Generating Systolic Array Accelerators With Reusable Blocks

Liancheng Jia and Liqiang Lu
Peking University

Yun Liang
Peking University

Xuechao Wei
Alibaba, China

***Abstract*—Systolic array architecture is widely used in spatial hardware and well-suited for many tensor processing algorithms. Many systolic array architectures are implemented with high-level synthesis (HLS) design flow. However, existing HLS tools do not favor of modular and reusable design, which brings inefficiency for design iteration. In this article, we analyze the systolic array design space, and identify the common structures of different systolic dataflows. We build hardware module templates using Chisel infrastructure, which can be reused for different dataflows and computation algorithms. This remarkably improves the productivity for the development and optimization of systolic architecture. We further build a systolic array generator that transforms the tensor algorithm definition to a complete systolic hardware architecture. Experiments show that we can implement systolic array designs for different applications and dataflows with little engineering effort, and the performance throughput outperforms HLS designs.**

Tensor algebra is a prevalent tool of modern computer applications and is increasingly deployed onto various embedded devices. Such a trend demands specialized hardware accelerators. Systolic array architecture that features with high computation parallelism and data reusability using an array of processing elements (PEs) are widely adopted in accelerator designs. Google's TPU uses a systolic array for the matrix multiply unit.[1] Systolic architectures are also used in many other applications like convolution, FFT, and matrix decomposition.

Digital Object Identifier 10.1109/MM.2020.2997611
Date of publication 26 May 2020; date of current version 30 June 2020.

Published by the IEEE Computer Society

In general, the systolic array architecture is composed of a grid of PEs that run in parallel and buffers that transfer data between PEs and memory, and the PE is further composed of a compute cell and registers that store and transfer intermediate data. Each PE reads operands from its neighbors and process calculation in the compute cell. The result and operands are transferred to the neighbor PEs at the next time step. The systolic array architecture avoids nonlocal interconnection and reduces the bandwidth pressure because only the edge PEs connects to the memory, which makes it friendly for modern spatial architectures.

The systolic array architecture has three important features. First, the structure is highly modular, but the functionality of each module is complex. The same PE structure is replicated and connected regularly to build the systolic array. The complexity mainly comes from the controller for data transferring in PEs and memory buffers. Second, there is a large design space for systolic arrays. For the PE level, there are different dataflows, datatype, compute cell algorithm, data vectorization, etc. The PE-array level configuration includes array shape, buffer reuse strategy, and number of time steps, etc. Finally, the systolic architecture is reusable for different designs. The same computation logic can be reused for different systolic dataflows of the same algorithm, and the same systolic array structure can be reused with different computation cell logics to implement various algorithms.

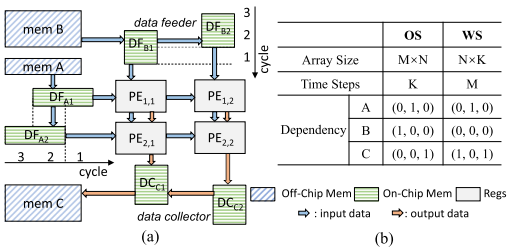
Prior works often use three approaches to implement systolic architecture. 1) Low-level HDL implementation.[2] The systolic array is manually implemented for a certain algorithm. This gives high performance, but the development is tedious and time-consuming. Meanwhile, this limits the design space that can be explored and cannot scale to other algorithms. 2) High-level synthesis (HLS).[3–5] HLS allows programmers to use software-programming language such as C for hardware development. It promotes productivity but the design is still difficult. The computation logic is tightly bound with the dataflow, making it hard to scale to different dataflows and algorithms. 3) There has been a growing interest in building a domain-specific language (DSL) as a front-end of HLS. The advantage of DSL over pure HLS is the separation of computation and dataflow definition in DSL programming, which was first proposed by an image processing DSL Halide.[6] Following works extends Halide to generate HLS code for hardware synthesis.[7,8] By using DSLs, the dataflow can be flexibly modified with little efforts without changing the algorithm description. However, the existing DSL compiler often generate unreadable HLS code which limits the optimization opportunity. The performance of DSL+HLS solutions are often slower than manual HDL design because of the lack of low-level optimizations.

In this article, we propose to generate systolic array designs which consider both productivity and performance. For performance consideration, we choose to use RTL level code to avoid HLS generation. For productivity, we build reusable hardware component templates which can be used by different systolic designs.

In this article, we propose to generate systolic array designs which consider both productivity and performance. For performance consideration, we choose to use RTL level code to avoid HLS generation. For productivity, we build reusable hardware component templates which can be used by different systolic designs. Our proposed generator can build different systolic arrays with the combination of different templates. Specifically, we use Chisel infrastructure[9] which enables cycle-level control as well as parameterized and modular hardware generation. We build configurable templates for the reusable components such as pipeline controllers, data feeders (DFs), and collectors. The generated components are connected together to form the complete design of systolic array. The complete systolic array can be generated with computation algorithm, dataflow definition, and the predefined templates.

The contribution of this article can be summarized with the following.

- We extract reusable modules of systolic arrays and build parameterized and modular hardware module template to express different functionality and configurations.



		OS	WS
Array Size		M×N	N×K
Time Steps		K	M
Dependency	A	(0, 1, 0)	(0, 1, 0)
	B	(1, 0, 0)	(0, 0, 0)
	C	(0, 0, 1)	(1, 0, 1)

Figure 1. Systolic array and PE structure for matrix multiplication. (a) Systolic array topology. (b) Structural parameters for GEMM.

- We build a systolic array generator to generate Chisel implementation of systolic arrays with the compute cell and algorithm-level configurations.
- We evaluate our systolic array generator using several tensor applications and compare with prior works to show its performance and productivity.

Experiment on Xilinx VU9P FPGA shows that our generated systolic architecture for GEMM achieves the frequency of 322 MHz on integer data and 264 MHz on floating-point, which is faster than most prior works on the same device. Our generator is able to generate complete systolic array architecture with less than 10% input codes compared to HDL, which remarkably promotes productivity.

SYSTOLIC ARRAY ARCHITECTURE

Our generator framework targets a general systolic architecture which covers most design alternatives for systolic arrays. In this section, we show the major components of a systolic array including four categories.

- The topology of systolic array architecture.
- The compute cell inside each PE that contains the core algorithm.
- The dataflow configurations, including PE size, data movement direction, number of time steps, and tensor index addressing.
- The controller that controls the behavior of systolic arrays including data validity and PE execution status.

We illustrate the building process of the systolic array with the example of a matrix multiplication algorithm, but it can be extended to other systolic algorithms.

Systolic Architecture Topology

Figure 1(a) shows a typical topology structure of systolic array. The major part is the 2-D grid of PEs that connects locally. The PEs on the edge connect to DFs and data collectors (DC), which contain SRAM buffers. DFs send data from memory to PEs and data collectors receive data from PEs. The first DF/DC modules connect to off-chip memory. Since the bandwidth between off-chip memory and the systolic accelerator is often limited, input data is reused multiple times in DFs. The reuse of data can be represented with loop tiling. The innermost loop-nest is directly mapped into the systolic array execution, and the outer loop-nest uses buffers in DF modules for data reusing.

PE Compute Cell Design

The core component of a systolic array is the compute cell inside each PE that performs the actual computation. The compute cell can be configured with an arithmetic function, a data type, and a data vectorization degree, which enables an SIMD processing of vectorized data. Additionally, the compute cell can use some external IPs provided by third-party, such as the floating-point IP for Xilinx FPGAs.

Systolic Dataflow

The systolic dataflow is the mapping between the computation loop nest and the systolic array execution. The execution inside the array can be represented as a dataflow tuple (x, y, t), which means the calculation at index (x, y) of the PE array at cycle t. The mapping process is a linear transformation and previous works have extensively studied the mapping algorithm.[3] We use an open-source systolic dataflow compiler[10] as the front-end of our systolic generator. It generates the dataflow tuple with the loop definition and linear transformation function. For the matrix multiplication, there exists two well-known dataflows: Output stationary (OS) and weight stationary (WS). The matrix multiplication of two $M \times N$ and $N \times K$ matrices can be defined as

$$C[m, n] += A[m, k] \times B[k, n] \quad (1)$$

where $0 < m < M, 0 < n < N, 0 < k < K$.

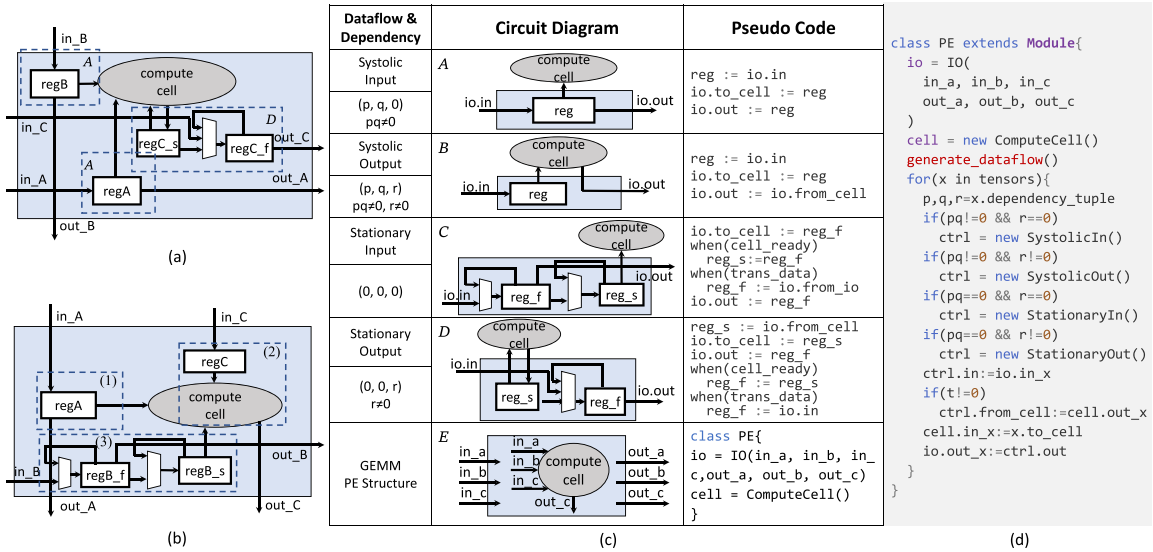


Figure 2. Implementation of systolic PE based on dataflow templates. (a) PE for OS dataflow. (b) PE for WS dataflow. (c) PE and controller templates. (d) Unified PE implementation.

From the mapping between loop indexes and dataflow tuple, the structural parameters such as the array size, the direction of data movement, the number of time steps, and tensor indexes that read, and write at each cycle can also be generated by the compiler. Figure 1(b) shows some of the structural parameters for both dataflows including array size, time steps, and data dependence. The data dependence is expressed with a tuple $d_x = (p, q, r)$ for each tensor element x. p, q stands for spatial dependence and r is temporal dependence. For OS dataflow, each C element depends on the result of the same PE at the previous cycle, so $d_C = (0, 0, 1)$. Element of matrix A receives data from PE on the left, and B receives from upper PE. They do not depend on the result so there's no temporal dependence. $d_A = (0, 1, 0)$ and $d_B = (1, 0, 0)$. For WS dataflow, matrix B stays in PE and matrix C flows vertically. Finally, the compiler also generates the tensor index that each PE executes at every cycle which affects the DF and collector's addressing.

Systolic Array Controller

The data movement scheme between systolic array PEs are different for each tensor elements, and they are controlled separately. For example, the *systolic* tensors transfers to adjacent PEs every cycle, but the *stationary* tensors stays in the PE during the calculation pipeline, but uses double buffer to transfer data simultaneously. There are complex data dependencies in the PE execution pipeline, and it requires a subtle cycle-level controller to perfectly overlap the pipelines in order to maximize the throughput. The controller should determine whether one data element should stay inside a particular PE, or should transfer to its neighbor PE. It also controls the execution status of each PE at every cycle. The controller requires careful design and optimization, and existing automatic-generated HLS code often fails to perform well.

GENERATING SYSTOLIC ARRAYS WITH REUSABLE COMPONENTS

The design space of systolic array involves a complex set of configurations. The configurations are divided into two categories indicating whether each of them can be parameterized. The *parameterize-able* configurations can be directly implemented into module class parameters, which is naturally supported by Chisel. There remain two important parts that are not parameterize-able: compute cell and PE controller. The compute cell algorithm must be manually implemented because it often relies on external IPs or customized algorithm implementation. The controller of different dataflows

might vary a lot, which brings great complexity for implementation and optimization.

Decoupled Generation of Controller and PE Structure

Figure 2(a) and (b) shows the PE diagram for OS and WS dataflow in GEMM computation. The I/O ports of two PEs and compute cells are identical, but they are connected with different controller modules, and the controller modules are related to each operand's data dependence. This enables the decoupled generation of pipeline controllers and other systolic array components since the pipeline controllers are independent with the application, and PE structure is independent with systolic dataflow.

The data dependence of tensors in the systolic array can be divided into four types based on whether the spatial part and temporal part of its dependence tuple is equal to zero. If the spatial part is zero, the tensor elements stays inside PE during each pipeline stage, otherwise it flows through PEs. If the temporal part is zero, the calculation does not generate partial results for the element to be used by next time step, and vice versa.

We implement four templates of PE pipeline controllers corresponding to their dependence type. The circuit diagram and pseudo code are presented in Figure 2(c). The controllers for systolic data movement (1) and (2) are straightforward because the tensor elements always transfer to its neighbor PEs every cycle. For stationary data (3) and (4), the controller becomes complex. One register is used to transfer data with the compute cell (reg_s), and the other is used to transfer data with adjacent PEs (reg_f). The two data transfers are concurrent. When the computation pipeline is executing, reg_s only transfer data with the compute cell and reg_f transfer data with other PEs at the same time. When the pipeline finishes, the two registers communicate to update their data for the next computation pipeline.

Since the dataflow compiler can automatically generate the dependence tuples, it requires no extra manual effort for choosing the pipeline controllers for each tensor. According to the dependence tuples in the "Systolic Dataflow" section, the PE of OS dataflow contains two modules (1) and one module (4). PE for WS dataflow contains one (1), one (2), and one (3).

The complete PE architecture can be built by connecting the controllers for each tensor with the PE I/O ports and the compute cell ports. Both compute cell and PE ports are identical for the same application regardless of dataflow. Figure 2(d) shows the pseudocode of the PE module structure of the generator. First, the generator instantiates a compute cell, and then it chooses the controller for each tensor according to the dataflow expressed in the dependence tuple. Finally, each inner modules are connected together to form the complete PE.

Reusable Design of Other Components

Similar as the modular design of controller inside each PE, we also build DF/DC module templates according to the four data dependence types. DFs use double buffer RAM for concurrent data transaction with PE and off-chip memory. We also define new parameters for the DF class to express different data reusing methodology for loop tiling. There's no data reuse in dc modules, and buffers are used for data serialization since the off-chip memory has low bandwidth.

We use functional programming feature provided by Chisel to implement the data addressing for both on-chip and off-chip memory in DF/DC modules. We define a function $f(x, t)$ which indicates that the address of $x\text{th}$ DF/DC module reads or writes at cycle t. The function f is generated with dataflow compiler, and is used as a parameter of DF/DC templates. In this way, our generator can support arbitrary tensor accessing scheme.

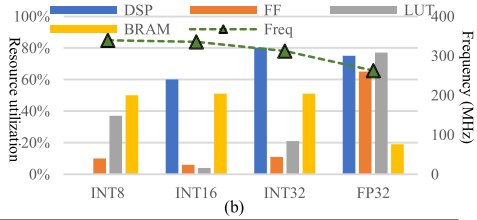
Complete Generation of Systolic Array

After the PE architecture and on-chip memory buffers are generated, we still need a controller at the top level to maintain the data synchronization of each operands. It connects the on-chip buffer with edge PEs, and determines whether the PE's input data is valid at each cycle. For example, WS dataflow requires one operand to stay inside all PEs before the other operand is pushed into PE array, and the controller maintains the dependence relationship.

Finally, we conclude the generation process of the systolic array into a complete compilation

	[4]	[6]	[3]	[7]	[5]	Ours	Ours	Ours
Platform	VU9P	ARRIA-10	ARRIA-10	VU9P	ARRIA-10	VU9P	VU9P	VU9P
Devlopment	HLS	DSL+HLS	HDL	DSL+HLS	HLS	Chisel	Chisel	Chisel
Data type	float32	float32	float32	float32	fixed16	fixed16	float32	float32
DataFlow	OS	OS	OS	-	OS	OS	WS	OS
Array Size	19x8	10x8	10x16	-	19x8	16x16	16x10	16x10
Vectorization	8	16	8	-	8	16	8	8
BRAM	89%	51%	-	43%	61%	51%	17%	19%
DSP	89%	89%	99%	36%	49%	60%	75%	75%
FF	39%	-	-	33%	-	6%	65%	65%
LUT	49%	-	-	38%	-	5%	75%	78%
MHz	228.8	215	312.5	250	231.85	322.5	263.1	264.6
Gop/s	555.4	549	800	245.9	1171.3	2642	673.3	677.3

(a)



(b)

	LOCs (Config)	LOCs (Cell)	LOCs (HDL)	Ours/HDL (%)	Perf. (Gop/s)
GEMM	19	24	1349	3.19	2642
Dynamic-Bits	19	237	2609	9.81	218
SW	22	32	518	10.42	-

(c)

Figure 3. Experiment results. (a) GEMM performance and resource utilization compared with prior works. (b) Performance and resource comparison of data types. (c) Evaluation of different applications.

flow. A user needs to specify the loop definition (with tiling) and compute cell definition. The generation flow first uses the dataflow compiler to generate dataflow-related configurations. The compiler generates different dataflows by using different transformation functions. Then, the compiler use the configurations to generate PE controllers, PE structure, DF/DC modules as well as the top-level controller. Connection wires are linked between them according to the data movement directions to generate the complete implementation of the systolic array. Chisel's toolchain compiles it into Verilog code for FPGA synthesis.

EVALUATION

Experimental Setup

We evaluate the performance and programming efficiency of our systolic generator with GEMM and other tensor applications, and compare the result of GEMM with several existing HLS-based works.[3,7,8] The systolic array designs are synthesized and implemented on Xilinx VU9P FPGA platform with Xilinx Vivado 2018.2. For floating-point multiplication, we use Xilinx's Floating-Point IP and integrate it into Chisel implementation as a *BlackBox* module.

FPGA Performance Comparison for GEMM

We compare the performance and resource utilization of our generated systolic array architecture with several state-of-the-art systolic array implementations in Figure 3. We use the problem size of $M = N = K = 256$. On Xilinx VU9P platform, our implementation with floating-point data type shows 677.3GOp/s which achieves 1.2× and 2.7× speedup compared with the work by Cong and Wang[3] and Lai *et al.*[7] The improvement comes from two aspects. First, our implementation uses Chisel's ready-valid interface for data communication, which avoids the unified HLS programming interface that leads to extra data dependence, and the complex finite state machine generated by HLS compiler. Second, we optimize frequency by checking the critical path and locating the Chisel code that caused the timing delay. We optimize our design by simplifying combinational logic, inserting registers, and removing unnecessary dependence between registers. Consequently, our systolic array design is highly optimized in RTL-level.

The manually optimized HDL solution[2] still has the best performance with 800GOp/s. High-performance HLS code requires manual loop transformations such as loop flattening, loop perfectization, etc. The optimizations are complex in HLS level, but straightforward in HDL level. Although the DSL+HLS solutions have high development productivity, but these frameworks lack the low level optimization and only show limited performance.

Performance Comparison of Different Dataflow and Data Types

We evaluate the performance of different dataflow and data types for GEMM.

Dataflow. We do not observe great performance variation between OS and WS dataflows, and their frequency are similar when other configurations are same. However, WS dataflow is not as flexible as OS because the reduction size is fixed to the size of PE array.

Data Type. We evaluate the DSP usage and frequency of four different data types: INT8, INT 16, INT32, and FP32. The result is shown in Figure 3(b). The 8-bit calculation has the best frequency, and it uses LUTs instead of DSPs for multiplication. Floating point has the lowest frequency because it consumes most LUT and FF resources which affects FPGA place-and-routing.

Evaluation of Generation Productivity

To evaluate the productivity of systolic array generation, we choose three systolic array applications and compare the line of codes (LOCs) of our proposed systolic generator and manual HDL design. Dynamic GEMM refers to GEMM with dynamic bit width,[11] and SW refers to Smith-Waterman algorithm.[12] Since SW uses a one-dimensional systolic array for only 2-level loop, we do not evaluate the performance here. Our proposed generator only requires 0.03× 0.1× LOCs compared with generated HDL, which proves the productivity of our proposed design.

CONCLUSION

We propose an high-performance and reusable systolic array generator. We extract reusable hardware components that appears in different dataflows and applications. We decouple the generation of complex pipeline controller with PE and compute cell structure, and build parameterized hardware templates for each of them. With the integration of systolic dataflow compiler, our generator uses the algorithm definitions and compute cells to instantiate the hardware templates and connect them together to build the complete systolic architecture. Experiments show that our proposed generator can achieve better performance compared with existing HLS-based solutions with 1.2× speedup, and efficiently simplify the generation process of systolic array architecture.

ACKNOWLEDGMENTS

This work was supported in part by the Beijing Natural Science Foundation (No. JQ19014, L172004) and in part by the Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES

1. N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
2. D. J. M. Moss *et al.*, "A customizable matrix multiplication framework for the intel harpv2 xeon +fpga platform: A deep learning case study," in *Proc. Int. Symp. Field-Program. Gate Array*, 2018, pp. 107–116.
3. J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *Proc. Int. Conf. Comput.-Aided Des.*, 2018, pp. 1–8.
4. X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. 54th ACM/EDAC/IEEE Des. Autom. Conf.*, 2017, pp. 1–6.
5. X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management," in *Proc. 56th Annu. Des. Autom. Conf.*, 2019, Art. no. 125.
6. J. Ragan-Kelley *et al.*, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 519–530.
7. Y. Lai *et al.*, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. Int. Symp. Field-Program. Gate Arrays*, 2019, pp. 242–251.
8. N. K. Srivastava *et al.*, "T2S-Tensor: Productively generating high-performance spatial hardware for dense tensor computations," in *Proc. 27th Ann. Int. Symp. Field-Program. Custom Comput. Mach.*, 2019, pp. 181–189.
9. J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proc. DAC Des. Autom. Conf.*, 2012, pp. 1212–1221.
10. H. Genc, "A DSL for systolic arrays," 2018. [Online]. Available: https://github.com/hngenc/systolic-array
11. H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 764–775.
12. C. W. Yu, K. Kwong, K.-H. Lee, and P. H. W. Leong, "A smith-waterman systolic cell," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2003, pp. 375–384.

Liancheng Jia is working toward the Ph.D. degree with the Center for Energy-Efficient Computing and Application, Peking University. His current research interests include high-performance computation in GPU and embedded systems. Jia received the B.S. degree in computer science from Peking University. Contact him at jlc@pku.edu.cn.

Yun(Eric) Liang is currently an Associate Professor (with tenure) with the School of Electrical Engineering and Computer Science, Peking University. His research focuses on heterogeneous computing (GPUs, FPGAs, ASICs) for emerging applications such as AI and big data, computer architecture, compilation techniques, programming model and program analysis, and embedded system design. He has authored more than 80 scientific publications in premier international journals and conferences in related domains. His research has been recognized by best paper award at FCCM 2011 and ICCAD 2017 and best paper nominations at PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, CODES+ISSS 2008. He serves as an Associate Editor for the *ACM Transactions in Embedded Computing Systems* and *Embedded System Letters*, and serves in the program committees in the premier conferences in the related domain including (HPCA, MICRO, DAC, ASPLOS, PACT, PPoPP, CGO, ICCAD, ICS, FPGA, FCCM). He is the corresponding author of this article. Contact him at ericlyun@pku.edu.cn.

Liqiang Lu is currently working toward the Ph.D. degree with the School of Electrical Engineering and Computer Science, Peking University. His research focuses on algorithm-level and architecture-level optimizations of FPGA for machine learning applications. Lu received the B.S. degree from the Institute of Microelectronics, Peking University, in 2017. Contact him at liqianglu@pku.edu.cn.

Xuechao Wei is currently a Software Engineer with Alibaba, Hangzhou, China. His research focuses on computer architecture, accelerator design, and synthesis. Wei received the Ph.D. degree from the Center for Energy-Efficient Computing and Applications, Peking University. Contact him at xuechao.wei@pku.edu.cn.