# ECE Final Project Report

**FALL 2016**

Arnav Agarwal
Karan Usgaonkar
Saksham Saini

## Introduction:

Convolutional Neural Networks is one of the many ways that Machine Learning is implemented. Machine Learning is a subsection of computer science that is gaining a great deal of traction in recent years because it enables machines the ability to learn without being explicitly programmed. Machine learning becomes incredibly useful to solve problems where an explicitly programmed algorithm is unfeasible. Some examples would be spam filtering, network security scans, search engines, computer vision, and more. A prime example to intuitively understand machine learning is the way Amazon makes recommendations for potential products you might be interested in based on the items you’ve purchased/looked at previously.

## Project Focus:

In this particular project, we are targeting the performance of our algorithm to complete the forward propagation step (shown in Figure No.1) of the Convolutional Neural Network algorithm through parallelization and subsequent optimizations.

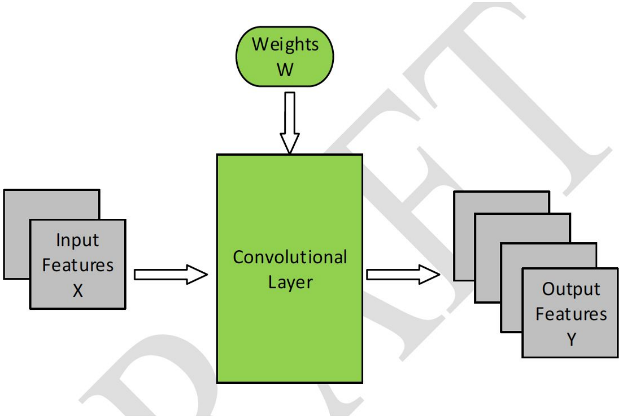


Figure No. 1: The Forward Propagation Step of a Convolutional Layer.

All images from course book or course slides - © David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016

## In-depth analysis of the problem:

The project was to write an improved version of the given serial code - the forward propagation step of a convolutional neural network. Since the task at hand involves dealing with a lot of data, most of which can be parallelized, the solution to the problem was to implement the project in CUDA. Since CUDA uses Graphic Processing Units (GPUs) for computation as compared to serial code, which uses CPUs, several computations can be carried out on the data simultaneously instead of synchronously. GPUs have several ALUs, thereby enabling it to perform computations on larger amounts of data at a time as compared to CPUs. Therefore, it can be used to increase the performance of the task at hand manifolds.

Since there can be several approaches to implementing the forward layer of a CNN, our main aim was to find the best solution to improve the speed of the program for large amounts of data. To implement an algorithm which would give us maximum benefit of using parallelization, understanding and analyzing the layout of data is important. Since GPU's don't have as much memory storage as CPUs, the main concern while parallel programming is to make the most use of the DRAM bursts(taking care of memory coalescing), so as to reduce the accesses to global memory and therefore performing computations at much faster speeds without having a backlog due to the constant memory accessed.

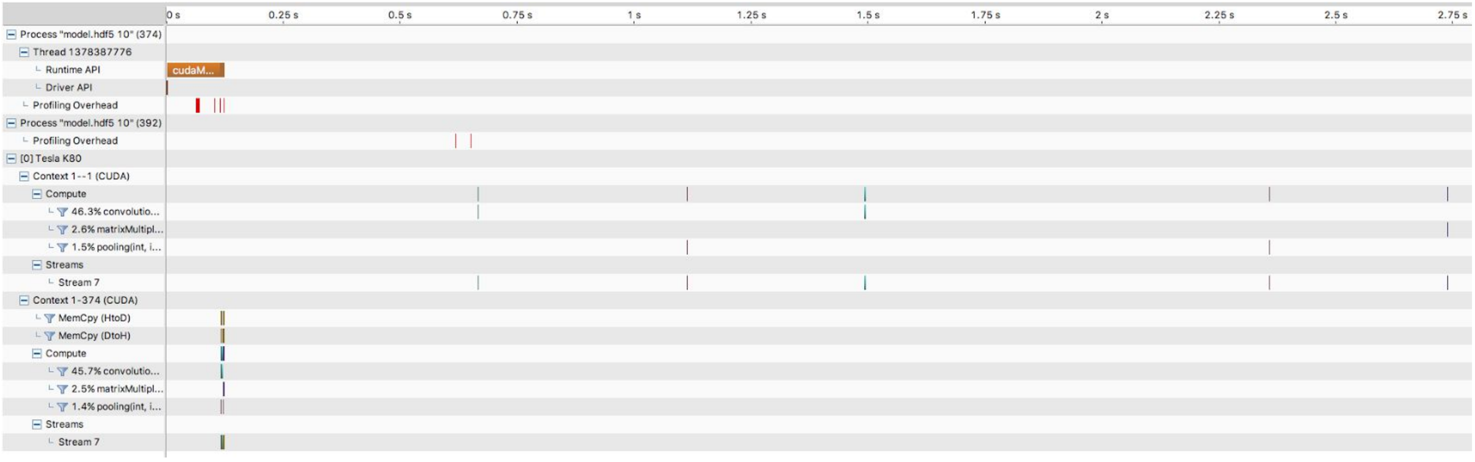
## The approach:

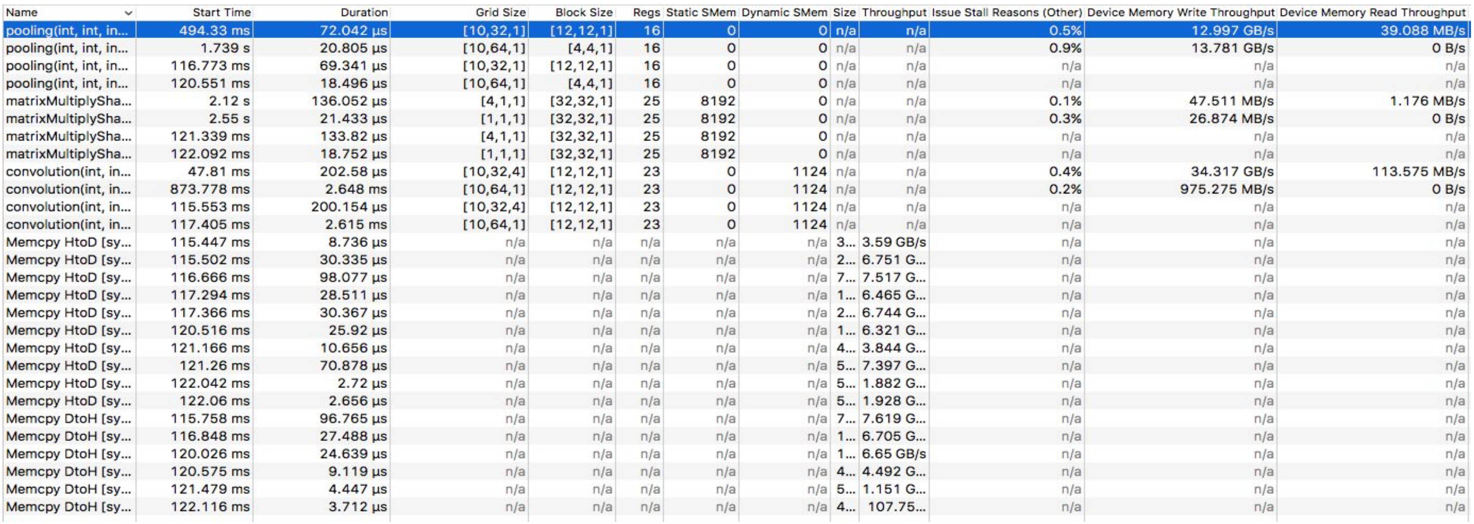
To achieve our goal, we started out with understanding the serial code and algorithms to implement the task at hand. Our first implementation was to use the convolution technique described in our textbook (Figure 16.13) to modify the conv_forward_valid function of the given code. This modification gave us a significantly better performance that the serial code. The next step we took was to parallelize all the remaining functions, namely - average_pool, relu4, relu2, fully_forward and argmax. Our implementation was to understand the serial code and transform it as it is. Parallelization of relu4, relu2 and argmax did not give any significant improvement and were therefore discarded, though the parallel code for average_pool and fully_forward gave a decent boost.

Our next step was to improve on our performance even more by reducing the convolution to a matrix multiplication as described in section 16.4 of our textbook. Using the algorithm provided, we unrolled the input and filters to two matrices to be multiplied to give the resulting output, which is what we would get using a normal convolution. The matrix multiplication was done using shared memory to make most use of data already read. Also, since there was a huge amount of data, we used the streaming feature of CUDA to get the most out of our implementation. On using the new approach, our speed was increased almost 6x as compared to the normal convolution approach.

# NVProf and NVVP Results:

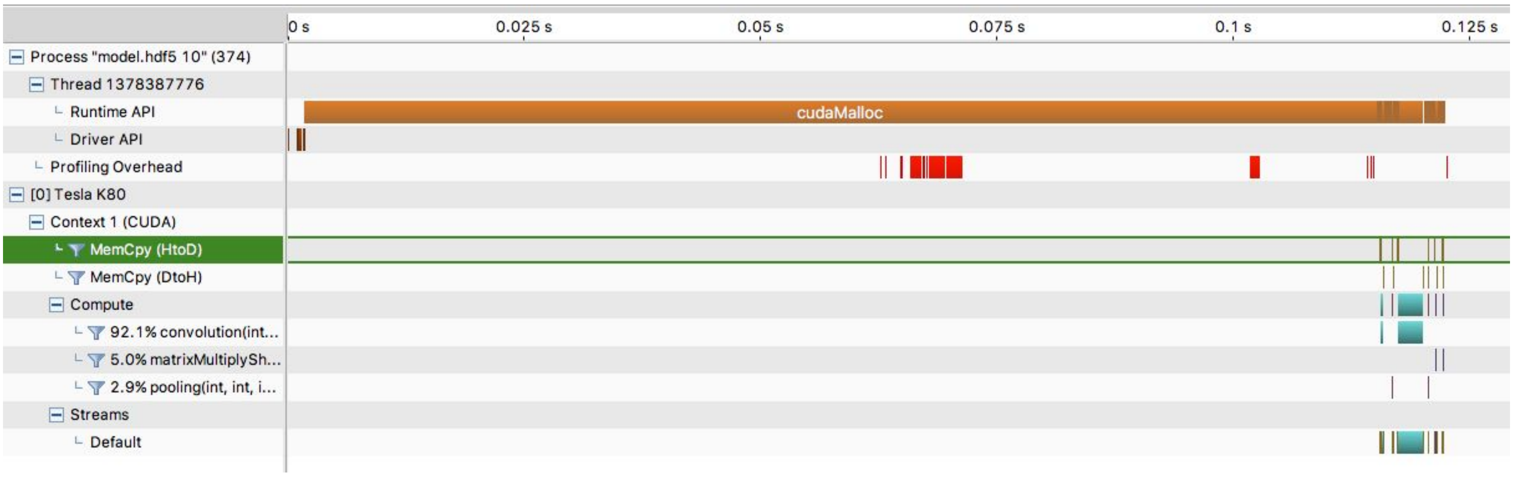
## Analysis NVProf



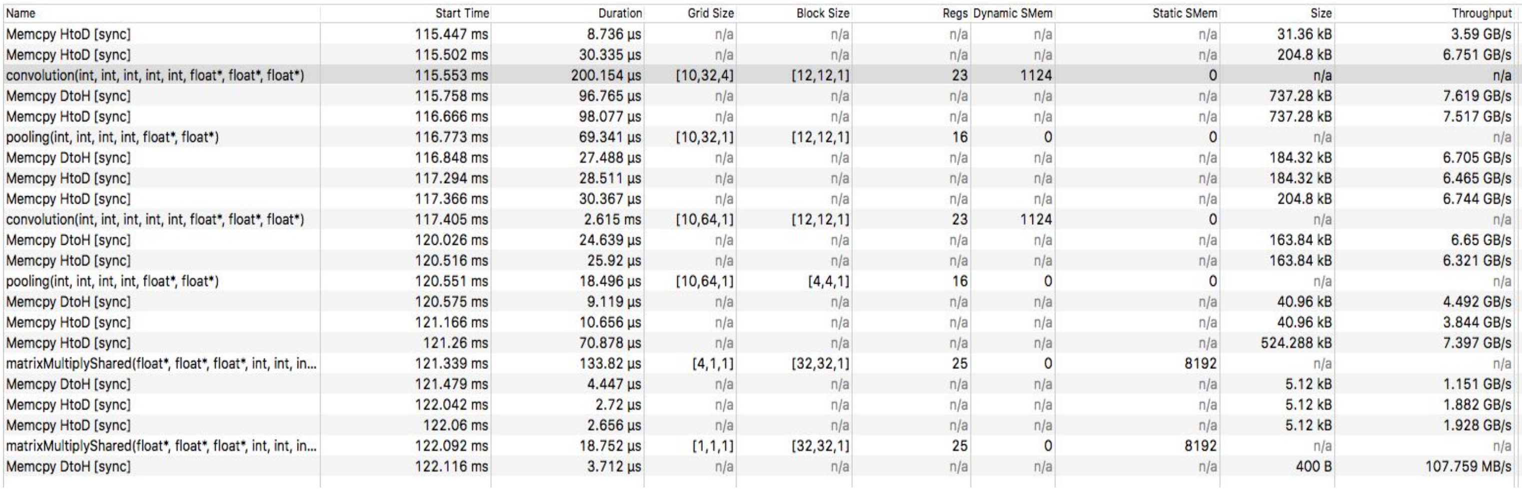


| Name | Start Time | Duration | Grid Size | Block Size | Regs | Static SMem | Dynamic SMem | Size | Throughput | Issue Stall Reasons (Other) | Device Memory Write Throughput | Device Memory Read Throughput |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pooling(int, int, in... | 494.33 ms | 72.042 µs | [10,32,1] | [12,12,1] | 16 | 0 | 0 | n/a | n/a | 0.5% | 12.997 GB/s | 39.088 MB/s |
| pooling(int, int, in... | 1.739 s | 20.805 µs | [10,64,1] | [4,4,1] | 16 | 0 | 0 | n/a | n/a | 0.9% | 13.781 GB/s | 0 B/s |
| pooling(int, int, in... | 116.773 ms | 69.341 µs | [10,32,1] | [12,12,1] | 16 | 0 | 0 | n/a | n/a | n/a | n/a | n/a |
| pooling(int, int, in... | 120.551 ms | 18.496 µs | [10,64,1] | [4,4,1] | 16 | 0 | 0 | n/a | n/a | n/a | n/a | n/a |
| matrixMultiplySha... | 2.12 s | 136.052 µs | [4,1,1] | [32,32,1] | 25 | 8192 | 0 | n/a | n/a | 0.1% | 47.511 MB/s | 1.176 MB/s |
| matrixMultiplySha... | 2.55 s | 21.433 µs | [1,1,1] | [32,32,1] | 25 | 8192 | 0 | n/a | n/a | 0.3% | 26.874 MB/s | 0 B/s |
| matrixMultiplySha... | 121.339 ms | 133.82 µs | [4,1,1] | [32,32,1] | 25 | 8192 | 0 | n/a | n/a | n/a | n/a | n/a |
| matrixMultiplySha... | 122.092 ms | 18.752 µs | [1,1,1] | [32,32,1] | 25 | 8192 | 0 | n/a | n/a | n/a | n/a | n/a |
| convolution(int, in... | 47.81 ms | 202.58 µs | [10,32,4] | [12,12,1] | 23 | 0 | 1124 | n/a | n/a | 0.4% | 34.317 GB/s | 113.575 MB/s |
| convolution(int, in... | 873.778 ms | 2.648 ms | [10,64,1] | [12,12,1] | 23 | 0 | 1124 | n/a | n/a | 0.2% | 975.275 MB/s | 0 B/s |
| convolution(int, in... | 115.553 ms | 200.154 µs | [10,32,4] | [12,12,1] | 23 | 0 | 1124 | n/a | n/a | n/a | n/a | n/a |
| convolution(int, in... | 117.405 ms | 2.615 ms | [10,64,1] | [12,12,1] | 23 | 0 | 1124 | n/a | n/a | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 115.447 ms | 8.736 µs | n/a | n/a | n/a | n/a | n/a | 3... | 3.59 GB/s | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 115.502 ms | 30.335 µs | n/a | n/a | n/a | n/a | n/a | 2... | 6.751 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 116.666 ms | 98.077 µs | n/a | n/a | n/a | n/a | n/a | 7... | 7.517 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 117.294 ms | 28.511 µs | n/a | n/a | n/a | n/a | n/a | 1... | 6.465 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 117.366 ms | 30.367 µs | n/a | n/a | n/a | n/a | n/a | 2... | 6.744 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 120.516 ms | 25.92 µs | n/a | n/a | n/a | n/a | n/a | 1... | 6.321 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 121.166 ms | 10.656 µs | n/a | n/a | n/a | n/a | n/a | 4... | 3.844 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 121.26 ms | 70.878 µs | n/a | n/a | n/a | n/a | n/a | 5... | 7.397 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 122.042 ms | 2.72 µs | n/a | n/a | n/a | n/a | n/a | 5... | 1.882 G... | n/a | n/a | n/a |
| Memcpy HtoD [sy... | 122.06 ms | 2.656 µs | n/a | n/a | n/a | n/a | n/a | 5... | 1.928 G... | n/a | n/a | n/a |
| Memcpy DtoH [sy... | 115.758 ms | 96.765 µs | n/a | n/a | n/a | n/a | n/a | 7... | 7.619 G... | n/a | n/a | n/a |
| Memcpy DtoH [sy... | 116.848 ms | 27.488 µs | n/a | n/a | n/a | n/a | n/a | 1... | 6.705 G... | n/a | n/a | n/a |
| Memcpy DtoH [sy... | 120.026 ms | 24.639 µs | n/a | n/a | n/a | n/a | n/a | 1... | 6.65 GB/s | n/a | n/a | n/a |
| Memcpy DtoH [sy... | 120.575 ms | 9.119 µs | n/a | n/a | n/a | n/a | n/a | 4... | 4.492 G... | n/a | n/a | n/a |
| Memcpy DtoH [sy... | 121.479 ms | 4.447 µs | n/a | n/a | n/a | n/a | n/a | 5... | 1.151 G... | n/a | n/a | n/a |
| Memcpy DtoH [sy... | 122.116 ms | 3.712 µs | n/a | n/a | n/a | n/a | n/a | 4... | 107.75... | n/a | n/a | n/a |

## Timeline NVProf



All images from course book or course slides - © David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016



| Name | Start Time | Duration | Grid Size | Block Size | Regs | Dynamic SMem | Static SMem | Size | Throughput |
|---|---|---|---|---|---|---|---|---|---|
| Memcpy HtoD [sync] | 115.447 ms | 8.736 µs | n/a | n/a | n/a | n/a | n/a | 31.36 kB | 3.59 GB/s |
| Memcpy HtoD [sync] | 115.502 ms | 30.335 µs | n/a | n/a | n/a | n/a | n/a | 204.8 kB | 6.751 GB/s |
| convolution(int, int, int, int, int, float*, float*, float*) | 115.553 ms | 200.154 µs | [10,32,4] | [12,12,1] | 23 | 1124 | 0 | n/a | n/a |
| Memcpy DtoH [sync] | 115.758 ms | 96.765 µs | n/a | n/a | n/a | n/a | n/a | 737.28 kB | 7.619 GB/s |
| Memcpy HtoD [sync] | 116.666 ms | 98.077 µs | n/a | n/a | n/a | n/a | n/a | 737.28 kB | 7.517 GB/s |
| pooling(int, int, int, int, float*, float*) | 116.773 ms | 69.341 µs | [10,32,1] | [12,12,1] | 16 | 0 | 0 | n/a | n/a |
| Memcpy DtoH [sync] | 116.848 ms | 27.488 µs | n/a | n/a | n/a | n/a | n/a | 184.32 kB | 6.705 GB/s |
| Memcpy HtoD [sync] | 117.294 ms | 28.511 µs | n/a | n/a | n/a | n/a | n/a | 184.32 kB | 6.465 GB/s |
| Memcpy HtoD [sync] | 117.366 ms | 30.367 µs | n/a | n/a | n/a | n/a | n/a | 204.8 kB | 6.744 GB/s |
| convolution(int, int, int, int, int, float*, float*, float*) | 117.405 ms | 2.615 ms | [10,64,1] | [12,12,1] | 23 | 1124 | 0 | n/a | n/a |
| Memcpy DtoH [sync] | 120.026 ms | 24.639 µs | n/a | n/a | n/a | n/a | n/a | 163.84 kB | 6.65 GB/s |
| Memcpy HtoD [sync] | 120.516 ms | 25.92 µs | n/a | n/a | n/a | n/a | n/a | 163.84 kB | 6.321 GB/s |
| pooling(int, int, int, int, float*, float*) | 120.551 ms | 18.496 µs | [10,64,1] | [4,4,1] | 16 | 0 | 0 | n/a | n/a |
| Memcpy DtoH [sync] | 120.575 ms | 9.119 µs | n/a | n/a | n/a | n/a | n/a | 40.96 kB | 4.492 GB/s |
| Memcpy HtoD [sync] | 121.166 ms | 10.656 µs | n/a | n/a | n/a | n/a | n/a | 40.96 kB | 3.844 GB/s |
| Memcpy HtoD [sync] | 121.26 ms | 70.878 µs | n/a | n/a | n/a | n/a | n/a | 524.288 kB | 7.397 GB/s |
| matrixMultiplyShared(float*, float*, float*, int, int, in... | 121.339 ms | 133.82 µs | [4,1,1] | [32,32,1] | 25 | 0 | 8192 | n/a | n/a |
| Memcpy DtoH [sync] | 121.479 ms | 4.447 µs | n/a | n/a | n/a | n/a | n/a | 5.12 kB | 1.151 GB/s |
| Memcpy HtoD [sync] | 122.042 ms | 2.72 µs | n/a | n/a | n/a | n/a | n/a | 5.12 kB | 1.882 GB/s |
| Memcpy HtoD [sync] | 122.06 ms | 2.656 µs | n/a | n/a | n/a | n/a | n/a | 5.12 kB | 1.928 GB/s |
| matrixMultiplyShared(float*, float*, float*, int, int, in... | 122.092 ms | 18.752 µs | [1,1,1] | [32,32,1] | 25 | 0 | 8192 | n/a | n/a |
| Memcpy DtoH [sync] | 122.116 ms | 3.712 µs | n/a | n/a | n/a | n/a | n/a | 400 B | 107.759 MB/s |

As seen from the NVProf and NVVP results above, we found that allocating CUDA memory takes the longest time and therefore should be done together/as much together as possible. Another interesting thing that we found was that convolution took the most time after cudaMalloc, followed by pooling and then matrix multiplication. This helped us to streamline our process of optimization. We optimized convolution first, then pooling, followed by matrix multiplication in the end. This resulted in runtimes which were magnitude times better than the serial code which we were using earlier.

## Successful Optimizations:

### Optimization 1: Unrolling the Input Feature Mapping

Unrolling in general is the alteration of the input of the convolution kernel in such a way that all elements that needed to compute a single result/output will be stored as one sequential block. This changes the forward propagation step into a case of matrix multiplication (which we can optimize as well). At an abstract level, the usage of unrolling the input feature map involves rearranging input elements by concatenating the input features into a single matrix where each row will general one output feature. An example of this unrolling has been shown in Figure No. 4 below.

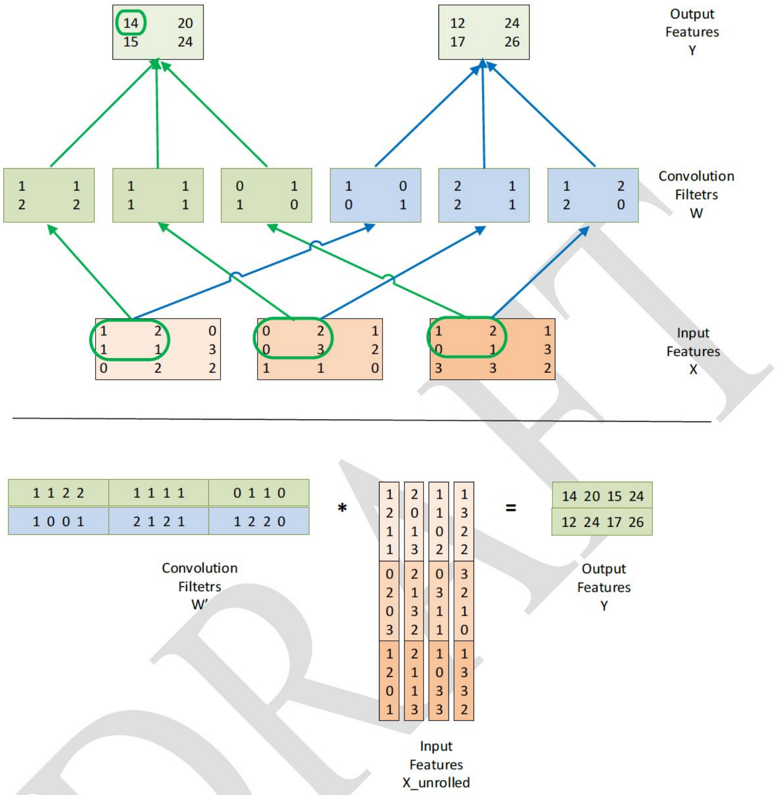


Figure No. 4: Example of Unrolling The Input Feature Mapping

The downside of this implementation is that there are several elements which need to be loaded into memory again and again for multiplication. This can be overcome by unrolling small pieces of the input data and multiplying it with the unrolled filter weights. This also leads to another problem, which is not making maximum use of the GPUs if the matrix multiplication is small, as it will not use the GPU to its full potential. Even though our implementation improved the performance a lot, this approach can have a lot more optimizations to improve the performance quite a bit.

All images from course book or course slides - © David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016

Optimization 2: Usage of CUDA Streams

CUDA streams in general enabled us to allow simultaneous execution of a kernel while copying data between the device and host memory. This in general allows us to increase the degree of parallelism, which improves runtime. More specifically, CUDA Streams enabled us to allow kernel execution while CudaMemCpy functions ran at the same time. At a high level, our style of usage of CUDA Streams was to divide the large dataset into segments and then overlap the transfers and computations of adjacent segments of the vector using multiple streams. Since each stream is structured as a queue, it helps in keeping steps organized to ensure that there is no computation that happens prematurely (i.e. before the relevant data is accessible by the threads that need the relevant data before making those computations).

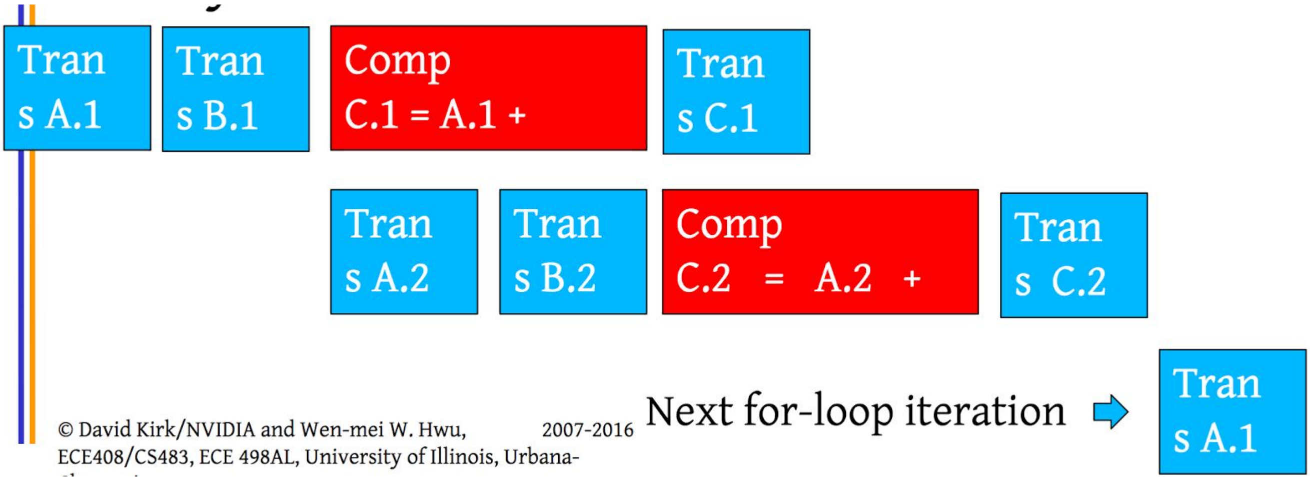


Figure No. 5: Diagrammatic Description of Overlapping Computation & Transfer Operations of Streams

All images from course book or course slides - © David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016

## Optimization 3: Shared Matrix Multiplication

Shared Matrix Multiplication helps by reducing the amount of time taken to obtain data directly from global memory. Obtaining data individually for every thread directly from global memory takes more time as compared to pulling a segment of input data into the block (using the block's shared memory). The shared memory is visible by all threads of that block. The time taken for threads to access data from shared memory is far lesser than to access data from global memory directly. Since the forward propagation step's performance is dependent on the performance of the style of matrix multiplication used, shared matrix multiplication was one of the optimizations used. The actual calculations made by each thread and a high level diagram of how the tiles are used to create the final result are illustrated below in Figure No. 6 and Figure No. 7 respectively.

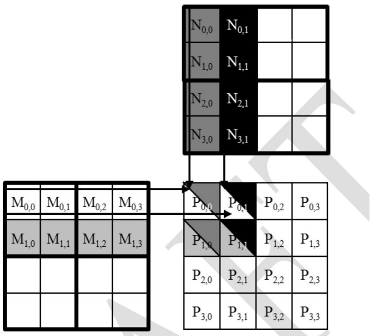
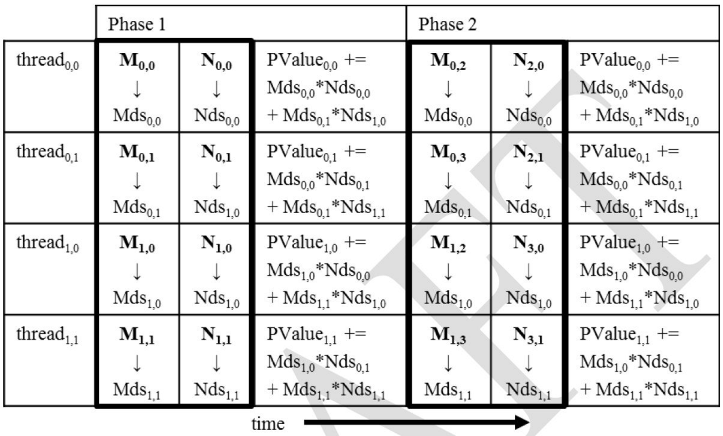

$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Figure No. 6: A Multiplication Substep for Tiled Matrix Multiplication



| | Phase 1 | | | Phase 2 | | |
|---|---|---|---|---|---|---|
| $\text{thread}_{0,0}$ | $\mathbf{M_{0,0}}$<br>↓<br>$Mds_{0,0}$ | $\mathbf{N_{0,0}}$<br>↓<br>$Nds_{0,0}$ | $PValue_{0,0}$ +=<br>$Mds_{0,0}*Nds_{0,0}$<br>$+ Mds_{0,1}*Nds_{1,0}$ | $\mathbf{M_{0,2}}$<br>↓<br>$Mds_{0,0}$ | $\mathbf{N_{2,0}}$<br>↓<br>$Nds_{0,0}$ | $PValue_{0,0}$ +=<br>$Mds_{0,0}*Nds_{0,0}$<br>$+ Mds_{0,1}*Nds_{1,0}$ |
| $\text{thread}_{0,1}$ | $\mathbf{M_{0,1}}$<br>↓<br>$Mds_{0,1}$ | $\mathbf{N_{0,1}}$<br>↓<br>$Nds_{1,0}$ | $PValue_{0,1}$ +=<br>$Mds_{0,0}*Nds_{0,1}$<br>$+ Mds_{0,1}*Nds_{1,1}$ | $\mathbf{M_{0,3}}$<br>↓<br>$Mds_{0,1}$ | $\mathbf{N_{2,1}}$<br>↓<br>$Nds_{0,1}$ | $PValue_{0,1}$ +=<br>$Mds_{0,0}*Nds_{0,1}$<br>$+ Mds_{0,1}*Nds_{1,1}$ |
| $\text{thread}_{1,0}$ | $\mathbf{M_{1,0}}$<br>↓<br>$Mds_{1,0}$ | $\mathbf{N_{1,0}}$<br>↓<br>$Nds_{1,0}$ | $PValue_{1,0}$ +=<br>$Mds_{1,0}*Nds_{0,0}$<br>$+ Mds_{1,1}*Nds_{1,0}$ | $\mathbf{M_{1,2}}$<br>↓<br>$Mds_{1,0}$ | $\mathbf{N_{3,0}}$<br>↓<br>$Nds_{1,0}$ | $PValue_{1,0}$ +=<br>$Mds_{1,0}*Nds_{0,0}$<br>$+ Mds_{1,1}*Nds_{1,0}$ |
| $\text{thread}_{1,1}$ | $\mathbf{M_{1,1}}$<br>↓<br>$Mds_{1,1}$ | $\mathbf{N_{1,1}}$<br>↓<br>$Nds_{1,1}$ | $PValue_{1,1}$ +=<br>$Mds_{1,0}*Nds_{0,1}$<br>$+ Mds_{1,1}*Nds_{1,1}$ | $\mathbf{M_{1,3}}$<br>↓<br>$Mds_{1,1}$ | $\mathbf{N_{3,1}}$<br>↓<br>$Nds_{1,1}$ | $PValue_{1,1}$ +=<br>$Mds_{1,0}*Nds_{0,1}$<br>$+ Mds_{1,1}*Nds_{1,1}$ |

Figure No. 7: Illustration of Thread Specific Operations for Matrix Multiplication

### Optimization 4: Modifying the CMakeLists.txt

CMakeFile is the file used to set up the compilation instructions for our project. NVIDIA provides several options to compile the program according to the project's requirements. The given CMakeLists.txt was modified to achieve a significant boost in the speed of the given project. Running the project using the given file gave a runtime of about 8.2 seconds on an average with the final code, all optimizations included. The following flags were added -

- `--restrict` - Programmer assertion that all kernel pointer parameters are restrict pointers
- `--use_fast_math` - Make use of fast math library
- `--optimize` - Specify optimization level for host code

Inclusion of these flags in the compilation process led to a speed up of almost 6x as compared to the original file. The final runtime achieved was around 1.4 seconds. The main cause for the increase in performance was the use of the fast math library as it significantly brought down the computation time for the arithmetic computations being performed in our code.
The flags were modified after referring to the documentation provided on the website:
http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

### Optimization 5: Using the serial code along with reduction to matrix multiplication

On running code for smaller batches, using the reduction technique slowed down the process as compared to using the straightforward convolution method in the serial code. Therefore, on analysis of the program with respect to the given datasets, we saw for datasets test2.hdf5 and test10.hdf5, results for the serial code convolution were better while for dataset test100.hdf5 and testfull.hdf5, the reduced matrix multiplication results were significantly faster. Therefore, we used the input batch size as a threshold, that is, batch size greater than 100 use the reduced to matrix multiplication method while under it, the batches use the serial code. This optimization improved performance for all batch sizes.

## Unsuccessful Optimizations:

### Optimization 1: Texture Mapping

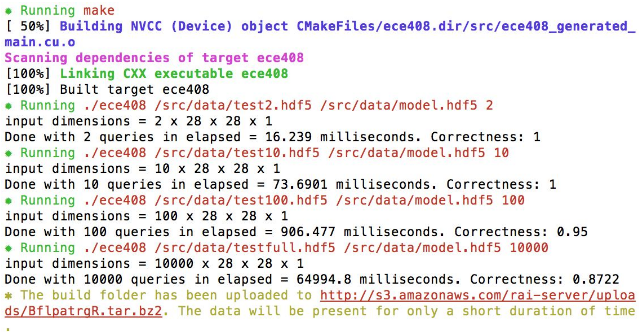
We tried to use texture memory to increase the performance and reduce throughput as texture memory is faster than global memory in most cases. However, for matrix multiplication done for CNN, using texture memory does not result in any improvements on the runtime of the CNN program and therefore, we took out the texture mapping kernel function.

Optimization 2: Parallelization of ReLU

We tried to parallelize the rectifier linear unit, however this resulted in no speed up to the program. We also tried to implement its functionality in the matrix multiplication itself, though that resulted in a higher accuracy. This probably means that implementation weeded out some functionality that was needed in the CNN program. Therefore, we decided not to include the parallel ReLU or the implementation without the reLU function in the program.

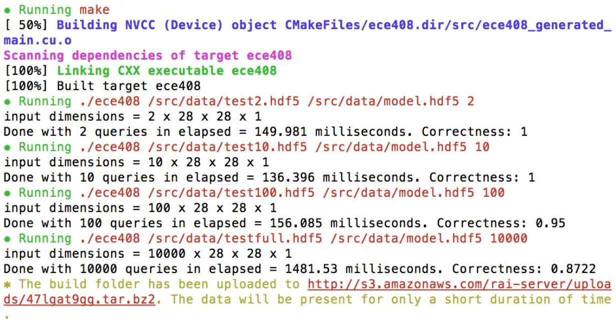
## The Results:

.



```
* Running make
[ 50%] Building NVCC (Device) object CMakeFiles/ece408.dir/src/ece408_generated_
main.cu.o
Scanning dependencies of target ece408
[100%] Linking CXX executable ece408
[100%] Built target ece408
* Running ./ece408 /src/data/test2.hdf5 /src/data/model.hdf5 2
input dimensions = 2 x 28 x 28 x 1
Done with 2 queries in elapsed = 16.239 milliseconds. Correctness: 1
* Running ./ece408 /src/data/test10.hdf5 /src/data/model.hdf5 10
input dimensions = 10 x 28 x 28 x 1
Done with 10 queries in elapsed = 73.6901 milliseconds. Correctness: 1
* Running ./ece408 /src/data/test100.hdf5 /src/data/model.hdf5 100
input dimensions = 100 x 28 x 28 x 1
Done with 100 queries in elapsed = 906.477 milliseconds. Correctness: 0.95
* Running ./ece408 /src/data/testfull.hdf5 /src/data/model.hdf5 10000
input dimensions = 10000 x 28 x 28 x 1
Done with 10000 queries in elapsed = 64994.8 milliseconds. Correctness: 0.8722
* The build folder has been uploaded to http://s3.amazonaws.com/rai-server/uploa
ds/BflpatrgR.tar.bz2. The data will be present for only a short duration of time
.
```

*Figure No. 2*



```
* Running make
[ 50%] Building NVCC (Device) object CMakeFiles/ece408.dir/src/ece408_generated_
main.cu.o
Scanning dependencies of target ece408
[100%] Linking CXX executable ece408
[100%] Built target ece408
* Running ./ece408 /src/data/test2.hdf5 /src/data/model.hdf5 2
input dimensions = 2 x 28 x 28 x 1
Done with 2 queries in elapsed = 149.981 milliseconds. Correctness: 1
* Running ./ece408 /src/data/test10.hdf5 /src/data/model.hdf5 10
input dimensions = 10 x 28 x 28 x 1
Done with 10 queries in elapsed = 136.396 milliseconds. Correctness: 1
* Running ./ece408 /src/data/test100.hdf5 /src/data/model.hdf5 100
input dimensions = 100 x 28 x 28 x 1
Done with 100 queries in elapsed = 156.085 milliseconds. Correctness: 0.95
* Running ./ece408 /src/data/testfull.hdf5 /src/data/model.hdf5 10000
input dimensions = 10000 x 28 x 28 x 1
Done with 10000 queries in elapsed = 1481.53 milliseconds. Correctness: 0.8722
* The build folder has been uploaded to http://s3.amazonaws.com/rai-server/uploa
ds/47lgat9gg.tar.bz2. The data will be present for only a short duration of time
.
```

*Figure No. 3*

The aim of parallelization of our code was to increase the efficiency of the program by decreasing the run time as much as possible. If we compare Figure No. 2 and Figure No. 3, we can begin to understand the impact of parallelization of the code on runtime. Below is a table constructed to compare the runtime side-by-side:

| Input Size (dimensions) | Serial Code Runtime Speed | Final Parallelized Code Runtime Speed |
|---|---|---|
| **2** x 28 x 28 x 1 | 16.239 milliseconds | 149.981 milliseconds |
| **10** x 28 x 28 x 1 | 73.6901 milliseconds | 136.396 milliseconds |
| **100** x 28 x 28 x 1 | 906.477 milliseconds | 156.085 milliseconds |
| **10000** x 28 x 28 x 1 | 64994.8 milliseconds | 1481.53 milliseconds |

As we can see in the above table, green implies shorter runtime while red implies longer runtime. Serial code is faster than parallel code only for small inputs (here 2 and 10 batches input data size). As soon as the input data size crosses a certain threshold, parallel code is significantly faster than serial code (here seen with input data size of 100 and 1000 batches). As a result, there is a remarkable difference in runtime when using parallel algorithms to achieve the same goal.

All images from course book or course slides - © David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016

## Further Improvements:

Our initial aim of improving performance by using parallelization of the given code was achieved. Even though, there can be several more optimizations that can improve the runtime even further(using one thread for several computations, better memory coalescing and so on), our implementation speeds up the serial code by about 43x for a batch of 10000 input matrices.

Another optimization we think could be helpful is using texture memory instead of global memory wherever possible. We only tried to use it with matrix multiplication, however given the fact that it is faster than global memory, we believe that it will result in significant improvement in performance.

## Team & Contribution:

All of the 3 team members helped with both report writing as well as writing the code for parallelized operations.

Arnav Agarwal: Junior, Computer Engineering
Saksham Saini: Senior, Computer Engineering
Karan Usgaonkar: Senior, Computer Engineering

All images from course book or course slides - © David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016