Canalis: A Throughput-Optimized Framework for Real-Time Stream Processing of Wireless Communication

KUAN-YU CHEN, THOMAS MASON NELSON, ALIREZA KHADEM, MORTEZA FAYAZI, SANJAY SRI VALLABH SINGAPURAM, RONALD DRESLINSKI, NISHIL TALATI, HUN-SEOK KIM, and DAVID BLAAUW, University of Michigan, Ann Arbor, MI, USA

Stream processing, which involves real-time computation of data as it is created or received, is vital for various applications, specifically wireless communication. The evolving protocols, the requirement for high-throughput, and the challenges of handling diverse processing patterns make it demanding. Traditional platforms grapple with meeting real-time throughput and latency requirements due to large data volume, sequential and indeterministic data arrival, and variable data rates, leading to inefficiencies in memory access and parallel processing. We present Canalis, a throughput-optimized framework designed to address these challenges, ensuring high-performance while achieving low energy consumption. Canalis is a hardware-software co-designed system. It includes a programmable spatial architecture, Flux Stream Processing Unit (FluxSPU), proposed by this work to enhance data throughput and energy efficiency. FluxSPU is accompanied by a software stack that eases the programming process. We evaluated Canalis with eight distinct benchmarks. When compared to CPU and GPU in mobile SoC to demonstrate the effectiveness of domain specialization, Canalis achieves an average speedup of 13.4× and 6.6×, and energy savings of 189.8× and 283.9×, respectively. In contrast to equivalent ASICs of the benchmarks, the average energy overhead of Canalis is within 2.4×, successfully maintaining generalizations without incurring significant overhead.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; *Multiple instruction, multiple data*; **Systolic arrays**; *Multicore architectures*; **System on a chip**; *Pipeline computing*; • **Hardware** → **Digital signal processing**; **Application specific instruction set processors**; **Application specific processors**; **Hardware-software codesign**;

Additional Key Words and Phrases: Wireless Communication, Stream Processing, Throughput-Optimized, Software Stack

Kuan-Yu Chen was with University of Michigan, Ann Arbor, MI 48109, USA when he contributed to this work. He is now with Tenstorrent USA, Inc., Austin, TX 78735, USA.
Authors' Contact Information: Kuan-Yu Chen (corresponding author), University of Michigan, Ann Arbor, MI, USA: e-mail: knyuchen@umich.edu; Thomas Mason Nelson, University of Michigan, Ann Arbor, MI, USA: e-mail: nelsontm@umich.edu; Alireza Khadem, University of Michigan, Ann Arbor, MI, USA: e-mail: arkhadem@umich.edu; Morteza Fayazi, University of Michigan, Ann Arbor, MI, USA: e-mail: fayazi@umich.edu; Sanjay Sri Vallabh Singapuram, University of Michigan, Ann Arbor, MI, USA: e-mail: singam@umich.edu; Ronald Dreslinski, University of Michigan, Ann Arbor, MI, USA: e-mail: rdreslin@umich.edu; Nishil Talati, University of Michigan, Ann Arbor, MI, USA: e-mail: talatin@umich.edu; Hun-Seok Kim, University of Michigan, Ann Arbor, MI, USA: e-mail: hunseok@umich.edu; David Blaauw, University of Michigan, Ann Arbor, MI, USA; e-mail: blaauw@umich.edu.

ACM 1936-7414/2024/11-ART61
https://doi.org/10.1145/3695880

ACM Reference format:
Kuan-Yu Chen, Thomas Mason Nelson, Alireza Khadem, Morteza Fayazi, Sanjay Sri Vallabh Singapuram, Ronald Dreslinski, Nishil Talati, Hun-Seok Kim, and David Blaauw. 2024. Canalis: A Throughput-Optimized Framework for Real-Time Stream Processing of Wireless Communication. *ACM Trans. Reconfig. Technol. Syst.* 17, 4, Article 61 (November 2024), 32 pages.
https://doi.org/10.1145/3695880

1 Introduction

The rapid development of **Internet of Things (IoT)**, autonomous vehicles, and artificial intelligence increases the demand for efficient and effective stream processing in embedded applications. It is particularly crucial in the wireless communication domain due to the real-time constraints and the massive volume of data involved. Existing conventional compute platforms and programming languages cannot fully address the demands. The challenges include poor support for buffer-free stream processing, and difficulty executing workloads with diverse compute patterns efficiently.

Wireless communication workloads possess unique characteristics and constraints and require high-performance, flexible computing platforms. Protocols are continuously evolving and diverse, each involving different computational kernels that consist of compute-intensive loops exhibiting diverse processing patterns in data access and computation.

Given these challenges, spatial architecture emerges as a promising solution. Spatial architectures connect an array of **processing elements (PEs)** through on-chip networks, delivering **data-level parallelism (DLP)** and high data reuse, thus minimizing memory access. However, the stream processing model presents its unique set of challenges, such as managing real-time sequential data arrival, indeterministic data availability, and variable data rates.

Previous works have made substantial progress on mapping wireless communication kernels and workloads on spatial architectures, as well as developing spatial architectures with complete software stacks for various domains. Yuan et al. [104] and REVEL [94] execute single kernels efficiently using different techniques but do not demonstrate complete workloads with multiple kernels. Yuan et al. [104] utilize the reconfiguration of functional units while REVEL [94] focuses on executing inductive matrix algorithms with a hybrid systolic array accompanied by a compiler. Other works focus on mapping the entire workloads. AsAP [8, 102, 103] and Troung et al. [88] map workloads onto the computational fabric, employing asynchronous communication between cores. Tran et al. [87] demonstrate the benefits of connecting a programmable spatial architecture with configurable accelerators. Meanwhile, DAP [17] provides fast reconfigurability in reprogramming speed. However, the works discussed above, excluding REVEL [94], lack a software stack, hindering programmability and the efficient mapping of workloads to the architectures.

In other application domains, spatial architectures with software stacks have become increasingly popular. HyCube [45] provides single-cycle communication between distant functional units using a reconfigurable interconnect, and Ultra elastic CGRA [86] optimizes irregular loops using Dynamic Voltage Frequency Scaling. Riptide [34] aims for minimum energy consumption with tag-less dataflow and embedded control flow in a Network on Chip; DRIPs [84] map entire pipelined streaming workloads onto spatial architecture, enabling dynamic balancing of resource allocation; REVAMP [10] focuses on transforming homogeneous spatial architectures into heterogeneous ones based on application characteristics. Many of these works fall short in addressing stream processing, typically buffering data in the memory system before processing. This load/store assist model leads to high and irregular output latencies.

Goal and Approach. Our goal is to design a throughput-optimized framework incorporating a novel architecture with a co-designed software stack. The goal is to achieve the performance and

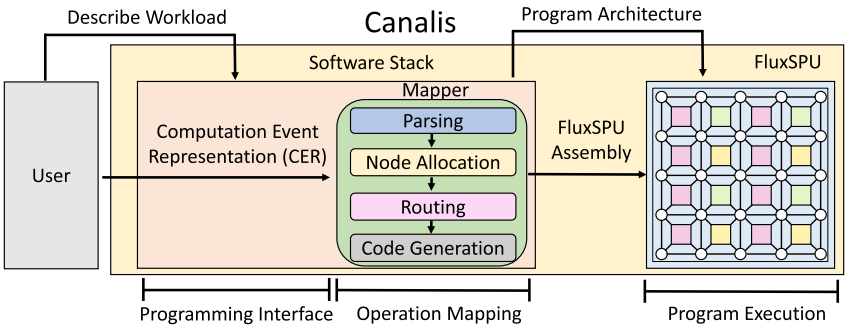


Fig. 1. Canalis is a throughput-optimized framework for real-time stream processing of wireless communication workloads. It includes a heterogeneous programmable spatial architecture, FluxSPU, and a co-designed software stack, that assists the user in programming FluxSPU through the CER programming interface and transform it into FluxSPU machine code. CER, Computation Event Representation; FluxSPU, Flux Stream Processing Unit.

efficiency of an **Application-Specific Integrated Circuits (ASICs)** implementation, yet retain the flexibility of programmable systems, merging the best attributes of both. Our framework, Canalis (Figure 1), is specifically designed and optimized to improve the throughput of real-time stream processing of wireless communication workloads that have little or no control flow, as shown in our provided examples and benchmarks in later sections. Canalis is designed and implemented with the following approaches:

First, we analyze the challenges of stream processing by contrasting the direct execution model with the conventional load/store assist model. In the direct execution model, data are not buffered within the memory system, which avoids costly memory accesses associated with latency and power consumption. Such a data processing strategy proves significantly more efficient for stream processing in the wireless communication domain, where the need for memory access is minimal.

Second, after mapping wireless communication workloads onto spatial architectures, we identify a common structural pattern in each processing node and introduce a domain-specific **Instruction Set Architecture (ISA)**, **Flux Stream Processing Unit (FluxSPU)** ISA (Section 4). This ISA, co-designed with microarchitectural optimizations (Section 5), specifically enhances the proposed architecture's ability to efficiently handle loop structures and reduces the **initiation interval (II)** and significantly increases the throughput of each processing node.

Third, we connect the nodes to form the FluxSPU fabric (Section 6), a **Multiple Instruction Multiple Data (MIMD)** spatial architecture, that resembles vector chaining [76] and results in higher utilization of functional units per cycle. FluxSPU utilizes the direct execution model and eliminates the need for global memory, replacing it with versatile queue structures between PEs.

Lastly, to effectively program FluxSPU, we develop a co-designed software stack (Section 7) including an intuitive programming interface, **Computation Event Representation (CER)**, and a mapper to map and translate CER into FluxSPU machine code. CER simplifies programming of FluxSPU by providing a simple syntax that reflects the characteristics of the architecture and dataflow. *CER does not function as a hardware generator;* its sole purpose is to assist the user in programming the FluxSPU architecture without diving into the details of mapping operations and applying optimizations provided by the ISA.

Results. Canalis is implemented in Register Transfer Level and a commercial 28 nm standard cell library. It is evaluated with eight distinct benchmarks, including complete workloads of Bluetooth and WiFi applications. These workloads incorporate multiple kernels executing concurrently and

seamlessly pipelined. We compare Canalis against several baselines: a mobile CPU, a mobile **Graphics Processing Unit (GPU)**, and equivalent ASIC implementations for each benchmark. Compared to the former two, Canalis achieves an average speedup of 13.4× and 6.6×, area normalized speedup of 29.6× and 130.5×, and energy savings of 189.8× and 283.9×, respectively, due to the parallelism achieved by MIMD processing. When compared to equivalent ASIC implementations, Canalis is within 2.4× of energy efficiency.

Key contributions of this work include:

(1) Analysis of the challenges associated with executing wireless communication workloads and stream processing on conventional compute models and identifying inefficiencies.
(2) Identification of a common program structure in processing nodes of spatial architectures after mapping wireless communication workloads, coupled with proposed optimizations across the new FluxSPU ISA and microarchitecture specifically tailored to target these structures.
(3) Proposal of the FluxSPU ISA, an instruction set optimized to enhance loop operations and reduce the II for individual nodes within the FluxSPU fabric. This includes a detailed quantification of the effectiveness of these optimizations with examples.
(4) Design of the FluxSPU fabric, a spatial architecture crafted to support real-time stream processing of wireless communication workloads.
(5) Development of a co-designed software stack that includes a programming interface, CER, specifically designed to describe stream processing workflows. This interface, along with a mapper, efficiently maps computations onto the FluxSPU.
(6) Implementation of the Canalis framework (including architecture and software stack) and comparisons against a mobile CPU, a mobile GPU, and equivalent ASIC implementations.

2 Background

Real-time streaming wireless communication workloads exhibit unique characteristics and constraints. They require specialized computing platforms that are both high-performance and adaptable. This section explores the aspects of the target application domain and the potential for programmable spatial architecture as a solution.

2.1 Wireless Communication Workload Characteristics

Wireless communication features constantly evolving protocols, distinctive computational requirements, and a wide range of applications from IoT devices to radar systems. The advent of 6G and other emerging technologies necessitates computational platforms that can adapt to these changing demands. This poses a significant challenge for ASICs, which despite their optimized performance for particular tasks, face difficulties with scalability and adaptability.

This inflexibility in specialized hardware solutions leads to the "accelerator wall" [24, 30]. The time and resources required to develop these dedicated accelerators, combined with their inflexibility, put them at risk of becoming obsolete before recouping their initial investment [79], especially given the pace at which wireless communication technologies evolve.

Critical to these workloads is the need for high throughput and minimal latency, essential for meeting the demands of the protocols. Consequently, there is a need for platforms capable of either concurrent kernel execution or rapid reconfiguration, thereby optimizing resource allocation and processing timeframes. Moreover, wireless communication workloads often involve multiple DSP algorithm kernels in pipelined configurations [55]. These kernels, designed for streamed data handling, limit the applicability of conventional caching mechanism, and increase the reliance on direct memory access, underscoring the importance of efficient, concurrent kernel processing.

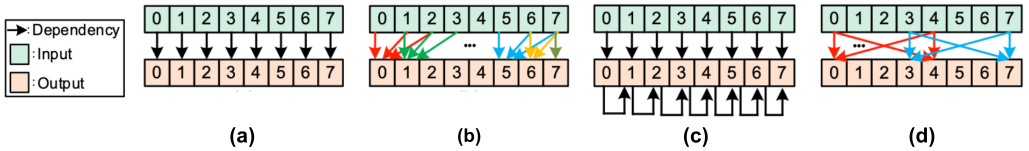


Fig. 2. Diverse data access and compute patterns common in wireless communication including (a) operation on each data sample, (b) continuous and overlapped access, and (c) feedback loop (d) strided access.

The nature of computation within these workloads is also diverse (Figure 2). Tasks range from those that can leverage DLP—such as Symbol Modulation and **Finite Impulse Response (FIR)** filters—to more complex patterns like the feedback mechanisms in **Cascade Integrate-Comb (CIC)** filters or large-stride data handling in **Fast Fourier Transform (FFT)** [32]. These varying patterns present distinct challenges in achieving efficient parallelization and resource allocation.

The computational challenges posed by evolving wireless communication protocols highlight the need for platforms capable of rapid processing and chained execution of diverse kernels with minimal memory access overhead. As these systems must frequently undergo updates or reprogramming, ensuring these operations can be performed within the tight constraints imposed by protocol throughput requirements is essential.

2.2 Real-Time Stream Processing Challenges

Real-time stream processing for wireless communication faces several efficiency challenges. Conventional load/store assist models, with their reliance on memory hierarchies for parallel processing, are ill-suited for handling the sequential nature of streaming data, a limitation highlighted in Figure 3. Analysis based on the Roofline model [97] identifies these operations as memory-bound, with data transfer constraints significantly reducing computational throughput, despite high theoretical peak performance. Wireless communication workloads exacerbate this inefficiency, particularly through the indeterministic arrival of data. Event-driven IoT systems are examples of this unpredictability, which conventional load/store architectures incur overheads due to the need for buffering data before processing. The continuous processing demands in certain wireless communication kernels such as filtering further strain these architectures, rendering typical batch processing and buffering techniques less effective, since the repetition of the "tail" of the last batch of data is required to ensure correct computation results. Variable data rates present additional complications, challenging the efficiency of conventional systems, especially when these systems must accommodate data sampling rate inconsistencies with large input buffers, leading to suboptimal resource utilization.

Several studies [20, 41, 65] have investigated strategies for decoupling memory access from computation. However, the unpredictable nature of real-time data streams in wireless communication requires more resilient solutions. The direct execution model (Figure 3(b)) proposes a viable alternative. This approach minimizes reliance on memory hierarchies, reducing associated data access and movement overheads. It seeks to blend the high-performance aspects of ASICs with the flexibility of programmable architectures, combining the best of two worlds as opposed to conventional compute platforms.

2.3 Limitations of Conventional Programmable Platforms

Conventional programmable platforms exhibit fundamental inefficiencies in real-time stream processing. General-Purpose Processors provide a broad computational utility but suffer inefficiencies in scenarios requiring optimized pipeline management and minimal branch operations [39]. Their architectural framework, although flexible, isn't inherently designed to accommodate the specific

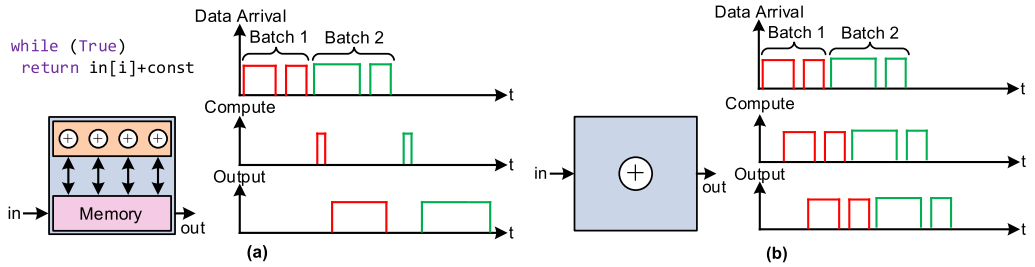


Fig. 3. Comparison of (a) load/store assist model and (b) direct execution model when executing constant addition to an input data stream. The input arrives in sequence and the arrival time is indeterministic. The compute time of model (a) is shorter but overall latency is longer than model (b) due to memory access.

throughput and latency demands, particularly when data reuse is minimal. In contrast, GPUs excel in scenarios that capitalize on high parallelism. However, their performance degrades with workloads presenting intricate data dependencies, common in wireless communication tasks (Figure 2(c)). This limitation stems from their parallel processing nature, which is not adaptive to the serialized data patterns found in such workloads, leading to suboptimal resource utilization. Field-Programmable Gate Arrays offer efficient customization for task-specific operations. Their shortfall becomes evident in their reconfiguration latency, which is detrimental to tasks necessitating real-time processing adaptability [89].

2.4 Programmable Spatial Architecture

Spatial architectures are recognized for enabling DLP and high data reuse, crucial for minimizing memory access overheads. They achieve this by utilizing an array of PEs interconnected through on-chip networks, aligning well with the demands of wireless communication [50, 51, 61].

The PEs of spatial architectures are diverse, extending from fixed-function arithmetic units to those with fully programmable cores that support various dataflow or Von Neumann-style computations. While architectures based on fixed-function units are efficient in terms of throughput and energy, they lack the flexibility required to adapt to evolving workloads in wireless communication. This limitation underscores the importance of programmable spatial architectures [21, 43, 96]. By spatially distributing operations originally time-multiplexed, these architectures reduce switching activity in the datapath, providing reconfigurability atop the benefits of DLP and data reuse.

The broad spectrum of programmable spatial architectures requires comprehensive classification methods to facilitate understanding and comparison. REVEL [94] presents a comprehensive taxonomy based on execution scheduling (static or dynamic) and PEs use (dedicated or shared). Chen et al. [18] focus on task scheduling dynamics of spatial architectures from two aspects, mapping and task issuing. Our proposed architecture, FluxSPU, employs a static mapping approach with dynamic task issuing, featuring shared, time-multiplexed operations for each PE. The architecture adheres to a static schedule and mapping scheme due to the deterministic computation patterns of the kernels involved. However, the execution timing is dynamic and based on data availability [69], a crucial adaptation that enables real-time stream processing efficiency.

3 Why Develop a New Framework?

The goal of Canalis is to develop a framework to meet the demands of real-time stream processing in wireless communication systems. Our solution includes improvements at different levels of abstractions from a single PE to the entire spatial architecture. Canalis is accompanied by a

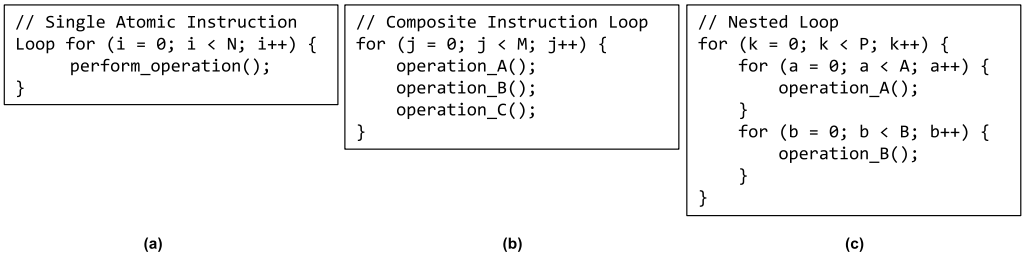


Fig. 4. Examples of "Basic Structure," including (a) Standalone Atomic Instruction Loop, (b) Composite Instruction Loop, and (c) Composite Instruction Loop containing multiple Atomic Instruction Loop.

co-designed software stack, which not only optimizes system performance but also provides a more intuitive interface for subject matter experts.

3.1 Optimizing Throughput Independent of Frequency

It is crucial to maintain protocol-specified throughput when executing stream processing of wireless communication workloads. Essentially, throughput is the product of frequency and output per cycle. From an energy conservation perspective, it is preferable to minimize frequency, thereby necessitating the maximization of output per cycle (i.e., the throughput independent of frequency) all within the boundaries of stream processing constraints.

Programmable spatial architectures stand out in this regard due to their innate support for **task-level parallelism (TLP)** and data reuse, essential for enhancing per-cycle output (Section 2.4). However, we do not introduce additional DLP due to the real-time processing considerations (Section 2.2). On the other hand, parallel input data (e.g., from a **Multiple-Input Multiple Output (MIMO)** system) continue to be processed concurrently. We further eliminate reprogramming time for complete workloads that contain diverse kernels by simultaneously mapping multiple kernels on the fabric, and connecting the output of one kernel to the input of another [17, 84], thereby reducing memory access and re-programming overhead.

Wireless communication workloads typically exhibit a periodic nature, wherein the same operational schedule recurs throughout each loop iteration. This repetition aligns the concept of modulo scheduling [29, 58]. Triggered by data arrival [69], such dynamic scheduling maintains system agility, effectively reducing the II—the number of cycles between successive iterations, which directly affects the overall throughput.

We recognize that the ultimate throughput is constrained by the PE with the lowest throughput. Prior works [8, 86] have utilized circuit techniques to overcome this bottleneck. In contrast, we identify that after operations are spread out across PEs, a "Basic Structure" of program is common in each PE, especially in the wireless communication domain, where operations are periodic. Multiple examples of such "Basic Structure" are depicted in Figure 4, illustrating a "Composite Instruction Loop" that encompasses several "Atomic Instruction Loops," each dedicated to a discrete operation. Therefore, we propose an innovative ISA and microarchitecture optimized for this "Basic Structure," ensuring a consistent one computation per cycle for each PE.

Furthermore, within each PE, we implemented pipelined **Functional Unit (FU)** reordering that provided **Instruction Level Parallelism (ILP)** within each PE while pertaining the order of data sequence in and out of the PE. The optimizations applied to the proposed Canalis framework are summarized in Figure 5. On a system level, FluxSPU fabric can be statically partitioned to support concurrent execution of independent workload instances. Although this does not enhance the throughput of individual workloads, it increases the throughput of the fabric and the overall hardware utilization.

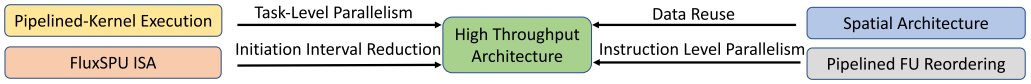


Fig. 5. Optimizations from all levels of abstraction to enhance Canalis throughput.

3.2 Spatial Architecture for Stream Processing of Wireless Communication

While the FluxSPU architecture shares high-level similarities with prior works [34, 65, 72], it distinguishes itself by integrating enhancements essential for stream processing. Notably, we have adapted the fabric boundaries to the demands of real-time processing scenarios by substituting traditional load/store operations with push/pop mechanisms. This modification treats input and output data as sequentially ordered streams, catering to the continuous and sequential arrival nature of dataflow, rather than viewing them as discrete data units.

Within each PE, we have implemented microarchitectures co-designed with the proposed ISA to further optimize throughput. Additionally, we cater to domain-specific computational demands by integrating specialized functional units, such as CORDIC and dividers, complementing the standard computations of adders, multipliers, and logical units.

3.3 Software Stack for Programming the FluxSPU Fabric

Achieving high performance and efficiency often compromises the presence of a robust software stack. Many prior programmable spatial architectures [8, 70, 88, 101–103] targeting stream processing lack a user-friendly programming interface, creating a barrier for subject matter experts [90] unfamiliar with low-level details. On the other hand, most existing spatial architectures that incorporate software stacks typically offer compiler support through high-level languages like C/C++ [10, 83, 86] or require additional program annotations [33, 34, 94]. These methods, however, predominantly depend on the conventional load/store memory model, failing to accommodate the direct execution model essential for real-time stream processing as discussed in Section 2.2. To address these issues, we introduce a software stack, which includes the CER, a programming interface that deviates from the typical abstractions of high-level languages, offering a more immediate description of the computational events. Alongside CER is a mapper for distributing computations across PEs efficiently while applying unique optimizations presented by our ISA. Designed as an abstract representation of computations, CER appeals to professionals who are comfortable with conceptual models like block diagrams, providing an interface that melds familiarity with technical precision.

4 FluxSPU ISA

The FluxSPU ISA is tailored to meet three primary objectives: maximize each node's throughput, minimize datapath switching during program execution, and ensure adaptability to continuous data streams. These goals come to fruition through several optimizations. First, FluxSPU adopts a distinct programming model that perceives operands as continuous streams of arbitrary lengths, diverging from traditional approaches that handle discrete data or vectors of fixed lengths. Second, the ISA incorporates a specialized field in certain instructions, referred to as the "Throughput Enhancer." This field facilitates the seamless integration of computations, combining loop controls within a single instruction. Finally, the microarchitecture of FluxSPU nodes (detailed in Section 5) is meticulously refined to bolster these enhancements, avoiding overheads and reducing switching activity of the datapath.

4.1 High-Level Model of FluxSPU Nodes

Figure 6 illustrates the high-level model of a FluxSPU Switch and a FluxSPU PE. Both types of nodes are composed of two primary components: the *Control* and the *Datapath*. Instructions are loaded into the *Control* of PEs and Switches before program execution commences.

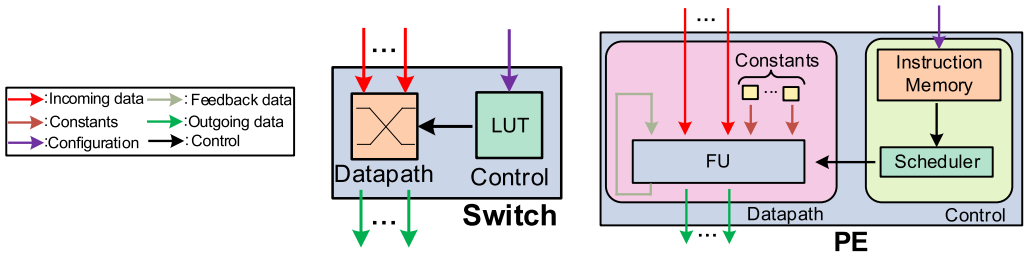


Fig. 6. High-level model of FluxSPU Switch and PE. LUT, lookup table.

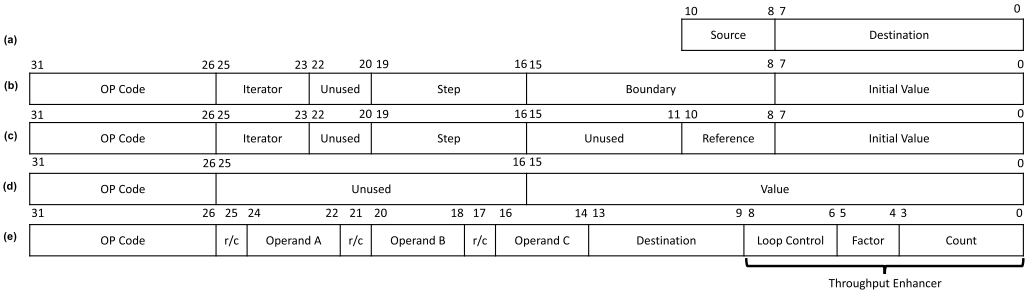


Fig. 7. Structure of different types of instructions, including (a) Route Configuration of Switch, (b) *Start Loop* instruction for constant-bounded loop, (c) *Start Loop* instruction for polyhedral loop, (d) instruction to load constant, and (e) generic action type instruction supporting up to three operands.

Switch Programming Model. Switches operate under a static configuration. Once set, data entering a port is consistently routed (or multicasted) to the designated output port(s) for the duration of the program. The software stack prevents route conflicts within the Switches, eliminating the need for arbiters to ensure the correct output sequence. This approach also circumvents the overhead associated with tagging systems prevalent in architectures that require arbitration. Multiple routes can coexist in a switch as long as there are no conflicts, which is guaranteed by the software stack. Figure 7(a) details the structure of a Switch configuration.

PE Programming Model. During each cycle of program execution within a PE, the scheduler within the *Control* fetches one instruction from the instruction memory and issues it to the *Datapath* when the *Datapath* is ready. The model is simplified for better understanding; therefore, back-pressure between the scheduler and the instruction memory can be disregarded for now (i.e., the scheduler has a sufficiently large buffer for storing instructions).

Throughout the computational process, the FU accesses operands from three types of sources: external input data streams, constant values, or feedback data streams. After the computation completes, results can be multicast to external output data streams or fed back into feedback data streams. This multicast capability realizes the overlapped data access pattern illustrated in Figure 2(b), and the feedback mechanism enhances throughput for feedback loop access patterns as shown in Figure 2(c). The feedback data stream ensures that the datapaths for kernels containing feedback such as Infinite Impulse Response or simple counters can be constructed within a PE instead of hopping through the fabric, reducing the initial interval [86] and increasing the throughput. Moreover, it reduces hardware utilization as the routing is kept inside the PE.

PE instructions fall into three categories: *Loop*, *Configuration*, and *Action*, all encoded in a 32-bit format, with their structures depicted in Figure 7. *Loop* type instructions guide the program's execution flow but are not issued to the *Datapath*. *Configuration* type instructions handle computation-related configurations, such as loading constants, and while they reach the *Datapath*,

they do not *initiate* computations. Computation events are *initiated* by issuing *Action* type instructions to the *Datapath*. However, even if computation events are *initiated*, the computation will only be *triggered* upon data arrival [69], similar to a dataflow machine.

It's imperative to distinguish between *initiating*, *triggering*, and *executing* a computation within the FluxSPU programming model. A computation is *initiated* when the scheduler issues an instruction to the *Datapath*. However, the actual occurrence of the computation depends on data availability; the *Datapath*, at this juncture, enters a ready state, poised for action. Only when valid data becomes available following the *initiation* does the computation *trigger* and start to *execute*, transitioning from a dormant state to active processing. As certain operations might take multiple clock cycles to *complete*, while each PE can only *initiate* one operation and *trigger* another operation in a single cycle, it can *execute* and *complete* multiple operations in the same cycle. This nuanced interplay ensures precision in execution, contingent directly on real-time data readiness, and maximizes hardware utilization.

4.2 Loop Type Instructions and Configuration Type Instructions

Loop instructions support ordinary loops and polyhedral loops (Figure 7(b) and (c)). There are two types of *Loop* instructions. *Start Loop* initiates a loop, setting the starting value, boundary, incremental step of the iterator, and return address, while *Evaluate Loop* determines whether to loop back or proceed. Both the starting value and boundary can be assigned a constant value, as seen in standard loops, or derive the current value of another ongoing loop, enabling the execution of polyhedral loops, which are integral to many complex computational tasks in wireless communication workloads [94]. FluxSPU is specifically designed to support only data-independent loop operations, aligning with the needs of the target application. In the domain of wireless communication, where data-dependent conditional operations are infrequent, the use of ternary operations by FluxSPU is sufficient. These operations facilitate the conditional "masking off" of results based on predefined criteria, thus optimizing execution flow. This method avoids the complexity and overhead associated with more intricate branching mechanisms, which are rarely necessary given the predictable data patterns typical in wireless communication scenarios.

Configuration instructions (Figure 7(d)) set up the computational environment, usually at the very start of the program. Due to their infrequent use, and the decision to separate them into dedicated instructions rather than adopting a Very Long Instruction Word approach, performance remains uncompromised while keeping the instructions compact.

4.3 Action Type Instructions

In addition to a conventional ISA which specifies the operation, the source, and the destination, the *Action* type instructions (Figure 7(e)) implement multiple optimizations. Support of data multicast provides a more flexible dataflow mapping for spatial architectures, and FluxSPU adds additional fields that specify accessing mechanisms including *Read* and *Consume*, at the data streams' interfaces (r/c field in Figure 7(e)). *Read* allows repeated data reading, while *Consume* removes data after a single use. This approach provides flexible data alignment and different consumption rates of data streams. Common use cases of *Read* include upsampling and filtering (Figure 2(b)).

The FluxSPU ISA deviates from traditional emphasis on register transfer and data dependency and focuses on the sequence of computations. This approach is exemplified in its handling of data streams, operating directly on the "head" of the input stream without the prerequisite of a defined "stream length." The interplay between this methodology and the use of loops enables the creation of sequential data streams of arbitrary length, which significantly amplifies processing efficiency and system adaptability. Furthermore, since the computation is only *triggered* upon data arrival and the order of data in the output data streams is honored by simply appending new data

Table 1. Operations Supported by FluxSPU ISA

Type	Operations	Latency
A (Arithmetic)	Pass, Pop, Add/Subtract (Overflow or Saturate), Logical Complex Conjugate, Shift, Bit Reversal, Ternary, Compare	1 Cycle
M (Multiplier)	Multiplication and Arithmetic Shift, Multiplication by Conjugate and Arithmetic Shift	3 Cycles
D (Data Scratchpad)	Array Read/Write, FIFO Queue Read and Write	3 Cycles
N (Non-Linear)	Division[a], Square Root[a], Complex Magnitude, Complex Argument, Complex Rotation, Cartesian to Polar Conversion	7 Cycles

[a]The divisor for division, or the input to square root must be real or in polar representation.

to the "tail" without replacing any data, Read-after-Write hazards are handled inherently while Write-after-Read and Write-after-Write hazards are eliminated. It is important to preserve the order of data in the data streams transferred between PEs and Switches so that "tags" are not needed. The optimizations provided by the FluxSPU ISA will strictly follow this constraint while aiming to enhance the throughput.

The FluxSPU ISA encompasses a wide variety of operations, extending beyond basic arithmetic and logical functions to include domain-specific computations such as division and rotation, all of which are summarized in Table 1. Moreover, FluxSPU ISA supports operations related to scratchpad memories as FUs inside the PEs. These scratchpad memories are integral for various data buffering purposes, including the temporary storage of data for swift and efficient access during computations, the storage of constant values such as matrix weights to ensure consistency across various operations, and the use of lookup tables to preclude repetitive calculations by providing immediate access to pre-computed values. A particularly noteworthy feature is the **First In First Out (FIFO)** mode. This mode is vital for averting deadlocks in computations, particularly in instances where both operands originate from the same data stream, yet there is a discrepancy in their sequential positions within that stream. An illustrative example of this can be seen in Table 2(g), where the FIFO mode's systematic data processing approach is crucial for preserving the integrity of the operational sequence and effectively managing dependencies inherent within the data stream.

The most unique characteristic of the *Action* type instructions is the "Throughput Enhancer" field. This field is dedicated to reducing IIs, significantly enhancing the overall performance of the FluxSPU architecture. In the next subsection, we will walk through the optimizations provided by the FluxSPU ISA, with an emphasis on the "Throughput Enhancer" field.

4.4 Increasing Per-PE Throughput

FluxSPU focuses on enhancing frequency-independent throughput, aiming to sustain the application-demanded throughput at a lower operational frequency and, in turn, reduce energy consumption. Equivalent ASIC implementations of workloads often realize an ideal II and maximum throughput. Therefore, FluxSPU's aspiration is to emulate this level of performance through various optimizations. Since the Switches are statically configured and do not require arbitration, they always achieve the theoretical maximum throughput. As a result, the PEs are the main optimization targets.

We will elucidate the efficacy of each optimization with different simple example (Figures 8 to 10) and another example (Figure 11) that combines all optimizations. Given the PE's capability to multicast, we opt to express throughput in terms of "average computations *triggered* per cycle," providing a more intuitive understanding than the conventional "average output data per cycle."

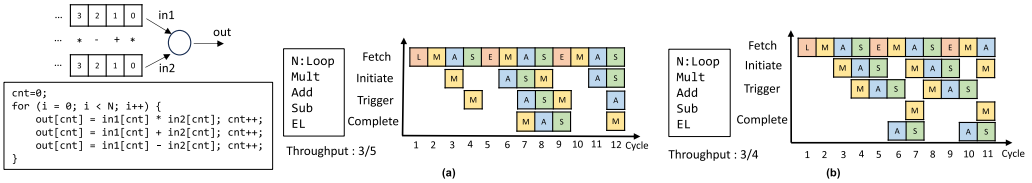


Fig. 8. Simple example program showing difference between (a) before and (b) after implementing pipelined instructions and FU reordering, resulting in throughput increase by 1.25×. EL, evaluate loop.

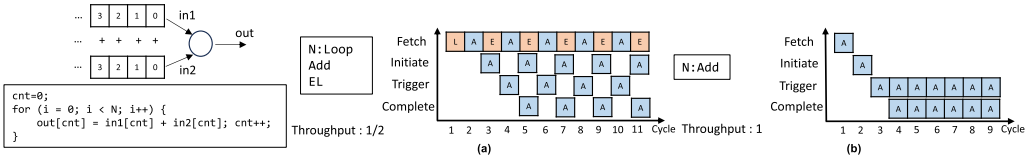


Fig. 9. Simple example program showing difference between (a) before and (b) after implementing Embedded Atomic Instruction Loop Count, resulting in throughput increase by 2×.

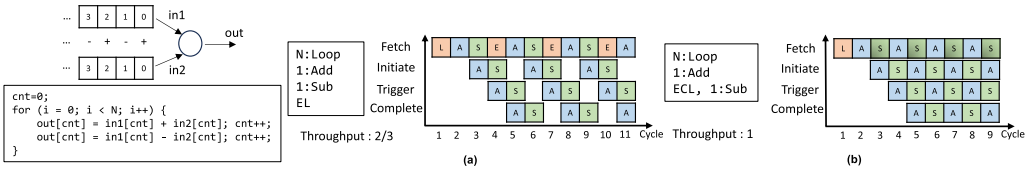


Fig. 10. Simple example program showing difference between (a) before and (b) after implementing Embedded Composite Instruction Loop Evaluation, resulting in throughput increase by 1.5×. ECL, End Composite Loop.

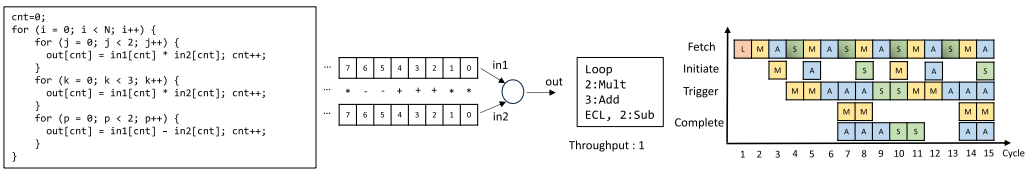


Fig. 11. Example program with all optimizations implemented, showing that FluxSPU can achieve maximum throughput for any code with the "Basic Structure" in Figure 4.

Since each PE is a single-issue machine, the theoretical maximum throughput is one computation *triggering* per cycle.

The programs selected for this illustration perform straightforward element-wise computations on two data streams, encompassing operations such as multiplication, addition, and subtraction. Notably, we stipulate that the multiplication operation takes three cycles, underscoring that our optimizations are designed to be compatible with both single-cycle and multi-cycle FUs. To concentrate exclusively on the PE's performance in the context of the proposed architectural enhancements, we proceed under the assumption of continuous, uninterrupted input data streams.

Instruction Pipelining. Combining the single-issue of instructions with pipelined multi-cycle FUs, FluxSPU removes structural hazards. Furthermore, FluxSPU supports pipelined instruction issuing, a strategy that allows for the *initiation* of new computations immediately after all computations in the previous instruction are *triggered*. This method eliminates the need to wait for these computations to *complete*, ensuring a smooth, uninterrupted flow of instructions.

In terms of computation and data output, FluxSPU *initiates* computations in-order, in alignment with the sequence of both input data streams and program instructions, yet *completes* them out-of-order due to the disparate latencies of FUs. Despite this, FluxSPU upholds the integrity of the output data stream, keeping it in-order. This consistency is achieved not by inserting **No Operations (NOPs)**, which would adversely affect throughput, but rather through a low-overhead reordering mechanism, the specifics of which will be elaborated upon in Section 5. The effectiveness of this optimization is shown in Figure 8 with an example program.

However, even with the pipelined issuance of *Action* type instructions, a limitation is presented by the *Loop* type instructions. These instructions require continual reassessment, thereby hindering the attainment of the ideal throughput—one computation *initiation* per cycle. The subsequent optimizations will specifically address *Loop* type instructions to further increase throughput.

Embedded Atomic Instruction Loop Count. The "Throughput Enhancer" field embeds Atomic Instruction Loop Counts, reducing II associated with evaluating loop returns. To ensure scalability, especially for large loop counts, the "Factor" field facilitates the decomposition of these larger counts into instruction sequences with smaller loop segments, all without impacting performance due to the pipelined nature of all instructions. Furthermore, FluxSPU supports infinite loops, crucial for adapting to unknown data quantities in real-time processing scenarios at compile time. An example of the effectiveness of this optimization is shown in Figure 9.

FluxSPU's spatial design redistributes instructions that would traditionally be time-multiplexed. Essentially, a loop consisting of several operations on a data stream is restructured as multiple, pipelined, and repeating instructions across different PEs. This rearrangement of operational order—akin to the vector chaining strategy in the Cray machines [76]—enhances parallelism and efficiency in data processing.

Embedded Composite Instruction Loop Evaluation. To further optimize performance, FluxSPU introduces an extra filed (Loop Control) in the "Throughput Enhancer," specifically for evaluating the return condition of the current loop when the condition is straightforward, such as an increment/decrement of an iterator, or an unconditional loop back. This reduces II for Composite Instruction Loops that encompass multiple Atomic Instruction Loops by eliminating the need for a separate instruction for evaluation. Instead, the evaluation is seamlessly incorporated into the last instruction of the loop, enhancing processing efficiency. An example is shown in Figure 10. However, for more complex evaluations, an explicit *Evaluate Loop* instruction remains necessary, although such scenarios are infrequent. Crucially, this optimization does not extend the critical path, as the loop evaluation and the FU operations are decoupled, allowing for parallel execution. This design decision ensures that comprehensive loop processing enhancements do not compromise overall system performance. An example program that employs all optimizations is shown in Figure 11.

5 FluxSPU Microarchitecture

Figure 12 presents the detailed microarchitecture of a FluxSPU PE. Each PE in the FluxSPU fabric connects to four neighboring Switches and no direct connection exists between any two PEs. Therefore, only the Switches are responsible for routing data, allowing the PEs to focus on computation. Section 4.1 covered the high-level programming model of FluxSPU PEs and Switches. In this section, we delve into the specific microarchitectures that support the FluxSPU ISA.

5.1 Inner-Loop and Outer-Loop Handling in FluxSPU PE

The instruction buffer hides the latency of instruction fetch and decoding. *Loop* type instructions are managed by the Loop Control Unit in the *Control*, which contains multiple Loop Handlers, each responsible for tracking an ongoing loop. As depicted in Figure 7(c) and (d), the instruction specifies

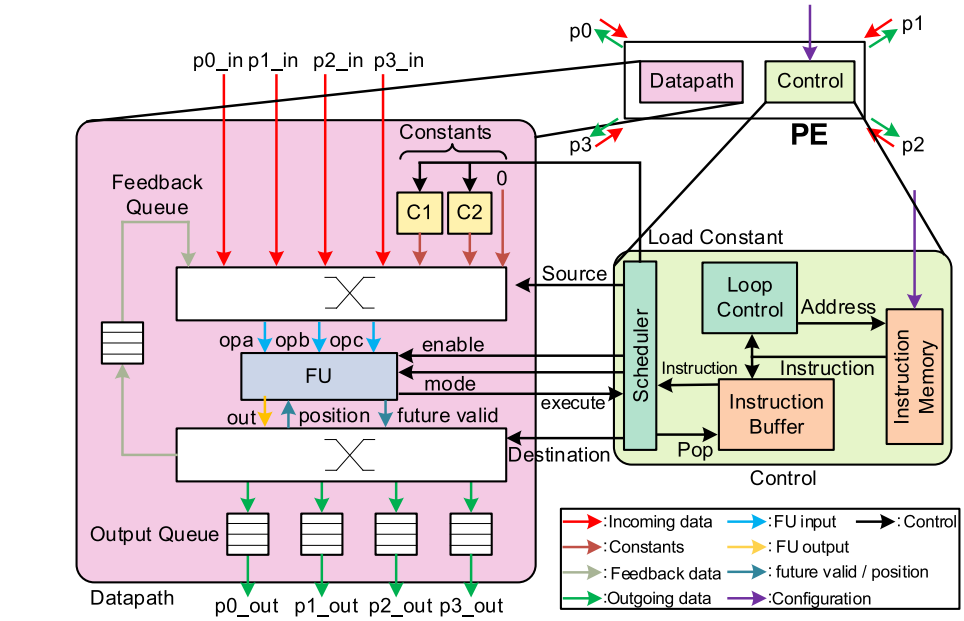


Fig. 12. Architecture of a PE. The FU has eight possible sources and five possible destinations that are configured by the scheduler along with the mode of the FU. The Loop Control Unit handles the flow of the programs. opa, operand a; opb, operand b; opc, operand c.

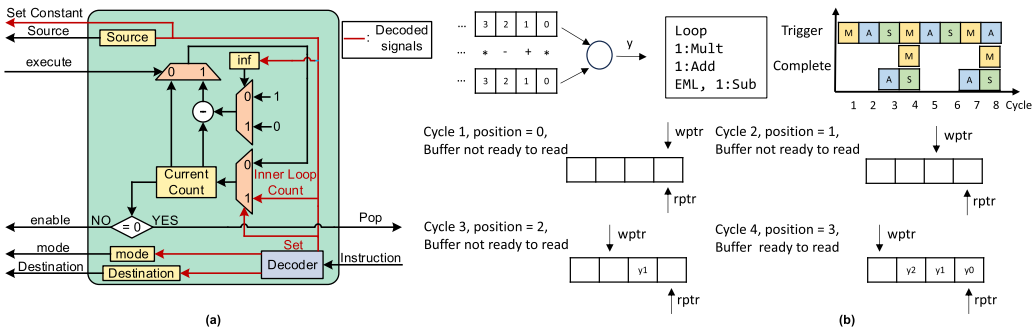


Fig. 13. (a) Microarchitecture of PE scheduler including a decoder that decodes incoming instruction, a current count keeping track of the loop count, and configuration register for the source, mode and destination of the functional unit. (b) Example of pipelined FU reordering maintaining high throughput and correct output data sequence for FUs with different execution latencies. EML, End Monolithic Loop; inf, infinite loop or not; rptr, read pointer; wptr, write pointer.

which Loop Handler to allocate, a decision made during assembly generation in the software stack. Upon fetching a *Start Loop* instruction, a Loop Handler is allocated, storing the current iteration count, the branching condition, and the iterator's step. The Loop Handler receives updates upon fetching an *Evaluate Loop* instruction.

The scheduler (Figure 13(a)) manages the Atomic Instruction Loops using a single counter. After the final computation in the *Datapath* is *triggered*, the scheduler *initiates* the subsequent computation in the *Datapath*, encompassing the source of the operands, the FU's mode, and the output's destination. With each computation's *initiation*, the counter decrements.

5.2 Data Stream Handling in Switches and PEs

FluxSPU adopts an asynchronous valid-ready handshaking mechanism, which relieves the compiler of fine-grained scheduling and facilitates connections between different throughput kernels. This paradigm is adept at adapting to unpredictable data arrival and consumption, ensuring optimal dataflow handling. For each operation on data streams, there are producers, consumers, and a crossbar in between. In managing backpressure, the crossbar will only access the producers and consumers and *trigger* the computation (or routing in Switches) when all the producers and consumers are "ready"; this state implies that the producers have valid data and the consumers have "space" in the output buffer. Since FluxSPU utilizes static mapping, predetermined by the Mapper in the software stack, no collision will occur, and the crossbars are implemented with simple Sum-of-Product (OR-AND) structures instead of a more costly arbitration network.

5.3 Pipelined FU Reordering

The buffers that receive output data from the FUs and contribute to the feedback stream and output streams in the *Datapath* are specifically designed for two purposes: handling backpressure for multi-cycle operations and ensuring the correct sequencing of data for out-of-order FU completion, as discussed in Section 4.4. The data in the buffers are still read out in-order to maintain the correct sequence for computation. However, data can be written into the buffer in any order, transforming the buffers from standard queues into something akin to a hybrid between a scratchpad and a FIFO.

As shown in Figures 12 and 13(b), when a computation is *triggered*, a "future valid" signal is sent to the destination buffers, advancing the write pointers within. Simultaneously, the current value of the write pointers ("position" in Figure 13(b)) is sent to the FU, ensuring that when the computation completes, the data will be directed to the correct entry in the buffer. As the buffers are shallow, this methodology results in a low overhead implementation since the "positions" require just 2-3 bits. This approach is a departure from traditional systems that would typically rely on an additional reordering buffer, introducing unnecessary complexity and potential performance hits.

Using an additional control signal to maintain backpressure is common in prior works (e.g., "allocation" signal in [34] PEs). However, FluxSPU further facilitates a form of low-cost ILP within each PE by ensuring that computations are efficiently reordered. By eliminating the need for NOP insertions typically required to maintain output sequence integrity, FluxSPU achieves higher computational throughput without the overhead commonly associated with such enhancements.

6 FluxSPU Fabric

The FluxSPU fabric connects the PEs and the Switches into a scalable 2D array, as shown in Figure 14. FluxSPU leverages MIMD parallelism, enabling PEs to execute different programs simultaneously. This feature allows for efficient concurrent kernel execution and TLP across PEs, spatially distributing operations that might otherwise be time-multiplexed.

6.1 Topology

There are two unidirectional links between connected nodes in the FluxSPU fabric, providing maximum inter-node throughput. On the FluxSPU fabric's boundaries are Fabric Ports that connect to receiving/transmitting components with minimal buffering in an embedded system or stream interfaces in an **Systems-on-Chip (SoC)**. Communication between the Fabric Ports and the Switches follows the same asynchronous valid-ready handshaking mechanism described in Section 5.2, realizing the direct processing model proposed in Figure 3 and is adaptive to unpredictable data arrival.

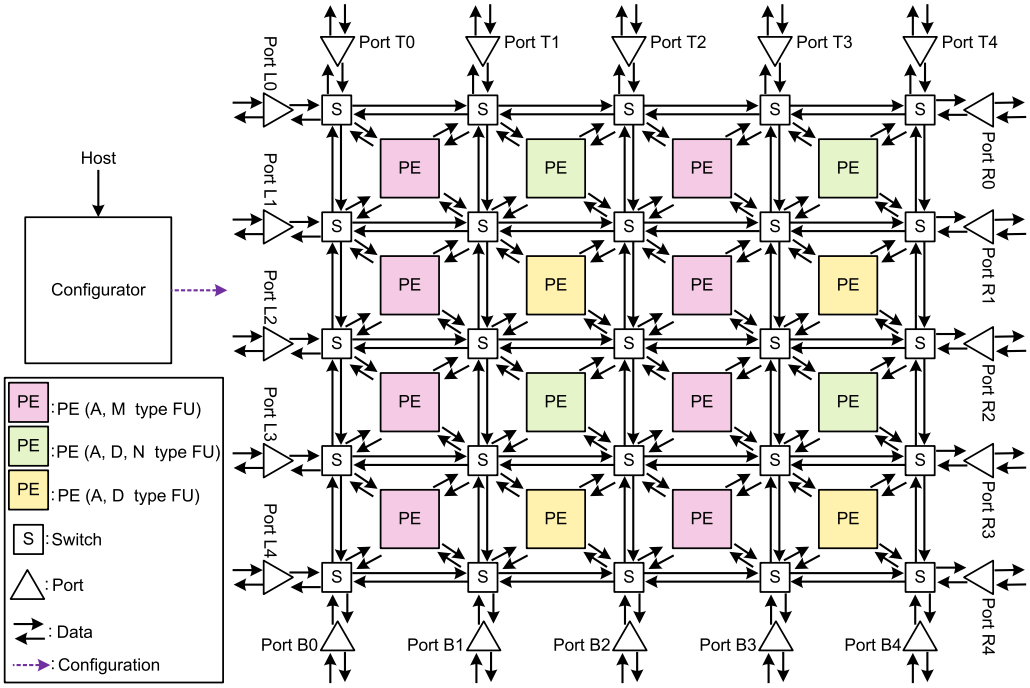


Fig. 14. FluxSPU architecture overview, using a 4×4 array as an example. There are three types of PEs, each contain different types of functional units. All connections between PEs, Switches, and ports are systolic. The Configurator programs each PE and switch through multi-hop configuration buses before program execution.

The Switches in FluxSPU connect to neighboring nodes in all eight directions. With multiple potential routes between nodes, static configuration of Switches is efficient and effective, reducing the need for runtime changes. Although communication between PEs must occur through Switches, the impact on performance is negligible due to pipelined computation.

6.2 Locally Heterogeneous, Globally Homogeneous Fabric

The functional units in the FluxSPU fabric exhibit a locally heterogeneous, yet globally homogeneous distribution. Locally, each PE's functional units mirror real-world usage patterns, with operations like addition being more common than others, such as division [10, 34]. This local heterogeneity allows for specialization within the bounds of generalization, minimizing resource wastage from underutilized functional units. Globally, FluxSPU's architecture ensures a homogeneous distribution, maintaining equal computational capacity and operational diversity across all PEs, which is consistent with FluxSPU's ability to execute multiple concurrent workload instances.

The PE's functional units are categorized into four primary types (Table 1): single-cycle arithmetic (Type A), multi-cycle multiplier (Type M), multi-cycle non-linear (Type N), and data scratchpad (Type D). We specifically distinguish multipliers as Type M because of their prevalence in kernels such as filtering. However, considering they often pair with additions to form Multiply-Accumulate operations, their density within the fabric is halved.

6.3 FluxSPU Fabric Configurator

Prior to program execution on the FluxSPU fabric, instructions and configurations are loaded into the PEs and Switches via the Configurator (Figure 14). Given the FluxSPU fabric's capacity

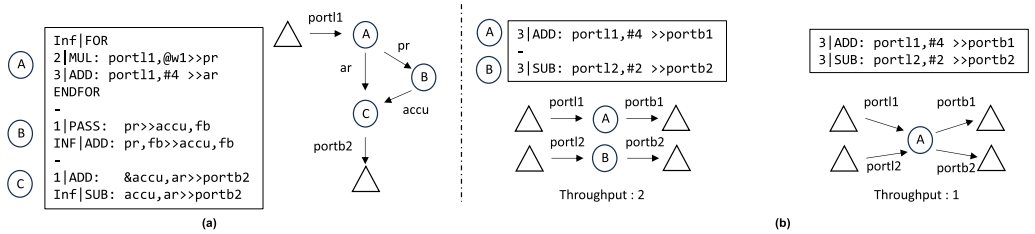


Fig. 15. (a) Example of a CER with three nodes and its corresponding DSG (b) comparison of same computation results with different resource allocation controlled by the CER. DSG, Data Stream Graph.

for simultaneous execution of independent workloads, the Configurator employs an additional network, designed to avoid disruption of any ongoing computations. This unidirectional and multi-cycle network imposes minimal overhead, attributable to its infrequent use (hence, low switching activity) and the short routing distance between nodes.

Notably, the adoption of a multi-cycle approach, while inducing increased latency for individual instruction loading, serves an advantageous purpose. This latency is effectively obscured by initiating the loading process with the node situated at the greatest distance from the Configurator. Such a methodical sequence of operations ensures that while the instruction traverses the longest path, subsequent loading activities for other nodes are concurrently underway, thereby maximizing time efficiency and preserving system performance momentum.

7 Canalis Software Stack

The Canalis software stack comprises two integral components. The first is the CER, an accessible programming interface designed specifically for the FluxSPU. It simplifies the process of executing the desired computations on the FluxSPU. It's important to note that the *CER does not function as a hardware generator nor does it compile programs into a hardware description language*; its sole purpose is to facilitate the programming of the FluxSPU. The second component, known as the mapper, plays a crucial role following the CER's utilization. The mapper efficiently assigns the programs developed with the CER onto the nodes within the FluxSPU, converting them into the specific instructions of FluxSPU ISA.

7.1 FluxSPU Programming Interface: CER

The introduction of CER aims to provide Canalis users with sufficient architectural detail of FluxSPU to realize the target computation, relieving the complexity of mapping computations, routing data between nodes, and applying optimizations inherent in the FluxSPU ISA. For users familiar with FluxSPU, CER programs can be further enhanced to utilize the full potential of the architecture. CER strikes a balance between abstraction and flexibility with the following features:

Data Stream Graph (DSG). In contrast to traditional approaches that represent programs as a **Dataflow Graph (DFG)**, CER employs a DSG. In a DSG, each node corresponds to a specific node in the FluxSPU fabric, with each edge between nodes symbolizing data streams from a producer node to a consumer node. An example of CER and its corresponding DSG is shown in Figure 15(a). The DSG does not contain routing information, leaving these decisions to the Mapper. One notable distinction between a DFG and a DSG is that in the latter, each node accommodates multiple instructions. If each node in the DSG was replaced with that node's DFG, the result would be the DFG for the whole fabric. These instructions correspond to individual programs containing multiple instructions loaded into the FluxSPU array's nodes. Various instructions within the same node may

access the same incoming edges (data streams) or contribute outputs to the same outgoing edges, aligning with the programming models (Section 4.1).

The CER describes the DSG using expressions akin to the FluxSPU ISA. It is required for the programmers to differentiate nodes in their CER programs. The requirement is crucial, as it may lead to different throughput when mapped onto the FluxSPU fabric even with the same computation results, as illustrated in Figure 15(b). This approach provides users greater authority in managing the balance between performance and resource utilization. During the mapping process, multiple nodes in the CER may be mapped onto the same FluxSPU node. The Mapper ensures this consolidation does not compromise overall throughput, only merging nodes that can be realized by a single Switch as multiple routes can coexist in a Switch.

CER Computation. The computation expression within the CER framework closely mirrors the structure of *Action* type instructions. It encompasses the number of executions, operation type, input operands, and output streams. Each *Action* type instruction in CER has the following: the number of repetitions of the instruction, either a positive integer or "Inf" for infinity, then "—" and the opcode, then ":" and one or multiple input operands, separated by commas when there are more than one, then "»" followed by the output operands, also separated by commas. The framework defines five distinct operand types:

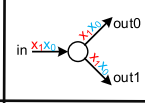
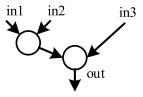
(1) Static Constants: Symbolized by a preceding "#," these constants are analogous to the immediates in traditional ISAs. *Configuration* type instructions are employed to load them.
(2) Dynamic Constants: Identified by an "@" symbol before the operand name, their exact values remain elusive at the programming phase, yet they are loaded from the Configurator prior to program execution. This distinction fosters code adaptability, especially in computations like filtering where the dataflow remains consistent but weight values vary with each run.
(3) Feedback Stream: Denoted by the keyword "fb," this operand establishes a localized feedback dataflow. It significantly bolsters throughput and conserves hardware resources. However, this edge isn't explicitly represented in the DSG as it is local to a node.
(4) Inter-Node Stream: These streams represent the edges in the DSG, characterized by one producer node and one consumer node. Even though FluxSPU's ISA supports multicasting, distinct output streams must be represented individually. Inter-node streams are denoted by strings beginning with a letter.
(5) Fabric Ports: Prefaced with the "port" keyword, these streams are variants of inter-node streams where one end is already mapped during programming. Their pivotal role becomes evident in subsequent mapping phases.

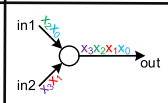
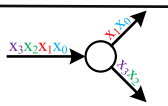
An "&" symbol can be prefixed to input streams (feedback, inter-node, and ports) to support the *Read* operations detailed in Section 4.3, allowing for referencing the "head" of a given stream. By default, references of streams cause the head to be consumed.

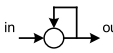
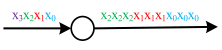
Loops in CER. Atomic Instruction Loops of one repeated instruction in CER are seamlessly integrated into computations, echoing the approach found in FluxSPU's ISA. This means that their presence is implicit within computational expressions, without the need for distinct loop constructs. On the other hand, Composite Instruction Loops for multiple different instructions in CER are explicitly represented using the "FOR" and "ENDFOR" constructs. These structures embed the loop count directly, as demonstrated in Figures 15(a) and 16(a), with the same format as *Action* type instructions. It is significant to highlight that CER is equipped to handle polyhedral loops. To facilitate this, the loop counts—whether for the "FOR" construct or the associated computation—are denoted by variables with a "iter" prefix, followed by the start value and step value, all separated by commas. This convention sets the stage for a later association with specific Loop Handlers during

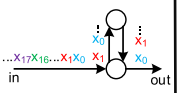
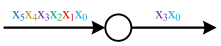
Table 2. Example Mapping of Different Dataflow on Canalis

Name	Dataflow	CER	Name	Dataflow	CER

(a) Multicast		`inf|PASS: in>>out0,out1`	(b) Cascade		`inf|MUL: in1,in2>>pd`
					`-`
					`inf|ADD: pd,in3>>out`

(c) Merge		`inf|FOR`
		`1|PASS: in1>>out`
		`1|PASS: in2>>out`
		`ENDFOR`
			(d) Split		`inf|FOR:`
					`2|PASS: in>>out0`
					`2|PASS: in>>out1`
					`ENDFOR`

(e) Feedback		`inf|ADD: in,fb>>out,fb`	(f) Up-sample		`inf|FOR:`
					`2|PASS: &in>>out`
					`1|PASS: in>>out`
					`ENDFOR`

(g) Stagger		`inf|FOR:`
		`16|PASS: in>>fh`
		`16|SUB: dh,in>>out`
		`ENDFOR`
		`-`
		`inf|FIFO: fh>>dh`
			(h) Down-sample		`inf|FOR:`
					`1|PASS: in>>out`
					`2|POP: in>>`
					`ENDFOR`

(a) Multicast: A single data stream is sent to multiple destinations; (b) Cascade: The output of one node is the input of another node; (c) Merge: Multiple data streams merged into a single interleaved stream; (d) Split: A single data stream is sent to multiple streams, time-interleaved; (e) Feedback: The output stream feedbacks to be computed with the input stream; (f) Up-sample: Each input data are repeated; (g) Stagger: An input stream is divided into two parts, and samples distant in arrival sequence are paired. An additional PE is used to prevent deadlock; and (h) Down-sample: Data are periodically discarded in an input data stream before sent to the output stream.

the phase of assembly code generation. The CER representation of common dataflow in wireless communication are summarized in Table 2.

7.2 Mapping CER to FluxSPU

The mapping phase encompasses several key steps: parsing the CER to form the corresponding DSG, determining the mapping order, allocating nodes, routing, and generating the FluxSPU assembly code. Given that the CER is closely aligned with the FluxSPU ISA, the tasks of parsing it into a DSG and generating assembly code are relatively straightforward. Consequently, the subsequent discussion will primarily center on the mapping order, node allocation, and routing.

Mapping Order Determination. During the parsing phase, apart from generating the DSG to establish relationships between nodes, an "Instruction Dependency Graph" is also constructed. This graph is pivotal in ascertaining the sequential order in which different nodes are mapped. The procedure initiates with producer nodes that are identified as Fabric Ports, and the traversal ensues along the dataflow of the program. For every instruction, its position or "score" in the sequence is determined by taking the highest score among its producers and then incrementing that value by one. The score of Fabric Ports as producers are 0 as the initial condition. If an instruction's producers cannot be traced back to a Fabric Port, the producer–consumer relationship is temporarily inverted. This inversion continues until the node's producer can be traced back to a Fabric Port. The underlying principle of this method hinges on starting from what is most known or established. Since most operations possess a higher number of input operands compared to outputs, it's imperative to initiate the tracing from the absolute "roots" of all producers, which are the Fabric Ports serving as inputs. The rationale for reversing the order in certain cases is to consistently begin with Fabric Ports. Failing to adhere to this approach risks mapping a node without any reference or constraints, potentially

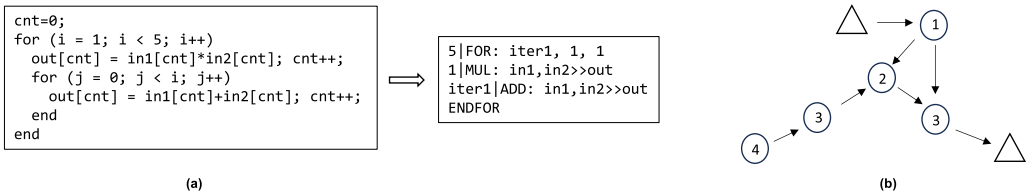


Fig. 16. (a) Example of polyhedral loop representation in CER (b) scores assigned to each node of the Instruction Dependency Graph in Mapping stage.

compromising the integrity of the entire mapping sequence and resulting in multiple "backtracks" in the subsequent allocation process. An example of such scoring is shown in Figure 16(b).

Node Allocation and Routing. After establishing the mapping sequence, the process advances to allocate specific nodes in the FluxSPU fabric for each node in the constructed DSG. For every DSG node to be mapped, FluxSPU nodes nearest to its already mapped neighbors (both producers and consumers) are considered as candidates to ensure efficient inter-node communication. It is imperative that the chosen FluxSPU node contains the necessary functional units. Due to the non-uniform distribution of functional units across the FluxSPU fabric, this step not only guarantees the node's computational capability but also aims for a compact overall mapping as the minimum number of "Basic Blocks" (2×2 array of PEs containing all functional units) is calculated during parsing. Another key consideration is the feasibility and distance of routing between nodes. In this regard, the mapper employs the A-star algorithm [40] to find routing paths between nodes. Lastly, by referencing the DSG, the number of "unmapped neighbors" a node has is determined, ensuring the selected node has ample surrounding space for potential future allocations.

8 Methodology

Canalis Modeling. We implemented Canalis (10×10 FluxSPU Array) using a custom cycle-accurate model to calculate performance metrics. A prototype of Canalis is implemented and synthesized using Synopsys Design Compiler with commercial 28 nm standard cell technology. To generate precise power consumption measurements, we use extensive test vectors to derive the Switching Activity Interchange Format, which serves as input for the gate-level synthesized designs. While Canalis is synthesized at 500 MHz, it has the capability to scale up to 1.25 GHz due to its short critical path. However, Canalis' high throughput enables it to meet protocol requirements at lower frequency, improving energy efficiency.

Benchmarks. Canalis' performance is evaluated using a diverse set of eight benchmarks. This suite encompasses three commonly utilized kernels and five wireless communication workloads, summarized in Table 3. These benchmarks were specifically chosen as they encapsulate the characteristics of pipelined kernels, disparate processing patterns, and variable data rates discussed in Sections 2 and 3. FIR covers the continuous and overlapped access in Figure 2 while FFT requires strided access. GeMM showcase Canalis' ability to handle parallel input. Individual kernels in the workloads executed displays Canalis' adaptability including feedback loop (frequency modulation, CIC), change of data rate (up-sampling/down-sampling), and domain-specific computations (equalization).

Baselines. We compare Canalis with the CPU and GPU separately in the Snapdragon 855 Mobile SoC across all benchmarks to assess the performance speedup and energy savings gained from specialization. Additionally, to understand the energy overhead associated with a more generalized approach, we compare with the equivalent ASIC implementations for each individual benchmark.

— *CPU: Arm Cortex-A76 (@2.8 Ghz):* Out-of-Order Core, using optimized CPU libraries for FIR (CMSIS-DSP [4]), FFT (PFFFT [5]), GeMM (XNNPACK [6]), and trigonometric functions(ARM

Algorithm 1: Mapping Order Determination

```
1: procedure COMPUTE_SCORE(Graph G)
2:     legal_list ← {}                                          ▷ empty map
3:     removed_list ← []                                        ▷ empty list
4:     for all (node_key, node) ∈ G.nodes do
5:         if node.parents = ∅ and node_key[0] = 'q' then
6:             append node_key to removed_list
7:             for all child_key ∈ node.children do
8:                 remove node_key from G.nodes[child_key].parents
9:         else
10:            legal_list[node_key] ← node
11:
12:    visited ← ∅                                              ▷ empty set
13:    for all key ∈ legal_list do
14:        if legal_list[key].parents = ∅ then
15:            legal_list[key].score ← 0
16:            insert key into visited
17:    while |visited| < |legal_list| do
18:        for all key ∈ legal_list do
19:            if key ∉ visited and legal_list[key].parents ⊆ visited then
20:                legal_list[key].score ← 1 + max(legal_list[parent_key].score : parent_key ∈
   legal_list[key].parents)
21:                insert key into visited
22:
23:    for all node_key ∈ removed_list do
24:        remove node_key from removed_list
25:        if ∃ child_key ∈ legal_list ∩ G.nodes[node_key].children then
26:            node.score ← 1 + max(legal_list[child_key].score : child_key ∈ legal_list ∩
   G.nodes[node_key].children)
27:            legal_list[node_key] ← node
28:        G.nodes ← legal_list
29:        CATEGORIZE_BY_SCORE(G)
```

Optimized Routines [1]). The implementation is compiled with level 3 optimization using the *Android NDK r23c* framework.

— *GPU: Qualcomm Adreno 640 (@685 Mhz):* Includes 384 ALUs with peak 899 GFLOPS compute throughput. It uses optimized GPU libraries for FFT (clFFT [3]) and GeMM (clBLAS [2]). The implementation is compiled with level 3 optimization using the *Adreno OpenCL SDK v1.5* framework.

— *ASIC:* We created equivalent ASIC models for each benchmark and synthesized them at the frequency that meets the required throughput listed in Table 3.

9 Evaluation

9.1 Qualitative Comparison with Prior Works

The design of Canalis is specifically tailored for stream processing in the field of wireless communication. This framework is distinguished by the co-design of its ISA and microarchitecture,

Table 3. Summary of Benchmarks

Name	Kernels	Input Sample Rate	Output Sample Rate
Standalone Kernels	FIR, FFT, GeMM	N/A	N/A
Bluetooth Modulation	Binary Frequency Shift Keying (BFSK), Up-sampling, FIR, Frequency Modulation	1 M sample/s	4 M sample/s
Bluetooth Demodulation	Mixer, CIC, find frequency, FIR, Down-sampling, BFSK Demodulation	16 M sample/s	1 M sample/s
Wi-Fi Modulation	Quadrature Amplitude Modulation (QAM), IFFT, Guard Interval (GI) Insertion	16 M sample/s	20 M sample/s
Wi-Fi Demodulation	Mixer, CIC, GI Removal, Down-sampling FFT, Channel Estimation, Equalization, QAM Demodulation	80 M sample/s	16 M sample/s
Acquisition	Mixer, CIC, autocorrelation, find max index	80 M sample/s	N/A

which are optimized to maximize throughput for prevalent computing patterns within the target domain. Unlike most prior works that depend heavily on load/store operations, Canalis focuses on real-time processing, enhancing performance and significantly reducing latency. Additionally, Canalis introduces an intuitive programming interface that mirrors the natural dataflow of the architecture, thereby simplifying system programming.

Despite its specialized capabilities, Canalis has its limitations, primarily its lack of support for complex branching. This design choice is influenced by the characteristics of common kernels in wireless communication, such as FIR filtering and FFT processing, where computations are deterministic. This simplicity allows Canalis hardware to maintain low overhead and high efficiency. However, it also represents a tradeoff between performance and versatility, limiting its applicability to specific domains. While this focus narrows the scope of its application, it ensures that Canalis excels in its target domain: wireless communication.

Due to its uniqueness, directly comparing Canalis with prior works is challenging. Most existing works employ varied architectures and computational paradigms that do not perfectly align with the specialized design and objectives of Canalis. This discrepancy necessitates an experimental setup that compares Canalis against mobile CPUs, mobile GPUs, and equivalent ASICs, as shown in this section. Consequently, we provide qualitative comparisons between Canalis and various prior works that share some, albeit not all, of the characteristics of Canalis.

ASIC implementations such as AsAP [8, 102, 103], Troung et al. [88], Tran et al. [87], and DAP [17] target the same application domain—wireless communication—as Canalis and are capable of mapping entire workloads onto a computational fabric. The communication between cores varies, with AsAP [8, 102, 103] and Troung et al. [88] employing asynchronous communication, while DAP [17] synchronizes data transfers. Canalis aligns more closely with the former due to its ability to adapt better to real-time data arrival. However, what distinguishes Canalis from these works is their lack of a software stack, which limits their programmability and the effectiveness of mapping workloads. Canalis addresses this issue with a co-designed software stack that enhances accessibility while retaining high performance.

Other prior works, including REVAMP [10], Ultra elastic CGRA [86], and Riptide [34], also provide both hardware and software stacks. Similar to these works, Canalis recognizes the effectiveness of heterogeneous distributions of cores with different functionalities. However, their focus is more on

Table 4. Area and Energy Breakdown of FluxSPU

Name		Area (um^2)	Leakage Power (uW)	Total Power (mW)
Scheduler		1,215.2	7.6	0.39
Instruction Memory		1,425.3	9.5	0.15
PE Datapath		6,352.4	28.3	2.12
Functional Units	A Type	524.6	2.6	0.21
	M Type	3,521.7	19.2	4.45
	D Type	5,426.2	14.9	1.35
	N Type	11,329.1	45.3	7.24
Functional Units Average		7,830.8	31.0	4.92
Configuration Bus		212.4	3.4	0.13
Name		Area (mm^2)	Leakage Power (mW)	Total Power (mW)
PE Average		0.017	0.080	7.709
Switch		0.005	0.030	2.100
FluxSPU Total		2.248	11.586	774.506

general-purpose computation, in contrast to Canalis' domain-specific focus, which allows for more application-specific optimizations as discussed in Section 4. As a result, Canalis achieves a higher operating frequency and throughput (in terms of operations per cycle) for similar kernels mapped onto the fabric. Additionally, both Ultra elastic CGRA [86] and Canalis focus on reducing the II but employ different approaches. The former utilizes circuit-level techniques, while Canalis focuses on architectural and microarchitectural optimizations.

9.2 Power and Area Analysis of FluxSPU

Table 4 shows the power and area breakdown for FluxSPU at 500 MHz. The primary source of both, particularly power, originates from the functional units and datapath. FluxSPU's area is 2.25 mm^2 with a power consumption of 774.5 mW. This aligns with FluxSPU's goal to emulate ASIC-like implementation, where the majority of resources are dedicated to compute units.

9.3 Energy Savings Analysis

For the benchmarks that represent real-world workloads, we emulate real-time stream processing scenarios by measuring the *energy consumption per output data sample* of Canalis and the baselines as they meet the throughput requirements listed in Table 3. This approach follows our emphasis on real-time performance, targeting neither maximum performance nor minimum power, but rather a finely tuned balance of both. For standalone kernels, we measure power consumption across all baselines by maintaining uniform throughput, similar to the requirements of the workloads. To achieve fair emulation, we pipelined memory access, kernel launch, and execution for multiple batches of data on CPU and GPU, similar to Figure 3(a). The results are summarized in Figure 17, where certain benchmarks with unmet throughput requirements by the CPU and GPU are marked in a different color. For those that did not meet the requirement, we report the energy per output sample for the highest throughput they can achieve. Specific failures include the CPU failing WiFi Demodulation (4.6×) and Acquisition (20×), and the GPU failing Acquisition (80×), with both workloads requiring the highest throughput, as shown in Table 3. Acquisition's unique challenge includes a kernel of returning the index of the maximum data, which requires reduction and hinders DLP. Canalis overcomes these challenges and achieves energy savings of 189.8× and 283.9× against CPU and GPU, respectively, and is within 2.4× of equivalent ASICs. The focus on real-time

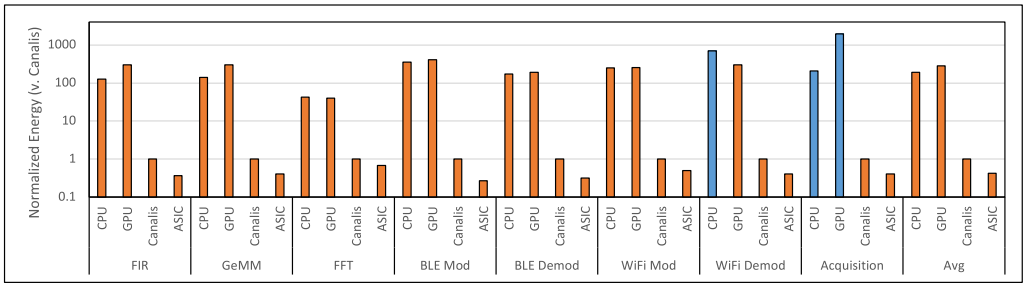


Fig. 17. Energy (vs. Canalis) of CPU, GPU, Canalis, and ASIC across benchmarks. The bars in blue did not meet the throughput requirements. The average energy overhead of Canalis against equivalent ASICs is within 2.4×.

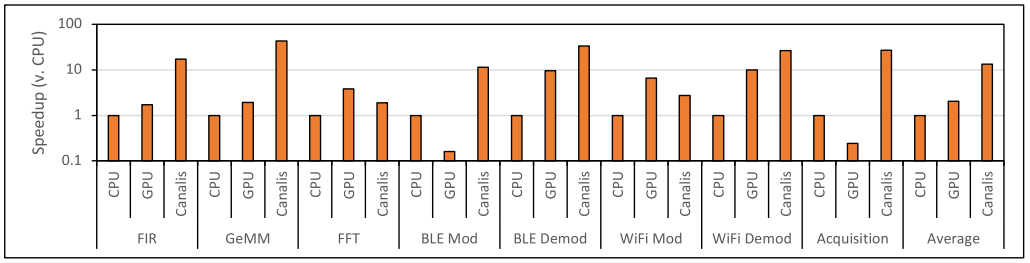


Fig. 18. Speedup (vs. CPU) of CPU, GPU, and Canalis. Canalis achieves speedup of 13.4× and 6.6×, respectively.

requirements maintains not only the necessary throughput but also the conservation of energy. By ensuring "just enough" performance, our approach avoids excessive buffering and computation followed by idleness, a common pitfall. By optimizing for the exact needs of the application without overcompensation, we contribute to both system efficiency and sustainability.

9.4 Speedup against Mobile CPU and GPU

For the benchmarks that represent real-world workloads, we measure the performance speedup of Canalis against the baselines by evaluating the throughput (i.e., amount of data processed in a second). Unlike the energy analysis, where we focus on real-time requirements, here our emphasis is on evaluating the highest compute power each platform can achieve. This method allows us to predict the performance for future standards, as running at the throughput requirement would merely lead to uniform performance across platforms.

The results are summarized in Figure 18. A significant performance drop for the GPU is observed in the benchmarks of Bluetooth Modulation and Acquisition. Bluetooth Modulation contains frequency modulation that forms a feedback loop and Acquisition requires reduction, both of which present challenges for parallel execution. Additionally, the compute intensity (amount of computations needed for each data value) is low for streaming wireless communication workloads compared to the amount of data being processed, due to the large volume of data and common practice of oversampling. Consequently, the difference in performance between CPU and GPU is not as significant for individual kernels, as the compute intensity is smaller, so the benefits of having more compute power are not as prominent. When compared against CPU and GPU, Canalis achieves an average speedup of 13.4× and 6.6×, respectively. After adjusting for technology [82], the area normalized speedups are 29.6× and 130.5×, respectively.

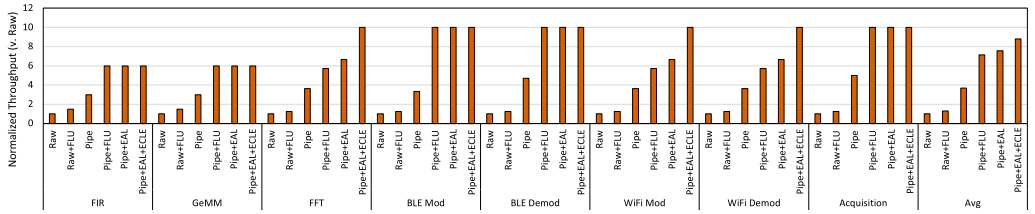


Fig. 19. Throughput comparisons of different optimizations, including Instruction Pipeline (Pipe), Embedded Atomic Instruction Loop Count (EAL), and Embedded Composite Loop Evaluation (ECLE). While loop unrolling is not needed when applied EAL and ECLE, we still included scenarios with Full Loop Unrolling (FLU).

This performance analysis not only provides insight into the full potential of Canalis but also allows us to forecast how it might perform under future standards such as 6G.

9.5 ISA Optimization and Throughput Improvement

Figure 19 illustrates the normalized throughput across various benchmarks with incremental optimizations as described in Section 4. We included **Full Loop Unrolling (FLU)** in our baselines, which assumes infinite instruction memory. Our method, which embeds both Composite Instruction Loop Evaluation and Atomic Instruction Loop Counts within instructions, negates the need for loop unrolling. The analysis reveals that: (a) Throughput without instruction pipelining is hampered by multi-cycle operations such as rotation. (b) FLU with instruction pipelining can match optimized performance but is impractical. (c) Embedding Composite Instruction Loop Evaluation offers minimal gains over Atomic Instruction Loop Counts embedding due to spatial distribution of operations and the ability to hide explicit Composite Instruction Loop Evaluation overhead with Atomic Instruction Loops with higher loop count. Thus, our approach substantially enhances performance without the complexity of traditional methods.

10 Related Work

This section discusses and compares Canalis with related works other than those covered in previous sections. These works include programmable wireless communication accelerators, spatial architectures, dataflow architectures, and vector machines.

Programmable Wireless Communication Accelerator. Various programmable implementations have been developed, targeting specific kernels [13, 70, 101, 105] or entire workloads [8, 55, 67, 88, 99, 102, 103]. Most of these works provide high-performance and efficiency but do not offer a software stack that enables ease of programming and scalability. Unlike the end-to-end frameworks discussed in previous sections, these particular implementations primarily focus on hardware aspects, leaving a gap in software support. Canalis extends beyond this limitation by integrating a robust software stack, enhancing programmability, and paving the way for applications outside of wireless communication.

Spatial Architecture. Spatial architectures are a compelling approach for accelerating computations across a diverse range of domains, some serving as standalone structures [9, 15, 16, 22, 23, 28, 31, 44, 53, 56, 57, 59, 62–64, 72, 77, 78, 80] while others are integrated into the datapaths of processors [11, 37, 38, 98]. Canalis is versatile and can adapt to both. As a standalone architecture, Canalis can perform near frontend executions. On the other hand, its straightforward push-pop interface allows seamless integration within more sophisticated processors or SoC. What differentiates Canalis is its direct execution model (Section 2) that enables real-time data processing without intermediate data storage. The co-design of CER provides a concise representation of the programming model.

Some works target spatial architecture generation [14, 19, 33, 54, 60, 71, 93] while others focus on compilers for spatial architectures [42, 58]. In contrast, the software stack provided by Canalis including the CER does not generate hardware nor does it compile programs into a hardware description language; its sole purpose is to simplify the programming of the FluxSPU architecture.

Dataflow Architectures and Vector Machines. Dataflow architectures [7, 12, 26, 52, 83] and hybrid dataflow Von-Neumann architectures [66, 100] launch execution upon data arrival, enabling dynamic scheduling, while vector machines [20, 35, 49, 75] utilize DLP. Canalis is closer to dataflow architectures and adapts to indeterministic data arrival, similar to dataflow machines, but avoids the complexities of tag-matching by leveraging application-specific characteristics, ISA optimization, and pipelined FU reordering (Section 5). This ensures that the sequence of data is maintained without additional overhead. Compared to vector machines, Canalis primarily utilizes pipeline parallelisms, only resorting to DLP when data arrive concurrently (e.g., MIMO systems).

Streaming Architecture. Streaming optimizations can be broadly divided into two approaches. The first approach [46, 65, 68, 91, 92, 95] focuses on optimizing memory access patterns, leveraging micro-architecture and compiler strategies to form coherent data streams from memory. Although efficient in many scenarios, this approach might not be suited for unpredictable, real-time data processing (Section 2). The second approach, which is closer to Canalis, is designed to handle real-time incoming data streams [25, 73, 81], processing them directly as they arrive.

Language, ISA, and Compiler Support. Language and ISA support play a critical role in bridging the gap between hardware capabilities and application requirements. Triggered Instructions [69] present a control paradigm for spatial architectures, where scheduling is triggered by data arrival. Though similar to Canalis, it prioritizes flexibility, whereas Canalis emphasizes throughput since the order of operations is deterministic. Additionally, there have been significant efforts [27, 36, 85] to provide language, ISA extension, and compiler support for streaming applications. These works focus on augmenting traditional compute platforms, different from Canalis, which targets a new architecture. Furthermore, some prior research [47, 48, 74] has focused on creating domain-specific languages for describing spatial architectures and generating hardware. Canalis takes a different approach with a co-designed software stack that simplifies the representation of computations using CER and maps computation on the proposed architecture. Canalis' software stack does not generate hardware, but focuses on assisting the users program FluxSPU.

11 Conclusion

In the domain of wireless communication, conventional computing platforms struggle to meet the demands of real-time stream processing. We present Canalis, a throughput-optimized framework designed to achieve efficient stream processing with minimized energy consumption. It includes a programmable spatial architecture, FluxSPU, co-designed with a software stack including a programming interface, CER, and a mapper. Canalis simplifies the programming process, achieving an optimal balance between power consumption and performance.

We evaluate Canalis using eight wireless communication benchmarks. The results show its superior performance and efficiency over existing mobile CPU and GPU while being close (2.4×) to individual equivalent ASIC implementations. Compared to state-of-the-art libraries on a mobile CPU and a mobile GPU, Canalis achieves an average speedup of 13.4× and 6.6×, area normalized speedup of 29.6× and 130.5×, and energy savings of 189.8× and 283.9×, respectively.

References

[1] ARM-software. 2023. *Arm Optimized Routines*. Retrieved from https://github.com/ARM-software/optimized-routines
[2] clMathLibraries. 2017. *clBLAS*. Retrieved from https://github.com/clMathLibraries/clBLAS
[3] clMathLibraries. 2016. *clFFT*. Retrieved from https://github.com/clMathLibraries/clFFT

[4] ARM-software. 2024. *CMSIS-DSP*. Retrieved from https://github.com/ARM-software/CMSIS-DSP

[5] marton78. 2022. PFFFT: A pretty fast FFT and fast convolution with PFFASTCONV. Retrieved from https://github.com/marton78/pffft

[6] GitHub. 2024. XNNPACK. Retrieved from https://github.com/google/XNNPACK

[7] Arvind and R. S. Nikhil. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers* 39, 3 (1990), 300–318. DOI: https://doi.org/10.1109/12.48862

[8] B. M. Baas. 2003. A parallel programmable energy-efficient architecture for computationally-intensive DSP systems. In *Proceedings of the 37th Asilomar Conference on Signals, Systems & Computers*, Vol. 2, 2185–2189. DOI: https://doi.org/10.1109/ACSSC.2003.1292368

[9] James Balfour, William Dally, David Black-Schaffer, Vishal Parikh, and JongSoo Park. 2008. An energy-efficient processor architecture for embedded systems. *IEEE Computer Architecture Letters* 7, 1 (2008), 29–32. DOI: https://doi.org/10.1109/L-CA.2008.1

[10] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: A systematic framework for heterogeneous CGRA realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM, New York, NY, 918–932. DOI: https://doi.org/10.1145/3503222.3507772

[11] M. Bedford Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. 2003. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA-9 '03)*, 341–353. DOI: https://doi.org/10.1109/HPCA.2003.1183551

[12] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. 1995. Cyclo-static data flow. In *Proceedings of the 1995 International Conference on Acoustics, Speech, and Signal Processing*, Vol. 5, 3255–3258. DOI: https://doi.org/10.1109/ICASSP.1995.479579

[13] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. 2017. KiloCore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits* 52, 4 (2017), 891–902. DOI: https://doi.org/10.1109/JSSC.2016.2638459

[14] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. 2004. Spatial computation. *SIGOPS Operating Systems Review* 38, 5 (Oct. 2004), 14–26. DOI: https://doi.org/10.1145/1037949.1024396

[15] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra. In *Proceedings of the 2022 IEEE Symposium on VLSI Technology and Circuits*, 70–71. DOI: https://doi.org/10.1109/VLSITechnologyandCir46769.2022.9830509

[16] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. 2000. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proceedings of the Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications (FPL '00)*. Springer-Verlag, Berlin, 605–614.

[17] Kuan-Yu Chen, Chi-Sheng Yang, Yu-Hsiu Sun, Chien-Wei Tseng, Morteza Fayazi, Xin He, Siying Feng, Yufan Yue, Trevor Mudge, Ronald Dreslinski, Hun-Seok Kim, and David Blaauw. 2022. A 507 GMACs/J 256-core domain adaptive systolic-array-processor for wireless communication and linear-algebra kernels in 12nm FINFET. In *Proceedings of the 2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, 202–203. DOI: https://doi.org/10.1109/VLSITechnologyandCir46769.2022.9830330

[18] Longlong Chen, Jianfeng Zhu, Yangdong Deng, Zhaoshi Li, Jian Chen, Xiaowei Jiang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. An elastic task scheduling scheme on coarse-grained reconfigurable architectures. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (2021), 3066–3080. DOI: https://doi.org/10.1109/TPDS.2021.3084804

[19] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. 2018. An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '18)*, 55–67. DOI: https://doi.org/10.1109/MICRO.2018.00014

[20] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The Reconfigurable Streaming Vector Processor (RSVPTM). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO '36)*. IEEE Computer Society, 141.

[21] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of the 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC '14)*. IEEE, 1–6.

[22] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 9–16. DOI: https://doi.org/10.1109/FCCM.2014.12

[23] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling. 1998. Specifying and compiling applications for RaPiD. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, 116–125. DOI: https://doi.org/10.1109/FPGA.1998.707889

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

[25] A. DeHon. 1996. DPGA utilization and application. In *Proceedings of the 4th International ACM Symposium on Field-Programmable Gate Arrays*, 115–121. DOI: https://doi.org/10.1109/FPGA.1996.242438

[26] Jack B. Dennis and David P. Misunas. 1974. A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Computer Architecture News* 3, 4 (Dec. 1974), 126–132. DOI: https://doi.org/10.1145/641675.642111

[27] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. 2021. Unlimited vector extension with data streaming support. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA '21)*. IEEE Press, 209–222. DOI: https://doi.org/10.1109/ISCA52012.2021.00025

[28] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. 1996. RaPiD - Reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL '96)*. Springer-Verlag, Berlin, 126–135.

[29] M. M. Fernandes, J. Llosa, and N. Topham. 1999. Distributed modulo scheduling. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 130–134. DOI: https://doi.org/10.1109/HPCA.1999.744349

[30] Adi Fuchs and David Wentzlaff. 2019. The accelerator wall: Limits of chip specialization. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*. IEEE, 1–14.

[31] Mingyu Gao and Christos Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA '16)*, 126–137. DOI: https://doi.org/10.1109/HPCA.2016.7446059

[32] Mario Garrido, J. Grajal, M. A. Sanchez, and Oscar Gustafsson. 2013. Pipelined radix-2^k feedforward FFT architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21, 1 (2013), 23–32. DOI: https://doi.org/10.1109/TVLSI.2011.2178275

[33] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: An ultra-low-power, energy-minimal CGRA-generation framework and architecture. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 1027–1040. DOI: https://doi.org/10.1109/ISCA52012.2021.00084

[34] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. RipTide: A programmable, energy-minimal dataflow compiler and architecture. In *Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 546–564. DOI: https://doi.org/10.1109/MICRO56248.2022.00046

[35] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. MANIC: A vector-dataflow architecture for ultra-low-power embedded systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. ACM, New York, NY, 670–684. DOI: https://doi.org/10.1145/3352460.3358277

[36] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. 2002. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X '02)*. ACM, New York, NY, 291–303. DOI: https://doi.org/10.1145/605397.605428

[37] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51. DOI: https://doi.org/10.1109/MM.2012.51

[38] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 503–514.

[39] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, 37–47. DOI: https://doi.org/10.1145/1815961.1815968

[40] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. DOI: https://doi.org/10.1109/TSSC.1968.300136

[41] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient execution of memory access phases using dataflow specialization. *ACM SIGARCH Computer Architecture News* 43, 3S (Jun. 2015), 118–130. DOI: https://doi.org/10.1145/2872887.2750390

[42] Olivia Hsu, Alexander Rucker, Tian Zhao, Kunle Olukotun, and Fredrik Kjolstad. 2022. Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture. DOI: https://doi.org/10.48550/ARXIV.2211.03251

[43] K. T. Johnson, A. R. Hurson, and B. Shirazi. 1993. General-purpose systolic arrays. *Computer* 26, 11 (1993), 20–31. DOI: https://doi.org/10.1109/2.241423

[44] Anthony Mark Jones and Mike Butts. 2006. TeraOPS hardware: A new massively-parallel MIMD computing fabric IC. In *Proceedings of the 2006 IEEE Hot Chips 18 Symposium (HCS '07)*, 1–15. DOI: https://doi.org/10.1109/HOTCHIPS.2006.7477853

[45] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC '17)*. ACM, New York, NY, Article 45, 6 pages. DOI: https://doi.org/10.1145/3061639.3062262

[46] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. 2001. Imagine: Media processing with streams. *IEEE Micro* 21, 2 (2001), 35–46. DOI: https://doi.org/10.1109/40.918001

[47] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, 115–127. DOI: https://doi.org/10.1109/ISCA.2016.20

[48] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A language and compiler for application accelerators. *ACM SIGPLAN Notices* 53, 4 (Jun. 2018), 296–311. DOI: https://doi.org/10.1145/3296979.3192379

[49] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. 2004. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture,* 52–63. DOI: https://doi.org/10.1109/ISCA.2004.1310763

[50] H. T. Kung. 1988. Systolic communication. In *[1988] Proceedings of the International Conference on Systolic Arrays*, 695–703. DOI: https://doi.org/10.1109/ARRAYS.1988.18106

[51] H. T. Kung. 1982. Why systolic architectures? *Computer* 15 (1982), 37–46.

[52] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proceedings of the IEEE* 75, 9 (1987), 1235–1245. DOI: https://doi.org/10.1109/PROC.1987.13876

[53] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. 1998. Space-time scheduling of instruction-level parallelism on a raw machine. *ACM SIGPLAN Notices* 33, 11 (Oct. 1998), 46–57. DOI: https://doi.org/10.1145/291006.291018

[54] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *2011 38th Annual International Symposium on Computer Architecture (ISCA '11)*, 129–140.

[55] Yuan Lin, Hyunseok Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. 2006. SODA: A low-power architecture for software radio. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA '06)*, 89–101. DOI: https://doi.org/10.1109/ISCA.2006.37

[56] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. 2014. CGPA: Coarse-grained pipelined accelerators. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. ACM, New York, NY, 1–6. DOI: https://doi.org/10.1145/2593069.2593105

[57] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, 161–171. DOI: https://doi.org/10.1109/ISCA.2000.854387

[58] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology (FPT '02),* 166–173. DOI: https://doi.org/10.1109/FPT.2002.1188678

[59] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proceedings of the Field Programmable Logic and Application, 13th International Conference (FPL '03).* Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa (Eds.), Lecture Notes in Computer Science, Vol. 2778, Springer, 61–70. DOI: https://doi.org/10.1007/978-3-540-45234-8_7

[60] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A. Horowitz, Pat Hanrahan, and Priyanka Raina. 2023. APEX: A framework for automated processing element design space exploration using frequent subgraph analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS '23),* Vol. 3, ACM, New York, NY, 33–45. DOI: https://doi.org/10.1145/3582016.3582070

[61] O. Menzilcioglu, H. T. Kung, and S. W. Song. 1989. Comprehensive evaluation of a two-dimensional configurable array. In *[1989] Proceedings of the 19th International Symposium on Fault-Tolerant Computing*. Digest of Papers, 93–100. DOI: https://doi.org/10.1109/FTCS.1989.105549

[62] Mirsky and DeHon. 1996. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 157–166. DOI: https://doi.org/10.1109/FPGA.1996.564808

[63] Takashi Miyamori and Kunle Olukotun. 1998. REMARC: Reconfigurable multimedia array coprocessor (Abstract). In *Proceedings of the 1998 ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA '98)*. Jason Cong and Sinan Kaptanoglu (Eds.), ACM, 261. DOI: https://doi.org/10.1145/275107.275164

[64] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. 2001. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '34)*. IEEE Computer Society, 40–51.

[65] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, 416–429.

[66] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous von Neumann/dataflow execution models. *ACM SIGARCH Computer Architecture News* 43, 3S (Jun. 2015), 298–310. DOI: https://doi.org/10.1145/2872887.2750380

[67] John Oliver, Ravishankar Rao, Paul Sultana, Jedidiah Crandall, Erik Czernikowski, Leslie W. Jones IV, Diana Franklin, Venkatesh Akella, and Frederic T. Chong. 2004. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*. IEEE Computer Society, 150.

[68] Sérgio Paiágua, Frederico Pratas, Pedro Tomás, Nuno Roma, and Ricardo Chaves. 2013. HotStream: Efficient data streaming of complex patterns to multiple accelerating kernels. In *Proceedings of the 2013 25th International Symposium on Computer Architecture and High Performance Computing*, 17–24. DOI: https://doi.org/10.1109/SBAC-PAD.2013.17

[69] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered instructions: A control paradigm for spatially-programmed architectures. *ACM SIGARCH Computer Architecture News* 41, 3 (Jun. 2013), 142–153. DOI: https://doi.org/10.1145/2508148.2485935

[70] Ardavan Pedram, Andreas Gerstlauer, and Robert A. van de Geijn. 2014. Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator. *IEEE Transactions on Computers* 63, 8 (2014), 1854–1867. DOI: https://doi.org/10.1109/TC.2014.2315627

[71] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating configurable hardware from parallel patterns. *ACM SIGPLAN Notices* 51, 4 (Mar. 2016), 651–665. DOI: https://doi.org/10.1145/2954679.2872415

[72] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Plasticine: A reconfigurable accelerator for parallel patterns. *IEEE Micro* 38, 3 (2018), 20–31.

[73] G. Quenot, C. Coutelle, J. Serot, and B. Zavidovique. 1993. A wavefront array processor for on the fly processing of digital video streams. In *Proceedings of the International Conference on Application Specific Array Processors (ASAP '93)*, 101–108. DOI: https://doi.org/10.1109/ASAP.1993.397124

[74] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (Jun. 2013), 519–530. DOI: https://doi.org/10.1145/2499370.2462176

[75] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A vector RDA for sparsity. In *MICRO-54: Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. ACM, New York, NY, 1022–1035. DOI: https://doi.org/10.1145/3466752.3480047

[76] Richard M. Russell. 1978. The CRAY-1 computer system. *Communications of the ACM* 21, 1 (Jan. 1978), 63–72. DOI: https://doi.org/10.1145/359327.359336

[77] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. 2004. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. *ACM Transactions on Architecture and Code Optimization (TACO)* 1, 1 (Mar. 2004), 62–93. DOI: https://doi.org/10.1145/980152.980156

[78] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M. S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. 2006. Distributed

microarchitectural protocols in the TRIPS prototype processor. In *MICRO '06*. IEEE Computer Society, 480–491. DOI: https://doi.org/10.1109/MICRO.2006.19

[79] Mahadev Satyanarayanan, Nathan Beckmann, Grace A. Lewis, and Brandon Lucia. 2021. The role of edge offload for hardware-accelerated mobile devices. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications (HotMobile '21)*. ACM, New York, NY, 22–29. DOI: https://doi.org/10.1145/3446382.3448360

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

[81] Sander Smets, Toon Goedemé, Anurag Mittal, and Marian Verhelst. 2019. 2.2 A 978GOPS/W flexible streaming processor for real-time image processing applications in 22 nm FDSOI. In *Proceedings of the 2019 IEEE International Solid- State Circuits Conference (ISSCC '19)*, 44–46. DOI: https://doi.org/10.1109/ISSCC.2019.8662346

[82] A. Stillmaker and B. Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal* 58 (2017), 74–81. Retrieved from http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/

[83] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. 2007. The wavescalar architecture. *ACM Transactions on Computer Systems* 25, 2, Article 4 (May 2007), 54 pages. DOI: https://doi.org/10.1145/1233307.1233308

[84] Cheng Tan, Nicolas Bohm Agostini, Tong Geng, Chenhao Xie, Jiajia Li, Ang Li, Kevin J. Barker, and Antonino Tumeo. 2022. DRIPS: Dynamic rebalancing of pipelined streaming applications on CGRAs. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*, 304–316. DOI: https://doi.org/10.1109/HPCA53966.2022.00030

[85] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, Berlin, 179–196.

[86] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-elastic CGRAs for irregular loop specialization. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*, 412–425. DOI: https://doi.org/10.1109/HPCA51647.2021.00042

[87] Anh T. Tran, Dean N. Truong, and Bevan M. Baas. 2008. A complete real-time 802.11a baseband receiver implemented on an array of programmable processors. In *Proceedings of the 2008 42nd Asilomar Conference on Signals, Systems and Computers*, 165–170. DOI: https://doi.org/10.1109/ACSSC.2008.5074384

[88] Dean N. Truong, Wayne H. Cheng, Tinoosh Mohsenin, Zhiyi Yu, Anthony T. Jacobson, Gouri Landge, Michael J. Meeuwsen, Christine Watnik, Anh T. Tran, Zhibin Xiao, Eric W. Work, Jeremy W. Webb, Paul V. Mejia, and Bevan M. Baas. 2009. A 167-processor computational platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits* 44, 4 (2009), 1130–1144. DOI: https://doi.org/10.1109/JSSC.2009.2013772

[89] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys* 51, 4, Article 72 (Jul. 2018), 39 pages. DOI: https://doi.org/10.1145/3193827

[90] Markus Voelter. 2021. Programming vs. that thing subject matter experts do. In *Proceedings of the Leveraging Applications of Formal Methods, Verification and Validation: 10th International Symposium on Leveraging Applications of Formal Methods (ISoLA '21)*. Springer-Verlag, Berlin, 414–425. DOI: https://doi.org/10.1007/978-3-030-89159-6_26

[91] Zhengrong Wang and Tony Nowatzki. 2019. Stream-based memory access specialization for general purpose processors. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, 736–749. DOI: https://doi.org/10.1145/3307650.3322229

[92] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. 2022. Near-stream computing: General and transparent near-cache acceleration. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '22)*. IEEE, 331–345. DOI: https://doi.org/10.1109/HPCA53966.2022.00032

[93] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing programmable spatial accelerators. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*, 268–281. DOI: https://doi.org/10.1109/ISCA45697.2020.00032

[94] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA '20)*, 703–716. DOI: https://doi.org/10.1109/HPCA47549.2020.00063

[95] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. 2019. CASCADE: High throughput data streaming via decoupled access-execute CGRA. *ACM Transactions on Embedded Computing Systems* 18, 5s, Article 50 (Oct. 2019), 26 pages. DOI: https://doi.org/10.1145/3358177

[96] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. 2016. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *Proceedings of the 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS '16)*, 235–244. DOI: https://doi.org/10.1109/SAMOS.2016.7818353

[97] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 4 (Apr. 2009), 65–76. DOI: https://doi.org/10.1145/1498765.1498785

[98] Wittig and Chow. 1996. OneChip: An FPGA processor with reconfigurable logic. In *1996 Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 126–135. DOI: https://doi.org/10.1109/FPGA.1996.564773

[99] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. 2009. AnySP: Anytime anywhere anyway signal processing. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, 128–139. DOI: https://doi.org/10.1145/1555754.1555773

[100] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. 2014. Hybrid dataflow/von-Neumann architectures. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1489–1509. DOI: https://doi.org/10.1109/TPDS.2013.125

[101] A. K. Yeung and J. M. Rabaey. 1995. A 2.4 GOPS data-driven reconfigurable multiprocessor IC for DSP. In *Proceedings of the International Solid-State Circuits Conference (ISSCC '95)*, 108–109. DOI: https://doi.org/10.1109/ISSCC.1995.535451

[102] Zhiyi Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. 2006. An asynchronous array of simple processors for dsp applications. In *Proceedings of the 2006 IEEE International Solid State Circuits Conference*. Digest of Technical Papers, 1696–1705. DOI: https://doi.org/10.1109/ISSCC.2006.1696225

[103] Zhiyi Yu, Michael J. Meeuwsen, Ryan W. Apperson, Omar Sattari, Michael Lai, Jeremy W. Webb, Eric W. Work, Dean Truong, Tinoosh Mohsenin, and Bevan M. Baas. 2008. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits* 43, 3 (2008), 695–705. DOI: https://doi.org/10.1109/JSSC.2007.916616

[104] Fang-Li Yuan and Dejan Marković. 2014. A 13.1GOPS/mW 16-core processor for software-defined radios in 40nm CMOS. In *Proceedings of the 2014 Symposium on VLSI Circuits*. Digest of Technical Papers, 1–2. DOI: https://doi.org/10.1109/VLSIC.2014.6858388

[105] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. M. Rabaey. 2000. A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing. *IEEE Journal of Solid-State Circuits* 35, 11 (2000), 1697–1704. DOI: https://doi.org/10.1109/4.881217

Received 20 January 2024; revised 18 July 2024; accepted 28 August 2024