Performance Optimization of Diagonal Matrix Multiplication

Aman Choudhary (*amanc@iisc.ac.in*)

1. Problem Statement

Given two matrices A and B, we need to compute their **Diagonal Matrix Multiplication** (DMM) efficiently. DMM can be visualised as follows:

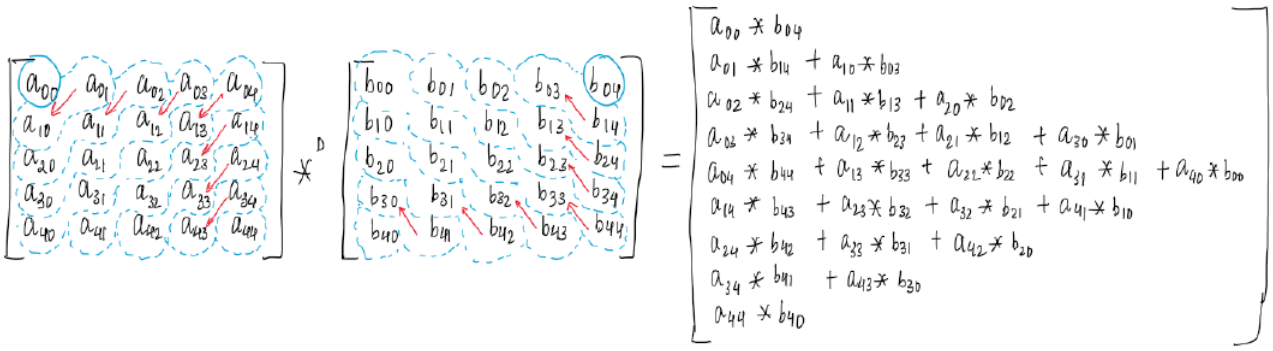

Figure 1: *Diagonal Matrix Multiplication*

2. Recap

- Up till now, we have already optimized our reference version of DMM to some extent. We started with a single threaded implementation, and improved its memory access patterns. Thereafter, we leveraged multiple-threads to gain further speedup. *Figure* 2 briefly summarizes the optimization we did in **Part 1** of the assignment.

- We observe that the maximum speedup was achieved for *input*_16384.*in*, which is approximately *7.7*x. Although, employing multiple threads enables us to parallelize our program to some degree, we can do better. This is where we harness the power of **CUDA** API, to make our program run faster.

3. Enter the CUDA

3.1 First Optimization

- To achieve maximum amount of thread-level parallelism in our program, we will use one CUDA thread for every matrix element. As a result, we would need N^2 threads to perform the multiplication.

- The entire operation is split into **two phases**. In the first phase we, multiply each element of A with its corresponding mapping in B, and store the result back into A. In the next phase, we add the computed products, to appropriate cells in the *Output* array.

Table 1(a)

Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	489.69	2541.39	13547.20
Single Thread	165.26 (-66.25 %)	674.91 (-73.44 %)	3088.68 (-77.20 %)
Multi Thread	96.80 (-80.23 %)	398.85 (-84.31 %)	1761.78 (-87.00 %)

Table 1(b)

Method / Data	input_4096.in	input_8192.in	input_16384.in
Single Thread	2.96x	3.77x	4.39x
Multi Thread	5.06x	6.37x	7.69x

Figure 2: *Table 1(a) summarises the execution time in milliseconds, and drop percentages. Table 2(b) depicts corresponding speedup*

Phase 1

- Let us take a toy example, to illustrate the idea. *Figure* 3 shows two sample matrices A and B of size 16×16. Consequently, our $CUDA$ kernel is composed of 16×16 threads, split across 16 thread blocks. The thread block is laid out as a 4×4 2D grid. Finally, the threads inside the thread block are also arranged in a 2D fashion with dimensions 4×4.
- A pair of elements, one each from A and B is thus assigned per thread. Its sole job is to multiply these two elements and store the result back into A. We have only shown how first four elements of A in each thread block, are mapped to their counterpart in B.
- Coming back to our original implementation, the GPU of the native system supported a maximum of 1024 threads per thread block. Hence, we chose the size of our thread block to be 32×32 accordingly.

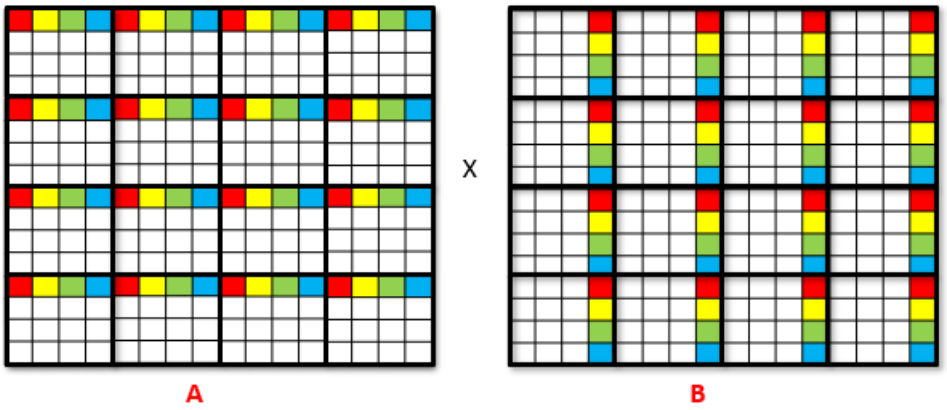

Figure 3: *Phase1 of DMM: Computing the products*

Phase 2

- *Figure* 4(*a*) recalls how the products computed are added to the *Output* array for a sample example of $N = 5$. We illustrate the computations done in Phase 2, by bringing back our toy example. As shown in *Figure*4(*b*), we would need to add elements sitting in A, row-by-row to the output array.

- For this phase, we design a 1D *CUDA* kernel consisting of N threads. For the toy example, these $N = 16$ threads, are split across 4 thread blocks, having 4 threads each. In our actual implementation, the thread blocks consist of 1024 threads.

- We launch the kernel N times iteratively, once for every row. The threads of the kernel add the elements of their designated row, appropriately into the destination.

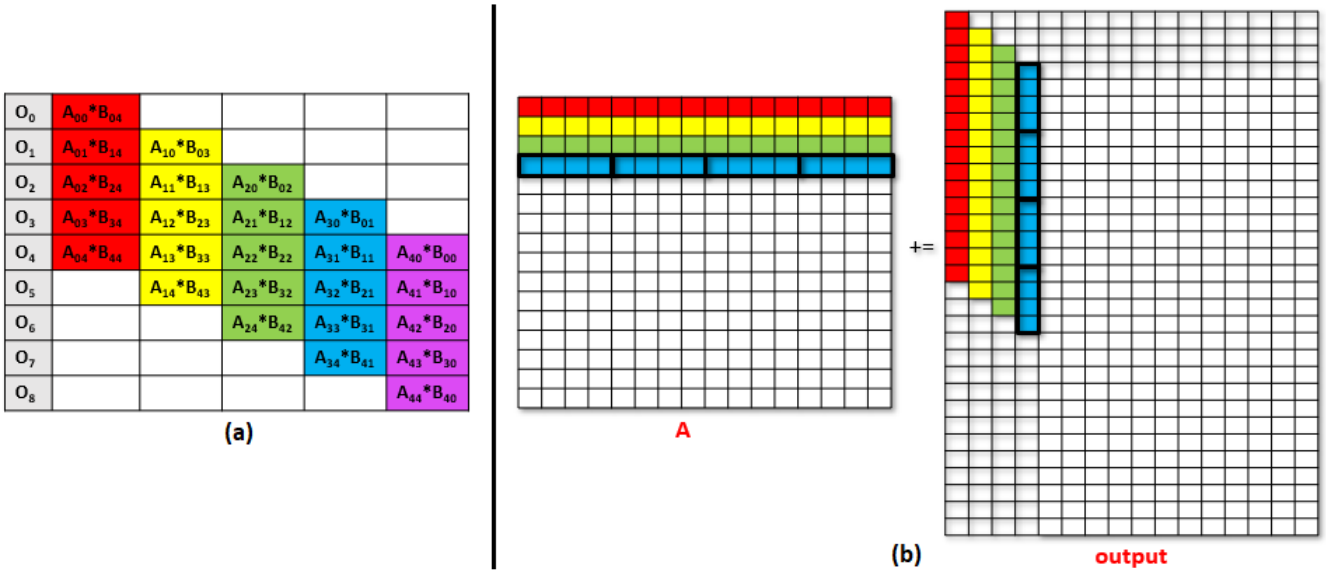

Figure 4: *(a) Adding products to Output array for 5 × 5 matrix*
(b) Phase 2: Adding the products stored in A, row-by-row

3.2 Results

- As expected, a substantial improvement was observed in the performance. Moreover, the speedup increases as the size of the data-set grows. Specifically, we achieved 21 times speedup for *input*_16384.*in*, in contrast to only 7.7 times for the multi-threaded version.

- We note that the *GPU*_1 version of *DMM* fares poorly with respect to the reference implementation, for small data sets (*input*_128.*in*). This is because *CUDA* programs have an unavoidable setup cost in the form of allocating memory on the GPU, as well as copying data between the host and device, which is absent while working solely on CPU.

NOTE: All measurements for execution times were calculated as mean over 5 different runs. The experiments were performed on host (which is different from Part 1) with the following configuration: **CPU** - Intel(R) Xeon(R) CPU @ 2.20GHz, **GPU** - Tesla P100, **OS** - Ubuntu 18.04.5 LTS.

Table 2(a)

Method / Data	input_128.in	input_4096.in	input_8192.in	input_16384.in
Reference	0.10	583.88	2856.17	10617.90
GPU_1	114.29	148.73	233.38	494.31

Table 2(b)

Measure / Data	input_128.in	input_4096.in	input_8192.in	input_16384.in
Drop	+ 114190% %	-74.53 %	-91.83 %	-95.34 %
Speedup	0x	3.93x	12.24x	21.48x

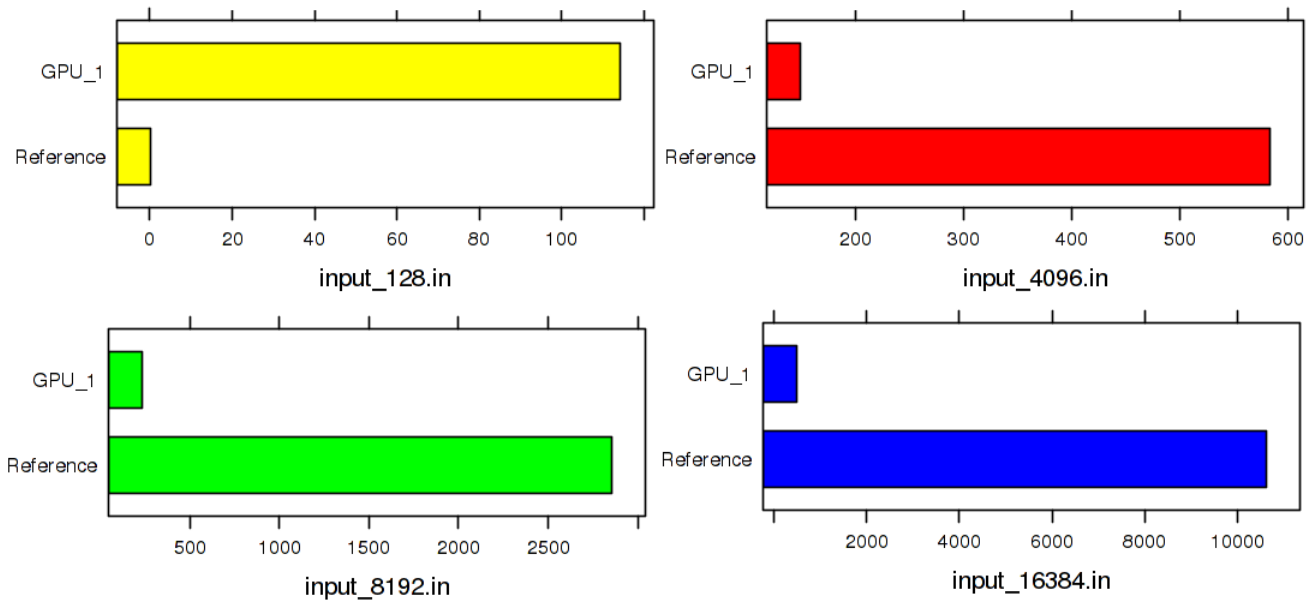

Figure 5: *Comparison of GPU_1 (First Optimization) with Reference for different data sets*

3.3 Second Optimization

- In our first optimization, we are doing our work in two phases. Although, the first phase achieves maximum parallelism possible by employing N^2 threads, the second phase fails to do the same. Even though the individual iterations are N fold parallel, there remains an element of sequential execution, as we add elements iteratively, one row every iteration.

- The second approach to parallelize DMM is inspired by the very first reference implementation. We can use $2 * N - 1$ threads, one for every diagonal in the matrix. Each thread is responsible for computing one single value in the *Output* array by multiplying the appropriate diagonals. *Figure* 6 illustrates the same.

- This approach was also motivated by the fact that we are launching the $CUDA$ kernel N times in Phase 2. As N grows, for example $N = 16K$, we would be incurring overhead in associated calls to the $CUDA$ API. By having only a single kernel do the job in one go would be a certain plus for performance.

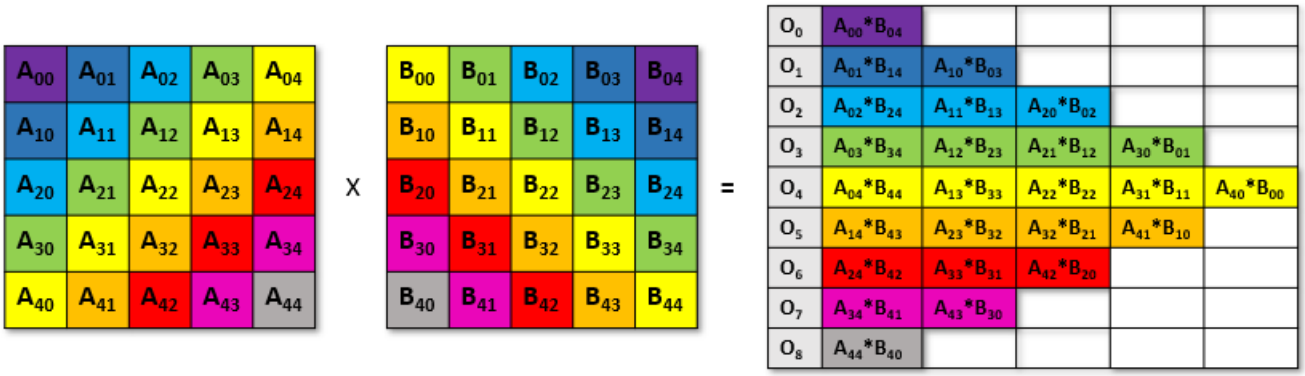

Figure 6: *Performing DMM diagonally*

3.4 Results

- The second optimization did perform better compared to the first one, however, the margin is not too substantial. For e.g, the improvement in execution time is only 7 and 11 milliseconds for *input_4096.in* and *input_8192.in* respectively.
- The reduction in execution time for *input_16384.in* is roughly 34 ms, which is somewhat better compared to the smaller data-sets. The impact of this optimization would be more profound as we increase the size of the data-sets.
- However, the key observation lies, that the execution time for the *CUDA* kernel is not the major bottleneck in performance, instead it is the time taken for memory transfer between the host and device. On running *nvprof* on both the programs, it was identified that, roughly 90 % of the time is spent in copying, and the kernel execution time makes for only 10 % of the overall time.
- The time taken to move data from host to device is a function of the underlying memory bandwidth between them. Since, its a hardware constraint, our scope for optimization reaches a certain blockade. However, all is not lost. We can still improve memory transfers to some extent by using a technique where we allocate our matrices directly on pinned (page-locked) memory using *cudaMallocHost()*. This eliminates the initial cost of staging data from pageable to pinned host memory.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   86.24%  318.68ms         2  159.34ms  158.46ms  160.22ms  [CUDA memcpy HtoD]
                    9.04%  33.400ms     16384  2.0380us  1.7590us  18.048us  copy(int, int, int*, int*)
                    4.72%  17.436ms         1  17.436ms  17.436ms  17.436ms  multiply1(int, int*, int*, int*)
                    0.00%  16.320us         1  16.320us  16.320us  16.320us  [CUDA memcpy DtoH]

            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   92.69%  311.65ms         2  155.82ms  152.64ms  159.01ms  [CUDA memcpy HtoD]
                    7.31%  24.572ms         1  24.572ms  24.572ms  24.572ms  multiply2(int, int*, int*, int*)
                    0.00%  12.192us         1  12.192us  12.192us  12.192us  [CUDA memcpy DtoH]
```

Figure 7: *Profiling of GPU_1 and GPU_2*

Table 3(a)

Method / Data	input_128.in	input_4096.in	input_8192.in	input_16384.in
Reference	0.10	583.88	2856.17	10617.90
GPU_1	114.29	148.73	233.38	494.31
GPU_2	116.02	141.12	211.82	460.23

Table 3(b)

Measure / Data	input_128.in	input_4096.in	input_8192.in	input_16384.in
Drop	+ 115920 %	-75.83 %	-92.58 %	-95.67 %
Speedup	0x	4.14x	13.48x	23.07x

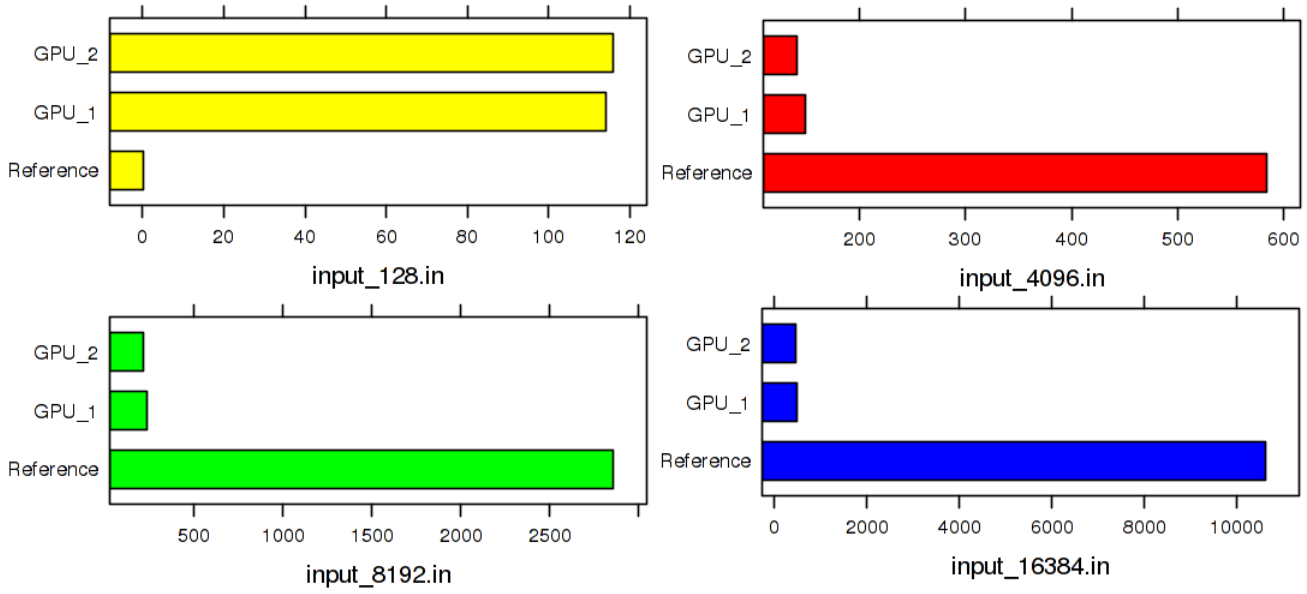

Figure 8: *Comparison of GPU_2 (First Optimization) with Reference amd GPU_1 for different data sets*

4. Conclusion

- While implementing *CUDA* programs, we need to carefully design our kernels. This includes giving careful thought to the selection of grid structure of the thread blocks, and the underlying threads. An efficient mapping of the elements to the threads, enables us to achieve maximum amount of parallelization possible.

- Although shifting from CPU to GPU enables us to achieve parallelism in our program, it has an associated cost, in the form of *CUDA* context initialization, and also for transferring data between host and device. Only beyond a certain threshold of data size, does the performance gains achieved by parallelization overshadow the incurred overheads. So, we must keep this trade-off in mind before jumping to GPU.

5. References

- *https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/*
- *https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/*
- *https://stackoverflow.com/questions/5909485/cuda-device-pointer-manipulation*
- *https://forums.developer.nvidia.com/t/devicequery-error-on-ubuntu-16-04-cuda-8-installation-gtx-1060/51823*