CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2018; **00**:1–13
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

cuThomasBatch & cuThomasVBatch, CUDA Routines to Compute Batch of Tridiagonal Systems on NVIDIA GPUs

Pedro Valero-Lara*[1] and Ivan Martínez-Pérez[1] and Raül Sirvent[1] and Xavier Martorell[12] and Antonio J. Peña[1]

Barcelona Supercomputing Center (BSC), Barcelona, Spain.[1], *Universitat Politècnica de Catalunya, Barcelona, Spain.*[2]

SUMMARY

The solving of tridiagonal systems is one of the most computationally expensive parts in many applications, so that multiple studies have explored the use of NVIDIA GPUs to accelerate such computation. However, these studies have mainly focused on using parallel algorithms to compute such systems, which can efficiently exploit the shared memory and are able to saturate the GPUs capacity with a low number of systems, presenting a poor scalability when dealing with a relatively high number of systems. The *gtsvStridedBatch* routine in the *cuSPARSE* NVIDIA package is one of these examples, which is used as reference in this article. We propose a new implementation (*cuThomasBatch*) based on the Thomas algorithm. Unlike other algorithms, the Thomas algorithm is sequential, and so a coarse-grained approach is implemented where one CUDA thread solves a complete tridiagonal system instead of one CUDA block as in *gtsvStridedBatch*. To achieve a good scalability using this approach, it is necessary to carry out a transformation in the way that the inputs are stored in memory to exploit coalescence (contiguous threads access to contiguous memory locations). Different variants regarding the transformation of the data are explored in detail. We also explore some variants for the case of variable batch, when the size of the systems of the batch have different size (*cuThomasVBatch*). The results given in this study prove that the implementations carried out in this work are able to beat the reference code, being up to 5× (in double precision) and 6× (in single precision) faster using the latest NVIDIA GPU architecture, the Pascal P100.

Received . . .

KEY WORDS: Tridiagonal Linear Systems, Scalability, Thomas Algorithm, PCR, CR, Parallel Processing, cuSPARSE, CUDA.

1. INTRODUCTION

Recently, due to the huge parallel capacity of the current and upcoming NVIDIA GPUs, it is more and more popular the use of these accelerators to compute more than one problem (the so called batch) simultaneously [14, 2, 12, 9]. The solving of tridiagonal linear systems is required in many problems of industrial and scientific interest. Alternating direction implicit methods [5], spectral Poisson solvers [15, 16], cubic spline approximations, numerical ocean models [4], preconditioners for iterative linear solvers [3], the simulation of the human brain [12] are just a few number of examples where the solving of tridiagonal systems is necessary. Usually, a high number of tridiagonal systems must be solved in these applications, in which, in some cases, this process takes most of the total computation time.

*Correspondence to: pedro.valero@bsc.es.

The state-of-the-art method to deal with a tridiagonal system is the called Thomas algorithm [11]. However, previous works [17, 6, 15, 16, 7] have explored the use of other parallel algorithms to solve tridiagonal systems on GPUs. Although these algorithms are parallel, they need a higher number of operations with respect to the Thomas algorithm.

The use of parallel methods presents some additional drawbacks to be dealt with in GPU computing. For instance, it would be difficult to compute those systems that compromise a size bigger than the maximum number of threads per CUDA block (1024) or shared memory (64KB in P100 [8]), being in need of a significant amount of temporary extra storage [7], and forcing us to execute the problem by batches, when the requirements exceed these limitations. Also, parallel methods suffer from a poor numerical accuracy [7, 17]. Other problems are the computationally expensive operations such as atomic accesses and synchronizations necessary to compute these methods on NVIDIA GPUs. As reference code we have chosen *gtsvStridedBatch*, as this is the only routine (based on the CR-PCR algorithm [17, 7]) into the standard library cuSPARSE NVIDIA package [7], which solves multiple, a batch of, tridiagonal systems.

We propose a new implementation based on the Thomas algorithm avoiding high expensive computational operations, such as synchronizations and atomic accesses. Our codes, *cuThomasBatch* for fixed batch, the systems of the batch share the same size, and *cuThomasVBatch* for variable batch, the systems of the batch have different sizes, are able to compute a high number of tridiagonal systems of any size in one call (CUDA kernel), using one thread per system instead of one CUDA block per system, as in *gtsvStridedBatch*. However in order to achieve a good performance using the proposed approach, we need to modify the data layout used to store the set of inputs in global memory to efficiently exploit the memory hierarchy of the GPUs (coalescing accesses to GPU memory). We evaluate the scalability of both approaches, *gtsvStridedBatch* and *cuThomasBatch*, for computing multiple and independent tridiagonal systems on NVIDIA GPUs. The present work extends the previously published works [13] with additional contributions. This work includes a new approach for the *cuThomasBatch*, which makes use of a unified vector in order to exploit the memory hierarchy more efficiently. Also, we extend the previous work implementing a new variant called *cuThomasVBatch* to deal with variable batch, multiple independent tridiagonal systems with different sizes. This is particularly interesting for the simulation of the human brain [12].

The rest of the paper is structured as follows. Section 2 briefly introduces the problem at hand and the different methodologies to deal with it. In Section 3 we present the specific parallel features for the resolution of multiple tridiagonal systems. Section 4 shows the performance achieved and finally the conclusions are outlined in Section 5.

2. TRIDIAGONAL LINEAR SYSTEMS

The state-of-the-art method to solve tridiagonal systems is the called Thomas algorithm [15]. Thomas algorithm is a specialized application of the Gaussian elimination that takes into account the tridiagonal structure of the system. Thomas algorithm consists of two stages, commonly denoted as forward elimination and backward substitution.

Given a linear $Au = y$ system, where A is a tridiagonal matrix:

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

The forward stage eliminates the lower diagonal as follows:

$$c'_1 = \frac{c_1}{b_1}, \quad c'_i = \frac{c_i}{b_i - c'_{i-1} a_i} \quad \text{for } i = 2, 3, \ldots, n-1$$

DOI: 10.1002/cpe

$$y'_1=\frac{y_1}{b_1},\ y'_i=\frac{y_i-y'_{i-1}a_i}{b_i-c'_{i-1}a_i} \quad \text{for } i=2,3,\ldots,n-1$$

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

$$u_n=y'_n, u_i=y'_i-c'_iu_{i+1} \text{ for } \ i=n-1,n-2,\ldots,1$$

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

Cyclic Reduction (CR) [17, 6, 15, 16] is a parallel alternative to Thomas algorithm. It also consists of two phases (reduction and substitution). In each intermediate step of the reduction phase, all even-indexed (i) equations $a_ix_{i-1}+b_ix_i+c_ix_{i+1}=d_i$ are reduced. The values of a_i, b_i, c_i and d_i 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:

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

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

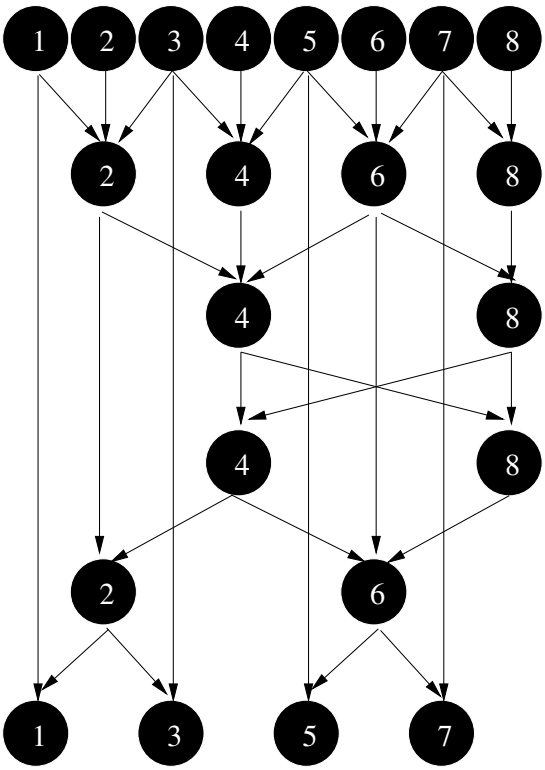

Figure 1. Access pattern of the CR algorithm.

Parallel Cyclic Reduction (PCR) [17, 6, 15, 16] 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 $k=2^{j-1}$:

DOI: 10.1002/cpe

$$a'_i = \alpha_i a_i, b'_i = b_i + \alpha_i c_{i-k} + \beta_i a_{i+k}$$
$$c'_i = \beta_i c_{i+1}, y'_i = b_i + \alpha_i y_{i-k} + \beta_i y_{i+k}$$
$$\alpha_i = \frac{-a_i}{b_{i-1}}, \beta_i = \frac{-c_i}{b_i}$$

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 2 sketches the corresponding access pattern.

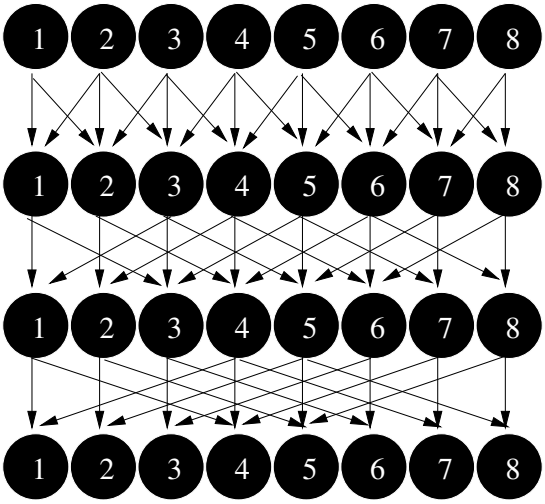

Figure 2. Access pattern of the PCR algorithm.

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 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 [17].

In fact, hybrid combinations that try to exploit the best of each algorithm have been explored [17, 10, 1, 6, 15, 16]. Figure 3 illustrates the access pattern of the CR-PCR combination proposed in [17]. CR-PCR reduces the system to a certain size using the forward reduction phase of CR and then it 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 [7], one of the implementations evaluated in this work.

There are more algorithms apart from the ones above mentioned to deal with tridiagonal systems, such as those based on Recursive Doubling [17], among others. However we have focused on those, which were proven to achieve a better performance and were implemented in the reference library [7].

3. IMPLEMENTATION OF CUTHOMASBATCH

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, first we focus on presenting the different data layouts proposed and analyze the

DOI: 10.1002/cpe

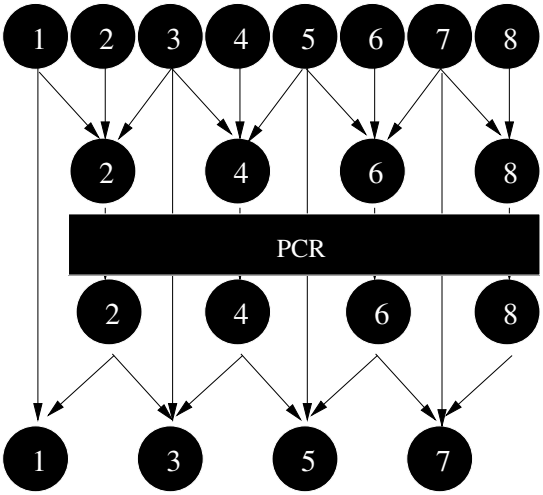

Figure 3. Access pattern of the CR-PCR algorithm.

impact of these on the overall performance. Two different data layouts were explored: *Flat* and *Full-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, initially, 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.

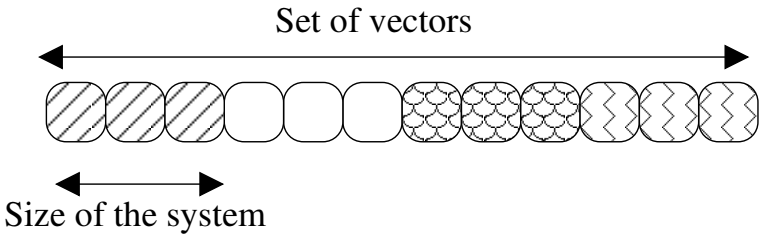

Figure 4. Example of the *Flat* data layout.

For sake of clarity, Figure 4 and 5 illustrate a simple example composed by four different tridiagonal systems of three elements each. Please, note that we only illustrate one vector per system in both Figures, but in the real scenario we would have 4 vectors per tridiagonal system 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 and CUDA thread mappings. As shown later, the differences found in the data layouts studied have important consequences on the scalability.

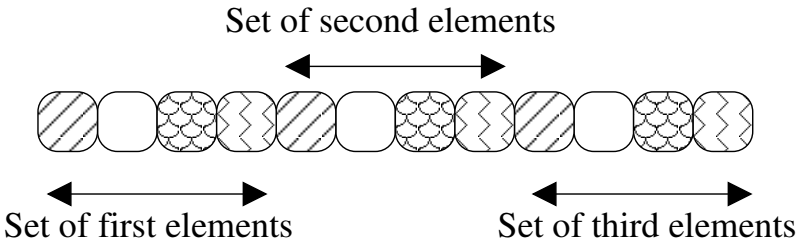

Figure 5. Example of the *Full-Interleaved* data layout.

DOI: 10.1002/cpe

Additionally, we have explored another data-layout, *Unified-Vector*. In this case, we attempt to analyze the hierarchy of memory by exploiting the temporal locality that exits among the different vectors (*a*, *b*, *c*, and *u/y* in Section 2). Basically, every thread, immediately after a_i is computed, has to compute also b_i and c_i, in the forward step. In the backward step, the process is similar, but in the opposite order. Using this data layout, we want to take advantage of this characteristic of the *Thomas* algorithm. For the sake of clarity, Figure 6 graphically illustrates the data layout proposed for a simple batch composed by only 2 independent tridiagonal systems.

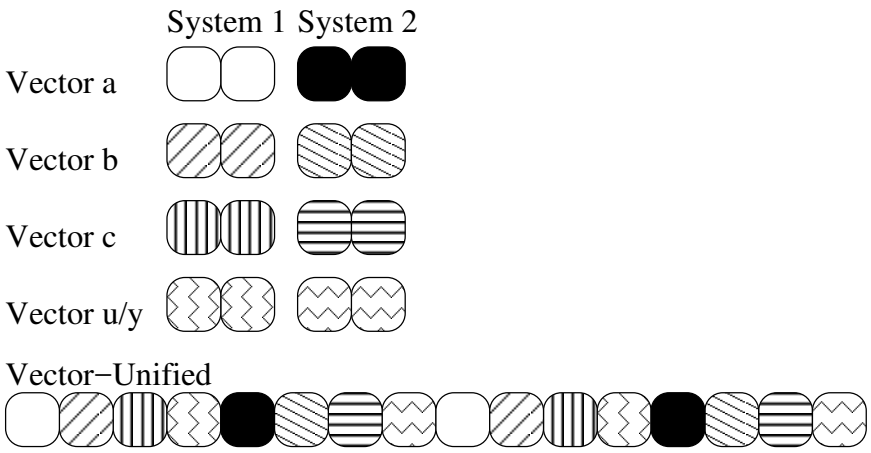

Figure 6. Example of the *Unified-Vector* data layout.

Next, we explore the different proposals about the CUDA thread mapping on the data layouts above described. Figure 7 illustrates the different CUDA thread mappings studied in this paper. Figure 7-top shows a coarse-grain scheme where a set of tridiagonal ($S_1, .., S_n$ in Figure 7) systems is mapped onto a CUDA block, so that each CUDA thread fully solves one 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. Using the *Flat* data layout we can not exploit coalescence when exploiting one thread per tridiagonal system (coarse approach) [15]; however by interleaving (Figure 5) the elements of the vectors (*a*, *b*, *c*, *u* and *y* in Section 2), contiguous threads access to contiguous memory locations. 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 (Section 2) on *Full-Interleaved* data layout (Figure 5).

On the other hand, previous studies have explored the use of the fine-grain scheme based on CR-PCR [17, 6, 15, 16] using the *Flat* data layout. In this case (Figure 7-bottom), 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 stored in the shared memory of the GPU). However, 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. Although 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). Besides, it is small (up to 64KB in the architecture used [8]) 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 [7]) is based on this approach, CR-PCR (Section 2) on *Flat* data layout (Figure 4).

DOI: 10.1002/cpe

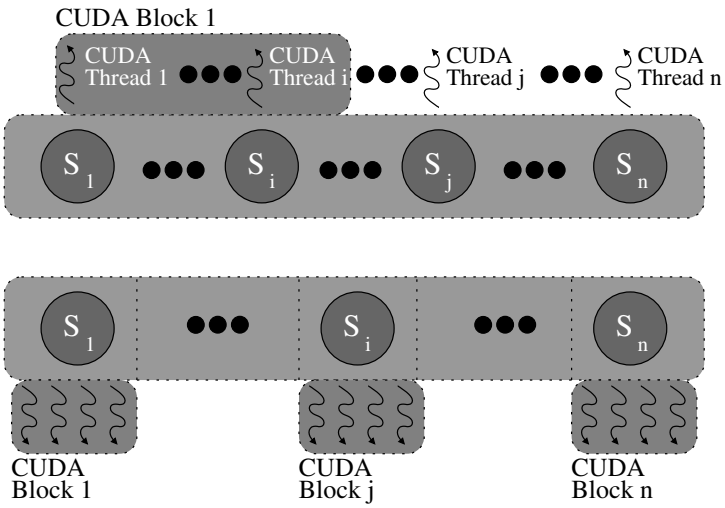

Figure 7. Coarse (top) and fine (bottom) CUDA thread mapping.

3.1. Variable Batch

In this sub-section, we describe the techniques used to deal with Variable Batch, a batch of tridiagonal systems with different sizes. Figure 8 graphically illustrates a simple example of a Variable Batch composed by three vectors of different sizes. To deal with Variable Batch, we make use of a widely used and extended technique very popular in parallel programming, the so called *padding*. This technique basically consists of filling with null elements those memory locations between two different elements of different size. This is particularly interesting and beneficial for GPU-based architectures, where the pattern of accesses to memory is so important to achieve coalescing and reduce the impact of the high latency. However, in our particular scenario, we have to adapt this technique to the data-layout used, *Full-Interleaved*. An example of this is illustrated by Figure 8.

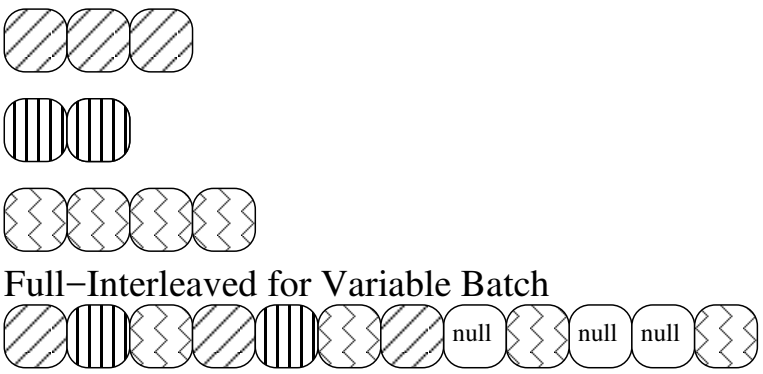

Figure 8. Example of the *Full-Interleaved* data layout for Variable Batch (Padding).

Regarding the CUDA thread mapping, we follow the approach based on using one CUDA thread per tridiagonal system (Figure 7-top). The reference code (*gtsvStridedBatch*) does not offer the possibility to compute batches of tridiagonal systems of different sizes. We have evaluated two different approaches, one called *Computing Padding* (CP) and one called *No Computing Padding* (NCP). While the *CP* approach computes the null elements located between different inputs, which does not affect the final result, the *NCP* only computes the no-null elements. This last can be shown as a more efficient approach, but it is in need of an extra parameter (vector) which stores the size of the systems and it suffers from divergence. Those threads that are in charge of computing small

DOI: 10.1002/cpe

systems, stop before others which have to compute large systems. This provokes not only divergence among different threads in the same CUDA block, but also no coalescing memory accesses, which can affect performance considerably.

4. PERFORMANCE ANALYSIS

To carry out the experiments, we have used one of the latest NVIDIA GPUs, the P100 Pascal composed of 3584 cores and 16GB HBM2 at 732 GB/s of global memory, and 2× IBM PowerNV 8335-GTB with 10 cores each. This node is a Linux (Red Hat 7.3) machine, on which we have used the next configuration (compiler versions and flags): gcc 4.8.5, cuda 8.0, -arch=sm_60 -fopenmp -O3 -std=c++11. The codes evaluated in this section are available in a publicly accessible repository [†].

4.1. Scalability

We have evaluated the performance of each of the approaches, *gtsvStridedBatch*, *cuThomasBatch* and *cuThomasBatch-Unified Vector* using both, single and double precision operations. Our test cases consist of computing 256, 2,560, 25,600 and 256,000 tridiagonal systems of 64, 128, 256 and 512 elements each. 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 test cases (64-512) can be fully executed by one CUDA block using *gtsvStridedBatch*. 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.

To analyze in detail the scalability of all the implementations, we graphically illustrate in Figures 9 and 10 the speedup for both, single and double precision operations, against the sequential counterpart including the performance achieved by the multicore execution (20 cores in 2 sockets IBM Power8 node, 10 cores each). The implementation based on multicore basically uses OpenMP pragma (*#pragma omp for*) on the 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 of about 90 in single precision and close to 80 in double precision, *cuThomasBatch* scales much more (Figure 9 and Figure 10), achieving a speedup peak of about 280 in single precision and about 180 for double precision operations. It is remarkable that the use of different precisions has a higher impact in *cuThomasbatch* than in *gtsvStridedBatch*. It is important to note that the multicore OpenMP implementation outperforms, in some cases (25,600 and 256,000 systems), the *gtsvStridedBatch* implementation when executing systems of small size (64). Regarding the *cuThomasBatch-Unified Vector*, this only outperforms the *cuThomasBatch* performance in a few cases (256 systems of size 64, for instance). The scalability of this approach is considerably worse, as it suffers from a fall in performance when a high number of systems is executed.

Figure 11 graphically illustrates the speedup achieved by our implementation (*cuThomasBatch*) against the *cuSPARSE* routine. There is no clear trend regarding the benefit achieved using *cuThomasBatch*, as it depends on many different factors, such as size of systems, number of the systems, granularity, CUDA Block size, among other. However, this implementation is always better than the *gtsvStridedBatch* routine, obtaining a speedup range from 1.25 to 6.1 in single precision and from 1.21 to close to 5 for double precision operations.

To evaluate both implementations, *gtsvStridedBatch* and *cuThomasBatch*, more deeply, we make use of the NVIDIA profiler (NVPROF) to obtain some metrics like bandwidth and efficiency (Warp Execution Efficiency in NVPROF). The peak bandwidth on P100 is 732 GB/s, however the ECC (Error Correcting Code), which is natively supported in the HBM2 memory, causes a fall in the bandwidth of about 15% [8]. So the real bandwidth in P100 is about 622 GB/s. To obtain these

[†]BSC-GitLab, https://pm.bsc.es/gitlab/run-math/cuThomasBatch-cuThomasVBatch

DOI: 10.1002/cpe

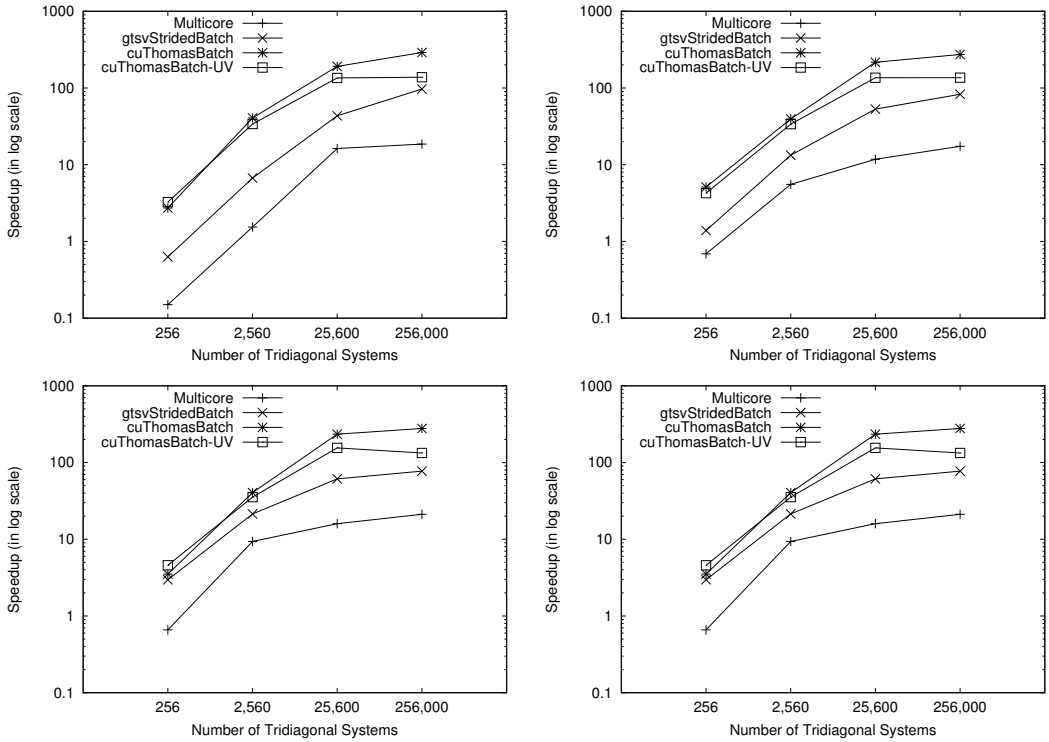

Figure 9. Speedup (execution time of each of the approaches divided by the execution time of the sequential CPU code) of *Multicore(Multi)*, *cuThomasBatch*, *gtsvStridedBatch* and *cuThomasBatch-UnifiedVector(UV)* using single precision operations for computing multiple, 256-256,000 tridiagonal systems of different sizes: 64 (left-top), 128 (right-top), 256 (left-bottom) and 512 (right-bottom).

metrics, we have used the biggest test case, 256,000 systems of 512 elements each using double precision. Unlike our implementation, the *gtsvStridedBatch* routine is composed by two kernels, *pcrGtsvBatchFirstPass* and *pcrGtsvBatchSharedMemKernelLoop*. While the first is more focused on global memory operations, the second concentrates more operations on shared memory. As commented, the algorithm used in *gtsvStridedBatch* (see Section 2) can exploit efficiently the shared memory. The bandwidth achieved by *pcrGtsvBatchFirstPass* is about 503 GB/s (about 80% of the real bandwidth). In *pcrGtsvBatchSharedMemKernelLoop* the bandwidth is much smaller, about 192 GB/s, but this kernel focuses on shared memory operations, achieving a bandwidth on this memory of about 5,670 GB/s. In *cuThomasBatch* the bandwidth achieved is 527 GB/s, we do not make use of shared memory (Section 2). The kernels of the *cuSPARSE* routine, *pcrGtsvBatchFirstPass* and *pcrGtsvBatchSharedMemKernelLoop*, are able to achieve an efficiency of 99.9% and 94.1% respectively. In our implementation the efficiency is 100%.

It is also 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 [12].

4.2. Numerical Accuracy

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. 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. The error (accuracy

DOI: 10.1002/cpe

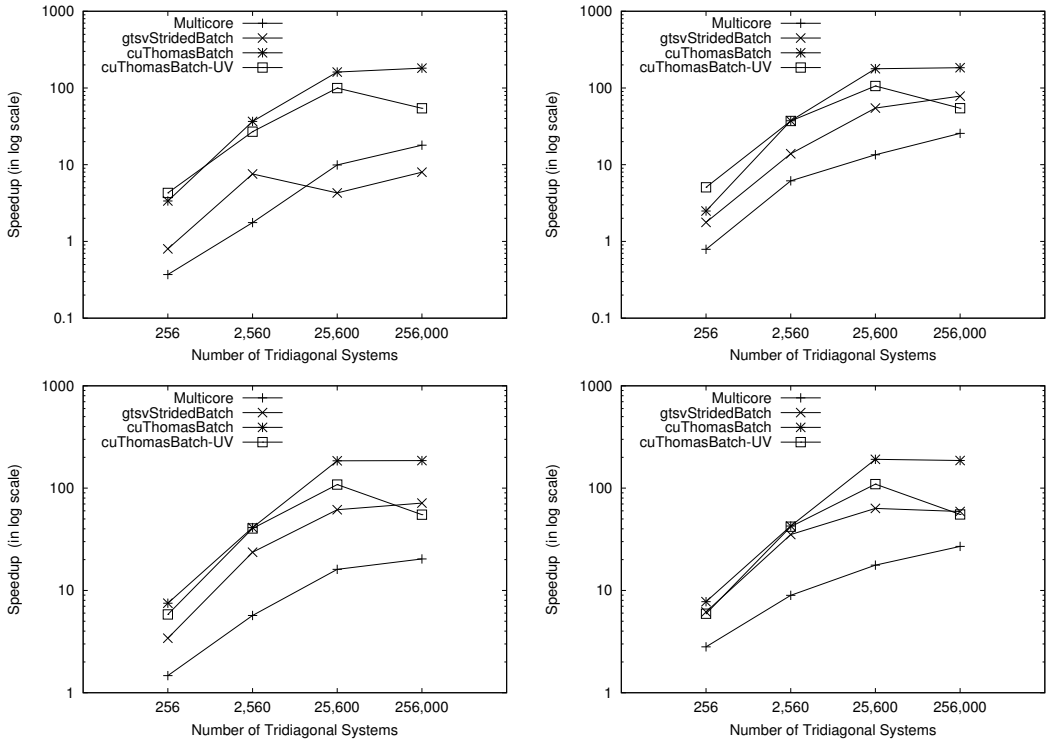

Figure 10. Speedup (execution time of each of the approaches divided by the execution time of the sequential CPU code) of *Multicore(Multi)*, *cuThomasBatch*, *gtsvStridedBatch* and *cuThomasBatch-UnifiedVector(UV)* using double precision operations for computing multiple, 256-256,000 tridiagonal systems of different sizes: 64 (left-top), 128 (right-top), 256 (left-bottom) and 512 (right-bottom).

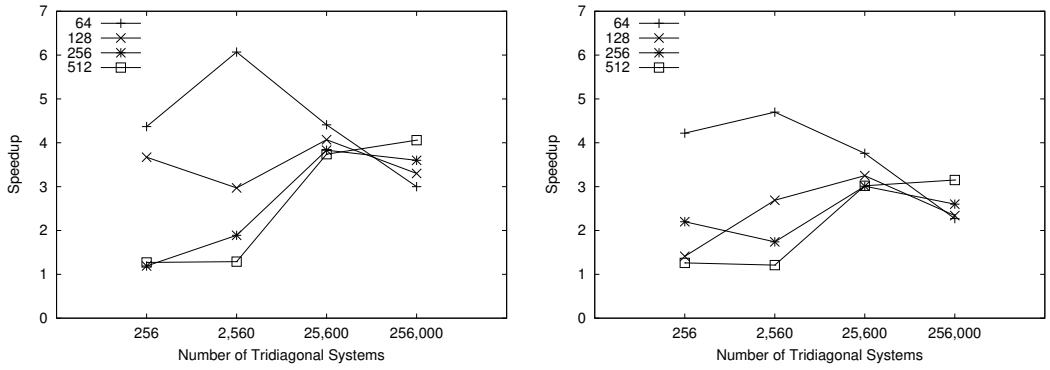

Figure 11. Speedup (execution time of the *cuThomasBatch* with respect to the execution time of the *gtsvStridedBatch*) using both, single precision (left) and double precision operations (right), for computing multiple, 256-256,000 tridiagonal systems using different sizes: 64, 128, 256 and 512.

with respect to the result obtained by the sequential CPU code) in *cuThomasBatch* is zero. This is because of the intrinsic characteristics of the Thomas algorithm. On the other hand, the error using *gtsvStridedBatch* is between 7×10^{-8} and 9×10^{-8} for single precision and between 1×10^{-16} and 2×10^{-16} for double precision on the tests evaluated. One additional important characteristic of our implementation is that the results, in terms of numerical accuracy, are reproducible. This is an important characteristic for multiple applications.

DOI: 10.1002/cpe

4.3. Memory Occupancy

As commented in Section 1, the use of parallel methods requires an additional amount of temporary extra storage [7]. In particular *gtsvStridedBatch* is in need of $m \times (4 \times n + 2048) \times sizeof(< type >)$ more memory than the *cuThomasBatch*, being m and n the number of systems and the size of the systems respectively [7]. This implies, for instance, that *gtsvStridedBatch* requires about $2\times$ more memory capacity than *cuThomasBatch* to compute 256,000 tridiagonal systems of 64 elements each or about 1GB extra memory to compute 256,000 tridiagonal systems of 512 elements each (Figure 12).

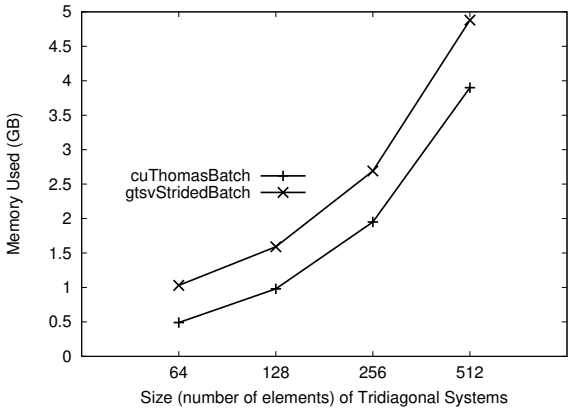

Figure 12. Memory used by *gtsvStridedBatch* and *cuThomasBatch* to compute 256,000 tridiagonal systems using double precision.

4.4. Variable Batch, cuThomasVBatch

Here we evaluate the variant of *cuThomasBatch* for variable batch (batch of tridiagonal systems with different sizes), *cuThomasVBatch*. To evaluate both variants proposed, *No Computing Padding* (NCP) and *Computing Padding* (CP), 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-coalescence memory accesses, causing and important underutilization of the computational capacity of our GPU architecture. We also make use of NVPROF to achieve some metrics like memory bandwidth and efficiency. While the bandwidth achieved by the *NCP* variant is about 205 GB/s, when executing 256,000 tridiagonal systems of 256-512 elements each, the *CP* is able to achieve a bandwidth of about 525 GB/s (about 85% of the peak bandwidth) for the same test case. The efficiency is bigger using the *CP* approach (77%) than the *NCP* one (57%). 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 equivalent to the execution time of a fixed-size batch (*cuThomasBatch*) of 512. As commented in Section 3, 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 the *cuThomasBatch* for the maximum size of system computed by *CP*.

5. FINAL REMARKS AND FUTURE WORK

Our implementation is able to outperform the *cuSPARSE* implementation, even when a low number of systems is computed. This is because of a simpler management of CUDA threads, as we do

DOI: 10.1002/cpe

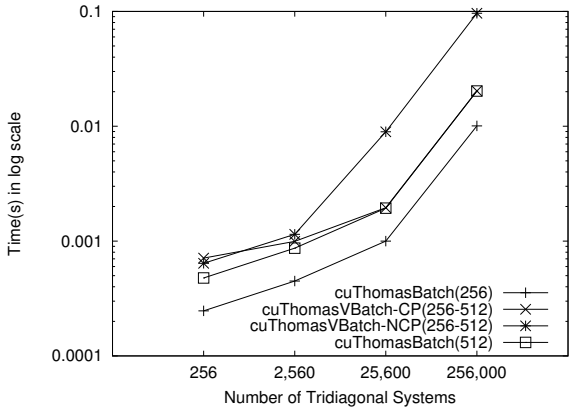

Figure 13. Execution time (using double precision) for the two variants, *No Computing Padding* and *Computing Padding*, of the *cuThomasVBatch*.

not have to deal with synchronizations, atomic operations and the limitations regarding the size of shared memory and CUDA blocks. In this paper, the authors have extended previous work by exploring new variants, such as the *Unified-Vector* approach, using new GPU architectures (P100) and implementing the variable batch version (*cuThomasVBatch*). One important consequence of this work is the inclusion of the implementation of the *cuThomasBatch* into in the next release of the NVIDIA's reference library for sparse linear algebra operations (*cuSPARSE*) with the name of *gtsvInterleavedBatch*.

As future work, we plan to use the implementations above described, in particular the *cuThomasVBatch*, to solve and accelerate one of the major steps in the simulation of the behavior of the human brain, into the European Flagship Project, Human Brain Project. We also plan to apply the same ideas explored and evaluated in this work to other linear algebra operations.

6. ACKNOWLEDGMENTS

This project was funded 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 and the valuable feedback provided by Lung Sheng Chien and Alex Fit-Florea. Antonio J. Peña is cofinanced by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva fellowship number IJCI-2015-23266.

REFERENCES

1. Andrew Davidson, Yao Zhang, and John D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *IEEE International Parallel and Distributed Processing Symposium*, May 2011.
2. Jack J. Dongarra, Sven Hammarling, Nicholas J. Higham, Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. The design and performance of batched BLAS on modern high-performance computing systems. In *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, pages 495–504, 2017.
3. A. Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, 1997.
4. George R Halliwell. Evaluation of vertical coordinate and vertical mixing algorithms in the hybrid-coordinate ocean model (hycom). *Ocean Modelling*, 7(34):285 – 322, 2004.
5. Ching-Tien Ho and S. Lennart Johnsson. Optimizing tridiagonal solvers for alternating direction methods on boolean cube multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 11(3):563–592, 1990.

DOI: 10.1002/cpe

6. Hee-Seok Kim, Shengzhao Wu, Li wen Chang, and Wen mei W. Hwu. A scalable tridiagonal solver for GPUs. *2013 42nd International Conference on Parallel Processing*, 0:444–453, 2011.
7. NVIDIA. cuSPARSE. *CUDA Toolkit Documentation*.
8. NVIDIA. The most advanced datacenter accelerator ever built featuring pascal gp100, the world's fastest gpu. In *White paper: NVIDIA Tesla P100*, pages 1–45, 2017.
9. Mohammad Sajid, Zahid Raza, and Mohammad Shahid. Energy-efficient scheduling algorithms for batch-of-tasks (bot) applications on heterogeneous computing systems. *Concurrency and Computation: Practice and Experience*, 28(9):2644–2669, 2016.
10. N. Sakharnykh. Efficient tridiagonal solvers for adi methods and fluid simulation. In *NVIDIA GPU Technology Conference*, September 2010.
11. C. de Boor S.D. Conte. *Elementary Numerical Analysis*, volume 1. McGraw-Hill, 1976.
12. Pedro Valero-Lara, Ivan Martínez-Perez, Antonio J. Peña, Xavier Martorell, Raúl Sirvent, and Jesús Labarta. cuhinesbatch: Solving multiple hines systems on gpus human brain project*. In *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, pages 566–575, 2017.
13. Pedro Valero-Lara, Ivan Martínez-Perez, Antonio J. Peña, Raúl Sirvent, and Xavier Martorell. Nvidia gpus scalability to solve multiple (batch) tridiagonal systems. implementation of cuthomasbatch. In *Parallel Processing and Applied Mathematics - 12th International Conference, PPAM 2017*, 2017.
14. 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.
15. Pedro Valero-Lara, Alfredo Pinelli, Julien Favier, and Manuel Prieto Matias. Block tridiagonal solvers on heterogeneous architectures. In *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 609–616, 2012.
16. Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matias. 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. *SIGPLAN Not.*, 45(5):127–136, January 2010.

DOI: 10.1002/cpe