A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms

Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, Tony Nowatzki
University of California, Los Angeles
Los Angeles, California, US
{jian.weng, sihao, seanzw, vidushi.dadu, tjn}@cs.ucla.edu

***Abstract*—Dense linear algebra kernels are critical for wireless, and the oncoming proliferation of 5G only amplifies their importance. Due to the *inductive* nature of many such algorithms, parallelism is difficult to exploit: parallel regions have fine-grain producer/consumer interaction with iteratively changing dependence distance, reuse rate, and memory access patterns. This causes a high overhead both for multi-threading due to fine-grain synchronization, and for vectorization due to the non-rectangular iteration domains. CPUs, DSPs, and GPUs perform order-of-magnitude below peak.**

Our insight is that if the nature of inductive dependences and memory accesses were explicit in the hardware/software interface, then a spatial architecture could efficiently execute parallel code regions. To this end, we first extend the traditional dataflow model with first class primitives for inductive dependences and memory access patterns (streams). Second, we develop a hybrid spatial architecture combining systolic and dataflow execution to attain high utilization at low energy and area cost. Finally, we create a scalable design through a novel vector-stream control model which amortizes control overhead both in time and spatially across architecture lanes.

We evaluate our design, REVEL, with a full stack (compiler, ISA, simulator, RTL). Across a suite of linear algebra kernels, REVEL outperforms equally-provisioned DSPs by 4.6×—37×. Compared to state-of-the-art spatial architectures, REVEL is mean 3.4× faster. Compared to a set of ASICs, REVEL is only 2× the power and half the area.

***Keywords*-Spatial Architecture; Reconfigurable Accelerator; Software/Hardware Codesign; Digital Signal Processor**

I. Introduction

Dense linear algebra kernels, like matrix factorization, multiplication, decomposition and FFT, have for decades been the computational workhorses of signal processing across standards, specifications, and device settings. The oncoming proliferation of 5G wireless is only further pushing the computational demands, both in performance and energy efficiency [1], [2]. Driven by needs of higher capacity [3] and applications like augmented and virtual reality [4], new standards will require signal processing at more than an order-of-magnitude higher throughput and lower latency. Relying on fixed-function ASICs alone has many drawbacks: design effort, extra on-chip area, and lack of flexibility; these are especially relevant for wireless, where standards are continually changing (4G,LTE,5G,etc).

Despite their ubiquity, many important dense matrix operations are far from trivial to parallelize and compute at high hardware efficiency on programmable architectures. Figure 1 shows the throughput of a modern CPU, DSP, and GPU running common DSP algorithms, compared to an ideal ASIC with the same computation resources as the DSP (methodology in Section VII). An order of magnitude of performance is lost.

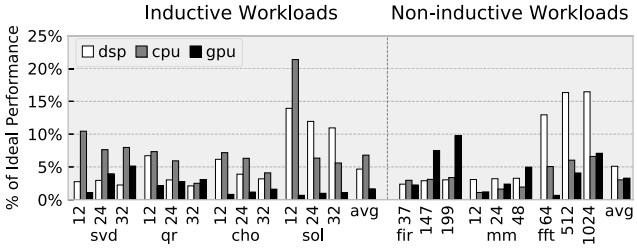

Fig. 1: Percent ideal performance. CPU: Xeon 4116–Intel MKL, DSP: TI C6678–TI DSPLIB, GPU: TITAN V–NVIDIA libraries

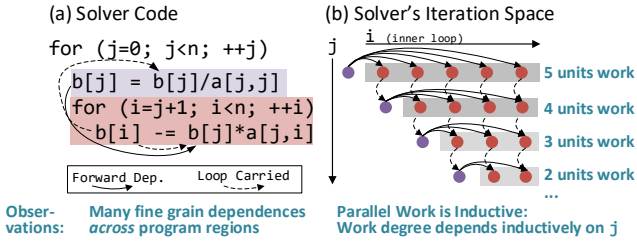

Fig. 2: Inductive Workload Example: Triangular Linear Solver

The primary challenge comes from the commonly *inductive* nature of the parallelism present, coupled with the commonly small matrices in this domain. Informally, many DSP algorithms are inductive in that they build on intermediate computations iteratively and at a fine-grain. A simple inductive workload is the triangular solver in Figure 2(a), which updates a shrinking subset of a vector repeatedly based on a division in the outer loop. Figure 2(b) shows the dependences between iterations of the inner and outer loop. This contrasts sharply with a non-inductive workload like dense matrix multiplication, where all parallelism is available immediately and there are no dependences (besides reduction).

Though simple, this workload demonstrates two common properties: 1. fine-grain producer/consumer relationships between program regions and 2. many aspects of the program execution are a function of an outer loop induction variable – in this case that includes the amount of parallel work, the rate of data reuse, the dependence distance, and the memory access pattern. Hence, we refer to any aspect of execution that varies with an outer-loop induction variable as *inductive*.

CPUs, GPUs, and DSPs rely heavily on vectorization and multithreading; both are hindered by inductive behavior. Vectorization has a high overhead because the total iterations of parallel work changes inductively, therefore the iteration count

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00063

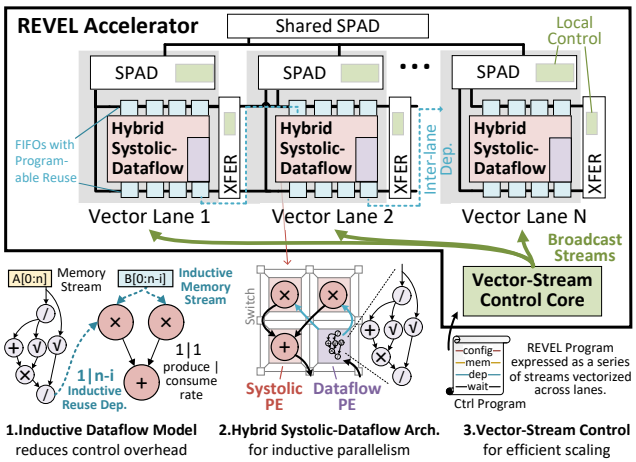

Fig. 3: Proposed Architecture and Three Novel Aspects

will frequently not be divisible by the vector length, causing under-utilization. Frequent dependences between program regions due to inductive updates make thread-synchronization overhead too high.

Dataflow models and their corresponding architectures [5]–[7] are promising because they can express dependences with inductive patterns, enabling parallelism of regions even with fine-grain dependences. Spatial dataflow architectures [8]–[12] help make dataflow architectures practical and energy-efficient. However, we demonstrate quantitatively that these are still factors under peak performance due to either control overheads or serialization.

Goal and Approach: Our goal is to create the next-generation programmable DSP architecture, capable of both inductive and non-inductive parallelism, by leveraging principles of dataflow, spatial architectures and specialization. We summarize the proposed design, REVEL (Reconfigurable Vector Lanes), and novel aspects of this work (see Figure 3):

First, we create an *inductive dataflow* execution model, which extends dataflow execution with first class primitives for inductive dependences and memory access streams. This specialization enables effective vectorization through implicit masking based on the relationship of stream to vector length, and allows expression of inductive behavior at low control overhead.

Second, we develop a *hybrid systolic/dataflow* architecture, which can use efficient *systolic* execution for inner-loops regions that must execute at a high rate, and a more flexible and *tagged dataflow* execution for other operations off the critical path. This enables parallelism across program regions at high utilization with low hardware overhead.

Third, we scale the design by using lanes, each having the hybrid architecture and local scratchpad (SPAD). To control the operation and interaction of all lanes with low overhead, we develop the vector-stream control model, which enables a single core to control program regions with fine-grain dependences mapped across lanes. This is a generalization of the stream-dataflow ISA [13] to enable amortization of control through time (streams) and space (vector lanes).

Our contributions are:

- Identification of fine-grain inductive behavior as a primary challenge for many dense linear algebra kernels.
- Taxonomy of spatial architectures which explains their tradeoffs for DSP workloads.
- *Inductive dataflow* execution model which expresses inductive memory and dependences as a first-class primitive to reduce control overhead and make vectorization profitable.
- A novel *hybrid systolic-dataflow* architecture to specialize for the exposed inductive parallelism.
- REVEL accelerator design and its novel *vector-stream control* model which amortizes control in time and space.
- Full stack implementation, open-source compiler, simulator[1], and comparison against state-of-the art DSPs, CPUs, GPUs and spatial architectures.

Results: A single 1.25GHz REVEL unit can outperform a 2.1GHz OOO core running highly-optimized MKL code on DSP workloads by mean 9.6×, with an area normalized speedup of 1089×. Compared to a DSP, REVEL achieves between 4.6×-37× lower latency, It is half the area of an equivalent set of ASICs and within 2× average power.

Paper Organization: We characterize the workloads and challenges for existing architectures in Section II, and the promise and challenges for spatial architectures in Section III. We then develop inductive dataflow and the hybrid architecture in Section IV. Our accelerator, REVEL, and its vector-stream control model is described in Section V, and its compiler is in Section VI. Finally, we cover methodology, evaluation and additional related work in Sections VII, VIII, and IX.

II. Workload Characterization

In this section, we explain how the inductive nature of many linear algebra workloads creates parallelism that is difficult to exploit. We start with workload background, then discuss inductive workload characteristics and why they hamper traditional architectures.

A. Why these particular DSP workloads?

Figure 4 shows typical 4G/5G transmitter/receiver stages:

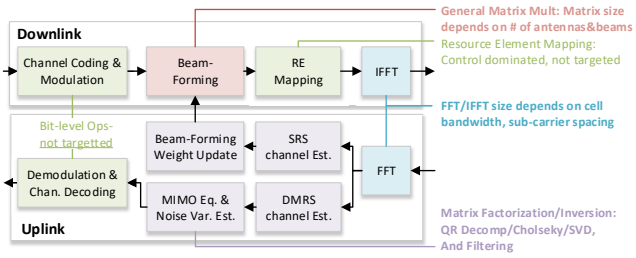

Fig. 4: Typical 4G/5G Transmitter/Receiver Pipeline

Kernels we do not target: Channel coding and modulation involve mostly bit-level arithmetic. RE mapping is a short resource allocation phase which is not computation intense.

[1] Open-source release of simulator, compiler, and workload implementation: https://github.com/PolyArch/stream-specialization-stack

Kernels we target: The Beamforming stage involves mostly GEMM, coming from spatial signal filtering [14]. Filters and FFT of several varieties are also common [15]–[17].

Challenging inductive workloads are mostly within MIMO equalization and channel estimation. These include Singular Value Decomp., used for noise reduction [18], QR Decomposition, used for signal detection for (MIMO) [19], and Cholesky Decomposition and Solver used in channel estimation and equalization [20], [21].

Why are matrices small?: The parameters often depend on the number of antennas and beams (in the range of 12-32 would be common) [22].

B. Inductive Workload Properties

To explain the characteristics and challenges of inductive workloads, we use Cholesky decomposition as a representative example. Figure 5(a) shows the algorithm with inter-region dependences annotated. Cholesky contains a point, a vector, and a matrix computation region. We use the term *region* to refer to a set of statements at one loop-nest level.

What makes a workload "Inductive"?: The inductive property indicates that certain aspects of the execution iteratively change based on an outer loop induction variable. In this example, loop trip counts all depend on outer-loop variables, creating the triangular iteration space in Figure 5(b), which is also shrinking on each outer k loop iteration. Being inductive affects all aspects of Cholesky's execution:

- **Inductive Parallelism** The amount of parallelism that is made available by completing various statements iteratively changes. For example, the amount of parallel work in the vector and matrix regions triggered by the production of `inv` and `invqsrt` is iteratively less.
- **Inductive Dependences** The dependence distance or number of times a value is reused iteratively changes. For example, `inv` is reused `n-k` times in the vector region.
- **Inductive Memory Access** The maximum stride of contiguous memory changes iteratively. For example, the access to array `a` within the matrix region has contiguous length `n-j` (assuming row-major order).

Parallelization Challenges: Both vectorization and multi-threading are hampered by inductive behaviors.

Vectorization would be useful for parallelizing the work within each region, but this is hampered by inductive parallelism. To explain, vectorization is accomplished by choosing sets of iterations (size of the vector length) to combine into their vector equivalents, where ideally they have contiguous memory access - this is known as tiling. An inductive iteration space cannot be tiled perfectly into sets with contiguous memory access.

In the Cholesky example, the iterations within the inner loop (rows of the matrix region within the figure), have contiguous access. Figure 5 shows a possible tiling with vector-length of four. The common way to execute remaining iterations is with a scalar loop – scalar iterations quickly dominate due to

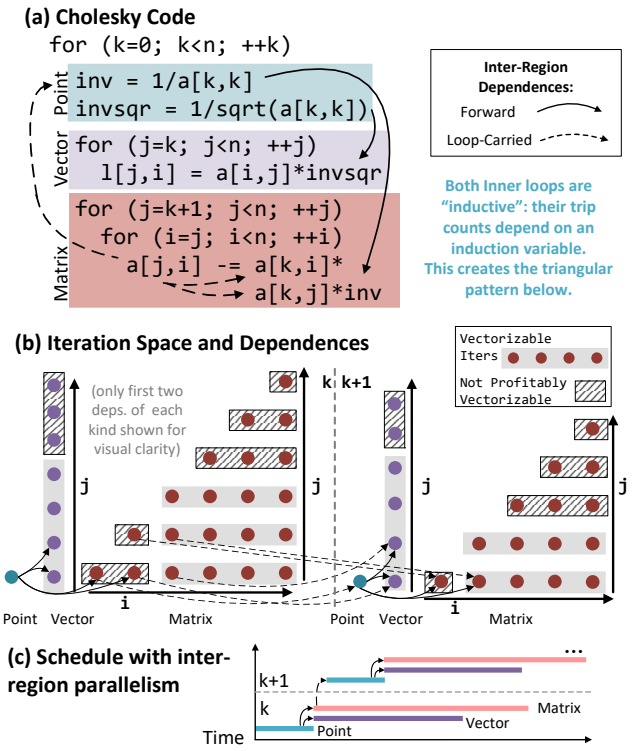

Fig. 5: Inductive Workload Example: Cholesky

Amdahl's law. For the matrix region with $n = 24$ and vector-width of 8, only 55% of iterations can be vectorized, for a maximum speedup of less than 2×. With vector size 16 the maximum speedup drops to 1.2×.

Multi-threading could be used for parallelizing across regions; Figure 5(c) depicts such a schedule, where all regions are executed in pipeline-parallel fashion. This schedule would at least require synchronizing the loop-carried dependence between subsequent matrix regions.

The granularity of these dependences is problematic for the commonly small matrices. Figure 6 shows the cumulative dependence distance of inter-region dependences across workloads. Most dependences are around a thousand instructions. Synchronizing multiple times through shared memory (for blocking) within this range is impractical. Empirically, Intel's optimized MKL library does not attempt multi-threading until a matrix size of 128 on Cholesky, and even then it *hurts* performance (details in Figure 24 on Page 11).

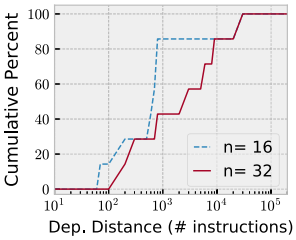

Fig. 6: Inter-region Dep. Granularity (n is matrix dimension)

III. Spatial Designs and Challenges

Spatial architectures expose low-level computation and network resources to the hardware/software interface. They are attractive for inductive workloads, as they rely less on multi-threading and vectorization, and instead leverage pipeline par-

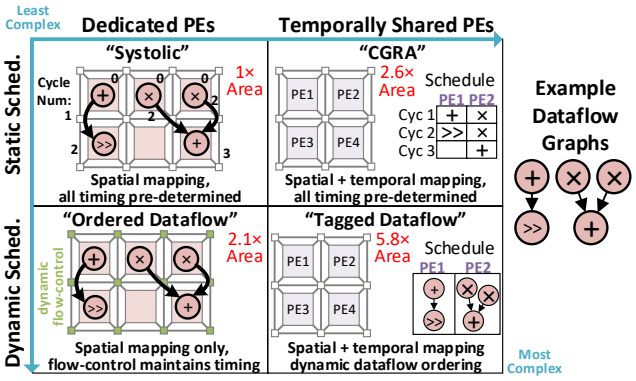

Fig. 7: Spatial Architecture Taxonomy with Example Program

allelism. We describe here a taxonomy of spatial architectures, and use this to explain the unique challenges and limitations that motivate a novel dataflow execution model and spatial architecture.

A. Spatial Architecture Taxonomy

We categorize spatial architectures in two dimensions: 1. static or dynamic scheduling: is the timing of all instructions determined statically or dynamically? 2. dedicated or shared PEs: is each PE dedicated to one instruction or shared among multiple instructions? Figure 7 depicts this taxonomy, along with how a simple computation graph is mapped to each. It also includes a relative PE area estimate[2]. Table I gives examples of each category. We explain each quadrant[3]:

- **"Systolic"** architectures have dedicated PEs, and the compiler schedules events to allow perfect pipelining of computation. Our definition includes systolic designs with a reconfigurable network. They are the simplest and smallest because PEs lack flow control.
- **"CGRAs"** (Coarse Grain Reconfigurable Architectures) add temporal multiplexing capability, so regions can be larger. They keep the hardware simplicity of static schedules, but require an instruction memory and register file.
- **"Ordered Dataflow"** augments systolic with limited dataflow semantics: instruction firing is triggered by data arrival; however, it is simpler than tagged dataflow as the order of firing is predetermined.
- **"Tagged Dataflow"** architectures allow data-triggered execution at each PE, with the capability or reordering instructions. They are the most expensive due to tag matching of some form, but also the most flexible and tolerant to variable latencies.

B. Spatial Architecture Challenges

To understand their tradeoffs, we implement and evaluate the two extremes within the spatial architecture taxonomy: systolic and tagged dataflow (hence just "dataflow"). The

[2]Methodology in Section VII; 64-bit PE; shared PEs have 32-instr. slots and 8 reg-file entries. Area does not include floating point (FP) units, but even with FP multiply, dataflow is $>4\times$ the systolic.

[3]We use these nicknames loosely: Our use of "systolic" is broader than in the literature, while our use of "CGRA" refers only to traditional CGRAs.

	Dedicated PE	Temporally Shared PE
Static Scheduling	**"Systolic"** Warp [23], FPCA [24], Softbrain [13], Tartan [10], Piperench [25]	**"CGRA"** MorphoSys [26], Remarc [27], MATRIX [28], ADRES [29], WaveFlow [30]
Dynamic Scheduling	**"Ordered Dataflow"** DySER [11], Q100 [31], Plasticine [12]	**"Tagged Dataflow"** TRIPS [9], SGMF [32], Trig. Insts [33], WaveScalar [8], dMT-CGRA [34]

TABLE I: Classifying Architectures within Spatial Taxonomy

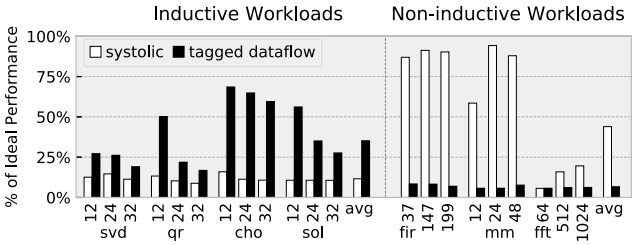

Fig. 8: Performance relative to ideal for Spatial Architectures

dataflow design most resembles Triggered Instructions [33], and the systolic most resembles Softbrain [13]. The total execution resources matches the TI DSP (8 cores, 16 MACs each, per-core SPAD). We develop and tune each algorithm for both architectures, and develop a cycle-level simulation framework and compiler (Section VII). Performance tradeoffs are in Figure 8, and are explained in the following paragraphs. Though spatial architectures are much faster than CPUs/GPUs/DSPs, they still do not achieve full throughput, and each type of architecture favors different workloads.

Challenge 1: Inductive Parallelism: Statically scheduled spatial architectures (systolic and CGRAs) cannot achieve parallelism across regions in inductive workloads, as inductive dependences do not have a static distance, which is required by static scheduling. Also, this limitation helps explain why the vast majority of CGRA compilers *only attempt parallelization of inner loops* [35]–[38]. The systolic architecture in Figure 8 also serializes each program region, which is another reason why it performs poorly on inductive workloads.

Ordered dataflow could parallelize across loops, as instruction firing is dynamic. However, as ordered dataflow only maps one instruction per PE, infrequently executed outer-loop program regions achieve quite low utilization.

Tagged dataflow architectures avoid the above challenges but suffer energy costs. They also do not fare well on non-inductive workloads with regular parallelism, as its difficult for the compiler to create a perfectly pipelined datapath; even a single PE with contention from two instructions in the inner loop will halve the throughput.

Challenge 2: Inductive Control: A common issue is control overhead from inductive memory accesses and dependences. We demonstrate with a simple reuse and inductive reuse pattern shown in Figure 9, along with the dataflow program representation. Traditional spatial architectures require these extra instructions to maintain effectively a finite state machine (FSM) to track the data-reuse for a given number of iterations,

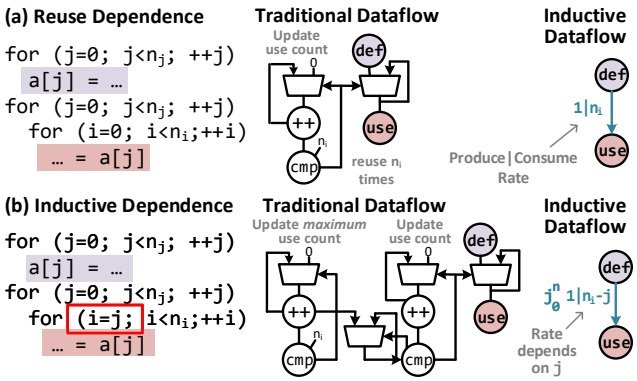

Fig. 9: Control overhead of Dependences in Dataflow Models

which is especially problematic for inductive dependences. These extra instructions can show up differently depending on the architecture; for systolic these execute on a control core (which can easily get overwhelmed); for dataflow these can be executed on the spatial fabric. The overhead from these instructions is the primary reason why in Figure 8, dataflow does not reach maximum throughput.

Summary: Existing spatial architectures are promising but have a combination of problems: control overhead, cannot achieve inductive parallelism, and/or high energy overhead.

IV. Specializing Spatial Architectures for Inductive Behavior

Based on our analysis, spatial architectures would be promising if they could keep their benefits of flexible parallelism without being burdened by control overhead and hardware complexity. Our solution, discussed in this section, is to specialize the dataflow model for inductive behavior and introduce codesigned hardware.

A. Inductive Dataflow Model

We develop here an execution model called *inductive dataflow*, which extends a traditional dataflow model with: 1. capability of expressing inductive dependence patterns (as dependence streams), 2. capability to express inductive memory patterns (as memory streams), and 3. semantics for stream interaction with vectorized computation.

Preliminaries: We begin with a simple dataflow model where nodes either represent: 1. a computation performed over inputs in the order received, or 2. a memory stream, which we define as a memory location and access pattern. We refer to the subset of the program region which are computation nodes as the *computation graph*.

Inspired by synchronous dataflow processes [39], an edge may be labeled by a production—consumption rate. Consumption > 1 indicates reuse of a value. (e.g. data is reused multiple times within a subsequent loop). Production > 1 means that several iterations occur before producing a value (e.g. only the first array item has a dependent computation).

We refer to these patterns of dependences as a *dependence streams*. For example a dependence stream may describe dependences between outer and inner loops, as shown by the example in Figure 9(a). Having dependence streams as first class primitives avoids the overhead of expressing the corresponding FSM in traditional dataflow instructions.

Inductive Memory and Dependence Primitives: In order to express inductive dependence streams, we add the capability to specify the relationship between the outer-loop induction variable and the production to consumption rate. An abstract example is in Figure 9(b), under "inductive dataflow", where the notation j_0^n means j varies from 0 to n.

Similarly, it is useful to express inductive memory access patterns (e.g. triangular). Therefore we extend the definition of memory streams to include the relationship to the outer-loop induction variable. Examples of a non-inductive (rectangular) stream and inductive (triangular) stream are below in Figure 10, along with a simple notation that uses `[0:n]` to define the range of the stream.

```
for j=0 to nj
  for i=0 to ni
    ... = a[j][i]
```

Notation: a[0:n_j,0:n_i]

(a) Non-inductive Stream

```
for j=0 to nj
  for i=0 to ni − j∗s
    ... = a[j][i]
```

Notation: $j_0^{n_j}$ a[j, 0:n_i-j×s]

(b) Inductive Stream

Fig. 10: Memory Address Stream Type Comparison

Note that the purpose of including these in the execution model is not to increase expressiveness, *rather it is to open an opportunity for specialization*.

Solver Example: Figure 11 (below) shows the `solver` kernel expressed in inductive dataflow. Circular nodes are compute instructions, and rectangular nodes are memory streams; inductive patterns are shown in blue. Memory accesses within the inner loop depend on `j`, so are inductive. The dependence between the inner and outer loop are also inductive.

Inductive Dataflow Vectorization: Often it is useful to apply vectorization to the program region which requires the most work, usually the inner loop. To vectorize, a load stream may connect to multiple compute nodes, and the subsequent

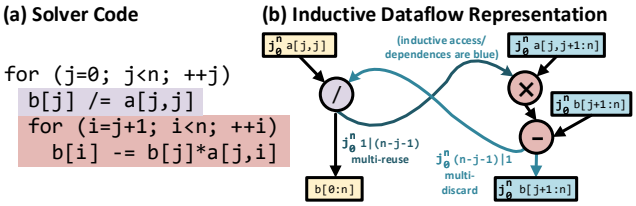

Fig. 11: Solver's Dataflows and Ordered Dependences

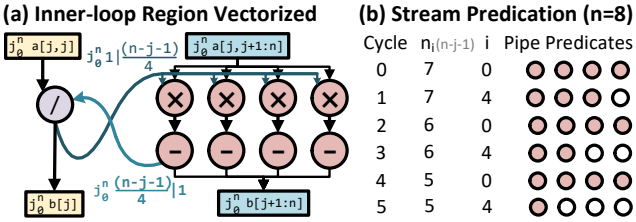

Fig. 12: Implicit Vector Masking with Solver Kernel

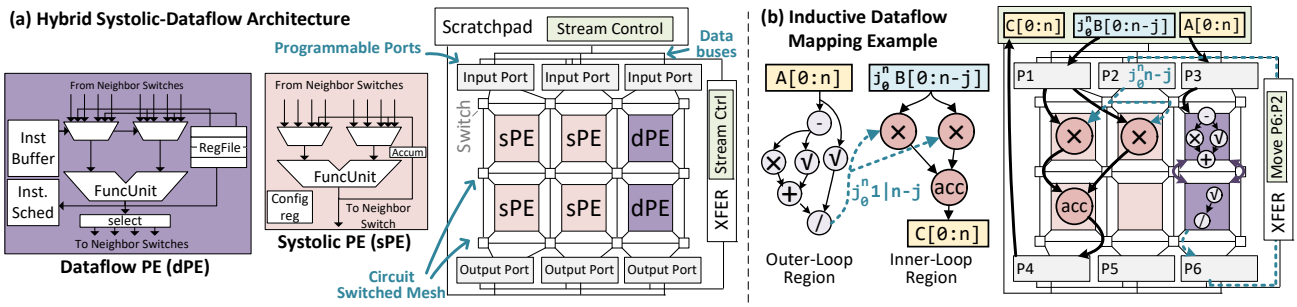

Fig. 13: Hybrid Systolic Dataflow Tiles, Overall Architecture, and Inductive Dataflow Mapping Example

values of the stream are distributed round-robin to each (a write stream consumes from connected nodes similarly). Figure 12(a) shows solver with the inner loop vectorized.

However, as with vector architectures, the iteration count of a loop (especially an inductive one) may not align with the vector width. The advantage of expressing inductive streams as a primitive is that we can assign meaning to the completion of the outer-loop (here the `j` loop). Specifically, we add the semantics that nodes which have not received data are implicitly predicated off. This predication is carried to the output through dependent instructions; at the output, streams which write memory ignore instances with invalid predicates. Figure 12(b) shows the state of the stream over several cycles (with `n`=8). This approach enables stall-free pipelined execution of the vectorized region.

B. Hybrid Systolic-Dataflow

With the execution model defined, we now develop an architecture which can leverage the additional semantics of inductive dataflow for highly-efficient execution.

Hybrid Rationale: As discussed in Section III, one of the main tradeoffs between systolic and dataflow architectures is the efficiency of simplified control in systolic versus the ability to exploit more general parallelism in dataflow.

Our rationale begins with the observation that in any kernel, some of the concurrent program regions correspond to more total work than others, hence they would execute more frequently. These regions are generally in the inner-most loops (e.g. matrix region in Cholesky). Our insight is that it is only these inner loop regions that make up the bulk of the work, and executing these on the efficient systolic array would yield the majority of the benefits. If we can provide a much smaller dataflow fabric for non-inner loop regions, and a way for all regions to communicate, this would be low cost and high efficiency. *Tagged* dataflow is specifically attractive because it can both temporally multiplex instructions and be resilient to the timing variation caused by inductive behavior.

Hybrid Architecture Design: Our overall approach is that we embed the dataflow architecture within the systolic architecture's network, so that functional units (FUs) within the dataflow component may be used for systolic execution. To enable communication between the regions, we use programmable ports which realize the inductive dependence semantics. Figure 13(a) shows the systolic and dataflow PEs, and how they are embedded into the overall design; Figure 13(b) shows an example program mapping.

PEs and Integration: *Dataflow PEs* are based on Triggered Instructions [33], [40]. An instruction scheduler monitors the state of architecture predicates and input channels (with tags) to decide when to fire an instruction. A triggered instruction will read operands (registers or inputs from neighboring switches) to the FU for processing. *Systolic PEs* are much simpler; they fire their dedicated instruction whenever any input arrives, are configured with a single register, and only have a single accumulator register (rather than a register file). The compiler is responsible for ensuring all systolic PE's inputs arrive at exactly the same cycle.

All PEs are embedded into a circuit-switch mesh network with no flow control. To map program regions onto the systolic PEs, the compiler configures the switches to implement the dependences. Only `1|1` production|consumption edges may be mapped to the circuit switched mesh. If a program region is mapped to the dataflow PEs, routers between the PEs will be statically configured to route to each other, so that there is a communication path between the PEs. Dataflow PEs time-multiplex these links to communicate.

Execution of Concurrent Program Regions: The *input and output ports* buffer data until it is ready to be consumed by the PEs or written to scratchpad. Ports are implemented as programmable FIFOs that have fixed connections to locations within the mesh. They may be associated with systolic program regions or dataflow regions.

Systolic regions have the execution semantics that one instance of the computation begins when *all* of its inputs are available. To support multiple systolic program regions which fire independently, the readiness of each region's ports are tracked separately. On the other hand, ports associated with dataflow regions may immediately send data. In the example in Figure 13, the inner-loop region uses three systolic PEs and the outer-loop region uses both dataflow PEs.

Maintaining Dependences: The **XFER unit** works with the ports to implement inter-region dependences. It arbitrates access to data buses on the output ports and input ports. If a dependence is non-inductive, the stream controller within the XFER unit is simply configured with a stream to send data from an output to an input port (multiple dependence streams will contend for the bus). For a dependence with reuse

(consumption>1), a hardware-specialized FSM is configured within the input port to track the number of times the data should be reused before popping the data. If a dependence discards some outputs (production>1), then the output port is configured to implement an FSM to track the number of times an output should be discarded. In the example, the inductive reuse dependence is routed from P6 to P2 by the XFER unit, then reused according to the FSM in P2.

Stream Predication for Vectorization: Ports also implement stream predication. The FSM at the port compares the remaining iterations with the port's vector length. If the iterations left is non-zero and less than the vector length, the stream control unit sends the data padded with zeroes for the unused vector-lanes, along with additional meta-information which indicates that the those vector-lanes should be predicated off.

Memory Access: Finally, a stream control unit in the scratchpad will arbitrate streams' access into (or out of) the mesh, by writing (or reading) its own set of data buses. Inductive memory access is supported in the same way as dependences: the FSM in the input or output port is configured to reuse or discard according to the inductive pattern. Another benefit of the reuse support in the port is that it reduces scratchpad bandwidth for those accesses.

Overall, the systolic-dataflow architecture achieves both high efficiency execution for the vast majority of the computation, and flexible parallelism where it is needed.

C. Applicability to Other Architectures

The inductive dataflow model opens an opportunity for specializing common control patterns and also enables efficient vectorization of these patterns. It can be independently applied to *any* spatial architecture by introducing dependence stream and memory stream primitives. For example, if a traditional CGRA (static-scheduled/shared-PE) is extended with hardware and a software interface to specify an inductive memory stream primitive (and if it supports stream predication), vectorization of an inductive region can be achieved.

The principle behind the hybrid-systolic dataflow design – to combine temporally-shared and dedicated-PEs in one spatial architecture – is also broadly applicable. For example, Plasticine [12], a dynamic dedicated-PE spatial architecture, could be augmented with tagged dataflow PEs for low-rate computations, enabling higher overall hardware utilization.

V. REVEL Accelerator

Thus far we have described a spatial architecture to execute inductive-dataflow programs and a microarchitecture which can take advantage of this specialized program representation. What remains is to develop an approach for scaling the design, as well as a specific ISA. We address these aspects with the REVEL accelerator proposed in this section. We first discuss flexible scaling with multiple lanes, then cover the vector-stream ISA which lets a single simple control core coordinate all lanes.

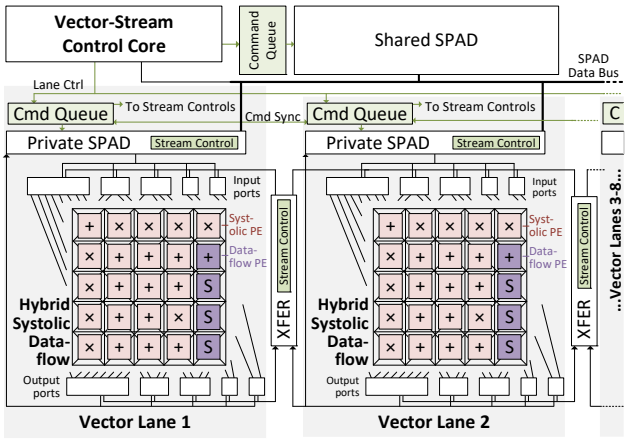

Fig. 14: REVEL Accelerator Architecture

A. REVEL Architecture

Multi-lane Rationale: Scaling up the design requires consideration on how to coordinate parallel units with low overhead, while maintaining the ability to exploit inductive parallelism. A possible approach is a monolithic architecture with large spatial array and very-wide scratchpad. However, the monolithic design has several drawbacks:

- Average routing distance increases, increasing average latency of communication.
- Necessity of reconfiguring the entire array at one time, which reduces flexibility.
- Long compilation times for the spatial architecture as routing/scheduling complexity increases.

We take the alternative approach of using multiple *lanes*, where each lane is comprised of the hybrid systolic-dataflow architecture and some control. Lanes can independently execute their own program regions, while also communicating using dependence streams or through shared scratchpad memory. Also, since each lane can be programmed separately, they can either work together or on independent tasks.

We first explain the high-level accelerator operation, then describe the ISA and vector-stream approach.

REVEL Overview: The accelerator (Figure 14) is composed of a number of lanes, a low power VonNeumann control core, and a shared scratchpad (serves as the external memory interface). The control core constructs "stream commands" and ships these asynchronously to the lanes. Stream commands are buffered in local *command queues* until the hardware is ready, and ensures they execute in program order. Streams are issued to the XFER unit and scratchpad controller. At this point, execution of the spatial fabric lane is as described in the previous section. The XFER unit is extended to support communication between program regions mapped to consecutive lanes. Since dataflow PEs are larger, they are grouped on the right side of the spatial fabric to enable simpler physical design.

B. Vector-stream ISA and Control Model

An ideal control-model would both rely on simple-to-reason-about sequential programming, and amortize control

	Pattern Params	Source Params	Dest. Params	
`StoreStream`	c_i, c_j n_j, n_i, s_{ji}	out_port	local_addr	*Lane Bitmask (all)*
`LoadStream`		local_addr		
`Const`	n_1, n_2, s	val_1, val_2	in_port, n_c, s_c	
`XFER`	n_p, s_p	out_port		
`Configure`		local_addr		
`Barrier Ld/St&Wait`				

TABLE II: REVEL's Vector-Stream Control Commands

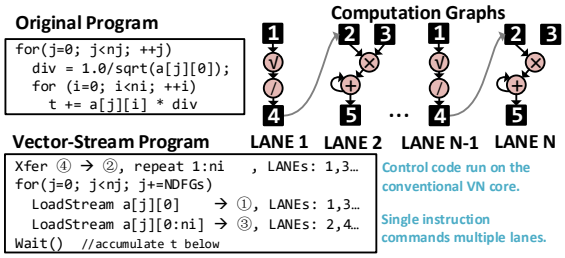

Fig. 16: A Vector Stream Control Example

over parallel units and through time to prevent the control code from becoming the bottleneck. For this we leverage stream-dataflow [13], which is a decoupled access-execute ISA that describes execution as the interaction of a VonNeumann control program and a computation graph, decoupled by streams. Our approach is to develop a version of this ISA which is vectorized across lanes, and can support inductive access. We first describe the extensions for controlling a single lane of inductive dataflow, then discuss the vector-stream approach to extend to multiple lanes.

Control Model: Stream dataflow [13] is an ISA for decoupled access execute, where a VonNeumann control program constructs memory and dependence streams and synchronizes stream access and execution of computation graphs. "Ports" are named identifiers representing the interface between streams and computation nodes; they are the software identifiers for FIFOs discussed earlier. A typical program phase begins with the control program requesting configuration of the spatial architecture for one or more program regions. The control program defines and issues streams through *stream commands*, which perform memory access and communication. Finally the program waits until the offloaded region is complete.

Figure 15(a) shows solver's streams represented in the encoding we develop below, and Figure 15(b) shows the corresponding control program (single lane). Numbers indicate ports, chosen by the programmer or compiler.

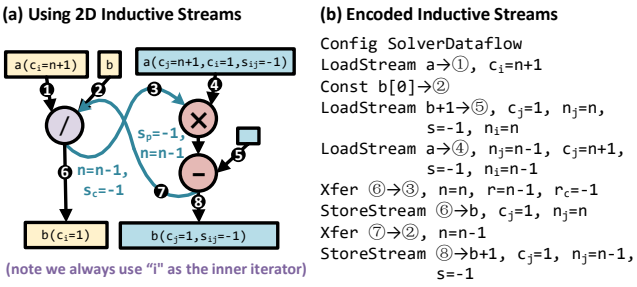

Fig. 15: Example of REVEL Control Program Encoding

We explain REVEL's stream commands here (summary in Table II). `LoadStream`/`StoreStream` define memory streams to move data between memory and the computation graph (through input/output ports). Their pattern parameters are a starting address, stride (c_i, c_j) and length (n_i, n_j) in two dimensions. To support inductive behavior, a *stretch* (s_{ji}) parameter encodes the change of iterator j in the trip count for dimension i. `XFER` defines dependence streams (within and across regions), encoded with a source and destination port, production and consumption rate (n_p and n_c), and two "stretch" parameters (s_p and s_c) for encoding the inductive change to these parameters. `Const` sends a constant into a port, and can stream an inductive pattern of two constants, e.g. 0,0,0,1,0,0,1,0,1. `Barrier_Ld/St` are fences on scratchpad loads and stores, useful for double buffering. `Wait` blocks until stream completion.

Vector-stream Control: To scale to multiple lanes, we *vectorize* the stream-dataflow ISA. Figure 16 shows the conceptual approach with a simple high-level example, where two different regions are mapped to odd and even lanes, and streams are broadcast to them separately. Streams may either be broadcast to all lanes identically (e.g. all lanes read the same location in local memory), or they can be modified locally by adding an offset to the starting address and/or length parameters (a multiple of the lane id). This allows a single command to direct each lane to read a separate slice of an array. For flexibility, commands are only received by relevant lanes, which are specified by a bitmask.

Implementing vector-stream control with inter-region dependences extending between lanes requires some coordination in hardware – specifically to ensure that all streams which use the same port execute in program order. Normally, command queues only ensure that streams assigned to *that lane* execute in program order, but we need to establish order across lanes. This is accomplished by sending the destination lane a placeholder stream. The destination's command queue informs the source's when the placeholder is issued for the destination port, and the source's command queue informs the destination's when the placeholder can be removed (using `cmd sync` bus in Figure 14).

Overall, vector-streams offer more control amortization than either vectorization or streaming alone, as it amortizes both in *space* across lanes, and in *time* through streams.

End-to-end Example: Figure 17 demonstrates REVEL's abstractions by showing how Cholesky may be expressed as a combination of vector-stream control code (c) and dataflow configuration for each lane (d). The parallelization strategy for the optimized code is:

1) Vectorize the inner loop (leveraging stream predication)
2) Designate outer-loop regions for dataflow execution to amortize execution resources, and map scalar, vector, matrix regions to one lane.

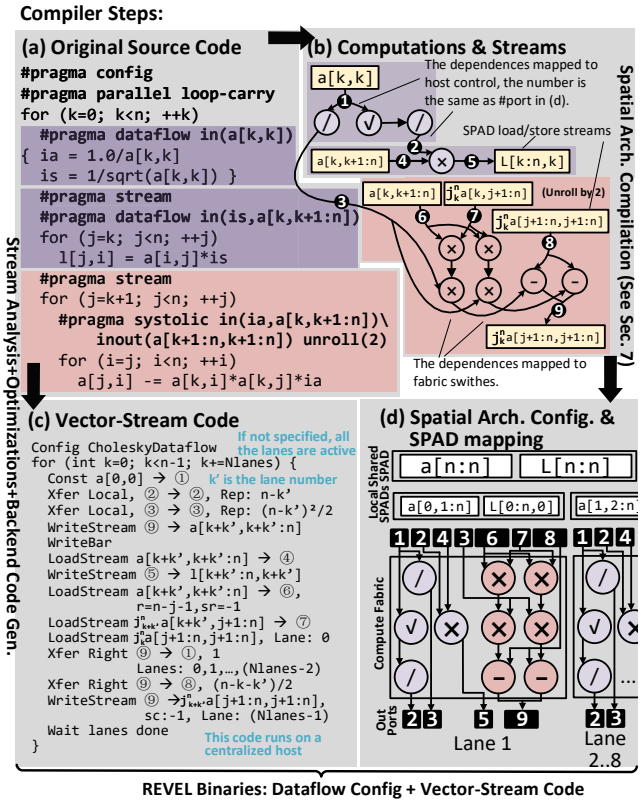

Fig. 17: Mapping Cholesky to REVEL with pragma support

3) Parallelize the outer k loop across lanes.

VI. Programming and Compilation

Programming REVEL involves five basic responsibilities:

1) Dividing the work onto lanes and partitioning data.
2) Deciding which program regions execute concurrently.
3) Extracting memory/dependence streams; inserting barriers.
4) Decoupling the computation graph and vectorizing it.
5) Mapping computation onto spatial PEs and network.

We developed an LLVM/Clang-based compiler which relies on pragma-hints for 1-2 and automates 3-5. Figure 18 shows the pipeline of compilation, and Figure 17 shows Cholesky undergoing transformation from C with pragmas to REVEL abstractions. We next explain the compiler stack.

Pragma-based Compilation: The pragmas are inspired by OpenMP's tasks [41]. Each offloaded code region is similar to a light-weight thread: it has an independent flow and potentially input/output dependences. Figure 17 (a) shows how Cholesky is annotated with the following pragmas:

`#pragma dataflow/systolic` specifies whether a program region is offloaded to the systolic or dataflow architecture. It has two optional clauses: The `unroll` clause specifies the number of iterations to offload in one computation instance, determining resource use; the `in/out/inout` clause specifies data dependence distance patterns.

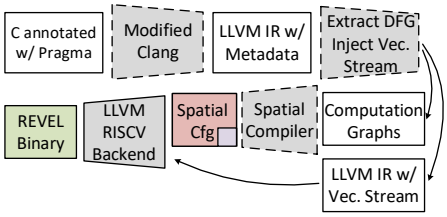

Fig. 18: REVEL's Software Stack

`#pragma stream` indicates that the memory operations under this loop level can be hoisted outside and encoded with vector-stream instructions. It guarantees alias freedom, except for those specified explicitly.

`#pragma config` indicates that regions within the following scope will execute concurrently on spatial fabric.

`#pragma parallel` indicates the portion of the program to be parallelized across lanes. The clause `loop-carry` is a hint to detect and enforce loop-carried dependences by analyzing the inputs/outputs specified by `in/out/inout` clauses between adjacent iterations[4].

Computation Graph and Vector-Stream Extraction: The first step is to extract the computation graph (using slicing [42]) and streams from offloaded program regions. A configuration instruction is inserted at the site of `#pragma config`, and is pointed to spatial configuration bits after they are generated. Streams are extracted by determining the access pattern, for which we use LLVM scalar evolution [43] analysis. The encoded streams are injected at the site of `#pragma stream`. Dependence pattern metadata is used to see if some memory streams can be converted to dependence streams to save memory traffic. Scratch barriers are inserted to keep data consistent. Finally, empty loops (due to being completely offloaded) are removed.

Spatial Architecture Compiler: The spatial architecture compiler maps computation graphs of all concurrent program regions to systolic and dataflow PE resources, and generates the configuration (similar role as [35], [37], [44]–[46]). The graph is first "unrolled" (i.e. vectorized) by a degree determined by pragma meta-data if specified, otherwise the most critical loop (estimated based on expected instruction count) is unrolled to fill the spatial fabric.

The responsibilities of the dataflow compiler are to map instructions to PEs, dependences to the network, and to make timing decisions. For the systolic regions, all operand timing paths must be equalized, and no resources may be shared between instructions. For the dataflow computation graphs, the goal is to minimize resource contention. Usually instructions belonging to systolic/dataflow regions map to the corresponding PEs (systolic-PEs or dataflow-PEs); However, dataflow instructions may map to the systolic fabric if it has low utilization, and systolic instructions to the dataflow fabric to reduce latency or network congestion, provided that there are enough resources. To balance these objectives, we adapt a prior heuristic [47] to use simulated annealing, similar

[4]Support for this pragma is ongoing, some kernels require intervention.

Revel Lane (× 8)	Spatial Fabric	PEs Div/Sqrter SubwrdSIMD Dataflow PE	14 add, and 3 sqrt/div, 9 mult Lat.: 12 Cyc., Thr.: 1/5 Cyc. 4-way Fixed-point, 2-way FP 1x1 (32 Instruction Slots)
	Vector Ports	Width Depth	2×512, 2×256, 1× 128, 1× 64 bit 4-entry FIFO
	Stream Ctrl	Stream Table Cmd Queue	8 Entries for 8 concurrent streams 8-Entry Cmd Queue
	SPAD	Structure Bandwidth	8Kb, Single-bank 512 Bits (1R/1W Port)
	Net.	Data Bus Spatial Mesh	2×512 Bit (SPAD+XFER) 64-bit Circuit-switched Mesh
Ctrl Core	RISCV ISA [51], 5-stage, single-issue, 16kb d$, insts. added for stream-commands		
Shr. SPD	Structure: 128Kb, Single-bank Bandwidth: 512 Bits (1R/1W Port)		
Net.	Inter-lane: 512 Bit Data Bus (8-bit Cmd Sync) Shared scratchpad Bus: 512 Bit Shared Bus		

TABLE III: REVEL Parameters

to Pathfinder [48]. All concurrent computation graphs are mapped simultaneously to find the best overall schedule.

VII. Evaluation Methodology

REVEL Modeling: Table III shows REVEL hardware parameters. All blocks are modeled at a cycle level in a custom simulator, which is integrated with a gem5 model of a RISCV inorder core [49], [50], extended for vector-stream control. To compare against state-of-the-art spatial architectures, we create a custom simulator for each. We synthesized REVEL's prototype using Synopsys DC, 28nm tech library. The design meets timing at 1.25GHz. An open source triggered instructions implementation was our reference for the temporal fabric [40]. Results from synthesis are used to create an event-based power model and area model.

ASIC Analytical Models: These optimistic models (Table IV) are based on the optimized algorithms, and are only limited by the algorithmic critical path and throughput constraints, with equivalent FUs to REVEL. ASIC area and power models only count FUs and scratchpad.

SVD	QR	MM
$4dm+2\text{QR}(n)+\lceil\frac{n^3}{8_{\text{vec}}}\rceil$	$7dn+2\sum_{i=1}^{n}(i+\lceil\frac{i}{2_{\text{vec}}}\rceil n)$	$\lceil\frac{n}{8_{\text{vec}}}\rceil mp$
Solver	**FFT**	**Cholesky**	**Centro-FIR**
$2\sum_{0}^{n-1}\max(\lceil\frac{i}{4_{\text{vec}}}\rceil, d+2)$	$\frac{n}{8_{\text{vec}}}\log n$	$\sum_{i=1}^{n-1}\max(\lceil\frac{i^2}{2_{\text{vec}}}\rceil, 4\text{d})$	$\lceil\frac{n-m+1}{4_{\text{vec}}}\rceil m$

TABLE IV: Ideal ASIC Models. m, n, p are the matrix dims. (except SVD, where m is the number of iterations and Centro-FIR, where m is the filter size), x_{vec} indicates x-vectorized, and d is the latency of div/sqrt.

Workload Versions: We evaluate batch size 1 and 8, requiring different optimizations: For batch 1, REVEL spreads work across lanes (if possible), and for batch 8 each lane operates over one input. Table V shows data-sizes and # lanes in batch 1.

Workload	Data Size	Lanes
SVD	**12**,16,24,**32**	1
QR	**12**,16,24,**32**	8
Cholesky	**12**,16,24,**32**	8
Solver	**12**,16,24,**32**	1
FFT	**64**,128,512**1024**	1
GEMM	**12**,**48**x16x64	8
FIR	**37**,**199**x1024	8

TABLE V: Workload Parameters. "small","large" sizes bolded

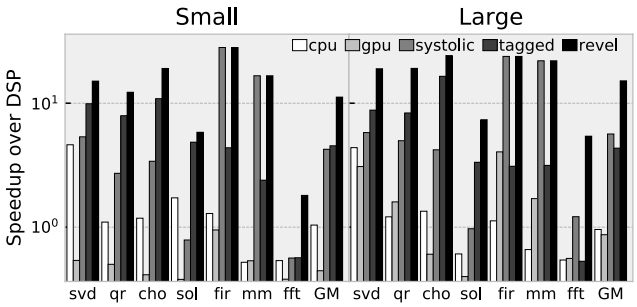

Fig. 19: Performance Tradeoffs (batch size=1)

Comparison Methodology: For fairness we compare designs with similar ideal max. FLOPs (except GPU, which has more):

- **TI 6678 DSP (@1.25GHz)** 8-core DSP, each core has 16-FP adders/multipliers, using DSPLIB_C66x_3.4.0.0.
- **OOO Core: Intel Xeon 4116 (@2.1GHz)** Conventional OOO processor using highly-optimized Intel MKL library. (8 cores used)
- **GPU: NVIDIA TITAN V (@1.2GHz)** GV100 graphics processor using cuSOLVER, cuFFT, and cuBLAS NVIDIA CUDA library as our gpu benchmark. GPU's peak FLOPs is >10× *higher* than REVEL.
- **Spatial:** The *systolic* design is similar to Softbrain [13], and *dataflow* is similar to Triggered Insts. [33]. FUs and #lanes are the same.

VIII. Evaluation

Our evaluation has four main goals. First to quantify the speedups over state-of-the-art CPUs, DSPs, Spatial, and GPUs. Second, to characterize the sources of benefits behind the specialization of inductive parallelism, as well as the remaining bottlenecks. Third, to understand the sensitivity to architecture features. Finally, to compare the area/power/performance with ASICs. Overall, we find that REVEL is consistently better than all state-of-the-art designs, often by an order of magnitude.

A. Performance

Overall Speedup: Speedups over DSP for batch 1 are shown in Figure 19. The DSP and CPU have similar mean performance. REVEL attains up to 37× speedup, with geomean of 11× and 17× for small and large data sizes. REVEL is 3.5× and 3.3× faster than dataflow and systolic.

Performance for batch 8 is in Figure 20. For small and large sizes, REVEL gets a speedup of 6.2× and 8.1× over the DSP and CPU. REVEL's dataflow/vector-stream model provides 4.0× speedup over dataflow, and 2.9× over systolic.

REVEL provides factors speedup over state-of-the-art.

REVEL vs CPU Parallelism: Figure 21 shows the scaling of REVEL's performance against the MKL's library's CPU version for different sizes of Cholesky and thread counts. Observe that when multi-threading is first enabled in MKL (>= matrix size 128), it actually hurts performance. This is because of the inherent fine-grain dependences, which REVEL supports natively.

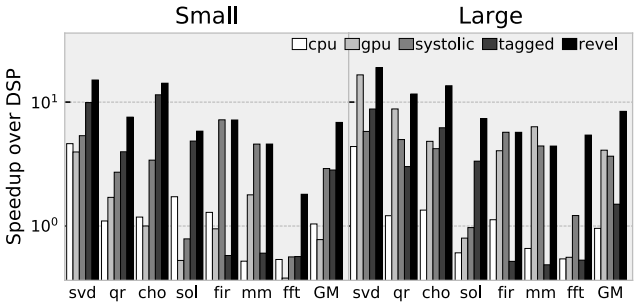

Fig. 20: Performance Tradeoffs (batch size=8)

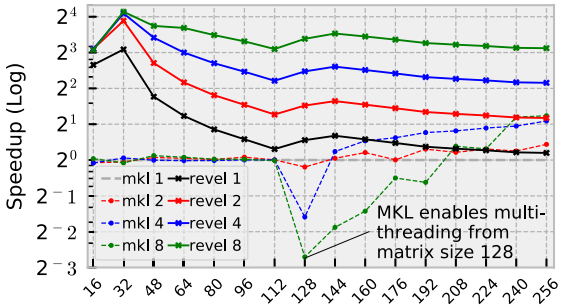

Fig. 21: CPU vs REVEL Scaling

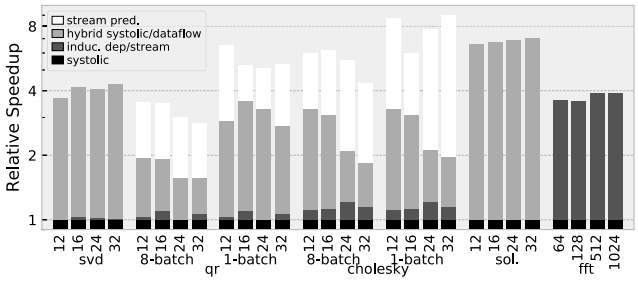

Fig. 22: Performance Impact of Each Mechanism.

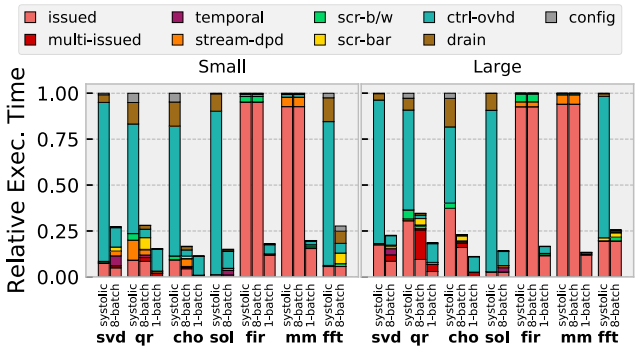

Fig. 23: REVEL's Cycle-level bottlenecks

Inductive dataflow can parallelize much finer-grain dependences than with CPU threading.

Benefits from Hardware/Software Mechanisms: To understand the sources of improvement, we evaluate four versions of REVEL with increasingly advanced features. We start with the systolic, then add inductive streams, hybrid systolic-dataflow, and finally stream predication to enable efficient vectorization. Figure 22 shows the results.

Inductive memory and dependence streams improve all workloads by reducing control and increasing parallelism. Even FFT benefits by using inductive reuse to reduce scratchpad bandwidth. QR and SVD have complex outer-loop regions, so do not benefit as much until after adding hybrid systolic-dataflow, which enables more resource allocation for inner-loop regions. Solver was also accelerated by the heterogeneous fabric because it is latency sensitive, and collapsing less critical instructions can reduce latency. The vectorized workloads also receive large gains from stream predication by reducing the overheads of vectorization.

The vector-stream ISA and hybrid systolic-dataflow architecture together enable high performance.

Cycle-Level Bottlenecks: Figure 23 overviews REVEL's cycle-level behavior, normalized to systolic. To explain the categories, *issue* and *multi-issue* means that one or multiple systolic regions fired, and *temporal* means only a temporal dataflow fired during that cycle. All other categories represent overhead, including the *drain* of the dedicated fabric, *scr-b/w* and *scr-barrier* for bandwidth and synchronization, *stream-dpd* for waiting on dependences, and *ctrl-ovhd* for waiting on the control core.

The clearest trend is that our design reduces the control overhead dramatically. For some kernels, REVEL is able to execute multiple regions in the same cycle, especially for larger matrices. One outlier is FFT with small data; it requires multiple reconfigurations, each requiring the pipeline to drain.

Exploiting inductive parallelism increases parallel work and reduces control, enabling better performance.

Dataflow PE Allocation: Tagged dataflow PEs are helpful on inductive workloads, but expensive. A tagged-dataflow PE costs $>5\times$ more area than a systolic PE ($2822\mu m^2$ versus $16581\mu m^2$). Figure 24 shows REVEL's performance and area sensitivity. SVD has the largest demand on dataflow PEs, so are affected the most. The effects on other workloads are neglectable, so we choose 1 dataflow PE to minimize the area penalty.

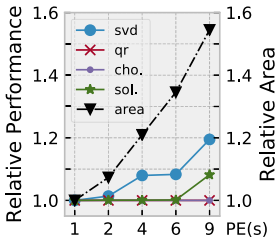

Fig. 24: Temporal region sensitivity

B. Area and Power Comparison

Breakdown: Table VI shows the power/area breakdown; the largest source (especially power) comes from FP units. REVEL is 1.93mm^2, and 1.63 Watts.

Comparing against CPU and DSP: Figure 25 shows the relative performance/area normalized to the CPU after adjusting the technology. The DSP achieves a high performance/mm^2, and REVEL is able to achieve even higher performance with a moderate area overhead. REVEL has $1089\times$ performance/mm^2 advantage over the OoO core, and $7.3\times$ over the DSP.

Comparing against ASIC: Table VII shows performance-normalized area overhead over ASIC analytical models.

		area(mm^2)	power(mw)
Compute Fabric	Dedi. Net. (24)	0.06	71.40
	Temp. Net. (1)	0.02	14.81
	Func. Units	0.07	74.04
	Total Fabric	0.13	160.25
Control (ports/XFER/str. ctrl)		0.03	62.92
SPAD-8KB		0.06	4.64
1 Vector Lane		0.22	207.90
Control Core		0.04	19.91
REVEL		1.93	1663.3

TABLE VI: Area and Power Breakdown (28nm)

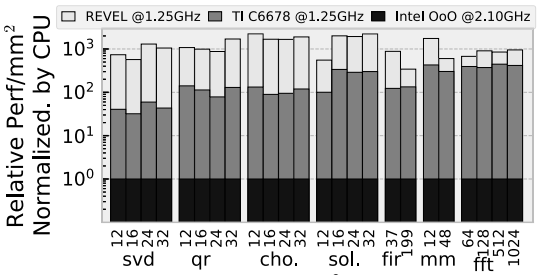

Fig. 25: Relative performance/mm^2 normalized to CPU.

REVEL is mean 2.0× power. This is mostly due to the control logic (ports, bus, etc.) and reconfigurable networks. It is 0.55× the area of the combined ASIC. This is optimistic for ASICs in that it assumes perfect pipelining and no control power.

REVEL is on par with ASICs-level efficiency.

Workloads	SVD	QR	Cho.	Sol.	FIR	MM	FFT	Mean
Power Ovhd.	2.8	2.0	1.9	1.6	2.0	1.9	1.9	2.0
Area Ovhd.	3.3	2.4	2.3	2.2	2.3	2.3	2.8	2.5/0.55

TABLE VII: Power/Area overheads to ideal ASIC (iso-perf)

IX. Additional Related Work

In this section, we discuss work other than that previously covered by the spatial architecture taxonomy in Section III-A.

Synchronous Dataflow Variants: The inductive production to consumption rates in our dataflow model is inspired by the static rates in synchronous dataflow [39] (SDF). SDF was developed as a specialization of existing dataflow models which could be statically scheduled. Cycle-static dataflow [52] extends SDF with periodically changing rates, and heterochronous dataflow [53] extends SDF to enable an FSM to step through predefined rates. None of the above were applied to spatial architectures or handle inductive dependences.

StreamIt [54] is a language and runtime with somewhat similar semantics to vanilla SDF, and was evaluated on RAW [55], a (mostly) static/shared-PE spatial architecture.

Outer-loop Parallelism: Prabhakar *et al.* develops "nested-parallelism," which enables coupling of datapaths with nested parallel patterns [56]. Inductive parallelism is a generalization of nested-parallelism, and we can achieve a higher utilization due to hybrid systolic-dataflow execution.

Some CGRA compilers target nested loops [57], [58], but only parallelize the epilogue and prologue of subsequent loop nests. Recent work has made progress in pipelining imperfect nests [59], but does not parallelize across multiple region instances. CGRA Express [60] allows a CGRA to use the first row of its PEs in VLIW mode during outer loops. Concurrent execution across inner and outer regions is not attained. None of the above handle inductive dependences.

Flexible Vectorization: Vector-threading techniques also marshal independent execution lanes for vectorized execution when useful [61]–[64]. The RISC-V vector extension supports configurable vector-length and implicit vector masking [65]. Vector-length is limited by physical registers (REVEL's streams are arbitrary length), and inductive access is not supported, so the vector length would have to be reset on each iteration. These architectures are also not spatial, so cannot exploit pipelined instruction parallelism.

Some spatial dataflow models use predicates for control [6], [66]. These do not use streams for vector predication. dMT-CGRA [34] adds inter-thread communication for a spatial-dataflow GPU [32], [67].

DSP Accelerators: Many application/domain-specific reconfigurable designs have targeted DSP algorithms. Fasthuber et. al [68] outline the basic approaches. One representative example includes LAC [69], targeted at matrix factorization. Our architecture allows more general programmability.

Stream-based ISAs and Reuse: Many prior architectures have used memory-access stream primitives [13], [24], [31], [70]–[74]. To our knowledge, no prior work has incorporated inductive patterns into such streams.

X. Conclusion

This work identified that existing techniques, vector, multi-threading, and spatial, all experience challenges in achieving high-performance on dense linear algebra workloads, largely due to inductive program behavior.

We found that it is possible to specialize for this behavior by encoding inductive properties into the hardware/software interface. This was the approach behind inductive dataflow, a model that allows the expression of parallelism within inductive program regions and across them. Our taxonomy of spatial architectures also makes it clear why a hybrid architecture – one that combines systolic fabrics for efficiency and dataflow fabrics for flexible parallelism – can achieve the best of both. Finally, this work develops a scalable design: REVEL, by leveraging a vector-stream ISA that amortizes control in space and time. With a full stack implementation, our evaluation against four state-of-the-art designs demonstrates many factors of speedup and energy efficiency.

On one hand, our contribution is what we believe to be a superior digital signal processor. What is perhaps more important is the principle of hardware/software specialization of complex but general control and memory patterns, useful for developing next generation programmable accelerators.

XI. Acknowledgments

We sincerely thank Naxin Zhang, Zhiguo Ge, Jing Tao, and Mian Lu for their thoughtful discussions and domain expertise.

This work was supported by an NSF CAREER award CCF-1751400 and gift funding from HiSilicon.

REFERENCES

[1] I. T. Union, "Minimum requirements related to technical performance for IMT-2020 radio interface(s)," 2017.

[2] S. Chen and J. Zhao, "The requirements, challenges, and technologies for 5g of terrestrial mobile telecommunication," *IEEE communications magazine*, vol. 52, no. 5, pp. 36–43, 2014.

[3] H. Tullberg, P. Popovski, Z. Li, M. A. Uusitalo, A. Hoglund, O. Bulakci, M. Fallgren, and J. F. Monserrat, "The metis 5g system concept: Meeting the 5g requirements," *IEEE Communications magazine*, vol. 54, no. 12, pp. 132–139, 2016.

[4] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[5] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *ISPASS*, 2005.

[6] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial Computation," in *ASPLOS XI*.

[7] Arvind and R. S. Nikhil, "Executing a program on the MIT Tagged-Token Dataflow Architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, 1990.

[8] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 291–, IEEE Computer Society, 2003.

[9] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and t. T. Team, "Scaling to the end of silicon with edge architectures," *Computer*, vol. 37, pp. 44–55, July 2004.

[10] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: Evaluating spatial computation for whole program execution," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 163–174, ACM, 2006.

[11] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, pp. 38–51, Sept. 2012.

[12] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 389–402, ACM, 2017.

[13] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 416–429, ACM, 2017.

[14] R. Mudumbai, G. Barriac, and U. Madhow, "On the feasibility of distributed beamforming in wireless networks," *IEEE Transactions on Wireless communications*, vol. 6, no. 5, 2007.

[15] H. Johansson *et al.*, "Polyphase decomposition of digital fractional-delay filters," *IEEE signal processing letters*, vol. 22, no. 8, pp. 1021–1025, 2015.

[16] R. Zhao, "Wls design of centro-symmetric 2-d fir filters using matrix iterative algorithm," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, pp. 34–38, July 2015.

[17] F. Mintzer, "On half-band, third-band, and nth-band fir filters and their design," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 30, pp. 734–738, Oct 1982.

[18] M. Dendrinos, S. Bakamidis, and G. Carayannis, "Speech enhancement from noise: A regenerative approach," *Speech Communication*, vol. 10, no. 1, pp. 45–57, 1991.

[19] D. Patel, M. Shabany, and P. G. Gulak, "A low-complexity high-speed qr decomposition implementation for mimo receivers," in *2009 IEEE International Symposium on Circuits and Systems*, pp. 33–36, May 2009.

[20] P. Salmela, A. Happonen, T. Jarvinen, A. Burian, and J. Takala, "Dsp implementation of cholesky decomposition," in *Joint IST Workshop on Mobile Future, 2006 and the Symposium on Trends in Communications. SympoTIC '06.*, pp. 6–9, June 2006.

[21] P. Darwood, P. Alexander, and I. Oppermann, "Lmmse chip equalisation for 3gpp wcdma downlink receivers with channel coding," in *ICC 2001. IEEE International Conference on Communications. Conference Record (Cat. No.01CH37240)*, vol. 5, pp. 1421–1425 vol.5, 2001.

[22] D. Tse and P. Viswanath in *Fundamentals of Wireless Communication*, New York, NY, USA: Cambridge University Press, 2005.

[23] M. Annaratone, E. A. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp Computer: Architecture, Implementation, and Performance," *IEEE Transactions on Computers*, vol. 36, pp. 1523–1538, December 1987.

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

[25] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer, "Piperench: a coprocessor for streaming multimedia acceleration," in *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, 1999.

[26] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, pp. 465–481, May 2000.

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

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

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

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

[31] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 255–268, ACM, 2014.

[32] D. Voitsechov and Y. Etsion, "Single-graph multiple flows: Energy efficient design alternative for gpgpus," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 205–216, IEEE Press, 2014.

[33] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered instructions: A control paradigm for spatially-programmed architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 142–153, ACM, 2013.

[34] D. Voitsechov and Y. Etsion, "Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays," *arXiv preprint arXiv:1801.05178*, 2018.

[35] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEE Proceedings - Computers and Digital Techniques*, vol. 150, pp. 255–61–, Sept 2003.

[36] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEE Proceedings - Computers and Digital Techniques*, vol. 150, pp. 255–, Sep. 2003.

[37] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 166–176, 2008.

[38] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–10, May 2013.

[39] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 7, pp. 1235–1245, September 1987.

[40] T. J. Repetti, J. a. P. Cerqueira, M. A. Kim, and M. Seok, "Pipelining a triggered processing element," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 96–108, ACM, 2017.

[41] TheOpenMPTeam, "Openmp." https://openmp.org, 1997-2019.

[42] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg," in *PACT*, pp. 341–351, 2013.

[43] R. A. Van Engelen, "Efficient symbolic analysis for optimizing compilers," in *International Conference on Compiler Construction*, pp. 118–132, Springer, 2001.

[44] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Instruction scheduling for a tiled dataflow architecture," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pp. 141–150, 2006.

[45] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, (New York, NY, USA), pp. 46–57, ACM, 1998.

[46] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 495–506, ACM, 2013.

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

[48] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for fpgas," in *Third International ACM Symposium on Field-Programmable Gate Arrays*, pp. 111–117, Feb 1995.

[49] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[50] A. Roelke and M. R. Stan, "RISC5: Implementing the RISC-V ISA in gem5," in *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[51] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[52] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. 3255–3258 vol.5, May 1995.

[53] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 18, no. 6, pp. 742–760, 1999.

[54] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*, pp. 179–196, Springer, 2002.

[55] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 291–303, October 2002.

[56] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 651–665, ACM, 2016.

[57] S. Yin, D. Liu, Y. Peng, L. Liu, and S. Wei, "Improving nested loop pipelining on coarse-grained reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 2, pp. 507–520, 2016.

[59] S. Yin, X. Lin, L. Liu, and S. Wei, "Exploiting parallelism of imperfect nested loops on coarse-grained reconfigurable architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 3199–3213, Nov 2016.

[58] J. Lee, S. Seo, H. Lee, and H. U. Sim, "Flattening-based mapping of imperfect loop nests for cgras?," in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–10, Oct 2014.

[60] Y. Park, H. Park, and S. Mahlke, "Cgra express: Accelerating execution using dynamic operation fusion," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, (New York, NY, USA), pp. 271–280, ACM, 2009.

[61] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 129–140, ACM, 2011.

[62] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, (Washington, DC, USA), pp. 52–, IEEE Computer Society, 2004.

[63] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis, "Vector lane threading," in *2006 International Conference on Parallel Processing (ICPP'06)*, pp. 55–64, Aug 2006.

[64] J. Kim, S. Jiang, C. Torng, M. Wang, S. Srinath, B. Ilbeyi, K. Al-Hawaj, and C. Batten, "Using intra-core loop-task accelerators to improve the productivity and performance of task-based parallel programs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 759–773, ACM, 2017.

[65] R. Foundation, "Working draft of the proposed risc-v v vector extension." https://github.com/riscv/riscv-v-spec.

[66] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley, "Dataflow predication," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 89–102, IEEE, 2006.

[67] D. Voitsechov and Y. Etsion, "Control flow coalescing on a hybrid dataflow/von neumann gpgpu," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 216–227, Dec 2015.

[68] R. Fasthuber, F. Catthoor, P. Raghavan, and F. Naessens, *Energy-Efficient Communication Processors: Design and Implementation for Emerging Wireless Systems*. Springer Publishing Company, Incorporated, 2013.

[69] A. Pedram, A. Gerstlauer, and R. van de Geijn, "Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator," *IEEE Transactions on Computers*, no. 1, pp. 1–1, 2014.

[70] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, (Los Alamitos, CA, USA), pp. 3–13, IEEE Computer Society Press, 1998.

[71] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (rsvp)," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 141–, IEEE Computer Society, 2003.

[72] G. Weisz and J. C. Hoe, "Coram++: Supporting data-structure-specific memory interfaces for fpga computing," in *25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sept 2015.

[73] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. Ayguad, and M. Valero, "Advanced pattern based memory controller for fpga based hpc applications," in *2014 International Conference on High Performance Computing Simulation (HPCS)*, pp. 287–294, July 2014.

[74] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), pp. 736–749, ACM, 2019.