A Parallel FPGA-based FFT Architecture for High-Performance Hybrid Computing

Vitor C. F. Gomes
Andrea S. Charão
Universidade Federal de Santa Maria
vconrado, andrea@inf.ufsm.br

Haroldo F. C. Velho
Instituto Nacional de Pesquisas Espaciais
haroldo@lac.inpe.br

Abstract

Field Programmable Gate Arrays (FPGAs) have been increasingly used to improve performance of computational applications. In recent years, such devices have also been employed to build hybrid reconfigurable computing systems as the Cray XD1. Hybrid systems raise a number of challenges compared to FPGA-only designs, as for example the efficient data transfer between the CPU and FPGA, during computations. In this paper, we present an FPGA-based architecture for performing a parallel floating point Fast Fourier Transform (FFT) of one-dimensional data. FFT cores are broadly available for FPGAs, but mainly for accelerating FFT computation on hardware. In contrast, our design is targeted to a high-performance hybrid computing system where CPU and FPGA cooperate in calculating the FFT. It also explores design strategies for parallel FFT computation and overlapping computations with data transfers. To validate our approach, we present experimental results obtained on a Cray XD1 hybrid system, while performing floating point FFTs for varying input data sizes. The results show that our design is efficient in avoiding bottlenecks between memory and computing modules, and achieves speedups on the target hybrid architecture.

1. Introduction

In the past few years, hybrid reconfigurable systems have emerged as promising next-generation platforms for high-performance computing. Such systems connect general-purpose processors and modern Field Programmable Gate Arrays (FPGAs), usually through a custom interconnection technology. The main idea is to incorporate FPGAs as co-processors into high-performance systems, in order to accelerate computation-intensive parts of performance-critical applications. This approach has been investigated in some recent research works [26, 6, 27, 8], which explore the combined, parallel FPGA and CPU computing power.

Parallelism is a key issue while designing efficient applications targeted to hybrid reconfigurable systems. In such systems, there are multiple parallel granularities to explore, ranging from the fine-grain parallelism inherent in FPGAs to the coarser parallel grain of multiple interconnected CPU-FPGA nodes. In this work, we focus our attention on the parallelism provided by a single node containing both an FPGA and a general-purpose processor. Designing efficient applications for such platform is a challenge, because there is a number of factors limiting their parallel performance, namely the efficiency of data transfers between CPU and FPGA, the workload partition considering the hardware/software resources and the design choices for the FPGA-based architecture, which are application-dependent.

Some computational kernels are widely available as FPGA modules, as occurs with Fast Fourier Transforms (FFTs). This kernel is recurrent in many computation-intensive applications, from image processing to atmospheric simulation. The computational cost of an FFT motivates the parallelization of its computations, as well as its implementation on hardware. Significant speedups can be achieved for FFTs, mainly for two or three-dimensional transforms on distributed or shared-memory parallel architectures [14, 3, 19]. FPGA-based FFT modules also achieve high-performance and are suitable for a variety of FPGA-centric applications [17, 18, 7], but are not designed to explore parallelism on hybrid nodes and to cope with data transfer costs. For one-dimensional FFTs, speedups are harder to obtain due to the high cost of data transfer ($O(N)$), compared to its computational cost ($O(NlogN)$). In this case, the parallel efficiency is highly sensitive to the number of points on the input data, so it is beneficial for application developers/tuners to be able to reuse an FPGA-based FFT design for different sizes of input vectors, without needing to re-synthesize the hardware.

In this paper, we present a parallel FPGA-based FFT architecture aimed to efficiently exploit high-end hybrid computing systems. This architecture makes an aggressive use of design alternatives to hide data transfer costs and to keep the FPGA busy, working in parallel with the CPU, so the resulting schema leverages intra-node and intra-FPGA parallelism to solve the 1D FFT. Our design is based on floating-point arithmetic and can cope with a range of input data partition sizes. The CPU computes the other partition of the FFT input data and processes the final data synchronization. With such flexible design, experimental tuning and analysis require no effort of hardware reconfiguration.

To validate our approach, we implemented this architecture on a Cray XD1 hybrid system. Our implementation choices considered the different modes provided by the system for data sharing between the CPU and the FPGA. We analyzed the performance of our hybrid solution for varying input data sizes and with different workload partitions and mappings. The results show that our hybrid design achieves speedups compared to a CPU-only FFT implementation.

The rest of this paper is organized as follows. Section 2 briefly provides some background on parallel FFT algorithms and their implementation on FPGA. It also discusses some related work focused on hybrid designs for high-performance reconfigurable computing systems. Section 3 presents the target computing systems for this work and describes the communication modes provided for hybrid applications on the Cray XD1. Sections 4 and 5 present our hybrid architecture design and explain some implementation details. Section 6 describes our experimental analysis and discusses the results, while Section 7 presents some final considerations.

2. Background and Related Work

2.1. Fast Fourier Transform

Fourier transforms are linear transformations used in several scientific and engineering applications. In their discrete formulation, these transforms are usually the core of computational applications such as digital signal processing and solution of partial differential equations, just to name a few. The Discrete Fourier Transform (DFT) of a sequence of N numbers can be computed as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad k = 0, 1, ..., N-1 \quad (1)$$

where $W_N = e^{-2\pi\sqrt{-1}/N}$ is a trigonometric coefficient known as the twiddle factor. The Fast Fourier Transform algorithm (FFT) [9] computes DFT reducing the complexity from $O(N^2)$ to $O(NlogN)$.

There are many ways to structure the FFT algorithm. One variant is the radix-2 algorithm: it takes a divide-and-conquer approach, which operates on an N-point data set, where N is a power of 2. Its basic operation is known as “butterfly” and consists of two complex additions and a complex multiplication. The radix-2 algorithm yields the smallest butterfly unit, which allows for greater flexibility in studying the design space [2, 23, 9, 5, 18]. In this work, we chose a radix-2 butterfly in order to design a flexible hybrid solution for the 1D FFT, allowing for experimental analysis without hardware reconfiguration. We also focused on the 1D FFT because it may be used as a building block for higher-dimensional transforms.

There are also several ways to calculate an FFT in parallel. The binary-exchange algorithm can be used to minimize the communication [16]. Figure 1 illustrates an eight-point FFT computational flow with reordered inputs. This structure provides an ideal optimal-cost parallel time complexity of $O(logN)$ when computed with N processors. Each step of the FFT algorithm operates on points with increasingly distant indices. In the last step, radix-2 butterflies operates on points i and $(N/2) + i$, with i varying from 0 to $(N/2) - 1$. In case of partitioning the FFT algorithm in two parallel tasks (light gray and dark gray in Fig. 1), the last step of the FFT cannot be achieved without data movement between the two tasks. These data dependencies are found earlier in computational flow with greater number of parallel tasks. With this structure, data communication must occur in the $logP$ last steps, where P is the number of parallel tasks computing the FFT. When the communication cost is high, it may be advantageous to assign (N/P)-point FFTs to each task and perform the last steps sequentially on a single task.

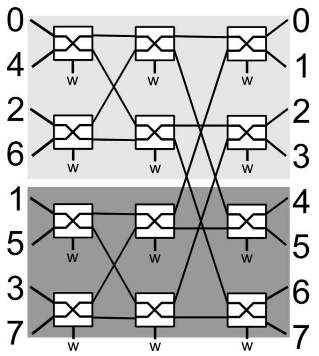

Figure 1. Computational flow for an 8-point FFT

2.2. FFT and Hybrid Reconfigurable Computing Systems

As a computationally intensive algorithm, FFT is suitable for hardware acceleration and FPGA-based implementation. Hardware FFT architectures have been extensively investigated in a number of research works [22, 18, 17, 24, 15] and are currently available as modules from different vendors [13, 1, 25]. These architectures differ on their goals and target applications. Some are focused on low resource usage and low power consumption, while others are concerned with high-throughput and high-performance. They also differ in terms of the FFT algorithm (radix-2 or higher order radices) and the hardware implementation strategies.

To achieve high performance and throughput, FPGA-based FFT architectures usually employ pipelined constructs to operate on continuous data streaming at clock rate [22]. When resource usage is a constraint, the so-called burst FFT architectures are predominant [22]. They process a data frame at a time, as the FFT unit only operates on another frame when the previous frame is finished. One can also consider a mixed strategy, which consists on adding functional and memory units to the burst architecture, in order to overlap computations with data I/O.

In hybrid applications for reconfigurable computing systems, the FPGA usually needs to read a fixed width input vector to compute the FFT. In this case, real-time data streaming is not a requirement for the FPGA-based FFT, but a burst architecture would restrict parallelism. Considering that we are interested on high-performance FFT computation, a mixed strategy is best suited for our case.

While FPGA-centric designs for the FFT are around for more than a decade, to our knowledge there is not yet a thorough investigation of hybrid designs that leverages the power of using both the FPGA and the CPU to compute this transform. This approach is rather recent, due to the new reconfigurable computing systems brought to the market over the past few years. Some related work on hybrid designs address different computational kernels and are limited to linear algebra operations and optimization problems [26, 6].

3. High-Performance Reconfigurable Systems

FPGAs have achieved sufficient gate density and functional capability to support high-performance floating-point operations required by many scientific kernels. In the past few years, high-performance computing vendors, like Cray, SGI and SRC, introduced hybrid computing systems such as Cray XD1, XT3/XT4 and XT5h, SGI RASC and SRC-6 MAP. These systems have been explored in some research works on hybrid computing [27, 20, 21, 26, 6]. In our work, we have used a Cray XD1 system to carry out experimental analysis, so the following sections present the main characteristics of its hybrid architecture.

3.1. Cray XD1

Our Cray XD1 system is made up of six interconnected nodes (blades), each one containing two AMD Opteron general-purpose processors and one Xilinx Virtex II Pro FPGA. Figure 2 shows the architecture of an XD1 node (blade). The reconfigurable device has direct access to four banks of QDR II SRAM. Through a RapidArray processor, the FPGA can also access the DRAM of the processors [10, 11]. While developing hybrid programs for the XD1, two key issues are moving data between the FPGA and the processors and efficiently using the memory hierarchy available to FPGA designs.

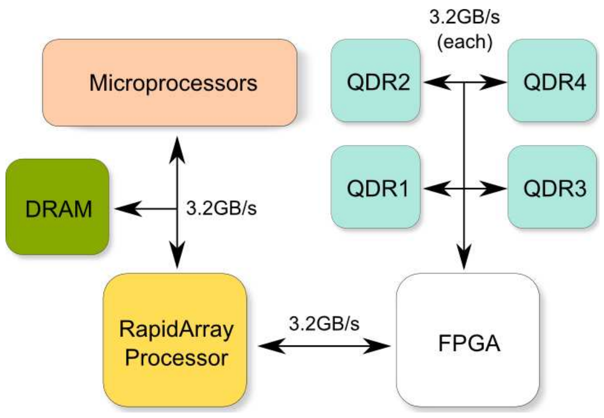

Figure 2. Cray XD1 blade architecture

3.2. Communication Modes

To allow communication between the FPGA and the CPUs, Cray provides the RapidArray Transport Core (RTCore) and the einlib library. This library provides a C-language interface for software applications to share memory blocks with the FPGA. These blocks are called FPGA transfer regions (FTRs). The library also allows to issue write and read requests from the application to the FPGA, at the same abstraction level of file reads and writes.

The RTCore provides a VHDL entity to handle requests from the C program to the FPGA. This core has two interfaces with distinct functions. The **Fabric Request** interface comprises the signals of processor requests and FPGA responses. Using this interface, the communication is managed by software logic and keeps the CPU occupied. The second interface, **User Request**, allows the FPGA to access the DRAM memory which is shared with the processors. After the software sends the shared transfer region address to the FPGA (using Fabric Request interface), it can read up to eight quadwords in a burst. The maximum allowed size for a transfer region is 2MB [12, 10]. This limitation may hinder the transfer of greater amounts of data, which need be done with FPGA-CPU synchronization or multiples FTRs. In a previous work [4], we analyzed both interfaces to measure the performance of each communication mode. For up to 16KB, the Fabric Request is faster than User Request. So, in the current work, we use the Fabric Request interface to communicate status registers with the FPGA, and the User Request interface to transfer the FFT data set. Another important reason for choosing the User Request interface to transfer large amounts of data is because processors are freed to perform other computing tasks during the transfer. The theoretical maximum bandwidth of RTCore is 1.6GB/s on each way. Despite that, in our previous work, we achieved a maximum bandwidth of 87MB/s using the User Request and 12MB/s using the Fabric Request interface to tranfer 2MB from CPU to FPGA.

3.3. Memory Hierarchy

The hybrid architecture of XD1 provides the FPGA with access to different memory levels. The DRAM is the upper level, largest memory bank and has a non-constant latency. This memory can be accessed using the User Request interface. The lower level memory comprises four banks of QDR II SRAM with 4MB each. The SRAM read latency is 8 cycles. Also, Xilinx Virtex-Pro FPGAs has an internal memory that can be accessed in one cycle latency. In this work, we used the three memory levels. The DRAM memory is used to transfer input data between the processor and the FPGA. Two banks of SRAM are used to store partial results of the computations and the twiddle factors. The internal memory of the FPGA is used to record the status, communication signals between architecture units and to implement a cache.

4. Hybrid Architecture Description

Our hybrid architecture aims to harness the power of both the CPU and the FPGA to compute the 1D FFT, so as to keep both devices fully occupied to ensure the performance of the hybrid execution. To do so, it aggressively explore multiple parallel and latency hiding opportunities in the hybrid system.

To parallelize the FFT, we use the binary-exchange algorithm and perform the task partitioning and mapping onto the processor and the FPGA. Each task calculates a range of the input data. We consider only a single node of the hybrid system, as we are primarily focused on parallel interactions between the CPU and the FPGA. At the end of computations, the processor performs the final steps of the FFT algorithm (cf. Section 2.1), to avoid the communication overhead of many data transfers.

To hide communication costs, the control of data transfer is left to the FPGA, that has inherent parallelism. This solution keeps the CPU free to collaborate on the FFT computation. The cost of communication handling by the FPGA is overlapped with the beginning of FPGA computations while the data transfer is processed. As stated in Section 2.2, our FPGA design uses a burst FFT strategy mixed with a streaming behavior [22], so the FPGA accepts multiple input data sizes. The workload partitioning between FPGA and CPU can be adjusted to optimize the load balance. This will be discussed in Section 6.1.

In our FPGA design, presented in Figure 3, we made many efforts to reduce the latency of memory accesses and the bottleneck of arithmetic kernels, by using prefetching, parallel data fetching and parallel floating-point computations. In the following paragraphs, we describe these operations considering the architectural components in Figure 3.

4.1. Communication Unit

The Communication Unit in Figure 3 is an interface between the Rapid Array Transport Core, that links the FPGA to the processors/DRAM, and the other FPGA blocks. This entity manages Fabric Request interface requests from software, which handle control configurations like FFT start and state registers. Furthermore, it is responsible for transfering the data set and twiddle factors from blade memory to FPGA QDR banks.

To overlap the data movement cost with computations, this unit provides the FFT Control Unit with a transfer state register which indicates the progress of the data copying. This technique allows the FFT Control Unit to begin the FFT computation while the whole transfer is being processed.

At the end of the transfer, this unit waits a signal from the FFT Control Unit that enables copying of the output result data back to the node memory. At the end of output data transfer, a state register is changed to indicate the end of FFT computation.

4.2. FFT Control Unit

The FFT Control Unit manages the FFT computation. It is responsible to address input data items to be computed in the Butterfly Cores Unit. It also gets the twiddle factor from Twiddle Factor Prefetcher.

Fig. 4 presents the pseudocode for the FFT Control Unit. This algorithm is similar to the one used in a software-based FFT computation. After each iteration of the main loop, the FFT Control Unit and the Butterfly Cores Unit are synchronized to avoid prefetching an element that is being calculated.

An important characteristic of this unit is that it does not wait for each butterfly computation to be finished. It overlaps the data fetching with butterfly operations. This approach keeps the Butterfly Cores Unit occupied as much as possible.

```
for step from 1 to n by 2
   for j from 0 to step by 1
      read twiddle factor
      for i from j to n by step*2
         read data[i]
         read data[i + step]
         send data to butterflies
      end for
   end for
   while butterflies not idle do
      nothing
   end while
end for
```

Figure 4. Pseudocode for FFT Control Unit

4.3. Butterfly Cores Unit

The Butterfly Cores Unit is the computational core of our architecture. This entity receives FFT Control Unit requests with two points and a twiddle factor to be computed. To perform parallel computations, this unit has multiple butterfly cores implementing radix-2 computations. These cores can operate in parallel, because there are not dependencies among butterflies in a single step of the FFT.

When a request is received from the FFT Control Unit, the data are delivered to a butterfly core to compute the operation, that provides two new points to be stored into QDR SRAM. Using multiple butterflie, this unit coordinates the task distribution among butterfly cores and forwards the computed points to the Data Item Handler.

4.4. Memory Controllers and Multiplexers

Our architecture has two memory control units, namely the Twiddle Factor Prefetcher and the Data Item Handler, whose main purpose is to reduce the latency of data read accesses.

The Twiddle Factor Prefetcher is able to load a memory position before it is requested by the FFT Control Unit. This is possible because the twiddle factor is organized in access order in the QDR bank. This technique reduces the read latency from 8 to 1 cycle.

The Data Item Handler manages the access to data items. Its goal is to improve the throughput of read accesses made by the FFT Control Unit. This requires two QDR banks allowing access to two memory positions in each request. Using this approach it is possible to feed the Butterfly Cores Unit avoiding idle butterfly cores.

As presented in Figure 3, there are three distinct units that need to perform read or write requests to the Data Item Handler. To allow these concurrent operations, we use two multiplexers controlled by the Communication Unit that change the data flow as necessary.

5. Implementation

We developed two cooperating programs to validate our FFT-hybrid architecture and to analyze its performance be-

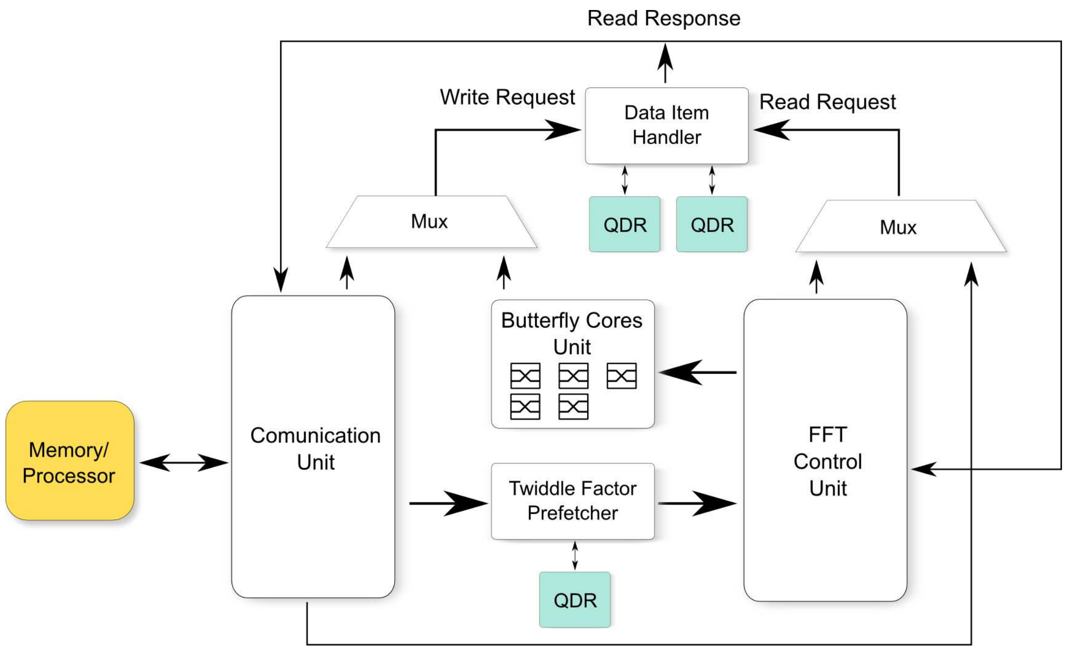

Figure 3. Block diagram of the parallel FPGA-based FFT

havior. They implement the FPGA-side and the CPU-side logic and computations. The following paragraphs describe these implementations.

5.1. FPGA-side

For the FPGA-side, the units described in section 4 were implemented using VHDL. The hardware description uses 64 bits to represent each data item, comprised of two 32 bits floating-point elements representing the real and imaginary parts of a complex number.

The Butterfly Core Unit was implemented with five butterfly cores. Each one performs a radix-2 computation using two complex adders and a complex multiplier. More butterfly cores are not currently possible because this configuration already uses 97% of the total area of the FPGA.

To distribute the tasks among the butterflies, all butterflies are connected in a task bus and each one has a busy flag that is snooped by the following butterflies. When a request arrives, each idle butterfly verifies if previous butterflies are occupied. If so, it gets the job and marks its status as busy. This approach simplifies the work distribution among the butterflies while avoiding the use of more area of the FPGA.

Each butterfly core uses six 32-bit floating-point adders and two multipliers to build two complex adders and a complex multiplier. This configuration was chosen to reduce the latency in computing the butterfly operation. For the floating-point operations, we used the Xilinx Floating-Point Core providing a 32-bit IEEE754 compliant implementation of adders and multiplies.

Our implementation of FFT in FPGA accepts a maximum of 2^{19} points. This limitation is due to the amount of QDR memory available to the FPGA. We further limit our hybrid implementation to 2^{18} points in order to avoid handling multiple FPGA transfer regions.

5.2. CPU-side

For the CPU-side implementation, we developed a C-language program to drive the hybrid execution, along with a software-only FFT for comparison purposes. This program is organized in four functions:

- *soft_fft*: this function implements the FFT as presented in the pseudocode in Figure 4 and it is used to compute the CPU task in the hybrid execution. This function is also used to perform a software-only FFT over the entire data set, for comparison purposes. Differently from the FPGA-based FFT, this implementation does not have input size limitations;
- *fpga_fft_start*: this function manages the FPGA-based FFT execution. It sends the FTR address and the input data size to the FPGA and changes a state register to indicate the start of FFT in FPGA;
- *fpga_fft_wait*: this function waits for the end of the FPGA-based FFT execution polling the state register of the FPGA;
- *soft_last_steps_fft*: computes the last steps of an N-point FFT in CPU. This functions is necessary to join the hybrid (CPU and FPGA) results.

While a FFT CPU-based execution calls only the function *soft_fft*, the hybrid execution calls all four functions, as follows: *fpga_fft_start*, *soft_fft*, *fpga_fft_wait* and *soft_last_steps_fft*.

6. Experimental Results

Our experimental analysis was carried out on a Cray XD1 system available at XXX[1]. We focused our analysis on a single node of the XD1 and used only a single processor, because we were mainly concerned with hybrid parallel interactions between CPU and FPGA. To synthesize, place and route the hardware description we used Xilinx ISE 10.1. For the C-program, we used GCC 3.3.3 compiler. Our hybrid design runs at 160MHz on the FPGA and uses a single CPU (2.4 GHz AMD Opteron) on a single Cray XD1 node.

We organized our analysis into two sets of experiments. In the first set, we investigated the workload partitioning between the CPU and the FPGA for our parallel 1D FFT, in order to determine task distribution schemes leading to high resource usage for both devices. In the second set, we compared our hybrid parallel solution to the CPU-only sequential version described in the previous section (*soft_fft*), so as to analyze the performance of our hybrid FFT implementation. The following paragraphs present and discuss these experiments in more detail.

6.1. Workload Partitioning

A simple way to partition a task within a node is to execute the computationally intensive part on the FPGA and use the processor only for control. However, in this case, the computing power of the processor is mostly wasted [27]. The workload partitioning between FPGA and general-purpose processors have to be addressed to fully utilize the devices. This is not a trivial job and it is an important decision from the performance point of view [21].

To investigate this issue for the 1D FFT, we measured execution times of our FPGA-based FFT implementation for increasing data sample sizes, up to the 2^{18}-point limit. It is important to notice that this FFT is mainly executed in

[1]Omitted to avoid identification of authors' institution

FPGA, the CPU just generates the twiddle factor and coordinates the start and the end of the computation. We also measured execution times for our CPU-only FFT implementation with the same data samples, up to 2^{20} (1,048,576) points (as the software implementation is not limited at the input data size).

With these results at hand, we determined the comparative experimental performance of both implementations for each data sample size. The results are presented in Table 1. The execution times shown in Table 1 can be used for granularity analysis and workload partitioning. The last column in that table presents the best workload partitioning between CPU and FPGA, chosen for further experiments. For example, when executing a hybrid $65,536$-point FFT, three quarters of the FFT are computed in the general-purpose processor and one quarter in FPGA, because its best partitioning scheme is closer than 3:1. To avoid the communication on the last two cycles, because of its high cost, the last two steps are processed in CPU (for this radix-2 algorithm). The number of last steps calculated in CPU is $log_2(P)$, where P is the number of partitions of the FFT input data. In this case, each partition means a N/P FFT.

Table 1. Workload partition

		Time (ms)		
log_2N	Input size	FPGA	CPU	Partition
10	1,024	0.94	0.19	3 : 1
12	4,096	3.45	0.79	3 : 1
14	16,384	14.68	4.33	3 : 1
16	65,536	63.76	20.59	3 : 1
17	131,072	132.73	47.59	3 : 1
18	262,144	276.76	191.25	1 : 1
19	524,288	–	461.59	1 : 1
20	1,048,576	–	979.56	3 : 1

In all results presented in Table 1, the CPU execution performed better than FPGA. However, there is an approximation of the performance of FPGA and CPU computation. It suggests that with greater number of points, the FFT execution on the FPGA could be faster than on the CPU. Furthermore, the execution time of FFT on FPGA has approximately linear growth.

6.2. Performance of the Hybrid FFT

Using the results presented in the previous section, we analyzed the performance of the hybrid FFT by means of the speedup factor between the hybrid and the CPU-only execution times. We considered data sample sizes from 2^{10} to 2^{20}. For each input data size, we used the best task partitioning scheme presented in the previous section. Table 2 presents our measurements and the speedup obtained for each input data size.

Table 2. Hybrid versus CPU-only results

		Time (ms)		
log_2N	Input size	Hybrid	CPU	Speedup
10	1,024	0.66	0.19	0.29
12	4,096	1.72	0.79	0.46
14	16,384	5.66	4.33	0.77
16	65,536	23.29	20.59	0.88
17	131,072	49.77	47.59	0.96
18	262,144	150.63	191.25	1.27
19	524,288	310.48	461.59	1.49
20	1,048,576	813.57	979.56	1.20

In these experimental results, we observe that the CPU-only version is faster for smaller data sizes, up to 2^{17} points. We also notice that speedups are achieved for our hybrid parallel FFT for 2^{18}, 2^{19} and 2^{20} points. The efficiencies for these executions are 64, 75 and 60% respectively. The decreasing speedup of the test with 2^{20} points can be explained by the task granularity. In this case, we used the 3:1 distribution between the CPU and the FPGA. A different distribution was not used because of the limited input memory size of our FPGA FFT implementation.

7. Conclusions and Future Work

In this paper, we have proposed an FPGA-based architecture for performing a parallel 1D FFT in a hybrid reconfigurable computing system. In our design, both the CPU and the FPGA cooperate to compute the FFT, contrasting with the usual approach were the FPGA response must overcome the CPU without considering data transfer costs.

Our key design decisions were to hide intra-node and intra-FPGA data movement costs as much as possible, so as to improve parallelism, and to partition the workload considering the comparative experimental performance between CPU and FPGA for the FFT. The resulting FPGA-based design is flexible in terms of the input data length. Moreover, our design allows for the data in a frame to be streamed to the FPGA without blocking the CPU. Our results show that this approach achieves speedups for a range of input data sizes and is able to harness the overall computing power of the hybrid system.

As future work, we plan to use higher-order radices for the FFT, considering that small data samples are unusual on real-world applications. This may reduce the resource usage on the FPGA, so that the remaining space could be used to accommodate more computing cores, leading to new alternatives for the parallel grain. Another future research direction concerns the workload partitioning, which is currently performed manually and could benefit from strategies aiming to reduce the burden from developers.

References

[1] 4DSP Inc. EEE-754 Floating Point FFT/IFFT IP core. http://www.4dsp.com/fft.htm.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-d fft. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 34–40, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] A. Ali, L. Johnsson, and J. Subhlok. Scheduling fft computation on smp and multicore systems. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 293–301, New York, NY, USA, 2007. ACM.

[4] Anonymous. Publication details omitted to hide authors' identity.

[5] J. H. Bahn, J. Yang, and N. Bagherzadeh. Parallel fft algorithms on network-on-chips. *Information Technology: New Generations, Third International Conference on*, 0:1087–1093, 2008.

[6] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan. Hardware/software integration for fpga-based all-pairs shortest-paths. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 152–164, Washington, DC, USA, 2006. IEEE Computer Society.

[7] C. Chao, Z. Qin, X. Yingke, and H. Chengde. Design of a high performance fft processor based on fpga. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 920–923, New York, NY, USA, 2005. ACM.

[8] D. Chavarría-Miranda, J. Nieplocha, and I. Gorton. Hardware-accelerated components for hybrid computing systems. In *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, pages 1–8, New York, NY, USA, 2008. ACM.

[9] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[10] Cray Inc. *Cray XD1 System Overview*. Mendota, MN, USA, 2005.

[11] Cray Inc. *Design of Cray XD1 QDR II SRAM Core*. Mendota, MN, USA, 2005.

[12] Cray Inc. *Design of Cray XD1 RapidArray Transport Core*. Mendota, MN, USA, 2005.

[13] Dillon Engineering, Inc. Fast Fourier Transform (FFT) IP Cores for FPGA and ASIC. http://www.dilloneng.com/fft_ip.

[14] M. Frigo and S. G. Johnson. Parallel FFTW. http://www.fftw.org/parallel/parallel-fftw.html.

[15] C. Gonzalez-Concejero, V. Rodellar, A. Alvarez-Marquina, E. M. d. Icaya, and P. Gomez-Vilda. An fft/ifft design versus altera and xilinx cores. In *RECONFIG '08: Proceedings of the 2008 International Conference on Reconfigurable Computing and FPGAs*, pages 337–342, Washington, DC, USA, 2008. IEEE Computer Society.

[16] A. Gupta and V. Kumar. The scalability of fft on parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(8):922–932, 1993.

[17] H. He and H. Guo. The realization of fft algorithm based on fpga co-processor. *Intelligent Information Technology Applications, 2007 Workshop on*, 3:239–243, 2008.

[18] K. S. Hemmert and K. D. Underwood. An analysis of the double-precision floating-point fft on fpgas. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:171–180, 2005.

[19] L. Johnsson, D. Mirkovic, R. Mahasoom, and F. Mwandia. Parallel fft. http://www2.cs.uh.edu/ mirkovic/fft/parfft.htm.

[20] V. Kindratenko and D. Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–22, Washington, DC, USA, 2006. IEEE Computer Society.

[21] V. V. Kindratenko, C. P. Steffen, and R. J. Brunner. Accelerating scientific applications with reconfigurable computing: Getting started. *Computing in Science and Engg.*, 9(5):70–77, 2007.

[22] J. M. Palmer. The hybrid architecture parallel fast fourier transform (hapfft). Master's thesis, Brigham Young University, 2005.

[23] D. Takahashi and Y. Kanada. High-performance radix-2, 3 and 5 parallel 1-d complex fft algorithms for distributed-memory parallel computers. *J. Supercomput.*, 15(2):207–228, 2000.

[24] J. A. Vite-Frias, R. d. J. Romero-Troncoso, and A. Ordaz-Moreno. Vhdl core for 1024-point radix-4 fft computation. In *RECONFIG '05: Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05) on Reconfigurable Computing and FPGAs*, page 24, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Xilinx Inc. Fast Fourier Transform. http://www.xilinx.com/products/ipcenter/FFT.htm.

[26] L. Zhuo and V. K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 2, Washington, DC, USA, 2005. IEEE Computer Society.

[27] L. Zhuo and V. K. Prasanna. Scalable hybrid designs for linear algebra on reconfigurable computing systems. *IEEE Trans. Comput.*, 57(12):1661–1675, 2008.