Exploring Parallel Programming

Threads and CUDA Applied to Matrix Multiplication

Euclides Palma Paim
Applied Computing Graduate Program (PIPCA)
University of Vale do Rio dos Sinos (UNISINOS)
São Leopoldo, Brazil
euclidespaim@gmail.com

***Abstract*—This article discusses the subject parallel programming, are presented theoretical and practical concepts on the subject. This paper provides a description of the implementation and testing of a solution using parallelism with threads. Further describes the results obtained in the implementation and testing of a parallel solution using CUDA. These performance tests was realized for comparison using the best known algorithm for the linear problem and parallel solution. The problem approach in this article referred to a multiplication of square matrix utilizing both solution mentioned below.**

***Index Terms*—Parallel programming, threads, CUDA, tests.**

I. INTRODUCTION

There is a continual demand for greater computational speed for computer system problems. Such problems often need huge repetitive calculation on large amounts of data to give valid results. Computations must to be completed within an acceptable time period. There are some problems that have a specific deadline for the computations as weather forecasting, petroleum or gas prospection.

These areas are grand challenge for today's computers, large execution time is always contradictory face the massive power of computation available. One way of increasing computational speed is by using multiple processor operating together on a single problem. The overall problem is split into parts, each of which is performed by a separate processor in parallel. Writing programs for this form of computation is known as parallel programing. The computing environment for parallel programing could be a specially designed computer system containing multiple processor or several independent computers connected. The idea is that *n* computers could provide up to *n* times the computational speed of a single computer. Obviously this is hypothetical situation that rarely achieved in practice. Problems commonly cannot be divided in parts, transfer to another computers, processed, gathered and synchronized without lose time in data transfer and communications.

A parallel computer is not a new idea, for example [1] writes about a computer capable of executing an arbitrary numbers of sub-programs simultaneously in 1959.

Parallel programming requires suitable computing platforms, which we can describe as either a single computer with multiple internal processors or multiple interconnected computers. A conventional computer consists of a processor executing a program stored in a main memory [4]. A natural way to extend the single processor model is creating a cluster or grid who will processor connected to multiple memory modules, such that each processor can access any memory module in a so-called shared memory configuration. The connection between the processors and memory is through some form of interconnection network.

In a single processor computer, a single stream of instructions is generated from the program. In 1996 [2] created a classification for computers and called this a *single instruction stream-single data stream* (SISD) computer. For example von-Neumann traditional machines. For a common function multiprocessor system, each processor has a separate program and one instruction stream is generated from each program for each processor. Clasified by [2] as a type of computer *multiple instruction stream-multiple data stream* (MIMD) computer.

No matter what class of computer you use, to achieve an improvement in speed through the use of parallelism, it is necessary to divide the computation into tasks or processes that can be executed simultaneously. The size of a processes can be described by its *granularity.* In coarse granularity, each process contains sequential instructions in large number and takes a considerable time to execute. In fine granularity, a process consist of a few instructions or even one instruction. Sometimes granularity is described as the size of the computation between communication and synchronization points. Generally, we want to increase the granularity to reduce the cost of process creation and interprocess communication.

It is particularly desirable in message passing where reduce communication overhead is crucial, because of the significant time taken by inter computer connection. The coast of the operations needs to be objective to make it less expensive in time of computation. Granularity is related in [4] with the number of processors being used. The ratio

$$\text{Granularity} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{comp}}{t_{comm}} \quad (1)$$

can be used as a metric and maximize the granularity while maintaining sufficient parallelism.

A measure of relative performance between a multiprocessor system and a single processor system is the *speedup factor*, defined in [4] as

$$S(n) = \frac{\text{Execution time using one processor}}{\text{Execution time using multiprocessor}} = \frac{{}^{t}s}{{}^{t}p} \qquad (2)$$

where ${}^{t}s$ is the execution time on a single processor and ${}^{t}p$ is the execution time on a multiprocessor. $S(n)$ gives the increase in speed in using a multiprocessor . For comparing a parallel solution with a sequential solution we will use a fastest sequential algorithm for running on a single processor.

It is reasonable that some parts of computation cannot be divided at all into concurrent processes. The Amdahl's argument [3] claim that the performance gain that can be obtained by improving a particular part of the system is limited by the fraction of time that the part is used by the system during operation. In other words, periods when not all processor can be performing useful work, extra computations in parallel version not seen in the sequential version and the communication time for sending messages, has a potential to improve the overhead of a parallel version and limit the speedup.

II. Threads, Processes and Multithreading

In concurrent programming, there are two basic units of execution: processes and threads. Here we are mostly concerned with threads. A computer system normally has many processes and threads, even in systems that only have a single execution core. Processing time for a single core is shared among processes and threads through an operational system feature called time slicing.

Threads and processes provide an execution environment although creating a new thread require fewer resources than creating a new process. Threads differ from traditional multitasking operating system process. Threads can exist within a process, every process has at least one. These share the process's resources, including memory and open files. This makes more efficient, but potentially more problematic for communication. Processes are typically independent, while threads are subsets of a process and don't have separate address spaces.

Another approach in parallel programming is multithreading. This is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of a single process. These threads share the process's resources, but are able to execute independently.

To the programmer's point of view, there are advantages and disadvantages in dividing an application into multiple threads. On the one hand it facilitates the development, it is possible to develop the program into modules, testing them in isolation, rather than writing a single block of code. On the other hand, multithreaded work becomes more complicated due to the interaction between them.

III. GPU Computing

To understand the architecture behind the graphics processing units (GPU) is necessary to look for the CPU architecture and compare both. CPUs are designed to get maximum performance from a stream of instructions, which operates on diverse data, such as integers and floating-point calculations, and performs random memory accesses, branching, etc. Architects worked to extract more parallelism of instructions and launch as many instructions as possible in parallel in CPUs. The problem is that there is a limit to the parallelism that is possible to get out of a sequential stream of instructions and consequently, increasing the number of calculating units is useless, since they remain unused most of the time.

Differently, the operation of a GPU is elegantly simple. The job consists of taking a group of polygons, on the one hand, and generating a group of pixels on the other. The polygons and pixels are independent of each other, and so can be processed by parallel units. That means that a GPU can stay free to devote a large part of its die to calculating units which, unlike those of a CPU, will actually be used.

We used the CPU, or several CPUs) for office and Internet applications and GPUs were good only for drawing pretty pictures faster. But an event change all that, the appearance of programmability in GPUs. The idea of using graphics accelerators for mathematical calculation is not recent. The first traces of it go back to the 1990s. Initially it was very primitive and limited. In 2003 a new stage was reached but the only way to get access to the GPUs resources was to use one of the two APIs existing: Direct3D or OpenGL. Consequently, researches who wanted to harness the GPU's processing power had to work with these APIs. The problem was that those individuals weren't necessarily experts in graphics programming, which seriously complicated access to the technology. This difference between two areas of technology leveraged the development of solution that simplify the use of the resources of computation on GPUs.

One of the first efforts was a set of extensions to the C language presented by Stanford University called BrookGPU. Concretely, Brook proposed to encapsulate all the management part of the 3D API and expose the GPU as a coprocessor for parallel calculations. The project Brook had merits to be the first to bring General Purpose Graphic Processing Units (GPGPU) to the public knowledge.

The Compute Unified Device Architecture or CUDA is a parallel computing platform and programming model created by NVIDIA that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

CUDA has been widely deployed, since its introduction in 2006, through thousands of applications and published research papers. Applications used in astronomy, biology, chemistry, physics, data mining, manufacturing, finance, and other computationally intense fields are increasing using CUDA to deliver the benefits of GPU acceleration.

The company has chosen to use a rather special terminology that can be hard to grasp. First we need to

understand what a thread is in CUDA, because the term doesn't have quite the same meaning as a CPU thread. A thread on the GPU is a basic element of the data to be processed. Unlike CPU threads, CUDA threads are extremely "lightweight," meaning that a context change between two threads is not a costly operation.

Another term frequently encountered in the CUDA documentation is warp. A warp in CUDA, then, is a group of 32 threads, which is the minimum size of the data processed in SIMD fashion by a CUDA multiprocessor. This granularity frequently is hard to be used by programmers, so in CUDA, instead of manipulating warps directly, you work with blocks that can contain 64 to 512 threads.

These blocks are put together in grids. The advantage of the grouping is the number of blocks processed simultaneously by the GPU are closely linked to hardware resources. The number of blocks in a grid make it possible to totally abstract that constraint and apply a kernel to a large quantity of threads in a single call, without worrying about fixed resources.

Finally other terms frequently used in the CUDA API are host and device. The first designates the CPU and the following refer to the GPU.

IV. Results Achieve Exploring Threads

During this research we analyzed a series of data obtained using threads on machines type *t2.micro* and *g2.2xlarge* virtualized available in *aws.amazon.com*. In order to exploit the resources of concurrent computing tasks, we developed this study based on the measurement of code runtime executed in a controlled environment. Compared to the execution of a sequential matrix multiplication algorithm with the same resources and with different resources.

Data was obtained from a computational load generated by using square matrices algorithm which requires sequential time complexity of $O(n^2)$. The way used to implement the matrix multiplication algorithm was allocate one thread to compute each line and column of the matrix. The following data describes the relationship between the size of the arrays and the runtime using threads, Table I.

TABLE I. Table Threads System.

	Table Column CPU		
N = 100	0,129989	N = 600	5,421470
N = 200	0,531282	N = 700	8,319280
N = 300	1,186866	N = 800	11,467733
N = 400	2,254106	N = 900	15,007080
N = 500	3,529937	N = 1000	20,192607

The results gather here shows relation between time of execution and size of the matrices and it is described in Fig. 1. The diagram shows a linear behavior related with a increase in the size of the array.

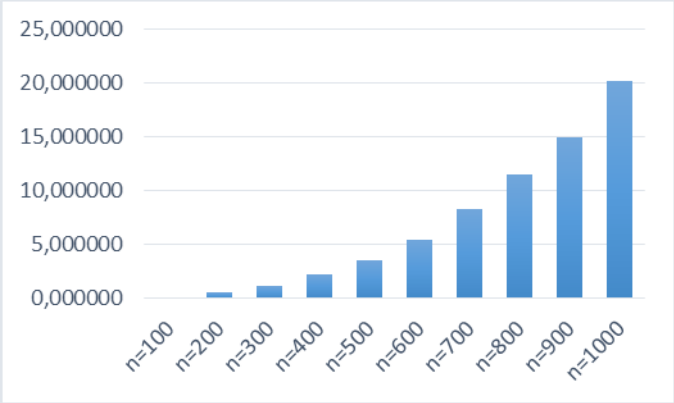

Fig. 1. Graph shows time versus number of rows and columns.

TABLE II. Time with Large Resources

	Table Column CPU		
N = 100	0,330897	N = 600	12,495379
N = 200	1,255050	N = 700	18,400707
N = 300	3,065345	N = 800	25,384760
N = 400	5,392766	N = 900	33,604839
N = 500	9,062683	N = 1000	40,860793

However the results collected in *g2.2xlarge* machine whith eight cores running the same algorithm reveal a relevant increase of time execution. This behavior might describe a overhead promoted by the bottleneck in comunication resources or even a bad use of them. The diagram in Fig. 2 show the growth of time running the matrix multiplication with increase of data.

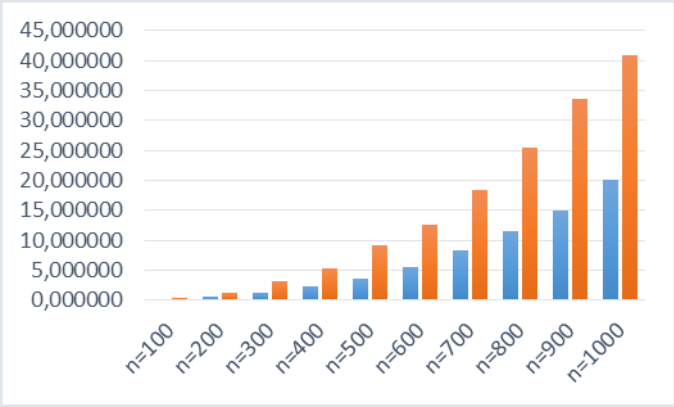

Fig. 2. Compare between machines

V. Results Achieve Exploring GPU

There are several kinds of memory on a CUDA device, each with different scope, lifetime, and caching behavior, know how is the best solution is the goal [5]. In order to compare the runtime computation exploring different approaches related to parallel programming we adopted solutions have been

implemented running on device with global variables and the use of shared memory. These results were compared to results obtained from running the same work load multiplication square matrix size "n" on the host. In the survey we found results supporting the use of GPU through CUDA implementation reducing the computation time compared to host.

For such tests *g2.2xlarge* virtualized machine were used, this machine has high performance NVidia GPUs, each with 1536 CUDA cores and 4 GB of video memory, Intel Xeon E5-2670 processors (Sandy Bridge) high frequency and 15GB RAM. The operating system used was the Ubuntu and CUDA interface available through AMI, release 7.0, V7.0.27.

The results were collected in three stages and depict the growth in relation of time of computation and amount of data. The next table show time of processing executed on CPU, time increases consistently with respect to increasing the size of calculation into matrices.

TABLE III. TIME COMPUTING HOST

	Time CPU		
N = 100	0,0061380	N = 800	70,7780400
N = 200	0,0594650	N = 900	103,9549600
N = 300	0,2075470	N = 1000	112,5256900
N = 400	0,5935220	N = 2000	126,9203600
N = 500	1,5190770	N = 3000	593,9959930
N = 600	14,1679700	N = 4000	1302,3463500
N = 700	46,7894000		

a. Times obtained using CPU

We notice a performance gain when performing matrix multiplication using overlap computation of GPU devices. We can see the result obtained using global memory in Table IV.

TABLE IV. TIME COMPUTING DEVICE GLOBAL

N	**Time GPU**		
N = 100	0,000212	N = 6000	10,884820
N = 1000	0,058246	N = 7000	17,307934
N = 2000	0,415002	N = 8000	25,826821
N = 3000	1,370736	N = 9000	36,817905
N = 4000	3,228621	N = 10000	50,386388
N = 5000	6,290777		

a. Times obtained using global variable

The results obtained using coalesced global memory were virtually the same, not reducing the execution time for the same amount of rows and columns. But with a substantial gain compared with the execution on CPU, Table V shows the times obtained using coalesced global memory.

TABLE V. DEVICE GLOBAL COALESCED MEMORY

N	**GPU Global (coalesced)**		
n=100	0,000221	n=6000	10,8994430
n=1000	0,056394	n=7000	17,2983500
n=2000	0,412831	n=8000	25,8280860
n=3000	1.366.812	n=9000	36,7643020
n=4000	3,2378500	n=10000	50,4193770
n=5000	6,3030230		

The Table VI show the results collected utilizing shared memory for square matrix multiplication on device. We do not observe performance gains and increased speedup with the use of coalesced global memory this work

TABLE VI. TIME COMPUTING ON DEVICE WITH SHARED MEMORY

	GPU Shared Memory		
N = 100	0,000182	N = 6000	6,437063
N = 1000	0,030215	N = 7000	9,747571
N = 2000	0,242912	N = 8000	15,220369
N = 3000	0,787267	N = 9000	20,640033
N = 4000	1,923507	N = 10000	29,695140
N = 5000	3,575261		

The best results are acquired running CUDA with shared memory. We compared the time found with each category about GPU programming described in this article. Fig. 3 demonstrate results found among them all. Comparisons were made between the algorithms developed in CUDA and their sequential version. The implementation of matrix multiplication, the CUDA model had great time advantage reaching speedup of more than 700x.

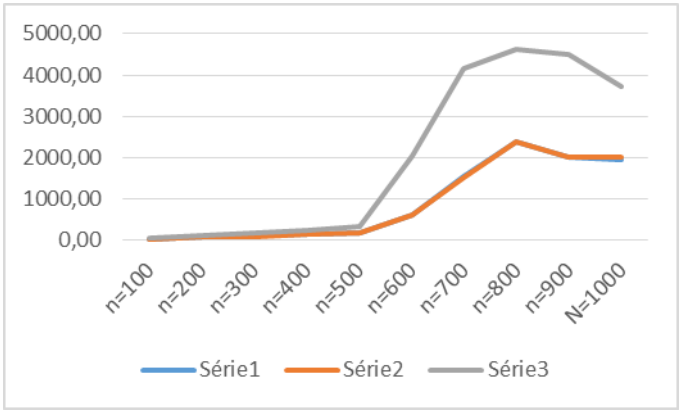

Fig. 3. Comparison of speedup with increase in data.

VI. Conclusion

The continuous increasing in parallel resources and GPU computation devices, allow possibilities not envisioned before. This amount of resources facilitates the research of parallel architecture. Though not so simple to develop the knowledge need to implement a real parallel solution there are large documentation about the matter accessible. Shared memory is a powerful feature for writing well optimized CUDA code. Access to shared memory is much faster than global memory access because it is located on chip.

In this article we discusses some aspects related of how to efficiently access memory with CUDA. In the survey we found results supporting the use of GPU through CUDA implementation reducing the computation time of a specific case conducting matrix multiplication. On the other hand the results collected using *threads* show up conflicting deserving more analysis in future works.

Notwithstanding research has shown to be relevant gathered data involving memory access in Compute Unified Device Architecture. We about the use of memory in different features and a way to use shared memory that enable global memory coalescing, as demonstrated in this paper.

References

[1] J. Holland, "A universal computer capable of executing an arbitrary number of sub-programs simultaneously," in *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference* (IRE-AIEE-ACM '59 (Eastern)). ACM, New York, NY, USA, 108-113. (1959)

[2] M. Flynn, "Some computer organizations and their effectiveness"; Department of Computer Science, The Johns Hopkins University, Baltimore, Md. 21218.

[3] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities" in AFIPS Conference Proceedings (30): 483–485. (1967).

[4] B. Wilkinson and M. Allen, "Numerical Algorithms" in *Parallel Programming*, 1st ed. Upper Sandle River, New Jersey, USA vol. 1. Prentice-Hall, 1999, pp.301–310.

[5] M. Harris, "How to access Global Memory Efficiently in CUDA C/C++ Kernels" [Online]. http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels, Accessed on: Oct, 2015.

[6] REVISTABW. CUDA: Multiplicação de Matrizes, Revista Brasileira de Web, 2014. [Online]. Available: http://www.revistabw.com.br/revistabw/cuda-multiplicacao-de-matrizes/. Accessed on: Oct, 2015.