International Journal of
Computer & Software Engineering

Research Article | Open Access

Special Issue: Internet of Things

Comparative Study of the Performance of the Lagrange Implementation on GPU and CPU Using CUDA

Youness Rtal* and Abdelkader Hadjoudja

Department of Physics, Laboratory of Electronic Systems, Information Processing, Mechanics and Energy, Faculty of Sciences, Ibn Tofail University, Kenitra, Morocco

Abstract

GPUs (Graphics Processing Units) are microprocessors attached to graphics cards dedicated to the display and manipulation of graphics data. In a few years, these microprocessors (GPUs) have occupied all modern graphics cards and become very important tools for massively parallel computing. These processors are practical tools for the development of several areas such as image processing, video and audio coding and decoding, solving a physical system with one or more unknowns... Their advantages: faster processing and lower energy consumption than the power of the central processing unit (CPU). In this paper, we will define and implement the Lagrange interpolation method on GPU and CPU to compute the density of a metal at different Ti temperatures using the CUDA C parallel programming model from NVIDIA which is used to increase the computational performance by exploiting the power of the GPU. Our goal is to compare the performance of the Lagrange interpolation method implementation on CPU and GPU processors and to infer the efficiency of using GPUs for parallel computing.

Publication History:

Received: May 31, 2021
Accepted: June 28, 2021
Published: June 30, 2021

Keywords:

GPU, CPU, CUDA, Lagrange Interpolation, Parallel computing

Abbreviations

CPU: Central Processing Unit, GPU: Graphical Processing Unit, CUDA: Compute Unified Device Architecture

Inroduction

The main purpose of interpolation is to interpolate known data from discrete points. In this case, the value of the function between these points can be estimated. 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 which allows to interpolate the different data points like the temperature of each point, in order to calculate some physical quantities. When the number of points 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's 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 processing units (GPU). CUDA, as a high-level language, has changed the whole perspective of GPU programming. It has reinforced interest accelerating tasks usually performed by general-purpose processors GPUs. Despite these languages, it is difficult to exploit these complex architectures efficiently. This problematic endeavor 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 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 modeling 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 in terms of performance (speed and energy efficiency). Therefore, it is one of the most interesting areas 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's main task is to calculate 3D functions; these types of calculations are very complex to perform on the CPU (central processing unit).

The evolution of the GPU over the years aims towards a better floating-point performance.

CUDA architecture has multiple processor cores that work together to consume all the data provided in 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. The arithmetic power of the GPGPU is the result of its specialized computing architecture [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 of implementation of the Lagrange interpolation method on CUDA and GPU processors and conclude the efficiency of using GPUs for parallel computing.

***Corresponding Author:** Dr. Youness Rtal, Department of Physics, Laboratory of Electronic Systems, Information Processing, Mechanics and Energy, Faculty of Sciences, Ibn Tofail University, Kenitra, Morocco; E-mail: youness.pc4@gmail.com

Citation: Rtal Y, Hadjoudja A (2021) Comparative Study of the Performance of the Lagrange Implementation on GPU and CPU Using CUDA. Int J Comput Softw Eng 6: 166. doi: https://doi.org/10.15344/2456-4451/2021/166

The upcoming sections of this paper is as the following: 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.

CUDA Architecture and the Hardware Used

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 an extension of the C or C ++ programming languages for general computing. CUDA is well adapted and efficient for highly parallel algorithms. It is necessary to have multiple threads to increase the performance of the algorithms while running on the GPU. The higher the number of threads, the better the performance. The main idea of CUDA is to have thousands of threads running in parallel. All these threads execute the same code, called the kernel. All these threads are executed using the similar instructions and different data. Each thread knows its ID address, and based on this own ID address, it determines the data elements it has to treat. [7] A CUDA program consists of a few steps executed on the host (CPU) or a GPU device. In the host code, the implementation of the data parallelism phases in non-existent. In some cases, data parallelism is weak in the host code. In the device code, phases with high data parallelism are executed. A CUDA program is a unified source code that includes both the host and device code. The host code is a simple C code compiled using only the standard C compiler. It can be said to be an ordinary CPU process. The device code is written using CUDA keywords for parallel tasks, called kernels and their associated data structures. In some cases, kernels can be run on the CPU if no GPU device is available, but this functionality is provided using an emulation function. The CUDA SDK provides these features. The CUDA architecture consists of three essential parts, which help the programmer to efficiently use all the computing capabilities of the graphics card on the system in question. The CUDA architecture divides the GPU device into grids, blocks, and threads in a hierarchical structure, as shown in Figure 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].

A grid is a group of many threads running the same kernel. These threads are not synchronized. Each call to CUDA from the CPU is made through a single grid. On multi-GPU systems, grids cannot be shared between different GPUs as they use many grids for maximum efficiency. The grids are made up of many blocks. Each block is a logical unit containing several coordination threads and a certain amount of shared memory. The blocks are no longer shared between the multiprocessors. Each block in a grid uses the same program. A built-in variable, "blockIdx", can be used to identify the current block. Blocks themselves are made up of many threads that run on the individual cores of multiprocessors, but unlike grids and blocks, they are not limited to a single core; there are around 65,535 blocks in

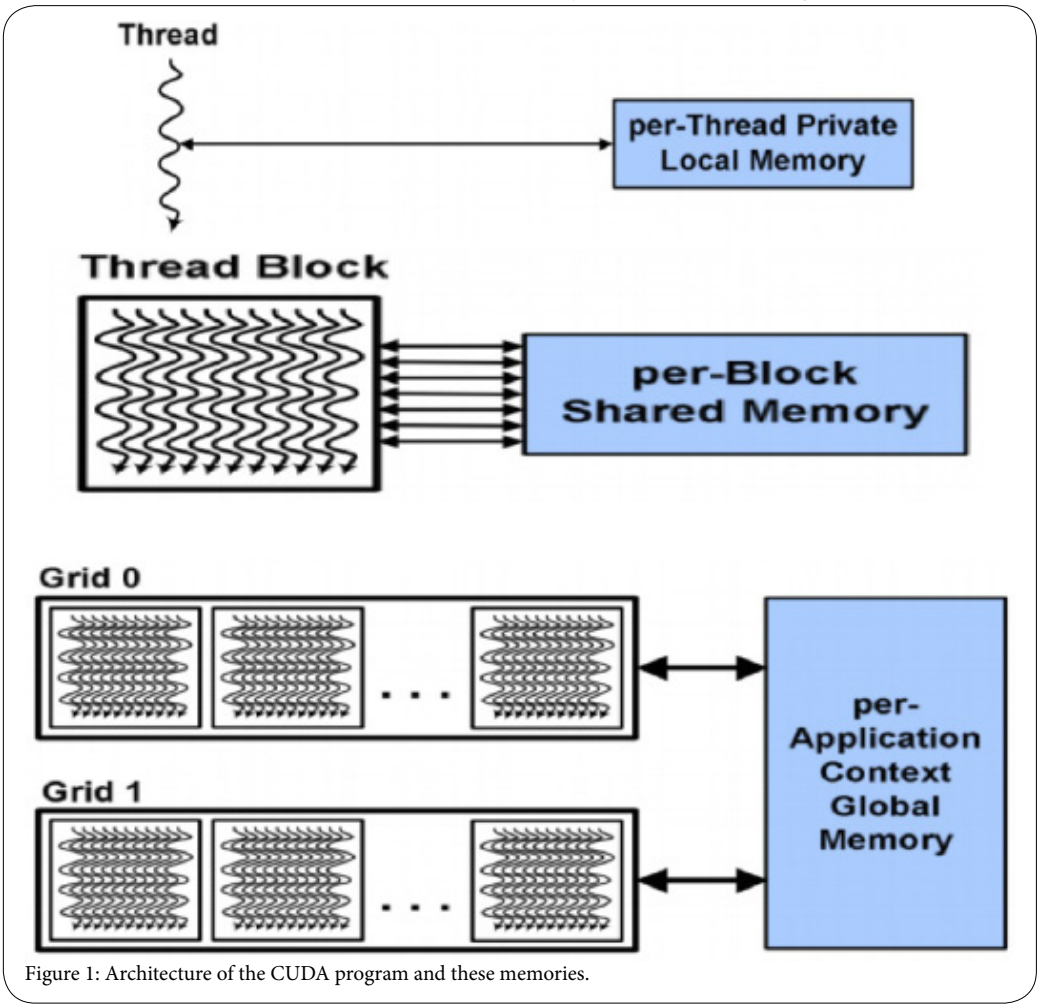

Figure 1: Architecture of the CUDA program and these memories.

a GPU. Like blocks, each thread has its ID called "threadIdx". Thread IDs can be 1D, 2D, or 3D, depending on the block dimensions. The thread ID is relative to the block in which it is located. Threads have a certain amount of registered memory [10,11]. Usually, there can be 512 threads per block.

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.

To write the program in CUDA C, you need to follow the following steps:

1. Write the program code in C / C ++.
2. Edit program written in CUDA parallel code using library functions provided by SDK, this library allows copying data from host to device and vice versa.
3. Allocate and enter data into CPU memory.
4. Allocate the same amount of GPU memory using the "CudaMalloc" library.
5. Copy the data to the GPU memory using the "CudaMemCpy" library with the "CudaMemcpyHostToDevice" procedure.
6. Process data in GPU memory using kernel calls. Kernel calls is a way to transfer this data from the CPU to the GPU by specifying the number of grids, blocks and threads.
7. Copy the final data into the processor memory using the CudaMemCpy library with the "CudaMemcpyDeviceToHost" procedure.
8. To Free up memory from GPUs or other threads using the "Cudafree" library [10].

The Lagrange Interpolation Method and the Code to Implement

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 L_i 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 x_i and has the following properties:

$$L_i(X_j) = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases} \quad i, j = 0, .., n.$$

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

1. I the smallest closed interval containing x and the x_i
2. $\phi(x)= (x-x_0)\ (x-x_1)\ldots\ (x-x_n)$

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

$$e(x) \quad \frac{(\ \ 1)(\ \ )}{(\ \ \ 1)!}\ (x) \qquad (3)$$

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 in order to compare the performance of the implementation, Table 1 below represents the sodium densities in some Ti temperatures.

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

Table 1: Sodium densities at different temperatures.

From the data in the Table 1 the number of points is equal to 3, then the Lagrange polynomial (1) will be of degree 2. This polynomial is written:

$$R(T) = \sum_{i=1}^{n} R(T_i).\prod_{j=1,\ j\neq i}^{n} \frac{(T - T_j)}{(T_i - T)_j} \qquad (4)$$

The list below represents the Lagrange polynomial interpolation algorithm program to be implemented on GPU and CPU processors using CUDA C to compute 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
```

Results and Discussion

The measurements are made over the execution time of the implementation of the Lagrange interpolation method to calculate the density at different temperature points T_i. The unit of measure for execution time is 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:

Temperature T_i(°C)	Density *Ri* in (kg / m³)	CPU Time T_s in (ms)	GPU Time T_p in (ms)	Speed up (T_s/T_p)
100	927.56	8.32	0.25	33.28
150	915.48			
200	903.23			
251	890.55			
290	880.73			
305	876.93			
500	826.01			
800	742.45			

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

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), while the GPU processes the data in parallel (several tasks simultaneously), which implies the efficiency of the GPU processors for parallel computing.

In parallel computing, the Speed Up gives an idea about the execution speed of a parallel algorithm compared to a sequential algorithm. In our case Speed up = execution time on CPU / execution time on GPU. Table 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 algorithm (3). This shows that the calculation by GPU is more efficient than CPUs in terms of speed and energy efficiency, this optimality is the result of a good choice of the size of the block used and depending on the number of processors in the graphics card.

Conclusion

More and more computers are integrating graphics processors or GPUs into their configurations, offering significant computing power. This computing power is intended exclusively for programs handling graphical and non-graphical data such as physics problem solving. However, we believe that we can use this computing power of GPUs in other ways. We have demonstrated in this paper successfully the implementation of the Lagrange method using CUDA C to calculate density at different temperatures and we have found that GPUs outperform CPUs in terms of execution speed which shows efficiency of performance. use of GPUs in parallel computing.

Competing Interests

The authors declare that they have no competing interests.

References

1. CUDA C programming guide version 6.5. NVDIA Corporation.
2. Arora M (2012) The Architecture and Evolution of CPU-GPU Systems for General Purpose-Computing.
3. Tarditi D, Puri S, Oglesby J (2006) Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. ACM SIGARCH Computer Architecture News 34: 325-335.
4. NVIDIA (2008) NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0.
5. Ghorpade J, Parande J, Kulkarni M, Bawaskar A (2012) GPGPU processing in CUDA architecture. Advanced 12 Computing: An International Journal.
6. Berrut JP, Mittelmann H (1997) Lebesgue constant minimizing linear rational interpolation of continuous functions over the interval. Comput Math Appl 33: 77-86.
7. Wikipedia.
8. Ledjfors C (2008) High Level GPU Programming. Department of Computer Science Lund University.
9. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, et al. (2008) A Performance Study of General-Purpose Applications on Graphics Processors Using-CUDA. Journal of Parallel and Distributed Computing 68: 1370-1380.
10. Yadav K, Mittal A, Ansari MA, Vishwarup V (2012) Parallel Implementation of Similarity Measures on GPU Architecture using CUDA.
11. Lippert A (2009) NVIDIA GPU Architecture for General Purpose Computing.
12. Werner W (1984) Polynomial interpolation: Lagrange versus Newton. Math Comp 43: 205-217.
13. http://ark.intel.com/Product.aspx?id=30784
14. http://www.nvidia.com/object/geforce_8500.html