Int J Parallel Prog (2015) 43:1078–1102
DOI 10.1007/s10766-014-0323-8

BPLG: A Tuned Butterfly Processing Library for GPU Architectures

J. Lobeiras · M. Amor · R. Doallo

Received: 4 January 2014 / Accepted: 11 September 2014 / Published online: 26 September 2014

Abstract In order to increase the efficiency of existing software many works are incorporating *GPU* processing. However, despite the current advances in *GPU* languages and tools, taking advantage of their parallel architecture is still far more complex than programming standard multi-core *CPUs*. In this work, we present a library based on a set of building blocks that enable to easily design well-known algorithms with little effort. More specifically, we implement butterfly algorithms with this library, that is, a set of orthogonal signal transforms and an algorithm to solve tridiagonal equations systems. Thanks to the parametrization of the building blocks, the library can be easily tuned depending on the desired *GPU* architecture. This generic approach can be used to easily design these GPU algorithms while obtaining competitive performance on two recent NVIDIA GPU architectures, which results specially interesting from the productivity point of view.

Keywords Signal processing · FFT · DCT · Hartley · Tridiagonal equation system · GPGPU · CUDA · Tuned library

1 Introduction

Graphic Processing Unit (*GPU*) architectures are powerful parallel processors optimized for intensive arithmetic operations, performing specially well in regular

J. Lobeiras (✉) · M. Amor · R. Doallo
Computer Architecture Group (GAC), University of A Coruña (UDC), A Coruña, Spain
e-mail: jacobo.lobeiras@udc.es

M. Amor
e-mail: margarita.amor@udc.es

R. Doallo
e-mail: ramon.doallo@udc.es

algorithms with reduced flow control. Their major disadvantage is that even with standard languages, such as *OpenCL* [1] or *CUDA* [2], coding for *GPU* architectures tends to be more complex due to their special features and memory hierarchy. Developing efficient algorithms may be a challenge even for experienced programmers.

In this work we present a library for butterfly algorithms, Butterfly Processing Library for *GPUs* (BPLG), that is based on a set of building blocks for the construction of the different algorithms, allowing to reduce code complexity and development time. Specifically, our design makes extensive usage of template metaprogramming [3]. Template programming for *GPU* libraries have proved to be very useful in other works, for instance *Thrust* [4] or *Bolt* [5], giving a powerful set of tools to the programmer. Furthermore, the functions of this library can be tuned depending on various factors, such as the number of registers, the shared memory size or the desired parallelism level. Specifically, our library can be applied to support butterfly algorithms that can be represented using a butterfly communication pattern, where a problem of size N is solved in $log_R(N)$ steps and each step executes N/R parallel butterfly computations over R data elements.

Using this library is possible to design GPU algorithm such as signal transforms [(*Fast Fourier Transform* (*FFT*), *Hartley* transform and *Discrete Cosine Transform* (*DCT*)] or a tridiagonal solver algorithm, showing that our work obtains competitive performance with other libraries. The *FFT* is a very important operation for many applications, such as image and digital signal processing, filtering and compression, partial differential equation resolution or large number manipulation. A variant of the *FFT* algorithm was defined to work on real data [6]. This is useful in many fields like audio processing where it is known that the input signal will only take real values. The *Hartley Transform* [7] also operates on real data, but in contrast to the real *Fourier* transform which produces an output with complex data, the result will also be real. The *DCT* [8] is a widely used algorithm for multimedia processing and lossy compression which also works on real data. The resolution of tridiagonal equation systems is another interesting problem in scientific computing. It is used in many applications like fluid simulation or the *Alternating Direction Implicit* method for heat conduction and diffusion equations.

In this work two main contributions are included. The first one is the *CUDA* implementation of a compact yet very efficient tuned library using a *CPU*-like methodology based on a set of functions used as building blocks; its careful design allows to greatly reduce the complexity of the code without compromising performance. The second is the extension of the work proposed in [9] with the addition of a tridiagonal system solver algorithm based on the same design methodology as the signal transforms.

1.1 Related Work

Regarding the signal transforms, for *CPUs* there are many commonly used implementations, such as the *FFTW* [10], which is able to perform the four signal transforms. Other libraries, such as *Intel IPP* [11] or *Spiral* [12], do not include the *Hartley* transform. Many *GPU FFT* implementations have also appeared, for example [13,14] were designed for the initial *Tesla GPU* architecture and based on a

hierarchic communication scheme, *Nukada FFT* [15], optimized for 3D problems and multi-*GPU*, or [16], designed to execute large problems on clusters. However, these papers are mostly focused on the *FFT*, so their contributions may not be easily applicable to other algorithms. Another interesting approach are auto-tuning libraries, for instance [17] is designed for *CUDA* and explores a wide range of combinations, or *MPFFT* [18], an *OpenCL* library which is based on compiler technology. Maybe, the most well-known *GPU FFT* implementations are *NVIDIA's CUFFT* [19] and *AMD's clAmdFft* [20] included in the *APPML* library.

As far as we know, none of the *GPU* libraries cited so far directly supports the *DCT* or the *Hartley* transform. Many works [21,22] address specialized implementations of the *DCT* for small 2D blocks used in image or video processing, but only a few works [23,24] explain generalized *GPU* versions that cover a wider range of problem sizes.

With respect to resolution of tridiagonal systems, there are many well-known serial [25] and parallel [26,27] *CPU* algorithms. The interest of these problems also motivated the design of parallel *GPU* implementations. Some methods include cyclic reduction implementations [28], a derived version of the *SPIKE* algorithm [29], and other hybrid algorithms like [30] or [31] (which later was used in the *CUDPP* [32] library). *NVIDIA* also proposed its own tridiagonal solver, included in the *CUSPARSE* [33] library. Our work proposes a *GPU* solution based on the Wang and Mou algorithm [34], which has good numerical stability for diagonal dominant matrices or when no pivoting is needed. This algorithm offers excellent performance due to its suitability for *GPU* architectures, thanks to the regular structure based on a successive doubling method.

2 Butterfly Algorithms

Many parallel algorithms are based on a divide and conquer strategy, where the main problem is recursively subdivided until reaching the base case or a point that can be easily managed by the threads. Most common signal transforms can be decomposed this way to simplify the computations and distribute the work. For instance, one well-known case is the *Cooley–Tukey* [35] *FFT*. One interesting aspect of this algorithm is the data communication pattern (see Fig. 1a), which is also used in many other algorithms and is commonly described as a butterfly pattern. The *Cooley–Tukey FFT* performs a bit-reversal reordering at the beginning of the transform, but thanks to the flexibility in the communication pattern the permutation can be distributed among the stages of the algorithm. The signal transforms of this work will use a variable radix version of the *Stockham* [36] algorithm (see Fig. 1b), as it is more suited to the *GPU* architecture. Specifically, each transform of size $N = 2^n$ is computed in $log_R N$ stages using a radix-R algorithm which computes N/R butterfly operations per stage. If $m = N \ mod \ R \neq 0$ then a mixed-radix approach is used [6], computing a radix-m stage at the beginning.

There are other divide and conquer algorithms that use variations of these communication patterns besides the signal transforms, for instance a similar approach can be applied to the resolution of tridiagonal equation systems using the Wang and Mou

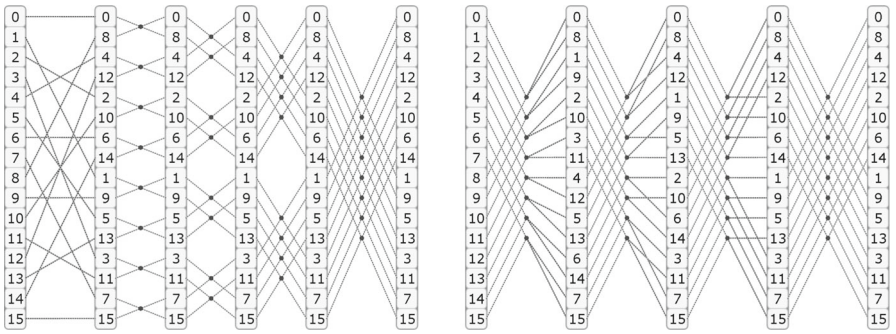

Fig. 1 Example of communication patterns in two FFT algorithms for radix-2 and N = 16. **a** Cooley–Tukey algorithm. **b** Stockham algorithm

algorithm [34]. The computation will be also divided in $log_R N$ stages, operating the rows in groups of R elements and following a pattern similar to the decimation in time *Cooley–Tukey* algorithm as seen in Fig. 1a, but excluding the initial bit-reversal stage. For instance, suppose the solution of a tridiagonal system $Ax = b$ composed of the following four equations is needed:

$$a_0x_0 + a_1x_1 + a_2x_2 = b_0, \quad a_0 = 0 \tag{1}$$
$$a_1x_1 + a_2x_2 + a_3x_3 = b_1 \tag{2}$$
$$a_2x_2 + a_3x_3 + a_4x_4 = b_2 \tag{3}$$
$$a_3x_3 + a_4x_4 + a_5x_5 = b_3, \quad a_5 = 0 \tag{4}$$

where a_i are known coefficients, x_i are the unknown variables and b_i are the independent terms. Notice that for N equations the first coefficient a_0 and the last coefficient a_{n+1} are always 0. In this case, instead of operating on signal data the butterfly is defined to operate on triads of equations labeled **L**eft, **C**enter and **R**ight. For convenience each equation is internally represented as a *float4* data type (each row has three unknowns, one per diagonal, plus the independent term). Figure 2a displays how the algorithm would handle the four equations to obtain the solution. Each box is one triad and the numbers inside the rows represent non-zero coefficients. Only the corresponding subindex of a_i is displayed in the figure, for instance Eq. 2 becomes [1 2 3] in C_2. Initially the members of each triad are initialized with the values of the corresponding equation, that is $L_i = C_i = R_i = Eq\ i$. Following the triads are operated pairwise until at the last stage $L_i = \{a_0, a_1, a_{n+1}, b_0\}$, $C_i = \{a_0, a_i, a_{n+1}, b_i\}$ and $R_i = \{a_0, a_n, a_{n+1}, b_n\}$. At this point the solution is computed as $x_i = b_i/a_i$, with $1 \leq i \leq N$.

Figure 2b show details of how each pair of triads is combined by the Wang and Mou algorithm. Each circle represents a reduction operation, where one equation is used to exchange one of the unknowns in another equation. The small vertical numbers present an example of how the algorithm would perform the last stage for $N = 8$, operating the second triad with the sixth triad. Notice that due to the recursive nature

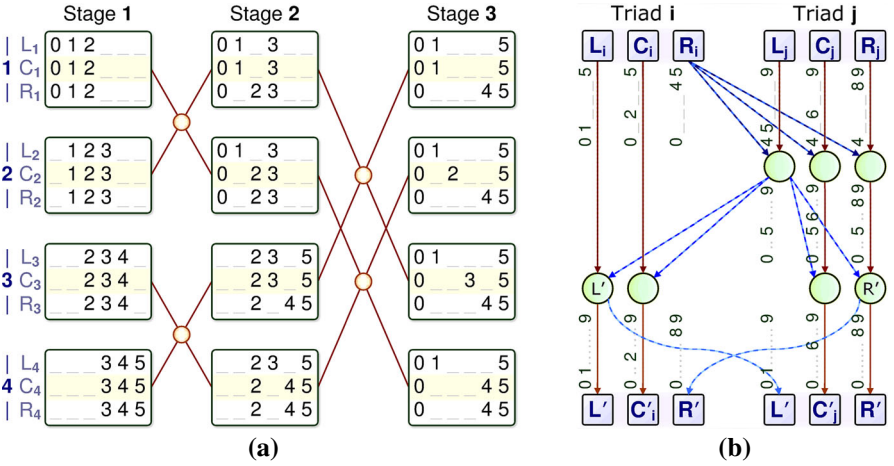

Fig. 2 Data communication pattern for the Wang and Mou tridiagonal algorithm. **a** Data exchange among triads for N = 4. **b** Detail of the butterfly operator

of the algorithm the left triad coincides with the second output from Fig. 2a. At the end of the sequence both left equations will be identical (see L'), the same happens with both right equations (see R').

3 BPLG Basic Functions

This section describes the basic functions that will be used to build our library. The building blocks of the algorithms are created in several layers, and each function only performs a small part of the work. Thanks to the behavior of templates many optimizations will take place at compile-time, reducing code complexity or avoiding temporal registers for function calls. Furthermore, more efficient code is generated using the additional information that is provided to the *CUDA nvcc* compiler about things like the problem size, the thread configuration or the radix-sequence. In fact, when this information is known at compile-time, private thread data reordering is performed using register renaming and data are directly accessed in the original structures instead of navigating through pointers. Another advantage is that array data can be processed using loops without being a major efficiency concern because static loops will be fully unrolled. In particular this avoids dynamic addressing of register arrays, which in the current *GPU* generations produces local memory spilling (see *CUDA C Best Practices Guide* [2], Section 6.2.3).

All the described functions were designed to operate in any space of the *GPU* memory hierarchy, but it is recommended to use data in registers for computations because they offer the highest bandwidth. However, when making heavy use of registers it is important to check the compiler statistics, because if too many registers are used the compiler will generate local memory spilling with the consequent performance loss. Due to this space limitation, to handle the processing of a large problem the input data has to be split in smaller chunks among the threads collaborating in each *CUDA*

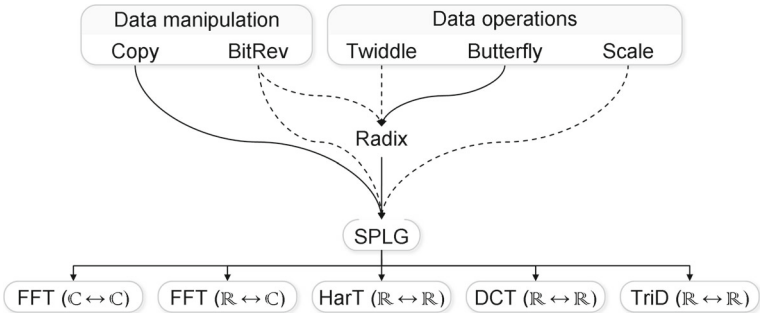

Fig. 3 Classification and module dependences of the building blocks involved in the library

block. Each thread usually operates over a subset of data stored in registers, therefore it should be aware of the relative position in the data input.

In summary, the proposed library is mainly composed of optimized high level blocks that implement the basic functions and one parametrized kernel for each algorithm that properly combines these blocks. The building blocks can be classified into computation (butterfly, twiddles or signal scaling) and reordering operations (bit-reversal or the family of strided copy operations). Figure 3 presents a scheme with the classification of the principal functions used to build the five algorithms: Complex *FFT*, Real *FFT*, Hartley Transform, *DCT* and Tridiagonal System Solver. The solid line indicates a dependency while the dashed line represents an optional component. Figure 4 presents the templates that will become the building blocks of our library. Size-dependent specializations and function overloads are omitted due to space constraints, however most of the code is present in the figure.

3.1 Reordering Blocks

Data are passed among the different blocks as pointers, without worrying about the number of modules cooperating. The blocks are combined in a final kernel with the appropriate code to manage the parallel work distribution, creating the sequences of the different algorithms.

Data is moved inside the *GPU* using the *Copy* function (see Fig. 4a), which reads an array X of N elements with stride ss and writes the result to a different array Y of the same size N but with stride sd: $Y\,[1:N:sd] := X\,[1:N:ss]$. Both data buffers can be pointers to arrays in global memory, shared memory or registers; the input buffer can also point to read-only constant memory or texture memory. The source X and destination Y do not have to reside in the same memory space, however it is recommended to minimize memory transfers, specially on the lower levels of the *GPU* memory hierarchy where bandwidth is a precious resource. The strides sd and ss are optional parameters and by default consecutive data is assumed. When the strides are known at compile time, the data offsets can also be precomputed. The caller function has the responsibility to use the adequate strides in order to minimize bank conflicts or to perform coalescent global memory access. Thanks to the C++ function overloading

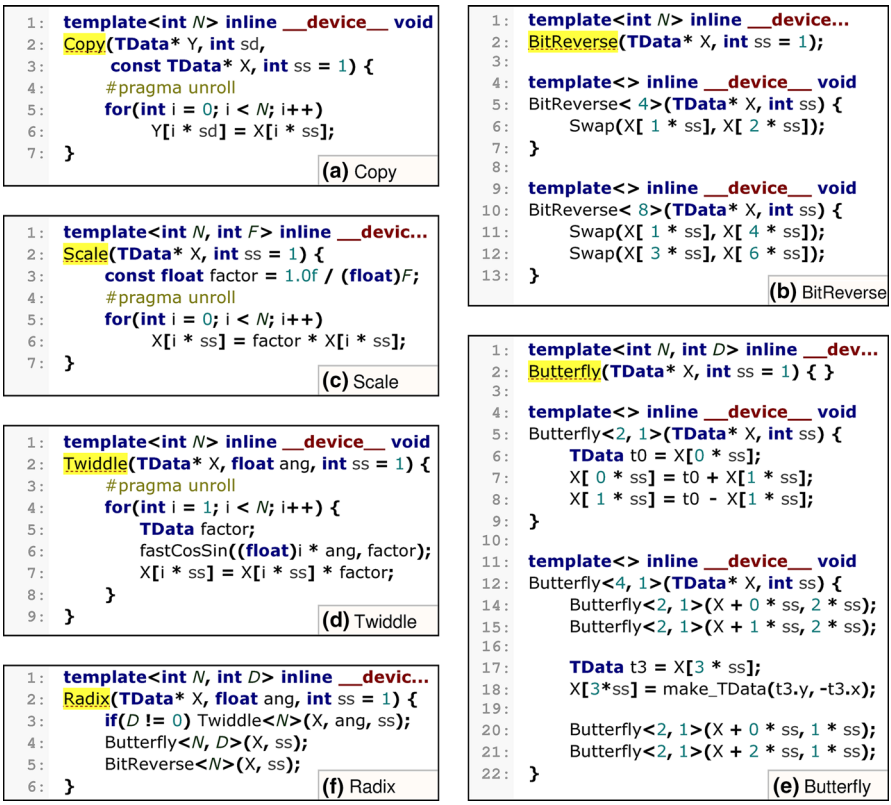

Fig. 4 Template code for each one of the building blocks used by the signal algorithms

mechanism it is possible to define source stride without explicitly specifying a value for the destination stride. Observe that the number of elements is known at compile time, thus, the unroll directive in line 4 will instruct the compiler to unroll the loop in line 5 and optimize the stride expression whenever possible.

The bit-reversal operation is a binary data permutation efficiently performed by the *BitReverse* function (see Fig. 4b). It reads an array of N elements with stride sd and writes the result to the same array using the same stride: $X[1:N:sd] := X[1:N:sd]$. For example, the function prototype (line 1) and the $N = 4$ (lines 4–7) and $N = 8$ (lines 9–13) specializations are displayed in the figure. The definition of a series of specializations usually results more efficient than the same operation performed by a size-independent generalized algorithm. Furthermore, if sd is known at compile-time the function performs static indexing and the source code will be optimized by the compiler into a simple register renaming.

3.2 Computing Blocks

In contrast to the reordering blocks, the computing blocks modify their input data. The different algorithms will be defined combining the basic reordering and computing

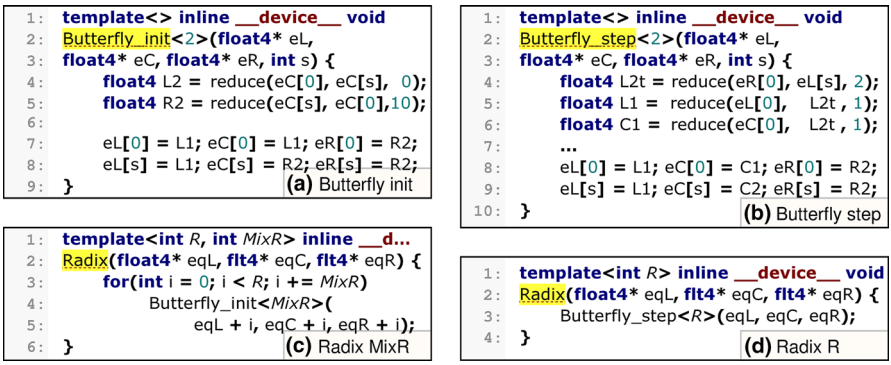

Fig. 5 Specialized template code for the tridiagonal solver algorithm

blocks. Note that small auxiliary functions may be required for some tasks, for instance, our *DCT* and *Hartley* implementations are derived from the complex *FFT*, therefore a pre-processing or post-processing filtering stage is required [37]. The real *FFT* is also a specialized version of the complex *FFT* algorithm that treats the real input signal as a complex array of half the length, avoiding redundant computations [37]. Last, tridiagonal systems have a similar algorithm structure, but use another data exchange pattern and a radically different radix operator as will be seen in Fig. 5.

If data scaling for inverse transform is desired the $Scale$ operator can be applied to the data array (see Fig. 4c). The input data (X) is multiplied by a scalar value, which is inversely proportional to the specified scaling factor F. In our case, the value of F is the complete problem size being processed, which is used in line 3 to calculate at compile-time the corresponding $factor$ value. Then, for each element of the array, the scaling factor is applied (see line 6) taking into account the optional stride parameter ss. As the number of iterations is known at compile-time the loop in line 5 will be fully unrolled. Furthermore, if the stride is known at compile time, the data offsets can also be precomputed.

The $Butterfly$ function handles the computations associated to each radix stage, receiving a data vector X and a stride value ss. The function prototype is indicated in Fig. 4d (lines 1 and 2), however the actual computations are performed by the corresponding specializations (in the example line 5 for $N = 2$ and line 12 for $N = 4$). Two sets of specializations are defined depending on the direction of the transform ($D = 1$ for forward transforms as in the figure, and $D = -1$ for inverse transforms). Observe that when $N > 2$ the operation is implemented splitting the input and calling the same operation again over each part (with a different data pointer X and stride ss) until reaching a two element butterfly. This is not a real recursion, because a different specialization is called. All the stride and pointer arithmetics involved in the $Butterfly$ function template will be resolved at compile-time, without any runtime performance penalty. In the case of tridiagonal systems, as detailed in Sect. 2, the butterfly operator is quite different. Figure 5 displays two examples of the corresponding code. $Butterfly\_step$ (Fig. 5b) is the general case and $Butterfly\_init$ (Fig. 5a) is a specialization optimized for the first stage of the algorithm, where the three equations

of each triad are equal. These functions work over equations, progressively reducing the variables as defined by the algorithm (see Fig. 2b). As the equations are represented by a single *float4* no information is provided about the relative position of the variables, therefore the *reduce* function receives a parameter to properly align the equation data for the reduction operation.

The twiddle factors are computed and applied by the corresponding $Twiddle$ function template (see Fig. 4e). It just multiplies the elements of the input array X accessed with a stride ss by a complex number. This value is derived from the location of the element being operated (the iteration index i, declared in line 4 and used in lines 6 and 7) and the twiddle angle ang that was specified when calling the function. As in other examples, the size parameter N is used to statically unroll the loop in line 4. In line 5 a complex variable is defined to hold the twiddle factor. The trigonometric computation is performed in line 6 using the iteration number, the value specified by ang and the $fastCosSin$ function (which will call the *CUDA* native $\_\_sincosf$ intrinsic).

Finally, the $Radix$ function (see Fig. 4f) is used to perform the computations of each stage. The parameter N determines the radix size, which is used to select the basic $Twiddle$, $Butterfly$ and $BitReverse$ kernels to call inside the template. When the transform direction D is 0 operations are disabled, thus only the bit-reversal (line 5) and other data permutation take place. As in the previous cases, the data is stored in the array X and will be accessed using the stride specified by ss. The optional ang parameter is only used to specify the corresponding twiddle for line 3 in multi-stage *FFTs* after the first stage. A mixed-radix overload is defined for those cases when the input array X contains data from different signals, which have to be processed in batch mode without interactions among them.

Figure 5 presents two examples of the $Radix$ function used by the tridiagonal algorithm. Tridiagonal systems do not use the bit-reversal operator nor the computation of the twiddle, however they still require the code to handle mixed-radix cases. This task is performed by the $MixR$ specialization (Fig. 5c), which will always perform the first stage of the algorithm, thus instead of calling the general $Butterfly\_step$ function (as in Fig. 5d) it will use $Butterfly\_init$.

4 Algorithm Design Based on BPLG

Thanks to the functions that were introduced in Sect. 3 it is possible to generate the code of the different algorithms in an easy and compact way. Each algorithm is implemented by a single main kernel, which is a template with four parameters: the problem size N, the transform direction D used in the orthogonal transforms, the radix size R and the amount of shared memory S. These parameters are known at compile-time, therefore conditional execution and many function calls involving offsets and strides can be easily optimized by the compiler as stated before. When $S > N$ each *CUDA* block will be assigned to process S/N independent problems in batch mode, therefore increasing performance for large quantities of small problems. Following we will describe two of the library kernels in order to explain the general form of the algorithms with more detail.

```
1:  template<int N, int D, int R, int S> __global__ void
2:  SPLG_DCT(float* src, int size) {
3:      // Define some thread/group identifiers and their memory offsets
4:      // Register/ShMem static allocation based on template parameters
5:
6:      // DCT pre-processing, loads data from GlbMem into registers
7:      if(D >= 0) {
8:          copy<R>(shm + threadXY, S / R, src + dctPos, S / R);
9:          __syncthreads();
10:         packDCT<N, R, D, S>(reg, shm, threadId, batchId);
11:     } else ...
12:
13:     // The first mixed-radix stage is always computed
14:     if(D < 0) scale<R, N/2>(reg); // Inverse transform scaling
15:     radix<R, MixR, D>(reg, R / MixR);
16:
17:     // This loop computes the remaining radix stages
18:     for(int accRad = MixR; accRad < N; accRad *= R) {
19:
20:         // Obtains strides and offsets from the iteration and thread id
21:         int readOffset  = ... , readStride = ... ;
22:         int writeOffset = ... , writeStride = ... ;
23:
24:         // Reordering stage in shared memory, Reg->Shm->Reg
25:         __syncthreads();
26:         copy<R>(shm + writeOffset, writeStride, reg);
27:         __syncthreads();
28:         copy<R>(reg, shm + readOffset, readStride);
29:
30:         // Computation with data in registers
31:         float ang = getAngle<D, R>(accRad, threadId >> cont);
32:         radix<R, D>(reg, ang);
33:     }
34:
35:     // DCT post-processing, stores the result from registers to ShMem
36:     __syncthreads();
37:     if(D >= 0) {
38:         radixDCT<N, R, D, S>(reg, threadId);
39:         copy<R>(shm + shmPos, N / R, reg);
40:     } else ...
41:
42:     // Stores the final result from ShMem to GlbMem
43:     __syncthreads();
44:     copy<R>(src + dctPos, S / R, shm + threadXY, S / R);
45: }
```

Fig. 6 Kernel code for the DCT algorithm

4.1 Signal Processing Transforms

As explained in Sect. 3 the real *FFT*, the *DCT* and the *Hartley* transform will be derived from the complex *FFT* algorithm using a pre-processing and/or post-processing stage [37]. In Fig. 6, the kernel code for the *DCT* algorithm is presented. This example will also allow us to display the usage of the functions that were introduced in Sect. 3.

The kernel has four template parameters which are supplied at compile-time (see N, D, R and S in line 1). The signal transforms are performed in-place and the kernel only requires two runtime parameters (see line 2): a data pointer src (used as both input and output) and $size$ (which is the actual signal size and can be used to add padding in order to process non power of two problems). Regarding the structure of the algorithm, it can be divided into six main sections:

(1) Initialization section (represented by lines 3 and 4), where thread and group identifiers are used to obtain the global memory offsets. In this section, the registers and shared memory resources are statically allocated as two simple arrays (shm and reg) based on the kernel template parameters.
(2) Pre-processing stage (lines 6–11), which loads data from global memory and may also perform some operations over it depending on the particular algorithm and the direction parameter. For instance, in the case of the forward *DCT* data has to be reordered before the execution of the radix stages ($packDCT$ in line 10), while in the case of the *Complex-FFT* data is loaded directly to registers with no pre-processing.
(3) First radix stage of the algorithm (lines 13–15). When $N \ mod \ R \neq 0$ a mixed-radix algorithm is performed. The optional template argument $MixR$ (computed at compile-time) is used to perform several independent radix operators of smaller size with data in the registers. This section also includes the signal scaling operation for the inverse transform ($scale$ in line 14).
(4) Remaining radix stages (lines 17–33). They are computed using a loop and each iteration is composed of a reordering stage (lines 24–28), which uses shared memory to exchange information among the threads, and a computing stage (lines 30–32). The data exchange is easily performed by the $copy$ function (lines 26 and 28) when called with different offsets and strides (obtained in lines 20–22). The computing stage is performed by the $radix$ function (line 32), which only requires the angle for the current iteration and thread (obtained in 31).
(5) Post-processing stage (lines 35–40), that depending on the algorithm may perform some computations or reorder the previous results.
(6) Results are written to global memory (lines 42–44). In the *DCT* signal data will reside in shared memory, while in the case of the *Complex-FFT* no post-processing is required, therefore data can be directly written from registers after the final radix stage (either line 15 or line 32, depending on N).

4.2 Tridiagonal System Algorithm

Figure 7 presents the code structure for tridiagonal systems, which shares many similarities with Fig. 6. Observe that in this case problem data is stored in a sparse format (see lines 2 and 3 with the function prototype), using four separate arrays as in *NVIDIA's CUSPARSE* library (*gtsvStridedBatch* function). There are three read-only buffers for the diagonals ($srcL$ for lower, $srcC$ for main and $srcR$ for upper) plus another read/write buffer ($dstX$) for the right-hand-side term, that will be also used to store the solution at the end of the kernel. The read only buffers are declared as *const __restrict* pointers, which in the *Kepler* architecture enables to optimize the memory

```
1:  template<int N, int D, int R, int S> __global__ void
2:  SPLG_TRI(const float* __restrict srcL, const float* __restrict srcC,
3:               const float* __restrict srcR, float* dstX, int stride)
4:  {
5:      // Define some thread/group identifiers and their memory offsets
6:      // Register/ShMem static allocation based on template parameters
7:      Float4 regL[R], regC[R], regR[R];
8:      __shared__ Float4 shm[N > R ? S : 1];
9:
10:     // Load data from GlbMem into registers
11:     switch(R) {
12:         case 2:       // Loads data using float2
13:             regC[0] = ... ; regC[1] = ... ; break;
14:         case 4:       // Loads data using float4
15:             regC[0]=...; regC[1]=...; regC[2]=...; regC[3]=...; break;
16:         default: ...  // Otherwise loads data for radix > 4
17:     }
18:
19:     // The first mixed-radix stage is always computed
20:     radix<R, MixR>(regL + i, regC + i, regR + i);
21:
22:     // The loop computes the remaining radix stages
23:     for(int accRad = MixR; accRad < N; accRad *= R) {
24:
25:         // Obtaines strides and offsets from the iteration and thread id
26:         int readOffset = ... , readStride = ... , readLo = ... , readHi = ... ;
27:         int writeOffset = ... , writeStride = ... ;
28:
29:         // Reordering stage in shared memory, Reg->Shm->Reg
30:         if(accRad > MixR) __syncthreads();
31:         copy<R>(shm + writeOffset, writeStride, regC);
32:         __syncthreads();
33:         copy<R>(regC, shm + readOffset, readStride);
34:         copy<R>(regL, shm + readLo, readStride);
35:         copy<R>(regR, shm + readHi, readStride);
36:
37:         // Computation with data in registers
38:         radix<R>(regL, regC, regR);
39:     }
40:
41:     // Stores the final result from registers to GlbMem
42:     #pragma unroll
43:     for(int i = 0; i < R; i++)
44:         dstX[glbOffset + glbStride * i] = regC[i].w / regC[i].y;
45: }
```

Fig. 7 Kernel code for the tridiagonal algorithm

access using texture cache. In contrast to signal processing algorithms there are no pre-processing or post-processing stages and the code can be divided into five main sections:

(1) Initialization section (lines 5–8). Thread and group identifiers are used to obtain global memory offsets. Register data (line 7) and shared memory space (line 8) are allocated for the equations. The equations are represented by the customized *Float4* data type, and each row requires three equations ($regL$, $regC$ and $regR$) to store the triads in the registers. However, to minimize shared memory usage, a single equation array shm will be allocated.

(2) Load data from global memory (lines 10–17). The algorithm can easily perform coalescent read operations to load data at the beginning of the algorithm without any shared memory reordering stage. Instead of accessing a single data element per memory request, we will use 64 bit loads for radix-2 (lines 12 and 13) to fetch 2 consecutive elements from each array, or 128 bit loads for radix-4 (lines 14 and 15), which provides four consecutive elements. These elements can be directly used as the input arguments of the first radix stage. Remember that initially the three equation of the triad will be equal, therefore only $regC$ needs to be initialized.

(3) First radix stage of the algorithm (lines 19–20). In a similar fashion to the signal processing algorithms, in order to reduce the number of processing stages (and consequently the number of intra-block synchronizations) it is possible to define a radix-R algorithm. The first radix stage is separated from the rest because when $N \ mod \ R \neq 0$ an initial mixed-radix $MixR$ stage is performed, furthermore, at the beginning the three equations from each row (left, center and right) will be identical. The different radix operators can be defined either recursively or using a specialization for the desired size, which helps reducing the number of private registers.

(4) Remaining radix stages (lines 22–39). Each iteration of the loop (line 23) reorganizes data (lines 29–35) and then performs a radix stage (line 38). In lines 26 and 27 the offset and strides for the data exchange are efficiently computed using bit masks, binary operators and displacements. Following, the data exchange is performed using shared memory.

An interesting optimization over the original algorithm proposed in [34] is to use only the center equation when performing the exchange. Each thread still requires to read $3 \times R$ equations (lines 33–35) to perform the radix-R stage, however it only needs to write R equations (line 31). This is based on a property of the algorithm, which relies on the fact that the left and right equations are equal to two of the center equations. More specifically, in stage i the left equation of row j can be obtained as $Eq_{left} \rightarrow 2^i \times \lfloor j/2^i \rfloor$ and the right equation $Eq_{right} \rightarrow 2^i \times (1 + \lfloor j/2^i \rfloor) - 1$. This way, the shared memory bandwidth required by the algorithm can be reduced.

(5) Results are written to global memory (lines 41–44). Due to the topology of the Cooley–Tukey algorithm when using decimation in time, coalescence will be good for equation systems where $N \geq 32 \times R$. Even for smaller problems coalescence will not be an issue thanks to the cache hierarchy of the *GPU*.

5 Obtaining of Tuning Parameters

One of the main requirements for *GPU* performance is to explicitly expose sufficient parallelism. The parallelism is controlled through the *CUDA* grid configuration (block parallelism) and block size (thread parallelism). In order to tune the kernel and configure the execution for optimal parallelism it is recommended to determine the main performance limiting factor. The most relevant factors are the number of registers assigned per thread, the shared memory per block, the desired number of concurrent blocks per *SM* (up to 16 in *Kepler* and up to 8 in *Fermi*), and last, the block size (up to 1,024 threads per block in both architectures). Blocks are not started until enough

resources are available, and once started they lock their resources until completion. The programmer can tweak the balance between the number of simultaneous blocks and the number of threads per block in order to offer the hardware enough independent instructions to accommodate multi-issue scheduling and enough tasks to take advantage of latency hiding techniques. Note that large block sizes may result in more expensive synchronizations, furthermore a high number of very light tasks may introduce some overhead. In fact, one of the most time consuming tasks in order to achieve high performance on the *GPU* is profiling and tuning the code to find the right balance among the resources.

5.1 Streaming Multiprocessor (SM) Parallelism

Regarding the *SM* parallelism limiting factor, suppose that B_{SM} is the number of blocks that can be processed simultaneously by each *SM*. It is given by the expression:

$$B_{SM} = Min(B^r_{SM}, B^s_{SM}, B^l_{SM}, B^{max}_{SM}) \tag{5}$$

where B^r_{SM} is the number of blocks limited by the registers available in the *SM* and how many are allocated for each block, and is computed as:

$$B^r_{SM} = R^{max}_{SM}/R_B, \text{ with } R_B = R_t \times L. \tag{6}$$

with R^{max}_{SM} the total number of registers per *SM* (65,536 in *Kepler* and 32,768 in *Fermi*) and R_B the number of registers used by each block, which is computed as the number of registers per task R_t multiplied by the number of tasks per block L (can be adjusted independently for each problem size). The second term of Expression 5 (B^s_{SM}) is the number of blocks limited by shared memory, computed as:

$$B^s_{SM} = S^{max}_{SM}/S_B, \text{ with } S_B = sizeof(D_t) \times R \times L \tag{7}$$

where S^{max}_{SM} is the size of the shared memory (49,152 bytes for both architectures) and S_B is the amount of shared memory reserved for each block, which is computed as the size of the data type (4 bytes for *float* data, 8 bytes for *complex* data and 16 bytes for *float4* used in tridiagonal equations) multiplied by the product of the number of registers R used to store signal data in each thread (depends on the desired radix size) and the number of tasks L created in each *CUDA* block. The third term of Expression 5 (B^l_{SM}) is the maximum number of blocks limited by the number of warps in-flight of the architecture:

$$B^l_{SM} = 32 \times L^{max}_{SM}/L \tag{8}$$

where L^{max}_{SM} is the *SM* warp limit of the architecture (64 for *Kepler* and 48 for *Fermi*). Finally, B^{max}_{SM} is the last term of Expression 5 and represents the *SM* block limit of the hardware, which is 16 for *Kepler* and 8 for *Fermi*.

5.2 Batch Execution in Order to Increase Parallelism

In our algorithms each thread performs the computations associated to a R butterfly in each stage, with data being stored in private registers. Thus, barring temporal data storage used by the compiler for any operations, our algorithm would require at least $sizeof(D_t) \times R$ bytes, where D_t is the data type used by the algorithm. Each problem is processed by $L_1 = N/R$ tasks and, in order to increase the thread parallelism, when L_1 is low the number of tasks per block can be increased using batch execution L_2. Therefore, each block will be composed of $L = L_1 \times L_2$ tasks. Although computations are entirely performed in registers, data interchanges within each block rely on shared memory, which should be large enough to fit the data stored in registers during the exchanges.

Depending on the data type, the problem size and the desired batch execution, the required shared memory will be $S_B = sizeof(D_t) \times N \times L_2$ bytes. The maximum size using real signal data is $N = 8{,}192$, for complex signals is $N = 4{,}096$, and for tridiagonal systems is $N = 2{,}048$. These limits are given by the maximum shared memory that can be allocated for a single block. Bigger problems would require a different approach, like a staggered data exchange or a multi-pass algorithm, which will be studied in a future work.

5.3 Simultaneous Block Processing Optimization

Our main objective in the optimization of the algorithm is to maximize B_{SM}, adjusting R and $L2$ to tune the algorithm for each problem size on each architecture. To illustrate our proposal, suppose that we are trying to optimize the processing of the complex *FFT* with $N = 128$ and $R = 4$ for the *Kepler* architecture. According to the compiler this kernel requires 28 registers, therefore $B^r_{SM} = 65{,}536/(28 \times L)$. On the other hand, $B^s_{SM} = 49{,}152/(sizeof(D_t) \times 4 \times L)$ and $B^l_{SM} = 32 \times 64/L$. In order to maximize these expressions, they can be rewritten assuming that $B^x_{SM} = 16$, which is the maximum allowed value given by B^{max}_{SM}. Consequently: $B^r_{SM} \rightarrow L = 65{,}536/(28 \times 16) = 146.3$, $B^s_{SM} \rightarrow L = 49{,}152/(8 \times 4 \times 16) = 96$, and $B^l_{SM} \rightarrow L = 32 \times 64/16 = 128$. Therefore, the recommended values for L are 96 or, rounding to the next power of two, 128 (in practice both configurations offer nearly identical results). Regarding the number of threads per block, $L = L1 \times L2$ and $L_1 = 128/4 = 32$, which is exactly one warp, therefore in this case $L2 = 128/32 = 4$, thus 4 warps will be working in batch mode processing different signals within each block and 12 blocks will be simultaneously processed by each *SM*. In the example, using $R = 4$ the algorithm will perform $n = log_4(128) = 3.5$ stages, which means one radix-2 stage followed by three radix-4 stages (the optimal R will be analyzed in Sect. 6).

For more information, the final configuration table for the complex *FFT* is presented in Table 1. Observe that for $N > 512$ B_{SM} keeps decreasing as more shared memory S_B is reserved for each block, however the performance impact is mitigated by the increase in thread parallelism L_1.

Table 1 Tuning configuration for the complex FFT algorithm

N	n	R	R_t	L_1	L_2	S_B	B_{SM}
4	2	2	14	2	64	2,048	16
8	3	2	18	4	32	2,048	16
16	4	2	18	8	16	2,048	16
32	5	2	18	16	8	2,048	16
64	3	4	28	16	8	4,096	12
128	3.5	4	28	32	4	4,096	12
256	4	4	27	64	2	4,096	12
512	4.5	4	28	128	1	4,096	12
1,024	5	4	27	256	1	8,192	6
2,048	$3.\overline{6}$	8	37	256	1	16,384	3
4,096	4	8	36	512	1	32,768	1

6 Experimental Results

Table 2 describes the test platforms that were used in this work. All the tests were evaluated in single precision using problem sizes in the range $N = \{4, \ldots, 4{,}096\}$. Batch execution is used to process $2^{24}/N$ different problems, therefore, therefore as the input signal increases the number of batch executions decreases. All the data resides on the *GPU* device memory at the beginning of each test, so there are no data transfers to *CPU* during the benchmarks to prevent interactions with other factors in the study. Our implementation makes extensive usage of shared memory for data exchange among the tasks, thus, the default *GPU* cache configuration with 48 *KB* of shared memory and 16 *KB* of *L1* cache was used.

The performance of the complex *FFT* will be expressed in *GFlops* through the commonly used expression: $5N \cdot log_2(N) \cdot b \cdot 10^{-9}/t$ [35], where N is the size of the input, b is the total number of signals processed and t is the time in seconds. A similar expression $2.5N \cdot log_2(N) \cdot b \cdot 10^{-9}/t$ can be used for the real *FFT*. As far as we know there is no standardized expression for the *DCT* and the *Hartley* transform, therefore to offer a similar measurement we will use the same formula as the real *FFT*.

Table 2 Description of the test platforms

	Platform 1	Platform 2
CPU	Core i7 2600	Core 2 Duo E8400
Memory	8 GB DDR3 1333	2 GB DDR3 1333
OS	Win7 x64 SP1	WinXP x64 SP2
Compiler	MSVC 2010 SP1	MSVC 2010 SP1
GPU	GeForce 580	GeForce Titan
Driver	v320.17, SDK 5.0	v320.17, SDK 5.0

6.1 Orthogonal Signal Transforms Performance

In this section we will analyze the global performance of the complex *FFT*, real *FFT*, *DCT* and *Hartley* transforms in the two *GPU* architectures. Obviously the more powerful *Kepler GPU* will offer better performance, but it is also interesting to check the scalability of the algorithm. For comparison purposes, *NVIDIA's CUFFT 5.0* results will be included in the figures. As the *CUFFT* does not support the *Hartley* and the *DCT* transforms, they were implemented using a similar strategy to *BPLG*. However, as we do not have access to the source code, this requires to launch at least two separate kernels, thus limiting the maximum performance. As far as we know there are no other general *GPU* implementations for these algorithms. The *DCT* included in the *CUDA SDK* is an specialized version for small 2D blocks with a fixed size of 8×8, and other publications [24] do not offer comparable benchmarks.

6.1.1 Balancing Warp and Block Parallelism

As mentioned in Sect. 5, it is important to configure the kernel for optimal *SM* usage. In fact, finding the right balance between thread-level and block-level parallelism usually results a time-consuming task for programmers. To facilitate the study of this factor Table 3 presents the complex *FFT* performance on *Platform 2* for three different radix configurations depending on the signal size N and the tasks per block L (due to space constraints only the more significant power of two values are represented). Unavailable configurations are shaded in gray, while the best cases are marked in bold. The first group shows the results for the radix-2 algorithm. Observe that with some exceptions $L = 128$ tends to offer the best performance. The next group repeats the analysis for radix-4. Excluding a few cases, the best configuration is now $L = 256$. Finally, the third group displays the results for radix-8. In this case, the best performance is usually obtained for $L = 64$, however the results do not outperform the radix-4 version. The only exceptions are the two last cases, $N = 2{,}048$ with $L = 256$ and $N = 4{,}096$ which requires $L = 512$.

Table 3 Impact of the task number for the FFT using Radix-2, Radix-4 and Radix-8 (Platform 2)

	Radix-2			Radix-4				Radix-8			
N	L64	L128	L256	L128	L256	L512	L1024	L64	L128	L256	L512
4	143.6	***143.7***	142.1	111.3	***111.8***	106.1	80.6				
8	211.6	***220.3***	220.2	171.3	***172.9***	155.4	117.0	***116.1***	115.9	114.9	108.0
16	222.8	266.6	***271.0***	***227.4***	224.4	207.4	170.3	***123.5***	123.1	121.0	98.1
32	273.9	***344.7***	343.0	302.5	***308.4***	299.4	203.2	***236.9***	235.7	228.5	175.9
64	369.2	***434.0***	432.4	438.7	***439.4***	432.9	265.4	***358.9***	353.1	339.0	266.5
128	316.5	***371.6***	370.4	511.9	***513.2***	495.1	283.8	***436.5***	418.7	395.8	248.1
256		***391.3***	391.1	584.2	***585.9***	562.0	325.7	***574.3***	560.1	519.3	309.9
512			***383.0***	***646.2***	628.5	576.1	324.9	***618.9***	596.0	562.8	343.5
1024					***701.9***	628.4	364.7		***663.6***	619.6	368.2
2048						***629.5***	360.5			***667.3***	401.3
4096							***396.0***				***436.1***

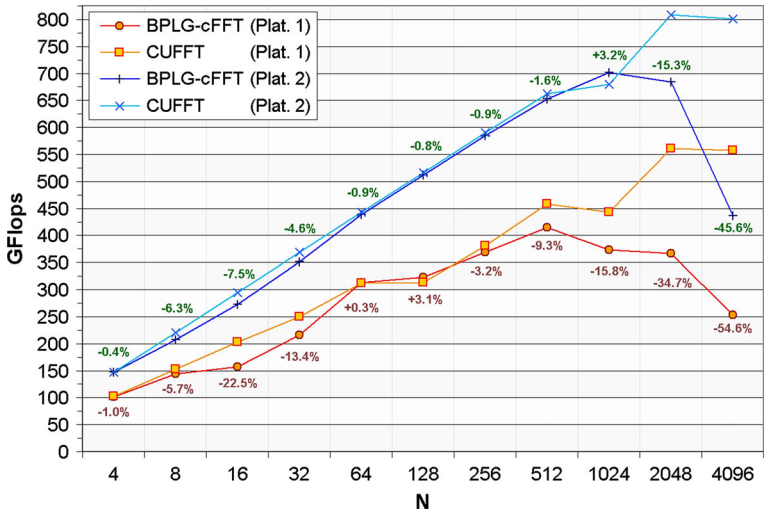

Fig. 8 Algorithm performance for the complex FFT algorithm

Comparing the results for the different radix values it can be observed that for $N = \{4 \ldots 32\}$ the best performance is obtained with the radix-2 version of the algorithm. For $N = \{64 \ldots 1{,}024\}$ the situation changes and radix-4 performs better. Finally, for $N = \{2{,}048, 4{,}096\}$ the radix-8 version comes ahead. According to our tests, going to a radix-16 configuration would not increase the performance.

6.1.2 Complex FFT Performance

Figure 8 shows the performance of the complex *FFT* on both platforms. As it can be observed, on Platform 2 our generic approach (*BPLG-cFFT*) offers very similar performance to the *CUFFT* for problem sizes up to $N = 1{,}024$ with 701.9 *GFlops*, while the *CUFFT* only offers 680.2 *GFlops*. For bigger problems the shared memory becomes the main limiting factor of our algorithm. Surprisingly, although the reduced complexity of the proposed algorithm, the average advantage of the *CUFFT* is only 7.3 %.

Our *BPLG-cFFT* algorithm is able to adapt quite well to the *Fermi* architecture of Platform 1, however in this case the *CUFFT* is very optimized and has more advantage (around 14.2 % on average). Nonetheless, in some cases our algorithm is able to overtake the *CUFFT*, for instance, for $N = 128$ we achieve 322.9 *GFlops*, while *NVIDIA's* implementation only offers 313.2 *GFlops*.

6.1.3 Real FFT Performance

Next, Fig. 9 analyzes the performance of the real *FFT*. Although the number of transforms per second is higher than the complex *FFT*, each transform performs fewer arithmetic operations, thus penalizing the *GFlop* estimation. The optimal radix is similar to the complex *FFT*, but as half the data is required it is displaced one location.

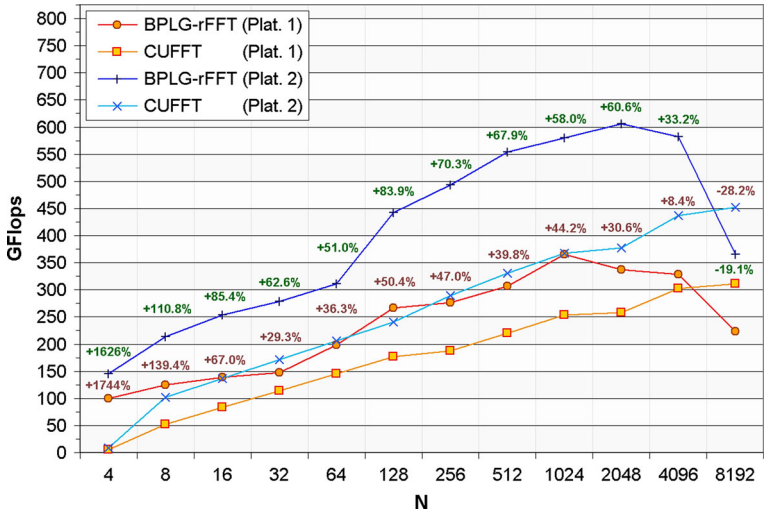

Fig. 9 Algorithm performance for the real FFT algorithm

For instance, while for $N = 2{,}048$ *BPLG-cFFT* performs better with radix-8, *BPLG-rFFT* would obtain the optimal behavior using radix-4. The performance scaling is not so proportional to the signal size, but is quite good, specially compared to the *CUFFT*. Excluding the outlier case for $N = 4$, the average improvement over the *CUFFT* on Platform 1 is on average 42.2 %, while on Platform 2 reaches a remarkable 60.4 %. Observe that our *BPLG-rFFT* algorithm executed on Platform 2 has very similar performance to the *CUFFT* executed on Platform 1, which is quite more powerful. According to *NVIDIA's* profiler the *CUFFT* is launching two separate kernels for each transform, therefore requiring twice the global memory bandwidth for the same signal size.

6.1.4 Discrete Cosine Transform

Figure 10 displays the test results for *DCT* algorithm on both platforms. The *CUFFT* results are also displayed, but recall that *NVIDIA's* library does not directly support this transform, therefore it is computed with the aid of a second filtering kernel, doubling the global memory bandwidth requirements. As expected *BPLG-DCT* outperforms the *CUFFT* version, moreover, in many cases it is able to offer more than twice the computation rate. On average our library is around 87.1 % faster on Platform 1 and 149.0 % faster on Platform 2. According to the profiler analysis, the pre-processing and post-processing stages introduce some overhead, and the shared memory access pattern generates more replays than the real *FFT*, which explains that lower performance. For instance, for $N = 1{,}024$ our *DCT* reaches 142.9 *GFlops* on Platform 2 and 346.3 *GFlops* on Platform 1, while in the real *FFT* transform *BPLG-rFFT* was able to obtain 365.9 *GFlops* and 580.2 *GFlops*, respectively.

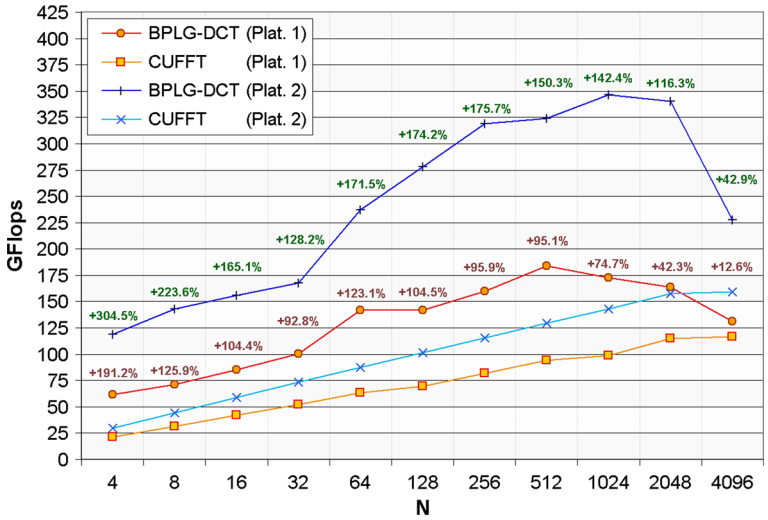

Fig. 10 Algorithm performance for the discrete cosine transform algorithm

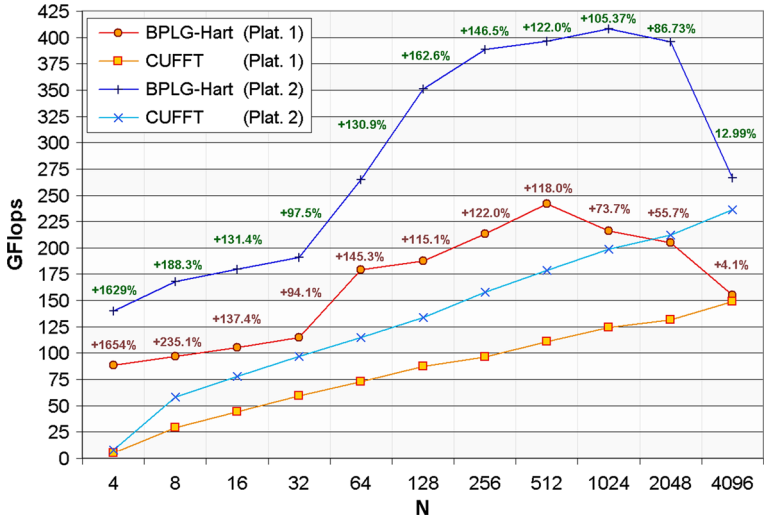

Fig. 11 Algorithm performance for the Hartley transform algorithm

6.1.5 Hartley Transform

Figure 11 presents the execution results of our library for the *Hartley* algorithm using both platforms. Once again the *CUFFT* version is computed using the *FFT* and filtering stage kernel, offering about half the performance. More specifically, on Platform 1 our library is on average 110 % faster than the *CUFFT*, while on Platform 2 our mean advantage is 118 % (in both cases excluding the outlier observed for $N = 4$). Notice that although the overall graphic outline is very similar to the *DCT*, both *GPU*

Table 4 Impact of the task number for tridiagonal systems using Radix-2 and Radix-4 for BPLG-TS (Platform 2)

	Radix-2						Radix-4				
N	L32	L64	***L128***	L256	L512	L1024	L32	L64	L128	L256	L512
4	6003	9272	***9610***	9512	9228	5511	7635	***9314***	6326	6365	6478
8	5459	7412	***7568***	7535	7072	4433	5724	***6509***	6468	6216	4098
16	3887	5367	***5479***	5474	5231	3634	5390	***7396***	7299	7094	4463
32	3194	4670	***4786***	4700	4550	3257	5545	***6744***	6641	6265	4264
64	3252	5289	***5656***	5431	5014	3384	5292	***6993***	6845	6353	3870
128		4652	***5013***	4872	4061	3054	4666	***5838***	5721	5288	4568
256			***4501***	4037	3677	2777		***6030***	5768	5420	3195
512				***3582***	3348	2551			***4522***	4482	3072
1024					***3092***	2350				***4332***	2814
2048						***2179***					***2693***

libraries obtain better results in the *Hartley* transform. For instance, for $N = 1{,}024$ *BPLG-Hart* obtains 216.2 *GFlops* on Platform 1 and 408.1 *GFlops* on Platform 2. The reason behind this is the shared memory access pattern of the filtering stage, which is simpler in the *Hartley* transform.

6.2 Tridiagonal Equation Systems Performance Analysis

The performance of the tridiagonal solver will be measured in million rows per second, using the formula $N \cdot b \cdot 10^{-6}/t$, where N is the number of single-precision equations per tridiagonal system, b is the total number of problems processed and t is the time in seconds. In our tests the batch size was defined as $b = 2^{22}/N$, therefore all the tests will process the same number of rows. Two reference points are provided: one is *NVIDIA's CUSPARSE* library (v5.0) and the other is the algorithm presented in [31], which is available as part of the *CUDPP* library (v2.1).

6.2.1 Balancing Warp and Block Parallelism

Once again it is important to find the optimal balance between thread-level parallelism and block-level parallelism to maximize *SM* utilization. Table 4 shows the tridiagonal system resolution performance of *BPLG-TS* on *Platform 2* depending on the radix size R, the problem size N and the number of tasks per block L. Unavailable configurations are shaded in gray, while the best cases are marked in bold.

Observe that $L = 128$ tends to offer better performance for radix-2, only the two first system sizes ($N = 4$ and $N = 8$) perform faster with $L = 64$. If R is increased to radix-4 $L = 64$ is the preferred option, though $L = 128$ is competitive at the beginning, as size increases the gap between $L = 64$ and $L = 128$ becomes wider. In this case, radix-8 results are not provided because the algorithm requires too many registers, which incurs in local memory spilling and the consequent performance degradation. Notice that there is a similar resource balance when using radix-2 and $L = 128$ compared to radix-4 and $L = 64$, as both process 256 equations per block. Nonetheless, radix-4 is usually better because it minimizes shared memory exchanges

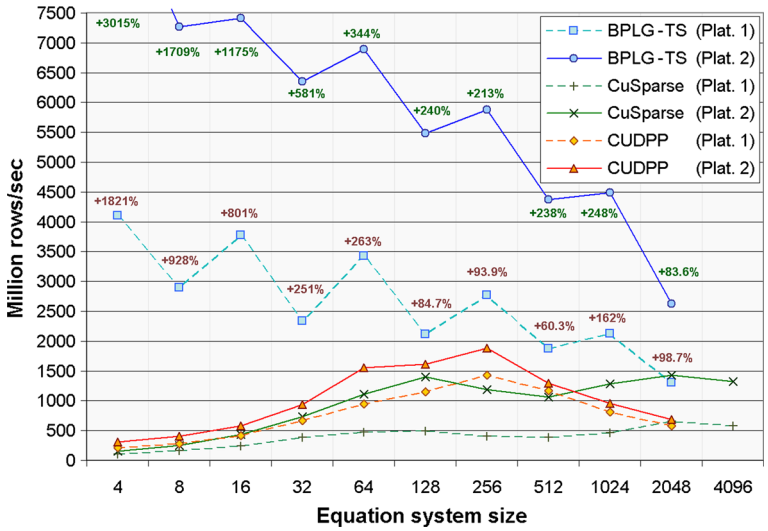

Fig. 12 Algorithm performance for tridiagonal equation system resolution

and synchronizations, which are more expensive compared to the signal processing algorithms because more data is exchanged in each stage.

6.2.2 Tridiagonal System Resolution Performance

As it can be seen in the Fig. 12, *BPLG-TS* implementation offers excellent performance compared with other two state-of-the-art solutions. The batch support provides a very good startup, which makes the library suitable for small problems (up to 9,609.8 Mrows/s for $N = 4$ running on Platform 2). The percentages near the lines of *BPLG-TS* represent the performance improvement over the fastest one from the other two implementations (*CUDPP* or *CUSPARSE*) running on the same platform. Even in the worst case *BPLG-TS* achieves over an 83.6 % advantage over *CUSPARSE*. Furthermore, *BPLG-TS* executed on Platform 1 is usually faster than the other two running on Platform 2. The jagged outline observed in $N = 32$ or $N = 128$, is due to the mixed-radix stage. Performance tends to decrease with the number of stages because more synchronizations are required and the shared memory becomes a scarce resource, which reduces *SM* block parallelism. As in the case of the signal transforms, for equation systems where $N > 2{,}048$ a different approach would be required, for instance using a multi-kernel algorithm. This will be considered on future version.

Regarding the other two libraries, *CUSPARSE* and *CUDPP*, the observed performance difference between the two *GPU* architectures is smaller, specially in the case of *CUDPP*. Surprisingly, *NVIDIA's* own library usually offers the worst performance. Profiling *CUSPARSE* reveals that it is launching several kernels to solve the batch of problems, therefore the global memory bandwidth becomes a limiting factor. Only for the larger problems it is able to amortize the cost of the multi-kernel approach, which hints that it was probably designed with large equation systems in mind. Last,

regarding *CUDPP*, at the beginning is quite similar to *CUSPARSE*, however as the problem size increases it gets better, being able to improve *NVIDIA's* library performance. Nonetheless, for $N > 512$ it becomes slower. This behavior is easily explained by the fact that each system is assigned to a single *CUDA* block. For small systems the kernel will be launched with a single warp per block, furthermore some of the threads will be idle. For larger problems the shared memory becomes a limiting factor.

7 Conclusions

In this work, we designed a library based on a set of functions which enabled the implementation of several well-known butterfly algorithms for *CUDA GPUs*, namely the complex and real version of the *FFT*, the *DCT* and the *Hartley* signal transform algorithms, as well as a tridiagonal equation system solver. The resulting implementations were tested on different platforms, comparing the efficiency with other state of the art libraries. Our approach provides excellent performance in many cases, an average 60.4 % advantage in the real *FFT* or over 200 % advantage for tridiagonal equation systems, and as far as we know it is also the fastest general purpose implementation of the *DCT* and the *Hartley* transforms. Regarding the complex *FFT*, *BPLG* still offers competitive results with respect to the *CUFFT*, especially considering that our primary focus on this work has been to provide a library flexible enough to improve *GPU* programmability.

One of the most interesting features of the proposed library is the modular design based on small building blocks. The building blocks of the algorithms where implemented using high-level *C++* templates, with emphasis on flexibility and configurability. The library parameters were adjusted to obtain good efficiency on two recent *GPU* architectures.

There are many interesting topics as future work, one of the most promising topics is to address the design and optimization of these orthogonal transforms based on a series of formal algebraic operators, which will control the data mapping to the hardware resources. We also plan to extend the algorithms for bigger signals and enable support for other data types and 2D transforms.

Acknowledgments This research has been supported by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Reference Groups, cofunded by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P)

References

1. Khronos OpenCL Working Group: The OpenCL specification (2011)
2. NVIDIA: CUDA C Best Practices Guide (SDK Document.). V5.0. (2012)
3. Vandevoorde, D., Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley, Boston (2002)
4. Bell, N.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems, Jade Edition. Morgan Kaufmann (2011)
5. Sander B.,: Bolt: A C++ Templater Library for HSA. Presented in AMD Fusion Developer Summit '12 (2012)

6. Chu, E., George, A.: Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms. Computational Mathematics Series. CRC Press, Boca Raton (2000)
7. Hartley, R.V.L.: A more symmetrical Fourier analysis applied to transmission problems. In: Proc. of the Institute of Radio Engineers (IRE), vol. 30(3), pp. 144–150 (1942)
8. Ahmed, N., Natarajan, T., Rao, K.R.: Discrete cosine transform. IEEE Trans. Comput. **C–23**(1), 90–93 (1974)
9. Lobeiras, J., Amor, M., Doallo, R.: SPLG: a tuned signal processing library for GPU architectures. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2013), vol. 1, pp. 184–191 (2013)
10. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005)
11. Intel: Intel Integrated Performance Primitives for Intel Architecture, Reference Manual. Volume 1: Signal Processing (2012)
12. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: code generation for DSP transforms. In: Proc. of the IEEE, on "Program Generation, Optimization, and Platform Adaptation", vol. 93(2), pp. 232–275 (2005)
13. Govindaraju, N., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High performance discrete Fourier transforms on graphics processors. In: Proc. of the 2008 ACM/IEEE conference on Supercomputing (SC '08), pp. 2:1–2:12. IEEE Press (2008)
14. Volkov, V., Kazian, B.: Fitting FFT onto the G80 Architecture. University of California, Berkeley, Tech. rep. (2009)
15. Nukada, A., Matsuoka, S.: Auto-tuning 3-D FFT Library for CUDA GPUs. In: Proc. of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), pp. 1–10 (2009)
16. Chen, Y., Cui, X., Mei, H.: Large-scale FFT on GPU clusters. In: ICS '10: Proc. of the 24th ACM Intl. Conference on Supercomputing, pp. 315–324 (2010)
17. Dotsenko, Y., Baghsorkhi, S.S., Lloyd, B., Govindaraju, N.K.: Auto-tuning of fast Fourier transform on graphics processors. In: Principles and Practice of Parallel Programming (PPoPP '11), pp. 257–266 (2011)
18. Li, Y., Zhang, Y., Liu, Y., Long, G., Jia, H.: MPFFT: an auto-tuning FFT library for OpenCL GPUs. J. Comput. Sci. Technol. **28**(1), 90–105 (2013)
19. NVIDIA: CUDA CUFFT Library. V5.0. (2012)
20. AMD: AMD Math Libraries, OpenCL Fast Fourier Transform (clAmdFft) (2012)
21. Wang B., Álvarez-Mesa M., Ching C., Juurlink B.: An optimized parallel IDCT on graphics processing units. In: 18th International Conference on Parallel Processing Workshops (EuroPar '12), pp. 155–164. Springer, Berlin (2013)
22. Guptda, M., Garg, A.K.: Analysis of image compression algorithm using DCT. Int. J. Eng. Res. Appl. (IJERA) **2**(1), 512–521 (2012)
23. Kim, C.G., Choi, Y.S.: A high performance parallel DCT with OpenCL on heterogeneous computing environment. Multimed. Tools Appl. **64**(2), 475–489 (2013)
24. Panella, M., Basset, L.: An efficient GPU implementation of modified discrete cosine transform using CUDA. Int. J. Comput. Sci. Inf. Secur. **10**(5), 23–30 (2012)
25. Thomas, L.H.: Elliptic Problems in Linear Differential Equations over a Network. Columbia University, Tech. rep. (1949)
26. Polizzi, E., Sameh, A.H.: A parallel hybrid banded system solver: the SPIKE algorithm. Parallel Comput. **32**(2), 177–194 (2006)
27. Intel: Intel Math Kernel Library, Reference Manual. V10.2. (2009)
28. Göddeke, D., Strzodka, R.: Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. IEEE Trans. Parallel Distrib. Syst. (TPDS) **22**(1), 22–32 (2011). (Special Issue on HPC with Accelerators)
29. Chang, L.-W., Stratton, J.A., Kim, H.-S., Hwu, W.W.: A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12), pp. 27:1–27:11. IEEE Computer Society Press (2012)
30. Kim, H.-S., Wu, S., Chang, L.-W., Hwu, W.W.: A scalable tridiagonal solver for GPU. In: Intl. Conf. on Parallel Processing, pp. 444–453. IEEE Comp. Society (2011)

31. Zhang, Y., Cohen, J., Owens, J.D.: Fast tridiagonal solvers on the GPU. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), pp. 127–136 (2010)
32. CUDA Data Parallel Primitives Library. V2.1. (2013)
33. NVIDIA: CUSPARSE Library. V5.0. (2012)
34. Wang, X., Mou, Z.G.: A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In: IEEE Symposium on Parallel and Distributed Processing, pp. 810–817 (1991)
35. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Math. Comput. **19**(90), 297–301 (1965)
36. Stockham, T.G.: High-speed convolution and correlation. In: Proceedings of the Spring Joint Computer Conference, pp. 229–233 (1966)
37. Keith, J.: The Regularized Fast Hartley Transform. Signals and Communication Technology. Springer, Berlin (2010)