FCUDA-NoC: A Scalable and Efficient Network-on-Chip Implementation for the CUDA-to-FPGA Flow

Yao Chen, Swathi T. Gurumani, *Member, IEEE*, Yun Liang, Guofeng Li, Donghui Guo, *Senior Member, IEEE*, Kyle Rupnow, *Member, IEEE*, and Deming Chen, *Senior Member, IEEE*

***Abstract*— High-level synthesis (HLS) of data-parallel input languages, such as the Compute Unified Device Architecture (CUDA), enables efficient description and implementation of independent computation cores. HLS tools can effectively translate the many threads of computation present in the parallel descriptions into independent, optimized cores. The generated hardware cores often heavily share input data and produce outputs independently. As the number of instantiated cores grows, the off-chip memory bandwidth may be insufficient to meet the demand. Hence, a scalable system architecture and a data-sharing mechanism become necessary for improving system performance. The network-on-chip (NoC) paradigm for intrachip communication has proved to be an efficient alternative to a hierarchical bus or crossbar interconnect, since it can reduce wire routing congestion, and has higher operating frequencies and better scalability for adding new nodes. In this paper, we present a customizable NoC architecture along with a directory-based data-sharing mechanism for an existing CUDA-to-FPGA (FCUDA) flow to enable scalability of our system and improve overall system performance. We build a fully automated FCUDA-NoC generator that takes in CUDA code and custom network parameters as inputs and produces synthesizable register transfer level (RTL) code for the entire NoC system. We implement the NoC system on a VC709 Xilinx evaluation board and evaluate our architecture with a set of benchmarks. The results demonstrate that our FCUDA-NoC design is scalable and efficient and we improve the system execution time by up to 63× and reduce external memory reads by up to 81% compared with a single hardware core implementation.**

Manuscript received May 20, 2015; revised September 8, 2015 and October 26, 2015; accepted October 26, 2015. Date of publication December 8, 2015; date of current version May 20, 2016. This work was supported by the Research Grant for the Human-Centered Cyber-Physical Systems Programme within the Advanced Digital Sciences Center through the Agency for Science, Technology and Research, Singapore.

Y. Chen is with the College of Electronic Information and Optical Engineering, Nankai University, Tianjin 300071, China, and also with the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA (e-mail: yaochen@mail.nankai.edu.cn).

S. T. Gurumani is with the Advanced Digital Sciences Center, Singapore 138632 (e-mail: swathi.g@adsc.com.sg).

Y. Liang is with the School of Electrical Engineering and Computer Science, Peking University, Beijing 100871, China (e-mail: ericlyun@pku.edu.cn).

G. Li is with the College of Electronic Information and Optical Engineering, Nankai University, Tianjin 300071, China (e-mail: ligf@nankai.edu.cn).

D. Guo is with the School of Information Science and Engineering, Xiamen University, Xiamen 361006, China (e-mail: dhguo@xmu.edu.cn).

K. Rupnow is with Advanced Digital Sciences Center, Singapore 138632 (e-mail: k.rupnow@adsc.com.sg).

D. Chen is with the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA (e-mail: dchen@illinois.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2015.2497259

***Index Terms*— CUDA, high-level synthesis (HLS), network-on-chip (NoC), parallel languages.**

I. Introduction

HIGH-LEVEL synthesis (HLS) has increasingly been adopted in hardware design to improve design time and to perform design space exploration. Development, debug, and design space exploration in high-level languages allow improved breadth of exploration and reduced designer's effort.

A variety of input languages have been used with HLS, including Java [1], Haskell [2], [3], C/C++ [4]–[8], OpenCL [9]–[11], C# [12], SystemC [13], [14], and CUDA [15]–[18]. In general, using serial languages, such as C/C++, HLS tools use user input and automatic parallelization to generate a single, monolithic accelerator kernel. In contrast, using parallel languages, HLS tools generate small simple accelerators for independent threads of computation with the intention that multiple accelerators are instantiated to scale implemented parallelism. As a popular parallel programming language, there are many existing kernels in CUDA, and CUDA-to-FPGA (FCUDA) can explore kernel computation with FPGAs as an accelerator [15]–[18]. This also provides a common programming language that can program heterogeneous computing platforms that contain both graphic processing units (GPUs) and FPGAs [18].

In the FCUDA flow [15]–[18], each hardware core has private on-chip memory and computation logic, and multiple cores are instantiated to improve throughput and latency. This throughput-oriented synthesis allows fine-grained scaling of the parallelism but also places stress on on-chip communication and external memory bandwidth. When instantiating many cores, they must share access to external memory ports. Furthermore, the cores may process overlapping data; thus, the opportunity to share data on-chip can reduce off-chip bandwidth pressure. For example, with cores accelerating matrix multiplication (Fig. 1), independent blocks process overlapping input data that can be shared on-chip.

For a multicore accelerator design, cores must be interconnected to share access to external memory ports, as well as to enable intercore communication for data-sharing. Cores may be interconnected through a shared bus, point-to-point connections, or a network-on-chip (NoC). Shared busses are area efficient but do not scale in total bandwidth as the number of cores increases. In contrast, point-to-point interconnections

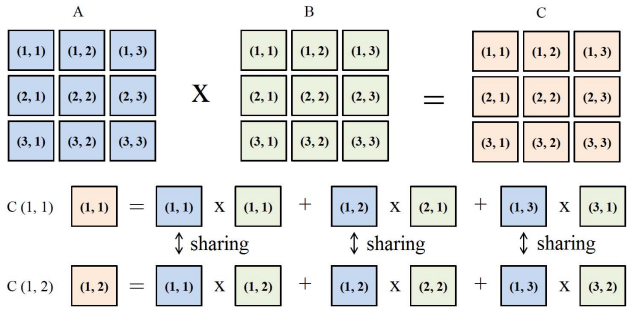


Fig. 1. Example of data sharing.

scale the total bandwidth, but the $O(n^2)$ scaling in connection area is infeasible for large designs. Between these two options, NoC interconnects balance bandwidth scaling and area consumption.

NoC router designs have been well studied in recent years [19]–[28]. In this paper, we significantly enhance an existing NoC router to provide flexible bandwidth depending on area-performance tradeoff, and integrate it into our prior FCUDA flow and study the scalability and efficiency of the NoC design. We develop an automated flow that generates a fully configurable NoC system with a set of input parameters, such as network size, number of router ports, flit widths, and type of external memory (DDR2/DDR3). In addition, we develop a distributed data-sharing mechanism in our NoC system that efficiently reuses on-chip data and reduces off-chip bandwidth pressure. Efficient data sharing is critical to the NoC to improve performance and scalability before external bandwidth is saturated.

This paper contributes to the study of throughput-oriented HLS with the following.

1) A method for transparently integrating HLS-generated cores with an NoC.
2) A scalable and fully configurable NoC that effectively integrates many HLS cores into a system-level design.
3) A high-performance data-sharing mechanism that allows efficient external bandwidth utilization.
4) A flexible NoC generation flow that allows exploration and selection of application-specific NoC architecture.
5) An HLS-enabled fully automated system generation flow that generates the entire NoC design for a specific application.

The rest of this paper is organized as follows. We first review related works in HLS and NoC design in Section II. We discuss the FCUDA hardware core generation flow in Section III and the NoC design in Section IV. Finally, we present and analyze the results from the experimental evaluation of our FCUDA-NoC systems in Section V.

II. Related Work

HLS has heavily been studied, with active projects in both industry and academia [1]–[18]. Many of these approaches target single monolithic cores with all parallelism optimizations internal to the core design. However, several works have considered a throughput-oriented design [9]–[11], [15]–[18], [29], where the cores are intended to be multiply instantiated at the system level to improve system throughput. This approach allows for scaling through multiple instantiation, but the design and optimization of the cores' interconnection is understudied.

There is significant prior work in the design and generation of general networks for both application-specific integrated circuit (ASIC)- and FPGA-based systems [19]–[28], [30]–[32]. In prior work, networks are either designed independently as a generic network used with a variety of cores [22], [24]–[26], [30]–[33], or as an application-specific NoC [20], [21], [27]. Several such works include automatic generation of NoCs for FPGA platforms [22], [25], [34]–[36].

Despite these earlier works, there is no prior work integrating NoC designs with HLS-produced cores. In order to effectively integrate HLS cores and a network, it is important to transparently retain the core's interface with scalable performance. Thus, in this paper, we first extend our prior FCUDA synthesis flow [15]–[18] to enable network support for the HLS-generated hardware computing cores. We then design a fully configurable NoC router based on a prior open-source router [22]. In addition, we develop a fully automated NoC generation framework that produces a configurable and complete NoC system providing scalable and efficient performance with multiple instantiated HLS-generated cores.

III. Background

Our FCUDA-NoC platform customizes the FCUDA flow to enable transparent integration of the HLS-generated accelerator cores in an NoC, including both efficient external memory bandwidth sharing and a mechanism for automatically sharing data between cores' local memories. Before we discuss the customizations to the FCUDA flow, the NoC router architecture, and design of the data-sharing mechanism, we first introduce the FCUDA flow and the original NoC router and interconnection architecture.

A. CUDA-to-RTL Flow

We choose our existing CUDA-to-RTL for FPGA (FCUDA) flow [15]–[18] for integration with the NoC system. We preferred the FCUDA flow for the following reasons: 1) earlier work [18] has shown that there is an intrinsic advantage in using a parallel language input for HLS to capture parallel computations more naturally; 2) CUDA is widely used and can provide a common programming language to program heterogeneous compute platforms containing both GPUs and FPGAs; and 3) FCUDA can explore kernel computation with FPGAs as an alternative accelerator for the existing CUDA kernels. Our FCUDA flow is based on source-to-source translation of accelerator kernels written in CUDA. CUDA kernels may be annotated with user pragmas to define optimizations, such as loop unrolling, loop pipelining, data merging, and grouping of computation and communication operations. Alternatively, the design space automation [16] and compute/data partitioning [17] may be performed automatically.

After optimization, our flow translates the original CUDA code into C code annotated with pragmas for Xilinx Vivado

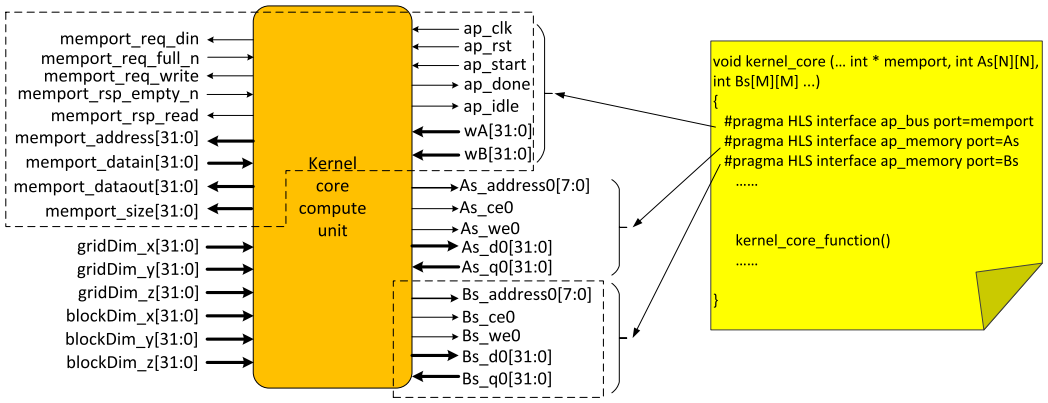


Fig. 2. Interface of a generated kernel core.

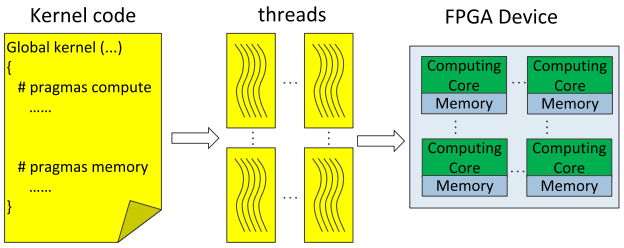


Fig. 3. FCUDA mapping.

HLS. Vivado is then used to synthesize the C code into RTL that can be subsequently implemented on an FPGA. Vivado produces a single accelerator core that uses the Advanced eXtensible Interface-compatible ap_bus interface. An example kernel core along with the snippet of the code using pragmas is shown in Fig. 2. The arguments As and Bs use ap_memory interface, and memport uses the ap_bus interface protocol. The core is shown to be generated appropriately with the corresponding protocols. The core also has ports for gridDims and blockDims to specify workload distribution. Though Vivado generates efficient single core accelerators, instantiation of multiple cores, connection of communication signals between the cores, and platform integration with memory controllers are left to the user. Furthermore, during synthesis, Vivado HLS has no contextual knowledge about the number of cores that will be instantiated, the total amount of work per accelerator core or the potential for data-sharing between cores; the optimization of data-sharing must be performed during core integration.

Each accelerator core is allocated private on-chip block random access memory (BRAM), registers, and functional units. One accelerator core execution will perform computation corresponding to one or more CUDA thread block's work, as shown in Fig. 3. Thus, optimization within a single core explores fine-grained parallelism within a small group of threads, and coarse-grained parallelism is further explored through multiple instantiation of accelerator cores. These explorations trade resource sharing among threads in the same core, total work per-core execution and parallelism among multiple cores.

B. Existing NoC Architecture

We develop our NoC architecture based on an existing open-source NoC originally designed for Xilinx Virtex-4 FPGAs [22]. The open-source router is designed to have four or five input/output ports. Input data are buffered through first-inputs–first-outputs (FIFOs) at the input ports: if the desired output port is available, data can propagate in a single cycle. Each output port can transmit a single flit per cycle, and hence the input FIFO will be emptied as fast as it can be filled. However, if no output port would be available, the input FIFO will quickly fill up. To avoid overruns, the input module will signal the sender that no further data should be sent as soon as it reaches almost full level. Thus, this back-pressure signal stalls the neighboring routers before FIFOs are full. The design uses wormhole routing to avoid the need for larger packet buffers and to reduce latency. The NoC is statically routed: during network generation, all routes are statically precomputed and filled into each router's routing table. XY-routing is used to avoid adding complexity to the hardware. In this paper, we select a 2-D-mesh interconnect topology for its good bandwidth and scalability characteristics and simple organization [37]. We customize the original, open-source router to extend the maximum total network size and maximum connection width. We will discuss our modifications to the router and other NoC features in Section IV.

IV. FCUDA-NoC Platform

Our FCUDA-NoC platform generation consists of multiple stages and is shown in Fig. 4. The platform generation script takes in CUDA code along with a set of input parameters describing the network, including network size, packet size, and directory size. In the first stage, CUDA is translated into annotated C code, with parameterized inputs corresponding to block dimension, thread dimension, and block and thread indices in CUDA. Using these input parameters, the C-level implementation computes the workload distribution [15], [16]. In the second stage, we use Vivado HLS to generate a computing core from the annotated C code; this core can be instantiated multiple times: each core has unique dimension and index parameter settings to distribute total workload. In the NoC integration stage, a single compute core is integrated into a wrapper module with a customizable NoC router, dual-ported BRAMs for local storage, and arbitration logic. Multiple instantiations of the wrapper module are connected in a top-level module. The generated top-level module instantiates multiple cores, connects them in the 2-D-mesh topology,

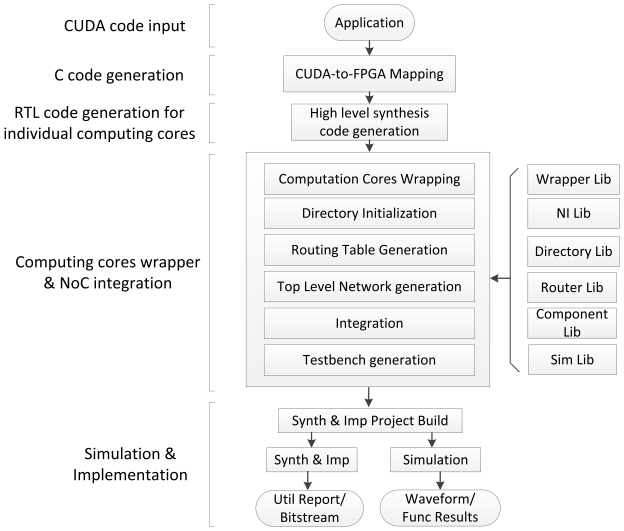


Fig. 4. NoC tool flow.

and connects the computation cores to the memory controller. Our platform generation supports DDR2-, DDR3-, and BRAM-based memory models. The output from this third stage is the generated top-level module and the corresponding testbench files. Finally, the simulation and implementation projects are built for the generated NoC system automatically.

In order to automate core generation, NoC generation and platform integration, as well as optimize data-sharing opportunity, we develop a set of new features on top of the original infrastructure, including address space mapping, BRAM accessibility, and distributed control. We will now discuss the core generation in detail.

A. CUDA-to-RTL Core Generation

In an NoC-based system, because all communications pass through the network, all ports must be merged into a single memory space. Furthermore, to integrate the cores transparently, local memories must be integrated so that both the core and the network can interact with the memories. To map all input data into a single memory space, each input array is assigned an offset address. A data-sharing mechanism must additionally maintain a mapping between external addresses and the location of data in local BRAMs. Finally, each core has private control. We will now discuss the implementation of these new features in more detail.

1) Address Space Mapping: Inside the CUDA kernels, interfaces to external memory are pointers to device memory. Vivado HLS converts these pointers to independent communication ports. However, this will increase the number of ports if the CUDA kernel tends to use multiple pointers. Furthermore, NoC routers do not scale well with the number of ports per router. We solve this problem by combining all the memory interfaces into one port. We map multiple individual arguments into a single external port with internally applied offsets.

Although Vivado-generated cores use valid memory addresses to request data, the cores do not maintain a mapping between external addresses and values stored in the local (scratchpad style) BRAM storage. The cores iteratively request and write back data to different data portions in external

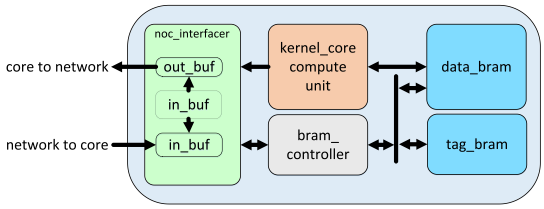


Fig. 5. Wrapped kernel core.

memory, the same physical location in local storage may be reused multiple times for different external memory addresses. Thus, without an additional mapping between external memory addresses and local storage location, we cannot determine if a particular global address has a local copy. We generate a mapping function to ensure that for any external memory address which can determine whether valid data for that address is stored in a BRAM or not. A one-to-one mapping ensures that there is no false data sharing. We implement the mapping function statically using the number of thread blocks, thread block dimensions, and instantiated core number to determine the storage location in local BRAMs.

2) BRAM Accessibility: BRAMs are assigned to the cores to temporarily store the requested data from external memory and intermediate computing results. Xilinx Vivado can take advantage of dual-ported BRAMs to improve memory throughput. However, to integrate these BRAMs into an NoC transparently, we reserve one BRAM port for the NoC for memory transactions and data-sharing transactions, and the other port for the compute core, as shown in Fig. 5. To implement sharing of the dual-ported BRAMs, we export all input and output arrays as function parameters so that we can instantiate and connect the BRAMs in the NoC integration stage. Thus, the BRAM can be accessed either by the core or by the NoC interface through a BRAM controller.

3) Control: The original FCUDA flow creates a single top-level module that includes several core instantiations. It also has a centralized control for mapping the computation onto separate cores [15]–[17], as shown in Fig. 6(a). There is one set of external interfaces to the core, and the centralized control is responsible for workload distribution to the cores. However, for improved scalability, each core can compute workload distribution independently using the per-core block dimension, thread dimension, and block and thread index parameters [Fig. 6(b)]. These parameters are fixed for any particular system design; logic synthesis may perform local optimizations of each core instantiation.

B. Data-Sharing Mechanism

A critical aspect of the performance scalability for an FCUDA-NoC generated system is the ability to efficiently share on-chip data. An efficient sharing mechanism enables better use of external bandwidth as well as reducing average request latency when latency of an on-chip access is less than the latency of an off-chip access. Thus, an efficient sharing mechanism improves both performance and scalability of the networks before external bandwidth is saturated.

In this paper, it is also critical that the sharing mechanism is transparent; cores are synthesized as independent blocks

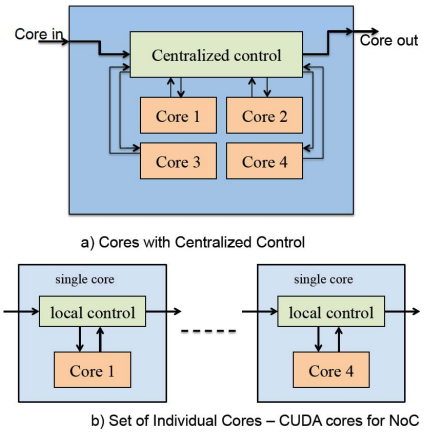


Fig. 6. Centralized control and decentralized control.

of datapath, memory, and control that are not designed to facilitate sharing of memory values with other cores. Thus, we develop a directory-based data-sharing mechanism that serves as an intermediary between the HLS-generated cores and the external memory system. Using the address mapping described in Section IV-A1, we can perform a lookup to determine if the requested data are already on-chip. To further reduce external memory bandwidth demand, we also support automatic merging of multiple memory requests to the same address. We will now discuss these features in more detail.

Our directory-based data sharing is conceptually similar to cache coherence [38]–[41], but with several key distinctions. In the CUDA programming model, cores may share input data, but outputs are not guaranteed to be coherent. The CUDA programming model supports local synchronization within a thread block, which would be implemented inside one computation core, or global synchronization between kernel calls through global memory, which would be implemented as separate core types. Thus, we do not need to track data updates; the directory-based sharing is purely a performance optimization, and we do not need to track every item in the memory space to guarantee correctness. Thus, we can customize the number of tracked memory locations (directory size) without affecting correctness.

1) Directory-Based Data Sharing: In the FCUDA-NoC system, each node in the network is connected to an NoC router which consists of a directory and a routing logic part. External memory addresses are statically assigned to a home directory in the routers, such that the location of the directory for any external address is statically known. However, in order to effectively scale the system, the directory is not necessarily allocated enough storage to simultaneously track all memory addresses. As discussed above, this does not affect correctness but may affect sharing opportunity; if a corresponding directory entry is not found for an address, the access will produce an external memory access. A large number of directory entries ensure that when data are on-chip, it will be found correctly but also consumes resources that may have been used for additional cores. We will examine the relation between directory size and performance in Section V.

Input addresses are split into tag, index, and offset fields similar to typical caching. The index maps each address in the

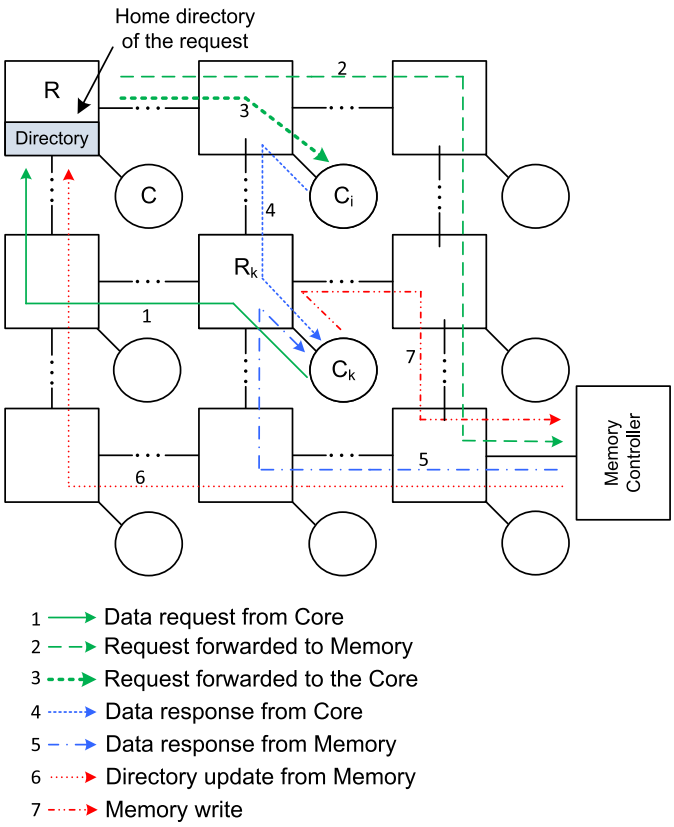


Fig. 7. Directory protocol.

system to a single directory index using the mapping function, as described in Section IV-A1. An update to the directory simply erases the prior information that was stored at the index.

Each directory entry contains a tag field, location field, and valid bit. As in typical caching, the tag is used to distinguish between addresses that can be stored to the same directory index. The location field contains the network address of the core that contains the requested piece of data, and the valid bit specifies whether or not the entry contains valid data.

The directory protocol is shown in Fig. 7. On a memory read (1), the request is first routed to the home directory node for that address. If the directory has a tag mismatch or invalid entry, the request is forwarded to the memory controller (2), and once data returns, the directory is updated (6), and the requesting core receives data (5). The memory controller sends two network packets, one to the directory and one to the requesting core to separate the data payload and directory update. If the directory has a tag match, the request is instead forwarded to the core with the data (3), which forward the data to the requesting core (4).

On a memory write, the directory system is ignored. As per the CUDA programming model, it is not permitted for other thread blocks (other instantiated hardware 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 (7). Only memory controller responses are allowed to update directory entries; this simplifies the protocol as routers and cores cannot generate update packets.

If there are multiple simultaneous requests for the same (off-chip) data, there would be multiple external memory requests. These duplicated requests will increase average memory access latency and external memory bandwidth use. Thus, we also track outstanding memory requests at the addresses' home node. When a subsequent memory request reads the directory and determines that the mapping is not valid but has already produced an outstanding memory request, it will wait for the data to return instead of producing an additional memory access. When the data returns to the core that produced the original memory request, it will also be forwarded to the second requesting core.

There is a possibility that in the intermediate time between moving the waiting memory transaction from the address' home node to the actual location data will return to, the data are already evicted before being forwarded. This may cause the request to wait indefinitely for the data. Thus, we also have a waiting timeout; if a waiting request is not serviced in the specified number of cycles, it will produce an external memory request.

The directory system is implemented in a single dual-ported BRAM shared between all the input ports of the NoC router. In the 2-D mesh, there are four directional links and the link to the core; because the core is more likely to generate addresses mapped to its own router, the core is given priority to the directory to ensure low-latency address lookups. Furthermore, to ensure forward progress and to improve sharing opportunities, directory updates are given priority over read accesses.

C. CUDA-to-RTL NoC Router Architecture

The generated network for an FCUDA-NoC system should have several characteristics in order to make the system scalable in both the number of instantiated cores as well as the performance of the resulting system. Thus, an NoC router must be as follows.

1) *Area Efficient:* The router design should have minimal area overhead.
2) *Performance Efficient:* The router design should not be the critical path, or affect achievable frequency.
3) *Flexible Bandwidth:* The router should be customizable to trade implementation area and communication bandwidth.

Thus, to meet these goals, we extend the original open-source router design in order to increase flexibility in the design and implementation of the NoC. Before we discuss details of router customization, we first introduce the packet format for the interface protocol and the routing protocol.

1) Packet and Flit Format: In our statically routed NoC, the packet format contains standard fields for tracking packet validity, backpressure, and the next hop as well as the source, destination, address, and data for the packet. The fields contained in a packet are shown in Table I. In order to simplify the communication protocol, the packet and flit are designed to have the same header fields based on signals in Table I. Fig. 8 shows the packet format and flit format in the network. The link width between two components in the network is equal to the flit size, consisting of address, backpressure, flags, and data payload. A packet may require multiple flits at smaller flit sizes. In our NoC, the routers and cores are individually addressed, so that data response packets are routed directly to cores, and directory lookups or updates are routed to the cores' routers. For different network sizes, the n bits of *dest* and *src* are thus chosen, such that all routers and cores can be assigned unique addresses. We additionally reserve a single address to represent the memory controller. Thus, the total required bits can be computed as in

$$2^n \geq 2 \times (\text{node number}) + 1. \quad (1)$$

In order to further reduce the connection width between routers, the *data* and *addr* fields can be divided into small parts and be transported in multiple flit payloads.

In addition, to support the transparent data-sharing mechanism, we extend the types of packets to include directory-lookup and directory-update packets for determining whether an external memory address is already in local BRAMs, and updating the storage location, respectively. A list of packet types is shown in Table II.

2) NoC Router: We extend the original router design presented in Section III-B to improve flexibility in both target platform as well as architectural parameters for the network design. First, we eliminate use of any device specific hardware primitives. Then, we parameterize router parameters. Fig. 9 shows the block diagram of the extended router. The router consists of two main components: 1) a user configurable directory and 2) a user configurable routing logic. The router is

TABLE I
LIST OF FIELDS IN A PACKET

Name	Description
sendokbit	Backpressure bit, remote node is ready for receiving data
sendbit	Valid bit, set value if packet is valid
nexthop	Pre-computed output port this packet should be routed
lastbit	Last data in a transaction
dest	Destination node of this packet
src	Source node of this packet
type	Packet type
data	Data payload of the packet
addr	Requested address for data

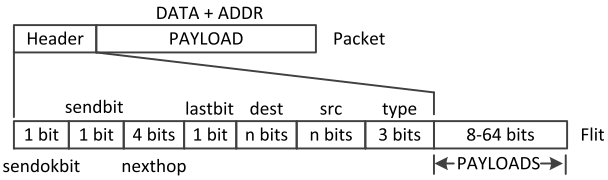


Fig. 8. Packet and flit format.

TABLE II
LIST OF PACKET TYPES

Type	Description
TYPE_REQUEST	A memory read request
TYPE_RESPONSE_ADDR	A directory update
TYPE_RESPONSE_DATA	A data response
TYPE_C_REQ	Request redirected to a core
TYPE_WRITE	Memory write
TYPE_OUTSTANDING	Outstanding request match

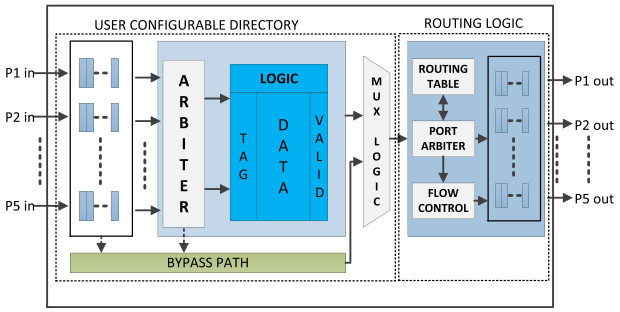


Fig. 9. Architectural block diagram of FCUDA-NoC router.

shown with a configurable number of input/output ports, each with its own FIFOs. We further explain and refer to this figure while discussing the router components in the following. The router features are in the following:

1) configurable number of input/output ports;
2) configurable network size;
3) configurable flit payload size, 1/2/4/8 bytes;
4) user configurable directory entry size;
5) user-specified routing table.

We now discuss the details of the important parameters for the router components.

Flit Size: The flit size corresponds to the width of the communication channel between routers, and thus higher flit sizes will also correspond to increased routing resources as the system size scales. The flit header size is changed with the network size accordingly, as we described in Section IV-C1. As a decisive parameter for the flit width, the flit payload size can be configured in increments of 1 byte (1/8 of the data payload size). In this paper, we test flit payload sizes of 1, 2, 4, and 8 bytes, which corresponds to 28, 36, 52, and 84 bits with a 5-bit address field (up to 3×3 network, addressing routers and cores indvidually), and 36, 44, 60, and 92 bits with a 9-bit address (up to 12×12 network). The wormhole flow control used in the original open-source router is retained in our router [22]. The last bit signal is used when the flit size is smaller than the packet size to ensure that all data in one packet can be sent without interruption.

Input/Output Ports: Each input and output port is associated with FIFO buffers that depend on the flit setting. The total capacity of the buffers is user specifiable, and may store multiple packets. Depending on the packet type, the input ports will either communicate with the directory system (for data sharing), or go through the bypass path to the routing logic to send the flit to specified output port or port buffers. The bypass path is shown in Fig. 9 (green region) connecting the input FIFOs directly to the routing logic through a multiplexor. Only address requests and directory-update packets access the directory, and both packets only access the directory presented in address' home node. Thus, most router traversals will not need to access the directory and we optimize for this common case of skipping directory access through a bypass path to minimize latency. In addition, the arbiter (Fig. 9) is responsible for arbitrating BRAM port access between multiple input ports using a simple round-robin scheme. Without port access conflicts and directory access, the router can receive/send one flit in two clock cycles. With a directory access, it requires between two to eight additional clock cycles including cycles for buffering of flit data and accessing the directory entry.

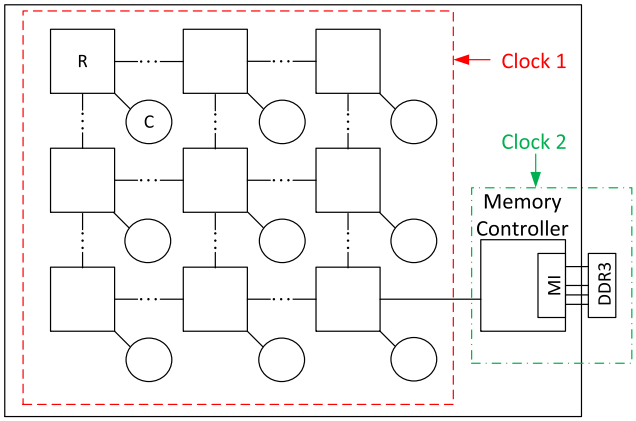


Fig. 10. Generated system architecture. Key R: FCUDA-NoC router. Key C: CUDA kernel hardware core.

Routing and Topology: The NoC is statically routed using *XY*-routing and a one-hot encoded routing table for determining the next hop based on final destination The wormhole flow control of the open-source router is retained in our design [42]. In regular topologies, static routing is straightforward to compute and implement. The static routing minimizes resources spent on routing logic.

Although our router architecture could implement multiple different topologies, we concentrate on the 2-D-mesh topology for an efficient tradeoff in area, cross-sectional bandwidth, scalability, and routability of the communication links. The generated system architecture is shown in Fig. 10. Each computation core (C) is connected to a router (R) using one port of the router, while the other ports are used to connect to the other routers in a 2-D-mesh topology. The NoC and the memory operate at two different clock frequencies, and communication between the memory controller and the NoC is through asynchronous FIFO embedded in the memory controller. This is designed to enable the network part to run at a different clock frequency from the memory controller, thus enables support for different external memories (DDR2/DDR3).

We automatically generate routing tables for the network, using a modified A*-based mechanism [43] to compute the all-pairs shortest path between the cores in the network. The routing tables are generated and automatically populated in the routers by our automated NoC generation flow.

V. Experiments

We will now present the experimental evaluation of our FCUDA-NoC systems. First, we will demonstrate the scalability of the NoC architecture in terms of area consumption and its effect on the maximum number of instantiated cores. We will then demonstrate performance scalability with and without the on-chip data-sharing feature. We also present the impact of on-chip data sharing in reducing the total number of external memory transactions. Finally, we will evaluate the impact of different network settings on the system performance.

TABLE III
CUDA KERNELS

Application	Maximum Data Dimension	Description
Matrix multiplication (*mm*)	16384 × 16384 array	Computes multiplication of two arrays
Coulombic potential (*cp*)	16384 × 16384 array, 512 atoms	Computation of electronic potential in a volume containing charged atoms
Discrete cosine transform (*dct*)	4096 × 4096 array, 128 constants	Computation to transform an image from spatial domain to frequency domain
Inverse discrete cosine transform(*idct*)	4096 × 4096 array, 128 constants	Computation to reconstruct a sequence from its DCT coefficients
1D-convolution (*conv1d*)	16384 × 16384 array	A signal processing function that gives the area overlap between two functions

TABLE IV
ROUTER COMPARISON

	LUTs	Regs	Cycle/hop
STU router [33]	1692	1345	7
CONNECT router [34]	1578	460	2
FCUDA-NoC router	1823	890	2

For our experiments, we use five benchmark applications. The primary goal of this paper is to evaluate the NoC, in particular, the benefits of on-chip data-sharing feature. Thus, we select benchmarks with both low and high data-sharing opportunities. The applications, data dimensions, and descriptions of the CUDA kernels are presented in Table III.

For our experiments, we initially perform functional simulation using ModelSim 10.1 and then perform logic synthesis using Xilinx Vivado 14.2 targeting a Xilinx Virtex-7 XC7VX690T FPGA chip. For simulations, we use the DDR3 memory model and generate a memory controller (mig7.0.1) using the memory interface generator tool integrated in Xilinx Vivado 14.2 [44]. All designs are synthesized on a 64-core AMD Opteron server with 256 GB of RAM. We then perform board-level implementation for the designs that are synthesized (including placement and routing) within the routing resource limit using a Xilinx VC709 platform, which contains a Xilinx Virtex-7 FPGA. The VC709 platform contains DDR3 memory. We use the highest achievable clock frequency for implementing our designs on the board. We also validated our simulation-based latency in clock cycles with on-board measurements and found out that the variation was $<1.5\%$. The main difference between cycle accurate simulation and on-board implementation is caused by the nondeterministic effects of arbitration and memory initialization time. In addition, we use functional simulation to estimate the runtime of larger designs that cannot fit on the VC709 due to routing resource restrictions, to evaluate network scalability and demonstrate capabilities on future FPGAs that may have more routing resources.

In the initial experiments, we synthesize the individual network routers and cores to gather area and achievable frequency information. We then analytically compute the maximum number of instantiable cores, and compare these analytical estimates to fully synthesized NoC systems.

A. Resource Usage

We first compare our FCUDA-NoC router to two other popular open-source routers with their corresponding best performance settings in terms of resource usage and clock cycles per hop in Table IV. Both of these two routers are well known as their high-performance router implementation. When all the routers are set to 64-bits data width, the

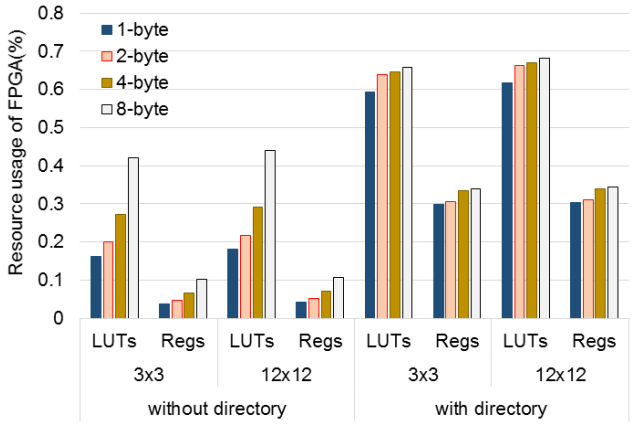


Fig. 11. Router resource usage for different flit payload sizes and different address field sizes with and without directory.

CONNECT [34] router has the smallest resource utilization and the same cycle/hop efficiency as our FCUDA-NoC router. The CONNECT router provides a Web page-based generator to produce an optimized network, but it is not straightforward to integrate computation cores and to customize the network. The Stanford University router [33] has similar resource consumption as our FCUDA-NoC router but has lower cycle/hop efficiency. Compared with these other routers, our selected original router has similar or superior area and performance.

We then synthesize the individual NoC routers and FCUDA accelerator cores independently to evaluate resource usage and achievable frequency. Fig. 11 shows the resource usage of a single router for different flit data payload sizes and address field sizes from 5 bit (up to 3×3 network) to 9 bit (up to 12×12 network), the routing table size is changing with the network size as well. As the data payload, address field and routing table size increase, there is only minor variation in the per-router resource usage. In all the cases, a single router uses $<0.68\%$ of the Virtex-7 FPGA, with lookup table (LUT) usage as the constraining resource. The router without a directory does not use BRAM resources; with a directory, each router consumes a single BRAM, which corresponds to 0.03% of available BRAMs (one block out of 2940 blocks of 18-Kbit RAMs). All versions of the router have achievable frequency greater than 400 MHz when synthesized individually.

We also synthesize each FCUDA core in both the original version [15]–[17] as well as the version that merges memory interfaces and enables dual-ported memory use, for NoC integration (Fig. 12). Similar to the NoC routers, each individual core consumes $<1.2\%$ of the target Virtex-7 FPGA, and the LUT usage is the limiting resource in all the cases. The version for NoC integration consumes slightly more resources. Because each FCUDA core must use only one port

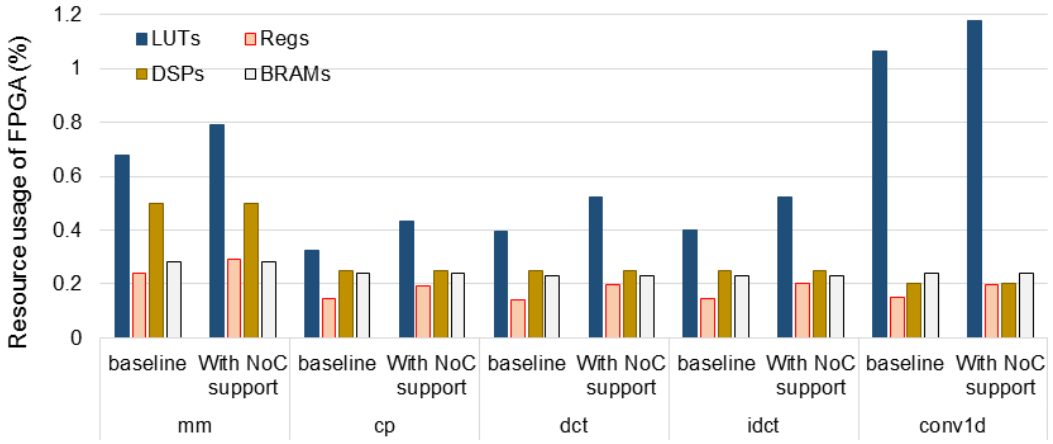


Fig. 12. Kernel core resource utilization for different benchmarks with/without NoC support.

of the dual-ported memories, the internal logic is partially serialized, requiring additional registers and control logic. All the individual benchmark FCUDA cores with and without NoC support can achieve a clock frequency higher than 142 MHz.

Comparing the synthesis results of the router and core, we find that the smallest router is less than half the size of the smallest FCUDA core, and even the largest router version is roughly the same size as the FCUDA cores. Using the synthesis data of individual cores and NoC routers, we build a simple analytical model to estimate the resource consumption of the network as we scale the number of nodes; we do not consider routing and connection overhead. With this model, we analytically compute the maximum number of cores that can be instantiated in our Xilinx Virtex-7 FPGA. Let us assume N is the number of computing nodes, also equal to the number of routers in our system; R_{Res} is the resource consumption of the NoC router with the smallest flit setting without directory; $\Delta R'_{\text{Res}}$ is the additional resource consumption of increasing the flit width; ΔD_{Res} is the additional resource consumption of adding the directory system on top of the router; $\sum_0^N C_{\text{Res}}$ is the total resource consumption of the kernel cores, and M_{Res} is the constant resource consumption of the external memory controller instantiated in our system. We now model the sum of all resources S_{Res} for an ideal implementation that has zero-cost core interconnection using

$$S_{\text{Res}} = \sum_0^N \left(R_{\text{Res}} + \Delta D_{\text{Res}} + \Delta R'_{\text{Res}}\right) + \sum_0^N C_{\text{Res}} + M_{\text{Res}}. \tag{2}$$

Setting R_{Res} and $\Delta R'_{\text{Res}}$ to 0 demonstrates the maximum number of instantiable FCUDA cores if we did not have the network router overhead. Next, we also calculate the maximum number of instantiations using the smallest (1-byte flit payload, no directory) and the largest (full data payload, with directory) versions of the NoC router. Fig. 13 shows the maximum instantiable cores for each combination of FCUDA core and router, and since our system is in a 2-D-mesh topology, the sizes are shown as $n * n$. Though the router overhead is fixed in terms of the number of resources, the impact on the number of instantiable cores is dependent on the size of the FCUDA kernel core. *cp* being the smallest kernel core

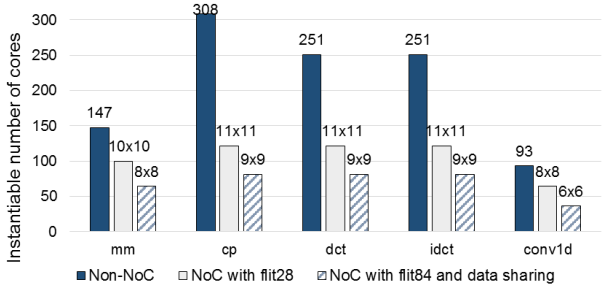


Fig. 13. Number of cores that can be instantiated in the target FPGA.

(from Fig. 12), adding the router significantly reduces the number of instantiable cores, while the difference in instantiable cores is less for *conv1d*. On average, the smallest and the largest router reduces maximum instantiable cores by 41.8% and 65.4%, respectively. This optimistic modeling of resources helps the user to estimate the maximum possible network size for each of the benchmark application. In practice, a synthesized design will use more resources than our analytical estimation, since we do not attempt to model additional resources due to routing and resource contention.

In addition, we estimated the routing capacity of our targeted platform using the FPGA routing resource estimation method presented in [45] and determine that a 64-router network consumes more than 95% of the global routing resources. For this reason, placement and routing failed for networks of 64 or more nodes. After we perform full FCUDA-NoC system synthesis (Section V-B), we determine that our analytically computed area consumption underestimates actual area by 5.4% on average. However, this simple analytical model helps estimate the maximum instantiable network either based on resource usage or routing capacity. With sufficient routing ability, the user can maximize instantiations using a small router, or sacrificing maximum instantiable cores to use a larger router. With constrained routing resources, the user can freely maximize router and core features, understanding that the maximum network size may be constrained by routing resources rather than total design area. The impact of the data sharing will be explored later in this section.

Our FCUDA-NoC architecture can potentially instantiate large networks of cores. Depending on core design, the

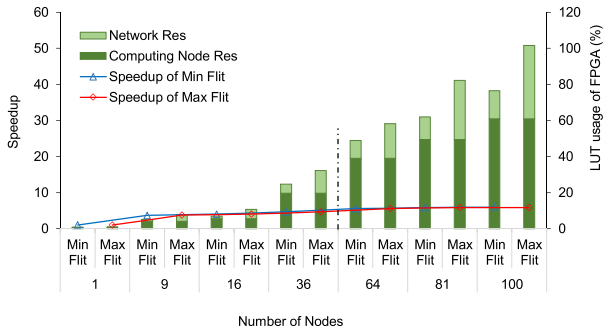


Fig. 14. Performance and resource scaling of *mm* without directory.

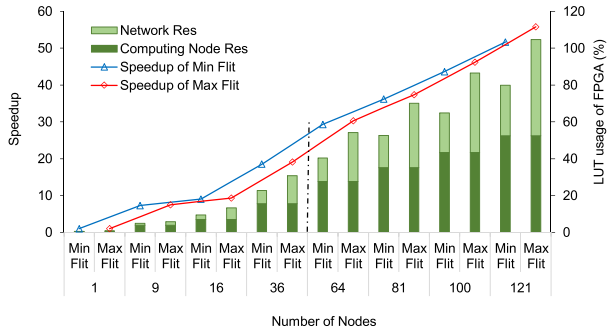


Fig. 15. Performance and resource scaling of *cp* without directory.

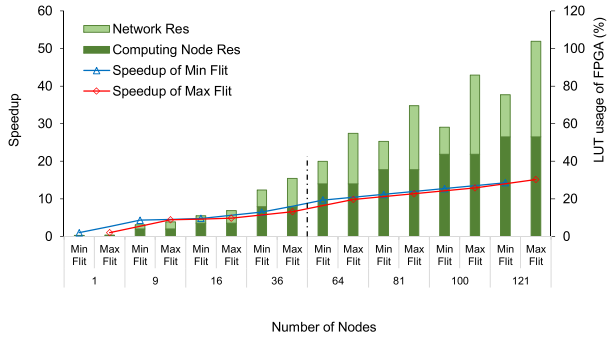


Fig. 16. Performance and resource scaling of *dct* without directory.

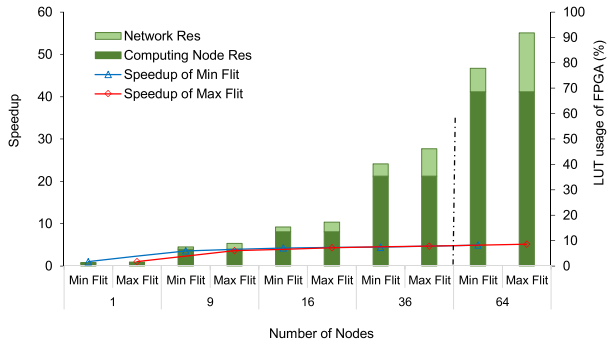


Fig. 17. Performance and resource scaling of *conv1d* without directory.

Virtex-7 may have sufficient area for over 100 cores, yet the maximum network size is <64. However, increasing the number of instantiated cores does not necessarily guarantee improved performance. Furthermore, for efficient workload distribution, our mesh topology requires a square array of cores with the mesh dimensions being an exact divisor of the CUDA kernel workload. Thus, in the board-level performance evaluations, we use network sizes of 1, 3×3, 4×4, and 6×6 cores.

Although network sizes of 8×8 and above are infeasible on our Virtex-7, we use frequency and resource consumption results to demonstrate the network scalability assuming the availability of a sufficiently capable architecture. In Section V-B, we will demonstrate the area and performance scaling properties of our FCUDA-NoC platform with different network settings followed by a study of the effectiveness of on-chip data sharing.

B. Scalability Evaluation

We now study the scalability of the NoC systems generated for the benchmarks. The network size scales from one core to the maximum mesh network size that can fit on our VC709 platform.

First, we present the scaling of performance and resource utilization without on-chip data sharing for the benchmarks in Figs. 14–17. To evaluate scalability, we compute speedup compared with designs with a single core. We only present the LUT utilization data, since we demonstrated earlier that LUTs are the limiting resources for scaling of our network. The network size impacts the achievable clock frequency of the system; we achieve 140 MHz for all one core implementation, 125 MHz for the 9 and 16 core implementation, and 100 MHz for the 36 core implementation. We use 100-MHz clock for the larger networks that are infeasible on the selected Virtex-7. Results of *dct* and *idct* benchmarks are nearly the same; we only present the results of *dct*. We display resource scaling and performance speedup of both minimum flit size (1-byte flit payload) and maximum flit size (full flit payload). The data size used in this experiment is equal to (biggest network size)×(data per-core compute). We observe that adding more cores to the network does not necessarily improve the performance (*mm* and *conv1d*) as external memory bandwidth may become saturated and be the bottleneck after a certain network size for memory intensive benchmarks. However, if the benchmark is computation intensive (*cp*, *dct*, and *idct*), we observe a proportional increase in performance as the network size is increased.

Next, we enable on-chip data sharing and use a directory size of 512 entries and present the scaling of performance and resource utilization for the benchmarks in Figs. 18–21. We preserve the same format, as shown in Figs. 14–17, for consistency and easier comparison. We observe that on-chip data sharing has significantly improved the performance scaling of *mm* benchmark by over 5.26× due to the potentially higher sharing opportunity. We also observe improved performance in *cp* and *dct*(*idct*) due to on-chip data sharing, despite not being memory bandwidth limited. Adding the directory-based on-chip data sharing has an impact on the LUT utilization: the average overhead in terms of LUT usage of adding directories is between 2.35% in small networks and 15.6% in large networks. Overall, the resource utilization curve and the performance curve have similar trends. This proves that additional resources for network improvement and data sharing improve the system performance. How-

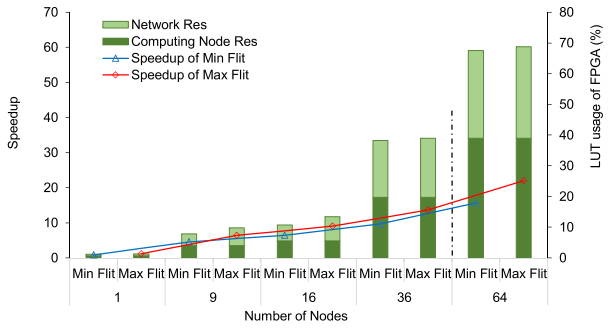


Fig. 18. Performance and resource scaling of *mm* with directory.

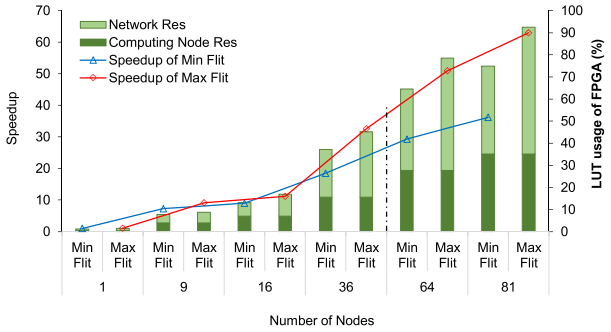


Fig. 19. Performance and resource scaling of *cp* with directory.

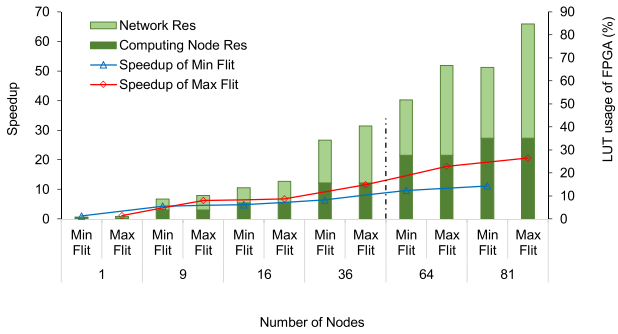


Fig. 20. Performance and resource scaling of *dct* with directory.

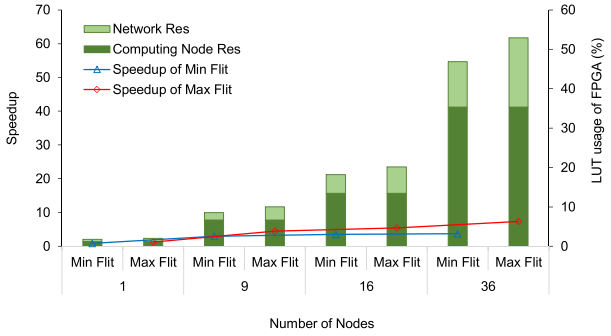


Fig. 21. Performance and resource scaling of *conv1d* with directory.

ever, *conv1d* has negligible performance improvement due to on-chip sharing. This could be due to the limited data reuse characteristic of the benchmark. To further understand the data reuse potential of the benchmarks, we evaluate the reduction in total number of memory reads for each of the benchmarks.

For evaluating the number of memory accesses, we select the maximum flit payload size of 8 byte and 512-entry directory size for the network in order to eliminate the impact of the flit width and directory size. For the baseline, we use a network

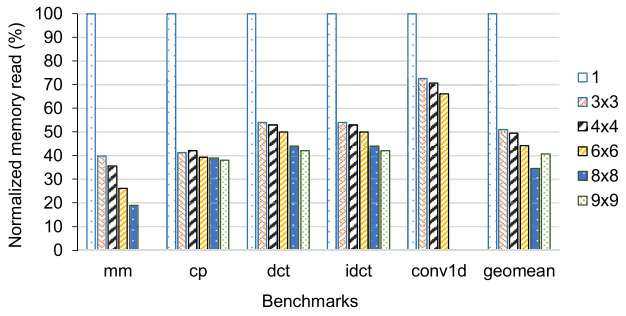


Fig. 22. Normalized off-chip memory read of different benchmarks with different network sizes.

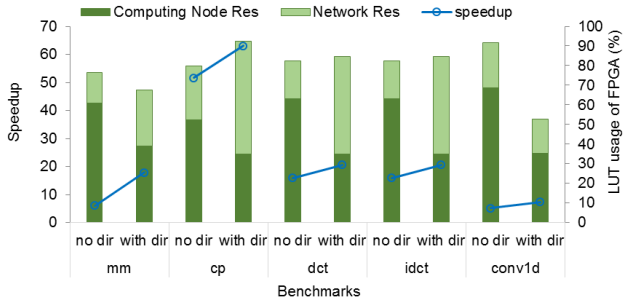


Fig. 23. Highest achievable speedup without and with data sharing.

with flit payload size of 8 byte and data sharing disabled. Thus, in the baseline case, the number of external memory reads is equal to the total number of memory reads by all cores as all requests have to access external memory. Fig. 22 shows the normalized external memory reads in the directory-enabled network compared with the baseline network. Since the on-chip data reuse is enabled by the directory system and the directories are distributed in every router, the total directory size increases when the network scales. This translates to increased on-chip data reuse before the reuse ratio reaches the upper limit of the application. *mm* has the highest opportunity for data reuse and has reduced the number of external memory reads by 81%. This reduced memory accesses translated to improved performance scaling in Fig. 18. On the other hand, *conv1d* has the least opportunity for data sharing and has reduced the memory reads only by 33.85%. Though the data reuse ratio for *conv1d* increases as the network scales, the overhead of on-chip communication also increases and limits any performance improvement. We confirmed from the collected run-time information that the average data access latency for *conv1d* is longer compared with other benchmarks.

To enable direct comparison, Fig. 23 shows the highest achievable speedups without data sharing and with data sharing for the benchmarks. It also shows the corresponding resource utilization. The highest achieved speedups with directory-enabled NoC are consistently higher than the speedup achieved without directories for all benchmarks. On average, the highest achievable speedup without directory is 15.8×, and it goes up to 21.6× with directory enhancement. In some benchmarks, such as *mm* and *conv1d*, we observe that the directories not only obtain higher speedup but also achieve them with fewer number of cores (hence, comparatively fewer resources) than the NoC with no directories. Overall, the results demonstrate that though the directory system and net-

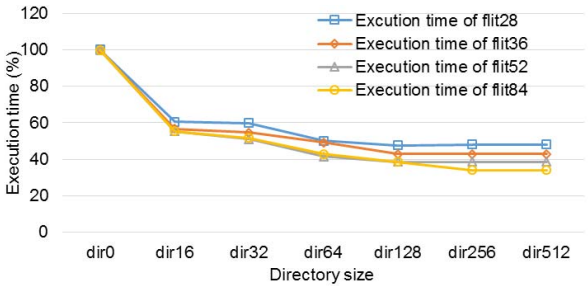


Fig. 24. System execution time of 36 cores *mm*.

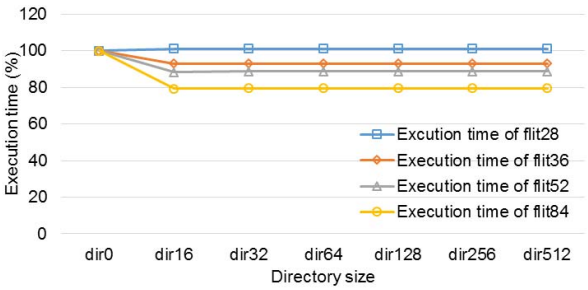


Fig. 25. System execution time of 36 cores *cp*.

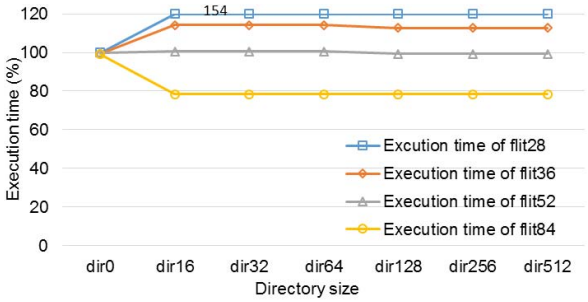


Fig. 26. System execution time of 36 cores *dct*.

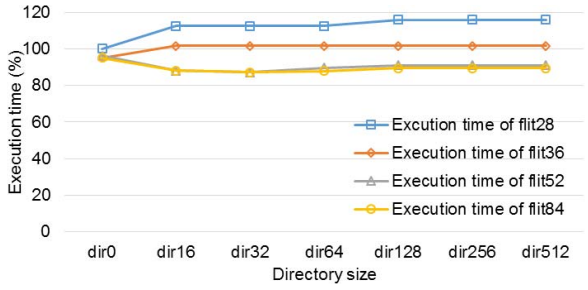


Fig. 27. System execution time of 36 cores *conv1d*.

work enhancements will consume additional resources, which can otherwise be used for instantiating more cores in the system, they significantly improve the system performance for applications with high data reuse opportunity. We now explore different network settings and study their impact on system performance.

C. Application-Specific Network Setting Exploration

We now study the impact of flit size and directory size on the system execution time. Network with 1-byte flit payload size and no data sharing is considered as the baseline, and the results of the other network settings are normalized to the baseline and shown in Figs. 24–27. The graphs show the impact of network settings on a 6×6 network for all the benchmarks.

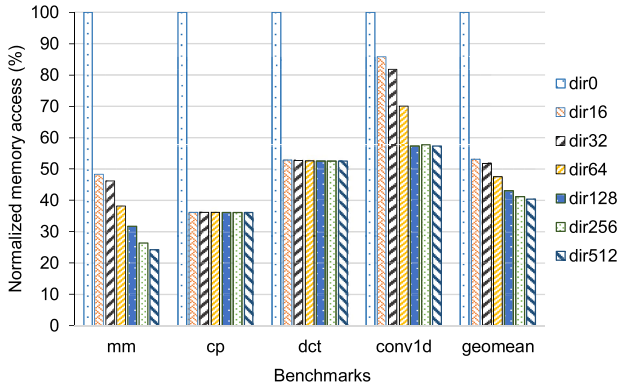


Fig. 28. Memory read of different benchmarks with 36 cores.

The size of directory varies from 0 to 512 in powers of 2, and *dir0* represents no data sharing. The directory is designed to enable on-chip data reuse, and increasing directory size improves on-chip data sharing before the sharing ratio reaches the upper limit of the particular benchmark. The difference in data reuse characteristics of the benchmarks also lead to different requirements of network settings, especially directory sizes. In general, higher flit sizes provide better performance at the cost of additional LUT resources. Smaller flits, though resource efficient, cause long on-chip communication latency. In the case of *mm*, there is high data-sharing opportunity, as each core has a potential of reusing as much as 83.3% of its data items from other cores. Although other benchmarks have sharing opportunity, the data in the above-mentioned figures demonstrate that the replacement of data items in on-chip storage together with the temporal locality of other cores' requests yields little additional benefit if we can track the 16 most recent data items. Other data items are either unlikely to be requested by other cores or unlikely to be on-chip by the time they are requested. As the directory system needs the whole data address to trace the location of the on-chip data, a smaller flit size requires more buffering time to get the whole address data, and leads to long execution time overhead.

Finally, we present the impact of directory size on the memory accesses in Fig. 28. Intuitively, higher directory sizes should increasingly reduce external memory accesses, since a large number of reads will be serviced by on-chip data. We observe this general trend in the results, though each benchmark attains the maximum memory reduction at a different directory size. Flit sizes have no effect on the external memory accesses and only impact the system performance by affecting on-chip latencies. The performance improvement of *mm* is strongly related to memory read reduction, while *cp*, *dct*, *idct*, and *conv1d* benefit from both memory reduction and increased flit size.

In summary, the on-chip data reuse feature in our FCUDA-NoC significantly improves system performance for memory intensive applications, such as *mm*. For applications that are computation intensive, we require data sharing and increased flit width to improve system performance. In the case, a tradeoff between resources usage and performance is desired, or for benchmarks that are not significantly affected by flit size, users can reduce the flit size accordingly. Using the configurable network router generator, we enable selection of a

customized NoC design based on the individual characteristic of the input application.

VI. Conclusion

In this paper, we presented an NoC architecture integrated with our prior FCUDA flow. We designed a configurable directory enhanced NoC router to enable on-chip data sharing. Our automated flow to generate the entire NoC system at RTL level with CUDA code input, and implement the architecture on a Xilinx VC709 evaluation board. Our FCUDA-NoC architecture is highly parameterized and allowed exploration of the NoC's design space. The design choices in our system enable creating an NoC that considers tradeoffs between router features and total network size to suit the application and meet performance requirements.

With our configurable FCUDA-NoC architecture, we demonstrated the capability of generating a complete NoC system with CUDA code input. Using the FCUDA-NoC architecture, we achieved a speedup of upto 63× (40.6× on average) and reduced external memory reads by upto 81% (56% on average) compared with a single hardware core implementation. In future work, we plan to develop new topology generation methods in order to generate different network topologies (e.g., torus versus mesh) for different benchmarks, as well as explore clustering and additional network hierarchy exploration.

References

[1] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A Java-compatible and synthesizable language for heterogeneous architectures," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2010, pp. 89–108.

[2] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "CλaSH: Structural descriptions of synchronous hardware using Haskell," in *Proc. 13th Euromicro Conf. Digit. Syst. Design, Archit., Methods, Tools (DSD)*, Sep. 2010, pp. 714–721.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *Proc. 3rd ACM SIGPLAN Int. Conf. Funct. Program. (ICFP)*, 1998, pp. 174–184.

[4] Calypto. *Catapult C Synthesis*. [Online]. Available: http://www.calypto.com/catapult_c_synthesis.php, accessed Nov. 20, 2015.

[5] A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2011, pp. 33–36.

[6] Xilinx. *Vivado High-Level Synthesis*. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, accessed Sep. 30, 2013.

[7] Synopsys. *Synphony C Compiler*. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx, accessed Aug. 20, 2014.

[8] H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow, "High-level synthesis with behavioral level multi-cycle path analysis," in *Proc. 23rd Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2013, pp. 1–8.

[9] T. S. Czajkowski *et al.*, "From OpenCL to high-performance hardware on FPGAS," in *Proc. 22nd Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2012, pp. 531–534.

[10] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: Low-power high-performance computing with reconfigurable devices," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, Aug./Sep. 2010, pp. 458–463.

[11] Altera. *Altera SDK for OpenCL*. [Online]. Available: http://www.altera.com/products/software/opencl/opencl-index.html, accessed Mar. 3, 2015.

[12] D. Greaves and S. Singh, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Proc. 16th Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2008, pp. 3–12.

[13] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[14] Cadence. *Cadence Introduces New High-Level Synthesis Technology*. [Online]. Available: http://www.cadence.com/cadence/newsroom/features/pages/feature.aspx?xml=ctosilicon, accessed Sep. 1, 2015.

[15] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Proc. IEEE 7th Symp. Appl. Specific Process. (SASP)*, Jul. 2009, pp. 35–42.

[16] A. Papakonstantinou *et al.*, "Multilevel granularity parallelism synthesis on FPGAs," in *Proc. IEEE 19th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2011, pp. 178–185.

[17] A. Papakonstantinou, D. Chen, W.-M. Hwu, J. Cong, and Y. Liang, "Throughput-oriented kernel porting onto FPGAs," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, May/Jun. 2013, pp. 1–10.

[18] S. T. Gurumani, H. Cholakkal, Y. Liang, K. Rupnow, and D. Chen, "High-level synthesis of multiple dependent CUDA kernels on FPGA," in *Proc. 18th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2013, pp. 305–312.

[19] M. S. Abdelfattah and V. Betz, "The power of communication: Energy-efficient NoCs for FPGAs," in *Proc. 23rd Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2013, pp. 1–8.

[20] L. Benini, "Application specific NoC design," in *Proc. Design, Autom., Test Eur. (DATE)*, vol. 1. Mar. 2006, pp. 1–5.

[21] D. Bertozzi and L. Benini, "Xpipes: A network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits Syst. Mag.*, vol. 4, no. 2, pp. 18–31, 2004.

[22] A. Ehliar and D. Liu, "An FPGA based open source network-on-chip architecture," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2007, pp. 800–803.

[23] R. Gindin, I. Cidon, and I. Keidar, "NoC-based FPGA: Architecture and routing," in *Proc. 1st Int. Symp. Netw.-Chip (NOCS)*, May 2007, pp. 253–264.

[24] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Des. Test Comput.*, vol. 22, no. 5, pp. 414–421, Sep./Oct. 2005.

[25] C. Hilton and B. Nelson, "PNoC: A flexible circuit-switched NoC for FPGA-based systems," *IEE Proc.-Comput. Digit. Techn.*, vol. 153, no. 3, pp. 181–188, May 2006.

[26] S. Kwon, S. Pasricha, and J. Cho, "POSEIDON: A framework for application-specific network-on-chip synthesis for heterogeneous chip multiprocessors," in *Proc. 12th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2011, pp. 1–7.

[27] S. Murali and G. De Micheli, "SUNMAP: A tool for automatic topology selection and generation for NoCs," in *Proc. 41st Design Autom. Conf.*, Jul. 2004, pp. 914–919.

[28] S. Vangal *et al.*, "An 80-tile 1.28TFLOPS network-on-chip in 65 nm CMOS," in *IEEE Int. Solid-State Circuits Conf. (ISSCC), Dig. Tech. Papers*, Feb. 2007, pp. 98–99 and 589.

[29] M. Shafique, L. Bauer, W. Ahmed, and J. Henkel, "Minority-game-based resource allocation for run-time reconfigurable multi-core processors," in *Proc. Design, Autom., Test Eur. Conf. Exhibit. (DATE)*, Mar. 2011, pp. 1–6.

[30] H. Bokhari, H. Javaid, M. Shafique, J. Henkel, and S. Parameswaran, "SuperNet: Multimode interconnect architecture for manycore chips," in *Proc. 52nd Annu. Design Autom. Conf. (DAC)*, 2015, Art. ID 85.

[31] H. Bokhari, H. Javaid, M. Shafique, J. Henkel, and S. Parameswaran, "Malleable NoC: Dark silicon inspired adaptable network-on-chip," in *Proc. Design, Autom., Test Eur. Conf. Exhibit. (DATE)*, Mar. 2015, pp. 1245–1248.

[32] H. Bokhari, H. Javaid, M. Shafique, J. Henkel, and S. Parameswaran, "darkNoC: Designing energy-efficient network-on-chip with multi-Vt cells for dark silicon," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2014, pp. 1–6.

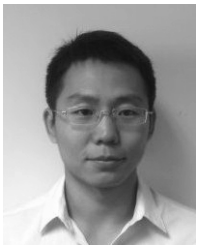
[33] D. U. Becker, "Efficient microarchitecture for network-on-chip routers," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, 2012.

[34] M. K. Papamichael and J. C. Hoe, "CONNECT: Re-examining conventional wisdom for designing nocs in the context of FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2012, pp. 37–46.

[35] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri, "LiPaR: A light-weight parallel router for FPGA-based networks-on-chip," in *Proc. 15th ACM Great Lakes Symp. VLSI (GLSVLSI)*, 2005, pp. 452–457.

[36] Y. Lu, J. McCanny, and S. Sezer, "Generic low-latency NoC router architecture for FPGA computing systems," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2011, pp. 82–89.

[37] S. Kumar *et al.*, "A network on chip architecture and design methodology," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2002, pp. 105–112.
[38] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-network coherence filtering: Snoopy coherence without broadcasts," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO-42)*, Dec. 2009, pp. 232–243.
[39] B. Zhang and B. Ravindran, "Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory," in *Principles of Distributed Systems* (Lecture Notes in Computer Science), vol. 5923, T. Abdelzaher, M. Raynal, and N. Santoro, Eds. Berlin, Germany: Springer-Verlag, 2009, pp. 48–53.
[40] N. Eisley, L.-S. Peh, and L. Shang, "In-network cache coherence," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO-39)*, Dec. 2006, pp. 321–332.
[41] L. Huang, Z. Wang, and N. Xiao, "An optimized multicore cache coherence design for exploiting communication locality," in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*, 2012, pp. 59–62.
[42] W. Zhang, L. Hou, J. Wang, S. Geng, and W. Wu, "Comparison research between XY and odd-even routing algorithm of a 2-dimension 3X3 mesh topology network-on-chip," in *Proc. WRI Global Congr. Intell. Syst. (GCIS)*, vol. 3. May 2009, pp. 329–333.
[43] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering route planning algorithms," in *Algorithmics of Large and Complex Networks*, J. Lerner, D. Wagner, and K. A. Zweig, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 117–139.
[44] *7 Series FPGAs Memory Interface Solutions; [User Guide]*, Xilinx, Inc., San Jose, CA, USA, 2011.
[45] A. H. Lam, "An analytical model of logic resource utilization for FPGA architecture development," M.S. thesis, Dept. Elect. Comput. Eng., Univ. British Columbia, Vancouver, BC, Canada, 2010.



Yao Chen received the B.Eng. degree in electronic science and technology from Nankai University, Tianjin, China, in 2010, where he is currently pursuing the Ph.D. degree in electronic science and technology.

His current research interests include network-on-chip design and high-level synthesis.

Swathi T. Gurumani (M'15) received the B.E. degree in electronics and communications from the University of Madras, Chennai, India, and the M.S. and Ph.D. degrees in computer engineering from the University of Alabama in Huntsville, Huntsville, AL, USA.

He is currently a Principal Research Engineer with the Advanced Digital Sciences Center, Singapore, and a Senior Research Affiliate with the Coordinated Sciences Laboratory, University of Illinois at Urbana–Champaign, Urbana, IL, USA. His current research interests include high-level synthesis, reconfigurable computing, and hardware/software co-design.

Yun Liang received the B.S. degree from Tongji University, Shanghai, China, in 2004, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

He was a Research Scientist with the Advanced Digital Science Center, University of Illinois at Urbana–Champaign, Urbana, IL, USA, from 2010 to 2012. He has been an Assistant Professor with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, since 2012. His current research interests include GPU architecture and optimization, heterogeneous computing, embedded system, and high-level synthesis.

Dr. Liang received the best paper award in the International Symposium on Field-Programmable Custom Computing Machines in 2011, and best paper award nominations in CODES+ISSS in 2008 and the Design Automation Conference in 2012. He serves as a Technical Committee Member of the Asia and South Pacific Design Automation Conference (ASPDAC), the Design, Automation & Test in Europe Conference, and the International Conference on Automation Science and Engineering. He was the TPC Subcommittee Chair of ASPDAC'13.

Guofeng Li received the B.S. degree in physics and the M.S. and Ph.D. degrees in electronic science from Nankai University, Tianjin, China, in 1982, 1985, and 2002, respectively.

He is currently a Professor with Nankai University, where he serves as the Director of the Center of Experimentation.

Prof. Li is an Executive Council Member of the China Physics Society.

Donghui Guo (M'97–SM'15) received the B.S. degree in radio physics and the M.S. and Ph.D. degrees in semiconductor from Xiamen University, Xiamen, China, in 1988, 1991, and 1994, respectively.

He joined Xiamen University as a Faculty Member in 1994, where he has been a Full Professor since 2002. He held a post-doctoral position with the City University of Hong Kong, Hong Kong. He was a Research Fellow, Senior Scientist, and Visiting Scholar with the University of Ulster, Londonderry, U.K., the Lawrence Berkeley Laboratory, University of California at Berkeley, Berkeley, CA, USA, and the University of Illinois at Urbana–Champaign, Urbana, IL, USA, respectively. He served as the Vice Dean of the School of Information Science and Technology with Xiamen University from 2008 to 2012. He is currently the Director of the IC Design & IT Research Center. His current research interests include artificial intelligence, network computing, IC design, nanodevice, and BioMEMS.

Kyle Rupnow (M'00) received the B.S. degree in computer engineering and mathematics and the M.S. and Ph.D. degrees in electrical engineering from the University of Wisconsin–Madison, Madison, WI, USA, in 2003, 2006, and 2010, respectively.

He is currently a Research Scientist with the Advanced Digital Sciences Center, University of Illinois at Urbana–Champaign, Urbana, IL, USA. His current research interests include high-level synthesis, reconfigurable computing, and systems management of compute resources.

Prof. Rupnow has served on the Program Committees of Field Programmable Gate Arrays, Field-Programmable Custom Computing Machines, Field-Programmable Technology, Field Programmable Logic and Applications, and ReConfig. He has also served as a Reviewer of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, *ACM Transactions on Design Automation of Electronic Systems*, the IEEE TRANSACTIONS ON VLSI SYSTEMS, and *ACM Transactions on Reconfigurable Technology and Systems*.

Deming Chen (S'01–M'05–SM'11) received the B.S. degree from the University of Pittsburgh, Pennsylvania, PA, USA, in 1995, and the M.S. and Ph.D. degrees from University of California at Los Angeles, Los Angeles, CA, USA, in 2001 and 2005, respectively, all in computer science.

He is currently a Professor with the ECE Department, University of Illinois at Urbana–Champaign, Urbana, IL, USA. His current research interests include system-level and high-level synthesis, nanosystems design and nanocentric CAD techniques, GPU and reconfigurable computing, hardware security, and computational genomics.

Dr. Chen is a Technical Committee Member for a series of conferences and symposia, including FPGA, ASPDAC, ICCD, ISQED, DAC, ICCAD, DATE, ISLPED, FPL, etc. He obtained the Achievement Award for Excellent Teamwork from Aplus Design Technologies in 2001, the Arnold O. Beckman Research Award from UIUC in 2007, the NSF CAREER Award in 2008, and five Best Paper Awards for ASPDAC'09, SASP'09, FCCM'11, SAAHPC'11, and CODES+ISSS'13. He is included in the List of Teachers Ranked as Excellent in 2008. He received the ACM SIGDA Outstanding New Faculty Award in 2010, and the IBM Faculty Award in 2014 and 2015. He also served as TPC Subcommittee or Track Chair, Session Chair, Panelist, Panel Organizer, or Moderator for some of these and other conferences. He is the General Chair for SLIP'12, the CANDE Workshop Chair in 2011, the Program Chair for PROFIT'12, and Program Chair for FPGA'15. He is or has been an Associated Editor for TCAD, TODAES, TVLSI, TCAS-I, JCSC, and JOLPE. He is the Donald Biggar Willett Faculty Scholar.