Revet: A Language and Compiler for Dataflow Threads

Alexander C. Rucker†, Shiv Sundram†, Coleman Smith†, Matthew Vilim‡, Raghu Prabhakar‡,
Fredrik Kjølstad†, and Kunle Olukotun†
Stanford University†, SambaNova Systems, Inc.‡
acrucker@alumni.stanford.edu, shiv1@stanford.edu, csmith89@stanford.edu, matthew.vilim@sambanova.ai,
raghu.prabhakar@sambanova.ai, kjolstad@cs.stanford.edu, and kunle@stanford.edu

2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00016

***Abstract*—Spatial dataflow architectures such as reconfigurable dataflow accelerators (RDA) can provide much higher performance and efficiency than CPUs and GPUs. In particular, vectorized reconfigurable dataflow accelerators (vRDA) in recent literature represent a design point that enhances the efficiency of dataflow architectures with vectorization. Today, vRDAs can be exploited using either hard-coded kernels or MapReduce languages like Spatial, which cannot vectorize data-dependent control flow. In contrast, CPUs and GPUs can be programmed using general-purpose threaded abstractions.**

The ideal combination would be the generality of a threaded programming model coupled with the efficient execution model of a vRDA. We introduce Revet: a programming model, compiler, and execution model that lets threaded applications run efficiently on vRDAs. The Revet programming language uses threads to support a broader range of applications than prior parallel-patterns approaches, and our MLIR-based compiler lowers this language to a generic dataflow backend that operates on streaming tensors. Finally, we show that mapping threads to dataflow out-performs GPUs, the current state-of-the-art for threaded accelerators, by 3.8×.

I. Introduction

Spatial dataflow accelerators eliminate the overheads of modern von Neumann machines, including instruction fetch, dynamic scheduling, caching, and speculation, by statically scheduling computation. In particular, vectorized reconfigurable dataflow accelerators (vRDA) [16], [17], [34]–[36] are a new class of hardware that uses a large grid of compute and memory to exploit both *vector* (SIMD) and *pipeline* parallelism. Furthermore, coarse-grained pipelining enables on-chip kernel fusion: a program can be pipelined across the entire chip without intermediate materializations to DRAM. Overall, vRDAs maximize compute and memory bandwidth while lowering overhead.

However, vRDAs are limited by their programming model. Currently, users program vRDAs with either hard-coded kernels directly expressed as low-level machine configurations [41], [42] or hierarchical MapReduce code (e.g., the Spatial [20] language). Programming vRDAs with libraries of hard-coded kernels limits them to that small set of operations and prevents efficient fusion across operations, while prior MapReduce models limit programs to parallel loops over a control-flow-free inner loop body. This eliminates algorithms with any data-dependent inner iteration because they require control flow, exceeding MapReduce's limits.

Although vRDAs are more efficient, GPUs currently dominate the compute-accelerator market. Imperative programming models, including the *threaded* SIMT model used by GPUs [5], are more powerful than MapReduce because they support parallelism over data-dependent structured control-flow like `if` statements and `while` loops. This generality gap between MapReduce and SIMT inspires our key question: can we program vRDAs with thread-based programming languages?

In this paper, we introduce Revet, a compiler that uses dataflow threading [41] to map a simple, yet expressive imperative language to vRDAs. Dataflow threads increase the flexibility of vRDAs by moving control-flow decisions from a specialized control plane (with extremely limited bandwidth) to the faster data plane, which executes them as spatial routing decisions. Revet introduces a control-flow to dataflow lowering pass supporting a variety of control-flow constructs including `while` loops, `if` statements, and nested parallel `foreach` loops. In turn, this control flow enables more asymptotically efficient algorithms to solve user-facing problems.

We describe several optimizations for Revet. These include efficient scratchpad orchestration for common access patterns, like data-dependent sequential reads and writes, with the ease of accessing caches. Other optimizations lower the number of compute units needed for the resulting dataflow. Finally, we describe the composable abstract machine model that Revet targets, which is based on the Aurochs vRDA [41]. By defining a machine model that uses a hierarchy of barriers to provide guarantees about control flow, Revet makes it possible to compile arbitrary code that could not be compiled on Aurochs. This includes code with nested `while` loops, parallel-patterns `foreach` loops inside dataflow threads, and dataflow threads inside `foreach` loops—all with guaranteed correctness.

Our key contributions are:

1) a vRDA abstract machine that adds hierarchical parallelism to Aurochs's per-lane control flow (Section III),
2) a language that captures parallelism and memory locality for applications with nested data-dependent control flow in a way that can map complex programs to vRDAs (Section IV), and
3) a compiler that optimizes and lowers our new language to streaming, vectorized, and pipelined dataflow on our abstract machine (Section V).

2378-203X/24/$31.00 ©2024 IEEE
DOI 10.1109/HPCA57654.2024.00016

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

We demonstrate Revet's flexibility by compiling a variety of applications drawn from data analytics, data structure traversal, geospatial analytics, and string analytics, none of which can be expressed in MapReduce. We use cycle-accurate simulation to show that Revet outperforms a V100 GPU by a geomean 3.8× on a 4.3× smaller vRDA, resulting in an area-adjusted speed up of 16×. We also analyze how Revet uses the vRDA's hardware units and demonstrate that our optimization passes enable significantly more efficient use of hardware resources.

II. Background

Revet compiles to a machine model based on Aurochs [41], a vRDA for dataflow threads. The current state of the art programming model for vRDA compilation is Spatial [20], which uses a parallel-patterns approach. Revet retains Spatial's support for explicit, user-facing parallelism while using dataflow threads to improve the flexibility of vectorization and add support for sequential control flow within parallel sections.

a) Parallel-Patterns Dataflow: Plasticine [35] is a vRDA that maps programs to a grid of compute and memory resources, as shown in Figure 1. Specifically, Plasticine is a grid of vectorized compute units (CUs) and memory units (MUs), arranged in a checkerboard pattern and surrounded by DRAM address generators (AGs). The CUs, MUs, and AGs are connected by a programmable network guaranteeing exactly-once, in-order delivery [44], and the entire chip runs at a fixed clock frequency. Dataflow architectures are frequently network-limited [44], so efficiently using network resources is critical. Plasticine's [35] network and per-unit input buffers are a mixture of vector (512 b) and scalar (32 b) resources. To maintain 100% dataflow throughput, each unit uses input buffers at the pipeline head to account for network path-length imbalances.

SARA [45] lowers the Spatial [20] language to streaming dataflow *contexts,* which were each mapped to one or more CUs, MUs, or AGs based on their size. First, SARA maps the instructions inside the basic block to pipeline stages. Then, SARA extracts control logic, which for parallel-patterns code is a nested series of counter-driven loops, and maps it to specialized control hardware. Finally, SARA maps data values associated with innermost loops to vector dataflow while mapping other values to scalar dataflow. SARA's controllers track one instance of control state per CU and can make one control-flow decision per cycle. This control-plane/data-plane split results in a control plane with extremely limited bandwidth.

Furthermore, SARA's controllers are used to interpret data boundaries in on-chip links. For example, if a link transmits three data elements a, b, c it is up to the receiver to determine the correct allocation of these data to loop iterations. Thus, the transmitted data for $[a], [b, c]$ and $[a, b], [c]$ is identical. For irregular programs, the complexity introduced by transmitting controller information can be significant.

b) Aurochs: Aurochs [41] was introduced to support irregular database algorithms, like hash-joins and index traversal, which cannot be mapped to Plasticine's rigid parallel-patterns

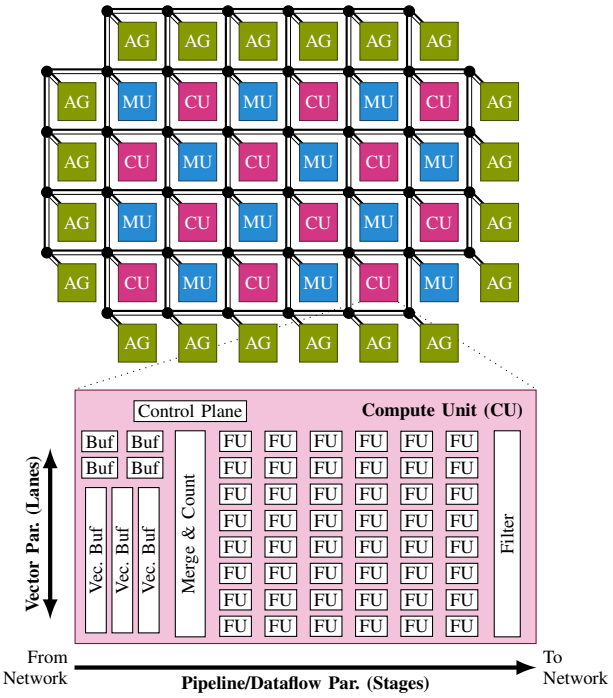


Fig. 1. A diagram showing Revet's layout and vector/pipeline parallelism across functional units (FUs) within a compute unit [41]. For simplicity, only eight lanes are shown.

model. Aurochs extends Plasticine's pipeline with new logical functionality [36], [42] while keeping a linear layout. To support merge-sorting and inner joins, the input buffers can be interleaved using a merge unit and broadcast using a counter-based control unit. After the pipeline-head logic has permuted the inputs, data then enters the pipeline, where six stages (each with an element-wise, statically-mapped instruction over 16 lanes) process it. Finally, data exits through an optional filtering stage to the network.

Aurochs introduced the *dataflow threads* model, where every *thread* is simply a set of live values that are kept together in the pipeline. Using the filtering stage and subsequent merging, Aurochs emulates basic control flow on these threads. For example, filters can select between **`if`** and **`else`** branches, a data-dependent subset of threads for recirculation in a **`while`** loop, or a set of threads to be dropped entirely. When routing decisions send a thread's live values to a CU, the CU does its computation and yields a modified set of values representing the new thread state.

Aurochs has two key limitations that Revet addresses. First, Aurochs can not use the lower-overhead scalar network because its dataflow threads lack the *hierarchy* needed to associate a scalar with vectorized data. This can lead to a shared value being copied into multiple threads and recirculated through the network repeatedly instead of being sent once and broadcast at the receiver. In turn, this limitation prevents Aurochs from enabling fine-grained parallel patterns *within* a data flow thread.

Second, Aurochs's ad hoc filtering mechanism lacks the

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

ability to support arbitrary compiled control flow, because the machine model lacks an efficient mechanism for grouping and synchronizing threads. Specifically, Aurochs used a timeout mechanism for synchronizing threads inside a recirculating region—if no activity is observed at the loop-head block for a given number of cycles, then Aurochs assumes that all threads have exited the loop body. However, this mechanism breaks down for nested loops because a thread may be recirculating inside an inner loop for an arbitrary amount of time.

III. A Generic Model of Dataflow

Having introduced Aurochs [41], the initial implementation of dataflow threads, we now discuss how Revet formalizes and extends the dataflow threads abstraction, starting with the on-chip dataflow format.

A. Structured-Link Tensor Format (SLTF)

Parallel-patterns dataflow operates on tensors coordinated using synchronized, counter-chain-driven controllers. However, in Revet, senders and receivers do not share synchronized controllers, so the sender must *encode* its control decisions—and those made by upstream senders—so that they can be communicated to downstream units. Encoding control-flow data can be done by selectively sending data to only some receivers or by changing metadata. Revet uses a structured-link tensor format (SLTF) to encode this control metadata. The SLTF uses a small number of additional bits per on-chip link to count the number of elements being sent and encode a barrier indicating the number of nested loops that are being terminated. Then, when a parallel-patterns reduction operation receives a loop termination, it sends the current value of the reduction value and resets the accumulator to its initial value.

a) Embedding Control with Data: Revet uses a structured on-chip data representation to capture live variables inside threads and hierarchy information across groups of threads. The live variables within each thread are sent as parallel tensors, where ordering associates live values across tensors. Hierarchy is encoded as done-tokens, or barriers (Ω_n), to indicate the end of dimensions. For brevity, we represent barriers as Ω_n to indicate the end of dimension n, starting with Ω_1 to indicate the end of the lowest dimension.

Intuitively, the hierarchy metadata represents ragged k-dimensional tensors, where the number of dimensions is fixed but each dimension can have a variable size. For example, the two-dimensional tensor [[0, 1], [2]] would be encoded as [0, 1, Ω_1, 2, Ω_2] in the on-chip network. Here, Ω_2 implies an Ω_1, after element 2, due to the tensor dimensions forming a strict hierarchy and there being scalar elements in the tensor.

Adding this hierarchy to on-chip links using an out-of-band encoding is inexpensive. We assume that at most one barrier can be sent per on-chip vector and that $n \leq 15$. This is far lower than observed loop nesting levels and less than 1% overhead relative to a 512-bit (16×32-bit) vector (assuming the vector link contains four bits for the barrier level and a length encoding that overlaps the last lane except for one bit).

b) Composability: Handling the empty-tensor edge case is essential to composability: without precise control for empty tensors, reductions could not compose with downstream operations. Therefore, to use embedded control metadata for synchronization, Revet must precisely track empty lists. For example, in our abstraction, the three 2-D tensors [[]] and [[],[]] and [] have unique representations (Ω_1, Ω_2 vs. $\Omega_1, \Omega_1, \Omega_2$ vs. Ω_2). Although all three of these tensors contain no actual data, they represent different control-flow structures: an outer loop running one iteration with a zero-length inner loop, an outer loop running twice with zero-length inner loops, or an outer loop that does not run. Therefore, when passed to an additive reduction, they must yield distinct results: [0], [0,0], and [].

B. Streaming Tensor Primitives

Given a format for on-chip links that can embed and propagate control-flow decisions, we now describe the streaming primitives that implement local control decisions like parallelism and branching. These primitives are used for individual basic-block edges, and they respect our structured-link tensor format (Section III-A), so they can be composed arbitrarily. Together, they provide the sequencing, iteration, and selection needed for arbitrary algorithms. Revet's machine model requires that primitives respect the SLTF for composability:

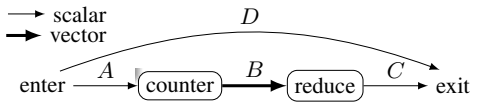
1) Every barrier that enters a primitive exits that primitive exactly once, in order, and
2) Thread data (stored in the SLTF) is not reordered with respect to barriers. Data can be reordered in between barriers.

By obeying these conditions, Revet's primitives can rely on the behavior of nested primitives to guarantee correctness. For example, a **`while`** loop containing an **`if`** statement can rely on the **`if`** statement not modifying barriers or reordering threads across barriers. Similarly, an **`if`** statement can contain a parallel-patterns **`foreach`** loop on one of its branches—this is useful for cases like periodically loading a vector of data from DRAM to SRAM.

a) Element-Wise Operations: Element-wise operations process one or more tensors: for example, two tensors may be added to yield a third tensor. Memory operations are also element-wise operations: an allocation transforms a void value into a pointer, a read transforms an address into a result, and a write transforms an address and data into a void value. Element-wise operations do not change the ordering, hierarchy, or number of dataflow threads. Therefore, in this section's examples (Figures 2 to 4), these operations can take place along any graph edge.

Revet provides memory ordering guarantees within a *thread.* Therefore, the machine model must guarantee ordering for memory operations' side effects within a basic block. To do so, it relies on data-free *void* tokens like SARA's CMMC [45]: these are generated by memory operations as results and are inserted as operands. Finally, these void tokens are carried through basic block transitions, like merges, to guarantee that basic blocks execute in order.

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.



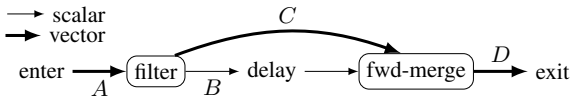
Tensor Abstraction:

$A, C, D:$	$[t_1, t_2]$
$B:$	$[[t_{1.1}, t_{1.2}, t_{1.3}],$ $[t_{2.1}, t_{2.2}, t_{2.3}, t_{2.4}]]$

SLTF:

$A, C, D:$	t_1, t_2, Ω_n
$B:$	$t_{1.1}, t_{1.2}, t_{1.3}, \Omega_1$ $t_{2.1}, t_{2.2}, t_{2.3}, t_{2.4}, \Omega_{n+1}$

Fig. 2. A **foreach** loop: a 1-D tensor of threads is expanded into two dimensions and then contracted. Hierarchical-tensor and streaming-barrier (SLTF) views of data are shown. For simplicity, element-wise operations are elided. They could be added along any dataflow edge between complex primitives.



Tensor Abstraction:

$A:$	$[t_1, t_2, t_3, t_4, t_5]$
$B:$	$[t_3]$
$C:$	$[t_1, t_2, t_4, t_5]$
$D:$	$[t_1, t_2, t_4, t_5, t_3]$

SLTF:

$A:$	$t_1, t_2, t_3, t_4, t_5, \Omega_n$
$B:$	t_3, Ω_n
$C:$	$t_1, t_2, t_4, t_5, \Omega_n$
$D:$	$t_1, t_2, t_4, t_5, t_3, \Omega_n$

Fig. 3. In a filter-merge operation (**if** statement), a vector of threads is partitioned into two vectors, one for each branch. Here, link B is mapped as scalar to avoid overprovisioning network resources for a rare execution case. If links B and C were equally common, both could be mapped to vector dataflow resources at the cost of additional network congestion.

b) Expansion, Reduction, & Flattening: Expansion primitives (Figure 2) enlarge tensors to express map operations. The simplest expansion primitive is broadcasting, which takes a k- and a $(k+1)$-dimensional tensor and repeats every element in the first tensor along the last dimension of the second one. Counters can also expand tensors: a counter takes three k-D tensors (min, max, and step) and transforms them into a $(k+1)$-D tensor. Reduction (Figure 2) uses an associative operation to coalesce the last tensor dimension into one element, lowering each barrier by one level. Flattening also removes a level of hierarchy from barriers but leaves elements untouched. Taken individually, these operations do not obey the SLTF constraints mentioned previously because they modify barrier levels. However, an expansion/reduction pair can be used to wrap arbitrary code to implement a **foreach** block. Similarly, an expansion/flattening pair can be used to duplicate threads without adding a level of hierarchy, thus implementing a **fork** statement. Both **foreach** and **fork** obey the SLTF constraints.

c) Acyclic Subgraphs: Filtering & Forward Merging: Tensor filtering (Figure 3) takes an element tensor and a predicate tensor and returns only the elements for which the predicate evaluates to true. For example, an **if** statement would use a filter operation to mask off elements so that each element goes to either the **if** block or the **else** block. Barriers are passed through unmodified, creating two tensors from one moving forward through the pipeline.

Forward merging (Figure 3) is used at the beginning of a basic block that has two *forward* branches into it. Merging interleaves elements from the lowest tensor dimension eagerly: whenever either input is ready to send, the merge can pass it through. In Figure 3, this is evident when t_3, which branches onto a slow path, exits the merge last. To preserve thread state, the merge can take multiple tensors (corresponding to all the live variables in a thread) and ensure that they are merged atomically. Because the merge keeps per-thread data together, and threads within a hierarchy level are unordered, it preserves programming model correctness.

When the merge unit reaches a barrier in the streaming on-chip input, it stalls that link until it reaches an equal barrier in the opposite link. This limits reordering (e.g., between two branches of the same **if** statement) to one level of the tensor hierarchy, so threads in a parallel region do not cross barriers and remain synchronized to their parent thread. By waiting for barriers to arrive, the **if** statement is tolerant of network effects including delays and bandwidth limits.

d) Cyclic Subgraphs: Forward-Backward Merging: Like forward merging, forward-backward merging (Figure 4) interleaves incoming threads. However, unlike forward merging, forward-backward merging combines tensors resulting from backward branches (e.g., at the head of a **while** loop). Forward merging would not work for this case: the backward branch can only send a barrier *after* the merge sends a barrier, but the merge can only send a final barrier once it receives one from the backward branch. Therefore, forward-backward merge uses different logic to break this would-be cyclic dependency. Intuitively, the forward-backward merge at the loop header takes a 1-D tensor of input elements at a time and iterates it to form a 2-D tensor of executed loop bodies.

A natural loop will have one header block, which is the meeting point of all forward edges into the loop and all backward edges, and the forward-backward merge is located at this loop header. The forward-backward merge primitive starts by outputting values from the forward branch into the loop body until it receives a done-token. The done-token causes the merge to stall inputs on the forward branch, and the loop header will use barrier semantics to ensure the loop body is empty before allowing more threads to enter. Because the loop header is the sole entry point to a natural loop, it can *reassign* barrier levels inside its loop as long as barriers exiting the loop are correct. Specifically, the loop header *adds* a level to incoming barriers, so it can reserve the lowest barrier Ω_1 for checking whether the loop body is empty. The merge will send a Ω_1 token to terminate its sent data; it will continue to send this token every time it appears at the backward-branch input.

When the loop body is empty (all the threads executing the while loop have terminated), the backward branch will receive two Ω_1 tokens in a row, which will cause the forward-backward merge to send a done token at one level higher than that originally received on the forward-branch link. Edges leaving the body then lower all barriers by one level, eliminating the added Ω_1 barriers and restoring input barriers to their correct levels. This ensures that cyclic regions respect the same barrier constraints as acyclic ones, making them composable. Unlike

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

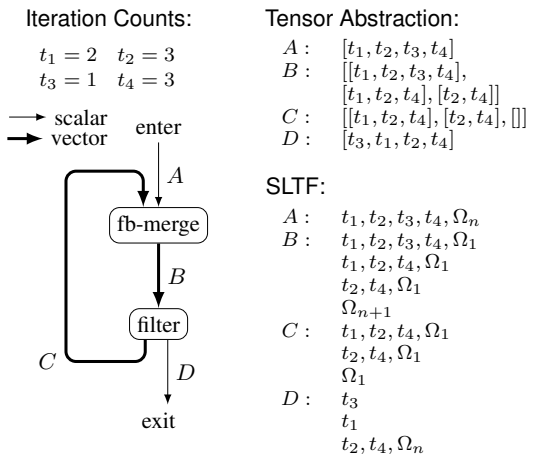


Fig. 4. The operation of a forward-backward merge unit (**while** loop) showing how threads iterate repeatedly. This figure shows a scalar entry, under the assumption that each dataflow thread entering on link A will traverse links B and C multiple times.

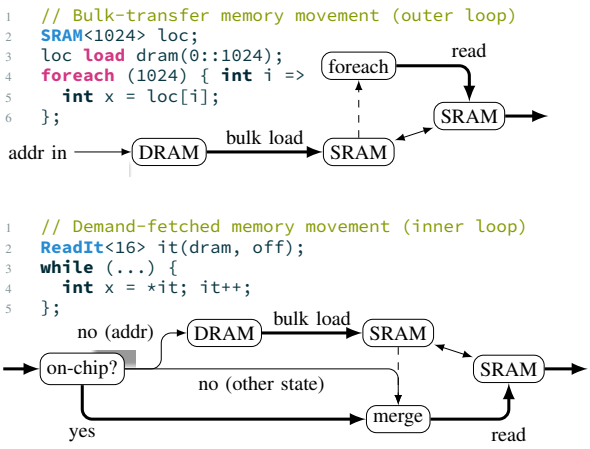


Fig. 5. Above, Spatial [20] requires that memory is explicitly transferred before the start of a parallel section. Below, Revet uses control flow to coordinate transfers without stalls.

Aurochs, the lack of timeouts for detecting loop completion makes this abstraction usable for loops with arbitrarily long loop bodies (like nested while loops).

C. Mapping to Virtual Hardware

Finally, these primitives must be mapped to a hardware model. We assume an overall compute-unit structure based on Aurochs and Plasticine [35], [41]. In our model, mergers, counters, and broadcasts are at the beginning of the pipeline. The pipeline-head logic will stall inputs as needed to meet merging constraints, wait for all inputs to be available for element-wise operations, and send the correct barriers through the pipeline. For example, the pipeline-head logic for a forward merge will take elements from each input and concatenate them until a barrier appears on one input. Once the barrier arrives at an input, no further elements will be taken from it until an equivalent barrier appears on the other input; at that point, both barriers will be dequeued and sent as a single barrier. Broadcasts are handled by repeating the element at the head of an input buffer across all valid lanes, and popping from the input once the corresponding barrier (Ω_1 for a one-level broadcast, Ω_2 for two-level, etc.) arrives. Element-wise operations happen inside the pipeline, with the barriers inserted by the pipeline-head logic propagated unmodified. Reductions, filters, flattening, and vector-to-scalar conversion happen at the end of the pipeline, along with any reduction in barrier level.

As shown in Figure 4, on-chip SLTF links can be allocated to scalar or vector resources. Because they contain their own metadata, SLTF links are agnostic to the bandwidth of network resources—a scalar link can send up to one data element and one barrier per cycle, while a vector link can send up to 16 data elements and one barrier per cycle. For example, a vector with two elements and one barrier (t_1, t_2, Ω_1) can be sent on a vector link in one cycle, but would require two cycles on a scalar link: t_1, then t_2, Ω_1. If two barriers are sent (Ω_1, Ω_2), two cycles would be required on both vector and scalar links. Because Revet primitives do not consider timing information, sending a vector on a scalar link over multiple cycles is semantically equivalent to sending it all at once on a vector link.

SLTF links are allocated to network resources based on expected bandwidth. For example, a **while** loop with a high average trip count would use a scalar entry and vector backedge link to save resources at the loop header. Conversely, a **while** loop written to scan an open-addressed hash table may be expected to rarely loop, and may be provisioned with a vector entry and scalar backedge.

IV. The Revet Language

In the previous section, we formalized the dataflow threads machine model. Revet compiles a structured and imperative programming language, which is familiar to many programmers, to this abstract machine.

A. Key Revet Language Features

The language has two new features compared to languages like C. First, Revet requires user-annotated parallelism in the form of **foreach**, **replicate**, and **fork** statements. Inside parallel regions, Revet supports the fine-grained control flow expected in an imperative language. Second, Revet uses iterators to efficiently orchestrate DRAM to SRAM transfers for data-dependent access patterns inside these sequential sections.

a) Flexible Threaded Parallelism: By default, Revet's code execution is sequential, with mutable variables. To enable parallelism, Revet has explicitly parallel **foreach** loops to eliminate any potential barriers to parallelism (e.g., aliasing); these *child* loop bodies correspond to *threads*. Each thread's program statements run sequentially, but the execution order across threads is unsequenced. Threads inside a **foreach** have a read-only view of their parent's variables, but they can dereference pointers allocated by the parent to perform

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

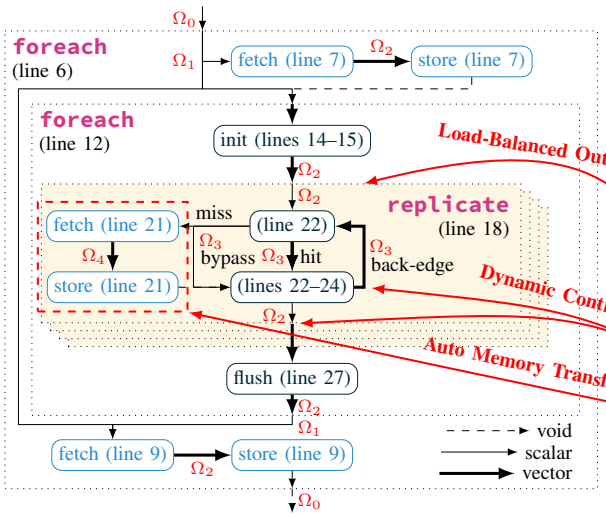


Fig. 6. Basic dataflow for Figure 7 showing hierarchical parallel sections and access-pattern-optimized memories, as specified by the input program before optimization. In Revet code, control-flow *becomes* dataflow, so dataflow arrows represent control operations.

```
1  DRAM<char> input;  DRAM<int> offsets;  DRAM<int> lengths;
2
3  void main(int count) {
4    foreach (count by 1024) { int outer =>
5      ReadView<1024> in_view(offsets, outer);
6      // in_view is loaded here
7      WriteView<1024> out_view(lengths, outer);
8      // This foreach will be rewritten
9      // as a hierarchy-less fork statement.
10     foreach (1024) { int idx =>
11       pragma(eliminate_hierarchy);
12       int len = 0;
13       int off = in_view[idx];
14       // Transition from vector dataflow to scalar
15       // dataflow that is internally vectorized.
16       replicate (4) {
17         // Stateful control within parallel regions,
18         // including stateful updates to the iterator.
19         ReadIt<64> it(input, off);
20         while (*it) {
21           len++;
22           it++; // Iterator is dynamically reloaded.
23         };
24       };
25       out_view[idx] = len;
26     };
27     // out_view is flushed here.
28   };
29 }
```

Fig. 7. Revet code for `strlen()`. Code in the highlighted box could not be expressed in Spatial [20]. The simplicity of Revet's programming model is highlighted by red arrows, which show how standard imperative language features map to new dataflow features.

memory writes. Finally, a **foreach** thread may return a value, which is associatively reduced and returned to the parent.

Revet, unlike prior work (such as Spatial [20], Plasticine [35], or GPUs), supports flexible nested parallelism, where lanes of a vector-parallel program can have further-nested vector loops. Nested parallelism enables scalar to vector *broadcasting*, which uses fewer on-chip resources, and flexible parallelism allows vector regions to be nested inside other vector regions. This is important for emulating caches: all threads start in a vector outer region and either traverse a vector cache hit path or a scalar miss path with a further-nested vector DRAM load. Without this flexibility, the vector hit path would preclude vectorization of DRAM loads.

Revet uses a two-step approach to extract parallelism beyond a single vectorized datapath. First, a **foreach** loop transitions code execution from scalar to vector. Next, a **replicate** statement transitions from a *single vector* dataflow to *multiple scalar* dataflow. Finally, a **foreach** or **while** loop can be used inside the **replicate** to finally create multiple vector dataflow pipelines. To build **replicate**, Revet uses the filter operator to distribute threads across the parallel inner regions and the forward-merge operator to combine them at the end.

Finally, Revet supports dynamic thread spawning and termination using the **fork** construct, unlike GPUs that spawn threads in rigid blocks at kernel launch time. While **foreach** creates threads beneath a parent, **fork** creates new threads at the same hierarchy level. This is similar to a POSIX fork [38], except Revet can fork an arbitrary number of threads. We also provide an **exit** operation that terminates thread execution without returning a value to the reduction.

fork is generally more expensive than **foreach**: the latter can use scalar-to-vector broadcasting to reduce on-chip dataflow requirements, while the former must duplicate all live variables because no hierarchy is added. However, **foreach** loops have static restrictions on branching—a child thread cannot branch outside the **foreach**. Because **fork** does not add hierarchy, there are no such restrictions.

b) Access-Pattern Optimized Memories: Memory access patterns that are easy for programmers are hard for hardware and vice versa, and this dichotomy is most evident when programmers are writing sequential code. Programmers would rather access memory one word at time, relying on hardware like caches to keep access times low. Conversely, hardware prefers loading an entire vector from DRAM into an SRAM scratchpad and then accessing the scratchpad explicitly. We balance the burden of achieving high performance in a scratchpad-based design between the programmer and the compiler using the semantics shown in Table I.

The programmer starts by choosing an appropriate access mode. For example, affine accesses with known dimensions should use views, which coordinate large-tile transfers and can be accessed within **foreach** loops. Conversely, data-dependent sequential accesses should use iterators, which our compiler maps to optimized hardware that coordinates small-block transfers dynamically for each thread based on local control-flow decisions, as shown in Figure 5.

To support these primitives, and maintain Spatial's [20] support for multi-buffered on-chip SRAM, Revet supports dynamic allocation of on-chip memories. Like Spatial, Revet requires that on-chip SRAMs have a compile-time fixed size. However, unlike Spatial and SARA [45], Revet supports out-of-order allocation and deallocation to enable reordered thread execution.

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

TABLE I
ON-CHIP MEMORY ADAPTERS. ONLY ARRAY-DECAY MEMORIES CAN BE ALLOCATED OUTSIDE A `foreach` LOOP AND ACCESSED INSIDE.

Access Pattern & Name	Read	Write	Array-Decay
`SRAM<type,size>()`	Yes	Yes	Yes
Small auto-fetched and -stored tiles:			
`ReadView<size>(dram, base)`	Yes		Yes
`WriteView<size>(dram, base)`		Yes	Yes
`ModifyView<size>(dram, base)`	Yes	Yes	Yes
Linear read, optionally with peek `tile` elements ahead:			
`ReadIt<tile>(dram, seek)`	Yes		
`PeekReadIt<tile>(dram, seek)`	Yes		
Linear write (output iter):			
`WriteIt<tile>(dram, seek)`		Yes	
Linear write with manual flush. May overwrite a word where a sub-word is touched:			
`ManualWriteIt<tile>(dram, seek)`		Yes	

B. Case Study: `strlen`

Figure 6 shows how Revet's parallel constructs work together to map the `strlen` computation in Figure 7 across spatially distributed, parallel pipelines. Specifically, the body of the **`while`** loop on line 20 and the implicit fill path for the **`ReadIt`** are critical code sections, so Revet uses explicit parallelism to run them on multiple vector pipeline. The outer **`foreach`** (line 10) will first transform a scalar value into a vector of threads. The **`replicate`** (line 16) will use outer-loop (non-vector) parallelism to distribute those threads across the chip as multiple scalar pipelines (instead of one vector pipeline). Next, the **`while`** statement automatically adds vector parallelism again within each inner pipeline, because threads are executing independently on their own lanes. Finally, the **`ReadIt`** transfer path is scalar (refills are infrequent), so the implicit **`foreach`** inside it can be vectorized again. Revet's flexible programming model lets the compiled dataflow code transition between vector and scalar execution without explicit programmer intervention.

V. IMPLEMENTATION

In this section, we describe the practical details behind our prototype implementation, which follows the stages shown in Figure 8. Revet's compiler starts by parsing the language and eliminating several constructs implemented to improve programmer productivity, including views and iterators. Then, the compiler performs optimizations to improve the efficiency of the generated dataflow. At this point, the IR still contains constructs like **`while`** loops and **`if`** statements, but it has been rewritten to be physically realizable (e.g., bulk memory accesses have been converted to **`foreach`** loops) and to generate a more optimal dataflow output (e.g., small **`if`** statements have been converted to predication). Then, the compiler lowers the CFG representation to a dataflow format—effectively, this consists of rewriting basic blocks as infinitely large virtual CUs and replacing structured control flow constructs with the corresponding Revet primitives discussed in Section III-B. Finally, the arbitrary-size virtual CUs are split so that they meet the physical constraints (operation count, input/output count) of our vRDA backend.

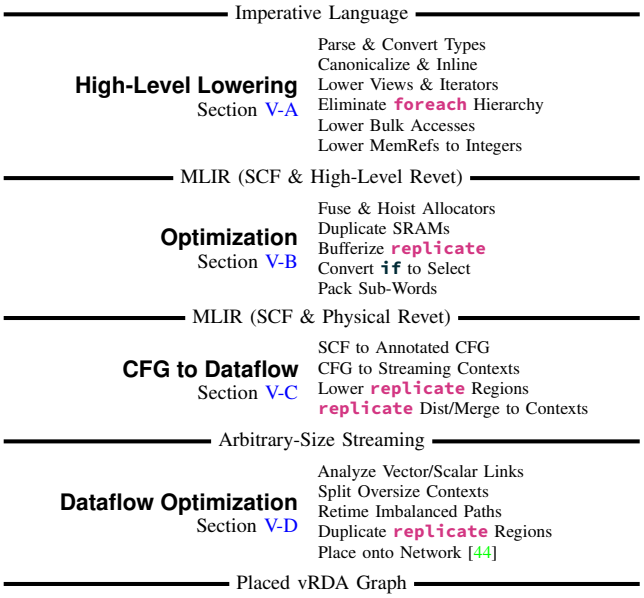


Fig. 8. Revet's compiler passes and IRs.

A. Front-End Lowering

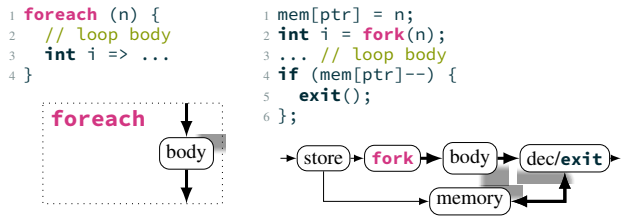
Revet uses a front-end [32] that emits code into an MLIR [23] representation as a mixture of the structured control flow dialect (SCF) and a custom Revet dialect that captures our custom front-end features (Section IV). Our front-end also inserts type conversion operations as needed. Then, we progressively lower the high-level Revet memory operations (e.g., iterators) until every memory is expressed as an SRAM with scalar accesses and perform hierarchy elimination if requested.

a) View & Iterator Lowering: We rewrite Revet views and iterators (Table I) into MemRefs (MLIR's annotated memory type) and integers. Views are simple: allocations are replaced with a MemRef allocation and a bulk load (if needed) and deallocations are replaced with a MemRef deallocation and a bulk store (if needed). These primitives are more efficient for sub-word types because a backend-inserted bulk store can process 32 bits per cycle.

Iterators are slightly more complicated: the basic **`ReadIt`**, for example, has a MemRef buffer, a local pointer (8 bits to reduce dataflow overhead), and a global pointer (in SRAM). The global pointer is fetched and incremented only when the local pointer wraps around. Because dereference is less common, we fill read iterators' buffers only at dereference to decrease the amount of hardware mapped. **`WriteIt`**s can be flushed at increments or deallocation, which would naïvely require two store paths. To avoid this, the **`ManualWriteIt`** takes an input at increment indicating the last loop iteration to elide the deallocation flush.

b) Foreach Hierarchy Elimination: **`foreach`** regions are eventually lowered to streaming tensor operations (Sec-

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.



(a) Hierarchy (b) Flattened

Fig. 9. Hierarchy elimination (**foreach** to **fork**).

tion III-B), which use barriers to sequence threads. However, barriers can limit parallelism inside **while** loops, where they force a total flush of the loop body before new threads can enter the loop. Because barriers change side-effect ordering, they cannot be automatically eliminated, so we only rewrite **pragma**-annotated **foreach** statements.

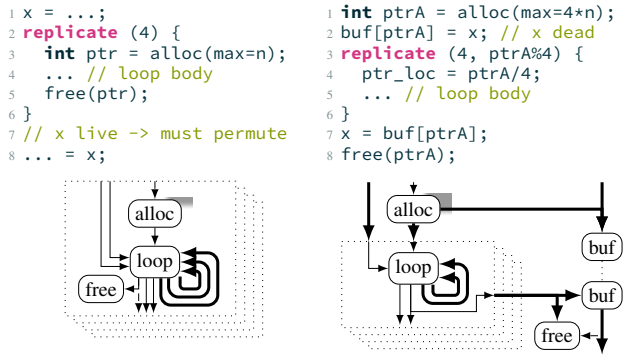
When rewriting these statements, we initialize a memory location with the number of elements expected and execute a **fork** to create hierarchy-less threads, as shown in Figure 9 . Instead of reduction, threads atomically fetch and decrement the shared memory location. If zero elements remain, the thread is the last one and iteration is complete; otherwise, the thread exits. This removes the strict synchronization between **foreach** loops that would otherwise be imposed by SLTF bariers: when using **fork** statements, the straggling children of one parent can be interleaved with those of the next parent.

B. Optimization

At this point, our IR is in a mix of SCF, standard arithmetic operations, and physical memory operations. We run several rewrite passes before lowering the code to dataflow, in addition to existing MLIR passes. These passes rewrite the high-level IR to increase dataflow performance.

a) SRAM Allocator Optimizations: To avoid fragmentation, Revet's on-chip allocation relies on compile-time-determined fixed-size buffers at each memory. For example, if an SRAM buffer is specified as $64\,\mathrm{B}$ (matching the vector width), we rewrite every memory access as: $\mathrm{ptr} \times 64 + \mathrm{off}$. This transformation means that every integer within a range $[0, \max)$ is a valid pointer, and one pointer can be used at multiple memories as long as it is in range. By default, the maximum pointer is the size of a single MU divided by the thread-local buffer size, but users can increase the thread count using a **pragma**, which will cause multiple MUs to be inserted to increase storage.

Allocation fusion lowers the number of pointers that must be tracked in dataflow. We fuse all allocations in a basic block, taking the intersection of valid pointers for the memories to be fused. Because each allocator is sampling from a range defined by a single maximum value, the fused range is defined by the smallest maximum pointer across all memories in the basic block. Finally, Revet loads these pointers into a queue stored in a memory unit, so allocation pops a pointer from this queue and deallocation pushes it back.



(a) Unoptimized (b) Optimized

Fig. 10. Allocator hoisting outside and buffering of a live value around a **replicate**. The low bits of the hoisted pointer steer threads to a **replicate** region, and the high bits are used within it.

b) Allocator Hoisting & Bufferization: If a **replicate** region contains one allocation after fusion, we can increase its range, using the low bits to point to a specific region and the high bits to address an SRAM buffer within that region. This has two benefits. First, it lowers resource requirements by vectorizing allocation (Figure 10), with one allocator globally instead of one per region. Second, it provides native round-robin load balancing: regions only receive new allocations after they complete existing ones.

replicate regions take advantage of Revet's unordered-threads abstraction, and do not maintain ordering. Therefore, when a thread enters a **replicate**, all of its live values would have to be sent into the **replicate**, even if they are not used inside it, creating excessive network congestion. Instead, we *reuse* the single live pointer into a **replicate** (if one has been hoisted) to bufferize live values around it, inserting an SRAM to store the value (Figure 10). Then, we replace all uses after the replicate with a load from this SRAM, so the value is not live through the replicate.

c) If-to-Select Conversion: Naïve dataflow would assign a compute unit to each branch of an **if** statement, but **if** statements without inner loops would just leave empty lanes. Therefore, we inline all **if** statements that lack inner loops, replacing them with conditional moves and predicating memory operations. This is more powerful than MLIR's default of only rewriting empty **if**s.

d) Sub-Word Packing: Every variable that is live into a merge operation consumes a significant number of network resources and input buffers, leading to congestion. Therefore, we identify sub-word values (**int8** and **int16**) that are live into or out of **while** loops. In a naïve dataflow lowering, each of these would have to be promoted to an **int32** because threads execute on 32-bit lanes. However, this would waste buffers and on-chip links, which are critical resources when mapping. Therefore, we pack these values into a single **int32**, making sure to minimize permutation for nested loops. We also optimize AND, shift, and OR operations that can be expressed as sub-word reads and writes on fixed boundaries.

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

C. CFG to Dataflow Lowering

In this section, we describe a prototype system that operates on MLIR's SCF dialect using specially-annotated CFGs as an intermediate format to ensure our hierarchical CFG's constraints are respected.

a) Graph Annotation: Between SCF and dataflow, we use an annotated CFG that indicates which edges should be forward merges and which edges should be forward-backward merges using specialized block terminators. We also flatten `foreach` regions into the main CFG and replace their input and output edges with special counter and reduce terminators (like Tapir [37]). We ensure that every block has no more than two predecessors to meet our hardware's merge constraint.

b) Basic-Block Inputs: When mapping a block, we start by identifying all live-in variables and determining whether they should be broadcast using a mapping table of blocks to nesting depths. For example, a block following an `if` statement will have two sets of live-ins (one from each branch, followed by a merge operation) in addition to broadcasts.

In case there are no live-in values, a data-less void value is inserted between blocks to guarantee a correct number of live-outs. This void value is chained through pipelined memory operations to guarantee a data dependence to enforce ordering after contexts have been split. After cloning inputs, we clone the basic block as element-wise operations.

c) Basic-Block Outputs: Basic-block live-out values are mapped based on the terminator operation. Unconditional branches map to unconditional outputs: each lane in the pipeline is sent as an output. Conditional branches map to filter outputs, taking a condition and a value per lane; only lanes matching the condition are sent. `foreach` block terminators map to counters, which are later moved to the head of their destination context, and reductions map to a reduction operation at the end of the context pipeline. Finally, outputs exiting a while-loop region strip hierarchy without reduction.

d) Replicate: The logic to connect `replicate` regions to the enclosing CFG is implemented using the filter and merge primitives shown in Section III: each thread is broadcast to a filter before every contained CFG and only threads with matching keys are forwarded. Finally, return values from the contained CFG are merged using a tree of forward merge units.

To speed compilation, we use late unrolling for `replicate` regions: we create one node with code and multiple nodes for references, which are duplicated immediately before placement. We then insert work-distribution and output-merging logic using the filter and forward-merge primitives from Section III. To avoid a single slow `replicate` region stalling a hoisted allocator and starving faster regions, we insert link-retiming buffers in the work-distribution logic.

D. Dataflow Optimization

In the previous subsection, we described lowering to a *virtual* streaming IR, which is a one-to-one mapping of control-flow constructs to dataflow. Here, we describe passes that transform the virtual streaming IR into a *physical* streaming IR. These include passes like splitting that ensure our IR can map to on-chip units and optimization passes like retiming.

TABLE II
RDA PARAMETERS USED IN OUR EVALUATION.

Compute units (200)	16 lanes, 6 stages, 6 vec/scal regs/lane/stage
Memory units (200)	16 banks, 256 KiB total
Buffers (per unit)	4×256 word vec., 4×64 word scal.
Outputs (per unit)	4 vector, 4 scalar
Network	3× vector, 6× scalar, dynamic
DRAM	HBM2, ∼900 GB/s, 32B burst

a) Vector/Scalar Link Analysis: Because some buffers can only store scalars, accurately mapping virtual links to either vector or scalar physical links is important—especially for merges. Only two vector-vector merges fit in a context (a total of four vector buffers), but four scalar-vector merges fit, halving the number of resources required. However, if a high-throughput link is mapped to a scalar physical link, then its throughput will be only $^1/_{16}$ of peak. Therefore, we treat links as vector by default, except blocks following `while` loops and the entrances and exits of `replicate` regions and the main program graph. However, a `pragma` can override this.

b) Splitting, Retiming, & Placement: Initially, compute operations are mixed with memory operations in contexts, and a single context may have memory operations at two or more memories and an impossible number of inputs, outputs, or operations. We first place every memory operation into its own context, and then split over-size contexts. We next insert buffers to avoid deadlock and mitigate path-delay imbalances [45]. Finally, we place the partitioned graph using previously proposed tools [44], prioritizing deeply nested nodes.

VI. Evaluation

After discussing our methodology, we discuss Revet's performance and how optimizations improve generated code and out-perform industrial baselines on a variety of applications.

A. Methodology

We evaluate Revet using a cycle-accurate vRDA simulation including a model of HBM2 memory [14], [19], [44] against real-world baseline designs on a variety of kernels.

a) Hardware Model: To evaluate the dataflow threads programming model and backend abstraction in a physically-constrained environment, we use an abstract machine model based on Plasticine [35]. Our abstract vRDA comprises 200 compute units (CUs), 200 memory units (MUs), and 80 DRAM address generators (AGs) connected by a flexible on-chip network [44]; the parameters we use are shown in Table II. Our vRDA backend uses a non-timesliced design for all CUs and the network. We estimate area as that of Capstan [36] with the logic from Aurochs [42] added in, giving a total area of approximately $189\,\mathrm{mm}^2$ in a $15\,\mathrm{nm}$ educational process with a clock frequency of $1.6\,\mathrm{GHz}$. Because the $15\,\mathrm{nm}$ library lacks a memory compiler, prior work used SRAMs scaled from a $28\,\mathrm{nm}$ industrial library. This is 4.3× smaller than Nvidia's V100 GPU, our primary baseline [28].

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

TABLE III
APPLICATIONS AND DATA DISTRIBUTIONS USED TO TEST REVET. KEY FEATURES ARE SHOWN; APPLICATIONS USE ADDITIONAL FEATURES (E.G., `replicate` AND `WriteView` FOR DATA STORAGE). †THIS APPLICATION RAN SLIGHTLY *slower* FOR LARGER DATASETS (1.4×).

	Lines	Description	Per-Thread Dataset	Key Features	Scale (MiB) Revet	GPU	CPU
isipv4	34	DFA regex	90% valid addresses, 10% 'INVALID'	`replicate` (×2)	38	1527	15275
ip2int	41	Parsing	Random IPv4 addresses	`replicate` (×2)	52	2070	20700
murmur3	62	Data hashing	64 B blobs	`ReadIt`	136	1360	13600
hash-table	56	Hash-table lookup	int32 keys/values, 10^8 slots, 25% load	`ReadIt`	16	2400	24000
search	54	Exact-match search	Find 'Moby Dick', 256 B chunks of 'Moby Dick'	`PeekReadIt`, `while` (×2)	260	41†	26000
huff-dec	40	Decompression	64 codes, 16-bit max length	`ReadIt`	171	1714	17140
huff-enc	58	Compression	64 codes, 16-bit max length	`ManualWriteIt`	171	1714	17140
kD-tree	74	Count points in rect.	10^8-point grid, random searches yield 16 points	`fork`	40	800	8000

TABLE IV
RESOURCES USED BY REVET APPLICATIONS.

	Parallelization		Inner			Outer			Replicate		Retime/Buffer (MU)			Total			HBM2 (%)		
	Outer	Lanes	CU	MU	AG	CU	MU	AG	CU	MU	Deadlock	Buffer	Retime	CU	MU	AG	Read	Write	Total
isipv4	3×9	432	81	54	27	23	10	6	43	10	0	12	73	147	159	33	83.0	0.5	83.5
ip2int	3×10	480	90	30	30	23	10	6	46	10	0	12	79	159	141	36	68.5	13.1	81.6
murmur3	14	224	112	28	14	11	5	3	21	7	42	5	20	144	107	17	73.9	4.1	78.0
hash-table	16	256	112	32	16	13	4	2	23	6	48	4	22	148	116	18	29.6	2.3	32.0
search	8	128	120	40	8	10	4	2	12	3	32	4	13	142	96	10	66.3	0.8	67.1
huff-dec	9	144	135	45	18	7	3	1	13	4	27	2	41	155	122	19	17.1	31.6	48.7
huff-enc	9	144	126	45	18	10	4	2	13	4	27	4	43	149	127	20	35.0	17.5	52.5
kD-tree	5	80	110	55	65	3	1	0	7	2	10	0	36	120	104	65	57.1	0.2	57.3

To ensure that we accurately model hardware, we split Revet's compiled programs to map to the blocks provided by our vRDA machine model. Our splitting constraints are the number of pipeline stages, registers, inputs, and outputs; we assume that merge units, constant inputs to merges, counters, and void inputs do not consume resources beyond their associated input buffers and registers. Furthermore, to respect MU and AG mapping limits, we only map address generation contexts where all inputs are scalar and output-accumulation contexts where the only operation is a void reduction. We further assume that operations can read and write 8- or 16-bit sub-registers and a small skid-buffer when reshaping vectors to fit on scalar links.

b) Baselines: We evaluate Revet against an Nvidia V100 [15] GPU and an Intel Xeon CPU. All of our applications have independent threads running under a parallel region, so we scale problem sizes across platforms so that each reaches its peak performance and report normalized performance in GB/s. This ensures that baselines have the best performance possible and sets a lower bound on Revet's performance. Application sizes are reported as the sum of input and output data sizes, except for kD-tree, which uses the size of the fetched points that are counted.

GPU tests were performed on an AWS p3.2xlarge instance using CUDA 11.6, RAPIDS 22.04 [29], and cuCollections [27] running Linux 5.13.0 and Nvidia driver 510.47.03. For all benchmarks except kD-tree, we use nvprof to measure only kernel runtime, which excludes device/host transfers, barriers, and CUDA stream synchronization. kD-tree uses host timers because the RAPIDS implementation uses multiple kernels. CPU tests were performed on an AWS m6i.16xlarge using GCC 11.2.0 with -O3 and OpenMP. The CPU is a 3rd generation Xeon Platinum at 3.5 GHz with 64 threads and 205 GB/s of DDR4 bandwidth. For both baselines, benchmarks were run 25 times and the average of all runs was taken.

c) Applications: We use a variety of applications to evaluate Revet, as shown in Table III. These applications are selected to focus on Revet's *new* functionality, so they all represent applications that cannot be compiled to Plasticine or other vRDAs. Because threads provide a superset of MapReduce functionality, any code that could be compiled by Spatial could also be mapped by Revet. They are drawn from a variety of domains including string analytics, data-structure traversal, search, and generic data-processing algorithms like hashing. We focus on discrete kernels to avoid inter-kernel overheads in the baselines; the GPU tree traversal is the only multi-kernel baseline.

When evaluating applications, we assume that runtime is a function of bulk throughput, data size, and initialization time: $\text{runtime} = \text{size}/\text{throughput} + \text{init}$. Furthermore, all of our benchmarks exploit abundant, non-communicating threaded parallelism, so the amount of work can be increased without changing the *nature* of the work. Revet also uses static SRAM allocation instead of caches, which means that threads do not interfere with each other, and adding more threads will not decrease aggregate system throughput.

Therefore, for every platform, we use the largest data sizes feasible to measure throughput, which yields trivially short initialization times. Although the data sizes for Revet are relatively small, the inclusion of initialization time means that throughput would only *increase* with larger datasets.

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

TABLE V
PERFORMANCE EVALUATION, INCLUDING TESTS WITH IDEAL MODELS FOR SRAM (S), NETWORK (N), AND DRAM (D).

	Revet	V100		CPU		Ideal (Speedup ×)		
	GB/s	GB/s	×	GB/s	×	D	SN	SND
isipv4	443	121	3.65	7.3	60.6	1.04	1.07	1.18
ip2int	508	381	1.33	9.1	55.9	1.42	1.03	1.55
murmur3	628	218	2.88	122.2	5.1	1.55	1.07	2.37
hash-table	42	40	1.05	7.4	5.7	2.70	1.00	3.23
search	481	51	9.45	120.6	4.0	1.37	1.18	1.38
huff-dec	380	97	3.91	19.0	20.1	0.98	1.07	1.08
huff-enc	409	172	2.38	35.0	11.7	1.01	1.17	1.18
kD-tree	52	1.5	33.93	3.4	15.3	1.28	0.92	1.65
geomean			3.81		13.9	1.35	1.06	1.59

B. Resource Requirements & Performance

Revet generates resource-efficient vRDA configurations. Furthermore, using Revet, a vRDA out-performs a GPU on a variety of applications and is DRAM-bandwidth-limited for many of them as well.

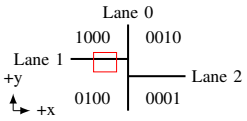
a) Resource Breakdown: Our first evaluation shows the vRDA resources required by Revet-generated code in Table IV. It is challenging for a vRDA application to make 100% use of resources due to on-chip network constraints [44], so we scale outer parallelism to use 70% usage of the critical resource (CU, MU, or AG).

Using outer and vector parallelism, Revet can provide hundreds of SIMD-parallel lanes. Furthermore, some applications (isipv4 and ip2int) are outer-parallelized at two levels: tile loads/stores for thread arguments/results and the inner loops. For these, up to three vectorized streams (48 lanes) process tiles of thread inputs/outputs while thirty vectorized (480 lanes) streams process the inner while loops. Although Revet maps fewer vector lanes than a GPU has CUDA cores, Revet lanes process multiple pipelined instructions per cycle.

Revet has minimal resource overhead: for all of our applications, most mapped CUs are used for inner-loop operations, and only a few CUs and MUs are used for workload distribution and buffering live values around replicates. MUs are also used for retiming, but these can frequently be shared with those inserted for deadlock avoidance.

b) Throughput: Table V shows each design's throughput. On average, Revet is 3.8× faster than the GPU; when estimated die area is taken into account, this gap grows to over 16×. Furthermore, isipv4, ip2int, and murmur3 use over 75% of the peak HBM2 bandwidth; hash-table is limited by DRAM activations. The greater geomean performance improvement from ideal DRAM (+35%, D) than ideal on-chip resources (+6%, SN) also shows that our applications are well-mapped.

The GPU performs best when each thread processes a small amount of data (ip2int and isipv4 read about 13 B per thread). On applications like murmur3 and search, the GPU is slowed down by the longer data involved (64 B and 256 B). Although SIMT supports 32 threads per cycle, the GPU can process fewer independent accesses through its L1 cache: therefore, independent threads cannot run at full throughput unless they access nearby addresses. This is because the GPU expects, and requires, coalescing for cached levels of the memory hierarchy (i.e., everything except explicit SRAM): the L1 cache can only execute a certain number of tag checks per cycle [24]. Revet does not have this problem: because iterators are mapped in SRAM, they execute in parallel without tag checks.



Lane (dim)	0 (x)	1 (y)	2 (y)
Ignore	**0000**	**0011**	**1100**
Less-Than	**1100**	**0100**	~~0001~~
Greater-Than	~~0011~~	**1000**	**0010**
Lane Out	1100	1111	1110

Fig. 11. Three lanes work together to traverse a folded kD-tree. Each lane performs one comparison.

Search performs better on the vRDA because Revet's support for efficient branching enables the asymptotically-efficient Boyer-Moore [4] algorithm. Boyer-Moore is complex because each thread is independently matching backwards along the pattern or computing an offset; Revet uses nested `while` loops to support this behavior. In addition to a poor search algorithm, the GPU's constraints also force a poor algorithm for tree traversal. Because CUDA does not support recursion (like the CPU) or `fork` statements (like Revet), every iteration of its quad-tree traversal must write into a large array. However, because each iteration only selects a few children, little parallelism is extracted to amortize inter-kernel overheads.

c) Aurochs Comparison: Finally, we compare to Aurochs [41], a primitive implementation of dataflow threads. Most Revet applications cannot run on Aurochs because it lacks support for the local allocation needed for intra-thread locality. The tree traversal benchmark is supported. The more efficient on-chip primitives supported by Revet allow the kD-tree implementation to be over 11× faster than the Aurochs tree traversal. First, Aurochs lacked support for thread-local storage, which results in up to 10 live variables traversing its pipeline that have to be duplicated whenever threads are forked; Revet can store these variables in SRAM.

Second, Aurochs does not support fine-grained parallelism via `foreach` loops. Our kD-tree uses a `foreach` loop to vectorize 15 comparisons for every node, which are ANDed together to identify which child nodes should be traversed. Figure 11 shows a simplified version that uses three lanes to traverse two tree levels. Every lane's comparison starts with a mask for regions that are ignored: for instance, Lane 1 ignores the right two regions. Then, the lane compares its partition value against the query's minimum and maximum ranges, which produces a per-lane output. In the example, Lane 2's comparison, which produces a validity mask of 1110, is overridden by Lane 0, which determines that 0010 and 0001 are invalid. Therefore, with only 64 B loaded from DRAM per node, Revet can handle a 16-ary tree.

C. Optimizations

In Section V, we discussed several optimization passes for Revet. In this section, we discuss how these either save resources or improve performance directly.

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

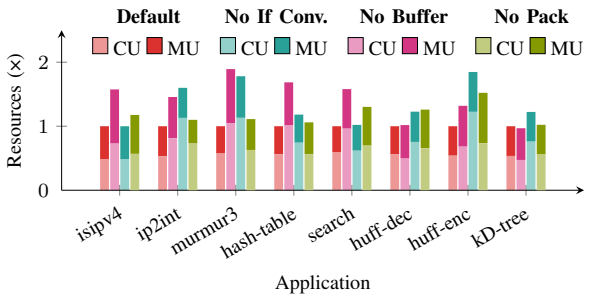


Fig. 12. Resource increase (CUs and MUs) when turning off different Revet optimization passes.

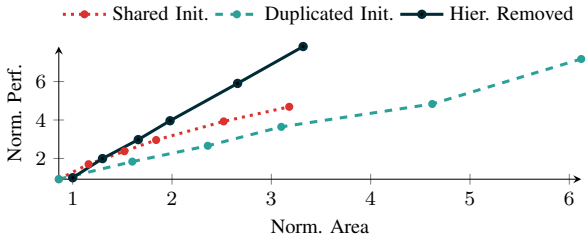


Fig. 13. Performance vs. area with and without hierarchy removal (ideal SRAM, network, and DRAM models). Hierarchy removal moves the scaling curve up and left, with more performance at lower area.

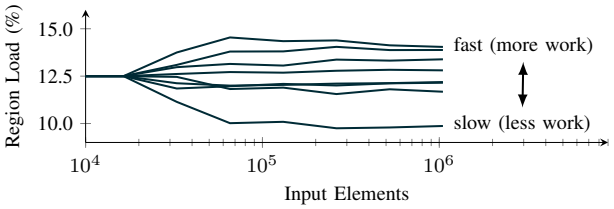


Fig. 14. Per-region load vs. number of inputs for search. `replicate` regions are assigned work based on throughput.

a) Resource-Saving Optimizations: Figure 12 shows the effect of disabling enhanced if-to-select conversion, allocator hoisting and replicate bufferization, and variable packing. Not all optimizations improve all applications: for example, `if` to select conversion has no impact on isipv4, which has no convertible `if` statements. These passes lower resource requirements by either reducing the number of basic blocks (If Conv) or live variables that have to be permuted in the pipeline (Buffer and Pack). However, extracting pointers after allocator hoisting can add resources, as seen in the Buffer column. Without these resource-saving optimizations, only kD-tree would be able to hit the outer-parallelism factors that we target in our evaluation because it is AG-limited.

b) Hierarchy Removal: Figure 13 shows how hierarchy removal improves area-performance scaling, using murmur3 as a case study. Generally, every application loads a tile of thread initialization data from DRAM, computes the thread results, and stores the data back. With hierarchy removal, applications run on small tiles which can coexist in the pipeline. The hierarchical case uses large tiles that *cannot* coexist: one tile must be done inside a while loop before another can start executing. These large tiles can either be loaded outside the `replicate` (shared initialization logic) or inside the `replicate` (duplicated init.).

With tiles loaded outside replicated regions, hierarchy actually slightly reduces area by limiting overhead. However, as outer-parallelism increases, the parallelism allocated to each replicated region decreases, which leads to a widening performance gap. If tiles are instead loaded inside replicated regions, hierarchy can achieve similar performance but with area increases from duplicated tile loads and stores.

c) Load Balancing: Figure 14 shows how allocation hoisting improves search's performance on the hybrid network, where different outer-parallel regions run at different throughputs. This handles a slow `replicate` region that is 30% slower than the fastest one, and is better than Plasticine's [35] fixed work allocation, which would be bottlenecked by the slow region. For small amounts of work (left), the allocator is able to assign buffers to all incoming threads without starving. Therefore, every region gets an equal amount of work (12.5% of the input). As the amount of work increases, the allocator runs out of buffers, so not every thread is able to run in the first wave. Then, faster regions free their allocated buffers first, and regions are assigned new work only after they complete existing work. This creates a feedback loop that leads to slower regions receiving less work (less than 10%) and faster regions more (14%), avoiding a 21% slowdown if all regions ran at the slowest region's speed.

VII. Related Work

In this section, we discuss how Revet differs from prior work, including SIMT, programmable dataflow machines, scratchpad management, and parallel languages.

a) SIMT & Vector-Threads: SIMT models like CUDA [30] and Vector-Threads [22] are the dominant programming model for GPUs and the inspiration for Revet. In CUDA GPUs, 32 threads form a warp which can execute one instruction per cycle, so inactive threads waste an execution slot. Revet can avoid this problem using spatial execution. Furthermore, CUDA's thread blocks prevent efficient dynamic thread spawning, while dataflow threads permits easy thread duplication in spatial pipelines [1].

b) Programmable Dataflow: Plasticine [35], targeted by Spatial [20] and SARA [45], only has one FSM per compute unit. Therefore, one iteration has to complete before the next one can start! HLS [8] suffers from the same global-FSM problem, because it slices C programs into control logic (FSM) and datapath. Aurochs [41] was the first vRDA to support dataflow threading using the relational-algebra operations introduced by Gorgon [42]. Aurochs lacked composable control-flow primitives and as a result did not support high-level compilation. Furthermore, Aurochs did not support per-thread SRAM buffers and could not send on-chip scalar values, both of which are needed for efficient dataflow. Unlike Capstan [36] and SAM [13], which enable direct loops over sparse data

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

structures, Revet optimizes for easier-to-achieve parallelism across threads.

Stream-join [26], [43], as proposed in the SPU [9], is another paradigm for dataflow computing, but also suffers from the single-FSM problem. Instruction-based designs (where a CGRA is integrated with a CPU, like SNAFU [11] and MANIC [12]) suffer from the single-FSM problem as well, because the CPU can only logically execute one thread at a time. Similarly, time-scheduling (e.g., Fifer [25]) virtualizes hardware to provide the abstraction of more resources without changing the underlying compute model. Virtualizing multiple computing contexts can provide the abstraction of multiple FSMs, but the need for reconfiguration means that only one FSM can run at a time.

Other approaches like Fleet [40] and CoRAM [6] help support streaming on FPGAs but require RTL for the streaming algorithms. Others have mapped complicated programs to tagged dataflow [2], including Kahnian networks [18] and the Monsoon processor [31]. Revet is more efficient and powerful because it targets vectorized, pipelined dataflow without the need for tags and does so from an imperative language.

c) Data Orchestration: By targeting dataflow hardware with scratchpads instead of caches, Revet can improve efficiency relative to von Neumann approaches (albeit by eliminating the potential for reuse across threads). Other approaches like Buffets [33] and Stash [21] also automatically orchestrate scratchpad memory hierarchies. However, Revet's approach is unique by natively supporting multithreaded accesses and reusing the logic of a vRDA to do so.

d) Languages & Compilers: Others have proposed streaming-native domain-specific languages (DSLs) to capture dataflow behavior like StreamIt [39] and Spidle [7]. However, the goal of Revet is not to expose streaming to the user, but rather to expose an imperative language and lower it to a dataflow backend. Finally, Cilk [3] and OpenMP [10] provide C extensions for parallelism, like Revet; however, these languages use their extensions to target multicore CPUs and cannot lower to dataflow.

VIII. Conclusion

We introduce Revet, a compiler that takes threaded imperative code and lowers it to run on a vectorized RDA. Revet enables control flow in the presence of abundant unordered parallelism on an architectural paradigm that previously only supported control-flow-free parallel sections. Thus, Revet provides SIMT's *threaded* abstraction—one control flow decision per lane, per cycle—while also demonstrating intelligent scratchpad allocation that eliminates the need for caches for a wide range of threaded applications. On a variety of real-world applications, Revet is 3.8× faster than a GPU on a 4.3× smaller chip and over 13× faster than a CPU.

Acknowledgements

We would like to thank Muhammad Shahbaz, Olivia Hsu, Tian Zhao, and the anonymous reviewers for their feedback on this paper. This work was supported in part by the NSF under grant numbers 1937301, 2028602, CCF-1563078, and 1563113. This research was also supported in part by the Stanford Data Analytics for What's Next (DAWN) Affiliate Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] A. Adinets. CUDA dynamic parallelism API and principles. [Online]. Available: https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/

[2] Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, monitors, and dataflow," *ACM SIGOPS Operating Systems Review*, vol. 11, no. 5, pp. 159–169, 1977.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[4] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 777–786, 2004.

[6] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: an in-fabric memory architecture for FPGA-based computing," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011, pp. 97–106.

[7] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu, "Spidle: A DSL approach to specifying streaming applications," in *International Conference on Generative Programming and Component Engineering*. Springer, 2003, pp. 1–17.

[8] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[9] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 924–939.

[10] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[11] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: an ultra-low-power, energy-minimal CGRA-generation framework and architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1027–1040.

[12] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia, "Manic: A vector-dataflow architecture for ultra-low-power embedded systems," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 670–684.

[13] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 710–726.

[14] JEDEC, "High bandwidth memory (HBM) DRAM," *Jesd235*, vol. 16, 2013.

[15] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[16] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped Google's TPUv4i: industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1–14.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[18] G. Khan, "The semantics of a simple language for parallel programming," *Information Processing*, vol. 74, pp. 471–475, 1974.

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.

[19] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2015.

[20] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis *et al.*, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 296–311.

[21] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 707–719, 2015.

[22] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* IEEE, 2004, pp. 52–63.

[23] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: a compiler infrastructure for the end of Moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[24] T. Lloyd, K. Ali, and J. N. Amaral, "GPUCheck: detecting CUDA thread divergence with static analysis," 2019.

[25] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[26] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 416–429.

[27] Nvidia. cuCollections. [Online]. Available: https://github.com/NVIDIA/cuCollections

[28] ——. Nvidia Tesla V100 GPU architecture. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[29] ——. Rapidsai. [Online]. Available: https://docs.rapids.ai/

[30] ——, "CUDA C programming guide," 2013.

[31] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 82–91, 1990.

[32] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[33] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 137–151.

[34] R. Prabhakar and S. Jairath, "SambaNova SN10 RDU: accelerating software 2.0 with dataflow," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–37.

[35] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[36] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector RDA for sparsity," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1022–1035.

[37] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 249–265.

[38] The Open Group. fork. [Online]. Available: https://pubs.opengroup.org/onlinepubs/009696799/functions/fork.html

[39] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: a language for streaming applications," in *International Conference on Compiler Construction*. Springer, 2002, pp. 179–196.

[40] J. Thomas, P. Hanrahan, and M. Zaharia, "Fleet: A framework for massively parallel streaming on FPGAs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 639–651.

[41] M. Vilim, A. Rucker, and K. Olukotun, "Aurochs: An architecture for dataflow threads," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 402–415.

[42] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, "Gorgon: Accelerating machine learning from relational data," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 309–321.

[43] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 268–281.

[44] Y. Zhang, A. Rucker, M. Vilim, R. Prabhakar, W. Hwang, and K. Olukotun, "Scalable interconnects for reconfigurable spatial architectures," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 615–628.

[45] Y. Zhang, N. Zhang, T. Zhao, M. Vilim, M. Shahbaz, and K. Olukotun, "SARA: scaling a reconfigurable dataflow accelerator," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1041–1054.

Authorized licensed use limited to: Stanford University. Downloaded on July 24,2024 at 05:30:46 UTC from IEEE Xplore. Restrictions apply.