

ECE 408 Final Project
CNN Algorithm Optimization
Fall 2016

Justin Yang - jyang101
William Yu - wyu13
Evan Dotterer - dottere2

Final Runtime: ~1.5 seconds (average output shown below)


```
* 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 = 1575.73 milliseconds. Correctness: 0.8722
```

Our plan at the beginning was to simply parallelize every function in the sequential code. Later on, compiler optimizations, memory coalescing, and small changes here and there helped us bring our performance to the level it is at now.


NVProf, NVVP results (for each kernel)

Matrix Multiply Kernel

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	153600	487.474 GB/s	
Shared Stores	12800	40.623 GB/s	
Global Loads	25600	20.311 GB/s	
Global Stores	256	203.114 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	192256	548.611 GB/s	

L2 Cache

L1 Reads	51200	20.311 GB/s	
L1 Writes	512	203.114 MB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	51712	20.515 GB/s	

The shared memory transfer rate for this kernel is much higher than in other kernels. This kernel does significantly more calculations than any of the others so it makes sense that it has one of the larger rates.

matrixMultiply(float*, float*, float*, int, int, int, int, int, int)

Duration	126.621 μ s
Grid Size	[4,4,1]
Block Size	[16,16,1]
Registers/Thread	25
Shared Memory/Block	2 KiB
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

The matrix multiplication used a tile size of 16, we experimented with sizes 8 and 32 but the accuracy and runtime suffered as a result.


Place Into Y Kernel

placeIntoY(float*, float*, int, int, int)


Duration	5.696 μ s
Grid Size	[4,1,1]
Block Size	[1024,1,1]
Registers/Thread	10
Shared Memory/Block	0 B
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

Seeing how this was a pretty small kernel, we were happy with the fact that its duration was so short.

L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	128	1.798 GB/s	
Global Stores	4096	14.385 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	4224	16.183 GB/s	

L2 Cache

L1 Reads	512	1.798 GB/s	
L1 Writes	4096	14.385 GB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	4608	16.183 GB/s	

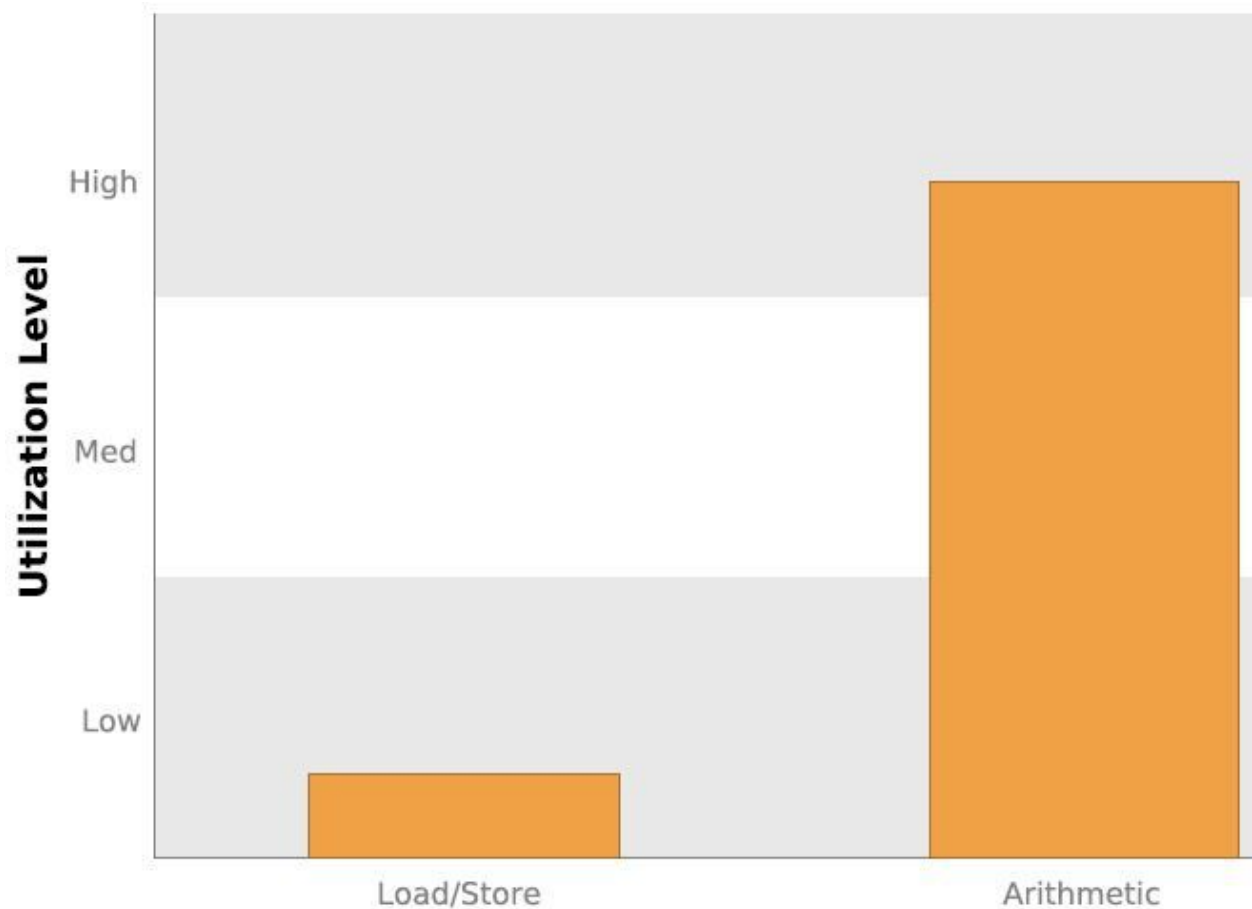
We didn't end up using any sort of shared memory for this kernel due to its size, we didn't think that it would have much of an impact on the performance.

Subsample Kernel

subsample(float*, int, int, int, int, float*, int, int, int, int, int)

Duration	182.907 μ s
Grid Size	[450,1,1]
Block Size	[1024,1,1]
Registers/Thread	20
Shared Memory/Block	0 B
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

For this kernel, the long run time is due to the large amount of work that it is doing, which can be seen in the following graph.

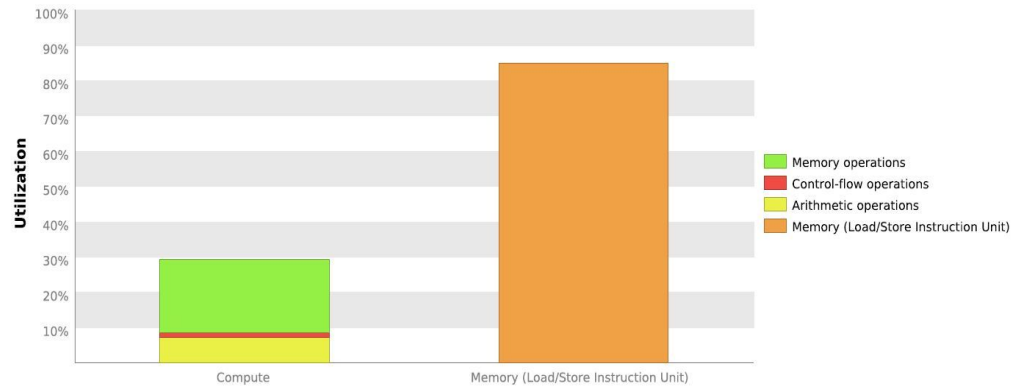


Unroll Input Kernel

unroll_InputOptimized(int, int, int, int, int, float*, float*, int)

Duration	74.494 μ s
Grid Size	[2,1,1]
Block Size	[1024,1,1]
Registers/Thread	18
Shared Memory/Block	0 B
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

The graph below that shows the utilization of this kernel follows our methodology trying to optimize since it is primarily memory transactions, rather than control flow or arithmetic operations that are taking up most of the space in the kernel.

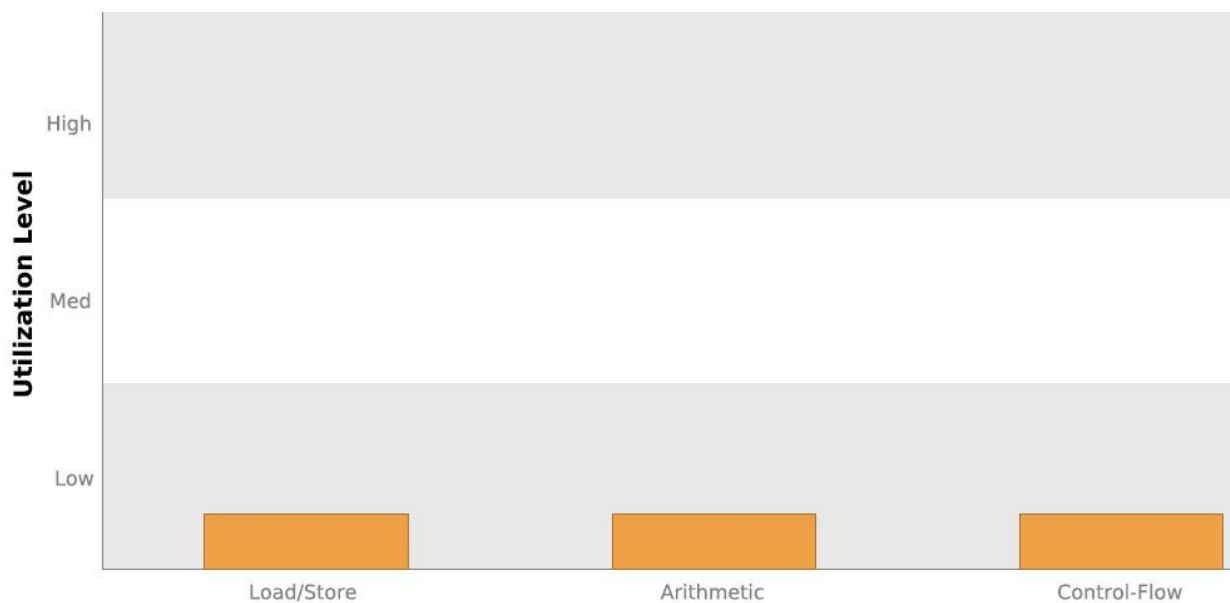


Unroll Weights Kernel

unroll_W(int, int, int, float*, float*)

Duration	30.335 μ s
Grid Size	[1,1,1]
Block Size	[1024,1,1]
Registers/Thread	14
Shared Memory/Block	0 B
Shared Memory Requested	112 KiB
Shared Memory Executed	112 KiB
Shared Memory Bank Size	4 B

Since the weights is generally going to be of a smaller size than the input, it makes sense that it's going to be faster than unroll input since they share the same algorithm. We managed to get a good balance between all the arithmetic, control, and load/store operations for this kernel as can be seen by the graph below



Compiler Optimizations

We checked to see if we were in debug mode before setting the debug flags using

```
message("CMAKE_BUILD_TYPE = ${CMAKE_BUILD_TYPE}")  
if(CMAKE_BUILD_TYPE MATCHES DEBUG)  
Set(CUDA_NVCC_FLAGS_DEBUG ${CUDA_NVCC_FLAGS_DEBUG} "-G")  
endif(CMAKE_BUILD_TYPE MATCHES DEBUG)
```

This cut a significant portion of our runtime off. Without the check, our code runs at around 7800 ms but with it removed, our code runs at 1700 to 1900 ms, a significant speedup.

The compiler optimization -O was used since it allows the compiler to optimize the host code as it sees fit. This resulted in a 200 to 300 ms speedup. We chose level 3 since it had the greatest impact on our runtime.

```
list(APPEND CUDA_NVCC_FLAGS -O3)  
list(APPEND CUDA_NVCC_FLAGS -use_fast_math)
```

The compiler optimization -use_fast_math was used in order to shorten some of the arithmetic operations that were being done in the code. It approximates single-precision floating point operations and flushes denormal values to 0. This had a minor runtime reduction of 100-200 ms.

OVERALL OPTIMIZATIONS

Given the sequential code for CNN, we were able to implement numerous optimizations to parallelize the algorithm resulting in faster runtime. We added optimizations to all layers of the CNN which included convolution, subsampling, and the forward operation along with spreading the work of the relu function to the former layers. Once we added the parallel features, we further optimized the runtime of those functions to maximize the capability of the GPU.

Convolution Forward Layer

We implemented the convolution forward layer by transforming the process into a matrix multiplication. Four main device code operations, unroll_InputOptimized, unroll_W, matrixMultiply, and placeIntoY, were used to modularize the convolution layer. A final host code function, convLayer_foward, was used to call these device functions on each

batch of input features. These functions with their implementations to fully optimize the GPU are shown below.

unroll_InputOptimized (Device)

We unrolled the input features into a matrix that later serves as one of the operands of the later matrix multiplication. In our implementation, we took in one batch of input features at a time and unrolled those features into an array, X_unroll. Each thread unrolls $K \times K$ elements and given the way the data was organized, we tried to maximize the contiguous access of those threads to utilize the DRAM bursts. The data was organized such that the channel dimension was placed in contiguous memory; thus, we made sure each sequential thread accesses a different channel.

unroll_W (Device)

We unrolled the weights into a matrix that serves as the other operand in the matrix multiplication. We took in all weights and each thread unrolls $K \times K$ elements into the matrix. The data was organized such that the output feature index dimension was placed in contiguous memory; thus, each sequential thread would access a different output feature to optimize memory coalescing.

matrixMultiply (Device)

This function does a simple matrix multiplication of two input arrays of arbitrary size with their widths and height specified as parameters. We used tiled shared memory in implementing our matrix with each tile being 16×16 .

placeIntoY (Device)

This function takes the product of the matrix multiplication and places the solution, Y_unroll, into memory in the correct data orientation where each output feature is in contiguous memory.

Trial 1 - In order to achieve memory coalescing, we had each thread do one column of the Y_unroll matrix. Thus, we used $K \times K$ threads each do one column. Each thread would put the column into the correct place into the deviceY memory.

Trial 2 - Due to the large amount of threads that a kernel allows, we wanted to optimize the fact that each thread does an entire column. If the column is large, each thread would still do large amounts of work. Therefore, we increased the number of threads so that each thread each does one element. Each thread would still access contiguous elements in the array. This resulted in a 250 ms decrease in runtime.

convLayer_forward (Host)

Trial 1 - When implementing this function, we first used a for loop that goes through N, the number of batches, iterations in which each iteration passes a batch through the above functions. In getting each of the batches, we allocated three device memory instances that is the size of one input feature batch and two output feature batches respectively that acts as both the unrolled input feature matrix, the result of the matrix multiplication, and the final output feature in the correct format. In each iteration, we mem copied one input feature batch from the host data given into deviceX, the array that contains the unrolled data. We pass this data through the unroll_InputOptimized that unrolls the matrix. We also unrolled the weights in each iteration as well. Once the matrix multiplication is complete, we unroll the output into the correct dimension format and mem copied the output into the correct location of the final output array.

Trial 2 - After implementing the above function, we realized we could further optimize it by utilizing streams. Each batch being done in each iteration requires kernel calculations and memory being copied back and too from host. Having each iteration use the default stream, we would not be able to mem copy while calling the kernel functions. Thus, we used two streams that in each iteration resulting in incrementing the iterator by two every time. This way while one stream is copying the memory from device to host, the other stream can already begin the convolution kernel calculation of its respective batch. Vice versa in the other direction once the calculation is completed. This resulted in a faster runtime.

Trial 3 - In our third trial, we realized that the memory copies from device to host or host to memory caused a large buildup in runtime. Just initializing a memcpy requires a large amount of time. Thus doing this thousands of time can definitely have a negative effect on the runtime. Due to the iterations depending on the number of batches (in this case 10,000) we would do 20,000 mem copies total. Therefore, in this trial we would minimize the number of memory copies. We realized that in the previous trials we copied each batch to and from host. The solution we found was to keep the entire memory in the GPU. There is no point in copying it back to host. Thus, we still do each batch at a time, but instead of copying to the device and then to host one by one, we allocate the entire X data into the device. Each function would then take in the entire data X along with an index. This index would indicate where, in the X array, to start the calculations. Streams are still used with their respective index. Also, we realized that each input output pair in each batch uses the same weights. Therefore, we only need to call unroll_W once. This resulted into a further decrease in runtime of 1.3 seconds.

Trial 4 - Adding two more streams per iteration caused the runtime to decrease by 200 ms. However, any more streams than that, accuracy falls.

Subsampling layer

This layer just averages each element with its surroundings. Since each output element is independent in calculations to the other output elements, we parallelized this process.

Subsample (Device)

Trial 1 – Originally, our subsample device function takes in one batch of input features and have each thread calculate only one output element by averaging two input elements. The data elements are organized such that each input feature is in contiguous memory. Thus, sequential threads access different input features to maximize memory coalescing.

Trial 2 – This trial goes along with the change and optimization in the host function in trial 2 of subsampling_layer, shown below. By taking in one batch of input features at a time and having each thread calculate one output element, we realize this code is limited to the iteration of the host code. The threads only begin by calculating the next batch only when we are in the next iteration in the host code. Instead, we changed this subsample device function to take in all batches of the input features. I increased the number of threads, and have all the threads equal to the amount of output elements in each feature in each batch. Thus the total amount of threads being used is equal to the number of output elements * number of output features * number of batches. As with the other trial, each thread would then calculate an output element, but with the larger number of threads, we would be able to calculate the batches simultaneously rather than iteratively.

Subsampling_layer (Host)

Trial 1 – Similar to the first trial of the convolution layer, we have a for loop that calculates the subsampling of a batch in each iteration. We would Memcpy one batch at a time and store it in device memory from the host, do the calculations on that single batch, and mem copy that batch output back to the host memory. Therefore, again we are moving data back and forth from host for each batch. This trial uses the Trial 1 code for subsample device code.

Trial 2 – In this trial, we optimized streams. We used two streams in each iteration to fully synchronize the kernel functions along the memory copying. We then increased the iterator by two each time. However, this did **not** have much of a speed increase.

Trial 3 – After realizing that moving data back and to from host for **each** iteration was completely inefficient, we decided to keep the entire data in the device. Therefore, using the device code in trial 2, the subsampling layer would have all threads parallelize the calculations for the batches, not just for the output features. Because of this, the iterative loop and consequently, the uses of the streaming, were not used anymore. Only two mem copies are now used instead of 20,000 mem copies greatly increases the speed.

Optimization of Relu function

Trial 1 - We realized the relu calls in the host code was just sequentially going through each element in the array and clamping the values if they go under zero. Thus, instead of re-scanning through the data sequentially, we decided to parallelize this as each value is independent of the other. Each thread would check one value to see if it is under zero. If it is, it would clamp it to zero. A further optimization to improve this strategy is to improve the code divergence that this would cause.

Trial 2 - We removed the relu functions and function calls, and added if statements in Y_unroll and matrix multiplication to store negative values as 0. Instead of creating a separate device function that requires mem copies and threads to be initialized, we decided to just check this clamping simultaneously while doing the Y_unroll and matrix multiplication. There is a trade off with this, as adding if statements in the Y_unroll and matrix multiplication would definitely cause thread divergence. Although due to the shorter runtime of implementing this vs having the original sequential relu code or with the trial 1, the trade off of synthesizing the relu in other functions outweigh the costs of thread divergence.

Fully Forwarding Layer

Originally a sequential function, we decided to parallelize this. Luckily, it was a simple matrix multiplication. Similar to the matrix multiplication code we used for convolution,

we used tiled shared memory of size 16 by 16 to optimize quick memory accesses from shared as well as memory coalescing when accessing the memory.

We initially had a teammate's shared matrix multiplication code from previous MPs, but eventually realized that its code was not optimal. In our old code, `__syncthreads` was incorrectly placed in the 16-iteration for loop when calculating the output value for that tile. This would cause all threads to hold when incrementing the output value even though each output value is calculated independently. The `__syncthreads` was then moved outside of this loop resulting in a runtime decrease of ~200 ms.

To further optimize the shared memory matrix multiplication, we tried using many different tile widths ranging from absurdly small to absurdly large, but in the end, a tile size of 16 (default) had the best runtime and accuracy.

Further Optimizations to be Done

If we continued to work on this project, there are many optimizations that still could have been implemented. We realized that the `zeros` function given in `utils.hpp` (that allocates the host memory based on the dimensions) accumulates to a total of 300 ms of runtime when nothing but those allocations are done. We used `malloc` instead which resulted in a faster time when those functions are isolated, but when testing with all the functions, the runtime did not change.

Next, we simultaneously calculated each batch in the subsampling layer to have a decrease in runtime. However, for the convolutional layer, although we are not mem copying in each iteration, we are still calculating each batch at a time iteratively in the for loop. Thus each batch calculation is still dependent on the sequential iterations of the host. A further optimization would be to calculate those batches simultaneously, which in our opinion, would have a huge influence in a further decrease in runtime. Lastly we had problems with choosing the correct constants for the 5 x 5 kernel weight in the Winograd algorithm. Perhaps this algorithm may have been faster than the matrix multiplication technique as our kernel weight sizes are small.

Member contributions

Justin Yang - convolution code, subsampling, report

William Yu - convolution optimization, matrix multiplication, forwarding, report

Evan Dotterer - parallelization, profiling, report

Special thanks to Professor Hwu and the 408 TAs!