Matrix Multiplication
Assignment 2
CS6023

Anant Shah
EE16B105
Department of Electrical Engineering

September 16, 2018

Abstract

Matrix multiplication is an operation whose operation time can be drastically reduced using parallel processing techniques. With the help of the memory hierarchy in a GPU, such as registers, shared memory and global memory, optimal performance can be achieved. This report looks at the running time for matrix multiplication for different grid and block dimensions along with the use of global memory and shared memory. Matrix multiplication is also demonstrated using the tiling algorithm which utilizes the shared memory in a Streaming Multiprocessor.

1 Matrix Multiplication using Global Memory

The matrix multiplicands are square matrices whose width is $N = 8192$, and their $(i, j)^{th}$ element is defined as

$$A_{i,j} = i * 2.1 + j * 3.2$$

The block dimension used is $(16, 16)$ which implies that the number of threads in the x-direction is 16 and the number of threads in the y-direction is 16. For the operation of matrix multiplication, each thread calculates one cell in the output matrix. The dimension of the block is 2-D but the threads that form a warp are chosen in a row-major order, hence the first 2 rows in a block form a warp, the next two form a warp and so on. Hence in one warp, threads are accessing the same data from the global memory(as required by matrix multiplication) hence giving a very low operational intensity. Due to this, this algorithm is bound in the memory region by the roof-line model and hence is inefficient.

Now, the matrix multiplication is performed using two variations:

1. The fastest varying index is '.x' in the kernel.
2. The fastest varying index is '.y' in the kernel.

On timing the kernel execution time for each, the following time values were obtained :

Sr No.	Time(s) [fastest varying index is '.x']	Time(s) [fastest varying index is '.y']
1	53.964	11.347
2	53.991	11.341
3	54.000	11.341
4	53.976	11.345
5	53.972	11.340
Average Time(s)	53.981	11.343

It can clearly be seen that there is a speed-up with a factor of :

$$speedup - factor = 4.76$$

Consider the first case when the fastest varying index is ".x". Let the thread in a block be indexed by

$$tid = [ty][tx]$$

The row and column that it will be calculating in the output matrix will be

$$a = P[row][col]$$

where

$$row = blockIdx.y * blockDim.y + tx$$

$$col = blockIdx.x * blockDim.x + ty$$

In the second case, where the fastest varying index is ".y", the corresponding element in the output matrix will be

$$row = blockIdx.y * blockDim.y + ty$$

$$col = blockIdx.x * blockDim.x + tx$$

The reason for the speed-up is the way the two variations access elements of the matrix multiplicands from the global memory. When threads in a warp(following SIMD instructions) access contiguous memory locations from the global memory, the hardware realizes the condition and coalesces all the accesses into one "burst" hence enabling faster movement of data. If random data elements are accessed, the kernel will hit stalls due to the low latency global memory accesses and hence decrease the performance of the operation. This is the reason why we see a speed up when the fastest varying index is ".y". Consider the case of a (16,16) dimensional block. The first 2 rows will form a warp, next two another warp and so on as warps are created by grouping threads by first listing them out in row major order. It is important to note that the threads in a warp are working in SIMD fashion and hence are accessing data parallely. When the fastest varying index is ".x" all the threads in a warp access different rows from the first multiplicand but the same column from the second multiplicand. The rows that are accessed are continuous(M[i],M[i+1],M[i+2],..), but since matrices in the global memory are stored in row-major order, these accesses do not form a burst section and hence we see the slow kernel run-time in the first case. However in the second case, the threads in a warp access the same row from the first multiplicand and different columns from the second multiplicand. These columns which are being accessed are one after the other and hence fall in a burst section as threads are accessing their element in parallel. For example : N[i][j],N[i][j+1],N[i][j+2],etc. are all accessed simultaneously by different threads. Since the access of elements fall in burst sections, we see a much faster run-time.

The kernel run-time was observed by varying the block dimensions. The block dimensions (32,16) and (32,32) were run using separate #define statements and hence are not a part of the code. The run-time for the same are depicted in the table below(the readings are an average of 10 readings taken) :

Block-Dimension	4*4	4*8	8*4	8*8	8*16	16*8	16*16	16*32	32*16	32*32
Average Time(s)	39.394	30.182	19.857	17.493	17.971	11.854	11.295	11.175	14.005	14.027

The plot for the kernel run-time vs the block-dimension is depicted in Figure 1.

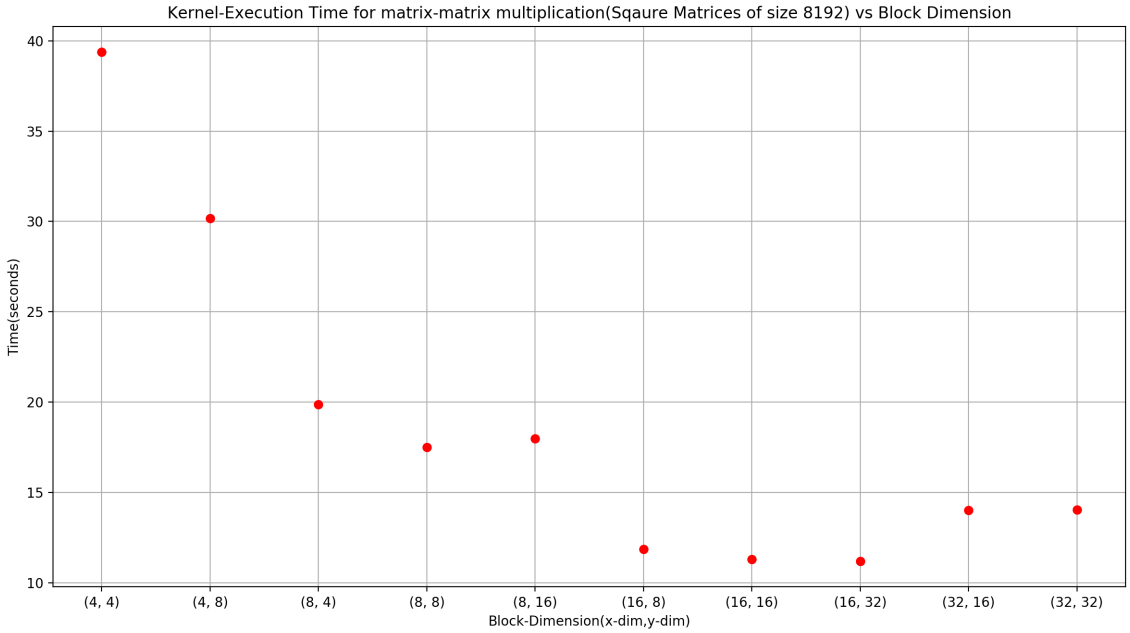

Figure 1 : Kernel-Runtime vs Block-Dimension

2 Matrix Multiplication using Shared Memory

As mentioned earlier, we need to increase the operational intensity(operations/byte) and hence we need an algorithm which will re-use data. Matrix multiplication is perfect to implement this as threads(that calculate one output cell) share the same row or the same column to compute their output. GPUs have a certain memory hierarchy which allow this kind of implementation. The different types of memories in a GPU are :

1. Registers - these are local to a thread
2. Shared Memory - these reside on the SM and are shared by the threads in a block
3. Global Memory - typical D-RAM memory and resides off the SMs
4. Constant Memory - memory which is read-only and off the SMs

All these memories have their certain specification such as read-write access, etc. but the main point is that accessing data from the shared memory is much faster than accessing data from the global memory. Hence, once we load the data from the global memory to the shared memory, all the threads in a block can access that data and hence reduce the run-time drastically. However, we are constrained by the size of the shared memory on the SM. There is another constraint on the number of registers per thread but that is less important as compared to the constraint on the size of the shared memory. For the Tesla K40C, the size of the memories are given below :

1. Maximum Shared Memory Per Block - $48kB$
2. Shared Memory Per Multiprocessor - $48kB$
3. Maximum Registers Per Block - 65536
4. Registers Per Streaming Multiprocessor - 65536

From the values above, we can clearly see that if a block utilizes more than the allocated shared memory, the program will not run. This is clearly shown in the example below.

Consider the problem of finding the matrix multiplication of two sqaure matrices with size $N = 32$ and $(i, j)^{th}$ element as

$$a_{i,j} = i * 2.1 + j * 3.2$$

We now want to use shared memory to compute the product. To use shared memory, all the threads in a $(16, 16)$ block load the corresponding elements co-operatively into the shared memory. We will need to load $16 * 32$ elements(16 rows of elements corresponding to the 16 threads in the y dimension of the block and 32 corresponding to all the 32 columns) from the first multiplicand and 32*16 elements(16 columns of elements correspoding to the all the rows and the number of threads in the x dimension of the block) from the second multiplicand into the shared memory. The shared memory can accomodate this data as its size(32*32*8) is less than the full capacity of the shared memory. However, if the size of the multiplicands is now $N = 8192$, the shared memory required per block will exceed the maximum limit. Following similar calculations as above, the total amount of data that would be required in the shared memory per block would be 32*8192*8 bytes ,which comes out to be around $1.98MB$, which is far exceeding the maximum limit and hence we see an error at compile-time itself. Shown below is the run-time for the 32*32 multiplication and the compile time message for the 8192*8192 multiplication :
1. N=32 : Run-Time(milli-seconds) : 0.1527040005
2. N = 8192 : ptxas error : Entry function '_Z9matrixMulPdS_S_i' uses too much shared data (0x100000 bytes, 0xc000 max)

3 Tiling

To overcome the constraint on the amount of shared memory, the algorithm of tiling is introduced. Instead of loading all the elements(shared by the threads in a block) at once, tiles of the data are co-operatively stored into the shared memory. The first and most necessary condition for this algorithm to work is that the thread accesses should match spatially and temporally, or in simpler words, they should have similar schedules. Second, the tile size should be chosen such that the total space required to store the tiled elements does not exceed the shared memory capacity. The elements from each tile are co-operatively loaded into the shared memory and then computed upon. After all threads in the block finish their computations for that tile, their partial sums are stored in their respective registers after which the next tile is co-operatively loaded into the shared memory and then the same procedure is followed until the end of the row or column is reached. We are basically dividing the loads as compared to the previous example where everything was loaded at once. This partial loading allows us to by-pass the shared memory size constraint.

The global memory traffic is reducing by a factor of N, where N is the tile-width. As compared to the D-RAM accesses, another factor which plays a role is the burst section. CUDA devices allow high global memory access efficiency by organizing memory accesses of threads efficiently such that they are coalesced. Threads in a warp execute the same instruction at a time(following the SIMD pattern), so when they load data from the global memory, the hardware detects if they are accessing contiguous memory locations. If they are, the hardware coalesces all theses accesses into one "burst". We have to note that in tiling, threads are parallely loading data from the global memory into the shared memory and hence since threads in one row of a block access contiguous elements, they lie in one burst section and hence give a faster access time. Also since we are re-using the data(sharing data between threads) the operational intensity has increased hence pushing this operation into the compute bound region of the roof-line model and hence giving us maximum performance. Due to the above reasons, we expect to see a considerable improvement in the run-time using tiling as compared to without using shared memory.

The tile width needs to be chosen accordingly so as to obtain maximum performance. The factors it depends on are :

1. Size of the shared memory - The tile size should not exceed that of the shared memory.
2. Maximum number of threads per block - The number of threads per block should not be too large as that will lead to stalls due to the sync threads command after the loading phase and after the partial sum calculation for a tile.
3. Maximum number of threads per warp - Since threads in a block are grouped into warps based on the row major order, the layout of threads in a block need to be such that threads in a warp access contiguous memory locations in the global memory so as to form burst sections.
4. Size of the burst section(does not play that significant of a role as the burst sections in reality are pretty large)

Given below are the run-time of two different procedures for matrix multiplication(sqaure matrices with $N = 8192$), one with tiling as shown in this section and one without using shared memory as described in section 1 of this report. The tile width is chosen to be equal to the block-width and hence each thread has to load just one element from each matrix multiplicand from the global memory.

1. Without Shared Memory : Run-Time(seconds) - 11.295
2. With Tiling : Run-Time(seconds) - 4.819

The tile-width is now varied(keeping the tile-width and block-width same) and the kernel execution time for matrix multiplication using tiling is observed. The average run-times(average of 5 readings) are shown in the table below :

Sr. No.	Time(s)-Tile(4*4)	Time(s)-Tile(8*8)	Time(s)-Tile(16*16)	Time(s)-Tile(32*32)
1	30.734	8.045	4.822	4.191
2	30.659	7.997	4.819	4.185
3	30.657	7.997	4.817	4.173
4	30.658	7.997	4.817	4.173
5	30.658	7.997	4.817	4.174
Average Time(s)	30.673	8.007	4.819	4.179

It is seen from the table that the optimal tile-size is $(32, 32)$. The plot for the kernel-time execution is show in Figure 2 :

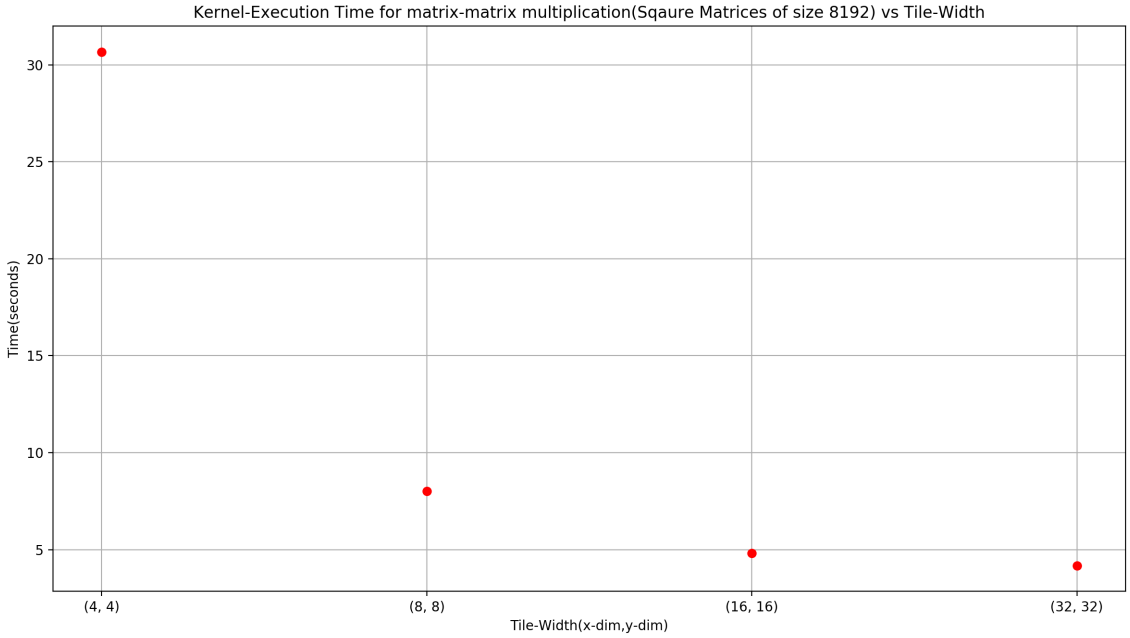

Figure 2 : Kernel-Execution Time for matrix multiplication vs Tile-Width

4 Rectangular Matrix Multiplication

A kernel is written to perform matrix multiplication for rectangular matrices. The block of threads is still considered to be 2-D with block dimensions of (16,16). Conditions are added in the code so that for arbitrary block dimensions, unnecessary elements are not calculated(elements which go out of the bound of the matrices). This will lead to control divergence in some cases. The size of one of the input matrix is 4096*8192 and the size of the other matrix is 8192*16384. The $(i, j)^{th}$ element of each matrix is given by

$$a_{i,j} = i * 2.1 + j * 3.2$$

The kernel execution time for the matrix multiplication is (average of 5 readings):
Run-Time(seconds) : 11.367

5 Bonus

As mentioned in the sections above, the tiling algorithm decreases the run-time drastically due to its efficient use of shared memory. So for rectangular matrix multiplication where the input sizes are 8192*16384 and 16384*32768, the fastest run-time obtained was :
Run-Time(seconds) : 33.466

The block-size and tile-size configuration for the run-time obtained above are :

1. Block-Dimension - (32,32)
2. Tile-Dimension - (32,32)

The roof-line model for the Tesla K40C is shown in Figure 3.

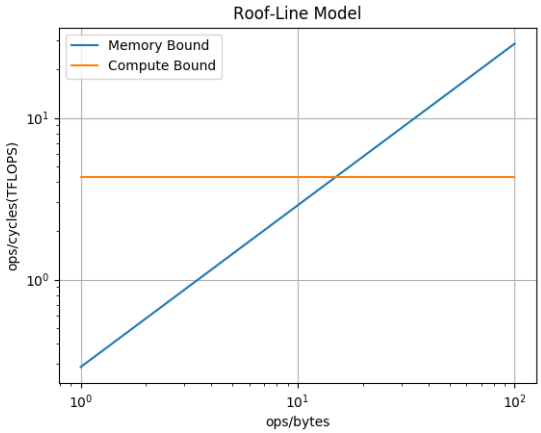

Figure 3 : Roof-Line Model for the Tesla K40C