2
Implementation Background

2.1) Neural Network Acceleration

By studying the history of neural networks, it became apparent that the ideas were there from the 1970s but data and computation power was missing. With the help of the internet and the digitization of the world, immense amount of data has been amassed. Now, more than ever, there is need for more computational power which will enable processing of big datasets and training of deeper and larger neural networks.

A prime example of such a network is GPT-3 [41]. GPT-3 is an NLP (Natural Language Processing) focused neural network. It has been trained on a dataset of 300 billion words. It has 175 billion parameters and requires 3.14E+23 FLOPS to be trained. This is an astronomical number.

The fastest GPU for neural network training is Nvidia's A100. Let us assume that this network can be trained using 16 bit floating point precision (bfloat16) instead of 32 bit. Assuming that parameters and activations are quite sparse, peak performance of the A100 on sparse bfloat16 data is 624 TFLOPS. On one GPU it would take approximately 16 years to train this network. Of course, the 624 TFLOPS figure is very far off from what is practically achievable which is probably closer to 150 TFLOPS. In addition, the creators of GPT-3 show empirically that neural network accuracy scales as a power-law of model size, which suggests that progress in deep neural networks is limited mainly by available computation power.

While focus is slowly shifting away from using GPUs and onto custom domain specific silicon for neural network accelerators like Google's TPU [42], it is more important than ever to find new strategies for performance optimization on hardware we already have, be it GPUs or FPGAs.

2.1.1) Convolution Algorithms

A prime candidate for acceleration in convolutional neural networks is the convolution layer which takes close to 95% of the computation power needed to run a neural network for inference or for training. Strategies to achieve that are discussed in this paper [43] by Andrew Lavin and Scott Gray. They show that there are 2 main methods to accelerate convolution. A method using FFT (Fast Fourier Transform) and one using Winograd's Minimal Filtering Algorithms on which we will focus on as they are ideal for the small filter sizes used in state-of-the-art computer vision neural networks. These algorithms can considerably reduce the number of multiplications needed for computing the convolution layer. In addition, most of the arithmetic operations are dense matrix multiplies of sufficient dimensions to be computed efficiently. The memory requirements are also light compared to FFT-based convolution algorithms. These factors make practical implementations possible on modern GPUs and especially FPGAs which have a limited amount of DSP resources.

2.1.1.1) Direct Convolution

For a better understanding of the current and following sections, it is important to keep in mind the dimensions of the inputs and outputs to the layer. The input tensor to the convolution layer has the following dimensions: $Batch\ Size \times Height_{in} \times Width_{in} \times Channels$. The weight tensor has the following dimensions $Height_W \times Width_W \times Channels \times Filters$. The output tensor has the following dimensions $Batch\ Size \times Height_{out} \times Width_{out} \times Filters$. As already mentioned, during convolution a small matrix slides over a larger matrix. Thus, the height and width dimensions of the output depend on the stride used for sliding over the input and also on the padding used. Size of each output spatial dimension is given by the following formula $(W - F + 2P)/S + 1$ where W represents the input's spatial dimension, F the receptive field size of the convolution layer neurons and P the amount of zero padding on the borders. Usually, the image is zero-padded on its borders in order to ensure that the output maps have the same spatial dimensions as the input.

To reiterate, convolution correlates F filters with C channels of size $H_w \times W_w$ against a minibatch of N images with C channels of size $H_{in} \times W_{in}$. We will name image elements as $x_{n,h_{in},w_{in},c}$ and filter elements as $w_{h_w,w_w,c,f}$. A single output $y_{n,h_{out},w_{out},f}$ can be computed in the following way:

$$y_{n,h_{out},w_{out},f} = \sum_{c=1}^{C} \sum_{h_w=1}^{H_w} \sum_{w_w=1}^{W_w} x_{n,h_{in}+h_w,w_{in}+w_w,c}\ w_{h_w,w_w,c,f}$$

Equation 1: Convolution Computation

The direct convolution computation is also presented below in code.

```
// Convolution
for (n=0; n<BATCH_SIZE; n++){
    for (h_out=0; h_out<HEIGHT_OUT; h_out++){
        for (w_out=0; w_out<WIDTH_OUT; w_out++){
            for (f=0; f<FILTERS; f++){
                temp=0;
                for (c=0; c<CHANNELS; c++){
                    for (h_w=0; h_w<HEIGHT_W; h_w++){
                      for(w_w=0; w_w<WIDTH_W; w_w++){ //element-wise multiplication

                          temp += input[n][c][h_w + h_out*STRIDE][w_w + w_out*STRIDE]
                                                                  * weights[f][c][h_w][w_w];

                      }
                    }
                }
                out[n][f][h_out][w_out] = temp;
            }
        }
    }
}
```

Figure 1: Code Representation of Convolution Computation

2.1.1.2) Winograd Convolution

Winograd's minimal filtering algorithm [45] for computing m outputs with an r-tap FIR filter, which we call $F(m,r)$, requires $\mu(F(m,r)) = m + r - 1$ multiplications. It is possible to nest such 1D algorithms $F(m,r)$ and $F(n,s)$ to form minimal 2D algorithms for computing $m \times n$ outputs with an $r \times s$ filter, which we call $F(m \times n, r \times s)$. These require $\mu\big(F(m \times n, r \times s)\big) = \mu\big(F(m,r)\big)\mu\big(F(n,s)\big) = (m + r - 1)(n + s - 1)$ multiplications.

Fast filtering algorithms can be written in matrix form as:

$$Y = A^T[(Gg) \odot (B^T d)]$$

Equation 2: 1D Fast Filtering Algorithm Matrix Function

where $\odot$ indicates element-wise multiplication, g represents the filter and d the input, while the rest of the matrices are constant.

A minimal 1D algorithm $F(m, r)$ is nested with itself to obtain a minimal 2D algorithm, $F(m \times m, r \times r)$ like so:

$$Y = A^T[(GgG^T) \odot (B^T dB)]A$$

Equation 3: 2D Fast Filtering Algorithm Matrix Function

where now g is an $r \times r$ filter and d is an $(m + r - 1) \times (m + r - 1)$ image tile.

As per a paper by Nvidia [44], these constant matrices are generated, for a tile of size $F(m \times m, n \times n)$, by the following procedure.

$$A^T = (V_{(m+n-1)\times m})^T S_A$$

$$G = S_G (V_{(m+n-1)\times(m+n-1)})^{-T}$$

$$B^T = S_B (V_{(m+n-1)\times n})$$

Where $V_{a \times b}$ is a trimmed Vandermonde matrix for Homogeneous Coordinate polynomials. This can be expressed directly with the following:

$$\begin{bmatrix} {f_0}^0 {g_0}^{b-1} & {f_0}^1 {g_0}^{b-2} & \cdots & {f_0}^{b-1} {g_0}^0 \\ {f_1}^0 {g_1}^{b-1} & {f_1}^1 {g_1}^{b-2} & \cdots & {f_1}^{b-1} {g_1}^0 \\ \vdots & & & \vdots \\ {f_{a-1}}^0 {g_{a-1}}^{b-1} & {f_{a-1}}^1 {g_{a-1}}^{b-2} & \cdots & {f_{a-1}}^{b-1} {g_{a-1}}^0 \end{bmatrix}$$

Where (f_i, g_i) are given unique homogeneous coordinates, we will call polynomial points. Note that the chosen (f_i, g_i) must be the same provided for all matrices $(A^T, G$ and $B^T)$. S_A, S_g and S_B are given diagonal square matrices where $S_A S_B S_G = I$. When referring to these matrices, we will only list the diagonal values as a vector; these values follow the diagonal square matrix from left-right.

As Andrew Lavin and Scott Gray [43] found out, algorithms for $F(m \times m, r \times r)$ can be used to compute convolution layers with $r \times r$ kernels. Each image channel is divided into tiles of size $(m + r - 1) \times (m + r - 1)$, with $r - 1$ elements of overlap between neighboring tiles, yielding $P = [H/m][W/m]$ tiles per channel, C. $F(m \times m, r \times r)$ is then computed for each tile and filter combination in each channel, and the results are summed over all channels. Substituting $U = GgG^T$ and $V = B^T dB$, we have:

$$Y = A^T [U \odot V] A$$

Labeling tile coordinates as $(\tilde{x}, \tilde{y})$, we rewrite the convolution computation formula for a single image i, and filter f as:

$$\begin{aligned} Y_{i,f,\tilde{x},\tilde{y}} &= \sum_{c=1}^{C} D_{i,c,\tilde{x},\tilde{y}} * G_{f,c} \\ &= \sum_{c=1}^{C} A^T \left[U_{f,c} \odot V_{c,i,\tilde{x},\tilde{y}} \right] A \\ &= A^T \left[\sum_{c=1}^{C} U_{f,c} \odot V_{c,i,\tilde{x},\tilde{y}} \right] A \end{aligned}$$

Equation 4: Winograd Convolution

Thus, we can reduce over C channels in transform space, and only then apply the inverse transform A to the sum. This amortises the cost of the inverse transform over the number of channels. We examine the sum

$$M_{i,f,\tilde{x},\tilde{y}} = \sum_{c=1}^{C} U_{f,c} \odot V_{c,i,\tilde{x},\tilde{y}}$$

and simplify the notation by collapsing the image/tile coordinates $(i, \tilde{x}, \tilde{y})$ down to a single dimension, b. We also label each component of the element-wise multiplication separately, as (ξ, ν), yielding:

$$M_{f,b}^{(\xi,\nu)} = \sum_{c=1}^{C} U_{f,c}^{(\xi,\nu)} V_{c,b}^{(\xi,\nu)}$$

This equation is just a matrix multiplication, so we can write:

$$M^{(\xi,\nu)} = U^{(\xi,\nu)} V^{(\xi,\nu)}$$

Matrix multiplication can be efficiently implemented on CPU, GPU, and FPGA platforms, owing to its high computational intensity. Thus, we have arrived at an efficient implementation for Winograd's Fast Filtering algorithms.

A minimal algorithm for $F(4,3)$ has the form:

The data transform takes overlapping input data and creates $6x6$ Winograd Tiles. A tile will next go through the element wise multiplication and then will pass through the output transform which reduces it to a $4x4$ tile.

The data transform uses 13 floating point instructions, the filter transform uses 8, and the inverse transform uses 10.

Applying the nesting formula yields a minimal algorithm for $F(4 \times 4, 3 \times 3)$ that uses $6 \times 6 = 36$ multiplies, while the standard algorithm uses $4 \times 4 \times 3 \times 3 = 144$. This is an arithmetic complexity reduction of 4.

The 2D data transform uses $13(6 + 6) = 156$ floating point instructions, the filter transform uses $8(3 + 6) = 72$, and the inverse transform uses $10(6 + 4) = 100$.

The number of additions and constant multiplications required by the minimal Winograd transforms increases quadratically with the tile size [47]. Thus, for large tiles, the complexity of the transforms will overwhelm any savings in the number of multiplications.

The magnitude of the transform matrix elements also increases with increasing tile size. This effectively reduces the numeric accuracy of the computation, so that for large tiles, the transforms cannot be computed accurately without using high precision datatypes [45].

I have chosen this particular minimal filtering algorithm to implement on GPU and FPGA, because it reduces the number of multiplications needed while being numerically accurate. While this seems to be

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

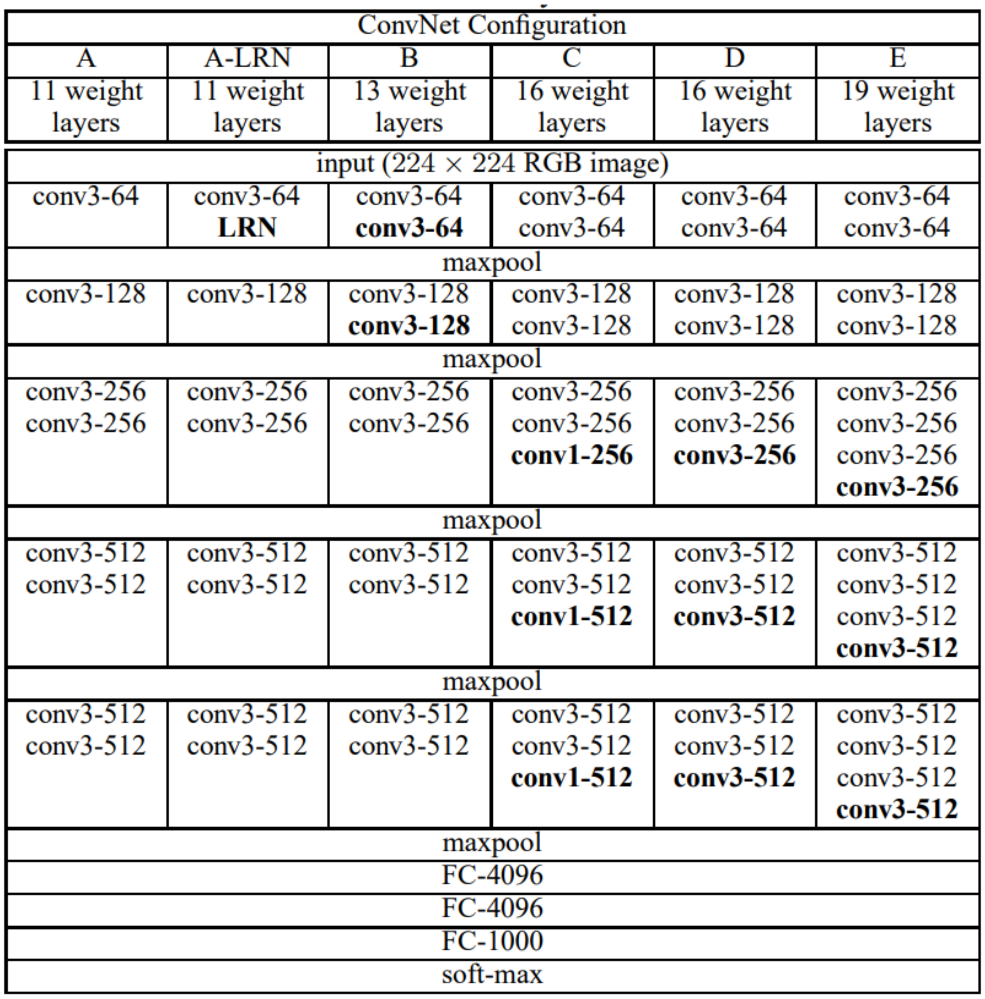
$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}$$

the best algorithm suited for FPGAs which have a limited number of DSPs, the same cannot be said with any certainty for GPUs as the big 6×6 tile size is prohibitive to fusing the 3 steps of the computation in one kernel. In addition, there is almost no data re-use in the transform steps, which makes it impossible to properly utilise the memory hierarchy to limit latencies and hide memory transfers, leading to a low computation intensity, memory bound implementation on GPUs. Thus, depending on the GPU used and its peak FLOPS to bandwidth ratio, the $F(2 \times 2, 3 \times 3)$ algorithm, which has a significantly smaller 4×4 input tile size, might be better suited for the task.

As already mentioned, the convolution layer is the most compute intensive layer. Since, I will be implementing it with the minimal filtering algorithm mentioned above, the matrix multiply step is what will be the focus of my optimization efforts because of its high compute intensity. The rest of the computation steps will be structured around it.

Note that up to this point the procedure for the forward pass of the convolution layer was shown. Fortunately, computing the gradients is trivial as Winograd's minimal filtering algorithm can be thought of as a collection of matrix multiplications.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64	conv3-64	conv3-64	conv3-64	conv3-64
	LRN	**conv3-64**	conv3-64	conv3-64	conv3-64
maxpool					
conv3-128	conv3-128	conv3-128	conv3-128	conv3-128	conv3-128
		conv3-128	conv3-128	conv3-128	conv3-128
maxpool					
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
			conv1-256	**conv3-256**	conv3-256
					conv3-256
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	**conv3-512**	conv3-512
					conv3-512
maxpool					
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
			conv1-512	**conv3-512**	conv3-512
					conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2: Original VGG Configurations

2.2) Target Neural Network

I wanted to integrate Winograd Convolution in an established neural network to see how overall performance is affected. VGG16, due to its simple and uniform architecture of 3×3 convolution filters, appears to be a good choice to showcase the $F(4 \times 4, 3 \times 3)$ algorithm's performance.

While the VGG network has already been discussed in the Neural Network Background section its architecture will be shown here in greater detail.

The creators of the VGG network [28] had one goal in mind. They wanted to prove that deeper neural networks performed better than shallower networks. They achieved their goal by creating a 19 layer network, quite deep for the time, which significantly outperformed previous state-of-the-art networks.

In addition, they showed that larger filter sizes weren't needed to achieve good results. If 2 convolution layers of 3×3 filters are stacked together, they have an effective 5×5 receptive field while if 3 of them are stacked, it results in an effective 7×7 receptive field. This way the network is more flexible, as there are more non-linearities used while at the same time reducing the number of parameters. This can be seen if we assume that both the input and the output of a three-layer 3×3 convolution stack has C channels, the stack is parameterized by $(3^2C^2) = 27C^2$ weights; at the same time, a single 7×7 conv. layer would require $(7^2C^2) = 49C^2$ parameters, i.e. 81% more. This can be seen as imposing regularisation on the 7×7 conv. filters, forcing them to be decomposed through the 3×3 filters (with non-linearity injected in between). The original paper first came out with 6 different VGG networks shown in Figure 22. The most notable ones being the D and E configurations also known as VGG16 and VGG19 respectively with a staggering 138 and 144 million parameters respectively.

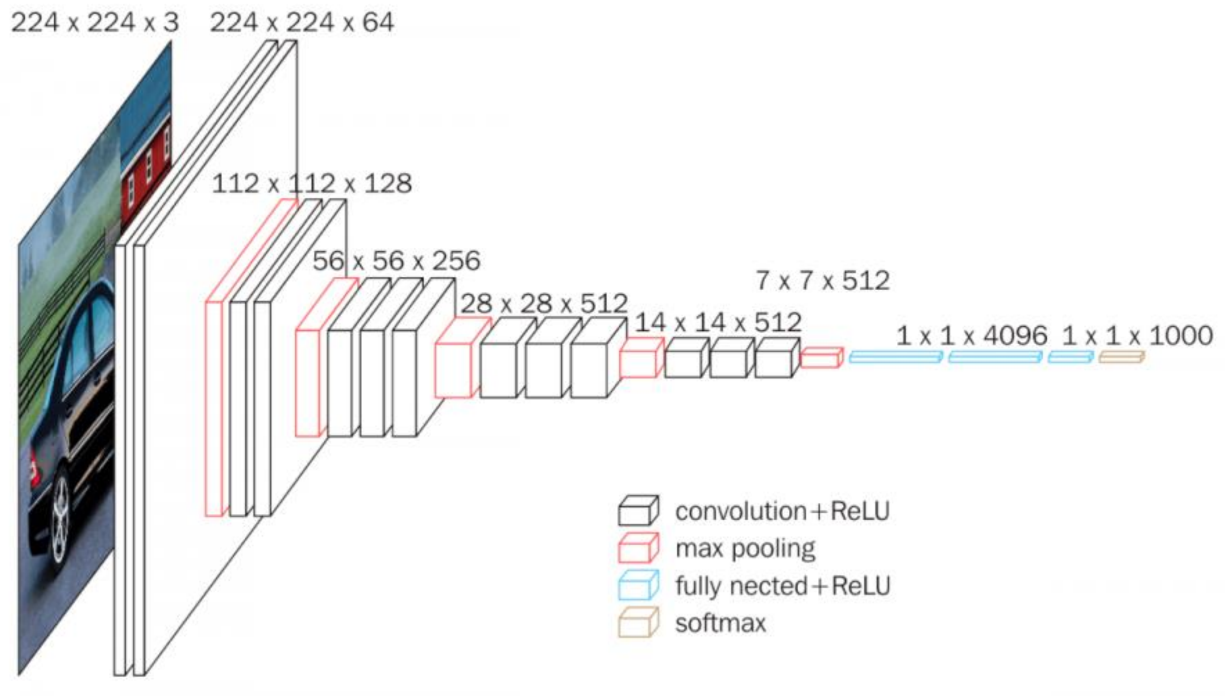


Figure 3: Spatial Representation of VGG16's Activation Maps

At the time, there was no Batch Normalization layer and the authors weren't aware of Glorot's and Bengio's [19] initialization scheme. Consequently, they first trained the A network which was shallow enough to be trained despite using random weight initialization. They then copied the weights of the first 4 convolution layers and those of the fully connected ones to the E configuration network and started training from scratch. They used a batch size of 256, an SGD with momentum optimizer, a regularization strength of 5e-4 and a starter learning rate of 0.01. In addition, they used data augmentation in the form of image rotation and random colour shifts as well as random central zooming of the image.

They used Caffe to train the network using 4 Nvidia Titan Black GPUs (GK110b) and training for 74 epochs. They achieved a 74.5% top-1 accuracy and 92% top-5 accuracy on the Imagenet classification task using a single network. A spatial representation of VGG16's layers can be seen in Figure 23.

2.3) GPU Architecture

For the purposes of this thesis, the GP104 GPU will be used. It is manufactured on a 16 nm FinFet node and has a total of 7.2 Billion transistors on a 314 mm² die. It consists of 4 GPCs (Graphics Processing Clusters) and 8 Memory Controllers which are responsible for handling the 8GB of GDDR5X VRAM, running

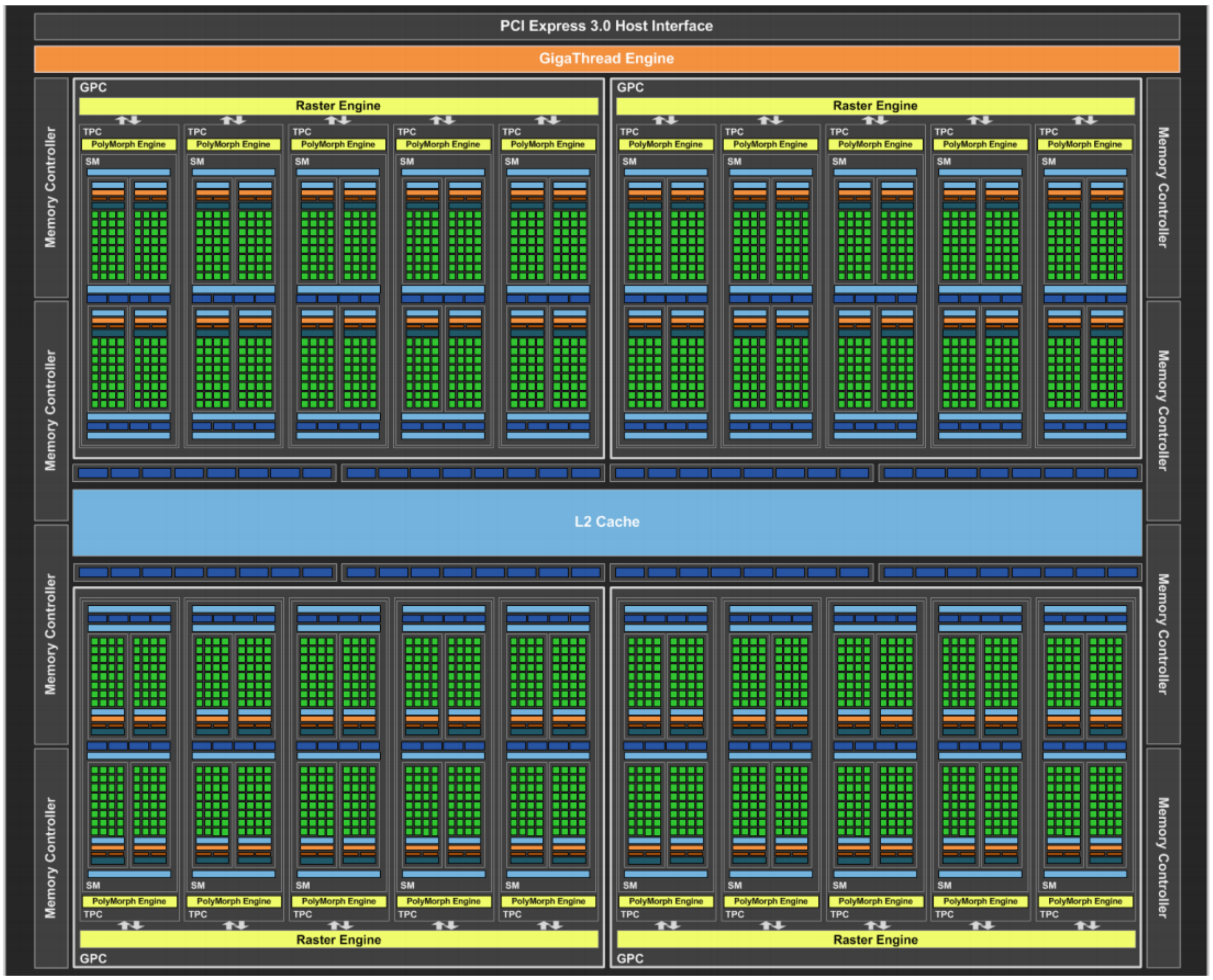


Figure 4: Block Diagram of GP104

at 10 Gbps, which is connected to them through a 256-bit bus yielding a peak memory bandwidth of 320 GB/s. The 2MB L2 cache is positioned in the middle of the GPU, sandwiched between the GPCs. Each GPC consists of 5 SMs (Streaming Multiprocessors) for a total of 20 SMs. A top-level block diagram of the GP104 GPU can be seen in Figure 24.

Each SM (Figure 25) contains 128 CUDA cores, a 256KB register file partitioned into 65536 32-bit registers, a 96KB Shared Memory unit, partitioned in 32 banks, which acts as a programmer accessible L1 cache, a total of 48KB of non-accessible L1 cache and 8 texture units which perform specialised load instructions. There is also an instruction cache, 4 warp schedulers with 2 dispatch units each, 32 Load/Store units and 32 Special Function Units. The SM is the core processing unit of the GPU. An SM is designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs a SIMT (Single-Instruction, Multiple-Thread) architecture. The instructions are pipelined, leveraging instruction-level parallelism within a single thread, as well as extensive thread-level parallelism through simultaneous hardware multithreading. Unlike most modern CPU cores, instructions are issued in order and there is no branch prediction which makes it the job of the programmer (and the compiler by extension) to extract sufficient parallelism.

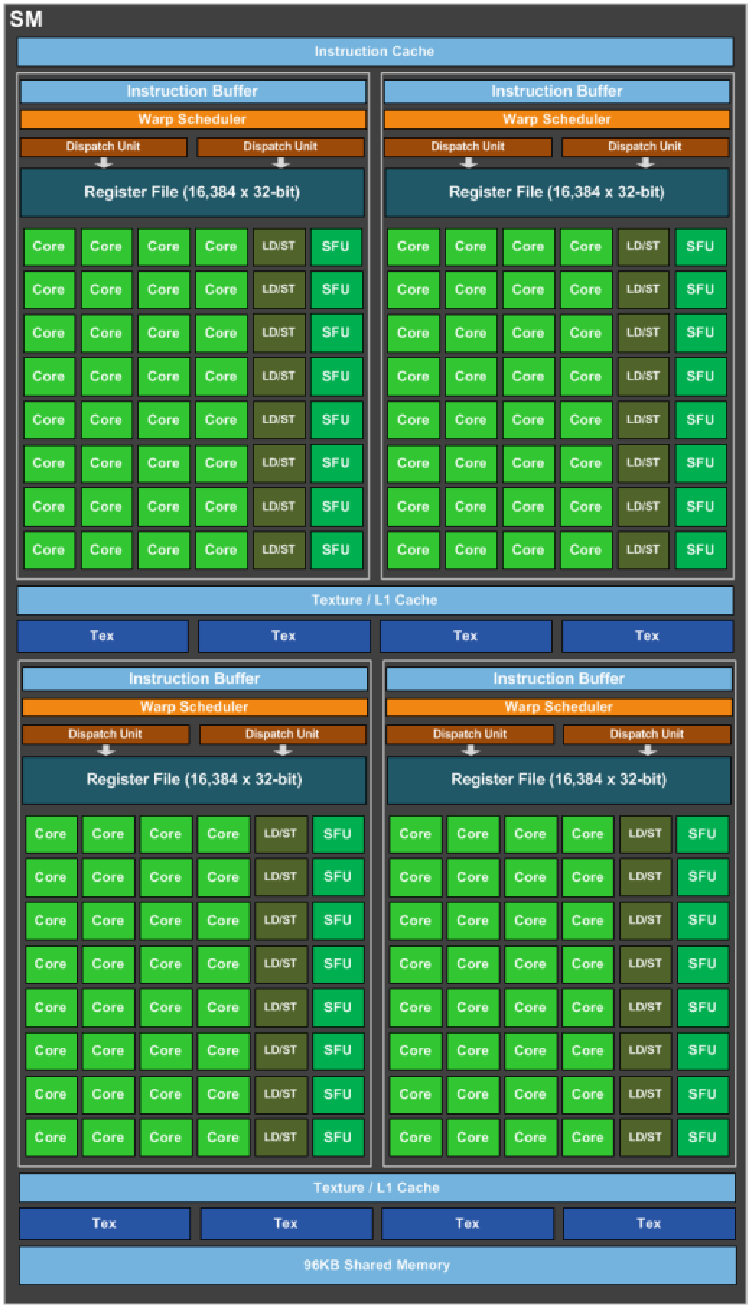


Figure 5: GP104 SM Block Diagram

To leverage the capabilities of the massively parallel GPU architecture, a unique programming model is employed. CUDA extends C/C++ by allowing programmers to define functions, which run on the GPU, called kernels. Execution threads are organised in a 3-dimensional space in order to make vector, matrix, and tensor indexing easier. They are grouped together into a Thread Block which is expected to reside in an SM. A Thread Block may contain up to 1024 threads. Thread Blocks are organised into a 3-dimensional space called a Grid. They are required to execute independently and it must be possible to execute them in any order. As threads within a block are expected to reside in a single SM, they can cooperate by sharing data through Shared Memory, which shares the same lifetime as the block, and by blocking their execution until all threads have completed a set of instructions.

When a CUDA program on the host CPU invokes a Kernel Grid, the blocks of the Grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a Thread Block execute concurrently on one multiprocessor, and multiple Thread Blocks can execute concurrently on one multiprocessor. As Thread Blocks terminate, new blocks are launched on the vacated multiprocessors.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called Warps. Individual threads composing a Warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

When a multiprocessor is given one or more Thread Blocks to execute, it partitions them into Warps and each Warp gets scheduled by a Warp Scheduler for execution. The way a block is partitioned into Warps is always the same; each Warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

A Warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a Warp agree on their execution path. If threads of a Warp diverge via a data-dependent conditional branch, the Warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a Warp; different Warps execute independently regardless of whether they are executing common or disjoint code paths.

SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads.

The execution context (program counters, registers, etc.) for each Warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the Warp. Therefore, switching from one execution context to another has no cost, and at every instruction issue time, a Warp Scheduler selects a Warp that has threads ready to execute its next instruction and issues the instruction to those threads.

Finally, it is important to note that the number of blocks, and by extensions Warps, that can reside and be processed together on the multiprocessor for a given kernel, depends on the number of registers and shared memory used by the kernel and the amount of registers and Shared Memory available on the multiprocessor.

3
VGG16 Implementation on GPU

3.1) Convolution Implementation

I implemented, the Winograd $F(4 \times 4, 3 \times 3)$ algorithm discussed above, on the GPU. To do this, I used C++ and Nvidia's CUDA 11 extension. I will focus mostly on the matrix multiplication part of the Winograd convolution as, being compute heavy, it can be thoroughly optimized. The input, weight and output transforms are all memory bound which makes them less interesting from an optimization standpoint.

3.1.1) Batched Matrix Multiply

The following analysis follows the optimization steps I took to accelerate Matrix Multiplication which is of utmost importance for accelerating convolutional neural networks. Performance can vary depending on the size of the arrays involved in Matrix Multiplication. Since, matrix multiplication in the target neural network is done with relatively small arrays, I thought best to center my optimization efforts around the case of batched square 256×256 matrix multiplication. This size is indicative of matrix multiplications that appear in the target neural network. I should also note that 32 bit floating point datatypes are used for arithmetic operations in the following analysis.

I break up the optimization steps into 4 stages. Each stage provides a sizable improvement in the execution time achieved in the previous stage and explains the way of thinking one would employ in order to optimize other similar computation intensive problems.

3.1.1.1) CPU Batched Matrix Multiply Implementation

It is important for understanding the following sections to have a grasp of the underlying algorithm for batched matrix multiplication. The code presented in Figure 6 serves this purpose. The dimensions of the arrays taking part in this computation are shown as a comment on top of the code presented. Keep in mind that this is unoptimised single threaded code which does not use vector instructions (AVX) so it is not indicative of a high performance CPU implementation. It is solely used for understanding the computation steps, verifying the GPU's output and to gauge any difference in numerical accuracy.

```
// V WIN_TILExBATCH_SIZExTILESxCHANNELS
// U WIN_TILExCHANNELSxFILTERS
// out WIN_TILExBATCH_SIZExTILESxFILTERS
template<class T>
void mmCPU(const T* __restrict__ U, const T* __restrict__ V, T* output){
   for(int n=0; n<BATCH_SIZE; n++){
       for(int t=0; t<V_DIM; t++){
           for(int f=0; f<U_DIM; f++){
               for(int i=0; i<WIN_TILE; i++){
                   T temp=0;
                   for(int c=0; c<ITER_DIM; c++){
                       int V_idx = i*ITER_DIM*V_DIM*BATCH_SIZE + n*V_DIM*ITER_DIM + t*ITER_DIM + c;
                       int U_idx = i*ITER_DIM*U_DIM + c*U_DIM + f;

                       temp += V[V_idx] * U[U_idx];
                   }
                   output[i*U_DIM*V_DIM*BATCH_SIZE + n*U_DIM*V_DIM + t*U_DIM + f] = temp;
               }
           }
       }
   }
}
```

Figure 6: Simple CPU Matrix Multiplication

3.1.1.2) Nvidia Batched Matrix Multiply Implementation

Nvidia provides its internally developed closed source Linear Algebra library called CUBLAS. To extract as much performance as possible CUBLAS is written mainly in assembly. The CUBLAS library will be used as a benchmark for the below analysis.

Code for realizing the above CPU code on the GPU using CUBLAS can be found in Figure 7. Due to CUBLAS using row-major storage format, instead of C/C++ column-major, calling the required function properly requires great care.

```
void mmNVIDIA(const float* V, const float* U, float* output, const int BATCH_SIZE, const int
V_DIM, const int U_DIM, const int ITER_DIM){
   cublasHandle_t handle;
   cublasCreate(&handle);

   cublasStatus_t status;

   float alpha = 1.0f;
   float beta = 0.0f;


   status = cublasSgemmStridedBatched( handle,
                              CUBLAS_OP_N,
                              CUBLAS_OP_N,
                              U_DIM, V_DIM, ITER_DIM,
                              &alpha,
                              U, U_DIM,
                              U_DIM*ITER_DIM,
                              V, ITER_DIM,
                              V_DIM*ITER_DIM,
                              &beta,
                              output, ITER_DIM,
                              V_DIM*U_DIM,
                              BATCH_SIZE);
   if (status != CUBLAS_STATUS_SUCCESS) printf("mm failed\n");
```

```
    cublasDestroy(handle);
 }
```

Figure 7: Calling CUBLAS Strided Btached SGEMM function

Table 1, it can be seen how Nvidia's implementation performs on Batched Matrix Multiplication of 256×256 matrices. Due to the way GPUs work, the same kernel does not always execute in the same amount of time. Thus, when presenting metrics, the average of many runs is presented in order to pinpoint the actual real world performance.

Table 1: Nvidia Strided Batched SGEMM metrics

Execution Time	5.755ms
FLOP efficiency	73.58%
Achieved Occupancy	0.249
Device Memory Read Throughput	59.123 GB/s
Device Memory Write Throughput	52.9 GB/s
Executed IPC	3.54
L2 Hit Rate (Reads)	71.93%
L2 Throughput (Reads)	211.19 GB/s
Shared Memory Load Throughput	1772.7 GB/s
Shared Memory Store Throughput	318.33 GB/s
Global Memory Load Efficiency	100%
Global Memory Store Efficiency	100%

The most important metric is execution time. However, there is also a subset of other useful metrics, provided by Nvidia's profiler tool and depicted in Table 1, which can be used to infer where performance bottlenecks lie.

As can be seen, CUBLAS Strided Batched SGEMM takes on average 5.755ms to execute and that is the time to beat.

3.1.1.3) Custom Batched Matrix Multiply Implementation

Before beginning the analysis, I should note that the code in the following section is written in C++, only because Nvidia does not make available an assembler. The Matrix Multiplication operation is used extensively throughout neural networks and as such should be optimized as much as possible by bypassing the compiler which does not output optimal assembly code.

Stage 1

In Figure 8, an extremely basic implementation of matrix multiply is presented which will be used as a basis for further optimization. I decided on employing 256 threads per thread block for this stage. In the code presented, each thread computes a single element of the output matrix by multiplying the ITER dimension of matrix V with that of matrix U and adding up the results. Performance of this kernel is not good. Metrics can be seen in Table 2.

```
__global__ void mmGPU1(float* V, float* U, float* output){
   int V_read_idx = blockIdx.z*BATCH_SIZE*V_DIM*ITER_DIM + blockIdx.x*blockDim.x*ITER_DIM
                  + threadIdx.x*ITER_DIM;

   int U_read_idx = blockIdx.z*U_DIM*ITER_DIM + blockIdx.y;

   int out_idx = blockIdx.z*BATCH_SIZE*V_DIM*U_DIM + blockIdx.x*blockDim.x*U_DIM +
threadIdx.x*U_DIM
                  + blockIdx.y;

   float temp = 0;
   for(int i=0; i<ITER_DIM; i++){
       temp += V[V_read_idx + i] * U[U_read_idx + i*U_DIM];
   }
   output[out_idx] = temp;
}
```

Figure 8: Stage 1 Kernel

Table 2: Stage 1 Kernel Metrics

Execution Time	1.218s
FLOP efficiency	0.32%
Achieved Occupancy	0.6
Device Memory Read Throughput	89 GB/s
Device Memory Write Throughput	3.36 GB/s
Executed IPC	0.05
L2 Hit Rate (Reads)	77.09%
L2 Throughput (Reads)	387.02 GB/s
Shared Memory Load Throughput	0 GB/s
Shared Memory Store Throughput	0 GB/s
Global Memory Load Efficiency	12.5%
Global Memory Store Efficiency	12.35%
Requested Global Load Throughput	61.041 GB/s
Global Load Throughput	488.33 GB/s

This kernel is over 200 times slower than Nvidia's implementation. There are multiple reasons why this is so, but the most important one is lack of memory coalescing which can be deduced by the fact that Global Memory Efficiency metrics are below 100%.

Most modern Nvidia GPUs cache VRAM accesses in L2 cache which can service transactions of a minimum of 32 bytes. In addition, a warp can coalesce its threads memory accesses into fewer transactions depending on word size and the pattern of memory accesses. Thus, if a 4 byte (float32) memory access generates a 32 byte memory transaction, throughput is divided by 8. Which is exactly what happens in the above code and can be seen by looking at the difference between Requested Global Throughput vs Global Throughput. If global memory accesses within a warp are in blocks of at least 32 sequential bytes, then Global Memory Efficiency will be at 100%.

Stage 2

During the second optimization stage I will focus mainly on basic memory access optimizations. Namely:

- Memory Coalescing
- Shared Memory Buffering

In order to coalesce memory accesses, I decided on reading batches of 64 sequential bytes from both arrays into shared memory, which will serve as a buffer for reads and writes. To do this, I used 16 sequential numbered threads from the same warp in order to batch memory transaction requests.

As every warp in the same thread block has access to the same address space of shared memory, care must be taken, when using shared memory, to avoid any race conditions, caused by reading/writing data that has yet to be written/read by other warps in the same thread block. To this end, thread execution is synced where appropriate.

Because 16 elements of U's U_DIM dimension are read while the number of threads per thread block are kept the same as before (256), each thread will compute 16 elements of the output matrix. This is an added bonus of memory coalescing and in combination with buffering the inputs, has the effect of increasing compute-to-memory ratio.

The Implementation of the aforementioned improvements is presented below.

```
template<const int THREADS_X, const int SMEM_WIDTH> //THREADS_X = 256, SMEM_WIDTH = 16
__global__ void mmGPU2(float* V, float* U, float* output){
   int V_read_idx = blockIdx.z*BATCH_SIZE*V_DIM*ITER_DIM + blockIdx.x*THREADS_X*ITER_DIM
                   + (threadIdx.x/SMEM_WIDTH)*ITER_DIM + (threadIdx.x%SMEM_WIDTH); //memory coalescing

   int U_read_idx = blockIdx.z*U_DIM*ITER_DIM + blockIdx.y*SMEM_WIDTH
                   + (threadIdx.x/SMEM_WIDTH)*U_DIM + (threadIdx.x%SMEM_WIDTH); //memory coalescing

   int out_idx = blockIdx.z*BATCH_SIZE*V_DIM*U_DIM + blockIdx.x*THREADS_X*U_DIM + blockIdx.y*SMEM_WIDTH
                    + (threadIdx.x/SMEM_WIDTH)*U_DIM + threadIdx.x%SMEM_WIDTH; //memory coalescing

   __shared__ float V_smem[THREADS_X*SMEM_WIDTH];
   __shared__ float U_smem[THREADS_X*SMEM_WIDTH];

   for(int i=0; i<THREADS_X/SMEM_WIDTH; i++){
       int U_smem_wr_idx = (threadIdx.x/SMEM_WIDTH) + (threadIdx.x%SMEM_WIDTH)*THREADS_X
                                                                     + i*(THREADS_X/SMEM_WIDTH);

       U_smem[U_smem_wr_idx] = U[U_read_idx + i*(THREADS_X/SMEM_WIDTH)*U_DIM]; //load U to shared memory
   }

   float temp[SMEM_WIDTH] = {0};

   for(int n=0; n<ITER_DIM/SMEM_WIDTH; n++){

       for(int k=0; k<THREADS_X/SMEM_WIDTH; k++){
           int V_smem_rd_idx = V_read_idx + k*(THREADS_X/SMEM_WIDTH)*ITER_DIM + n*SMEM_WIDTH;

           V_smem[threadIdx.x + k*THREADS_X] = V[V_smem_rd_idx]; //load V to shared memory
       }

       __syncthreads(); //wait all threads to finish writing to smem

       for(int j=0; j<SMEM_WIDTH; j++){
           for(int i=0; i<SMEM_WIDTH; i++){ //compute
               temp[j] += V_smem[threadIdx.x*SMEM_WIDTH + i] * U_smem[i + j*THREADS_X + n*SMEM_WIDTH];
           }
       }

       __syncthreads(); //wait all threads to finish reading from smem
   }

   for(int i=0; i<SMEM_WIDTH; i++){
       V_smem[threadIdx.x*SMEM_WIDTH + i] = temp[i]; //buffer output
   }

   __syncthreads(); //wait all threads to finish reading from smem

   for(int i=0; i<THREADS_X/SMEM_WIDTH; i++){
       output[out_idx + i*(THREADS_X/SMEM_WIDTH)*U_DIM] = V_smem[threadIdx.x + i*THREADS_X]; //write out
   }
}
```

Figure 9: Stage 2 Kernel

Table 3: Stage 2 Kernel Metrics

Execution Time	29.7ms
FLOP efficiency	12.94%
Achieved Occupancy	0.3736
Device Memory Read Throughput	88.635 GB/s
Device Memory Write Throughput	9.49 GB/s

Executed IPC	0.76
L2 Hit Rate (Reads)	45.47%
L2 Throughput (Reads)	163.28 GB/s
Shared Memory Load Throughput	1824 GB/s
Shared Memory Store Throughput	341 GB/s
Global Memory Load Efficiency	100%
Global Memory Store Efficiency	98%
Requested Global Load Throughput	161.05 GB/s
Global Load Throughput	161.05 GB/s

As expected, there is a substantial improvement (1.218s vs 29.7ms) in execution time but performance isn't in the same ballpark as Nvidia's implementation.

Stage 3

There are some further improvements that could be made on the above code to possibly half the execution time but there is a fundamental problem with using inner products to compute matrix multiplication on the GPU. The number of floating point operations needed to multiply 2 matrices is O(N3) yet the amount of data needed is O(N2) which results in high compute intensity. In order to take advantage of that, very large caches would be needed to avoid reading the same elements repeatedly from slow external memory.

A solution to this problem is to use outer products instead. This way, each column of V and row of U needs to be loaded only once, for computing their outer product and accumulating the result in the output matrix. A weak point of this method is that the output matrix would need to be stored in a large and at the same time fast cache to avoid going out to external memory. Because such large caches do not exist in modern GPUs, the output matrix needs to be tiled by performing tiled matrix multiply as shown in Figure 31.

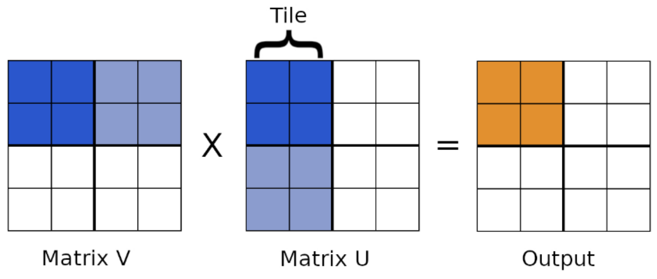


Figure 10: Tiled Matrix Multiplication

As we are dealing with relatively small matrix sizes, which mostly fit in the L2 cache of the GP104, I decided to use tiles of size $64x64$, compared to Nvidia's 128×128, in order to increase L2 hit rate and thus reduce average memory access latency, while at the same time substantially limiting the required memory bandwidth relative to the non-tiled version. The memory bandwidth required for the reads by the 64x64 version can be approximated as follows:

- Each thread loads 64 bytes per loop
- Each loop requires 512 FMA instructions plus miscellaneous instructions (≈ 550 clocks per loop)
- There are 2560 cores clocked at 1607 MHz
- $(2560 \times 1607 \times 10^6 \times 64)/550 \ \approx \ 446\ GB/s$

A GP104 offers a peak memory bandwidth of only 320 GB/s which is further reduced to around 260 GB/s due to read-write turnaround overhead of the GDDR5X memory. Fortunately, the L2 cache will service most of the needed bandwidth and the code won't be memory bottlenecked.

Another way to increase performance is to maximise the ratio of FMA instructions by using wider load/store instructions to transfer more data per instruction. Thus, instead of loading each float one by one, 4 floats are loaded with one instruction. The benefit to this is threefold. There are fewer load/store instructions to execute, memory latency decreases and memory bandwidth increases. The downside is that data must be quad aligned which practically means that the size of the array data is loaded from must be a multiple of 4.

Last but not least, it is imperative to make use of register blocking to facilitate data reuse. I decided on loading 8 elements of V and 8 of U.

To summarize, the optimization performed in Stage 3 and shown in code in Figure 11 are the following:

- ❖ Switch from Inner Products to Outer Products.
- ❖ Perform Matrix Multiplication with the use of Tiling.
- ❖ Choose a small tile size to maximise L2 hit rate.
- ❖ Use wide load/store instructions.
- ❖ Perform Register Blocking.

```
template<const int THREADS_X,const int SMEM_V_X,const int SMEM_V_Y,const int SMEM_U_Y,const int
SMEM_U_X,const int REG_V,const int REG_U>
__global__ void mmGPU4(float* V, float* U, float* output){
   int V_base_idx = blockIdx.z*V_DIM*ITER_DIM/4 + blockIdx.x*(SMEM_V_Y*ITER_DIM/4)
                  + (threadIdx.x/(SMEM_V_X/4))*(ITER_DIM/4) + (threadIdx.x%(SMEM_V_X/4));

   int U_base_idx = (blockIdx.z/BATCH_SIZE)*U_DIM*ITER_DIM/4 + blockIdx.y*(SMEM_U_X/4)
                  + (threadIdx.x/(SMEM_U_X/4))*(U_DIM/4) + threadIdx.x%(SMEM_U_X/4);

   int out_base_idx = blockIdx.z*V_DIM*U_DIM/4 + blockIdx.x*(SMEM_V_Y*U_DIM/4) + blockIdx.y*(SMEM_U_X/4)
                  + (threadIdx.x/8)*(U_DIM/4)*8 + (threadIdx.x%8);

   __shared__ float V_smem[SMEM_V_X*SMEM_V_Y];
   __shared__ float U_smem[SMEM_U_X*SMEM_U_Y];

   float result_thread[REG_V*REG_U] = {0}; //initialise result buffer
   float V_reg[REG_V];
   float U_reg[REG_U];
   float U_temp[(SMEM_U_X*SMEM_U_Y)/THREADS_X];
   float V_temp[(SMEM_V_X*SMEM_V_Y)/THREADS_X];

   //fetch 64x8 tiles from external memory until 256/8 tiles have been fetched
   for(int n_gmem=0; n_gmem<(ITER_DIM + SMEM_V_X - 1) / SMEM_V_X; n_gmem++){
       //load U to registers
       for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X)/4; i++){
           int U_idx = U_base_idx + i*(THREADS_X/(SMEM_U_X/4))*(U_DIM/4) + n_gmem*SMEM_U_Y*(U_DIM/4);

           reinterpret_cast<float4*>(U_temp)[i] = reinterpret_cast<float4*>(U)[U_idx]; //wider load
       }
       //store U to shared memory
       for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X)/4; i++){
           int idx = threadIdx.x + i*THREADS_X;

           reinterpret_cast<float4*>(U_smem)[idx] = reinterpret_cast<float4*>(U_temp)[i]; //wider load
       }
```

```
        //load V to registers
        for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X)/4; i++){
            int V_idx = V_base_idx + i*(THREADS_X/(SMEM_V_X/4))*(ITER_DIM/4) + n_gmem*SMEM_V_X/4;

            reinterpret_cast<float4*>(V_temp)[i] = reinterpret_cast<float4*>(V)[V_idx]; //wider load
        }
        //store V to shared memory
        for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X)/4; i++){
            int idx = threadIdx.x + i*THREADS_X;

            reinterpret_cast<float4*>(V_smem)[idx] = reinterpret_cast<float4*>(V_temp)[i]; //wider load
        }

        __syncthreads(); // wait for threads to stop writing to shared memory

        //fetch 64x1 tiles from shared memory until 8/1 tiles have been fetched
        for(int n_smem=0; n_smem<SMEM_U_Y; n_smem++){
            //load V data from shared memory to registers (register blocking)
            for(int i=0; i<REG_V; i++){
                V_reg[i] = V_smem[ (threadIdx.x/8)*SMEM_V_X*8 + i*SMEM_V_X + n_smem];
            }
            //load U data from shared memory to registers (register blocking)
            for(int i=0; i<REG_U/4; i++){
                int idx = (threadIdx.x%8) + i*8 + n_smem*8*2;
                reinterpret_cast<float4*>(U_reg)[i] = reinterpret_cast<float4*>(U_smem)[idx]; //wide load
            }
            //compute outer products
            for(int i=0; i<REG_V; i++){
                for(int j=0; j<REG_U; j++){
                    result_thread[i*REG_U + j] += V_reg[i] * U_reg[j];
                }
            }
        }

        __syncthreads(); //wait for threads to stop reading from shared memory
    }

    //write output to VRAM
    for(int i=0; i<REG_V; i++){
        for(int j=0; j<REG_U/4; j++){
            int out_final_idx = out_base_idx + i*U_DIM/4 + j*8;
            reinterpret_cast<float4*>(output)[out_final_idx] =
                                                reinterpret_cast<float4*>(result_thread)[i*REG_U/4 + j];
//wider store
        }
    }
}
```

Figure 11: Stage 3 Kernel

Table 4: Stage 3 Kernel Metrics

Execution Time	7.52ms
FLOP efficiency	58.22%
Achieved Occupancy	0.2476
Device Memory Read Throughput	44.627 GB/s
Device Memory Write Throughput	41.28 GB/s
Executed IPC	2.835
L2 Hit Rate (Reads)	86.56%
L2 Throughput (Reads)	329.14 GB/s

Shared Memory Load Throughput	3304.8 GB/s
Shared Memory Store Throughput	330.48 GB/s

As can be seen from Table 4, average execution time dropped to 7.5ms which constitutes a substantial decrease (4X) relative to the previous stage. Metrics show total external memory throughput at 86 GB/s, well below the 270 GB/s limit. L2 cache read hit rate is at 86.6% and total L2 read throughput is at 330 GB/s. Taking into account the 45 GB/s bandwidth provided by the external memory, it becomes apparent that the L2 cache provides 285 GB/s or, 86.3%, of the required read bandwidth, which confirms the previous assumptions.

Note that 330 GB/s is well below the 446 GB/s required for close to peak performance and also that FLOP efficiency sits at around 58%. Since the code doesn't seem to be bandwidth or compute limited, I deduce that memory hierarchy latency is the limiting factor.

Stage 4

In order to extract the remaining performance, a more methodological approach must be taken.

There is no point in having thousands of compute cores unless they can be fed them data to process. As already discussed, matrix multiplication is a relatively compute heavy workload and to take advantage of that all stages of the memory hierarchy need to be leveraged to their fullest. In addition, how data travels in the GPU needs to be considered.

$$VRAM \rightarrow L2 \rightarrow Texture\ Cache \rightarrow Registers \rightarrow Shared\ Memory \rightarrow Registers \rightarrow VRAM$$

Figure 12: Data Travel Paths in the GPU

The paths shown in Figure 12 all have different access latencies that need to be hidden to maximise performance. Average VRAM memory latency is around 600 clocks, L2 latency is around 100 clocks and shared memory latency seems to hover around 20-30 clocks. The GPU, by using fast context switching between threads, can in part hide these latencies, provided that occupancy is high, but extra programming effort is required to completely hide them. To achieve that goal, a double buffering scheme can be used to bring data in from VRAM to shared memory and from shared memory to registers. This also enables the removal of a synchronization instruction in the main loop for a small extra performance uplift.

As shared memory is partitioned in 32 banks, extra care must be taken to ensure that there are no bank conflicts between threads of a warp when accessing shared memory as these lead to increased access latency. A bank conflict will occur when 2 or more threads in the same warp access shared memory addresses that their difference is a multiple of 32 ($addr1\ mod\ 32\ ==\ addr2\ mod\ 32$). Fortunately, if 2 or more threads try to access the same shared memory address, a bank conflict will not occur as the data will be broadcast to all eligible threads.

In each iteration of the main loop, a $64x8$ tile is fetched from VRAM and stored into shared memory. Since tiles will be fetched from both V and U arrays the required shared memory for each thread block will be 4KB. Each block will consist of 64 threads. Ideally, at least 512 threads need to run per SM to keep GPU occupancy high in order to hide latencies. Thus, 8 thread blocks per SM need to be scheduled which puts the required shared memory at 32KB. As mentioned, the reads will be double buffered, so a minimum of 64KB is required. I pad the shared memory wherever required to avoid bank conflicts.

The 512 threads per SM constraint allows for a budget of 128 registers per thread considering each SM provides 65536 registers. An analysis on the number of required registers is hereafter performed.

Since the tile size is $64x64$ and 64 threads per thread block are used, each thread needs 64 registers to store the output tile. Each iteration, 2 $64x8$ tiles will be fetched by using 64 threads so 16 additional registers are required. As in previous iterations of the code, register blocking will be used. Usage of 8 registers for V and 8 for U, add up to another 32 registers with double buffering which makes for a total of 96. Of course, more registers will be needed for storing and calculating addresses etc. but hopefully 32 will be enough.

Unfortunately, I was not able to double buffer the shared memory to register reads. I tried to write the code in different ways but the compiler wouldn't have it, opting instead to 'optimise' away the 'extra' code.

Another thing to be mindful of is the instruction cache. Specifically, its size which if exceeded can increase instruction fetch latency dramatically. Nvidia does not provide any details on the size of the instruction cache so by performing some tests and dissecting assembly code and binaries, I estimate its size to be 8KB. Instruction size stands at 8 bytes. Every 3 instructions there is an 8 byte control code injected that needs to be taken into account as well. Thus, an unrolled loop might contain at most 768 instructions in order to fit in the instruction cache. So, I leave the main loop rolled while unrolling all the inner loops.

Ideally, V would be transposed but to do this explicitly would be costly. So, in previous iterations, data was read from V by using quad loads which forced the usage of normal loads to read data from shared memory. This is not ideal for a couple of reasons. First, more instructions are used as for every loop each thread loads from VRAM 8 floats but reads from shared memory 64 floats. In addition, this part of the computation is extremely latency sensitive because, as discussed above, I was not able to double buffer the shared memory reads. So, I decided to load single floats from VRAM while loading quad floats from shared memory, which appears to provide a sizable performance boost.

Another trick that I should mention is the use of templates and/or global const variables for constant values. Instead of storing constant values to VRAM and fetching them when they are needed, which takes up bandwidth and more importantly registers, immediate instructions can be used to integrate the constant values into the instructions, by telling the compiler that these variables are compile time constants. In addition, the GPU uses special registers to store common values such as threadIdx.x, blockDim.x etc. While this is helpful, a read from these special registers requires 12+ clocks. Thus, by replacing, for example, blockDim.x, with the constant variable THREADS_X, this issue can be avoided. These optimizations can increase performance by up to 20%.

Finally, the launch_bounds directive informs the compiler of the number of threads per block and the minimum number of blocks one wants to run on an SM. As mentioned above, ideally, 8 blocks should be running on an SM. However, if setting the launch_bounds accordingly, performance is degraded due to aggressive register reduction by the compiler. Setting the blocks to run on an SM equal to 1 circumvents this problem but makes it possible for the compiler to use more registers than we would like. Thus, experimentation with the launch_bounds directive is required.

To summarize, the following optimization steps are applied in stage 4:

- ❖ Avoid Shared Memory Bank Conflicts.
- ❖ Double Buffer VRAM to Shared Memory Reads.
- ❖ Unroll Loops.
- ❖ Use Quad Loads on Shared Memory instead of VRAM.
- ❖ Use Templates to force Compile Time Constants.
- ❖ Experiment with launch_bounds directive.

As code size has more than doubled compared to Stage 3, the full code of Stage 4 is presented in the Appendix.

Table 5: Stage 4 Kernel Metrics

Execution Time	5.365ms
FLOP efficiency	85.45%
Achieved Occupancy	0.246
Device Memory Read Throughput	67.582 GB/s
Device Memory Write Throughput	56.16 GB/s
Executed IPC	3.957
L2 Hit Rate (Reads)	85.05%
L2 Throughput (Reads)	449.58 GB/s
Shared Memory Load Throughput	2696.6 GB/s
Shared Memory Store Throughput	449.43 GB/s

As can be seen by the performance results presented in Table 5, these improvements are enough to surpass Nvidia's implementation. Stage 4 ends up using 124 registers which is below the 128 limit previously set. The code makes use of almost 86% of the peak compute performance of the GPU. Executed IPC is close to 4 which is the max throughput of FP32 arithmetic instructions. 452 GB/s of L2 read throughput is needed to achieve this level of performance, very close to the 446 GB/s hypothesized earlier.

By dissecting the assembly code produced by the compiler, 75 non FMA instructions are found in the main loop for a total of 587 instructions. Because of the dual dispatch units available to each warp scheduler, 45 instructions in the main loop are dual issued, effectively making for a total of 542 instructions. So, 94.5% of the 'effective' instructions are FMA instructions which suggests that there is still some performance left on the table.

If Nvidia made available an assembler and this code were written in assembly, double buffering the shared memory loads should be possible. By doing so, I speculate that 95% of peak compute performance could be reached.

It should be noted that no checks for out of bounds reads/writes have been set, which will happen if inputs are not multiples of 64. Setting them by using if statements before every read/write is trivial. The interesting part here is that in order to avoid branches and divergent threads, the GPU uses predicated instructions. These can be conceptually thought of as instructions which are executed normally but the read/write step is never performed if the predicates(if statements) evaluate to false. Unfortunately, the compiler does a poor job of handling registers when close to the register limit (128) set through the Launch Bounds directive. Even though there are 7 separate predicate registers per thread, and that limit is not surpassed, the compiler, in some cases, cannot keep registers below the 128 mark which results in having 6 active blocks per SM. Fortunately, this only reduces performance by about 3%.

Section Summary

In this section, the optimization procedure of the batched matrix multiply operation was presented. The same general procedure can be applied to other, similarly compute heavy, problems to reach high level of compute utilization on the GPU.

The main takeaway from this chapter should be that it does not matter if the hardware is capable of great peak compute numbers if the memory hierarchy is not properly utilized to provide the compute units with data fast enough. In stage 2 it was shown that just 2 very basic optimizations decreased execution time exponentially. In stage 3, it was shown that sometimes, it is crucial to reformulate the problem to make it fit into the constraints the hardware sets. In stage 4, it was shown that great care must be taken to understand all the details of the underlying architecture, the compiler and the programming language used. It is through this iterative process that great performance results were achieved.

For reference, a table providing metrics from all stages is presented below.

Table 6: Metrics from every Stage + Nvidia

Stages	**Stage 1**	**Stage 2**	**Stage 3**	**Stage 4**	**Nvidia**
Execution Time	1.218s	29.7ms	7.52ms	5.365ms	5.755ms
FLOP efficiency	0.32%	12.94%	58.22%	85.45%	73.58%
Achieved Occupancy	0.6	0.3736	0.2476	0.246	0.249
Device Memory Read Throughput	89 GB/s	88.635	44.627	67.582	59.123
Device Memory Write Throughput	3.36 GB/s	9.49 GB/s	41.28 GB/s	56.16 GB/s	52.9 GB/s
Executed IPC	0.05	0.76	2.835	3.957	3.54
L2 Hit Rate (Reads)	77.09%	45.47%	86.56%	85.05%	71.93%
L2 Throughput (Reads)	387 GB/s	163 GB/s	329 GB/s	449 GB/s	211 GB/s
Shared Memory Load Throughput	0 GB/s	1824 GB/s	3304 GB/s	2696 GB/s	1772 GB/s
Shared Memory Store Throughput	0 GB/s	341 GB/s	330 GB/s	449 GB/s	318 GB/s
Global Memory Load Efficiency	12.5%	100%	100%	100%	100%
Global Memory Store Efficiency	12.35%	98%	100%	100%	100%

3.1.2) Padding

Returning to convolution implementation details, I thought it prudent to mention a few things about padding as it is an important operation which is seldom mentioned.

First, I should note that all border padding is done implicitly.

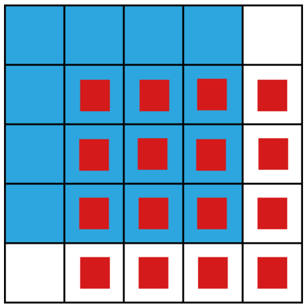


Figure 13: Pixels in blue denote the output of the output transform while pixels with a red square denote the pixels that should be inputs to the Max-Pool layer

As mentioned in the Basic Background chapter, to keep the output dimension the same as the input's, the input is padded on its border with zeros. However, the Winograd Convolution algorithm introduces a further need for padding the input. Take for example the 3 last layers of VGG16. They have spatial dimensions of 14×14. Noting that the Winograd Convolution works on overlapping 6×6 tiles, notice that an input of 18×18 is required. This suggests that there is need for extra padding due to the nature of the Winograd Algorithm. An input of 18×18 translates to an output of 16×16 which is larger than 14×14. Thus, the outer border must be discarded. However, as the RELU and, more importantly, the Max-Pool layer are fused with the output transform, a significant problem appears. The output transform produces 4×4 tiles which is a perfect fit for the Max-Pool layer which works on 2×2 tiles. Unfortunately, during the last convolutional layers of VGG, the borders should be discarded which results in a tile shift of sorts that is better explained by looking at Figure 34. The blue pixels are the output of the output transform while the ones that are marked red should be the input to the Max-Pool Layer. As each thread computes an output tile, this proves to be problematic. To solve this problem, I also compute, when necessary, the white pixels which are parts of 3 different output tiles. This, in theory, results in almost triple the memory bandwidth required to compute the output transform which is already memory bound. Fortunately, most of it is serviced by the L2 cache which results in just a 5% performance hit. The same problem appears when backpropagating on the input transform.

3.2) General Optimizations

Before proceeding to general optimizations, it is important to showcase the algorithm for computing the whole network. In Figure 35, a flowchart is presented which shows the general steps of the training algorithm which includes both Inference and Backpropagation.

Apart from the matrix multiply algorithm presented above, multiple matrix multiply kernels have been written for different array transpositions by following the same optimization procedure. This is necessary for achieving good overall performance when computing any Neural Network as otherwise matrices would need to be explicitly transposed which takes up extra memory bandwidth on an already memory bound implementation. To this end, I have written 8 different matrix multiply kernels. 3 are used for the convolutional layers, another 3 are optimized specifically for the Fully Connected layers and the other 2 deal with the first convolutional layer. These 2 last kernels are especially important as I take advantage of the $Channel$ dimension being so small during the first layer and optimise for it. This results in halving of the kernel's execution time.

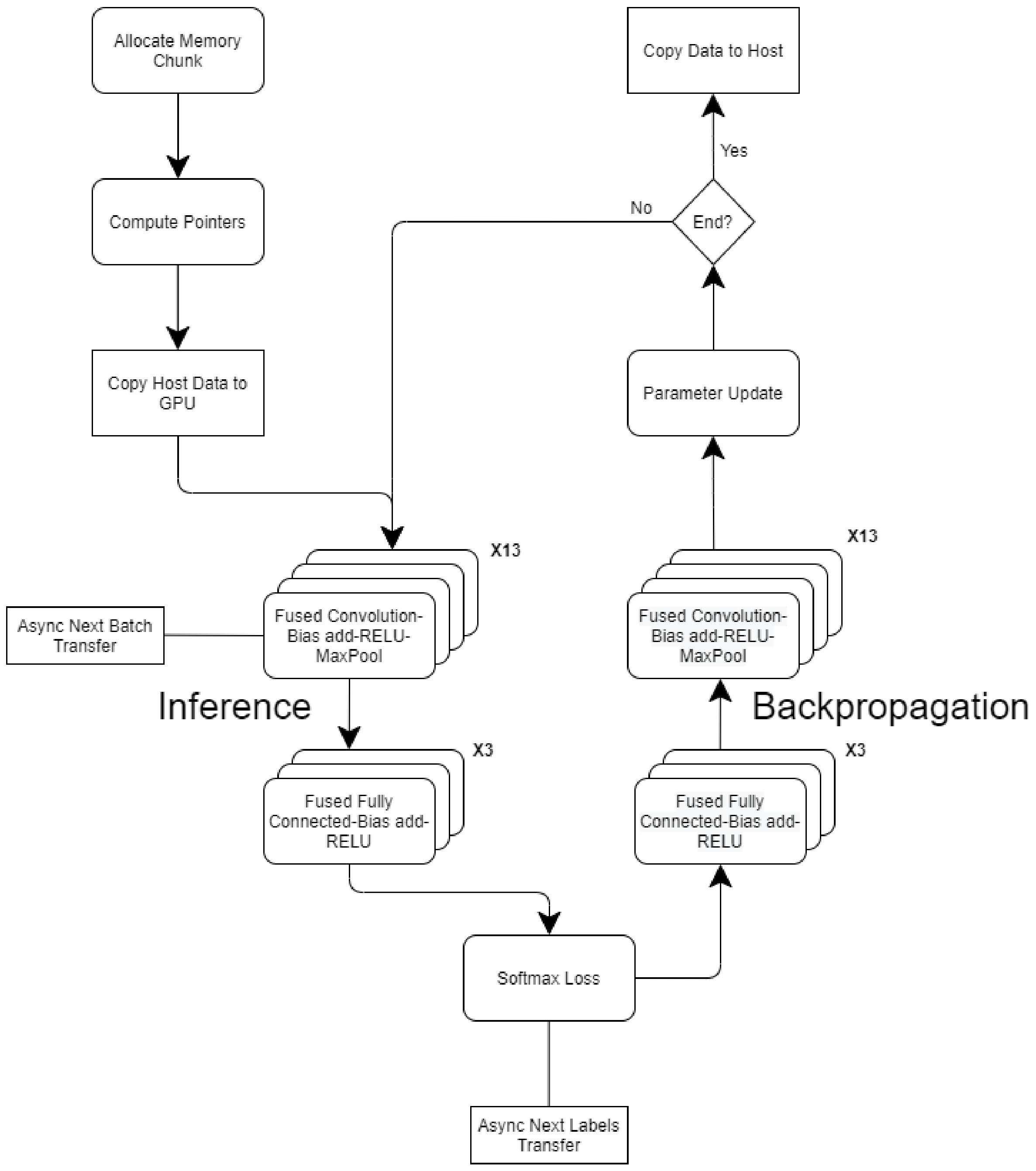

Allocate Memory Chunk
Compute Pointers
Copy Host Data to GPU
Copy Data to Host
Yes
No
End?
Parameter Update
X13
Async Next Batch Transfer
Fused Convolution-Bias add-RELU-MaxPool
X13
Fused Convolution-Bias add-RELU-MaxPool
Inference
Backpropagation
X3
Fused Fully Connected-Bias add-RELU
X3
Fused Fully Connected-Bias add-RELU
Softmax Loss
Async Next Labels Transfer

Figure 14: VGG16 GPU Flowchart

There is one last optimization that is possible on the matrix multiply kernels. So far, I have been scheduling tiles of the V_DIM dimension as blocks in the x dimension of the GPU's grid and of the $WIN\_TILES \times BATCH\_SIZE$ dimension as blocks in the z dimension of the grid. In VGG16, there are layers who have a small V_DIM dimension and a large U_DIM dimension, which is far from the square matrices I have optimized for. Since data on the V matrix has the following dimension order $(WIN\_TILES \times BATCH\_SIZE) \times (V\_DIM) \times (CHANNELS)$ some data from the z dimension can be moved to the x block dimension like so $(WIN\_TILES \times BATCH\_SIZE \,/\, Y) \times (V\_DIM \times Y) \times (CHANNELS)$ in order to make the matrix more square, get it closer to a number which is a multiple of 64 to eliminate predicated reads/writes which evaluate to false, and optimise data re-use in the L2 cache. The optimal Y value can be found quickly by trial and error. This seemingly unimportant optimization can improve performance by up to 25% in cases where there is considerable disparity between the V_DIM and U_DIM dimensions.

Layer Fusion is performed whenever the constraints of the GPU make it possible. By combining the RELU, bias addition and Max-Pool layers with the convolutional layer memory bandwidth requirements are significantly reduced. If fusion wasn't performed, each layer would have to read and write all of its data to VRAM while with layer fusion data isn't written to VRAM until all layers are computed. I estimate that this leads to a reduction of at least 20% of the overall training execution time.

By using CUDA events and streams I manage, when the GPU deems it possible, to concurrently execute multiple kernels. Up to 10% better performance is seen when executing 2 kernels concurrently than executing them in order. Concurrent kernel execution is only possible when a kernel is small enough that the GPU has resources available to schedule another kernel.

Another important optimization is to save the indices of data which pass through the RELU and max-pooling layers. By doing so, the memory footprint is reduced as there is no need to save the input and output arrays of each layer for backpropagation, while at the same time increasing computational intensity by avoiding unnecessary data transfers inside the GPU. I use char (C/C++ data type for byte size) arrays to save the indices in a one-hot format as this is the smallest data type supported by the GPU. One further optimization that could be done is to group together 8 indices for every char instead of just one to reduce memory bandwidth required when transferring the arrays. However, performance benefits of this would be marginal.

Allocating each array needed separately, leads to Out of Memory errors, possibly due to memory fragmentation. Thus, a huge chunk of memory is allocated all at once and then pointers are computed for every array. This is more efficient, as allocating lots of small arrays can take a considerable amount of time. I note that pointer computation led to a problem that was extremely hard to debug. I noticed that kernels would have up to 50% worse performance seemingly at random. I found out, and I think it is important to mention as I didn't find any reference to this in Nvidia's documentation, that when pointers to arrays weren't 256-byte aligned, there was significant performance degradation especially in memory bound kernels. I later discovered that cudaMalloc also returned pointers that are 256-byte aligned which validated my finding.

Finally, data must be continuously transferred from the CPU (Host) to the GPU and vice-versa, stalling computation. This is problematic as the transfers are done over a PCI-E 3.0 x16 bus which provides a maximum of 16GB/s bandwidth compared to the GPU's 320 GB/s and the RAM's 50GB/s. Often, when the CPU is busy, bandwidth drops below 12 GB/s. In addition, the GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or "pinned", host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory. I avoid the cost of the transfer between pageable and pinned host arrays by directly allocating host arrays in pinned memory. This alleviates the bandwidth drop often noticed when the CPU is busy. More importantly, usage of pinned memory is required for overlapping host to GPU data transfers with computation. With proper use of streams and events, I read and write data to the host concurrently with computation which has a positive effect on performance.

All these optimizations work in tandem to make it possible to achieve the significant performance gains I present in the following section.

3.3) Performance Results

Below, the execution time of each convolutional layer during Inference is compared with Nvidia's equivalent using the $F(4 \times 4, 3 \times 3)$ algorithm in both cases. Note that the NCHW data format is used for running all cuDNN kernels as it is quite a bit faster than the NHWC format.

Table 7: Execution Time Comparison of Inference Convolutional Layers

Forward Layers	**Nvidia**	**Custom (vs Nvidia)**
1st Conv Layer	9.1 ms	9.06 ms (-0.44%)
2nd Conv Layer	18 ms	16.65 ms (-7.5%)
3rd Conv Layer	7.1 ms	6.65 ms (-6.34%)
4th Conv Layer	9.9 ms	9.9 ms (same)
5th Conv Layer	4.22 ms	4.08 ms (-3.3%)
6th Conv Layer	6.74 ms	7.04 ms (+4.45%)
7th Conv Layer	6.74 ms	7.04 ms (+4.45%)
8th Conv Layer	3.2 ms	3.31 ms (+3.4%)
9th Conv Layer	5.73 ms	5.97 ms (+4.2%)
10th Conv Layer	5.73 ms	5.97 ms (+4.2%)
11th Conv Layer	1.83 ms	1.86 ms (+1.6%)
12th Conv Layer	1.83 ms	1.86 ms (+1.6%)
13th Conv Layer	1.83 ms	1.86 ms (+1.6%)
Total Time	81.95 ms	81.25 ms **(-0.86%)**

An execution time comparison for fully connected layers is presented in the table below.

Table 8: Execution Time Comparison of Inference Fully Connected Layers

Fully Connected Layers	**Nvidia**	**Custom (vs Nvidia)**
1st FC Layer	1.82 ms	1.7 ms (-6.6%)
2nd FC Layer	0.31 ms	0.29 ms (-6.45%)
3rd FC Layer	0.1 ms	0.2 ms (+100%)
Total Time	2.23 ms	2.19 ms **(-1.8%)**

It becomes apparent from the above comparisons that there is not much room for improvement during Inference. My results match Nvidia's. The good news is performance can be improved a bit by fusing layers. For example, the bias addition, the activation function and the pooling layer where applicable, are all fused together in convolution's output transform. It should be noted, that during inference, Nvidia also fuses convolution with the activation function and the bias addition but doesn't do the same with the pooling layer.

The Fully Connected layers, despite contributing 90% of the network's parameters, take up less than 3% of the network's execution time during Inference. The same behaviour can also be seen during backpropagation, which further cements the need to accelerate convolution.

I deemed it important that the custom VGG implementation is as easy to use as possible. So, I decided to integrate it into Tensorflow, which gives the ability to call the custom VGG16 implementation like any other Tensorflow function. This allows for it to be used along with all the other functionalities that Tensorflow has to offer.

In the table below, results are also presented for Tensorflow. I used a batch size of 32 in Tensorflow as well and the same SGD with momentum optimizer as the one used to train the custom network. Tensorflow 2.2 was used for the below comparisons. I should also note that Tensorflow uses Nvidia's cuDNN library for GPU acceleration. The Nvidia implementation was written using the cuDNN library and all possible optimizations were performed in order for the comparison to be as fair as possible.

Table 9: Total Inference Execution Time Comparison

Inference	**Nvidia**	**Custom (vs Nvidia)**	**Tensorflow**	**Custom TF (vs Tensorflow)**
Total Time	91.5 ms	84 ms **(-8.2%)**	116 ms	86 ms **(-25.9%)**

I achieve a small but important 8.2% reduction in execution time compared to Nvidia which is due to the layer fusion discussed above. Integrating the custom network into Tensorflow appears to add a small overhead. Despite this, a significant improvement compared to Tensorflow's implementation of VGG16 is seen. I can only speculate as to why Tensorflow appears to be so much slower than even Nvidia's implementation. It may be using non-fused layers, the heuristic algorithm, which chooses which algorithm to use to compute the different layers, might be making non optimal decisions, or the checks that

Tensorflow performs to make sure that the data has the right dimensions, is of the correct data type etc. might eat into performance.

While the per-layer results above include only the convolution or fully connected layer as the effect of layer fusion isn't very noticeable, during backpropagation I present results for the fused layers as these are indicative of the overall performance benefits.

Table 10: Execution Time Comparison of Backpropagation Layers

Backward Layers	**Nvidia**	**Custom (vs Nvidia)**
1st Conv Layer	27.6 ms	9.4 ms (-65.9%)
2nd Conv Layer	56.5 ms	26 ms (-53.9%)
3rd Conv Layer	19 ms	11.2 ms (-41.05%)
4th Conv Layer	26 ms	15.1 ms (-41.9%)
5th Conv Layer	10 ms	8 ms (-20%)
6th Conv Layer	15.6 ms	12.2 ms (-21.8%)
7th Conv Layer	16.9 ms	12 ms (-29%)
8th Conv Layer	7 ms	5.9 ms (-15.71%)
9th Conv Layer	11.7 ms	10.8 ms (-7.69%)
10th Conv Layer	12.2 ms	10.7 ms (-12.3%)
11th Conv Layer	3.9 ms	3.45 ms (-11.54%)
12th Conv Layer	3.9 ms	3.45 ms (-11.54%)
13th Conv Layer	4.2 ms	3.4 ms (-19.05%)
1st FC layer	3.75 ms	3.6 ms (-4%)
2nd FC layer	0.96 ms	0.65 ms (-32.3%)
3rd FC layer	0.4 ms	0.18 ms (-55%)
Total	219.6 ms	136 ms **(-38.06%)**

A sizable performance improvement is noticed during backpropagation. There are multiple reasons for this. First of all, Nvidia appears to recalculate the data and the weight transforms for the convolution layer, the same transforms that were computed during Inference. This reduces performance dramatically as it 'wastes' memory bandwidth. This is further validated by the fact that in the deeper layers of the network, the transforms reduce in size compared to earlier ones, which reduces the execution time difference between mine and Nvidia's implementation. I speculate that Nvidia made this decision in order to offer the flexibility to choose different algorithms for computing convolution layers during inference and backpropagation or to reduce the memory footprint. Also, Nvidia doesn't seem to fuse any of the layers during backpropagation which greatly reduces compute intensity.

It is important to note the tremendous effect the custom 3 channel matrix multiply, that was previously discussed, has in reducing execution time during the first convolution layer.

Below, I present total training time results. I have not included Nvidia in these comparisons as they do not provide optimizer or softmax functions.

Table 11: Total Training Time Comparison

Training	**Tensorflow**	**Custom (vs Tensorflow)**	**Custom TF (vs Tensorflow)**
Time/Batch	355 ms	243 ms **(-31.55%)**	252 ms **(-29%)**
Images/Second	90	132 **(+46.67%)**	127 **(+41.1%)**

As with Inference, I achieve excellent results improving performance over Tensorflow by more than 40%. Note again the 4% overhead added after integrating the custom model into Tensorflow. The ease of use Tensorflow provides makes this small performance hit well worth it.

Note that there is an extra 23ms added to execution time compared with adding up the times for inference and backpropagation layers. This is due to the weight update step. The use of momentum instead of vanilla SGD requites that 2 'copies' of the weights are kept. One is velocity and the other one is the weights themselves. If an update method like Adam were to be used another 'copy' would need to be kept which would increase execution time by another 10ms.

References

1. McCulloch, W.S., Pitts, W. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 5, 115–133 (1943). https://doi.org/10.1007/BF02478259

2. Hebb, D. O. (1949). The organization of behavior; a neuropsychological theory. Wiley.

3. HUBEL, D. H., & WIESEL, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. The Journal of physiology, 148(3), 574–591. https://doi.org/10.1113/jphysiol.1959.sp006308

4. Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

5. Roberts, L. G. (1963). Machine perception of three-dimensional solids (Doctoral dissertation, Massachusetts Institute of Technology).

6. Sah, Chih-Tang; Wanlass, Frank (1963). "Nanowatt logic using field-effect metal-oxide semiconductor triodes". 1963 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. VI: 32–33. doi:10.1109/ISSCC.1963.1157450

7. Huffman, D.A. (2012). Impossible Objects as Nonsense Sentences.

8. M. B. Clowes. 1971. On seeing things. Artif. Intell. 2, 1 (January 1971), 79–116. DOI:https://doi.org/10.1016/0004-3702(71)90005-1

9. The Roots of Backpropagation : From Ordered Derivatives to Neural Networks and Political Forecasting. New York: John Wiley & Sons. ISBN 0-471-59897-6.

10. Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biol. Cybernetics 36, 193–202 (1980). https://doi.org/10.1007/BF00344251

11. David Marr. 1982. Vision: A Computational Investigation Into the Human Representation and Processing of Visual Information. The MIT Press. ISBN: 9780262514620

12. Hopfield J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences of the United States of America, 79(8), 2554–2558. https://doi.org/10.1073/pnas.79.8.2554

13. Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. Nature 323, 533–536 (1986). https://doi.org/10.1038/323533a0

14. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

15. Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A neural probabilistic language model. J. Mach. Learn. Res. 3, null (3/1/2003), 1137–1155.

16. Hinton GE, Osindero S, Teh YW. A fast learning algorithm for deep belief nets. Neural Comput. 2006 Jul;18(7):1527-54. doi: 10.1162/neco.2006.18.7.1527. PMID: 16764513

17. Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09). Association for Computing Machinery, New York, NY, USA, 873–880. DOI:https://doi.org/10.1145/1553374.1553486

18. J. Deng, W. Dong, R. Socher, L. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

19. Glorot, X. & Bengio, Y.. (2010). Understanding the difficulty of training deep feedforward neural networks. Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, in Proceedings of Machine Learning Research 9:249-256 Available from http://proceedings.mlr.press/v9/glorot10a.html .

20. Scherer D., Müller A., Behnke S. (2010) Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In: Diamantaras K., Duch W., Iliadis L.S. (eds) Artificial Neural Networks – ICANN 2010. ICANN 2010. Lecture Notes in Computer Science, vol 6354. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-15825-4_10

21. K. Jarrett, K. Kavukcuoglu, M. Ranzato and Y. LeCun, "What is the best multi-stage architecture for object recognition?," 2009 IEEE 12th International Conference on Computer Vision, 2009, pp. 2146-2153, doi: 10.1109/ICCV.2009.5459469.

22. Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10). Omnipress, Madison, WI, USA, 807–814.

23. Glorot, X., Bordes, A. & Bengio, Y.. (2011). Deep Sparse Rectifier Neural Networks. Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, in Proceedings of Machine Learning Research 15:315-323 Available from http://proceedings.mlr.press/v15/glorot11a.html .

24. A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In ICML, 2013.

25. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. ArXiv, abs/1207.0580.

26. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. Commun. ACM 60, 6 (June 2017), 84–90. DOI:https://doi.org/10.1145/3065386

27. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J.M., Tran, J., Catanzaro, B., & Shelhamer, E. (2014). cuDNN: Efficient Primitives for Deep Learning. ArXiv, abs/1410.0759.

28. Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR, abs/1409.1556.

29. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S.E., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 1-9.

30. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770-778.

31. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A.C., & Bengio, Y. (2014). Generative Adversarial Networks. ArXiv, abs/1406.2661.

32. Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ArXiv, abs/1502.03167.

33. Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How Does Batch Normalization Help Optimization? NeurIPS.

34. John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. J. Mach. Learn. Res. 12, null (2/1/2011), 2121–2159.

35. Kingma, D.P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. CoRR, abs/1412.6980.

36. Cubuk, E.D., Zoph, B., Shlens, J., & Le, Q.V. (2020). Randaugment: Practical automated data augmentation with a reduced search space. 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 3008-3017.

37. Antoniou, A., Storkey, A., & Edwards, H. (2017). Data Augmentation Generative Adversarial Networks. ArXiv, abs/1711.04340.

38. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015 IEEE International Conference on Computer Vision (ICCV), 1026-1034.

39. Keskar, N., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P.T. (2017). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. ArXiv, abs/1609.04836.

40. Kandel, I., & Castelli, M. (2020). The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. ICT Express, 6, 312-315.

41. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. ArXiv, abs/2005.14165.

42. Jouppi, N., Young, C., Patil, N., Patterson, D.A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T., Gottipati, R., Gulland, W., Hagmann, R., Ho, C.R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K.A., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., & Yoon, D. (2017). In-datacenter performance analysis of a tensor

processing unit. 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 1-12.

43. Lavin, A., & Gray, S. (2016). Fast Algorithms for Convolutional Neural Networks. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 4013-4021.
44. Vincent, K., Stephano, K.J., Frumkin, M., Ginsburg, B., & Demouth, J. (2017). On Improving the Numerical Stability of Winograd Convolutions. ICLR.
45. Shmuel Winograd. Arithmetic complexity of computations, volume 33. Siam, 1980
46. Shmuel Winograd. On multiplication of polynomials modulo a polynomial. SIAM Journal on Computing, 9(2):225–229, 1980.
47. V. Madisetti. The Digital Signal Processing Handbook. Number v. 2 in Electrical engineering handbook series. CRC, 2010
48. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015 IEEE International Conference on Computer Vision (ICCV), 1026-1034.
49. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A.G., Adam, H., & Kalenichenko, D. (2018). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2704-2713.
50. Jain, S.R., Gural, A., Wu, M., & Dick, C. (2020). Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks. *arXiv: Computer Vision and Pattern Recognition*.
51. Goyal, R., Vanschoren, J., Acht, V.V., & Nijssen, S. (2021). Fixed-point Quantization of Convolutional Neural Networks for Quantized Inference on Embedded Platforms. *ArXiv, abs/2102.02147*.
52. Wu, H., Judd, P., Zhang, X., Isaev, M., & Micikevicius, P. (2020). Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. *ArXiv, abs/2004.09602*.
53. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., & Keutzer, K. (2021). A Survey of Quantization Methods for Efficient Neural Network Inference. *ArXiv, abs/2103.13630*.
54. Han, S., Pool, J., Tran, J., & Dally, W.J. (2015). Learning both Weights and Connections for Efficient Neural Network. *ArXiv, abs/1506.02626*.
55. Blalock, D.W., Ortiz, J.J., Frankle, J., & Guttag, J.V. (2020). What is the State of Neural Network Pruning? *ArXiv, abs/2003.03033*.
56. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M., & Dally, W.J. (2016). EIE: Efficient Inference Engine on Compressed Deep Neural Network. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 243-254.
57. *Cybernetics: Or Control and Communication in the Animal and the Machine.* Paris, (Hermann & Cie) & Camb. Mass. (MIT Press) ISBN 978-0-262-73009-9; 1948

58. Shannon, C.E., Weaver, W., & Wiener, N. (1949). The Mathematical Theory of Communication. *Physics Today, 3*, 31-32.

59. Minsky, M. (1952). A neural-analogue calculator based upon a probability model of reinforcement. *Harvard University Psychological Laboratories, Cambridge, Massachusetts*

60. Minsky, Marvin; Papert, Seymour (1988). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

61. Hochreiter, S. (1998). The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *Int. J. Uncertain. Fuzziness Knowl. Based Syst., 6*, 107-116.

Appendix

```
template<const int SMEM_SIZE>
__inline__ __device__ void swap_pointers_add(float*& p1, float*& p2, int idx){
   p1 = p1 + (idx&1)*SMEM_SIZE - (!(idx&1))*SMEM_SIZE;
   p2 = p2 - (idx&1)*SMEM_SIZE + (!(idx&1))*SMEM_SIZE;
}


//fetch 64x8 tiles from external memory until 256/8 tiles have been fetched
template<const int THREADS_X,const int SMEM_V_X,const int SMEM_V_Y,const int SMEM_U_Y,const int
SMEM_U_X>
__inline__ __device__ void fetch_from_VRAM(float* U_temp,float* U,float* V_temp,float* V,
                                           const int V_base_idx,const int U_base_idx,const int n_gmem){
   //load U to registers
   #pragma unroll
   for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X)/4; i++){
       int U_idx = U_base_idx + i*(THREADS_X/(SMEM_U_X/4))*(U_DIM/4) + n_gmem*SMEM_U_Y*(U_DIM/4);
       reinterpret_cast<float4*>(U_temp)[i] = reinterpret_cast<float4*>(U)[U_idx];
   }

   //load V to registers
   #pragma unroll
   for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X); i++){
       int V_idx = V_base_idx + i*(ITER_DIM) + n_gmem*SMEM_V_X;
       V_temp[i] = V[V_idx];
   }

}

template<const int THREADS_X,const int SMEM_V_X,const int SMEM_V_Y,const int SMEM_U_Y,
       const int SMEM_U_X,const int SMEM_SIZE,const int SMEM_V_SIZE>
__inline__ __device__ void store_to_SMEM(float* U_temp,float* V_temp,float* smem){
   //store U to shared memory
   #pragma unroll
   for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X)/4; i++){
       int idx = (SMEM_V_SIZE)/4 + threadIdx.x + i*THREADS_X;
       reinterpret_cast<float4*>(smem)[idx] = reinterpret_cast<float4*>(U_temp)[i];
   }

   //store V to shared memory
   #pragma unroll
   for(int i=0; i<((SMEM_U_X*SMEM_U_Y)/THREADS_X)/4; i++){
       int idx = (threadIdx.x/8)*(8+1) + (threadIdx.x%8) + i*(THREADS_X + THREADS_X/8 );
       reinterpret_cast<float4*>(smem)[idx] = reinterpret_cast<float4*>(V_temp)[i];
   }
}


//fetch 64x1 tiles from shared memory until 8/1 tiles have been fetched
template<const int SMEM_V_X,const int SMEM_V_Y,const int SMEM_SIZE,const int SMEM_V_SIZE,
       const int REG_V,const int REG_U,const int THREADS_X>
__inline__ __device__ void fetch_from_SMEM(float* V_reg,float* U_reg,float* smem, const int n_smem){
   //load V data from shared memory to registers
   #pragma unroll
   for(int i=0; i<REG_V/4; i++){
       int idx = (threadIdx.x/8)*(8 + 1) + i*(THREADS_X + THREADS_X/8) + n_smem;
       reinterpret_cast<float4*>(V_reg)[i] = reinterpret_cast<float4*>(smem)[idx];
   }
   //load U data from shared memory to registers
   #pragma unroll
   for(int i=0; i<REG_U/4; i++){
       int idx = (SMEM_V_SIZE)/4 + (threadIdx.x%8) + i*8 + n_smem*8*2;
       reinterpret_cast<float4*>(U_reg)[i] = reinterpret_cast<float4*>(smem)[idx];
```

```
    }
}

template<const int REG_V,const int REG_U>
__inline__ __device__ void compute(float* V_reg,float* U_reg,float* result_thread,const int n_smem){
   //compute outer products
   #pragma unroll
   for(int i=0; i<REG_V; i++){
       #pragma unroll
       for(int j=0; j<REG_U; j++){
           result_thread[i*REG_U + j] += V_reg[i] * U_reg[j];
       }
   }
}


template<const int REG_V,const int REG_U>
__inline__ __device__ void write_to_VRAM(float* result_thread,float* output,const int out_base_idx){
   //write output to VRAM
   #pragma unroll
   for(int i=0; i<REG_V; i++){
       #pragma unroll
       for(int j=0; j<REG_U/4; j++){
           int out_final_idx = out_base_idx + i*U_DIM/4 + j*8;
           reinterpret_cast<float4*>(output)[out_final_idx] =
                                           reinterpret_cast<float4*>(result_thread)[i*REG_U/4 + j];
       }
   }
}

template<const int THREADS_X,const int SMEM_V_X,const int SMEM_V_Y,const int SMEM_U_Y,const int
SMEM_U_X,
       const int REG_V,const int REG_U,const int SMEM_SIZE,const int SMEM_V_SIZE>
__global__ __launch_bounds__(64,1) void mmGPU8(float* V, float* U, float* output){
   int V_base_idx = blockIdx.z*V_DIM*ITER_DIM + blockIdx.x*(SMEM_V_Y*ITER_DIM)
                   + (threadIdx.x/SMEM_V_X)*ITER_DIM*8 + (threadIdx.x%SMEM_V_X);

   int U_base_idx = (blockIdx.z/BATCH_SIZE)*U_DIM*ITER_DIM/4 + blockIdx.y*(SMEM_U_X/4)
                   + (threadIdx.x/(SMEM_U_X/4))*(U_DIM/4) + threadIdx.x%(SMEM_U_X/4);


   __shared__ float smem[2*SMEM_SIZE];

   float result_thread[REG_V*REG_U] = {0};
   float V_reg[REG_V];
   float U_reg[REG_U];


   float U_temp[(SMEM_U_X*SMEM_U_Y)/THREADS_X];
   float V_temp[(SMEM_V_X*SMEM_V_Y)/THREADS_X];

   float* smem_buf = &smem[0];// set buffer pointer
   float* smem_comp = &smem[0];// set compute pointer

   fetch_from_VRAM<THREADS_X,SMEM_V_X,SMEM_V_Y,SMEM_U_Y,SMEM_U_X>
                                                (U_temp,U,V_temp,V,V_base_idx,U_base_idx,0);
   store_to_SMEM<THREADS_X,SMEM_V_X,SMEM_V_Y,SMEM_U_Y,SMEM_U_X,SMEM_SIZE,SMEM_V_SIZE>
                                                                   (U_temp,V_temp,smem_buf);
   __syncthreads(); // wait for threads to stop writing to shared memory

   smem_buf = &smem[SMEM_SIZE];// swap buffer pointer

   #pragma unroll 1
   for(int n_gmem=0; n_gmem<(ITER_DIM + SMEM_V_X - 1) / SMEM_V_X - 1; n_gmem++){


fetch_from_VRAM<THREADS_X,SMEM_V_X,SMEM_V_Y,SMEM_U_Y,SMEM_U_X>
```

```
                                                    (U_temp,U,V_temp,V,V_base_idx,U_base_idx,n_gmem+1);

        #pragma unroll
        for(int n_smem=0; n_smem<SMEM_U_Y; n_smem++){
            fetch_from_SMEM<SMEM_V_X,SMEM_V_Y,SMEM_SIZE,SMEM_V_SIZE,REG_V,REG_U,THREADS_X>
                                                                    (V_reg,U_reg,smem_comp,n_smem);

            compute<REG_V,REG_U>(V_reg,U_reg,result_thread,n_smem);
        }

        store_to_SMEM<THREADS_X,SMEM_V_X,SMEM_V_Y,SMEM_U_Y,SMEM_U_X,SMEM_SIZE,SMEM_V_SIZE>
                                                                          (U_temp,V_temp,smem_buf);

        __syncthreads(); // wait for threads to stop writing to shared memory

        swap_pointers_add<SMEM_SIZE>(smem_comp,smem_buf,n_gmem+1); // swap smem pointers
    }

    #pragma unroll
    for(int n_smem=0; n_smem<SMEM_U_Y; n_smem++){
        fetch_from_SMEM<SMEM_V_X,SMEM_V_Y,SMEM_SIZE,SMEM_V_SIZE,REG_V,REG_U,THREADS_X>
                                                                    (V_reg,U_reg,smem_comp,n_smem);

        compute<REG_V,REG_U>(V_reg,U_reg,result_thread,n_smem);
    }

    int out_base_idx = blockIdx.z*V_DIM*U_DIM/4 + blockIdx.x*(SMEM_V_Y*U_DIM/4) + blockIdx.y*(SMEM_U_X/4)
                    + (threadIdx.x/8)*(U_DIM/4)*8 + (threadIdx.x%8);

    write_to_VRAM<REG_V,REG_U>(result_thread,output,out_base_idx);
}
```

Appendix Figure 1: Stage 4 Batched Matrix Multiply Kernel Implementation