ECE 408 Final Project: Convolutional Neural Network

Saurabh Mishra, Ziyun He, Naphat Lertratanakul

Table of Contents

Problem Specification

Machine learning has been used in many applications to train or adapt a situation from the experience received in data sets. To be more effective in producing correct results, training and decision making has to be computed using massive amounts of data. This type of practice has recently been more practical due to the accessibility of inexpensive, massively parallel GPUs that are great for performing repetitive tasks in parallel on a wider scale of data entries.

With the convenience of GPUs, we are able also able to classify new data based on the training sets. Since we have acquired the training data, we need a fast and efficient way to classify new, large input data sets. The input is many sets of 28x28 images, each expected to represent an image of a digit from 0-9, inclusive. We want to classify these images into one of the 10 different digit categories.

Since the image may have complex features (in this case, numbers may have different fonts and handwritings), we rely on the deep learning area of machine learning to automatically detect these features from the raw data through multiple levels of feature representation. Because of this, deep learning usually requires much more data in order for the system to automatically detect a relevant number of patterns. As a result, we decided to use the Convolutional Neural Network (CNN), one of the most popular procedures in deep learning, because it is easier to train and can be generalized much better than other procedures.

Before designing the parallel algorithms that fit under the CNN procedure, we need to know the limitations in the hardware's parallel computing capabilities. We will be testing our code on the rai server, which contains one 1 device supporting CUDA and the following specifications:

Table 1: Test Device Specifications

Device Name	Tesla K80
Computational Capabilities	3.7
Maximums:	
Threads per Block	1024
Threads per Multiprocessor	2048
Shared Memory per Block	48 KiB
Shared Memory per Multiprocessor	112 KiB
Block Dimensions	[1024, 1024, 64]
Grid Dimensions	[2147483647, 65535, 65535]

Warps per Multiprocessor	64
Blocks per Multiprocessor	16
Memory:	
Global Memory Size	11.172 GiB
Constant Memory Size	64 KiB
L2 Cache Size	1.5 MiB
Memcpy Engines	2
Multiprocessor:	
Multiprocessors	13
Clock Rate	823.5 MHz
Threads per Warp	32

From the device specifications, we are able to perform our process with multiple threads, but we should keep in mind that the most efficient number of threads is still limited by the number of cores in the machine.

Approach

After we have identified our problem and looked at the hardware limitations, we started developing an elementary parallel algorithm for each function each serial implementation in the forward operation. Initially, we were given the barebones serial layout. The initial forward operation layout had 10 layers total, with 5 different layers: convolution, rectified linear, average pooling, fully forward (connected), and argument maximum. For a batch size of 10,000, the serial program took 20 minutes and 19 seconds to classify, with the following breakdown on the times of each function:

Table 2: Running Time of Serial Functions

Function	**Running Time (seconds)**
Convolution Forward Valid (1)	382.006
Rectified Linear Unit 4D (1)	2.99
Average Pool (1)	10.690

Convolution Forward Valid (2)	798.785
Rectified Linear Unit 4D (2)	0.833
Average Pool (2)	2.478
Fully Forward (1)	20.115
Rectified Linear Unit 2D	0.018
Fully Forward (2)	0.171
Arg Max	0.002

From the above running times of each function, we observed that parallelizing the forward convolution function would reduce a huge portion of the runtime. We started with a basic parallelization of the convolution and average pooling. That reduced the overall running time of the program from 20 minutes and 19 seconds to 1 minute and 26 seconds. After successfully parallelizing the convolution and average pooling, we moved on to parallelizing the matrix multiplication (fully forward), rectified linear and argument maximum functions. This reduced the running time to 1 minute and 4 seconds for a batch of 10,000. The following table shows the timings of the functions we parallelized first.

Table 3: Basic Parallelization Running Time

Kernel Function	**Running Time**
Convolution Forward Valid (1)	11.523 s
Rectified Linear Unit 4D (1)	48.694 ms
Average Pool (1)	437.946 ms
Convolution Forward Valid (2)	78.431 s
Rectified Linear Unit 4D (2)	44.006 ms
Average Pool (2)	370.836 ms
Fully Forward (1)	450.335 ms
Rectified Linear Unit 2D	193.02 μs
Fully Forward (2)	7.889 ms
Arg Max	339.771 μs

After we converted the individual serial implementation functions to parallel, we started implementing optimal parallel algorithms, hardware optimizations and compiler optimizations.

Optimizations and Outcomes

When implementing the most basic parallelled implementation of the convolution and average pool functions, we took timings around each cudaMalloc(...), which showed on average 150ms for each cudaMalloc. As a result, we tried to call cudaMalloc as few times as possible. Instead, we reused allocated memory and copied the necessary inputs into the allocated memory to use in the specific kernel functions. As mentioned in the previous section, this implementation with the basic parallelization reduced the time to at best 1 minute and 4 seconds. The simpler optimizations we implemented afterwards (tiled matrix multiplication and accumulators) showed the same or worse performances. From analyzing the timeline profiling of the program on the NVIDIA Visual Profiler (NVVP), we observed that cudaMemcpy(...) had a lot of overhead - more than cudaMalloc - and that hindered our speedup progress. What was worse, each kernel function launch is delayed by the cudaMemcpy. Also, we later discovered on NVVP that there were only 2 Memcpy engines and each kernel function required on average 4 memcpy's with this implementation.

By refactoring the code to represent the past assignments (the straightforward order of cudaMalloc, cudaMemcpy [from host to device], kernel call, cudaMemcpy [from device to host], and cudaFree), we gained a forward operation running time of 30.6 seconds. To further reduce the number of calls to memcpy, the parameters from the original function were modified for each input array dimension. Since the dimension arrays are of fixed size, regardless of the input set size, passing in each individual dimension value did not require memcpy or malloc. This reduced the time by 5-6 seconds.

The basic optimizations that were effective after the code refactor are tiled matrix multiplication and accumulators. The accumulator was added to both the convolution and average pool functions to reduce the number of writes to global memory. The writes to global memory in the convolution was reduced from (filter_height)*(filter_width)*(num_channels) to 1. The number of global writes in the average pool function was reduced from $(\text{pool\_size})^2$ to 1. The tiled matrix multiplication was implemented as an optimization for the fully forward function, because storing the inputs in tiled blocks in shared memory reduced the number of global reads.

Originally, the basic parallel implementation of matrix multiplication required (y_height)*(y_width)*(x_width) + (y_height)*(y_width)*(w_height) global reads, where y is the output matrix, x and w are the input matrices, and x_width = w_height. After the tiled matrix multiplication implementation, the number of global reads is ceil(((y_height)*(y_width)*(x_width) + (y_height)*(y_width)*(w_height))/16.0), which is 1/16 of the original number of global reads.

Reducing the number of global reads and writes should be a considered optimization in parallel algorithms, because fetching and writing data from and to global memory uses more clock cycles and therefore has a higher latency. When fetching from shared memory, the latency

is much lower. In this case, the share memory tiles were fairly small (16x16), so the size limit of shared memory was not a concern. Adding in these two optimizations reduce the program runtime by a few hundreds of milliseconds.

The next optimization we considered was with the CUDA compilation flags in the CMakeFile. Since our device has a K80 card with 3.7 computation capabilities, we can set ARCH_FLAGS to 3.7 because we are only testing on that architecture. Specifying the compute capability may be necessary depending on the NVIDIA GPU. Otherwise, the ARCH_FLAGS may also be set to "Auto," but since we are testing on this one device, we were able to specify this flag. Another flag that was set for the CUDA CC compiler is "use_fast_math." This compiling tag uses a special unit in the multiprocessor to perform fast math functions in a single instruction instead of many instructions in the normal implementation. Since we did not use any special math functions, such as tanh or sigmoid, as the non-linear function bias value, this tag did not show any noticeable speedup in the program. The flag that played a significant impact to the runtime was the debugging option. This was the most important flag to the speedup, since when a program is compiled to be viewed in debug mode, the arrangement of some of the code cannot be optimized otherwise certain lines and values stored cannot be examined in the debugger. We can change the debugging options, but once we have functional code, turning off this flag is fine. Adding in these changes to the CMakeFile reduced the running time from ~25 seconds to ~8 seconds.

Since the the dimensions of the input features maps and filter weights are always consistent throughout the process and they are needed for almost every function, we tried to utilize the concept of constant memory and allocate the dimensions into it. However, the result was not significant. We didn't see a dramatic increase in efficiency. We think this is due to the amount of data being reused has not reached a scale of significance. We still decided to leave the implementation in due to the amount of usage across all functions.

Another optimization that we implemented was tiled convolution, because this reduces the number of global reads by storing inputs into shared memory, similar to the tiled matrix multiplication. By doing that, data will be reused TILE_WIDTH times. Along with implementing the tiled convolution, we combined the rectified linear function, since we are checking the output values for whether they are less than zero. Combining this check in convolution means that we do not have to set up and call another kernel function, and this takes away the cudaLaunch overhead of a kernel function. The tiled convolution reduced the program runtime from ~8 seconds to ~2.5 seconds. The combination of the rectified linear check into the tiled convolution function had a speedup of ~200 milliseconds.

When designing the blocks and grids for our tiled convolution, we realized that each convolution should be designed differently and be given individual kernel function to obtain the optimum runtime overall. Therefore, after the initial work with the regular tiled convolution that works universally, we customized the block size and dimensions according to the dimensions of each input. For example, for the second convolution, we used a tile size of 16, which enables each iteration to generate two outputs instead of one to account for the output dimension of 8.

We did see a slight improvement with the run time. However, the improvement was not as significant as the matrix multiplication method we tried later.

At the end of forward_operation function, we noticed two consecutive events. The first was the execution of the argmax kernel, and the second was a memcpy from device to host. Originally, these two functions occurred serially. This is because we require the output from the kernel function to copy over to the host variable. This gave us some room for optimization. Using asynchronous streams, we could divide the work happening in the argmax kernel into two separate function calls. The first call would operate on half of the data, and the second call would operate on the other half. After the first call finishes, we can begin transferring over the first half of the data from device memory to host memory. As this transfer is occurring, we can perform the GPU computations in the argmax kernel function over the second half of the data. This allows us to run a little more of the code in parallel and reduce our runtime by a few hundreds of milliseconds.

Table 4: Optimized Parallel Running Time

Kernel Function	**Running time**
Convolution Forward Valid (1)	374.624 ms
Average Pool (1)	202.355 ms
Convolution Forward Valid (2)	2.002 s
Average Pool (2)	36.9 ms
Fully Forward (1)	13.815 ms
Rectified Linear Unit	89.589 μs
Fully Forward (2)	221.82 μs
Arg Max (stream 1)	17.887 μs
Arg Max (stream 2)	18.304 μs

After reaching a bottleneck in our runtime, we decided to put our emphasis on the methods that are taking the most of our runtime: the two convolutions. We knew that this had to be further optimized, because the profiling analysis of the optimized code indicated that 89.7% of the computations being performed were from the second convolution layer. In addition to the tiled convolutions, we decided to try to reduce the convolution layer to matrix multiplication. By unrolling the convolution input and filter arrays into respective matrices, a matrix multiplication can be performed instead of a convolution. We chose to do the unrolling method because, GPUs have been optimized to perform matrix multiplications really well. By following the book and embedding it into the main framework, we decided to have four major components in this

algorithm. First, we unrolled the input weights into the corresponding dimensions we need, namely, from conv1[K,K,C,M] to conv1_unrolled[M,C,K,K]. Then, we unrolled input x one sample at a time, since we are afraid of the memory footprint for keeping all sample input feature maps for a mini-batch being prohibitively large. After forming the two unrolled matrices, we computed the result by utilizing the tiled matrix multiplication function that we originally designed for fully forward function. To help embed the function into the framework and reuse it for the second convolution, we transposed the matrix multiplication result back into the right dimensions. To much of a surprise, our implementation added 200ms to the runtime compared to our previous implementation with tiled convolution. Matrix multiplication convolution introduced more overhead to the entire process since we had to unroll the inputs into the matrices we wanted before performing matrix multiplication and rerolling the result back. In addition, since we could only perform the process one sample at a time due to the memory restriction, the time taken to loop through all of the samples proved that this method should not be considered compared to the tiled convolution.

Although our previous method with unrolling failed due to the large overhead from creating the unrolled matrices, we didn't give up on attempting the reduction of the convolution layer to matrix multiplication. After all, we have learned from theory that by using tiled matrix multiplication, each replicated input feature map element and convolution filter weight element is reused TILE_WIDTH times. The textbook also encourages us to do matrix multiplication when the TILE_WIDTH is larger than K^2*$TILE\_WIDTH^2/(TILE\_WIDTH-K+1)^2$. Therefore, we tried to reimplement unrolling by accessing the corresponding inputs directly from the input pointer according to their indices instead of obtaining the specific matrices. When we tested it out, we found some inconsistencies on the results. While it improved our overall efficiency compared to our original tiled convolution, it slowed down the performance for the first convolution. Through some trial and error, we decided to use a hybrid model of convolution: tiled convolution for the first convolution within the model and matrix multiplication for the second convolution within the model. This method has reduced our runtime by about 600ms.

The final optimization that impacted our speedup was coalesced memory accesses. Regarding coalesced memory access, we noticed that the average_pooling kernel function wasn't taking advantage of the DRAM bursts. We printed out the memory addresses that each warp was accessing, and saw that each thread (in a warp) was accessing an address 128 bytes away from the next thread. To rectify this, we modified the dimensions of the grid and block so that each thread in a warp would access consecutive indices in the global input array. This helped us achieve a speedup of approximately 200ms. When we checked NVVP on the Global Load & Store Efficiency data, which indicates whether multiple threads are accessing a full warp of memory during one fetch, the value for both increased from 12.5% before coalesced memory access was implemented to 100% after it was implemented.

Final Results

In our final implementation, we have two convolution kernels (one that uses shared memory and one that is performed using a reduction), one average pool kernel, one fully forward kernel, one relu kernel and two arg max kernels. The first convolution layer is done with the tiled

convolution method, since the overall computation time was better with the first set of input compared to the unrolling method. This method makes use of shared memory to reduce the number of global reads. The second convolution layer is performed using the reduction method of unrolling the input data set and convolution filters into matrices, performing matrix multiplications, and rerolling the product into the original form. This was very effective for the larger convolution since there was more data being processed in the second convolution. The average pooling kernel is implemented with coalesced memory load and store, and it reduces the amount of writing to global memory by using an accumulator. The fully forward kernel performs a tiled matrix multiplication to reduce the number of global reads. There is only one relu kernel function that was needed and that is only called once because the comparison made in this function can be made within the convolution kernels. There are now two arg max kernels because cudaStreams was implemented to copy back the output faster. We now half of the output data back at a time.

The final version of our project gave us an approximate runtime of 1.13 seconds. The following table shows the speedup of the overall program and each individual function (obtained from NVVP) compared to the serial version.

Table 5: Runtime Comparison Between Serial and Optimized Parallel Implementations

Function	**Serial Running Time**	**Optimized Parallel Running Time**
Convolution Forward Valid (1)	382.006 s	-
Tiled Convolution Forward Valid	-	383.423 ms
Rectified Linear Unit 4D (1)	2.99 s	-
Average Pool (1)	10.690 s	55.349 ms
Convolution Forward Valid (2)	798.785 s	-
Unrolled Convolution Reduction	-	1.057 s
Rectified Linear Unit 4D (2)	832.665 ms	-
Average Pool (2)	2.478 s	12.741 ms
Fully Forward (1)	20.115 s	13.877 ms
Rectified Linear Unit 2D	18.282 ms	-
Rectified Linear	-	89.49 μs
Fully Forward (2)	170.981 ms	222.202 μs
Arg Max	2.252 ms	-

Arg Max 1 + 2	-	17.92 μs + 18.559 μs
Total	**20 minutes 19 seconds**	**1.13 seconds**

"-" in certain cells in the table indicates that this function is not in the specified implementation.

The runtime of the Convolutional Neural Network (CNN) is much better than before and our parallelized and optimized implementation resulted in a significant speedup.

Future Work

Due to the time constraints of the project and the scope of the course, there were a few optimizations suggested on NVVP that we were unable to implement for our CNN. A few of the suggested optimizations were coalescing the memory accesses/stores, using shared memory more efficiently, execute functions in warps and perform more kernel concurrency.

With coalesced memory accessing and storing, the unrolled convolution kernel currently has a Global Load Efficiency of only 18%. What we planned on doing was rearranging the loop ordering of the kernel function to access neighboring memory. As for the fully forward matrix multiplication, the Global Store Efficiency is 62.5%. What could have been improved in this implementation is custom block sizes depending on the kernel. The first fully forward kernel actually has 100% Global Load and Store Efficiency, but due to the different sizes of inputs and outputs, that may not be the case with all matrix multiplications done with the fully forward function.

When performing unrolling for the reduced convolution, using shared memory would have been more efficient since, on average, an input array entry is read more than once, which uses more global reads than necessary.

Since arg max finds the maximum of each row of 10 entries, each thread is responsible for a row, which results in low warp execution efficiency (49.9%). What could have been done to improve this was transposing the input array so that each row was a column and each thread accessed sequential columns to a full warp.

As for the low kernel concurrency result, if we had better knowledge on the use of cudaStreams, then we could stream the correctly ordered output from one kernel to be the input of another and increase the kernel concurrency (percentage of two kernels being executed in parallel). Even with the better knowledge of streaming, the kernel concurrency rate is still limited due to the input dependency of each kernel function.

If we were to redo this project, researching more on the device specifications would have also helped to determine the threading and block limitations, as well as the memory limitations. This would have saved us much debugging time when experimenting with the more challenging optimizations.

Contributions

Saurabh Mishra:

My first set of tasks focused on code for the basic parallel implementation of conv_fully_valid kernel, relu4 kernel, average_pool kernel, and fully forward kernel. Next, I worked on adding asynchronous streams and events into the forward_operation function. This focused mostly on streaming the argmax kernels to speedup the device to host memcpy function at the end of forward_operation. Finally, I worked on the unrolling optimization for convolution and adding memory coalesced access to average_pool.

Ziyun He:

I first parallelized the argmax function within our first iteration of optimization. Then I focused on the optimization on convolution with shared memory and unrolled matrix multiplication (unrolled and direct accessing methods).

Naphat Lertratanakul:

One of my first tasks involved the basic and tiled parallel matrix multiplication implementation for the fully forward kernel. After that, I debugged and fixed the parallel implementation of the argmax kernel. Our original parallel implementation had only three cudaMalloc's and many cudaMemcpy's and that was creating a bottleneck effect on our performance, so I refactored the code to a more optimal speed. Throughout the entire process, I collected the profiling NVPROF information from the builds and analyzed them under NVVP. I also included the build and test instructions in the USAGE file.