Parallel Convolutional Neural Networks
Final Project Report

Nathan Beauchamp, Rishi Thakkar, Melissa Jin
beaucha3, rrthakk2, mqjin2
Group Name: Fore Oh Ate
ECE 408 Fall 2016

I. Abstract.

The Convolutional Neural Network (CNN), a deep learning algorithm with many applications in machine learning and image processing, is very computation-intensive and highly parallel in nature. As such, it is well-suited for GPU acceleration using Nvidia's CUDA framework. Many types of parallel CNN implementations are possible, but of those tested, a matrix-multiplication-based convolution layer implementation featuring asynchronous data transfers and high compute-to-global-memory-access ratio was found to have the best performance.

II. Introduction.

Machine learning is an emergent computing field that seeks to develop dynamic algorithms whose logic can be gleaned from input datasets. In particular, machine learning is useful in applications for which it is impossible or infeasible to design a standard algorithm to do the job. Such applications often involve large input data such as audio or image signals. In order to classify the features of these data using a conventional mapping algorithm, a prohibitively large amount of logic would be necessary. However, machine learning solves this problem by detecting patterns in the input data and using these patterns to train the application logic. Two subcategories of classification/pattern detection algorithms exist. In the first subcategory, the programmer has to manually define the data features relevant for pattern detection (Hwu & Kirk, 2016, p. 366). However, in the second subcategory, the machine-learning program will automatically transform the raw data, gleaning the features necessary for classification. This is referred to as "deep learning" and is often done in a hierarchical manner, with the raw data being transformed into a more abstract representation at each step (Hwu & Kirk, 2016, p. 366). This transformation can be visualized as a “feed-forward” network, with the transformation at each step being dependent on previous results for training datasets. An example feed-forward network is shown in Figure 1 below.

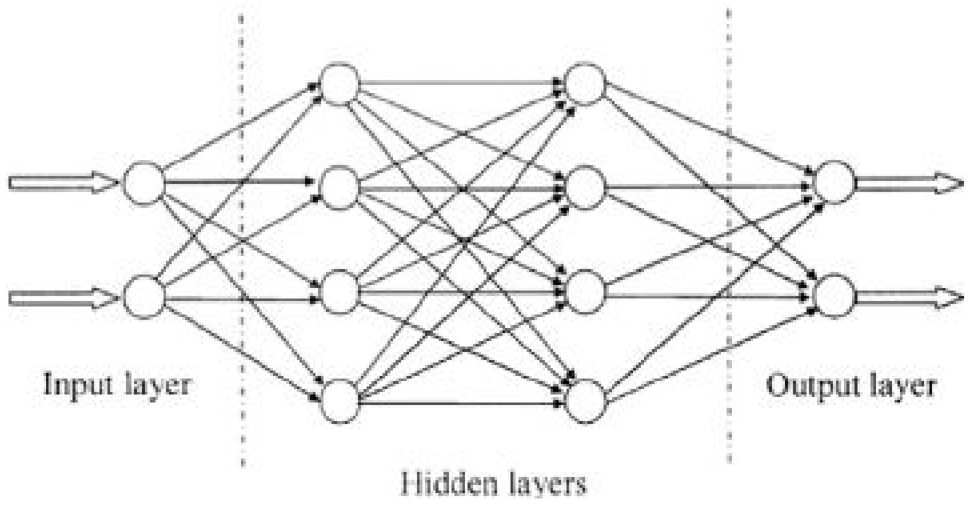


Figure 1: An example feed-forward network (Ata, 2015, p. 536).

For our final project, we examined a specific deep learning algorithm called the Convolutional Neural Network (CNN) and its applications in handwritten digit recognition. The CNN is a hierarchical feature extractor consisting of convolution layers, subsampling layers, and full connection layers. One of the most well-known neural networks for handwritten digit classification is LeNet-5, a CNN whose forward step has a total of seven layers (Hwu & Kirk, 2016, p. 368). A graphic depicting the layers of this CNN is shown in Figure 2 below.

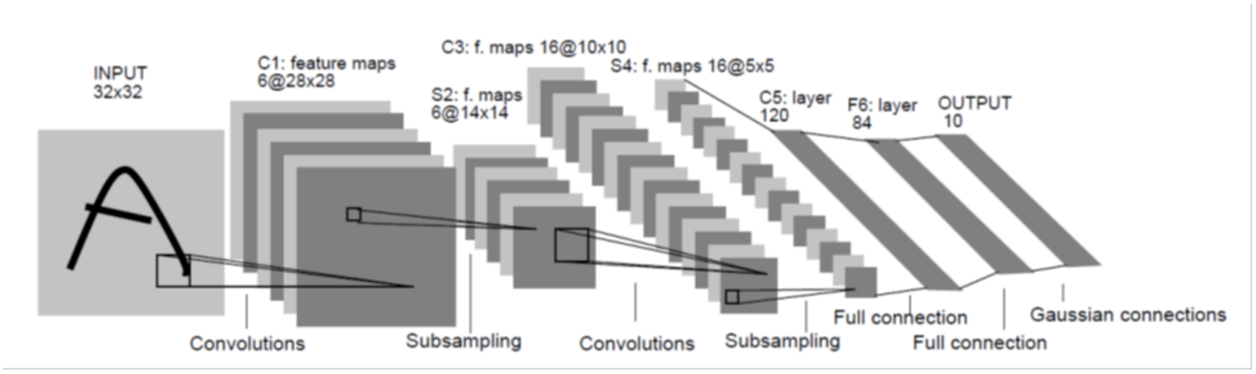


Figure 2: Layers of the forward operation of the LeNet-5 CNN (Hwu & Kirk, 2016, p. 368).

Each step (or layer) of a CNN involves applying some operation to one or more input feature maps and producing one or more output feature maps as a result. In LeNet-5, we begin with a 32x32 input image and convolve it with a filter bank to produce a certain number of 28x28 output feature maps. The weights contained within the filter bank are determined by training the network on a great number of reference datasets. After convolving, we subsample (pool) each result in order to average neighboring pixels, getting 14x14 feature maps as a result. We then repeat both steps once to get 5x5 feature maps. For the version of LeNet-5 shown in Figure 2, six feature maps are produced on the first iteration, and 16 are produced on the second iteration. However, for the version we worked with, 32 feature maps are produced on the first iteration, and 64 are produced on the second iteration. Next, these feature maps are passed through a "fully connected" layer that utilizes matrix multiplication to connect each output to all inputs. As the final step of LeNet-5, a Gaussian layer is applied to generate a 10-element vector, with each element representing the probability that the input image contains a particular digit.

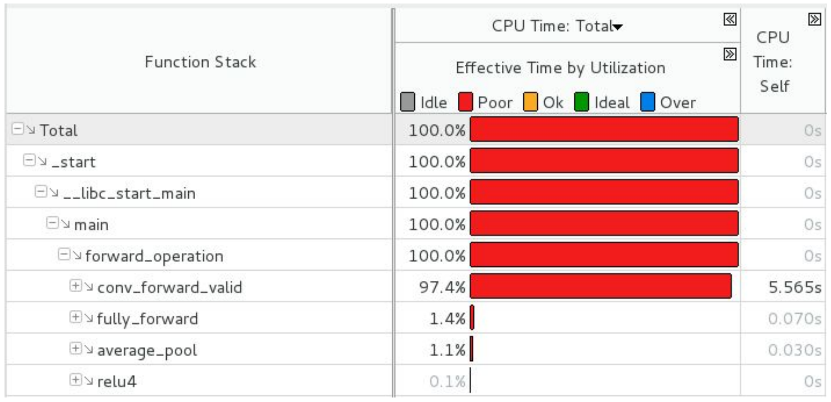
As can be seen from the previous description and from Figure 2, convolutional neural networks entail a large number of steps that produce progressively larger numbers of outputs.

Furthermore, a large amount of work is done by each layer in the process of producing its results. This is magnified significantly if many input images are run through the CNN in a batch process. These effects, along with the intrinsic parallelism of the convolution step, make the CNN quite suitable for GPU acceleration. In the following sections, we will discuss the pitfalls of the sequential CNN implementation and how these can be improved by implementing the CNN using work-efficient CUDA kernels executing in parallel on a GPU.

III. Critical Sections of Sequential CNN.

As we have seen, CNNs are very useful in solving problems that are hard to describe and solve algorithmically. The only problem with them is the amount of time they can require to run. Due to the large number of internal layers and nodes, the sequential code is not scalable to large sample sets. To understand what parts of the sequential code needed to be parallelized, we decided to profile the sequential code using Intel's processor performance analysis tool called VTune. Through this, we were able to find the hotspots[1] that are taking up the most time during the execution and then target those sections as we started our parallel implementation. Figure 3 below shows the results of profiling the sequential code. As we can see, a staggering 97.3% of the runtime is spent within the conv_forward_valid function. This means that the convolution layer is clearly the bottleneck when it comes to the execution speed. This means that if we don't decrease the runtime of the convolution layer, we won't be able to see much of a speed up even if we parallelize the rest of the code. Due to this, we decided that our base parallel implementation should parallelize the conv_forward_valid function (which will be discussed in the next section).

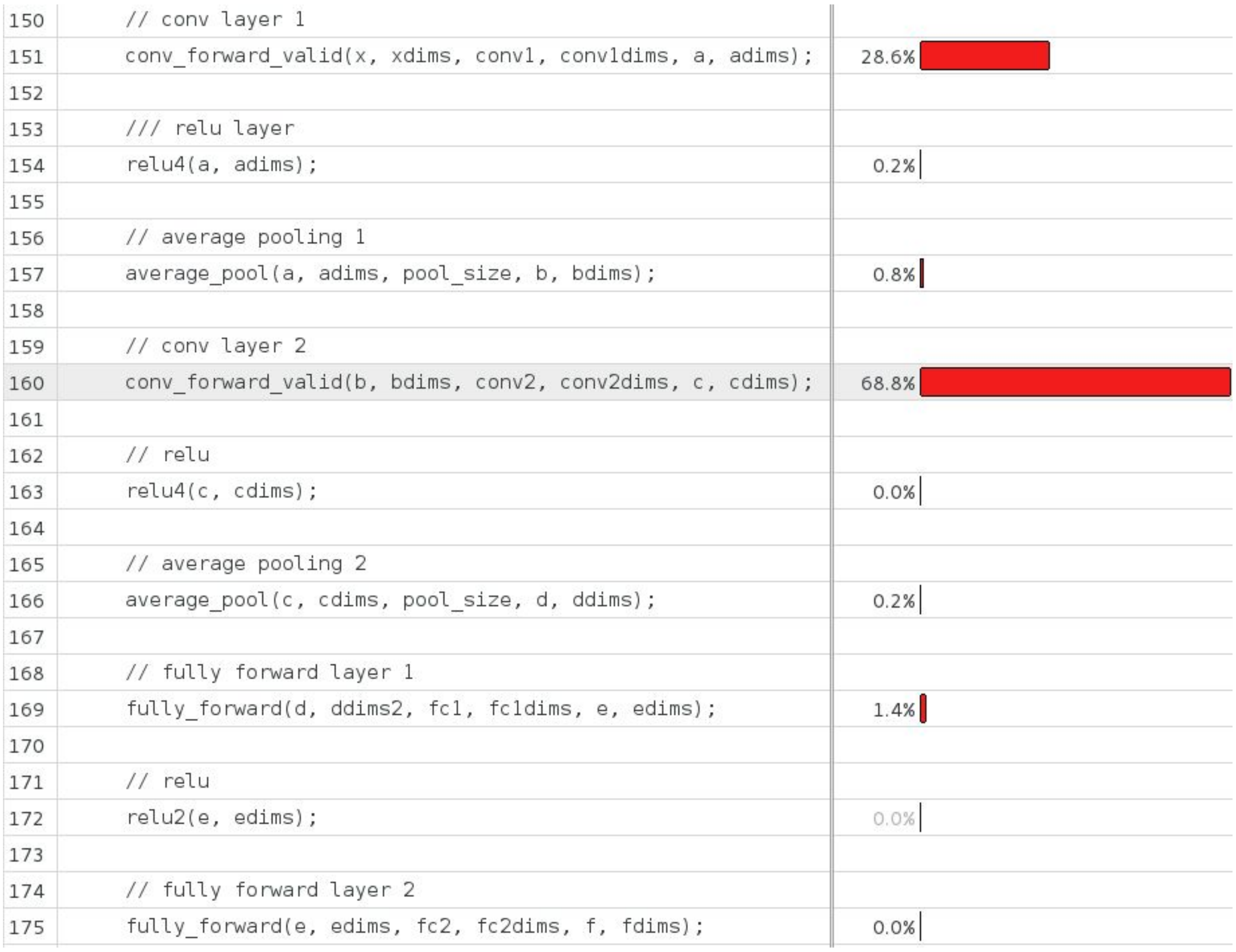
[1] Hotspots are sections of code that incur a large execution time.



Function Stack	CPU Time: Total — Effective Time by Utilization (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	CPU Time: Self
Total	100.0%	0s
_start	100.0%	0s
__libc_start_main	100.0%	0s
main	100.0%	0s
forward_operation	100.0%	0s
conv_forward_valid	97.4%	5.565s
fully_forward	1.4%	0.070s
average_pool	1.1%	0.030s
relu4	0.1%	0s

Figure 3: Top-down runtime tree of sequential code

Another important realization that came out of profiling the sequential code is the division of work between the layers. Figure 4 below clearly shows that the second convolutional layer does the most amount of work (68.8%). Based on this, we realized that we needed to design a highly work-efficient convolution kernel that will work well with variable input sizes. We also realized that we needed to carefully consider the impact of memory bandwidth and how we would handle the copying back and forth of memory. This will be discussed in more detail in later sections.



```
    // conv layer 1
    conv_forward_valid(x, xdims, conv1, conv1dims, a, adims);       28.6%

    /// relu layer
    relu4(a, adims);                                                  0.2%

    // average pooling 1
    average_pool(a, adims, pool_size, b, bdims);                      0.8%

    // conv layer 2
    conv_forward_valid(b, bdims, conv2, conv2dims, c, cdims);       68.8%

    // relu
    relu4(c, cdims);                                                  0.0%

    // average pooling 2
    average_pool(c, cdims, pool_size, d, ddims);                      0.2%

    // fully forward layer 1
    fully_forward(d, ddims2, fc1, fc1dims, e, edims);                 1.4%

    // relu
    relu2(e, edims);                                                  0.0%

    // fully forward layer 2
    fully_forward(e, edims, fc2, fc2dims, f, fdims);                  0.0%
```

Figure 4: Division of work within the forward_operation function.

IV. **Base Parallel Implementation.**

Throughout the duration of the final project, we adopted a modular approach to work in which we incrementally applied optimizations, carefully evaluating the results before moving forward. Because the conv_forward_valid function is the primary computation bottleneck of the forward operation of a CNN, we decided to first create a base parallel implementation that only parallelizes the forward convolutional layer. The following is a summary of our considerations and design methodology that we followed when creating the basic CUDA implementation. For each convolution step, assuming that we have a total of N samples, M output feature maps (each with height H_{out} and width W_{out}), and C input feature maps, we will have a total of $M * C$ filters in our filter bank. Each filter will have height K and width K. This implies that there will be $N * M * H_{out} * W_{out}$ output elements in total, each of which requires $C * K * K$ multiply-add operations to compute. This lends itself well to a simple thread mapping in which each thread handles the computation of a single output element. With this mapping, we will launch many threads (at least $N * M * H_{out} * W_{out}$), and each thread will be doing a fair amount of computation. This enables us to achieve both a high degree of parallelism and a large compute-to-bandwidth ratio. It was also relatively straightforward to translate the given sequential code into the base parallel implementation; the only challenging part was the development of a per-thread indexing scheme. However, as this basic kernel is a straightforward translation, it does not utilize advanced CUDA programming techniques such as shared memory or streams. Also, because adjacent threads map to adjacent elements in the same input feature map- despite the fact that the input data is stored such that the input feature map is the lowest order dimension- the kernel does not achieve good memory coalescing. As a result, this implementation is fairly inefficient in terms of global memory bandwidth. Exact profiling results from NVPROF on a batch size of $N = 100$ are shown below.

Figure 5 shows the timeline for a given run of the base implementation. Since in this version of the code we have only developed a kernel for the convolutional layer, NVPROF is only able to record information about the execution of this one function. The information recorded matches what we saw in the sequential implementation, as the second call to the convolutional layer function seems to take the most amount of time to run. Even though this implementation has its downfalls, which we will cover next, it is definitely better than the sequential implementation as it is able achieve an approximate 5x speedup from 10 seconds to 2.2 seconds. The image also shows some of the problems that exist within this kernel. We can graphically see that there are no overlapping sections of activity in either computation or memory copies. This means that at most we are using ⅓ of the GPU's resources at a given time.

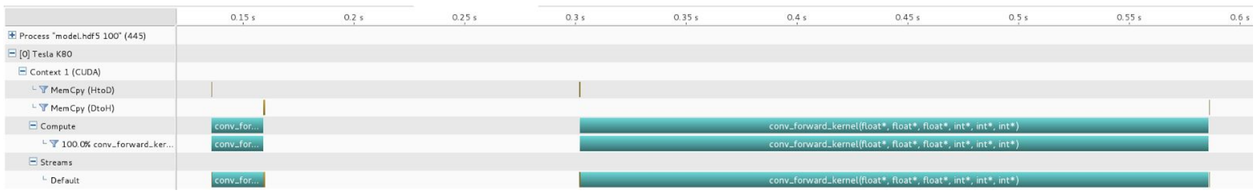


Figure 5: Sample run of base parallel implementation of CNN.

Also, if we take look at the some of the runtime details of the GPU in the table below, we can see that due to the structure of the kernel, we are utilizing the GPU resources very inefficiently. The table below shows that on average our global memory load efficiency over the two kernel calls is 19.2 % and our average global memory store efficiency is 12.5%. Both of these are horrendously low values and indicate that there is close to zero memory coalescing for our kernel. Thus, we are not even remotely making use of the global memory bandwidth of the GPU. There is also a clear problem with control divergence. With a average warp execution efficiency of 62.9%, we can clearly see that almost 40% of the GPU's resources are being wasted. Even though this version sees a fairly large and scalable speed up when compared to the sequential code, there are some clear issues which we went on to address in future implementations.

	Global Memory Load Efficiency	Global Memory Store Efficiency	Warp Execution Efficiency
Convolution Layer 1	25.9%	12.5%	77.2%
Convolution Layer 2	12.5%	12.5%	50.3%

V. Optimization I- Parallelizing Other Components of the CNN.

In the previous section, we identified that global memory throughput had become the bottleneck for the performance of our convolution kernel. However, as we aimed to make incremental improvements to our design, we decided to first parallelize most of the remaining components of the convolutional neural network before attempting to revise our convolution kernel. Namely, we created parallel implementations for the average_pool, relu4, and relu2 functions. We decided to not parallelize the argmax function as it doesn't perform enough computation to warrant GPU acceleration. Also, although the fully_forward function certainly deserves GPU acceleration[2], we decided to wait to parallelize it until we incorporated the unrolling/matrix multiplication optimization into our convolution kernel. This is because the fully_forward function performs matrix multiplication, which we would write for this later optimization in any case. Overall, since the average pool function has a high degree of parallelism and performs multiple computations to generate each output element (effectively, it averages pool_size $\times$ pool_size blocks of input elements), it is a natural candidate for parallelization. The relu functions also possess a high degree of parallelism as they simply loop over every input element and change negative elements to zeros as they are found. As expected,

[2] Fully_forward performs matrix multiplication, which benefits substantially from parallelization.

parallelizing these functions led to some performance increase; however, as they are not the execution bottleneck, this increase was not incredibly significant. Exact profiling results from NVPROF on a batch size of $N = 100$ are shown below.

The image below (Figure 6) shows a sample run for the first optimization on the CNN base parallel code. As stated above, this was an intermediate implementation which was used as a stepping stone for future implementations to build upon. Due to this, we didn't make any major changes to the convolution kernel. Instead we just parallelized a few of the other layers of the CNN. Based on the profiling of the sequential code, we knew that the bottleneck for the code is the convolution function and since this wasn't really modified, there wasn't a very large speedup during this optimization. We saw an average speed up of about 1.1 times from approximately 2.2 seconds to 2 seconds. The key difference between figure 5 and figure 6 is that the empty spots between calls to conv_forward_kernel have been filled with GPU executions of the intermediate layer kernels.

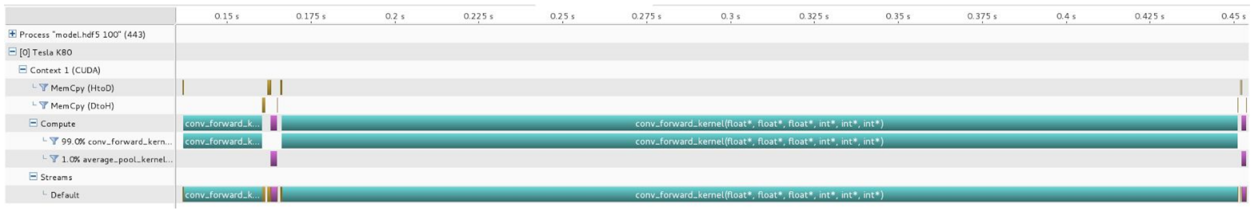


Figure 6: Sample run for the first optimization of the parallel CNN

Since we didn't really change the structure of the code during this optimization, we didn't end up fixing any of the problems we had acknowledged in the base implementation. Due to this, the average global memory load efficiency was 15.9 %, average global memory store efficiency was 12.5%, and the warp execution efficiency was a 64.6%. These are approximately the same as the they were before and so these problems continued to persist into the next version where we then saw some improvement.

VI. Optimization II- Shared Memory in Convolution Kernel.

Next, since the convolution kernel still remained the bottleneck for the CNN calculation, we decided to optimize it by incorporating the use of shared memory. Data in shared memory is stored in on-chip memory, which has relatively low (on the order of five clock cycles) access latency; in contrast, the global memory is implemented with DRAM, which suffers from long latencies consisting of hundreds of clock cycles and congested access paths (Hwu & Kirk, 2016, p. 71). If multiple threads need to use an element in global memory, then it would be beneficial to read the element once and store it in shared memory for other threads in the same block to access. The benefit of shared memory usage can be quantified by analyzing the reduction in the total number of global memory accesses on a per-block basis. If $(TILE\_SIZE + mask\_width - 1)^2$ input elements are loaded into shared memory- along with a total of $mask\_width^2$ filter elements- then a total of $2 * TILE\_SIZE * mask\_width^2$ global memory accesses are replaced by shared memory accesses (accounting for both input and filter elements). Accounting for the global memory accesses necessary for loading elements into shared memory, the total reduction in the number of global memory accesses per block is $2 * TILE\_SIZE * mask\_width^2 - (TILE\_SIZE + mask\_width - 1)^2 - mask\_width^2$. Although this optimization was not used for the final version of the code, the use of shared memory did provide a significant performance speedup from the base implementation. A quantitative analysis of said speedup is provided below, along with profiling results from NVPROF on a batch size of $N = 100$.

The sample run for optimization 2, as shown in figure 7, presents a similar runtime trajectory to that of optimization 1. The major difference now is the structure of the convolutional kernel which uses a shared memory implementation. Through the use of this, the overall code sped up by a factor of 1.25 on average, increasing from 2 seconds to 1.6 seconds. Though this difference does not seem that large for a batch size of 100, this implementation is scalable. Due to this, the difference for a batch size of 10000 is much more noticeable (from 50 seconds to 19 seconds). This scaling feature is what makes the shared memory implementation such a feasible option. This decrease in execution time is also seen if you compare the timestamps between figure 7 and figure 6. Even though these timestamps are technically incorrect due to the profiling overhead, they show the relative increase in speed between the two versions.

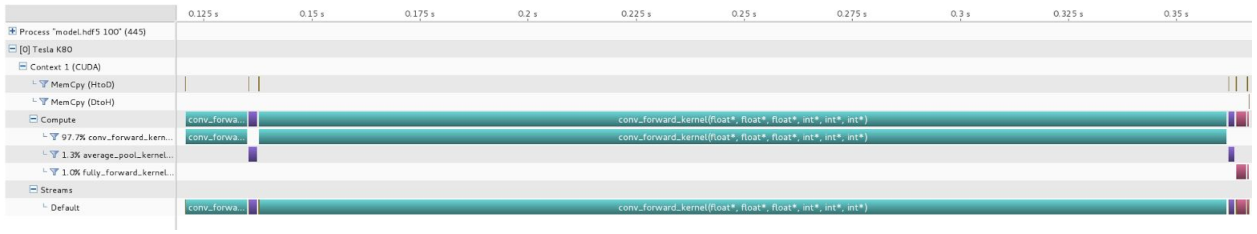

0.125 s
0.15 s
0.175 s
0.2 s
0.225 s
0.25 s
0.275 s
0.3 s
0.325 s
0.35 s
Process "model.hdf5 100" (445)
[0] Tesla K80
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
97.7% conv_forward_kern...
1.3% average_pool_kernel...
1.0% fully_forward_kernel...
Streams
Default
conv_forwa...
conv_forward_kernel(float*, float*, float*, int*, int*, int*)

Figure 7: Sample run of shared memory implementation of parallel CNN.

In this version, we tackled some of the issues that have existed since implementing the base parallel code. The problems with the base parallel code were the lack of efficiency of the global memory and the control divergence that inherently existed within the structure of the code. By assigning the block and grid dimensions more intelligently, we were able to increase the warp execution efficiency to 70% for the convolution layer. The way we handled the problem of global memory is very interesting. Since we aren't changing the indexing scheme drastically, the access to global memory is still just as inefficient. However, since we are using shared memory, the number of times that we have to access global memory directly decreases and this in the end decreases the congestion of the bus. While the shared memory implementation is quite good, it is still no match for the unrolling/matrix multiplication kernel which we will discuss in Section VIII.

VII. Optimization III- Optimizing CUDA Memory Copies.

Third, we decided to optimize the efficiency of our calls to cudaMemcpy. When performing a kernel call, we typically have to copy data from the host to the input device memory before the kernel call, and then copy the output back from the device to the host after the kernel call. Since each layer in the CNN requires at least one kernel call, we would be performing memory copies to the device and back to the host for each layer. We realized that the input to each successive kernel is the output of the previous kernel. Therefore, we decided to eliminate the memory copies in between kernel calls and simply pass the pointers to the device data. This allowed us to simply copy memory from the host to the device once before the first kernel launch and copy memory from the device back to the host after the final kernel has completed execution. Because this optimization did not involve an algorithmic transformation, we did not find it necessary to profile it separately using NVPROF. Since we are simply reducing the amount of work we do copying back and forth, we are certain to observe speedup-

and in fact, we observed a speedup of approximately 100 ms for a batch size of $N = 100$ (a reasonably significant performance increase).

VIII. Optimization IV- Matrix Multiplication for Convolution Layer.

Even after incorporating shared memory into the convolution kernel, it still proved to be the bottleneck of the program. Consequently, we decided to try another optimization by using matrix multiplication to implement the convolution layer. Previously, we launched one kernel to perform the entire convolution layer, using each thread to generate one output element of an output feature map for a particular sample. However, by reformulating the convolution problem as a matrix multiplication problem, we can get a significant performance increase by using the efficient tiled matrix multiplication algorithm. With tiled matrix multiplication, we will get increased memory coalescing and less control divergence when compared to tiled convolution. The matrix multiplication algorithm is also well suited for GPU computation because it is highly parallel and because it utilizes a large number of floating point operations relative to its global memory accesses. To implement this technique, we had to modify the format of the input data and mask to the convolution layer through a process called "unrolling." In the process of unrolling, we rearranged and duplicated elements from the data in the input matrix, placing the results into a matrix with each column holding the values necessary to compute output elements at a given position within an output feature map. We also needed to rearrange the filter bank data so that each row holds the weights for one output feature map. The unrolling and matrix multiplication process is shown in Figure 8 below.

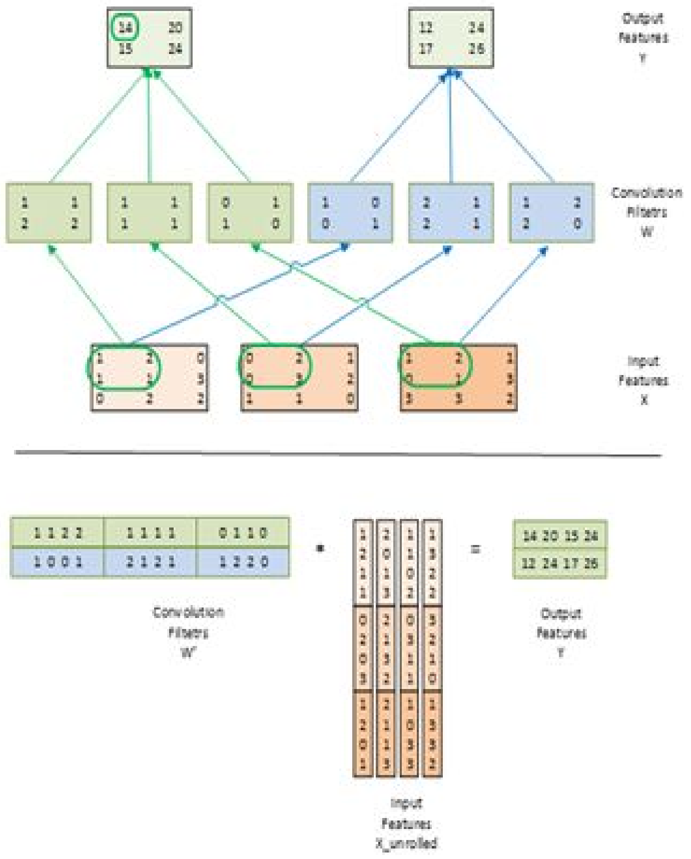


Figure 8: Convolution through Matrix Multiplication (Hwu & Kirk, 2016, p. 382).

Then, because our remaining kernels expect feature map data arranged such that the map index corresponds to the lowest order dimension, we had to rearrange the output feature map data produced by the tiled matrix multiplication to match their specifications. We wrote separate kernels to perform each of the three previously described data transformations. Ultimately, none of the unrolling/data reorganization procedures incurred significant overhead relative to the amount of time spent computing; therefore, the relative efficiency of tiled matrix multiplication to convolution resulted in a significant performance increase for our overall program. Runtime results and NVPROF profiling support this, as shown below.

Figure 9 represents the runtime timeline of a sample from optimization 4. This is the optimization where we saw another major stride from one generation to another. As explained above, the unroll kernel and a tiled Matrix Multiplication kernel combined can be a lot faster than any other implementation we have explored so far. In optimization 2, we saw a great speed up because we were using shared memory to reduce our access to global memory. However, our accesses to global memory were still very inefficient. Matrix multiplication plays a very important role here because due to its very structure, its global memory accesses will be coalesced. Due to this, we get a combination of coalesced and reduced memory accesses. Also when compared to a convolution operation, a matrix multiplication operation has a lot less control divergence. So in general it makes a lot better use of the GPU hardware. Based on all of these reasons, we saw a speedup of 3.2x from 1.6 seconds to 500 ms. This speed up is clearly seen when you compare the relative timestamps of figure 9 and figure 7.

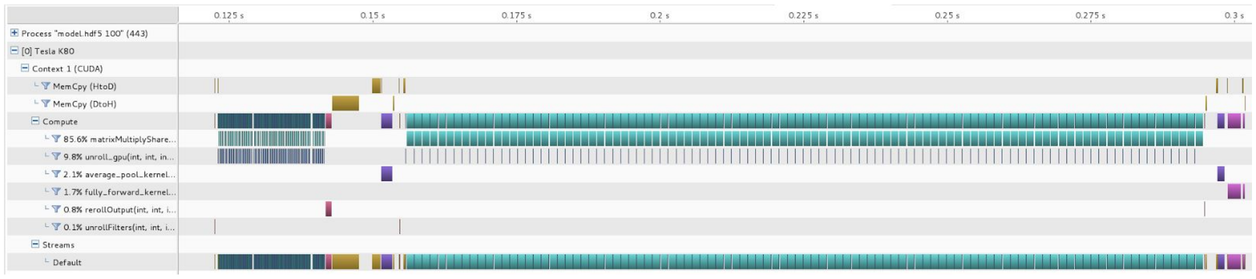

0.125 s
0.15 s
0.175 s
0.2 s
0.225 s
0.25 s
0.275 s
0.3 s
Process "model.hdf5 100" (443)
[0] Tesla K80
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
85.6% matrixMultiplyShare...
9.8% unroll_gpu(int, int, in...
2.1% average_pool_kernel...
1.7% fully_forward_kernel...
0.8% rerollOutput(int, int, i...
0.1% unrollFilters(int, int, i...
Streams
Default

Figure 9: Sample run of unroll and matrix multiplication kernel

The speedup seen above is also quite prominent in GPU statistics provided by nvprof. On average, the matrix multiplication kernel had an average global memory load efficiency of 89.05 %, an average global memory store efficiency of 100%, and an average warp execution efficiency of 96%. On the other hand, the unroll kernel had an average global memory load efficiency of 44.15 %, an average global memory store efficiency of 100%, and an average warp execution efficiency of 100%. The load efficiency is reduced for the unroll kernel because, unlike for the matrix multiplication kernel, the unroll kernel fails to achieve well-coalesced memory access. However, because the majority of the execution time is spent in the matrix multiplication kernel, this is not a huge performance-limiting factor. In general, these averages for the convolution layer are a lot better than we have seen for any of the implementations so far. The only thing left to do at this point is to add streams and try to achieve task parallelism.

IX. <u>Optimization V- CUDA Streams and Asynchronous Memory Copies.</u>

For our fifth optimization, we used multiple CUDA streams to handle our unroll_gpu and matrixMultiply kernel calls, thus allowing the CPU and GPU to work in parallel. A total of three CUDA streams were active, each responsible for copying over the input data and then invoking a kernel to either unroll input element data into a matrix or perform matrix multiplication. Before this optimization was implemented, one call to the convolution kernel was made only after all the input data was transferred from the host to the device memory. Performing the data transfer then the kernel call serially results in the GPU remaining idle while the CPU is transferring the data and the CPU to remain idle while the GPU is performing the computations. The implementation of multiple streams allows for one stream to be copying data on the CPU while another stream performs the kernel call for the convolution.

This optimization was somewhat helpful, resulting in a speedup of about 300 ms. We did not see a huge speedup after implementing this optimization because it only supplies a benefit to the first convolution layer. Because the first convolutional layer is the first step in the forward operation of the CNN, it is the only one that involves host-to-device memory copies. As a result, it is the only step that benefits from asynchronous memory copies and concurrency between memory copies and kernel execution. As before, we generated profiling results for this code using NVPROF; these are shown below.

A key feature in the implementation of optimization 4 is that the unroll and matrix multiplication operations are broken up with respect to each sample. This means that stratified calls are made to unroll and then matrix multiplication for each input sample. This implies that this structure is a perfect candidate for the use of streams. Due to this, we decided to have each iteration of the for-loop perform kernel calls for three different input samples, each in a different stream queue. For the first convolutional layer, we will also need to perform memory copies from the host to the device. This will be done asynchronously; the fact that we use three streams per host function call comes in handy here as it allows us to a partial pipelined timing. This timing is shown below in Figure 10, and the stream utilization is shown in Figure 11. There are some discrepancies, however, because you do not need to copy back the output since the device pointer itself can be passed to the next function. This in the best case will result in us make use of 2/3 of the GPU hardware. Based on this implementation, we are able to see a very large speed up of 9.09x from 500 ms to 55 ms. The unfortunate thing is that this is not scalable and so as input size increases, the runtimes for these two asymptotically approach a constant difference from each other.

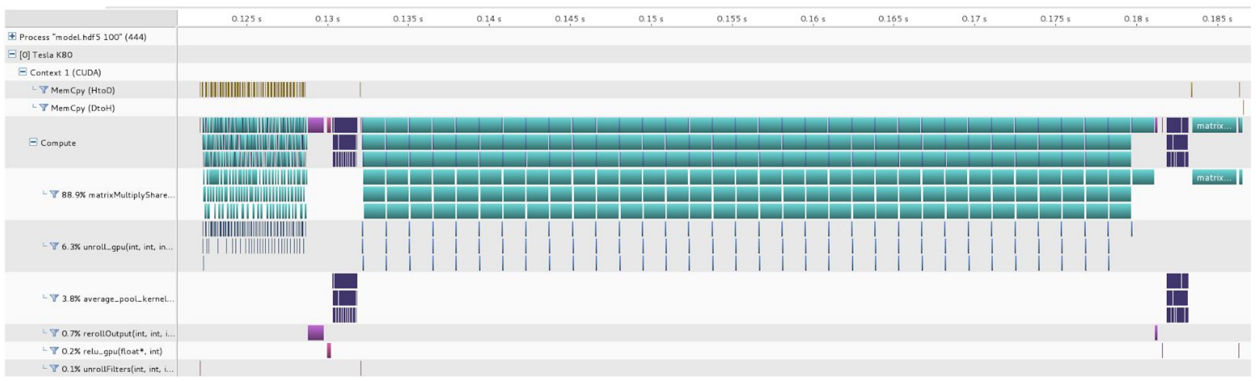

0.125 s
0.13 s
0.135 s
0.14 s
0.145 s
0.15 s
0.155 s
0.16 s
0.165 s
0.17 s
0.175 s
0.18 s
0.185 s
Process "model.hdf5 100" (444)
[0] Tesla K80
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
88.9% matrixMultiplyShare...
6.3% unroll_gpu(int, int, in...
3.8% average_pool_kernel...
0.7% rerollOutput(int, int, i...
0.2% relu_gpu(float*, int)
0.1% unrollFilters(int, int, i...
matrix...

Figure 10: Sample run of unroll and matrix multiplication kernel with streams

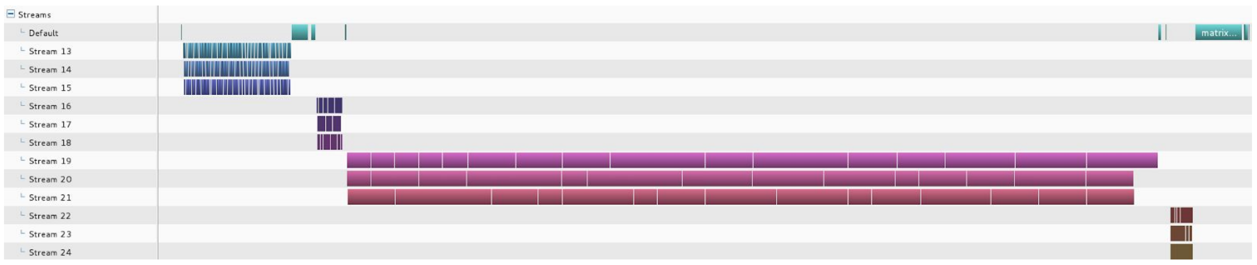


Figure 11: Stream utilization of the program in Figure 10

Compared to the implementation without streams, the global memory load and store efficiencies remain the same, which makes sense as we are not modifying our overall access pattern. In fact, essentially all memory metrics are the same between the two implementations; however, as we are utilizing streams and asynchronous memory copies in optimization 5, memory operations are simply performed earlier. This leads to an overall decrease in execution time.

Compared to the base parallel implementation, we achieved a speedup of 181.81x from 10 seconds to 55 ms on a 100-sample input. We also achieved a great increase in compute-to-global-memory-access ratio, as shown by Figures 12 and 13 below.

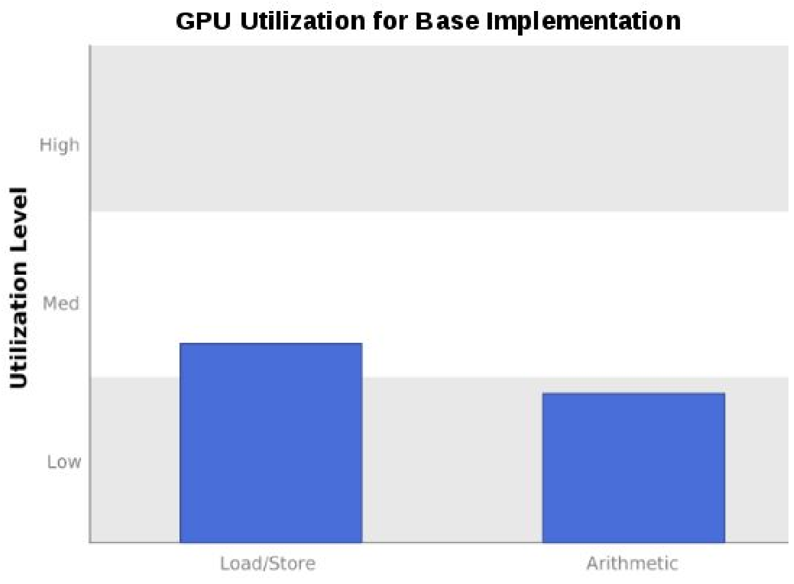


Figure 12: GPU Utilization for our base parallel implementation.

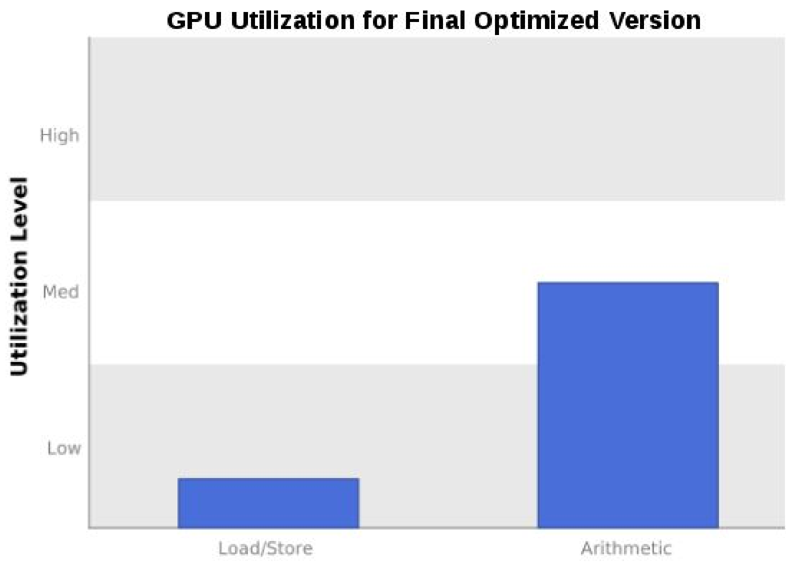


Figure 13: GPU Utilization for our optimized parallel code.

X. Compiler Flag Modifications.

Not all optimizations have to consist of algorithmic transformations or code purifications. In some cases, the execution speed of a given piece of code depending on how the compiler is equipped to optimize it. After several tests, we decided to add the following two NVCC compiler options to our CMakeLists.txt: **list(APPEND CUDA_NVCC_FLAGS -Xptxas --verbose)**, and **list(APPEND CUDA_NVCC_FLAGS -O3)**. The first compiler option is not performance-related; rather, it serves to display useful compilation information for the developer. This was included for our own edification. The second compiler option serves as an optimization flag; this tells the compiler to perform optimizations on the code during the compilation process. Specifically, this flag tells the compiler to optimize the code for execution time, even if more memory is used in the process. This flag worked quite well for us, cutting our code's execution time approximately in half. We also decided to remove the following line from our CMakeLists.txt: set(CUDA_NVCC_FLAGS_DEBUG ${CUDA_NVCC_FLAGS_DEBUG} "-G"). This line tells the compiler to generate debug symbol information and retain it in the executable file. This information is useful for profiling such as that done in NVPROF; however, due to overhead created by the debugging symbols, it can be a severe performance limiter. In fact, removing this flag led to our code's execution speed again increasing by a factor of 2.

XI. Other Attempted Optimizations.

Throughout the course of our work on the final project, we attempted a variety of optimizations in an effort to improve performance. As shown in the sections above, many of these were successful and improved performance by a measurable amount. However, some of the optimizations we tried failed to significantly improve performance, or did not work in general. This section provides a brief overview of some of our attempted optimizations that did not end up influencing our final product.

First, we attempted to use CUDA streams to copy the data back from the device to the host after the convolution kernel. However, we determined that this operation was pointless and would not result in any decrease in the execution time because pointers to the output data could simply be passed between kernels. Passing pointers is definitely the superior approach as it involves a simple four-byte assignment rather than sequences of many-kilobyte memory copies.

Another optimization we attempted was the use of shared memory in the average pooling kernel. Loading data elements into shared memory is beneficial as long as said data elements are accessed multiple times during the computation. Upon further analysis, we realized that the elements in the input array are accessed exactly once; therefore, if they were loaded into the shared memory, they would never be used again by another thread. As a result, using shared memory in the average pooling kernel did not result in an improvement in the kernel's global memory bandwidth; in fact, it slightly slowed the kernel's execution speed.

Third, due to the simplicity of the relu2 and relu4 functions, we tried optimizing the code by performing their operations inside other kernels to reduce the number of kernel calls. The purpose of these functions is to replace any negative numbers output by the convolution and fully_forward steps of the forward operation process with zeros. Therefore, incorporating this operation into the convolution kernel (or, later, into the tiled matrix multiplication kernel) should have been a straightforward task. However, when we tried this, we noticed that the correctness value inexplicably deviated from the expected result for the 10000-sample set[3]. As a result, we decided to instead write separate kernels for the relu2 and relu4 functions; this resulted in the expected correctness while also providing a slight performance increase over the sequential code.

Next, we attempted to use constant memory to store our convolution filters. This was motivated by several factors. First, the filter values do not change over the course of the forward operation and each filter value is used in the computation of a number of output elements; as a result, placing the filter elements in constant memory would save a lot of unnecessary global memory accesses. Furthermore, as determined via a device query, the Tesla K80 GPU contains 65,536 Bytes of constant memory, which is enough space to hold the filters required for the first convolution step (those contain $C * M * K * K * 4\frac{bytes}{float} = 1 * 32 * 5 * 5 * 4 = 3200$ Bytes in total). However, because we have to reorganize the filter data in order for our unroll/matrix multiplication implementation to work correctly, it makes no sense to use constant memory for the filters. This is because we would eventually have to copy the values into global memory anyways as constant memory contents cannot be determined dynamically during a kernel launch.

Finally, we attempted to use half-precision floats in order to increase the speed of our floating-point computations. However, the GPU that we have access to through RAI is a Tesla K80; although this GPU supports the half-precision floating point datatype for compression purposes, it does not support computation using half-precision floats. Such functionality was first introduced in 2016 with CUDA 8 and Nvidia's Pascal generation of GPUs (Harris). As a result, we were unable to effectively implement this optimization. Had we had access to a supported GPU, we would have been able to successfully implement this optimization.

[3] For this particular sample set, we got a correctness of 0.8727 instead of the expected 0.8722.

Furthermore, as two half-precision floating point operations can be performed in parallel by a single execution core, the performance of our code could have effectively doubled.

XII. Conclusion.

In conclusion, our final parallel implementation of the Convolutional Neural Network was able to achieve a great performance boost relative to the sequential code and to previous iterations of our parallel code. This was achieved primarily due to the incorporation of the efficient unrolling/matrix multiplication implementation of the forward convolutional layer. Other factors that played a role in the performance boost included advanced CUDA concepts such as streams and asynchronous memory copies. Overall, each team member contributed a lot to the group effort, and the diversity of optimizations that we were able to consider would not have been possible without this. The distribution of work was approximately equal, with each team member's work constituting 33.33% of the total. The three of us all collaborated on the initial and shared-memory implementations; after that, we generally worked on separate optimizations with each person putting in equal amounts of work. For example, Nathan worked on the unrolling, Melissa worked on streams, and Rishi worked on profiling using NVPROF. At the end, the three of us synthesized our results, compiled a final version of the code, and collaborated to write the final report.

XIII. **References.**

Ata, R. (2015). Artificial neural networks applications in wind energy systems: a review. *Renewable and Sustainable Energy Reviews*, *49*, 534-562. http://dx.doi.org/10.1016/j.rser.2015.04.166.

Harris, M. (2016, October 19). Mixed-Precision Programming with CUDA 8 [Blog post]. Retrieved from Parallel For All website: https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/.

Hwu, W.-M. W., & Kirk, D. B. (2016). Application case study: Machine learning. In W.-M. W. Hwu & D. B. Kirk (Authors), *Programming massively parallel processors: A hands-on approach* (3rd ed., pp. 345-368) [PDF]. Retrieved from https://wiki.illinois.edu/wiki/display/ece408f16/Book+Chapters?preview=/602518692/603851747/3rd-Edition-Chapter16-case-study-DNN-FINAL.pdf.