# CSE 260 PA #2 Report

Zhengyang Wang

zhw030@ucsd.edu

Fan Jin

f1jin@ucsd.edu

**You must follow this guideline for a well-written and organized report. Full credit will be awarded only if the guidelines mentioned in the Report format document are followed.** ***You are required to follow the format described below. That is, your report must clearly label the sections below. You may include additional sections (e.g Introduction) if you wish.***

****Document your work in well-written, 5-10 page (including figures) report [the length is just a guide, if you need fewer pages that's fine as long as you report the required information]. More than 10 pages is probably overkill. ( Reports that require extensive time for us to read will be penalized up to -10 pts if report is not organized well). The report presents your results, analyzes them, and offers insights as to why things behaved the way they did. If you improved your code in a sequence of steps, document the process.****

## Section (1) - Development Flow

### Q1.a) Describe how your program works (pseudo code is fine, do not include all of your code in the write-up).

Our final program follows Vasily Volkov's idea in 2020[1]. We utilize shared memory as well as Instruction-level Parallelism and find lower occupancy can also achieve great performance.

First of all, our program performs blocked Matrix Multiplication in GPU kernel. We divide Matrix C into multiple blocks of TA*TB, and then each thread block will handle the calculation of one C block. Therefore, each thread block will first load one A block of TA*TW, and then one B block of TW*TB. After that, the thread block will perform the calculation of corresponding C block and save it back to the Matrix C.

Since Kepler-based GPU has a large amount of shared memory, we can use them to get rid of the slow loading from the global memory. Therefore, we load block A and block B respectively into our pre-defined share memory part. In addition, our program implements different Instruction-level Parallelism by setting different values of TA, TB, TW as well as bx, by.

We test our program by many parameter combinations and finally we choose block A of 64*16 and block B of 16*64. As for bx and by, we choose 16*16 to leverage occupancy, achieving 16 outputs per thread in our program.

The core algorithm for computation is following:

```
Threads in a block that handles block C in [I, J]
Each thread(tx,ty)  handles 4*4 cij in a block C
c[0:15]=0
shared memory: a[64][16], b[16][64]
```

```
For kk from 0 to total number of block (in terms of TW)
    For each Block A in row I
        Load Block A into Shared a
    For each Block B in column J
        Load Block B into Shared b
    Local barrier
    For each subblock( threadblock size) in Block C ( order : k, i ,j )
        Thread handles corresponding cij in subblock:
        c[i*4+j] += a[ty+i*16][k]*b[k][tx+j*16]
    Local barrier
Save back into Matrix C
```

**Q1.b) What was your development process? What ideas did you try during development?**

First, we used shared memory approach, following the code example in class. We choose 32*32 as our thread block size, which makes full use of shared memory. Without any further optimization, we achieved the performance of 270 GFlops when n = 1024. The program improves the performance by near 200 GFlops compared to naïve method.

Then, Instruction-level Parallelism (ILP) was adopted with 4 instructions per thread in parallel. Each thread handles 4 outputs of Matrix C. In details, we choose 32*32 as our block size for block A, B and C. As for thread block, it's 32*8 correspondingly. The performance achieved was between 340 and 380 for the given test n, indicating that ILP works well. Following this success, we refactored our code to support different matrix block size as well as thread block size. After trying different combinations of the output number per thread and matrix block size, with 16 outputs per thread, we achieved over **600 GFlops when n = 512, 1024 and 2048, and near 380 GFlops when n = 256**. All of our results meet the requirements of this assignment.

For 16 outputs per thread, we keep bx*by as 16*16, the size of block C as 64*64. We observe a little difference when we adjust TW for block A and block B, and then we finally choose TW=16.

We also find that the default Grid dimension is based on the size of thread block. Since our thread block handles larger matrix block than itself, the default setting will allocate more thread blocks than we need, which might cost more time to manage those blocks. Thus, we modify the setGrid.cu to make it consistent with our real block.

**Q1.c) What ideas worked well, what didn't work well, and why. Feel free to plot or chart results from experiments that did or did not end up in your final implementation and, as possible, provide evidence to support your theories.**

During our experiments, we find there is a trade-off between Instruction-level Parallelism and Occupancy. The block size is equally important for both thread block and matrix block.

Following the order of our attempt, the use of shared memory leads a large improvement.

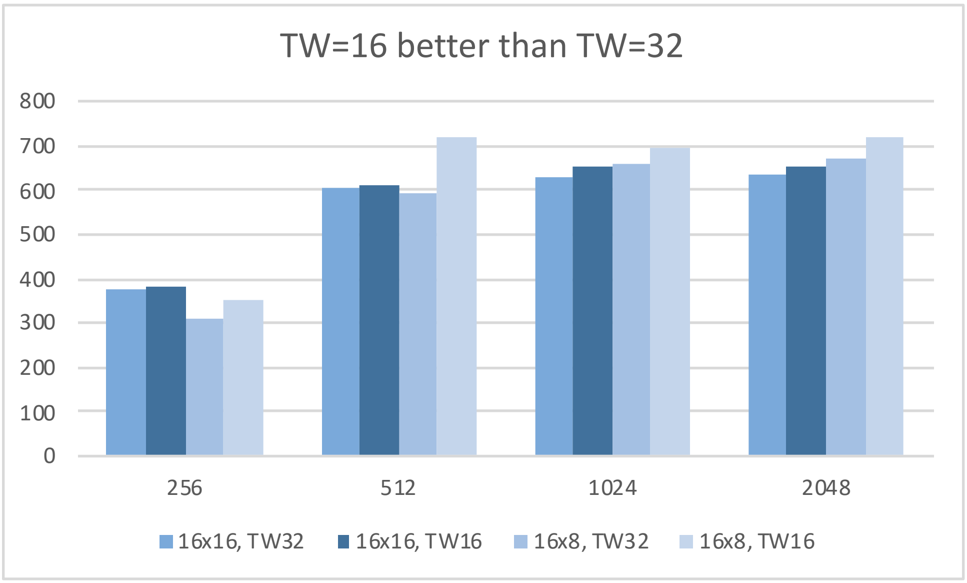
With the increase of thread block size as well as the corresponding shared memory, we can find the performance increases. It's because we improve the occupancy of our program. At first attempt, we only consider the performance on 512, and tries different approach to do simple optimizations. The following chart shows the improvement when we follow the course slides.

Simple Attempt when n = 512

| Naïve | 16*16 block | 32*32 block | 32*8 block with 4 outputs ILP |
|---|---|---|---|
| 91.7 | 226.5 | 257.4 | 361.7 |

So far, all optimizations mentioned in slides show their effectiveness in our Matrix Multiplication. Then we refactor our code to explore more about ILP.

Our first finding is that when we use 16 outputs per thread as ILP can produce an impressive result on the given test points. However, there are many ways to achieve 16 outputs per thread. For example, when we fix bx and by as 16 respectively, and also select block C size as 64*64. The size of block A and block B can vary, which depends on the selection of TW. (A is 64*TW and B is TW*64). From our first intuition, a larger TW will make each thread handles more operations, which will improve the performance.



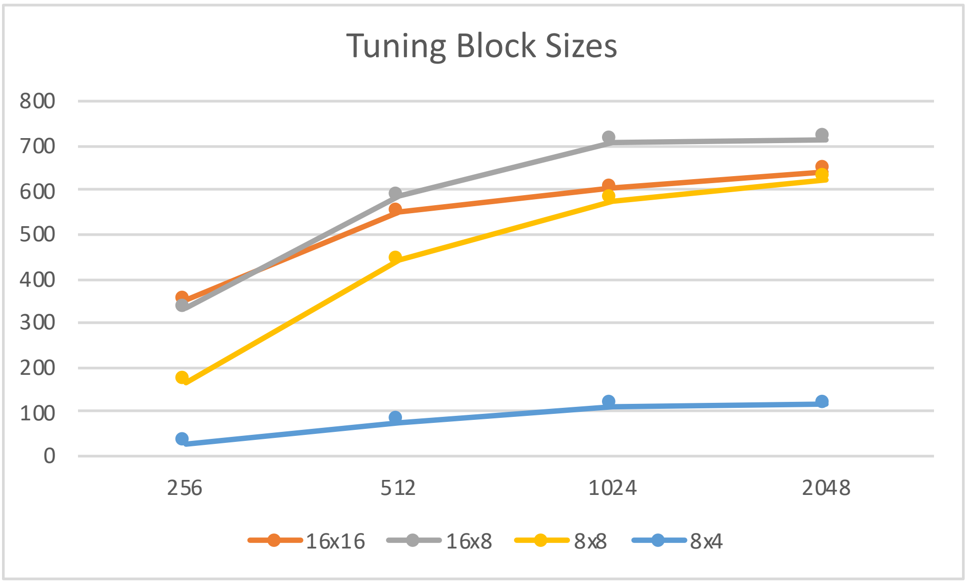
However, the result is different. The result shows that when we fix block size and outputs per thread, the performance will drop a little when TW increases to some extent. We think the reason is the change of arithmetic latency. When TW increases, although the calculation of each block C will require less times to sync threads, the arithmetic latency, however, increases. Plus, a higher TW requires more shared memory space, causing memory overhead.

## Section (2) - Result

**Your implementation will be graded on its performance at the following matrix sizes: n=256, n=512, n=1024, n=2048**

**Q2.a) For the problem sizes n=256, 512, 1024 and 2048, plot the performance of your code for a few different (at least 3) different block sizes.**

**That is, for n=256, 512, 1024 and 2048 plot the performance for at least 3 different block sizes. If your code has limitations on block size, please state the reason for that limitation.**



**Q2.b) Your report should explain the choice of optimal block sizes for each N(matrix size). Why are some sizes or geometries higher performance than others?**

Choice of block size: The block sizes should be divisor of TW, which is 16. The smaller the block size, the more ILP we favor. There are 16 outputs per thread and most TLP when bx=16, by=16, which has good performance. There are too many outputs per thread when bx=4, by=4, thus the performance is much worse. For different matrix size N, we find block size pair (16,8) and (16,16) has the similar performance when N is small and (16,16) is better than (16,8). But when N increases, the performance of (16,8) shows advantage over (16,16). This is because smaller matrix N favors TLP more, the merits of ILP will become obvious when N is larger. In summary, both (16,16) and (16,8) achieve the balance between ILP and TLP.

**Q2.c) Mention the peak GF achieved and the corresponding block size for each matrix size analyzed in the previous questions in the table.**

| N | Peak GF | Block Size |
|---|---|---|
| 256 | 382.9 | 16*16 |
| 512 | 719.1 | 16*8 |
| 1024 | 693.0 | 16*8 |

| | | |
|---|---|---|
| 2048 | 717.2 | 16*8 |

Conclusion: Although 16*8 has better peak GF for most N values, we would rather pick 16*16 for final submission because its performance is qualified for N=256. The aligned sizes (multiple of 64) have better performance for two reasons. First, there is no padding block. Second, our implementation uses a trick to judge if N is multiple of 64. If so, we use a reduced version and get rid of a lot of conditional branches inside the loops. We believe the second reason dominates as branches are very expensive.

## Section (3)

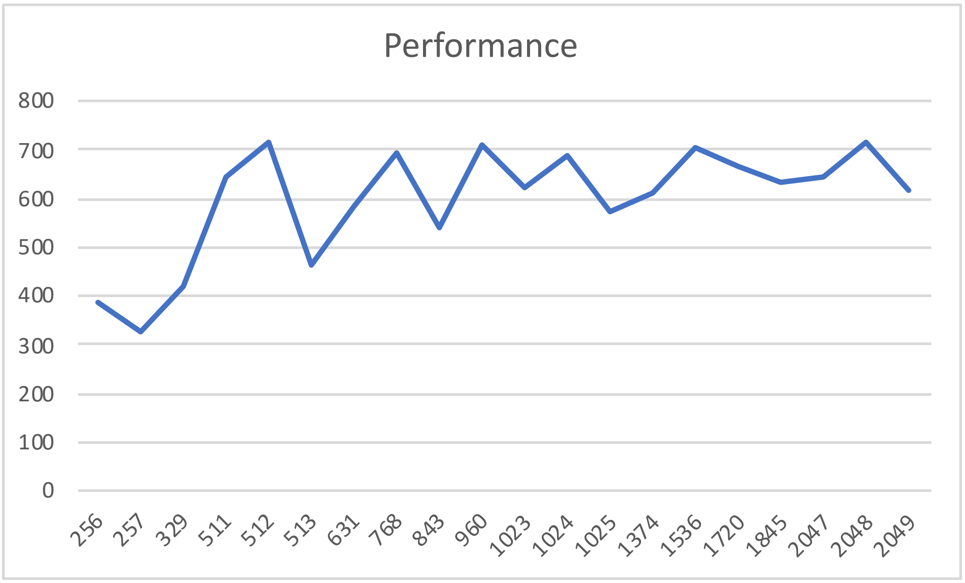
**Q3.a) For n=256, 512, 1024, and 2048 compare your best result with the naive implementation.**

| N | Best Result | Naïve Implementation |
|---|---|---|
| 256 | 382.9 | 88.5 |
| 512 | 719.1 | 91.7 |
| 1024 | 693.0 | 90.2 |
| 2048 | 717.2 | 87.7 |

Conclusion: For N=256, our result has a speedup of 4. For larger N values, the speedup is around 8. The naïve implementation is not sensitive to the change of matrix size and has little difference in terms of performance. Our implementation has the most advantage over the naïve one when N is large.

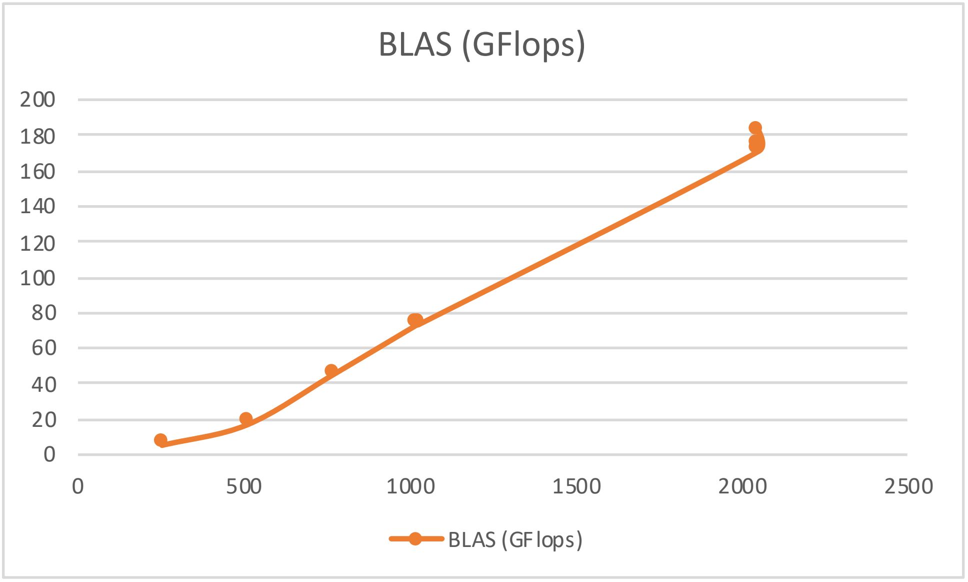
## Section (4) - Analysis

**Q4.a) For at least twenty values of N within range (256 - 2049), plot your performance using the best block size you determined for n=1024 in step (2). Use at least the values in the table below, but add other values too. Compare your results to the multi-core BLAS results in the table below. (for the values we gave you in the table. For other values, just report your cuda numbers).**



Conclusion: Despite occasional dips for smaller Ns, we reach the goal of 380 GF/s for N=256 and 600 GF/s for larger Ns. Performance gains are observed at multiple-of-64 N values.

**Q4.b) Explain how the shape of your curve is different or the same to the BLAS values and theorize as to why that might be. You may refer to either this plot or the plot of speedup (5).**



Our curve has a common shape with BLAS in that we have lower GF/s at smaller N due to copy and device initialization overhead. However, our curve has much less performance drop at small N values compared to BLAS. For example, our performance at N = 256 is about 50% as much as N = 1024, while the BLAS is only 10% as much.

Another difference is the behavior around power-of-2 N values. The BLAS achieves same performance for N = 1023, 1024 and 1025, but out curve shows a huge performance gain at N =

1024 over its neighboring values. We attribute this phenomenon to the fact that our implementation gets rid of expensive conditional branch operations when N is a multiple of 64. Similar results can be found at N = 1536 and 2048. We conclude from this observation that the GPU achieves the best performance when N is a multiple of its TLP dimensions.

**Q4.c) For the twenty or so values of performance, identify and explain unusual dips or irregularities in performance.**

As mentioned in Q4.a, there is performance dip when N is relatively small. Also, we notice something different from the CPU kernel in PA1, where we meet performance dips at power-of-2 N values unless we fix it using other techniques. Here, we have performance gains instead, for N = 256, 768, 1024, 1536 and 2048.

**Q4.d) Please include the following table in this format in your report as well as a graph mentioned above: (you must have these n values but you should have more (20)).**

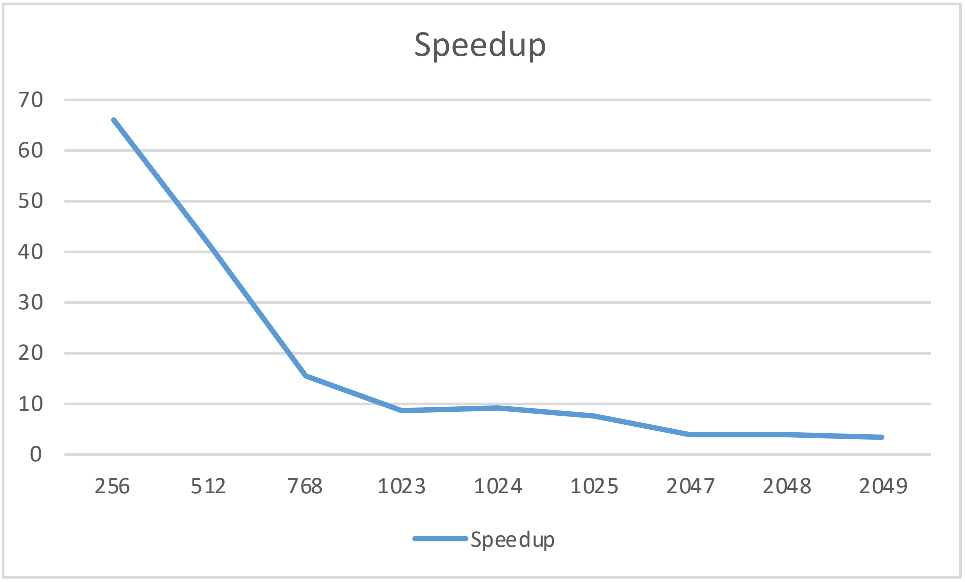
To meet the performance requirement, we choose bx=16, by=16 as our final submission parameters (in OPTIONS.TXT), which are not the best for n = 1024, but not bad for n = 256.

When n = 1024, the best performance is achieved by bx=16 and by=8. Therefore, **we report the results of both parameters and choose 16*8 to analyze in the report ( to meet report requirement), choose 16*16 to submit in the code( to meet performance requirement).**

| N | BLAS (GFlops) | Your Result (GFlops) (16x8) | Your Result (GFlops) (16x16) |
|---|---|---|---|
| 256 | 5.84 | 357.3 | 386.7 |
| 257 |  | 304.4 | 328.2 |
| 329 |  | 420.8 | 401.9 |
| 511 |  | 646.5 | 506.8 |
| 512 | 17.4 | 717.4 | 612.7 |
| 513 |  | 463.1 | 411.2 |
| 631 |  | 581.5 | 559.8 |
| 768 | 45.3 | 693.4 | 625.8 |

|  |  |  |  |
|---|---|---|---|
| 843 |  | 542.6 | 513.4 |
| 960 |  | 709.4 | 626.9 |
| 1023 | 73.7 | 623.6 | 577.6 |
| 1024 | 73.6 | 689.4 | 640.3 |
| 1025 | 73.5 | 573.3 | 512.0 |
| 1374 |  | 608.9 | 550.6 |
| 1536 |  | 704.0 | 634.1 |
| 1720 |  | 664.5 | 592.6 |
| 1845 |  | 630.8 | 563.0 |
| 2047 | 171 | 643.6 | 577.1 |
| 2048 | 182 | 717.7 | 640.7 |
| 2049 | 175 | 616.2 | 546.9 |

**Q4.e) Plot the above results as a speedup ratio (S).**



Conclusion: Opposite to the trend of speedup over naïve implementation, the speedup over BLAS is negatively related to N. This is due to the fact that BLAS performance grows much faster with N.

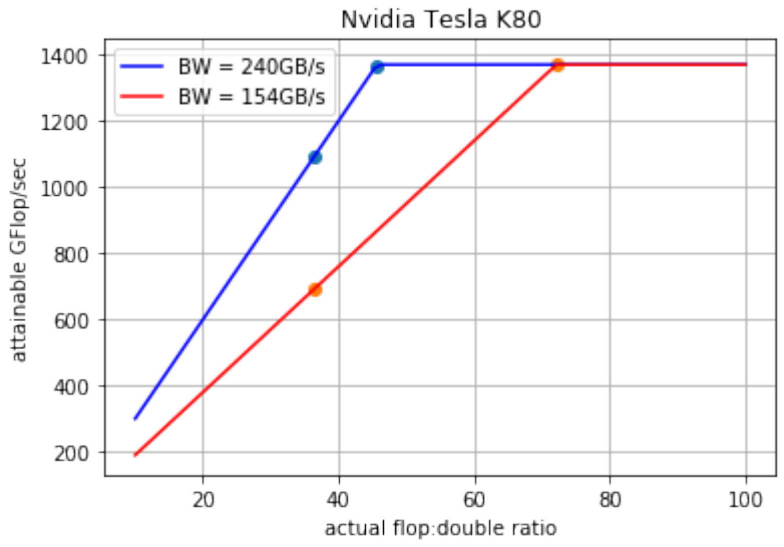
## Section (5)

**Q5.a) Using the raw bandwidth of 240GB/sec to global memory(theoretical max), plot a roofline model (log-log) or (lin-lin) for the GPU and plot your achieved n=1024 number on this plot.**

Estimated slope = estimated bandwidth = 240GB/s = 30G double/s

Roof line = peak GF/s = 13 SMX * 64 DP cores/SMX * 2 ops/cycle * 823.5 MHz = 1370 GF/s (K80 has 2GPU on it, in our experiment we only use one)

Estimated Optimal q = peak performance / slope = 1370/s / 30G double/s = 45.6.



**Q5.b) Estimate the value of q in ops/doubleword. Consider that the actual BW is less than 240GB/sec - Volkov thesis measures 154 GB/sec, plot this roofline and calculate the new "q" value. How has the value of q been affected by the change in BW?**

Old Estimated q = actual performance / old slope = 693GF/s / 30G double/s = 23.1

New slope = new bandwidth = 154GB/s = 19G double/s.

New optimal q = peak performance / new slope = 72.1.

New estimated q = actual performance / new slope = 693GF/s / 19G double/s = 36.5.

Consider the actual bandwidth, and the estimated q increases from 23.1 to 36.5.

## Section(6) - Potential Future work

**What ideas did you have that you did not have a chance to try?**

In this assignment, we introduce shared memory into our program and also explore the great impact of Instruction-level Parallelism. However, all the experiments we have done are focusing on GPU specifically characters on our initial simple Matrix Multiplication. Therefore, we have many things to try.

First, we can continue to mine the attributes of K80. For example, we know that K80 also has read-only memory and it might be used to cache something for multiple loading.

Second, we can try different form of Matrix Multiplication. Currently, all of our experiments are loading a row of block A, a column of block B to calculate a block C. We can change the order of calculation. For instance, we can follow Volkov's algorithm in 2008[2] or other Matrix Multiplication Routines we have known in Assignment 1. The different order of calculations may result in very different performance.

Finally, we noticed that no single configuration achieves optimal performance for all matrix sizes. For N < 512, block size 16 by 16 is the best. For larger N values, block size 16 by 8 is the best. However, we do not have the chance to select the configuration before entering the CUDA kernel, because we are not allowed to modify files other than the kernel. If we introduced a branch statement *inside* the kernel, it would cause conditioning divergence within a warp and, consequently, lead to performance drop. If allowed, we would modify mmpy.cu and put the conditional branch there.

The combinations on the thoughts of GPU Specifications and Matrix Multiplication Routines will always make sense. There is large room for us to try by trial and error.

## Section (7) - References (as needed)

[1] Volkov, Vasily. "Better performance at lower occupancy." Proceedings of the GPU technology conference, GTC. Vol. 10. 2010.

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