# Exploring Automatically Generated Platforms in High Performance FPGAs

Panagiotis Skrimponis[b], Georgios Zindros[a], Ioannis Parnassos[a], Muhsen Owaida[b], Nikolaos Bellas[a], and Paolo Ienne[b]

[a] *Electrical and Computer Engineering Department, University of Thessaly, Greece*
[b]*Ecole Polytechnique Federale de Lausanne, Switzerland*

***Abstract.*** Incorporating FPGA-based acceleration in high performance systems demands efficient generation of complete system architecture with multiple accelerators, memory hierarchies, bus structures and interfaces. In this work we explore a set of heuristics for complete system generation, with the objective of developing automatable methodology for system level architectural exploration and generation. Our experimental analysis on two test cases demonstrates that applying a set of system optimization heuristics incrementally on a baseline system configuration, we can converge to efficient system designs and reach target performance.

***Keywords.*** FPGA, High Level Synthesis

## 1. INTRODUCTION

Recent advances in FPGA technology and High Level Synthesis (HLS) methodologies have placed reconfigurable systems on the roadmap of heterogeneous High Performance Computing (HPC). FPGA accelerators offer superior performance, power and cost characteristics compared to a homogeneous CPU-based platform, and are more energy efficient than GPU platforms.

However, the biggest obstacle for adoption of FPGA technology in HPC platforms is that FPGA programming still requires intimate knowledge of low-level hardware design and long development cycles. These characteristics make HDLs an unsuitable technology to implement an HPC application on an FPGA.

On the other hand, HLS tools allow designers to use high-level languages and programming models such as C/C++ and OpenCL [4][5]. By elevating the hardware design process at the level of software development, HLS not only allows quick prototyping, but also enables architectural exploration. Most of the HLS tools offer optimization directives to inform the HLS synthesis engine about how to optimize parts of the source code. The HLS synthesizer implements hardware accelerators optimized for performance or area according to these directives.

This approach does not exploit the capability of modern FPGAs to implement architectures that may include multiple hardware accelerators, memory hierarchies, bus structures and interfaces. What is really needed is a methodology to automatically realize and evaluate multiple-accelerator architectures implementing complex software applications. The complexity of such a methodology is high owing to the fact that reconfigurable platforms offer multiple degrees of design freedom and a potential large set of pareto-optimal designs.

In this paper we introduce a systematic architectural evaluation of application mapping onto High Performance FPGAs that operate as accelerators to a Linux box. The focus is to exploit all the interesting system-level architectural scenarios, in order to build a tool flow that can choose automatically the best system-level architecture for each application. Based on the experimental evaluation, we want to gain insight of optimal application-dependent architectures so that we later automate this process. We compare the performance of all hardware solutions with the performance of the software code running on a high performance CPU (the Host Unit).

Several research efforts studied the problem of Host-FPGA interface and on-chip communication channels. Vipin et al. [1], developed a framework for using PCIe communication between Host CPU and FPGA. He also used HLS standard bus interfaces for on chip communications. However, it overlooked the customization of the bus interfaces, as well as custom memory hierarchies. Other recent works considered generating latency insensitive channels [2] and shared memories [3] between multiple FPGA accelerators. In this work we seek further level of customization for the system component.

## 2. DESIGN SPACE EXPLORATION

Listing 1 shows the pseudocode of the *Blur* filter, one of the applications under evaluation. The algorithm first applies a horizontal and then a vertical 3-tap low pass filter to an incoming image, temporarily storing the output of the horizontal filter to the memory. This pseudo code is optimized for a CPU execution, not for a hardware design, which leads to drawbacks when it is processed by HLS tools. This code results into two hardware accelerators, which have to communicate via a large memory implemented either as an external DRAM or as an on-chip BRAM (if there is enough BRAM in the FPGA). In order to incorporate FPGAs as part of a modern heterogeneous system, we exploited the standardization of communication abstractions provided by modern high-level synthesis tools like Vivado HLS to create our adaptive system.

The dark shaded logic of Fig. 1 is generated by the Vivado toolset, based on instructions by the system developer. Two accelerators are instantiated and are connected with a BRAM memory through an AXI4-master interconnect. This baseline

```
    blur_hor:
    for (i = 0; i < Height; i++)
     for (j = 0; j < Width; j++)
     tmp(i,j)=(inp(i,j-
1)+inp(i,j)+inp(i,j+1)/3

    blur_ver:
    for (i = 0; i < Height; i++)
     for (j = 0; j < Width; j++)
     out(i,j)=(tmp(i-
1,j)+tmp(i,j)+tmp(i+1,j)/3
```

**Listing 1. Horizontal and Vertical Blur filter**

architecture is extended by automatically exploring the number and type of the various resources. Such resources include the accelerators in terms of throughput, area, latency and number. It also includes the bus structure and number of separate buses, the number and type of memories, and the interface to the Host unit.

In addition to the customizable part of the architecture, extra resources are required for communication with the Host unit. We use an open-source framework, RIFFA to provide an abstraction for software developers to access the FPGA as a PCIe-based accelerator [6].

The RIFFA hardware implements the PCIe Endpoint protocol so that the user does not need to get involved with the connectivity details of the accelerator. From the accelerator side, RIFFA provides a set of streaming channel interfaces that send and receive data between the CPU main memory and the customizable logic. On the Host unit, the RIFFA 2.0 architecture is a combination of a kernel device driver and a set of language bindings. RIFFA provides a very simple API to the user that allows for accessing individual channels for communicating data to the accelerator logic.

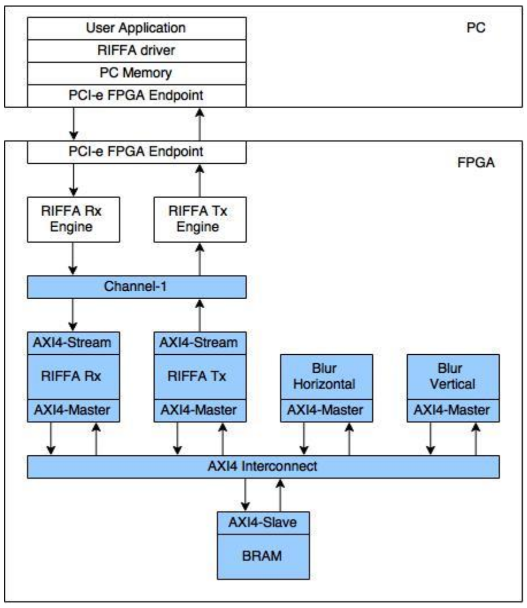


**Figure 1.** Baseline platform architecture used for our experimental evaluation. The dark shaded area shows the customizable logic.

Figure 2 shows two indicative architectural scenarios, expanding the baseline architecture of Fig. 1. We can effectively duplicate the customizable logic using an extra RIFFA channel (Figure 2i). Even better, the streaming, point-to-point nature of the *Blur* application allows us to use the AXI4-Stream protocol to channel data between consumer and producers (Figure 2ii). Some configurations may use external DDR3 memory (which includes an FPGA DDR3 memory controller) to be able to accommodate larger images at the cost of increasing latency and worse performance. To navigate through the large design space smartly, we devised a set of heuristics for making design decisions:

1. Keep data local as close as possible to the accelerator. The goal here is to minimize read/write latency between the accelerator and data memory.
2. Minimize shared resources between independent accelerators. Following this guideline helps eliminating collisions between multiple accelerators while accessing communication channels and memory ports.
3. Overlap data transfers with accelerators execution. The objective here is to minimize accelerators idle time waiting for data to be available.

Our design space exploration approach starts from the baseline architecture of Fig. 1. We then incrementally make design decisions while considering the aforementioned heuristics and evaluating the effects of the taken decisions on overall system performance.

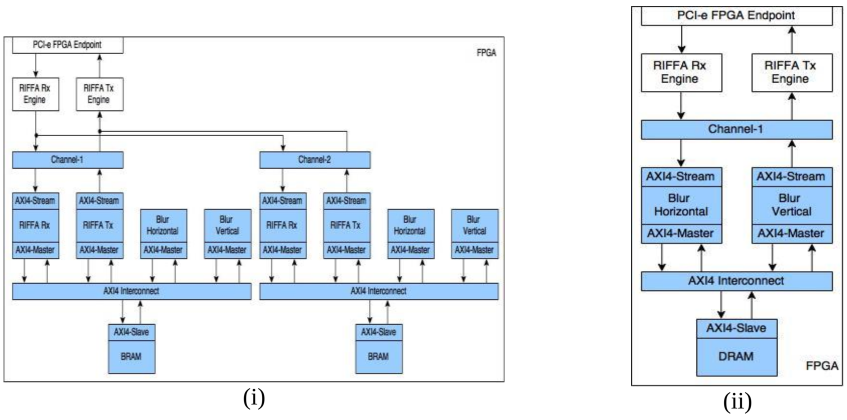


**Figure 2.** Two interesting architectural scenarios. (i) Duplication of the baseline architecture using two RIFFA channels. (ii) Using AXI streaming interface between RIFFA channels, the two accelerators and the DRAM.

## 3. EXPERIMENTAL EVALUATION

### *3.1. Methodology*

In this section, we present our architectural exploration study for two applications shown in **Table 1**. For each application, we have laid out a multitude of architectural scenarios spanning different points at the area versus performance space. Software code is optimized with HLS pragmas (e.g., pipeline) with minimal code changes. We use Vivado HLS 2014.4 for hardware implementations on the VC707 evaluation board (XC7VX485T FPGA device). All results are reported after placement and routing. The same C code is executed in an E5520 Intel Xeon quad-core processor running at 2.27 GHz and the performance results are compared. Besides the *Blur* filter already described in the previous section, we have evaluated a Monte Carlo simulation application.

**Table 1.** Applications used for architectural exploration

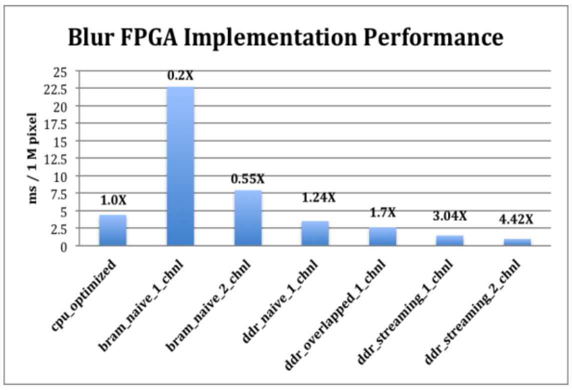
| App. | Description | Input Set |
|---|---|---|
| Monte Carlo | Monte Carlo simulations in a 2D space | 120 Points, 5000 Walks per point |
| Blur | Blur 2D filter | 4096×4096 image |

Monte Carlo (MC) was developed to provide an alternative method of approaching multi-domain, multi-physics problems. It breaks down a single Partial Differential Equation (PDE) problem to produce a set of intermediate problems. The core kernel of this application performs random walks from initial points in a 2D grid to estimate the boundary conditions of the intermediate problems. MC is a heavily compute bound application with double precision arithmetic operations and calls to mathematical libraries for FP arithmetic, trigonometric and logarithmic computations, etc. It has minimal bandwidth requirements.

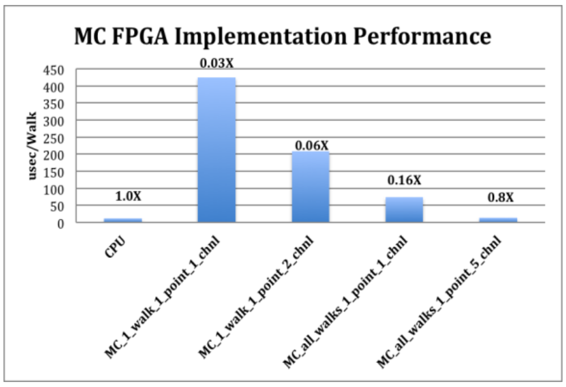
### *3.2. Results*

Figure 3 presents the performance results for the two test cases each with different implementation scenarios. Table 2 shows the area requirements of each platform.

**Blur.** Six implementation scenarios are studied for the Blur application. The first scenario represents the baseline architecture in Fig. 1 (`bram_naive_1_chnl`). The host CPU sends a single row for the horizontal blur kernel for processing and waits for the result from the accelerator before sending the next row until the entire image is processed. The same is done for the vertical blur, but here 3 rows are needed for the vertical blur to start. This scenario is reasonable when there is not enough on-chip or off-chip memory to save the whole image, then we partition the image into smaller partitions that fit on the available memory resources. Another version of this scenario (`bram_naive_2_chnl`) replicates the hardware of a single RIFFA channel on 2 channels to exploit parallelism in the Blur application. While the second scenario



(*i*)



(b)

**Figure 3.** Experimental performance results for the Blur and Monte-Carlo applications. The numbers above the bars are improvement over the optimized CPU implementation.

improves on the performance of the BRAM naive implementation it is still worse than that of the optimized CPU implementation. Sending small chunks of data over the PCIe is not efficient because we pay the overhead of initiating a PCIe read/write transaction many times. As a result, the PCIe transactions occupy two thirds of the total execution time.

Scenario three of the Blur (`ddr_naive_1_chnl`) makes use of the off-chip DDR to store the whole image instead of partitioning it into multiple portions. Using a DDR provides few benefits; The PCIe read/write transactions consume less time compared to the first scenarios because we eliminate the overhead of initiating PCIe transactions. The second benefit of the DDR is that we do not need to send the horizontal blur output back to the host CPU, but keep it in the DDR for the vertical blur to process it, then write back to the host CPU the result of the vertical blur. As such, the third scenario achieves improvement over optimized CPU time.

To improve performance of scenario #3, we allow the horizontal blur to start as soon as the first row of the image is stored in the DDR and not wait for the whole image to be loaded. We also allow writing data back to the host CPU even before the vertical blur accelerator finishes execution. This is demonstrated in the fourth scenario (`ddr_overlapped_1_chnl`). The overlapping of accelerators execution with FPGA-

**Table 2.** Area results for the various configurations in terms of resource utilization for the XC7VX485T FPGA

| **Benchmark** | **LUT** | **FF** | **BRAM** | **DSP48** |
|---|---|---|---|---|
| **BLUR** | | | | |
| bram_naive_1_chnl | 15% | 10% | 5% | 11% |
| bram_naive_2_chnl | 28% | 20% | 10% | 23% |
| ddr_naive_1_chnl | 20% | 15% | 6% | 5% |
| ddr_overlapped_1_chnl | 20% | 15% | 6% | 5% |
| ddr_streaming_1_chnl | 13% | 9% | 4% | 3% |
| ddr_streaming_2_chnl | 21% | 14% | 6% | 6% |
| **Monte-Carlo (MC)** | | | | |
| MC_1_walk_1_point_1_chnl | 8% | 5% | 2% | 5% |
| MC_1_walk_1_point_2_chnl | 13% | 9% | 3% | 9% |
| MC_all_walks_1_point_1_chnl | 9% | 7% | 2% | 5% |
| MC_all_walks_1_point_2_chnl | 16% | 12% | 3% | 9% |

Host data transfers is possible because of the regular access patterns of the blur kernels. While this scenario eliminates most data transfers overhead, moving data between DDR and the accelerators introduces a non-negligible overhead.

To improve further and minimize the DDR-Accelerator communication overhead, we use AXI-stream interfaces for horizontal and vertical blur accelerators (`ddr_streaming_1_chnl, ddr_streaming_2_chnl`). Instead of storing the image in the DDR, the horizontal blur accelerator uses AXI-stream interface to read data from the channel FIFOs and write result to the DDR. The vertical blur will read data from the DDR, process it and send the results directly to the RIFFA channel through an AXI-stream interface. In this scenario we eliminate 60% of the DDR-Accelerator data movements. This is possible because of the streaming nature of the blur kernels. This scenario achieves the best performance compared to the CPU implementation. Moreover, this scenario consumes less area (see Table 2) than the DDR naive and overlapped implementations, which allows allocating more replicas of the accelerators to exploit parallelism and improve performance as the case in *ddr_streaming_2_chnl*.

**Monte-Carlo simulation.** MC is a compute intensive kernel with minimal memory accesses. Hence the different implementation scenarios are made of different accelerator configurations and by instantiating multiple instances of the accelerator. In the baseline scenario, the accelerator is configured to perform a single walk of a single point per invocation, and a single accelerator is allocated (*MC_1_walk_1_point_1_chnl*). This scenario performs much worse than the CPU implementation. Double precision operations have larger latency on FPGAs than a CPU. Also, Vivado HLS libraries of trigonometric operators (sin, cos) are not pipelined and less efficient than their CPU counterparts. The strength of the FPGA is to perform more computations in parallel. Unfortunately, the MC kernel computations of a single walk are sequential and cannot be parallelized. As such, the FPGA baseline implementation performs badly. To improve performance we need to exploit coarser-grain parallelism across multiple points and walks. The second scenario allocates two accelerator instances to parallelize the computations of walks for a single point (*MC_1_walk_1_point_2_chnl*). It reduces the execution time to half, but still worse than the CPU. We need to allocate around 40 accelerator replicas to reach the CPU performance.

Another aspect to improve on is to minimize the accelerator invocation overhead by coarsening the computations granularity per a single accelerator instance. This allows for pipelining computations of multiple walks, which will have a strong impact on performance. The third scenario minimizes accelerator invocation overhead by configuring the accelerator to perform all the walks of a single point per invocation (*MC_all_walks_1_point_1_chnl*). The fourth scenario allocates five accelerators to parallelize computations (*MC_all_walks_1_point_5_chnl*) across multiple points. The last scenario saturates the FPGA resources and almost achieves near CPU performance.

## 4. CONCLUSION

Reaching target performance does not have a trivial solution. Customizing the accelerator configuration while using a fixed system architecture is not enough to compete with state-of-the-art CPUs. It is essential to customize the system architecture to reach this goal, especially in applications where data movement overhead dominates overall performance. In this effort we studied few directions in system-level architectural exploration to orchestrate an approach for customizing system-level components, such as number and type of bus interfaces, memory hierarchies, and number of accelerators. We considered different data transfer protocols as a way to minimize data movement overhead. We intend to study other types of applications to extract more efficient ways of system customization as a preliminary for building an automatable methodology for custom system generation.

## 5. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: THALES grant number MIS 379416.

## REFERENCES

[1] K. Vipin, S. Shreejith, D. Gunasekera, S. A. Fahmy, N. Kapre. System-Level FPGA Device Driver with High-Level Synthesis Support. International Conference on Field Programmable Technology (FPT), Kyoto, Japan, December 9-11, 2013.

[2] K. Fleming, H. J. Yang, M. Adler, J. Emer. The LEAP FPGA Operating System. 24th International Conference on Field Programmable Logic and Applications (FPL). Munich, Germany, September 2-4, 2014.

[3] H. J. Yang, K. Fleming, M. Adler, J. Emer. LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories. 22nd International Symposium on Field Programmable Custom Computing Machines (FCCM). Boston, USA, May 11-13, 2014.

[4] *Vivado Design Suite User Guide: High Level Synthesis*. Online at www.xilinx.com.

[5] *Altera OpenCL SDK Programming Guide.* Online at www.altera.com.

[6] M. Jacobsen, R. Kastner. *RIFFA 2.0: A reusable integration framework for FPGA accelerators.* 23rd International Conference on Field programmable Logic and Applications (FPL). Porto, Portugal, September 2-4, 2013.