Report

Computing C = A×B on GPU

5th May 2021

OVERVIEW

Creating two matrices, A and B, each of size (N × N). Initialising the matrices to random floating point numbers.
Writing an **CUDA** code for computing C = A×B. Reporting the times taken for the codes by varying the size of the problem from N = 100 . . . 10000.

Approach-1

A naive implementation on GPUs which assigns one thread to compute multiplication elements of resultant matrix. Here, each thread loads one row of matrix A and one column of matrix B from global memory, does the inner product, and stores the result back to matrix C in the global memory as shown in figure-1*.

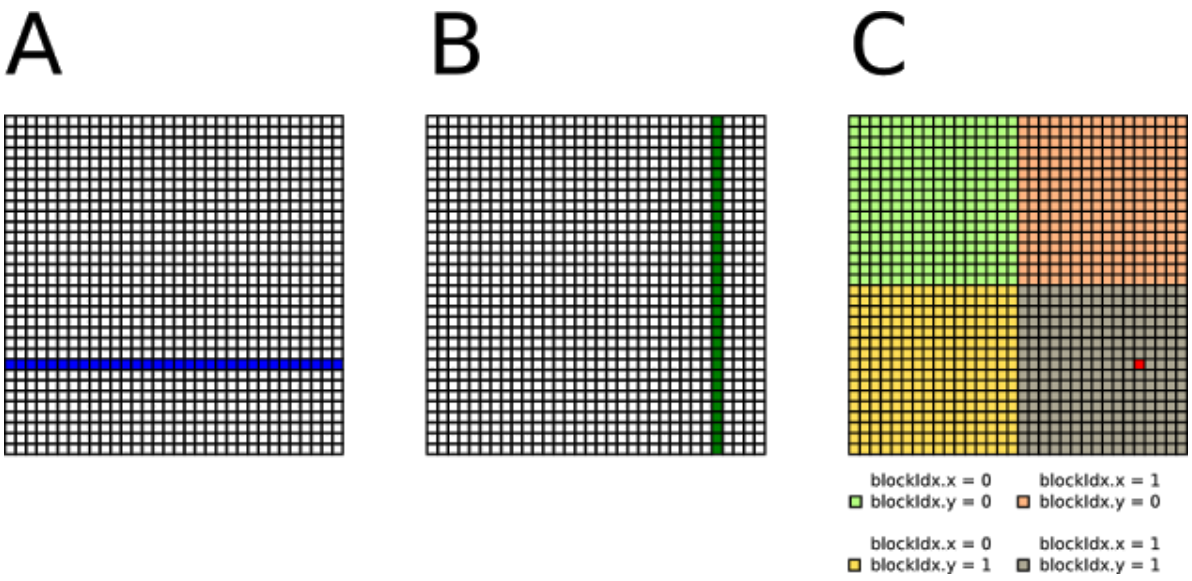

Figure-1*

****Reference: http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda***

Algorithm

- Allocate dynamic memory using malloc() with the size (N × N ×size of float) for A (mat1) , B (mat2) and C (mul) for both CPU and GPU i.e. define A_cpu, B_cpu, C_cpu in the CPU memory and A_gpu, B_gpu, C_gpu in the GPU memory
- Initialize random floating numbers to A_cpu and B_cpu. Copy the memory from CPU to GPU i.e. A_cpu to A_gpu and B_cpu to B_gpu.
- Define block size and grid dimension where 16 rows consists of one Block dimension or Block sIze and the grid dimension is N/Block dimension.
- Now, the GPU kernel is defined such that each thread loads one row of matrix A and one column of matrix B and does the inner product.
- Store the result back to matrix C i.e. C_gpu to C_cpu

Approach-2 : Tiling*

Considering an example of 32 x 32 matrix multiplication as shown in figure-2* where a 32 x 32 matrix divided into four 16 x 16 tiles. Lets spawn four thread blocks each for one tile computation.

Here, each thread block will spawn threads equal to the number of elements in each row/column to compute the inner product. Thus, four thread blocks each with 16 x 16 threads will be created.

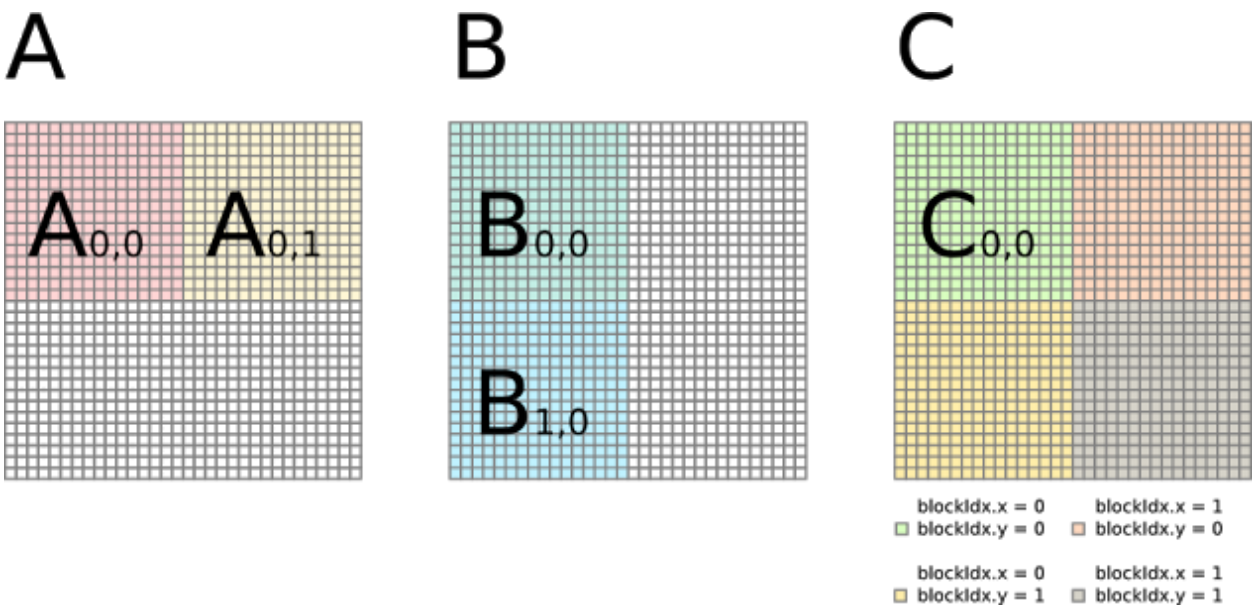

Figure-2*

**Reference: http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda*

One thread block loads one tile of A and one tile of B from cpu memory to gpu memory, performs computation, and stores temporal result of C in register. After all the iteration is done, the thread block stores one tile of C into cpu memory.

For example, a thread block can compute C0,0 in two iterations: C0,0 = A0,0 B0,0 + A0,1 B1,0.

In the tiled implementation, the amount of computation time would be less as compared to the nominal approach and similar can also be observed from the observation table.

Algorithm

- Allocate dynamic memory using malloc() with the size (N × N ×size of float) for A (mat1) , B (mat2) and C (mul) for both CPU and GPU i.e. define A_cpu, B_cpu, C_cpu in the CPU memory and A_gpu, B_gpu, C_gpu in the GPU memory
- Initialize random floating numbers to A_cpu and B_cpu. Copy the memory from CPU to GPU i.e. A_cpu to A_gpu and B_cpu to B_gpu.
- Define block size and grid dimension where 16 rows consists of one Block dimension or Block sIze and grid dimension is N/Block dimension.
- Define shared memory A_tile and B_tile with tile size equal to Block dimension i.e.e 16 in our case.
- Accumulate C “tile by tile” by loading one tile of A and one tile of B into shared memory such that corner cases are taken care of by checking if N is exactly divisible by Block Dimension (i.e. 16).
- Thus, accumulating one tile of C from tiles of A and B in shared memory.
- Store the result back to matrix C i.e. C_gpu to C_cpu

Observations

The results of time taken to compute C = A×B for GeForce GTX 1080 Ti are reported in the table below. The time reported is “real time”. The actual real-time spent in running the process from start to finish as if it was measured by a human with a stopwatch by its definition.

N	Approach-1					Approach-2 Tiling Optimisation
	Block Dimension 2	Block Dimension 4	Block Dimension 8	Block Dimension 16	Block Dimension 32	Block Dimension 16
	Time (sec)	Time (sec)	Time (sec)	Time (sec)	Time (sec)	Time (sec)
5	0.7624	0.7505	0.777	0.721	0.7709	1.6187
100	1.56238	1.5488	1.5399	1.5403	1.57206	1.52282
200	1.52274	1.53034	1.52374	1.5003	1.5421	1.53276
500	1.61162	1.54354	1.57614	1.54268	1.5527	1.54674
1000	1.83066	1.67482	1.63742	1.6076	1.62478	1.60436
2000	2.36358	2.0584	1.91416	1.79964	1.78254	1.75144
5000	7.04638	4.50176	3.90198	3.65404	3.7649	2.8694
10000	47.88654	31.20024	17.43132	15.46396	17.64976	7.95086

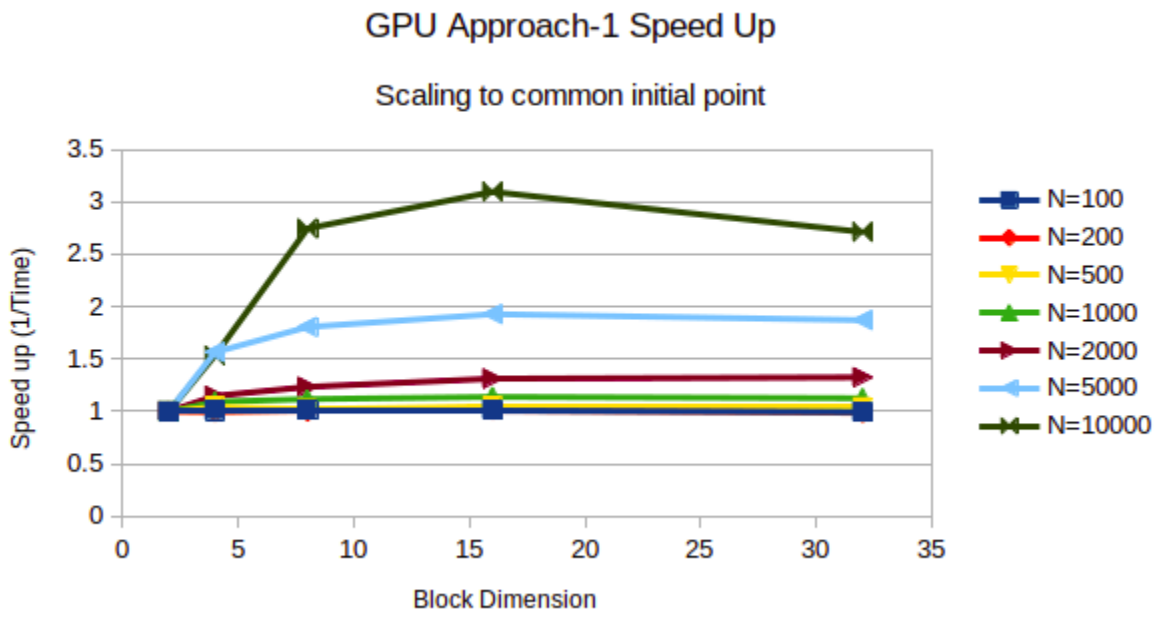

Figure -3

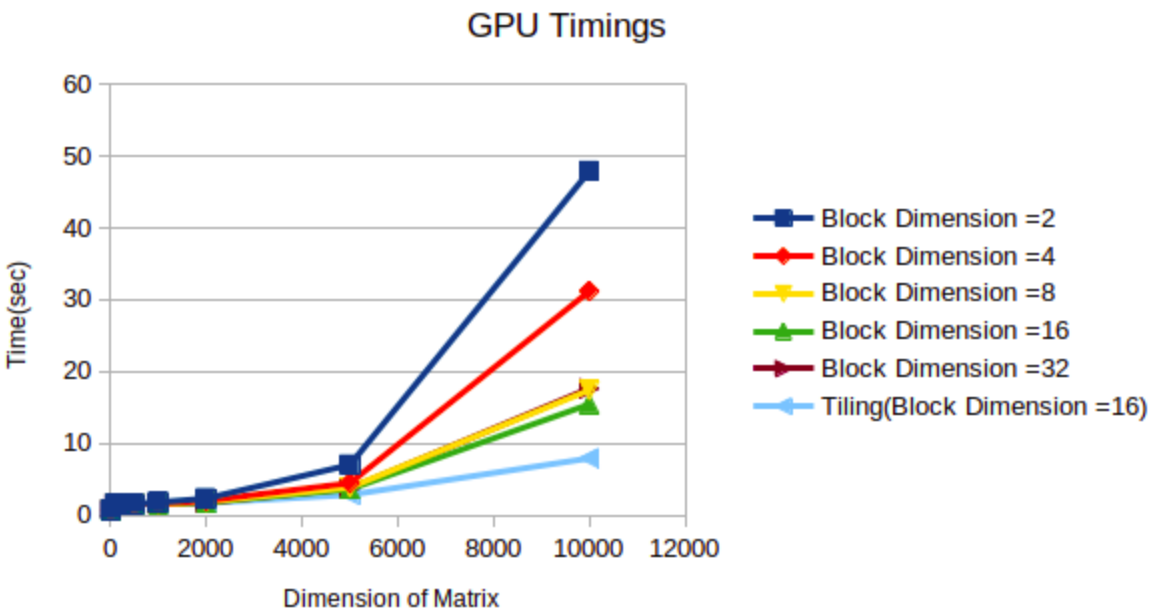

Figure -4

Conclusions

- As the dimension of matrices increases, the speedup increases as shown in figure 3. Also, the parallelisation overheads can be observed for N<500 since there isn't significant speedup observed.
- As anticipated, tiling optimization takes less time to compute than approach-1 as shown in figure-4.
- Time to compute increases for Block Dimension =32 than Block Dimension =16 because of the limit on maximum number of threads per block i.e. 1024. After the limit is reached, multiple parallel operations are serialised.

CPU and GPU Info

.....

//

NVIDIA-SMI 435.21 [GeForce GTX 1080 Ti] Driver Version: 435.21 CUDA Version: 10.1

NVIDIA Corporation GP102 [GeForce GTX 1080 Ti] (rev a1)

GPU Engine Specs

- CUDA Cores3584
- Graphics Clock (MHz) 1480
- Processor Clock (MHz) 1582
- Graphics Performance high-20000

Memory Specs

- Standard Memory Config 11 GB GDDR5X
- Memory Interface Width 352-bit
- Memory Bandwidth (GB/sec) 11 Gbps

//

CPU Info:

No. of processors : 16

vendor_id : GenuineIntel

model name : Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz

OS: Ubuntu 20

//

N--> Dimension of matrices

To run code for Approach-1 Matrix Multiplication on GPU

Simply run $ sudo ./HW3a.sh

for logic correctness and timing analysis

----------or----------

Compile: $ nvcc HW_3a.cu -o normal_GPU.o

Output: $ time ./normal_GPU.o <N>

Example:

```
$ nvcc HW_3a.cu -o normal_GPU.o
$ time ./normal_GPU.o 500
```

To run code for Approach-2 Tiling optimised Mat Mul on GPU

Simply run $ sudo ./HW3b.sh

for logic correctness and timing analysis

----------or----------

Compile: $ nvcc HW_3b.cu -o tiling_GPU.o

Output: $ time ./tiling_GPU.o <N>

Example:

```
$ nvcc HW_3a.cu -o tiling_GPU.o
$ time ./tiling_GPU.o 1000
```

.....