Cuda Implementation of Checkered Matrix Multiplication(CMM)

Arman Gupta (armangupta@iisc.ac.in),

Indian Institute of Science , Bangalore

1. Abstract

Graphics Processing Unit are specifically designed to accelerate computer graphics workloads and are exceptionally good in parallel processing. In the following experiment, we will implement Checkered Matrix Multiplication using CUDA and analyse its performance on GPU. We will also explore the further optimisation to improve the efficiency of our program.

2. Introduction

Checkered Matrix Multiplication (CMM) is a variant of matrix multiplication. We are given 2 matrices A and B of size N * N where N= 2^k, k is a natural number and k >= 2. We need to produce the output matrix of size (N/2) * N. Following is the explanation of the algorithm:

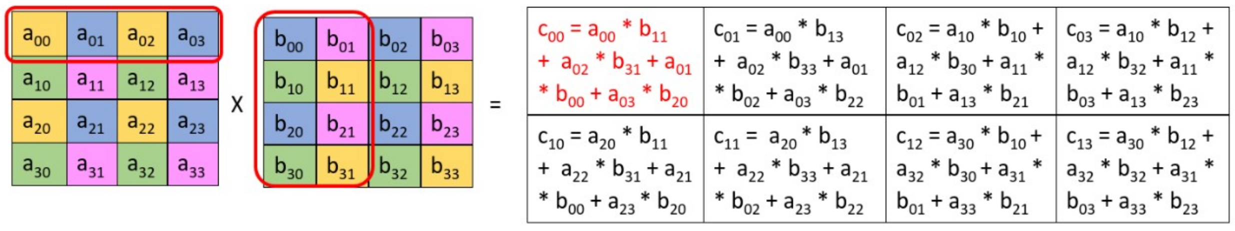
Let us define even and odd positions in matrices A and B. In matrix A, we traverse the elements row-wise, and an element A ij is at even position if j is even and odd otherwise. In matrix B, we traverse the elements column-wise, and an element B ij is even if i is even, odd otherwise.

Let us define columnset in matrix B as a pair of consecutive columns, e.g., 0 th and 1 st columns of B form the 0 th columnset, 2 nd and 3 rd columns of B form the 1 st columnset and so on.

We follow a pattern to multiply the matrices as described below. Note that a row in matrix A is multiplied with a columnset in B.

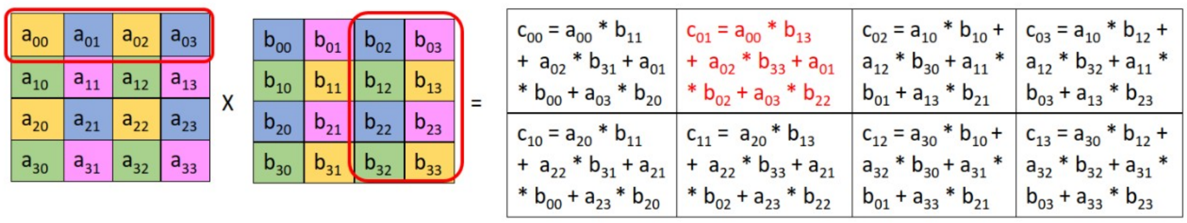
<u>Step 1</u>: Let us take the 0 th row of matrix A (say rowA0) and multiply it with 0 th columnset of matrix B (= {0 th column of B, 1 st column of B}) to get C 00 .

The even elements rowA1 are multiplied with odd elements of 1 st column in the columnset (marked by yellow in figure). And the odd elements of rowA1 are multiplied with even elements of 0 th column in the columnset (marked by blue in figure).


a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
X
b_{00} b_{01} b_{02} b_{03}
b_{10} b_{11} b_{12} b_{13}
b_{20} b_{21} b_{22} b_{23}
b_{30} b_{31} b_{32} b_{33}
=

$c_{00} = a_{00} * b_{11} + a_{02} * b_{31} + a_{01} * b_{00} + a_{03} * b_{20}$	$c_{01} = a_{00} * b_{13} + a_{02} * b_{33} + a_{01} * b_{02} + a_{03} * b_{22}$	$c_{02} = a_{10} * b_{10} + a_{12} * b_{30} + a_{11} * b_{01} + a_{13} * b_{21}$	$c_{03} = a_{10} * b_{12} + a_{12} * b_{32} + a_{11} * b_{03} + a_{13} * b_{23}$
$c_{10} = a_{20} * b_{11} + a_{22} * b_{31} + a_{21} * b_{00} + a_{23} * b_{20}$	$c_{11} = a_{20} * b_{13} + a_{22} * b_{33} + a_{21} * b_{02} + a_{23} * b_{22}$	$c_{12} = a_{30} * b_{10} + a_{32} * b_{30} + a_{31} * b_{01} + a_{33} * b_{21}$	$c_{13} = a_{30} * b_{12} + a_{32} * b_{32} + a_{31} * b_{03} + a_{33} * b_{23}$

<u>Step 2</u>: Next, we shift the columnset, it now consists of column 2 and 3. Let us take 0 th row of matrix A (rowA0) and multiply it the 1 st columnset of matrix B to get C 01 .
Similar to previous step, the even elements rowA0 are multiplied with odd elements of 1 st column in the columnset, and the odd elements of rowA0 are multiplied with even elements of 0 th column in the columnset.

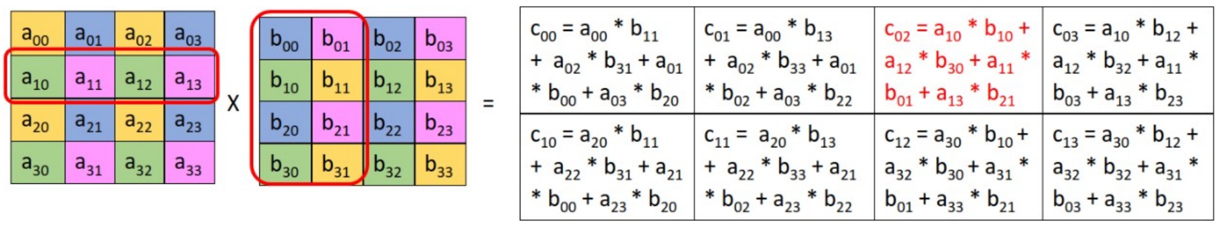


$c_{00} = a_{00} * b_{11} + a_{02} * b_{31} + a_{01} * b_{00} + a_{03} * b_{20}$	$c_{01} = a_{00} * b_{13} + a_{02} * b_{33} + a_{01} * b_{02} + a_{03} * b_{22}$	$c_{02} = a_{10} * b_{10} + a_{12} * b_{30} + a_{11} * b_{01} + a_{13} * b_{21}$	$c_{03} = a_{10} * b_{12} + a_{12} * b_{32} + a_{11} * b_{03} + a_{13} * b_{23}$
$c_{10} = a_{20} * b_{11} + a_{22} * b_{31} + a_{21} * b_{00} + a_{23} * b_{20}$	$c_{11} = a_{20} * b_{13} + a_{22} * b_{33} + a_{21} * b_{02} + a_{23} * b_{22}$	$c_{12} = a_{30} * b_{10} + a_{32} * b_{30} + a_{31} * b_{01} + a_{33} * b_{21}$	$c_{13} = a_{30} * b_{12} + a_{32} * b_{32} + a_{31} * b_{03} + a_{33} * b_{23}$

Note that each row contributes to only half row elements in matrix C, now to obtain the other half elements in this row, we move to the 1 st row in matrix A (say rowA1)

Step 3: We again start from the first columnset i.e., columnset = {0 th column of B, 1 st column of B}. We multiply the 1 st row of matrix A (rowA1) with this columnset.

However, this time, the even elements rowA1 are multiplied with odd elements of 0 th column in the columnset (marked by green in figure). And the odd elements of rowA1 are multiplied with even elements of 1 st column in the columnset (marked by pink in figure).



$c_{00} = a_{00} * b_{11} + a_{02} * b_{31} + a_{01} * b_{00} + a_{03} * b_{20}$	$c_{01} = a_{00} * b_{13} + a_{02} * b_{33} + a_{01} * b_{02} + a_{03} * b_{22}$	$c_{02} = a_{10} * b_{10} + a_{12} * b_{30} + a_{11} * b_{01} + a_{13} * b_{21}$	$c_{03} = a_{10} * b_{12} + a_{12} * b_{32} + a_{11} * b_{03} + a_{13} * b_{23}$
$c_{10} = a_{20} * b_{11} + a_{22} * b_{31} + a_{21} * b_{00} + a_{23} * b_{20}$	$c_{11} = a_{20} * b_{13} + a_{22} * b_{33} + a_{21} * b_{02} + a_{23} * b_{22}$	$c_{12} = a_{30} * b_{10} + a_{32} * b_{30} + a_{31} * b_{01} + a_{33} * b_{21}$	$c_{13} = a_{30} * b_{12} + a_{32} * b_{32} + a_{31} * b_{03} + a_{33} * b_{23}$

Please note that, every row odd numbered row (o-based indexing), follows the same rule as A1.
Whereas, every even numbered row, follows the same rule as rowA0 mentioned previously.
Also, the product of the i th row of matrix A and the j th column of matrix B will be placed in the cell C[floor(i/2)][j] if i is even and in C[floor(i/2)][j+N/2] if i is odd.

3. Processor , Tools and Techniques

The codes are run on Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz having 4 physical cores with 2 threads each and the NVIDIA GeForce GTX 1650 graphics processor with total memory of 4096 MB , 896 Cuda Cores , 128-bit memory interface, 8 GT/s Maximum PCIe Link Speed and Maximum PCIe Link Speed of 16 (PCIe 3.0).
The performance analysis is done using the nvprof tool. Each of the code are run separately 5 times for each Input sets and the data shown here in this report is average of 5 runs.

4. Reference CMM

We are considering the unoptimised implementation of the CMM provided to us as our reference code and use it to compare the execution time of cuda implementation of the code.

5. Cuda Implementation of CMM

Before implementing the cuda program, we must first understand the overheads that come into existence when we switch from CPU to GPU based program. Following below are the few overheads:

1. Cuda Memory Allocation Overhead (Cuda Malloc): Cuda Memory Allocation is the mandatory step in the implementation of the Cuda version of the program and is inevitable.
2. Data copy from Host to Device (Cuda memcpy HtoD): This overhead arises from transferring the data from RAM to the GPU device memory.
3. Data copy from Device to Host (Cuda memcpy DtoH): This overhead arises from transferring the data from GPU device memory to RAM.
The important task is performing in kernel only hence apart of kernel execution ,rest of the activities are overheads.

5.1 Cuda Kernel Implementation

Cuda Kernel is implemented in such a way that one thread is calculating one element of the output matrix. Thus if the size of output matrix is (N/2) * N , there will be (N/2) * N threads. The output matrix (indicating the threads) is divided into square blocks and the size of the block is decided dynamically. If the input size is less than equal to 32 then the block size is N/2 * N/2 else the block size is 32 *32. This block size is decided based on the experiments done using various block sizes. The row and the columns set corresponding to each element in output matrix is then calculated using the following

```
rowA = blockIdx.y*blockDim.y+threadIdx.y;
colSetB = (blockIdx.x*blockDim.x+threadIdx.x)*2;
```

where blockDim represent the block dimension denoted by (blockDim.x , blockDim.y), blockIdx represents the block Id in the 2-D grid denoted by (blockIdx.x , blockIdx.y) and inside each block, each thread has its own thread id denoted by (threadIdx.x , threadIdx.y).

5.2 Performance Analysis

Though the program works fine for input set of size 128 but it fails to bring out the efficiency that we have gained switching from CPU to GPU due to cuda overheads mentioned above. Thus we will also analyse the input set of size 1024 , 2048 and 4096 to bring out the efficiency of Cuda Program w.r.t. reference program. Note that the execution time of reference program is taken from last experiment (Optimisation of the Single threaded and multithreaded CMM) and the execution time of the cuda implementation is calculated using Cuda Event API i.e. cudaEvent_t , cudaEventRecord() and cudaEventCreate().

Execution Time :

	128	1k	2K	4K
Reference	3.56 ms	2058.27 ms	99280.84 ms	954188.25 ms
Cuda Version	0.48 ms	68.63 ms	496.56 ms	3460.75 ms

For the sake of completeness we will also measure time in various GPU activities and API Calls for input size 128 ,1k ,2k and 4k

Break Down of time involve in GPU Activities :

	128		1K		2K		4K	
GPU Activities	Time	Time %	Time	Time %	Time	Time %	Time	Time %
Kernel	198.65us	94.10	58.013ms	97.38	399.86ms	97.84	2.92874s	98.76
CUDA memcpy HtoD	9.0570us	4.29	1.3857ms	2.33	6.2098ms	1.52	24.457ms	0.82
CUDA memcpy DtoH	3.3920us	1.61	176.58us	0.30	2.6369ms	0.65	12.443ms	0.42

Break Down of time involve in API Calls :

	128		1K		2K		4K	
API Calls	Time	Time%	Time	Time %	Time	Time%	Time	Time%
cudaMalloc	92.740 ms	99.31	87.123ms	58.72	97.577ms	19.21	101.26 ms	3.30
cudaMemcpy	269.24 us	0.29	60.630ms	40.86	409.81ms	80.67	2.96667s	96.68

In the table, we have mentioned those API Calls which are taking significant amount of execution time. As we can observe from above tables that overheads is taking considerable time. For example, In case of input set of size 128 these overheads dominates the total time taken by the input (the kernel is running for 198.65 microseconds while cudaMalloc api calls are taking 92.740ms).

5.3 Effective Bandwidth

We calculate effective bandwidth by timing specific program activities and by knowing how our program accesses data. We use the following equation.

$$BW\text{Effective} = (R\text{B}+W\text{B}) / (t * 10\hat{}9)$$

Here,*BW*Effective is the effective bandwidth in units of GB/s, *R*B is the number of bytes read per kernel, *W*B is the number of bytes written per kernel, and *t* is the elapsed time given in seconds calculated using CUDA memcpy HtoD and CUDA memcpy DtoH time.

Input Size	128	1K	2K	4K
Effective Bandwidth (GB/s)	13.16	6.712	4.74	4.55

6. Further Optimizations

1. Using Pinned Memory : Host (CPU) data allocations are pageable by default which our current program is using. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or “pinned”, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory.

2. Using Register Caching : This optimization technique develops a virtual caching layer for threads in a single warp. This software abstraction, which is implemented on the top of the NVIDIA GPU shuffle primitive, helps optimize kernels that use shared memory to cache thread inputs. When kernel is transformed by applying this optimization, the data ends up being distributed across registers in the threads of each wrap, and shared memory accesses are replaced with accesses to registers in other threads by using shuffle, thereby enabling significant performance benefits.

3. Using Unified Memory with Asynchronous Prefetching and Memory Advice: This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. But Unified memory may incur the CPU as well as GPU page fault in case required block is not present in the memory. In order to reduce the page faults in case of unified memory, we can inform Cuda driver to prefetch the necessary data structure in advance using cudaMemPrefetchAsync(). We can further improve the program performance by providing the memory advices like read mostly in case our host array is mostly used for reading purpose (like Matrix A and Matrix B in our case) by calling cudaMemAdvise() function.

7. Conclusion

GPU provides us exceptionally good performance in parallel processing but it comes with overheads too. When the input size is small, these overheads dominate the execution time. Hence, the input size must be large enough to shadow these overheads, i.e. the overheads must not dominates the execution time ,then only we can notice the true efficiency of using GPU for our program.

8. Reference

[1] Nvidia Blog https://developer.nvidia.com/
[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#env-vars
[3] https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/
[4] https://forums.developer.nvidia.com/t/how-to-calculate-the-theoretical-memory-bandwidth
[5] https://developer.nvidia.com/blog/register-cache-warp-cuda/

This experiment does not include L1 cache related performance counter due to the issue in the gpgpu-sim itself. You can check here https://github.com/gpgpu-sim/gpgpu-sim_distribution/issues/211