Stream-Dataflow Acceleration

Tony Nowatzki[*†] Vinay Gangadhar[†] Newsha Ardalani[†] Karthikeyan Sankaralingam[†]

[*]University of California, Los Angeles [†] University of Wisconsin, Madison

tjn@cs.ucla.edu vinay,newsha,karu@cs.wisc.edu

ABSTRACT

Demand for low-power data processing hardware continues to rise inexorably. Existing programmable and "general purpose" solutions (eg. SIMD, GPGPUs) are insufficient, as evidenced by the order-of-magnitude improvements and industry adoption of application and domain-specific accelerators in important areas like machine learning, computer vision and big data. The stark tradeoffs between efficiency and generality at these two extremes poses a difficult question: how could domain-specific hardware efficiency be achieved without domain-specific hardware solutions?

In this work, we rely on the insight that "acceleratable" algorithms have broad common properties: high computational intensity with long phases, simple control patterns and dependences, and simple streaming memory access and reuse patterns. We define a general architecture (a hardware-software interface) which can more efficiently expresses programs with these properties called *stream-dataflow*. The dataflow component of this architecture enables high concurrency, and the stream component enables communication and coordination at very-low power and area overhead. This paper explores the hardware and software implications, describes its detailed microarchitecture, and evaluates an implementation. Compared to a state-of-the-art domain specific accelerator (DianNao), and fixed-function accelerators for MachSuite, Softbrain can match their performance with only 2× power overhead on average.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; *Reconfigurable computing*; *Data flow architectures*; Single instruction, multiple data; Special purpose systems;

KEYWORDS

Streaming, Dataflow, Architecture, Accelerator, Reconfigurable, CGRA, Programmable, Domain-Specific

ACM Reference format:
Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080255

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06...$15.00
https://doi.org/10.1145/3079856.3080255

1 INTRODUCTION

Data processing hardware is vital to the global economy – from the scale of web services and warehouse computing, to networked Internet of Things and personal mobile devices. As application needs in these areas have evolved, general purpose techniques (even SIMD and GPGPUs) are not sufficient and have fallen out of focus, because of the energy and performance overheads of traditional VonNeumann architectures.

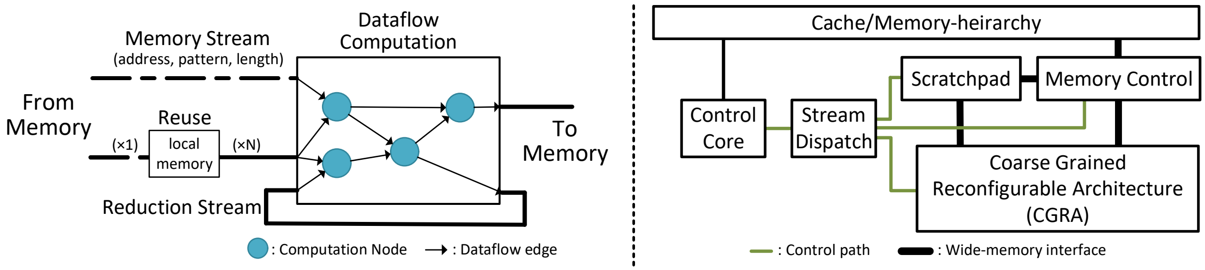
Instead, application-specific and domain-specific hardware is prevailing. For large scale computing, Microsoft has deployed the Catapult FPGA accelerator [25] in its datacenters, and likewise for Google's Tensor Processing Unit for distributed machine learning [12]. Internet of things devices and modern mobile systems on chip (SOCs) are already laden with custom hardware, and innovation continues in this space with companies (eg. Movidius) developing specialized processors for computer vision [11].

While more narrow hardware solutions are effective, they pose many challenges. As algorithms change at an alarming rate, hardware must be redesigned and re-verified, which is burdensome in terms of development cost and time-to-market. As a corollary, innovation in algorithms becomes more difficult without access to flexible hardware. Furthermore, programmable hardware can be time-shared across applications, while domain-specific cannot, making it more costly in terms of silicon. Finally, from the academic viewpoint, it is difficult to formalize and apply improvements from domain-specific hardware to the broader field of computer architecture – limiting the intellectual impact of such work.

Ideally, what we require is hardware that is capable of executing data-intensive algorithms at high performance with much lower power than existing programmable architectures, while remaining broadly applicable and adaptable.

An important observation, as alluded to in the literature [9, 21], is that typically-accelerated workloads have common characteristics: 1. High computational intensity with long phases; 2. Small instruction footprints with simple control flow, 3. Straightforward memory access and re-use patterns. The reason for this is simple: these properties lend themselves to very efficient hardware implementations through exploitation of concurrency. Existing data-parallel hardware solutions perform well on these workloads, but in their attempt to be far more general, sacrifice too much efficiency to supplant domain-specific hardware. As an example, short-vector SIMD relies on inefficient general pipelines for control and address generation, *but accelerated codes typically do not have complex control and memory access*. GPGPUs hide memory latency using hardware for massive-multithreading, *but accelerated codes' memory access patterns can usually be trivially decoupled without multithreading*.

To take advantage of this opportunity, this work proposes an



(a) Stream-Dataflow Architecture Abstractions

(b) Softbrain: Stream-Dataflow Implementation

Figure 1: Stream-Dataflow Abstractions & Implementation

architecture and execution model for acceleratable workloads, whose hardware implementation can approach the power and area efficiency of specialized designs, while remaining flexible across application domains. Because of its components, it is called *stream-dataflow*, and exposes these basic abstractions:

- A dataflow graph for repeated, pipelined computations.
- Stream-based commands for facilitating efficient data-movement across components and to memory.
- A private (scratchpad) address space for efficient data reuse.

Figure 1(a) depicts the programmer view of stream-dataflow, consisting of the dataflow graph itself, and explicit stream communication for memory access, read reuse and recurrence. The abstractions lead to an intuitive hardware implementation; Figure 1(b) shows our high-level design. It consists of a coarse grain reconfigurable architecture (CGRA) and scratchpad, connected with wide buses to memory. It is controlled from a simple control core, which sends stream commands to be executed concurrently by the memory control engine, scratchpad control engine and the CGRA. This coarse grain nature of the stream-based interface enables the core to be quite simple without sacrificing highly-parallel execution. The stream access patterns and restricted memory semantics also enable efficient address generation and coordination hardware.

Relative to a domain specific architecture, a stream-dataflow processor can reconfigure its datapath and memory streams, so it is far more general and adaptable. Relative to existing solutions like GPGPUs or short-vector SIMD, the power and area overheads are significantly less on amenable workloads. An implementation can also be deployed flexibly in a variety of settings, either as a standalone chip or as a block on a SoC; it could be integrated with virtual memory, and either use caches or directly access memory.

In this paper, we first define stream-dataflow, describe its execution model, and explain why it provides specialization benefits over existing architectures. We then discuss the ISA and programmability before describing the microarchitecture of our implementation, Softbrain. To demonstrate the generality of this architecture and this implementation's capabilities, we compare against a state-of-the-art machine learning accelerator, as well as fixed function accelerators for MachSuite workloads. Our evaluation shows it can achieve equivalent performance to the accelerators, with orders-of-magnitude area and power efficiency improvement over CPUs. Compared to the machine learning accelerator, we average only 2× power and area overhead. On the broader set of MachSuite workloads, compared to custom ASICs, the average overhead was 2× power and 8× area.

2 MOTIVATION AND OVERVIEW

For a broad class of data-processing algorithms, domain-specific hardware provides orders of magnitude performance and energy benefits over existing general purpose solutions. By definition, the strategy that domain-specific accelerators employ is to limit the programming interface to support a much narrower set of functionality suitable for the domain, and in doing so simplify the hardware design and improve efficiency. We further hypothesize that the efficiency gap between domain-specific and general purpose architectures is fundamental to the way general purpose programs are expressed at an instruction-level, rather than a facet of the microarchitectural mechanisms employed.

So far, existing programmable architectures (eg. SIMD, SIMT, Spatial) have shown some promise, but have only had limited success in providing a hardware/software interface that enables the same specialized microarchitecture techniques that more customized designs have employed.

Therefore, our motivation is to discover what are the architectural abstractions that would enable microarchitectures with the execution style and efficiency of a customized design, at least for a broad and important class of applications that have long phases of data-processing and streaming memory behavior. To get insights into the limitations of current architectures and opportunities, this section examines the specialization mechanisms of existing programmable hardware paradigms. We then discuss how their limitations can inspire a new set of architecture abstractions. Overall, we believe that the stream-dataflow abstractions we propose could serve as the basis for future programmable accelerator innovation.

2.1 Specialization in Existing Approaches

Several fundamentally different architecture paradigms have been explored in an attempt to enable programmable hardware specialization; prominent examples are depicted in Figure 2. We discuss their specialization capabilities in three broad categories:
1. Reducing the per-instruction power and resource access costs,
2. Reducing the cost of memory addressing and communication, and
3. Reducing the cost of attaining high execution resource utilization.
Table 1 provides a summary, and we discuss these in detail below.

SIMD and SIMT: Both SIMD and SIMT provide fixed-length vector abstractions in their ISA, which enables microarchitectures that amortize instruction dispatch and enable fewer, wider memory accesses. The specification of computation at the instruction-level, however, means that neither can avoid instruction communication through large register files.

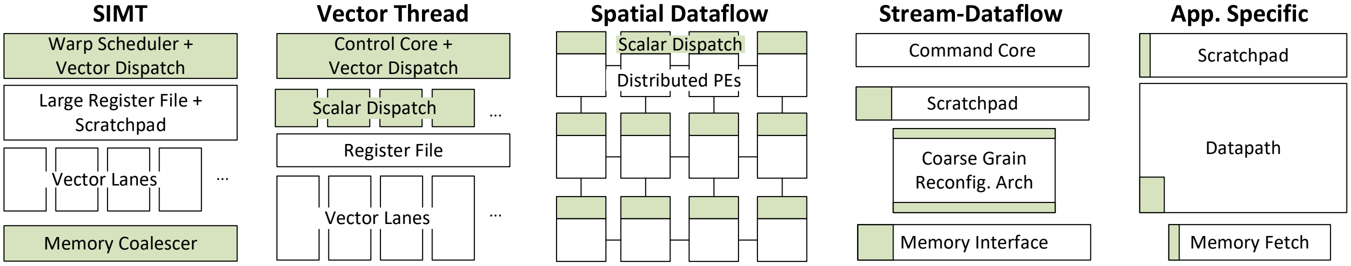


Figure 2: High-Level Architecture Organizations (Green regions denote control)

	Potential Specialization Capability	SIMD	SIMT	Vector Threads	Spatial Dataflow	Stream-Dataflow
Instr.	Amortize instruction dispatch	Yes	Yes	Yes SIMD/ No Scalar	Somewhat	Yes
	Reduce control divergence penalty	No	Somewhat	Yes	Yes	Somewhat
	Avoids large register file access	No	No	No	Yes	Yes
Memory	Coalesce spatially-local memory access	Yes	Yes	Yes SIMD/ No Scalar	No	Yes
	Avoid redundant addr. gen. for spatial access	Yes	No	Yes SIMD/ No Scalar	No	Yes
	Provide efficient memory for data reuse	No	Yes	No	No	Yes
Util.	Avoid multi-issue logic	No	Yes	No	Yes	Yes
	Avoid multi-threading logic and state	Yes	No	Yes	Yes	Yes

Table 1: Architectural Specialization Capabilities (Assumption: High-parallelism, Small-Footprint Compute Kernels)

In addition, neither architecture enables inexpensive support of high hardware utilization. Because the vector length is fixed and relatively short, short-vector SIMD processors constantly rely on the general purpose core to dynamically schedule parallel instructions. Scaling issue width, reordering logic and register file ports is expensive in area and power. SIMT exposes massive-multithreading capability to enable high hardware utilization, by allowing concurrent execution of many warps, which are groups of threads that issue together. This requires large register files to hold live state, warp-scheduling hardware, and incurs cache pressure from many independent warps.

Both SIMT and SIMD have some unique limitations. SIMD's specification of control-flow through masking and merging introduces additional instruction overhead. Also, typical short-vector SIMD extensions lack programmable scratchpads for efficient data reuse. SIMT threads are programmed to operate on scalar data, and threads within a warp typically perform redundant address generation for spatially local access, and additional logic coalesces common cross-thread access patterns.

Vector Threads ([15, 16]): A vector-thread architecture is similar to SIMD, but exposes the programming capability to specify both SIMD-style and scalar execution of the computation lanes. While this does eliminate the control divergence penalty, it faces many of the same limitations as SIMD. Specifically, it cannot avoid register file access due to the instruction-level specification of computation, and the limited vector-length means the control core's pipeline is relied on to achieve high utilization.

Spatial Dataflow ([2, 23, 32, 33]): Spatial dataflow architectures expose the communication channels of an underlying computation fabric through their hardware/software interface. This enables a distributed instruction dispatch, and eliminates the need for register file access between live instructions. The dispatch overheads can be somewhat amortized using a configuration step. The distributed nature of this abstraction also enables high utilization without the need for multi-threading or multi-issue logic.

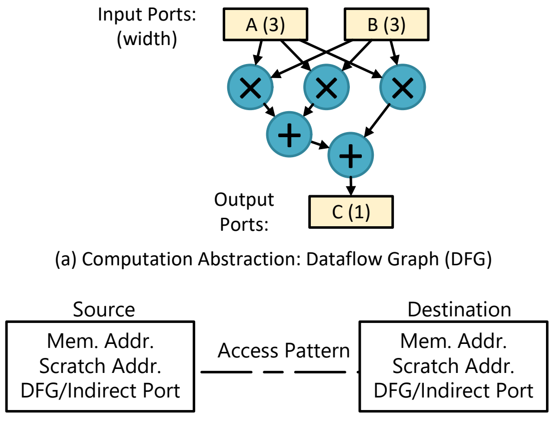
However, these architectures are unable to specialize the memory access to the same degree. Microarchitectural implementations typically perform redundant address generation and issue more and smaller cache accesses for spatially-local access. This is because the spatially distributed memory address generation and accesses are more difficult to coalesce into vector-memory operations.

Summary and Observations: First, being able to specify vectorized memory access is extremely important, not just for parallelism and reducing memory accesses, but also for reducing address generation overhead. On the other hand, though vectorized instructions do reduce instruction dispatch overhead, the separation of the work into fixed-length instructions requires inefficient operand communication through register files and requires high-power mechanisms to attain high utilization. Exposing a spatial dataflow substrate to software solves the above, but complicates and disrupts the ability to specify and take advantage of vectorized memory access.

2.2 Opportunities for Stream-Dataflow

At the heart of the previous observations lies the fundamental tradeoff between vector and spatial architectures: *vector architectures expose a far more efficient parallel memory interface, while spatial architectures expose a far more efficient parallel computation interface*. For it to be conceivable that a programmable architecture can be competitive with an application-specific or domain-specific one, it must expose both efficient interfaces.

Opportunities and Overview: While achieving the benefits of spatial and vector architectures in the general case is perhaps impossible,



(a) Computation Abstraction: Dataflow Graph (DFG)

(b) Communication Abstraction: Stream Command

Figure 3: Stream-Dataflow Abstractions

we argue that it *is* possible in a restricted but important workload setting. In particular, many data-processing algorithms exhibit the property where their computation and memory access components can be specified independently. Following the principles of decoupled access/execute [30], we propose an architecture combining stream and dataflow abstractions – stream-dataflow. The stream component exposes a vector-like memory interface, and the dataflow component exposes a spatial specification of computation.

The following is a brief overview of the abstractions, as shown in Figure 3. The stream interface provides support for an ordered set of stream commands, which is embedded within an existing Von Neumann ISA. Stream commands specify long and concurrent patterns of memory access. Expressible patterns include contiguous, strided, and indirect. We add a separate "scratchpad" address space, which can be used to efficiently collect and access re-used data. Finally, the dataflow interface exposes instructions and their dependences through a dataflow graph (DFG). The input and output interfaces of the DFG are named ports with configurable width, that are the sources and destinations for stream commands.

Microarchitecture: A standard hardware implementation consists of a coarse grained reconfigurable architecture (CGRA) for computation, a programmable scratchpad, stream engines to process the commands for memory or scratchpad access and a control core to generate stream commands. A stream dispatcher enforces architectural dependences between streams. The control core can execute arbitrary programs, but is programmed to offload as much work as possible to the stream-dataflow hardware.

Capabilities and Comparison: Stream-dataflow implementations enable coalesced memory access and avoid redundant address generation by breaking down streaming access into memory-interface size requests. Flexible-length (typically long) stream commands mean the control core can be very simple, as it is needed only to generate streams (not to manage their execution). High utilization is provided without multi-threading or multi-issue pipelines by using a dataflow computation substrate, which also avoids large register file access to communicate values. A disadvantage is that fine-grain control relies on predication (power overhead), and coarse-grain control requires reconfiguration (performance overhead). Luckily, both are somewhat rare in typically accelerated codes.

3 STREAM-DATAFLOW ARCHITECTURE

In this section we describe the stream-dataflow architecture through its programming abstractions, execution model and ISA.

3.1 Abstractions

Stream-dataflow abstracts computation as a dataflow graph and data movement with streams and barriers (Figure 3). Commands expressing these abstractions are embedded in a general program. The following describes these abstractions and associated commands.

Dataflow Graph (DFG): The DFG is an acyclic graph containing instructions and dependences, ultimately mapped to the computation substrate. Note that we do support cycles for direct accumulation, where an instruction produces a value accumulated by a later instance of itself. More general cyclic dependences *are* supported through "recurrence streams," which we describe later.

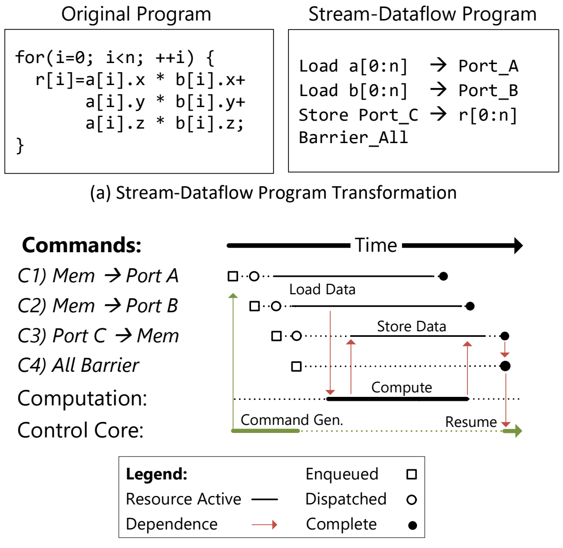
DFG inputs and outputs are named ports with explicit vector widths, which serve as the inputs and outputs of data streams, facilitating communication. For every set of inputs that arrive at the input ports, one set of outputs are generated. One iteration of the entire dataflow-graph execution through input ports, computation nodes and output ports is a *computation instance*. Dataflow graphs can be switched through a configuration command.

Streams: Streams are defined by a source architectural location, a destination and an access pattern. Since a private scratchpad address space is exposed, locations are either a memory or programmable scratchpad address, or a named port. Ports either represent communication channels to the inputs or outputs of the DFG, or they can be indirect ports which are used to facilitate indirect memory access. Streams from DFG outputs to inputs support recurrence. Access patterns for DFG ports are first-in-first-out (FIFO) only, while access patterns for scratchpad and memory can be more complex (linear, strided, repeating, indirect, scatter-gather etc.), but a restricted subset of patterns may be chosen at the expense of generality.

Streams generally execute concurrently, but streams with the same DFG port must logically execute in program order, and streams that write from output DFG ports wait until that data is available. Also, streams from DFG outputs to DFG inputs can be used to represent inter-iteration dependences.

Barriers and Concurrency: Barrier instructions serialize the execution of certain types of commands. They include a location, either scratchpad or memory, and a direction (read or write). The semantics is that any following stream command must logically enforce the happens-before relationship between itself and any prior commands described by the barrier. For example, a scratch read barrier would enforce that a scratch write which accesses address *A* must come after a scratch read of *A* issued before the barrier. Barriers can also optionally serialize the control core, to coordinate when data is available to the host.

Note that in the absence of barriers, all streams are allowed to execute concurrently. Therefore, if two stream-dataflow commands read and write the same scratchpad or memory address, with no barrier in-between them, the semantics of that operation are undefined. The same is true between the host core memory instructions and the stream-dataflow commands. The programmer or compiler is responsible for enforcing memory dependences.



Original Program

```
for(i=0; i<n; ++i) {
  r[i]=a[i].x * b[i].x+
       a[i].y * b[i].y+
       a[i].z * b[i].z;
}
```

Stream-Dataflow Program

```
Load a[0:n]   → Port_A
Load b[0:n]   → Port_B
Store Port_C  → r[0:n]
Barrier_All
```

(a) Stream-Dataflow Program Transformation

(b) Stream-Dataflow Execution

Figure 4: Program Transformation and Execution

3.2 Programming and Execution Model

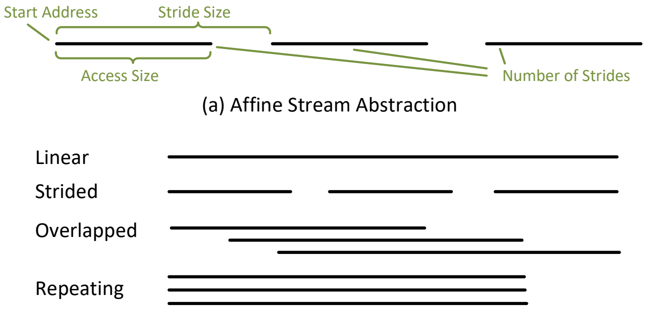
A stream-dataflow program consists of a set of configure, stream, and barrier commands, that interact with and are ordered with respect to the instructions of a general program.

Figure 4(a) shows a vector dot-product code region before and after transformation to the stream-dataflow architecture. The memory streams for the accesses of a, b and r are now explicitly represented, and the computation has been completely removed (it is the DFG in Figure 3). Also note that the loop is completely removed as well (as the loop control is now implicitly coupled with the stream length), signifying the vast reduction in the total number of dynamic instructions required on the control core.

Execution Model: Stream-dataflow programs execute in *phases*, each starting with a stream command to initiate data movement and ending at a final barrier which synchronizes the control core. Phases have arbitrary length consisting of many computation instances. A simple example in Figure 4(b) demonstrates how the execution model exposes concurrency. The state of the stream commands, CGRA, and control core is shown over time. For each stream we note where it was enqueued from the control core, dispatched to execute in parallel, and completed, and we mark the duration in which it has ownership of the source resource for data transfers. The red arrows show dependences between events.

To explain, the first two commands are generated on the control core. They are enqueued for execution and dispatched in sequence, as there are no resource dependences between them. Both streams share the memory fetch bandwidth. As soon as 3 items worth of data are available on ports A and B (3 being the port width), the computation begins. Meanwhile the last two commands are generated and enqueued. As soon as one instance of the computation is complete, the computed data starts streaming to memory. When all data is released into the memory system, the barrier command's condition is met, and the control core resumes.

Performance: The abstractions and execution model lead to intuitive implications for achieving higher-performance. First, the DFG size should be as large as possible to maximize instruction parallelism. Second, streams should be as "long" as possible to avoid instruction overheads on the control core. Third, reused data should be pushed to the scratchpad to reduce bandwidth to memory.



(a) Affine Stream Abstraction

(b) Supported Access Patterns

Figure 5: 2D Affine Access Patterns

3.3 Stream-Dataflow ISA

Because much of a stream-dataflow processor is exposed to the software, careful consideration must go into the encoding of the architecture abstractions into a specific ISA. We discuss here an instance of a stream-dataflow ISA, and its stream commands are summarized in Table 2. These commands would be embedded into the ISA of the control core, typically as 1-3 instructions in a fixed-width RISC ISA.

DFG Specification: The ISA exposes the configurable network of the underlying computation substrate as well as the available communication channels (called vector ports) between data streams and the computation substrate. These two features correspond to an instruction mapping and vector port mapping component of the DFG specification. Instead of proposing a specific DFG encoding, we discuss here the basic requirements for both mapping specifications, and their impact on programmability.

For instruction mapping, most hardware implementations will employ a configuration-based substrate, meaning there will be some maximum number of allowable instructions of each type, per configuration. The hardware may also impose restrictions on the number of inputs or outputs, as well as operand bandwidth.

For vector port mapping, the ISA defines: 1. the number of vector ports, 2. their width (maximum data words transferable per cycle), 3. their depth (the associated buffer size), and 4. their connections to the computation substrate. All of these affect programmability; for example, if there are no vector ports wide enough for a DFG's port, then that DFG cannot be mapped to the hardware. Also, the maximum recurrence length in the program must not exceed the buffering capacity (depth) of the vector ports, or else deadlock can occur[1]. However, it should be noted that the scratchpad or memory can be used for longer dependence chains with the use of barriers.

Though the above introduces significant complexity, our implementation shows that a compiler can automate instruction and vector port mapping. The programmer uses a simple configuration command, SD_Config, to load the DFG configuration from memory.

[1] This is because the recurrence stream would not ever be able to reserve the corresponding vector port. In practice lengths of 16-64 scalar values seem to be sufficient.

Command Name	Parameters	Description
SD_Config	Address, Size	Set CGRA configuration from given Address
SD_Mem_Scratch	Source Mem. Addr., Stride, Access Size, Strides, Dest. Scratch Addr.	Read from memory with pattern to scratchpad
SD_Scratch_Port	Source Scratch Addr., Stride, Access Size, Strides, Input Port #	Read from scratchpad with pattern to input port
SD_Mem_Port	Source Mem. Addr., Stride, Access Size, Strides, Input Port #	Read from memory with pattern to input port #
SD_Const_Port	Constant Value, Num Elements, Input Port #	Send constant value to input port
SD_Clean_Port	Num Elements, Output Port #	Throw away some elements from output port
SD_Port_Port	Output Port #, Num Elements, Input Port #	Issue Recurrence between Input/Output Ports
SD_Port_Scratch	Output Port #, Num Elements, Scratch Addr.	Write from port to scratchpad
SD_Port_Mem	Output Port #, Stride, Access Size, Strides, Destination Mem. Addr.	Write from port to memory with pattern
SD_IndPort_Port	Indirect Port #, Offset Addr., Destination Port	Indirect load from port
SD_IndPort_Mem	Indirect Port #, Output Port #, Offset Addr.	Indirect store to address in indirect port
SD_Barrier_Scratch_Rd	-	Barrier for Scratchpad Reads
SD_Barrier_Scratch_Wr	-	Barrier for Scratchpad Writes
SD_Barrier_All	-	Barrier to wait for all commands completion

Table 2: Stream-Dataflow ISA (Mem: Memory, Scratch: Scratchpad, IndPort: Indirect Vector Port)

Stream Specification: To balance between efficiency and generality, we focus on supporting common address patterns for which we can build efficient hardware. One such pattern is *two-dimensional affine* accesses, defined by an *access size* (size of lowest level access), *stride* (size between consecutive accesses), and *number of strides*. This abstraction, along with different example patterns it can generate, are shown in Figure 5. More formally, these are accesses of the form a[C*i+j], where induction variables *i* and *j* increment from 0 to an upper bound.

The other pattern we support is *indirect*. The source for an indirect stream is another stream's destination port, which it uses to generate memory addresses. The origin stream's data is either treated as an offset from a starting address, or as a pointer value. Indirect streams can be chained to create multi-indirect access patterns (eg. a[b[c[i]]]). We emphasize that only the lowest level of a data-structure access should conform to one of these patterns. The control core is free to generate arbitrary starting addresses to allow more complex overall patterns.

Finally, for supporting control and software pipelining, we added streams for sending constants (SD_Const_Port) and for clearing unneeded values from ports (SD_Clean).

With the possible sources and destinations defined, as well as the possible access patterns, the commands which specify these streams fall-out naturally (See Table 2). Stream commands have parameters for source, destination, pattern and length.

Barrier Specification: This ISA provides three barrier commands. The first two synchronize the reading and writing of the scratchpad memory, and the final one guarantees the phase is complete and the data is visible in the memory system.

Example Program: Figure 6 is an example neural network classifier, essentially a dense matrix-vector multiply of synapses and input neurons. The figure shows the original program, stream-dataflow version and execution behavior. The transformation to a stream-dataflow version pulls the entire loading of neurons and synapses out of the loop using long stream commands (lines 4-7). Here, the input neurons are loaded into scratchpad while simultaneously reading the synapses into a port. Inside the single loop (lines 10-13) are streams which coordinate the accumulator reset, and then send out the final value of each output neuron to memory. Note that the number of instructions executed by the control core is reduced by roughly a factor of Ni.

Original Program:

```
1  //Synapse and input/output neurons  (Ni=784, Nn=10)
2  uint16_t synapse[Nn][Ni], neuron_i[Ni], neuron_n[Nn];
3
4  for (n = 0; n < Nn; n++) { //for each output neuron
5    sum = 0;
6    for (i = 0; i < Ni; i++) //for each input neuron
7      sum += synapse[n][i] * neuron_i[i];
8
9    neuron_n[n] = sigmoid(sum); }
```

Stream-Dataflow Program:

```
1  SD_Config(classifier_config)  //Load CGRA Config.
2
3  //Scratchpad load and memory to port loading
4  SD_Mem_Port(synapse,     Ni * 2, Ni * 2, Nn, Port_S);
5  SD_Mem_Scratch(neuron_i, Ni * 2, Ni * 2, 1,  0);
6  SD_Barrier_Scratch_Wr();
7  SD_Scratch_Port(0, Ni * 2, Ni * 2, 1, Port_N);
8
9  for (n = 0; n < Nn; n++){ //for each output neuron
10   SD_CONST(Port_R,    0,   Ni/4-1);
11   SD_CONST(Port_R,    1,   1);
12   SD_Clean(Port_C,    Ni/4-1);
13   SD_Port_Mem(Port_C, 1, &neuron_n[n]);
14 }
15 SD_Barrier_All();
```

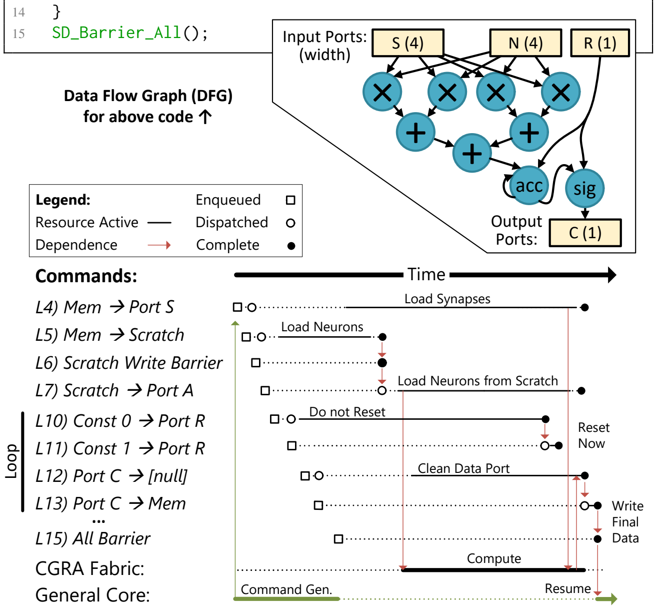


Figure 6: Example Program and Execution

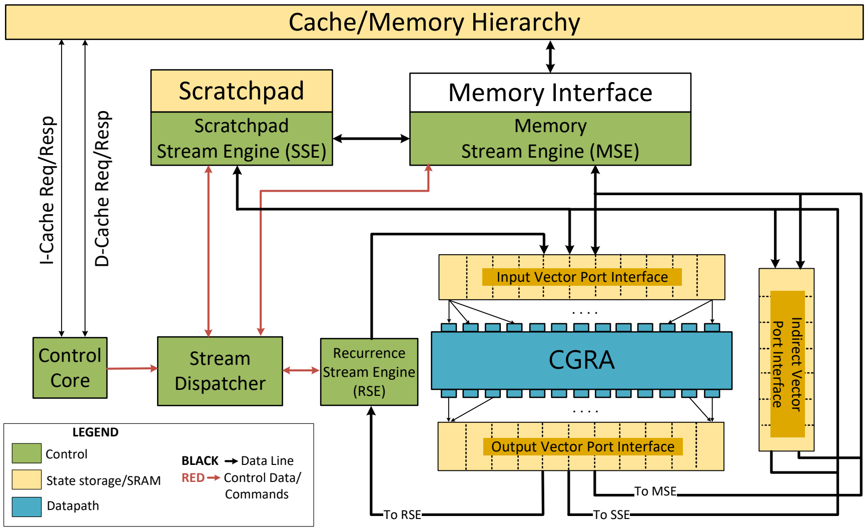

Cache/Memory Hierarchy
I-Cache Req/Resp
D-Cache Req/Resp
Scratchpad
Scratchpad Stream Engine (SSE)
Memory Interface
Memory Stream Engine (MSE)
Input Vector Port Interface
CGRA
Output Vector Port Interface
Indirect Vector Port Interface
Control Core
Stream Dispatcher
Recurrence Stream Engine (RSE)
LEGEND
Control
State storage/SRAM
Datapath
BLACK → Data Line
RED → Control Data/ Commands
To RSE
To MSE
To SSE

Figure 7: Softbrain Overview

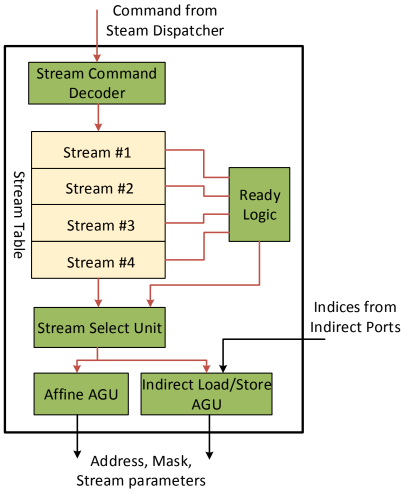


Figure 8: Stream Request Pipeline

4 A STREAM-DATAFLOW MICROARCHITECTURE

Our goal in constructing a microarchitecture for the stream-dataflow ISA is to provide efficiency as close as possible to an application or domain-specific design. Therefore, we adopt two primary design principles: We avoid introducing large or power hungry structures, especially multi-ported memories, and we take full advantage of the concurrency provided by the ISA. Here we describe our microarchitecture implementation, *Softbrain*, and how it accomplishes the above goals by leveraging stream-dataflow architecture primitives.

4.1 Overview

At a high level, we combine a low power control core to generate stream commands, a set of stream-engines to efficiently interface with memories and move data, and a deeply-pipelined reconfigurable dataflow substrate for efficient parallel computation (see Figure 7). There are five primary component types:

- **Control Core** - A low-power single-issue inorder core which generates stream-dataflow commands to the stream dispatcher. It facilitates programmability with low power and area cost.
- **Stream Dispatcher** - This unit manages the concurrent execution of the stream engines by tracking stream resource dependences and issuing commands to stream engines.
- **Stream Engines** - Data access and movement is carried out through three "stream engines", one for memory (facilitating wide access to memory), one for scratchpad (efficient data reuse), and one for DFG recurrences (for immediate reuse without memory storage). The stream-engines arbitrate the access of their respective memory and scratchpad resources across streams assigned to them.
- **Vector Ports** - The vector ports are the interface between the computations performed by the CGRA and the streams of incoming/outgoing data. In addition, a set of vector ports not connected to the CGRA are used for storing the streaming addresses of indirect loads/stores.
- **CGRA** - The coarse grained reconfigurable architecture enables pipelined computation of dataflow graphs. The spatial nature of the CGRA avoids the overheads of accessing register files or memories for live values.

Stream Command Lifetime: The lifetime of a stream command is as follows. First, the control core generates the command and sends it to the stream dispatcher. The stream dispatcher issues the command to the appropriate stream engines once any associated resources (vector ports and stream-engine table entries) are free. The data transfer for each stream is carried out by the stream engine, which keeps track of the running state of the stream over its lifetime. When the stream completes, the stream engine notifies the dispatcher that the corresponding resources are free, enabling the next stream-command to be issued.

4.2 Stream Dispatch and Control Core

The role of the stream dispatcher is to enforce resource dependences on streams (and barrier commands), and coordinate the execution of the stream-engines by sending them commands. Figure 9 shows the design and internal resource management mechanisms of the stream dispatcher. Stream requests from the control core are queued until they can be processed by the command decoder. This unit consults with resource status checking logic to determine if a command can be issued, if so it will be dequeued. Barrier commands block the core from issuing further stream commands until the barrier condition is resolved. We explain in more detail below.

Resource Management: Subsequent streams that have the same source or destination port must be issued in program order, ie. the dynamic order of the streams on the control core. The stream dispatch unit is responsible for maintaining this order, and does so by tracking vector port and stream engine status in a scoreboard. Before issuing a stream, it checks the state of these scoreboards.

The state of a vector port is either *taken*, *free*, or *all-requests-in-flight*. A port moves from *free* to *taken* when issued (by the resource assigner), and that stream logically owns that resource while in flight.

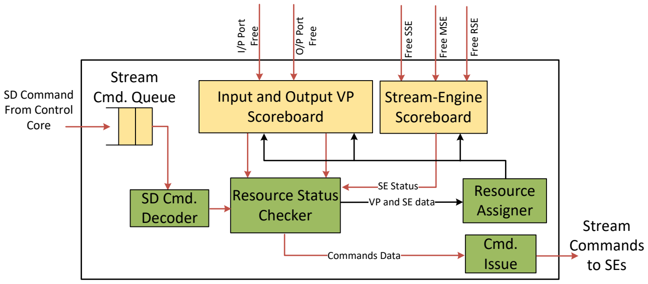


Figure 9: Stream Dispatcher Microarchitecture

When the stream is finished, the associated stream engine notifies the dispatcher to update the scoreboard entry of the vector port to the *free* state. The *all-requests-in-flight* state indicates all requests for a memory stream are completely sent to the memory system (but have not arrived). This state exists as an optimization to enable two memory streams using the same port to have their requests in the memory system at the same time.

Barriers: The stream dispatcher must also coordinate the ordering of streams given barrier instructions. Our simple microarchitecture only implements scratchpad barriers, and simply blocks commands beyond the barrier until the condition is met (eg. no outstanding scratchpad writes). Other active streams can continue to perform useful work while the stream dispatcher waits, which is how forward progress can be guaranteed for a correct program.

Interface to the Control Core: The interface between the dispatcher and the control core is a command bus, and commands are queued until issued. The dispatcher can stall the core, which would happen if the command queue is full, or the `SD_Barrier_All` command is in the command queue.

4.3 Stream Engines

Stream engines manage concurrent access to various resources (memory interface, scratchpad, output vector port) by many active streams. They are critical to achieving high parallelism with low power overhead, by fully utilizing the associated resources through arbitrating stream access.

Stream engines are initiated by receiving commands from the stream dispatcher. They then coordinate the address generation and data transfer over the lifetime of the stream, and finally notify the dispatcher when the corresponding vector ports are freed (when the stream completes). The stream engines each have their own 512-bit wide bus to the input and output vector ports. The stream dispatcher ensures that concurrent streams have dedicated access to their vector ports. Indirect access is facilitated by vector ports which are not connected to the CGRA, which buffer addresses in flight. Below we describe the central control unit in each stream engine, the stream request pipeline, followed by the specific design aspects of each.

Stream Request Pipeline: Similar hardware is required in each stream engine to arbitrate access to the underlying resource, and we call this a stream request pipeline. On each cycle, this unit selects one of the multiple active streams and facilitates its address generation and data transfer. Figure 8 shows the template for a stream request pipeline, which is tailored slightly for each type of stream engine. When a command is issued to a stream engine, it is first decoded and any relevant state is placed in the stream table. This maintains the set of active streams, where each row contains the associated data for a stream. Ready logic observes the validity of streams, and will connect to external backpressure signals (eg. remaining space in input vector port) for determining which streams are ready to issue. A stream is ready if its destination is not full and its source has data. A selector prioritizes a ready stream.

The state of the selected stream will be sent to an appropriate address generation unit (AGU), either affine or indirect, which computes the next 64-byte aligned address. These units also generate a mask to indicate words relevant to the stream. The affine AGU generates the minimal number of memory requests by examining the access size, stride size and number-of-strides parameters. The indirect AGU takes address values from an indirect port. This unit will attempt to coalesce up to four increasing addresses in the current 64-byte line.

Memory Stream Engine: This unit delivers data from or to the memory system, which in case of Softbrain is a wide-interface cache. The read and write engines have their own independent stream request pipelines. The memory *read* engine has buffering for outstanding requests, and uses a balance arbitration unit for stream priority selection (described in Section 4.5). The backpressure signal for memory reads to the CGRA is the number of entries free in the buffers associated with the vector ports. For handling backpressure on scratchpad writes, a buffer sits between the MSE and SSE. This buffer is allocated on a request to memory to ensure space exists. The memory *write* engine uses the data available signals from vector ports for priority selection.

Scratchpad Stream Engine: This unit is similar to the above, except that it indexes a scratchpad memory. A single-read, single-write ported SRAM is sufficient, and its width is sized proportional to the maximum data consumption rate of the CGRA. Similar to the memory stream engine, the backpressure signal is the number of free buffer entries on the vector ports. If there are no entries available (ie. there is backpressure), then the corresponding stream will not be selected for loading data.

Reduction/Recurrence Stream Engine: A reduction/recurrence stream engine delivers data from the output to input vector ports for efficiently communicating dependences. It also is used for delivering "constants" from the core. It does not require the AGU, but does use a similar backpressure mechanism as the above units.

4.4 Computation and Dataflow Firing

Vector Port Interface: Vector ports are the interface between the CGRA and stream engines, and are essentially 512-bit wide FIFOs that hold values waiting to be consumed by the CGRA. Each vector port can accept or send a variable number of words per cycle, up to 8 64-bit words. On the CGRA side, vector ports attach to a heterogeneous set of CGRA ports, which are selected to spread incoming/outgoing values around the CGRA to minimize contention. This mapping is fed to the DFG scheduler to map ports of the program DFG to hardware vector ports. Dataflow firing occurs in a coarse-grained fashion, when one instance worth of data is available on all relevant vector ports, all of the relevant data is simultaneously released into the CGRA.

CGRA: Our CGRA is a deeply pipelined execution substrate similar to prior designs [7, 29]. It has a circuit-switched mesh of processing elements, where each contains a set of pipelined functional units. Predication support at each FU allows mapping of local data-dependent control flow, and FUs store re-used constants and accumulators. It differs from the referenced designs in that there is no flow-control inside the mesh (we found this reduced the network area by about half). This is enabled by the synchronized dataflow firing of input vectors. The lack of flow-control also requires the DFG compiler to ensure delay-matching along all computation paths, including to the output vectors ports. The CGRA’s datapath is 64-bit in our implementation, and functional units can perform multiple sub-word operations including 32-bit and 16-bit.

The configuration of the CGRA (datapath and constants) and vector ports is initialized by the `SD_Config` command, which is processed by the memory stream engine. Configuration can be completed in less than 10 cycles if the data is cached.

4.5 Cross-cutting design issues

Buffering and Deadlocks: The Softbrain unit must avoid deadlock by balancing the stream requests to different vector ports. This can be local to a stream engine, as each stream engine owns a single resource. Deadlock can occur, for example, when many long-latency operations for a single port fill the request pipeline to memory, but data is needed on another port for the CGRA computation to achieve forward progress. This can happen if one of the streams is strided, causing a lower relative memory bandwidth. We solve this issue by adding a balance unit to the memory load stream engine. It tracks the amount-of-unbalance for each active vector port, and heavily unbalanced ports are de-prioritized.

Memory Coherence: Because Softbrain’s memory interface directly accesses the L2 cache, there is the possibility of incoherence between the control core’s L1 and the L2. To avoid incoherent reads from the L2 to the stream engines, the control core’s L1 is write-through. To avoid incoherent reads on the L1, Softbrain sends L1 tag invalidations as it processes the stream.

Role of the Compiler and Programmer: In this work, we express programs directly in terms of intrinsics for the stream-dataflow commands (see Figure 6 on page 6). So the primary Compiler task is to generate an appropriate CGRA configuration for the DFG and vector port mapping. The DFGs are specified in a simple graph language, and we extend an integer linear optimization based scheduling approach from prior work [22].

Though programming is low-level, the primitives are more flexible than their SIMD counterparts. Compiling to a stream-dataflow ISA from a higher level language (OpenCL/OpenMP/OpenAcc) seems practical and quite useful, especially to scale the design to more complex workloads. This is future work.

Integration: Softbrain can be integrated into a broader system in a number of ways, including as a unit in an SoC, through unified virtual memory, or as a standalone chip. In this work we assume a standalone device for evaluation purposes. It is possible to support integration to a unified virtual memory with coherent caches. Address translation could be supported at the L2 level (making L1 and L2 virtual) if dedicated accelerator access is assumed, or by integrating TLBs in the memory stream engine.

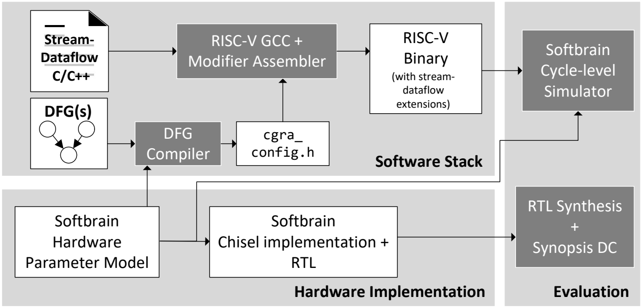

Stream-Dataflow C/C++
RISC-V GCC + Modifier Assembler
RISC-V Binary (with stream-dataflow extensions)
Softbrain Cycle-level Simulator
DFG(s)
DFG Compiler
cgra_ config.h
Software Stack
Softbrain Hardware Parameter Model
Softbrain Chisel implementation + RTL
RTL Synthesis + Synopsis DC
Hardware Implementation
Evaluation

Figure 10: Software Stack

There are other design aspects critical to certain settings and systems, like support for precise exceptions, backwards compatibility, security and virtualization. These are deferred for future work.

5 IMPLEMENTATION

Here we discuss the implementation in terms of the hardware, software stack, and simulator used in the evaluation. We also discuss how this setup can be used in a practical hardware/software workflow. Figure 10 shows an overview.

Hardware: We implemented the design from Section 4 in Chisel [1]. The design is parameterizable (eg. CGRA size, FU types, port widths, scratchpad size, etc), and uses an architecture description file which is shared with the software stack and simulator.

Software Stack: For programming, we create a simple wrapper API that is mapped down into the RISCV-encoding of the stream-dataflow ISA. We modified a GCC cross compiler for RISCV with stream-dataflow ISA extensions, and implemented our own DFG compiler. The DFG to CGRA mapper extends prior work [22].

Simulator: We implement a cycle-level RISC-V based simulator for the control core and Softbrain. The Softbrain simulator is a simple module, which takes stream-dataflow commands and integrates with the core’s cache interface for load/store requests.

Hardware/Software Workflow: In practice, the hardware would be provisioned once per chip family. For instance, if it is known ahead of time that all data types for a particular market were a maximum of 16-bit (eg. for machine learning), this could be incorporated into the functional unit composition of the CGRA. Here, an architect either uses existing knowledge or profiles applications from the domain(s) in question. Then they would adjust *only* the FU mix and scratchpad size, recording this in the hardware parameter model file. Even in this case, no Chisel or hardware interfaces need modification.

For each application, the developer constructs DFGs of the computation component, and writes the stream coordination program (stream commands embedded into a C program).

6 EXPERIMENT METHODOLOGY

Workloads: To compare against domain specific accelerators, we use the deep neural network (DNN) workloads from the DianNao accelerator work [3], including classifier, convolutional and pooling layers. They have high data-parallelism and memory regularity, and vary in their re-use behavior (and thus memory-bandwidth).

To capture a broader understanding of the efficiency and generality tradeoffs, we consider MachSuite [26], a set of typically-accelerated workloads. Unlike the DNN workloads, these capture wider program behaviors like regular and irregular memory access patterns, data-dependent control, and varying computation intensity. We compare these designs against application specific versions.

Power & Area: For the power and area of our Softbrain implementation, we synthesize the Chisel-generated verilog with Synopsis DC and the ARM 55nm technology library, which meets timing at 1GHz. We use Cacti [19] for SRAM and caches estimates.

Comparison Methodology: For the DNN workloads we compare against the DianNao accelerator using a simple performance model. This model optimistically assumes perfect hardware pipelining and scratchpad reuse; it is only bound by parallelism in the neural network topology and by memory bandwidth. We take the power/area numbers from the relevant publication [3]. For comparison points, we consider single-threaded CPU implementations, running on a i7 2600K Sandy Bridge machine. We also compare against GPGPU implementations of these workloads written in CUDA, running on a Kepler-based GTX 750. It has 4 SMs and 512 total CUDA cores.

For comparing to MachSuite accelerators, we use Aladdin [28], a pre-RTL accelerator modeling tool. Aladdin determines the fixed-function accelerator performance, power, and area given a set of prescribed hardware transformations (eg. loop unrolling, loop flattening, memory array partitioning and software pipelining, which impact the datapath and scratchpad/caches sizes).

7 EVALUATION

In this section, we address five important questions for evaluating Softbrain, and we list them here with brief answers for each:

(1) What are the sources of its power and area overhead?
• CGRA network and control core.

(2) Can it match the speedup of a domain specialized accel.?
• Yes

(3) Is the stream-dataflow paradigm general?
• Yes, All DNN and most MachSuite are implementable using the stream-dataflow abstractions.

(4) What are its limitations in terms of generality?
• Code properties that are not suitable are arbitrary memory-indirection and aliasing, control-dependent loads, and bit-level manipulations.

(5) How does stream-dataflow compare to application-specific?
• Only 2× power and 8× area overhead.

7.1 Domain-Specific Accelerator Comparison

Here we explore the power and area of Softbrain compared to a domain-specific accelerator for deep neural networks, DianNao. Our approach is to compare designs with equivalent performance, and examine the area and power overheads of Softbrain.

Area and Power Comparison: To make an intuitive comparison, we configure the Softbrain's functional units to meet the needs of the DNN workloads. Here, we need four-way 16-bit subword-SIMD multipliers and ALUs, and a 16-bit sigmoid. We also include 8 total tiles (Softbrain units), which enables Softbrain to reach the same number of functional units as DianNao.

		area(mm^2)	power (mw)
Control Core + 16kB I & D$		0.16	39.1
CGRA	Network	0.12	31.2
	FUs (4×5)	0.04	24.4
	Total CGRA	0.16	55.6
5×Stream Engines		0.02	18.3
Scratchpad (4KB)		0.1	2.6
Vector Ports (Input & Output)		0.03	3.6
1 Softbrain Total		**0.47**	119.3
8 Softbrain Units		3.76	954.4
DianNao		2.16	418.3
Softbrain / DianNao Overhead		1.74	2.28

Table 3: Area and Power Breakdown / Comparison (All numbers normalized to 55nm process technology)

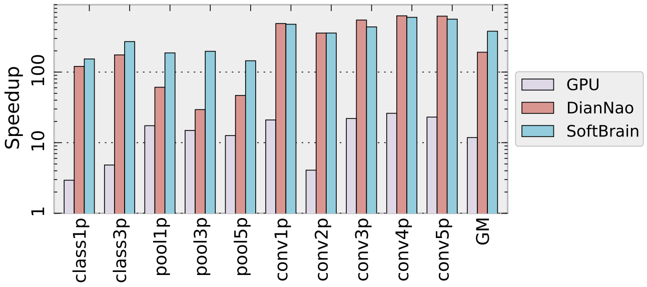


Figure 11: Performance on DNN Workloads.

Table 3 shows the breakdowns of area and power. All the analysis is normalized to 55nm process technology. For the power calculations here, we use the maximum activity factors across the DNN workloads. The majority of the area comes from the CGRA network, consuming about one-fourth of the total area and power. The other large factor is the control core, which consumes a third of the power and the area. Compared to the DianNao accelerator, Softbrain is only about 1.75× more power and a little over twice as large.

Performance: Figure 11 shows the speedups of the Kepler GPU, DianNao and Softbrain for the three classes of DNN workloads. Overall, the GPU is able to obtain up to 20× performance improvement, while DianNao and Softbrain achieve similar performance, around 100× or more on some workloads. The reason is intuitive: both architectures use the same basic algorithm, and they are able to keep 100s of FUs active in every cycle by decoupling memory access from deep pipelined computation. Softbrain does see some advantage in pooling workloads, as its more flexible network allows it to reuse many of the subsequent partial sums in neighboring computations, rather than re-fetching them from memory. This allows it to reduce bandwidth and improve speedup.

7.2 Stream-Dataflow Generality

Here we attempt to distill the limitations of the stream-dataflow processor in terms of its generality. To this end, we study a broader set

Implemented Codes	Stream Patterns	Datapath
`bfs`	Indirect Loads/Stores, Recurrence	Compare/Increment
`gemm`	Affine, Recurrence	8-Way Multiply-Accumulate
`md-knn`	Indirect Loads, Recurrence	Large Irregular Datapath
`spmv-crs`	Indirect, Linear	Single Multiply-Accumulate
`spmv-ellpack`	Indirect, Linear, Recurrence	4-Way Multiply-Accumulate
`stencil2d`	Affine, Recurrence	8-Way Multiply-Accumulate
`stencil3d`	Affine	6-1 Reduce and Multiplier Tree
`viterbi`	Recurrence, Linear	4-Way Add-Minimize Tree
Unsuitable Codes	**Reason**	
`aes`	Byte-level data manipulation	
`kmp`	Multi-level indirect pointer access	
`merge-sort`	Fine-grain data-dependent loads/control	
`radix-sort`	Concurrent reads/writes to same address	

Table 4: Workload Characterization

of typically-accelerated workloads from MachSuite, and provision a single design for them. We first characterize our implementations of these workloads and the limitations we discovered.

Softbrain Provisioning: To provision Softbrain's FU resources, we implemented stream-dataflow versions of the MachSuite workloads targeting a maximum of 20 DFG instructions (the same size we used for the DianNao comparison). We then provisioned Softbrain's FU mix to the maximum needed across workloads. Note that for consistency we used 64-bit integer/fixed-point datatypes[2]. Hereafter, we refer to this as the broadly provisioned Softbrain.

Softbrain Generality: Table 4 summarizes the architectural abstractions used in the stream-dataflow program implementations. It describes the streaming patterns and datapath structure[3].

We found that each architectural feature was useful across several workloads. Affine accesses were used in `gemm` and `stencil` codes to reduce access penalties. Indirect loads or stores were required in four workloads (`bfs` and `knn`, and `spmv` versions). Recurrence patterns were useful across most workloads, mainly for reduction variables. The size and configuration of the datapath varies greatly, from reduction trees, SIMD-style datapaths, and more irregular datapaths, suggesting that the flexible CGRA was useful.

There were four workloads that we found could not be efficiently implemented in stream-dataflow. The `aes` encryption workload required too much byte-level manipulation (access words less than 16-bit) that made it difficult to justify offloading onto a coarse-grained fabric. The `kmp` string matching code requires arbitrary-way indirect loads, and the architecture can only support a finite amount of indirection in an efficient way. The `merge-sort` code contains fine-grain data-dependent loads and control instructions, used to decide which list to read from next. The `radix-sort` workload has several phases where reads or writes during that phase could be to the same address (and we don't provide hardware support for implicit store-load forwarding).

Overall, Softbrain is quite general and applicable across many data-processing tasks, even some with a significant degree of irregularity. Its limitations are potentially addressable in future work.

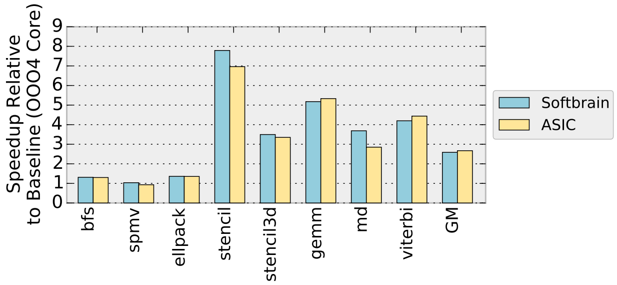


Figure 12: Softbrain Performance Comparison to ASIC

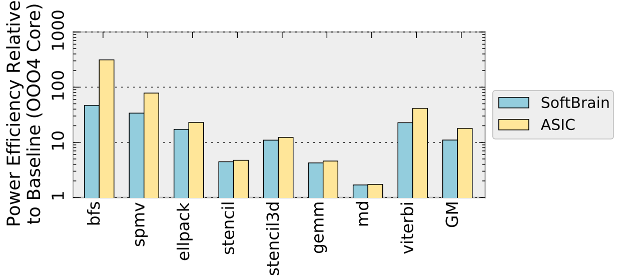


Figure 13: Softbrain Power Comparison to ASIC

7.3 Application-Specific Comparison

In this section we compare the broadly provisioned Softbrain to customized ASICs generated for each application, in terms of their power, performance, energy and area.

ASIC Design Point Selection: For comparing the broadly provisioned Softbrain with a workload-specific ASIC, we chose to do an iso-performance analysis, while secondarily minimizing ASIC area and power. To explain in detail - for each workload we explore a large ASIC design space by modifying hardware optimization parameters, and find the set of ASIC designs within a certain performance threshold of Softbrain (within 10% where possible). Within these points, we chose a Pareto-optimal ASIC design across power, area, and execution time, where power is given priority over area.

Performance: For performance evaluation of Softbrain to ASIC, the execution cycles obtained from our Softbrain RISC-V based simulator is compared to the execution cycles of the benchmark-specific custom accelerator generated from Aladdin. Figure 12 shows the performance of Softbrain compared to benchmark specific Pareto optimal ASICs. For both ASIC and Softbrain, the execution cycles are normalized to a SandyBridge OOO4(4-wide) core, with both performing achieving 1-7x speedup. In most cases we found an ASIC design with similar performance to Softbrain[4].

Power, Area & Energy vs. ASIC Designs: As explained above, we choose the iso-performance design points for power, energy and area comparison of Softbrain to ASIC.

For power analysis, we consider that only benchmark-specific FUs are active during execution in Softbrain's CGRA along with other support structures, including the control core, stream engines, scratchpad and vector ports. The static power and area for Softbrain are obtained from the synthesized design, and the dynamic power estimates reported are based on the activity factor of each module.

[2] Using floating point would have decreased the relative overheads of Softbrain versus an ASIC, but also decreased the area/power benefits of acceleration slightly.

[3] There were 4 additional workloads which we have not yet implemented, but do fit into the stream-dataflow paradigm: `fft`, `md`(gridding version), `nw` and `backprop`.

[4] Note that for some workloads (eg. stencil, md) there is an ASIC deign point with better performance than Softbrain, but falls outside the performance threshold.

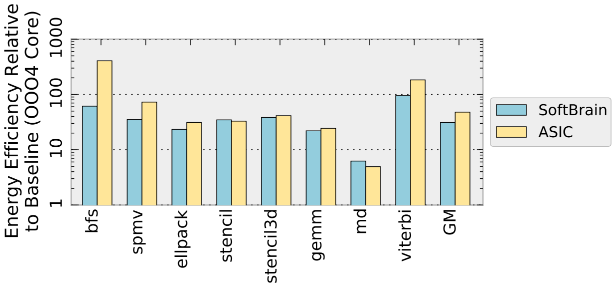


Figure 14: Softbrain Energy Comparison to ASIC

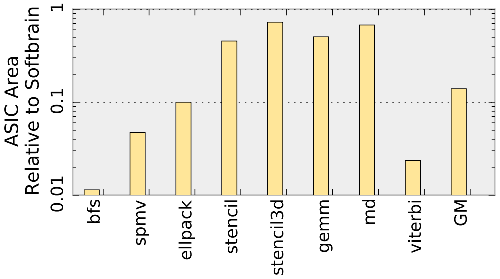


Figure 15: Softbrain Area Comparison to ASIC

Energy estimates are straightforward to get from execution cycles and dynamic power estimates. ASIC area and power are obtained from Aladdin, using 40nm technology, and are normalized to 55nm.

Figure 13 shows the power savings (efficiency) over a Sandybridge OOO4 core[5] as the baseline. Both ASIC and Softbrain have a large power savings of up to 300x compared to the OOO4 core, which is expected because of the power and area which the OOO4 core spends on supporting generality. ASICs have better power efficiency than Softbrain overall, but only by 2× across all benchmarks. With some workloads, ASIC has almost the same power as Softbrain, and this is due to the fact that Aladdin instantiates larger memory structures (scratchpads, buffers etc.) for loop-unrolling, essentially flattening the array data-structures in order for the design space to have performance points close to Softbrain. Note that we include the local memory structures of the ASICs in their power estimation as Softbrain also has a programmable scratchpad. Most of the additional power consumption in Softbrain is because of the generality supporting structures, such as the CGRA network, which is capable of mapping a wide variety of possible DFGs.

Figure 14 shows the energy efficiency comparison of Softbrain and the ASICs, showing a high efficiency advantage for both compared to the baseline OOO4 core. The energy consumption of Softbrain is within 2x of ASIC, and this is mainly due to the difference in power consumption.

Figure 15 shows the area comparison. As Softbrain's area is fixed across benchmarks, the results show ASIC area relative to Softbrain's area. We do not include the ASIC designs' memory structures area in their estimates, as most of the time these workloads have streaming behavior and ASICs can achieve the same performance with more parallel FUs, rather than larger storage structures[6]. The mean Softbrain area is 8× that of the ASIC, which is expected as Softbrain is programmable and must run all workloads. From another perspective, Softbrain is area efficient, as including all eight MachSuite accelerators would have required 2.54× as much area as only including Softbrain.

Overall, Softbrain is competitive with ASICs in terms of performance, power, energy and area, even with the hardware necessary to support significant programmability. This demonstrates that there is scope to develop programmable architectures by tapping the right synergy between the algorithm properties of typically accelerated workloads and the microarchitectural mechanisms supporting stream-dataflow execution.

8 RELATED WORK

Streaming in Data Parallel Architectures: The concept of exposing streams in a core's ISA to communicate to reconfigurable hardware was proposed in the Reconfigurable Streaming Vector Processor (RSVP) [5]. RSVP uses similar descriptions of affine streams and dataflow graphs, but have several fundamental limitations. RSVP's communication patterns for any given computation cannot change during an execution phase. This reduces the flexibility of the types of patterns that can be expressed (eg. outputs that are used for reduction and sometimes written to memory). It also hampers the ability to prefetch data across different phases – one phase must complete before the data for another phase can be fetched, a high overhead for short phases. Next, the inter-iteration dependence distance in RSVP can only be 1, which limits programmability. Finally, the address space of RSVP's "scratchpad memory" is not exposed to the programmer, which only map linear portions of the address space. This disallows some optimizations where sparse data representations are compacted into the scratchpad and read multiple times.

An early foundational work in this area is Imagine [13], which is a scalable data parallel architecture for media processing. Imagine uses concepts of streams for explicit communication between memory and a so-called stream register file which acts as a scratchpad for communicating between memory and execution units, as well as between subsequent kernels. Streams are restricted to being linear and have relatively small maximum size. Streams are also not exposed in the lower level interface for controlling the execution resources: a cluster of VLIW pipelines are activated in SIMD fashion by a microcontroller. A stream based ISA in this context could reduce the complexity of the controlling VLIW core. From a high-level, Imagine can be viewed as a set of stream-dataflow processors which read all memory through the scratchpad, and where the reconfigurable fabric is replaced by more rigid SIMD+VLIW execution units.

Triggered instructions [23] is a spatial architecture featuring some streaming memory capability to feed its dataflow fabric. The computing fabric is more flexible in terms of control flow and total instruction capacity, but is likely higher power and area.

The problem of efficiently interfacing with reconfigurable hardware also occurs in an FPGA computation offloading environment. CoRAM++ [36] enables data-structure specific API interfaces for transferring data to FPGA-synthesized datapaths, which is implemented with specialized soft-logic for each supported data-structure. This interface is primarily based on streams.

Removing Data-Parallel Architecture Inefficiencies: Several works attempt to specialize existing data parallel architectures.

[5] We consider the dynamic power of 1 core at 32nm and scale it to 55nm.

[6] Including the memory structure area for the ASIC estimates would make Softbrain look better in comparison.

One example for SIMT is exploiting value structure to eliminate redundant affine address and value computations [14]. The SGMF architecture adds dataflow extensions to GPGPUs [34]. Another work explores a set of mechanisms for data parallel execution, and evaluates these for the TRIPS spatial architecture [27]. Many of the data-parallel program attributes they define are targeted here, but we use a different set of mechanisms. We previously discussed the relationship to vector-thread architectures like Maven VT [16].

Softbrain can be viewed as an instance of the prototypical accelerator we defined in prior work [21], called LSSD. It leverages a multi-issue inorder SIMD core to bring data to the reconfigurable array (rather than stream engines), which requires higher energy due to the core's issue width, and is more difficult to program or compile for because of the required modulo scheduling.

Heterogeneous Cores: There is a large body of work on combining general purpose cores and reconfigurable or otherwise specialized engines. A number of these designs target irregular workloads, like Composite Cores [18] for code phases with low ILP, or XLOOPS [31] for loops with inter-iteration dependence patterns, or SEED [20] for phases with non-critical control. We consider those designs to be mostly orthogonal to the stream-dataflow ISA.

Programmable accelerators targeting data parallelism (eg. DySER [7] or Libra [24]) could benefit from the architectural interfaces we propose, and it would be interesting if compilers could automatically target such architectures, perhaps by extending existing SIMD-style vectorization approaches [8].

A highly related work is that of Memory Access Dataflow [10], which is another access-execute style architecture. It consists of a general core, compute accelerator, and reconfigurable address-generation fabric. We considered a similar approach for address generation, but found that for the access patterns we needed to support, the area overheads would have exceeded multiple factors.

A recurring issue for resource-exposed architectures is binary compatibility. VEAL uses dynamic compilation to translate loops in the baseline ISA into a template loop accelerator which has simple address generators and a modulo-scheduled programmable engine [6]. Similar techniques have been proposed to dynamically compile for CGRAs [35], and are applicable to stream-dataflow.

Streaming in Domain Specific Accelerators: Many domain-specific accelerators use streaming and dataflow abstractions. Eyeriss is a domain-specific accelerator for convolutional neural networks, using streaming access to bring in data, as well as a dataflow substrate for computation [4]. A recent work, Cambricon [17], proposes SIMD instruction set extensions which can perform the stream-like access patterns found in DNNs. Outside the domain of machine learning, Q100 [37] is an accelerator for performing streaming database queries. It uses a stream-based abstraction for accessing database columns, and a dataflow abstraction for performing computations.

9 DISCUSSION AND CONCLUSIONS

This paper has proposed a new execution model and architecture, stream-dataflow, which provides abstractions that balance the tradeoffs of vector and spatial architectures, and attain the specialization capabilities of both on an important class of data-processing workloads. We show that these primitives are sufficiently general to express the execution of a variety of deep-learning workloads and many workloads from MachSuite (a proxy for workloads that an ASIC could be considered for). In terms of our hardware implementation, we have developed an efficient microarchitecture, and our evaluation suggests its power and area is only small factors more than domain-specific and ASIC designs.

We envision that this architectural paradigm can have a radical simplifying effect on future chips by reducing the number of specialization blocks. Instead, a stream-dataflow type fabric can sit alongside CPU and GPU processors, with functionality synthesized on the fly as programs encounter suitable phases for efficient offloading. This not only reduces the area and complexity of having vast arrays of specialized accelerators, it also can mitigate growing design and verification costs. In such a broad setting, it will be critical to develop effective compilation tools that can balance the complex tradeoffs between parallelism and data reuse that these architectures provide. Providing dynamic compilation support for a stream-dataflow architecture could bring specialized-architecture efficiency to the masses.

Overall, we believe that stream-dataflow has a large role to play going forward. Just as RISC ISAs defined and drove the era of pipelining and the multi-decade dominance of general purpose processors, for the era of specialization we need new architectures and ISA abstractions that match the nature of emerging workloads. Stream-dataflow can serve as this generation's ISA to drive further microarchitecture and architecture-level innovations for accelerators.

10 ACKNOWLEDGMENTS

We would first like to thank the anonymous reviewers for their detailed questions and suggestions which helped us to clarify the presentation. We want to thank Preyas Shah for setting up the automated Synopsys toolchain for area-power analysis. We would also like to thank Sophia Shao and Michael Pellauer for their thoughtful suggestions and advice during the revision process. This work was supported by the National Science Foundation, grant CCF-1618234.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 1216–1225. DOI:https://doi.org/10.1145/2228360.2228584

[2] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. 2004. Scaling to the End of Silicon with EDGE Architectures. *Computer* 37, 7 (July 2004), 44–55. DOI:https://doi.org/10.1109/MC.2004.65

[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 269–284. DOI:https://doi.org/10.1145/2541940.2541967

[4] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 367–379. DOI:https://doi.org/10.1109/ISCA.2016.40

[5] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The Reconfigurable Streaming Vector Processor (RSVP). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 141–. http://dl.acm.org/citation.cfm?id=956417.956540

[6] Nathan Clark, Amir Hormati, and Scott Mahlke. 2008. VEAL: Virtualized Execution Accelerator for Loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 389–400. DOI:https://doi.org/10.1109/ISCA.2008.33

[7] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (Sept. 2012), 38–51. DOI:https://doi.org/10.1109/MM.2012.51

[8] Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. 2013. Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 341–352. DOI:https://doi.org/10.1109/PACT.2013.6618830

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

[10] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient Execution of Memory Access Phases Using Dataflow Specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 118–130. DOI:https://doi.org/10.1145/2749469.2750390

[11] Mircea Horea Ionica and David Gregg. 2015. The Movidius Myriad Architecture's Potential for Scientific Computing. *IEEE Micro* 35, 1 (2015), 6–14. DOI:https://doi.org/10.1109/MM.2015.4

[12] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and others. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA '17*.

[13] Brucek Khailany, William J Dally, Ujval J Kapasi, Peter Mattson, Jinyung Namkoong, John D Owens, Brian Towles, Andrew Chang, and Scott Rixner. 2001. Imagine: Media processing with streams. *IEEE micro* 21, 2 (2001), 35–46. DOI:https://doi.org/10.1109/40.918001

[14] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. 2013. Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 130–141. DOI:https://doi.org/10.1145/2485922.2485934

[15] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. 2004. The Vector-Thread Architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*. IEEE Computer Society, Washington, DC, USA, 52–. DOI:https://doi.org/10.1145/1028176.1006736

[16] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 129–140. DOI:https://doi.org/10.1145/2000064.2000080

[17] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 393–405. DOI:https://doi.org/10.1109/ISCA.2016.42

[18] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. 2012. Composite Cores: Pushing Heterogeneity Into a Core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 317–328. DOI:https://doi.org/10.1109/MICRO.2012.37

[19] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories* (2009), 22–31.

[20] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 298–310. DOI:https://doi.org/10.1145/2749469.2750380

[21] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. 2016. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 27–39. DOI:https://doi.org/10.1109/HPCA.2016.7446051

[22] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. 2013. 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)*. ACM, New York, NY, USA, 495–506. DOI:https://doi.org/10.1145/2491956.2462163

[23] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 142–153. DOI:https://doi.org/10.1145/2485922.2485935

[24] Yongjun Park, Jason Jong Kyu Park, Hyunchul Park, and Scott Mahlke. 2012. Libra: Tailoring SIMD Execution Using Heterogeneous Hardware and Dynamic Configurability. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 84–95. DOI:https://doi.org/10.1109/MICRO.2012.17

[25] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. DOI:https://doi.org/10.1109/MM.2015.42

[26] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 110–119.

[27] Karthikeyan Sankaralingam, Stephen W. Keckler, William R. Mark, and Doug Burger. 2003. Universal Mechanisms for Data-Parallel Architectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 303–. DOI:https://doi.org/10.1109/MICRO.2003.1253204

[28] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 97–108. DOI:https://doi.org/10.1109/MM.2015.50

[29] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. 2000. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. Comput.* 49, 5 (May 2000), 465–481. DOI:https://doi.org/10.1109/12.859540

[30] James E. Smith. 1982. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture (ISCA '82)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 112–119. DOI:https://doi.org/10.1145/357401.357403

[31] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. 2014. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 583–595. DOI:https://doi.org/10.1109/MICRO.2014.31

[32] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 291–. DOI:https://doi.org/10.1109/MICRO.2003.1253203

[33] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (March 2002), 25–35. DOI:https://doi.org/10.1109/MM.2002.997877

[34] Dani Voitsechov and Yoav Etsion. 2014. Single-graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 205–216. DOI:https://doi.org/10.1109/ISCA.2014.6853234

[35] Matthew A Watkins, Tony Nowatzki, and Anthony Carno. 2016. Software Transparent Dynamic Binary Translation for Coarse-Grain Reconfigurable Architectures. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 138–150. DOI:https://doi.org/10.1109/HPCA.2016.7446060

[36] Gabriel Weisz and James C Hoe. 2015. CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing. In *25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. DOI:https://doi.org/10.1109/FPL.2015.7294017

[37] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. 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)*. ACM, New York, NY, USA, 255–268. DOI:https://doi.org/10.1145/2541940.2541961