Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects

YI-HSIANG LAI, ECENUR USTUN, and SHAOJIE XIANG, Cornell University, USA
ZHENMAN FANG, Simon Fraser University, Canada
HONGBO RONG, Intel, USA
ZHIRU ZHANG, Cornell University, USA

FPGA-based accelerators are increasingly popular across a broad range of applications, because they offer massive parallelism, high energy efficiency, and great flexibility for customizations. However, difficulties in programming and integrating FPGAs have hindered their widespread adoption. Since the mid 2000s, there has been extensive research and development toward making FPGAs accessible to software-inclined developers, besides hardware specialists. Many programming models and automated synthesis tools, such as high-level synthesis, have been proposed to tackle this grand challenge. In this survey, we describe the progression and future prospects of the ongoing journey in significantly improving the software programmability of FPGAs. We first provide a taxonomy of the essential techniques for building a high-performance FPGA accelerator, which requires customizations of the compute engines, memory hierarchy, and data representations. We then summarize a rich spectrum of work on programming abstractions and optimizing compilers that provide different trade-offs between performance and productivity. Finally, we highlight several additional challenges and opportunities that deserve extra attention by the community to bring FPGA-based computing to the masses.

CCS Concepts: • **Hardware → Hardware-software codesign**; **Hardware accelerators**; **Reconfigurable logic applications**; • **Computer systems organization → Data flow architectures**; **High-level language architectures**; **Reconfigurable computing;**

Additional Key Words and Phrases: Field-programmable gate array, high-level synthesis, hardware acceleration, domain-specific language

ACM Reference format:
Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4, Article 17 (September 2021), 39 pages.
https://doi.org/10.1145/3469660

This work is supported in part by NSF/Intel CAPA Award No. 1723715, NSERC Discovery Grants No. RGPIN-2019-04613 and No. DGECR-2019-00120, and Canada Foundation for Innovation John R. Evans Leaders Fund.

Authors' addresses: Y.-H. Lai, E. Ustun, S. Xiang, and Z. Zhang, Cornell University, 320 Rhodes Hall, Ithaca, NY 14853, USA; emails: {yl2666, eu49, sx233, zhiruz}@cornell.edu; Z. Fang, Simon Fraser University, 8888 University Drive, Burnaby, BC V5A 1S6, Canada; email: zhenman@sfu.ca; H. Rong, Intel, 2200 Mission College Blvd, Santa Clara, CA 95054, USA; email: hongbo.rong@intel.com.

1936-7406/2021/09-ART17 $15.00
https://doi.org/10.1145/3469660

1 INTRODUCTION

FPGA-based accelerator design primarily concerns performance and energy efficiency. An FPGA programmer has the full freedom to (1) create deep custom pipelines that are composed of simple (often low-bitwidth) compute units instead of full-blown ALUs, (2) construct highly parallel and distributed control logic and on-chip storage, and (3) schedule dataflows in an explicit way to minimize off-chip memory accesses without using caches. This is in stark contrast with programming microprocessors (i.e., CPUs) and general-purpose graphic processing units (i.e., GPUs), where the underlying hardware architecture (instruction pipeline, memory hierarchy, etc.) is fixed, and the control flow of a software program drives the instruction-based execution on the hardware. In other words, FPGAs can be reconfigured/customized for a specific application or a domain of applications to exploit its massive fine-grained parallelism and on-chip bandwidth. This often leads to a higher compute throughput, lower energy consumption, and a more predictable latency, when compared to CPUs and GPUs.

Such great potential and flexibility do come at a substantial cost—the very low productivity of programming FPGAs to achieve high performance in the real world. Even for seemingly simple kernels (e.g., matrix matrix multiply, convolution, sparse matrix vector multiply), it is not uncommon for expert FPGA programmers to spend several months, or even more than one year, to build an accelerator that is both functional and performant on an actual device [213]. In a sense, the extreme flexibility of fine-grained programmable logic on an FPGA is both its greatest asset and its biggest liability. The end result is that FPGA-based acceleration is one of the most promising approaches to solving challenging problems across many domains in principle, but is within reach for only a few in practice, i.e., large enterprises that can afford teams of hardware and systems experts for high-value applications [26, 88].

Why is it so hard to program FPGAs? First, it requires a paradigm shift of thinking—most programmers are used to von Neumann machines and they tend to think sequentially, with parallelization added later as optimizations (at the level of threads, loops, etc.). However, an FPGA is a spatial architecture that features a massive amount of compute units and memory elements such as **look-up tables (LUTs)**, DSP slices, registers, block RAMs, and more recent ultra RAMs and **network-on-chip (NoC)**. These hardware resources are distributed over the fabric (usually two-dimensional) and run concurrently. Therefore, programmers have to think spatially and parallel in the first place for FPGA-based acceleration. Unlike CPUs, FPGAs typically do not have a predefined hardware cache hierarchy. To keep feeding data at a sufficient rate to the many parallel compute engines, programmers need to build user-managed buffers to maximally utilize both off- and on-chip memory bandwidth. This further adds to the programming complexity.

Second, conventional FPGA design tools mainly target hardware design experts instead of software programmers. It takes significant effort to manually create and optimize accelerator architectures using the traditional **register-transfer-level (RTL)** methodology. One must wrestle with low-level **hardware description language (HDL)** descriptions and **computer-aided design (CAD)** tools to implement a rich set of hardware customizations such as fixed-point arithmetic, pipelining, banked memories, and double buffering. Even worse, synthesizing an RTL design to a bitstream usually takes hours, or even days. This lengthy compile cycle makes **design space exploration (DSE)** on FPGAs prohibitively expensive.

Third, FPGA designs have poor debuggability. For instance, when a synthesized design runs into a deadlock on FPGAs, there is no easy way to stop the execution with a breakpoint and retrieve a snapshot of the states of the design. Usually, only CPU-based hardware emulation and cycle-accurate RTL simulation may help. The former does not model some important details of a real FPGA accelerator and may behave inconsistently with the actual on-device execution. The latter is too slow for a complex design, since the simulation is running at a very low level of design abstraction. Moreover, it is difficult to map the signal names in the final netlist back to variables in the original design, as the variable names are often mangled during RTL synthesis and technology mapping.

In short, it is an enormous challenge to achieve both high performance and high productivity for FPGA programming. While similar productivity-performance tension also exists for CPUs and GPUs, the problem is remarkably worse on FPGAs. As a result, *there is a dire need for new compilation/synthesis techniques and programming frameworks to enable productive design, exploration, generation, and debugging of FPGA-based accelerators based on high-level languages.* In the past 10–15 years, numerous research efforts have attempted to address this grand challenge of software-defined FPGA acceleration, and many exciting progresses have been made in both academia and industry.

Notably, recent years have seen promising development on **high-level synthesis (HLS)** for FPGAs [57]. This is evidenced by the wide availability of commercial C/OpenCL-based HLS compilers such as Xilinx Vivado/Vitis HLS [259], Intel SDK for OpenCL [122], and Microchip LegUp HLS [180]. The field of FPGA-based acceleration is also more vibrant than ever, as there is a growing number of HLS-synthesized designs that accelerate a plethora of emerging applications such as deep learning [247, 263, 270, 277], genomics [102, 116], graph analytics [30], and packet processing [118, 244].

HLS allows programmers who are not specialized in hardware to more easily create a functional FPGA design. However, one cannot expect the existing HLS compilers to automatically transform a vanilla software program into a high-performance accelerator, certainly not in a push-button manner. Instead, the original program often needs to be heavily optimized or restructured before the benefits of custom acceleration can be exploited. To this end, programmers and tools have to "collaborate," to carry out a set of optimizations to *customize* the target hardware for a given application. Performance-critical customizations must be implemented, no matter by the programmers or the tools. How much automation should be expected from the tools, and how much control should be given to the programmer, largely determine the design of the programming abstraction and directly impact the productivity of the programming process.

There are a number of recent efforts that survey the fundamental FPGA technologies, common hardware customization techniques, and HLS tools. Trimberger reviews how Moore's Law has driven the invention, expansion, and accumulation of FPGA technologies in the past three decades [237]. Kastner et al. [138] and Cong et al. [50, 52] describe some common set of optimization techniques for FPGAs using Vivado HLS from software developers' view. Licht et al. discuss a more comprehensive set of HLS optimizations for high-performance computing using both Vivado HLS and Intel OpenCL, including various techniques to enable pipelining, scaling, and efficient memory access [77]. Cong et al. [57], Nane et al. [191], and Numan et al. [193], respectively, survey the evolution of academic and commercial HLS tool development from 1980s to 2010, late 1990s to 2014, and 2000 to 2019.

In this article, we focus on the recent advances in the past 10 to 15 years on programming and synthesis for software-defined FPGA acceleration. Guided by the roofline model [255], we survey a prominent set of hardware customization techniques that systematically optimize the application-/domain-specific FPGA accelerator designs to achieve high performance and energy efficiency. We particularly focus on the techniques that are unique to custom accelerator designs, instead of the well-known code optimizations that are established for CPU or GPU targets. We also discuss various state-of-the-art synthesis techniques, programming abstractions, and representative tools that facilitate the automatic generation of customized accelerator architectures from software

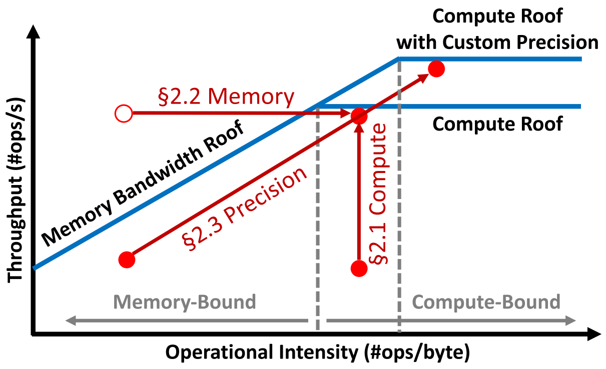


Fig. 1. Impact of different hardware customization techniques depicted in the roofline model—x-axis represents the operational density and y-axis represents the throughput; custom compute engines improve the throughput by concurrently executing more operations per second; custom memory hierarchy can move memory-bound design points toward a more compute-bound region by increasing data reuse and memory bandwidth utilization; custom data representations can benefit both custom compute engines and memory hierarchy, thus further lifting the compute roof.

programs, which is broader than previous surveys on HLS toolchains only [57, 73, 191, 193]. We further highlight several key challenges the community must address to enable the mainstream adoption of FPGA-based computing—the goal is to let domain experts, not just hardware specialists, produce efficient accelerators with fast compile-edit-run cycles.

2 ESSENTIAL CUSTOMIZATION TECHNIQUES FOR FPGA-BASED ACCELERATION

In this section, we focus on discussing various hardware customization techniques to address the performance challenge of FPGA programming. Our goal here is to provide the audience insights into *how to systematically design high-performance accelerators given the massive amount of customizable resources on FPGAs*. These hardware customizations can be done manually, automatically, or semi-automatically. To classify the hardware customizations in a systematic way, we leverage the widely used roofline model [255] that visualizes the performance bottlenecks and optimization directions.

As shown in Figure 1, in a roofline model, the x-axis represents the operational intensity (i.e., number of operations per byte of off-chip data access) while the y-axis represents the throughput (i.e., number of operations per second). We can plot different design points (in red) on the diagram according to the operational intensity and throughput. The highest achievable throughput of each design point is limited by the compute roof and memory bandwidth roof (blue lines). In the left part, the designs are memory-bound where the attainable throughput is limited by the off-chip memory bandwidth. In the right part, the designs are compute-bound, where the maximum throughput is determined by the amount of physically available computing resources (e.g., the maximum number of DSPs and LUTs on an FPGA). Similar to the terminology defined in Reference [255], here we use "operations" as a generic term to indicate the essential primitives that characterize the compute intensity of a given workload. Evidently, the meaning of this term is application specific. For example, multiply accumulate (MAC) operations are commonly used for DSP and deep learning applications. For FPGA-based acceleration, these operations may exploit customized data types, which we discuss in a later section.

We classify the hardware customization techniques into the following major categories (also shown in Figure 1):

1. **Custom Compute Engines**. For compute-bound designs, custom compute engines can be developed to move the accelerator throughput toward the compute roof. Inside such a compute engine, designers typically explore the following pipeline and parallelization techniques to execute more operations concurrently to improve its throughput [50, 53, 84]: (1) accelerator-unique fine-grained custom pipeline, which is often deeply pipelined and tightly coupled with fine-grained operator-level parallelization, different from CPUs and GPUs, (2) coarse-grained parallelization that further parallelizes multiple fine-grained pipelines, which can be both homogeneous (i.e., data parallelism) and heterogeneous (i.e., task parallelism) parallelization, similar to multicore processors, and/or (3) accelerator-unique coarse-grained pipeline that is composed of multiple fine-grained pipelines.
2. **Custom Memory Hierarchy**. For memory-bound designs, the accelerator memory hierarchy can be customized to move the design close to the (off-chip) memory bandwidth roof and move it right toward a compute-bound design. Different from CPUs/GPUs where their memory hierarchy is pre-designed and (almost) fixed, one unique opportunity (and challenge) for FPGAs is that their memory hierarchy is flexible and can be fully customized. Common optimizations include: (1) custom on-chip buffering and/or caching to improve data reuse and exploit much higher on-chip memory bandwidth, and (2) streaming optimization and/or custom network-on-chip to enable direct on-chip communication between multiple computing elements and/or bypass off-chip memory access.
3. **Custom Data Representations**. Finally, for both compute- and memory-bound applications, custom data representations with a reduced (or widened) bitwidth can play a vital and unique role in further improving the accelerator throughput. On the one hand, it reduces the bytes of off-chip data access required by computing operations and benefits the custom memory hierarchy. On the other hand, with reduced bit-width, a single fine-grained custom pipeline consumes much fewer resources and one can accommodate more such pipelines in the custom compute engine to achieve more coarse-grained parallelism and a higher throughput. Hence, with the custom precision, the compute roof is further moved up as the same FPGA can now run more operations.

In the following, we discuss more details of these hardware customization techniques and their interplay.

2.1 Custom Compute Engines

The primary objective of customizing the compute engines is to improve the throughput of the accelerator, i.e., moving upward in the roofline diagram (see Figure 1). To achieve this goal, an FPGA accelerator needs to make the best use of the on-chip resources to maximize hardware parallelism and compute efficiency by instantiating many **processing elements (PEs)** in parallel. In this article, we use "PE" as a generic term and loosely define it as a replicable hardware building block that executes a fine-grained loop (or function) pipeline with tens to hundreds operations. Similar to the notion of operations used in the roofline model, the exact function of a PE is highly application specific. To be more precise, we can break down *throughput* into three major factors, as shown in the following equation:

$$\begin{aligned} \textbf{\textit{Throughput}}\ (\text{OPs/sec}) = {} & \textbf{\textit{Hardware Parallelism}}\ (\text{max OPs/cycle}) \\ & \times \textbf{\textit{Compute Utilization}}\ (\text{busy OPs/max OPs}) \\ & \times \textbf{\textit{Clock Frequency}}\ (\text{cycles/sec}) \end{aligned} \tag{1}$$

$$\textit{Hardware Parallelism} = \textit{\#PEs} \times \textit{PE-Level Parallelism} \propto \textit{\#PEs} \times \frac{\textit{\#OPs per PE}}{\textit{II}}.$$

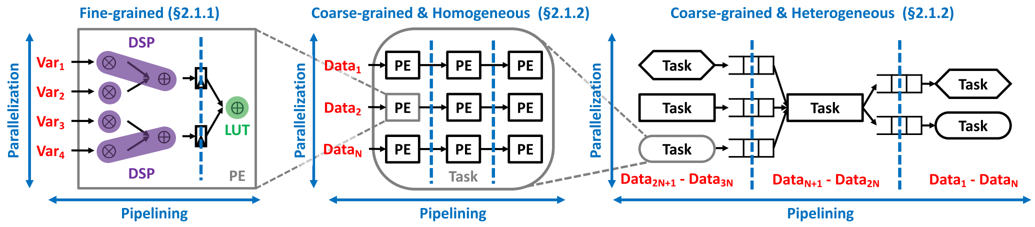


Fig. 2. Major forms of custom compute engines—At a fine granularity, each PE runs a custom pipeline with a low II. In addition, optimizations such as operation chaining (i.e., multiple operations scheduled into one cycle as combinational logic) can reduce the area and sometimes the II. We can then connect multiple PEs to build a coarse-grained heterogeneous or homogeneous pipeline (e.g., systolic arrays). Meanwhile, with optimizations such as PE duplication, we can achieve coarse-grained homogeneous parallelization (e.g., data-level parallelism). We can also build a coarse-grained heterogeneous pipeline using heterogeneous tasks composed of homogeneous PEs (e.g., task-level parallelism).

Here *hardware parallelism* represents the maximum number of operations that can be concurrently executed per cycle with the given accelerator architecture, which typically exploits both parallelization and pipelining. This term can further be decomposed into a product of coarse-grained parallel factor (i.e., the number of PEs) and fine-grained PE-level pipeline parallelism that is typically reflected by the pipeline initiation interval (II)—the smaller the II, the higher the pipeline throughput. *Compute utilization* is defined to be a ratio that captures the utilization of the physical compute resources, namely, the functional units that execute the operations defined in the software algorithm. These functional units should be kept as busy as possible (ideally near 100% utilization). *Clock frequency* determines the actual operating clock rate of the hardware circuits that run on the FPGA.

Unlike CPUs and GPUs where the clock rate is fixed and often at the order of GHz for a given device, the operating frequency of an FPGA accelerator is usually one order-of-magnitude lower and highly depends on the degree of pipelining and parallelization, as well as the resource usage of the underlying architecture. Hence, FPGA programmers must significantly improve the other two factors (i.e., hardware parallelism and compute utilization) and explore intricate design trade-offs amongst the three factors.

In the following, we introduce a set of optimization techniques for customizing the compute engines of an FPGA accelerator, which are classified by four dimensions. First, in terms of the parallelism form, we have *parallelization* and *pipelining*. Second, in terms of granularity, we have *fine-grained* and *coarse-grained* optimizations. We refer to the intra-PE parallelization/pipelining as fine-grained optimizations. In the HLS terminology, the scope of such optimizations is limited to a loop or function body where the inner loops, if any, are unrolled. We call the inter-PE optimizations as coarse-grained. Third, the composition of the parallel or pipelined PEs can be either *homogeneous* or *heterogeneous*. Finally, these PEs can be scheduled *statically* at compile time and/or *dynamically* at run time.

Figure 2 gives an overview of custom compute engines and examples of optimizations from the first three dimensions (i.e., parallelism, granularity, and composition). Starting with fine-grained optimizations (the left figure), the primary goal is to reduce the II of pipelines inside each PE. To achieve that, one can apply techniques such as modulo scheduling, operation chaining and multi-pumping, loop transformation, and dynamic scheduling. Moving forward, at a coarse granularity, we focus on inter-PE optimizations. Depending on the composition of PEs, different techniques are proposed. With homogeneous composition (the middle figure), we focus on data-level parallelism. For instance, one can perform PE duplication for parallelization and build systolic architectures for pipelining. With heterogeneous composition (the right figure), we focus on task-level parallelism, where we have dataflow pipelining and multithreading for parallelization.

2.1.1 Fine-grained Pipelining (and Parallelization). In contrast to general-purpose processors, FPGAs allow a programmer to build a deep application-specific pipeline that obviates the need for instruction decoding, branch prediction, and out-of-order logic, which incur nontrivial overhead in both performance and energy. In this section, we discuss the common techniques to improve the PE-level throughput through fine-grained pipelining, parallelization, as well as some of the associated trade-offs between resource efficiency and frequency.

Modulo Scheduling—Pipelining, especially loop pipelining, is one of the most important optimizations in HLS, because it allows successive loop iterations to be overlapped during execution (the same technique also applies to overlapping the function invocations). Typically enabled by modulo scheduling [211], loop pipelining creates a schedule for a single loop iteration so that the same schedule can be repeated at a constant **initiation interval (II)**. Minimizing the II is considered the foremost objective of pipeline scheduling. By reducing the II of a loop pipeline, we improve the throughput by increasing the hardware parallelism, as shown in Equation (1). While II determines the amount of parallelism, it is inherently limited by two major factors (1) inter-iteration dependence (i.e., recurrence) between operations in different loop iterations, and (2) the available amount of resources (e.g., memory bandwidth, limited DSPs). In fact, modulo scheduling is an NP-Hard problem in the presence of both recurrence and resource constraints [211].

There exist a rich set of heuristics that tackle the problem by making different trade-offs in solution quality and compile time. Several state-of-the-art HLS tools employ the more versatile heuristic based on a **system of integer difference constraints (SDC)**, which can naturally handle various hardware-specific design constraints [22, 62, 68, 274]. SDC-based modulo scheduling is rooted in a linear programming formulation and can globally optimize over constraints that can be represented in the integer difference form, including both intra- and inter-iteration dependencies. Notably, however, resource constraints can only be converted to the difference constraints in a heuristic way. To address this limitation, Dai et al. [68] recently propose to couple SDC with Boolean **satisfiability (SAT)** so that both timing and resource constraints can be exactly modeled in a practically scalable solution to modulo scheduling.

Operation Chaining and Multi-Pumping—The pipeline efficiency can be further improved by several operation-level optimizations that exploit the unique characteristics of key FPGA building blocks such as LUTs and DSP units. Operation chaining is one example, where a cluster of dependent operations can be scheduled combinationally and mapped to just a few LUTs or DSPs [267, 274]. As shown in Figure 3(b), chaining can reduce II by shortening the latency on the critical dependence cycle. More chaining opportunities can be exposed when the effects of the downstream technology mapping are also taken into consideration during scheduling. For example, MAPS [233] shows that mapping-aware scheduling can chain more operations in a way to shorten latency and significantly reduce the LUT usage for logic-operation-intensive applications. This approach is extended to support modulo scheduling in Reference [278]. Aggressive operation chaining can reduce area and allow more PEs to be allocated, although it may increase the critical path delay of the resulting circuit. Hence, when using this optimization, an FPGA compiler (or programmer) must balance the trade-off between the hardware parallelism and clock frequency.

Multi-pumping [21, 24] is another FPGA-unique optimization, which can be applied to sidestep the resource constraint imposed by the limited number of on-chip DSP units (especially on embedded FPGAs). It is important to note that the hardened DSP units on commercial FPGAs can run at a much higher frequency (typically 2×) than the surrounding logic blocks. Therefore, it is possible to map two multiply operations to a single DSP unit in one cycle, where each multiply completes

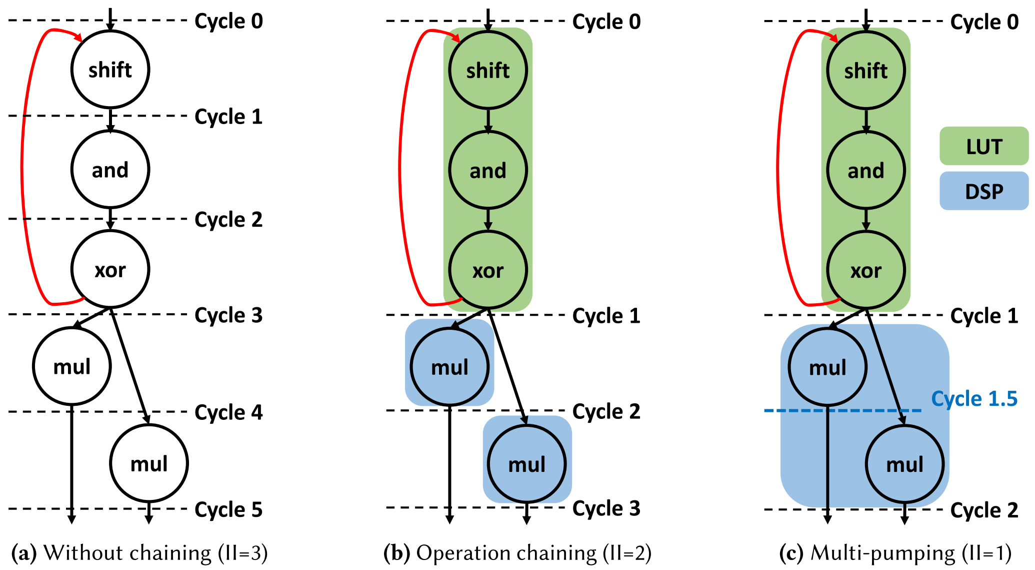


Fig. 3. Examples of FPGA-specific fine-grained scheduling optimizations—The red arrow denotes an inter-iteration dependence in the control-dataflow graph. Here, we assume only one DSP unit is available. (a) Scheduling without chaining, where the II is bottlenecked by the recurrence; (b) operation chaining, where the logical operations `shift`, and, and xor are chained and mapped into a single-level of LUTs, which reduces the delay on the recurrence. Thus, the II is now limited by the resource constraint, i.e., two `mul` operations but only one DSP; (c) multi-pumping, where we map two `mul` operations into a single DSP operating at a 2× higher frequency. By resolving the the resource constraint, we achieve II = 1.

within half of the system clock cycle. This optimization is illustrated in Figure 3(c). Multi-pumping can reduce pipeline II when the limited number of DSPs becomes the resource constraint. Additionally, it can reduce the DSP usage for a single PE and thus increase the hardware parallelism.

Loop Transformations that Improve Pipelining—Modulo scheduling typically does not alter the underlying control data flow graph of the loop subject to pipelining. However, in many cases, the inter-iteration dependencies can be removed (or alleviated) through code transformations in multi-dimensional iteration space. A helpful tutorial is given in Reference [77], which summarizes a number of loop transformations that resolve the II-limiting recurrences through reordering and interleaving nested loops.[1] Polyhedral compilation is also increasingly used to improve the efficiency of pipelining [15, 155, 161, 188, 207, 283].

To further increase the hardware parallelism, HLS designs commonly use loop unrolling in combination with pipelining to increase the number of parallel operations per pipeline. Unlike unrolling on CPUs, the unfolded copies of the loop body require additional hardware on FPGAs. Nevertheless, an unrolled loop body is usually smaller than the original version, as unrolling often enables additional code optimizations such as constant propagation. Some loop transformations are also useful for increasing the utilization of the pipeline by minimizing "bubbles" due to the filling/draining of the pipeline. For example, loop flattening (also known as loop coalescing) can be applied to coalesce a nested loop into a single-level flattened loop so that the pipeline continuously executes the innermost loop without much frequent switching to an outer loop.

[1]The same paper [77] also includes a comprehensive table that summarizes the commonalities and differences between traditional CPU-oriented code transformations and their counterparts in HLS. Hence, we do not repeat those discussions here.

Dynamic Scheduling—Certain data dependencies or resource contentions may occur in an input-dependent manner, which are known as data or control hazards in the terminology of a processor pipeline. These hazards may result in poor performance with the mainstream HLS tools, as they cannot be resolved statically with compile-time analysis and transformations. There are ongoing efforts to improve statically scheduled HLS to better handle these hazards [7, 69, 78, 159]. For example, Dai et al. [71] propose to insert application-specific hazard detection units into the pipeline generated by a modulo scheduler. Variable-latency operations can also be handled in a cost-effective way through pipeline flushing [70]. Another active line of research proposes to generate dataflow circuits to enable dynamically scheduled pipeline that yields better-than-worst-case performance in the presence of dynamic data dependencies or variable-latency operations [81, 117, 130–132]. However, extensive fine-grained handshaking is required to implement dynamic scheduling, which often results in nontrivial overhead in both resource usage and clock frequency. To address this challenge, a hybrid scheduling method is proposed, which partitions the input program into statically and dynamically scheduled portions to improve efficiency [34].

2.1.2 Coarse-grained Pipelining and Parallelization. The coarse-grained optimizations involve inter-PE parallelization and pipelining [59], where the composition of the PEs can be either homogeneous or heterogeneous. Compared to CPUs/GPUs, FPGAs have additional flexibility in customizing the shape and topology of the parallel PE array and the synchronization mechanism between PEs.

Inter-PE Parallelization—For computations that can be executed independently, one can allocate multiple PEs to execute them in a data-parallel fashion. This method is also known as PE duplication or compute unit replication. There are many ways to perform PE duplication. One example is unrolling the outer loop [50]. After loop unrolling, each unrolled inner-loop body becomes a PE that can be executed in parallel. A more automated approach is implemented in the Fleet framework [235], which automatically generates massively parallel PEs along with a high-performance memory controller. Many efforts realize homogeneous parallelization by leveraging useful features in the input language [37, 198]. For instance, FCUDA [198] is a compilation flow that maps CUDA code to FPGAs. This allows programmers to use the high-level CUDA APIs to describe coarse-grain data-level parallelism, where each thread-block inside a CUDA kernel becomes a hardware PE that can be executed in parallel with other PEs.

Software multithreading constructs can be used to explicitly specify coarse-grained parallelism for multiple (often heterogeneous) PEs. To this end, Hsiao et al. [113] propose a technique called thread weaving, which statically schedules computation in each thread with the awareness of other threads, and guarantees that the shared resources are accessed in a time-multiplexed and non-interfering manner. There are also approaches that resolve synchronization dynamically. FlexArch [28] is one example, which schedules the computations dynamically via work stealing [12] and assigns them to customized PEs. With work stealing, the computations can be more efficiently distributed and balanced across PEs.

Inter-PE Dataflow Pipelining—FPGAs programmers can also compose PEs (or tasks) into a coarse-grained dataflow pipeline that is typically dynamically scheduled. By overlapping the executions of different PEs and passing data between them with a streaming interface or Ping-Pong buffers, one can overlap the PE execution and decouple the compute from memory accesses to hide the off-chip memory access latency (more details in Section 2.2). Such dataflow pipelining is well-suited for many application domains, such as image processing [35, 208, 249], machine learning [226, 254], graph processing [67, 142, 194, 268, 281], and network switching [199]. Current HLS tools typically provide (vendor-specific) pragmas to realize dataflow pipelines of different forms. In general, the dataflow architecture is more scalable if the PEs are connected by asynchronous

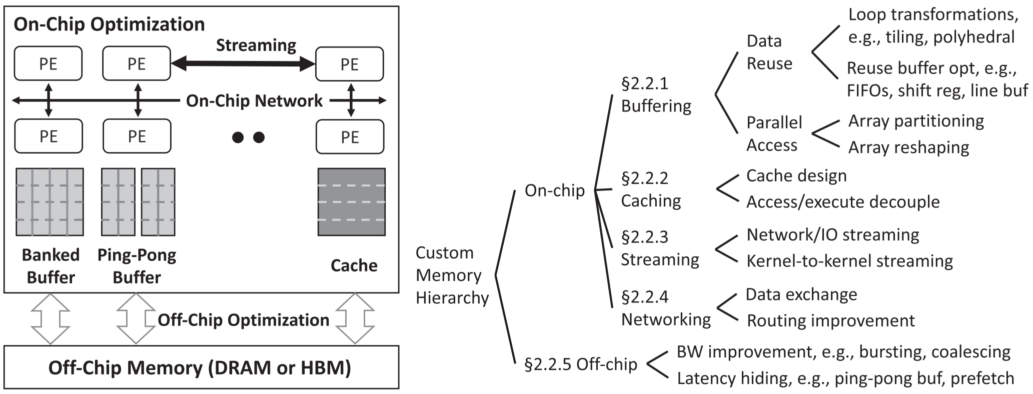


Fig. 4. Overview of customizations for custom memory hierarchy.

channels, but at the cost of additional handshaking logic between PEs; the pipeline is more efficient but perhaps less scalable (in terms of timing closure) if its stages are connected by wires or registers pulsed by the same clock.

Similar to coarse-grained parallelization, a dataflow pipeline can be composed either heterogeneously or homogeneously. To construct heterogeneous pipelines, an example is the TAPA framework [36], which defines a programming interface to describe the dataflow parallelism within an application. The ElasticFlow architecture [234] is another example that implements the loop body as a multi-stage pipeline. There also exist some application-specific dataflow architectures [35, 108, 112, 214]. For instance, SODA [35] is a framework that generates a dataflow architecture for stencil computation, where data elements are updated with some fixed patterns. With the SODA framework, computations from different pipeline stages can be executed simultaneously with designated forwarding units, which also serve as reuse buffers.

Systolic arrays represent a well-known class of homogeneous pipelines, where connected PEs work rhythmically (shown in Figure 2). For each time step, each PE reads from its neighbors, processes the data, and forwards the results to its neighbors. Systolic arrays are also considered as a generalization of one-dimensional pipelining. There is a long line of research that focuses on generating high-performance systolic arrays [16, 58, 58, 106, 151, 217, 229, 245, 254].

2.2 Custom Memory Hierarchy

As shown by the roofline model in Figure 1, the key for memory hierarchy customization is to reduce the number of off-chip data accesses required by computing operations, i.e., increase the operational intensity. First, one can explore on-chip buffering and caching techniques to improve the on-chip data reuse. This is also often a key enabler of low-latency and high-throughput custom compute engines. Second, one can explore streaming optimization and custom on-chip networks to bypass off-chip memory access and enable fast communication between multiple compute engines. Finally, for the essential off-chip memory accesses that cannot be avoided, one can explore data access reorganization techniques to fully utilize the off-chip memory bandwidth and prefetching techniques to hide the off-chip memory access latency. An overview of these memory customization techniques is illustrated in Figure 4.

2.2.1 On-chip Buffer Optimization. Unlike general-purpose processors that have a fixed and pre-designed multi-level cache system, FPGAs provide a fully customizable on-chip memory hierarchy that can harvest the bandwidth of a massive amount of distributed registers, LUT RAMs,

block RAMs (BRAMs), and **ultra RAMs (URAMs)**. One of the most common and effective optimizations is to customize the on-chip buffers based on the application-specific memory access behavior. The FPGA programmers may use different types of reuse buffers such as (1) shift registers, line buffer, and window buffer that are predefined by HLS vendors [259], and (2) user-customized buffer structures. For both vendor-provided and user-customized reuse buffers, programmers often need to perform various loop transformations.

To minimize the required on-chip memory usage, an important technique is to apply loop tiling and carefully select the right tile size to balance the computation and memory access with the minimum buffer size. Loop fusion can also be leveraged to reuse one buffer for multiple loops. Besides these commonly used loop transformations, several accelerator-specific optimizations can also be applied. For example, one can use shift register, line buffer, and/or window buffer to avoid duplicate buffers required by multiple compute operations. The line buffer and/or window buffer is a more generalized version of a shift register, where it buffers just enough elements on-chip for data reuse as required by the computing engine and it shifts at every new iteration/cycle. As a result, only a minimum amount of new data has to be read from the off-chip memory into the buffer every clock cycle. One can also use small streaming FIFOs (first in, first out) to dataflow between multiple computing engines and avoid large on-chip buffers, which we discuss in more detail in Section 2.2.3.

Another unique optimization for FPGA accelerators is to fully utilize the heterogeneous memory resources to increase parallel on-chip data accesses so that the compute units are not idling due to lack of data, namely, higher compute utilization in Equation (1). Such on-chip data are typically large and require distributed BRAMs and URAMs, which are composed of hundreds to thousands of dual-port SRAM banks. Each physical memory port allows up to one read or one write—with no more than 36 bits for BRAM and no more than 72 bits for URAM—in one cycle. To enable parallel on-chip data access, the key is to apply the *memory banking* optimization in HLS to partition a large on-chip buffer into multiple small buffers that can be mapped onto multiple BRAM or URAM banks. When each data item has fewer than 36 bits for BRAM or 72 bits for URAM, *on-chip memory coalescing* can be further applied to stack partitioned smaller arrays to increase the port access bitwidth (or word width) of the BRAM and URAM banks.

Several FPGA HLS tools often provide directives for users to specify such array partitioning and reshaping to realize these optimizations [122, 259]. For example, Xilinx Vivado HLS [259] provides pragmas for users to partition an array at multiple dimensions, in a block, cyclic, or complete fashion. Recent years have also seen an active body of research on automatic array partitioning. For array accesses with affine indices based on the loop indices, several studies [48, 55, 250, 251] find that polyhedral compiler analyses can statically find array partitioning solutions to avoid on-chip memory bank conflicts and automate the partitioning process. In general, the polyhedral compilation is shown to be effective in co-optimizing the loop pipelining, parallelization, data reuse, and array partitioning together given a polyhedral program [207, 283]. For stencil applications, prior efforts have demonstrated success with polyhedral analyses [35, 56], where they can find the optimal reuse buffer setting and partition the buffer in a way to minimize both off-chip memory accesses and the on-chip buffer size. For non-affine programs, a few studies have taken a profiling-driven approach to automatically partition the arrays with trace-based address mining [32, 282].

2.2.2 On-chip Cache Optimization. Most of the special-purpose accelerators target applications with regular memory accesses using the aforementioned on-chip buffering optimizations. On-chip caching provides another attractive alternative to accelerate applications with memory access patterns that are hard to predict statically; it also eliminates the tedious programming effort to explicitly manage the on-chip buffering.

As one of the early studies, the work in Reference [38] evaluates the multi-port L1 cache design for parallel accelerators and finds the performance highly depends on the cache architecture and interface. LEAP-scratchpad [4] is another early effort to provide multi-level cache support for FPGA accelerators. It abstracts away the memory system from accelerator designers. Another effort similar to (but different from) the cache abstraction is the connected RAM (CoRAM) approach [47]. CoRAM virtualizes a shared, scalable, and portable buffer-based memory architecture for the computing engines, which naturally comes at the cost of performance and resource overhead.

On several FPGA SoC devices from Intel and Xilinx, a shared coherent cache is available between the hardened ARM CPU and the FPGA accelerators [123, 260]. For datacenter FPGAs, efforts such as Intel Xeon-FPGA multi-chip package and IBM CAPI also provide coherent shared cache memory support for Xeon/PowerPC CPUs and FPGAs. A quantitative evaluation of modern CPU-FPGA platforms with and without coherency support can be found in References [43, 44]. However, such cache designs are shared by the CPU and FPGA; there are no practically available, dedicated on-chip cache for the FPGA accelerators themselves yet. It is a potential area for more research.

2.2.3 Streaming Optimization. Besides on-chip buffering and caching optimizations, a unique optimization for FPGA accelerators is streaming, which is high-performance and resource-efficient as it helps minimize both off- and on-chip memory accesses. To enable streaming, the data access pattern needs to be in a strict sequential order, with each data item read/written only once (plus read and write cannot be mixed). If there are data reuse opportunities, then the streaming optimization needs to be combined with the aforementioned buffering and/or caching optimizations. HLS vendors typically provide predefined data types and structures, as well as directives, for users to specify streaming access [122, 259].

Streaming optimizations have been widely used in image/video processing and machine learning applications. It is also crucial for network processing applications [258] to directly stream the data from the **network interface controller (NIC)**. For example, the P4 HLS programming language [258] is developed to enable fast and efficient network processing with streaming support. More recently, both Xilinx and Intel provide support for direct host-to-accelerator streaming and accelerator-to-accelerator streaming to bypass the off-chip memory and on-chip memory access [122, 259]. Some initial efforts have attempted to provide compiler support to identify and automate the kernel-to-kernel streaming for Intel OpenCL programs on FPGAs [129, 160].

2.2.4 On-chip Communication Networks. Building an efficient on-chip communication network is also important for minimizing the overhead of accessing off- and on-chip memories. **Network-on-chip (NoC)** is a popular option in this regard as it can further ease the routing and improve the operating frequency of the FPGA accelerators. One of the early survey papers for FPGA NoC [2] compares different hard, soft, and mixed NoC approaches, demonstrates the effectiveness of NoCs and points out the need for hard NoCs. Indeed, the recent Xilinx Versal architecture has included a hardened NoC [231]. The Hoplite series [89, 134, 135, 252] have explored various optimizations for soft overlay NoCs on FPGAs to minimize its resource overhead, improve deflection and priority-aware routing, and provide stall-free buffers. Several studies [31, 126, 279] also integrate soft NoC generation for HLS accelerators. We refer the interested readers to these papers for more details. The FPGA products with a hard NoC will trigger more research in the programming tool support to efficiently place-and-route computing elements on FPGAs and effectively leverage the high-speed NoC.

2.2.5 Off-chip Memory Optimization. After reviewing the aforementioned optimizations that aim to reduce the off-chip data transfers, we further discuss how to optimize the essential off-chip memory accesses that cannot be avoided easily. Here, we mainly focus on the **direct memory access (DMA)** from a DDR or **high-bandwidth memory (HBM)**. For the communication optimization between the host program and the FPGA accelerators, we refer the interested readers to References [43, 44] for more details.

There are two major types of off-chip memory optimizations. The first type attempts to hide the off-chip memory access latency from the compute engines. One of the common techniques is to use the double (or ping-pong) buffer technique [50, 53], which decouples the memory access (read or write) from execution via coarse-grained pipelining. Several HLS tools have automated the generation of the double buffers. Another optimization is to prefetch the data to the on-chip buffer/cache [29, 104].

The second type of optimization aims to fully utilize the off-chip memory **bandwidth (BW)**, which is nontrivial for FPGAs. A recent study in Reference [167] characterizes the off-chip DRAM and HBM bandwidth for HLS programs under a comprehensive set of factors, including (1) the number of concurrent memory access ports, (2) the data width of each port, (3) the maximum burst access length for each port, and (4) the size of consecutive data accesses. To fully utilize the off-chip memory bandwidth, programmers have to carefully tune these parameters based on the insights summarized in Reference [167]. Some common optimizations include *memory bursting* to increase the size of consecutive data access and the *off-chip memory coalescing* to increase the data access bitwidth. The Falcon Merlin compiler [54, 60] has partially automated some of these optimizations. The recent HBM Connect work [39] further proposes a fully customized HBM crossbar to better utilize HBM bandwidth.

2.3 Custom Data Representations

Exploiting custom data representation is a vital optimization to achieve efficient hardware acceleration on FPGAs. In contrast to CPUs/GPUs, which employ arithmetic units with a fixed bitwidth, FPGAs allow programmers to use customized numeric types with the precision tailored for the given application. When properly leveraged, such flexibility can lead to substantially improved efficiency in both custom compute engines and the custom memory hierarchy. In this section, we discuss two major aspects of custom data representations: (1) parameterized numeric types and (2) automatic bitwidth analysis and optimization techniques.

Parameterized Numeric Types—Most FPGA HLS tools support arbitrary-precision integer and fixed-point data types. For example, Xilinx Vivado HLS provides `ap_(u)int` and `ap_(u)fixed` classes in C++ [259], while Intel HLS compiler uses `ac_(u)int` and `ac_(u)fixed` [122], which were originally developed by Mentor.

Using arbitrary-precision integer types, multiple low-bitwidth data elements can be packed together into a wide bit vector without increasing the footprint on the main memory. The packed data can then be read/written in a single memory transaction, which greatly improves the bandwidth utilization and the overall operational intensity of the accelerator. In addition, operations with reduced bitwidth require fewer resources, and thus more PEs can potentially be allocated on the same FPGA device to increase hardware parallelism. As a result, the compute roof is further lifted up, as illustrated in Figure 1. For some applications, one can pack hundreds or thousands of bits into a single integer and perform bitwise operations very efficiently using the distributed LUTs and registers on the FPGA fabric. One example is calculating the hamming distance between two wide binary vectors via bitwise XORs. This kernel is used in many domains such as telecommunication, cryptography, and machine learning (e.g., binary neural networks [90, 103, 157, 262, 273], hyper-dimensional computing [119, 120, 215]).

Parameterized fixed-point types are also extensively used in FPGA design [99, 171, 227]. Fixed-point values are essentially integers with a predetermined position for the binary point. Their range is determined by the number of bits to the left of the binary point, while the precision depends on those to the right of it. Unlike floating-point units, fixed-point arithmetic units do not require expensive logic to manipulate the mantissa and exponent through rounding and normalization. Hence, on an FPGA, fixed-point operations typically have a shorter latency and consume much fewer resources than their float-point counterparts.

In some cases, fixed-point types may cause nontrivial accuracy degradation if the represented data have a wide dynamic range. Hence, efficient floating-point computation is desired. To this end, some recent FPGA devices (e.g., Intel Arria 10) offer hardened **floating-point units (FPU)**, which obviate the need to perform the aggressive fixed-point conversion for an accuracy-sensitive application. Besides relying on FPUs that are strictly compliant with the IEEE 754 standard, the FPGA programmers can also leverage several existing libraries and tools that generate custom FPUs with reduced bitwidth [9, 127, 248]. For instance, FloPoCo is an open-source C++ framework that can generate customized FPUs in synthesizable VHDL [75].

There is an active line of research that explores new floating-point formats to accelerate machine learning workloads. **Brain floating-point (bfloat)** (originally proposed by Google) [246] is a truncated version of the IEEE single-precision floating-point format, which is now supported by the Intel Agilex FPGAs [121]. In addition, multiple recent efforts have implemented Posit arithmetic operators on FPGAs [25, 228]. Most recently, Microsoft has proposed MSFP [74], which is a form of the block floating-point format, where the data in an entire tensor block share a single exponent. Hardened MSFP units have been added to the latest Intel Stratix 10 NX FPGAs [181].

Automatic Bitwidth Optimization—For an FPGA accelerator, the bitwidth settings of the numeric types can be a major determinant of its resource usage, the achievable frequency, and hence the throughput. It often requires a nontrivial amount of manual effort to quantize floating-point values into fixed-point types with the appropriate bitwidth. Hence, there has been a large body of research that attempts to automate the float-to-fixed conversion process. With the existing approaches, *range analysis* is first performed (typically by a compiler analysis pass or a profiler) to obtain the minimum and maximum values of each variable in a given application. Note that such range analysis is also useful for reducing the bitwidth of the integer values. Afterward, *bitwidth allocation* is carried out to quantize or downsize the variables to a certain bitwidth. Finally, the resulting accuracy and other performance/area metrics need to be evaluated through estimation or simulation.

There are two popular methods to perform range analysis. The first method is to statically analyze a program that exploits the information on compile-time constants (e.g., loop bounds) and additional user-provided hints (often through pragmas or attributes) [13, 14, 136, 144, 195, 240]. The second one is to determine the input-dependent value ranges at run time [95, 146]. Static analysis often relies on interval arithmetic [109] and affine arithmetic [76] to infer the bound of each variable. In contrast, dynamic analysis can achieve additional reductions in bitwidth, although it may also introduce errors when the input samples do not cover some of the outliers. There are also hybrid approaches that attempt to combine compile- and run-time methods. For instance, Klimovic and Anderson propose to first perform static analysis according to the common-case inputs suggested by the users while leveraging a run-time mechanism to fall back to software execution when outliers occur [146].

For bitwidth allocation, prior arts mostly adopt methods such as simulated annealing and satisfiability modulo theory [136, 144, 153, 195, 240]. It is worth noting that both range analysis and bitwidth allocation are computationally hard problems and can be very time consuming to solve.

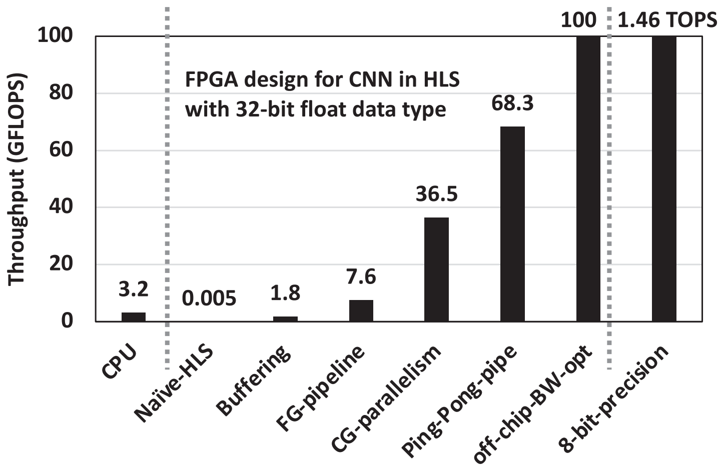


Fig. 5. Step-by-step performance breakdown of our Caffeine FPGA accelerator [271]—Here, we accelerate a VGG16 CNN model using major HLS optimization techniques discussed in Sections 2.1–2.3; the default data type is 32-bit floating-point before we apply the 8-bit fixed-point optimization and FPGA platform is a medium-size Alpha Data PCIe-7v3 FPGA board [271].

To address this challenge, Kapre and Ye propose a GPU-accelerated tool flow to automate and speed up the bitwidth optimization process for the FPGA-targeted HLS designs [136].

2.4 Case Study on a CNN Accelerator

To demonstrate the performance impact of the hardware customization techniques discussed in Section 2, we perform a case study on the acceleration of deep **convolutional neural networks (CNNs)**. This study is based on the Caffeine FPGA accelerator [271] for the VGG16 convolution layer using Xilinx Vivado HLS. Here, we mainly focus on demonstrating the performance impact of the aforementioned common customizations, including (1) on-chip buffering (both reuse improvement and banking) of the weights, input and output feature maps, (2) fine-grained pipeline and parallelism of the matrix multiplication (denoted as FG-pipeline), (3) coarse-grained parallelism to duplicate the pipeline (denoted as CG-parallelism), (4) ping-pong buffer to hide the off-chip access latency (denoted as Ping-Pong-pipe), (5) off-chip memory bandwidth optimization using bursting and coalescing (denoted as off-chip-BW-opt), and finally (6) 8-bit custom precision to quantize the convolutional layer. The target hardware platform is a medium-size Alpha Data PCIe-7v3 FPGA board. We refer the interested readers to the original Caffeine paper [271] for a more detailed accelerator design and experimental setup.

Figure 5 presents the step-by-step performance breakdown of those optimizations. Without any optimizations, the naïve HLS code is hundreds of times slower than the CPU version. In particular, on-chip buffering is the key technique that allows the custom compute engines to have sufficient data. It brings the FPGA design closer to the CPU performance. Note that the buffering result presented here actually includes some of the fine-grained pipeline optimizations, since Xilinx HLS tool automatically pipelines the innermost loops. With optimized fine-grained parallelism, the design starts to achieve a 2.4× speedup over the CPU. By further exploring the coarse-grained parallelism, it achieves another 4.8× speedup. Double buffering achieves another 1.9× speedup while off-chip memory bandwidth optimization provides an additional 1.5× speedup. Using custom data representations with 8-bit fixed-point types boosts the accelerator performance by another 14.6× speedup.

3 REPRESENTATIVE PROGRAMMING ABSTRACTIONS AND COMPILERS

There is a wide spectrum of programming models and compilers for FPGAs, such as HLS, polyhedral compilers, domain-specific languages, and new accelerator design languages. Due to the large body of existing work, we can only survey a (small) subset of the representative and more recent efforts. When describing a language/compiler, we focus on its productive features (e.g., metaprogramming, parametric types), performance features (e.g., supported hardware customizations), generality, and portability across FPGAs and other hardware targets.

3.1 High-level Synthesis (HLS)

Compared with HDL, HLS is a jump in productivity of programming FPGAs, at some reasonable cost of performance. HLS tools provide programmers high-level abstractions and derive efficient RTL from them, saving programmers from extensive hand-coding and tuning using low-level HDLs [57]. Consequently, HLS tools have been increasingly deployed for FPGA-based accelerations.

There is a large body of HLS tools. Below, we briefly review them from the aspects of input languages, programming models, and compiler transformations. For more comprehensive details on HLS, one may read surveys specifically on this topic [57, 73, 191, 193]. From the perspective of input languages, there are HLS tools based on C/C++ and other languages. From the perspective of programming models, HLS tools can be classified as control- or data-flow. From the perspective of compiler transformations, HLS tools can also be classified as purely pragma-driven or automatic polyhedral compilation.

C-based and Non-C-based HLS—State-of-the-art HLS compilers from major FPGA vendors allow a computation for acceleration to be described in C/C++, which is then synthesized into an FPGA accelerator either in the form of RTL or directly in bitstream by invoking downstream CAD tools. Representative commercial tools include Intel OpenCL SDK [66], Intel HLS [122], Intel OneAPI [124], Xilinx triSYCL [140], Microchip LegUp [23, 179], Mentor Catapult HLS [178], Cadence Stratus HLS [19], Bambu [204], GAUT [64], and Xilinx Vivado/Vitis HLS [57, 259]. The C-based HLS languages and compilers are easily accessible to programmers who are familiar with CPU programming. The same design can execute on both CPUs and FPGAs, and the HLS tools further provide basic performance portability across different target FPGA devices (from the same vendor). However, achieving high-performance with HLS requires nontrivial hardware knowledge and is very different from the conventional performance tuning process of CPU software programming. Programmers typically need to apply many vendor-specific pragmas that direct an HLS tool to generate the desired accelerator architecture. Also, programmers often have to significantly restructure their code to manually implement a number of customizations described in Section 2.

As FPGAs become increasingly common, FPGAs have also been added as a target of other popular languages, including Python and Java. These languages generate either C-based HLS code or HDL code. For example, Hot & Spicy [225] translates Python code, which is written in a subset of Python syntax and annotated with types and pragmas, into synthesizable Xilinx Vivado HLS code; Synthesijer [232] maps Java into VHDL and Verilog, while Juniper [96, 133] maps Java to Xilinx Vivado HLS code.

Control- and Data-flow HLS—Mainstream HLS tools are based on control-flow (i.e., imperative) languages. In an untimed sequential program, the portion to be accelerated can be synthesized into fully timed HDL modules realizing the same functionality on FPGAs. Often, programmers can add pragmas/annotations in the accelerated portion to expose parallelism. HLS tools fall into this category include Xilinx Vivado/Vitis HLS [57, 259], Intel OpenCL SDK [66], Microchip LegUp [179], Mentor Catapult HLS [178], Intel oneAPI [124], Xilinx triSYCL [140], Bambu [204], Cadence Stratus HLS [19], and GAUT [64].

FPGA is naturally fit for dataflow execution due to its massive distributed hardware resources. A dataflow HLS program expresses a dataflow graph. For example, Maxeler [174], CAPH [218], and OpenDF [11] use meta-languages (e.g., CAL, MaxJ) to define a graph structure, and synthesize the nodes into individual RTL modules that are connected with communication channels. Other dataflow HLS tools (e.g., FAST-LARA [97] and OXiGen [203]) provide a high-level abstraction (e.g., C/C++ structs) to represent the dataflow structure.

Pragma-Driven and Automatic Polyhedral Compilation—Existing HLS tools commonly allow programmers to manually insert pragmas that direct the compiler to transform loops for performance. Recent years have also seen an active body of research on FPGA HLS that builds on polyhedral compiler frameworks to perform many useful loop transformations in a fully automated fashion [8, 58, 59, 161, 188, 207, 245, 283].

Polyhedral compilation is a powerful approach to analyzing and optimizing loop nests that are **static control parts (SCoP)** [10, 15, 49, 98, 161, 163]. SCoP is a subclass of general loop nests with constant strides, affine bounds, and statically predictable conditionals that are affine inequalities of the associated loop iterators. Such a restricted form of loop nests is commonly seen in a broad range of numerical computations such as dense linear or tensor algebra, image processing, and deep learning algorithms. A polyhedral compiler uses parametric polyhedra as an **intermediate representation (IR)**. The polyhedra represents (either perfectly or imperfectly) nested loops and their data dependencies. Such an IR enables many useful dataflow analyses and effective compositions of a sequence of loop transformations. Polyhedral analyses and code transformations have been extensively used in optimizing compilers targeting CPUs and GPUs, especially for high-performance computing.

For FPGA-targeted HLS, loop pipelining is a critical performance optimization but usually is applicable only to loops with statically known dependencies. Liu et al. [161] extend loop pipelining so that uncertain dependencies (i.e., parameterized by an undetermined variable) and non-uniform dependencies (i.e., dependence distance varying between iterations) can also be handled. They build a polyhedral model to characterize loop-carried dependencies, statically schedule the loops without the dependencies, and insert a controller to dynamically respect the dependencies during execution, if needed. Pouchet et al. [207] leverage the expressiveness of polyhedral compilation to optimize critical resources, including off-chip memory bandwidth and on-chip scratchpad capacity, exploit data reuse, and promote memory references to on-chip buffers automatically for minimizing communication volumes. Zuo et al. [283] map optimized polyhedral IR to C for the purpose of HLS and FPGA so that FPGA resources are minimized without a performance penalty. PolySA/AutoSA [58, 245] automatically builds systolic arrays from a plain C/C++ code without sophisticated loop annotations or manual code transformations. The framework compiles the code into polyhedral IR and explores various space mapping and time schedules corresponding to different systolic array configurations.

3.2 Domain-specific Languages (DSLs)

Using DSLs can simplify the work of both programmers and compilers to identify and exploit opportunities for advanced customizations that are commonly used in a specific application domain. For this reason, it is not surprising to see many domain- or application-specific languages emerging in the past several years either for FPGAs (and maybe also ASICs) [79, 137, 182, 192, 196, 230] or re-purposed from CPUs or GPUs to FPGAs [79, 137, 196, 197]. These languages provide a limited number of, but optimized, functions/operators specific to "hot" or important domains and applications (e.g., image processing, machine learning, network packet processing, software-defined radios, and even controllers of FPGA systems).

The narrower focus of the application domains enables DSLs to provide high-level programming interfaces for high productivity, and at the same time, very specialized implementations for high performance. DSLs could be much more productive than HLS to express FPGA and domain-specific customizations. Below, we briefly review a few domains with some representative DSLs.

Linear Transform—A linear transform expressed as a matrix-vector multiplication (e.g., $\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ for two-point **discrete Fourier transform (DFT)**) can have many different algorithmic implementations. For example, DFT can have many fast versions such as Pease **fast Fourier transform (FFT)**, Cooley-Tukey FFT, mixed-radix FFT, and Bluestein FFT. Complex algorithms can be composed of multiplication, permutation, Kronecker product, and so on, of matrices representing linear transforms. Spiral [182] generates hardware for linear transforms widely used in signal processing and other domains, such as discrete Fourier and cosine transforms. Given a linear transform of a fixed size, Spiral chooses and expresses an algorithm as a formula, rewrites the formula based on rules and user-provided hardware directives, and finally generates synthesizable RTL. The hardware directives customize the mapping of the formula to hardware for reusing hardware components. The latency, throughput, and area of the mapping can be estimated quantitatively.

Image Processing—RIPL [230] defines a set of operations at the levels of pixels, window, and image. These operations are compiled into dataflow actors, which are finite state machines. The dataflow actors are lowered into CAL dataflow language [82] and then into Verilog. Darkroom [107] is a functional language restricted to stencil operations on images and automatically generates a schedule that is a pipeline of operations with minimum-sized line-buffers between them. Rigel [108] extends Darkroom for generating multi-rate pipelines, where the pipeline stages can fire with different rates. Hipacc [212] generates optimized OpenCL/CUDA code for image processing kernels running on FPGAs/GPUs.

Many computing patterns in image and video processing can be concisely described as nested loops in a declarative programming paradigm, as can be seen from several DSLs [107, 145, 177, 253]. Halide proposes decoupling an *algorithm* from a *schedule* for a compute [210]. Here the algorithm is a declarative specification of the compute. The schedule then specifies how to optimize the algorithm for performance. Programmers only specify what optimizations to do, while the actual implementation of the optimizations is left to a compiler. In this way, both productivity and performance can be achieved. Halide was originally only for CPUs and GPUs. Halide-HLS [208] extends Halide to target FPGAs. It generates a high-throughput image processing pipeline with line buffers, FIFOs, and also the glue code between the host and the FPGA accelerator.

Machine Learning—Many promising tools have emerged in this hot domain, such as TVM [27, 187], TABLA [172], DNNWeaver [221], DNNBuilder [272], Caffeine [271], and HLS4ML [79]. TVM [27] builds a deep learning compiler stack on top of Halide IR, supporting both CPUs and GPUs. TVM programs can target FPGAs as a back end by using VTA, a programmable accelerator that uses a RISC-like programming abstraction to describe tensor operations [187]. TABLA [172] implements an accelerator template, uniform across a set of machine learning algorithms that are based on stochastic gradient descent optimization. DNNWeaver [221] maps a Caffee [128] specification to a dataflow graph of macroinstructions, schedules the instructions, and uses hand-written, customizable templates to generate an accelerator. HLS4ML [79] translates neural network models learned by popular machine learning frameworks like Keras [139], PyTorch [201] and TensorFlow [1] into Xilinx Vivado HLS code, which generates bitstreams to run on Xilinx FPGAs.

Networking, Data Analytics, and Other Domains—P4FPGA framework [244] translates a network packet processing algorithm written in the P4 language into BlueSpec System Verilog for simulation and synthesis for FPGAs, and provides runtime support such as the communication between the host and the synthesized accelerator. Xilinx SDNet [258] is another FPGA-targeting framework for packet processing that supports the P4 language. Spark to FPGA Accelerator [265] generates HLS C code from Apache Spark program written in Scala for big data applications. FSM-Language [5] leverages Haskell to simplify the specification of finite state machines, which are common in hardware design.

3.3 Emerging Trends and Accelerator Design Languages

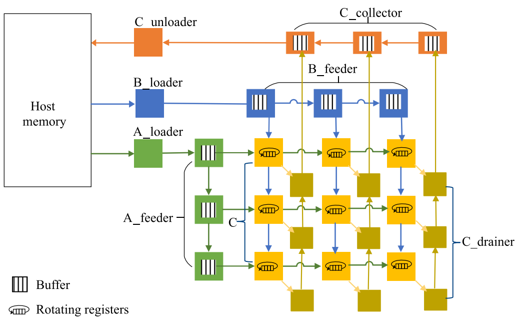
Recent years have seen several trends in accelerator design languages. First, *DSLs are becoming increasingly generalized, mixing imperative and declarative paradigms and/or embedded in general host languages.* While DSLs offer many advantages in productivity and compilation for individual application domains, more general-purpose language constructs are needed to (1) bridge the gaps between popular domains, (2) provide programmers with greater control on important customizations, (3) and serve as a compilation target for multiple high-level DSLs. Along this direction, we see new languages like DHDL, Spatial, and HeteroCL. DHDL [148] describes hardware with a set of parameterizable architectural templates that capture locality, memory access, and parallel compute patterns at multiple levels of nesting. DHDL is embedded in Scala and thus leverages Scala's language features and type system. Spatial [147] is a successor of DHDL, with more hardware-centric abstractions. HeteroCL [150] is embedded in Python and blends declarative symbolic expressions with imperative code.

Second, *new programming models are providing more explicit controls to programmers for more predictable behaviors of the generated accelerators.* It has become a common practice for languages to be designed with high-level types and parameterized constructs modeling reconfigurable hardware components. These hardware components are typically implemented manually with high performance and maybe parameterized for flexibility. Take several examples. Spatial [147] builds on parallel patterns, which map to efficient hardware templates; reduction can be done in registers or memory; programmers can build a custom memory hierarchy with at least three levels (DRAM, SRAM, and register); data can be stored on-chip in specific resources like look-up tables and specialized structures like queues, stacks, and line buffers. HeteroCL [150] allows programmers to customize memory (via creating a custom memory hierarchy through banking, reuse buffers, and data streaming), compute (various loop transforms), and data types (bit-accurate types and quantization); systolic arrays and stencils can be efficiently built/handled by leveraging PolySA/AutoSA [58, 245] and SODA [35]. Dahlia [192] resembles traditional C-based HLS, but enhances the type system to prevent unsafe, simultaneous uses of the same hardware resource; with the type system, programmers can explicitly enforce some of the unwritten rules only known by the HLS experts (e.g., the unrolling factor should divide the banking factor of the arrays accessed inside the loop to achieve the best performance-area trade-off). Aetherling [80] abstracts sequential or parallel hardware modules into a set of data-parallel operators and features types for space sequence and time sequence that encode the parallelism and throughput of the hardware modules. Well-typed operators can thus be composed into statically scheduled streaming circuits.

Third, *separation of concerns becomes popular in language designs.* Inspired by Halide [210], HeteroCL [150], and T2S [151, 213, 229] separate algorithm definition from hardware customization, which leads to higher productivity. In the algorithm definition, HeteroCL allows both imperative programming and declarative programming, making the language more general compared to other DSLs such as Halide and TVM. For hardware customization, HeteroCL supports all three essential

```
 1 /* Temporal definition */
 2 A(i, j, k) = select(j == 0, a(i, k), A(i, j-1, k));
 3 B(i, j, k) = select(i == 0, b(k, j), B(i-1, j, k));
 4 C(i, j, k) = select(k == 0, 0, C(i, j, k-1))
 5                          + A(i, j, k) * B(i, j, k);
 6
 7 /* Spatial mapping */
 8 // Custom compute engines
 9 A.merge_ures(B, C)
10  .space_time_transform({i, j},       // Space loops
11                       {0, 0, 1});   // Time vector
12 // Custom memory hierarchy
13 C.isolate_producer_chain(a, {A_loader, A_feeder})
14  .isolate_consumer_chain(C,
15             {C_drainer, C_collector, C_unloader});
16 A_loader.remove(j);
17 A_feeder.scatter(i).buffer(A, i, Buffer::Double);
18 B_loader.remove(i);
19 B_feeder.scatter(j).buffer(B, j, Buffer::Double);
20 C_drainer.gather(C, i);
21 C_collector.gather(C_drainer, j);
```

(a) SuSy program



(b) Generated systolic architecture

Fig. 6. Matrix-matrix multiplication in SuSy and the generated hardware architecture—SuSy adopts separation of concerns by decoupling temporal definition (L1–5) from spatial mapping (L7–21).

techniques described in Section 2. For custom data representations, HeteroCL supports arbitrary-precision integers and fixed-point types. For custom compute engines, HeteroCL supports pipelining, unrolling, and several other loop transformations. For custom memory hierarchy, HeteroCL supports several on-chip buffering optimizations such as memory banking and data reuse.

T2S is designed based on an observation that "no matter how complicated an implementation is, every spatial piece of it must be realizing a part of the functionality of the original workload, and they are communicating based on production-consumption relationship" [213]. Therefore, a programmer could specify a temporal definition and a spatial mapping. The temporal definition defines the original workload functionally, while the spatial mapping defines how to decompose the functionality and map the decomposed pieces onto a spatial architecture. The specification precisely controls a compiler to actually implement the loop and data transformations specified in the mapping. So far, T2S focuses on expressing high-performance systolic arrays, which are often the most efficient way for accelerating a workload on an FPGA. While previous studies focus on how a systolic array computes, the input/output data paths are barely discussed. However, I/O is often the most complicated part in a real-world implementation. T2S allows a full systolic system to be built, including the core systolic array for a compute, other helper arrays for I/O data paths, host pre- and post-processing, and host and FPGA communication. T2S has two implementations, T2S-Tensor [229] and SuSy [151] for building asynchronous and synchronous arrays, respectively. Figure 6 illustrates SuSy with matrix multiplication. SuSy expresses the temporal definition as **uniform recurrence equations (UREs)**, creates PEs with a space-time transform, and connects the PEs via shift registers. Note that UREs and space-time transformation are the theoretical foundation of most systolic arrays.

4 CHALLENGES AND OPPORTUNITIES AHEAD

It is clear that the latest generation of FPGA-targeted compilers and programming languages have made significant progress in providing broad language coverage and robust compilation technology. As a result, FPGA programmers can now more productively implement various

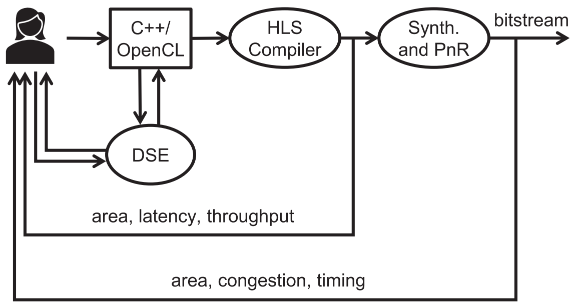


Fig. 7. Current flow to compile a high-level specification to a bitstream.

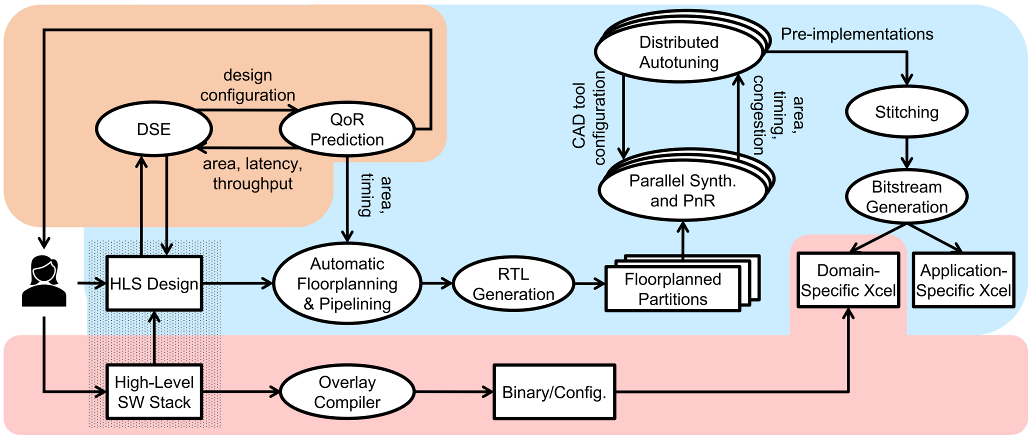


Fig. 8. Our vision to compile a high-level specification to a bitstream within minutes.

important hardware customization techniques to build an efficient accelerator with **quality of results (QoRs)**, in many cases, comparable to or better than manual RTL designs. Despite this encouraging development, current FPGAs are not fully software programmable—at least not in the conventional manner that works for microprocessors, even with the introduction of HLS. Figure 7 depicts the current design flow to compile a high-level specification to a bitstream. At each design stage of this flow, programmers need to analyze reports generated by the tools and modify high-level specification of the design accordingly. This is an iterative and labor-intensive process until design constraints are met. It may take hours or days to compile HLS-generated RTL to a bitstream due to the slow physical design steps. Not surprisingly, software-inclined developers often fail to appreciate the complexity involved in implementing new functionalities on FPGA devices to achieve a high performance.

In light of the challenges with the current flow and recent advancements in FPGA technology and other related fields, we are envisioning a new flow in Figure 8 that will enable much higher productivity for both hardware designers and software programmers. It has the potential to compile a high-level specification to a bitstream within minutes, and with significantly less manual effort involved. In this new flow, we also believe domain-specific overlay architectures will play an increasingly important role in providing a programming experience that resembles software development. Furthermore, integration with the high-level software stack is included to increase adoption of FPGAs by software programmers.

The rest of this section discusses several remaining challenges with FPGA programming and outlines the opportunities ahead. To be specific, we are touching upon five objectives: (1) quicker physical design closure, (2) faster design space exploration, (3) higher hardware abstraction, (4) higher software abstraction, and (5) easier debugging. In Section 4.1, we discuss accelerating the physical design closure of an application-specific accelerator. In Section 4.2, we explore how to further speed up design space exploration by means of various **machine learning (ML)** techniques. In Section 4.3, we discuss the benefits and opportunities introduced by overlays, namely, the soft domain-specific accelerators. In Section 4.4, we outline the requirements for further raising the software abstraction of the FPGA accelerators through tighter integration with high-level software stack. Finally, in Section 4.5, we discuss the challenges and progresses made related to debugging.

4.1 Quicker Physical Design Closure

Even though HLS offers high productivity for FPGA designers, RTL designs generated by the HLS tools may be prone to **placement-and-routing (PnR)** issues and timing degradation during implementation. Such deficiencies are aggregated by the long run time of the physical design stages. It is vital to generate the bitstream of an accelerator within minutes, and we believe this can be made possible with the integration of two strategies, respectively: (1) HLS designs should be floorplanned and pipelined effectively through physical layout awareness; (2) time-consuming physical design should be accelerated through design modularity and parallel PnR. There are a number of studies implementing these strategies separately. However, we envision that the key to minute-scale design closure is to integrate these two strategies into a flow as shown in Figure 8.

Physical design objectives are not directly addressed by HLS optimizations, and furthermore, objectives of different design stages can be conflicting. Therefore, achieving design closure in an efficient manner is only possible by making HLS optimizations aware of the physical layout. It is essential to identify and resolve routing and timing issues prior to RTL instead of resorting to the time-consuming physical design stages. In the minute-scale flow that we envision, ease and acceleration in timing closure can be realized by means of three approaches: First, predictions on HLS and physical design objectives are needed to guide DSE. Second, predictions on low-level physical design metrics are back-annotated to high-level source code, making it easier for FPGA programmers to locate and resolve potential congestion and timing issues early in the design flow. Third, predictions on physical design objectives can be used to effectively floorplan and pipeline a high-level design, facilitating frequency improvement and parallel compilation of physical design stages. Some of these strategies have been separately addressed by prior work.

Recently, Zhao et al. use ML models to estimate routing congestion from HLS IR and back-annotate this congestion information to high-level source code [276]. These back-annotations can be used by HLS designers to resolve congestion and improve frequency. Another line of work attempts to identify common communication and computation patterns that lead to timing degradation and propose techniques to resolve them [61, 101]. Cong et al. leverage pipeline transfer controllers on the datapaths where certain data broadcast patterns are detected [61]. Guo et al. identify and resolve common data and control broadcast structures generated by HLS compilers [101] A recent framework called AutoBridge [100] addresses timing degradation of large HLS-generated designs on multi-die FPGA devices, which is caused by the difficulty in predicting interconnect delay at the HLS level. AutoBridge floorplans a given dataflow design across multiple dies in a coarse-grained manner, while die-crossing connections are pipelined to improve frequency. These work show significant improvement in maximum frequency achieved.

The running time of placement and routing has been one of the major bottlenecks in FPGA design. We envision that exploiting structural modularity of designs will be the key to speeding up the downstream flow, because modularity allows both parallelization and reuse of physical design effort. An encouraging example is RapidRoute [162], which builds custom routers exploiting reuse and parallelism of communication structures. In the minute-scale flow that we envision, floorplanning HLS designs facilitates parallel downstream compilation with recent advancements in pre-implementation [152] and partial reconfiguration [256]. Partitions resulting from HLS floorplanning can be implemented in parallel, while also optimized using distributed autotuning frameworks such as LAMDA [239]. Resulting PnR solutions, which are referred to as pre-implementations, can be combined together using a stitching methodology provided by RapidWright [152] to construct the full design.

Many past efforts on fast PnR went into parallelizing the placement and routing algorithms based on simulated annealing, leveraging multiple CPU cores [87, 93, 114, 168, 169, 242]. Another line of work parallelizes analytic placement algorithms that have been shown to scale better to large-scale designs [156, 202]. More recently, Lin et al. developed an open-source project called DREAMPlace [158], which casts analytical placement of VLSI circuits to training a neural network, leading to significant speedup with the use of deep learning toolkits. In order to parallelize routing algorithms, a number of papers explore coarse-grained approaches where signals are assigned to different cores for parallel processing [94, 111, 183, 222–224, 243], many of which are coupled with parallel maze expansion for a single net.

Another strategy to speed up the PnR is realized with partial reconfiguration, where a design is decomposed to enable separate compilation of leaf modules communicating through a packet-switched network [200, 256]. Integral to these algorithmic advancements, in recent years, we are also seeing advancements in open-source platforms that allow designers to build custom solutions to keep up with the increasing design complexity and productivity demands [152, 189, 190].

4.2 Faster Design Space Exploration (DSE)

FPGA programmers need to explore a wide range of hardware customization options, while fundamental design objectives are area, latency, and power. Programmers can configure HLS directives to guide the synthesis process in compliance with their design objectives. These directives constitute a large and complex search space, which makes manual or exhaustive exploration highly inefficient. Therefore, many methodologies have been proposed to perform efficient DSE in HLS through a number of multi-objective optimization techniques.

Surrogate modeling is performed especially when objective function's behavior is complex and its evaluation is time-consuming [205]. Replacing costly evaluations with approximations lead to faster DSE convergence, bringing about a trade-off between speed and accuracy that needs to be calibrated carefully. Consequently, ML methods have been used as objective function estimators in HLS to account for optimizations that take place in logic synthesis and physical design. Dai et al. leverage ML models to narrow the gap between resource usage prediction in HLS and actual post-routing results [72]. Zhao et al. propose congestion prediction at the HLS level and map congested regions to HLS IR operations [276]. Ustun et al. perform operation delay prediction in HLS by leveraging emerging graph learning techniques, specifically **graph neural networks (GNNs)** [238]. Makrani et al. leverage stacked regression ensemble models to predict optimal resource usage and maximum achievable frequency values of a given post-HLS design [173]. A number of papers leverage such surrogate modeling to speed up DSE. Kurek et al. model an objective function with a Gaussian process and use an SVM classifier to estimate whether design constraints are met [149]. Lo et al. propose a sequential model-based optimization technique in which Gaussian process-based modeling of the objective function is enhanced with estimations provided by the CAD tool at multiple design stages [164–166]. Mehrabi et al. also leverage a Gaussian process, but unlike prior work, they perform multi-objective Bayesian optimization targeting latency and heterogeneous resource usage [176].

Another line of work builds analytical models to eliminate the need to wait for performance results reported at the end of HLS toolflow during DSE. Zhong et al. propose a dynamic analysis method in which applications are profiled and dynamic data dependence graphs are constructed [280]. Dependence information is then used to predict performance accurately across the search space of directive configuration. Zhao et al. propose a metric-guided DSE framework where performance and resource models are constructed with dataflow graph analysis across a wider range of directives [275]. Furthermore, leveraging prior knowledge to speed up convergence has been applied to analytical models [85].

4.3 Raising Hardware Abstraction

The previous sections mostly focus on the application-specific hardware synthesis, such as HLS, since building a fully application-customized accelerator can theoretically harness an FPGA's peak potential. However, the downsides of this approach are also evident—long compilation time and unpredictable design closure. An alternative is to target an overlay, which refers to a virtual hardware architecture that executes instructions and is "laid" on top of the physical FPGA fabric [17, 170, 264]. Overlays offer a more constrained configuration space, thus admitting faster compilation with bitstream reuse across applications from the same domain.

The idea of implementing customizable soft-core processors on FPGAs was initially introduced in mid 1990s in References [125, 154] and revisited in References [63, 264], which are early examples of overlay. The concept of an FPGA overlay was later used in References [17, 143, 175]. Recent overlay architectures specialize the reconfigurable fabric for specific computational patterns such as vectorizable code [46, 219, 220, 266], GPU-like SIMT parallelism [6, 170], or key operators and layer types for deep neural network [3, 88, 209]. Clearly, the recent development of overlay architectures represents a promising direction to enable true software-based programming on commercial off-the-shelf FPGAs. Nevertheless, there remain several challenges and a host of new research opportunities related to the compilation and synthesis for overlays.

First, building the domain-specific compiler for a new and evolving overlay can be a significant engineering undertaking. Compared to CPUs/GPUs, the **instruction set architecture (ISA)** for an overlay is typically simpler but often evolving more rapidly. Overlays need to scale in capability (e.g., PE datapath, off-chip memory interfaces) and size (e.g., PE counts, buffer capacities, spanning a single or multiple FPGAs) to adapt to changes in application demands and cost requirements. Therefore, an overlay compiler must be retargetable to a "moving target" ISA. Second, most overlay architectures are currently implemented at RTL, and by hardware experts, with a high development cost. Hence, they are often only available for hot domains or applications (e.g., deep learning these days [3, 27, 261]). Existing overlays might not necessarily match well with new algorithms or workloads. Hence, there is also a need to enable software programmers, instead of hardware experts, to quickly build an application-specific overlay on an FPGA, possibly leveraging existing HLS compilers and a set of high-level customizable IPs.

4.4 Raising Software Abstraction

Tighter integration with the software stack, with an emphasis on system-level performance, is also very important and often overlooked. Existing FPGA accelerators are typically packaged in a vendor-specific format such as encrypted netlists, HDL modules, and HLS C/C++ functions, all of which assume hardware-centric interfaces. Thus, nontrivial non-recurring engineering cost is incurred almost every time a new FPGA accelerator gets integrated into the software stack. To enable mainstream adoption of FPGAs in the software community, it is essential to support the automatic compilation from high-level user-facing programming interfaces such as Java/Scala for big data applications and Python-based DSLs (e.g., PyTorch, TensorFlow) for AI experts and data scientists. The end goal is to provide programming abstractions and tools that empower users with software-only backgrounds to productively use FPGAs near their capability.

From end users' standpoint, what really matters is to achieve high end-to-end performance at the system level, instead of just optimizing one accelerator. Hence, co-optimizations are needed among multiple accelerators, and between accelerators and the processor where software is running. Some recent efforts such as the Blaze system [33, 115] have studied the efficient programming and runtime support to integrate pre-synthesized FPGA accelerators into the widely used big data computing framework Apache Spark, which is based on Java Virtual Machine. Efficient CPU and FPGA co-optimization has also been considered in Reference [51].

Another important direction is to leverage virtualization so FPGA accelerators can be deployed as easily as traditional software services. Achieving this goal requires the development of new runtime and architectural support for efficient **dynamic partial reconfiguration (DPR)**. More concretely, efficient coarse-grained abstractions of heterogeneous FPGA resources (such as virtual tiles) are necessary to speed up DPR and enable virtualization of the accelerators that may scale in size. In addition, new software runtimes are needed to reconfigure the FPGA fabric on demand and perform intelligent resource management and scheduling based on the varying workload requirements. Several recent attempts have shown promising results on virtualizing the FPGA fabric, such as AmorphOS [141] and VITAL [269], while much more work is needed before FPGA virtualization becomes readily usable for the masses.

4.5 Easier Debugging

One of the biggest challenges for software-defined hardware accelerators is *what you see in the software program is not necessarily what you get in the hardware*. Therefore, effective debugging support for HLS—handling both the correctness- and performance-related issues—is essential to advance FPGA programming. Specifically, a good HLS debugging tool for software-defined hardware accelerators needs to achieve the following goals. First, it needs to maintain the software to hardware mapping so that the debugging is observable at the original software code level. Second, it needs to be flexible so that users can choose to observe any part of the program at any given time. Third, it needs to be fast for debugging complex designs, since many bugs are only triggered at certain execution stages of the program or at certain combinations of the inputs. Fourth, it needs to support both correctness and performance debugging, since writing a correct program is just the first step and the ultimate goal of hardware acceleration is to achieve high performance. Fifth, it needs a user-friendly interface similar to software debuggers like GNU debugger.

While algorithm-level bugs can be detected and fixed using software simulation, the challenges lay in debugging the generated hardware design, which often requires hardware simulation at RTL or on-board debugging (also known as in-system debugging). Unfortunately, today's RTL simulation tools are often too slow to debug practical designs with a large amount of input data. Therefore, current approaches often favor on-board debugging, which can be done by inserting `printf` [122, 259] and/or assertions [65, 105, 236], or by recording traces during execution [20, 83, 86, 92, 110, 186]. The printf- and assertion-based approaches are not flexible enough as they require programmers to manually insert code wherever they want to observe the program variable changes. Moreover, programmers must rebuild the bitstream, which is time consuming.

Most recent efforts focus on the trace-based approach. Currently, commercial tools like Xilinx ChipScope [257] and Intel SignalTap require users to select the monitoring signals in the RTL signal list, which is hard for users to observe in the original software code level. To address this deficiency, recent academic studies in References [20, 83, 86, 91, 92, 110, 184–186, 206] try to maintain cross-mapping between the software variables and the corresponding registers/signals in hardware so that users can select variables at the software level. Studies in References [18, 20, 83, 92, 110, 206] also provide a software-like debugging interface such that users can insert breakpoints, step through the code and inspect variable values. One issue for the trace-based approach is that the trace buffers are limited by the on-chip RAM size, which limits the time period and the number of variables that can be traced in a single build. There are multiple techniques explored to address this issue. For example, the work in Reference [91] only stores values that have changed; the work in Reference [92] restructures variable values from other variables offline; and the work in Reference [86] compresses the control-flow trace of HLS programs. Overlays for debugging have also been proposed in References [83, 92, 110] to avoid repeated compilation and enable dynamic tracing of more variables.

Besides the aforementioned correctness debugging, there are also recent studies on the support of performance debugging to help programmers identify the performance bottlenecks [40–42, 45, 65, 216, 241]. Clearly, there is a great need for more research in the debugging tools that are observable at the software level, flexible, fast, and support both correctness and performance debugging. Significantly faster hardware simulation techniques are needed. The trace-based on-board debugging built with overlay techniques looks promising. Research automating functional and performance debugging also deserves more attention.

5 CONCLUSION

This article surveys the recent programming flows as well as the essential compilation and synthesis techniques that enable software-defined FPGA acceleration. We particularly focus on the customization techniques that are unique to FPGA accelerator designs, instead of those code optimizations well-established for CPU or GPU targets. In addition, we highlight existing challenges and future opportunities of FPGA-based computing including faster design closure, higher hardware/software abstraction, and easier debugging. We envision that this article can serve as a useful guide for both academic researchers and industry practitioners, who are interested in developing high-performance FPGA accelerators using high-level software programs.

ACKNOWLEDGMENTS

We thank the reviewers and editorial committee for their helpful feedback.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. Retrieved from https://arXiv:1603.04467.

[2] Mohamed S. Abdelfattah and Vaughn Betz. 2014. Networks-on-Chip for FPGAs: Hard, soft or mixed? *ACM Trans. Reconfig. Technol. Syst.* 7, 3 (2014), 1–22.

[3] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, et al. 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18)*. 411–4117.

[4] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. 2011. Leap scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'11)*.

[5] Jason Agron. 2009. Domain-specific language for HW/SW Co-Design for FPGAs. In *IFIP Working Conference on Domain-Specific Languages.*

[6] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. 2016. FGPU: An SIMT-architecture for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 254–263.

[7] Mythri Alle, Antoine Morvan, and Steven Derrien. 2013. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Proceedings of the Design Automation Conference (DAC'13)*.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2019).

[9] Samridhi Bansal, Hsuan Hsiao, Tomasz Czajkowski, and Jason H. Anderson. 2018. High-level synthesis of software-customizable floating-point cores. In *Proceedings of the Design, Automation, and Test in Europe (DATE'18)*.

[10] Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*.

[11] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl Von Platen, Marco Mattavelli, and Mickaël Raulet. 2009. OpenDF: A dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Comput. Architect. News* 36, 5 (2009), 29–35.

[12] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5, 720–748.

[13] David Boland and George A. Constantinides. 2010. Automated precision analysis: A polynomial algebraic approach. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'10)*.

[14] David Boland and George A. Constantinides. 2012. A scalable approach for automated precision analysis. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'12)*.

[15] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*.

[16] Uday Bondhugula, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2007. Automatic mapping of nested loops to FPGAs. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Parallel Programming (PPoPP'07)*.

[17] Alexander Brant and Guy G. F. Lemieux. 2012. ZUMA: An open FPGA overlay architecture. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'12)*.

[18] Pavan Kumar Bussa, Jeffrey Goeders, and Steven J. E. Wilton. 2017. Accelerating In-System FPGA debug of high-level synthesis circuits using incremental compilation techniques. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'17)*.

[19] Cadence. 2020. Stratus High-Level Synthesis. Retrieved from https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf.

[20] Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. 2014. Source-level debugging for FPGA high-level synthesis. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14)*. 1–8.

[21] Andrew Canis, Jason H. Anderson, and Stephen D. Brown. 2013. Multi-pumping for resource reduction in FPGA high-level synthesis. In *Proceedings of the Design, Automation, and Test in Europe (DATE'13)*.

[22] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14)*.

[23] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'11)*.

[24] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, et al. 2013. From software to accelerators with LegUp high-level synthesis. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES'13)*.

[25] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. 2019. Deep positron: A deep neural network using the posit number system. In *Proceedings of the Design, Automation, and Test in Europe (DATE'19)*.

[26] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO'16)*.

[27] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.

[28] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. 2018. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *Proceedings of the International Symposium on Microarchitecture (MICRO'18)*.

[29] Tao Chen and G. Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the International Symposium on Microarchitecture (MICRO'16)*.

[30] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2019. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'19)*.

[31] Yao Chen, Swathi T. Gurumani, Yun Liang, Guofeng Li, Donghui Guo, Kyle Rupnow, and Deming Chen. 2016. FCUDA-NoC: A scalable and efficient network-on-chip implementation for the CUDA-to-FPGA flow. *IEEE Trans. Very Large Scale Integr. Syst.* 24, 6 (2016), 2220–2233.

[32] Yu Ting Chen and Jason H. Anderson. 2017. Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'17)*.

[33] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. 2016. When spark meets FPGAs: A case study for next-generation DNA sequencing acceleration. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud'16)*.

[34] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*.

[35] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)* (2018).

[36] Yuze Chi, Licheng Guo, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending high-level synthesis for task-parallel programs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*.

[37] Jongsok Choi, Stephen Brown, and Jason Anderson. 2013. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Proceedings of the International Conference on Field Programmable Technology (FPT'13)*.

[38] Jongsok Choi, Kevin Nam, Andrew Canis, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. 2012. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'12)*.

[39] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. 2021. HBM connect: High-performance HLS interconnect for FPGA HBM. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*.

[40] Young-Kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. FLASH: Fast, parallel, and accurate simulator for HLS. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* (2020).

[41] Young-kyu Choi and Jason Cong. 2017. HLScope: High-level performance debugging for FPGA designs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'17)*.

[42] Young-kyu Choi and Jason Cong. 2018. HLS-based optimization and design space exploration for applications with variable loop bounds. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18)*.

[43] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proceedings of the Design Automation Conference (DAC'16)*.

[44] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2019. In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms. *ACM Trans. Reconfig. Technol. Syst.* (2019).

[45] Young-kyu Choi, Peng Zhang, Peng Li, and Jason Cong. 2017. HLScope+: Fast and accurate performance estimation for FPGA HLS. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'17)*.

[46] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G. F. Lemieux. 2011. VEGAS: Soft vector processor with scratchpad memory. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'11)*.

[47] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An in-fabric memory architecture for FPGA-based computing. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'11)*.

[48] Alessandro Cilardo and Luca Gallo. 2015. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Trans. Architect. Code Optimiz.* 11, 4 (2015).

[49] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. 2005. Facilitating the search for compositions of program transformations. In *Proceedings of the International Symposium on Supercomputing (ICS'05)*.

[50] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. 2018. Best-effort FPGA programming: A few steps can go a long way. Retrieved from https://arXiv:1807.01340.

[51] Jason Cong, Zhenman Fang, Muhuan Huang, Libo Wang, and Di Wu. 2017. CPU-FPGA coscheduling for big data applications. *IEEE Design Test* 35, 1 (2017), 16–22.

[52] Jason Cong, Zhenman Fang, Muhuan Huang, Peng Wei, Di Wu, and Cody Hao Yu. 2018. Customizable computing–from single chip to datacenters. *Proc. IEEE* 107, 1 (2018), 185–203.

[53] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding performance differences of FPGAs and GPUs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18).*

[54] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. 2016. Source-to-source optimization for HLS. *FPGAs Softw. Program.* (2016).

[55] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Design Autom. Electron. Syst.* 16, 2 (2011), 1–25.

[56] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. 2016. An optimal microarchitecture for stencil computation acceleration based on nonuniform partitioning of data reuse buffers. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 35, 3 (2016), 407–418.

[57] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 30, 4 (2011), 473–491.

[58] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18).*

[59] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *Proceedings of the Design Automation Conference (DAC'18).*

[60] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2017. Bandwidth optimization through on-chip memory restructuring for HLS. In *Proceedings of the Design Automation Conference (DAC'17).*

[61] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality aware transformation for high-level synthesis. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18).*

[62] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the Design Automation Conference (DAC'06).*

[63] James Coole and Greg Stitt. 2010. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'10).*

[64] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. 2008. GAUT: A high-level synthesis tool for DSP applications. *High-Level Synth.* (2008).

[65] John Curreri, Seth Koehler, Alan D. George, Brian Holland, and Rafael Garcia. 2010. Performance analysis framework for high-level language applications in reconfigurable computing. *ACM Trans. Reconfig. Technol. Syst.* 3, 1 (2010), 1–23.

[66] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. 2012. From OpenCL to high-performance hardware on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'12).*

[67] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16).*

[68] Steve Dai, Gai Liu, and Zhiru Zhang. 2018. A scalable approach to exact resource-constrained scheduling based on a joint SDC and SAT formulation. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18).*

[69] Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. 2017. Enabling adaptive loop pipelining in high-level synthesis. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers.*

[70] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. 2014. Flushing-enabled loop pipelining for high-level synthesis. In *Proceedings of the Design Automation Conference (DAC'14).*

[71] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. 2017. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17).*

[72] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F. Y. Young, and Zhiru Zhang. 2018. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18).*

[73] Luka Daoud, Dawid Zydek, and Henry Selvaraj. 2014. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. *Adv. Syst. Sci.* (2014).

[74] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. 2020. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Adv. Neural Info. Process. Syst.* (2020).

[75] Florent De Dinechin and Bogdan Pasca. 2011. Designing custom arithmetic data paths with FloPoCo. *IEEE Design Test Comput.* 28, 4 (2011), 18–27.

[76] Luiz Henrique De Figueiredo and Jorge Stolfi. 2004. Affine arithmetic: Concepts and applications. *Numer. Algor.* (2004).

[77] Johannes de Fine Licht, Simon Meierhans, and Torsten Hoefler. 2018. Transformations of high-level synthesis codes for high-performance computing. Retrieved from https://arXiv:1805.08288.

[78] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. 2020. Toward speculative loop pipelining for high-level synthesis. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 39, 11 (2020), 4229–4239.

[79] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrument.* (2018).

[80] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20).*

[81] Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. 2019. Compositional dataflow circuits. *ACM Trans. Embed. Comput. Syst.* 18, 1 (2019), 1–27.

[82] Johan Eker and J. Janneck. 2003. CAL language report: Specification of the CAL actor language. *ERL Tech. Memo UCB/ERL* (2003).

[83] Fatemeh Eslami and Steven J. E. Wilton. 2018. Rapid triggering capability using an adaptive overlay during FPGA debug. *ACM Trans. Design Autom. Electron. Syst.* 23, 6 (2018), 1–25.

[84] Zhenman Fang, Farnoosh Javadi, Jason Cong, and Glenn Reinman. 2019. Understanding performance gains of accelerator-rich architectures. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP'19).*

[85] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca P. Carloni, and Laura Pozzi. 2020. Leveraging prior knowledge for effective design-space exploration in high-level synthesis. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 39, 11 (2020), 3736–3747.

[86] Pietro Fezzardi, Marco Lattuada, and Fabrizio Ferrandi. 2017. Using efficient path profiling to optimize memory consumption of on-chip debugging for high-level synthesis. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 1–22.

[87] Christian Fobel, Gary Grewal, and Deborah Stacey. 2014. A scalable, serially equivalent, high-quality parallel placement methodology suitable for modern multicore and GPU architectures. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14).*

[88] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18).*

[89] Tushar Garg, Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. 2020. HopliteBuf: Network calculus-based design of FPGA NoCs with provably stall-free FIFOs. *ACM Trans. Reconfig. Technol. Syst.* 13, 2 (2020).

[90] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. 2018. ReBNet: Residual binarized neural network. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18).*

[91] Jeffrey Goeders and Steven J. E. Wilton. 2014. Effective FPGA debug for high-level synthesis generated circuits. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14).*

[92] Jeffrey Goeders and Steven J. E. Wilton. 2016. Signal-tracing techniques for In-System FPGA debugging of high-level synthesis circuits. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 36, 1 (2016), 83–96.

[93] Jeffrey B. Goeders, Guy G. F. Lemieux, and Steven J. E. Wilton. 2011. Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition. In *Proceedings of the International Conference on Reconfigruable Computing and FPGAs (ReConFig'11).*

[94] Marcel Gort and Jason H. Anderson. 2010. Deterministic multi-core parallel routing for FPGAs. In *Proceedings of the International Conference on Field Programmable Technology (FPT'10).*

[95] Marcel Gort and Jason H. Anderson. 2013. Range and bitmask analysis for hardware optimization in high-level synthesis. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'13).*

[96] Ian Gray, Yu Chan, Jamie Garside, Neil Audsley, and Andy Wellings. 2015. Transparent hardware synthesis of Java for predictable large-scale distributed systems. Retrieved from https://arXiv:1508.07142.

[97] Paul Grigoraş, Xinyu Niu, Jose G. F. Coutinho, Wayne Luk, Jacob Bower, and Oliver Pell. 2013. Aspect driven compilation for dataflow designs. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP'13).*

[98] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly–performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* (2012).

[99] Sikender Gul, Muhammad Faisal Siddiqui, and Naveed Ur Rehman. 2019. FPGA based real-time implementation of online EMD with fixed point architecture. *IEEE Access* 7 (2019), 176565–176577.

[100] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*.

[101] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency. In *Proceedings of the Design Automation Conference (DAC'20)*.

[102] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*.

[103] Peng Guo, Hong Ma, Ruizhi Chen, Pin Li, Shaolin Xie, and Donglin Wang. 2018. FBNA: A fully binarized neural network accelerator. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18)*.

[104] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2017. Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures. *ACM Trans. Architect. Code Optimiz.* 14, 2 (2017), 1–27.

[105] Mohamed Ben Hammouda, Philippe Coussy, and Loïc Lagadec. 2014. A design approach to automatically synthesize ANSI-C assertions during high-level synthesis of hardware accelerators. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS'14)*.

[106] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. 2008. PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC'08)*.

[107] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4 (2014), 144:1–144:11.

[108] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.* 25, 4 (2016), 1–11.

[109] Timothy Hickey, Qun Ju, and Maarten H. Van Emden. 2001. Interval arithmetic: From principles to implementation. *J. ACM* 48, 5 (2001), 1038–1068.

[110] Daniel Holanda Noronha, Ruizhe Zhao, Jeff Goeders, Wayne Luk, and Steven J. E. Wilton. 2019. On-Chip FPGA debug instrumentation for machine learning applications. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'19)*.

[111] Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*.

[112] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. 2008. Optimus: Efficient realization of streaming applications on FPGAs. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES'08)*.

[113] Hsuan Hsiao and Jason Anderson. 2019. Thread weaving: Static resource scheduling for multithreaded high-level synthesis. In *Proceedings of the Design Automation Conference (DAC'19)*.

[114] Bohu Huang and Haibin Zhang. 2013. Application of multi-core parallel computing in FPGA placement. In *Proceedings of the International Symposium on Instrumentation and Measurement, Sensor Network and Automation (IMSNA'13)*.

[115] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. 2016. Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale. In *Proceedings of the ACM Symposium on Cloud Computing*.

[116] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W Hwu, and Deming Chen. 2017. Hardware acceleration of the pair-HMM algorithm for DNA variant calling. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*.

[117] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic CGRAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'13)*.

[118] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA workflow for line-rate packet processing. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'19)*.

[119] Mohsen Imani, Samuel Bosch, Sohum Datta, Sharadhi Ramakrishna, Sahand Salamat, Jan M. Rabaey, and Tajana Rosing. 2019. QuantHD: A quantization framework for hyperdimensional computing. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 39, 10 (2019), 2268–2278.

[120] Mohsen Imani, Sahand Salamat, Behnam Khaleghi, Mohammad Samragh, Farinaz Koushanfar, and Tajana Rosing. 2019. SparseHD: Algorithm-hardware co-optimization for efficient high-dimensional computing. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*.

[121] Intel. 2019. Intel Agilex F-Series FPGAs & SoCs. Retrieved from https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex/f-series.html.

[122] Intel. 2020. Intel High Level Synthesis Compiler Pro Edition: Reference Manual. Retrieved from https://www.intel.com/content/www/us/en/programmable/documentation/ewa1462824960255.html.

[123] Intel. 2020. Intel SoC FPGAs. Retrieved from https://www.intel.ca/content/www/ca/en/products/programmable/soc.html.

[124] Intel. 2020. The oneAPI Specification. Retrieved from https://www.oneapi.com/.

[125] Christian Iseli and Eduardo Sanchez. 1993. Spyder: A reconfigurable VLIW processor using FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*.

[126] Asif Islam and Nachiket Kapre. 2018. LegUp-NoC: High-level synthesis of loops with indirect addressing. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18)*.

[127] Manish Kumar Jaiswal and Ray C. C. Cheung. 2013. Area-efficient architectures for double precision multiplier on FPGA, with run-time-reconfigurable dual single precision support. *Microelectr. J.* 44, 5 (2013), 421–430.

[128] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the International Conference on Multimedia*.

[129] Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu. 2020. Boyi: A systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*.

[130] Lana Josipovic, Philip Brisk, and Paolo Ienne. 2017. An out-of-order load-store queue for spatial computing. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 1–19.

[131] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*.

[132] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative dataflow circuits. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'19)*.

[133] Juniper. 2020. Juniper: Java Platform for High-performance and Real-time Large-scale Data. Retrieved from http://www.juniper-project.org/.

[134] Nachiket Kapre et al. 2018. Hoplite-Q: Priority-aware routing in FPGA overlay NoCs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18)*.

[135] Nachiket Kapre and Jan Gray. 2015. Hoplite: Building austere overlay NoCs for FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'15)*.

[136] Nachiket Kapre and Deheng Ye. 2016. GPU-Accelerated high-level synthesis for bitwidth optimization of FPGA datapaths. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*.

[137] Soguy Mak karé Gueye, Gwenaël Delaval, Eric Rutten, Dominique Heller, and Jean-Philippe Diguet. 2018. A domain-specific language for autonomic managers in FPGA reconfigurable architectures. In *Proceedings of the International Conference on Autonomic Computing (ICAC'18)*.

[138] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel programming for FPGAs. Retrieved from https://arXiv:1805.03648.

[139] Keras. 2020. Keras. Simple. Flexible. Powerful. Retrieved from https://keras.io/.

[140] Ronan Keryell and Lin-Ya Yu. 2018. Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended Abstract of technical presentation. In *Proceedings of the International Workshop on OpenCL*.

[141] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.

[142] Soroosh Khoram, Jialiang Zhang, Maxwell Strange, and Jing Li. 2018. Accelerating graph analytics by co-optimizing storage and access on an FPGA-HMC platform. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*.

[143] Jeffrey Kingyens and J. Gregory Steffan. 2011. The potential for a GPU-Like overlay architecture for FPGAs. *Intl. J. Reconfig. Comput.* (2011).

[144] Adam B. Kinsman and Nicola Nicolici. 2009. Finite precision bit-width allocation using SAT-Modulo theory. In *Design, Automation, and Test in Europe (DATE'09)*.

[145] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* (2017).

[146] Ana Klimovic and Jason H. Anderson. 2013. Bitwidth-optimized hardware accelerators with software fallback. In *Proceedings of the International Conference on Field Programmable Technology (FPT'13)*.

[147] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*.

[148] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.

[149] Maciej Kurek, Tobias Becker, Thomas C. P. Chau, and Wayne Luk. 2014. Automating optimization of reconfigurable designs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'14)*.

[150] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'19)*.

[151] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. 2020. SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'20)*.

[152] Chris Lavin and Alireza Kaviani. 2018. RapidWright: Enabling custom crafted implementations for FPGAs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18)*.

[153] D.-U. Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. 2006. Accuracy-guaranteed bit-width optimization. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 25, 10 (2006), 1990–2000.

[154] David M. Lewis, Marcus H. van Ierssel, and Daniel H. Wong. 1993. A field programmable accelerator for compiled-code applications. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*.

[155] Peng Li, Louis-Noël Pouchet, and Jason Cong. 2014. Throughput optimization for high-level synthesis using resource constraints. In *Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT'14)*.

[156] Wuxi Li, Meng Li, Jiajun Wang, and David Z. Pan. 2017. UTPlaceF 3.0: A parallelization framework for modern FPGA global placement. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'17)*.

[157] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. 2018. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* (2018).

[158] Yibo Lin, Zixuan Jiang, Jiaqi Gu, Wuxi Li, Shounak Dhar, Haoxing Ren, Brucek Khailany, and David Z. Pan. 2020. DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 40, 4 (2020), 748–761.

[159] Junyi Liu, Samuel Bayliss, and George A. Constantinides. 2015. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'15)*.

[160] Ji Liu, Abdullah-Al Kafi, Xipeng Shen, and Huiyang Zhou. 2020. MKPipe: A compiler framework for optimizing multi-kernel workloads in OpenCL for FPGA. In *Proceedings of the International Symposium on Supercomputing (ICS'20)*.

[161] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. 2017. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 37, 9 (2017), 1802–1815.

[162] Leo Liu, Jay Weng, and Nachiket Kapre. 2019. RapidRoute: Fast assembly of communication structures for FPGA overlays. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*.

[163] Qiang Liu, George A. Constantinides, Konstantinos Masselos, and Peter Y. K. Cheung. 2007. Automatic on-chip memory minimization for data reuse. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'07)*.

[164] Charles Lo and Paul Chow. 2016. Model-based optimization of high level synthesis directives. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'16)*.

[165] Charles Lo and Paul Chow. 2018. Multi-fidelity optimization for high-level synthesis directives. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18)*.

[166] Charles Lo and Paul Chow. 2020. Hierarchical modelling of generators in design-space exploration. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'20)*.

[167] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 2021. Demystifying the memory system of modern datacenter FPGAs for software programmers through microbenchmarking. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*.

[168] Adrian Ludwin and Vaughn Betz. 2011. Efficient and deterministic parallel placement for FPGAs. *ACM Trans. Design Autom. Electron. Syst.* 16, 3 (2011), 1–23.

[169] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. 2008. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'08)*.

[170] Rui Ma, Jia-Ching Hsu, Tian Tan, Eriko Nurvitadhi, David Sheffield, Rob Pelt, Martin Langhammer, Jaewoong Sim, Aravind Dasu, and Derek Chiou. 2019. Specializing FGPU for persistent deep learning. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'19)*. 326–333.

[171] Xiaoyin Ma, Walid A. Najjar, and Amit K. Roy-Chowdhury. 2015. Evaluation and acceleration of high-throughput fixed-point object detection on FPGAs. *IEEE Trans. Circ. Syst. Video Technol.* 25, 6 (2015), 1051–1062.

[172] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A unified template-based framework for accelerating statistical machine learning. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'16).*

[173] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. 2019. Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'19).*

[174] Maxeler. 2020. Maxeler High-performance Dataflow Computing Systems. Retrieved from https://www.maxeler.com/products/software/maxcompiler/.

[175] Séamas McGettrick, Kunjan Patel, and Chris Bleakley. 2011. High performance programmable FPGA overlay for digital signal processing. In *Proceedings of the International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC'11).*

[176] Atefeh Mehrabi, Aninda Manocha, Benjamin C. Lee, and Daniel J. Sorin. 2020. Prospector: Synthesizing efficient accelerators via statistical learning. In *Proceedings of the Design, Automation, and Test in Europe (DATE'20).*

[177] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2016. Hipacc: A domain-specific language and compiler for image processing. *IEEE Trans. Parallel Distrib. Syst.* 27, 1 (2016), 210–224.

[178] Mentor. 2020. Catapult High-Level Synthesis. Retrieved from https://s3.amazonaws.com/s3.mentor.com/public_documents/datasheet/hls-lp/catapult-high-level-synthesis.pdf.

[179] Microchip. 2020. LegUp 9.1 Documentation. Retrieved from https://download-soc.microsemi.com/FPGA/HLS-EAP/docs/legup-9.1-docs/index.html.

[180] Microchip. 2020. Microchip Acquires High-Level Synthesis Tool Provider LegUp to Simplify Development of PolarFire FPGA-based Edge Compute Solutions. Retrieved from https://www.microchip.com/en-us/about/news-releases/products/microchip-acquires-high-level-synthesis-tool-provider-legup.

[181] Microsoft. 2020. A Microsoft Custom Data Type for Efficient Inference. Retrieved from https://www.microsoft.com/en-us/research/blog/a-microsoft-custom-data-type-for-efficient-inference/.

[182] Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2012. Computer generation of hardware for linear digital signal processing transforms. *ACM Trans. Design Autom. Electron. Syst.* 16, 3 (2012), 1–23.

[183] Yehdhih Moctar, Mirjana Stojilović, and Philip Brisk. 2018. Deterministic parallel routing for FPGAs based on galois parallel execution model. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18).*

[184] Joshua S. Monson and Brad Hutchings. 2014. New approaches for in-system debug of behaviorally synthesized FPGA circuits. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14).*

[185] Joshua S. Monson and Brad L. Hutchings. 2015. Using source-level transformations to improve high-level synthesis debug and validation on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'15).*

[186] Joshua S. Monson and Brad L. Hutchings. 2018. Enhancing debug observability for HLS-based FPGA circuits through source-to-source compilation. *J. Parallel Distrib. Comput.* 117 (2018), 148–160.

[187] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: An open hardware-software stack for deep learning. Retrieved from https://arXiv:1807.04188.

[188] Antoine Morvan, Steven Derrien, and Patrice Quinton. 2013. Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 32, 3 (2013), 339–352.

[189] Kevin E. Murray, Mohamed A. Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. 2020. SymbiFlow and VPR: An open-source design flow for commercial and novel FPGAs. *IEEE Micro* (2020).

[190] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, et al. 2020. VTR 8: High-performance CAD and customizable FPGA architecture modelling. *ACM Trans. Reconfig. Technol. Syst.* 13, 2 (2020), 339–352.

[191] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 35, 10 (2016), 1591–1604.

[192] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20).*

[193] Mostafa W. Numan, Braden J. Phillips, Gavin S. Puddy, and Katrina Falkner. 2020. Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains. *IEEE Access* 8 (2020), 174692–174722.

[194] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA framework for vertex-centric graph computation. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'14)*.

[195] William George Osborne, Ray C. C. Cheung, José Gabriel F. Coutinho, Wayne Luk, and Oskar Mencer. 2007. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'07)*.

[196] Ganda Stephane Ouedraogo, Matthieu Gautier, and Olivier Sentieys. 2014. A frame-based domain-specific language for rapid prototyping of FPGA-based software-defined radios. *EURASIP J. Adv. Signal Process.* 1 (2014), 1–15.

[197] M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2016. FPGA-based accelerator design from a domain-specific language. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'16)*.

[198] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the Symposium on Application Specific Processors (SASP'09)*.

[199] Philippos Papaphilippou, Jiuxi Meng, and Wayne Luk. 2020. High-performance FPGA network switch architecture. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*.

[200] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. 2018. Case for fast FPGA compilation using partial reconfiguration. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18)*.

[201] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. Retrieved from https://arXiv:1912.01703.

[202] Ryan Pattison, Christian Fobel, Gary Grewal, and Shawki Areibi. 2015. Scalable analytic placement for FPGA on GPGPU. In *Proceedings of the International Conference on Reconfigruable Computing and FPGAs (ReConFig'15)*.

[203] Francesco Peverelli, Marco Rabozzi, Emanuele Del Sozzo, and Marco D. Santambrogio. 2018. OXiGen: A tool for automatic acceleration of C functions into dataflow FPGA-based kernels. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*.

[204] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'13)*.

[205] Christian Pilato, Daniele Loiacono, Antonino Tumeo, Fabrizio Ferrandi, Pier Luca Lanzi, and Donatella Sciuto. 2010. Speeding-Up expensive evaluations in high-level synthesis using solution modeling and fitness inheritance. *Comput. Intell. Exp. Optimiz. Problems* (2010).

[206] Jose P. Pinilla and Steven J. E. Wilton. 2016. Enhanced source-level instrumentation for FPGA in-system debug of high-level synthesis designs. In *Proceedings of the International Conference on Field Programmable Technology (FPT'16)*.

[207] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'13)*.

[208] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming heterogeneous systems from an image processing DSL. *ACM Trans. Architect. Code Optimiz.* 14, 3 (2017), 1–25.

[209] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song et al. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 26–35.

[210] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* (2013).

[211] B. Ramakrishna Rau. 1994. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the International Symposium on Microarchitecture (MICRO'94)*.

[212] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. 2017. Generating FPGA-Based image processing accelerators with hipacc. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'17)*.

[213] Hongbo Rong. 2017. Programmatic control of a compiler for generating high-performance spatial hardware. Retrieved from https://arXiv:1711.07606.

[214] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. 2018. ST-Accel: A high-level programming platform for streaming applications on FPGA. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18)*.

[215] Sahand Salamat, Mohsen Imani, Behnam Khaleghi, and Tajana Rosing. 2019. F5-HD: Fast flexible FPGA-based framework for refreshing hyperdimensional computing. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'19)*.

[216] Andrew G. Schmidt, Neil Steiner, Matthew French, and Ron Sass. 2012. HwPMI: An extensible performance monitoring infrastructure for improving hardware design and productivity on FPGAs. *Int. J. Reconfig. Comput.* (2012).

[217] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. 2002. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *J. VLSI Signal Process. Syst. Signal Image Video Technol.* 31, 2 (2002), 127–142.

[218] Jocelyn Sérot, François Berry, and Sameer Ahmed. 2013. CAPH: A language for implementing stream-processing applications on FPGAs. *Embed. Syst. Design FPGAs* (2013).

[219] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. 2014. Soft vector processors with streaming pipelines. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'14)*.

[220] Aaron Severance and Guy Lemieux. 2012. VENICE: A compact vector processor for FPGA applications. In *Proceedings of the International Conference on Field Programmable Technology (FPT'12)*.

[221] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Proceedings of the International Symposium on Microarchitecture (MICRO'16)*.

[222] Minghua Shen and Guojie Luo. 2015. Accelerate FPGA routing with parallel recursive partitioning. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'15)*.

[223] Minghua Shen and Guojie Luo. 2017. Corolla: GPU-accelerated FPGA routing based on subgraph dynamic expansion. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*.

[224] Minghua Shen, Guojie Luo, and Nong Xiao. 2020. Coarse-grained parallel routing with recursive partitioning for FPGAs. *IEEE Trans. Parallel Distrib. Syst.* 32, 4 (2020), 884–899.

[225] Sam Skalicky, Joshua Monson, Andrew Schmidt, and Matthew French. 2018. Hot & spicy: Improving productivity with python and HLS for FPGAs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'18)*.

[226] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2020. End-to-end optimization of deep learning applications. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*.

[227] Roman A. Solovyev, Alexandr A. Kalinin, Alexander G. Kustov, Dmitry V. Telpukhov, and Vladimir S. Ruhlov. 2018. FPGA implementation of convolutional neural networks with fixed-point calculations. Retrieved from https://arXiv:1808.09945.

[228] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. 2020. Comparison of arithmetic number formats for inference in sum-product networks on FPGAs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'20)*.

[229] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. 2019. T2S-Tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*.

[230] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. 2018. RIPL: A parallel image processing language for FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 11, 1 (2018), 1–24.

[231] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Bala Jayadev, Jeff Cuppett, Abbas Morshed, Brian Gaide, and Ygal Arbel. 2019. Versal network-on-chip (NoC). In *Proceedings of the Symposium on High-Performance Interconnects (Hot Interconnects'19)*.

[232] Synthesijer. 2020. Synthesijer GitHub. Retrieved from https://github.com/synthesijer/synthesijer.

[233] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. 2015. Mapping-aware constrained scheduling for LUT-Based FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'15)*.

[234] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. 2015. Elasticflow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'15)*.

[235] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A framework for massively parallel streaming on FPGAs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[236] Tim Todman and Wayne Luk. 2013. Runtime assertions and exceptions for streaming systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'13)*.

[237] Stephen M. Steve Trimberger. 2018. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology: This paper reflects on how Moore's law has driven the design of FPGAs through three epochs: The age of invention, the age of expansion, and the age of accumulation. *IEEE Solid-State Circ. Mag.* 10, 2 (2018), 16–29.

[238] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'20)*.

[239] Ecenur Ustun, Shaojie Xiang, Jinny Gui, Cunxi Yu, and Zhiru Zhang. 2019. LAMDA: Learning-assisted multi-stage autotuning for FPGA design closure. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*.

[240] Shervin Vakili, J. M. Pierre Langlois, and Guy Bois. 2013. Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 32, 12 (2013), 1853–1865.

[241] Anshuman Verma, Huiyang Zhou, Skip Booth, Robbie King, James Coole, Andy Keep, John Marshall, and Wu-chun Feng. 2017. Developing dynamic profiling and debugging support in OpenCL for FPGAs. In *Proceedings of the Design Automation Conference (DAC'17)*.

[242] Chris C. Wang and Guy G. F. Lemieux. 2011. Scalable and deterministic timing-driven parallel placement for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'11)*.

[243] Dekui Wang, Zhenhua Duan, Cong Tian, Bohu Huang, and Nan Zhang. 2020. ParRA: A shared memory parallel FPGA router using hybrid partitioning approach. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 39, 4 (2020), 830–842.

[244] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*.

[245] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*.

[246] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: The secret to high performance on cloud TPUs. *Google Cloud Blog* (2019).

[247] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*.

[248] Xiaojun Wang and Miriam Leeser. 2010. VFloat: A variable precision fixed- and floating-point library for reconfigurable hardware. *ACM Trans. Reconfig. Technol. Syst.* 16, 3 (2010), 1–23.

[249] Yu Wang, James C. Hoe, and Eriko Nurvitadhi. 2019. Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*.

[250] Yuxin Wang, Peng Li, and Jason Cong. 2014. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'14)*.

[251] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. 2013. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the Design Automation Conference (DAC'13)*.

[252] Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. 2017. HopliteRT: An efficient FPGA NoC for real-time applications. In *Proceedings of the International Conference on Field Programmable Technology (FPT'17)*.

[253] Richard Wei, Lane Schwartz, and Vikram Adve. 2017. DLVM: A modern compiler infrastructure for deep learning systems. Retrieved from https://arXiv:1711.03016.

[254] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the Design Automation Conference (DAC'17)*.

[255] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[256] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, Raphael Rubin, and André DeHon. 2019. Reducing FPGA compile time with separate compilation for FPGA building blocks. In *Proceedings of the International Conference on Field Programmable Technology (FPT'19)*.

[257] Xilinx. 2012. ChipScope Pro Software and Cores (UG029). Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/chipscope_pro_sw_cores_ug029.pdf.

[258] Xilinx. 2020. SDNet Packet Processor User Guide. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1012-sdnet-packet-processor.pdf.

[259] Xilinx. 2020. Vitis High-Level Synthesis User Guide. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[260] Xilinx. 2020. Zynq UltraScale+ MPSoC. Retrieved from https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[261] Yu Xing, Shuang Liang, Lingzhi Sui, Xijie Jia, Jiantao Qiu, Xin Liu, Yushun Wang, Yi Shan, and Yu Wang. 2019. DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 39, 10, 2668–2681.

[262] Li Yang, Zhezhi He, and Deliang Fan. 2018. A fully onchip binarized convolutional neural network FPGA implementation with accurate inference. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'18)*.

[263] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, and Kurt Keutzer. 2019. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'19)*.

[264] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. 2007. Exploration and customization of FPGA-based soft processors. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 26, 2 (2007), 266–277.

[265] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. 2018. S2FA: An accelerator automation framework for heterogeneous computing in datacenters. In *Proceedings of the Design Automation Conference (DAC'18)*.

[266] Jason Yu, Guy Lemieux, and Christpher Eagleston. 2008. Vector processing as a soft-core CPU accelerator. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'08)*. 222–232.

[267] David C. Zaretsky, Gaurav Mittal, Robert P. Dick, and Prith Banerjee. 2007. Balanced scheduling and operation chaining in high-level synthesis for FPGA designs. In *Proceedings of the International Symposium on Quality Electronic Design (ISQED'07)*.

[268] Hanqing Zeng and Viktor Prasanna. 2020. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*.

[269] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the cloud. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 845–858.

[270] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'15)*.

[271] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 38, 11 (2019), 2072–2085.

[272] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18)*.

[273] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. 2021. FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*.

[274] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'13)*.

[275] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'17)*.

[276] Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. 2019. Machine learning based routing congestion prediction in FPGA high-level synthesis. In *Proceedings of the Design, Automation, and Test in Europe (DATE'19)*.

[277] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*.

[278] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang. 2015. Area-efficient pipelining for FPGA-targeted high-level synthesis. In *Proceedings of the Design Automation Conference (DAC'15)*.

[279] Zhipeng Zhao and James C. Hoe. 2017. Using vivado-HLS for structural design: A NoC case study. Retrieved from https://arXiv:1710.10290.

[280] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. 2016. Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators. In *Proceedings of the Design Automation Conference (DAC'16)*.

[281] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput graph processing framework on FPGA. *IEEE Trans. Parallel Distrib. Syst.* 30, 10, 2249–2264.

[282] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. 2017. A new approach to automatic memory banking using trace-based address mining. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*.

[283] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*.

Received March 2021; revised May 2021; accepted May 2021