Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 566–575

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

cuHinesBatch: Solving Multiple Hines systems on GPUs Human Brain Project *

Pedro Valero-Lara[1], Ivan Martínez-Pérez[1], Antonio J. Peña[1], Xavier Martorell[1,2], Raül Sirvent[1], and Jesús Labarta[1,2]

[1] Barcelona Supercomputing Center (BSC) Barcelona, Spain.
{pedro.valero, ivan.martinez, antonio.pena, raul.sirvent, jesus.labarta}@bsc.es
[2] Universitat Politècnica de Catalunya
xavim@ac.upc.edu

Abstract

The simulation of the behavior of the Human Brain is one of the most important challenges today in computing. The main problem consists of finding efficient ways to manipulate and compute the huge volume of data that this kind of simulations need, using the current technology. In this sense, this work is focused on one of the main steps of such simulation, which consists of computing the Voltage on neurons' morphology. This is carried out using the Hines Algorithm. Although this algorithm is the optimum method in terms of number of operations, it is in need of non-trivial modifications to be efficiently parallelized on NVIDIA GPUs. We proposed several optimizations to accelerate this algorithm on GPU-based architectures, exploring the limitations of both, method and architecture, to be able to solve efficiently a high number of Hines systems (neurons). Each of the optimizations are deeply analyzed and described. To evaluate the impact of the optimizations on real inputs, we have used 6 different morphologies in terms of size and branches. Our studies have proven that the optimizations proposed in the present work can achieve a high performance on those computations with a high number of neurons, being our GPU implementations about 4× and 8× faster than the OpenMP multicore implementation (16 cores), using one and two K80 NVIDIA GPUs respectively. Also, it is important to highlight that these optimizations can continue scaling even when dealing with number of neurons.

Keywords: Human Brain, Neuron, Hines Algorithm, Parallel Computing, GPUs, Multicore, CUDA.

*This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 720270 (HBP SGA1), from the Spanish Ministry of Economy and Competitiveness under the project Computación de Altas Prestaciones VII (TIN2015-65316-P) and the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Paral·lels (2014-SGR-1051). We thank the support of NVIDIA through the BSC/UPC NVIDIA GPU Center of Excellence. Antonio J. Peña is cofinanced by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva fellowship number IJCI-2015-23266.

1 Motivation

Today, we can find multiple initiatives that attempt to simulate the behavior of the Human Brain by computer simulations [9, 5, 7]. This is one of the most important challenges in the recent history of computing with a large number of practical applications. The main constraint is being able to simulate efficiently a huge number of neurons using the current computer technology. One of the most efficient ways in which the scientific community attempts to simulate the behavior of the Human Brain consists of computing the next 3 major steps [6]: The computing of 1) the Voltage on neuron morphology, 2) the synaptic elements in each of the neurons and 3) the connectivity between the neurons. In this work, we focus on the first step which is one of the most consuming time steps of the simulation. Also it is strongly linked with the rest of steps. All these steps must be carried out on each of the neurons. The Human Brain is composed by about 14 thousand million of neurons, which are completely different among them in size and shape.

The standard algorithm used to compute the Voltage on neurons' morphology is the Hines algorithm [8]. This algorithm is based on the Thomas algorithm [2], which solves tridiagonal systems. Although the use of GPUs to compute the Thomas algorithm has been deeply studied [16, 15, 4, 17, 3], the differences among these two algorithms, Hines and Thomas, makes us impossible to use the last one as this can not deal with the sparsity of the Hines matrix.

Previous works [1] have explored the use of other algorithms based on the Stone's method [10]. Unlike Thomas algorithm, this method is parallel. However, it is in need of a higher number of operations ($20n \log 2n$) with respect to the ($8n$) operations of the Thomas algorithm to solve one single system of size n. Also, the use of parallel methods present some additional drawbacks to be dealt with. For instance, it would be difficult to compute those neurons that compromise a size bigger than the maximum number of threads per CUDA block (1024) or shared memory (48KB). Other problems are the computationally expensive operations such as atomic accesses and synchronizations necessary to compute this method. Each neuron presents a particular morphology and so a different scheduling (preprocessing) must be applied to each of them which makes even more difficult its implementation.

Unlike the work presented in [1], where a relatively low number of neurons (128) is computed using single precision operations, in this work we are able to execute a very high number of neurons (up to hundreds of thousands) using double precision operations. We have used the Hines algorithm, which is the optimum method in terms of number of operations, avoiding high expensive computational operations, such as synchronizations and atomic accesses. Our code is able to compute a high number of systems (neurons) of any size in one call (CUDA kernel), using one thread per Hines system instead of one CUDA block per system. Although multiple works have explore the use of GPUs to compute multiple independent problems in parallel without transforming the data layout [13, 14, 11, 12], the particular characteristics of the sparsity of the Hines matrices forces us to modify the data layout to efficiently exploit the memory hierarchy of the GPUs (coalescing accesses to GPU memory). These modifications have not been explored previously, which are deeply described and analyzed in the present work.

This paper is structured as follows. Section 2 briefly introduces the physical problem at hand and the general numerical framework that has been selected to cope with it: Hines algorithm. In Section 3 we present the specific parallel features for the resolution of multiple Hines systems, as well as the parallel strategies envisaged to optimally enhance the performance. Section 4 shows the performance achieved by the proposed techniques. Also, this section studies the influence of different morphologies on performance. Finally, the conclusions are outlined in Section 5.

2 Hines Algorithm

In this section, we describe the numerical framework behind the computation of the Voltage on neurons morphology. It follows the next general form:

$$C\frac{\partial V}{\partial t} + I = f\frac{\partial}{\partial x}(g\frac{\partial V}{\partial x}) \tag{1}$$

where f and g are functions on x-dimension and the current I and capacitance C [6] depend on the voltage V. Discretizing the previous equation on a given morphology we obtain a system that has to be solved every time-step. This system must be solved at each point:

$$a_i V_{i+1}^{n+1} + d_i V_i^{n+1} + b_i V_{i-1}^{n+1} = r_i \tag{2}$$

where the coefficients of the matrix are defined as follow:

$$\begin{gathered}
\text{upper diagonal: } a_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2} \\
\text{lower diagonal: } b_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2} \\
\text{diagonal: } d_i = \frac{C_i}{\Delta_t} - (a_i + b_i) \\
\text{rhs: } r_i = \frac{C_i}{\Delta_t} V_i^n - I - a_i(V_{i-1}^n - V_i^n) - b_i(V_{i+1}^n - V_i^n)
\end{gathered}$$

The a_i and b_i are constant in the time, and they are computed once at start up. Otherwise, the diagonal (d) and right-side-hand (rhs) coefficients are updated every time-step when solving the system.

The discretization above explained is extended to include *branching*, where the spatial domain (neuron morphology) is composed of a series of one-dimension *sections* that are joined at branch points according to the neuron morphology.

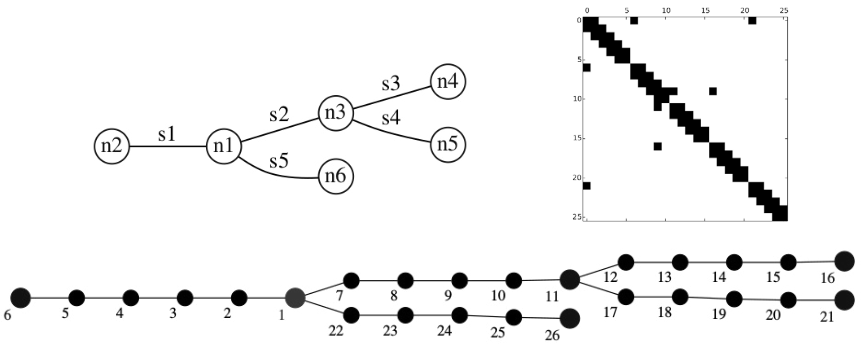

Figure 1: Example of a neuron morphology and its numbering (left-top and bottom) and sparsity pattern corresponding to the numbering followed (top-right).

For sake of clarity, we illustrate a simple example of a neuron morphology in Figure 1. It is important to note that the graph formed by the neuron morphology is an acyclic graph, i.e. it has no loops. The nodes are numbered using a scheme that gives the matrix sparsity structure that allows to solve the system in linear time.

To describe the sparsity of the matrix from the numbering used, we need an array (p_i $i \in [2 : n]$) which stores the parent indexes of each node. The pattern of the matrix which illustrates the morphology shown above is graphically illustrated in Figure 1.

The Hines matrices feature the following properties: they are symmetric, the diagonal coefficients are all nonzero and per each off-diagonal element, there is one off-diagonal element in the corresponding row and column (see row/column 7, 12, 17 and 22 in Figure 1).

Given the aforementioned properties, the Hines systems ($Ax = b$) can be efficiently solved by using an algorithm similar to Thomas algorithm for solving tri-diagonal systems. This algorithm, called Hines algorithm, is almost identical to the Thomas algorithm except by the sparsity pattern given by the morphology of the neurons whose pattern is stored by the p vector. An example of the sequential code used to implement the Hines algorithm is illustrated in pseudo-code in Algorithm 1.

Algorithm 1 Hines algorithm.

```
1:  void solveHines(double *u, double *l,   double *d,
2:                  double *rhs,     int *p, int cellSize)
3:  // u → upper vector, l → lower vector
4:  int i;
5:  double factor;
6:  // Backward Sweep
7:  for $i = cellSize - 1 \to 0$ do
8:      factor    = u[i] / d[i];
9:      d[p[i]]   -= factor × l[i];
10:     rhs[p[i]] -= factor × rhs[i];
11: end for
12: rhs[0] /= d[0];
13: // Forward Sweep
14: for $i = 1 \to cellSize - 1$ do
15:     rhs[i] -= l[i] × rhs[p[i]];
16:     rhs[i] /= d[i];
17: end for
```

3 Implementation of Multiple Hines Solve on GPUs

An efficient memory management is critical to achieve a good performance, but even much more on those architectures based on a high throughput and a high memory latency, such as the GPUs. In this sense, we focus on presenting the different data layouts proposed and analyze the impact of these on the overall performance. Three different data layouts were explored: *Flat*, *Full-Interleaved* and *Block-Interleaved*. While the *Flat* data layout consists of storing all the elements of each of the systems in contiguous memory locations, in the *Full-Interleaved* data layout, first, we store the first elements of each of the systems in contiguous memory locations, after that we store the set of the second elements, and so on until the last element. Similarly to the *Full-Interleaved* data layout, the *Block-Interleaved* data layout divides the set of systems in groups of systems of a given size (BS), whose elements are stored in memory by using the strategy followed by the *Full-Interleaved* approach.

For sake of clarity, Figure 2 illustrates a simple example composed by four different Hines systems of three elements each. Please, note that we only illustrate one vector per system in Figure 2, but in the real scenario we would have 4 vectors per Hines system (Pseudocode 1) on which are carried out the strategies above described. As widely known, one of the most important requirements to achieve a good performance on NVIDIA GPUs is to have contiguous threads accessing contiguous memory locations (coalescing memory accesses). This is the main motivation behind the proposal of the different data layouts. Our GPU implementation consists

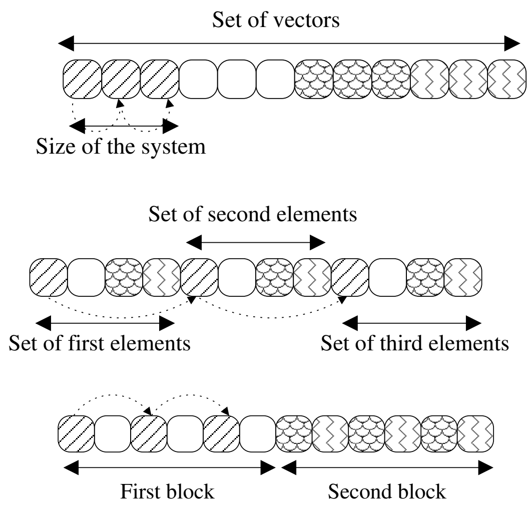

Figure 2: Example of the different data layouts proposed, *Flat* (top), *Full-Interleaved* (center), *Block-Interleaved* (bottom) with a *BS* equal to 2, for four Hines systems of three elements each. Point-line represents the jumps in memory carried out by the first thread/system.

of using one thread per Hines system. As commented in Section 1, we decided to explore this approach to avoid dealing with atomic accesses and synchronizations, as well as to be able to execute a very high number of neurons of any size. Using the *Flat* data layout we can not exploit coalescence; however by interleaving (*Full-Interleaved* data layout) the elements of the vectors (u, l, d, rhs and p in Pseudocode 1), contiguous threads access to contiguous memory locations. Although, we exploit coalescence in the memory accesses by using this approach, the threads have to jump in memory as many elements as the number of systems to access the next element of the vector(s) (Point-lines in Figure 2). This could cause an inefficient use of the memory hierarchy. This is why we study an additional approach, the called *Block-Interleaved* data layout. Using this approach we reduce the number of elements among consecutive elements of the same system, and so the jumps in memory are not as big as in the previous approach (*Full-Interleaved*), while keeping the coalesced memory accesses. Also, the use of the *Block-Interleaved* data layout can take advantage better of the growing importance of the bigger and bigger cache memories in the memory hierarchy of the current and upcoming CUDA architectures.

3.1 Implementation based on Shared Memory

Unlike the previous approaches, here we explore the use of shared memory for our target application. The shared memory is much faster than the global memory; however it presents some important constraints to deal with. This memory is useful when the same data can be reused either by the same thread or by other thread of the same block of threads (CUDA block). Also, it is small (until 48KB in the architecture used) and its use hinders the exchange among blocks of threads by the CUDA scheduler to overlap accesses to global memory with computation.

As we can see in Pseudocode 1, in our problem the elements of the vectors a, d, b, rhs and p are reused in the *Forward Sweep* after computing the *Backward Sweep*. However, the granularity used (1 thread per system) and the limit of the shared memory (48KB) prevents from storing all the vectors in shared memory. To be able to use shared memory we have to use the *Block-Interleaved* data layout. The number of systems to be grouped (*BS*) is imposed by the size of the shared memory. In order to address the limitation of shared memory, we only store the *rhs* vector, as this is the vector on which more accesses are carried out. In this sense, the more systems are packed in shared memory, the more accesses to shared memory are carried out.

4 Performance Analysis

To carry out the experiments, we have used an heterogeneous node[1] composed of 2× Intel Xeon E5-2630v3 (Haswell) with 8 cores and 20 MB L3 cache each, and 2× K80 NVIDIA GPU (Kepler) with a total of 4992 cores and 24 GB GDDR5 of global memory each. Each K80 is composed of 2×logic GPUs similar to K40. This node is a Linux (Red Hat 4.4.7-16) machine, on which we have used the next configuration (compilers version and flags): gcc 4.4.7, nvcc (CUDA) 7.5, -O3, -fopenmp, -arch=sm_37. The code evaluated in this section is available in a public access repository[2].

To evaluate the different implementations described in the previous section, we have used real configurations (neurons' morphologies)[3]. In particular, 6 different neurons were used, which can be divided into 6 different categories regarding their sizes and number of branches. More details are described in Table 1. We have considered these 6 different morphologies, as a wide range of the neurons fall into the chosen morphologies.

Name	**Size**	**#Branches**	**Code Name**	**neuron ID**
small-low	76	7	299-DG-IN-Neuron2	NMO_00076
small-high	76	29	202-2-19nj	NMO_00076
medium-low	305	30	59D-40X	NMO_00302
medium-high	319	157	Culture-9-5	NMO_00319
big-low	695	66	28-2-2	NMO_00695
big-high	691	341	HSE-fluoro02	NMO_00691

Table 1: Summary of the neurons used.

In this section, 5 different implementations are analyzed. One is based on OpenMP *Multicore* using the *Flat* data layout (Section 3). This implementation makes use of an OpenMP pragma (#pragma omp for) on the top of the for loop which goes over the different independent Hines systems to distribute blocks of systems over the available cores. The rest of implementations are based on GPU. Basically, we have one implementation per each of the data layout described: *Flat*, *Full-Interleaved* and *Block-Interleaved.* Additionally, we study the use of shared memory (*Block-Shared*) over *Block-Interleaved.* There are multiple different configurations regarding the block-size (*BS*) and CUDA block for the last two scenarios (*Block-Interleaved* and *Block-Shared*). For sake of clarity we focus on one of the possible test cases to evaluate these two approaches. The benefit shown for these two implementations is similar to the rest of test-cases.

First, we evaluate the *Multicore*, *Flat* and *Full-Interleaved* for 256, 2,560, 25,600 and 256,000 *medium-high* (Table 1) neurons (Fig. 3). On those test cases that do not compromise a high number of neurons (256 and 2,560), *Multicore* obtains better performance than the GPU-based

[1]MinoTauro, `https://www.bsc.es/ca/innovation-and-services/supercomputers-and-facilities/minotauro`
[2]BSC-GitLab, `https://pm.bsc.es/gitlab/imartin1/cuHinesBatch`
[3]`http://www.neuromorpho.org/`

implementations. This is mainly because of the parallelism of these tests, which is not enough to saturate GPU and this can not reduce the impact of the high latency by overlapping execution and memory accesses. The use of multicore (16 cores and 2 sockets) supposes a speedup (over sequential execution) about 2 for 256 neurons and about 6 for 256,000 neurons. As shown, *Flat* is not able to scale, even on those test-cases that involve a high number of neurons, being even slower than multicore execution, achieving a maximum speedup about 2. This is because of the memory access pattern which can not exploit coalescing (contiguous threads access to contiguous memory locations). On the other hand, *Full-Interleaved* turns up as the best choice, being faster than *Multicore* and *Flat*, when dealing with a high number of neurons (25,600 and 256,000). Unlike *Flat*, *Full-Interleaved* takes advantage of coalescing when accessing to global memory. As expected, this has an impressive impact on performance, being *Full-Interleaved* about 20× and 25× faster than sequential code when computing 25,600 and 256,000 neurons respectively on one K80 GPU. The use of multiple GPUs is only beneficial on those test cases with an enough computational load where a high number of neurons must be computed (25,600 and 256,000 neurons), with an extra benefit close to the ideal scaling (about 1.9× faster than using one K80 GPU).

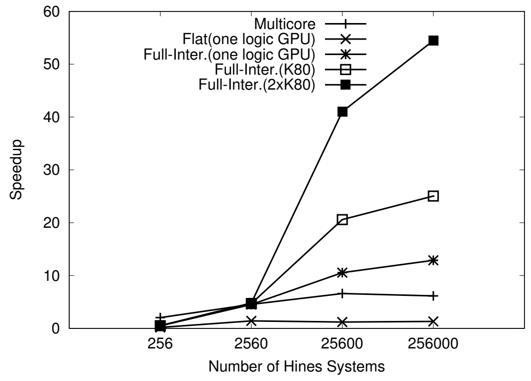

Figure 3: Performance (speedup over sequential execution) achieved by multicore (16 cores, 2 sockets) and the GPU-based approaches, *Flat* and *Full-Interleaved* (using different number of GPUs), using medium-high neurons.

As it is not possible to have the control on CUDA scheduler, we have explored a high number of different combinations regarding block-size (*BS*) for the *Block-Interleaved* approach (Sec. 3). For sake of clarity, and given the huge number of different test-cases possible, we have focused on one particular scenario. It consists of computing 256,000 medium-high neurons using different block sizes (*BS*) and fixing the size of the CUDA block (number of threads per block). This is a characteristic case among the tests carried out, as the features (sizes and number of branches) of the morphology used is in between of the other two morphologies. As shown in Fig. 4-left, some of the cases are slightly better than the *Full-Interleaved* approach, being about a 2% faster.

Next we analyze the performance of the *Block-Shared* implementation. We focus on the same scenario used for the *Block-Interleaved*. Fig. 4-right graphically illustrates the performance achieved by the *Block-Shared* and the other approaches. Although using shared memory is better than the performance achieved by the *Flat* approach, it is much smaller than the *Full-Interleaved* counterpart. For this particular scenario (medium-high morphology), a very low number of systems saturate the capacity of the shared memory (48KB). Also, the data reuse

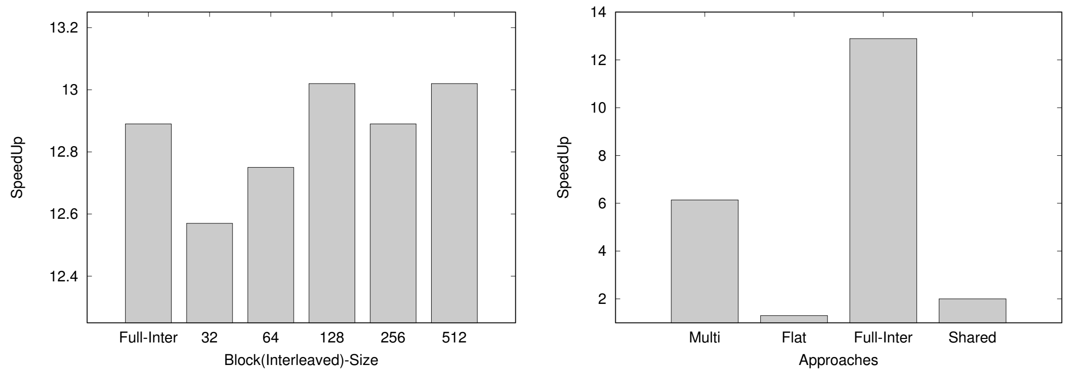

Figure 4: (Left) Performance (speedup over sequential execution) achieved by the *Block-Interleaved* approach for multiple *BS* (32, 64, 128, 256, 512) for a CUDA Block size equal to 128. (Right) Performance (speedup over sequential execution) achieved by the *Block-Shared* implementation, *Flat*, *Full-Interleaved* (Full-Inter) and *Multicore* (Multi) using 16 cores. The test-case consisted of computing 256,000 medium-high neurons, using one of the two logic GPUs in one K80 NVIDIA GPU.

is low using one-thread per Hines system. These drawbacks do not allow to achieve a better performance when the shared memory is used.

Finally, we evaluate the impact on performance of the particularities of each of the morphologies (Table 1). The performance achieved by the *Flat* is not included as it was proven to be very inefficient. As shown in Fig. 5, both approaches, *Multicore* and *Full-Interleaved*, show a similar trend in performance independently of the neurons' morphology. In particular the peak speedup achieved on the different morphologies does not vary significantly (47×-55×).

After comparing the performance achieved by Multicore and GPU, now we focus on evaluating the efficiency of our GPU implementation. To do that, we make use of *nvprof*[4]. We do not obtain results very different depending on the input (number and shape of neurons). In all cases, we obtain more than 99% of efficiency (sm_efficiency). It is also achieved a bandwidth (Global Load Throughput) close to 160GB/s, being the theoretical peak equal to 240 GB/s and the effective about the bandwidth achieved by our implementation. As most of the GPU applications, our implementation is memory bound and this is reflected by a low occupancy (about 24%).

5 Final Remarks

In this work several optimizations on NVIDIA GPUs for the Hines Algorithm have presented. Three different GPU-based implementations were deeply described and analyzed. This computation is indeed one of the main bottlenecks in the simulation of the Human Brain.

Block-Interleaved is positioned as the fastest approach against the others when dealing with a high number of neurons. However this implementation is difficult to tune being not possible to know the best configuration in advance. In contrast, *Full-Interleaved* is almost as fast as *Block-Interleaved* and is not in need of being tuned a priori. It is important to highlight that both approaches require to modify the data layout by interleaving the elements of the vectors.

[4]nvprof -m achieved_occupancy,sm_efficiency,gld_throughput,gst_throughput,gld_efficiency, gst_efficiency ./run

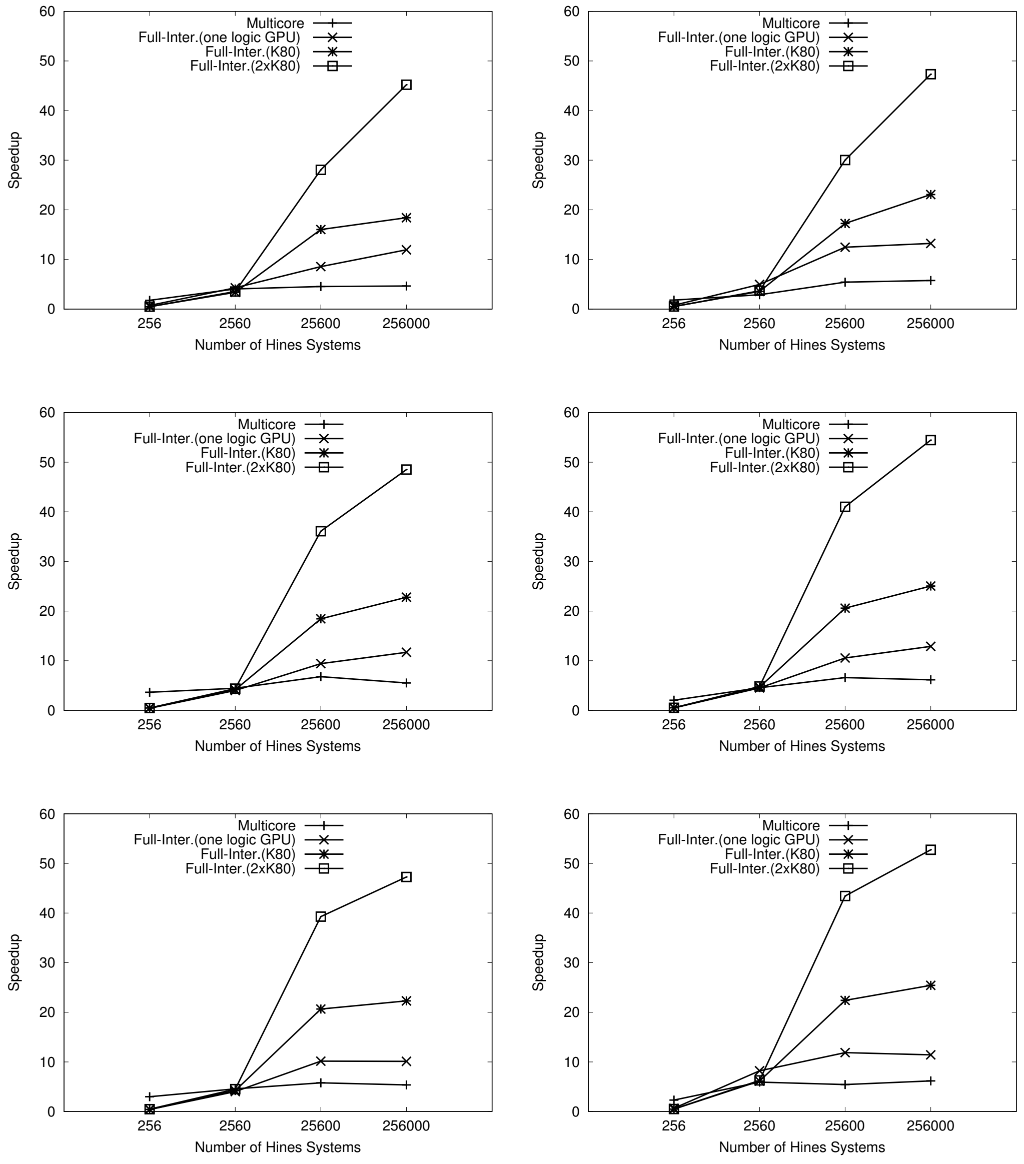

Figure 5: Performance (speedup over sequential execution) achieved for computing multiple (256, 2,560, 25,600, 256,000) neurons using different morphologies: *small-low* (top-left), *small-high* (top-right), *medium-low* (center-left), *medium-high* (center-right), *big-low* (bottom-left) and *big-high* (bottom-right).

This preprocessing compromises an irrelevant cost with respect to the whole process, as for our target application (the simulation of the Human Brain), this is carried out just once at the very beginning of the simulation. Using *Full-Interleaved* we obtain a similar behavior in terms of performance when different morphologies are considered. This is particularly interesting to evaluate the scalability and robustness of the implementation. As overview, while multicore

(16 cores and 2 sockets) gives us a maximum speedup against sequential execution about 6×, using 2×K80 NVIDIA GPUs it is achieved a peak speedup about 55×.

As future work, we plan to implement a version of *Full-Interleaved* that can deal with different morphologies in parallel.

References

[1] Roy Ben-Shalom, Gilad Liberman, and Alon Korngreen. Accelerating compartmental modeling on a graphical processing unit. *Frontiers in Neuroanatomy*, 7:4, 2013.

[2] Samuel Daniel Conte and Carl W. De Boor. *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill Higher Education, 3rd edition, 1980.

[3] cuSPARSE. Nvidia-cuda toolkit documentation. *http://docs.nvidia.com/cuda/cusparse/*.

[4] Andrew A. Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Anchorage, Alaska, USA*, pages 956–965, May 2011.

[5] Ecole Polytechnique Federale de Lausanne (EPFL). The Blue Brain Project. *http://bluebrain.epfl.ch/*.

[6] Sandra Diaz-Pier, Mikal Naveau, Markus Butz-Ostendorf, and Abigail Morrison. Automatic generation of connectivity for large-scale neuronal network models through structural plasticity. *Frontiers in Neuroanatomy*, 10:57, 2016.

[7] European Commission Future and Emerging Technologies Flagship. Human Brain Project (HBP). *https://www.humanbrainproject.eu/*.

[8] Michael Hines. Efficient computation of branched nerve equations. *International Journal of Bio-Medical Computing*, 15(1):69 – 76, 1984.

[9] National Institutes of Health. Brain research through advancing innovative neurotechnologies (brain). *https://www.braininitiative.nih.gov/about/index.htm*.

[10] Harold S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM*, 20(1):27–38, January 1973.

[11] Pedro Valero-Lara. Multi-gpu acceleration of DARTEL (early detection of alzheimer). In *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, pages 346–354, 2014.

[12] Pedro Valero-Lara, Poornima Nookala, Fernando L. Pelayo, Johan Jansson, Serapheim Dimitropoulos, and Ioan Raicu. Many-task computing on many-core architectures. *Scalable Computing: Practice and Experience*, 17(1):32–46, 2016.

[13] Pedro Valero-Lara and Fernando L. Pelayo. Towards a more efficient use of gpus. In *International Conference on Computational Science and Its Applications, ICCSA 2011, Santander, Spain, June 20-23, 2011*, pages 3–9, 2011.

[14] Pedro Valero-Lara and Fernando L. Pelayo. Analysis in performance and new model for multiple kernels executions on many-core architectures. In *IEEE 12th International Conference on Cognitive Informatics and Cognitive Computing, ICCI*CC 2013, New York, NY, USA, July 16-18, 2013*, pages 189–194, 2013.

[15] Pedro Valero-Lara, Alfredo Pinelli, Julien Favier, and Manuel Prieto-Matías. Block tridiagonal solvers on heterogeneous architectures. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA, Leganes, Madrid, Spain*, pages 609–616, July 2012.

[16] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matías. Fast finite difference poisson solvers on heterogeneous architectures. *Computer Physics Communications*, 185(4):1265–1272, 2014.

[17] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, Bangalore, India*, pages 127–136, January 2010.