Use of Multiple GPUs to Speedup the Execution of a Three-Dimensional Computational Model of the Innate Immune System

To cite this article: M P Xavier et al 2014 J. Phys.: Conf. Ser. 490 012075

View the article online for updates and enhancements.

Related content
- Triggering events with GPUs at ATLAS
  S Kama, J Augusto Soares, J Baines et al.
- GPUs for statistical data analysis in HEP: a performance study of GooFit on GPUs vs. RooFit on CPUs
  Alexis Pompili, Adriano Di Florio and CMS Collaboration.
- Multi-GPU Jacobian accelerated computing for soft-field tomography
  A Borsic, E A Attardo and R J Halter
Use of Multiple GPUs to Speedup the Execution of a Three-Dimensional Computational Model of the Innate Immune System

M P Xavier¹, T M do Nascimento², R W dos Santos³, M Lobosco⁴
¹,²,³,⁴FISIOCOMP Lab, Federal University of Juiz de Fora, Brazil
E-mail: ¹micael.peters@ice.ufjf.br, ²tiago.nascimento@uab.ufjf.br,
³rwdsantos@yahoo.com, ⁴marcelo.lobosco@ice.ufjf.br

Abstract. The development of computational systems that mimics the physiological response of organs or even the entire body is a complex task. One of the issues that makes this task extremely complex is the huge computational resources needed to execute the simulations. For this reason, the use of parallel computing is mandatory. In this work, we focus on the simulation of temporal and spatial behaviour of some human innate immune system cells and molecules in a small three-dimensional section of a tissue. To perform this simulation, we use multiple Graphics Processing Units (GPUs) in a shared-memory environment. Despite of high initialization and communication costs imposed by the use of GPUs, the techniques used to implement the HIS simulator have shown to be very effective to achieve this purpose.

1. Introduction
Nowadays, the scientific community makes large use of computational tools to better understand the behaviour of distinct complex systems, such as those used to mimic the functioning of the immune system. The development of such systems are essential, for example, to better understand the role of each cell and molecule in an immune response. However, such systems can demand a large amount of computational resources, in particular computation time. The use of parallel algorithms is mandatory in such cases.

This work describes the use of multiple Graphics Processing Units (GPUs), in a shared-memory environment, to reduce the computation time required to simulate the temporal and spatial dynamics of some human immune system (HIS) cells and molecules in a small three-dimensional section of tissue when this tissue is exposed to pathogens. The sequential version of the code can run for days, depending on the parameters used in simulations, such as the size of the tissue. A first attempt to reduce this huge computation time employed a single GPU [1] to perform the parallel processing. Although the parallel version achieved a respectable speedup of about 72 times over its sequential counterpart[1], a better speedup is required to deal with large tissues or more detailed simulations.

This work is organized as follows. Section 2 describes the versions of code developed in order to execute the innate HIS simulations on a multi-GPU platform. Computational results obtained experimentally are presented in Section 3. Finally, our conclusions and plans for future works are presented in the last section.
2. Multiple GPUs
The first version of the code used to simulate the temporal and spatial behaviour of some cells and molecules of the innate HIS in a small three-dimensional section of tissue [1] uses CUDA in a single GPU to reduce its execution time. This single-GPU version of the code is extended in this work to use multiple GPUs that can be available in a single machine in order to speedup even more the execution time. To manage these multiple GPUs, PThreads API [2] is used, so each host (or CPU) thread is responsible for invoking a CUDA kernel in a particular CUDA device. The use of multiple host threads is necessary to reduce the imbalance caused when a single thread launches all kernels in distinct GPUs. If a large number of GPUs are available, when the single host thread finishes to launch the last kernel in the last GPU, probably the first kernel launched in the first GPU has advanced a lot in its work, or even finished it. Since kernels are executed independently by each GPU, it is necessary to synchronize host threads at each time-step in order to compute populations correctly. This is done using the `cudaStreamSynchronize` directive at the end of each time iteration, so the main host thread synchronize its execution with each of the CUDA kernels launched before starting a new time-step.

To solve the system of Partial Differential Equations (PDEs) that describes the behaviour of the innate HIS cell and molecules [1], the finite difference method was employed for the spatial discretization and the explicit Euler method for the time evolution. The discretization of the chemotaxis term uses the First-Order Upwind scheme[3] that discretize the hyperbolic PDEs through the use of differences with bias in the direction given by the signal of the characteristics speeds. The upwind scheme uses an adaptive or solution-sensitive stencil to numerically simulate more precisely the direction of information propagation. With multiple GPUs, the discretized space is divided among devices, so each one will operate on a specific slice of the original space such that the whole tissue is processed by the group of GPUs. Splitting was done by dividing the x dimension of a \((N_x, N_y, N_z)\) mesh that describes the tissue by the number of GPUs, \(N_g\), remaining a \(((\frac{N_x+N_g-1}{N_g}), N_y, N_z)\) mesh to be calculated by each device. The mesh is organized contiguously in the device memory as an unidimensional vector and the access to the points in the tissue was done linearly.

In CUDA, the execution configuration hugely impacts the performance of the application. In this work we choose a fixed block size, 128 threads, based on the memory demands of each thread. Then, a function was created to automatically generate the grid size. This function calculates the grid size taking into account that a thread computes a single point, and that the grid is unidimensional. Also, a buffer was implemented to avoid race condition among CUDA threads. Its role is quite simple: two values at times \(t-1\) and \(t\) are stored for each point \((x, y, z)\) of a given population of cells. The value at time \(t\) is accessed only by the thread that is producing it, while the other one, \(t-1\), is accessed by threads in neighbourhood that needs to read it. Thus, a thread at time \(t\) only gets access to data produced by its neighbours at time \(t-1\). These two buffer entries change their meaning at each time step to avoid data copy.

To correctly compute each point, each CUDA thread has to access its neighbours data, some of which located at distinct GPUs due to data splitting among GPUs. These data, called boundaries, are necessary simultaneously to execute the computation in two distinct GPUs. In order to deal with this issue, three distinct approaches were implemented.

In the first version, boundary data is copied between two GPUs using for this purpose the host memory space. The data copy process is divided into two steps: in the first one, boundary data is copied from the source GPU to the host memory. The final step copies data from the host memory to the destination GPU.

Although the first version of the code is quite simple, the data copy between GPU and CPU, and vice-versa, may degrade performance. To reduce the cost of memory copy operations, a second version uses the concept of unified address space among GPUs. This concept is implemented using the UVA (Unified Virtual Addressing) functions [4], so one GPU can access
directly the memory positions located on a distinct one. To allow the access of a distinct GPU
to its local memory space, it’s necessary to call first the function cudaEnablePeerAccess. The
access granted by a call to this function is unidirectional, so a separate symmetric call to this
function is required. However, it is not possible to grant access simultaneously to two or more
GPUs to a single memory space. For example, if one GPU has to access the memory space of
two adjacent GPUs in a single iteration, it has to enable a peer access with both GPUs, one at
a time and at each time-step, before accessing their memory positions, which can impact the
performance negatively.

To solve the issues mentioned with the second version, a third version was implemented.
This new version calls cudaEnablePeerAccess only once, during the initialization of the code,
and then data can be copied among GPUs. The main difference between this version and the
previous one is that in this version data must be copied explicitly, using cudaMemcpy, while
in the previous version data is accessed directly, since the memory space is unified. Also, when
compared to the first version, this version eliminates the use of the host memory space as a
temporary buffer.

3. Experimental Results
This section presents the speedups obtained by the three distinct versions of our multi-GPU
code. They were obtained in a four AMD 6272 Linux system, with 128GB of main memory.
This machine has four Tesla M2050 GPUs, each with 512 CUDA cores and 5.2 GB of global
memory. Linux 3.9.2, CUDA driver version 5.0, nvcc release 5.0 and gcc version 4.7.2 were used
to run and compile all versions of the code.

In order to evaluate the performance gains obtained by the multi-GPU version, experiments
using four distinct mesh sizes: 50×50×50, 100×100×100, 150×150×150 and 200×200×200
points. The mesh discretization generates about 1,000 points for each 1 mm³ of tissue. One
day of infection takes about one million of interactions to be computed. Since the execution
time of each interaction is extremely regular, and the objective of this paper is to evaluate the
techniques, nor the biological results, in this work we report the result for 10,000 time steps.
The execution times obtained by all versions of the code were measured 5 times and the standard
deviation was lower than 2.2%. At each execution, Linux time application was used to measure
the time spent in program execution.

The relative speedups to the single-GPU version of the code are presented in Table 1, using
the Equation 1 to compute the speedups:

\[ S_g = \frac{t_1}{t_g} \]  

where \( t_1 \) is the single-GPU execution time and \( t_g \) is multi-GPU execution time with 4 GPUs.
Values greater than 1.0 represent a speedup, while values less than 1.0 represent a slowdown.

Table 1 presents the obtained results. First, the second version of the code always slowdowns,
regardless the size of the mesh. Our profile shows that the high cost imposed to call the function
cudaEnablePeerAccess is responsible for this slowdown. The third version of code obtained the
best results in all scenarios. Finally, as the mesh size increases, the relative speedup of the third
version of code becomes better. For the largest mesh size, the speedup obtained is about 82.5% of
the maximum speedup that could be obtained, a linear speedup of 4 using four GPUs.

4. Conclusion
This work presented an implementation of an innate HIS simulator on a multi-GPU platform.
Three distinct versions of code were developed with this purpose. Surprisingly, experimental
results have shown that the best results were obtained by the version that explicitly copy data
between GPUs. This occurs due to the restrictions imposed by CUDA shared memory primitives.
Table 1. Speedups obtained by the multi-GPU versions over the single-GPU one. Average execution times for 10,000 time steps are expressed in seconds.

<table>
<thead>
<tr>
<th>Mesh</th>
<th>Version</th>
<th>Runtime average (s)</th>
<th>Standard deviation</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 × 50 × 50</td>
<td>single-GPU</td>
<td>30.08</td>
<td>0.5 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.1</td>
<td>29.35</td>
<td>1.2 %</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.2</td>
<td>79.51</td>
<td>0.3 %</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.3</td>
<td>22.60</td>
<td>0.5 %</td>
<td>1.3</td>
</tr>
<tr>
<td>100 × 100 × 100</td>
<td>single-GPU</td>
<td>158.36</td>
<td>0.1 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.1</td>
<td>93.35</td>
<td>2.2 %</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.2</td>
<td>331.11</td>
<td>0.2 %</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.3</td>
<td>66.01</td>
<td>0.5 %</td>
<td>2.4</td>
</tr>
<tr>
<td>150 × 150 × 150</td>
<td>single-GPU</td>
<td>512.45</td>
<td>0.0 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.1</td>
<td>230.34</td>
<td>0.5 %</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.2</td>
<td>843.12</td>
<td>0.2 %</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.3</td>
<td>183.07</td>
<td>0.8 %</td>
<td>2.8</td>
</tr>
<tr>
<td>200 × 200 × 200</td>
<td>single-GPU</td>
<td>1224.86</td>
<td>0.0 %</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.1</td>
<td>440.20</td>
<td>0.3 %</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.2</td>
<td>1698.64</td>
<td>0.3 %</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>multi-GPU v.3</td>
<td>373.62</td>
<td>0.7 %</td>
<td>3.3</td>
</tr>
</tbody>
</table>

The best multi-GPU version was up to 3.3× faster than the parallel code that uses one GPU and up to 72×3.3 = 237.6 times faster than the sequential code. As future work, we propose two new extensions. The first one will try to overlap the boundary transfers with computation. To achieve this goal, the computation of both the chemotaxis and the Laplacian operator for each point could be divided in two steps, the computation of internal points and boundary points, using for this purpose two CUDA kernels. Since the amount of boundaries points is smaller than internal points, its computation theoretically will finish first, so data copy can start early, while internal points computation is taking place. The computation and communication executions can be executed concurrently with the use of CUDA streams. The second proposed version will extend our work to deal with the situation where GPUs are spread across a cluster of computers. For this purpose, the MPI API will be used to establish communication between distinct nodes.

Acknowledgements
The financial support by CNPq, CAPES, UFJF, FINEP and FAPEMIG is greatly acknowledged.

References