2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines

Integrated CUDA-to-FPGA Synthesis with Network-on-Chip

Swathi T. Gurumani*, Jacob Tolar†, Yao Chen†¶, Yun Liang‡, Kyle Rupnow§ and Deming Chen*†
*Advanced Digital Sciences Center, Singapore Email: swathi.g@adsc.com.sg
†University of Illinois at Urbana-Champaign, USA Email: {jstolar2,yaochen,dchen}@illinois.edu
‡School of EECS, Peking University, China Email: ericlyun@pku.edu.cn
§Nanyang Technological University, Singapore Email: k.rupnow@ntu.edu.sg
¶College of Electronic Information and Optical Engineering, Nankai University, China

***Abstract*—Data parallel languages such as CUDA and OpenCL efficiently describe many parallel threads of computation, and HLS tools can effectively translate these descriptions into independent optimized cores. As the number of instantiated cores grows, average external memory access latency can be a significant factor in system performance. However, although each core produces outputs independently, the cores often heavily share input data. Exploiting on-chip data sharing both reduces external bandwidth demand and improves the average memory access latency, allowing the system to improve performance at the same number of cores. In this paper, we develop a network-on-chip coupled with computation cores synthesized from CUDA for FPGAs that enables on-chip data sharing. We demonstrate reduced external bandwidth demand by up to 60% (average 56%) and total application latency in cycles by up to 43% (average 27%).**

I. Introduction

High level synthesis (HLS) has become a frequently applied design technique for fast design time and reduced design space exploration effort. HLS is heavily studied in both academia and industry; a variety of language inputs have been proposed as the input source for HLS tools, including C/C++ [15], SystemC [7], CUDA [13], OpenCL [5]. Serial languages require significant user input or complex automatic parallelization, while parallel languages enable us to perform design exploration at multiple granularity levels by grouping and reorganizing independent computations.

Each HLS generated core consumes area and external memory bandwidth. State-of-the-art HLS tools based on parallel languages tend to instantiate as many cores as possible [9, 14]. However, this throughput oriented synthesis has several problems. First, a high number of cores increase the number of simultaneous external memory requests and exacerbate the average memory access latency. Also, neglecting data reuse and sharing among the cores may lead to reduction in overall system performance.

Fundamentally, there are three classes of techniques to improve off-chip bandwidth and average access latency: caching, memory queue management, and contention management. In this work, we develop a network-on-chip (NoC) model in order to allow HLS-generated cores to share data (improve caching), use outstanding requests buffers to group memory requests (memory queue management), and generate a network between the cores to efficiently scale to large numbers of cores (contention management). Based on cores generated using FCUDA (CUDA-to-FPGA) flow [13, 14], we customize the cores and implement a network and sharing mechanism that interfaces with the cores transparently. We demonstrate that this NoC alleviates the external bandwidth problem, achieving 56% reduction in memory requests and 27% reduction in total execution latency on average.

This paper adds to the state-of-the-art of HLS through:

- A NoC model that transparently interfaces with FCUDA cores, allowing scalable instantiation of cores.
- A set of features enabling efficient NoC implementation. We design a directory-based sharing mechanism that allows cores to share on-chip data within the limited coherency of the CUDA programming model, and an outstanding request mechanism that groups memory requests to minimize duplication of requests.
- An automated flow to generate configurable NoCs integrated with FCUDA cores.

This paper is organized as follows: Section II discusses related work. We present the NoC design integrated with FCUDA flow in Section III. We present the experimental results in Section IV and conclusion in Section V.

II. Related Work

There is significant prior work in the design of network-on-chip for both ASIC- and FPGA-based systems [2, 3, 6, 8, 10, 11, 12]. In these prior works, the network is designed independently as a generic network that can be used with a variety of computation cores [6, 8, 10, 11], or as a network specifically optimized for a particular application or domain [2, 3, 12]. In this work, we have the particular challenge of developing a NoC description that integrates transparently with HLS-generated computation cores. Furthermore, it is important to automate the generation, integration and configuration of the network, as it allows efficient design space exploration using HLS tools.

III. NoC for FCUDA Flow

We leverage our existing FCUDA flow [13, 14] to generate hardware cores from CUDA kernels. The flow generates

978-1-4799-5111-6/14 $31.00 © 2014 IEEE
DOI 10.1109/FCCM.2014.14

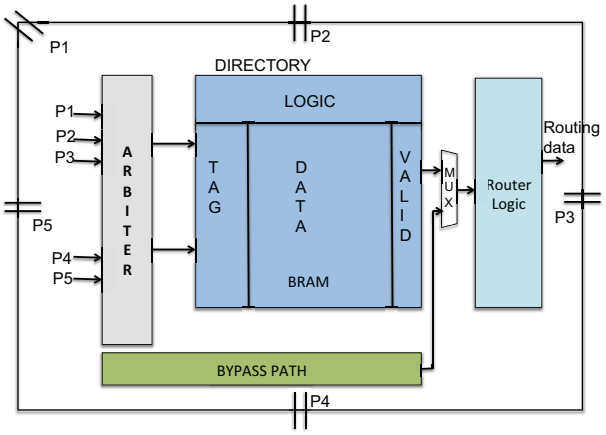

Figure 1. Integration of Directory in Router

annotated C-code from CUDA kernels and then we use Vivado HLS to generate RTL. The cores are generated with a standard hand-shaking interface for core execution, and state monitoring. This interface is simple, but it is important that we retain the exact protocol when inserting multiple cores into a NoC. In this section, We present our NoC design, its integration with FCUDA cores, and the techniques we developed to improve data sharing and communication between cores.

A. Baseline Implementation

Our baseline NoC implementation is based on an open-source packet switching NoC architecture [6]. We have extended this open-source NoC to remove the limitation on network size, improve configurability of design features, and automate the process of NoC generation and computation of static routing tables. In this work, we work with a 2D mesh of compute cores. We now present our specific NoC enhancements to improve on-chip data sharing.

B. Directory-based NoC

First, we enhance the baseline NoC with a directory that keeps track of on-chip data to enable sharing and thus improve average memory access time. Conceptually, this directory system is similar to directory-based cache coherence [1, 4], but has several key distinctions. Importantly, although this directory does keep track of location of some on-chip data, it does not precisely track all on-chip data or maintain coherency between the computation cores. In the CUDA programming model, cores may share input data, but programmers cannot depend on coherency – data written by one core cannot be assumed to be correctly read by other cores. Since the FCUDA programming model is the same as the underlying CUDA model, we do not need to keep track of data updated by the cores and this simplifies our directory protocol.

1) Directory Protocol: In our design, we statically assign addresses to a home node (router) where the directory information for that address is stored. The directory storage is not sufficient to store entries for every address; therefore accesses that do not find an associated entry assume that the data is not available on-chip and produce off-chip accesses. Thus, as we scale to larger directory sizes, it becomes more likely that data is available on-chip, but the directory also consumes resources.

Input addresses are split into tag, index and offset fields similar to typical caching. Also, each directory entry contains a tag field, location field and valid bit. As in caching, the tag is used to distinguish between addresses that can be stored to the same physical location. The location field contains the network address of the core that contains the requested piece of data in its memory, and the valid bit specifies whether the mapping is currently valid.

On a memory read, the request is first routed to the home node (router) for that address. That router checks its local directory; if there is a tag match and valid location mapping, the request is forwarded to the core that currently holds the data. If there is a tag mismatch or invalid entry, the request is forwarded to the memory controller. On return from the memory controller, the directory entry at the home router is updated with the tag and location of the newly received data. Because the core that receives the data may be different from the router node that tracks the directory entry, the memory controller generates two network packets to separately transmit the data update and the directory update.

On a memory write, the directory system is unused; as per the CUDA programming model, it is not permitted for other thread blocks (other cores) to view data updates. Thus, memory writes do not update the directory, but may produce invalidations if a memory location changes from a valid to invalid mapping. This significantly simplifies the protocol so that only the memory controller can update directory entries.

2) Merging Outstanding Requests: The directory system is designed to enable sharing for data that is already on-chip. However, if there are multiple simultaneous requests for the same data that is not on-chip, there would be multiple requests. These duplicate requests will increase average memory access latency and use external bandwidth inefficiently. Thus, we augment the directory system to track outstanding memory requests. Memory requests that are currently outstanding (currently being processed by the memory controller) will be tracked by the corresponding home node.

When a memory request arrives at an address' home node, we first determine if another request to the same address is already being processed in the memory system. If there is an outstanding request, the second request will wait until the data returns rather than producing a duplicate memory request. In the case that the data is expected to arrive at a core other than the home node, the second memory request is forwarded to the other core to await the data. Due to this additional step, it is possible that the data is evicted

before being found and forwarded. Thus, we add a timeout feature that forwards the request to the memory controller if the request is not serviced within the specified number of cycles.

3) Directory Design: The directory is integrated into the NoC router at its input ports. Figure 1 shows the integration of directory into the NoC router. The latency of a NoC is critical to overall performance and thus we make design decisions to reduce latency. Only the address requests and directory update packets access the directory, and only at the home node of the memory address. Thus most packets will not require access to the directory of a router it is passing through. Thus, to minimize the average case latency, we add paths to bypass the directory.

The directory is implemented in a single BRAM that is shared among all input ports using a simple round-robin policy. We use dual-port BRAMs and partition inputs into two groups that each share one of the two ports. In the mesh, there are four direction links (N,S,E,W) plus the core link; as the core is more likely to generate addresses that map to its own router, the core is placed in the partition with only two sharing ports to ensure access to the directory.

C. Integrating NoC with FCUDA Flow

In order to connect the generated cores via NoC, we make specialization to cores and we integrate our changes into FCUDA flow.

1) Memory Ports Combination: Each core is required to read and write to external memory. In the CUDA kernels, the interfaces to external memory are implemented as pointers in the device function. Vivado HLS converts these pointers to bus communication protocol and we connect them to NoC routers. However, each pointer is implemented as a separate port. We solve this problem by combining all the memory interfaces into one port. By using the same memory interface but with different offsets, we can reconstruct the different pointers used in the original kernel.

2) BRAMs Visibility: FCUDA translates CUDA shared memory to BRAMs on the FPGA. The directory system is built to take advantage of those BRAMs. However, the cores generated are a black box and BRAMs are not visible outside the core. We therefore transform these arrays into function parameters and Vivado synthesizes a fast interface to these BRAMs while also allowing external access.

3) Address Mapping: Vivado generated cores bring external memory into local BRAM storage. Although the cores generate valid external addresses, they do not maintain a mapping between external addresses and the location of data in the local BRAMs. We use a static mapping function implemented using the number of thread blocks, and thread block dimensions to ensure that we can determine if a data item is already on-chip based on the external address.

4) Decentralized Control: Existing flow creates a single top-level module that includes several core instantiations with centralized control. However, in order to connect the cores to the NoC, we make changes to the flow to individually synthesize the cores, each with its own control to operate completely independently of other cores.

D. Automated NoC Tool Flow

The NoC and its integration with FCUDA cores are fully automated and integrated into a single cohesive tool chain. Using the tool chain, we automatically generate configurable NoCs integrated with the cores. The NoC design involves computation cores wrapping, top-level network generation, directory systems and integration of NoCs with cores.

IV. Results

We use five CUDA kernels for our evaluation: matrix multiplication (*mm* - multiplication of two arrays), 1D-convolution (*conv1d* - signal processing function that computes area overlap between two functions), coulombic potential (*cp* - computation of electronic potential in a volume containing charged particles), dct (*dct* - transformation from spatial domain to frequency domain) and inverse-dct (*idct* - reconstruction of sequence from dot coefficients). The results of both *dct* and *idct* are similar and hence, we present only the results of *dct* here. We use both the floating-point and integer versions for matrix multiplication. We evaluate our NoC design in terms of performance and external memory accesses. We perform functional simulation of our NoC using modelsim by connecting the NoC to a memory controller and a DDR2 model. The memory controller is generated using the Memory Interface Generator (MIG v3.6) from the Xilinx IP generator tool.

For each benchmark, we use three different network sizes (3 x 3, 6 x 6, and 8 x 8). We scale the data together with the network size proportionally. The total data size is equal to the product of the number of cores and the size of data that one core processes. For example, each core of *mm* works on 256 entries of data and for 64 cores network (8 x 8 NoC), the data size of each array is 16384. For our NoC design, we fix the directory size to be 256 entries. We use the NoC with directory but with disabled sharing as our baseline solution.

Figure 2 shows the comparison of latency in terms of clock cycles when normalized to the baseline configuration. We achieve an average of 16%, 23% and 43% clock cycle reduction for a 3 x 3, 6 x 6 and 8 x 8 NoC, respectively. All benchmarks except *conv1d* show improved performance as we scale the network. The performance of *conv1d* degrades as we increase the size of the network. This is attributed to the sharing overhead and we study the external memory reads to understand possible data sharing in benchmarks.

Figure 3 shows the number of external memory reads comparison. For the baseline, where the data sharing is disabled, the number of external memory reads is equal to the sum of all memory reads by the cores as all the requests have to go to external memory. The data sharing

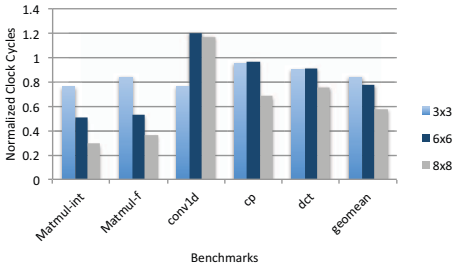

Figure 2. Performance Comparison (clock cycles)

enabled by our NoC has reduced the total memory reads by 53%, 56% and 60% for a 3 x 3, 6 x 6 and 8 x 8 NoC, respectively. For *conv1d*, the number of memory reads is only reduced by 14% and 8% for 6 x 6 and 8 x 8 network. The performance loss of *conv1d* in Figure 2 is partly due to the smaller memory reads reduction.

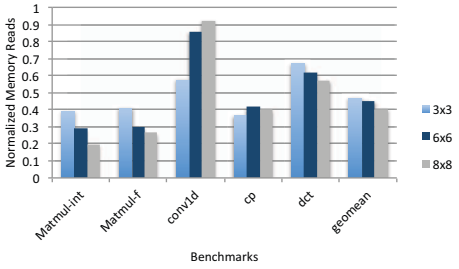

Figure 3. Comparison of External Memory Reads

Finally, we study the impact of the number of cycles that an outstanding merged request waits for the data to be available on-chip before accessing external memory. Figure 4 compares the latency and memory reads between timeout values of 100 and 500 clock cycles for the *mm* benchmark. The timeout value of 100 cycles is considered a baseline and the normalized values are presented along y-axis. We find that the configurable timeout value needs to increase as we scale the size of the network for maximum performance benefit. This is intuitive because with larger networks, the overhead of the packet traversal across the network is higher and outstanding requests have to wait for longer cycles.

V. Conclusion

We developed a network-on-chip integrated with FCUDA flow that enables on-chip data sharing. CUDA programming model allowed us to customize the NoC design with simplicity. On the other hand, we added features including directory-based sharing and outstanding request to exploit data sharing. Our NoC flow automatically generates configurable NoCs integrated with FCUDA cores. We demonstrate reduced external bandwidth demand by 56% and total application latency by 27% for a set of applications.

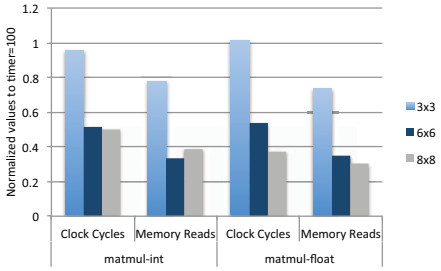

Figure 4. Varying Configurable Timer - Comparison of Clock Cycles and Memory Reads

Acknowledgment

This work was supported by A*STAR under HSSP grant and in part by the CFAR Center, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] K. Alnaes, E. Kristiansen, D. Gustavson, and D. James. Scalable Coherent Interface. In *CompEuro '90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pages 446–453, 1990.

[2] L. Benini. Application Specific NoC Design. In *DATE '06. Proceedings*, volume 1, pages 1–5, 2006.

[3] D. Bertozzi and L. Benini. Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, 2004.

[4] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *Computers, IEEE Transactions on*, C-27(12):1112–1118, 1978.

[5] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From OpenCL to high-performance hardware on FPGAs. In *FPL*, pages 531–534, 2012.

[6] A. Ehliar and D. Liu. An FPGA Based Open Source Network-on-Chip Architecture. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 800–803, 2007.

[7] Forte Design Systems. *Cynthesizer*, 2012. http://www.forteds.com/products/cynthesizer.asp.

[8] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421, 2005.

[9] S. T. Gurumani, H. Cholakkal, Y. Liang, K. Rupnow, and D. Chen. High-level synthesis of multiple dependent CUDA kernels on FPGA. In *ASP-DAC*, pages 305–312, 2013.

[10] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched NoC for FPGA-based systems. *Computers and Digital Techniques, IEE Proceedings* -, 153(3):181–188, 2006.

[11] S. Kwon, S. Pasricha, and J. Cho. POSEIDON: A framework for application-specific Network-on-Chip synthesis for heterogeneous chip multiprocessors. In *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 1–7, 2011.

[12] S. Murali and G. De Micheli. SUNMAP: a tool for automatic topology selection and generation for NoCs. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 914–919, 2004.

[13] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. mei W. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Symposium on Application Specific Processors*, pages 35–42, 2009.

[14] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, W. mei W. Hwu, and J. Cong. Multilevel Granularity Parallelism Synthesis on FPGAs. In *Field-Programmable Custom Computing Machines*, pages 178–185, 2011.

[15] H. Zheng, S. Gurumani, L. Yang, D. Chen, and K. Rupnow. High-level Synthesis with Behavioral level Multi-Cycle Path Analysis. In *FPL*, 2013.