SNAFU: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture

Graham Gobieski   Ahmet Oguz Atli   Kenneth Mai   Brandon Lucia   Nathan Beckmann

gobieski@cmu.edu   aatli@andrew.cmu.edu   kenmai@ece.cmu.edu   blucia@andrew.cmu.edu   beckmann@cs.cmu.edu

Carnegie Mellon University

Abstract—**Ultra-low-power (ULP) devices are becoming pervasive, enabling many emerging sensing applications. Energy-efficiency is paramount in these applications, as efficiency determines device lifetime in battery-powered deployments and performance in energy-harvesting deployments. Unfortunately, existing designs fall short because ASICs' upfront costs are too high and prior ULP architectures are too inefficient or inflexible.**

We present SNAFU, the first framework to flexibly generate ULP coarse-grain reconfigurable arrays (CGRAs). SNAFU provides a standard interface for processing elements (PE), making it easy to integrate new types of PEs for new applications. Unlike prior high-performance, high-power CGRAs, SNAFU is designed from the ground up to minimize energy consumption while maximizing flexibility. SNAFU saves energy by configuring PEs and routers for a *single* operation to minimize switching activity; by minimizing buffering within the fabric; by implementing a statically routed, bufferless, multi-hop network; and by executing operations in-order to avoid expensive tag-token matching.

We further present SNAFU-ARCH, a complete ULP system that integrates an instantiation of the SNAFU fabric alongside a scalar RISC-V core and memory. We implement SNAFU in RTL and evaluate it on an industrial sub-28 nm FinFET process across a suite of common sensing benchmarks. SNAFU-ARCH operates at <1 mW, orders-of-magnitude less power than most prior CGRAs. SNAFU-ARCH uses 41% less energy and runs 4.4× faster than the prior state-of-the-art general-purpose ULP architecture. Moreover, we conduct three comprehensive case-studies to quantify the cost of programmability in SNAFU. We find that SNAFU-ARCH is close to ASIC designs built in the same technology, using just 2.6× more energy on average.

Index Terms—**Ultra-low power, energy-minimal design, reconfigurable computing, dataflow, CGRA, Internet of Things (IoT).**

I. INTRODUCTION

TINY, ultra-low-power (ULP) sensor devices are becoming increasingly pervasive, sophisticated, and important to a number of emerging application domains. These include environmental sensing, civil-infrastructure monitoring, and chip-scale satellites [69]. Communication consumes lots of energy in these applications, so there is a strong incentive to push ever-more computation onto the sensor device [22]. Unfortunately, widely available ULP computing platforms are fundamentally inefficient and needlessly limit applications. New architectures are needed with a strong focus on ULP (<1 mW), *energy-minimal* operation.

Sensing workloads are pervasive: The opportunity for tiny, ULP devices is enormous [41]. These types of embedded systems can be deployed to a wide range of environments, including harsh environments like the ocean or space [19]. Sensors on board these devices produce rich data sets that require sophisticated processing [45, 46]. Machine learning and advanced digital signal processing are becoming important tools for applications deployed on ULP sensor devices [22].

This increased need for processing is in tension with the ULP domain. The main constraint these systems face is *severely limited energy*, either due to small batteries or weak energy harvesting. One possible solution is to offload processing to a more powerful edge device. However, communication takes much more energy than local computation or storage [22, 40]. The only viable solution is therefore to process data locally and transmit only a minimum of filtered/preprocessed data, discarding the rest. This operating model has a major implication: the capability of future ULP embedded systems will depend largely on the energy-efficiency of the onboard compute resources.

Existing programmable ULP devices are too inefficient: Commercial-off-the-shelf (COTS) ULP devices are general-purpose and highly programmable, but they pay a high energy tax for this flexibility. Prior work has identified this failing of COTS devices and has addressed some of the sources of inefficiency [15, 23, 27, 47, 73]. Specifically, MANIC [23] targeted instruction and data-movement energy, the majority of wasted energy in COTS devices. MANIC is a big improvement over COTS devices, but we show that designs like MANIC still fall short due to high switching activity in the shared execution pipeline, which is a significant inefficiency at ULP-scale. Eliminating these overheads can reduce energy by nearly half, proving that, despite their low operating power, existing ULP designs are not energy-minimal.

ASICs can minimize energy, but they are too inflexible: For any application, a custom ASIC will minimize energy consumption. E.g., prior work has demonstrated extreme energy efficiency on neural networks when all hardware is specialized [7, 9, 38, 59]. But this efficiency comes at high upfront cost and with severely limited application scope. Applications in the ULP sensing domain are still evolving, increasing the risk that an ASIC will quickly become obsolete. Moreover, cost is a major consideration in these applications, making ASIC development even harder to justify [63].

Ultra-low-power CGRAs are the answer: The goal of this paper is to address the energy-efficiency shortcomings of prior designs while maintaining a high degree of design flexibility and ease of programmability. Our solution is SNAFU,[1]

---

[1]Simple Network of Arbitrary Functional Units.
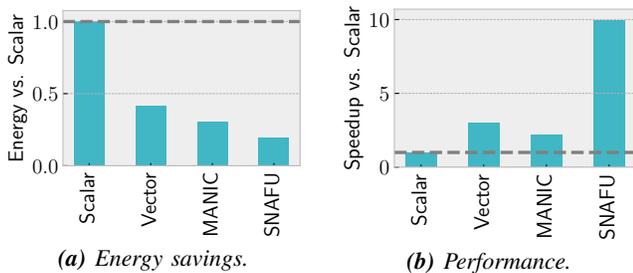
(a) Energy savings. *(b) Performance.*

Figure 1: SNAFU-ARCH's energy and performance normalized to a scalar baseline. On average, SNAFU uses 81% less energy and is 9.9× faster, or 41% less energy and 4.4× faster than MANIC.

a framework to generate ULP, energy-minimal coarse-grain reconfigurable arrays (CGRAs). SNAFU CGRAs execute in a *spatial vector-dataflow* fashion, mapping a dataflow graph (DFG) spatially across a fabric of processing elements (PEs), applying the same DFG to many input data values, and routing intermediate values directly from producers to consumers. The insight is that spatial vector-dataflow minimizes instruction and data-movement energy, just like MANIC, but also eliminates unnecessary switching activity because operations do not share execution hardware.

The major difference from most prior CGRAs [21, 24, 25, 33, 42, 51, 58, 62, 68, 71, 72, 74, 75] is the extreme design point — SNAFU operates at *orders-of-magnitude lower energy and power budget*, demanding an exclusive focus on energy-minimal design. SNAFU is designed from the ground up to minimize energy, even at the cost of area or performance. For example, SNAFU schedules only one operation per PE, which minimizes switching activity (energy) but increases the number of PEs needed (area). As a result of such design choices, SNAFU comes within 2.6× of ASIC energy efficiency while remaining fully programmable.

SNAFU generates ULP CGRAs from a high-level description of available PEs and the fabric topology. SNAFU defines a standard PE interface that lets designers *"bring your own function unit"* and easily integrate it into a ULP CGRA, along with a library of common PEs. The SNAFU framework schedules operation execution and routes intermediate values to dependent operations while consuming minimal energy. SNAFU is easy to use: it includes a compiler that maps vectorized C-code to efficient CGRA bitstreams, and it reduces design effort of tape-out via top-down synthesis of CGRAs.

Contributions: This paper contributes the following:

- We present SNAFU, the first flexible CGRA-generator for ULP, energy-minimal systems. SNAFU makes it easy to integrate new functional units, compile programs to energy-efficient bitstreams, and produce tape-out-ready hardware.
- We discuss the key design choices in SNAFU that minimize energy: scheduling at most one operation per PE; asynchronous dataflow without tag-token matching; statically routed, bufferless, multi-hop NoC; and producer-side buffering of intermediate values.
- We describe SNAFU-ARCH, a complete ULP system-on-chip with a CGRA fabric, RISC-V scalar core, and memory. We implement SNAFU-ARCH in an industrial sub-28 nm FinFET

process with compiled memories. SNAFU-ARCH operates at <1 mW at 50 MHz. SNAFU-ARCH reduces energy by 81% vs. a scalar core and 41% vs. MANIC; and improves performance by 9.9× vs. a scalar core and 4.4× vs. MANIC.
- Finally, we quantify the cost of programmability through three comprehensive case studies that compare SNAFU-ARCH against fixed-function ASIC designs. We find that programmability comes at relatively low cost: on average, SNAFU-ARCH takes 2.6× more energy and 2.1× more time than an ASIC for the same workload. We break down SNAFU-ARCH's energy in detail, showing that it is possible to close the gap further while retaining significant general-purpose programmability. These results call into question the need for extreme specialization in most ULP deployments.

Road map: Sec. II motivates SNAFU. Sec. III gives an overview of SNAFU, and Secs. IV, V, and VI describe it. Secs. VII and VIII present our evaluation methodology and results. Finally, Sec. IX compares SNAFU to ASICs, and Sec. X concludes.

II. BACKGROUND AND MOTIVATION

Ultra-low-power embedded system are constrained by their energy-efficiency, not raw performance. Existing commercial ULP platforms are not energy-efficient, and prior research designs still fall short. CGRAs offer a possible solution, but prior CGRAs either focus on high-performance or sacrifice design flexibility. SNAFU reconciles the demands for flexibility and energy efficiency, letting designers easily generate ULP CGRAs designed from the ground up to minimize energy.

A. Ultra-low-power embedded systems

Ultra-low-power embedded systems operate in a wide range of environments without access to the power grid. These devices rely on batteries and/or energy harvested from the environment to power their sensors, processors, and radios. Energy efficiency is the primary determinant of end-to-end system performance in these embedded systems.

Battery-powered: Efficiency ⇒ Lifetime: For battery-powered devices [14, 61], energy efficiency determines device lifetime: once a single-charge battery has been depleted the device is dead. Rechargeable batteries are limited in the number of recharge cycles, and even a simple data-logging application can wear out the battery in just a few years [32, 46].

Energy-harvesting: Efficiency ⇒ Performance: For energy-harvesting devices [12, 28, 29, 76, 77], energy efficiency determines device performance. These devices store energy in a capacitor and spend most of their time powered off, waiting for the capacitor to recharge. Greater energy efficiency leads to less time waiting and more time doing useful work [20].

Offloading is much less efficient than local processing: Often ULP embedded systems include low-power radios that can be used to transmit data for offloaded processing. Unfortunately, this is not an efficient use of energy by the ULP device [22, 40]. Communication over long distances bears a high energy and time cost. Instead, energy is better spent doing as much onboard computation as possible (e.g. on-device machine inference),

	Ultra-low-power CGRAs			High-performance CGRAs			
	ULP-SRP [34]	**CMA [55]**	**IPA [17]**	**HyCube [33]**	**Revel [75]**	**SGMF [71]**	**SNAFU**
Fabric size	3×3	8×10	4×4	4×4	5×5	8×8 + 32 mem	N×N (6×6 in SNAFU-ARCH)
NoC	Neighbors only	Neighbors only	Neighbors only	Static, bufferless, multi-hop	Static & dynamic NoCs (2×)	Dynamic routing	Static, bufferless, multi-hop
PE assignment	Static	Static	Static	Static	Static *or* dynamic	Dynamic	Static
Time-share PEs?	Yes	Yes	Yes	Yes	Yes	Yes	No
PE firing	Static	Static	Static	Static	Static *or* dynamic	Dynamic	Dynamic
Heterogeneous PEs?	No	No	No	No	Yes	Yes	Yes
Buffering (approx.)	—	—	188 B / PE	272 B / PE	≈1 KB / PE	≫1 KB / PE	40 B / PE
Power	22 mW	11 mW	3–5 mW	15–70 mW	160 mW	20 W	<1 mW
MOPS/mW (approx.)	30–100	100–200	140	60–90	60	60	305

Table I: Architectural comparison of SNAFU to several prior CGRAs.

and then relaying only the minimal amount of processed (e.g., filtered or compressed) data.

Our goal is to minimize energy: Thus, our overriding goal is to maximize end-to-end device capability by minimizing the energy of onboard compute. This goal is a big change from the typical goal of maximizing performance under a power or area envelope, and it leads SNAFU to a different design point that prioritizes energy efficiency over other metrics.

B. Energy-minimal design

Designing for energy-minimal, ULP operation is different than designing for other domains. This is partly because the ULP domain is at such a radically different scale that small changes have an outsized impact, but also because prioritizing energy over area and performance opens up new design tradeoffs. Unfortunately, existing ULP devices are *not* energy-minimal, and prior research has only begun to understand and address their sources of inefficiency.

COTS MCUs: Existing commercial ULP devices include ultra-low-power microcontrollers like the MSP430 [31] or Arm M0 [1]. These MCUs, while microarchitecturally very simple, are not efficient because they pay a high price for general programmability [23]. Energy is primarily wasted in supplying instructions (fetch, decode, control) and data (register file, caches), not performing useful work [30]. These overheads account for a majority of total system energy.

Reducing instruction-supply energy: Vector execution reduces instruction energy overhead by amortizing fetch, decode, and control over many operations. There is a long history of vector machines that span multiple computing domains [5, 8, 11, 13, 36, 37, 54]. Unfortunately, traditional vector designs use the large vector register file (VRF) to store intermediate results, exacerbating the energy overhead of supplying data.

Reducing data-supply energy: MANIC [23] proposed *vector-dataflow execution* to eliminate unnecessary VRF accesses. MANIC buffers a window of vector instructions and identifies how data flows between instructions. Next, MANIC iterates through instructions for each vector element (cf. iterating through vector elements for each instruction), bypassing the VRF to forward intermediate values between instructions. MANIC was designed as a simple modification to a scalar core in which all operations share an execution pipeline.

Unfortunately, sharing the pipeline significantly increases switching activity as data and control signals toggle between operations.

Designing for ULP: MANIC's vector-dataflow execution model is successful at reducing overall energy, but also serves as a good example for how the ULP domain is different. First, while VRF energy is significant, it is not as high as higher-level memory modeling tools report [57, 65]. The overestimate is likely due to the small size of ULP memories (a few KBs) being out-of-scope for these tools. To evaluate ULP energy-efficiency accurately, compiled memories are a must. Second, at ULP-scale where efficiency is measured in µW, even small amounts of switching in the pipeline logic is a significant cost — a sharp contrast with high-performance designs wherein logic is effectively free [16].

Designing to minimize energy: Even more fundamentally, energy-minimal designs can save energy by making tradeoffs that are unattractive in traditional designs. As explained in Sec. V, SNAFU realizes this opportunity primarily by trading area for energy: SNAFU-ARCH consumes 41% less energy than MANIC, but is 1.8× larger.

C. CGRA architectures

There is a rich literature on CGRA architectures. These architectures balance reconfigurability with energy efficiency and performance. However, most prior CGRAs target much higher power domains, and their design decisions do not translate well to the ULP domain. The few CGRAs targeting ULP operation (<1 mW) are not flexible and leave energy savings on the table.

What is a CGRA?: A coarse-grained reconfigurable array comprises a set of processing elements connected to each other via an on-chip network. These architectures are coarse in that the PEs support higher-level operations, like multiplication, on multi-bit data words, as opposed to bit-level configurability in FPGAs. They are also reconfigurable in that the PEs can often be configured to perform different operations and the NoC can be configured to route values directly between PEs. This lets applications map a dataflow graph onto the CGRA fabric, e.g., the body of a frequently executed loop. Many CGRAs also support SIMD operation, amortizing the cost of (re)configuration across many invocations. As a result, CGRAs can approach ASIC-like energy-efficiency and performance [52].
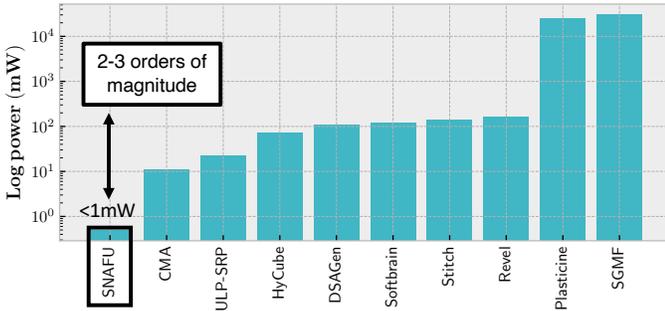
Figure 2: SNAFU generates ultra-low-power CGRA fabrics that operate in a power regime not well explored by existing CGRAs.

Figure 3: Overview of SNAFU. SNAFU is a flexible framework for generating ULP CGRAs. It takes a *bring-your-own functional unit* approach, allowing the designer to easily integrate custom logic tailored for specific domains.

Most CGRAs target a higher-power domain: Fig. 2 compares the operating power of SNAFU to several recent CGRAs. The difference is stark. A few high-performance designs operate at power comparable to a conventional CPU/GPU [58, 71]. Most CGRAs target a "low-power" regime at roughly 100 mW [33, 51, 68, 74, 75]. Even these designs are two orders-of-magnitude higher power than the ULP regime SNAFU targets.

Prior ultra-low-power CGRAs: There has been some work on ULP CGRAs in the CAD community [17, 34, 55]. Quantitative comparison is hard due to technology differences, and it is not always clear what is included in reported energy numbers. These designs use VLSI techniques to reduce power (e.g., low-voltage design, fine-grain clock/power gating) that are complementary to SNAFU. In contrast, our focus is *architecture* for flexibility and minimal energy. One of SNAFU's goal is to let designers generate a ULP CGRA at reduced VLSI effort.

Contrasting SNAFU with prior CGRAs: Table I compares SNAFU's design to prior CGRAs along several dimensions. Similar to DSAGEN [74] but unlike most prior work, SNAFU is a CGRA-*generator*, so fabric size is parameterizable. SNAFU minimizes PE energy by statically assigning operations to specific PEs and, unlike prior low-power CGRAs [17, 33, 34, 55, 68], minimizes switching by not sharing PEs between operations. Likewise, to minimize NoC energy, SNAFU implements a statically configured, bufferless, multi-hop NoC, similar to HyCube [33]. This NoC is a contrast with prior ULP CGRAs [17, 34, 55] that restrict communication to a PE's immediate neighbors. Unlike many prior CGRAs that are statically scheduled, SNAFU implements dynamic dataflow firing to support variable latency FUs. Dynamic dataflow firing is essential to SNAFU's flexibility and ability to support arbitrary, heterogeneous PEs in a single fabric. SNAFU avoids expensive tag-token matching [25, 58] by disallowing out-of-order execution, unlike high-performance designs [48, 56, 67, 71]. Finally, since buffers are more expensive than combinational logic, SNAFU minimizes buffering throughout the fabric, leading to much less state per PE than prior CGRAs.

These differences are discussed further in Sec. V. The takeaway is that, unlike prior work, SNAFU is consistently biased towards minimizing energy, even at the expense of area and performance. The end result is that SNAFU is flexible and general-purpose, while still achieving extremely low operating power and high energy-efficiency.
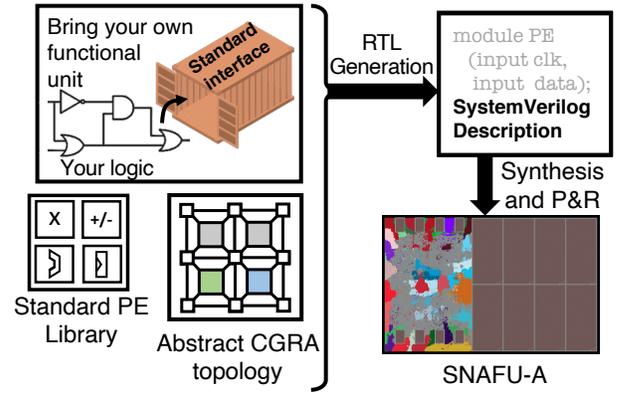
III. OVERVIEW

SNAFU is a framework for generating energy-minimal, ULP CGRAs and compiling applications to run efficiently on them. SNAFU-ARCH is a complete ULP system featuring a CGRA generated by SNAFU, a scalar core, and memory.

SNAFU is a flexible ULP CGRA generator: SNAFU is a general and flexible framework for converting a high-level description of a CGRA to valid RTL and ultimately to ULP hardware. Fig. 3 shows SNAFU's workflow. SNAFU takes two inputs: a library of processing elements (PEs) and a high-level description of the CGRA topology. SNAFU lets designers customize the ULP CGRA via a *"bring your own functional unit"* approach, defining a generic PE interface that makes it easy to add custom logic to a generated CGRA.

With these inputs, SNAFU generates complete RTL for the CGRA. This RTL includes a statically routed, bufferless, multi-hop on-chip network parameterized by the topology description. It also includes hardware to handle variable-latency timing and asynchronous dataflow firing. Finally, SNAFU simplifies hardware generation by supporting top-down synthesis, making it easy to go from a high-level CGRA description to a placed-and-routed ULP design ready for tape out.

SNAFU-ARCH is a complete ULP, CGRA-based system: SNAFU-ARCH is a specific, complete system implementation that includes a CGRA generated by SNAFU. The CGRA is a 6×6 mesh topology composed of PEs from SNAFU's standard PE library. SNAFU-ARCH integrates the CGRA fabric with a scalar core and 256 KB of on-chip SRAM main memory. The resulting system executes vectorized, RISC-V programs [60] with the generality of software and extremely low power consumption (≈300 μW). Compared to the RISC-V scalar core, SNAFU-ARCH uses 81% less energy for equal work and is 9.9× faster. Compared to MANIC (a state-of-the-art general-purpose ULP design), SNAFU-ARCH uses 41% less energy and is 4.4× faster. Compared to hand-coded ASICs, SNAFU-ARCH uses 2.6× more energy and is 2.1× slower.

Example of SNAFU in action: Fig. 4 shows the workflow to take a simple vectorized kernel and execute it on an ULP
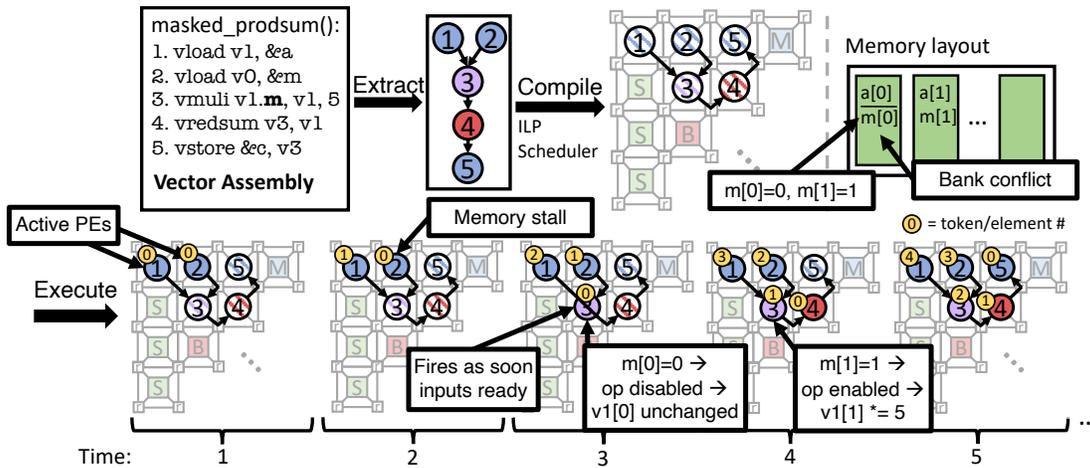
Figure 4: An example execution on a SNAFU CGRA fabric. The DFG is extracted from vectorized C-code, compiled to a bitstream, and then executed according to asynchronous dataflow firing.

CGRA generated by SNAFU. This kernel multiplies values at address &a by 5 for the elements where the mask m is set, sums the result, and stores it to address &c. SNAFU's compiler extracts the dataflow from the kernel source code and generates a bitstream to configure the CGRA fabric. The scalar core configures the CGRA fabric and kicks off fabric execution using three new instructions (vcfg, vtfr, vfence), after which the CGRA runs autonomously in SIMD fashion over arbitrarily many input data values. The fabric executes the kernel using asynchronous dataflow firing:

① In the first timestep, the two memory PEs (that load a[0] and m[0]) are enabled and issue loads. The rest of the fabric is idle because it has no valid input values.

② The load for a[0] completes, but m[0] cannot due to a bank conflict. This causes a stall, which is handled transparently by SNAFU's scheduling logic and bufferless NoC. Meanwhile, the load of a[1] begins.

③ As soon as the load for m[0] completes, the multiply operation can fire because both of its inputs have arrived. But m[0] == 0, meaning the multiply is disabled, so a[0] passes through transparently. The load of a[1] completes, and loads for a[2] and m[1] begin.

④ When the predicated multiply completes, its result is consumed by the fourth PE, which keeps a partial sum of the products. The preceding PEs continue executing in pipelined fashion, multiplying a[1] × 5 (since m[1] == 1) and loading a[3] and m[2].

⑤ Finally, a value arrives at the fifth PE, and is stored back to memory in c[0]. Execution continues in this fashion until all elements of a and m have been processed and a final result has been stored back to memory...

The next three sections describe SNAFU. Sec. IV describes the SNAFU ULP CGRA-generator. Sec. V describes how SNAFU minimizes energy. And Sec. VI describes SNAFU-ARCH.

IV. DESIGNING SNAFU TO MAXIMIZE FLEXIBILITY

SNAFU is designed to generate CGRAs that minimize energy, maximize extensibility, and simplify programming. For the architect, SNAFU automates synthesis from the top down
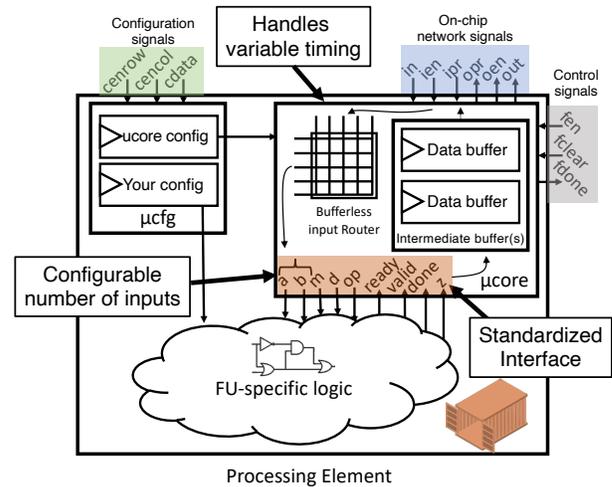

Figure 5: SNAFU provides a standardized interface that makes integrating new types of PEs trivial. SNAFU handles variable-latency logic, PE-specific configuration, and the sending and receiving of values from the on-chip network.

and provides a *"bring your own functional unit"* interface, allowing easy integration of custom FUs into a CGRA. For the application programmer, SNAFU is designed to efficiently support SIMD execution of vectorized RISC-V C-code, using a custom compiler that targets the generated CGRA.

A. Bring your own functional unit (BYOFU)

SNAFU has a generic PE microarchitecture that exposes a standard interface, enabling easy integration of custom functional units (FUs) into the CGRA. If a custom FU implements SNAFU's interface, then SNAFU generates hardware to automatically handle configuring the FU, tracking FU and overall CGRA progress, and moderating its communication with other PEs. There are few limitations on the sort of the logic that SNAFU can integrate. SNAFU's interface is designed to support variable latency and currently supports up to four inputs, but could be easily extended for more. The PE can have any number of additional ports and contain any amount of internal state.

Fig. 5 shows the microarchitecture of a generic SNAFU processing element, comprising two components: μcore and μcfg. The μcore handles progress tracking, predicated execution, and communication. The standard FU interface (highlighted orange) connects the μcore to the custom FU logic. The μcfg handles (re-)configuration of both the μcore and FUs.

Communication: The μcore handles communication between the processing element and the NoC, decoupling the NoC from the FU. The μcore is made up of an input router, logic that tracks when operands are ready, and a few buffers for intermediate values. The input router handles incoming connections, notifying the internal μcore logic of the availability of valid data and predicates. The intermediate buffers hold output data produced by the FU. Before an FU (that produces output) fires, the μcore first allocates space in the intermediate buffers. Then, when the FU completes, its output data is written to the allotted space, unless the predicate value is not set, in which case a fallback value is passed through (see below). Finally, the buffer is freed when all consumers have finished using the value. These intermediate buffers are the *only data buffering in the fabric*, outside of internal FU state. The NoC, which forwards data to dependent PEs, is entirely bufferless.

The FU interface: SNAFU uses a standard FU interface for interaction between a PE's μcore and FU. The interface has four control signals and several data signals. The four controls signals are op, ready, valid, and done; the μcore drives op and the FU is responsible for driving the latter three. op tells the FU that input operands are ready to be consumed. ready indicates that the FU can consume new operands. valid and done are related: valid says that the FU has data ready to send over the network, and done says the FU has completed execution. The remaining signals are data: incoming operands (a, b), predicate operands (m, d), and the FU's output (z).

The FU-μcore interface allows the μcore to handle variable-latency logic, making the FU's outputs available only when the FU completes an operation. The μcore raises back-pressure in the network when output from an FU is not ready and stalls the FU (by keeping op low) when input operands are not ready or there are no unallocated intermediate buffers. When the FU asserts both valid and done, the μcore forwards the value produced by the FU to dependent PEs via its NoC router.

Progress tracking and fabric control: The fabric has a top-level controller that interfaces with each μcore via three 1-bit signals. The first enables the μcore to begin execution, the second resets the μcore, and the third tells the controller when the PE has finished processing all input. The μcore keeps track of the progress of the FU by monitoring the done signal, counting how many elements the FU has processed. When the number of completed elements matches the length of the computation, the μcore signals the controller that it is done.

Predication: SNAFU supports conditional execution through built-in support for vector predication. The μcore delivers not only the predicate m, but also a fallback value d — for when the predicate is false — to the FU. When the predicate is true, the FU executes normally; when it is false, the FU is still triggered

so that it can update internal state (e.g., memory index for a strided load), but the fallback value is passed through.

Configuration services: The μcfg handles processing element configuration, setting up a PE's dataflow routes and providing custom FU configuration state. Router configuration maps inputs (a, b, m, d) to a router port. The μcfg forwards custom FU configuration directly to the FU, which SNAFU assumes handles its own internal configuration. The μcfg module contains a configuration cache that can hold up to six different configurations. The cached configurations reduce memory accesses and allow for fast switching between configurations. This improves both energy-efficiency and performance. It also benefits applications with dataflow graphs too large to fit onto the fabric. These applications split their dataflow graph into multiple sub-graphs. The CGRA executes them one at a time, efficiently switching between them via the configuration cache. Note, however, that even with the configuration cache, each fabric configuration is intended to be re-used across many input values before switching, unlike prior CGRAs that multiplex configurations cycle-by-cycle (Sec. II).

B. SNAFU's PE standard library

SNAFU includes a library of PEs that we developed using the BYOFU custom FU interface. The library includes four types of PEs: a basic ALU, multiplier, memory (load/store) unit, and scratchpad unit.

Arithmetic PEs: There are two arithmetic PEs: the basic ALU and the multiplier. The basic ALU performs bitwise operations, comparisons, additions, subtractions, and fixed-point clip operations. The multiplier performs 32-bit signed multiplication. Both units are equipped with the ability to accumulate partial results, like PE #4 (vredsum) in Fig. 4.

Memory PEs: The memory PEs generate addresses and issue loads and stores to global memory. The PE operates in two different modes, supporting strided access and indirect access. The memory PE also includes a "row buffer," which eliminates many subword accesses on accesses to a recently-loaded word.

Scratchpad PEs: A scratchpad holds intermediate values produced by the CGRA. The scratchpad is especially useful for holding data communicated between consecutive configurations of a CGRA, e.g., when the entire dataflow graph is too large for the CGRA. The PE connects to a 1 KB SRAM memory that supports stride-one and indirect accesses. Indirect access is used to implement permutation, allowing data to be written or read in a specified, permuted order.

C. Generating a CGRA fabric

Generating RTL: Given a collection of processing elements, SNAFU automatically generates a complete energy-minimal CGRA fabric. SNAFU ingests a high-level description of the CGRA topology and generates valid RTL. This high-level description includes a list of the processing elements, their types, and an adjacency matrix that encodes the NoC topology. With this high-level description, SNAFU generates an RTL header file. The file is used to parameterize a general RTL

description of a generic, energy-minimal CGRA fabric, which can then be fed through standard CAD tools.

NoC and router topology: SNAFU generates a NoC using a parameterized bufferless router model. The router can have any input and output radix and gracefully handles network back-pressure. Connections between inputs and outputs are configured statically for each configuration. Routers are mux-based because modern CAD tools optimize muxes well.

Top-down synthesis streamlines CAD flow: Following RTL generation, SNAFU fabrics can be synthesized through standard CAD tools from the top down without manual intervention. Top-down synthesis is important because SNAFU's bufferless, multi-hop NoC introduces combinational loops that normally require a labor-intensive, bottom-up approach to generate correct hardware. Industry CAD tools have difficulty analyzing and breaking combinational loops (i.e., by adding buffers to disable the loops). SNAFU leverages prior work on synthesizing FPGAs (which face the problem with combinational loops in their bufferless NoCs) from the top down to automate this process [39, 43]. SNAFU partitions connections between routers and PEs and uses timing case analysis to eliminate inconsequential timing arcs. SNAFU is the first framework for top-down synthesis of a CGRA, eliminating the manual effort of bottom-up synthesis.

D. Compilation

The final component is a compiler that targets the generated CGRA fabric. Fig. 4 shows the compilation flow from vectorized code to valid CGRA configuration bitstream. The compiler first extracts the dataflow graph from the vectorized C code. SNAFU asks the system designer (not the application programmer) to provide a mapping from RISC-V vector ISA instruction to a PE type, including the mapping of an operation's inputs and output onto an FU's inputs and output. This mapping lets SNAFU's compiler seamlessly support new types of PEs.

Integer linear program (ILP) scheduler: The compiler uses an integer linear program (ILP) formulation to schedule operations onto the PEs of a CGRA. The scheduler takes as input the extracted dataflow graph, the abstract instruction→PE map, and a description of the CGRA's network topology. The scheduler's ILP constraint formulation builds on prior work on scheduling code onto a CGRA [53]. The scheduler searches for subgraph isomorphisms between the extracted dataflow graph and the CGRA topology, minimizing the distance between spatially scheduled operations. At the same time, the ILP adheres to the mappings in the abstract instruction→PE map and does not map multiple dataflow nodes or edges to a single PE or route. To handle PEs that are shared across multiple fabric configurations (e.g., scratchpads holding intermediate data), programmers can annotate code with instruction *affinity*, which maps a particular instruction to a particular PE.

Scalability: Prior work has found scheduling onto a CGRA fabric to be extremely challenging and even intractable [18, 35, 50, 74], limiting compiler scalability to small kernels. However, this is not the case for SNAFU's compiler because SNAFU's

hardware makes compilation much easier: SNAFU supports asynchronous dataflow firing and does not time-multiplex PEs or routes. Together, these properties mean that the compiler need not reason about operation timing, making the search space much smaller and simplying its constraints. As a result, SNAFU's compiler can find an optimal solution in seconds even for the most complex kernels that we have evaluated.

Current limitations: If a kernel is too large to fit onto the CGRA or there is resource mismatch between the kernel and the fabric, the tool relies on the programmer to manually split the vectorized code into several smaller kernels that can be individually scheduled. This is a limitation of the current implementation, but not fundamental; a future version of the compiler will automate this process.

V. DESIGNING SNAFU TO MINIMIZE ENERGY

SNAFU's design departs from prior CGRAs because it is designed from the ground-up to *minimize energy*. This difference is essential for emerging ULP applications (Sec. II), and it motivates several key features of SNAFU's CGRA architecture. This section explores these differences and explains how they allow SNAFU to minimize energy.

A. Spatial vector-dataflow execution

Prior ULP systems: The state-of-the-art in ULP architecture is MANIC [23]. As discussed in Sec. II, MANIC introduces *vector-dataflow execution*, which amortizes instruction fetch, decode, and control (vector) and forwards intermediate values between instructions (dataflow). MANIC's vector-dataflow implementation parks intermediate values in a small "forwarding buffer," instead of the large vector register file (VRF).

MANIC reduces energy and adds negligible area, but its savings are limited by two low-level effects that only become apparent in a complete implementation. First, compiled SRAMs are cheaper and scale better than suggested by high-level architectural models [57, 65]; i.e., MANIC's savings from reducing VRF accesses are smaller than estimated. Second, MANIC multiplexes all instructions onto a shared execution pipeline, causing high switching activity in the pipeline logic and registers as control and data signals toggle cycle-to-cycle. Both effects limit MANIC's energy savings.

How SNAFU reduces energy: SNAFU reduces energy by implementing *spatial* vector-dataflow execution. Like vector-dataflow, SNAFU's CGRA amortizes a single fabric configuration across many computations (vector), and routes intermediate values directly between operations (dataflow). But SNAFU *spatially* implements vector-dataflow: SNAFU *buffers intermediate values locally* in each PE (vs. MANIC's shared forwarding buffer) and *each PE performs a single operation* (vs. MANIC's shared pipeline). Note that this design is also a contrast with prior CGRAs, which share PEs among multiple operations to increase performance and utilization (Table I).

As a result, SNAFU reduces both effects that limit MANIC's energy savings. We estimate that the reduction in switching activity accounts for the majority of the 41% of energy savings that SNAFU achieves vs. MANIC. The downside is that SNAFU

takes significantly more area than MANIC. This tradeoff is worthwhile because ULP systems are tiny and most area is memory and I/O (see Sec. VIII). SNAFU's leakage power is negligible despite its larger area because we use a high-threshold-voltage process.

B. Asynchronous dataflow firing without tag-token matching

The rest of this section discusses how SNAFU differs from prior CGRAs, starting with its dynamic dataflow firing.

Execution in prior CGRAs: Prior CGRAs have explored both static and dynamic strategies to assign operations to PEs and to schedule operations [75]. Static assignment and scheduling is most energy-efficient, whereas fully dynamic designs require expensive tag-matching hardware to associate operands with their operation. A static design is feasible when all operation latencies are known and a compiler can find an efficient global schedule. Static designs are thus common in CGRAs that do not directly interact with a memory hierarchy [25, 33, 51].

How SNAFU reduces energy: SNAFU is designed to easily integrate new FUs with unknown or variable latency. E.g., a memory PE may introduce variable latency due to bank conflicts. A fully static design is thus not well-suited to SNAFU, but SNAFU cannot afford full tag-token matching either.

SNAFU's solution is a hybrid CGRA with static PE assignment and dynamic scheduling. ("Ordered dataflow" in the taxonomy of prior work [75].) Each PE uses local, asynchronous dataflow firing to tolerate variable latency. SNAFU avoids tag-matching by enforcing that values arrive in-order. This design lets SNAFU integrate arbitrary FUs with little energy or area overhead, adding just $\approx 2\%$ system energy to SNAFU-ARCH. The cost of this design is some loss in performance vs. a fully dynamic CGRA. Moreover, asynchronous firing simplifies the compiler, as discussed above, because it is not responsible for operation timing.

C. Statically routed, bufferless on-chip network

NoCs in prior CGRAs: The on-chip network (NoC) can consume a large fraction of energy in high-performance CGRAs, e.g., more than 25% of fabric energy [33, 51]. Buffers in NoC routers are a major energy sink, and dynamic, packet-switched routers cause high switching activity. Prior ULP CGRAs avoid this cost with highly restrictive NoCs that limit flexibility [17, 34, 55].

How SNAFU reduces energy: SNAFU includes a statically-configured, bufferless, multi-hop on-chip network designed for high routability at minimal energy. Static circuit-switching eliminates expensive lookup tables and flow-control mechanisms, and prior work showed that such static routing does not degrade performance [33]. The network is bufferless (a PE buffers values it produces; see below), eliminating the NoC's primary energy sink (half of NoC energy or more [44]). As a result, SNAFU's NoC takes just $\approx 6\%$ of system energy.

D. Minimizing buffers in the fabric

Buffering of intermediate values in prior CGRAs: Prior CGRAs maximize performance by forwarding values to de-
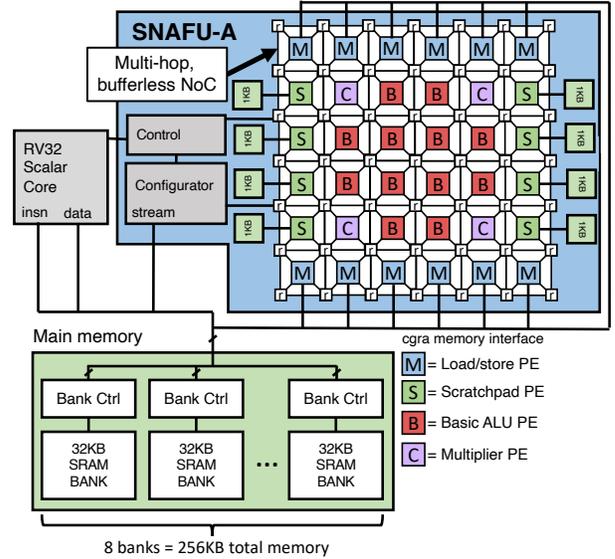

Figure 6: Architectural diagram of SNAFU-ARCH. SNAFU-ARCH possesses a RISC-V scalar core tightly coupled with the SNAFU fabric. Both are attached to a unified 256 KB banked memory.

pendent PEs and buffering them in large FIFOs, freeing a producer PE to start its next operation as early as possible. If a dependent PE is not ready, the NoC or dependent PE may buffer values. This approach maximizes parallelism, but duplicates intermediate values unnecessarily.

How SNAFU reduces energy: SNAFU includes minimal in-fabric buffering at the producer PE, with none in the NoC. Buffering at the producer PE means each value is buffered exactly once, and overwritten only when all dependent PEs are finished using it. In SNAFU-ARCH, producer-side buffering saves $\approx 7\%$ of system energy vs. consumer-side buffering. The cost is that a producer PE may stall if a dependent PE is not ready. SNAFU minimizes the number of buffers at each PE; using just four buffers per PE by default (Sec. VIII evaluates SNAFU's sensitivity to the number of buffers per PE).

VI. SNAFU-ARCH: A COMPLETE ULP SYSTEM W/ CGRA

SNAFU-ARCH is a complete ULP system that includes a CGRA fabric generated by SNAFU integrated with a scalar RISC-V core and memory.

A. Architectural overview

Fig. 6 shows an overview of the architecture of SNAFU-ARCH. There are three primary components: a RISC-V scalar core, a banked memory, and the SNAFU fabric. The SNAFU fabric is tightly coupled to the scalar core. It is a 6×6 mesh possessing 12 memory PEs, 12 basic-ALU PEs, 8 scratchpad PEs, and 4 multiplier PEs. The RTL for the fabric is generated using SNAFU and the mesh topology shown. The memory PEs connect to the banked memory, while the scratchpad PEs each connect to 1 KB outside the fabric.

The RISC-V scalar core implements the E, M, I, and C extensions and issues control signals to the SNAFU fabric. The banked memory has eight 32 KB memory banks (256 KB total). In total there are 15 ports to the banked memory: thirteen

Instruction	Purpose
`vcfg <len> <addr>`	Load a new fabric configuration and set vector length.
`vtfr <val> <pe>`	Communicate scalar value to fabric.
`vfence`	Start fabric execution and wait.

Table II: New instructions to interface with CGRA.

from the SNAFU fabric and two from the scalar core. The twelve memory PEs account for the majority of the ports from the fabric. The final port from the fabric allows the SNAFU configurator to load configuration bitstreams from memory. Each bank of the main memory can execute a single memory request at a time; its bank controller arbitrates requests using a round-robin policy to maintain fairness.

B. Example of SNAFU-ARCH in action

SNAFU-ARCH adds three instructions to the scalar core to interface with the CGRA fabric, summarized in Table II. We explain how they work through the following example.

The SNAFU fabric operates in three states: idle, configuration, and execution. During the idle phase the scalar core is running and the fabric is not. When the scalar core reaches a `vcfg` instruction, the fabric transitions to the configuration state. The scalar core passes a vector length and a bitstream address (from the register file) to the fabric configurator (see Fig. 6). The configurator checks to see if this configuration is still in the fabric's configuration cache (Sec. IV-A). If it is, the configurator broadcasts a control signal to all PEs and routers to load the cached configuration; otherwise, it loads the configuration header from memory. The header tells the configurator which routers and which PEs are active in the configuration. Then the configurator issues a series of loads to read in configuration bits for the enabled PEs and routers.

Once this has completed, the configurator stalls until the scalar core either reaches a `vtfr` instruction or a `vfence` instruction. `vtfr` lets the scalar core pass a register value to the fabric configurator, which then passes that value to a specific PE (encoded in the instruction). This allows PEs to be further parameterized at runtime from the scalar core. `vfence` indicates that configuration is done, so the scalar core stalls and the fabric transitions to execution. Execution proceeds until all PEs signal that they have completed their work (Sec. IV-A). Finally, the scalar core resumes execution from the `vfence`, and the fabric transitions back into the idle state.

VII. EXPERIMENTAL METHODOLOGY

We implemented SNAFU-ARCH as well as three baselines entirely in RTL and synthesized each system using an industrial sub-28 nm FinFET process with compiled memories. We evaluated the systems using post-synthesis timing, power, and energy models. Additionally, we placed and routed SNAFU-ARCH (see Fig. 7) to validate top-down synthesis.

Software-hardware stack: We developed a complete software and hardware stack for SNAFU. We implemented SNAFU-ARCH, its 256 KB banked memory, and its five-stage pipelined RISC-V scalar core in RTL and verified correctness by running full applications in simulation using both Verilator [66] and
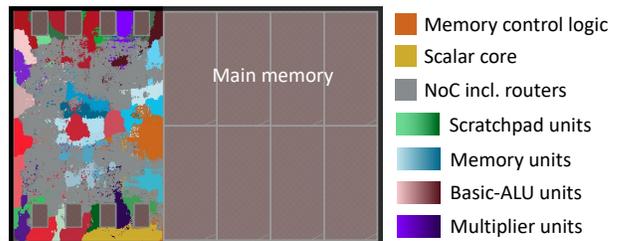

Figure 7: Layout of SNAFU-ARCH. Memory PEs are blue, scratchpad PEs are green, basic-ALU PEs are red, multiplier PEs are purple, the scalar core is yellow, and main-memory control logic is orange. The remaining grey regions contain routers and wires.

	Parameter	Values
	Frequency	50 MHz
	Main memory	256 KB
	Scalar register #	16
Vector	Vector register #	16
	Vector length	16/32/64
	Window size (for MANIC)	8
SNAFU-ARCH	Fabric dimensions	6×6
	Memory PE #	12
	Basic-ALU PE #	12
	Multiplier PE #	4
	Scratchpad PE #	8

Table III: Microarchitectural parameters.

Cadence Xcelium RTL simulator [6]. We synthesized the design using Cadence Genus [2] and an industrial sub-28 nm, high-threshold voltage FinFET PDK with compiled memories. Next, we placed and routed the design using Cadence Innovus [3]; Fig. 7 shows the layout. We also developed SNAFU's compiler that converts vectorized C-code to an optimal fabric configuration and injects the bitstream into the application binary. Finally, we simulated full applications post-synthesis, annotated switching, and used Cadence Joules [4] to estimate power.

Baselines: We compare SNAFU-ARCH against three baseline systems: *(i)* a RISC-V scalar core with a standard five-stage pipeline, *(ii)* a vector baseline that implements the RISC-V V vector extension, and *(iii)* MANIC [23], the prior state-of-the-art in general-purpose ULP design. The scalar core is representative of typical ULP microcontrollers like the TI MSP430 [31]. Each baseline is implemented entirely in RTL using the same design flow. Table III shows their microarchitectural parameters. The vector baseline and MANIC both have a single vector lane, which minimizes energy at the cost of performance.

Benchmarks: We evaluate SNAFU-ARCH, MANIC, and the vector baseline across ten benchmarks on three different input sizes, shown in Table IV. We use random inputs, generated offline. For MANIC and the vector baseline, each benchmark has been vectorized from a corresponding plain-C implementation. For SNAFU-ARCH, these vectorized benchmarks are fed into our compiler to produce CGRA-configuration bitstreams. In cases where the benchmarks contain a permutation, the kernel is manually split and pieces are individually fed into our compiler.

Metrics: We evaluate SNAFU-ARCH and the baselines primarily on their energy efficiency and secondarily on their performance.

9

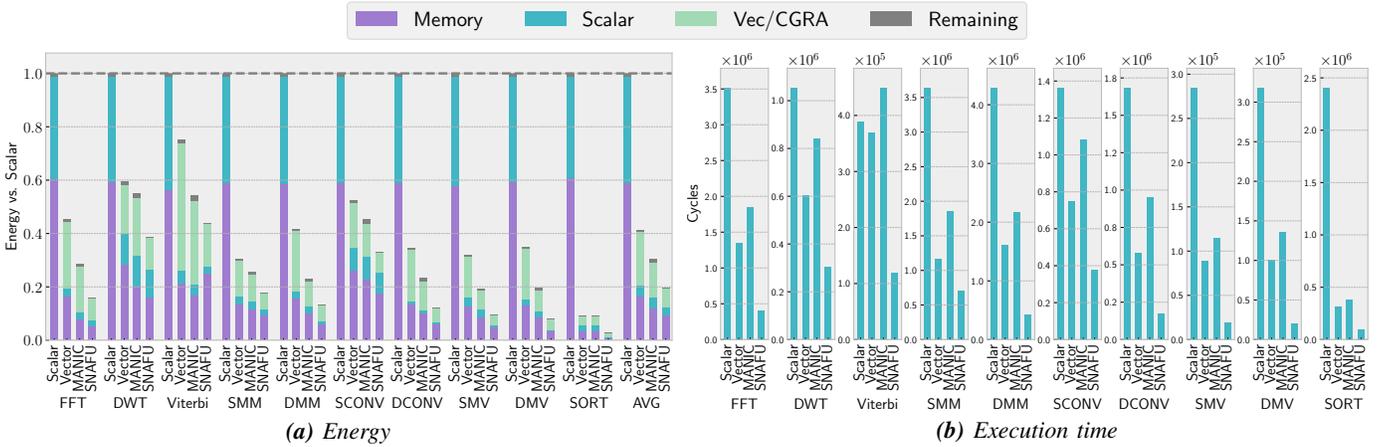*(a) Energy*   *(b) Execution time*

Figure 8: Energy and execution time, normalized to the scalar baseline, across ten applications on large inputs. On average, SNAFU-ARCH uses 81%, 57%, 41% less energy and is 9.9×, 3.2×, and 4.4× faster than the scalar design, vector baseline, and MANIC, respectively.

Name	Description	Small	Medium	Large
FFT	2D Fast-fourier transform	16×16	32×32	64×64
DWT	2D Discrete wavelet trnsfrm	16×16	32×32	64×64
Viterbi	Viterbi decoder	256	1024	4096
Sort	Radix sort	256	512	1024
SMM	Sparse matrix-matrix	16×16	32×32	64×64
DMM	Dense matrix-matrix	16×16	32×32	64×64
SMV	Sparse matrix-dense vector	32×32	64×64	128×128
DMV	Dense matrix-dense vector	32×32	64×64	128×128
Sconv	Sparse 2D convolution	16×16,	32×32,	64×64,
	filter:	3×3	5×5	5×5
Dconv	Dense 2D convolution	16×16,	32×32,	64×64,
	filter:	3×3	5×5	5×5

Table IV: Benchmarks and their input sizes.

We measure the full execution of each benchmark after initializing the system, and we measure efficiency by the energy to execute the complete benchmark normalized to either the scalar baseline or SNAFU-ARCH. We measure performance by execution time (cycles) or speedup normalized to either the scalar baseline or SNAFU-ARCH.

VIII. EVALUATION

We now evaluate SNAFU-ARCH to show: (1) SNAFU-ARCH is significantly more energy-efficient than the state-of-the-art. (2) Secondarily, SNAFU-ARCH significantly improves performance over the state-of-the-art. (3) SNAFU-ARCH is an optimal design point across input, configuration cache, and intermediate-buffer sizes. (4) SNAFU is easily extended with new PEs to improve efficiency. (5) Significant opportunities remain to improve efficiency in the compiler.

A. Main results

Fig. 8 shows that SNAFU-ARCH is much more energy-efficient and performant vs. all baselines. The figure shows average energy and speedup of SNAFU-ARCH normalized to the scalar baseline. SNAFU-ARCH uses 81%, 57%, and 41% less energy than the scalar, vector, and MANIC baselines, respectively. SNAFU-ARCH is also highly performant; it is 9.9×, 3.2×, and 4.4× faster than the respective baselines.

1) SNAFU-ARCH saves significant energy: Fig. 8 shows detailed results for all ten benchmarks. Fig. 8a breaks down ex-

ecution energy between memory, the scalar core, vector/CGRA, and remaining (other). SNAFU-ARCH outperforms all baselines on each benchmark. This is primarily because SNAFU-ARCH implements spatial vector-dataflow execution. Vector, MANIC, and SNAFU-ARCH all benefit from vector execution (SIMD), significantly improving energy-efficiency and performance compared to the scalar baseline by eliminating much of the overhead of instruction fetch and decode. However, SNAFU-ARCH benefits even more from vector execution because, once SNAFU-ARCH's fabric is configured, it can be re-used across an unlimited amount of data (unlike the limited vector length in the vector baseline and MANIC).

Moreover, only SNAFU-ARCH takes advantage of spatial dataflow. The vector baseline writes all intermediate results to the vector register file, which is quite costly. MANIC eliminates a majority of these VRF accesses (saving 27% energy compared to the vector baseline) by buffering intermediate values in a less expensive "forwarding buffer." However, MANIC shares a single execution pipeline across all instructions, which significantly increases switching activity. SNAFU-ARCH, on the other hand, executes a dataflow graph spatially. Each PE only handles a single operation and routes are configured statically. This leads to significantly less dynamic energy because intermediate values are directly communicated between dependent operations and there is minimal switching in PEs.

2) SNAFU-ARCH also greatly improves performance: Fig. 8b shows the execution time (in cycles) of all benchmarks and systems. Across the board, SNAFU-ARCH is faster — from 3.2× to 9.9× on average, depending on the baseline. SNAFU-ARCH achieves this high-performance by exploiting instruction-level parallelism in each kernel, which is naturally achieved by SNAFU's asynchronous dataflow-firing at each PE.

3) SNAFU-ARCH is ultra-low-power and has a small footprint: The SNAFU-ARCH fabric operates between 120 μW and 324 μW, depending on the workload, achieving an estimated 305 MOPS/mW. This operating power domain is two to five orders-of-magnitude less than most prior CGRA designs. Leakage power is also insignificant (<3%) because SNAFU-ARCH uses a high-threshold-voltage process.
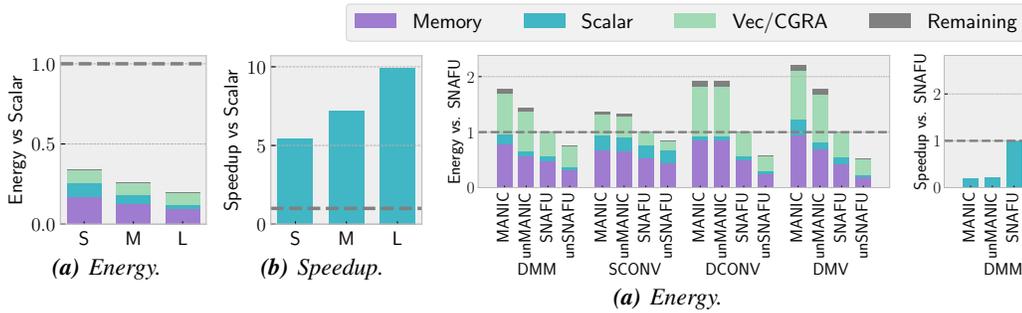
Figure 9: SNAFU-ARCH vs. the scalar design across three input sizes – small (S), medium (M) and large (L).

Figure 10: Energy and speedup of MANIC, unMANIC (w/ loop unrolling), SNAFU-ARCH, and unSNAFU-ARCH (w/ loop unrolling), normalized to SNAFU-ARCH. unSNAFU-ARCH uses 31% less energy and is 2.2× faster SNAFU-ARCH; MANIC benefits much less.

In addition, SNAFU-ARCH is tiny. The entire design in Fig. 7, including compiled memories, is substantially less than $1\,mm^2$. Note, however, that SNAFU-ARCH saves energy at the cost of area: SNAFU-ARCH occupies 1.8× more area than MANIC and 1.7× more than the vector baseline. Given SNAFU-ARCH's tiny size, we judge this to be a good tradeoff.

Benchmark analysis: SNAFU-ARCH is especially energy-efficient on dense linear algebra kernels and sort. SNAFU-ARCH uses on average 49% less energy on average for DMM, DMV, and DConv vs. 35% less on average for SMM, SMV, and SConv. This is because the dense linear algebra kernels take full advantage of coalescing in the memory PEs and generally have fewer bank conflicts, reducing energy and increasing performance (5.8× vs. 3.8×).

Sort is another interesting benchmark because MANIC barely outperforms the vector baseline, while SNAFU-ARCH reduces energy by 72%. (The scalar baseline performs terribly due to a lack of a good branch predictor.) This gap in energy can be attributed to the unlimited vector length of SNAFU-ARCH: the vector length of both vector and MANIC baselines is 64, but the input size to Sort is 1024. SNAFU-ARCH is able to sort the entire vector with minimal re-configuration and buffering of intermediate values.

B. Sensitivity studies

We characterize SNAFU-ARCH by running applications on three different input sizes. Further, we find the optimal configuration of SNAFU-ARCH by sweeping the size of the configuration cache and the number of intermediate buffers.

Energy-efficiency and performance improve on larger workloads: Fig. 9a shows SNAFU-ARCH's energy across three different input sizes: small (S), medium (M), and large (L). For most applications, SNAFU-ARCH's benefits increase with input size. (But SNAFU-ARCH is faster and more efficient at all input sizes.) As input size increases, SNAFU-ARCH generally widens the gap in energy-efficiency with the scalar baseline, from 67% to 81%. SNAFU-ARCH also improves vs. the vector baseline from 39% to 57% and vs. MANIC from 37% to 41% (not shown). The primary reason for this improvement is that, with larger input sizes, SNAFU-ARCH can more effectively amortize the overhead of (re)configuration.

This trend is even more pronounced in the performance data. Fig. 9b shows the speedup of SNAFU-ARCH normalized to the
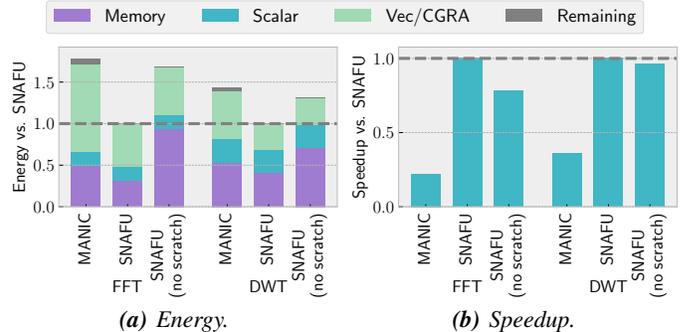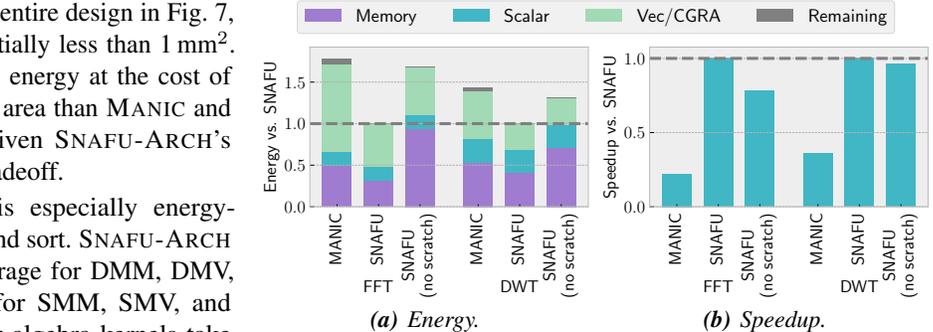

Figure 11: Energy and speedup of MANIC, SNAFU-ARCH w/ and w/out scratchpads, normalized to SNAFU-ARCH. Scratchpads improve energy-efficiency by 34% and performance by 13%.

scalar baseline. SNAFU-ARCH is 9.9×, 3.2×, 4.4× faster than the scalar baseline, vector baseline, and MANIC on the large input size and 5.4×, 2.4×, and 3.4× faster on the small input.

SNAFU's optimal parameterization: We considered designs with different configuration cache sizes (1, 2, 4, 6, and 8) and different numbers of intermediate buffers (1, 2, 4, and 8). For all applications except FFT, DWT, and Viterbi, configuration-cache size makes little difference. FFT, DWT, and Viterbi realize an average 10% energy savings with a size of six entries. This is because these applications have up to six phases of computation, and each phase requires a different fabric configuration. Similarly, most applications are insensitive to the number of intermediate buffers. With too few buffers, PEs stall due to lack of buffer space. Two buffers is enough to eliminate most of these stalls, and four buffers is optimal.

C. Case studies

We conduct two case studies *(i)* to show that there are opportunities to further improve performance and energy efficiency with only software changes; and *(ii)* to demonstrate the flexibility of SNAFU's BYOFU approach.

Loop-unrolling leads to significantly improved energy and performance: We show the potential of further compiler optimizations through a case study on loop unrolling. Fig. 10 shows the normalized energy and speedup of MANIC and SNAFU-ARCH with and without loop unrolling on four different applications. With loop unrolling, SNAFU-ARCH executes four iterations of an inner loop in parallel (vs. one iteration without
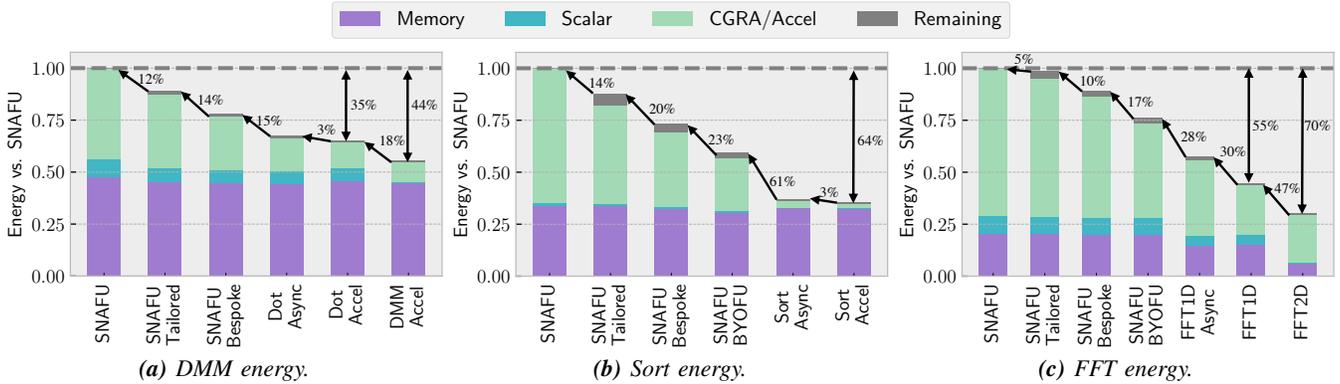
Figure 12: Quantifying the cost of SNAFU's programmability on three benchmarks. Each subfigure shows, from left-to-right, the effect of gradually removing SNAFU-ARCH's programmability and increasing its specialization, culminating in a hand-coded ASIC. Overall, SNAFU-ARCH's energy is within $2.6\times$ on average of the ASICs', and can be easily specialized to narrow the gap at low upfront design cost.

loop unrolling). On average, with loop unrolling, SNAFU-ARCH's energy efficiency improves by 85%, 71%, 62%, and 33% vs. scalar, vector, MANIC, and SNAFU-ARCH without unrolling. The performance results are even more significant: with loop unrolling, SNAFU-ARCH's speedup improves to $19\times$, $7.5\times$, $11\times$, and $2.2\times$ vs. the same set of baselines. These results make it clear that SNAFU-ARCH can effectively exploit instruction-level parallelism and that there is an opportunity for the compiler to further improve efficiency.

SNAFU makes it easy to add new FUs: Initially, SNAFU-ARCH did not have scratchpad PEs. However, FFT and DWT produce permuted results that must be persisted between reconfigurations of the fabric. Without scratchpad units, these values were being communicated through memory.

Leveraging SNAFU's standard PE interface, we were able to quickly add scratchpad PEs to SNAFU-ARCH with minimal effort — we just made SNAFU aware of the new PE, without *any* changes to SNAFU's framework. Fig. 11 shows the normalized energy and speedup of SNAFU-ARCH with and without scratchpads for FFT and DWT. Persisting intermediate values to main memory is quite expensive: without scratchpads, SNAFU-ARCH consumes 54% more energy and is 16% slower on average. The flexibility of SNAFU allowed us to easily optimize the SNAFU-ARCH fabric for FFT and DWT at low effort, without affecting other benchmarks. The next section explores the implications for programmability and specialization.

IX. THE COST OF PROGRAMMABILITY

With the mainstream acceptance of architectural specialization, the architecture community faces an ongoing debate over the ideal form and degree of specialization. Some results suggest a need for drastic specialization to compete with ASICs [26, 64, 70]; whereas others argue that programmable designs can compete by adopting a non-von Neumann execution model [49, 52].

We contribute to this debate by performing an apples-to-apples comparison of a programmable design (SNAFU-ARCH) against three hand-coded ASICs, demonstrating that the cost of programmability is low in the ULP domain. We compare end-

to-end systems in the same technology and design flow using an industrial PDK with compiled memories. Our results thus avoid pitfalls of prior studies based on simulations, analytical models, or extrapolations from different technologies.

We find that, on average, SNAFU-ARCH uses $2.6\times$ more energy and $2.1\times$ more time than an ASIC implementation. Breaking down the sources of inefficiency in SNAFU-ARCH, we find that SNAFU lets designers trade off programmability and efficiency via simple, incremental design changes. SNAFU makes *selective* specialization easy, letting the architect focus their design effort where it yields the most efficiency gains.

SNAFU is within $2.6\times$ of ASIC energy: Fig. 12 shows the energy of DMM, Sort, and FFT on large inputs. The leftmost bars in Fig. 12 represent SNAFU-ARCH and the rightmost bars represent a fixed-function, statically scheduled ASIC implementation of the same algorithm. SNAFU-ARCH uses as little as $1.8\times$ and on average $2.6\times$ more energy than the ASICs. To explain the gap, we now consider intermediate designs that build up from the ASIC back to SNAFU-ARCH.

SNAFU-ARCH, inner loops, & Amdahl's Law: SNAFU maps only inner loops to its fabric and runs outer loops on the scalar core, limiting its benefits to the fraction of dynamic execution spent in inner loops. To make a fair comparison, we built ASICs for DMM and FFT that accelerate only the inner-loop of the kernel (DOT-ACCEL and FFT1D-ACCEL), just like SNAFU-ARCH. These designs add 33% energy to run outer loops on the scalar core, reducing the energy gap to $2.2\times$ (vs. $2.5\times$ for these benchmarks previously). A future version of SNAFU could eliminate this extra scalar energy by mapping outer loops to its fabric [75].

Asynchronous dataflow firing adds minimal overhead: Next, we add asynchronous dataflow firing to the ASIC designs (∗-ASYNC bars). Comparing ASYNC designs to the ASICs shows that asynchronous dataflow firing adds little energy overhead, just 3% in DMM and Sort. The 30% overhead in FFT-ASYNC is inessential and could be optimized away: SNAFU's current implementation of asynchronous dataflow firing adds an unnecessary pipeline stage when reading scratchpad memories in the ASIC designs.

Closing the gap with negligible design effort: Next, we

12

consider variants of SNAFU-ARCH to break down the cost of software programmability. SNAFU-BESPOKE hardwires the fabric configuration, eliminating unused logic during synthesis (like prior work [10]), removing SNAFU-ARCH's software programmability. SNAFU-BESPOKE uses 54% more energy than the ASYNC designs. The gap in energy with ASYNC can be attributed to SNAFU's logic for predicated execution and the operation set that SNAFU implements, which is not well suited to every application. For instance, SORT-ACCEL can select bits directly, whereas SNAFU must do a vshift and vand.

BYOFU makes it easy to specialize where it matters: SNAFU's flexible, "bring your own functional unit" design (Sec. IV-A) makes it easy to add missing operations to improve efficiency. To illustrate this, Sort-BYOFU and FFT-BYOFU improve SNAFU-BESPOKE's efficiency by adding specialized PEs to the fabric. For Sort, we add a PE that fuses vshift and vand operations. For FFT, we size scratchpads properly for their data. In both cases, the energy savings are significant. SNAFU-BESPOKE uses 20% more energy than the BYOFU designs, and the BYOFU designs come within 44% of the ASYNC ASIC designs. These savings come with much lower design effort than a full ASIC, and we expect the gap would narrow further if more BYOFU PEs were added to the fabric.

The cost of software programmability is low: Next, SNAFU-TAILORED specializes the CGRA to eliminate extraneous PEs, routers, and NoC links, but is not hardwired like SNAFU-BESPOKE or SNAFU-BYOFU — i.e., SNAFU-TAILORED is where the design becomes programmable in software. SNAFU-TAILORED uses only 15% more energy than SNAFU-BESPOKE, illustrating that the cost of software programmability is low.

The original SNAFU-ARCH design uses just 10% more energy than SNAFU-TAILORED. This gap includes the cost of PEs, routers, and links that are not needed by every application, but may be used by some. This gap is also small, suggesting that *general-purpose* programmability also has a low cost.

The above comparisons yield three major takeaways. The big picture is that *the total cost of SNAFU's programmability is 2–3×* in energy and time vs. a fully specialized ASIC. While significant, this gap is much smaller than the $25\times$ (or larger) gap found in some prior studies [26].[2] Whether further specialization is worthwhile will depend on the application; for many applications, a 2–$3\times$ gap is good enough [63].

Moreover, for applications that benefit from more specialization, *SNAFU allows for selective specialization with incremental design effort* to trade off efficiency and programmability. Fig. 12 shows that by tailoring the fabric topology, hardwiring configuration state, or partially specializing PEs, designers can get within $2\times$ of ASIC efficiency at a small fraction of ASIC design effort. Designers can incrementally scale their design effort to the degree of specialization appropriate for their application.

Finally, digging deeper, we can better understand the cost of programmability as separate costs incurred at design time (i.e., hardware implementation quality) and run time (i.e., overhead for running software). With current synthesis tools, *software programmability itself is surprisingly cheap, but carries a hidden design-time cost:* the gap between SNAFU and SNAFU-BESPOKE is just 27%, whereas the gap between SNAFU-BESPOKE and ASICs is $2.1\times$. Even when runtime reconfigurability is removed from a design, synthesis tools cannot produce circuits similar to a hand-coded ASIC because they do not understand the intent of RTL. Barring a breakthrough in synthesis, this challenge will remain. SNAFU provides a path forward: BYOFU lets a designer specialize for critical operations, enabling programmable designs to compete with ASICs at a fraction of the design effort and without forfeiting programmability.

X. CONCLUSIONS

This paper presented SNAFU, a framework for generating ultra-low-power CGRAs. SNAFU maximizes flexibility while minimizing energy. It takes a *bring your own functional unit* approach, allowing easy integration of custom logic, and it minimizes energy by aggressively favoring efficiency over performance throughout the design. We used SNAFU to generate SNAFU-ARCH, a complete ULP CGRA system that uses 41% less energy and is $4.4\times$ faster than the prior state-of-the-art general-purpose ULP system. Moreover, SNAFU-ARCH is competitive with ASICs and can be incrementally specialized to trade off efficiency and programmability.

XI. ACKNOWLEDGMENTS

REFERENCES

[1] "Cortex-m0." [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0

[2] "Genus synthesis solution." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[3] "Innovus implementation system." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html

[4] "Joules rtl power simulation." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/power-analysis/joules-rtl-power-solution.html

[5] "Nvidia turing gpu architecture." [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[6] "Xcelium logic simulation." [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html

[7] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *ISVLSI*, 2016.

[8] K. Asanovic, J. Beck, B. Irissou, B. E. Kingsbury, and J. Wawrzynek, "T0: A single-chip vector microprocessor with reconfigurable pipelines," in *ESSCIRC 22*.

[9] K. Bong, S. Choi, C. Kim, S. Kang, Y. Kim, and H.-J. Yoo, "14.6 a 0.62 mw ultra-low-power convolutional-neural-network face-recognition processor and a cis integrated with always-on haar-like face detector," in *ISSCC*, 2017.

[10] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Bespoke processors for applications with ultra-low area and power constraints," in *ISCA 44*, 2017.

[11] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvptm)," in *ISCA 36*, 2003.

---

[2]The smaller gap comes from comparing to a programmable design that exploits *both* vector and dataflow techniques to improve efficiency [49, 52].

[12] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *ASPLOS 23*, 2018.

[13] Cray Computer, "U.S. Patent 6,665,774," December 2003.

[14] D. Culler, J. Hill, M. Horton, K. Pister, R. Szewczyk, and A. Wood, "Mica: The commercialization of microsensor motes," *Sensor Technology and Design, April*, 2002.

[15] 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.

[16] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, Jun. 2020.

[17] 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.

[18] S. Dave, M. Balasubramanian, and A. Shrivastava, "Ramp: Resource-aware mapping for cgras," in *DAC 55*, 2018.

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

[20] H. Desai and B. Lucia, "A power-aware heterogeneous architecture scaling model for energy-harvesting computers," *IEEE Computer Architecture Letters*, vol. 19, no. 1, 2020.

[21] 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.

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

[23] 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 52*, 2019.

[24] 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.

[25] 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.

[26] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[27] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," in *ISCA 32*, 2005.

[28] J. Hester, L. Sitanayah, and J. Sorber, "Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors," in *Proc. of the 13th ACM Conference on Embedded Networked Sensor Systems*, 2015.

[29] J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless internet of things," in *SenSys 15*, 2017.

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

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

[32] N. Jackson, "lab11/permamote," Apr 2019. [Online]. Available: https://github.com/lab11/permamote

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

[34] 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.

[35] M. Kou, J. Gu, S. Wei, H. Yao, and S. Yin, "Taem: fast transfer-aware effective loop mapping for heterogeneous resources on cgra," in *DAC 57*, 2020.

[36] C. Kozyrakis and D. Patterson, "Overcoming the limitations of conventional vector processors," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, 2003.

[37] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *ISCA 31*, 2004.

[38] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *ISSCC*, 2018.

[39] A. Li, T.-J. Chang, and D. Wentzlaff, "Automated design of fpgas facilitated by cycle-free routing," in *FPL 30*, 2020.

[40] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi, "Implementing software on resource-constrained mobile sensors: Experiences with impala and zebranet," in *MobiSys 2*. New York, NY, USA: ACM, 2004.

[41] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *SNAPL 2*, 2017.

[42] 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.

[43] P. Mohan, O. Atli, O. Kibar, M. Z. Vanaikar, L. Pileggi, and K. Mai, "Top-down physical design of soft embedded fpga fabrics," in *FPGA*. ACM, 2021.

[44] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *ISCA 36*, 2009.

[45] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith, "Towards battery-free {HD} video streaming," in *NSDI 15*, 2018.

[46] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, "Camaroptera: A batteryless long-range remote visual sensing system," in *ENSSys 7*, 2019.

[47] L. Nazhandali, B. Zhai, A. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin, and D. Blaauw, "Energy optimization of subthreshold-voltage sensor network processors," in *ISCA 32*, 2005.

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

[49] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *HPCA*, March 2016, pp. 27–39.

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

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

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

[53] 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.

[54] Nvidia, "Nividia jetson tx2," 2019. [Online]. Available: https://developer.nvidia.com/embedded/develop/hardware

[55] 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.

[56] 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.

[57] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *DATE*, 2015.

[58] 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.

[59] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA 43*, 2016.

[60] Riscv, "riscv-v-spec," Apr 2019. [Online]. Available: https://github.com/riscv/riscv-v-spec

[61] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. Moura, and L. Soibelman, "Sensor andrew: Large-scale campus-wide sensing and actuation," *IBM Journal of Research and Development*, vol. 55, no. 1.2, 2011.

[62] 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.

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

[64] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *ISCA 41*, 2014.

[65] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An Integrated Cache, Timing, Power, and Area Model," Compaq Western Research Laboratory, Tech. Rep., February 2001.

[66] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, DAC*, 2004.

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

[68] 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.

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

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

[71] 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.

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

[73] B. A. Warneke and K. S. Pister, "17.4 an ultra-low energy microcontroller for smart dust wireless sensor networks," 2004.

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

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

[76] A. Wickramasinghe, D. Ranasinghe, and A. Sample, "Windware: Supporting ubiquitous computing with passive sensor enabled rfid," in *RFID*, April 2014.

[77] H. Zhang, J. Gummeson, B. Ransford, and K. Fu, "Moo: A batteryless computational rfid and sensing platform," *Department of Computer Science, University of Massachusetts Amherst., Tech. Rep*, 2011.