Portable and efficient FFT and DCT algorithms with the Heterogeneous Butterfly Processing Library

Sergio Vázquez[a,b,*], Margarita Amor[b], Basilio B. Fraguela[b]

[a] *CITIC, Universidade da Coruña, Spain*
[b] *Grupo de Arquitectura de Computadores, Universidade da Coruña, Spain*

Abstract

The existence of a wide variety of computing devices with very different properties makes essential the development of software that is not only portable among them, but which also adapts to the properties of each platform. In this paper, we present the Heterogeneous Butterfly Processing Library (HBPL), which provides optimized portable kernels for problems of small sizes that allow using orthogonal transform algorithms such as the FFT and DCT on different accelerators and regular CPUs. Our library is implemented on the OpenCL standard, which provides portability on a large number of platforms. Furthermore, high performance is achieved on a wide range of devices by exploiting run-time code generation and metaprogramming guided by a parametrization strategy. An exhaustive evaluation on different platforms shows that our proposal obtains competitive or better performance than related libraries.

Keywords: Signal processing, tuned library, Open Computing Language (OpenCL), Heterogeneous platform, GPUs.

1. Introduction

Accelerators, such as the Graphic Processing Units (GPUs) or the Intel®Xeon Phi™, are powerful parallel processors that can offer a great performance and

*Corresponding author

Email addresses: sergio.vazquez@udc.es (Sergio Vázquez), margartita.amor@udc.es (Margarita Amor), basilio.fraguela@udc.es (Basilio B. Fraguela)

Postprint submitted to Journal of Parallel and Distributed Computing (JPDC)October 23, 2018

therefore are increasingly used for scientific computing and engineering. However, such performance requires a very careful and much more complex programming than that of standard processors, as these devices have many more hardware control elements, such as various types of memory and more levels of parallelism. This way, the existence of libraries that provide implementations of important algorithms that are tunable according to the characteristics of each device and offer a large degree of portability is critical for the exploitation of these systems. In this line, this paper presents the Heterogeneous Butterfly Processing Library (HBPL), which provides portable implementations of butterfly algorithms [1] that can be used both on regular CPUs as well as on a wide range of accelerators. Importantly, the portability offered by HBPL is not only functional, but also in terms of performance, as it provides parameterization mechanisms that allow to adapt the underlying code to each architecture at hand. Our work is a generalization and a natural evolution of [2], which proposed the Butterfly Processing Library for GPUs (BPLG), a library focused on the same algorithms, but restricted to NVidia GPUs due to being developed on CUDA. BPLG achieves high performance for Butterfly algorithms and it is based on a set of tuned blocks developed using template metaprogramming that operate on limited size problems that fit directly into the shared memory of GPUs. These algorithms are based on a divide-and-conquer strategy, where the problem is recursively subdivided into subproblems until a base case. This way the calculations are simplified and the work is distributed. One example is the Cooley-Tukey FFT [3], which uses a data communication pattern known as butterfly computation [1].

In [4] a tuning strategy that addresses the design of algorithms for large problem sizes was proposed. In order to provide functional portability, our new HBPL library relies on the ubiquitous OpenCL heterogeneous computing standard [5]. OpenCL provides a common programming framework that is supported by a wide range of devices and vendors, such as regular CPUs, GPUs, FPGAs and DSPs. Unfortunately, it does not provide performance portability, the consequence being that a given code written in OpenCL can be very

well suited for some architectures and thus provide very good performance on them, while it can provide a extremely poor performance in other systems. The performance portability of our proposal is achieved by means of a structured approach that develops complex algorithms as combinations of simpler individual blocks. Each basic block performs very concrete tasks and its code is generated under the control of a series of parameters that can be tuned for each architecture in order to provide the best performance. A common approach to control code generation used in [2] is template metaprogramming [6], which is straightforward in CUDA, since it has been available in that framework for years. OpenCL, however, suffers from a slower evolution than CUDA, which is not surprising as it is an standard on which many parties have to agree in order to approve new versions of it. This way, templates are supported in OpenCL only after the approval of the 2.2 version of the standard in May 2017, and as of today no implementations conform to this version of the standard [7].

Fortunately, the research community has developed many tools that enhance the usage of OpenCL. Of particular interest from the point of view of metaprogramming and thus the development of HBPL on OpenCL, is the Heterogeneous Programming Library (HPL[1]) [8]. This is a C++ framework focused on the programming of heterogeneous systems enabling portability in a simple way. While HPL supports the usage of regular OpenCL C kernels [9], just simplifying the host-side of the application, it also provides a language embedded in C++ that is translated at runtime into OpenCL C kernels. This latter possibility not only allows the development of programs that include the host and device code in the same source file, but it also enables key metaprogramming possibilities. Namely, HPL allows to use templates in the embedded language kernels, their outcome being reflected in the OpenCL code generated. Moreover, HPL allows to insert standard C++ statements and expressions in these kernels, which enables the insertion of runtime computed values and the selection of the code to generate in the underlying OpenCL kernels. Both mechanisms have proved to be very

[1] Available at http://hpl.des.udc.es

useful for the development of performance portable codes on top of OpenCL in the existing implementations of the standard [10]. This way our HBPL library relies on HPL as an intermediate layer on top of OpenCL that supports the metaprogramming mechanism required to generate the code of our kernels under the control of tunable parameters.

The main algorithms provided by HBPL are widely used orthogonal transformations such as the Fast Fourier Transform (FFT) [3] and the Discrete Cosine Transform (DCT) [11]. The library is designed for problems sizes that can be stored in the local memory, as most FFTs and DCTs used in many fields are small. **The FFT is a very important operation that is used in the processing of digital images and signals, filters, compression, equations resolution in partial derivatives or convolutions. The DCT is an algorithm that works on real signals and is widely used in the field of multimedia processing. In both cases, the library supports the use of two-dimensional square transforms.**

The rest of this paper is structured as follows. **The related work is discussed in the next Section, which is followed by an introduction to the basics of the HBPL in Section 3.** Then, Section 4 explains the construction of signal transform algorithms on top of HBPL, while Section 5 explains the parameterization of the library. This is followed by an evaluation in Section 6 and our conclusions and future work proposals in Section 7.

2. Related work

There are many efficient proposals for the implementation of the FFT algorithm in CPUs such as FFTW [12], Intel IPP [13] or Spiral [14, 15, 16]. With respect to GPU proposals, there are an interesting number of auto-tuning solutions, which achieve high performance. For instance, in [17] a wide range of combinations are generated and the best solution is selected. An auto-tuning methodology for

OpenCL based on compiler technology is proposed in MPFFT [18]. The first stage generates FFT code for arbitrary size; and, the second stage uses dynamic programming to evaluate the tree that represents possible factorizations for computing FFT. In [19], an approach for 3D problems and multi-GPU is presented using auto-tuning, but this approach does not explicitly consider some main performance factors, such as the right balance between the high number simultaneous task and the proper utilisation of shared resources. An alternative solution based on a small number of efficient parametrisable kernels that allows the generation of efficiente CUDA FFT kernels using only three parameters is presented in [20]. Approaches focused on large 1D FFT on a single coprocessor include [21, 22, 4]. Another proposal for solving FFT in a sparse format is presented in [23]. Maybe, the most well-known GPU FFT implementations are NVidia's *cuFFT* [24] and *clFFT* [25], included in the clMathLibraries originated from AMD.

The DCT algorithm, thanks to its ability to compress power (predominantly part of the signals for its reconstruction), is very used in multimedia algorithms of lossless compression [26, 27, 28], such as the JPEG image format [29], compression of audio in MP3 format [30] or compression of video in MPEG format [31]. There are few libraries for GPUs that support the DCT transform directly. One of them is BPLG [2], the library in which this work relies. Also, [32] explains a general release for GPUs that covers problems of a wide size range, while there are other proposals [33, 34] designed for small 2D blocks that are used in image or video processing.

3. HBPL

This paper presents the HBPL library, composed by the orthogonal transform algorithms FFT and DCT. As BPLG, HBPL is mainly composed of a

series of optimized low-level blocks, which implement the basic functions, and of a series of parameterized kernels that combine these blocks correctly for each algorithm and platform.

The construction blocks that form the algorithms provided by our library can be classified into two types: computing blocks (Butterfly, Twiddle, Radix), which perform arithmetic operations on data, and reordering blocks (Copy), which make changes in the position of storage of the data. These optimized blocks are implemented in our library as fuction templates that exploit metaprogramming. As mentioned above, these blocks are used for the creation of parametrizable kernels with the appropriate code to handle the distribution of the parallel work according to the parameters presented in Section 5. The use of template metaprogramming allows to perform optimizations at compile time [35]. In this way, the number of temporal records required for the called functions and the code complexity are reduced.

This library is focused on a set of parallel prefix algorithms, denoted as Butterfly algorithms. This is a class of algorithms that solve a problem of size N in $\log_r(N)$ steps that are applied in sequence. Each step is composed of N/r independent and thus parallel computations, each one of such computations being in charge of the processing of r data elements, this value r being called radix. As we can see, in each step N elements are processed, since N/r computations of size r are applied in parallel. The rationale for the existence of the $\log_r(N)$ steps is to achieve that each final output of the destination depends on all of the initial N input elements, which happens in many algorithms such as the FFT. At a rate of r elements processed together per step, this requires a minimum of $\log_r(N)$ steps. In addition, each stage must be connected to the next one using a different regular pattern so that after all the stages are performed the aforementioned all-to-all property holds. The pattern of each communication can be algorithmically represented using a binary notation for the indices of the elements used in each stage and connecting indices obtained by manipulating (in this case inverting) different subsets of bits in each stage. These algorithms may be depicted by a directed acyclic oriented graph called prefix circuit [36]. Given

a radix r this graph represents the r-size computations by small circles called Butterfly Nodes, while the r input data processed by each node are represented by lines that converge in it, and the r results appear as lines departing from the node. There are many examples of algorithms of this type, such as signal orthogonal transforms or some tridiagonal system solvers. Our FFT proposal is based on the Stockham algorithm [37]. Figure 1 illustrates the pattern of this algorithm with $N = 16$ using radix 2. Each constructed block in the figure is associated to a part of the algorithm represented by one of the construction blocks quoted above and which will be discussed below for the FFT. The 16 elements of the input are divided into portions of 4 elements that are calculated with the FFT. Then, the result is multiplied by the rotation factor. HBPL functions have been designed to work at any level of the memory hierarchy. For the realization of the computation (radix stages), we will use the private memory, since it offers a better efficiency. Therefore, it is important to check the target hardware specifications, since an abuse of the registers will generate register spilling to global memory, with the corresponding loss of performance. On the other hand, the input data will be stored in the local memory, so that it may be shared between the different work-items, which is the OpenCL terminology for parallel threads. Thus, the synchronization between the threads of a group is performed in the local memory. The different functions of the blocks are invoked using inline expansion, so that when compiling the code, the header is replaced by the body of the function. This improves the kernel performance by not having to make the extra function calls. Another optimization used in these functions is loop unrolling, which improves the code performance at the expense of incrementing the code size. Finally, the use of templates using HPL allow the application of recursion based on specializations where the statics loops are fully unrolled, so that each call is made to a different specialization. Also, the templates allows to reduce the code complexity by making many optimizations at compile-time.

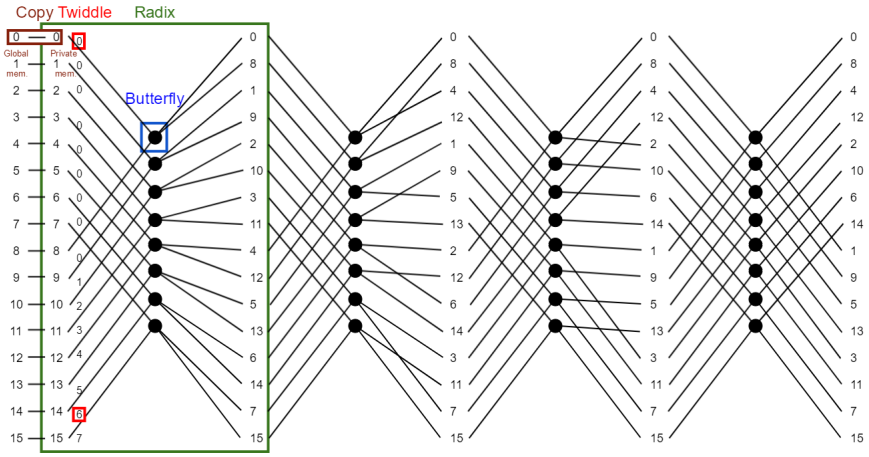

Figure 1: FFT algorithm for *radix-2* and N=16 using the Stockham pattern.

3.1. Computing blocks

This section describes the computing blocks that modify the input data. The algorithms are composed of several layers and each block performs a small part of the algorithm.

`Butterfly` performs the computation of the Butterfly Node and has been implemented simulating a recursion that is performed by calling different specializations. This block is in charge of the computation associated with each *radix* stage, in other words, it is in charge of performing a portion of the calculation of the r size algorithm that will be combined with the rest to obtain the final result. The `Butterfly` block recursively calls each inline specialization until the base case of $radix = 2$ is reached, splitting the input in two. There are two sets of specializations defined depending on the direction (forward/backward) of the transform.

The rotation factors are calculated in the block `Twiddle`. This block multiplies the input data by a complex number whose magnitude is equal to one, so that only the phase of the data is affected. `Twiddle` is in charge of calculating the sine and cosine according to the given angle and the position within the radix block. Once this value has been calculated, it multiplies this result by

each element performing the correspondent unrolling.

The calculation of factors can be based on fast native sine and cosine functions, which can however lead to a loss of accuracy, or by standard functions which may be slower but have good accuracy. Our library allows to choose which option to follow.

`Radix` performs the computation of each stage of the `Butterfly` block. It has N/r inputs and N/r outputs, and it calculates the FFT for a specific point. It is responsible for both subdividing and combining the different radix size portions that will be computed in the `Butterfly` block, calling the appropriate `Twiddle` and `Butterfly` block according to the size of the radix used.

3.2. Reordering blocks

The reordering blocks move the data so that the computing blocks can operate efficiently on the data.

The only reordering block in our library is `Copy`, which is responsible for copying data between different memories of the device. The data may be located either in the global memory or the local memory and the inputs and the outputs do not need to be in the same memory region. Each data copy is performed applying an unrolling factor according to the size of the radix.

4. Signal processing transforms using HBPL

This section discusses the design of butterfly algorithms using the HBPL blocks described in Section 3. For this, the kernels that express these algorithms are written in terms of the HBPL basic blocks. Namely, each algorithm is implemented by a couple of kernels parameterized by a template that specifies the size (`N`), direction (`DIR`) of the transform (forward/backward), radix size (`r`) and size of local memory to use (`SHM`).

Before going into detail in the implementations, we must explain some features of HPL used in our blocks and kernels. One of these characteristics is that HPL allows to express invocations to functions within a code in two different

approaches that have different consequences. First, if a function `f` is invoked using the regular syntax `f(args)`, then its body will be inlined inside the calling function. This has the advantage of reducing the invocation cost, although it will increase the code size if `f` is invoked from different points in the program. Second, HPL allows the invocation of `f` with the syntax `call(f)(args)`, which gives place to a separate function and the regular call. For efficiency reasons, our code uses the first approach.

A second point of interest is that the `barrier(LOCAL)` invocations perform block-level barriers that ensure the consistency of the local memory. Finally, as explained in [8], the control structures in HPL are those of C, with an ending underscore (`if_`, `for_`, etc.) and when they are used in an HPL kernel, the associated C keyword appears in the associated OpenCL kernel. Nevertheless, as mentioned in Section 1 and in [8], regular C control structures can also appear in HPL kernels. In this case they are evaluated at runtime controlling the generation of the OpenCL code. For example in our kernels, `for` is extensively used in order to generate unrolled versions of loops, since the `for` loop will be executed at runtime, giving place to a different copy of its body in the generated OpenCL code for each one of its iterations. Similarly, `if` statements allow to choose at runtime between different pieces of code at any point in an heterogeneous function.

We now describe in turn the FFT and DCT signal transforms expressed using our HBPL library.

4.1. FFT

This kernel, whose pseudocode is shown in Listing 1, receives as input the data to process and it is in charge of carrying out the horizontal transform. Its structure is as follows:

1) The first part is the initialization of the kernel parameters. Line 2 initializes the indices. Then, line 3 initializes the displacements that will be used to access the data. Finally, lines 4 and 5 reserve the memory for the records and local memory.

```
1  template<N,DIR,r,SHM> fft(src := input complex data array) {
2  //       initialization of the identifiers
3  //       initialization of the offsets
4           reg := initialization of r registers
5           shm := initialization of local memory of size SHM
6
7           copy<r>(src, reg) // copy r elements from source to reg
8
9           mixrad := obtain the mixed radix
10          radix<mixrad,DIR>(reg) //        first mixrad step
11
12          from i=mixrad to N with i*= r {
13 //                offsets recalculation
14
15 //                local memory synchronization
16                  copy<r>(reg, shm) // copy r elements from reg to
                        shm
17 //                local memory synchronization
18                  copy<r>(shm, reg) // copy r elements from shm to
                        reg
19
20                  ang := angle rotation based on i
21                  radix<r,DIR>(reg, ang) // apply rotation
22          }
23          copy<r>(reg,src) // copy r elements from reg to src
24 }
```

Listing 1: HBPL FFT kernel

2) Line 7 belongs to the preprocessing stage, in which a copy of the data to process is made to the previously reserved records.

3) The processing for mixed radix [37] is performed in lines 9 to 10. It is responsible for calling the butterfly function.

4) Lines 12 to 22 carry out in a cyclical way the radix stages, that is, in each iteration of this loop a butterfly stage, such as the ones depicted in Figure 1, is performed. Each stage first updates the positions of the readings and displacements, as line 13 shows. After this a reorganization stage using the local

memory is performed (lines 15 to 18). Finally, function `Radix` performs the actual computations, which requires as input the angle with which the rotation will be performed, as we can see in lines 20 and 21. The radix function is in charge of calling the butterfly and twiddle functions.

5) Finally, the results are copied to the global memory in line 23.

4.2. DCT

This algorithm has a structure very similar to that of the FFT, as it is based on it. One of the main differences between both algorithms is that DCT operates on real data while the FFT works on complex data. Because of this, in the DCT it is necessary to perform a preprocessing and a post-processing stage, which basically do the data conversion. Listing 2 shows the pseudocode of the kernel in charge of the horizontal transform, which works in the following way:

1) The first part is the initialization of the kernel parameters. Concretely, lines 2 initializes the indices, line 3 initializes the displacements that will be used to access the data, and finally, lines 4 and 5 reserve memory for the records and local memory.

2) Once the parameters have been initialized, a pre-processing stage is performed in lines 7-13. Here the real input data are copied to the registers, which hold complex data. This transformation is made because the blocks of our library are encoded to work with complex data. This decision was taken because this increases the generality of the implementation, just requiring a pre/post-processing stage in the kernels to convert from real to complex when the data to process are not complex values.

3) A first processing step is executed for mixed radix [37] in lines 15 and 16.

4) The remaining radix stages are done in lines 18 to 28. These computations are carried out in a loop, whose body begins with the update of the positions of reading and displacement at line 19. Once the indexes are updated, a reorganization stage takes place using the local memory in lines 21 to 24. Finally, function `Radix` performs a computation stage, as shown in lines 26 and 27.

```
1  template<N,DIR,r,SHM> dct(Array<float,1> src) {
2  //       initialization of the identifiers
3  //       initialization of the offsets
4           reg := initialization of r registers
5           shm := initialization of local memory of size SHM
6
7           if(DIR > 0) { // forward convert double to complex
8                   copyDCT<r>(src, shm); // convert double to complex
9  //               local memory synchronization
10                  packDCT<N,r,DIR,SHM>(shm,reg); }
11          if(DIR < 0) { // backward
12                  copyDCT<r>(src,reg);  // convert double to complex
13                  radixDCT<N,r,DIR,SHM>(reg); }
14
15          mixrad := obtain the mixed radix
16          radix<mixrad>// first mixrad step
17
18          from i=mixrad to N with i*= r {
19 //               offsets recalculation
20
21 //               local memory synchronization
22                  copy<r>(reg, shm); // copy r elements from reg to shm
23 //               local memory synchronization
24                  copy<r>(shm, reg); // copy r elements from shm to reg
25
26                  ang := angle rotation based on i
27                  radix<r>(reg, ang) // apply rotation
28          }
29
30 //       local memory synchronization
31          if(DIR > 0) { // forward
32                  radixDCT<N,r,DIR,SHM>(reg);
33                  copy<r>(reg, shm); }
34          if(DIR < 0) { // backward
35                  packDCT<N,r,DIR,SHM>(reg, shm); }
36
37 //       local memory synchronization
38          copyDCT<r>(shm, src); // convert complex to double
39 }
```

Listing 2: HBPL DCT kernel

5) In the post-processing stage, which takes place in lines 30 to 35, the data in the local memory is rearranged in order to write them to the global memory.

6) Finally, the results are copied from the local memory to the global memory in lines 37 and 38.

5. Tuned strategy for Butterfly algorithms

One of the main requirements for high performance on accelerators is to obtain sufficient parallelism. This parallelism can be exploited by running the maximum possible number of simultaneous OpenCL work-groups, which we denote as block parallelism, as well as the largest possible number of OpenCL work-items per work-group, which we call thread parallelism.

In order to find the optimal configuration, the factor limiting the performance must be computed and this depends of the device architecture.

Table 1 displays some values that describe the architectures used in our evaluation in terms of resources that affect their performance. We now describe in turn these parameters. In this table, R_{CU}^{max} represents the total registers per Computation Unit, S_{CU}^{max} represents the maximum local memory size per Compute Unit, w represents the *wavefront*/*warp* in AMD(64)/NVidia(32), L_{CU}^{max} stands for the maximum number of w-size threads that can be run concurrently in the Compute Unit, B_{CU}^{max} is the maximum number of blocks that can be processed simultaneously per Computation Unit due to inherent hardware limitations and R_{thread}^{max} represents the maximum number of register that can be used by each thread. We have not been able to locate all the values required for the Intel Xeon in any manual, white paper or website, so it is not possible to apply to it the calculations discussed in this section. As we will see in Section 6, this does not preclude the applicability of our library to this system.

5.1. Block (work-group) parallelism

The block parallelism phase of our tuning strategy is focused in obtaining the maximum number of work-groups or blocks of threads per Compute Unit

Device	R_{CU}^{max}	S_{CU}^{max}	w	L_{CU}^{max}	B_{CU}^{max}	R_{thread}^{max}
NVidia GTX 750Ti (Maxwell [38])	64KB	96KB	32	64	32	255
NVidia K20m (Kepler [39])	64KB	48KB	32	64	16	255
AMD FirePro S9150 (Hawaii [40], [41])	64KB	64KB	64	256	16	112
Intel Xeon E5-2660	128KB	32KB	128?	8192	480?	?

Table 1: Device architecture information.

that can be processed in parallel (B_{CU}). The model must take into account that the number of active blocks per Compute Unit is limited by resources such as the registers and the local memory available as well as by intrinsic hardware limitations of the device. This fact, already observed in BPLG, is summarized in the next equation:

$$B_{CU} = \min(B_{CU}^{R},\ B_{CU}^{s},\ B_{CU}^{max}) \tag{1}$$

where each term has the following meaning and value:

- B_{CU}^{R} represents the maximum number of blocks that can be simultaneously active according to the number of registers of the Compute Unit:

$$B_{CU}^{R} = R_{CU}^{max}/R_B \ = \ R_{CU}^{max}/(R_t \times L) \tag{2}$$

  where R_{CU}^{max} represents the total registers per Computation Unit and R_B is the number of registers per block, which is obtained multiplying the number of registers per thread R_t by the number of threads per block L.

- B_{CU}^{s} represents the number of blocks that can be executed simultaneously as a function of the local memory available:

$$B_{CU}^{s} = S_{CU}^{max}/S_B \ \ = S_{CU}^{max}/(sizeof(D_t) \times r \times L) \tag{3}$$

  where S_{CU}^{max} represents the local memory size and S_B the local memory by block. Each block stores the data of the L threads, where each thread operates on r elements of type D_t

- B_{CU}^{max} is the maximum number of blocks that can be processed simultaneously per Computation Unit due to inherent hardware limitations.

The purpose of our block parallelism tuning strategy is to calculate tuned parameters that allow to maximize Eq. (1) considering the values that describe the architecture. This requires calculating the amount of private registers P, local memory S and threads per block L required for the execution of a code. We consider values that are powers of two $P = 2^p$, $S = 2^s$ and $L = 2^l$, so that in what follows we refer to the exponents p, s and l, respectively. It must be noted that the local memory required is equal to the private registers per block, and thus $s = p + l$.

Table 2 shows the values of the tuner parameters (s, p, l) computed in order to exploit the block parallelism in the Nvidia GPUs that will be used in our experiments in Section 6. In this Table, w_B represents the *wavefronts*/*warps* per block and the size of the *radix* is $r = P = 2^p$ since one register per element is used.

Below is an example for the calculation of the Table 2 values for the NVidia GTX 750Ti.

$$\begin{aligned} S_B = S_{CU}^{max}/B_{CU}^{max} = 96KB/32 = 3072B \\ S = S_B/sizeof(Dt) = 3072B/8B = 384 < 2^9 \rightarrow s = 8 \\ radix\text{-}2 \rightarrow P = 2 \rightarrow p = 1 \rightarrow l = s - p = 8 - 1 = 7 \rightarrow L = 128 \\ radix\text{-}4 \rightarrow P = 4 \rightarrow p = 2 \rightarrow l = s - p = 8 - 2 = 6 \rightarrow L = 64 \\ radix\text{-}8 \rightarrow P = 8 \rightarrow p = 3 \rightarrow l = s - p = 8 - 3 = 5 \rightarrow L = 32 \end{aligned} \tag{4}$$

These values maximize the block parallelism available in the accelerator.

5.2. Thread (work-item) parallelism

The aim of the thread level parallelism is to increase the number of threads per block that are executed (L). For this, we must take into account that each thread is in charge of performing the computation associated to a *radix-r*

Device	w_B	p	s	l
NVidia GTX 750Ti	1	3	8	5
	2	2	8	6
	3	1	8	7
	4	1	8	7
NVidia K20m	1	3	8	5
	2	2	8	6
	3	1	8	7
AMD FirePro S9150	1	3	10	7
	2	2	10	8
	3	1	9	8
	4	1	9	8

Table 2: FFT optimal parameters.

butterfly. Therefore, given a problem of size N, a total of $L_1 = N/r$ are used to process it. When L_1 is small, we resort to batch execution, which consists in processing simultaneously L_2 different problems of the same size, so that the total number of threads in use is $L = L_1 \times L_2$.

The thread parallelism is maximized by considering the values that imply the largest w_B, which are those associated to the smallest *radix (p)* as Table 2 shows. It must be taken into account that as the problem size $N = 2^n$ grows, keeping a small *radix* implies increasing the number of operations. Thus, when $n \gg p$ the radix is increased. For large problems ($n > s$) using a s larger than the optimum one is considered in order to simplify the memory access patterns. Table 3 shows the best theoretical configuration obtained considering this. For the AMD GPU the model has been adapted in order to maximize the use of the local memory, so that the optimal parameters are as follows: for $n = 2, 3, 4 \rightarrow p = 1, s = 9, l = 8$, for $n = 5, 6 \rightarrow p = 2, s = 10, l = 8$, for $n = 7, 8, 9 \rightarrow p = 3, s = 10, l = 8$ and for $n = 10, 11, 12 \rightarrow p = 4, s = n, l = n - p$.

Device	N	r	R_t	L_1	L_2	B_{CU}
NVidia GTX 750Ti	4	2	15	2	32	32
	8	2	20	4	16	32
	16	2	21	8	8	32
	32	2	20	16	4	32
	64	4	29	16	4	32
	128	4	29	32	2	32
	256	4	29	64	1	32
	512	4	29	128	1	32
	1024	8	40	128	1	24
	2048	8	40	256	1	24
	4096	8	40	512	1	24
NVidia K20m	4	2	15	2	64	16
	8	2	20	4	32	16
	16	2	21	8	16	16
	32	2	20	16	8	16
	64	4	29	16	8	12
	128	4	29	32	4	12
	256	4	29	64	2	12
	512	4	29	128	1	12
	1024	4	29	256	1	12
	2048	8	40	256	1	12
	4096	8	40	512	1	12
AMD FirePro S9150	4	2	11	2	128	8
	8	2	19	4	64	8
	16	2	19	8	32	8
	32	4	20	8	32	4
	64	4	25	16	16	4
	128	8	48	16	8	4
	256	8	48	32	4	4
	512	8	48	64	2	4
	1024	16	105	64	1	4
	2048	16	113	128	1	2
	4096	16	117	256	1	1

Table 3: FFT best theoretical parameters.

6. Results

This section evaluates the performance of HBPL comparing it with the original BPLG [2] on CUDA, as well as the libraries *cuFFT* [24] on CUDA and *clFFT* [25] on OpenCL. The experiments also consider a regular CPU in which HBPL is also compared with a well-known CPU library such as FFTW [12]. All the tests have been performed using from 4 to 4096 single-precision elements and with the data already loaded in the memory of the device at the beginning of each experiment. It is important to comment that the *clFFT* library performs the calculation of the rotation factors on the CPU when the kernel is created, the factors then being included as constants in the string of the kernel to compile. Thus in this case, this computation is not included in the execution of the kernel, which provides a slight artificial improvement in performance in the repeated execution of the kernel for its measurement. The performance of the different libraries is measured in GFlops:

$$5Nlog_2(N)iterations10^{-9}/t \tag{5}$$

where *iterations* is the number of signals processed (applying the direct and the inverse transform) and t the time in seconds, using all the experiments $t = 10$ seconds.

Table 4 describes the platforms and associated software used for the tests. On platforms with NVidia GPUs, the CUDA-based measurement relied on the 6.5 CUDA SDK. **Tests performed using the more recent 8.0 CUDA SDK supported by the drivers of the Platform 2 yielded very similar results, the maximum cuFFT performance difference between both versions being below 3%. The CUDA performance shown is thus also representative for newer software platforms.** Our library uses the native functions `native_` for the calculation of the sine and cosine functions in the GPUs, because in these platforms they provide important improvements in performance without incurring in accuracy penalties.

	Platform 1	Platform 2	Platform 3	Platform 4
CPU	i7-4790	E5-2660	E5-2650v2	E5-2660
Memory	32 GB	64 GB	64 GB	64 GB
OS	Ubuntu 14.04	CentOS 6.7	CentOS 6.7	CentOS 6.7
Device	NVidia GTX 750Ti 2047 MB [GM107-400-A2] v346.72 GNU C++ 4.8.4 CUDA OpenCL 1.2	NVidia K20m 5119 MB [GK110] v367.48 GNU C++ 4.8.2 CUDA OpenCL 1.2	AMD FirePro S9150 16192 MB [Hawaii XT GL] GNU C++ 4.8.2 OpenCL 2.0	Intel®Xeon E5-2660 2.20 GHz/ 3.2 GHz Sandy Bridge-EP GNU C++ 4.8.2 Intel OpenCL 1.2 *b.* 8

Table 4: Platforms used in the experiments.

Our library has two installation modes, which we call analytical and empirical parametrization, respectively. The analytical installation is applicable only on those devices where all the parameters in Table 1 are available, because it relies on the model described in Section 4, whose results are displayed in Table 3, to parametrize the blocks of the library algorithms. As a result this installation requires a totally negligible time. The empirical parametrization, which is applicable in all the devices, performs an exhaustive search for the best parameters within the subset of parameters combinations allowed by our model by timing each combination. This guarantees optimal results at the cost of a more time-consuming installation. For example, the maximum runtime observed in our experiments for this empirical parametrization and execution in each device of among all the different combinations of radix, L and input size, which was reached on the NVidia platforms with a total of 160 runs, was 34 minutes.

6.1. FFT

First, for each platform except the Intel Xeon, a table is shown with the results of the best theoretical configurations reflected in Table 3 and the best result obtained using a wide range of combinations for the radix size and the number of threads per block. In addition, dedicated NVidia and AMD GPU profilings are discussed below.

N	Model	Empirical parametrization	Difference (%)
4	45.17	45.17	0
8	67.95	67.95	0
16	90.1	90.1	0
32	105.81	112.79	6.19
64	135.94	135.94	0
128	158.29	158.29	0
256	181	181	0
512	201.74	204.17	1.19
1024	224.89	226.82	0.85
2048	248.8	248.8	0
4096	205.65	205.65	0

Table 5: Comparison between the maximum FFT performance (in GFlops) obtained using the model and the empirical parametrization in the NVidia 750Ti.

Table 5 shows the difference between the performance obtained by our tuning strategy and the best performance obtained using a wide range of combinations for the different parameters on the NVidia 750Ti. The result is quite good, as in most cases the analytical model matches the best possible result, with a maximum performance difference of 6.2%. The profiling results for this platform are shown in Table 6. They include the number of registers R_t, the local memory used S, the occupancy (percentage) % and the GFlops depending on the input size, the radix and the number of threads per block. As can be observed, the occupancy is reduced from $n > 10$, and thus these case must be studied with other strategy.

Table 7 shows the difference between the performance obtained by our tuning strategy and the best performance obtained using a wide range of combinations for the different parameters on the Nvidia K20m. In this case, the parameterizable model obtains nearly perfect results, with only a very small deviation for $n = 8$.

n	L	r	Rt	S	Ocu(%)	Gflops
2	128	2	15	2056	100	**45.17**
	256	2	15	4104	100	44.86
3	128	2	20	2056	100	**67.95**
	256	2	20	4104	100	67.49
4	128	2	21	2056	100	**90.02**
	256	2	21	4104	100	89.54
5	128	2	20	2056	100	105.81
	64	4	29	2056	100	**112.79**
6	64	4	29	2056	100	**135.94**
	128	4	29	4104	100	135.06
7	64	4	29	2056	100	**158.29**
	128	4	29	4104	100	157.33
8	64	4	29	2056	100	**181**
	128	4	29	414	100	179.96
9	128	4	29	4104	100	201.74
	512	4	29	16392	100	**204.17**
10	512	4	29	16392	100	**226.82**
	128	8	40	8200	75	224.89
11	512	4	31	16392	38	209.02
	256	8	40	16392	75	**248.8**
12	1024	4	31	32776	50	165.57
	512	8	40	32776	50	**205.65**

Table 6: NVidia 750Ti FFT profiling.

Table 8 shows the difference between the performance obtained by our tuning strategy and the best performance obtained using a wide range of combinations for the different parameters on the AMD FirePro S9150. The analytical model employed also achieves very good performance, in fact the result obtained is very close to the best possible result, with a maximum deviation of just 1.6%. Table 9 shows the profiling results for the AMD FirePro, which include the number of registers R_t, the local memory used S, the occupancy (percentage) % and the GFlops depending on the input size, radix and the threads per block. Notice that the maximum number of registers for $n < 11$ is 112 and there is no local memory spilling. For $n \geq 11$, the configuration violates the optimal

N	Model	Empirical parametrization	Difference (%)
4	103.22	103.22	0
8	151.76	152.36	0.39
16	180.2	180.2	0
32	236.42	236.42	0
64	313.78	313.78	0
128	365.36	365.36	0
256	419.84	419.84	0
512	407.8	407.8	0
1024	414.45	414.45	0
2048	290.06	290.06	0
4096	243.04	243.04	0

Table 7: Comparison between the maximum FFT performance (in GFlops) obtained using the model and the empirical parametrization in the NVidia K20m.

resource factors, but the optimal implementation must be studied with other strategy. Finally, it should be observed that the occupancy is high for small size problems while for big size problems the occupancy decreases as the problem size increases.

Table 10 shows the results obtained in the Intel Xeon by the FFT. In this case, since there is no theoretical model, it shows the data obtained by the empirical parametrization, depending on the radix and L.

Next we compare the performance achieved by the BPLG, *cuFFT*, *clFFT* and HBPL FFT algorithms as a function of the input size. Figure 2 shows the performance achieved by the four libraries for different problem sizes of FFT in the NVidia 750Ti (Maxwell) GPU. We can see that the HBPL library developed in this work offers a performance very similar to that of BPLG and *cuFFT*, with increasing performance as the input size grows with the exception of $N = 4096$ for HBPL. Comparing HBPL with the *clFFT* library, both libraries

N	Model	Empirical parametrization	Difference (%)
4	149.51	151.38	1.25
8	232.83	232.83	0
16	313.63	313.63	0
32	382.08	385.37	0.86
64	468.8	469.59	0.17
128	544.56	544.56	0
256	621.91	631.96	1.62
512	706.43	706.43	0
1024	750.41	750.41	0
2048	792.19	792.19	0
4096	796.37	796.37	0

Table 8: Comparison between the maximum FFT performance (in GFlops) obtained using the model and the empirical parametrization in the AMD FirePro.

have a similar behaviour up to the problem size $N = 128$, from which the *clFFT* performance begins to drop considerably; to the point that HBPL is 4.13 times faster than *clFTT* for the largest problem size tested, 4096. This is surely due to the fact that *clFFT* is an evolution of *clAMDFFT* [42], which is optimized for AMD architectures and thus cannot exploit all the performance of the NVidia hardware as the problem size increases.

Figure 3 shows the performance of the libraries considered when running FFT for different input sizes on the NVidia K20m (Kepler) GPU. In this case, HBPL performs worse than BPLG and *cuFFT* for problems larger than 256. A very similar behaviour can be also seen in the BPLG library, which also suffers a degradation of performance with larger input sizes. This is not surprising given its particular focus on small problems sizes. Regarding the other portable solution, *clFFT* clearly provides a worse performance, always lagging with respect to the other libraries. In this accelerator our proposal is up to 4.64 faster than the *clFFT*. In addition, just like in the NVidia 750Ti GPU, the *clFFT* library performance drops further from $N = 256$, showing the bad adaptation to NVidia platforms commented in the previous experiments.

n	L	r	Rt	S	Ocu(%)	Gflops
2	64	2	11	1024	100	**151.38**
	128	2	11	2048	80	147.82
	256	2	11	4096	100	149.51
3	64	2	19	1024	100	227.83
	128	2	19	2048	80	232.55
	256	2	19	4096	100	**232.83**
4	64	2	19	1024	100	308.42
	128	2	19	2048	80	311.6
	256	2	19	4096	100	**313.63**
5	64	2	19	1024	100	278.35
		4	25	2048	80	381.89
	128	2	19	2048	80	274.16
		4	25	4096	80	**385.37**
	256	2	19	4096	100	280.2
		4	25	8192	80	382.08
6	64	4	25	2048	80	**469.59**
	128	4	25	4096	80	452.72
	256	4	25	8192	80	468.8
7	64	4	25	2048	80	474.26
		8	48	4096	40	537.1
	128	4	25	4096	80	478.44
		8	48	8192	40	**544.56**
	256	4	25	8192	80	482.76
		8	48	16384	40	521.38

n	L	r	Rt	S	Ocu(%)	Gflops
8	64	4	25	2048	80	546.97
		8	48	4096	40	**631.96**
	128	4	25	4096	80	547.26
		8	48	8192	40	621.91
	256	4	25	8192	80	553.93
		8	48	16384	40	620.85
9	64	8	48	4096	40	696.92
	128	8	48	8192	40	**706.43**
	256	8	48	16384	40	685.17
10	64	16	105	8192	20	**750.41**
	128	8	56	8192	40	702.3
		16	105	16384	20	736.58
	256	8	56	16384	40	701.94
		16	105	32768	20	683.71
11	128	16	113	16384	20	**792.19**
	256	8	60	16384	40	757.76
		16	113	32768	20	744.25
12	256	16	117	32768	20	**796.37**

Table 9: AMD FirePro FFT profiling.

Figure 4 compares the performance of the FFT algorithm of the HBPL and *clFFT* libraries on the AMD FirePro GPU. Notice that the CUDA-based libraries cannot be executed in this architecture, further motivating the development of portable approaches as HBPL. As it can be seen, HBPL provides better performance than *clFFT*, reaching a maximum speedup with respect to *clFFT* of 1.39, and being only slightly surpassed by that library in the $N = 8$ case. It also deserves to be mentioned that *clFFT* suffers a great loss of performance

N	Radix-2					Radix-4					Radix-8					Radix-16				
	L64	L128	L256	L512	L1024	L64	L128	L256	L512	L1024	L32	L64	L128	L256	L512	L16	L32	L64	L128	L256
4	9.83	9.89	9.54	9.79	9.76	21.36	**23.58**	23.3	22.86	23.5										
8	7.76	7.68	7.59	7.59	7.49	11	10.92	10.9	10.89	10.55	34.39	34.96	34.97	**35.31**	35.2					
16	16.64	16.63	16.15	16.4	15.19	14.6	14.52	14.45	14.41	14.36	12.41	12.37	12.46	12.14	12	43.98	45.89	45.39	**46.05**	45.44
32	16.69	16.4	16.24	15.97	14.68	24.9	25.46	**25.5**	14.66	23.19	15.19	15.25	15.16	14.9	14.81	13.7	13.71	13.63	13.63	13.54
64	16.08	16.24	15.34	15.53	14.07	29.74	29.37	30.13	29.3	27.67	47.7	47.27	47.43	**47.82**	46.65	16	15.42	16.03	15.98	15.83
128	16.54	16.05	15.54	15.04	13.44	26.01	25.56	25.42	23.99	22.27	31.79	31.56	31.32	29.96	28.79	43.94	45.85	**46.52**	45.28	45.54
256		15.72	15.22	14.66	12.96	29.45	28.8	28.49	26.29	25.47	35.79	35.43	34.56	33.6	32.36	46.96	49.64	50.14	**52.41**	50.91
512			15.22	14.54	12.91		24.77	24.31	22.53	21.99		**38.8**	38.62	36.85	35.86		31.71	31.66	31.46	30.94
1024				14.3	12.69			28.09	26.14	24.65			30.32	30.47	28.63			35.2	**35.3**	33.88
2048					14.1				23.67	22.14				33.23	31.42				**39.24**	37.88
4096										24.24					34.51					**41.13**

Table 10: Xeon FFT performance (GFlops).

for the largest input size, 4096.

Figure 5 shows the performance achieved by the FFT implementations of the HBPL, the *clFFT* and the FFTW for different input sizes in the Intel Xeon CPU E5-2660, which again, does not support the CUDA-based alternatives. HBPL exceeds the performance of the FFTW library for almost every value except for $n = 32$, where the performance drops by half. The clFFT library surpasses the HBPL library in more than the half of the cases, however clFFT is not applicable for sizes below 256 due to a problem with the vectorizer of the Intel OpenCL platform.

Figure 6 shows the speedup that our HBPL library achieves for different problem sizes of the FFT with respect to clFFT, the only other portable alternative. This allows the figure to consider all the platforms tested. It can be seen that on NVidia platforms, HBPL achieves a higher speedup relative to clFFT for the larger sizes. In the FirePro GPU, however, the performance difference between HBPL and clFFT is very uniform, with the exception of a speedup bump for HBPL for $n = 4096$. The Intel Xeon is the only platform where clFFT is usually faster than HBPL, the problem being that clFFT does not work for most of the problem sizes tested, as shown in Figure 5.

In general, the kernels developed for the FFT transform in HBPL offer a proper performance when compared with BPLG under CUDA, having both

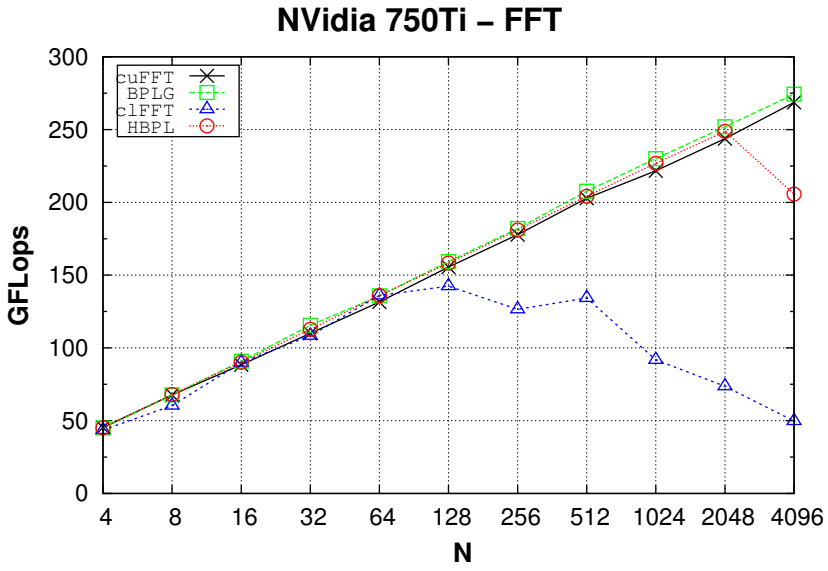

Figure 2: FFT on the NVidia 750Ti (Maxwell) GPU.

libraries similar problems for large problem sizes. Regarding the non-NVidia platforms, the comparison of HBPL with clFFT in them is very satisfactory.

As mentioned in the introduction, our library also supports two-dimensional square transforms. We illustrate their performance in Table 11, which shows the performance for the FFT algorithm with 2 dimensions in the AMD platform. It can be seen how from sizes of 64×64, with 4096 total elements, the performance starts to drop.

6.2. DCT

Our evaluation of the performance achieved for the DCT algorithm only relies on the HBPL and BPLG libraries because *cuFFT* and *clFFT* do not provide a version of this algorithm, which further motivates the development of our library for non-NVidia platforms. Table 12 compares the performance of BPLG and HBPL for different problem sizes when running the DCT algorithm in the four systems considered in this manuscript. We can see that in the NVidia

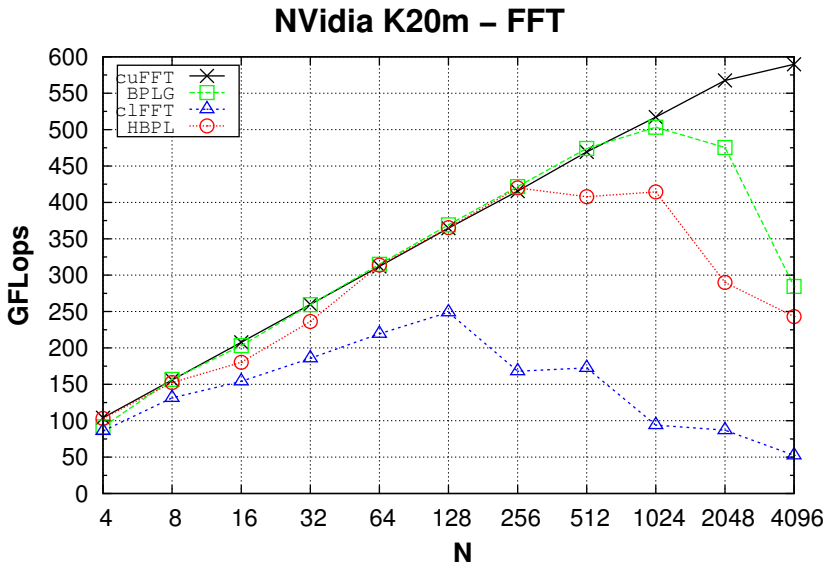

Figure 3: FFT on the NVidia K20m (Kepler) GPU.

N	Radix-2			Radix-4			Radix-8				Radix-16				
	L16	L32	L64	L16	L32	L64	L8	L16	L32	L64	L4	L8	L16	L32	L64
4×4	157.31	157.26	**157.78**	145.20	149.02	150.80									
8×8	239.17	240.32	**242.53**	208.57	219.90	215.72	208.83	178.05	188.06	192.49					
16×16	305.12	304.12	306.66	309.42	**314.69**	313.32	300.90	280.18	283.68	285.86	296.05	285.30	215.95	222.55	220.42
32×32	246.94	251.44	259.76	318.47	321.57	334.98	310.57	333.62	347.40	**348.96**	259.77	336.70	291.29	289.67	278.13
64×64		245.36	257.09	281.62	290.61	324.02	326.95	359.91	374.33	**395.03**	299.66	386.42	358.07	360.38	341.57
128×128			266.80		309.42	366.08		291.11	311.11	329.12		343.31	354.57	364.54	**377.78**
256×256						345.18			324.45	348.16			314.61	352.47	**352.70**
512×512										**336.42**					336.32
1024×1024															**332.24**

Table 11: AMD 2D FFT performance (GFlops).

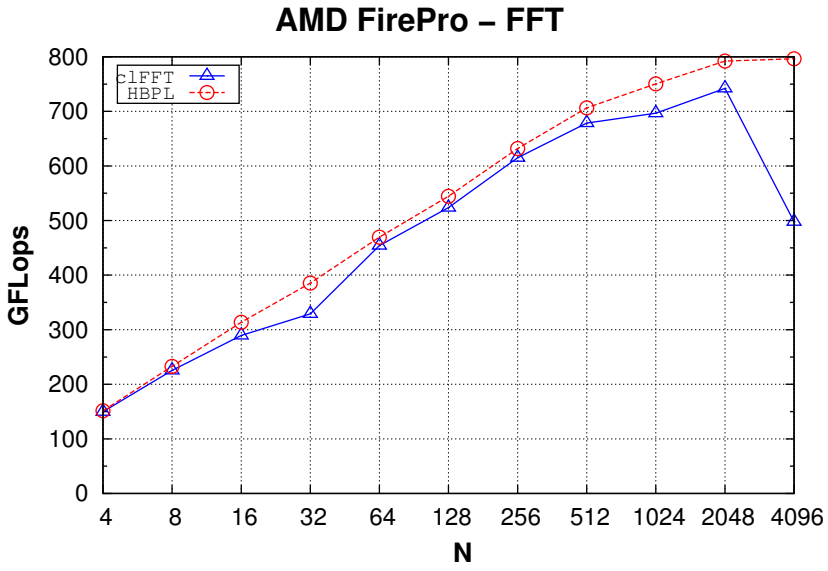

Figure 4: FFT on the AMD FirePro GPU.

GPUs both libraries have a similar behaviour, BPLG offering somewhat better performance. As for the FirePro GPU, the performance is far superior to any other platform tested, reaching its maximum for the 64×64 problem size, from which it slightly drops in performance. Also, as expected, the regular Xeon CPU provides way less computing power than any of the GPUs considered, as they have many more computational units. This justifies the predominant interest in accelerators.

7. Conclusions

We have presented the design and implementation of a series of portable and parameterizable kernels that allow the calculation of the orthogonal signal transforms FFT and DCT. These transforms are widely used in many fields, such as image, digital signal and multimedia processing. Our portable implementations follow a parameterizable strategy, which provides Butterfly algorithms designed taking into account that they have a divide-and-conquer structure.

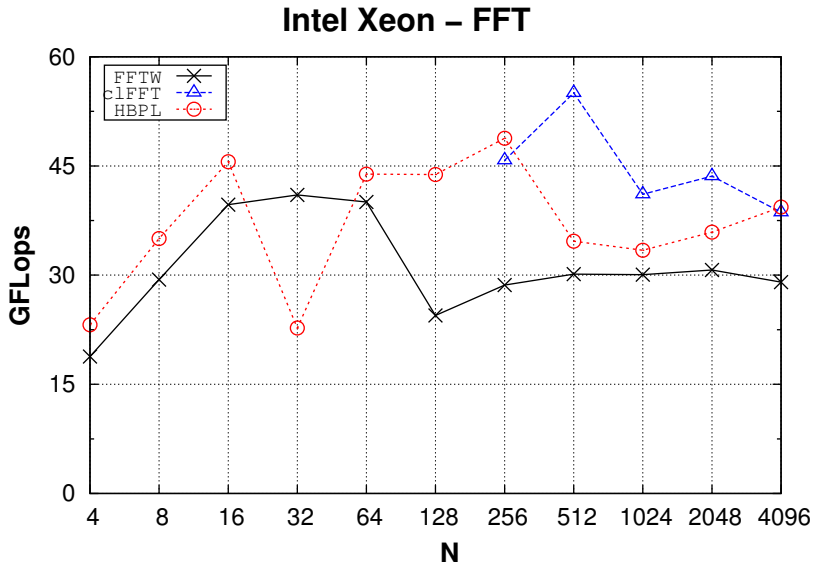

Figure 5: FFT on the Intel Xeon E5-2660.

The predecessor of this library, BPLG, provides a very good performance, but it can only be used in NVidia GPUs, thus providing very little portability. Our proposal, called HBPL for Hetereogeneous Butterfly Processing Library, relies however on OpenCL, the standard for heterogeneous computing, which allows to execute these codes on a wide range of devices. Since basic OpenCL lacked some of the features required for our purpose, our implementation was performed on top of HPL, a framework that extends OpenCL with several capabilities that allow to exploit templates and runtime coe generation on the existing implementations of the standard. Our implementation heavily relies on these features in order to allow the adaptation of the code to the underlying architecture and the properties of the problem to solve so as to attain good portable performance.

The algorithms provided by HBPL offer a performance similar to the BPLG library, with the difference that ours are portable, which allows to use them beyond NVidia devices. Comparing HBPL with the *clFFT* library, the other

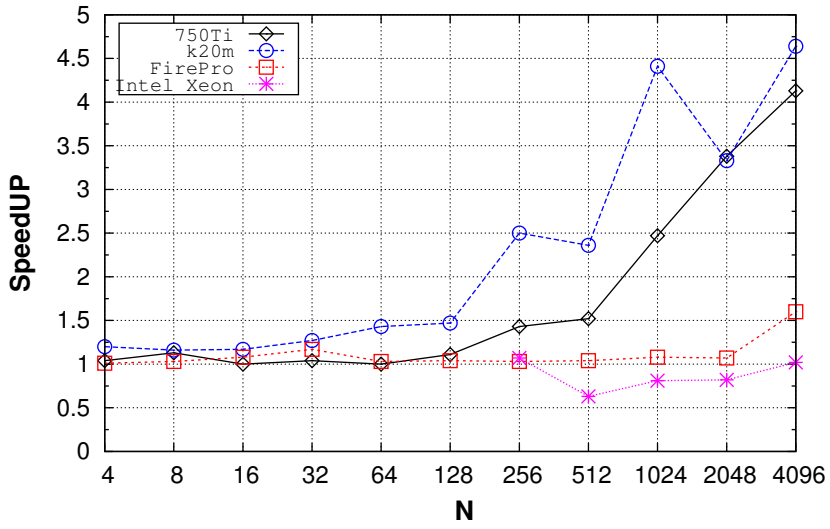

Figure 6: Speedup of HBPL with respect to clFFT for different problem sizes of the FFT.

Platform	Library	4×4	8×8	16×16	32×32	64×64	128×128	256×256	512×512	1024×1024
NVidia 750Ti	HBPL	37.89	38.23	49.65	55.93	79.43	71.55	81.42	87.11	53.78
	BPLG	40.73	44.71	53.76	65.27	94.58	87.82	100.95	88.13	74.35
NVidia K20m	HBPL	68.67	86.99	99.05	75.2	97.72	99.08	109.81	97.4	79.96
	BPLG	81.7	93.37	111.83	79.28	109.24	107.93	120.91	126.41	86.55
AMD FirePro	HBPL	157.78	242.53	314.69	348.96	395.03	377.78	352.7	336.42	332.24
Intel Xeon	HBPL	2.48	3.66	4.44	3.88	5.97	6.17	6.8	5.98	5.78

Table 12: Xeon FFT performance (GFlops).

portable alternative for this kind of algorithms, two advantages can be seen: first, HBPL achieves better performance than *clFFT* on all the GPUs analyzed, and second, unlike *clFFT* our version can be used for any problem size on Intel platforms such as a regular Xeon CPU.

8. Acknowledgments

This research has received financial support from the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (TIN2016-75845-P), by the Government of Galicia (Xunta de Galicia) co-founded by ERDF

funds under the Consolidation Programme of Competitive Reference Groups (Ref. ED431C 2017/04) and the Consolidation Programme of Competitive Research Units (Ref. R2014/049 and Ref. R2016/037) as well as by the Xunta de Galicia (Centro Singular de Investigación de Galicia accreditation 2016-2019) and the European Union (European Regional Development Fund, ERDF) under Grant Ref. ED431G/01.

References

[1] C. J. Weinstein, Quantization effects in digital filters (1969) 96.

[2] J. Lobeiras, M. Amor, R. Doallo, BPLG: A tuned Butterfly Processing Library for GPU architectures, Int. J. Parallel Program. 43 (6) (2015) 1078–1102.

[3] J. W. Cooley, J. W. Tukey, An Algorithm for the Machine Calculation of Complex Fourier Series, Mathematics of Computation 19 (90) (1965) 297–301.

[4] A. P. Diéguez, M. Amor, J. Lobeiras, R. Doallo, Solving large problem sizes of index-digit algorithms on GPU: FFT and Tridiagonal System Solvers, IEEE Transactions on Computers. 67 (1) (2018) 86–101.

[5] OpenCL 2.2 Specification, https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf (2017).

[6] D. Vandevoorde, M. M. Josuttis, C++ Templates: The Complete Guide, 1st Edition, Addison-Wesley Professional, 2002.

[7] Khronos Group: Conformant Products - OpenCL, https://www.khronos.org/conformance/adopters/conformant-products/opencl, [Last visit: July 25, 2018] (2017).

[8] M. Viñas, Z. Bozkus, B. B. Fraguela, Exploiting heterogeneous parallelism with the Heterogeneous Programming Library, J. Parallel and Distributed Computing 73 (12) (2013) 1627–1638.

[9] M. Viñas, B. B. Fraguela, Z. Bozkus, D. Andrade, Improving OpenCL programmability with the Heterogeneous Programming Library, International Conference on Computational Science (ICCS 2015) 51 (2015) 110–119.

[10] J. F. Fabeiro, D. Andrade, B. B. Fraguela, Writing a performance-portable matrix multiplication, Parallel Computing 52 (2016) 65–77.

[11] N. Ahmed, T. Natarajan, K. R. Rao, Discrete Cosine Transfom, IEEE Trans. Comput. 23 (1) (1974) 90–93.

[12] M. Frigo, S. Johnson, The design and implementation of fftw3 93 (2005) 216 – 231.

[13] Intel Integrated Performance Primitives, https://software.intel.com/en-us/intel-ipp (2018).

[14] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo, Spiral: Code generation for dsp transforms, Proceedings of the IEEE 93 (2) (2005) 232–275.

[15] B. B. Fraguela, Y. Voronenko, M. Püschel, Automatic tuning of discrete fourier transforms driven by analytical modeling, in: 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2009, pp. 271–280.

[16] T. Popovici, T.-M. Low, F. Franchetti, Large bandwidth-efficient FFTs on multicore and multi-socket systems, in: To appeari in IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018.

[17] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, N. K. Govindaraju, Auto-tuning of Fast Fourier Transform on Graphics Processors, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, 2011, pp. 257–266.

[18] Y. Li, Y.-Q. Zhang, Y.-Q. Liu, G.-P. Long, H.-P. Jia, Mpfft: An auto-tuning fft library for opencl gpus, Journal of Computer Science and Technology 28 (1) (2013) 90–105.

[19] A. Nukada, S. Matsuoka, Auto-tuning 3-D FFT Library for CUDA GPUs, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, 2009, pp. 1–10.

[20] J. Lobeiras, M. Amor, R. Doallo, Designing efficient index-digit algorithms for CUDA GPU architectures, IEEE Trans. Parallel Distrib. Syst. 27 (5) (2016) 1331–1343.

[21] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, D. Kim, Tera-scale 1D FFT with low-communication algorithm and Intel Xeon Phi coprocessors, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, 2013, pp. 1–12.

[22] D. Takahashi, Implementation of Parallel 1-D FFT on GPU Clusters, in: Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering, 2013, pp. 174–180.

[23] C. Wang, S. Chandrasekaran, B. Chapman, cusfft: A high-performance sparse fast fourier transform algorithm on gpus, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 963–972.

[24] NVidia CUDA Fast Fourier Transform library (cuFFT 9.2), https://developer.nvidia.com/cufft (2018).

[25] clFFT: a software library containing FFT functions written in OpenCL, https://github.com/clMathLibraries/clFFT (2016).

[26] A. Watson, Image Compression Using the Discrete Cosine Transform, Mathematica Journal 4 (1) (1994) 81–88.

[27] M. P. V. Chauhan, P.K. Nathaney, K. Rai, A Novel Approach to Video Compression Technique using Variable Block Sizes in Motion Estimation Process, International Journal of Electronics and Computer Science Engineering (IJECSE) (2012) 1.

[28] Y. Wang, M. Vilermo, Modified Discrete Cosine Transform: Its Implications for Audio Coding and Error Concealment, J. Audio Eng. Soc 51 (1/2) (2003) 52–61.

[29] M. Guptda, A. Garg, Analysis of Image Compression Algorithm using DCT, International Journal of Engineering Research and Applications (IJERA) 2 (1) (2012) 512–521.

[30] M. Mathew, V. Bhat, S. M. Thomas, C. Yim, Modified mp3 encoder using complex modified cosine transform, in: Proceedings of the Internacional Conference on Multimedia and Expo, Vol. 2, 2003, pp. II–709–12 vol.2.

[31] J. R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, T. Wiegand, Comparison of the Coding Efficiency of Video Coding Standards - Including High Efficiency Video Coding (HEVC), IEEE Transactions on Circuits and Systems for Video Technology 22 (12) (2012) 1669–1684.

[32] C. G. Kim, Y. S. Choi, A High Performance Parallel DCT with OpenCL on Heterogeneous Computing Environment, Multimedia Tools Applications 64 (2) (2013) 475–489.

[33] B. Wang, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, An Optimized Parallel IDCT on Graphics Processing Units, in: Proceedings of the 18th International Conference on Parallel Processing Workshops, Euro-Par'12, 2013, pp. 155–164.

[34] M. Guptda, A. Garg, Analysis of Image Compression Algorithm using DCT, Journal of Engineering Research and Applications 2 (1) (2012) 512–512.

[35] T. Veldhuizen, C++ Templates as Partial Evaluation, in: Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), 1999, pp. 13–18.

[36] R. E. Ladner, M. J. Fischer, Parallel prefix computation, J. ACM 27 (4) (1980) 831–838.

[37] T. G. Stockham, Jr., High-speed Convolution and Correlation, in: Proceedings of the April 26-28, 1966, Spring Joint Computer Conference, AFIPS '66 (Spring), 1966, pp. 229–233.

[38] Maxwell Tunning Guide, http://docs.nvidia.com/cuda/maxwell-tuning-guide/ (2017).

[39] Kepler Tunning Guide, http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html (2017).

[40] GCN whitepaper, https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf (2012).

[41] AMD OpenCL opitimization guide, http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide.pdf (2014).

[42] The initial open source release of clFFT, https://github.com/clMathLibraries/clFFT/releases/tag/v2.0 (2013).