Virtual Reconfigurable Functional Units on Shared-Memory Processor-FPGA Systems

Fernando Passe and Vanessa C. R. Vasconcelos
Departamento de Informática,
Universidade Federal de Viçosa,
Campus Viçosa,
Viçosa-MG 36570-900, Brazil
Email: {fernando.passe,vanessa.vasconcelos}@ufv.br

Ricardo Ferreira
Departamento de Informática,
Universidade Federal de Viçosa,
Campus Viçosa,
Viçosa-MG 36570-900, Brazil
Email: ricardo@ufv.br

Lucas B. Silva and J. A. Nacif
Departamento de Informática,
Universidade Federal de Viçosa,
Campus Florestal,
Florestal-MG 35690-000, Brazil
Email: {lucas.braganca,jnacif}@ufv.br

***Abstract*—Recently, FPGA-based platforms with direct coherent access to processor memory have been proposed. However, the FPGAs have traditionally been viewed as difficult to program for the software community and high performance community. This work proposes a framework to explore the FPGA-processor platforms as an opportunity for accelerating applications with streams and irregular parallelism. A virtual layer is mapped onto the FPGA, and a high dataflow graph could be translate and mapped dynamically to the FPGA accelerator. The proposed framework add a dynamic reconfiguration layer to the Intel® shared-Memory Processor-FPGA platform.**

***Keywords*—FPGA, Shared-Memory, Processor-FPGA.**

I. Introduction

One challenge traditionally associated with FPGAs is to model a computation as an accelerated function, and then translate it and map it into the reconfigurable architecture. Moreover, hardware description languages (HDL) are not friendly for the larger software community. In addition, the processor and the FPGA system should efficiently exchange data. Recently, accelerated computing systems that support coherent shared memory between processors and FPGAs, including direct read/write to the processor's cache [1] become available. Intel®, Altera®, Xilinx®, IBM® and Microsoft® have respectively developed products that integrate cache-coherent shared-memory processors and FPGAs at the system level [1], [2], [3], [4], [5], [6].

We propose to extend our previous work [7], [8], [9] on virtual CGRA and Dataflow graphs to the new context introduced by the Intel® QuickAssist Technology.

As previously said, this technology integrates cache-coherent shared-memory processors and FPGAs, i.e. the FPGA can read from processor's cache, it can compute the read data and it can write the results back on the cache, where they will be available to processor future access.

There are many advantages to FPGA computation, among these are the abundant hardware parallelism and the FPGA high adaptability. Both features are extremely useful and they can be largely used. The parallelism can speed up processes and the adaptability allows future modifications.

On the other hand, there are difficulties to handle FPGA computation. Hardware coding is far from trivial, handling all the desired parallelism is not usual for programmers. Generating and loading the bitstream onto the hardware are also problematic, because in so far as the platform is growing, the bitstream will take longer to be ready, thus more complex features will need more time to be checked. Consequently, identifying errors becomes even harder.

We propose a parameterized architecture to exploit run-time reconfiguration in Intel®/Altera® platform by using a dataflow graph model on a flexible CGRA based on global interconnection model placed as an Accelerated Function Unit (AFU). At run-time the Intel® Software API will be used transparently by the programmer, and the dataflow graph will be instantiated onto a virtual CGRA, as shown in Fig. 1. There is no need neither to synthesize the AFU nor upload its bitstream on the FPGA. This will be done once, then a dataflow graph which expresses spatial and temporal parallelism will be configured and executed as a dynamic virtual FPGA accelerator.

Section II introduces samples of dataflow graphs. Section III presents the virtual CGRA architectures. Section IV describes the proposed methodologies to map the dataflow graph onto the virtual AFU. Finally, Section V discusses the on-going work.

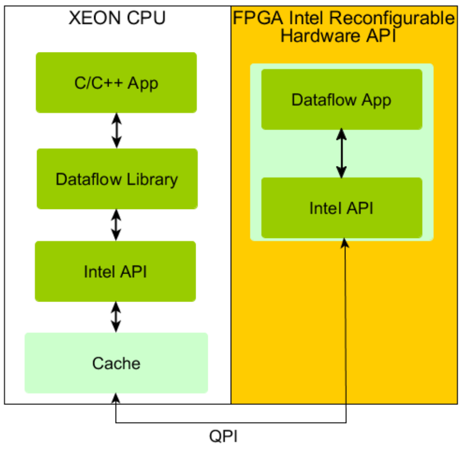


Fig. 1. Dataflow framework

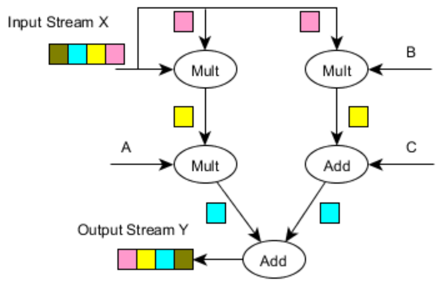


Fig. 2. Dataflow Graph for a $ax^2 + bx + c$.

II. Dataflow Graph

First, assuming a simple dataflow graph as shown in Fig. 2. The nodes implement input/output operations and/or logic/arithmetic functions. Let us suppose the stream computation $ax^2 + bx + c$. Fig. 2 shows the dataflow graph where there are spatial and temporal (pipeline) parallelism. X is an input stream data. Assuming, one clock cycle per operation. The computation latency will be 3 clock cycles. However, the throughput will be one new result per cycle due the pipeline execution. In addition, five operations are executed every clock cycle and the operator occupation rate is 100%, the hardware is always doing useful computation, since three stream instances are overlapped. Our goal is to provide a high level model to describe parallel stream based tasks. The nodes could be arithmetic or logic operators. The input and output nodes will receive/send stream data. The stream could be sent to the AFU on configuration time or it could be done as a cache read/write through the QPI Intel® interface – Intel®'s proprietary interconnect protocol between Intel cores or other IP – at run-time as depicted in Fig. 3. Following sections will detail the mapping process.

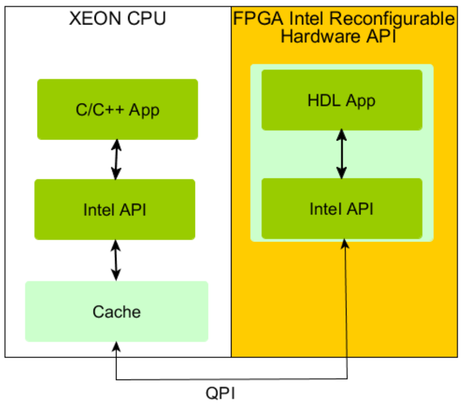


Fig. 3. Dataflow mapped as an AFU.

Dataflow graphs could also implement control flow operators. We propose to implement a library of control flow operators as depicted in Fig. 4 based on previous dataflow work. Fig. 4a shows functional operators like an adder or a multiplier. There are also the copy, split or fanout operators which split one input flow onto two output flows. Fig. 4b depicts a merge operator. This operator could be used to implement control flow structure as displayed in Fig. 4c, where an IF flow is implemented based on Tgate/Fgate operators. The input condition C will allow the input A or B to propagate toward the output if C is True or False. There are also other control flow operators as Sep and Pas depicted in Fig. 4d.

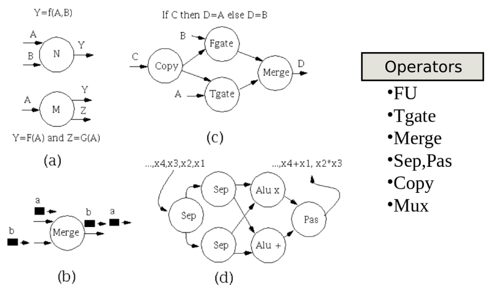


Fig. 4. Data and control flow operators.

The Sep (Serial to Parallel) operator sends the input data by switching (or alternating) the output. The Pas (Parallel to Serial) node merges data by switching the input data. These operators could be useful to simplify the control flow to split stream data as shown in Fig. 4d, where every chunk of 4 data will be applied to the following computation. Our goal is to provide a library of data and control flow operators to allow the users to design generic and flexible AFU. Fig. 5a-d shows four possibilities to implement an IF flow, and Fig. 5e depicts an example of a while loop based on previous dataflow work [10].

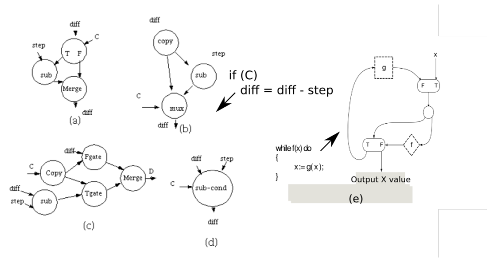
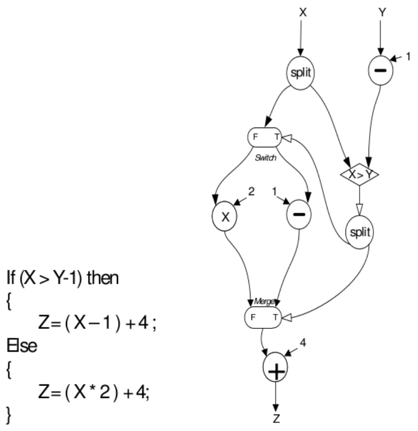


Fig. 5. Dataflow operators

Fig. 6 shows how a C code with an if/else block could be translated by using control and data operators.

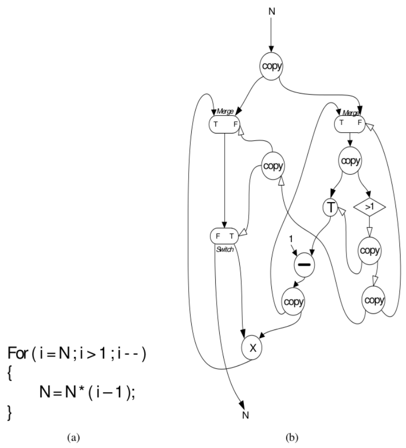
A For block could also be implemented as depicted in Fig. 7, where the factorial function is computed as proposed in [10].

The first step is to model a target function as a dataflow graph by using a component library. The next step is to map the dataflow onto the FPGA architecture. The following sections will introduce a virtual framework architecture and the mapping approaches.



```
If (X > Y-1) then
{
    Z=(X−1)+4;
Else
{
    Z=(X*2)+4;
}
```

Fig. 6. If/Else block and its dataflow



```
For(i=N;i>1;i--)
{
    N=N*(i−1);
}
```

Fig. 7. For block and its dataflow

III. Target Architecture

We propose a Virtual Reconfigurable Accelerated Function Unit (VR AFU) as a virtual layer implemented on the top of FPGA AFU. A dataflow graph will be mapped at run-time onto the target VR AFU. The VR AFU is synthesized once and the FPGA bitstream is uploaded on the FPGA as an AFU. This section describes the VR AFU architecture and the next section will present the mapping steps.

The traditional flow is depicted in Fig. 8a where the dataflow will be designed and then implemented at synthesis time. The AAL/AFU interface will be used during run-time to send/receive data to be computed by the accelerator. We propose the flow shown in Fig. 8b, where the virtual AFU (VR AFU) bitstream is uploaded once at synthesis time. One or more graphs could be sent and computed at run-time onto our proposal VR AFU.

Most CGRA architecture are based on mesh topology [2]. Although the mesh is scalable and regular, the mapping depends on placement and routing algorithms. Our previous work showed that the mapping could be simplified by using

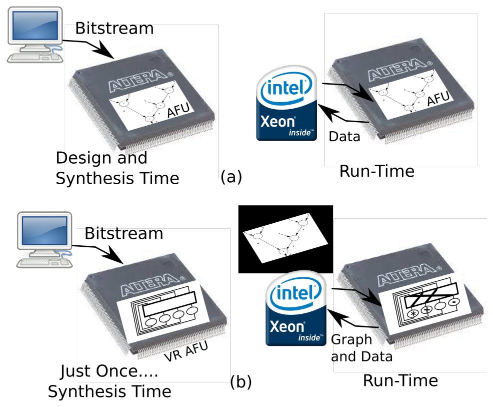


Fig. 8. (a) Traditional flow on FPGA; (b) Proposed flow on FPGA

a global network [7]. The VR AFU architecture is also based on a global interconnection. A VR AFU consists of a set of Functional Units (FUs), which could be data operator or control flow operator, and an interconnection network. We propose a flexible VR AFU by providing libraries of FU and interconnection networks. Users could also add a specific FU to the library.

The basic architecture is displayed in Fig. 9. The configuration memory has the opcode to program the FU operations and the interconnections. The FUs also have local registers to keep data inside the AFU. For small sizes (up to 16 units), the network could be implemented by using a crossbar. For larger sizes (up to 256 units), we propose to use multistage networks [2]. A third option is a hierarchical interconnection with cluster of VR AFU. We propose also to allow pipeline inside the interconnection structure to provide TLP (Thread or/and Task Level parallelism).

IV. Mapping

The mapping step starts from a dataflow specification to generate the scheduling, the placement and routing on the target VR AFU. For ease of explanation, suppose a homogeneous FU set and a stream dataflow graph. Fig. 10 shows a simple example of dataflow graph and its final mapping. In addition to the graph, the proposal solution should also provide the input and output streams through the QPI layer.

First the data is applied onto the input stream, and it sends it to FU1 (mult) and FU2 (adder). Then, the second stream data arrives on FU1 and FU2, and stream 1 is sent to FU3 to compute the subtraction. Finally, the divisor is applied to the stream data 1 on the third cycle. The streams 2 and 3 are also in the pipeline. Heterogeneous FU sets as well as large graphs could be managed. Suppose a small target AFU with 4 FUs. Suppose an unbalanced dataflow graph of 6 operators as shown in Fig. 11a. We propose to apply our previous modulo scheduling approaches [1], [2] to place and route by using two configurations (red and blue), see Fig. 11b. Every two clock cycle a new stream data is inserted. The latency is 4 clock cycles, however the throughput is 2 clock cycles.

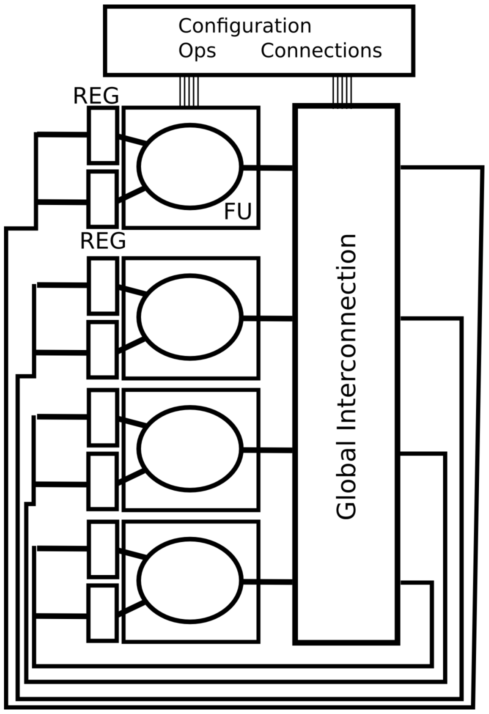


Fig. 9. Basic architecture

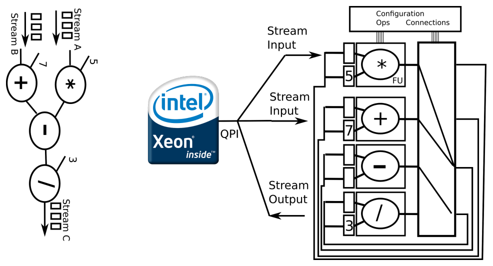


Fig. 10. Dataflow graph and its final mapping

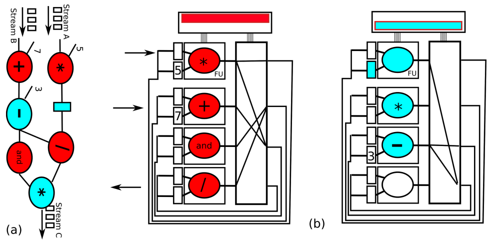


Fig. 11. (a) Unbalanced dataflow graph; (b) Routing configurations

A new stream data will arrive every two clock cycles. There are two configurations. The first configuration maps 4 operations. The results from the adder and multiplier of the configuration 1 will be sent to the bypass and the subtractor on the configuration 2. Then, these operators will send the results to the AND and the divisor on configuration 1. At the same time, configuration 1 will receive a new stream data from the inputs, therefore there is an iteration overlap. Finally, the results from the AND and the divisor are sent to the final multiplier on configuration 2, the pipeline is filled and every two clock cycles, a new stream is processed. Therefore the architecture support spatial and temporal parallelism and it could dynamically map a generic dataflow graph.

V. Final Considerations

This paper presents a framework to map virtual dataflow graphs at run-time into shared-Memory Processor-FPGA Intel® platform. An ongoing work is implementing FPGA pre-configuration and dataflow and data run-time transfer to validate our proposal. Our goal is also to produce didactic examples to explore the Intel®/Altera® platform and tools.

Acknowledgments

We would like to thank CAPES, CNPq, and FAPEMIG for the financial support.

References

[1] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-fpga systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 264–273. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847269

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, May 2015. [Online]. Available: http://dx.doi.org/10.1109/MM.2015.42

[3] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, "A reconfigurable computing system based on a cache-coherent fabric," *Reconfigurable Computing and FPGAs, International Conference on*, vol. 0, pp. 80–85, 2011. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ReConFig.2011.4

[4] *Convey Personality Development Kit Reference Manual*, 5th ed., ©Convey Computer™ Corporation, 1302 East Collins, Richardson, TX 75081, USA, April 2012. [Online]. Available: http://www.conveysupport.com/alldocs/ConveyPDKReferenceManual.pdf

[5] B. Wile, "Coherent accelerator processor interface (capi) for power8 systems," IBM Systems and Technology Group, September 2014. [Online]. Available: http://www.nallatech.com/wp-content/uploads/CAPI_POWER8.pdf

[6] *Zynq-7000 All Programmable SoC Overview*, 1st ed., Xilinx, Inc., January 2016. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[7] R. Ferreira, W. Denver, M. Pereira, S. Wong, C. A. Lisbôa, and L. Carro, "A dynamic modulo scheduling with binary translation: Loop optimization with software compatibility," *Journal of Signal Processing Systems*, pp. 1–22, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11265-015-0974-8

[8] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro, "An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '11. New York, NY, USA: ACM, 2011, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/2038698.2038728

[9] R. Ferreira, J. M. P. Cardoso, A. Toledo, and H. C. Neto, *Data-Driven Regular Reconfigurable Arrays: Design Space Exploration and Mapping*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 41–50. [Online]. Available: http://dx.doi.org/10.1007/11512622_6

[10] J. M. P. Cardoso, *Self-loop Pipelining and Reconfigurable Dataflow Arrays*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 234–243. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27776-7_25