Applications of GPU Computing to Control and Simulate Systems

Miguel Romero Hortelano
Escuela Técnica Superior de Ingeniería Informática, UNED,
mromero@scc.uned.es

Abstract

This work deals with the new programming paradigm that exploits the benefits of modern Graphics Processing Units (GPUs), specifically their capacity to carry heavy calculations out for simulating systems or solving complex control strategies in real time.

Keywords: Control theory, GPU accelerators, high performance computing and physical simulation.

1 INTRODUCTION

The potential of graphics processors, which are known as GPUs, has demonstrated to be a valuable tool to carry out the most diverse scientific tasks where it is necessary a high computational power [9, 10]. These GPUs have become an integral part of mainstream computing systems [19] due to their larger capacity of handling huge calculations in comparison with traditional solutions based on CPUs (Central Process Units).

Traditionally, mathematical algorithms that represent the behaviour of physical systems had been implemented to run on the CPU, since classical computers had no other processing unit. Microprocessors based on CPU allowed giga floating-point operations per second (GFLOPS) to the desktop and hundreds of GFLOPS to cluster servers.

However, the most current computers integrate the GPU, which was born to absorb graphics tasks from the CPU. Initially, GPU implemented a limited number of graphics primitive operations to accelerate operations for drawing arcs, circles, rectangles and triangles. Today, this device can carry out complex geometric calculations such as the rotation, translation and manipulation of vertices in 3D. To do so, GPUs have evolved from fixed function rendering devices into programmable parallel processors architectures. Thus, the current GPU is not only a quick graphics engine but also an exceedingly parallel programmable processor [19] integrated by a streaming processing model giving rise to the concept of GPGPU (General-Purpose Graphics Processing Unit). The computing performance of these kind of processors are currently 100 TFLOPS [16], greatly exceeding the performance that a CPU-based architecture can offer.

The success of this concept is unquestionable, not only considering that principal manufacturers have even developed GPGPU accelerators for scientific and technical computing without graphics engine strictly speaking (i.e. Nvidia® Tesla® [16]) but also considering that small single-board computers such as Raspberry Pi integrate a GPU allowing to run more sophisticated control algorithms in real time within Internet of Things (IoT) environment.

It is well known that certain non-LTI control strategies depend on a high computing capacity to be applicable in a real time system. There exist many implementations; without the intention of being exhaustive; we can mention: Predictive control strategies with constraints [3, 2], where the optimal control law is obtained by minimizing a given cost function subject to a set of constraints at each sampling time. Adaptive control [12], where it is used a set of techniques for automatic adjustment of the controller in real time for achieving a certain system performance, etc.

On the other hand, simulation of physical phenomena has been ever a challenge for computer developers due to computer power needed to solve a physical model numerically [20]. From one of the first general purpose computers ENIAC (used to design ballistics tables to estimate where artillery shells will impact depending of projectile mass, wind, gunpowder charges, etc.) until nowadays modern computers which have been programmed to obtain accurate physical simulations for science, engineering, etc.

It is a fact that physical model simulation is an active and interesting research topic whose evolution is tightly linked to the development of rapid computer algorithms, which make possible to solve heavy numerical problems such as physical model simulations with infinite memory or hereditary phenomena. The fractional calculus is frequently used to mathematically describe this kind of phenomena, in view of its valuable characteristics for modelling infinite memory behaviours by means of

its fractional operators [6, 32]. It is in the field of the linear viscoelasticity where its practical application gains more relevance, replacing the ordinary derivatives in the classical models [26, 13].

Thus, the aim of this work will be to show some applications where the use of GPU computing paradigm supposes to take a step forward in terms of performance when a system is simulated or controlled. To do so, this paper has been structured as follows: In section 2 the GPU computing paradigms are described. Section 3 presents some applications of GPU computing applied to control and systems simulations. Finally, section 4 draws the main conclusions of this work.

2 GPU COMPUTING ESSENTIALS

High performance computing has found a strong pillar in GPU computing due to the performance gain achieved through new graphics processing units. Numerous researchers and GPU computing programmers have reported a speed increment between 10 and 100 times of their applications and algorithms using these new graphics processing units [9].

Early CPUs were designed to execute general-purpose applications integrating, generally, one processing unit per processor. This traditional architecture allows to run only one task at any point in time. Therefore, software applications were written as sequential programs attending to Von Neumann principles. During this period, execution speed relied on advances in hardware (CPU cache, branch prediction, increase in clock frequency, etc.) and backward compatibility.

Due to lack of heat dissipation in processor die, extreme transistor miniaturization (near theoretical limits) and energy consumption issues; processors vendors have had to leap to models with multiple processing units (processor cores) in order to increase processing power, departing progressively from ancient sequential programming principles [27]. Therefore, execution speed increments depend now on the simultaneous use of multiple compute resources to complete the work faster. A sequential program that continues running only on one of the processor cores is not a rival in terms of performance.

On the other hand, due to the need of getting more and more realistic scenes for professional environment and personal entertainment, customized effects came out by means of programmable *shaders* on graphics hardware. A *shader* is a program to determine the final surface properties of a 3D−object allowing to calculate rendering effects with a high degree of flexibility. With the arrival of the Microsoft® API, DirectX® 10, all types of *shaders* were unified using a common-*shader* core (see Figure 1). This unified *shader* model has permitted the GPU to evolve into a powerful programmable processor with enormous arithmetic capability, substantially greater than a high-end CPU of its time.

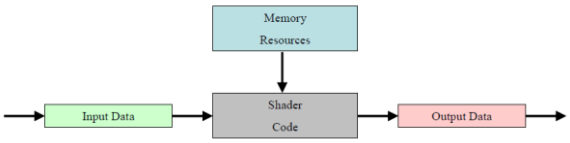


Figure 1: Data flow in a *shader* stage diagram.

In the following, we will show how to take advantage of this technology using the different available programming environments.

2.1 GPU COMPUTING SOFTWARE

GPU's hardware architecture enables massive parallel computing by means of platforms whose internal configurations depend on their manufacturers. Nowadays, the main GPU manufacturers are NVIDIA® (www.nvidia.com) and AMD® (www.amd.com). Both of them lead currently graphics card market and are staunch defenders of the GPGPU programming paradigm.

2.1.1 CUDA®

CUDA (Compute Unified Device Architecture) is a parallel computing platform with an application programming interface (API) designed and created by Nvidia for programming its GPUs. Therefore, it is incompatible with any other hardware from other manufacturers. There is support for most general-purpose programming languages such as C, C++, Python, etc. [17]

2.1.2 MATLAB® and SIMULINK®

The latest versions of Matlab allow to accelerate the execution of the programs created in this environment as long as the hardware is compatible with CUDA [23]. To do so, it is only necessary that the Matlab function accepts “gpuArray” input arguments. The list of supporting functions is so huge and includes functions to calculate the discrete Fourier transform, matrix manipulation, trigonometric calculations, etc. (For a complete list of supporting functions, see [14]). Therefore, if these functions are called with at least one gpuArray as an input argument, the specific function is executed on the GPU and automatically accelerated.
As far as Simulink is concerned, the number of system blocks supported is getting more and more

important, although up until now there are limitations for their use.

2.1.3 OPENCL™

OpenCL (Open Computing Language) is not only an open royalty-free standard language but a whole framework for parallel programming that permits writing programs which can be executed across heterogeneous platforms and accelerates parallel computation. This open standard is maintained by the Khronos Group [11] and it has been adopted by principal graphics hardware manufacturers: NVIDIA®, AMD®, INTEL®, etc. There is also support for most general-purpose programming languages such as C, C++, Python, etc.

2.1.4 DIRECT HARDWARE PROGRAMMING

Normally, it is an extremely complicated practice because of the little information about the programming at low level of the different GPUs and it is usually required much development time. This practice is only recommended when there are not available higher level programming interfaces as CUDA, OpenCL, etc. but the development is worth the effort.

For example, the small single-board computer, Raspberry Pi, was initially developed for computing teaching science purposes. However, due to its valuable hardware resources and low price, its success integrating all types of applications is unquestionable.

Raspberry Pi integrates a GPU together with a CPU, which is been used by developers to extract a performance peak. Although there is a good attempt to implement the OpenCL standard for the GPU of Raspberry Pi that is called VC4CL [30], it is not fully functional and its version (1.2) is a bit old. However, there exist general purpose codes developed freely by programmers to take advantage of GPU potential as [7], where the Fourier transform is accelerated and [22], where it is presented a GPU-accelerated implementation of matrix multiply function for Raspberry Pi, which can be applied on control and simulation.

Finally, Table 1 summarizes the main characteristics of the previous GPU programming environments as well as a comparison between them.

3 APPLICATIONS IN CONTROL AND SIMULATION

In the following, we will show some applications that expose the interest of programming these devices, GPUs, to control or simulate systems.

3.1 CONTROL APPLICATIONS

The recent presence of control strategies more and more sophisticated for non-LTI systems or the need of satisfy the increasing computational demands to control systems with a large number of states may require the help of intensive computing units in the control devices, if these ones are available. The LTI control systems in the state space have an equivalent matrix form (1)

$$\left.\begin{aligned} x(k+1) &= Ax(k)+Bu(k) \\ y(k) &= Cx(x)+Du(k) \end{aligned}\right\} \quad (1)$$

If the number of states of the LTI system (1) is large, and the state matrix A is dense, then it is necessary to compute the inverse of a sequence of large-scale dense matrices. In [1] the large-scale dense matrices inversion with application in control has been studied and optimized for computational systems with GPUs.

On the other hand, there exist many non-LTI control strategies that require solving complex mathematical problems in each sampling time. For example, the model predictive control (MPC) with constraints is based on the prediction of the future process outputs by means of minimization of a cost function within a time windows in each sampling time [25]. It leads to the resolution of a mathematical optimization problem known as quadratic programming (2).

$$\begin{aligned} &\min_x \ f(x) = a^T x + \frac{1}{2} xGx \\ &\text{subject to} \quad C^T x - b \geq 0 \end{aligned} \quad (2)$$

Following this line, several authors have proposed algorithms to compute this kind of problems using GPUs. For example, in [8] a way to accelerate the sequential quadratic programming on GPUs is introduced.

In our case, we have based on the algorithm developed by D. Goldfarb and A. Idnani [5] and its C++ implementation by L. Di Gaspero [4] to parallelize it and optimize it, using the GPU programming paradigm. It has been developed using the OpenCL API in C++.

Figure 2 shows the results obtained using our implementation in OpenCL in comparison with the implementation of L. Di Gaspero. One observes that the performance increment depends on N (number of variables). The larger this is, the larger the increment in performance is obtained. However, there not exists this performance increment when values of N below 1200 are used as it is shown on Figure 3, where the relationship between the performances of both implementation in function of N is illustrated.

Table 1 GPU Programming Environments Comparison

Programming Environment	Compatible Hardware	Learning Curve	Programming Language
CUDA	Only Nvidia hardware	Medium	Most general-purpose programming languages
OpenCL	Principal graphics hardware manufacturers	Medium	Most general-purpose programming languages
Matlab	Only Nvidia hardware	Low	Only Matlab language
Direct Hardware Programming	Only target hardware	High	Low-level languages as C and assembler

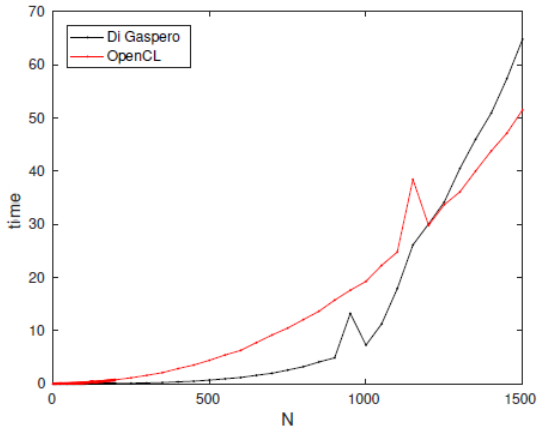


Figure 2: Execution time of quadratic programming implementations.

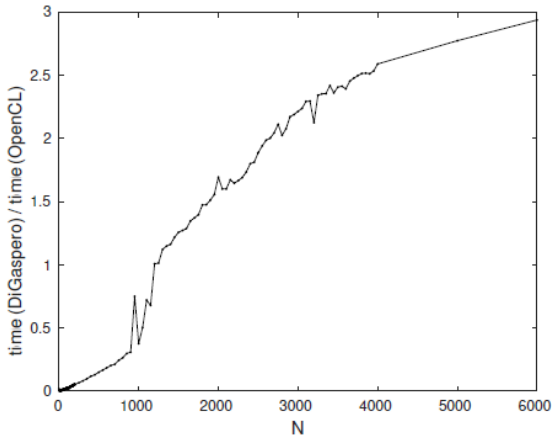


Figure 3: Performance comparison.

3.2 PHYSICAL SIMULATIONS

It is well known that realistic physical simulations need simulations models with complex algorithms which are very time-consuming. However, the high performance of GPUs combined with high levels of parallelism of calculations could achieve a dramatic reduction in the computation time for these simulations.

Using this idea, physical simulation applications have appeared taking advantage of this computational power increment. For example, in [31] it is presented a study to quantify the cost and accuracy benefits of using high-order unstructured schemes on GPUs for scale-resolving simulations of unsteady flows. In [21] the simulation of compressible fluids is dealt using different techniques to increment the performance in comparison with CPUs implementations.

On the other hand, the viscoelasticity is a physical property of materials, which has both viscous and elastic characteristics when they are undergone deformation. Due to this dualism, the mathematical models that represent it often require an intense numerical computation that is a tedious and time-consuming task. The viscous effect is represented by a purely viscous damper and the elastic effect is represented by a purely elastic spring (see Figure 4). Due to the resulting computational cost, many authors have opted to use the GPU resources to achieve the necessary performance. For example, in [28] it is used a new GPU finite element scheme of anisotropic viscoelasticity for soft tissue simulation using CUDA. A model for simulating elastic wave propagation using the Kelvin-Voigt model of viscoelasticity is proposed in [29], where the simulation is accomplished via Matlab using the GPU and the Parallel Computing Toolbox.

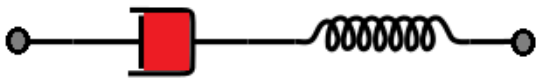


Figure 4: A viscoelasticity model.

Moreover, a new way to obtain more accurate models of viscoelasticity has arisen during the last years, which is based on the use of fractional calculus to describe this physical behaviour [13].

Fractional calculus is a mathematical branch that deals with derivatives and integrals of real or oven complex order [18]. Fractional order operators are usually represented using the notation D^{α}. Positive values of α correspond to fractional order derivates

and negative values of α correspond to fractional order integrals. The numerical evaluation of this operator is accomplished using the Grünwald-Letnikov differintegral −GL− (3) that is a generalization of the well-known definition of the first-order derivative:

$$D^{\alpha} f(t)_{t=kh} = \lim_{h \to 0} h^{-\alpha} \sum_{i=0}^{\infty} (-1)^i \binom{\alpha}{i} f(kh - ih) \qquad (3)$$

with h the sampling period.

The discrete function (3) has got an unlimited number of terms. Precisely, because of this specific feature, fractional calculus has become valuable tool for capturing and describing complex effects related to infinite memory behaviours (i.e. polymer viscoelasticity). Nevertheless, the so-called short memory principle [24] is generally used due to only the recent past plays an important role in evaluating D^{α}. It corresponds to n-term truncated series, paying a penalty in the form of some inaccuracy.

At the beginning, it is not necessary to parallelize using the GPU the computation of the binomial coefficients $(-1)^i \binom{\alpha}{i}$. This calculus can be done off-line using this well-known recursive algorithm [15]. However, it is necessary to parallelize the long sum of the previous *n*-terms binomial coefficients multiplied by the functions values *f(kh-ih)*. To do so, a program that calculates the dot product of two vectors (see Figure 5) is created using OpenCL.

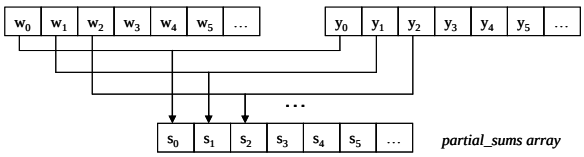


Figure 5: Computation of products.

In the following, it is highlighted the performance increment obtained using the previous implementation on GPU in comparison with a sequential implementation running on CPU. So, two computers with different graphics hardware (Nvidia 230M and Nvidia 560GTX) and quite similar CPUs have been used. Figure 6 shows the corresponding execution times of GL functions with different number of terms, *n*={5120, 51200, 512000}. It is observable the performance increment obtained around 2.5 times faster in the 230M case and 10 times faster in the 560GTX case in comparison with the sequential execution without using GPU optimization. Moreover, Figure 6 also illustrates the performance of 560GTX that is between 3.5 and 5 times faster than 230M. Obviously, better hardware means better performance.

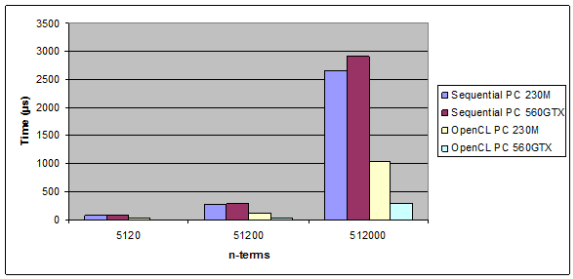


Figure 6: Execution time of the GL implementations.

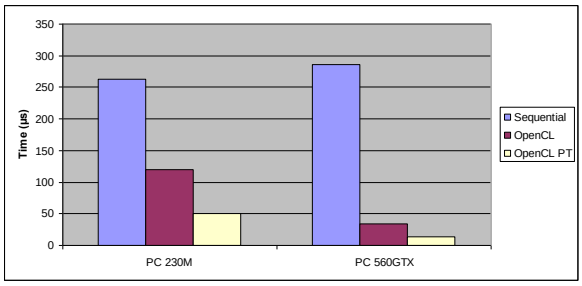


Figure 7: Execution times for comparison with PT.

Furthermore, we have adapted our source code using the programming methodology proposed by AMD, which is called "Efficient dot Product Implementation Using Persistent Thread" −PT−. So, Figure 7 presents the results for the case of *n* equal to 51200 terms with GPU code optimization using and not using PT, and sequential execution on CPU. One observes the performance increment of GPU with PT implementation is about 2.5 times faster than GPU without PT implementation.

4 CONCLUSIONS

In this paper, we have presented some applications that expose the benefits of GPU programming to obtain an important performance boost, because of the powerful programmable processors that the GPUs include with an enormous arithmetic capability, substantially greater than a high-end CPU.

It has also been shown two own developments. On the one hand, we have presented a implementation for GPU to resolve a quadratic programming problem for predictive control strategies using OpenCL and, on the other hand, we have shown other implementation for GPU, where a parallel algorithm has been developed to boost numerical evaluation of fractional operators, which represent the physical behaviour of viscoelasticity in a lot of models for simulation. In both cases, the advantages of using the GPU to perform the calculations have been demonstrated.

Acknowledgments

The author wish to acknowledge the work of students D. José A. Chico and D. Francisco de la Hoz and the GPU Grant Program of Nvidia for its support.

References

[1] Benner, P., Ezzatti, P., Ortí, Q., Enrique, S., and Remón Gómez, A. (2013). "Matrix inversion on CPU-GPU platforms with applications in control theory". Concurrency and Computation: Practice and Experience **25**(8). 1170-1182.

[2] Camacho, E.F., and Bordóns, C., (2004). Model Predictive Control. 2nd edition. Springer.

[3] Clarke, D.W., Mohtadi, C., and Tuffs, P.S., (1987). "Generalized predictive control. Part I. The basic algorithm". Automatica **23**(2), 137-148.

[4] Gaspero, L.D. (2018). QuadProg++. url: http://www.diegm.uniud.it/digaspero/index.php

[5] Goldfarb, D., and Idnani, A. (1983). "A numerically stable dual method for solving strictly convex quadratic programs". Mathematical Programming, **27**(1). 1-33.

[6] Hilfer, R., (2000). Applications of Fractional Calculus in Physics. World scientific publishing Co. Pte. Ltd. Singapore.

[7] Holme, A., (2018). FFT using GPU in Raspberry Pi url: https://www.raspberrypi.org/blog/accelerating-fourier-transforms-using-the-gpu/

[8] Hu, X., Douglas, C.C., and Lumley, R. (2017). "GPU accelerated sequential quadratic programming". Proceedings of the 16th DCABES Symposium. 3-6.

[9] Hwu, W., (2011). GPU Computing Gems Emerald Edition. Applications of GPU Computing Series. Elsevier Inc., USA.

[10] Hwu, W., (2012). GPU Computing Gems Jade Edition. Applications of GPU Computing Series. Elsevier Inc., USA.

[11] Khronos OpenCL Working Group. (2018). The OpenCL 2.2 specification, url: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf

[12] Landau, I.D., Lozano, R., M'Saad M., and Karimi, A., (2011). Adaptive Control. Algorithms, Analysis and Applications. 2nd edition. Springer.

[13] Mainardi, F. (2010). Fractional Calculus and Waves in Linear Viscoelasticity. An Introduction to Mathematical Models. Imperial College Press, UK.

[14] Matlab. (2018). "Run Built-In Functions on a GPU". url: https://es.mathworks.com/help/distcomp/run-built-in-functions-on-a-gpu.html#bsloua3-1

[15] Monje, C.A., Chen Y.Q., Vinagre, B.M., and Xue, D. (2010). Fractional-order Systems and Controls: Fundamentals and Applications. Springer-Verlag. UK.

[16] Nvidia. (2018). "Tesla Data Center GPUs for Server". url: https://www.nvidia.com/en-us/data-center/tesla/.

[17] Nvidia. (2018). "CUDA Zone". url: https://developer.nvidia.com/cuda-zone

[18] Oldham, K.B., and Spanier, J. (1974). The Fractional Calculus. Academic Press. New York.

[19] Owens, J.D., Houston M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C. (2008). "GPU computing". Proceedings of the IEEE **96**, 879-899.

[20] Pang, T., (2006). An Introduction to Computational Physics. Cambridge University Press, USA.

[21] Pekkila, J., Vaisala, M., Kapyla, M., Kapyla, P., and Anjum, O. (2017). "Methods for compressible fluid simulation on GPUs using high-order finite differences". Journal of Computer Physics Communications **217**, 11-22.

[22] Pi GEMM (2018). url: https://github.com/jetpacapp/pi-gemm

[23] Ploskas, N., and Samaras, N. (2016). GPU Programming in Matlab. Elservier Inc., UK.

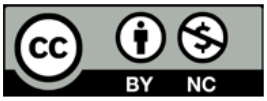
[24] Podlubny., I. (1999). Fractional Differential Equations. Mathematics in Science and Engineering. Vol. 198 Academic Press. USA.

[25] Romero, M., de Madrid, A.P., Mañoso, C., and Milanés, V. (2015). "Low speed hybrid generalized predictive control of a gasoline-propelled car". Isa Transactions **57**. 373-381.

[26] Stiassnie, M. (1979). “On the application of fractional calculus for the formulation of viscoelastic models”. Journal of applied mathematical modelling **3**(4), 300-302.

[27] Sutter, H., and Laurs, J. (2005). “Software and the concurrency revolution”. Journal ACM Queue **3**(7). 54-62.

[28] Taylor, Z.A., Comas, O., Cheng, M., Passenger, J. Hawkes, D.J., Atkinson, D., and Ourselin, S. (2009). “On modelling of anisotropic viscoelasticity for soft tissue simulation: Numerical solution and GPU execution”. Medical Image Analysis **13**, 234-244.



© 2018 by the author. Submitted for possible open access publication under the terms and conditions of the Creative Commons Attribution CC-BY-NC 3.0 license (https://creativecommons.org/licenses/by-nc/3.0).

[29] Treeby, B.E., Jaros, J., and Rohrbach, D. (2014). “Modelling elastic wave propagation using the k-Wave MATLAB toolbox”. Proceedings of the IEEE IUS, Chicago, USA, 146-149.

[30]VC4CL(2018).url:https://github.com/doe300/VC4CL

[31] Vermeire, B.C., Witherden, F.D., and Vincent, P.E. (2017). “On the utility of GPU accelerated high-order methods for unsteady flow simulations: a comparison with industry-standard tools”. Journal of Computational Physics **334**, 497-521.

[32] West, B.J., Bologna, M., and Grigolini, P. (2003). Physics of Fractal Operators. Springer-Verlag. USA.