Predictable Accelerator Design
with Time-Sensitive Affine Types

Rachit Nigam   Sachille Atapattu   Samuel Thomas   Zhijing Li
Theodore Bauer   Yuwei Ye   Apurva Koti   Adrian Sampson   Zhiru Zhang
Cornell University
USA

Abstract

Field-programmable gate arrays (FPGAs) provide an opportunity to co-design applications with hardware accelerators, yet they remain difficult to program. *High-level synthesis* (HLS) tools promise to raise the level of abstraction by compiling C or C++ to accelerator designs. Repurposing legacy software languages, however, requires complex heuristics to map imperative code onto hardware structures. We find that the black-box heuristics in HLS can be *unpredictable:* changing parameters in the program that should improve performance can counterintuitively yield slower and larger designs. This paper proposes a type system that restricts HLS to programs that can predictably compile to hardware accelerators. The key idea is to model consumable hardware resources with a *time-sensitive affine type system* that prevents simultaneous uses of the same hardware structure. We implement the type system in Dahlia, a language that compiles to HLS C++, and show that it can reduce the size of HLS parameter spaces while accepting Pareto-optimal designs.

CCS Concepts: • **Software and its engineering** → *Constraints.*

Keywords: Affine Type Systems, High-Level Synthesis

1 Introduction

While Moore's law may not be dead yet, its stalled returns for traditional CPUs have sparked renewed interest in specialized hardware accelerators [28], for domains from machine learning [31] to genomics [56]. Reconfigurable hardware—namely, field-programmable gate arrays (FPGAs)—offer some of the benefits of specialization without the cost of custom silicon. FPGAs can accelerate code in domains from databases [11] to networking [2] and have driven vast efficiency improvements in Microsoft's data centers [46, 19].

However, FPGAs are hard to program. The gold-standard programming model for FPGAs is register transfer level (RTL) design in hardware description languages such as Verilog, VHDL, Bluespec, and Chisel [40, 5]. RTL requires digital design expertise: akin to assembly languages for CPUs, RTL is irreplaceable for manual performance tuning, but it is too explicit and verbose for rapid iteration [53].

FPGA vendors offer *high-level synthesis* (HLS) or "C-to-gates" tools [58, 16, 42, 10] that translate annotated subsets of C and C++ to RTL. Repurposing a legacy software languages, however, has drawbacks: the resulting language subset is small and difficult to specify, and minor code edits can cause large swings in hardware efficiency. We find empirically that smoothly changing source-level hints can cause wild variations in accelerator performance. Semantically, *there is no HLS programming language:* there is only the subset of C++ that a particular version of a particular compiler supports.

This paper describes a type system that restricts HLS to programs whose hardware implementation is clear. The goal is *predictable* architecture generation: the hardware implications are observable in the source code, and costly implementation decisions require explicit permission from the programmer. Instead of silently generating bad hardware for difficult input programs, the type system yields errors that help guide the programmer toward a better design. The result is a language that can express a subset of the architectures that HLS can—but it does so predictably.

The central insight is that an affine type system [54] can model the restrictions of hardware implementation. Components in a hardware design are finite and expendable: a subcircuit or a memory can only do one thing at a time, so a program needs to avoid conflicting uses. Previous research has shown how to apply substructural type systems to model classic computational resources such as memory allocations
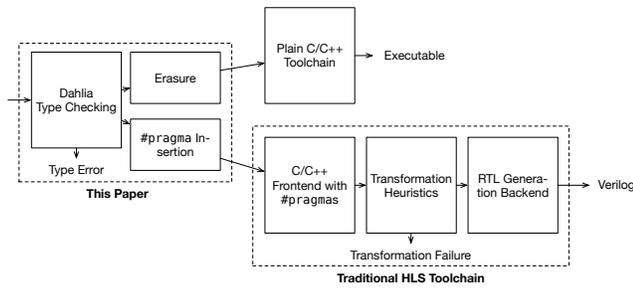
Figure 1. Overview of a traditional high-level synthesis toolchain and how Dahlia layers type safety on top.

and file handles [24, 7, 36, 54] and to enforce exclusion for safe shared-memory parallelism [23, 6, 13]. Unlike those classic resources, however, the availability of hardware components changes with time. We extend affine types with *time sensitivity* to express that repeated uses of the same hardware is safe as long as they are temporally separated.

We describe Dahlia, a programming language for predictable accelerator design. Dahlia differs from traditional HLS in two ways: (1) Dahlia makes the hardware implementation for each language construct manifest in the source code instead of leaving this decision up to the HLS middle-end, and (2) Dahlia uses its *time-sensitive affine types* to reason about the hardware constraints and reject programs that would require complex transformation to implement in hardware. We implement a compiler for Dahlia that emits annotated C++ for a commercial HLS toolchain. We show that predictability pitfalls exist in both industrial and recent academic tools and that Dahlia's reasoning can help alleviate these issues.

The contributions of this paper are:

- We identify predictability pitfalls in HLS and measure their effects in an industrial tool in Section 2.
- We design Dahlia (Section 3), a language that restricts HLS to predictable design spaces by modeling hardware constraints using *time-sensitive affine types*.
- We formalize a time-sensitive affine type system and prove syntactic type soundness in Section 4.
- We empirically demonstrate Dahlia's effectiveness in rejecting unpredictable design points and its ability to make area–performance trade-offs in common accelerator designs in Section 5.

2 Predictability Pitfalls in Traditional HLS

Figure 1 depicts the design of a traditional high-level synthesis (HLS) compiler. A typical HLS tool adopts an existing open-source C/C++ frontend and adds a set of *transformation heuristics* that attempt to map software constructs onto hardware elements along with a backend that generates RTL code [15, 10]. The transformation step typically relies on a constraint solver, such as an LP or SAT solver, to satisfy resource, layout, and timing requirements [25, 17]. Programmers can add #pragma hints to guide the transformation—for example, to duplicate loop bodies or to share functional units.

```
1  int m1[512][512], m2[512][512], prod[512][512];
2  int sum;
3  for (int i = 0; i < 512; i++) {
4    for (int j = 0; j < 512; j++) {
5      sum = 0;
6      for (int k = 0; k < 512; k++) {
7        sum += m1[i][k] * m2[k][j];
8      }
9      prod[i][j] = sum; } }
```

Figure 2. Dense matrix multiplication in HLS-friendly C.

HLS tools are best-effort compilers: they make a heuristic effort to translate *any* valid C/C++ program to RTL, regardless of the consequences for the generated accelerator architecture. Sometimes, the mapping constraints are unsatisfiable, so the compiler selectively ignores some #pragma hints or issues an error. The generated accelerator's efficiency depends on the interaction between the code, the hints, and the transformation heuristics that use them.

The standard approach prioritizes automation over predictability. Small code changes can yield large shifts in the generated architecture. When performance is poor, the compiler provides little guidance about how to improve it. Pruning such *unpredictable* points from the design space would let programmers explore smaller, smoother parameter spaces.

2.1 An Example in HLS

Programming with HLS centers on arrays and loops, which correspond to memory banks and logic blocks. Figure 2 shows the C code for a matrix multiplication kernel. This section imagines the journey of a programmer attempting to use HLS to generate a fast FPGA-based accelerator from this code. We use Xilinx's SDAccel [57] compiler (v2018.3.op) and target an UltraScale+ VU9P FGPA on an AWS F1 instance [1] to perform the experiments in this section.

Initial accelerator. Our imaginary programmer might first try compiling the code verbatim. The HLS tool maps the arrays m1, m2, and prod onto on-chip memories. FPGAs have SRAM arrays, called *block RAMs* (BRAMs), that the compiler allocates for this purpose. The loop body becomes combinational logic consisting of a multiplier, an adder, and an accumulator register. Figure 3a depicts this configuration.

This design, while functional, does not harness any parallelism that an FPGA can offer. The two key metrics for evaluating an accelerator design are performance and area, i.e., the amount of physical chip resources that the accelerator occupies. This initial configuration computes the matrix product in 841.1 ms and occupies 2,355 of the device's lookup tables (LUTs). However, the target FPGA device has over 1 million LUTs, so the programmer's next job is to expend more of the FPGA area to improve performance.

Loop unrolling. The standard tool that HLS offers for expressing parallelism is an UNROLL annotation, which duplicates the logic for a loop body. A programmer might attempt
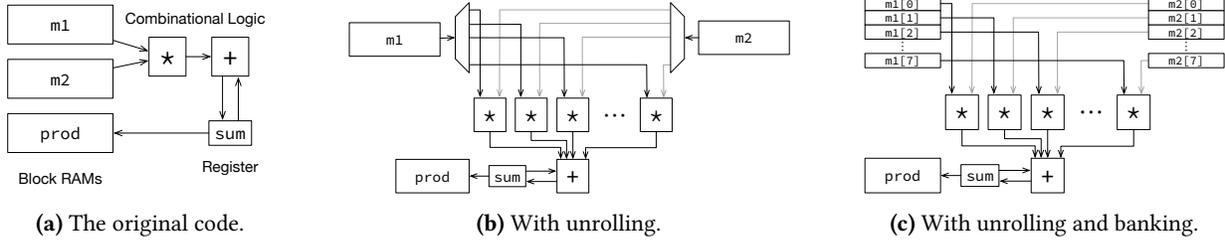
(a) The original code.   (b) With unrolling.   (c) With unrolling and banking.

Figure 3. Three accelerator implementations of the matrix multiplication in Figure 2.

(a) Unrolling without partitioning.   (b) Unrolling with 8-way partitioning.   (c) Unrolling and banking in lockstep.
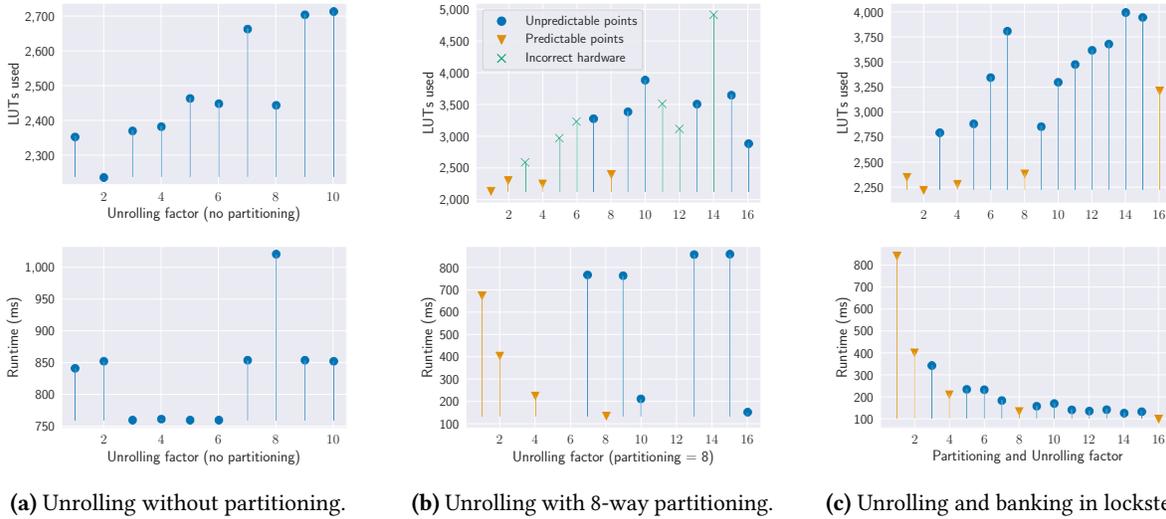
Figure 4. Look-up table count (top) and execution latency (bottom) for the kernel in Figure 2 with varying parameters.

to obtain a better accelerator design by adding this annotation to the innermost loop on lines 6–8 in Figure 2:

```
#pragma HLS UNROLL FACTOR=8
```

This unrolling directive instructs the HLS tool to create 8 copies of the multiplier and adder, called *processing elements* (PEs), and attempt to run them in parallel. Loop unrolling represents an area–performance trade-off: programmers can reasonably expect greater unrolling factors to consume more of the FPGA chip but yield lower-latency execution.

The UNROLL directive alone, however, fails to achieve this objective. Figure 4a shows the effect of various unrolling factors on this code in area (LUT count) and performance (latency). There is no clear trend: greater unrolling yields unpredictably better and worse designs. The problem is that the accelerator's memories now bottleneck the parallelism provided by the PEs. The BRAMs in an FPGA have a fixed, small number of *ports*, so they can only service one or two reads or writes at a time. So while the HLS tool obeys the programmer's UNROLL request to duplicate PEs, its scheduling must serialize their execution. Figure 3b shows how the HLS tool must insert additional *multiplexing* hardware to connect the multipliers to the single-ported memories. The additional hardware and the lack of parallelism yields the unpredictable performance and area for different PE counts.

Memory banking to match parallelism. To achieve expected speedups from parallelism, accelerators need to use multiple memories. HLS tools provide annotations to *partition* arrays, allocating multiple BRAMs and increasing the access throughput. The programmer can insert these partitioning annotations to allocate 8 BRAMs per input memory:

```
#pragma HLS ARRAY_PARTITION VARIABLE=m1 FACTOR=8
#pragma HLS ARRAY_PARTITION VARIABLE=m2 FACTOR=8
```

Banking uses several physical memories, each of which stores a subset of the array's data. The compiler partitions the array using a "round-robin" policy to enable parallel access. In this example, elements 0 and 8 go in bank 0, elements 1 and 9 go in bank 1, etc.:

(Each shade represents a different memory bank.) Figure 3c shows the resulting architecture, which requires no multiplexing and allows memory parallel access.

Combining banking and unrolling, however, unearths another source of unpredictable performance. While the HLS tool produces a good result when both the banking factors and the loop unrolling factor are 8, other design choices perform worse. Figure 4b shows the effect of varying the unrolling factor while keeping the arrays partitioned with factor 8. Again, the area and performance varies unpredictably

with the unrolling factor. Reducing the unrolling factor from 9 to 8 can counter-intuitively *improve* both performance and area. In our experiments, some unrolling factors yield hardware that produces incorrect results. (We show the area but omit the running time for these configurations.)

The problem is that some partitioning/unrolling combinations yield much simpler hardware than others. When both the unrolling and the banking factors are 8, each parallel PE need only access a single bank, as in Figure 3c. The first PE needs to access elements 0, 8, 16, and so on—and because the array elements are "striped" across the banks, all of these values live in the first bank. With unrolling factor 9, however, the first PE needs to access values from *every* bank, which requires complicated memory indirection hardware. With unrolling factor 4, the indirection cost is smaller—the first PE needs to access only bank 0 and bank 4.

From the programmer's perspective, the HLS compiler silently enforces an unwritten rule: *When the unrolling factor divides the banking factor, the area is good and parallelism predictably improves performance. Otherwise, all bets are off.* Figure 4b labels the points where the unrolling factor divides the banking factor as *predictable points*. The HLS compiler emits no errors or warnings for any parameter setting.

Banking vs. array size. Even if we imagine that a programmer carefully ensures that banking factors exactly match unrolling factors, another pitfall awaits them when choosing the amount of parallelism. Figure 4c shows the effects of varying the banking and unrolling factor in our kernel *together*. The LUT count again varies wildly.

The problem is that, when the banking and unrolling factors do not evenly divide the sizes of the arrays involved, the accelerator needs extra hardware to cope with the "leftover" elements. The memory banks are unevenly sized, and the PEs need extra hardware to selectively disable themselves on the final iteration to avoid out-of-bounds memory accesses.

Again, there is a predictable subset of design points when the programmer obeys the unwritten rule: *An array's banking factor should divide the array size.* Figure 4c highlights the predictable points that follow this rule. Among this subset, the performance reliably improves with increasing parallelism and the area cost scales proportionally.

2.2 Enforcing the Unwritten Rules

The underlying problem in each of these sources of unpredictability is that the traditional HLS tool prioritizes automation over programmer control. While automation can seem convenient, mapping heuristics give rise to implicit rules that, when violated, silently produce bad hardware instead of reporting a useful error.

This paper instead prioritizes the predictability of hardware generation and making architectural decisions obvious

in the source code. HLS tools *already* contain such a predictable subset hidden within their unrestricted input language. By modeling resource constraints, we can separate out this well-behaved fragment. Figure 1 shows how our checker augments a traditional HLS toolchain by lifting hidden compiler reasoning into the source code and rejecting potentially unpredictable programs.

The challenge, however, is that the "unwritten rules" of HLS are never explicitly encoded anywhere—they arise implicitly from non-local interactions between program structure, hints, and heuristics. A naïve syntactic enforcement strategy would be too conservative—it would struggle to allow flexible, fine-grained sharing of hardware resources.

We design a type system that models the constraints of hardware implementation to enforce these constraints in a composable, formal way. Our type system addresses *target-independent* issues—it prevents problems that would occur even on an arbitrarily large FPGA. We do not attempt to rule out resource exhaustion problems because they would tie programs to specific target devices. We see that kind of quantitative resource reasoning as important future work.

3 The Dahlia Language

Dahlia's type system enforces a safety property: that the number of simultaneous reads and writes to a given memory bank may not exceed the number of ports. While traditional HLS tools enforce this requirement with scheduling heuristics, Dahlia enforces it at the source level using types.

The key ideas in Dahlia are (1) using substructural typing to reason about consumable hardware resources and (2) expressing time ordering in the language to reason about when resources are available. This section describes these two core features (Sections 3.1 and 3.2) and then shows how Dahlia builds on them to yield a language that is flexible enough to express real programs (Sections 3.3–3.6).

3.1 Affine Memory Types

The foundation of Dahlia's type system is its reasoning about memories. The problem in Section 2.1's example is conflicting simultaneous accesses to the design's memories. The number of reads and writes supported by a memory per cycle is limited by the number of ports in the memory. HLS tools automatically detect potential read/write conflicts and schedule accesses across clock cycles to avoid errors. Dahlia instead makes this reasoning about conflicts explicit by enforcing an affine restriction on memories.

Memories are defined by giving their type and size:

```
let A: float[10];
```

The type of A is mem float[10], denoting a single-ported memory that holds 10 floating-point values. Each Dahlia memory corresponds to an on-chip BRAM in the FPGA. Memories resemble C or Java arrays: programs read and mutate the contents via subscripting, as in A[5] := 4.2.

Because they represent static physical resources in the generated hardware, memory types differ from plain value types like float by preventing duplication and aliasing:

```
let x = A[0];   // OK: x is a float.
let B = A;      // Error: cannot copy memories.
```

The affine restriction on memories disallows reads and writes to a memory that might occur at the same time:

```
let x = A[0];   // OK
A[1] := 1;      // Error: Previous read consumed A.
```

While type-checking A, the Dahlia compiler removes A from the typing context. Subsequent uses of A are errors, with one exception: identical reads to the same memory location are allowed. This program is valid, for example:

```
let x = A[0];
let y = A[0];   // OK: Reading the same address.
```

The type system uses access capabilities to check reads and writes [18, 22]. A read expression such as A[0] acquires a non-affine *read capability* for index 0 in the current scope, which permits unlimited reads to the same location but prevents the acquisition of other capabilities for A. The generated hardware reads once from A and distributes the result to both variables x and y, as in this equivalent code:

```
let tmp = A[0];   let x = tmp;   let y = tmp;
```

However, memory writes use affine *write capabilities*, which are use-once resources: multiple simultaneous writes to the same memory location remain illegal.

3.2 Ordered and Unordered Composition

A key HLS optimization is parallelizing execution of independent code. This optimization lets HLS compilers parallelize and reorder dependency-free statements connected by ; when the hardware constraints allow it—critically, when they do not need to access the same memory banks.

Dahlia makes these parallelism opportunities explicit by distinguishing between *ordered* and *unordered* composition. The C-style ; connector is unordered: the compiler is free to reorder and parallelize the statements on either side while respecting their data dependencies. A second connector, ---, is ordered: in A --- B, statement A must execute before B.

Dahlia prevents resource conflicts in unordered composition but allows two statements in ordered composition to use the same resources. For example, Dahlia accepts this program that would be illegal when joined by the ; connector:

```
let x = A[0]
---
A[1] := 1
```

In the type checker, ordered composition *restores* the affine resources that were consumed in the first command before checking the second command. The capabilities for all memories are discarded, and the program can acquire fresh capabilities to read and write any memory.

Together, ordered and unordered composition can express complex concurrent designs:

```
let A: float[10];  let B: float[10];
{
  let x = A[0] + 1
  ---
  B[1] := A[1] + x  // OK
};
let y = B[0];  // Error: B already consumed.
```

The statements composed with --- are ordered with each other but *unordered* with the last line. The read therefore must not conflict with either of the first two statements.

Logical time. From the programmer's perspective, a chain of ordered computations executes over a series of *logical time steps*. Logical time in Dahlia does not directly reflect physical time (i.e., clock cycles). Instead, the HLS backend is responsible for allocating cycles to logical time steps in a way that preserves the ordering of memory accesses. For example, a long logical time step containing an integer division might require multiple clock cycles to complete, and the compiler may optimize away unneeded time steps that do not separate memory accesses. Regardless of optimizations, however, a well-typed Dahlia program requires at least enough ordered composition to ensure that memory accesses do not conflict.

Local variables as wires & registers. Local variables, defined using the let construct, do not share the affine restrictions of memories. Programs can freely read and write to local variables without restriction, and unordered composition respects the dependencies induced by local variables:

```
let x = 0;  x := x + 1;  let y = x;  // All OK
```

In hardware, local variables manifest as wires or registers. The choice depends on the allocation of physical clock cycles: values that persist across clock cycles require registers. Consider this example consisting of two logical time steps:

```
let x = A[0] + 1  ---  B[0] := A[1] + x
```

The compiler must implement the two logical time steps in different clock cycles, so it must use a register to hold x. In the absence of optimizations, registers appear whenever a variable's live range crosses a logical time step boundary. Therefore, programmers can minimize the use of registers by reducing the live ranges of variables or by reducing the amount of sequential composition.

3.3 Memory Banking

As Section 2.1 details, HLS tools can *bank* memories into disjoint components to allow parallel access. Dahlia memory declarations support bank annotations:

```
let A: float[8 bank 4];
```

In a memory type mem $t[n$ bank $m]$, the banking factor m must evenly divide the size n to yield equally-sized banks.

HLS tools, in contrast, allow uneven banking and silently insert additional hardware to account for it (see Section 2.1).

Affine restrictions for banks. Dahlia tracks an affine resource for each memory bank. To physically address a bank, the syntax $M\{b\}[i]$ denotes the ith element of M's bth bank. This program is legal, for example:

```
let A: float[10 bank 2];
A{0}[0] := 1;
A{1}[0] := 2;   // OK: Accessing a different bank.
```

Dahlia also supports logical indexing into banked arrays using the syntax $M[n]$ for literals n. For example, A[1] is equivalent to A{1}[0] above. Because the index is static, the type checker can automatically deduce the bank and offset.

Multi-ported memories. Dahlia also supports reasoning about multi-ported memories. This syntax declares a memory where each bank has two read/write ports:

```
let A: float{2}[10];
```

A memory provides k affine resources per bank where k is the number of ports in a memory. This rule lets multi-ported memories provide multiple read/write capabilities in each logical time step. For example, Dahlia accepts this program:

```
let A: float{2}[10];
let x = A[0];
A[1] := x + 1;
```

Dahlia does not guarantee data-race freedom in the presence of multi-ported memories. Programs are free to write to and read from the same memory location in the same logical time step and should expect the semantics of the underlying memory technology. Extensions to rule out data races would resemble race detection for parallel software [44, 38].

Multi-dimensional banking. Banking generalizes to multi-dimensional arrays. Every dimension can have an independent banking factor. This two-dimensional memory has two banks in each dimension, a total of $2 \times 2 = 4$ banks:

```
let M: float[4 bank 2][4 bank 2];
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The physical and logical memory access syntax similarly generalizes to multiple dimensions. For example, M{3}[0] represents the element logically located at M[1][1].

3.4 Loops and Unrolling

Fine-grained parallelism is an essential optimization in hardware accelerator design. Accelerator designers duplicate a block of logic to trade off area for performance: n copies of the same logic consume n times as much area while offering a theoretical n-way speedup. Dahlia syntactically separates parallelizable *doall* for loops, which must not have any cross-iteration dependencies, from sequential while loops, which

may have dependencies but are not parallelizable. Programmers can mark for loops with an unroll factor to duplicate the loop body logic and run it in parallel:

```
for (let i = 0..10) unroll 2 { f(i) }
```

This loop is equivalent to a sequential one that iterates half as many times and composes two copies of the body in parallel:

```
for (let i = 0..5) {  f(2*i + 0);  f(2*i + 1)  }
```

The doall restriction is important because it allows the compiler to run the two copies of the loop body in parallel using unordered composition. In traditional HLS tools, a loop unrolling annotation such as #pragma HLS unroll is always allowed—even when the loop body makes parallelization difficult or impossible. The toolchain will replicate the loop body and rely on complex analysis and resource scheduling to optimize the unrolled loop body as well as it can.

Resource conflicts in unrolled loops are errors. For example, this loop accesses an unbanked array in parallel:

```
let A: float[10];
for (let i = 0..10) unroll 2 {
  A[i] := compute(i)  // Error: Insufficient banks.
}
```

Unrolled memory accesses. Dahlia uses special *index types* for loop iterators to type-check memory accesses within unrolled loops. Index types generalize integers to encode information about loop unrolling. In this example:

```
for (let i = 0..8) unroll 4 { A[i] }
```

The iterator i gets the type idx{0..4}, indicating that accessing an array at i will consume banks 0, 1, 2, and 3. Type-checking a memory access with i consumes all banks indicated by its index type.

Unrolling and ordered composition. Loop unrolling has a subtle interaction with ordered composition. In a loop body containing ---, like this:

```
let A: float[10 bank 2];
for (let i = 0..10) unroll 2 {
  let x = A[i]
  ---
  f(x, A[0]) }
```

A naive interpretation would use parallel composition to join the loop bodies at the top level:

```
for (let i = 0..5) {
  { let x0 = A[2*i]      ---  f(x0, A[0]) };
  { let x1 = A[2*i + 1]  ---  f(x1, A[0]) } }
```

However, this interpretation is too restrictive. It requires *all* time steps in each loop body to avoid conflicts with all other time steps. This example would be illegal because the access to A[i] in the first time step may conflict with the access to A[0] in the second time step. Instead, Dahlia reasons about

unrolled loops *in lockstep* by parallelizing *within* each logical time step. The loop above is equivalent to:

```
for (let i = 0..5) {
  { let x0 = A[2*i]; let x1 = A[2*i + 1] }
  ---
  { f(x0, A[0]);      f(x1, A[0]) } }
```

The lockstep semantics permits this unrolling because conflicts need only be avoided between unrolled copies of the same logical time step. HLS tools must enforce a similar restriction but leave the choice to black-box heuristics.

Nested unrolling. In nested loops, unrolled iterators can separately access dimensions of a multi-dimensional array. Nested loops also interact with Dahlia's read and write capabilities. In this program:

```
let A: float[8 bank 4][10 bank 5];
for (let i = 0..8) {
  for (let j = 0..10) unroll 5 {
    let x = A[i][0]
    ---
    A[i][0] := j;  // Error: Insufficient write
  } }              //        capabilities.
```

The read to array *A*[*i*][0] can be proved to be safe because after desugaring, the reads turn into:

```
let x0 = A[i][0];  let x1 = A[i][0] ...
```

The access is safe because the first access acquires a read capability for indices i and 0, so the subsequent copies are safe. Architecturally, the code entails a single read *fanned out* to each parallel PE. However, the write desugars to:

```
A[i][0] := j;  A[i][0] := j + 1 ...
```

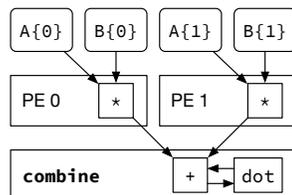which causes a write conflict in the hardware.

3.5 Combine Blocks for Reduction

In traditional HLS, loops can freely include dependent operations, as in this dot product:

```
for (let i = 0..10) unroll 2 { dot += A[i] * B[i] }
```

However, the += update silently introduces a dependency between every iteration which is disallowed by Dahlia's doall for-loops. HLS tools heuristically analyze loops to extract and serialize dependent portions. In Dahlia, programmers explicitly distinguish the non-parallelizable reduction components of for loops. Each for can have an optional combine block that contains sequential code to run after each unrolled iteration group of the main loop body. For example, this loop is legal:

```
for (let i = 0..10)
unroll 2 {
  let v = A[i] * B[i];
} combine {
  dot += v;
}
```


There are two copies of the loop body that run in parallel and feed into a single reduction tree for the combine block.

The type checker gives special treatment to variables like v that are defined in for bodies and used in combine blocks. In the context of the combine block, v is a *combine register*, which is a tuple containing all values produced for v in the unrolled loop bodies. Dahlia defines a class of functions called *reducers* that take a combine register and return a single value (similar to a functional fold). Dahlia defines +=, -=, *=, /= as built-in reducers with infix syntax.

3.6 Memory Views for Flexible Iteration

In order to predictably generate hardware for parallel accesses, Dahlia statically calculates banks accessed by each PE and guarantees that they are distinct. Figure 5a shows the kind of hardware generated by this restriction—each PE is directly connected to a bank.

To enforce this hardware generation, Dahlia only allows simple indexing expressions like A[i] and A[4] and rejects arbitrary index calculations like A[2*i]. General indexing expressions can require complex indirection hardware to allow any PE to access any memory bank. An access like A[i*i], for example, makes it difficult to deduce which bank it would read on which iteration. For simple expressions like A[j+8], however, the bank stride pattern is clear. Traditional HLS tools make a best-effort attempt to deduce access patterns, but subtle changes in the code can unpredictable prevent the analysis and generate bad hardware.

Dahlia uses *memory views* to define access patterns that HLS compilers can compile efficiently and to convince the Dahlia type checker that a parallel access will be predictable. The key idea is to offer different *logical* arrangements of the same underlying physical memory. By logically re-organizing the memory, views can simply reuse Dahlia's type-checking to ensure that complex access patterns are predictable. Furthermore, this allows views to capture the hardware cost of an access pattern in the source code instead of relying on black-box analysis in HLS tools. For Dahlia's HLS C++ backend, views are compiled to direct memory accesses.

The rest of this section describes Dahlia's memory views and their cost in terms of hardware required to transform bank and index values to support the iteration pattern.

Shrink. To directly connect PEs to memory banks, Dahlia requires the unrolling factor to match the banking factor. To allow lower unrolling factors, Dahlia provides *shrink views*, which reduce the banking factors of an underlying memory by an integer factor. For example:

```
let A: float[8 bank 4];
view sh = shrink A[by 2]; // sh: float[8 bank 2]
for (let i = 0..8) unroll 2
  sh[i]; // OK: sh has 2 banks. Compiled to: A[i].
```

The example first defines a view sh with the underlying memory A and divides its banking factor by 2. Dahlia allows
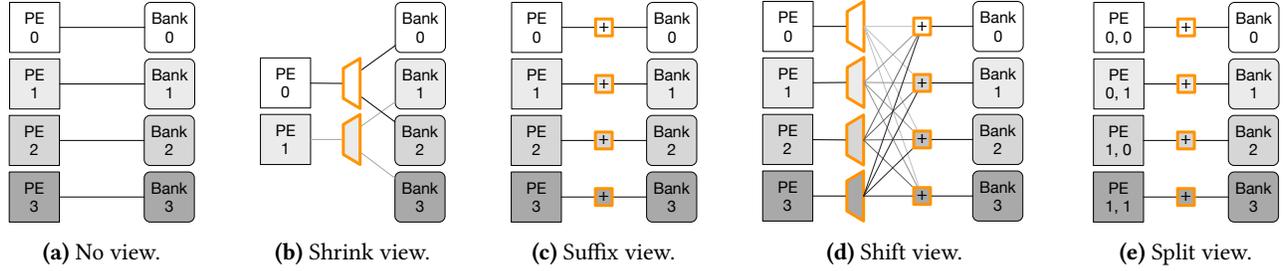
(a) No view.   **(b)** Shrink view.   **(c)** Suffix view.   **(d)** Shift view.   **(e)** Split view.

Figure 5. Hardware schematics for each kind of memory view. Highlighted outlines indicate added hardware cost.

`sh[i]` here because each PE will access a distinct set of banks. The first PE accesses banks 0 and 2; the second accesses banks 1 and 3. The hardware cost of a shrink view, as Figure 5b illustrates, consists of multiplexing to select the right bank on every iteration. The access `sh[i]` compiles to `A[i]`.

Suffix. A second kind of view lets programs create small slices of a larger memory. Dahlia distinguishes between suffixes that it can implement efficiently and costlier ones. An efficient *aligned suffix* view uses this syntax:

```
view v = suffix M[by k * e];
```

where view `v` starts at element $k \times e$ of the memory `M`. Critically, k must be the banking factor of `M`. This restriction allows Dahlia to prove that each logical bank in the view maps to the same physical bank while the indices are offset by the indexing expression. The hardware cost of a suffix view is the *address adapter* for each bank. A view access `v{b}[i]` is compiled to `M{b}[e + i]`.

For example, generating suffixes in a loop results in this pattern, where the digits in each cell are the indices, the shades represent the banks, and the highlighted outline indicates the view:

```
let A: float[8 bank 2];
for (let i = 0..4) {
  view s = suffix A[by 2*i];
  s[1]; // reads A[2*i + 1]
}
```


A suffix view defined using `view v = suffix M[by k*e]` and accessed using `v[i]` is compiled to `M[k*e + i]`.

Shift. Shifted suffixes are like standard suffixes but allow unrestricted offset expressions:

```
view v = shift M[by e];
```

Since `e` is unrestricted, Dahlia assumes that *both* the bank and the indices need to be adapted and that each PE accesses *every* bank. Figure 5d shows the hardware cost of a shift view: each PE is connected to every bank and the index expression is transformed using an address adapter. The distinction between suffix and shift views allows Dahlia to capture the cost of different accessing schemes.

Even in this worst-case scenario, Dahlia can reason about the disjointness of bank accesses. This loop is legal:

```
let A: float[12 bank 4];
for (let i = 0..3) {
  view r = shift A[by i*i];  // r: float[12 bank 4]
  for (let j = 0..4) unroll 4
    let x = r[j];  // accesses A[i*i + j]
}
```

The view `r` has a memory type, so Dahlia can guarantee that the inner access `r[j]` uses disjoint banks and is therefore safe to parallelize. An access `r[i]` to a view declared with `shift M[by e]` compiles to `M[e + i]`.

Split. Some nested iteration patterns can be parallelized at two levels: globally, over an entire array, and locally, over a smaller window. This pattern arises in blocked computations, such as this dot product loop in C++:

```
float A[12], B[12], sum = 0.0;
for (int i = 0; i < 6; i++)
  for (int j = 0; j < 2; j++)
    sum += A[2*i + j] * B[2*i + j];
```

Both the inner loop and the outer loop represent opportunities for parallelization. However, Dahlia cannot prove this parallelization to be safe:

```
let A, B: float[12 bank 4];
view shA, shB = shrink A[by 2], B[by 2];
for (let i = 0..6) unroll 2 {
  view vA, vB = suffix shA[by 2*i], shB[by 2*i];
  for (let j = 0..2) unroll 2 {
    let v = vA[j] + vB[j];
  } combine {
    sum += v; }}
```

While Dahlia can prove that the inner accesses into the views can be predictably parallelized, it cannot establish the disjointness of the parallel copies of the views `va` and `vb` created by the outer unrolled loop.

Split views allow for this reasoning. The key idea is to create logically *more* dimensions than the physical memory and reusing Dahlia's reasoning for multidimensional memories to prove safety for such parallel accesses. A *split view* transforms a one-dimensional memory (left) into a two-dimensional memory (right):

$x \in$ variables $a \in$ memories $n \in$ numbers

$b ::= \textbf{true} \mid \textbf{false}$ $v ::= n \mid b$

$e ::= v \mid \textbf{bop } e_1\, e_2 \mid x \mid a[e]$

$c ::= e \mid \textbf{let } x = e \mid c_1 \,\text{—}\, c_2 \mid c_1 \,;\, c_2 \mid \textbf{if } x\ c_1\ c_2 \mid$
$\qquad \textbf{while } x\ c \mid x := e \mid a[e_1] := e_2 \mid \textbf{skip}$

$\tau ::= \textbf{bit}\langle n \rangle \mid \textbf{float} \mid \textbf{bool} \mid \textbf{mem } \tau[n_1]$

Figure 6. Abstract syntax for the Filament core language.

0	1	2	3	4	5	6	7	8	9	10	11

0	1	5	6	8	9
2	3	6	7	10	11

Using these split-view declarations:

```
view split_A = split A[by 2];
view split_B = split B[by 2];
```

Each view has type `mem float[2 bank 2][6 bank 2]`. A row in the logical view represents a "window" for computation. The above example can now unroll both loops, by changing the inner access to:

```
let v = split_A[j][i] * split_B[j][i];
```

As Figure 5e illustrates, split views have similar cost to aligned suffix views: they require no bank indirection hardware because the bank index is always known statically. They require an address adapter to compute the address within the bank from the separate coordinates. A split view declared `view sp = split M[by k]` on a memory `M` with k banks translates the access `sp[i][j]` to `M{bank}[idx]` where:

$$\text{bank} = i * k + (j \bmod b) \quad \text{idx} = \left\lfloor \frac{j}{b} \right\rfloor$$

4 Formalism

This section formalizes the time-sensitive affine type system that underlies Dahlia in a core language, Filament. We give both a large-step semantics, which is more intelligible, and a small-step semantics, which enables a soundness proof.

4.1 Syntax

Figure 6 lists the grammar for Filament. Filament statements c resemble a typical imperative language: there are expressions, variable declarations, conditions, and simple sequential iteration via `while`. Filament has ordered composition $c_1 \,\text{—}\, c_2$ and unordered composition $c_1 \,;\, c_2$. It separates memories a and variables x into separate syntactic categories. Filament programs can only declare the latter: a program runs with a fixed set of available memories.

4.2 Large-Step Semantics

Filament's large-step operational semantics is a *checked semantics* that enforces Dahlia's safety condition by explicitly tracking and getting stuck when it would otherwise require two conflicting accesses. Our type system (Section 4.3) aims to rule out these conflicts.

The semantics uses an environment σ mapping variable and memory names to values, which may be primitive values or memories, which in turn map indices to primitive values. A second context, ρ, is the set of the memories that the program has accessed. ρ starts empty and accumulates memories as the program reads and writes them.

The operational semantics consists of an expression judgment $\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, v$ and a command judgment $\sigma_1, \rho_1, c \Downarrow \sigma_2, \rho_2$. We describe some relevant rules here, and the supplementary material lists the full semantics and proof [39].

Memory accesses. Memories in Filament are mutable stores of values. Banked memories in Dahlia can be built up using these simpler memories. The rule for a memory read expression $a[n]$ requires that a not already be present in ρ, which would indicate that the memory was previously consumed:

$$\frac{a \notin \rho_1 \qquad \sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, n \qquad \sigma_2(a)(n) = v}{\sigma_1, \rho_1, a[e] \Downarrow \sigma_2, \rho_2 \cup \{a\}, v}$$

Composition. Unordered composition accumulates the resource demands of two commands by threading ρ through:

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_2, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \,;\, c_2 \Downarrow \sigma_3, \rho_3}$$

If both commands read or write the same memory, they will conflict in ρ. Ordered composition runs each command in the same initial ρ environment and merges the resulting ρ:

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_1, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \,\text{—}\, c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3}$$

4.3 Type System

The typing judgments have the form $\Gamma_1, \Delta_1 \vdash c \dashv \Gamma_2, \Delta_2$ and $\Gamma, \Delta_1 \vdash e : \tau \dashv \Delta_2$. Γ is a standard typing context for variables and Δ is the affine context for memories.

Affine memory accesses. Memories are affine resources. The rules for reads and writes check the type of the index in Γ and remove the memory from Δ:

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \textbf{bit}\langle n \rangle \dashv \Delta_2 \qquad \Delta_2 = \Delta_3 \cup \{a \mapsto \textbf{mem } \tau[n_1]\}}{\Gamma, \Delta_1 \vdash a[e] : \tau \dashv \Delta_3}$$

Composition. The unordered composition rule checks the first statement in the initial contexts and uses the resulting contexts to check the second statement:

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_2 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \,;\, c_2 \dashv \Gamma_3, \Delta_3}$$

Ordered composition checks both commands under the same resource set, Δ_1, but threads the non-affine context through:

$$\frac{\Gamma_1, \Delta_1 \vdash c_1 \dashv \Gamma_2, \Delta_2 \qquad \Gamma_2, \Delta_1 \vdash c_2 \dashv \Gamma_3, \Delta_3}{\Gamma_1, \Delta_1 \vdash c_1 \,\text{—}\, c_2 \dashv \Gamma_3, \Delta_2 \cap \Delta_3}$$

The rule merges the resulting Δ contexts with set intersection to yield the resources not consumed by either statement.

4.4 Small-Step Semantics

We also define a small-step operational semantics for Filament upon which we build a proof of soundness. We claim that the small-step semantics, when iterated to a value, is equivalent to the big-step semantics. The semantics consists of judgments $\sigma_1, \rho_1, e \rightarrow \sigma_2, \rho_2, e'$ and $\sigma_1, \rho_1, c \rightarrow \sigma_2, \rho_2, c'$ where σ and ρ are the environment and the memory context respectively. The main challenge is sequential composition, which uses an intermediate command form $c_1 \overset{\rho}{\sim} c_2$ to thread ρ to c_1 and c_2. The supplementary material has full details.

4.5 Desugaring Surface Constructs

Filament desugars surface language features present in Dahlia.

Memory banking. A banked memory declaration like this:

```
let A: float[m bank n];
```

desugars into several unbanked memories:

```
let A_0: float[m/n]; let A_1: float[m/n]; ...
```

Desugaring transforms reads and writes of banked memories to conditional statements that use the indexing expression to decide which bank to access.

Loop unrolling. Desugaring of `for` loops uses the technique described in Section 3.4, translating from:

```
for (let i = 0 .. m) unroll k { c_1 --- c_2 ... }
```

into a `while` loop that duplicates the body:

```
let i = 0;
while (i < m/k) {
  { c_1[i ↦ k*i+0]; c_1[i ↦ k*i+1] ... }
  ---
  { c_2[i ↦ k*i+0]; c_2[i ↦ k*i+1] ... }
  ...
  i++;  }
```

where $c[x \mapsto e]$ denotes substitution.

Memory views. For views' operational semantics, a desugaring based on the mathematical descriptions in Section 3.6 suffices. To type-check them, however, would require tracking the underlying memory for each view (transitively, to cope with views of views) and type-level reasoning about the bank requirements of an access pattern. Formal treatment of these types would require an extension to Filament.

Multi-ported memories. Reasoning about memory ports requires quantitative resource tracking, as in bounded linear logic [21]. We leave such an extension of Filament's affine type system as future work.

4.6 Soundness Theorem

We state a soundness theorem for Filament's type system with respect to its checked small-step operational semantics.

Theorem. *If* $\emptyset, \Delta^* \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\emptyset, \emptyset, c \overset{*}{\rightarrow} \sigma, \rho, c'$ *and* $\sigma, \rho, c' \nrightarrow$, *then* $c' = \mathbf{skip}$.

where Δ^* is the initial affine context of memories available to a program. The theorem states that a well-typed program never gets stuck due to memory conflicts in ρ. We prove this theorem using progress and preservation lemmas:

Lemma 1 (Progress)**.** *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\Gamma, \Delta \sim \sigma, \rho$, *then* $\sigma, \rho, c \rightarrow \sigma', \rho', c'$ *or* $c = \mathbf{skip}$.

Lemma 2 (Preservation)**.** *If* $\Gamma, \Delta \vdash c \dashv \Gamma_2, \Delta_2$ *and* $\sigma, \rho, c \rightarrow \sigma', \rho', c'$, *then* $\Gamma', \Delta' \vdash c' \dashv \Gamma'_2, \Delta'_2$ *and* $\Gamma', \Delta' \sim \sigma', \rho'$.

In these lemmas, $\Gamma, \Delta \sim \sigma, \rho$ is a well-formedness judgment stating that all variables in Γ are in σ and all memories in Δ are not in ρ. Using an extension of the syntax in Figure 6, we prove the lemmas by induction on the small-step relation [39].

5 Evaluation

Our evaluation measures whether Dahlia's restrictions can improve predictability without sacrificing too much sheer performance. We conduct two experiments: (1) We perform an exhaustive design space exploration for one kernel to determine how well the restricted design points compare to the much larger unrestricted parameter space. (2) We port the MachSuite benchmarks [49] and, where Dahlia yields a meaningful design space, perform a parameter sweep.

5.1 Implementation and Experimental Setup

We implemented a Dahlia compiler in 5200 LoC of Scala. The compiler checks Dahlia programs and generates C++ code using Xilinx Vivado HLS's `#pragma` directives [58]. We execute benchmarks on AWS F1 instances [1] with 8 vCPUs, 122 GB of main memory, and a Xilinx UltraScale+ VU9P. We use the SDAccel development environment [57] and synthesize the benchmarks with a target clock period of 250 MHz.

5.2 Case Study: Unrestricted DSE vs. Dahlia

In this section, we conduct an exhaustive design-space exploration (DSE) of a single benchmark as a case study. Without Dahlia, the HLS design space is extremely large—we study how the smaller Dahlia-restricted design space compares. We select a blocked matrix multiplication kernel (gemm-blocked from MachSuite) for its large but tractable design space. The kernel has 3 two-dimensional arrays (two operands and the output product) and 5 nested loops, of which the inner 3 are parallelizable. We define parameters for the 6 banking factors (two dimensions for each memory) and 3 unrolling factors. (A full code listing appears in the supplementary material [39].) We explore a design space with banking factors of 1–4 and unrolling factors of 1, 2, 4, 6, and 8. This design space consists of 32,000 distinct configurations.

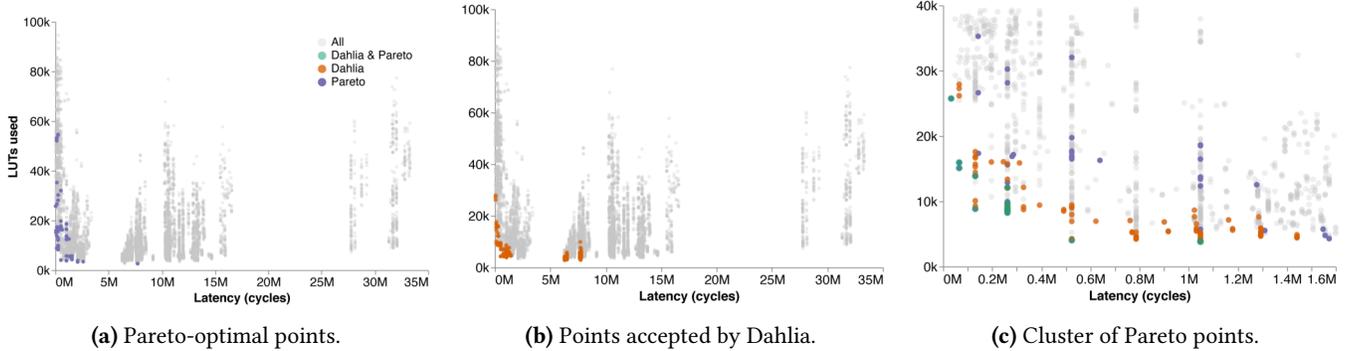We exhaustively evaluated the entire design space using Vivado HLS's estimation mode, which required a total of

(a) Pareto-optimal points.   **(b)** Points accepted by Dahlia.   **(c)** Cluster of Pareto points.

Figure 7. Results from exhaustive design space exploration for gemm-blocked.

2,666 compute hours. We identify Pareto-optimal configurations according to their estimated cycle latency and number of lookup tables (LUTs), flip flops (FFs), block RAMs (BRAMs), and arithmetic units (DSPs).

Dahlia accepts 354 configurations, or about 1.1% of the unrestricted design space. But the smaller space is only valuable if it consists of *useful* design points—a broad range of Pareto-optimal configurations. Figures 7a and 7b show the Pareto-optimal points and the subset of points that Dahlia accepts, respectively. (Pareto optimality is determined using all objectives, but the plot shows only two: LUTs and latency.) Figure 7c shows a zoomed-in view of the tight cluster of Pareto points in the bottom-left of the first two graphs. Dahlia-accepted points lie primarily on the Pareto frontier and allow area-latency trade-offs. The optimal points that Dahlia rejects expend a large number of LUTs to reduce BRAM consumption which, while Pareto optimal, don't seem to be of practical use.

5.3 Dahlia-Directed DSE & Programmability

We port benchmarks from an HLS benchmark suite, Mach-Suite [49], to study Dahlia's flexibility. Of the 19 MachSuite benchmarks, one (backprop) contains a correctness bug and two fail to synthesize correctly in Vivado, indicating a bug in the tools. We successfully ported all 16 of the remaining benchmarks without substantial restructuring.

From these, we select 3 benchmarks that exhibit the kind of fine-grained, loop-level parallelism that Dahlia targets as case studies: sencil2d, md-knn, and md-grid. As the previous section illustrates, an unrestricted DSE is intractable for even modestly sized benchmarks, so we instead measure the breadth and performance of the much smaller space of configurations that Dahlia accepts. For each benchmark, we find all optimization parameters available in the Dahlia port and define a search space. The type checker rejects some design points, and we measure the remaining space. We use Vivado HLS's estimation mode to measure the resource counts and estimated latency for each accepted point. Figure 8 depicts the Pareto-optimal points in each space. In each plot, we also highlight the effect a single parameter has on the results.

The rest of this section reports quantitatively on each benchmark's design space and reports qualitatively on the programming experience during the port from C to Dahlia.

stencil2d. MachSuite's stencil2d is a filter operation with four nested loops. The outer loops scan over the input matrix and the inner loops apply a 3×3 filter. Our Dahlia port unrolls the inner two loops and banks both input memories. We use unrolling factors from 1 to 3 and bank each dimension of the input array by factors 1 to 6. The resulting design space has 2,916 points. Dahlia accepts 18 of these points (0.6%), of which 8 are Pareto-optimal within the set.

Figure 8a shows the Pareto frontier among the Dahlia-accepted points. The figure uses color to show the unrolling factor for the innermost loop. This unrolling factor has a large effect on the design's performance, while banking factors and the other loop explain the rest of the variation.

The original C code uses single-dimensional arrays and uses index arithmetic to treat them as matrices:

```
for (r=0; r<row_size-2; r++)
  for (c=0; c<col_size-2; c++)
    for (k1=0; k1<3; k1++)
      for (k2=0; k2<3; k2++)
        mul = filter[k1*3 + k2] *
              orig[(r+k1)*col_size + c+k2];
```

In the Dahlia port, we must use proper two-dimensional arrays because the compiler rejects arbitrary indexing expressions. Using views, programmers can decouple the storage format from the iteration pattern. To express the accesses to the input matrix orig, we create a shifted suffix view (Section 3.6) for the current window:

```
for (let row = 0..126) {
  for (let col = 0..62) {
    view window = shift orig[by row][by col];
    for (let k1 = 0..3) unroll 3 {
      for (let k2 = 0..3) unroll 3 {
        let mul = filter[k1][k2] * window[k1][k2];
```

The view makes the code's logic more obvious while allowing the Dahlia type checker to allow unrolling on the inner two loops. It also clarifies why parallelizing the outer loops would
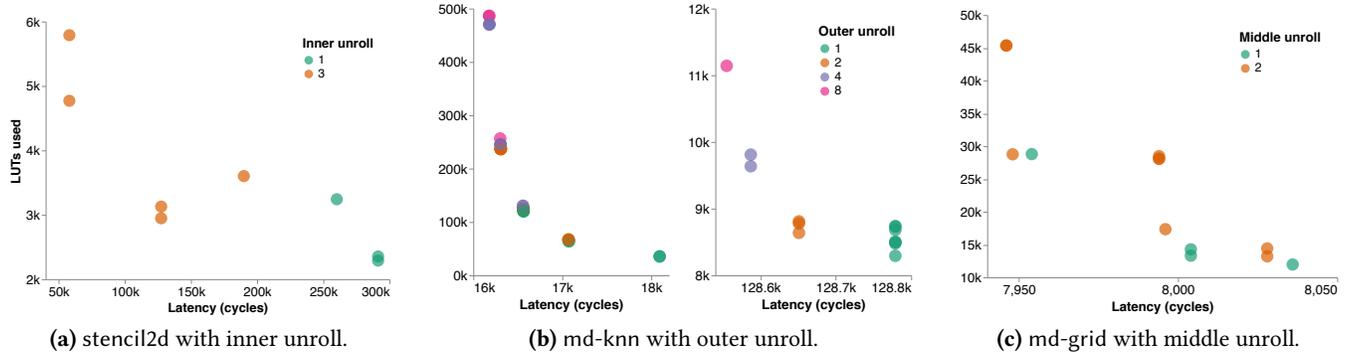
403

(a) stencil2d with inner unroll. **(b)** md-knn with outer unroll. **(c)** md-grid with middle unroll.

Figure 8. The design spaces for three MachSuite benchmarks. Each uses a color to highlight one design parameter.

be undesirable: the parallel views would require overlapping regions of the input array, introducing a bank conflict.

md-knn. The md-knn benchmark implements an *n*-body molecular dynamics simulation with a *k*-nearest neighbors kernel. The MachSuite implementation uses data-dependent loads in its main loop, which naïvely seems to prevent parallelization. In our Dahlia port, however, we hoist this serial section into a separate loop that runs before the main, parallelizable computation. Dahlia's type system helped guide the programmer toward a version of the benchmark where the benefits from parallelization are clear.

For each of the program's four memories, we used banking factors from 1 to 4. We unrolled each of the two nested loops with factors from 1 to 8. The full space has 16,384 points, of which Dahlia accepts 525 (3%). 37 of the Dahlia-accepted points are Pareto-optimal.

Figure 8b shows two Pareto frontiers that Dahlia accepts at different scales. The color shows the unrolling factor of the outer loop. The frontier on the right uses an order of magnitude fewer resources but is an order of magnitude slower. In this kernel, the dominant effect is the memory banking (not shown in the figure), which determines which frontier the designs fall into. The outer unrolling factor (shown in color) affects the two regimes differently: on the right, it allows area–latency trade-offs within the frontier; on the left, it acts as a second-order effect that expends LUTs to achieve a small increase in performance.

md-grid. Another algorithm for the same molecular dynamics problem, md-grid, uses a different strategy based on a 3D grid implemented with several 4-dimensional arrays. It calculates forces between neighboring grid cells. Of its 6 nested loops, the outer three are parallelizable. We use banking factors of 1 to 4 for each dimension of each array, and we try unrolling factors from 1 to 8 for both loops. The full space has 21,952 points, of which Dahlia accepts 81 (0.4%). 13 of the Dahlia-accepted points are Pareto-optimal.

Figure 8c again shows the Pareto-optimal design points. The *innermost* loop unrolling factor (not shown in the figure) determines which of three coarse regimes the design falls

into. The color shows the *second* loop unrolling factor, which determines a second-order area–latency trade-off within each regime. Unrolling enables latency-area trade-offs in both the cases.

6 Future Work

Dahlia represents a first step toward high-level semantics for accelerator design languages. It leaves several avenues for future work on scaling up from kernels to full applications and expressing more hardware implementation techniques.

Modularity. Dahlia's type system relies on a closed-world assumption. A compositional type system would enable reuse of abstract hardware modules without "inlining" them, like functions in a software language. The primary challenge in modular accelerator design is the balance between abstraction and efficiency: a more general module is likely to be less efficient. An abstraction mechanism must also cope with the timing of inter-module interactions: some interfaces are latency-insensitive while others rely on cycle-level timing.

Polymorphism. Dahlia's memory types are monomorphic. Polymorphism would enable abstraction over memories' banking strategies and sizes. A polymorphic Dahlia-like language could rule out invalid combinations of abstract implementation parameters before the designer picks concrete values, which would help constrain the search space for design space exploration.

Pipelining. Pipelined logic is a critical implementation technique for high-level synthesis. Dahlia does not reason about the timing of pipeline stages or their resource conflicts. Extensions to its type system will need to reason about the cycle-level latency of these stages and track the fine-grained sharing of logic resources.

Direct RTL generation. The current Dahlia compiler relies on a commercial C++-based HLS compiler as its backend. It generates directives that instruct the HLS tool to generate hardware according to the program's Dahlia types, but the unpredictability of traditional HLS means that results can still vary. Future compilers for Dahlia-like languages might
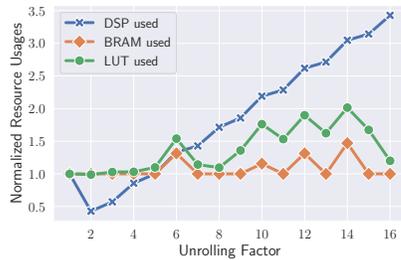
Figure 9. Resource utilization for gemm-ncubed in Spatial normalized to the design without unrolling.

generate RTL directly and rely on the simpler input language avoid the complexity of unrestricted HLS.

7   Related Work

Dahlia builds on a long history of work on safe systems programming. Substructural type systems are known to be a good fit for controlling system resources [7, 24, 54, 13, 36]. Dahlia's enforcement of exclusive memory access resembles work on race-free parallel programming using type and effect systems [8] or concurrent separation logic [41]. Safe parallelism on CPUs focuses on data races where concurrent reads and writes to a memory are unsynchronized. Conflicts in Dahlia are different: *any* simultaneous pair of accesses to the same *bank* is illegal. The distinction influences Dahlia's capability system and its memory views, which cope with the arrangement of arrays into parallel memory banks.

Dahlia takes inspiration from other approaches to improving the accelerator design process, including HDLs, HLS, DSLs, and other recent accelerator design languages.

Spatial. Spatial [32] is a language for designing accelerators that builds on *parallel patterns* [43], which are flexible hardware templates. Spatial adds some automation beyond traditional HLS: it infers a banking strategy given some parallel accesses. Like HLS, Spatial designs can be unpredictable. Figure 9 shows resource usage for the matrix multiplication kernel from Section 2 written in Spatial. (A full experimental setup appears in the supplementary material [39].) For unrolling factors that do not evenly divide the memory size, Spatial will sometimes infer a banking factor that is not equal to the unrolling factor. In these cases, the resource usage abruptly increases. A type system like Dahlia could help address these predictability pitfalls in Spatial.

Better HDLs. Modern hardware description languages [5, 35, 14, 4, 55, 30, 40] aim to address the shortcomings of Verilog and VHDL. These languages target register transfer level (RTL) design. Dahlia targets a different level of abstraction and a different use case: it uses an imperative programming model and focuses exclusively on computational accelerators. Dahlia is not a good language for implementing a CPU, for example. Its focus on acceleration requires the language and semantics to more closely resemble software languages.

Traditional HLS. Existing commercial [58, 29, 37, 9] and academic [47, 10, 42, 59] high-level synthesis (HLS) tools compile subsets of C, C++, OpenCL, or SystemC to RTL. While their powerful heuristics can be effective, when they fail, programmers have little insight into what went wrong or how to fix it [34]. Dahlia represents an alternative approach that prioritizes programmer control over black-box optimization.

Targeting hardware from DSLs. Compilers to FPGAs and ASICs exist for DSLs for image processing [26, 27, 45, 51] and machine learning [20, 52]. Dahlia is not a DSL: it is a general language for implementing accelerators. While DSLs offer advantages in productivity and compilation for individual application domains, they do not obviate the need for general languages to fill in the gaps between popular domains, to offer greater programmer control when appropriate, and to serve as a compilation target for multiple DSLs.

Accelerator design languages. Some recent languages also focus on general accelerator design. HeteroCL [33] uses a Halide-like [48] scheduling language to describe how to map algorithms onto HLS-like hardware optimizations, and T2S [50] similarly lets programs describe how generate a spatial implementation. Lime [3] extends Java to express target-independent streaming accelerators. CoRAM [12] is not a just a language; it extends FPGAs with a programmable memory interface that adapts memory accesses, akin to Dahlia's memory views. Dahlia's focus on predictability and type-driven design makes it unique, as far as we are aware.

8   Conclusion

Dahlia exposes predictability as a new design goal for HLS tools. Predictability comes at a cost—it can rule out design points that perform surprisingly well because of a subtle convergence of heuristics. We see these outliers as a worthy sacrifice in exchange for an intelligible programming model and robust reasoning tools. We hope to extend Dahlia's philosophy to bring predictability to the rest of the reconfigurable hardware system stack, from the language to the LUTs.

Acknowledgments

References

[1] Amazon Web Services. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX Symposium on Networked System Design and Implementation (NSDI)*.

[3] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[4] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco E. T. Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *Euromicro Conference on Digital System Design: Architectures, Methods and Tools*.

[5] 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 *Design Automation Conference (DAC)*.

[6] Henry G. Baker. "Use-once" Variables and Linear Objects: Storage Management, Reflection and Multi-threading. *SIGPLAN Notices* 30, 1 (Jan. 1995), 45–52.

[7] J Bernardy, Mathieu Boespflug, Ryan Newton, Simon L. Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[8] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[9] Cadence. Stratus High-Level Synthesis. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[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 *International Symposium on Field-Programmable Gate Arrays (FPGA)*.

[11] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: big data on little clients. In *International Symposium on Computer Architecture (ISCA)*.

[12] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Field programmable gate arrays (FPGA)*.

[13] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*.

[14] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. 2017. A Pythonic approach for rapid hardware prototyping and instrumentation. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[15] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. 2006. Platform-Based Behavior-Level and System-Level Synthesis. In *International SoC Conference*.

[16] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 30, 4 (April 2011), 473–491.

[17] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Design Automation Conference (DAC)*.

[18] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear regions are all you need. In *European Symposium on Programming (ESOP)*.

[19] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-scale DNN Processor for Real-time AI. In *International Symposium on Computer Architecture (ISCA)*.

[20] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware system synthesis from Domain-Specific Languages. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[21] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. *Theoretical Computer Science* 97, 1 (April 1992), 1–66.

[22] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2013. Rely-guarantee references for refinement types over aliased mutable data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[23] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[24] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based Memory Management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[25] Sumit Gupta, Rajesh Gupta, Nikil D. Dutt, and Alexandru Nicolau. 2004. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer.

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

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

[28] John L. Hennessy and David A. Patterson. A New Golden Age for Computer Architecture. *Communications of the ACM (CACM)* 62, 2 (Jan. 2019), 48–60.

[29] Intel. Intel High Level Synthesis Compiler. https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html

[30] Jane Street. HardCaml: Register Transfer Level Hardware Design in OCaml. https://github.com/janestreet/hardcaml.

[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on*

Computer Architecture (ISCA).

[32] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[33] 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 *International Symposium on Field-Programmable Gate Arrays (FPGA).*

[34] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N Do, and Deming Chen. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering* (2012).

[35] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[36] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *High Integrity Language Technology (HILT).*

[37] Mentor Graphics. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[38] Mayur Naik, Alexander Aiken, and John Whaley. 2006. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[39] 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: Supplemental Material.* Technical Report.

[40] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Conference on Formal Methods and Models for Co-Design (MEMOCODE).*

[41] Peter W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science* 375 (April 2007), 271–307.

[42] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *International Conference on Field-Programmable Logic and Applications (FPL).*

[43] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 651–665.

[44] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. 2006. LOCKSMITH: context-sensitive correlation analysis for race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[45] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).

[46] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck,

Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Y. Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA).*

[47] Andrew R Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. 2008. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *International Symposium on Field-Programmable Gate Arrays (FPGA).*

[48] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[49] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *IEEE International Symposium on Workload Characterization (IISWC).*

[50] Hongbo Rong. 2017. *Programmatic Control of a Compiler for Generating High-performance Spatial Hardware.* Technical Report. arXiv preprint 1711.07606. https://arxiv.org/abs/1711.07606.

[51] Jeff Setter. Halide-to-Hardware. https://github.com/jeffsetter/Halide-to-Hardware.

[52] 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 *IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[53] Stuart Sutherland, Don Mills, and Chris Spear. 2007. Gotcha Again: More Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know. In *Synopsys Users Group (SNUG) San Jose.* https://lcdm-eng.com/papers/snug07_Verilog%20Gotchas%20Part2.pdf

[54] Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).*

[55] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *Summit oN Advances in Programming Languages (SNAPL).*

[56] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A Genomics Co-processor Provides Up to 15,000X Acceleration on Long Read Assembly. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[57] Xilinx Inc. SDAccel: Enabling Hardware-Accelerated Software. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[58] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. UG902 (v2017.2) June 7, 2017. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf.

[59] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. 2008. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis: From Algorithm to Digital Circuit*, Philippe Coussy and Adam Morawiec (Eds.). 99–112.