TECHNICAL REPORT
SEBASTIÁN GALGUERA ORTEGA
SEPTEMBER 2018
MULTINUCLEUS PROGRAMMING

Abstract

The following document describes the process that was followed in order to multiply two matrices, using sequential and parallel approaches with OpenMp and Nvidia's CUDA. The aim of the experiment was to identify and benchmark the characteristics of both approaches. Tables are provided in order to describe the nature of the tools and procedures involved.

1. Introduction

The project consists of a file called MatrixMultiplication.cu, which offers three main functions (aside of helper and utility functions): multiplyMatrixOnHost that follows linear algebra rules to get the dot product of two matrices; multiplyMatrixOnHostThreads that uses OpenMp to automatically parallelize the multiplication; and multiplyMatrixOnGPU, that uses a CUDA kernel in order to achieve the same goal. Basically, the main objective of this task was to tune the models and register the performance gain or loss of the parameter tunning.

Of course, one could think that all parallelizing tools will always achieve a better run-time performance. However, this is not necessary the case. As it will be seen later on, the OpenMp solution doesn't seem to get an important improvement as the matrices scale on magnitude, in contrast to CUDA, and even sometimes it can be actually slower than its sequential counter-part.

2. Development

The main tests were executed on a remote server via secure shell. The sizes of the computed square matrices were 1000 for the first iteration, 2000 for the second and 4000 for the third. Also, the number of threads and the block and grid sizes were tuned in order to make a comparison between the parameter settings.

The computer specifications:
Brand: Alienware 32 GB RAM
CPU: 3.9 GHZ
CPU Cores: 6
GPU: GeForce GTX 670
GPU Cores: 1344

The following table specifies the average elapsed time measured in milliseconds of each function for each size of matrices. It can be seen how a size of 4000 kills the server.

TABLE OF AVERAGE ELAPSED TIME (ms)	1000	2000	4000
multiplyMatrixOnHost	5816.540527	75801.437500	KILLED
multiplyMatrixOnHostThreads	1826.907349	20670.330078	KILLED
multiplyMatrixOnGPU (512,1)	66.374031	494.974548	KILLED

Now, a table of improvement, a quantity that is a function of time of execution:

TABLE OF IMPROVEMENT (S(n)=T(1)/T(n))	1000	2000	4000
multiplyMatrixOnHost	0	0	KILLED
multiplyMatrixOnHostThreads	3.18	3.66	KILLED
multiplyMatrixOnGPU (512,1)	87.63	153.14	KILLED

The following table describes the average time of matrices with a size of 1000 with different 2D kernel configurations. The 2D kernel was selected because the way matrices multiplications work is logically correlated with the structure of 2D CUDA memory allocation and access.

TABLE OF KERNEL CONFIGURATIONS (2D)	AVERAGE TIME (1000)
(512,1)	66.703568
(1,512)	1170.515747
(256,2)	67.666557
(2,256)	455.979675
(64,16)	62.734070
(16,64)	60.294834
(32,32)	63.219215

It can be seen how performance was gained with the (16,64) configuration of the 2D kernel. It is interesting to see how simply tuning these parameters can make a model better or worse. Also, it can be noticed that CUDA seems to be more efficient than OpenMp. Even though OpenMp is easy to scale for programmers because of its easy-to-write code, it doesn’t actually scale up that much on performance.

3. Conclusion

It can be concluded that understanding the different mechanisms of parallelization tools can help us achieve better computation times. Also, what suits a type of problem may not suit others of the same domain. Effectuating tests and recording metrics is the best way to truly know if one’s code has been improved or if there is more room for evolution.

In the best case scenario of this report, the computation time of the worst performing setup of matrices of size 1000 was 816.540527 sequentially, while the best performing setup had a time interval of 60.294834. Nevertheless, it is also necessary to evoke the fact that this time, implementing the CUDA optimization was quite fast and so it was economically viable to do so. However, sometimes the performance gains don't offer a good enough incentive for a software development to change its structure to a parallel one, given the high cost of implementing complex optimizations. Lastly, it is obvious that better computation time makes technology more adaptive, more useful and more powerful. Being able to re-think how a problem can be solved, but now in a parallel way, is one of the skills that are needed today and will be needed for the next century.

3. References and Figures

"GEFORCE RTX™." *NVIDIA*, NVIDIA, 1 Sept. 2018, www.nvidia.com/es-la/.

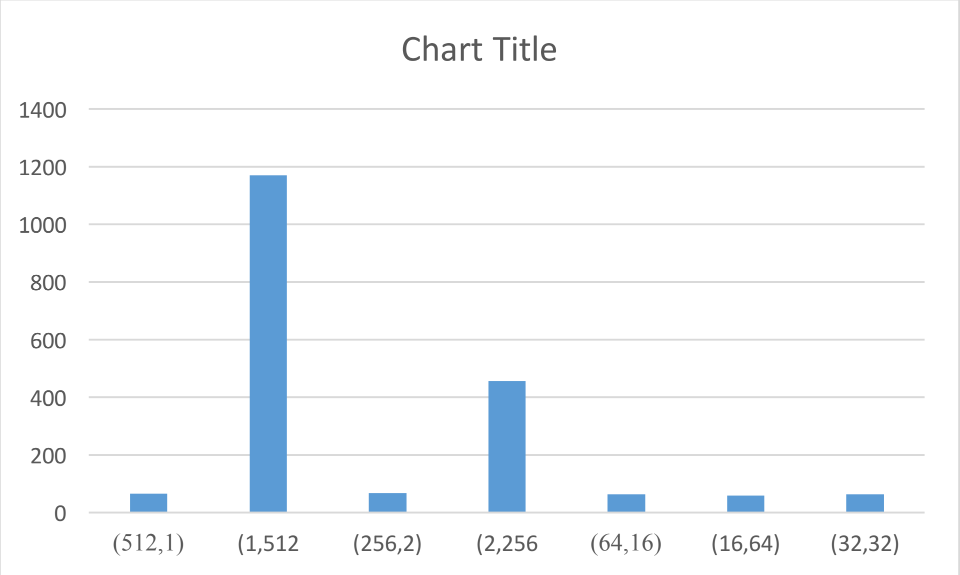
Chart Title
(512,1)
(1,512
(256,2)
(2,256
(64,16)
(16,64)
(32,32)