FPGA HLS Today: Successes, Challenges, and Opportunities

JASON CONG and JASON LAU, University of California, Los Angeles
GAI LIU and STEPHEN NEUENDORFFER, Xilinx Inc.
PEICHEN PAN, Falcon Computing Solutions Inc.
KEES VISSERS, Xilinx Inc.
ZHIRU ZHANG, Cornell University

The year 2011 marked an important transition for FPGA high-level synthesis (HLS), as it went from prototyping to deployment. A decade later, in this article, we assess the progress of the deployment of HLS technology and highlight the successes in several application domains, including deep learning, video transcoding, graph processing, and genome sequencing. We also discuss the challenges faced by today's HLS technology and the opportunities for further research and development, especially in the areas of achieving high clock frequency, coping with complex pragmas and system integration, legacy code transformation, building on open source HLS infrastructures, supporting domain-specific languages, and standardization. It is our hope that this article will inspire more research on FPGA HLS and bring it to a new height.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; **Methodologies for EDA;**

Additional Key Words and Phrases: High-level synthesis, design automation, design tools, electronic design automation, FPGA architecture, FPGA acceleration, customizable computing, hardware compiler, hardware description, reconfigurable applications

ACM Reference format:
Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 51 (August 2022), 42 pages.
https://doi.org/10.1145/3530775

1 INTRODUCTION

The year 2011 was an important milestone for FPGA **high-level synthesis (HLS)** technology. Early that year, Xilinx, the largest FPGA provider, completed the acquisition of the startup company AutoESL Design Technologies Inc., whose HLS tool AutoPilot was showing increasing adoption by FPGA customers worldwide. Late that year, *IEEE Transactions on Computer-Aided Design* published a keynote paper entitled "High-Level Synthesis for FPGAs: From Prototyping to

Authors are listed in alphabetical order.
This work was partially supported by CRISP, one of six JUMP centers, and the industrial partners of the Center for Domain-Specific Computing (CDSC; https://cdsc.ucla.edu).
Authors' addresses: J. Cong (corresponding author) and J. Lau, University of California, 468A Engineering VI, Los Angeles, California 90095; emails: cong@cs.ucla.edu, lau@cs.ucla.edu; G. Liu, S. Neuendorffer, and K. Vissers, Xilinx Inc, 2100 Logic Dr, San Jose, CA 95124; emails: gail@xilinx.com, stephen.neuendorffer@amd.com, kees.vissers@gmail.com; P. Pan, 1580 Summerfield Dr Campbell, CA 95008; email: peichenp@yahoo.com; Z. Zhang, Cornell University, 320 Rhodes Hall, Ithaca, NY 14853; email: zhiruz@cornell.edu.

1936-7406/2022/08-ART51 $15.00
https://doi.org/10.1145/3530775

Deployment" [1]. Using AutoPilot as the main example, the paper highlighted the progress of the HLS community in multiple dimensions, including (i) robust support of C/C++-based synthesis (AutoPilot was the first HLS tool that built upon and extended the then-nascent LLVM compiler infrastructure [2], which received wide adoption later on); (ii) a platform-based approach, which provided detailed modeling of various building blocks on an FPGA and integration with FPGA vendors' IP libraries; (iii) advances in core synthesis and optimization, such as the use of the scheduling algorithm based on the system of difference constraints (SDC) [3, 4] and memory optimization [5]; and (iv) an integrated simulation and verification. Moreover, the paper reported detailed benchmark results of AutoPilot on multiple real-life examples by both BDTI [6] and Xilinx. The first benchmarking example was an implementation of the optical flow algorithm carried out by BDTI. Its result showed that on comparably priced consumer-grade FPGA and DSP targets, the AutoPilot implementation achieved an approximately 30 times better throughput per dollar than the optimized DSP implementation. For the amount of source modification, BDTI rated the DSP processor implementation "fair," whereas it rated the AutoPilot implementation "good," indicating that less source code modification was necessary to achieve high performance with AutoPilot. The second benchmarking was carried out by Xilinx on a sphere decoder implementation for a multi-input multi-output (MIMO) wireless communication system with about 4,000 lines of C code. Compared to expert-level manual RTL designs, AutoPilot used fewer resources while meeting the same performance target (225 MHz on a Xilinx Virtex-5 FPGA).

Encouraged by these results, the FPGA industry rolled out the HLS tools in the subsequent years. Xilinx introduced its C/C++-based HLS tool derived from AutoPilot in 2012 as Vivado HLS [7, 8] (and renamed it to *Vitis HLS* in 2019). Altera first introduced an OpenCL-based HLS tool named *Altera SDK for OpenCL* in 2013. After Intel's acquisition of Altera, Intel also offers a C++-based HLS tool named *HLS Compiler* [9]. HLS-based design methodology has been widely adopted, as evidenced by a large number of research publications in many fields, such as **machine learning (ML)** (e.g., [10–22]), bioinformatics (e.g., [23–28]), data processing (e.g., [29–32]), graph processing (e.g., [33]), image processing (e.g., [34, 35]), networking (e.g., [36]), scientific computing (e.g., [37–42]), security (e.g., [43]), and video processing (e.g., [44]).

In this context, it is timely to take a holistic view of the recent developments in the field of FPGA HLS. There have been several prior efforts that survey a comprehensive set of existing FPGA HLS tools, the associated programming models, as well as the underlying synthesis and optimization techniques [45–48]. In this article, we focus on assessing the progress of the *deployment* of HLS technology in the past decade (10 years after the predicted transition *from prototyping to deployment* as proclaimed in earlier work [1]) and laying out a road map for further innovation. First, we highlight the successes of HLS in several important application areas, including deep learning (Section 2.1), video transcoding (Section 2.2), graph processing (Section 2.3), and genome sequencing (Section 2.4). Then, we discuss the challenges faced by the HLS technology and the opportunities for further research and development, especially in the areas of achieving high clock frequency (Section 3.1), coping with complex pragmas and system integration (Section 3.2), legacy code transformation (Section 3.3), support **domain-specific languages (DSLs)** (Section 3.5), and the recent effort on open source HLS infrastructures (Section 3.4). It is our hope that this article can inspire more researchers to bring the HLS technology to a new height.

2 SUCCESSES

There are many successful FPGA applications and deployments using HLS, and it is not an easy job to select a few to highlight. After considerable deliberation, we chose the following four applications. The first application is deep learning (Section 2.1): this is one of the most active research areas for FPGA acceleration. We shall showcase some recent progress on ultra-low-bitwidth design

and implementation for deep learning acceleration. The second is video transcoding (Section 2.2): this is an area with considerable commercial success. We will highlight a real-time high-definition and high video quality HEVC [49] encoder built by NGCodec [50], which was acquired by Xilinx in 2019. The third application is graph processing (Section 2.3): this is a challenging application with highly irregular memory accesses and varying computation patterns. We will present the latest progress on this topic. The fourth is genome sequencing (Section 2.4): this is an important application in bioinformatics that has been accelerated on both FPGAs and GPUs. We use it as a case study to explain why FPGAs can outperform GPUs on many applications despite the considerably lower clock frequency in FPGA designs. The following sections give more details.

2.1 Deep Learning

In the past decade, deep learning has shown great success in many application areas, such as image recognition, video surveillance, and natural language processing. Hardware acceleration of deep learning inference has received much attention. One of the earliest, also probably the most cited FPGA-based deep learning accelerator, was published in early 2015 [10] based on the HLS technology to accelerate multi-layer **convolutional neural networks (CNNs)**, which is the computation kernel for most deep learning applications. The whole system was implemented in a single Xilinx Virtex-7 485T FPGA chip and used a DDR3 DRAM for external storage. A MicroBlaze, an RISC soft processor core developed for Xilinx FPGAs, was used to assist the accelerator startup, communication with the host CPU, and time measurement, and so forth. An AXI4lite bus was used for command transfer, and another AXI4 bus was for data transfer. The CNN accelerator worked as an IP on the AXI4 bus. This work was significant in several ways. First, using HLS, it was able to explore more than 1,000 accelerator configurations under the roof-line model and converge to a solution that is optimized both for computation and communication. Second, using HLS, one graduate student carried out the implementation in less than 6 months, ahead of the release of Google's TPU [51], an ASIC accelerator of deep learning, by a much larger design team. Third, it demonstrated close to a 5× speedup and a 25× energy reduction compared to a 16-thread CPU implementation. Inspired by this result, a large number of innovative deep learning accelerators were designed on FPGAs, and many of them were implemented with HLS using C++ (e.g., [52–54]) or OpenCL (e.g., [55–57]). We highlight two of the recent efforts in the rest of this section.

2.1.1 FINN. The underlying arithmetic of **deep neural networks (DNNs)** is structurally simple; however, their computational workload is enormous and comes along with equally challenging memory demands for storing and accessing the vast amount of model parameters associated with trained networks.

A feasible and promising approach to satisfy the cost and power constraints of a wide range of application domains is the quantization of inputs, activations, and model parameters to a reduced numerical precision. The resulting reduction in computes and memory requirements provides a range of design alternatives, allowing increased throughput and power savings to be balanced against a reduced accuracy. In some cases, quantized networks can also generalize better than non-quantized networks. FPGAs are ideal devices for exploring and building low-precision DNN inference engines. Given their bit-level customizability, FPGAs allow fine control over numerical precision in different parts of a system.

The FINN project [58] focuses on leveraging FPGA reconfigurability to map reduced–precision DNNs onto Xilinx FPGAs. This mapping is a direct mapping, where a specific network is implemented employing a dedicated composition of **Sliding Window Units (SWUs)**, **Processing Elements (PEs)**, and **Matrix-Vector-Threshold Units (MVUs)**, as shown in Figures 1 and 2. This direct mapping results in a dedicated implementation of the network at hand, leveraging the

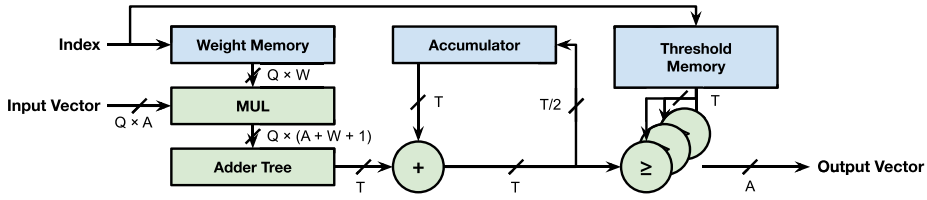
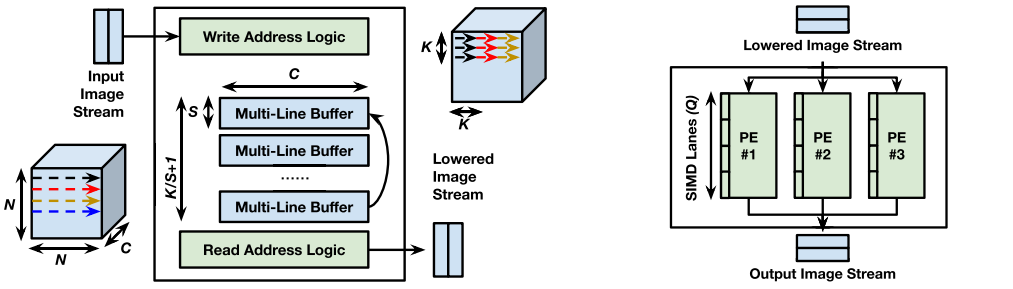


Fig. 1. PE as a basic compute component.



(a) Sliding Window Unit (SWU). (b) Matrix-Vector-Threshold Unit (MVU).

Fig. 2. SWU and MVU block diagram.

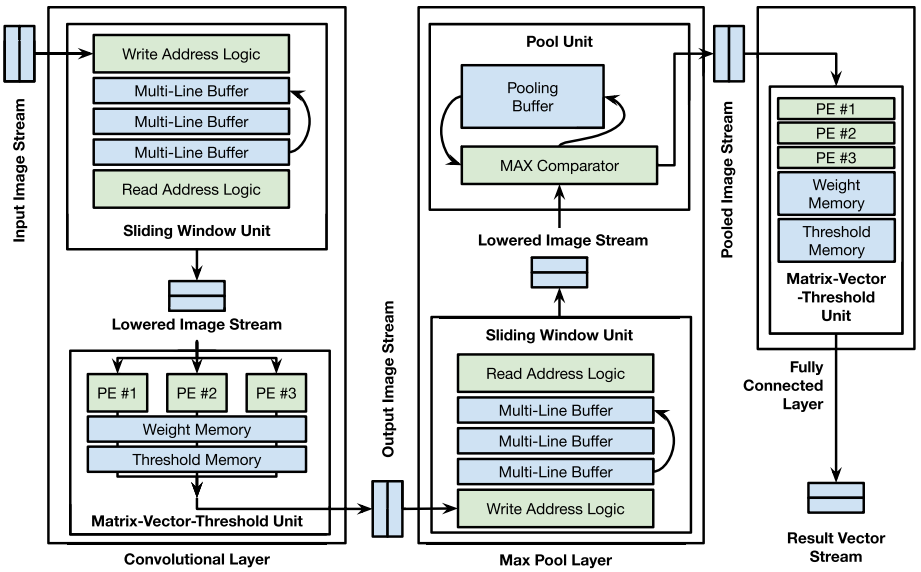


Fig. 3. Possible layer compositions.

FPGA programmability using HLS. In this approach, all layers of the DNN model are implemented concurrently in a dataflow style, where often all weights are stored on-chip. In this architecture, different layers in a quantized neural network can be readily implemented with different precisions. In addition, the latency/throughput tradeoff for different layers can be tuned, enabling the processing rate of different layers to be well matched. The resulting large design space for individual layers is implemented by automatically configuring a library of kernel functions combined into a dataflow pipeline, as illustrated in Figure 3.

The FINN tool is implemented as an ML-specific HLS system. The input to the FINN tool is a quantized neural network description created in PyTorch using the Brevitas library [59], exported as an ONNX representation. The FINN front-end creates a dataflow graph model of an ML network from the quantized neural network representation, given in ONNX. It then optimizes the graph, applying transformations that enable more efficient implementation of the network in terms of the available kernel functions. FINN then computes implementation parameters for each node in the graph so that the execution time of each layer is roughly the same, resulting in an efficient, balanced pipeline. The implementation parameters are also chosen to maximize performance given the resources available in a particular device. Each node is then implemented by synthesizing the corresponding templatized C++ code (consisting primarily of pipelined loops) with Vivado HLS. The FINN framework generates the HLS source code, using loops with UNROLL pragma and **initiation interval (II)** = 1. As a result, the user does not write or optimize HLS code directly. Some examples of the code can be found in GitHub [60]. The combined network can then be placed and routed into a design specialized for the original network.

The ability to optimize multiple processes in a dataflow model together provides the opportunity to automatically support a wide range of implementations with minimal user guidance, based on the ability to characterize the area/throughput tradeoff of the kernel functions. Many ML systems focus on the implementation of large, high-precision networks; however, FINN tends to focus on implementing quantized networks with very high throughput. Smaller networks can be completely unrolled and pipelined to the point where the complete network is evaluated every clock cycle (II = 1), resulting in more than 200M inferences per second and nanosecond-scale latencies in modern FPGAs. This capability, supported by HLS, has the potential to enable the use of ML in a wide variety of very high rate data processing systems. A related approach is taken in the ongoing work with hls4ml [61].

2.1.2 FracBNN. The previous discussions mainly focused on efficient implementations of existing ML models/networks on the FPGA. However, the FPGA hardware has unique advantages such as efficient support for bit-level operations and customizable memory hierarchies. Besides the quantization technique discussed in Section 2.1.1, these advantages also allow us to design FPGA-specific ML models. Here we present a case study of an algorithm-hardware co-design of the ML model tailored to the FPGA fabric.

There has been significant interest in accelerating deep learning applications on the FPGAs using **binary neural networks (BNNs)**. Since BNNs have 1-bit weights and activations, common operations such as convolutions are simplified into bitwise operations. These bitwise operations can be efficiently implemented on the FPGAs using the **lookup table (LUT)**-based logic. BNNs also reduce the memory requirement, making them one of the popular choices for an FPGA-based acceleration. Existing FPGA-based BNN implementations suffer from relatively low accuracy due to the reduced numerical accuracy. In addition, existing BNN models commonly use floating-point weights and activations in the input layer, leading to reduced resource efficiency because the floating-point resources cannot be reused with other layers. FracBNN [22] addresses these two challenges by (i) exploiting fractional activations to substantially improve the accuracy of the BNNs and (ii) binarizing the input layer using a novel thermometer encoding that retains high accuracy.

Specifically, FracBNN employs a dual-precision activation scheme where an additional binary convolution is computed if the particular convolution layer is likely to affect the final accuracy in a significant way. This is done by using a learnable threshold on the population count of the first binary convolution applied on the most significant bit of the activation. To binarize the input layer without significantly losing the accuracy, FracBNN increases the number of channels in the

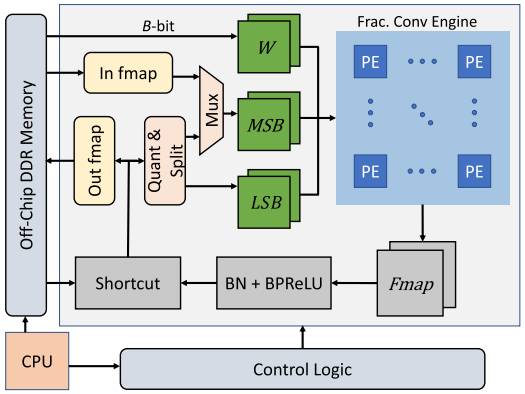


Fig. 4. FracBNN accelerator architecture.

input layer by treating each 8-bit pixel as an 8-dimensional binary vector. Instead of using the conventional fixed-point representations used by existing work [19, 62, 63], FracBNN encodes these input binary vectors using a novel thermometer encoding scheme to equalize the weights of different bit positions. Compared to the best-known BNN design on the FPGA, FracBNN improves the top-1 accuracy by 28.9% with a 2.5× reduction in model size on ImageNet. This is the first time that a BNN-based approach achieves an accuracy level that is comparable to MobileNetV2 [64] on realistic datasets. In addition, the authors successfully demonstrated applying FracBNN to high-accuracy, low-power, real-time image classification on an embedded FPGA device.

Figure 4 shows the overall architecture of the FracBNN accelerator. To execute a fractional convolution, the 2-bit feature maps and the 1-bit weights are fetched from the on-chip block RAM and the off-chip DDR memory, respectively. Since FracBNN processes the network layer by layer, it is able to store the low-precision feature maps in the block RAMs to improve the memory access latency. However, the 1-bit weights need to be stored off-chip due to the limited on-chip memory resources on the embedded FPGAs that FracBNN targets. After obtaining the feature maps, FracBNN first splits each 2-bit feature into MSB and LSB. It then packs the bits along the channel dimension into B-bit arbitrary precision integers for concurrent accesses. Since FracBNN loads the weights from the DDR, it must pack them into B-bit vectors to align the precision with the feature maps. To balance parallelism and resource utilization, it selects $B = 64$ for the CIFAR-10 design and $B = 32$ for the ImageNet design. The accelerator feeds the collection of weights and feature maps to the convolution engine. Meanwhile, it fetches the auxiliary parameters, including thresholds and weights in the BatchNorm; it also loads from DDR the activations of the fixed-point shortcuts, which are the residual connections adapted from Liu et al. [65].

HLS plays an important role in FracBNN because it significantly shortens the design time compared to the RTL-based methodology while achieving a satisfactory **quality of results (QoR)**. Two graduate students completed the entire FracBNN HLS design over the course of a few months.[1] In addition, HLS enables a fast exploration of various design choices such as operator fusion and buffer customization, which turns out to be crucial for the system performance. The existing overlay-based approaches [66, 67] commonly only support a predefined set of network primitives, which cannot easily execute new neural network models such as FracBNN with fractional activations. Compared to the overlay-based techniques, HLS provides a productive way to implement network-specific architectures on FPGAs.

[1]In contrast, it took nearly the same effort for another graduate student to implement a functional (but less optimized) RTL design for a single convolutional layer. A significant amount of time was spent on debugging the Verilog code.

2.1.3 HLS-Based DNN Compilers. Furthermore, given the growing interest in deep learning acceleration, a number of HLS-based domain-specific compilers have been developed, including systolic array compilers [41, 68–71] for CNN kernels, and end-to-end compilation for the entire neural networks, such as Caffeine [72, 73], DNNBuilder [74], and FlexCNN [75]. These compilers consider domain-specific or application-specific optimization and can achieve much improved results for ML acceleration. For example, the systolic array compiler AutoSA [41] uses the polyhedral model, explores a large solution space of systolic array designs, and achieves the best-reported INT-16 and INT-8 acceleration results so far on an FPGA. FlexCNN [75] can accelerate a complex neural network model OpenPose (with 86 convolution layers) for human pose recognition with consideration of layer-specific adaptive tiling, and achieve real-time performance with better energy efficiency than both CPUs and GPUs.

2.1.4 Challenges. The HLS-based design methodology enables the productive development of the aforementioned DNN accelerators. Popular DNN models typically feature regular compute and memory access patterns, which can easily be expressed by nested affine loops that are well supported by existing HLS tools. The commercial HLS tools also provide many useful language constructs (e.g., arbitrary-precision integer/fixed-point types) and optimization directives/pragmas (e.g., unrolling, pipelining, memory partitioning) that are essential for achieving fast DNN execution on FPGAs.

However, several hurdles remain before the users can enjoy an even shorter turnaround between ML model development and its deployment on the FPGAs. One challenge pertains to the verification methodology of the synthesized design. An HLS user typically uses C/RTL co-simulation to validate the functionality of the HLS outputs. In some cases, co-simulation is also necessary to obtain a more accurate performance estimate before the on-board testing. Unfortunately, even on a reduced test dataset, the co-simulation of a compute-intensive DNN model is quite a time-consuming process that could require many CPU hours.

The inaccurate QoR reporting by the HLS tool can be another factor that slows down the design iterations. In fact, the post-HLS resource usage often differs significantly from the actual result after low-level RTL synthesis and technology mapping [76]. This problem is more pronounced for quantized low-bitwidth DNNs, where an overestimation of LUT/FF counts may mislead the designer to choose a sub-optimal architecture with a low parallelization factor.

2.2 Video Transcoding

Video encoding is an interesting application for HLS and also represents significant challenges. The latest generation of codecs, such as HEVC and AV1, is quite complex, and designing and verifying the operation of encoders and decoders is a lengthy process. In addition, although the relevant standards and desired throughput largely dictate the requirements for video decoding, video encoding is much more open. Given the complexity of modern codecs, it is impractical to explore the entire coding space directly. Instead, the design of video codecs requires complex algorithmic tradeoffs to balance video throughput, encoded bitrate, resource usage, and video quality without exploring all coding options. This combination of algorithmic complexity, uncertain design tradeoffs, and a need for fast innovation make video encoding an ideal target for HLS [44].

The rest of this section describes the structure and implementation of a real-time, high-definition, low bitrate, high video quality HEVC [49] encoder built by NGCodec [50].[2]

2.2.1 Design Constraints. Overall, processing 1080P60 HD video with a resolution of 1920 × 1080 per frame requires processing (on average) of approximately 125 Mpixels/second. However,

[2]NGCodec was acquired by Xilinx in 2019.

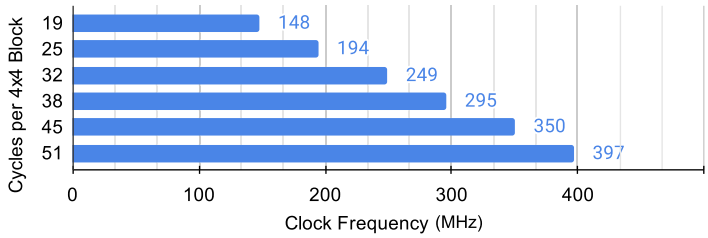


Fig. 5. Required HLS clock frequency target vs. number of cycles to process each 4 × 4 block for 1080P60 video.

```
1: for ctu in image do
2:     for (pipeline II=25) block in ctu do
3:         Read block
4:         Process block
5:         Write block
6:     end for
7: end for
```

(a) Simple coding style.

In this code, the processing that happens on each block is not described in a structured way. The intention to execute at a particular rate (one block every 25 cycles) is expressed, but this may be difficult for a tool to implement efficiently.

```
1: for ctu in image do
2:     for block in ctu do
3:         for (pipeline II=1) i ∈ [1, 25] do
4:             if i = 1 then Read block
5:             end if
6:             Process step i of block
7:             if i = 25 then Write block
8:             end if
9:         end for
10:    end for
11: end for
```

(b) Optimized coding style.

In this code, the processing of each block is explicitly broken into 25 steps that are highly similar. Each block will still take 25 cycles to process and can typically be implemented more efficiently by existing tools.

Fig. 6. Coding styles for blocked processing.

unlike pixel-based video processing where it is natural to target a clock frequency identical to the pixel rate, a block-based encoder has more freedom to decouple the clock rate from the pixel rate. HEVC is fundamentally structured around coarse granularity blocks of 64 × 64 pixels, called a **Coding Tree Unit (CTU)**. Each CTU must usually be completely processed at each pipeline stage of the encoder before the next CTU can be processed. In addition, when being coded, CTUs may be decomposed hierarchically at different granularities, from 32 × 32 blocks down to 4 × 4 blocks. Efficient encoding must make use of all of these decomposition options.

The NGCodec typically processes 4 × 4 blocks pixels as a unit, with most components targeting a clock rate of 200 MHz and 25 cycles per block, although some components are engineered to run at 300 or 400 MHz to reduce resource requirements. This rate is sufficient to support 1080P60 resolution (1920 × 1080 × 60), as shown in Figure 5. Generally speaking, each 16-pixel block must be processed at each stage of the encoder before the next 16-pixel block can be processed. Higher resolutions, such as 4K30, are achieved by processing more than one CTU in parallel.

Multiple coding styles are possible to achieve this block-processing style, as shown in Figure 6. A simple coding style is shown in Figure 6(a), requesting an II of 25 from HLS. This approach works well for functions where opportunities for resource sharing are relatively rare. It relies on the HLS tool to identify opportunities for resource sharing, which typically happens only at the granularity of individual operations. Figure 6(b) illustrates a more complex coding style. This coding style works well when an algorithm has a regular structure and can be expressed in a way where very similar processing happens during each step of the inner loop. The code explicitly describes resource sharing in the design by expressing the processing of each block as a number of steps, enabling the hardware implementation of the step (Figure 6(b), line 6) to be reused at a coarser level of granularity. Any muxing necessary would be generated only at the boundary of

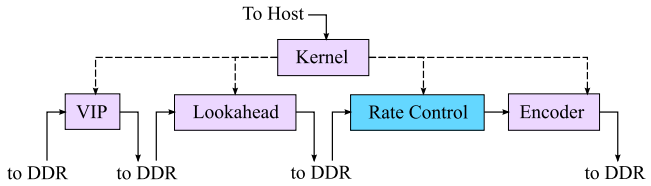


Fig. 7. Video encoder architecture.

the coarse-grained block. Note, however, that this coding style has less flexibility and can make it more difficult to explore different tradeoffs between II and clock speed. It can also be more difficult to automatically pipeline algorithmic dependencies between blocks.

Another important design constraint comes from the need for real-time processing. A non-real-time encoder can iterate sequentially over different encoding options to find a good parameter option, whereas a real-time encoder must narrow down the potential encoding choices to a few good candidates and evaluate those candidates. The more candidates that must be explored, the more resources must typically be used. This is particularly important when making bit-rate tradeoffs since the final number of encoded bits is typically hard to predict.

2.2.2 Architecture. Figure 7 illustrates the overall architecture of the encoder. The design consists of several top-level IP modules integrated with Vivado IP Integrator (IPI): the Kernel module, the **Video Interface Processor (VIP)** modules, the Lookahead modules, the Rate Control module, and the Final Encoder module. The top-level modules essentially implement a frame-level processing pipeline in this order with intermediate results stored in external memory buffers. These modules connect with the PCIe and DDR interfaces in the shell through several generic IP blocks implementing AXI interconnects, AXI streams, clock and reset generation, and so forth. In total, the design consists of about 100K lines of HLS C code and supports real-time encoding at 4K30 ($3840 \times 2160 \times 30$) on a Xilinx Alveo U200 card.

The Kernel module is a single HLS component, implementing an AXI slave interface visible to the host through memory-mapped access. Primarily, this block provides a communication interface for the host to configure and synchronize with the encoder. The host also has independent access to the DRAM interface, meaning that the video data itself does not pass through the Kernel module.

The Video Interface Processor module (another stand-alone HLS block) is responsible for reading input video data and converting it to a canonical YUV420 format used by the rest of the design. The VIP module reorders incoming data, enabling it to be processed as CTU blocks of 64×64 pixels, and writes the result back to DDR. The VIP block is also responsible for managing the placement of these buffers in memory and for multiplexing the processing of multiple video streams to be processed by the rest of the design.

The Lookahead module consists of 10 individual HLS blocks and is primarily responsible for performing coarse granularity motion estimation of the input content. This coarse motion estimation represents the basic temporal characteristics of the incoming video and can be used to detect scene changes. The motion vectors determined by this block are written back to DDR.

The Rate Control module executes once for each encoded frame to determine the encoding parameters for the next frame. The encoding parameters indirectly determine the cost (i.e., the number of bits) required to encode the resulting frame. To achieve a roughly constant output bitrate over short periods of time, the design implements a feedback control algorithm to adjust these parameters. This algorithm combines estimates from the Lookahead module with information about the number of bits utilized to encode the previous frame to determine the encoding parameters

of the current frame. Because of the relatively low processing rate of this block, a MicroBlaze soft processor implements the control algorithm instead of a specialized HLS circuit.

The Encoder module reads CTUs along with coarse motion vectors from DDR and properly encodes them. The resulting compressed bitstream is stored back in DDR. The Encoder module consists of approximately 20 individual HLS blocks. This module computes candidate motion estimation vectors and estimates the cost of encoding a CTU using each motion vector. It also generates candidate encodings using intra-frame prediction and estimated encoding costs for CTUs that cannot be predicted well with motion estimation. From these different encoding options, the module selects the lowest cost encoding for each CTU. Note that the choice of the lowest cost encoding for an entire CTU must consider many possible hierarchical decompositions of the CTU into smaller blocks. This process determines a predictor and a residual for the CTU. The encoder transforms and quantizes the residual using a discrete cosine transform and recreates and stores the decoded result locally to correctly determine predictions for future blocks and frames. Last, the encoder applies an entropy code to generate the final encoded bitstream. The entropy code accounts for the statistics of the encoded bitstream, enabling the use of fewer bits to represent the most likely representations and more bits to represent less likely representations.

Note that the overall performance of this design relies on constructing a balanced pipeline between different processing blocks. Unlike a typical component in a sequential program where faster is almost always better, the components in the encoder are designed to operate concurrently at certain important frame rates and resolutions with real-time requirements in mind. Balancing performance with area usage and avoiding data bottlenecks by keeping data local to the device is more critical to the design than absolute performance. A typical profiling-based approach of finding the longest-running task in the encoder and offloading it to the FPGA does not work well because of Amdahl's law; there is no single component of the design that dominates the computation time.

2.2.3 Challenges. Large designs like video encoders include many design challenges. Some of these challenges are unique to using HLS; however, others are common for hardware design, either with FPGA or ASIC technology. We anticipate that future HLS research could address these issues.

Many challenges arise simply from the size of the design. Given the large codebase and many independent components, the hierarchical design flow adopted by NGCodec, where changes to individual components can be isolated from each other, is valuable. This approach speeds up tool execution for each component, enabling faster design iteration and optimization. A hierarchical design approach also enables independent verification of components, since each component can be simulated at the RTL level using the C models of other components. In addition, overall timing closure can be simplified by first ensuring that each component meets timing in isolation before being integrated in a final design. Unfortunately, the separation between IP component composition and the top-level C++ simulation model does provide an opportunity for inconsistencies to occur. Although some HLS tools have included explicit hierarchical concepts, such as Synfora's Tightly Coupled Accelerator Block (TCAB), we expect a need for widespread use of these concepts, particularly to reduce discrepancies between hardware design flows and HLS at the top level.

However, even given a hierarchical design flow and reliance on relatively fast C/C++ simulation, simulation time can still be a bottleneck. Simulating the execution of the NGCodec design requires several minutes for a single 1920×1080 video frame on a modern server. In addition, the motion estimation and rate control functionality require five to eight frames of input data before any output is created. The simulation speed is limited primarily by the fact that synthesizable code is typically only compiled onto a single core by current compilers, which lack the ability to leverage multiple cores available in modern systems. In addition, synthesizable code is often optimized for HLS implementation and existing compilers are typically poor at generating highly

optimized vector instructions for simulation from it. Although simulation with smaller frame sizes can reduce the amount of data to be processed, future tools would likely benefit from infrastructure to make better use of fine-grained and coarse-grained parallelism during simulation. This capability may come hand-in-hand with the adoption of more concurrent programming models, such as the Adaptive Dataflow programming API in the Xilinx Vitis tools [77].

Another challenge emerges because different components of the design have vastly different compute and memory requirements. For instance, some portions of the design performing motion estimation require large on-chip memories but relatively simple computation, whereas other portions of the design performing encoding transformations, like the discrete cosine transform, necessitate significant logic resources but relatively little memory. Although each component of the design can be relatively easily implemented out of context in the target FPGA, combining different modules in the same devices increases placement constraints and makes the overall design challenging to place and route. Ideally, future HLS tools will better assess how designs map to large complex devices. We discuss some recent approaches to this challenge in Section 3.1.

2.3 Graph Processing

Besides deep learning and video transcoding, FPGA-based graph processing is also gaining traction as an alternative to CPU- or GPU-based solutions. Graphs are widely employed to represent applications in various domains such as social networks, road networks, and biological systems. Graph processing is typically memory-bound due to low compute-to-memory ratio and irregular memory access patterns. Modern server-class FPGAs are commonly equipped with **high-bandwidth memories (HBMs)** [78], where multiple HBM channels can be concurrently accessed to significantly boost the performance of graph processing. In addition, FPGAs have the flexibility to customize the memory hierarchy and the data layout to fit the application logic, which can improve the design throughput and/or energy efficiency beyond what is achievable from CPUs/GPUs. A rich set of prior FPGA-based graph processing works targets a specific graph algorithm, such as BFS [79, 80], PageRank [81, 82], and shortest path [83, 84]. Recently, ENIAD [85] reported record-breaking results on the Green Graph500 benchmarks using a hardware-software co-design approach to near-data processing. This work demonstrates the performance and efficiency advantage of using FPGAs for graph analytics on realistic workloads. FPGA-based graph processing frameworks exist that can handle multiple graph algorithms. Examples include GraphGen [86], GraphOps [87], HitGraph [88], and ThunderGP [33]. Notably, ThunderGP uses HLS to implement the graph accelerator on FPGAs; it employs the gather-apply-scatter abstraction [89] to model various graph algorithms and realizes the model through an efficient accelerator template. However, these frameworks require generating separate bitstreams for different graph algorithms.

GraphLily is a graph linear algebra overlay designed in HLS that can achieve efficient and practical acceleration of graph processing workloads on HBM-equipped FPGAs [90]. GraphLily has a unique advantage over the existing approaches—it provides a unified bitstream to handle multiple graph algorithms instead of generating a separate bitstream for each algorithm. GraphLily supplies efficient, memory-optimized **sparse-matrix dense-vector multiplication (SpMV)** and **sparse-matrix sparse-vector multiplication (SpMSpV)** accelerators working on graph data in the form of a novel sparse matrix storage format. Hu et al. [90] designed these accelerators using HLS, which significantly reduces the development time compared to RTL designs. In addition, it presents a programming interface similar to GraphBLAS [91] that allows streamlined porting of existing graph applications to the FPGA accelerator. The GraphLily framework supports a wide range of graph applications by converting different workloads into the same underlying computation patterns, which does not require generating/loading separate bitstreams for different applications. Compared to the state-of-the-art CPU and GPU implementations of the same application,

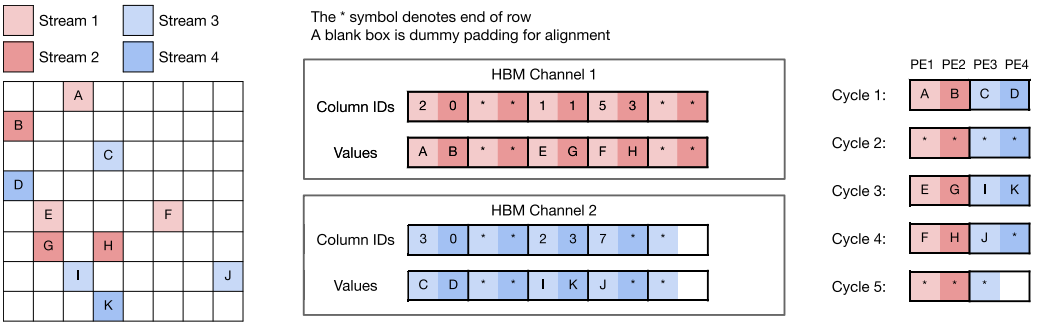


Fig. 8. Example of the CPSR sparse matrix storage format with two HBM channels. (a) A sample 8×8 sparse matrix. (b) The layout of the sparse matrix. Two streams of rows are packed together and stored in one HBM channel. (c) Memory accesses at every cycle. Only accesses to values are displayed; accesses to column IDs follow the same pattern. CPSR enables streaming, vectorized access to each channel, and concurrent access to two channels. CPSR allows four PEs to run in parallel.

GraphLily achieves both better throughput and energy efficiency. The rest of this section describes the techniques in GraphLily and detailed comparisons with other implementations.

2.3.1 Customized Matrix Storage Format. GraphLily adopts an architecture based on the notion of PEs. It distributes the matrix-vector multiplication task across multiple PEs, and each PE is responsible for executing part of the task in a parallel fashion with respect to the other PEs. Traditional sparse matrix storage formats such as compressed sparse row format (CSR) are not suitable for exploiting the parallelism in the context of PE-based architectures. This is because CSR allocates data needed by different PEs at non-consecutive locations in memory, preventing intra-channel vectorization and inter-channel concurrent access to the HBMs. The ELLPACK format [92] partially solves this problem by padding the matrix rows but incurs significantly high storage overhead when the degrees of the matrices are not uniform—which is true for most real-world graphs.

One key contribution of GraphLily is a novel storage format called **cyclic packed streams of rows (CPSR)** designed specifically for HBM-equipped devices. CPSR allows streaming, vectorized memory accesses to each HBM channel by packing consecutive rows, and concurrent memory accesses to multiple HBM channels by interleaving the row packs in a cyclic manner. Figure 8 shows an example of CPSR applied on an HBM with two channels and targeting a 4-PE accelerator for SpMV. In this scheme, a dedicated PE processes each row of the matrix, and GraphLily assigns the task of processing each row to the available PEs in an interleaved fashion. As the execution timeline shows in Figure 8(c), the PEs are able to fully utilize the available memory bandwidth from the HBM channels, thus achieving high throughput. Unlike several existing sparse formats, converting a graph from CSR to CPSR is lightweight since it does not sort and reorder the vertices. Hence, the preprocessing cost can be further amortized over multiple iterations, runs, and algorithms.

2.3.2 Accelerator Architecture. GraphLily implements an FPGA overlay that executes a set of computing primitives commonly found in graph analytics applications. Figure 9 shows the architecture of the SpMV accelerator in GraphLily. For each HBM channel, a cluster of PEs is instantiated to saturate the memory bandwidth. Each PE is a three-stage pipeline that reads out the value from the output buffer, updates the result, and writes back the updated value. The SpMV accelerator is fully designed with HLS, significantly reducing the design effort. Similar to GraphBLAS, the PE

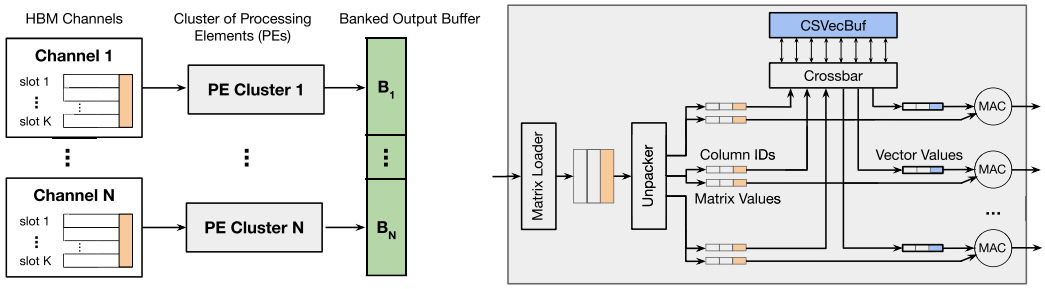


Fig. 9. Architecture of the SpMV accelerator in GraphLily. (a) Overall structure. (b) The PE cluster.

```
DenseVector bfs(SparseMatrix Adj, int src,
                int num_iter) {
  // Initialize the frontier vector
  SparseVector frontier = {src};
  // Initialize the distance vector
  DenseVector distance(Adj.num_rows);
  for (int i=0; i<Adj.num_rows; i++)
    {distance[i] = -1;}
  distance[src] = 0;
  for (int iter=1; iter<=num_iter; iter++) {
    // Perform graph traversal
    // Use SpMV or SpMSpV depending on the
    // frontier size
    if (frontier.size > threshold) {
      frontier = graphblast::SpMV<BoolSemiring>(
                 Adj, frontier, distance);
    } else {
      frontier = graphblast::SpMSpV<BoolSemiring>(
                 Adj, frontier, distance);
    }
    // Update distance
    graphblast::Assign(distance, frontier, iter);
  }
  return distance;
}
```

(a) GraphBLAST.

```
// A graph algorithm is expressed as a collection
// of modules
class BFS : graphlily::ModuleCollection {
  // Specify the modules and load the bitstream
  void init() {
    this->SpMV = graphlily::SpMVModule<BoolSemiring>;
    this->SpMSpV =
            graphlily::SpMSpVModule<BoolSemiring>;
    this->Assign = graphlily::AssignModule;
    this->load_bitstream(
            "graphlily_overlay.bitstream");
  }
  // Format the matrix and send it to the device
  void prepare_matrix(SparseMatrix Adj) {
    AdjCPSR = this->SpMV.format(Adj);
    this->SpMV.to_device(AdjCPSR); //to HBM
    AdjPackedCSC = this->SpMSpV.format(Adj);
    this->SpMSpV.to_device(AdjPackedCSC); //to DDR
  }
  // Compute BFS by scheduling the modules
  // The logic is the same as in GraphBLAST
  DenseVector run(int src, int num_iter) {
    ...
  }
};
```

(b) GraphLily.

Fig. 10. Example code of implementing BFS in GrahBLAST [91] vs. GraphLily.

supports three types of semirings that the user can select at runtime to execute a wide range of applications such as PageRank, breadth-first search, and single-source shortest path.

In addition to SpMV, GraphLily also implements an SpMSpV accelerator that reads the matrix from the DDR instead, which provides sufficient bandwidth as SpMSpV has a lower degree of parallelism and it is less bandwidth-hungry. For SpMSpV, the compressed sparse column format is used, and the accelerator implements an arbitrated crossbar to dispatch the elements of the input matrix to the corresponding rows based on their row IDs.

2.3.3 Runtime Support. GraphLily builds a layer of middleware that eases the programming of the accelerators. The middleware exposes each accelerator to users as a hardware module. Users construct graph algorithms by specifying the necessary modules and scheduling the execution order of the modules. To streamline the design process, GraphLily exposes to the user a set of APIs that resemble the GraphBLAST APIs. The exposed APIs can be used to manage the data transfers to and from the FPGA device, as well as invoking various compute tasks on the device. Figure 10 illustrates an example of specifying a BFS algorithm using the GraphLily APIs.

Table 1. GraphLily's SpMV Throughput (in MTEPS) and Bandwidth Efficiency (MTEPS/(GB/s)) Compared to CPU (MKL) and GPU (cuSPARSE) Implementations

Dataset	Throughput (MTEPS)			Bandwidth Efficiency (MTEPS/(GB/s))		
	MKL	cuSPARSE	GraphLily	MKL	cuSPARSE	GraphLily
googleplus	2,542	13,643	7,002	9.0	28.2	24.6
ogbl-ppa	2,065	9,007	8,492	7.3	18.6	29.8
hollywood	2,202	11,277	8,736	7.8	23.3	30.7
pokec	1,504	5,271	4,064	5.3	10.9	14.3
ogbn-products	1,556	2,501	6,434	5.5	5.2	22.6
orkut	1,807	5,332	6,973	6.4	11.0	24.5
Geometric mean	1,912	6,783	6,751	6.8	14.0	23.7

2.3.4 Comparisons with Other Approaches. Table 1 summarizes the experimental results of GraphLily compared to the state-of-the-art CPU and GPU counterparts. The CPU and GPU implementations use vendor-provided sparse libraries, including MKL (2019.5) on the CPU and cuSPARSE (10.1) on the GPU. The CPU experiments are conducted on a two-socket 32-core 2.8-GHz Intel Xeon Gold 6242 machine with 384 GB of DDR4 memory providing 282-GB/s bandwidth. For GPU experiments, a GTX 1080 Ti card is utilized, which has 3,584 CUDA cores running at a peak frequency of 1,582 MHz and 11 GB of GDDR5X memory providing 484-GB/s bandwidth. The datasets are taken from widely used benchmarks in social networks and graph neural networks.

GraphLily achieves a geometric mean throughput of 6,751M of **traversed edges per second (MTEPS)**, which is 3.5× higher than MKL running with 32 threads and matches cuSPARSE. The geometric mean bandwidth efficiency of GraphLily is 23.7 MTEPS/(GB/s), which is 3.5× higher than MKL and 1.7× higher than cuSPARSE. In terms of power consumption, GraphLily consumes 44 W, which is only 16% of MKL and 29% of cuSPARSE. The reason for the performance/efficiency advantage of GraphLily is twofold. First, the CPSR format can better utilize the HBM bandwidth than the CSR format. Specifically, CPSR enables vectorized access to each HBM channel and concurrent access to multiple HBM channels. Second, the hardware architecture is co-designed to exploit the advantages of the CPSR format. The fully pipelined PE and the vector buffers ensure that the packets received from HBM are immediately processed at high efficiency.

2.3.5 The Role of HLS. The GraphLily overlay is entirely implemented in Vivado HLS using 3,500 lines of HLS C++ code. The HLS-based design methodology offers significantly higher productivity by allowing the designer to focus more on the high-level algorithmic and architectural design without worrying about low-level implementation details. In addition, the tight integration of Vivado HLS within the Vitis toolchain allows streamlined host-device co-design. The HLS-generated accelerators are readily plugged into the Xilinx Alveo U280 FPGA platform using the GraphLily middleware based on the Xilinx Runtime Library [93].

However, designs generated by the current HLS tools often suffer from clock frequency degradation due to long wire delays caused by broadcast structures [94]. In the GraphLily design, the large output buffer incurs a high-fanout broadcast structure. The authors follow the method proposed in the work of Guo et al. [94] to reduce wire delays utilizing a pipelined multi-level tree structure. This optimization increases the frequency from 145 to 165 MHz. Besides high-fanout nets, HBM and the multi-die architecture bring additional challenges to achieving high frequency with HLS. A large HLS design utilizing multiple SLRs (super logic regions) can generate cross-SLR data and control signals. Current HLS tools are often not able to accurately identify such cross-SLR connections due to the lack of physical layout information during HLS. Such under-pipelined signals commonly become the clock frequency bottleneck of the whole design. It is evident that improving HLS tools' timing closure capability, especially for large and/or highly congested designs, is

crucial for wider adoption of the HLS-based design methodology. We shall discuss this in more detail in Section 3.1.

2.4 Acceleration of Genome Sequencing

Whole-genome sequencing is the process of obtaining the order of nucleotide base pairs of an organism's genome. At the biochemical level, technologies produce sequencing results for unordered fragments of the whole genome, which are called *reads*. Assembling those reads into a contiguous sequence is often a computational bottleneck in genome sequencing [95]. There are typically three stages in a modern sequence assembly flow: Overlap, Layout, and Consensus [96, 97]. The Overlap stage determines which pairs of reads have matching sub-sequences. The Layout step concatenates continuous reads using the overlap information to build the result sequences. Finally, the Consensus step picks the most likely nucleotide sequence.

FPGA-based accelerators have been proposed for all three stages, many of which are reported to outperform their GPU counterparts [25, 98–101]. Although some efforts [28, 102–104] implement their proposed accelerators in RTL, we see a trend toward utilizing HLS due to the high complexity of genome applications, especially when algorithmic optimizations are involved [23, 25, 26, 101, 105–107]. For example, Guo et al. [25] transform the algorithm of the overlap stage into a functionally equivalent yet HLS-friendly version to facilitate hardware acceleration. Lo et al. [26] performed hardware-algorithm co-design and memory optimizations to speed up the consensus stage.

Among these works, we focus on the overlap acceleration work by Guo et al. [25] as a case study for three reasons. First, detecting overlaps between read pairs is the most time-consuming task in the flow. It could take several CPU months to detect the overlaps in the human genome. Guo et al.'s FPGA accelerator achieves a 28× speedup over the multi-threading CPU baseline. Second, they achieved a fully pipelined streaming architecture on the FPGA with minimal and readable code changes, which is a good example of HLS's success. Third, they performed a quantitative architectural comparison with a CUDA-based GPU implementation and provided insights on choosing the right hardware acceleration platform for new applications.

2.4.1 *Acceleration Design.*

Target task. Guo et al. [25] accelerate Minimap2 [108]. Minimap2 is a state-of-the-art genomics tool that excels in speed and QoR. They focus on the *chaining* [109, 110] step, which is a bottleneck in the tool. In the steps before chaining, short exact *matches* between read pairs are extracted. The chaining step performs dynamic programming to find sequences of matches with consistent distances on two reads. When two reads have a long sequence of matches (i.e., a long *chain*), it implies that they may originate from the same sub-sequence of the whole genome.

In the chaining algorithm [108], a *match* between read a and b is represented by a 3-tuple of (x, y, l), describing an exact match of length l: "$a_{x-l+1} \dots a_x$" = "$b_{y-l+1} \dots b_y$." Those matches are sorted by x and then by y and the chaining score $f(t)$ representing the quality of the best chain ending at match t is determined. The chaining score $f(t)$ is calculated by evaluating the potential quality of continuing chains from its 64 previous matches s:

$$f(t) = \max\{l(t), \max_{s<t}\{f(s) + w(s,t)\}\}$$

$$w(s,t) = -\beta(s,t) + \alpha(s,t),$$

where $l(t)$ is the length of match t, $\beta(s,t)$ quantifies the inconsistency in location between matches s and t, and $\alpha(s,t)$ is the contribution of match t to the total matching length if it is added to a chain ending at match s. The details of function β and α are omitted for simplicity. Pseudocode

```
Input: match[]: ordered list of matches between a pair of reads
Output: π[]: the predecessors and f[]: the chaining scores.
1: for t = 1 to n do
2:     for s ∈ [max(0, t − 64), t − 1] do
3:         w[s][t] ← (−β + α)(match[s], match[t])
4:     end for
5:     f[t] ← max_j {f[s] + w[s][t]}
6:     π[t] ← arg max_j {f[s] + w[s][t]}
7:     if f[t] < match[t].l then
8:         f[t] = match[t].l; π[t] = ∅
9:     end if
10: end for
```

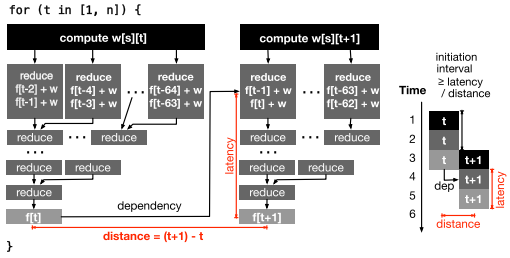
(a) Original chaining algorithm.

```
Input: match[]: ordered list of matches between a pair of reads
Output: π[]: the predecessors and f[]: the chaining scores.
1: Initialize all f[t] ← match[t].l, and π[t] ← ∅ in parallel.
2: for (pipeline) s = 1 to n do
3:     for (parallel) t ∈ [s + 1, min(n, s + 64)] do
4:         w[s][t] ← (−β + α)(match[s], match[t])
5:         if f[t] < f[s] + w[s][t] then
6:             f[t] = f[s] + w[s][t]; π[t] = s
7:         end if
8:     end for
9: end for
```

(b) Optimized chaining parallelization.

```
Input: match[]: ordered list of matches between a pair of reads
Output: π[]: the predecessors and f[]: the chaining scores.
1: for t = 1 to n do
2:     for (parallel) s ∈ [max(0, t − 64), t − 1] do
3:         w[s][t] ← (−β + α)(match[s], match[t])
4:     end for
5:     Reduce 64 values of f[s] + w[s][t] to 32 in parallel.
6:     Reduce 32 values to 16 in parallel.
7:     Reduce 16 values to 8 in parallel.
8:     Reduce 8 values to 4 in parallel.
9:     Reduce 4 values to 2 in parallel.
10:    Reduce 2 values to 1 to set f[t] and π[t].
11:    if f[t] < match[t].l then
12:        f[t] = match[t].l; π[t] = ∅
13:    end if
14: end for
```

(c) Poorly parallelized chaining.



(d) Example of loop pipelining of the outer loop in Algorithm 11(c). There exists a dependency, and therefore it has a minimum initiation interval of *latency*/1.

Fig. 11. Different implementations of the chaining algorithm.

for this algorithm is shown in Figure 11(a). Lines 2 through 4 compute the weight $w(s,t)$. Line 6 finds the maximum score of $f(s) + w(s,t)$ in the inner max of the equation, and line 6 records the corresponding s as the predecessor of the chain. The remaining code from lines 7 through 9 computes the outer max of the equation and records the predecessor as $\emptyset$ if $l(t)$ is greater.

Guo et al. [25] identify several optimizations to optimize the basic chaining algorithm. These optimizations are portable and can largely be applied to both FPGA and GPU designs.

A. Intra-task parallelism. A straightforward optimization leverages a six-stage reduction tree to obtain the maximum score as shown in Figure 11(c). The reduction operations from lines 5 through 10 compute the maximum of each pair of values in parallel. However, this code is still sub-optimal because those six reduction stages have to be executed sequentially, preventing pipelining of the outer loop (line 1). This pipeline's II has a recurrence lower bound [4] of $II \geq latency/distance = latency$ as shown in Figure 11(d), which is the latency of seven comparisons.

Alternatively, the order of computation can be changed by interchanging the two loops. At each iteration of the outer loop, a set of 64 running maximums is updated (lines 3 through 8), and the final value of one $f(t)$ is computed (line 2). Figure 11(b) displays the optimized algorithm, where a full parallelization of the inner updating loop (line 3) is possible. The new loop recurrence is that $f[i]$ (line 5) requires $f[k]$ from preceding iterations (line 6). Therefore, the II is bounded by only the latency of a single comparison.

B. Data reuse. Without data reuse, achieving a performance of 1B iterations per second in this application requires a memory bandwidth of 7,345 Gbps—far exceeding any off-the-shelf solutions. However, by storing the metadata, scores, and predecessors of matches i+1 to i+64 in the local

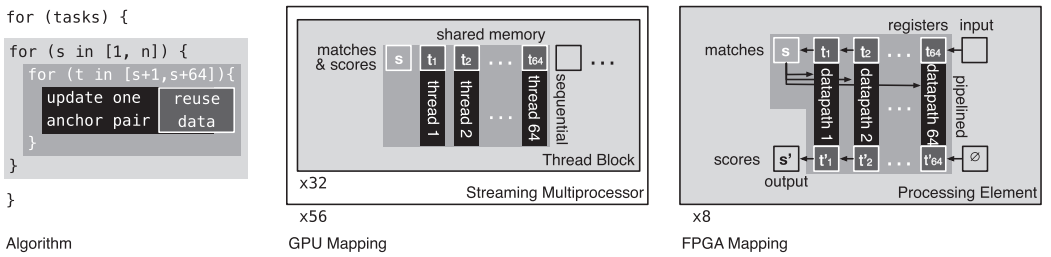


Fig. 12. Different acceleration implementations of Algorithm 11(b) on the GPU and the FPGA.

memory of the accelerator, only 14 bytes of main memory are accessed per iteration. This dramatically reduces the required bandwidth from the original 7,345 Gbps to 112 Gbps.

C. Task-level parallelism. At the task level, pairwise tasks are independent and can be executed in parallel. However, it is impractical to solely rely on task-level parallelism. Otherwise, the reuse data will overflow the accelerator's on-chip storage. On top of the overflow issue, there is another challenge: if two reads do not overlap, they will have few matches, usually less than 10. Those fruitless tasks incur overhead that hurts efficiency. Guo et al. [25] address these problems by leveraging multiple levels of parallelism. They further concatenate different tasks and separate them with tags at runtime, thus decreasing the overhead of the fruitless tasks.

2.4.2 Implementation. This section describes how the preceding optimizations can be implemented differently on the FPGA and the GPU. The architectural difference is illustrated in Figure 12.

Data reuse in the acceleration design forms a concurrent-accessed sliding window of length 64. In the FPGA design, registers store the matches and the scores as the window. Each set of registers stores one element, enabling parallel access in each clock cycle. However, in a GPU implementation, shared memory spread across different banks enables parallel access. Both FPGA and GPU designs reuse the data identically and distribute it for optimal access.

Intra-task parallelism, specifically the pipelining and the parallelization of the loops, are designed as a pipeline and parallel datapaths on the FPGA, respectively. As demonstrated in Figure 12, at each cycle, one datapath collects one match from the input window and the i-th match to calculate the new score to update the output window. Those datapaths are executed in parallel. The overall architecture of one PE handles iterations of the outer loop in a fully pipelined fashion. At each cycle, a PE takes an input *match* and produces one maximal *score*. On the GPU, the parallelized inner loop became multiple threads that each thread updates one score. This decision was reasonable as the large data size for reuse would prevent the outer loop from being mapped to the threads. Although the outer loop could be implemented as software pipelining on the GPU, the authors found no performance benefit experimentally. Sufficient degrees of intra-task level parallelism were used in the implementation and could be an explanation for this.

Task-level parallelism on the FPGA is accomplished by duplicating the PE by eight. Since each PE is fully pipelined, the array of PEs processes eight matches from different tasks in each clock cycle, generating eight corresponding scores. For optimal PE utilization, a task scheduler adaptively dispatches tasks to the idle PEs. For GPU, each task is assigned to a thread block.[3] On an NVIDIA P100 GPU, there would be around 56 tasks executed in parallel. By interleaving 32 tasks, it is possible to hide the instruction latency and keep a high compute resource occupancy.

[3]In GPU architecture, a thread block represents a group of threads that share data [111]. GPU dispatches thread blocks to streaming multiprocessors that have sufficient storage and computational resources.

2.4.3 Success in Performance. It is long debated whether we should use FPGAs or GPUs for acceleration. We find this genomics application suitable for a side-by-side comparison because the optimization methods employed are general enough to be ported to a GPU without compromise.

For a fair comparison, Guo et al. [25] use an FPGA and a GPU of the same 16-nm process generation. They perform the FPGA experiment on a Xilinx Virtex UltraScale+ VU9P FPGA and implement the GPU kernel on an NVIDIA Tesla P100 GPU. Both accelerators are connected to the host by PCIe Gen 3 × 16 links. On a 4.6-GB public *C. Elegans 40× Sequence Coverage* data [112] obtained from a PacBio sequencer, they achieve 250 MHz with eight FPGA PEs. This design saturates the PCIe bandwidth and executes in 6.84s, 21× faster than the multi-thread software on a 14-core Xeon E5-2680v4 CPU (142.5 seconds). As a proof-of-concept for future interconnects, the authors further increase the PE number to 16 while lowering the frequency to 200 MHz. Assuming unlimited host-device communication bandwidth, the same task would execute in 5.13 seconds, 28× faster than a CPU. However, the GPU kernel is around 4× slower than the FPGA design, executing in 20.09 seconds on the same data.

It is intriguing to note that the theoretical analysis of Guo et al. [25] hypothesizing that an FPGA could indeed achieve a higher performance than a GPU is in agreement with the experimental results. In summary, their fully pipelined FPGA design allows the processing rate to be 16 scores at 200 MHz with 16 PEs, which is 3,200M scores per second. However, even if the GPU computing resource is perfectly utilized, at least 49 cycles per thread on 64 CUDA threads are required to obtain one score. With all 3,584 CUDA cores ideally utilized at 1,303 MHz, the estimated best performance is $3,584/49/64 \times 1{,}303 = 1{,}489$M scores per second, more than 2× lower than what an FPGA achieves.

Since instruction count is the main factor in the performance analysis, Guo et al. [25] study the compiler-generated GPU instructions and find 49 instructions mandatory and challenging to simplify. Among them are 21 arithmetic operations, 20 control instructions, and eight memory accesses. This helps us to understand an FPGA's advantage over a GPU in the following dimensions.

Control logic. A total of 40% of the instructions in the GPU kernel are control instructions. With an instruction-based processor, control instructions are inevitable and represent overhead if they cannot be executed concurrently with data processing instructions. However, on an FPGA, these control operations are implemented using customized logic, which is usually integrated into pipeline stages with limited impact on the overall throughput. For example, determining if it is better to start a new chain than continue a previous one requires three control instructions (including predicated instructions) on the GPU. In contrast, only one pipelined multiplexer stage is generated for the FPGA, which adds at most one cycle of latency. To determine if an application falls into this category, one may check if there are a lot of conditional branching or predicate operations.

Non-standard bitwidth usage. Among the 20 arithmetic operations, most of them are operating on 17-bit data. Although the GPU units are capable of processing 64/32-bit floating-point numbers with exceptional performance, GPUs support a limited set of reduced bitwidth operations. In contrast, FPGAs can perform arbitrary bitwidth operations using LUT resources and optimized wider precision arithmetic using DSP resources. Implementing designs with reduced bitwidth operations can significantly reduce resource usage and allow more PEs to fit into a device.

Structured data types. FPGAs can store structured data in multiple BRAM banks or registers, enabling access to multiple fields concurrently. In contrast, on a GPU, multiple fields must be fetched and unpacked with several instructions. When an application processes structured data and many fields are utilized altogether, FPGA acceleration is likely to be suitable. For example, the FPGA implementation of this algorithm accesses the (x, y, l) fields in the match metadata in parallel on every clock cycle.

2.4.4 Challenges in Development. We analyze Guo et al. [25] open source code on GitHub[4] as a case study and present the strengths and weaknesses of the FPGA HLS and the GPU CUDA in terms of development effort. While reviewing some successes of the high-level abstraction power of HLS, we identify several improvements needed to broaden the spectrum of the FPGA HLS users.

Quantitatively, to implement the computational kernel, the effort difference in terms of **lines of code (LOC)** between the FPGA HLS and the GPU CUDA is insignificant. The main function to compute the scores has 60 LOC in HLS and 64 LOC in CUDA with the same functionality, excluding empty lines and comments. However, the majority of the HLS code changes from the original code are compiler directives, a.k.a. *pragmas* (14 LOC), and extra local variables (9 LOC). The semantics of the HLS code is equivalent to a sequential program, even though code restructuring techniques, such as loop transformations, are required. In contrast, the GPU code changes are mostly architecture-specific (28 LOC), such as inserting multi-threading and warp-level primitives. Without any platform-specific knowledge, one could easily understand the functionality of the HLS code, but it might be challenging to understand its CUDA counterpart.

The simpler *programming model* of HLS also allows *nested parallelism* with arbitrary levels and *functional simulation* of HLS code on a CPU with a regular compiler. *Streaming and pipelining* can be expressed in HLS as queues or pragmas. Pipeline stages will be automatically inferred. Although Udupa et al. [113] attempted to simplify the software pipelining on a GPU, pipeline programming remains laborious. *Memory customization* in HLS is also able to generate a distributed memory architecture customized to fit the particular program.

However, the complete FPGA HLS implementation has 524 LOC in total, which is 3.7× the CUDA code (142 LOC). Among them are around 84 LOC of data type conversions for host-device communication, 48 LOC of compiler directives for hardware specification, 46 LOC of double buffering, 38 LOC of manual registering for boosting frequency, 32 LOC of PE duplication, and many other trivial details. The reason for those efforts is that to write a *vanilla* HLS program, such as for the Xilinx HLS or Intel HLS compilers, a significant amount of glue code is necessary to make the host-device communications, intra-device data accesses, layout-aware registering, and many other details for the code to be efficient [94, 114]. Fortunately, recent work finds [115, 116] that the glue code can be templated and automatically explored, and modern HLS made significant improvements in reducing those efforts, which we shall discuss more in Sections 3.1 and 3.2.

Some other drawbacks in the current HLS flow also hinder users from leveraging the benefit of FPGA acceleration. For example, many have long complained about the lengthy turnaround time from coding to hardware evaluation. Although designs can be evaluated at a high level using the HLS tools, actually running a design in a device requires executing logic synthesis, placement, routing, and bitstream generation, which can take many hours for complex designs. In contrast, the CUDA compiler can finish the compilation in seconds. As a result, an iterative refinement-based methodology can be more easily used in GPU development, and the developers could efficiently explore different design options to find the optimal one. In addition, CUDA kernels can print out debug messages to the console during the accelerator execution, whereas FPGAs require more sophisticated tools for on-board debugging.

3 RESEARCH CHALLENGES AND OPPORTUNITIES

Despite the increasing adoption of HLS and its successful applications to multiple areas for FPGA acceleration, it is not a simple task to achieve high-performance FPGA designs using HLS. In this section, we shall discuss the challenges faced by the HLS users and the opportunities for further research and development to enhance the HLS technology to overcome these challenges. We believe

[4] https://github.com/UCLA-VAST/minimap2-acceleration.

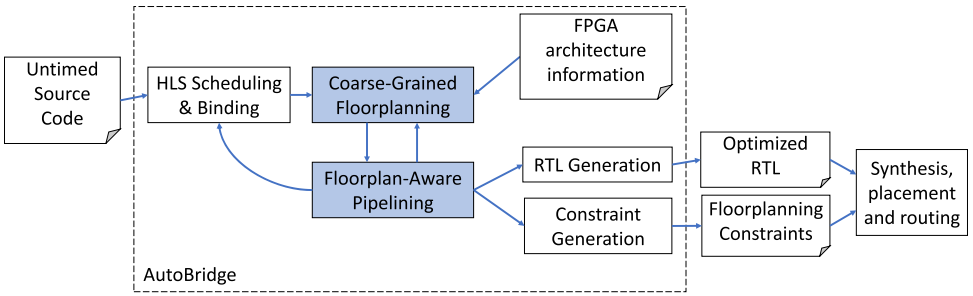


Fig. 13. The overall flow of the AutoBridge framework.

that the following five areas need more attention, including achieving high clock frequency designs using HLS (Section 3.1), coping with complex pragmas and code transformation needed to achieve high performance (Section 3.2), legacy code refactoring needed to comply with HLS standard (Section 3.3), the need for open source HLS infrastructures (Section 3.4), support DSLs (Section 3.5), and standardization (Section 3.6). The following sections go into the details of these topics.

3.1 Improving Clock Frequency

There remains a significant gap in the achievable frequency between an HLS design and a hand-crafted RTL one. Operator delays can be characterized through profiling to a reasonable accuracy; however, interconnect delays are much harder to predict and optimize for, which often become the frequency bottleneck for HLS-based designs. One common issue for estimating the interconnect delay is that the downstream FPGA implementation tool can introduce significant delay variations in placement and routing, which is difficult if not impossible to forecast during HLS. In addition, the timing issue is worsened as modern FPGAs are becoming increasingly heterogeneous and often span multiple dies [117, 118]. Such long routing paths can become the limiting factor of the maximum frequency for large designs.

To address this issue, Guo et al. [116] propose AutoBridge as an automated framework that couples a coarse-grained floorplanning step with pipelining during HLS compilation. First, the AutoBridge framework identifies the likely long wires during HLS synthesis, especially those that cross the die boundaries. Second, AutoBridge inserts pipeline registers automatically on these long wires to allow the downstream implementation tool to place such wires across die boundaries without degrading the maximum achievable frequency. Additionally, this technique often prevents the placer from aggressively packing the logic onto a single die, which in turn helps to reduce the local congestion and further improve the design frequency.

Figure 13 shows the high-level flow of AutoBridge as part of a modern HLS tool, where the two major steps in AutoBridge are highlighted. The coarse-grained floorplanning step models the FPGA device as a 2-dimensional grid and maps the dataflow functions onto slots of the grid. The optimization algorithm considers the resource usage of each function and minimizes the total cost of slot-crossing wires. This problem is solved using a top-down partitioning-based placement algorithm, where the functions are iteratively partitioned into one of the two halves of the grid until the function-to-slot mapping is fully specified. The floorplan-aware pipelining step aims to pipeline every cross-slot connection to facilitate timing closure. One challenge of arbitrarily inserting pipeline registers is that it may degrade the system throughput. This is especially true in the context of dataflow networks when the latencies on a set of reconvergent paths become imbalanced. To solve this issue, AutoBridge applies a latency balancing technique to optimally insert registers to balance the total added latency of every pair of reconvergent paths while minimizing

the area cost of the inserted registers. In addition to the communication channels between dataflow processes, AutoBridge extends to other signal types such as control signals, input/output scalars, and array arguments. This helps the timing closure for various types of HLS-based designs.

Across a set of 43 designs with different characteristics, AutoBridge improves the average frequency from 147 to 297 MHz (a 102% improvement) with no loss of throughput and a negligible change in resource utilization. Noticeably, AutoBridge was able to successfully complete 16 of the designs that were originally unroutable. Besides the significant frequency improvements from AutoBridge, it opens the door for future research and applications of physically aware HLS techniques where the impact on physical design is modeled and optimized for the early stages of HLS.

3.2 Simplifying Directives and System Integration

Although HLS has lifted the design abstraction from RTL to C/C++, in practice, extensive source code rewriting including pragma insertion is often necessary to achieve good performance. Code rewriting requires not only knowledge of hardware microarchitecture design but also familiarity with the coding style and proper use of optimization directives such as pragmas of the HLS tool. Table 2 shows a list of 20 commonly used pragmas offered by Vitis/Vivado HLS for performance optimization. For the genomics application in Section 2.4, there are 524 lines of C code for the chaining computation kernels. Among those, 216 lines are for FPGA-specific optimizations, including 48 lines of pragmas. Moreover, the programmer has to manually separate the C code to be executed on the CPU and an FPGA and manage the data transfer between them. To add those pragmas or make the code changes, the users need to know FPGA hardware and have a microarchitecture of the hardware design in mind. Unfortunately, many software developers may not have the required knowledge and background.

To reduce or eliminate the gap between the software/algorithm development and hardware acceleration, source-to-source code transformation techniques have been introduced to automate part of the pragma insertion and program transformation [41, 115, 119, 120]. A good example is the Merlin Compiler developed by Falcon Computing Solutions (acquired by Xilinx in 2020). The Merlin Compiler takes computation kernels to be accelerated in C/C++ and generates optimized kernels that can be utilized to generate high-performance FPGA accelerators. In addition, it generates a host library with proper interfaces to communicate with FPGA kernels that can be linked to users' applications. Overall, the Merlin Compiler provides an easy-to-use FPGA programming environment for software developers, and we shall discuss it in more detail in the rest of this section.

3.2.1 Programming Model. The Merlin Compiler provides an OpenMP-like high-level programming model for FPGA acceleration. Programmers only need to focus on two high-level concepts: parallelization and pipelining. Parallelization specifies spatial parallelism using multiple computation units, whereas pipelining enables temporal parallelism with overlapping computation in time.

The Merlin Compiler provides the following simple pragmas for users to designate their directives to accelerate their computation kernels:

(1) `#pragma ACCEL parallel [factor=N]`
(2) `#pragma ACCEL pipeline`
(3) `#pragma ACCEL tile [factor=N]`.

By inserting pragmas at proper locations in their program, developers convey to Merlin the intended parallelism to explore in a similar way that HPC developers use OpenMP to develop multi-core, multi-thread programs. In contrast, Vitis HLS has 20+ pragmas, many of which are

Table 2. Frequently Used Optimization Pragmas in Vitis HLS

Pragma	Description
pragma HLS dataflow	Enables task-level pipelining to overlap functions and loops with each other
pragma HLS dependence	Provides additional loop dependency information for parallelization
pragma HLS loop_flatten	Flattens a nested loop to a single loop hierarchy with improved latency
pragma HLS loop_merge	Merges consecutive loops into a single loop to reduce latency and resource usage
pragma HLS occurrence	Allows less-executed operations to be pipelined at a slower rate
pragma HLS pipeline	Enables instruction-level pipelining to allow concurrent operation execution
pragma HLS unroll	Creates multiple copies of the loop body for parallelization of the iterations
pragma HLS aggregate	Collects and groups the data fields of a struct into a single scalar
pragma HLS array_partition	Splits an array into multiple entities for parallel access
pragma HLS array_reshap	Merges multiple elements in an array into one element for parallel access
pragma HLS disaggregate	Deconstructs the data fields of a struct into multiple scalars
pragma HLS stream	Implements a variable as a FIFO and specifies the depth
pragma HLS function_instantiate	Creates task-specific modules to handle specific conditions of a function call
pragma HLS inline	Removes a function as a separate entity in the hierarchy
pragma HLS interface	Specifies the implementation protocol of a function's ports
pragma HLS allocation	Limits the resource allocation in the implemented RTL
pragma HLS bind_op	Specifies the implementation of operations in the code
pragma HLS bind_storage	Specifies the memory type of a variable in the code
pragma HLS expression_balance	Rearranges operations with associative and commutative laws to reduce latency
pragma HLS latency	Specifies the minimum and/or maximum latency for the completion of a region

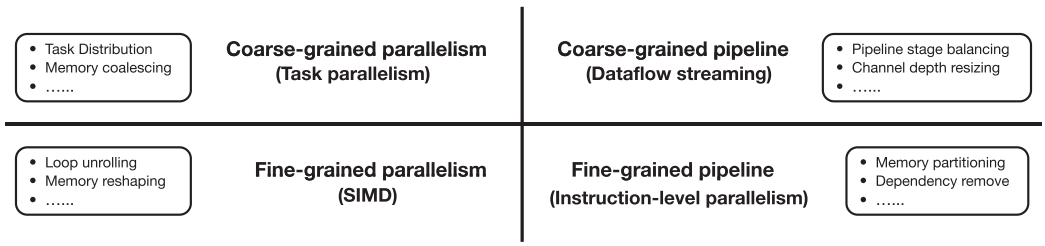


Fig. 14. Parallelism explored in Merlin and associated optimizations.

hardware/architecture-oriented. Figure 14 illustrates the four types of parallelism explored in Merlin. They can be automatically explored with the Merlin Compiler depending on whether the `parallel` and the `pipelining` pragmas are placed at the inner or outer loops of the source code.

More specifically, a user can apply the Merlin `parallel` pragma to inner or outer loops. When it is applied to an innermost loop, the user wants to explore fine-grained parallelism similar to the concept of single-instruction-multiple-data (SIMD), where multiple hardware units operate on consecutive data elements. When it is applied to an outer loop, the user wants to explore coarse-grained parallelism similar to parallel threads running on multi-core CPUs.

The Merlin `pipeline` pragma can also be applied to inner or outer loops. When the user employs it on the innermost loop, they intend to explore fine-grained pipelining similar to instruction pipelining where the execution of multiple instructions can be partially overlapped. When they use it on an outer loop, they intend to explore coarse-grained pipelining with double-buffers and/or streaming channels similar to multiple threads running in a partially overlapped fashion to execute consecutive iterations of the loop.

The Merlin `tiling` pragma can be applied to loop nests on large datasets. It is quite similar to loop tiling in CPU programming to take advantage of data locality and fast cache by bringing chunks of data to FPGA chips. The utilization of the pragma achieves data locality because data

```c
#define N 1024
#pragma ACCEL kernel
void matrix_vector_product(float a[N * N], float b[N], float c[N]) {
    #pragma ACCEL parallel factor=2
    for (int i = 0; i < N; i++) {
        c[i] = 0.;
        #pragma ACCEL parallel factor=32
        for (int j = 0; j < N; j++)
            c[i] += a[i * N + j] * b[j];
    }
}
```

(a) Input C code to Merlin

```c
#define N 1024
void matrix_vec_product(ap_uint<512> a[N*N/16], ap_uint<512> b[N/16], ap_uint<512> c[N/16]) {
    #pragma HLS interface m_axi port=a offset=slave bundle=gmem
    #pragma HLS interface m_axi port=b offset=slave bundle=gmem
    #pragma HLS interface m_axi port=c offset=slave bundle=gmem
    #pragma HLS interface s_axilite port=a,b,c,return
    float b_buf[2][32][32];
    #pragma HLS array_partition variable=b_buf complete dim=1
    #pragma HLS array_partition variable=b_buf complete dim=3
    coalesced_memcpy(b_buf, b, 0, sizeof(float) * N);
    for (int i = 0; i < N / 2; i++) {
        float a_buf[2][32][32];
        #pragma HLS array_partition variable=a_buf complete dim=1
        #pragma HLS array_partition variable=a_buf complete dim=3
        float c_buf[2];
        int size_twolines = sizeof(float) * 2 * N;
        coalesced_memcpy(a_buf, a, i * size_twolines, size_twolines);
        for (i_sub = 0; i_sub < 2; i_sub++) {
            #pragma HLS unroll
            mvm_sub(b_buf[i_sub], c_buf[i_sub], a_buf[i_sub]);
        }
        int size_tworesults = 2 * sizeof(float);
        coalesced_memcpy(c, c_buf, i * size_tworesults, size_tworesults);
    }
}

void mvm_sub(float b_buf[32][32], float &c_buf, float a_buf[32][32]) {
    #pragma HLS inline off
    c_buf = ((float )0.);
    for (int j = 0; j < 32; j++) {
        #pragma HLS pipeline
        for (int j_sub = 0; j_sub < 32; ++j_sub) {
            #pragma HLS unroll
            c_buf += a_buf[j][j_sub] * b_buf[j][j_sub];
        }
    }
}
```

(b) Output HLS C code generated by Merlin

Fig. 15. Optimizations that can be automatically done by Merlin.

accessed in a chunk are typically neighboring data in different array dimensions. Loop tiling can further enable data reuse and overlapping of computation and data transfer. The tiling size/factor is configurable to provide a tradeoff between the performance and resource utilization.

As an example, the function MV in Figure 15 computes a $(1024 \times 1024) \times 1024$ matrix-vector product. The Merlin pragmas indicate that the user desires to generate an accelerator with two dot-product engines and each dot-product engine contains 32 MAC units.

3.2.2 Automated Optimizations. To actually implement the parallelism expressed using the simple/intuitive Merlin pragmas, lots of microarchitecture details need to be filled in through code rewriting and HLS pragma insertion. For instance, in the matrix-vector product example, as shown

in Figure 15, dot-product and MAC engines need to be generated. The data in matrix a and vector b must be brought on-chip at the right time and fed in the right order to the different compute engines. On-chip buffers are introduced to store temporary data in the proper layout. Furthermore, design techniques such as memory burst and coalescing have to be used to improve the efficiency and bandwidth of off-chip/on-chip data transfers. In addition, data in the on-chip buffers need to be split into banks to avoid data access contentions. Figure 15(b) shows the final HLS code generated by Merlin from the matrix-vector product code in Figure 15(a).

The Merlin Compiler contains a set of optimizations that Merlin can apply automatically, or the users can specify manually to enable the parallelism through Merlin pragmas to arrive at optimal hardware implementations. Figure 14 lists some of the optimizations Merlin can carry out automatically to optimize the hardware microarchitecture. Next, we describe some of them.

Off-chip memory burst/coalescing. Off-chip memory access is one of the system performance bottlenecks. The Merlin Compiler automatically optimizes off-chip memory communications to match the desired parallelism expressed in pragmas or automatically detected by Merlin. Burst mode tries to bring in consecutive chunks of data to better utilize the off-chip memory bandwidth. Burst requires the burst length to be long enough. The Merlin Compiler detects possible continuous off-chip memory accesses and generates local buffers if necessary to enable burst data transfer. Coalescing is another technique for improving the utilization of the off-chip memory bandwidth. Memory coalescing combines multiple data accesses in one read or write. In the matrix-vector product example, Merlin changes ports a, b, c to 512-bit wide so the resulting accelerator can read/write 16 floating-point numbers in one off-chip memory access. It may generate on-chip buffers to enable burst/coalescing. In many cases, the Merlin Compiler employs a combination of burst and coalescing to best utilize the bandwidth to achieve maximal parallelism.

Data reuse. Data reuse is a widely used technique in FPGA hardware design. On-chip memory (or cache) is utilized to store the data temporarily to avoid repeated access to off-chip memory. For instance, in the matrix-vector product example, Merlin stores vector b on-chip using local buffers to avoid repeated off-chip access, which is much slower. In more complex cases, Merlin can detect stencil computation (e.g., [119]), then apply loop transformation to enable efficient data reuse.

Data prefetching. Data prefetching is a technique that tries to overlap the data transfer with computation in hardware design. When the computation part is running on a dataset, the communication part is fetching the next dataset. When appropriate, Merlin can insert double-buffers to enable the parallelism between computation and communication. For example, it applies automatically data prefetching in coarse-grained pipelining. In the matrix-vector product example, if the i-loop is further pipelined, Merlin will turn the buffer for a (a_buf) into a double-buffer. If Merlin can detect an in-order data access, it can generate an even more efficient dataflow streaming architecture without the need for double-buffers to achieve higher computation and communication overlapping with minimal hardware overheads.

On-chip memory reshaping/banking. On-chip memory access can greatly affect the parallelism that can be achieved. FPGA on-chip memories typically have only two ports, which can severely limit the parallelism due to memory access contentions. The Merlin Compiler can automatically change the shape of memory or break a large memory into multiple banks to match the desired parallelism and eliminate memory access contentions [5, 121]. In the matrix-vector product example, the buffer for a is reshaped to $2 \times 32 \times 32$. Furthermore, the first and third dimensions are completely partitioned so the two dot-product engines and the 32 MACs engines within each dot-product engine will have their own dedicated memory banks to avoid any access conflict.

Loop optimizations. Loops are where most of the parallelism and data locality come from. The Merlin Compiler can automatically parallelize/pipeline some loops. Depending on their trip counts,

Table 3. Experimental Results Comparing Manual HLS with Merlin

	Average Latency	Average #Pragmas
Manual HLS	7,667,994	20.94
Merlin	7,461,147	1.34
Manual/Merlin	1.03×	15.6×

the Merlin Compiler can select the right parallelism to improve the final hardware performance. It will also restructure the code to enable auto or user-specified parallelization/pipelining. Merlin also tries to eliminate data dependencies through techniques such as reduction, data reuse, and loop interchange. As shown in Figure 15(b), the Merlin-generated code for the matrix-vector product example contains two parallelized loops and one pipelined loop.

3.2.3 Results. Several experiments have employed the Merlin Compiler successfully to accelerate many real-world applications. In one experiment, the authors utilized 50+ highly optimized Vitis OpenCV HLS library functions to check Merlin's ability to reduce HLS pragmas. These functions on average contain more than 20 hardware-oriented HLS pragmas. The Merlin Compiler can eliminate almost all HLS pragmas with on average just a little more than one Merlin pragma while achieving comparable or slightly better performance as indicated in Table 3.

Such drastic reduction in HLS pragmas comes from the high-level OpenMP-like Merlin pragmas and the automated optimization capability in Merlin. As a result, one can achieve significant acceleration on an FPGA with little hardware design knowledge. Given that the Merlin Compiler can significantly reduce the number of pragmas needed for insertion, it has been used by several DSL tools, such as S2FA [122], HeteroCL [123], and PyLog [124], as part of the FPGA backend synthesis flow, or by **Design Space Exploration (DSE)** tools, such as AutoDSE [125], for fully automated HLS without any pragmas. Recently, Xilinx decided to make the Merlin Compiler open source [126] so that it can be a useful source-to-source code transformation infrastructure to facilitate the research community to develop more advanced automated code transformation and optimization capabilities on top of it. After Xilinx's acquisition of Falcon Computing in 2020, the automated code transformation and optimization techniques in Merlin are being gradually incorporated into the Vitis HLS tool from Xilinx. We expect to see a significant simplification of pragma usage in Vitis HLS in the future.

Although the Merlin Compiler can automatically perform a large portion of code reorganizations and insert appropriate pragmas related to FPGA-specific optimization, some further code changes, such as the loop exchange used in the genomic sequencing example in Section 2.4, are required for performance optimization. Such optimizations are required on any computing platform, including multi-core CPUs, many-core GPUs, and TPUs, and they are not FPGA-specific. It is beyond the current scope of the Merlin Compiler. Some research works (e.g., [127]) have used polyhedral frameworks to perform such code transformations.

3.3 Transformation of Legacy Code

As many programs were originally designed for CPUs, running them on an FPGA often involves rewriting a large portion of the code in HLS, which could take several person-weeks or even person-months. This effort adds to the cost of FPGA designing and hinders its wider application in these fields. We observe the classical 80-20 rule apply to FPGA HLS design as well, where to rewrite the code that constitutes less than 20% of the runtime, it takes 80% of the refactoring efforts of the whole application. It is imperative to eliminate or at least reduce those efforts and to enable the programmers to focus on the logic and the performance bottlenecks.

```
struct Node {
    int val;
    Node *left, *right;
};
void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node));
}
void traverse(Node *curr) {
    if (curr == NULL) return;  /* ... */
    traverse(curr->left);
    traverse(curr->right);
}

void func_dna(int *); void func_rna(int *);
typedef void genome_func(int *);
genome_func *route(int *data) {
    return (((header *)data)->type == TYPE_DNA) ? func_dna : func_rna;
}
void thread_worker(int *data) {
    genome_func *func = route(data);
    try {
        func(data);
        /* ... and calls functions for other stages */
    } catch (exception &e) {
        /* ... */
    }
}

char read_1[SIZE], read_2[SIZE], *ptr;
// stores 'A', 'C', 'G', 'T' as 0, 1, 2, 3
void choose_read() {
    ptr = (which ? read_1 : read_2);
}
```

Fig. 16. A schematic example of a legacy genomics application in C++ code.

Among the attempts to reduce the efforts, the source-to-source transformation seems to be promising because it automates the common practice, can be FPGA-vendor independent, and has the potential of further parallelizing or pipelining the output C/C++ code for performance optimizations. In this section, we discuss the common properties of legacy code that prevent direct code porting from a CPU to an FPGA. We present one of the source-to-source solutions, HeteroRefactor [120], which performs dynamic analysis of the program and automates the refactoring of legacy code. We further point out the unsolved challenges and potential future directions.

3.3.1 Challenges of Legacy Code. Given the differences between a CPU and an FPGA, HLS supports a subset of the C/C++ language. However, a program that originally targets the CPU usually involves unsupported features. In an extreme case demonstrated in Figure 16, 33 LOC could yield up to eight errors. Those errors confuse new HLS users and require experts' manual correction efforts. We list the common challenges of porting a CPU legacy program for HLS as follows.

Pointer support. There is no unified address space for on-chip memory on an FPGA. In fact, deeper and wider memories can be composed using multiple on-chip memory blocks (BRAMs or URAMs) that utilize customized logic and interconnects. As a result, it is challenging to give each memory location an identifier and to provide universal access. Ramanathan et al. [128] statically analyzed the pointer usages to reduce the requirement of global connections. However, the challenge remains when the application accesses memory dynamically, thus the support for pointers is limited in most HLS tools. As a rule of thumb, only statically analyzable pointers can be synthesized without producing errors. As a result, many commonly used pointer features, including

nested pointers (line 3), *global pointers* (line 32), and many *pointer operations*, such as comparison (line 9), require code refactoring into their equivalences to be taken by HLS tools as valid input.

Dynamic on-chip memory management. Besides the lack of unified address space and access interconnect, on-chip memory requires customizations, such as port width specifications, for efficient access. Therefore, it is challenging to dynamically reallocate the resources of one data type for another due to different access patterns. Thus, HLS tools do not support on-chip dynamic memory management (line 6). Although Giamblanco and Anderson [129] and Xue and Thomas [130] supported memory management by providing data structure templates, they require significant manual refactoring to be adapted. Liang et al. [131] proposed source-to-source transformation. However, static sizing of each data type's heap is needed.

Recursion. Similarly, recursive functions (line 10), which need dynamic storage in a stack of the execution states, are not supported by HLS tools. A manual transformation into loops is necessary. Thomas [132] provides a DSL on top of C++ to simplify this transformation.

Bitwidth. The CPU programs usually over-provision the variable bitwidth to the standard data types (line 29) supported by the processor. A direct adaption of those programs will lead to resource waste, which could cause performance degradation and higher energy consumption. Lee et al. [133] and Stephenson et al. [134] performed static analysis to automatically reduce the bitwidth.

Polymorphism. HLS does not support *virtual functions*, or *function pointers* (line 20), due to the lack of unique identifiers for the implemented function modules. In addition, because of the aforementioned memory model, it does not handle complex data type reinterpretation (line 17).

Exception handling. Similar to recursion, the lack of function execution states in a stack further causes the exception handling (line 24) challenging to implement on an FPGA. Although usually implemented as a trivial stack pop in CPU programs, exception handling on an FPGA requires the message passing to the exception handling module and the termination of all intermediate modules.

Multi-threading. Although supported in HLS to have multiple functions executed concurrently, many existing programs employ CPU thread execution models, like `pthreads`, to spawn multiple workers. Choi et al. [135] supported the `pthreads` model in HLS. Instead of putting stages into different threads for pipelining, these programs, written in this manner, often have multiple stages executed in the same thread (line 23). In HLS, pipeline stages of multiple tasks are often preferred over multiple homogenous PEs, since the stages could be optimized for a specific task, thus improving performance and reducing resource usage.

3.3.2 Automated Refactoring. To address these challenges, Lau et al. [120] proposed to automatically refactor the legacy code for HLS compatibility in their work, HeteroRefactor. They observed that the major challenge in supporting legacy code is the dynamic property of CPU programs. Traditionally, to refactor the code, one needs to understand the algorithm to determine the bounds. For example, the data width of a color pixel is 24 bits, the maximum size of a tree is the input length, and so forth. Using dynamic invariant analysis techniques, they extended the commonly used tools in the software engineering community to collect FPGA-specific properties. Using this knowledge, HeteroRefactor can transform the program to be synthesizable and selectively offload the program to the FPGA when the invariants are not violated. Figure 17 demonstrates the overall framework of HeteroRefactor. HeteroRefactor consists of two parts, *dynamic invariants collection*, which detects the program properties, and *refactoring*, which uses the properties to perform FPGA-specific transformations.

Dynamic invariants collection. Two approaches are supported in the HeteroRefactor framework, *Daikon-based* and *refactoring-based detection*, applicable to different scenarios and extensible to a

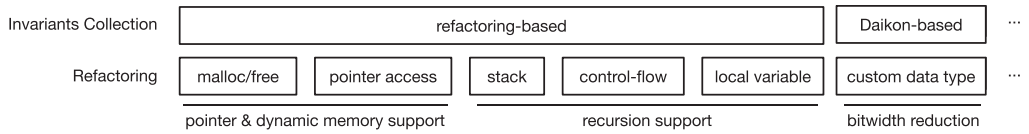


Fig. 17. The overall framework of HeteroRefactor.

broader range of invariants. By executing the program using typical input data, the likely properties could be collected and therefore an understanding of the algorithm is obtained for the refactoring.

The Daikon-based detection approach supports the dynamic detection of a wide range of invariants by utilizing the state-of-the-art detector, Daikon [136]. Although the Daikon framework already supports many properties, it requires extensions for HLS refactoring. In HeteroRefactor, the authors proposed two types of FPGA-specific invariants: (1) the required bitwidth of a variable, and (2) the number and type of elements in an array. Besides those properties, through standard specifications in Daikon, one could further extend HeteroRefactor for addressing other challenges.

Additionally, to reduce the instrumentation overhead and support non-standard invariants, Lau et al. [120] proposed a refactoring-based approach. Although powerful, the Daikon detector needs tens of minutes to instrument a complex program, and its support is limited to a given static program. We could collect some invariants with lower costs if we embed the instrumentation in the program and optimize them with the compilers. For example, HeteroRefactor modifies memory allocation/de-allocation function calls and adds tracing points at the entry/exit of recursive functions. In this way, one could obtain the size of dynamically allocated elements and the depth of the necessary function call stack during normal program execution. However, program modification, which is not supported in Daikon, is required in some of the invariant detections, such as the precision of a floating-point variable. Using the refactoring-based approach, HeteroRefactor performs differential execution of two programs: the original program and the modified low-precision program. The differences are empirically assessed to determine the minimum precision to maintain an acceptable precision loss.

Refactoring. HeteroRefactor addressed four aforementioned challenges: pointer support, dynamic memory management, recursion, and bitwidth. Using the collected invariants, Lau et al. [120] performed transformations using the ROSE compiler [137].

To support pointers and dynamic memory management, HeteroRefactor (1) rewrites `malloc`/`free` and (2) modifies the pointer accesses to array accesses. With the collected sizes of dynamic allocations, HeteroRefactor could create preallocated arrays with optimized access width per data type, and allot memory resources using a buddy memory system. It then converts the pointers into indexes in the arrays before changing the pointer operations into index arithmetic instructions. From there, it transforms the pointer access into the access of the corresponding element in the array.

To support recursion, HeteroRefactor (1) creates a stack for the local variable context, (2) rewrites the control flow, and (3) modifies the variable accesses. The invariants collected in the detection step determine the depth of the stack. HeteroRefactor converts the recursion to a `while` loop, where a function call pushes the new context into the stack and continues the loop, and a function return pops the stack and restores the execution state. Local variable accesses are redirected to the stack.

To optimize the variable bitwidth, HeteroRefactor (1) modifies the data width, (2) changes the arithmetic operators, and (3) propagates the type changes. It reduces the width of an integer using

`ap_uint<k>` or `ap_int<k>` provided by Vitis HLS, where `k` is the collected invariant. Utilizing Thomas's templatized soft floating-point type library [138], HeteroRefactor optimizes the precision of a floating-point variable. It modifies the operators of the variables accordingly to the reduced bitwidth and finally propagates the type changes to their references to keep the program valid.

HeteroRefactor further creates a guard protection system in the refactored kernel to monitor if the dynamic invariants are maintained for unforeseen data. When the dynamic invariants are violated, the program execution will fall back to the CPU to ensure correctness.

Experimental result. Lau et al. [120] employed 10 programs originally designed for a CPU to evaluate HeteroRefactor, including complex recursion programs like Aho-Corasick string matching, merge sort and Strassen's matrix multiplication, and real-world applications like face detection, 3D rendering from Rosetta benchmark suite [139], and color conversion from OpenCV. Those programs are ported to an FPGA for on-board execution using HeteroRefacor with no human intervention, effectively reducing the efforts of manual rewriting the program with 1.05× more lines of the legacy code on average. Furthermore, by optimizing the bitwidth of integers and the precision of floating numbers, BRAM usage is decreased by 41%, and DSP usage is cut by 50%, correspondingly.

3.3.3 Future Work. Although HeteroRefactor has addressed many of the challenges, some issues remain unsolved. To name a few, *global pointer* remains unsupported because of the complexity of access optimizations, and *polymorphism* support is not implemented. To the best of the authors' knowledge, no existing work has supported the C++ *exception handling* in HLS. LegUp has supported the multi-threading model [135]. However, the support is limited to `pthreads` and requires explicit data flow pipelining, which is less common in CPU programs. For most of the existing legacy code, it remains impractical to directly synthesize them into hardware in a push-button fashion.

With the increasing complexity of the applications, more available resources on FPGAs, and the trends of embedded lightweight processors like ARM processors or AI Engines [140] on the fabrics, it might be a good time to revisit the hybrid hardware-software architecture, like the one proposed in the work of Canis et al. [141] with an MIPS soft-processor compiled with the LegUp HLS. Combining the power of an FPGA's customizability and instruction processor's compatibility, we could expect future HLS to have greater programmability while producing designs with a higher performance given that the embedded processor can run with a much higher frequency.

3.4 Open Source HLS

Given the complexity of HLS, combining technology from compilation, simulation, hardware synthesis, processors, and system infrastructure to achieve high-quality results, large projects seem somewhat inevitable. Although many research projects have focused on aspects of HLS over the years [1], open source projects have had particular impacts. LegUp [141, 142] was one significant open source project implementing a complete end-to-end flow to RTL including integration with a processor subsystem. This enabled synthesizing a wide variety of input code, including multi-threaded code using the pthreads library and OpenMP APIs.[5] Bambu [143] and Dynamatic [144] are other significant open source HLS projects that are still under active development.

Recently, the presence of widely available commercial HLS tools has heavily shifted the areas of active research. Rather than focusing on low-level synthesis techniques, most current research projects instead focus on higher-level concerns, as discussed in Section 3. These research projects

[5]Although originally open source, LegUp was transitioned to closed-source source development and eventually acquired by Microsemi in 2020.

tend to generate synthesizable C code or interface with commercial tools through public interfaces. For instance, Xilinx Vitis HLS publishes a direct IR interface that can be directly used by higher-level tools [145]. Although the presence of proprietary tools has enabled many new areas of research, reliance on complex closed-source tools has made innovative research on low-level synthesis techniques somewhat less common, since it typically requires significant engineering to compare favorably to commercial tools.

One solution to this engineering complexity is to leverage best-in-class infrastructure. Although LLVM [2] has formed the basis for several HLS tools, the core CDFG representation that it supports only addresses some aspects of an end-to-end HLS flow. The MLIR [146] project offers a promising framework for next-generation HLS tools that can help to address this complexity. MLIR provides a core of basic functionality integrated with LLVM that can be easily extended to support a wide variety of abstractions through *dialects*. Several dialects included with MLIR are convenient for HLS tools, including abstractions of structured control flow and affine loops. Another key capability useful for HLS is that MLIR supports code regions with a variety of basic structures. In addition to supporting code regions describing a static single assignment (SSA)-style sequential control-flow graph of operations, MLIR code regions may also describe a concurrent graph of operations with arbitrary feedback, similar to circuit descriptions in a hardware-oriented compiler. ScaleHLS is the first attempt to implement HLS transformations using the MLIR framework [147, 148].

3.5 Domain-Specific Languages

Programming high-performance FPGA applications with traditional C++-based HLS tools demands a deep understanding of the underlying hardware details, which is quite different from traditional software programming. In addition, designers often need to insert hardware-centric pragmas to achieve the QoR goals as discussed in Section 3.2. These issues hinder the wide adoption of HLS and limit the users to a small group of designers trained with HLS-specific knowledge.

Using DSLs can simplify the work of both programmers and compilers to identify and exploit opportunities for advanced customizations 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 or re-purposed from CPUs or GPUs to FPGAs. For example, Halide-HLS [149], HeteroHalide [150], and GENESIS [151] build on Halide [152] to synthesize optimized image processing pipelines for FPGAs. Darkroom [153] and Rigel [154] capture image processing algorithms as direct acyclic graphs of basic image processing operations and generate efficient hardware accelerators for FPGAs. Heterogeneous image processing acceleration (Hipacc) [155] is another DSL producing low-level code for image processing kernels on FPGAs. TVM [156, 157] is a Halide-inspired compilation framework for deep learning compilation, which supports multiple hardware backends including FPGAs. T2S-Tensor [70] is a Halide-based DSL that allows the programmers to specify a dense tensor computation in functional notation along with decoupled spatial optimization directives. It generates high-performance systolic arrays for FPGAs and CGRAs. SuSy [71] enables the specification of systolic algorithms in the form of uniform recurrence equations and compiles them to FPGAs. OptiML [158], a Scala-embedded ML DSL leveraging the Delite compiler framework [159], is an automated design tool for implementing FPGA accelerators from high-level programs.

These languages provide a limited number of optimized functions/operators specific to "hot" or important domains and applications (image processing, ML, network packet processing, etc.). 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. DSL-based HLS could be much more productive than C-based HLS to express certain applications on FPGAs and achieve domain-specific customizations.

Table 4. Supported Compute, Quantization, and Memory Customization Primitives in HeteroCL

Primitive	Description
C.split(i, v)	Split loop i of operation C into a two-level nested loop with v as the factor of the inner loop.
C.fuse(i, j)	Fuse two sub-loops i and j of operation C in the same nest loop into one.
C.reorder(i, j)	Switch the order of sub-loops i and j of operation C in the same nest loop.
P.compute_at(C, i)	Merge loop i of the operation P to the corresponding loop level in operation C.
C.unroll(i, v)	Unroll loop i of operation C by factor v.
C.parallel(i)	Schedule loop i of operation C in parallel.
C.pipeline(i, v)	Schedule loop i of operation C in pipeline manner with a target initiation interval v.
quantize(t, d)	Quantize a list of tensors t from floating to fixed point type d.
downsize(t, d)	Downsize a list of tensors t of integers to integers d with smaller bitwidth.
C.partition(i, v)	Partition dimension i of tensor C with a factor v.
C.reshape(i, v)	Pack dimension i of tensor C into words with a factor v.
memmap(t, m)	Map a list of tensors t with mode m (vertical or horizontal) to new tensors.
P.reuse_at(C, i)	Create a reuse buffer storing tensor P, which are reused at dimension i of operation C.

One recent example is HeteroCL [123, 160], which has recently been proposed to improve the design productivity and quality of FPGA-based hardware accelerators, especially for ML applications. HeteroCL is composed of a Python-based DSL and an automated compilation flow that maps the input algorithm into special-purpose accelerators through HLS. Similar to Halide [152] and TVM [156], HeteroCL separates an algorithm specification from a temporal compute schedule such as loop reordering, tiling, unrolling, and pipelining. Unlike the previous approaches that mainly focus on CPUs/GPUs, HeteroCL further decouples the algorithm from memory architectures and data quantization schemes, which are both essential for efficient hardware customization. In addition, HeteroCL extends TVM to blend a tensor-style declarative program with imperative code, which provides high design productivity for developing ML applications and at the same time allows the framework to support a broader range of applications. To achieve high performance on FPGAs, HeteroCL can generate efficient spatial architectures such as systolic arrays and stencil-based dataflow architectures.

On the compute side, HeteroCL provides a rich set of primitives to allow the user to perform various loop transformations and parallelizing optimizations. Data type customization plays an important role in FPGA-based accelerators due to the flexibility of the underlying hardware. HeteroCL enables users to explore quantization and downsizing optimizations on a per-variable level. Thanks to the algorithm-customization decoupling, the users can conveniently explore the accuracy vs. QoR tradeoff due to data type customization without needing to rewrite the algorithm part of the design. HeteroCL also provides a set of primitives for common memory customization to explore various memory reuse and bandwidth improvements such as partitioning, reshaping, and reuse buffer construction. Table 4 summarizes the set of customization primitives currently supported in HeteroCL.

HeteroCL supports three types of backends. First, HeteroCL supports a general backend that generates HLS C/C++ or OpenCL. In addition, HeteroCL incorporates a stencil backend based on the SODA framework [119], which automatically implements the stencil patterns with optimized dataflow architecture that minimizes the on-chip reuse buffer size. Finally, HeteroCL can analyze the user-specified "systolic" macros and generate annotated HLS C++ code as an input to the AutoSA framework [41]. The AutoSA framework performs DSE and optimization and synthesizes efficient systolic array architectures on the FPGA.

At the moment, most existing FPGA-targeted DSLs [71, 149, 153, 161] including HeteroCL either leverage commercial off-the-shelf HLS tools as a backend or directly generate the RTL code. Going forward, we envision many of these languages and frameworks can benefit from integrating with an open source and reusable compiler infrastructure and/or intermediate representation

(e.g., MLIR [146], Calyx [162]) to minimize repeated efforts and maximize the number of hardware targets.

3.6 Standardization

Historically, the development of hardware description languages in the early 1980s and the adoption of the Verilog [163] and VHDL [164] standards in the early 1990s became a critical inflection point for chip designers. Although early integrated circuits were simple enough to design and layout by hand without automation, improving manufacturing technology enabled larger devices where these design techniques were not practical. The adoption of these languages for simulation and (eventually) logic synthesis allowed more complicated VLSI devices. The de facto standardization of the synthesizable subset of these languages, based on the features implemented by a wide variety of tool vendors, allowed designs to be largely portable between different tools and technology. Today, IP developers create components based on these standards, enabling a wider market reach.

Although HLS offers even more opportunities to abstract from the details of a particular implementation technology, there has currently been little progress toward standardization. Although some tools support portability between different ASIC and FPGA technologies, it is generally difficult or impossible to compile the same code with HLS tools provided directly by FPGA vendors. Significant differences in supported features, libraries, and even the core abstractions make portability difficult [165]. Although libraries can help to hide some of these differences [166], it seems unlikely that without coordinated effort from tool vendors that the situation will change anytime soon.

One recent approach to standardization is SYCL, which provides a single-source programming environment for CPUs, GPUs, and FPGA accelerators [167]. SYCL focuses on a processor plus accelerator architecture model to describe computation, abstracting the communication and synchronization between the two. SYCL also forms the basis of Intel's Data Parallel C++ (DPC++) toolchain [168]. Unfortunately, although SYCL presents a single-source approach, achieving performance still typically requires a device-specific coding style, often leveraging vendor extensions. It remains to be seen whether accelerator compilers, including HLS for FPGAs, advance to the point where most code can be ported from one device to another while still achieving performance.

4 DISCUSSIONS AND CONCLUSION

The deployment of the FPGA HLS technology has matured significantly in the past decade, and we see widespread usage of the HLS tools for FPGA designs, especially for accelerated computing, ranging from academic research projects to commercial products. We chose to highlight successes in four different selected areas, including deep learning, video transcoding, graph processing, and genome sequencing. We also identified several challenges faced by the HLS technology and the opportunities for further research and development, especially in the areas of achieving high clock frequency, coping with complex pragmas and system integration, legacy code transformation, supporting DSLs, building on open source HLS infrastructures, and standardization, which are discussed in detail in Section 3. Before we conclude, we would like to discuss the suitability of HLS and highlight several additional challenges and research opportunities.

4.1 Suitability of HLS

In this article, we focus on the HLS design methodology and tools that start with untimed behavior descriptions, mostly in C/C++ but also possibly in other DSLs as discussed in Section 3.5. There are other ways to improve the RTL-based design productivity with better abstraction using tools like BlueSpec [169] and Chisel [170]. Although effective, they still require cycle-accurate specification as input. For this reason, we do not classify them as HLS tools.

Given our definition of HLS tools with an emphasis on raising the design abstraction to untimed behavioral specifications, the current HLS tools are well suited for synthesizing various computation, communication, and data processing algorithms into accelerators or data processing engines on FPGAs. Either the throughput or the overall latency can be set up as design objectives (under the FPGA resource constraints). But these HLS tools are often not flexible enough to handle cycle-by-cycle behavior defined by a finite state machine (FSM), as in the design of some I/O interfaces and double-pumping memories, which naturally prefers a cycle-accurate description. Hence, it often requires RTL-based scaffolding or the use of commercial IP integration tools to connect HLS-generated blocks using on-chip networks or shared memory controllers.

It is possible to use the available pragmas in existing HLS tools to enforce absolute/relative timing constraints or even nearly cycle-accurate behavior, but it is not our recommended methodology. Some recent ASIC-focused projects have reported promising results in using HLS to develop more control-intensive modules such as arbitrated crossbars, caches, NoC routers, and processor cores [171, 172]. We believe that such efforts can later be leveraged in FPGA HLS design as well.

4.2 Further Research Needs

As a supplement, we would like to highlight several additional challenges and research opportunities in the following, even though they are not directly related to the core HLS technology.

Fast performance debugging. When an HLS C program fails to meet the performance requirement, an interesting and challenging question is how to add additional pragmas and/or restructure the program for better latency or throughput. CPU and GPU programmers can use established tools like VTune [173] and NSight [174] for performance tuning. These tools exploit the built-in hardware performance counters and provide line-by-line profiling results. Such capability is greatly lacking for FPGA designs. HLScope made an important first step for both static performance estimation [175] and dynamic on-chip monitoring [176]. But more tools are needed in this area.

Quick 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 issues and timing degradation during implementation. Such deficiencies are compounded by the long runtime of the physical design stages. It is vital to generate the bitstream of an accelerator within minutes, and we believe that this can be made possible with the integration of two strategies: (1) HLS designs should be floorplanned and pipelined effectively through physical layout awareness, and (2) time-consuming physical design should be accelerated through design modularity and parallel placement and routing. There are several studies [116, 177] leveraging these strategies separately. However, we envision that the key to minute-scale design closure is to integrate these two strategies into a unified flow.

Efficient DSE. FPGA programmers need to explore a wide range of hardware customization options, whereas 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, efficient DSE in HLS with multi-objective optimization techniques is highly desired. A recent bottleneck analysis based DSE is an encouraging example in this direction [125].

We are sure that this list is not exhaustive, but its entries are representative of the most urgent needs faced by the FPGA HLS community. We look forward to another exciting and productive decade for the HLS technology to further mature.

ACKNOWLEDGMENTS

The authors would like to thank the FINN team for the work in Section 2.1.1, the NGCodec team, particularly Michael Scott and Adam Malamy who provided feedback on Section 2.2, Yuwei Hu and Yichi Zhang from Cornell University for their comments on Sections 2.3 and 2.1.2, respectively, and Marci Baun for editing the article.

REFERENCES

[1] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[2] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. 75–86.

[3] 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)*. 433–438.

[4] 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)*. 211–218.

[5] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems* 16, 2 (April 2011), Article 15, 25 pages. http://dx.doi.org/10.1145/1929943.1929947

[6] BDTI. n.d. BDTI Certified Results for the AutoESL AutoPilot High-Level Synthesis Tool. Retrieved July 27, 2021 from https://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot.

[7] Xilinx. n.d. Vivado Design Suite User Guide: High-Level Synthesis UG902 (v2012.2). Retrieved July 28, 2021 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf.

[8] ChipEstimate.com. n.d. Xilinx Unveils Vivado Design Suite for the Next Decade of 'All Programmable' Devices. Retrieved July 28, 2021 from https://www.chipestimate.com/Xilinx-Unveils-Vivado-Design-Suite-for-the-Next-Decade-of-/Xilinx/Technical-Article/2012/06/12.

[9] M. Sussmann and T. Hill. 2017. Intel HLS Compiler: Fast Design, Coding, and Hardware. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01274-intel-hls-compiler-fast-design-coding-and-hardware.pdfasoftoday.

[10] 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)*. 161–170. http://dx.doi.org/10.1145/2684746.2689060

[11] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-Sun Seo, and Yu Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 16–25. http://dx.doi.org/10.1145/2847263.2847276

[12] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'17)*. 152–159.

[13] Mohammad Samragh, Mohammad Ghasemzadeh, and Farinaz Koushanfar. 2017. Customizing neural networks for efficient FPGA implementation. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'17)*. 85–92.

[14] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ deep learning accelerator on Arria 10. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. 55–64. http://dx.doi.org/10.1145/3020078.3021738

[15] 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)*. 15–24. http://dx.doi.org/10.1145/3020078.3021741

[16] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. 65–74. http://dx.doi.org/10.1145/3020078.3021744

[17] 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)*. 11–20. http://dx.doi.org/10.1145/3174243.3174253

[18] Ke Xu, Xiaoyun Wang, and Dong Wang. 2019. A scalable OpenCL-based FPGA accelerator for YOLOv2. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*. 317–317.

[19] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, et al. 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)*. 23–32. http://dx.doi.org/10.1145/3289602.3293902

[20] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. 2019. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'19)*. 73–82. http://dx.doi.org/10.1145/3289602.3293915

[21] Akshay Dua, Yixing Li, and Fengbo Ren. 2020. Systolic-CNN: An OpenCL-defined scalable run-time-flexible FPGA accelerator architecture for accelerating convolutional neural network inference in cloud/edge computing. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'20)*. 231–231.

[22] 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)*. 171–182. http://dx.doi.org/10.1145/3431920.3439296

[23] Yu-Ting Chen, Jason Cong, Jie Lei, and Peng Wei. 2015. A novel high-throughput acceleration engine for read alignment. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'15)*. 199–202.

[24] Gowthami Jayashri Manikandan, Sitao Huang, Kyle Rupnow, Wen-Mei W. Hwu, and Deming Chen. 2016. Acceleration of the Pair-HMM algorithm for DNA variant calling. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'16)*. 137–137.

[25] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. 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)*.

[26] Michael Lo, Zhenman Fang, Jie Wang, Peipei Zhou, Mau-Chung Frank Chang, and Jason Cong. 2020. Algorithm-hardware co-design for BQSR acceleration in genome analysis toolkit. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'20)*. 157–166. http://dx.doi.org/10.1109/FCCM48280.2020.00029

[27] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. 2020. Using OpenCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. 86–96. http://dx.doi.org/10.1145/3373087.3375300

[28] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. 2018. SMEM++: A pipelined and time-multiplexed SMEM seeding accelerator for DNA sequencing. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18)*.

[29] Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu, Amin Abazari, and Ryan Kastner. 2016. Resolve: Generation of high-performance sorting architectures from high-level synthesis. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 195–204. http://dx.doi.org/10.1145/2847263.2847268

[30] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-storage acceleration of database sort on SmartSSD. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. 262–272. http://dx.doi.org/10.1145/3431920.3439298

[31] Muhuan Huang, Kevin Lim, and Jason Cong. 2014. A scalable, high-performance customized priority queue. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14)*. 1–4.

[32] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-accelerated near-storage sorting. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'21)*. 106–114. http://dx.doi.org/10.1109/FCCM51124.2021.00020

[33] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. 69–80.

[34] Davide Conficconi, Eleonora D'Arnese, Emanuele Del Sozzo, Donatella Sciuto, and Marco D. Santambrogio. 2021. A framework for customizable FPGA-based image registration accelerators. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. 251–261. http://dx.doi.org/10.1145/3431920.3439291

[35] Young Kyu Choi, Jason Cong, and Di Wu. 2014. FPGA implementation of EM algorithm for 3D CT reconstruction. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'14)*. 157–160.

[36] Juan Camilo Vega, Marco Antonio Merlini, and Paul Chow. 2020. FFShark: A 100G FPGA implementation of BPF filtering for Wireshark. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'20)*. 47–55.

[37] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. 2017. Energy efficient scientific computing on FPGAs using OpenCL. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. 247–256. http://dx.doi.org/10.1145/3020078.3021730

[38] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. 153–162. http://dx.doi.org/10.1145/3174243.3174248

[39] Dajung Lee, Andrei Hagiescu, and Dan Pritsker. 2019. Large-scale and high-throughput QR decomposition on an FPGA. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*. 337–337.

[40] Yu Zou and Mingjie Lin. 2020. Massively simulating adiabatic bifurcations with FPGA to solve combinatorial optimization. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. 65–75. http://dx.doi.org/10.1145/3373087.3375298

[41] 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)*.

[42] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-Kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'22)*. 65–77. http://dx.doi.org/10.1145/3490422.3502357

[43] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Yujin Nam, Jung Hee Cheon, and Rob A. Rutenbar. 2020. Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'20)*. 56–64.

[44] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2016. High level synthesis of complex applications: An H.264 video decoder. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 224–233. http://dx.doi.org/10.1145/2847263.2847274

[45] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, et al. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604.

[46] 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.

[47] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2020. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1014–1029.

[48] 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 Transactions on Reconfigurable Technology and Systems* 14, 4 (2021), 1–39.

[49] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. 2012. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology* 22, 12 (2012), 1649–1668.

[50] Xilinx. n.d. NGCodec Hardware HEVC Encoding (UG1408). Retrieved April 29, 2022 from https://www.xilinx.com/publications/user-guide/partner/ug1408-ngcodec-hevc.pdf.

[51] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*. 1–12.

[52] 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)*. 15–24.

[53] 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)*. 23–32.

[54] Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wen-Mei Hwu, and Deming Chen. 2019. FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge. In *Proceedings of the Design Automation Conference (DAC'19)*. 1–6.

[55] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 16–25.

[56] Jialiang Zhang and Jing Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. 25–34.
[57] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL deep learning accelerator on Arria 10. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. 55–64.
[58] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems* 11, 3 (Dec. 2018), Article 16, 23 pages. http://dx.doi.org/10.1145/3242897
[59] GitHub. n.d. Brevitas: A PyTorch Library for Quantization-Aware Training. Retrieved April 29, 2022 from https://github.com/Xilinx/brevitas.
[60] GitHub. n.d. FINN Code Example. Retrieved April 29, 2022 from https://github.com/Xilinx/finn-hlslib/blob/vitis-hls/mvau.hpp#L147-L179.
[61] Fast Machine Learning Lab. n.d. hls4ml. Retrieved April 29, 2022 from https://fastmachinelearning.org/hls4ml/reference.html.
[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR'16)*.
[63] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR'18)*.
[64] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR'18)*.
[65] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. 2020. ReActNet: Towards precise binary neural network with generalized activation functions. In *Proceedings of the European Conference on Computer Vision*.
[66] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, et al. 2018. Serving DNNs in real time at datacenter scale with project brainwave. In *Proceedings of the International Symposium on Microarchitecture (MICRO'18)*.
[67] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, 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)*.
[68] 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)*. 1–6.
[69] 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)*. 1–8.
[70] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, 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)*. 181–189.
[71] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, 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)*. 1–9.
[72] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2018. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2018), 2072–2085.
[73] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'16)*. Article 12, 8 pages. http://dx.doi.org/10.1145/2966986.2967011
[74] 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)*. 1–8.
[75] 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)*. 133–139.
[76] 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)*. 129–132.

[77] Xilinx. n.d. Versal ACAP AI Engine Programming Environment UG1076 (v2021.1). Retrieved April 29, 2022 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1076-ai-engine-environment.pdf.
[78] Xilinx. 2019. Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance. Retrieved April 29, 2022 from https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf.
[79] 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)*. 105–110.
[80] Jialiang Zhang and Jing Li. 2018. Degree-aware hybrid graph traversal on FPGA-HMC platform. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. 229–238.
[81] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Optimizing memory performance for FPGA implementation of PageRank. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'15)*.
[82] Alberto Parravicini, Francesco Sgherzi, and Marco D. Santambrogio. 2021. A reduced-precision streaming SpMV architecture for personalized PageRank on FPGA. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'21)*. 378–383.
[83] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Accelerating large-scale single-source shortest path on FPGA. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshop (IPDPS'14)*. 129–136.
[84] Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for power-law graphs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'22)*.
[85] Jialiang Zhang and Jing Li. 2021. ENIAD: A reconfigurable near-data processing architecture for web-scale AI-enriched big data service. In *Proceedings of the Hot Chips Symposium*.
[86] 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)*.
[87] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A dataflow library for graph analytics acceleration. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*.
[88] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput graph processing framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264.
[89] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 17–30.
[90] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'21)*.
[91] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, et al. 2016. Mathematical foundations of the GraphBLAS. In *Proceedings of the International Conference on High Performance Extreme Computing (HPEC'16)*. IEEE, Los Alamitos, CA, 1–9.
[92] David R. Kincaid, Thomas C. Oppe, and David M. Young. 1989. *ITPACKV 2D User's Guide*. Technical Report. Center for Numerical Analysis, Texas University, Austin.
[93] GitHub. n.d. Xilinx Runtime Library (XRT). Retrieved August 20, 2021 from https://github.com/Xilinx/XRT.
[94] 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)*. 1–6. http://dx.doi.org/10.1109/DAC18072.2020.9218718.
[95] Pauline C. Ng and Ewen F. Kirkness. 2010. Whole genome sequencing. In *Genetic Variation*. Springer, 215–226.
[96] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, et al. 2012. Comparison of the two major classes of assembly algorithms: Overlap–layout–consensus and de-Bruijn-graph. *Briefings in Functional Genomics* 11, 1 (2012), 25–37.
[97] Jason R. Miller, Sergey Koren, and Granger Sutton. 2010. Assembly algorithms for next-generation sequencing data. *Genomics* 95, 6 (2010), 315–327.
[98] Srinivas Aluru and Nagakishore Jammula. 2014. A review of hardware acceleration for computational genomics. *IEEE Design & Test* 31, 1 (2014), 19–30. http://dx.doi.org/10.1109/MDAT.2013.2293757
[99] Santhi Natarajan, N. KrishnaKumar, M. Pavan, Debnath Pal, and S. K. Nandy. 2018. ReneGENE-DP: Accelerated parallel dynamic programming for genome informatics. In *Proceedings of the International Conference on Electronics, Computing, and Communication Technologies (CONECCT'18)*. IEEE, Los Alamitos, CA, 1–6.
[100] Arun Subramaniyan, Jack Wadden, Kush Goliya, Nathan Ozog, Xiao Wu, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2021. Accelerated seeding for genome sequence alignment with enumerated radix trees. In *Proceedings of the International Symposium on Computer Architecture (ISCA'21)*. 388–401.

[101] Chris Rauer, George S. Powley, Mir Ahsan, and Nicholas Finamore Jr. 2017. *Accelerating Genomics Research with OpenCL and FPGAs*. Technical Report. Intel Corporation.

[102] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H. Oh, Krste Asanovic, Jae W. Lee, and Lisa Wu Wills. 2020. Genesis: A hardware acceleration framework for genomic data analysis. In *Proceedings of the International Symposium on Computer Architecture (ISCA'2)*. 254–267. http://dx.doi.org/10.1109/ISCA45697.2020.00031

[103] Corey B. Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, and Walter L. Ruzzo. 2012. Hardware acceleration of short read mapping. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'12)*. 161–168.

[104] James Arram, Kuen Hung Tsoi, Wayne Luk, and Peiyong Jiang. 2013. Reconfigurable acceleration of short read mapping. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'13)*. 210–217.

[105] Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner. 2014. Hardware accelerated novel optical de novo assembly for large-scale genomes. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14)*. 1–8.

[106] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, et al. 2019. FPGA accelerated INDEL realignment in the cloud. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'19)*. 277–290.

[107] Jason Cong, Zhenman Fang, Muhuan Huang, Libo Wang, and Di Wu. 2017. CPU-FPGA co-optimization for big data applications: A case study of in-memory Samtool sorting. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. 291–291.

[108] Heng Li. 2018. Minimap2: Pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.

[109] Yuta Suzuki. 2019. Informatics for PacBio long reads. In *Single Molecule and Single Cell Sequencing*. Springer, 119–129.

[110] Nan Du, Jiao Chen, and Yanni Sun. 2019. Improving the sensitivity of long read overlap detection using grouped short k-mer matches. *BMC Genomics* 20, 2 (2019), 190.

[111] Nvidia. 2011. Nvidia CUDA C programming guide. *Nvidia Corporation* 120, 18 (2011), 8.

[112] Pacific Biosciences. 2014. Caenorhabditis Elegans 40x Coverage Dataset. Retrieved April 29, 2022 from http://datasets.pacb.com.s3.amazonaws.com/2014/c_elegans/list.html.

[113] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2009. Software pipelined execution of stream programs on GPUs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'09)*. 200–209. http://dx.doi.org/10.1109/CGO.2009.20

[114] 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. *arXiv preprint arXiv:1807.01340* (2018).

[115] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. 2016. *Source-to-Source Optimization for HLS*. Springer International Publishing, Cham, Switzerland, 137–163. http://dx.doi.org/10.1007/978-3-319-26408-0_8

[116] 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)*.

[117] Xilinx. 2020. Xilinx UltraScale Plus Architecture. Retrieved April 29, 2022 from https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[118] Intel. 2020. Intel Stratix 10 FPGA. Retrieved April 29, 2022 from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf.

[119] 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'18)*. 1–8.

[120] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. 2020. HeteroRefactor: Refactoring for heterogeneous computing with FPGA. In *Proceedings of the International Conference on Software Engineering (ICSE'20)*. 493–505.

[121] 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)*. 199–208. http://dx.doi.org/10.1145/2554688.2554780

[122] 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)*. 1–6.

[123] 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)*. 242–251.

[124] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen-mei Hwu. 2021. PyLog: An algorithm-centric python-based FPGA programming and synthesis flow. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'21)*. 227–228.

[125] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2020. AutoDSE: Enabling software programmers design efficient FPGA accelerators. *arXiv preprint arXiv:2009.14381* (2020).

[126] GitHub. n.d. The Merlin Compiler. Retrieved August 30, 2021 from https://github.com/Xilinx/merlin-compiler.git.

[127] 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)*. 29–38.

[128] Nadesh Ramanathan, George A. Constantinides, and John Wickerson. 2020. Precise pointer analysis in high-level synthesis. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'20)*. 220–224. http://dx.doi.org/10.1109/FPL50879.2020.00044

[129] Nicholas V. Giamblanco and Jason H. Anderson. 2019. A dynamic memory allocation library for high-level synthesis. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'19)*. 314–320. http://dx.doi.org/10.1109/FPL.2019.00057

[130] Zeping Xue and David B. Thomas. 2016. SynADT: Dynamic data structures in high level synthesis. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'16)*. 64–71. http://dx.doi.org/10.1109/FCCM.2016.26

[131] Tingyuan Liang, Jieru Zhao, Liang Feng, Sharad Sinha, and Wei Zhang. 2018. Hi-DMM: High-performance dynamic memory management in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2555–2566. http://dx.doi.org/10.1109/TCAD.2018.2857040

[132] David B. Thomas. 2016. Synthesisable recursion for C++ HLS tools. In *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'16)*. 91–98. http://dx.doi.org/10.1109/ASAP.2016.7760777

[133] Dong-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk. 2005. MiniBit: Bit-width optimization via affine arithmetic. In *Proceedings of the Design Automation Conference (DAC'05)*. 837–840. http://dx.doi.org/10.1109/DAC.2005.193931

[134] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. 2000. Bidwidth analysis with application to silicon compilation. *ACM SIGPLAN Notices* 35, 5 (May 2000), 108–120. http://dx.doi.org/10.1145/358438.349317

[135] Jongsok Choi, Stephen D. Brown, and Jason H. Anderson. 2017. From pthreads to multicore hardware systems in LegUp high-level synthesis for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2867–2880.

[136] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (2007), 35–45.

[137] Dan Quinlan, Markus Schordan, Rob Mazke, Pei-Hung Lin, Jim Leek, Justin Too, Chuhua Liao, et al. 2019. *ROSE Compiler Framework*. Technical Report. Lawrence Livermore National Lab (LLNL), Livermore, CA.

[138] David B. Thomas. 2019. Templatised soft floating-point for high-level synthesis. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'19)*. 227–235.

[139] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, et al. 2018. Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. 269–278.

[140] Sagheer Ahmad, Sridhar Subramanian, Vamsi Boppana, Shankar Lakka, Fu-Hing Ho, Tomai Knopp, Juanjo Noguera, Gaurav Singh, and Ralph Wittig. 2019. Xilinx First 7nm Device: Versal AI Core (VC1902). In *Proceedings of the Hot Chips Symposium*. 1–28.

[141] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, et al. 2013. From software to accelerators with LegUp high-level synthesis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'13)*. IEEE, Los Alamitos, CA, 1–9.

[142] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions in Embedded Computing Systems* 13, 2 (Sept. 2013), Article 24, 27 pages. http://dx.doi.org/10.1145/2514740

[143] PANDA. n.d. The PandA/Bambu Project. Retrieved July 28, 2021 from https://panda.dei.polimi.it/.

[144] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. 2021. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 97–118.

[145] GitHub. n.d. Xilinx Vitis HLS LLVM 2020.2. Retrieved August 12, 2021 from https://github.com/Xilinx/HLS.

[146] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the*

International Symposium on Code Generation and Optimization (CGO'21). 2–14. http://dx.doi.org/10.1109/CGO51591.2021.9370308

[147] Hanchen Ye, Cong Hao, Hyunmin Jeong, Jack Huang, and Deming Chen. 2021. ScaleHLS: Achieving scalable high-level synthesis through MLIR. In *Proceedings of the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'21)*.

[148] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2021. ScaleHLS: Scalable high-level synthesis through MLIR. *arXiv preprint arXiv:2107.11673* (2021).

[149] 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 Transactions on Architecture and Code Optimization* 14, 3 (2017), Article 26, 25 pages.

[150] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'20)*.

[151] Akari Ishikawa, Norishige Fukushima, Akira Maruoka, and Takuro Iizuka. 2019. Halide and GENESIS for generating domain-specific architecture of guided image filtering. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS'19)*.

[152] 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. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*.

[153] 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 Transactions on Graphics* 33, 4 (2014), Article 144, 11 pages.

[154] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics* 35, 4 (2016), Article 85, 11 pages.

[155] 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)*.

[156] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, 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)*.

[157] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: An open hardware-software stack for deep learning. *arXiv preprint arXiv:1807.04188* (2018).

[158] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the International Conference on Machine Learning (ICML'11)*.

[159] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro* 31, 5 (2011), 42–53.

[160] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. 2022. HeteroFlow: An accelerator programming model with decoupled data placement for software-defined FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'22)*.

[161] Yuze Chi, Licheng Guo, Jason Lau, Young-Kyu Choi, Jie Wang, and Jason Cong. 2021. Extending high-level synthesis for task-parallel programs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'21)*. 204–213. http://dx.doi.org/10.1109/FCCM51124.2021.00032

[162] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[163] IEEE. 1996. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Standard 1364–1995. IEEE, Los Alamitos, CA. http://dx.doi.org/10.1109/IEEESTD.1996.81542

[164] ANSI/IEEE. 1994. *IEEE Standard VHDL Language Reference Manual*. ANSI/IEEE Standard 1076-1993. IEEE, Los Alamitos, CA. http://dx.doi.org/10.1109/IEEESTD.1994.121433

[165] 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. http://dx.doi.org/10.1109/ACCESS.2020.3024098

[166] Johannes de Fine Licht and Torsten Hoefler. 2019. hlslib: Software engineering for hardware design. *CoRR* abs/1910.04436 (2019). http://arxiv.org/abs/1910.04436.

[167] Khronos. 2021. SYCL 2020 Specification Revision 3. Retrieved April 29, 2022 from https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[168] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2021. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*. Apress. http://dx.doi.org/10.1007/978-1-4842-5574-2

[169] Rishiyur Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE'04)*. IEEE, Los Alamitos, CA, 69–70.

[170] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the Design Automation Conference (DAC'12)*. 1212–1221.

[171] Brucek Khailany, Evgeni Krimer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, et al. 2018. A modular digital VLSI flow for high-productivity SoC design. In *Proceedings of the Design Automation Conference (DAC'18)*.

[172] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. 2019. What you simulate is what you synthesize: Designing a processor core from C++ specifications. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'19)*.

[173] James Reinders. 2005. *VTune Performance Analyzer Essentials*. Intel Press.

[174] Thomas Bradley. 2012. *GPU Performance Analysis and Optimisation*. NVIDIA Corporation.

[175] 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)*. 691–698. http://dx.doi.org/10.1109/ICCAD.2017.8203844

[176] 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)*. 125–128. http://dx.doi.org/10.1109/FCCM.2017.44

[177] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. 2022. RapidStream: Parallel physical implementation of FPGA HLS designs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'22)*. 1–12. http://dx.doi.org/10.1145/3490422.3502361

Received September 2021; revised January 2022; accepted April 2022