Theme Article: Top Picks

Towards General-Purpose Acceleration: Finding Structure in Irregularity

Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki
University of California Los Angeles

***Abstract*—Programmable hardware accelerators (e.g., vector processors, GPUs) have been extremely successful at targeting algorithms with regular control and memory patterns to achieve order-of-magnitude performance and energy efficiency improvements. However, they perform far under the peak on important irregular algorithms, like those from graph processing, database querying, genomics, advanced machine learning, and others. This work posits that the primary culprit is specific forms of irregular control flow and memory access. By capturing the problematic behavior at a domain-agnostic level, we propose an accelerator that is sufficiently general, matches domain-specific accelerator performance, and significantly outperforms traditional CPUs and GPUs.**

The slowing improvements of technology scaling are raising the demand for specialized hardware accelerators, especially for increasingly difficult problems. While general-purpose data-processing hardware, like GPUs or other vector architectures, are effective on *regular* algorithms, those with irregularity in their control flow or memory access patterns suffer in performance. As evidence, many domain-specific accelerators have been proposed for "irregular" domains like graph processing,[3,9] compressed neural networks,[4,6,10] databases[12] and genomics. Compared to such architectures, GPUs lose in performance and/or energy efficiency by order-of-magnitude. On the other hand, domain-agnostic architectures are widely applicable, which is valuable for economies of scale and robustness to algorithm change. An important question then is whether it is possible to build a programmable accelerator that is equally as capable as GPUs and vector processors, but better suited to irregular algorithms.

Digital Object Identifier 10.1109/MM.2020.2986199
Date of publication 16 April 2020; date of current version 22 May 2020.

The first step toward this goal is to recognize what makes an algorithm irregular. In computer architecture, the concept of irregularity is often used informally to indicate behavior that causes inefficiency. We argue that to first order, the root cause of most irregularity is *data dependence*. Data dependence can appear in many forms, including control, memory address calculation, conditional memory accesses, reuse and parallelism structure.

Consider the concrete example of sorting algorithms. Merge sort has regular memory and irregular control. The memory that is read or written at each step is predetermined, but the control flow decisions depend on the relative order of the lists to be merged at each step. This data-dependent control prevents speculative execution from being effective, and it also prevents vectorization (cannot know the operands from each lane in advance). Conversely, the radix-sort algorithm has regular control, but irregular memory access (bin increment and scatter). This prevents prefetching from being effective, and also prevents vectorization with standard instructions. In general, data dependence interferes or complicates the fundamental mechanisms that parallel processors use to extract performance.

Insight: To address these challenges, our key observation is that it is not necessary to handle *arbitrary* irregularity because data dependence manifests in common forms across domains. This work suggests two specific forms are critical: *stream join* and *alias-free indirection* (AF-Indirect).

Stream join is defined by in-order processing of data, where only the relative order of consumption and production of new data is dependent on control decisions. These joins are surprisingly common, including merge sort, database joins, and inner product sparse tensor operations. AF-Indirect is characterized by memory access with data-dependent addresses, but where the only memory dependencies are read–modify–write. Relevant kernels include radix-sort, outer product sparse-tensor operations, hash joins, histograms, and synchronous graph processing (e.g., page rank).

These data-dependence forms are not mutually exclusive; in fact they can be thought of as different ways to relax regular (non-data-dependent) algorithms (see Figure 1). An example of a

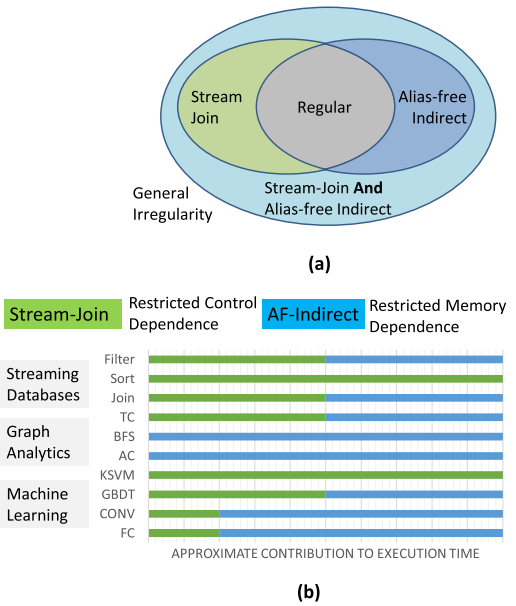

Figure 1. Restricted data-dependence forms cover many algorithms. (a) Algorithm classification. (b) Dependence forms coverage.

workload that requires both forms is triangle counting in graphs. At each vertex, AF-Indirect is used to locate a neighbor node's adjacency list, and stream join can be used to find common neighbors (each indicating a triangle).

Approach: Critically, we find that these restricted forms of data-dependence can serve as abstractions, which can be exploited in hardware. We use this insight to construct our approach to design a "general-purpose" accelerator. Specifically, we start with an architecture known to work well for regular algorithms: a systolic-style coarse-grained reconfigurable architecture (CGRA) with streaming memory support. We then develop hardware and software mechanisms for our two restricted data-dependence forms.

Our design is called the sparse processing unit (SPU). SPU supports fully pipelined stream joins with a systolic CGRA augmented with a novel dataflow-control model. SPU supports high-bandwidth AF-Indirect (load/store/update) with a banked scratchpad with aggressive reordering and embedded compute units for atomic update. Data dependence complicates the support for finer grain data types [naive subword single-instruction–multiple-data (SIMD) is insufficient]. Therefore, we add support to SPU to enable *decomposing* the reconfigurable network

and wide memory access into power-of-two finer grain resources while maintaining data-dependence semantics.

Evaluation and contribution: We study machine learning (ML) as our primary domain, and graph processing and databases to demonstrate generality. SPU achieves between 1.8–7× speedup on artificial intelligence (AI)/ML applications, and SPU's ability to retain performance on dense algorithms led to 4.5× speedup. On graph and database applications, SPU achieves similar performance to domain-specific accelerators with modest performance and power overheads.

Our primary contributions in this work are the identification of the two common exploitable data-dependence forms, and an ISA and hardware mechanisms to support them. More broadly, we believe that taking a domain-agnostic approach can lead to novel insights and foster knowledge transfer across domains.

EXPLOITABLE DATA-DEPENDENCE FORMS

We observe that two restricted forms of data dependence are sufficient to cover many algorithms: *stream join* and *AF-Indirect*. In this section, we first define these forms and give intuition on their performance challenges for existing architectures, and then overview our proposal.

Preliminary Term—"Streams": Both of the dependence forms rely on the concept of *stream* abstractions, so we briefly explain. Streams are simply an ordered sequence of values. Relevant to this work are memory streams, which are sequences of loads or stores with a well-defined pattern.[7,12] Streams are similar to vector accesses, but have no fixed length.

Stream Join

An interesting class of algorithms iterates over each input (each stream) in order, but the total order of operations (and perhaps whether an output is produced) is data dependent. Two relevant kernels are shown in Figure 2. Sparse vector multiplication (a) iterates over two sparse lists (in CSR format) where indices are stored in sorted order, and performs the multiplication if there is a match. The core of the merge kernel (b) iterates over two sorted lists, and at each step outputs the smaller item. Even though the data structures, data types, and purpose are very different, their relationship to data dependence is the same: they both have stream access, but the relative ordering of stream consumption is data dependent (they reuse data from some stream multiple times).

Stream-join definition: A program region that is regular except that the reuse of stream data and the production of outputs may depend on the data.

Problem with CPUs/GPUs and motivation: Because of their data-dependent nature, stream-joins introduce branch mispredictions for CPUs. For GPGPUs, vectorization becomes difficult due to control divergence of single-instruction–multiple threads (SIMT) lanes; also, the memory pattern can diverge between lanes, causing bank conflicts.

To visualize the problem for CPUs, see Figure 2, which shows both the traditional dataflow and proposed stream-join dataflow representation for the examples above. Here, black arrows represent data dependence, and green arrows indicate control.

Figure 2(a) shows that the inner product dataflow can be mapped to a dataflow-based processor like an out-of-order core, but only at low throughput. To explain, note that there is a loop-carried dependence through the control-dependent increment and memory access. This prevents perfect pipelining, and the throughput is limited to one instance of this computation every n cycles, where n is the total latency of these instructions.

Insight: Our insight is that from the perspective of the memory, the control dependence is mostly unnecessary, as most loads at the line-granularity will be performed anyways. Therefore, to break the dependence, we need to separate the loads from computation (this is what memory streams do), then expose a pipelined mechanism for controlling the order of data consumption. In the sparse vector example, we would like to reuse the larger of the two index values for consideration (*data-dependent reuse*). If the comparison instruction can treat its inputs like a queue, and specify the reuse behavior (i.e., pop the smaller element), this can be accomplished in a pipelined fashion.

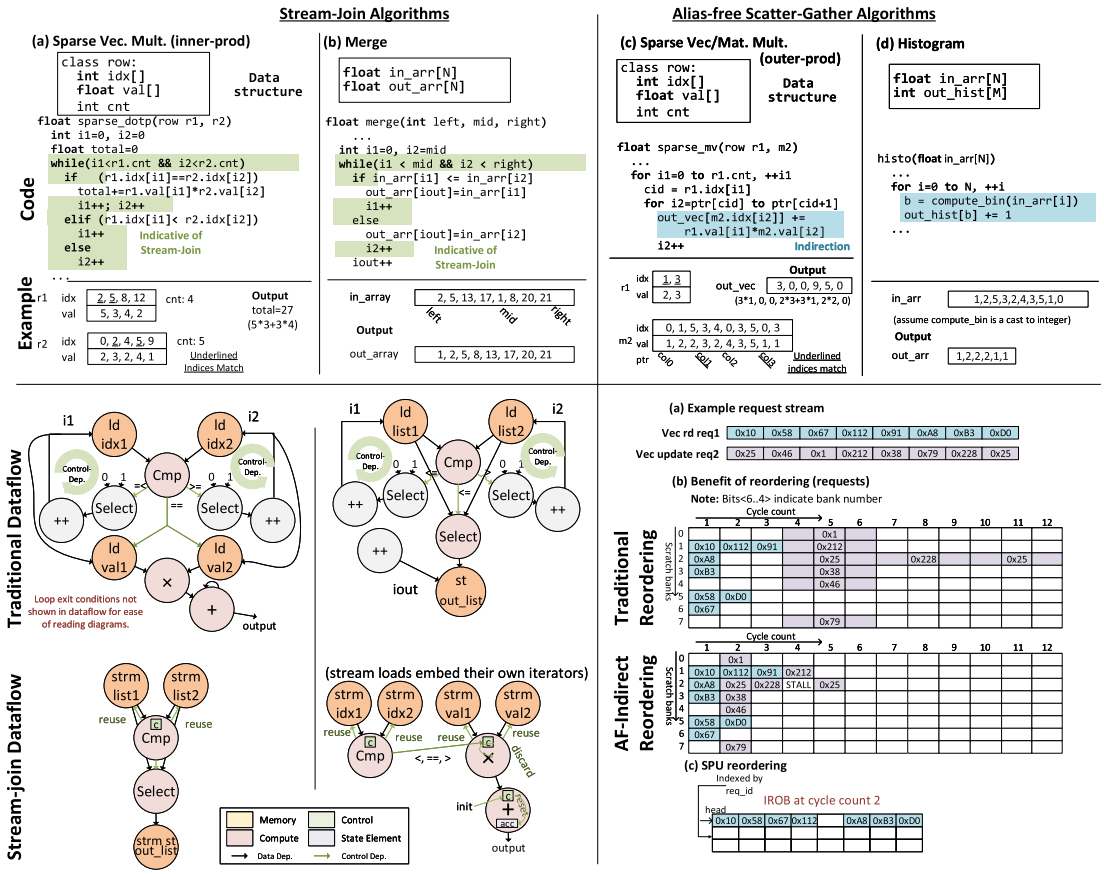

Figure 2. Example restricted dependence form algorithms.

Also, the multiply–accumulate operation is performed on only matching indices, so we should discard some of these computations/data. Therefore, in addition to data-dependent reuse, we also require *data-dependent discard*.

The merge example [see Figure 2(b)] has a surprisingly similar form and control dependence loop to the sparse multiplication, where the computation is replaced by selecting the smaller item. A similar approach of decoupling streams and applying data-dependent reuse and discard will break the control dependence loop and enable high throughput.

Our stream-join proposal: We find the desired behavior can be accomplished with a simple and novel control flow model for full-throughput systolic execution. In this model, each instruction may reuse its inputs, discard the computation, or reset a register based on a dataflow input. Figure 2 shows the examples written in this model.

To enable flexible control interpretation, each instruction embeds a simple configurable mapping function from the instruction output and control input to the control operations

$$f(\text{inst\_out}, \text{control\_in}) \rightarrow \text{reuse1}, \text{reuse2}, \text{discard}, \text{reset}.$$

Alias-Free Indirection

Many algorithms rely on indirect read, write, and update to memory, often showing up as a[f(b[i])]. Figure 2 shows two examples: The sparse-vector/sparse-matrix outer product (c) works by performing all combinations of non-zero multiplications, and accumulating in the correct location in a dense output vector. Histogram (d) is straightforward. Both perform a read–modify–write access to an indirect location. This can be viewed as two dependent streams. Another important observation is that there are no unknown aliases between streams—the only dependence is between the load and store of the indirect update.

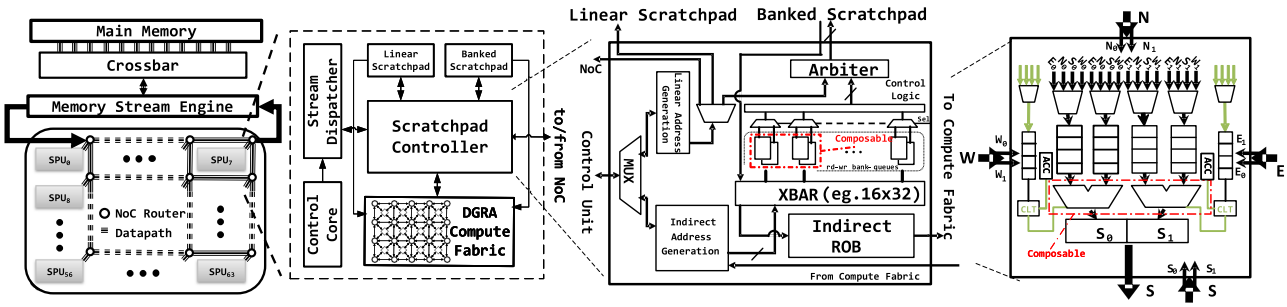

Figure 3. SPU microarchitecture. (a) Sparse Processing Unit (SPU). (b) SPU Core. (c) Scratchpad Controller. (d) DGRA Processing Element.

AF-Indirect Definition: A program region that is regular (including no implicit dependencies) except that the address of one memory stream may depend on another, and a stream can encode a read–modify–write operation.

Problem for CPUs/GPUs and our motivation: On CPUs, indirect memory is possible with scatter/gather, however the throughput is limited given the limited ports to read/write vector-length number of cache lines simultaneously. Also, not leveraging alias-freedom means a reliance on expensive load-store queues.

Although GPUs can use their banked scratchpads for faster indirect access, the following two reasons limit the indirect throughput. 1. No reordering of requests across subsequent vector warp accesses.[11] Doing so in a GPU would require dependence checking of in-flight accesses, as they cannot guarantee alias freedom. 2. Atomic updates to the same scratchpad bank are not pipelined even though they access different memory locations. The reason is that the lock bits for atomic operations are shared among multiple addressable locations.[1] The coarse-granularity locking is required to reduce the locking overhead.

To visualize the inefficiency of a typical GPU scratchpad, see Figure 2, which shows how scratchpad vector requests (corresponding to indirect read and atomic update, respectively) are served on a GPU. For simplicity, we assume a warp size of 8. As GPUs do not reorder requests across warps, the update request vector is issued after the completion of all read requests. For updates in GPU, we assume one lock-bit per scratchpad bank. Here, the three-cycle nonpipelineable operation further worsens the overhead of bank conflicts.

Insight: Our insight is that the dependence check is not required between subsequent requests (e.g., corresponding to different static loads) if alias freedom is known. Further, the atomic operations required in these algorithms are often low latency integer arithmetic logic unit (ALU) operations. Therefore, for dependence check, the maximum number of possible conflicting addresses is usually low (i.e., 1 less than the atomic update latency). Hence, we could compare with absolute addresses instead of relying on a serializing lock bit mechanism.

Our AF-indirect proposal: We find that the desired behavior can be accomplished by: 1) exposing alias-freedom in the hardware–software interface to enable interleaving across vectors: and 2) storing absolute address of pending atomic updates (maximum 2) to enable pipelining of nonconflicting addresses. Figure 2 shows how SPU is able to reorder requests, and also able to pipeline atomic update requests with initiation with no bubbles. The stall is introduced in the presence of "real" dependencies, for example, see cycle-4 in AF-Indirect reordering in Figure 2. This is limited to a maximum two-cycle bubble.

SPARSE PROCESSING UNIT

In this section, we first overview the primary aspects of the design, and then provide the details of stream-join-enabled systolic-CGRA and the banked memory exposed to knowledge of AF-Indirect.

Figure 3(a) shows the proposed SPU architecture. SPU cores are integrated into a mesh network-on-chip (NoC). Each core is composed

of the specialized memory and compute fabric: decomposable granularity reconfigurable architecture (DGRA), together with a control core for coordination among streams.

Communication/synchronization: SPU provides two specialized mechanisms for communication. First, we include the multicast capability in the network. Data can be broadcast to a subset of cores, using the relative offset in the scratchpad. As a specialization for loading main memory, cores issue their load requests to a centralized memory stream engine, and data can be multicast from there to relevant cores. For synchronizing on data-readiness, SPU uses a dataflow-tracker-like mechanism to wait on a count of remote-scratchpad writes.

SPU Core

The basic operation of each core [see Figure 3 (b)] is that the control core will first configure the DGRA for a particular dataflow computation, and then send stream commands to the scratchpad controller to read data or write to the DGRA, which itself has an input and output port interface to buffer data.

Stream-join compute fabric: DGRA: We augment a systolic CGRA to support stream-join control and dataflow computation with arbitrary data types. Figure 3(d) shows the microarchitecture of a DGRA processing element (PE) (green color represents control).

To implement *control interpretation*, we add a control lookup table (CLT) to each functional unit (FU), which determines a mapping between the control inputs and possible control operations. This mapping is configured along with the dataflow computation graph. During dataflow operation, CLT consumes one of the dataflow inputs to produce control signals for the ALU (discard), associated registers (reset), and FIFOs connected to ALU inputs (reuse).

In the DGRA, we enable each coarse-grained resource to be able to be decomposed to powers-of-two fine-grain resources. For computation, the decomposable PE can split each coarse-grained input into multiple finer-grained inputs [16-b inputs in Figure 3(d)], which are used to feed two separate lower granularity ALUs. Correspondingly, CLT and registers are also composable.

To route the data from PEs, the network of the DGRA is decomposable into multiple parallel finer-grain subnetworks (minimum 8 b). For flexible routing, we add the ability for incoming values to shift one subnetwork per switch hop.

Alias-freedom-exposed banked memory: Because our workloads often require a mix of linear and indirect arrays simultaneously, for example, streaming read of indices (direct) and associated values (indirect), we begin our design with two logical scratchpad memories, one highly *banked* and one *linear*. In this design, both exist within the same address space. Hence, memory streams may access locations in a remote core's scratchpad using the similar interface for linear and indirect streams.

The role of the scratchpad controller [see Figure 3(c)] is to generate requests for reads/writes to the linear scratchpad, and reads/writes/updates to the indirect scratchpad. A control unit assigns the scratchpad streams, and their state is maintained in either linear or indirect stream address generation logic. The controller should then select between any concurrent streams for address generation and send it to the associated scratchpad to maximize expected bandwidth. The linear address generator's operation is simple—create wide scratchpad requests using the linear access pattern.

The indirect address generator creates a vector of requests by combining each element of the stream of addresses (coming from the compute fabric, explained in the "Exploitable Data-Dependence Forms" section) with each element in the parent stream (i.e., b[i] in a[f(b[i])]). This vector of requests is sent to an arbitrated crossbar for distribution to banks, and a set of queues buffer requests for each static random access memory (SRAM) bank until they can be serviced.

Since there are no conflicts among indirect read/write requests, the requests are serviced from the top of the bank queue as soon as the scratchpad data bus becomes available. For atomic update requests, the requests can be serviced when both scratchpad read and write buses are available, and the updated address

does not conflict with the pending updates issued from the same bank. As the ordering of the data returned from read requests is critical for dataflow operations, we employ an indirect read reorder buffer (IROB) that maintains incomplete requests in a circular buffer (see Figure 2). IROB entries are deallocated in-order when a request's data is sent to the compute unit.

Control ISA: We leverage an open-source stream-dataflow ISA[7] for the control core's implementation of streams, and add support for indirect reads/writes/updates, stream-join dataflow model, and typed dataflow graph. The ISA contains stream instructions for the data transfer, including reading/writing to main memory and scratchpad.

METHODOLOGY

SPU: We implemented SPU's DGRA in Chisel, and implemented with an industry 28-nm technology. We built an SPU simulator in gem5, using a RISCV ISA for the control core.

Architecture comparison points: Table 1 shows the characteristics of the architectures we compare against, including their on-chip memory sizes, FU composition, and memory bandwidth. We also address whether an inorder processor is sufficient by comparing against "SPU-inorder," where the DGRA is replaced by an array of eight inorder cores (total of 512 cores). For reference, we also compared against a dual-socket Intel Skylake CPU, with 24 cores.

Workload implementations: We implement SPU kernels (both dense/sparse) for each workload, and use a combination of libraries and hand-written code to compare against CPU/GPU versions.

Table 1. Characteristics of evaluated architectures.

Characteristics	GPU	SPU-inorder	SPU
Processor	GP104	In-order	SPU-core
Cache+Scratch	4064 kB	2560 kB	2560 kB
Cores	1792	512	64 SPU cores
FP32 Unit	3584	2048	2432
FP64 Unit	112	512	160
Max Bw	243 GB/s	256 GB/s	256 GB/s

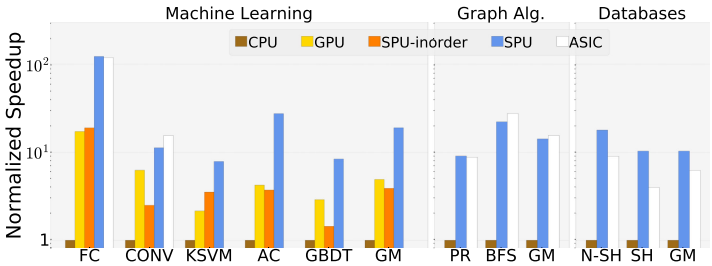

Figure 4. Overall performance.

EVALUATION

Our evaluation broadly addresses the question of whether restricted data-dependence forms exposed to an ISA (and exploited in hardware) can help achieve general-purpose acceleration.

Comparison to general-purpose accelerators: Figure 4 shows how SPU fairs against CPU and GPU for workloads across ML, graph processing, and databases.

The workloads with a stream-join pattern—kernel support vector machines (KSVM), TPCH sort heavy queries (SH), gradient boosting decision trees (GBDT)—achieve speedup up to 10× speedup over CPU due to avoiding the throughput-limiting cyclic dependence loop and lower computational density. The GPU also suffers from hardware underutilization as control leads to masking in vector lanes.

On workloads with AF-Indirect—fully connected layer (FC), convolution layer (CONV), arithmetic circuits (AC), Graph, TPCH not sort-heavy queries (N-SH)—both GPU and SPU use a histogram-based approach. However, SPU's aggressive reordering of indirect updates in the compute-enabled scratchpad far outperforms the limited ordering in GPU.

Finally, the ability to support both stream-join and AF-Indirect enables the use of new compression techniques like run-length encoding efficiently. These techniques effectively reduce the required memory bandwidth, thus improving performance.

Even though SPU-inorder can relieve some of the vectorization overheads suffered by GPU, it is insufficient due to lower peak throughput.

Domain accelerator comparison: Accelerators for FC,[4] CONV,[10] and Graphs[3] all employ compute-enabled banked memory to achieve high

indirect throughput. SPU is able to remain within 57% of its performance. The difference is due to other specializations, e.g., higher radix NoC in Graph application-specified integrated circuit (ASIC) and higher buffer access bandwidth in CONV ASIC.

In non-sort-heavy database workloads, the dense version of SPU performs similar to ASIC. With efficient stream joins, SPU is able to catch up to and surpass database ASIC, which spends significant area resources on specialized sorting units.

Benefit of decomposability: In general, we achieve datawidth-proportional speedup by adding decomposability. In comparison to subword-SIMD, SPU can see 2.6× speedup by being able to vectorize run-length decoding, which involves control-serializing computation. Similarly, branches in AC also benefit from decomposability (3×). Overall, SPU achieves geomean speedup of 2.12× speedup with decomposability.

Area and power: The two major sources of SPU's area are the scratchpad banks and DGRA, together occupying more than two-third of the total; DGRA is the major contributor to power (assuming all PEs are active).

Compared to the whole design, adding stream-join control in the systolic CGRA increases area by 6.6% and power by 17.5%. Decomposability costs another 3.1% area and 7.7% power.

CONCLUSION

This work identifies two forms of data-dependence, which are highly specializable and are broadly applicable to a variety of algorithms. By defining a specialized execution model and codesigned hardware, SPU, we enabled the efficient acceleration of a large range of workloads. We observed up to order-of-magnitude speedups and significant power reductions compared to modern CPUs and GPUs while remaining flexible.

More important than the proposed design is how the approach of identifying and abstracting common dependence forms can influence the field.

Systematic understanding of irregular accelerators: Our restricted forms of data-dependence can be used to classify and understand the fundamental capabilities of domain-specific accelerators. Table 2 shows the scope of several existing domain-specific accelerators. What we see is that each generally specializes for only one form out of stream-join, AF-Indirect or regular algorithms.

Table 2. Analysis of related works.

Exploitable dependence forms	Specialized architectures
No data-dependence	TPUv1—Dense ML
	GPU—Dense
	LSSD[8]—Dense
	SPU
Stream-Join	Q100[12]—Database
	Sparse ML[6]—Sparse Algebra
	ExTensor[5]—Tensor
	SPU
AF-Indirect	SCNN[10]—DNN CONV
	EIE[4]—DNN FC
	OuterSPACE[9]—Sparse Algebra
	Graphicionado[3]—Graphs
	SPU

Beyond simply understanding the space, this way of viewing algorithm's interaction with architecture can improve the portability of techniques across domains. Consider the context of accelerators for sparse linear algebra. SPU's design can join sparse lists at one element per cycle (per PE). An idea proposed for a sparse ML accelerator is to vectorize the join,[6] so that N elements can be joined at once from each list (requiring $N \times N$ comparisons). The ExTensor accelerator,[5] designed for multidimensional sparse tensor ops, goes further. It demonstrates that a hierarchical list intersection (a form of stream join) can be more work efficient by skipping a variable number of unmatched items in a single step. To further reduce the memory bandwidth overhead of sparsity, SparTen[2] proposed a bit-vector representation of indices. Thus, the matched indices can be found using efficient

bit-level operations. These optimizations can apply to SPU. More importantly, by finding structure and commonality in the dependence forms across domains, it becomes clear how to apply these optimizations to other superficially different problems, like database join or decision tree training.

Impact on general-purpose processors: This work focused on reconfigurable dataflow-like processors for implementing dependence-form specialization. While it was convenient, other architectures can equally benefit from such specialization.

- *Indirection in GPUs*: A conceivable extension to a GPU ISA could enable the annotation of a program region as being alias-free indirect (informed by programmer or compiler). This would allow GPU scratchpads to eliminate memory dependence checking and enable aggressive reordering, leading to reduced impact of bank conflicts and higher throughput. NVIDIA's tensor core is precedence that such specialization is feasible.
- *Stream-join SIMD*: Stream-join control could be supported in a CPU, for example, through extensions to SIMD operations. An approach could be to add specialized instructions, which allow treating registers as FIFOs, and the branch instructions may control the order of data consumption (using simple finite-state machine at FIFOs).
- Hybrid FPGAs: Recent FPGAs (Xilinx Alveo) include neural network accelerator units, demonstrating the need for specialization of even reconfigurable hardware. Increasing these units' flexibility to be similar to SPU could simultaneously provide many of the same efficiency benefits as an ASIC while also retaining the fundamental value proposition of FPGAs: broad workload efficiency while retaining fine-grain reprogrammability.

Other exploitable data-dependence forms: It is possible that there may be alternate formulations or definitions of restricted data-dependence forms, which could lead to new opportunities for specialization. For example, a coarser grain form of data dependence than we have explored is data-dependent parallelism (aka dynamic parallelism). At the other end of the spectrum could be data-dependent data types, where at a fine grain, the data-type size is chosen to meet the precision requirements. One could imagine exposing these forms as first-class primitives in the hardware/software interface, and each could be plausibly useful in many domains.

We think it is important not to forget that systems and application experts are constantly innovating new algorithms, and are now doing so with deep knowledge of the underlying hardware. Our results support the notion that a rigid architecture can limit certain algorithmic approaches from being viable.

Effect on algorithms: Finally, we think it is important not to forget that systems and application experts are constantly innovating new algorithms, and are now doing so with deep knowledge of the underlying hardware. Our results support the notion that a rigid architecture can limit certain algorithmic approaches from being viable. Therefore, we believe that incorporating support for structured irregularity into existing and new programmable architectures can lead to innovations in novel algorithms and data structures.

ACKNOWLEDGMENTS

We would like to thank G. Van den Broeck and A. Choi for their insights and help with arithmetic circuits workloads. We would also like to thank D. Ott and P. Subrahmanyam for their thoughtful conversations on the nature of irregularity and data dependence. This work was supported in part by the National Science Foundation under Grant CCF-1751400 and Grant CCF-1937599 and in part by the gift funding from VMware.

REFERENCES

1. J. Gomez-Luna, J. M. Gonzalez-Linares, J. I. Benavides Benitez, and N. Guil Mata, "Performance modeling of atomic additions on GPU scratchpad memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 11, pp. 2273–2282, Nov. 2013.

2. A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 151–165.
3. T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, Oct. 2016, pp. 1–13.
4. S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 243–254.
5. K. Hegde *et al.*, "ExTensor: An accelerator for sparse tensor algebra," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 319–333.
6. A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, "Fine-grained accelerators for sparse machine learning workloads," in *Proc. 22nd Asia South Pacific Design Autom. Conf.*, 2017, pp. 635–640.
7. T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 416–429.
8. T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Pushing the limits of accelerator efficiency while retaining programmability," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Mar. 2016, pp. 27–39.
9. S. Pal *et al.*, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2018, pp. 724–736.
10. A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 27–40.
11. NVIDIA Whitepaper, "*Cuda C best practices guide*," May 2019. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf
12. 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 *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2014, pp. 255–268.

Vidushi Dadu is currently working toward the Ph.D. degree with the Department of Computer Science, University of California Los Angeles. Her current research focuses on hardware–software codesign to enable general-purpose acceleration. Dadu received the B.Tech. degree in electronics and communication engineering from the Indian Institute of Technology Roorkee. She is a student member of IEEE. Contact her at vidushi.dadu@cs.ucla.edu.

Jian Weng is currently working toward the Ph.D. degree with the Department of Computer Science, University of California Los Angeles. His research interests include analyzing and designing reconfigurable spatial architectures along with the associated compilation techniques. Weng received the B.Eng. degree in computer science from Shanghai Jiao Tong University. He is a member of the Association of Computing Machinery. Contact him at jian.weng@cs.ucla.edu.

Sihao Liu is currently working toward the Ph.D. degree with the Department of Computer Science, University of California Los Angeles. His research interests include spatial architecture prototyping and design space exploration. Liu received the B.Eng. degree in electrical engineering from Xi'an Jiaotong University. He is a student member of IEEE. Contact him at sihao@cs.ucla.edu.

Tony Nowatzki is currently an Assistant Professor with the Department of Computer Science, University of California Los Angeles. His research interests include architecture and compiler codesign and novel hardware/software interfaces. Nowatzki received the Ph.D. degree in computer science from the University of Wisconsin-Madison. He is a member of IEEE. Contact him at tjn@cs.ucla.edu.