Selecting the Best Tridiagonal System Solver Projected on Multi-Core CPU and GPU Platforms

Pablo Quesada-Barriuso[1], **Julián Lamas-Rodríguez**[1], **Dora B. Heras**[1], **Montserrat Bóo**[1], **Francisco Argüello**[1]
[1]Centro de Investigación en Tecnoloxías da Información (CITIUS), Univ. of Santiago de Compostela, Spain

Abstract— *Nowadays multicore processors and graphics cards are commodity hardware that can be found in personal computers. Both CPU and GPU are capable of performing high-end computations. In this paper we present and compare parallel implementations of two tridiagonal system solvers. We analyze the cyclic reduction method, as an example of fine-grained parallelism, and Bondeli's algorithm, as a coarse-grained example of parallelism. Both algorithms are implemented for GPU architectures using CUDA and multi-core CPU with shared memory architectures using OpenMP. The results are compared in terms of execution time, speedup, and GFLOPS. For a large system of equations,* 2^{22}*, the best results were obtained for Bondeli's algorithm (speedup* 1.55*x and* 0.84 *GFLOPS) for multi-core CPU platforms while the cyclic reduction (speedup* 17.06*x and* 5.09 *GFLOPS) was the best for the case of GPU platforms.*

Keywords: Tridiagonal system solver, multi-core CPU, OpenMP, GPU, CUDA

1. Introduction

The tridiagonal system solvers are widely used in the field of scientific computation, especially in physical simulations. The tridiagonal matrix algorithm, also know as the Thomas Algorithm [1], is one of the most known algorithms used to solve tridiagonal systems. It is based on the Gaussian elimination [2] to solve dominant diagonal systems of equations. As it is not suitable for parallel implementations, new algorithms such as cyclic reduction [3], [4] or recursive doubling [5] were developed to exploit fine-grained parallelism. Other parallel tridiagonal solvers like [6], [7], [8] were designed with a coarse-grained parallelism in mind.

OpenMP is designed for shared memory architectures [9]. The development in recent years of cluster-like architectures favoured the Multiple Instructions Multiple Data programming languages, particularly MPI. However, recent multi-core architectures have increased interest in OpenMP which has led us to choose this API for our CPU implementations.

The Computed Unified Device Architecture [10] developed by NVIDIA provides support for general-purpose computing on graphics hardware [11] with a fine-grained and coarse-grained data parallelism. However, in most cases, neither sequential nor parallel algorithms can be implemented directly into the GPU. The algorithm's design needs to be adapted in order to exploit each architecture.

In this paper we present and compare a parallel implementation of cyclic reduction as described in [12] and one of Bondeli's algorithm [8]. Both algorithms are valid for large systems of equations. They have been implemented for multi-core CPUs using OpenMP and for GPU computing using CUDA. The data access patterns and workflows of the algorithms are divided into several stages, making them good candidates for a parallel implementation using OpenMP, but a challenge for a GPU version owing to the specific features of this architecture. We shall analyze the performance of the cyclic reduction given that it was one of the first parallel algorithms described for solving tridiagonal systems. The cyclic reduction algorithm was focused towards fine-grained parallelism which could be achieved into the GPU architecture. And we shall analyze Bondeli's algorithm as an example of coarse-grained parallelism in a divide-and-conquer fashion, more suited to a multi-core CPU architecture.

In Section 2 we briefly present the proposed algorithms. An overview of the OpenMP programming model and the implementation of the algorithms are presented in Section 3. In Section 4 we present the GPU Architecture and CUDA programming model with the implementations of the algorithms. The results obtained for each proposal are discussed in Section 5. Finally, the conclusions are presented in Section 6.

2. Tridiagonal System Algorithms

The objective is solving a system of n linear equations $A\vec{x} = \vec{d}$, where A is a tridiagonal matrix, i.e.:

$$\begin{bmatrix} b_1 & c_1 & & 0 \\ a_2 & b_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ 0 & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}. \tag{1}$$

We shall denote each equation of this system as:

$$E_i \equiv a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \tag{2}$$

for $i = 1, \dots, n$ where $x_0 = x_{n+1} = 0$.

In the following subsections, the tridiagonal system solvers of cyclic reduction and Bondeli's algorithm are presented.

2.1 Cyclic reduction

The cyclic reduction algorithm [12] starts with a *forward reduction* stage where the system is reduced until a unique

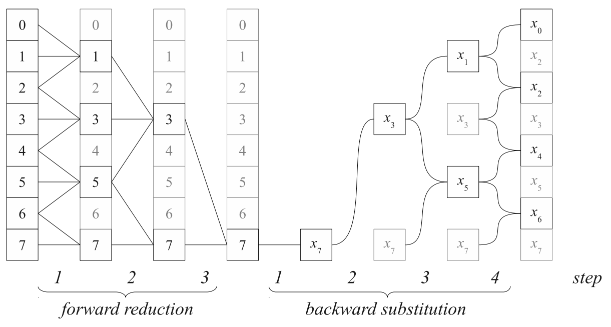

Fig. 1: Dependencies in cyclic reduction for an 8-equation system.

unknown is obtained. At each step s of this stage the new system is reduced to half the number of unknowns and new equations are generated:

$$E_i^{(s)} = \alpha_i^{(s-1)} E_{i-2^{s-1}}^{(s-1)} + E_i^{(s-1)} + \beta_i^{(s-1)} E_{i+2^{s-1}}^{(s-1)}, \qquad (3)$$

with $i = 2^s, \ldots, n$, a step size 2^s, $\alpha_i^{(s-1)} = -a_i^{(s-1)}/b_{i-2^{s-1}}^{(s-1)}$ and $\beta_i^{(s-1)} = -c_i^{(s-1)}/b_{i+2^{s-1}}^{(s-1)}$. The forward reduction stage needs $\lfloor \log_2 n \rfloor$ steps to complete.

The second stage, *backward substitution*, computes first the unique unknown x_{n-1} of the equation $E_{n-1}^{(s)}$, with $s = \lfloor \log_2 n \rfloor$, generated in the last step of the forward reduction. The values of the next unknowns are computed using the previously solved ones. At each step $s = \lfloor \log_2 n \rfloor - 1, \ldots, 0$ of the backward substitution, the Equation (3) is reformulated as:

$$x_i = \frac{d_i^{(s)} - a_i^{(s)} x_{i-2^s} - c_i^{(s)} x_{i+2^s}}{b_i^{(s)}}. \qquad (4)$$

for $i = 2^s, \ldots, n$, with step size 2^{s+1}.

Figure 1 shows a scheme of the resolution of the cyclic reduction algorithm for an 8-equation system. In the first stage, each square represents an equation $E_i^{(s)}$ in the tridiagonal system. The operation described by expression (3) is represented with lines in the figure. In the second stage, a square represents one of the unknowns x_i whose value is calculated at each step, and dependencies in expression (4) are represented with lines.

2.2 Bondeli's algorithm

The Bondeli's method [8] is based on a divide-and-conquer strategy to solve a tridiagonal system of equations. The system is initially divided into blocks of size $k \times k$:

$$\begin{bmatrix} B_1 & C_1 & 0 & \ldots & 0 & 0 \\ A_2 & B_2 & C_2 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & A_p & B_n \end{bmatrix} \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_n \end{bmatrix} = \begin{bmatrix} \vec{d}_1 \\ \vec{d}_2 \\ \vdots \\ \vec{d}_n \end{bmatrix}. \qquad (5)$$

where each term A_i, B_i, and C_i represents a matrix block of size $k = n/p$, being $p > 1$ an arbitrary positive integer. Each vector $\vec{x}_i$ and $\vec{d}_i$ has also k elements. It is assumed that n is divisible by p and $k > 1$.

This method has three different steps: solving each block of the linear tridiagonal systems, building and solving an intermediate tridiagonal system, and computing the unknowns using the results of the first and second steps. Each step is describe below:

1) Solving the following $3p - 2$ tridiagonal systems:

$$\begin{aligned} &B_i \vec{y}_i = \vec{d}_i && i = 1, \ldots, p, && (6a) \\ &B_1 \vec{z}_1 = \vec{e}_k && i = 1, && (6b) \\ &B_p \vec{z}_{2p-2} = \vec{e}_1 && i = p, && (6c) \\ &B_i \vec{z}_{2i-2} = \vec{e}_1 && i = 2, \ldots, p-1, && (6d) \\ &B_i \vec{z}_{2i-1} = \vec{e}_k && i = 2, \ldots, p-1. && (6e) \end{aligned}$$

where $\vec{y}_i$ (for $i = 1, \ldots, p$) and $\vec{z}_j$ (for $j = 1, \ldots, 2p-2$) are vectors of unknowns of size k, and $\vec{e}_1^{\,T} = (1, 0, \ldots, 0)$ and $\vec{e}_k^{\,T} = (0, 0, \ldots, 1)$ are two vectors of size k. During the first step, the original system is divided into blocks and the generated subsystems are solved to obtain the values of $\vec{y}_i$ and $\vec{z}_i$.

2) From the results obtained in the previous step, a new tridiagonal system $\mathscr{H}\vec{\alpha} = \vec{\beta}$ of size $2p - 2$ is built and solved (variables are the same than defined in [8]):

$$\begin{bmatrix} s_1 & t_1 & & \\ r_2 & s_2 & \ddots & \\ & \ddots & \ddots & t_{2p-3} \\ & & r_{2p-2} & s_{2p-2} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{2p-2} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{2p-2} \end{bmatrix}. \qquad (7)$$

3) Lastly, in the third step, the values of the unknowns $\vec{x}_i$ in the original system are computed from the solution $\vec{\alpha}$ of the intermediate system and the values of $\vec{z}_j$ obtained in the first step. The values of $\vec{x}_i$ are calculated as:

$$\vec{x}_i = \begin{cases} \vec{y}_1 + \alpha_1 \vec{z}_1 & i = 1 \\ \vec{y}_i + \alpha_{2i-2} \vec{z}_{2i-2} + \alpha_{2i-1} \vec{z}_{2i-1} & i < p \\ \vec{y}_p + \alpha_{2p-2} \vec{z}_{2p-2} & i = p. \end{cases} \qquad (8)$$

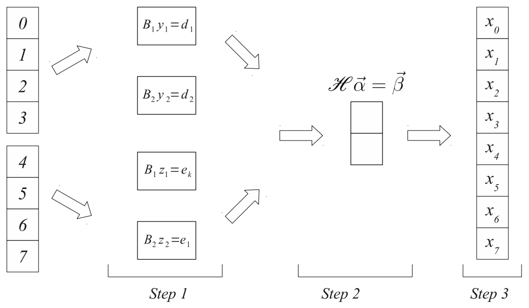

Fig. 2: Steps in Bondeli's algorithm for an 8-equation system divided into 2 blocks.

Figure 2 shows each step of Bondeli's algorithm for an 8-equation system divided into 2 blocks of size 4. Each square represents an Equation (6) and (7) in steps 1 and 2 respectively, and the computation of an unknown in step 3. In this example $3p - 2 = 4$ subsystems are solved in the first step and an intermediate tridiagonal system of size $2p - 2 = 2$ is built and solved in the second step. Note that Equations (6d) and (6e) are not generated in this example due to the small number of blocks of the first step.

3. Tridiagonal System Solvers Implementations with OpenMP

In this section we present an overview of the OpenMP programming model and the parallel implementations of the two algorithms introduced in Section 2.

3.1 OpenMP Overview

OpenMP is the standard Application Program Interface (API) for parallel programming on shared memory architectures [9]. Communication and coordination between threads is expressed through read/write instructions of shared variables and other synchronization mechanisms. It comprises compiler directives, library routines and environment variables and is based on a fork-join model (see Figure 3) where a master thread creates a team of threads that work together on single program multiple data (SPMD) constructs.

In shared memory architectures OpenMP threads access the same global memory where data can be *shared* among them or exclusive (*private*) for each one. From a programming perspective, data transfer for each thread is transparent and synchronization is mostly implicit. When a thread enters a parallel region it becomes the master, creates a thread team and forks the execution of the code among the threads and itself. At the end of the parallel region the threads join and the master resumes the execution of the sequential code.

Different types of worksharing constructs can be used in order to share the work of a parallel region among the threads [13]. The *loop construct* distributes the iterations of one or more nested loops into *chunks*, among the threads in the team. By default, there is an implicit barrier at the end of a loop construct. The way the iterations are split depends on the schedule used in the loop construct [13]. On the other hand, the *single construct* assigns the work on only one of the threads in the team. The remaining threads wait until the end of the single construct owing to an implicit barrier. This type of construct is also known as non-iterative worksharing construct. Other types of worksharing constructs are available.

Although there are implicit communications between threads through access to shared variables or the implicit synchronization at the end of parallel regions and worksharing constructs, explicit synchronization mechanisms for mutual exclusion are also available in OpenMP. These are critical or atomic directives, lock routines, and event synchronization directives.

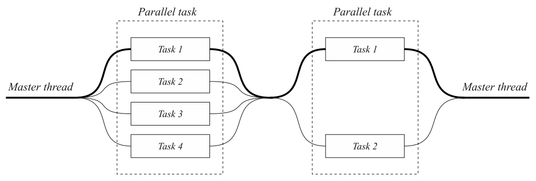

Fig. 3: OpenMP fork-join model.

OpenMP is not responsible for the management of the memory hierarchy but certain issues regarding cache memory management should be borne in mind. There are two factors that determine whether a loop schedule is efficient: data locality and workload balancing among iterations. The best schedule that we can choose when there are data locality and a good workload balance is static with a chunk size of $q = n/p$, where n is the number of iterations and p the number of threads. In other cases dynamic or guided schedules may be adequate.

When a cache line, shared among different processors, is invalidated as a consequence of different processors writing in different locations of the line, false sharing occurs. False sharing must be avoided as it decreases performance due to cache trashing. One way to avoid false sharing is to divide the data that will be accessed by different processors into pieces of size multiple of the cache line size. A good practice to improve cache performance is to choose a schedule with a chunk size that minimizes the requests of new chunks and that is multiple of a cache line size.

3.2 OpenMP implementations

In this section we present our OpenMP proposals, for the implementation of the tridiagonal system solvers: cyclic reduction and Bondeli's algorithm.

3.2.1 Cyclic reduction

For the two stages of the cyclic reduction algorithm we distribute $q = n/p$ equations among the threads, where n is the

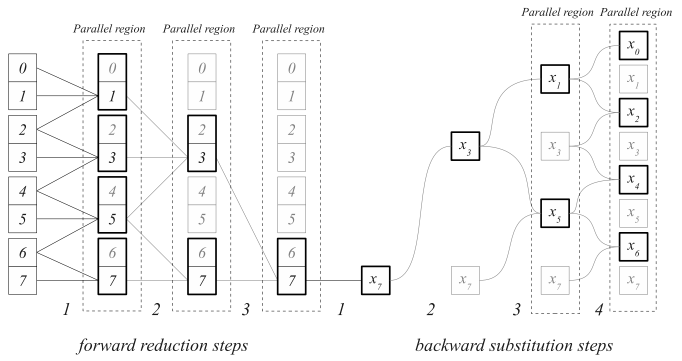

Fig. 4: Work shared among the threads in the forward reduction and backward substitution stage of cyclic reduction for an 8-equation system.

number of equations and p the number of threads. The storage system of this implementation employs five arrays: three matrix diagonals, right-hand side, and unknowns. All arrays are shared among the threads to satisfy the data dependencies.

Figure 4 (*left*) shows the access pattern for each thread in the forward reduction stage for the case of an 8-equation system. Each thread reduces q consecutive equations (2 equations in Figure 4) except for the last $\log_2 p$ steps, where p is the number of threads, as there are more threads than there are equations to reduce. An implicit barrier at the end of each step keeps the reduction stage synchronized. Figure 4 (*right*) shows the access pattern for each thread in the backward substitution stage, where the work is shared among the threads in the same way as in the reduction stage.

3.2.2 Bondeli's algorithm

For this implementation the different steps that form Bondeli's algorithm, described in Section 2.2, are executed in the same parallel region saving the time needed to create new thread teams when entering a new parallel region. We use implicit barriers to synchronize the different steps. The storage system of these implementations consists of five arrays that store the three diagonals, the right-hand coefficients and the unknowns of the tridiagonal system. The values of vectors $\vec{y}_i$ and $\vec{z}_j$ computed in the first step are stored in two separate arrays, and the intermediate system $\mathscr{H}\vec{\alpha} = \vec{\beta}$ described in the second step is stored in a similar way to the original system; i.e. in five arrays. All arrays are shared among the threads in order to read and write the data needed in each step.

The tridiagonal system is divided into blocks of size $k \times k$ where $k = n/p$, where n is the number of equations and $3p-2$ the number of threads. The resulting subsystems are solved in the first step of the algorithm. This is the most costly step where

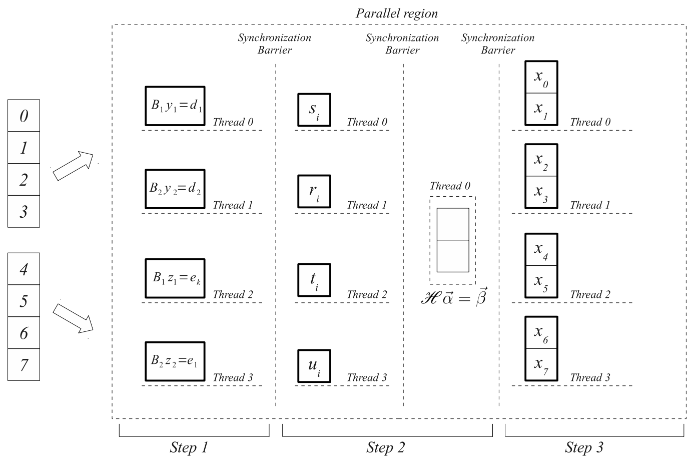

Fig. 5: Work shared among the threads in the different steps of the implementation of Bondeli's algorithm for an 8-equation system.

the subsystems are solved using the Thomas algorithm [1].

Figure 5 shows the worksharing constructs in the different steps of Bondeli's algorithm for an 8-equation system divided into 2 blocks of size 4. The work shared among the threads in this implementation for each step of the algorithm is as follows:

1) First step.- *Solve the linear tridiagonal systems of size* $k = n/p$. Each thread requests one subsystem of equations $B_i\vec{y}_i = \vec{d}_i$, one $B_1\vec{z}_1 = \vec{e}_k$ or $B_p\vec{z}_{2p-2} = \vec{e}_1$, Equations (6a)-(6c), Section 2.2. In our implementation the original tridiagonal system is split into 2 blocks, so Equations (6d)-(6e) do not need to be solved. A barrier at the end of the last directive synchronizes all the threads before the next step.
2) Second step.- *Compute* α_i *by solving the tridiagonal system* $\mathscr{H}\vec{\alpha} = \vec{\beta}$. The elements s_i, r_i, t_i of $\mathscr{H}$ and u_i of $\vec{\beta}$ (see [8] for details) are calculated in parallel by the different threads. The system $\mathscr{H}\vec{\alpha} = \vec{\beta}$ of $2p-2$ equations (2 equations in Figure 5) is solved by one thread. The thread team is synchronized before the next step.
3) Third step.- *Compute the solution with two SAXPY-operations*. The final solution is computed sharing $q = n/t$ equations among the threads, where n represents the number of equations and t the number of threads.

4. Tridiagonal System Solver Implementations with CUDA

In this section we present an overview of the GPU architecture and the CUDA implementations of the proposed algorithms introduced in Section 2.

4.1 GPU Overview

CUDA technology is the Compute Unified Device Architecture for NVIDIA programmable Graphic Processor Units [10]. This architecture is organized into a set of streaming multiprocessors (SMs) each one with many-cores or streaming processors (SPs). These cores can manage hundreds of threads in a Simple Program Multiple Data (SPMD) programming model. The number of cores per SM depends on the device architecture [10], i.e. the NVIDIA's G80 series has a total of 128 cores in 16 SMs each one with 8 SPs. Figure 6 shows a schema of a CUDA-capable GPU.

A CUDA program, which is called a kernel, is executed by thousands of threads grouped into blocks. The blocks are arranged into a grid and scheduled in any of the available cores enabling automatic scalability for future architectures. If there are not enough processing units for the blocks assigned to a SM, the code is executed sequentially. The smallest number of threads that are scheduled is known as a warp. All the threads within a warp execute the same instruction at the same time. The size of a warp is implementation defined and it is related to shared memory organization, data access patterns and data flow control [10], [14].

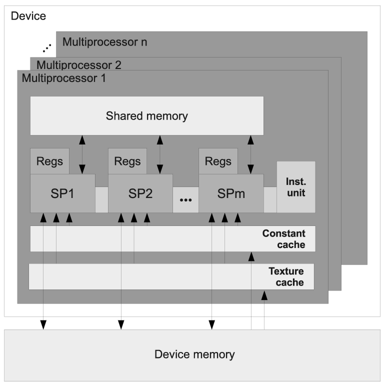

Fig. 6: A schema of streaming multiprocessors in a CUDA-capable GPU.

The memory hierarchy is organized into a *global memory* and a read-only *constant and texture memories*, with special features such as caching or prefetching data. These memories are available for all the threads. There is an on-chip *shared memory* space available per block enabling extremely fast read/write access to data but with the lifetime of the block. Finally, each thread has its own local and private memory.

There are mechanisms to synchronize threads within a block but not among different blocks. Due to this restriction data cannot be shared among blocks. This becomes a challenge when a thread needs data which have been generated outside its block.

4.2 CUDA implementations

In this section we present our proposals, using CUDA, for the implementation of the tridiagonal solvers under analysis.

The major challenge in pursuing an efficient implementation of a tridiagonal system solver in the GPU is the distribution of the system among the thread blocks. For small systems, the solution of the entire system can be computed using only shared memory and the register space without writing the intermediate results back to global memory. This can result in substantial performance improvements, as can be observed in [15], [16].

In the case of very large tridiagonal systems, the equations must be distributed among thread blocks, and a mechanism must be implemented to combine the partial operations performed by these individual blocks and to obtain the global solution. In standard FFT approaches, which present similar access patterns, the transformation of a large sequence can be computed by combining FFTs of subsequences that are small enough to be handled in shared memory. Data are usually arranged into a bidimensional array, and the FFTs of the subsequences are computed along rows and columns [17], [18]. This technique cannot be applied directly to tridiagonal systems solvers, since the factorization of a tridiagonal system into subproblems to be independently computed is not as regular as the FFT case.

4.2.1 Cyclic reduction

When the cyclic reduction method is implemented in global memory, the forward reduction stage can be implemented through a loop executed by the host, which calls a CUDA kernel at each step. The forward reduction kernel call invokes one thread for each equation E_i of the system that is modified in the current step and executes the operations stated in Expression (3). In this stage, the number of invoked threads drops by a factor of two at each new step. Once the forward reduction is completed, the backward substitution begins, and a second kernel is called, where an unknown x_i is assigned to each thread which computes the value thereof by applying Expression (4). At each of the second kernel calls the number of threads increases by a factor of two.

In our GPU implementation of the cyclic reduction, we used the shared memory space as much as possible [19]. The forward reduction stage is done in two kernels. A first kernel splits the equations into several non overlapping blocks. Each thread within a block copies an equation from global memory into the shared memory of its block. This kernel then solves as many steps of the reduction stage as possible with the data stored in the shared memory. Finally the intermediate results are copied back to global memory. The second kernel computes the equations that can not be solved due to data dependencies with the adjacent blocks. These equations are computed directly into global memory what is more efficient in this case due to the small number of equations remaining. Both kernels are successively executed until the forward reduction stage is completed. The backward substitution stage can be computed in a similar way.

4.2.2 Bondeli's algorithm

In this implementation, the different stages of Bondeli's algorithm, described in Section 2.2, are executed in several kernels because of the global synchronization needed in each stage [19]. The original tridiagonal system is split in small subsystems which can be solved independently. All subsystems are solved in just one kernel call, and the solution is nearly entirely performed in the shared memory space. This is possible due to the small size of the subsystems, which can now be assigned to different blocks of threads and be solved independently in parallel using the cyclic reduction algorithm.

5. Results

We have evaluated our proposals on a PC with an Intel Core 2 Quad Q9450 with four cores at 2.66 GHz and 4 GB of RAM. The main characteristics of the memory hierarchy are described in [20]. Each core has a L1 cache divided into 32 KB for instructions and 32 KB for data. With respect to L2, it is an unified cache of 6 MB shared by 2 cores (12 MB in total). Cache lines for the L1 and L2 caches are 64 bytes wide. The code has been compiled using gcc version 4.4.1 with OpenMP 3.0 support under Linux. For the CUDA implementations we ran the algorithms on a NVIDIA GeForce GTX 295. The CUDA code has been compiled using nvcc also under Linux. In both cases the code was compiled without optimizations.

The results are expressed in terms of execution times, speedups and GFLOPS. The execution times were obtained as an average of one hundred executions. The speedup are calculated for both OpenMP and CUDA implementations with respect to the sequential implementation of the Thomas algorithm. The execution times for the OpenMP implementations are those corresponding to 4 threads. These times include the creation of the thread team which is OpenMP implementation defined. As the final goal is to execute the tridiagonal system solvers within a larger computational algorithm, the data transfer between the host and the device in the CUDA implementations is not included in the execution times.

5.1 OpenMP

Figure 7 shows the results for the OpenMP implementations of the cyclic reduction and Bondeli's algorithm for increasing system sizes from 2^{10} to 2^{22} in steps of 2^2. The best results for the cyclic reduction implementation are obtained for 2^{16}-equation systems, which presents an speedup of 2.15x. The decrease in speedup for larger systems is due to the storage requirements of the algorithm. For example, a system of 2^{20} equations requires 20 MB of memory which is more than the 12 MB of the L2 cache. Consequently, the number of cache misses increases and, given that the data accesses do not present spatial locality, the execution time also increases.

For small systems Bondeli's algorithm presents lower speedups than the cyclic reduction algorithm, mainly due to the high arithmetic requirements of the algorithm. Nevertheless, its speedup curve presents a more linear behavior than for the cyclic reduction algorithm because the intermediate subsystems are solved with the Thomas algorithm, which presents a sequential access data pattern. Another reason is a good workload balance. As shown in Figure 7, a speedup of 1.14x is achieved for a 2^{16}-equation system, lower than for the case of cyclic reduction.

Table 1: Performance results obtained for a 2^{20}-equation system (CR denotes cyclic reduction).

Algorithm	Thomas	CR	Bondeli
Sequential			
Time	0.0547 s	——-	——-
GFLOPS	0.14		
OpenMP			
Time	——	0.0507 s	0.0365 s
GFLOPS		0.44	0.78
CUDA			
Time	——	0.0029 s	0.0058 s
GFLOPS		15.33	17.50

Table 1 shows a summary of the performance results we have obtained for the different implementations for a 2^{20}-equation tridiagonal system using single floating points arithmetic. The OpenMP cyclic reduction exhibits a speedup of 1.08x and Bondeli's algorithm 1.50x, with 0.44 and 0.78 GFLOPS, respectively. The GFLOPS rate for Bondeli's algorithm is nearly twice that for the cyclic reduction owing to the higher arithmetic requirements of the first as mentioned above.

The key for the OpenMP implementations is performing a good schedule that maximizes cache locality, especially in cyclic reduction, and selecting the optimal block size in Bondeli's algorithm to ensure a good workload balance. Memory cache plays an important role in the speedup of the cyclic reduction method. The best results for Bondeli's algorithm are achieved for large systems due to the memory requirements of the cyclic reduction that can not be satisfied by the multicore memory hierarchy.

5.2 CUDA

Figure 8 shows the speedups of the CUDA implementations for the cyclic reduction and Bondeli's algorithm varying the

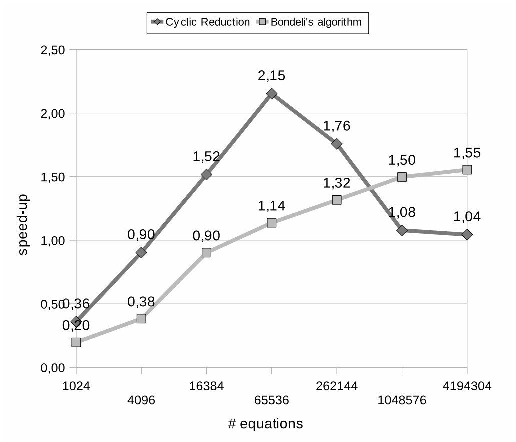

Fig. 7: OpenMP speedup of cyclic reduction and Bondeli's algorithm with different tridiagonal system sizes.

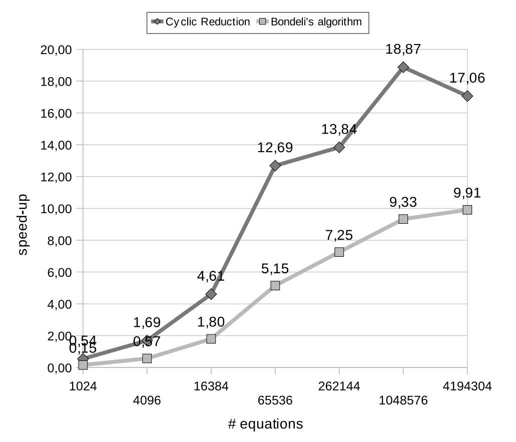

Fig. 8: CUDA speedup of cyclic reduction and Bondeli's algorithm with different tridiagonal system sizes.

system size. For small tridiagonal systems, the speedup is low, as the available parallel hardware cannot be fully exploited due to the small number of computations associated. For larger systems, the GPU shows good performance in terms of speedup. Both algorithms scale well as the system size increases. This is a typical behavior for the GPUs, which need to be fed with thousands of data to exploit their computational capabilities to the full [14]. A maximum speedup of 18.87x is achieved for the cyclic reduction implementation for a system of 2^{20} equations and a GFLOPS rate of 15.53 (see Table 1). A lower speedup is obtained by Bondeli's algorithm, with 9.3x for the same system size but a higher GFLOPS rate of 17.50. The reason is, as for the OpenMP implementations, the higher number of computations associated to Bondeli's algorithm.

6. Conclusions

We have analyzed two tridiagonal system solvers (cyclic reduction and Bondeli's algorithm) exhibiting different levels of parallelism in two different parallel hardware platforms: multi-core architectures (CPU) and GPUs platforms. The implementations are analyzed in terms of speedups and GFLOPS with respect to a sequential implementation of the Thomas algorithm. These algorithms were executed in a PC with an Intel Core 2 Quad Q9450 and a NVIDIA GeForce GTX 295.

In OpenMP, for small systems the cyclic reduction algorithm presents a better speedup than Bondeli's algorithm. This is mainly due to the high computational requirements of Bondeli's algorithm. When the system size increases over 2^{16} equations (large systems) the performance of the cyclic reduction implementation decreases as the data storage requirements for the algorithm exceed the capacity of the L2 cache, increasing the data movement. The speedup changes more linearly for Bondeli's algorithm because of the good workload balance and because it uses the Thomas algorithm which present sequential accesses to memory, for solving the subsystems generated by the algorithm. For systems of 2^{20} equations the best results are a speedup of 1.50x obtained by Bondeli's algorithm and a GFLOPS rate of 0.78.

The results in CUDA are always better for cyclic reduction. The complex algorithm structure and the different types of stages involved in Bondeli's algorithm makes the implementation thereof a challenge. For a 2^{20} equation system, the best results of the CUDA implementation are a speedup of 18.87x with a GFLOPS rate of 17.50.

The results have shown how the cyclic reduction solver, that exhibits fine-grained parallelism, achieves always best results in the GPU comparing to the Bondeli's algorithm, with a coarse-grained parallelism. This also happens in a multicore CPU when the system of equations was small enough to fit into the cache. When the size of the system is larger than 2^{20}, Bondeli's algorithm presents better results.

In conclusion, fine-grained parallelism problems often achieve highly positive results in the GPU, although the time invested in adapting other types of problems also has its reward in terms of speedup.

ACKNOWLEDGEMENTS

This work was supported in part by the Ministry of Science and Innovation, Government of Spain, and FEDER funds under contract TIN 2010-17541, and by the Xunta de Galicia under contracts 08TIC001206PR and 2010/28.

References

[1] L. H. Thomas, "Elliptic problems in linear difference equations over a network," *Watson Sci. Comput. Lab. Rept*, 1949.

[2] F. Gauss, "Theory of motion of heavenly bodies," 1809.

[3] B. L. Buzbee, G. H. Golub, and C. W. Nielson, "On direct methods for solving poisson's equations," *SIAM Journal Numerical Analysis*, vol. 7(4), pp. 627–656, 1970.

[4] R. W. Hockney, "A fast direct solution of poisson's equation using fourier analysis," *ACM*, vol. 12, pp. 95–113, 1965.

[5] H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," *Journal of the ACM*, vol. 20, pp. 27–30, 1973.

[6] S. M. . S. D. Müller, "A method to parallelize tridiagonal solvers parallel computing," vol. 17, pp. 181–188, 1991.

[7] H. H. Wang, "A parallel method for tridiagonal equations," *ACM Trans. Math. Softw.*, vol. 7, no. 2, pp. 170–183, 1981.

[8] S. Bondeli, "Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations," *Parallel Comput.*, vol. 17, no. 4-5, pp. 419–434, 1991.

[9] OpenMP Architecture Review Board, "OpenMP application program interface, Specification," 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[10] NVIDIA, "Cuda technology," Nvidia Corporation, 2007. [Online]. Available: http://www.nvidia.com/CUDA

[11] M. Harris, "General-purpose computation on graphics hardware," 2002. [Online]. Available: http://www.gpgpu.org

[12] S. Allmann, T. Rauber, and G. Runger, "Cyclic reduction on distributed shared memory machines," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference*, vol. 0, p. 290, 2001.

[13] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[14] D. B. Kirk and W. m. W. Hwu, *Programming Massively Parallel Processors: a Hands-on Approach*. Massachussets: Elsevier, Burlington, 2010.

[15] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 127–136.

[16] D. Goddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on gpus applied to mixed-precision multigrid," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 22–32, 2011.

[17] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 2:1–2:12.

[18] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-d fft kernel for gpus using cuda," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 5:1–5:11.

[19] "Tridiagonal system solvers, internal report," Department of Electronics and Computer Science, University of Santiago de Compostela, Santiago de Compostela, Spain," Technical report, 2011.

[20] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide*, May 2011. [Online]. Available: http://www.intel.com/products/processor/manuals