVIDIA GeForce GTX 1050 Ti

The GPGPU used in my experimental setup is a somewhat old processor that was introduced in 2016. According to [7], the processor contains a total of 768 CUDA cores. The specifications are somewhat low compared to more modern consumer and data center GPGPUs. I believe that this experimental setup is still valid, however, as all tests will be run on this processor and, therefore, will be comparatively equal.

I will be generating the code for my neural networks using configuration files. My tool will pars these to produce CUDA C++ source and header files specialized for the network to be produced. These will allow easy implementation of high-performance neural networks with minimal effort.
The network generation tool will take a configuration file in JSON format. I chose this format because of its easy-to-understand nature. The tool will then extract the network parameters from this file and begin generating output files [Figure 5].

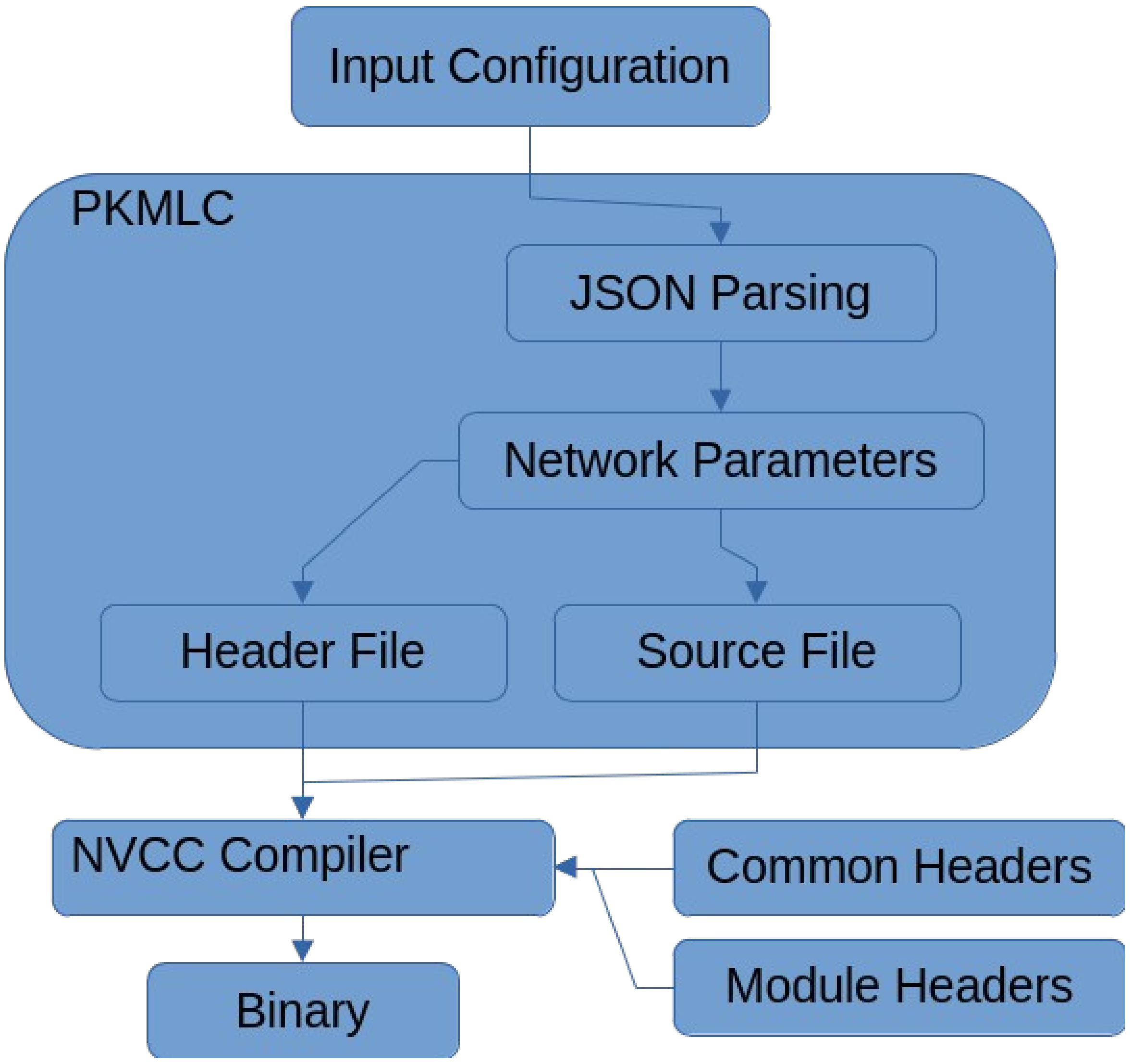


Figure 5: Flowchart showing the flow of data through the network generator tool.

The parameters provided to the network generator must contain data outlining the resulting C++ class name, the floating-point type, the required modules, the GPU grid and block dimension, and the neural network layers.

Modules will be specified by a relative path inside of the modules section [Appendix A]. These relative paths will point to a directory containing a file titled ‘manifest.json’ and one or more CUDA C++ header files. These modules must provide a C++ class with a set of functions for passing data forward and backward through the network [Appendix B].

Each module manifest may specify that the module is gated. This will cause the layer to force thread synchronization before the threads enter the kernel [Figure 6]. The gated attribute will be required to be set to true for any layer that plans to take data from more than just the neuron immediately preceding the activating neurons. Specifying gated layers properly is essential because one of the input neurons may not have completed calculating its final value before the layer that needs that value is done executing.

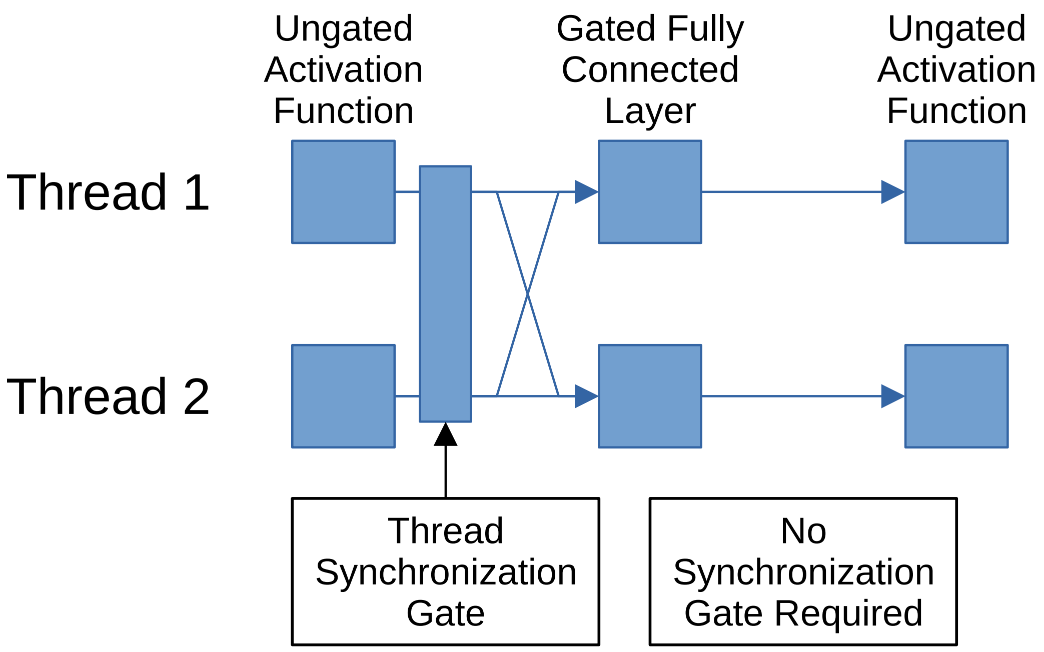

Ungated Activation Function
Gated Fully Connected Layer
Ungated Activation Function
Thread 1
Thread 2
Thread Synchronization Gate
No Synchronization Gate Required

Figure 6: A gated layer forces all threads to become synchronized before being able to execute.

The forward and backward pass functions that are required of the module will differ between a gated and ungated module [Figure 7]. This is because an ungated layer's neuron will only require a single input, whereas a gate layer neuron will rely on the inputs of many different inputs.

Gated	**Ungated**
`__device__ static inline PKML::float_t forward_gated( uint32_t thread_index, PKML::float_t * input, PKML::float_t * alloc )`	`__device__ static inline PKML::float_t forward_ungated( PKML::float_t input, PKML::float_t * alloc )`
`__device__ static inline void backward_gated( uint32_t thread_index, PKML::float_t * costs, PKML::float_t * input, PKML::float_t cost, PKML::float_t * alloc )`	`__device__ static inline PKML::float_t backward_ungated( PKML::float_t input, PKML::float_t output, PKML::float_t * alloc )`

Figure 7: Module header functions for gated and ungated modules.

The network configuration will provide the grid and block dimension to the network generator at compile time. This will determine the amount of threads running on the GPU. This is a critical element in neural network optimization. This is because each SM may only process a single block at a time, so having too many or too few blocks may cause some SMs to be unused. The block dimension will determine how many threads each SM must execute before continuing to the next block on the grid. Having the block dimension too low will underutilize the SM before the next block is swapped in.

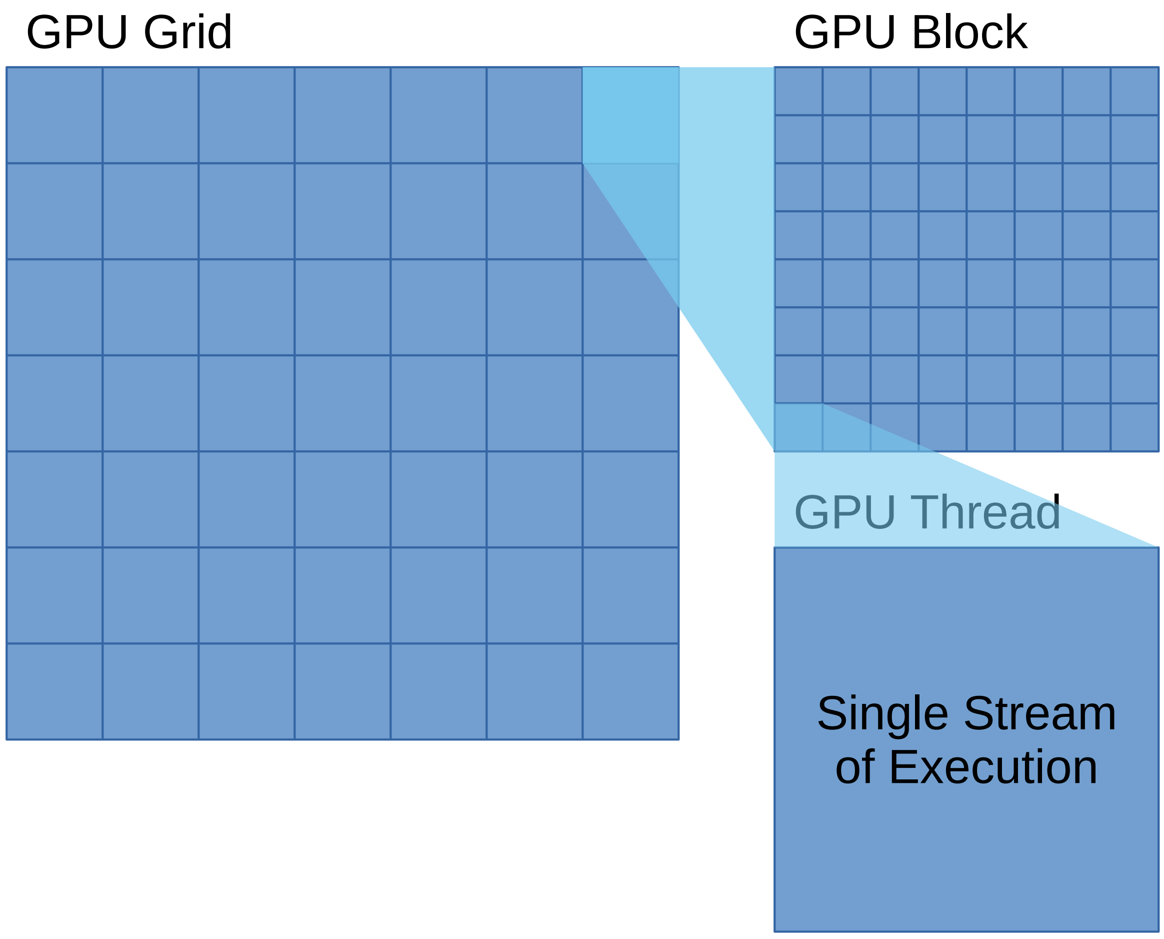


Figure 8: GPU grid, blocks, and threads.

There may also be threads that are required to wait during the execution of layers. This issue will be solved out of the module code by the generated source file. Thread execution will be blocked on specific threads to allow a one-to-one ratio of threads to neurons during the activation of each layer.

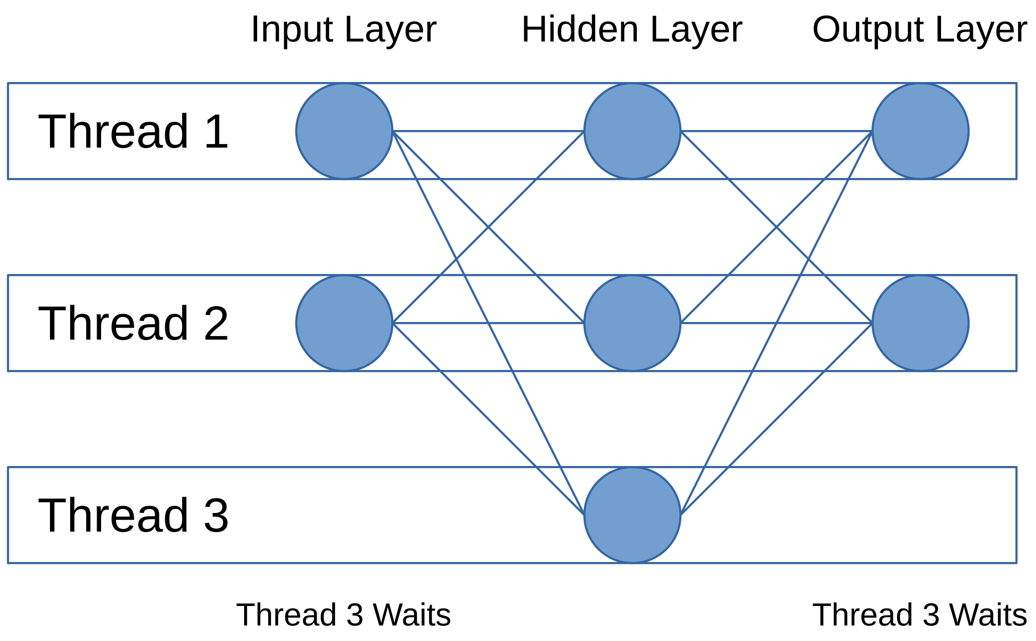


Figure 9: Threads must wait if they hit a layer with reduced neurons.

This method of thread control will cause a small amount of thread divergence as some threads will have to wait as others execute the forward and backward methods of the layers of the neural network. I believe that this will be negligible to the final performance of the application because this will allow the entire network training sequence to happen on the GPU without any CPU intervention. This should provide a much more significant performance benefit than the loss that the divergent threads may incur [12] [13].

The layers of the network will be supplied as an array of objects containing the module name, the input dimension, the output dimension, and an optional field for extra data to be passed to the network [Figure 8]. The module name will directly correspond to a class name defined in the modules included by the network configuration file. The input and output dimensions are specified as arrays, with each element corresponding to the size of that axis of the layer size. When memory is allocated for the layers these multidimensional values will be flattened to an array the size of the product of all elements of the dimension. The ‘params’ field will contain any other information that may be important to the function of the layer. This is mainly reserved for hyperparameters such as learning rate or convolution size.

```
{
   "module_name": "FullyConnected",
   "input_dimension":  [ 32, 32, 3 ],
   "output_dimension": [ 32, 32 ],
   "params": {
      "learning_rate": 0.0001
   }
},
```

Figure 10: A definition for a single fully connected layer.

The floating-point type provided in the network configuration will affect the numerical type used for all calculations in the resulting program. The tool supports double-precision, single-precision, and half-precision floating-point formats. With each reduction in precision, the memory bandwidth will be halved. This will allow for faster training of neural networks when memory bandwidth is the constraining factor.

The class name field in the network configuration will specify the desired name for the resulting C++ class. This will allow multiple networks to be compiled and linked together within the same project. This will be very important if this tool is to be used for real-world programs.

4 Methodology

The first network I will be training will be a network to simulate an XOR gate. This will be a simple beginning test to verify the validity of the training algorithms.

The following network I experiment on will be trained on the CIFAR10 dataset [8]. This dataset contains 50000 32x32 training images with three 8-bit color channels. These images all fall into 1 of 10 mutually exclusive categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, or truck. I will be using the binary version of this dataset, which I will pass into my GPU-bound dataset class [Figure 5]. I will then use this data to train my network.

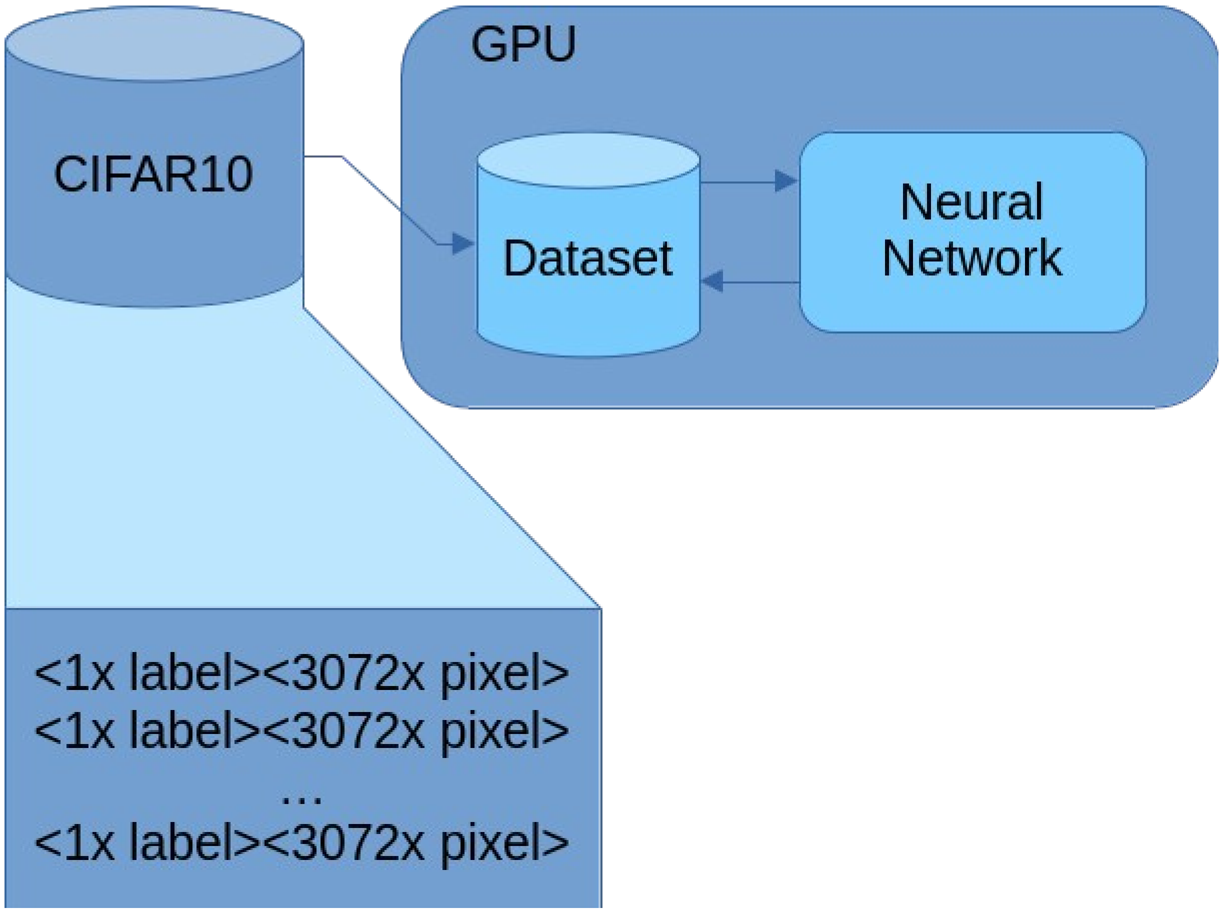


Figure 11: Flow of data from dataset to GPU.

The network I will be using for this dataset is a 5-layer multi-layer perceptron (MLP) network [Figure 6]. This network will have 32x32x3 input neurons, which will be supplied with the 32x32 three-channel image. This data will then flow through progressively constricting hidden layers until reaching the output layer. The output layer will consist of 10 neurons corresponding to the ten possible output categories in the CIFAR10 dataset [8].

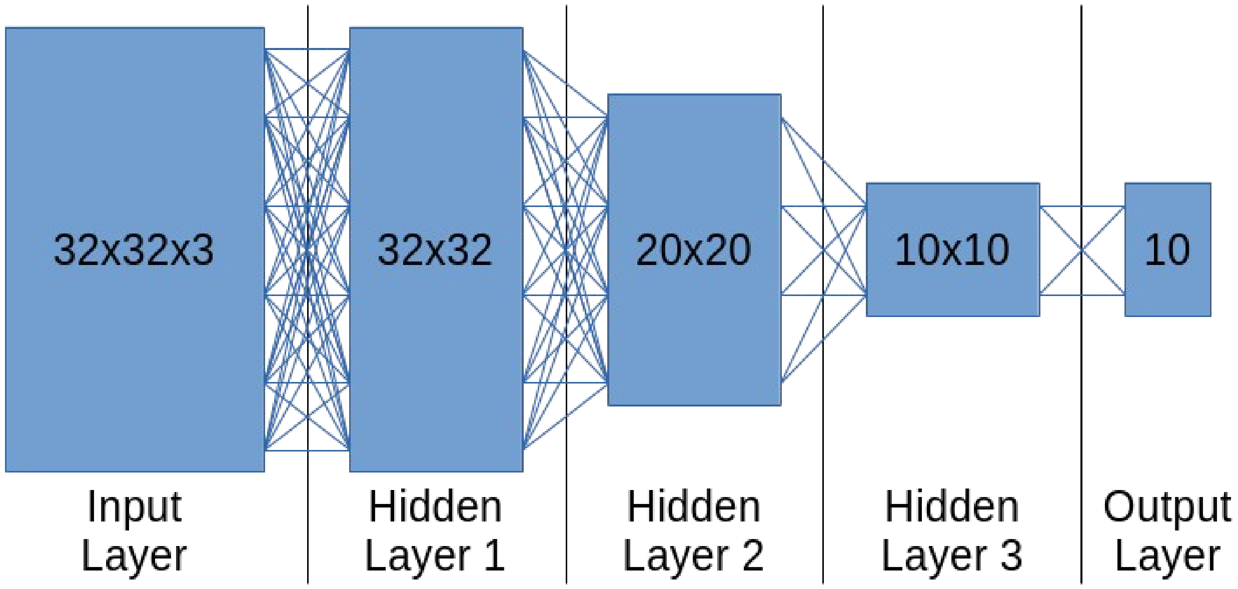
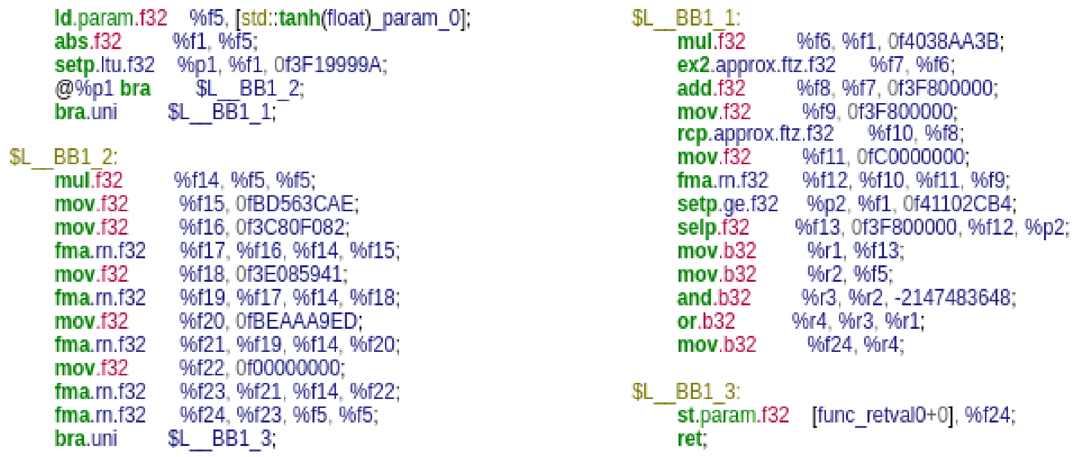


Figure 12: The network to be trained on the CIFAR10 dataset.

Each layer will be connected fully to the next layer. This will mean each neuron from one layer will be connected to every neuron on the next layer. I chose this layer type because of its simplicity and ease of implementation. While a fully-connected network may not be the most optimal type of network for image classification [9], because this network will only be compared to other fully-connected networks, this will not be an issue during the evaluation of the algorithms.

The network will be using a tanh function for the activation function applied to each layer. I felt that this was a good choice for the activation function because of its easy implementation and similarity to biological neuron activation rate [10, p. 687]. The downside of using a tanh function is its poor performance because of the large amount of instructions required to evaluate it [Figure 7]. Another possible choice was the Sigmoid activation function. I chose not to use this, however, because of its reduced performance compared to Tanh.



```
    ld.param.f32    %f5, [std::tanh(float)_param_0];
    abs.f32         %f1, %f5;
    setp.ltu.f32    %p1, %f1, 0f3F19999A;
    @%p1 bra        $L__BB1_2;
    bra.uni         $L__BB1_1;

$L__BB1_2:
    mul.f32         %f14, %f5, %f5;
    mov.f32         %f15, 0fBD563CAE;
    mov.f32         %f16, 0f3C80F082;
    fma.rn.f32      %f17, %f16, %f14, %f15;
    mov.f32         %f18, 0f3E085941;
    fma.rn.f32      %f19, %f17, %f14, %f18;
    mov.f32         %f20, 0fBEAAA9ED;
    fma.rn.f32      %f21, %f19, %f14, %f20;
    mov.f32         %f22, 0f00000000;
    fma.rn.f32      %f23, %f21, %f14, %f22;
    fma.rn.f32      %f24, %f23, %f5, %f5;
    bra.uni         $L__BB1_3;

$L__BB1_1:
    mul.f32         %f6, %f1, 0f4038AA3B;
    ex2.approx.ftz.f32      %f7, %f6;
    add.f32         %f8, %f7, 0f3F800000;
    mov.f32         %f9, 0f3F800000;
    rcp.approx.ftz.f32      %f10, %f8;
    mov.f32         %f11, 0fC0000000;
    fma.rn.f32      %f12, %f10, %f11, %f9;
    setp.ge.f32     %p2, %f1, 0f41102CB4;
    selp.f32        %f13, 0f3F800000, %f12, %p2;
    mov.b32         %r1, %f13;
    mov.b32         %r2, %f5;
    and.b32         %r3, %r2, -2147483648;
    or.b32          %r4, %r3, %r1;
    mov.b32         %f24, %r4;

$L__BB1_3:
    st.param.f32    [func_retval0+0], %f24;
    ret;
```

Figure 13: PTX output for the std::tanh function.

Training will consist of running the dataset through the network using code generated by my tool. The input to the PKMLC tool will be the configuration file provided in Appendix A. This will generate the CUDA C++ source and header file needed to train and activate the specified network.

I will be training the network in 50000 sample intervals. In between these training intervals, I will get an average cost for 100 randomly picked elements. I will then dump this average cost into a CSV file to be analyzed later to track the effectiveness of my training algorithm.

I will also be profiling the network using the Nvidia profiler tool ‘nvprof’, this tool is able to provide detailed breakdowns of the time GPU programs spend on each kernel launch [Figure 13]. I will be using this tool to compare different versions of my project. This will be very useful for comparing different unroll factors on loops.

```
==25881== NVPROF is profiling process 25881, command: ./a.out
==25881== Profiling application: ./a.out
==25881== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  6.2080us         1  6.2080us  6.2080us  6.2080us  test_k(float*)
      API calls:   96.53%  123.26ms         1  123.26ms  123.26ms  123.26ms  cudaMalloc
                    1.72%  2.2006ms         1  2.2006ms  2.2006ms  2.2006ms  cudaFree
                    1.66%  2.1194ms         1  2.1194ms  2.1194ms  2.1194ms  cudaLaunchKernel
                    0.07%  90.551us       114     794ns      75ns  36.284us  cuDeviceGetAttribute
                    0.01%  11.136us         1  11.136us  11.136us  11.136us  cuDeviceGetName
                    0.00%  3.2040us         1  3.2040us  3.2040us  3.2040us  cuDeviceGetPCIBusId
                    0.00%  1.0460us         3     348ns     117ns     770ns  cuDeviceGetCount
                    0.00%     424ns         2     212ns      90ns     334ns  cuDeviceGet
                    0.00%     315ns         1     315ns     315ns     315ns  cuDeviceTotalMem
                    0.00%     312ns         1     312ns     312ns     312ns  cuModuleGetLoadingMode
                    0.00%     174ns         1     174ns     174ns     174ns  cuDeviceGetUuid
```

Figure 14: Output from the 'nvprof' tool.

5 Results

The first experiment I performed was to train a network simulating an XOR gate with my tool [Figure 15]. I chose this as my first experiment to check if the training algorithms were working at all. I chose to simulate an XOR gate because of its complexity and ease of training.

Input	Output
00	0
01	1
10	1
11	0

Figure 15: A truth table for an XOR gate.

An XOR gate is somewhat more difficult to train on a neural network because it requires a hidden layer to be present. This is because the XOR gate is impossible to map to a single matrix multiplication [Figure 16].

A.	$A\begin{bmatrix}0\\0\end{bmatrix}=[0]$
B.	$A\begin{bmatrix}1\\0\end{bmatrix}=[1]$
C.	$A\begin{bmatrix}0\\1\end{bmatrix}=[1]$
D.	$A\begin{bmatrix}1\\1\end{bmatrix}=[0]$

$$A=\begin{bmatrix}1 & 1\end{bmatrix} \quad For\ A,B$$

$$A=\begin{bmatrix}0 & 0\end{bmatrix} \quad For\ D$$

Impossible

Figure 16: A single layer network cannot simulate an XOR gate.

To create the XOR network, I used the configuration file provided in Appendix C. The network is a fully-connected 2-layer MLP network [Figure 17].

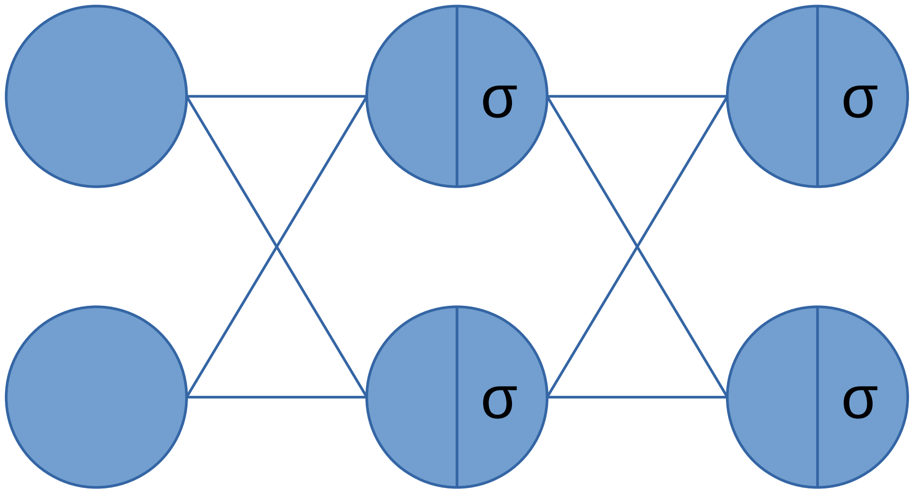


Figure 17: The network being used to simulate an XOR gate.

I will be training this network in intervals of 4000 training examples. Each interval will also calculate the total squared cost of the network over the dataset. This will be calculated by taking the square difference between each output and the expected output and adding them all together [Figure 18].

$$totalcost = \sum_{i=0}^{n_{dataset}-1} \left(\sum_{j=0}^{n_{output}-1} \left((dataset_{ij} - output_j)^2 \right) \right)$$

Figure 18: Equation for the total cost of a neural network.

I will be evaluating the validity of my training algorithm by measuring the total cost over these iterations. If this data approaches zero, I will know that my neural network is optimizing well and that the training algorithm is valid and working.

The resulting data I received after training the network for 100 iterations shows a downward-sloping cost that approaches zero [Figure 19]. These are the results I was hoping for and proves the validity of my network training tool.

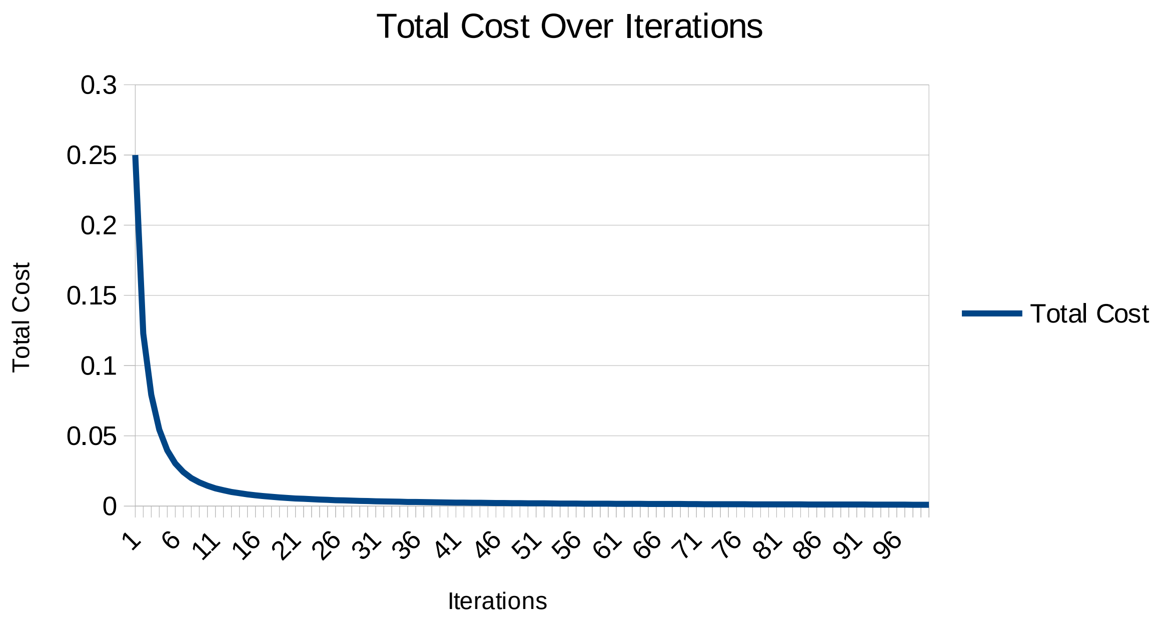


Figure 19: Average cost over 100 iterations of training with 4000 values.

The second experiment I performed was a comparison between the execution times of the network training function with and without the fully-connected layer loops expanded.

To perform this experiment I compiled two versions of the CIFAR10 network described in Figure 11. One of these binaries was compiled to expand the loops inside of the fully-connected layer forward and backward code segments. The other will use an unexpanded version. I ran both of these versions separately on my server and recorded the results using the ‘nvprof’ profiling tool [Figure 20].

Loop Unrolling Disabled	
train_k minimum execution time	266.417s
train_k maximum execution time	267.825s
train_k average execution time	267.557s
Loop Unrolling Enabled	
train_k minimum execution time	270.518s
train_k maximum execution time	272.736s
train_k average execution time	271.332s

Figure 20: Comparison of execution times with loops rolled and unrolled.

These results surprised me as I expected the loop unrolling to be made possible by the compile time provided network parameters to improve the execution times for the GPU program. This was a significant issue when developing my tool, as its entire success relies on its training algorithm's performance.

I sought to improve these results by modifying the unroll factor in the fully-connected layer's propagation functions. I evaluated unroll factors of 10 and 20. These unroll factors will cause the compiler to unroll the loop out to a maximum of 10 or 20 times. This will result in code that still requires branching but will have a much-reduced iteration count [Figure 21]

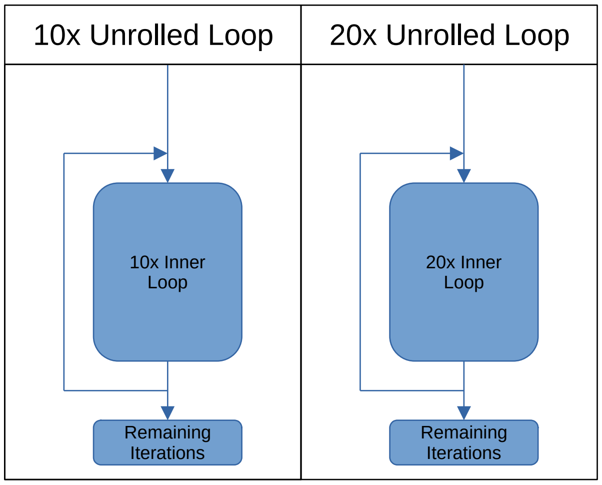


Figure 21: 10x unrolled loop vs. 20x unrolled loop.

Unfortunately, the smaller unrolling factors still did not achieve the performance that a standard loop was able to achieve. This is most likely because of the compiler’s more advanced selection of unroll factors.

The final experiment I performed was to train a network on the CIFAR10 dataset. This test will be done to examine the training algorithm’s effectiveness on deeper neural networks. During this training, an approximate average cost will be calculated at regular intervals to gauge the performance of the network training algorithms.

The approximate average cost will be calculated by retrieving a sample of 250 dataset elements. These elements will then be forwarded through the network, where the square error of all of them will be added together. This will then be divided by 250 to normalize the results [Figure 22].

$$cost = \frac{\sum_{i=0}^{250-1} \left(\sum_{j=0}^{n_{outputs}-1} (outputs_j - dataset_{(rand())j})^2 \right)}{250}$$

Figure 22: Calculation for the approximate average cost.

This cost will be calculated after every 50000 training sets have been passed through the network. This will allow the network to process the entire dataset between each cost evaluation.

I will be running this network for a total of 19 iterations, each running 50000 sets through the network. This will allow an initial view of the speed at which the average cost is being reduced. This will also allow me to analyze other elements of the network that may need to be tweaked, such as the learning rate or other hyperparameters.

The program was still able to produce a slight reduction in average cost with the amount of time it was given to train. It does, however, appear to plateau at around an average cost of about 0.175 [Figure 23].

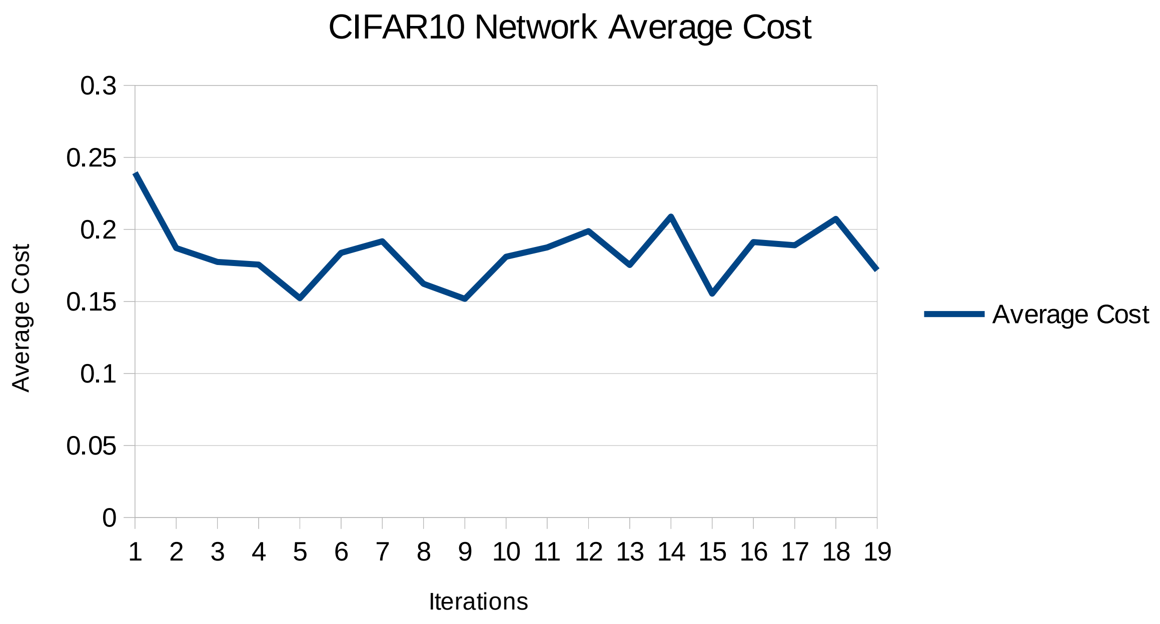


Figure 23: The average cost of the CIFAR10 network over iterations.

This plateauing nature may be caused by a number of reasons. I think the biggest reason for the lack of improvement is the shorter training time I was able to give to this experiment. Nineteen iterations may not have given the network enough time to find an optimal configuration.

The issues in this experiment may also be caused by a very low learning rate provided in the network configuration. This may be low enough that the network is unable to keep reducing its average cost at a reasonable speed.

In all I believe this experiment went quite well. There is a clear drop in cost towards the beginning of Figure 23, which shows that the training algorithms are working to some degree. This proves that the training algorithm is able to handle more extensive networks and training sets.

6 Discussion

Unfortunately, the expansion of conditional loops did not result in a faster program. I believe one reason for the decreased speed when loop expansion was applied is the GPU cache. I suspect that the cache is unable to load all the instructions from the expanded loop code segments. This is because the loop generates too many instructions for the cache to store, whereas, in an unexpanded loop, the size of the generated instructions is relatively few, so they may all be cached while the program runs.

I believe that the failure of the unrolled loops to improve performance may also be due to the compiler’s heuristics for predicting the adequate unroll factor for loops. When compiling a test program with a single loop, the compiler may automatically produce an unrolled loop [Figure 24].

```
        ld.param.u64    %rd4, [void test_k<100>(float*)_param_0];
        cvta.to.global.u64      %rd5, %rd4;
        mov.u32         %r4, 0;

$L__BB0_1:
        ld.global.f32   %f1, [%rd5];
        add.f32         %f2, %f1, %f1;
        st.global.f32   [%rd5], %f2;
        ld.global.f32   %f3, [%rd5+4];
        add.f32         %f4, %f3, %f3;
        st.global.f32   [%rd5+4], %f4;
        ld.global.f32   %f5, [%rd5+8];
        add.f32         %f6, %f5, %f5;
        st.global.f32   [%rd5+8], %f6;
        ld.global.f32   %f7, [%rd5+12];
        add.f32         %f8, %f7, %f7;
        st.global.f32   [%rd5+12], %f8;
        add.s64         %rd5, %rd5, 16;
        add.s32         %r4, %r4, 4;
        setp.ne.s32     %p1, %r4, 100;
        @%p1 bra        $L__BB0_1;

        ret;
```

Figure 24: Code generated from a loop set to run for 100 iterations.

Figure 24 shows the result of compiling a for loop set to execute 100 times. The compiler, in this case, decided to unroll the loop by a factor of 4. This can be seen by the amount of load, add, and store sequences the compiler produces inside of the loop. This result is the result of many compiler heuristics [11], which will generally be faster than a chosen unroll factor.

I do believe that my tool may still have a place for use in high-performance machine learning settings. While the full loop unrolling was unable to boost performance since the network parameters are provided at compile time, the compiler is able to perform these more well-informed loop expansions, which should give performance benefits over loops that are not provided with compile-time constant limits.

7 Conclusion and Future Work

The development of my tool has been able to accomplish its primary goal of allowing the implementation of neural networks with parameters provided at compile time easily. This is facilitated through configuration files, which are provided to the network generator. I believe this will be a great help to me and others in the future wishing to implement high performance GPU bound neural networks easily and quickly.

Unfortunately the complete success of the project was not fully achieved as I was not able to get improved performance from fully unrolling the loops generated by the program. I also was not able to compare my application to other tools available currently, so it is not clear whether this method of providing compile-time network parameters will have a significant effect on performance.

I believe that even though the tool did not perform as expected with the entire loop unrolling, there is still potential for this type of program to improve neural network training speeds. By providing network parameters at compile-time, you allow the compiler to make optimizations that would otherwise not be possible.

I believe that in the future I will be able to add to this project to be able to make it competitive against other neural network tools. I believe this is true because giving more information to a compiler generally can only result in improved execution performance. The main goal for future work will be to hyper-optimize the module and main code of this project to bring it on par with other tools similar to it.

REFERENCES

[1] Harris M., and Buck I. *Chapter 34. GPU Flow-Control Idioms* https://developer.nvidia.com/gpugems/gpugems2/part-iv-general-purpose-computation-gpus-primer/chapter-34-gpu-flow-control-idioms (04/28/2024).

[2] Reznik A., *et al Why GPUs are essential for AI and high-performance computing* https://developers.redhat.com/articles/2022/11/21/why-gpus-are-essential-computing (04/28/2024).

[3] Flensted T. *Top 23 AI Chip Makers of 2024 - Statistics & Facts* https://seo.ai/blog/ai-chip-makers (04/28/2024).

[4] Zhang L., Wahib M., and Matsuoka S. *Understanding the Overheads of Launching CUDA Kernels* https://www.hpcs.cs.tsukuba.ac.jp/icpp2019/data/posters/Poster17-abst.pdf (04/28/2024).

[5] Jang H., Park A., and Jung K. *Neural Network Implementation Using CUDA and OpenMP In Proceedings 2008 Digital Image Computing: Techniques and Applications.* pp. 155-161. (Canberra, ACT, Australia, 2008).

[6] Sierra-Canto X., Madera-Ramirez F., and Uc-Cetina V. *Parallel Training of a Back-Propagation Neural Network Using CUDA In Proceedings 2010 Ninth International Conference on Machine Learning and Applications*. pp. 307-312. (Washington, DC, 2010).

[7] *GeForce GTX 1050 Ti | Specifications | GeForce.* https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1050-ti/specifications/ (04/28/2024)

[8] Krizhevsky A., Nair V., and Hinton G. *The CIFAR-10 dataset.* https://www.cs.toronto.edu/~kriz/cifar.html (04/28/2024)

[9] Malach E., and Shalev-Shwartz S. *Computational Separation Between Convolutional and Fully-Connected Networks*. https://arxiv.org/abs/2010.01369 (04/28/2024)

[10] Lau M., and Lim K. *Review of Adaptive Activation Function in Deep Neural Network In Proceedings 2018 IEEE-EMBS Conference on Biomedical Engineering and Sciences (IECBES).* pp. 686-690. (Sarawak, Malaysia, 2018).

[11] Murthy G. S. *et al*. *Optimal loop unrolling for GPGPU programs In Proceedings 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* pp. 1-11. (Atlanta, GA, 2010).

[12] Kim S., Oh S., and Yi Y. *Minimizing GPU Kernel Launch Overhead in Deep Learning Inference on Mobile GPUs In Proceedings 22nd International Workshop on Mobile Computing Systems and Applications.* pp. 57-63. (New York, NY, 2021).

[13] Bialas P., and Strzelecki A. *Benchmarking the Cost of Thread Divergence in CUDA In Proceedings International Conference on Parallel Processing and Applied Mathematics.* pp. 570-579. (Krakow, Poland, 2015).

APPENDIX A: PKML Definition for a Network to Solve CIFAR10

```
{
  "modules": [
    "modules/fully_connected.pkmlm",

    "modules/tanh.pkmlm"
  ],
  "network": {
    "class_name": "InstanceCifar10",
    "float_type": "float",

    "grid_dim": 6,
    "block_dim": 512,

    "layers": [
      {
        "module_name": "FullyConnected",
        "input_dimension":  [ 32, 32, 3 ],
        "output_dimension": [ 32, 32 ],
        "params": { "learning_rate": 0.0001 }
      },
      {
        "module_name": "Tanh",
        "input_dimension":  [ 32, 32 ],
        "output_dimension": [ 32, 32 ]
      },
      {
        "module_name": "FullyConnected",
        "input_dimension":  [ 32, 32 ],
        "output_dimension": [ 20, 20 ],
        "params": { "learning_rate": 0.0001 }
      },
      {
        "module_name": "Tanh",
        "input_dimension":  [ 20, 20 ],
        "output_dimension": [ 20, 20 ]
      },
      {
        "module_name": "FullyConnected",
        "input_dimension":  [ 20, 20 ],
        "output_dimension": [ 10, 10 ],
        "params": { "learning_rate": 0.0001 }
      },
      {
        "module_name": "Tanh",
        "input_dimension":  [ 10, 10 ],
        "output_dimension": [ 10, 10 ]
      },
      {
        "module_name": "FullyConnected",
        "input_dimension":  [ 10, 10 ],
        "output_dimension": [ 10 ],
        "params": { "learning_rate": 0.0001 }
      },
      {
        "module_name": "Tanh",
        "input_dimension":  [ 10 ],
        "output_dimension": [ 10 ]
      }
    ]
  }
}
```

APPENDIX B: Fully-Connected Layer Module Header and Manifest

```
#pragma once

#include <cmath>
#include <memory>

#include <pkml.hpp>

struct FullyConnected_params {
    PKML::float_t learning_rate;
};

template<typename input_dimension, typename output_dimension, FullyConnected_params params>
struct FullyConnected {
    struct allocation_t {
        PKML::float_t weights[output_dimension::element_product][input_dimension::element_product];
        PKML::float_t biases[output_dimension::element_product];
    };

    static inline void init(PKML::float_t * alloc) {
        allocation_t * host = new allocation_t;
        for (std::size_t i = 0; i < output_dimension::element_product; i++) {
            for (std::size_t j = 0; j < input_dimension::element_product; j++) {
                host->weights[i][j] = 0.001;
            }
        }
        for (std::size_t i = 0; i < output_dimension::element_product; i++) host->biases[i] = 0.001;

        cudaMemcpy(alloc, host, sizeof(allocation_t), cudaMemcpyHostToDevice);

        delete host;
    }

    __device__ static inline PKML::float_t forward_gated(uint32_t thread_index, PKML::float_t * input, PKML::float_t * alloc) {
        allocation_t & allocation = *((allocation_t *) alloc);

        PKML::float_t sum = 0;
#pragma unroll
        for (std::size_t i = 0; i < input_dimension::element_product; i++) {
            sum = PKML::Math::fma(input[i], allocation.weights[thread_index][i], sum);
        }
        return sum;
    }

    __device__ static inline void backward_gated(uint32_t thread_index, PKML::float_t * costs, PKML::float_t * input, PKML::float_t cost,
PKML::float_t * alloc) {
        allocation_t & allocation = *((allocation_t *) alloc);

#pragma unroll
        for (std::size_t i = 0; i < input_dimension::element_product; i++) {
            costs[i] = PKML::Math::fma(
                cost,
                allocation.weights[thread_index][i],
                costs[i]
            );

            allocation.weights[thread_index][i] = PKML::Math::fma(
                -params.learning_rate,
                PKML::Math::mul(cost, input[i]),
                allocation.weights[thread_index][i]
            );
        }

        allocation.biases[thread_index] = PKML::Math::fma(
            -params.learning_rate,
            cost,
            allocation.biases[thread_index]
        );
    }

    static constexpr std::size_t memory_requirement = sizeof(allocation_t) / sizeof(PKML::float_t);
};
```

```
{
  "class_file": "fully_connected.hpp",
  "class_name": "FullyConnected",
  "gated": true
}
```

APPENDIX C: XOR Network Configuration File

```
{
    "modules": [
        "modules/fully_connected.pkmlm",

        "modules/sigmoid.pkmlm"
    ],
    "network": {
        "class_name": "InstanceXor",
        "float_type": "float",

        "grid_dim": 1,
        "block_dim": 4,

        "layers": [
            {
                "module_name": "FullyConnected",
                "input_dimension":  [ 2 ],
                "output_dimension": [ 4 ],
                "params": { "learning_rate": 0.5 }
            },
            {
                "module_name": "Sigmoid",
                "input_dimension":  [ 4 ],
                "output_dimension": [ 4 ]
            },
            {
                "module_name": "FullyConnected",
                "input_dimension":  [ 4 ],
                "output_dimension": [ 1 ],
                "params": { "learning_rate": 0.5 }
            },
            {
                "module_name": "Sigmoid",
                "input_dimension":  [ 1 ],
                "output_dimension": [ 1 ]
            }
        ]
    }
}
```