Diagonal Matrix Multiplication (DMM) using CUDA

Shashank Singh (shashanksing@iisc.ac.in)
Indian Institute of Science, Bangalore, India

GPUs are exceptionally good at parallel computing i.e. when a certain series of transformations is to be applied to several pieces of data. In this programming exercise, we attempt to leverage and analyze performance benefits of performing Diagonal Matrix Multiplication on a GPU using the CUDA programming model. We will also explore possible optimizations and tweaks to further improve performance.

1 Introduction to DMM

The DMM program in context takes two matrices A and B of size $N \times N$ and produces a target column matrix C of size $2N-1$, such that

$$\overset{A}{\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}} *^{D} \overset{B}{\begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}} = \overset{C}{\begin{bmatrix} a_{00} * b_{01} \\ a_{01} * b_{11} + a_{10} * b_{00} \\ a_{11} * b_{10} \end{bmatrix}}$$

The C matrix can be computed in a lot of ways and each way has its own memory access pattern which leads to different performances.
The matrix C can be computed in the following way :

$$C[k] = \sum A[i][j] * B[j][N-1-i] \text{ such that } i + j = k$$

The formula shows that each element $A[i][j]$ will be multiplied with an element $B[i][j]$ to produce a partial sum for computing $C[i+j]$. All such partial sums can be added to obtain the matrix C.

We will evaluate the performance of CUDA version of DMM and explore the bottlenecks and the scope of various optimizations to further reduce the run time. We will use the following formula for calculating speedup.

$$\text{Speedup} = \frac{\text{Run time of reference version}}{\text{Run time of version being compared (CUDA)}}$$

2 About the Experiments

All the experiments were run on Google Colaboratory provided by Google. The server runs on *Intel(R) Xeon(R) CPU @ 2.20GHz* with *12 GB* of RAM. The GPU installed on this server was $Tesla\ T4$ GPU with 15079 MB of device memory.

All the run time data were collected by taking difference of timestamps between function relevant function calls and the values shown here are average values of 5 runs. The GPU execution break down was collected by profiling the binary with *nvprof* on different input datasets. The values shown for this are an average of 3 runs using *nvprof* for every input data-set.

3 Reference (Naive) DMM

The pre-provided reference function calculates the output array by going diagonally along matrix A and matrix B. Ther reference version is executed on CPU. We will take the run time of this method as a reference to gauge the performance of the CUDA version.

4 DMM implementation in CUDA

The CUDA implementation of DMM involves following steps :

- Allocating memory for input and output matrices
- Copying data from host memory to device memory
- Launching the CUDA kernel
- Copying the output data from device to host memory
- Freeing up the device memory

4.1 The CUDA kernel

The CUDA kernel is designed in a way that each thread computes one of the $2N-1$ elements of the output matrix. So, we have total $2N-1$ threads which are grouped into $1-dimensional$ thread blocks. The number of threads per block is decided dynamically by taking a minimum of the value of N and 512. The value 512 was experimentally determined and preferred due to better performance on the machine on which experiments were done.

Please refer *Figure 1* for a graphical illustration of how the work is distributed among the threads.

4.2 Overheads Observed (CUDA Related)

There are a few overheads that come into picture when we switch to a GPU like GPU initialization, device memory allocation overhead, copying data to/from device memory etc. The CUDA kernel executions are the only productive operations for us, everything else is an overhead. Some of the overheads are discussed below :

- **CUDA Initialization Overhead -** This is when a newly created process initializes CUDA drivers and runtime. This is also referred to as *Lazy Initialization* where the initialization of CUDA runtime is deferred till a CUDA call requires it to be done. This overhead is a function of OS, CUDA runtime and driver latencies and is out of our control.
- **CUDA Memory Copy (Host to Device) -** This is the overhead for copying input data from RAM to device memory (GPU memory). The host memory in our case is pageable memory which has low Host-To-Device transfer rate. As input size grows, this overhead takes more and more share of the total time.

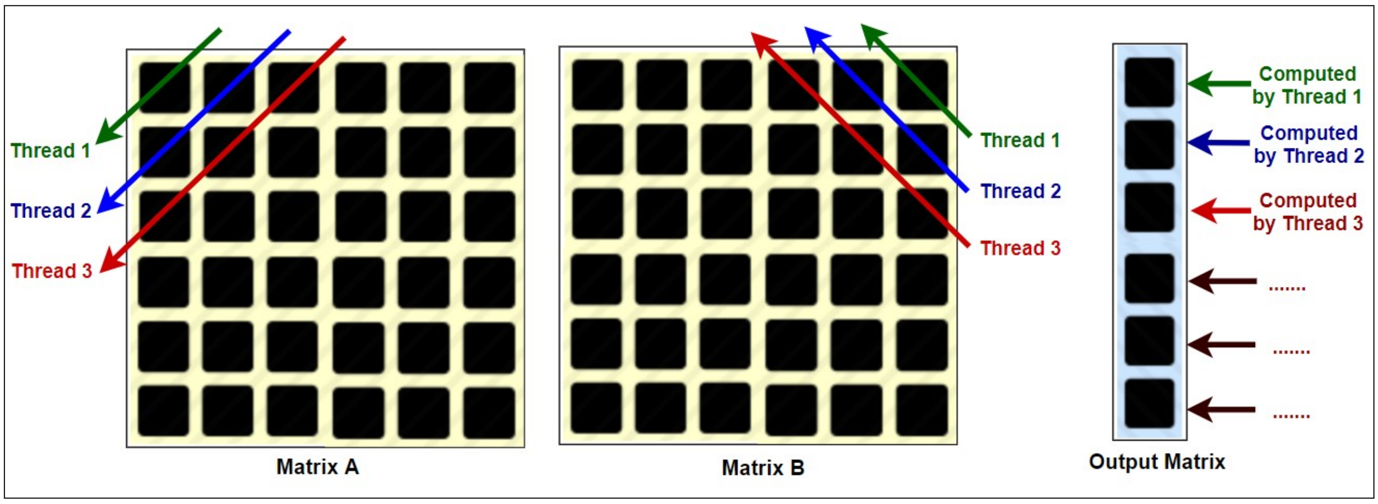

Figure 1: *A diagram illustrating the work distribution and memory access pattern among the GPU threads*

- **CUDA Memory Copy (Device to Host) -** This is the overhead for copying computed data from device memory to host memory (RAM). This is very low in our case as the output data is of size *2N-1* as compared to input of size $2N^2$.

4.3 Optimizations Attempted

The following optimizations were either attempted or considered while implementing the CUDA version of this program :

1. **Optimizations in the kernel implementation -** A few arithmetic and memory access optimizations were tried in the kernel implementation but they don't have any observable effect on the overall runtime due to small execution time of the kernel. The kernel execution constitutes a very small fraction of total execution time and so any kernel code optimization doesn't yield significant results in this case.

2. **Parallel Copying and execution of the CUDA kernel -** Considering that a lot of time is spent in copying data to device memory, we could overlap the copy operation with the kernel execution using streaming or asynchronous copying. But, given the small kernel size where it takes $2-4\%$ of total time, this effort doesn't seem to be worth it as we don't have a lot of execution time to overlap the copying time with.

3. **Breaking down the kernel into 2 smaller kernels for first half and second half of the output matrix -** Since, the kernel contains a different loop logic for first half and the second half of the output matrix, an attempt was made to split the kernel into 2 smaller kernels, each computing the first half and second half separately. The expectation was that this will eliminate the if-else branches and improve the kernel performance. However, on profiling this, the total kernel time became twice instead of an expected improvement. So, this optimization was not included.

4.4 Performance Analysis

The CUDA implementation of the program was analyzed from different points of view. We will evaluate net speedup and runtime improvement compared to the reference implementation. The runtime was calculated by timestamping method. Further, $nvprof$ was used to analyze the GPU activity and break down the GPU time into GPU activities and API calls.

4.4.1 Run Time

Table 1 contrasts the running times and speedup obtained using CUDA implementation with respect to the reference implementation. It can be observed that CUDA implementation gives a significant speedup and the speedup increases drastically with an increase in input size.

	4K	8K	16K	32K
Reference	390.0	1916.7	8714.9	46055.2
CUDA Version	173.5	267.7	631.1	2234.7
Speedup	2.2	7.2	13.8	20.6

Table 1: *Average running time (in ms) comparison and speedup*

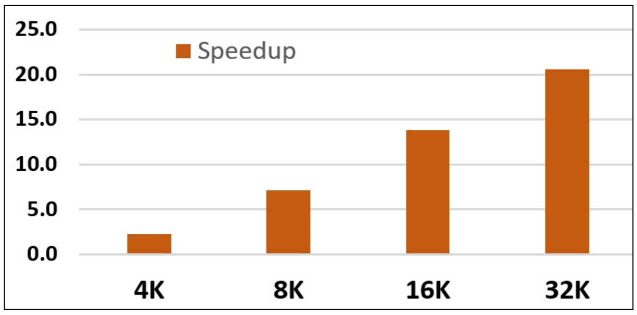

Figure 2: *Speedup trend with increase in input size*

The running time for CUDA version includes CUDA initialization time also. This is a one time cost which can be excluded from an evaluation point of view. The drastic betterment of speedup on increasing input size can be attributed to small kernel execution time and the fixed large initialization overhead (Refer *Tables 2 and 3*).

4.4.2 GPU Execution Time Break Down

The program binary was profiled using the command-line profiler *nvprof*. *Table 2* shows the different GPU activities and API calls which constitute the execution time for the GPU implementation of the program and $Table3$ shows the percentage distribution of these events.

A point to note is that this data was collected by *nvprof*, so it includes profiler overheads which may increase the time of execution and so these values can't be directly compared to the values in *Table 1* where data was collected through timestamps.

	4K	**8K**	**16K**
CUDA Initialization	188.14	188.20	188.12
memcpy HostToDevice	30.21	116.87	465.51
Kernel Execution	5.11	11.29	30.11
memcpy DeviceToHost	0.005	0.009	0.015

Table 2: *Avg time(in ms) in different CUDA events and API Calls*

	4K	**8K**	**16K**
CUDA Initialization	84.19%	59.49%	27.51%
memcpy HostToDevice	13.52%	36.94%	68.08%
Kernel Execution	2.29%	3.57%	4.40%
memcpy DeviceToHost	0.002%	0.003%	0.002%

Table 3: *Percentage distribution of time in different CUDA events and API Calls (Using data from the previous table)*

As evident from data in the tables, a good amount of time is spent in CUDA Initialization, where the CUDA Driver is initialized for every new process when it makes the first CUDA API call. In this, a separate dummy *cudaFree(0)* call was made to separate and identify this initialization time.

However, as the input size increases, the percentage for initialization time goes down and is taken over by *cudaMemcpy* for copying data from RAM to GPU Memory. The CUDA kernel is one of the least time taking activities in GPU. This is because the operation each thread has to perform is computationally very simple.

4.4.3 Overall Result

After considering the speedup data and the performance statistics, we can conclude that the CUDA version of the program works considerably faster than the reference version. The optimal results are seen when the input is large enough that the time lost to overheads is compensated in full by the speedup gained in executing the operation on GPU.

5 Further Scope of Optimizations

There's a lot of scope for a better speedup using GPUs. Some of the points are as follows :

1. **Using pinned (or page locked) memory for input instead of a regular *new* or *malloc* operation -** Our version currently uses regular pageable memory for storing the input which has slower Host-To-Device transfer rates. This is done in *main.cu* file which we are not to modify. If this were to be made as pinned memory by calling *cudaMallocHost()* instead, we could drastically improve the Host-To-Device data transfer overhead. However, repercussions on host programs and OS have to be taken into consideration before allocating pinned memory.
2. **Operating the GPU in Persistence Mode -** From a performance point of view, operating the GPU in persistence mode will keep the GPU in higher power states and the perceived initialization time can be mitigated. Persistence mode affects only power-up time for the device, the GPU context initialization is not affected by this. However, this may have serious impact on power consumption and hardware over-utilization. So, this must only be done if we really need the GPU ready for most of the time.
3. **A more sophisticated operation would lead to a comparatively better speedup -** The break down of GPU execution time reveals that a very small portion of time is being spent in the productive task i.e. computing the output matrix. A lot of time is being spent in copying data. If we were to have a more sophisticated operation, like a geometric transformation or a cryptography operation, we could truly leverage the power of GPU processing. We could also employ optimizations like asynchronous copy and execution.

6 Conclusion

GPUs are extremely good at performing an operation over large amount of data units i.e. they are optimized for throughput rather than latency. So, it is necessary that we employ only those computations on GPU which can justify for the overheads that we incur in doing so. Also, the data and the operation to be performed must have embedded parallelism to be exploited. The CUDA kernel design also plays a vital role in execution time. This exercise was a great learning opportunity and a pragmatic introduction to the world of GPU programming.

7 References

[1] https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

[2] https://stackoverflow.com/questions/12126252/what-does-nvidia-gpu-do-with-device-memory-0x0

[3] https://stackoverflow.com/questions/15166799/any-particular-function-to-initialize-gpu-other-than-the-first-cudamalloc-call

[4] https://stackoverflow.com/questions/45360006/what-does-persistence-mode-actually-do-which-reduces-cuda-startup-time