CS406 - Matrix Permanent on CUDA

Baris Sevilmis

May 25, 2019

1 Introduction

Purpose of this homework is to implement Matrix Permanent Calculation Code as efficient as possible. Nijenhuis and Wilf Algorithm is used for efficiency, and it is be parallelized using various methods that will be explained in further sections of this report. CUDA framework is utilized for parallelization.

Section 2 provides information about Nijenhuis and Wilf Algorithm for Matrix Permanent calculation. Part 3 ensures necessary details about different CUDA implementations for Nijenhuis and Wilf Algorithm. Chapter 4 demonstrates results for CUDA implementations, including execution times and speed up values, besides, results of CUDA implementations will be compared with OpenMP CPU parallel implementation from Report 1. Lastly, Section 5 provides information about possible future approaches for better speed-up.

2 Nijenhuis and Wilf Algorithm for Matrix Permanent

Nijenhuis and Wilf Algorithm is an alternative method to naive matrix permanent calculation. Algorithm 1 provides pseudocode of the sequential algorithm. The parallel version of the algorithm will be provided in the next section. For the sake of convenience, it would be reasonable to explain the algorithm shortly. Initially, x and p values are initialized by the following operations in Algorithm 1. Consecutively within a loop of $1 \rightarrow 2^{N-1} - 1$, gray code of current and previous iterations are determined to find changing bits and changing bit is used for updating x. Product of x values is used for updating p, which at the end determines the final result of matrix permanent. From the parallel perspective, some issues require handling, such as breaking loop dependencies. Next section of the report will be including all the necessary details for parallel implementation issues and their solutions.

Algorithm 1 Nijenhuis and Wilf Algorithm

```
1:  procedure MATRIX PERMANENT(matrix M)
2:      N ← size(M)
3:      for i ← [0 → N − 1] do
4:          for j ← [0 → N − 1] do
5:              sumCol ← M[i][j]
6:          lastCol ← M[i][N-1]
7:          x[i] ← lastCol-sumCol/2
8:          p ← p*x[i]
9:      for m ← [1 → 2^(N−1) − 1] do
10:         y ← GrayCode(m)
11:         yp ← GrayCode(m-1)
12:         z ← ChangingBit(y,yp)
13:         s ← FindSign(y,z)
14:         s ← ProdSign(m)
15:         for n ← [0 → N − 1] do
16:             x[n] ← lastCol-sumCol/2
17:             p ← p*x[n]
        return result ← 4*(n%2)-2 ∗ p
```

3 Parallel Nijenhuis and Wilf Algorithm on CUDA

Parallelizing Nijenhuis and Wilf Algorithm cannot be performed directly as there are some crucial points to consider. It is evident that parallelization will be mainly performed over the loop $1 \rightarrow 2^{N-1} - 1$, since N is not vast on its own. The main problem for the parallelization is the dependency of x and p values over the main iteration space. For parallelization, understanding gray code plays a crucial role. Gray code values are the same size as the main loop, which is $1 \rightarrow 2^{N-1} - 1$. However, their ordering is based on the bit differences between consecutive values. Each consecutive integer has specifically 1-bit difference. The following formulation can be used to understand this concept:

Iteration	Gray Code Value	Bitwise Representation
1	1	00000001
2	3	00000011
3	2	00000010
4	6	00000110
5	7	00000111
⋮	⋮	⋮

Table 1: Greycode Order

Index of changing bits is used to pick specific columns of the input matrix M. If the sequence of changing bits is examined, it will become clear that changes over the original x values are temporary. In other words, once a specific column z is added, it is subtracted after 2^z iterations. Besides, without finding changing bits, we can directly compute set bit indexes of specific gray code order and add corresponding indexes of the matrix M to the original x values every iteration. This way, iteration space can be traversed independently; however, finding set bit indexes of gray codes, at each iteration would be computational wise very inefficient. Therefore, this pattern should be used more carefully to obtain better timing results.

To increase efficiency, going over these set bits of some gray code values would be enough. Every thread needs to have an initial x value, which will be computed by going over their initial gray code values set bits only once. As corresponding set bit indexes of gray code within matrix M will be added over original x values, every thread will have their initial x values. One important thing is to make x private for each thread so that race condition over x values will be resolved. Another essential detail is to schedule threads so that their updates over x values will be consecutive. This way, every thread will update their x values independently from each other with correct results and update the p value by with exact x products independently. To keep x arrays as separate arrays, there exist several methods including Shared Memory Usage and varying structure of the x array. These methods will be discussed in later sections of this report. As we are working on GPU, the most straightforward way to sum all distinct p values is by using **atomicAdd**. This functionality will only be used after thread finishes working on its chunk of values. Algorithm 2 provides pseudocode for Matrix Permanent on CUDA.

Determining $BLOCK\_SIZE$ & $THREAD\_SIZE$ is signicant, however they are changed on different approaches such as shared memory approach. These specific values are mentioned in next section.

Algorithm 2 CUDA Kernel - Nijenhuis and Wilf Algorithm

procedure CUDA MATRIX PERMANENT($M, p_{init}, X_{init}, CHUNK, N$)
    tid ← *blockIdx.x * blockDim.x + threadIdx.x*
    xCuda ← X_{init}
    pCuda ← *0*
    m ← *CHUNK*tid*
    y ← *GrayCode(m)*
    sb[:] ← *FindSetBitIndexes(y)*
    for $n \in xCuda$ **do**
        *xCuda[tid*N + n]* ← *M[n*N + sb[:]]*
    for $index \in [CHUNK * tid \rightarrow CHUNK * (tid + 1)$ **do**
        y ← *GrayCode(index)*
        yp ← *GrayCode(index-1)*
        z ← *ChangingBit(y,yp)*
        s ← *FindSign(y,z)*
        prodSign ← *ProdSign(m)*
        for $n \in xCuda$ **do**
            *xCuda[tid*N + n]* ← *s*M[n*N + sb[:]]*
            pCuda ← *x[n]*prodSign*
    atomicAdd($p_{init}, pCuda$)

3.1 Naive CUDA Approach

Although, there exist different methods to pass 2-dimensional array structures into CUDA kernel, using 1-dimensional array structure instead of 2-dimensional array structures happen to be much easier to utilize. Therefore, transpose of our M-matrix, containing input data, is converted into a 1-dimensional array. This way, instead of NXN matrix structure, we obtain N^2 size array consisting of input integer values. For this reason, accessing specific indices is done by $[i * N + j]$ instead of $[i][j]$, given $i, j \in N$.

In the naive approach, kernel algorithm follows the same set of instructions as in CPU implementation. Every thread operates on the gray code values on their chunks, where they update their own x array. Their x array is initialized beforehand on host function, in which x array has the total size of $Thread\_Amount * N * sizeof(double)$. In CUDA kernel, every thread picks their own x array by $(threadIdx.x + blockDim.x * blockIdx.x) * N$. Figure 1 demonstrates x array on the CUDA kernel in case of the naive approach. In other words, Every thread has its chunks of size N in the following manner. Unfortunately, this kind of memory access on x array lacks bandwidth utilization. Therefore, Section 3.2 explains a better approach resulting in a much better speed-up.

FindingSetBit() & *ChangingBit*() are done by built in CUDA intrinsics such as *__ffsll()* or *__popcll*() in terms of efficiency. On the other hand, bitwise operations are used for *GrayCode*() & *FindSign*() operations.

BLOCK_SIZE for naive CUDA approach is chosen as $1024 * 32$ and *THREAD_SIZE* is chosen as 1024.

Figure 1: Naive x array on CUDA kernel($Size = Thread\_Amount * N$)

3.2 Memory Coalescing

To achieve a decent speed-up in CUDA, memory access utilization has the utmost importance. For this reason, the previous section fails in ensuring any speed-up.

Memory accesses on the x array are changed, in which x elements of a specific thread are spread across the whole x array. To access successive index, a thread has to increase its own index by $(threadIdx.x + blockDim.x * blockIdx.x) * Thread\_Amount$. Each threads corresponding x elements are in consecutive memory addresses. Since threads in a specific warp are executing the same instruction at the same time, threads will be accessing consecutive memory blocks, resulting in higher bandwidth utilization. Figure 2 demonstrates x array layout. As a consequence, Memory Coalescing is achieved up to a certain degree and produce much lower execution times and much higher speedup values.

$BLOCK\_SIZE$ for Memory Coalescing CUDA approach is chosen as $1024 * 32$ and $THREAD\_SIZE$ is chosen as 1024 If it was possible to supply more threads within a block, this approach would be much more efficient.

Figure 2: Coalesced x array on CUDA kernel($Size = Thread\_Amount * N$)

3.3 Shared Memory

Using shared memory instead of global memory has its advantages and disadvantages. Unfortunately, maximum shared memory size supported by the testing environment supports only 48KB of memory. Considering every block uses only 48KB of shared memory, thread size in a specific block does need to decrease to a certain level for efficient utilization of provided shared memory.

As for the advantages, accessing shared memory occurs at very high speeds compared to global memory. Therefore, with much more lower thread amount in a specific block, Shared Memory Utilization provides better speedups compared to Memory Coalescing technique. For this approach, Memory Bank size is set 8 bytes for double precision data values to avoid bank conflicts in shared memory.

$BLOCK\_SIZE$ for Shared Memory CUDA approach is chosen as $1024 * 64$ and $THREAD\_SIZE$ is chosen as 128 for double precision and 256 for single precision. Code 1 demonstrates application method of shared memory within CUDA kernel. Given our parallel platforms, we are only allowed to allocate 1024 threads within a single block, therefore shared memory approach improves performance on high levels.

```
1    long long int ind = ((threadIdx.x) * N);
2    extern __shared__ double x[];
3    double* temp_xCuda = x;
4    double* my_xCuda = temp_xCuda + ind;
5    for(int i = 0; i < N; i++)
6        my_xCuda[i] = xCuda[i];
7    __syncthreads();
```

Code 1: Dynamic Shared Memory Usage

3.4 Shared Memory & Memory Coalescing

As Shared Memory and Memory Coalescing approaches derive in decent speed-ups, a hybrid approach containing both of these approaches, namely shared memory but with consecutive memory block accesses is attempted. For this approach, Memory Bank size is set 8 bytes for double precision data values to avoid bank conflicts in shared memory.

$BLOCK\_SIZE$ for Hybrid CUDA approach is chosen as $1024 * 64$ and $THREAD\_SIZE$ is chosen as 128 for double precision and 256 for single precision.

4 Results

C/C++ with CUDA framework is used as the programming language for parallel implementation as well as the sequential implementation. Testing of algorithms were done in GTX Titan-X with the following specs:

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

Naive Approach on large matrices takes a lot of time to execute, therefore is only added to Results section to demonstrate difference of execution times. Unfortunately, it result of Naive approach for $40x40$ input matrix can not be seen due to its high complexity and requirement of ver high amount of time. Table 2 depicts all the execution time results of previously mentioned approaches, as well as the results to check correctness of codes.

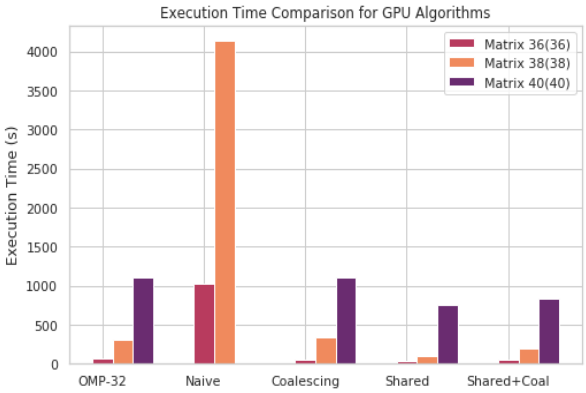
Makefile contains all the necessary informations to run codes, in any case following command can be used to compile and execute given codes:

- *nvcc -o < executable > < code > -O3 -Xcompiler -fopenmp -Xcompiler -O3*
- **./**< *executable* > < *input* > < *device_number* >

Matrix Permanent on CPU and GPU (Execution Times)									Results	
Input	OMP-32 Th.	Naive	Mem. Coal.		Shared Mem.		Hybrid			
			Double	Float	Double	Float	Double	Float	Double	Float
25x25	0.0989s	0.293s	0.31034s	0.2445s	0.099s	0.0996s	0.1324s	0.2228s	1.49e+17	1.49e+17
36x36	70.407s	1024.9s	58.8175s	34.015s	30.4829s	16.2776s	47.056s	33.637s	4.35e+15	4.32e+15
38x38	313.80s	4130.1s	343.02s	144.52s	105.628s	67.4062s	199.812s	142.28s	3.92e+16	3.92e+16
40x40	1106.06s	-	1106.16s	661.04s	747.079s	276.681	840.974s	616.59s	8.40e+11	7.73e+11

Table 2: Execution Time(s) Results

Figure 3 depicts comparison of execution times, in which left subplot demonstrate execution time comparisons of mentioned approaches starting from input matrix of size $36x36$ to $40x40$. On the other hand, right subplot shows comparison of these algorithms on input matrix of size $25x25$. Figure 4 illustrates changes in results for the set of efficient approaches due to single precision.



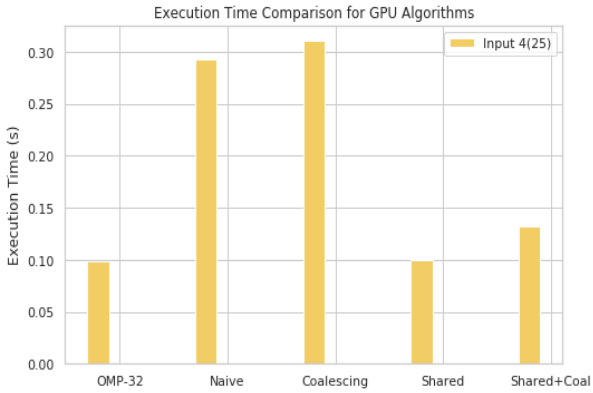
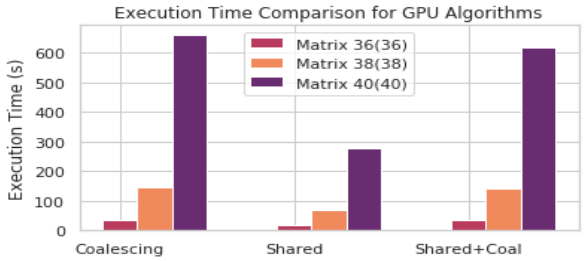


Figure 3: Execution Time Comparison (Double)



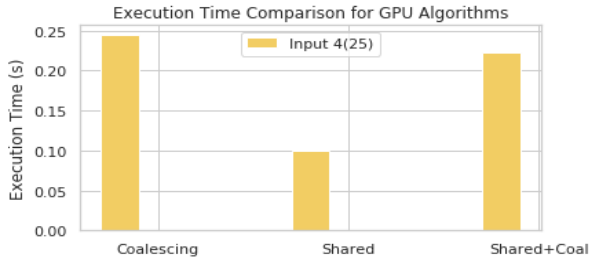


Figure 4: Execution Time Comparison (Float)

5 Possible Future Work

As a possible improvement, $BLOCKS$ and $THREADS$ can be assigned in a way that every block deals with a specific x where each thread within a block will be responsible for a particular x index. This way hot spot of computation will be decomposed. Each block will consist of only N threads, however giving more threads may make up the difference in speed-up, additionally considering shared memory usage.

Unfortunately, operations such as $FindChangingBit()$ and $GreyCode()$ will need to be computed by each thread within a block instead of being calculated only once. Besides, providing the kernel with large amounts of blocks may cause excessive overhead.

In any case, a prototype of this approach is almost completed and will be attempted on the same set of provided inputs shortly.