# Pipestitch: An energy-minimal dataflow architecture with lightweight threads

Nathan Serafin Souradip Ghosh Harsh Desai Nathan Beckmann Brandon Lucia
{nserafin,souradig,harshd}@andrew.cmu.edu beckmann@cs.cmu.edu blucia@andrew.cmu.edu
Carnegie Mellon University

## ABSTRACT

Computing at the extreme edge allows systems with high-resolution sensors to be pushed well outside the reach of traditional communication and power delivery, requiring high-performance, high-energy-efficiency architectures to run complex ML, DSP, image processing, etc. Recent work has demonstrated the suitability of CGRAs for energy-minimal computation, but has focused strictly on energy optimization, neglecting performance. Pipestitch is an energy-minimal CGRA architecture that adds *lightweight hardware threads* to ordered dataflow, exploiting abundant, untapped parallelism in the complex workloads needed to meet the demands of emerging sensing applications. Pipestitch introduces a programming model, control-flow operator, and synchronization network to allow lightweight hardware threads to pipeline on the CGRA fabric. Across 5 important sparse workloads, Pipestitch achieves a 3.49× increase in performance over RipTide, the state-of-the-art, at a cost of a 1.10× increase in area and a 1.05× increase in energy.

## CCS CONCEPTS

• **Computer systems organization → Data flow architectures**.

**ACM Reference Format:**

Nathan Serafin, Souradip Ghosh, Harsh Desai, Nathan Beckmann, and Brandon Lucia. 2023. Pipestitch: An energy-minimal dataflow architecture with lightweight threads. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614283

## 1 INTRODUCTION

Advances in sensors, batteries, and energy harvesting have made it possible to deploy sophisticated sensing capability at the "extreme edge," i.e., beyond the reach of traditional communication and power infrastructure [15]. The potential for extreme-edge sensing is enormous, enabling applications like infrastructure monitoring, wearable health sensing, and chip-scale satellites [9, 54].

Today, most sensors offload processing to the cloud. However, communicating data off-device consumes orders-of-magnitude more energy than sensing itself: communication is the limiting factor on lifetime and duty cycle. Prior work has shown that *on-device* processing of sensed data is achievable and provides large benefits [15], e.g., extending battery lifetime from weeks to years. However, extensive on-device compute means *extreme-edge devices are highly sensitive to the efficiency of computation.*

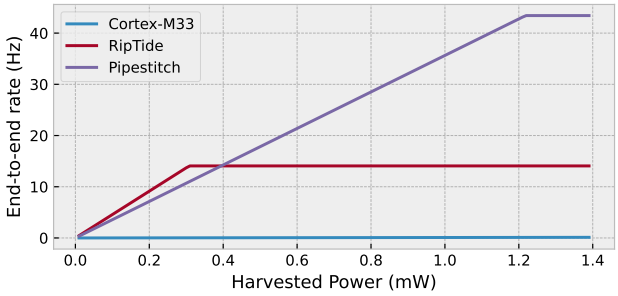



**Figure 1: End-to-end rate (e.g., frames per second) for DNN inference vs. input power in a modeled energy-harvesting system. Inference runs on a Cortex-M33, RipTide, or Pipestitch. Rate depends on both compute time and energy-collection time. Pipestitch provides the best end-to-end rate with larger harvested power, since total time becomes dominated by compute time.**

In particular, *sparse* computations are important at the extreme edge. Prior work has shown that many applications (e.g., ML [22] and DSP [23]) can be made sparse at little loss in accuracy. While sparsity is generally beneficial to reduce unnecessary work [19, 20, 41], it is *absolutely essential* with the tight constraints on memory and energy at the extreme edge [29].

***The problem: Prior energy-minimal architectures run sparse applications too slowly.*** CPUs waste most of their energy on instruction fetch, pipeline control, and register-file access [25]. Recent work has developed *energy-minimal dataflow architectures* [13, 14, 16] that target the ultra-low-power regime. These architectures are coarse-grained reconfigurable arrays (CGRAs): a grid of processing elements (PEs) connected by an on-chip network. A program's instructions are mapped onto PEs, and values are routed directly from producers to consumers, eliminating most of the energy wasted in CPUs.

Energy-minimal architectures have thus far focused exclusively on improving energy efficiency, ignoring performance. Unfortunately, current energy-minimal designs are too slow on sparse computations to put all available energy to good use. Fig. 1 shows an example, modeling end-to-end rate of an energy-harvesting system running a sparse DNN [10]. It compares a Cortex-M33; RipTide, the state-of-the-art energy-minimal design; and Pipestitch, our proposed design. When input power is extremely low (a few µW),


*MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada*

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614283

end-to-end rate is determined only by energy efficiency (pJ/op). As input power increases to a few mW (reasonable for existing energy-harvesting systems [11]), end-to-end rate plateaus as systems reach their peak performance. Fig. 1 shows that RipTide cannot achieve 30 Hz (a standard rate for video processing) on this workload, regardless of input power — it is just too slow. *All of the energy harvested above its peak (a few hundred µW) is wasted.* Furthermore, the Cortex-M33 CPU is so inefficient that it achieves no appreciable performance, even at 10 mW input power.

Hence, the motivating question for this work is: ***How can we improve performance without losing efficiency?*** Performance and efficiency are both key for sparse computations on extremely small power budgets. While blunt hammers like dynamic voltage and frequency scaling (DVFS) can improve performance, they pay a steep energy price. Can we improve performance without sacrificing so much efficiency?

***Why do prior energy-minimal architectures perform poorly on sparse applications?*** Prior energy-minimal architectures are unable to exploit loop-level parallelism in sparse applications. Sparse applications often have long carried-dependence chains that limit parallelism within inner loops, leading to poor utilization. Extracting parallelism within these inner loops requires microarchitectural techniques like reordering and speculation that are too expensive for energy-minimal systems. Moreover, it is difficult for a compiler to recognize potential parallelism *across* inner-loop nests (i.e., between outer-loop iterations) in the presence of potentially-aliasing memory references.

***Opportunity: Parallelism is abundant in sparse applications, but it is not exploited in energy-minimal architectures.*** Parallelism is abundant in most DSP and ML workloads, even those that are sparse. The programmer can easily annotate independent work, allowing parallelism to be exploited to achieve significant performance gains. However, current energy-minimal designs lack architectural support to exploit this coarse-grained parallelism in the face of long carried-dependence chains. Loop unrolling (e.g., executing multiple inner-loop nests in parallel) simply throws area at the problem, increasing device cost and leakage power, without solving the underlying issue. *Energy-minimal designs require new, efficient architectural and microarchitectural mechanisms to exploit parallelism in sparse workloads.*

***Pipestitch: An energy-minimal dataflow architecture with lightweight threads.*** Our solution to the above challenges is *Pipestitch*, a new energy-minimal architecture that improves performance by adding *lightweight dataflow threads* [56] to an energy-minimal CGRA (Fig. 2). Users write Pipestitch programs in C using the well-known `foreach` construct to express unordered parallelism within loop nests [27]. Pipestitch executes threads in parallel by pipelining loop iterations across a single, shared set of instructions mapped to its CGRA fabric. Pipestitch fully pipelines inner loops even in the presence of loop-carried dependences.

Pipestitch exploits thread parallelism while keeping its hardware simple. In particular, Pipestitch maintains *ordered-dataflow execution* [18, 43, 61, 65], where tokens are kept in-order, eliminating tags. Pipestitch maintains an order across running threads; but to achieve high throughput on imbalanced sparse workloads,

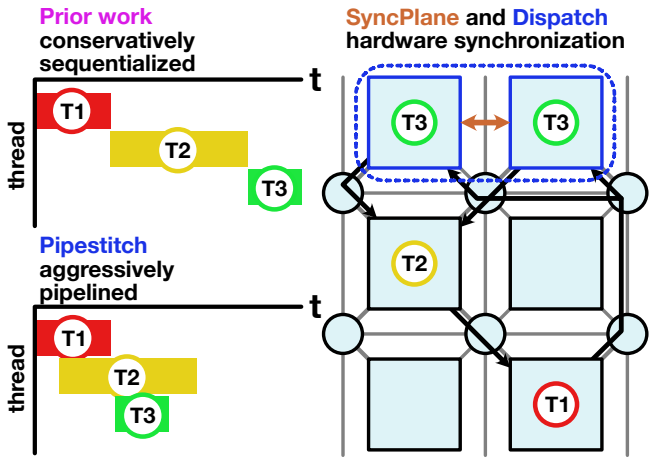



**Figure 2: Pipestitch adds architectural and microarchitectural support for pipelining lightweight threads on energy-minimal CGRAs. Prior state-of-the-art energy-minimal dataflow architectures conservatively sequentialize threads. Pipestitch aggressively pipelines them via a new `dispatch` instruction and SyncPlane control network.**

*Pipestitch allows threads to complete out-of-order* so that stragglers do not harm utilization.

Supporting threads on ordered dataflow requires new architectural and microarchitectural support. Pipestitch introduces the `dispatch` gate to launch and synchronize threads. Threads launch whenever inputs are ready; a new thread can launch every cycle. To maintain ordering, synchronization between `dispatch` gates is required. *Pipestitch introduces the SyncPlane,* a new, lightweight control plane connecting relevant PEs to coordinate control signals.

***Summary of results: Pipestitch significantly improves performance and utilization while adding minimal hardware, energy, and programmer overhead.*** We implement a complete Pipestitch hardware system, including scalar core and main memory, in RTL and Pipestitch's compiler in LLVM. We synthesize Pipestitch in a commercial, sub-28nm process with compiled memories and evaluate it on seven applications representing both dense and sparse computations. Compared to RipTide, the prior state-of-the-art energy-minimal architecture, Pipestitch improves performance by 3.49× on average for threaded apps, by 2.55× on average for all apps, and by up to 3.86× (on sparse matrix slicing), while increasing area by just 1.10× and energy/op by just 1.05× on threaded apps and 1.11× for all apps.

Leveraging these results, Fig. 1 shows that Pipestitch enables extreme-edge systems to increase performance and avoid stranded energy. Pipestitch achieves up to 3× better end-to-end inference rate, fully utilizing energy up to an input power of 2 mW. Moreover, at low input power, Pipestitch nearly matches the state-of-the-art's energy efficiency, showing that its peak performance comes with a small tax on efficiency.

***Contributions.*** This paper...

- ...motivates and quantifies the need for improved performance on sparse workloads at the extreme edge.

- ...identifies the key sources of poor performance in state-of-the-art energy-minimal architectures.
- ...presents Pipestitch, a spatial-dataflow architecture that adds efficient support for lightweight threads.
- ...introduces the `dispatch` control-flow operator to the ISA to launch and manage threads.
- ...adds the SyncPlane, a lightweight control plane for thread synchronization.
- ...provides a programming model, compiler, and microarchitecture that supports lightweight dataflow threads while keeping hardware simple and energy-efficient.
- ...evaluates a complete RTL implementation of Pipestitch, showing that it significantly improves performance while adding small efficiency and programmability overheads.

***Road map.*** Sec. 2 gives background and motivates the need for more performance. Sec. 3 describes Pipestitch, and Sec. 4 presents its implementation. Sec. 5 evaluates Pipestitch, and Sec. 6 concludes.

# 2 BACKGROUND AND MOTIVATION

Multi-year device deployments at the extreme edge perform a variety of increasingly complex sensing applications (e.g., wildlife monitoring, chip-scale satellites, etc.) under a very constrained energy budget [8, 30, 54] . They must harvest energy from their environment or maximize their lifetime on a battery; energy consumption, not power consumption, is the key metric [5]. The challenge is to maximize useful computation within stringent energy constraints.

***Compute at the extreme edge must be efficient and flexible.*** Performing data analysis *on-device* is key to *intelligent* and highly *energy-efficient* sensor systems. The alternative of communicating and offloading work off-device has a high energy cost [15]. Flexibility and programmability are critical to adapt to changing application requirements [14].

***Extreme-edge systems must exploit sparsity.*** Complex DSP and ML requires *sparse* formats (e.g., CSR) to fit in small onboard memories, reduce per-access energy, and eliminate unnecessary computations [3, 19, 21, 29]. *Efficient support for* ***sparse computation*** *is essential* for applications at the extreme edge.

***Existing architectures are too inefficient, too inflexible, or too slow.*** Von Neumann architectures lack the energy efficiency required for on-device processing at the extreme edge, wasting up to 90% of their energy on data movement, instruction fetch, and pipeline control [13, 16, 25]. Alternatively, application-specific integrated circuits (ASICs) sacrifice programmability for high energy efficiency and performance. However, specialization risks obsolescence as sensing applications evolve over a years-long sensor deployment [49]. Accelerators for sparse tensor algebra [24, 38, 39, 41, 46, 51, 66] achieve high performance, but at a cost in flexibility, and the microarchitectural techniques (such as large CAMs, memory reordering, and frequent reconfiguration) are too energy-intensive for the extreme edge. Energy-minimal architectures like RipTide [14] provide efficiency and flexibility, but are slow on sparse workloads. To enable sparse workloads at the extreme edge, a balance of flexibility, energy efficiency, and performance is required.

## 2.1 Architecture for the extreme edge

***CGRAs balance energy efficiency and programmability.*** A considerable body of work has shown *coarse-grained reconfigurable array* (CGRA) architectures [2, 4, 6, 7, 12, 17, 18, 26, 31–37, 40, 42, 43, 45, 47, 48, 50, 52, 53, 55, 57, 58, 60, 62] to be an excellent choice for fast, efficient, and programmable computing. CGRAs *spatially distribute* compute across a grid of processing elements (PEs) that execute operations and communicate intermediate values via a network on-chip (NoC) between PEs. This spatial distribution allows CGRAs to exploit instruction and data locality far more efficiently than von Neumann architectures, effectively eliminating overheads such as instruction fetch/decode, register-file access, and switching activity from a shared pipeline [13]. Unlike ASICs, CGRAs also retain flexibility, and can execute a wide range of code written in high-level languages. To run code on the CGRA, a compiler extracts a dataflow graph (DFG), maps operations onto PEs, and configures the NoC to connect PEs.

***No CGRA currently provides the right blend of speed, efficiency, and programmability for sparse computation at the extreme edge.*** High-performance CGRAs like SPU [7] and Fifer [35] do not focus on the energy efficiency required at the extreme edge. Revel [62] seeks a more balanced blend of performance and efficiency; however, its hybrid design requires tagged-token matching and reordering, which is too costly for the extreme edge. HyCUBE [26, 59] and Ultra-elastic CGRAs [55] achieve a mix of performance and energy efficiency closer to what is required at the extreme-edge, but they still consume too much energy, and are restricted in the applications they can support. HyCUBE only supports affine loops, while UE-CGRA only supports cross-iteration dependences in singly-nested irregular loops.

***Energy-minimal dataflow architectures enable extreme-edge computing.*** Prior work on *energy-minimal dataflow architectures* [13, 14, 16] demonstrated that energy-focused CGRA designs and compilation techniques achieve near-ASIC energy efficiency on programs written in C. The state-of-the-art, RipTide, achieves high efficiency by offloading entire kernels onto the CGRA fabric, which eliminates an Amdahl bottleneck in prior CGRA designs that only target inner loops. RipTide minimizes energy via four key techniques: *(i)* mapping one operation to each PE to minimize switching, *(ii)* disallowing re-ordering to avoid tagging values, *(iii)* eliminating buffers in the NoC, and *(iv)* reusing NoC routers to implement control flow. RipTide's compiler supports arbitrary control flow and memory access, including nested and irregular loops. It maps high-level C to RipTide's CGRA fabric, inserting operators to implement arbitrary control flow while keeping tokens in order through diverging paths, loops, and memory accesses. However, RipTide is unable to execute sparse applications quickly.

## 2.2 Drawbacks of energy-minimal dataflow

While prior energy-minimal dataflow architectures demonstrate that sophisticated computing at the extreme edge is feasible, they ***neglect performance***, which is a *limiting factor* for *large, sparse applications*. There is an opportunity to significantly increase application value by increasing performance on sparse computations.

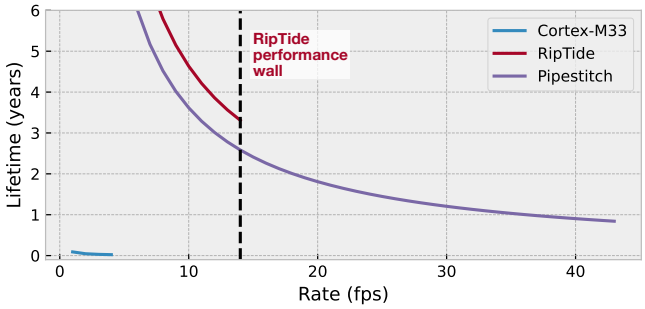



**Figure 3: Device lifetime on a D-cell battery vs. DNN inference rate for a Cortex-M33, RipTide, and Pipestitch. Pipestitch can achieve much higher inference rates than RipTide, while maintaining useful lifetimes at performance targets that both can reach.**

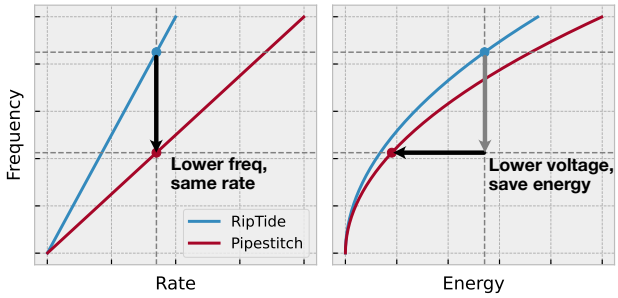



**Figure 4: Applying DVFS to Pipestitch could save energy when performance is not critical. Because Pipestitch does the same work in fewer cycles, it is able to run at a lower clock frequency at iso-performance. This allows voltage to be scaled down, which, as this cartoon shows, could result in significant energy savings.**

***The need for speed.*** We saw in Sec. 1 that poor performance on sparse computation leaves energy stranded in an energy-harvesting deployment. Similar issues affect lifetime on battery-powered deployments. Fig. 3 shows device lifetime while powered by a D-cell battery for the same workload and systems as Fig. 1. To meet the most stringent lifetime requirements, a singular focus on energy-efficiency is required. RipTide lasts nearly 5 years at 10 fps because of its energy-minimal design (see left side of Fig. 1). However, this comes at the expense of performance: RipTide is unable to achieve high framerate. In contrast, Pipestitch achieves much higher framerates and still supports multi-year deployments.

***Performance bottlenecks in sparse computations.*** Sparse computation kernels frequently require irregular loops to traverse compressed representations. Fig. 5 (a) shows the loop nest for counting non-zeros in a map, a representative sparse computation. While the outer loop is regular (i.e., each iteration is independent from all others), the inner loop has data-dependent control, as it must wait for the next value of `p`. RipTide supports this loop nest, but will *conservatively sequentialize* iterations of the outer loop, leading to poor utilization of the inner-loop PEs and poor performance. By contrast, Pipestitch will execute independent outer-loop iterations as threads and *aggressively pipeline* them through inner-loop PEs, improving performance and utilization (Fig. 2 and Fig. 5 (b,c)).

***What about DVFS?*** DVFS is a well established technique for trading performance and energy, but using it to improve performance comes at a great cost in energy. To a first order, energy degrades quadratically with performance, as $P \propto V^2 f$ and $V \propto f$ [63]. Any major scaling in performance incurs a prohibitive energy cost.

Pipestitch improves performance by reducing the number of cycles required to do the same work, independent of voltage and frequency. DVFS is a complementary technique to Pipestitch. Fig. 4 shows a cartoon of DVFS applied to Pipestitch and RipTide. If performance is not a primary concern, Pipestitch can reduce its clock frequency while continuing to match RipTide's performance. Depending on the application and VLSI technology, DVFS could allow Pipestitch to run at *lower* energy than RipTide while maintaining the same performance. (In practice, technology limits how far DVFS can be pushed [55].) Conversely, increasing RipTide's frequency and voltage to match Pipestitch's performance severely degrades energy efficiency.

***Pipestitch provides the right balance between generality, performance, and energy efficiency.*** Flexibility, performance, and energy efficiency are at odds; no prior system achieves the right balance for the extreme edge. Pipestitch is the only architecture that is flexible, performant, and energy-efficient on all the workloads important at the extreme edge. Pipestitch is a new point on the Pareto frontier that achieves *(i)* far higher performance than prior energy-minimal architectures and *(ii)* far higher energy efficiency than prior high-performance architectures. Pipestitch thus sheds light on the fundamental tradeoffs between energy and performance.

## 3 PIPESTITCH OVERVIEW

Pipestitch is an energy-minimal, ordered-dataflow architecture that achieves high performance in sparse applications by adding lightweight dataflow threads to aggressively pipeline independent operations. Pipestitch exposes threads to the programmer through the `foreach` construct, asking the programmer to specify which loops are parallelizable (Fig. 5 (a)). Each iteration of a `foreach` loop becomes a thread, with the loop body being pipelined (Fig. 5 (b)). Pipestitch exposes threads to the hardware with a new dataflow synchronization primitive, called the *dispatch* operator, which ensures correct threaded ordered-dataflow execution (Fig. 5 (b)). To preserve token ordering, `dispatch` operators synchronize over the *SyncPlane*, a lightweight control plane. Threads are pipelined through the dataflow graph mapped onto a single collection of PEs, with operations from different threads executing in parallel on different PEs, improving throughput and utilization (Fig. 5 (b,c)).

***The* `dispatch` *operator.*** Pipestitch's `dispatch` operator (Fig. 6) spawns and continues threads while maintaining ordering. It selects between thread-spawn tokens (from the outer loop) and thread-continuation tokens (from the backedge in the inner loop). A collection of `dispatch` operators sits at the top of the inner loop, with one `dispatch` for each input into the loop body (`p` and `c` in Fig. 5 (b)). The `dispatches` coordinate to either *(i)* inject a *complete* set of input tokens from the outer loop to spawn a new thread (i.e.,

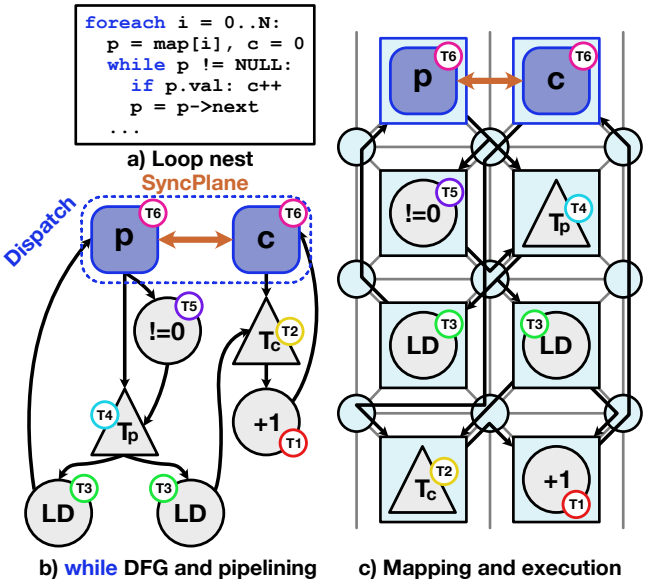



**Figure 5: Pipestitch overview. (a) Pipestitch compiles loop-nests with independent outer-loop iterations, (b) pipelining independent `foreach` iterations as threads on the DFG. (c) Pipestitch maps and executes these DFGs on a CGRA fabric, with threads synchronizing over the SyncPlane, a lightweight control plane.**

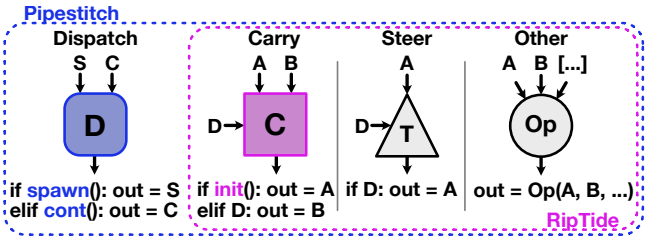



**Figure 6: Pipestitch ISA. Pipestitch adds the `dispatch` operator while continuing to support all RipTide operators. `dispatch` and `carry` operators both sit at the top of a loop body handling loop-carried dependencies, but `dispatch` operators are able to manage many threads in the loop body, while `carry` operators block all but one. `steer` operators forward or drop a token based on a decider. All other operators (e.g., `add`) are represented by a bubble labelled with the op.**

in Fig. 5 (a), `map[i]` and `0` are inputs for `p` and `c`, respectively); or *(ii)* inject a *complete* set of tokens from the inner loop, carrying an existing thread's dependences around another iteration of the loop (the backedges to `p` and `c` in Fig. 5 (b)).

Ordering threads' tokens through `dispatch` operators requires synchronization. Ordering is challenging because different threads' executions of a loop body may follow control-flow paths of different lengths. For correct execution, tokens from different threads must propagate around the loop in the same order that threads were spawned: while running, threads always stay in order. To eliminate stragglers, threads can terminate out-of-order.

***The SyncPlane.*** `dispatch` operators maintain thread ordering by communicating over the SyncPlane, a lightweight synchronization network (Fig. 5 (b,c)). The SyncPlane allows all `dispatch` operators to accept either a complete set of thread-continuation tokens from the inner loop or a complete set of thread-spawn tokens from the outer loop, even though a set of tokens may not all arrive on the same cycle. The SyncPlane conveys a small set of single-bit control signals between all `dispatch` operators, and contains reduction trees to give each `dispatch` a global view of the thread inputs.

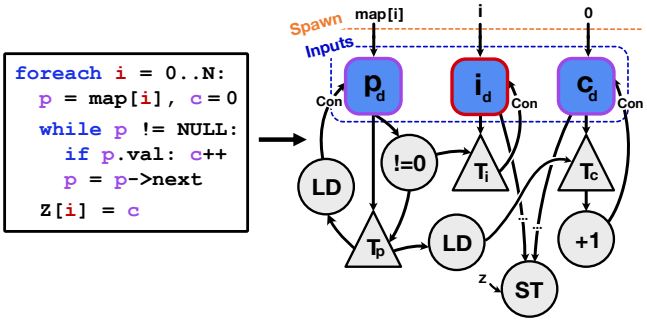



**Figure 7: The Pipestitch software stack. The programmer uses `foreach` to mark independent work. The compiler generates a threaded DFG for the `foreach` loop, inserting dispatches to manage and pipeline threads. (This DFG is partially simplified to highlight the inner loop; e.g., `F` `steer`s, abbreviated by "...", would route live-out values to the `store`.) `dispatch`es are inserted for all carried dependences and invariants of the inner loop: e.g., `p` and `c`, and for the invariant `i`. Each `dispatch` receives a token from the outer loop, representing the thread Spawn point, and a backedge from the inner loop, representing thread continuation (Con).**

***Pipestitch's compiler.*** Finally, Pipestitch provides a compiler that, given a program with independent work marked with `foreach`, produces a DFG containing appropriately-placed `dispatch` operators to allow safe pipelining (see Fig. 7). The *input set* of a thread is the set of all values unique to the thread. A `dispatch` gate is inserted for each value in the input set, and those values are carried through the loop until thread termination. The *spawn point* of a thread is the interface between the outer and inner loops, and is a cut through the edges connected to the `spawn` inputs of the `dispatch` operators making up the input set.

## 4 PIPESTITCH IMPLEMENTATION

Pipestitch supports lightweight threads by bringing together a new ISA for thread control, microarchitecture implementing efficient thread pipelining, and a simple, intuitive programming model for parallelizing code. The programmer writes `foreach` loops and Pipestitch compiles them to lightweight threads that run using the same mapping of a loop-body onto a single collection of PEs. The `dispatch` operator handles thread spawn and ordering. The SyncPlane enables lightweight synchronization between all `dispatch` gates to ensure that tokens remain in order even as threads launch, continue, and terminate. This provides correct threaded execution without the need for tags.

## 4.1 The Pipestitch Programming Model

Pipestitch asks the programmer to explicitly mark independent parallel work. In the vein of other parallel programming models (e.g., OpenMP [1]), the programmer marks outer loops containing independent loop nests with a `foreach` keyword. Pipestitch compiles and executes the program with each iteration of the `foreach` loop executing with sequential semantics as an independent thread. A loop with dependencies that is incorrectly marked `foreach` will execute with undefined semantics.

## 4.2 The Pipestitch Threading Model

Each independent iteration of a marked `foreach` loop in Pipestitch will be executed as a thread. A thread's *live variables* are implicitly defined as the set of values used during the execution of the thread's loop body. The initial values of a thread's live variables flow into the thread when the thread spawns at the spawn point in the DFG. Updated values of a thread's live variables may also flow back to the thread's spawn point via loop-carried dependences (i.e., control backedges), allowing inner loops to be irregular (Fig. 7). Each thread receives its own live variables. A thread may update any of its live variables in each iteration, and the live variables are a thread's *only* state. They are maintained as the flow of values between operations that manipulate these variables, living only in PE buffers.

Pipestitch can spawn a new thread when the initial value of each of the thread's live variables is available at the spawn point. If a new thread's initial values are available every cycle, Pipestitch can spawn a new thread on every cycle. A thread terminates when no more values flow through the DFG representing its computation.

## 4.3 Executing Lightweight Dataflow Threads

Pipestitch pipelines a thread's work through the single set of PEs onto which the compiler mapped the thread's loop body (Fig. 8 (b)), eliminating pipeline bubbles that would otherwise be present (Fig. 8 (a)). During execution, each thread's tokens flow from the top of the thread body (from the spawn point for a new thread, or via a backedge for an existing thread) and traverse through the mapped PEs. After filling the pipeline of PEs with different threads' operations, the system runs fully pipelined, with operations from different threads firing on consecutive cycles in each PE.

## 4.4 Synchronized Threads and Ordering

Pipestitch adds the *dispatch operator*, a new ISA operator that spawns and continues threads while maintaining inter-thread ordering (Fig. 6). Each live variable (initially the input set) in a thread corresponds to a single `dispatch` operator (Fig. 7). Throughout execution, `dispatch` operators select between spawn and continuation tokens to keep the pipeline full while maintaining ordered dataflow.

***Single-input threads.*** A single-input thread will have a single `dispatch` operator for that input. Without multiple live variables to synchronize, the `dispatch` can always select either a spawn token or a continuation token without breaking ordering. Since each thread is independent and each thread's only state is that single live variable, any interleaving of spawn and continuation tokens at the `dispatch` will provide correct execution. In this case, the continuation token is arbitrarily preferred.

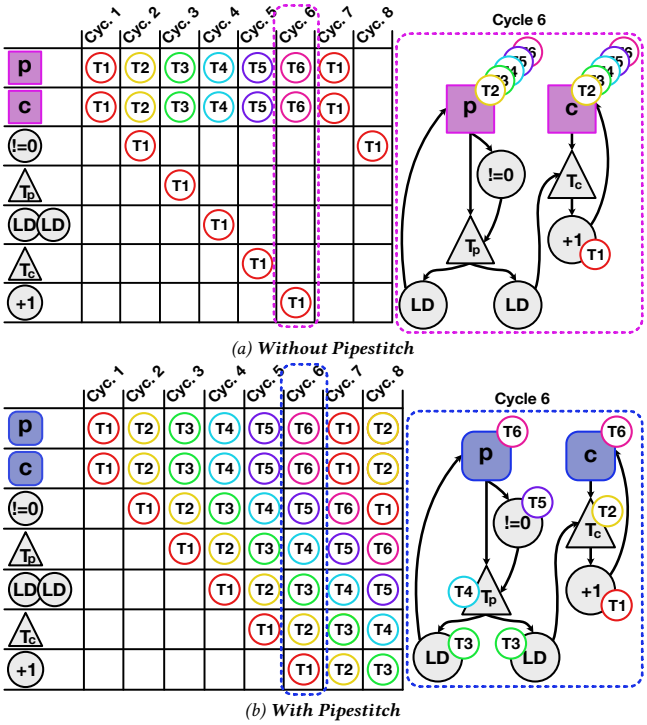



**Figure 8: Execution of threads on a DFG with and without Pipestitch. (a) Without Pipestitch, threads (represented by circles) are conservatively blocked at the head of the inner loop. Only one thread executes in the inner loop at any time, leading other threads to wait at the top, causing poor utilization and performance. (b) With Pipestitch, independent threads are unblocked and pipelined through the inner loop, leading to high utilization and performance.**

***Multi-input threads.*** A thread with $n$ inputs will have $n$ `dispatch` operators, one for each input. With multiple live variables involved, synchronization is required. Spawn and continuation tokens are each ordered: all of thread 1's spawn tokens will arrive before any of thread 2's spawn tokens, and vice versa. However, *spawn and continuation tokens are not synchronized*: nothing can be said about when thread 1's continuation tokens will arrive in relation to when thread 2's spawn tokens arrive. Furthermore, with carried-dependence chains in the DFG having different lengths, it is possible that continuation tokens arrive on different cycles. Fig. 9 shows this, and (a) shows an ordering violation (and incorrect execution) that can occur if `dispatch` greedily accepts any available tokens. Thread 2's `p` continuation token arrives at the corresponding `dispatch` on cycle 2, but its `c` continuation token does not arrive until cycle 4. Meanwhile, thread 3's spawn tokens for both `p` and `c` arrive at the corresponding `dispatch`es on cycle 3. The greedy `dispatch`es will accept thread 2's `p` continuation token on cycle 2, both of thread 3's spawn tokens on cycle 3, and thread 2's `c` continuation token on cycle 4. The `dispatch` operators have different output orderings, leading to incorrect execution.

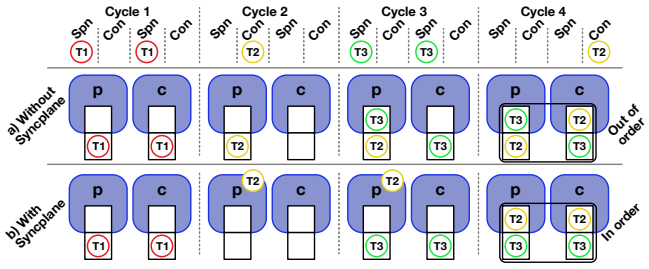



**Figure 9: Pipestitch's `dispatch` operator controls thread launch and execution by selecting between thread-spawn and thread-continuation tokens, and provides synchronization to ensure ordering is preserved. If carried-dependence chains in the thread body have different lengths, spawn and continuation tokens can arrive at different times (yellow tokens on cycles 2 and 4). (a) If `dispatch` operators greedily accept tokens as they arrive, ordering violations can occur. (b) Pipestitch ensures correct execution by synchronizing among `dispatch` operators over the SyncPlane, allowing tokens to be held until they can be safely accepted.**

To overcome this, `dispatch` operators must synchronize, making sure that a complete set of thread inputs are accepted at the same time. Fig. 9 (b) shows this. On cycle 2, thread 2's `p` continuation token will be held, because the `dispatch` gate is aware that the corresponding continuation token for `c` is not available. The token will continue to be held through cycle 3 as the corresponding `c` token is still not available. Since both of thread 3's spawn tokens are available, those tokens will be accepted. On cycle 4, thread 2's `c` continuation token becomes available, so both of thread 2's tokens are accepted. This leaves the `dispatch` operators agreeing on the ordering of threads, allowing correct execution. If both a complete set of spawn tokens and a complete set of continuation tokens is available, it is inconsequential which is accepted, and the continuation set is arbitrarily preferred.

***Avoiding deadlock.*** The number of threads that run in parallel is limited by the total number of buffers in the PEs executing the threads. If accepting a set of spawn tokens would fill the last buffers in the thread body, deadlock would occur because the continuation tokens cannot be accepted by the `dispatch` gates. To avoid this, we draw from bubble flow control [44], and ensure that `dispatch` operators all have two available buffer slots before accepting a new spawn token.

***Token-selection logic.*** Putting together the conditions for ensuring ordering and avoiding deadlock, Fig. 10 shows the logic for `dispatch` token selection.

## 4.5 Pipestitch Microarchitecture

Fig. 11 shows the microarchitecture of Pipestitch. It leverages the existing RipTide microarchitecture to allow fair comparison, but is not specifically tied to it. The fabric is an array of heterogeneous PEs, a statically-routed, circuit-switched, combinational data NoC, and a combinational SyncPlane for synchronization between dispatch gates. PEs buffer tokens at their inputs, with a buffer for each input. As will be discussed, control-flow (CF) and memory PEs also have an output buffer. The PE is divided into a common μCore for

```
1    if AND(valid(cont)) and not OR(full(out))
2        push(cont)
3    else if AND(valid(spawn)) and AND(twoSlots(out))
4        push(spawn)
```

**Figure 10: `dispatch` token-selection logic. The `cont` token is selected if all `dispatch` operators have that token available and at least one free slot in their output buffers. The `spawn` token is selected if all `dispatch` operators have that token available and at least *two* free slots in their output buffers. If both tokens could be selected, Pipestitch prefers the `cont` token.**

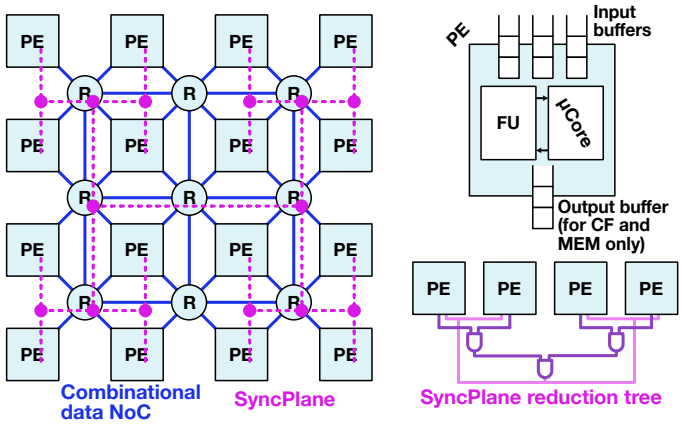



**Figure 11: Pipestitch microarchitecture. Pipestitch is a fabric of heterogeneous PEs connected by a combinational, circuit-switched NoC for communicating data, and a combinational SyncPlane for communicating control signals between `dispatch` operators. PEs buffer tokens at their inputs, and some PE types additionally buffer their outputs. The SyncPlane is a reduction tree that connects all control-flow PEs. For simplicity, heterogeneity of PEs is not shown.**

control and interfacing with the NoC, and a PE-specific functional unit (FU). Routers are combinational switches with additional logic to manipulate the routing table for control-flow-in-NoC. The SyncPlane carries and reduces control signals between all CF PEs (to which `dispatch` operators are mapped).

## 4.6 Synchronization with the SyncPlane

To communicate control signals between `dispatch` operators, Pipestitch uses a secondary control plane, the SyncPlane. It carries each of the one-bit control signals used in thread synchronization (Fig. 10) from the `dispatch` gates to a reduction tree, returning the reductions to each `dispatch`. Implementing this as a secondary plane with built-in reductions allows inter-`dispatch` synchronization to conserve resources. The alternatives of using wide data links to carry one-bit control signals, or adding reduction nodes to the DFG, would increase mapping time and consume valuable links and PEs.

## 4.7 Reducing Pipeline Stalls

Pipeline stalls across thread operations are a major performance consideration in Pipestitch. To maximize throughput, Pipestitch must keep threads running smoothly through the pipeline with

a minimum of stalls. Because Pipestitch does not re-order tokens across threads, a stall in any thread will delay all threads. Sources of stalls include transient effects, such as memory-bank conflicts, and throughput mismatches due to split-join subgraphs and carried-dependence chains of different lengths in the DFG.

To minimize the impact of stalls, Pipestitch buffers tokens at the input of PEs (destination buffering). While buffering tokens at the output (source buffering) requires only a single copy of each token, it leads to stalls if consumers do not fire at the same time or if consumers do not have all input tokens available. This is especially problematic for imbalanced split-joins. Fig. 12 (a) shows that source buffering on an imbalanced split-join results in unavoidable stalls, reducing throughput. Fig. 12 (b) shows that destination buffering eliminates stalls and achieves full throughput, at the cost of some additional buffering for high fan-in operations.

There are two exceptions. First, memory PEs initiate loads and then must wait some cycles before data is returned. If a downstream operator exerts backpressure after the load initiates, the PE must have somewhere to put the data when it comes back from memory, so memory PEs are provisioned output buffers. This extra sequential stage increases latency, so these output buffers are bypassed if downstream is not exerting backpressure. Second, `dispatch` operators need to reason about the state of their output buffer as part of their ordering-preservation logic. As an implementation detail, reasoning about the state of another PE's input buffer would be infeasible in our current microarchitecture, so we provision each CF PE with an output buffer. We also observe that provisioning this extra buffering helps performance, as CF PEs are particularly sensitive to stalls.

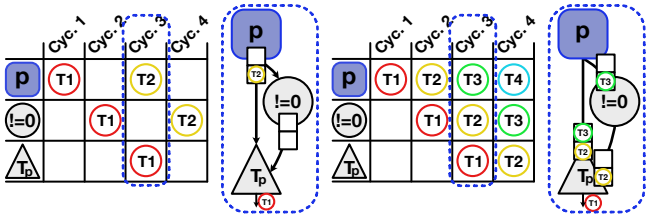
## 4.8 Compiling for Pipestitch

Pipestitch compiles applications written in C to a DFG by extending RipTide's compiler to support `foreach` and `dispatch`. To parallelize `foreach` loops, Pipestitch adds a compiler pass that recognizes the `foreach` loop nest and inserts `dispatch` operators in the DFG.

***Inserting* `dispatch`.** `dispatch` operators are added for all thread tokens *generated in the outer loop* that are *live in a dataflow through the inner loop*, where there is potential for ordering violations when decoupling and pipelining outer-loop iterations (and their inner-loop instances). These values come in the form of loop-carried dependences and loop invariants of the inner loop, which we define as the thread's input set.

The compiler first inserts `dispatch` operators into the DFG for loop-carried dependences of the inner loop. The initial value of a carried dependence is set to the `spawn` input of the `dispatch` operator in the DFG. The `continue` input is set to the backedge in the DFG that produces the carried value in the inner loop. Fig. 7 shows this in practice; `dispatches` $\texttt{p}_\texttt{d}$ and $\texttt{c}_\texttt{d}$ are inserted for carried dependences `p` and `c`. $\texttt{p}_\texttt{d}$ has its `spawn` input set to `map[i]` and its `continue` input to the backedge arriving from $\texttt{T}_\texttt{p}$. $\texttt{c}_\texttt{d}$ is configured identically.

The compiler handles loop invariants similarly, inserting an additional `dispatch` and `steer` to reproduce the token for the lifetime of the thread. The invariant token and the loop exit condition serve as inputs to this steer, which then feeds the `continue` input of the `dispatch` in the DFG. Fig. 7 demonstrates this transformation;




*(a)* ***Source buffering*** *(b)* ***Destination buffering***

**Figure 12: Source buffering limits throughput in imbalanced split joins. (a) With source buffering, an operator that does not have all of its input tokens available must stall upstream to keep the token from being dropped. This leads to mandatory stalls in an imbalanced split join. (b) With destination buffering, the operator can stash the inputs that are ready, freeing upstream to continue execution, and achieving full throughput.**

`dispatch` $\texttt{i}_\texttt{d}$ is inserted for the loop invariant `i`. In this example, `i` simply passes in a dataflow through the inner loop so the thread can use it to execute `Z[i] = c` afterwards. $\texttt{i}_\texttt{d}$'s `spawn` input is set to `i` and its `continue` input is set to the backedge arriving from $\texttt{T}_\texttt{i}$, inserted specifically for $\texttt{i}_\texttt{d}$.

***Increasing token storage.*** To provide enough storage for live values, Pipestitch maps all control-flow operators to PEs, rather than routers, so that each operator is allocated token storage.

***Selectively enabling* `dispatch`.** Loops with initiation interval (II) of 1 perform perfectly, eliminating any potential benefit from threading. Invoking threading in this case will force control flow to map to PEs, reducing array efficiency with no performance gain. We adopt a heuristic of only invoking `dispatch` pipelining if inner-loop II $> 1$. Control-flow executing combinationally in routers does not increase II, so II is the number of non-control-flow operators in the longest cycle of the inner loop.

This heuristic is effective, but it is not perfect: a program with an inner-loop II of 1 but a poorly-balanced outer loop could also benefit from threading. Threading forces control flow to PEs, which actually benefits this scenario; it decouples unbalanced paths in the outer loop and increases inner-loop II. Threads would be the better option to gain performance now that the inner-loop II $> 1$. We do not consider this observation for our heuristic and leave this optimization for future work.

***Mapping.*** CGRA mapping is a hard problem [13, 14, 28, 64], but RipTide finds a SAT-based solution fast while producing sufficient mapping quality. We adopt RipTide's solution and extend it with an override to keep CF on PEs. Additionally, we add rules to keep CF operators on PEs if they follow bypassing memory operators, even if CF in NoC is requested. Doing so avoids a combinational loop between bypass and CF in NoC.

# 5 EVALUATION

We evaluate a full implementation of Pipestitch with a compiler and RTL hardware description. We find that Pipestitch substantially improves performance with small energy overhead.

## 5.1 Methodology

Pipestitch is implemented entirely in RTL, building on RipTide [14] for convenience and to allow fair comparison of energy overhead. Pipestitch adds the `dispatch` operator, SyncPlane, destination buffering, and bypass for memory units. Pipestitch's compiler adds analysis of loop nests and `dispatch` insertion to RipTide's compiler in LLVM 12.0.0. Pipestitch's mapper adds the option to map all control flow on PEs to RipTide's SAT mapper. We heuristically compile programs with threads if II>1, and without threads otherwise.

We compare Pipestitch to running on the small RISC-V control core, and to running on RipTide. Both RipTide and Pipestitch use an $8 \times 8$ fabric with identical PE mix: 16 arithmetic, 2 multiply, 28 control flow, 14 memory, and 4 stream PEs (see [14]). Both Pipestitch and RipTide are configured with a buffer depth of 4, and all designs have a 256kB memory and a 50MHz clock. We synthesize all designs in a commercial, sub-28nm process using Genus 20.11. We simulate full applications on the synthesized design using Xcelium 22.03 to obtain toggle counts and use those counts to annotate the design in Joules 20.11 and obtain energy estimates. This methodology provides reasonably accurate estimates of energy and power, but it does have limitations in modeling wiring, clock distribution, and glitching. While these limitations affect absolute measurements, they should not significantly affect trends between the designs.

## 5.2 Applications

We evaluate six kernels from image processing and linear algebra, and a 4-layer sparse DNN with a 235 kB memory footprint as a full application. Our benchmark kernels are dense matrix multiplication (DMM), sparse matrix dense vector multiplication (SpMV), dithering (Dither), sparse matrix slicing (SpSlice), sparse matrix sparse vector multiplication with dense output (SpMSpVd), and sparse matrix sparse matrix multiplication with dense output (SpMSpMd). The 4-layer DNN is designed to classify MNIST, and is composed of SpMSpVd and fused sparsify/ReLU. Layer sparsities range from 97% to 75%. We evaluate each benchmark on random inputs; further details are in Table 1.

These applications showcase Pipestitch's advantages over RipTide for irregular applications, while also demonstrating Pipestitch's ability to run regular applications at the same or better performance. DMM and SpMV are regular, with affine loop nests and II=1; we execute both *without* threads, running similarly to RipTide. These applications are well-supported by RipTide; they show that Pipestitch retains the efficiency of RipTide in its best domain. Dither has an affine innermost loop, but the body of the loop contains irregular control-flow which prevents inner-loop pipelineing. SpSlice, SpMSpVd, and SpMSpMd operate on sparse inputs and all have irregular control-flow with long dependence cycles and non-affine loop nests. RipTide struggles to extract high performance from these irregular applications, and they show the diversity of applications for which Pipestitch is beneficial.

## 5.3 Pipestitch is Fast

Pipestitch achieves a geomean speedup of 2.55× across all applications, and a geomean speedup of 3.49× across the five workloads that benefit from threading. Fig. 13 shows speedup for Pipestitch and RipTide over the scalar baseline. Pipestitch enjoys high performance in the five sparse applications with irregular control and long inner-loop II. In comparison, RipTide's carry operations block loop pipelining on these long paths, limiting performance and leaving resources under-utilized.

**Table 1: Benchmark parameters**

| Benchmark | Input size | Sparsity | Threaded? |
|---|---|---|---|
| DMM | $64 \times 64$ | — | ✗ |
| SpMV | $64 \times 64$ | 0.90 | ✗ |
| Dither | $128 \times 128$ | — | ✓ |
| SpSlice | $64 \times 64$ | 0.89 | ✓ |
| SpMSpVd | $128 \times 128$ | 0.90 (matrix & vector) | ✓ |
| SpMSpMd | $64 \times 64$ | 0.89 (both matrices) | ✓ |
| DNN | $28 \times 28$ | 0.75 - 0.97 | ✓ |

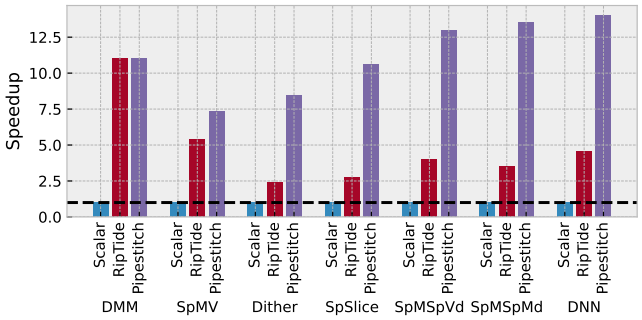



**Figure 13: Speedup compared to scalar for RipTide and Pipestitch. Per our II heuristic, we run DMM and SpMV as unthreaded applications on both RipTide and Pipestitch. Pipestitch maintains performance on DMM, and improves performance on SpMV with destination buffering reducing stalls on an imbalanced split-join. The remaining four kernels have irregular control, and threaded configurations are selected for Pipestitch, which achieves significant speedup. DNN on Pipestitch uses threaded SpMSpVd, and unthreaded sparsify/ReLU.**

## 5.4 Pipestitch has Low Energy Overhead

Pipestitch improves performance with a small energy overhead: a geomean increase of 1.11× across all apps, and 1.05× for threaded apps. Fig. 14 shows energy for Pipestitch and RipTide compared to the scalar baseline. Pipestitch incurs overhead with destination buffering and mapping CF onto PEs in threaded applications, but this cost is low for the performance gains achieved. Pipestitch has higher average power consumption than RipTide, with a geomean power increase of 3.66× across the five threaded applications, but energy is the key metric for the extreme edge.

## 5.5 Pipestitch Improves EDP

Pipestitch has a lower EDP for threaded applications, with a geomean reduction of 2.29× across applications. Fig. 15 shows EDP for Pipestitch and RipTide, normalized to RipTide. Pipestitch's large speedup and small energy cost allows it to reduce EDP for all threaded benchmarks. Because DMM already runs well on RipTide,

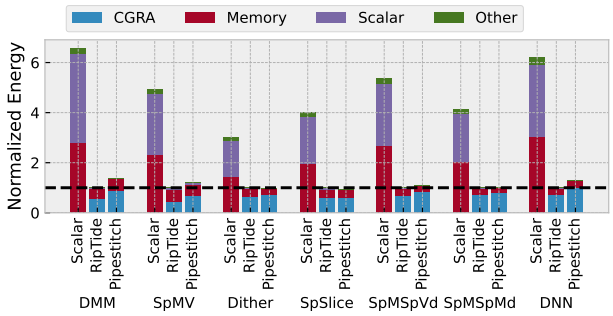



**Figure 14: Normalized energy for Pipestitch and RipTide compared to the scalar baseline. Pipestitch incurs a cost with its destination buffering and CF on PEs, but this cost is low for the performance gains achieved. DMM sees the greatest increase in energy because RipTide already runs this application with good performance.**

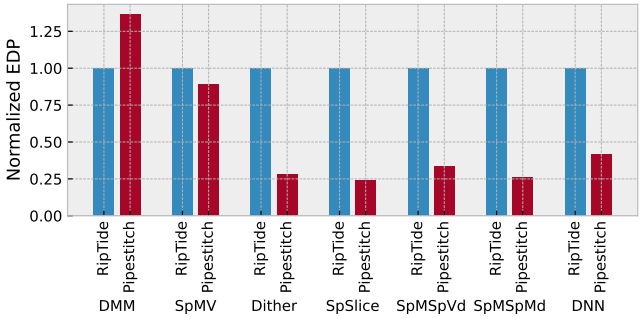



**Figure 15: Normalized EDP for Pipestitch compared to RipTide. Pipestitch improves EDP for all threaded apps due to its large speedup and small energy cost. For DMM, Pipestitch maintains the performance of RipTide while incurring an energy cost with destination buffering.**

Pipestitch is only able to match performance, and the energy cost of destination buffering leads to an increase in EDP.

## 5.6 Pipestitch has Low Area Overhead

Pipestitch's implementation is small, with a reasonably small total area of approximately 1.0 $mm^2$. Fig. 16 shows an area breakdown for Pipestitch, showing that Pipestitch's area is primarily dedicated to its compute fabric (65.2%), with its 256kB memories making up 33.2% of the total design. Pipestitch's fabric is 1.10× larger than RipTide's, due to the additional buffering and SyncPlane.

## 5.7 Pipestitch Increases IPC

By aggressively pipelining operations from different threads, Pipestitch increases the number of executed instructions per cycle (IPC), with a geomean IPC increase of 2.80×. We measure IPC as the total number of times all PEs fired during execution divided by the total number of cycles executed. Fig. 17 shows IPC for Pipestitch and RipTide across kernels. Pipestitch achieves a significant improvement in IPC on threaded benchmarks, with a geomean IPC improvement of 4.30×. DMM and SpMV do not use

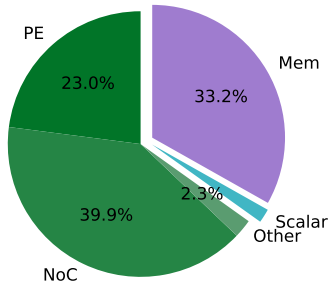



**Figure 16: Area breakdown of Pipestitch. Total area is approximately 1.0 $mm^2$ and is mostly spent on the CGRA fabric in the NoC.**

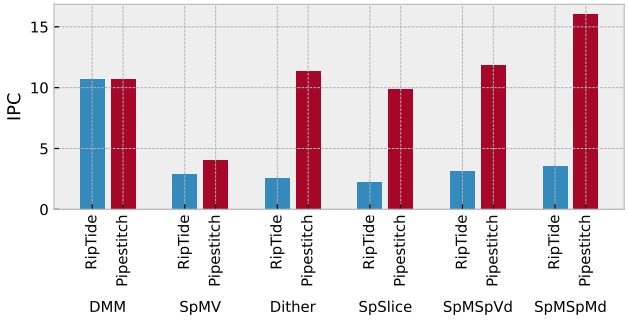



**Figure 17: IPC across kernels for RipTide and Pipestitch. IPC is defined as the total number of times all PEs fire divided by the total number of cycles. Pipestitch achieves a significant improvement in IPC on the four threaded kernels.**

threads, executing their affine, streamed loops with a full pipeline. The data show that for stream-friendly applications that do not use threads, Pipestitch's IPC meets RipTide's.

For threaded applications, Pipestitch substantially improves inner-loop IPC. Fig. 18 shows per-unit IPC for Pipestitch and RipTide, which is IPC normalized by the number of PEs required to execute. The figure breaks per-unit IPC between inner and outer loops. For a loop (inner or outer), its per-unit IPC is the total number of instruction firings by PEs mapped to that loop divided by the total number of cycles (i.e., loop IPC) normalized by the number of PEs that execute that loop. Per-unit IPC captures the underutilization of infrequently firing PEs. Outer-loop PEs that fire once per inner-loop execution have low IPC, underutilizing the PE. Inner-loop PEs that fire every cycle have high IPC, fully utilizing the PEs. Pipestitch's main improvement is to enable multiple threads' inner-loop work to execute in parallel, with more inner-loop PEs executing their operation on each cycle. On the threaded benchmarks, Pipestitch achieves a 3.62× improvment in inner-loop IPC. Pipestitch also improves outer-loop IPC, but by less than inner-loop IPC. On the threaded benchmarks, Pipestitch achieves a 3.51× improvement in outer-loop IPC. Outer-loop IPC improvement is limited by the outer loop's ability to spawn new threads. Long-running inner-loop threads may preclude future thread spawns, limiting outer-loop IPC at a cost in utilization, but not in performance.

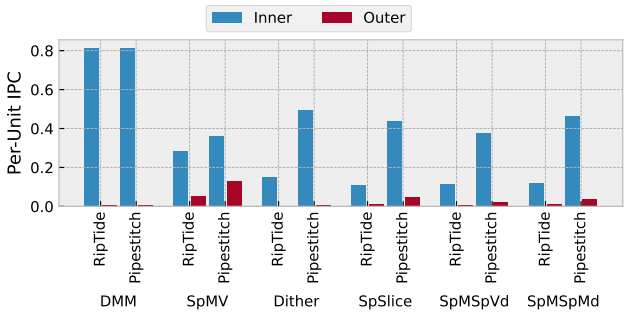



**Figure 18: IPC per PE for inner and outer loops. IPC per PE is the number of times all PEs mapped to the loop fire, divided by the total number of cycles and the number of PEs mapped to the loop.**

## 5.8 Pipestitch Balances Speed and Overhead

Pipestitch departs from RipTide's energy-focused microarchitecture, enabling peformance with a small or negligible energy overhead. Two major differences between Pipestitch and RipTide are that Pipestitch uses input buffers, while RipTide uses output buffers, and Pipestitch maps control-flow operations onto PEs, while RipTide maps them into its NoC. Fig. 19 characterizes these differences and their performance, showing that Pipestitch's design choices prioritize performance without introducing a high energy overhead. The data characterize the cost of source vs. destination buffering, using normalized performance for RipTide, Pipestitch, and PipeSB, a Pipestitch variant that uses source buffers (like RipTide). PipeSB incurs a geomean slowdown of 1.13× compared to RipTide. The problem is backpressure on multicast outputs; any imbalance in paths will result in backpressure, and the token must be held in the source buffer until backpressure clears, thus stalling all paths, not just the slow path.

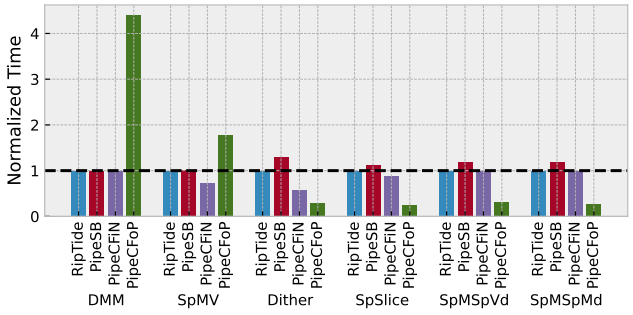



**Figure 19: Normalized time across kernels with RipTide, Pipestitch, and PipeSB, a source-buffered fabric with `dispatch` and the Sync-Plane. For Pipestitch, we further divide results into PipeCFiN, where control-flow operations are mapped into the NoC when possible, and PipeCFoP, where all control-flow operations are mapped onto PEs (i.e., using the same hardware, but different mappings). PipeCFiN and PipeCFoP follow the II heuristic, so DMM and SpMV are unthreaded kernels for both.**

The plot also shows the effect of Pipestitch's choice to put threaded applications' control-flow operations on PEs (PipeCFoP), instead of using RipTide's control-flow-in-NoC mechanism (PipeCFiN). PipeCFoP and PipeCFiN use the same hardware, but different compilations. DMM and SpMV are run unthreaded for both. The data show that one of PipeCFiN or PipeCFoP is always fastest: PipeCFiN on non-threaded applications, as CFiN maintains low inner-loop II, and PipeCFoP for threaded applications, as CFoP provides buffering to support deep pipelines. Recall that Pipestitch reasons heuristically about whether to thread an application, and an application with II=1 (e.g., DMM and SpMV) compiles control-flow operations into the NoC and does not use threads.

## 5.9 Pipestitch can Exploit More Buffering

To maximize throughput, Pipestitch needs enough buffering to hide imbalanced split-join and carried-dependence-chain lengths. Fig. 20 shows speedup as buffer depth is increased beyond 4, as used elsewhere. Performance improves as imbalances are resolved, at which point additional buffering ceases to help. Increasing buffer depth will come at a cost in area and energy, providing a tradeoff between performance and energy that can be made at design time.

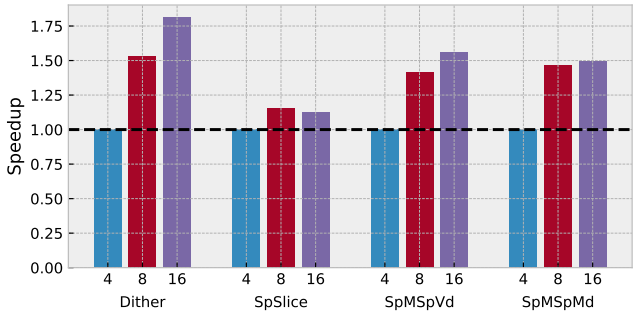



**Figure 20: Speedup with increasing buffer depth for threaded kernels. Increasing buffer depth improves performance, but comes with a cost in area and energy. All other experiments use a depth of 4 to provide fair comparison with RipTide.**

As an alternative to increasing hardware buffer size, the compiler could attempt to mitigate imbalance in the DFG. Long paths are often fundamentally long, giving the compiler little option for rearranging the graph to avoid imbalance, but the compiler could pad short paths with no-op nodes which provide buffering, at the cost of a larger graph that makes mapping more difficult and requires more PEs.

## 5.10 Pipestitch's Compiler Quickly Produces High-Quality Mappings

Pipestitch compiles and maps threaded programs in a reasonable amount of time, with a cost in more PEs consumed, but a benefit in performance due to better pipelining. Fig. 21 shows the number of PEs generated by the compiler for each of our workloads across three configurations: RipTide, Pipestitch with control flow in the NoC (PipeCFiN), and Pipestitch with control flow on PEs (PipeCFoP). By default, RipTide maps control-flow operations to the NoC.

PipeCFiN has only a modest increase in PE count, but has poor performance for threaded kernels because it lacks in-PE buffering

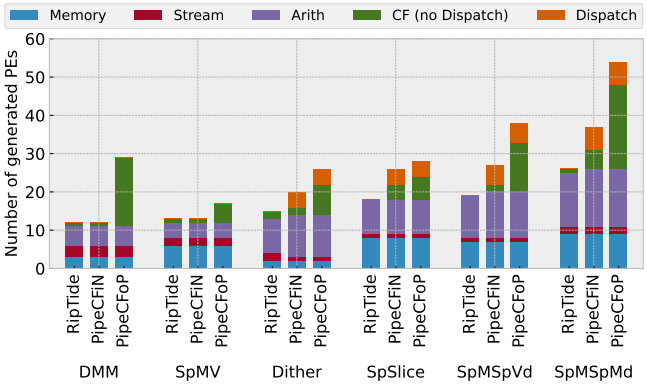



**Figure 21: Operator counts across all benchmark kernels for RipTide and Pipestitch. For Pipestitch, results are split into PipeCFiN, (control flow mapped to the NoC when possible), and PipeCFoP (all control flow mapped to PEs). To achieve better performance, Pipestitch consumes more PEs, especially for control flow.**

to support deep pipelines. For the four threaded benchmark kernels (Dither, SpSlice, SpMSpVd, SpMSpMd), PipeCFiN increases PE count 28% on average over RipTide. Operator counts for DNN (not shown in Fig. 21) are identical to SpMSpVd, which is the largest kernel used in DNN. The 28% increase is due to the addition of `dispatch` operators, each of which maps to a PE. Compounding the cost, a `dispatch` gate prevents instruction fusion optimizations, such as affine stream generator insertion, leaving unfused instructions on their own PEs. PipeCFiN also restricts which control-flow operations map to the NoC to avoid combinational loops. Due to microarchitecural details, a combinational loop can be activated if a downstream control-flow operation takes an input from an upstream memory unit that performs a bypass. Pipestitch disables the combinational loop, requiring the involved operation to map to a PE.

PipeCFoP has a steeper cost in PE count, but the ample buffering in these PEs supports deep pipelines and high performance for threaded workloads. PipeCFoP requires 33% more PEs over PipeCFiN and 70% more PEs over RipTide on average, which is due to mapping all control flow onto PEs. While PipeCFoP consumes more PEs for operations, *this tradeoff is a win for performance when threading*, especially for irregular applications with imbalanced dependence paths — as demonstrated in Sec. 5.8 and Fig. 19.

For non-threaded benchmarks (DMM and SpMV), `dispatch` operators are not inserted and Pipestitch's restrictions for in-NoC control flow do not force any control-flow operations onto PEs. Consequently, PipeCFiN consumes the same number of PEs as RipTide. Sec. 5.8 and Fig. 19 show that Pipestitch with control flow in-NoC is best for non-threaded kernels, so there is *no tradeoff* in the number of generated PEs to achieve performance in this configuration.

## 6 CONCLUSION

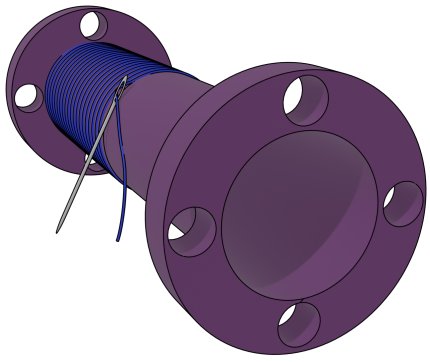
We have presented Pipestitch, an energy-minimal dataflow architecture with support for lightweight threads to improve performance without sacrificing energy efficiency. Pipestitch exploits `foreach` parallelism to scale performance on sparse computations, fully exploiting all available energy in extreme-edge deployments to maximize application value. Pipestitch explores the tradeoff between performance and efficiency, and provides a new point on the Pareto frontier. Pipestitch improves performance by 3.49× over the state-of-the-art while increasing area by just 1.10× and energy by just 1.11×.

Pipestitch opens multiple directions for future work to further improve performance and ease compilation. For small kernels, future designs could "unroll" multiple copies of the inner-loop and distribute outer loop iterations spatially in addition to temporal pipelining. This would require `dispatch` gates to synchronize across multiple instances of the same DFG. Future designs could also selectively time-multiplex low-utilization operations on PEs, freeing PEs for other work. Time-multiplexing trades performance for energy by increasing switching activity; choosing when and what to time-multiplex is an unexplored dimension of this tradeoff.

Another challenge for CGRA architectures is the feasibility and latency of compilation. CGRA compilation is challenging partly because it requires whole-program optimization to place-and-route the program. Pipestitch takes an initial step towards modularizing compilation by compiling the inner-loop nest once and sharing it among multiple "calling" threads (i.e., outer-loop iterations). Future work could develop this into a full spatial ABI for sharing modular code blocks on a dataflow fabric.

## ACKNOWLEDGEMENTS

We thank the reviewers for their time and thoughtful feedback. This work was supported by NSF CCF-1815882, and by Semiconductor Research Corporation (SRC) Artificial Intelligence Hardware (AIHW), a Global Research Collaboration (GRC) program. Nathan Serafin was supported by Apple's Fellowship in Integrated Systems, and Souradip Ghosh by the U.S. Department of Energy Computational Science Graduate Fellowship (DESC0022158).



# REFERENCES


[1] 2021. OpenMP Application Programming Interface 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[2] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: A Systematic Framework for Heterogeneous CGRA Realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 918–932. https://doi.org/10.1145/3503222.3507772

[3] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Haotian Tang, Hanrui Wang, Ligeng Zhu, and Song Han. 2022. Enable Deep Learning on Mobile Devices: Methods, Systems, and Applications. *ACM Trans. Des. Autom. Electron. Syst.* 27, 3, Article 20 (mar 2022), 50 pages. https://doi.org/10.1145/3486618

[4] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 184–189.

[5] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *ASPLOS*.

[6] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A Fully Pipelined and Dynamically Composable Architecture of CGRA. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 9–16. https://doi.org/10.1109/FCCM.2014.12

[7] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 924–939.

[8] Bradley Denby and Brandon Lucia. 2020. Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System. In *ASPLOS 25*.

[9] Brad Denby, Emily Ruppel, Vaibhav Singh, Shize Che, Chad Taylor, Fayyaz Zaidi, Swarun Kumar, Zac Manchester, and Brandon Lucia. 2022. Tartan Artibeus: A Batteryless, Computational Satellite Research Platform. (2022).

[10] Harsh Desai and Brandon Lucia. 2020. A Power-Aware Heterogeneous Architecture Scaling Model for Energy-Harvesting Computers. *IEEE Computer Architecture Letters* 19, 1 (2020).

[11] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. 2022. Camaroptera: A Long-Range Image Sensor with Local Inference for Remote Sensing Applications. *ACM Trans. Embed. Comput. Syst.* 21, 3, Article 32 (may 2022), 25 pages. https://doi.org/10.1145/3510850

[12] Milovan Duric, Oscar Palomar, Aaron Smith, Osman Unsal, Adrian Cristal, Mateo Valero, and Doug Burger. 2014. EVX: Vector execution on low power EDGE cores. In *DATE*.

[13] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. 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, 1027–1040.

[14] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–564. https://doi.org/10.1109/MICRO56248.2022.00046

[15] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *ASPLOS*.

[16] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. MANIC: A Vector-Dataflow Architecture for Ultra-Low-Power Embedded Systems. In *MICRO*.

[17] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. 2000. PipeRench: A reconfigurable architecture and compiler. *Computer* 33, 4 (2000).

[18] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012).

[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[20] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[21] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*.

[22] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems* 28 (2015).

[23] Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price. 2012. Simple and practical algorithm for sparse Fourier transform. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 1183–1194.

[24] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[25] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *ISSCC*.

[26] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *DAC*.

[27] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. 2004. The vector-thread architecture. In *ISCA 31*.

[28] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. 2022. LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 444–459.

[29] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-Device Training Under 256KB Memory. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[30] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. https://doi.org/10.4230/LIPIcs.SNAPL.2017.8

[31] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*. Springer, 61–70.

[32] Ethan Mirsky, Andre DeHon, et al. 1996. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources.. In *FCCM*, Vol. 96. 17–19.

[33] Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. 2006. Tartan: evaluating spatial computation for whole program execution. *ACM SIGARCH Computer Architecture News* 34, 5 (2006).

[34] Takashi Miyamori and Kunle Olukotun. 1999. REMARC: Reconfigurable multimedia array coprocessor. *IEICE Transactions on information and systems* 82, 2 (1999), 389–397.

[35] Quan M Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1064–1077.

[36] Chris Nicol. 2017. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing. *WaveComputing WhitePaper* (2017).

[37] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *ISCA 44*.

[38] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. 2021. SparseAdapt: Runtime control for sparse linear algebra on a reconfigurable accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1005–1021.

[39] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[40] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. 2013. Triggered instructions: a control paradigm for spatially-programmed architectures. *ACM SIGARCH Computer Architecture News* 41, 3 (2013).

[41] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH computer architecture news* 45, 2 (2017), 27–40.

[42] Hyunchul Park, Yongjun Park, and Scott Mahlke. 2009. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 370–380. https://doi.org/10.1145/1669112.1669160

[43] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *ISCA 44*.

[44] V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo, and J.M. Prellezo. 2001. The Adaptive Bubble Router. *J. Parallel Distrib. Comput.* 61, 9 (sep 2001), 1180–1208. https://doi.org/10.1006/jpdc.2001.1746

[45] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity.

arXiv:2104.12760 [cs.AR]

[46] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 347–358.

[47] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA 30*.

[48] Karu Sankaralingam, Tony Nowatzki, Greg Wright, Poly Palamuttam, Jitu Khare, Vinay Gangadhar, and Preyas Shah. 2021. Mozart: Designing for Software Maturity and the Next Paradigm for Chip Architectures. In *IEEE Hot Chips 33 Symposium, HCS 2021, Palo Alto, CA, USA, August 22-24, 2021*. IEEE, 1–20. https://doi.org/10.1109/HCS52781.2021.9567306

[49] Mahadev Satyanarayanan, Nathan Beckmann, Grace A. Lewis, and Brandon Lucia. 2021. The Role of Edge Offload for Hardware-Accelerated Mobile Devices. In *HotMobile*.

[50] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. 2000. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.* 49, 5 (2000), 465–481. https://doi.org/10.1109/12.859540

[51] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[52] Cheng Tan, Manupa Karunaratne, Tulika Mitra, and Li-Shiuan Peh. 2018. Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables. In *ISCA 45*.

[53] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. 2020. OpenCGRA: An open-source unified framework for modeling, testing, and evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 381–388.

[54] Frank Tavares. 2019. Kicksat 2. https://www.nasa.gov/ames/kicksat

[55] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-elastic cgras for irregular loop specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 412–425.

[56] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. 2021. Aurochs: An architecture for dataflow threads. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 402–415.

[57] Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. *ACM SIGARCH computer architecture news* 42, 3 (2014).

[58] Dani Voitsechov, Oron Port, and Yoav Etsion. 2018. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. In *MICRO 51*.

[59] Bo Wang, Manupa Karunarathne, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. 2019. Hycube: A 0.9 v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications. In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 133–136.

[60] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: synthesizing programmable spatial accelerators. In *ISCA 47*.

[61] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 703–716. https://doi.org/10.1109/HPCA47549.2020.00063

[62] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *HPCA*.

[63] Neil Weste and David Harris. 2011. *CMOS VLSI Design: A Circuits and Systems Perspective* (4th ed.). Addison-Wesley.

[64] Dhananjaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. 2021. HiMap: Fast and scalable high-quality mapping on CGRA via hierarchical abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[65] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961

[66] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

Received 29 April 2023; revised 9 July 2023; accepted July 24 2023