Improving resource usage in large FPGA accelerators

Antonio Filgueras*†, Carlos Álvarez*†, Daniel Jiménez-González*†
*Barcelona Supercomputing Center, Barcelona, Spain
†Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: {antonio.filgueras, daniel.jimenez, carlos.alvarez}@bsc.es

Keywords—FPGA, High-performance computing, Heterogeneous computing.

I. Extended Abstract

A. Introduction

In modern FPGA devices, place and route has become a difficult task for the underlying FPGA implementation tools. This is caused by an increase of device size and complexity. As devices grow in size and number of resources, their topology also grows in complexity. Larger devices are divided in different regions. While this allows to pack a larger number of resources in a single device, it creates a new set of challenges in order to obtain good quality of results while using as many resources as possible. Devices such as Xilinx's Alveo accelerators are comprised of multiple regions called Super Logic Regions (SLR). Crossing from one region to another adds some delay to signal propagation. This can hurt overall timing if implementation tool decides to scatter a single accelerator among different SLRs. Thus, the design may not reach operating frequencies expected by the user. In a similar fashion as the SLRs, they usually have multiple independent memory banks that interface with DDR modules. This requires memory allocations and interconnection to be manually managed by the user, causing extra burden to users. Otherwise, the design will not be able to take profit of the aggregated available bandwidth.

We propose methods to improve resource and bandwidth usage that allow a user to direct how a design is built and implemented while maintaining device abstraction and minimal development overhead.

B. Design

1) Memory access: In order to address memory access contention, data allocations are scattered across device's different memory banks. This is done by adding interleaver modules between accelerators or PCIe and the memory interconnection. These modules are shown in the right diagram of figure 1 as the highlighted blocks. These modules scatter memory accesses across all available memory modules. This allows memory accesses to a single piece of data to be performed in parallel by different accelerators as different parts of the same data structure can be stored in different banks. This increases the overall available bandwidth.

However, early designs consisted of two level interconnection, as shown in the left diagram of figure 1. While this allowed reducing resource usage due to each module being much simpler, it also prevents accesses from being performed in parallel due as a single bus is connecting both interconnection modules. In order to solve this issue, interconnection has been reworked so it can be implemented in single stage as shown on right hand side of figure 1. Then high performance configuration options can be enabled to allow parallel accesses to different memory modules.

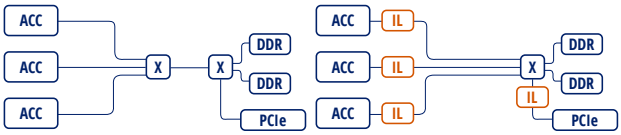
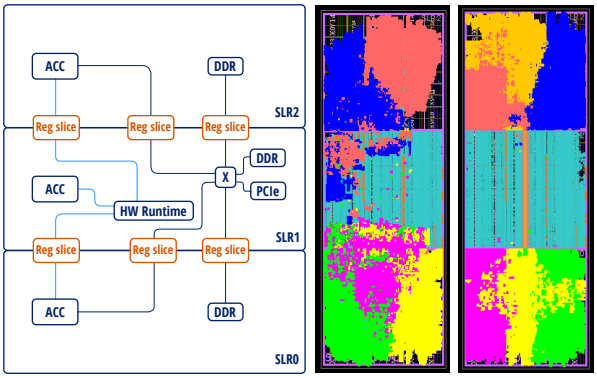


Fig. 1: Interconnection modes: 2 stage vs. single stage with interleave modules.

2) Placement: In order to prevent accelerators to be placed among different SLRs and properly implement region crossing, users are allowed to assign accelerator instances to SLR regions. Implementation tool is then instructed not to allow any resource belonging to an accelerator to be placed outside the region assigned by the user. Along with this, register slices are then properly inserted between accelerator interfaces and interconnection infrastructure when needed. This improves timing as region crossings are done in a controlled fashion.

Figure 2a shows a diagram of a design containing multiple accelerators. Each of them is restricted to its region, and SLR crossings are done by special register slices.



(a) (b) (c)

Fig. 2: Placed accelerator diagram and implemented designs.

Figures 2b and 2c show a physical view of the device without (2b) and with (2c) placement options enabled. Highlighted cells correspond to resources belonging to different accelerator instances. Note that the highlighted cells belonging

to accelerators are not scattered among different SLRs in figure 2c. Also note that device resource usage is higher and we can fit one extra accelerator therefore increasing application performance.

C. Evaluation

1) Experimental setup: Benchmarks used for evaluation have been implemented using OmpSs@FPGA and Vivado HLS pragmas. Experiments are run a Xilinx Alveo U200 (XCU200-FSGD2104). This card is attached to a host via PCIe. The host itself consists of dual Intel Xeon CPU X5680 @ 3.33GHz.

2) Benchmarks: To perform the evaluation, we used two different applications that cover a wide range of characteristics of HPC benchmarks: a matrix multiplication (Matmul) and a Cholesky decomposition. All applications have been implemented by dividing computation in smaller tiles so they can be efficiently implemented in an FPGA. All applications use single precision floating point data.

The matrix multiplication benchmark is a well-known embarrassingly parallel application with a regular dependence pattern. The application operates with three square matrices of size $N \times N$, A, B and C, and computes $C = C + A \times B$.

The Cholesky benchmark performs a Cholesky decomposition of a Hermian, positive definite matrix into a lower triangular matrix, which multiplied by its transpose results in the original matrix. I.e., the application generates an output matrix L from an input C, assuring that $C = L \times L^T$ providing that C fulfils the restrictions. The code uses four kernels: *gemm*, *trsm*, *syrk* and *potrf*. In addition, the particularity of this benchmark is that the potrf kernel is hard to accelerate in FPGA due to its memory access pattern. Previous evaluations [1] demonstrate that it is faster to execute it in the host rather than in the FPGA. We use the OpenBLAS implementation of the kernel, which is in fact a Cholesky decomposition of a single block.

3) Results: Figure 3 shows how proposed features affect these applications. It shows performance (in GFLOPS) of the applications in three different cases. First, without using any of the proposed features (Baseline), then using the proposed memory access improvements (Memory) and finally using both improvements in memory access as well as placement (Mem & placement).

In the Cholesky application, bandwidth increase due to interconnection and interleave features have not a dramatic impact. This is due to the application complexity as it involves different kernels with a critical path more demanding than the Matmul application Also one of them (*potrf*) is run in the main CPU. We are also able to fit two more *gemm* accelerators in the improved version compared to baseline or memory improved version due to the improvements regarding accelerator placement. This further increases performance, but as is the case with memory related optimizations, complex application structure prevents performance improvements from being dramatic even though performance increase is still significant (18%).

In Matrix multiply we can see a big performance increase (50%). This is caused, on one hand, due to increased available

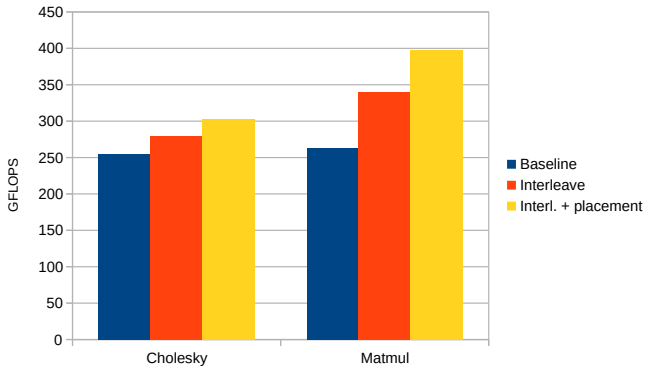


Fig. 3: Application performance.

bandwidth that memory access interleaving and single level interconnection provide. This result is significant not only by itself but because it paves way to even more improvements in future prototypes that use new memory subsystems like HBM. On the other hand, we were able to place two more accelerator instances in the placement improved version, increasing usage of available computational resources, which allows further performance increase over the memory improved version.

D. Conclusion

This paper presents an extension of the OmpSs@FPGA ecosystem in order to more efficiently use available resources. Proposed extensions allow better use of available memory bandwidth through automated memory access interleaving. It also allows improved computational resource usage through accelerator placement improvements. Moreover this is done without causing extra burden to the user and abstracting low level architectural features.

II. Acknowledgment

This work is supported by the TEXTAROSSA project G.A. n.956831, as part of the EuroHPC initiative from Spanish Government (PID2019-107255GB-C21/AEI /10.13039/501100011033), and from Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328).

References

[1] J. M. de Haro, J. Bosch, A. Filgueras, M. Vidal, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Ompss@fpga framework for high performance fpga computing," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2029–2042, 2021.

Antonio Filgueras earned a degree in Computer Science from Universitat Politècnica de Catalunya (UPC) in 2012. Currently, he's a PhD student in the departemt of Computer Architecture at UPC and working in the OmpSs@FPGA team within the Programming Models group at BSC, focused primarily on programming models for reconfigurable systems in High Performance Computing. He has been a researcher in the AXIOM, Legato, EuroEXA and MEEP European projects among others.