Double-Precision Matrix Multiply on CUDA

Parallel Computation (CSE 260), Assignment II

Mithun Chakaravarrti Dharmaraj
A53235626, mdharmar@eng.ucsd.edu

Sai Kishan Pampana
A53253039, spampana@eng.ucsd.edu

November 6, 2014

Motivation

Today, processors are much more faster than memory. As we know, "A system can only perform as fast as the slowest link". By making use of DRAMs for running programs we probably might not exploit the entire power our processors can deliver. To optimize the problem of matrix multiplication, it's essential to understand the memory hierarchy of the system we're working on.

Since we know that memory operations such as read & write are slow, we ultimately want to reduce it for every computation we make. The task at hand simply came down to increasing the number of floating point operations performed per load/store. This is technically called Computational Intensity (Q).

Assumptions

1. All matrices are square matrices of size N.
2. All matrices consist of double-precision floating point values.
3. Parameters (grid & block sizes) are chosen in such a way compute resources are utilized to the fullest.
4. A and B are the source matrices. C is the target matrix.

1. Environment

The double-precision matrix multiply was optimized for the following runtime environments. All benchmarks & performance analysis were carried out on host & GPU with the following configuration.

1.1 Host CPU configuration

- 2 octa-core Intel Xeon CPU E5-2640 v3 @ 2.60GHz
- 32 KB L1 instruction cache
- 32 KB L1 data cache
- 256 KB L2 cache
- 20480 KB L3 cache

1.2 GPU configuration

The computations we're done on a node with 2 NVIDIA Tesla K80 GPU cards. Tesla K80 GPU cards has the following configuration.

Feature	NVIDIA Tesla K80 card
GPUs	2 x Kepler GK210 GPUs
Max CUDA cores	2 x 2496
Max SMX Units	2 x 13
Onboard Memory	24 GB GDDR5 with ECC disabled
Memory Bandwidth	480 GB/s with ECC disabled
Core clock	562 MHz
Boost clock	875 MHz
Peak Single Precision (base clocks)	5.60 TFLOPS (both GPUs combined)
Peak Double Precision (base clocks)	1.87 TFLOPS (both GPUs combined)
Peak Single Precision (GPU Boost)	8.73 TFLOPS (both GPUs combined)
Peak Double Precision (GPU Boost)	2.91 TFLOPS (both GPUs combined)

Feature	Kepler GK210 GPU
Compute Capability	3.7
Threads per Warp	32
Max Warps per SMX	64
Max Threads per SMX	2048
Max Thread Blocks per SMX	16
32-bit Registers per SMX	128 K
Max Registers per Thread Block	64 K
Max Registers per Thread	255
Max Threads per Thread Block	1024
Shared Memory Configurations *(remainder is configured as L1 Cache)*	16KB + 112KB L1 Cache 32KB + 96 KB L1 Cache 48 KB + 80 KB L1 Cache
Max Shared Memory per Thread Block	48KB

1.3 Software

- nvcc - NVIDIA (R) Cuda compiler driver v 7.0
- GNU/Linux, Kernel version 2.6.32-573.22.1.el6.x86_64

Notations

- $i_{B,x}$ and $i_{B,y}$ are the block index in x or y dimension, i.e. blockIdx.x/y.
- $dim_{B,x}$ and $dim_{B,y}$ are the block dimension in x or y dimension, i.e. blockDim.x/y.

- $i_{T,x}$ and $i_{T,y}$ are the thread index in x or y dimension, i.e. threadIdx.x/y.
- $dim_{G,x}$ and $dim_{G,y}$ are the grid dimension in x or y dimension, i.e. gridDim.x/y.

2. Performance study on naïve algorithm

The naïve algorithm uses a constant block size and grid dimension.

```
griddim.x = ceil(N/blockdim.x)
griddim.y = ceil(N/blockdim.y)
```

This ensures that enough threads are spawned, such that every value of C is computed in a separate thread (n^2 threads).

```
c = 0

for k = 0 . . . N - 1
   a = A[(i_B,y * dim_B,y + i_T,y) * N + k]
   b = B[k * N + (i_B,x * dim_B,x + i_T,x)]
   c += a * b

C[(i_B,y * dim_B,y + i_T,y)* N + (i_B,x * dim_B,x + i_T,x)] = c
```

We can tune two parameters of this basic algorithm to realize best performance. These two parameters are $dim_{B,x}$ and $dim_{B,y}$. Given our hardware configuration, a block can contain a maximum of 1024 threads. Therefore, $dim_{B,x} \times dim_{B,y} <= 1024$. Also, $dim_{B,x}$ and $dim_{B,y}$ can't be over 1024 since it's the maximum block size.

To measure performance of the naïve algorithm with respect to varying block sizes, we modified the statically compiled block dimensions from the `Makefile` to be able to modify block sizes from command line interface. This provided us with the flexibility to script out a routine which collects average performance in Gflop/s for all possible $dim_{B,x}$ and $dim_{B,y}$. For matrix sizes N = 256, 512 and 1024 average performance over 10 reps were considered. While for matrix size N = 2048, we considered 3 reps.

The data collected was then plotted on a heat map to analyse performance across different Ns for varying block sizes. The graph shows that the performance increases monotonically in a sawtooth manner until block sizes 16 x 64, & then follows a decreasing trend. Figure 2.1 shows the performance of the basic algorithm at different block sizes (powers of 2) for n = 256, n = 512, n = 1024, and n = 2048. For n = 2048

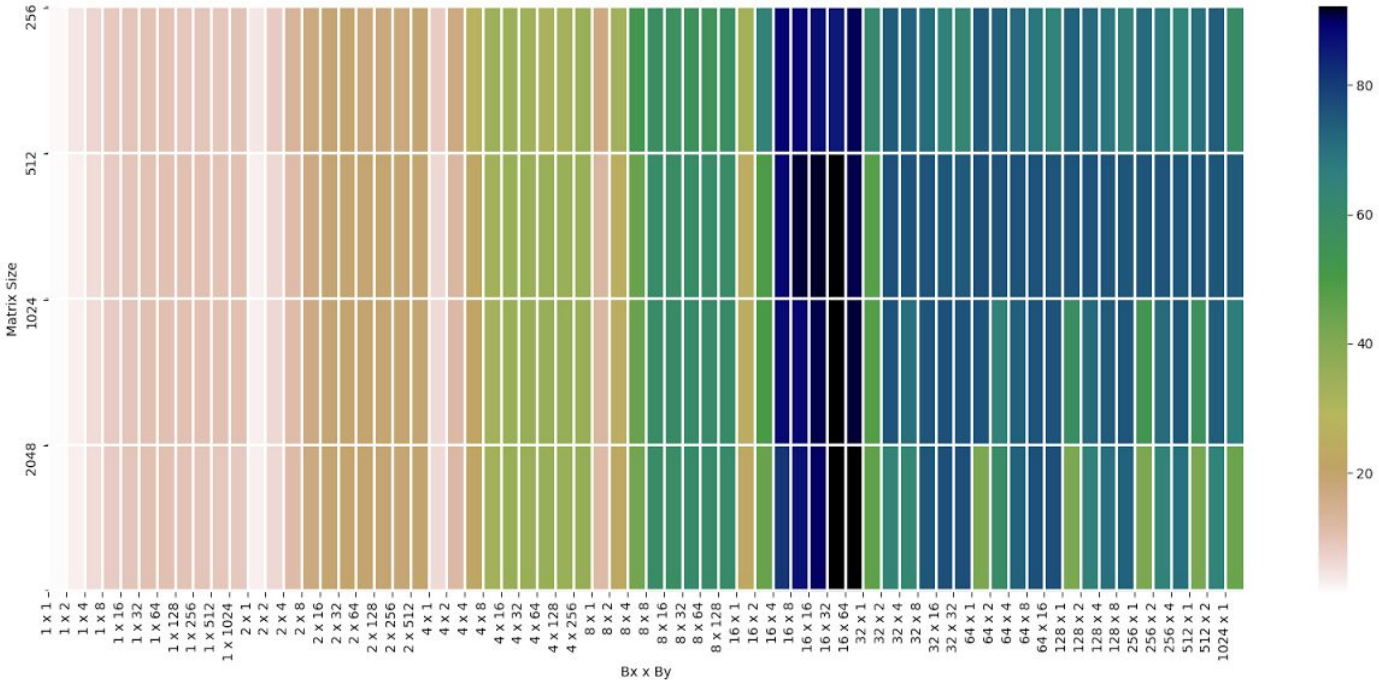


Figure 2.1 Naïve Matrix Multiplication Algorithm at different block sizes

Matrix Size N	Peak Performance	Bx x By
256	90.39 Gflop/s	16 x 64
512	92.04 Gflop/s	16 x 32
1024	92.19 Gflop/s	16 x 32
2048	91.97 Gflop/s	16 x 64

3. Optimizations

This section explains on the optimizations techniques that were considered to result in an algorithm with performance beyond 360 Gflop/s.

3.1 Optimizations using Shared Memory

The naïve matrix multiplication algorithm doesn't make use of the onboard Shared Memory. As we have seen earlier during our study on the Tesla K80 GPU cards, shared memory is MUCH faster than the global memory which we've been using in the basic algorithm. To put it in numbers, GK210 GPUs have shared memory bandwidth of 240 GB/s which amounts to 480 GB/s for a Tesla K80. We can expect considerable performance boost from this step.

3.1.1 Increasing Computation-to-Memory Ratio by Tiling

The current basic algorithm spawns N^2 threads where each thread acts independently. In this step, we force the usage of shared memory by applying block algorithm. In this method, small blocks of A and B matrices are computed by combined effort of the individual threads in the block. Each thread now loads a single value of the needed rows of A block & columns of B block into the shared memory. The reads to get the required values for multiplication, then happens from shared memory instead of global memory, thereby reducing the global

memory traffic. The threads cooperate and coordinate towards the computation of the target matrix C. Figure 3.1 shows how a typical block tiling algorithm is adapted to force usage of shared memory.

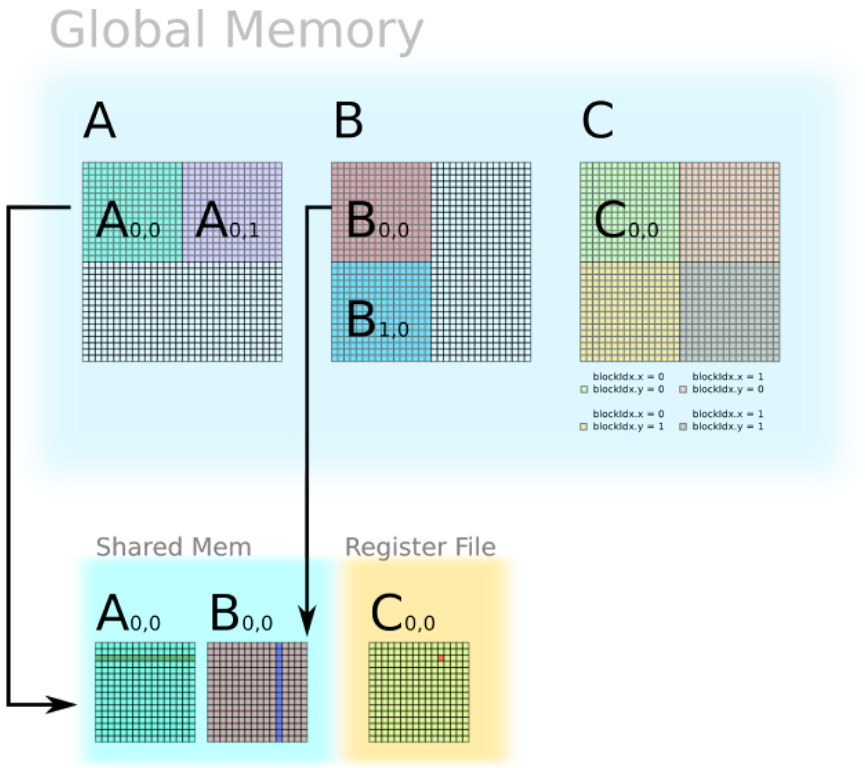


Figure 3.1 Block Tiling Algorithm for Shared Memory usage (5KK73 GPU assignment website)

3.1.2 Feasible Parameter size

We used square block sizes to implement the block algorithm for shared memory. ie., dim_B x dim_B. This means that every element loaded into the blocks is being reused dim_B times. Reusability could mean lesser global memory traffic. We chose to optimize our algorithm with square block sizes because, that way we take full advantage of the shared memory. Non-Square block sizes $dim_{B,x}$ x $dim_{B,y}$ would not be effective because, the threads end up reading values that are not needed for the current block. This means that some values needed for multiplication still needs to be accessed from global memory. Considering this drawback, we did not implement non-square block sizes for shared memory.

3.1.3 Coalescing Accesses to Global Memory

While optimizing for shared memory accesses, we figured we can parallely optimize the memory accesses for Global memory as well. This would mean that the matrices A and B are accessed in rows sequentially and not in columns. This can be done by modifying the indices while loading A block & B block (referred to as A_s and B_s) from source matrices. This would mean that the read accesses are coalesced & consolidated by GPU, resulting in peak bandwidth rates.

3.1.4 Performance study on Shared Memory and Coalesced Global Memory optimizations

Blocking algorithm is implemented with square block sizes. Let's call the size, dim_B for convenience.

```
_shared_ A_s[dim_B][dim_B], B_s[dim_B][dim_B]
c = 0

for kk = 0 . . . dim_{G,y}

    A_s[i_{T,y}][i_{T,x}] = A[(i_{B,y} * dim_B + i_{T,y}) * N + kk * dim_B * i_{T,x}]
    B_s[i_{T,y}][i_{T,x}] = B[(kk * dim_B * i_{T,y}) * N + (i_{B,x} * dim_B + i_{T,x})]

    (synchronize)

    for k = 0 . . . dim_B

        c += A_s[i_{T,y}][k]   * B_s[k][i_{T,x}]

    (synchronize)

C[(i_{B,y} * dim_B + i_{T,y}) * N + (i_{B,x} * dim_B + i_{T,x})] = c
```

We need to synchronize all threads twice:

- once after we buffer A_s and B_s to ensure that the whole block is buffered when the multiplications start
- once after we computed the multiplications to ensure that no data from the next block is already loaded while some threads still compute multiplications for the previous block.

The current modification from the naïve matrix multiplication algorithm has imposed a strict cap on the parameter dim_B. Let's measure the performance of our algorithm for varying block sizes with two conditions imposed $dim_{B,x} = dim_{B,y}$ and $dim_{B,x} \times dim_{B,y} <= 1024$.

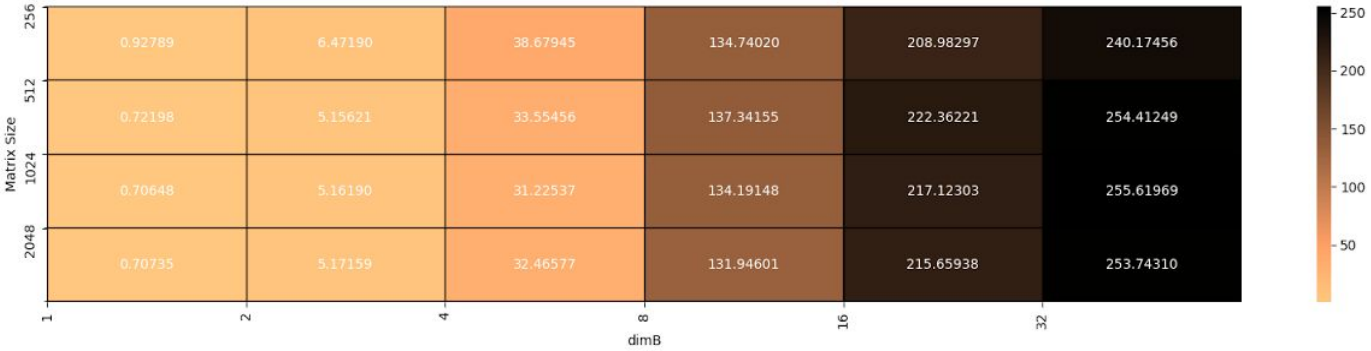


Figure 3.2 Performance of Matrix multiplication after using shared memory & coalescing

As we can see from Figure 3.2, the performance of our algorithm has significantly improved over the naïve algorithm. The peak performance of our algorithm right now is at 255.62 GFlop/s. But however, for smaller block sizes, ie., $dim_B = 1 \text{ or } 2$ the performance achieved by our algorithm is less than that of the naïve one. This is because of the overhead caused due to bringing the blocks to shared memory. Because obviously, what are we asking using shared memory? When $dim_B = 1 \text{ or } 2$, we merely bring in 1 to 4 elements to shared memory and expect the performance to increase. But the overhead is not counterbalanced by the computational speed achieved on smaller blocks as we expected. For larger blocks however, using shared memory gives a good boost over the naïve code - almost 2.77 times more GFlop per second. Let's analyse to get the block dimension that gives the best performance.

Matrix Size N	Peak Performance	Bx x By
256	240.17 Gflop/s	32 x 32
512	254.41 Gflop/s	32 x 32
1024	255.62 Gflop/s	32 x 32
2048	253.74 Gflop/s	32 x 32

As we can see, 32 x 32 blocks are giving the best performance for our algorithm across all block sizes. Let's carry this block size over, for future optimizations on top of Shared Memory. In addition to this, it's to be noted that we are avoiding memory bank conflicts, because of the chosen block size & tile size. This way we assure that the threads spawned don't access the same shared memory banks.

3.2 Inducing Instruction Level Parallelism

According to a paper on Understanding Latency hiding on GPUs by Vasily Volkov,

"Faster codes run at lower occupancy"

We may be losing performance, by continually trying to maximise occupancy. We can try to hide the arithmetic & memory latency using fewer threads. By just using shared memory, we have a huge bandwidth gap to attain peak performance. The larger the bandwidth gap, the more data must come from registers. This may require many registers = low occupancy. The only way to get closer to the peak performance is by increasing the use of registers per thread. This can be done by using fewer threads. The idea behind this method is to make a single thread compute multiple outputs for neighbouring cells of the target matrix. This means more data is local to a thread in registers, which may decrease shared memory accesses. Lower occupancy should not be a problem because, the fewer threads perform more work in parallel, thereby increasing instruction level parallelism.

3.2.1 Computing 8 outputs per thread

Right now, the algorithm using shared memory computes one output per thread. This algorithm uses 30 registers per thread. We do 2 multiply and adds (MAD) for every two shared memory accesses which amounts to 8 bytes/flop. We are now bound by shared memory bandwidth to attain peak performance. In our following implementation, while keeping the grid dimensions constant in both x and y directions, we reduce the block size by a factor of 8 in y direction. This way we end up using 8x lesser threads, decreasing the occupancy of our algorithm. In our following implementation, a single thread computes the following values of C.

1. $C[((i_{B,y} * dim_B + i_{T,y}) + 0*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$
2. $C[((i_{B,y} * dim_B + i_{T,y}) + 1*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$
3. $C[((i_{B,y} * dim_B + i_{T,y}) + 2*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$
4. $C[((i_{B,y} * dim_B + i_{T,y}) + 3*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$
5. $C[((i_{B,y} * dim_B + i_{T,y}) + 4*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$
6. $C[((i_{B,y} * dim_B + i_{T,y}) + 5*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$
7. $C[((i_{B,y} * dim_B + i_{T,y}) + 6*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$
8. $C[((i_{B,y} * dim_B + i_{T,y}) + 7*dim_B/8) * N + (i_{B,x} * dim_B + i_{T,x})]$

With this implementation, each thread now has 8 memory stores with 16 MADs but uses the same amount of shared memory. But, we have increased the register usage from 30 to 63.

3.2.2 Feasible parameter size

From our previous study on algorithm using shared memory, we decided to carry over the block size of 32 x 32. But however, since we reduce the number of threads per block by a factor of how many outputs each thread computes, we are going to have the following block dimensions.

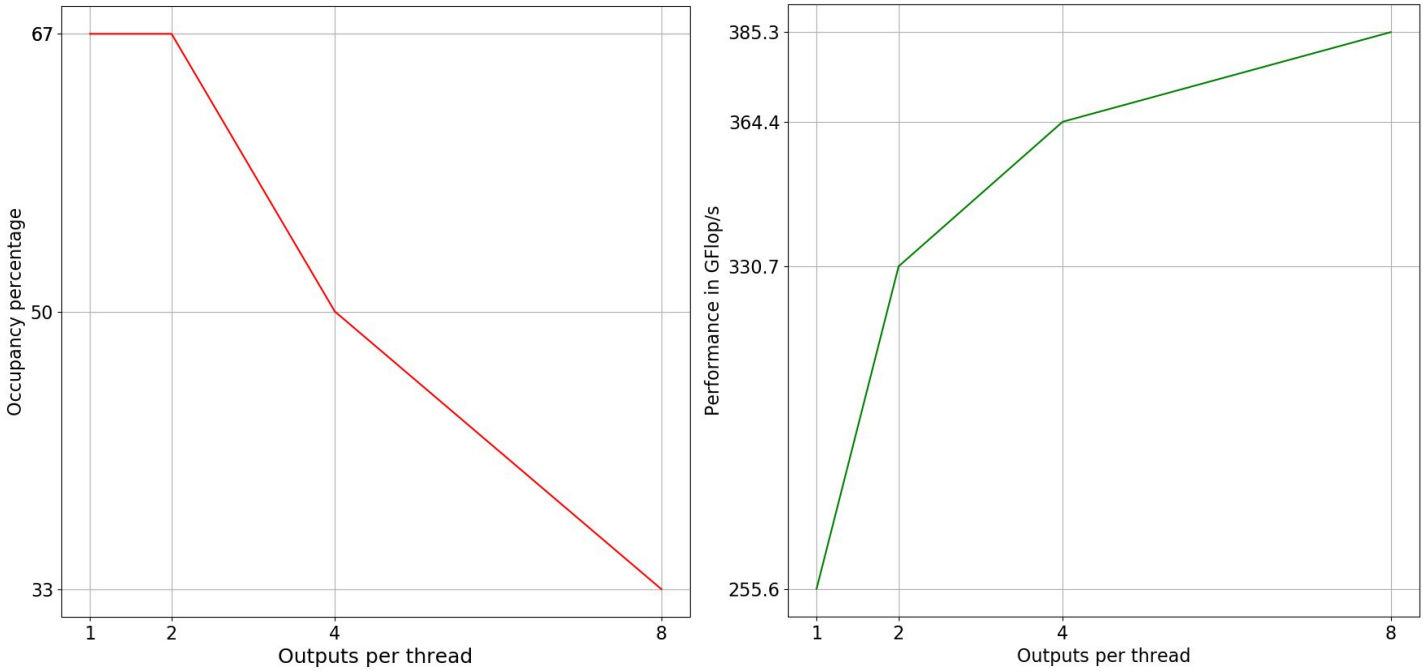
$$dim_{B,x} = dim_B$$
$$dim_{B,y} = dim_B \ / \ \text{outputs per thread}$$

3.2.3 Performance study with stepwise increase on outputs per thread

From 1 output per thread, we measured the performance for in steps of 2^i amounting to 8 outputs per thread. We measured the performance of our algorithm at every step for a matrix of size N = 1024.

Outputs per thread	Peak Performance	Registers per thread	Threads per block	Thread blocks per SMX	Occupancy
1	255.62 Gflop/s	20	1024	1	67%
2	330.72 Gflop/s	28	512	2	67%
4	364.39 Gflop/s	41	256	3	50%
8	385.26 Gflop/s	63	128	4	33%

Let's plot peak performance and occupancy vs outputs per thread.



At 8 outputs per thread, we have reduced the shared memory accesses from 8 B/flop. As we'd expect, increasing occupancy will increase performance only upto a certain point. Beyond this point, our algorithm

becomes memory bound, ie., we increase the computations as much as possible per thread and each thread has a memory latency which is hindering performance. To avoid this, we need to increase computations per thread to try and hide the memory latency. This is possible by computing more outputs per thread.

4. Drawbacks, Limitations and Observations

4.1 Drawbacks of increasing ILP by unrolling by more than 8 computes per thread

Following the presentation by Volkov, we wanted to increase the unrolling by more than 8. The first value we tried was 16 following the pattern of increment so far. To our surprise we could not gain performance by unrolling it by 16. We were curious about this and tried to see if we can find the source of bottleneck, the first thing we experimented on was to see if the 16 unrolling would affect even in the case of floating point number. We saw that in the case of floating point number, we were actually having a slight increase in performance when compared to 8. At the end we concluded that in the case of 16 unrolling for double precision we do not have enough threads to hide the various latencies. But in the case of float, since our latencies are not as long as double, we will be able to hide the latencies with a fewer amount of threads.

After concluding that 16 is not an option for unrolling we were curious as to find if we have a value between 8 and 16 which would give us the optimal performance for the square tiles. The first value we tried was 9 with a 36x4 thread block but with 9 we had a huge drop in the performance and this was not expected. On further inspection we understood that the problem over here was the memory coalescing. In this case when x dimension of the thread block will be 36, so the first warp would take 32 consecutive values from the global memory but the second warp would be reading 4 values of A from the first row and the remaining 28 from a different row and we lose memory coalescing because of this. We can expected similar drop in performance for the values between 9 and 15 as even in this case we will have trouble with memory coalescing.

Based on the above two observations we believe for that given restrictions of shared memory and the use of only square tiles, 8 computes per threads would give up the optimum performance.

4.2 Shared memory limitations on GK210

Based on the observation from the previous sections, we can see that we would have problems with memory coalescing when we are working on square tiles whose dimensions are not multiples of 32 (number of warps scheduled at once). After this we wanted to see if we could have an increase in the performance by increasing dimension such that they still follow the restriction stated above. Our tile size at this moment was 32 x 32, so the next tile size which we could have tired was 64 x 64. But when we modified the code to this tile size, we faced a problem and the code did not compile. Digging into this we realised that we were crossing the limit for the shared memory. The max shared memory for a multiprocessor was 112KB which was well above the amount of memory 64x64 tiling was using but the issue was with the limit of shared memory for a thread block. Per each thread block we are allowed a maximum of 48KB and because of this we can clearly remove 64x64 tiling as an option.

Based on this section and the discussion of the previous section we believe for the case of square tiling we can get an optimum performance with a 32x32 tile size.

5. Performance Study

Our final algorithm averages out at 390 Gflop/s for a matrix size of 1024 x 1024. Our algorithm works for matrix sizes which are not powers of 2 as well. But the performance drops, for such cases. This can be attributed to the fringes formed because of the parameters chosen (grid dimensions & block dimensions). We measured peak performance of our algorithm at different values of N with $dim_{B,x} = 32$ and $dim_{B,y} = 4$. Figure 5.1 shows how much faster our implementation of Matrix multiply algorithm performs against the BLAS benchmark.

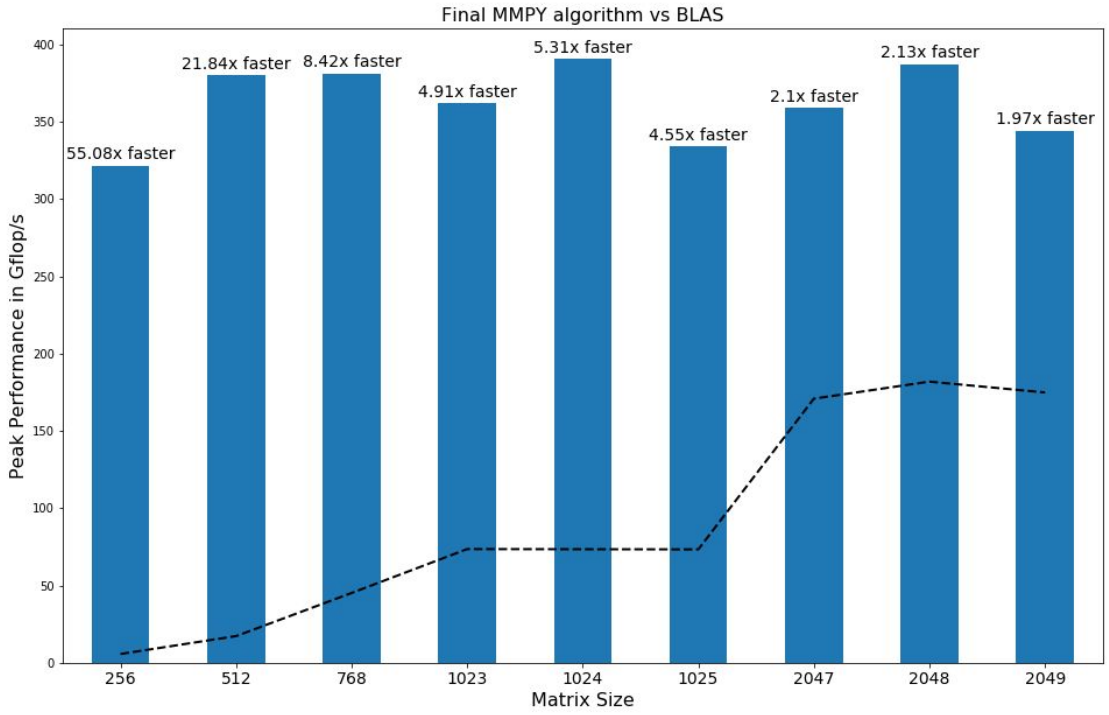


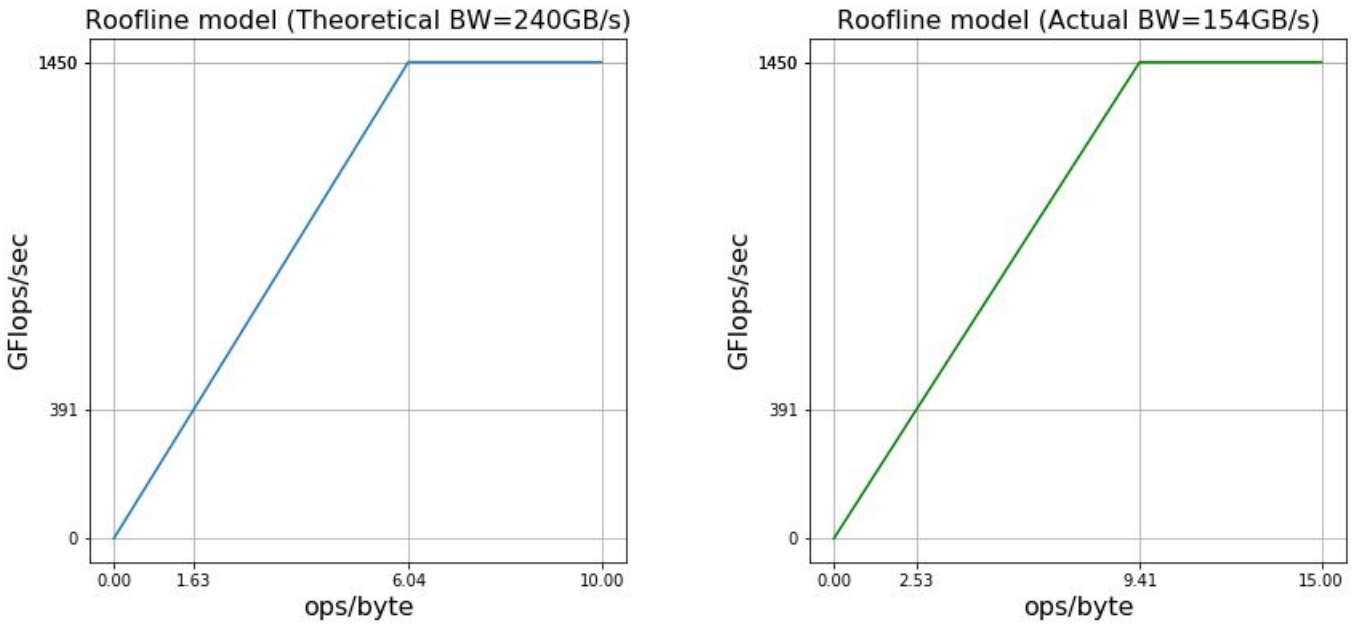
Figure 5.1 Speed up ratio (S) against BLAS benchmark

5.1 Computational Intensity (Q) of our algorithm

Assuming double-precision peak performance of about 1.45 TFlop/s, let's tabulate to see how much fraction of this theoretical peak we attain with our algorithm.

Matrix Size	Peak Performance	% of the theoretical peak
256	324.95 Gflop/s	22.18%
512	381.44 Gflop/s	26.30%
1024	390.71 Gflop/s	26.94%
2048	387.04 Gflop/s	26.69%

Using GK210 GPUs bandwidth of 240GB/s to global memory, let's plot a roofline model for the GPU along with the achieved N=1024 performance on this plot. From the plot, let's estimate the value of Q in ops/byte. Volkov's dissertation states that the actual bandwidth is less than 240GB/s - and measures this as 154 GB/s. Let's see how Q varies with the actual memory bandwidth.



Q = 1.63 **Q = 2.53**

Figure 5.2 Roofline model for theoretical and actual bandwidths.

It make sense that computational intensity increases with decrease in memory bandwidth which could mean our matrix multiply algorithm is working as expected.

3.4 Future Improvements

All our implementation was done using square tiles for all the matrices (A, B, C). One of the possible things that can be done is to try out tiles that are rectangular for (A, B) or for (A, B, C). Using a rectangular tiles we can avoid the bottleneck in the case of shared memory and have tile sizes of 64 x 32 and 32 x 64 for A and B respectively and still have enough shared memory. Using a rectangular tile we might see an improvement even in the case when we are unrolling by 16 as we can cleverly create enough threads to hide the latency. **Thread granularity,** is another thing that can be tried as a future implementation. The basic idea is to make the thread from one thread block compute values that are across blocks. This would increase the ILP and so we might see an improvement in the performance as well.