# RipTide

## A programmable, energy-minimal dataflow compiler and architecture

Graham Gobieski,[*] Souradip Ghosh,[*] Marijn Heule,[*] Todd Mowry,[*] Tony Nowatzki,[†] Nathan Beckmann,[*] Brandon Lucia[*]
[*] Carnegie Mellon University [†] University of California at Los Angeles
{gobieski, souradip}@cmu.edu tjn@cs.ucla.edu {mheule, tcm, beckmann}@cs.cmu.edu blucia@andrew.cmu.edu

***Abstract*—Emerging sensing applications create an unprecedented need for energy efficiency in programmable processors. To achieve useful multi-year deployments on a small battery or energy harvester, these applications must avoid off-device communication and instead process most data locally. Recent work has proven coarse-grained reconfigurable arrays (CGRAs) as a promising architecture for this domain. Unfortunately, nearly all prior CGRAs support only computations with simple control flow and no memory aliasing (e.g., affine inner loops), causing an Amdahl efficiency bottleneck as non-trivial fractions of programs must run on an inefficient von Neumann core.**

**RipTide is a co-designed compiler and CGRA architecture that achieves both high programmability and extreme energy efficiency, eliminating this bottleneck. RipTide provides a rich set of control-flow operators that support arbitrary control flow and memory access on the CGRA fabric. RipTide implements these primitives without tagged tokens to save energy; this requires careful ordering analysis in the compiler to guarantee correctness. RipTide further saves energy and area by offloading most control operations into its programmable on-chip network, where they can re-use existing network switches. RipTide's compiler is implemented in LLVM, and its hardware is synthesized in Intel 22FFL. RipTide compiles applications written in C while saving 25% energy v. the state-of-the-art energy-minimal CGRA and 6.6× energy v. a von Neumann core.**

***Keywords*-Energy-minimal, ultra-low-power, programmable, general-purpose, reconfigurable, CGRA, dataflow, compiler.**

## I. Introduction

Recent advances in machine learning, sensor devices, and embedded systems open the door to a wide range of sensing applications, such as civil-infrastructure or wilderness monitoring, public safety and security, medical devices, and chip-scale satellites [86]. To achieve long (e.g., 5+ year) deployment lifetimes, these applications rely on on-device processing to limit off-device communication. Computing at the extreme edge calls for ultra-low-power (<1 mW), *energy-minimal*, and *programmable* processing [29].

***Why programmable?*** The need for extreme energy efficiency suggests a role for application-specific integrated circuits (ASICs), but ASICs come with several major disadvantages. Computations in smart sensing applications are diverse, spanning deep learning, signal processing, compression, encoding, decryption, planning, control, and symbolic reasoning [28]. Only a programmable solution can support all of these, as it is infeasible to build an ASIC for every conceivable task [45, 78]. Moreover, the rapid pace of change in these applications (e.g., due to new machine learning algorithms [41]) puts specialized hardware at risk of premature obsolescence, especially in a multi-year deployment [78]. Finally, by targeting all computations, programmable designs can achieve much greater scale than specialized designs — perhaps trillions of devices [81]. Scale reduces device cost, makes advanced manufacturing nodes economically viable, and mitigates carbon footprint [34].

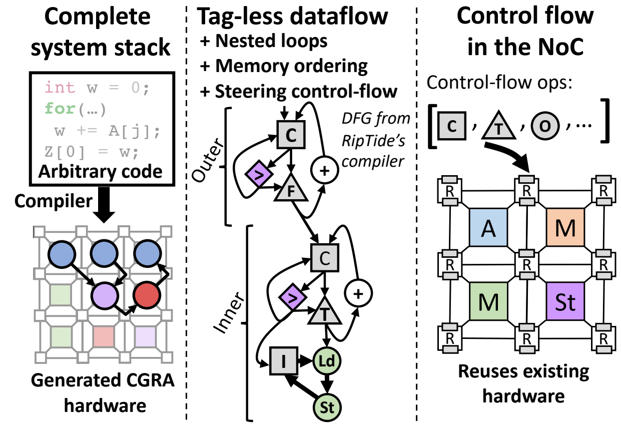


**Figure 1:** RipTide is a co-designed compiler and CGRA architecture that executes programs written in a *high-level language* with *minimal energy* and high performance. RipTide introduces new control-flow primitives to support common programming idioms, like deeply nested loops and irregular memory accesses, while minimizing the energy overhead. RipTide implements control flow *in the NoC* to increase utilization and ease compilation.

Unfortunately, traditional programmable cores are very inefficient, typically spending only 5% to 10% of their energy on useful work [27, 30, 39]. The architect's challenge is thus to reconcile generality and efficiency.

***CGRAs are both programmable and efficient.*** Recent work has shown that coarse-grained reconfigurable arrays (CGRAs) can achieve energy efficiency competitive with ASICs while remaining programmable by software [27, 66, 97]. As shown in Fig. 1, a CGRA is an array of processing elements (PEs) connected by an on-chip network (NoC). CGRAs are programmed by mapping a computation's dataflow onto the array, i.e., by assigning operations to PEs and configuring the NoC to route values between dependent operations. A CGRA's efficiency derives from avoiding overheads intrinsic to von

Neumann architectures, specifically instruction fetch/control and data buffering in a centralized register file.

In the context of ultra-low-power sensing applications, SNAFU [27] is a CGRA-generation framework designed from the ground up to minimize energy, in contrast to prior, performance-focused CGRAs (Sec. II). SNAFU CGRAs reduce energy by 5× v. ultra-low-power von Neumann cores, and they come within 3× of ASIC energy efficiency.

***What's the problem?*** Amdahl's Law tells us that to achieve significant end-to-end benefits, CGRAs must benefit the vast majority of program execution. CGRAs must support a wide variety of program patterns at minimal programmer effort, and they must provide a complete compiler and hardware stack that makes it easy to convert arbitrary application code to an efficient CGRA configuration. Unfortunately, prior CGRAs struggle to support common programming idioms efficiently, leaving significant energy savings on the table.

On the hardware side, many prior CGRAs support only simple, regular control flow, e.g., inner loops with streaming memory accesses and no data-dependent control [27, 64, 74]. To support complex control flow and maximize performance, other CGRAs employ expensive hardware mechanisms, e.g., associative tags to distinguish loop iterations, large buffers to avoid deadlock, and dynamic NoC routing [61, 70, 83, 93]. In either case, energy is wasted: from extra instructions needed to implement control flow unsupported by the CGRA fabric, or from inefficiency in the CGRA microarchitecture itself.

On the compiler side, mapping large computations onto a CGRA fabric is a perennial challenge. Heuristic compilation methods often fail to find a valid mapping [62, 73], and optimization-based methods lead to prohibitively long compilation times [13, 62]. Moreover, computations with irregular control flow are significantly more challenging to compile due to their large number of control operations, which significantly increase the size of the dataflow graph. To avoid these issues, SNAFU requires hand-coded vector assembly, restricting programs to primitives that map well onto a CGRA. Vector assembly sidesteps irregular control, but makes programming cumbersome [27, 64, 101].

## RIPTIDE'S APPROACH AND CONTRIBUTIONS

RipTide is a co-designed CGRA compiler and architecture that supports arbitrary control flow and memory access patterns without expensive hardware mechanisms. Unlike prior low-power CGRAs, ***RipTide can execute arbitrary code***, limited only by fabric size and routing. ***RipTide saves energy by offloading more code onto the CGRA***, where it executes with an order-of-magnitude less energy than a von Neumann core. In particular, RipTide supports deeply nested loops with data-dependent control flow and aliasing memory accesses, as commonly found in, e.g., sparse linear algebra. These benefits are realized via the following contributions:

***RipTide's instruction set architecture supports complex control while minimizing energy.*** RipTide adopts a *steering* control paradigm [11, 24, 83], in which values are only routed to where they are actually needed. To support arbitrary nested control without tags, RipTide introduces new control-flow primitives, such as the *carry gate*, which selects between tokens from inner and outer loops. RipTide also optimizes the common case by introducing operators for common programming idioms, such as its *stream generator* that generates an affine sequence for, e.g., streaming memory accesses.

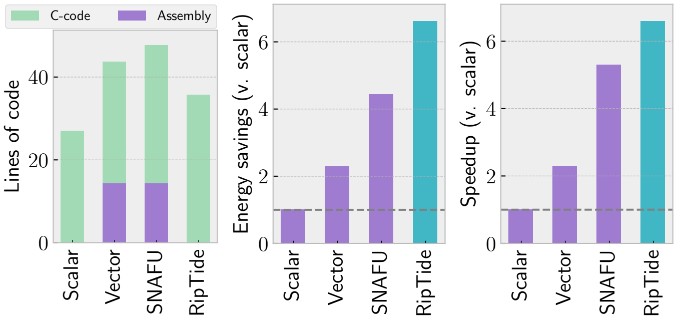


**Figure 2:** RipTide improves energy efficiency and performance on average across ten benchmarks over the state of the art, while compiling programs from high-level C (v. vector assembly in SNAFU).

***RipTide's (almost) free lunch: offloading control flow into the on-chip network.*** RipTide implements its new control flow primitives without wasting energy or PEs by leveraging existing NoC switches. The insight is that a NoC switch already contains essentially all of the logic needed for steering control flow, and, with a few trivial additions, it can implement a wide range of control primitives. Mapping control-flow into the NoC frees PEs for arithmetic and memory operations, so that RipTide can support deeply nested loops with complex control flow on a small CGRA fabric.

***RipTide compiles C programs to an efficient CGRA configuration.*** RipTide is easy to program: it compiles functions written in a high-level language (currently, C) and employs novel analyses to safely parallelize operations. We observe that, with steering control flow and no program counter, conventional transitive reduction analysis fails to enforce all memory orderings, so we introduce *path-sensitive transitive reduction* to infer orderings correctly. RipTide implements arbitrary control flow without associative tags by enforcing strict ordering among values, leveraging its new control operators. RipTide maps programs onto the CGRA by formulating place-and-route as a SAT instance or integer linear program. The SAT formulation finds configurations quickly ($< 3$ min), while the ILP formulation yields configurations that use 4.3% less energy.

***Summary of results.*** We implement a complete RipTide system in RTL and synthesize it in Intel 22FFL, an industrial sub-28nm FinFET process with compiled memories. Including core and memory, RipTide's area is just $\approx 0.5\text{mm}^2$. Across ten benchmarks, ranging from linear algebra to graph search, RipTide reduces energy by 25% v. SNAFU, the state-of-the-art energy-minimal design, and improves performance by 17% (Fig. 2). At nominal voltage with random inputs, RipTide achieves 180 MOPS/mW (including main memory) on `dmm`.

RipTide consumes just 2.4× more energy than equivalent ASICs for dmm, sort, and fft, and RipTide achieves these benefits on software written in C.

We identify several methodological challenges in measuring CGRA efficiency. The choice of metric can skew reported efficiency by more than 10× — e.g., RipTide achieves 1970 fabric MIPS/mW on dmm, which is often reported as MOPS/mW in prior work. Surveying prior work, we find that RipTide is 2.4× more energy-efficient than prior, performance-oriented CGRAs with comparable data [89, 96].

Finally, we compare RipTide to off-the-shelf ULP systems on end-to-end neural network inference. RipTide consumes 1900× less energy than an TI MSP430, 490× less energy than an ARM Cortex-M3, and 6.5× less energy than our own energy-minimal scalar core — and RipTide is 7.7× faster.

***Broader implications on architecture.*** We perform an in-depth case study of dmm, comparing RipTide to an ASIC implemented in the same design flow. RipTide is competitive on energy and performance, but consumes significantly more area than the ASIC. ASICs thus offer an area advantage over CGRAs, but this advantage disappears in SoC designs with a large number of ASIC blocks. Given the large advantages gained by software programmability, we argue that energy-minimal CGRAs like RipTide have a compelling edge over ASICs for the majority of computations.

***Road map.*** Sec. II covers background, and Sec. III gives an overview of RipTide. Secs. IV, V, and VI present RipTide's architecture, compiler, and microarchitecture, respectively. Secs. VII and VIII evaluate RipTide, and Sec. IX concludes by discussing RipTide's broader implications.

## II. Background

RipTide is motivated by emerging energy-constrained sensing applications. CGRAs avoid inefficiencies of von Neumann cores, but prior CGRAs have limited programmability or efficiency. This section motivates RipTide's contributions in the context of prior work.

### *A. Context: Computing at the extreme edge*

Long-lived, sensor-based applications (e.g., wilderness monitoring, public safety, tiny satellites) [23, 52] run on batteries or harvested energy and are often inaccessible once deployed. These energy sources impose a tight design constraint; e.g., an amortized power budget of 65 µW with a AA battery over five years. Given the high energy cost of off-device communication, sensor devices should process data *locally* to capitalize on their limited energy [29].

On-device compute efficiency thus has a significant impact on device value, but unfortunately existing architectures fail to meet the needs of these applications. Application-specific integrated circuits (ASICs) offer high compute efficiency, but are too inflexible for most long-lived, energy-constrained applications because they cannot adapt as applications change [79]. Moreover, ASIC efficiency comes at a high, upfront cost in design, verification, and manufacturing, limiting their applicability to a few stable workloads.

Alternatively, programmable cores offer flexibility, but burn most energy (upwards of 90%) on instruction fetch, pipeline control, and register-file access. Luckily, this inefficiency is not fundamental. Real programs have abundant instruction and data locality that von Neumann cores exploit poorly, as instructions share an execution pipeline and communicate through a register file. Exploiting this locality is the key to reconciling efficiency and programmability.

### *B. CGRAs can dramatically improve efficiency*

CGRA architectures [7, 14, 15, 19, 26, 31, 32, 42, 53, 56–60, 64, 71, 72, 74–77, 80, 84, 85, 89, 93, 94, 97, 98] are designed with this locality in mind and can reduce energy v. a von Neumann core by a large factor. The key to CGRA efficiency is *spatial distribution* of compute and communication. Rather than time-multiplex all instructions on a shared pipeline, which significantly increases switching activity [27], CGRAs spatially distribute instructions across processing elements (PEs). Rather than send values through a register file, PEs communicate them via on-chip network (NoC).

***CGRA applicability limits efficiency.*** Hence, to minimize energy, one would ideally run an entire program on the CGRA. Two considerations prevent this. First, any unsupported computations (e.g., outer loops, irregular memory access) must run on the core, creating an Amdahl bottleneck on end-to-end efficiency. Second, large computations cannot fit on CGRAs and must be broken into smaller ones, with intermediate values spilled to scratchpads or main memory.

### *C. Limitations of prior CGRAs*

Prior CGRAs mostly target loop nests with easily analyzable, regular control; see the examples in Table I. Especially in the ultra-low-power (ULP) domain, most CGRAs are "systolic," i.e., statically scheduled with fixed operation latencies.

***Why only simple control?*** Like nearly all processors, prior CGRAs primarily *maximize performance* under an area or power budget. Key CGRA metrics have been PE utilization and initiation interval (i.e., cycles between the start of consecutive loop iterations). For this reason, it does not make sense for most CGRAs to support outer loops with low utilization or irregular loops and memory accesses that require expensive hardware to maintain performance. Resources are better spent on unrolling regular inner loops to improve performance.

Revel [98] and ultra-elastic CGRAs [89] stand out from prior work in recognizing and partially addressing these limitations (Table I). Revel [98] supports outer loops with a hybrid architecture that maps tight inner loops to a systolic array and outer loops to a tagged-token dataflow fabric. Ultra-elastic CGRAs [89] accelerate singly nested irregular loops through a ratiochronous clocking scheme. Despite this added support, these designs are still limited to a subset of common program idioms and target a different, performance-oriented domain than RipTide.

**Table I:** Qualitative comparison of RipTide to prior work. RipTide's goal is to *minimize energy* by executing *entire functions* on the CGRA fabric. To achieve this, RipTide supports arbitrary control flow and irregular memory access, whereas prior CGRAs are limited to affine loops or a subset of common program idioms. RipTide compiles lightly annotated C to an efficient CGRA configuration.

| | **Revel [98]** | **UE-CGRA [89]** | **Wavescalar [83]** | **SNAFU [27]** | **RipTide** |
|---|---|---|---|---|---|
| **Goal** | ⏱ Performance / Area | ⏱ Performance or ⚡Energy | ⏱ Performance | ⚡ Energy | ⚡ Energy |
| **Power** | 100s mW | 10s mW | 1000s mW | < 1 mW | < 1 mW |
| **Target** | Imperfectly nested loops | Irregular inner loops | Arbitrary programs | Affine inner loops | Arbitrary functions |
| **Code changes** | Loop pragmas | None | None | Vector assembly | Function annotation |
| **Exemplar program** | `void foo (...) {`<br>`  #pragma config ...`<br>`  for (i = 0..n) ...`<br>`    #pragma stream`<br>`    #pragma dataflow`<br>`    for (j = i..n)`<br>`      ... = a[i][j]`<br>`}` | `void foo (...) {`<br>`  ...`<br>`  for (i = 0..n) {`<br>`    while (a[i] != 0)`<br>`      a[i] = ...`<br>`  ...`<br>`  }`<br>`}` | `void foo (int * a,`<br>`          int * b) {`<br>`  while (!q.empty()) {`<br>`    n = q.pop()`<br>`    for (i in 0..n)`<br>`      if (b[a[i]]) ...`<br>`  }`<br>`}` | `void foo (...) {`<br>`  for (i = 0..n) ...`<br>`    vlh v1, a + i`<br>`    vlh v2, b`<br>`    vadd v3, v1, v2`<br>`    vsh b + i, v3`<br>`  ...`<br>`}` | `#riptide void foo`<br>`  (int * restrict a, b) {`<br>`  while (!q.empty()) {`<br>`    n = q.pop()`<br>`    for (i in 0..n)`<br>`      if (b[a[i]]) ...`<br>`  }`<br>`}` |

***Relevance to RipTide: Energy is our goal, not performance.*** RipTide is designed to *minimize energy*. Many choices in RipTide only make sense in this context. Extreme efficiency requires offloading as much of programs as possible onto the CGRA. Tricks like loop unrolling do not save energy; in fact, they can *cost* energy by running more instructions on the core (e.g., for outer loops or setup).

Hence, *energy-minimal CGRAs need to support arbitrary control flow and memory access patterns,* but they must do so while maintaining high energy efficiency. This means they cannot add expensive microarchitectural mechanisms or significantly increase program size. *RipTide reconciles these conflicting goals by offloading most control operations to its NoC,* where they reuse existing circuitry, do not consume scarce PE resources, and avoid expensive buffering.

SNAFU [27] is a recent framework for generating energy-minimal CGRAs. SNAFU adopts many microarchitectural techniques to save energy, but requires vector assembly and is thus limited to simple control flow (Table I). RipTide targets the same domain and uses SNAFU as its baseline design.

### *D. Dynamic dataflow architectures*

Closely related to CGRAs are classic dynamic dataflow architectures. These designs also express programs as a dataflow graph of dependent operations, but, unlike CGRAs, are designed as a *replacement* for cores, not a co-processor. They thus support arbitrary, complex control and, to support large programs, they store the dataflow graph in memory and execute it on a shared execution pipeline [24, 61, 70, 83]. In this respect, they resemble von Neumann cores, and unfortunately lose much of the energy benefits of CGRAs' spatial distribution.

Some dataflow architectures have combined spatial and temporal execution [57, 71, 76, 83, 93–95] (see, e.g., Wavescalar in Table I). These performance-oriented designs require expensive microarchitectural mechanisms to maintain performance with highly variable memory latency (e.g., associative tag matching due to operand re-ordering, or large on-chip buffers to hide latency with memory-level parallelism).

***Relevance to RipTide: Balancing generality and efficiency.*** RipTide also targets arbitrary code written in high-level languages. But to maintain energy efficiency, RipTide spatially distributes instructions and avoids expensive microarchitectural mechanisms. In particular, RipTide employs *ordered-dataflow* scheduling, which tolerates variable latency and also avoids tag matching by disallowing out-of-order execution so that inputs are always matched on arrival. Ordered dataflow potentially loses performance v. out-of-order execution, but, again, RipTide focuses on energy. (Regardless, the performance loss is small at ULP scale because main memory fits in a single cycle.)

Moreover, like other dataflow architectures [10, 24, 33, 57, 65, 83], RipTide adopts a *steering* control paradigm ($\phi^{-1}$), where values are routed only to dependent operations. Steering minimizes energy because values are only sent where they are actually needed, unlike predication or selection ($\phi$) control used in some CGRAs [27, 76]. However, since loop iterations may take different paths through control flow, steering risks token re-ordering, which would lead to incorrect results in RipTide's ordered-dataflow model. RipTide's compiler guarantees correct ordering by inserting ordering operations where necessary.

### *E. Dimensions of CGRA architecture*

There are many dimensions of CGRA architecture that affect energy efficiency, throughput, and programmability. RipTide is carefully designed along each dimension to reduce energy and maximize programmability.

***Programming interface.*** CGRAs can be programmed in many different ways. Some [20, 27] expose low-level interfaces that require expert knowledge. Others [74] develop domain-specific languages [82] that simplify compilation and the architecture, but also limit the scope of supported applications. Still others [83] take a more general-purpose approach, compiling from languages originally developed for CPUs. RipTide takes this approach to maximize programmability. We choose to compile from C because C is popular in embedded applications.

***PE type(s).*** CGRAs offer a wide range of design choices, including PE operation set, PE complexity, and NoC. A CGRA's PEs typically support arithmetic, logic, memory accesses, or more specialized functionality [17–19, 27, 75, 92, 97, 100]. PEs may be homogeneous or heterogeneous; the latter is more area- and energy-efficient, but creates a combinatorially large design space [7]. RipTide is a heterogeneous design, with PEs specialized for arithmetic and memory. One could view RipTide's programmable NoC routers as specialized PEs that support only control operations.

***Mapping is hard.*** Like hardware synthesis, a CGRA compiler must find a layout of operations that fits within fabric resources with valid routes between all producers and consumers. In a performance-focused CGRA, the compiler must also reason about timing to maximize utilization and minimize initiation interval [5, 6, 22, 36, 43, 44, 50, 51, 69, 99, 102]. This analysis is further complicated by control flow (e.g., branches) and operations with variable latencies (e.g., loads). With such a large search space, optimization-based methods often do not converge in a reasonable time [63, 67] and heuristic approaches can yield poor results.

RipTide's focus on energy also helps with mapping. Non-heuristic methods like SAT and integer linear programming (ILP) are feasible in RipTide because its compiler need not reason about timing or utilization, greatly simplifying mapping. RipTide can also offload control-flow operations to its NoC, freeing up scarce PEs for the mapper to use for other operations.

***CGRA memory-ordering model.*** A CGRA design must ensure correct memory-operation ordering. Some CGRAs and dataflow designs enforce total memory ordering or intra-thread ordering [83, 93], and work on optimized ordering has been limited to small, acyclic DFGs with help from hardware disambiguation [90]. RipTide's compiler uses a new path-sensitive ordering graph reduction analysis that reduces ordering overheads for arbitrary cyclic DFGs and ensures correctness in RipTide's execution model.

## III. RipTide Overview

RipTide is a compiler and microarchitecture for ultra-low-power, energy-minimal CGRAs. At its core is an ISA (Sec. IV) that supports arbitrary control flow without expensive associative tag matching. The compiler (Sec. V) transforms programs written in high-level C into dataflow graphs using this ISA. It also enforces memory ordering and optimizes programs by fusing loop induction variables into single stream operators. RipTide's CGRA fabric efficiently executes compiled programs (Sec. VI). It minimizes switching activity by assigning a single operation per PE. Additionally, it reuses hardware in the NoC to implement control-flow operations without wasting PE resources. RipTide improves energy efficiency / performance by 25% / 17% v. prior energy-minimal CGRAs [27] and by 6.6× / 6.2× v. a ULP scalar core.

## IV. RipTide Instruction Set Architecture

RipTide provides a rich set of control-flow operators to support complex programs. Its ISA, shown in Table II, has six categories of operators: arithmetic, multiplier, memory, control flow, synchronization, and streams. (Multiplication is split from other arithmetic because, to save area, only some PEs can perform multiplication.) We now highlight the control-flow, synchronization, and stream operators.

**Table II:** RipTide's instruction set architecture (ISA).

| Operator(s) | Category | Symbol(s) | Semantics |
|---|---|---|---|
| Basic binary ops | Arithmetic | $+, -, <<, != $, etc. | *a* op *b* |
| Multiply, clip | Multiplier | $*$, `clip` | *a* op *b* |
| Load | Memory | `ld` | `ld` *base*, *idx*(, *dep*) |
| Store | Memory | `st` | `st` *base*, *idx*, *val*(, *dep*) |
| Select | Control Flow | `sel` | *cond* ? *val*0 : *val*1 |
| Steer, carry, invariant | Control Flow | (`T` \| `F`), `C`, `I` | See Fig. 3 |
| Merge, order | Synchronization | `M`, `O` | See Fig. 3 |
| Stream | Stream | `STR` | See Fig. 3 |

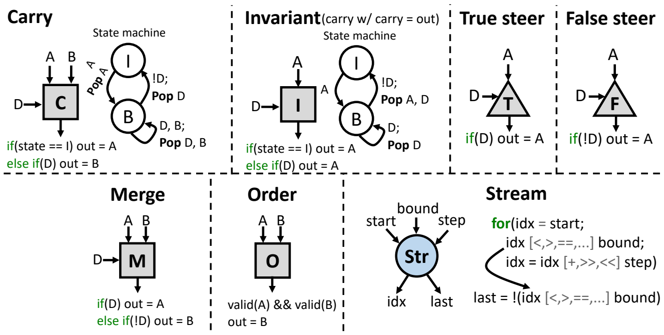


**Figure 3:** Semantics of control-flow operators in RipTide.

### A. Control-flow operators

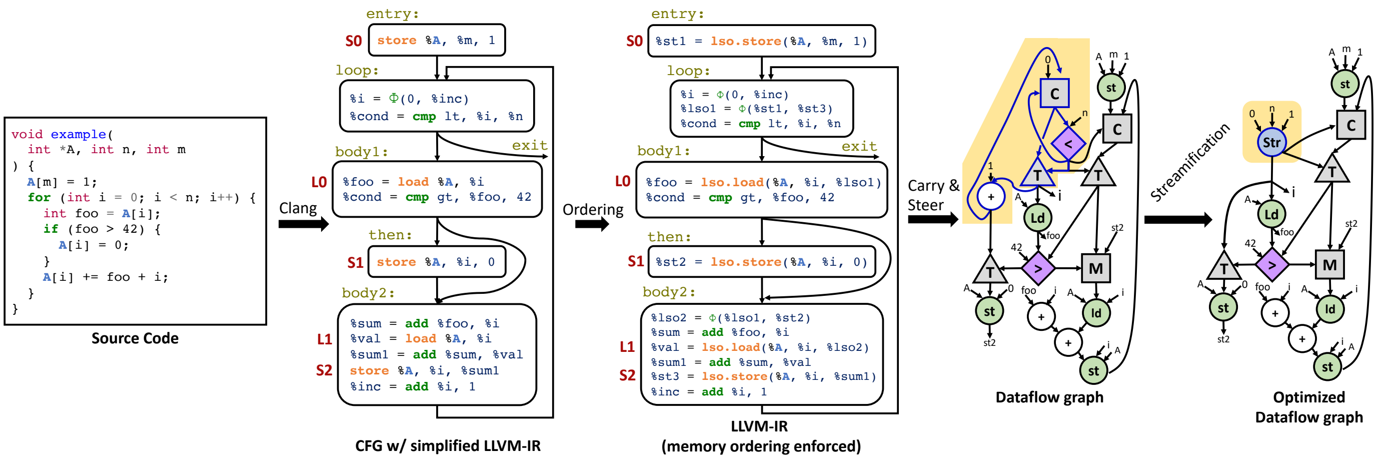
RipTide's operators are illustrated in Fig. 3. Whenever a value is read, it is implied that the operator waits until a valid token arrives for that value over the NoC. Tokens are buffered at the inputs if they are not consumed or discarded.

***Steer.*** Steers ($\phi^{-1}$) come in two flavors — True and False — and take two inputs: a decider, `D` and a data input, `A`. If `D` matches the flavor, then the gate passes `A` through; otherwise, `A` is discarded. Steers are necessary to implement conditional execution, as they gate the inputs to disabled branches.

***Carry.*** Carry represents a loop-carried dependency and takes a decider, `D`, and two data values, `A` and `B`. Carry has the internal state machine shown in Fig. 3. In the *Initial* state, it waits for `A`, and then passes it through and transitions to the *Block* state. While in *Block*, if `D` is True, the operator passes through `B`. It transitions back to *Initial* when `D` is False, and begins waiting for the next `A` value (if not already buffered at the input).

Carry operators keep tokens ordered in loops, eliminating the need to tag tokens. All backedges are routed through a carry operator in RipTide. By not consuming `A` while in *Block*, carry operators prevent outer loops from spawning a new inner-loop instance before the previous one has finished. (Iterations from one inner-loop may be pipelined if they are independent, but entire instances of the inner loop will be serialized.)

***Invariant.*** The invariant operator is a slight variation of carry. It represents a loop invariant and can be implemented as a carry with a self-edge back to `B`. Invariants are used to generate a new loop-invariant token for each loop iteration.



Source Code

```
void example(
  int *A, int n, int m
) {
  A[m] = 1;
  for (int i = 0; i < n; i++) {
    int foo = A[i];
    if (foo > 42) {
      A[i] = 0;
    }
    A[i] += foo + i;
  }
}
```

CFG w/ simplified LLVM-IR

```
entry:
S0  store %A, %m, 1
loop:
    %i = Φ(0, %inc)
    %cond = cmp lt, %i, %n
body1:                                   exit
L0  %foo = load %A, %i
    %cond = cmp gt, %foo, 42
then:
S1  store %A, %i, 0
body2:
    %sum = add %foo, %i
L1  %val = load %A, %i
    %sum1 = add %sum, %val
S2  store %A, %i, %sum1
    %inc = add %i, 1
```

LLVM-IR
(memory ordering enforced)

```
entry:
S0  %st1 = lso.store(%A, %m, 1)
loop:
    %i = Φ(0, %inc)
    %lso1 = Φ(%st1, %st3)
    %cond = cmp lt, %i, %n
body1:                                   exit
L0  %foo = lso.load(%A, %i, %lso1)
    %cond = cmp gt, %foo, 42
then:
S1  %st2 = lso.store(%A, %i, 0)
body2:
    %lso2 = Φ(%lso1, %st2)
    %sum = add %foo, %i
L1  %val = lso.load(%A, %i, %lso2)
    %sum1 = add %sum, %val
S2  %st3 = lso.store(%A, %i, %sum1)
    %inc = add %i, 1
```

**Figure 4:** RipTide's frontend and middle-end components. The frontend compiles C code to LLVM-IR using clang. The middle-end produces an optimized dataflow graph (DFG) that enforces memory ordering and RipTide's control paradigm.

### B. *Synchronization operators*

***Merge.*** The merge operator enforces cross-iteration ordering by making sure that tokens from different loop iterations appear in the same order, regardless of the control path taken within by each loop iteration. The operator takes three inputs: a decider, D, and two data inputs, A and B. Merge is essentially a mux that passes through either A or B, depending on D. But note that only the value passed through is consumed.

***Order.*** The order operator is used to enforce memory ordering by guaranteeing that multiple preceding operations have executed. It takes two inputs, A and B, and fires as soon as both arrive, passing B through.

### C. *Stream operators*

Streams generate a sequence of data values, which are produced by evaluating an affine function across a range of inputs. These operators are used in loops governed by affine induction variables. A stream takes three inputs: start, step, and bound. It initially sets its internal idx to start, and then begins iterating a specified arithmetic operator f as idx' = f(idx, step).

A stream operator produces two output tokens per iteration: idx itself, and a control signal last. last is False until idx reaches bound, whereupon it is True and the stream stops iterating. last is used by downstream control logic to, e.g., control a carry operator for outer loops.

## V. RipTide Compiler

RipTide compiles, optimizes, and maps high-level C code to RipTide's CGRA fabric. Its compiler has a frontend, middle-end, and backend. The frontend uses clang to compile C to LLVM's intermediate representation (IR). The middle-end manipulates the LLVM IR to insert control-flow operators from Sec. IV and enforce memory ordering; then it translates the IR to a dataflow graph (DFG) representation and optimizes the DFG by transforming and fusing subgraphs, reducing operator count by 27% (Fig. 4). The backend takes the DFG as input and maps operators onto the CGRA, producing a configuration bitstream in minutes.

### A. *Memory-ordering analysis*

RipTide maps sequential code onto a CGRA fabric in which many operations, including memory, may execute in parallel. For correctness, some memory operations must execute in a particular order. RipTide's middle-end computes required orderings between memory operations present in the IR and adds control-flow operations to enforce those orderings.

***Constructing a memory-operation ordering graph.*** The first step to enforcing memory ordering is to construct an ordering graph (OG) that encodes dependences between memory operations. RipTide uses alias analysis to identify memory operations that may or must access the same memory location (i.e., alias), adding an arc between the operations in the OG accordingly. RipTide makes no assumptions on the alias analysis and need not consider self-dependences because repeated instances of the same memory operation are always ordered on its CGRA fabric. Fig. 5 shows a basic, unoptimized OG in the top left for an example function.

***Pruning the ordering graph.*** The OG as computed can be greatly simplified. Prior work has simplified the OG with improved alias analysis [38] and by leveraging new control-flow primitives [12,54]. These efforts are orthogonal to RipTide. RipTide simplifies the OG by eliminating redundant ordering arcs that are already enforced by data and control dependences. RipTide finds data dependences by walking LLVM's definition-use (def-use) chain from source to destination and removes *ordering* arcs for dependent operations [90]. For instance, in example's CFG from Fig. 4, **S2** is data-dependent on **L1**, so there need not be an ordering arc in the OG. This is reflected in the blue-outlined arc from **L1** to **S2** that is pruned in the OG in Fig. 5. Similarly, control dependences order some memory operations if the execution of the destination is control-dependent on the source. RipTide analyzes the CFG to identify control dependences between memory operations and removes

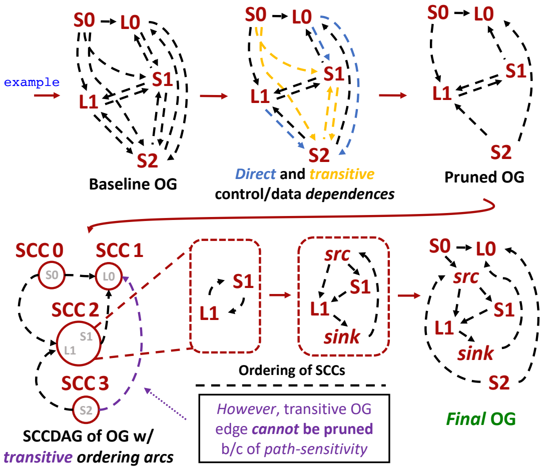


**Figure 5:** RipTide's middle-end enforces memory ordering. For `example`, an ordering graph (OG) that is iteratively pruned and reduced.

those orderings from the OG. In `example`'s CFG from Fig. 4, the arc from **L0** to **S1** in Fig. 5 is pruned using this analysis.

***Transitive memory-ordering analysis.*** Two dependent memory operations are transitively ordered if there is a path (of ordering arcs) in the OG from source to destination. RipTide finds and eliminates redundant arcs that are transitively ordered by other control- and data-dependence orderings. This reduces the number of operations required to enforce ordering by 18% v. unoptimized ordering.

To simplify its OG, RipTide uses transitive reduction (TR) [3], which prior work deployed to simplify ordering relation graphs for parallel execution of loops [54, 55]. We apply TR to the OG, which converts a (potentially cyclic) ordering graph into an acyclic graph of strongly connected components (the SCCDAG). Traditional TR eliminates arcs between SCCs, removes all arcs within each SCC, and adds arcs to each SCC to form a simple cycle through all vertices.

We modify the algorithm in two ways to make it work for RipTide's OG. First, arcs in the inserted cycle must be compatible with program order instead of being arbitrary. Second, the inserted arcs must respect proper loop nesting, avoiding arcs directly from the inner to outer loop. To handle these arcs, we add synthetic loop entry and exit nodes to each loop (shown as `src` and `sink` nodes at the bottom of Fig. 5). Any arc inserted that links an inner loop node to an outer loop node instead uses the inner loop's exit as its destination. Symmetrically, an arc inserted that links an outer loop node to an inner loop node has the inner loop's entry as its destination. With these two changes, the SCCDAG is usable for TR.

However, we observe that applying existing TR analysis to the OG in RipTide fails to preserve required ordering operations. The problem is that a source and destination may be ordered along one (transitive) path, and ordering along another (direct) path may be removed as redundant. Execution along the transitive path enforces ordering, but along the direct path does not, which is incorrect. Fig. 5 shows a scenario where path-sensitivity is critical. The path, **SCC3**(**S2**)→**SCC1**(**L0**), should not be eliminated in TR because the alternative path, **SCC3**(**S2**)→**SCC2**(**L1**)→**SCC1**(**L0**), does not capture the direct control-flow path from **S2** to **L0** via the backedge of the loop. This problem arises due to RipTide's steering control and lack of a program counter to order memory operations.

To correctly apply TR to remove redundant ordering arcs, RipTide introduces *path-sensitive* TR, which confirms that a transitive ordering path subsumes all possible control-flow paths before removing any ordering arc from the OG. With this constraint in place, RipTide can safely use transitive reduction.

***Enforcing ordering constraints.*** Memory operators in RipTide produce a control token on completion and can optionally consume a control token (*dep* in Table II) to enforce memory ordering. The middle-end encodes ordering arcs as defs and uses of data values in the IR (as seen in the IR transform of loads and stores in Fig. 4) before lowering them as dependences in the DFG. For a memory operator that must receive multiple control signals, the middle-end inserts order operations (Sec. IV) to consolidate those signals.

### *B. Control-flow operator insertion*

The compiler lowers its IR to use RipTide's control paradigm by inserting RipTide control-flow operators into the DFG.

***Steer.*** The compiler uses the control dependence graph (CDG) [16] to insert steers. For each consumer of a value, the compiler walks the CDG from the producer to the consumer and inserts a steer operator at each node along the CDG traversal if it has not already been inserted by a different traversal. The steer's control input is the decider of the basic block that the steer depends on, and its data input is the value or the output of an earlier inserted steer.

***Carry and invariant.*** For loops, the compiler inserts a carry operator for loop-carried dependences and an invariant operator for loop-invariant values into the loop header. A carry's data input comes from the loop backedge that produces the value. An invariant's data input comes from values defined outside the loop. These operators should produce a token only if the next iteration of the loop is certain to execute; to ensure this behavior, the compiler sets their control signal to the decider of the block at the loop exit.

***Merge.*** If two iterations of a loop may take different control-flow paths that converge at a single join point in the loop body, either may produce a token to the join point first. But for correctness, the one from the earlier iteration must produce the first token. The compiler inserts a merge operator at a join point in the CFG to ensure that tokens flow to the join point in iteration order. The control signal `D` for the merge operator is the decider of nearest common dominator of the join point's predecessor basic blocks. Since the earlier iteration sends its control signal first and RipTide does not reorder tokens, the merge operator effectively blocks the later iteration until the earlier iteration resolves.

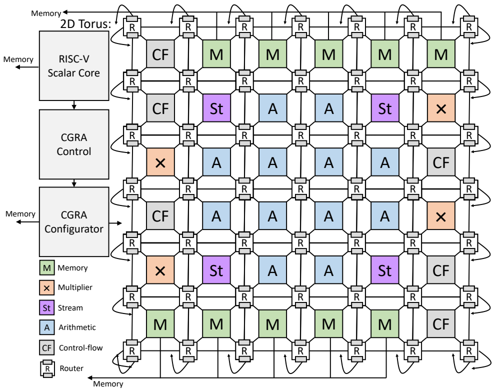


Figure 6: RipTide's ULP CGRA fabric.

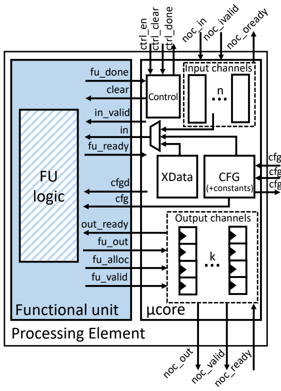


Figure 7: PE microarchitecture

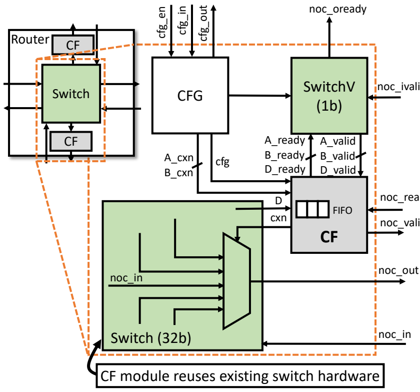


Figure 8: Router microarchitecture

### C. Stream fusion

RipTide performs target-specific operator fusion on the DFG to reduce required operations and routes by combining value *stream generators* with loop control logic and address computation logic. RipTide supports streams and applies them for the common case of a loop with an affine loop governing induction variable (LGIV). A stream makes loop logic efficient by fusing the LGIV update and the loop exit condition into a single operator. In the DFG, loop iteration logic is represented by the exit condition, an update operator, the carry for the LGIV's value, and the steer that gates the LGIV in a loop iteration. The middle-end fuses these operators into a single stream operator and sets the stream's initial, step, and bound values. Fig. 4 shows stream compilation, where the operators for loop iteration logic (outlined in blue in the DFG) are fused into a stream operator. RipTide applies induction-variable analysis [2, 4] to find affine LGIVs. RipTide also identifies address computations, maps these to an affine stream if possible, and fuses the stream into the memory operator.

### D. Mapping DFGs to hardware

RipTide's backend takes a DFG and a CGRA topology description and generates scalar code to invoke RipTide and a bitstream to configure the RipTide fabric. This involves finding a mapping of DFG nodes and edges to PEs, control-flow modules (Sec. VI-D), and links. Mapping can be difficult, and there is much prior work on heuristic methods that trade mapping quality for compilation speed [6, 36, 37, 43, 44, 50, 51, 99, 102]. RipTide has two advantages v. this prior work. First, RipTide does not time-multiplex operations, so it only needs to schedule operations in space, not time. Prior compilers unroll loops to reason about operation timing and identify the initiation interval, increasing program size. Second, RipTide targets energy efficiency, not performance. Rather than optimize for initiation interval, it need only focus on finding a valid solution, since leakage is insignificant.

RipTide provides two complementary mappers: one based on boolean satisfiability (SAT) and another based on integer linear programming (ILP) that minimizes the average routing distance. The SAT-based mapper runs quickly, taking $<3$ min for our most complex benchmark, whereas the ILP-based mapper yields 4.3% avg. energy savings v. SAT (Sec. VIII-C).

***Problem description.*** The constraints of the ILP and SAT formulations are similar (see Appendix A for a complete, formal description). The formulations ensure that every DFG vertex is mapped to a hardware node, that every edge is mapped to a continuous route of hardware links, and that the inputs and outputs of a vertex match the incoming and outgoing links of a hardware node. They further disallow the mapping of multiple DFG vertices to a single hardware node, the sharing of hardware links by multiple edges with different source vertices, and the mapping of a DFG edge through a control-flow module when a DFG vertex is mapped to that module. Together these are the necessary constraints to produce not only a valid mapping, but also a good mapping (SAT is close to ILP in terms of energy).

## VI. RipTide Microarchitecture

RipTide is an energy-minimal coarse-grained reconfigurable array (Fig. 6). The 6×6 fabric contains heterogeneous PEs connected via a bufferless, 2D-torus NoC. A complete RipTide system contains a CGRA fabric, a RISC-V scalar core, and a 256KB (8×32KB banks) SRAM main memory.

### A. Tagless dataflow scheduling

RipTide implements asynchronous dataflow firing via *ordered dataflow* (Sec. II). By adding ordering operators where control may diverge, RipTide ensures that tokens always match on arrival at a PE, obviating the need for tags. Tagless, asynchronous firing has a low hardware cost (one bit per input plus control logic), and it lets RipTide tolerate variable operation latency (e.g., due to bank conflicts) while eliminating the need for the compiler to reason about operation timing.

### B. Processing elements

RipTide's PEs perform all arithmetic and memory operations in the fabric. Fig. 7 shows the microarchitecture of a PE. The PE includes a functional unit (FU) and the µcore. The µcore interfaces with the NoC, buffers output values, and interfaces with top-level fabric control for PE configuration.

***Functional units.*** The µcore exposes a generic interface using a latency-insensitive ready/valid protocol to make it easy to add new operators. Inputs arrive on `in_data` when `in_valid` is high, and are consumed when `fu_ready` is high. The FU reserves space in the output channel by raising `fu_alloc` (e.g., for pipelined, multi-cycle operations), and output arrives on `fu_data` when `fu_valid` is high. `out_ready` supplies back pressure from downstream PEs. The remaining signals deal with top-level configuration and control.

***Communication.*** The µcore decouples NoC communication from FU computation. The µcore tracks which inputs are valid, raises backpressure on input ports when its FU is not ready, buffers intermediate results in output channels, and sends results over the NoC. Decoupling simplifies the FU.

***Configuration.*** The µcore handles PE and FU configuration, storing configuration state in a two-entry *configuration cache* that enables single-cycle reconfiguration. Additionally, the µcore enables the fabric to overlap reconfiguration of some PEs while others finish computation on an old configuration.

***PE types.*** RipTide includes a heterogeneous set of PEs:

- *Memory PEs* issue loads and stores to memory and have a "row buffer" that coalesces non-aliasing subword loads.
- *Arithmetic PEs* implement basic ALU operations, e.g., compare, bitwise logic, add, subtract, shift, etc.
- *Multiplier PEs* implement multiply, multiply + shift, multiply + fixed-point clip, and multiply + accumulate.
- *Control-flow PEs* implement steer, invariant, carry, merge, and order (Sec. IV) — but most of these are actually implemented in RipTide's NoC (see below).
- *Stream PEs* implement common affine iterators (Sec. IV).

### C. Bufferless NoC

RipTide connects PEs via a statically configured, multi-hop, bufferless on-chip network with routers. Instead of buffering values in the NoC, PEs buffer values in their output channel. NoC buffers are a primary energy sink in prior CGRAs [27,42], and RipTide completely eliminates them. Similarly, RipTide's NoC is statically routed to eliminate routing look-up tables and flow-control mechanisms.

### D. Control flow in the NoC

Control-flow operators are simple to implement (often a single multiplexer), but there are many of them. Mapping each to a PE wastes energy and area, and can make mapping to the CGRA infeasible. Among our ten benchmarks, 46% of operations are control flow, and eight benchmarks do not map if each control-flow operator requires a dedicated PE.

We observe that much of the logic required to implement control flow is already plentiful in the NoC. Each NoC switch is a crossbar that can be re-purposed to mux values for control. Thus, to implement each control-flow operator, RipTide manipulates a switch's routing and ready/valid signals to provide the desired functionality.

RipTide's router microarchitecture is shown in Fig. 8. The router shares routing configuration and its data and valid crossbars with the baseline NoC. RipTide adds a control-flow module (CFM) at a configurable number of output ports (in our case, two output ports). The CFM determines when to send data to the output port and manipulates inputs to the data switch to select which data is sent.

```
cxn = A_cxn
forever:
    A_ready = D_ready = 0
    if A_valid && D_valid: # wait for A and D
        # if D is true, pass through A;
        # else discard A
        noc_valid = D
        A_ready = D_ready = noc_ready || !D
        if D: wait for noc_ready
```

*(a) Steer (True flavor).*

```
forever:
    # begin in Initial state
    if A_valid:
        cxn = A_cxn         # pass through A
        noc_valid = A_valid
        D_ready = A_ready = noc_ready
        B_ready = xxx       # don't care
    wait for noc_ready
    # transition to Block state
    do until D_valid && !D:
        cxn = B_cxn         # pass through B
        noc_valid = B_valid
        D_ready = B_ready = noc_ready
        A_ready = false     # hold A at input
        wait for noc_ready
```

*(b) Carry.*

**Figure 9:** Implementing control flow using NoC control signals.

***Control-flow module.*** The CFM takes eight inputs and produces five outputs that control router configuration and dataflow through the network. The inputs are:

- `cfg`: configuration of the CFM (i.e., opcode);
- `A_valid`, `B_valid`, `D_valid`: whether inputs are valid;
- `D`: value of the decider;
- `A_cxn` and `B_cxn`: input ports for `A` and `B`; and
- `noc_ready`: backpressure signal from the output port.

From this, the CFM produces outputs:

- `A_ready`, `B_ready`, and `D_ready`: upstream backpressure signals that allow the CFM to block upstream producers until all signals required are valid;
- `noc_valid`: the valid signal for the CF's output; and
- `cxn`: which port (`A_cxn` or `B_cxn`) to route to the output port on the data switch.

***Supported operations.*** The CFM can be configured for routing or for the control operators in Sec. IV. Routing, e.g., `out = A`, is simple: just set `cxn = A_cxn`, `noc_valid = A_valid`, and `A_ready = noc_ready`.

Other operators are more involved, but each requires only a small state machine. Fig. 9 is pseudocode for steer and carry operators (Sec. IV). A steer forwards `A` if `D` is true; otherwise, it discards `A`. To implement steer, the CFM waits for `A` and `D` to be valid. If `D` is true, then `noc_valid` is raised, and the `noc_ready` signal propagates upstream to `A` and `D` and the CFM waits for `noc_ready`, i.e., for the value to be consumed. If `D` is false, then `noc_valid` is kept low, and `A_ready` and `D_ready` are raised to discard these tokens.

Carry is more complicated. Carry begins in *Initial* state, waiting for a valid `A` token. It forwards the token and transitions to *Blocked* state, where it forwards B until it sees a false D token. See the pseudocode in Fig. 9b for details.

***Control flow in the NoC adds small hardware overheads.*** Implementing control flow in the NoC is far more energy- and area- efficient than in a PE, saving an estimated 40% energy and 22% area v. CGRA with all CF operations mapped to PEs (All PEs in Fig. 16). The CFM deals only with narrow control signals and the 1b decider value D. It does not need to touch full data signals at all; these are left to the pre-existing data switch. Importantly, this means that the CFM adds no data buffers. Instead, the CFM simply raises the *_ready signals to park values in the upstream output channels until they are no longer needed.

By contrast, implementing control flow in a PE requires full data-width muxes and, if an entire PE is dedicated to control, an output channel to hold the results. Nevertheless, RipTide is sometimes forced to allocate a PE for control flow. Specifically, if a control-flow operator takes a software-supplied value or a constant other than -1, 0, or 1, it requires µcore support.

***Buffering of decider values.*** The CFM provides a small amount of buffering for decider values. This is because loop deciders often have high fanout, which means that the next iteration of a loop is likely blocked by one or more downstream consumers. To remove this limitation, RipTide provides a small amount of downstream buffering for 1b decider signals, improving performance with minimal impact on area. The CFM uses run-length encoding to buffer up to eight decider values with just 3b of additional state, yielding up to 3.8× performance (on dmm) at an area of cost of $< 1\%$.

## VII. Experimental Methodology

We evaluate a complete RipTide system: the compiler built using LLVM and the microarchitecture fully implemented in RTL in Intel 22FFL, an industrial, sub-28nm FinFET process.

***Compiler.*** RipTide's compiler passes extend LLVM 12.0 [48] and we compile workloads with -Oz to optimize code size. RipTide's compiler middle-end uses LLVM's flow-insensitive alias analyses for memory ordering. We evaluate both RipTide's SAT and ILP mappers (see Sec. V-D), but unless otherwise specified we use the ILP mapper. The ILP mapper uses CVXPY [25] and Gurobi 9.5 [35]. The SAT mapper uses CaDiCal [9] to rewrite and simplify the problem's clauses and then uses a new parallel SAT solver, developed concurrently and based on YalSAT [8], to find a valid mapping.

***Hardware.*** RipTide is implemented completely in RTL, including the 6×6 CGRA, RISC-V (RV32EMC) scalar core, and 256KB SRAM main memory. We use Cadence Xcelium to verify correctness and measure performance. We synthesize RipTide using Cadence Genus and a high-threshold-voltage, FinFET PDK with compiled memories. To estimate power, we simulate full benchmarks post-synthesis and use Cadence Joules to estimate power from annotated switching activities.

***Baselines.*** The evaluation compares to several baselines—scalar, vector, SNAFU, and three ASICs—also implemented entirely in RTL, using the same design flow. All baselines and RipTide use the same scalar core and main memory. The scalar baseline is a simple, six-stage microcontroller. The vector baseline adds a single-lane co-processor [30]. SNAFU is the state-of-the-art energy-minimal CGRA.

***Benchmarks.*** We evaluate ten workloads important to the ULP domain on random inputs. For the vector baseline, we vectorized all code by hand (except dfs, which does not vectorize well). SNAFU uses the vectorized code to generate its bitstreams. For RipTide, we compile and run the plain C implementation of each benchmark. The exceptions are sort, for which we use merge sort on the scalar core and radix sort for RipTide (because it maps entirely onto the fabric), and dmm, for which, where explicitly noted, we tune its C implementation to maximize efficiency by unrolling loops to align memory accesses with memory banks.

## VIII. Evaluation

We evaluate RipTide to show that it is easy to program in a high-level language and uses 25% less energy than the state-of-the-art energy-minimal design, while improving performance by 17% on average and up to 2.5×. Moreover, control flow in the NoC is essential for large workloads and reduces energy by up to 2.3×.

### *A. Main results*

***RipTide compiles high-level code to its fabric.*** RipTide compiles, schedules, and runs ten applications on its 6×6 fabric. For all but fft, RipTide offloads the entire benchmark onto the fabric, including outer loops. For fft, a 6×6 fabric does not have enough arithmetic or multiplier PEs, so we split fft into two separate functions. Finally, RipTide maps and runs dfs, which is *not possible* for the vector and SNAFU baselines (×s in the figures).

***RipTide saves energy.*** Fig. 10 presents energy of the scalar, vector, SNAFU, and ASICs normalized to RipTide. RipTide reduces energy by 6.6× v. scalar, 3.1× v. vector, and 25% v. SNAFU. RipTide uses less energy across the board. Fig. 10 breaks energy into memory, scalar, vector/CGRA, and CGRA NoC. RipTide saves energy v. scalar and vector because it does not fetch instructions, re-uses its configuration across many inputs, and forwards operands directly from producers to consumers. RipTide uses less energy than SNAFU by reducing scalar computation: RipTide runs outer loops on the fabric, but SNAFU runs them on the scalar core. RipTide's scalar core fetches 86% fewer instructions than SNAFU's, eliminating pipeline control, register-file access, and instruction fetch — as seen by RipTide's lower memory energy in Fig. 10.

The only benchmark for which memory energy increases v. SNAFU is fft. SNAFU uses scratchpads in the fabric for fft, which reduces main memory energy. Even without scratchpads, RipTide shows an overall energy reduction. (RipTide currently lacks a programming interface for scratchpads, but can easily support them in hardware.)

sconv shows how control-flow costs in RipTide move from scalar core to the fabric (e.g., steer, carry). While RipTide

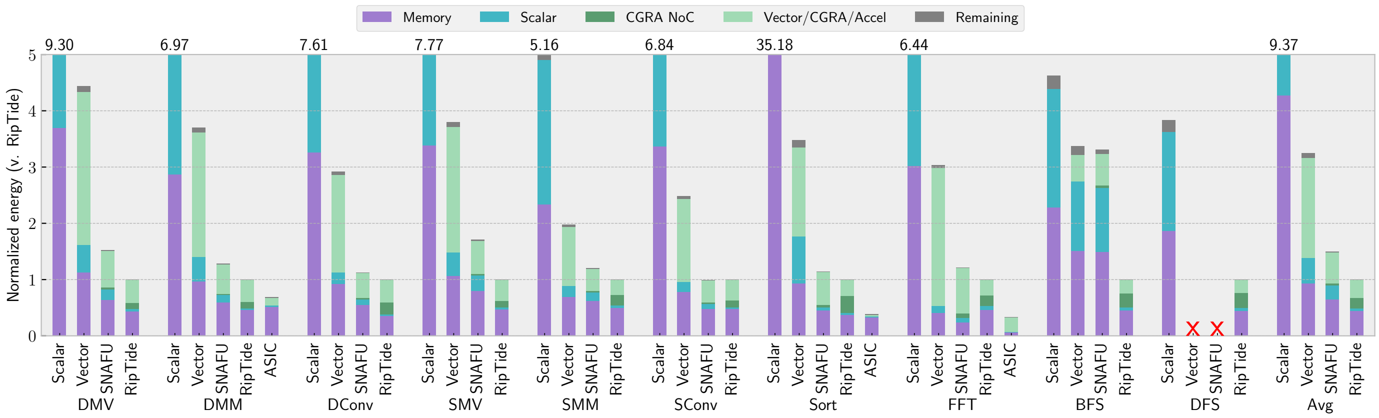


**Figure 10:** Energy (v. RipTide) of scalar, vector, SNAFU, RipTide across ten benchmarks. RipTide uses 25% less energy than SNAFU.

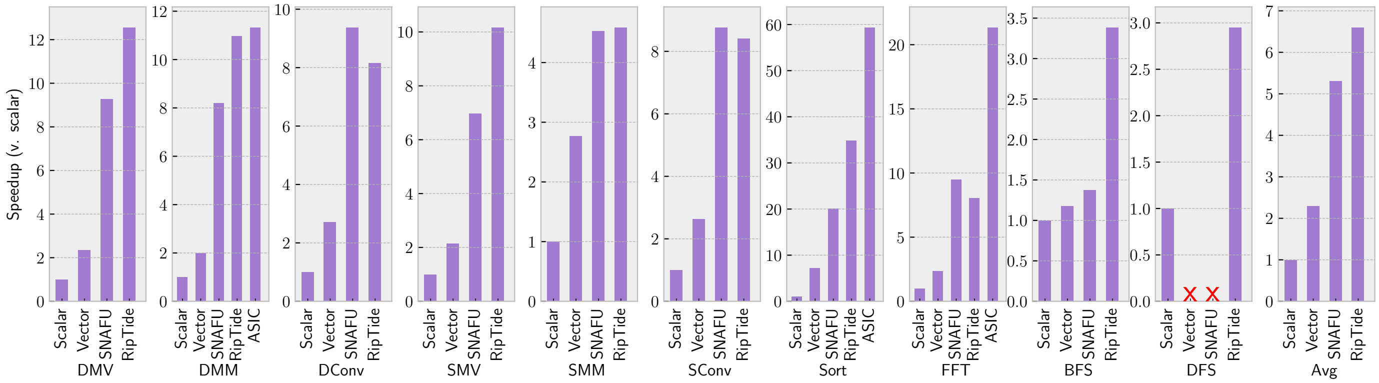


**Figure 11:** Speedup (v. scalar) of scalar, vector, SNAFU, and RipTide across ten benchmarks. RipTide is 17% faster than SNAFU.

reduces scalar energy, it adds fabric energy (v. SNAFU) to support outer loops. Scalar execution is a small fraction of overall energy for sconv, so RipTide provides no benefit on this benchmark. Moreover, comparing fabric energy for SNAFU and RipTide on sconv shows that RipTide's microarchitectural additions cost little energy.

***RipTide runs C programs with near-ASIC efficiency.*** Fig. 10 also compares RipTide to hand-coded, fixed-function ASICs for dmm, sort, and fft. RipTide uses 2.4× more energy on average than the ASICs while compiling programs directly from C. RipTide compares especially favorably to dmm, using 46% more energy. The data show that the cost of RipTide's programmability is low.

***RipTide is faster than prior energy-minimal CGRAs.*** Fig. 11 shows performance normalized to scalar. RipTide is 6.2×, 3.4×, and 17% faster than scalar, vector, and SNAFU. RipTide does especially well on bfs, with a 2.5× speedup v. SNAFU. The benefit comes from RipTide's ability to run bfs's irregular outer loop on the fabric, whereas SNAFU is bottlenecked on the scalar core because its fabric runs only inner loops. The only benchmarks where RipTide underperforms SNAFU are dconv, sconv, and fft. The difference in performance comes down to implementation: we tailor applications to each architecture to minimize energy, not maximize performance. On SNAFU, we re-order loops to maximize vector length, minimizing scalar work but adding some memory accesses; on RipTide, we can avoid these memory accesses by accumulating intermediate results in the fabric.

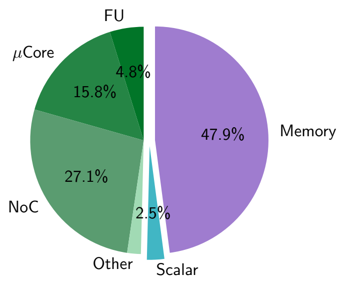


**Figure 12:** Area breakdown for a complete RipTide system. Area is dominated by the CGRA fabric and main memory (SRAM + arbitration logic), eaching taking about half of the system. The scalar core is just 2.5% of system area. The NoC takes 54% of CGRA area, and PEs (μcore + FUs) take 42%.

***RipTide is tiny and has extremely low power consumption.*** The complete RipTide system (CGRA, memory, and scalar core) is ≈ 0.5mm$^2$. Fig. 12 breaks down system area among its components. RipTide operates between 320μW and 910μW, with negligible leakage (< 3%) due to RipTide's high-threshold-voltage process. Overall, the complete system, including memory, achieves 180 MOPS/mW running a hand-tuned C implementation of dmm that unrolls twice along the output column dimension. Without tuning, RipTide achieves 141 MOPS/mW on dmm.

**Table III:** Comparison of RipTide to other low-power CGRAs. RipTide supports a broader set of programs while improving energy efficiency.

| | **HyCube testchip**$^\star$ **[96]** | **HM-HyCube REVAMP**$^\diamond$ **[7]** | **UE-CGRA**$^\dagger$ **[88, 89]** | **SNAFU**$^\diamond$ **[27]** | **RipTide**$^\diamond$ *(this work)* | |
|---|---|---|---|---|---|---|
| Irregular loops | ✗ | ✗ | ✓ | ✗ | ✓ | |
| Loop nesting | ✗ | ✗ | ✗ | ✗ | ✓ | |
| Memory ordering | ✗ | ✗ | ✗ | ✗ | ✓ | |
| Variable-latency ops | ✗ | ✗ | ✗ | ✗ | ✓ | |
| Node | 40LP | 22 | TSMC 28 | Intel 22FFL | Intel 22FFL | |
| Fabric dimensions | 4×4 | 6×6 | 8×8 | 6×6 | 6×6 | |
| Fabric area (mm$^2$) | — | 0.2 | 0.25 | 0.27 | 0.25 | |
| Frequency (MHz) | 488 | 100 | 750 | 50 | 50 | |
| Memory size (KBs) | 4 | 64 | 64 | 256 | 256 | |
| Benchmark | fft | Linear algebra | fft | fft | fft | dmm$^\ddagger$ |
| Fabric power (mW) | — | 8.4 | 14.0 | 0.54 | 0.24 | 0.50 |
| System power (mW) | 140 | — | 16.7 | 0.74 | 0.52 | 0.91 |
| Performance (MOPS) | 5380 | — | 625 | 71 | 62 | 164 |
| Fabric efficiency (MOPS/mW) | — | 103 | 45 | 134 | 254 | 328 |
| System efficiency (MOPS/mW) | 26 | — | 38 | 97 | 117 | 180 |

$^\star$ Silicon implementation. $^\dagger$ Post-P&R simulation. $^\diamond$ Post-synthesis simulation. $^\ddagger$ Hand-tuned C software.

### B. RipTide v. prior low-power CGRAs

Table III compares RipTide against several recent CGRAs. We compare designs across their general-purpose programmability, architectural parameters, and reported performance, power, and efficiency. RipTide supports a broader range of programs and is more energy-efficient than prior CGRAs.

***Making a fair comparison.*** Table III gives absolute numbers for different designs and does not re-scale them to normalize the node. These numbers are our best effort at accurately characterizing prior designs v. RipTide. Few prior CGRAs admit meaningful comparison, however, because prior work reports performance, power, and efficiency inconsistently.

Concrete numbers for energy efficiency are hard to come by. Many prior CGRAs focus on performance [74, 93] or mapping [44, 50, 99] and report metrics (e.g., initiation interval) that are not the focus of RipTide. Others report relative results [59, 89] or use high-level models [59, 98] that make quantitative comparison difficult.

Differences in measurement methodology also make it challenging to compare reported results. Prior CGRAs often report *total* operation count, including, e.g., loads, stores, and loop control, or are unclear about which operations are counted [21, 46, 68]. These numbers, though often reported as MOPS [21, 46], are closer to MIPS as defined for traditional CPUs. Table III counts only essential arithmetic operations,[1] and we have verified with the authors of other designs in Table III that they count MOPS the same way. Finally, many prior CGRAs report power for the fabric only, excluding, e.g., the core and memory [7, 47, 68]. We report both fabric and full-system power, and focus on the latter. *Full-system MOPS/mW is the most important metric for the applications targeted by RipTide.*

***RipTide is more programmable than prior CGRAs.*** Table III highlights a number of programming features supported by RipTide that are unsupported by prior CGRAs. In addition to making RipTide easier to program (Table I), these features improve energy efficiency by allowing RipTide to offload a larger fraction of a program onto the efficient CGRA fabric.

***RipTide is more energy-efficient than prior CGRAs.*** RipTide is the most energy-efficient CGRA by a significant margin. Scaled to 22nm, both the HyCube testchip [96] and UE-CGRA [88] achieve roughly 48 full-system MOPS/mW on fft. RipTide achieves 117 full-system MOPS/mW, which is 2.4× better, even including RipTide's larger memory and despite fft being the only kernel to not fit entirely on RipTide's fabric. On dmm with loop unrolling, efficiency improves to 180 full-system MOPS/mW.

Using a different measurement methodology changes the absolute results dramatically, highlighting the challenge of making apples-to-apples comparisons between CGRAs. If we count *all* operations, instead of only essential arithmetic (i.e., MIPS), RipTide achieves 400 MIPS/mW on fft. If we measure only the fabric, RipTide achieves 254 MOPS/mW and 859 MIPS/mW on fft — increasing reported efficiency by 7.3× v. full-system MOPS/mW.

Measuring only the fabric, a recent version of HyCube generated by REVAMP [7] achieves 103 fabric-only MOPS/mW, averaging across several linear algebra benchmarks. (A tuned, heterogeneous fabric achieves 172MOPS/mW.) RipTide achieves 328 fabric-only MOPS/mW on unrolled dmm. Meaningful comparisons thus require a detailed understanding of what is being measured.

***RipTide's area is similar to prior CGRAs.*** RipTide is somewhat larger than prior CGRAs (7000 μm$^2$/PE for RipTide, v. 5500 μm$^2$/PE for HyCube [7][2] and 3900 μm$^2$/PE for UE-CGRA). Differences in NoC design help explain these discrepancies. UE-CGRA has no routers, instead routing values through PEs. HyCube's NoC accounts for 24% of fabric area, with each PE containing a $4 \times 4$ crossbar switch. RipTide's NoC

[1] Specifically, $2n^3$ ops for dmm and $10n\log_2 n + O(n)$ ops for fft.

[2] HM-HyCube generated using REVAMP [7]. The HyCube testchip [96] area includes I/O pads, etc., and is not directly comparable.

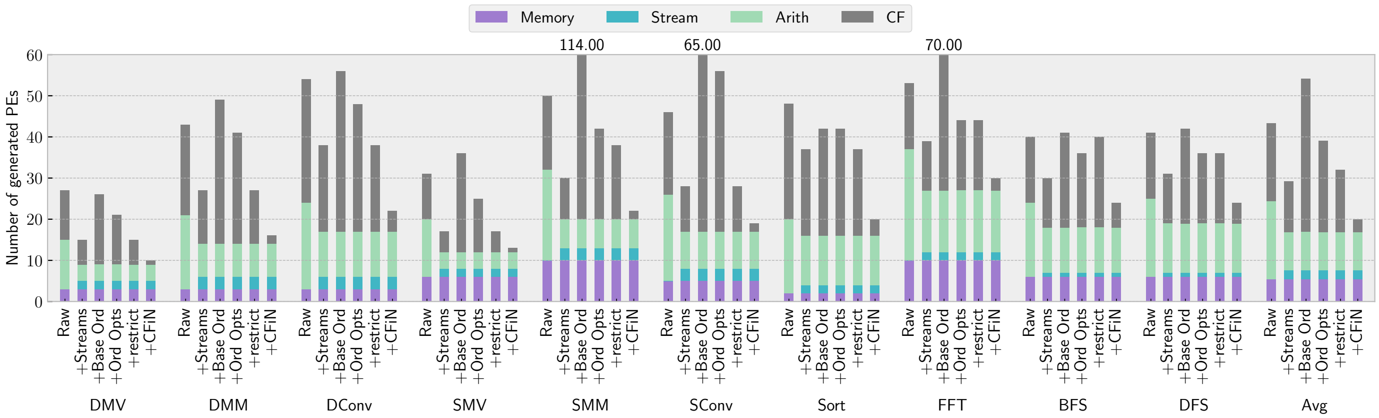


**Figure 13:** Operator counts for ten different benchmarks. Starting with an unoptimized, unordered baseline (Raw), compiler optimizations reduce operator counts while enforcing memory ordering, making it feasible to map benchmarks to hardware.

accounts for 54% (0.14µm$^2$) of fabric area, but offers more connectivity (more links and $8 \times 8$ switches) and capability (dynamic flow control and control-flow operators). SNAFU's 2D-mesh NoC is 0.11µm$^2$ and uses the same switch design as RipTide, showing that RipTide's 2D-torus topology and CFMs add modest overhead ($<$ 0.03µm$^2$).

***RipTide targets a different design point.*** As currently evaluated, RipTide is much slower than prior CGRAs. We evaluate RipTide at 50 MHz, v. 100s of MHz for prior designs. RipTide has significant slack at 50 MHz and could run much faster. We have not yet pushed frequency further due to RipTide's bufferless NoC and top-down synthesis flow, which requires additional tooling to estimate worst-case critical path. (A similar problem arises in FPGAs.) Frequency in the 10s of MHz is common in ULP microcontrollers.

Nevertheless, lower frequency means that RipTide's raw performance is well below prior CGRAs: on fft, 62 MOPS for RipTide v. 5,380 MOPS for HyCube and 625 MOPS for UE-CGRA. Factoring out frequency, RipTide achieves 1.24 ops/cycle v. 11 ops/cycle for HyCube and 0.83 ops/cycle for UE-CGRA. RipTide's lower ops/cycle is partly by design: RipTide trades performance for efficiency by mapping a single operation to each PE, whereas HyCube maps multiple operations per PE to maximize utilization. This tradeoff makes sense for RipTide because it targets applications that are limited by energy, not performance (Sec. II).

Combining RipTide's low frequency and high energy efficiency yields extremely low power consumption. RipTide draws 2–3 orders-of-magnitude less power than HyCube and UE-CGRA. Only SNAFU and RipTide draw less than 1 mW — and this is the entire system, including the 256KB main memory.

### C. Compiler characterization

RipTide's compiler effectively optimizes dataflow graphs, reducing operation count by 27% (or by 52% excluding operations mapped to the NoC) while enforcing memory ordering v. an unoptimized DFG without ordering. The compiler also reduces programmer effort: RipTide compiles from C with no hand-coded assembly, requiring just 8.7 added LoC on average over the original C (mostly for wrappers). Lastly, the compiler is fast — the SAT mapper finds a solution to each benchmark in $<3$ min and uses only 4.7% more energy than the ILP mapper.

***RipTide's compiler reduces operation count.*** Reducing operation count is important because operations consume PEs in RipTide's fabric. Fig. 13 shows operation counts by type with different optimizations applied. The first bar is an unoptimized DFG mapped to RipTide. This DFG requires many PEs to map to hardware and may yield incorrect results because it does not enforce memory ordering. The second bar adds streams, operator fusion, and redundant control-flow elimination, reducing operation count by 33%. The third bar adds unoptimized memory ordering, which *increases* operation count by 82% to ensure correctness. Mapping this graph to hardware is challenging due to its size. The fourth bar applies RipTide's ordering optimizations (Sec. V), reducing operation count (v. the third bar) by 18%. The fifth bar adds programmer-inserted annotations on pointers (C's restrict keyword) to better inform LLVM's alias analysis, reducing operation count by 16%. The last bar removes control-flow operations that map to RipTide's NoC, reducing the number of operations on PEs by 35%, demonstrating the benefit of RipTide's control flow in the NoC. Between RipTide's compiler optimizations and implementation of control flow in the the NoC, RipTide reduces operations mapped to PEs by 52% (first v. last bar) while enforcing memory ordering.

***RipTide reduces programmer effort.*** Fig. 14 counts code additions, including lines of code (LoC) in C, assembly, and restrict annotations. RipTide has no hand-written assembly, compiling directly from C, while 32% and 27% of the LoC for the vector and SNAFU baselines are hand-written assembly. On average, vector adds 17 LoC v. scalar, SNAFU adds 21 LoC v. scalar, and RipTide adds just 8.7 lines. Annotations in RipTide represent a small fraction of the overall LoC, just 11.2% and, on average, the programmer adds 4.5 restrict annotations per benchmark.

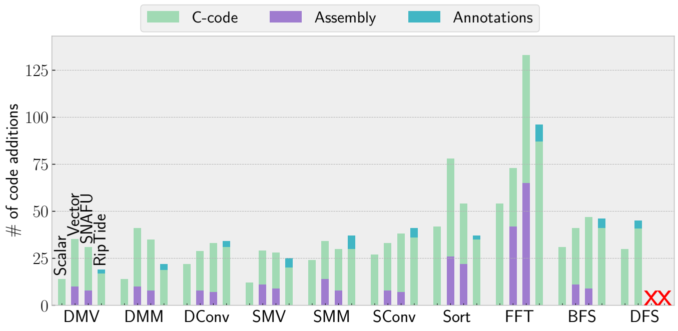
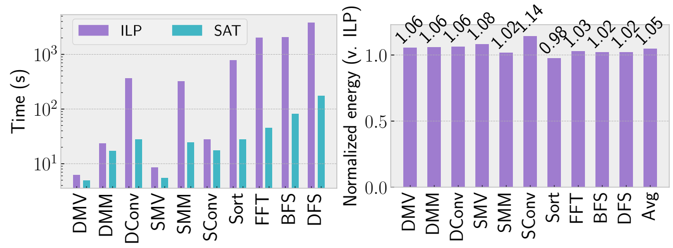


**Figure 14:** The number of code additions for ten benchmarks running on scalar, vector, SNAFU, and RipTide. RipTide requires no hand-coded assembly unlike vector and SNAFU.



*(a) Compilation time.*

*(b) Normalized energy.*

**Figure 15:** Compilation time (16 threads, Intel i9-9900K) and normalized energy (v. ILP) of SAT and ILP mappers. SAT is 15.1× faster than ILP, but uses 4.7% more energy.

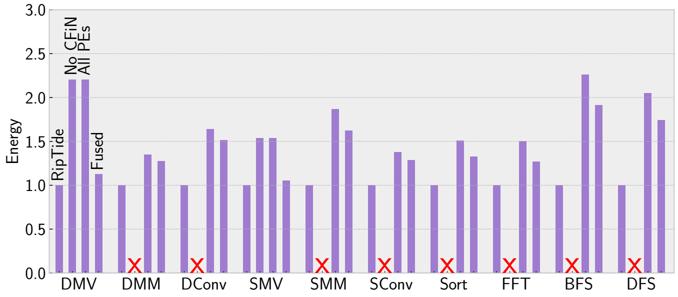
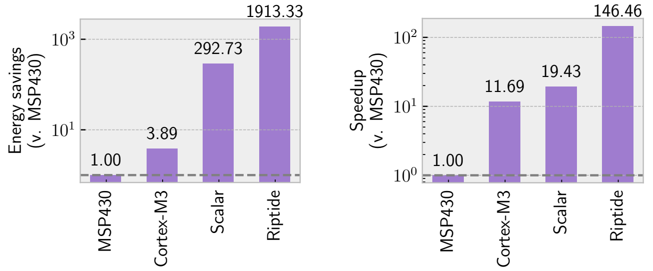


**Figure 16:** Control flow in the NoC saves energy. RipTide uses 45%, 40%, and 27% less energy than RipTide w/ No CFiN, a fabric where all CF ops are PEs ("All PEs"), and a fabric that fuses CF ops into PEs ("Fused").



*(a) Energy savings.*

*(b) Speedup.*

**Figure 17:** Energy savings and speedup of DNN inference on RipTide v. MSP430, ARM Cortex-M3, and our scalar core.

***ILP v. SAT.*** Fig. 15a shows the end-to-end compilation times for RipTide using its SAT and ILP mappers. Fig. 15b compares the energies of the resulting mappings. SAT is 15.1× faster than ILP on average, finding solutions to most benchmarks in under a minute. Rapid compilation makes SAT appropriate for iterative software development. On the other hand, ILP produces mappings that use 4.3% less energy on average, making it ideal for final optimization prior to deployment.

The consistently narrow energy gap between SAT and ILP suggests that good solutions are dense in RipTide; i.e., any valid mapping found by SAT is close to the optimal energy from ILP. RipTide does not time-multiplex PEs, so mapping affects energy largely through routing distance. But the loss in routing distance is constrained by routability (i.e., any valid mapping will tend to place dependent operations close to one another), and the energy impact of routing distance in RipTide is reduced by its bufferless NoC. These observations help to explain why SAT performs well in RipTide.

### *D. Control flow in the NoC saves energy & area*

Fig. 16 quantifies the benefits of implementing control flow in the NoC. From left to right, the plot shows energy on:

- RipTide: Control flow implemented in the NoC (CFiN).
- No CFiN: Control flow mapped to PEs on a 6×6 fabric.
- All PEs: Control flow mapped to PEs, and fabric size increased as necessary to fit each benchmark.
- Fused: One control-flow operator fused into each PE, and fabric sized increased as necessary to fit each benchmark.

We synthesize the first two configurations to estimate energy, while for the latter two configurations we extrapolate energy using area and power estimates for control-flow modules and PEs derived from the synthesized configurations.

RipTide uses the least energy: 45% less than No CFiN, 40% less than All PEs, and 27% less than Fused. RipTide's energy benefit stems from CFiN avoiding the overhead of a full PE. Other configurations also have unique problems. No-CFiN is possible only for dmv and smv, which are small enough to map to the same RipTide fabric; other workloads have too many control-flow operators to map. The All PEs and Fused configurations add many control-flow PEs, wasting energy and area: RipTide is 22% and 17% smaller than All PEs and Fused, respectively.

### *E. End-to-end case study: RipTide makes saving energy easy*

To evaluate the experience of developing for RipTide we deployed a full application — DNN inference — to the fabric. This experiment also allowed us to demonstrate the efficiency and performance of RipTide v. commercial, off-the-shelf ULP microcontrollers.

The DNN we chose is a derivative of LeNet [49] and has four layers: two convolution layers separated into three sublayers followed by two fully connected layers. Every layer is offloaded to RipTide's fabric, including convolution, fully connected, activation, pooling, and normalization layers. It was a straightforward process once we had working scalar code — the compiler worked out of the box.

Fig. 17 shows the energy and performance of RipTide running inference v. two COTS MCUs — TI MSP430FR5994 [40] and Arm Cortex-M3 [1] — as well as our scalar design. For the MSP430 and Cortex-M3, we run the network on real hardware and use a digital multimeter to measure current draw, which matches datasheets. Fig. 17a shows the massive energy savings of RipTide. RipTide achieves 64MOPS/mW, which is 1900× more than the MSP430 (0.03 MOPS/mW), 490× more than the Cortex-M3 (0.13 MOPS/mW), and 6.6× more than our scalar design (9.5 MOPS/mW). Even accounting for technology scaling, RipTide still saves roughly 321× v. MSP430 and 83× energy v. Cortex-M3. The reason for such poor efficiency in the MCUs v. RipTide (and even our scalar design) seems to be their non-volatile main memory. Fig. 17b shows that RipTide is also significantly faster: by 146× v. MSP430, 13× v. Cortex-M3, and 7.7× v. our scalar design.

## IX. Conclusion: Implications for General-Purpose Architecture and Dark Silicon

Fig. 10 and Fig. 11 compare the energy and performance for `dmm` on RipTide v. an equivalent ASIC. RipTide does not compromise much on energy or performance — coming within 46% and 3%, respectively — but it is not a free lunch. There is a high area cost for RipTide's programmability: RipTide is 57× larger than the ASIC.[3] The question is, is RipTide's programmability worth the extra area?

RipTide area is inflated partly because of low utilization on PEs that perform outer loops. RipTide only supports one operation per PE, so entire PEs are consumed even if an operation fires rarely. A future design could revisit this constraint to allow limited time-multiplexing, either at a fine [98] or coarse [59] granularity.

Regardless, the area difference shows potentially large cost savings from ASICs, so long as a computation is performed frequently enough to overcome ASICs' upfront design and verification costs. Standardized, pervasive tasks like JPEG compression and wireless communication protocols are good candidates for ASICs. But if the computation is prone to change or used infrequently, then this cost advantage rapidly disappears.

Some have proposed that, with increasing transistor budgets and stagnating power budgets, processors should embrace extreme heterogeneity and assemble a large number of distinct ASICs [87,91]. The "garden of ASICs" approach lets architects do something with extra transistors, but it significantly increases system design and verification cost. Moreover, it creates herculean challenges in system integration, as there is no standard programming interface for ASICs, obsolescence is monotonic and likely inevitable, and programs must be somehow partitioned between ASICs and cores with accompanying data-coordination issues.

RipTide suggests an alternative approach. Rather than spend area on ASICs that will idle most of the time, instead build an energy-minimal, programmable dataflow fabric. The two designs take similar area with a few dozen ASICs. And the dataflow fabric is cheaper to design, more broadly applicable, and easier to use — programs can be simply compiled for a different target. Finally, as a general-purpose design, programmable dataflow fabrics can create a self-sustaining ecosystem that aggregates optimizations and achieves sufficient scale to justify cutting-edge silicon. All told, while dataflow fabrics like RipTide are not a replacement for ASICs, they will play an important role in improving the efficiency of general-purpose processing as designs are increasingly constrained by energy instead of area, and they will reduce the demand for specialized hardware to accelerate the majority of applications.

## X. Acknowledgments

We thank the reviewers for their time and thoughtful feedback. This work was supported by NSF CCF-1815882, Graham Gobieski was supported by the Apple Scholars in AI/ML fellowship, and Souradip Ghosh by the U.S. Department of Energy Computational Science Graduate Fellowship (DE-SC0022158). Parts of the compiler development were supported by Semiconductor Research Corporation (SRC) Artificial Intelligence Hardware (AIHW), a Global Research Collaboration (GRC) program. We would also like to thank the authors of HyCube [42, 96], REVAMP [7], and UE-CGRA [88, 89] for their help in gathering data for comparison.

## References

[1] "Stm32l152re." [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l152re.html

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972. [Online]. Available: https://doi.org/10.1137/0201008

[4] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 242–249. [Online]. Available: https://doi.org/10.1145/190347.190423

[5] M. Balasubramanian, S. Dave, A. Shrivastava, and R. Jeyapaul, "Laser: A hardware/software approach to accelerate complicated loops on cgras," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1069–1074.

[6] M. Balasubramanian and A. Shrivastava, "Pathseeker: a fast mapping algorithm for cgras," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 268–273.

[7] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "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*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 918–932. [Online]. Available: https://doi.org/10.1145/3503222.3507772

[8] A. Biere, "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014," in *Proc. of SAT Competition 2014 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, A. Balint, A. Belov, M. Heule, and M. Järvisalo, Eds., vol. B-2014-2. University of Helsinki, 2014, pp. 39–40.

[3] This is without including main memory, which is half of chip area. Also, `dmm` is an extreme case; e.g., RipTide is 9× larger than `fft`. On `fft`, the ASIC yields larger improvements in energy (saving 67%) and performance (by 62%) v. RipTide. This is because RipTide has too few resources to offload the entire `fft` kernel and the ASIC uses scratchpads for twiddle factors.

[9] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger, "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020," in *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Report Series B, T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1. University of Helsinki, 2020, pp. 51–53.

[10] M. Budiu, P. Artigas, and S. Goldstein, "Dataflow: A complement to superscalar," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, 2005, pp. 177–186.

[11] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 2005, pp. 177–186.

[12] D.-K. Chen and P.-C. Yew, "Redundant synchronization elimination for doacross loops," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 459–470, 1999.

[13] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[14] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "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, 2017, pp. 184–189.

[15] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of cgra," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 9–16.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[17] V. Dadu, S. Liu, and T. Nowatzki, *PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators*. IEEE Press, 2021, p. 595–608. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00053

[18] V. Dadu and T. Nowatzki, *TaskStream: Accelerating Task-Parallel Workloads by Recovering Program Structure*. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–13. [Online]. Available: https://doi.org/10.1145/3503222.3507706

[19] 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.

[20] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, vol. 41, no. 7, 2008.

[21] S. Das, D. Rossi, K. J. Martin, P. Coussy, and L. Benini, "A 142mops/mw integrated programmable array accelerator for smart visual processing," in *ISCAS*, 2017.

[22] S. Dave, M. Balasubramanian, and A. Shrivastava, "Ureca: Unified register file for cgras," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1081–1086.

[23] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *ASPLOS 25*, 2020.

[24] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, no. 4, 1975.

[25] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.

[26] M. Duric, O. Palomar, A. Smith, O. Unsal, A. Cristal, M. Valero, and D. Burger, "Evx: Vector execution on low power edge cores," in *DATE*, 2014.

[27] 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.

[28] G. Gobieski, N. Beckmann, and B. Lucia, "Intermittent deep neural network inference," in *SysML*, 2018.

[29] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *ASPLOS*, 2019.

[30] 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 *MICRO*, 2019.

[31] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, 2000.

[32] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, 2012.

[33] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 12–23.

[34] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H.-H. S. Lee, G.-Y. Wei, D. Brooks, and C.-J. Wu, "Chasing carbon: The elusive environmental footprint of computing," *IEEE Micro*, 2022.

[35] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[36] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1284–1291.

[37] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Branch-aware loop mapping on cgras," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[38] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, p. 848–894, jul 1999. [Online]. Available: https://doi.org/10.1145/325478.325519

[39] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[40] T. Instruments, "Msp430fr5994 sla," 2017. [Online]. Available: http://www.ti.com/lit/ds/symlink/msp430fr5994.pdf

[41] 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.

[42] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *DAC*, 2017.

[43] M. Karunaratne, C. Tan, A. Kulkarni, T. Mitra, and L.-S. Peh, "Dnestmap: mapping deeply-nested loops on ultra-low power cgras," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[44] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4d-cgra: Introducing branch dimension to spatio-temporal application mapping on cgras," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[45] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor, "Moonwalk: Nre optimization in asic clouds," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 511–526. [Online]. Available: https://doi.org/10.1145/3037697.3037749

[46] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications," in *ICFPT*, 2012.

[47] Y. Kim and R. N. Mahapatra, "Hierarchical reconfigurable computing arrays for efficient cgra-based embedded systems," in *Proceedings of the 46th Annual Design Automation Conference*, 2009, pp. 826–831.

[48] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, Mar. 2004.

[49] Y. Le Cun, L. Jackel, B. Boser, J. Denker, H. Graf, I. Guyon, D. Henderson, R. Howard, and W. Hubbard, "Handwritten digit recognition: Applications of neural network chips and automatic learning," *IEEE Communications Magazine*, vol. 27, no. 11, 1989.

[50] J. Lee and T. E. Carlson, "Ultra-fast cgra scheduling to enable run time, programmable cgras," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1207–1212.

[51] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "Chordmap: Automated mapping of streaming applications onto cgra," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 306–319, 2021.

[52] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent Computing: Challenges and Opportunities," Dagstuhl, Germany, 2017. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7131

[53] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[54] S. Midkiff and D. Padua, "A comparison of four synchronization optimization techniques," in *Intl. Conf. on Parallel Processing*, vol. 2, 1991, pp. 9–16.

[55] S. P. Midkiff and D. A. Padua, "Compiler algorithms for synchronization," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1485–1495, 1987.

[56] E. Mirsky, A. DeHon *et al.*, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources." in *FCCM*, vol. 96, 1996, pp. 17–19.

[57] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, 2006.

[58] T. Miyamori and K. Olukotun, "Remarc: Reconfigurable multimedia array coprocessor," *IEICE Transactions on information and systems*, vol. 82, no. 2, pp. 389–397, 1999.

[59] 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.

[60] C. Nicol, "A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing," *WaveComputing WhitePaper*, 2017.

[61] R. S. Nikhil *et al.*, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on computers*, vol. 39, no. 3, 1990.

[62] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *PACT 27*, 2018.

[63] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: ACM, 2018, pp. 36:1–36:15. [Online]. Available: http://doi.acm.org/10.1145/3243176.3243212

[64] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *ISCA 44*, 2017.

[65] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 298–310.

[66] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, no. 3, 2017.

[67] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," *ACM SIGPLAN Notices*, vol. 48, no. 6, 2013.

[68] N. Ozaki, Y. Yasuda, M. Izawa, Y. Saito, D. Ikebuchi, H. Amano, H. Nakamura, K. Usami, M. Namiki, and M. Kondo, "Cool mega-arrays: Ultralow-power reconfigurable accelerator chips," *IEEE Micro*, vol. 31, no. 6, 2011.

[69] J. Pager, R. Jeyapaul, and A. Shrivastava, "A software scheme for multithreading on cgras," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 1, pp. 1–26, 2015.

[70] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, p. 82–91, may 1990. [Online]. Available: https://doi.org/10.1145/325096.325117

[71] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel *et al.*, "Triggered instructions: a control paradigm for spatially-programmed architectures," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013.

[72] H. Park, Y. Park, and S. Mahlke, "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*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 370–380. [Online]. Available: https://doi.org/10.1145/1669112.1669160

[73] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik, "Chlorophyll: Synthesis-aided compiler for low-power spatial architectures," *SIGPLAN Not.*, vol. 49, no. 6, p. 396–407, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594339

[74] 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 *ISCA 44*, 2017.

[75] A. Rucker, M. Vilim, T. Zhao, Y. Zhang, R. Prabhakar, and K. Olukotun, "Capstan: A vector rda for sparsity," 2021.

[76] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *ISCA 30*, 2003.

[77] K. Sankaralingam, T. Nowatzki, G. Wright, P. Palamuttam, J. Khare, V. Gangadhar, and P. Shah, "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, 2021, pp. 1–20. [Online]. Available: https://doi.org/10.1109/HCS52781.2021.9567306

[78] M. Satyanarayanan, N. Beckmann, G. A. Lewis, and B. Lucia, "The role of edge offload for hardware-accelerated mobile devices," in *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 22–29. [Online]. Available: https://doi.org/10.1145/3446382.3448360

[79] M. Satyanarayanan, N. Beckmann, G. A. Lewis, and B. Lucia, "The role of edge offload for hardware-accelerated mobile devices," in *HotMobile*, 2021.

[80] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[81] P. Sparks, "A route to a trillion devices," *Arm WhitePaper*, 2017.

[82] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, pp. 1–25, 2014.

[83] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *MICRO 36*, 2003.

[84] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, "Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables," in *ISCA 45*, 2018.

[85] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 381–388.

[86] F. Tavares, "Kicksat 2," May 2019. [Online]. Available: https://www.nasa.gov/ames/kicksat

[87] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *DAC*, 2012.

[88] C. Torng and P. Pan, "Ue-cgra hpca 2021 artifact," Mar 2021. [Online]. Available: https://github.com/cornell-brg/torng-uecgra-scripts-hpca2021

[89] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 412–425.

[90] N. Vedula, A. Shriraman, S. Kumar, and W. N. Sumner, "Nachos: Software-driven hardware-assisted memory disambiguation for accelerators," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 710–723.

[91] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, 2010.

[92] 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)*, 2020, pp. 309–321.

[93] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," *ACM SIGARCH computer architecture news*, vol. 42, no. 3, 2014.

[94] D. Voitsechov, O. Port, and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," in *MICRO 51*, 2018.

[95] E. Waingold *et al.*, "Baring It All to Software: Raw Machines," in *IEEE Computer*, September 1997.

[96] B. Wang, M. Karunarathne, A. Kulkarni, T. Mitra, and L.-S. Peh, "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, 2019, pp. 133–136.

[97] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: synthesizing programmable spatial accelerators," in *ISCA 47*, 2020.

[98] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA*, 2020.

[99] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[100] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "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*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541961

[101] Y. Yang, J. S. Emer, and D. Sanchez, "Spzip: architectural support for effective data compression in irregular applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1069–1082.

[102] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for cgra modulo scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2201–2219, 2020.

## APPENDIX

**Table IV:** Inputs & variables of ILP & SAT formulations.

| **Input** | **Explanation** |
|---|---|
| $V$ | Set of DFG vertices |
| $E$ | Set of DFG edges |
| $N$ | Set of hardware nodes (PEs & CF-modules) |
| $F$ | Set of CF-modules $F \subset N$ |
| $R$ | Set of hardware routers |
| $L$ | Set of hardware links |
| $C_{el}(E,L) = 1$ if $e$ can map to $l$ | Edge-link compatibility matrix |
| $C_{vn}(V,N) = 1$ if $v$ can map to $n$ | Vertex-node compatibility matrix |
| $H_{ln}(L,N) = 1$ if $l$ originates from $n$ | Link-to-node matrix |
| $H_{nl}(N,L) = 1$ if $l$ comes from $n$ | Node-to-link matrix |
| $H_{lr}(L,R) = 1$ if $l$ originates from $r$ | Link-to-router matrix |
| $H_{rl}(R,L) = 1$ if $l$ comes from $r$ | Router-to-link matrix |
| **Variable** | **Explanation** |
| $M_{vn}(V,N) = 1$ if $v$ is mapped to $n$ | Vertex-to-node matrix |
| $M_{el}(E,L) = 1$ if $e$ is mapped to $l$ | Edge-to-link matrix |

Table IV lists the inputs and variables of the SAT and ILP formulations for mapping. The goal of either mapper is to solve for $M_{vn}$ and $M_{el}$, which map a DFG's vertices to hardware PEs and CF-modules (hardware nodes) and a DFG's edges to hardware links, respectively. Matrices $C_{vn}$ and $C_{el}$ capture the compatibility of a DFG's vertex-to-hardware node (i.e., a memory operation must be mapped to a memory PE) and a DFG's edge-to-hardware link (to make sure ports match), respectively. The remaining matrices $H_{nl}$, $H_{ln}$, $H_{rl}$, $H_{lr}$, describe the topology of the CGRA fabric by specifying the connectedness of links to hardware nodes and routers.

**Table V:** ILP formulation.

**Objective:** ***minimize*** $\sum_{e \in E, l \in L} M_{el}(e,l)$ *subject to*

| **Constraint** | **Explanation** |
|---|---|
| $\forall e \in E, l \in L, M_{el}(e,l) \leq C_{el}(e,l)$ | Edges are mapped to compatible links |
| $\forall v \in V, n \in N, M_{vn}(v,n) \leq C_{vn}(v,n)$ | Vertices are mapped to compatible nodes |
| $\forall v \in V, \sum_{n \in N} M_{vn}(v,n) = 1$ | Every vertex must be mapped to a node |
| $\forall n \in N, \sum_{v \in V} M_{vn}(v,n) \leq 1$ | No node can be used by more than one vertex |
| $\forall e \in E, r \in R, \sum_{l \in L} M_{el}(e,l) H_{lr}(l,r) = \sum_{l \in L} M_{el}(e,l) H_{rl}(r,l)$ | Flow into a router must equal the flow out |
| $\forall e \in E, n \in N \mid n \notin F, \sum_{l \in L} M_{e,l}(e) H_{nl}(n,l) = M_{vn}(src(e),n)$ | If a vertex is mapped to a non-CF node, then the output edges are mapped to outgoing links |
| $\forall e \in E, n \in N \mid n \notin F, \sum_{l \in L} M_{e,l}(e) H_{ln}(l,n) = M_{vn}(dst(e),n)$ | If a vertex is mapped to a non-CF node, then the input edges are mapped to incoming links |
| $\forall l \in L, e_1 \in E, M_{el}(e_1,l) + max_{e_2 \in E \mid src(e_1) \neq src(e_2)} M_{el}(e_2,l) \leq 1$ | Edges that do not share the same source are not mapped to the same links |
| $\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e,l) H_{ln}(l,n) + \sum_{v \in V} M_{vn}(v,n) \geq \sum_{l \in L} M_{el}(e,l) H_{nl}(n,l)$ | Unused CF-modules can pass through edges |
| $\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e,l) H_{ln}(l,n) \leq \sum_{v \in V} M_{vn}(v,n) + \sum_{l \in L} M_{el}(e,l) H_{nl}(n,l)$ | Unused CF-modules can pass through edges |
| $\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e,l) H_{nl}(n,l) \geq M_{vn}(src(e),n)$ | If a vertex is mapped to a CF node, then the output edges are mapped to outgoing links |
| $\forall e \in E, n \in F, \sum_{l \in L} M_{el}(e,l) H_{ln}(l,n) \geq M_{vn}(dst(e),n)$ | If a vertex is mapped to a CF node, then the input edges are mapped to incoming links |

$src(e) := v \in V$ and v is the source of e | $dst(e) := v \in V$ and v is the destination of e

Table V describes the (binary) ILP formulation. The formulation minimizes average routing distance given the constraints in the table.

**Table VI:** SAT formulation.

| **Clause** | **Explanation** |
|---|---|
| $\forall e \in E, l \in L \mid C_{el}(e,l) = 0, \neg M_{el}(e,l)$ | Edges are mapped to compatible links |
| $\forall v \in V, n \in N \mid C_{vn}(v,n) = 0, \neg M_{vn}(v,n)$ | Vertices are mapped to compatible nodes |
| $\forall v \in V, ExactlyOne(\{M_{vn}(v,n) \mid n \in N\})$ | Every vertex must be mapped to a node |
| $\forall n \in N, AtMostOne(\{M_{vn}(v,n) \mid v \in V\})$ | No node can be used by more than one vertex |
| $\forall r \in R, e \in E, \vee_{l \mid H_{rl}(r,l)} M_{el}(e,l) \iff \vee_{H_{lr}(l,r)} M_{el}(e,l)$ | An edge mapped to incoming link to a router must also be mapped to an outgoing link |
| $\forall r \in R, e \in E, AtMostOne(\{M_{el}(e,l) \mid l \in L \; and \; H_{rl}(r,l)\})$ | An edge can only be mapped to a single outgoing link of a router |
| $\forall e \in E, n \in N \mid n \notin F, \vee_{l \mid H_{nl}(n,l)} M_{el}(e,l) \iff M_{vn}(src(e),n)$ | If a vertex is mapped to a non-CF node, then the input edges are mapped to incoming links |
| $\forall e \in E, n \in N \mid n \notin F, \vee_{l \mid H_{ln}(l,n)} M_{el}(e,l) \iff M_{vn}(dst(e),n)$ | If a vertex is mapped to a non-CF node, then the output edges are mapped to outgoing links |
| $\forall l \in L, e_1 \in E, e_2 \in E \mid src(e_1) \neq src(e_2), \neg M_{el}(e_1,l) \vee \neg M_{el}(e_2,l)$ | Edges that do not share the same source are not mapped to the same links |
| $\forall e \in E, n \in F, K_{nl}(e,n) \vee \neg M_{vn}(src(e),n)$ | If a vertex is mapped to a CF node, then the output edges are mapped to outgoing links |
| $\forall e \in E, n \in F, K_{ln}(e,n) \vee \neg M_{vn}(dst(e),n)$ | If a vertex is mapped to a CF node, then the input edges are mapped to incoming links |
| $(\forall e \in E, n \in F, (K_{ln}(e,n) \vee K_n \vee \neg K_{nl}(e,n)) \wedge$<br>$(\neg K_{ln}(e,n) \vee K_n(n) \vee K_{nl}(e,n)) \wedge \; (\neg K_{ln}(e,n) \vee \neg M_{vn}(src(e),n))$ | Unused CF-modules can pass through edges |
| $\forall e \in E, n \in F, l \in L \mid H_{ln}(l,n), \neg M_{el}(e,l) \vee K_{ln}(e)$ | An edge mapped to an output link of CF-module cannot be mapped to an input of the CF-module |

$src(e) := v \in V$ and v is the source of e | $dst(e) := v \in V$ and v is the destination of e | $K_{nl}(e,n) := \vee_{l \mid H_{nl}(n,l)} M_{el}(e,l)$ | $K_{ln}(e,n) := \vee_{l \mid H_{ln}(l,n)} M_{el}(e,l)$ | $K_n(n) := \vee_{v \in V} M_{vn}(v,n)$

Table VI describes the SAT formulation. Since there is no objective, the formulation may yield longer routes, duplicate routes or routes with cycles. We post-process the routes to find the shortest between two nodes.