US 20250258987A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2025/0258987 A1**

Lucia et al. (43) **Pub. Date: Aug. 14, 2025**

(54) **ULTRA-LOW POWER COURSE-GRAINED RECONFIGURABLE ARRAY FABRICS**

(71) Applicant: **CARNEGIE MELLON UNIVERSITY**, Pittsburgh, PA (US)

(72) Inventors: **Brandon Lucia**, Pittsburgh, PA (US); **Nathan Beckmann**, Pittsburgh, PA (US); **Graham Gobieski**, Pittsburgh, PA (US)

(73) Assignee: **CARNEGIE MELLON UNIVERSITY**, Pittsburgh, PA (US)

(21) Appl. No.: **19/171,591**

(22) Filed: **Apr. 7, 2025**

Related U.S. Application Data

(63) Continuation of application No. 17/572,925, filed on Jan. 11, 2022, now Pat. No. 12,271,671.

(60) Provisional application No. 63/228,184, filed on Aug. 2, 2021, provisional application No. 63/143,061, filed on Jan. 29, 2021.

Publication Classification

(51) **Int. Cl.**

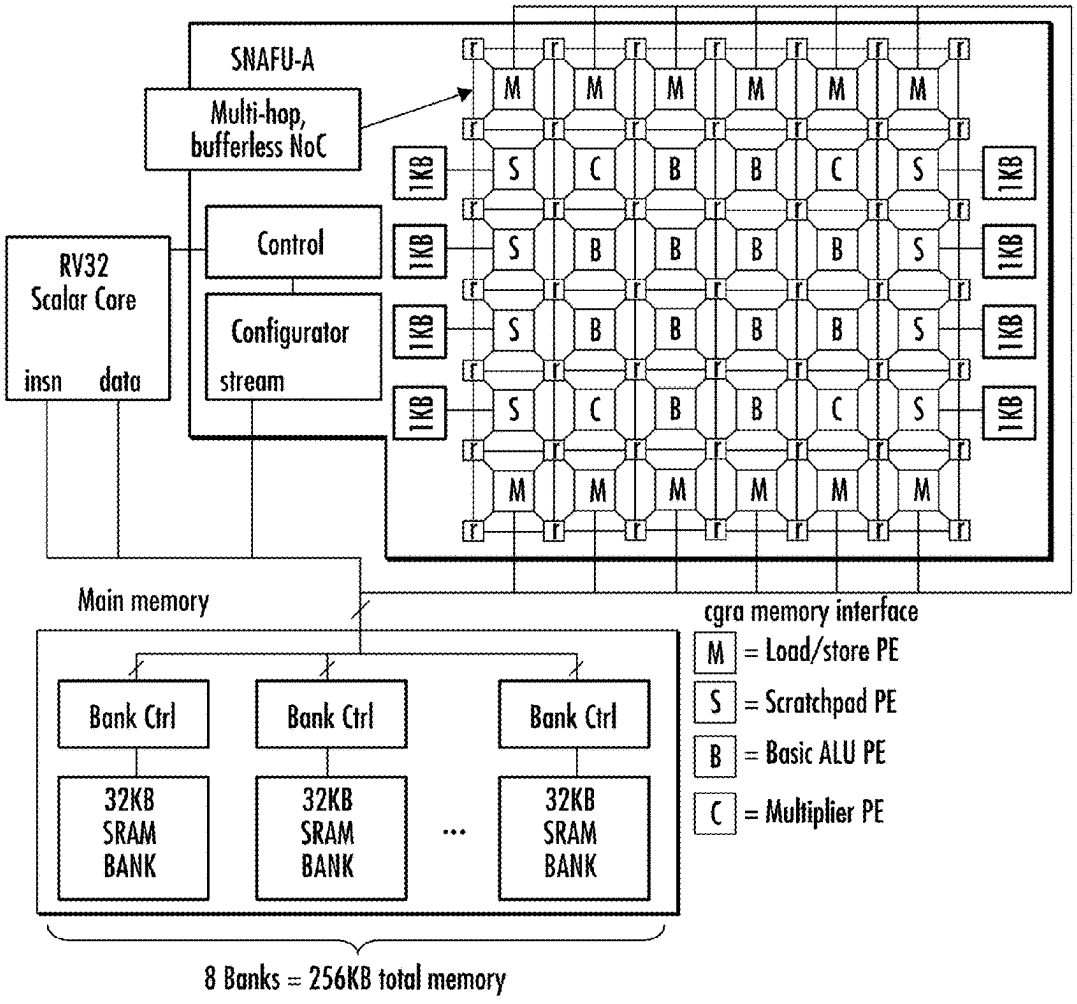
G06F 30/34	(2020.01)
G06F 15/78	(2006.01)
G06F 30/327	(2020.01)
G06F 111/16	(2020.01)
G06F 115/02	(2020.01)

(52) **U.S. Cl.**

CPC ***G06F 30/34*** (2020.01); ***G06F 30/327*** (2020.01); *G06F 15/7825* (2013.01); *G06F 2111/16* (2020.01); *G06F 2115/02* (2020.01)

(57) **ABSTRACT**

Disclosed herein is an energy-minimal coarse-grain reconfigurable fabric that executes in a spatial vector-dataflow fashion, mapping a dataflow graph spatially across a fabric of processing elements, applying the same dataflow graph to many input data values, and routing intermediate values directly from producers to consumers. The spatial vector-dataflow minimizes instruction and data-movement energy and also eliminates unnecessary a switching activity because operations do not share execution hardware.



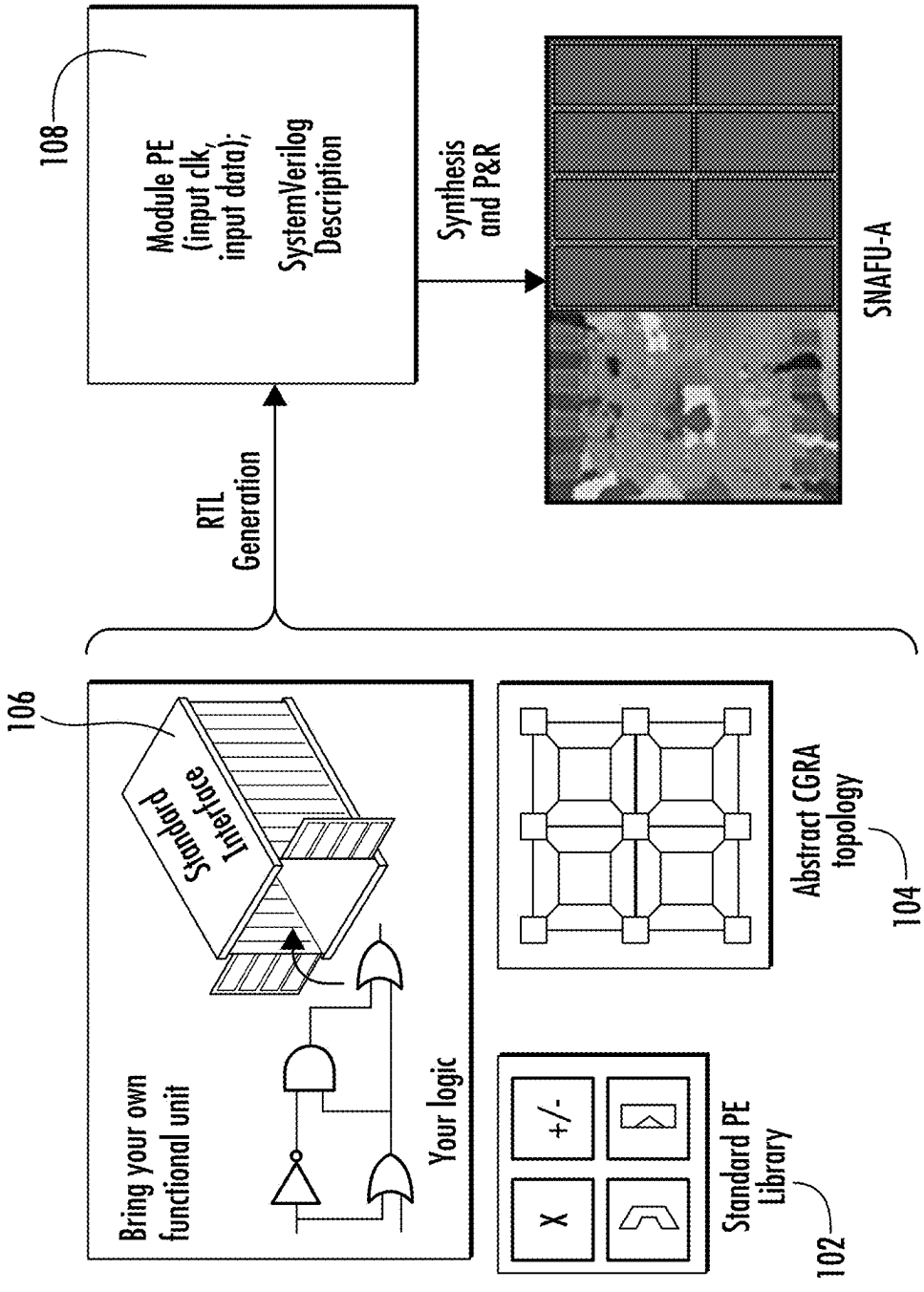

108
Module PE
(input clk,
input data);
SystemVerilog
Description
Synthesis
and P&R
SNAFU-A
RTL
Generation
106
Standard
Interface
Bring your own
functional unit
Your logic
Abstract CGRA
topology
104
X
+/-
Standard PE
Library
102

FIG. 1

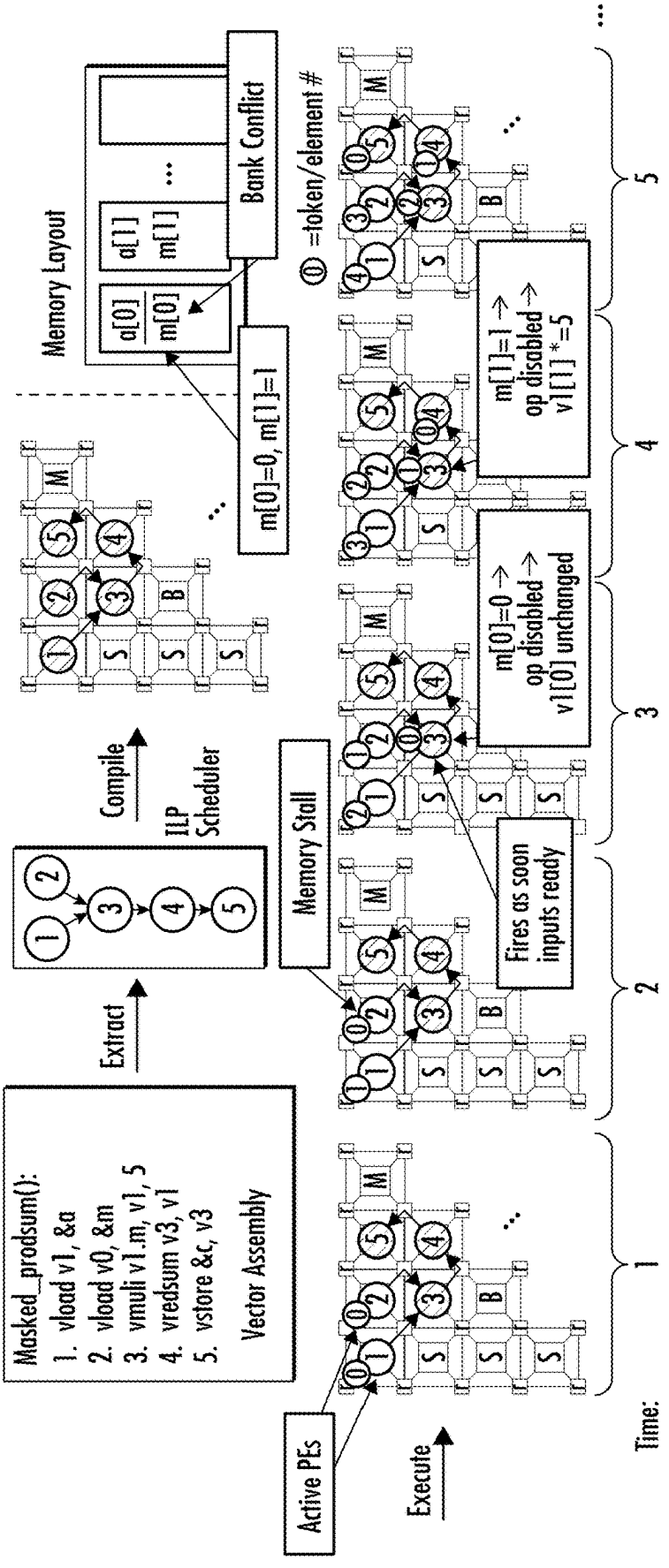

Masked_prodsum():
1. vload v1, &a
2. vload v0, &m
3. vmuli v1.m, v1, 5
4. vredsum v3, v1
5. vstore &c, v3
Vector Assembly
Extract
ILP Scheduler
Compile
Memory Layout
a[0] m[0]
a[1] m[1]
Bank Conflict
m[0]=0, m[1]=1
=token/element #
Memory Stall
Fires as soon inputs ready
m[0]=0 → op disabled → v1[0] unchanged
m[1]=1 → op disabled → v1[1]*=5
Active PEs
Execute
Time:
1
2
3
4
5

FIG. 2

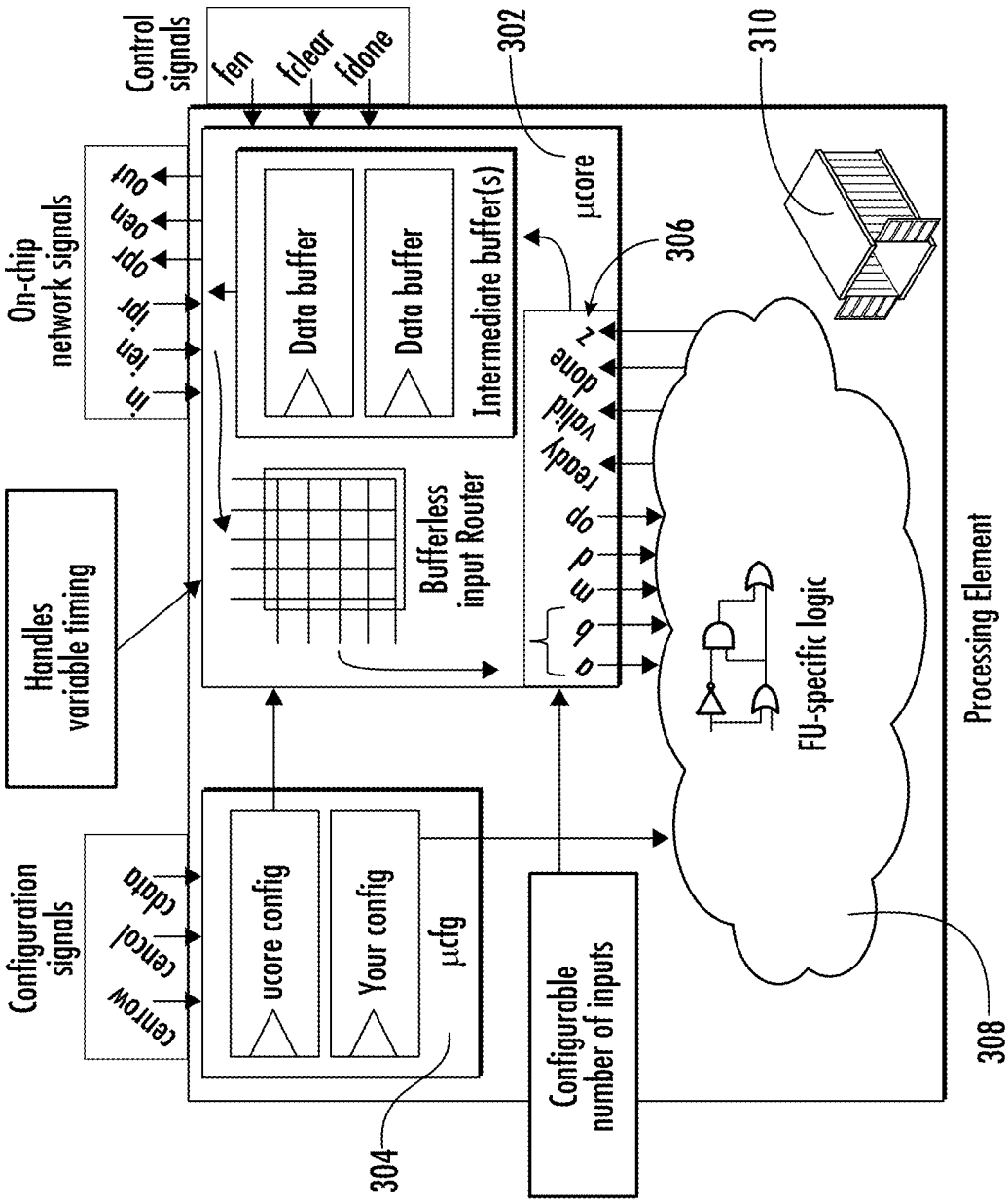

Control signals
fen
fclear
fdone
302
310
On-chip network signals
out
oen
opr
ipr
ien
in
Data buffer
Data buffer
Intermediate buffer(s)
µcore
306
Bufferless input Router
Handles variable timing
ready
valid
done
z
op
p
m
q
o
FU-specific logic
Processing Element
Configuration signals
cenrow
cencol
cdata
ucore config
Your config
µcfg
Configurable number of inputs
304
308

FIG. 3

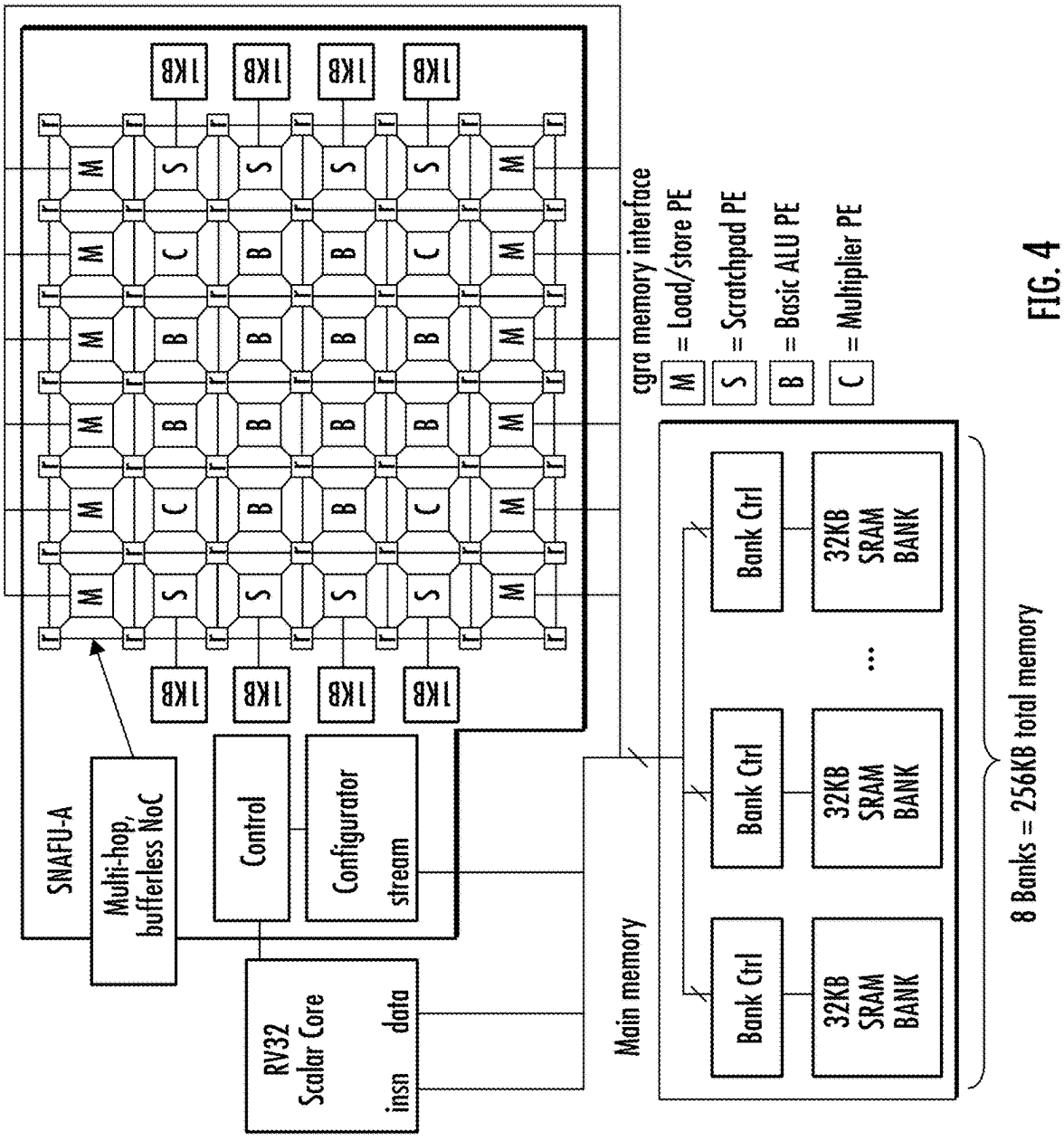

SNAFU-A
Multi-hop, bufferless NoC
Control
Configurator
stream
RV32 Scalar Core
insn
data
Main memory
cgra memory interface
M = Load/store PE
S = Scratchpad PE
B = Basic ALU PE
C = Multiplier PE
1KB
Bank Ctrl
32KB SRAM BANK
8 Banks = 256KB total memory

FIG. 4

ULTRA-LOW POWER COURSE-GRAINED RECONFIGURABLE ARRAY FABRICS

RELATED APPLICATIONS

[0001] This application is a continuation of U.S. patent application Ser. No. 17/572,925, filed Jan. 11, 2022, which claims the benefit of U.S. Provisional Patent Applications Nos. 63/143,061, filed Jan. 29, 2021, and 63/228,184, filed Aug. 2, 2021. The contents of these applications are incorporated herein in their entireties.

GOVERNMENT INTEREST

[0002] This invention was made with the United States government support under CCF1815882 awarded by the National Science Foundation. The U.S. government has certain rights in the invention.

BACKGROUND

[0003] 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.

[0004] 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. Communication is a large part of the energy budget in such applications, so there is a strong incentive to push computation onto the sensor device.

[0005] The potential uses for tiny ULP devices is enormous. These types of embedded systems can be deployed to a wide range of environments, including harsh environments like the ocean or space. Sensors on board these devices produce rich data sets that require sophisticated processing. Machine learning and advanced digital signal processing are becoming important tools for applications deployed on ULP sensor devices.

[0006] 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. 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 computing resources.

[0007] For battery-powered devices, 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 exhaust a battery in just a few years.

[0008] For energy-harvesting devices, energy efficiency determines device performance. These devices generate energy and store the energy in a capacitor. Such devices spend most of their time powered off, waiting for the capacitor to recharge.

[0009] Often, ULP embedded systems include low-power radios that can be used to transmit data for off-board processing. However, this is not an efficient use of energy by the ULP device. 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), and then relaying only the minimal amount of processed (e.g., filtered or compressed) data.

[0010] Commercial-off-the-shelf (COTS) ULP devices are general-purpose and highly programmable, but this flexibility comes with an associated high energy cost. Instruction and data-movement energy accounts for the majority of wasted energy in COTS devices. Eliminating these overheads can reduce energy requirements by nearly half, proving that, despite their low operating power, existing ULP designs are not energy-minimal.

[0011] For any application, a custom ASIC can minimize energy consumption. For example, extreme energy efficiency on neural networks can be achieved when all hardware is specialized. However, 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.

[0012] 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 can result in a large 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. Widely available ULP computing platforms are fundamentally inefficient and needlessly limit applications. Therefore, new architectures are needed with a strong focus on ULP (e.g., <1 mW), energy-minimal operation.

SUMMARY

[0013] The disclosed invention addresses the energy efficiency shortcomings of prior designs while maintaining a high degree of design flexibility and ease of programmability. The solution of the disclosed invention is a simple network of arbitrary functional units (referred to herein as "SNAFU"), which acts as 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 spatial vector-dataflow minimizes instruction and data-movement energy and also eliminates unnecessary switching activity because operations do not share execution hardware.

[0014] A CGRA 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 on-chip network ("Network-on-Chip", referred to herein as "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.

[0015] The major difference between SNAFU and most prior art CGRAs 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× ASIC energy efficiency while remaining fully programmable.

[0016] 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 functional 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 includes a compiler that maps vectorized C-code to efficient CGRA bitstreams and reduces design effort of tape-out via top-down synthesis of CGRAs.

[0017] SNAFU is 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. Key design choices in SNAFU to minimize energy include: 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.

[0018] Also included herein is an exemplary description of a complete ULP system-on-chip with a CGRA fabric, RISC-V scalar core, and memory (referred to herein as SNAFU-ARCH). SNAFU-ARCH is implemented in an industrial sub-28 nm FinFET process with compiled memories. SNAFU-ARCH operates at <1 mW at 50 MHz, reduces energy by 81% versus a scalar core and 41% vs. MANIC (an architecture using the vector-dataflow execution model) and improves performance by 9.9× versus a scalar core and 4.4× vs. MANIC.

[0019] Energy-minimal designs can save energy by making tradeoffs that are unattractive in traditional designs. SNAFU realizes this opportunity primarily by trading area for energy: SNAFU-ARCH consumes 41% less energy than MANIC but is 1.8× larger. Thus, the overriding goal of this invention is to maximize end-to-end device capability by minimizing the energy of onboard computing. 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.

[0020] 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.

BRIEF DESCRIPTION OF THE DRAWINGS

[0021] FIG. **1** is a block diagram showing the workflow of SNAFU, showing the "bring your own functional unit" approach which allows easy integration of custom logic tailored for specific domains.

[0022] FIG. **2** is a block diagram showing an exemplary execution on a SNAFU CGRA fabric. The DFG is extracted from vectorized C-code, compiled to a bitstream and executed according to asynchronous dataflow firing.

[0023] FIG. **3** is a block diagram showing the microarchitecture of a generic SNAFU processing element, comprising two components: μcore and μcfg.

[0024] FIG. **4** is a block diagram of an exemplary embodiment of a CGRA implemented using SNAFU.

DETAILED DESCRIPTION

[0025] SNAFU is a general and flexible framework for converting a high-level description of a CGRA to valid register transfer logic (RTL) and ultimately to ULP hardware. FIG. **1** shows SNAFU's workflow. SNAFU takes two inputs: a library of processing elements (PEs) **102** and a high-level description of the CGRA topology **104**. SNAFU lets designers customize the ULP CGRA via a "bring your own functional unit" approach, defining a generic PE interface **106** that makes it easy to add custom logic to a generated CGRA.

[0026] With these inputs, SNAFU generates complete RTL **108** for the CGRA. RTL **108** includes a statically routed, bufferless, multi-hop, on-chip network parameterized by the topology description **104**. 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.

[0027] FIG. **2** shows the workflow to take a simple vectorized kernel and execute it on a ULP 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:

[0028] In timestep 1, 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.

[0029] In timestep 2, the load for a[0] completes, but the load for m[0] cannot complete 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.

[0030] In timestep 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.

[0031] In timestep 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 (because m[1]==1) and loading a[3] and m[2].

[0032] In timestep 5, 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.

Goal: Maximizing Flexibility

[0033] SNAFU is designed to generate CGRAs that minimize energy, maximize extensibility, and simplify programming. For the architect, SNAFU automates synthesis from the top down and provides a "bring your own functional unit" interface, allowing easy integration of custom functional units (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.

[0034] Bring Your Own Functional Units—SNAFU has a generic PE microarchitecture that exposes a standard interface, enabling easy integration of custom 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 sorts of logic that SNAFU can integrate. SNAFU's interface is designed to support variable latency and currently supports up to four inputs but could easily be extended to support more inputs. The PE can have any number of additional ports and contain any amount of internal state.

[0035] FIG. **3** shows the microarchitecture of a generic SNAFU processing element, comprising two components: μcore **302** and μcfg **304**. The μcore **302** handles progress tracking, predicated execution, and communication. The standard FU interface **306** connects the μcore **302** to the custom FU logic **308**. The μcfg **304** handles (re-)configuration of both the μcore **302** and FU **308***s*.

[0036] The μcore **302** handles communication between the processing element and the NoC, decoupling the NoC from the FU **308**. The μcore **302** 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 logic of μcore **302** of the availability of valid data and predicates. The intermediate buffers hold output data produced by the FU **308**. Before an FU **308** (that produces output) fires, the μcore **302** first allocates space in the intermediate buffers. Then, when the FU **308** 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 (as described 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.

[0037] SNAFU uses a standardized FU interface **310** for interaction between a PE's μcore **302** and FU **308**. The FU interface **306** has four control signals and several data signals. The four controls signals are op, ready, valid and done. The μcore **302** drives op and FU **308** is responsible for driving the latter three. op tells FU **308** that input operands are ready to be consumed. The ready signal indicates that FU **308** can consume new operands. The valid and done signals are related: The valid signal indicates that FU **308** has data ready to send over the network, and the done signal indicates that FU **308** has completed execution. The remaining signals are data: incoming operands (a & b), predicate operands (m & d), and the output (z) of FU **308**.

[0038] The interface between FU **308** and μcore **302** allows μcore **302** to handle variable-latency logic, making the output of FU **308** available only when FU **308** completes an operation. μcore **302** raises back-pressure in the network when output from an FU is not ready and stalls the FU (by keeping the op signal low) when input operands are not ready or when there are no unallocated intermediate buffers. When FU **308** asserts both the valid and done signals, μcore **302** forwards the value produced by FU **308** to dependent PEs via its NoC router.

[0039] The fabric has a top-level controller that interfaces with each μcore **302** via three 1-bit signals. The first enables μcore **302** to begin execution, the second resets μcore **302**, and the third tells the controller when the PE has finished processing all input. μcore **302** keeps track of the progress of FU **308** by monitoring the done signal and counting how many elements FU **308** has processed. When the number of completed elements matches the length of the computation, μcore **302** signals the controller that it is done.

[0040] SNAFU supports conditional execution through built-in support for vector predication. μcore **302** delivers to FU **108** not only the predicate m, but also a fallback value d which is delivered when the predicate is false. When the predicate is true, FU **308** executes normally; when it is false, FU **308** is still triggered so that it can update its internal state (e.g., memory index for a strided load), but the fallback value is passed through.

[0041] μcfg **306** handles processing element configuration, setting up a PE's dataflow routes and providing a custom FU configuration state. Router configuration maps inputs (a, b, m & d) to a router port. μcfg **306** forwards the custom FU configuration directly to FU **308**, which SNAFU assumes handles its own internal configuration. The μcfg **306** 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 subgraphs. 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).

[0042] PE Library—SNAFU includes a library of PEs that we developed using the custom FU interface. The library includes four types of PEs: a basic ALU, multiplier, memory (load/store) unit, and scratchpad unit.

[0043] There are two types of 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. **2**.

[0044] Memory PEs generate addresses and issue loads and stores to global memory. The memory 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.

[0045] A scratchpad PE holds intermediate values produced by the CGRA. The scratchpad PE 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 scratchpad 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.

[0046] Generating a CGRA Fabric—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.

[0047] 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 multiplexors.

[0048] 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 synthesizes CGRAs (which face the problem with combinational loops in their bufferless NoCs) from the top down to automate this process. 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.

[0049] Compilation—The final component is a compiler that targets the generated CGRA fabric. FIG. **2** 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.

[0050] 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. 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.

[0051] Scheduling onto a CGRA fabric is an extremely challenging problem, 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 simplifying 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.

[0052] 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.

Goal: Minimizing Energy

[0053] 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, 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.

[0054] Spatial Vector-Dataflow Execution—The state-of-the-art in ULP architecture is MANIC. As discussed herein 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).

[0055] 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. (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.

[0056] 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.

[0057] As a result, SNAFU reduces both effects that limit MANIC's energy savings. 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 occupied by memory and I/O. SNAFU's leakage power is negligible despite its larger area because we use a high-threshold-voltage process.

[0058] 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.

[0059] Prior CGRAs have explored both static and dynamic strategies to assign operations to PEs and to schedule operations. 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.

[0060] 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 the art). 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 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.

[0061] Statically Routed, Bufferless On-Chip Network—The on-chip network (NoC) can consume a large fraction of energy in high-performance CGRAs (e.g., more than 25% of fabric energy). 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.

[0062] 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. Such static routing does not degrade performance. The network is bufferless (a PE buffers values it produces, eliminating the NoC's primary energy sink (e.g., half of NoC energy or more). As a result, SNAFU's NoC takes just ≈6% of system energy.

[0063] Minimizing Buffers in the Fabric—Buffering of intermediate values in prior CGRAs: Prior CGRAs maximize performance by forwarding values to dependent 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.

[0064] 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 7% of system energy versus 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.

SNAFU-ARCH: A Complete ULP System Using CGRA

[0065] SNAFU-ARCH is a complete ULP system that includes a CGRA fabric generated by SNAFU integrated with a scalar RISC-V core and memory.

[0066] Architectural Overview-FIG. **4** 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.

[0067] 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 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 between requests using a round-robin policy to maintain fairness, so that all requests are eventually serviced.

[0068] Example of SNAFU-ARCH in Action—SNAFU-ARCH adds three instructions to the scalar core to interface with the CGRA fabric. The following example explains how they work.

[0069] 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 μcfg 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, as shown in FIG. **4**. The configurator checks to see if this configuration is still in the fabric's configuration cache. 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.

[0070] Once this has completed, the configurator stalls until the scalar core either reaches a vtfr instruction or a vfence instruction. The vtfr instruction 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. Finally, the scalar core resumes execution from the vfence, and the fabric transitions back into the idle state.

[0071] A framework for generating ultra-low-power CGRAs has been presented herein. 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. SNAFU was used to generate SNAFU-ARCH, a complete ULP CGRA system that uses ~41% less energy and is ~4.4 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.

1. An ultra-low-power course-grain reconfigurable array fabric comprising:
- a plurality of processing elements;
- a bufferless on-chip network interconnecting each of the plurality of processing elements; and
- a top-level controller for controlling each processing element.

2. The fabric of claim **1** wherein each processing element comprises:
- a functional unit defining the function of the processing element;
- an interface module for handling communication between the functional unit and the network; and
- a configuration module;
- wherein a standard interface is defined between the functional unit and the interface module.

3. The fabric of claim **2** wherein the interface module comprises:
- an input router for handling incoming connections and tracking availability of valid data and predicates;
- logic for tracking when operands are ready; and
- a plurality of buffers for holding intermediate values.

4. The fabric of claim **3** wherein the plurality of buffers hold output data produced by the functional unit until needed by another processing element.

5. The fabric of claim **3** wherein the configuration module performs the functions of:
- configuring the interface module and the functional unit of the processing element;
- setting up dataflow routes for the processing element; and
- providing a custom configuration state for the functional unit.

6. The fabric of claim **5** wherein the configuration unit maps inputs of the standard interface to a port on the input router.

7. The fabric of claim **5** wherein the functional unit handles its configuration internally based on the configuration state.

8. The fabric of claim **5** wherein the configuration module configuration module performs the further function of:
- reconfiguring the processing element to one of a plurality of configurations held in a configuration cache.

9. The fabric of claim **2** wherein the standard interface comprises:
- one or more inputs to and at least one output from the functional unit;
- an op signal, controlled by the interface module informing the functional unit that input operands are ready to be consumed;
- a ready signal, controlled by the functional units, informing the interface module that the functional unit can consume new operands;
- a valid signal, controlled by the functional unit, indicating that the functional unit has data ready to be sent over the network; and
- a done signal, controlled by the functional unit, indicating that the functional unit has completed execution.

10. The fabric of claim **2** wherein the interface module forwards outputs of the functional unit to other, dependent processing elements when the functional unit asserts both the valid and done signals.

11. The fabric of claim **10** wherein the inputs and outputs of the functional unit comprise
- a predicate value; and
- a fallback value;
- wherein, when the interface module asserts the predicate, the functional unit is triggered and executes normally; and
- wherein, when the predicate is not asserted, the functional unit is triggered, but returns the fallback value as an output.

12. The fabric of claim **2** wherein the functional unit is of a type selected from a group consisting of:
- a memory type to generate addresses and issue loads and stores to a global memory;
- a scratchpad type to hold intermediate values produced by the fabric;
- an ALU type to perform bitwise operations, comparisons, additions, subtractions, and fixed-point clip operations;
- a multiplier type to perform 32-bit signed multiplication; and
- a user-define type to implement user specified functionality.

13. The fabric of claim **2** wherein the top-level controller performs the functions of:
- informing an individual the processing element to begin execution;
- resetting an individual processing element; and
- receiving a signal from an individual processing element indicating that the processing element has completed processing of all inputs.

14. An ultra-low power system comprising:
- the fabric of claim **2**;
- a processor; and
- a memory bank.

15. The system of claim **14** wherein the processor is a scaler RISC-V core.

16. The system of claim **14** wherein the processor issues control signals to the fabric.

17. The system of claim **14** wherein the memory is coupled to processing elements within the fabric whose functional units are configured to interface with the memory.

18. The system of claim **14** wherein the fabric operates in one of three states:
- an idle state in which the processor is running and the fabric is not;
- a configuration state in which the processor sends configuration to the fabric; and
- a running state in which the fabric is executing and the processor is idle.

* * * * *