Sirop: A Small IR for HLS with Parallel Patterns

Louis Hildebrand
McGill University
Montreal, Canada
louis.hildebrand@mail.mcgill.ca

Christophe Dubach
McGill University & Mila
Montreal, Canada
christophe.dubach@mcgill.ca

Abstract

Designers of custom streaming accelerators traditionally use HDLs (Hardware Description Languages), but this is time-consuming and requires advanced hardware expertise. C-based HLS (High-Level Synthesis) offers a higher level of abstraction and faster design time, but still requires some hardware expertise and performance is often left on the table. A promising direction is to use HLS with high-level functional parallel patterns such as `map` and `reduce`. Prior works have shown that high performance is achievable this way. However, designing such compiler systems is challenging because the optimizer must handle a large number of language primitives and interactions between them.

This paper introduces a minimal functional IR (Intermediate Representation) for hardware design, Sirop, which can express both pipelining and spatial parallelism with just five primitives. High-level operators from prior works are represented as syntax sugar and lowered to the core language. This simplifies hardware generation and optimization.

Sirop is compared with existing compilers on a set of image processing and linear algebra benchmarks. The Sirop designs use 61% fewer ALMs (Adaptive Logic Modules) than Aetherling, 68% fewer ALMs than Shir, and 76% fewer ALMs than the Intel HLS compiler, all for the same throughput.

CCS Concepts: • **Hardware** → **High-level and register-transfer level synthesis**; • **Software and its engineering** → *Compilers.*

Keywords: intermediate representation, high-level synthesis, functional programming

ACM Reference Format:
Louis Hildebrand and Christophe Dubach. 2026. Sirop: A Small IR for HLS with Parallel Patterns. In *Proceedings of the 27th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '26), June 15–16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3814943.3816175

LCTES '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2721-4/2026/06
https://doi.org/10.1145/3814943.3816175

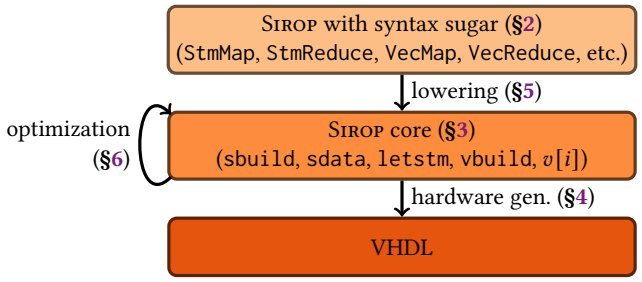

Figure 1. Steps in the Sirop compilation flow.

1 Introduction

Designing specialized accelerators is common in many domains from image processing to machine learning. Such accelerators offer higher performance and power efficiency than general-purpose processors. This is achieved by exploiting the fixed computational patterns found in the application and mapping these to spatial and pipeline parallelism.

Although specialized accelerators have significant benefits, they are difficult to design. Low-level HDLs such as VHDL and Verilog require deep hardware expertise. Even higher-level HDLs like Chisel [3] and Spatial [23], which simplify the process, demand hardware expertise.

C-based HLS lets designers define their hardware in C, with annotations manually added to guide the optimizer. This reduces design time but this still requires some expertise and the hardware may not reach peak performance [19, 29, 35].

A promising approach is to use HLS with parallel patterns from the functional programming paradigm, such as `map` and `reduce` [11, 19, 24, 25, 36]. Rewrite rules are used to lower operators step by step through different abstraction layers and apply optimizations such as operator fusion. This approach can produce high performance designs. However, implementing and testing a hardware template for each operator requires significant effort. At the time of writing, Shir [36] includes over 40 stream and vector operators and Aetherling [11] has over 30. It is even harder to define rewrite rules to optimize all *combinations* of operators. Moreover, some rewrite rules cannot be defined because the result would not be expressible with the available language primitives.

This paper addresses these challenges by introducing a small IR, Sirop, with just five primitives for stream and vector operations. All the familiar parallel patterns can be expressed as syntax sugar and lowered to the core Sirop language. Optimization and hardware generation are then performed on the smaller core language, as shown in figure 1.

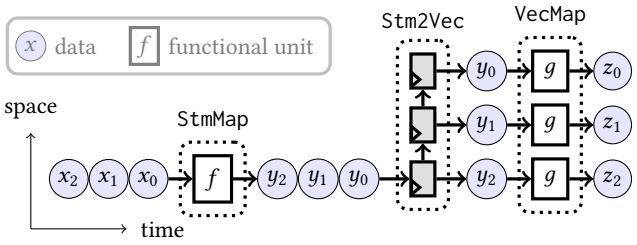

Figure 2. Stream and vector functional primitives.

The main contribution of this paper is the minimal IR for functional HLS (section 3). Its usefulness in practice is demonstrated by an optimizing compiler, which is described in sections 4 to 6. Specifically, the compiler demonstrates that the language has the following properties:

- It is easily translated to hardware (section 4).
- It can express familiar functional patterns for HLS, such as `StmMap` and `StmReduce` (section 5).
- It enables some more general optimizations, such as fusion of any two streaming operators (section 6).

Moreover, section 7 evaluates the compiler on a set of image processing and linear algebra kernels for an FPGA. The Sirop compiler generates more resource-efficient designs than prior works, including a commercial compiler (Intel HLS) and two state-of-the-art functional HLS compilers (Shir and Aetherling). This is thanks to its ability to fuse any streaming operators and its finer control over each operator's implementation during lowering and optimization.

2 Background and Overview

2.1 Functional Programming Primitives

Some prior works have used data-parallel patterns [15] from functional programming for hardware design. LiftHLS [24], Aetherling [11], and Shir [36] are recent examples. At the algorithmic level, operators are typically applied on sequences of data, which can be nested. Examples of such operators are shown below. Their semantics should be self-explanatory given their names and types.

$$\begin{aligned}
\texttt{count} &: (n\texttt{: Int}) \to \texttt{Seq}[\texttt{Int}, n] \\
\texttt{map} &: \texttt{Seq}[A, n] \to (A \to B) \to \texttt{Seq}[B, n] \\
\texttt{reduce} &: \texttt{Seq}[A, n] \to ((A, A) \to A) \to \texttt{Seq}[A, 1] \quad (n \geq 1) \\
\texttt{concat} &: \texttt{Seq}[A, n] \to \texttt{Seq}[A, m] \to \texttt{Seq}[A, n + m] \\
\texttt{zip} &: \texttt{Seq}[A, n] \to \texttt{Seq}[B, n] \to \texttt{Seq}[(A, B), n] \\
\texttt{split} &: (m\texttt{: Int}) \to \texttt{Seq}[A, nm] \to \texttt{Seq}[\texttt{Seq}[A, m], n] \\
\texttt{join} &: \texttt{Seq}[\texttt{Seq}[A, m], n] \to \texttt{Seq}[A, nm]
\end{aligned}$$

Here, n, m are natural numbers; A, B are types; and $\texttt{Seq}[A, n]$ is the type of a sequence of n elements of type A.

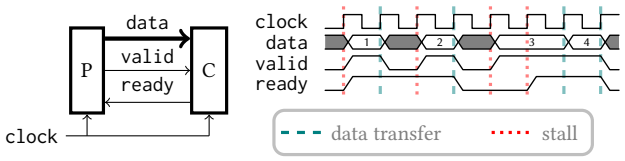

Figure 3. Streaming handshake protocol. The producer raises the `valid` signal when it has data to send and the consumer raises the `ready` signal when it can receive data.

```
(I0: Stm[u16, 4]) => (I1: Stm[u16, 4]) =>
  StmZip(I0, I1)
  .StmMap( @(x: u16, y: u16) => x * y )
  .StmReduce( @(sum: u16, x: u16) => sum + x )
```

Figure 4. Functional-style dot product of two streams. In this paper, the notation $x.f(y, z, \ldots)$ is equivalent to $f(x, y, z, \ldots)$.

2.2 Stream and Vector Hardware Primitives

A key advantage of hardware acceleration is the ability to customize the level of spatial and pipeline parallelism. Therefore, functional HLS tools typically expose at least two sequence types: *streams* and *vectors*. This paper uses the "stream" and "vector" terminology from LiftHLS [24] and Shir [36]. Aetherling's `TSeq` and `SSeq` (time and space sequences, respectively) are similar [11].

A *stream* can only be read once, one element at a time, in order. Operations on streams are processed in time, enabling pipeline parallelism. By contrast, *vectors* support random access, even access to multiple elements at once. Operations on vectors are processed in space, enabling spatial parallelism.

Most functional primitives have two versions: one for streams and one for vectors. There is `StmMap` and `VecMap`, `StmReduce` and `VecReduce`, and so on. There are also primitives `Stm2Vec` and `Vec2Stm` for converting between streams and vectors. These can be implemented in hardware as serial-in parallel-out and parallel-in serial-out shift registers. Figure 2 shows examples of stream and vector primitives.

Streams may be statically scheduled [11] or dynamically scheduled [20, 24, 36]. This paper uses dynamic scheduling with the handshake protocol illustrated in figure 3.

2.3 Overview

Figure 1 summarizes the Sirop compiler. The input is a program using stream and vector functional operators, like the dot product program in figure 4. Prior works [11, 36] have shown ways to lower operators like `map` to their stream and vector variants; this paper focuses on the IR instead.

The compiler lowers the program to the core language, which has only five stream and vector primitives: a vector constructor (`vbuild`) and extractor ($v[i]$), a stream constructor (`sbuild`) and extractor (`sdata`), and a `letstm` construct to share streams. These will be defined in the next section.

Type	T	::=	$\texttt{bool} \mid \texttt{i}_{n^+} \mid \texttt{u}_n \mid \texttt{fix}_{n,k} \mid T_1 \to T_2$	Booleans, numbers, functions
		$\mid$	$(T_0, \ldots, T_{n-1}) \mid \texttt{Vec}[T, n] \mid \texttt{Stm}[T, n]$	Collections
Data type	D	::=	$\texttt{bool} \mid \texttt{i}_{n^+} \mid \texttt{u}_n \mid \texttt{fix}_{n,k} \mid (D_0, \ldots, D_{n-1}) \mid \texttt{Vec}[D, n]$	
Expression	e	::=	$x \mid (x\colon T) \mapsto e \mid e_1(e_2)$	Variables, functions, function applications
		$\mid$	$e_1 + e_2 \mid e_1 * e_2 \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \ldots$	Standard logical, arithmetic operations
		$\mid$	$(e_0, \ldots, e_{n-1}) \mid e.n$	Tuple constructor and extractor
		$\mid$	$\texttt{vbuild}(n, (x\colon T) \mapsto e) \mid e_1[e_2]$	Vector constructor and extractor
		$\mid$	$\texttt{sbuild}(n)(e_1,\ e_2)\{\ \vec{a}\ \}\{\ \vec{p}\ \} \mid \texttt{sdata}(e)$	Stream constructor and extractor
		$\mid$	$\texttt{letstm}[n]\ x = e_1\ \texttt{in}\ e_2$	Stream sharing
Accumulator	a	::=	$(x\colon D) = \{\texttt{init:}\ e_1,\ \texttt{next:}\ e_2\}$	
Producer	p	::=	$(x\colon \texttt{Stm}[D, n]) = \{\texttt{stm:}\ e_1,\ \texttt{ready:}\ e_2\}$	

Figure 5. Abstract syntax of Sirop. $n, k \in \mathbb{N}$ and $n^+ \in \mathbb{N}^+$ are natural numbers. $\texttt{i}_{n^+}$ and $\texttt{u}_n$ are the types of n-bit signed and unsigned integers, respectively. $\texttt{fix}_{n,k}$ is for fixed-point numbers with n total bits and k fractional bits. $\vec{e}$ is an arbitrary number of expressions $e_1, e_2, ..., e_n$. Most logical and arithmetic operations are omitted for brevity; they are standard.

The eager reader can find the lowered code for the dot product example in figure 7, but it will be explained later.

Finally, after running several optimization passes, the compiler generates a VHDL description of a dynamically-scheduled streaming accelerator. Section 7 will show that Sirop generates more resource-efficient hardware than the Intel HLS compiler, Aetherling [11], and Shir [36].

3 The Core Language

The core language is a typed lambda calculus with expressions for booleans, integers, fixed-point numbers, tuples, vectors, and streams, as shown in figure 5.

Vectors. Vectors have only one constructor:

$$\texttt{vbuild}(n, f) = [f(0), f(1), ..., f(n-1)]$$

Vectors also have an extractor, $v[i]$, which returns the element at index i in vector v. The `vbuild` primitive was introduced by prior work on compiling functional array programs to imperative code [37].

Stream Constructor. The idea of having just one constructor and one extractor per collection type can be extended to streams. The expression

$$\texttt{sbuild}(n)(\texttt{data},\ \texttt{valid})\ \{\ \vec{a}\ \}\ \{\ \vec{p}\ \}$$

defines a stream of length n whose output is given by `data` and `valid`. The stream has a set of accumulators $\vec{a}$ modeled as recurrence equations. The stream also has a set of input producing streams $\vec{p}$, each with a corresponding `ready` expression. The `next`, `ready`, `data`, and `valid` expressions can depend on any accumulators and producers.

`sbuild` can implement any of the higher-level streaming operators (e.g., `StmMap`, `StmReduce`). This will be discussed further in section 5. Intuitively, it has the required properties:

- It can choose when its output is valid. For example, `StmReduce` only has valid output after reading its entire input.
- It has some internal state, like the counter in `StmCount`.
- It has zero or more input streams and, at any given time, it can choose whether to read a given input. For example, `StmConcat` only reads its second input after exhausting its first input.

Stream Extractor. To read the current data from a producer, the consumer must use the `sdata` primitive. `sdata` can only be used within a stream and its value is undefined if the corresponding `ready` expression evaluates to `false`. *All* producers for which the `ready` expression evaluates to `true` must have valid data before the consumer will proceed.

Stream Sharing. A stream e_1 can be shared by many consumers using the construct $\texttt{letstm}[n]\ x = e_1\ \texttt{in}\ e_2$. x can appear multiple times in e_2. `letstm` has a FIFO (First-In First-Out) queue whose capacity is at least n elements, which allows consumers to read the shared stream at different time steps. However, no consumer can read element $i + n$ before all consumers have read element i. For example, suppose s is a stream of length $n \geq 2$. Then $\texttt{letstm}[1]\ x = s\ \texttt{in}\ \texttt{StmConcat}(x, x)$ will get stuck, but $\texttt{letstm}[n]\ x = s\ \texttt{in}\ \texttt{StmConcat}(x, x)$ is valid.

Example. Figure 7 shows how the dot product program from figure 4 is expressed in the core language. The three distinct high-level operators `StmReduce`, `StmMap` and `StmZip` are all represented by `sbuild` and `sdata`. Figure 7 also shows its behaviour over time.

Type Checking. Type checking is straightforward; most rules are standard. Figure 6 shows typing rules for `vbuild`, `sbuild`, and `letstm`.

There are restrictions on the contents of collections *in the core language*. Collections can only contain "data types" D as defined in figure 5. This means streams cannot be nested, which greatly simplifies hardware generation. Section 5 will show that expressions with nested streams can be lowered to expressions in the core language that satisfy this constraint.

$\boxed{\Gamma \vdash e : T}$ in context Γ, e has type T

$\boxed{\Gamma|\Gamma' \vdash a : D}$ in external context Γ and internal context Γ', a has type D

$\boxed{\Gamma|\Gamma' \vdash p : T}$ in external context Γ and internal context Γ', p has type T

$$\frac{\Gamma' = \Gamma,\ a_i.x : D_i,\ p_i.x : T_i \quad \forall i.\Gamma|\Gamma' \vdash a_i : D_i \quad \forall i.\Gamma|\Gamma' \vdash p_i : T_i \quad \Gamma' \vdash d : D \quad \Gamma' \vdash v : \texttt{bool}}{\Gamma \vdash \texttt{sbuild}(n)(d, v)\ \{\ \vec{a}\ \}\ \{\ \vec{p}\ \} : \texttt{Stm}[D, n]}$$

$$\frac{\Gamma \vdash f : \mathsf{u}_k \to D \quad 2^k - 1 \geq n - 1}{\Gamma \vdash \texttt{vbuild}(n, f) : \texttt{Vec}[D, n]}$$

$$\frac{a.T = D \quad \Gamma \vdash a.\texttt{init} : D \quad \Gamma' \vdash a.\texttt{next} : D}{\Gamma|\Gamma' \vdash a : D}$$

$$\frac{p.T = T \quad \Gamma \vdash p.\texttt{stm} : T \quad \Gamma' \vdash p.\texttt{ready} : \texttt{bool}}{\Gamma|\Gamma' \vdash p : T}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Stm}[T_2, m] \quad \Gamma, x : \texttt{Stm}[T_2, m] \vdash e_2 : \texttt{Stm}[T_1, n]}{\Gamma \vdash \texttt{letstm}[k]\ x = e_1\ \texttt{in}\ e_2 : \texttt{Stm}[T_1, n]}$$

Figure 6. Typing judgments and rules for `sbuild`, `vbuild`, and `letstm`. Γ maps variables to types. For `sbuild`, there is an "internal" context Γ' which includes the variables defined in the `sbuild` and an "external" context Γ which does not.

```
1  (I0:Stm[u16,4]) => (I1:Stm[u16,4]) =>
2    sbuild(1)(sum+sdata(p),i+1==4) {/*StmReduce*/
3      (i:u16)   = { init: 0, next: i + 1 },
4      (sum:u16) = { init: 0, next: sum+sdata(p) }
5    }{(p: Stm[u16, 4]) = {
6        stm: sbuild(4)(                    /*StmMap*/
7          sdata(p).0 * sdata(p).1, true
8        ) {} {
9          (p: Stm[(u16, u16), 4]) = {
10           stm: sbuild(4)(                /*StmZip*/
11             (sdata(p0), sdata(p1)), true) {} {
12             (p0: Stm[u16, 4]) = {
13               stm: I0, ready: true },
14             (p1: Stm[u16, 4]) = {
15               stm: I1, ready: true }},
16           ready: true } },
17       ready: true } }
```

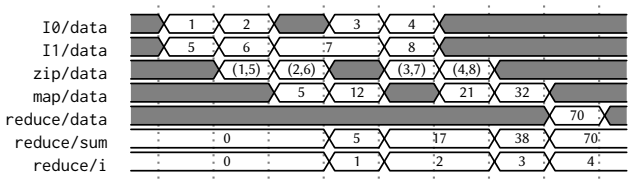

Figure 7. The dot product of two streams, expressed in the core language. The outermost `sbuild` (`StmReduce`) adds up the products using accumulator `sum`. It uses counter `i` to know when to stop. In the timing diagram, grey sections show where `valid` is `false`. `reduce/data` is the final dot product. `reduce/sum` and `reduce/i` are the accumulators in `StmReduce`. When the `StmMap` does not have valid data (because `I0` stalled earlier), the accumulators are not updated.

Summary. The core language of Sirop includes only five main primitives for vectors and streams. There is a constructor and extractor for vectors, `vbuild` and $v[i]$, borrowed from prior work. There is also a constructor and extractor for streams, `sbuild` and `sdata`. Finally, `letstm` allows a producer to have multiple consumers. The three stream-related primitives are one of the main contributions of this work.

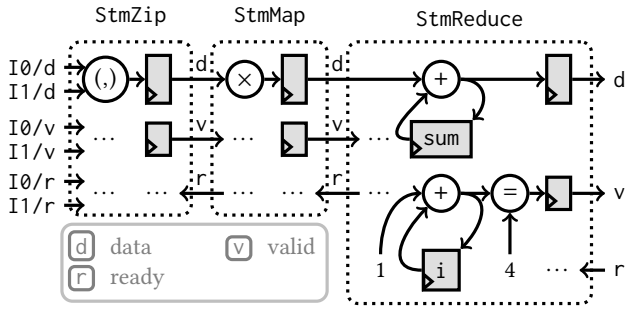

Figure 8. Hardware for the dot product example from figure 7. Handshake logic and register initial values are omitted.

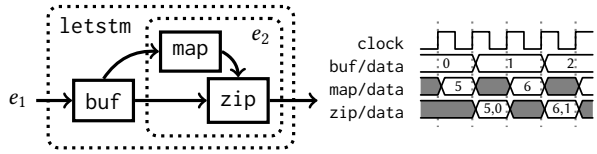

Figure 9. Block and timing diagrams for `letstm`$[1]$ $x = e_1$ `in` `StmZip`(`StmMap`$(x, +5), x$), where e_1 is the stream 0, 1, 2, ... Each element of e_1 will be read at cycle t by `map` and then at cycle $t + 1$ by `zip`.

4 Hardware Generation

An expression in the core IR can be straightforwardly translated to a streaming accelerator expressed in an HDL. The Sirop VHDL generator makes no optimization decisions; instead, all optimization decisions are made on the core IR. This will be discussed in section 6.

Combinational Logic. Expressions with no streams correspond to combinational logic. Tuples and vectors are represented by VHDL records and arrays, respectively.

Streams. Streams are translated to sequential logic. Each `sbuild` is mapped to one VHDL component. For example, the dot product program from figure 7 would be translated to a three-stage pipeline as shown in figure 8. Each accumulator

$$\mathcal{L}[\![\textbf{StmCount}(n : T)]\!] = \texttt{sbuild}(n)(i, \texttt{true})\ \{\ (i : T) = \{\ \texttt{init} : 0,\ \texttt{next} : 1 + i\ \}\ \}\ \{\ \}$$

$$\mathcal{L}[\![\textbf{StmReduce}(s : \texttt{Stm}[D, n],\ f)]\!] = \texttt{sbuild}(1)(\texttt{if (first) then sdata}(p)\ \texttt{else}\ f(\texttt{acc}, \texttt{sdata}(p)), 1 + i == n)$$
$$\{(\texttt{acc} : D) = \{\ \texttt{init} : \_,\ \texttt{next} : \texttt{if (first) then sdata}(p)\ \texttt{else}\ f(\texttt{acc}, \texttt{sdata}(p))\ \},$$
$$(\texttt{first} : \texttt{bool}) = \{\ \texttt{init} : \texttt{true},\ \texttt{next} : \texttt{false}\ \},\ (i : \texttt{u}_{\lceil \log_2(n+1) \rceil}) = \{\ \texttt{init} : 0,\ \texttt{next} : 1 + i\ \}\ \}$$
$$\{\ (p : \texttt{Stm}[D, n]) = \{\ \texttt{stm} : s,\ \texttt{ready} : \texttt{true}\ \}\ \} \quad \textit{where } n \geq 1$$

$$\mathcal{L}[\![\textbf{StmConcat}(s_0 : \texttt{Stm}[D, n], s_1 : \texttt{Stm}[D, m])]\!] = \texttt{sbuild}(n + m)(\texttt{if}\ i == n\ \texttt{then sdata}(p_1)\ \texttt{else sdata}(p_0), \texttt{true})$$
$$\{\ (i : \texttt{u}_{\lceil \log_2(n+1) \rceil}) = \{\ \texttt{init} : 0,\ \texttt{next} : \texttt{if}\ i == n\ \texttt{then}\ i\ \texttt{else}\ 1 + i\ \}\ \}$$
$$\{\ (p_0 : \texttt{Stm}[D, n]) = \{\ \texttt{stm} : s_0,\ \texttt{ready} : i \neq n\ \},\ (p_1 : \texttt{Stm}[D, m]) = \{\ \texttt{stm} : s_1,\ \texttt{ready} : i == n\ \}\ \}$$

$$\mathcal{L}[\![\textbf{StmZip}(s_0 : \texttt{Stm}[D_0, n],\ s_1 : \texttt{Stm}[D_1, n])]\!] = \texttt{sbuild}(n)(\ (\texttt{sdata}(p_0),\ \texttt{sdata}(p_1)),\ \texttt{true}\ )\ \{\ \}\ \{$$
$$(p_0 : \texttt{Stm}[D_0, n]) = \{\ \texttt{stm} : s_0,\ \texttt{ready} : \texttt{true}\ \},\ (p_1 : \texttt{Stm}[D_1, n]) = \{\ \texttt{stm} : s_1,\ \texttt{ready} : \texttt{true}\ \}\ \}$$

$$\mathcal{L}[\![\textbf{StmMap}(s, (x : T) \mapsto e)]\!] = [s/x]\ \mathcal{R}[\![\mathcal{S}[\![e]\!]]\!] \qquad \mathcal{L}[\![\textbf{StmSplit}(s,\ m)]\!] = s \qquad \mathcal{L}[\![\textbf{StmJoin}(s)]\!] = s$$

Figure 10. Rules for lowering high-level stream operations into the core Sirop language. $\mathcal{L}[\![\cdot]\!]$ is the lowering transformation. $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]$ are the streamification and resetting transformations, respectively. $[e_1/x]e_2$ denotes substitution.

is represented by a register with the given initial value. The output (`data` and `valid`) is also stored in a register to avoid long combinational paths in the pipeline. Finally, for each input producer p, the `ready` signal is set using p.`ready`.

Algorithm 1 shows the pseudocode for converting an `sbuild` to VHDL. Notably, lines 2–7 describe the logic for making an `sbuild` wait for its consumer or producers.

Stream Sharing. `letstm` also translates to a sequential VHDL component. The queue is implemented as a circular buffer with one pointer per consumer. The VHDL entity has parameters for the queue capacity, number of consumers, etc. The Sirop VHDL generator simply instantiates this entity with appropriate parameters and connects its ports to the producer and consumers. Figure 9 is a block diagram of the hardware generated from a simple program with `letstm`.

Other Targets. Although this paper focuses on VHDL, Sirop is not limited to hardware. For example, one could compile Sirop to software where each `sbuild` is mapped to a thread and the handshake interfaces use queues and semaphores. The implementation is left for future work.

5 Lowering

Sirop can express all the functional parallel patterns that are typically used for hardware design: `StmMap`, `StmReduce`, `VecMap`, `VecReduce`, etc. The lowering transformation $\mathcal{L}[\![\cdot]\!]$ converts them all to the core language.

`vbuild` appears in prior works [37, 42]. For example:

$$\mathcal{L}[\![\texttt{VecMap}(v : \texttt{Vec}[T, n], f)]\!] = \texttt{vbuild}(n, i \mapsto f(v[i]))$$

Figure 10 shows a few examples of stream-related lowering rules. These are applied in a bottom-up manner, so producers are lowered before their consumers.

As mentioned in section 3, to simplify hardware generation, the core language does not support nested streams.

Algorithm 1 `sbuild` hardware generation

```
1  def emit(s, consumer):
2    # Handshake logic
3    send = s.valid_sig and consumer.ready_sig[s]
4    arpv = all( # all required producers valid
5      not s.ready_expr[p] or p.valid_sig
6      for p in s.producers)
7    can_step = arpv and (send or not s.valid_sig)
8    # Output registers
9    s.data_sig = create_register(name="data",
10     init=None, update=s.data,
11     update_if=can_step or send )
12   s.valid_sig = create_register(name="valid",
13     init=False, update=s.valid and arpv,
14     update_if=can_step or send )
15   # Accumulators
16   for a in s.accumulators:
17     create_register(name=a.name, init=a.init,
18       update=a.next, update_if=can_step )
19   # Producers
20   for p in s.producers:
21     s.ready_sig[p] =
22       s.ready_expr[p] and can_step
23     emit(p.stm, s)
```

Nevertheless, nested streams can be used in the higher-level language. Operations on nested streams are removed during lowering by flattening. Like the rest of the lowering pass, this is fully automatic; no manual refactoring is required.

Since the lowering pass flattens streams, `StmSplit` and `StmJoin` are trivial (figure 10). A useful consequence is that expressions like `StmSplit` $\circ f \circ$ `StmJoin` will always lower to just f. If `StmSplit` and `StmJoin` were primitives, the optimizer would need yet another rewrite rule for this.

```
1  // /*RESET*/ stands for:
2  //   i != 2 && in_cnt == 2 && out_cnt == 1
3  //     || in_cnt == 2 && out_cnt == 0 && i == 2
4  sbuild(10)(
5    if first then sdata(p) else sum + sdata(p),
6    i == 2
7  ) {
8    (sum: u16) = {
9      init: 0,
10     next: if /*RESET*/ then 0
11           else if (first) then sdata(p)
12           else sum + sdata(p) },
13   (first: bool) = {
14     init: true,
15     next: /*RESET*/ },
16   (i: u4) = {
17     init: 0,
18     next: if /*RESET*/ then 0 else 1 + i },
19   (in_cnt: u32) = {
20     init: 0,
21     next: if /*RESET*/ then 0 else 1+in_cnt },
22   (out_cnt: u32) = {
23     init: 0,
24     next: if /*RESET*/     then 0
25           else if (i == 2) then 1 + out_cnt
26           else                  out_cnt }
27 }{(p: Stm[u16, 30]) = { stm: s, ready: true } }
```

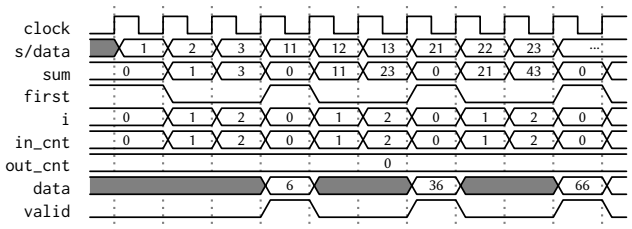

Figure 11. The lowered form of expression 1 along with a timing diagram that illustrates its behaviour with input $[[1,2,3],[11,12,13],[21,22,23],\ldots]$.

For an example of lowering `StmMap`, consider the following expression, which sums the rows of a 2D stream:

$$s.\texttt{StmMap}((\texttt{row} : \texttt{Stm}[\texttt{u}_8, 3]) \mapsto \texttt{row.StmReduce}(+)) \quad (1)$$

The lowered expression for $\texttt{StmReduce}(\texttt{row}, +)$ has already been shown in figures 7 and 10. To implement `StmMap`, all we need is to reset the accumulators in `StmReduce` at the end of each row. Figure 11 shows the lowered row sum expression. In general, lowering `StmMap` works by a "stream resetting" transformation $\mathcal{R}[[\cdot]]$. For each `sbuild` in the body of f:

1. Add a new accumulator to count valid outputs (e.g., `out_cnt` on lines 22–26 in figure 11).
2. Add accumulators to count inputs from each producer (e.g., `in_cnt` on lines 19–21 in figure 11).
3. Reset all accumulators, even the new counters, once the expected number of inputs have been read and the expected number of valid outputs have been produced.
4. Update the length of the stream.

```
1  sbuild(10)(
2    if first then sdata(p) else sum + sdata(s),
3    i == 2
4  ) {
5    (sum: u16) = {
6      init: 0,
7      next: if (i == 2) then 0
8            else if (first) then sdata(s)
9            else sum + sdata(s) },
10   (first: bool) = { init: true, next: i == 2 },
11   (i: u4) = {
12     init: 0,
13     next: if (i == 2) then 0 else 1 + i }
14 }{(p: Stm[u16, 30]) = { stm: s, ready: true } }
```

Figure 12. Figure 11 after simplification (section 6.2). Notice that both `in_cnt` and `out_cnt` have been removed.

The technique described above works for $\texttt{StmMap}(s, f)$ if $f : \texttt{Stm}[D_1, n_1] \to \texttt{Stm}[D_2, n_2]$. It can be generalized to functions of type $D_1 \to D_2$ and $D_1 \to \texttt{Stm}[D_2, n]$ by the "streamification" transformation $\mathcal{S}[[\cdot]]$, which transforms the function so its input and output are both streams. For example, if $f : D_1 \to D_2$, the streamified function $\mathcal{S}[[f]]$ is

$$(x : \texttt{Stm}[D_1, 1]) \mapsto \texttt{sbuild}(1)(f(\texttt{sdata}(p)), \texttt{true})\ \{\ \}\ \{\ (p : \texttt{Stm}[D_1, 1]) = \{\ \texttt{stm} : x,\ \texttt{ready} : \texttt{true}\ \}\ \}$$

Functions of type $\texttt{Stm}[D_1, n] \to D_2$ are not allowed. In hardware terms, a function that has a sequential input must produce a sequential output. Functions like `StmReduce` return a stream of length one to preserve the notion of time.

6 Optimizations

This section outlines optimizations in the Sirop compiler that improve resource usage or maximum clock frequency.

6.1 Partial Evaluation

Partial evaluation performs computations at compile time to improve runtime performance. The Sirop optimizer performs simplifications such as inlining calls to function literals and applying arithmetic identities (e.g., $e + 0 \rightsquigarrow e$).

A particularly interesting example is the fusion rule for `vbuild`, which has been used in prior works [37, 42] and is similar to shortcut deforestation [14]:

$$\texttt{vbuild}(n, f)[i] \rightsquigarrow f(i)$$

Familiar cases like map/map fusion follow from this rule.

$$\begin{aligned}
&\mathcal{L}[[\texttt{VecMap}(\texttt{VecMap}(v, f), g)]] \\
&= \texttt{vbuild}(n, i \mapsto g(\texttt{vbuild}(n, j \mapsto f(v[j]))[i])) \\
&= \texttt{vbuild}(n, i \mapsto g(f(v[i]))) \\
&= \mathcal{L}[[\texttt{VecMap}(v, g \circ f)]]
\end{aligned}$$

$$\begin{aligned}
&\texttt{Fuse}_x[\![\ \texttt{sbuild}(n_c)(d_c,\ v_c)\ \{a_1,\ \ldots,\ a_\ell\}\ \{\ p_1,\ \ldots,\ p_m,\ (x : \texttt{Stm}[T, n_p]) = \{\\
&\qquad\qquad \texttt{stm} : \texttt{sbuild}(n_p)(d_p,\ v_p)\ \{\ a_{\ell+1},\ \ldots,\ a_{\ell+j}\ \}\ \{\ p_{m+1},\ \ldots,\ p_{m+k}\ \},\ \texttt{ready} : r_c\ \}\ \}\ ]\!]\\
&= \texttt{sbuild}(n_c)(d_f,\ v_f)\ \{\ a_{f,1},\ \ldots,\ a_{f,\ell+j}\ \}\ \{\ p_{f,1},\ \ldots,\ p_{f,m+k}\ \}\\
&\textbf{where}\ d_f = [d_p/\texttt{sdata}(x)]\ d_c \qquad\qquad v_f = ([d_p/\texttt{sdata}(x)]\ v_c) \wedge (r_c \implies v_p)\\
&a_{f,i} = \begin{cases} \{\ \texttt{init} : a_i.\texttt{init},\ \texttt{next} : \texttt{if}\ (r_c \implies v_p)\ \texttt{then}\ [d_p/\texttt{sdata}(x)]\ a_i.\texttt{next}\ \texttt{else}\ a_i.x\ \} & i \le \ell \\ \{\ \texttt{init} : a_i.\texttt{init},\ \texttt{next} : \texttt{if}\ (v_p \implies r_c)\ \texttt{then}\ a_i.\texttt{next}\ \texttt{else}\ a_i.x\ \} & \text{else} \end{cases}\\
&p_{f,i} = \begin{cases} \{\ \texttt{stm} : p_i.\texttt{stm},\ \texttt{ready} : \texttt{if}\ (r_c \implies v_p)\ \texttt{then}\ [d_p/\texttt{sdata}(x)]\ p_i.\texttt{ready}\ \texttt{else}\ \texttt{false}\ \} & i \le m \\ \{\ \texttt{stm} : p_i.\texttt{stm},\ \texttt{ready} : \texttt{if}\ (v_p \implies r_c)\ \texttt{then}\ p_i.\texttt{ready}\ \texttt{else}\ \texttt{false}\ \} & \text{else} \end{cases}
\end{aligned}$$

Figure 13. The fusion rule for streams. Occurrences of $\texttt{sdata}(x)$ must be replaced by the producer data d_p. Furthermore, the `next` and `ready` expressions must be updated so that the consumer waits for the producer when needed and vice-versa.

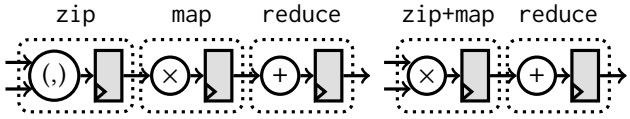

Figure 14. Dot product hardware before and after fusion.

$$\begin{aligned}
&\texttt{Fission}[\![\ \texttt{sbuild}(n)(f(d),\ v)\ \{\ \vec{a}\ \}\ \{\ \vec{p}\ \}\ ]\!]\\
&= \texttt{sbuild}(n)(f(\texttt{sdata}(x)),\ \texttt{true})\ \{\}\ \{\ (x : \texttt{Stm}[D, n]) = \{\\
&\qquad \texttt{stm} : \texttt{sbuild}(n)(d,\ v)\ \{\ \vec{a}\ \}\ \{\ \vec{p}\ \},\ \texttt{ready} : \texttt{true}\ \}\ \}
\end{aligned}$$

Figure 15. The fission rule for streams. f must not have any free variables.

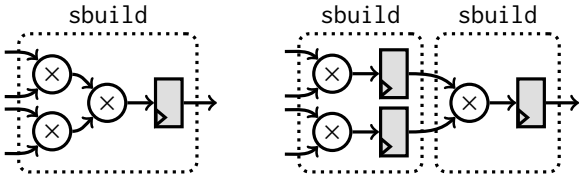

Figure 16. Parallel reduction before and after fission.

In software, fusion is a well-known technique for eliminating the overhead of intermediate data structures [9, 14, 40, 45]. Vector fusion has similar benefits.

6.2 Basic `sbuild` and `letstm` Simplifications

There are several obvious ways to simplify the accumulators in `sbuild`. Unused accumulators are deleted; if an accumulator's value never changes then that value is inlined; and duplicate accumulators are merged. For example, in the timing diagram of figure 11, it is clear that `i` and `in_cnt` always have the same value and that `out_cnt` is always zero. Figure 12 shows the simplified expression.

For $\texttt{letstm}[n]\ x = e_1\ \texttt{in}\ e_2$, the compiler can inline e_1 (i.e., rewrite to $[e_1/x]\ e_2$) if x appears at most once in e_2. This completely eliminates the resource usage due to `letstm`.

6.3 Stream Fusion and Fission

As with vectors, it is sometimes helpful to fuse stream operators. This eliminates the overhead of the handshake interface. For instance, in the dot product example (figure 8), the optimizer will fuse the zip and map stages, as in figure 14.

Figure 13 shows the stream fusion rule, which essentially reproduces the handshake protocol within one `sbuild`. Only one rule is needed. By lowering an expression and then applying the one rule to rule them all, any combination of streaming operators can be fused: `StmMap` with `StmMap`, `StmMap` with `StmReduce`, etc. If each high-level operator was atomic, a rule would be needed for each combination. Furthermore, in prior works, the set of operators is not closed under fusion—some fusion rules are not expressible because the result is not in the list of built-in operators. For instance, fusing `StmZip` with `StmZip` would yield an operator with three inputs. No such operator exists in Shir or Aetherling.

To avoid generating long combinational paths, the optimizer uses a cost model for combinational delay. Fusion is only applied if the cost model estimates that the timing requirements will still be met. For example, if $C[\![e]\!]$ is the cost of expression e, then

$$\begin{aligned}
C[\![e_1 + e_2]\!] &= 33 + \max(C[\![e_1]\!], C[\![e_2]\!])\\
C[\![e_1 * e_2]\!] &= 100 + \max C[\![e_1]\!], C[\![e_2]\!]
\end{aligned}$$

The maximum allowable delay is (somewhat arbitrarily) set at 100. Thus, up to three additions can be performed per cycle but multiplication cannot be chained with any other operations. This is highly simplistic; cost modelling is not the focus of this paper. Given a more sophisticated model, it would be straightforward to integrate it into the compiler.

Sirop also supports the opposite transformation: splitting one `sbuild` into two. For example, in figure 16, the optimizer splits up the `sbuild` to avoid performing two multiplications in one cycle. Fission is guided by the same cost model as fusion. As with fusion, only one rule is needed; see figure 15.

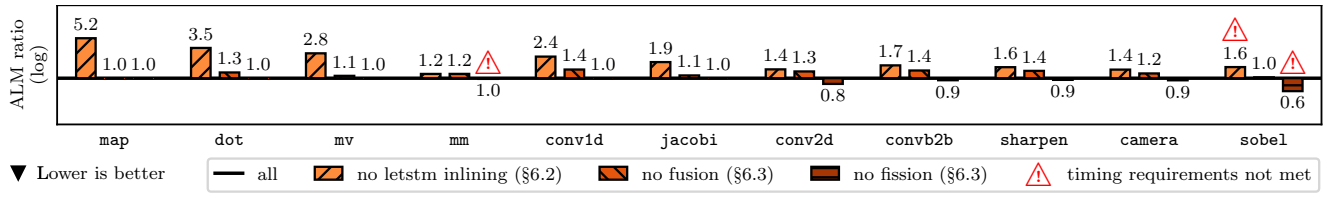

Figure 17. ALM usage ratio (`other` / `baseline`) when *turning off* various optimization passes.

7 Evaluation

This section evaluates Sirop's hardware generation and optimization capabilities. First, the effectiveness of the optimizations described in section 6 is demonstrated. Next, Sirop is compared with Aetherling [11], Shir [36], and the Intel HLS compiler to show that Sirop is expressive and can be used to generate fast and resource-efficient hardware.

7.1 Methodology and Benchmarks

Sirop is evaluated on a superset of the benchmarks from Aetherling [11] and Shir [36], as shown in table 1. Designs are synthesized using Quartus Prime 19.2 Pro Edition for an Arria 10 GX FPGA. The target clock frequency is 175 MHz, like in the Aetherling paper. Resource usage is measured and timing closure is verified post-place and route. Quartus reports the numbers of ALMs (Adaptive Logic Modules), which include lookup tables and registers; BRAMs (Block RAMs), which are on-chip memory units; and DSPs (Digital Signal Processing units), which are used for multiplication.

For each program, a testbench checks correctness and reports the latency, in clock cycles, from the first input to the last output. To shorten simulation time, testbenches for large programs use inputs with fewer rows (e.g., `conv2d` is tested with a 16×1920 input rather than 1080×1920).

Generating VHDL via Sirop for all 124 programs in sections 7.2, 7.3, and 7.4 takes about eight minutes on a laptop with an Intel Core i7 CPU and 16 GB of RAM running Ubuntu 22.04. The slowest program takes less than 30 seconds.

Table 1. Benchmarks used to evaluate Sirop.

Name	Input Streams	Description
`map`	$200 \times \mathtt{u}_8$	Add five to each element
`dot`	$840 \times \mathtt{u}_{16}$ (both)	Dot product
`mv`	$256 \times 256 \times \mathtt{u}_{16}$ $256 \times \mathtt{u}_{16}$	Matrix-vector product
`mm`	$256 \times 256 \times \mathtt{u}_{16}$ $256 \times 256 \times \mathtt{u}_{16}$	Matrix-matrix product
`conv1d`	$16 \times \mathtt{i}_8$	1-D convolution
`jacobi`	$1024 \times 128 \times \mathtt{u}_8$	Four-point stencil $\frac{1}{4}(x_{i-1,j} + x_{i,j-1} + x_{i,j+1} + x_{i+1,j})$
`conv2d`	$1080 \times 1920 \times \mathtt{u}_{32}$	3×3 convolution
`convb2b`	$1080 \times 1920 \times \mathtt{u}_{32}$	3×3, then 2×2 conv.
`sharpen`	$1080 \times 1920 \times \mathtt{u}_{32}$	Image sharpening
`camera`	$1080 \times 1920 \times \mathtt{u}_{32}$	Demosaic and sharpen
`sobel`	$1080 \times 1920 \times \mathtt{u}_{32}$	Sobel-Feldman operator

7.2 Effects of Optimizations

Setup. Figure 17 shows the effects of the optimizations from section 6 on ALM usage and clock frequency. BRAM usage follows similar trends to ALM usage. DSP usage and latency do not change much because the Sirop compiler does not change the degree of spatial parallelism. The degree of parallelism can instead be chosen while lowering from the "algorithmic level" (e.g., `map`, `reduce`) to the "architectural level" (e.g., `StmMap`, `VecReduce`), as will be discussed in section 7.4.

Results. *Disabling* `letstm` inlining and fusion tends to increase ALM usage—i.e., when enabled, these passes save resources. `letstm` inlining is particularly effective because most variables are used only once. `sbuild` accumulator simplification has little effect here, but disabling it does sometimes hinder other optimizations in the experiment in section 7.4. Disabling partial evaluation often yields designs so poor the simulator crashes, so it is not included in the figure.

Disabling stream fission sometimes decreases resource usage, especially for the `sobel` benchmark. However, this may also lead to failure to meet the timing requirements.

In short, `letstm` inlining and fusion reduce resource usage while fission is important for achieving timing closure.

7.3 Comparison with Prior Works

Setup. Figure 18 compares Sirop with prior works: the Intel HLS compiler, Shir, and Aetherling. For Intel HLS, each benchmark is implemented in C++ using the `ihc::pipe` streaming interface. The `#pragma unroll` directive is added wherever spatial parallelism is desired. The shortest program, `map`, takes 5 lines of C++. The longest, `camera`, is about 140 lines. In Aetherling, each benchmark is written in the higher-level language L^{seq} using operators like `map` and `reduce`. Whenever possible, benchmark code written by the Aetherling authors themselves is used. In Shir, each benchmark is implemented at the "architectural level" using operators like `StmMap` and `StmReduce`. The Sirop programs use the same operators as the Shir programs. For illustration, appendix A shows the source code for the `mv` benchmark.

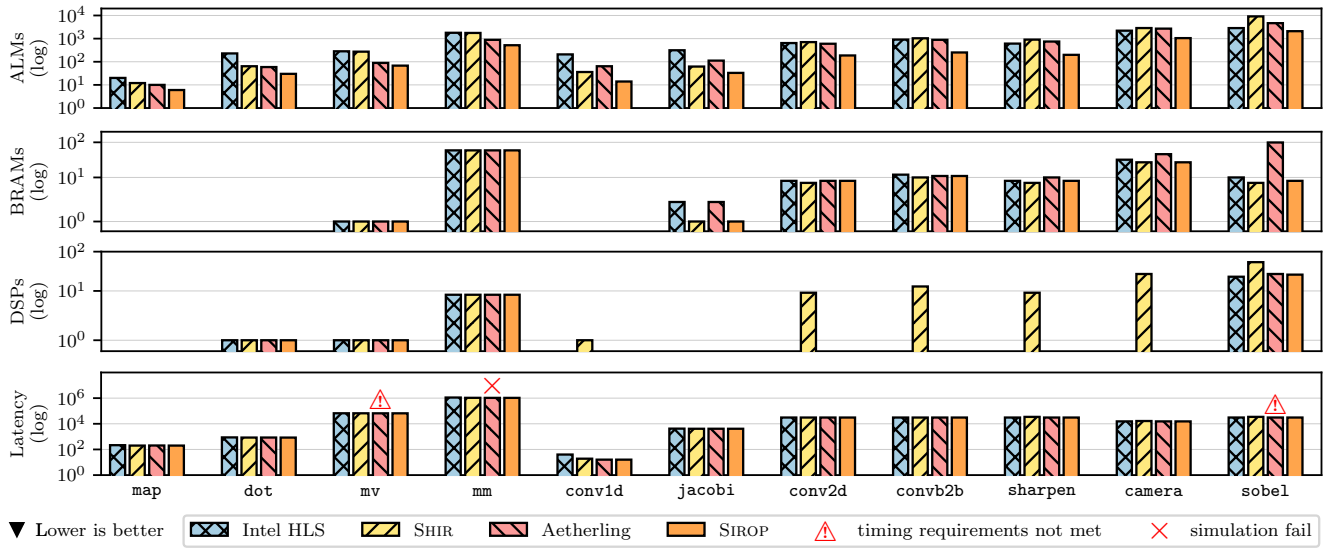

Figure 18. Resource usage and latency of designs generated by Sirop compared to prior works.

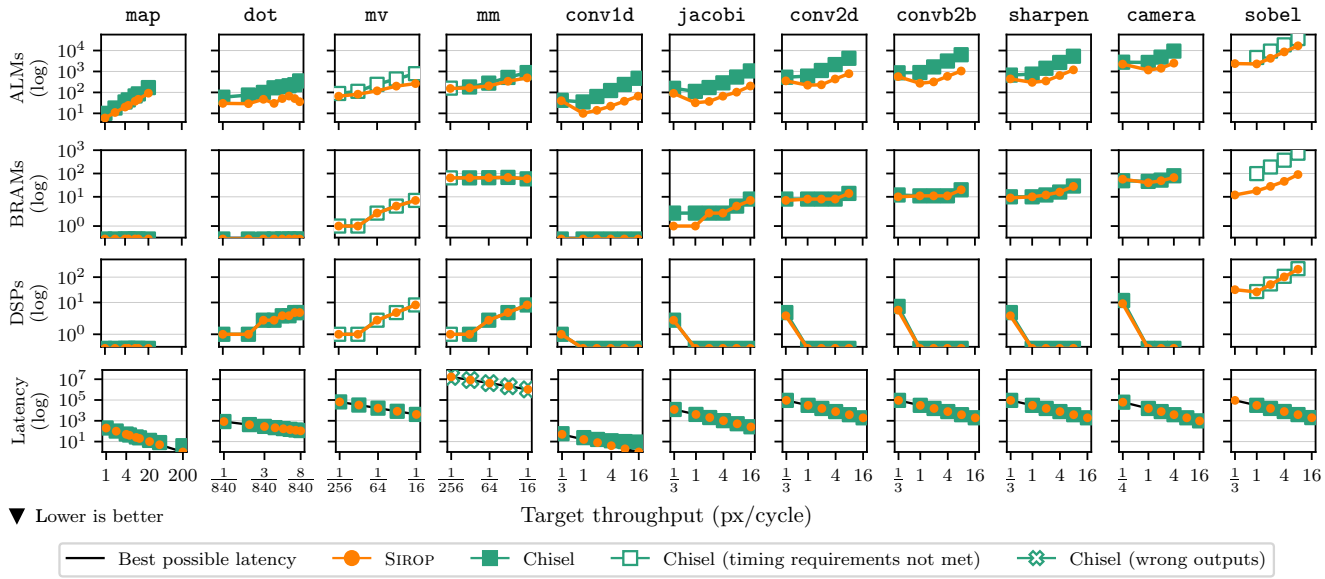

Figure 19. Resource usage and latency of designs generated using Sirop compared to Aetherling's Chisel backend. The x-axis is the target throughput given to the Aetherling compiler, which determines the degree of spatial parallelism. The "best possible latency" is max(`in_len`, `out_len`) / `throughput`. The plots are log-log because the target throughputs generally increase exponentially. An empty shape means the design does do not meet the timing requirements or requires too many ports.

Results. Sirop consistently generates designs with fewer ALMs than all prior works: on average, 76% fewer than Intel HLS, 68% fewer than Shir, and 61% fewer than Aetherling. BRAM and DSP usage is usually similar across tools. No DSPs are needed for `conv1d`, `jacobi`, `conv2d`, `convb2b`, `sharpen`, and `camera` because the kernels only multiply by powers of two and can therefore use bitwise shifts.

One factor in Sirop's better ALM efficiency is its ability to fuse any two stream operators, as discussed in section 7.2. Another reason is that, in Sirop, it is straightforward to omit or remove unnecessary accumulators. In the `map` benchmark, for instance, Shir uses a hardware template that is general enough to handle nested streams. The general template has two counters. In Sirop, by contrast, the final design does not have any counters. The stream resetting transformation (section 5) does add input and output counters; however, for a 1D stream, these counters will be unused and it is trivial for the optimizer to remove them (section 6.2).

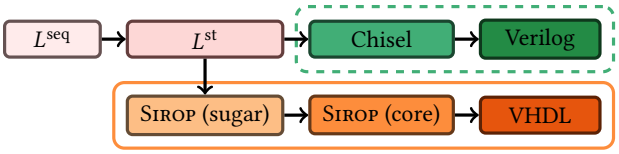

Figure 20. Chisel and Sirop compilation flows.

Notes. The Aetherling `mm` design produces incorrect results. One stream reaches the multiplier with a delay of one cycle, whereas the other has a delay of two cycles. Sirop uses dynamic scheduling, which rules out such issues.

For the `sharpen`, `sobel`, and `camera` benchmarks, Shir requires the programmer to manually insert registers (using a `Registered` primitive in the IR). Without manual intervention, latency increases by a factor of 4, 4.8, and 2.8, respectively, and the `sobel` design does not meet the timing requirements. Some Aetherling designs also do not meet the timing requirements. Intel HLS and Sirop achieve timing closure and reach the expected throughput for every benchmark without manual intervention.

7.4 Design Space Exploration with Aetherling

Setup. Sirop is also evaluated by using it as a backend to Aetherling, as shown in figure 20. Benchmarks are written in Aetherling's high-level language, L^{seq}. Aetherling lowers these programs to its "space-time" language with streams and vectors, L^{st}. The lowering process aims to minimize resource usage while achieving a given steady-state throughput. This step determines the degree of spatial parallelism. The resulting L^{st} programs are parsed and compiled using Sirop. Aetherling's existing backend emits Chisel [3], which is then translated to Verilog. With the Chisel backend, Aetherling has been shown to produce designs with lower resource utilization than Spatial [23], a high-level HDL, and Halide-HLS [33], an extension of the Halide image processing language.

Results. Figure 19 shows the resource usage for different target throughputs. As expected, higher throughputs require more spatial parallelism and thus more resources. Like in section 7.3, the designs produced by Sirop always use fewer ALMs than those produced by Aetherling's Chisel backend. Figure 19 also shows that, for both Chisel and Sirop, the measured latency is close to the ideal latency for the given target throughput. This shows that Sirop can accurately express all the same design points as Aetherling.

Note. Some designs require more ports than the target device provides. For example, moving 16 values of type u_{32} in and out per cycle requires at least $2 \cdot 16 \cdot 32$ ports, which is too many; thus, `conv2d` cannot be synthesized at a throughput of 16 with the bitwidth set by the Aetherling authors. This is the case for both Sirop and Aetherling.

Table 2. Lines of code for the hardware generation step.

	Scala	Haskell	VHDL	Verilog
Aetherling [11] + Chisel [3]	1,411 + 8,577	771	–	168
Shir [36]	3,931	–	8,009	–
Sirop	2,657	–	289	–

Due to a minor bug in Aetherling's L^{st} to Chisel conversion, no Verilog is generated for `sobel` with throughput 1/3. Furthermore, as mentioned in section 7.3, Aetherling's `mm` designs produce wrong outputs.

7.5 Engineering Effort

We now show how Sirop simplifies compiler development.

Lines of Code. Table 2 shows the size of each functional HLS compiler's hardware generator. The Sirop VHDL generator is mostly written in Scala, with some VHDL for the `letstm` hardware template. It uses no external dependencies except for logging and file I/O. Hardware generation in Aetherling itself is more concise, but this is because it targets Chisel, a higher-level HDL. The Chisel project is far larger than Sirop's hardware generator. Shir has over 100 hardware templates for various operators, which add up to over 8,000 lines of VHDL.

Extensibility. It is difficult to express a 2D convolution in Shir without using host memory. To obtain the results in section 7.3, a new streaming operator, `StmSlide2D`, is added to both Shir and Sirop. In Shir, this requires providing a VHDL implementation for the operator. In total, the change took 95 lines of VHDL and 109 lines of Scala code. In Sirop, the operator is instead lowered to the core language, which took only 88 lines of Scala code. The Sirop optimizer and hardware generator can handle the new operator with no changes. The Shir optimizer, by contrast, would need new rewrite rules to be able to transform the new operator.

In addition to Sirop being a higher-level target, the compiler includes features that simplify testing and debugging. For example, it has a simple interpreter that can be used to test the correctness of the new operator directly in Scala and inspect the value of each `sbuild` accumulator at each time step. In Shir, one must write and run a VHDL testbench.

Summary. The simplicity of the core language enables more concise hardware generation than prior works. Furthermore, it is easy to extend the higher-level language with new syntax sugar. All this comes without sacrificing hardware quality. In short, this new, minimal set of streaming primitives is a strong foundation for functional HLS.

8 Related Work

Functional HLS. Shir [36] and LiftHLS [24] compile functional parallel patterns to dynamically-scheduled streaming accelerators. A key feature of Shir is its ability to represent and optimize host memory accesses. There has also been recent work exploring resource sharing in Shir for large designs [21]. Coarse-grained resource sharing in Sirop is left for future work, as are host memory accesses and memory-related transformations such as banking. We expect many of the techniques used in Shir to be transferable to Sirop.

Aetherling [11] compiles similar functional programs to streaming accelerators with static schedules. All Aetherling operators have fixed latency and communicate at fixed rates. This enables static scheduling with no overhead—not even a control unit—but makes it impossible to support variable-latency operations. For example, consider a program that decides whether a stream contains a given element. In Sirop, the design can produce an affirmative result immediately if the element is found, even if it occurs near the beginning of the stream. In Aetherling, by contrast, one would have no choice but to consume the entire stream before answering.

Shir, LiftHLS, and Aetherling all have many stream and vector operators, which complicates hardware generation and optimization. Sirop has a far more compact IR and generates more resource-efficient accelerators.

Imperative HLS. Many HLS tools use imperative languages, especially C and C++. This is convenient for C and C++ programmers, but imperative languages are a poor fit for hardware design. On one hand, many features (e.g., dynamic memory allocation) do not map well to hardware. On the other hand, it can be difficult to infer pipeline and spatial parallelism from loop-based code. C-based HLS frameworks such as the Intel HLS compiler, Xilinx HLS, and LegUp [5] rely on pragmas to guide synthesis. Other tools, such Halide-HLS [33], HeteroCL [26], Allo [6], and POM [48] use a separate scheduling language to clarify intent. In practice, the generated hardware tends to have unpredictable performance and source code with pragmas is non-portable [19, 29].

Some authors have tried addressing the portability and reliability issues. Dahlia [29] has a type system to enforce unwritten rules of HLS. AnyHLS [32] generates more portable C-HLS code starting from a yet higher-level language. One can also augment C-HLS with functional parallel skeletons [19]. However, forgoing imperative programming for HLS altogether might be more reliable long-term.

There has been work [1, 6, 13, 30, 46, 48, 49] on using MLIR [27] to optimize hardware designs at multiple levels of abstraction. Although the Sirop compiler is currently implemented as a standalone project, the Sirop IR could just as well be implemented as an MLIR dialect. Sirop is particularly well-suited to implementing the parallel patterns used in prior works [11, 19, 24, 25, 36] while being easy to translate to hardware.

Other Hardware Languages. Some languages have been designed to map well to hardware while providing a higher level of abstraction than traditional HDLs. Notable examples include Esterel [12], Bluespec [31], CλaSH [2], Spatial [23], Fleet [41], and Reticle [43]. There are also embedded HDLs and libraries (e.g., Chisel [3], Lava [4], PyRTL [8], MyHDL) that allow metaprogramming in a popular host language. All these are still best suited for users with hardware design experience. Languages like Shir aim to be even higher-level, letting users focus only on the algorithm. This paper has the same use case in mind, although it focuses on the IR rather than the highest-level language.

Domain-Specific Languages. There are many DSLs for HLS in machine learning [10, 38, 44, 47], image processing [7, 16, 17, 28, 33, 34, 39], etc. Sirop aims to be general enough to generate streaming accelerators in any domain.

Minimal Array Languages. `vbuild` is used in prior IRs for software, including for destination-passing style [37] and idiom recognition [42]. Other works on deforestation [9, 14, 40] use similar ideas. This paper applies the principle of using a minimal IR, but in a new context (HLS) and with a new IR.

Dataflow Programming. Sirop is similar to KPNs (Kahn Process Networks) [22] in that programs consist of many "computing stations" (`sbuild`) running in parallel. However, to minimize resource usage, each `sbuild` node communicates with its consumer directly, without a queue. Sending data must therefore be blocking. FIFO queues are only added for `letstm`, and these have a fixed capacity. Also, unlike in KPNs, `sbuild` can wait for *many* producers at once; this allows `StmZip` to be implemented without any memory.

Sirop, like Shir and Aetherling, currently only supports acyclic pipelines. This limitation could perhaps be lifted by adding a fourth stream primitive `letrec` $x = e_1$ `in` e_2, where e_1 can use x to access its own output.

9 Conclusion

This paper has demonstrated the benefits of using a minimal functional language for high-level synthesis. With fewer primitives, there are fewer hardware templates to implement and test. Furthermore, key optimizations like stream fusion and fission can be implemented once and for all. All the familiar functional primitives can be expressed as syntax sugar; this new language is *more* expressive than Aetherling and at least as expressive as the subset of Shir that does not use memory. Moreover, Sirop generates fast and resource-efficient hardware—there is no loss of hardware quality.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), grants RGPIN-2020-05889 and RGPIN-2026-06336, as well as the CIFAR AI Chairs Program.

Data Availability Statement

The artifact for this paper is available on Zenodo [18]. It includes Sirop, Shir, Aetherling, and Intel HLS source code for all benchmarks. One example is provided below for easy reference. The artifact also includes a parser that translates any L^{st} program to an equivalent high-level Sirop program. This shows that Sirop is at least as expressive as Aetherling.

A Benchmark Source Code

The following listings contain the source code for the `mv` benchmark, which performs matrix-vector multiplication.

Sirop. The Sirop implementation of matrix-vector multiplication is very similar to the dot product code in figure 4. Here, the matrix `mat` is a 2D stream and `StmMap` is used to compute the dot product row-by-row.

```
(mat : Stm[Stm[u16, 256], 256]) =>
(vec :     Stm[u16, 256]      ) =>
    mat.StmMap( (row: Stm[u16, 256]) =>
        /* dot(row, vec) */
        StmZip(row, vec)
        .StmMap( (x) => x.0 *% x.1)
        .StmReduce( (x) => x.0 +% x.1 ) )
```

Shir. The Shir source code looks superficially different from the Sirop source code because Shir is embedded in Scala rather than having a custom parser. However, the actual operators used are identical.

```
val u16 = IntType(16)
val mat = ParamDef(OrderedStreamType(
  OrderedStreamType(u16, 256), 256 ))
val vec = ParamDef(OrderedStreamType(u16, 256))
val row = ParamDef(OrderedStreamType(u16, 256))
val xy = ParamDef(TupleType(u16, u16))
val acc = ParamDef()
val x = ParamDef()
ArchLambdas(Seq(mat, vec),
  MapOrderedStream(
    ArchLambda(row,
      // StmReduce
      FoldOrderedStream(
        // (acc, x) => acc + x
        ArchLambdas(Seq(acc, x),
          AddInt(Tuple2(
            ParamUse(acc),
            ParamUse(x) ) ) ),
        // StmMap
        MapOrderedStream(
          // (xy) => xy.0 * xy.1
          ArchLambda(xy,
            TruncInteger(
              MulInt(ParamUse(xy)),
              16 ) ),
          // StmZip(row, vec)
          Zip2OrderedStream(Tuple2(
            ParamUse(row),
            ParamUse(vec) )) ) ) ),
    ParamUse(mat) ) )
```

Aetherling. The Aetherling source code differs from the Sirop and Shir code in that one must explicitly repeat the input vec. Sirop and Shir automatically insert a streaming operator to perform this repetition.

```
big_mvm =
  let mat =
        com_input_seq "I0"
        (Proxy::Proxy (Seq 65536 Atom_UInt16))
  in
  let vec =
        com_input_seq "I1"
        (Proxy::Proxy (Seq 256 Atom_UInt16))
  in
  -- Doesn't work :( --------------------------
  -- mapC (\row -> dot row vec) $
  --   partitionC (Proxy @256) (Proxy @256) mat
  -- ----------------------------------------
  let repeated_vec = unpartitionC $
                     up_1dC (Proxy @256) $
                     partitionC (Proxy @1)
                                (Proxy @256)
                                vec in
  -- Element-wise product of vec with each row
  let products2d = partitionC (Proxy @256)
                              (Proxy @256) $
                   map2C (\x -> \y -> mulC $
                            atom_tupleC x y)
                         mat repeated_vec in
  -- Sum each row
  unpartitionC $ mapC (reduceC addC) products2d
```

Intel HLS. The Intel HLS code is written in C++. Inputs and outputs are provided using the `ihc::pipe` class, which exposes a ready/valid interface like the one in figure 3. The input vec is saved in an array so it can be read repeatedly.

```
#include "HLS/ac_int.h"
#include "HLS/hls.h"

constexpr int N = 256;
template<unsigned SystemID> class PipeID {};
ihc::pipe<class PipeID<0>, uint16> mat;
ihc::pipe<class PipeID<1>, uint16> vec;
ihc::pipe<class PipeID<2>, uint16> out;

component void matvec() {
    uint16 vec_arr[N];
    for (int i = 0; i < N; i++) {
        uint16 sum = 0;
        for (int j = 0; j < N; j++) {
            uint16 v;
            if (i == 0) {
                v = vec.read();
                vec_arr[j] = v;
            } else {
                v = vec_arr[j];
            }
            sum += mat.read() * v;
        }
        out.write(sum);
    }
}
```

References

[1] Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. 2022. Bridging Python to Silicon: The SODA Toolchain. *IEEE Micro* 42, 5 (2022). doi:10.1109/MM.2022.3178580

[2] C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis.* PhD Thesis - Research UT, graduation UT. University of Twente, Netherlands. doi:10.3990/1.9789036538039

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. doi:10.1145/2228360.2228584

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. *SIGPLAN Not.* 34, 1 (1998). doi:10.1145/291251.289440

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 24 (Sept. 2013). doi:10.1145/2514740

[6] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI (June 2024). doi:10.1145/3656401

[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 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. doi:10.1145/2967938.2967969

[8] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL '17)*. doi:10.23919/FPL.2017.8056860

[9] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. doi:10.1145/1291151.1291199

[10] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Dylan Rankin, Ryan Rivera, Sioni Summers, Nhan Tran, and Zhenbin Wu. 2019. Fast Inference of Deep Neural Networks for Real-time Particle Physics Applications. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. doi:10.1145/3289602.3293986

[11] 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 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. doi:10.1145/3385412.3385983

[12] Stephen A. Edwards. 2002. High-Level Synthesis from the Synchronous Language Esterel. In *11th IEEE/ACM International Workshop on Logic & Synthesis (IWLS 2002)*. https://www.cs.columbia.edu/~sedwards/papers/edwards2002high-level.pdf

[13] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, Andrew Young, and Richard Xia. 2021. MLIR as Hardware Compiler Infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*.

[14] Andrew Gill, John Launchbury, and Simon Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. doi:10.1145/165180.165214

[15] Sergei Gorlatch and Murray Cole. 2011. *Parallel Skeletons.* doi:10.1007/978-0-387-09766-4_24

[16] 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 (July 2014). doi:10.1145/2601097.2601174

[17] 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 (July 2016). doi:10.1145/2897824.2925892

[18] Louis Hildebrand and Christophe Dubach. 2025. *Sirop: A Small IR for HLS with Parallel Patterns (LCTES '26 Artifact).* doi:10.5281/zenodo.20090976

[19] Lana Josipović, Nithin George, and Paolo Ienne. 2016. Enriching C-based high-level synthesis with parallel pattern templates. In *International Conference on Field-Programmable Technology (FPT '16)*. doi:10.1109/FPT.2016.7929527

[20] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. doi:10.1145/3174243.3174264

[21] Tzung-Han Juang, Christof Schlaak, and Christophe Dubach. 2023. Let Coarse-Grained Resources Be Shared: Mapping Entire Neural Networks on FPGAs. *ACM Trans. Embed. Comput. Syst.* 22, 5s (Sept. 2023). doi:10.1145/3609109

[22] Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress*. https://api.semanticscholar.org/CorpusID:18030506

[23] 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 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. doi:10.1145/3192366.3192379

[24] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. 2019. High-level synthesis of functional patterns with Lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019)*. doi:10.1145/3315454.3329957

[25] Morihiro Kuga, Kansuke Fukuda, Motoki Amagasaki, Masahiro Iida, and Toshinori Sueyoshi. 2017. High-level Synthesis based on Parallel Design Patterns using a Functional Language. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART '17)*. doi:10.1145/3120895.3120918

[26] 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 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. doi:10.1145/3289602.3293910

[27] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*. doi:10.1109/CGO51591.2021.9370308

[28] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20)*. doi:10.1145/3373087.3375320

[29] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types.

In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. doi:10.1145/3385412.3385974

[30] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. doi:10.1145/3445814.3446712

[31] Rishiyur S. Nikhil. 2008. *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*. doi:10.1007/978-1-4020-8588-8_8

[32] M. Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. 2020. AnyHLS: High-Level Synthesis With Partial Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020). doi:10.1109/TCAD.2020.3012172

[33] 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. Archit. Code Optim.* 14, 3, Article 26 (Aug. 2017). doi:10.1145/3107953

[34] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. 2017. Generating FPGA-based image processing accelerators with Hipacc: (Invited paper). In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. doi:10.1109/ICCAD.2017.8203894

[35] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. 2011. A study of high-level synthesis: Promises and challenges. In *9th IEEE International Conference on ASIC*. doi:10.1109/ASICON.2011.6157401

[36] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. 2022. Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators. *ACM Trans. Archit. Code Optim.* 19, 2 (Jan. 2022). doi:10.1145/3501768

[37] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. doi:10.1145/3122948.3122949

[38] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. doi:10.1109/MICRO.2016.7783720

[39] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. 2018. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 11, 1, Article 7 (March 2018). doi:10.1145/3180481

[40] Josef Svenningsson. 2002. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. doi:10.1145/581478.581491

[41] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. doi:10.1145/3373376.3378495

[42] Jonathan Van der Cruysse and Christophe Dubach. 2024. Latent Idiom Recognition for a Minimalist Functional Array Language Using Equality Saturation. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '24)*. doi:10.1109/CGO57630.2024.10444879

[43] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. 2021. Reticle: a virtual machine for programming modern FPGAs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. doi:10.1145/3453483.3454075

[44] Stylianos I. Venieris and Christos-Savvas Bouganis. 2016. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '16)*. doi:10.1109/FCCM.2016.22

[45] Philip Wadler. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990). doi:10.1016/0304-3975(90)90147-A

[46] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. doi:10.1145/3489517.3530631

[47] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2019). doi:10.1109/TCAD.2017.2785257

[48] Weichuang Zhang, Jieru Zhao, Guan Shen, Quan Chen, Chen Chen, and Minyi Guo. 2024. An Optimizing Framework on MLIR for Efficient FPGA-based Accelerator Generation. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. doi:10.1109/HPCA57654.2024.00017

[49] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A. Constantinides. 2022. POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. doi:10.1109/FPL57034.2022.00044

Received 2026-03-20; accepted 2026-05-01