Ajay Thakkar

2/21/2024

CPE 810

Lab 1

Lab Questions:

1. How many floating operations are being performed in your dense matrix multiply kernel if the matrix size is N times N? Explain.

Each element in a matrix multiplication algorithm will consist of two floating point operations, one for multiplying the corresponding matrix values and another for adding it to the total sum for a given row and column. For two N x N matrices, a dot product for one element in the resulting matrix will compute N elements. This will be done N^2 times for the resulting matrix. Therefore, there are N^3 computations and two floating point operations for each of these computations giving 2*N^3 floating point computations.

2. How many global memory reads are being performed by your kernel? Explain.

Without using the tiling algorithm, there are two global memory access for every computation. Therefore, for two N x N matrices, there would be 2 * N^3 global memory accesses according to the same logic of the floating point computations. However, when using the tiling algorithm, we load in a chunk of each matrix into shared memory only once, and then go onto the next chunk of the two matrices. This means there is one global memory read for every element in each of the two input matrices, or 2 * N^2 global memory reads.

3. How many global memory writes are being performed by your kernel? Explain.

There is a global memory write for every element in the resulting matrix, so for an N x N resulting matrix there would be N^2 global memory writes.

4. Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

Tuning the tile size can improve performance, as reducing the tile size can allow for better pipelining between different threads. As well as this, implementing data reuse would allow for less global memory reads if data that is needed again is already in the shared memory. Ensuring memory coalescence is also another way to improve speedup, since this drastically reduces the time for global memory access.

5. Suppose you have matrices with dimensions bigger than the max thread dimensions allowed in CUDA. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication in this case.

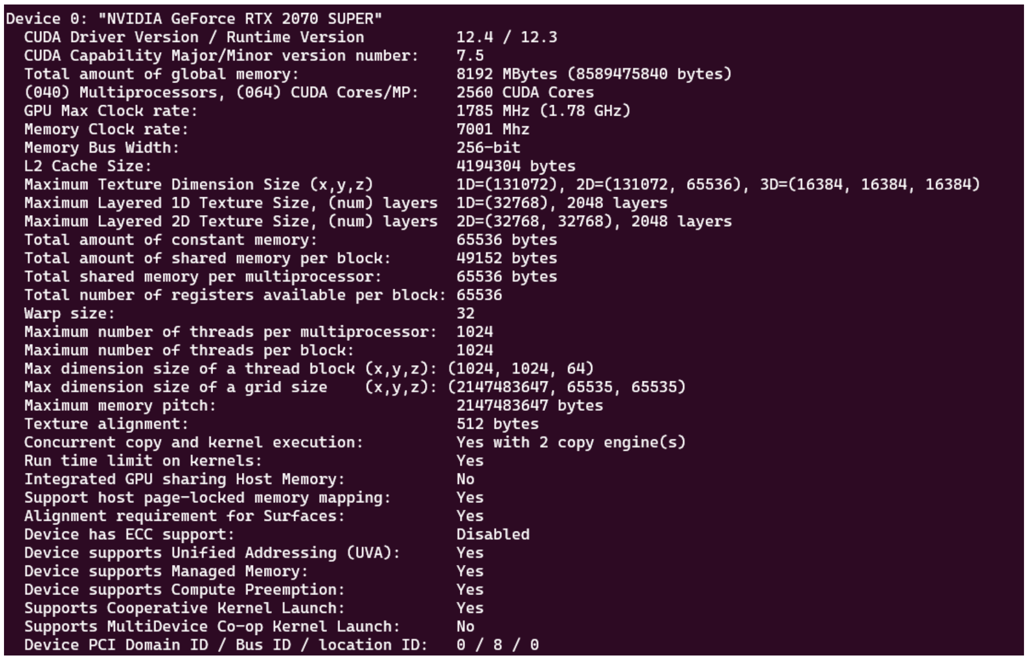
One way to handle matrices with bigger dimensions is to have every thread be responsible for two data points instead of just one. This can be done by making the grid size half of the resultant matrix's width. Instead of each thread being responsible for only one element in the resultant matrix, it will now be responsible for two elements. Two column indices are calculated by simply multiplying the normal column index. Bounds checking can ensure that both of these column indices are within range of the resultant matrix. With two column indices, two dot products can be calculated from each thread, allowing a matrix with bigger dimensions to fit within a smaller amount of threads at the expense of reduced parallelism.

6. Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform matrix multiplication algorithm that would perform the multiplication out of place.

To compute a matrix multiplication on matrices that would not fit in global memory, the logic of tiling can be used before kernel invocation. A full resultant matrix will be allocated on the device, and the input matrices can be divided into chunks on the host side. Then, a for loop can be used to launch multiple sequential kernel to compute partial sums with the sub-matrices and store these partial sums in the corresponding element in the device's resultant matrix. These partial-sums will be compounded from the multiple kernel invocations, allowing for a full matrix multiplication. This allows for a smaller amount of global memory usage by only loading a chunk of the input matrices into global memory.

Implementation Details:

This program utilizes a NVIDIA GeForce RTX 2070 Super with the following specifications:



```
Device 0: "NVIDIA GeForce RTX 2070 SUPER"
  CUDA Driver Version / Runtime Version          12.4 / 12.3
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 8192 MBytes (8589475840 bytes)
  (040) Multiprocessors, (064) CUDA Cores/MP:    2560 CUDA Cores
  GPU Max Clock rate:                            1785 MHz (1.78 GHz)
  Memory Clock rate:                             7001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 8 / 0
```

WSL was used for hosting the CUDA toolkit, running the program on the GPU, and for debugging using cuda-gdb.

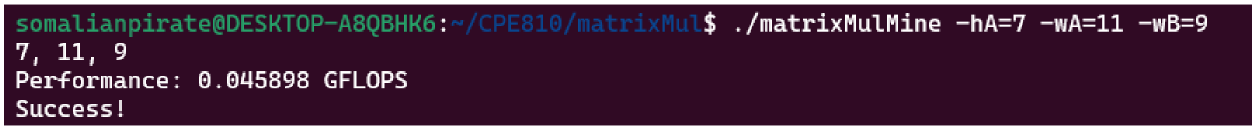
The “main” function first checks for command line arguments when running the program that specify the height and width of matrix A and the width of matrix B. This is done with the use of helper functions from the CUDA samples repository. Then the matrix dimensions are passed to the matMul function, which will handle pre-processing and post-processing before and after the kernel invocation.

The matMul function first allocates memory on the host with the given dimensions and fills one matrix with 1s and the other matrix with .01s. This means the resulting matrix should have every value set to the width of matrix A times .01 which is useful for checking accuracy of the result. After creating the host matrices, memory is allocated on the device for the input and output matrices, and the input matrices are copied from the host to the device. Before invoking the kernel, the block and grid dimensions are set using a pre-defined tile size. Then, some helper functions are used for timing the completion of all kernel launches and calculating the GFLOPS that the GPU performs. Finally the device resultant matrix is copied to the host, an accuracy check is performed, and the memory on the host and device are freed.

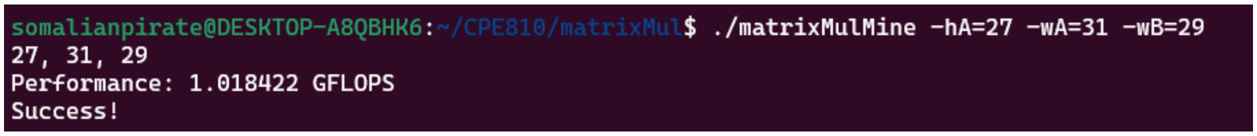
The kernel itself first initializes two shared memory sub-matrices for tiling. Then, the row and column of the thread are calculated and stored. Next, a for loop is used to loop through a variable number of phases depending on the size of the matrices and the block size. Inside, if-else statements are used for bounds checking to store data from the global memory into the two sub-matrices. If the data point is within bounds, it is stored in the sub-matrix, otherwise a zero is placed in the sub-matrix for zero padding. Then, for barrier synchronization the syncthreads function is called before computing the partial dot product of the two tiles. Another syncthreads function is called to make sure all threads are finished computing before loading in the new tiles. This is done for all the phases until the sums are finished, and they are finally placed in the resultant device matrix.

Experimental Results:

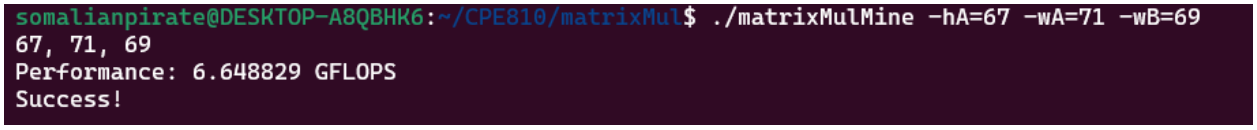
Block Size: 8



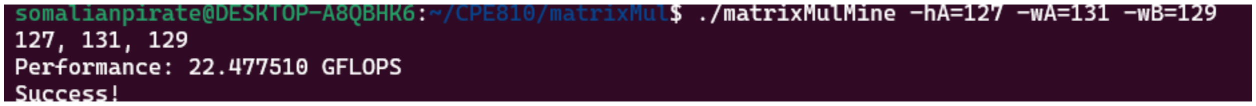
```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=7 -wA=11 -wB=9
7, 11, 9
Performance: 0.045898 GFLOPS
Success!
```



```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=27 -wA=31 -wB=29
27, 31, 29
Performance: 1.018422 GFLOPS
Success!
```

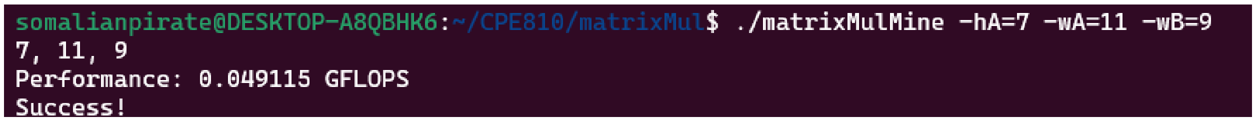


```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=67 -wA=71 -wB=69
67, 71, 69
Performance: 6.648829 GFLOPS
Success!
```

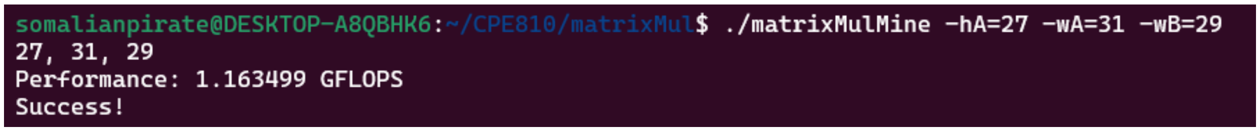


```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=127 -wA=131 -wB=129
127, 131, 129
Performance: 22.477510 GFLOPS
Success!
```

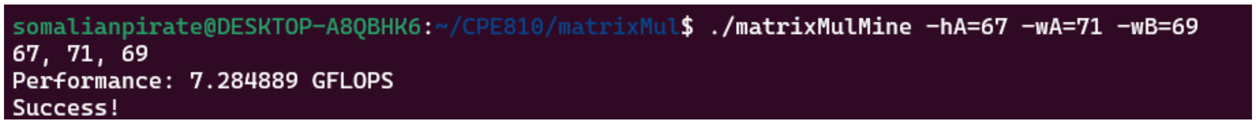
Block Size: 16



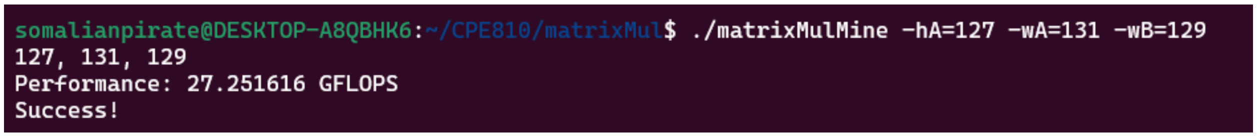
```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=7 -wA=11 -wB=9
7, 11, 9
Performance: 0.049115 GFLOPS
Success!
```



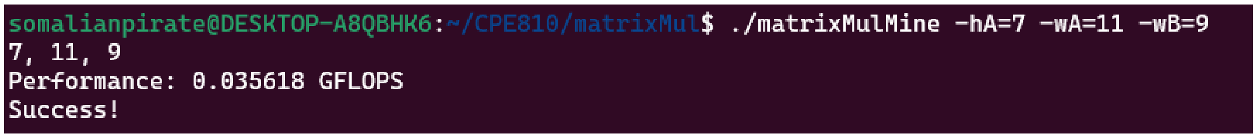
```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=27 -wA=31 -wB=29
27, 31, 29
Performance: 1.163499 GFLOPS
Success!
```

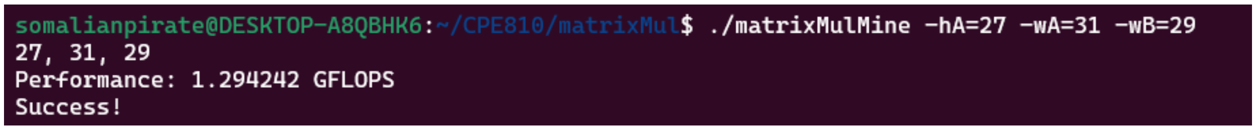


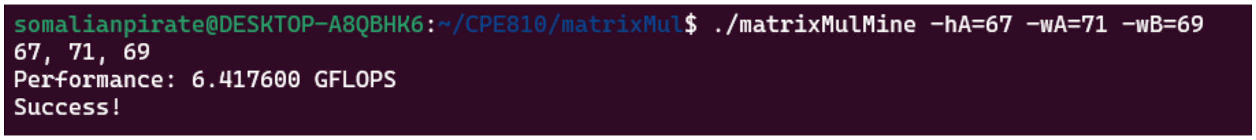
```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=67 -wA=71 -wB=69
67, 71, 69
Performance: 7.284889 GFLOPS
Success!
```

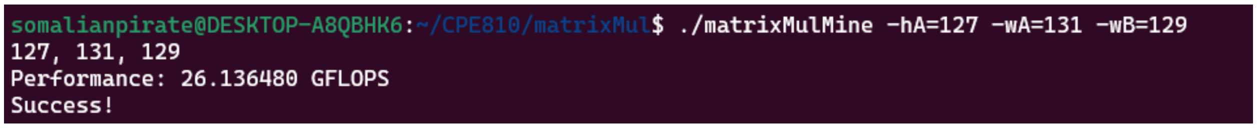

```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=127 -wA=131 -wB=129
127, 131, 129
Performance: 27.251616 GFLOPS
Success!
```

Block Size: 32


```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=7 -wA=11 -wB=9
7, 11, 9
Performance: 0.035618 GFLOPS
Success!
```


```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=27 -wA=31 -wB=29
27, 31, 29
Performance: 1.294242 GFLOPS
Success!
```


```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=67 -wA=71 -wB=69
67, 71, 69
Performance: 6.417600 GFLOPS
Success!
```


```
somalianpirate@DESKTOP-A8QBHK6:~/CPE810/matrixMul$ ./matrixMulMine -hA=127 -wA=131 -wB=129
127, 131, 129
Performance: 26.136480 GFLOPS
Success!
```

Discussion:

The first observation that can be made is that as more data points are used, the performance of the kernel increases. This means that it has not yet reached its bottleneck in terms of how many floating point operations it has to compute, and going above the biggest matrix sizes in the experiments launches too many threads for my device to handle. Another observation corresponds to the tradeoff between having smaller and larger block sizes. We can see that a block size of 8 is significantly worse than a block size of 16 or 32. This would lead us to believe that bigger block sizes perform better, but the block size of 32 is worse than the block size of 16.

This is because as block sizes get smaller, there is both better pipelining in the CUDA cores between memory accesses and computations. As well as this, a smaller tile size allows for a higher chance of memory coalescence, since a bigger tile will load in more memory from global memory lowering chances of the memory being in the same area.