A Flexible Interconnection Structure for Reconfigurable FPGA Dataflow Applications

Gianluca Durelli, Alessandro A. Nacci, Riccardo Cattaneo,
Christian Pilato, Donatella Sciuto, and Marco D. Santambrogio

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy
{*durelli, nacci, rcattaneo, pilato, sciuto, santambr*}*@elet.polimi.it*

***Abstract*—Dataflow applications have proven to be well-suited for hardware implementation due to their intrinsic pipelined nature. Furthermore a wide range of algorithms, ranging from image analysis to map-reduce tasks, can be expressed using this paradigm. At the same time Field Programmable Gate Arrays (FPGA) start to be employed as hardware accelerators also in high-end systems coupled with General Purpose Processors (GPP). In this work we propose a programmable interconnection structure which permits to dynamically reconfigure the functionality of an FPGA implementing dataflow applications. A detailed analysis of the proposed solution shows that it is effectively able to increase the overall system flexibility helping in reducing the overall workload execution up to 25%, while at the same time reducing its variance.**

I. Introduction

Miniaturization of electronics components is pushing field programmable gate arrays (FPGAs) limits always forth. For instance, they contain enough reconfigurable logic to make it is possible to build large systems-on-chip (SoC), allowing for large and complex designs to be implemented directly on a single chip. Many high performance computing (HPC) systems (like Maxeler Ltd.'s workstations [1]) and everyday desktops and notebooks (like Apple's MacBook Pro, featuring a Lattice LFXP2-5E FPGA) make use of dedicated FPGAs to accelerate performance-critical portions of code of their applications or improve the overall performance/watt figure of the design.

Given the availability of these platforms, an open research issue is the crisp characterization of the class of algorithms that is convenient to implement in dedicated hardware, the most attractive feature of these platforms due to the intrinsically parallel nature of the reconfigurable logic. A widely accepted practical result is that any algorithm that may be implemented as a deeply-pipelined design made of highly predictable and small stages with a low amount of control logic is a good candidate to be efficiently mapped in hardware [2]. Accurate design of the hardware logic leads to designs that respect these conditions. Examples of applications successfully ported to hardware are image processing filters [3], digital signal processing frameworks [4], stock options pricing [5] and cryptographic algorithms, such as AES [6]. Additionally, some stages may be shared among different component libraries, leading to a scenario where it is feasible to partially reuse some hardware stages for different applications.

Designing and programming these architectures is not usually straightforward; for this reason, specific programming paradigms have been developed. Particularly relevant is the data flow programming approach [2], which particularly fits these scenarios. In the dataflow approach, applications are described by means of graphs of operations where each node executes a specific arithmetic function, getting data from the inputs and putting data on the outputs. Data "flows" from one node to the next, hence *dataflow* approach.

There are commercially successful platforms relying on the dataflow paradigm in the context of HPC; in this work we focus on the FPGA-accelerated, HPC workstations by Maxeler Ltd. These platforms allow the developer to accelerate his/her application by offloading the "hot code" to the FPGA in a suitable dataflow representation, while keeping the most "control intensive" portions of the application on the CPU. Data is moved from main memory to FPGA and vice-versa by means of an interface like PCI-e in a streaming fashion. The components realizing the various stages of the pipeline (called "kernels") are written in Java and may be packed into components libraries to improve code and component reuse. Moreover, in order to maximize the utilization of the reconfigurable logic and the overall system, we consider a scenario where many different instances of these hardware libraries are instantiated (possibly at runtime) and invoked by the host code at different times and in an unpredictable order. In this situation, it is almost impossible to determine *a priori* which are the optimal connections among the various kernels and the endpoints to/from the host machine, thus requiring a means for redirecting traffic. Even though Maxeler provides a manager to realize such connections, this must be configured at design time and might not be subsequently modified at runtime. Of course, since the FPGA is reconfigurable, it is possible to devise a solution where all the connections among the various kernels are reconfigured to accomplish the new users' needs, but the amount of logic to reconfigure might be non negligible, thus introducing large reconfiguration overheads. Moreover, partial bitstreams of all the possible configurations must be anticipated at design time. To cope with this kind of issues, a typical solution in the context of embedded systems is the introduction of a interconnection infrastructure, called crossbar [7], [8], which is a hardware switch capable of connecting each of the N inputs to one of its M outputs.

The contribution of this paper is the realization of a dataflow oriented $N \times M$ crossbar scheme aimed at dynamic

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.127

reconfiguring interconnections between cores at run-time in order to support the instantiation of hardware computation pipelines among different kernels even when they have not been foresee at design time. We provide details about our implementation and analyze a case study showing how this interconnection component provides up to a 25% speedup in workload execution time while decreasing, at the same time, its standard deviation leading to more predictable executions.

The rest of the paper is organized as follows. Section II introduces some of the most relevant related works in this field of research. Then, Section III introduces the proposed solution and discusses some alternatives, along with their drawbacks, while Section IV presents the details of the devised architecture, and Section V analyzes the overhead introduced by the interconnection component in term of area occupation an frequency. Subsequently, Section VI presents a case study involving an image processing pipeline, while Section VII analyzes the results of introducing our interconnection component when designing a hardware implementation for the presented case study. Finally, conclusions and future directions of work are given in Section VIII.

II. Related Work

Much research focuses on the design of interconnection components between IPs in an embedded system, whereas dynamic connection of multiple kernels in a HPC and dataflow context is a pretty novel research. In this section, we compare our solution to some of the most relevant IP-interconnection related methodologies in literature.

A straightforward means for connecting different IPs to the host system is a bus-based solution like ARM AMBA/AXI [9], [10] or IBM's CoreConnect. The main drawback of these solutions is that they share the same bus signals to route traffic among all the IPs connected to the bus, thus creating a bottleneck whenever the pipeline is "warm" (i.e. when all the cores are moving data among them). This scenario requires to account for a bus whose peak throughput is the sum of the peak throughputs of all the cores plus the arbitration traffic. This is unnecessary since dataflow applications are highly predictable, with a precise ordering among the invoked kernels, and the traffic flows constantly from one single kernel to another, so a point-to-point topology is more appropriate.

A relevant field of research in IP interconnection schemes is that of Networks on Chip (NoC) architectures [11]–[13], where communication among different processors is a main issue for which dedicated solutions have been proposed. The main concept underlying NoCs is the presence of a specialized network for routing data between all the IPs in the system [11]. They differ from common bus-based communication architectures since the dedicated routing networks explicitly recognize the peculiarities of different aspects in the communication process between IPs. This leads to a specialization of the interconnection network, ultimately yielding better scalability and flexibility in the presence of a large number of cores. Many routing schemes exist in order to minimize different metrics like power [14], routing algorithm complexity [15], peak traffic disparity between network links [16], latency and available bandwidth [17]. NoCs represent state-of-the-art solutions to the intercommunication issue and scale very well with the number of IPs [12], but in the context of HPC-oriented dataflow architecture they have the drawback of adding an unnecessary overhead to the amount of routed data – the routing header – that our solution does not incur in. In fact, in a static dataflow architecture the IP interconnection scheme – namely, the call graph – is known at the beginning of the computation and does not change until its end.

More relevant to the embedded systems domain is the crossbar component [7], [8], [18], which is a network topology where data is forwarded from N sources to M destinations in a straightforward way. In [7] the authors implement the crossbar mechanism as a means for connecting all the inputs to all the outputs, demonstrating effective scalability to up to 9 connected nodes. Differently from their design, we do not require to explicitly state many low level details (like the width of the data bus) since this is taken care of by Maxeler's toolchain and is implicit in the Java code written to describe the pipeline stages. Moreover, they do not explicitly consider the problem of management of data flowing from one core to the other, and the problems associated with the memorization of intermediate values. Also in [8], [18] a NoC crossbar communication infrastructure is presented. However they cannot be effectively used in a completely pipelined solution since the overhead introduced by the NoC management. In the next sections we will show how to properly address these issues.

III. Proposed Approach

The basic idea behind this work is to obtain a flexible and reconfigurable interconnection structure for FPGA dataflow applications. The dataflow programming model of computation describes the computation structurally (computing in space) rather than specifying a sequence of processor instructions (computing in time). In other words, the dataflow model of computation focuses on the flow of data within the applications. A dataflow architecture is a computer architecture where the computation is performed directly by a set of parallel processing elements connected each other through some kinds of communication channels. Summarizing, the main components of a dataflow architecture are the *processing elements* (PEs) (usually organized in a pipeline) and the *communication infrastructure*. This work aims to introduce the concept of flexibility into such kind of systems focusing on the latter: the communication infrastructure.

In this work, the idea of flexibility concerns with the possibility of changing at run-time the functionality of the pipeline of the dataflow application. For instance, let us consider the architectures shown in Figure 1 where the first functionality is composed of the processing elements PE1-PE2-PE3-PE4 and the second one is composed of PE1-PE2-PE5-PE6.

In order to allow to switch between the two functionalities and, at the same time, reuse the first two components (i.e., PE1 and PE2), a reconfigurable communication element must be introduced inside the overall architecture: this element must provide a simple interface to rapidly change (in few clock cycles) the communication flow of the pipeline. The

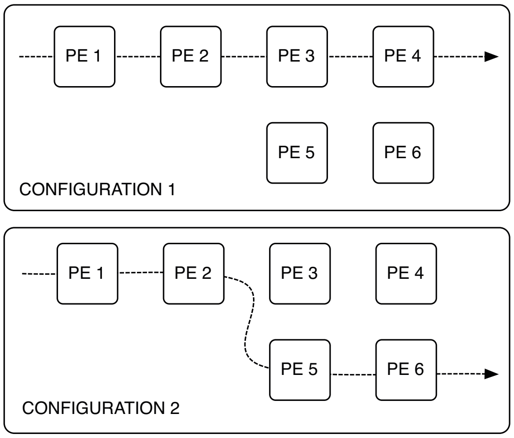

Fig. 1: Pipeline reconfiguration example

approach proposed in this paper is thus to introduce such a component, namely a *Reconfigurable Interconnection Element* (as shown in Figure 2) that allows to dynamically select at runtime the desired pipeline structure. Considering the previous example, to realize the two configurations shown in Figure 1, the resulting *Reconfigurable Interconnection Element* must be configured with the schema proposed in Figure 3.

In particular, the proposed approach aims at realizing a reconfigurable ***pipelined*** interconnection element that can be easily configured via ***software*** (through standard CPU directives). In fact, to be software configurable is relatively simple: just a simple API must be designed. Moreover, having a software interface seems to be a reasonable solution since CPUs are usually available on dataflow architectures for at least two main reasons: first, computation is generally hybrid (imperative and dataflow); second, a controller is needed in any case to manage the data transfers with the memory and the dataflow processing cores.

On the other hand, in order to design a ***pipelined*** interconnection element, it is necessary to take into account some details. First, since we are targeting the dataflow class of applications, to avoid to empty the pipeline, it is important not to stop the computation and the flow of data while the reconfiguration process is performed. Second, it is necessary to correctly balance the overall architecture placing FIFOs among the stages to manage unbalanced pipelines. Finally, the goal of the proposed approach is to realize a flexible interconnection element that is able to work with general purpose dataflow processing elements, for instance, supporting different input/output data size.

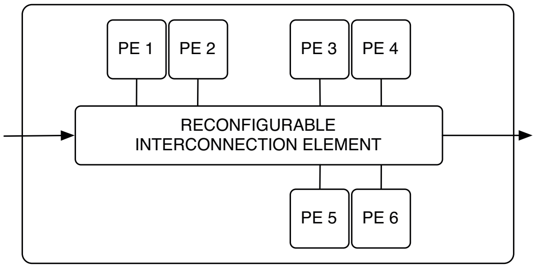

Fig. 2: Reconfigurable Interconnection Element example

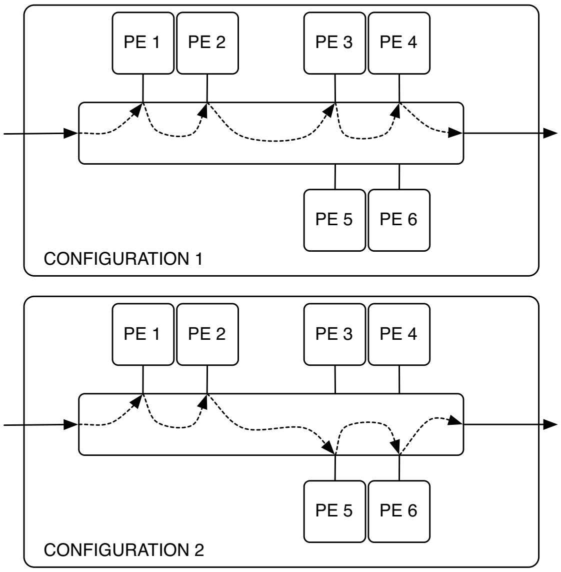

Fig. 3: Reconfigurable Interconnection Element example

As presented in the next section (Section IV), the proposed approach has been implemented and tested on the Maxeler HPC Dataflow platform, that is a commercial platform for computationally intensive applications. Here, a pipelined and software-reconfigurable interconnection element has been deployed on FPGA. The results of the proposed approach is then discussed in Section V.

IV. Implemented Architecture

The proposed approach presented in Section III has been designed, deployed and tested on the Maxeler High Performance Computing (HPC) Dataflow platform. Maxeler Technologies Ltd. [19] is a company that produces High Performance Computing (HPC) systems that use the dataflow paradigm in order to speed up the computation. More precisely, the key factor of the Maxeler platform is the interaction between a standard Intel x86_64 CPU and a so called *Data Flow Engine* (DFE) which is the board used to exploit hardware acceleration, connected through a high speed PCI Express interface. Maxeler's DFEs are composed of one or more Xilinx's FPGAs, and they represent the portion of the system devoted to the actual hardware computation. In Maxeler's platform, FPGAs are employed as *static* hardware accelerators that are programmed with the dataflow paradigm to obtain huge speed-ups especially on streaming applications. Programming an application for a Maxeler platform requires to write two different types of code:

- The *host code*, which runs on the *CPU*;
- The *dataflow code*, which runs on the *DFE*.

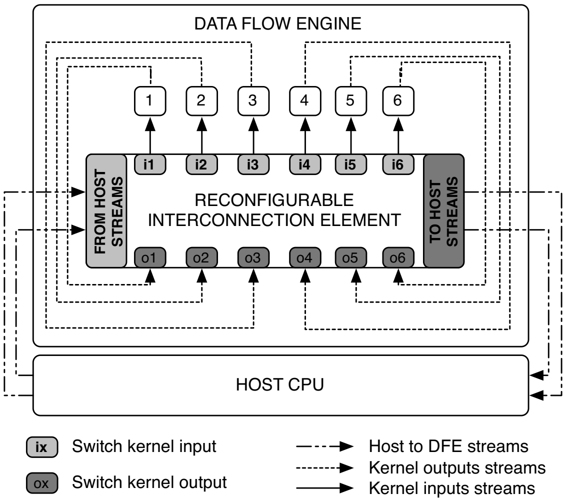

Fig. 4: Pipeline reconfiguration example

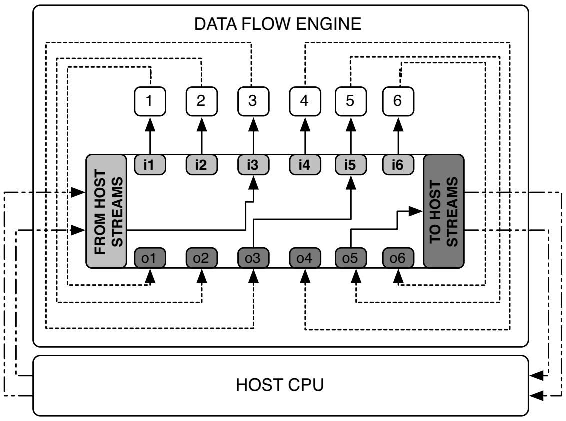

Fig. 5: Interconnection Example

The host code can be written in either C/C++ or FORTRAN, while the dataflow code must be written in Java. A logical DFE is composed of two main elements:

- *Kernels* are hardware data-paths implementing the arithmetic and logic computation needed within the algorithm. They perform all the actual computation inside the DFE;
- *Manager* comprises all the logic that manages the data flow between Kernels and off chip I/O in the form of streams. Its goal is to manage the connections among the Kernels, and among the Kernels and the host application.

Since all the computing elements inside a DFE must be kernels, and since some computation must be performed by the *Reconfigurable Interconnection Element*, this component has been realized as a generic kernel inside the overall architecture. The example reported in Figure 4 will be used as reference within this section. Precisely, Figure 4 shows a generic Maxeler architecture with 6 kernels all connected to the central *Reconfigurable Interconnection Element*. The example architecture presents also two input streams from the host and two output streams to the host. As it is shown, the input and the output of each kernel is connected to the interconnection element.

To obtain a flexible and reconfigurable component, the *Reconfigurable Interconnection Element* is widely parametrizable. Particularly, it contains both static and dynamic parameters. The static parameters must be set at design time; the dynamic parameters have to be set at run-time and allow to effectively have the run-time communication reconfiguration.

Static parameters are used to set the maximum size of the *Reconfigurable Interconnection Element*, so the number of maximum kernels that can be connected and the number of input/output streams from/to the host that can be managed. The dynamic parameters can be changed at runtime from the host with simple C function calls and are used to manage the data flow inside the pipeline. The dynamic parameters are:

- `data_size[]`: size of the data that has to be read and write from/to host;
- `kernel_ticks[]`: kernel tick from which the stream becomes active (for each kernel);
- `interconnection_vector[]`: switch logic, i.e., the internal interconnection structure. It is an array that specifies which output is connected to which input.

These dynamic parameters are simply represented with arrays since the hardware components inside the Maxeler platform are described with Java.

The first parameter, `data_size[]`, is necessary since each kernel in the Maxeler platform must be aware of the number of data it has to process: in fact, a kernel becomes active when the first valid data to process arrives as input and remains active just for the size of the input data block. It is an array because this parameter must be specified for each kernel connected to the interconnection element.

The second parameter, `kernel_ticks[]`, is an array that represents the number of ticks to wait in order to obtain a valid value from the *i-th* kernel. In order to well understand the meaning of this value, it is necessary to explain that in the Maxeler architecture, a kernel becomes active only if all its inputs are valid; otherwise it is in idle state without producing any output. This property brings to an issue that must be solved: let us consider the Figure 5 where an instance of a generic dataflow architecture with *Reconfigurable Interconnection Element* is represented. In this example, the data flow follows this path: *Host - Kernel_3 - Kernel_5 - Host*.

It is clear that the *interconnection element* will be inactive until the inputs *o3* and *o5* are valid (since they are inputs for the interconnection element). Anyway, this inputs cannot be valid until the *Kernel_3* computes its output values. Unfortunately, *Kernel_3* will never start to compute since it will not receive any input from the inactive *interconnection element*. This leads to a *dead-lock* state. To solve this problem, all the inputs

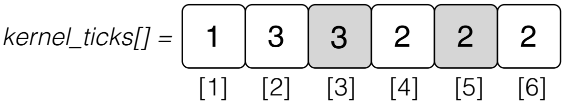

Fig. 6: Kernel ticks array example

of the *interconnection elements* are ignored by default using a multiplexer. In this way, the *interconnection element* starts in active state. Using a counter and the `kernel_ticks[]` parameters is finally possible to understand when to select the input coming from a connected kernel and when ignore it. For instance, let us consider again Fig. 5 and also Figure 6 where a possible instance of the `kernel_ticks[]` array is represented. Since only *Kernel_3* and *Kernel_5* are connected in the example, the data we need to care about is that *Kernel_3* takes 3 ticks to execute its task and that *Kernel_5* takes 2 ticks. Knowing this information and using a counter, now it is possible to know that the *interconnection element* has to consider the input coming from *Kernel_3* and *Kernel_5* after 3 ticks and after 5 (3+2) ticks respectively.

The last dynamic parameter is an array that describes the interconnection schema of the element: the `interconnection_vector[]`. Every stream in the architecture is associated with an identifier value (*stream_id*). This *stream_id*s are used to correctly map the data stream inside the reconfigurable interconnection element. For instance, let us consider Figure 8 where an instance of the `interconnection_vector[]` that implement the interconnection of Figure 5 is presented. Each cell of the vector represents which is the input stream of the *i-th* kernel. The first *n-th* cells of the array represents the output streams to the host (where *n* is determined by a static parameter).

The following sections will provide a detailed analysis of the hardware implementation and will show the case of study, along with the obtained experimental results.

V. Hardware Implementation Analysis

The proposed solution has been implemented for the Maxeler HPC computing platform and exploiting their automated design flow. Accordingly to this flow, we specified the system described in Section IV, logic and interconnections, in a Java-like language and then the synthesis has been carried out automatically by Maxeler High Level Synthesis tool (HLS), namely the *MaxCompiler*. This choice greatly simplified the design of the system, however since the design process is automatically done by the tools, the final solution may not be the best one in terms of performance. However, hardware design is a complex task and other manufacturers, such as Xilinx, are proposing HLS tools (see Vivado HLS [20]) to implement hardware architectures; we believe that in order to develop scalable and reusable cores, HLS tools will play a fundamental role in future years and results here illustrated may be improved by newer tools. Since the proposed architecture is a configurable $N \times N$ crossbar, it suffers from the same limitations, basically the area increases proportionally to the number of inputs and maximum frequency decreases accordingly. We analyze these two parameters implementing different architectures by varying the number of kernels and communication channels with the host. The architectures have been implemented in two forms: the former is the basic system where each host channel is connected to a kernel while the latter makes use of the proposed interconnection element. The kernels used in this test implement only an add operation; this choice is justified by the fact that in order to analyze the maximum bandwidth achievable by the proposed interconnection element we must be sure that other kernels do not have long combinational paths.

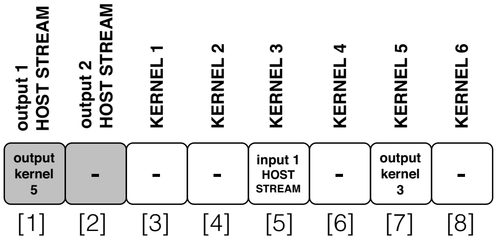

Fig. 8: `interconnection_vector` example

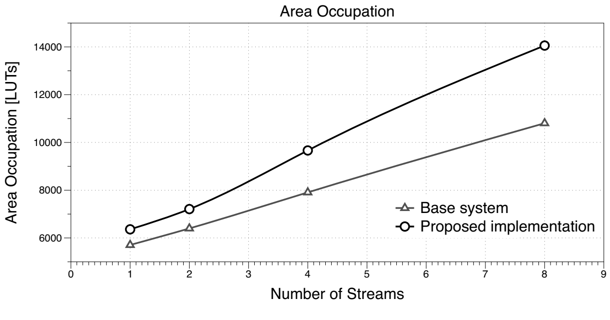

Fig. 7: Area comparison

Results about area occupation are reported in Figure 7 and, as expected, the area overhead increases with the number of controlled streams, this is not surprising since in a $N \times N$ crossbar switch the area should be proportional to N^2. However, considering the large amount of logic available in modern FPGAs, this is not a limitation. For example, in our environments, we used a Xilinx Virtex6 XC6VSX475T FPGA with 297,600 LUTs and the biggest overhead in the figure (i.e.,

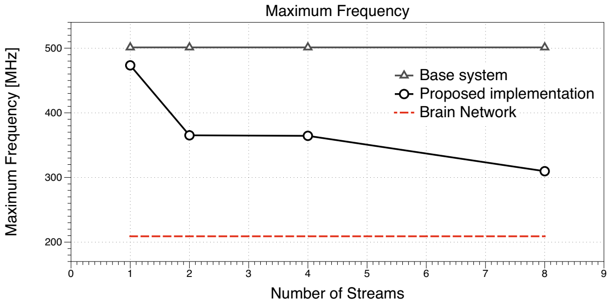

Fig. 9: Maximum Frequency Comparison

3,000 LUTs) roughly corresponds to 1% of the total space.

Data about maximum frequency achievable with our solution in comparison with the basic system is reported in Figure 9. As we can expect, the maximum frequency decreases as we increment the number of controlled inputs because of the switching logic, while in the basic system it remains the same, there are N parallel pipelines in the system and they do not influence each other. The basic system has a constant maximum frequency of 503MHz, while our implementation ranges from 475MHz, when controlling only one stream, down to 309MHz when controlling 8 streams.

It is worth noting that these examples have been crafted to analyze the overheads and that, when we implement a real application, the kernels are much more complicated. For a fair comparison, we reported in the figure also the frequency of a real complex application implemented by Maxeler which is a *Brain Network* image analyzer running on their platform. As we can see our reconfigurable interconnection will not influence the *Brain Network* simulator since it is capable of achieving a higher maximum frequency. This result cannot be obviously obtained in all the cases, but we will show in Section VII that our solution is effective not only with respect to the application execution time, directly influenced by the frequency, but also to other metrics such as the variance in the execution time.

VI. Case Study

This section describes the case study we used to evaluate our solution. As described before, our reconfigurable interconnection structure can be configured to change the connections between the cores on the hardware device in order to implement different pipeline structures without reconfiguring the logic of the device. First of all, a requirement for all the applications that our interconnection supports is that they can be expressed in a dataflow fashion. This is not a limitation since nowadays a large number of applications have been expressed in this paradigm in order to exploit instruction level parallelism in the pipeline. As an example, applications that can be represented with this paradigm are image and audio processing filters, cryptographic algorithms such as *Advanced Encryption Standard (AES)* [6] and map-reduce algorithms, as recently suggested by some researchers [21]. To keep the analysis more simple we concentrate on the case of the image processing pipeline which permits to illustrate benefits and limitations of our solution, this analysis can then be generalized to other scenarios or their combination as well. We now introduce the case study of the image processing pipeline, detail the applications we intend to be able to implement and illustrate the architectures we designed to perform comparisons illustrated in Section VII.

A. Image Processing Pipeline

Image processing applications can be naturally expressed as a pipeline. Each stage of the pipeline implements a single filter and the application output is derived by the sequential application of all the filters in the pipeline to the original input. Furthermore the filters can be differently combined to obtain different results, or they can be omitted from the pipeline when they are not needed. For instance if we want to perform noise reduction on an image we will employ a *Gaussian Blur* filter, but if we are interested only in the color intensity, since it is the component carrying the largest information, we may apply a *Gray Scale* filter before reducing the noise in the image; this last assumption is true only when the initial image is not already in a *Gray Scale* form. By this simple example we already derived three different cases where the application pipeline is function of the application goal and its inputs, but the same filters are employed. This flexibility in the implementation of image pipelines starting from the same kernels justifies the adoption of the proposed interconnection component.

B. Applications

The applications we used as test cases are simple image filters that can be differently combined to obtain different results, or the same result starting from different inputs. We divided these applications in 4 classes each one identified by the length of the corresponding pipeline (from 1 to 4 stages). Applications in the first class are the basic filters that are combined in different ways in the other classes.

Class 1:

- Gray Scale (GS): Converts an input image in its gray scale representation; its input is a RGB image.
- Gaussian Blur (GB): Performs the noise reduction on an input image; its input is a gray scale image or a RGB image.
- Edge Detection (ED): Performs the edge detection on the input expressed as a gray scale image.

Class 2:

- GS+GB: Applies GS and GB filters sequentially; its input is a RGB image and the output is the correspondent gray scale image with reduced noise.
- GS+ED: Applies GS and ED filters; its input is a RGB image, while the output image shows only the edges.
- GB+ED: Applies the two filters on a input in a gray scale format and obtain the binary image showing the edges.

Class 3:

- Canny (GS+GB+ED): This is a known variation of the edge detection filters which relies on the fact that the most interesting information to find the edges resides in the color intensity (gray scale representation) and that the ED filter is influenced by the noise in the original image. Canny filter then implements a pipeline of GS+GB+ED to obtain a better edge detection.

Class 4:

- GS+GB+GB+ED: This is a variation of the previous algorithm where the noise reduction step is performed twice. The amount of noise rejected by the GB filter is proportional to the size of the filter which results in a larger area occupation for a dataflow implementation. However, it has been demonstrated that iteratively applying a smaller filter to the same image can improve the amount of rejected noise [22]. This filter is then a canny variant with a better noise reduction.

C. Architectures

When it comes to implement a FPGA hardware architecture for the image processing pipeline we basically have 4 choices:

A) The first architecture consists in placing all the filters on the FPGA without considering their possible connections at run time. If an application needs to connect two or more filters they are simply executed iteratively. Our implementation of this solution is represented in Figure 10 where two copies of each filter are instantiated.

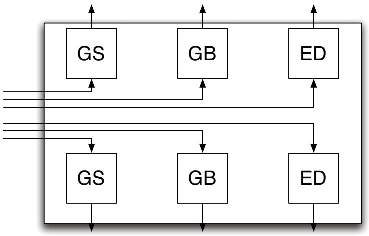

Fig. 10: Architecture A

B) The second choice consists in having the previous solution as the main case and to exploit the FPGA reconfiguration capability to implement more complex filters. For instance in our example we have the Canny algorithm which is a widely known application so we decided to realize an architecture optimized for this purpose and combine its utilization with the previous one; the architecture specialized for the Canny algorithm is represented in Figure 11.

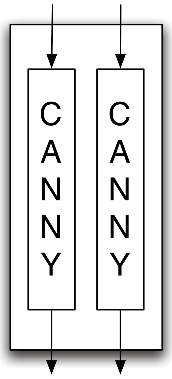

Fig. 11: Architecture B

C) The third solution consists in implementing both A and B without the need for reconfiguration. Given the large amount of space available on recent FPGAs there should be no problems related to number of LUTs needed to implement the functionality; however problems can appears when we analyze the interconnection between the FPGA and the source providing data. FPGA I/O pins are generally limited and it may not be possible to place a large number of independent components on the FPGA, at some point we will need to serialize the inputs to be able to connect all the filters. In our platform we are forced to a maximum of 8 input streams; the corresponding solution is reported in Figure 12.

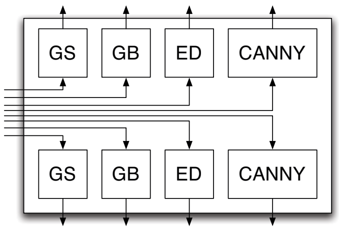

Fig. 12: Architecture C

D) The last option we have considered consists in implementing the architecture A where the filters are connected by means of our interconnection structure. This architecture is reported in 13.

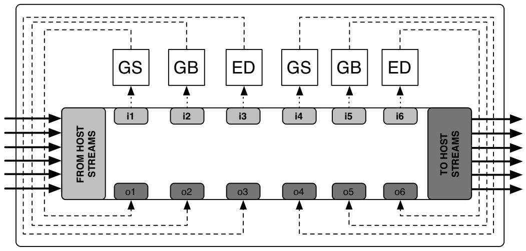

Fig. 13: Architecture D

VII. Evaluation Results

This section compares the performance of the architectures implemented using the applications described in Section VI analyzing both benefits and limitations of using our reconfigurable interconnection structure.

A. Single Application performance

The first experiment wants to compare how the different architectures behave when they have to execute an application from the classes used as case study. This experiment wants to show the benefits and the overheads that our solution introduces. For all the four classes of applications that we analyzed, we executed real tests on the target platform considering three different input sizes: small (1MB), medium (10MB), large (100MB). We collected the total execution time for every instance by averaging over 100 identical executions to reduce the measurement error. In fact, a measurements variance is introduced by the operating system that is responsible for HW/SW data transfer. Architectures A and B show the same behavior for the classes 1, 2 and 4 since, in these cases, the application from A is used; for the class 3 the architecture tailored for Canny algorithm is used instead, the reconfiguration overhead is not taken in account in this experiment. Figures from 14 to 17 show the normalized execution time (with respect to the architecture B) of the architectures presented in Section VI-C.

Figure 14 shows the execution time for a pipeline composed of a single kernel. As expected, architectures A and B achieve the best performance, while architecture C and D have an

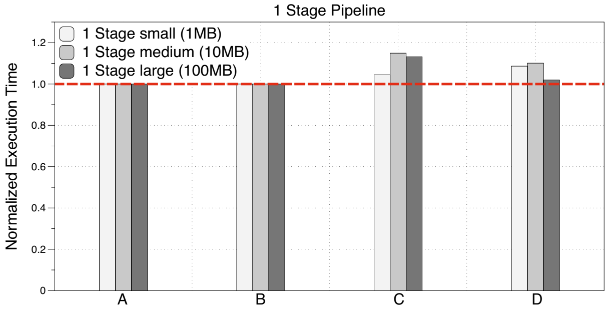

Fig. 14: Class 1 applications

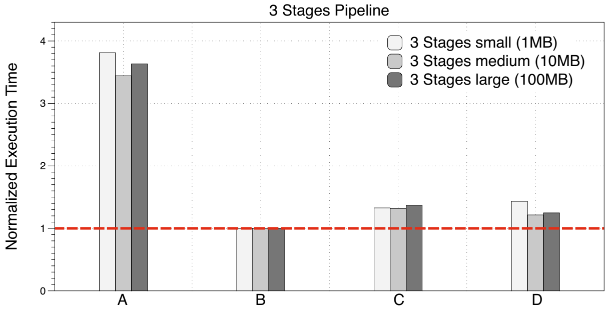

Fig. 15: Class 3 applications: Canny

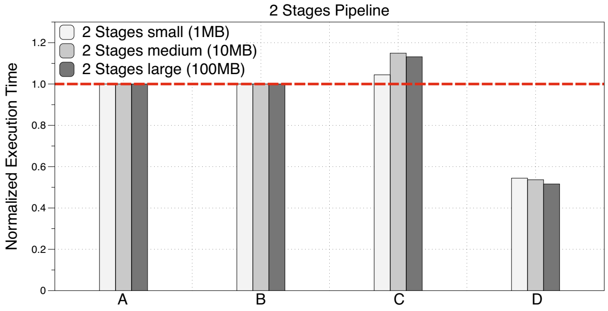

Fig. 16: Class 2 applications

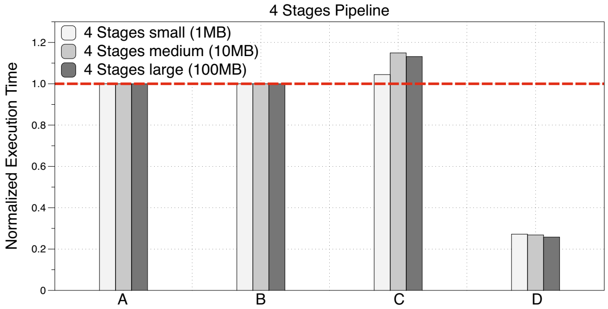

Fig. 17: Class 4 applications

overhead around 10-15%. About C, the overhead is due to the management of a larger number of input streams (this behavior is present also in other tests). Instead the architecture D (the solution proposed in this paper) has the drawback of increasing the pipeline stages; when the data enters our architecture, it passes through the interconnect module (1st stage), goes to the filter (2nd stage) and finally passes again in the interconnection module (3rd stage).

Figure 16 depicts the execution time when executing an application composed of a 2-stage pipeline. None of the architectures from A to C can implement this pipeline, so they need to transfer all the data from the CPU to the FPGA to compute the first stage, then transfer the data back, and finally send it again through PCI express link to compute the second stage. On the contrary, our solution simply configures the interconnections to implement a 2-stage pipeline, avoiding to transfer the data over PCI multiple times. In this case, our solution provides a speed-up around 2x.

Figure 15 represents the execution time for a 3-stage pipeline which in our case is the Canny application. Architecture B clearly achieves the best results in this situation since its logic is tailored for this application. Architecture A needs to execute 3 times resulting in a 350% overhead. Architectures C and D present similar results resulting in a 10-20% overhead. The motivation of this overhead is the same as the first case, but here our solution (architecture D) has implemented a 3-stage pipeline being able to obtain results comparable with other architectures having that pipeline statically configured. Furthermore the overhead in this case has to be further analyzed considering the reconfiguration time that architecture B has to suffer to change its configuration if needed.

Finally, Figure 17 shows the case of a 4-stage pipeline. As in our second case, none of the architectures has such a pipeline. Therefore, in this situation, our solution obtains a 4x speed-up since it can implement such pipeline when requested without applying reconfiguration or performing multiple executions.

With this experiment, we showed that our solution for configuring interconnections at runtime can realize the best pipeline and greatly improve the performance of an application. In particular, this is interesting when the pipeline is not statically available and it allows to maintain an acceptable overhead when ad-hoc architectures are available.

B. *Workload performance*

In this second test, we want to analyze the impact of our solution when used in a system with the goal of executing a workload composed of a mix of applications. Every mix consists in 50 applications randomly chosen from the one presented in Section VI-B and we randomly generated 1,000 different workloads.

In order to gather results about workload execution time, we must be able to integrate the architectures with the host CPU. Since designing a Runtime Manager able to schedule pipelines on a hardware architecture is beyond the scope of this work, we designed a simple simulator to gather data needed for this experiment. The simulator schedules applications on the hardware trying to extract the maximum degree of parallelism and starts to execute applications in the same order they appear in the workload mix. The simulator can be configured to schedule at most N applications at the same scheduling step. The simulator estimates the execution times using the data extracted from the real architectures provided in Section VI-C.

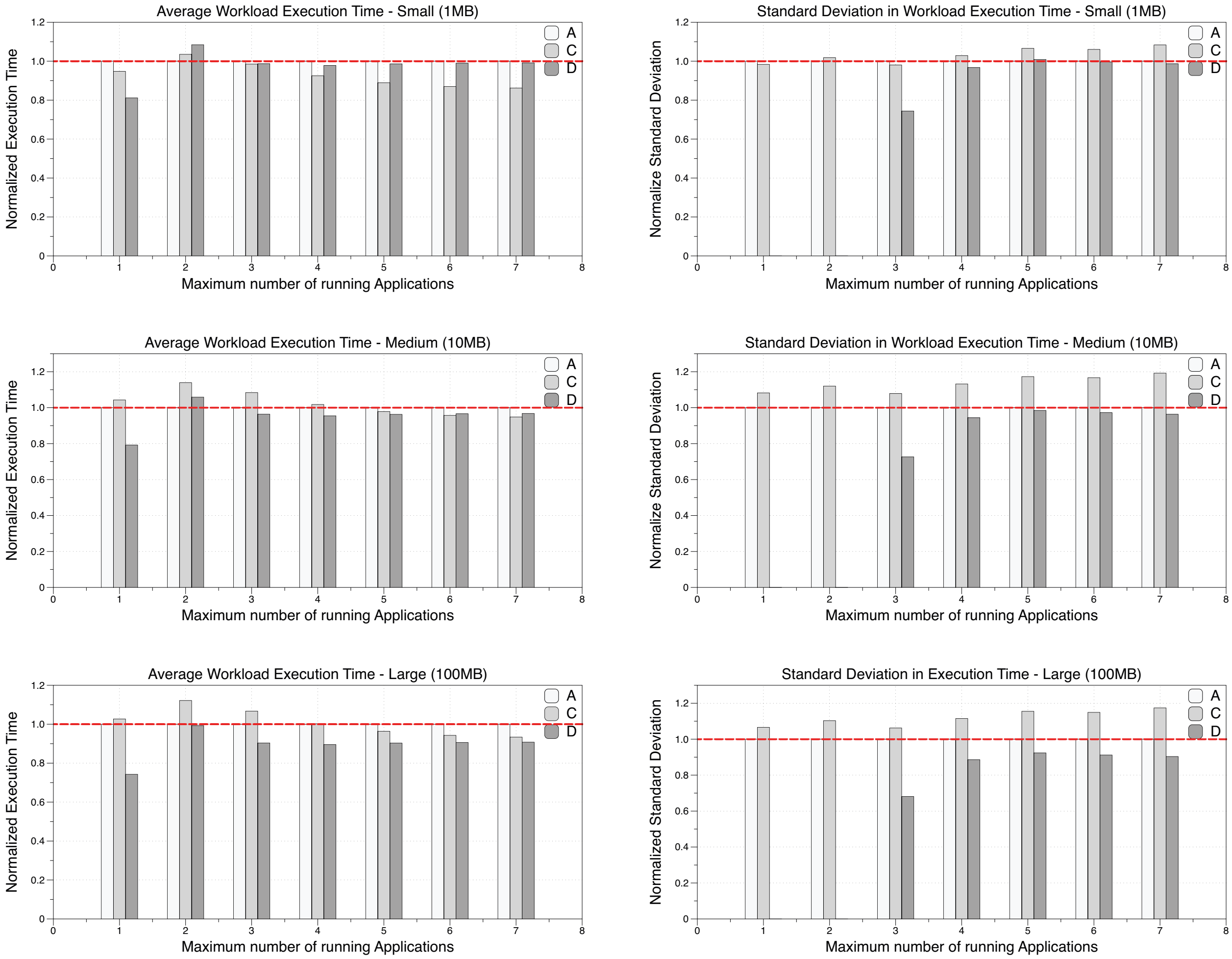

Fig. 18: Normalized execution times, on the left, and standard deviations, on the right, w.r.t. architecture A.

Table I reports the average execution time and the standard deviation when executing a workload mix composed of small, medium and large instances of our test applications when no parallelism among them is extracted by the simulator. As the table clearly shows, when we need to perform a hardware reconfiguration to implement the best physical pipeline, the case of architecture B is clearly not an option since it leads to an overhead up to 15x due to the reconfiguration overhead which takes 1.20s to be performed.

Figure 18 shows execution times, on the left, and standard deviations, on the right, when the simulator is allowed to co-schedule multiple application (from 1 up to 7) on the hardware device extracting as much parallelism as possible. This experiment wants to show how the architectures behave in the case of different system load, which in our simulator is represented by the possibility to schedule multiple applications at the same time; in this case our solution achieves a speed-up up to 20-25%. As we can see from the graphs reporting the execution time, when the simulator is allowed to schedule at most one application, the architecture that adopts our interconnection component is able to obtain the best results (up to 20% of performance improvement) independently of the workload input size. This can be simply explained by the results provided in the previous sections since no parallelism between applications in the workload is extracted. When we increase the extracted parallelism, our solution is not the best choice in terms of execution time for workloads with a small input: in fact, it gets comparable results with respect to the solution A. However, in this case, we do not get great advantages since the data transfer time is negligible and architecture C reaches up to 15% of speed-up. Considering the medium input workloads, we did not experience particular gains or losses and our solution (architecture D) gets comparable performance to the fastest one in the system when altering the system load. Finally on a large input size our architecture is the best one in all the scheduled situations obtaining a constant 10% speed-up when the simulator is allowed to schedule more than 3 applications at the time, a 25% speed-up when it has to schedule a single application and no benefits or losses when 2 applications are scheduled. The case of a large input size is the most relevant since it is the one most widely adopted when performing HPC computing that

TABLE I: Workload average execution time and standard deviation for implemented architectures on different input size.

	Small (1MB)		Medium (10MB)		Large (100MB)	
	Avg[s]	$\sigma[s]$	Avg[s]	$\sigma[s]$	Avg[s]	$\sigma[s]$
A	1.48	0.11	13.12	0.94	134.86	9.63
B	15.75	4.90	26.30	4.99	136.77	9.75
C	1.41	0.10	13.68	1.01	138.51	10.27
D	1.20	0.00	10.39	0.00	100.18	0.00

is the situation we want to tackle. Finally, the graphs about standard deviation show that our solution effectively reduce variance in execution times in case of low load situations. Increasing the system load, our solution has an overhead at least comparable with other solutions, while for large input size applications the standard deviation is reduced at least by 10%. Considerations about variations in execution times in a HPC environment are fundamentals since unexpected latencies in the application execution may causes delays in other tasks of the same workload (possibly running on other machines) and have to synchronize before the completion of the application.

VIII. Conclusions

In this work, we proposed and implemented a configurable interconnection structure for hardware dataflow HPC applications able to dynamically instantiate different pipelines structures on the hardware device changing the way the pipeline stages are connected. This solution becomes effective when we have a hardware device containing basic high-level functions and we want to connect them in different ways that are not always predictable at design time. We demonstrated that the proposed implementation is effective on a image processing dataflow architecture where our interconnection component is used to create different pipelines at runtime avoiding the need of transferring data to and back from the device reducing the computation time. Results showed benefits regarding the overall execution time of a generic workload (up to 25%) and a lower standard deviation (possibly a fixed behavior on low loaded systems) with respect to different static and reconfigurable hardware solutions.

Currently we are working on the implementation of the Runtime Manager that has been simulated in this work to demonstrate that such architecture can benefit from different application scheduling and pipeline stages mapping policies. Finally we will try to integrate partial reconfigurable hardware cores in the architecture instead of static ones to exploit the flexibility of this interconnection to ease the mapping and relocation of such cores onto the architecture.

Acknowledgments

This work was partially funded by the European Commission in the context of the FP7 FASTER project (#287804).

References

[1] “Maxworkstation,” Maxeler Technologies Ltd. [Online]. Available: http://www.maxeler.com/products/desktop/

[2] E. A. Lee and T. Parks, “Dataflow process networks,” in *Proceedings of the IEEE*, 1995, pp. 773–799.

[3] A. Gunzinger, S. Mathis, and W. Guggenbuhl, “The synchronous dataflow machine: a computer architecture for real time image processing,” in *Pattern Recognition, 1990. Proceedings., 10th International Conference on*, vol. ii, jun 1990, pp. 436 –441 vol.2.

[4] B. Bhattacharya and S. Bhattacharyya, “Parameterized dataflow modeling for dsp systems,” *Signal Processing, IEEE Transactions on*, vol. 49, no. 10, pp. 2408 –2421, oct 2001.

[5] J. Pan, L. Xue, Z. Huang, and Q. Lin, “Numerical simulation of the stock option pricing,” in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 5, oct. 2010, pp. V5–195 –V5–197.

[6] J.-F. Wang, S.-W. Chang, and P.-C. Lin, “A novel round function architecture for aes encryption/decryption utilizing look-up table,” in *Security Technology, 2003. Proceedings. IEEE 37th Annual 2003 International Carnahan Conference on*, oct. 2003, pp. 132 – 136.

[7] D. Bafumba-Lokilo, Y. Savaria, and J.-P. David, “Generic crossbar network on chip for fpga mpsocs,” in *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, june 2008, pp. 269 –272.

[8] S. Murali, L. Benini, and G. De Micheli, “An application-specific design methodology for on-chip crossbar generation,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 1283 –1296, july 2007.

[9] Y. Kim, K. Park, and M. Kim, “Amba based multiprocessor system,” in *System-on-Chip, 2003. Proceedings. International Symposium on*, nov. 2003, pp. 41 – 42.

[10] M. Ramirez, M. Daneshtalab, J. Plosila, and P. Liljeberg, “Noc-axi interface for fpga-based mpsoc platforms,” in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, aug. 2012, pp. 479 –480.

[11] W. Dally and B. Towles, “Route packets, not wires: on-chip interconnection networks,” in *Design Automation Conference, 2001. Proceedings*, 2001, pp. 684 – 689.

[12] L. Benini and G. De Micheli, “Networks on chips: a new soc paradigm,” *Computer*, vol. 35, no. 1, pp. 70 –78, jan 2002.

[13] L. shiuan Peh and W. J. Dally, “Flit-reservation flow control,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 3, pp. 194–205, 2000.

[14] K. Srinivasan and K. Chatha, “A technique for low energy mapping and routing in network-on-chip architectures,” in *Low Power Electronics and Design, 2005. ISLPED ’05. Proceedings of the 2005 International Symposium on*, aug. 2005, pp. 387 – 392.

[15] R. Gindin, I. Cidon, and I. Keidar, “Noc-based fpga: Architecture and routing,” in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, may 2007, pp. 253 –264.

[16] M. Puthal, V. Singh, M. Gaur, and V. Laxmi, “C-routing: An adaptive hierarchical noc routing methodology,” in *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, oct. 2011, pp. 392 –397.

[17] Y.-C. Lan, M. Chen, A. Su, Y.-H. Hu, and S.-J. Chen, “Flow maximization for noc routing algorithms,” in *Symposium on VLSI, 2008. ISVLSI ’08. IEEE Computer Society Annual*, april 2008, pp. 335 –340.

[18] D. Gaoming, Z. Duoli, S. Yukun, M. Liang, H. Ning, and G. Minglun, “Performance evaluation of fpga based crossbar noc architecture,” in *Solid-State and Integrated-Circuit Technology, 2008. ICSICT 2008. 9th International Conference on*, oct. 2008, pp. 2058 –2061.

[19] “Maxeler Technologies Ltd. website.” [Online]. Available: http://www.maxeler.com

[20] “Xilinx Vivado Design Suite, available at http://www.xilinx.com.”

[21] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, “Fpmr: Mapreduce framework on FPGA,” in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 93–102.

[22] Y. Park, B. Min, and J. Kim, “A new method of illumination normalization for robust face recognition,” *Progress in Pattern Recognition, Image Analysis and Applications*, pp. 38–47, 2006.