Bulletin of the *Transilvania* University of Braşov
Series III: Mathematics and Computer Science, Vol. 3(65), No. 2 - 2023, 227-236
https://doi.org/10.31926/but.mif.2023.3.65.2.19

ADVANCES IN CUDA FOR COMPUTATIONAL PHYSICS

Delia SPRIDON[1]

Communicated to:
International Conference on Mathematics and Computer Science ,
September 15-17, 2022, Braşov, Romania, 4rd Edition - MACOS 2022

Dedicated to Professor Radu Păltănea on the occasion of his 70th anniversary

Abstract

Advances in the graphics processing unit (GPU) development led to the opportunity for software developers to increase the execution speed for their programs by massive parallelization of the algorithms using GPU programming. NVIDIA company developed an arhitecture for parallel computing named Compute Unified Device Architecture (CUDA) which includes a set of CUDA instructions and the hardware for parallel computing.

Computational Physics is an interdisciplinary field which is in continuous progress and which studies, develops and optimizes numerical algorithms and computational techniques for their application in solving various physics problems. Computational Physics has applicability in all sub-branches of physics and related fields such as: biophysics, astrophysics, plasma physics, biomechanics, fluid physics, etc. Moreover, with the evolution of technology in the last few decades, this relatively new field has helped to quickly obtain results in these fields, facilitating the connection between theoretical and experimental physics.

In this paper, some of the latest researches and results obtained in computational physics by using GPU computing with CUDA architecture are reviewed.

2000 *Mathematics Subject Classification:* 65Y05, 68Q10.

Keywords: parallel programming, computational physics

[1]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: delia.cuza@unitbv.ro

1 Introduction

At the beginnings of '90, the parallel programming started to be more and more necessary when the 3D graphics applications, especially gaming, began to be developed [14]. The widespread GPU programming was introduced at the beginnings of the 21'st century, was helped by the advances in hardware development and led to the necessity of research activity for the programmers so that known algorithms to be rethought in a parallel approach in order to reduce de processing time in this new industry. Moreover, during the last years, the necessity of simulating real life led to an incredible development of parallel programming .

The graphics processing units (GPUs) are electronic circuits consisting of up to thousands of microprocessors designed especially for parallel computations. Thus, besides their specific use for visualization and image processing, GPUs are more and more used to increase significantly the speed up of scientific computations [19]. Taking into account that GPUs were initially designed for image processing, one of the first other kind of application is matrix computing. Thus, special library for matrix computations were developed by using GPUs benefits. The Compute Unified Device Architecture (CUDA) is an arhitecture having a hardware part (NVIDIA GPU) and a software part (a set of instructions and libraries) and that can be used for general purpose computation on GPUs. This architecture has many libraries as integral components, such as cuBLAS, cuSOLVER, cuRAND, CUDA Math Library etc., designed for matrix computation and other basic algorithms for solving systems of ordinary differential equations, for machine learning and many other. Recent studies show the improvements that parallel programming can bring in various fields, taking advantage of the graphics cards of personal computers that are increasingly accessible and can be used to run various parallel algorithms. Thus, combining high performances and low-costs, GPU programming is currently used in fields such as molecular dynamics [14], medical imaging [15], financial simulation [24], geoscience simulations [20], fast 2D interpolations [4], graphs theory [6] etc.

2 About CUDA

The research for increasing execution speed and, implicitly, for reducing the complexity of CUDA kernels is continuously growing, also due to recently advances in GPU. However, the challenge of porting the applications to CUDA is still a technical and practical issue to be solved for programmers. In a CUDA implementation of an algorithm, the programmer needs to take care of the data transfer management between Central Processing Unit CPU and GPU, to ensure an optimal GPU memory usage and to pack GPU code in separate functions. The first step in creating a CUDA program is to design an algorithm that can be parallelized. This involves breaking down the task into smaller sub-tasks that can be executed independently. Moreover, a CUDA based program consists in two kind of codes: the set of instructions that runs on CPU named host code, and the set of intruction that runs on GPU named device code. A schema of

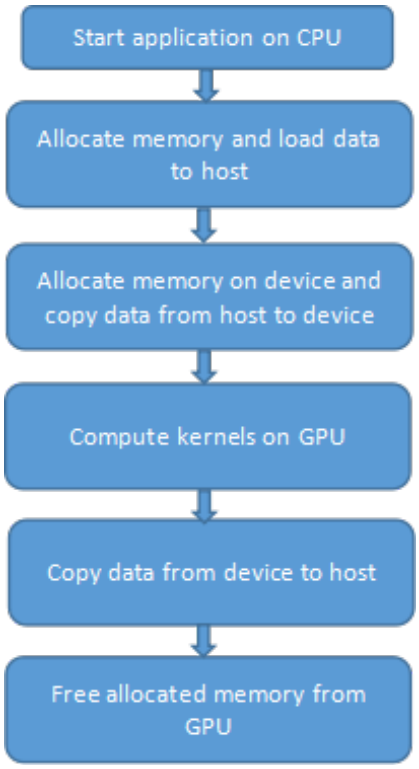

Figure 1: CUDA program work-flow

a CUDA work-flow is presented in Figure 1. The applications start running on CPU and the host code manages also the device code. The data that need to be processed are loaded on host memory, the necessary memory is allocated onto device and the data is loaded on the memory of the device using CUDA API calls such as "$cudaMalloc()$" and "$cudaMemcpy()$". The device kernels are called from host and runs on GPU taking advantage of GPU's ability of processing intensive tasks that can be executed on parallel.To launch a kernel, we need to specify the number of threads and the number of blocks to be used. This is done using the "<<<>>>" syntax in CUDA. Once the kernel has been launched, it will execute on the GPU. Each thread will execute the same code, but with different data. The data for each thread is accessed using the thread index, which is provided by CUDA. In order to ensure that all threads have completed their work before moving on to the next step, the threads need tobe synchronized using the "$\_syncthreads()$" function. Once the kernel has completed its work, we need to transfer the data back from the device to the host. Finally, the memory that was allocated on the device needs to be free using "$cudaFree()$".

Overall, the workflow of a CUDA program involves designing an algorithm that can be parallelized, allocating memory on the device, transferring data to and from the device, launching kernels, executing the kernel code, synchronizing threads, transferring data back to the host, and cleaning up allocated memory.

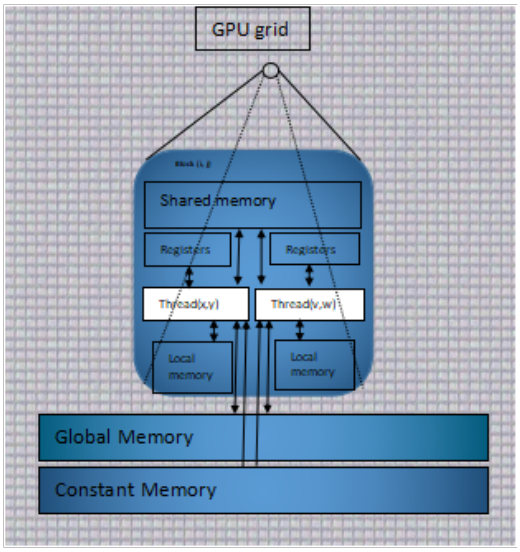

Figure 2: GPU memory hierarchy

The memory management plays an important role for best results with CUDA programming. Also, it is necessary to know the memory hierarchy of GPU so this could be use as efficient as possible. The GPU's memory levels (global memory, constant memory, shared memory, local memory and registers) are presented in Figure 2 .

- Global memory of the GPUs is the slowest memory to access, but there it is the largest. This can be managed by CPU using cudaMalloc, cudaFree, cudaMemcpy or cudaMemset functions, and therefore it is allocated / deallocated or set from CPU.
- Constant memory is part of the main memory of GPUs and all threads can read the same constant memory. The values on constant memory is set by CPU before launching kernels.
- Shared memory can be accesed very fast and it is used for fast communications between threads inside a block.
- Local memory is also slow being part of the main memory and it is automatically used by threads when the memory for registers is no longer available.
- Registers are the variables declared in kernels and they represent the fastest accessed memory. When the memory for registers run out, the local memory is used.

The results presented in few of the latest researches computational physics by using CUDA programing show significanly accelerated processes when using the GPU memory in a smart way. For example, in [22], the authors propose an accelerated solver for the 2D smoothed particle hydrodynamics by using in their

implementation a new caching method for CUDA shared memory. Thus, by using this method, the proposed software can be succesfully used for solving complex physicsl problems in computational fluid dynamics, or any other continuum medium simulators [16].

3 Latest results of CUDA programming for computational physics

Considering the wide variety of applications in the field of Computational Physics, the research studies in this domain are numerous. The acceleration of processes in computational physics is usually of great importance for obtaining the desired results as fast as possible. GPU programming is a suitable approach for obtaining very good execution time when a massive parallelization is possible, as the next studies will show.

CUDA for Fourier transform

One of the most important numerical intruments used for signal processing, image processing, or analysis of differential equations is the Fast Fourier Transform (FFT). The Fourier transform is the mathematical transform that decomposes certain functions into frequency components.

The formula for direct Fourier transform of an integrable function f :$\mathbb{R}$ -¿$\mathbb{C}$ is given by Equation 1. Equation 2 is the inverse Fourier transform.

$$F(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} \, dx \tag{1}$$

$$f(x) = \int_{-\infty}^{\infty} F(\xi) e^{2\pi i x \xi} \, d\xi \tag{2}$$

In mathematics, the Fourier transform is an operation that is applied to complex functions and produces another complex function that contains the same information as the original function, but reorganized by component frequencies. For example, if the original function ($f(x)$) is a time-dependent signal, its Fourier transform ($F(\xi)$) decomposes the signal by frequency (ξ) and produces a spectrum of it. The same effect is obtained if the initial function has as its argument the position in a uni- or multidimensional space (x), in which case the Fourier transform reveals the uni- or multidimensional spectrum of the spatial frequencies that make the input function. The computational complexity of discrete Fourier transform algorithm is of order $O(N^2)$ and for the fast Fourier transform, it is $O(N log(N))$ for a N-size signal. However, dealing in various applications with big data FFT calculations may lead to challenges regarding the execution time or energy consumption. For solving this issues, sparse fast Fourier transform was developed with a sub-linear computational complexity. Nevertheless, in various applications, depending on its size, execution time might still be a problem. Therefore, research

papers for accelerating the FFT by using GPU were published starting with 2003 [17, 10, 12]. A CUDA library was developed for FFT algorithm (cuFFT), recent study show the possibility of increasing the speed up for it. Thus, in [21], a massive parallel approach of sparse FFT is proposed, and the authors prove that their algorithm performs over 10x faster than the classical cuFFT library, and over 28x faster than the CPU parallelization of FFT. An other research group [2] also proposes GPU sparse FFT algorithm based on parallel optimization and the authors mention that this algorithm "leads enormous speedups". Moreover, in a very recent research [23], authors propose an hybrid MPI - CUDA implementation for nonequispaced discrete Fourier transformation using parallel threads launched from CPU nodes for managing the thread-level parallelism in multiple GPU devices. The authors prove that using hybrid parallelization, an increased improvement in computational efficiency is obtained without losing the computational precision. Also, their method can balance in a dynamic way the connection between performance and throughput capacity by modifying the number of computer nodes used for parallel computations [23].

To resume, the execution time of FFT (Fast Fourier Transform) algorithms can be greatly improved by leveraging the parallel computing capabilities of GPUs using CUDA. The execution time of FFT on GPUs can be influenced by several factors, including the size of the input data, the complexity of the FFT algorithm, and the number of GPU cores used. In general, the execution time of FFT on a GPU can be significantly faster than on a CPU. For example, for an input size of 4096 elements, the execution time of FFT on a CPU can be several milliseconds, while on a GPU it can be completed in less than 1 millisecond. The execution time of FFT on a GPU can also be affected by the type of FFT algorithm used. There are several FFT algorithms available, including Cooley-Tukey, Rader, and Bluestein. Each algorithm has its own strengths and weaknesses, and some may be better suited for parallel execution on GPUs than others. Overall, the execution time of FFT on a GPU using CUDA can be significantly faster than on a CPU, especially for larger input sizes. By leveraging the parallel computing capabilities of GPUs, developers can achieve significant speedups and improve the performance of their applications.

CUDA for solving Poisson's Equation

Poisson's equation is an elliptic partial differential equation, a generalization of Laplace's equation that is very important in theoretical physics. The general formula of Poisson is given by Equation 3, where $\varphi(r)$ is a scalar potential that needs to be determined and $f(r)$ is the so called source function.

$$\nabla\varphi(r) = f(r) \tag{3}$$

For example, the Poisson equation can be used to calculate the gravitational potential at distance r from a central point mass m, or for calculating the electrostatic potential for a known charge density distribution [11]. Moreover, Poisson equation plays an very important role in fluid dynamics [9] or in quantum mechan-

ical continuum solvation [3]. Thus, considering the importance of this eqution and to its wide variety of applications, many researches for solving Poisson equation by using numerical methods were published. For exemple, in 2015, Zhichen Feng and Zheng-Mao Sheng propose a new numerical method for solving the Poisson equation by using a physical model [8]. This proposed method has a computational complexity of $O(N)$ in certain conditions, and it can be used for any geometry or mesh style or in large scale parallel simulations. Other researches propose different approaches for solving the Poisson equation in different context [13, 1]. However, modern software requires fast solution for big computational problems. As mention before, GPU is a solution to obtain fast result for problems where parallel computation is possible. The steps that could be followed to solve the Poisson equation using CUDA are presented bellow:

- Define a 3D grid of points that cover the region of interest, where each point represents a node in the numerical solution.
- Allocate memory on the GPU for the grid points and the function values.
- Specify the boundary conditions for the problem, which will be used to set the values of the function on the boundary of the grid.
- Initialize the function values at the interior points of the grid to an initial guess.
- Use iterative numerical methods, such as the Jacobi or Gauss-Seidel method, to solve for the function values at the interior points of the grid.
- Once the solution has converged, transfer the function values back from the GPU to the host for further analysis.
- Free the memory that was allocated on the GPU.

Few researches that propose a fast solving Poisson equation method using GPU are presented in the literature. In 2011, Decyk and Singh proposed the use of cuFFT library for finding the solution of 2D Poisson equation for periodic boudary conditions [7].

In [5], a solver for 2D Poisson equation is proposed by using a parallelized version of Gauss-Seider algorithm. This algorithm is a generalization of Jacobi algorithm and the authors use CUDA to obtain a high execution speed. The speed up for the CUDA implemented algorithm depends on the size of the problem and the number of used threads. Thus, the speed up was up to 300 for 160000 nodes when 200 threads where used [5] (Figure 3).

4 Conclusions and perspectives

To conclude, eventhough many of the well known algorithms used in various applications have been already parallelized and many of them are already added

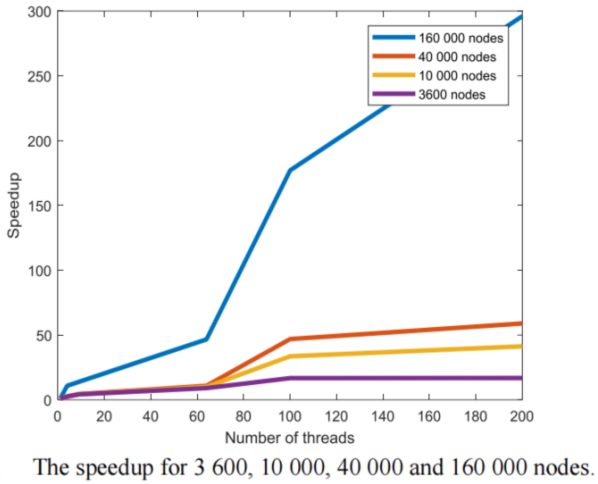

Figure 3: The speed-up for various number of nodes [5]

to CUDA library, new methods for optimization and for increasing the speedup can still be found. Execution time is crucial in many computational physics problems, and so, any improvement in this direction is still necessary. Hybrid parallel algorithms (CPU-GPU) are continuously developed in order to obtain high performance computing results with minimum costs. Therefore, cheap hardware solutions could be use for solving execution time issues in computational science by using strong parallelization that GPU provides.

References

[1] Adelmann, A., Arbenz, P. and Ineichen, Y., *A fast parallel Poisson solver on irregular domains applied to beam dynamics simulations*, J. Comput. Phys. **229**, (2010), 4554–4566.

[2] Artiles, O. and Saeed, F., *GPU-SFFT: A GPU based parallel algorithm for computing the Sparse Fast Fourier Transform (SFFT) of k-sparse signals*, 2019 IEEE International Conference on Big Data, 2019.

[3] Chen Z., and Wei, G.-W., *Differential geometry based solvation model. III. Quantum formulation*, The Journal of Chemical Physics **135** (2011) 194108.

[4] Ciupală, L., Deaconu, A., and, Spridon, D., *IDW map builder and statistics of air pollution in Braşov*, Bull. Transilv. Univ. Braşov Ser. III **63** (2021), 247-256.

[5] Clouthier-Lopez, J., Fernández, R.B., and de León, D.A.S., *A Parallel Implementation on CUDA for Solving 2D Poisson's Equation*, Research in Computing Science, **147** (2018), no. 12, 183–191.

[6] Deaconu, A.M., and Spridon, D., *Adaptation of Random binomial graphs for testing network flow problems algorithms*, Mathematics **9** (2021), Article number 1716.

[7] Decyk, V.K., and Singh, T.V., *Adaptable Particle-in-Cell algorithms for graphical processing units*, Comput. Phys. Commun. **182** (2011), 641–648.

[8] Feng, Z. and Sheng, Z.-M., *An approach to numerically solving the Poisson equation*, Physica Scripta, **90** (2015), no. 6.

[9] Fletcher, C.A.J., *Computational Techniques for Fluid Dynamics*, vol 1, 2nd edn Springer, (Berlin, 1991.

[10] Govindaraju, N.K. and Manocha, D., *Cache-efficient numerical algorithms using graphics hardware*, Parallel Comput. **33** (2007), 663-684.

[11] Griffiths, D.J., *Introduction to Electrodynamics*, Upper Saddle River, NJ: Prentice-Hall, 1999.

[12] Gu, L., Li, X. and Siegel, J., *An empirically tuned 2d and 3d FFT library on CUDA GPU*, In: 24th ACM International Conference on Supercomputing, New York, NY, USA, ACM, 2010, 305–314.

[13] Guillet, T., and Teyssier, R., *A simple multigrid scheme for solving the Poisson equation with arbitrary domain boundaries*, Journal of Computational Physics, **230**, (2011), 4756–4771.

[14] Harju, A., Siro, T. , Canova, F. F., Hakala, S., and Rantalaiho, T., *Computational Physics on Graphics Processing Units*, Lecture Notes in Computer Science **7782** (2013), 3-26.

[15] Kalaiselvi, T., Sriramakrishnan, P., and Somasundaram, K., *Survey of using GPU CUDA programming model in medical image analysis*, Informatics in Medicine Unlocked **9** (2017), 133-144.

[16] Lind, S. J., Rogers, B. D. and Stansby, P.K., *Review of smoothed particle hydrodynamics: towards converged Lagrangian flow modelling*, Proceedings of the Royal Society A **476** (2020), no. 2241.

[17] Moreland, K. and Angel, E., *The FFT on a GPU*, 2003 ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. HWWS '03, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association, 112–119, 2003.

[18] Papadakis, A.P., Ioannou, A. and Almady, W., *KYAMOS Software – CUDA aware MPI Solver for Poisson equation*, Journal of Multidisciplinary Engineering Science and Technology, **7** (2020), no. 12, 13208-13221.

[19] Volkov, V. , and Demmel, J., *Benchmarking GPUs to Tune Dense Linear Algebra*, 2008 ACM/IEEE Conference on Supercomputing, Austin, TX, USA.

[20] Walsh, S., Saar, M., Bailey, P., and Lilja, D., *Accelerating geoscience and engineering system simulations on graphics hardware*, Computers and Geosciences **35** (2009), 2353-2364.

[21] Wang, C., Chandrasekaran S. and Chapman, B., *cusFFT: A High-performance sparse Fast Fourier Transform algorithm on GPUs*, 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 963-972, 2016.

[22] Winkler, D., Rezavand, M. , Meister, M., and Rauch, W., *gpuSPHASE—A shared memory caching implementation for 2D SPH using CUDA*, Comput. Phys. Commun. **235** (2019), 514-516.

[23] Yang, S.C., and Wang, Y.L., *A hybrid MPI-CUDA approach for nonequispaced discrete Fourier transformation*, Comput. Phys. Commun. **258**, (2021), 107513.

[24] https://developer.nvidia.com/industries/financial-services, https://developer.nvidia.com/industries/financial-services.