Customizable Computing — From Single-Chip to Datacenters

Jason Cong, *Fellow, IEEE,* Zhenman Fang, Muhuan Huang, Peng Wei, Di Wu, and Cody Hao Yu
Center for Domain-Specific Computing, University of California, Los Angeles

***Abstract*—Since its establishment in 2009, the Center for Domain-Specific Computing (CDSC) has focused on customizable computing. We believe that future computing systems will be customizable with extensive use of accelerators, as custom-designed accelerators often provide 10-100X performance/energy efficiency over the general-purpose processors. Such an accelerator-rich architecture presents a fundamental departure from the classical von Neumann architecture, which emphasizes efficient sharing of the executions of different instructions on a common pipeline, providing an elegant solution when the computing resource is scarce. In contrast, the accelerator-rich architecture features heterogeneity and customization for energy efficiency; this is better suited for energy-constrained designs where the silicon resource is abundant and spatial computing is favored—which has been the case with the end of Dennard scaling. Currently, customizable computing has garnered great interest; e.g. this is evident by Intel's $17B acquisition of Altera in 2015 and Amazon's introduction of FPGAs in its AWS public cloud.**

In this paper we present an overview of the research programs and accomplishments of CDSC on customizable computing, from single-chip to server node and to data centers, with extensive use of composable accelerators and field-programmable gate-arrays (FPGAs). We highlight our successes in several application domains, such as medical imaging, machine learning, and computational genomics. In addition to architecture innovations, an equally important research dimension enables automation for customized computing. This includes automated compilation for combining source-code-level transformation for high-level synthesis with efficient parameterized architecture template generations, and efficient runtime support for scheduling and transparent resource management for integration of FPGAs for datacenter-scale acceleration with support to the existing programming interfaces, such as MapReduce, Hadoop, and Spark, for large-scale distributed computation. We shall present the latest progress in these areas, and also discuss the challenges and opportunities ahead.

***Index Terms*—Customizable Computing, Specialized Acceleration, Accelerator-Rich Architecture, CPU-FPGA, FPGA Cloud.**

I. Introduction

SINCE the introduction of the microprocessor in 1971, the improvement of processor performance in its first thirty years was largely driven by the Dennard scaling of transistors [1]. This scaling calls for reduction of transistor dimensions by 30% every generation (roughly every two years) while keeping electric fields constant everywhere in the transistor to maintain reliability (which implies that the supply voltage needs to be reduced by 30% as well in each generation). Such scaling not only doubles the transistor density each generation and reduces the transistor delay by 30%, but also at the same time improves the power by 50% and energy by 65% [2]. The increased transistor count also leads to more architecture design innovations, such as better memory hierarchy designs and more sophisticated instruction scheduling and pipelining support. These combined factors led to over 1,000 times performance improvement of Intel processors in 20 years (from the 1.5um generation to the 65 nm generation), as shown in [2].

Unfortunately, Dennard scaling came to an end in the early 2000s. Although the transistor dimension continues to be reduced by 30% per generation according to Moore's law, the supply voltage scaling had to almost come to a halt due to the rapid increase of leakage power, which means that transistor density can continue to increase, but so can the power density. In order to continue meeting the ever-increasing computing needs, yet maintaining a constant power budget, simple processor frequency is no longer scalable and there is a need to exploit the parallelism in the applications to make use of the abundant number of transistors. As a result, the computing industry entered the era of parallelization in early 2000s, with tens to thousands of computing cores integrated in a single processor, and tens of thousands of computing servers connected in a warehouse-scale data center. However, the studies in the late 2000s showed that such highly parallel, general-purpose computing systems would soon again face serious challenges in terms of performance, power, heat dissipation, space, and cost [3], [4]. There is a lot of room to be gained by customized computing, where one can adapt the architecture to match the computing workload for much higher computing efficiency using various kinds of customized accelerators. This is especially important as we enter a new decade with a significant slowdown of Moore's Law scaling.

So, in 2008 we submitted a proposal entitled "Customizable Domain-Specific Computing" to the National Science Foundation (NSF), where we look beyond parallelization and focus on domain-specific customization as the next disruptive technology to bring orders-of-magnitude power-performance efficiency. We were fortunate that the proposal was funded by the Expeditions in Computing Program, one of the largest investments by the NSF Directorate for Computer and Information Science and Engineering (CISE), which led to the establishment of the Center for Domain-Specific Computing (CDSC) in 2009 [5]. This paper highlights a set of research results from CDSC in the past decade.

Our proposal was motivated by the large performance gap between a totally customized solution using an application-specific integrated circuit (ASIC) and a general-purpose processor shown in several studies. In particular, we quoted a 2003 case study of the 128-bit key AES encryption algorithm [6], where an ASIC implementation in a 0.18um CMOS technology achieved a 3.86Gbits/second processing rate at 350mW power consumption, while the same algorithm coded

in assembly languages yielded a 31Mbits/second processing rate with 240mW power running on a StrongARM processor, and a 648Mbits/second processing rate with 41.4W power running on a Pentium III processor. This implied a performance/energy efficiency (measured in Gbits/second/W) gap of a factor of 85X and 800X, respectively, when compared with the ASIC implementation.

The main source of energy inefficiency was due to the classical von Neumann architecture, which was an ingenious design proposed in 1940s when the availability of computing elements (electronic relays or vacuum tubes) was the limiting factor. It allows tens or even hundreds of instructions to be multiplexed and executed on a common datapath pipeline. However, this general-purpose, instruction-based architecture comes with a high overhead for instruction fetch, decode, rename, schedule, etc. In [7] it was shown that for a typical superscalar out-of-order pipeline, the actual compute units and memory account for only 36% of the energy consumption, while the majority of the energy consumption (i.e., the remaining 64%) is for supporting the flexible instruction-oriented general-purpose architecture. After more than five decades of Moore's Law scaling, however, we now can integrate tens of billions of transistors on a single chip. The design constraint has shifted from compute resource limited to power/energy-limited. Therefore, the research at CDSC focuses on extensive use of customizable accelerators, including fine-grain field-programmable gate arrays (FPGAs), coarse-grain reconfigurable arrays (CGRAs), or dynamically composable accelerator building blocks at multiple levels of computing hierarchy for greater energy efficiency. In many ways, such accelerator-rich architectures are similar to a human brain, which has many specialized neural microcircuits (accelerators), each dedicated to a different function (such as navigation, speech, vision, etc.). The computation is carried out spatially instead of being multiplexed temporally on a common processing engine. Such high degree of customization and spatial data processing in the human brain leads to a great deal of efficiency—the brain can perform various highly sophisticated cognitive functions while consuming only about 20W, an inspiring and challenging performance for computer architects to match.

Since the establishment of CDSC in 2009, the theme of customization and specialization also received increasing attention from both the research community and the industry. For example, Baidu and Microsoft introduced FPGAs in their data centers in 2014 [8], [9]. Intel acquired Altera, the second-largest FPGA company in 2015 in order to provide integrated CPU+FPGA solutions for both cloud computing and edge computing [10]. Amazon introduced FPGAs in its AWS computing cloud in 2016 [11]. This trend was quickly followed by other cloud providers, such as Alibaba [12] and Huawei [13]. It is not possible to cover all the latest developments in customizable computing in a single paper. This paper chooses to highlight the significant contributions in the decade-long effort from CDSC. We also make an effort to point out the most relevant work. But it is not the intent of this paper to provide a comprehensive survey of the field, and we regret for the possible omissions of some related results.

The remainder of this paper is organized as follows. Section II discusses different levels of customization, including the chip level, server-node level, and datacenter level, and presents the challenges and opportunities at each level. Sections III presents our research on compilation tools to supporting the easy programming for customizable computing. Sections IV presents our runtime management tools to deploy such accelerators in servers and datacenters. We conclude the paper with future research opportunities in Section V.

II. Levels of Customization

Our research suggests that customization can be enabled at different levels of computing hierarchy, including chip-level, server-node level, and datacenter-level. This section discuss the customization potential at each, and the associated arhitecture design problems, such as 1) How flexible should the accelerator design be, from fixed-function accelerator design to composable accelerators to programmable fabric; 2) How to design the corresponding on-chip memory hierarchy and network-on-chip efficiently for such accelerator-rich architectures? 3) How to efficiently integrate the accelerators with the processor? We leave the compilation and runtime support to the next section.

A. Chip-Level Customization

1) Overview of accelerator-rich architectures: Since the establishment of CDSC, we have explored various design options for the chip-level customizable accelerator-rich architectures (ARAs). In such ARAs, a sea of heterogeneous accelerators are customized and integrated into the processor chips, in companion with a customized memory hierarchy and network-on-chip, to provide orders-of-magnitude performance and energy gains over conventional general-purpose processors. Figure 1 presents an overview of our ARA research scope including customization for compute resources, on-chip memory hierarchy, and network-on-chip. An open source simulator called PARADE [14] is developed to perform such architectural studies. In companion with the PARADE simulator, a wide range of applications, including those in medical imaging, computer vision and navigation, and commercial benchmarks from PARSEC, are used to evaluate the designs [14].

Customizable compute resources. As shown in Figure 1, our first ARA design (ARC [15], [16]) features dedicated accelerators designed for a specific application domain. ARC features a global hardware accelerator manager to support sharing and arbitration of multiple cores for a common set of accelerators. It uses a hardware-based arbitration mechanism to provide feedback to cores to indicate the wait time before a particular accelerator resource becomes available and lightweight interrupts to reduce the OS overhead. Simulation results show that, with a set of accelerators generated by a high-level synthesis tool [17], it can achieve an average of 18x speedup and 24x energy gain over an Ultra-SPARC CPU core for a wide range of applications in medical imaging, computer vision and navigation, as well as commercial benchmarks from PARSEC [18]. From this study, we also noticed that many accelerators in a given domain can be decomposed into a

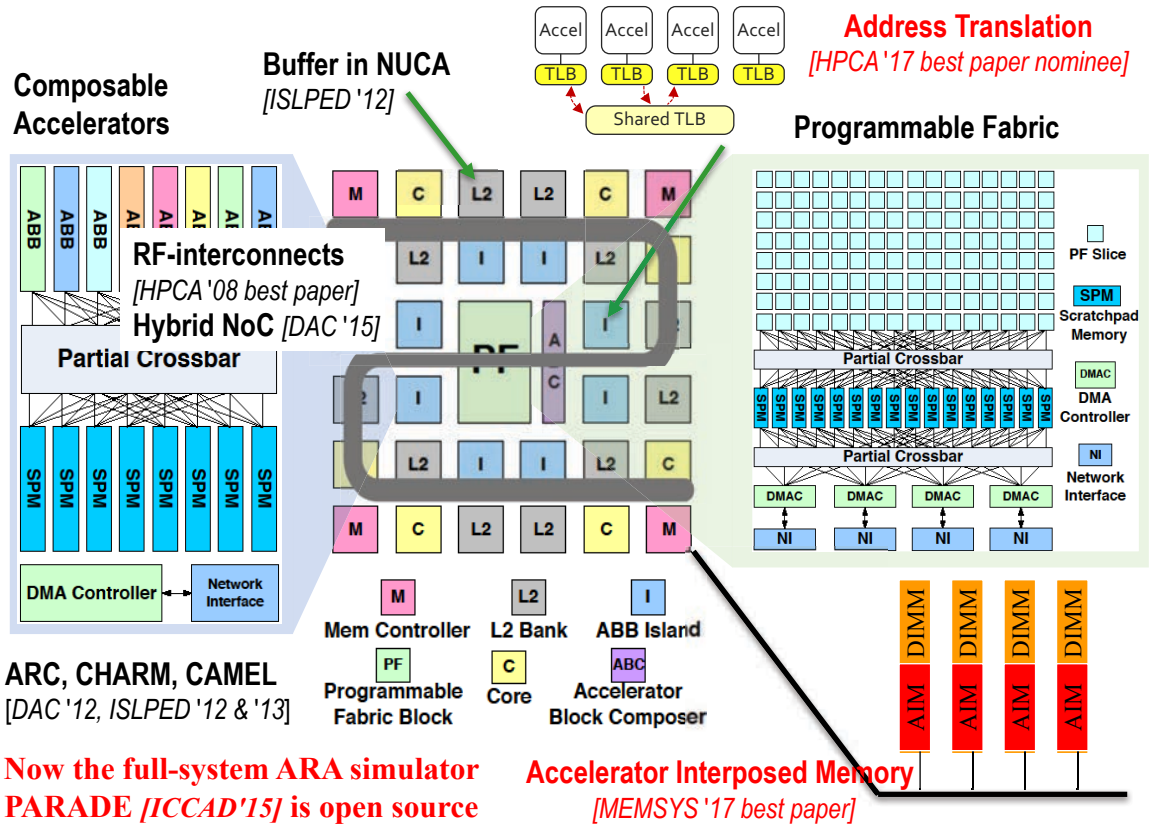


Fig. 1: An overview of Accelerator-Rich Architectures (ARAs).

set of primitive computations, such as low-order polynomials, square-root, and inverse computations. So, our second-generation ARA (CHARM [7], [19]) uses a set of accelerator building blocks (ABBs), which are grouped into ABB islands, to compose accelerators based on current system demand. The composition of accelerators are statically determined at the compilation time, but dynamically allocated from a resource pool at runtime by an on-chip accelerator building block composer (ABC), leading to a much more efficient resource utilization. With respect to the same set of medical imaging benchmarks, the experimental results on CHARM demonstrate over 2x better performance than ARC with similar energy-efficiency for medical imaging applications. In order to support new workloads which were not considered in the ABB designs, our third-generation ARA (CAMEL [20]) uses a programmable fabric to provide even more adaptability and longevity to the design. Accelerator building blocks could be synthesized in the programmable fabric to match varying demand, from new emerging domains or algorithmic evolution in the existing application domains.

In our ARA work, all accelerators are loosely coupled with CPU cores in a sense that they do not belong to any single core, but can be shared by all the cores via network-on-chip. In fact, they share L2 cache with the CPU cores (more discussion about the memory customization in the next subsection). Alternative approaches from other research groups explored the use of tightly coupled accelerators by extending a processor core with customized instructions or functional units for lower latency [21], [22]. In terms of the granularity of the customized accelerators, commercial field programmable fabrics (FPGAs) provide ultra-fine-grained reconfigurability that sacrifices some efficiency and performance for generality, while coarse-grained reconfigurable arrays (CGRAs) [23]–[25] provide composable accelerators with near-ASIC performance and FPGA-like configurability. We expect future chips to have more computing heterogeneity with different trade-off between programmability and efficiency, including various CPU cores, dedicated accelerators, composable accelerators, fine-grain and coarse-grain programmable fabrics, as well as SIMD cores in a single chip to satisfy the computing demands of the ever-changing applications.

Customizable on-chip memory hierarchy. In an accelerator-based architecture, buffers are commonly used as near memories for accelerators. An accelerator needs to fetch multiple input data elements simultaneously with predictable or even fixed latency to maximize its performance. To achieve this goal, we engaged in a series of studies to customize the on-chip memory hierarchy to investigate both the dedicated buffers for accelerators [26]–[28], and hybrid and adaptive cache and buffer designs shared between CPU cores and accelerators [29]–[31], as shown in Figure 1.

For the dedicated buffers for accelerators, we often have to partition a buffer into multiple on-chip memory banks to maximize on-chip memory bandwidth. We developed the general theory and algorithms for cyclic memory partitioning to remove memory access conflict at each clock cycle to enable efficient pipelining [26], [27]. For stencil applications, we also develop an optimal non-uniform memory partitioning scheme that is guaranteed to simultaneously minimize the on-chip buffer size and off-chip memory access [28]. These results

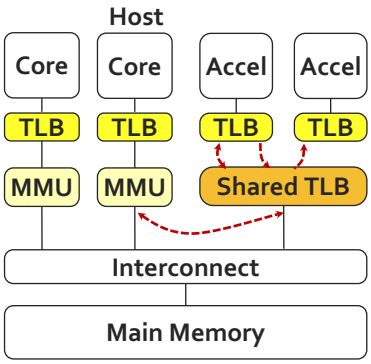


Fig. 2: Address translation support for ARA.

can be used for both ASIC and FPGA accelerator designs.

One representative example of adaptive cache shared between CPU cores and on-chip accelerators is our Buffer-in-NUCA (BiN) work [31], which dynamically allocates buffers of competing cores and accelerators in a non-uniform cache architecture (NUCA). BiN features: 1) a global buffer manager responsible for buffer allocation to all accelerators on-chip; 2) a dynamic interval-based global allocation method to assign spaces in NUCA caches to accelerators that can best utilize the additional buffer space, and 3) a flexible paged allocation method to minimize accelerator-to-buffer distance and limit the impact of buffer fragmentation, with only a small local page table at each accelerator. Compared to the alternative approaches using the accelerator store (AccStore) scheme [32] and the Buffer-integrated-Cache (BiC) scheme [33] for sharing buffers and/or caches among accelerators, BiN improves performance by 32% and 35% and reduces energy by 12% and 29% for medical imaging applications.

To improve the ARA programmability and avoid unnecessary memory copy between CPU cores and accelerators, a unified memory space between them is essential. In order to support such unified memory space, we have to provide efficient address translation support in ARAs. We characterize the memory access behavior of customized accelerators to drive the TLB augmentation and MMU designs, as shown in Figure 2. First, to support bulk transfers of consecutive data between the scratchpad memory of customized accelerators and the memory system, we present a relatively small private TLB design (with 32 entries per accelerator instance) to provide low-latency caching of translations to each accelerator. Second, to compensate for the effects of the widely used *data tiling* techniques, we design a level-two TLB (with 512 entries) to be shared by all accelerators to reduce private TLB misses on common virtual pages, eliminating duplicate page walks from accelerators working on neighboring data tiles that are mapped to the same physical page. This two-level TLB design effectively reduces page walks by 75.8% on average. Finally, instead of implementing a dedicated MMU which introduces additional hardware complexity, we propose simply leveraging the host per-core MMU for efficient page walk handling. This mechanism is based on our insight that the existing MMU cache and data cache in the host core side satisfies the demand of customized accelerators with minimal overhead. Using applications in the four domains mentioned at the beginning of Section II-A, our evaluation demonstrates

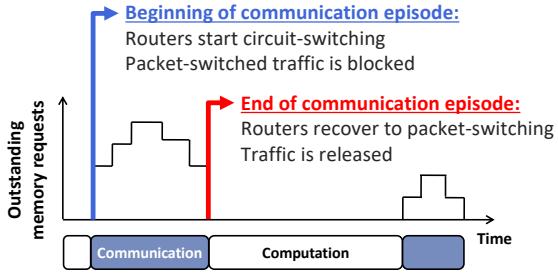


Fig. 3: Hybrid network-on-chip with predictive reservation.

that the combined approach achieves 7.6x average speedup over the naive IOMMU approach, and is only 6.4% away from the performance of the ideal address translation [34].

Customizable network-on-chip (NoC). The throughput of an accelerator is often bound by the rate at which the accelerator is able to interact with the memory system. As shown in Figure 1, on one hand, we explored the use of radio frequency interconnect (or RF-I) over on-chip waveguided transmission lines [35] as on-chip interconnect to provide high aggregate bandwidth, low latency via signal propagation at the speed of light, and customizable point-to-point communications (through frequency multiplexing). On the other hand, we developed a hybrid network-on-chip based on the conventional on-chip interconnect technology but with a hybrid circuit switching and packet switching to improve the performance. In particular, it uses predictive reservation (HPR) [36], shown in Figure 3, based on the observation that accelerator memory accesses usually exhibit predictable patterns, creating highly utilized network paths. By introducing circuit-switching to cover accelerator memory accesses, HPR reduces per-hop delays for accelerator traffic. Unlike previous circuit-switching proposals, HPR eliminates circuit-switching setup and tear-down latency by reserving circuit-switched paths when accelerators are invoked. We further maximize the benefit of path reservation by regularizing the communication traffic through TLB buffering and hybrid-switching. The combined effect of these optimizations reduces the total execution time by 11.3% over a packet-switched mesh NoC. A more detailed survey of most of these techniques covered in this subsection can be found in [37].

2) Simulation environment and in-depth analysis: To better evaluate ARA designs, we developed an open-source simulation infrastructure called PARADE: the Platform for Accelerator-Rich Architectural Design and Exploration. In addition, we performed in-depth analysis to provide insights into how ARAs can achieve the large performance and energy gains.

PARADE simulation infrastructure [14]. As shown in Figure 1, the PARADE infrastructure models each accelerator quickly by leveraging high-level synthesis (HLS) tools, so that users can easily describe the accelerators in high-level C/C++ languages. We provide a flow to automatically generate either dedicated or composable accelerator simulation modules that can be directly plugged into PARADE through the customizable NoC. We also provide a cycle-accurate model of the hardware global accelerator manager that efficiently manages accelerator resources in the accelerator-rich design. PARADE is integrated with the widely used cycle-accurate

full-system simulator gem5 [38], which models the CPU cores and the cache memory hierarchy. By extending gem5, PARADE also provides a cycle-accurate model of the coherent cache/scratchpad with shared memory between accelerators and CPU cores, as well as a customizable network-on-chip. In addition to performance simulation, PARADE also models the power, energy and area using existing toolchains including McPAT [39] for the CPU, and HLS and RTL tools for accelerators. A wide range of applications with pre-built accelerators, including those in medical imaging, computer vision and navigation, and commercial benchmarks from PARSEC, are also released with PARADE.

Performance analysis. To gain deep insights into the big performance gains, we conduct an in-depth analysis of ARAs and observe that ARAs achieve performance gains from both computation and memory access customization: ARAs (a single fixed-function accelerator instance in ARC) get 15x speedup over CPUs (a single X86 CPU core) in the computation, and 25x speedup in the memory access. For computation customization, ARAs exploit both fine-grained and coarse-grained parallelism to generate a customized processing pipeline without instruction execution overhead. For memory access customization, ARAs exploit a tile-based three-stage read-compute-write execution model that reduces the number of memory accesses and improves the memory-level parallelism (MLP). We quantitatively evaluate the performance impact of both factors and surprisingly find that the dominating contributor to the ARA memory access performance improvement is the improved MLP rather than the widely-expected memory access reduction. In fact, we find that existing GPU accelerators also benefit from the improved MLP through using different techniques. The totally customized processing pipeline of ARAs further provide an average of 1.4x speedup over GPUs. On overage, ARAs are 18x more energy efficient than GPUs, at the same technology node and the same number of GPU stream multiprocessors and ARA accelerator instances.

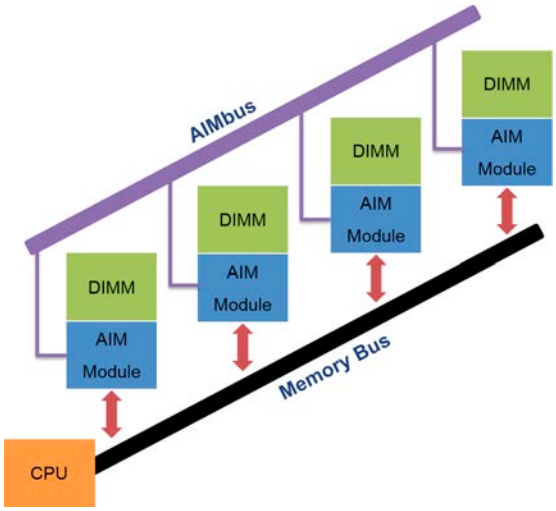


Fig. 4: An overview of the AIM architecture.

3) Near data acceleration: As we improve the computing efficiency with the extensive use of accelerators, memory bandwidth is becoming an increasing limitation. To address this issue, our recent accelerator-interposed memory (AIM) work [40] proposes to move the accelerators close to the memory system, as shown in Figure 4. To avoid the high memory access latency and bandwidth limitation of CPU-side acceleration, we design accelerators as a separate package, called an AIM module, and physically place an AIM module between each DRAM DIMM module and conventional memory bus network. Such an AIM module consists of an FPGA chip to provide flexible accelerator designs and two connectors, one to the memory bus and the other to each DIMM. A set of AIM modules can be introduced to an off-the-shelf computer with minimal modification to the existing software (to enable accelerator offloading). The overall memory capacity and bandwidth scales well with the increasing number of DIMMs in the system. Experimental results for genomics applications show that AIM achieves up to 3.7x better performance than the CPU-side acceleration, When there are 16 instances of accelerators and DIMMs in the system. The AIM approach is a viable alternative to 3D stacked memory [41], [42] and could be more economical. We believe one of the future trends is to move accelerators closer to the data, where they can have more data access bandwidth, as well as lower data access latency.

B. Sever-Node Level Customization

Due to the high cost and long design cycle of ASIC implementations, we did not implement he accelerator-rich architectures (ARA) in a single silicon chip. Instead, we use the server-node level integration of CPU+FPGA to support customizable computing by implementing various accelerators on FPGAs[1] With such node-level customization, we are able to achieve many interesting, often impressive acceleration results.

1) A Case Study of FPGA Accelerator Integration: This section presents the result we achieved for accelerating the CS-BWAMEM [43] algorithm, a Spark-based [44] parallel implementation for the widely used BWA-MEM DNA sequencing algorithm [45], to illustrate the opportunities and challenges for acceleration using CPU+FPGA based configuration. Specifically, we highlight the acceleration of one key computation kernel of this program, the Smith-Waterman (S-W) algorithm [46], and present the challenge and solution for accelerator integration in the overall Spark-based application.

S-W FPGA accelerator. We first describe our FPGA accelerator design for the S-W algorithm in the CS-BWAMEM software. The S-W algorithm is based on two-dimensional dynamic programming algorithm with quadratic time complexity, and is one of the major computation kernels of most DNA sequencing applications. It is widely used for aligning two strings with a predefined scoring system to achieve the best alignment score, and many prior studies have proposed a variety of hardware accelerators for the algorithm. These accelerators basically share the common scheme of exploring the "anti-diagonal" parallelism in the S-W algorithm, and achieve good performance improvement for single S-W

[1]Note accelerators in this subsection are different to ASIC accelerators simulated in Section II-A, here we are using FPGAs to accelerator certain software functions.. Such systems are more cost efficient (based on off-the-shelf components) and more scalable (e.g. one may attach multiple FPGAs to a processor or upgrade the processor independent of the FPGAs).

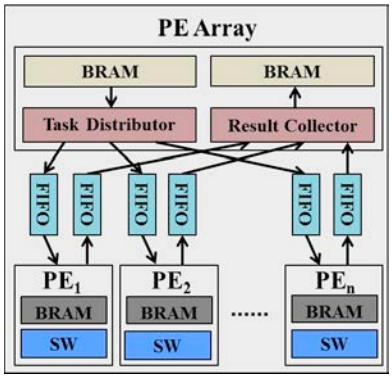


Fig. 5: PE-Array based Smith-Waterman accelerator design.

task [46]. However, this methodology does not work well for the S-W implementation in the CS-BWAMEM application due to the following reasons. First, the inner-task parallelism is actually broken because the CS-BWAMEM software applies extensive pruning. The pruning strategy results in 2x speedup, but excludes the "anti-diagonal" parallelism. Moreover, the efficiency of prior accelerators relies on the regularity of the S-W input. CS-BWAMEM features a large number of S-W tasks with highly variant input sizes (due to the unpredictable outcome of initial exact seed matching step for each read), which does not fit well for these accelerators.

Nevertheless, the DNA sequencing application feature a large degree of task-level parallelism, i.e., one has to align billions of short reads, which implies billions of independent S-W tasks. Given these observations, in [47] we propose a S-W accelerator design with a completely different methodology, as shown in Fig. 5. The proposed design features an array of processing elements (PEs) to process multiple S-W tasks in parallel. Each PE processes a S-W task in a sequential way instead of exploring the "anti-diagonal" parallelism. This leads to a long processing latency of each S-W task, but a simplified PE design with very small resource consumption. As a result, the PE can be duplicated over 100 duplicates and the task-level parallelism is well explored. Moreover, this PE design is compatible with the pruning strategies, and is not affected by the irregularity of the S-W input size. The "task distributor" in Fig. 5 feeds that each PE with sufficient tasks and the "result collector" assures the eventual in-order completion. As a result, the proposed design demonstrates 24x speedup over 12-core CPUs and 6x speedup over prior accelerator designs exploring the "anti-diagonal" parallelism [47].

Challenges of hardware accelerator integration. Despite the substantial speedup by FPGA acceleration, the integration of FPGA accelerators into big-data computing frameworks like Spark is not easy. First, the CPU-FPGA communication overhead offsets the performance improvement of the FPGA acceleration. In particular, if the payload of each transaction is fairly small (if one sends one short-read a time for alignment), the communication overhead could easily become the dominant performance factor. Another challenge is to efficiently share the FPGA resource among multiple CPU threads. To address these two challenges, we developed the batch processing and Accelerator-as-a-Service (AaaS) approaches [48].

Batch processing. Apache Spark programs are mainly written in Java and/or Scala, and run on Java virtual machines (JVMs) that do not support the use of FPGAs by default. While the Java native interface (JNI) serves as a standard tool to address this issue, it does not always deliver an efficient solution. In fact, if we invoke the FPGA accelerator routine in a straightforward way once per S-W function, the system performance will become over 1000x slower. The main reason for this performance degradation is the tremendous JVM-FPGA communication overhead aggregated through all the invocations of the S-W accelerator. To be specific, in our system, each S-W invocation of the software version on the CPU costs no more than 20μs on average. Meanwhile, a complete routine of a S-W accelerator invocation involves: 1) data copy between a JVM and a native machine, 2) DMA transfer between a native machine and an FPGA board though PCIe, and 3) computation on the FPGA board. The communication process alone, including 1) and 2), costs over 25ms per invocation. That is, even if an accelerator could reduce the computation time of the S-W kernel down to 0, the communication overhead easily erase any performance gain.

To amortize the communication overhead, we batch a group of reads together and offload them to the FPGA board to improve the bandwidth utilization. In fact, any Spark-based MapReduce program offers a large degree of parallelism in the map stage. It is feasible and highly effective to conduct batch processing for CS-BWAMEM. Specifically, we merge a certain number of CS-BWAMEM's map tasks into a new map function, and conduct a series of code transformations to batch the S-W kernel invocations from different map tasks together. This approach substantially improves the system performance and turns the 1000x slowdown back to 4x speedup compared to the single-thread software implementation.

Accelerator-as-a-service (AaaS). Due to the high performance of FPGA accelerators, offloading a single-thread CPU workload onto the FPGA usually makes the FPGA underutilized, which leaves opportunities for FPGA accelerators to be shared by multiple threads in a single node. The major challenge is how to efficiently manage the FPGA accelerator resources among multiple CPU threads. To tackle this challenge, we propose an Accelerator-as-a-service (AaaS) [2] framework and implement the FPGA management in a node-level accelerator manager.

The AaaS framework abstracts the FPGA accelerator and its management software on the CPU (called accelerator manager (AM)) as a *server*, and treats each CPU thread as a *client*. Client threads communicate with AM via a hybrid of JNI and network sockets. Different client threads send requests independently to the AM to accelerate S-W batches, and the AM processes the requests in a first-come-first-serve way. The AaaS framework enables sharing of the FPGA resource among many CPU threads, and retains 3x speedup over the multi-thread software.

In fact, this example motivated us to develop a more general runtime system to support the accelerator integration

[2]As explained in this section, the AaaS concept we propose is different to the one Amazon AWS uses.

TABLE I: Classification of modern high-performance CPU-FPGA platforms

	Private Memory	Shared Memory
Peripheral Interconnect (e.g., PCIe)	Alpha Data [49], Microsoft Catapult [8]	IBM CAPI [51], Intel HARPv2 [55]
Processor Interconnect (e.g., QPI)	N/A	Convey HC-1 [53], Intel HARPv1 [54] Intel HARPv2 [55]

for all Apache Spark programs, which will be presented in Section IV.

2) Characterization of CPU-FPGA Platforms: The performance and energy efficiency offered by FPGA accelerators encouraged the development of a variety of CPU-FPGA platforms. The choice of the best platform may vary depending on the application workloads. So, we carried out a systematic study to characterize the existing CPU+FPGA platforms and present guidelines for platform choice for acceleration.

We classify the existing CPU+FPGA platforms in Table I according to their physical integration mechanisms and the memory models. The most widely used integration scheme is to connect an FPGA to a CPU via the PCIe bus, with both components using (separate) private memories. Many FPGA boards built on top of Xilinx or Intel FPGAs use this way of integration because of its extensibility. One example is the Alpha Data FPGA board [49] with the Xilinx FPGA fabric that can leverage the Xilinx SDAccel development environment [50] to support efficient accelerator design using high-level programming languages, including C/C++ and OpenCL. This platform was used in the preceding section for CS-BWAMEM acceleration. Nevertheless, vendors like IBM also support a PCIe connection with a coherent, shared memory model for easier programming. For example, IBM has been developing the Coherent Accelerator Processor Interface (CAPI) on POWER8 [51] for such an integration, and has used this platform in the IBM data engine for NoSQL acceleration [52]. More recently, closer CPU-FPGA integration becomes available using a new class of processor-to-processor interconnects such as front-side bus (FSB) and the newer QuickPath Interconnect (QPI). These platforms tend to provide a coherent shared memory, such as the FSB-based Convey system [53] and the Intel HARP family [54]. While the first generation of HARP (HARPv1) connects a CPU to an FPGA only through a coherent QPI channel, the second generation of HARP (HARPv2) adds two non-coherent PCIe data communication channels between the CPU and the FPGA, resulting in a hybrid CPU-FPGA communication model.

To better understand and compare these platforms, we conducted a quantitative analysis using micro-benchmarks to measure the effective bandwidth and latency of CPU-FPGA communication on these platforms. The results lead to the following key insights (see [56] for details):

Insight 1: The host-to-accelerator effective bandwidth can be much lower than the peak physical bandwidth (often the "advertised bandwidth" in the product datasheet). For example, the Xilinx SDAccel runtime system running on a Gen3x8 PCI-e bus achieves only 1.6 GB/s CPU-FPGA communication bandwidth to end users, while the PCIe peak physical bandwidth is 8 GB/s bandwidth [56]. Evaluating a CPU-FPGA platform using these advertised values is likely to result in a significant overestimation of the platform performance. Worse still, the relatively low effective bandwidth is not easy to achieve. In fact, the communication bandwidth for a small size of payload can be 100x smaller than the maximum achievable effective bandwidth. A specific application may not always be able to supply each communication transaction with a sufficiently large size of payload to reach a high bandwidth (which was encountered in our accelerator design of another kernel of CS-BWAMEM [57]). For streaming applications, the recent work on ST-Accel [58] greatly improved the CPU-FPGA latency and bandwidth with an efficient host-FPGA communication library, which supports zero-copy (to eliminate the overhead of buffer copy during the data transferring) and operating system kernel bypassing (to minimize the data transferring latency).

Insight 2: Both the private-memory and shared-memory platforms have opportunities to outperform each other. In general, a private-memory platform like Alpha Data reaches a lower CPU-FPGA communication latency and bandwidth because it has to transfer data from the host memory to the device memory on the FPGA board first in order to be accessed by the FPGA fabric, while its shared-memory counterpart allows the FPGA fabric to directly retrieve data from the host memory, thus simplifying the communication process and improving the latency and bandwidth. The opportunity of private-memory platforms, nevertheless, comes from the cases when the data in the FPGA device memory are reused by the FPGA accelerator multiple times, since the bandwidth of accessing the local device memory is generally higher than that of accessing the remote host memory. This is particularly beneficial for iterative algorithms like logistic regression where a large amount of read-only (training) data are iteratively referenced for many times while only the weight vector is being updated [59]. This trade-off is modeled in [56] to help accelerator designers estimate the effective CPU-FPGA communication bandwidth given the reuse ratio of the data loaded to the device memory. For latency-sensitive applications like high-frequency trading, online transaction processing, or autonomous driving, the shared-memory platform is preferred since it features a simpler communication stack and lower latency. Another low-latency configuration is to have FPGAs connected to the network switches directly, as done with the Microsoft Azure SmartNIC [60]. It provides not only low-latency processing of the network data and but also excellent scalability to form large programmable fabrics. However, since FPGAs on the Microsoft Azure is not yet open to the public, we could not provide a quantitative evaluation.

Insight 3: CPU-FPGA memory coherence is interesting, but not yet very useful in accelerator design, at least for now. The newly announced CPU-FPGA platforms, including CAPI, HARPv1 and HARPv2, attempt to provide memory coherence support between the CPU and the FPGA. Their implementation methodologies are similar: constructing a coherent cache on the FPGA fabric to realize the classic coherency protocol with the last-level cache of the host CPU.

However, although the FPGA fabric supplies megabytes of on-chip BRAM blocks, only 64KB (the HARP family) or 128KB (CAPI) of them are organized into the coherent cache. That is, these platforms maintain memory coherence for only less than 5% of the on-chip memory space, leaving the majority on-chip memory (BRAMs) to be managed by application developers, which defeat the original goal of providing simpler programming interface via memory coherency. Moreover, the current implementation of coherence cache access is not efficient. For example, the coherent cache access latency of the Intel HARP platform is up to 80ns, while the data stored in the on-chip BRAM blocks can be retrieved in only one FPGA cycle (5ns) [56]. Also, the coherent cache provides much less parallel access capability compared to the scratchpads that can potentially feed thousands of data per cycle. In fact, all existing CPU-FPGA platforms support only single-port caches, i.e., the maximal throughput of these cache structures is only one transaction per cycle, resulting in very limited bandwidth. As a consequence, for now accelerator designers may still have to explicitly manage FPGA on-chip memories.

C. Datacenter-Level Customization

Since many big-data applications require more than one compute server to run, it is natural to consider cluster-level or datacenter-level customization with FPGAs. Moreover, given the significant energy consumed by modern datacenters, energy reduction using FPGAs in the datacenter has the most impact. Since 2013, we explore the design options in heterogeneous datacenters with FPGA accelerators via quantitative studies on a wide range of systems, including an Xeon CPU cluster, an Xeon cluster with FPGA accelerator attached to the PCI-E bus, a low-power Atom CPU cluster, and a cluster of embedded ARM processors with on-chip FPGA accelerators.

To evaluate the performance and energy efficiency of various accelerator-rich systems, several real prototype hardware systems are built to experiment with real-world big-data applications.

1) Small-Core with On-Chip Accelerators: We built a customized cluster of low-power CPU cores with on-chip FPGA accelerator. The Xilinx Zynq SoC was selected as the experimental heterogeneous SoC, which includes a processing system based on dual ARM A9 cores and a programmable FPGA logic. The accelerators are instantiated on the FPGA logic and can be reconfigured during runtime. We build a cluster of eight Zynq nodes. Each node in the cluster is a Xilinx ZC706 board, which contains a Xilinx Zynq XC7Z045 chip. Each board also has 1GB of on-board DRAM and a 128GB SD card used as a hard disk. The ARM processor in the Zynq SoC shares the same DRAM controller as well as address space with the programmable fabrics. The processor can control the accelerators on the FPGA fabrics using two system buses. The memory is shared through four high-performance memory buses (HPs) and one coherent memory bus (ACP). All the boards are connected to a gigabit Ethernet switch.

A snapshot of the system is shown in Figure 6.The hardware layout of the Zynq boards and their connection is shown

Fig. 6: Snapshot of the prototype cluster

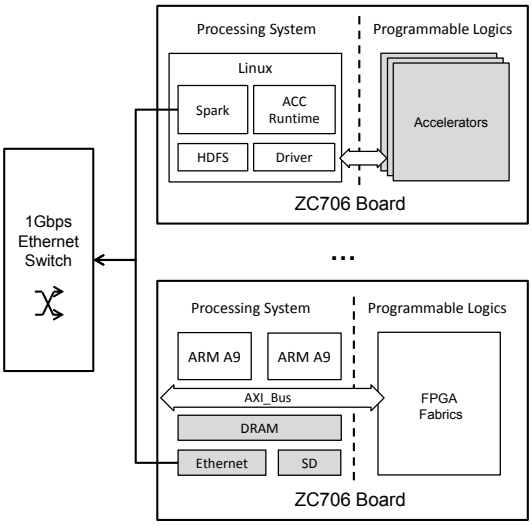


Fig. 7: System overview of the prototype cluster

in Figure 7 in the bottom box for the ZC706 board. The software setup and accelerator integration method are shown in the upper box in Figure 7. A lightweight Linux system is running on the ARM processors of each Zynq board; this provides drivers for peripheral devices such as Ethernet and SD card, and also controls the on-chip FPGA fabrics. To instantiate accelerators on the FPGA, we design a driver module to configure the control registers of the accelerators as memory-mapped IOs, and use DMA buffers to facilitate data transfers between the host system and the accelerators. Various accelerators are synthesized as FPGA configuration bitstreams and can be programmed on the FPGA at runtime.

2) Big-Core with PCIE-connected Accelerators: Similar to existing GPGPU platforms, FPGA accelerators can also be integrated into normal server nodes using the PCIE bus. Taking advantage of the energy efficiency of the FPGA chips, these PCIE accelerator boards usually do not require an external power supply, which makes it possible to deploy FPGA accelerators into datacenters without the need to modify existing infrastructures. In our experiments, we integrate AlphaData (AD) FPGA boards into our Xeon cluster shown in Figure 8, which has 20 Xeon CPU servers connected with both 1G and 10G Ethernet. Each server contains an FPGA board with a Xilinx Virtex-7 XC7VX690T-2 FPGA chip and 16GB of on-board memory.

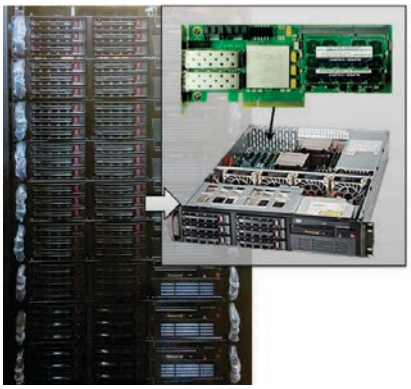


Fig. 8: Experimental cluster with standard server node integrated with PCI-E based FPGA board from AlphaData

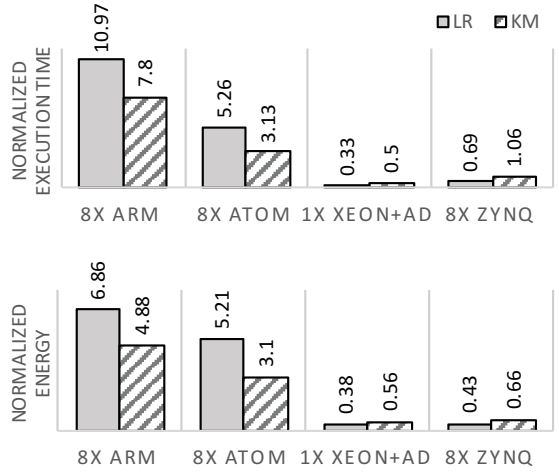


Fig. 9: Execution time (above) and energy consumption (below) normalized to the results on one Xeon server.

3) Baseline Small-Core and Big-Core Systems: For comparison purpose, we used a cluster of eight nodes of Intel Atom CPUs and a cluster of eight nodes of embedded ARM cores as the baseline of small-core CPU systems. The ARM cluster is the same as our prototype presented earlier in this section. For the baseline of big-core CPU systems, we re-use the server cluster in Figure 8 but without activation of the FPGAs.

4) Evaluation Results: We measure the total application time, including the initial data load and communication. The energy consumption is calculated by measuring the average power consumption during operation using a power meter and multiplying it by the execution time, since we did not observe significant variations of system power during the execution in our experiments. All the energy consumption measurements also include a 24-port 1G Ethernet switch.

a) Small-Core vs. Big-Core Systems: We first evaluate the performance and energy consumption between big-core with FPGA and small-core with FPGA using two popular machine learning algorithms: logistic regression (LR) and and k-means (KM) clustering. Figure 9 illustrates the execution time and energy consumption of running LR and KM applications on different systems. Notably, although the Atom or ARM processor have much lower power, it suffers long runtime for these applications. As a result, both the performance and energy efficiency of pure Atom and ARM clusters are worse than the single Xeon server, which confirms the argument in [61] that low-power cores could be less energy-efficient for computation-intensive workloads.

b) Big-Core vs. Big-Core + FPGA: We then present the effectiveness of FPGA accelerators in a common datacenter setup. Figure 9 includes the comparison between a CPU-only cluster and a cluster of CPUs with PCIE FPGA boards using LR, KM. For the machine learning workloads where most of the computation can be accelerated, FPGA can contribute to significant speedup with only a small amount of extra power. More specifically, the big-core plus FPGA configuration achieves 3.05× and 1.47× speedup for LR and KM respectively and reduces the overall energy consumption to 38% and 56% of the baseline respectively (which implies a 2-3x energy reduction).

c) Small-Core + FPGA vs. Big-Core + FPGA: Several observations can be drawn from the results in Figure 9. First, for both small-core and big-core systems, the FPGA accelerators provide significant performance and energy-efficiency improvement—not only for kernels but also for the entire application. Second, compared to big-core systems, small-core systems benefit more from FPGA accelerators. This means that it is more crucial to provide accelerator support for future small-core-based datacenters. Finally, although the kernel performance on eight Zynq FPGAs is better than one AD FPGA, the application performance of Xeon with AD-FPGA is still 2× better than Zynq. This is because on Zynq, the non-acceleratable part of the program, such as disk I/O and data copy, is much slower than Xeon. On the other hand, the difference in energy-efficiency between Xeon plus FPGA and Zynq is much smaller.

In parallel to the effort by CDSC on incorporating and enabling FPGAs in computing clusters, a number of large datacenter operators started supporting FPGAs in private or public clouds. Baidu and Microsoft announced using FPGAs in their datacenters in 2014 [8], [9], so far only for first-party internal use. Amazon introduced FPGAs in its AWS public computing cloud in late 2016 [11] for third-party use. This trend was quickly followed by other public cloud providers, such as Alibaba [12] and Huawei [13].

III. Compilation Support

The successful adoption of customizable computing depends on ease of programming of such accelerator-rich architectures (ARAs). Therefore, a significant part of CDSC research has been devoted to develop the compilation and runtime support for ARAs. Since the chip-level ARA is still at its infancy (although we did develop a compilation system for composable accelerator building blocks discussed in Section II-A based on efficient pattern matching [62]), we focus most of our effort on improving the acceleration design and deployment on FPGAs for server-node level and datacenter-level integration.

In this section we present the compilation support that automatically generates accelerators from user-written functions in high-level programming languages such as C/C++. We first

introduce the modern commercial high-level synthesis (HLS) tool and illustrate the challenges of its programming model to generate high-performance accelerators in Section III-A. To solve these challenges, we present the Merlin compiler framework [63], [64] along with a general architecture template for automated design space exploration in Section III-B and Section III-C, respectively. Finally, in Section III-D, we show that special architecture templates, such as systolic array [65], can be incorporated into the Merlin compiler to achieve much higher performance for targeted applications (in this case for deep learning).

A. Challenges of Commercial High-Level Synthesis Tools

In recent years, the state-of-the-art commercial HLS tools, Xilinx Vivado HLS [66](based on AutoPilot [17]), SDAccel [50] and Intel FPGA SDK for OpenCL [67], have been widely used to fast prototype user-defined functionalities in C-based languages (e.g., C/C++ and OpenCL) on FPGAs without involving register-transfer level (RTL) descriptions. In particular, for the OpenCL based flow, it provides a set of APIs on the host (CPU) side to abstract away the underlying implementation details of protocols and drivers to communicate with FPGAs. On the kernel (FPGA) side, the tool compiles a user input C-based program with pragmas to the LLVM intermediate representation (IR) [68] and performs IR-level scheduling to map the accelerator kernel to the FPGA. Although these HLS tools indeed improve the FPGA programmability (compared to RTL based design methodology), they are still facing some challenges.

Challenge 1: Tedious OpenCL routine. The OpenCL programming model for an application host requires the programmer to use OpenCL APIs to create an OpenCL context, load the accelerator bitstream, specify CPU-FPGA data transfer, configure accelerator interfaces, launch the kernel, and collect the results. For example, a kernel with two input and one output buffers as its interface will require roughly 40 code statements with OpenCL APIs in the host program. Clearly, it is too tedious to be done manually by programmers.

Challenge 2: Impact of code transformation on performance. The input C code matters a lot to the HLS synthesis result. For example, the HLS tool always schedules a loop with a variable trip-count to be executed sequentially even if it does not have carried dependency. However, in this case, applying loop tiling with a suitable tiled size could let the HLS tool to generate multiple processing elements (PEs) and schedule them to execute tasks in parallel. As a result, heavy code reconstruction with hardware knowledge is usually required for designers to deliver high-performance accelerators, which creates substantial learning barrier for a typical software programmer.

Challenge 3: Manual design space exploration (DSE). Finally, assuming the C program has been well reconstructed, the modern HLS tools further require designers to specify the task scheduling, external memory access, and on-chip memory organization using a set of pragmas. This means that designers have to dig into the generated design and analyze its performance bottleneck, or even use trial-and-error approach to realize the best position and value for pragmas to be specified.

B. The Merlin Compiler

To address these challenges and enable software programmers with little circuit and microarchitecture background to design efficient FPGA accelerators, the researchers in CDSC developed CMOST (Customization, Mapping, Optimization, Scheduling and Transformation) [69], a push-bottom source-to-srouce compilation and optimization framework, to generate high-quality HLS friendly C or OpenCL from fairly generic C/C++ code with minimal programmer intervention. It has been further extended by Falcon Computing Solutions [70] to become a commercial strength tool named the Merlin compiler [63], [64]. The Merlin compiler is a system-level compilation framework that adopts an OpenMP-like [71] programming model—i.e., a C-based program with a small set of pragmas to specify the accelerator kernel scope and task scheduling.

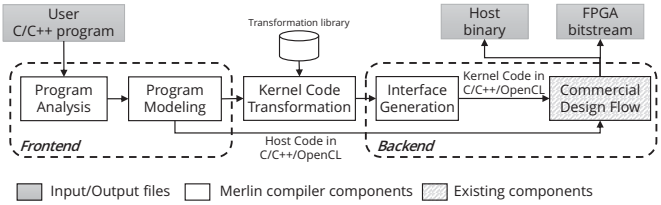


Fig. 10: The Merlin Compiler execution flow

Figure 10 presents the Merlin compiler execution flow. It leverages the ROSE compiler infrastructure [72] and polyhedral framework [73] for abstract syntax tree (AST) analysis and transformation. The front end stage analyzes the user program and separates host and computation kernel. It then analyzes the data transfer and inserts necessary OpenCL APIs to the host code so that *Challenge 1* can be eliminated. In addition, the kernel code transformation stage performs source-to-source code transformation according to user-specified pragmas, as shown in Table II. Note that the Merlin compiler will perform all necessary code reconstructions to make a transformation effective. For example, when performing loop parallelism, the Merlin compiler not only tiles and unrolls a loop but also conducts memory partitioning for the sake of avoiding bank conflict [26]. This approach largely address *Challenge 2* as it allows the programmers to use some simple pragmas to specify the code transformation without considering any underlying architecture issues. After both the host and kernel code are prepared, the back end stage launches the commercial HLS tool to generate the host binary as well as FPGA bitstream.

TABLE II: Kernel Pragmas of Merlin Compiler

Transformation	Target	Parameters	Description
Data tiling	Loop	tilesize=S	Tile the loop and create on-chip buffers to cache the data with size S.
	Example: #pragma Accel data_tiling tilesize=16		
Memory Coalescing	Buffer	bitwidth=b	Pack DRAM buffer to b bits.
	Example: #pragma Accel bitwidth variable=buf factor=512		
Pipeline	Loop	N/A	Create a coarse- or fine-grained pipeline (dataflow).
	Example: #pragma Accel pipeline		
Parallelism	Loop	factor=N	Tile the loop and create N processing elements (PEs).
	Example: #pragma Accel parallel factor=4		

Moreover, the Merlin compiler can further improve the FPGA programmability by making "semi-automatic" design optimization : instead of manually reconstructing the code to make one optimization operation effective, programmers now can simply place a pragma and let the Merlin compiler do the necessary changes. The ongoing work includes developing an automated DSE framework that leverages reinforcement learning algorithms to efficiently explore the design space [74] for code transformation. This will fully address *Challenge 3*.

Given the total flexibility of FPGA designs, the accelerator design space is immense. One way to manage the search complexity is to use certain architecture templates as a guide when appropriate. We shall discuss two architecture template in the next two sections.

We also observe and summarize some common computational patterns for most cases. Accordingly, we develop a general architecture template [75], which we will present in the next subsection, to rapidly identify the optimal design point for the case that can be fit in.

C. Support of CPP Architecture

The Merlin compiler is particularly suitable for the Composable, Parallel and Pipeline (CPP) architecture [75], as shown in Figure 11. Many designs map well to the CPP architecture, which facilitates the high-performance accelerator design with the following features:

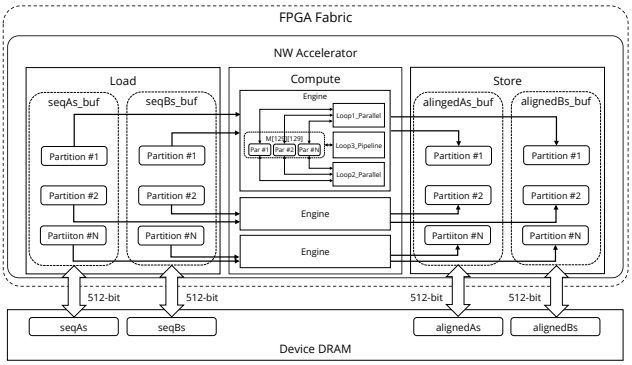


Fig. 11: The Example of CPP Architecture

(1): Coarse-grained pipeline with data caching. The overall CPP architecture consists of three stages: `load`, `compute` and `store`. The user-written kernel function only corresponds to the `compute` module instead of defining the entire accelerator. The inputs are processed block by block, i.e., iteratively loading a certain number of sequence pairs into on-chip buffers (Stage `load`) while the outputs are stored back to DRAM (Stage `store`). Different blocks are processed in a pipelined manner so that off-chip data movement only happens in the `load` and `store` stages, leaving the data accesses of computation completely on chip.

(2): Loop scheduling. The CPP architecture maps every loop statement presented in the computation kernel function to either 1) a circuit that processes different loop iterations in parallel, 2) a pipeline where the loop body corresponds to the pipeline stages, or 3) recursive composition of 1) and 2). Such a regular structure allows us to effectively search for the optimally solution.

(3): On-chip buffer reorganization. In the CPP architecture, all on-chip BRAM buffers are partitioned to meet the port requirement of parallel circuits, where the number of partitions of each buffer is determined by the duplication factor of the parallel circuit that connects to the buffer.

We note that the CPP architecture is general enough to cover broad classes of applications. Specifically, CPP architecture is applicable as long as the computational kernel is synthesizable and "cacheable", i.e. the input data can be partitioned and processed block by block. Any Map-Reduce [76] or Spark [44] programs fall into this category. For example, we could apply the CPP architecture to more than 80% of Machsuite [77] benchmarks. But computational kernels that have extensive random accesses to a large memory footprint, such as the breadth-first search (BFS) algorithm or page-rank algorithm algorithm of large graphs, are not suitable for the CPP architecture.

One of the most important advantages of having the CPP architecture is that we can define a clear design space and derive a set of analytical models to quantify the performance and resource utilization. It makes the automatic design space exploration practical. In [75], we develop several pruning strategies to reduce the design space so that it can be exhaustively searched in minutes. The evaluation result shows that our automatic DSE achieves on average a 72× speedup and 260.4× energy improvement for a broad class of computation kernels compared to the out-of-box synthesis results by SDAccel [50].

D. Support of Systolic Array Architecture for Deep Learning

Systolic array [65] is another architecture template supported by the Merlin compiler. The general systolic array support is still under study. Our initial focus is to support the design of convolutional neural network (CNN) accelerator with systolic array.

CNN is one of the key algorithms for the deep learning applications, ranging from image/video classification, recognition, and analysis to natural language understanding, advances in medicine, and more. The core computation in the algorithm can be summarized as a convolution operation on the multiple dimensional arrays. Since the algorithm offers the potential of massive parallelization and extensive data reuse, FPGA implementations of CNN have seen an increased amount of interest from academia [78]–[87]. Among these, systolic array is proved to be a promising architecture [84], [88]. A systolic array architecture is a specialized form of parallel computing with a deeply pipelined network of PEs. With the regular layout and local (nearest neighbour) communication, which is suitable for large-scale parallelism on FPGAs with high clock frequency.

In order to support systolic array architecture in the Merlin compiler, we first implemented a high throughput systolic array design template in OpenCL with parametrized PE and buffer sizes. In addition, we defined a new pragma for programmers to specify the code segment, as shown in Code 1,

where the loop bounds are the constants of a CNN layer configuration. As a result, our goal is to map Code 1 to the predefined template with the optimal performance. The solution space is large due to the following degree of freedom 1) selecting three loops in Code 1 to map to 2-D systolic array architecture with in-PE parallelism (note that some loops cannot be mapped to the 2-D systolic architecture and we developed necessary and sufficient conditions for mapping), 2) selecting the suitable PE array shape to maximize the resource efficiency and operating frequency, and 3) determining the data reuse strategy under the on-chip resource constraint. The detailed analysis of the design space can be found in [88].

Code 1: A Convolutional Layer with The Merlin Pragma

```
#pragma ACCEL systolic auto
for(o = 0; o < O; o++) // Output feature #
for(i = 0; i < I; i++) // Input feature #
for(c = 0; c < C; c++) // Feature column
for(r = 0; r < R; r++) // Feature row
for(p = 0; p < K; p++) // Kernel weight
for(q = 0; q < K; q++) // Kernel height
  OUT[o][r][c] += W[o][i][p][q] *
                  IN[i][r+p][c+q];
```

Since all these design challenges and their interplay need to be considered in a unified way to achieve a global optimal, we develop a highly accurate analytical model ($< 2\%$ error on average) to estimate the design throughput and resource utilization given a design configuration. Furthermore, to reduce the design space, we present two pruning strategies to prune the design space while preserving the optimality. (i) We consider the resource usage efficiency. Since the clock frequency will not drop significantly even with high DSP utilization due to the scalability of the systolic PE array architecture we adopted, we can prune the design points with low DSP utilization.

(ii) We consider the data reuse strategies. We know that BRAM sizes in the implementation are always rounded up to the power of two, so we prune the design space by only exploring the power-of-two data reuse strategies. The pruned design space of data reuse strategies can still cover the optimal solution in the original design space because 1) our throughput object function is a monotonic non-decreasing function of the BRAM buffer size, and 2) BRAM utilization is the same as another strategy whose values have the same rounding up the the power of two. By applying the pruning on the data reuse strategies, the design space reduces exponentially so that we are able to perform an exhaustive search to find the best strategy and result in an additional 17.5× saving on the average search time for AlexNet convolutional layers. In fact, our DSE implementation is able to exhaustively explore the pruned design space with the analytical model in several minutes, which was several hundreds of hours when exploring the full design space. Evaluation results show that our design achieves up to 1171 Gops on Intel Arria 10 with full automation [88].

We would like to point out that although many accelerator design efforts in the industry are still done using RTL programming for performance optimization, as such the database acceleration effort at Baidu [89] and the deep learning acceleration effort at Microsoft [90], we believe that the move to high-level programming language based accelerator designs is inevitable, especially when the FPGAs are introduced in the public clouds. The potential user base for FPGA designs is much larger. Our goal is to support high-level programming flow to "democratize customizable computing".

IV. Runtime Support

After accelerators being developed using the compilation tool, they need to be integrated with the applications and deployed onto computing servers or datacenters with runtime support.

Modern big data processing systems, such as Apache Hadoop [76] and Spark [44], have evolved to an unprecedented scale. As a consequence, cloud service providers, such as Amazon and Microsoft, have expanded their datacenter infrastructures to meet the ever-growing demands for supporting big data applications. One key question is: *How can we easily and efficiently deploy FPGA accelerators into state-of-the-art big data computing systems like Apache Spark [44] and Hadoop YARN [91]?* To achieve this goal, both programming abstractions and runtime support are needed to make these existing systems programmable to FPGA accelerators.

A. Programming Abstraction

In this subsection, we present the Blaze framework to provide accelerator-as-a-service [92] (see Section II-B1 for the motivation), which provides programming abstraction and runtime support for easy and efficient FPGA (and GPU as well) deployments in datacenters. To provide a user-friendly programming interface for both application developers and accelerator designers, we abstract accelerators as software libraries so that application developers can use the hardware accelerators as if they are using software code while accelerator designers can easily package their design to a shared library.

1) Application Interface: The Blaze programming interface for user applications is designed to support accelerators with minimal code changes. To achieve this, we extend the Spark `RDD` to `AccRDD` which supports accelerated transformations. Blaze is implemented as a third-party package that works with the existing Spark framework[3] without any modification of Spark source code. Thus, Blaze is not specific to a particular version of Spark. We explain the usage of `AccRDD` with an example of logistic regression shown in Listing 2.

In Listing 2, training data samples are loaded from a file and stored to an RDD `points`, and are used to train `weights` by calculating gradients in each iteration. To accelerate the gradient calculation with Blaze, first the RDD `points` needs to be extended to AccRDD `train` by calling the Blaze API `wrap`. Then an accelerator function, `LogisticAcc`, can be passed to the `.map` transformation of the AccRDD. This accelerator function is extended from the Blaze interface `Accelerator` by specifying an accelerator id and an

[3]Blaze also supports C++ applications with similar interfaces, but we will mainly focus on Spark applications in this paper.

optional `compute` function for the fall-back CPU execution. The accelerator id specifies the desired accelerator service, which in the example is "LRGradientCompute." The fall-back CPU function will be called when the accelerator service is not available. This interface is provided with fault-tolerance and portability considerations. In addition, Blaze also supports caching for Spark broadcast variables to reduce JVM-to-FPGA data transfer.

Code 2: Blaze application example (Spark Scala)

```
val points = sc.textFile(filePath).cache()
val train = blaze.wrap(points)
for (i <- 0 until ITERATIONS) {
  bcW = sc.broadcast(weights)
  val gradients = train.map(
     new LogisticAcc(bcW) ).reduce(a + b)
  weights -= gradients
}
class LogisticAcc(w: Broadcast_var[V])
   extends Accelerator[T, U] {
  val id: String = "LRGradientCompute"
  def call(p: T): U = {
    localGradients.compute(p, w.value)
  }
  ...
}
```

The application interface of Blaze can be used by library developers as well. For example, Spark MLlib developers can include Blaze-compatible codes to provide acceleration capabilities to end users. With Blaze, such capabilities are independent of the execution platform. When accelerators are not available, the same computation will be performed on CPU. In this case, accelerators will be totally transparent to the end users. In our evaluation, we created several implementations for Spark MLlib algorithms such as logistic regression and K-Means using this approach.

2) Accelerator Interface: For accelerator designers, the programming experience is decoupled with any application-specific details. An example of the interface implementing the "LRGradientCompute" accelerator is shown in Listing 3.

Code 3: Blaze accelerator example (C++)

```
class LogisticTask : public Task {
public:
  LogisticTask(): Task(NUM_ARGS)
  // overwrite the compute function
  virtual void compute() {
    int num_elements = getInputLength(...);
    double *in = (float*)getInput(...);
    double *out = (float*)getOutput(...);
    // perform computation
    ...
  }
};
```

Our accelerator interface hides details of FPGA accelerator initialization and data transfer by providing a set of APIs. In this implementation, for example, the user inherits the provided template, `Task`, and the input and output data can be obtained by simply calling `getInput` and `getOutput` APIs. No explicitly OpenCL buffer manipulation is necessary for users. The runtime system will prepare the input data and schedule it to the corresponding task. The accelerator designer can use any available programming framework to implement an accelerator task as long as it can be integrated with an interface in C++.

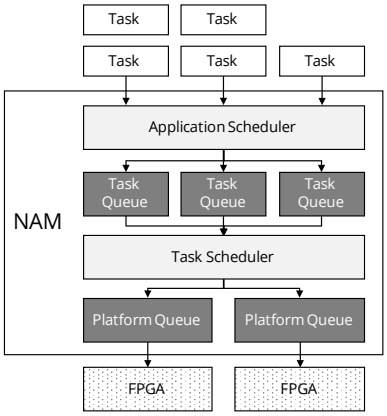


Fig. 12: Node accelerator manager design to enable FPGA accelerators as a service (AaaS).

B. Node-Level Runtime Management

Blaze facilitates Accelerator-as-a-service (AaaS) in the Node Accelerator Manager (NAM) through two levels of queues: *task queues* and *platform queues*. The architecture of NAM is illustrated in Figure 12. Each *task queue* is associated with a "logical accelerator", which represents an accelerator library routine. When an application task requests a specific accelerator routine, the request is put into the corresponding *task queue*. Each *platform queue* is associated with a "physical accelerator", which represents an accelerator hardware platform such as an FPGA board. The tasks in *task queue* can be executed by different *platform queues* depending on the availability of the implementations. For example, if both GPU and FPGA implementations of the same accelerator library routine are available, the task of that routine can be executed on both devices.

This mechanism is designed with three considerations: 1) application-level accelerator sharing, 2) minimizing FPGA reprogramming, and 3) efficient overlapping of data transfer and accelerator execution to alleviate JVM-to-FPGA overhead.

In Blaze, accelerator devices are owned by NAM rather than individual applications, as we observed that in most big-data applications, the accelerator utilization is less than 50%. If the accelerator is owned by a specific application, then much of the time it will be spent in idle, wasting the FPGA resource and energy. The application-level sharing inside NAM is managed by a scheduler that sits between application requests and *task queues*. Our initial implementation is a simple first-come-first-serve scheduling policy. We leave the exploration of different policies to future work.

The downside of providing application sharing is the additional overheads of data transfer between the application process and NAM process. For latency-sensitive applications, Blaze also offers a reservation mode where the accelerator device is reserved for a single application, i.e., a NAM instance will be launched inside the application process.

The design of the *platform queue* focuses on mitigating the large overhead in FPGA reprogramming. For a processor-

based accelerator such as GPU to begin executing a different "logical accelerator," it simply means loading another program binary, which incurs minimum overhead. With FPGA, on the other hand, the reprogramming takes much longer (can be 1∼2 seconds). Such a reprogramming overhead makes it impractical to use the same scheme as the GPU in the runtime system. In Blaze, a second scheduler sits between *task queues* and *platform queues* to avoid frequent reprogramming of the same FPGA device. Its scheduling policy is similar to the GAM scheduling to be presented in the next subsection.

CPU-FPGA Co-Management. In our initial Blaze runtime, after the computation-bound kernel is offloaded to the accelerators, the CPU stays idle, which wastes the computing resources. To address this issue, we further propose a dataflow execution model and an interval-based scheduling algorithm to effectively orchestrate the computation between multiple CPU cores and the FPGA on the same node, which greatly improves the overall system resource utilization. In our case study on genome data in-memory sorting, we find that our adaptive CPU-FPGA co-scheduling achieves 2.6x speedup over the 12-threaded CPU baseline [93].

C. Datacenter-Level Runtime Management

Recall that the Blaze runtime system integrates with Hadoop YARN to manage accelerator sharing among multiple applications. Blaze includes two levels of accelerator management. A global accelerator manager (GAM) oversees all the accelerator resources in the cluster and distributes them to various user applications. Node accelerator managers (NAMs) sit on each cluster node and provide transparent accelerator access to a number of heterogeneous threads from multiple applications. After receiving the accelerator computing resources from GAM, the Spark application begins to offload computation to the accelerators through NAM. NAM monitors the accelerator status, handles JVM-to-FPGA data movement and accelerator task scheduling. NAM also performs a heartbeat protocol with GAM to report the latest accelerator status.

Blaze execution flow. During system setup, the system administrator can register accelerators to NAM through APIs. NAM reports accelerator information to GAM through heartbeat. At runtime, user applications request containers with accelerators from GAM. Finally during application execution time, user applications can invoke accelerators and transfer data to and from accelerators through NAM APIs.

Accelerator-centric scheduling. In order to solve the global application placement problem considering the overwhelming FPGA reprogramming overhead, we propose to manage the logical accelerator functionality, instead of the physical hardware itself, as a resource to reduce such reprogramming overhead. We extend the *label-based scheduling* mechanism in YARN to achieve this goal: instead of configuring node labels as 'FPGA,' we propose to use accelerator functionality (e.g., 'KMeans-FPGA,' 'Compression-FPGA') as node labels. This helps us to differentiate applications that are using the FPGA devices to perform different computations. We can delay the scheduling of accelerators with different functionalities onto the same FPGA to avoid reprogramming as much as possible. Different from the current YARN solution, where node labels are configured into YARN's configuration files, node labels in Blaze are configured into NAM through command-line. NAM then reports the accelerator information to GAM through heartbeats, and GAM configures these labels into YARN.

Our experiment results on a 20-node cluster with 4 FPGA nodes show that static resource allocation and the default resource allocations (i.e., YARN resource scheduling policy) are 27% and 22% away from theoretical optimal results, while our proposed runtime is only 9% away from the optimal results.

At this point, the use of GAM is limited, as the public cloud providers do not yet allow multiple users to share FPGA resources. The NAM is very useful for accelerator integration, especially with a multi-threaded host program or to bridge different level of programming abstraction (e.g. from the JVM to FPGAs). For example, the NAM is used extensively in the genomic acceleration pipeline developed by Falcon Computing [70].

V. Concluding Remarks

This paper summarizes research contributions from the decade-long research of CDSC on customizable architectures at chip level, server-node level, and datacenter level, as well as the compilation and runtime support. Compared to classical von Neumann architecture with instruction-based temporally multiplexing, these architectures achieve significant performance and energy efficiency gain with extensive use of customized accelerators via spatial computing. They are gaining greater importance as we come to near the end of Moore's Law scaling. There are many new research opportunities.

With Google's success of TPU chip for deep learning acceleration [94], we expect a lot more activities on chip-level ARA in the coming years. The widely used GPUs are in fact a class of important and efficient chip-level accelerators for SIMD and SPMD workloads, which may further refine and specialize to certain application domains (e.g. deep learning and autonomous driving).

FPGAs remain to offer a very good trade-off of flexibility and efficiency. In order to compete with ASIC based accelerators in terms of performance and energy efficiency, we suggest FPGA vendors to consider two directions to further refine the FPGA architectures: (i) include coarser-grain computing blocks, such as SIMD execution units or CGRA-like structures, and (ii) simplify the clocking and I/O structures, which were introduced mostly for networking and ASIC prototyping applications. Such simplification will not only save the chip area to accommodate more computing resources, but also has the potential to greatly shorten the compilation time (which is a serious shortcoming of existing FPGAs), as it will make placement and routing a lot easier. Another direction is to build efficient overlay architectures on top of existing FPGAs to avoid the long compilation time.

In terms of compilation support, we see two promising directions. On one hand, we further increase the level of programming abstraction to support domain-specific languages (DSLs), such as Caffe [87], [95] TensorFlow [96], Halide [97]

and Spark [74]. In fact, these DSLs have initial supports for FPGA compilation already and further enhancement is ongoing. On the other hand, we shall consider specialized architecture supports to enable better design space exploration to achieve the optimal synthesis results. We have good success with the support of stencil computation [28], [98], systolic arrays [88], and the CPP (composable parallel and pipelining) architecture [75]. We hope to capture more computation patterns and corresponding microarchiteture templates, and incorporate them in our compilation flow.

Finally, we are adding the cost optimization as an important metric in our runtime management tools of accelerator deployment in datacenter applications [99], and also considering the extension of more popular runtime systems, such as Kubernetes [100] and Mesos [101] for acceleration support.

ACKNOWLEDGMENT

We thank all the CDSC faculty members, postdocs, and students for their contribution. A list of all contributors can be found in our CDSC website: https://cdsc.ucla.edu/people. This work is partially supported by the Center for Domain-Specific Computing under the NSF InTrans Award CCF-1436827; funding from CDSC industrial partners including Baidu, Fujitsu Labs, Google, Huawei, Intel, IBM Research Almaden and Mentor Graphics; C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; and Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA).

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[3] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011, pp. 365–376.

[4] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable domain-specific computing," *IEEE Design Test of Computers*, vol. 28, no. 2, pp. 6–15, March 2011.

[5] U. Newsroom, "Nsf awards ucla $10 million to create customized computing technology," 2009. [Online]. Available: http://newsroom.ucla.edu/releases/ucla-engineering-awarded-10-million-97818

[6] P. Schaumont and I. Verbauwhede, "Domain-specific codesign for embedded security," *Computer*, vol. 36, no. 4, pp. 68–74, April 2003.

[7] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14, 2014, pp. 180:1–180:6.

[8] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA-41*, 2014.

[9] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, "Sda: Software-defined accelerator for large-scale dnn systems," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, Aug 2014, pp. 1–23.

[10] "Intel to Start Shipping Xeons With FPGAs in Early 2016," 2016. [Online]. Available: http://www.eweek.com/servers/intel-to-start-shipping-xeons-with-fpgas-in-early-2016.html

[11] "Amazon ec2 f1 instance," 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[12] "Alibaba f2 instance," 2017. [Online]. Available: https://www.alibabacloud.com/help/doc-detail/25378.htm#f2

[13] "Huawei fpga-accelerated cloud server," 2017. [Online]. Available: http://www.huaweicloud.com/en-us/product/fcs.html

[14] J. Cong, Z. Fang, M. Gill, and G. Reinman, "Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 380–387, 2015.

[15] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich cmps," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, 2012, pp. 843–849.

[16] ——, "Architecture support for domain-specific accelerator-rich cmps," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 131:1–131:26, Apr. 2014.

[17] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," 2011.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, 2008.

[19] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12, 2012.

[20] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *International Symposium on Low Power Electronics and Design (ISLPED)*, Sept 2013, pp. 305–310.

[21] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1258–1270, Oct 2005.

[22] N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. Al Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini, "Exploring architectural heterogeneity in intelligent vision systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 1–12.

[23] H. Park, Y. Park, and S. Mahlke, "Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 370–380.

[24] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sept 2012.

[25] B. D. Sutter, P. Raghavan, and A. Lambrechts, "Coarse-grained reconfigurable array architectures," in *Handbook of Signal Processing Systems*. Springer, 2013.

[26] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *TODAES*, 2011.

[27] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 199–208.

[28] J. Cong, P. Li, B. Xiao, and P. Zhang, "An optimal microarchitecture for stencil computation acceleration based on non-uniform partitioning of data reuse buffers," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 77:1–77:6.

[29] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman, and Y. Zou, "An energy-efficient adaptive hybrid cache," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ser. ISLPED '11, 2011, pp. 67–72.

[30] Y.-T. Chen, J. Cong, H. Huang, B. Liu, C. Liu, M. Potkonjak, and G. Reinman, "Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '12, 2012, pp. 45–50.

[31] J. Cong, M. A. Ghodrat, M. Gill, C. Liu, and G. Reinman, "Bin: A buffer-in-nuca scheme for accelerator-rich cmps," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12, New York, NY, USA, 2012, pp. 225–230.

[32] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The accelerator store: A shared memory framework for accelerator-based systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 48:1–48:22, Jan. 2012.

[33] C. F. Fajardo, Z. Fang, R. Iyer, G. F. Garcia, S. E. Lee, and L. Zhao, "Buffer-integrated-cache: A cost-effective sram architecture for handheld and embedded platforms," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11, 2011, pp. 966–971.

[34] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 37–48.

[35] M. F. Chang, J. Cong, A. Kaplan, M. Naik, G. Reinman, E. Socher, and S. W. Tam, "Cmp network-on-chip overlaid with multi-band rf-interconnect," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Feb 2008, pp. 191–202.

[36] J. Cong, M. Gill, Y. Hao, G. Reinman, and B. Yuan, "On-chip interconnection network for accelerator-rich architectures," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15, 2015, pp. 8:1–8:6.

[37] Y.-T. Chen, J. Cong, M. Gill, G. Reinman, and B. Xiao, "Customizable computing," *Synthesis Lectures on Computer Architecture*, vol. 10, no. 3, pp. 1–118, 2015.

[38] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, , and D. A. Wood, "The gem5 simulator," *Computer Architecture News*, 2011.

[39] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 469–480.

[40] J. Cong, Z. Fang, M. Gill, F. Javadi, and G. Reinman, "Aim: Accelerating computational genomics through scalable and noninvasive accelerator-interposed memory," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, 2017, pp. 3–14.

[41] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *VLSI Technology (VLSIT), 2012 Symposium on*. IEEE, 2012, pp. 87–88.

[42] J. Kim and Y. Kim, "Hbm: Memory solution for bandwidth-hungry processors," in *2014 IEEE Hot Chips 26 Symposium (HotChips)*.

[43] Y.-T. Chen, J. Cong, S. Li, M. Peto, P. Spellman, P. Wei, and P. Zhou, "CS-BWAMEM: A fast and scalable read aligner at the cloud scale for whole genome sequencing," *High Throughput Sequencing Algorithms and Applications (HITSEQ)*, 2015.

[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, 2012, pp. 2–2.

[45] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *ArXiv e-prints*, Mar. 2013.

[46] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195 – 197, 1981.

[47] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 199–202.

[48] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When apache spark meets fpgas: a case study for next-generation dna sequencing acceleration," in *HotCloud*, 2016.

[49] *ADM-PCIE-7V3 Datasheet*, Xilinx, 1 2017, rev. 1.3.

[50] "SDAccel Development Environment," http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[51] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[52] B. Brech, J. Rubio, and M. Hollinger, "IBM Data Engine for NoSQL - Power Systems Edition," IBM Systems Group, Tech. Rep., 2015.

[53] T. M. Brewer, "Instruction set innovations for the convey hc-1 computer," *IEEE Micro*, no. 2, pp. 70–79, 2010.

[54] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia *et al.*, "A reconfigurable computing system based on a cache-coherent fabric," in *ReConFig*, 2011.

[55] D. Sheffield, "Ivytown xeon+ fpga: The harp program," 2016.

[56] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *DAC-53*, 2016.

[57] M. C. F. Chang, Y. T. Chen, J. Cong, P. T. Huang, C. L. Kuo, and C. H. Yu, "The smem seeding acceleration for dna sequence alignment," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 32–39.

[58] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong, "St-accel: A high-level programming platform for streaming applications on fpga," in *The 26th IEEE International Symposium on Field-Programmable Custom Computing Machines, 2018.*, ser. FCCM '18, 2018.

[59] J. Cong, M. Huang, D. Wu, and C. H. Yu, "Invited - heterogeneous datacenters: Options and opportunities," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 16:1–16:6.

[60] "Azure SmartNIC," 2018. [Online]. Available: https://www.microsoft.com/en-us/research/project/azure-smartnic/

[61] L. Keys, S. Rivoire, and J. D. Davis, "The search for Energy-Efficient building blocks for the data center," in *Computer Architecture*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1 Jan. 2012, pp. 172–182.

[62] J. Cong, H. Huang, and M. A. Ghodrat, "A scalable communication-aware compilation flow for programmable accelerators," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2016, pp. 503–510.

[63] J. Cong, M. Huang, P. Pan, Y. Wang, and P. Zhang, "Source-to-source optimization for HLS," in *FPGAs for Software Programmers*. Springer International Publishing, 2016.

[64] J. Cong, M. Huang, P. Pan, D. Wu, and P. Zhang, "Software infrastructure for enabling fpga-based accelerations in data centers: Invited paper," in *ISLPED*, 2016.

[65] H. T. Kung and C. E. Leiserson, *Algorithms for VLSI Processor Arrays*, 1979.

[66] "Xilinx Vivado HLS," http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm.

[67] "Intel SDK for OpenCL Applications," https://software.intel.com/en-us/intel-opencl.

[68] "LLVM Language Reference Manual," http://llvm.org/docs/LangRef.html.

[69] P. Zhang, M. Huang, B. Xiao, H. Huang, and J. Cong, "CMOST: A system-level fpga compilation framework," in *DAC-52*, 2015.

[70] "Falcon Computing Solutions, Inc," http://falcon-computing.com.

[71] "OpenMP," http://openmp.org.

[72] "Rose Compiler Infrastructure," http://rosecompiler.org/.

[73] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong, "Improving polyhedral code generation for high-level synthesis," in *CODES+ISSS*, 2013.

[74] C. H. Yu, P. Wei, P. Zhang, M. Grossman, V. Sarker, and J. Cong, "S2FA: An accelerator automation framework for heterogeneous computing in datacenters," in *DAC '18*.

[75] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *DAC '18*.

[76] "Apache Hadoop," https://hadoop.apache.org, accessed: 2016-05-24.

[77] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *IISWC*, 2014.

[78] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A Programmable Parallel Accelerator for Learning and Classification," in *PACT*, 2010.

[79] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A Massively Parallel Coprocessor for Convolutional Neural Networks," in *ASAP*, 2009.

[80] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," *ISCA*, 2010.

[81] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for Convolutional Networks," in *FPL*, 2009.

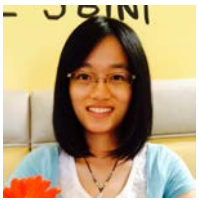
[82] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for Convolutional Neural Networks," in *ICCD*, 2013.

[83] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.

[84] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *FPGA*, 2016.
[85] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *FCCM*, 2016.
[86] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *FPGA*, 2016.
[87] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *ICCAD*, 2016.
[88] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *DAC*, 2017.
[89] J. Ouyang, W. Qi, Y. Wang, YichenTu, J. Wang, and B. Jia, "Sda: Software-defined accelerator for general-purpose big data analysis system," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, Aug 2016, pp. 1–23.
[90] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating deep neural networks at datacenter scale with the brainwave architecture," in *2017 IEEE Hot Chips 29 Symposium (HotChips)*.
[91] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.
[92] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 456–469.
[93] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu, "Cpu-fpga coscheduling for big data applications," vol. PP, no. 99, 2017, pp. 1–4.
[94] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12.
[95] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *TCAD*, 2018.
[96] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 152–159.
[97] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 26:1–26:25, Aug. 2017.
[98] Y. Chi, P. Zhou, and J. Cong, "An optimal microarchitecture for stencil computation with data reuse and fine-grained parallelism," in *26th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, 2018.
[99] P. Zhou, Z. Ruan, Z. Fang, J. Cong, M. Shand, and D. Roazen, "Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '18, 2018.
[100] "Kubernetes," 2018. [Online]. Available: https://kubernetes.io/
[101] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

Jason Cong received his B.S. degree in computer science from Peking University in 1985, his M.S. and Ph. D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1987 and 1990, respectively. Currently, he is a Distinguished Chancellor's Professor in the Computer Science Department (and the Electrical Engineering Department with joint appointment) of University of California, Los Angeles, also a Co-founder, the Chief Scientific Advisory and the Chairman of Falcon Computing Solutions. He was elected to the IEEE and ACM Fellows in 2000 and 2008, respectively, and the National Academy of Engineering in 2017.

Zhenman Fang received his joint B.S. degree in Software Engineering from Fudan University, China and Computer Science from University College Dublin, Ireland in 2009, and his Ph.D degree in Computer Science from Fudan University, China in 2014. From 2014 to 2017, Zhenman worked as postdoc at UCLA, in the Center for Domain-Specific Computing and Center for Future Architectures Research. Currently, Zhenman is a Staff Software Engineer at Xilinx, San Jose, and an Adjunct Professor in School of Engineering Science, Simon Fraser University (SFU), Canada. He will join SFU as a Tenure-Track Assistant Professor in Dec 2018. His research lies at the intersection of heterogeneous and energy-efficient computer architectures, big data workloads and systems, and system-level design automation. He is a member of the ACM and IEEE.



Muhuan Huang is a Software Engineer at Google. Her research interests include scalable system design, customized computing, and big data computing infrastructures. She received her PhD degree in computer science from UCLA in 2016.

Peng Wei is a Ph.D. student in the Computer Science Department at the University of California, Los Angeles (UCLA). He received his B.S. degree and M.S. degree in Computer Science from Peking University, China, in 2010 and 2013, respectively. Currently, he is working at the Center for Domain-Specific Computing of UCLA as a graduate student researcher. His research interests include heterogeneous cluster computing, high-level synthesis and computer architecture.

Di Wu is a staff software engineer at Falcon Computing Solutions, Inc., where he leads the engineering team for genomics data analysis acceleration. He received his Ph.D. from UCLA in 2017, under the advisory of Prof. Jason Cong. His research was focused on application acceleration and runtime system design for large-scale applications.

Cody Hao Yu received the B.S. and M.S. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2011 and 2013, respectively. Cody is pursing a PhD degree in University of California, Los Angeles. He also serves as a part-time software engineer at Falcon Computing Solutions at Los Angeles since 2016, and a summer intern at Google X in 2017. His research interests include the abstraction and optimization of heterogeneous accelerator compilation for datacenter workloads.