Centro Singular de Investigación en Tecnoloxías da Información

The Split-and-Merge Method in General Purpose Computation on GPUs

Francisco Argüello
Departamento de Electrónica e Computación, Universidade de Santiago de Compostela
francisco.arguello@usc.es

Dora B. Heras
Centro Singular de Investigación en Tecnoloxías da Información
dora.blanco@usc.es
http://citius.usc.es

Montserrat Bóo
Departamento de Electrónica e Computación, Universidade de Santiago de Compostela

Julián Lamas-Rodríguez
Centro Singular de Investigación en Tecnoloxías da Información
julian.lamas@usc.es
http://citius.usc.es

This work was supported in part by Xunta de Galicia under contracts 08TIC001206PR and 2010/28, and by Ministry of Education and Science of Spain and FEDER funds under contract TIN2007-67537-C03-01.

Abstract

The split-and-merge method is an algorithm design paradigm sometimes used in the field of parallel computing. It is applied to multilevel algorithms such as the wavelet transforms and some tridiagonal system solvers. In this paper we present the application of the method in the context of general purpose computation on GPUs. The split-and-merge method allows us to efficiently use the CUDA parallel programming model, where a multithreaded program is partitioned into blocks of threads that execute independently from each other. Thus we can solve the data dependency problem at the block boundaries and efficiently take advantage of the memory hierarchy of the GPU. The results obtained show a significant acceleration compared with the direct implementation of the algorithms on the GPU.

1 Introduction

A modern GPU is a highly parallel architecture that can handle thousands of threads running concurrently. Originally designed for graphics processing, nowadays GPUs can be used to accelerate a wide range of applications. In the CUDA parallel programming model [1], a multithreaded program is partitioned into blocks of threads that execute independently from each other. This model guides the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order. That is, blocks can be executed in parallel if there are available units, otherwise, they will be executed sequentially. This is shown in Fig. 1.

CUDA threads may access data from multiple memory spaces during their execution [2]. From the programmer's point of view, the most important ones are the global memory and the shared memory. The global memory can be accessed by all threads, and data stored here is available throughout the program execution. In contrast, each block has its own shared memory which is visible only to the threads of the block and which has the same lifetime as the block. Shared memory enables cooperation between threads in a block; however, it is not accessible to the threads of the other blocks. As it is on-chip, shared memory is much faster than global memory. When multiple threads in a block use the same data, shared memory can be used to access the data from global memory only once.

In a computation model based on blocks, when data have to be processed one block at a time in sequential systems or computed over multiple processors in parallel systems, the major difficulty occurs in the computations near data boundaries. Some algorithms have a multilevel computation scheme; i.e., they are structured into a set of stages in each of which the partial results are computed from data located in nearby positions [3, 4, 5]. This is the case of wavelets and other orthogonal transforms, and of some tridiagonal system solvers [6, 7]. Depending on the type of architecture, overlapping techniques, such as data replication near data boundaries or non overlapping techniques based on extra communications, can be efficient solutions [8].

In some cases, the direct application of these techniques on multilevel algorithms can be very costly in terms of memory and/or inter-processor communication operations. This problem has been studied extensively in [9] for the computation of the wavelet transform. Either the blocks are given sufficient overlapped data to carry on the whole computation

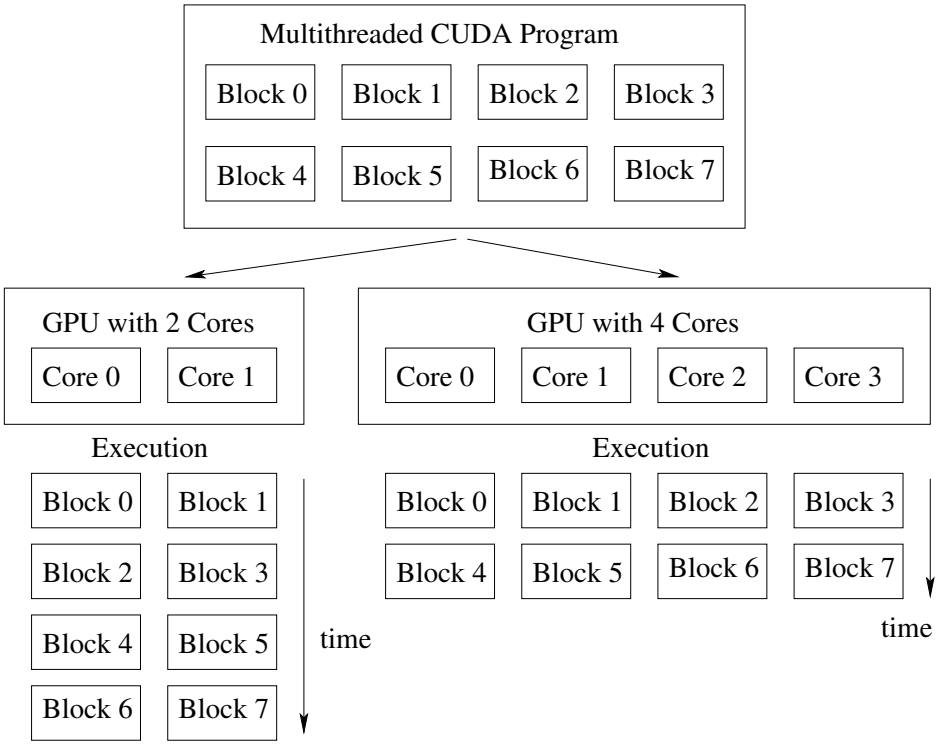

Figure 1: A CUDA program is partitioned in blocks of threads that execute independently from each other.

without communicating with each other, or alternatively, they have to communicate data after each level has been computed. The first approach, overlapping, requires that input data near the block boundaries be given to both blocks. Since each block has to compute its own partial results for multiple levels, this overlap can be quite substantial. For example, in the case of a wavelet transform or of some tridiagonal system solvers, the overlap increases exponentially with the increase of the level. In the second approach, non-overlapping, input data is not overlapped so the memory requirement is relaxed, but boundary samples need to be exchanged at each decomposition level.

Jiang and Ortega [9] present a boundary postprocessing technique named split-and-merge (SM) in the context of architecture design for computing wavelet transforms. The key observation is that the partially computed results can also be stored back to their original locations and the transform can be continued anytime later as long as these partially computed results are preserved. The idea is motivated by the standard overlap-add technique which initially performs the operations that can be carried out with the data contained in each block and subsequently completes the computation by performing the remaining operations. This idea is extended to the case of multilevel wavelet decompositions.

Some algorithms for solving tridiagonal systems of equations have a dependency graph similar to that of the wavelet transform. [10] has a comparison of the different tridiagonal system solvers from the point of view of their implementation on the GPU. The aforementioned study takes into account the use of the shared memory of the GPU; however, it focuses more on the projection of tridiagonal systems than on the study of the SM method.

In this paper we study the application of the SM method in the context of General Purpose Computation on GPUs. The architecture of computation in blocks considered

is CUDA, which allows us to execute the same code on different GPU models. In each execution of a kernel (function of CUDA code called from the main program), blocks are executed in parallel if there are enough processors available, or otherwise sequentially. Hence data communication between blocks is not possible. Additionally there is a synchronization problem: during the execution of a kernel only those threads that run within of the same block can be synchronized. Finally, an efficient use of the memory hierarchy implies the use of shared memory, which is faster than the global one. All these limitations of the architecture will be taken into consideration in the application of the SM method to GPU programs.

The rest of this paper is organized as follows: Section 2 presents the class of algorithms considered and the application of the SM method; in Section 3 we implement these algorithms in CUDA studying the parameters leading to the best performance: size of the sections of split and merge, memory space used in each case, reduction of the number of threads, etc; Section 4 describes the extension of the method to two dimensions; in Section 5 we perform the evaluation; and finally, in Section 6 we present the conclusions.

2 The split-and-merge method

In this section we present the application of the SM method to the GPU, taking some widely used algorithms as models. The selection of algorithms was conducted with the objective of having a wide range of data-flow diagrams.

In many useful algorithms that operate on arrays or matrices, data are read from memory, processed, and partial results are stored back to their original positions. The results are usually computed, using not only the data point whose position is rewritten, but also from data located in nearby positions [11]. Some of these algorithms have a multilevel computation scheme; i.e., they are structured in a set of stages in each of which the operation described above is performed. In these algorithms the computation of the successive levels frequently needs input data from increasingly distant positions, and/or that downsampled data sequences from the previous level are used. This is true for of wavelets and other orthogonal transforms, and for some tridiagonal system solvers [6, 7].

In the class of the multilevel algorithms considered, the i-th partial result of a certain level (x'_i) is obtained from the input data of the previous level (x_i) as

$$x'_i = f(...,x_{i-2a},x_{i-a},x_i,x_{i+a},x_{i+2a},...). \tag{1}$$

We consider that the distance at which the data is located, that is, the parameter a, increases with the level. In practice, this parameter usually increases as a power of two; i.e., the first stages of the algorithm take the form:

$$\begin{aligned} x'_i &= f(...,x_{i-2},x_{i-1},x_i,x_{i+1},x_{i+2},...) && (2)\\ x''_i &= f(...,x'_{i-4},x'_{i-2},x'_i,x'_{i+2},x'_{i+4},...) && (3)\\ x'''_i &= f(...,x''_{i-8},x''_{i-4},x''_i,x''_{i+4},x''_{i+8},...). && (4) \end{aligned}$$

When this type of algorithm is to be computed by blocks of threads on the GPU, the major difficulty is found in the computations near block boundaries. The SM method allows

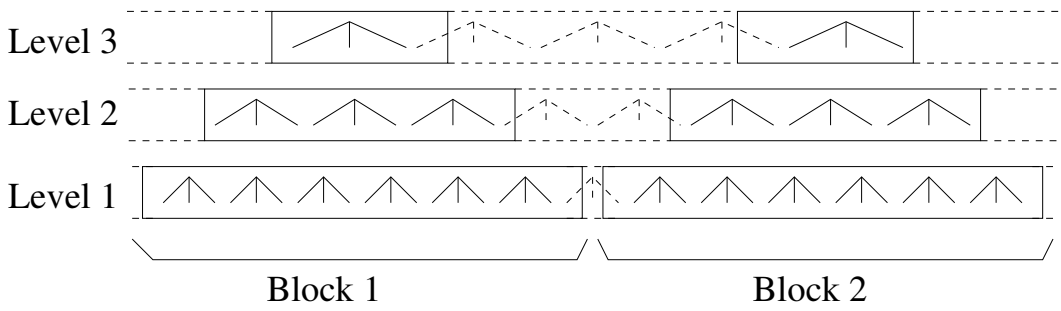

Figure 2: Application of the SM method to a multilevel algorithm. Solid lines mark the data that can be processed independently within each block, while dashed lines mark the data to be processed later.

each block to process data at different levels of the algorithm. The number of data that can be processed within each block decreases with increasing level, as illustrated in Fig. 2 (in this and in the following figures, levels are arranged from bottom to top and the butterfly diagrams, interconnecting inputs and outputs, represent the data dependencies at each level). The partially computed results are sent back to the global memory and the remaining computations can be carried out later by other thread blocks.

We can classify these algorithms using two different criteria: whether or not they can be computed in-place and whether or not they require decimation as the level increases. In an in-place algorithm, the partial results, as soon as they are computed, can be immediately written on the positions of the input data used in their calculation. In the not-in-place algorithms, the original data cannot be overwritten until the calculation of all partial results has been completed. Therefore, a not-in-place algorithm requires double the amount of memory than an in-place one. Often, an algorithm is in-place or not depending not on the nature of the algorithm, but on how the computations are organized. We consider that an algorithm is in-place if it is in accordance with the CUDA model; i.e., whether the algorithm can be computed in-place using the thousands of threads that run in parallel on the GPU. For example, according to this criteria, an algorithm that computes

$$x'_i = f(x_{i-1}, x_i, x_{i+1}), \quad i = 0, 1, 2, 3, \ldots, \tag{5}$$

is not-in-place because the input data x_i is needed to obtain the results x'_{i-1}, x'_i and x'_{i+1}, which are computed by different threads. However, an algorithm that computes

$$x'_i = f(x_{i-1}, x_i, x_{i+1}), \quad i = 0, 2, 4, 6, \ldots, \tag{6}$$

is in-place as in this case the input data x_i is only needed by the thread that computes x'_i.

In algorithms without decimation, the number of data and computations is kept constant over all levels. In algorithms with decimation, the data sequence is downsampled at each level. Usually, a downsample factor of two is used; i.e., only half of the data from a certain level progresses to the next one. If there is a sufficient number of levels, the decimation will progress to achieve a single data point; i.e., the algorithm will have a pyramid-like structure.

In algorithms where the number of levels is high, it may be more efficient to apply the SM method more than once. In this case, each split and merge stage will cover some levels of the algorithm. This is shown in Fig. 3 for a pyramid-like algorithm. In this figure, the levels are arranged from bottom to top, each trapezoid or triangle represents a section of

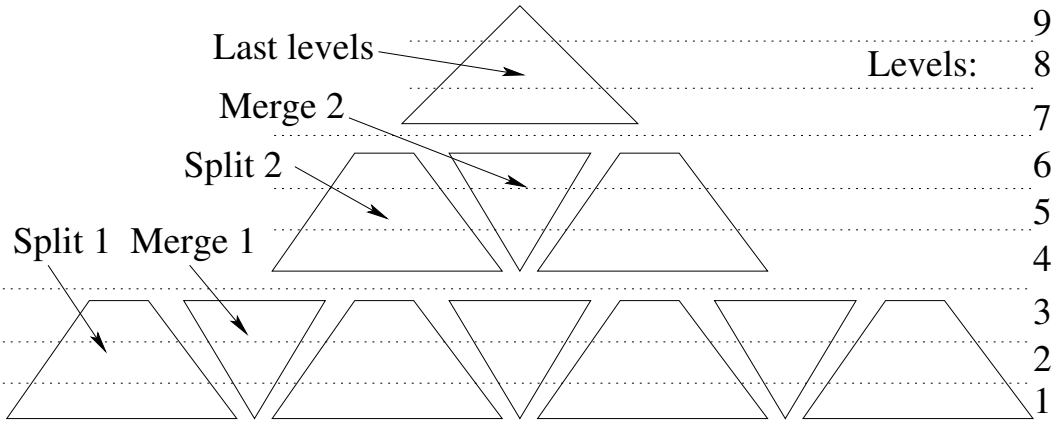

Figure 3: Successive application of the SM method to a pyramid-like multilevel algorithm.

split or merge, respectively, each one of these sections contains a set of computations that can be executed independently, and the width of the trapezoid or triangle at each level is proportional to the number of computations that are performed at that level. The number of levels which are covered by each stage is a parameter that can be adjusted to achieve lower execution times and, in general, will depend on the number of computations and memory accesses that are performed at each level of the algorithm. For each stage of split and merge the data blocks should be selected to maximize the number of partial results which can be computed with the data contained within these blocks. For example, in the algorithms with decimation, blocks do not include data that are not required as input to the computations of the levels.

Examples of multilevel algorithms to which the method is applicable are the wavelet transforms and some tridiagonal system solvers. In the case of wavelet transformations the method is applicable to both lifting and filter bank algorithms. It is also applicable to 1D and 2D transformations. For solving tridiagonal systems of equations, there is a wide variety of resolution procedures, including the cyclic reduction and the recursive doubling algorithms, to which the SM method is applicable. The algorithms that will be used in this paper as models of application of the SM method are described briefly below.

The Cohen-Daubechies-Feauveau (9,7) wavelet is a transformation widely used in signal and image processing [12]. This transformation can be computed in-place, using the known lifting scheme, so that the partial results overwrite the input data used in their calculation [13]. Each level in this transformation includes four lifting steps, in each of which each partial result is computed from three data points as (scale factors are omitted):

$$x_{i+1} = x_{i+1} + \alpha(x_i + x_{i+2}),\ i = 0,2,4,\ldots, \quad (7)$$
$$x_i = x_i + \beta(x_{i+1} + x_{i-1}),\ i = 0,2,4,\ldots, \quad (8)$$
$$x_{i+1} = x_{i+1} + \gamma(x_i + x_{i+2}),\ i = 0,2,4,\ldots, \quad (9)$$
$$x_i = x_i + \delta(x_{i+1} + x_{i-1}),\ i = 0,2,4,\ldots. \quad (10)$$

Depending on the application, the transformation may comprise one or more levels. At the lowest level all data are processed, but in the following levels the data processed is half of that processed in the previous level (a quarter if the sequence is two-dimensional),

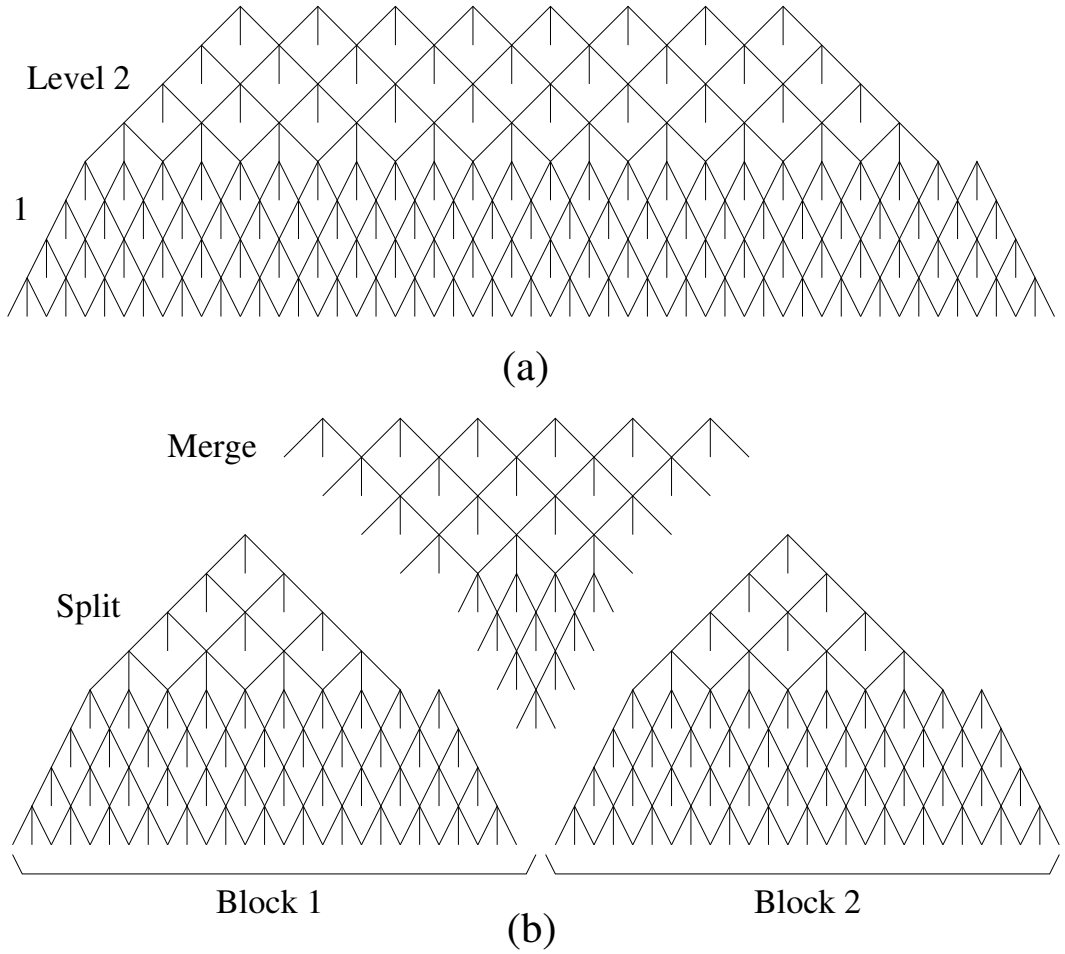

Figure 4: A two-level (9,7) wavelet (with lifting). (a) Dependency graph. (b) Application of the SM method using two blocks.

which is called decimation. The low-frequency coefficients are passed on to the next level for further analysis and the high-frequency coefficients are stored as a result. Moreover, the computation of the lifting steps of the successive levels needs data from increasingly distant positions (in Eqs. (7)-(10), $i = 4^k$ at level 2, $i = 8^k$ at level 3, etc).

The dependency graph of a two-level (9,7) wavelet is shown in Fig. 4.a, which shows illustrating that each level consists of four lifting steps. The application of the SM method to this transformation in the context of architecture design is presented in [9]. We shall adapt the method so that the wavelet can be computed efficiently on GPUs using CUDA. Fig. 4.b shows the application of the SM method to this wavelet using two blocks. The split section obtains the partial results that can be computed independently from the data contained within each block, while the remain operations are carried out in the merge section.

The Daubechies D4 wavelet is a transformation that uses two filters of length 4, so that, at each filter step, it takes four inputs and generates two partial results [14]. Then the filters are moved two positions and two more partial results are generated. This transformation can be computed using lifting steps as in the previous wavelet, but the following equations represent the filter bank version:

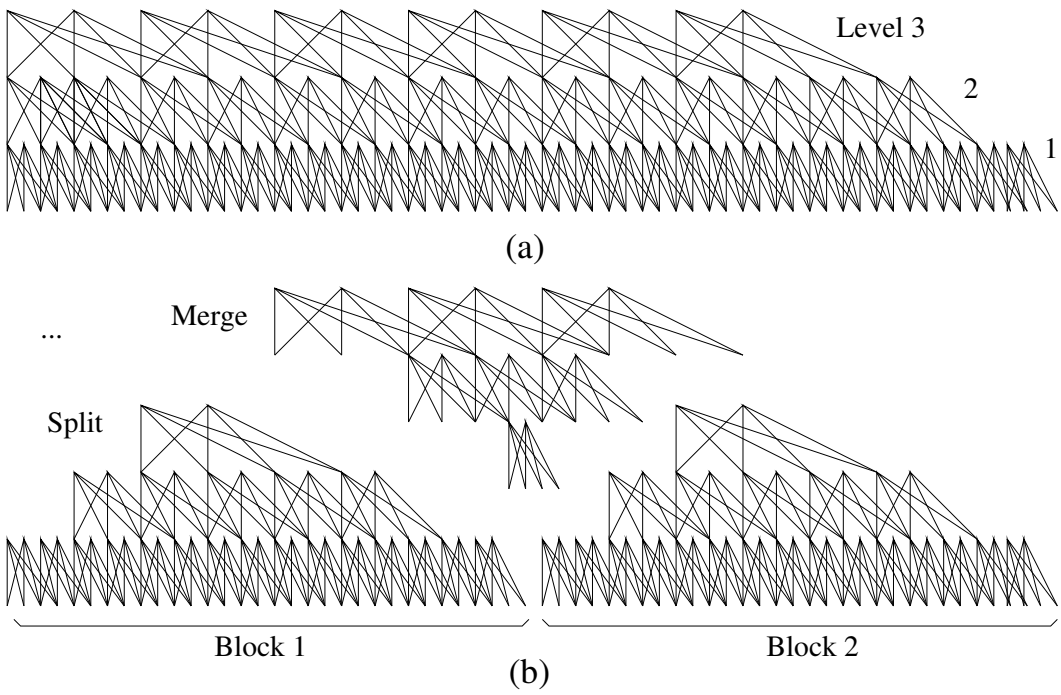

Figure 5: A three-level D4 wavelet (filter bank). (a) Dependency graph. (b) Application of the SM method using a not-in-place scheme.

$$x_i = h_0 x_i + h_1 x_{i+1} + h_2 x_{i+2} + h_3 x_{i+3},\ i = 0,2,4,\ldots, \quad (11)$$

$$x_{i+1} = h_3 x_i - h_2 x_{i+1} + h_1 x_{i+2} - h_0 x_{i+3},\ i = 0,2,4,\ldots. \quad (12)$$

In Fig. 5.a we show a dependency graph for a three-level transformation. The disadvantage of using a scheme without lifting is that, when the transformation is computed in parallel using CUDA, partial results cannot overwrite the input data used in their calculation, hence it requires twice the memory than the lifting scheme. Figure 5.b shows the application of the SM method to a two-block section of this wavelet. Since this algorithm is not-in-place, special care must be taken during the computation of the split section to avoid do overwriting the partial results that will subsequently be needed in the merge section. These results must also be transferred from shared memory to global memory when the computation of the split section is completed.

The cyclic reduction is a method for the resolution of tridiagonal systems which is widely used on parallel computers [7]. It consists of two phases: forward reduction and backward substitution. Each level of the forward reduction phase has a dependency graph similar to a lifting step of the (9,7) wavelet. Each partial result is computed from three data points located in positions with a separation power of two which increases with the level. Moreover, all calculations can be performed in-place. In contrast to the (9,7) wavelet, the pyramid must be computed complete; i.e., it requires $\log_2 N$ levels for systems of equations of size N.

The dependency graph for solving a system of 16 equations is shown in Fig. 6.a (in this figure, the computations are arranged from left to right). During the forward reduction phase, each data point represents a set of four coefficients in the system (one coefficient

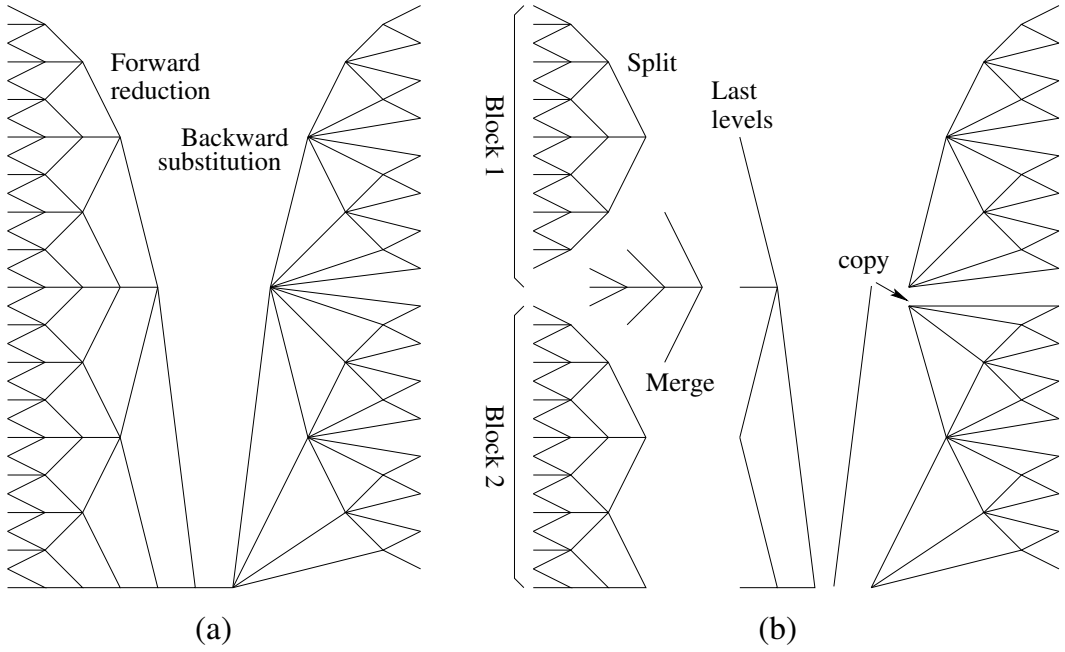

Figure 6: Cyclic reduction. (a) Dependency graph. (b) Application of the SM method using 2 blocks.

from each diagonal and one right-hand coefficient). Fig. 6.b shows the application of the SM method to the forward reduction phase. Since it is a complete pyramid, once the SM method has been applied, some remaining levels must be computed additionally. The backward substitution phase is of no interest to us since each block can be computed independently with the additional copy of a single data point.

In the previous algorithms the data are decimated at each level. Some packet wavelet transforms, in contrast, do not apply decimation, that is, the number of computations remains constant throughout all levels of the algorithm. The wavelet packet transforms are used, for example, in the calculation of the "best basis", which is a minimal representation of the data relative to a particular function in applications that include noise reduction and data compression [15, 16]. In these multilevel transformations, as in the previous cases, the distance to the locations of the input data increases exponentially (as powers of two) with the increase of the level. In Fig. 7 we show the dependency graph of a 3-level wavelet packet based on the D4 wavelet (filter bank). The application of the SM method to a two-block section of this transform is shown in Fig. 7.b. Since this computation scheme is not in-place, care must be taken in transferring the partial results computed in the split section that will be needed after in the merge section from the shared memory to the global memory.

3 Implementation of the SM method on the GPU

In this section we present the implementation details of the SM method on the graphics card, taking into account the parameters that influence the performance of the algorithms on the GPU: number of levels in each section, use of the memory spaces, reduction of the number of threads, and size of the split and merge sections. In a direct implementation of

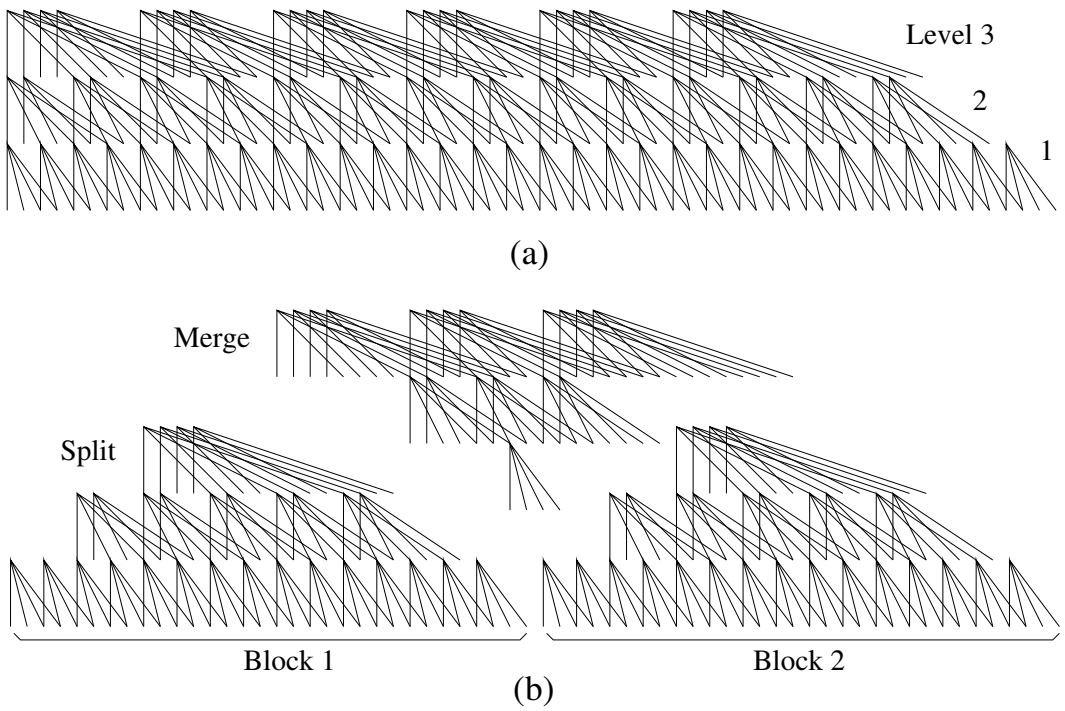

Figure 7: A 3-level packet D4 wavelet (filter bank). For clarity, only half of each butterfly is shown. (a) Dependency graph. (b) Application of the SM method.

the multilevel algorithms on GPUs, each level of the algorithm can be computed by calling a kernel (there will be the same number of calls as there are levels). This is possible because there is a global synchronization of all threads between calls to kernels. The drawback of the direct implementation is that in each level that comprise the algorithm all input data is read and all partial results are written in the global memory, slower.

The SM method allows an efficient use of the memory spaces of the GPU. Specifically, we can compute several levels of the algorithm with the data contained in the shared memory, faster. The data transfer between global memory and shared memory is performed only at the beginning and end of the split section. All intermediate operations can be performed with the data stored in the shared memory of the blocks. The merge section, although it often requires fewer operations than the split section, also can be computed in this way.

In Fig. 8 we show the execution times and speedups for the algorithms described in the previous section on the GPU, comparing the direct implementation in global memory with the one based on the SM method, for different problem sizes. We show the results obtained for the (9,7) wavelet (with lifting), the cyclic reduction method of resolution of tridiagonal systems, the D4 wavelet (filter bank), and the packet D4 wavelet (filter bank). The speedup shown is calculated by dividing the execution time of the direct implementation in global memory by that of the SM version. The graphics card used is an Nvidia GTX 480 with 1538 MB of memory and the block size is set to 256 threads. For a problem size of 2^{20} the speedups obtained are 4.01x, 2.12x, 1.92x, and 1.40x, respectively. As an example, taking as base the sequential implementations in CPU (on one core of an Intel Q9450 at 2.66 GHz in Linux, compiled with gcc with the O3 optimization) the speedups of the SM versions are 35.7x, 57.2x, 25.6x, and 55.6x, respectively.

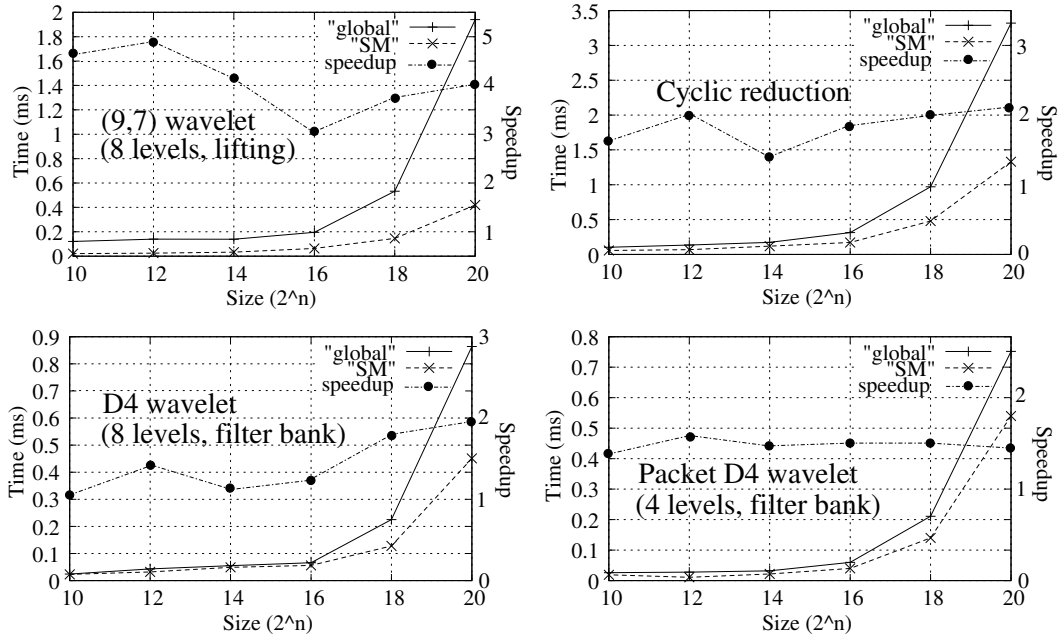

Figure 8: Execution times (left scale) and speedup (right scale) of several algorithms with direct implementation (in global memory) and using the SM method on a GPU Nvidia 480 GTX.

Next, we turn to study in detail the parameters that can be adjusted in the SM method to render the implementation more efficient. Table 1 shows, for different values of the parameters, the execution times of a 8-level (9,7) wavelet of size 2^{20}, and Table 2 shows the same for the algorithm cyclic reduction of a system of 2^{20} equations. The first two rows of the tables show the execution times of direct implementations that use only global memory, while the remaining rows show the execution times obtained using the SM method with different parameters. The speedup values are calculated taking as reference the direct implementation on GPU (global memory and $T = 1$, first row in the table). The adjustable parameters are explained below.

The measurements were performed on three different Nvidia cards: 9800 GT (1 GB of RAM), 295 GTX (900 MB), and 470 GTX (1538 MB). The 9800 GT has 14 multiprocessors and 112 CUDA cores, 16 kB of shared memory per multiprocessor, 1.5 GHz processor clock, and 57 GB/s memory bandwidth. The GTX 295 has 30 multiprocessors, 240 CUDA cores, 16 kB of shared memory per multiprocessor, 1.2 GHz processor clock, and 112 GB/s memory bandwidth (per GPU). Finally, the GTX 480 has 15 multiprocessors and 480 CUDA cores, 48 kB of shared memory and 16 kB of L1 cache (or vice versa) per multiprocessor, 1.4 GHz processor clock, and 177 GB/s memory bandwidth. The maximum number of resident blocks per multiprocessor is 8, while the maximum number of threads per block is 512 for the two first GPUs and 1024 for the GTX 480. Codes were written in C using the version 3.2 of Nvidia CUDA and executed under Linux.

Performance measures were obtained as an average of one thousand executions of each algorithm. The time associated with the data transfer between CPU and GPU memory, which is the same for all algorithms, is not included in the execution times. This time can be ignored if the wavelet is used as one step in a large computational algorithm fully executed on the GPU.

Type	L	Split	Merge	T	9800 GT	295 GTX	470 GTX
Global	1	-	-	1	12.63 (1.00x)	4.26 (1.00x)	1.94 (1.00x)
Global	1	-	-	$1/2$	16.06 (0.79x)	3.95 (1.08x)	1.10 (0.97x)
SM	1	Shared	Shared	1	2.75 (4.59x)	0.95 (4.49x)	0.63 (3.11x)
SM	1	Shared	Shared	$1/2$	2.41 (5.25x)	0.81 (5.29x)	0.53 (3.66x)
SM	1	Shared	Global	1	3.48 (3.63x)	1.37 (3.16x)	0.85 (2.30x)
SM	1	Shared	Global	$1/2$	2.86 (4.42x)	1.08 (3.94x)	0.64 (3.03x)
SM	2	Shared	Shared	1	2.39 (5.29x)	0.92 (4.62x)	0.53 (3.70x)
SM	2	Shared	Shared	$1/2$	1.70 (7.46x)	0.72 (5.93x)	0.42 (4.60x)
SM	2	Shared	Global	1	3.55 (3.56x)	1.44 (2.95x)	0.87 (2.24x)
SM	2	Shared	Global	$1/2$	2.36 (5.35x)	1.05 (4.04x)	0.58 (3.36x)
SM	4	Shared	Shared	1	4.46 (2.83x)	1.61 (2.64x)	0.72 (2.72x)
SM	4	Shared	Shared	$1/2$	2.75 (4.60x)	1.05 (4.07x)	0.49 (3.94x)
SM	4	Shared	Global	1	5.56 (2.27x)	2.10 (2.03x)	1.27 (1.53x)
SM	4	Shared	Global	$1/2$	3.37 (3.75x)	1.46 (2.92x)	0.74 (2.61x)

Table 1: Execution times (in milliseconds) and speedups of different implementations of the (9,7) wavelet of size 2^{20}, 8 levels, with lifting, computed on GPU using blocks of 256 threads.

Type	L	Split	Merge	T	9800 GT	295 GTX	480 GTX
Global	1	-	-	1	15.15 (1.00x)	5.47 (1.00x)	3.32 (1.00x)
Global	1	-	-	$1/2$	19.62 (0.77x)	4.91 (1.11x)	3.40 (0.98x)
SM	4	Shared	Shared	1	5.07 (2.99x)	2.58 (2.12x)	1.40 (2.37x)
SM	4	Shared	Shared	$1/2$	4.88 (3.10x)	2.36 (2.32x)	1.33 (2.50x)
SM	4	Shared	Global	1	4.94 (3.07x)	2.78 (1.97x)	1.55 (2.14x)
SM	4	Shared	Global	$1/2$	5.59 (2.71x)	2.54 (2.15x)	1.44 (2.31x)

Table 2: Execution times (in milliseconds) and speedups of different implementations of the cyclic reduction algorithm of a system of 2^{20} equations computed on GPU using blocks of 256 threads.

Number of levels in the split and merge sections

As explained in section 2, when the number of levels of the algorithm is high, it may be more efficient to apply the SM method several times in succession. In the case of the (9,7) wavelet, this can be seen in the second column of Table 1, where L specifies the number of levels that includes each stage SM. When $L = 1$ each stage SM covers one level (eight stages SM are performed in total since the wavelet of the example has eight levels), when $L = 2$ there are four stages SM of two levels each, and when $L = 4$ there are two stages of four levels each. For this wavelet, the lowest execution time was achieved for $L = 2$. In other algorithms it will depend on the number of computations that each level includes and on the number of data having to be accessed from memory. In the cyclic reduction algorithm, the most efficient case is $L = 4$.

Use of global or shared memory in the merge section

The merge section covers the computations that cannot be performed in the split section with the data contained within a block. Therefore, in principle, the merge section performs a smaller number of computations than the split section does. In some cases it may be advantageous to compute the merge section in global memory. In column 4 of Tables 1 and 2 we compare the performance of the implementations with the merge section computed in global memory and in shared memory. In almost all these cases, lower computation times are achieved by also computing the merge section in shared memory.

Reduction of the number of threads

In some multilevel algorithms each stage of computation generates a number of partial results which is half the number of data used for their calculation. This can be seen in each lifting step of the (9,7) wavelet (shown in Fig. 4.a) and in each level of the cyclic reduction (Fig. 6.a). In a direct implementation of such algorithms on the GPU, while all threads cooperate in accessing data from global memory, only half of the threads carry out computations. In some cases performance may be increased by halving the number of threads. Each thread will perform twice the number of memory accesses than in the previous case but the same number of arithmetic operations. The sequencing of the accesses to global memory (which is otherwise limited by the bandwidth of the bus) can be compensated by the absence of idle threads. The advantages of this strategy can be seen in column 5 of Tables 1 and 2, where T=1 indicates that the number of threads is the same as in the original implementation and T=1/2 that the number of threads is halved.

Size of the split and merge sections

Until now we have considered that the split section performs the maximum number of computations that can be done with the data contained within each block and that the merge section performs the remaining computations. This is not the only alternative, as there is also the possibility of moving part of the computations from the split section to the merge section. This is shown in Fig. 9, where the parameter W adjusts the respective widths of the split and merge sections. The execution times obtained for different values of W are shown in Table 3. It can be seen that the times obtained are very similar in most

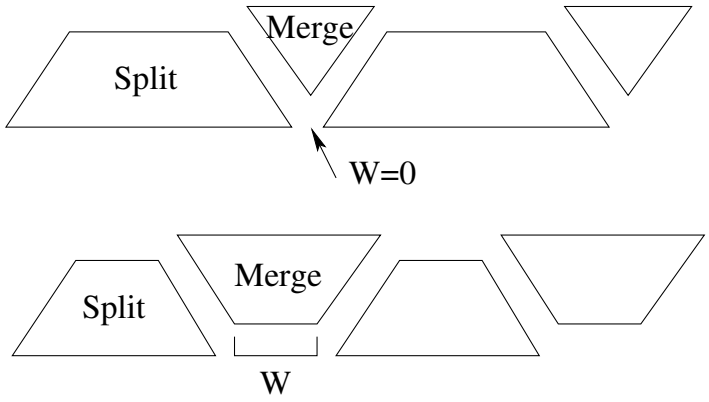

Figure 9: Two possible sizes for the split and merge sections. The upper part of the figure shows the case where the width of the merge section is the minimum possible ($W = 0$), while the bottom shows a case $W \neq 0$.

	(9,7) wavelet			Cyclic reduction		
W	9800 GT	295 GTX	480 GTX	9800 GT	295 GTX	480 GTX
0	1.70	0.72	0.42	4.88	2.36	1.33
16	1.97	0.74	0.43	11.93	2.41	1.36
32	2.03	0.71	0.43	6.77	2.34	1.37
64	2.18	0.70	0.41	7.45	2.35	1.32
128	2.35	0.71	0.38	10.16	2.36	1.30

Table 3: Execution times (in milliseconds) for different sizes of the split and merge sections.

cases. This result could be expected as the number of operations is the same in all cases, although with different distribution of operations between the split and merge sections. For example, for the (9,7) wavelet computed on the 295 GTX and for $W = 0$, the split section consumes 640 μs and the merge section 82 μs, while for $W = 128$, these times are 491 and 220 μs, respectively.

4 Extension of the SM method to two dimensions

The SM method can be extended easily to the two-dimensional case. In this section we analyze the performance of different partitioning schemes of a 2D sequence on the GPU. As a model of 2D multilevel algorithm, we consider the 2D wavelet transformation. The basic scheme of computation of the transformation of a 2D data sequence is the row-column decomposition. This scheme consists of two phases, an initial one that computes one-dimensional transformations on each of the rows, and a second one which does the same on each of the columns. An alternative scheme, known as square decomposition, alternates and combines computations on rows and columns. When the 2D transformation is performed on the GPU, the factor that most influences the performance is the distribution of computations among the blocks of threads.

Fig. 10 shows three possible partitioning schemes of the 2D array of data among the blocks of threads, suitable for an implementation with global memory. The first scheme

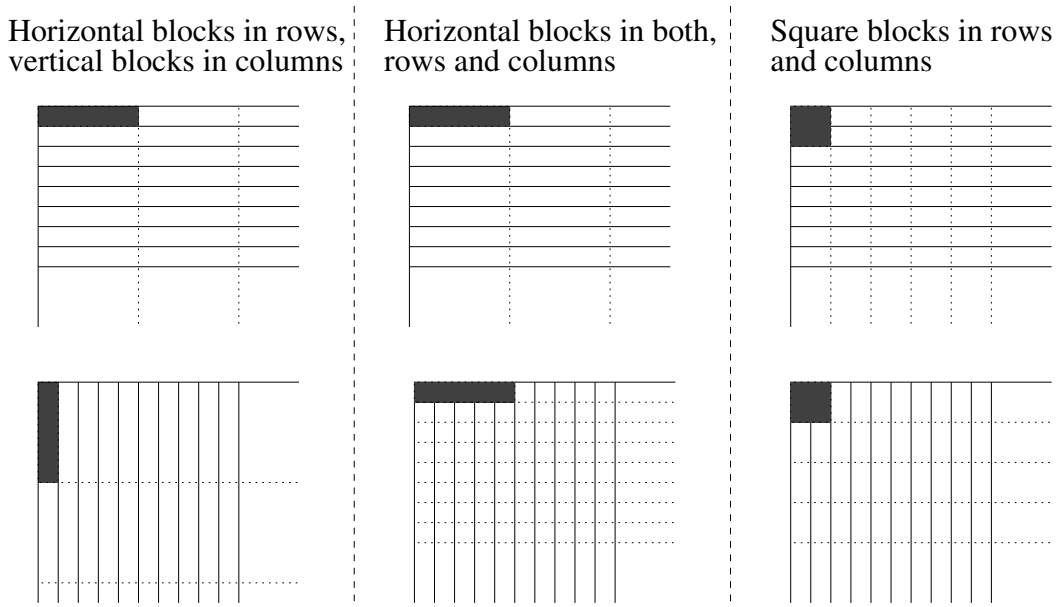

Figure 10: Three different partitioning schemes of a 2D transform on the GPU using global memory. They use the row-column decomposition and different types of partitions of rows and columns among the blocks of threads.

involves an initial phase of partition of the rows in horizontal blocks followed by a second phase of partition of the columns in vertical blocks. The shaded rectangles in the figure shows each of these blocks. This partitioning results in a poor implementation as the vertical blocks of threads have highly inefficient memory access patterns. The second approach solves this problem by carrying out the block partition of the second phase in the same way as in the first one; i.e., using horizontal blocks also for the columns. Thus the threads of a block perform access to memory data maintaining coalescence and minimizing the stride [2]. In the row transformation phase, each block will perform the transformation of part of a row. However, in the column transformation phase, each block will compute in parallel a partial result of a set of columns. Finally, the third scheme shows a row-column decomposition with square blocks. Depending on the size of the blocks, the efficiency of this scheme may be similar to that of the previous one.

The application of SM method to the 2D transformation enables us to make efficient use of the shared memory. In Fig. 11 we show two partitioning schemes that use the SM method. In the left-hand part of the figure, the SM method is applied to a row-column decomposition taking horizontal blocks for the rows and square blocks for the columns. In the row transformation phase, the split and merge operations are carried out in shared memory in the same way as the one-dimensional sequences. The column transformation is carried out in square blocks, using the SM method in parallel on multiple columns. The right part of the figure, by contrast, implements a square decomposition. The 2D data sequence is partitioned into square blocks, and a single split section is computed using the data stored in the shared memory of each block, transforming both rows and columns. Next, two merge sections are computed, an initial one that completes the transformation of the rows and a second one which completes the columns. Table 4 shows the execution times obtained for the different implementations, where can be seen that the two SM schemes

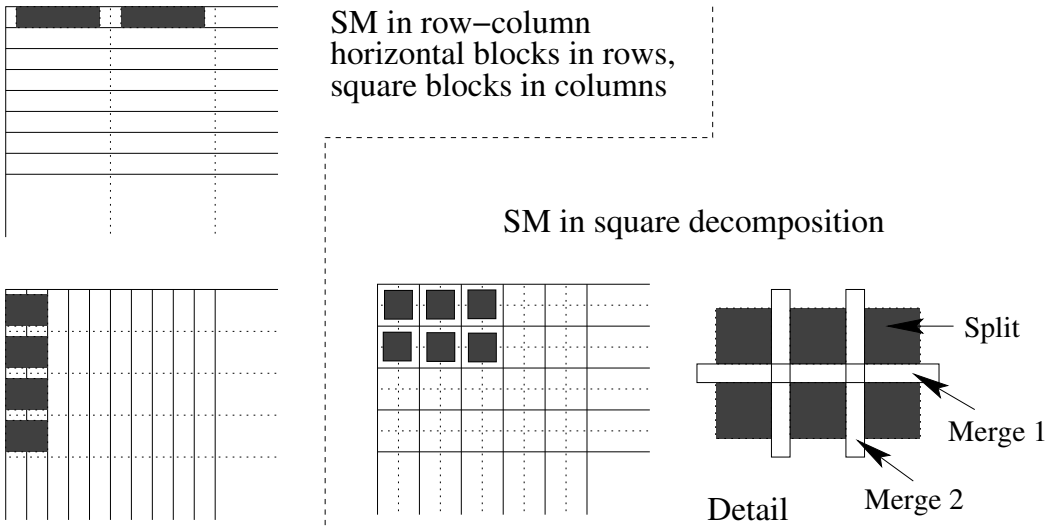

Figure 11: Two partitioning schemes of a 2D transform on the GPU using the SM method.

Type	Rows	Columns	9800 GT	295 GTX	480 GTX
Global	256×1	1×256	572.05 (1.00x)	132.70 (1.00x)	20.32 (1.00x)
Global	256×1	256×1	44.52 (12.9x)	11.92 (11.1x)	4.94 (4.12x)
Global	16×16	16×16	44.52 (12.8x)	10.80 (12.3x)	5.21 (3.90x)
SM	256×1	16×16	12.47 (45.9x)	4.33 (30.7x)	2.25 (9.03x)
SM	16×16	16×16	14.95 (38.3x)	3.77 (35.2x)	2.43 (8.41x)

Table 4: Execution times (in milliseconds) and speedups for the (9,7) wavelet of size 2048×2048, 4 levels, with lifting, computed on GPU using blocks of 256 threads. We specify the geometry of the blocks in the phases of transformation of rows and columns.

give similar performance. The lower speedup in the GTX 480 compared to other GPUs is due to this model having a L1 cache which allows data reuse even in algorithms that do no use the shared memory.

5 Comparison with previous proposals

The mapping of the wavelet transform on modern GPUs has been explored in several works. Tenllado *et al.* [17] present the implementation of five wavelet families comparing the actual performance of the filter bank and lifting schemes. The analysis is strictly focused in 2D using a strategy of row-column decomposition. The 3D graphics API OpenGL is used to organize data into streams, transfer those data streams to and from the GPU as 2D textures, upload kernels, and perform the sequence of kernel calls dictated by the application flow. A high level shading language, Cg, is used to code the fragment programs. In the filter bank scheme, the horizontal kernels read from the top half of the allocated texture and write into the bottom half, splitting the image into left-hand and right-hand boundaries and inner columns. The vertical filtering is analogous to the horizontal filtering but the image is divided into upper and lower boundaries and inner rows. Downsampling is performed by

Wavelet	[17]	[18]	SM
D4, FBS, 5 levels	9.12	-	8.53
D4, LS, 5 levels	17.9	-	9.67
(9,7), FBS, 5 levels	16.5	-	16.15
(9,7), LS, 5 levels	20.7	-	14.37
D4, FBS, 1 level	-	4.41	3.77
D4, LS, 1 level	-	8.05	6.02

Table 5: Execution times (in milliseconds) of 2D wavelet transforms of size 4 megapixels. The GPUs used are: 7800 GTX in [17], C870 in [18], and 8800 GTS in SM.

specifying input areas that are twice as large as the output ones. In the lifting scheme, every lifting step is performed by a different kernel, and the CPU main program chains and serializes these kernels to satisfy data dependencies.

Table 5 (column 2) shows the execution times obtained by these authors for the 2D wavelet transform using the lifting scheme (LS) and the filter bank scheme (FBS). In this same table (in column 4) we also include the times obtained by our implementation, although the execution conditions are not exactly the same. In [17] the experiments were performed on an Nvidia 7800 GTX (the most powerful model of the 7th generation GeForce by Nvidia) and the programming model is based on OpenGL and Cg. Since this generation of GPU does not support CUDA, the table shows the times obtained by our implementation on an Nvidia 8800 GTS (which is the second most powerful model of the 8th generation). The experiments consist of computing five levels of transformation on matrices of size 4 megapixels (2048×2048 in our case). The SM algorithm is slightly more efficient in the case of the filter-bank scheme while the efficiency is higher for the lifting scheme since, in this case, the SM method saves a greater number of accesses to the global memory.

Franco *et al.* [18] compute the 2D wavelet transform using the row-column decomposition with a matrix transposition as an intermediate step between the transformations of rows and columns. This solves the problem of non-coalesced accesses to the global memory during the transformation of the columns. They also carry out a comparison between the filter bank and lifting schemes. The implementation was done in CUDA using, in the case of the lifting scheme, a kernel for each lifting step. Table 5 (column 3) shows the execution times obtained by the authors on an Nvidia Tesla C870. It is a high-end GPU from the Nvidia G80 processor. We compare these execution times with those obtained by our implementation on an Nvidia 8800 GTS, which is a commodity card of the same generation with slightly lower performance. The experiments consist of one level of transformation on matrices of size 2048×2048.

There are not many implementations in the bibliography of tridiagonal systems solvers on the GPU using shared memory. An implementation of cyclic reduction on GPU is presented in [19], but this solution is designed to solve hundreds of tridiagonal systems in parallel with 512 equations at most. In the case of a single large system, the SM method, or some other similar technique, would need to be applied to divide the system among the blocks of threads and solve the data dependency problem at the block boundaries. Hence the results obtained by these authors are not comparable to those presented in this paper.

6 Conclusions

In this work we have presented a study of the split-and-merge method in the context of general purpose computation on GPU. In particular, we have considered the (9,7) wavelet (lifting version), the cyclic reduction method of resolution of tridiagonal systems, the D4 wavelet (filter bank version), and the packet D4 wavelet. These algorithms were selected because they represent a wide variety of multilevel algorithms: in-place and not-in-place, with and without decimation, pyramid-like, etc. Therefore, the method is general enough to be applicable to any algorithm of this type.

The method allows us to solve the data dependency problem at the block boundaries in the CUDA parallel programming model. In this way, we can efficiently exploit the thousands of threads running in parallel on the GPU and make extensive use of the shared memory, which is faster that the global one. In the study we have taken into account the different parameters that influence the performance of the algorithms on the GPU, such as the size of the split and merge sections, the use of the memory spaces and the reduction of the number of threads. We have also considered the extension of the method to the two-dimensional case. The comparison with previous works shows a reduction of the execution time of the algorithms associated with a suitable partition of the data in the shared memory of the GPU.

Furthermore, this method could be extended to compute a more general class of algorithms, including those which consist of several stages with data dependencies between them, efficiently on the GPU. In order to exploit the memory hierarchy of the GPU, it could be advantageous to begin computing the subsequent stages before completing the previous stages. Operations that cannot be performed with the available data within the thread blocks can be completed later, as long as the partially computed results are stored in the global memory of the GPU. The exploitation of shared memory (and in general, of the memory hierarchy) usually provides positive results on the GPU.

References

[1] Nvidia CUDA Programming Guide. Version 2.0, 2010.

[2] Nvidia CUDA Best Practices Guide. Version 3.0, 2010.

[3] R.C. Gonzalez and R.E. Woods, Digital Image Processing, Prentice Hall, 2007.

[4] T.J. Barth, T. Chan, and R. Haimes (Eds.), Multiscale and Multiresolution Methods: Theory and Applications, Springer, 2001.

[5] N.A. Dodgson, M.S. Floater, and M.A. Sabin (Eds.), Advances in Multiresolution for Geometric Modelling, Springer, 2004.

[6] S. Mallat, A Theory for Multiresolution Signal Decomposition: The Wavelet Representation, IEEE Trans. Pattern Anal. Machine Intell., 11 (1989) 674-693.

[7] S. Allmann, T. Rauber, and G. Runger, Cyclic Reduction on Distributed Shared Memory Machines, in Proc. 9th Euromicro Workshop on Parallel and Distributed Processing, (2001) 290-297.

[8] P. González, J.C. Cabaleiro, and T.F. Pena, Parallel Computation of Wavelet Transforms Using the Lifting Scheme, Journal of Supercomputing, 18 (2001) 141-152.

[9] W. Jiang and A. Ortega, Lifting Factorization-Based Discrete Wavelet Transform Architecture Design, IEEE Trans. Circuits and Systems for Video Technology, 11 (2001) 651-657.

[10] Pablo Quesada-Barriuso, J. Lamas-Rodríguez, D.B. Heras, M. Bóo, and F. Argüello, Selecting the Best Tridiagonal System Solver Protected on Multi-Core CPU and GPU platforms, Proceedings of PDPTA'11 - 17th Int'l Conference on Parallel and Distributed Processing Techniques and Applications, 2001.

[11] Press *et al.*, Numerical Recipes in C: the Art of Scientific Computing, Cambridge University Press, 1992.

[12] A. Cohen, I. Daubechies, and J.-C. Feauveau, Biorthogonal bases of compactly supported wavelets, Communications on Pure and Applied Mathematics, 45 (1992) 485-560.

[13] I. Daubechies and W. Sweldens, Factoring Wavelets Transforms into Lifting Steps, J. Fourier Anal. Appl., 4 (1998) 247-269.

[14] I. Daubechies, Ten Lectures on Wavelets, SIAM, 1992.

[15] F.G. Meyer, A.Z. Averbuch, and J.-O. Stromberg, Fast Adaptive Wavelet Packet Image Compression, IEEE Trans. Image Processing, 9 (2000) 792-800.

[16] P.E. Tikkanen, Nonlinear Wavelet and Wavelet Packet Denoising of Electrocardiogram Signal, Biological Cybernetics, 80 (1998) 259-267.

[17] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado, Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting, IEEE Trans. Parallel and Distributed Systems, 19 (2008) 299-310.

[18] J. Franco, G. Bernabe, J. Fernandez, M.E. Acacio, and M. Ujaldon, The GPU on the 2D Wavelet Transform. Survey and Contributions, in Proc. Para 2010: State of the Art in Scientific and Parallel Computing, Extended Abstract No. 264J, 2010.

[19] Y. Zhang, J. Cohen, and J.D. Owens, Fast Tridiagonal Solvers on the GPU, in Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of parallel programming, (2010) 127-136.

Author biographies

1. Francisco Argüello received the BS and PhD degrees in Physics from the University of Santiago, Spain in 1988 and 1992, respectively. He is currently an Associate Professor in the Department of Electronic and Computer Engineering at the University of Santiago de Compostela, Spain. His current research interests include signal and image processing, computer graphics, parallel and distributed computing, and quantum computing.

2. Dora B. Heras received a MS degree in Physics in 1994 and a PhD in 2000 from the University of Santiago de Compostela (Spain). She is currently an Associate Professor in the Department of Electronics and Computer Engineering at the same University. Her research interests include parallel programming for irregular codes, performance analysis and improvement, specially regarding the behavior of irregular codes on the memory hierarchy, and computer graphics for high performance computing.

3. Montserrat Bóo received the BS and PhD degrees in Physics from the University of Santiago de Compostela (Spain) in 1993 and 1997, respectively. Currently she is Associate Professor in the Department of Electronics and Computer Eng. at the University of Santiago de Compostela. Her interests are in the area of VLSI digital signal and image processing, computer graphics, GPGPU and computer arithmetic.

4. Julián Lamas-Rodríguez received his BS degree in Computer Engineering from the University of Coruña, Spain, and his MS in Videogame Creation from the University Pompeu-Fabra in Barcelona, Spain, both in 2006. In 2009 he joined the Computer Architecture Group of the Department of Electronics and Computer Science in the University of Santiago de Compostela, where he is currently pursuing the PhD degree in Information Technologies. The scope of his research is centered in high performance computing using graphics processors.