CNN Acceleration for Electrode Recordings

University of Toronto – ECE1782 Project Report

Corey Kirschbaum (997384165) | Lexi Zhu (1002620005) | Fabián Torres Álvarez (1007354765) | Kareem Ibrahim (1008219465)

Abstract *– The Adaptive Neurorehabilitation Systems Lab (ANS) research team, at University of Toronto, has been working on processing electrode recordings, worn by patients, for the peripheral nervous system (PNS), to help towards medical treatments. The ANS produced a paper on this matter, utilizing a convolutional neural network (CNN) to classify patterns from the processed electrode recordings, initially in response to implanted electrodes on lab rats. The goal of this project is to attempt to optimize the convolutional neural network (CNN) presented in their paper. Their design was initially implemented in Keras during development and then ported over to an FPGA to be used in implantable devices, in collaboration with the Intelligent Sensory Microsystems Laboratory (ISML), another research team from the University of Toronto. The ANS and ISML research teams found that the FPGA implementation did not perform inference fast enough, at around 10 seconds per classification, and suffered from memory utilization issues. However, their Keras Python implementation was stated to run much faster at 0.5 – 0.6 ms, and was suitable for their needs as they specify the acceptable maximum window for classifying is around 7 ms. However, after experimentally running a TensorFlow CNN model to validate the numbers reported in the paper, the conclusion that the inference time reported might be incorrect was reached and should be 5 - 6 ms. The main objective of this project is to attempt and provide an alternate, specialized design that utilizes GPUs and CUDA to optimize the CNN inference. The idea is that if GPUs are used, as they do not fit on an implant device, a cloud-based solution could be made to handle processing multiple patient electrode recordings by the CNN model. Hence, the goal of this project in optimizing the CNN model, with CUDA for execution on GPU, which can later be applied to a server implementation.*

The hypothesis at the start of the project was that it might be possible to do better than the FPGA implementation, especially if it performed around 10 sec per inference. However, regarding optimizing over Keras and TensorFlow, this was believed to be very difficult as these libraries have been optimized over time to perform well. Additionally, these libraries use other libraries like cuDNN and TensorRT from Nvidia which make use of specialized GPU functionalities that we may not even have access to. Also, given that we do not know the GPU used by the ANS lab, if they utilized cuDNN, if they were using Tensor Cores, or any other improvements over the UG lab machines due to resource limitations, it could be very difficult.

A few strategies were taken and attempted in the final CNN implementations. The best result obtained was around 13 ms, which is better than the FPGA implementation, but still not as fast as TensorFlow. There are however a few future optimizations that can still be performed to improve upon this result such as using the im2Col strategy, which is popular for convolution by making one large matrix multiplication/dot product or utilizing CUDA warp level primitives. If hardware optimizations like utilizing tensor cores were to be considered in the future, then a more up to date/expensive GPU would have to be used.

I. INTRODUCTION

Working with the Adaptive Neurorehabilitation Systems Lab (ANS) and the Intelligent Sensory Microsystems Laboratory (ISML) at University of Toronto, this project aims to accelerate the existing convolutional neural network (CNN) proposed by the research team to extract physiologically relevant information, like joint angles, from electrode recordings from the peripheral nervous system (PNS) of rodents. The CNN is related to the team's research for developing new treatments for PNS lesions, which in the future could be applied to humans. A recurrent neural network (RNN) was also developed by the ANS team to predict joint angles based on the predicted firing patterns from the CNN. The RNN will not be addressed in this project but could be a future area of optimization interest.

Recording and stimulating the PNS is becoming important in bioelectric systems, with peripheral nerve recording in humans still a challenge. Multi-contact nerve cuff electrode configurations have the potential to improve recording selectivity [1]. The CNN developed by the ISML team is used to associate recordings of individual naturally evoked compound action potentials (CAPs) with neural pathways of interest, by exploiting the spatiotemporal patterns in multi-contact nerve cuff recordings [1].

The ANS and ISML teams attempt to have an implantable device as their final system, since implanted peripheral neural interfaces have seen success with reduction of phantom pain or the restoration of sensation in amputees, treatment for overactive bladder, modulation of inflammatory activity using the vagus nerve, implanted functional electrical stimulation for movement restoration, phrenic nerve stimulation for obstructive sleep apnea [1].

The ANS team decided to implement a CNN for its ability to pick out structural similarities within neighbouring pixels through convolutional operations performed in 2D, allowing for improved integration of spatial and temporal information encoded within recordings [1]. The designed CNN is comprised of three fundamental building blocks: convolutional, pooling, and a fully connected layer. Convolutional layers convolve an input image with a square filter which slides along the image with a stride. The pooling layer groups neighbouring pixels and represents them as a single pixel. Traditionally max pooling is used, which takes the maximum value from neighbouring pixels. Pooling layers tend to result in an output with smaller dimensionality than the input. The fully connected layer then takes the output of the convolutional and max pooling layers and outputs a non-linear transformation of these features, for the purpose of classification. See Figure 2 below and the background section for more details.

A. Current Design

Initially, the ANS research team developed their CNN in Keras for evaluating their neural network design. Their Keras solution was documented in their paper to take 0.5 – 0.6 ms for each classification inference when running on an Intel i7-6700 CPU @ 3.40 GHz with 24GB RAM. However, they did not specify the GPU model they used. Compared to our own experiment, this result is at least an order of magnitude faster, which makes us think either the exact CNN model or methodology used to obtain results was not fully disclosed or the results were incorrectly presented. Their paper also

specifies a 5 – 7 ms window for classifying [1]. During our experiments with Keras and Tensorflow, the CNN model took around 4 – 8 ms on average, and as the window for inference specified in the paper is in the same range it is believed that the paper has inconsistencies in its reported results and model. We believe that the result from the paper should be 5-6 ms instead of the 0.5 - 0.6 ms. As well, the paper specifies that its maxpool layers are using a stride of 1, which does not make sense as pooling layers should reduce the dimensions and using a stride of 1 with same padding results in no change in dimensions, which contradicts their text. The research team did send us code for a more up to date CNN model, which had the maxpool using a stride equal to the pooling size of 2. Hence, we will be using a stride of 2 for max pooling.

The latest design by the ANS and ISML teams is an implant to be worn by patients and therefore has a limitation on physical dimensions, power consumption, and throughput of the accelerator. Previous attempts have been made at implementing the CNN on an FPGA, which can be placed inside an implant. However, they have not had acceptable performance in terms of throughput, taking in the order of 10 seconds per inference. This number again is questionable to us, but as we do not have any information regarding their FPGA CNN implementation design in terms of how they performed the layers (convolution, maxpooling, fully connection layer, memory utilization), we will have to take this timing result as is.

Figure 1 provides the CNN architecture which was presented in the ANS research group's paper [1], with the maxpooling stride equaling pool size=2 (from assumption above). The input is a 56x100 matrix generated by processing the electrode signals.

Layer	Kernel/ number of nodes	Stride	Padding	Activation function
Conv_1	$8 \times 8 \times 64$	1	Same	*ReLu*
Max Pooling_1	2×2	2	Same	
Conv_2	$4 \times 4 \times 64$	1	Same	*ReLu*
Max Pooling_3	2×2	2	Same	
Conv_3	$2 \times 2 \times 64$	1	Same	*ReLu*
Fully Connected_1	256			*ReLu*
Output	3			*softmax*

'Same' means that the output of the layer is the same size as the input.

Fig. 1. CNN Architecture [1]

B. Proposed Solution

We propose a solution that uses a GPU to compute the CNN inference. As most GPUs require fans and are physically too large to place on a wearable or implant device, we are proposing a cloud-based solution. In such a solution, one or more servers would be taking requests from at least one patient, queueing up requests and either processing them in pipeline or in parallel depending on the resource limitations of the GPU and the implementation. The implementation will then be capable of running multiple instances of the CNN with a focus on latency, throughput, and memory utilization. The idea is to utilize streams and asynchronous processing of requests on the GPU to allow for the maximum number of CNN's/requests to be processed.

Our first objective, though, is to provide a CUDA solution that optimizes performance over the FPGA solution. If time permits, we will then tackle the evaluation of our CUDA solution versus the Keras solution. It would be expected that if we were able to integrate cuDNN or implement our own specialized solution it might be faster than the more general Keras-Python solution or cuDNN. However, cuDNN might be using hardware features Nvidia has access to, but we do not. As well, cuDNN does depend on the GPU version and can only make use of tensor cores if available. If we can provide a solution that performs well on any GPU regardless of tensor cores that might make for a more affordable solution (not requiring costly high-end GPUs that may not be available in the labs). However, we should note that cuDNN was not used during implementation as lab UG machines did not have the library installed.

This problem was chosen for our project as the group members have an interest in machine learning, interest in improving our knowledge on GPU performance utilization, and it relates to a practical real-world problem for a research team.

C. Challenges Faced

Many challenges were faced during this project. There was the challenge of obtaining information from the ANS and ISML research groups as communication was quite limited. Hence, information pertaining to validating the model structure and behaviour, obtaining model code and weights (a working model in Keras and an FPGA) was either hard or were not obtained. The workaround for this was the use of Google Colab and a local GPU on a group member's home machine to code and run a TensorFlow implementation. With TensorFlow being the back end of Keras, we expect Colab to have similar performance. This gave us a means of validating our CUDA implementation's output and comparing the timing/performance result to that presented in the ANS paper [1]. We used self generated weights (all ones), as the weight values themselves do not matter for our purposes since this project is only interested in optimizing performance. If proper weights are needed in the future, code can be written to parse a file containing the weights and load them into our implementation.

Challenges were faced with the UG lab machines as well, as the cuda-gdb did not work, the hardware did not support tensor cores, and the lab machines did not have the cuDNN library installed and we were unable to install it ourselves.

There were challenges in understanding the CNN architecture, the way it operates, and how the layers work. As our implementation focused on a specialized CUDA GPU CNN implementation, we needed to fully understand how the layers operate in order to implement them. Understanding how stride and padding work, and their implementation was difficult. Additionally, designing for parallel execution of GPU CNNs in a server was considered during the implementation phase.

During implementation, we ran into issues regarding GPU resource limitations; not all filters fit into constant memory, streams and asynchronous design did not help with performance since matrix sizes were not large enough, and for the same reason shared memory didn't help. Significant time was spent deciding how to best utilize memory, the order of processing of the CNN, and how to reuse GPU memory to

minimize alloc/dealloc. We also explored using templates, pragma unroll, and large caching arrays, but ran out of registers which caused the GPU code not to run as not enough resources were available for scheduler.

II. Background

Figure 2 presents a simple view of what a traditional CNN networks look like, containing an input matrix (ex. image), a sequence of convolutions with ReLU, pooling layers, and a deep neural network (DNN) also called fully connected linear layer. The pooling layers tend to decrease the size of the matrix result, while the convolution tends to increase the number of output channels. The reason for this behaviour is that convolution attempts to find patterns in the input data and pooling tends to average out the findings of the convolution to have a model as general as possible. The CNN then ends with a fully connected layer to classify the input into discrete categories.

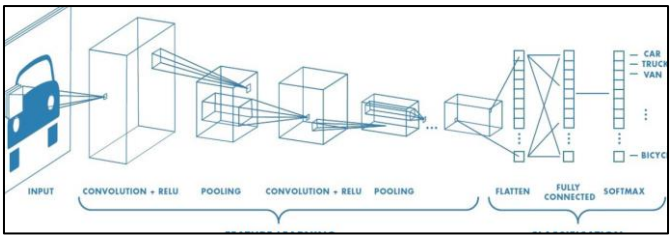


Fig. 2. Convolution Neural Network (CNN) [5]

A. Convolution

Figure 3 below demonstrates a simple example of a one-dimensional convolution, modified from [3], to illustrate how stride and padding relate to the output size. In the example, the input has a length of 10 elements and the filter has size 3. Zero padding is added at the beginning and end of the input to give an output of the same size as the input (10 elements), this strategy is known as ”same” padding. As with any convolution operation, the filter is shifted across the input, multiplying the overlapping elements, and adding them together, as shown in the example.

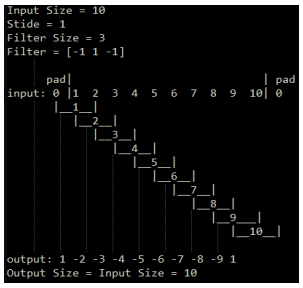


Fig. 3. 1D Convolution Example

For a 2D input matrix, 2D filters are used and padding is potentially applied to each boundary of the input matrix (top, left, right, bottom).

Figures 4 and 5 below present a multi-channel convolution example. Padding is added along the boundaries while using stride of 1 to ensure the output is the same size as the input. This example contains 3 input channels, which can be interpreted as applying convolution to an RGB image. Three independent 2D filters, with their corresponding weights (one for each input channel), and one bias term are used per output channel. This convolution generates two output channels; hence two sets of filters and biases are used.

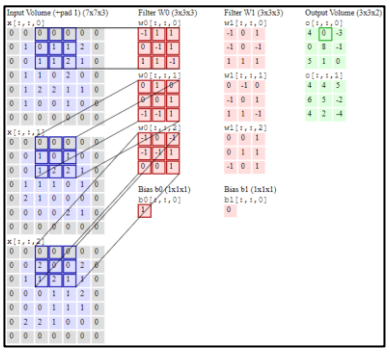


Fig. 4. Multi-Channel 2D Convolution Example – 1st Output Channel [4]

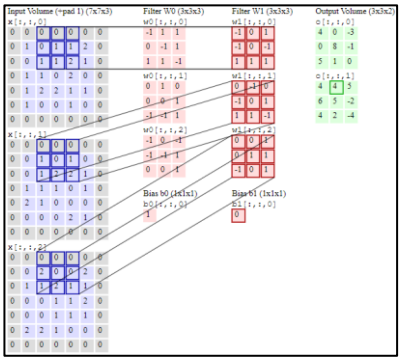


Fig. 5. Multi-Channel 2D Convolution Example – 2nd Output Channel [4]

B. Max Pooling

Figure below shows an example of max pooling, where a pool filter size of 2x2 is used and the max element with the filter area is used per the output.

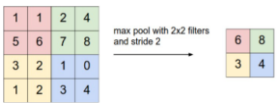


Fig. 6. Max Pooling Example [4]

Like convolutions, TensorFlow and Keras provide two strategies for zero padding on the boundaries of the matrix named “valid” and “same”.

“Valid” padding adds no zeros at all, shifting the filter over the inputs and stopping when an edge is reached. Because of this, some input values are discarded when the input size does not perfectly line up with the filter.

“Same” padding consists of symmetrically adding zeros to boundaries of the input in such a way that if the filter stride was 1, the output size would be equal to the input. This strategy results in a somewhat larger output size but has the advantage that none of the input samples are discarded, minimizing the risk of losing important information.

“Same” padding was the option implemented in our CUDA code as it is the one used by the target CNN from [1].

C. Fully Connected Layer

The fully connected layer takes an array of inputs of size 1xM, a matrix of weights with MxN dimension, and performs a dot product such that the output=Wx + b, where b is an array of biases of 1xN. This results in an output array of size 1xN.

An activation function such as ReLU (inner layers) or Softmax (last layer for classifying) is then used on the output of the fully connected layer.

III. Implementation

A. GPU Used

The GPU used was the GTX980 [2], present on the UG lab machines at University of Toronto. It has 4GB of GDDR5 memory which makes up the global memory (GMEM), 64KB of constant memory, 96KB of shared memory (SMEM), and 16 Maxwell streaming multiprocessors (SMs) with 4 warps per SM. Each warp scheduler can dispatch two instructions per warp every clock.

B. Architecture Diagram

Figure 7 below presents an illustration of what the convolution layers of Figure 1 look like when implemented. The ReLU activation function and max pooling are not shown for the sake of simplicity, but they are placed between each of the 3 convolution layers. A stride of 1 is used with padding "same" for convolution. Hence, the convolution operation will output a matrix with the same dimensions as the input. "Same" padding means that the appropriate number of zeros is added to the input so that the output dimension is the same as the input.

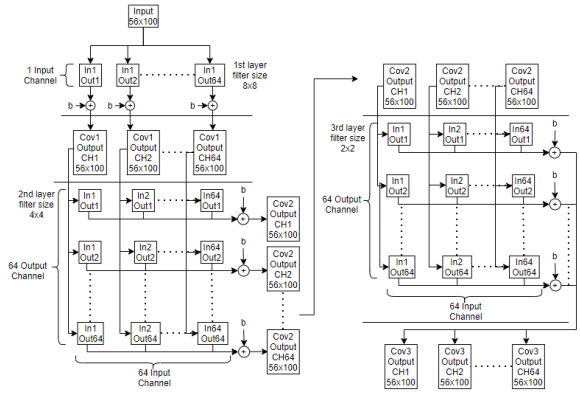


Fig. 7. IML CNN Convolution Layers

C. Storing Filters Weights

There are multiple ways to implement convolution in CUDA. However, there are implementation limitations based on where filter weights are stored in memory.

Our first idea was using GPU constant memory to store filter weights as they do not change during inference. Constant memory uses global memory (GMEM) but allows for caching results, so that if multiple threads read the same constant memory address at the same time it will only count as a single read from memory. Constant memory should be faster than directly using GMEM to store the filter weights. An alternative is to place the weights in GMEM and use CUDA kernel functions to copy the weights into shared memory (SMEM). Constant memory seems to be a fast means to get started, and it was the approach taken during development.

The amount of constant memory available based on the GPU used is 64KB.

Layer 1 filter weights: (8 * 8) * 64 out + 64 bias = 4160 neurons

Using weights of type float, = 4160 neurons * 4 bytes/float
= 16,640 bytes = 16.64 KB (1)

Layer 2 filter weights: (4 * 4) * 64in * 64out + 64bias
= 65,600 neurons
= 65,600 neurons * 4 bytes/float
= 262,400 bytes = 262.4 KB (2)

Layer 3 filter weights: (2 * 2) * 64 in * 64 out + 64bias
= 16,448 neurons
= 16,448 neurons * 4 bytes/float
= 65,792 bytes = 65.792 KB (3)

Total for convolution filter weights = 344,832 bytes
= 344.832 KB (4)

344.832 KB is greater than the total size of the constant memory available on the GPU. The first option is to place filter weights in GMEM and copy them into shared memory in the GPU kernel code. The second option is to only utilize enough constant memory for the first layer (64 filter channels) and overwrite constant memory when executing next layer. The third option is to use a combination of both. In our implementation, we chose the second option, utilizing enough constant memory for only 64 filters, based on the size of the first layer's filters (8x8). This seemed the quickest way to get started with the implementation. However, with this choice, sequential parallelization/pipelining of convolution layers is limited to the number of filters of size 64 that fit in constant memory.

The maximum filter size is 8x8 (the first layer), or 16.64KB as shown in (1). Hence, we can only pipeline floor(64KB / 16.64KB) = 3 CNN inferences, which is not a lot. Hence, the bottleneck is size of constant memory.

If the GMEM or shared memory approaches (second option) had been taken then we could get around the constant memory limitation, and maybe shared memory would have performed better than constant memory. This could be something to try in the future.

Something to also think about, but is not of importance to our CNN model, is if the weights could not fully fit in GMEM. In this case another option could be utilizing virtualization, that is accessing weights on host machine across the data bus (PCIe), but this is not necessary for our relatively small CNN.

D. Storing Channel Matrices

Next, we consider the amount of memory needed to store the input and output matrices to assess the impact of memory allocation.

1) *Allocate Matrix per 1 input:1 output*

In the worst case, every convolution and max pooling is performed independently, each taking in a single input channel and writing to a single output channel. In that case the amount of GMEM needed is:

input = 56*100
conv1 = 56*100*64 (once done don't need input)
maxpool1 = 28*50*64 (once done don't need conv1)
conv2 = 28*50*64*64 (once done don't need maxpool1)
maxpool2 = 14*25*64 (once done don't need conv2)
conv3 = 14*25*64*64 (once done don't need maxpool2)

Total memory = 7,644,000 floats * 4 bytes/float
= 30,576,000 bytes = 30.576 MB

As GMEM is in the gigabytes, there should be enough GMEM not matter what implementation approach is taken.

E. Convolution

Based on the architecture diagram shown above in Figure 7, layers operate sequentially, each layer taking in the results of the previous one. The sequential layered/barrier nature of the architecture could lend itself to pipelining inference results, allowing for parallel processing of different inputs each executing at a different layer within the CNN. If such an approach were taken, GPU memory and resources could be further reused, helping with performance by minimizing reallocation. However, a pipelined approach limits parallelization to the number of layers, or the amount of memory available for use, unless the GPU has enough resources for running multiple CNNs simultaneously or multiple GPUs are available.

The first convolution layer applies 64 filters independently to one input channel matrix to generate 64 output channels (64 independent convolution operations). Hence, each of the 64 convolutions in the first layer can be performed in parallel with streams, or in a single stream GPU operation. If performed in parallel, a performance benefit might occur because of concurrency. If performed in a single stream, then all future operations, on any channel in the CNN, will have to wait for all 64 channels in the first layer to complete. However, in this case the input matrix will only have to be read in once from the GPU's global memory (GMEM).

In any case, the convolution operation cannot be performed in-place, that is the output written into the same memory as the input, which would reduce memory utilization and allocation time. This is because, with blocks in the GPU's grid executing in any order by the scheduler and the fact that convolution requires reading a filter square area of the input each time. If in-place processing occurred, there would be a possibility for overwriting input elements that might be required later by another grid block.

For the second and third convolution layers, due to the potential synchronization issues that could result in incorrect output, parallelism can only occur in relation to input or output channels. That is, if everything ran in parallel there is a potential concern that performing additions could be incorrect due to lack of synchronization. As an example, if all 64 input channels, that are used to generate output channel 1, ran in parallel using a single output matrix for each filter, the results will need to be added to the final output matrix, taking an extra step. As well, if all 64 input channels are utilizing the same output matrix, to minimize memory allocation, and some convolutions of the filters run exactly at the same time, they will each pull the same value from the shared output matrix, add their result, and write it out to the shared output matrix, resulting in the updates to the shared output matrix being dropped. Only if separate output files are used will there be no issues. In terms of running the 64 output channels in parallel, as they already produce separate result matrices, they can be performed in parallel without synchronization concerns. However, as the filters in this scenario are applied to the same input, to reduce the number of times the input elements are read from GMEM, a single thread could read the input once and then process the read-in/cached input on all filters.

1) Handling Padding

When performing convolution, padding may be necessary depending on filter size. From [3, 6] we can figure out the amount of padding needed.

W = input dimension
F = filter dimension
P = padding amount
S =stride amount

$$\text{OuputSize} = (W - F + 2P) / S + 1 \quad (5)$$

In our case, we used Padding="same" and stride=1, so the convolutions have enough padding for the output size to equal the input size. Setting output size to input size in (5) we get (6), which rearranges to (7).

$$W = (W - F + 2P) / 1 + 1 \quad (6)$$

$$P = (F - 1) / 2 \text{ (Equation for padding)} \quad (7)$$

This equation can be used to get the padding needed for each side [8]. Figure 8 below shows a simple example of how these paddings relate to the input matrix to convolution operation.

totalPaddingHeight = filterSize - 1
totalPaddingWidth = filterSize - 1
topPadding = totalPaddingHeight / 2
leftPadding = totalPaddingWidth / 2
bottomPadding = totalPaddingHeight - topPadding
rightPadding = totalPaddingWidth – leftPadding

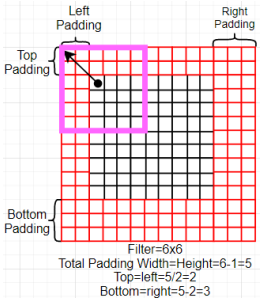


Fig. 8. Convolution Padding Boundry Exaample

There are two main ways of handling padding in the convolution implementation in CUDA.

The first simple option would be to pre-pad the input matrix and then pass it to a kernel to apply the convolution. In this case, every input matrix to the convolution function will either be a different size depending on the amount of padding needed, or the largest padding amount will be used for all. That is, for each convolution layer, as the filter sizes are different, the amount of padding for each layer is different. If we want to reuse allocated GPU memory so that performance is not affected by memory allocation and deallocation, then the best option is making all matrix sizes be with the largest padding amount and have the convolution kernel figure out where to start/process from in the padded input matrix. In this option, an input and output will be allocated with max padding size added to the 56x100.

It should be noted that convolution cannot be performed in-place as threads across GPU blocks cannot be synchronized. We were thinking of exploring the idea of reusing the input matrix by synchronizing threads or moving data to shared memory, but we currently do not believe it is possible.

The second option is to mathematically figure out the elements needed for filter convolution shifted by padding and handle the matrix boundary with "if" condition checks. This means that input and output do not need to be padded, saving some allocation space. This was implemented using only global memory and it worked. This second option was also modified to utilize shared memory to improve performance, though found due to the small input sizes utilizing shared memory in this instance made no difference to performance.

2) Handling Padding – Shared Memory

Utilizing shared memory on the input to the convolution, requires copying it from GMEM to shared memory. As each GPU block in the grid will work on a section of the input matrix, handling the boundaries of the block is important. That is, as each boundary element is involved in its own convolution filter overlap, a padding of this outlier area needs to be performed. When a block's edge is at the boundary of the input matrix, then the shared memory for that block will be padded with zeros. When a block's edge is processing an area within the input matrix, then the shared memory for that block will be padded with the values from the input matrix. Figure 9 shows an example of the issue of a filter square (purple) crossing the boundary of a block (light green). The purple dot is the element in which the current thread position in the block is processing the convolution. It's overlapping filter area is shifted by padding amount to become (x-leftPad, y-topPad). Hence, the purple filter area crosses the block boundary, which is why shared memory needs to also contain the padded data.

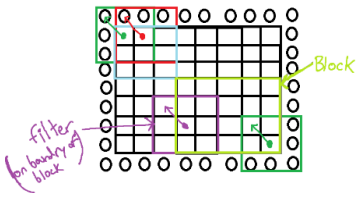


Fig. 9. Filter and Block Boundry Issue for Shared Memory

When adding shared memory utilization to the GPU function for convolution, each thread in a block copies its element from input matrix in GMEM to the correct position in shared memory for the block. However, dealing with the padding issue describe above, we implemented setting up a block's padded area in shared memory in parallel, to reduce the time spent. Figure 10 below shows an illustrative view of how block threads were selected to copy input matrix to shared memory. All threads in the block were utilized to enable parallelization, even if a thread in the block was outside the input matrix area. The black square is the input matrix. The blue area sets up the top and bottom padding. The beige area sets up the left and right padding, in their range. The inner purple area sets up the corners, plus the bit of extra side area.

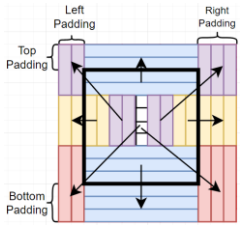


Fig. 10. Parallelizing Block Threads for Convolution Shared Memory Copy

F. General Matrix to Matrix Multiplication (GEMM)-based Convolution

Another implementation we tried is GEMM-based convolution, which transforms the convolution into a simpler problem, by use of one large matrix multiplication (dot product).

The heart of deep learning applications is matrix multiplication known as Generalized Matrix Multiplication (GEMM). It's used for fully connected layers, Recurrent Neural Networks, and Convolutional Neural Networks.

In general, convolution is a dot product between multiple copies of the filter with different overlapped parts of the input matrix. If we were able to arrange the filter and input data into a 2D arrays that when multiplicated generates the same convolution output, then we can use optimized matrix multiplication implementations. The re-layout of the input image into a matrix is called image to column (im2col). Considering the normal convolution operation shown in Fig. 11, the same convolution can be done as a matrix multiplication as shown in Fig. 12.

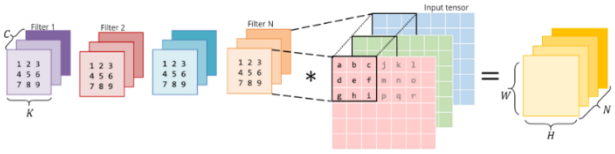


Fig. 11. Normal 3*3 convolution operation

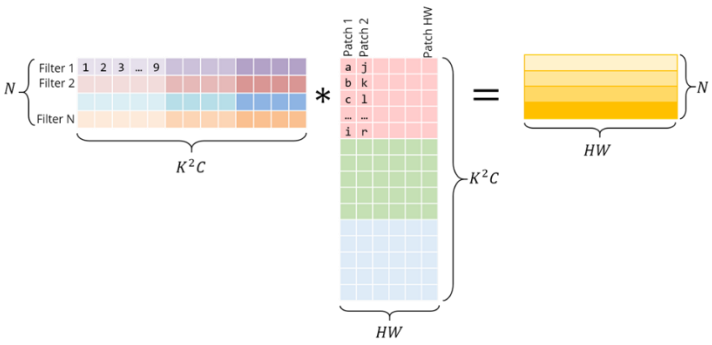


Fig. 12. GMEMM-based 3*3 convolution

To achieve this form of convolution, we need to convert the inputs in the shown way which consumes much more memory resources. The output matrix doesn't need to be converted at all. Despite the time delay of the conversion and the need for extra memory resources, we can benefit from the optimized implementations of matrix multiplications in libraries like cuBLAS.

The implementation of the convolution is divided into two main operations: first, an im2col transform kernel is implemented to transform the input in the form of a single column. Each thread handles stacking filter elements into the output column. The main convolution function calls the transform function before calling cublasSetMatrix and cublasSgemm to do the actual matrix multiplication between the adapted input and filters. All operations are done in a ping pong manner as the output of each layer is the input of the next one while reallocating the correct sizes each time.

G. Max Pooling

Max pooling layers are functionally very similar to convolutional layers, with the exception that they do not need

weights or biases, greatly simplifying memory management. As such, most of the analysis done for convolutional layers also apply to max pooling layers. For this reason, max pool layers cannot be run in-place and separate memory needs to be allocated for inputs and outputs.

Because of the computation structure, max pool layers were implemented with output parallelization, where each kernel thread computes each output pixel independently. "Same" padding was implemented to match the ANS research paper, with the method from [8] and simple "if" clauses.

```
int px_pre = (in_cols % STRIDE == 0) ? max(POOL_SIZE
- STRIDE, 0) : max(POOL_SIZE - (in_cols % STRIDE), 0);
px_pre /= 2;

int py_pre = (in_rows % STRIDE == 0) ? max(POOL_SIZE
- STRIDE, 0) : max(POOL_SIZE - (in_rows % STRIDE),
0);
py_pre /= 2;

// Shift start of max pool by px_pre and py_pre
int i_y_min = o_row * STRIDE - py_pre;
int i_x_min = o_col * STRIDE - px_pre;
```

It is worth noting that max pool layers do not have weights or biases but depending on the stride the input pixels may be read multiple times. This suggests that it could be possible to further optimize these layers by copying the inputs into constant or shared memory. However, our CNN implementation uses stride equal to the filter size, so inputs are only read once, and we would see no gain from this strategy.

H. *Flatten*

As stated previously, the CNN implemented includes 3 convolution layers, the output of the last layer is an array of pointers that points to the convolution output of each of the 64 channels, each channel with an output size of 14x25. To pass these outputs to a fully connected layer, we must first flatten the convolutional outputs to get an array of size 1x22400 (64 channels x 14 x 25 = 22400). This flattening was done in CUDA by:

1. Allocating device memory of (22400 * sizeof(float))
2. Passing each channel output pointer to the kernel and calling the kernel 64 times
3. Using 14x25=350 threads to copy each channel output to their respective locations in GMEM

I. *Fully Connected Layers*

There are 2 fully connected layers implemented in our solution. The first one has an input of size 1x22400 directly from the flattened result of the third convolution layer. It is multiplied with a weight matrix of size 22400x256 and then added with a 1x256 bias array. The second fully connected layer has input size 1x256 from the output of the first layer, and is multiplied with a weight matrix of 256x3, and then added with a 1x3 bias array.

The matrix multiplication calculations were done in CUDA by using 1 thread for each column of the weight matrix (thread per one of the 256 output) and multiplying with the entire input array. The thread then adds the sum of that multiplication result with 1 bias element to produce the respective output element. Figure 13 illustrates this implementation. This method of having threads read contiguously column-wise, instead of having each thread read row-wise, results in a 5 ms gain.

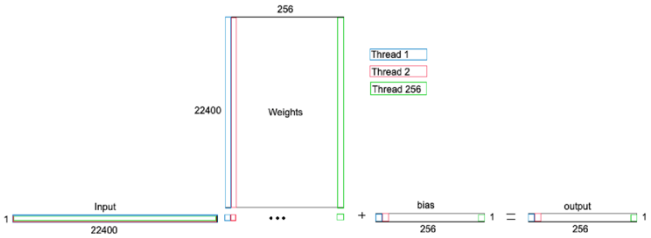


Fig. 13. Matrix multiplication using threads (first fully connected layer)

The output of the first fully connected layer (size 1x256) is passed through a ReLU activation function while the output of the second fully connected layer (size 1x3) is passed through a Softmax function to obtain the result of the CNN.

IV. Evaluation

A. *ANS Research Paper*

The ANS research paper specifies that the CNN inference took around 0.5-0.6 ms. However, this result seems incredibly fast especially when compared to our own attempt with TensorFlow and Keras (section B and C below). As well, the paper specifies a window of 5-7 ms for inference time. Hence, we believe in the paper they might have meant 5-6 ms. We were not able to confirm with the ANS team, as well as the GPU model used or any other information that could help with understanding their performance or implementation beyond the CNN model.

B. *Google Colab*

Google Colab was used to attempt to recreate the CNN model presented in the ANS paper, creating a Jupyter notebook to build a TensorFlow/Keras model of the CNN and get timing results. Doing this not only gave use some timing information to go with, but also helped with debugging our own CUDA implementation by comparing output/behaviour results. The GPU used by Google Colab was the Tesla K80. Researching online to get a better understanding of how TensorFlow and Keras in Google Colab utilize the GPU, we do not believe it is using tensor cores. It is specified that GPUs with compute capability of 7.0 or higher can utilize Tensor Cores with mixed precision by specifying float16 instead of float32 for data [17]. This is because Tensor Cores accelerate float16 matrix multiplications and convolutions. As Tesla K80 has a compute capability of 3.7, Tensor Cores are not supported. We are not sure if Colab is using the cuDNN library as we cannot verify (use the terminal) when using a free account version, but we believe that it is.

Trying to get Colab results when purely CPU is used for CNN inference it specified "Cov2D op currently only supports the NHWC tensor format on the CPU, but NCHW was used". Hence, we were unable to get a CPU result with Colab.

Table 1 below presents timing results for the CNN inference using Colab. The 1st column shows the first run of the Colab session (after connecting), which takes much longer than subsequent runs within the same session (3rd Avg column). This may be due to the Colab session finding servers/GPUs to run on, setting up the model on the GPU and

memory, and running the TensorFlow code. Overall, we are not entirely sure how Colab and TensorFlow process the model and execute the code, if it all runs on a single machine or if the code is being split up and sent to multiple machines, although one could think that Colab just sends the Jupyter notebook to a machine and runs the TensorFlow code on it. In that case the timing results would be due to TensorFlow's behaviour. The 2nd column in Table I shows that when running the model after Colab makes a session connection the timing is much faster. 3rd column shows an average timing result when assumed no setup is performed, that is session connection in Colab and TensorFlow has already been performed. Hence, if the 3rd column is more representative of the actual CNN inference time, then it is around 4 to 30 ms with lots of variance. This result is also close to the assumed 5 – 6 ms result in the paper.

TABLE I. GOOGLE COLAB CNN RUNTIME

(ms)	**Initial Run**	**1st Follow Up Run**	**Avg**	**Variance**	**Min**	**Max**
No Batching	10797.88	702.28	6.45	17.33	4.29	16.14
Batching 10	10153.2	72.22	5.02	0.06	4.42	7.66
Batching 100	10538.24	169.84	11.27	535.27	3.94	29.2

C. Local GPU and CPU Tensorflow Model

One member in the group had a GPU on their computer and so we further investigated the inconsistencies found in Colab. The GPU used was GTX 1650, the CPU was Intel Core (TM) i7-10750H with 16 GB installed RAM. Random initialization was used for the CNN model weights to attempt and mitigate any potential caching performed by TensorFlow. Table II presents a summary of the findings. cuDNN was enabled, although GTX 1650 does not support tensor cores. This gives us an understanding of the expected best performance that our implementation would compete against. The results show that the first iteration is still much slower than subsequent runs, hence the behavior a TensorFlow set up issue. The average time seems to be around 3-8 ms which is in the same range as paper and Colab timings.

TABLE II. LOCAL GPU TENSORFLOW RUNTIME – NO BATCHING, WITH CUDNN

(ms)	**Iteration**	**Avg**
GPU	1739	3 - 8
GPU Batch 200	2235.1	3
CPU	60000	60 - 80

D. Cuda Parallel CNN

As cuDNN library was not available to us on the UG machines, we stopped considering utilizing a CUDA library and went straight to implementing our own CNN in CUDA from scratch. From group members findings from ECE1782 lab2, performance gains were obtained by utilizing CUDA streams and reusing/reducing the amount of allocated GPU memory. Shared memory was also utilized in lab2, but did not help with performance gains, only with solving some synchronization issues for data correction. Hence, the first implementation idea was to evaluate if using streams for asynchronous convolution and max pooling would help. This initial investigation for parallelizing convolution was performed in the simple.cu code. Figure 14 below demonstrates the structure to this first attempt. A stream is created per output channel, with the first stream copying the input data over and a recorded event is used to sync streams so that they execute their respective GPU kernel function after the input and their convolution filter weight is copied over. Constant memory is used to store the convolution filter weights. The convolution padding was calculated and handled with if condition boundary checks in the GPU function, with the input data stored in GMEM (no shared memory). This first attempt at the first convolution layer took 2 ms.

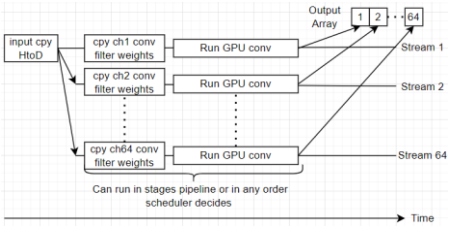


Fig. 14. Convolution Layer 1 First Attempt

Utilizing shared memory, as described in the implementation section, the first convolution layer still took 2 ms. As the amount of work required by the first layer seems too small due to the fast execution time and small input size, in order to really evaluate the effect of using shared memory we increased the input size from 56x100 to 2048x2048. In this scenario, when not using shared memory, the runtime was 652 ms. When using shared memory, the runtime was 8 ms. Hence, when operating on large data sizes (large input matrix for convolution), shared memory greatly improves performance. However, for the small size used in the project (56x100), using shared memory doesn't seem to make a difference.

Extending this design to the rest of the CNN, cnnParallel.cu code is our first CUDA attempt at the full CNN implementation, and takes around 26 to 30 ms, even when using shared memory for convolution. The figure below shows the stream usage, with 128 streams used, though the scheduler is expected to run a maximum of 16 at a time. The first convolution layer uses the first 64 streams, with an event recorded per convolution stream completion. The second and third convolution layers are more complicated due to having multiple input channels. In the second layer, the next 64 streams are used (65-128) and they begin once the appropriate stream from 1-64 per channel processing has completed its convolution and max pooling, with all input channels on the same output channel being processed on the same stream sequentially. Then, in the third layer, the 64 streams are swapped so that streams 1 to 64 are used for convolution and they wait for streams 65 to 128 to complete. It should be noted each stream will only wait for one stream (that produced that channels convolutions input), as well the filter weights are copied asynchronously to constant memory such that only 64 are used, and when a convolution completes, on the same stream, the filter weights for the next sequential input channel to be processed is copied over before convolution.

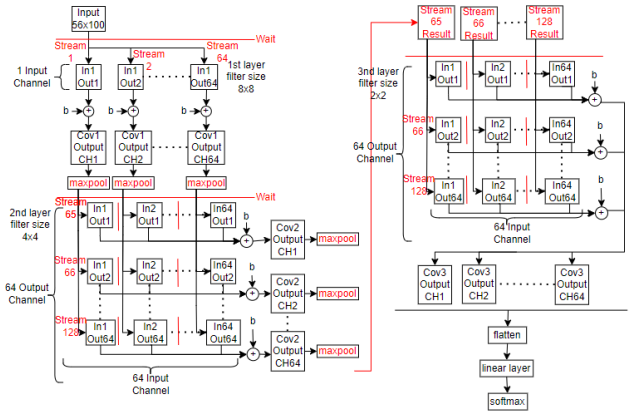


Fig. 15. Parallel CNN Stream Usage

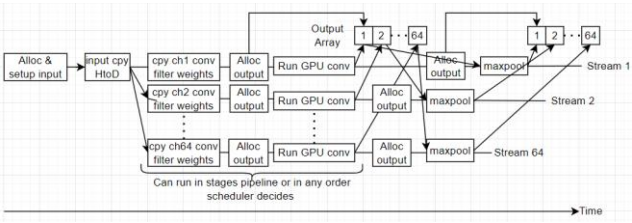
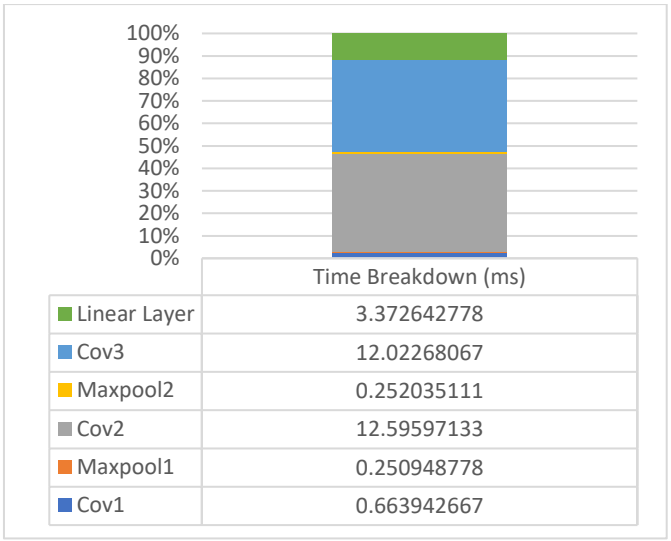


Fig. 16. Parallel CNN First Layer – Stream Behaviour



	Time Breakdown (ms)
Linear Layer	3.372642778
Cov3	12.02268067
Maxpool2	0.252035111
Cov2	12.59597133
Maxpool1	0.250948778
Cov1	0.663942667

Fig. 17. Parallel CNN - Percent Time Breakdown

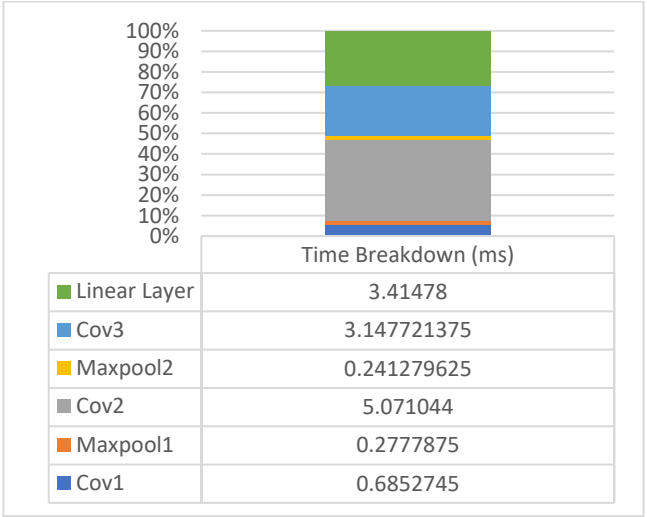
```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   85.07%  54.183ms      8256  6.5620us  3.5200us  40.256us  device_CNN(int, int, flo
int)
                   12.08%  7.6952ms      8261     931ns     608ns  1.8785ms  [CUDA memcpy HtoD]
                    1.91%  1.2194ms         2  609.70us  15.296us  1.2041ms  linear(float*, float*, f
                    0.72%  460.61us       128  3.5980us  3.2960us  4.1600us  max_pool_2d(float*, int,
                    0.21%  135.94us        64  2.1240us  2.0800us  3.3600us  flatten(float*, float*,
                    0.00%  1.2480us         1  1.2480us  1.2480us  1.2480us  [CUDA memcpy DtoH]
      API calls:   48.29%  95.453ms         5  19.091ms  47.011us  93.387ms  cudaHostRegister
                   24.19%  47.824ms         2  23.912ms  338.89us  47.485ms  cudaDeviceReset
                   11.79%  23.310ms      8450  2.7580us  2.1450us  340.47us  cudaLaunchKernel
                    9.86%  19.499ms      8256  2.3610us  2.0910us  16.693us  cudaMemcpyToSymbolAsync
                    2.62%  5.1698ms      8255     626ns     500ns  309.37us  cudaStreamWaitEvent
                    0.98%  1.9357ms         5  387.15us  6.7750us  1.8980ms  cudaMemcpy
                    0.62%  1.2307ms         2  615.33us  10.090us  1.2206ms  cudaDeviceSynchronize
                    0.61%  1.2152ms       328  3.7040us  1.8960us  157.48us  cudaMalloc
                    0.43%  855.93us       128  6.6860us  1.3050us  201.67us  cudaStreamCreate
                    0.15%  299.17us       128  2.3370us  1.4990us  61.341us  cudaStreamDestroy
                    0.09%  177.90us       129  1.3790us  1.2030us  4.1230us  cudaEventRecord
                    0.09%  171.15us        64  2.6740us  1.7020us  17.554us  cudaFree
```

Fig. 18. Parallel CNN perf Results

E. *Cuda Multi-Channel CNN*

The 30 ms timing result from parallel CNN is slower than the maximum window of 7 ms specified in the ANS paper and is slower than the 5-6 ms TensorFlow provides. Hence, the alternate design of multi-channel CNN was considered. The multi-channel CNN design attempts to reduce GMEM memory access by reading the input matrix once and instead of running all channel filters for that input matrix in parallel, it is done in a single stream with a single GPU kernel function call. This is done by having each GPU thread cache the overlapping convolution filter region of the input matrix. Using -Xptxas="-v" compile argument, the cache (static array defined in GPU function) is using registers as storage, as the "bytes cmem" is not affects but the "bytes stack frame" is. Then, each thread loops over the 64 convolution filters to calculate the convolution result using the cached input and filter weights (output[ch][offset] = loop(cache[i]*filter[i])). The convolution padding is still being handled the same, see Figure 9, with the filter region per thread element starts from elements index (x,y) shifted by padding (x-padLeft, y-padTop). The convolution then is performed sequentially, that is, in Figure 7, in the first convolution layer one GPU kernel is executed caching the input and processing all 64 output filters using the cached input. Then in layer 2 and 3, a single kernel function is executed to process each column, or all output channels per the same input channel. With serial execution of rows, or serial execution across input channels.

This design processes the full CNN in 13 ms, which is much faster than parallel CNN design. As well, shared memory doesn't seem to improve performance, unless the input matrix size is increased. The Figures below show a time breakdown for this CNN implementation, with running everything up to the end of the 3rd convolution layer (no flattening or linear layer) takes 10 ms or less, and that convolution layers 2 and 3 take the most time. As 13 ms is still outside the 7 ms window, another design option will need to be taken.



	Time Breakdown (ms)
Linear Layer	3.41478
Cov3	3.147721375
Maxpool2	0.241279625
Cov2	5.071044
Maxpool1	0.2777875
Cov1	0.6852745

Fig. 19. Multi-Channel CNN – Percent Time Breakdown

```
ug55:~/ECE1782GPU_Project% nvprof ./a.out
==13935== NVPROF is profiling process 13935, command: ./a.out
Total Time: 14
==13935== Profiling application: ./a.out
==13935== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   58.04%  6.4080ms       124  51.677us  34.560us  67.328us  device_CNN_Multi_v1_middle(
                   20.97%  2.3152ms       329  7.0370us     608ns  1.8508ms  [CUDA memcpy HtoD]
                   10.61%  1.1714ms         2  585.71us  15.264us  1.1562ms  linear(float*, float*, floa
                    4.20%  463.84us       128  3.6230us  3.2960us  4.4480us  max_pool_2d(float*, int, in
                    2.74%  302.82us         1  302.82us  302.82us  302.82us  device_CNN_Multi_v1_single(
                    1.71%  188.77us        64  2.9490us  2.8800us  3.2320us  flatten(float*, float*, int
                    0.94%  103.62us         2  51.808us  36.032us  67.584us  device_CNN_Multi_v1_last(in
                    0.77%  84.864us         2  42.432us  22.016us  62.848us  device_CNN_Multi_v1_first(i
                    0.01%  1.2800us         1  1.2800us  1.2800us  1.2800us  [CUDA memcpy DtoH]
      API calls:   62.00%  96.711ms         5  19.342ms  48.062us  94.592ms  cudaHostRegister
                   29.04%  45.291ms         2  22.645ms  370.29us  44.920ms  cudaDeviceReset
                    5.15%  8.0285ms       324  24.779us  2.6670us  74.257us  cudaMemcpyToSymbol
                    1.24%  1.9346ms         6  322.44us  6.6000us  1.8663ms  cudaMemcpy
                    1.03%  1.6023ms         6  267.04us  6.6720us  1.1720ms  cudaDeviceSynchronize
                    0.69%  1.0825ms       328  3.3000us  1.8150us  150.90us  cudaMalloc
                    0.59%  913.97us       323  2.8290us  2.2730us  30.680us  cudaLaunchKernel
                    0.09%  148.14us         1  148.14us  148.14us  148.14us  cuDeviceTotalMem
```

Fig. 20. Multi-Channel CNN perf Results

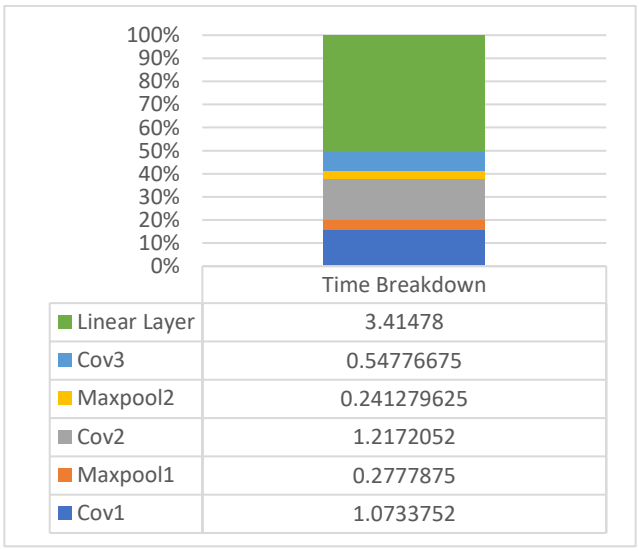
F. *Modified Covolution – Thrid Attempt*

A modified convolution implementation was attempted after completing the evaluation of the previous two methods, to attempt reducing memory reads of input for better performance. This method utilizes the same architecture of the past two methods but with an alternate convolution implementation for the GPU function. This convolution method performs one read per an input element in the input matrix, and then writes the value out to the appropriate location in the output matrix in GMEM. The appropriate position is based on the filter, that is the read value is written out to multiple positions in the output matrix as a single element is used by multiple filter windows, due to the stride being one. However, this method did not provide better performance. When integrated with the parallel CNN it gave 44 – 46 ms. When integrated with the multi-channel CNN it gave 76 – 77 ms. The #define Enable_Version3_Conv is used to enable this implementation in the code.

G. *GEMM-based Convolution*

The GEMM-based Convolution was the last thing we tried to improve the performance of our implementation, so we didn't have much time to complete all possible optimizations. The implementation is generic enough to run for any size of convolution on any device. We didn't have the time to integrate the full CNN network after implementing the Convolution with cuBLAS matrix multiplication. However, merging the timing results from the convolution layers of this implementation with the timing results seen for the other layers in the previous implementations, to see the potential performance gain if completed. This design could potentially process the full CNN in only 6.7 ms which is very close to the required timing window.

Figure 21 shows the time breakdown between different layers while Figure 22 shows the profiling results for the convolutions layers only. It's clear that the use of gemm from cuBLAS helped to reduce the timing performance with most time invested in the transform function to prepare the input modified matrix to be multiplied. Also, transform_image kernel is the function used to transform the input image into column and allocate input and output memories to be reused in next convolution layers.



	Time Breakdown
Linear Layer	3.41478
Cov3	0.54776675
Maxpool2	0.241279625
Cov2	1.2172052
Maxpool1	0.2777875
Cov1	1.0733752

Fig. 21. GEMM-based Convolution – Percent Time Breakdown

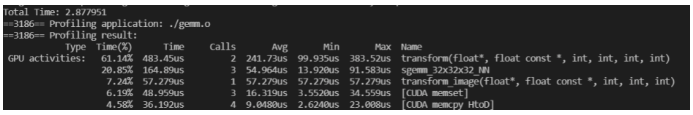


Fig. 22. GEMM-based Convolution perf Results

V. Related Work

Machine learning has grown in popularity because of improvements in hardware and availability of GPUs. GPUs have made machine learning algorithms feasible for execution, making model training using large datasets and large models with lots of parameters possible. Libraries like objax, TensorFlow, Keras, PyTorch have made machine learning programming more feasible to the masses. On the other hand, lower-level libraries like cuBLAS and cuDNN, for more optimized GPU utilization and API availability further made machine learning performance more available to the public and researchers. It should be noted that if cuBLAS and cuDNN are installed, Python and high-level libraries like Keras and TensorFlow will use these Cuda libraries to try and provide optimal performance when running functions calls like conv2d [10]. However, other libraries require users to specify using cuDNN (like Torch) [10].

As convolution, followed by linear layer, are the most performance, resources, and time heavy part of a CNN neural network architecture, it is important to optimize the convolution.

GPUs and these high-level ML libraries have become so popular that companies like Google provide remote hardware that can be used with them. Google Colab provides free and paid versions to run Jupyter notebooks, where you can run TensorFlow on a GPU. AWS is also a server solution if you want to provide a full system solution but is usually not the best option if you just want to develop and run models.

Researching how convolution is usually performed on GPUs, we found multiple algorithms that are used. The first and straight forward is to apply the convolution formula directly, However, there are more complex approaches to do convolution.

General Matrix to Matrix Multiplication (GEMM)-based algorithm transforms the input filter into an intermediate matrix where each row contains all filter elements flattened as 1D array. Inputs are also transformed into an intermediate matrix in a way that the dot product computed by GEMM of these two matrices generates the same output of the convolution formula. Then, a final transformation of the output matrix is done to put it in the appropriate format. The idea is that GEMM-based algorithms can benefit from highly optimized GEMM functions on GPUs from the cuBLAS library. The main downside of using such a method is the growing amount of memory usage due to the intermediate matrices. The large memory allocation is due to the redundant input elements to be stored due to the overlapping of filter locations. In [10], they state that cuDNN, if used, mitigates this memory efficiency issue by using "lazy tiles fetching". This method reads in fixed-sized submatrices of the generated large input and filter matrices and computes parts of the output matrix, computing tiles while fetching the next tile by using asynchronous memory copying from host to device. This method is stated to help hide memory latency associated with data transfers and making more utilization of the GPU. This GEMM solution relates to the popular im2col based solution

which turns the entire convolution into one large matrix multiplication, see figure below. This type of solution would benefit from shared memory as the input size becomes much larger. Hence, this may perform better than our implementation. We already tried to implement it and showed better performance, but we believe that it could be optimized more.

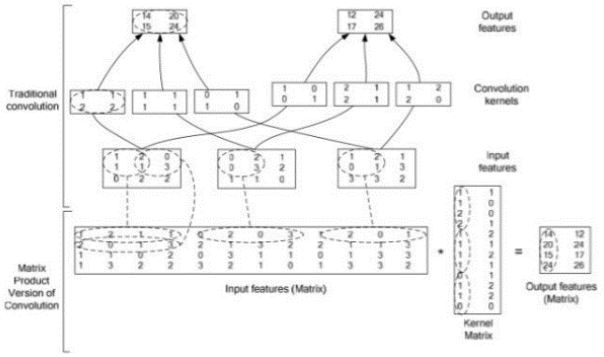


Fig. 23. GEMM based solution [10]

Winograd-based algorithms try to reduce the number of multiplications by doing some transformations described in [18]. The reduction in multiplications implies increase of additions which are less costly. The number of additions depends on the size of the input and increases quadratically with it which makes this algorithm suitable for only small input sizes. For large inputs, the Winograd algorithm divides the input into multiple tiles that overlap and compute the output for each of them before combining into the convolution output.

FFT-based convolution algorithms use the idea that a convolution in time is equivalent to point-wise multiplication in the frequency domain. So, FFT is done on the input matrices then they are point-wise multiplicated before doing inverse FFT to get the convolution output. Those transformations are too costly to implement only a single layer of convolution, but it may be useful for a long stack of convolutions. So, the potential of FFT-based convolution comes with larger number of inputs or filters.

CUDA Deep Neural Network library (cuDNN) is the reference library for deep neural network implementation on NVIDIA GPUs. In cuDNN, there are several implementations that could be called using the cudnnConvolutionForward API. The implementations available include the described famous algorithms (GEMM – Winograd and FFT). For the GEMM, there are 3 variants implemented which differ in how the temporary matrix is calculated either stored or computed on the fly which trades off memory with speed. In [19], they tested different convolution parameter configurations from the widely used CNNs using different cuDNN implementations. Their results show that selecting the best algorithm is totally dependent on the number of inputs and filter sizes and the GPU architecture and if it has optimized Tensor Cores or not. They provided a set of guidelines that we could have used if we have access to cuDNN libraries on lab GPUs because it makes it easier for developers to utilize the GPU without the need to implement and integrate with cuBLAS and asynchronous memory transfers and utilizing streams. Hence, it makes it easier to use and debug your system.

VI. Conclusion

Many challenges were faced when implementing the CNN inference model on CUDA. Some of the main issues we faced involved constant memory limitations when storing filter weights for convolution, parallel synchronization concerns, how to implement padding, and inaccessibility to information such as the original model code and weights. The constant memory challenge was overcome by only utilizing enough constant memory for the first layer (64 filter channels) and overwriting constant memory when executing the next layers. To avoid synchronization errors, parallelism was only used in relation to input or output channels. Padding was implemented by mathematically figuring out the elements needed for filter convolution shifted by padding and handling the matrix boundary with "if" condition checks. The result meant that input and output did not need to be padded, thus saving some allocation space. Shared memory was also utilized in an attempt to improve performance. Lastly, to overcome the challenge of not having access to the original model code and the weights used during inference, Google Colab and a local GPU on a group member's home machine was used to code and run a TensorFlow implementation, giving us a means of validating our implementation's output and comparing the performance result to that presented in the ANS paper. Self-generated weights were used, as the weight values themselves do not matter for our purposes of optimizing performance.

The two main approaches to implementing the CNN model were the CUDA parallel CNN, which gave a performance of 26-30 ms, and the more optimized attempt, CUDA Multi-Channel CNN, which gave a performance of around 13 ms. While this last approach was able to achieve the initial goal of improving from the FPGA implementation of 10 seconds per inference, our CUDA CNN on a GPU solution is still slower than the Keras Python implementation of 5-6 ms presented in the ANS paper but is still somewhat close. However, an attempt at utilizing GEMM/im2col was also performed for only the convolution layers. This GEMM approach wasn't fully extended to the entire CNN model (only was able to complete the convolution layers due to time). However, for the convolution layers (1, 2, 3) it gave a time of around 2.6 - 2.7 ms. Adding the other CNN layers timing to the intermediate results with the GEMM solution, this solution, when complete, could potentially provide the necessary timing of around 6.7 ms which is pretty much in the 5-6 ms, less than 7ms window, range needed. The GEMM solution then, not only provides the necessary timing for the electrode classifying, but also performs very close to that of Keras (3-8 ms) and Colab (4-30ms).

Future Work

There are still some design choices available for optimizing the CNN. GPU memory could be better utilized by storing convolution filter weights in GMEM and then copied to shared memory, instead of being limited by constant memory size. Maybe texture memory could be utilized to get around the constant memory size limitation. The GEMM convolution could be extended to the full CNN to utilizing im2col and cuBLAS or utilizing latest GPU technology with tensor cores and cuDNN. Developing a more optimized Keras/Tensorflow program, or even making use of Cuda warp-level primitives. Warp-level primitives like "__shfl_down_sync" allow for data exchange between threads in a warp without the need for shared memory. Hence,

each thread could read an element from the input matrix, to convolution, and then shift the value to the appropriate thread in the filter region so that the convolution can be performed without each thread reading its entire filter region in the input. That is, warp-level primitives will reduce the same elements being read in by different threads (duplicate work) or even the need for shared memory.

Additionally, after having the final optimized CNN design, implementing the server solution will also have to be completed. The server will take requests from multiple clients and will need to process them as quickly as possible. Maybe, depending on if the same sensor readings occur overtime, results can be cached to improve response times. CNN pipeline or parallelization will have to be investigated, depending on resource limitation of GPU. Even cost, maybe it's cheaper to pay for access to Cloud GPU or solution versus buy our own GPU's and servers, though it's probably cheaper, safer/secure (medical data), and less network/machine latency when having our own dedicated hardware.

CODE REPOSITORY

The source code is currently in a private repo, so to view it we attached the code with this report.

Git:
https://github.com/CoreyUWK/ECE1782GPU_Project.git

Google Colab CNN Model:
- ECE1782_Project_CNN_Model.ipynb
- ece1782_project_cnn_model.py

Convolution Weights and Constant Memory:
- cnn_weights.cu

Cuda Parallel CNN:
- cnnParallel.cu

Cuda Multi-Channel CNN:
- convolutionMulti.cu

Modified Convolution:
- #define Enable_Version3_Conv in cnnParallel.cu and convolutionMulti.cu

GEMM-based convolution:
- gemm_conv.cu

Utilities and Misc.:
- utils.cu
- simple.cu
- relu.cu
- max_pool_2d.cu
- fullyConnectedLayer.cu

Output Results:
- cov1_out.txt
- cov2_out.txt
- cov3_out.txt

In the parallel and multi-channel CNN implementations there are the following defines than can be enabled or played with; PRINTDATA, SHMEM, Free_Memory, INPUT_WIDTH, INPUT_HEIGHT, STRIDE, ENABLE_LINEAR_LAYER.

REFERENCES

[1] R. G. L. Koh, M. Balas, A. I. Nachman, and J. Zariffa, "Selective peripheral nerve recordings from nerve cuff electrodes using convolutional neural networks," *J Neural Eng*, vol. 17, no. 1, p. 016042, Jan. 2020, doi: 10.1088/1741-2552/ab4ac4.

[2] Whitepaper Nvidia GeForce GTX 980. Retrived on December 22, 2021 from https://www.microway.com/download/whitepaper/NVIDIA_Maxwell_GM204_Architecture_Whitepaper.pdf

[3] What is the difference between 'SAME' and 'VALID' padding in tf.nn.max_pool of tensorflow?. Retrieved on December 22, 2021 from https://stackoverflow.com/questions/37674306/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-t

[4] CS231n Convolutionl Neural Networks for Visual Recognition. Retrieved on December 22, 2021 from https://cs231n.github.io/convolutional-networks/

[5] Sumit Saha. 2018. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. Retrieved December 22, 2021 from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[6] A. Zhang, Z. Lipton, M. Li, A. Smola, et al. Dive into Deep Learning: Convolution Networks Padding and Stride. Retrieved December 22, 2021 from https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html

[7] How to calculate optimal zero padding for convolutional neural networks?. Retrived December 22, 2021 from https://stats.stackexchange.com/questions/297678/how-to-calculate-optimal-zero-padding-for-convolutional-neural-networks

[8] MMA. 2019. Implementing 'SAME' and 'VALID' padding of Tensorflow in Python. Retrived December 22, 2021 from https://mmuratarat.github.io/2019-01-17/implementing-padding-schemes-of-tensorflow-in-python

[9] C. Li, Y. Yang, M. Feng, S. Chakradhar, H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on GPUs," SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 1-12

[10] P. Lin. 2017. CUDA Optimization Design Tradeoff for Autonomous Driving. Retrieved December 22, 2021 from https://medium.com/@paichul/cuda-optimization-design-tradeoff-for-autonomous-driving-f2261ed258f1

[11] A. Shenoy. 2019. How Are Convolutions Actually Performed Under the Hood?. Retrieved December 22, 2021 from https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf

[12] CUDA implementation of convolution with im2col algorithm. Retrieved December 22, 2021 from https://github.com/piojanu/CUDA-im2col-conv

[13] G. Lu, W. Zhang and Z. Wang, "Optimizing GPU Memory Transactions for Convolution Operations," 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020, pp. 399-403, doi: 10.1109/CLUSTER49012.2020.00050. Retrieved December 22, 2021 from https://ieeexplore.ieee.org/abstract/document/9229640 and https://www.youtube.com/watch?v=zdsOMWW0Qso&t=481s

[14] K. Bai. 2019. A Comprehensive Introduction to Different Types of Convolutions in Deep Learning. Retrieved December 22, 2021 from https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

[15] L. Brown. 2015. cuDNN v2: Higher Performance for Deep Learning on GPUs. Retrieved December 22, 2021 from https://developer.nvidia.com/blog/cudnn-v2-higher-performance-deep-learning-gpus/

[16] M. Harris. 2012. How to Implement Performance Metrics in CUDA C/C++. Retrived December 22, 2021 from https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/

[17] 2019. Mixed Precision. Retrieved December 22, 2021 from https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/mixed_precision.ipynb and https://www.tensorflow.org/guide/mixed_precision

[18] S. Winograd, Arithmetic Complexity of Computations. Society for Industrial and Applied Mathematics, 1980. doi: 10.1137/1.9781611970364.

[19] M. Jordà, P. Valero-Lara and A. J. Peña, "Performance Evaluation of cuDNN Convolution Algorithms on NVIDIA Volta GPUs," in IEEE Access, vol. 7, pp. 70461-70473, 2019, doi: 10.1109/ACCESS.2019.2918851.

[20] P. Chen, M. Wahib, S. Takizawa, S. Matsuoka, "Pushing the Limits for 2D Convolution Computation On Cuda-enabled GPUs," HPC-163 Issue 22, 2018. Retrieved December 23, 2021 from https://ipsj.ixsq.nii.ac.jp/ej/?action=repository_action_common_download&item_id=186051&item_no=1&attribute_id=1&file_no=1

[21] Y. Lin, V. Grover. 2018. Using CUDA Warp-Level Primitives. Retrieved December 22, 2021 from https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[22] Prof. Mike Giles. Lecture 4: warp shuffles, and reduction / scan operations. Oxford University Mathematical Institute. Retrieved December 22, 2021 from https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec4.pdf

[23] M. Xu et al., "DeepCache: Principled Cache for Mobile Deep Vision," MobiCom 2018 Computer Vision and Pattern Recognition. https://arxiv.org/abs/1712.01670

[24] Lecture 11: Programming on GPUs (Part 1). Retireved December 22, 2021 from https://www3.nd.edu/~zxu2/acms60212-40212/Lec-11-GPU.pdf

[25] C. Cooper. Boston Univerisity: GPU Computing with CUDA Lecture 5 - More Optimizations. 2011. Retreived December 22, 2021 from https://www.bu.edu/pasi/files/2011/07/Lecture5.pdf