Comparative Analysis of OpenACC, OpenMP and CUDA using Sequential and Parallel Algorithms

Cleverson Lopes Ledur, Carlos M. D. Zeve, Julio C. S. dos Anjos[1]
Universidade Luterana do Brasil
Information Systems
BR 116, n. 5.724, Moradas da Colina - Guaba/RS
[1]Federal University of Rio Grande do Sul - UFRGS
Informatics Institute - INF
Parallel and Distributed Processing Group - GPPD
Av. Bento Gonalves, 9500, Porto Alegre, Brazil
ledur@null.net, [1]jcsanjos@inf.ufrgs.br, carlos.zeve@gmail.com

Abstract

With the increased processing required in the last years, and the search for devices with better performance, started in computing a need to parallelize processing, making it possible to support the performance of software and algorithms requiring high processing pattern. It's possible to use the processing power of devices like the GPU to run parallel software with a better execution time. In this work, will be evaluated the performance of three programming parallel models using CUDA, OpenMP and OpenACC with three different applications.

1. Introduction

With the evolution of technology and the constant need to provide more resources for users, there is the possibility of using all hardware resources available for the software development. GPUs have a different design philosophy of CPUs, shaped by the rapid growth of the video game industry which exerted enormous pressure on the ability to perform floating-point calculations for massive frame in video games. This has motivated manufacturers of GPUs casts them focused in the area of floating-point calculations. It's possible develop parallel programs to run on GPUs using APIs and parallel programming models such as CUDA, OpenMP and OpenACC. These APIs will be used and analyzed in order to obtain results for a comparison of speedup and runtime.

2. Parallel Programming

Parallel Programming consists in the uses of multiple processors to perform different parts of the same program simultaneously. The main objective would be to reduce the total execution time of processing [2].

Two of the main reasons for using parallel programming would be to reduce the time required to troubleshoot and solve complex and larger problems.

Parallel computing allows us to take advantage of computing resources not available locally or underutilized, overcoming limitations of memory when the memory available on a single computer is not enough to solve the problem and overcome the physical limits of speed and miniaturization that currently restrict the ability to start construction of sequential computers increasingly faster [3].

2.1. Parallel Architectures

Current architectures with high number of processing cores in parallel offer new software needs. A sequential computer software has to provide a sequence of operations to perform in the processor.

The parallel programming provides a sequence of operations on each processor to perform the task executions on parallel, including operations that coordinate and integrate the separate processors in a coherent computing. This need for coordination of activities parallel computing requires a new dimension to the programming of computer processes. Algorithms for specific problems should be reformulated in a way that creates processing operations to be performed on different processors [3].

There are currently two main architectures in parallel computing:

Shared memory all the individual processors have access to a common shared memory, allowing the shared use of multiple data values and data structures stored in memory.

Passage memory each processor has its own local memory, and processors share data by passing messages to each other via a communication network.

One of the main difficulties of shared memory architecture is memory contention. When many processors try to access the shared memory in a short period of time, the memory will not be available to accommodate all requests simultaneously, and some processors will have to wait until others receive the processing results.

There are some techniques that help to reduce memory contention and make the system more efficient. One such technique is to set a local cache in each processor, which is used to keep a recent copy of values used in memory. Another technique for reducing memory contention is the division of shared memory into separate modules that can be accessed in parallel by different processors. The shared information is spread by individual memory modules, reducing the probability of simultaneous access to the same memory by multiple processors.

2.2. GPU Programming

CPU and GPU is a powerful combination since the CPU consist of cores optimized for some serial processing, while the GPU consist of thousands of smaller cores designed for parallel performance. Serial parts of the code are executed by the CPU while the parallel parts are executed by GPU.

GPU computing is the use of a graphics processing unit together with a CPU to accelerate general purpose applications in science and engineering [1].

The GPU offers unprecedented performance for applications to transfer the processing intensive parts of the application to the GPU, while the rest of the code is still being executed by the CPU. From a user perspective, the application just run with a speed significantly better.

GPU is also used in complex mathematical and geometric calculations, due to its ability to process vectors or matrices with extreme efficiency. This great speed and power for mathematical calculations come from the fact that modern GPUs possess more processing circuits with data caching and flow control to the CPU. More specifically, the GPU has been especially designed to resolve problems which may be expressed in terms of the computations with multiple parallel data - the same program is executed for each data element in parallel - with high need for arithmetic [4].

Because the same program is executed for each piece of data, there is little need for a unit of flow control sophisticated, and because it runs on various data in parallel and has high request to calculations, the latency of memory access can be hidden calculations instead of caches giants.

3. Programming Languages

3.1. CUDA

CUDA is a computing platform and a model of parallel programming that enables dramatically increasing the performance of using the computing power of the graphics processing unit (GPU). The idea behind the language is that developers have available the powers of the graphics processing unit (GPU) to perform massive operations or not their software faster than using the CPU.

The processing flow for CUDA is not so complex. Data is copied from main memory to the graphics processing unit. After that, the processor allocates the process to the GPU, which then performs the tasks simultaneously in their core. After that, the result is the opposite, it is copied from the GPU memory to the main memory.

3.2. OpenACC

OpenACC is an API (Application Programming Interface) that provides a set of compilation directives, runtime libraries, and environment variables that can be used to write parallel programs in Fortran, C and C + + to run on devices accelerators, including GPUs. It was initially developed by the Portland Group (PG), Cray Inc., and supported by NVIDIA also the CAPS enterprise.

There are some facilities that have OpenACC compared to CUDA, since he abstains from the developer details such as data transfer between the host and the memory of the device, temporary data storage, kernel boot time mapping of threads and parallelism. OpenACC allows the programmer to write the code in a way that if the parallelism is ignored by the developer, it can continue with the same code without changes in the final result.

One big difference that has OpenACC over CUDA, is the use of compilation directives that facilitate the parallelization of the code. Thus, developers can start writing their algorithms sequentially and subsequently introduce directives OpenACC in the algorithm. It's like to give hints for the compiler turn the code parallel.

3.3. OpenMP

The OpenMP standard was developed and it is maintained by the group OpenMP Architecture Review Board

(ARB) formed from some Big companies such as SUN Microsystems, SGI, IBM, Intel and others, that in the end of 1997, gathered force to create a standard parallel programming for shared-memory architectures.The OpenMP API and focuses on a set of directives that supports the creation of parallel programs with shared memory through the implementation of an automatic and optimized set of threads. Its features can now be used in languages Fortran 77, Fortran 90, C and C + +.

The advantages of using OpenMP can be displayed on simplicity and little change in the codes, the robust support for parallel programming, ease of understanding and use of directives, one support nested parallelism and the possibility of dynamic adjustment of the number of threads used.

4. Metodology

The general goals of this work is to carry out, based on tests and observations, the comparison between an application developed sequentially, the same application ported to CUDA, OpenACC and OpenMP. It is possible to analyse the differences of time executions among the application developed sequencially about CUDA, OpenACC and OpenMP applications.

A computer with the following hardware configurations will be used to run the applications:

Motherboard ASUS M5A78L-M

Processor AMD Athlon(tm) II X2 270 Processor, 2048 KB L2 Cache, Ext Clock 200 MHz

System Memory DIMM0 2048, DIMM1 2048

GPU NVIDIA GeForce GTX 650

OS Ubuntu 12.04

Another issue that will be discussed with the data is that the details of programming abstraction that allows OpenACC, will impact negatively or positively on the results. The specific objectives of the project are:

1. Evaluate the complexity of programming and portability of the application between languages.
2. Evaluate the runtime of the application.
3. Evaluate the runtime of the application in CUDA.
4. Evaluate the runtime of the application in OpenACC.
5. Evaluate the runtime of the application in OpenMP.
6. Evaluating the Speedup of the application.
7. Evaluating the Speedup of the application in CUDA.
8. Evaluating the Speedup of the application in OpenACC.
9. Evaluating the Speedup of the application in OpenMP.

4.1. Comparison Details

Facing the possibility of parallelizing software to use the resources of the GPU, three models were chosen to allow this implementation to be carried out a comparative analysis of application implemented in parallel and sequentially in order to obtain results Speedup and Runtime.

It was implemented three application that require a high interaction processing and the calculation. These application are:

Mandelbrot set The Mandelbrot set is just a set of points in the Argand plane. The application create a image file and set for each pixel a color in order to create the fractal.

N-Queens In the general n-queens problem, a set of n queens is to be placed on an n x n chessboard so that no two queens attack each other.

Matrix Multiplication matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix.

These application were implemented sequentially in C and later in Cuda, OpenACC and OpenMP in parallelized form. Each code were compiled and executed ten times to produce a execution time average and to be possible to compare with other codes. With the compilation of algorithms and execution of tests using the Time command in Linux before the calling command for the binary file, it was possible to generate the total execution time of the application.

4.2. Execution Time

The execution time of a program can be defined as the time from the first processor starts executing until the last processor to finish.

The computation time is the time spent in computing, excluding communication time and idle. Downtime arises when a processor runs out of tasks, which can be minimized with a proper load distribution and overlapping computation with communication.

The communication time is the time that the application spends to send and receive messages.

The runtime can be decomposed in the computation time, communication and idle time.

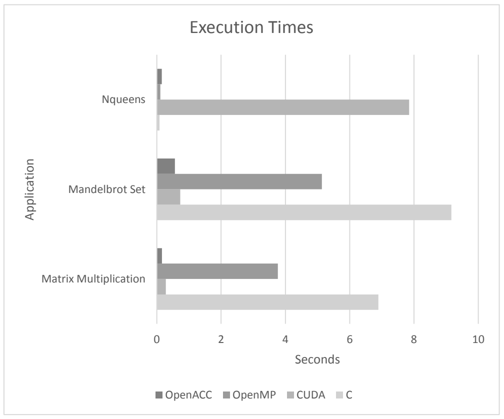
4.3. Speedup

Speedup is the speedup observed when performing a particular process p processors in relation to the implementation of this process in one processor.

5. Evaluation

Will be explained in the next subsections the results for each different code tested. Each application ran ten times and was made an average of execution times.

Below are the results of tests performed in sequential and in the parallel languages for the three applications.



Graph 1 - Execution Time Results

Matrix multiplication algorithm is easy to be parallelized because your code is simple and easy of understanding. When parallelized, he earns a lot at runtime, since it is just a simple mathematical operation that is performed on each processing core of the GPU.

In this test, were used two arrays of 834x834 for the calculation.

In the results, can be considered that the sequential code takes a long time to calculate the matrix multiplication, once there are many repeated operations to realize. In the parallel code, this operations are realized in many cores in the same time. The faster execution it have in this test is from OpenACC. Mandelbrot Set also has ease of parallelization, since the calculations to define the color of each pixel fractal can be done in parallel, because it has no dependence on the data being processed simultaneously.

In this test, was created an image of 4096 x 4096 pixels by the algorithm.

Like in the Matrix Multiplication, the Mandelbrot Set is a good algorithm to parallelize because it runs a matrix to set the pixel color relative to the position. In this case, it was found a better performance with the OpenACC algorithm as you can see in the Graph 1.

In this test, were used a recursive algorithm to calculate a board with 10 queens. There are many solutions for the nqueens problem, but in this tests were used just one recursive solution.

As in this application is used recursive code, the performance using CUDA is slow because the number or cores used to solve the problem is reduced. The CPU execution is faster in comparison with the other codes. The big difference among the OpenMP, OpenACC and CUDA execution time, can be explained because the compiler ignore directives that will not produce good results in the parallel code.

5.1. Conclusions

Before the work presented, it can be concluded that parallel programming is increasingly present in the near future, not only in massive computing software, but also in systems of small and medium businesses to generate more speed and providing the programmer more options to exploit the hardware resources.

In the tests, OpenACC presented a excellent execution time compared with the other languages. CUDA presented goods execution times too, but the complexity to construct codes is bigger than OpenACC and OpenMP.

Developers who want to use the parallelism must change their programming paradigms to meet the needs that arise as speed and better performance software applications in order to increase the capacity calculations and processing possible.

References

[1] D. B. Kirk. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. 2013.

[2] S. d. L. Martins. Programacao paralela. (10), March 2013.

[3] P. S. Pacheco. *An Introduction to Parallel Programming*. University of San Francisco, 2013.

[4] J. Tortugo. Cuda: Modelo de programacao paralela. (24), March 2013.

[5] J. R. M. Viana. Programacao em gpu: Passado, presente e futuro. (24), March 2013.