Predictable Accelerator Design with Time-Sensitive Affine Types

Rachit Nigam
Cornell University
rnigam@cs.cornell.edu

Sachille Atapattu
Cornell University
sa2257@cornell.edu

Theodore Bauer
Cornell University
tjb272@cornell.edu

Adrian Sampson
Cornell University
asampson@cs.cornell.edu

Zhiru Zhang
Cornell University
zhiruz@cornell.edu

Abstract

Interest in reconfigurable hardware accelerators is surging ahead, but the programming models for designing them are lagging behind. High-level synthesis (HLS) compilers repurpose legacy software languages, yielding opaque and *ad hoc* language subsets that translate unpredictably into hardware. We describe Dahlia, a language and type system that makes HLS more productive and predictable. Dahlia uses substructural typing to model consumable hardware resources and incorporates a notion of logical time to model the pervasive concurrency of accelerator designs. Its *time-sensitive affine types* guarantee that a program is free of conflicting uses of memories. Dahlia programs have a standard imperative semantics but make their hardware implementations explicit and predictable. We formalize time-sensitive affine types and state a soundness theorem for a core calculus for HLS languages. We implement Dahlia in a compiler targeting a commercial HLS toolchain and demonstrate that it can express 18 benchmarks from MachSuite. Dahlia's type checking rules out HLS pitfalls that can lead to poor implementations and aids effective design space exploration. Despite its restrictiveness, Dahlia admits hardware implementations that are nearly as efficient as designs from traditional HLS.

1. Introduction

While Moore's law may not be dead yet, its stalled efficiency returns for traditional CPUs have sparked renewed interest in specialized hardware accelerators [28], for domains from machine learning [31] to genomics [51]. Reconfigurable hardware—namely, field-programmable gate arrays (FPGAs)—offer some of the benefits of specialization without the cost of custom silicon. They have driven vast efficiency improvements in datacenters at Microsoft [20, 42].

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 [1], VHDL [2], Bluespec [38], and Chisel [7]. RTL programming requires digital design expertise: akin to assembly programming for CPUs, RTL programming is irreplaceable for manual performance tuning, but it is too explicit and verbose for productive engineering in most cases [48].

FPGA vendors offer *high-level synthesis* (HLS) or "C-to-gates" tools [12, 18, 40, 53] 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 seemingly benign code edits can cause large changes in the generated hardware. A complex set of heuristics in an HLS compiler determines whether and how to map imperative code onto a hardware architecture. 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 makes HLS programming simpler and more predictable. The idea is to define a restricted imperative language that also has well-defined hardware meaning. The goal is *not* to transparently compile arbitrary imperative programs—instead, a type system restricts the set of programs that the compiler can translate. The type system's overarching goal is predictability: for any well-typed program, the path to hardware should be clear.

The central insight is that an affine type system [49] 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 of any given component. Previous research has shown how to apply substructural type systems to model classic computational resources such as memory allocations and file handles [9, 24, 36, 49] and to enforce exclusion for safe shared-memory parallelism [8, 14, 22]. We extend this line of work to model the elements of a hardware

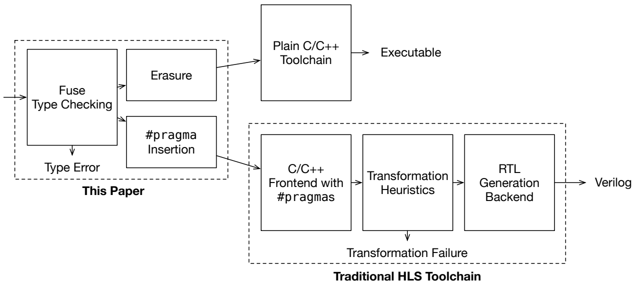


Figure 1: Overview of a traditional high-level synthesis toolchain and how Dahlia layers type safety on top.

design and introduce *time-sensitive affine types* to restrict their simultaneous, parallel use.

We describe Dahlia, a programming language that uses time-sensitive affine types to make HLS-based accelerator design more predictable. Dahlia has two competing goals:

- *Imperative erasure.* Dahlia programs look like ordinary imperative programs, with familiar concepts like variables, loops, and procedural abstraction. Dahlia adds types and other annotations, but a straightforward erasure can produce a standard imperative program that produces the same results.
- *Hardware predictability.* Well-typed Dahlia programs have obvious hardware equivalents. Important architectural decisions are explicit in the program source code, not implicit in the compiler toolchain.

These goals are in tension because they require Dahlia to reconcile two interpretations for every well-typed program: an imperative semantics and a hardware structure. Dahlia's type system restricts the language to programs that meet both goals. Together, these properties help make Dahlia intelligible to experienced software developers without hiding any of the *essential complexity* of hardware design.

This paper describes the design of the Dahlia language and formalizes the core of Dahlia's time-sensitive affine type system and its corresponding operational semantics. We describe a compiler implementation that translates Dahlia to restricted C++ code for the Vivado HLS toolchain [53]. Our empirical evaluation demonstrates that Dahlia is expressive enough to implement the MachSuite HLS benchmarks [45] and its predictability guarantees come at a minimal cost in the efficiency of the generated hardware.

2. Traditional HLS vs. Type Safety in Dahlia

This section illustrates how the design goals of Dahlia differ from a traditional HLS toolchain. The central difference in philosophy is that traditional HLS tools are best-effort compilers: they make a heuristic effort to translate *any* valid C/C++ program to RTL. Dahlia only attempts to compile programs that have a predictable hardware implementation.

Figure 1 depicts the design of a traditional HLS toolchain. 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 [17]. The transformation step typically generates resource and layout constraints and sends them to a constraint solver, such as a linear programming or SAT solver [16, 25]. Programmers can add `#pragma` hints to guide the transformation—for example, to suggest how to map a multiplication operator onto soft or hard logic. The heuristics either succeed in generating a hardware architecture or fail, in which case the programmer needs to adjust the code or hints and try again. There are two problems with this approach. (1) Hints are local, but their effects are global. HLS toolchains can only check simple validity and cannot diagnose complex interdependencies between transformations. (2) Hints cede too much influence to heuristics in the middle end of the compiler, making it difficult for programmers to control implementation decisions.

2.1 An Example in HLS

Programming with HLS centers on arrays and loops, which correspond to memory banks and logic blocks. For example, this simple C loop multiplies a vector by a scalar:

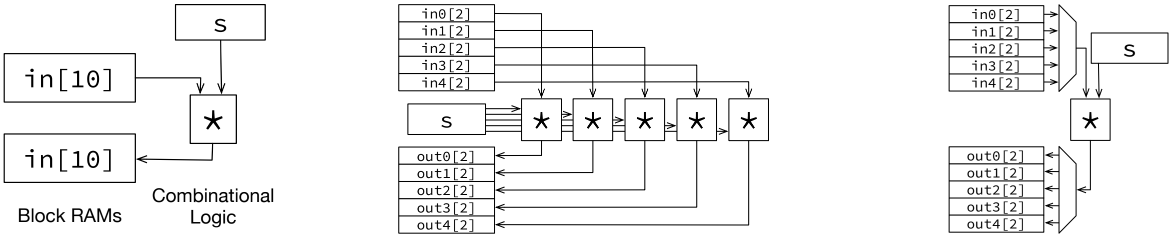
```
void vsmul(float in[10], float out[10], float s) {
  for (int i = 0; i < 10; i++) {
    out[i] = in[i] * s;
  }
}
```

Under an HLS interpretation, the `in` and `out` arrays become on-chip memories. FPGAs have many built-in SRAM arrays, called *block RAMs* (BRAMs), that the HLS compiler allocates for this purpose. The loop body becomes a single logic block consisting of a multiplier with wires to the two BRAMs. Figure 2a depicts this FPGA configuration. The design iteratively invokes the multiplier 10 times over $10n$ clock cycles, where n is the cycle latency of the loop-body logic.

This design, while functional, does not fully exploit the hardware. A standard enhancement is to make several copies of the multiplier logic to run many loop iterations in parallel. To implement common hardware optimizations like this one, HLS tools use C `#pragma` annotations. In Xilinx's popular Vivado HLS compiler [53], for example, the annotation to generate five copies of the loop body is:

```
for (int i = 0; i < 10; i++) {
#pragma HLS UNROLL factor=5
```

However, inserting this annotation alone is insufficient to get the desired speedup. To run multiple copies of the loop body in parallel, the hardware needs to read five values from the `in` memory simultaneously. Typical FPGAs have single- or dual-ported BRAMs that cannot service five simultaneous requests. To get the intended design, an HLS programmer needs to *bank* the arrays, which divides their data among multiple hardware memories. Again using Vivado HLS's annotation syntax, the programmer adds:



(a) The original loop. (b) Five-way loop unrolling and memory banking. (c) Banking without unrolling.

Figure 2: Three architectures for a vector–scalar multiplication accelerator.

```
#pragma HLS ARRAY_PARTITION variable=invec factor=5
#pragma HLS ARRAY_PARTITION variable=outvec factor=5
```

This `#pragma` annotation allocates 5 BRAMs per array, and the program generates hardware as in Figure 2b.

Producing a good accelerator for this program depends on a non-local relationship between an unrolled loop and the memories it accesses. This interdependence is hidden at the source level, so the HLS tool cannot detect or report misalignments between optimization hints.

The problem compounds as programmers iteratively seek better efficiency through design space exploration (DSE) by tuning parameters like banking and unrolling factors. If the programmer tries a design point with banking factor 5 and unrolling factor 1, the compiler will add *additional* hardware to implement the banked accesses, resulting in the inferior design in Figure 2c. This unpredictability means that DSE must explore a complex, discontinuous search space.

2.2 Design Principles for Dahlia

Dahlia's goal is to lift the important hardware implementation decisions out of code hints and ad-hoc checks in the HLS toolchain and explicitly reason about their relationship using a type system. Figure 1 shows how Dahlia augments a traditional HLS toolchain. The Dahlia type checker either certifies the program as safe for predictable HLS or rejects programs that violate its rules. The compiler can emit output in two modes: via *type erasure*, it produces a C++ program for an ordinary CPU toolchain; via *pragma insertion*, it produces annotated code that forces a commercial HLS tool to generate the accelerator architecture dictated by the Dahlia source code.

Dahlia uses three design principles:

Semantic equivalence. A well-typed Dahlia program must produce the same outputs when we compile it for a CPU (via erasure) and for an FPGA (via pragma insertion). Semantic equivalence ensures that programmers can always test and verify correctness using software semantics.

Predictability over automation. Important optimizations that the compiler performs should have explicit representations in the source program. While such a ruthlessly explicit programming model may seem less convenient than the implicit, best-effort optimizations in traditional HLS, Dahlia prioritizes obviousness over convenience. Dahlia makes principled choices to keep some decisions implicit to benefit from high level designs. Register vs. wire distinction local variables and pipelining strategies in Dahlia programs are implicit

Conservative safety. Dahlia's type system is conservative: it must reject every program that does not have a predictable hardware implementation, but it can also reject benign programs (such as programs with complicated parallel memory access patterns). Consequently, some traditional HLS optimizations that rely on subtle global properties will be inexpressible in well-typed Dahlia programs.

3. The Dahlia Language

This section introduces the Dahlia language.

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. Single-ported SRAM hardware can only service a single read or write per cycle. HLS tools automatically detect potential conflicts and schedule accesses across cycles to avoid errors. Dahlia instead makes this reasoning about conflicts explicit by enforcing an affine restriction on memories.

Programs define memories by giving their type and size:

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

The type of `A` is `mem float[10]`. Each Dahlia memory corresponds to an on-chip BRAM in the FPGA. A memory resembles a C or Java array: programs can read and mutate its contents by subscripting, as in `A[5] := 4.2`. Because they represent static physical resources in the generated accelerator design, 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.
```

Dahlia models a single-ported BRAM design that can service one read or write per clock cycle. (State-of-the-art FPGAs can also offer dual-ported BRAMs and FIFO memories [52], but the current Dahlia type system focuses on the general single-ported SRAM case.) The affine restriction on memories disallows reads and writes to a memory at the same time:

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

When it type checks the read to `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 [19, 23]. A read expression such as `A[0]` acquires a *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 *write capabilities*, which are use-once resources: multiple simultaneous writes to the same memory location remain illegal.

3.2 Parallel vs. Sequential Composition

A simple affine treatment of memory types would be too restrictive: it would only allow a single access to each memory. Clearly, Dahlia needs a way to allow multiple accesses to the same memory *when they cannot conflict:* that is, when they are separated in time. To support this temporal reasoning, Dahlia lets programs describe when statements may run in parallel and when they must be sequential.

Dahlia has two forms of statement composition. The C-style `;` connector used in the above examples allows parallel execution: in c_1 `;` c_2, the statements c_1 and c_2 are unordered in time, so Dahlia requires them to be memory conflict free. A second connector, `---`, represents sequential composition: in c_1 `---` c_2, the commands run serially, so their memory accesses may safely overlap. For example, Dahlia accepts this program with conflicting accesses:

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

Sequential 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.

Sequential composition introduces a notion of logical time. From the programmer's perspective, a chain of sequentially composed computations executes over a series of *logical time steps*. Logical time in Dahlia orders computations with respect to each other, but it does not directly reflect physical time—i.e., clock cycles. Instead, the compiler 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 sequencing to ensure that memory accesses are valid and non-conflicting. A minimal allocation of sequential composition therefore forms a lower bound on the number of clock cycles that a computation requires.

Together, parallel and sequential composition can express complex concurrent designs:

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

The statements composed with `---` are ordered with respect to each other but are *unordered* with read from `B`. The read must therefore not conflict with either of the first two statements.

3.3 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 Dahlia semantics respect dependencies via local variables, even in the presence of parallel composition. This program consisting of a single logical time step, for example, produces the same result as in a standard imperative interpretation:

```
let x = 0;  x := x + 1;  let y = x;
```

Regardless of how many clock cycles it requires, the compiler must respect this code's read-after-write dependencies and place the incremented value of `x` into `y`. The `;` connector does not *require* statements to run in parallel in the final hardware implementation—it only *allows* them to do so, up to the constraints of correct data flow.

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
```

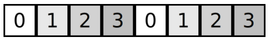
The compiler must implement the two logical time steps in different clock cycles, so it must use a register to hold `x`. In the absence of optimizations, registers appear whenever a variable's live range crosses a logical time step boundary. Therefore, programmers can minimize the use of registers by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

Register allocation and cycle timing is implicit in Dahlia as it is in traditional HLS. This choice represents the hypothesis that managing individual registers is not a first-order concern in most accelerator designs, so sharper safety–convenience trade-offs are best left to other language features.

3.4 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:

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



0	1	2	3	0	1	2	3

A banked memory consists of several physical memories, each of which stores a subset of the array's data. The compiler partitions the array's data cyclically, using a "round-robin" policy. In this example, the first and fourth element go in bank 0, the second and fifth element go in bank 1, and so on. This pattern allows logic to access adjacent values in parallel.

In a memory type `mem` $t[n\ \texttt{bank}\ m]$, the banking factor m must evenly divide the size n to yield equally-sized banks. (Traditional HLS tools, in contrast, allow uneven banking and silently insert additional hardware to account for the complicated access patterns that it induces.)

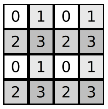
Affine restrictions for banks. Dahlia tracks an affine resource for each memory bank. To physically address a bank, the syntax $M\{b\}[i]$ denotes the ith element of M's bth bank. This program is legal, for example:

```
decl 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-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, for a total of $2 \times 2 = 4$ banks:

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



0	1	0	1
2	3	2	3
0	1	0	1
2	3	2	3

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]`.

3.5 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. In Dahlia, parallelism manifests as `for` loop unrolling. Dahlia's `for` loops are *doall* loops: they must not have cross-iteration dependencies. 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)
}
```

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. In Dahlia, the type system enforces constraints that enable straightforward parallelization without complex code transformations and heuristics. Resource conflicts in unrolled loops are errors. For example, this unrolled loop is illegal because it accesses an unbanked array in parallel:

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

This loop body writes `A[i]` and `A[i+1]` simultaneously, which Dahlia's affine memory restrictions disallow.

From a software perspective, Dahlia's restrictions on `for` loops may seem extreme. However, we find empirically that real-world accelerator designs use simple parallelism patterns that fit within these constraints. For non-parallelizable code, Dahlia also supports a sequential `while` loop (see Section 3.9).

Unrolled memory accesses. To type check memory accesses within unrolled loops, Dahlia uses special *index types* for loop iterators. 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 sequential composition. Loop unrolling has a subtle interaction with sequential composition. In a loop body containing `---`, like this:

```
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 implements 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. Traditional HLS tools must enforce a similar restriction, but the choice is left to the 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:

```
decl 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 is safe because it reuses a single memory access, but the write is unsafe because it would store multiple values into the same location in parallel. Dahlia rejects the write because it would require acquiring a write capability multiple times.

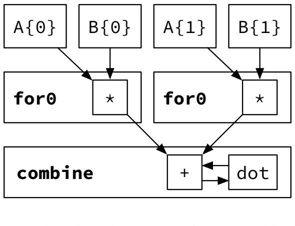
3.6 Combine Blocks for Reduction

While Dahlia's `for` loops prevent cross-iteration dependencies, accelerators often need to reduce the results of parallel loop iterations. 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. The HLS compiler needs to heuristically analyze loops to extract dependent portions and then pattern-match on the accumulation to emit a reduction tree. 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 and generates the hardware (right):

```
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 both values produced for `v` in the unrolled loop bodies. To use the combine register, 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.

3.7 Memory Views for Flexible Iteration

To check memory accesses for conflicts, Dahlia needs to know which memory banks of `A` an indexing expression `A[e]` will require. to track the bank requirement of each indexing expression. While an expression `A[k*i + j]` might be useful for accessing within a window of `A`, for example, deducing the bank requirements of arbitrary indexing arithmetic like this is challenging and likely to be incomplete. Traditional HLS tools use a best-effort approach to analyzing some indexing arithmetic to generate good hardware, but it is difficult to draw a clear line between predictable, efficient indexing and complex, unanalyzable indexing that forces the HLS toolchain to be conservative and serialize the entire loop.

Dahlia instead introduces *memory views:* a small set of index-manipulation combinators that, when composed, can express complex iteration patterns while maintaining type safety. The key idea is that *memory types can represent different logical arrangements of the same underlying physical memory*. A view of a memory also has a memory type, but a different one from the underlying physical memory—and the different type can allow different patterns of parallel access. Because views adapt memory types to memory types, they compose. Section 5.5 reports on case studies that use views in real benchmark code.

Programs can access views with subscripting, just like physical memories. View accesses have implications both for type checking and for the generated hardware. At compile time, a view access determines which banks of the underlying memory to consume. At run time, a view access invokes hardware logic to compute new indices and route requests to the appropriate memory banks.

Dahlia provides a library of views that, together, cover a broad space of useful iteration patterns. The rest of this section describes each of the four built-in memory views and depicts its hardware implementation and cost.

Shrink. Shrink views reduce the banking factor of an underlying memory by an integer factor. Reduced banking factors are useful when the program needs to access a memory from a loop whose unrolling factor does not exactly match the memory's banking factor. This loop is illegal, for example:

```
decl A: float[8 bank 4];
for (let i = 0..8) unroll 2 {
  A[i]; // Error: Banking factor is 4, not 2.
}
```

However, accessing `A[i]` is guaranteed to be free of bank conflicts: the first unrolled iteration pair will use banks 0 and 1, the second banks 2 and 3, and so on. To allow access from a loop with unrolling factor 2, a program can create a view of type `mem float[8 bank 2]` that refers to the same underlying storage:

```
view sh = shrink A[by 2];
```

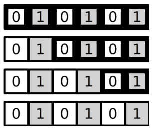
Then, accessing `sh[i]` in the loop is legal. The cost of a shrink view is indirection hardware to let the loop body switch between different subsets of the memory's banks.

Suffix and prefix. A second kind of view lets programs create small slices of a larger memory. Dahlia divides the problem into suffixing (taking the last n elements) and prefixing (taking the first n elements). 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 `v` starts at element $k \times e$ of the memory `M`. Critically, k must be the banking factor of `M`. Aligning to the banking factor ensures that Dahlia statically knows the bank of `M` for every index in `v`: namely, `v` has the same banking factor as `M` and the bank for every element in `v` is equal to the bank in the underlying memory. For example, generating suffixes in a loop results in this pattern, where the digits in each cell are bank numbers and the highlighted blocks indicate the view:

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



The view in each iteration aligns to the bank boundaries in `A`. As a result, the generated hardware does not require any bank indirection—accesses through the view can statically connect to a single bank and just shift the indices to each.

Prefix views have no hardware cost: they change neither the index nor the bank number for any access. They statically restrict accesses by producing a memory with a smaller size.

Rotation. Rotation suffixes are like standard suffixes but allow unrestricted offset expression. Dahlia does not reason about these offsets since parallelizing the context that creates them is disallowed. They come at a higher hardware cost because they need to generate bank indirection hardware. This loop is legal, for example:

```
for (let i = 0..4) {
  view r = rotate A[by i*i];  r[j];
}
```

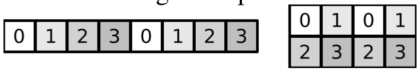
And the access in the loop is equivalent to `A[i*i + j]`.

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:

```
decl A: float[8 bank 4];  decl B: float[8 bank 4];
let sum = 0.0;
for (let i = 0..4) {
  for (let j = 0..2) {
    let v = A[2*i + j] * B[2*i + j];
  } combine {
    sum += v;
  }
}
```

Both the inner loop and the outer loop represent opportunities for parallelization. To allow unrolling each loop, Dahlia requires banking corresponding to the unrolling factor.

Split views let programs logically block arrays to allow parallelism. This left diagram represents `A` or `B` above:



A split view transforms this one-dimensional memory into a two-dimensional memory (right), where each dimension contains logical chunks for the computation. No new physical memory is necessary: a split view remaps the same underlying array to a 2D memory type. Using these declarations:

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

Each view has type `mem float[4 bank 2][2 bank 2]`. The above example can now unroll both loops with factor 2 and use the access expressions `split_A[i][j]` and `split_B[i][j]` to load the operands.

Split views have similar cost to aligned suffix views: they require no bank indirection hardware because the bank index is always known statically. They only incur a cost to compute the address within the bank from the separate coordinates.

3.8 Pipelining

Dahlia loops support a `pipeline` annotation that lets loop iterations overlap without duplicating hardware. Pipeline parallelism runs different logical time steps of the same loop concurrently, so all time steps must be free of conflicts with each other. Dahlia therefore allows pipelining only in loops that do not contain sequential composition, implying that their bodies must already be memory conflict free, and uses an initiation interval based on intra-loop dependencies. Reasoning about complex pipelining is left for future work.

3.9 Other Features

Dahlia supports peripheral features to facilitate design.

Arbitrary precision integers. Dahlia has arbitrary-precision integers types, written `bit<n>` or `ubit<n>` for unsigned integers. The compiler infers the minimum bit width for unannotated variables.

Bounds analysis. HLS tools silently return incorrect answers for out-of-bounds indices. To help avoid this pitfall, the Dahlia compiler warns when an access may be out of bounds.

Inlined functions. Dahlia programs can define and invoke functions with an inlining semantics: every call incurs the cost of duplicating the function body.

While loops. Dahlia's `while` loops are sequential: they do not have the dependency restrictions of its `for` loops, but they also cannot be unrolled. With this distinction, Dahlia

$$\begin{array}{l}
x \in \text{variables} \qquad\qquad a \in \text{memories} \\
n \in \text{numbers} \qquad b ::= \mathbf{true} \mid \mathbf{false} \qquad v ::= n \mid b \\
e ::= v \mid \mathbf{bop}\ e_1\ e_2 \mid x \mid a[e] \\
c ::= e \mid \mathbf{let}\ x = e \mid c_1 \mathbin{-\!\!-} c_2 \mid c_1\ \mathbf{;}\ c_2 \mid \mathbf{if}\ e\ c_1\ c_2 \mid \\
\qquad \mathbf{while}\ e\ c \mid x := e \mid a[e_1] := e_2 \\
\tau ::= \mathbf{bit}\langle n \rangle \mid \mathbf{float} \mid \mathbf{bool} \mid \mathbf{mem}\ \tau[n_1\ \mathbf{bank}\ n_2]
\end{array}$$

Figure 3: Abstract syntax for the Filament core language.

permits quick prototyping while making the path to higher-performing designs clear.

4. Formalism

This section formalizes the time-sensitive affine type system that underlies Dahlia in a core language, Filament.

4.1 Syntax

Figure 3 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 sequential composition $c_1 \mathbin{-\!\!-} c_2$ and parallel composition $c_1\ \mathbf{;}\ c_2$. Filament separates memories a and variables x into separate syntactic categories. Programs can only declare the latter: a program begins with a fixed set of memories and cannot create new ones.

4.2 Operational Semantics

We define a big-step operational semantics for Filament. We use a *checked semantics* that explicitly tracks memory accesses and gets 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, which maps 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; an accompanying technical report [3] lists the full semantics.

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 $a[n]$ looks up the value for a at n in σ. It also requires that a not already be present in ρ, which would indicate that the memory was previously consumed:

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

The right-hand side of the conclusion adds a to ρ_2 to mark a as used. The rule for writes similarly enforces that $a \notin \rho_1$:

$$\frac{\begin{array}{c}\sigma_1, \rho_1, e \Downarrow \sigma_2, \rho_2, n \qquad \sigma_2, \rho_2, e \Downarrow \sigma_3, \rho_3, n \\ a \notin \rho_1 \qquad \sigma_3(a)(n) := v\end{array}}{\sigma_1, \rho_1, a[e_1] := e_2 \Downarrow \sigma_3, \rho_3 \cup \{a\}}$$

Composition. Parallel composition requires that two commands executing in parallel accumulate their resource demands by threading ρ through both commands:

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

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

$$\frac{\sigma_1, \rho_1, c_1 \Downarrow \sigma_2, \rho_2 \qquad \sigma_2, \rho_1, c_2 \Downarrow \sigma_3, \rho_3}{\sigma_1, \rho_1, c_1 \mathbin{-\!\!-} c_2 \Downarrow \sigma_3, \rho_2 \cup \rho_3}$$

The final ρ merges both sets to summarize the composition.

Loops. Filament supports standard while loops with sequential semantics:

$$\frac{\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \mathbf{false}}{\sigma_1, \rho_1, \mathbf{while}\ e\ c \Downarrow \sigma_2, \rho_2}$$

$$\frac{\begin{array}{c}\sigma_1, \rho_1, e_1 \Downarrow \sigma_2, \rho_2, \mathbf{true} \\ \sigma_2, \rho_2, c \mathbin{-\!\!-} \mathbf{while}\ e\ c \Downarrow \sigma_3, \rho_3\end{array}}{\sigma_1, \rho_1, \mathbf{while}\ e\ c \Downarrow \sigma_3, \rho_3}$$

Each iteration of a loop runs sequentially. Dahlia's unrolled `for` loops can be implemented by desugaring to Filament's simpler loops (see Section 3.5).

4.3 Type System

We formalize time-sensitive affine types in Filament with respect to the operational semantics. 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 (use-at-most-once) resources. The rules for reads and writes get the type of the memory from Γ and remove the memory from Δ:

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

$$\frac{\begin{array}{c}\Gamma, \Delta_1 \vdash e_1 : \mathbf{bit}\langle n \rangle \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash e_2 : \tau \dashv \Delta_3 \\ \Delta_2 = \Delta_3 \cup \{a \mapsto \mathbf{mem}\ \tau[n_1]\}\end{array}}{\Gamma, \Delta_1 \vdash a[e_1] := e_2 \dashv \Gamma, \Delta_3}$$

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

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

This way, variables defined in the first statement are available in the second one, but the memories consumed in the first are unavailable in the second.

Sequential composition checks both commands with the same set of resources, but the variables defined in the first are threaded through to the second:

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

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

Control constructs. The rule for **if** checks the condition and both branches in the same initial Δ context:

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \tau \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash c_1 \dashv \Gamma_2, \Delta_3 \\ \Gamma, \Delta_2 \vdash c_2 \dashv \Gamma_3, \Delta_4}{\Gamma, \Delta_1 \vdash \textbf{if } e\ c_1\ c_2 \dashv \Gamma, \Delta_2 \cap \Delta_3 \cap \Delta_4}$$

This policy represents a hardware implementation strategy where only one branch is active at a time—where the condition enables one logic block and disables the other. In a different implementation, the hardware could run both branches and multiplex the result. The program must then ensure that the branches are conflict free, suggesting an alternative rule:

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \tau \dashv \Delta_2 \qquad \Gamma, \Delta_2 \vdash c_1 \dashv \Gamma_2, \Delta_3 \\ \Gamma, \Delta_3 \vdash c_2 \dashv \Gamma_3, \Delta_4}{\Gamma, \Delta_1 \vdash \textbf{if } e\ c_1\ c_2 \dashv \Gamma, \Delta_4}$$

The Dahlia compiler uses the first rule, but changing the implementation strategy would only require this local change to the type system.

The type rule for `while` loops is straightforward: it prevents conflicts between the condition and the loop body. The remaining rules follow a standard imperative semantics [3].

4.4 Soundness Theorem

We state a soundness theorem for Filament's type system with respect to its checked operational semantics. The theorem says that a well-typed program does not get stuck:

Theorem. *If* $\Gamma_1, \Delta_1, c \vdash \Gamma_2, \Delta_2$*, then* $\emptyset, \emptyset, c \Downarrow \sigma, \rho$.

The theorem implies that the type system rules out stuckness due to memory conflicts in ρ. We sketch a proof by induction in the accompanying technical report [3].

5. Evaluation

The Dahlia type system is intentionally conservative: it rules out programming patterns and optimizations that it deems unsafe or undesirable. We use a traditional HLS benchmark suite to investigate this trade-off by measuring the efficiency of accelerators (Section 5.4) and reporting case studies from the porting process (Section 5.5).

5.1 Implementation

We implemented a compiler for Dahlia in 3800 LoC of Scala. The compiler checks Dahlia programs and generates C++ using Xilinx Vivado HLS's `#pragma` directives [53] to implement aspects of Dahlia's semantics such as memory banking, loop unrolling, and function inlining. The generated code uses Vivado HLS's `ap_int` types for arbitrary-precision integers.

5.2 Benchmarks

Our evaluation uses MachSuite, a benchmark suite for traditional HLS [45]. Table 1 lists the MachSuite benchmarks we use. MachSuite consists of 19 benchmarks; we eliminate one, `backprop`, because the data generation tool fails to compile and implementation contains a functional bug and fails its own output check.

Baseline optimizations. The MachSuite benchmarks include some suggested HLS directives, but there is no standard high-performing version of each. To obtain a competitive baseline, we manually tuned each benchmark by adding and adapting the included directives. We use the best-performing design we found that fits within our target FPGA's resource budget. This design space exploration was not exhaustive: it is a best-effort attempt to obtain good accelerators for each benchmark. We plan to release the optimized version of MachSuite on publication.

Porting to Dahlia. We ported all 18 non-buggy MachSuite benchmarks from HLS C to Dahlia. The translation is mostly syntactic—the benchmarks do not exercise all the dark corners of C; they tend to use standard imperative features with obvious Dahlia equivalents. Exceptions included C preprocessor macros (which we either inlined by hand or converted to functions), global arrays (which we explicit passed as arguments), and `break` (which we implemented with conditionals). During the porting process, Dahlia's type errors guided the insertion of sequential composition.

We optimized each benchmark independently of the baseline's directives. Dahlia prevents some optimizations that are possible with traditional HLS directives, and it makes others more convenient to apply. The end result is a different accelerator design for each algorithm. Section 5.5 describes the porting experience in more detail.

5.3 Experimental Setup

We use Vivado HLS v2017.2 and Vivado v2017.2_sdx to synthesize RTL for both the baseline implementation and the code generated by Dahlia. We use Vivado HLS directives to generate AXI-Lite interfaces for top-level functions to ensure that the results include BRAMs to store the accelerator's inputs and outputs and that the designs are self contained.

We target ZedBoard [5], which contains a Xilinx Zynq-7000 SoC (XC7Z020) equipped with an Artix-7 FPGA [54]. Synthesized designs meet timing at a clock period of 7 ns

		C++						Dahlia					
Name	**Description**	**LOC**	**LUT**	**FF**	**BRAM**	**DSP**	**Cycles**	**LOC**	**LUT**	**FF**	**BRAM**	**DSP**	**Cycles**
aes	AES encryption	122	1637	1880	5	0	2 K	259	2080	1977	21	0	1 K
bfs-bulk	Simple breadth-first search	35	3037	2349	23	0	140 K	45	956	998	23	0	145 K
bfs-queue	BFS using work queue	42	4363	3665	23	0	35 K	65	933	947	23	0	31 K
fft-strided	Cooley–Tukey Fourier transform	30	3626	4989	22	50	168 K	39	3792	5429	8	56	168 K
fft-transpose	FFT, small-radix transforms	343	61660	36089	14	218	176 K	305	36524	23352	12	218	164 K
gemm-blocked	Matrix multiply, better locality	24	9543	14134	48	112	311 K	20	8737	13226	88	56	622 K
gemm-ncubed	Matrix multiply, naive	21	11362	17696	48	14	262 K	17	17565	26077	48	56	66 K
kmp	String matching with an FSM	37	855	940	11	0	356 K	45	675	714	11	0	356 K
md-grid	N-body, locality optimized	82	23258	28698	19	42	777 K	118	8371	12249	19	42	778 K
md-knn	N-body, k nearest neighbors	48	10985	15789	104	28	75 K	42	21873	24726	104	28	74 K
nw	Needleman–Wunsch alignment	80	33128	16666	48	0	83 K	101	2500	2093	46	0	83 K
sort-merge	Merge sort of 32-bit integers	44	896	799	4	0	90 K	65	610	489	6	0	98 K
sort-radix	Radix-4 integer sort	90	2650	2736	8	0	530 K	103	1419	1362	8	0	755 K
spmv-crs	Sparse matrix × vector product	20	3527	4343	50	14	21 K	25	3516	4162	32	14	18 K
spmv-ellpack	SPMV, more regular storage	18	9601	7947	56	14	48 K	11	9667	13430	138	14	53 K
stencil-stencil2d	3×3 filter on matrix	20	1152	1744	16	27	46 K	20	2584	2416	17	27	71 K
stencil-stencil3d	7-point stencil on 3D tensor	42	4030	4267	33	81	46 K	65	8040	7514	43	93	37 K
viterbi	Hidden Markov model inference	56	2002	2287	52	3	12 M	59	2064	2265	52	3	12 M

Table 1: MachSuite [45] benchmarks used in the evaluation. The LOC columns show the non-comment, non-blank lines of code for the original HLS baseline and the Dahlia port. The other columns show the FPGA resources occupied (look-up tables, flip-flops, block RAMs, and DSP logic blocks) and the estimated latency in cycles.

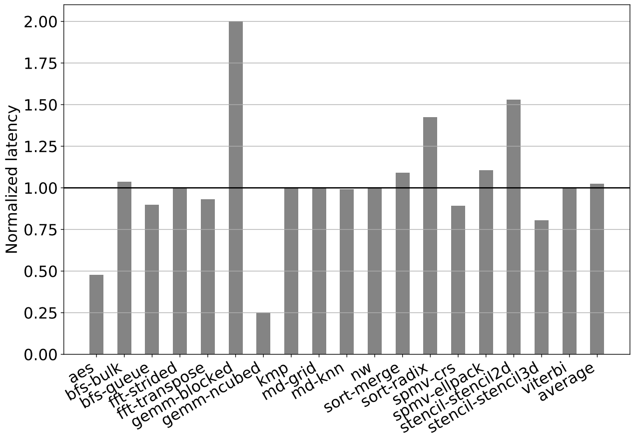


Figure 4: Latency for Dahlia normalized to the baseline.

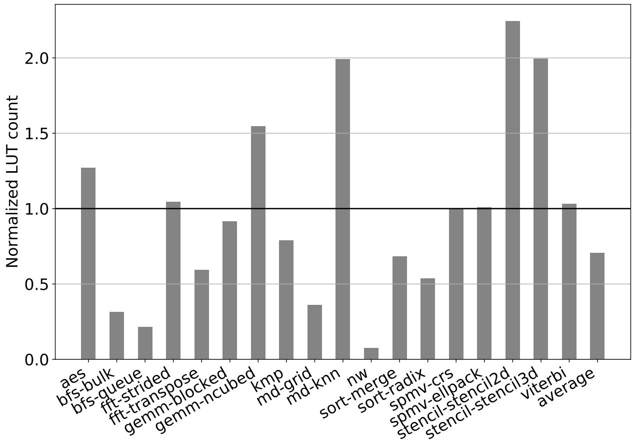


Figure 5: LUTs used for Dahlia normalized to the baseline.

(142 MHz). We report the post-synthesis resource utilization and Vivado HLS's latency estimates.

5.4 Accelerator Efficiency Impact

We measure the difference in hardware efficiency between the hand-tuned baseline C++ implementation for each benchmark and our Dahlia ports. Efficiency has two main factors: performance (clock cycles) and area (resource utilization), which in turn encompasses the FPGA's look-up tables (LUTs), flip-flops, block RAMs (BRAMs), and arithmetic units (DSP slices). Table 1 lists the resource counts and estimated cycles.

Figure 4 shows the normalized latency for each Dahlia port normalized to the HLS C++ baseline. The performance for the two sets of benchmarks is generally similar: on average, the latency for the Dahlia ports is 102% of the baseline. gemm-blocked suffers the largest slowdown: it is 2.0× slower than the baseline. The baseline takes advantage of the FPGA's dual-ported BRAMs by unrolling an inner loop by a factor greater than the data memory's banking factor. Dahlia conservatively models single-ported BRAMs; we see it as important future work to model multi-ported memories to allow more parallelism. The best speedup is gemm-ncubed, whose latency is 4.0× faster than the baseline. The baseline code uses single-dimensional arrays, which does not allow banking across multiple dimensions. The Dahlia port naturally supports multi-dimensional banking and unrolling. Finally, the Dahlia port for aes found a design point that uses unrolling extensively, which performs better than the heavily pipelined design in the baseline.

Figure 5 compares the benchmarks' LUT utilization. Resource usage is also generally similar: all benchmarks fit on the same target FPGA device, and the Dahlia ports use 70% of the baseline LUTs on average. The largest change from C++ to Dahlia is the nw benchmark: the baseline uses 33128 LUTs while the Dahlia port uses only 2500. The baseline used a heavily pipelined design, but the performance reward for the area cost is negligible. We traced the lack of speedup to two causes: the use of single-dimensional arrays, which prevents good memory locality, and the use of a ternary operator for a condition, which triggered a pathology in the HLS compiler that led to a needlessly bad design.

5.5 Programmability

Although Dahlia is more restrictive than traditional HLS by design, its restrictions did not prevent us from expressing any of the 18 non-buggy MachSuite benchmarks. Table 1 compares the lines of code for the baseline C++ and Dahlia implementation. The program size is similar in most cases, with the exception of aes, whose heavy use of C preprocessor macros and irregular memory accesses required additional lines to express in Dahlia. The rest of this section uses case studies to depict Dahlia's safety and expressiveness advantages.

Rotation views for stencil-like iteration. MachSuite's stencil-stencil2d is a standard window-based filter operation with four nested loops. The core computation iterates over 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 Dahlia, we must use proper two-dimensional arrays to avoid index math. Using views, programmers can decouple the storage format from the iteration pattern. To express the accesses to the input matrix `orig`, we create a rotation suffix view (Section 3.7) for the current window:

```
for (let r = 0..126) {
  for (let c = 0..62) {
    view window = rotate orig[by r][by c];
    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 be undesirable: the parallel views would require overlapping regions of the input array, introducing a bank conflict.

Split views for logical indexing. In aes, the `mixColumns` function accesses a buffer (bu) in blocks of 4 adjacent values:

```
for (i = 0; i < 16; i += 4) {
  a = bu[i]; b = bu[i+1]; c = bu[i+2]; d = bu[i+3];
```

This pattern benefits from banking bu with factor 4. The Dahlia port uses a split view to divide bu into logical blocks of 4:

```
view buf_view = split bu[by 4];
```

Subsequent code can index the buffer with two-dimensional subscripts as in `buf_view[i][3]` to prove the conflict freedom of parallel accesses.

Exposing the cost of offset iteration. The spmv-crs sparse matrix–vector multiplication iterates over pairs of adjacent values in a banked array of delimiters:

```
for (i = 0; i < N; i++) {
  begin = delimit[i];  end = delimit[i+1];
```

While this code looks innocuous, it interacts poorly with memory banking. Accessing an offset index requires additional indirection hardware to connect to a different bank. The Dahlia type checker rejects the naive translation of this code when `delimit` is banked, but a rotation view makes the additional hardware explicit:

```
view delimit_rot = rotate delimit[by 1];
```

The loop body uses `delimit_rot[i]` for the offset access.

Discouraging complex memory accesses. The gemm-blocked benchmark tiles a matrix multiplication in two dimensions. The baseline C++ implementation writes partial sums directly into the output array:

```
for (k = 0; k < block_size; ++k) {
  temp_x = m1[(i * row_size) + k + kk];
  for (j = 0; j < block_size; ++j) {
    mul = temp_x * m2[(k+kk)*row_size + j + jj];
    prod[(i*row_size) + j + jj] += mul;  ...
```

A naive port to Dahlia results in a type error because the += accumulation simultaneously reads from and writes to the output array. Dahlia's loop combiners (Section 3.6) encourage a more explicit style that accumulates into a local variable representing a register:

```
for (let k = 0..8) unroll 8 {
  let temp_x = m1_v[i][k];
  let mul = temp_x * m2_v[k][j];
} combine {
  prod_v[i][j] += mul;
}
```

By flagging conflicting memory accesses, the Dahlia type checker encourages writing a more explicit program that more directly corresponds to the generated hardware.

6. 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 [9, 14, 24, 36, 49]. Dahlia's enforcement of exclusive memory access resembles work on race-free parallel programming using type and effect systems [10] or concurrent separation logic [39]. 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.

Better HDLs. An emerging family of modern hardware description languages (HDLs) [6, 7, 15, 30, 35, 38, 50] aims to address the shortcomings of Verilog and VHDL by improving their safety and expressiveness. These language 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 an out-of-order RISC processor, for example. Its focus on computational acceleration requires the language and semantics to more closely resemble software languages.

Traditional HLS. Existing commercial [11, 29, 37, 53] and academic [12, 40, 43, 55] high-level synthesis (HLS) tools compile subsets of C, C++, OpenCL, or SystemC to RTL. These tools focus on powerful heuristics to search for hardware implementations that match the software semantics. While they can be effective, these mapping heuristics can also be complex, slow, and difficult to control: when they fail to produce good hardware, 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.

DSLs for hardware. Several domain-specific languages (DSLs) enable compilers to FPGAs and ASICs [21, 26, 27, 41, 47]. Dahlia is not a DSL: it is a general language for implementing computational accelerators. While DSLs offer advantages in productivity and compilation for individual 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 share Dahlia's goal of facilitating general accelerator design. HeteroCL [33] uses a Halide-like [44] scheduling language to describe how to map algorithms onto HLS-like hardware optimizations, and T2S [46] similarly lets programs describe how to transform imperative code into a spatial implementation. Lime [4] extends Java to express target-independent streaming accelerators. CoRAM [13] is not a just a language; it extends FPGAs with a programmable memory interface that adapts memory accesses, akin to Dahlia's memory views. Spatial [32] is most closely related: it relies on the composition of *parallel patterns*, which are higher-order programming constructs that come with hardware implementation strategies. The compiler automatically maps high-level constructs like maps and folds onto low-level resources while optimizing for utilization and latency. Dahlia differs from Spatial by embedding hardware semantics into the surface-level language, where the programmer can directly control the resource mapping. Spatial does not enforce the timing and memory access constraints that Dahlia's time-sensitive affine type system encodes and does not guarantee equivalence to sequential semantics when using the parallelization constructs.

7. Conclusion

Dahlia emphasizes rapid iteration over monolithic design. While the "waterfall" development model may be appropriate for manufacturing real silicon, long-running and black-box implementation tools are a poor fit for reconfigurable hardware. We hope to extend Dahlia's philosophy of programmer control to make the rest of the reconfigurable hardware stack, from the language to the LUTs, more predictable.

References

[1] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[2] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, Jan 2009.

[3] Anonymous for double blind review. Predictable accelerator design with time-sensitive affine types: Extended technical report and full semantics, 2019.

[4] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2010.

[5] Avnet. ZedBoard technical specifications. `http://www.zedboard.org/content/zedboard-0`.

[6] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. CλaSH: Structural descriptions of synchronous hardware using Haskell. In *Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC)*, 2012.

[8] H. G. Baker. "Use-once" variables and linear objects: Storage management, reflection and multi-threading. *SIGPLAN Notices*, 30(1):45–52, Jan. 1995.

[9] J. Bernardy, M. Boespflug, R. Newton, S. L. P. Jones, and A. Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[10] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009.

[11] 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`.

[12] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2011.

[13] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Field programmable gate arrays (FPGA)*, 2011.

[14] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*, 2015.

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

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

[17] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *International SoC Conference*, 2006.

[18] 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):473–491, Apr. 2011.

[19] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In *European Symposium on Programming (ESOP)*, 2006.

[20] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale DNN processor for real-time AI. In *International Symposium on Computer Architecture (ISCA)*, 2018.

[21] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2014.

[22] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.

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

[24] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[25] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, Jan. 2004.

[26] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics*, 33(4), 2014.

[27] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics*, 35(4), 2016.

[28] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM (CACM)*, 62(2):48–60, Jan. 2019.

[29] Intel. Intel High Level Synthesis Compiler. URL `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] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*, 2017.

[32] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun. Spatial: a language and compiler for application accelerators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[33] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[34] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012.

[35] D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A unified framework for vertically integrated computer architecture research. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

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

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

`level-synthesis/`.

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

[39] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375:271–307, Apr. 2007.

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

[41] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017.

[42] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[43] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2008.

[44] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. 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)*, 2013.

[45] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[46] H. Rong. Programmatic control of a compiler for generating high-performance spatial hardware. arXiv preprint 1711.07606. `https://arxiv.org/abs/1711.07606`, Nov. 2017.

[47] J. Setter. Halide-to-Hardware. `https://github.com/jeffsetter/Halide-to-Hardware`.

[48] S. Sutherland, D. Mills, and C. Spear. Gotcha again: More subtleties in the Verilog and SystemVerilog standards that every engineer should know. In *Synopsys Users Group (SNUG) San Jose*, 2007.

[49] J. A. Tov and R. Pucella. Practical affine types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[50] L. Truong and P. Hanrahan. A golden age of hardware description languages: Applying programming language techniques to improve design productivity, 2019.

[51] Y. Turakhia, G. Bejerano, and W. J. Dally. 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)*, 2018.

[52] Xilinx Inc. 7 Series FPGAs Memory Resources, . `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`.

[53] 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`.

[54] Xilinx Inc. Zynq-7000 SoC Data Sheet. `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`, .

[55] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A platform-based esl synthesis system. In *High-Level Synthesis*, pages 99–112. 2008.