Matrix Multiplication

Jonathan Lehman

February 22, 2012

To begin with, I kept some of the same principles from the matrix addition homework. I reused the method to fill Matrix A and Matrix B with random float values, where both are seeded with a different number to ensure that Matrix A is not identical to Matrix B. I also reused my method to check the arguments in general, to ensure that there is the correct number of arguments and that all the arguments are valid integers greater than or equal to 1 and less than the maximum integer value. Lastly, I reused my method to check the capabilities of the GPU to ensure that the block sizes, grid size, number of threads, and memory being used are valid. From the last homework, I learned that the maximum memory cannot be used on the GPU, purely in the array sizes (A, B, and C), because the GPU also needs local memory etc. and will cause an error that causes the GPU to silently die, which was the reason for my unbelievably low timings. Although, the memory check does not ensure that an error will not occur, it will act as a secondary check to ensure that the user does not enter matrix sizes that the GPU definitely will not be able to handle.

In order to account for the GPU errors that I was unaware of last time, I included a print statement with cudaGetErrorString(cudaGetLastError()), to determine if there were any errors on the GPU. Though it only returns the last error, it also indicates if there are no errors, and was instrumental in my coding to ensure that I was not causing any errors on the GPU, and even more so for testing, so that I can determine when the array sizes are too large and the timings are invalid, rather than being completely unaware as I was before.

Another thing that my research told me, was that timing the kernel using the CPU timing technique we had been using was not the most accurate way to time the kernel's runtime. Instead, I utilized CUDA events, which essentially create a start and stop variables before the kernel is run, use a cudaEventRecord on start before the kernel is run, a cudaEventRecord on stop after the kernel runs, along with an event synchronize, and finally a cudeaEventElapsedTime to record the timing result to a float. From what I have read, this is the most accurate way to record timings on the GPU.

To explain the logic of my program, I will start with the beginning. First the arguments are read, checked that they are valid, and then converted to integers because at this point it is know that they are integers and atoi will work properly on the string values. Then the program checks to see if the width of Matrix A equals the height of Matrix B. If these values are not the same, then the matrices cannot be multiplied. Rather than guess at which the user meant, the program exits with an error message. Most of the following checks, ensure that the matrix multiplication method will be able to get a correct result for the input. The program then checks to make sure that the dimensions of the resultant matrix, C, are each divisible by the block dimension, if not it changes both of these values to the same value, which is the closest integer that is divisible by the block dimension. For that calculation, and all following calculations I describe where a value is checked to be divisible by another number, and if it is not divisible, it sets the value such that it is divisible by the number, I use a method I created called nearestDivInt. Essentially, the method divides the value by the number and stores it into an integer. Then it multiplies this integer by the divisor, and the integer plus one by the divisor. Whichever of these values is closest to the original dividend is returned, making it the closest number divisible by the divisor. In the case where the dividend is smaller than the divisor, the method returns 0, but this is handled by setting the desired value to the divisor. Next, the program checks to make sure that the size of Matrix A and Matrix B are each divisible by the block size. If this is not the case for Matrix A, it changes Matrix A's height to the nearest divisible integer to the block size, and for Matrix B, it changes the width, because these values can be changed without affecting the capability of the matrices to be multiplied together, though these values will determine the size of the resultant matrix.

The last two checks are primarily to ensure that the kernel will run correctly and yield the correct results to the

resultant matrix, C. The program first ensures that the width of Matrix B is equal to the number of threads in the width of the grid, if not it changes this value. This check must be done because of the way the matrix multiplication is programmed. It is dependent on block and thread ID's to run properly, therefore if the grid is wider than the width of the matrix, then the indices will be thrown off, and the incorrect values will be stored in C. This could be removed if the elements were not added to shared memory based on the thread IDs, as long as the row is still divisible by the block size, which is a requirement already met, so it would not be a problem. Then it checks that there are more threads in the grid than elements to be processed and exits with an error if this is not the case. Initially I tried to make the code more versatile so it could handle this situation (remnants of this code still exist, because I intend to go back and fix this eventually, though they do not affect the current algorithm at all). Essentially I was going to take the same approach I took with the matrix addition, and have it calculate reps, an integer that represents the number of times that each thread should loop through and calculate another result, or in other words, the number of grid sizes it takes to fit the resultant matrix. With this, I was able to have C writing to the correct indices, however the problem was with the calculations, because yet again they were dependent on the thread and block IDs, and were thrown off on all iterations except the first. Again, this could be fixed by changing the method of storing elements to shared memory such that they do not rely on the thread and block IDs.

Lastly, I will explain the process of the actual matrix multiplication. Essentially I looped through sub sections/ sub-matrix blocks of Matrix A and B, storing these submatrices, which are the same size as the blocks, to shared memory, an array of shared memory for Matrix A and Matrix B each. What I needed to do this was a start value for A and B, as well as sizes of the blocks to iterate through. The start values for A and B would change depending on the number of repetitions required, number of elements each thread has to calculate, though this code will not be used unless the changes previously mentioned are made. The iteration through A is quite simple because it iterates by block size, and starts based on the block it is currently in, and the width of the array and the block size. This is simple because A iterates based on rows, which is how the array is stored in 1 dimension, however, B is more complicated because it must iterate by column. Initially I planed to rotate B so that it would also be able to iterate by row, and even begun coding an algorithm that would do this, though I realized that the rotation itself might be extremely inefficient, and although it wouldn't be included in the kernel run time, I decided to not waste my time in something that might not be very efficient, and might take me too long to get working properly. It would not only require time, but additional memory with a temporary array, so I found both of these reasons convincing enough to carry on without rotating Matrix B such that its orientation would appear to be by row.

The loop that iterates through the blocks, goes through the blocks until it reaches the last A block needed to calculate the current element being worked on, which also gets it the last B block it needs. The A value was used as the condition to stop the loop, because the loop could be stopped by either A or B, but A seemed simpler because of its simpler iteration through blocks. Once in the loop, the thread stores a value from Matrix A into the shared memory of Matrix A for the block, and a value from Matrix B into the shared memory of Matrix B for the block. The values are picked based on an index calculation, which is the part that depends on the thread IDs. This would most likely need to be changed to make the solution more versatile as mentioned earlier. A thread sync is called, because each thread in the block is adding an element to shared memory, such that each of them can independently make calculations in parallel. Then a for loop from 0 to the block size (such that each element from shared memory is used as needed), a value from Shared Matrix A, which iterates by row, and a value from Shared Matrix B, which iterates by column, (this part actually through me off for a while because I forgot that threadX is actually the column not the row etc.) are multiplied together, and added to float that sums up the values. This is also synced to ensure that the threads do not race ahead to the next calculation (this actually may not be necessary now that I am thinking about it, based on the implementation, but it is still included to be on the safe side). Outside of the loop through the blocks, the index of the resultant matrix is calculated (this method would also need to be changed for versatility), and value that was summed by the multiplication is set to the correct element in the Matrix C. After all threads run, the correct value is stored in Matrix C, and the memory is written back to the CPU from the GPU, and the timing result is printed, as well as the GPU errors if there are any.

Table 1: Small Matrices

Grid Width	Grid Height	A Height	A Width	B Height	B Width	Block Size	Time
80	100	1024	400	400	1280	16	0.0146
80	100	1000	400	400	1600	20	0.0256

Table 2: Block Size

Grid Width	Grid Height	A Height	A Width	B Height	B Width	Block Size	Time
1000	1000	10000	10000	10000	10000	10	30.0525
625	1000	10000	10000	10000	10000	16	27.2838
500	1000	10000	10000	10000	10000	20	38.0546
1000	1000	10000	10000	10000	16000	10	43.6542
2000	2000	16000	16000	16000	16000	8	143.9530
1600	2000	16000	16000	16000	16000	10	172.8966
1000	2000	16000	16000	16000	16000	16	139.5945
800	2100	16000	16000	16000	16000	20	206.7267

Table 3: Grid Height

Grid Width	Grid Height	A Height	A Width	B Height	B Width	Block Size	Time
625	1000	10000	10000	10000	10000	16	27.2838
625	1800	10000	10000	10000	10000	16	49.1097

**All tests were run on gpu2, with multiple users on the machine, and all timings are the result of an average of 5 runs. Not as many tests were run as I would have liked because it was difficult to get runs in when there were so many users running scripts.

Block Size:
From the testing results it seems that there is definitely a peak block size, which seems to be 16 in most cases. When testing square matrices (for simplicity) that were 10,000 x 10,000, the timing was 30.0525 seconds with a bock size of 10, 27.2838 with a block size of 16, and 38.0546 with a block size of 20, which is essentially the largest block size that can be run without going over the 512 thread per block limit. Again, when testing large matrices of 16,000 x 16,000, the timing was 143.9530 seconds with a block size of 8, 172.8966 seconds with a block size of 10, 139.5945 with a block size of 16, and 206.7267 with a block size of 20. Yet again, the timing was fastest when the block size was 16. It seems that this is the fastest because the warp size is 32 threads, and block sizes of 16 allow for 8 full warps, where as the block size of 8 allows for only 2 full warps, and the block sizes of 10 and 20 do not allow for an exact number of warps, but a number of warps, and some left over threads.

Grid Size:
Increasing the grid height should not affect the time, however, I realized that my code does not check for an out of bounds element until it tries to store the element to Matrix C. Therefore, a greater grid height, means more unnecessary work being done, however, this work is not needed. So the best time would be the smallest possible grid height to fit the elements. Grid width affects the matrix size, (because of the method used to multiply matrices) and cannot be isolated and tested in the same way. This is why the time increases substantially when the grid height is increased even though the matrices' sizes remain the same.

Matrix Size:
Changing the matrix sizes so that Matrix A is taller than Matrix B, or Matrix A is wider than Matrix B, or vice versa, does not appear to affect the timing as long as the size of the resultant Matrix C remains the same. Both of these tests

got about 49 seconds.

**Note: more timings would have been done if it did not take as long to get onto the GPU. Would it be better to do the testing on my own NVIDIA chip on my laptop, though not as powerful to get relative timings in the future?

Table 4: Matrix Size

Grid Width	Grid Height	A Height	A Width	B Height	B Width	Block Size	Time
625	1800	10000	16000	10000	10000	16	49.1100
1000	1800	10000	10000	10000	16000	16	49.1096