International Journal of Advances in Engineering and Management (IJAEM)
Volume 4, Issue 7 July 2022, pp: 195-199 www.ijaem.net ISSN: 2395-5252

Performance Evaluation of Matrix Products on Multicore Architectures

SharlonAlmida de Silva*, Claudio Scepke*, Natuan Agrawal**

*Student, Federal University of Pampa, Rio Grande do Sul, Brazil
**Professor, Amaravathi Institute of Technology, Tamil Nadu, India

Submitted: 01-07-2022 Revised: 04-07-2022 Accepted: 08-07-2022

ABSTRACT: Scientific applications tend to deal with large volumes of data, as is the case with simulations of natural phenomena, which demand high computational power. Alternatively, using multi-core computers for processing contributes to performance improvement. However, performing specific optimizations for the target architecture can further influence performance. Therefore, the objective of this work is to evaluate the impact of optimization techniques on application performance as well as test the performance of these techniques in multi-core and many-core architectures. For that, a matrix multiplication algorithm was chosen for the application of Loop Interchange, and Loop Tiling techniques. Furthermore, this algorithm was parallelized with OpenMP and CUDA to explore the different processing cores of the computational architectures used. The results show that algorithms optimized for a target architecture gain performance, and this gain can reach 11 times in sequential optimizations for cache memory and 100 times in parallel execution with OpenMP on Intel Xeon E5-2650 processors. Furthermore, this performance gain can be leveraged on the NVidia TITAN Xp GPU up to 1720 times.

KEYWORDS: Matrix Product, Algorithm Optimization, Multi-core, many-core architectures.

I. INTRODUCTION

Computing has revolutionized scientific research as it solves real science problems through numerically intensive calculations. Commonly real problems require a lot of input data because the more information is computed, the more accurate the results will be. But the more calculations are performed, the slower the execution, which implies the need for powerful machines to make it possible to obtain accurate results quickly.

Parallel computing is necessary to enable the simulation and understanding of phenomena such as galaxy formation, molecular dynamics, genetic sequencing, mathematical simulations, oil prospecting, protein synthesis, weather forecasting, and geophysical studies, among others (Navarro et al 2014). Many applications are not feasible to be solved through sequential computations in a single processing core because of the time they take to provide their results. In meteorology, for example, application execution time is unfeasible in sequential architectures, as the response would be obtained after the phenomenon occurs (Schepke et al 2013).

Professionals from different areas use computing to support their scientific research. The agility in the return of results during the execution of the applications is one of the factors that make possible the current scientific advance, requiring powerful computational resources for this purpose. Thus, in order to solve runtime problems and meet the processing needs of applications, there are multi-core and many-core parallel architectures. The parallelism provided in these multi-core and many-core machines makes it possible to run an application faster, as several tasks can be computed simultaneously on the various cores available in the hardware.

High-Performance Computing (HPC) is the area of study that makes it possible to reduce the execution time of applications through the execution and optimization of algorithms in multi-core and many-core architectures, enabling the resolution of problems each time. bigger in runtime smaller and smaller. For this research, two types of high-performance processors were used, the CPU (Central Processing Units, or Central Processing Unit) and the GPU (Graphics Processing Units, or Graphics Processing Units), which enable high computing capacity.

This work aimed to evaluate the performance of a matrix multiplication algorithm in multi-core and many-core architectures, highlighting the best optimization techniques used and comparing the execution time obtained between the Sequential, OpenMP (Open Multi-Processing) and CUDA (Compute Unified Device Architecture).

The rest of this article is organized as follows, Section 2 presents the methodology of this work. In Section 3 the results obtained are presented. Finally, Sections 4, 5, and 6 present the final considerations, the acknowledgments, and the references used, respectively.

II. METHODOLOGY

Matrices are fundamental operations of Linear Algebra. They are used for the representation of data in several areas of knowledge. An efficient algorithm for matrix calculus is important. When the order complexity increases, the operation demands greater performance capacity (Marquezan et al 2002). With the technological advance of the last decades, the algorithms need to be modified to better take advantage of the current computational power.

Efficient implementation of the matrix product is critical as it appears in many scientific applications, as it mathematically models a linear function. The matrix multiplication A(mxn) × B(mxn) is given from the dot product between the row elements of matrix A and the column elements of matrix B, generating the corresponding matrix Cmxn. Every matrix has an mxn order to represent the size of the rows and columns, respectively, and its elements are represented by the letters i(rows) and j(columns).

Although matrix multiplication is a simple operation, the way in which the elements are computed directly influences the performance of the operation. Depending on the way the programmer structures the algorithm, the execution time varies. An efficient implementation of calculus can reduce a run from many hours to minutes, or even seconds.

The fetching of instructions and data by the processor is related to the communication between the CPU and the main memory. High main memory access latency is one of the performance limitations. The use of cache memory serves to minimize the impact of this latency, since it stores data frequently used by the processor based on the principle of locality (Lee et al 2010).

Figure 1 illustrates how cache memory works. When an application is executed, data and instructions are loaded into the main memory, and the most accessed ones are kept in the cache to avoid higher latency fetching in the main memory. The closer the data is to the CPU, the faster it will be accessed. The CPU fetches data or instruction first in the cache levels, starting from the L1, L2, L3 levels to the main memory, if the data is found in one of the cache levels, the fetching ends.

Optimizations in the implementation of algorithms are necessary to better use the available hardware and obtain more performance when running applications. The looping techniques presented are useful when it comes to CPUs. But another way to gain performance is through the use of GPUs. A GPU is an architecture where there are hundreds or even thousands of processing cores, and its programming requires a programming interface. In this work, NVidia's CUDA interface was used (Kirk 2010).

The implementation of the algorithms presented in this work uses order square matrices 4096×4096in the C language. This value exceeds the cache memory size of the machine used, making the visualization of performance gain with cache optimization more remarkable. For the loop tiling technique, a block size equal to 16x16 was chosen, since this value fits completely in the L1 cache. For the parallel version with OpenMP, 16 threads were used, as the machine used has 16 physical cores. And for the CUDA version on the GPU, blocks of size 32×32 were used, limiting the number of threads per block to 1024. To analyze the results, the algorithms were submitted to 30 executions and the times were averaged. The number of runs chosen is due to the t-student setting to stabilize the standard deviation.

The environment is used as a workstation, with two Intel Xeon E5-2650 CPUs, each processor has 8 physical cores, allowing the execution of 32 threads with Hyper-Threading. Each core has a private L1 cache of 32" "And a private L2 cache of 256 KB. The last cache level is L3 with 20MBshared. An NVidia Titan Xp GPU with 3840 CUDA cores and 12GB of RAM was also used. For the CPU, the compiler used was the GCC with the -O2 optimization directive, and for the GPU, the NVCC from NVidia was used.

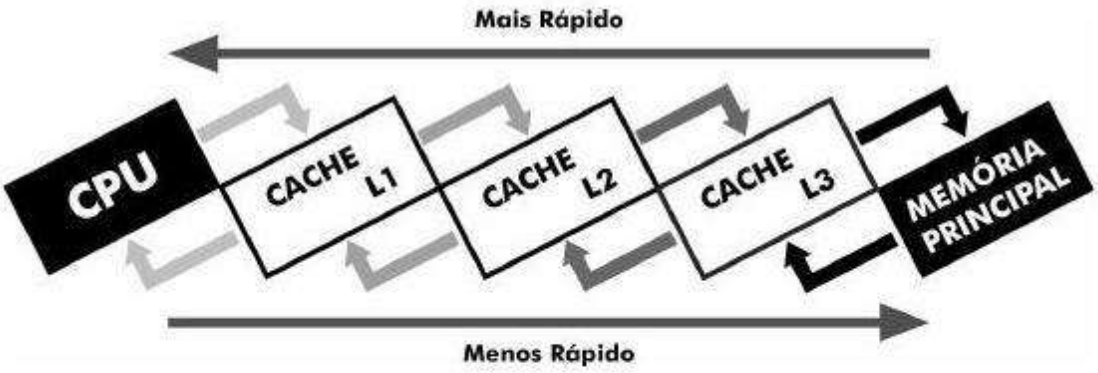

Figure 1. Cache memory.

III. RESULTS AND DISCUSSION

Table 1 presents the average runtime and performance gain of a parallel version over the respective sequential version (speedup) and the initial version time over each runtime collected from each implementation. Initially, the sequential execution time for a conventional array of order 4096 (Naive) without optimizations was 849.8 seconds.

Using loop tiling and loop interchange optimization, we obtained an execution time of 165.7and 76.9seconds, respectively. Therefore, it is concluded that if the computational environment to be used has few cores, the indicated optimization technique is a version of loop tiling or loop interchange since they exploit the spatial and temporal locality of the cache memory. In the tests, they guaranteed performance of 5.12e 11.04times in relation to the sequential version.

The parallel implementation with OpenMP achieved a gain of 52.8times for loop tiling and 100.39times for loop interchange in an architecture composed of 16 cores. Thus, it is noted that the multi-core computing environment allows the application to be scaled expressively. In the many-core architecture, the parallel potential allowed a performance gain of up to 1720.3times compared to the sequential version without optimization used in the multi-core architecture.

Table 1: Execution Time and Performance Gain.

Implementation	Version	time(s)	speedup	Gain over the initial version
sequential	naive	849.845	1.00	1.00
	Tiling	165.700	1.00	5.12
	interchange	76.918	1.00	11.04
OpenMP	naive	133.674	6.35	6.35
	Tiling	16.086	10.30	52.83
	interchange	8.465	9.08	100.39
CUDA	naive	0.494	1720.33	1720.33

Table 1: Execution Time and Performance Gain.

Figure 5. Multiplication of matrices A and B of order 4096x4096. The execution time presented in Table 1 is also illustrated in the graph of Figure 5, where the axis represents the implementation used and the y axis presents the respective execution time for an order matrix 4096×4096.

The performance gains are directly associated with the optimal use of cache memory and its locality principles from the loop techniques and the number of processing cores used in each architecture. Block sizes in the tiling loop (16x16) were previously tested to present the best test case in this research. In the case of GPUs, the best use of the architecture was using blocks of size 32×32, equivalent to up to 1024 threads per block.

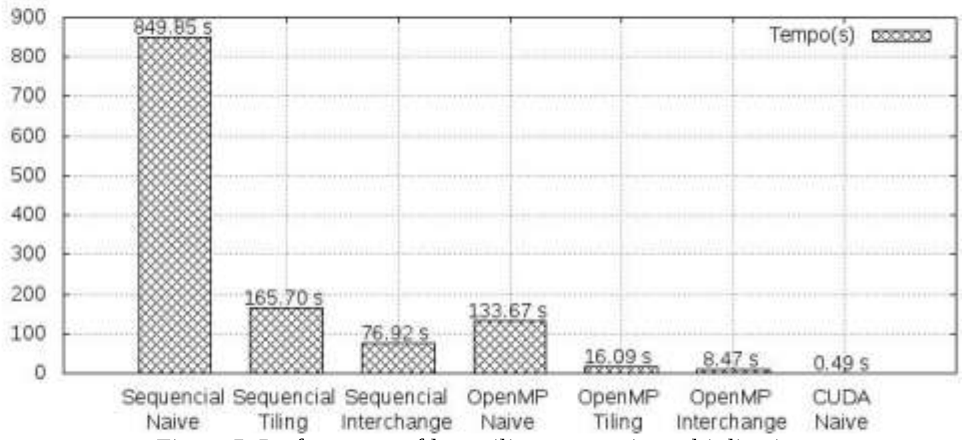

Figure 5. Performance of loop tiling on matrix multiplication

IV. CONCLUSION

Due to the need for faster computations to provide acceptable runtime results, loop optimizations and parallel execution in multi-core and many-core environments are necessary. Looping techniques improve the use of cache memory data, and parallel execution on multiple processing cores makes it possible to compute data concurrently. The sum of these techniques enables high performance gains.

The multi-core and many-core architectures allow parallel programming and task distribution to each core according to the problem's needs. The allocation of tasks and definition of cores to be used is up to the programmer to decide to provide the best way to obtain performance gains. Therefore, it is necessary that the programmer knows the architecture of the computational environment and knows how to use it in his favor since each application must be programmed and optimized for the target architecture.

REFERENCES

[1]. KIRK, DB; HWU, WW; Programming Massively Parallel Processors: A Hands-On Approach. 1st ed. San Francisco, CA, USA, 2010. Morgan Kaufmann Publishers Inc.

[2]. LEE, V. W.; et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. SIGARCH Comput. Archit. News, v.38, n.3, p.451-460, 2010.

[3]. MARQUEZAN, C.; et al. Análise de Complexidade e Desempenho de Algoritmos para Multiplicação de Matrizes. In: Escola Regional de Alto Desempenho (ERAD/RS). janeiro 2002; São Leopoldo; 2002. p.239-242.

[4]. NAVARRO, C.A.; et al. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. Communications in Computational Physics, v.15, n.2, p.285-329, 2014.

[5]. SCHEPKE, C.; et al. Online mesh refinement for parallel atmospheric models. International Journal of Parallel Programming, v.41, n.4, p.552-569, 2013.

[6]. Basel A. Mahafzah, "Performance evaluation of parallel multithreaded A* heuristic search algorithm." journal of Information Science 40, no. 3 (2014): 363-375.

[7]. Basel A. Mahafzah, "Parallel multithreaded IDA* heuristic search: algorithm design and performance evaluation." International Journal of Parallel, Emergent and Distributed Systems 26, no. 1 (2011): 61-82.

[8]. Al-Adwan, Aryaf, Basel A. Mahafzah, and Ahmad Sharieh. "Solving traveling salesman problem using parallel repetitive nearest neighbor algorithm on OTIS-Hypercube and OTIS-Mesh optoelectronic architectures." The Journal of Supercomputing 74, no. 1 (2018): 1-36.

[9]. S. Dutta, S. Manakkadu, and D. Kagaris. "Classifying performance bottlenecks in multi-threaded applications." In 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, pp. 341-345. IEEE, 2014.

[10]. S. Manakkadu, and S. Dutta. "Bandwidth based performance optimization of Multi-threaded applications." In 2014 Sixth International Symposium on Parallel

Architectures, Algorithms and Programming, pp. 118-122. IEEE, 2014.
[11]. Kavi, Krishna M., Roberto Giorgi, and Joseph Arul. "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation." IEEE Transactions on Computers 50, no. 8 (2001): 834-846.
[12]. Islam, Mahzabeen, Marko Scrbak, Krishna M. Kavi, Mike Ignatowski, and Nuwan Jayasena. "Improving node-level mapreduce performance using processing-in-memory technologies." In European Conference on Parallel Processing, pp. 425-437. Springer, Cham, 2014.
[13]. T. Janjus, K. Krishna, and B. Potter. "International conference on computational science, iccs 2011 gleipnir: A memory analysis tool." Procedia Computer Science 4 (2011): 2058-2067.