SHS Web of Conferences **119**, 07002 (2021) https://doi.org/10.1051/shsconf/202111907002
QQR'21

Comparative study of the implementation of the Lagrange interpolation algorithm on GPU and CPU using CUDA to compute the density of a material at different temperatures

Youness Rtal[1*] and *Abdelkader* Hadjoudja[1]

[1]University of Ibn Tofail, Faculty of Sciences, Department of Physics, Laboratory of Electronic Systems, Information Processing, Mechanics and Energy, B.P 242 Kenitra, Morocco

Abstract. Graphics Processing Units (GPUs) are microprocessors attached to graphics cards, which are dedicated to the operation of displaying and manipulating graphics data. Currently, such graphics cards (GPUs) occupy all modern graphics cards. In a few years, these microprocessors have become potent tools for massively parallel computing. Such processors are practical instruments that serve in developing several fields like image processing, video and audio encoding and decoding, the resolution of a physical system with one or more unknowns. Their advantages: faster processing and consumption of less energy than the power of the central processing unit (CPU). In this paper, we will define and implement the Lagrange polynomial interpolation method on GPU and CPU to calculate the sodium density at different temperatures Ti using the NVIDIA CUDA C parallel programming model. It can increase computational performance by harnessing the power of the GPU. The objective of this study is to compare the performance of the implementation of the Lagrange interpolation method on CPU and GPU processors and to deduce the efficiency of the use of GPUs for parallel computing.

1 Introduction

The primary purpose of interpolation is to interpolate known data from discrete points. In this case, we can estimate the value of the function between these points. This estimation method can be extended and used in various domains, namely, the derivation and numerical integration of polynomials. Lagrange interpolation [6, 12] is a method that allows interpolating the different data points, like the temperature of each point, to calculate some physical quantities. When the number of issues, increases the calculation becomes more and more difficult to solve by the central processing unit (CPU) in terms of speed; of execution and rapidity. That is why we need processors that treat these physical problems in a very efficient way and minimizes the time of execution. These processors, are called graphic

* Corresponding author:youness.pc4@gmail.com

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

processing units (GPU). CUDA, as a high-level language, has changed the whole perspective of GPU programming. It has reinforced interest in accelerating tasks usually performed by general-purpose processors GPUs. Despite these languages, it is not easy to exploit these complex architectures efficiently. This problematic endeavour is mainly because of the rapid evolution of graphics cards. That is, each generation brings its share of new features dedicated to high-performance computing acceleration. The details of these architectures remain secret because of the manufacturers' reluctance to disclose their implementations. These new features added to GPUs the result, from manufacturers simulating different architectural solutions to determine their validity and performance. The complexity and performance of today's GPUs present significant challenges for exploring new architectural solutions or modelling certain parts of the processor. Currently, GPU computing is growing exponentially, including processing mathematical algorithms in physics such as Lagrange interpolation [6, 12], physics simulation [8], risk calculation for financial institutions, weather forecasting, video and audio encoding [1]. GPU computing has brought a considerable advantage over the CPU regarding performance (Speed and energy efficiency). Therefore, it is one of the most exciting areas of research and development in modern computing. The GPU is a graphics-processing unit that mainly allows us to execute high-level graphics, which is the demand of the modern computing world. The GPU main task is to calculate three-dimensional applications this, kind of computation is very complicated to realize if we use only CPU (central processing unit). The evolution of the GPU over the years requires a very specific programming language such as CUDA C to improve computing performance.

The CUDA architecture has multiple cores that work together to consume all the input from the application. The processing of non-graphical objects on GPUs is known as GPGPU, which allows for very complex mathematical. Operations to be performed in parallel to achieve the best performance. [3, 9]

This paper will define and implement the Lagrange interpolation method that interpolates the temperature of sodium Na at different points to calculate the density at each temperature Ti. For this, we use GPU and CPU processors and the CUDA C programming language from Nvidia. The objective of this study is to compare the performance the Lagrange interpolation method on CUDA and GPU processors and to explore the efficiency of GPU for parallel computing. The upcoming section of this paper is as follows: in section 2, we present the CUDA architecture and the steps to be taken to write the code in CUDA C. In section 3, we will define the Lagrange interpolation method to solve our problem. In section 4, we will present the material used and the results of this implementation.

2 The CUDA program architecture and the hardware used.

2.1 The CUDA program architecture

The CUDA environment is a parallel computing platform and programming model invented by NVIDIA [4]. It allows to significantly increase computing performance by exploiting the power of the graphics-processing unit (GPU). CUDA C or C++ is a well-suited and helpful programming language for computing algorithms in parallel. The principal objective of CUDA is to run multiple threads in parallel to increase computational performance. Typically, the higher the number of threads, the better the performance. All threads execute the same code, called kernel. Each thread knows by its ID address, and based on this ID address; it, determines the data elements it has to work on [7]. One part of CUDA C program is executed on host (CPU) and the other part on GPU device. In the host code, the implementation of the data parallelism phases is non-existent. In some cases, data parallelism

is low in the host code and very high in the device code. CUDA C is a unified source code that includes both host and peripheral code. The host code is a simple code compiled using only the standard C compiler. The device code is written using CUDA instructions for parallel tasks, called kernels. In some cases, kernels can be executed on the CPU if no GPU device is present; this feature is given by the emulation function. The CUDA SDK offers these features. The CUDA architecture consists of three essential parts, and divides the GPU device into grids, blocks and threads in a hierarchical structure, as shown Figure in 1. Since there are several threads in a block and several blocks in a grid, and several grids in a single GPU, the parallelism with such a hierarchical architecture is very crucial. [2, 5]

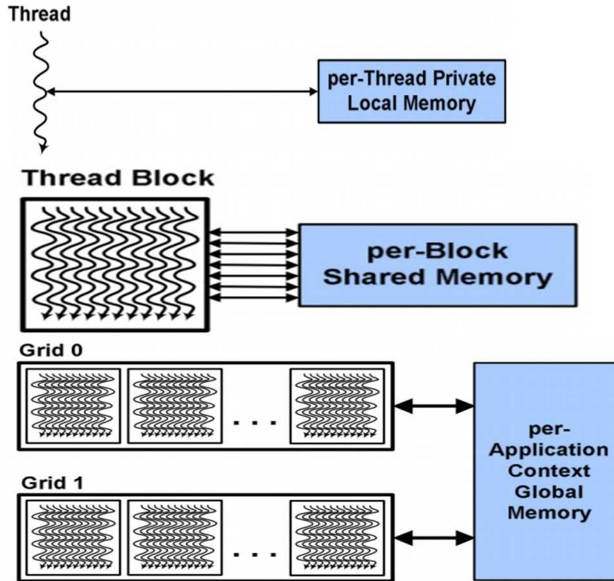

Fig.1. The architecture of the CUDA program and these memories.

A grid is composed of several blocks; each block contains some, threads. These threads are not synchronized and cannot be shared between different GPUs. Each grid consists of several blocks and each block contains several threads and a shared amount of memory. Blocks are no longer shared between multiprocessors. To identify the current block, we use the "blockIdx" instruction. The grids are composed of many threads that run on the different cores of the multiprocessors. There are around sixty-five thousand five hundred thirty-five blocks in a GPU. Each thread has its ID address called "threadIdx". Depending on the dimensions of the block, threads can be 1D, 2D or 3D. Threads have a certain amount of memory stored [10, 11]. In general, there are about 512 threads per block.
To write the program in CUDA C, you need to follow the following steps:

- Write the program in a standard C/C++,
- Modify the program written in CUDA C or C++ parallel code using the SDK library,
- Allocate CPU memory space and the same amount of GPU memory using the "CudaMalloc" function,
- Enter data into the CPU memory and copy this data to the GPU memory using the "CudaMemCpy" function with the "CudaMemcpyHostToDevice" parameter,
- Perform the processing in the GPU memory,
- Copy the final data to the CPU memory using the "CudaMemCpy" function with the parameter as "CudaMemcpyDeviceToHost",
- Free up GPU memory using the Cudafree function. [10]

2.2 The hardware used

The platform used in this study is a conventional computer, dedicated to video games and equipped with an Intel Core 2 Duo E6750 processor and an NVIDIA GeForce 8500 GT graphics card. All specifications for both platforms are available in [13, 14]. The processor is a dual-core, clocked at 2.66 GHz and considered entry-level in 2007.

The graphics card has 16 streaming processors running at 450MHz and was also considered entry-level in 2007. In terms of memory, the host has 2 GB, while the device has only 512 MB.

3 The Lagrange interpolation method and the code to implement

3.1 The Lagrange interpolation method

Let be $n + 1$ real xi discrete points and $n +1$ real y_i there is a single polynomial $p \in P_n$ such as $p(x_i) = y_i$ for $i = 0$ à n the construction of p is:

$$p(x) = \sum_{i=0}^{n} y_i L_i(x) \qquad (1)$$

With Li represents a Lagrange polynomial [6] where:

$$L_i(x) = \prod_{j=0;i\neq j}^{n} \frac{(x-x_j)}{(x_i-x_j)} \qquad (2)$$

Lagrange polynomial (2) depends on Xi and has the following properties:

$$L_i(x) = \begin{cases} 1; & i = j \\ 0; & \text{otherwire} \end{cases} \qquad \text{With, } i, j = 0, \dots, n$$

The error produced $\Phi(x)$ in the Lagrange interpolation can be helpful to control the quality of the approximation [6]. If f is $n + 1$ derivable on $[a, b]$, $\forall x \in [a, b]$ we note:

- I the most minor closed interval containing x and the x_i
- $\Phi(x) = (x - x_0)(x - x_1) \dots .(x - x_n)$

So, there are $\xi \in I$ such as:

$$e(x) = \frac{f^{(n+1)}}{(n+1)!} \Phi(x) \qquad (3)$$

In this paper, we will use formula (2) to interpolate the temperature of sodium at different points. The main objective of this Lagrange interpolation is to calculate the Ri density and implement these results on GPU and CPU processors using CUDA C to compare the performance of the implementation. Table 1 below represents the sodium densities in some Ti temperatures.

Table 1. Sodium densities at different temperatures.

Point i	1	2	3
Temperature T in (°C)	94	205	371
Density R(T) in (kg/m3)	929	902	860

From the data in the table, the number of points is equal to three; the Lagrange polynomial (1) will be of degree 2. This polynomial written:

$$R(T) = \sum_{i=1}^{n} R(T_i).\prod_{j=1,j\neq i}^{n} \frac{(T-T_j)}{(T_i-T_j)} \quad (3)$$

3.2 The Lagrange interpolation method

The code below represents the Lagrange polynomial interpolation algorithm to be implemented on GPU and CPU processors using CUDA C to calculate formula (4) at different temperatures Ti;

```
T = [94, 205, 371];
R = [929, 902, 860];
Ti = a;
Ri = lag(T,R, Ti)
function Ri = lag(T,R, Ti)
Ri = zeros(size(Ti));
n = length(R);
for i = 1 to n
    z = ones(size (Ti));
    for j = 1 to n, j ≠ i;
        z = z.∗ (T − Tj) /(Ti − Tj);
    end
    Ri = Ri + z ∗ R(i)
end
```

4 Results and discussion

The measurements are made over implementing the Lagrange interpolation method to calculate the density at different temperature points Ti. The unit of measure for execution time in milliseconds. The performance of the program implementation written in CUDA C of the Lagrange interpolation on GPU and CPU processors to calculate Ri at different temperatures are grouped in Table 2:

Table 2. Results of the implementation of the Lagrange interpolation method on CPU and GPU.

<table>
<tr><th>Temperature Ti (°C)</th><th>Density R_i in (kg / m^3)</th><th>CPU Time T_S in (ms)</th><th>GPU Time T_P in (ms)</th></tr>
<tr><td>100</td><td>927.56</td><td rowspan="6">8.32</td><td rowspan="6">0.25</td></tr>
<tr><td>150</td><td>915.48</td></tr>
<tr><td>200</td><td>903.23</td></tr>
<tr><td>251</td><td>890.55</td></tr>
<tr><td>305</td><td>876.93</td></tr>
<tr><td>800</td><td>742.45</td></tr>
</table>

The results grouped in Table 2 show the execution time on both CPU and GPU. It is noticeable that when the temperature increases, the Ri density decreases, and the execution time on the CPU is greater than the execution time on the GPU. The results of this implementation can be explained by the fact that the CPU process the data sequentially (task by task). In contrast, the GPU processes the data in parallel (several studies simultaneously), which implies the efficiency of the GPU processors for parallel computing.

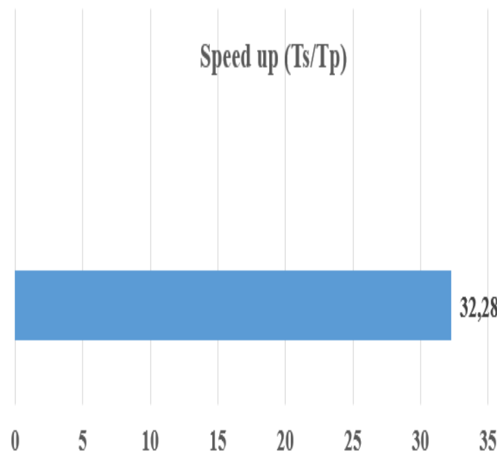

Fig.2. Speed up of the implementation of the Lagrange method.

In parallel computing, the Speed up gives an idea about the execution speed of a similar algorithm compared to a sequential algorithm. In our case, Speed up = execution time on CPU / execution time on GPU. Fig.2 shows that the Speed up of this implementation is 32.28; this, value depends on the execution time on GPU and CPU and the error defined in

the algorithm (3). This shows that the calculation by GPU is more efficient than CPUs in terms of Speed and energy efficiency. This optimality results from a good choice of the size of the block used and depending on the number of processors in the graphics card.

5 Conclusion

In the technological evolution, computers and instruments integrate into, their configurations graphic processors GPUs. These graphics processors are characterized by significant computing power; this, computing power is intended for programs dealing with graphical and non-graphical data, such as solving problems in physics, mathematics… etc. In this paper, we have successfully demonstrated the implementation of the Lagrange interpolation method using CUDA C to calculate the sodium density at different temperatures. As a result, we found that the performance of GPUs outperforms CPUs in terms of speed; and Speed of execution, which shows the efficiency of using GPUs in parallel computing.

References

1. "CUDA C programming guide version 6.5", NVIDIA Corporation, August 2014.
2. Manish Arora, "The Architecture and Evolution of CPU-GPU Systems for General Purpose-Computing ".
3. David Tarditi, Sidd Puri, Jose Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses", October 2006.
4. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0, 2008.
5. Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar, "GPGPU PROCESSING IN CUDA ARCHITECTURE" Advanced 12 Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012.
6. J.-P. Berrut and H. Mittelmann, Lebesgue constant minimizing linear rational interpolation of continuous functions over the interval, Comput. Math. Appl., 33 (1997).
7. Wikipedia- http://en.wikipedia.org/wiki/CUDA.
8. CalleLedjfors, "High-Level GPU Programming", Department of Computer Science Lund University.2008.
9. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron, "A Performance Study of General-Purpose Applications on Graphics Processors Using-CUDA".
10. Yadav K., Mittal A., Ansari M. A., Vishwarup V., "Parallel Implementation of Similarity Measures on GPU Architecture using CUDA".
11. Anthony Lippert - "NVIDIA GPU Architecture for General Purpose Computing".
12. W. Werner, Polynomial interpolation: Lagrange versus Newton, Math. Comp., 43 (1984), pp. 205– 217
13. http://ark.intel.com/Product.aspx?id=30784.
14. http://www.nvidia.com/object/geforce_8500.html.