CSE 260: Project 2
Winter 2018

Ashin George, Debosmit Majumder

Contents

1 Introduction

In recent times, GPUs have been omnipresent whenever there has been a need of performing matrix multiplication operations. GPUs with relatively smaller architectures and smaller pipelines can be replicated multiple times on a chip. Matrix multiplication is the integral part of Machine Learning and Artificial Intelligence, including but not limited to data mining and analytics. In a world where huge amounts of data are being generated that need to be processed, GPU significantly speed up the ever-increasing computational complexity of matrix multiplication[2]

The project involves performing a CUDA implementation of matrix multiplication kernel and improving its performance through various techniques. In the report we first present and analyze the results of various optimizations that we have applied. The next section gives an overall performance improvement over the naive matrix multiplication in CUDA.

The program has been written in C with support for CUDA extensions. The kernel has been executed on K80 GPU cards with 2 GK210 GPUs with a compute capability of 3.7. [1]

2 Analysis

2.1 Improvement over the Naive Kernel Implementation

The naive kernel implementation for matrix multiplication performs load operations for every value of A and B N times for a matrix of size N. This process is quite slow due to the increased latency suffered from the multiple load operations from the global memory. Here we exploit the shared memory concept of GPU.

2.2 Tiled Algorithm using Shared Memory

We divide the matrices into tiles. Each thread cooperates to load a tile of A and B into the shared memory. Each tile in the C matrix corresponds to a thread block. In this case, each thread loads one A and B. Also, each thread calculates the partial product of C and keeps on accumulating the value of C. Finally, it stores the values in memory.

The reduced global memory access improved the latency of the operation. Shared memory on GPU can be used by any thread within a thread block. This behaves kind of like caching in case of conventional processors where frequently accessed data is stored in memory closer to the device. In this case, As and Bs correspond to the data present in the shared memory. It was observed that the speedup obtained was 2.79 over the naive implementation.

The following figure compares matrix multiplication performance with and without tiling algorithm.

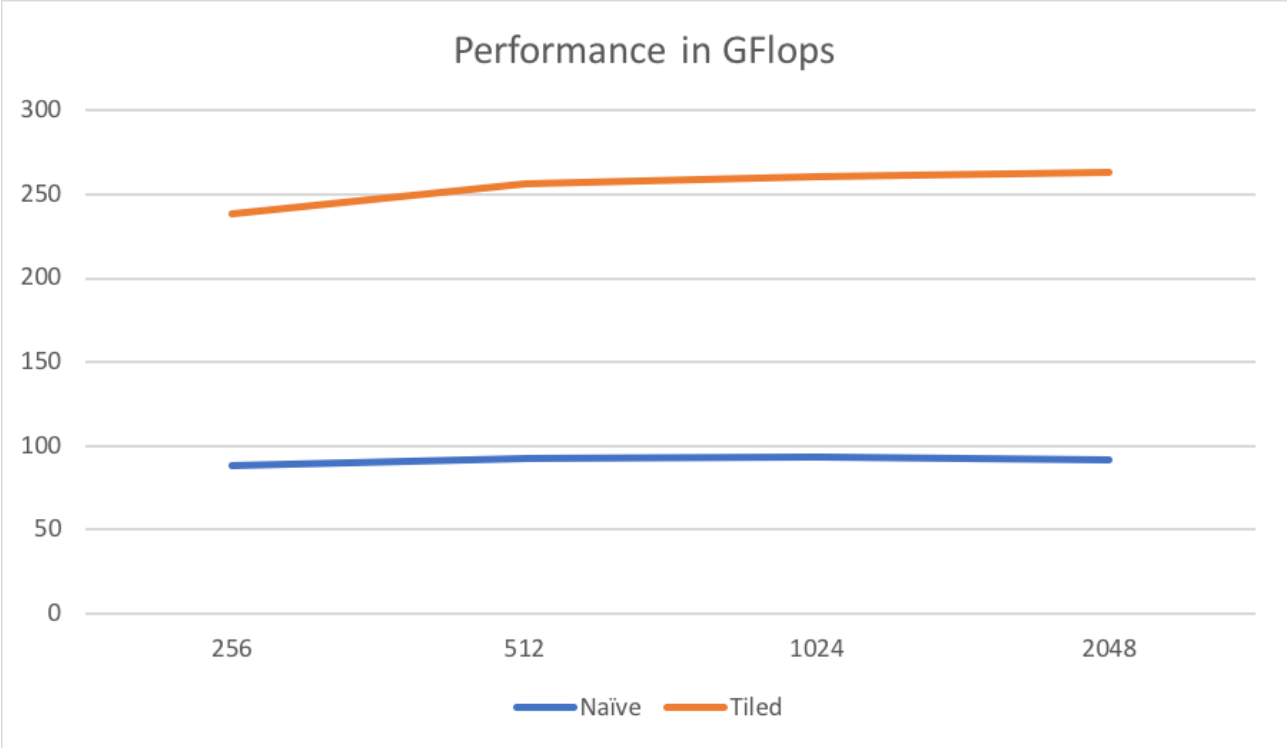

Figure 1: Tiled vs Naive Performance

2.3 Unrolling

Unrolling tries to reduce the occupancy by making each thread do more work. Sometimes more thread-level parallelism may not be advantageous due to phenomenon like thread divergence. Hence, unrolling is performed to increase instruction-level parallelism.

We manually unrolled the loops 8 times. Thus, each thread now performs 8 load operations and 8 multiplication-add operations. This we decrease our thread block size 4 times. This leads to lower occupancy. Hence now we obtain 8 outputs per thread.

We tried multiple unroll factors and observed that as the unroll factors increased the performance improved. This is because with increasing unroll factors, each thread is doing more work and we require lesser number of threads inside a block and thus lesser occupancy is attained. The speedups obtained for each case over when unrolling was not performed was observed to be 1.26 (Unroll Factor = 2), 1.45 (Unroll Factor = 4) and 1.47 (Unroll Factor = 8).

Memory Coalescing is a property warps benefit from, given contiguous aligned accesses of 128/256bytes. Warps schedule 32 threads to run simultaneously on a streaming multiprocessor. In our tiled code, each thread accesses one 8-byte element of each A and B in order. So, all the 32 threads will access 32*8 bytes of each A and B on the same row in contiguous order. So, most of the thread accesses will be aligned besides the edge cases in case the size of N is not a multiple of the block size. It has to be made sure that loads of A and B happen in order so that memory coalescing takes place.

The following histograms plot the performance of the matrix multiplication kernel for various different unrolling factors (0,2,4,8).

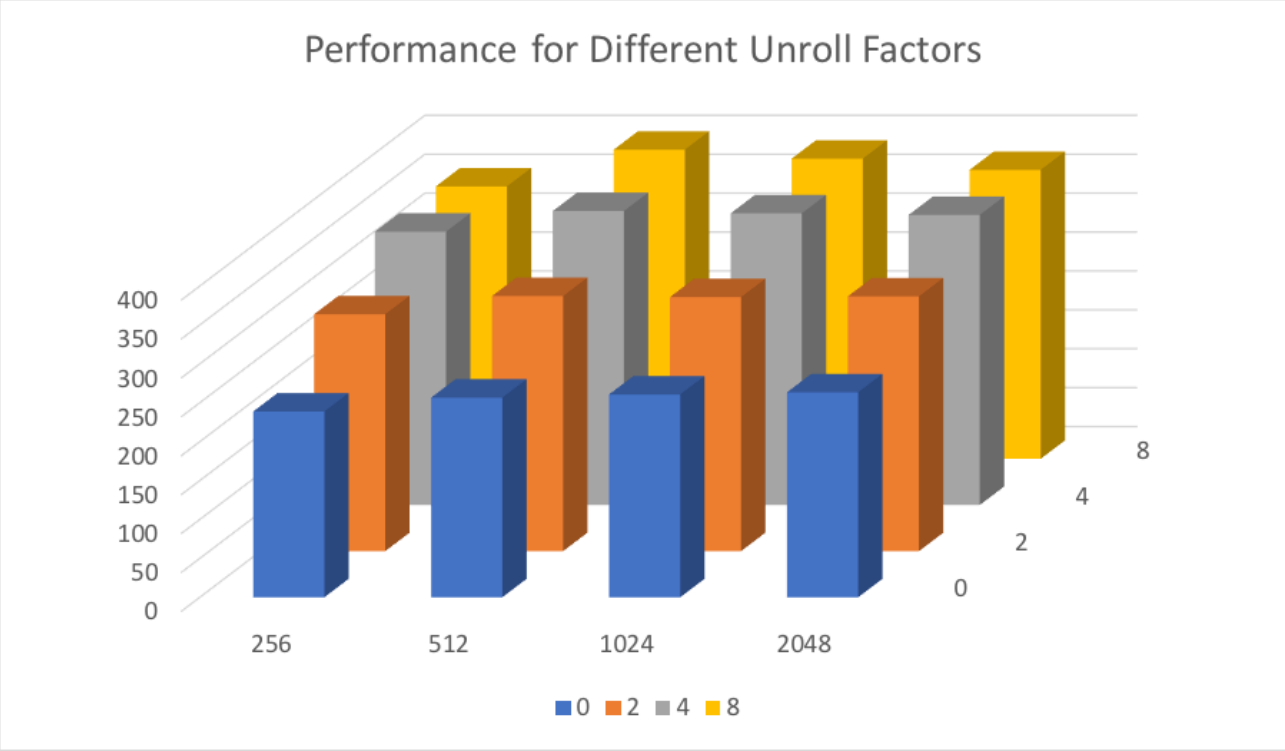

Figure 2: Performance for different Unroll Factors

2.4 LDG

The LDG instruction is a global memory load that uses the read-only texture memory cache. It has the advantage that it does not require the explicit use of textures. Explicit uses of textures causes a certain amount of code clutter and overhead (e.g. for API calls to bind textures). It is optimized for data locality (spatial locality) hence data reads by a warp which are physically adjacent benefit a lot from texture cache because it reduces memory traffic.[]

The following plot shows performance after using LDG instead of the conventional load.

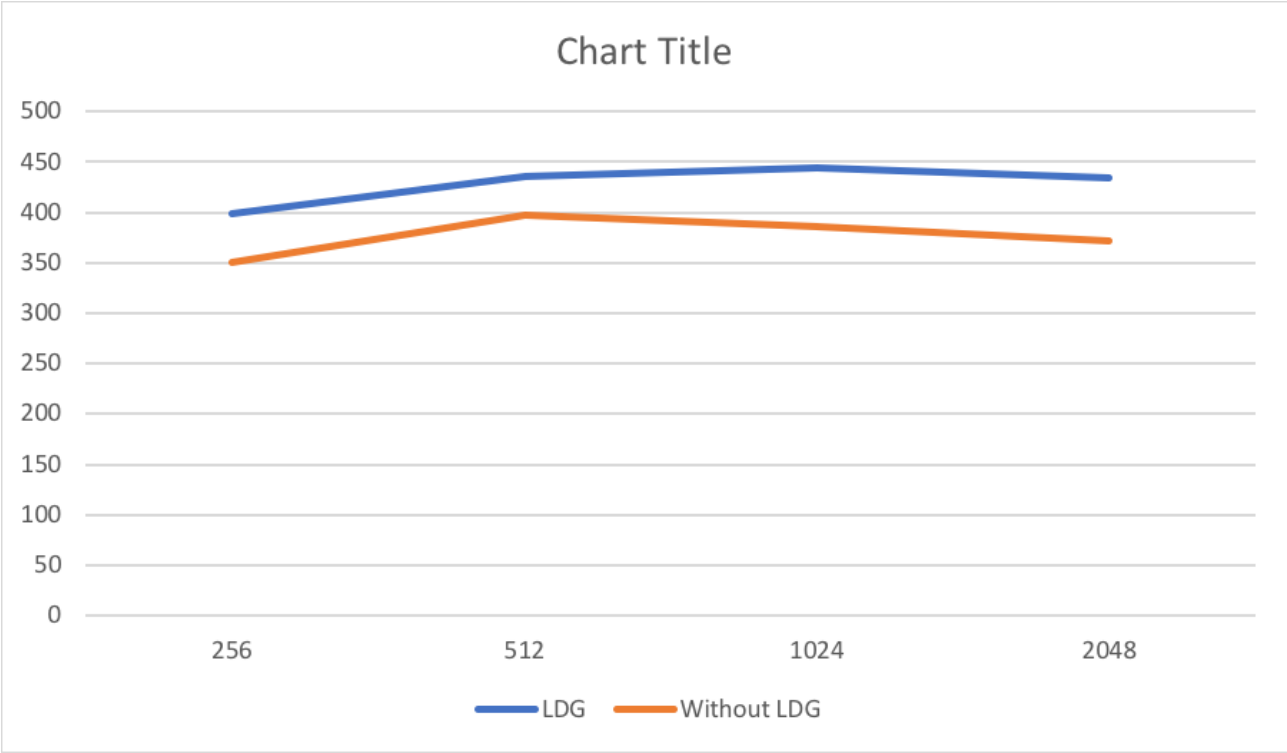

Figure 3: LDG Performance

3 Results

In this section we present the results of our optimizations.

3.1 Pseudocode

The pseudo code provides a high level overview of the optimizations that we implemented.

Listing 1: Psuedo Code

```
//Implementation of CUDA matrix multiplication kernel
__global__ void matMUL(N, A, B, C){
        __shared__ double As,Bs          //Declaration of Shared Memory

    //Declarations of Thread and Block Indices

    //Condition to check for divisibility of N with block size
    //Limiting the values of I and J to within N

    //kk goes from 0 to N/TW or, N/TW + int(bool(N%TW))
    for kk in (0,N/TW or N/TW + int(bool(TW)),1){
    //Loop is unrolled 8 times when N is multiple of TW
        //Copying the contents of the matrices
          from host memory to device memory i.e.
          from A,B to As,Bs
    **Each thread performs 8 loads of A and B

          for k in (0, min(TW,N-kk*TW), 1) {
                // performing addition and multiplication
                of As and Bs and storing it into
                register C
    **Each thread performs 8 addition and multiplication operations

        }
   //Stores are performed  on 8 locations of C after all
   the matrix calculations are done
   }
}
```

3.2 Performance

The following figure plots the performance for at least 3 different block sizes.

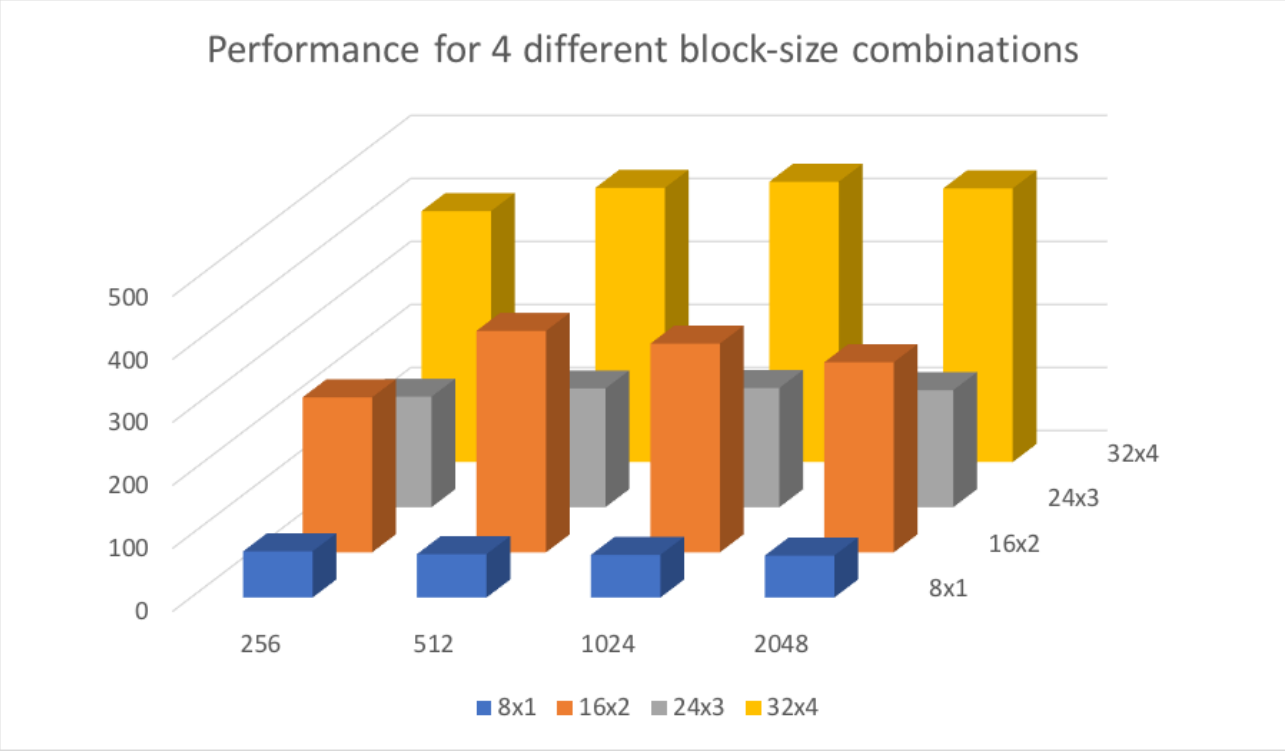

Figure 4: Performance for 4 different Block Sizes

The optimizations work for all block sizes which are smaller than 32. Block dimensions greater than 32 will result in *Cuda error:Invalid configuration argument* when the number of threads per block goes beyond 1024 as limited by hardware. Very large block dimensions(eg:16x64) results in ptxas error caused by using too much shared memory. The code will work for all combinations of thread block dimensions up to 32.
The optimal block sizes for our code are block configurations which have 8:1 ratio between the x-dimension and y-dimension. These dimensions allow for effective utilization of the loop unrolling. Of these configuration 32x4 gives the highest performance. This configurations allow an entire row of this block to be executed in a single warp. We believe that combined with loop unrolling 32x4 thread block gives the best tradeoff between GPU utilization and ILP by making efficient use of the available registers. This performance gain is further maximized when the matrix dimension is a multiple of 32 as seen in 7

3.3 Performance vs Naive

The following plot shows the performance obtained from the optimal block size that is 32X4 vs the naive implementation.

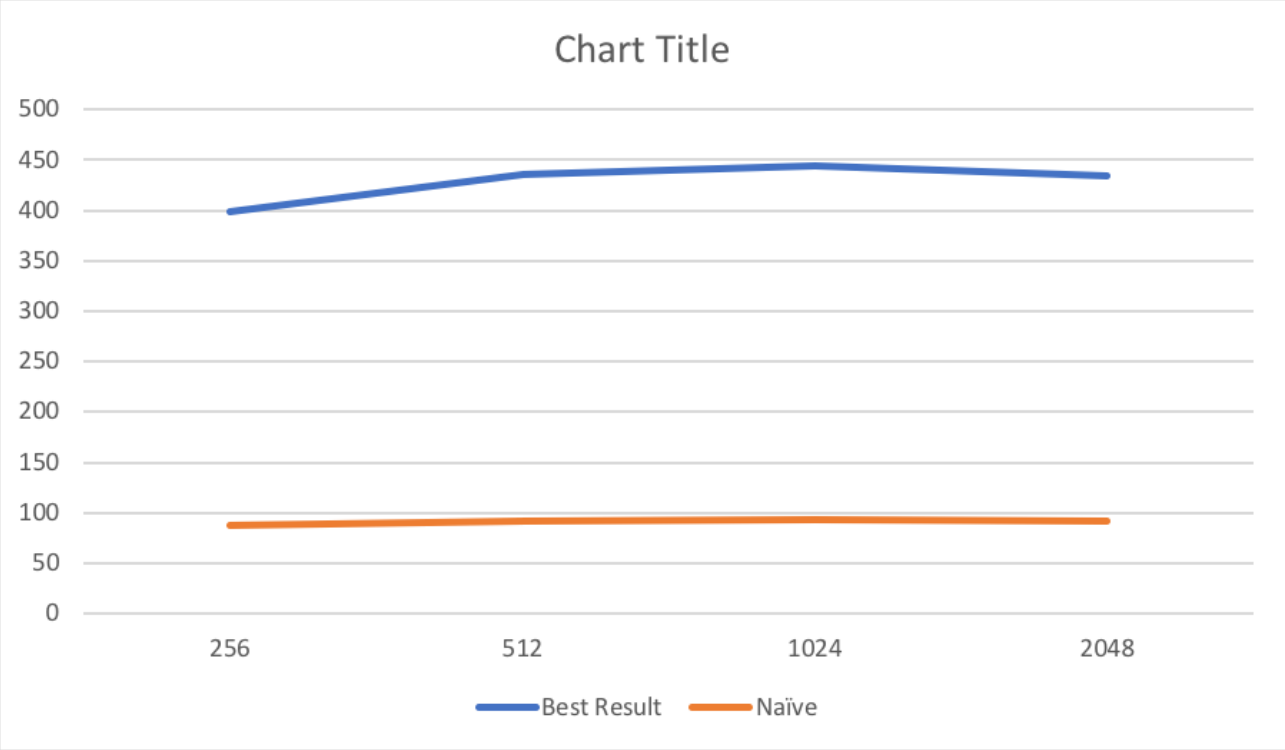

Figure 5: Performance for Best Result over Naive

3.4 Performance vs BLAS

The following plot shows the performance of our implementation over the BLAS implementation.

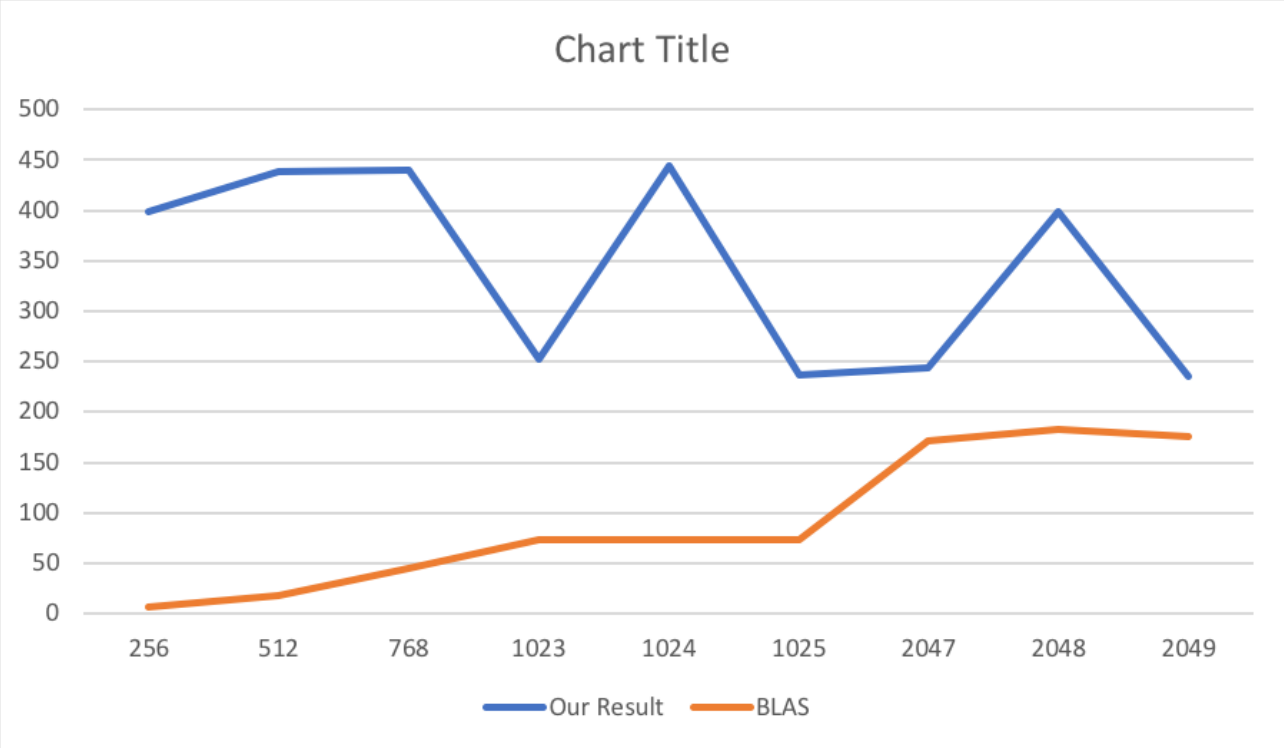

Figure 6: Performance for our implementation over BLAS

3.5 Performance of our implementation

The following plot shows the performance of our implementation for 24 points.

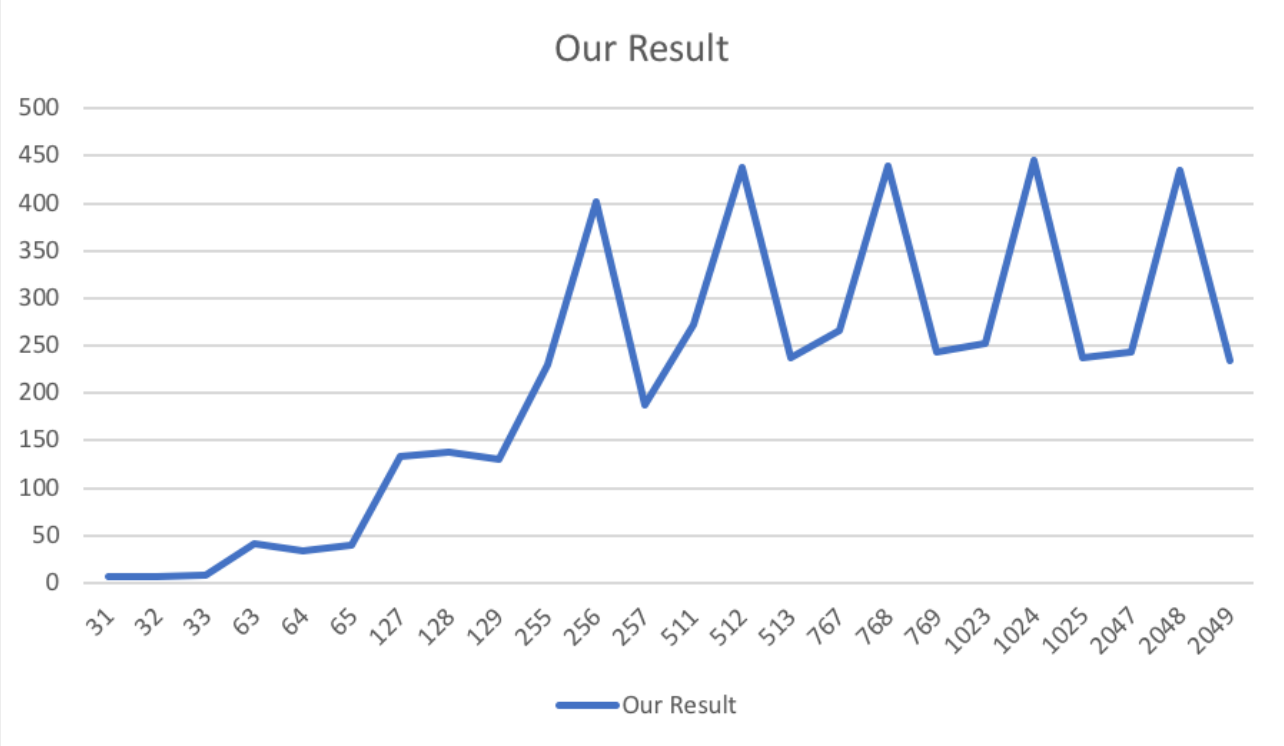

Figure 7: Performance for our implementation for 24 points

It can observed from the plot that our implementation shows some dips for those values of matrix size (N) that are non-multiple of the tile size. This is due to the fact that we are not unrolling for non-multiples of tile size. We are not performing unrolling because we are not gaining enough from unrolling for non-multiples. This may be due to thread divergence issue. Also, we ran NV profiler and found that a lot of control flow instructions are being executed.

BLAS is relatively stable because its performance does not depend on the matrix size as much.

3.6 Speedup

The following plot shows the speedup of our implementation over BLAS and Naive matrix multiplication kernels.

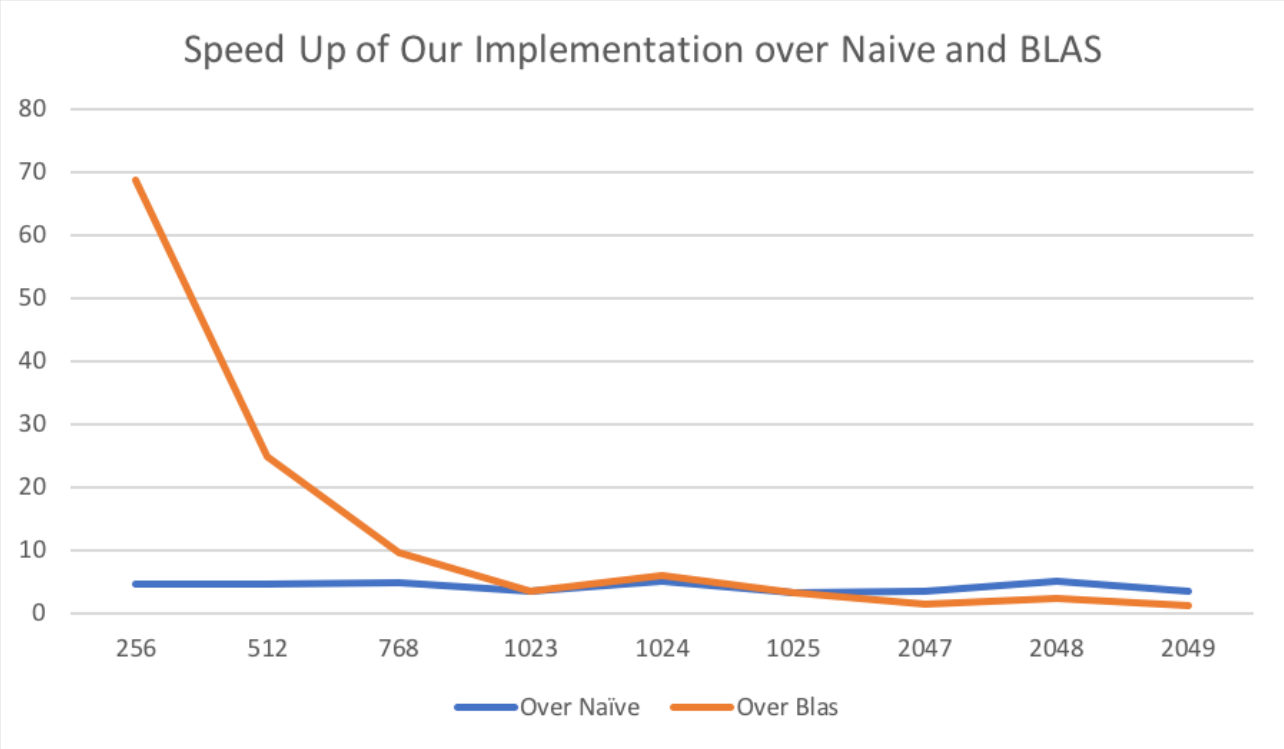

Figure 8: Speedup of our implementation over BLAS and Naive

3.7 Roofline Model

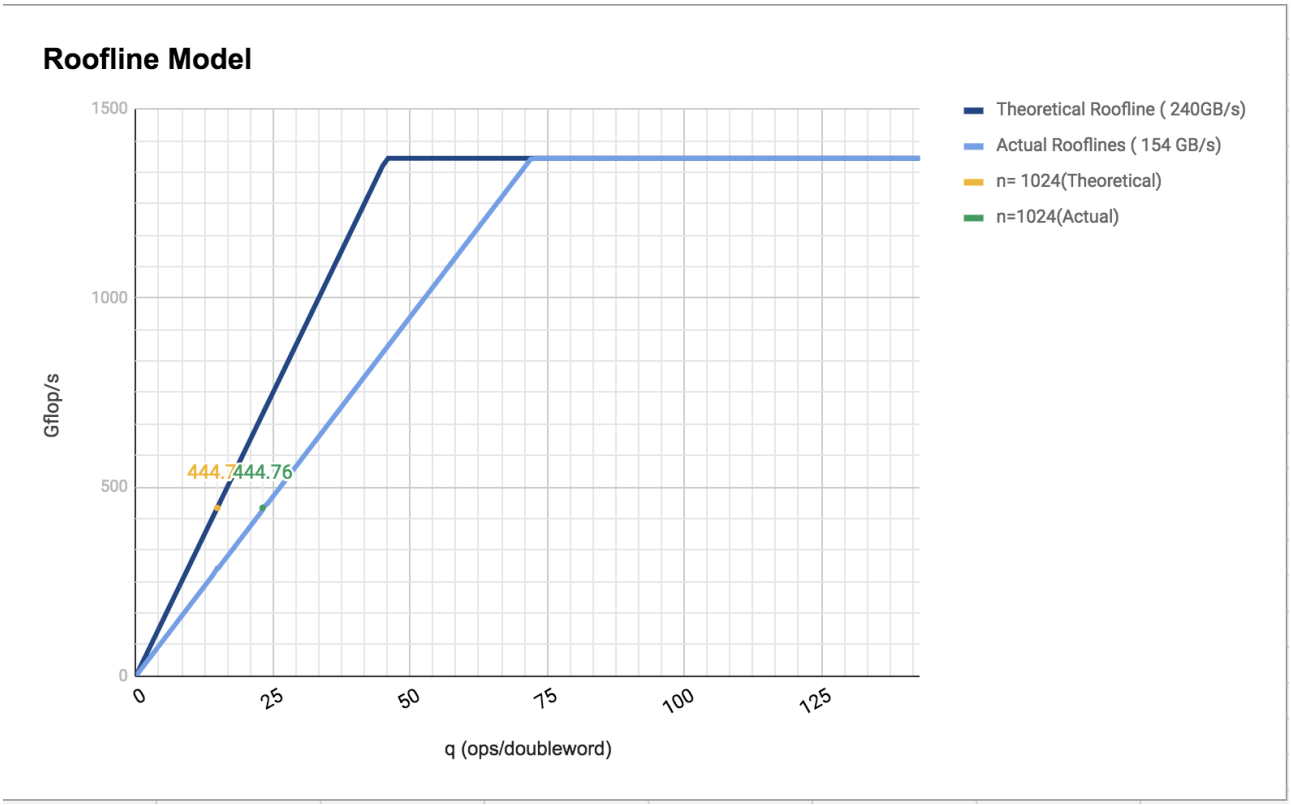

Figure 9: Performance for Naive implementation, our implementation and BLAS implementation

Roofline Model is the graph between the peak device performance vs the arithmetic intensity. The slope represents the memory bandwidth. The theoretical BW is 240 GB/s while the measured BW is 154 GB/sec. We plot the model based on matrix size of 1024. As expected, it is observed that as memory BW goes up, we require lesser Q value to attain the peak performance. The performance value obtained from the graph is 444.76 GFlops as measured on Sorken.

4 Potential Future Work

Prefetching is expected to improve the performance of the matrix multiplication kernel. This is because as we go over the multiple iterations of the loop, the data values can be pre-fetched so that they are already present in the L2 cache. This will result in reduced memory latency.

Unrolling the loop in the X-dimension as well would have improved performance. This is due to further reduction in occupancy and each thread doing more work. Thus, we will require lesser number of threads per thread block.

We could also incorporate unrolling for matrix size values (N) that are non-multiples of the tile size. This will again ensure significant performance improvement due to increase of work per thread[3]

References

[1] *Debugging Your CUDA Applications With CUDA-GDB*. Tech. rep. URL: http://developer.download.nvidia.com/GTC/PDF/1062_Satoor.pdf.

[2] K. Fatahalian, J. Sugerman, and P. Hanrahan. "Understanding the Efficiency of GPU Algorithms for Matrix-matrix Multiplication". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '04. Grenoble, France: ACM, 2004, pp. 133–137. ISBN: 3-905673-15-0. DOI: 10.1145/1058129.1058148. URL: http://doi.acm.org/10.1145/1058129.1058148.

[3] Vasily Volkov and James W. Demmel. "Benchmarking GPUs to Tune Dense Linear Algebra". In: *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–11.