IOPscience iopscience.iop.org

Home Search Collections Journals About Contact us My IOPscience

Use of GPUs to boost the performance of a lattice-free tumour growth model

This content has been downloaded from IOPscience. Please scroll down to see the full text.

2014 J. Phys.: Conf. Ser. 566 012019

(http://iopscience.iop.org/1742-6596/566/1/012019)

View the table of contents for this issue, or go to the journal homepage for more

Download details:

IP Address: 186.202.153.59
This content was downloaded on 29/06/2016 at 19:34

Please note that terms and conditions apply.

5th Young Researcher Meeting, Trieste 2014 IOP Publishing
Journal of Physics: Conference Series **566** (2014) 012019 doi:10.1088/1742-6596/566/1/012019

Use of GPUs to boost the performance of a lattice-free tumour growth model

Sabrina Stella[1], Roberto Chignola [2] and Edoardo Milotti[1,3]

[1]Department of Physics, University of Trieste, Italy
[2]Department of Biotechnology, University of Verona, Italy
[3]Istituto Nazionale di Fisica Nucleare, Sezione di Trieste, Italy

E-mail: `Sabrina.Stella@ts.infn.it`

Abstract. We recently developed a computational model of tumour growth. It is a cell-based model that can simulate the growth of multicellular tumour spheroids up to more than one million cells. The simulation program is very demanding and simulation time severely limits the integration of additional biological details, and indeed, at the moment, a typical simulation run requires tens of days to be completed. A new version of the code that exploits Graphics Processing Units (GPUs) to boost performance is being developed. In this paper we describe the design and implementation of a nearest-neighbour search (NNS) algorithm suitable to run on GPU. The algorithm will be integrated in the original code to manage the geometrical calculation in the simulation of the spheroid. Initially the stand alone NNS algorithm was tested for spheroids of different size: better efficency was obtained for bigger spheroids. Eventually the code was integrated in the whole simulation code and preliminary runs gave a speed up of about 5 for spheroids of relatively small size (15000 cells).

1. Introduction

Many computational problems in science and technology require the use of high performance computing solutions, since the huge amount of data and the complexity of the problem cannot be managed with single machines. Researchers then turn to highly parallel super-computer clusters, available in High Performance Computing (HPC) centres to solve their computational problems. A few years ago a new solution came from the video-game industry with the construction of small highly parallel processors, the Graphics Processing Unit (GPU). These are graphic accelerators designed to render scenes – that is to generate images – and relieve the computational load of a CPU, as a consequence more accurate images can be achieved in shorter time. The use of GPUs for scientific computing was not immediately possible, because these devices were designed to solve only those simple and repetitive tasks that are characteristic of the rendering process. Nevertheless, thanks to pioneering efforts and to the increasing interest of the scientific community, vendors were encouraged to propose new GPU architectures and programming models, to utilise these devices for more general problems. In 2007 the advent of the Nvidia CUDA architecture brought a considerable change in this scenario, with a new generation of graphic cards that could be easily programmed with extensions to standard C++ or FORTRAN code [1]. Nowadays, the community of CUDA programmers is steadily growing, just as the interest in the design of parallel algorithm suited to the GPU architecture [2, 3]. Not every algorithm can benefit from the computational power of the GPUs, because even those

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

algorithms that can be parallelised are not always portable on the GPU architecture. GPUs were originally designed for the so-called data-intense parallel problem, such as the rendering processes, where the same flux of instructions is executed on many cores on a huge amount of data. For this reason, even today, graphic cards are more suitable to run those algorithms that can be *data-parallelised* or that can be subdivided in small sub-problem based on group of data – and this happens to be the case of most computational problems in physical simulations.
The design of a proper parallel algorithm for GPU requires a basic knowledge of the device's architecture. The main difference between GPUs and CPUs is that the processing units in a GPU are simpler and smaller when compared to a traditional CPU, and this allows the integration of thousands of computing units in a GPU card, hence its computational power. In a recent GPU architecture called Kepler, the CUDA cores are grouped in multiprocessors (SMX) which have 192 cores (Tesla K20) and a single card can have more than one SMX (there are 13 SMXs in the Tesla K20). The GPU card has its own private memory with a high bandwidth. The memory is organised in a hierarchical structure: typically there is a high capacity and high latency access memory, called *global memory*, it is accessible by all the cores. Other different levels of memory are also present, they are characterised by lower latency time but also by lower capacity compared to the global memory and some of those are gradually more private to a single core (for details see [1]). The memory management task is partially left to the programmer who can obtain a better performance with a proper memory use.
A GPU program is organised in *threads*, each thread executes the same set of instructions – called a *kernel* – although on different data. Threads are grouped in *blocks*, and blocks are organized in a *grid* that can be 1-, 2- or 3-dimensional. The choice of the number of threads and the dimension of the grid and of the blocks – up to a hardware-dependent maximum – is left to the programmer. All this contributes to make the design and implementation of parallel algorithms for GPUs both interesting and useful, although not trivial. At present, parallel algorithms running on GPU are becoming quite widespread, both in research and in industry. In this paper we explain how we use the GPU to speed up the biophysical simulation of a numerical model of tumour growth.

2. Biophysical model of tumour growth

We have developed a biophysical model that simulates the growth of small avascular solid tumours. Originally, it was conceived to describe the evolution of a multicellular tumour spheroids (MCTS). A MCTS is an *in vitro* cell culture where cells are grown in such a way to prevent their adhesion to the walls of the culture well, and they cluster in a roughly spherical shape. Our model starts with the mathematical description of the individual cell cycle. Each cell is approximated as a soft sphere that receives nutrients from the environment, it can grow and proliferate and it interacts, both biochemically and mechanically, with the surrounding cells. The model includes a set of reaction-diffusion equations that describe the dynamics of a selected set of molecules. Threshold mechanisms are used to mark the transitions between the phases of the cell cycle, and a phenomenological model describes the cell-cell forces, which exhibit a repulsive or attractive behaviour according to the distance between the cells' centers. For a more complete description of the model see [4]. In a standard run we start with a single cell in a uniform environment, and as the run proceeds, cells grow and proliferate and eventually the cell aggregate can include as many as 1.5 million cells. The results of simulation runs show that an *in-silico* MCTS has properties comparable with those of real MCTS, such as its external shape and the distribution of dead cell in the aggregate (presence of a necrotic core) [5].

The program that implements the model is coded in C++, the disordered structure is managed with the CGAL library [6], its computational complexity depends roughly linearly on the number of cells. However, the number of cells grows fast, the initial growth is exponential in time, and it turns out that the CPU time for each simulation step steadily increases as

the simulated spheroid grows, and that a full simulation run takes tens of days. During the simulation, the spheroids do not evolve following a predetermined lattice structure but the cells are free to occupy every position on the 3D space. As a result, the computational complexity of each step depends in good part on the geometrical calculation performed with CGAL. As a consequence, the simulation is more realistic but requires the continuous evaluation of some geometrical parameters which are needed to solve the model's equations. In particular the list of neighbouring cells - of each cell in the spheroid - needs to be evaluated whenever the position of the cell's center changes beyond a given threshold; the movements are due to cell-cell forces and to cell division, when the spheroid is large these movements are so frequent that the neighborhood changes at each time step.
To improve performance we modify the code by moving the geometrical calculation onto a GPU. This was done by developing and implementing a proper parallel algorithm for searching the nearest neighbour cells, starting with a set of cells that changes dynamically.

3. Design and Implementation of a GPU-based adjacent-cell search algorithm

The nearest-neighbour search is a well known problem in the field of computational geometry. It can be solved using two different approaches: the brute force approach – i.e., by comparing the distance between each single cell and all the other cells, and finally selecting those with the smallest distances – or by performing a *space partitioning operation* to facilitate the search – i.e., by limiting the search operation on a given subspace.
Parallel formulations well-suited to GPUs of both algorithms have already been developed [7, 8, 9] and most of the experience has been garnered in the field of image processing.
In our case we started with the algorithm described in [9] , which considers a space partitioning operation using a cubic grid and we modified it in order to find the "adjacent cells", i.e., those cells that are in close contact with a query cell, and therefore contribute to the diffusion processes. The contact between two cells exists only if the distance between their center is smaller than the sum of their radii multiplied by a suitable extension factor; this last parameter is introduced to take into account the softness of cells. The extended radius – i.e., the radius multiplied by the extension factor – is also used to evaluate the contact surface.

3.1. Adjacent cell search algorithm scheme

The search of the adjacent cells in the whole spheroid can be easily parallelised on GPUs when threads are tasked to find the neighbourhood of individual cells. These operations are indeed independent of each other, therefore they can be executed in parallel. As already mentioned, the search operation is optimised by the space partitioning operation: first the bounding box of the spheroid is identified and then it is subdivided in equally sized small boxes (see fig.1). Next, each cell can be given a hash value, that is a number identifying the small box where it resides. This hash value can be used by the thread to identifyin subspace where performing the search operation, in particular each thread verifies the adjacency condition for the subset of cells included in the boxes having the same or adjacent hash values as the examined cell, that is it scans 27 small boxes (or less in case of cells belonging to the boundary boxes). A more complete description of the algorithm is given in [10].

3.2. Test run

The algorithm stand alone was implemented using the extention CUDA 5.0 for C++ and the Thrust library [11] - the parallel version of the Standard Template Library. The program was run on a GPU Nvidia Quadro 4000, provided with 256 cuda core of about 0.95 GHz, (32 cuda core/SM). The execution is performed opening a number of threads, equal or higher than the number of cells composing the spheroid, i.e. a thread for each biological cell. The threads were grouped in block of 64 (a block multiple of 32 thread can assure better performance) and

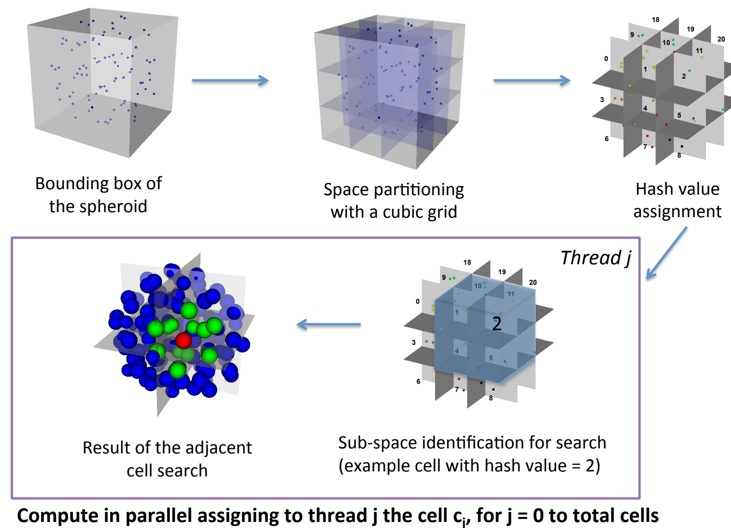

Figure 1. Main steps of the nearest-neighbour search (NNS) algorithm.

the blocks were arranged on a squared 2D grid in order to manage approximately 10^6 cells or threads; the grid's dimension changes according to the size of the problem, as an example for a spheroid of 40000 cells we used a 25x25 grid of threads blocks. The algorithm was tested for spheroids with different size, from hundreds to one million of cells. As expected the efficency of the algorithm increases with the size of spheroid. The execution time (in ms, calculated by the CUDA runtime API `cudaEventElapsedTime()[12]`) for the kernel responsible for the adjacent cell search (the last two steps in fig.1) is shown in fig.2. The search operation requires only 403 ms to determine the list of neighbors for a spheroid higher than one million of cells.

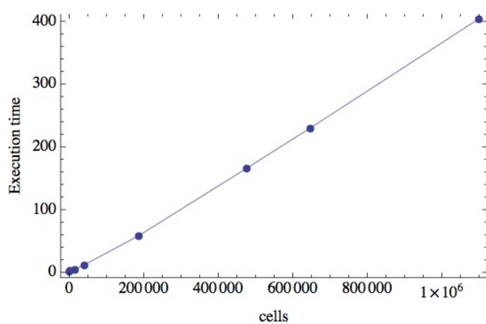

Figure 2. Execution time (ms) of the kernel responsible for the adjacent cell search as a function of the total number of cells (it does not include the space partitioning operation and the hash value assignment operation)

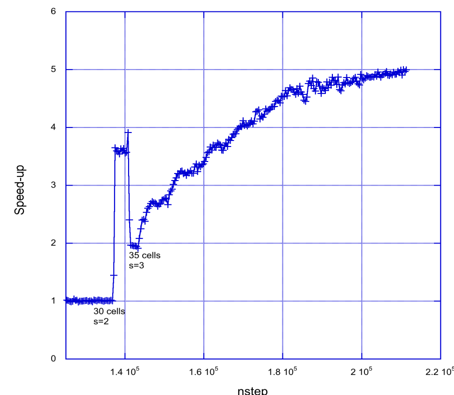

Figure 3. Speed up as a function of spheroid size for the geometrical calculation in the main program

3.3. Accelerating the tumour growth runs on a GPU using the NNS algorithm

A standalone version of the algorithm for the adjacent cell search was integrated in the simulation program. The main program is essentially a time loop which starts after initialisation and it

performs in sequence the following operations: 1. calculation of biochemistry and biomechanics of the cluster; 2 check for discrete cellular events; and, finally, 3. recalculation of the geometrical parameters, including the proximity relationships, if a mitosis event occurs, or if the cluster changed shape beyond a certain threshold. The new CPU-GPU version of the code was implemented by partially assigning to GPU the third step of the main loop. This is done by transferring on GPU all the geometrical data for the cells aggregate (positions of cell centres and radii), and running the nearest-neighbour search (NNS) algorithm to find the list of the adjacent cells. Moreover, a new condition was introduced to determine also the list of boundary cells [1]. These cells represent the interface with the environment and their correct evaluation is important for a correct growth of the spheroids. After the calculation, the GPU transfers to the CPU two vectors: `AdCells[N*30]` and `BC[N]` – where `N` is the total number of cells and 30 is the maximum number of neighbours for each cells – they store respectively the index of the adjacent cells and the list of boundary cells. These device-host (GPU-CPU) and host-device transfer operations are usually the most expensive ones, and have to be executed sparingly to get the best performance.

To evaluate the performance of this new version of the code we introduce a speed-up value, that is the ratio of the execution time on CPU and on CPU+GPU of the functions responsible for the geometry calculation. A preliminary result is given in fig.3: the speed-up increases with the simulated time step or similarly with the size of the spheroids, and this is due to the higher efficiency of the nearest neighbour search algorithm for larger spheroids. In these preliminary runs simulations stop at a relatively small size (about 15000 cells), where the maximum speed-up obtained is about 5. We are now developing a new version of the algorithm for the boundary cells identification: this is required to perform a full simulation.

4. Conclusions

The preliminary runs of the new code show an increasing speed-up value as a function of the spheroid size. A better performance is expected when more recent GPUs, such as TeslaK20 will be used. The forthcoming introduction of some common optimisation procedures – such as a better use of the memory hierarchy – is also expected to increase the speed-up.We also expect a large gain by moving on GPU the other critical step, i.e., the reaction-diffusion part of the model. A faster code is fundamental for the introduction of new biological detail in our model, with the final goal to get a more realistic biological environment that may be used to reveal some hidden properties of the tumour growth process.

References

[1] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`
[2] Couturier R 2014 *Designing Scientific Applications on GPUs* (CRC Press)
[3] Navarro C A, Hitschfeld N and Mateu L 2014 *Communications in Computational Physics* **15**
[4] Milotti E, Del Fabbro A and Chignola R 2009 *Computer Physics Communications* **180** 2166–2174
[5] Milotti E and Chignola R 2010*PLoS One* **5**
[6] `http://www.cgal.org`
[7] Garcia V, Debreuve E and Barlaud M 2008 *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on* (IEEE) pp 1–6
[8] Pan J and Manocha D 2011 *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (ACM) pp 211–220
[9] Pedro L, Teixeira J M, Farias T, Reis B, Teichrieb V and Kelner J 2012 *International Journal of Parallel Programming* **40** 313–330
[10] Stella S, Chignola R, Dogo F and Milotti E 2014 *High Performance Computing and Simulation (HPCS), 2014 International Conference on* 621 – 626
[11] https://code.google.com/p/thrust/
[12] http://docs.nvidia.com/cuda/cuda-runtime-api

[1] The condition for a cell to be considered on the boudary is to lie on the outermost small box layer