CS410: Parallel Computing

Assignment 4

1. Introduction

In this assignment, we implemented matrix multiplication on GPUs using CUDA-C and conducted various experiments related to parallelism, whose observations can be seen in this report.

2. Implementation

Version 0: Basic Implementation

__global__ *MatMul(A, B, C, n)*:

```
1. Compute the global row and col of the thread
2. int tmp = 0
3. for k=0...n:
4.      tmp += A[row][k]*B[k][col]
5. C[row][col] = tmp
```

In this basic version, we calculate each element of matrix C using one thread. We divide the whole matrix C into a 2-D grid of blocks, where each block consists of 32 threads along each dimension. For calculation of each element of C, an appropriate row of A and column of B are loaded from the global GPU memory into the local memory of the thread.

Version 1: Tiled Matrix Multiplication

__global__ *TiledMatMul(A, B, C, n)*:

```
1. Compute the global row and col of the thread
2. Initialize the shared memory s_a and s_b for storing a tile of
   matrix A and matrix B respectively.
3. int tmp = 0
4. for i=0...n/TILE_SIZE:
5.      Load the appropriate tile from A and B
6.      __syncthreads();
```

```
7.      Perform matrix multiplication on the tile loaded into
   memory and add the result to tmp.
8.      __syncthreads();
9. C[row][col] = tmp
```

In the tiled version, we divide the matrix A, B and C into tiles of size TILE_SIZE along each dimension. We assign the responsibility of calculating each element in the tile to a block of threads. We load a tile required for computing an element of C into the shared memory between threads in the same block and perform the matrix multiplication on this tile. We add the result of this matrix multiplication over all the tiles required for computing a tile of matrix C. The idea behind the tiled implementation is to reduce the cost of reading elements from the global memory.

We use **`__syncthreads()`** two times in the above function. The first usage is to ensure that all the threads within the same block load their appropriate element into the shared memory before any thread starts utilizing the shared memory for computing the inner product. The second usage is to ensure that all the threads have computed the inner product (utilized the shared memory) before a new element is loaded into the shared memory in the next iteration.

In both versions, we use **`tmp`** to store the intermediate results instead of directly adding to the C[i][j] because this leads to higher reading costs from the global memory.

3. Experiments

Note all the times mentioned in this section are in milliseconds.

Experiment 1:

Host to Device Data Transfer Time

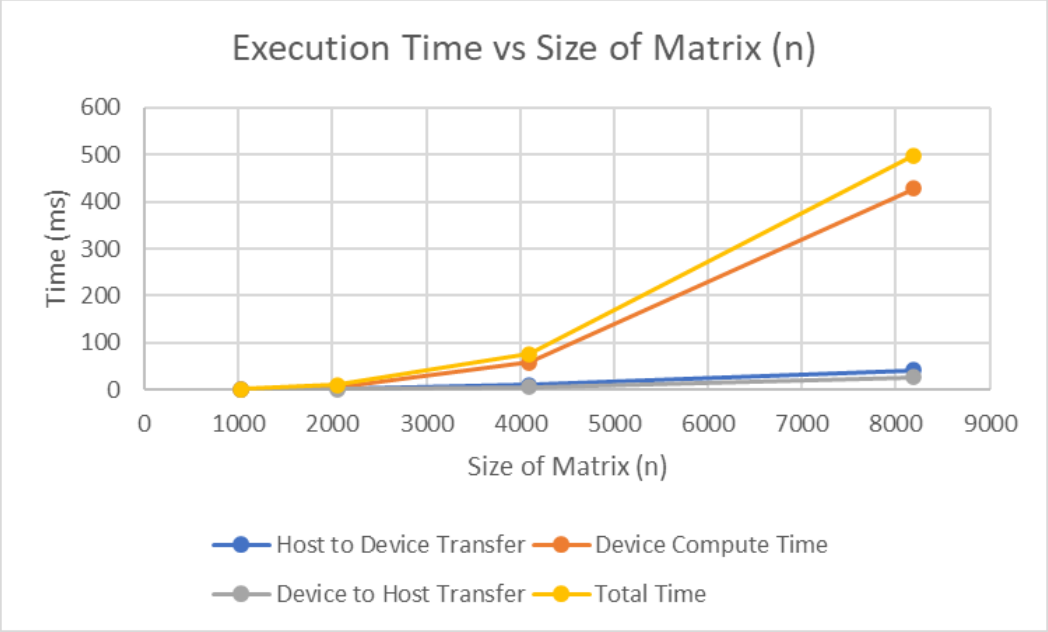
	Run 1	Run 2	Run 3	Average
n=1024	0.800544	0.78672	0.791744	0.793003
n=2048	2.81974	2.77811	2.77254	2.79013
n=4096	10.6396	10.6911	10.6095	10.64673
n=8192	42.1928	42.0825	41.845	42.0401

Device Compute Time (Kernel Execution Time)

	Run 1	Run 2	Run 3	Average
n=1024	1.3545	0.629792	0.63072	0.871671
n=2048	10.0978	4.72118	4.7256	6.51486
n=4096	86.8639	45.0792	45.0683	59.0038
n=8192	393.153	452.033	440.494	428.56

Device to Host Data Transfer Time

	Run 1	Run 2	Run 3	Average
n=1024	0.515264	0.523008	0.47296	0.503744
n=2048	1.72614	1.66733	1.72272	1.705397
n=4096	6.79869	6.50566	6.48589	6.596747
n=8192	28.7991	28.7384	25.5212	27.68623



NOTE: Total time = Host to Device Transfer time + Device Compute time + Device to Host Transfer time

In version 0, we exploit SIMD parallelism by making all the threads within the same block execute the same instruction on a different data. In other words, all the threads within the same block calculate the inner product between a row of A and a column of B. However, for the calculation of C[i][j], the thread with row=i and column=j computes the inner product between row i of A and column j of B. This way each thread works on different data. Moreover, to further exploit parallelism, each block is executed on a different core on the GPU.

In the graph above, we observe that kernel execution time (device compute time) forms the major part of the total execution time. This is because the data transfer operation is linear in the size of the matrix whereas matrix multiplication is naively cubic in size of the matrix. The total execution time increases as we increase the size of the matrix which is expected because the number of computations increases as we increase the size of the matrix.

Experiment 2:

Host to Device Data Transfer Time

	Run 1	Run 2	Run 3	Average
n=1024	0.817856	0.687552	0.702208	0.735872
n=2048	2.43264	2.4256	2.39939	2.41921
n=4096	9.18659	9.23485	9.26691	9.22945
n=8192	35.8636	36.0768	36.1177	36.01937

Device Compute Time (Kernel Execution Time)

	Run 1	Run 2	Run 3	Average
n=1024	0.97696	0.455424	0.455776	0.629387
n=2048	7.17782	3.31648	3.31632	4.60354
n=4096	58.3609	26.9154	26.9131	37.39647
n=8192	290.355	305.699	293.582	296.5453

Average Memory Related Operations Time per Block

	Run 1	Run 2	Run 3	Average
n=1024	0.018445	0.019194	0.019026	0.018888
n=2048	0.01887	0.02148	0.0214	0.020583
n=4096	0.024861	0.031606	0.031593	0.029354
n=8192	0.029693	0.029053	0.02943	0.029392

Average Compute Related Operations Time per Block

	Run 1	Run 2	Run 3	Average
n=1024	0.058927	0.058464	0.058591	0.05866
n=2048	0.062624	0.060976	0.061014	0.061538
n=4096	0.059279	0.053412	0.053421	0.055371
n=8192	0.055332	0.056179	0.055541	0.055684

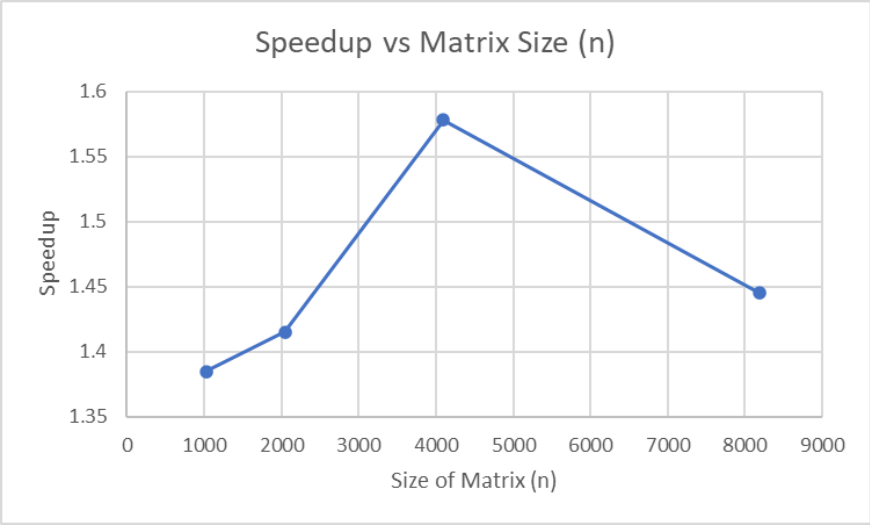
Device to Host Data Transfer Time

	Run 1	Run 2	Run 3	Average
n=1024	0.48624	0.460544	0.468288	0.471691
n=2048	1.70509	1.64406	1.66077	1.669973
n=4096	6.3584	6.5472	6.35763	6.421077
n=8192	25.7938	25.1496	27.3376	26.09367

Speed up calculation

	ver0	ver1	Speedup
n=1024	0.871671	0.629387	1.384953
n=2048	6.51486	4.60354	1.415185
n=4096	59.0038	37.39647	1.577791
n=8192	428.56	296.5453	1.445175

NOTE: The average values obtained in the above experiments have been used for speedup calculation. The speedup has been calculated as the ratio of the kernel execution time of ver0 to ver1. This is done because the rest of the costs will be similar for both versions.



In the tiled version implementation (version 1), we observe a speedup as compared to version 0. This is because in the tiled version due to the introduction of shared memory between threads in a block, we reduce the number of times a thread in a block reads from the global device memory. In the naive implementation (version 0), each row of A and column of B is accessed (read from the global device memory) TILE_SIZE times more than the tiled version (version 1). For example, if the TILE_SIZE=32, then each row of A and column of B is accessed 32 times more in version 0 as compared to version 1. This is because each element in the tile will load its corresponding element into the shared memory. Hence, the communication costs are reduced by a factor of 32 in the tiled version.

In version 1, also we exploit SIMD parallelism because each thread in the block computes an inner product between the row of a tile and a column of a tile (same instruction) but each thread utilizes a different row and column of tile (multiple data). Moreover, each block can be executed on a different core of the GPU.