# CSE260 Assignment 2 - Matrix Multiplication with CUDA

Author: Le Yang, Yongming He

## Section (1) - Development Flow

### Q1.a. How Program Works

Given matrix $A$ and $B$, the goal is to find their multiplication $C = A \times B$ where $A$, $B$, and $C$ are all $N$ by $N$ matrix. First, host memory is allocated for matrices $A$ and $B$. Then host memory is copied into device memory to do the calculation in the parallelized kernel.

In the kernel. We first allocate shared memory by defining two fixed size array *As* and *Bs* for each thread block to hold a small tile of matrix $A$ and matrix $B$. The dimension of the submatrix $As$ is $MC \times KC$ and the dimension of the submatrix $Bs$ is $KC \times NC$ where $MC$, $NC$, and $KC$ are predefined tile dimension parameters. Therefore, each thread block will calculate a $MC \times NC$ submatrix of matrix C by iterating through the $KC$ dimension of matrices A and B. After all, a $MC \times N$ row tile of $A$ is multiplied by a $N \times NC$ column tile of $B$, resulting in the $MC \times NC$ submatrix of $C$.

Each thread in a thread block will load $TM \times TN$ elements from matrix $A$ and $B$ to $As$ and $Bs$ respectively where $TM$ is calculated by $\frac{BLOCKTILE\_M}{BLOCKDIM\_Y}$ and $TN$ is calculated by $\frac{BLOCKTILE\_N}{BLOCKDIM\_X}$. With one thread being responsible for more than one loading, one thread block is able to fully load $As$ and $Bs$ into shared memory which can be accessed by all threads in the same block.

While iterating as $MC \times KC$ and $KC \times NC$ subtile through matrix $A$ and matrix $B$ respectively, each thread in a block will uniquely load $TM$ elements from the shared submatrix $As$ into the register matrix $sAs$ and uniquely load $TN$ elements from the shared submatrix $Bs$ into the register matrix $sBs$ which means elements loaded by one thread will not be loaded by another thread. Then matrix outer product is performed to calculate the partial sum of each element in the final submatrix $Cs$.

After finish the iteration, all elements in $Cs$ is correctly calculated and the resultes will be stored back into matrix $C$. After all thread blocks finish their job, matrix $C$ will be returned by the kernel.

To handle edge cases, when we load elements from matrix $A$ and matrix $B$, we will check if the index is out of range ($> N$). If the index is invalid, we will load 0 into the shared memory which will have no effect on the final result even it's used for computation. Also, when store the final results back into matrix $C$, we will check if index is valid. Besides, if $N$ is not divisible by tile size, one more block will be added to that dimension.

The progress for the kernel is shown in the following presudocode:

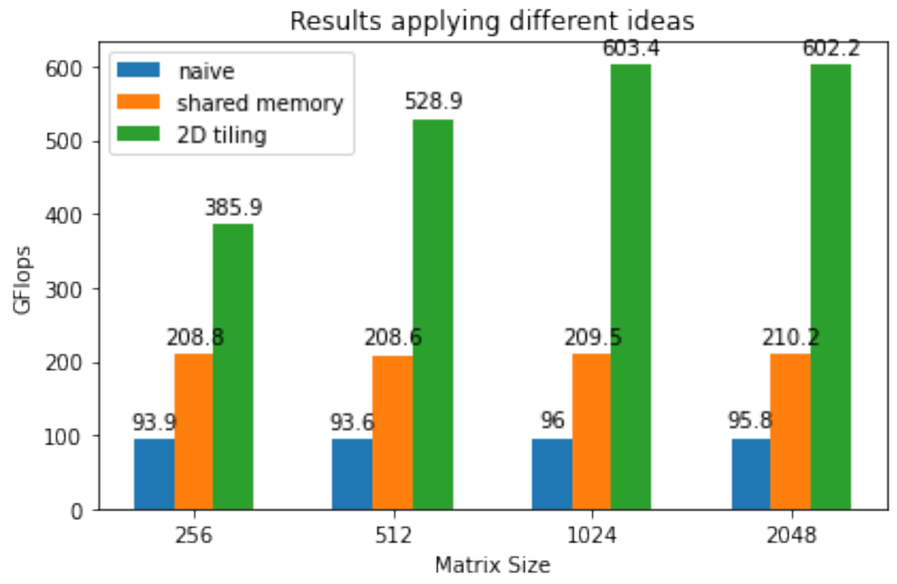
```
Algorithm 1: Matrix Multiplication Algorithm
  Input: N, C, A, B
  Output: C
1 Initilize shared memory As[MC][KC], Bs[KC][NC] register sAs[TM], sBs[TN], Cs[TM][TN]=0
  for $kk = 0 \cdots \frac{N}{KC} + (N\%KC! = 0)$ do
2     load A and B into shared memory As and Bs based on kk
3     _syncthreads()
4     for $k = 0 \cdots KC$ do
5         load corresponding vector into sAs and sBs based on K
6     using sAs and sBs to calculate the outer product to Cs
7     _syncthreads()
8 Store Cs into the corresponding postion of C
```

## Q1.b. Development Process

- As a baseline, after we set up the environment and had a basic understanding of the naive code, we run the given code and achieve about 90 Gflops.
- According to lecture 10 slides, we implemented the shared memory version of matrix multiplication. In this way, instead of loading all the data each thread needs by itself, each thread loads one element into shared memory. Therefore, the loading process is more paralellized and data are reused more often. We are able to achieve around 210 Gflops with shared memory.
- Based on lecture 11 slides, we implemented 2D tiling. We are able to achieve the required computation speed with this update.

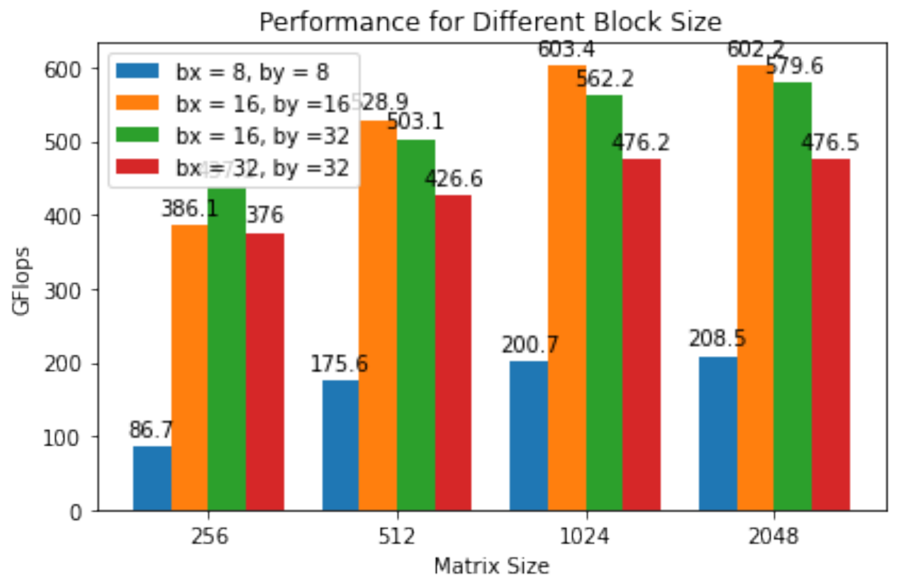
## Q1.c. Ideas

- we add `#pragma unroll` before each loop to perform loop unrolling. This can reduces the number of NOPs generated and also provides the compiler with a greater opportunity to generate parallel instructions.
- Shared memory worked well because it decreases the number of loading data and it is intrinsically faster memory than global meomry. Besides, the pseudocode on the slides has already embedded the idea of memory coalescing.
- 2D tiling also worked well. Instead of loading single element into shared memory, each thread loads multiple elements and computes multiple elements of matrix C. Thus, we need less amount of threads to do the same amount of computations as before. Decreased number of threads decreases thread syncronize time. With each thread doing more work, we increase instruction level parallelism. Moreover, each thread loads needed data from shared memory into registers, making the reuse of data more efficient.
- We tried to resolve bank conflicts when threads try to load the same part of matrix A and matrix B into their own shared memory. This happens when threads have the same thread_id but are from different thread blocks. However, this implementation did not influence the computation speed. It seems that the compiler handles this well for us.



## Section (2) - Result

### Q2.a. Performance of Different Problem Sizes



To generate the above graph, we fixed the tile size. How many outputs each thread computes depends on the thread block sizes. It is designed such that tile sizes are multiples of the thread block sizes. Bigger thread block sizes indicate less outputs from each thread and vice versa.

The only limitation on thread block sizes is that they must be divisible by the tile sizes. We did not handle edges cases where thread block sizes are not divisible by tile sizes. This causes error because we use tile sizes divided by block sizes to calculate the number of outputs each thread need to compute which is also used when declare fixed size holder for outputs and in for loop.

## Q2.b. Choice of Optimal Thread Block Sizes

To better coorporate the idea of thread warp. We intentionally design the number of threads in each block to be multiples of 32. We extensively do experiments on square block size. The bottleneck for such thread block size is 32 since the maximum number of threads in a block is 1024. As shown below, 16x16 works best for most of the matrix sizes (512, 1024, 2048). However, for matrices of size 256, we found 16x32 yields better peak performance.

If the block size is too small, occupancy is low and we lose lots of performance from thread level parallelism. If the block size is too big, although the occupancy is high and the kernel is highly parallelized in thread level. We spend a lot of time waiting for threads to synchronize. Smaller block size means more outputs from single thread which indicates higher instruction level parallelism. While there is a trade-off between instruction level parallelism and thread level parallelism, 16x16 tends to be a optimal choice for both kinds of parallelism.

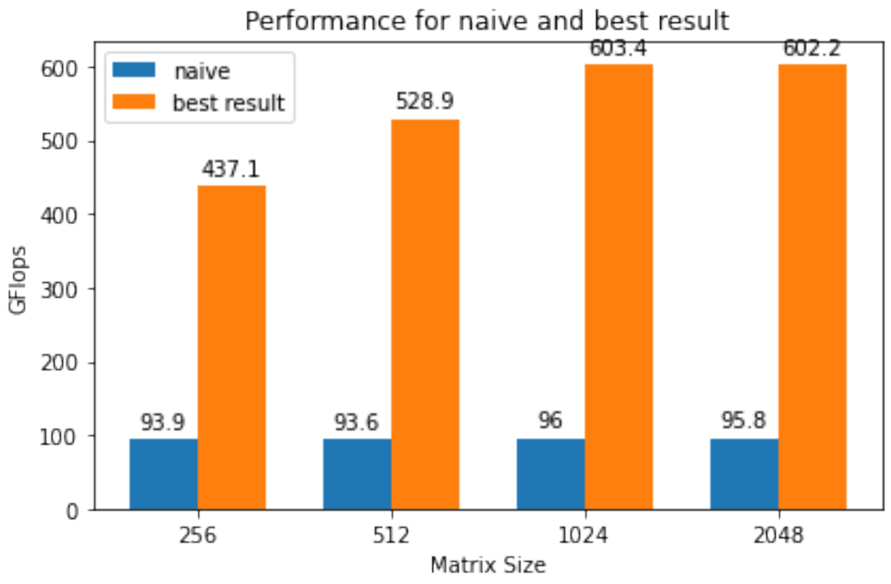
As to why the optimal block size for matrices with size 256 is 16x32 is because square block size does not take full advantage of register memory. Increase 1 dimension from 16 to 32, we get more instruction level parallelism. Besides, since 256 is small enough, the grid dimension is small which means we only have a few blocks. Therefore, synchronization duration does not increase too much. On the opposite, it is because for bigger matrices, we have more blocks, the increasement of syncronization time beats the increasement of performance resulted from increasement of block size. Thus, 16x16 is still the optimal block size for bigger matrices' computation.

## Q2.c. Peak GF Performance

| N | Peak GF | Thread Block Size |
|---|---|---|
| 256 | 437.1 | bx=16, by=32 |
| 512 | 528.9 | bx=16, by=16 |
| 1024 | 603.4 | bx=16, by=16 |
| 2048 | 602.2 | bx=16, by=16 |

## Section (3)

## Q3.a. Compare Best Result with the Naive Implementation

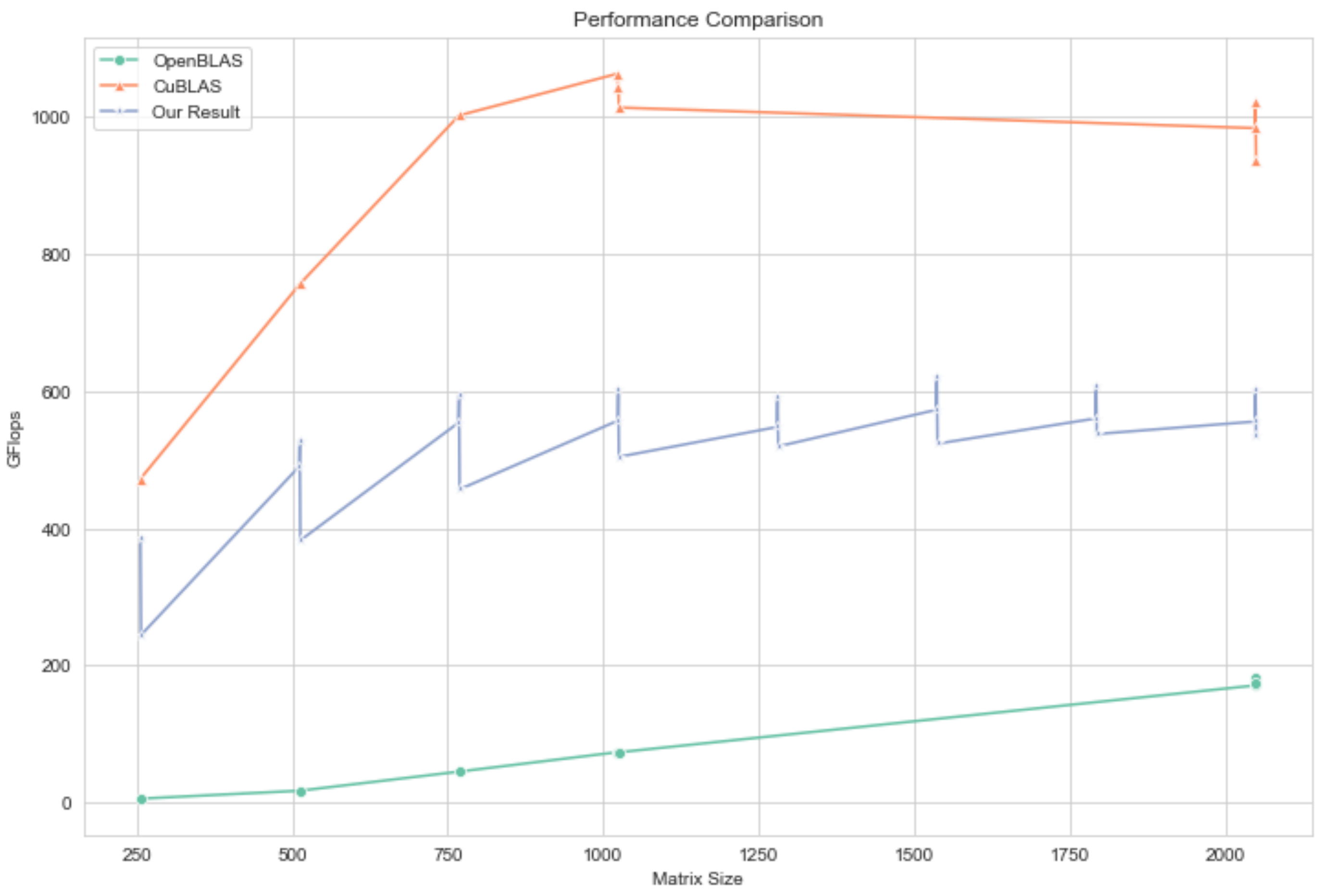


| **Speedup ratio** | **256** | **512** | **1024** | **2048** |
|---|---|---|---|---|
| Best result VS naive | 4.65 | 5.65 | 6.28 | 6.28 |

For $N$ = 256, 512, 1024 and 2048, we can see that our method can get a speedup over the naive results at about 6.28, which is a good improvement.

## Section (4) - Analysis

### Q4.a. Plot Performance



## Q4.b. Analysis of the Plot Compared to BLAS

We can see from the above plot that the shape of our curve is similar to the BLAS values in a way that as the matrix size become bigger but still smaller than $1024 \times 1024$, the performance is improving. Both performance stablize for matrix sizes that are bigger than 1024.

More importantly, BLAS and our implementation share the same irregularity around matrix sizes like 1024 and 2048 which are multiples of 2 (also multiples of our tile size). This is because they both use the idea of tiling. When the matrix size is not a multiple for tile size, there are computation wasted from zero padding.

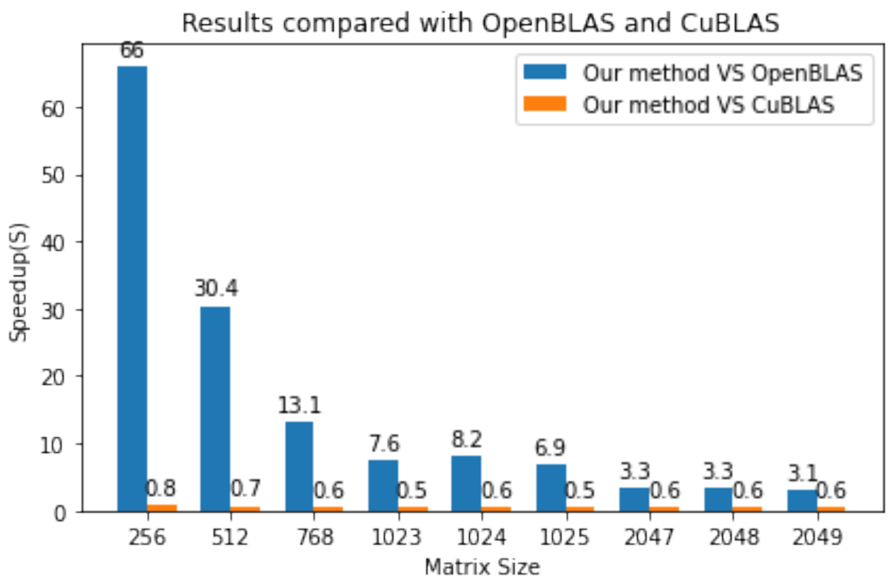
## Q4.c. Analysis of the Dips, Peaks or Irregularities

From the result, we can see many unusual dips and irregularities. This because when selecting additional $N$ for the experiment, we choose many Ns that are not divisible by tile size around those which are divisible by tile size. As the description in Q1.a., if N is not divisible by tile size, one more block will be added to that dimension and additional 0 will be load into shared memory. That is to say, we use padding-0 method to deal with the edge cases, which may waste the computing resources and hurt the performance. That is why we see a quick draw down of performance after matrix sizes that are multiples of tile size. Then, as $N$ increases, the percentage of zero padding in a tile decreases which causes the performance to increase. Finally, when $N$ is approaching next special value(multiple of tile sizes), we see an unusualy boost of performance and achieve the peak performance at such value. When $N$ is big enough, the performance stablize and repeat this process between adjacent special matrix sizes.

## Q4.d.

| N | BLAS (GFlops) | CuBLAS | Your Result (GFlops) |
|---|---|---|---|
| 256 | 5.84 | 472.5 | 385.9 |
| 512 | 17.4 | 756.5 | 528.9 |
| 768 | 45.3 | 1002.7 | 593.8 |
| 1023 | 73.7 | 1063.7 | 557.3 |
| 1024 | 73.6 | 1045.4 | 603.4 |
| 1025 | 73.5 | 1014.2 | 504.6 |
| 2047 | 171 | 984.2 | 556.1 |
| 2048 | 182 | 1021.6 | 602.2 |
| 2049 | 175 | 937.8 | 536.7 |
| 257 | | | 244.6 |
| 511 | | | 490.3 |
| 513 | | | 382.7 |
| 767 | | | 554.3 |
| 769 | | | 457.2 |
| 1279 | | | 548.1 |
| 1280 | | | 591.0 |
| 1281 | | | 519.8 |
| 1535 | | | 573.2 |
| 1536 | | | 621.4 |
| 1537 | | | 523.8 |
| 1791 | | | 560.6 |
| 1792 | | | 606.8 |
| 1793 | | | 537.9 |

## Q4.e Plot the above results comparing your implementation with OpenBLAS and cuBLAS (from table)

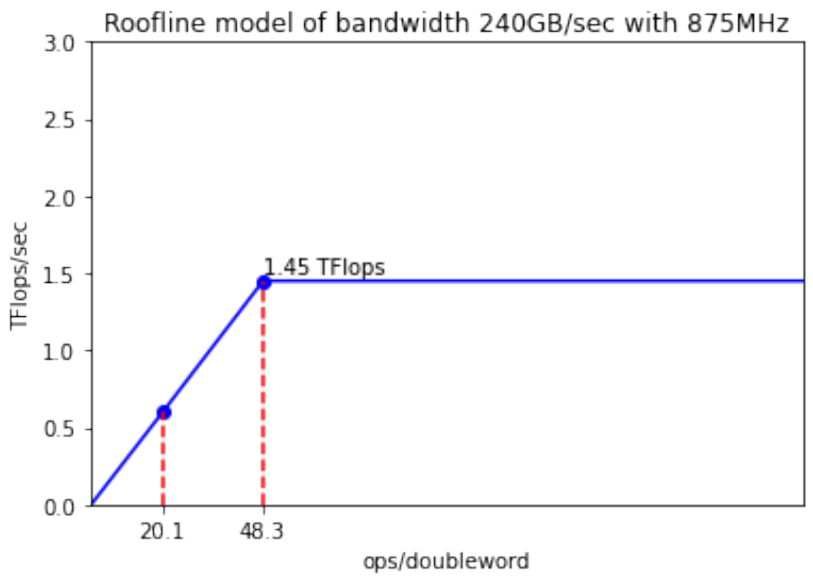


To compare our results with OpenBLAS and cuBLAS, we calculate the speedup ratio(S) of our results over OpenBLAS and cuBLAS and plot it. We can see that as the matrix size become larger, the speedup ratio of our results over OpenBLAS become smaller and finally stable at about 3.3, while the speedup ratio of our results over cuBLAS is always stable at about 0.6.

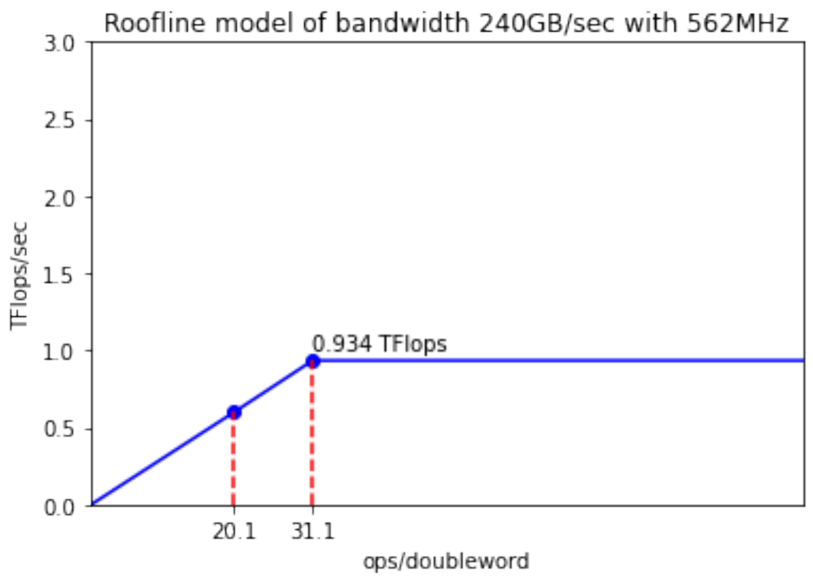
## Section (5)

### Q5.a Roofline Model

For the hardware we are using, we ser the boost frequency to 875 MHz. Therefore the peak performance will be $13\ SM \times 64\ DP \times 2\ ops \times 875\ MHz = 1.45TF$. When bandwidth is 240 GB/sec = 30 G doubles/sec, then q = 1.45 TFlops / 30G doubles/sec = 48.3 ops/doubleword. For our method, we can achieve 603.4 GFlops, so q = 603.4 GFlops / 30G doubles/sec = 20.1 ops/doubleword. The roofline model with the boost frequency is shown below.



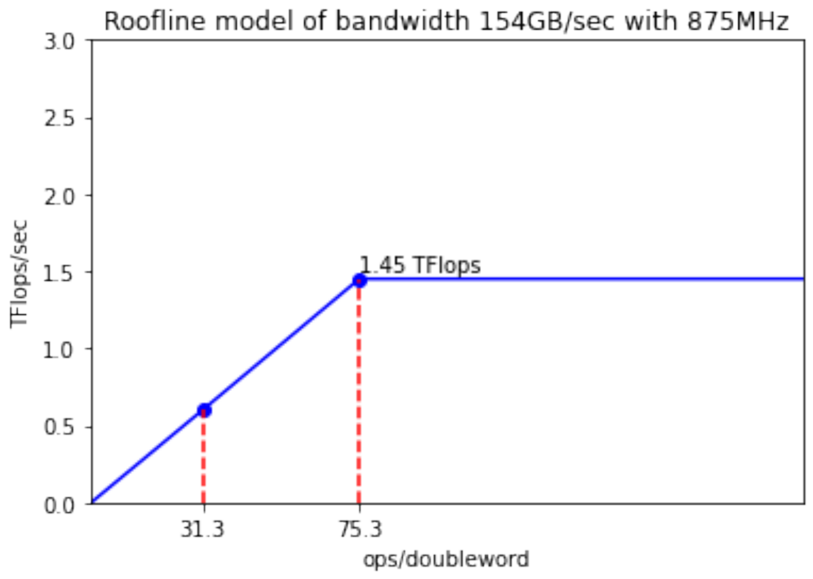
However, we need to consider DVFS effect, using nvprof tool, we can get our frequency at 562 MHz. Therefore the peak performance will be $13\ SM \times 64\ DP \times 2\ ops \times 562\ MHz = 934GF$. When bandwidth is 240 GB/sec = 30 G doubles/sec, then q = 934 GFlops / 30G doubles/sec = 31.1 ops/doubleword. The roofline model considering DVFS is shown below.



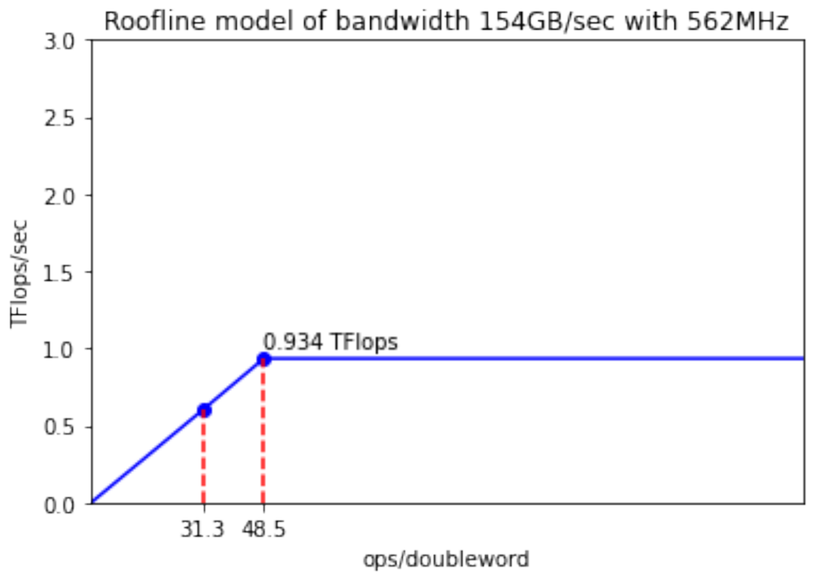
## Q5.b Estimate the value of q in ops/doubleword.

When the actural bandwidth is 154 GB/sec = 19.25G doubles/sec.
For our method, when n=1024, the result is 603.4 GFlops, therefore q = 603.4 GFlops / 19.25G doubles/sec = 31.3 ops/doubleword.

Using the boost frequency, then q = 1.45 TFlops / 19.25G doubles/sec = 75.3 ops/doubleword. Then the roofline model with boost frequency at 875MHz is shown below.



Considering the DVFS effect, then q = 934 GFlops / 19.25G doubles/sec = 48.5 ops/doubleword. Then the roofline model with frequency at 562MHz considering DVFS effect is shown below.



To achieve the same peak performance, we can see that Q increases as the bandwith decreases.

## Section (6) - Potential Future work

If we had more time, we want to handle edge cases where block sizes are not divisible by tile sizes. Therefor, we could have more flexible block sizes. Then we can further utilize register memory and try to use all of them.

## Section (7) - T4 Extra credits

## Section (8) - References

[1] Andrew Kerr, Duane Merrill, Julien Demouth and John Tran , CUTLASS: Fast Linear Algebra in Cuda C++, December 2017
[2] Volkov, Demmel, Benchmarking GPUs to Tune Dense Linear Algebra, SC2008
[3] Volkov, Better Performance at lower Occupancy, GTC2010
[4] https://docs.nvidia.com/cuda/archive/9.1/cuda-c-programming-guide/index.html