University of Pisa
MSc Computer Engineering
Computer Architecture

Parallelization of the Cofactor Method (Inverse Matrix Calculation)

Rispo Veronica
Sferrazza Gaetano

Academic Year 2021/2022

Contents

1 Overview

The main objective of this study was to optimize through the use of parallel programming the process of calculating the inverse of a square matrix, which is one of the most onerous operations when dealing with matrices, but also of great utility in solving systems of linear equations and therefore of some importance. More specifically, we focused on the algorithm associated with one of the various methods useful in carrying out such a calculation, namely the "*Method of Cofactors.*" The latter was chosen because of its high computational cost in the sequential version, which leads it in practice to be discarded when choosing an efficient method. Evaluations of the performance of this method were carried out mainly in terms of ***execution time*** and ***speedup*** between different versions of the algorithm, identifying the most relevant factors that determine their variation. Data extracted from the various tests performed were analyzed to gain insight into the variation of speedup and possible bottleneck in different system configurations, trying to identify the causes of their variation in performance.

2 System Configuration

The following are the system configurations on which the study was carried out and which framed all the choices made on the different experiments, thus defining the boundaries of expectations on the results due to the specific hardware configuration of the processing unit considered to improve the performance of the algorithm.

- **GPU**: GeForce RTX 2080 with Max-Q design with GDDR6 (quad data rate)
- **CPU**: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 2601 Mhz, 6 core, 12 logic processors
- **Theoretical Bandwidth**: Memory clock rate x Memory Bus Width x Data rate) : 10^9 = **768,13 Gb/s**

The images on the next page will show some more details in terms of hardware components and their corresponding implementation of the GPU model available to us.

```
CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 2080 with Max-Q Design"
  CUDA Driver Version / Runtime Version          11.6 / 11.6
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 8192 MBytes (8589606912 bytes)
  (046) Multiprocessors, (064) CUDA Cores/MP:    2944 CUDA Cores
  GPU Max Clock rate:                            1095 MHz (1.10 GHz)
  Memory Clock rate:                             6001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Figure 1: GPU characteristics form “deviceQuery” tool through Visual Studio

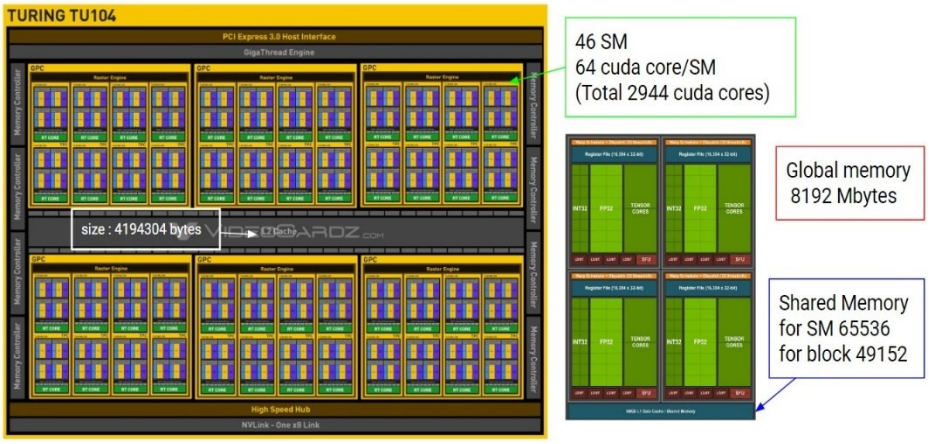

Figure 2: Hardware Architecture Components draw – NVIDIA GeForce RTX 2080 Max-QviceQuery” tool through Visual Studio

3 Tools

The following tools have been considered in order to perform the whole study:

- **IDE**: Visual Studio 2022
- **Toolkit**: Cuda toolkit v11.6
- **Profiler**: NVIDIA Nsight Compute
- **Microsoft Excel**: for simulation analysis and data visualization

4 The Cofactor Method Algorithm

Presenting how the Method of Cofactors algorithm operates to perform the inverse of a matrix is important to highlight which is the most onerous step and on which parallelization was performed to try to improve performance. Starting from the fact that the inverse of a matrix is only possible when such properties hold: *1) The matrix must be a* ***non-singular matrix*** ($det(A) \neq 0$), *2) The matrix must be a* ***square matrix*** *3) There exist an Identity matrix I for which*: $A \cdot A^{-1} = A^{-1} \cdot A = I$; the method is basically based on 3 main steps: *1) Compute of the elements of the cofactor matrix*: $cof_{ij}(A)^T = (-1)^{i+j}. \ det\ (Sub\_M_{ij})$ *2) Compute transposed matrix (Adjoint)*: $adj(A) = cof(A)^T$ *3) Compute inverse matrix*: $A^{-1} = \frac{1}{det(A)} \cdot (\ adj(A))$. The first step is where one can and should take act, another thing to note is what this step is heavily dependent on, that is the calculation of the determinant of the submatrix obtained by eliminating the i-th row and j-th column of the starting matrix whose inverse is to be calculated. This becomes crucial for optimization purposes for finding the cofactor, introducing a not trivial cost to the whole method in general.

5 Algorithms Assessment

Two different types of algorithms that could enable the implementation of the cofactor method were evaluated, taking into account what was said in the paragraph above. In fact What differentiates the two algorithms considered is the type of operations to be performed for the calculation of the determinant of the submatrix, more precisely the first version evaluated was based on ***La Place's method*** for calculating the determinant while the second exploits the well-known ***Elimination Of Gauss*** (MEG) approach. Of the two we chose to work with the implementation of the latter because it has a lower cost and is less constraining than that of La Place, just to give an example one negative aspect of the latter is the fact that it operates recursively and this leads to undesirable consequences on the scalability of the method when applied to large matrices.

6 Implementation

The implementation was concretized through the use of two programming languages. The ***C language*** for the sequential part of the algorithm, which was then converted into the respective parallel version through the ***Cuda C Language***. The thing we want to focus on in this section of the paper is the interpretation of the various parts of the source code. The program was written all in one file - **kernel.cu** - where there are parts that are commented out and left as such intentionally; this is because we proceeded from time to time to "undo" a certain part of the code that we did not need for a certain test, and if necessary we made it operational again according to the type of experiment to be performed. By doing this we tried to avoid the creation of too many files with different versions of the program resulting in confusion. For the variation of some parameters, constants were defined related to the size of the matrix and the size of the blocks associated with the various ***execution configurations*** useful for thread manipulation. The rest has been enriched as much as possible by comments useful for understanding the behavior of

the program in general. Regarding various functions used one can get an overview of them immediately at the beginning of the code, thanks to the Prototype/Definition Function setting.

7 Performance Evaluation

7.1 Performance Indexes

The following indexes have been used to evaluate the performance of the Algorithm:

- **Mean Execution Time**: the mean time takes to compute the inverse of matrix
- **Speedup**: Boost gained on performance between different versions from a basic version

In particular, to reduce the risk of errors and/or inconsistencies on the values obtained, a 90% *confidence interval* was calculated for the different mean run times. Given the low value of accuracy, the intervals are very small, which is why it is difficult to see their presence in the graphs.

- ***Note**: All the graphs related to the performance evaluations are in the **Power Point** document sent with this documentation, which will mainly serve as a general guideline related to the different phases of the study carried out.*

7.2 Sequential vs Parallel Algorithms

This initial comparison of the two versions gave us immediate feedback of the marked improvement in the performance of the algorithm associated with the cofactor method after the first phase of parallelization, and of how markedly superior, after a certain threshold, the computational capacity of the graphics processing unit used for nongraphical purposes made available by our GPU was. The analyses performed were of two types:

1) Analysis of performance as the size N of the matrix varied.

2) Analysis of performance as the threads varied on a fixed NxN matrix.

Feedback was first evaluated in terms of ***Execution Time***, both CPU and GPU side. Then through the ***SpeedUp*** obtained by considering the ratios between the execution times of the two methods in relation to the first analysis. The second analysis was done on the parallel version of the algorithm to observe which was the best configuration and what the relative gain could be once the speedup was calculated on this type of experiment. It was found, however, that the trend of the speedup was not as we expected. For this reason, we tried to find a solution through a new version of the base kernel.

7.3 Parallel Algorithms Variants

At this point we asked ourselves how to further improve the performance of this parallel version of the algorithm, and thus how to make the best use of GPU parallelism. We then focused on studying possible variations in terms of improvements to the algorithm and also in the use of the various threads made available by the various possible execution configurations. A second optimized

version of the base kernel was then identified, applying unroll and reducing the workload in terms of algorithm operations. The two types of analysis mentioned in the paragraph above were then carried out again. The first analysis between the base kernel version and the optimized version, and indeed an improvement was found in terms of execution time and relative speedup. We then proceed to the second type of analysis with the new version of the kernel but find that actually, the speedup trend has not changed. It was therefore realized that the problem might lie elsewhere.

7.4 New Prospective for the Problem Analysis

We then reflected on the fact that the problem of the unusual trend of the speedup could basically fall into two aspects: *1) A wrong memory usage* or *2) Incorrect workload balancing between threads*. Having fixed the ideas, the previously mentioned analyses were then carried out again by once again modifying the configurations for ad hoc experiments.

7.5 Memory Utilization Analysis

In order to test the hypothesis (Inability to use the ***shared memory*** of the blocks because of its small size) that led us to think that the problem might lie in memory the second type of analysis, mentioned earlier, was carried out again, but this time on a smaller matrix size. Analyzing the results and the actual memory usage using the Nsight profiler and the new speedup yielded a trend of the latter, unfortunately, still questionable.

7.6 Balancing Workload Analysis

In order to test the hypothesis (negligible change in elements assigned to a thread for a net change in execution time resulting in ***time sharing***) that led us to think that the problem might lie in improper workload balancing among the threads, the second type of analysis was carried out again but with the addition of a calculation associated with the number of matrix elements assigned to each thread at a specific stage of the test, to highlight the expected result by cross-comparison with the relative speedup obtained at the end of the experiment.

8 Conclusion

What we had set out to understand was, through experimentation and evaluation, whether it was possible to bring back to light a method now obscure in the field of operations on matrices, due to the possibility of being able to use more computing power by exploiting hardware/software parallelization. And in general this is what we found from our observations and experiments, as it is also true that the effort in attempting to maximize the performance of the algorithm in the parallel key was not entirely trivial, especially taking into account those sequential parts of the code forced to remain as such, thus reducing the possibility of increasing the *speedup* in accordance with what *Amdahl's law* states. However, what we can conclude based on the study that has been conducted is that the right way forward for more optimization of the algorithm is to improve the workload balancing *(bottleneck)* in order to make more efficient use of hardware/software parallelization and gain more speedup and scalability.