Oil & Gas Science and Technology - Rev. IFP Energies nouvelles **73**, 63 (2018)
© P. Valero-Lara et al., published by IFP Energies nouvelles, 2018
https://doi.org/10.2516/ogst/2018061

Available online at:
www.ogst.ifpenergiesnouvelles.fr

Numerical methods and HPC
A. Anciaux-Sedrakian and Q.H. Tran (Guest editors)

REGULAR ARTICLE OPEN ACCESS

Simulating the behavior of the Human Brain on GPUs

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

[1] Barcelona Supercomputing Center (BSC), C/Jordi Girona, 29, 08034 Barcelona, Spain
[2] Universidad Politècnica de Cataluña, Carrer de Jordi Girona, 1, 3, 08034 Barcelona, Spain

Received: 23 February 2018 / Accepted: 11 September 2018

Abstract. The simulation of the behavior of the Human Brain is one of the most important challenges in computing today. 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 and, 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 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. Two different approaches are studied, one for mono-morphology simulations (batch of neurons with the same shape) and one for multi-morphology simulations (batch of neurons where every neuron has a different shape). In mono-morphology simulations we obtain a good performance using just a single kernel to compute all the neurons. However this turns out to be inefficient on multi-morphology simulations. Unlike the previous scenario, in multi-morphology simulations a much more complex implementation is necessary to obtain a good performance. In this case, we must execute more than one single GPU kernel. In every execution (kernel call) one specific part of the batch of the neurons is solved. These parts can be seen as multiple and independent tridiagonal systems. Although the present paper is focused on the simulation of the behavior of the Human Brain, some of these techniques, in particular those related to the solving of tridiagonal systems, can be also used for multiple oil and gas simulations. Our studies have proven that the optimizations proposed in the present work can achieve 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 NVIDIA K80 GPUs respectively. Also, it is important to highlight that these optimizations can continue scaling, even when dealing with a very high number of neurons.

1 Motivation

Today, we can find multiple initiatives that attempt to simulate the behavior of the Human Brain by computer [1–3]. 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 three major steps [4]: 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 time consuming 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 11 billion 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 [5], which is based on the Thomas algorithm [6], that solves tridiagonal systems. Although the use of GPUs to compute the Thomas algorithm has been deeply studied [7–11], the differences among these two algorithms, Hines and Thomas, make us impossible to use the last one, as this cannot deal with the sparsity of the Hines matrix.

The solving of one Hines system can be also seen as a set of independent and non-independent triangular systems, which could be solved by using the Thomas algorithm. Previous works [12] have explored the use of other algorithms based on the Stone's method [13]. Unlike Thomas algorithm, this method is parallel. However, it is in need

* Corresponding author: pedro.valero@bsc.es

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 presents 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 (48 KB).

Unlike the work presented in [12], 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 explored the use of GPUs to compute multiple independent problems in parallel without transforming the data layout [14–17], the particular characteristics of the sparsity of the Hines matrices force us to modify the data layout to efficiently exploit the memory hierarchy of the GPUs (coalescing accesses to GPU memory).

The present work extends the previously published work [18] with additional contributions. This work includes a complete new approach to deal with one of the most important challenges in the simulation of the Human Brain, that is, dealing with simulations which involve neurons with different morphologies (*multi-morphology* simulations). To deal with this particular scenario, we must compute parts of the batch of neurons separately. These parts can be seen as multiple and independent tridiagonal systems. While the present paper is focused on the simulation of the behavior of the Human Brain, some of these techniques, in particular those related to the solving of tridiagonal systems, can be also used for multiple oil and gas simulations.[1]

This article 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 for *mono-morphology* simulations, as well as the parallel strategies envisaged to optimally enhance the performance. Section 4 shows the strategies proposed for dealing with the challenges presented in the *multi-morphology* simulations. The state-of-the-art references and the differences between these and the present work are presented in Section 5. Finally, the conclusions are outlined in Section 6.

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:

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

where f and g are functions on x-dimension and the current I and capacitance C [4] 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 follows:

$$\text{Upper diagonal}: a_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2}$$

$$\text{Lower diagonal}: b_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2}$$

$$\text{Diagonal}: d_i = \frac{C_i}{\Delta_t} - (a_i + b_i)$$

$$\text{rhs}: \; r_i = \frac{C_i}{\Delta_t}V_i^n - I - a_i(V_{i-1}^n - V_i^n) - b_i(V_{i+1}^n - V_i^n)$$

a_i and b_i are constant in the time, and they are computed once at start up. Otherwise, the diagonal (d) and right-hand-side (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.

For the 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 Fig. 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.

[1] https://tridiagonal.com/oil-and-gas

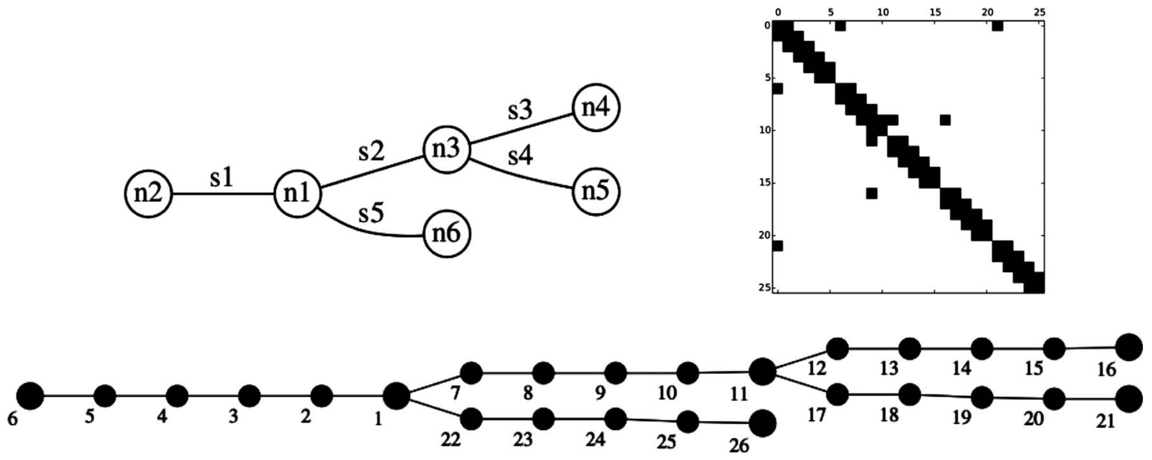


Fig. 1. Example of a neuron morphology and its numbering (left-top and bottom) and sparsity pattern corresponding to the numbering followed (top-right) [19].

Algorithm 1 Hines algorithm.

```
void solveHines(double*u, double *l, double *d,
double *rhs, int *p, int cellS ize)
// u ^ upper vector, l ^ lower vector
inti;
double factor;
// Backward Sweep
for i = cellS ize - 1 ^ 0 do
    factor = u[i] / d[i];
    d[p[i]] -= factor x l[i];
    rhs[p[i]] -= factor x rhs[i];
end for
[0] d[/= [0] hs r
// Forward Sweep
for i = 1 ^ cellS ize - 1 do
    rhs[i] -= l[i] x rhs[p[i]];
    rhs[i] /= d[i];
end for
```

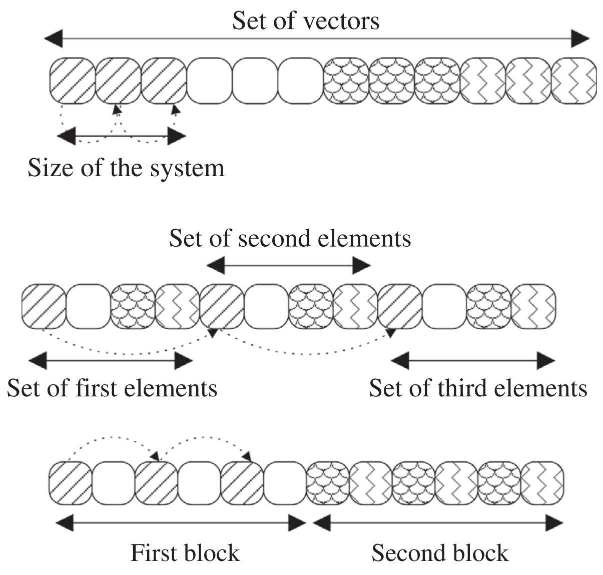


Fig. 2. Example of the different data layouts proposed, *Flat* (top), *Full-Interleaved* (center), *Block-Interleaved* (bottom) with a *BS* equal to 2, for four Hines systems of three elements each. Dotted lines represent the jumps in memory carried out by the first thread/system.

3 Implementation of batch mono-morphology Hines on GPUs

Efficient memory management is critical to achieve a good performance, but even much more on those architectures based on high throughput and high memory latency, such as 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, we start by storing 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 into 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 the 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 four vectors per Hines system (Pseudocode 1) on which the strategies above described are carried out. 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

Table 1. Summary of the neurons used.

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

(coalescing memory accesses). This is the main motivation behind the proposal of the different data layouts.

Our GPU implementation consists 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 cannot 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 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) (dotted lines in Fig. 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 hence 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 better advantage of the growing importance of the bigger and bigger cache memories in the memory hierarchy of the current and upcoming GPU 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, but 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 (up to 48 KB 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 (48 KB) prevent 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.

3.2 Performance analysis

For the experiments, we have used a heterogeneous node[2] 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 4.4.716) 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.[3]

To evaluate the different implementations described in the previous section, we have used real configurations (neurons' morphologies).[4] In particular, six different neurons were used, which can be divided into six different categories regarding their sizes and number of branches. More details are described in Table 1. We have considered these six different morphologies, as a wide range of the neurons fall into the chosen morphologies.

In this Section, five different implementations are analyzed. One is based on OpenMP *Multicore* using the *Flat* data layout (Sect. 3), which 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

[2] MinoTauro, https://www.bsc.es/ca/innovation-and-services/supercomputers-and-facilities/minotauro

[3] BSC-GitLab, https://pm.bsc.es/gitlab/imartin1/cuHines-Batch

[4] http://www.neuromorpho.org/

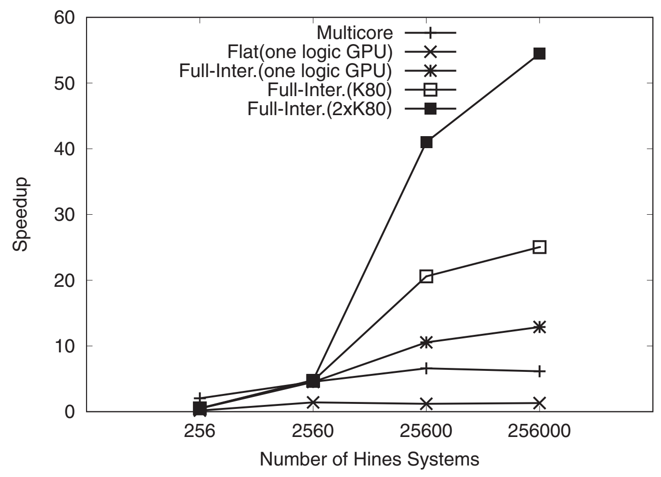


Fig. 3. Performance (speedup over sequential execution) achieved by *Multicore* (16 cores, 2 sockets) and the GPU-based approaches, *Flat* and *Full-Interleaved* (using different number of GPUs), using medium-high neurons.

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, 2560, 25 600 and 256 000 *mediumhigh* (Tab. 1) neurons (Fig. 3). On those test cases that do not compromise a high number of neurons (256 and 2560), *Multicore* obtains better performance than the GPU-based implementations. This is mainly because of the parallelism of these tests, which is not enough to saturate GPU and this cannot 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 of about 2. This is because of the memory access pattern which cannot exploit coalescing (contiguous threads access to contiguous memory locations). On the other hand, *Full-Interleaved* turns out 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).

As it is not possible to have control on the CUDA scheduler, we have explored a high number of different combinations regarding block-size (*BS*) for the *Block-Interleaved* approach (Sect. 3). For the sake of clarity, and given the huge number of different possible test-cases, 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 Figure 4a, 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*. Figure 4b 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 (48 KB). Also, the data reuse 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 (Tab. 1). The performance achieved by the *Flat* is not included as it was proven to be very inefficient. As shown in Figure 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.*[5] We do not obtain very different results depending on the input (number and shape of neurons). In all cases, we obtain more than 99% efficiency (sm_efficiency), as well as a bandwidth (Global Load Throughput) close to 160 GB/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%).

3.3 Remarks

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. It is 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. 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

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

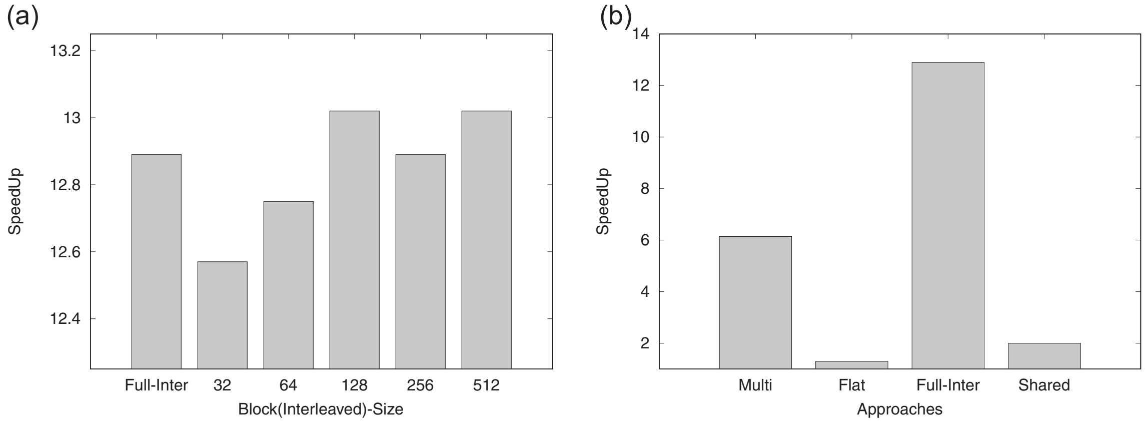


Fig. 4. (a) Performance (speedup over sequential execution) achieved by the *Block-Interleaved* approach for multiple *BS* (32, 64, 128, 256, 512) for a CUDA Block size equal to 128. (b) Performance (speedup over sequential execution) achieved by the *Block-Shared* implementation, *Flat, Full-Interleaved* (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.

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 of about 6×, reporting a peak speedup of about 55×, using 2× K80 NVIDIA GPUs.

4 Implementation of batch multi-morphology Hines on GPUS

In this section, we analyze the performance for the simulation of the behavior of the Human Brain using different morphologies *(multi-morphology)* in parallel. This study is the most important contribution of the present work. Although the simulations that involve mono-morphology (the same neuron replicated) scenarios can also be useful, in real world cases, the neurons are completely different among them. First, we evaluate the optimizations above described for this particular case. Figure 6 graphically illustrates the performance achieved for *multi-morphology* simulations using the strategies described in the previous section for the *mono-morphology* approach. To do this, we have used two different test cases. In both cases we use the same size to evaluate only the influence on performance for computing *multi-morphology* simulations. We generate different Hines matrices with a different ratio (% of branches with respect to the size of branches). For the sake of comparison, we also include those cases for *mono-morphology* simulations using the same morphology, size and branches ratio for the batch of neurons (Mono in Fig. 6). As shown (Fig. 6), when dealing with *multi-morphology* cases, we found an important fall in performance with respect to *mono-morphology* simulations. The most important cause of this behavior is given by the lack of coalescing memory accesses. When computing different morphologies in parallel, the off-diagonal elements (see Fig. 1 and Pseudocode 1) of these Hines matrices (neurons) are located in different positions from one to other, which makes difficult that consecutive CUDA threads of the same CUDA block access to contiguous memory positions causing an important fall in performance.

To minimize this problem, we propose a different approach which consists of computing each of the branch levels separately. Each branch can be seen as a tridiagonal system, so those branches of the same level could be computed simultaneously in one CUDA kernel. For the sake of clarity, Figure 7 shows a simple diagram with this idea. In the rest of this section, we focus on the implementation of a kernel, which makes use of some of the ideas previously presented, but to solve tridiagonal systems instead of Hines systems. At the end of this section, we evaluate the impact of this idea for *multi-morphology* simulations.

4.1 Tridiagonal linear systems

The state-of-the-art method to solve tridiagonal systems is the called Thomas algorithm [8], which a specialized application of the Gaussian elimination that takes into account the tridiagonal structure of the system. It consists of two stages, commonly denoted as forward elimination and backward substitution.

Given a linear $Au = y$ system, where $\boldsymbol{A}$ is a tridiagonal matrix:

$$\boldsymbol{A} = \begin{bmatrix} b_1 & c_1 & & & & 0 \\ a_2 & b_2 & c_2 & & & \\ & . & . & . & & \\ & & . & . & . & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \\ & & & & & \end{bmatrix}.$$

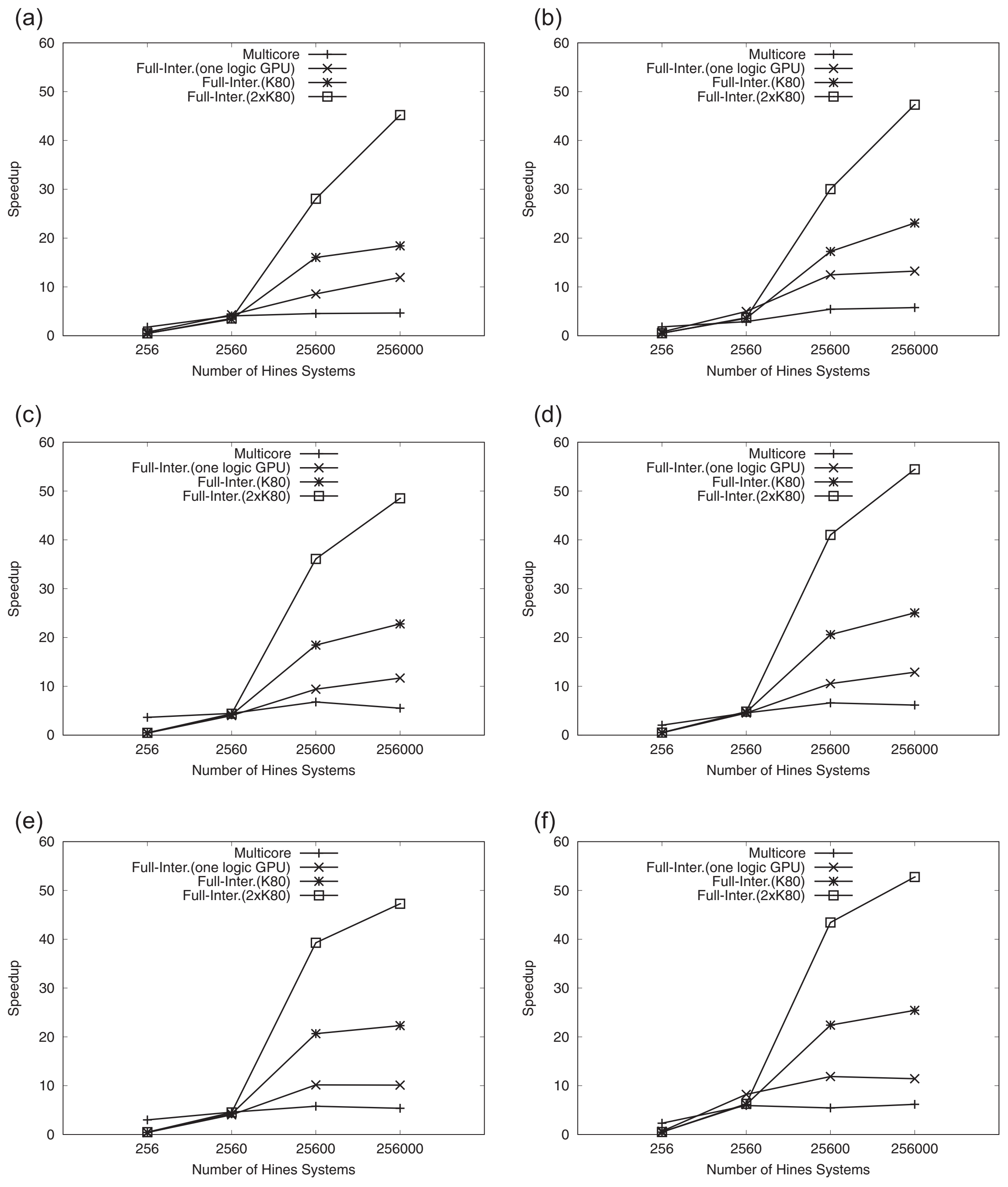


Fig. 5. Performance (speedup over sequential execution) achieved for computing multiple (256, 2560, 25 600, 256 000) neurons using different morphologies: *small-low* (a), *small-high* (b), *medium-low* (c), *medium-high* (d), *big-low* (e) and *big-high* (f).

The forward stage eliminates the lower diagonal as follows:

$$c_1' = \frac{c_1}{b_1},\ c_1' = \frac{c_1}{b_i - c_{i-1}' a_i} \quad \text{for } i = 2, 3, \ldots, n-1$$

$$y_1' = \frac{y_1}{b_1},\ y_1' = \frac{y_i - y_{i-1}' a_i}{b_i - c_{i-1}' a_i} \quad \text{for } i = 2, 3, \ldots, n-1$$

and then the backward stage recursively solves each row in reverse order:

$$u_n = y_n',\ u_i = y_i' - c_i' u_{i+1} \quad \text{for } i = n-1, n-2, \ldots, 1.$$

Overall, the complexity of Thomas algorithm is optimal: $8n$ operations in $2n - 1$ steps.

Cyclic Reduction (CR) [7, 8, 20, 21] is a parallel alternative to Thomas algorithm. It also consists of two phases (reduction and substitution). In each intermediate step of

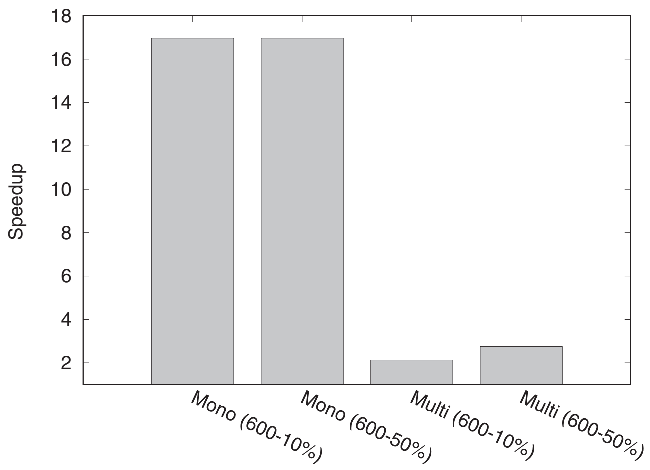


Fig. 6. Performance (speedup over sequential execution) achieved for computing 25 600 neurons using mono-morphologies (Mono) and multi-morphologies (Multi) of the same size and different percentages of branches (10% and 50%).

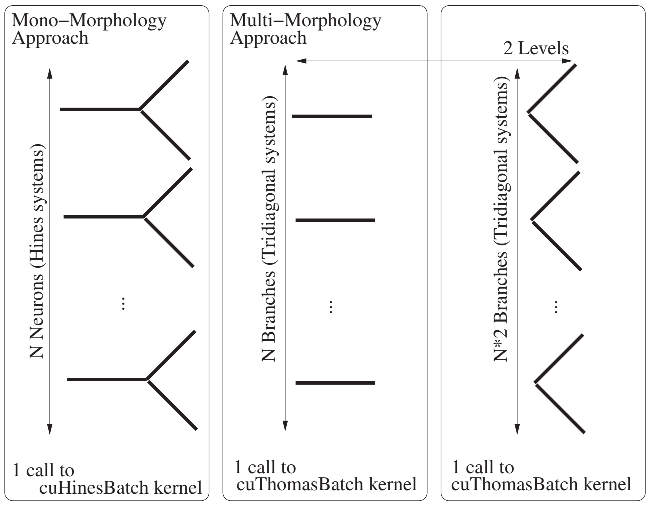

Mono–Morphology Approach
N Neurons (Hines systems)
1 call to cuHinesBatch kernel
Multi–Morphology Approach
N Branches (Tridiagonal systems)
1 call to cuThomasBatch kernel
2 Levels
N*2 Branches (Tridiagonal systems)
1 call to cuThomasBatch kernel

Fig. 7. Mono-Morphology (left) and Multi-Morphology (right) approaches.

the reduction phase, all even-indexed (i) equations $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ are reduced. The values of a_i, bi, ci and di are updated in each step according to:

$$a'_i = -a_{i-1}k_1, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2$$

$$c'_i = -c_{i+1}k_2, y'_i = y_i - y_{i-1}k_1 - y_{i+1}k_2$$

$$k_1 = \frac{a_i}{b_{i-1}}, k_2 = \frac{c_i}{b_{i+1}}.$$

After $\log_2 n$ steps, the system is reduced to a single equation that is solved directly. All odd-indexed unknowns x_i are then solved in the substitution phase by introducing the already computed u_{i-1} and u_{i+1} values:

$$u_i = \frac{y'_i - a'_i x_{i-1} - c'_i x_{i+1}}{b'_i}.$$

Overall, the CR algorithm needs $17n$ operations and $2\log_2 n - 1$ steps. Figure 8a graphically illustrates its access pattern.

Parallel Cyclic Reduction (PCR) [7, 8, 20, 21] is a variant of CR, which only has substitution phase. For convenience, we consider cases where $n = 2^s$, that involve $s = \log_2 n$ steps. Similarly to CR, a, b, c and y are updated as follows, for $j = 1,2,\ldots, s$ and $k = 2^{j-1}$:

$$a'_i = \alpha_i a_i, b'_i = b_i + \alpha_i c_{i-k} + \beta_i a_{i+k}$$

$$c'_i = \beta_i c_{i+1}, y'_i = b_i + \alpha_i y_{i-k} + \beta_i y_{i+k}$$

$$\alpha_i = \frac{-a_i}{b_{i-1}}, \beta_i = \frac{-c_i}{b_i}$$

finally the solution is achieved as:

$$u_i = \frac{y'_i}{b_i}.$$

Essentially, at each reduction stage, the current system is transformed into two smaller systems and after $\log_2 n$ steps the original system is reduced to n independent equations. Overall, the operation count of PCR is $12n \log_2 n$. Figure 8b sketches the corresponding access pattern.

We should highlight that, apart from their computational complexity, these algorithms differ in their data access and synchronization patterns, which also have a strong influence on their actual performance. For instance, in the CR algorithm synchronizations are introduced at the end of each step and its corresponding memory access pattern may cause bank conflicts. PCR needs less steps and its memory access pattern is more regular [20]. In fact, hybrid combinations that try to exploit the best of each algorithm have been explored [7, 8, 20–23]. CR-PCR reduces the system to a certain size using the forward reduction phase of CR and then solves the reduced (intermediate) system with the PCR algorithm. Finally, it substitutes the solved unknowns back into the original system using the backward substitution phase of CR. Indeed, this is the method implemented by the *gtsvStridedBatch* routine into the *cuSPARSE* package [11], one of the implementations evaluated in this work.

There are more algorithms, apart of the ones above mentioned, to deal with tridiagonal systems, such as those based on Recursive Doubling [20], among others. However, we have focused on those, which were proven to achieve a better performance and were implemented in the reference library [11].

4.1.1 Implementation of cuThomasBatch

In this section, we explore the different proposals about the CUDA thread mapping on the data layouts above

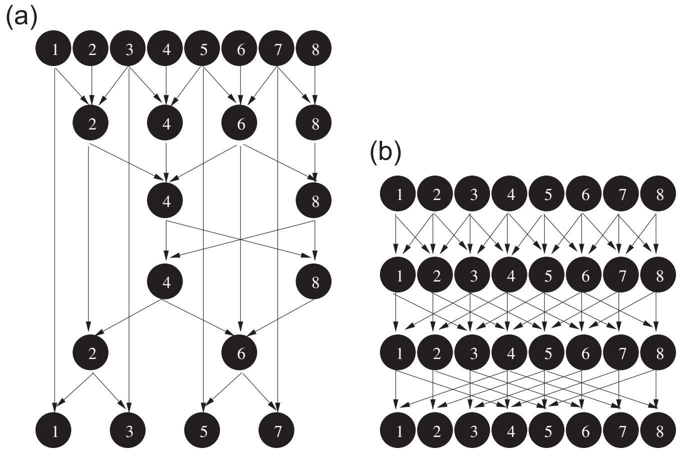


Fig. 8. Access pattern of the CR algorithm (a) and PCR algorithm (b).

described on pure-tridiagonal systems. In *cuThomasBatch* we use a coarse-grain scheme where a set of tridiagonal systems is mapped onto a CUDA block so that each CUDA thread fully solves a system. 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 tridiagonal systems of any size, without the limitation imposed by the parallel methods. As above presented, using the *Flat* data layout we cannot exploit coalescence when exploiting one thread per tridiagonal system (coarse approach); however, by interleaving (*Full-Interleaved* data layout) the elements of the vectors, contiguous threads access to contiguous memory locations. As previously described in Section 2, this approach does not exploit efficiently the shared memory of the GPUs since the memory required by each CUDA thread becomes too large. Our GPU implementation (*cuThomasBatch*) is based on this approach, *Thomas* algorithm on *Full-Interleaved* data layout. On the other hand, previous studies have explored the use of the fine-grain scheme based on CR-PCR [7, 8, 20, 21] using the *Flat* data layout. In this case, each tridiagonal system is distributed across the threads of a CUDA block so that the shared memory of the GPU can be used more effectively (both the matrix coefficients and the right hand side of each tridiagonal system are hold on the shared memory of the GPU). Nevertheless, computationally expensive operations, such as synchronizations and atomic accesses are necessary. Also this approach saturates the capacity of the GPU with a relatively low number of tridiagonal systems. Even when the shared memory is much faster than the global memory, 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 (up to 48 KB 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. Our reference implementation (the *gtsvStridedBatch* routine into the cuSPARSE package [11]) is based on this approach, CR-PCR on *Flat* data layout.

4.1.2 Performance analysis

To carry out the experiments, we have used one of the two logic Kepler GPUs into one K80 NVIDIA GPU. We have evaluated the performance of each of the approaches, *gtsvStridedBatch* and *cuThomasBatch*, using both, single and double precision operations. Two test cases were proposed. The first one (Figs. 9a and 10) consists of computing 256, 2560, 25 600 and 256 000 "small" tridiagonal systems of 64, 128, 256 and 512 elements each. Due to the memory capacity of our platform, we consider another test case (Figs. 9a and 11) for those systems with a bigger size (a higher number of elements), 1024, 2048, 4096 and 8192. In this case we could compute up to a maximum of 20 000 systems in parallel. We have considered this testbed to evaluate the scalability by increasing both, the size of the systems and the number of systems, taking into account the limitation of our platform. In particular, the size of the systems in the first test cases (64–512) can be fully executed by one CUDA block using *gtsvStridedBatch*. Nevertheless, those tests which need a higher size (1024 forward) must be computed following other strategies as commented before. Regarding the size of the tridiagonal systems, there is no characteristic size, as it depends on the nature of the applications, and because of that, we have considered different cases to cover all the range of possible scenarios. For the sake of numerical stability we force the tridiagonal coefficient matrix to be diagonally dominant ($|b_i| > |a_i| + |c_i|$, $\forall i = 0, \ldots, n$). We initialize the matrix coefficients randomly following the previous property.

Figure 9 graphically illustrates the speedup achieved by our implementation against the *cuSPARSE* routine. Even when interleaving the elements of the systems does not scale when computing a low number of systems (256 in Fig. 9a and 20–200 in Fig. 9b), being *gtsvStridedBatch* faster than our implementation, this last turns to be much faster for the rest of tests (2560–256 000 in Fig. 9a and 2000–20 000 in Fig. 9b). In most cases, independently of the size of the systems, bigger size means bigger speedup, achieving a speedup peak close to 4 in single precision and close to 3 in double precision.

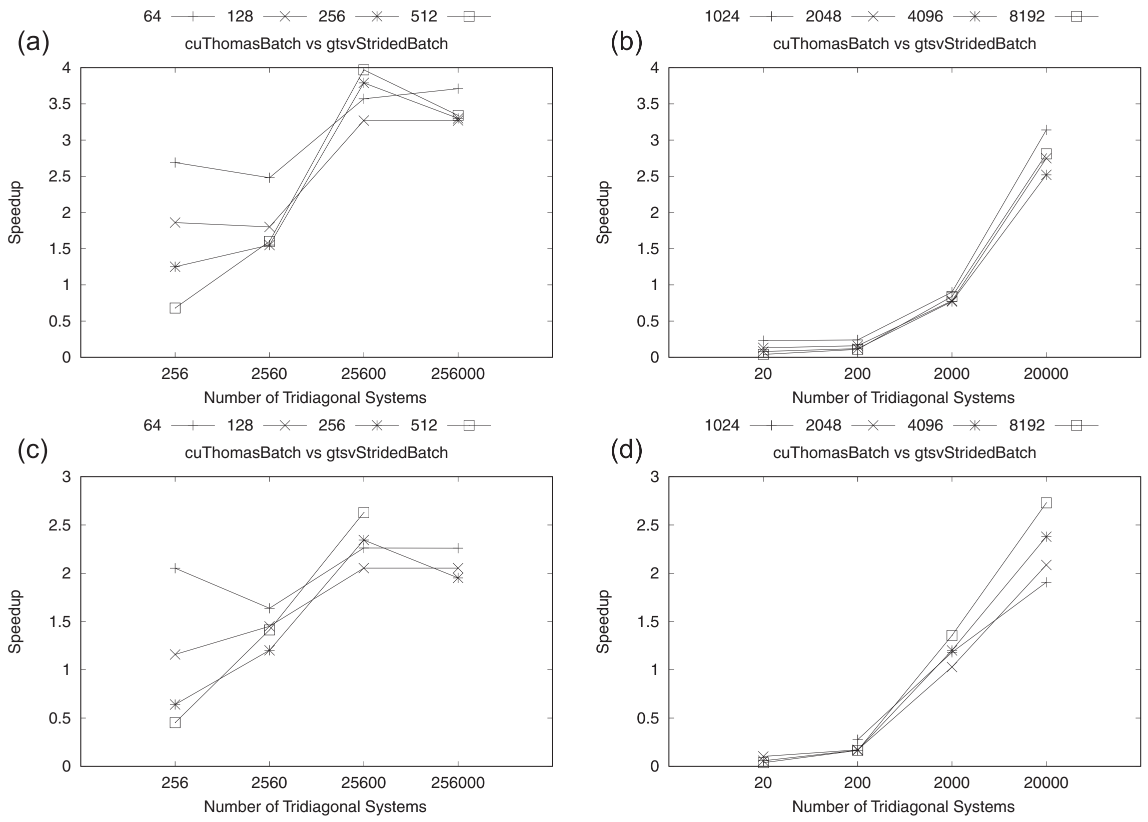


Fig. 9. cuThomasBatch performance, (execution time of *gtsvStridedBatch* divided by the execution time of cuThomas) using single (a, b) and double (c, d) operations for computing multiple, 256–256 000 (a, c) and 20–20 000 (b, d), tridiagonal systems using different sizes: 64–512 (a, c) and 1024–8192 (b, d).

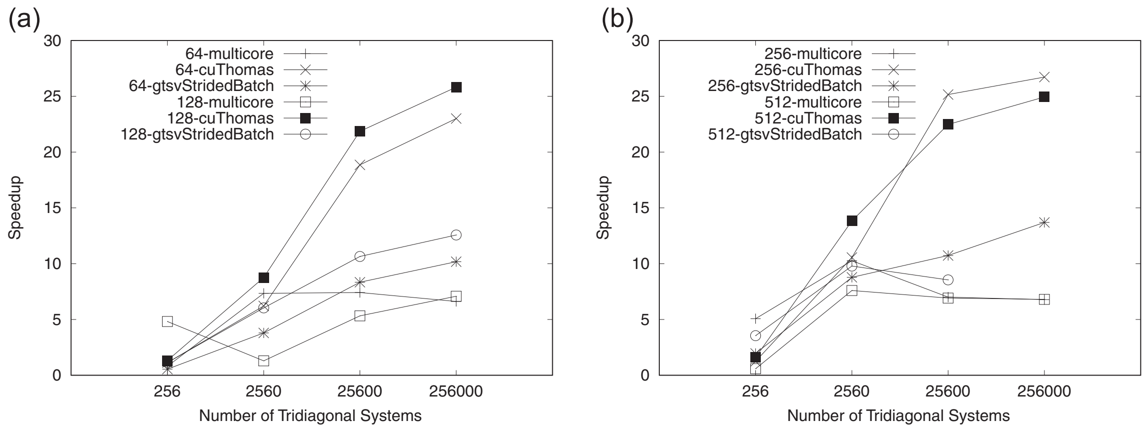


Fig. 10. Performance (speedup over sequential execution) achieved for computing multiple (256, 2560, 25 600, 256 000) tridiagonal systems using different sizes: 64, 128 (a) and 256, 512 (b).

To analyze in more detail the scalability of both implementations, we also show (Figs. 10 and 11) the speedup (for double precision operations) against the sequential counterpart including the performance achieved by the multicore execution (16 cores). The implementation based on multicore basically makes use of an OpenMP pragma *(#pragma omp for)* on top of the for loop which goes over the different independent tridiagonal systems to distribute blocks of systems over the available cores. While *gtsvStridedBatch* achieves a peak speedup about 10 from 2560 and systems, saturating the GPU capacity, *cuThomasBatch* continues scaling from 2560 (Fig. 10) and 2000 (Fig. 11) to 256 000 and 20 000, with a speedup peak of about 25. It is important to note that in some cases, the multicore OpenMP

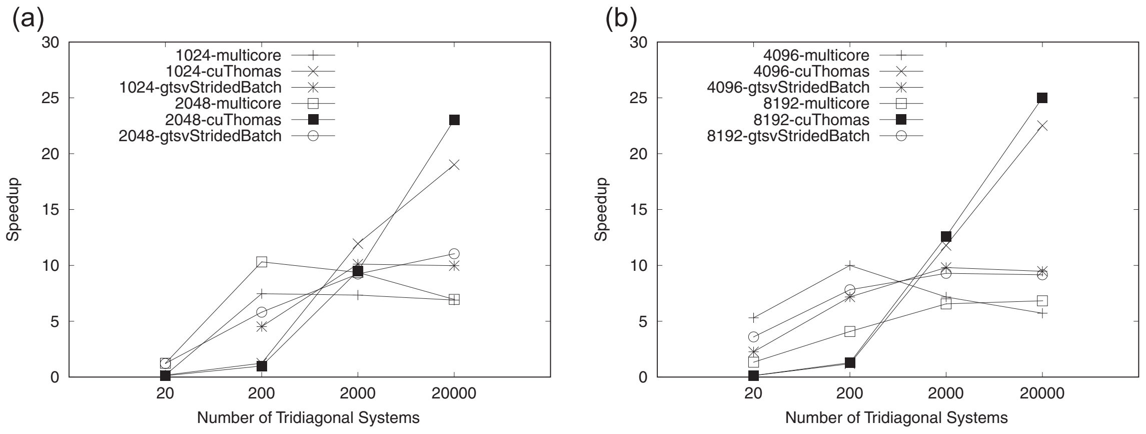


Fig. 11. Performance (speedup over sequential execution) achieved for computing multiple (20, 200, 2000, 20 000) tridiagonal systems using different sizes: 1024, 2048 (a) and 4096, 8192 (b).

implementation outperforms both GPU-based implementations for a low number of systems. This is mainly because of the parallelism of these tests, which is not enough for the GPU to reduce the impact of the high latency by overlapping execution and memory accesses.

The numerical accuracy is critical in a large number of scientific and engineering applications. In this sense, we compared the numerical accuracy of both parallel approaches against the sequential counterpart, increasing the size of the system. As shown in Figure 12a, *cuThomasBatch* presents a lower error and a more stable accuracy, being in some cases about 4× more accurate. This is because of the intrinsic characteristics of the Thomas algorithm.

As commented before, the use of parallel methods requires an additional amount of temporary extra storage [11].

In particular, *gtsvStridedBatch* is in need of $m \times (4 \times n + 2048) \times$ *size of (<type>)* more memory, being m and n the number of systems and the size of the systems respectively [11]. This supposes, for instance, that *gtsvStridedBatch* needs about 2× more memory capacity than *cuThomasBatch* to compute 20 000 tridiagonal systems of 8192 elements each (Fig. 12a).

It is important to highlight that *cuThomasBatch,* unlike the *gtsvStridedBatch*, is in need to modify the data layout by interleaving the elements of the vectors. This preprocessing does not compromise an important overhead with respect to the whole process, in those applications (numerical simulations) which have to solve multiple tridiagonal systems many times in a temporal range, as this is carried out just once at the very beginning of the simulation [18].

Finally, we have used the NVIDIA profiler to evaluate our *cuThomasBatch* in terms of occupancy and memory bandwidth achieved. In this sense, our implementation is able to achieve a high occupancy ratio of about 92% and a high bandwidth of about 140 GB/s. Although the memory bandwidth of our GPU is 240 GB/s, given that the ECC is activated, which causes a fall about 25% in the bandwidth, we obtain about 80% of the maximum bandwidth possible.

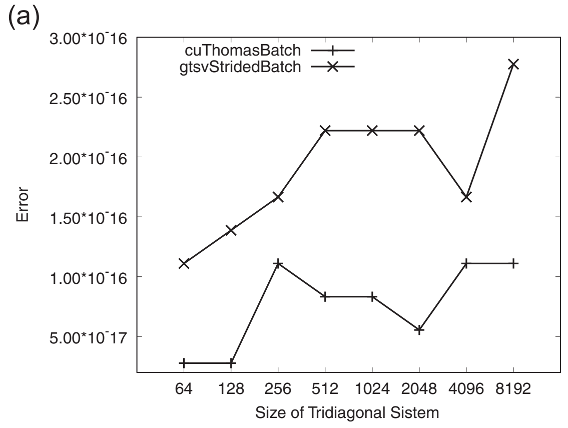
4.1.3 Variable Batch Thomas, cuThomasVBatch

Here we evaluate the variant of *cuThomasBatch* for variable batch (batch of tridiagonal systems with different sizes), *cuThomasVBatch* [24]. We study two different variants, *No Computing Padding* (NCP) and *Computing Padding* (CP). To evaluate these variants, we first initialize a batch of tridiagonal systems with a size chosen randomly between 256 and 512. We also compute two other cases to compute batches with the same size, one for 256 and one for 512 using *cuThomasBatch.* As Figure 13 illustrates, the variant based on *CP* is significantly more efficient and faster than the *NCP* counterpart. This is because, although the *NCP* needs less number of memory accesses and operations, this variant suffers from divergence and non-coalesced memory accesses, causing an important underutilization of the computational capacity of our GPU architecture. We also make use of NVPROF to extract some metrics like memory bandwidth and efficiency. While the bandwidth achieved by the *NCP* variant is about 28 GB/s, when executing 256 000 tridiagonal systems of 256–512 elements each, the *CP* is able to achieve a bandwidth of about 70 GB/s for the same test case. The efficiency is bigger using the *CP* approach (84%) than the *NCP* one (71%).

The performance (execution time) of the *CP* variant is bounded by the biggest size of the batch, as shown in Figure 13. The time for a variable batch *(cuThomasVBatch)* of 256–512 is similar to the execution time of a fixed-size batch *(cuThomasBatch)* of 512. This is because in *CP* we access and compute the *null* elements. This implies that, in terms of performance, the *CP* variant is equivalent to the execution of *cuThomasBatch* for the maximum system size computed by *CP.*

4.1.4 Remarks

cuThomasBatch and *cuThomasVBatch* are not able to saturate the GPU capacity when dealing with a low number of systems, however, they outperform the *cuSPARSE* implementation on a high number of tridiagonal systems. This is because of a simpler management of CUDA threads, as we do not have to deal with synchronizations, atomic



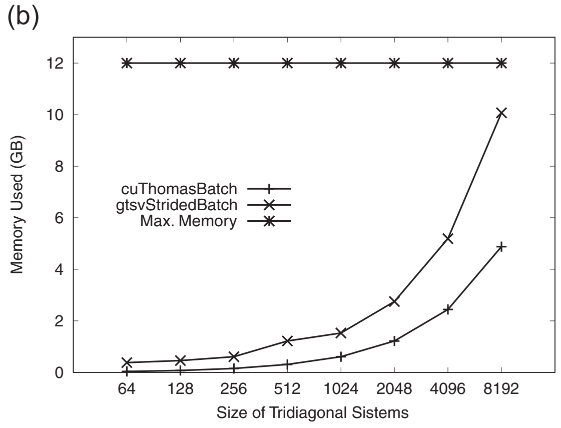


Fig. 12. Numerical accuracy achieved by both approaches, *gtsvStridedBatch* and *cuThomasBatch* (a), for double precision operations. Memory used by *gtsvStridedBatch* and *cuThomasBatch* to compute 20 000 tridiagonal systems (b) for double precision operations.

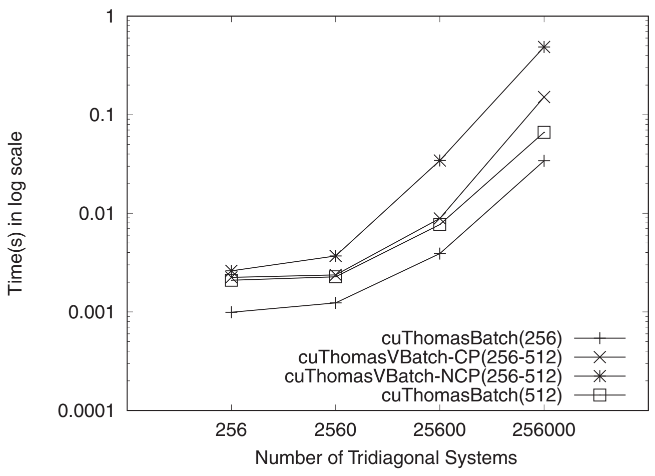


Fig. 13. Execution time (using double precision) for the two variants, *No Computing Padding* and *Computing Padding,* of the *cuThomasVBatch.*

operations and the limitations regarding the size of shared memory and CUDA blocks. The code *(cuThomasBatch)* and the optimizations carried out will be included in the next *cuSPARSE* release with the name *gtsvInterleavedBatch.*

4.2 Performance analysis of multi-morphology Hines

To carry out the experiments, we have used one of the two logic Kepler GPUs into one K80. Once the *cuThomasBatch* implementation has been evaluated, we decided to use this implementation to compute *multi-morphology* simulations, *i.e.* we use *cuThomasBatch* to compute the independent branches (tridiagonal systems) of the same level for a batch of the neurons (see Fig. 7). To carry out the performance analysis and evaluate the potential benefit of using the *multi-morphology* approach, we use a set of different morphologies composed by a different number of branch-levels (from 2 to 4). All neurons, independently of the number of branch-levels, have the same size (512).

We have followed a particular pattern to assign the size of the different branch-levels. This consists of, given a neuron size, using for the first level a size equal to half of the whole neuron, in our case 256, the size and the number of branches depend on the number of levels, following a binary-tree structure. For instance, for 4 branch-levels and a neuron size equal to 512, we have a first branch with a size equal to 256, the next level is composed by 2 branches of size equal to 32, the third branch-level is composed by 4 branches of 16 elements each and in the last level we have 8 branches of size 8.

Similarly, for two branch-levels and the same neuron size (512), we have a first branch of 256 and a second level of branches composed by two branches of 256. We could have used branches of different size in the experiments, but as the performance achieved for the variant of the *cuThomasBatch* implementation to deal with tridiagonal systems of different size is bounded by the maximum size of the batch of tridiagonal systems (branches), we decided to use the same size in order to not make more difficult the performance analysis. The motivation to use this structure is two-fold: (i) this structure is similar to the shape of the neurons and could be used for this kind of simulations, and (ii) this is one of the most characteristic structures used in multiple other applications, so that its integration in this kind of simulations could be more satisfactory than using other structures.

For the sake of comparison, we have also included in this study the performance achieved using the multi-core architecture and the *mono-morphology* approach. In the first, we compute a batch of neurons with different morphologies, all of them with a size equal to 512. In the last, we have used the same morphology for the batch of neurons with a size equal to 512. As shown, independently of the morphology used, the performance achieved for *mono-morphology* simulations is very similar (see Sect. 3.2). As in the previous analysis, we have used different number of neurons to be computed, from 256 unto 256 000.

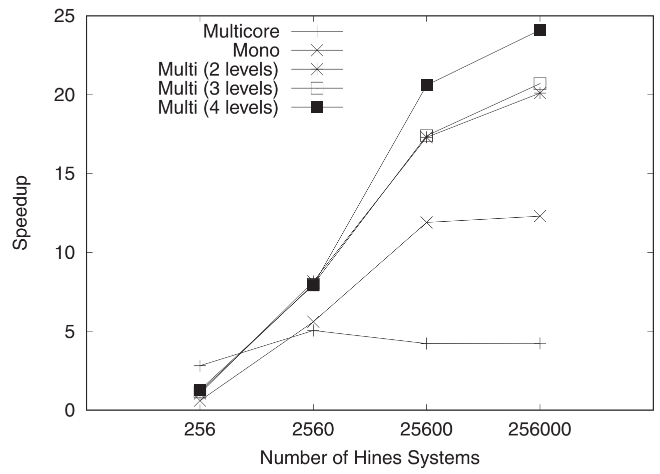


Fig. 14. Performance (speedup over sequential execution) achieved for computing from 256 to 256 000 neurons of size 512 using different morphologies on multicore architecture (Multicore in the graph) and GPU (Multi in the graph) respectively. We have also included the mono-morphology approach (Mono in the graph) for a batch of the same morphology of size 512.

It is important to note that in those cases which involve multiple levels (Multi (x level), for $x = 2$, 3 and 4, in Fig. 14), the number of tridiagonal systems (branches) to be computed is multiplied by the number of branches of such level. For instance, for the branch-levels case and 256 000 neurons, first we execute 256 000 × 8 tridiagonal systems of size 8 to compute the last level, then we compute 256 000 × 4 branches of size 16, after that we compute 256 000 × 2 tridiagonal systems of size 32 and finally the last *cuThomasBatch* kernel call computes 256 000 systems of size 256.

As we can see, the overhead of the *multi-morphology* approach due to the synchronization between the execution of the different levels of branches (see Fig. 6) is mitigated with the use of an efficient and fast solver, the *cuThomas-Batch*. It is also important to highlight that, although the *mono-morphology* approach presents a good performance (see Sect. 3.2), computing a batch of neurons with the same morphology, the memory access pattern is broken when accessing the off-diagonal elements of the Hines matrix (see Fig. 1), which makes difficult to achieve a good exploitation of the top-levels of the memory hierarchy, in particular the L2-cache. This consequence makes the multi-level execution (*multi-morphology* approach) faster, even if this has to deal with the synchronization among levels. To do this analysis we have used *nvprof*. Using the NVIDIA profiler *nvprof* we see that L2 Hits of the *multi-morphology* approach are about 60%, being this metric 2× better than in the *mono-morphology* approach. This has a direct consequence on the IPC (Instructions Per Cycle), which is about 1.5× bigger in the *multi-morphology* approach than in the *mono-morphology* approach (0.41 *vs.* 0.27). Other important metrics are L1 read/write requests and Load/Store Transactions to Global Memory, in both cases the *mono-morphology* approach requires about a 97% more requests than the *multi-morphology* approach.

The trend in performance of the *multi-morphology* approach is similar to that achieved by *cuThomasBatch* and *mono-morphology* computations on GPUs, that is, the more number of neurons to be computed the better. It is also important to note that for the 4-levels case (Multi (4 levels) in Fig. 14), the performance is better than the two levels and three levels case. This is because of a better exploitation of the memory hierarchy, which can be more efficient when the size of the tridiagonal systems (branches) is smaller.

5 Related work

In this section we explore the state-of-the-art and reference works for the parallelization of the Hines method. For the sake of clarity, we also highlight both, the differences found between these works and the work presented in this paper and the main contributions of our work. This work can be seen as an extension of the previously published paper by Valero-Lara *et al.* [18]. In this paper, the authors presented a novel implementation called *cuHinesBatch* to accelerate one of the most expensive computationally steps in the simulation of the Human Brain, that is the computation of the voltage capacitance on the neurons morphology. Although good results in terms of scalability and speedup were reported on mono-morphology simulations, this implementation turns out inefficient on real simulations, achieving a poor performance when computing neurons with different size and shape. This was clearly reported in the present work.

The main contribution of this work is a novel and highly scalable implementation able to deal with multi-morphology simulations based on *cuThomasBatch* implementation [25]. Although in this paper the *cuThomasBatch* was proven to be a fast implementation for batches of full-tridiagonal systems, this is not enough to compute the sparsity found in Hines matrices. Due to this, we proposed and developed a new approach called *cuThomasVBatch*.[6] As in other works, such as the work by Vooturi *et al.* [26], the idea to deal with multi-morphology simulations is to divide one Hines system into smaller tridiagonal systems. Vooturi *et al.* proposed two variants, one based on the use of the BLAS-2 routine *trsv,* which makes use of the cuSparse *gtsvStridedBatch* routine [11], and one called *TPT* using one thread per tridiagonal system. They reported that the first variant outperforms the second variant, achieving a peak speedup *versus* the sequential CPU code of about 3.5, when the second variant was only able to achieve a speedup of about 1.2. Our *cuThomasVBatch,* although using one thread per tridiagonal system as the second variant of Vooturi's work, is able to achieve a better performance than using the cuSparse *gtsvStridedBatch* routine, which is used in the first variant. This is mainly because of using interleaved data layouts, which allows us to exploit coalescing memory accessing on GPU memory. In fact, the

[6] The V in *cuThomasVBatch* means variable.

use of this data-layout is proven to be very efficient for batch operations not only in NVIDIA GPUs but also on INTEL multi-core processors [27]. As a proof of this, the *cuThomasBatch* implementation has been recently included into the cuSparse NVIDIA library as part of a novel routine called *gtsvInterleavedBatch* [11]. Other routines of the NVIDIA cuSparse library, such as the *gtsvInterleavedBatch*, also make use of this data-layout, as well as the INTEL MKL *compact* routines [28, 29]

The solve of tridiagonal systems is of the vital importance for multi-morphology (Hines) systems. In the literature, we can find multiple works which attempt to accelerate this kind of systems. We can divide these works into those which make use of pivoting to make the solving more numerically stable on bad-conditioned matrices, such as the works by Chang *et al.* [30], and those which assume that the matrix is well-conditioned (diagonal dominant), such as the work by Zhang *et al.* [31]. Since the Hines matrices are well-conditioned by definition, the computationally expensive operations like pivoting are not necessary. We can find multiple works for fast resolution on well-conditioned tridiagonal systems, such as the aforementioned work of Zhang *et al.* [31] and the work by László *et al.* [32]. Both works based on the use of a hybrid method to solve the tridiagonal systems, the first one is based on *PCR-CR* and the second one is based on *PCR-Thomas*. The *gtsvStridedBatch* routine of the cuSpare library is based on the Zhang's work. Unlike these works, our main target is to develop an implementation which does not saturate the GPU capacity with a relatively low number of Thomas matrices, since in our simulations we have a huge number of independent systems (neurons). Both approaches achieve a good performance for a relatively small number of systems, but cannot continue scaling when increasing the number of systems. Also, our implementation requires a significant lower amount of memory, which allows us to execute much larger simulations, as well as it is more numerically stable.

6 Final remarks

In this paper two different approaches have been presented, one for *mono-morphology* simulations (batch of neurons with the same shape) and one for *multi-morphology* simulations (batch of neurons, where every neuron has a different shape), to compute one of the most time consuming steps of the simulation of the behavior of the Human Brain, the Voltage on the morphology of the neurons. The *mono-morphology* approach is based on using the state-of-the-art and optimum Hines algorithm. This approach achieves a good performance, being about 55× faster than the sequential counterpart using 2× K80 NVIDIA GPUs and about 12× faster using one of the logic K40 GPUs of the K80 NVIDIA GPU.

Although these simulations are interesting and can be used in multiple test cases, in real world scenarios, the neurons are completely different among them. However, when the ideas explored for *mono-morphology* simulations are used for *multi-morphology* simulations, we find an important fall in performance with respect to the performance achieved on *mono-morphology* simulations. This is mainly due to the lack of coalescence in the memory accesses. To minimize this impact, we proposed a different approach for *multi-morphology* simulations, which consists of computing each of the branches of the neurons separately. Each of the branches can be seen as a tridiagonal system, and the branches of the same level can be computed in parallel, using the *cuThomasBatch* implementation. This implementation has proven to be very efficient, being faster than the routine *gtsvStridedBatch* of the NVIDIA library cuSPARSE. This implementation will be included in the next release of the cuSPARSE library with the name *gtsvInterleavedBatch*. Although the *multi-morphology* approach has to deal with the synchronization of different and independent levels of branches, using the *cuThomasBatch* implementation, we are able to achieve even a better performance than the obtained using the *mono-morphology* approach, being up to almost 25× faster than the sequential counterpart using one of the two logic GPUs of one K80 NVIDIA GPU. This is up to 2× faster than the performance achieved using the the *mono-morphology* approach.

Acknowledgments. 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), the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Parallels (2014-SGR-1051). We thank the support of NVIDIA through the BSC/UPC NVIDIA GPU Center of Excellence, and the European Union's Horizon 2020 Research and Innovation Program under the Marie Sklodowska-Curie Grant Agreement No. 749516.

References

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

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

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

4 Diaz-Pier S., Naveau M., Butz-Ostendorf M., Morrison A. (2016) Automatic generation of connectivity for large-scale neuronal network models through structural plasticity, *Front. Neuroanat.* **10**, 57. ISSN 1662-5129. doi: 10.3389/fnana.2016.00057.

5 Hines M. (1984) Efficient computation of branched nerve equations, *Int. J. Bio-med. Comput.* **15**, 1, 69–76.

6 Conte S.D., De Boor C.W. (1980) *Elementary numerical analysis: An algorithmic approach*, 3rd edn., McGraw-Hill Higher Education.

7 Valero-Lara P., Pinelli A., Prieto-Matias M. (2014) Fast finite difference Poisson solvers on heterogeneous architectures, *Comput. Phys. Commun.* **185**, 4, 1265–1272.

8 Valero-Lara P., Pinelli A., Favier J., Prieto-Matias M. (2012) Block tridiagonal solvers on heterogeneous architectures, in: *10th IEEE International Symposium on Parallel and*

Distributed Processing with Applications, ISPA, Leganes, Madrid, Spain, July, pp. 609–616.

9 Davidson A.A., Zhang Y., Owens J.D. (2011) 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, May, pp. 956–965.

10 Zhang Y., Cohen J., Owens J.D. (2010) Fast tridiagonal solvers on the GPU, in: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, Bangalore, India, January, pp. 127–136.

11 NVIDIA. *Nvidia-cuda toolkit documentation.* http://docs.nvidia.com/cuda/cusparse/.

12 Ben-Shalom R., Liberman G., Korngreen A. (2013) Accelerating compartmental modeling on a graphical processing unit, *Front. Neuroanat.* **7**, 4.

13 Stone H.S. (1973) An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, *J. ACM* **20**, 1, 27–38.

14 Valero-Lara P., Pelayo F.L. (2011) Towards a more efficient use of GPUs, in: *International Conference on Computational Science and Its Applications, ICCSA 2011*, Santander, Spain, June 20–23, pp. 3–9.

15 Valero-Lara P., Pelayo F.L. (2013) 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, pp. 189–194.

16 Valero-Lara P. (2014) Multi-gpu acceleration of DARTEL (early detection of alzheimer), in: *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014*, Madrid, Spain, September 22–26, pp. 346–354.

17 Valero-Lara P., Nookala P., Pelayo F.L., Jansson J., Dimitropoulos S., Raicu l. (2016) Many-task computing on many-core architectures, *Scalable Computing: Practice and Experience* **17**, 1, 32–46.

18 Valero-Lara P., Martínez-Perez I., Peña A.J., Martorell X., Sirvent R., Labarta J. (2017) CuHines-batch: Solving multiple Hines systems on GPUs Human Brain Project*, in: *International Conference on Computational Science, ICCS 2017*, Zurich, Switzerland, June 12–14, pp. 566–575.

19 Cumming B. (2010) Coreneuron overview. *CSCS – Swiss National Supercomputing Center.*

20 Zhang Y., Cohen J., Owens J.D. (2010) Fast tridiagonal solvers on the GPU, *SIGPLAN Not.* **45**, 5, 127–136.

21 Kim H.-S., Wu S.Z., Chang L.W., Hwu W.W. (2011) A scalable tridiagonal solver for GPUs, in: *2013 42nd International Conference on Parallel Processing*, pp. 444–453.

22 Sakharnykh N. (2010) Efficient tridiagonal solvers for adi methods and fluid simulation in: *NVIDIA GPU Technology Conference*, September.

23 Davidson A., Zhang Y., Owens J.D. (2011) An autotuned method for solving large tridiagonal systems on the GPU in: *IEEE International Parallel and Distributed Processing Symposium*, May.

24 Valero-Lara P., Martínez-Pérez I., Sirvent R., Martorell X., Peña A.J. (2019) cuThomasBatch and cuThomasVBatch, CUDA Routines to compute batch of tridiagonal systems on NVIDIA GPUs, *Concurrency and Computation: Practice and Experience.*

25 Valero-Lara P., Martínez-Perez I., Sirvent R., Martorell X., Peña A.J. (2017) NVIDIA GPUs scalability to solve multiple (batch) tridiagonal systems implementation of cuThomasBatch in: *Parallel Processing and Applied Mathematics - 12th International Conference, PPAM2017*, Lublin, Poland, Revised Selected Papers, Part I, September 10–13, pp. 243–253.

26 Vooturi D.T., Kothapalli K., Bhalla U.S. (2017) Parallelizing Hines matrix solver in neuron simulations on GPU, in: *24th IEEE International Conference on High Performance Computing, HiPC 2017*, Jaipur, India, December 18–21, pp. 388–397.

27 Dongarra J.J., Hammarling S., Higham N.J., Relton S.D., Valero-Lara P., Zounon M. (2017) The design and performance of batched BLAS on modern high-performance computing systems, in: *International Conference on Computational Science, ICCS 2017*, Zurich, Switzerland, June 12–14, pp. 495–504.

28 Intel. *Intel(r) math kernel library - introducing vectorized compact routines*, https://software.intel.com/en-us/articles/intelr-math-kernel-library-introducing-vectorized-compact-routines

29 Kim K.J., Costa T.B., Deveci M., Bradley A.M., Hammond S.D., Guney M.E., Knepper S., Story S., Rajamanickam S. (2017) Designing vector-friendly compact BLAS and LAPACK kernels, in: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*, Denver, CO, USA, November 12–17, pp. 55:1–55:12.

30 Chang L.-W., Stratton J.A., Kim H.-S., Hwu W.-W. (2012) A scalable, numerically stable, high-performance tridiagonal solver using GPUs, in: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC* '12, Salt Lake City, UT, USA, November 11–15, p. 27.

31 Zhang Y., Cohen J., Owens J.D. (2010) Fast tridiagonal solvers on the GPU, in: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010*, Bangalore, India, January 9–14, pp. 127–136.

32 László E., Giles M.B., Appleyard J. (2016) Many-core algorithms for batch scalar and block tridiagonal solvers, *ACMTrans. Math. Softw.* **42**, 4, 311–3136.