Programming with CUDA on GPUs – Matrix Multiplication

Aim:

To perform the basic (yet a highly time complex) matrix multiplication task, on GPU using CUDA, and compare the amount of performance boost that CUDA provides compared to Sequential & OpenMP versions on CPU.

Software Used:

GCC C compiler (version 11.2.1), CUDA, OpenMP.

CUDA Implementation:

This problem has been solved with 2 different approaches (namely Version-1 & Version-2), where both are expected to give the same performance for matrix of sizes up to 1024. But for matrices of sizes greater than 1024, Version-2 is expected to perform significantly faster than Version-1. The method followed in both the versions are delineated below:

Version-1:

In this method, each block is exclusively assigned to perform all the operation in a row of the output matrix. This means the max. 1024 threads which can be present inside a block is responsible for completing all the calculation required for that row.

This is done just by using the x-dimension of a grid and x-dimension of a block. An x-dimension of the grid is assigned to a unique row of output matrix, and the x-dimension of the thread is assigned to 1 cell in the row, or more than 1 cell in the row if nsize > 1024 (Since the maximum number of threads that can be present inside of a block, inclusive of all dimensions is limited to 1024).

Since the x-dimension of a grid can contain up to 2^{31}-1 blocks, we can run this algorithm to a Matrix Size of up to the size of 2^{31}-1 (although the computation for the maximum size would take several thousand years with today's GPUs due to the O(n^3) time complexity). This disadvantage of this method however is that, since only 1024 threads can run inside a block, and we are assigning an entire row for a block, this would cause each thread to compute for more than 1 cell in the output matrix, if nsize > 1024, and the number of cells each threads has to computer is given by $ceiling(nsize\ /\ 1024)$, which is proportional to the total execution time of this algorithm.

Version-2:

In this method, the disadvantage faced by Version-1 is addressed. Here instead of assigning a block to an entire row, we ensure that a thread in a block works on & only on 1 cell of the output matrix during execution. So, in order to address the case of nsize > 1024, we instead increase the number of blocks such that each block has to execute no more than 1024 cells in the output matrix.

This is done by additionally using the y-dimension of a grid to account for the cells in each row with indexes greater than 1024. Here the x-dimension of a grid is assigned to a unique row, and an x-dimension combined with all of it's y-dimensions takes care of all the cells in the row it is assigned to.

Now since we have assigned exactly 1 cell per thread, and a thread has to execute no more than 1 cell, this will provide significant speedups for nsize > 1024. For example, for nsize = 2048 we can expect a speedup of close to 2 times (on any recent GPU) since, the workload per thread has significantly reduced, further promoting parallelism. And the maximum nsize supported by this method is given by $65535 * 1024 \approx 2^{26}$, since the maximum y-dimension of a grid is 65535 & the maximum number of threads per block is 1024.

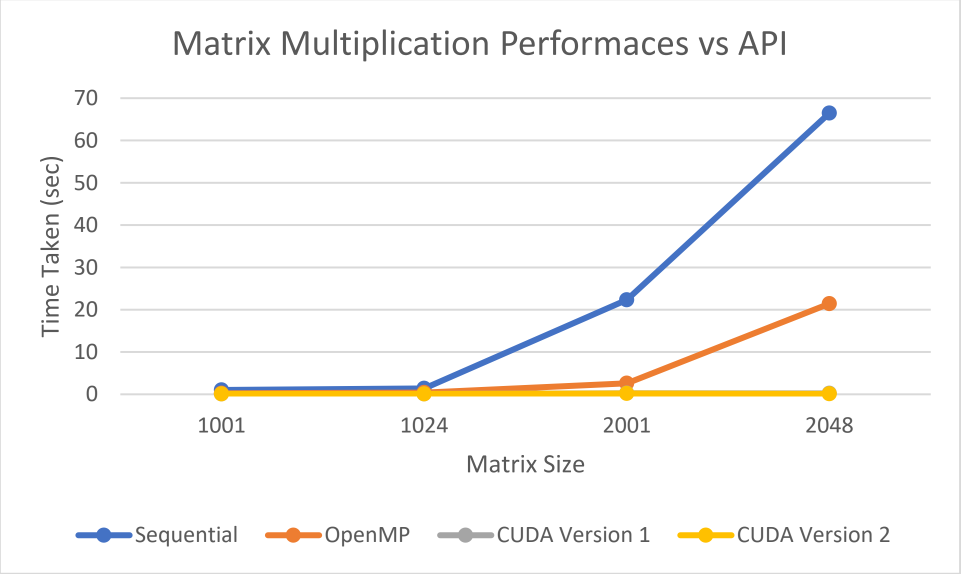
Performance Comparisons (Seq vs OpenMP vs CUDA) on gpu-node1:

In order to demonstrate the effectiveness of our CUDA program, we have run the same Matrix Multiplication Algorithm Sequentially, with OpenMP, with Version 1 & 2 of CUDA programs with matrix sizes of 1024, 2048, 1001 & 2001, and cross compare the time taken by each of the versions to run.

It is to be noted that the gpu-node1 runs with a NVIDIA GTX 1080, which is a powerful GPU containing 2560 CUDA Cores, with 28 Streaming Multiprocessors with a maximum of 128 CUDA Cores each. And the CPU it contains is an Intel i7-7700K, with 4 unlocked Cores & 8 threads.

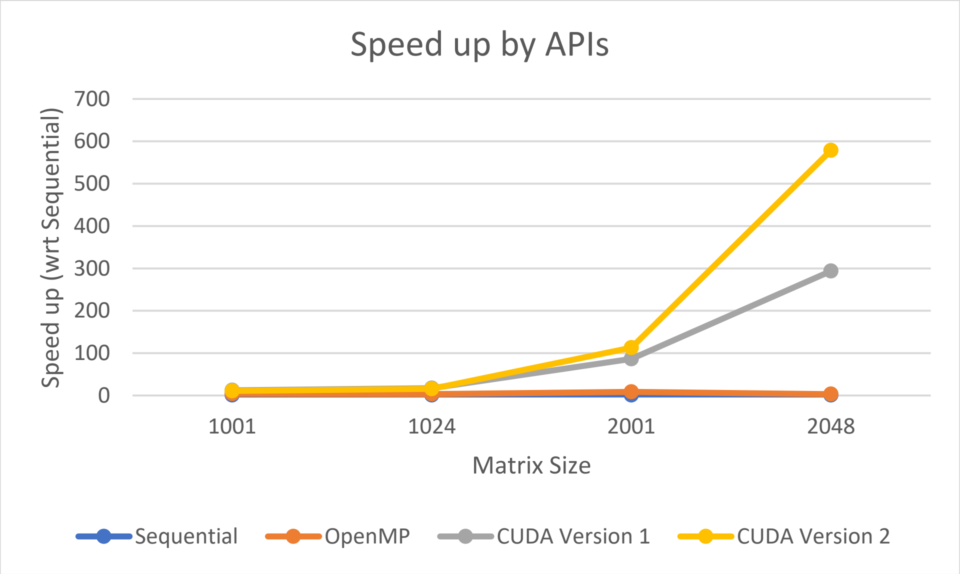
The performance comparison is as follows (all time units are in seconds):

API / Matrix Size	1001	1024	2001	2048
Sequential	1.02894	1.39945	22.3342	66.4837
OpenMP	0.247864	0.411193	2.60929	21.4269
CUDA Version 1	0.08397	0.081569	0.258896	0.22632
CUDA Version 2	0.096222	0.089706	0.198037	0.114906



The Speedup Provided by each version with respect to the Sequential Version is as follows:

API / Matrix Size	1001	1024	2001	2048
Sequential	1	1	1	1
OpenMP	4.151228	3.40339	8.559493	3.102815
CUDA Version 1	12.25363	17.15666	86.26707	293.7597
CUDA Version 2	10.69345	15.60041	112.7779	578.5921



Hence it can be observed from the above charts that, OpenMP is about 5 to 10 times faster than the Sequential Version, and CUDA outperforms by an outstanding margin, by being 10 to 600 times faster than the Sequential Version depending on the Matrix Size. It can also be noted that, the Version 1 of CUDA program is slightly faster for matrices of lower sizes, while the Version-2 CUDA program is much faster than Version-1 for matrix sizes greater than 1024, which is exactly the expected behaviour from the Algorithms.

Result:

Hence the C program for Matrix Multiplication Elimination is successfully implemented with CUDA with 2 different versions of CUDA. All of their performances are compared across different Matrix Sizes and the huge improvement in performance when using CUDA (especially with increasing Matrix Sizes) is clearly displayed, with respect to the Sequential and OpenMP versions. And the final results have been tabulated and plotted for readability.