Programming Assignment 2: Cuda Lab

Colin Drewes (PID: A15616787)

Zihao Kong (PID: A15502295)

Abstract

In this assignment we have optimized matrix multiplication for NVIDIA's Kepler GPU. GPU's are a vector for massive parallelism due to their number of threads. This makes them a natural choice for optimizing matrix multiplication. We utilized on-chip memory and efficient matrix blocking to achieve on average 500 Gflops/s on UCSD's sorken compute cluster. This paper will provide an analysis of our eventual method, and various performance benchmarks.

1 Development Flow

1.1 a

Our algorithm is inspired by Vasily Volkov's pseudo-code in *Benchmarking GPUs to Tune Dense Linear Algebra*[VD08]. The most significant draw from this paper for our team was the mixing of the on-chip shared memory for matrix B, and the use of the register file for vectors of matrix A. Volkov's original algorithm can be read in the referenced paper but here is a representation of his main idea:

Algorithm 1: Vasily Volkov's GPU Matrix Multiplication

```
Registers: a,c[1:16];
Shared memory: b[16][16];
Compute pointers in A, B and C with thread IDs while pointers in range do
    b[1:16][1:16] = next 16x16 block of B;
    for i ← 0 to 16 1 do
        a = next column of A;
        c[1] += a*b[i][1];
        c[2] += a*b[i][2];
        ...;
        c[16] += a*b[i][16]
    end
    local barrier;
    update pointers;
end
Merge c[1:16] with block of C matrix;
```

This has a few important differences from our implementation. Most importantly

this implementation is only for one "set" of threads, and not complete matrix multiplication. Implementing this requires "blocking" the original matrices so that we may perform the multiplication as described by this algorithm. With this said, we will describe our algorithm in depth for a specific example block size. While this is configurable with the bx and by flag to the make script, the most general values from out tests seem to be 16x16.

1.1.1 Grid Dimensions

Inside of *setGrid.cu* we have the opportunity to set the dimensions of our GPU "grid." This will determine how many "blocks" in the x and y direction we will have. In practice this will determine into what chunks we subdivide the input matrices A, B and C. First we subdivide the y dimension of the matrix into N / 16 blocks, where N is the size of the matrix. This is done by setting $gridDim.y = \frac{N}{BLOCKTILE_M}$. Next we subdivide the x demension of the grid. While in previous designs we proceeded very similarly by setting $gridDim.x = \frac{N}{BLOCKTILE_N}$, with this approach we have unrolled the matrix multiply and are calculating 16 vectors of C simultaneously. The unrolling is detailed in Volkov's presentation *Better Performance at Lower Occupancy*[Vol10]. It is for this reason we set $gridDim.x = \frac{N}{BLOCKTILE_N * BLOCKTILE_M}$ or in this case $\frac{N}{16*16}$. With this in place we can return to the details of our algorithm.

1.1.2 Initialization

First we define an array of C vectors, which are going to be held in registers. These vectors will be used to hold the results of one blocked matrix multiplication. We define an array of of 16 double elements, which the compiler will recognize as an array of vectors as we will see. We are performing loop unraveling on the naive approach which we would only consider a single vector of c at once. Because we do not know the previous state of the registers of C, we will zero them out by iterating through each of the elements in the array and setting the vector to be zero. This is quite fast as we effectively have a different thread performing each one of these tasks at once. Next we declare the shared memory for matrix B. Because we will be accessing B on a consistent basis it is more efficient to store it using the on-chip memory, even though this does occur some copying overhead. Finally, we initialize the register column as described by Vasily in his algorithm.

1.1.3 Multiplication

We are not ready to begin the actual matrix multiplication. This becomes much more implementation specific and where we deviate somewhat from Vasily's outline. His outline states that the multiplication continues as long as the matrix pointers remain in range. We will be iterating down matrix "blocks" instead for greater clarity. The incremental value will be $BLOCKDIM_N = 16$, this choice will become more clear later. First we must perform the copying from global GPU memory to

the on-chip memory for the particular block of B we will be utilizing. To achieve this we gather a column of threads with the threadIdx.y macro. This series of threads are just consectutive values to begin with, and we have to shift them to the appropriate location. First, we add block which is the index of the current block in the y direction. This will shift all of the threads to the appropriate block. Then, by multiplying by the matrix size N the threads will be wrapped around to form a column of threads. We wish to gather a 1D array of 2D blocks of B so we do similarly for the x direction. We multiply the blockIdx.y by the $BLOCKTILE_M = 16$ and add threadIdx.x to get a row of threads for each of the blocks. Finally, when we add these two quantities together we get a 1D array of 2D blocks as desired. We use this to have a thread copy a value of B to the on-chip shared memory we defined earlier. At this point there is error handling if the block size does not divide N we zero pad the remaining values. After this point we sync the threads because we need to have all of the values of B ready before we continue. Next, we iterate through the current block to multiply the columns of A with rows of B. We do this with the loop from 0 to $BLOCKTILE_N$. First we wish to gather the rows of A for the multiplication. To get to the appropriate row within the block we calculate $(block + i) * N$. The block value times N takes us to the appropriate block in the y direction. The i value times N takes the threads to the appropriate row within the block. Finally, we perform the multiplication by iterating through the vectors through each of the c vectors and summing the a row times B's column. This is done in much greater detail in the code but this is the rough analysis.

1.1.4 Write-Back

After each panel of C has been calculated, we must re-sync the threads before we can write the C registers to memory. Once this is complete we get a set if threads for each block in the grid. For each of these blocks we iterate through each vector in c and write that vector to C matrix in GPU global memory. We perform more bounds checking in this case to ensure we do not write more of C than is allocated in cases where the blocksize does not divide N. This completes our matrix multiplication implementation.

1.2 b

1.2.1 Shared Memory

It was abundantly clear from the start we would need to use on-chip shared memory for some amount of A and B. Our initial implementation loaded both of A and B into shared memory. This differs from our final implementation where we use registers for A and B in shared memory. Copying the matricies to shared memory incurs some overhead, but the overall runtime improvement was significant. Our naive approach which copied both A and B to shared memory increased our performance significantly to about 230 Gflops/s. However, based on Vasily's paper it was clear that this stressed certain blocks too much and slowed down performance. This is why we pivoted to just loading matrix A into registers and B in shared memory.

1.2.2 __restrict__

While working on our naive implementation with A and B loaded into shared memory, we tried to increase performance with *__restrict__* operator. This operator is used to specify pointers as being read only, which can allow the compiler to potentially put them in faster RO memory. By adding the restrict operator to the A and B matrix in our naive implementation our performance jumped to almost 300 Gflops/s. This seemed to be the most performance we could extract from this implementation, hence our shift to be more in line with Vasily's structure.

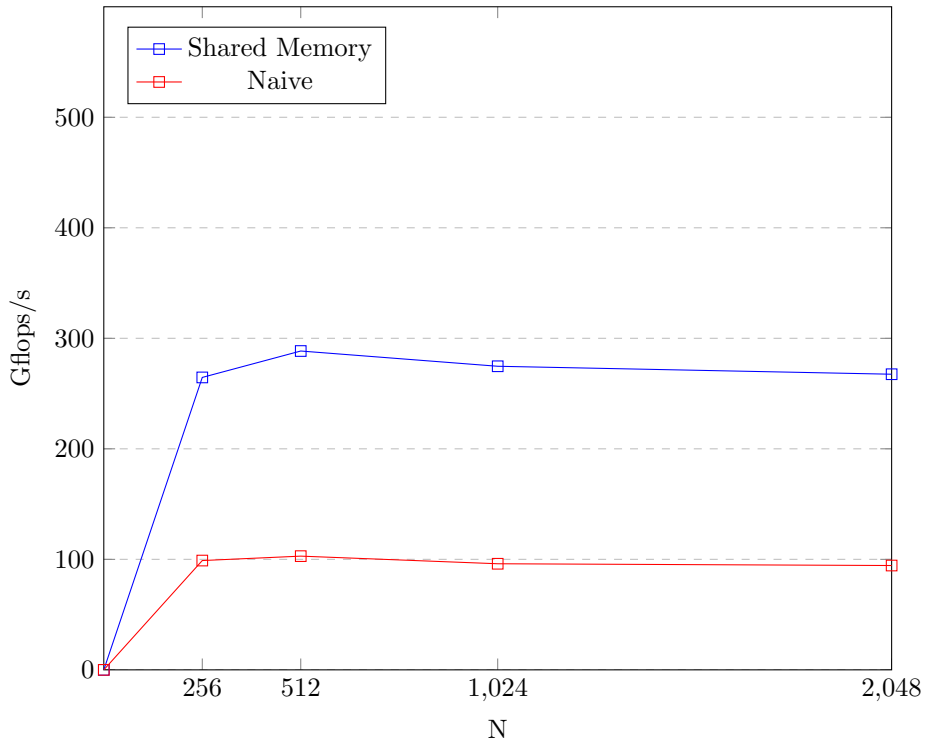
1.2.3 Register Use

As previously mentioned the key to improving our performance was loop unraveling through Vasiliy's method. This also involved moving matrix A to use registers instead of shared on-chip memory. Registers are a faster memory interface, but we have less of them, and so we shifted A to use them. The loop unrolling and shifting to greater register use resulted in a significant performance increase to about 500 Gflops/s on average. This is where we concluded our development.

1.3 c

What worked well was restructuring our memory hierarchy to favor on-chip and register memory. Our first attempt relied entirely on shared memory for matrix A and B. These were our rough performance results:

Performance Comparison Over Select N



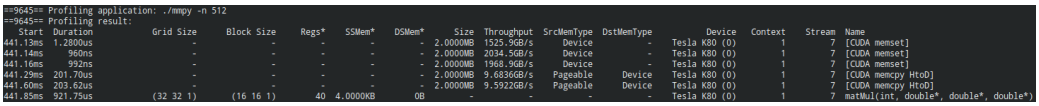
The naive implementation is just the starter code as given to us. In both cases we are using a bx and by value of 16. It was clear that restructuring our memory hierarchy to favor local memory was going to instrumental in our eventual solution. We observe the difference in shared memory use through the following command:

nvprof –print-gpu-trace ./mmpy -n 512

Though this prints out information for every call to matMul, we can just compare across a single one. From the naive implementation we see no use of shared memory:

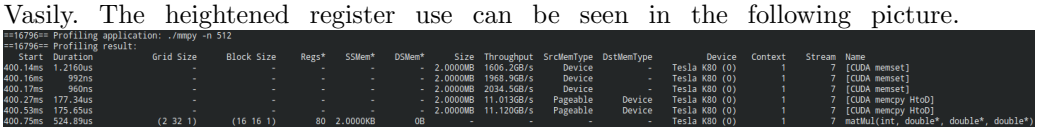
```
==7541== Profiling application: ./mmpy -n 512
==7541== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType          Device   Context    Stream  Name
335.91ms  1.2160us                    -               -         -         -         -  2.0000MB  1606.2GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
335.93ms     960ns                    -               -         -         -         -  2.0000MB  2034.5GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
335.94ms     960ns                    -               -         -         -         -  2.0000MB  2034.5GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
336.03ms  179.30us                    -               -         -         -         -  2.0000MB  10.893GB/s    Pageable      Device    Tesla K80 (0)         1         7  [CUDA memcpy HtoD]
336.29ms  179.39us                    -               -         -         -         -  2.0000MB  10.888GB/s    Pageable      Device    Tesla K80 (0)         1         7  [CUDA memcpy HtoD]
336.50ms  2.6170ms           (32 32 1)       (16 16 1)        32        0B        0B         -           -           -           -    Tesla K80 (0)         1         7  matMul(int, double*, double*, double*)
```

We see that no shared memory is used as well as minimal dependence on registers. Next we compared against the shared memory implementation where we expect to see a 2*16*16*64 block of memory being copied onto the chip. This is because we have a 16*16 block of A and B which all hold 64 bit floating point values. From this picture we can see that we are achieving as we desired.



```
==9645== Profiling application: ./mmpy -n 512
==9645== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType          Device   Context    Stream  Name
441.13ms  1.2800us                    -               -         -         -         -  2.0000MB  1525.9GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
441.14ms     960ns                    -               -         -         -         -  2.0000MB  2034.5GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
441.16ms     992ns                    -               -         -         -         -  2.0000MB  1968.9GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
441.29ms  201.70us                    -               -         -         -         -  2.0000MB  9.6836GB/s    Pageable      Device    Tesla K80 (0)         1         7  [CUDA memcpy HtoD]
441.60ms  203.62us                    -               -         -         -         -  2.0000MB  9.5922GB/s    Pageable      Device    Tesla K80 (0)         1         7  [CUDA memcpy HtoD]
441.85ms  921.75us           (32 32 1)       (16 16 1)        40  4.0000KB        0B         -           -           -           -    Tesla K80 (0)         1         7  matMul(int, double*, double*, double*)
```

However, registers are still a faster memory interface than shared memory, and it was Vasiliy's paper which made us realize we should emphasize this more. Before this we did try to reach our performance goal of roughly 450 Gflops/s through just this shared memory method. We altered the block size, tried using the restrict operator, used the pragma unroll, but it became clear that we needed the structural change as presented by Vasily. The heightened register use can be seen in the following picture.



```
==16796== Profiling application: ./mmpy -n 512
==16796== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType          Device   Context    Stream  Name
400.14ms  1.2160us                    -               -         -         -         -  2.0000MB  1606.2GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
400.16ms     992ns                    -               -         -         -         -  2.0000MB  1968.9GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
400.17ms     960ns                    -               -         -         -         -  2.0000MB  2034.5GB/s      Device           -    Tesla K80 (0)         1         7  [CUDA memset]
400.27ms  177.34us                    -               -         -         -         -  2.0000MB  11.013GB/s    Pageable      Device    Tesla K80 (0)         1         7  [CUDA memcpy HtoD]
400.53ms  175.65us                    -               -         -         -         -  2.0000MB  11.120GB/s    Pageable      Device    Tesla K80 (0)         1         7  [CUDA memcpy HtoD]
400.75ms  524.89us           (2 32 1)       (16 16 1)        80  2.0000KB        0B         -           -           -           -    Tesla K80 (0)         1         7  matMul(int, double*, double*, double*)
```

As can be seen the shared usage memory was halved and the register usage doubled. This proved to have a significant improvement on our performance as will become clear in the next section.

2 Results

2.1 a

We produce a trend line for tile size 8×8, 16×16, 16×8, 8×16. Our tile size is the same as thread block size, $BLOCKTILE_M$ is thread block's length, and $BLOCKTILE_N$ is thread block's height.

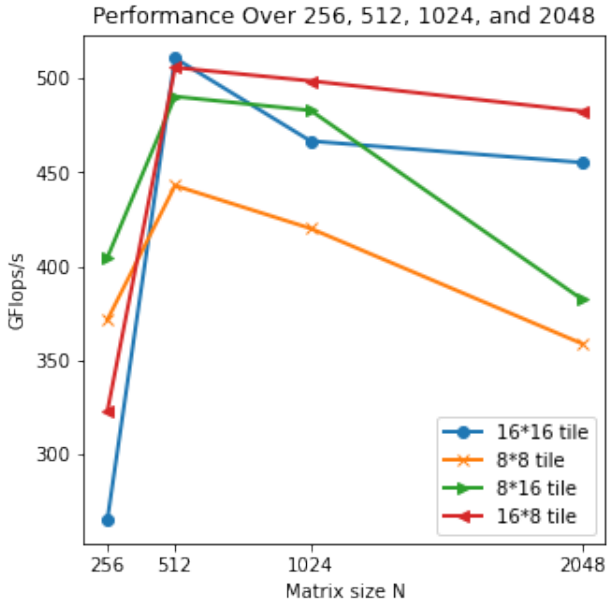


Fig. 1. Matrix Multiplication Performance over N=256 512 1024 2048

2.2 b

As shown in the graph, for N is equal to 256, the optimal thread block size is 8×16; for N is equal to 512, the optimal block is 8×8, and for N is equal to 16×16; for N is 1024, the optimal block is 16×8; and last, for N is 2048, the optimal block is 16×8. But in general, dealing with larger matrix sizes, a 16×8 thread block will outperform other choices.

2.3 c

For each matrix size, we recorded the peak performance with their thread block size.

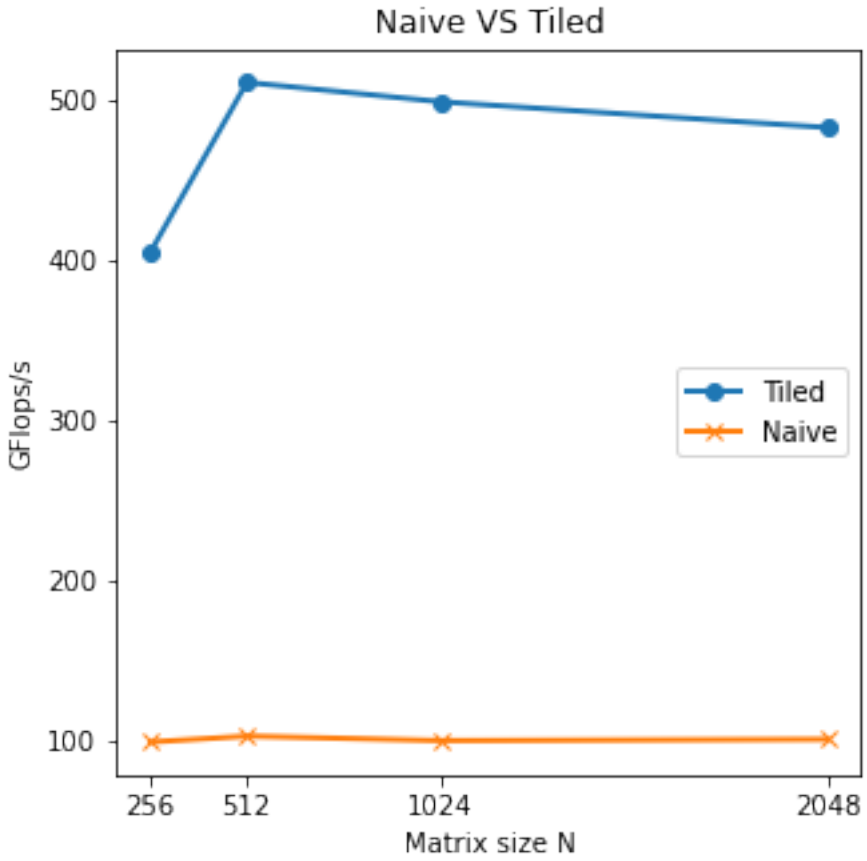
Table 1. *Tabel of Peak Performance*

N	Peak GF	Thread Block Size
256	404	8*16
512	510.8	16*16
1024	488.9	16*8
2048	482.6	16*8

3 Naive Comparison

3.1 a

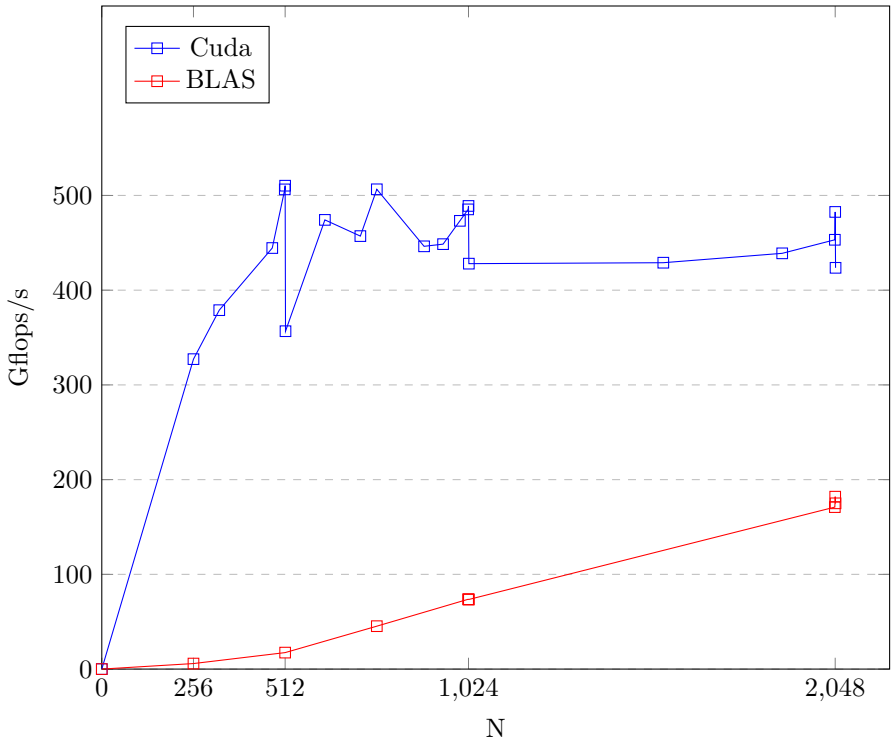
In Fig 2, we showed the performance difference between our optimal block size matrix multiplication and Naive matrix multiplication. The naive is around 100 GFlops/s and we reached around 4 times better performance.



4 Analysis

4.1 a

Performance Comparison Block Size 16*8



4.2 b

Our curve begins to reach its maximum Gflops/s value sooner than the BLAS values. In fact from this graph we cannot conclude yet whether BLAS has reached its maximal performance. From the graph our implementation reaches its best performance around matrices of size 512. However, BLAS is still increasing in performance all the way to the largest values of N. We also observe more variance in our implementation, especially around ± 1 any power of 2. This "jitter" will be addressed more in the next subsection. We do not expect the BLAS performance to match that of our GPU implementation just by the nature of the massive parallelism we can exploit with Cuda.

4.3 c

The most dramatic unusual dip in performance is the transition from N = 512 to N = 513. However, upon examining our code this is fairly expected. We divide the grid in its x direction into $N/(BLOCKTILE_M * BLOCKTILE_N)$ blocks. However, if $N\%(BLOCKTILE_M * BLOCKTILE_N)! = 0$, we add a new block into the x

direction. Furthermore, if $N\%BLOCKTILE_M! = 0$, then we have to expand the grid in the y dimension. In this case, $BLOCKTILE_M = 16$ and $BLOCKTILE_N = 8$, thus neither 16 nor 8*16 will divide evenly into N, thus we create a larger grid in both dimensions. We can verify this experimentally. By running nvprof with the –print-gpu-trace flag we get the following usage information for N = 512.

```
346.40ms  525.28us            (4 32 1)        (16 8 1)        80  1.0000KB        0B
```

This is what we expect, because 16 divides 512 in 32 and 16*8 divides 512 by 4. There is no wasted space in this situation. However, at N = 513, we get the following usage information:

```
367.85ms  759.67us            (5 33 1)        (16 8 1)        80  1.0000KB        0B
```

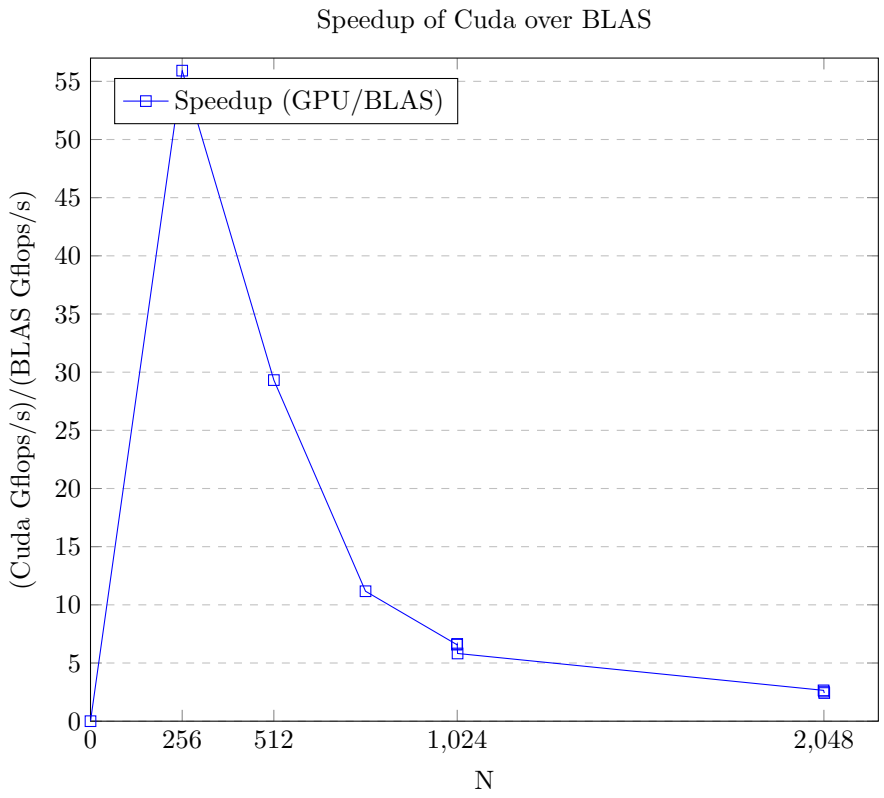
The grid dimension has expanded on both axes. However, because of the square geometry of this grid we will effectively have 33 more blocks along the x axes and 5 more along the y axes. There are still computations being performed on these values, but there value is not at all important in the final computation. They still count against the runtime. This is a worst case scenario, where we have 38 more blocks for only a single row/column in the input matrices. Such irregularities continue to exist at larger values of N. For example, N = 1024, and N = 1025 sees another irregular drop in performance. However, because the number of blocks we have to add in these irregular case grows linearly with N, and the overall number of blocks grows as N^2, it will become less important for larger values of N. This is just because the number of blocks which are performing multiplication at the expected Gflops/s will outweight the number of blocks which are wasted filled with zero values and have no effect on the final computation. The rest of our graph follows fairly expectidly.

4.4 d

Table 2. *Tabel of Peak Performance*

N	BLAS(GFlops)	Our Result(GFlops)
256	5.84	326.6
328	NA	378.6
476	NA	442.9
512	NA	506.4
512	17.4	510.3
513	NA	356.7
623	NA	474.2
722	NA	457.1
768	45.3	506.5
900	NA	446.3
953	NA	448.6
1000	NA	473.2
1023	73.7	485.3
1024	73.6	488.9
1025	73.5	428
1568	NA	429
1900	NA	438.9
2047	171	453.2
2048	182	463
2049	175	423.6

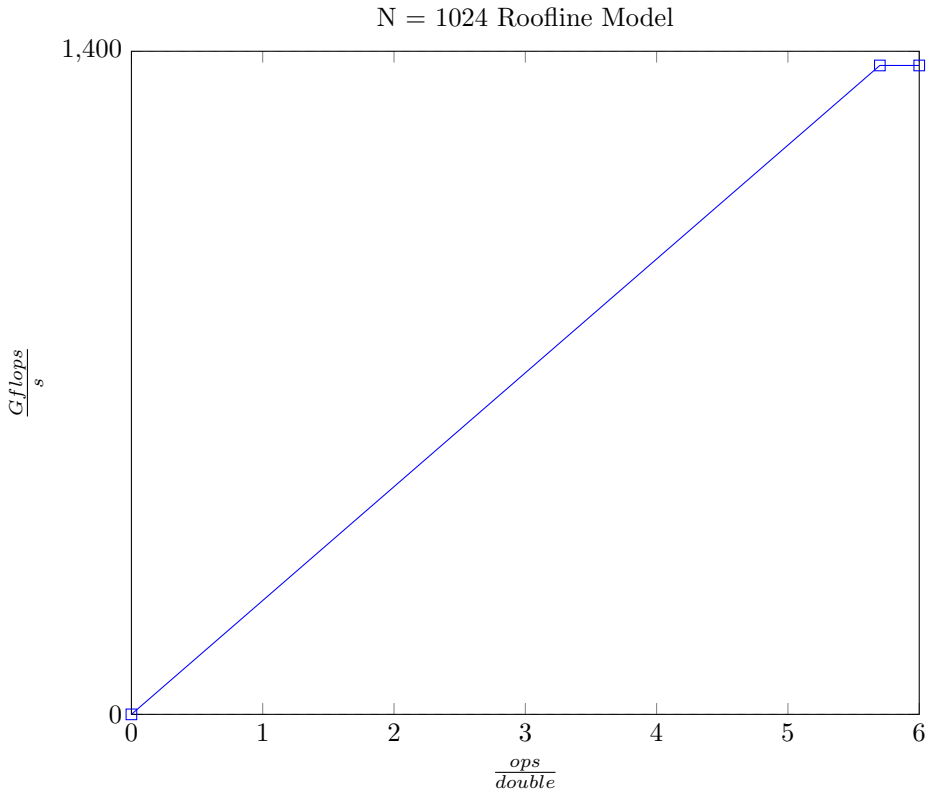
4.5 e


Speedup of Cuda over BLAS
Speedup (GPU/BLAS)
(Cuda Gflops/s)/(BLAS Gflops/s)
0
5
10
15
20
25
30
35
40
45
50
55
0
256
512
1,024
2,048
N

5 Bandwidth

5.1 a

When N is 1024, our algorithm achieve 488.9 GFlops/sec. We can calculate the peak performance, which is calculated by $2\frac{ops}{cycle} * 64\ \frac{cores}{SMX} * 13\ SMX \times 823.5MHz = 1.37\ \frac{TFlops}{sec}$. The slope of the graph is given by the bandwidth, which we are given as $240\frac{GB}{s}$. Thus we calculate the point reached starting at (0,0) with slope 240 till the y coordinate is 1370 MHz. This results in the following graph. This is just the equation $y = 240x$, up to the roof. If we use the y value of 488.9 GFlops/s, the q value is $2\frac{ops}{double}$.



5.2 b

In Volkov's thesis he measures the theoretical bandwidth as being 154 GB/s[Vol16]. We then recalculate the value of q. The important point to note is that the slope in the last graph is the bandwidth 250. This is the equation $y = 154x$. The value of q them becomes $1370/150 = 8.9$. This is the expected result as we recall from lecture 3 that more bandwidth means steeper slop which means hit peak performance at lower q. This is the opposite where bandwidth is decreased so we hit peak performance at higher q.

6 Future Work

In the future, we can alternatively try CUTLASS method presented in Nvidia's website. We also did not thoroughly explore different partitioning sizes between BLOCKTILEM, BLOCKTILEN as well as the thread block size. We could also optimize more specifically for the particular GPU architecture.

References

[VD08] Vasily Volkov and James W Demmel. "Benchmarking GPUs to tune dense linear algebra". In: *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE. 2008, pp. 1–11.

[Vol10] Vasily Volkov. "Better performance at lower occupancy". In: *Proceedings of the GPU technology conference, GTC*. Vol. 10. San Jose, CA. 2010, p. 16.

[Vol16] Vasily Volkov. "Understanding latency hiding on GPUs". PhD thesis. UC Berkeley, 2016.