Type-Directed Scheduling of Streaming Accelerators

David Durst
Stanford University
USA

Matthew Feldman
Stanford University
USA

Dillon Huff
Stanford University
USA

David Akeley
University of California, Los Angeles
USA

Ross Daly
Stanford University
USA

Gilbert Louis Bernstein
University of California, Berkeley
USA

Marco Patrignani*
Stanford University
USA

Kayvon Fatahalian
Stanford University
USA

Pat Hanrahan
Stanford University
USA

Abstract

Designing efficient, application-specialized hardware accelerators requires assessing trade-offs between a hardware module's performance and resource requirements. To facilitate hardware design space exploration, we describe Aetherling, a system for automatically compiling data-parallel programs into statically scheduled, streaming hardware circuits. Aetherling contributes a space- and time-aware intermediate language featuring data-parallel operators that represent parallel or sequential hardware modules, and sequence data types that encode a module's throughput by specifying when sequence elements are produced or consumed. As a result, well-typed operator composition in the space-time language corresponds to connecting hardware modules via statically scheduled, streaming interfaces.

We provide rules for transforming programs written in a standard data-parallel language (that carries no information about hardware implementation) into equivalent space-time language programs. We then provide a scheduling algorithm that searches over the space of transformations to quickly generate area-efficient hardware designs that achieve a programmer-specified throughput. Using benchmarks from the image processing domain, we demonstrate that Aetherling enables rapid exploration of hardware designs with different throughput and area characteristics, and yields results that require 1.8-7.9× fewer FPGA slices than those of prior hardware generation systems.

*Also with CISPA Helmholtz Center for Information Security.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI '20, June 15–20, 2020, London, UK

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7613-6/20/06...$15.00
https://doi.org/10.1145/3385412.3385983

CCS Concepts: • **Hardware → Hardware description languages and compilation**; • **Software and its engineering → Data types and structures**; *Data flow languages*; • **Computer systems organization** → *Data flow architectures.*

Keywords: space-time types, hardware description languages, scheduling, image processing, FPGAs

ACM Reference Format:
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 International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385983

1 Introduction

The need for energy-efficient computing is driving computer architects to aggressively explore new kinds of application-specific hardware accelerators [22]. Correspondingly, there is a growing need for tools that improve the productivity of creating custom hardware designs. One common approach is to directly compile high-level algorithm descriptions into hardware circuits. This strategy is reflected by recent domain-specific compilation systems targeting compute-intensive application domains such as machine learning [12, 26] and image processing [20, 21, 39, 44]. However, to arrive at efficient solutions, it is also common for a hardware designer to assess trade-offs between performance achieved and hardware resources used by exploring a space of different hardware designs for a single algorithm.

In this paper, we focus on the domain of classic image processing and present Aetherling, a system for automatically compiling programs in a high-level, data-parallel language into a range of streaming hardware designs featuring different throughputs and resource requirements. Rather than model statically scheduled, streaming hardware using synchronous dataflow representations [7, 21, 29, 44], our approach is to encode the parallelism (*space*) and throughput

(space per unit *time*) properties of streaming hardware interfaces in the type system of a high-level, but *space-time aware*, data-parallel language. As a result, all well-typed programs in the space-time language are guaranteed to compile to statically scheduled, acyclic, streaming hardware designs. Aetherling enables rapid exploration of such designs with different throughput and area trade-offs using a heuristic-driven search over rewrite rules that transform an input program in a hardware-agnostic language into different implementations in the space-time aware language. Although Aetherling does not guarantee that its statically scheduled designs minimize resource utilization, we show that Aetherling's output typically requires less control logic and memory overhead than dynamically scheduled designs.

Specifically, we make the following contributions:

1. We define a space-time intermediate language featuring: (a) sequence types that encode the parallelism and throughput of streaming hardware interfaces; and (b) operators that correspond to hardware modules with computable properties such as throughput, area, and delay. All well-typed programs in this language have a direct interpretation as statically scheduled, streaming hardware (Section 4). In this paper, implementations of these programs are restricted to acyclic accelerators without asynchronous control flow that only access off-accelerator memory at the beginning and end of the pipeline.
2. We provide rewrite rules for transforming programs expressed in a second (hardware unaware) functional data-parallel language (Section 3) into equivalent space-time language programs (Section 5).
3. We provide an algorithm for automatically transforming programs expressed in Aetherling's data-parallel input language to efficient space-time language programs that meet a specified throughput (Section 6).
4. We provide an implementation of the Aetherling input language, space-time language, and scheduler that synthesizes FPGA designs (Section 7). We schedule basic image processing programs using this system and demonstrate that the resulting designs require less area for control overhead than those produced by recent systems that generate image processing hardware from high-level language descriptions (Section 8).

We refer the reader to the supplementary material at http://aetherling.org for the full formalization (syntax, typing, and operational semantics) of Aetherling's input language and space-time language.

2 Overview

To better understand Aetherling's hardware synthesis goals, consider creating a circuit that convolves a 1D input stream with a 3-element filter (assume all filter weights are 1/3).

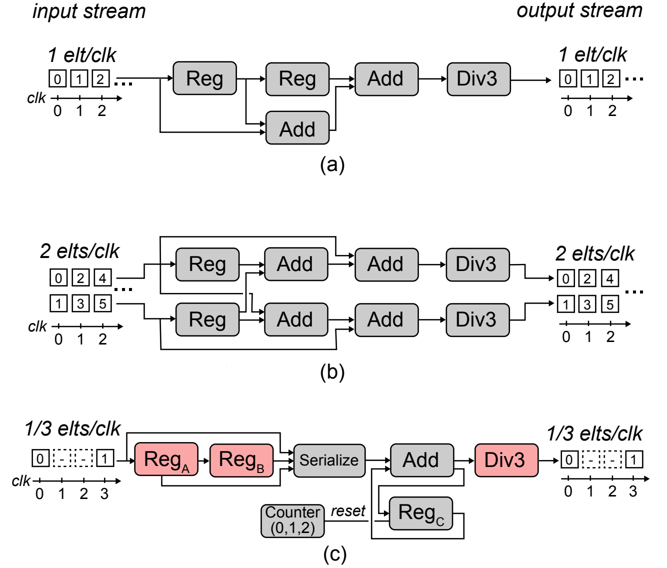

Figure 1. These three circuits compute the same 1D convolution, but utilize different amounts of hardware to achieve different output throughputs. In the bottom design, components labeled in red are underutilized and only emit output elements every third clock.

Pseudocode for this operation, defined on an input sequence `in`, is given below:

```
out[i] = (in[i-2] + in[i-1] + in[i]) / 3
```

One potential circuit for performing this operation is shown in Figure 1-a. In this circuit, stream elements arrive at the hardware's input interface at a rate of one element per clock. These elements stream through a chain of two registers that hold the previous two elements of the stream (a register accepts an input on clock i and emits it on clock $i+1$). The circuit sums the three most recent stream elements using two adders, then divides the result by three to compute the convolution output. Since all components of the circuit produce and consume elements at the same throughput (one element per clock), the circuit requires no additional storage for buffering intermediate data between components, yielding a resource-efficient design.

A hardware designer can also explore alternative designs that compute the same output, but use different amounts of hardware to achieve different throughputs. For example, Figure 1-b shows a design that doubles output throughput to two elements per clock at the cost of using four adders and two dividers. The circuit in Figure 1-c uses only a single adder, serializing per-output summation work over three clocks. As a result, the circuit's input/output interfaces transfer only one stream element every three clocks. This final design includes *underutilized hardware* (shown in red) that sits idle for two of every three clocks. Although all throughputs are not the same in this *multi-rate* design, all connected circuit

components produce and compute elements at the same rate, so no buffering hardware is required between operators.

Even in this simple convolution example, changing the circuit to adjust throughput is more complex than just duplicating (or removing) hardware resources from the basic one element per clock design. For example, the wiring pattern between the stateful components (registers) and combinational components (adders, dividers) needed to be changed between the designs in Figure 1-a and b. In Figure 1-c, new components (a unit for serializing stream elements to the adder over time and a counter for resetting a register's state to 0), along with underutilization of some components, are required for a correct design.

To reduce this complexity, the Aetherling system facilitates hardware design exploration by automatically handling the throughput-adjusting changes to a circuit. Aetherling programmers must only specify a data-parallel algorithm (without specifics of hardware implementation details) and a desired throughput for a hardware design to achieve. Given these inputs, Aetherling will automatically produce a statically scheduled, streaming hardware design that executes the specified algorithm at the required throughput.

To achieve this goal, the Aetherling system consists of:

1. A functional input language (L^{seq}) for expressing programs using standard data-parallel operations on finite-length sequences. Aetherling input programs do not specify hardware implementation details, so programmers can maintain focus on functional correctness. However, L^{seq} is constrained so that Aetherling can transform all valid programs into custom hardware implementations.
2. An intermediate language (L^{st}) that is similar to L^{seq}, but that models the throughput and area of a program's hardware implementation. A key aspect of L^{st} is *space-time aware* sequence types that define the parallelism and throughput of hardware interfaces by encoding *when* sequence elements are produced in addition to sequence length.
3. A set of rewrite rules for transforming any L^{seq} program into a semantically-equivalent L^{st} program.
4. A scheduling algorithm that, given a L^{seq} program P and a desired hardware circuit throughput T, uses the rewrite rules to schedule P into an equivalent L^{st} program with throughput T.
5. A compiler for transforming L^{st} programs into synthesizable Verilog hardware descriptions.

The following sections describe each of these components in greater detail.

3 Sequence Language

Aetherling's input language, called L^{seq}, is a data-parallel language embedded in Haskell with a first-class sequence data type and standard sequence operators. Most aspects of L^{seq} are similar to prior functional data-parallel languages [11], but L^{seq} is constrained so all programs can be compiled to hardware circuits. Most notably, L^{seq} is finitary: all sequences are of statically-known length, and the language only allows expressions that are DAGs of computations.

```
1  conv_math x =
2    map (\y -> div (tuple y 3)) (reduce add x)
3
4  conv1d input =
5    let shift_once = shift input
6    let shift_twice = shift shift_once
7    let window_tuple = map2 tuple_append
8      (map2 tuple shift_once shift_twice) input
9    let window = map tuple_to_seq
10     (partition N 1 window_tuple)
11   let result = map conv_math window
12   unpartition result
```

Figure 2. L^{seq} program for convolving elements of a 1D input sequence with a 3-element filter with weights 1/3.

```
add :: Int × Int -> Int
tuple :: t -> t' -> t × t'
tuple_append :: (t× ... ×t){n−1} -> t -> (t× ... ×t){n}
map :: (t -> t') -> Seq n t -> Seq n t'
map2 :: (t -> t' -> t'') -> Seq n t -> Seq n t' ->
  Seq n t''
reduce :: (t × t -> t) -> Seq n t -> Seq 1 t
shift :: Seq n t -> Seq n t
select_1d :: Int -> Seq n t -> Seq 1 t
tuple_to_seq :: Seq 1 (t× ... ×t){n} -> Seq n t
partition :: (no :: Int) -> (ni :: Int) ->
  Seq (no*ni) t -> Seq no (Seq ni t)
unpartition :: Seq no (Seq ni t) -> Seq (no*ni) t
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Figure 3. Signatures of L^{seq} operators (excerpts, n are natural numbers, a, b, c, t, t', t'' are types).

L^{seq} programs operate on homogeneous, fixed-length sequences. Type (`Seq n t`) represents a sequence containing n elements of type t. The length of the sequence is encoded using dependent types. Sequences can be arbitrarily nested.

Figure 2 presents the L^{seq} program for the 1D convolution example introduced in Section 2. The `input`, whose type is (`Seq n Int`), is tupled with two other sequences obtained by repeatedly `shift`ing `input` by one element to the right (Lines 5, 6). The result is a sequence of tuples (`window_tuple`) that each contain three consecutive elements of `input`. These tuples are converted to length-3 sequences (Lines 9, 10) so data-parallel operators `reduce` and `map` can be used to perform convolution arithmetic (Line 11).

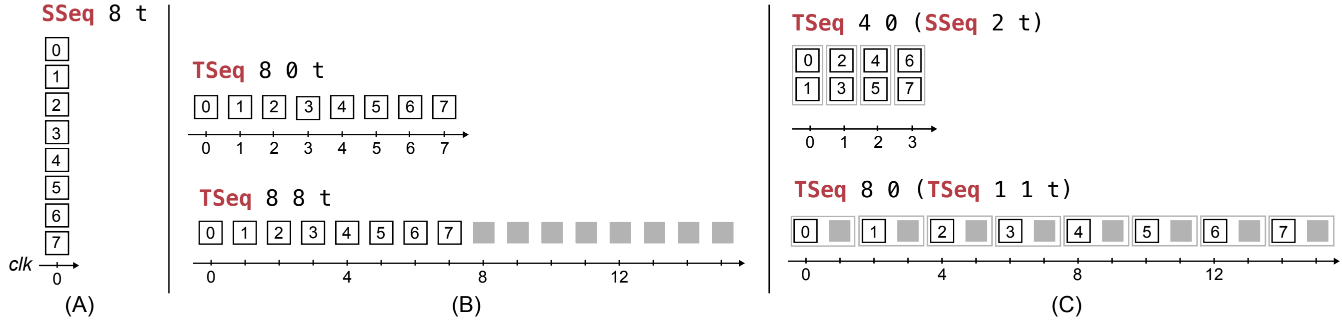

Figure 4. The `SSeq` and `TSeq` sequence types in L^{st} describe the number of elements processed by an operator and also when elements arrive at an operator's interface. Numbers in the boxes are the *indices* of each element (in the sequence) and not the contents. We maintain this notation in all figures in this paper.

Figure 3 provides type signatures for the subset of L^{seq} used in Figure 2. The semantics of most of these operators are standard and therefore omitted, but we describe the non-standard operators. The `(partition no ni)` operator creates nested sequences (or, matrices) by dividing a `(Seq (no*ni)t)` into `no` subsequences each of length `ni`. The `unpartition` operator performs the inverse operation, merging (or, flattening) nested sequences into a single sequence. Both operators preserve the order of input sequence elements so that `(unpartition . partition no ni)` is an identity operation.

4 Space-Time IR

Aetherling transforms L^{seq} programs into programs in its *space-time-aware* sequence language L^{st}. Like L^{seq}, L^{st} is a high-level language that operates on fixed-length sequences. However, all programs in L^{st} explicitly encode key properties of their hardware implementation. Notably, all sequence operators in L^{st} correspond to streaming hardware modules. L^{st} defines the interfaces to these modules using *space-time sequence types* which not only encode the lengths of sequences communicated between modules, but *when* elements in the sequences are communicated. In this paper, statements about the interface to a L^{st} operator can be viewed as a description of the corresponding hardware module.

4.1 Space-Time Sequence Types

L^{st} programs operate on two types of sequences: a *space sequence* `(SSeq n t)` and a *time sequence* `(TSeq n i t)`. These types encode the corresponding hardware's interfaces, so a well-typed L^{st} program describes a circuit where all modules are synchronized with their producers and consumers on the timing of sequence element communication. This synchronization enables use of efficient, statically-scheduled hardware modules, since no dynamic interface logic is needed to determine when new data arrives. Further, inter-module buffering between producers and consumers is not required, except to align element arrival clock cycles when two sequences are joined (Section 7).

`SSeq` encodes a fully parallel operator interface. An operator that produces a sequence of type `(SSeq n t)` emits `n` values of type `t` in parallel, over the same number of clocks required to emit a single element of type `t`. Sequences in L^{st} can be nested, so `t` can be another `SSeq`. Since parallel hardware is required to implement an operator that communicates all elements at once, we say elements of a `SSeq` are distributed "in space".

L^{st} sequence types encode the throughput of interfaces. For example, Figure 4-A illustrates the behavior of an interface described by `(SSeq 8 Int)`. All eight integers are communicated at the same time, so the type has a throughput of eight `Int`s per clock. Generally, the throughput of an operator producing a `(SSeq n t)` is `n` times the throughput of `t`.

`TSeq` encodes a fully sequential operator interface where sequence elements arrive over time. To describe interfaces that may be underutilized, the `TSeq` type describes when elements are communicated over the interface (*valid clocks*) as well as when the interface is idle (*invalid* clocks).

An operator that produces a `(TSeq n i t)` emits `n` values of type `t` over $n \times c$ clocks, where `c` is the number of clocks required to emit `t`. Then, the operator emits nothing for an additional $i \times c$ clocks. We refer to the number of clock cycles for an interface to accept or emit a program's entire sequence as the interface's *time*. In the example above, the interface's time is $(n + i) \times c$.

Figure 4-B illustrates interfaces described by types `(TSeq 8 0 Int)` and `(TSeq 8 8 Int)`. Notice that both interfaces emit eight integers over the first eight clocks. In the first case, the interface has a time of eight clocks. In the latter case, the valid elements are followed by eight invalid clocks (the gray boxes). Due to these invalids, `(TSeq 8 8 Int)` describes an interface with a time of 16 clocks to transfer the eight-element sequence, and its throughput is half that of `(TSeq 8 0 Int)`.

While it may seem like invalid clocks imply inefficient hardware designs, they are fundamentally required for L^{st} types *to describe the behavior of all hardware interfaces for the entire duration of a circuit's execution.* When operators

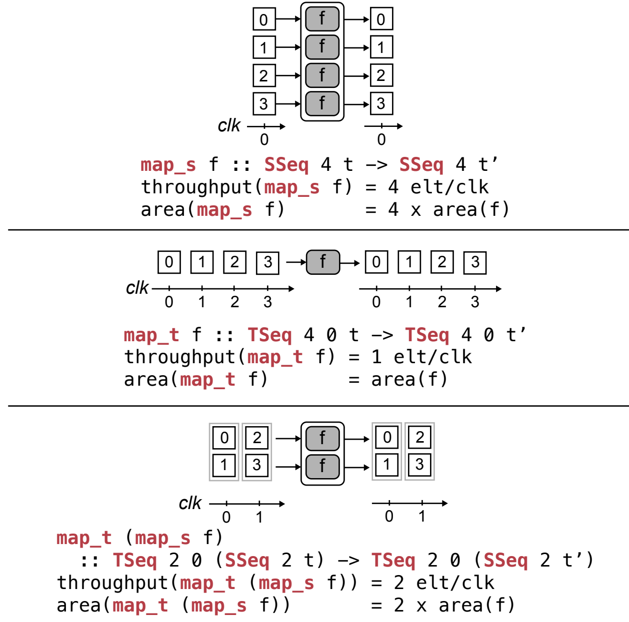

Figure 5. Three L^{st} operator expressions that apply `f` to all elements of the input sequence. The corresponding hardware implementations differ in throughput and area.

produce and consume sequences with different lengths (e.g., reductions, downsampling), invalid clocks enable Aetherling to describe both when the operators' interfaces are communicating valid data and when they are idle. The input and output sequences for all Aetherling's space-time operators (Section 4.2) *require the same amount of time to communicate.* As a result, the interfaces of all operators in a well-typed L^{st} program require the same amount of time.

Nesting `TSeq` and `SSeq` types can be used to describe a rich space of operator interfaces. For example, the top illustration in Figure 4-C provides an example of a *partially parallel* interface described by the type (`TSeq 4 0 (SSeq 2 Int)`). This interface always emits valid elements and transfers a length-8 sequence two elements at a time over four clocks. The bottom half of Figure 4-C illustrates how nesting `TSeq` types can be used to describe a fine-grained valid/invalid pattern on an underutilized interface.

4.2 Operators

To express hardware designs with different throughput and area, L^{st} provides parallel and sequential versions of all sequence operators previously seen in L^{seq}. These operators respectively implement the parallel and sequential interfaces described by the `SSeq` and `TSeq` types.

For example, the *spatial (or parallel) map operator* `map_s` operates on `SSeq`s. Given a function `f` (corresponding to a hardware circuit), the operator (`map_s f`) consists of `n` copies of `f` that simultaneously process all `n` (simultaneously arriving) elements of the input sequence. Similarly, the *temporal*

```
add :: (Int x Int) -> Int
tuple :: t -> t' -> t x t'
                    n−1                  n
tuple_append :: t× ... ×t -> t -> t× ... ×t
map_s :: (t -> t') -> SSeq n t -> SSeq n t'
map_t :: (t -> t') -> TSeq n i t -> TSeq n i t'
map2_s :: (t -> t' -> t'') -> SSeq n t ->
  SSeq n t' -> SSeq n t''
map2_t :: (t -> t' -> t'') -> TSeq n i t ->
  TSeq n i t' -> TSeq n i t''
reduce_s :: (t x t -> t) -> SSeq n t -> SSeq 1 t
reduce_t :: (t x t -> t) -> TSeq n i t ->
  TSeq 1 (n+i-1) t
shift_s :: SSeq n t -> SSeq n t
shift_t :: TSeq n i t -> TSeq n i t
select_1d_s :: Int -> SSeq n t -> SSeq 1 t
select_1d_t :: Int -> TSeq n i t -> TSeq 1 (n+i-1) t
reshape :: t -> t'
```

Figure 6. Signatures of L^{st} operators (excerpts).

map operator (`map_t f`) corresponds to a single copy of the `f` circuit that processes elements arriving at its interface sequentially in time. Figure 5 illustrates three examples of operators that map `f` onto all elements of an input sequence but do so with different throughputs.

Figure 6 provides signatures for a selection of L^{st} operators needed to express the 1D convolution application from Figure 2. Most operators are standard data-parallel operators; therefore, we defer definition of their semantics to the supplementary material. However, we call attention to two important details.

Multi-rate operators. `reduce_t` and `select_1d_t` are temporal multi-rate operators that accept `TSeq`'s of length `n` and emit `TSeq`'s of length 1. The output sequences of these operators contain additional invalid elements that ensure their input and output sequences have the same time.

Reshape operator. `reshape` is a data shuffling operator that converts between two space-time sequence types with the same element throughput (e.g., (`TSeq 1 3 (SSeq 4 Int)`) and (`TSeq 4 0 Int`)).[1] The correspondence of elements between the input and output sequences is defined by flattening. If both input and output sequences were standard Haskell lists, recursively flattening both lists and eliminating invalid values would produce two identical lists.

`reshape` is unique among L^{st} operators in that its resource requirements can vary significantly depending on the combination of input and output L^{st} types used. Implementations of `reshape` may range from being simple data serializers to needing to buffer an entire sequence (Section 7). We find this lack of cost transparency to not be an issue in practice because for our benchmarks Aetherling's scheduler (Section 6)

[1] `reshape` treats homogeneous tuples as `SSeq`'s.

```
conv_math :: SSeq 3 Int -> SSeq 1 Int
conv_math x =
   map_s (\y -> div (tuple y 3)) (reduce_s add x)

conv1d :: TSeq n 0 Int -> TSeq n 0 Int
conv1d input =
  shift_once :: TSeq n 0 Int
  let shift_once = shift_t input
  shift_twice :: TSeq n 0 Int
  let shift_twice = shift_t shift_once
  window_tuple :: TSeq n 0 (Int × Int × Int)
  let window_tuple = map2_t tuple_append
    (map2_t tuple shift_once shift_twice) input
  window :: TSeq n 0 (SSeq 3 Int)
  let window = map_t reshape
    (reshape window_tuple)
  result :: TSeq n 0 (SSeq 1 Int)
  let result = map_t conv_math window
  reshape result
```

Figure 7. L^{st} program for a 1D convolution emitting one pixel per clock.

only generates programs that utilize forms of `reshape` with resource-efficient implementations.

Figure 7 provides a L^{st} implementation of the 1D convolution example from Figure 2. The program emits one output per clock (`TSeq n Int`) and describes a circuit similar to that shown in Figure 1-a. Most operators in the L^{st} implementation are the temporal versions of their equivalents in L^{seq}. However, parallel operators `map_s` and `reduce_s` perform the arithmetic for each output element in parallel. Instances of `reshape` in the L^{st} code correspond to uses of `partition`, `unpartition`, and `tuple_to_seq` in the L^{seq} program.

4.3 Operator Properties

In addition to specifying the throughput and timing properties of their interfaces, all L^{st} operator definitions include two additional properties that describe their hardware implementations: *area* and *delay*. Using these per-operator definitions, Aetherling can compute the area and delay of any L^{st} program.

Area measures the hardware resources required to implement an operator (e.g., on an FPGA, resources include LUTs, DSPs, and BRAMs). The scheduling algorithm described in Section 6 uses L^{st} program area estimates during its search for area-efficient hardware implementations.

Delay measures the number of clock cycles from the first element of an input sequence arriving at an operator to the first element emitted by the operator. For example, (`map_t add`) is a fully combinational operator, so applying it to a (`TSeq 4 0 Int`) input means the operator has zero clocks of delay and the time of its input and output interfaces is four clocks. On the other hand, if `mul` is implemented by

```
⟦add⟧ ⟶ add                    ⟦reduce f⟧ ⟶ reduce_s ⟦f⟧
⟦tuple⟧ ⟶ tuple                ⟦reduce f⟧ ⟶ reduce_t ⟦f⟧
⟦map f⟧ ⟶ map_s ⟦f⟧            ⟦select_1d⟧ ⟶ select_1d_s
⟦map f⟧ ⟶ map_t ⟦f⟧            ⟦select_1d⟧ ⟶ select_1d_t
⟦map2 f⟧ ⟶ map2_s ⟦f⟧          ⟦tuple_to_seq⟧ ⟶ reshape
⟦map2 f⟧ ⟶ map2_t ⟦f⟧          ⟦unpartition⟧ ⟶ reshape
⟦shift⟧ ⟶ shift_s              ⟦partition no ni⟧ ⟶ reshape
⟦shift⟧ ⟶ shift_t
```

Figure 8. L^{seq} to L^{st} Direct Rewrite Rules (excerpts).

hardware with four pipeline stages, (`map_t mul`) when applied to the same input has a delay of four clocks and the time of its input and output interfaces is four clocks. The L^{st} compiler statically computes the delay of all operators to generate correct hardware implementations of L^{st} programs (Section 7). Therefore, Aetherling doesn't support variable-latency operators such as floating-point multipliers that are optimized to change their delay depending on the input.

The interested reader will find formulas for the area and delay of all L^{st} operators in the supplemental material.

5 Rewrite Rules

In this section, we describe the rewrite rules used to transform L^{seq} programs into L^{st}. We first define rewrite rules that yield fully-parallel and fully-sequential translations into L^{st} (Section 5.1). Then, we provide additional rewrite rules from L^{seq} to L^{seq} that are used to explore the space of partially-parallel translations (Section 5.2). Finally, we argue that these rewrite rules are semantics preserving (Section 5.4). For these rules, we cannot state that an operator and its rewriting are equivalent as the input and output values of L^{seq} and L^{st} programs have different types (`Seq` and `SSeq`/`TSeq` respectively). We therefore provide an isomorphism between `Seq` and `SSeq`/`TSeq` and state that a L^{seq} program and its translation into L^{st} have equivalent semantics up to this isomorphism (Section 5.3).

5.1 L^{seq} to L^{st} Direct Rewrite Rules

The direct rewrite rules from L^{seq} to L^{st} convert each L^{seq} operator to a L^{st} operator that is either fully parallel or fully sequential. Figure 8 shows the subset of these rewrite rules used for lowering the convolution example from L^{seq} in Figure 2 to L^{st} in Figure 7.

5.2 Nesting Rewrite Rules

To enable transformations that yield partially parallel L^{st} operators, Aetherling has rewrite rules from L^{seq} to L^{seq} which we call *nesting rewrite rules* (Figure 9). These rules take an operator with an input type of a single (`Seq (no*ni)t`) and

```
1  map f ⟶ map (map f)
2  map2 f ⟶ map2 (map2 f)
3  reduce f ⟶ reduce (map f) . map (reduce f)
4  shift in_seq ⟶
5    let fst_seq = shift . map (select_1d (ni-1)) in_seq
6    let (snd_seq:other_seq) =
7      [map (select_1d i) in_seq | i <- [0..ni - 2]]
8    let result_tuples =
9      foldl (\xs x -> map2 (map2 tuple_append) xs x)
10     (map2 (map2 tuple) fst_seq snd_seq) other_seq
11   map tuple_to_seq result_tuples
12 select_1d i ⟶
13   select_1d no (i//no) . map (select_1d ni (i%no))
```

Figure 9. L^{seq} Nesting Rewrite Rules (excerpts).

produce a nesting of operators with an input type of a nested (`Seq no (Seq ni t)`). By first applying nesting, then applying direct rewrite rules to the L^{seq} operators produced by nesting, Aetherling can produce partially-parallel L^{st} programs. For example, in the following code, application of the map nesting rule, followed by direct rewrites, converts a L^{seq} `map` on a sequence of 10 elements to a partially parallel L^{st} program that processes two elements at a time.

```
⟦map add :: Seq 10 (Int x Int)->
            Seq 10 Int⟧ ⟶
⟦map (map add):: Seq 5 (Seq 2 (Int x Int))->
            Seq 5 (Seq 2 Int)⟧ ⟶
map_t (map_s add):: TSeq 5 0 (SSeq 2 (Int x Int))->
            TSeq 5 0 (SSeq 2 Int)
```

While most of the nesting rules are standard [9], the one for `shift` is not. For example, consider a `shift` in L^{seq} which takes the input sequence $[0, 1, 2, 3]$ `::` (`Seq 4 Int`) to $[u, 0, 1, 2]$, where u signifies an undefined value. (We choose to shift in an undefined value to simplify hardware implementation.) We need to rewrite that `shift` in order to operate on nested sequences. Consider the case of (`Seq 3 (Seq 2 Int)`), where `shift` inputs $\left[\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}2\\3\end{bmatrix}\begin{bmatrix}4\\5\end{bmatrix}\right]$ and outputs $\left[\begin{bmatrix}u\\0\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}\begin{bmatrix}3\\4\end{bmatrix}\right]$. This can be done by first shifting the last row by one (Figure 9-Line 5) and placing it on the first row and then by moving all other rows (collected in Line 7) down by one (Lines 9, 10).[2]

5.3 Isomorphism

In order to relate elements of L^{seq} and L^{st} as they are manipulated by the rewrite rules, we provide three isomorphisms:

- one between `Seq` and nested `Seq`'s;
- ones between `Seq` and `SSeq`, `Seq` and `TSeq`.

[2]We invite the skeptical reader to try this on larger nested sequences too.

The combination of the three isomorphisms is used to say that the translation from L^{seq} to L^{st} programs that first applies the nesting rewrite rules and then applies the direct rewrite rules is semantics preserving [15].

The intuition behind the isomorphisms is that isomorphic types are inhabited by values that are sequences containing the same values in the same order but with possibly different distributions in time and space. More precisely, once flattened and stripped of invalids, isomorphic sequences must have the same values. So, (`Seq (no*ni)t`) is isomorphic to (`Seq no (Seq ni t)`) if flattening the nested sequence yields the first one. Thus, $[1, 2, 3, 4]$ `::` (`Seq 4 Int`) is isomorphic to $[[1, 2], [3, 4]]$ `::` (`Seq 2 (Seq 2 Int)`). On the other hand, (`Seq n t`) is isomorphic to (`SSeq n t`) if the two sequences have the same length and the same elements. Thus, $[5, 6]$ `::` (`Seq 2 Int`) is isomorphic to $[5, 6]$ `::` (`SSeq 2 Int`). Finally, (`Seq n t`) is isomorphic to (`TSeq n i t`) if the two sequences have the same length and the same elements, disregarding the possible trailing invalid values (indicated with i) in the second sequence. Thus, $[7, 8, 9]$ `::` (`Seq 3 Int`) is isomorphic to $[7, 8, 9, i, i]$ `::` (`TSeq 3 2 Int`).

From these, we conclude that $[1, 2, 3]$ `::` (`Seq 3 Int`) is isomorphic to $[[1, i], [2, i], [3, i], i]$ `::` (`TSeq 3 1 (TSeq 1 1 Int)`). Intuitively, this is because type (`Seq 3 Int`) is isomorphic to (`Seq 3 (Seq 1 Int)`). Then, (`Seq 1 Int`) is isomorphic to type (`TSeq 1 1 Int`), and from this we can derive that (`Seq 3 _`) is isomorphic to (`TSeq 3 1 _`).

5.4 Rewrite Rules Preserve Semantics

We argue that combinations of the rewrite rules are semantics preserving. The direct rewrite rules of Figure 8 trivially preserve semantics up to the isomorphism because they simply convert L^{seq} operators to the same ones in L^{st}. This is true even for the `partition` and `unpartition` rules since they are translated to `reshape`, and they all convert between types while preserving the same ordering of elements.

The nesting rewrite rules of Figure 9 are mostly standard nesting operations [9, 11]. In fact, rules for `map`, `map2`, and `reduce` (as well as for all elided operators) are well-known results from data parallel languages. The `select_1d` rule is a duplicate of the `reduce` rule. The `shift` nesting rewrite rule is similar to the stencils in Lift [18]. As such, it is also simple to see that the nesting rewrite rules preserve the isomorphism.

We defer formally proving that Aetherling's rewrite rules are semantics preserving up to the presented isomorphism to future work. We expect such results to build on similar semantics-preservation proofs [3, 4, 23, 33, 36]. Specifically, by relying on the type systems of L^{seq} and of L^{st}, we expect to build a cross-language logical relation that relates terms (and crucially values) of the two languages when they "behave the same". The key complexity of such a relation will be accounting for the isomorphism in order to determine when two sequences are related or, in technical terms, to determine what values are related at type (`Seq n t`).

6 Scheduling

In this section, we describe an algorithm that uses Aetherling's rewrite rules to transform L^{seq} programs into equivalent L^{st} programs with a specified output throughput T. We refer to this process as "scheduling" since the algorithm determines how to orchestrate a program's execution in space (via parallelism) and time.

Consider scheduling a L^{seq} program that outputs a sequence of 8 integers (`Seq 8 Int`) with a throughput of two ints per clock (T=2). Multiple L^{st} sequence types describe this throughput, for example:

```
TSeq 4 0 (SSeq 2 Int)
TSeq 2 2 (SSeq 4 Int)
TSeq 2 0 (TSeq 1 1 (SSeq 4 Int))
```

L^{st} programs that produce these types correspond to hardware with different area requirements. Therefore, the goal of scheduling is to find the *lowest area* L^{st} program that meets the specified throughput. Aetherling performs this optimization using a heuristic-driven search that involves (1) enumerating a set of L^{st} sequence types with the desired throughput and (2) using rewrite rules from Section 5 to convert the input L^{seq} program into L^{st} programs that output sequences with these types.

6.1 Enumerating Candidate Output Types

Enumerating single sequences. Aetherling's scheduler considers a space of program transformations that take each (`Seq n t`) type in the source L^{seq} program into a L^{st} type of one of the five following forms:

(1) `TSeq n i t`
(2) `TSeq n io (TSeq 1 ii t)`
(3) `TSeq n io (TSeq 1 ii (TSeq 1 ii t))`
(4) `SSeq n t`
(5) `TSeq no io (SSeq ni t)` (for n=no × ni)

These five rules do not generate all possible L^{st} types, but serve as heuristics for limiting the scheduler's search space. The first three forms yield L^{st} sequences that match a target throughput of one pixel per clock (or less, considering invalids). Forms (2) and (3) interleave valid and invalid clocks rather than creating a single burst like form (1). Implementations of programs with multi-rate operators can use more efficient `reshape`s when they avoid bursty behavior and emit valid data at regular intervals. In our experience, the two layers of nesting in form (3) are sufficient for processing data with two dimensions, such as images.[3] Forms (4) and (5) yield fully parallel (T=n) and partially parallel output respectively. Since underutilized parallel hardware is unlikely to be an area efficient solution, Aetherling only applies form (5) with the minimum ni that meets the specified throughput.

[3]Forms (2) and (3) describe the largest part of the search space since multiple values for the parameters io and ii may yield the desired throughput. The number of options for io and ii increases as n increases.

We demonstrate that, for our benchmarks in Section 8, there are area-efficient hardware implementations whose output interfaces lie within the space of L^{st} sequences obtained via these five sequence transformations. For example, when scheduling the 1D convolution with output type (`Seq 8 Int`) (Figure 2) with a throughput T=1/3, Aetherling searches over L^{st} programs with the following output types:

`TSeq 8 16 Int` (from form 1)
`TSeq 8 0 (TSeq 1 2 Int)` (from form 2)
`TSeq 8 4 (TSeq 1 1 Int)` (from form 2)
`TSeq 8 16 (TSeq 1 0 Int)` (from form 2)
`TSeq 8 16 (TSeq 1 0 (TSeq 1 0 Int))` (from form 3)
`TSeq 8 16 (SSeq 1 Int)` (from form 5)

Note that the second type given above describes the interface of the area-efficient hardware pipeline illustrated in Figure 1-c, which emits an element every third clock.

Enumerating nested sequences. The above transformations apply to a single `Seq` type in a L^{seq} program. When a L^{seq} program produces a nested sequence, Aetherling factors the target throughput T into target sub-throughputs T_i for each of the subsequences. For example, to schedule a program with type (`Seq 8 (Seq 2 Int)`) with throughput T, Aetherling will consider transformations that convert the innermost (`Seq 2 Int`) to L^{st} types with throughput T_1 and convert the outermost (`Seq 8 t`) to L^{st} types with throughput T_0, where $T_0 \times T_1 = T$.

In the example above, there are multiple valid assignments of T_0 and T_1. In general, given an arbitrarily nested L^{seq} type, the scheduler enumerates all valid assignments of subsequence throughputs by performing a prime factorization of T and distributing the factors as target subsequence throughputs T_i for each nested `Seq`. For each assignment, the scheduler then uses the five `Seq` transformation forms described above to enumerate possible L^{st} types for each nested `Seq`. This yields a collection of nested L^{st} sequence types that describe candidate hardware output interfaces.

6.2 Transforming Programs to Match Output Types

Given a L^{seq} program and a L^{st} output type T, the scheduler invokes the rewrite rules to transform the L^{seq} program into a L^{st} program of output type T. We first describe how to transform single L^{seq} operators into L^{st} operators of a specified type, then address full L^{seq} program transformations.

6.2.1 Transforming Individual Operators.

Base operators. L^{seq} operators that do not operate on `Seq` types (e.g., `add`) are trivially converted using applications of the appropriate L^{seq} to L^{st} direct rewrite rules (Figure 8).

Sequence operators with nesting rewrite rules. If a nesting rewrite rule for an operator exists, the scheduler repeatedly applies the nesting rewrite (zero or more times) until the resulting L^{seq} expression emits a nested `Seq` with the same nesting depth as the L^{st} target type. Then, the algorithm

applies direct L^{seq} to L^{st} rewrite rules to convert operators in the (potentially) nested L^{seq} expression into L^{st}. For example, consider transforming the L^{seq} expression:

```
map add :: Seq 4 (Int x Int) -> Seq 4 Int
```

into a L^{st} expression with output type `TSeq 2 0 (SSeq 2 Int)`.

Application of the `map` nesting rewrite rule produces:

```
map (map add) ::
    Seq 2 (Seq 2 (Int x Int)) ->
    Seq 2 (Seq 2 Int)
```

Then, application of L^{seq} to L^{st} direct rewrite rules for each `map` produces:

```
map_t (map_s add) ::
    TSeq 2 0 (SSeq 2 (Int x Int)) ->
    TSeq 2 0 (SSeq 2 Int)
```

Sequence operators that transform into `reshape`. Operators that have no nesting rewrite rules but produce `Seq` outputs (`partition`, `tuple_to_seq`) translate to `reshape`. Unlike the other operators in L^{st}, `reshape` is unique in that its input type is not fully determined given its output type. (By definition `reshape` converts between two sequence types.) The input type of `reshape` is determined during full-program scheduling, as described in the subsequent section.

6.2.2 Transforming Full Programs. Aetherling schedules full L^{seq} programs by performing a sequence of individual operator transformations. It begins with the last operator in the L^{seq} program DAG and continues to the front until all DAG nodes have been transformed.

Given a target output type for the program, the scheduler first transforms the last operator in the L^{seq} program DAG into an equivalent L^{st} expression that emits this type. With the exception of transformations that yield `reshape`, this process also determines the input type of the resulting L^{st} expression. Then, the process recurs on predecessor nodes in the L^{seq} DAG, using the input type for the current L^{st} expression as the predecessor's target output type in the next recursive call.

When transformations yield a `reshape`, the scheduler must choose a L^{st} input type for the `reshape`. It uses the process described in Section 6.1 to enumerate potential L^{st} input types for the `reshape`. For each of those types, it recursively calls the scheduling algorithm on the input L^{seq} expression to the `reshape`. Each input type for the `reshape` is a different candidate output type for the upstream expression. The scheduler picks the `reshape` input type that causes the `reshape` and the upstream L^{st} expression to have the least area. (The cost of scheduling is exponential in the number of `reshape`s encountered in a program.)

The program transformation process is repeated for all candidate program output types. Aetherling's scheduler computes the area of all generated L^{st} programs, and retains the program with the least area.

```
sharpen1d input =
  let blur = conv1d input
  let joined = map2 tuple input blur
  map sub joined
```

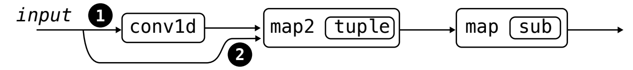

Figure 10. An image sharpening program with fork-join DAG structure. When transforming this L^{seq} program into L^{st}, a `reshape` may need to be inserted at point 1 or point 2 in the DAG to ensure all paths that consume `input` expect the same L^{st} input type.

6.2.3 Handling Fork-Join Structure. The program transformation algorithm described in Section 6.2.2 does not correctly transform DAGs with *fork-join structure* (i.e., when a single sequence is consumed by parallel paths that later merged together, Figure 10). Specifically, independent back-to-front transformation of operators on different DAG paths does not guarantee that all consumers of a sequence require the same L^{st} input type. We solve this problem by inserting a `reshape` on one of the paths so that all consumers require the same L^{st} input type.

It is possible to use `reshape` insertion to solve the type mismatch problem because the different L^{st} types required by each consuming path are guaranteed to have the same throughput. We defined `reshape` in Section 4.2 to convert between any two L^{st} types with the same throughput. The mismatched types must have the same throughput since they have the same length, as they are isomorphic to the same L^{seq} type, and the same time, as stated in Section 4.1.

7 Implementation

Aetherling's L^{seq} is implemented as a shallow embedded DSL in Haskell using the methodology from Bjesse et al. [8]. The L^{seq} type system is implemented using Haskell's lightweight dependent types [14]. The shallow embedding is then compiled to a deep embedding. All L^{seq} and L^{st} scheduling passes are performed on deep embeddings.

Operator Implementation. All L^{st} operators correspond to hardware generators written in Chisel [6], a hardware design language embedded in Scala. These generators produce Verilog. One particularly complex generator is `reshape`. We use efficient implementations of `reshape` for common cases, such as serialization that converts between sequential types like (`TSeq 1 2 (SSeq 3 Int)`) and parallel types like (`TSeq 3 0 Int`). These efficient implementations are sufficient to implement the benchmarks in Section 8. However, other possible input and output L^{st} types for `reshape` cannot be supported by our efficient specializations. We implement the general case of the operator by extending the memory-minimizing stream permutation approach of Koehn and Athanas [25] to support conversion between input and

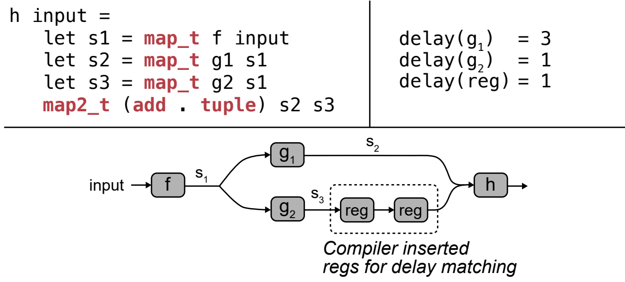

Figure 11. Sequence elements generated by the logic path featuring g2 (delay=1) must be delayed for two cycles so they arrive at the node h at the same time as the corresponding elements produced by g1 (delay=3). The Aetherling compiler relies on the ability to compute the delay of all paths through a program in order to insert the appropriate delay registers.

output sequences featuring invalid clocks. To maintain high clock rates, we perform a simple form of register retiming that inserts registers into operators that have long combinational path lengths.

Delay Matching. The L^{st} type system guarantees that all composed modules match throughputs and also expect the same order of valids and invalids. However, typing does not guarantee that all logic paths through a program DAG feature the same delay. (Recall delay is a property of an operator's internal hardware implementation, not a property of its interface.) As a result, when L^{st} DAGs feature fork-join structure, sequence elements generated from one path through the graph may be produced on an earlier clock than their corresponding elements in the other path. Figure 11 illustrates such a situation, where elements from module g2 (with delay 1) are generated before their corresponding elements from g1 (with delay 3). To ensure correct operation of this design, the Aetherling compiler must insert two registers into the lower path to ensure that corresponding elements arrive at the joining operator h at the same time. As described in Section 4.3, Aetherling is able to compute the delay along all paths of a program in L^{st}, and uses this information to insert the appropriate registers so that all paths through the program's DAG have matching delays.

FPGA Bitstream Generation. We generate clock rate and area numbers for Section 8 by synthesizing the Verilog to bitstreams for the Xilinx XC7K160TIFFV676-2L FPGA using the Xilinx Vivado Design Suite 2018.2.

8 Evaluation

We evaluated Aetherling by generating hardware implementations of several image processing benchmarks (Section 8.1). Using these benchmarks, we demonstrate Aetherling's ability to automatically generate hardware designs with different throughput-area trade-offs (Section 8.2). We also compare the efficiency of Aetherling's designs with the output of two recent hardware generation systems that prioritize design space exploration for data-parallel applications: Halide-HLS [39] and Spatial [26] (Section 8.3). Halide is an image processing DSL that is widely used in industry [40], and Halide-HLS extends Halide's "scheduling" primitives to describe hardware implementations of Halide programs. Spatial is a general-purpose language for creating hardware accelerators using data-parallel patterns. It exposes scheduling directives and has built-in design space exploration for tuning design parameters.

We choose to not compare against additional, prior systems for compiling image processing applications to accelerators (Rigel [21], Darkroom [20], RIPL [44], and Lift [28]) because available implementations of these systems do not fit within our evaluation framework of automatically producing multiple throughput-area trade-offs.

8.1 Benchmark Description

Our benchmarks consist of the following functions:

MAP adds a constant to every element of a 200-element input sequence. This is a test of Aetherling's ability to compile a L^{seq} program to a range of throughput-area trade-offs.

CONV is a 3×3 convolution (without boundary conditions). CONVB2B performs two back-to-back convolutions (3×3 filter, then 2×2). Since the reduction performed in a convolution runs at a higher throughput than downstream operators, these benchmarks feature "multi-rate circuits" where some interfaces are fully utilized while others have invalids.

SHARPEN implements an unsharp mask [24]. This is a common image processing operation that emphasizes high frequency image features (e.g., edges). This benchmark features a DAG with fork-join structure.

CAMERA is a simple version of a modern camera pipeline, a demosaic followed by a sharpen. The demosaic uses a position-dependent convolution.

In most implementations, we use single-channel (32 bits per pixel), 1920x1080 images and fixed-point multiplication and division. For CAMERA, we use three channels (96 bits per pixel). We synthesized designs at the following clock rates: 175 MHz for Aetherling, 161 MHz for Halide-HLS, and 125 MHz for Spatial. We tested the Aetherling designs using Verilator [42], the Spatial designs using VCS, and the Halide-HLS designs using Vivado HLS C simulation.

8.2 Exploring Space-Time Trade-offs

Aetherling enables rapid exploration of throughput-area trade-offs for a L^{seq} program. Figure 12 plots the throughput (in output pixels per clock) and area of designs generated by scheduling the five benchmarks. These designs exhibit significant structural variation, ranging from sub-one-pixel per clock designs requiring extra serializers and counters to higher throughput designs with throughput-specific wiring between shift buffers and arithmetic units.

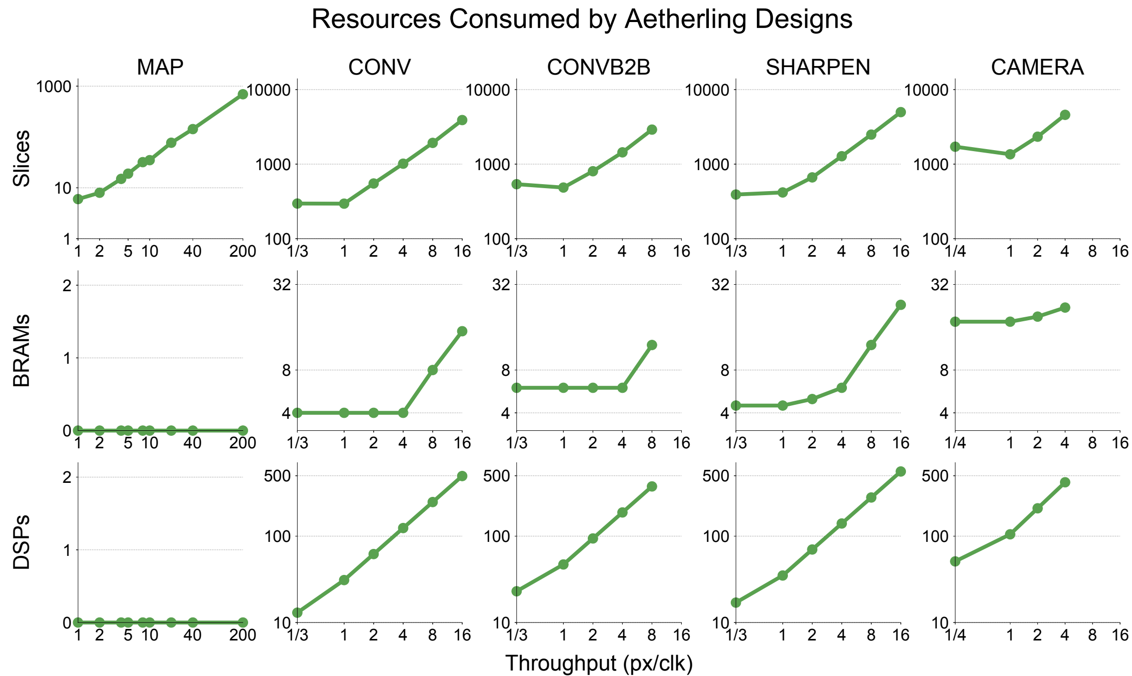

Figure 12. Aetherling generates designs matching a specified output throughput. The resources used for control logic, measured by slices, increases linearly with throughput above one pixel per clock. Designs with throughputs of less than one pixel per clock require additional overhead (such as serializers and counters), resulting in slightly higher slice utilization than the best one pixel-per-clock designs. Data points for high throughput CONVB2B and CAMERA designs are not given because they require too many DSPs for the target FPGA. The graphs use log-scale axes because the parallelism increases exponentially.

Figure 12 uses three metrics for area on the target FGPA: DSPs, BRAMs, and slices. DSPs measure the area used for resource intensive math: the multiplication and division for the convolution. BRAMs measure the area used for buffering rows of the image. (SHARPEN and CAMERA also use BRAMs for buffering needed to match delays.) Slices measure the area used for simple math (e.g., 32-bit adders) and control logic overhead, including counters and registers necessary to support the math and row buffers.

For throughputs above one, we observe a linear relationship between design throughput and slices. Designs for CONV, CONVB2B, and CAMERA at less than one pixel-per-clock have *slightly higher overhead* than the best one pixel-per-clock circuits. In these cases, the extra overhead required to underutilize the convolution hardware (recall the additional components in Figure 1-c) outweighs the area saved by removing parallel adders from the design (CONV only needs eight adders to perform a convolution at one pixel per clock).

We observe a linear relationship between throughput and DSPs. The relationship between throughput and BRAMs has two parts. BRAM capacity to store several rows of pixel data is the dominant constraint at low throughputs, so the number of required BRAMs is constant when targeting four pixels per clock and below. At eight pixels per clock and above, BRAM requirements scale linearly with throughput as additional BRAMs are needed to ensure sufficient read and write bandwidth [46]. SHARPEN's and CAMERA's requirements increase at a lower throughput due to additional, smaller BRAMs used for delay matching.

8.3 Efficiency Comparison

Figure 13 compares the area efficiency of designs produced by Aetherling to those of Spatial for all but the CAMERA benchmark (no Spatial implementation of CAMERA was available). Excluding the simple MAP experiment, Aetherling designs require 1.8-7.9× fewer slices to achieve the same throughput as Spatial. This reduction is due to more efficient control logic. Vivado's design reports for Spatial indicate that slices due to other sources are minimal. For example, the convolution's 32-bit adders are optimized into control logic modules.

Hardware generated by Aetherling does not require control circuity to dynamically manage communication between hardware modules. In contrast, Spatial uses control logic to interface arithmetic with its generated memories. The impact of the control logic, and thus the difference in slice counts between of Spatial and Aetherling designs, is lowest at two pixels per clock since Spatial is able to most efficiently control its memories at this throughput.

Differences in DSP and BRAM utilization between Aetherling and Spatial exist, but are not fundamental to the designs

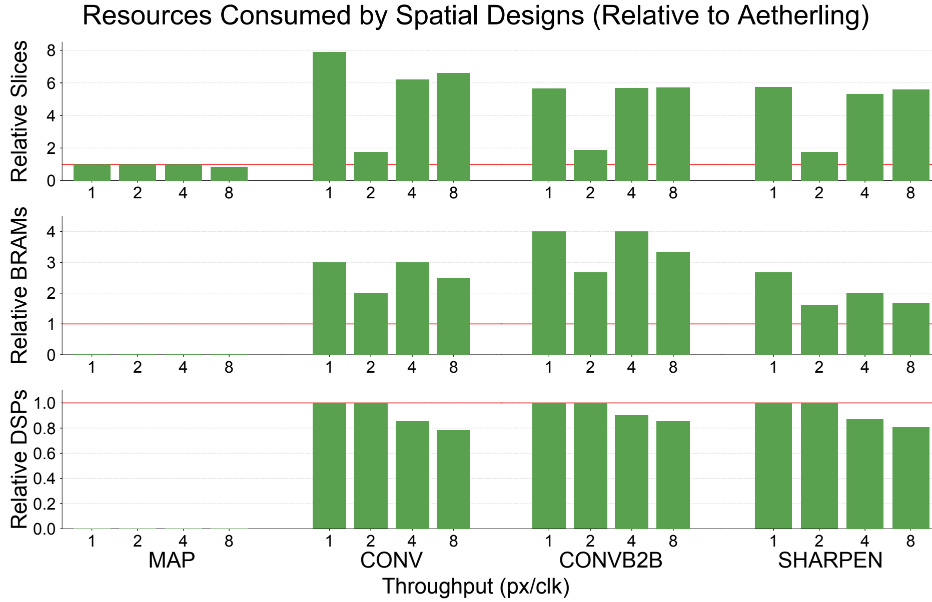

Figure 13. In nearly all benchmarks, designs generated by Spatial require more slices than Aetherling designs with the same throughput. MAP is simple, so Aetherling and Spatial require almost the same number of slices and no BRAMs or DSPs.

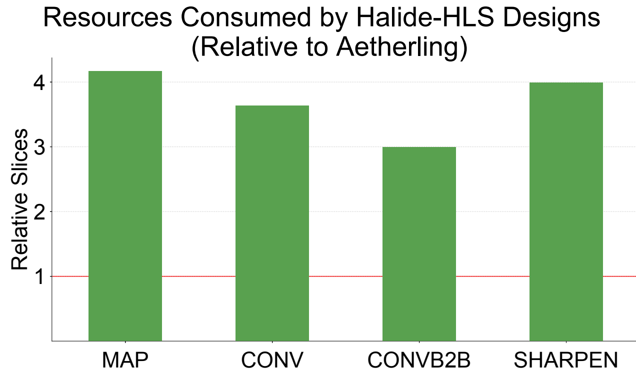

Figure 14. Designs generated by Halide-HLS require more slices than Aetherling designs. All designs have a throughput of one output pixel per clock.

of the systems. Spatial uses more BRAMs because its memory utilization analyzer currently stores more rows of the image than necessary. Spatial saves DSPs at higher throughputs through common subexpression elimination. Spatial's BRAM utilization and Aetherling's DSP utilization could be improved with additional compiler optimization passes.

Figure 14 compares the area efficiency of designs produced by Aetherling to those of Halide-HLS. Excluding the simple MAP experiment, Aetherling designs require 3-4× fewer slices to achieve the same throughput as Halide-HLS. As with Spatial, the plot shows that there's less control overhead compared to Halide-HLS. This is because Halide-HLS compiles programs to C code with high-level synthesis templates. The resulting designs employ dynamic control logic to synchronize each stage of computation.

We elide BRAM consumption in Figure 14 as Aetherling and Halide-HLS use the same number of BRAMs for all benchmarks. We elide DSP consumption because we were unable to direct Halide-HLS to synthesize fixed-point multipliers and dividers to DSPs (despite direct help from the authors of Halide-HLS). To ensure a fair comparison, we synthesized both the Aetherling and Halide-HLS applications in Figure 14 with shifts for the multipliers and dividers. The shifts have negligible impact on the number of slices, and use no DSPs. In Aetherling's CONV, the shifts use so few resources that they are elided from Vivado's area report.

The process of generating different designs using Halide-HLS and Spatial revealed limits to the design space exploration features of these prior systems. For example, although Spatial provides built-in support for tuning the tiling, scheduling, and loop parallelization factors of an algorithm, creating designs that achieved different throughputs with minimal control logic required structural change to the Spatial program. These changes were implemented *out of language* by metaprogramming the Spatial dataflow graph in Scala. We were unable to generate designs matching all desired throughputs using Halide-HLS, even after direct collaboration with that system's authors.

9 Related Work

The ubiquity of image processing, particularly in energy-constrained mobile environments, has motivated many efforts to compile domain-specific languages to image processing accelerators [13, 20, 21, 39, 44]. A number of these efforts [21, 44], as well as much prior work focusing on signal processing systems [17, 38], utilize the synchronous dataflow

(SDF) [29] and cyclo-static dataflow (CSDF) [7] models as mechanisms to match the throughput of processing nodes in a dataflow graph. SDF requires nodes to define how they produce and consume stream tokens so it can solve for firing rates that yield equal throughput. By constraining programs to DAGs, and encoding the number of tokens (and their arrival rates) in types, Aetherling reduces the problem of SDF throughput matching to the scheduler in Section 6 that simply matches types.

Other systems have taken approaches similar to Aetherling for encoding space-time properties. Rigel [21] uses both SDF and types that encode parallelism. Dahlia [34] uses affine types to produce predictable HLS designs by restricting memory access patterns. Both Cλash [5, 45] and Lift [28, 43] present data-parallel operators like those in Section 4 whose types can encode throughput. Aetherling extends these types to also encode the ordering of valids and invalids. The `TSeq` clock cycle ordering can be viewed as a simple version of the Signal [32] clock calculus and related work like LUSTRE [19]. A limitation of our approach is that, unlike the clock calculus, we cannot express some patterns of valid and invalid clocks. For example, the L^{st} types cannot encode a sequence of two `Int`s on one clock cycle and then one `Int` on the next. Future work on expanding the space of representable patterns may allow the scheduler to search for even more efficient throughput-area trade-offs.

Spatial [26] and HLS [1, 10, 30] are other approaches for compiling high level descriptions of algorithms to hardware with different throughput-area trade-offs. Programs written for both systems are expressed in languages where the semantics do not specify cycle-accurate hardware execution.

Both Spatial [27, 31] and HLS, when using systems like Aladdin [41], enable design space exploration that tunes parameters to trade-off throughput and resource utilization. Through this exploration process, Spatial and HLS allow the developer to search different types of trade-offs compared to Aetherling. For example, Spatial required metaprogramming in Section 8 because its design space exploration does not fundamentally change the circuit architecture to meet a specified throughput. Unlike Aetherling, Spatial enables exploring different communication patterns with off-accelerator memories that can't be described by `TSeq` and `SSeq`.

The Halide image processing language [40] was designed to enable rapid design space exploration for dense tensor applications such as image processing. Halide's solution is to define a separate scheduling language, where programs in this second scheduling language describe how to rewrite the Halide application DAG to explore program optimizations. While Halide scheduling has generally been performed manually by human programmers, search-based techniques for automatically generating schedules for Halide programs have now surpassed expert programmer performance [2]. Since our own approach to Aetherling scheduling (in Section 6) is structured as a sequence of choices about which rewrite rule to apply, it is likely that Halide's fast tree-search-based approach to automatic scheduling could be applied to scheduling Aetherling programs as well.

Although it is possible to reinterpret Halide's scheduling primitives as hardware design directives [39], Halide's primitives and compiler internals are CPU-centric and not intended for hardware design space exploration. We are interested in exploring if Aetherling's representations might serve as an alternative intermediate representation for Halide.

Type-directed program synthesis systems such as Synquid [37] and Myth [16, 35] demonstrate other techniques for synthesizing programs that satisfy a type constraint. These systems create software programs whose semantics satisfy the constraints of a target refinement type. Future work using their techniques could improve upon the scheduler's subroutine that finds the minimum area L^{st} program with a target output type and semantics specified by a L^{seq} program.

10 Conclusion

In this paper, we presented Aetherling, a type-directed approach to compiling data-parallel programs to statically scheduled, streaming hardware designs. Aetherling can produce a diverse set of hardware designs by changing only a single input parameter (desired throughput) to the scheduler. Due to the efficiency of statically scheduled module interfaces, when synthesizing FPGA designs for simple image processing benchmarks, Aetherling's designs require 1.8-7.9× fewer slices than those of prior hardware generation systems.

Going forward, future development of Aetherling will pursue extensions that increase application scope. We aim to support additional image processing and machine learning applications by permitting limited data-dependent memory accesses, data reuse, and stream blocking. Additionally, we will enlarge the space of designs that the scheduler can efficiently search. We also aim to formally prove the soundness of Aetherling's transformations to have stronger guarantees for a larger part of the workflow.

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) Graduate Research Fellowship Program under Grant No. (DGE-1656518), the NSF under Grant No. (CCF-1846502), the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency under agreements numbered FA8750-17-2-0095, FA8650-18-2-7861, a Stanford Graduate Fellowship, affiliates of the Stanford DAWN project and Stanford AHA Agile Hardware Center, and the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

References

[1] 2019. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html [Online; accessed 26-Mar-2020].

[2] 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. https://doi.org/10.1145/3306346.3322967

[3] Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming*. ACM, 157–168.

[4] Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. ACM, 431–444.

[5] C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. Ph.D. Dissertation. University of Twente, Netherlands. https://doi.org/10.3990/1.9789036538039 eemcs-eprint-23939.

[6] 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 *DAC Design Automation Conference 2012*. IEEE, 1212–1221.

[7] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. 1996. Cycle-static dataflow. *IEEE Transactions on signal processing* 44, 2 (1996), 397–408.

[8] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 174–184.

[9] Guy E. Blelloch. 1993. *NESL: A Nested Data-Parallel Language (Version 2.6)*. Technical Report. Pittsburgh, PA, USA.

[10] 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 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 33–36.

[11] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming* (Nice, France) *(DAMP '07)*. ACM, New York, NY, USA, 10–18. https://doi.org/10.1145/1248648.1248652

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[13] 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* (Haifa, Israel) *(PACT '16)*. ACM, New York, NY, USA, 327–338. https://doi.org/10.1145/2967938.2967969

[14] Richard A Eisenberg and Stephanie Weirich. 2013. Dependently typed programming with singletons. *ACM SIGPLAN Notices* 47, 12 (2013), 117–130.

[15] Conal Elliott. 2017. Generic functional parallel algorithms: Scan and FFT. *Proc. ACM Program. Lang.* 1, ICFP, Article 48 (Sept. 2017), 24 pages. https://doi.org/10.1145/3110251

[16] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. *ACM SIGPLAN Notices* 51, 1 (2016), 802–815.

[17] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. 2002. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, Vol. 36. ACM, 291–303.

[18] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 100–112.

[19] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

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

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

[22] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[23] Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In *Principles of Programming Languages*. ACM, 133–146. https://doi.org/10.1145/1926385.1926402

[24] Sang Ho Kim and Jan P Allebach. 2005. Optimal unsharp mask for image sharpening and noise removal. *Journal of Electronic Imaging* 14, 2 (2005), 023005.

[25] Thaddeus Koehn and Peter Athanas. 2016. Arbitrary streaming permutations with minimum memory and latency. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.

[26] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *ACM Sigplan Notices*, Vol. 53. ACM, 296–311.

[27] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. Ieee, 115–127.

[28] 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*. ACM, 35–45.

[29] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[30] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. 2012. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems* 16, 3 (2012), 31–51.

[31] Luigi Nardi, David Koeplinger, and Kunle Olukotun. 2019. Practical design space exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 347–358.

[32] Mirabelle Nebut. 2004. An overview of the Signal clock calculus. *Electronic Notes in Theoretical Computer Science* 88 (2004), 39–54.

[33] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *International Conference on Functional Programming*. ACM, 103–116.

[34] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Ted Bauer, Yuwei Yi, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine types. *Proceedings of the 41st ACM SIGPLAN Conference on Programming*

Language Design and Implementation (2020), to appear.

[35] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

[36] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Jan. 2019), 36 pages.

[37] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[38] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org. http://ptolemy.org/books/Systems

[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 Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 26.

[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. In *Acm Sigplan Notices*, Vol. 48. ACM, 519–530.

[41] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 97–108.

[42] Wilson Snyder and Jean-Philippe Lang. 2019. *Intro - Verilator - Veripool.* https://www.veripool.org/projects/verilator/wiki/Intro

[43] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices* 50, 9 (2015), 205–217.

[44] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. 2018. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11, 1 (2018), 7.

[45] Rinse Wester. 2015. *A transformation-based approach to hardware design using higher-order functions.* Ph.D. Dissertation. University of Twente. https://doi.org/10.3990/1.9789036538879

[46] Xilinx, Inc. 2019. *7 Series FPGAs Memory Resources: User Guide* (ug473 (v1.14) ed.). Xilinx, Inc.