GENERAL-PURPOSE COMPUTERS ARE widely used in our modern society. There were close to 24 million software programmers worldwide as of 2019 according to Statista. However, the performance improvement of general-purpose processors has slowed down significantly due to multiple reasons. One is the end of Dennard scaling,\textsuperscript{17} which scales transistor dimensions and supply power by 30\% every generation (roughly every two years), resulting in a twofold increase in transistor density and a 30\% reduction in transistor delay (or improvement in the processor frequency).\textsuperscript{2}

Although transistor density continues to double per generation according to Moore’s Law, increases in processor frequency were slowed or almost stopped with the end of Dennard scaling in the early 2000s (due to leakage-current concerns). The industry entered the era of parallelization, with tens to thousands of computing cores integrated in a single processor and tens of thousands of computing servers connected in a warehouse-scale datacenter. However, by the end of the 2000s, such massively parallel general-purpose computing systems were again faced with serious challenges in terms of power, heat dissipation, space, and cost.\textsuperscript{14,18}

Customized computing was introduced to further advance performance. Using special-purpose accelerators, customized computing achieves much higher efficiency by enabling the processor architecture to be adapted to match the computing workload.\textsuperscript{10,14,16,34}

The best-known customized computing example is probably the Tensor Processing Unit (TPU),\textsuperscript{23} announced by Google in 2017 for accelerating machine-learning (ML) workloads. Designed in 28nm CMOS technology as an application-specific integrated circuit (ASIC), TPU demonstrated a 196x performance/watts power-efficiency advantage over the general-purpose Haswell CPU, a leading server-class CPU at the time. One significant source of energy inefficiency of a general-purpose CPU comes from its long instruction pipeline, time-multiplexed by tens or even hundreds of different types of instructions, resulting in high energy overhead (64\% for a typical superscalar, out-of-order pipeline studied in Cong et al.\textsuperscript{9}). In contrast, domain-specific accelerators (DSAs) achieve their efficiency in the following five dimensions: the use of special data types and operations, massive parallelism, customized...
memory access, amortization of the instruction/control overhead, and algorithm and architecture co-design. When these factors are combined, a DSA can offer significant (sometimes more than 100,000x) speedup over a general-purpose CPU.

Given that DSAs are domain-specific, a key question is whether a typical application developer in a particular application domain can easily implement their own DSAs. For ASIC-based DSAs, such as TPUs, there are two significant barriers. One is design cost. According to McKinsey, designing an ASIC with a leading-edge technology (7nm CMOS) is close to 300 million USD, which is prohibitively high for most companies and developers. The second barrier is turnaround time. It usually takes more than 18 months from the initial design to the first silicon and even longer to production. During this time, new computation models and algorithms may emerge, especially in some fast-moving application fields, making the initial design outdated.

Considering these concerns, we think that field-programmable gate-arrays (FPGAs) are a viable alternative for DSA implementation. Given their programmable logic and interconnects, and customizable building blocks (BRAMs and DSPs), FPGAs can be customized to implement a DSA without going through a lengthy fabrication process, and they can be reconfigured to a new design in a matter of seconds. Moreover, FPGAs have become available in the public cloud, such as Amazon AWS F1 and Nimbix. Designers can create their own DSAs on the FPGA and use it at a rate of 1-2 USD per hour to accelerate the desired applications, even if FPGAs are not available in the local computing facility. Because of their affordability and fast turnaround time, we believe FPGAs can help to democratize customized computing, allowing millions of developers to create their own DSAs on FPGAs for performance and energy-efficiency advantages. Although DSAs implemented on an FPGA are less efficient than those on an ASIC due to lower circuit density and clock frequency, they can still deliver tens or hundreds of times better efficiency compared to CPUs.

However, to achieve the true democratization of customized computing, a
convenient and efficient compilation flow must be provided to enable a typical performance-oriented software programmer to create a DSA on an FPGA, either on premise or in the cloud. Unfortunately, this has not been the case. FPGAs used to be designed with hardware description languages, such as Verilog and VHDL, known only to the circuit designers. In the past decade, FPGA vendors introduced high-level synthesis (HLS) tools to compile C/C++/OpenCL programs to FPGAs. Although these HLS tools raise the level of design abstraction, they still require a significant amount of hardware design knowledge, expressed in terms of pragmas, to define how computation is parallelized and/or pipelined, how data is buffered, how memory is partitioned, and so on. As we will show, DSA design performance can vary from being 108x slower than a CPU (without performance-optimizing pragmas) to 89x faster with proper optimization. But such architecture-specific optimization is often beyond the reach of an average software programmer.

In this article, we highlight our research on democratizing customized computing by providing highly effective compilation tools for creating customized DSAs on FPGAs. It builds on HLS technology but greatly reduces (or completely eliminates in some parts/sections) the need for pragmas for hardware-specific optimization. This is achieved by high-level, architecture-guided optimization and automated design-space exploration. The next section, “Promise of Customizable Domain-Specific Acceleration,” uses two examples to showcase the efficiency of DSAs on FPGAs over general-purpose CPUs. Then we discuss the challenges of programming an FPGA, followed by our proposed solutions using architecture-guided optimization, such as systolic array or stencil computation, automated design-space exploration for general applications, and raising the abstraction level to domain-specific languages (DSLs). The article concludes with a discussion of future research directions. This article’s focus is on creating new DSAs on FPGAs instead of programming existing DSAs, such as GPUs and TPUs, which are also highly efficient for their target workloads. Some techniques covered in this article can be extended to the latter, such as supporting systolic arrays or stencil computation on GPUs. This article is based on a keynote speech given by one of its co-authors during the 2021 International Parallel and Distributed Processing Symposium (IPDPS).8

**Figure 1. A merge tree that merges 16 input streams simultaneously and outputs four elements per cycle.**

**Figure 2. Comparing sorting performance using CPUs and FPGA accelerators.**
number of leaves and root throughput for a given sorting problem and memory configuration.

Figure 2 shows the speedup that the customized merge-tree accelerator on the AWS F1 FPGA instance achieves over a 32-thread Intel Xeon CPU implementation (the absolute performance of baseline is 0.21 Gbits/s). Part of this impressive efficiency gain arises from the common advantages of DSA:

- **Specialized data type and operations.** Hardware mergers support any key and value width up to 512 bits.

- **Massive data parallelism.** The customized merge tree can merge 64 input streams concurrently and output 32 integer elements every cycle.

- **Optimized on-chip memory.** We optimize each input buffer preparing the input stream to its corresponding tree leaf to have enough space to hide the DRAM access latency.

However, additional FPGA-related features enable us to achieve such high efficiency.

- **Design choices tailored to hardware constraints.** We can tailor the customizable merge tree-based architecture according to the given workload and operating environment. For example, since the available DRAM bandwidth on AWS F1 is 32 Gbits/s for concurrent read and write, we tune the tree-root throughput to be the same amount and select the maximum number of tree leaves that fit into the on-chip resources. This minimizes the number of passes needed for merging.

- **Flexible reconfigurability.** FPGAs have a unique feature to support hardware reconfigurability at runtime. When sorting multiple large datasets from different domains, one may pre-generate multiple merge-tree configurations, each optimized for its own use case—for example, data size, key/data width. Then we reprogram the FPGA at runtime to adapt our accelerator to changing sorting demands. Also, when sorting terabyte-scale data stored on SSDs, the merge sort needs to go through two phases: the first to merge the data up to the DRAM size, the second to finally merge the data onto the SSD. We show that designers can reconfigure the FPGA to switch between these two phases efficiently in Qiao et al.\(^\text{21}\).

**DNN accelerators.** Our second example is FPGA-based acceleration of deep neural networks (DNNs). Owing to their greatly improved accuracy and efficiency, DNNs have been widely used for many artificial intelligence (AI) applications, ranging from computer vision and speech recognition to robotics. One of the earliest and probably the most cited FPGA-based DL accelerator was published in early 2015,\(^\text{43}\) which used HLS to accelerate multilayer convolution neural networks (CNN). The whole system was implemented on a single Xilinx Virtex-7 485T FPGA chip with a DDR3 DRAM. It demonstrated close to a 5x speedup and a 25x energy reduction compared to a 16-thread CPU implementation. Using HLS, it was able to explore more than 1,000 accelerator design choices based on the roofline model and converged to a solution that is optimized both for computation and communication. Several graduate students carried out this implementation in less than six months and completed it almost two years ahead of Google’s announcement of TPU,\(^\text{21}\) which was done by a much larger design team.

Microsoft also designed an FPGA-based DNN accelerator—Brainwave Neural Processing Unit (NPU)—\(^\text{3}\)—at a much larger scale and has widely deployed Brainwave NPUs in its cloud production. Table 1 summarizes the hardware specifications and benchmark results of Brainwave NPU implemented on Intel S10 NX and NVIDIA GPUs.\(^\text{3}\) The results show that FPGAs can not only achieve an order of magnitude better performance than GPUs for low-batch inferences, but also compete in high-batch inference cases. On the other hand, although there is a performance gap between FPGA-based NPUs and hardened NPUs, the reconfigurable nature of FPGAs allows designers to quickly adapt their designs to the emerging DL algorithms. The advantages of FPGA-based DNN accelerators are listed below:

- **Customized data type.** DNNs are highly compressible in data types since using a low-precision, customized, floating-point format has negligible impact on accuracy. For example, the Brainwave NPU employs a narrow floating-point process which contains 1-bit sign, 5-bit exponent, and 2-bit mantissas. Although some customized bit-width (for example, 16-bit and 8-bit) support is added in the latest GPUs, the FPGAs demonstrate the flexibility to go down to ultra-low bit width (1-2 bits) with dynamic customization,\(^\text{44}\) as such narrow precision data types can be mapped efficiently onto LUTs on FPGAs.

- **Synthesizable parameters.** The Brainwave NPU allows several parameters to be changed, including data type, vector size, number of data lanes, and size of the matrix-vector tile engine during synthesis. As a result, designers can specialize its architecture for various DNN models without expensive hardware updates. This gives FPGAs a distinct advantage over ASICs in terms of the costs and development cycles, as today’s deep-learning models are still evolving.

- **Low system overhead latency.** Part of the FPGA fabric supports efficient packet-processing schemes, allowing data to be offloaded onto the accelerator with extremely low latency—for example, FPGAs are programmed to process network packets from remote servers with little overhead and feed the data into the Brainwave NPU on the same die at the line rate. Such low-latency data offloading ensures that us-

| Hardware | Nvidia T4 | Nvidia V100 | Intel S10 NX |
|----------|----------|-------------|--------------|
| Peak FP16 TOPS | 65 | 125 | 143 |
| Peak INT8 TOPS | 130 | 63 | 143 |
| On-Chip Mem. (Mb) | 10 | 16 | 16 |
| Process | TSMC 12nm | TSMC 12nm | Intel 14nm |
| Speedup at batch-3 | 1x | 2.4x | 22.3x |
| Speedup at batch-6 | 1x | 2.1x | 24.2x |
| Speedup at batch-32 | 1x | 2.5x | 5.0x |
| Speedup at batch-256 | 1x | 2.3x | 1.6x |
ers can access the acceleration result in real time, whether the data is from the edge or the cloud. It also allows a large DNN model to be decomposed and implemented on multiple FPGAs with very low latency.

The recent effort by Amazon on advanced query acceleration of its Redshift database\textsuperscript{a} is another good example of FPGA acceleration in datacenters. However, wider adoption of FPGA acceleration has been constrained by the difficulty of FPGA programming, which is the focus of this article.

**FPGA Programming Challenges**

So far, it has not been easy for a typical performance-oriented CPU programmer to create their own DSAs on an FPGA to achieve the performance gain demonstrated in the preceding section, despite recent progress on high-level synthesis. HLS allows a designer to start with C/C++ behavior description instead of the low-level cycle-accurate register-transfer level (RTL) description to carry out FPGA designs, which significantly shortens turnaround times and reduces the FPGA development cycle.\textsuperscript{1,2,13} As a result, most FPGA vendors have commercial HLS tools—for example, Xilinx Vitis\textsuperscript{b} and Intel FPGA SDK for OpenCL.\textsuperscript{c} However, while hardware designers increasingly use HLS, software programmers still find it challenging to use existing HLS tools. For example, Code 1 (see Figure 3) shows the C code for one layer of CNN. When synthesized with the Xilinx HLS tool, the resulting microarchitecture is, in fact, 108x slower than a single-core CPU. As explained in Cong et al.,\textsuperscript{15} this is because the derived microarchitecture’s performance is limited by the following inefficiencies:

- **Inefficient off-chip communication.** Although off-chip memory bandwidth can support fetching 512 bits of data at a time, the HLS solution uses only 32 bits of the available bus. This is because the input arguments of the function in Code 1 (lines 1-3), which create interfaces to the off-chip memory, use a 32-bit, floating-point data type.
- **No data caching.** Lines 10 and 14 of Code 1 directly access the off-chip memory. Although this type of memory access will be cached automatically on a CPU, it will not be cached on an FPGA by default. Instead, the designer must explicitly specify which data needs to be reused using the on-chip memories (BRAM, URAM, or LUT).
- **No overlap between communication and computation.** A load-compute-store pipelined architecture is necessary to achieve good computation efficiency as done in most CPUs. However, this is not created automatically by the HLS tool based on the input C code.
- **Sequential loop scheduling.** The HLS tools require the designer to use synthesis directives in the form of pragmas to specify where to apply parallelization and/or pipelining. In their absence, everything will be scheduled sequentially.
- **Limited number of ports for on-chip buffers.** The default on-chip memory (that is, BRAM) has one or two ports. Without proper array partitioning, it can greatly limit performance since it restricts the amount of parallel accesses to the on-chip buffers, such as the Out buffer in Code 1.

Fortunately, these shortcomings are not fundamental limitations. They only exist because the HLS tools are designed to generate the architecture based on specific C/C++ code patterns. As such, we can resolve all of them and get to a 9,676x speedup. To achieve this, we first saturate the off-chip memory’s bandwidth by packing 16 elements and creating 512-bit data for each of the interface arguments. We then explicitly define our caching mechanism and create load-compute-store stages to decouple the computation engine and data-transfer steps, so that the compute engine works only with on-chip memory. Finally, we exploit UNROLL and PIPELINE pragmas to define the parallelization opportunities. We also use ARRAY_PARTITION pragmas as needed, which brings the total number of pragmas to 28 and the lines of codes to 150, to enable parallel access to the on-chip memory by creating more memory banks (ports). Table 2 compares the two microarchitectures in terms of number of resources, global memory bandwidth, and performance when we map them to the Xilinx Virtex Ultrascale+ VCU1525.

---

\textsuperscript{a} See http://bit.ly/3X6gjDB.
\textsuperscript{b} See http://bit.ly/3EezBrf.
\textsuperscript{c} http://bit.ly/3AmdjTw.
DSA Design Automation beyond HLS

To overcome the FPGA programming challenges discussed in the preceding section, in this section, we highlight our programmer-friendly compilation solutions for FPGAs. Figure 4 details our overall compilation flow from C, C++, or DSLs to FPGA acceleration. Our solutions include using architecture-guided optimization, such as systolic array or sliding window-based architecture for stencil applications, automated design-space exploration, and domain-specific language.

**Architecture-guided optimization.**

One of the challenges of existing HLS tools is that many pragmas are needed to specify a complex microarchitecture, such as a systolic array, for efficient implementation. Instead, we allow the programmer to simply mark the section of code suitable for a certain microarchitecture pattern and let the tool automatically generate complex and highly optimized HLS code for the intended microarchitecture. This is called architecture-guided optimization. In this section, we showcase two examples—compilations for systolic arrays and stencil computations.

**Automated systolic-array compilation.** Systolic-array architecture consists of a grid of simple and regular processing elements (PEs) which are linked through local interconnects. With the modular design and local interconnects, we can easily scale out this architecture to deliver high performance while achieving high energy efficiency. The Google TPU is an example of this architecture. It implements systolic arrays as the major compute unit to accelerate the matrix operations in the ML applications.

On the downside, designing a high-performance systolic array can be challenging. It requires expert knowledge of both the target application and the hardware. Specifically, designers need to identify the systolic-array execution pattern from the application, transform the algorithm to describe a systolic array, write the hardware code for the target platform, and tune the design to achieve optimal performance. Each step will take significant efforts, raising the bar to reap the benefits of such an architecture. For example, a technical report from Intel mentioned that such a development process will take months.
Parallelization support. This architecture exploits a high level of data reuse from the application that balances the computation and communication, resulting in a high computation efficiency. Table 3 compares the details of the AutoSA-generated design with the HLS-optimized baseline. Moreover, two high-level pragmas in Code 1 replace 28 low-level pragmas in the manual HLS design. Such significant low-level HLS pragma reduction is also consistently observed with the tools introduced in later sections.

Automatic systolic-array synthesis has been an important research topic for decades. AutoSA represents the state-of-the-art effort along this direction. At present, AutoSA targets only FPGAs. One recent work, Gemmini,\textsuperscript{26} generates systolic arrays for DL applications on ASICs. The Gemmini framework uses a fixed hardware template for generating systolic arrays for matrix multiplication. The general mapping methodology based on the polyhedral framework in AutoSA can be applied to Gemmini to further improve the applicability of the Gemmini framework.

Automated stencil compiler. Our second example of architecture-guided optimization is for the stencil computation, which uses a sliding window over the input array(s) to produce the output array(s). Many areas, such as image processing and scientific computing, widely use such a pattern. While the sliding window pattern seems regular and simple, it is non-trivial to optimize the stencil computation kernels for performance given its low computation-to-communication ratio and complex data-dependency patterns. Even worse, a stencil computation kernel can be composed of many stages or iterations concatenated with each other, which further complicates data dependency and makes communication optimizations harder to achieve. To overcome these challenges, we developed a stencil compiler named SODA\textsuperscript{1,5} with the following customized optimization for the stencil micro-architecture:

- **Parallelization support.** The stencil computation has a large degree of inherent parallelism, including both spatial parallelism (that is, parallelism among multiple spatial elements within a stage), and temporal parallelism of effort even for industry experts.

To cope with this challenge, we propose an end-to-end compilation framework, AutoSA,\textsuperscript{4} to generate systolic arrays on FPGAs. Figure 5 depicts the compilation flow of AutoSA. AutoSA takes a C code as the input that describes the target algorithm to be mapped to the systolic arrays. This code is then lowered to the polyhedral IR. AutoSA uses the polyhedral model,\textsuperscript{40} a mathematical compilation framework for loop-nest optimization. AutoSA checks if the input program can be mapped to a systolic array (legality check). After that, it applies a sequence of hardware optimizations to construct and optimize the PEs (in the stage of computation management) and the on-chip I/O networks for transferring data between PEs and the external memory (communication management). AutoSA introduces a set of tuning knobs that can either be changed manually or set by an autotuner. The output of this compiler is a systolic array design described in the target hardware language. At present, we support four different hardware back-ends, including Xilinx HLS C, Intel OpenCL, Catapult HLS, and data-flow specification in TAPA.\textsuperscript{7}

With the help of AutoSA, we could now easily create a high-performance systolic array for CNN as mentioned previously. AutoSA requires minimal code changes to compile. Designers only need to annotate the code region to be mapped to systolic arrays with two pragmas (Lines 5 and 17 in Code 1). Figure 6 shows the architecture of the systolic array with the best performance generated by AutoSA. For CNN, AutoSA generates a 2D systolic array by mapping the output channel $o$ and the image height $h$ to the spatial dimensions of the systolic array. Input feature maps $In$ are reused in-between PEs vertically, weights $W$ are reused across PEs horizontally, and output feature maps $Out$ are computed inside each PE and drained out through the on-chip I/O network. For the same CNN example from the FPGA Programming Challenges section, the AutoSA-generated systolic array achieved a 2.3x speedup over the HLS-optimized baseline. This is accomplished by a higher resource utilization and computation efficiency. The regular architecture and local interconnects make the systolic array scalable to fully use the on-chip resource.
(that is, parallelism among multiple temporal stages.) SODA exploits both fine-grained spatial parallelism and coarse-grained temporal parallelism with perfect data reuse by instantiating multiple processing elements (PE) in each stage and concatenating multiple stages together on-chip, respectively. Figure 7 illustrates the microarchitecture overview of a SODA DSA with three fine-grained PEs per stage and two coarse-grained stages. Within each stage, multiple PEs can read necessary inputs from the reuse buffers and produce outputs in parallel, exploiting spatial parallelism. Meanwhile, Stage 1 can directly send its outputs to Stage 2 as inputs, further exploiting temporal parallelism.

- **Data reuse support.** The sliding window pattern makes it possible to reuse input data for multiple outputs and reduce memory communication. CPUs (and to a less extent GPUs) are naturally at a disadvantage for such data reuse due to their hardware-managed, application-agnostic cache systems. FPGAs and ASICs, however, can customize their data paths to reduce memory traffic and achieve optimal data reuse (in terms of the least amount of off-chip data movement and the smallest possible on-chip memory size) without sacrificing the parallelism as seen in Chi et al. Figure 7 shows the reuse buffers in a SODA DSA, which read three new inputs in parallel, keep those inputs that are still needed by the PEs in the pipeline, and discard the oldest three inputs, which are no longer required. Thus, the SODA DSA accesses both the inputs and outputs in a streamed fashion, achieving the least amount of off-chip data movement. Moreover, SODA can generate reuse buffers for any number of PEs with the smallest possible on-chip memory size, independent of the number of PEs used, ensuring high scalability.

- **Computation reuse support.** Computation operations are often redundant and can be reused for stencil kernels. As an example, for a 17 x 17 kernel used in calcium image stabilization, we can dramatically reduce the number of multiplication operations from 197 to 30 while yielding the same throughput. However, computation reuse is often under-explored, since most stencil compilers are designed for instruction-based processors where the parallelization and communication reuse have more impact on the performance, while computation reuse is often just a by-product. For DSAs, we can fully decouple the computation reuse from parallelization and data reuse via data-path customization. SODA algorithms include: optimal reuse by dynamic programming (ORDP) that can fully explore the tradeoff between computation and storage when the stencil kernel contains up to 10 inputs, and heuristic search-based reuse (HSBR) for larger stencils to find near-optimal design points within limited time and memory constraints.

We developed the SODA compiler to automatically generate a DSA to consider optimizations in these three dimensions. SODA uses a simple domain-specific language (DSL) as input. As an example, Code 2 (see Figure 8) shows a blur filter written in SODA DSL. Higher-level DSLs can generate the SODA DSL, further reducing the threshold for software programmers and domain experts to benefit from DSAs. We will discuss this later in the article. SODA can automatically explore a large solution space for a stencil DSA, including unroll factor, iterate factor, tile sizes, and computation reuse.

Experimental results show that SODA, as an automated accelerator design framework for stencil applications with scalable parallelization, full communication reuse, and effective computation reuse, achieves a 10.9x average speedup over the CPU and a 1.53x average speedup over the GPU. This is achieved by more than 200x lines of HLS code generated from SODA DSL, with 34% of the lines of code being pragmas. Such an extensive amount of code required by the domain-specific architectural optimizations was only possible with a fully automated DSA compiler framework. Compared to SODA, a very recent work named ClockWork consumes less resources by compiling the entire application into a flat, statically scheduled module but at the cost of

| Table 3. Micro-architecture comparison for manually optimized HLS design and AutoSA-generated design. |
|-----------------|-----|-----|-----------------|-----|
| Architecture    | BRAM| DSP | Used DRAM BW (bits) | Speedup |
| Manually Optimized HLS | 78.0% | 47.2% | 512 | 1.0x |
| AutoSA           | 93.3% | 90.1% | 512 | 2.3x |

**Figure 7. SODA microarchitecture overview.**

**Figure 8. Code 2: Blur filter written in SODA.**

1. kernel: blur
2. unroll factor: 16
3. iterate factor: 1
4. input float: image(3000, *) # Tile size, which decides reuse buffer size
5. local float: blur_x(0, 0) = (image(0, 0) + image(1, 0) + image(2, 0)) / 3
6. output float: blur_y(0, 0) = (blur_x(0, 0) + blur_x(0, 1) + blur_x(0, 2)) / 3
long compilation time and scalability to the whole chip. This presents an interesting tradeoff. A common limitation of the approach taken by both SODA and ClockWork is that neither can create a quickly reconfigurable overlay accelerator to accommodate different stencil kernels. Both must generate different accelerators for different stencil patterns, which may take many hours. It will be interesting to explore the possibility of coming up with a stencil-specific programmable architecture to allow runtime reconfiguration for acceleration of different stencil applications.

**Automated program transformation and pragma insertion.** For general applications that do not match predefined computation patterns (such as systolic arrays and stencil computations), we carry out an automated local program transformation and pragma insertion based on the input C/C++ code. The recently developed Merlin Compiler\(^1\) partially addresses this need by providing higher-level pragmas. The Merlin Compiler’s programming model is like that of OpenMP, which is commonly used for multi-core CPU programming. As in OpenMP, it optimizes the design by defining a small set of compiler directives in the form of pragmas. Codes 3 and 4 (see Figure 9) show the similarity of the programming structure between the Merlin Compiler and OpenMP.

Table 4 lists the Merlin pragmas for the design architecture transformations. Using these pragmas, the Merlin Compiler will apply source-to-source code transformations and generate the equivalent HLS code with proper HLS pragmas inserted. The `fg PIPELINE` refers to the case where fine-grained pipelining is applied by pipelining the loop and unrolling all its inner loops completely. In contrast, the `cg PIPELINE` pragma applies coarse-grained pipelining by automatically creating double buffers between the pipelined tasks.

By introducing a reduced set of high-level pragmas and generating the corresponding HLS code automatically, the Merlin Compiler can greatly simplify FPGA programming. For example, we can optimize the Advanced Encryption Standard (AES) kernel from the MachSuite benchmark\(^3\) by adding only two Merlin pragmas to achieve a 470x speedup compared to when the kernel (without any changes) is synthesized using the Vitis HLS tool. However, the manually optimized HLS code uses 37 pragmas and 106 more lines of code to achieve the same performance.

Although the Merlin Compiler greatly reduces the solution space when optimizing a kernel, it still requires the programmer to manually insert the pragmas at the right place with the right option, which can still be challenging. To further reduce the DSA design effort, we have developed a push-button design-space exploration (DSE), called AutoDSE,\(^3\) on top of the Merlin Compiler. AutoDSE is designed to automatically insert Merlin pragmas to optimize the design based on either performance, area, or a tradeoff of the two.

As depicted in Figure 10, AutoDSE takes the C kernel as the input and identifies the design space by analyzing the kernel abstract syntax tree (AST) to extract the required information, such as loop trip-counts and available bit-widths. It then encodes the valid space in a grid structure that marks the invalid pragma combina-

---

**Table 4. Merlin pragmas with architecture structures.**

| Keyword   | Available Options | Architecture Structure |
|-----------|-------------------|------------------------|
| PARALLEL  | factor=<int>      | CG and FG parallelism  |
| PIPELINE  | mode=cg/fg        | CG or FG pipeline      |
| TILING    | factor=<int>      | Loop Tiling            |

CG: Coarse-grained; FG: Fine-grained

---

**Figure 9. Code 3: OpenMP (left) and Code 4: Merlin Compiler (right).**

```c
1 #pragma omp parallel for num_threads(16)  
2 for (int i = 0; i < N; i++) {  
3     c[i] = a[i] * b[i];  }
```

```c
1 #pragma ACCEL parallel factor = 16  
2 for (int i = 0; i < N; i++) {  
3     c[i] = a[i] * b[i];  }
```

---

**Figure 10. The AutoDSE framework as in Sohrabizadeh et al.\(^3\)**

![AutoDSE Framework Diagram](image-url)
Given that DSAs are domain-specific, a key question is whether a typical application developer in a particular application domain can easily implement their own DSAs.

Assess each point in milliseconds instead of several minutes to even hours. However, as pointed out in Schafer and Wang, directly using the HLS tool results in a higher accuracy. AutoDSE has been shown to outperform previous state-of-the-art works. Nevertheless, relying on the HLS tool slows down the search process considerably and limits the scope of exploration. To speed up the design-space exploration process, we are developing a single graph neural network (GNN)-based model for performance estimation to act as a surrogate of the HLS tool across different applications. Initial experimental results show that a GNN-based model can estimate the performance and resource usage of each design point with high accuracy in milliseconds. We are excited at the prospect of applying ML techniques to DSA synthesis.

Further raising the level of design abstraction. While architecture-guided optimizations and automated program transformation make it much easier to achieve high-performance DSA designs from C/C++ programs, the software community has introduced various DSLs for better design productivity in certain application domains. One good example is Halide, a widely used image-processing DSL, which has the advantageous property of decoupling the algorithm specification from performance optimization (via scheduling statements). This is very useful for image-processing applications; writing image-processing algorithms while parallelizing execution and optimizing for data locality and performance is difficult and time-consuming due to the large number of processing stages and the complex data dependency. However, the plain version of Halide only supports CPUs and GPUs. There is no way to easily synthesize the vast number of Halide programs to DSAs on FPGAs. The direct and traditional way is to rewrite programs in hand-optimized RTL code or HLS C code, which is very time-consuming. Our goal is to develop an efficient Halide-based compiler to DSA implementations on FPGAs.

Our approach is to leverage the recently developed HeteroCL language as an intermediate representation (IR). As a heterogeneous programming in-
HeteroHalide provides a Python-based DSL with a clean programming abstraction that decouples algorithm specification from three important types of hardware customization in compute, data, and memory architectures. HeteroCL further captures the interdependence among these different customizations, allowing programmers to explore various performance/area/accuracy tradeoffs in a systematic and productive way. In addition, HeteroCL produces highly efficient hardware implementations for a variety of popular workloads by targeting spatial architecture templates, including systolic arrays and stencil. HeteroCL allows programmers to efficiently explore the design space in both performance and accuracy by combining different types of hardware customization and targeting spatial architectures, while keeping the algorithm code intact.

On top of HeteroCL, we developed HeteroHalide,\textsuperscript{26} an end-to-end system for compiling Halide programs to DSAs. As a superset of Halide, HeteroHalide leverages HeteroCL\textsuperscript{24} as an IR to take advantage of its vendor neutrality, great hardware customization capability, and the separation of algorithms and scheduling. The multiple heterogeneous backends (spatial architectures) supported by HeteroCL enables HeteroHalide to generate efficient hardware code according to application type. Figure 11 shows the overall workflow of HeteroHalide. HeteroHalide greatly simplifies the migration effort from plain Halide since the only requirement is moderate modification on the scheduling part not on the algorithm. HeteroHalide automatically generates HeteroCL\textsuperscript{24} code, using both algorithm and scheduling information specified in a Halide program. Code 5 (see Figure 12) demonstrates a blur filter written in HeteroHalide. By only changing the scheduling part of the Halide code, HeteroHalide can generate highly efficient FPGA accelerators that outperform 28 CPU cores by 4.15x on average on 10 real-world Halide applications, including eight applications using the stencil backend, one using the systolic-array backend, and one using the general backend.\textsuperscript{26} HeteroHalide has achieved this not only by taking advantage of HeteroCL, but also by adding hardware-specific scheduling primitives to plain Halide. For example, when we compile plain Halide, the scheduling is applied directly at the IR level using immediate transformation (Line 9). This may result in loss of information during such transformations, which could prevent lower-level compilers from applying optimizations. We create extensions on Halide schedules, allowing some schedules to be lowered with annotations using lazy transformation (Line 7). By adding this extension to Halide, HeteroHalide can generate specific scheduling primitives at the HeteroCL backend level, thus emitting more efficient accelerators, and an image-processing domain expert will be able to leverage DSAs by just changing the scheduling part of the existing Halide code. For the following sample Halide code, HeteroHalide can generate 1,455 lines of optimized HLS C code with 439 lines of pragmas, achieving 3.89x speedup over 28 CPU cores using only one memory channel of the AWS F1 FPGA.\textsuperscript{26} Using all four memory channels, HeteroHalide can outperform the Nvidia A100 GPU (that is 2.5x more expensive on AWS) by 1.1x using schedules generated by the Halide auto-scheduler.\textsuperscript{17} We expect to see more gains when sparse computation is involved.

In general, HeteroHalide demonstrated a promising flow of compiling high-level DSLS via HeteroCL to FPGAs for efficient DSA implementation. In addition, the newly emerged MLIR compilation framework\textsuperscript{25} is also promising as an alternative IR, which we plan to explore in the future. More opportunities to improve HLS are discussed in a recent invited keynote paper in Cong et al.\textsuperscript{12}

**Conclusion**

In this article, we show that with architecture-guided optimizations, automated design-space exploration for code transformation and optimization, and support of high-level DSLS, we can provide a programming environment and compilation flow that is friendly to software programmers and empowers them to create their own DSAs on FPGAs with efficiency and affordability. This is a critical step toward the democratization of customized computing.

The techniques presented in this article are not limited to existing commercially available FPGAs, which were heavily influenced by communication, signal processing, and other industrial applications that dominated the FPGA user base in the early days. To address the growing needs for computing acceleration, several features have been added, such as the dedicated floating processing units in Intel’s Arria-10

---

**Figure 11. HeteroHalide\textsuperscript{26} overall workflow.**

**Figure 12. Code 5: Blur filter written in HeteroHalide.\textsuperscript{25}**

```
1 // Algorithm, same as plain Halide
2 func blur_x(blur_x); blur_y("blur_y");
3 blur_x(x, y) = (image(x, y) + image(x+1, y) + image(x+2, y)) / 3;
4 blur_y(x, y) = (blur_x(x, y) + blur_x(x, y+1) + blur_x(x, y+2)) / 3;
5
6 if (for hardware)
7     blur_x_lazy_unroll(x, 16); // Schedule for hardware, added for HeteroHalide
8 ) else { 
9     blur_x_unroll(x, 16); // Schedule for software, same as plain Halide
10 }
```
FPGA family and the latest AI processor engines in the Xilinx Versal FPGA family. We expect this trend will continue, for example, to possibly incorporate the concept of coarse-grained reconfigurable arrays (CGRAs) to reduce the overhead of fine-grained programmability and greatly reduce the long physical synthesis time suffered by existing FPGAs. Our compilation and optimization techniques are readily extensible to such coarse-grained programmable fabrics. For example, we have an ongoing project to apply our systolic array compile to the array of AI engines of the Versal FPGA architecture and add CGRA overlays to existing FPGAs.

In their 2018 Turing Award lecture, “A Golden Age for Computer Architecture,” Hennessy and Patterson concluded that “…the next decade will see a Cambrian explosion of novel computer architectures, meaning exciting times for computer architects in academia and in industry…” We fully agree. Our research aims at broadening participation on this exciting journey, so that not only computer architects but also performance-oriented software programmers can create customized architectures and accelerators on programmable fabrics to significantly improve performance and energy efficiency. We hope this article stimulates more research in this direction.

Acknowledgments

We would like to thank Marcia Baun for editing the paper. This work is supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program co-sponsored by DARPA, the CAPA award jointly funded by NSF (CCF-1723773) and Intel (36888881), and CDSC industrial partners.

See https://cdsc.uclal/partners/.

References

1. Bai, Y., Sohrabzadeh, A., Sun, Y., and Cong, J. Improving DNN-based accelerator design automation with meta learning. In Proceedings of the 59th ACM/IEEE Design Automation Conf. (2022), 1347–1350.
2. Borkar, S. and Chien, A.A. The future of microprocessors. Communications of the ACM 54, 5 (2011), 67–77.
3. Boutros, A. et al. Beyond peak performance. Comparing the real performance of AI-optimized FPGAs and GPUs. In 2020 Intern. Conf. on Field-Programmable Technology. 10–18. https://doi.org/10.1109/FPT51103.2020.00011.
4. Chen, Z., Blair, H.T., and Cong, J. LANMC: LSTM-assisted non-ridge motion correction on FPGA for calcium image stabilization. In Proceedings of the 2018 ACM/SIGDA Intern. Symp. on Field-Programmable Gate Arrays. 104–109.
5. Chi, Y. and Cong, J. Exploiting compilation reuse for stochastic computation. In Design Automation Conference (2020), 1–6.
6. Chi, Y., Cong, J., Wei, P., and Zhou, P. SODA: Stencil with optimized dataflow architecture. In 2018 IEEE/ACM Intern. Conf. on Computer-Aided Design. 1–8.
7. Chi, Y. et al. Extending high-level synthesis for task-parallel programs. In 2001 IEEE 26th Annual Intern. Symp. on Field-Programmable Custom Computing Machines. 204–213.
8. Cong, J. From parallelization to customization—Challenges and opportunities. In 2021 IEEE Intern. Parallel and Distributed Processing Symposium. 1–2.
9. Cong, J. et al. Accelerator-rich architectures: Opportunities and challenges. In 1st ACM/EDAC/IEEE Design Automation Conference, 1–6.
10. Cong, J. et al. CHARM: A compositional heterogeneous accelerator-rich microprocessor. In Proceedings of the 2012 ACM/IEEE Design Automation Symp. on Low Power Electronics and Design. 379–384.
11. Cong, J., Huang, M., Pan, P., Wu, D., and Zhang, P. Software infrastructure for enabling FPGA-based accelerations in data centers. In Proceedings of the 2018 Intern. Symp. on Low Power Electronics and Design. 1–6.
12. Cong, J. et al. FPGA HLS today: Successes, challenges, and opportunities. ACM Transactions on Reconfigurable Technology and Systems 15, 4 (2022), 1–42.
13. Cong, J. et al. CGRA-FPGAs: From prototyping to deployment. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30, 6 (2011), 933–946.
14. Cong, J., Sarkar, V., Petman, G., and Bui, A. Customizable domain-specific computing. IEEE Design & Test of Computers 28, 2 (2011), 6–15.
15. Cong, J., Wei, P., Yu, C.H., and Zhang, P. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In 2018 55th ACM/ESDA/IEEE Design Automation Conf. 1–6.
16. Daily, W.J., Turakhia, Y., and Han, S. Domain-specific hardware accelerators. Communications of the ACM 63, 7 (2020), 48–57.
17. Dennard, R.H. Design of ion-implanted MOSFETs with very small physical dimensions. IEEE J. of Solid-State Circuits 5, 9 (1974), 256–260.
18. Esmaeilzadeh, H., et al. Dark silicon and the end of multicore scaling. In 2011 38th Annual Intern. Symp. on Computer Architecture. 385–396.
19. Fowers, J. et al. A configurable cloud-scale CNN processor for real-time AI. In 2018 ACM/IEEE 45th Annual Intern. Symp. on Computer Architecture. 1–12.
20. Genc, H. et al. Automated domain-specific accelerators. In Proceedings of the 2019 conference on Architectural support for programming languages and operating systems. (2019); arXiv preprint arXiv:1911.08925.
21. Hennessy, J. and Patterson, D. A new golden age for computer architecture. Communications of the ACM 62, 2 (2019), 48–60.
22. Huff, D., Dai, S., and Hannahan, P. Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. 2021 ACM/SIGDA Intern. Symp. on Field-Programmable Gate Arrays. 145–148.
23. Jouppi, N. et al. In-data-center performance analysis of a tensor processing unit. In Proceedings of the 44th Annual Intern. Symp. on Computer Architecture. 1–12.
24. Lai, Y-H. et al. HeteroCL: A multi-paradigm and composable, parallel and pipeline architecture. In 2018 55th ACM/ESDA/IEEE Design Automation Conf. 1–6.
25. Liu, Y. et al. HeteroQL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In Proceedings of the 2019 ACM/SIGDA Intern. Symp. on Field-Programmable Gate Arrays. 242–251.
26. Lattner, C. and Adve, S. LLVM: A compiler infrastructure for the end of Moore’s law. (2002); arXiv preprint arXiv:2002.11054.
27. Li, J., Chi, Y., and Cong, J. HeteroHalide: From image processing to efficient FPGA acceleration. In Proceedings of the 2020 ACM/SIGDA Intern. Symp. on Field-Programmable Gate Arrays. 51–57.
28. Li, T-M. et al. Differentially scalable programming for image processing on CGRAs. Transactions on Graphics 37, 4 (2018), 1–13.
29. Liu, L. et al. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. ACM Computing Surveys 52, 6 (2019), 1–39.
30. Liu, S. et al. Over-Gen: Improving FPGA usability through domain-specific overlay generation. 2022 55th ACM/IEEE Design Automation Conf. 852–857.
31. McInnes, S. Semiconductor design and manufacturing: Achieving leading-edge capabilities. (August 2020); https://doi.org/10.1109/TSM.2020.3011333.
32. Qiao, W. et al. FANS: FPGA-accelerated near-storage sorting. In 29th IEEE Annual Intern. Symp. on Field-Programmable Custom Computing Machines (May 2021), 108–114; http://bit.ly/3IVBD.
33. Ragan-Kelley, J. et al. Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Transactions on Graphics. 31, 4 (July 2012), 1–12.
34. Reagen, B., Adolfo, R., Shao, Y-S., Wei, G-Y., and Brooks, D. MACHsuite: Benchmarks for accelerator design and customized architectures. In 2014 IEEE Intern. Symp. on Workload Characterization, 110–119.
35. Reagen, B., Shao, Y-S., Wei, G-Y., and Brooks, D. Quantifying acceleration: Power-performance trade-offs of application kernels in hardware. Intern. Symp. on Low Power Electronics and Design (2013), 39–40.
36. Rong, H. Programmatic control of a compiler for generating high-performance spatial hardware. (2017); arXiv preprint arXiv:1711.07606.
37. Samardzic, N. et al. Bonsai: High-performance adaptive merge tree sorting. In Proceedings of the ACM/IEEE 47th Annual Intern. Symp. on Computer Architecture (May 2020); https://doi.org/10.1145/3318508.3392670.
38. Schafer, B.C. and Wang, Z. High-level synthesis design space exploration: Past, present, and future. IEEE Transactions on Computation and deep learning of Integrated Circuits and Systems 39, 10 (2020), 2628–2638; https://doi.org/10.1109/TCSVT.2019.2943570.
39. Sohrabzadeh, A., Bai, J., and Cong, J. Automated accelerator optimization aided by graph neural networks. In Proceedings of the 59th ACM/IEEE Design Automation Conference (2022).
40. Sohrabzadeh, A., Yu, C.H., Gao, M., and Cong, J. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. ACM Transactions on Design Automation of Electronic Systems 27, 4 (2022), 1–27.
41. Verdoolaege, S. ESL: An integer set library for the polyhedral model. In Proceedings of the ACM/ESDA/IEEE Design Automation Conference (2019); https://doi.org/10.1145/3295503.3396210.

Yuze Chi is a Ph.D. candidate at University of California Los Angeles, Los Angeles, CA, USA.

Weikang Qiao is a Ph.D. candidate at University of California Los Angeles, Los Angeles, CA, USA.

Alefreh Sohrabzadeh is a Ph.D. candidate at University of California Los Angeles, Los Angeles, CA, USA.

Jie Wang is a Ph.D. candidate at University of California Los Angeles, Los Angeles, CA, USA.

Jason Cong is the Volgenau Chair for Engineering Excellence at University of California Los Angeles, Los Angeles, CA, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.