CS406 HW3

Adnan Kaan Ekiz

May 2019

Contents

1 Introduction

For the Homework 3 of the Parallel Computing Course, main idea was to efficiently create a matrix permanent calculation code. During the process of implementing the code, it is important to recognize and briefly mention some parts which seems crucial. For that reason, homework 3 report will present some idea about the algorithm and how it is implemented for the GPU and CPU run, some of the tricks and steps used in the implementation phase, and time result of impovements with an overall analysis in addition to some problems faced during the implementation.

2 Algorithm

To achieve the ultimate goal of low complexity, for this homework, I have decided to use the matrix permanent algorithm introduced by **Nijenhuis and Wilf**. For the sequential implementation, pseudocode of the algorithm as follows:

Algorithm 1 Matrix Permanent Algorith proposed by Nijenhuis and Wilf

```
1:  procedure PERMANENT[MATRIX M,ROWSIZE N]
2:      lastColumn ← M[:N-1]
3:      sum ← M[:]
4:      x ← lastColumn - (sum/2)
5:      for i in x do
6:          p ← p * x[i]
7:      for i in [0,exp(2,n-1)] do
8:          z ← findChangingBit()
9:          s ← findSign()
10:         prodSign ← exp(-1,i)
11:         x ← x ∗ M[:z]
12:         p ← p + [x1*x2....]
        return 4*(remainder(N,2)-2)*p
```

Hence, algorithm has a complexity of $O(2^{n-1}n)$, which is one of the best algorithms in terms of complexity. Main idea in the algorithm is to create a gray code order for the changing bits, and use the location of the bit to update the x array for the next computation to come.

2.1 Sequential Implementation

Given the pseudocode of the sequential algorithm above, algorithm has 2 different loop part:

1. In the first loop, computation of the x array is being created. This array will further used for the calculation of the permanent value at the end of the algorithm.

2. During the each iteration of the second loop, algorithm compares the current version of the bits in gray code with previous iteration's bits, hence finding the changing

bit as well as the sign which determines whether the column of the matrix will be incremented or decremented from the elements of the x array.

3. Finally, updating the p variable and determining the final result.

2.2 GPU Implementation

Given the pseudocode of the sequential algorithm above, algorithm has 2 different parts:

CPU

1. In the main function, computation of the x array is calculated by CPU. This array will further used for the calculation of the permanent value at the kernel function. This part us being done in the CPU part whereas next stage of the algorithm runs in GPU.
2. Algorithm then allocates needed memory in device(GPU) as well as needed shared memory space in GPU for any given SM(streaming multiprocessor).
3. Finally, CPU copies the result needed from GPU and does one more final computation in order to find the result.

GPU

1. At the beginning, GPU allocates shared memory for every multiprocessor since using shared memory gives considerable amount of speed up.
2. In kernel function, for each thread value of the x array is copied to shared arrays in threads.
3. Size of chunks as well as starting index for each array is determined to distribute the workload between threads.
4. Gray coded part is computed for each thread before the iteration of each chunk for every thread.
5. For each iteration of the thread in chunk, calculated p value is added to local variable which will be then added atomically to original p value after the iteration.

3 Tricks

3.1 Flat Array of X and Matrix

Since creating an N sized array of x can be problematic for each thread, I decided to use a flat matrix of x that has N*sizeof(dataType)*threadNum bytes of size which is stored as a global variable in GPU device.

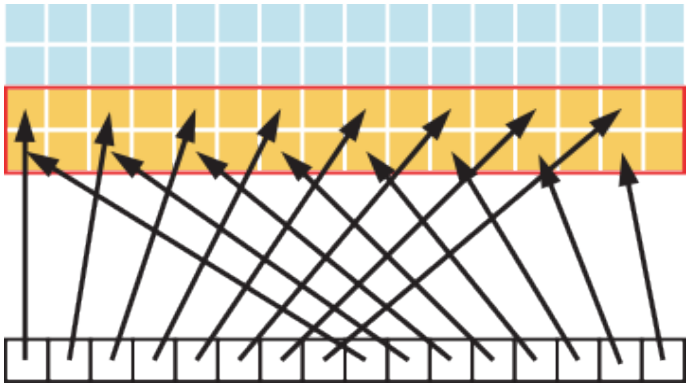
This approach allowed me to access each thread's own x values without facing any problem in terms of memory access. In addition to that, using flat array for memcopy and memory allocation simplified the memory transfer to/from GPU.

3.2 Shared Memory Idea

It is already known that usage of shared memory for each thread increases the speed of memory access dramatically. However, usage of shared memory access limits the usage of maximum threads. It is because there is a limit (48KB) for shared memory and increasing the amount of shared memory usage per thread decreases the number of threads that can be used in one block.

3.3 Coalesced Memory Access

A coalesced memory transaction is one in which all of the threads in a half-warp access global memory at the same time. The correct way to do it is just have consecutive threads access consecutive memory addresses.



Sometimes using coalesced memory access might be better than shared memory access since global memory is much bigger than shared one. It is also known that shared memory access is much faster in terms of time. This situation creates a payoff between the usage of these two attributes. Assuming that we can assign much more thread for a single block, coalesced memory access will do a better job since shared memory access limits the number of threads that can be used while coalesced memory access does not really limit the usage of threads for a single block.

3.4 Using Float Values Instead of Double

It also helps to use float values instead of double since double is represented in the memory with 8 bytes while float represented with 4. This information basically shows that if we use float for the shared memory arrays, this situation will decrease the memory we use to half.

Type	Bit width	Description	Range
bit	1	To define a bit of data	0 - 1
Char	8	Character or small integer.	signed: -128 to 127 unsigned: 0 to 255
Short Int	16	Used to define integer numbers	signed: -32768 to 32767 unsigned: 0 to 65535
Long int	32	Used to define integer numbers	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
Float	32	Define floating point number	1.175e-38 to 3.40e+38
Double	32	Used to define largest integer numbers	1.175e-38 to 3.40e+38 **Note: in other C ref 64 bit**
bool	8	Boolean value.	true or false
* (pointer)	width of memory	Use for addressing memory	Range of memory

Hence, increasing the number of thread we can use for a single block by 100% which also increases the speed of the algorithm due to more usage of threads for some block.

3.5 Manual Tuning of Thread Numbers

To limit the usage of threads for relatively small input sizes such as N=15, I used manual tuning in order to restrict the usage of threads and prevent possible slowdown.

Overall, I had to add some statements to the code but these statements doesn't prove to be harmful in terms of run time of the algorithm.

4 Results & Conclusion

Results showed that global memory proved to be the slowest among all of the algorithms. At the same time, when I used share memory decreased the runtime of the permanent algorithm considerably. Although shared memory had some effects on the algorithm, using float helped blocks to keep more threads since usage of shared memory is decreased due to representation of float(4 bytes). In conclusion, when I have shared memory to retrieve data from the matrix in addition x array, algorithm performed much better due to data retrieval from the memory. However, there was also a tradeoff between the usage of shared memory and coalesced access of the data, which was also described in the previous section. For the homework, shared memory usage proved to be much more efficient. Hence, I didn't really use coalesced access in the final implementation of the algorithm.

Experiments showed the average of 3 run time for any instance of the algorithm given in below while using the GPU device with id=0:

Approach	Global	Shared	Shared + Float	Input Size
Time(s)	0.0002	0.0023	0.0001	15
Time(s)	0.139	0.0037	0.0020	20
Time(s)	1.6	0.0114	0.0070	25
Time(s)	1185.1	31.44	12.7105	36
Time(s)	∞	135.472	112.905	38
Time(s)	∞	923.002	518.495	40

Since using global memory to compute the permanent takes to much time for bigger inputs such as N=38 or N=40, I showed the result of these algorithms with ∞ sign.

To compile: nvcc "filename" -O3 -o perm -arch=sm_61 -Xcompiler -O3 -Xcompiler -fopenmp

To run: .perm "txt file" devideId

Version1 (Global Memory Access): Name of the file is matPerm_hw3.cu

Version2 (Shared Memory Access): Name of the file is matPerm_hw3_v2.cu

Version1 (Shared Memory Access + Float Usage): Name of the file is matPerm_hw3_v3.cu

5 Problems Faces During Implementation

During the implementation phase of the homework, sometimes results are not clear or not precise due to some differences between GPU devices and some small differences in the code. That is why some results are not precise or not computed exactly as a result of implementation. However for small input sizes, it seems that this fact is not a problem.

Due to this fact, it can be hard to detect the problem since it is clear and precise for some cases, while it is not for others.

6 Future Work

For the future work and some possible improvements, possible ideas are the following:

- Some improvements can be used to increase the time of the CPU computations.
- Applying simd approach on GPU while iterating through the general loop that is assigned for each thread.

7 References

1 http://www.tech-faq.com/simd.html

2 https://www.go4expert.com/articles/builtin-gcc-functions-builtinclz-t29238/

3 https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

4 https://www.mathworks.com/matlabcentral/fileexchange/53784-matrix-permanent-using-nijenhuis-wilf-in-cmex