CS406 HW03 Report

Yiğit Aras Tunalı

May 2019

1 Problem Statement

The problem to be implemented and parallelized is the permanent of a matrix. The algorithm we followed was the one proposed by Ryser but for the sake of better complexity, the Nijenhuis and Wilf version was implemented. Since this task was already accomplished in the HW01, with CPU parallelization, this time I'll be picking up from where the CPU implementation was left and continue from there. Since already the approach of pre-calculating the start X array's of each thread and continuing from the next index for each thread was the most efficient parallelization technique I could have come up with last time, this time I'll be starting with this approach as well. Below are the 32 thread CPU ,results for the same matrices that will be used in the GPU implementation;

Matrix_Size	**25x25**	**36x36**	**38x38**	40x40
Execution Time	0.0989886s	70.4074s	313.803s	1106.06s

GTX TITAN X Engine Specs:	**GTX 980 Engine Specs:**
3072 CUDA Cores	2048 CUDA Cores
1000 Base Clock (MHz)	1126 Base Clock (MHz)
1075 Boost Clock (MHz)	1216 Boost Clock (MHz)
192 Texture Fill Rate (GigaTexels/sec)	144 Texture Fill Rate (GigaTexels/sec)
GTX TITAN X Memory Specs:	**GTX 980 Memory Specs:**
7.0 Gbps Memory Clock	7.0 Gbps Memory Clock
12 GB Standard Memory Config	4 GB Standard Memory Config
GDDR5 Memory Interface	GDDR5 Memory Interface
384-bit Memory Interface Width	256-bit Memory Interface Width
336.5 Memory Bandwidth (GB/sec)	224 Memory Bandwidth (GB/sec)

Above are the specs of the GPU's in Nebula. For the implementation of the algorithm the Titan X GPU have been used and the algorithm has been developed for only Titan X for the sole reason of it being better than the other.

2 Naive Parallel Algorithm

First approach that came to mind was to directly move the CPU algorithm to the GPU. This brought it's own problems native to the GPU with it. The main problem here was that, using $malloc()$ to allocate one thread's X array did not work as intended and caused all kinds of errors. The next logical step to do was to allocate the whole array of $(N * THREAD\_NUM * BLOCKS)$ size in CPU and then with the usage of $cudaMemcpy()$ function, move it to the global memory of the GPU. For this method and simplification of the memory access, the X arrays for each thread were concatenated back to back before being sent to the GPU, likewise the matrix M was flattened, column major-wise, and also sent to the global memory of the GPU. The representation of the X array was as follows;

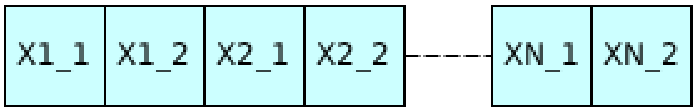

Figure 1: Structure of the X array

An example if there were N arrays with each having 2 elements is shown above, indexed as X1 for 1st array X1_1 for the 1st element of the 1st array. Unfortunately this method yielded abysmal results. The slow global memory access coupled with disadvantageous structuring of X array, the advantageous one will be discussed in the following sections, caused extremely long execution times. For the test runs of the algorithm the thread number for the kernel was tried with 32*1024,64*512, and 128*256, but all yielded similar results.

Matrix_Size	**25x25**	**36x36**	**38x38**	40x40
Execution Time	0.720018s	910s	(TOO LONG)	(TOO LONG)

3 Coalesced Memory Access Approach

Since the previously mentioned approach didn't work, after consulting with the Professor and getting suggested to try a more coalesced memory access approach. To achieve this goal the structure of the X array had to change. All the first elements of the X arrays' would follow all the second and so on until the last N'th element. Below diagram illustrates the structure of the new X array;

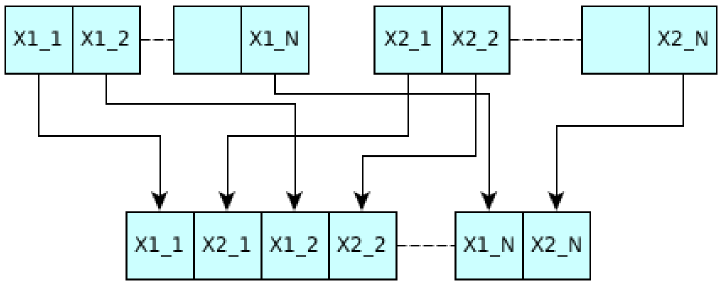

Figure 2: Updated Structure of the X array

The results from this approach was faster than the previous but still considerably slow when tried on $36x36$ or $40x40$ matrices.

Matrix_Size	**25x25**	**36x36**	**38x38**	40x40
Execution Time	0.0277034s	59.5024s	340.408s	(TOO LONG)

4 Shared Memory Approach

After some digging through CUDA material covered in class and some online sources, I have found out that shared memory access was much faster than the global memory access. Upon usage of device properties functions supplied by CUDA, I have found out that each block had 49KB shared memory. With a simple formula of $(49KB/8*N*THREADS)$ where N is the dimension of the matrix and THREADS being the number of threads per block (8 is the amount of bytes per float). This formula yields an approximation to how much memory would be necessary to give each thread in a block their own X array in the shared memory space they have. With this approach there was a considerable speed up (around 1.5 - 2 times) from the previous approach.

Matrix_Size	**25x25**	**36x36**	**38x38**	40x40
Execution Time	0.0165087s	27.4688s	92.0109s	762.536s

5 Shared and Coalesced Memory Access Approach

As a final approach, I have tried the same access method I tried in Figure 2. The speed-up was minimal and sometimes even performed slower. So as the final form of my code, I will be submitting the shared memory approach alone.

6 Using FLOAT and Shared Memory

Using FLOAT instead of DOUBLE for the calculations increase the speed of the algorithm but in some cases reduces the precision of the result,also lowered size for FLOAT results in utilizing much more threads (because larger X array can be fitted inside the shared memory region); Following are the results obtained by this approach;

Matrix_Size	**25x25**	**36x36**	**38x38**	40x40
Execution Time	0.00851195s	16.1557	67.1639	276.068

7 How to Run

Please use "nvcc permanent_hw3.cu -O3 -o perm -arch=sm_61 -Xcompiler -fopenmp -Xcompiler -O3" to compile the code and run as; (exec input_file machine_no). The code is optimized for Titan X GPU, so please use either 0 or 3 as the machine no (Shared memory size changes from machine to machine).