Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators

QIAOYI LIU*, JEFF SETTER*, DILLON HUFF, MAXWELL STRANGE, KATHLEEN FENG, MARK HOROWITZ, PRIYANKA RAINA, and FREDRIK KJOLSTAD,
Stanford University, USA

Image processing and machine learning applications benefit tremendously from hardware acceleration. Existing compilers target either FPGAs, which sacrifice power and performance for programmability, or ASICs, which become obsolete as applications change. Programmable domain-specific accelerators, such as coarse-grained reconfigurable arrays (CGRAs), have emerged as a promising middle-ground, but they have traditionally been difficult compiler targets since they use a different memory abstraction. In contrast to CPUs and GPUs, the memory hierarchies of domain-specific accelerators use *push memories*: memories that send input data streams to computation kernels or to higher or lower levels in the memory hierarchy and store the resulting output data streams. To address the compilation challenge caused by push memories, we propose that the representation of these memories in the compiler be altered to directly represent them by combining storage with address generation and control logic in a single structure—a unified buffer.

The unified buffer abstraction enables the compiler to separate generic push memory optimizations from the mapping to specific memory implementations in the backend. This separation allows our compiler to map high-level Halide applications to different CGRA memory designs, including some with a ready-valid interface. The separation also opens the opportunity for optimizing push memory elements on reconfigurable arrays. Our optimized memory implementation, the Physical Unified Buffer, uses a wide-fetch, single-port SRAM macro with built-in address generation logic to implement a buffer with two read and two write ports. It is 18% smaller and consumes 31% less energy than a physical buffer implementation using a dual-port memory that only supports two ports.

Finally, our system evaluation shows that enabling a compiler to support CGRAs leads to performance and energy benefits. Over a wide range of image processing and machine learning applications, our CGRA achieves 4.7× better runtime and 3.5× better energy-efficiency compared to an FPGA.

CCS Concepts: • **Computer systems organization → Reconfigurable computing;** • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Hardware accelerators, memory abstraction, polyhedral analysis, machine learning

*Q. Liu and J. Setter contributed equally to the article.

This work was supported in part by DARPA's DSSoC grant FA8650-18-2-7861 and the Stanford AHA Center.

Authors' address: Q. Liu, J. Setter, D. Huff, M. Strange, K. Feng, M. Horowitz, P. Raina, and F. Kjolstad, Gates Computer Science, 353 Serra Mall, Stanford, CA 94305; emails: {joeyliu, setter}@stanford.edu, dillonhuff@gmail.com, {mstrange, kzf}@stanford.edu, horowitz@ee.stanford.edu, {praina, kjolstad}@stanford.edu.

1544-3566/2023/03-ART26
https://doi.org/10.1145/3572908

ACM Reference format:
Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2023. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. *ACM Trans. Arch. Code Optim.* 20, 2, Article 26 (March 2023), 26 pages.
https://doi.org/10.1145/3572908

1 INTRODUCTION

Hardware accelerators have emerged as the method to implement complex algorithms on energy-constrained devices, as exemplified by the explosion of image processing and machine learning accelerators [5, 12, 19, 39, 42]. Two common hardware targets are FPGAs [7, 23] and ASICs [28]. However, FPGAs have low efficiency from being too fine-grained, and single-application ASICs cannot adapt to quickly evolving applications. Programmable domain-specific accelerators like those shown in Table 1 are promising, but have historically been challenging compiler targets.

A key compiler challenge is that efficient domain-specific accelerators use a different memory abstraction than CPUs and GPUs. General-purpose hardware architectures, like CPUs, issue load and store instructions to their memory system to *pull* in the data needed for computation, as is shown in Figure 1(a). They implicitly orchestrate data movement [36], their instructions normally contain the global address of the data, and they rely on hardware-managed caches to interpret the address and wait for requested data to be fetched from any of their memory hierarchy levels. However, by leveraging domain knowledge of the application, accelerators can perfectly prefetch the data by using memory units with software-managed memory controllers that are decoupled from the compute units. Accelerators can thus overlap the memory loads with computation and reduce the hardware overhead introduced by cache-like pull memory. We refer to such memory units as *push memories*, since they push data to the computational units instead of waiting for a CPU to issue a request for data.

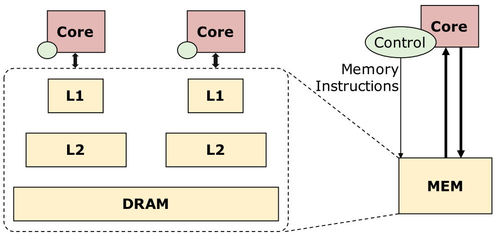
Unlike general-purpose hardware that has a centralized memory system, the memory systems on programmable push-memory accelerators are distributed. For instance, Figure 1(b) demonstrates the memory hierarchy of a **coarse-grained reconfigurable array (CGRA)** with a multi-banked global buffer (L2) and on-chip memory tiles (L1). To fully utilize the resources, push-memory accelerators often have parallel computation units fed by a number of discrete memories or a computation pipeline with memories and compute units interleaved with each other. This yields a unique programming challenge: The compilation target is not just a single piece of code but a set of programs (or configuration bit-streams) running on every memory's controller (green ovals in Figure 1(b)) that manage the data movement. Not only do the programs contain information on which data should be accessed (the addresses), but they must also align the timing of read and write events inside the buffers to synchronize the flow of data from buffers, through processing elements for computation, to another buffer.

Since push memories control both temporary storage and the flow of data, they account for a large fraction of the chip area and power in push-memory accelerators, as shown in Table 1. Unfortunately, unlike caches used by CPUs, programmable push-memory accelerators do not have a widely adopted hardware implementation for their memory systems. To minimize area and energy, these accelerators typically use their own unique custom implementation of push memory hardware optimized for specific applications or classes of applications. Thus, push memories require the compiler to optimize for a different memory abstraction for every application.

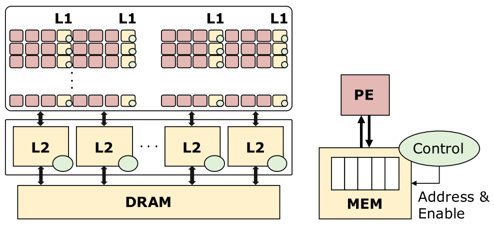
We address these challenges by creating a new push memory abstraction that we call a *unified buffer*, so named because it generalizes push memories for different application domains (such as image processing and machine learning) and different reconfigurable targets (such as our custom

Table 1. The Push Memories in Many Programmable Accelerators Account for a Large Percentage of Chip Area and Power

Domain	Accelerator	Area	Power
Multiple	Plasticine CGRA [38]	30%	Not specified
DNN	TPU [19]	37%	Not specified
DNN	Eyeriss [5]	67%	36–44%
DNN	Simba PEs [42]	41%	56%
Sparse DNN	EIE [12]	93%	59%



(a) Pull memory.



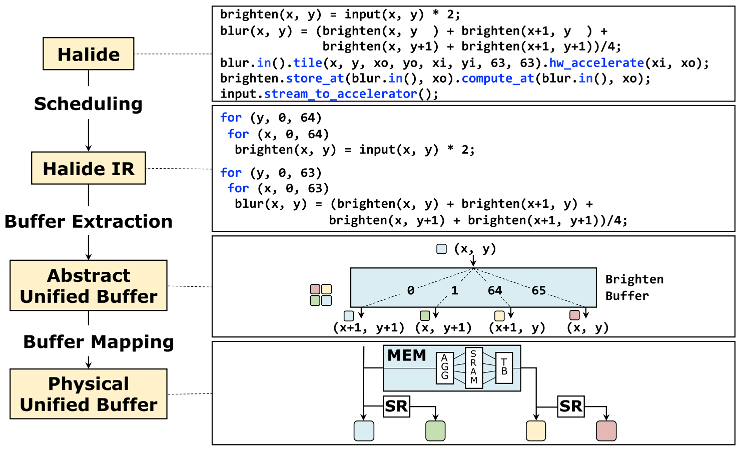
(b) Push memory.

Fig. 1. A centralized pull memory hierarchy in a CPU versus distributed push memory and control in a CGRA.

CGRA shown in Figure 9, ready-valid CGRAs, and FPGAs). The unified buffer abstraction allows us to compile a program to a single well-defined **intermediate representation (IR)**, perform application-specific optimization at that level, and then map to different hardware targets. This abstraction is described in more detail in Section 2. It also facilitates efficient push memory implementations that use custom circuits to create address generation and control logic that can be more sophisticated in how they use the SRAM. An example of an optimized implementation, our **Physical Unified Buffer (PUB)**, is described in Section 3. Taken together, the abstraction lets the compiler and hardware generator create more optimized solutions.

Using our unified buffer abstraction, we have developed a compiler backend for a subset of Halide [40] that supports tensor computations, stencils with affine indexing, and lookup tables. The compiler backend is described in Section 4 and targets programmable push memories. The compiler design is based on a simple observation: successful compilers refine and lower a program from a high-level description to a low-level description. This observation has a profound implication for compiling to programmable accelerators: If our target hardware contains high-level, optimized push memory primitives, then every stage in the compiler that deals with memories must also represent them at this level or higher. In particular, we propose that the representation of push memories in the compiler must combine storage, address generation, and control logic in a single structure—the unified buffer. Unified buffers serve as the interface inside the compiler between the application and the architecture. They define both the IR used by the compiler during push memory mapping and the logical behavior that the hardware architects must implement. By leveraging this IR, we create a robust compiler system that supports CGRAs as well as other hardware targets.

Figure 2 shows the compiler pipeline that takes a Halide program and transforms it into a composition of physical buffers. The example program brightens and blurs an image by multiplying each pixel by 2 and averaging 2×2 boxes. There are three main steps in the compiler: scheduling, buffer extraction, and buffer mapping. Section 4.1 describes the scheduling step that lowers Halide



```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.in().tile(x, y, xo, yo, xi, yi, 63, 63).hw_accelerate(xi, xo);
brighten.store_at(blur.in(), xo).compute_at(blur.in(), xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;
for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```

Fig. 2. The three compiler steps for a brighten-then-blur example. Scheduling generates tiled loops, from which buffer extraction emits the brighten unified buffer. This buffer is then mapped to shift registers (SR) and our optimized memory tile (MEM) with aggregator (AGG) and transpose buffer (TB).

programs to low-level Halide IR, following user-defined schedules that determine loop tiling, loop ordering, and where intermediate values are stored [40]. We reinterpret Halide scheduling commands to optimize for a CGRA with programmable push memories. Section 4.2 describes how the buffer extraction step extracts unified buffers from Halide IR. This step uses polyhedral techniques to determine the necessary ports, to summarize the statement instances that use each port and the values they write to or read from the port, and to calculate the cycle times when the instances use each port. Furthermore, we can apply optimization on this compiler abstraction automatically. Finally, the buffer mapping step takes as input an abstract unified buffer specification and derives a correct configuration for the hardware target. We describe mapping to our PUBs and other hardware targets in Section 4.3.

Taken together, we describe the compilation from an application specification to a configuration of custom hardware. Our contributions are as follows:

- a compiler abstraction of push memories, called a unified buffer, that represents data storage, address generation, and control in the same structure;
- a hardware memory primitive, called the PUB, that implements an efficient version of unified buffers for CGRA accelerators;
- a compiler that combines polyhedral analysis with vectorization to map unified buffers into configurations of physical buffers;
- an evaluation of compiling image processing and machine learning applications to a CGRA using our PUB.

2 THE UNIFIED BUFFER ABSTRACTION

Unified buffers separate the part of the compiler that analyzes programs to determine and optimize data movement from the part that implements the data movement by configuring physical memories. Therefore, they have two objectives:

(1) provide a precise description of the requirements of each push memory at its interface and
(2) maximize opportunities for independent optimization on each side of the interface.

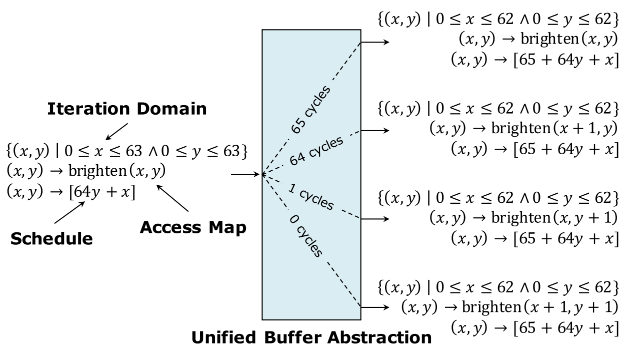

Iteration Domain
{(x, y) | 0 ≤ x ≤ 63 ∧ 0 ≤ y ≤ 63}
(x, y) → brighten(x, y)
(x, y) → [64y + x]
Schedule
Access Map
65 cycles
64 cycles
1 cycles
0 cycles
Unified Buffer Abstraction
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x, y)
(x, y) → [65 + 64y + x]
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x + 1, y)
(x, y) → [65 + 64y + x]
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x, y + 1)
(x, y) → [65 + 64y + x]
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x + 1, y + 1)
(x, y) → [65 + 64y + x]

Fig. 3. The unified buffer specifies the data movement between the brighten and blur kernels in Figure 2. Each port is defined by a polyhedral iteration domain and an access map that describe the data written to and read from the buffer. The schedule describes the cycle at which those values arrive at the port.

The first objective preserves the functionality of the application, while the second ensures its efficient implementation. Push memories are fundamentally defined by their data streams, so we need a compact representation of these streams. For this representation we use the polyhedral model [3], which provides a compact way to represent schedules and memory access patterns as integer sets and relations. Figure 3 shows the unified buffer that is generated to communicate between the brighten and blur stages of the example in Figure 2. This buffer accepts one pixel per cycle from the brighten compute kernel and delivers a 2×2 window of pixels per cycle (after a startup delay) to the blur kernel. To accommodate the required bandwidth, this unified buffer has five ports: one input port and four output ports. The data stream through each port is specified with three pieces of information:

- *The iteration domain* of the operations (statement instances) that use the port. The domain is defined by the bounds of loops in the loop nest.
- *The access map* of the operations, that maps each iteration domain point to a value read or written on the port.
- *The schedule* of the operations in the iteration domain. This schedule specifies the number of unstalled cycles between reset and each operation.

The iteration domain integer set and the access map and schedule relations are implemented using the polyhedral analysis tool ISL [47]. For the input port in our example, the iteration domain is the integer set

$$\{(x, y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq 63\}.$$

Since the brighten operation, which is the only user of the input port, is surrounded by a two-dimensional loop, the iteration domain has two index variables: an outermost index variable y and an innermost index variable x.

The unified buffer does not merely specify what operations use a port. To synthesize address generation code and optimize memory sizes, it must also specify what memory locations are accessed by those operations. To specify the memory locations, each port has an access map. For example, the second output port of the brighten buffer has the access map $(x, y) \rightarrow \text{brighten}(x + 1, y)$, which means the accessed value is to the right of each point in the operation's iteration space. The other output ports have different maps, thus collectively fetching the 2×2 stencil required by the blur kernel.

The polyhedral schedules used by the unified buffer map loop nests to cycle times in a hardware design. This contrasts with conventional polyhedral schedules, such as those produced by

Feautrier's algorithm [11] or PLUTO [3], that map elements of the iteration domain to multidimensional timestamps. While these algorithms essentially map loop nests to loop nests, our schedules map loop nests to the number of unstalled cycles after reset when each operation begins. To accommodate pipelined hardware designs, our schedules map several operations to the same timestamps.

The schedule is used to calculate when reads and writes occur. In our example, the schedule for the input port is the integer function $(x, y) \rightarrow 64y + x$. It specifies that the first write to the brighten buffer input port, at coordinate $(0, 0)$, happens $64 * 0 + 0 = 0$ cycles after execution begins and that the second brighten operation, at coordinate $(1, 0)$, happens after $64 * 0 + 1 = 1$ cycle. Furthermore, the output ports emit their first value after $65 + 64 * 0 + 0 = 65$ cycles, which is the time the buffer must delay the first value to generate the correctly aligned output. The internal distances refer to the number of cycles from when a value arrives at an input port to when it leaves an output port.

The schedules count unstalled cycles instead of clock cycles, to accommodate variable-latency operations. Since our applications are statically analyzable, our schedules guarantee that data dependencies are not violated, assuming the input data are valid and the output data can be stored. However, all hardware accelerators have to deal with variable-latency operations like main memory accesses. We accommodate variable latency by counting how many cycles the buffer has not been stalled by a dynamic operation. A stall occurs when any buffer input is invalid during a write (e.g., a DRAM read is late) or when any buffer cannot be read since the destination is not ready. During a stall, the cycle count is not incremented for all the unified buffers in the application, keeping the data waves aligned. Once the stall condition resolves, all the cycle counters resume.

In our target CGRA, the interface between the accelerator and the host memory system uses latency-insensitive channels, while the memory inside the CGRA uses (gated) cycle counters. Our Halide program tiles the inputs to create execution blocks that our compiler statically schedules. All statements within the tiled execution block are assigned a timestamp consistent with the global cycle accurate schedule. This context information ensures the scheduler adds enough buffering to allow internal compute kernel nodes to read the data from multiple predecessors simultaneously even if they are produced at different times. More details about this scheduler are given in Section 4.2. Between the tiled execution, we use double-buffered ready-valid channels. Thus, we only need to stall if the next tile has not been prefetched from DRAM into the accelerator, or the previous tile output has not been stored in DRAM before the current tile stops execution. For a CGRA implementation using ready-valid channels with buffet [36] style memory blocks that contain dependency checking capabilities, our compiler outputs address patterns and drops the cycle accurate information in its schedule. It thus lets the hardware handle execution timing and potential port conflicts.

The unified buffer interface describes the observed behavior of the memory at its interfaces, in terms of the operations in the original program. The unified buffer does not specify the internal implementation of its behavior and can be used to map to different hardware backends. Only externally visible scheduling and binding decisions are expressed. Crucially, unified buffers omit the physical capacity of the memory and the physical mapping of data to locations in memory. Thus, it is a precise specification in terms of familiar data structures for a compiler—the sets and relations of the polyhedral model—and leaves the architects considerable room to optimize the design. Next, we describe how we implemented this interface to design a high-performance, programmable push memory for image processing and machine learning applications.

3 PHYSICAL UNIFIED BUFFER

By creating a clear interface between the compiler and the underlying hardware, the unified buffer abstraction gives the hardware architect the freedom to explore a design space of *physical buffers* that implement this interface to find one that is both area and energy efficient. To explore the

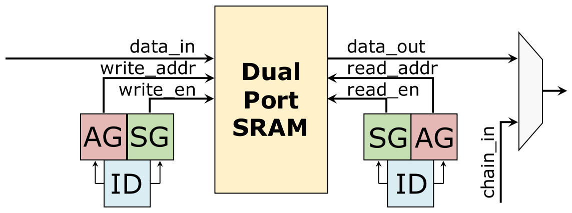


Fig. 4. A common physical buffer implementation with a dual-port SRAM. Two IterationDomain (ID) modules each drive an AddressGenerator (AG) and a ScheduleGenerator (SG) to orchestrate writes to and reads from the memory. The output has a multiplexer for memory chaining.

hardware design space, we have created a flexible physical buffer hardware generator. We look at a few hardware implementations of unified buffers, each with increasing efficiency. We call our final optimized version the PUB.

3.1 Dual-Port SRAM

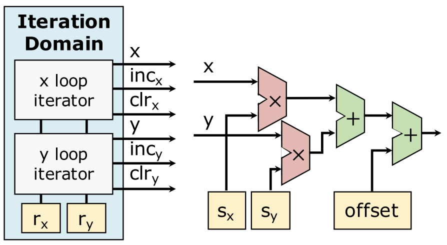
The simplest hardware implementation of a unified buffer wraps a dual-port SRAM with logic that computes the addresses and sequences of read/write enables for the iteration domain at each port (Figure 4). Since all implementations have a finite size, the design also contains logic for chaining multiple physical buffers into a larger buffer.

To implement a naïve physical buffer, we place three modules at the input and output ports of the memory. These modules are **IterationDomain (ID)**, **AddressGenerator (AG)**, and **ScheduleGenerator (SG)**. They provide implementations of the corresponding components on the ports of the unified buffer abstraction. The IterationDomain module implements counters corresponding to `for` loops, while the AddressGenerator and ScheduleGenerator modules implement mapping logic from an IterationDomain module to an address and a read/write enable for the associated memory port.

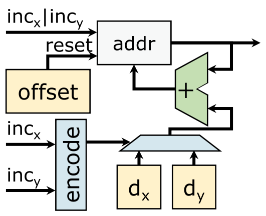
The AG and SG modules can be described as affine functions of the iteration domain, which can naïvely be implemented with hardware multipliers, as shown in Figure 5(a). We can replace the multipliers with adders by realizing that the incoming x and y values come from counters, so each multiplier output can be generated by a simple recurrence relation: $\text{out}(i+1) = \text{out}(i) + d$, where delta d is the amount the multiplier output increases with each update. Furthermore, we can reduce the required hardware to a single adder by realizing that at any update, only one loop variable is incrementing (and many may be reset). This means we can precompute how much the affine function should change when each loop increments, and we can express the affine function $\text{A}(x, y)$ as a state transition from iteration i to iteration $i+1$. This turns into the recurrence relation $\text{A}(x, y)_{i+1} = \text{A}(x, y)_i + (\texttt{inc}_y ? \, \text{d}_y : \texttt{inc}_x ? \, \text{d}_x : 0)$, where $\text{A}(x, y)_0 = \text{offset}$, $\texttt{inc}_y$, $\texttt{inc}_x$ are Booleans that indicate whether to increment and d_x, d_y are increment deltas. With this transformation, the whole function can be implemented with one adder as shown in Figure 5(b). Figure 5(c) shows an example of the relation between the strides, ranges, and deltas for a simple downsample-by-2 traversal of an 8×8 image. Since we only need the delta for one loop variable at a time, we only require a single adder and a register along with a multiplexer to increment the running address by the delta of the outermost loop variable that is incremented.

3.2 Wide-Fetch, Single-Port SRAM

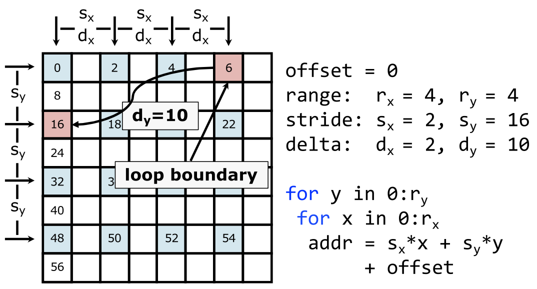
While dual-port SRAMs are often used for an FPGA or an ASIC, they are not the most efficient push memories for two reasons: First, dual-port SRAMs can be more than two times larger than their single-port counterparts for the same storage capacity while consuming 40% more energy



(a) Basic affine function implementation using multipliers.



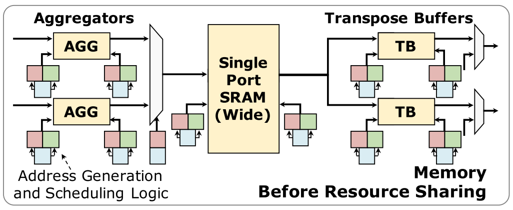
(b) Affine function as a recurrence relation.



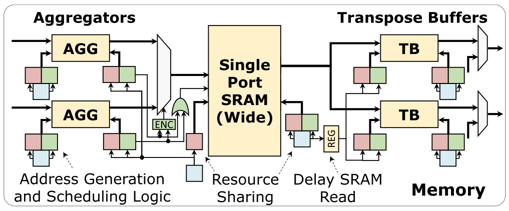
```
offset = 0
range:  r_x = 4, r_y = 4
stride: s_x = 2, s_y = 16
delta:  d_x = 2, d_y = 10

for y in 0:r_y
 for x in 0:r_x
  addr = s_x*x + s_y*y
       + offset
```

(c) An example of a downsample-by-2 over an 8×8 image.

Fig. 5. Area optimizations in the affine function hardware for address and schedule generation with a two-dimensional iteration domain. (a) An implementation that uses the value of the counters in the iteration domain. (b) An implementation that embeds the address delta between loop levels. (c) An example that shows the relationship between the strides and deltas.



(a) Before resource sharing.



(b) After resource sharing.

Fig. 6. Diagram of a PUB with a wide-fetch single-port SRAM, AGG, and TB. Sets of ID/AG/SG controllers control the input and output of each sub-component.

per access [33]. Second, energy per accessed byte is often lower if more data are fetched from an SRAM on each cycle [46]. Thus, in custom-designed memories, wide-fetch single-port memories are typically used to emulate multiple ports to improve energy per access.

To emulate simultaneous reads and writes with a single-port SRAM, we create a physical buffer that consists of three buffers, as shown in Figure 6. One of the buffers is a large SRAM, while two small buffers are placed on either side of the SRAM. The smaller buffers are implemented using registers/register files, and contain 8 to 16 words (two to four fetch blocks) when a four-word fetch SRAM is used. The small buffer between the input port and the SRAM (**aggregator (AGG)**) serves as a serial-to-parallel converter and the buffer between the SRAM and the output port (**transpose buffer (TB)**) serves as a transpose buffer for small blocks (4 × 4 for our design) and a parallel-to-serial converter. To maximize the utility of this buffer, we gave it two input and two output ports, the maximum a four-word-wide SRAM could support, and implemented logic to support port sharing. We instantiated an ID and AG at the select line of a multiplexer that chooses which port accesses the SRAM at any given time. Figure 6(a) shows a block diagram of the physical implementation of a push memory with two input ports and two output ports.

The performance of this memory depends on how much of the data in the wide fetch can be used. The design can maintain maximum performance in the common case, when the inner stride is one. The TB also allows this structure to transpose a matrix at full rate if it is fetched in 4×4 blocks.[1] To make the best use of this wide access memory requires the access patterns to be vectorized to automatically decouple the access pattern into sub-sequences and map them onto the controllers shown in Figure 6. This is performed in the vectorization pass described in the buffer mapping stage (Section 4.3). Of course, the bandwidth per port decreases when the access stride increases, since either the input or the output can always be streamed sequentially with inner stride = 1. In the worst case, the memory supports a throughput of four words every five cycles. For example, there is one write and four reads needed for four words.

We further optimized the energy and area of our PUB by observing that the sources and sinks of unified buffers have tightly coupled scheduling as any read from a memory ends in a write to a downstream memory in a statically determined number of cycles. In our hardware design after resource sharing from Figure 6(b), we only need one schedule generator to drive reads from the aggregator and subsequent writes to the SRAM. On the output side, this sharing is also possible, but a delay stage must be added between the schedule for reads from SRAM and writes to the transpose buffer since the SRAMs we use have a delay of one cycle for reads. With these optimizations, we have our final physical buffer design, called PUB. The PUB design uses programmable reads and writes on a single-port SRAM with wide fetches.

4 COMPILER DESIGN

Users of our system specify their applications in Halide, a high-level **domain-specific language (DSL)**. Halide separates the application algorithm from its schedule to isolate computation from optimizations in execution [40]. The algorithm specifies the computation of an output, while the schedule specifies the order in which the computation should be performed. Our compiler divides the problem of compiling Halide buffers to push-buffer implementations into three steps (shown in Figure 2):

(1) *Scheduling* leverages the Halide scheduling system, whose scheduling language controls loop transformations and that we extend with accelerator commands. We support a subset of the Halide input language that includes stencils, tensor computations, and lookup tables.
(2) *Buffer extraction* uses polyhedral scheduling techniques to turn the multidimensional iteration spaces of Halide loops into one-dimensional cycle times at every buffer port, thus yielding pipeline parallelism. The same step then extracts the full specification of each buffer port in the unified buffer abstraction, as shown in Figure 3.
(3) *Buffer mapping* maps the abstract unified buffers to physical buffers built from low-level hardware primitives based on the chosen compiler target.

We chose to keep the Halide scheduling language for tiling instead of placing it in the second step (like the PLUTO scheduling algorithm [3]). The reason is that a high-quality, general-purpose

[1]For a transpose, the SRAM is fetched in column order, where each fetch returns a short row of four elements. This access pattern fetches four stacked rows from the memory. The output port then reads the first element of each row, outputting the first column, and then the three other columns, before the next set of rows are fetched. When this inner loop completes, you have written the first four columns into the destination memory, and the outer SRAM loop moves to the next set of four columns.

Table 2. Halide Code Supported by Our Compiler Toolchain

Use case	Halide algorithm code	Support?	Memory data	Write Address	Read Address
Affine indexing	`out(x) = in(2*x) + 5 * x`	Yes	Data dependent	Affine	Affine
Stencil taps	`taps(x) = 1;` `taps(1) = 2;` `out(x) =` $\sum_{i=0}^{2}$ `taps(i)* in(x+i)`	Yes	Constant	Constant	Constant
Lookup table	`lut(x) = max(512, x*x);` `out(x) = lut(in(x))`	Yes	Constant	Constant	Data dependent
Non-affine indexing	`out(x) = in(x*y)`	No	Data dependent	Affine	*Non-affine*
Histogramming	`out(bin(x)) += 1`	No	Data dependent	*Data dependent*	*Data dependent*

Italicized fields are not supported by our system.

tiling algorithm for all dense linear algebra applications has not yet been found. As a result, we believe tiling is best left to either performance experts through a scheduling language or to domain-specific search procedures such as Reference [51]. Thus, we limit our use of polyhedral techniques to memory analysis and semantic-preserving loop fusion.

4.1 Scheduling

We extended the Halide scheduling language, which lets users define loop tiling but has no notion of push memories, to include commands to designate the portion of an application that should be placed on the accelerator. Figure 2 shows an example. The placement is done by defining the accelerator output with `hw_accelerate` and each of the accelerator inputs with `stream_to_accelerator`. After loop tiling, the user can designate the buffers that should become push memories, as opposed to fused with adjacent kernels, by using `store_at` and `compute_at`, along the lines of Reference [39]. Users can create a memory hierarchy by using the Halide scheduling directive `in` on buffers. Finally, the user can use `unroll` to designate that loops should be parallelized spatially as opposed to executed iteratively. After these scheduling directives, all following optimizations and mapping are performed automatically without user input.

Limitations to Addressing. The Halide frontend is used to specify our application, but our system does not handle some parts of the Halide language. Our compiler represents memory operations with three parts: memory read address, memory write address, and memory data. By analyzing these parts of a memory statement, we are able to classify which memory statements we can optimize, which statements are supported by our PUB hardware implementation, and which statements are not. Halide and our abstraction can express all of these different statements, but our backend compiler optimizations and hardware implementation handle a subset. Modifications to the mapper or hardware could enable and further optimize more of these use cases. Table 2 shows examples of supported and unsupported statements.

Our address controllers for PUB only handle affine expressions of index variables. During memory categorization, some memories are identified as unsupported if their indexing is data dependent or contains non-affine indexing. Our compiler identifies unsupported cases and throws an error that no mapping support exists for these memory operations. The fourth use case in Table 2 shows an unsupported example where the read address is non-affine. Histogramming is shown in the last row where the write addresses are data dependent, which is not supported. Data-dependent addressing introduces read-after-write dependencies that are not statically known, which require the compiler to be careful and conservative during scheduling. Non-affine expressions require the hardware to increment the address by a non-constant stride. Both of these cases are currently not supported by our compiler and hardware.

A special, supported case of data-dependent addresses is for a memory that holds constant, precomputed values. These include constant arrays and lookup tables as shown in the second and third use cases of Table 2. Stencil taps to a Gaussian filter are known statically during compile time, so we can preload them into registers. Another variant is where the read address is data dependent, also known as a lookup table. Our compiler analyzes the memory indices and memory values and identifies these cases. Lookup tables are precalculated with values and placed into memories functioning as basic SRAMs where the read address is connected to the data-dependent calculation.

Multiple Updates. One feature of Halide is multiple update statements to a single memory. A single memory can have values stored, and then particular addresses can be modified multiple times. For example, a memory could be initialized to store a constant, $mem(x, y) = 10$; then one column could be updated to a new value, $mem(x, 1) \mathrel{+}= in(x)$; and another update, $mem(2, y) \mathrel{+}= 3$. We choose to support only a single update statement for each memory in hardware so that basic accumulation is possible. Further updates are translated into their own memories, effectively converting the computation into a static single-assignment form.

One optimization for multiple updates is fusing unrolled reductions into a single statement. Our naive decision to use a memory for every update would result in a memory created for each update stage. Thus, a 3×3 convolution would have nine memory temporaries. Instead, we optimize this series of reduction statements, such as a series of adds, to a single statement. This effectively minimizes the number of memories by combining all of the compute kernels.

After these compiler passes and checks, the Halide compiler separates the Halide IR used for memories from the IR used for computation. The compute kernels are represented as graphs of operators and are used during the finishing steps. Meanwhile, the IR for memory operations is used for buffer extraction.

4.2 Buffer Extraction

The buffer extraction step analyzes the Halide IR to turn both loops and arrays into push memories expressed using the unified buffer abstraction. That is, Halide programs describe computation as operations on arrays over iteration domains defined by index variables. To compile the arrays to optimized push memories, buffer extraction analyzes array reads and writes to trace movement of values through memories. It then uses this information to distribute the control flow across the address generators in the push memories themselves.

Each static read from or write to a Halide buffer is given a unique port on the corresponding unified buffer. For each port, buffer extraction then computes an iteration domain, an access map, and a schedule. The iteration domain is the Cartesian product of the bounds of the loops surrounding the buffer access in the Halide IR and the access map is the buffer access expression. The main work of unified buffer extraction, however, is computing the cycle-accurate schedule that maps operations in the Halide program to the cycle times when they will happen in hardware.

Our cycle time scheduler exploits pipeline parallelism in two broad classes of workloads: stencil pipelines from image processing and coarse-grained pipelines from **deep neural networks (DNNs)**. Classical image processing applications, such as Harris corner detection, consist of many stencil operations that each produce pixels from small windows of input pixels. No single stage dominates the total compute cost and every pixel in a given stage depends on a small number of pixels in prior stages, making it cheap to create dedicated hardware for executing each pair of producer and consumer stages in parallel.

In DNNs, however, a single stage containing a large compute unit, such as a systolic array, dominates the compute cost of the application. Furthermore, values produced by that stage depend

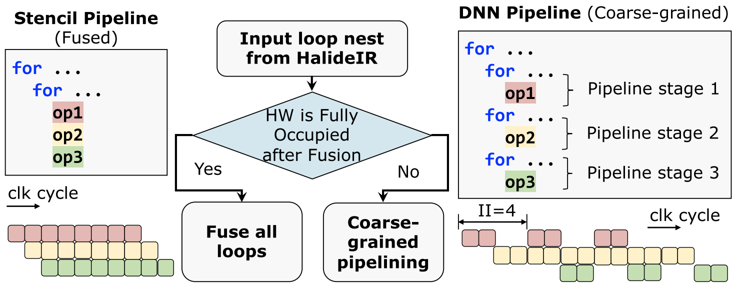


Fig. 7. During unified buffer extraction, the compiler chooses one of two schedulers based on temporal utilization of the compute hardware: either a stencil scheduler (left) or a coarse-grained scheduler (right). Each op corresponds to a buffer store and a set of loads.

on large number of values from prior stages, making it expensive to parallelize across stages. We create pipeline parallelization for both stencils and DNNs using line buffers and double buffers respectively.

The buffer extraction detects these two pipeline types separately. Then it selects the scheduling policy with a simple rule: If the most compute intensive stage in the pipeline can achieve full temporal utilization after loop fusion, then it uses a scheduling strategy that is tailored to stencil pipelines, which produces schedules that can be implemented efficiently using line buffers. Otherwise, if none of the stages can fully occupy the computation hardware after loop fusion, then it uses an algorithm tailored to the DNN-style pipeline, which uses coarse-grained pipeline parallelism and double buffering to maximize utilization of the most expensive compute unit, as shown in Figure 7. Both policies use the polyhedral analysis tool ISL [47] to compute data dependencies between operations and to solve the optimization problems used in formulating the schedule.

Stencil Pipeline. If the pipeline is classified as a stencil pipeline, then we apply the scheduling algorithm described by Huff et al. [15]. This algorithm produces a cycle-accurate schedule in two stages. First, it fuses all loop nests in the application into a single perfect loop nest. Then it computes a cycle-accurate schedule for the fused, perfect loop nest at an initiation interval of one (II = 1). The fusion is done incrementally, from the outermost loop level to the innermost. The fusion procedure uses an SDF-style constraint problem to set the relative rate and delay of each operation in a way that makes dependence distances as small and uniform as possible. To be specific, a global schedule is constructed with the constraint that the start of a statement must happen after the latest end time among all of its producer statements. The end time of a statement is calculated using the start time, the latency of memory loads, the latency of memory store, and the latency of its computation. This is demonstrated in the following equations:

$$Start(\texttt{stmt}) > End(\texttt{prod}) \quad \forall \texttt{prod} \in \texttt{stmt}'s\ producers, \tag{1}$$

$$End(\texttt{stmt}) = Start(\texttt{stmt}) + L_{mem\_ld} + L_{mem\_st} + L_{compute\_kernel}. \tag{2}$$

Once fusion is finished, we compute a cycle-accurate schedule for the loop nest using a standard **High level synthesis (HLS)** loop scheduler [56], which sets the II of every outer loop and delay of all operations in the innermost loop. Once the schedule is set, each unified buffer behavior is defined on its ports and the capacity is sized so that each pixel can be store until needed.

Coarse-grained Pipeline. The DNN buffer extraction creates a schedule for a double-buffered pipeline. This pipeline is coarse grained: Operations on one tile of an image proceed sequentially but are overlapped with operations on the next tile of the image. So, for example, while a computation is being performed on a tile that has already been loaded onto the accelerator, the next tile is being loaded onto the accelerator. Buffer extraction first identifies the overlapping tile loops to form the coarse-grained pipeline. It then walks from the root of the program inward and collects loop nests up to and including the innermost loop whose body is not a single perfect loop. These perfectly nested loops form the outer coarse-grained pipeline. We refer to the operations inside the pipeline as pipeline stages, but these stages are themselves typically extracted from loop nests. For instance, in the coarse-grained pipeline pseudo code shown in Figure 7, the outer coarse-grained loop, `for` loop on line 1, is pipelined and it contains three pipeline stages.

With the coarse-grained pipeline loops selected, the buffer extraction creates a cycle-accurate schedule for each pipeline stage using the same scheduler as for the stencil pipeline. It then creates the coarse-grained pipeline by laying out each pipeline stage sequentially and setting the **initiation intervals (IIs)** of the coarse-grained pipeline loops to the sequential execution latency. We refer to this as sequential scheduling.

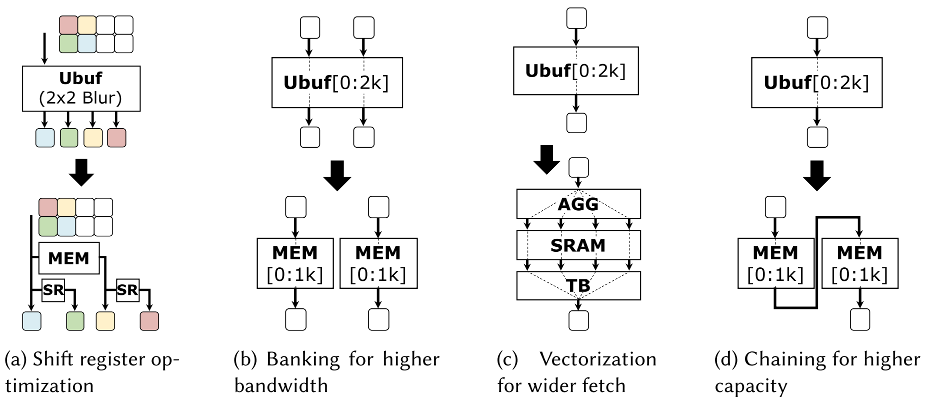
Next, the compiler applies double buffering to achieve better throughput, which reduces the IIs of the coarse-grained pipeline loops to the latency of the most compute-intensive stage. For example, the latency of operations in the DNN pipeline in Figure 7 are 2, 4, and 2 cycles, so the schedule has a coarse-grained pipeline with II = 4. A standard HLS loop scheduler uses `for` loops as boundaries of pipelining. Nested and imperfect loops at the boundary will result in redundant pipeline flush stages at the end of each loop [39, 52]. The same reasoning applies to our coarse-grained pipeline. To reduce this extra latency in DNN pipelines, buffer extraction applies loop perfection and loop flattening. Loop perfection pushes all coarse pipeline stages under one `for` loop with `if` guards, and loop flattening merges all loops above the coarse-grained pipeline into one single, merged loop.

4.3 Physical Buffer Mapping

With scheduling finished, all operations have been assigned to unstalled clock cycles (one-dimensional affine schedules), and the bandwidth of each memory is known. The next task of the compiler is to map the abstract unified buffers to implementations built out of the available physical primitives. This mapping produces the configuration bits for each physical buffer used in the design. In principle, the unified buffers can be mapped directly to physical buffers on the target accelerator. In practice, however, this is rarely possible for the following reasons:

- **Limited buffer bandwidth.** The physical buffers on the accelerator may not have sufficient bandwidth. For example, our PUB only has a single four-word-wide SRAM in each physical buffer, meaning that each buffer can support at most four memory operations per cycle. However, the unified buffer from our *brighten* example performs five memory operations per cycle, and many image processing patterns need even larger bandwidth.
- **Wide fetch width.** The accesses in the Halide program may have a bitwidth that is narrower than the bitwidth of the underlying SRAMs. For example, accesses to the four-word-wide SRAM in the physical buffers we built are done in vectors of four 16-bit integers, with four-word data vectors buffered in the aggregator and the transpose buffer between writes and reads to the SRAM.
- **Limited buffer capacity.** The cycle-accurate scheduler reduces storage requirements by bringing the consumer closer to the producer, but unified buffers may need more space than any single physical buffer.

Shift Register Optimization and Banking. To address the need for high bandwidth, each unified buffer must be broken into multiple smaller unified buffers to increase the number of memory ports. Our compiler uses two strategies for servicing high bandwidth accesses: shift register optimization and banking. Shift register optimization is possible whenever the dependence distance



(a) Shift register optimization (b) Banking for higher bandwidth (c) Vectorization for wider fetch (d) Chaining for higher capacity

Fig. 8. Transformations applied to map abstract unified buffers (Ubuf) to physical buffers (SR: Shift Register, AGG: Aggregator, TB: Transpose Buffer, MEM: memory tile).

(delay) between a port (the source) and another (the destination) is constant and the set of values that appear on the source is a superset of the values that appear on the destination. Our compiler performs an exhaustive shift register analysis that finds all opportunities to convert memory output ports into ports driven by shift registers fed from fewer memory ports. For instance, according to Figure 3, the buffer feeding the 2×2 blur kernel has four output ports, whose dependence distances to the input port are 0, 1, 64, 65 respectively. As shown in Figure 8(a), these three delays can be implemented with two shift registers and a physical buffer that delays by 64 cycles.

After shift register optimization, the remaining ports must be serviced from banks of memory with address generators (Figure 8(b)). Our compiler uses a version of an optimal banking algorithm for stencil computations [9]. It first groups the ports with overlapping schedules, meaning they access the buffer at the same time. If they access different parts of the buffer, then our compiler banks the memory, meaning it splits the memory into sub-memories dedicated to each port. This increases bandwidth over having the ports take turns reading. The compiler does the memory splitting by comparing the access maps of the ports to determine the block of the buffer that each needs. These blocks then become the banks. If the compiler cannot find a partition, then it falls back to exhaustive banking by creating a bank between each pair of input and output ports that access the same data.

Vectorization. To make efficient use of physical buffers with wide-fetch SRAMs, the access patterns of the buffers must be broken into sub-sequences with the same length as the SRAM fetch width as shown in Figure 8(c). At each input port of the buffer, this sub-sequence is assembled serially by the AGG. Once the aggregator is full, the sub-sequence is written to the SRAM. When the TB at an output port is empty, it receives a new sub-sequence from the SRAM, which it then sends out serially on the output port.

We can think of the introduction of the AGG, SRAM, and TB components as strip-mining the innermost loops of the original program and adding wide fetch-width loads and stores. The compiler generates the access maps and schedules at the SRAM ports and records them in the abstract unified buffer. It also adjusts the schedules of aggregator-SRAM and SRAM-transpose buffer transactions to minimize the storage requirement in the AGG and TB while respecting data dependencies and hardware resource limitations.

As mentioned in Section 3.2, when utilization of fetched words is too low, the buffer will be scheduled so that its overall rate is reduced. The transpose buffer keeps a local storage that helps during transposes, but cannot alleviate applications that have no locality.

Table 3. Halide Applications Used in the Evaluation Section

Application	Type	Description
gaussian	stencil	3×3 convolutional blur
harris	stencil	Corner detector using gradient kernels and non-maximal suppression
upsample	stencil	Up sampling by repeating pixels
unsharp	stencil	Mask to sharpen the image
camera	stencil	Camera pipeline with demosaicking, image correction, and tone scaling
laplacian	stencil	Gaussian and Laplacian pyramid with three levels
resnet	DNN	ResNet layer using multi-channel convolution
mobilenet	DNN	MobileNet layer using separable, multi-channel convolution
gemm	DNN	General matrix multiplication

Address Linearization. The access pattern in the unified buffer abstraction supports an arbitrary number of data dimensions, but a physical buffer requires the N-dimensional addresses to be converted to a single dimension. So, an inner product is applied between each N-dimensional address $\vec{a}$ and an offset vector $\vec{o}$ that encodes the memory layout: $\text{MEM}[a_0, a_1, \ldots, a_{N-1}] \rightarrow \text{MEM}[\Sigma^i a_i \cdot o_i]$.

Chaining. To map unified buffers with higher capacity than any one physical buffer, buffer mapping chains several buffers into a single logical buffer (Figure 8(d)). Each memory tile on the CGRA is assigned a unique tile ID. Our compiler statically analyzes the access map and the schedule of the unified buffer and partitions the access map into pieces implemented by multiple chained physical buffers. The following equations transform a logical address a in the access map into a tile ID and a physical address in the memory tile, using the capacity C of the memory tile:

$$\text{TileID}(a) = \text{floor}(a/C) \qquad \text{PhysicalAddress}(a) = a \bmod C.$$

Memory Hierarchy. Constructing a memory hierarchy involves copying from a unified buffer with a large capacity to a smaller unified buffer. On our CGRA, we refer to the large, outer memory as the global buffer, while the smaller ones are called memory tiles. The global buffer uses ready-valid signaling to connect to the processor's memory system, and will stall the execution engine if a block of data has not been loaded before the time it needs to be pushed to the memory tiles. Our unified buffer abstraction is agnostic to the memory hierarchy level until the mapping stage. We thus carry hierarchy information downstream until hardware mapping where we generate the correct configuration for each physical memory primitive.

Finishing Steps. After we have finished generating the configuration information for all the physical buffers, we map the compute kernels produced by the Halide frontend to **processing elements (PEs)** on the CGRA. We **place and route (PnR)** this mapped graph of PEs and physical buffers on the CGRA following standard multi-stage optimization by performing global PnR followed by detailed PnR to obtain the final configuration bitstream.

5 EVALUATION METHODOLOGY

To evaluate our compiler, we use it to compile the applications listed in Table 3 to CGRAs and compare the resulting performance to a Zynq UltraScale+ 7EV FPGA. The applications span stencil operations in image processing and tensor operations in deep neural networks as found in previous Halide scheduling papers [1, 32]. To generate an FPGA bitstream, our compiler transforms the buffers in each Halide application into unified buffers and applies all optimizations. We build upon the work by Huff et al. [15] to generate synthesizable C code that we feed into

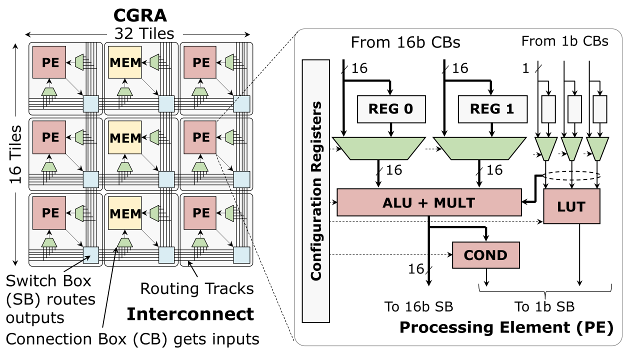


Fig. 9. Our CGRA is a 16 × 32 array of PE and MEM tiles. One-fourth of the tiles are MEMs and the rest are PEs. The memory tile contains the optimized PUB described in Section 3.2 depicted in Figure 6(b).

Vitis HLS. Our system generates identical synthesizable C as Huff et al., which compares favorably to other competitive DSL-FPGA systems [15]. The HLS output is fed into Xilinx's Vivado system that synthesizes, places, and routes the resulting design at 200 MHz. We use Vivado [50] to report resource consumption, energy use, and performance.

Our CGRA, shown in Figure 9, resembles an island-style FPGA, with LUTs replaced by PE tiles with 16 bit integer/floating-point ALUs, and BRAMs replaced by **memory (MEM)** tiles with different unified buffer implementations, including our optimized PUBs. The CGRA is embedded in a full system-on-chip. It directly connects to a large multi-banked, double-buffered memory called the global buffer. The global buffer has 16 banks; each bank is 256 kB and connects to a different section of the top edge of the CGRA. The data tiles required by the CGRA are first brought into the global buffer and then streamed into the CGRA. This allows computation on the current tile in the CGRA to be overlapped with the movement of the next tile into the global buffer. The global buffer provides deterministic access latency to the CGRA and hides the non-deterministic latency of the main memory. When targeting the CGRA, our compiler outputs a logical description of the design that is fed into custom mapping, placement, and routing tools designed for this CGRA. To generate power and area numbers, we created a complete Verilog design of the CGRA and used Cadence Genus and Innovus tools to synthesize, place, and route the MEMs and PEs of the CGRA in a 16-nm technology at 900 MHz. Power numbers are extracted from gate-level simulations.

6 EVALUATION

Using the unified buffer abstraction throughout the compilation process allows our compiler to map to many different implementations, ranging from FPGAs to our CGRAs with optimized PUBs. This is detailed in Section 6.1. Section 6.2 then uses this flexible backend to evaluate the performance gains by moving to a better physical memory implementation. This separation also allows all implementations to benefit from compiler optimizations that can improve the design by an order of magnitude as shown in Section 6.3. Finally, Section 6.4 shows that the capability to compile to coarse-grained accelerators is important, since our system is many times better than the same design mapped to an FPGA.

6.1 Backend Portability

The unified buffer abstraction enables our compiler to map to a broad set of physical memory implementations, as shown in Table 4. By keeping all of the application data stream information together, our backend can select the data it needs to configure the target hardware. Figure 10

Table 4. The Characteristics of our PUB Memory Primitive and Alternative Memory Implementations

Memory Backend	**PUB (ours)**	DP-SRAM + AG	DP-SRAM + PEs	Ready-valid (Buffet)	BRAM + LUTs
SRAM Macro	**SP**	DP	DP	DP	DP
Built-in AG	**Yes**	Yes	No	No	No
Control Protocol	**Static**	Static	Static	Ready-valid	Static
Accelerator Architecture	**CGRA**	CGRA	CGRA	ASIC	FPGA
Unified Buffer (ours)	✓	✓	✓	✓	✓
Vivado HLS [50]	✗	✗	✗	✗	✓
SODA [6]	✗	✗	✗	✗	✓
PolyEDDO [35]	✗	✗	✗	✓	✗

Our compiler, using the unified buffer abstraction, supports more memory implementations as compared to FPGA compilers and other accelerator compilers.

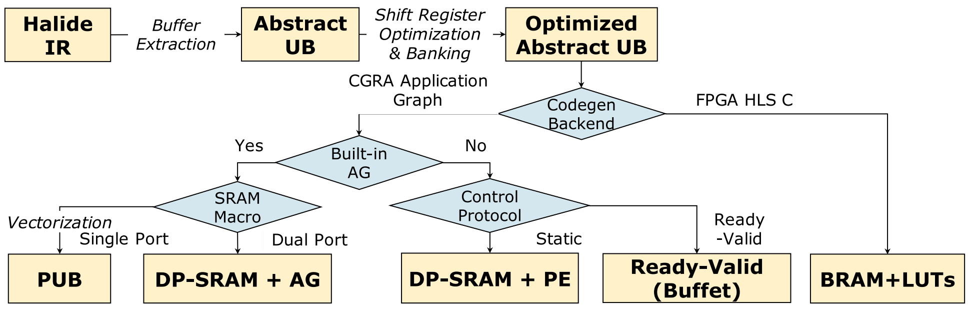


Fig. 10. Compilation process targeting different memory backends.

sketches the compilation process, denoting the classes of hardware our system can target. Our compiler first transforms the buffers in the Halide IR into unified buffers and applies optimizations including shift register optimization and banking to get the optimized unified buffers. These steps schedule all operations on the buffers, creating the addressing and scheduling information. The buffer mapping code generation separates into two different backends: one for generating configurations for CGRAs and one for targeting FPGAs. For FPGAs, the buffer specification is converted into C-loops that are then fed into an FPGA HLS tool to map the design onto **BRAMs + LUTs**. For CGRA targets, the buffer ports are connected to the hardware kernels forming an application graph that is then mapped onto the CGRA.

Our compiler encounters a divergence point in the CGRA backend based on whether controllers are pre-built into physical buffers. If the buffers do not contain controllers, then the compiler will generate the controllers needed for the specific stream pattern each buffer requires. These controllers are then added to the non-memory part of the application graph that is mapped into PEs on a CGRA. This part of the compilation covers machines that have simple **dual-ported (DP)** memories as physical buffers, **DP-SRAM+PE**, as well as more complex memories like **buffets** shown at the bottom right of Figure 10. A buffet [36] is a more sophisticated buffer implementation that tracks data dependencies between the read and write ports, and supports a ready-valid interface. Since the ready-valid protocol with dependency checking maintains almost all of the scheduling constraints,[2] when mapping to buffets, our compiler does not create a schedule generator and drops most of the cycle-level scheduling information. Of course, using these memories

[2]A few constraints remain. For example the shift-register technique to reduce memory ports needs to be carefully implemented in a ready-valid system.

Table 5. Total Memory Area and Energy for a 3×3 Convolution Using Different Implementations of the Physical on-chip Storage

	References	MEM Area (μm^2)	SRAM Area (%)	MEM Energy (pJ/access)	SRAM Energy (%)
Buffet[3]	[36]	14.3k	86	3.9	84
DP SRAM + PEs	[2, 29]	31.1k	40	4.8	70
DP SRAM + AG	[21, 38]	16.7k	74	3.6	92
4 Wide SP SRAM + AGG + TB + AGs	Ours	13.7k	42	2.5	61

Both area and energy decrease as we specialize the physical buffer. Total area and energy include control logic and address generation except for buffet.

requires that all the PEs in the CGRA support ready-valid ports. Adding this support to our CGRA was the major change needed to support buffets.

If the controllers are built into the physical buffers, then the compiler configures the embedded controllers to implement the stream access patterns on all of its ports. This path is taken for the physical buffers described in Section 3, **DP-SRAM+AG** and **PUB**, which are the left bottom branches of Figure 10.

Different from the previous compilers for reconfigurable hardware listed in Table 4 that target a specific architecture backend, the unified buffer abstraction separates the compiler frontend and its optimization machinery from the implementation details of different memory backends. This design allows it to target multiple memory backends on reconfigurable accelerators with little target-specific code. It also opens the opportunity to separately optimize the compiler frontend and the memory backend.

6.2 Hardware Optimization Case Study

We leverage our flexible backend to compare different approaches for physical buffer design on a CGRA. There are two major ways to build on-chip storage [37]: You can either use reconfigurable compute units to do both computation and address generation [2, 29] (DP-SRAM + PEs and buffet) or you can include address generators in the memory components [21, 38] (DP-SRAM + AG and PUB). Comparing the second row with the third row of Table 5 shows that even for a simple application, it is more efficient to use embedded address generators than to use the PEs on the target platform. Adding this logic to a dual-port 2048×16 bit SRAM (Figure 4) reduces the total unified buffer area by 46% and energy by 25% compared to implementing the addressing and control on PEs. We achieve further improvements by replacing the DP SRAMs with **single-port (SP)** SRAMs. The area of the dual-port $2{,}048 \times 16$-bit SRAM is around 2.5× larger than a single-port 512×64-bit SRAM with the same capacity. Thus, as the fourth row of Table 5 shows, even with the extra aggregation and transpose logic, using a wider single-port SRAM results in a buffer that is 18% smaller and consumes 31% lower energy than the best dual-ported version.

We synthesize a buffet implementation with the same dual-port $2{,}048 \times 16$-bit SRAM macro. The SRAM area proportion is lower than what is reported by Pellauer et al. [36] due to the added interconnect needed for CGRA reconfiguration. Although the buffet controller takes a smaller proportion of area, its function is only equivalent to the schedule generator in our PUB controllers and does not contain the address generation capability. Even if we ignore the missing address

[3]Since a buffet does not have an address generator in its design, the area and energy shown in the table do not include address generation.

Table 6. Memory Usage and Latency Comparison between Different Physical Memory Implementations

# of memories	PUB (ours)	DP+AG	Buffet
upsample	1	2	2
gaussian	1	2	6
harris	5	11	23
resnet	80	80	80

Latency (cycles)	PUB (ours)	DP+AG	Buffet
upsample	16399	16383	16401
gaussian	4095	4095	4100
harris	4095	4095	4139
resnet	9807	8739	8751

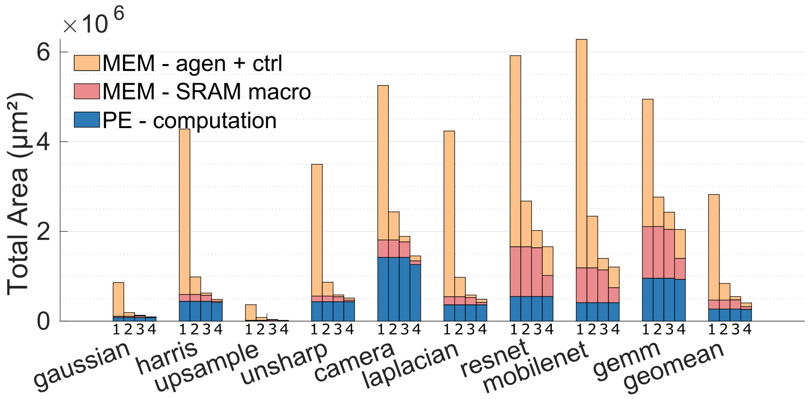


Fig. 11. Comparison of area usage for different memory implementations with stacked bars divided into MEM addressing and control area, MEM SRAM macro area, and PE area used for computation. Four different implementations of a physical buffer are evaluated: (1) a DP SRAM with unmodified PEs for address generation, (2) a DP SRAM with PEs optimized for address generation, (3) a DP SRAM with optimized address generator (AG), and (4) our final PUB with a single-port SRAM with fetch width of 4, AGG, and TB each with their AGs.

generator, our PUB memory is smaller and more area efficient. Using more complex controllers and single ported memories seems to be the best strategy for building physical buffers.

The advantage of the PUB implementation is even larger than Table 5 indicates. Our PUB's four-word-wide fetch allows it to support two input ports and two output ports, which double the peak read/write bandwidth compared to the dual-port memories. As shown in Table 6(left), PUB requires fewer physical buffers for all applications besides resnet.

While using wide-fetch memories has many benefits, it also has some costs. It requires the generated schedules to be padded to align with the fetch width. As shown in Table 6(right), this padding usually does not affect the latency, but can have a modest effect if the size of some of the data blocks is small, as it is in resnet.

Finally, we map nine applications to three CGRA architectures with different physical buffer implementations and evaluate their total area. To map to the memory backend without address controllers, we generate the controllers using PEs based on the number of dimensions in the unified buffer schedule. In our unoptimized version (1), we make no modifications to the PEs. In the optimized version (2), an operator for a counter is added to the PE; this change saves 67% area. As shown in Figure 11, using dedicated AG logic in the dual-port memory tile lets PEs be used exclusively for computation. Note that these area savings occur while the throughput stays the same. Furthermore, using a wide, single-port SRAM with more external ports saves silicon area, while expanding functionality. Both of these properties lead to an average 2.2× less total area needed to implement the same application as compared to DP + optimized PEs. From the breakdown in Figure 11, SRAM macro area reduces 3.3 times, and memory controller area reduces 4.5 times,

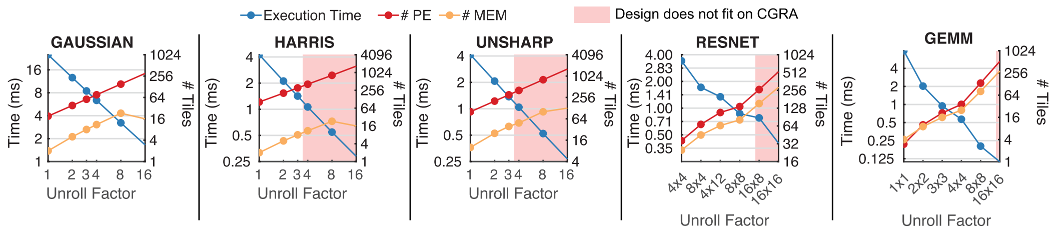


Fig. 12. Execution time versus resource utilization tradeoff by using Halide's scheduling. At high unrolling factors, designs do not fit on the CGRA (384 PEs, 128 MEMs); this is indicated on the charts by the red shaded regions.

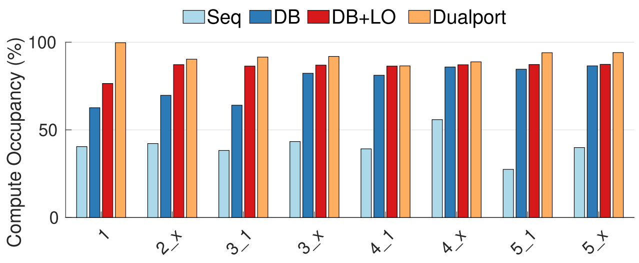


Fig. 13. Ablation study of the effectiveness of coarse-grained loop optimizations. For ResNet layers, compute occupancy increases as double buffering (DB) and loop optimizations (LO) are added. A dual-port memory leads to even higher compute occupancy.

while PE area remains the same. The area savings are even greater in deep learning applications, which are memory intensive.

This case study shows the importance of building physical buffers with efficient and customized controllers that can extract the most performance out of each memory macro. Performing these optimizations yielded a physical memory implementation that is half the area and energy of the original design.

6.3 Evaluation of Compiler Optimizations

Reinterpreting Halide Schedules for Dataflow Architectures. Halide provides the user with a language where scheduling primitives are used to explore a design space. As users, we can create Halide schedules to explore the tradeoff between reducing latency and using more resources. Figure 12 shows how the system performance scales when Halide's `unroll` scheduling primitive is reinterpreted to create parallel compute hardware. The execution time decreases linearly on the log scale, which means our designs scale well for execution time as more resources are used. The number of MEMs used decreases at very high unroll factors (16 for gaussian and harris) because our compiler replaces line buffers having fewer than 20 words with register chains. Each design eventually fails to map to our target CGRA when it exceeds 384 PEs or 128 MEMs (in Figure 12, designs that exceed available resources are shaded in red). Based on this investigation, an application designer would use an unroll factor of 16 for gaussian, 3 for harris, 3 for unsharp, 16×8 for resnet, and 8×8 for gemm to fully utilize our target CGRA.

Coarse-grained Loop Optimizations. We map all eight ResNet convolutional layers to our CGRA. Each layer is properly blocked with eight input channels and eight output channels computed in parallel. Our compiler then schedules the execution of each layer using the loop optimization describe in Section 4.2. As shown in Figure 13, we compare three versions of the coarse-grained

Table 7. Shift Register Optimization Replaces Memory Tiles with Registers or Wires

	Original MEMs	MEMs after optimization	Savings (%)	Registers added
gaussian	9	1	89%	6
harris	67	5	93%	30
unsharp	66	6	91%	40
camera	158	25	84%	26
resnet	136	81	40%	0

pipeline schedules based on compute occupancy, which is the proportion of time that the PEs are utilized. Compared to the sequential scheduling baseline, double buffering significantly increases compute occupancy by overlapping data transfers with computation. Applying loop flattening and loop perfection increases the compute occupancy by around 10%. Notice that this optimization is more effective for the early layers in the DNN where the feature maps have larger spatial sizes. Tiling the width and height of the input feature maps creates an overhead from extra nested tiling loops. The loop optimizations remove this overhead. While the 4-wide memories help with energy-efficiency, we also see that they hold occupancy back by approximately 10% as compared to using a dual-port RAM with a fetch width of 1. As mentioned in the prior section, the memory needs to wait for extra cycles while fetching useless data when the data do not align properly in the wide-fetch memories.

Shift Register Optimization. Another optimization we perform is the shift register optimization to reduce memory tile usage. As described in Section 4.3, memories with a small dependency distance between read and write can be replaced with registers or wires. Table 7 shows how many of the memories are replaced compared to the naïve implementation. In stencil applications, registers are used to reuse the adjacent pixels in stencil windows. In ResNet, data from the same input channel is reused by compute for different output channels in parallel. Instead of duplicating input memories, this optimization instantiates a single memory and broadcasts values.

6.4 System Level Evaluation

The unified buffer abstraction lets us successfully compile a wide range of applications in Table 3 onto a CGRA. Having the same compiler generate code for both CGRA and FPGA enables us to fairly measure the energy efficiency benefit of our CGRA architecture. Although we use the FPGA code generated by our own compiler as the baseline, our FPGA backend is based on Huff's work [15], which demonstrated state-of-the-art performance against the leading FPGA compilers [6].

Figure 14 shows the resulting energy/operation consumed. The more efficient unified buffer implementation and optimized 16 bit logic mean that the CGRA is 3.5× more efficient than the FPGA. Figure 15 shows the applications' time per pixel on the CGRA, FPGA, and a CPU. Time per pixel is the runtime divided by the total number of output values. Our CPU comparison is an Intel Xeon 4214 with 16.5 MB cache with a 2.2-GHz base frequency. We use the same Halide application code for each backend, then validate the output images against each other. The CGRA is able to outperform the CPU, and dominates the FPGA with 4.7× faster runtimes due to its higher clock frequency. With these comparisons, we see that the compiler optimizations coupled with an efficient memory design lead to a competitive accelerator design.

7 RELATED WORK

Creating tools for application-tailored accelerators is an active area of research and our system builds on these ideas.

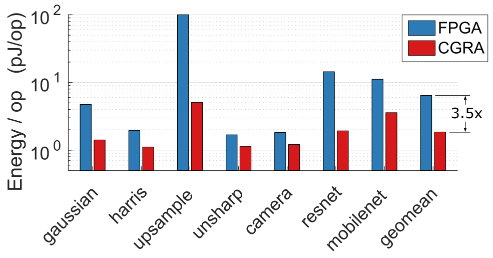


Fig. 14. Comparison of energy per operation for running kernels on a CGRA and FPGA.

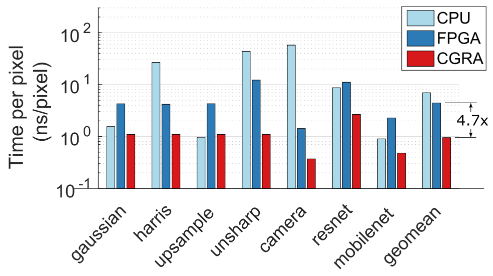


Fig. 15. Time per pixel on CGRA, FPGA, and CPU.

Compiler Frameworks. Neither conventional software compilers nor existing hardware compilers are well suited to target push memories. HLS tools such as Vivado [49], LegUp [4], Catapult [27], and others [17, 24], are designed to solve scheduling and resource binding problems at a finer granularity than those seen when compiling to push memories. Their strategy works well when targeting FPGAs or ASIC technology libraries, because the architectural primitives (such as registers and LUTs) are more fine-grained than the compiler IR instructions. When compiling to programmable push memory accelerators, where the architectural primitives are much more coarse-grained than a typical RISC instruction, this approach does not work. Academic languages such as Spatial [20], HeteroCL [22], and Exo [16] provide a more abstract programming model for accelerators, but the user must define the memory micro-architecture. Although HeteroCL uses a unified DSL frontend to describe their memory optimization and spatial architecture, their backend implementation still depends on separate frameworks. Exo provides more control to the user to enable more performance, but with this is increased complexity with user-defined memory management and accelerator functionality.

While HLS tools can translate the Halide IR directly to hardware, they do not support the memory optimizations we describe. Modern HLS tools such as Vivado HLS or Catapult HLS are well suited to arithmetic mapping and exploiting pipeline parallelism within the bodies of individual loops [25]. However, they perform limited memory [39] and cross-loop optimizations [57]. As a result, they are not good at exploiting pipeline parallelism across different loop nests in a computation and require a great deal of manual effort by users to create high-quality code for deep pipelines [22].

Push Memory Abstractions. Our unified buffer borrows from buffets [36], a buffer implementation idiom with **explicit decoupled data orchestration (EDDO)** that can be reused across multiple domains. Buffets are a hardware primitive, not a compiler abstraction, while our unified buffer is both. We improve productivity by using compiler techniques to extract, optimize, and map the buffers from high-level application code.

Recently a new abstraction called **hierarchical space-time (HST)** [35] was proposed to capture the memory behavior of EDDO architecture and an automated analysis/code generation framework called PolyEDDO is in progress. Different from our unified buffer that stays agnostic to the backend hardware, HST targets the EDDO architecture that relies on buffet controller's synchronization. It is not trivial for PolyEDDO to target other memory implementations like a scratchpad. Moreover, its frontend only supports perfect loop nests, while ours can support coarse-grained pipelines.

Some CGRA designs have proposed using PEs for calculating memory addresses [10, 26, 45]. However, most systems, like Plasticine [38], create dedicated addressing units associated with their push memories. Spatial [20] provides a high-level programming language for Plasticine but requires users to explicitly orchestrate data movement between different memories. SARA [55] improves upon the Plasticine compiler by scaling applications to utilize all hardware resources but leaves inter-loop optimizations to the user. Nowatzki [34] proposes a low-level programming model for stream dataflow accelerators. Since their memory architecture is a global scratchpad, their memory ISA contains dynamic scheduling that may not be suitable for accelerators with distributed push memories.

Domain-Specific Accelerator Generators. Other work seeks to automate FPGA and ASIC domain-specific accelerator design. Image processing accelerator generation languages such as Darkroom [13], Rigel [14], Aetherling [8], Hetero-Halide [23], HIPACC-FPGA [41], PolyMage-FPGA [7], SODA [6], and Halide-HLS [39] automatically generate FPGA implementations of image processing pipelines. These systems target either FPGAs that have large overheads or ASICs that are inflexible. AutoSA [48], AutoDSE [44], and Clockwork [15] are some systems that use polyhedral analysis for scheduling, but they do not consider CGRAs.

To efficiently execute DNNs, Zhang et al. [53] optimize DNN data blocking using double buffer structures and synthesize a pipelined FPGA accelerator from Caffe [18]. DNNWeaver [43] also generates synthesizable designs automatically from Caffe, with support for more types of layer implementations. DNNBuilder [54] proposes a fine-grained layer-based pipeline architecture with a line-buffer-based scheme to reduce FPGA on-chip memory usage. VTA [30, 31] provides a full hardware-software stack for DNN acceleration using a modified version of Halide IR. It proposes an ISA to map DNN layers onto optimized operators on their proposed FPGA accelerator. These domain-specific hardware generators reduce design effort when mapping a DNN to an accelerator. However, their frameworks heavily rely on the backend implementation. With the architectures determined, extending them to support new applications or more efficient hardware implementations would require significant development effort from domain experts.

8 CONCLUSION

The growing importance of accelerators, and their dependence on push memories for high hardware utilization makes creating compilers for this abstraction increasingly important. We address this challenge by creating a new abstraction for push memories called a *unified buffer*, which supports efficient hardware realization and is a tractable target for an optimizing compiler. We validate this approach's potential by creating a compiler that is able to efficiently map image processing and machine learning applications onto a push memory accelerator and by creating PUB, an optimized version of a physical unified buffer suitable for a CGRA.

REFERENCES

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. DOI: https://doi.org/10.1145/3306346.3322967

[2] Oguzhan Atak and Abdullah Atalar. 2013. BilRC: An execution triggered coarse grained reconfigurable architecture. *IEEE Trans. VLSI Syst.* 21, 7 (2013), 1285–1298. DOI: https://doi.org/10.1109/TVLSI.2012.2207748

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08).* Association for Computing Machinery, New York, NY, 101–113. DOI: https://doi.org/10.1145/1375581.1375595

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'11).* Association for Computing Machinery, New York, NY, 33–36.

[5] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circ.* 52, 1 (2016), 127–138.

[6] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*. IEEE, New York, NY, 1–8.

[7] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation*. Association for Computing Machinery, New York, NY, 327–338.

[8] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. Association for Computing Machinery, New York, NY, 408–422.

[9] Juan Escobedo and Mingjie Lin. 2018. Graph-theoretically optimal memory banking for stencil-based computing kernels. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. Association for Computing Machinery, New York, NY, 199–208.

[10] Xitian Fan, Di Wu, Wei Cao, Wayne Luk, and Lingli Wang. 2018. Stream processing dual-track CGRA for object inference. *IEEE Trans. VLSI Syst.* 26, 6 (2018), 1098–1111. DOI: https://doi.org/10.1109/TVLSI.2018.2797600

[11] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *Int. J. Parallel Program.* 21, 5 (1992), 313–347.

[12] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, New York, NY, 243–254. DOI: https://doi.org/10.1109/ISCA.2016.30

[13] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (2014), 11 pages. DOI: https://doi.org/10.1145/2601097.2601174

[14] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.* 35, 4, Article 85 (2016), 11 pages. DOI: https://doi.org/10.1145/2897824.2925892

[15] Dillon Huff, Steve Dai, and Pat Hanrahan. 2021. Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 186–194. https://doi.org/10.1109/FCCM51124.2021.00030

[16] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'22)*. Association for Computing Machinery, New York, NY, 703–718. DOI: https://doi.org/10.1145/3519939.3523446

[17] Intel Inc. 2022. Altera OpenCL. Retrieved from https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*. Association for Computing Machinery, New York, NY, 675–678.

[19] Norman Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA'17)*. Association for Computing Machinery, New York, NY, 1–12.

[20] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. Association for Computing Machinery, New York, NY, 296–311.

[21] Philipp Käsgen, Mohamed Messelka, and Markus Weinhardt. 2021. HiPReP: High-performance reconfigurable processor - architecture and compiler. In *Proceedings of the 31st International Conference on Field-Programmable Logic and Applications (FPL'21)*. IEEE Computer Society, Los Alamitos, CA, 380–381. DOI: https://doi.org/10.1109/FPL53798.2021.00074

[22] 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 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. Association for Computing Machinery, New York, NY, 242–251. DOI: https://doi.org/10.1145/3289602.3293910

[23] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. Association for Computing Machinery, New York, NY, 51–57.

[24] Maxeler Inc.2022. MaxCompiler. Retrieved from https://www.maxeler.com/products/software/maxcompiler.

[25] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An overview of today's high-level synthesis tools. *Des. Autom. Embed. Syst.* 16, 3 (2012), 31–51.

[26] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*. Springer, Berlin, 61–70.

[27] Mentor. 2019. *Catapult Synthesis User and Reference Manual*. Mentor, Wilsonville, OR.

[28] Mentor Graphics Inc. 2022. Catapult High Level Synthesis.

[29] Mirsky and DeHon. 1996. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, New York, NY, 157–166. DOI: https://doi.org/10.1109/FPGA.1996.564808

[30] Thierry Moreau, Tianqi Chen, and Luis Ceze. 2018. Leveraging the VTA-TVM hardware-software stack for FPGA acceleration of 8-bit ResNet-18 inference. In *Proceedings of the Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning (ReQuEST'18)*. Association for Computing Machinery, New York, NY, Article 5, 7 pages. DOI: https://doi.org/10.1145/3229762.3229766

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

[32] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling Halide image processing pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July2016), 11 pages. DOI: https://doi.org/10.1145/2897824.2925952

[33] Vivek Nautiyal, Gaurav Singla, Lalit Gupta, Sagar Dwivedi, and Martin Kinkade. 2017. An ultra high density pseudo dual-port SRAM in 16nm FINFET process for graphics processors. In *Proceedings of the IEEE International System-on-Chip Conference (SOCC'17)*. IEEE, New York, NY, 12–17.

[34] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. *SIGARCH Comput. Archit. News* 45, 2 (June2017), 416–429. DOI: https://doi.org/10.1145/3140659.3080255

[35] Angshuman Parashar, Prasanth Chatarasi, and Po-An Tsai. 2021. Hardware abstractions for targeting EDDO Architectures with the Polyhedral Model. In *Proceedings of the 11th International Workshop on Polyhedral Compilation Techniques (IMPACT'21)*. HiPEAC.

[36] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Association for Computing Machinery, New York, NY, 137–151. DOI: https://doi.org/10.1145/3297858.3304025

[37] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access* 8 (2020), 146719–146743.

[38] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA'17)*. Association for Computing Machinery, New York, NY, 389–402. DOI: https://doi.org/10.1145/3079856.3080256

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

[40] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM Sigplan Not.* 48, 6 (2013), 519–530.

[41] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. 2014. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'14)*. IEEE, New York, NY, 1–10.

[42] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. Association for Computing Machinery, New York, NY, 14–27. DOI: https://doi.org/10.1145/3352460.3358302

[43] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, New York, NY, 1–12.

[44] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems* 27, 4 (2022), 27 pages. https://doi.org/10.1145/3494534

[45] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-elastic CGRAs for irregular loop specialization. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, New York, NY, 412–425. DOI: https://doi.org/10.1109/HPCA51647.2021.00042

[46] Artem Vasilyev. 2019. *Evaluating Spatially Programmable Architecture for Imaging and Vision Applications.* Stanford University.

[47] Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *Proceedings of the International Congress of Mathematical Software (ICMS'10)*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, Berlin, 299–302.

[48] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. *The ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) (Virtual Event, USA)*. Association for Computing Machinery, New York, NY, 93–104. https://doi.org/10.1145/3431920.3439292

[49] Xilinx. 2019. *Vivado Design Suite User Guide High-Level Synthesis*. Xilinx, San Jose, CA.

[50] Xilinx Inc.2022. Vivado High Level Synthesis. Retrieved from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[51] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, New York, NY, 369–383.

[52] Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. 2010. Enhanced loop flattening for software pipelining of arbitrary loop nests. Technical Report. University of Washington, Seattle.

[53] 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 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. Association for Computing Machinery, New York, NY, 161–170.

[54] 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)*. Association for Computing Machinery, New York, NY, Article 56, 8 pages. DOI: https://doi.org/10.1145/3240765.3240801

[55] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a reconfigurable dataflow accelerator. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA'21)*. IEEE, New York, NY, 1041–1054. DOI: https://doi.org/10.1109/ISCA52012.2021.00085

[56] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*. IEEE, New York, NY, 211–218. DOI: https://doi.org/10.1109/ICCAD.2013.6691121

[57] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'13)*. Association for Computing Machinery, New York, NY, 9–18.

Received 1 June 2022; revised 2 November 2022; accepted 7 November 2022