CSE 260 WI 2019 PA2 Report

Jiawen Fang
jfang@ucsd.edu

Xinyu Xie
x1xie@ucsd.edu

1. Method

1.1 Introduction

In this PA, we did blocked matrix multiply (MM) on GPU using CUDA.To improve the performance, we took the advantages of GPU memory hierarchy (register and shared memory of NVIDIA's streaming multiprocessors), and used thread-level parallelism (TLP) along with instruction-level parallelism (ILP) to hide memory latency and arithmetic latency. In the following of this section, we will describe our method in details and discuss some method we tried but not selected as our final submission.

1.2 Final Method

We followed the idea of Volkov's presentation [1]. In the final submission, we use thread block size of {16, 16, 1}, i.e. the blockDim.x = 16, blockDim.y =16. We divide the matrics into several sub-blocks with size of 32 by 32. Each thread block will compute the MM result of a sub-block of C, and each thread inside the block will deal with 4 outputs for the ILP. The detailed analyses of block size are are discussed in the Section 2. During the computation of every sub-block, the thread block will need to loop a row of A' sub-blocks and a column of B's sub-blocks. For each step of the loop, a thread block will load a 32 * 32 sub-block of A and a 32 * 32 sub-block of B into the shared memory since the value of these two sub-blocks could be shared multiple times during the computation to reduce the times of load from the main memory. When we finish the computation of a sub-block of C, each thread will copy the 4 values of C from the registers to the main memory. The pseudo codes of our method are shown in the Algorithm 1.

```
by: blockIdx.y,  bx: blockIdx.x
ty: threadIdx.y, ty: threadIdx.x
I: by * BlockSize + ty
J: bx * BlockSize + tx
DOUBLE CS[4] = {0.0}
for (i = 0; i < ⌈N / BlockSize⌉; ++i)
    load [by][i]-th sub-block of A into shared mem AS
    load [i][bx]-th sub-block of B into shared mem BS
    for (j = 0; j < BlockSize; ++j)
        CS[0] = AS[ty][j] * BS[j][tx]
        CS[1] = AS[ty][j] * BS[j][tx+BlockSize]
        CS[2] = AS[ty+BlockSize][j] * BS[j][tx]
        CS[3] = AS[ty+BlockSize][j] * BS[j][tx+BlockSize]
C[I][J] = CS[0]
C[I][J+BlockSize] = CS[1]
C[I+BlockSize][J] = CS[2]
C[I+BlockSize][J+BlockSize] = CS[3]
```

Algorithm 1. The pseudo code of our method.

1.3 Other Method

1.3.1 Method in Volkov and Demmel ‘s 2008 Paper

We also tried the Volkov and Demmel’s method [2]. We divide C and A into rectangular sub-blocks and divide B into square sub-blocks. Then we do outer product MM on sub-blocks of A and B. First we load a sub-block of C into register and load a sub block of B into shared memory. Each thread inside the thread block only deal with one row of C. Then each step of loop, we load a value from the same row of A into register and multiplied with a row of B. However, the performance of this method is not very good. It only got 200 GFlops for the input size of 1024 by 1024. Thus we don’t select it as our final submission. The potential reason for the poor performance is that this method is limited by the number of registers thus the shared memory is not fully used.

1.3.2 16-Output ILP

We also tried 16-output ILP method to improve the performance of our method. We use the same thread block size with our final method but different matrix block size. We divide C, A and B into sub-blocks with size of 64 * 64, 64 * 16 and 16 * 64 respectively. Thus we could use most of the shared memory. The performances of the 16-output method are shown in the Figure 1.

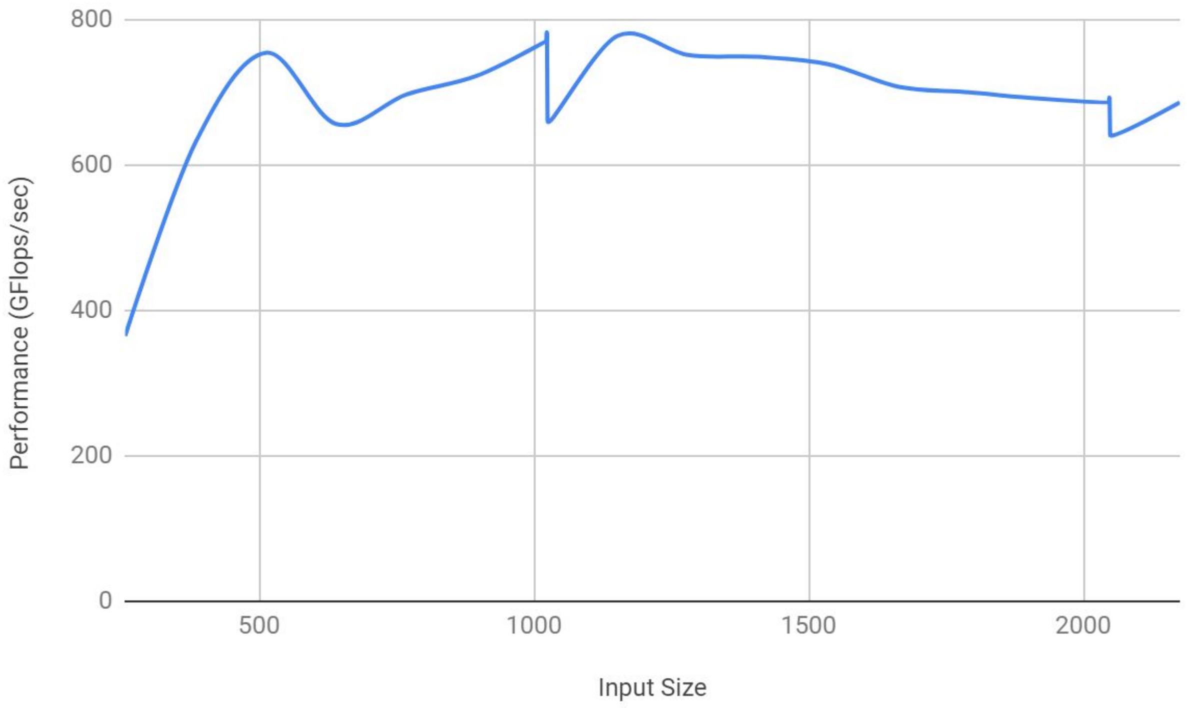

Figure 1. The performance of 16-output method.

However, the performance of 16-output method at input size of 256 * 256 is only 365 GFlops, very poor compared with larger input size. Though the performance of 16-output method at N of 1000+ is about **780 GFlops**, we still not selected this method as our final submission for the consideration on smaller sizes. The reason for the poor performance at smaller size may be that there are only 256 threads inside a thread block, but for input size of 256 * 256, the grid size is 4 * 4, thus the abilities of most SMXs are wasted.

2. Analysis on Block Size

In this section we will compare the performance of different thread block size. We did 3 sets of experiment on thread block size of 8 * 8, 16 * 16 and 32 * 32. For ILP implementation details, each thread of 8*8 block deals with 16 output while each thread of 32 * 32 block deals with 1 output. The 16 * 16 thread block are selected as our final submission. The performances of these three methods are shown in the Figure 2.

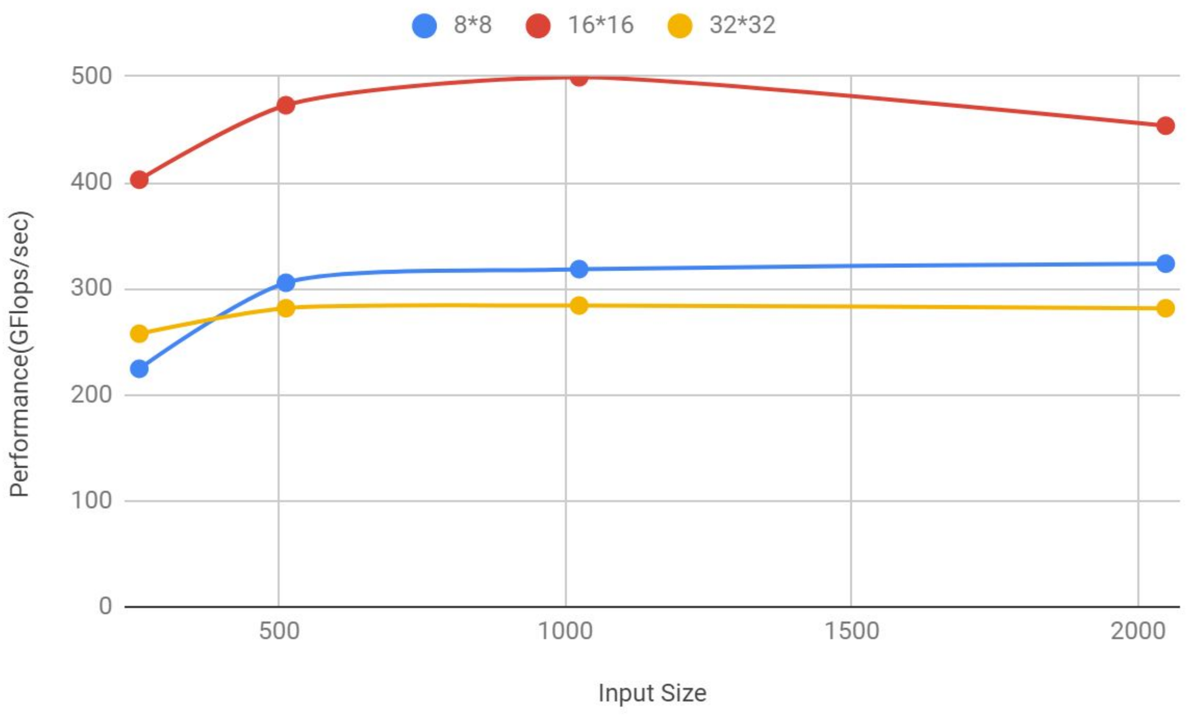

Figure 2. Comparison of different thread block size.

As shown in Figure 2, our 16 * 16 final version (red line) achieves 400+ GFlops on all input size from 256, 512, 1024 and 2048. While block size of 8 * 8 (blue line) and 32 * 32 (yellow line) versions only get about 300 GFlops performance. Since the matrix size is limited by the shared memory, all experiments were done using the same matrix size of 32 * 32. When we use a smaller thread block size (8 * 8), we do not make maximum utilization of available SMXs. However, larger thread block size (32 * 32) we use, less ILP we could make, which leads to more arithmetic latency. Thus we chose the thread block size of 16 * 16 as our final config to report performances of our method. The detailed performance of our method are described in Section 3.

3. Comparison with Naive Method

In this section we compare our implementation with naive version. In Figure 3, we can see that by tuning thread block size reasonably, we gain significant improvement compared to naive implementation.

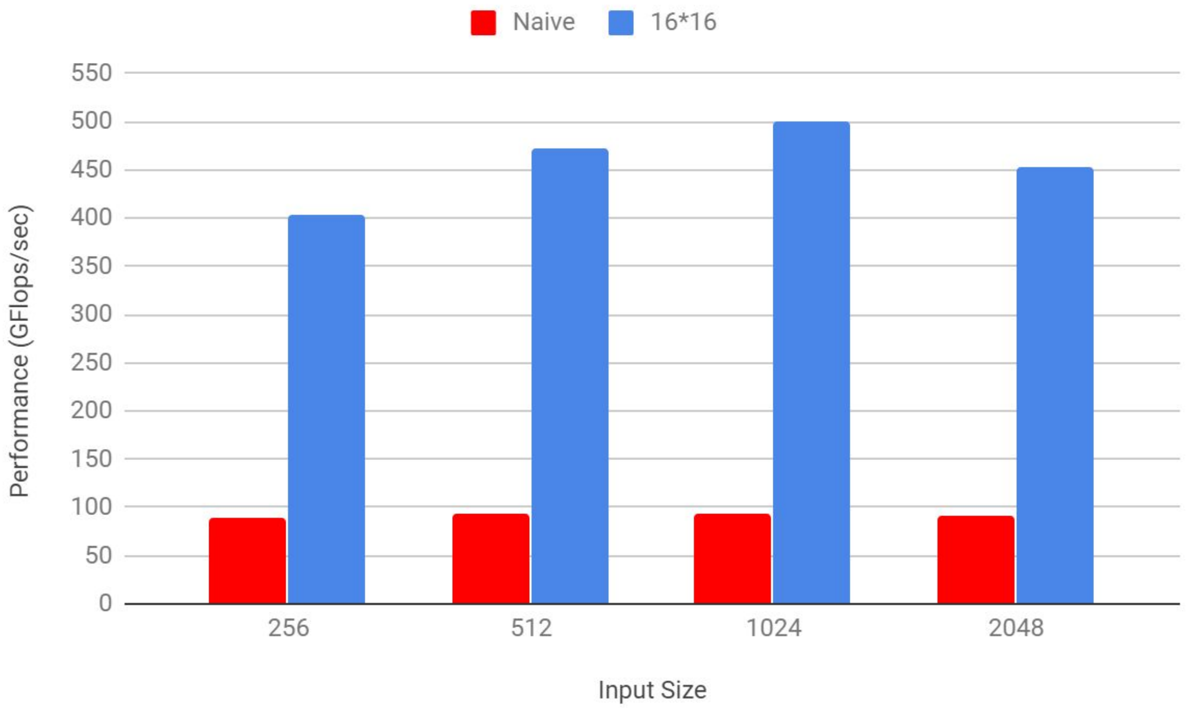

Figure 3. Comparison between our implementation and naive implementation.

4. Comparison with BLAS

In this section we compare our results with multi-core BLAS results. Table 1 and Figure 4 shows implementation using cuda have great advantage over the multi-core BLAS version. However, what’s interesting is that the shapes of the two curves in Figure 4 are very different. For our CUDA implementation, when the matrix size reaches some point, the performance is to some extent stable and decreases slowly, while for BLAS, as a whole trend, the performance keeps growing as the input size increases. In fact, as shown in Figure 5, as the input size grows, the advantage of our implementation to BLAS decreases.

The reason for this difference may result from several factors:

First, on GPU, thread size is much smaller than that on CPU while the thread number is much bigger vice verse. This means as the matrix size grows, the effects of ILP becomes greater and greater which overwhelms TLP.

Second, there might be something about cache. The cache on GPU is smaller than cache on CPU, which means the CUDA implementation is more limited by input size.

Finally, BLAS is a much more complicated implementation than ours. We think it may be able to dynamically adjust the thread size, threads number and memory usage strategy, which we fail to do in this work. Considering that another version of our work mentioned in section 1 shows great improvement compared with this one, and that we owe the failure of meeting requirements to static block size, we believe a dynamic implementation does matter a lot.

n	BLAS(GFlops)	Our Results(GFlops)	n	BLAS(GFlops)	Our Results(GFlops)
256	5.84	402.733167	1280		492.674817
384		435.064595	1408		480.566775
512	17.4	472.901606	1536		466.281698
640		476.711668	1664		479.514961
768	45.3	471.343963	1792		464.512117
896		497.555409	1920		457.059669
1023	73.7	496.145018	2047	171	446.958626
1024	73.6	499.294115	2048	182	453.610793
1025	73.5	454.27305	2049	175	426.104843
1152		498.315091	2176		451.393605

Table 1. Performance of our implementation and BLAS on 20 different input size.

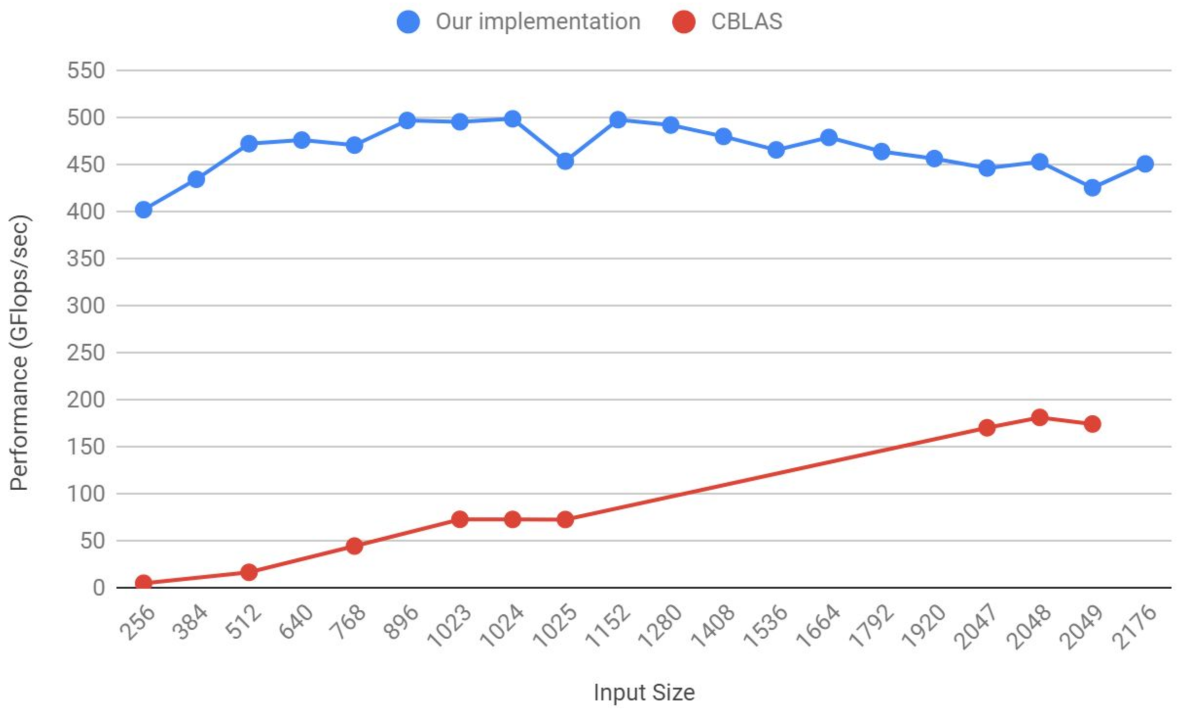

Figure 4. Comparison between our implementation and BLAS implementation.

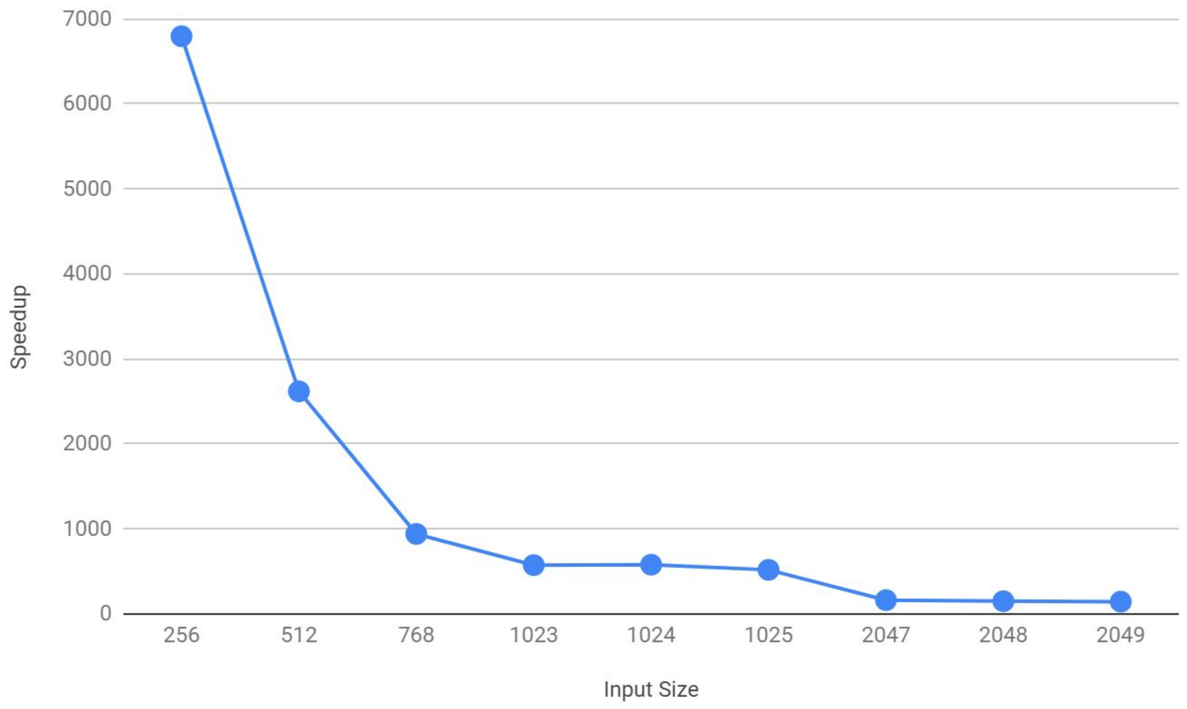

Figure 5. Speedup ratio compared to CBLAS implementation

We also notice that in Figure 4, the curve of our implementation shows some dips for those input size N which are not evenly divided by the thread block size (e.g. 1023, 1025 v.s. 1024; 2047, 2049 v.s. 2048). This is mainly because we deal with the non-multiple case by padding the block to the size we want with 0. For cases like 1024 v.s. 1025, we can see a certain performance drop caused by many additional sub matrix mostly filled with 0 (i.e. about 32+32=64 more warps are needed for 1025 case). Though the matrix size increases by only 1, the padding results a certain increasing in computation and load/save operations.
Another reason for improvement decrease is due to control divergence. For those cases where padding is not necessary, all threads within a warp can visit the same memory address, and thus the command can be broadcast to all chips at one time. But in other cases, since threads within a warp do not fall into the same branch, broadcasting cannot be done and commands must be assigned case by case, which reduces efficiency.

5. BW Analysis

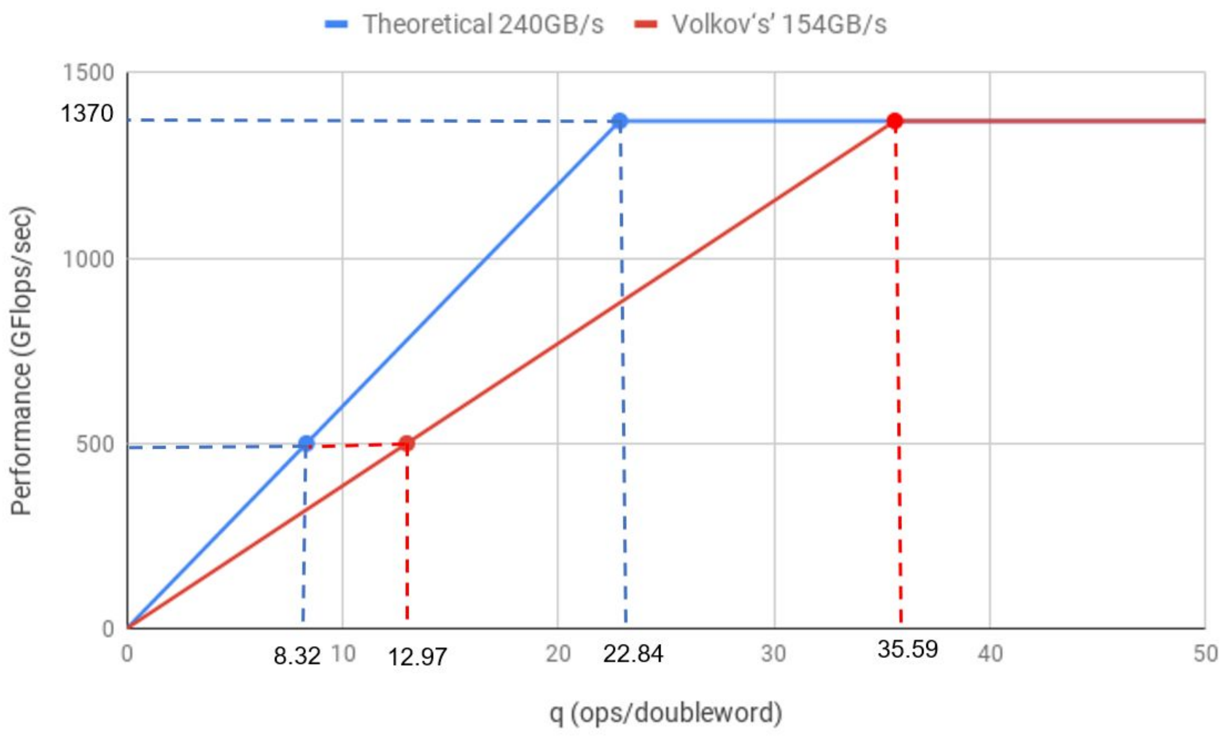

Figure 6. Roofline model of theoretical max bandwidth and Volkov's measurement. The black point indicates the performance of our implementation on input size = 1024 case.

Roofline model is the graph between the peak device performance vs the arithmetic intensity. The GPU used in this work has 13 SMX, 64 DP cores and each DP core can do 1FMA per cycle. Its clock speed is 823.5MHz, so the peak performance is:

$823.5MHz \times (2 \times 64 \times 13) = 1370.304GFlops/sec$

And with the theoretical max bandwidth 240GB/s, we have the turn point is:

$Q_{theory} = 1370.304GFlops/(240GB/4B) = 22.84ops/doubleword$

The roofline of theoretical max bandwidth is shown in Figure 6. Suppose we are using the max bandwidth when running, we have $q = 499.294/1370.304 \times 22.84 = 8.32ops/doubleword$ for our implementation at input size is 1024.

As to Volkov's measurement, the real bandwidth is 154GB/s and we plot another roofline model with this measurement following the same track. In this case, we have the turn point is:

$Q_{Volkov} = 1370.304GFlops/(154GB/4B) = 35.59ops/doubleword$

And $q = 499.294/1370.304 \times 35.59 = 12.97ops/doubleword$

As we can see from Figure 6, when BW decreases, the value of q increases, which means the program will reach the performance peak more slowly.

6. Future Work

Although our implementation has greatly increase the performance, there are some other work can be done to further improve it.

1) During working on this task, we found that if we make the warp size larger and let each thread deal with more outputs, we can improve the performance of high dimension (more than 512) matrix to over 750 GFlops/sec, but at the same time the performance of low dimension matrix decreases a lot. We think it's because for small matrix, a large warp and large thread reduces thread-level parallelism, cause the program to make very little use of the computation resource and increases thread blocking time. A possible solution is to dynamically adjust thread block size for different input.
2) Thread granularity is another method can be tried. The basic idea is to make the thread compute values cross blocks. It increases ILP and may make further improvement. But also the trade-off could be high which means no guarantee of improvement for this method.

7. Reference

[1] Vasily Volkov. Better Performance at Lower Occupancy. 2010. https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

[2] Vasily Volkov and James Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. 2008. http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf