CSE 260 PA #2

Environment

Device Tesla K80, clock speed 823.5 MHz, Global memory11.17GB, Memory Bus Width 384 bits, peak memory bandwidth 240.5 GB/sec, L2 cache size 1536 KB.

K80 GPU, Kepler microarchitecture, 13 SM, each SM has 64 DP. Max threads/ SM = 2048, Max threads / Block =1024, Max block / SM =16, Max Warp / SM =64.

CUDA Driver version: 9010, runtime version: 7000

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).

We use blocked matrix multiplication. For each block of the output matrix C, a thread block computes the inner product of a block row of A and a block column of B. To do that, this thread block loop over every pair of a block of A and the respective block of B. Inside the loop, each thread first loads part of A block and part of B block into shared memory, then computes part of the output block of C, finally store it back to the global memory.

```
In register initialize output block of $C_{ij}$ to zero
FOR each pair of block $A_{ik}$ in A and the respective block $B_{kj}$ in B
    Load $A_{ik}$ into shared memory
    Load $B_{kj}$ into shared memory
    Synchronize all the threads
    Compute product of $A_{ik}$ and $B_{kj}$ into $C_{ij}$
    Synchronize all the threads
Store $C_{ij}$ back to global memory
```

To load block matrix into shared memory, naively each thread may load one element of it. However, to decouple the dimension of block matrix and the dimension of thread block, each thread should load more than one element, provided that block matrix can be divided into several subblock.

To compute the product of two matrix blocks, naively each thread computes one element of the output, i.e. inner product of row and column. However, to exploit instruction-level parallelism, each thread should compute several elements which have no data dependency issue. This can be easily achieved if output matrix is further divided into smaller matrix.To save matrix back to the global memory, again each thread saves the elements it computes.

When dimension of input matrix cannot be divided by our predetermined block matrix size, when simply zero pad the cross-boundary blocks and ignore the elements of output respectively.

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

1. First we downloaded the starter code and learned the naive algorithm on CUDA. Also we set up our working environment in the server and AWS.

2. With the help of slides, we tried blocked matrix multiplication. We divided input A and B into square subblocks with the same dimension as thread blocks. Therefore, each thread could load one element of A and one element of B and compute one element of C. Synchronization was needed because computation should happen after shared memory loading was finished.
3. Global memory access should be coalesced so we checked every global memory access in our code including loading of A, B and saving of C. We made sure threadIdx.x was used in second indices of two-dimension, since in global memory matrix are stored in row-major.

4. Shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously, so we also needed to avoid bank conflicts. Here row-major blocked matrix were stored in shared memory, so we made sure threadIdx.x was used in second indices.

5. To handle dimensions of input that are not divisible by predefined block size, we zero padded input wherever needed, then continued computation as before, finally only write back elements of C that were not out of boundary.

6. With zero-padding, all of our loops would actually run a fixed number of times that was determined in compile time. We instructed the compiler to unroll those loops so that loop condition tests were removed and instruction-level parallelism was enhanced.

7. Then we found that in our algorithm blocks of C would use the same blocks of A and B but loaded it multiple times, which was surely a waste of time and resources. To alleviate that, we could make each block of C larger. On the other hand, larger block size meant more shared memory was needed since blocks of A, B, and C were of the same size. But we could make blocks non-square. In that way, we could fit larger C blocks in shared memory.

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.

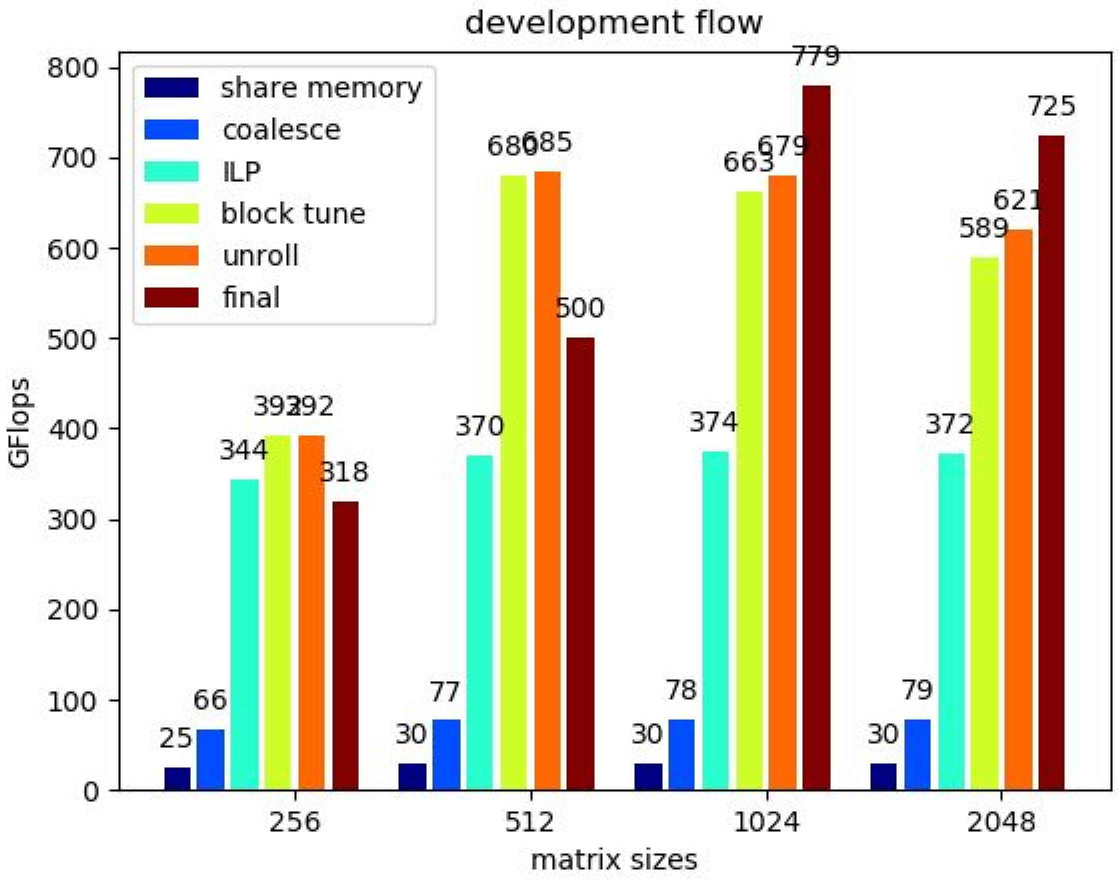
1. Blocking in shared memory works very well because shared memory has far less latency than global memory.

2. Coalescing global memory access and avoiding shared memory bank conflicts enables parallel memory operation across the warps so that the latency is minimized.

3. Doing more computation in one thread rather than distributing workload to more threads also helps a lot. Each SMX can host more blocks with fewer threads, enhancing parallelism among blocks. Computation of one element in C is inner product which has to be serialized. By allowing one thread to

compute more elements of C, instructions for different inner products can be parallelized, enhancing ILP.

4. Allowing blocks to be non-square enables us to utilize more shared memory and decrease the number of redundant loading of A and B. Suppose A block has dimension of M*K, B block K*M, C block M*N.



The above figure shows our development process. Staring from using shared memory, the performance is still poor due to the high loading time for loading matrix A and B from global memory to share memory. Then we implement the memory coalesce and further utilize instruction level parallelism to boost the performance. After that we shift to tune the thread block size and unroll the for loop to increase the ILP. Finally, we manage to fully utilize the whole shared memory on each SM. It can be seen from the figure, our final solution works very well when the matrix size is huge, but for smaller matrix sizes, the padding cost drops the overall performance slightly, as we enlarge the sub-block size and more padding operations are needed.

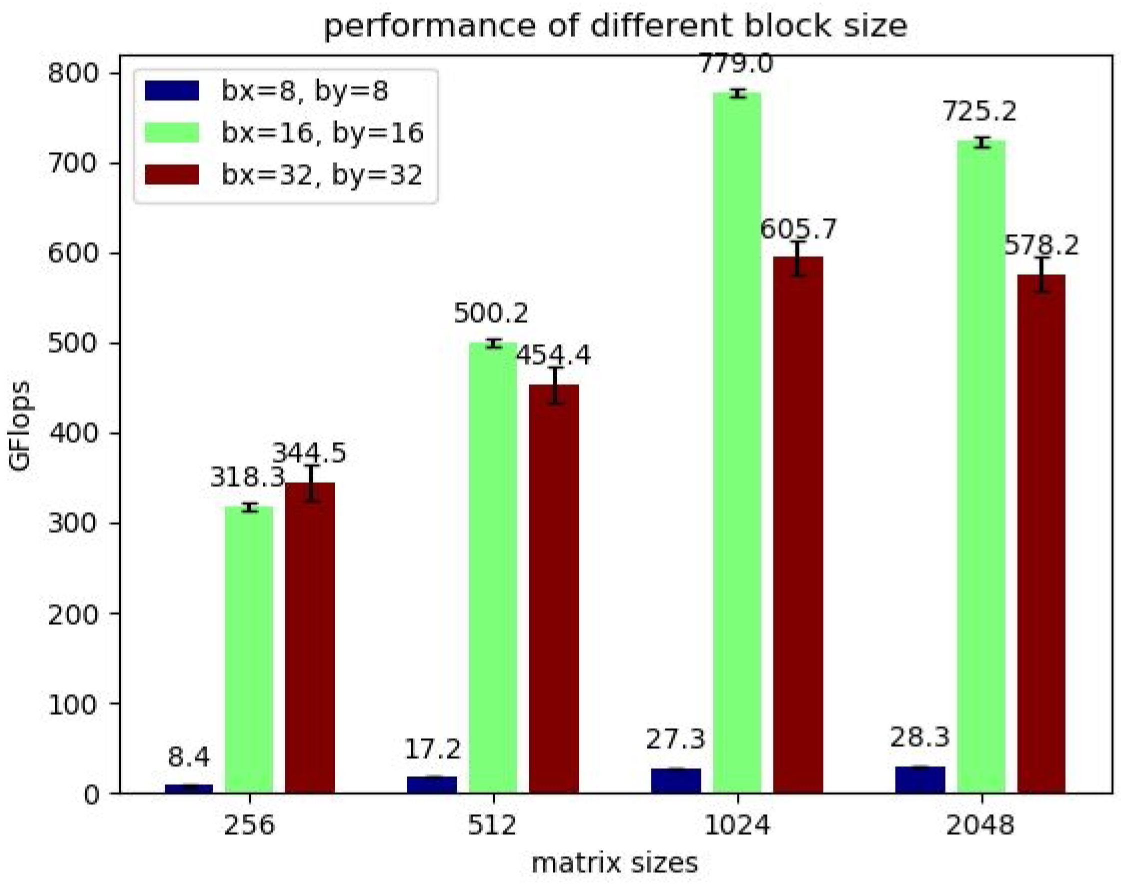
For the final version of our code, we optimize our block sizes to be 96, i.e. for matrix A, every time each thread block loads 96 *32 sizes, and for B is 32 * 96. Each thread block contributes to 96 * 96 element values in matrix C. The total shared memory used is 2 * 32 * 96 * 8 bytes, which can approximate the largest permitted shared memory of 0xc000 bytes without violating the memory coalescing rule.

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.**



We test on square block sizes, though our code also supports non-square configuration. But since there are at most 1024 threads in each thread block, bx and by can not exceed 32.

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?

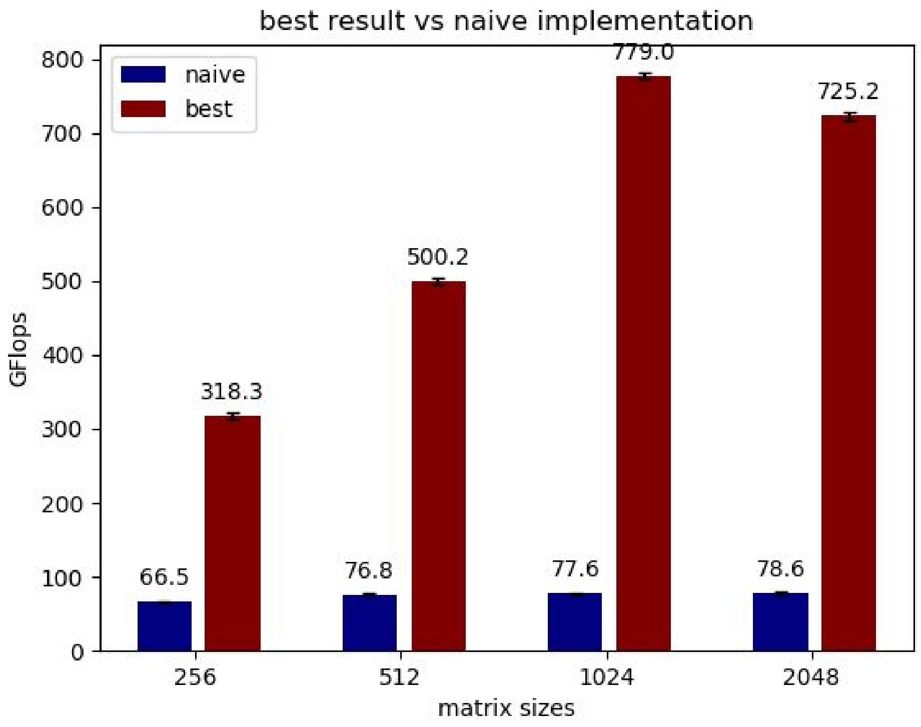
The optimal block size for our solution is 16 for both bx and by. It can be seen from the above figure, when the block size is small, in order to complete a 32 * 32 sub-block matrix in C, each thread is at heavy work loads and the occupancy is really small, thus influencing the performance seriously. When the block size is huge, it means the occupancy is high, but there are less instruction level parallelism and we can't fully utilize the cuda register resources, since there are 255 registers per thread.

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	344.5	bx=32, by=32
512	500.02	bx=16, by=16
1024	779	bx=16, by=16
2048	725.2	bx=16, by=16

Section (3)

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

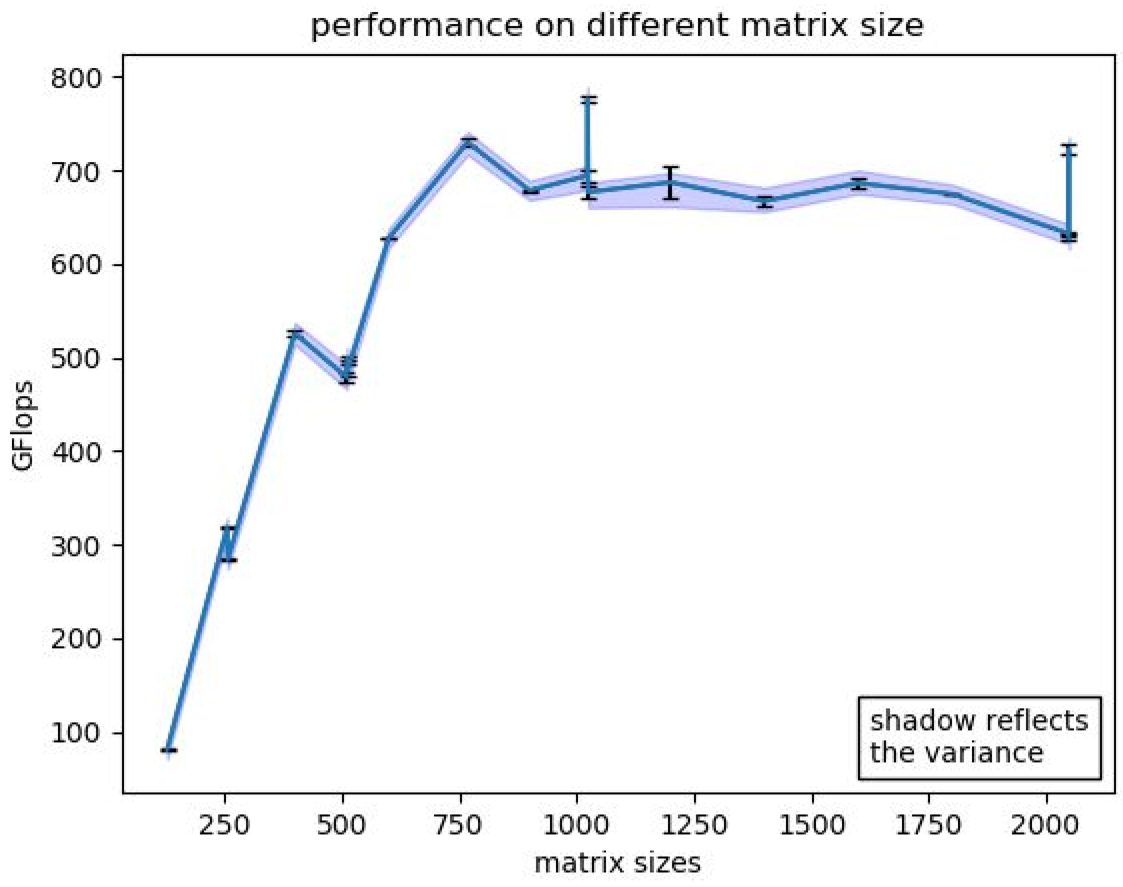


It can be seen from the figure, there is a huge gap between naive ones and our optimized ones. The number on each bar is the maximum performance in each size, and the interval stands for the range among the 3 measurements.

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).

We use square block size =16 for all the following experiments. For each matrix size, we repeat measuring 3 times. The median result is used for plotting. We also report the maximum and minimum each time. The plotting figure shows below.



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).

First of all, cuda code is much faster than the BLAS, as it leverages more memory bandwidth and computation resources. Secondly, they share the same trend, with the increase of the matrix size, the performance is boosting. This can contribute to the fact that we can better utilize the memory bandwidth with the increase of matrix size. Thirdly, for some specific values like 1024 and 2048, there is no significant increase of the GFlops in BLAS, but for cuda, since we can do memory coalesce to load much faster and also reduce the bank conflicts and padding cost, the cuda code can see a great jump in the performance curve.

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

The general trend is consistent with roofline models. At the beginning, with the increase of matrix size, the performance increases due to the utilization of memory bandwidth. After a certain level, the performance becomes stable. For specific values like 1024 or 2048, we further optimize our codes to do memory alignment and memory coalesce, so we can expect to see the huge performance boosting at these points. For some nearby points like 1023 or 2047, it can be really hard to align with the 128 bytes requirement, so the performance drops compared to 1024 or 2048. An interesting point is that for matix size equals 512, we expect it to be larger than 400 or so, but the fact is the inverse. We guess it may be caused by the compromise between matrix padding cost and memory coalescing. For small matrix size, the padding cost is low, but it can't fully utilize the bandwidth. For larger matrix size, the padding cost also increases. That may cause the irregularities in the plotting curve.

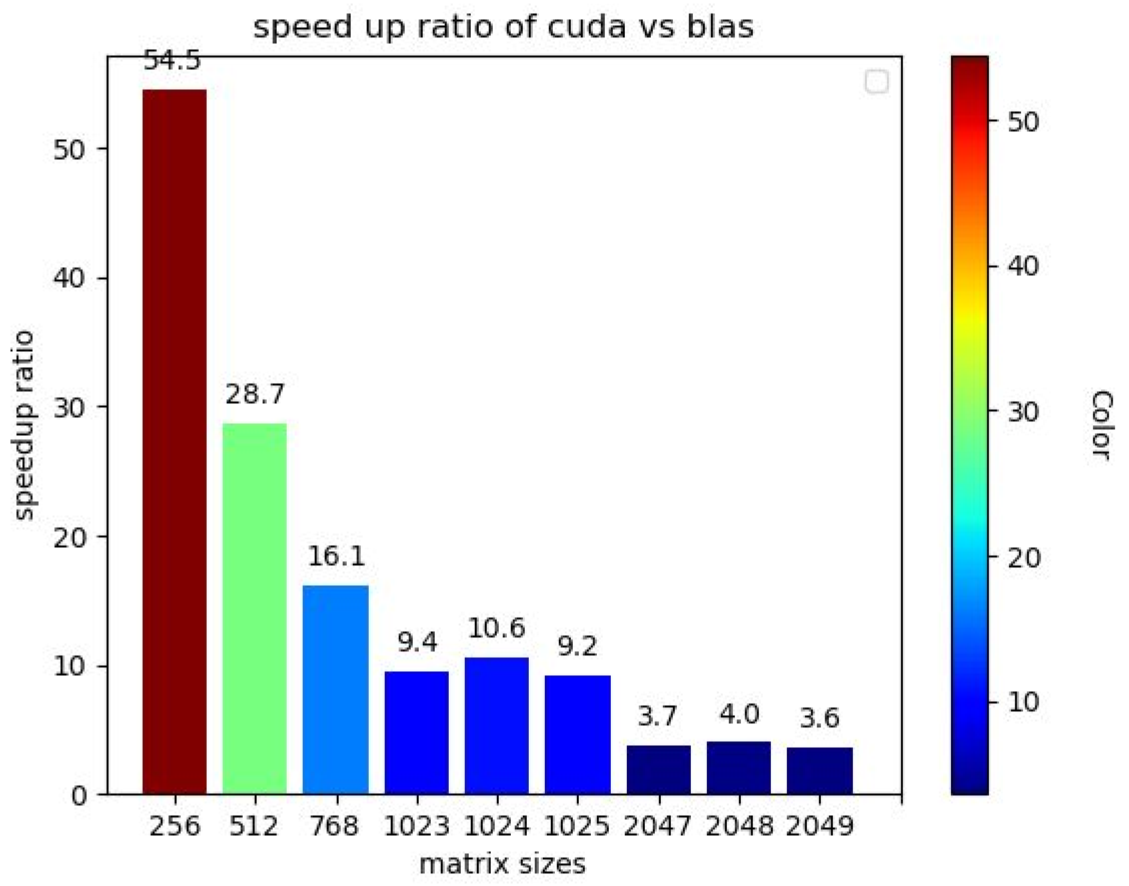
Q4.d) Please include the **following table in this format** in your report **as well as a graph mentioned above**: (you must have these n values but you should have more (20)).

The result in each cell reports the minimum and maximum GFlops achieved among 3 repetitions each time.

N	BLAS (GFlops)	Your Result (GFlops)
256	5.84	317.8, 318.3
512	17.4	497.3, 500.2
768	45.3	726.7, 730.8
1023	73.7	688.7, 694.6
1024	73.6	776.1, 779
1025	73.5	670, 676.6
2047	171	631.6, 632.8
2048	182	720.6, 725.2
2049	175	625.9, 628.2
128		80.9, 81.4
257		283.6, 284.3
400		524, 527.1
511		475.9, 482.3
513		483, 489.5

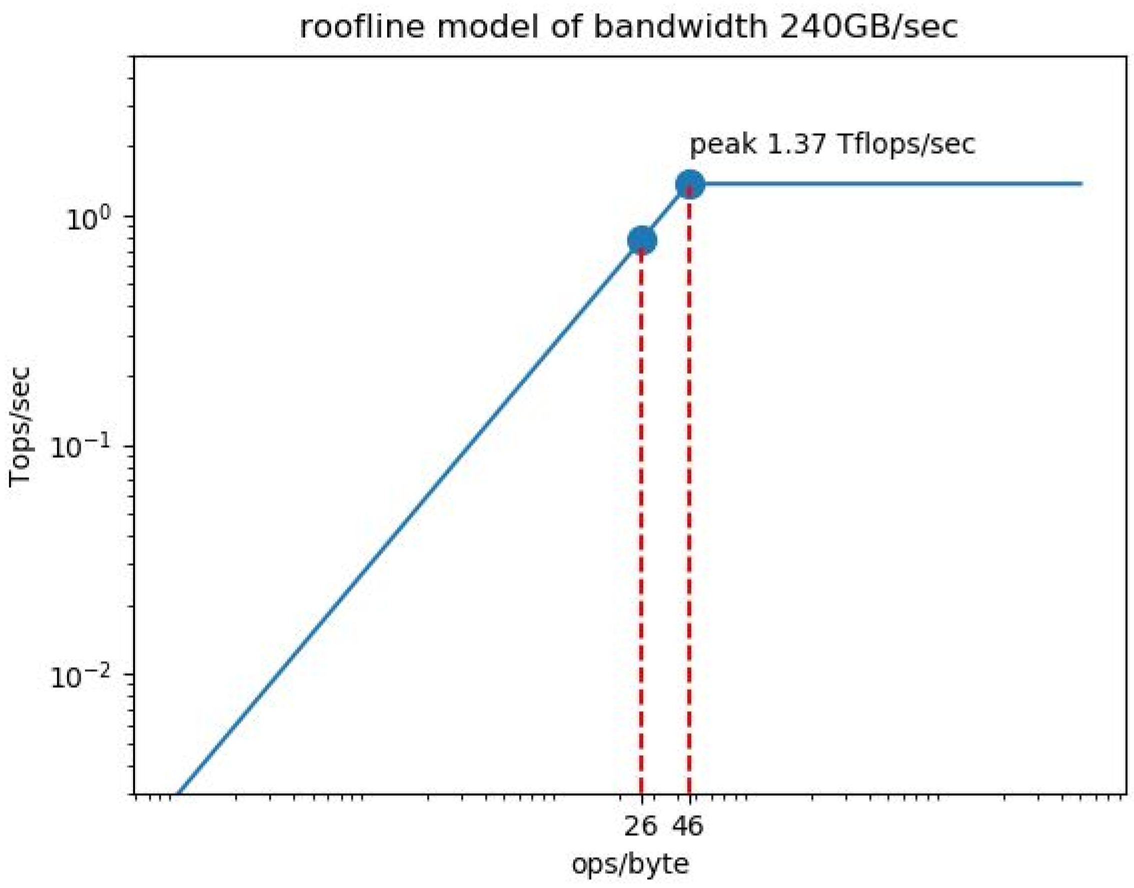
600		627.1, 627.8
900		677.9, 678.8
1200		671.1, 674.7
1400		665.6, 671.5
1600		684.8, 689.9
1800		674.1, 674.7

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

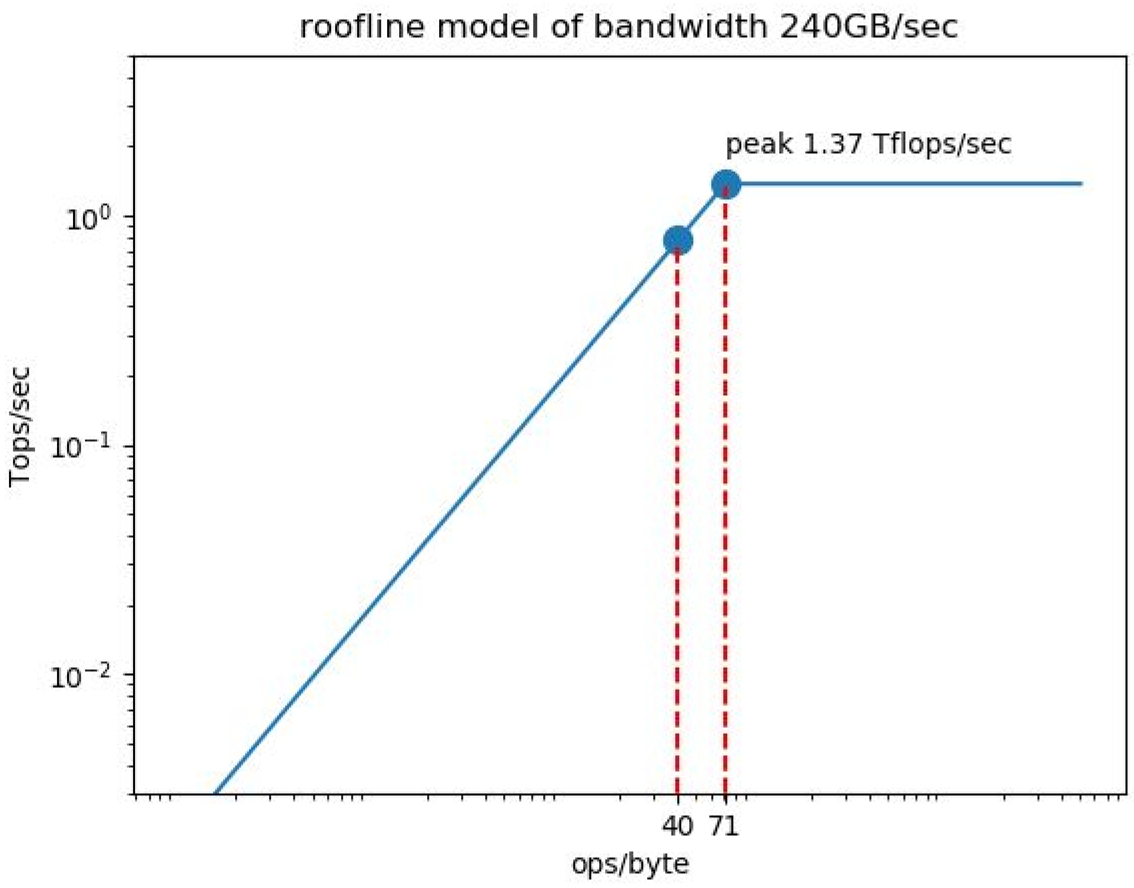


Section (5)

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.



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?



Under the condition of bandwidth 240 GB/sec, q is 26 opts/byte. Under the condition of bandwidth 154 GB/sec, q is 40 opts/byte. The estimated q increases as the bandwidth decreases. Because in order to achieve the same performance, we have to increase q to compensate for the loss of bandwidth so that we can catch up with the former case.

Section(6) - Potential Future work

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

We can further replace matrix inner product by outer product to use more registers. There are 255 registers for each thread in K80 GPU, but we only use 178 registers per thread. The speed of accessing thread is much faster than shared memory. So we can simply load a single matrix into shared memory and others put into registers. Then we can load much larger matrix blocks into shared memory and put more elements into register files, while conducting outer product like Volkov and Demmel's method[1]. We can also try prefetch operations inside our code to further decrease the memory loading cost.

[1] "Benchmarking GPUs to Tune Dense Linear Algebra." https://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf. Accessed 17 Feb. 2020.

Section (7) - References (as needed)

1. Ryoo, Shane, et al. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. 2008.
2. Garland, Michael, et al. "Parallel computing experiences with CUDA." *IEEE micro* 28.4 (2008): 13-27.
3. 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.
4. CUDA Optimization Tips, Tricks and Techniques", Stephen Jones GTC 2017