Wavefront Threading Enables Effective High-Level Synthesis

BLAKE PELTON, Microsoft, USA
ADAM SAPEK, Microsoft, USA
KEN EGURO, Microsoft, USA
DANIEL LO, Microsoft, USA
ALESSANDRO FORIN, Microsoft, USA
MATT HUMPHREY, Microsoft, USA
JINWEN XI, Microsoft, USA
DAVID COX, Microsoft, USA
RAJAS KARANDIKAR*, Microsoft, USA
JOHANNES DE FINE LICHT†, ETH Zurich, Switzerland
EVGENY BABIN, Microsoft, USA
ADRIAN CAULFIELD, Microsoft, USA
DOUG BURGER, Microsoft, USA

Digital systems are growing in importance and computing hardware is growing more heterogeneous. Hardware design, however, remains laborious and expensive, in part due to the limitations of conventional hardware description languages (HDLs) like VHDL and Verilog. A longstanding research goal has been programming hardware like software, with high-level languages that can generate efficient hardware designs. This paper describes Kanagawa, a language that takes a new approach to combine the programmer productivity benefits of traditional High-Level Synthesis (HLS) approaches with the expressibility and hardware efficiency of Register-Transfer Level (RTL) design. The language's concise syntax, matched with a hardware design-friendly execution model, permits a relatively simple toolchain to map high-level code into efficient hardware implementations.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; • **Computing methodologies → Concurrent programming languages**.

Additional Key Words and Phrases: Hardware Threading, Wavefront Threading, Wavefront Consistency

ACM Reference Format:
Blake Pelton, Adam Sapek, Ken Eguro, Daniel Lo, Alessandro Forin, Matt Humphrey, Jinwen Xi, David Cox, Rajas Karandikar, Johannes de Fine Licht, Evgeny Babin, Adrian Caulfield, and Doug Burger. 2024. Wavefront

*Author was at Purdue University during their contribution to this work. They are now at Microsoft.
†Author was at ETH Zurich during their contribution to this work. They are now at NextSilicon.

Authors' addresses: Blake Pelton, Microsoft, Redmond, USA, blakep@microsoft.com; Adam Sapek, Microsoft, Redmond, USA, adamsap@microsoft.com; Ken Eguro, Microsoft, Redmond, USA, eguro@microsoft.com; Daniel Lo, Microsoft, Redmond, USA, dlo@microsoft.com; Alessandro Forin, Microsoft, Redmond, USA, sandrof@microsoft.com; Matt Humphrey, Microsoft, Redmond, USA, mhumphr@microsoft.com; Jinwen Xi, Microsoft, Redmond, USA, jixi@microsoft.com; David Cox, Microsoft, Redmond, USA, coxdavid@microsoft.com; Rajas Karandikar, Microsoft, Redmond, USA, rakarand@microsoft.com; Johannes de Fine Licht, ETH Zurich, Zurich, Switzerland, definelicht@inf.ethz.ch; Evgeny Babin, Microsoft, Redmond, USA, evgenybabin@microsoft.com; Adrian Caulfield, Microsoft, Redmond, USA, acaulfie@microsoft.com; Doug Burger, Microsoft, Redmond, USA, dburger@microsoft.com.

ACM 2475-1421/2024/6-ART190
https://doi.org/10.1145/3656420

Threading Enables Effective High-Level Synthesis. *Proc. ACM Program. Lang.* 8, PLDI, Article 190 (June 2024), 25 pages. https://doi.org/10.1145/3656420

1 INTRODUCTION

The challenge of efficiently designing hardware using high-level programming languages has long been a focus of research. Hardware design, whether for flexible logic on Field-Programmable Gate Arrays (FPGAs) or hardened Application-Specific Integrated Circuits (ASICs), is dominated by Register-Transfer Level (RTL) Hardware Description Languages (HDLs) like Verilog and VHDL. These languages excel in expressiveness within the hardware domain, but suffer from a low level of abstraction that limits productivity. Designers must explicitly translate algorithmic intentions into low-level logic. While high-level HDLs such as Bluespec [44] and Chisel [3] provide richer abstractions, they remain largely tethered to the low-level hardware programming paradigm.

High-Level Synthesis (HLS) offers the promise of designing hardware using an imperative programming paradigm that has software development-like productivity. However, to date, HLS solutions have not seen widespread adoption due to challenging limitations. Automatically mapping sequential imperative code to efficient parallel hardware has proven to be a complex compiler design problem. Traditional HLS tools often suffer from unpredictable performance and lack language features to express various design choices. To circumvent these limitations, users often must resort to steering compiler decisions using hints and pragmas which negate the benefits of programming at a higher level of abstraction, limiting code reuse and composability.

Kanagawa is a high-level, imperative programming language developed specifically for hardware design. Kanagawa relies on a new execution model for imperative code called Wavefront Threading. This execution model enables straightforward and predictable mapping to efficient parallel hardware. Kanagawa offers expressiveness rivaling RTL, with the high level of abstraction and productivity of software development. Synthesized Kanagawa designs compare in quality to those manually written by experienced designers in RTL. Kanagawa programmers have leveraged its versatility to create a wide range of significant hardware designs, some of which are in large-scale production in Microsoft Azure. Kanagawa is sufficiently expressive to develop a full RISC-V core.

In the subsequent sections of this paper, we will illustrate how the Kanagawa execution model addresses the challenges of previous HLS tools, describe the Kanagawa language, explain how source code is mapped to hardware, and provide a quantitative assessment of Kanagawa versus production-quality RTL designs.

2 MOTIVATIONAL EXAMPLE

HLS tools commonly pipeline loops (*i.e.*, execute multiple iterations in parallel) to achieve high performance. Figure 1 shows the `CountIf Histogram` loop from Dai et al. [16], written in C. A similar example appears in Josipović et al. [24]. When pipelining the loop there is a read-after-write hazard on `hist[m]`[1]. How often this hazard must be addressed is data dependent. Assume that the histogram read, floating-point addition, and histogram write have a combined latency of L clock cycles. There are multiple options for handling this hazard, including:

(1) Statically schedule the loop so that different iterations do not concurrently execute the read-modify-write. For example, configure the pipeline initiation interval [47] to L. This approach provides limited throughput regardless of hazard frequency, but has the lowest resource usage. Figure 2 illustrates a statically scheduled execution for initiation interval=2.

[1]Iteration `i` writes on line 6 and iteration `i+1` reads on line 5

```
1  for (i=0; i<N; ++i) {
2    int m = feature[i];
3    float wt = weight[i];
4    if (m>THRESHOLD) {
5      float x = hist[m];
6      hist[m] = x + wt;
7    }
8  }
```

Fig. 1. `CountIf Histogram` in C.

`m=feature[i];`	I0{}		I1{}		I2{}			
`wt=weight[i];`		I0{m:5}		I1{m:7}		I2{m:7}		
`x=hist[m];`			I0{m:5}		I1{m:7}		I2{m:7}	
`hist[m]=x+wt;`				I0{m:5}		I1{m:7}		I2{m:7}
Time	0	1	2	3	4	5	6	7

Fig. 2. An execution of `CountIf Histogram` using static scheduling with $II = 2$. Three loop iterations ($I0$, $I1$, and $I2$) are shown with distinct colors. The value of `m` is shown at each iteration.

`m=feature[i];`	I0{}	I1{}	I2{}					
`wt=weight[i];`		I0{m:5}	I1{m:7}	I2{m:7}				
`x=hist[m];`			I0{m:5}	I1{m:7}	I2{m:7}	I2{m:7}		
`hist[m]=x+wt;`				I0{m:5}	I1{m:7}		I2{m:7}	
Time	0	1	2	3	4	5	6	7

Fig. 3. An execution of `CountIf Histogram` using dynamic scheduling. The dashed cell at time 4 shows when $I2$ must stall to resolve the read-after-write hazard.

(2) Dynamically detect hazards and stall when necessary. This option can improve best case throughput (*i.e.*, when hazards are rare) at the cost of extra logic resources. Figure 3 illustrates a dynamically scheduled execution.
(3) Interleave the computation among L intermediate histograms. Add a post-processing step to combine the histograms and produce the final result[2]. This option provides high throughput independent of hazard frequency but requires more memory and logic resources.
(4) Speculatively read and modify `hist` before previous writes have completed. Before writing to `hist`, check to see if there was a hazard and restart the iteration if necessary. This option also improves best case throughput and has resource costs comparable to option 2.

In imperative languages supported by traditional HLS tools (*e.g.*, C/C++), it is difficult to fully express all these design choices because the concurrency of loop iterations is not explicit and must be inferred by the compiler. For example, option 3 could be partially expressed by scaling the `hist` array size by L and adding a post-processing loop, but the programmer would have to rely on the compiler to recognize that it is safe to run up to L iterations of the primary loop concurrently or add tool-specific hints or pragmas.

Commercial HLS tools typically choose option 1, at the cost of throughput when hazards are rare. Research into dynamic HLS [24] proposes ways to achieve option 2 when hazards are common, at the cost of hardware resources. Other research [17, 19] proposes automated generation of speculative hardware (option 4, possibly with hints about the likely outcome of a conditional). Hybrid approaches are also possible, such as combining speculation with dynamic HLS [25].

[2]Assume the developer has determined floating-point addition in this application can be treated as associative.

The problem is fundamental to existing HLS approaches: sequential imperative code does not include enough context information for a tractable compiler to reliably generate efficient parallel hardware. Prior work [10, 16, 32, 33, 38] has shown that HLS tools struggle to generate efficient hardware while maintaining sequential semantics for basic control flow compositions such as nested/sequential/irregular loops and for non-trivial loop-carried dependencies.

3 KANAGAWA LANGUAGE

Kanagawa has a threaded execution model and associated memory consistency model that facilitates translation to hardware. Concurrency is exposed explicitly using threads. The language and execution model support fine-grained threads running imperative code. In this section, we describe the semantics of the execution and memory models using `CountIf Histogram` to illustrate them in practice. Three of the `CountIf Histogram` implementation options are discussed in this section and the fourth is described in Section 4.

3.1 Wavefront Threading

Kanagawa threads are lightweight, which encourages the expression of fine-grained concurrency, similar to how low per-thread overhead in GPU programming [40] facilitates the development of highly concurrent programs. It is common, for example, to execute each iteration of a loop using a different thread. The key property of the Wavefront Threading model that reduces the complexity of reasoning about highly concurrent code is that threads maintain relative order when executing imperative code. If thread $T0$ starts executing code *ahead of* thread $T1$, $T0$ will remain ahead of $T1$ throughout execution, unless the ordering is explicitly relaxed using dedicated control flow constructs (Section 3.5).

Preserving thread order during execution not only simplifies reasoning about correctness, but also eliminates the need for explicit synchronization in many cases. Consider the example of `CountIf Histogram`, which can be challenging to parallelize in a typical multi-threaded language if floating-point addition cannot be treated as associative. Preserving the order of additions would require a complex synchronization mechanism such as ticket locks [37], and the time spent acquiring locks would likely dominate the runtime. With Wavefront Threading, no synchronization is needed because thread ordering guarantees that the additions occur in the correct order. Another consequence of Wavefront Threading is that, during the execution of a function by a group of threads, each thread's position within the group is well defined. This information can often be used in lieu of synchronization. For instance, the first thread in a group can be designated to reset shared state to an initial value, while the final thread can handle post-processing tasks.

The term "wavefront" reflects how a computation progresses in time, similar to Wavefront parallelism [8, 29, 52]. As illustrated in Figure 4, each thread can be conceptualized as a wave flowing through the computation from top-left to bottom-right.

3.2 Shared State

State in Kanagawa is stored in named variables, with a strict separation between thread-local and shared state. Threads communicate through shared variables (class members or static local variables) and use synchronization mechanisms from the language and libraries to ensure correctness. There are no pointers, so aliasing is impossible.

```
1  uint32 a = x; // Read from shared variable x
2  uint32 b = y; // Read from shared variable y
3  x = a+b;      // Write to shared variable x
4  y = a*2;      // Write to shared variable y
```

a=x;	T0{}	T1{}	T2{}	T2{}				
b=y;		T0{a:2}	T1{a:2}	T1{a:2}	T2{a:5}			
x=a+b;			T0{a:2,b:3}		T1{a:2,b:3}	T2{a:5,b:4}		
y=a*2;				T0{a:2}		T1{a:2}	T2{a:5}	
x	2	2	2	5	5	5	9	9
y	3	3	3	3	4	4	4	10
Time	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Fig. 4. Ordering example code and execution. Each column represents a step of the global clock. Each of the first four rows corresponds to a different static memory access. Values of thread local variables are shown in curly braces. The next two rows show the values of the shared variables. Threads may arbitrarily stall but must maintain their relative ordering. This is shown by the stall of threads $T1$ and $T2$ at time 2.

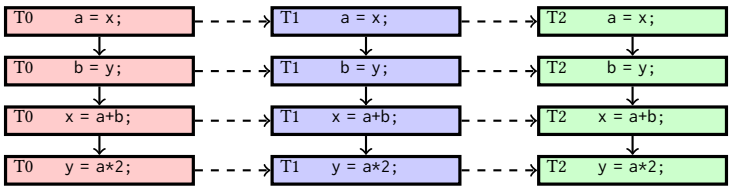

Fig. 5. Dependencies in ordering example. The statements executed by a particular thread are stacked on top of each other. Solid vertical dependence arcs represent the requirement that a particular thread must perform all memory accesses in program order. Dashed horizontal dependence arcs represent inter-thread constraints imposed by Wavefront Consistency.

3.3 Wavefront Consistency

Because Kanagawa models synchronous digital circuits, memory[3] consistency is defined in terms of a global clock, where each access to memory occurs at a particular clock cycle, and multiple accesses can occur at the same time (*i.e.*, the same clock cycle).

The model is similar to sequential consistency [30] in that any valid execution must behave as if all memory accesses performed by a thread are issued in the order defined by the program. Wavefront Consistency further restricts the set of valid executions based on thread ordering. If thread $T0$ is ahead of thread $T1$ and both threads execute the same memory access operation, then the model defines that the thread $T0$ will perform the memory access before thread $T1$.

A simple illustration of Wavefront Consistency is shown in Figure 4. This example contains four statements. The variables x and y are shared among threads, whereas a and b are thread-local. Each statement contains exactly one access to a shared variable. Figure 4 shows a valid execution of the ordering example by three threads ($T0$, $T1$, and $T2$). $T0$ executes ahead of $T1$, and $T1$ executes ahead of $T2$. A given thread executes statements from top to bottom (note that threads are not required to advance on each step of global clock, *e.g.*, time 2). Figure 5 shows dependencies between the four statements executed by the three threads.

[3]The term "memory" has different connotations in the programming languages and hardware design fields. In this work, we use the terms "memory" and "shared variable" synonymously to refer to state shared between threads.

3.4 Scheduling Constraints and Synchronization

Programmers can further limit the set of allowed executions of concurrent code using two language mechanisms: scheduling constraints and inter-thread synchronization.

Static scheduling constraints can be used to limit concurrency for a section of code, similar to how atomic read-modify-write operations limit the set of allowed executions in other consistency models [39]. Scheduling constraints are expressed using the `[[schedule(N)]]` attribute, which applies the following restrictions to a block of code:

- No more than `N` threads will be executing statements within the block at a time.
- All writes to shared state within the block will be scheduled for the same cycle (typically the last cycle in which a thread executes statements contained in the block). [4]
- If `N=1`, then all reads from shared state within the block for a given thread will be scheduled for the same cycle (not necessarily the same cycle as the writes).

The most commonly used `[[schedule(N)]]` is when `N=1`, and Kanagawa defines a convenient alias for this case: `atomic`. This is often used to protect a critical section.

The `[[thread_rate(N)]]` attribute can be used to specify the initiation interval of a function, limiting the rate at which threads can enter the function.

Scheduling constraints are used to implement the statically scheduled variant of `CountIf Histogram` (option 1). Figure 6 shows the code[5]. The implementation makes use of `pipelined_for`[6], a library function that executes a user-specified function `N` times, each with a separate thread. The `atomic` scheduling constraint ensures that only one thread can perform the read-modify-write sequence at a time. The `[[thread_rate(L)]]` attribute sets the initiation interval of `body` to L.

Inter-thread synchronization in Kanagawa can be expressed using common synchronization objects like mutexes, semaphores, or read-write locks, which are provided by the standard library. Most are built on top of a single language primitive, `wait_for`, which combines the `atomic` scheduling constraint with a loop. A statement `wait_for(expr)` evaluates `expr` for a single thread until `expr` evaluates to true. Accesses to shared variables within `expr` occur at the same time. `wait_for` maintains thread ordering, thus exhibiting head-of-line blocking.

Figure 7 shows Kanagawa code implementing dynamic scheduling (option 2). The shared variable `locks` implements an array of mutexes[7]. `wait_for` is used to stall a thread if there are in-flight accesses to the same histogram bucket. This code passes a lambda to `pipelined_for` rather than using a named function like the previous example.

Figure 8 demonstrates how to use scheduling constraints to implement a version of `CountIf Histogram` that operates on L separate histograms, all stored in the `hist` array (option 3). The `[[schedule(L)]]` constraint ensures that no more than L threads can be executing the read-modify-write sequence concurrently. Using `offset` ensures that each of these threads operates on a different intermediate histogram, eliminating the hazard. A post-processing step is added after the first loop, computing the final result. Here, a sequential `for` loop (discussed in more detail in Section 3.5.4) is nested inside the function called by `pipelined_for`. At any moment in time, there can be many threads executing the body of the `for` loop, each with a distinct value of `m`.

[4]When multiple writes are made to a shared variable within a block, intermediate values are local to a given thread. At the end of the `[[schedule(N)]]` block, the final value is written to the shared variable.

[5]When reading implementations of the running example, note that the variables `feature`, `weight`, `hist`, and `locks` are shared between threads. All other variables are thread-local.

[6]`pipelined_for` is a thin wrapper around a call to a batched function (see Section 3.5.3). It is named `pipelined_for` rather than `parallel_for` to express the notion that threads begin execution of the loop body in a well-defined order.

[7]An idiomatic implementation would use a library synchronization object rather than using `wait_for` directly, but the low-level version is shown here for clarity.

```
void static_count_if() {
  [[thread_rate(L)]] void body(uint32 i) {
    int32 m = feature[i];
    float32 wt = weight[i];
    if (m > THRESHOLD){
      atomic {
        float32 x = hist[m];
        hist[m] = x + wt;
      }
    }
  }

  pipelined_for(N, body);
}
```

Fig. 6. Statically scheduled `CountIf Histogram`.

```
bool[SIZE] locks = {};
inline bool lock(int32 m) {
  bool result = !locks[m];
  if (result) { locks[m] = true; }
  return result;
}
void dynamic_count_if() {
  pipelined_for(N, [](uint32 i) {
    int32 m = feature[i];
    float32 wt = weight[i];
    if (m > THRESHOLD) {
      wait_for(lock(m));
      float32 x = hist[m];
      hist[m] = x + wt;
      locks[m] = false;
    }
  });
}
```

Fig. 7. Dynamically scheduled `CountIf Histogram`.

```
void replicated_count_if() {
  pipelined_for(N, [](uint32 i) {
    int32 m = feature[i];
    float32 wt = weight[i];
    if (m > THRESHOLD) {
      uint32 offset = (i % L) * SIZE;
      [[schedule(L)]] {
        float32 x = hist[m + offset];
        hist[m + offset] = x + wt;
      }
    }
  });
  pipelined_for(SIZE, [](uint32 m) {
    float32 sum = 0.0;
    for (const auto l : L) {
      sum += hist[m + (l * SIZE)];
    }
    hist[m] = sum;
  });
}
```

Fig. 8. Replicated `CountIf Histogram`.

```
if (c) {
  shared_var = 1;
} else {
  x = shared_var;
}
```

Fig. 9. Ordering of conditional shared variable access.

3.5 Thread Ordering with Control Flow

In this section, we describe how thread ordering is affected by Kanagawa control flow constructs.

3.5.1 Function Calls. If thread $T0$ reaches a function call ahead of thread $T1$, then thread $T0$ will begin executing the body of the called function ahead of $T1$.

3.5.2 Asynchronous Functions. Kanagawa supports the concept of an asynchronous function call, in which a new thread executes the called function completely decoupled from the calling thread. This is the only case when total ordering among threads is not defined. If thread $T0$ reaches an asynchronous function call before thread $T1$, a new thread $T0'$ will start executing the body of the function with parameters from $T0$, ahead of another new thread $T1'$ with parameters from thread $T1$. No ordering is defined between threads $T0$ and $T0'$ nor $T1$ and $T1'$. Threads $T0$ and $T1$ continue executing without waiting for the function to return and their relative order is preserved. Asynchronous functions are building blocks which are commonly used in the implementation of control flow libraries like fork-join and rendezvous [31].

3.5.3 Batched Functions. Kanagawa supports structured concurrency with batched functions. When a thread calls a batched function, the function body is executed N times by a group of N threads, where N is specified by the first argument of the function call. The calling thread blocks until all nested threads finish execution. The value of the first parameter within the function body specifies an index of the current thread within the group, with the first thread starting at index 0. The remaining parameter values are set to argument values from the calling thread. If thread $T0$ reaches a call to a batched function ahead of thread $T1$, then all nested threads associated with $T0$ begin execution of the function ahead of nested threads associated with thread $T1$.

Kanagawa models structured concurrency as calls to batched functions to preserve the strict separation of thread-local states. The threads which execute a batched function do not have access to local state of the calling thread. Any data from the calling thread must be explicitly passed via parameters to the batched function. The standard library functions like `pipelined_for` provide convenient wrappers around batched functions that can be used like loops, with lexical closures used to explicitly pass values from thread-local state when needed.

3.5.4 Loops. Although the best performance is usually achieved by expressing loops using library functions like the aforementioned `pipelined_for`, Kanagawa also supports sequential `for` and `do-while` loops. Sequential loops are executed by existing threads just like any other statements. Each thread reaching a loop will execute all iterations of the loop before continuing to statements following the loop[8]. If thread $T0$ reaches a loop ahead of thread $T1$, then thread $T0$ will start executing the first iteration of the loop ahead of $T1$. The threads start executing subsequent iterations in the order in which they reach end of the loop body on the previous iteration. Threads executing subsequent iterations are also given priority over new threads wishing to enter the loop.

Kanagawa supports two different behaviors for threads exiting a loop. Ordered loops always preserve thread order. Logically, there is a barrier which ensures threads begin executing the statement following the loop in the order in which they entered the loop (even if the threads executed a different number of loop iterations). With unordered loops[9], threads begin executing the statement following the loop in the order in which they reach the end of the loop (*i.e.*, finish executing the last iteration).

3.5.5 Conditionals. Conditionals do not affect thread ordering. When a thread reaches a conditional, it executes statements in both branches, predicating side-effects based on the condition. The thread-ordering restriction of Wavefront Consistency applies to all shared variable accesses, including the accesses which are disabled by the predicate. For example, in Figure 9 if thread $T0$ (with `c=true`) reaches the conditional ahead of thread $T1$ (with `c=false`), then the read executed by thread $T1$ will execute after the write executed by thread $T0$. However, if the conditional in Figure 9 were reversed such that the read of `shared_var` appeared before the write, there is no guarantee that $T0$ (executing the write) would update `shared_var` before $T1$ read it.

3.6 Parallelism

Pipeline parallelism is the primary way of achieving parallelism in Kanagawa. Here we describe language constructs to replicate design elements if additional parallelism is needed.

3.6.1 Functions. Function calls can be inlined, effectively replicating the body of the function at the call site. Each replica contains replicas of all control flow constructs, scheduling constraints, and `static` local variables from the inlined function.

[8]Multiple threads may be executing the loop concurrently.
[9]An attribute `[[unordered]]` is used to mark a loop as unordered.

```
template <typename T, auto N>
inline auto map((T) -> auto f, T[N] x) {
  using result_t = decltype(f(x[0]));
  result_t[N] r;
  static for(const auto i : N) {
    r[i] = f(x[i]);
  }
  return r;
}
template <typename T, auto N>
inline T reduce((T, T) -> T f, T[N] x) {
  static if (N == 1) {
    return x[0];
  } else {
    const auto NewN = (N + 1) / 2;
    T[NewN] new_array;
    static for(const auto i : N/2) {
      new_array[i] = f(x[2*i], x[2*i + 1]);
    }
    static if ((N % 2) == 1) {
      new_array[NewN - 1] = x[N - 1];
    }
    return reduce(f, new_array);
  }
}
template <typename T, auto N, typename R>
inline R map_reduce((T) -> R map_fn,
                    (R, R) -> R reduce_fn,
                    T[N] x) {
  return reduce(reduce_fn, map(map_fn, x));
}
```

Fig. 10. Map-reduce implementation.

3.6.2 Classes. As in other object-oriented languages, classes in Kanagawa contain both code and data and support information hiding. There is no dynamic allocation of objects in Kanagawa, and thus the number of objects is known at compile time. Each instance of a class results in the replication of member variables and non-inline methods of that class.

3.6.3 Compile Time Control. `static for` loops execute at compile time. Each iteration of a `static for` loop defines unique instances for each control flow construct and memory access in the loop body. In some cases, it is more natural to express compile time iteration with recursion rather than loops. Figure 10 shows examples of both static loops and static recursion.

3.7 Code Composability and Reuse

Composable and reusable code is taken for granted in software programming, but has been historically elusive for hardware designers, especially at smaller the granularities of individual data structures, algorithms and design patterns.

Kanagawa facilitates building composable components by including various language constructs for creating abstractions without incurring extra hardware costs. The objective is to eliminate any justification for the ad-hoc coding style that relies on copy-and-paste as a code reuse mechanism, a practice all too common in hardware design. To achieve this, we draw from established programming language design concepts, such as first-class functions [20], lambdas [26, 50], lexical closures, object orientation, and modularity. We defined their semantics in the language to specifically adhere to the zero-cost abstraction principle [51].

In Kanagawa, functions are first-class values and can be stored in data structures, passed as function arguments, or returned as function results. The support for higher-order functions is complemented by the inclusion of lambdas (anonymous functions) and lexical closures. A key

element in achieving the zero-cost abstraction principle with respect to first-class functions is the static resolution of all function calls at compile time. Kanagawa does not employ the concept of function pointers nor dynamic dispatch function calls. Similarly lexical closures are implemented in a way that does not incur any overhead compared to regular function arguments.

Functions can be generic in terms of their parameters and return types. The static type system ensures compile-time type correctness while type inference frees the programmer from adding tedious type annotations. A strong static type system helps eliminate whole categories of bugs, which can be especially important in hardware designs.

Kanagawa comes with an extensive standard library that includes common data structures, algorithms and reusable implementations of idiomatic coding patterns. As an illustrative example, Figure 10 shows the implementation of a map-reduce algorithm as a generic higher-order function. It takes user-defined map and reduce functions as arguments, and uses compile-time recursion to implement a reduction tree which has a depth that depends on input size. The function is generic not only in terms of the size and type of its input but also logic depth and pipeline latency required for the map and reduce functions.

The library also contain larger Intellectual Property (IP) blocks, such as RISC-V processor, IEEE-754 floating point arithmetic library, compression codecs, cryptographic algorithms, and many others.

4 ADVANCED KANAGAWA EXAMPLE - SPECULATIVE COUNTIF HISTOGRAM

The implementation of the speculative version of `CountIf Histogram` (option 4) is more complex than the other examples. It takes advantage of various features in Kanagawa for code composability and reuse.

We first implement a generic speculative loop, shown in Figure 11. Similar to the `pipelined_for`, the iterations of `spec_pipelined_for` are concurrently executed by multiple threads. Loop iteration is governed by two user-provided functions: `body`, which speculatively generates results for a given iteration index, and `try_commit`, which commits the results after verifying that they have not been invalidated by another iteration running concurrently. In the event a mis-speculation is detected, the in-flight threads are drained without committing their calculated results. Subsequently, the loop restarts execution on the iteration that failed to commit.

When a thread executes the `atomic` block on line 13, it chooses a speculative loop iteration index `i` that it will execute. The first thread to enter this atomic block after a mis-speculation occurs will see `top_ok` is `false`. This thread is called a "flush thread". A flush thread will set `i` to the lowest iteration index that has not yet been committed (flush threads do not speculate). The following threads will speculate, incrementing loop iteration index values.

After choosing an iteration index, threads execute `body`, which can be arbitrarily complex (*e.g.*, contain deeply nested control flow). Next, threads execute the atomic block on line 24. In this block, a thread will determine if it should commit its results. Results are committed if there is not an unresolved mis-speculation and `try_commit` returns `true`. If `try_commit` returns `false`, then a new mis-speculation has been detected. That mis-speculation will be resolved when a flush thread reaches the atomic block on line 24.

Figure 12 implements `CountIf Histogram` with a call to `spec_pipelined_for`. The `body` function reads input data and speculatively reads a histogram value and computes a new result. `try_commit` compares a current histogram value with the prior speculative read. If these values match then the loop iteration is allowed to complete.

Implementing a computation like this would be difficult with traditional HLS because an HLS compiler would see false dependencies:

```
template <typename C, typename I, typename B>
inline void spec_pipelined_for(
  C count,
  (I)->B body,
  (B)->bool try_commit) {
  pipelined_do<uint8>([count, body, try_commit](uint8 _) {
    static bool top_ok = true;
    static bool bottom_ok = true;
    static I speculative_iteration = 0;
    static C completed_iterations = 0;
    I i;
    bool is_flush_thread = false;
    atomic {
      if (!top_ok){
        speculative_iteration = completed_iterations;
        is_flush_thread = true;
        top_ok = true;
      }
      i = speculative_iteration;
      speculative_iteration++;
    }
    B body_result = body(i);
    bool loop_is_done = false;
    atomic {
      loop_is_done = completed_iterations == count;
      if (is_flush_thread) { bottom_ok = true; }
      if (bottom_ok && !loop_is_done) {
        bool commit_result = try_commit(body_result);
        if (!commit_result) {
          bottom_ok = false;
          top_ok = false;
        } else { completed_iterations++; }
      }
    }
    return !loop_is_done;
  });
}
```

Fig. 11. Speculative loop.

```
void spec_count_if() {
  struct context{ int32 m; float32 prev; float32 sum; };
  spec_pipelined_for(N, [](uint32 i) {
    int32 m = feature[i];
    float32 wt = weight[i];
    float32 prev = hist[m];
    context c = {
      .m = m,
      .prev = prev,
      .sum = m > THRESHOLD ? prev + wt : prev
    };
    return c;
  }, [](context c) {
    bool result = false;
    float32 prev = hist[c.m];
    if (eq(prev, c.prev)) {
      result = true;
      hist[c.m] = c.sum;
    }
    return result;
  });
}
```

Fig. 12. Speculative `CountIf Histogram`.

- Read-after-write and write-after-write hazards on `top_ok`.
- Read-after-write hazards on `hist[m]`.

The Kanagawa compiler does not infer these dependencies because `top_ok` and `hist` are shared variables, and it is up to the programmer to handle inter-thread hazards.

Implementing a computation like this with traditional (*i.e.*, unordered) threads would be difficult because threads could overtake each other while executing `body`. Threads would either need to speculate that they would remain in order (which would frequently not be true), or a costly synchronization mechanism would be needed to restore order after executing `body`.

5 TRANSLATION TO HARDWARE

In this section, we describe a mapping from Kanagawa source code to hardware circuits suitable for FPGA or ASIC implementation. The overall compiler flow is discussed, followed by the implementation of specific constructs.

5.1 Compiler Overview

After parsing, template instantiation, and type-checking, the Kanagawa compiler converts the abstract syntax tree (AST) into a typical imperative Intermediate Representation (IR). Each function in the IR is represented as a control flow graph.

If-conversion [18] is applied to flatten all conditional statements, meaning that they will not break up the control flow graph. This approach is a straightforward method to implement the semantics described in Section 3.5.5. Functions with a single call site that do not have additional attributes (*e.g.*, asynchronous functions) are automatically inlined. Traditional peephole [36] and data-flow analysis [2] optimizations are applied on the IR. The primary purpose of these optimizations is to reduce hardware resource usage.

Next, the operations in each basic block are scheduled into a pipeline and imperative control is lowered to hardware flow control. Finally, Kanagawa utilizes CIRCT [1] to generate SystemVerilog output.

5.2 Control Flow Graph and Pipelines

There is a direct correspondence between the control-flow graph describing the source design and the generated hardware. Figure 13 shows such a control-flow graph for `replicated_count_if`[10]. Each basic block in the control-flow graph corresponds to a pipeline (comprising one or more pipeline stages) in the generated hardware and each edge in the control-flow graph corresponds to communication between pipelines, usually through FIFOs (*i.e.*, first-in-first-out queues of bounded size). A key property of this translation is that it can be described as a recursive composition of translations, specific to each control flow construct. For example, the translation of a `for` loop to hardware can be described independent of the context in which that loop appears.

The computation performed by a single pipeline stage can potentially execute in parallel with the computation performed by all other pipeline stages. The amount of concurrency (*i.e.*, active threads) in a design determines how much of this potential is utilized. At any moment in time, each thread is either assigned to a single pipeline stage or is queued in a FIFO[11]. Two threads cannot be assigned to the same pipeline stage at the same time. The values of local variables associated with a thread are stored in pipeline registers and FIFOs. This state flows through the generated

[10] A `return` statement has been added for clarity.

[11] The values of live local variables are stored in pipeline registers which are input to the pipeline stage where the thread resides, or inside the FIFO where the thread resides.

```
1  void replicated_count_if() {
2    // Basic Block 0
3    pipelined_for(N, [](uint32 i) {
4      // Basic Block 1
5      int32 m = feature[i];
6      float32 wt = weight[i];
7      if (m > THRESHOLD) {
8        uint32 offset = (i % L) * SIZE;
9        [[schedule(L)]] {
10         float32 x = hist[m + offset];
11         hist[m + offset] = x + wt;
12       }
13     }
14   });
15   // Basic Block 2
16   pipelined_for(SIZE, [](uint32 m) {
17     // Basic Block 3
18     float32 sum = 0.0;
19     for (const auto l : L) {
20       // Basic Block 4
21       sum += hist[m + (l * SIZE)];
22     }
23     // Basic Block 5
24     hist[m] = sum;
25   });
26   // Basic Block 6
27   return;
28 }
```

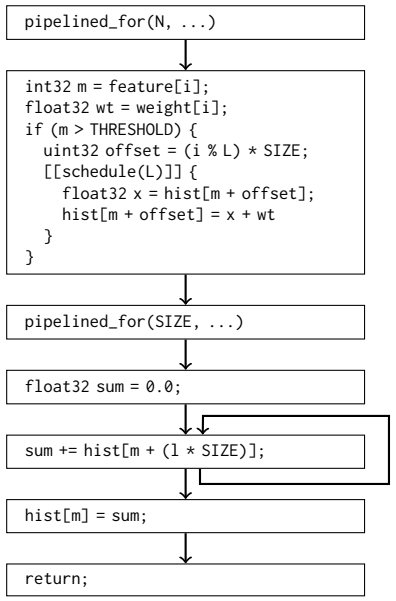

Fig. 13. Control-flow graph of `replicated_count_if`.

```
1  void replicated_count_if() {
2    <body1>
3  }
```

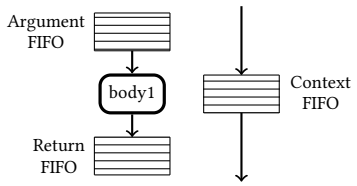

Fig. 14. Hardware realization of a function.

hardware (within and between basic blocks), accompanying their threads as they flow through the control-flow graph of the source design.

5.3 Translation Example

5.3.1 Function. Figure 14 illustrates the hardware realization of a non-inlined function using a fixed-latency pipeline and latency-insensitive FIFOs. The caller enqueues thread-local data into a set of two input FIFOs: argument and context. The argument FIFO holds values that are inputs to the entry basic block of the function. A hidden `call-site-id` argument is also enqueued into the argument FIFO, indicating the source of the function call. The context FIFO holds the values of thread-local variables that are live at the time of the function call. When the function returns, it places return values in a return FIFO. Function returns are implemented with dynamic dispatch and the `call-site-id` parameter determines which return FIFO receives the values. The basic block following each function call site combines the returned values with the stored thread-local variables from the context FIFO to continue execution. Note that the input and output FIFOs are still logically present in the design even if there are no arguments or return values, because they

```
1  void replicated_count_if() {
2    pipelined_for(N, [](uint32 i) {
3      <body2>
4    });
5
6    pipelined_for(SIZE, [](uint32 m) {
7      <body3>
8    });
9  }
```

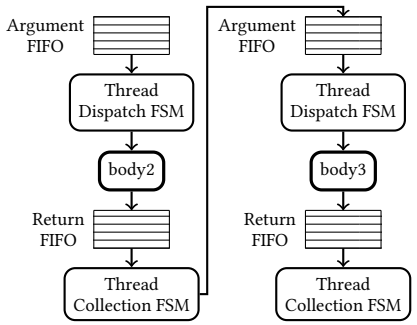

Fig. 15. Hardware realization of a call to a batched function (body1). Context FIFOs omitted for clarity.

```
1  pipelined_for(N, [](uint32 i) {
2    int32 m = feature[i];
3    float32 wt = weight[i];
4    if (m > THRESHOLD) {
5      uint32 offset = (i % L) * SIZE;
6      [[schedule(L)]] {
7        float32 x = hist[m + offset];
8        hist[m + offset] = x + wt;
9      }
10   }
11 });
```

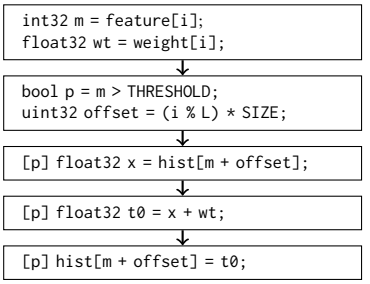

Fig. 16. Hardware realization of a basic block (body2).

```
1  float32 sum = 0.0;
2  for (const auto l : L) {
3    sum += hist[m + (l * SIZE)];
4  }
5  hist[m] = sum;
```

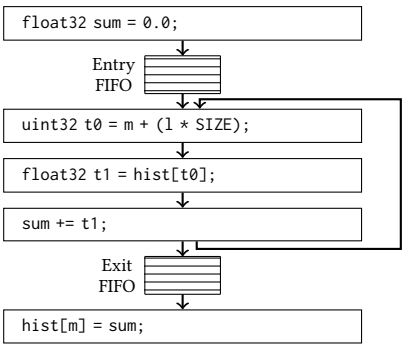

Fig. 17. Hardware realization of a loop (body3).

count the number of threads contained in each FIFO, tracking function entry and return. body1 represents the body of the function, described in Figure 15.

5.3.2 Batched Function. Figure 15 illustrates the hardware implementation of calls to batched functions. The body of each function is wrapped with two finite state machines (FSMs). The thread dispatch FSM dispatches a group of threads into the function body (at a maximum rate of one per clock cycle). The thread collection FSM waits for all threads in a group to finish execution of the function before enqueuing results into the return FIFO. Note that the generated hardware supports nested pipeline parallelism.

```
1 void function_with_wait_for() {
2 <Statements before wait_for>
3 wait_for(expr);
4 <Statements after wait_for>
5 }
```

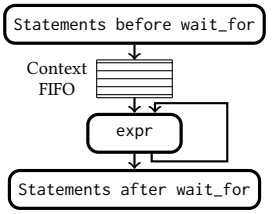

Fig. 18. Hardware realization of `wait_for(expr)`.

5.3.3 Basic Block. Figure 16 illustrates how a basic block (`body2`) translates to a pipeline [12]. The operations which comprise a basic block are scheduled into pipeline stages. A list scheduler is used to find a valid schedule with respect to the following constraints:

- Intra-thread data dependencies are honored for accesses to thread-local variables.
- Wavefront Consistency is preserved for accesses to shared variables.
- User-specified scheduling constraints (`[[schedule()]]`, `atomic`).
- Target logic depth per pipeline stage[13].

Scheduling constraints do not explicitly appear in the scheduled pipeline, because they are satisfied by the schedule. The requirements of Wavefront Consistency are honored by ensuring that the static schedule of accesses to shared variables by a single thread occur in program order. No special effort is needed to maintain inter-thread order because hardware pipelines maintain order by construction. The logic depth constraint is a soft constraint, used to assist in meeting a target operating frequency for the generated circuit, similar to scheduling in traditional HLS tools [13, 56]. It will be violated if necessary to honor the other constraints. It is assumed that the EDA tools consuming the output SystemVerilog will compensate by giving higher priority to cells along paths which violate the logic depth constraint when considering placement, routing, and in the case of an ASIC flow, logic cell threshold voltage and drive strength.

Beyond meeting the hard and soft constraints above, the current compiler does not yet include significant scheduling optimizations, a potential area for improvement. For example, Zhang and Liu [56] uses linear programming to minimize value lifetimes, which reduces the number of pipeline registers.

5.3.4 Sequential Loop. Figure 17 illustrates how a sequential loop [14] (`body3`) is realized. The body of the loop is implemented as a pipeline. Threads enter that pipeline either from the loop entry FIFO, or via a loopback path. When a thread reaches the end of the loop, it is either sent back to the start of the loop body for another iteration or it is sent into the loop exit FIFO, which is read by the pipeline corresponding to the basic block after the loop. Note that while in this case the loop body is a single basic block, in general the loop body can be an arbitrary control-flow graph.

5.3.5 Synchronization - `wait_for()`. As mentioned in Section 3.4, mutex, semaphores, and read-write locks are all Kanagawa standard library abstractions built using the `wait_for()` primitive. Figure 18 shows how `wait_for()` is implemented in hardware. Threads execute up to the `wait_for(expr)` statement then push the values of thread-local variables into a context FIFO. The `wait_for()` condition can be arbitrarily complex, but is evaluated atomically (*i.e.*, only for

[12]Complex expressions have been decomposed into simpler ones and if-conversion has been applied to the `if` statement.
[13]The number of logic operations between registers. This is specified as a compilation parameter outside of the code. It can be swept to fine-tune the design, finding a balance between clock frequency and resource use.
[14]A single thread executes the loop sequentially, but multiple threads can execute the loop concurrently.

the thread at the head of the FIFO) until it evaluates to true. When `expr` is satisfied, the thread continues execution of the statements after the `wait_for()`.

6 EVALUATION

6.1 HLS Comparison

In this section, we evaluate the `CountIf Histogram` using two existing HLS tools and the four Kanagawa implementations presented in the paper. For these experiments, $N = 512$, $SIZE = 32$, $THRESHOLD = 8$, and $L = 8$. For the HLS implementations, we use Xilinx Vitis HLS 2023.1.1 [54] and Dynamatic v2.0.0 [24]. The Dynamatic and Kanagawa implementations are compiled to VHDL and SystemVerilog, respectively, and then synthesized using Xilinx Vivado 2023.1.1. All implementations are synthesized for a 200 MHz clock on the Xilinx Alveo U250 card.

Table 1 shows the resource usage and latency (best- and worst-case depending on data) for the different implementations. Vitis HLS implements a static schedule. Dynamatic [24] implements a dynamic schedule, achieving lower best-case latency but requiring significantly more resources. This overhead occurs because Dynamatic implements a Load-Store Queue (LSQ). This structure alone adds 16,803 LUTs and 3,489 FFs.

The Kanagawa Static Scheduling implementation (Figure 6) has a schedule similar to Vitis HLS, resulting in similar latency. The Kanagawa solution uses slightly more resources due to the control FIFOs and memory bypass logic used, but on larger designs these overheads would likely be negligible. Importantly, Kanagawa allows the programmer to express other solutions in the design space. The Dynamic Scheduling (Figure 7) and Speculative Execution (Figure 12) solutions are able to reach lower best-case latency at the cost of more resources and higher worst-case latency. In this example, the Dynamic Scheduling implementation is always correct in its decision to obtain the lock, but for other applications where the lock may need to be acquired pessimistically, the Speculative Execution approach can lead to faster execution. The Dynamic Scheduling solution has worse worst-case latency than Static Scheduling because the most straightforward implementation of the lock adds latency when a conflict occurs (*i.e.*, the lock is not freed by one thread and acquired by another in the same cycle). A more sophisticated implementation could reduce this overhead. Compared to Dynamatic, the Kanagawa Dynamic Scheduling implementation has better best-case and worse worst-case latency. More importantly, though, the resource overhead of Dynamic compared to Static Scheduling is much more reasonable. The Kanagawa Replicated Histogram implementation from Figure 8 uses the most resources among the Kanagawa solutions, but has data-independent latency. The best-case latency is higher compared to the Dynamic Scheduling and Speculative Execution implementations, but the worst-case is significantly lower. Depending on the programmer's requirements, they are able to describe the most suitable implementation.

6.2 Application Benchmarks

In this section, we demonstrate that the Kanagawa language and toolflow can produce high quality hardware when targeting a variety of ASIC and FPGA platforms. The baseline for comparison is SystemVerilog implementations for several real-world applications. This RTL was either in-production or evaluated-for-production, each having multiple person-months to person-years of experienced engineering invested. These applications were re-implemented using Kanagawa.

The Kanagawa results are highly competitive for traditional quality metrics such as performance and resource requirements. Our results are generally comparable to or sometimes better than the hand-written SystemVerilog since many aspects of design-space exploration can be performed without code changes with Kanagawa and even architectural changes can often be made easily. Although the impact of Kanagawa on developer effort is difficult to accurately capture quantitatively,

Table 1. Xilinx Vitis HLS, Dynamatic, and Kanagawa implementations of `CountIf Histogram`.

	Vitis	Dynamatic	Kanagawa			
			Static Sched.	Dynamic Sched.	Replicated Hist.	Speculative Exec.
LUT	334 (1.00x)	17593 (57.67x)	395 (1.18x)	537 (1.61x)	825 (2.47x)	536 (1.60x)
FF	430 (1.00x)	4319 (10.04x)	608 (1.41x)	877 (2.04x)	1131 (2.63x)	745 (1.73x)
BRAM	1 (1.00x)	1 (1.00x)	1 (1.00x)	1 (1.00x)	2.5 (2.50x)	2.5 (2.50x)
DSP	2 (1.00x)	2 (1.00x)	2 (1.00x)	2 (1.00x)	4 (2.00x)	2 (1.00x)
Best-case Latency	4102 (1.00x)	1540 (0.38x)	4108 (1.00x)	530 (0.13x)	852 (0.21x)	569 (0.14x)
Worst-case Latency	4102 (1.00x)	4106 (1.00x)	4108 (1.00x)	5640 (1.37x)	852 (0.21x)	7723 (1.88x)
Lines of Code	53 (1.00x)	47 (0.89x)	60 (1.13x)	69 (1.30x)	82 (1.55x)	123 (2.32x)

Table 2. ASIC Benchmarks - Post Synthesis Area, Clock Frequency, and Simulation

		Normalized Area	Target Frequency	Normalized Throughput	Lines of Code
Decompress	RTL	1.00x	1.0 GHz	1.00x	8858 (1.00x)
	Kanagawa	1.00x	1.0 GHz	1.04x	4166 (0.49x)
Match Selection	RTL	1.00x	1.2 GHz	1.00x	12400 (1.00x)
	Kanagawa	1.01x	1.2 GHz	0.60x	2600 (0.21x)

Table 3. FPGA Benchmarks - Arria 10 and Stratix 10

		ALM	BRAM	Max Frequency	Norm. Throughput	Lines of Code
SDN-A10	RTL	42280 (1.00x)	1097 (1.00x)	200 MHz	1.00x	13221 (1.00x)
	Kanagawa	41956 (0.99x)	992 (0.90x)	300 MHz	4.00x	5528 (0.42x)
Network Prot.-S10	RTL	10133 (1.00x)	20 (1.00x)	312 MHz	1.00x	7228 (1.00x)
	Kanagawa	9495 (0.94x)	14 (0.70x)	410 MHz	1.45x	2430 (0.33x)
AES GCM-S10	RTL	12034 (1.00x)	161 (1.00x)	480 MHz	1.00x	4663 (1.00x)
	Kanagawa	11978 (1.00x)	163 (1.01x)	480 MHz	1.00x	1233 (0.26x)

Table 4. RISC-V Benchmark - Agilex 7

	ALM	BRAM	Max Frequency	CoreMark/MHz	CoreMark
Nios V/m[15]	1982 (1.00x)	2 (1.00x)	425 MHz	0.40 (1.00x) [14]	170 (1.00x)
Kanagawa	1320 (0.67x)	2 (1.00x)	320 MHz	1.06 (2.65x)	339 (1.99x)

we use lines-of-code as a proxy, roughly representing the complexity of initial development, testing, debugging, and future maintenance. The Kanagawa implementations are only 0.2x to 0.5x as many lines of code as the SystemVerilog implementations.

6.2.1 ASIC Benchmarks. To show the results for an ASIC platform, we examine two benchmarks targeting two different contemporary technology nodes. The first benchmark is a DEFLATE decompression module with a 16-bit input and 64-bit output datapath. This application is control-intensive, primarily because each byte of input can produce a variable number of output bytes. The second application implements match selection for compression. A distinctive characteristic of this benchmark is that, as per the baseline architectural design, it is largely single-threaded. The handwritten SystemVerilog and the SystemVerilog generated by the Kanagawa compiler were synthesized using the same production toolflow: a process-specific standard-cell library, Synopsys 2022.03 synthesis tools for regular logic, and a process-specific memory compiler for memory blocks. Given the nature of ASIC projects, the target clock frequency for both applications were part of the original design

[15]Quartus Pro Prime generates encrypted IP for the Nios V/m, the code is not viewable to users.

specification. The throughput of all implementations after synthesis was tested using SystemVerilog simulation and internal performance benchmarks.

As shown in Table 2, the Kanagawa implementation for both applications met timing and had area nearly identical to the handwritten RTL. For the decompression benchmark, the throughput of the Kanagawa implementation (an average of the performance across an internal suite of tests) was slightly faster than the handwritten RTL, due to minor micro-architectural differences. Beyond the slight performance advantage, the Kanagawa code was also less than half the length of the handwritten SystemVerilog for decompression and only 0.21x for match selection.

However, the throughput of the match selection benchmark was only 0.6x the handwritten RTL. This occurred for two reasons. First, Kanagawa excels at expressing concurrent pipelined computation, but the architectural design and interface of this application was fundamentally single-threaded. Second, the control structures that are generated by the Kanagawa compiler can increase end-to-end latency. Single-threaded computations are inherently latency sensitive, so these factors combine to decrease performance. These results are from a preliminary implementation, before planned latency and architectures optimizations were implemented.

6.2.2 FPGA Benchmarks. To show the results when targeting an FPGA, we examined three additional benchmarks implemented on two different Intel FPGAs (Arria 10 and Stratix 10). The first benchmark is a Software-Defined Networking (SDN) accelerator, responsible for performing 50Gbps packet parsing and modification on an Arria 10. This application is control-intensive, primarily due to the diversity of packet types it needs to handle. The second benchmark is a network protocol engine that generates packet headers and computes CRCs on a Stratix 10. This application was written in Kanagawa using highly parameterized code to enable architectural design-space exploration. The third benchmark is AES-256 GCM with a 128-bit input and output datapath on a Stratix 10. This is a dataflow-style application that must run at a comparatively high frequency, which makes effective pipelining important.

As with the handwritten ASIC benchmarks, the baseline for the FPGA benchmarks was in-production or evaluated-for-production SystemVerilog. Both the handwritten and Kanagawa implementations were synthesized using the same toolflow - Quartus Pro 19.1 for the Arria 10 and Quartus Pro Prime 21.4 for the Stratix 10. However, unlike the ASIC tests we did not fix the same timing constraints for both codebases. Rather, taking advantage of the much more flexible nature of FPGAs and the ease of generating and using different clocks, we performed a separate timing sweep for each implementation and report the best achievable clock rate for four compilation seeds.

As shown in Table 3, the Kanagawa implementation of the SDN and protocol engine benchmarks required fewer ALM and block RAM resources while able to run at higher clock rates than the handwritten SystemVerilog. The source code was also 0.42x and 0.33x the length, respectively.

Perhaps most importantly though, the throughput of these two applications for internal benchmarks was higher. In the case of the SDN accelerator, this performance advantage was primarily due to two factors. First, the clock rate was 1.5x higher. The amount of pipelining in the Kanagawa implementation could be changed without source code modifications, facilitating the search for a balance between resource utilization and clock rate. This increase in clock rate was directly responsible for 1.5x better throughput. Second, the ease of design-space exploration allowed the Kanagawa developers to find a higher throughput micro-architecture, requiring fewer cycles to process tasks. This was roughly responsible for an additional 2.6x better throughput. Combined, the Kanagawa implementation of the SDN accelerator was 4.0x faster than the handwritten RTL. Similar clock rate and latency optimizations occurred for the protocol engine benchmark, combining for 1.45x better performance compared to the RTL implementation.

The AES GCM benchmark was nearly identical to the hand-written implementation in terms of both resource requirements and performance, though only requiring 0.26x the lines of code. That said, the Kanagawa implementation did require two additional block memories due to slightly different FIFO arrangements between processing blocks.

We also use Kanagawa to implement a RISC-V processor and compare with a slightly different commercially available implementation, both mapped to Intel Agilex 7 FPGA using Quartus Pro Prime 22.4. The Kanagawa RISC-V processor implements a five-stage pipeline for the RV32I ISA. The Intel Nios V/m also implements a five-stage pipeline, but for the RV32IZicsr ISA, which adds control and status register instructions. Looking at the performance and resource requirements in Table 4, it is obvious that the processors have different architectural goals. Although this comparison is imperfect, it shows that Kanagawa can be used to implement a performant RISC-V processor with reasonable resource requirements. This is significant because this is generally not true for traditional HLS tools[34]. For example, without constructs to explicitly define concurrency, it can be difficult to express features such as control and data hazard handling, which are specific to how the processor is divided into pipeline stages.

6.2.3 Academic Research. Kanagawa has also been used in two recent papers. The first was Riazi et al. [48], which used Kanagawa to build an FPGA implementation of a Fully Homomorphic Encryption algorithm. Beyond implementing the cryptographic system itself (*e.g.*, polynomial modulo math, tuning the design to the available DSP and memory blocks), system-level optimizations were added (*e.g.*, batching data and multi-buffering communication with the host machine to hide PCIe latency). The Kanagawa FPGA implementation was faster than the Microsoft software library implementation. The second project to use Kanagawa was Bisheh-Niasar et al. [4], which accelerated a post-quantum key encapsulation mechanism. The authors built heavily parameterized Kanagawa code, allowing them to easily generate customized platforms. The Kanagawa FPGA implementation was compelling compared to previous work and required less than 0.20x the development time of a manual RTL implementation.

7 RELATED WORK

In this section, we qualitatively compare Kanagawa with other high-level hardware languages.

7.1 High-level HDL

Bluespec SystemVerilog [44] and Kôika [5] offer more comprehensible semantics than RTL languages by modeling hardware execution as the application of a set of atomic rules. Each computational step logically selects exactly one rule and applies it, giving one-rule-at-a-time (ORAAT) semantics. To improve hardware performance, dynamic scheduling is used to enable multiple rules to run in parallel, when parallel execution is equivalent to ORAAT execution.

Languages like Chisel [3] embed hardware description within a software language. Constructs from the host language can be used to algorithmically generate hardware descriptions.

While these languages raise the abstraction level compared to RTL, they still retain the low-level hardware programming paradigm. State elements are explicitly specified in these languages, along with transition functions or rules that complete on short time scales. Source code in these languages is less abstract than typical Kanagawa source and does not use imperative control flow.

7.2 Traditional HLS

As discussed in Section 6.1, high-level synthesis tools such as Vitis HLS [54], Intel HLS Compiler [22], LegUp [7], Catapult-C [49], and Cadence Stratus [6] take on the difficult task of transforming sequential source into parallel hardware.

```
T InsertionCell(hls::stream <int> &IN, hls::stream <int> &OUT){
  static int CURR_REG=0;
  int IN_A = IN.read();
  if(IN_A > CURR_REG) {
      OUT.write(CURR_REG);
      CURR_REG = IN_A;
  } else
      OUT.write(IN_A);
  return CURR_REG;
}
void InsertionSort(hls::stream <T> &IN, hls::stream <T> &OUT){
#pragma HLS DATAFLOW
  hls::stream <T> out1 , out2 , out3;
  // Function calls;
  InsertionCell1(IN, out1);
  InsertionCell2(out1, out2);
  InsertionCell3(out2, out3);
  InsertionCell4(out3, OUT);
}
```

Fig. 19. Sorting modules implemented with message passing.

7.2.1 Message Passing. An execution model supported by HLS tools is explicit task parallelism described as Kahn Process Networks [27] or Communicating Sequential Processes [21]. Tasks communicate through message passing. Each state element (except for message queues) is assigned to a specific task which is the only task that can directly access that state element. Concurrency between separate tasks is straightforward to map to coarse-grain hardware parallelism, while automatically extracting fine-grained parallelism from within a single sequential task remains difficult. This leads seasoned HLS engineers to express hardware with a fine-grained decomposition into many (small) task-parallel modules.

As the decomposition becomes more fine-grained, it becomes difficult to assign data structures to tasks. For example, each pipeline stage in a processor can be described as a separate task, sending and receiving messages to and from other stages. However, state elements such as the register file and program counter must be assigned to a single task which can access them directly. This can lead to sub-optimal implementations where instruction fetch, decode, and register reads all occur in a single task [34]. Another example is a network packet processor modeled as a set of parallel tasks which send packets as messages to one another. It is impossible to atomically update configuration data structures distributed throughout the design without draining all in-flight packets [53].

Figure 19 shows idiomatic examples of message passing HLS from the literature [35]. Matai et al. [35] begins with a discussion of how HLS tools cannot efficiently implement sorting algorithms described imperatively, and demonstrates how message passing produces better results. Distinguishing features of message passing HLS include:

- Function parameters are always queues (`hls::stream` in this case).
- Function calls occur unconditionally.
- Imperative control flow is used sparingly, with control implemented as data flow.

Kanagawa thread synchronization and communication mechanisms allow the expression of a wider range of designs than message passing HLS because Kanagawa threads can access shared state directly.

7.2.2 Shared Memory Concurrency. There has been some work on shared memory concurrency for HLS. LegUp [12] supports expressing coarse-grain concurrency with threads that behave like typical software threads. Threads are mapped to parallel hardware modules which can communicate through shared state. Ramanathan et al. [46] and Ramanathan et al. [45] describe modifications

to LegUp to support the semantics of C atomics. OpenCL supports shared memory concurrency through lightweight threads executing computational kernels, and synchronizing with constructs which map well to GPUs [15].

Rather than inheriting shared memory concurrency semantics from languages that map well to CPUs or GPUs, Kanagawa introduces semantics tailored for the reliable generation of efficient hardware.

7.2.3 Dynamic HLS. Dynamic HLS [24] implements the same sequential semantics as traditional HLS, but moves much of the responsibility of scheduling from the compiler to the hardware. Handshaking connections and load-store queues [23] in the generated hardware ensure that operations run in an order that maintains sequential semantics. Dynamic scheduling outperforms static scheduling in cases where a static scheduler has to make overly conservative decisions to accommodate worst-case behavior. As with traditional HLS, the constraint of implementing a sequential source language can lead to situations where the compiler must perform non-trivial transformations [9, 11] in order to achieve high throughput.

As we have shown with the `CountIf Histogram` example, Kanagawa allows the programmer to explicitly express a variety of solutions, each having different design trade-offs.

7.3 Spatial

Spatial [28] is an imperative, domain-specific language for application accelerators. Control is expressed with the composition of a set of control structures including parallel patterns like `Foreach` and `Reduce`. Like traditional HLS compilers, the Spatial compiler only pipelines loops when the compiler can ensure that the behavior of a pipelined loop matches the behavior of sequential implementation. Kanagawa allows the programmer to pipeline arbitrary code and gives the programmer scheduling constraints and synchronization tools to ensure correctness.

7.4 PDL

PDL [55] models a processor as a stream of threads flowing down a manually-scheduled pipeline, with one thread handling the execution of a single instruction. Hazards are resolved with the help of synchronization objects which are referenced by PDL code and implemented in RTL. Kanagawa generalizes this notion of multi-threading in the following ways:

- Kanagawa designs compile to hardware comprising multiple pipelines, with imperative control flow dictating how threads move between pipelines.
- Kanagawa designs are automatically pipelined, subject to user-specified constraints.
- Kanagawa is sufficiently general that synchronization objects can be implemented in Kanagawa directly.

7.5 Dahlia

Dahlia [41] is a novel HLS language in which the central tenet is that it only attempts to compile programs with predictable performance to hardware. It accomplishes this by defining a type system that, when met, allows the compiler to directly map computations to hardware. Programs that include potentially complex or unpredictable constructs, such as arbitrary array indexing, fail type checking. Kanagawa also aims for predictable performance, but achieves this by requiring the programmer to explicitly describe concurrency.

7.6 Calyx

Calyx [43] is an intermediate language which combines structural and imperative composition. Data paths and connectivity are described structurally, while the control logic for each module is

defined imperatively. Calyx and Kanagawa model imperative concurrency differently. In Calyx, disjoint subsets of code ("groups") can execute in parallel, whereas in Kanagawa multiple threads can be executing the same code concurrently (each with separate thread-local variables).

7.7 Filament

Filament [42] is a hardware description language with timeline types. The Filament type checker can catch bugs related to the composition of hardware modules such as the possibility of reading the output of a module at the incorrect clock cycle. Like other higher level HDLs, Filament raises the abstraction level compared to RTL, but is still low level relative to Kanagawa.

8 LIMITATIONS OF KANAGAWA

In our experience, Kanagawa is an easy-to-use, highly capable language with distinct advantages over existing alternatives. For example, several interns with only prior software experience have been able to learn Kanagawa and become proficient enough to create efficient FPGA implementations of non-trivial applications in fewer than 12 weeks. That said, hardware development is a long-standing and multi-faceted problem. We cannot claim that Kanagawa is the most appropriate tool for all situations. Compared to traditional HLS, Kanagawa relies on the programmer to discover and express concurrency with threads. Single-threaded code results in poor throughput and multi-threaded code often requires explicit synchronization to ensure correct behavior. Although Wavefront Consistency makes reasoning about concurrency easier than in typical multi-threaded programs, subtle synchronization errors can result in elusive bugs. We believe that formal verification could help to catch these bugs.

Kanagawa is also not as expressive as RTL. For example, tight coordination between threads is only possible for ordered threads that are executing the same statements. Threads executing disjoint regions of code cannot be as tightly coordinated. Also, as shown in the Match Selection benchmark in 6.2.1, it can be difficult to precisely control the latency of the generated output. This can affect single-threaded performance.

9 CONCLUSION

Kanagawa empowers programmers to build hardware designs with a highly expressive language, leveraging familiar programming language features that make code naturally composable and reusable. Kanagawa supports an execution model and a memory consistency model that make generating efficient hardware simple for the compiler and reduces the complexity of reasoning about highly concurrent code. In our experience, the language is easy to learn and new developers can become productive within weeks. Kanagawa also permits developers to iterate on complex designs with low friction. They can rapidly experiment with architectural choices to optimize performance and resource requirements. We have shown that Kanagawa can produce high-quality circuits in a production environment.

Looking ahead, generative AI for software development is a nascent area showing immense promise. We believe that Kanagawa is well poised as a platform for generative hardware development. The same characteristics that make it excel for manual development (*e.g.*, compact syntax, zero-cost abstractions, functional composability, and a strong, static type system) make it a highly practical target for large language AI models. Both human and AI Kanagawa developers can focus on the higher-level functional semantics because the Kanagawa compiler takes care of lower-level implementation details. This encourages more advanced, efficient, powerful, and reliable hardware solutions.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable feedback, suggestions, and insights that have significantly improved the quality of this paper. We also express our gratitude to esteemed academic partners for their collaboration and intellectual contributions, which have been crucial in developing this novel hardware language. The authors thank Mark Hill for his in-depth discussions and suggestion to frame the description of the Kanagawa semantics as a memory consistency model.

REFERENCES

[1] 2023. CIRCT. https://circt.llvm.org/. (accessed: 11.8.2023).
[2] F. E. Allen and J. Cocke. 1976. A Program Data Flow Analysis Procedure. *Commun. ACM* 19, 3 (mar 1976), 137. https://doi.org/10.1145/360018.360025
[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. https://doi.org/10.1145/2228360.2228584
[4] Mojtaba Bisheh-Niasar, Daniel Lo, Anjana Parthasarathy, Blake Pelton, Bharat Pillilli, and Bryan Kelly. 2023. PQC Cloudization: Rapid Prototyping of Scalable NTT /INTT Architecture to Accelerate Kyber. In *2023 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. 1–7. https://doi.org/10.1109/PAINE58317.2023.10318029
[5] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965
[6] Cadence Design Systems Inc. 2023. Stratus High-Level Synthesis. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html. (accessed: 11.2.2023).
[7] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 24 (sep 2013), 27 pages. https://doi.org/10.1145/2514740
[8] Bradford L Chamberlain, E Christopher Lewis, and Lawrence Snyder. 1999. Array language support for wavefront and pipelined computations. In *Workshop on Languages and Compilers for Parallel Computing*. Citeseer.
[9] Jianyi Cheng, Lana Josipović, George A. Constantinides, and John Wickerson. 2022. Dynamic Inter-Block Scheduling for HLS. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 243–252. https://doi.org/10.1109/FPL57034.2022.00045
[10] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2021. Exploiting the Correlation between Dependence Distance and Latency in Loop Pipelining for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 341–346. https://doi.org/10.1109/FPL53798.2021.00066
[11] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Dynamic C-Slow Pipelining for HLS. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–10. https://doi.org/10.1109/FCCM53951.2022.9786096
[12] Jongsok Choi, Stephen Brown, and Jason Anderson. 2013. From software threads to parallel hardware in high-level synthesis for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*. 270–277. https://doi.org/10.1109/FPT.2013.6718365
[13] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *2006 43rd ACM/IEEE Design Automation Conference*. 433–438. https://doi.org/10.1145/1146909.1147025
[14] Intel Corporation. 2023. *Nios® V Processor Reference Manual - Updated for Intel® Quartus® Prime Design Suite: 23.3*.
[15] Tomasz S Czajkowski, David Neto, Michael Kinsner, Utku Aydonat, Jason Wong, Dmitry Denisenko, Peter Yiannacouras, John Freeman, Deshanand P Singh, and Stephen D Brown. 2012. OpenCL for FPGAs: Prototyping a compiler. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. The World Congress in Computer Science, Computer Engineering, & Applied Computing.
[16] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. 2017. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 189–194. https://doi.org/10.1145/3020078.3021754
[17] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. 2020. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4229–4239. https://doi.org/10.1109/TCAD.2020.3012866

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041
[19] Jean-Michel Gorius, Simon Rokicki, and Steven Derrien. 2022. SpecHLS: Speculative Accelerator Design Using High-Level Synthesis. *IEEE Micro* 42, 5 (2022), 99–107. https://doi.org/10.1109/MM.2022.3188136
[20] Gerald Jay Sussman Harold Abelson. 1985. *Structure and Interpretation of Computer Programs*. The MIT Press.
[21] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (aug 1978), 666–677. https://doi.org/10.1145/359576.359585
[22] Intel Corporation. 2023. Intel High Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. (accessed: 11.1.2023).
[23] Lana Josipovic, Philip Brisk, and Paolo Ienne. 2017. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 125 (sep 2017), 19 pages. https://doi.org/10.1145/3126525
[24] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/3174243.3174264
[25] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative Dataflow Circuits. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 162–171. https://doi.org/10.1145/3289602.3293914
[26] Guy Lewis Steele Jr. 1976. LAMBDA: The Ultimate Declarative. (1976), 48. http://hdl.handle.net/1721.1/6091
[27] Gilles Kahn. 1974. The semantics of a simple language for parallel programming, IFIP 74.
[28] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379
[29] Sun-Yuan Kung, Arun, Gal-Ezer, and Bhaskar Rao. 1982. Wavefront Array Processor: Language, Architecture, and Applications. *IEEE Trans. Comput.* C-31, 11 (1982), 1054–1066. https://doi.org/10.1109/TC.1982.1675922
[30] L. Lamport. 1997. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.* 46, 7 (1997), 779–782. https://doi.org/10.1109/12.599898
[31] Henry Ledgard. 1983. *Reference Manual for the ADA Programming Language*. Springer-Verlag, Berlin, Heidelberg.
[32] Gai Liu, Mingxing Tan, Steve Dai, Ritchie Zhao, and Zhiru Zhang. 2017. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 11 (2017), 1817–1830. https://doi.org/10.1109/TCAD.2017.2664067
[33] Junyi Liu, John Wickerson, and George A. Constantinides. 2016. Loop Splitting for Efficient Pipelining in High-Level Synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 72–79. https://doi.org/10.1109/FCCM.2016.27
[34] Paolo Mantovani, Robert Margelli, Davide Giri, and Luca P. Carloni. 2020. HL5: A 32-bit RISC-V Processor Designed with High-Level Synthesis. In *2020 IEEE Custom Integrated Circuits Conference (CICC)*. 1–8. https://doi.org/10.1109/CICC48029.2020.9075913
[35] Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu, Amin Abazari, and Ryan Kastner. 2016. Resolve: Generation of High-Performance Sorting Architectures from High-Level Synthesis. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*. Association for Computing Machinery, New York, NY, USA, 195–204. https://doi.org/10.1145/2847263.2847268
[36] W. M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (jul 1965), 443–444. https://doi.org/10.1145/364995.365000
[37] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (feb 1991), 21–65. https://doi.org/10.1145/103727.103729
[38] Antoine Morvan, Steven Derrien, and Patrice Quinton. 2011. Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion. In *2011 International Conference on Field-Programmable Technology*. 1–10. https://doi.org/10.1109/FPT.2011.6132715
[39] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. 2020. *A Primer on Memory Consistency and Cache Coherence* (2nd ed.). Morgan & Claypool Publishers.
[40] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue* 6, 2 (mar 2008), 40–53. https://doi.org/10.1145/1365490.1365500
[41] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM*

SIGPLAN Conference on Programming Language Design and Implementation (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[42] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (jun 2023), 25 pages. https://doi.org/10.1145/3591234

[43] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[44] R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[45] Nadesh Ramanathan, George A. Constantinides, and John Wickerson. 2018. Concurrency-Aware Thread Scheduling for High-Level Synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 101–108. https://doi.org/10.1109/FCCM.2018.00025

[46] Nadesh Ramanathan, Shane T. Fleming, John Wickerson, and George A. Constantinides. 2017. Hardware Synthesis of Weakly Consistent C Concurrency. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/3020078.3021733

[47] B. R. Rau and C. D. Glaeser. 1981. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *SIGMICRO Newsl.* 12, 4 (dec 1981), 183–198. https://doi.org/10.1145/1014192.802449

[48] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1295–1309. https://doi.org/10.1145/3373376.3378523

[49] Siemens EDA. 2023. Catapult High-Level Synthesis and Verification. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/. (accessed: 11.1.2023).

[50] Guy Lewis Sussman Steele, Jr. and Gerald Jay. 1976. Lambda: The Ultimate Imperative. (1976), 41. http://hdl.handle.net/1721.1/5790

[51] Bjarne Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley Professional.

[52] Michael Wolfe. 1986. Loop Skewing: The Wavefront Method Revisited. *Int. J. Parallel Program.* 15, 4 (oct 1986), 279–293. https://doi.org/10.1007/BF01407876

[53] Xilinx Inc. 2023. Abstract Parallel Programming Model for HLS. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Abstract-Parallel-Programming-Model-for-HLS. (accessed: 11.1.2023).

[54] Xilinx Inc. 2023. Vitis HLS. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html. (accessed: 11.1.2023).

[55] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. 2022. PDL: A High-Level Hardware Design Language for Pipelined Processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 719–732. https://doi.org/10.1145/3519939.3523455

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

Received 2023-11-16; accepted 2024-03-31