

Implementing the Trapeze Method on GPU and CPU to Solve a Physical Problem using CUDA

Youness Rtal^a and Abdelkader Hadjoudja

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


Keywords: GPU, CPU, CUDA, Trapeze Method, Parallel Computing.

Abstract: Graphical Processing Units (GPUs) are microprocessors attached to graphics cards and dedicated to displaying and manipulating graphic data. Currently, such graphics cards GPUs occupy all modern graphics cards. In a few years, these microprocessors have become potent, tools for parallel computing. Such processors are practical instruments that develop several fields like video and audio coding and decoding, resolution of a physical system to one or more unknowns. Their advantages are faster processing and lower power consumption than the power of the central processing unit (CPU). This paper will define and implement the Trapeze method to solve a problem in physics that computes the execution time of a vehicle moving at a speed v using the CUDA C/C++ parallel computing platform. This is a programming model from NVIDIA that increases computational performance by exploiting the power of GPUs. This type of calculation can be helpful, to control the speed of vehicles by radars with precision. The objective, of this study, is to compare the performance of the implementation of the Trapeze method on CPU and GPU processors and deduce the efficiency of using GPUs for parallel computing.

1 INTRODUCTION

In most cases, the analytical functions, due to their complexities, are not analytically integral. In other patients, some parts are evaluated numerically at different points in the interval. We can obtain numerical approaches by the integrality of analytical functions. There are several integration methods, such as the trapezium method, Simpson's method, Newton-Cotes formulae and Gauss method. In this paper, we are interested in the frequently used trapezium method (Nadir, 2008) to solve physical problems using programming language and GPU processors. The emergence of high-level programming languages for graphics processing units (GPUs), has reinforced the interest in GPUs to accelerate tasks that work in parallel. Despite these new languages, it is difficult to use these complex architectures efficiently. Indeed, graphics cards are evolving rapidly, with each generation bringing its features dedicated to accelerating graphics routines and improving performance. The complexity and performance of today's GPUs present challenges

when exploring new architectural solutions or refining certain parts of the processor. GPU computing needs are increasing exponentially, including processing mathematical algorithms such as the trapezoid and Simpson algorithm (Nadir, 2008), physical simulation (CalleLedjfors, 2008), risk calculation for financial institutions, weather forecasting, video and audio encoding (NVIDIA Corporation, 2014). GPU computing has brought a massive, advantage over the CPU regarding performance (speed and energy efficiency). It is, therefore, one of the most exciting areas of research and development in modern computing. The GPU is a graphics-processing instrument that mainly allows executing graphics and calculating 3D functions. This kind of calculation is tough, to perform on the CPU (central processing unit), so GPUs can help programmers to work more efficiently because the evolution of the GPU over the years is oriented towards better performance. In 2006, NVIDIA introduced its massively parallel architecture called "CUDA" and changed the whole perspective of GPU programming. The CUDA architecture consists of

^a <https://orcid.org/0000-0003-0064-2233>

multiple cores that work together to handle all the data provided by the application. Using the GPU to process non-graphical objects is called GPGPU (general-purpose graphics processing unit), which performs highly complex mathematical calculations in parallel to achieve low power consumption and reduce execution time (David, Sidd, Jose, 2006; Shuai, Michael, Jiayuan, David, Jeremy W, Kevin, 2008).

In this paper, we consider a physical problem of a vehicle of mass m moving at speed V . This speed does not remain constant over time because of fluid friction. We will apply the second law of dynamics on the vehicle to calculate the distance travelled when its speed reduce from 30 m/s to 15 m/s using the trapezoidal method. We will implement this method on both CPUs and GPU using CUDA C/C++. The objective of this study is to examine the implementation of this method and to show the efficiency of using GPUs for parallel computing. This implementation may be helpful to control the speed of vehicles by precision. The upcoming section of this paper is as follows: in section 2, we present the architecture of the CUDA program. In section 3, we define the numerical integration method to solve the problem. In section 4, we will deliver, the hardware used and the results and discuss this implementation.

2 THE CUDA PROGRAM ARCHITECTURE

The CUDA environment is a parallel computing platform and programming model invented by NVIDIA (NVIDIA, 2008). It allows to significantly increase computing performance by exploiting the power of the graphics processing unit (GPU). CUDA C/C++ is an extension of the programming languages well suited and helpful for parallel algorithms. The main idea of CUDA is to have thousands of threads running 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 is characterized by its address ID. These threads execute using the exact instructions and different data (CUDA-Wikipedia). The program executed by CUDA consists of a few steps executed on the host (CPU) and a GPU device. In the host code, no data parallelism phases execute. In some cases, the data parallelism is weak in the host code (CPU) and strong in the peripheral (GPU) during execution. CUDA C or C++ is a platform for parallel computing that

includes host and peripheral code. The host code (CPU) is simple code compiled using only the C or C++ compiler, the device code (GPU) written using CUDA specific instructions for parallel tasks, called kernels. The kernels can be executed on the CPU if no GPU device is obtainable; this functionality is provided using a CUDA SDK function. The CUDA architecture consists of three main components, which allow programmers to use all the computing capabilities of the graphics card more efficiently. The CUDA architecture divides the GPU device into grids, blocks and threads in a hierarchical structure, as shown in Figure 1. Since several threads in a block and several blocks in a grid, and several grids in a single GPU, the parallelism resulting from a hierarchical architecture is very significant (Manish, 2012; Jayshree, Jitendra, Madhura, Amit, January 2012).

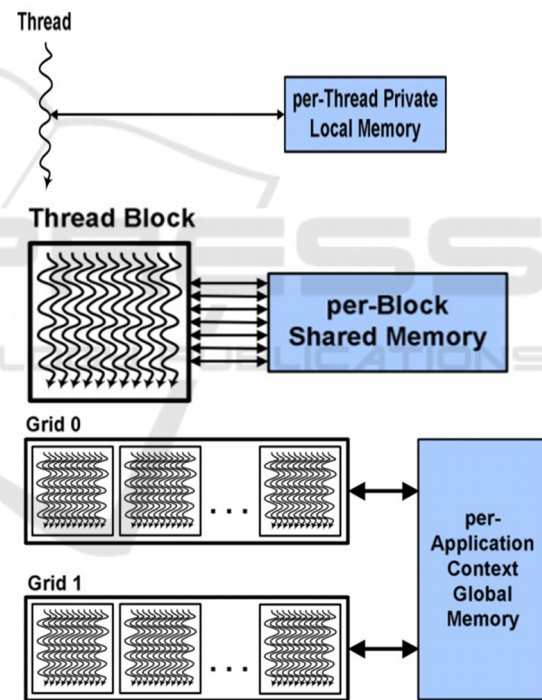


Figure 1: The architecture of the CUDA program and these memories.

A grid is composed of several blocks; each block contains several threads. These threads are not synchronized and cannot be shared between GPUs as they use many grids for maximum performance. Each call to CUDA by the CPU is made from a single grid. Each block is a logical unit containing multiple threads and some shared memory. Each block in a grid uses the same program, and to identify the current block number, the instruction "blockIdx" is

used. The blocks themselves contain are made up of threads that run on individual cores. Currently, there are about 65,535 blocks in a GPU. Each thread is characterised by its ID called "threadIdx". Thread IDs can be 1D, 2D or 3D, depending on the dimensions of the block. The thread ID is relative to the block in which it is located. Threads have a certain amount of registered memory (Anthony, 2009; Yadav, Mittal, Ansari M, Vishwarup, 2015). Usually, there can be 512 threads per block.

3 THE TRAPEZE INTEGRATION METHOD

The trapezoidal method (Nadir, 2008) consists of cutting the total area to be computed into small trapezoidal areas.

Either $f(x)$ the function to be integrated on $[a,b]$. The integral I from $f(x)$ written using the trapezium method (Nadir, 2008):

$$I = \int_a^b f(x).dx = \frac{h}{2} \cdot (f_1 + 2 \cdot f_2 + \dots + 2 \cdot f_i + \dots + 2 \cdot f_n + 2 \cdot f_{n+1}) + E$$

$$= \frac{h}{2} \cdot \left(f(x_1) + f(x_{n+1}) + 2 \cdot \sum_{i=2}^n f(x_i) \right) + E$$

Where: $h = \frac{b-a}{n}$, $x_i = a + (i-1) \cdot h$, $f_i = f(x_i)$ and $i = 1, 2, 3, \dots, n, n+1$

The term representing the error is:

$$E \approx -\frac{(b-a)}{12} \cdot h^2 \cdot \bar{f}'' \approx -\frac{(b-a)}{12 \cdot n^2} \cdot \bar{f}''$$

\bar{f}'' is the average of $f''(x)$ on the interval $[a,b]$. The error E is inversely proportional to the value of n^2 .

If the f given on regular intervals, $(x_i - x_{i+1} = h)$ the trapeze method can be written in the form:

$$I = h * \left(\text{sum}(f) - 0.5 * (f(1) + [f(\text{length}(f))]) \right)$$

With: $\text{sum}(f) = f_1 + \dots + f_i + \dots + f_n + f_{n+1}$

You can also make a program to calculate the integral I . This one called 'trapez_cu' for example, is listed below:

$$\text{function } I = \text{trapez_V}(f, h)$$

$$I = (\text{sum}(f) - (f(1) + f(\text{length}(f)))) / 2 * h;$$

In our study, we consider a mass vehicle $m = 2000$ kg moves at the speed of $V = 30$ m/s. Suddenly, the engine is disengaged at $t = 0$. The equation of motion after the instant $t = 0$ is given by:

$$m \cdot V \cdot \frac{dV}{dx} = -8,1 \cdot V^2 - 1200 \quad (1)$$

Where x represents the linear distance measured from $t = 0$.

In this equation (1), the term on the left represents the acceleration force, the 1^{er} the proper time describes the aerodynamic resistance exerted by the wind on the vehicle, the second term is the coefficient of friction. We want to calculate the distance beyond which the speed of the car is reduced to 15 m/s.

Equation (1), can be written as:

$$dx = -\frac{m \cdot V \cdot dV}{8,1 \cdot V^2 + 1200}$$

The integration of this equation gives:

$$x = -\int_{15}^{30} \frac{m \cdot V \cdot dV}{8,1 \cdot V^2 + 1200}$$

The trapezoid can evaluate the integral, if we give ourselves n intervals (or $n + 1$ dots), one can write:

$$V_i = 15 + (i-1)\Delta V$$

Where $i = 1, 2, 3, \dots, n+1$ and $\Delta V = \frac{45-15}{n} = \text{cte}$
By defining:

$$x_i = -\frac{m \cdot V_i}{8,1 \cdot V_i^2 + 1200}$$

Moreover, applying the trapezoidal method for integration, we obtain:

$$x \approx \Delta V \cdot \left(\sum_{i=1}^{n+1} f_i - 0,5 \cdot (f_1 + f_{n+1}) \right) \quad (2)$$

The following program list (trapeze.cu) allows calculating the value given by the formulation (2):

```
a = 15;
b = 30;
n = 1;
for k from 1 to 10
    n = 2 * n;
    h = (b - a) / n;
    i from 1 to n + 1;
        V = a + (i - 1) * h;
        f = 2000 * V / (8,1 * V.^2 + 1200);
        x = trapez_v(f, h)
    Error = abs(xexact - x) / xexact;
```

End.

To write and run the trapezoid program in CUDA C, follow these 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 using kernel calls. Kernel calls are a way to transfer control from the CPU to the GPU.
- 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. (Yadav, Mittal, Ansari M, Vishwarup, 2015)

4 EVALUATION OF THE PERFORMANCE OF THE IMPLEMENTATION

The results of "Table 1" show the evolution of the execution time on CPU and GPU processors as a function of the number of intervals n to calculate the distance x beyond which the car's speed is reduced by 30m/s à 15m/s by using the trapeze method. We notice that when the number of intervals n doubles, the solution x approaches its exact value and the relative error decreases to almost zero, and the

execution time on the GPU and CPU increases. The comparison of the execution time of the implementation on the GPU and the CPU tells us that the performance of the commission on the GPU is superior to that of the CPU in terms of speed and speed of execution. This implementation can be explained by the fact that the CPU processes data sequentially (task by task), while GPUs process data in parallel (several charges simultaneously), which implies the efficiency of using GPU processors for parallel computing.

4.1 Measurement Methodologies

The measurements function on the execution time of implementing the trapezoid method on both GPU and CPU processors. The unit of measure of the execution time is the millisecond. 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 (ark.intel.com; www.nvidia.com).

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 450 MHz and was also considered entry-level in 2007. In terms of memory, the host has 2 GB, while the device has only 512 MB. .

4.2 Results and Discussions

The performance of the implementation of the trapeze algorithm on GPUs and CPUs using CUDA C to calculate the distance travelled x are grouped in Table 1.

Table 1: Results of the implementation of the trapeze method on CPU and GPU.

Number of intervals n	The value x of distance in (m)	Percentage of relative error	CPU Time T_s in (ms)	GPU Time T_p in (ms)	Speed up (T_s/T_p)
2	127.58319	0.019111429	0.99	0.012	82.5
4	127.5258	0.00478998	1.59	0.021	75.71
8	127.51141	0.00119825	2.45	0.046	53.26
16	127.5078	0.00029965	4.01	0.091	44.06
32	127.5069	0.0007496	5.87	0.165	35.57
64	127.50662	0.00001878	8.36	0.25	33.44
128	127.50662	0.00000474	34.05	1.05	32.42
256	127.50661	0.00000123	141.47	4.42	32
512	127.506605	0.00000035	614.36	30.98	19.84
1024	127.50660	0.000000013	2517.57	253.48	9.94

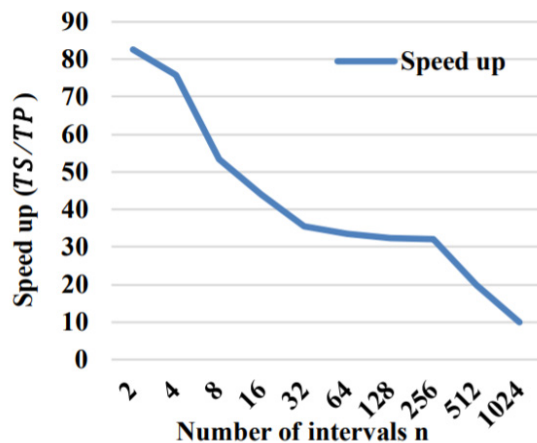


Figure 2: Evolution of Speed up as a function of the number of intervals.

In parallel computing, Speed Up refers to how faster a similar algorithm is compared to a corresponding sequential algorithm. In our case, Speed up = execution time on CPU / execution time on GPU. In our implementation, the Speed Up goes from 82.5 to 9.94 when the number of intervals n doubles and the solution tends to its exact value; it means that the performance of computing by GPU processors is faster than CPUs. This optimality results from a good choice of the size of the block used and according to the number of the processor in the graphics card used.

5 CONCLUSION

More and more computers integrate into, their configurations GPU graphics processors, which are characterized by significant computing power. This computing power is exclusively intended for programs manipulating graphical and non-graphical data, such as physical problems- solving. However, we believe that we can use this GPU computing power in other ways. In this paper, we have successfully demonstrated the implementation of the trapezoid method to calculate the execution time of posed problem. This method can be helpful, to control the speed of vehicles by precision, and we found that GPUs outperform CPUs in terms of execution time, which shows the efficiency of using GPUs in parallel computing.

ACKNOWLEDGEMENT

The authors would like to thank the referee for his valuable comments on the manuscript.

REFERENCES

- Anthony, L., 2009. NVIDIA GPU Architecture for General Purpose Computing.
- Manish, A., 2012. The Architecture and Evolution of CPU-GPU Systems for General Purpose-Computing.
- Shuai, C., Michael, B., Jiayuan, M., David, T., Jeremy W, S., Kevin, S., 2008. A Performance Study of General-Purpose Applications on Graphics Processors Using-CUDA.
- Jayshree, G., Jitendra, P., Madhura, K., Amit, B., 2012. GPGPU PROCESSING IN CUDA ARCHITECTURE. Advanced 12 Computing, An International Journal (ACIJ), Vol.3, No.1.
- Nadir, M., 2008. Full Equations Course. University of Milan, Algeria.
- David, T., Sidd, P., Jose, O., 2006. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses.
- Yadav, K., Mittal, A., Ansari M, A., Vishwarup, V., 2015. Parallel Implementation of Similarity Measures on GPU Architecture using CUDA.
- NVIDIA CUDA Compute Unified Device Architecture- Programming Guide, Version 2.0, June 2008.
- CalleLedjfors, 2008. High-Level GPU Programming. Department of Computer Science Lund University.
- NVIDIA Corporation, 2014. CUDA C programming guide version 4.2.
- Wikipedia- <http://en.wikipedia.org/wiki/CUDA>.
- <http://ark.intel.com/Product.aspx?id=30784>.
- http://www.nvidia.com/object/geforce_8500.html.