Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms

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

ABSTRACT

With slowing technology scaling, specialized accelerators are increasingly attractive solutions to continue expected generational scaling of performance. However, in order to accelerate more advanced algorithms or those from challenging domains, supporting *data-dependence* becomes necessary. This manifests as either data-dependent control (eg. join two sparse lists), or data-dependent memory accesses (eg. hash-table access). These forms of data-dependence inherently couple compute with memory, and also preclude efficient vectorization – defeating the traditional mechanisms of programmable accelerators (eg. GPUs).

Our goal is to develop an accelerator which is broadly applicable across algorithms with and without data-dependence. To this end, we first identify forms of data-dependence which are both common and possible to exploit with specialized hardware: specifically stream-join and alias-free indirection. Then, we create an accelerator with an interface to support these, called the Sparse Processing Unit (SPU). SPU supports alias-free indirection with a compute-enabled scratchpad and aggressive stream reordering and stream-join with a novel dataflow control model for a reconfigurable systolic compute-fabric. Finally, we add robustness across datatypes by adding decomposability across the compute and memory pipelines. SPU achieves 16.5×, 10.3×, and 14.2× over a 24-core SKL CPU on ML, database, and graph algorithms respectively. SPU achieves similar performance to domain-specific accelerators. For ML, SPU achieves 1.8-7× speedup against a similarly provisioned GPGPU, with much less area and power.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; **Data flow architectures**; *Heterogeneous (hybrid) systems.*

KEYWORDS

Irregularity, data-dependence, accelerators, generality, dataflow, systolic, reconfigurable, join, indirection

ACM Reference Format:
Vidushi Dadu, Jian Weng, Sihao Liu, Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3352460.3358276

	Kernel	**Stream Join** (Irreg. Control)	**Indirect Memory** (Irreg. Memory)	**Non-data-dep.** (Regular)
Machine Learning (ML)	Conv	N/A	Outer-prod. [72] (sparsity)	Dense Conv [29]
	FC/ KSVM	Inner-prod. [59] (Sparsity+better at skewed dist.)	Outer-prod. [40] (Sparsity+better for large datasets)	Dense MV
	GBDT	Sort-based [20] (Sparsity+accuracy on weighted data)	Histo [49] (Sparsity+accuracy on unweighted)	Not Possible
	Arith. Circits.	N/A	DAG Travers. (dag sparsity)	Chain of MM
Database	Join	Sort [106] O(Nlog(N))	Hash join [50] O(N)	Cartesian O(N^2)
	Sort	Merge O(Nlog(N))	Radix O(N)	Not Possible
	Filter	Gen filtered Col [106] (sparsity)	Gen. Column Indices (sparsity)	Maintain Bitvector
Graph	Page Rank	Pull [3] (O(VE) + no rd/wr dependency)	Push [39] (O(VE) + No latency stalls)	Dense MM (O(V^2))
	BFS	Pull [5] (O(VE) + better middle iters)	Push [39] (O(VE) + better beg/end iters)	Dense MM (O(V^2))

Table 1: Data-Dependence Forms Across Algorithms

1 INTRODUCTION

Trends in technology scaling and application needs are causing a broad push towards specialized accelerators. Examples pervade many domains, including graphs [9, 28, 39, 91, 108], AI/ML [10, 43, 47, 80, 85, 98, 107], databases [48, 50, 105, 106], systems [31–33, 110], and genomics [23, 36, 95, 96]). This trend is also true in industry [46, 62, 69, 79, 100].

Designs which are performance-robust across domains would be valuable for economies of scale. Furthermore, with a perishing Moore's Law, the approach of spending transistors on ever more non-programmable ASICs will become less effective [35]. However, the success of the above domain-specific accelerators suggests that existing general purpose data processing hardware (eg. GPGPUs [69], Intel MIC [30] & KNL [90]) are orders-of-magnitude lower in performance and/or energy efficiency. But why? Our insight is the following: *data-dependence*, in the form of data-dependent control and data-dependent memory access, fundamentally interferes with common mechanisms relied on by such processors.

To explain, consider the following two basic hardware principles: *decoupling* the memory access and computation pipelines, and *vectorizing* the computation across independent hardware units. GPUs are a classic example that exploit vectorization through the SIMT execution model, and perform decoupling by relying on many threads. Data-dependent control introduces thread divergence (bad for vectorization), and data dependent memory introduces non-contiguous loads (bad for vectorization) and further requires more threads to hide the likely higher latency (bad for decoupling). Other

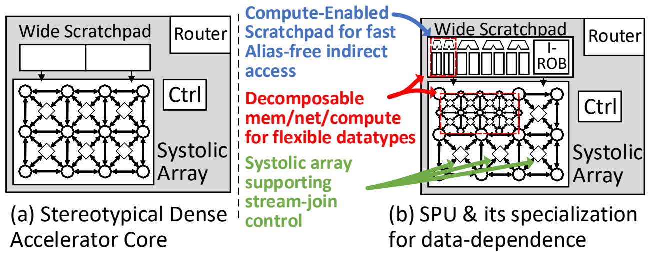

Wide Scratchpad
Router
Ctrl
Systolic Array
(a) Stereotypical Dense Accelerator Core
Compute-Enabled Scratchpad for fast Alias-free indirect access
Decomposable mem/net/compute for flexible datatypes
Systolic array supporting stream-join control
Wide Scratchpad
I-ROB
Router
Ctrl
Systolic Array
(b) SPU & its specialization for data-dependence

Figure 1: Our approach: Data-dependence Specialization

high-throughput data-processors face similar problems.

On top of this, supporting data-dependence makes computing or accessing arbitrary datatypes (8-bit,16-bit,32-bit) more difficult. The traditional approach of sub-word SIMD (used by eg. GPUs) does not suffice in the presence of data-dependent control or memory, as sub-words may have different control outcomes or memory addresses respectively. Hence, the effective bandwidth would have to be reduced to that achievable by the vector-lane's word-length.

Not only is data-dependence problematic, it is also extremely common. Table 1 outlines algorithms from several domains, which rely on data-dependent versions for one of many reasons: Some use a sparse representation to save computation and memory bandwidth/capacity. Others rely on data-structures that represent relationships like graphs or trees. Some use data-subsetting like a database filter or simply reorder data, like Sort or Join.

While some prior accelerators have mechanisms for data-dependence, they tend not to be programmable for different domains (eg. cannot run a DNN on a database accelerator), and they also tend to be inefficient on kernels which are not data-dependent (eg. cannot run a dense-matrix multiply on a sparse-matrix accelerator). Our goal is *to develop a programmable, domain-neutral accelerator which can efficiently execute data-dependent and non-data-dependent algorithms at high efficiency.*

Insight: Our key observation is that the data-dependence support required across data-processing domains is not arbitrary. Two basic forms cover a wide variety of data-dependent kernels: *stream-joins* and *alias-free indirection.*

Stream joins are defined by in-order processing of data, where the relative order of consumption and production of new data is dependent on control decisions. Alias-free indirection is characterized by memory access with data-dependent addresses, but where it can be guaranteed that there are no implicit dependences through aliasing. These restrictions can enable efficient hardware mechanisms, and while simple, these forms are quite general: the algorithms in Table 1 can be expressed as one or the other (or both).

Approach: Because our goal is to be performance-robust across algorithms, we start with an architecture known to work well for non-data-dependent: a systolic-style[1] coarse grained reconfigurable architecture (CGRA) with streaming memory support (extremely common, eg. [10, 19, 22, 46, 54, 98]) – see Figure 1(a). We then develop hardware and software mechanisms for our two data-dependence forms to enable fully-pipelined stream-join and high-bandwidth alias-free indirection at low overhead. Finally, to support a variety of datatypes, we add decomposability into the compute, network, and memory.

Our design, the Sparse Processing Unit (SPU), is shown in Figure 1(b). SPU supports fully-pipelined stream-joins with a systolic CGRA augmented with a novel dataflow-control model. SPU supports high-bandwidth alias-free indirection (load/store/update) with a banked scratchpad with aggressive reordering and embedded compute units. To flexibly support different datatypes, the hardware enables decomposing the reconfigurable network and wide memory access into power-of-two finer-grain resources while maintaining data-dependence semantics. Decomposability is more powerful than subword-SIMD alone, as it effectively lets dataflow of finer-grain datatypes flow independently, which is necessary for independent control flow and indirect memory access.

For the hardware/software interface, we augment a stream-dataflow ISA [65], which enables simple embedding of the memory and control primitives. For the overall design, SPU cores are connected using a traditional mesh network-on-chip (NoC) to create a high-performance multi-core accelerator.

Chosen Workloads: We study machine learning (ML) as our primary domain, and graph processing and databases to demonstrate generality. Chosen ML workloads cover the sparse and dense versions of the top-5 ML algorithms used by Facebook in 2018 [42]. FC and CNN are the core kernels used in state-of-the-art speech and image/video recognition. Arithmetic Circuits are graphical model representations which can be used to answer inference questions on probability distributions [87]. From graph processing we study page rank and BFS. From databases, we study a subset of TPC-H.

Results: We evaluate our approach across three domains:

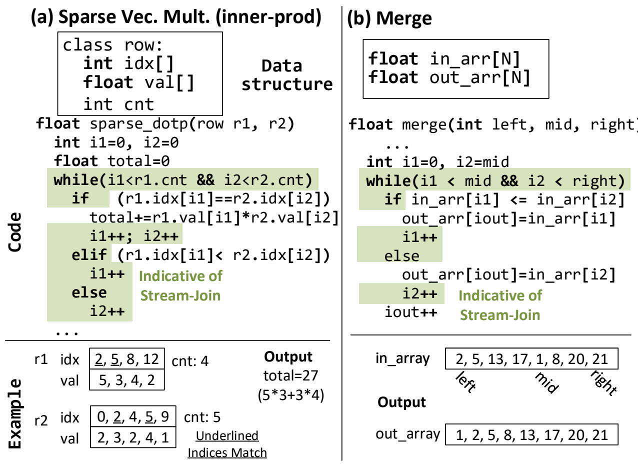
- **AI/ML:** SPU achieves 1.8-7× speedup over a similar GPU (NVIDIA P4000), using 24% power. Further, retaining capability to express dense algorithms led to up to 4.5× speedup.
- **Graph:** For both ordered and unordered algorithms, we achieve 14.2× performance over a 24-core SKL CPU, competitive with the scaled-up Graphicionado [39] accelerator.
- **Database:** For database workloads, we achieve 10.3× over the CPU, which is competitive with the Q100 [106] accelerator.

Our contributions are:

- Identifying two data-dependence forms which are highly-specializable, yet are general across many algorithms.
- Hardware/software codesign for the data-dependence forms: 1. Dataflow control model enabling pipelined stream-joins. 2. Scratchpad supporting high-bandwidth indirect access. 3. Architecture decomposability for different data-type sizes.
- Evaluation of SPU multicore across three domains (AI/ML,Graph,Database) using real-world datasets.

Paper Organization: First, we describe the two key data-dependence forms and their challenges and opportunities (Section 2). Then we describe codesigned abstractions and hardware mechanisms for specializing for data-dependent control (Section 3) and memory (Section 4). We then integrate these to create the proposed SPU accelerator, and explain its parallelism/communication mechanisms (Section 5). Finally we describe the experimental methodology, present our evaluation, and cover related work (Section 6, 7, 8).

[1]By systolic, we mean that processing elements only perform one operation, are fully-pipelined to execute one operation per cycle, and only communicate with neighbors.


(a) Sparse Vec. Mult. (inner-prod)
(b) Merge
Data structure
Code
Example
Indicative of Stream-Join
Indicative of Stream-Join
r1 idx 2, 5, 8, 12 cnt: 4
val 5, 3, 4, 2
r2 idx 0, 2, 4, 5, 9 cnt: 5
val 2, 3, 2, 4, 1
Output
total=27
(5*3+3*4)
Underlined Indices Match
in_array 2, 5, 13, 17, 1, 8, 20, 21
left
mid
right
Output
out_array 1, 2, 5, 8, 13, 17, 20, 21

```
class row:
  int idx[]
  float val[]
  int cnt
float sparse_dotp(row r1, r2)
  int i1=0, i2=0
  float total=0
  while(i1<r1.cnt && i2<r2.cnt)
    if  (r1.idx[i1]==r2.idx[i2])
      total+=r1.val[i1]*r2.val[i2]
      i1++; i2++
    elif (r1.idx[i1]< r2.idx[i2])
      i1++
    else
      i2++
  ...
```

```
float in_arr[N]
float out_arr[N]

float merge(int left, mid, right)
  ...
  int i1=0, i2=mid
  while(i1 < mid && i2 < right)
    if in_arr[i1] <= in_arr[i2]
      out_arr[iout]=in_arr[i1]
      i1++
    else
      out_arr[iout]=in_arr[i2]
      i2++
    iout++
```

Figure 2: Example Stream-Join Algorithms

2 EXPLOITABLE DATA-DEP. FORMS

We observe that two restricted forms of data-dependence are sufficient to cover many algorithms: *stream-join* and *alias-free indirection*. In this section, we first define these forms and give some intuition on their performance challenges for existing architectures, then explain how they guide our design.

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, used as architecture primitives in many prior designs [25, 26, 44, 65, 83, 102, 106]. Relevant to this work are memory streams, which are sequences of loads or stores. Streams are similar to vector accesses, but have no fixed length (for examples see Listings 1-3 on page).

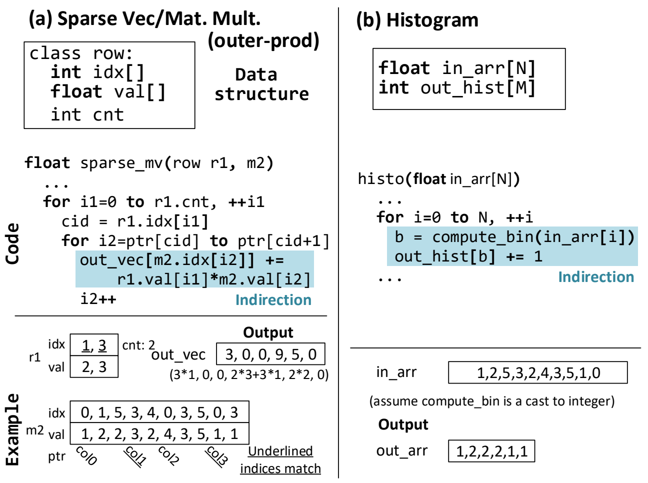
Preliminary Term – "Regular Algorithm": A regular algorithm is one with no data-dependent control decisions or memory addresses. Further, no implicit dependences are allowed through memory streams, created by aliasing. The data-dependence forms can be viewed as relaxations of regular algorithms.

2.1 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 rows (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, datatypes 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 which is regular, except that the re-use of stream data and production of outputs may depend on the data.

Problem for CPUs/GPUs: Because of their data-dependent nature, Stream-joins introduce branch-mispredictions for CPUs. For GPGPUs, control dependence makes vectorization difficult due to control divergence of SIMT lanes; also the memory pattern can diverge between lanes, causing L1 cache bank conflicts.



```
class row:
  int idx[]
  float val[]
  int cnt

float sparse_mv(row r1, m2)
  ...
  for i1=0 to r1.cnt, ++i1
    cid = r1.idx[i1]
    for i2=ptr[cid] to ptr[cid+1]
      out_vec[m2.idx[i2]] +=
            r1.val[i1]*m2.val[i2]
    i2++
```

```
float in_arr[N]
int out_hist[M]

histo(float in_arr[N])
  ...
  for i=0 to N, ++i
    b = compute_bin(in_arr[i])
    out_hist[b] += 1
  ...
```

Figure 3: Example Alias-Free Scatter/Gather Algorithms

Our Goal for stream-join: Create a dataflow control model which can execute stream-join at full bandwidth and utilization.

2.2 Alias-Free Indirection (AF-Indirect)

Many algorithms rely on indirect read, write, and update to memory, often showing up as `a[f(b[i])]`. Figure 3 shows two examples: The sparse-vector/sparse-matrix outer product (a) works by performing all combinations of non-zero multiplications, and accumulating in the correct location in a dense output vector. Histogram (b) is straightforward. The similarities here are clear: both perform an 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.

Alias-Free Indirection Definition: A program region which is regular (including no implicit dependences), except that memory streams may be dependent on each other.

Problem for CPUs/GPUs: On CPUs, indirect memory is possible with scatter/gather, however the throughput is quite limited given the limited ports to read/write vector-length number of cache lines simultaneously. As for indirect update, Intel AVX512 recently added support for conflict detection instructions. These *do not* improve the above cache-port throughput problem, only the instruction overhead – yet still any conflicts within the vector are *handled serially* with no reordering across vectors [45]. Also, not leveraging alias-freedom means a reliance on expensive load-store queues.

While GPUs have similar throughput issues for caches, their scratchpads are banked for faster indirect access. However, they do not reorder requests across subsequent vector warp accesses [103], which is important to get high indirect throughput. Doing so in a GPU would require dependence-checking of in-flight accesses, as they cannot guarantee alias freedom.

Our Goal for AF-Indirect: Create a stream-based hardware/software interface and microarchitecture enabling indirect access at full bandwidth through aggressive reordering.

Dependence Form Relationship: Finally, note that dependence forms are not mutually exclusive. An example is the histogram-based Sparse GBDT (Figure 13 on Page). Alias-free indirection is used for updating the histogram count, while a stream-join is used for iterating over the sparse feature values. Other examples include deep neural networks (indirection for matrix-multiply and stream-join to subsequently resparsify the output vector) and triangle counting in graphs (indirection to traverse the graph and stream-join to find intersecting neighbors).

3 STREAM-JOIN SPECIALIZATION

A conventional computational fabric which is proven to perform well for non-data-dependent codes is a systolic-execution array [19, 22, 46, 54, 98], as they are quite simple. Note we define a systolic-execution array as a set of processing tiles which together form a deep pipeline, where each tile executes a single logical instruction and only communicates with its neighbors. This definition is general enough that such designs can include a circuit-switched network [37, 57, 66, 88, 99], so we refer to these as systolic CGRAs.

In this section, we propose a novel control model to enhance a conventional systolic CGRA for stream-join. We also discuss supporting finer-grain datatypes at low overhead and high hardware utilization through decomposability.

3.1 Stream-join Control

Existing systolic arrays are unable to make control decisions beyond simple predication, as they do not account for data dependences in deciding when and how to produce or consume data.

We discuss a number of examples in Figure 4, for which show the original code and a traditional dataflow representation. Here, black arrows represent data dependence, and green arrows indicate control. The dataflow representation is quite similar to what is executed on an OOO core. These examples motivate the need for a new dataflow-control model; one which can express the data-dependence without expensive throughput-limiting control dependence loops; this figure also shows these codes represented in our stream-join dataflow model.

Merge Example: Consider the pseudo-code in Figure 4(a), which shows a simple merge kernel (only the part where both lists have data), for use in merge-sort for example. An item is selected and stored based on which of two items is smaller. This dataflow can be mapped to a systolic array, 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. Note that the same problem exists for the out-of-order core, and it is made even worse with the unpredictable data-dependent branch which would increase the average latency due to mispredictions.

However, note that from the perspective of the memory, the control dependence is unnecessary, as all loads will be performed anyways. Therefore, to break the dependence, we need to separate the loads from computation (luckily, decoupled streams do this already), then expose a mechanism for controlling the order of data consumption. Intuitively for this example, if the model treats

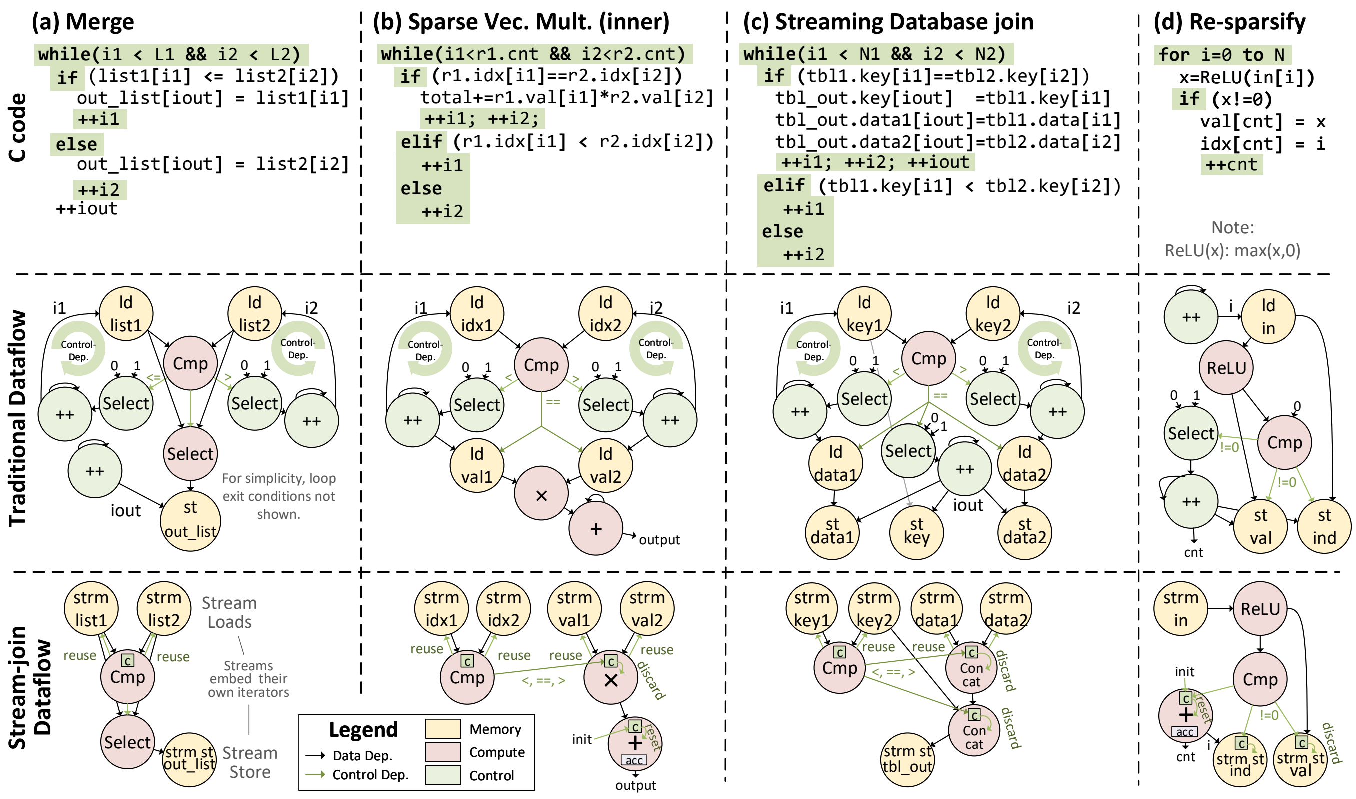

(a) Merge
```
while(i1 < L1 && i2 < L2)
  if (list1[i1] <= list2[i2])
    out_list[iout] = list1[i1]
    ++i1
  else
    out_list[iout] = list2[i2]
    ++i2
  ++iout
```
(b) Sparse Vec. Mult. (inner)
```
while(i1<r1.cnt && i2<r2.cnt)
  if (r1.idx[i1]==r2.idx[i2])
    total+=r1.val[i1]*r2.val[i2]
    ++i1; ++i2;
  elif (r1.idx[i1] < r2.idx[i2])
    ++i1
  else
    ++i2
```
(c) Streaming Database join
```
while(i1 < N1 && i2 < N2)
  if (tbl1.key[i1]==tbl2.key[i2])
   tbl_out.key[iout]  =tbl1.key[i1]
   tbl_out.data1[iout]=tbl1.data[i1]
   tbl_out.data2[iout]=tbl2.data[i2]
   ++i1; ++i2; ++iout
  elif (tbl1.key[i1] < tbl2.key[i2])
   ++i1
  else
   ++i2
```
(d) Re-sparsify
```
for i=0 to N
  x=ReLU(in[i])
  if (x!=0)
    val[cnt] = x
    idx[cnt] = i
    ++cnt
```
Note:
ReLU(x): max(x,0)

Figure 4: Stream-Join Control Model

incoming values like a queue, it is possible to "pop" the values as they are consumed. Essentially, what we require here within the computation fabric is the ability to perform *data-dependent reuse.*

Sparse Inner-Product Example: Figure 4(b) is a sparse-vector multiplication. Here, two pointers are maintained based on the comparison of corresponding item indices. Compared to merge, there is a similar control dependence and overhead. A similar approach could work here as well, decouple the streams and conditionally reuse indices (and values). The difference is that we only apply the multiply accumulate on matching indices, so we should discard some of this data. Therefore, in addition to data-dependent reuse, we also require *data-dependent discard.*

Database Join Example: Figure 4(c) shows an inner equijoin. It iterates over sorted keys, and concatenates equivalent keys and corresponding columns. It has a surprisingly similar form and control dependence loop to the sparse multiplication, where the computation is replaced with concatenation. A similar approach of decoupling streams and applying data-dependent reuse and discard will break the control dependence loop and enable high throughput.

Re-sparsification Example: Re-sparsification (Figure 4(d)) produces a sparse row from a dense stream. The dataflow version has a predicated increment and store, and can achieve a pipelined schedule (so can the OOO core if it has predicated stores, otherwise it would be serialized by mispredictions). This example demonstrates that the ability to discard (ie. filter) is useful on its own. It is also an example where predication is enough, whereas predication is insufficient in the other examples.

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. The basic idea is to allow each instruction to perform the following control operations: re-use inputs, discard instructions or reset a register based on a dataflow input.

Figure 5 shows the execution flow of the sparse vector multiplication when expressed as a stream-join, showing the fully-pipelined execution over several cycles. Dataflow values are represented as circles, and for simplicity they take one cycle to flow along a dependence. Sentinel values (infinity for indices and zero for values) are used to indicate the end of a stream; these allow the other stream to drain on stream completion. Also, the subsequent vector multiplications can begin without draining the pipeline.

Figure 4 shows all of the examples written in this model. Data-dependent operand re-use is useful in (a,b,c) to iterate over input streams in correct relative order. Data-dependent *discard* is also useful in (b,c,d) for ignoring data which is not needed. The data-dependent *reset* is useful in (b,d) for resetting the accumulator. In both examples, adding stream-join primitives to instruction execution either shrinks the throughput-limiting dependence chain or eliminates it completely, enabling a fully-pipelined dataflow.

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

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

Stream-join Overheads: In kernels where input data is discarded (eg. sparse-matrix multiply and database join), the transformation to stream-joins can cause additional loads. Theoretically the

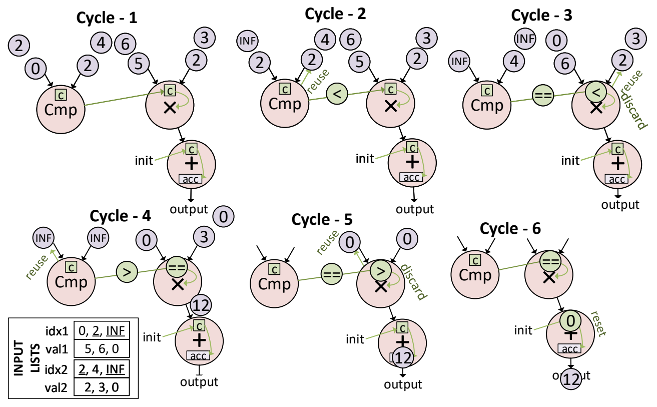


Figure 5: Execution diagram for join of two sorted lists.

worst case extra overhead compared to the original is $(value\_size + key\_size)/(key\_size)$. This can happen if there are extremely sparse matches, for example in databases. For such cases, we could only load values for matching indices; this would increase the latency of accessing values, but this can usually be hidden. For this, SPU provides efficient support for indirect accesses (Section 4).

3.2 Stream-join Compute Fabric: DGRA

Here we explain how we augment a systolic CGRA to support stream-join control. Its network is decomposable to support control semantics for smaller datatypes; thus we refer to the design as the decomposable granularity reconfigurable architecture: DGRA.

Stream-join Processing Element (PE) Implementation: The stream-join control model enables an instruction to 1. treat its inputs as queues that it can conditionally reuse, 2. conditionally discard its output value, and/or 3. conditionally reset its accumulator. Instructions may use their output or a control input to specify the conditions (ie. the control info).

To implement, we add a control lookup table (CLT) to each FU (Figure 6), which determines a mapping between the control inputs and possible control operations. For the inputs of this table, we use the lower two bits of either the instruction output or the control input. For the outputs, there are four possible control actions: reuse-first-input, reuse-second-input, discard-operation, reset-accumulator. Therefore, for a fully configurable mapping between the 2-bit input (four combinations) and 4-bit outputs, we require a 16-bit table, and one extra bit to specify whether the instruction output or control input should be used as input. This becomes additional instruction configuration.

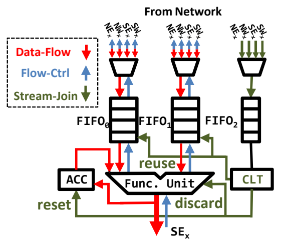


Figure 6: CLT integration

Supporting Decomposability: To support stream-join semantics with arbitrary datatypes, our approach is to support the principle of *decomposability* – the ability to use a coarse grain resource as multiple finer grain resources. Therefore, the network of the DGRA is decomposable into multiple parallel finer-grain sub-networks. It provides limited connectivity between these sub-networks. For this we require both a decomposable switch and PE.

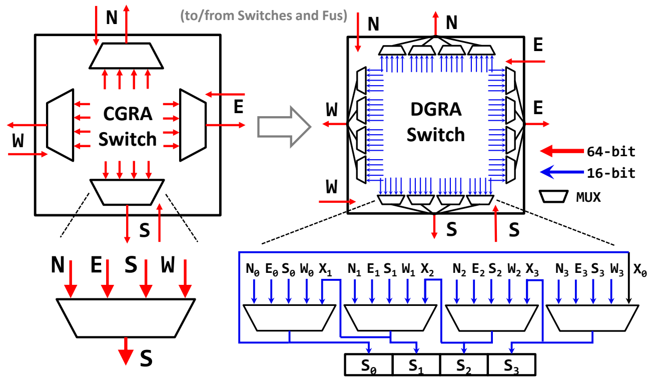

N
(to/from Switches and Fus)
CGRA Switch
E
W
S
DGRA Switch
64-bit
16-bit
MUX
N E S W
S
N₀ E₀ S₀ W₀ X₁ N₁ E₁ S₁ W₁ X₂ N₂ E₂ S₂ W₂ X₃ N₃ E₃ S₃ W₃ X₀
S₀ S₁ S₂ S₃

Figure 7: DGRA Switch

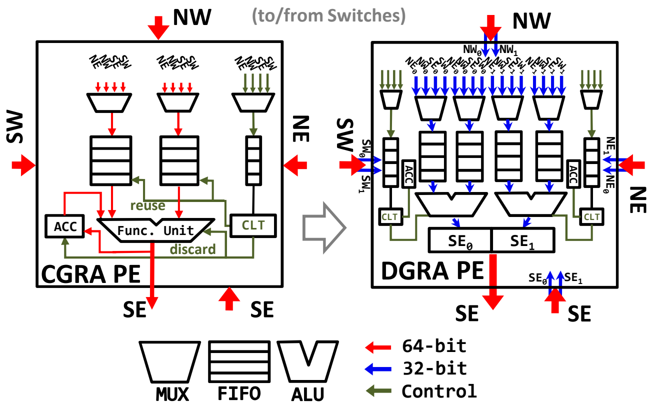


Figure 8: DGRA Processing Element

DGRA Switch: Figure 7 compares a CGRA switch to our DGRA switch. On the left is an implementation of a coarse grain switch, which has one Mux per-output. The DGRA switch decomposes inputs and outputs, and separately routes each 16-bit sub-network. Flow control is maintained separately with a credit path (not shown) for each subnetwork. For flexible routing, we add the ability for incoming values to change sub-networks. In the design this is done by adding an additional input to each output Mux, which uses the latched output of the previous Mux. This forms a ring, as shown by the "X" inputs.

DGRA PE: The decomposable PE (Figure 8) follows the same principles as the switch. Each coarse grain input of a FU can be decomposed into two finer-grain inputs which are used to feed two separate lower-granularity FUs. We replicate the CLT for each subnetwork so that each can have their own control semantics.

Mixed-precision Scheduling: Mapping dataflow graphs onto reconfigurable architecture is known as spatial scheduling (eg. [67, 73, 74, 109]). Adding decomposability increases the complexity due to managing more routing decisions due to subnetworks. At a very high-level, our approach combines the principle of stochastic-scheduling [64] and over-provisioning (eg. within Pathfinder [56]). At each iteration, we attempt to map (or re-map) a dataflow instruction and its dependences onto several different positions on the DGRA; the algorithm will typically choose the position with the highest objective, but will occasionally select a random position. To avoid getting stuck in local minima, we allow over-provisioning

```
for i=0 to n
  ... = a[i]              →  load(a[0:n])
```

Listing 1: Linear Stream

```
struct{int f1, f2} a[n]
for i=0 to n                 str1 = load(index[0:n])
  ind = index[i]             ind_load(addr=str1, offset_list={0,4})
  = a[ind].field1       →    ind_store(addr=str1, value = ...,
  = a[ind].field2                            offset_list={0})
  c[ind] = ...
```

Listing 2: Indirect Load and Store Streams

```
for i=0 to n                 str_ind = load(index[0:n])
  ind = index[i]             str_val = load(value[0:n])
  val = value[i]        →    update(addr=str_ind, val = str_val,
  histo[ind] += val                 opcode="add", offset_list={0})
```

Listing 3: Indirect Update Stream

```
for i=0 to n
  size = sub_size[i]         str_size = load(sub_size[0:n])
  for j=0 to size       →    data_dep_load(value[0:len=str_size])
    val = value[j]
```

Listing 4: Data-dependent Load Stream

compute and network resources and penalize over-provisioning in the objective function.

4 SPECIALIZING DATA-DEP. MEMORY

The main challenge for specializing for alias-free indirection is creating a high-bandwidth memory pipeline which aggressively reorders accesses. In order to explain our proposed microarchitecture, we first discuss the set of stream abstractions that are expressed to the hardware.

4.1 Sparse Memory Abstractions

As we explained earlier, we start with a simple non-data-dependent contiguous stream (Listing 1 shows an example). For specifying indirect loads or stores, we enable one streams' addresses to be dependent on another streams' values. Often, *array-of-structs* style data structures require several lookups offset from the base address. We add this capability with an "offset list", shown in the example in Listing 2.

Indirect updates (as in histogramming) could hypothetically be supported by using an indirect load stream as above, performing the reduction operation, and finally using an indirect store stream to the same series of addresses. However, this requires dynamic alias detection or eschewing pipeline parallelism to prevent aliasing read-/write pairs from being mis-ordered. Instead, we can leverage the alias-free property to add a specialized interface for indirect update. In our implementation, indirect update may perform common operations like add, sub, max, and min directly on the indirect-addressed data item. Listing 3 shows an example.

Often, streams consist of sub-streams with data-dependent length. For example, indirect matrix-vector multiplication requires access to columns with varying size (Figure 3, page). We enable streams to specify a data-dependent *length*, as in Listing 4.

4.2 Data-Dep Memory Microarchitecture

Armed with expressive abstractions, we develop a high-bandwidth and flexible scratchpad controller capable of high-bandwidth indirect access. 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

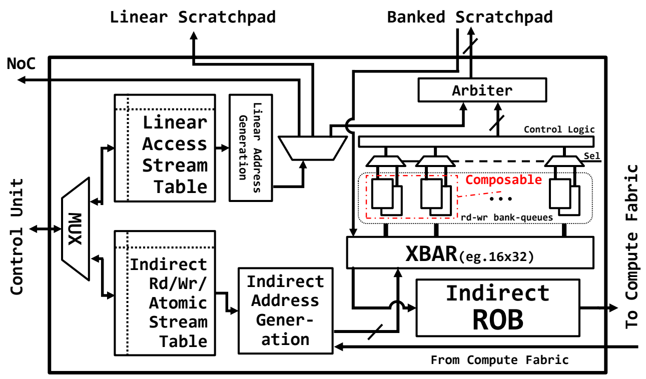


Figure 9: Scratchpad Controller

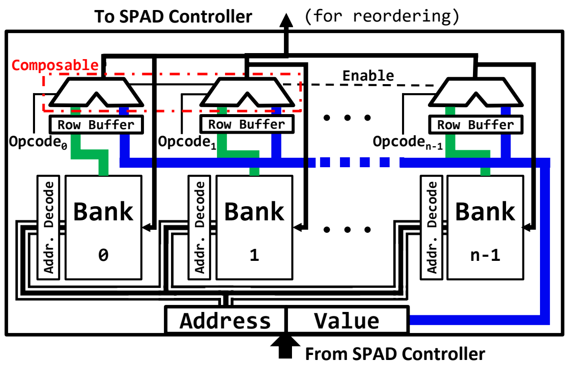


Figure 10: Compute-enabled Banked Scratchpad

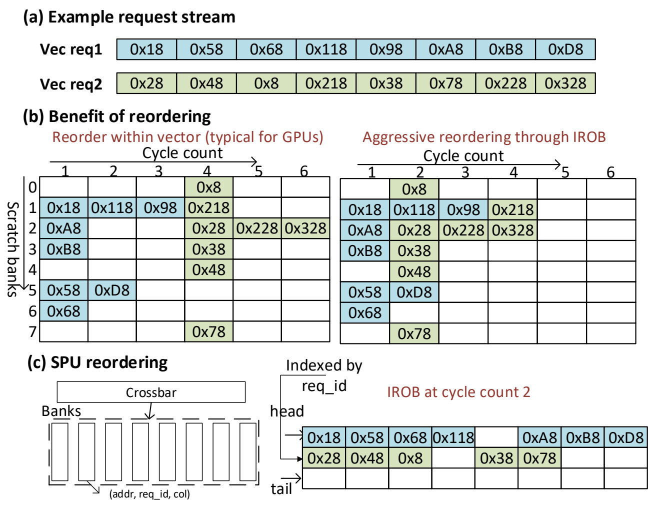
with two logical scratchpad memories, one highly *banked* and one *linear*. In this design, both exist within the same address space.

The role of the scratchpad controller (eventual design in Figure 9) 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 tables. The controller should then select between any concurrent stream for address generation and send 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 Section 3) with each element in the stream description's offset list. This vector of requests is sent to an arbitrated crossbar for distribution to banks, and a set of queues buffer requests for each SRAM bank (Figure 10) until they can be serviced. Reads, writes and updates are explained as follows:

Indirect Writes: Bank queues buffer both address and values. Importantly, because writes are not ordered with respect to anything besides barriers, requests originating from within the stream and across streams can be "mixed" within the bank queues without any additional hardware support. Mixing requests across multiple request-vectors helps to hide bank contention, a critical feature enabling higher throughput than traditional memories.

Indirect Updates: Indirect updates use the compute units within the scratchpad. To explain, the bank queues buffer the address, operation type and the operand for the update. Within the banked


(a) Example request stream

Vec req1	0x18	0x58	0x68	0x118	0x98	0xA8	0xB8	0xD8
Vec req2	0x28	0x48	0x8	0x218	0x38	0x78	0x228	0x328

(b) Benefit of reordering

Reorder within vector (typical for GPUs)

Scratch banks	1	2	3	4	5	6
0				0x8		
1	0x18	0x118	0x98	0x218		
2	0xA8			0x28	0x228	0x328
3	0xB8			0x38		
4				0x48		
5	0x58	0xD8				
6	0x68					
7				0x78		

Aggressive reordering through IROB

Scratch banks	1	2	3	4	5	6
0		0x8				
1	0x18	0x118	0x98	0x218		
2	0xA8	0x28	0x228	0x328		
3	0xB8	0x38				
4		0x48				
5	0x58	0xD8				
6	0x68					
7		0x78				

(c) SPU reordering

IROB at cycle count 2

0x18	0x58	0x68	0x118		0xA8	0xB8	0xD8
0x28	0x48	0x8		0x38	0x78		

Figure 11: Functioning of IROB. (bits<6..4> indicate bank number)

scratchpad, after the value is read, the associated compute unit executes, then writes the value back to the same location in the next cycle. We support only common integer operations within this pipeline (add, sub, min, max). The pipeline stalls only if subsequent updates are to the same address (max 2-cycle bubble).

Indirect Reads: In contrast to the above, the order of *reads* must be preserved. For performance, we would like to maintain the ability to mix requests from subsequent accesses to hide bank contention. This actually goes beyond what even modern GPUs are capable of, as they only reorder a single vector of requests at one time [7, 58, 103]. We believe the reason for this limitation on GPUs is the challenge in handling potential memory dependences. To explain, Figure 11(a) shows an example of two parallel indirect read requests. Figure 11(b) shows the difference between how a typical GPU approach would schedule transactions, and how an aggressive reordering approach would work. The ability to intermingle parallel requests can significantly increase throughput.

To accomplish this, we maintain ordering in a structure called an indirect read reorder-buffer (IROB), which maintains incomplete requests in a circular buffer. It is allocated an entry whenever a request is generated from the indirect address generator. For indirect reads, the bank queues maintain the address and row & column of the IROB. As results return from the banked scratchpad, they use this row & column to update the IROB. IROB entries are deallocated in-order when a request's data is sent to the compute unit. Overall, our abstractions enable expression of the alias-free property of indirect reads in hardware, which is what allows a simple hardware structure like the IROB to aggressively reorder across multiple requests without memory dependence checking.

Decomposability: The indirect scratchpad also requires decomposability to various datatypes. Multiple contiguous lanes are used in lock-step to support larger datatypes. Consider indirect store bandwidth for example: the 16×32 crossbar either supports 16 16-bit stores (to 32 logical banks), 8 32-bit stores (to 16 logical banks), or 4 64-bit stores (to 8 logical banks). We use the same approach for accessing the SRAM banks of the indirect scratchpad.

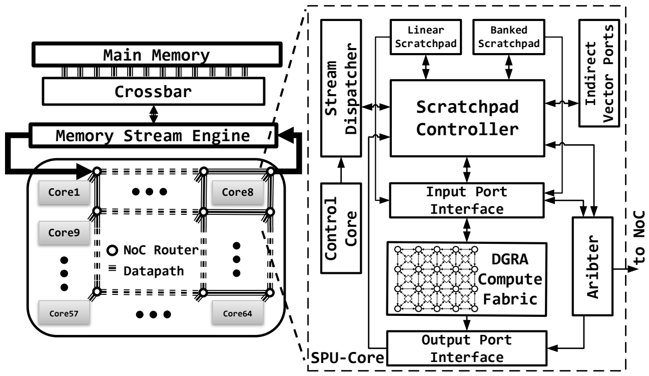


Figure 12: SPU Overview

5 SPARSE PROCESSING UNIT

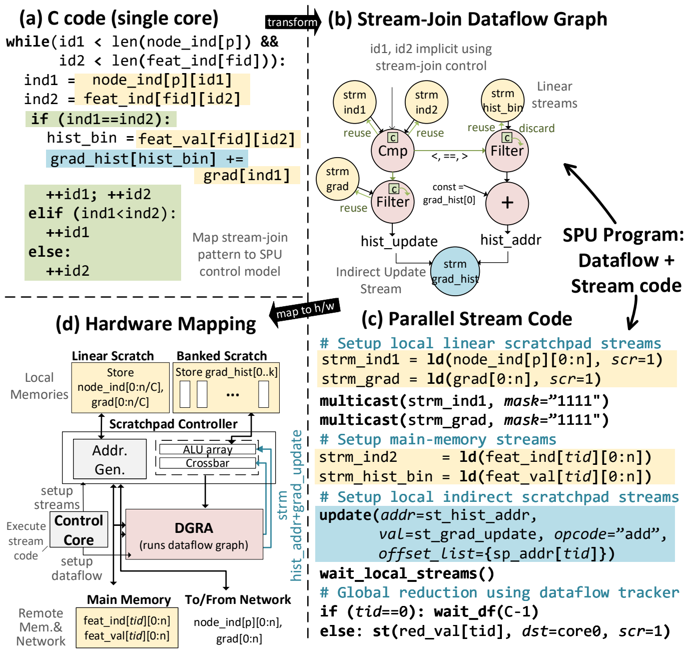
The sparse processing unit (SPU) is our overall proposed design. Each SPU core is composed of the specialized memory and compute fabric (DGRA), together with a control core for coordination among streams. In this section, we overview the primary aspects of the design, then discuss how we map our workloads to SPU's computation, memory, and network abstractions. Finally, we discuss the role of the compiler and possible framework integration.

SPU Organization: Figure 12 shows how SPU cores would be integrated into a mesh network-on-chip (NoC), along with the high-level block diagram of the core. The basic operation of each core 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.

Memory Integration: These workloads require shared access to a larger pool of on-chip memory. To enable this, our approach was to rely on software support, rather than expensive general purpose caches and coherence. In particular, SPU uses a partitioned global address space for scratchpad. Data should be partitioned for locality if possible. Streams may access remote memory over the NoC. We add remote versions of the indirect read, write, and update streams. Indirect write and update are generally one-way communication operations, but we provide support to synchronize on the last write/update of a stream for barrier synchronization. Other synchronization is described next.

Communication/synchronization: SPU provides two specialized mechanisms for communication. First, we include multicast capability in the network. Data can be broadcast to a subset of cores, using the same relative offset in scratchpad. As a specialization for loading main memory, cores issue their load requests to a centralized memory stream engine, and data can be multi-cast from there to relevant cores. For synchronizing on for data-readiness, SPU uses a dataflow-tracker-like [98] mechanism to wait on a count of remote-scratchpad writes.

Control ISA: We leverage an open-source stream-dataflow ISA [63, 65] 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.


(a) C code (single core)

```
while(id1 < len(node_ind[p]) &&
      id2 < len(feat_ind[fid])):
  ind1 =  node_ind[p][id1]
  ind2 = feat_ind[fid][id2]
  if (ind1==ind2):
    hist_bin = feat_val[fid][id2]
    grad_hist[hist_bin] +=
                        grad[ind1]
    ++id1; ++id2
  elif (ind1<ind2):
    ++id1
  else:
    ++id2
```

(c) Parallel Stream Code

```
# Setup local linear scratchpad streams
strm_ind1 = ld(node_ind[p][0:n], scr=1)
strm_grad = ld(grad[0:n], scr=1)
multicast(strm_ind1, mask=”1111")
multicast(strm_grad, mask=”1111")
# Setup main-memory streams
strm_ind2     = ld(feat_ind[tid][0:n])
strm_hist_bin = ld(feat_val[tid][0:n])
# Setup local indirect scratchpad streams
update(addr=st_hist_addr,
       val=st_grad_update, opcode=”add”,
       offset_list={sp_addr[tid]})
wait_local_streams()
# Global reduction using dataflow tracker
if (tid==0): wait_df(C-1)
else: st(red_val[tid], dst=core0, scr=1)
```

Figure 13: Example SPU Program Transformation: GBDT (Each core gets a subset of features to process i.e. fid=*tid*)

Programming Model: Programming SPU involves the following tasks: 1. partitioning work to multiple cores and data to the scratchpads to preserve locality, 2. extracting the dataflow graph, and possibly re-writing data-dependent control as a stream-join, 3. extracting streaming memory accesses, and 4. inserting communication/synchronization.

In terms of programming abstractions, an SPU's program consists of a dataflow graph language describing the computation (compiled to DGRA), along with a control program which contains the commands for streams (similar to stream-dataflow [65]). When a control program is instantiated, it is made aware of its spatial location, for efficient communication with its neighbors.

Example Program: To explain how to map programs to SPU abstractions, we use the example of GBDT in Figure 13. We show the key kernel of this workload, which is a histogram over sparse lists. Figure 13(a) shows the original kernel's C code. Figure 13(b) shows the extracted stream-join dataflow, and (c) shows the control program where memory accesses are represented as streams, which is expressed as C + intrinsics. Each stream loads (or stores) data to an input (or output) in the dataflow. Figure 13(d) shows how the SPU program is mapped to hardware for this algorithm. In hardware, the stream code executes on the control core, which creates streams to be executed on the scratchpad controller. In turn, the controller will deliver/receive data to/from the DGRA compute unit, which executes the dataflow.

The basic parallelization strategy is that each SPU core independently builds histograms corresponding to its allotted subset of features. As for how memory is distributed, the dataset is stored in main memory in sparse CSR format (feat_ind and feat_val). Accessing these requires linear memory streams. The linear scratchpads store the subset of instances which belong to the current working node. As node indices are common across all features, the

	Mech./ Wkld	Indirect memory	Stream join	Work partition across cores	Synchronization
Machine Learning (ML)	GBDT	Create feature histogram	Join train inst subset	Split features	Hierarch. reduce + broadcast
	KSVM	–	Matrix-mult for error calc	Split training instances	Hierarch. reduce + broadcast
	AC	Read + update child parameters	–	Split DAG levels	Pipelined communication
	FC	Accumulate activations	Resparsify	Split weight matrix rows	Broadcast of i/p activations
	CONV	Accumulate activations	Resparsify	Split weight matrix rows	Nearest neighbor comm.
Database	Merge Sort	–	Merge of 2 sorted lists	Uniform partition	Hierarchical Merge
	Hash Join	Cuckoo hash lookup	–	Smaller col replicated	Barrier until each core completes
	Sort Join	–	Join sorted lists	Equal-range partition	Same as above
Graph	Page Rank	Accumulate vert. rank	–	All vertices	Remote ind. 'add' update
	BFS	Relax vert. distance	–	Active vertices	Remote ind. 'min' update

Table 2: Mapping of Algorithms on SPU

corresponding data is broadcast across all cores. This is done in synchronous phases: in each phase, the stream is loaded from a predetermined core. Phases are not shown in the figure for brevity.

As for the dataflow, stream-join is used to iterate sparse feature indices and the indices generated due to subsetting the data at each decision tree node.

Indirection is used for histogramming: the histogram address and update values are produced in the dataflow, which are then consumed by the indirect update stream. In hardware, the stream is mapped to the indirect stream table in the scratchpad controller.

Workload Mapping: Table 2 details how we map each algorithm to the SPU architecture in terms of control, memory and communication ISA primitives, as well as the partitioning strategy.

Framework Integration: We envision that SPU can be targeted from frameworks like TensorFlow [8], Tensor Comp. [97], TVM [21] for machine learning, or from a DBMS or graph analytics framework [3, 92]. For integration, a simple library-based approach can be used, where programmers manually write code for a given machine learning kernel. This is the approach we take in this work. Automated compilation approaches, eg. XLA [2] or RStream [78] are also possible if extended for data-dependent algorithms.

6 METHODOLOGY

SPU: We implemented SPU's DGRA in Chisel [13], and synthesized using Synopsys DC with a 28nm UMC technology library. We use Cacti [60] for SRAMs and other components. When comparing to GPU power, we omit memory and DMA controllers. We built an SPU simulator in gem5 [14, 84, 94], using a RISCV ISA [11] for the control core.

Architecture Comparison Points: Table 3 shows the characteristics of the architectures we compare against, including their on-chip memory sizes, FU composition, and memory bandwidth. As for SPU, we provisioned the size of the DGRA to match the combined throughput of the scratchpads. We provisioned the total amount

Characteristics	GPU [27]	SPU-inorder	SPU
Processor	GP104	in-order	SPU-core
Cache+Scratch	4064KB	2560KB	2560KB
Cores	1792	512	64 SPU cores
FP32 Unit	3584	2048	2432
FP64 Unit	112	512	160
Max Bw	243GB/s	256GB/s	256GB/s

Table 3: Architecture characteristics of GPU, SPU-inorder and SPU

Workloads	CPU	GPU
GBDT	LightGBM [49]	LightGBM [49]
Kernel-SVM	LibSVM [18]	hand-written [12]
AC	hand-written [87]	hand-written [87]
FC	Intel MKL SPBLAS [1]	cuSPARSE [61]
Conv layer	Intel MKL-DNN [4]	cuDNN [24]
Graph Alg.	Graphmat [92]	-
TPCH	MonetDB [15]	-

Table 4: Baseline workload implementations

	Dataset	Size	Density	Dataset	Size	Density
GBDT	Cifar10-bn	50k,3k	1	Yahoo-bn	723k,136	0.05
#inst,#feat.	Higgs-bn	10M,28	0.28	Ltrc-bn	34k,700	0.008
KSVM	Higgs	10M,28	0.92	Connect	67k,700	0.33
#inst,#feat.	Yahoo	723k,136	0.59	Ltrc	34k,700	0.24
CONV	Vgg-3	802k,73k	0.47,0.4	Vgg-4	1.6M,147k	0.4,0.35
#act,#wgt	Alex-2	46k,307k	0.68,0.17	Res-1	150k,9.4k	0.99,0.1
FC	Res-fc	512,512k	0.26,0.84	Vgg-13	4K,16.8M	0.14,0.3
#act,#wgt	Alex-6	9K,37.7M	0.29,0.09	Vgg-12	25k,103M	0.42,0.06
AC	Pigs	622k	NA	Munin	3.1M	NA
#nodes	Andes	727k	NA	Mildew	3.7M	NA
Graph	Flickr	820K,9.8M	0.000015	NY-road	260K,730K	0.00005
#node,#edge	Fb-artist	50K,1.63M	0.0064	LiveJournal	4.8M,68.9M	0.000003

Table 5: Datasets

of memory on-chip for the working-sets of ML workloads, as they were our primary focus; this has some impact on workloads which have large working-sets and are expensive to tile.

As for comparison to real hardware, the GPU is the most relevant. We choose the NVIDIA P4000, as it has a slightly larger total throughput and similar memory bandwidth to SPU. We do not include CPU-GPU data-transfer time.

We also address whether an inorder processor is sufficient by comparing against "SPU-inorder", where the DGRA is replaced by an array of 8 inorder cores (total of 512 cores). For reference, we also compared against a dual socket Intel Skylake CPU (Xeon 4116), with 24 total cores.

Workload Implementations: We implement SPU kernels for each workload, and use a combination of libraries and hand-written code to compare against CPU/GPU versions. We compared against the best implementation (that we were aware of) for each workload on real hardware (Table 4). We implement kernels using both dense and sparse data-structures wherever possible (shown as SPU-dense/sparse).

Our choice of modest on-chip memory affects the implementations of graph processing and database workloads. For processing larger graphs, we follow a similar technique as proposed in Graphicionado [39] to split the graph into "slices" that fit in on-chip memory. Edges with a corresponding vertex in another slice are instead connected to a copy of that vertex; duplicates are kept consistent. An architecture with a larger on-chip memory (such as Graphicionado [39], which has 32MB on-chip memory) means

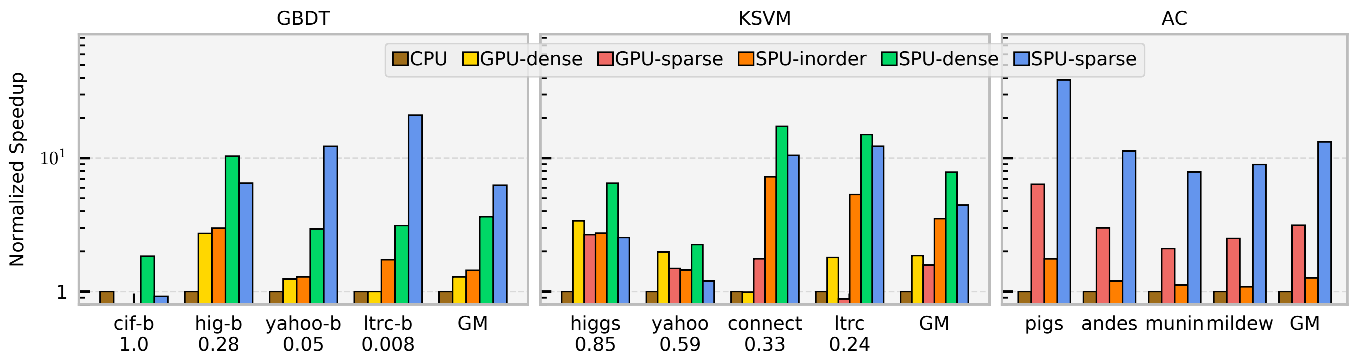


Figure 14: Performance on GBDT, KSVM, AC. (Computation density under benchmark name)

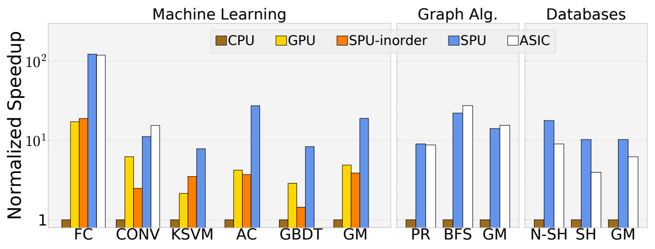


Figure 15: Overall Performance

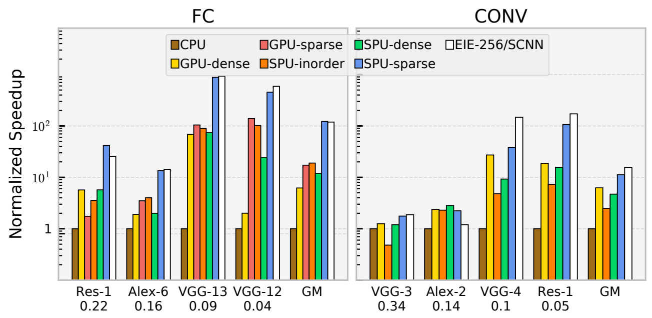


Figure 16: Performance on DNN. (Compute density under bench name)

less duplicates, and less overhead. The tradeoff is also relevant for database workloads. Hash-joins require the hash-table to fit on-chip to perform well.

Benchmarks: We used the datasets specified in Table 5. The uncompressed DNN model is obtained from Pytorch model zoo and the compression is done as described in [41] using distiller [6].

Domain-Specific Accelerator Modeling: We model all domain-specific accelerators using optimistic models appropriate to the domain, always considering memory and throughput limitations of actual data.

(1) **SCNN [72]:** We use a compute-bound model of SCNN according to the dataset density, assuming no pipeline overhead besides memory conflicts.
(2) **EIE [40]:** Mechanistic model of EIE at maximum throughput. We compare against the scaled version of EIE with 256 cores.
(3) **Graphicionado [39]:** We modeled a cycle-level approximation of its pipeline stages. We also compare against a version of this accelerator with the same peak-memory bandwidth as SPU by scaling Graphicionado to 32-cores and 32x32 crossbar.
(4) **Q100 [106]:** For fair comparison to Q100, we restrict SPU to 4 cores (approximately the same area as Q100). We hand-coded query plans for Q100, specified as a directed acyclic graph in which each node indicates a database operation (join, sort, etc.) supported by the Q100 hardware, and edges indicate producer-consumer dependencies. Our model of Q100 is an optimistic execution of this query plan under memory and compute bandwidth constraints, which we verified against baseline execution time and speedups given by Q100's authors. This query plan is used as a reference for the SPU version, so SPU and Q100 implement the same algorithm as much as possible.

7 EVALUATION

Our evaluation broadly addresses the question of whether data-dependencies exposed to an ISA (and exploited in hardware) can help achieve general-purpose acceleration. Here are the key take-aways, in part based on the overall performance results in Figure 15.

(1) SPU achieves high speedup over CPUs (for ML:16.5×, Graph:14.2× and DB:10.3×), and GPUs (ML:3.87×).
(2) Performance is competitive with domain-accelerators.
(3) Relying on inorder cores only for supporting data-dependence is insufficient.
(4) Architectural generality provides the flexibility to choose algorithmic variants depending on the algorithm and dataset.

7.1 Performance on Machine Learning

Here we discuss the per-workload performance results on ML workloads, the breakdown for GBDT/KSVM/AC is in Figure 14 and for DNN is in Figure 16. During our analysis, we refer to Figure 17, which describes the utilization of compute, scratchpad, network, and memory within SPU.

GBDT: Both GPUs and SPUs use a histogram-based approach, but SPU's aggressive reordering of indirect updates in the compute-enabled scratchpad far outperforms the limited reordering which GPUs can perform within a vector request. Further, SPU makes efficient use of multicast for communication of gradients. SPU-dense outperforms GPU dense, because histogramming is still required even with dense datasets. On a highly dense dataset like cifar, SPU-dense outperforms SPU sparse because of the extra bandwidth consumed by sparse data structures, which is an example of the benefit of having a flexible architecture.

KSVM: SPU's network enables efficient broadcast and reduction.

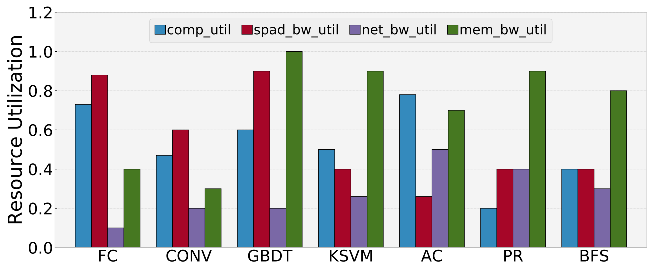


Figure 17: SPU Bottleneck on Machine Learning/Graph Workloads.

Since the dense version of KSVM is quite regular and the datasets are not sufficiently sparse, SPU-dense is generally better than its sparse version.

Arithmetic Circuits: AC heavily uses indirect memory in the DAG traversal and data-dependent control (actions depend on node type) that we support efficiently. SPU's network enables efficient communication for model parallelism, which would otherwise need to go through global memory on a GPU.

Sparse Fully Connected Layers: Figure 16 shows the per-workload performance for DNN. Using the alias-free indirection approach, we achieve high hardware utilization of the compute-enabled scratchpad. SPU outperforms GPU-sparse because it can also exploit dynamic sparsity of activations using stream-join.

Domain-accelerator Comparison: Compared to the EIE accelerator, SPU devotes more area to computation bandwidth and for providing high-throughput access to banked scratchpad, thus attaining similar performance at around half the area. Since the primary design goal of EIE is energy, it stores all weights in SRAM to save DRAM access energy; SPU trades-off lower area for higher energy.

Sparse Convolution: The best GPU algorithm was a dense winograd-based CNN. SPU is able to save computations by exploiting sparsity through the outer-product convolution using indirect memory, and dynamic resparsification.

Domain-accelerator Comparison: The performance of SPU on average is 0.76× that of SCNN. This is due to bandwidth sharing of the compute-enabled-memory scratchpad between computation and re-sparsification, whereas SCNN uses a separate non-configurable datapath. The performance difference increases for layers where re-sparsification is more intense. While comparing area is difficult, a simple scaling of SCNN's area suggests only 1.5× higher area for SPU (Section 7.5), a small price for significant generality.

SPU's Performance Bottleneck: Figure 17 shows the utilization for primary bottlenecks in SPU. Bank conflicts are the bottleneck for DNN workloads, and the effect is reduced for the fully-connected layer. GBDT is bottlenecked by scratchpad and memory bandwidth. Since AC uses model parallelism, it is bottlenecked by the network.

7.2 Performance on Graph and Databases

Here we discuss the per-workload performance results on graph (Figure 18) and database (Figure 19) domains.

Graph Workloads: SPU specializes the alias-free indirect updates to the destination vertices which would otherwise both be stalled due to load-store dependencies, and limited by inefficient bandwidth utilization due to accessing whole cache line for single accesses.

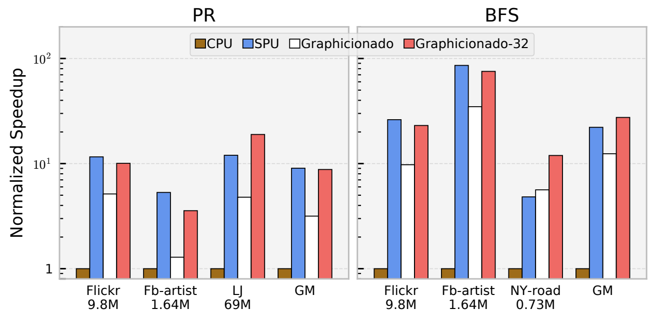


Figure 18: Performance on PR, BFS. (Edges under benchmark name)

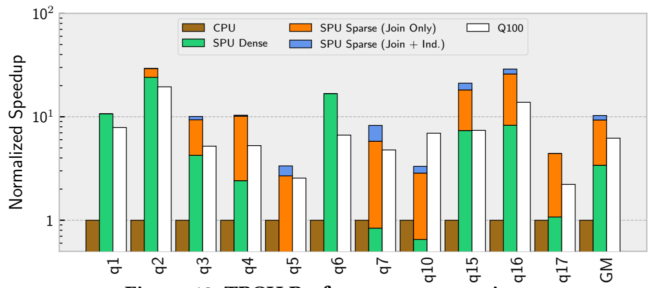


Figure 19: TPCH Performance comparison

For SPU, the network experiences high traffic because of remote indirect updates (Figure 17).

Domain-accelerator Comparison: While the designs are quite different, SPU's performance is similar to Graphicionado (8-cores) as it is exploiting similar parallelism strategies: both have a way to efficiently execute indirect memory access on a globally-addressed scratchpad. Even for the scaled-up version of Graphicionado (32-cores), it only exceeds SPU slightly for road graph due to the network contention on SPU's mesh. LiveJournal graph is an example where the graph fits in on-chip memory of Graphicionado but needs to be broken in 10 slices to be able to run on SPU. Here, SPU is 57% slower than the scaled-up Graphicionado due to both network contention and extra memory accesses for vertices which are duplicated while slicing.

TPCH Queries: Our primary goal in evaluating TPCH was to demonstrate generality. Figure 19 shows the per-query speedups of a 4-core SPU versus Q100, with three versions. SPU-dense allows only data-dependent discards (no joins or indirect memory on CGRA). Here, Joins and Sorts are performed on the control core. SPU-sparse (Join only) adds support for using the compute fabric for accelerating Sort (using merge-sort) and Join. When indirect-memory support is added, we additionally support hash-join if the smaller column fits within the scratchpad. With indirection enabled, we use a sort algorithm which applies radix-sort locally within local scratchpads, then use a merge-sort to aggregate across cores. Compared to CPU, SPU is significantly faster (10×), which is sensible given the significant data-dependence in queries, which serializes CPU execution.

Domain-accelerator Comparison: In queries which are non-sort heavy (Q1,Q2,Q6), the dense version of SPU performs adequately, and similar to the accelerator. On sort-heavy queries, stream-join within DGRA significantly reduces computation overhead, allowing SPU to catch up to Q100. Indirect access support helps to slightly improve sort's performance. Hash joins are significantly faster, but

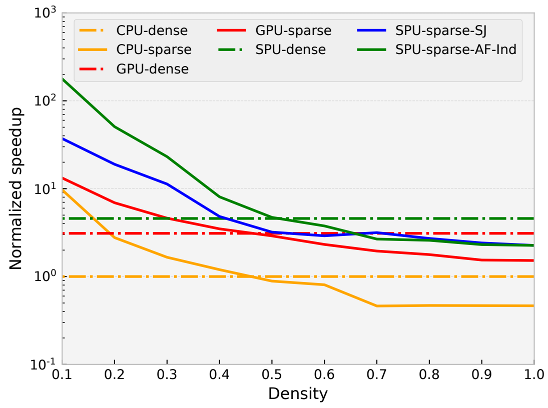


Figure 20: Performance Sensitivity (Matrix Multiply, dim: 9216×4096)

do not contribute much to speedup due to limited applicability (because of limited scratchpad size).

7.3 Sensitivity to Dataset Density

We demonstrate that it is useful to have both non-data-dependent and data-dependent support by studying performance sensitivity of a FC layer (Alex-6). Specifically we vary the dataset density with synthetic data, assuming uniform distribution of non-zero values. Figure 20 shows the performance comparison of different architectures executing matrix-vector multiply using dense and sparse data structures. At densities lower than 0.5, sparse versions perform better as they avoid superfluous computation and memory access. However, for densities greater than 0.5, extra memory accesses due to using sparse data-structures reduces the benefit.

As for the different sparse implementations for SPU, alias-free indirection outperforms stream-join at low densities, because it can avoid unnecessary index loads. Their performance converges at higher densities when this overhead is relatively less important. We observe that for the problems which can be expressed using indirection or stream-join, it is often the case that indirection works better. We believe this is the reason why recent accelerators which exploit sparsity use alias-free indirection [39, 40, 72]. However, there are kernels which might have an algorithmic advantage when expressed as a stream-join (examples in Table 1, Page). Indirection can also be inferior if there is not enough on-chip memory to hold the working-set, especially if the data is difficult to tile effectively.

7.4 Benefit of Decomposability

The speedup from decomposability is given below in Table 6.

Alg.	GBDT	Conv.	FC	KSVM	AC	BFS	PR
Speedup	2.27	2.67	2.67	2	3	2	1

Table 6: Perf. Speedup With Adding Decomposability

To explain, GBDT uses 16-bit datatypes and gets 2.27× speedup, because although we can increase the compute throughput by 4, memory bandwidth becomes a bottleneck. Conv, and FC use 16-bit datatypes, but only see a 2.6× improvement: although the multiplications could be done using subword-SIMD alone, decoding run-length encoding of indices involves control serializing computation, which needs decomposability. AC involves various bitwidths (ranging from 1-bit boolean to 32-bit fixed point) coupled with control flow. As DGRA allows bitwidths as small as 8-bit, we can merge instructions with smaller bitwidths even if their control flow is different, to achieve 3× throughput. As BFS and KSVM use 32-bit datatypes, they can be fully combined using DGRA, for 2× improvement.

		Area (mm^2)	Power (mW)
Control Core		0.041	10.1
SRAM (banked+linear)		0.196	21.2
Data Vector ports		0.012	1.4
Scratchpad Controller		0.094	18.1
DGRA	Network	0.107	130.2
	FUs (4x5)	0.124	115.9
	Total DGRA	0.230	246.1
1 SPU Total		0.573	297.0

Table 7: Area and Power breakdown for SPU (28nm)

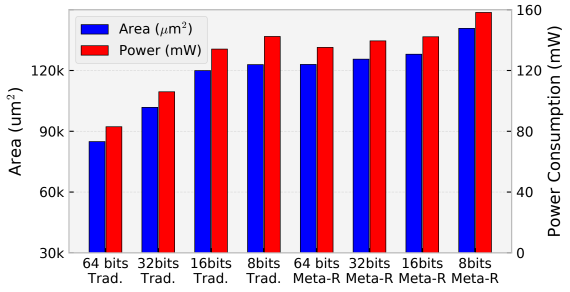


Figure 21: DGRA Area and Power Sensitivity

7.5 Area and Power

Sources of area and power: Table 7 shows the sources of area and power for SPU at 28nm. The two major sources of area are the scratchpad banks and DGRA, together occupying more than 2/3 of the total; DGRA is the major contributor to power (assuming all PEs are active).

Overhead with decomposability: Figure 21 shows the power and area cost of implementing the stream-join control and decomposability (simplest design is a standard 64-bit systolic CGRA). The stream-join control model costs about 1.7× area and power, and this is mostly due the complexity of dynamic flow control (rather than the control table). On top of this, decomposability costs around 1.2× area and power. Overall these are reasonable overheads given the performance benefits.

SPU's power and area comparison to the GPU: Estimates below show SPU has 4× lower power.

Alg.	GBDT	Conv.	FC	KSVM	AC
SPU (W)	21.16	20.73	21.18	21.43	16.48
GPU (W)	84.87	84.02	84.92	85.42	75.60

8 RELATED WORK

Table 8 gives a high-level overview of how we position SPU relative to select related work. In general, domain-specific accelerators target up to one form of data-dependence, while SPU has efficient support for both, and a domain-agnostic interface.

Architecture	Domain	Stream-Join	Alias-free Ind.	Non-data-dependent
Scnn/EIE[40, 72]	Sparse-NN	-	Very-High	-
Q100 [106]	DB	Very-High	-	-
Graphicion. [39]	Graph Alg.	-	Very-High	-
Sparse ML [59]	Sparse-MM	Very-High	-	-
PuDianNao [54]	NN	-	-	Very-High
Outersp [70]	Sparse-MM	-	Very-High	-
LSSD [66]	Agnostic	Low	Low	Very-high
Plasticine [76]	Agnostic	-	High	Very-high
SPU (ours)	**Agnostic**	**Very-High**	**Very-High**	**Very-High**
VT [52, 53]	Agnostic	High	High	High
Dataflow [16, 93]	Agnostic	Medium	Medium	High
GPU	Agnostic	Low	Medium	High
CPU	Agnostic	Low	Medium	Medium

Table 8: Analysis of Related Works (roughly least to most general)

Domain-specific Accelerators: Pudiannao [54] is an accelerator for multiple dense ML kernels. Several designs specialize sparse-matrix computations, including many for FPGAs [34, 38, 111]. Nurvitadhi et al. propose a sparse-matrix accelerator specialized for SVM [68]. Mishra et al. develop an in-core accelerator for sparse matrices, and demonstrate generality to many ML workloads [59].

From the database accelerator domain, we draw inspiration from Q100's ability to perform pipelined join and filtering [106] to create our general purpose stream-join model. DB-Mesh [17] is a systolic-style homogeneous array for executing nested-loop joins. WIDX [50] is database index accelerator focusing on indirect memory access. UDP [32] targets encoding and compression workloads, which both express data-dependence.

Domain-agnostic Vector Accelerators: Vector-threads (VT) architectures [52, 53, 82] have a flexible SIMD/MIMD execution model, where vector lanes can be decomposed into independent lanes to enable parallel execution for data-dependent codes. (GANAX [107] applies some of the same principles, but is specialized to ML). VT does not have spatial abstractions for computation, which SPU uses to expose an extra dimension of parallelism: pipeline parallelism. For example on VT, stream-joins would not execute at one item per cycle due to instruction overhead, but these computations can be pipelined on SPU. There are other less fundamental differences like SPU's support for programmer-controlled scratchpads with global address space.

Domain-agnostic Spatial Architectures: LSSD is a domain-agnostic multi-tile accelerator with CGRAs and simple control cores [66]. However, it lacks support for data-dependent control or memory, so is far less general.

General spatial-dataflow architectures (eg. WaveScalar [93], TRIPS [16]) can perform stream joins, but at much lower throughput (due to control dependence loop, see Figure 4). Triggered instructions [71] and Intel's CSA [104] can perform pipelined stream-joins, but require much more complex non-systolic PEs (>3× higher area [81]), and are also not capable of decomposability. They are also not specialized for alias-free indirection, and require parallel dependence checking. In concurrent work, Master of None [55] proposes a programmable systolic-style homogeneous reconfigurable array that can execute general database queries, as well as pipelined stream-join through a dedicated control network.

Plasticine [76, 77] is a tiled spatial architecture, composed of SIMD compute tiles and scratchpad tiles. Plasticine does not support stream-join dataflow, so would not be able to efficiently execute algorithms with this form of control-dependence. Plasticine also does not have compute-enabled globally-addressed scratchpads for high-bandwidth atomic update and flexible data sharing. Plasticine uses a parallel pattern programming interface [51, 75], while SPU provides a general purpose dataflow ISA, based on stream-dataflow [65]. While lower-level, SPU's ISA can more flexibly implement various computation/communication patterns. Recent work demonstrates efficient hash-joins for Plasticine [89]; such techniques could improve the performance and applicability of hash-joins in SPU.

Finally, lower precision control divergence is not supported in these architectures; SPU has the strongest support for arbitrary datatypes through its decomposable CGRA and memory. Note that while decomposability has been applied in other contexts, e.g. supporting multiple datatypes on a domain-specific CGRA [46, 86], we believe we are the first to apply decomposability to preserve independent flow-control.

General Purpose Processor Specialization: The decoupled-stream ISA [101] allows expression of decoupled indirect streams for general-purpose ISAs. It also enables decoupling of memory from control flow in stream-joins. However, it does not specialize for the stream-join computation or high-bandwidth indirect access.

9 CONCLUSION

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

More broadly, this work shows that data-dependence does not necessitate fixed-function hardware or massive arrays of inorder processors – many algorithms are fundamentally data-parallel and can be specialized provided the right architecture abstractions. We believe that an important implication of this work could be to inspire the communities in different domains (eg. machine learning and databases) to explore the use of less regular data-structures and novel algorithms with codesigned hardware.

10 ACKNOWLEDGMENTS

We would like to thank Guy Van den Broeck and Arthur Choi for their insights and help with Arithmetic Circuits workloads. We would also like to thank David Ott and Pratap Subrahmanyam for their thoughtful conversations on the nature of irregularity and data-dependence. This work was supported by an NSF CAREER award CCF-1751400, as well as gift funding from VMware.

REFERENCES

[1] [n. d.]. Intel Math Kernel library. [Online]. Available: http://software.intel.com/en-us/intel-mkl.
[2] [n. d.]. XLA: Domain-specific compiler for linear algebra to optimizes tensorflow computations. https://www.tensorflow.org/performance/xla
[3] 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice*

of Parallel Programming (PPoPP '13). ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[4] 2016. Intel(R) Math Kernel Library for Deep Neural Networks. "https://github.com/01org/mkl-dnn".

[5] 2017. Everything You Always Wanted to Know About Multicore Graph Processing but Were Afraid to Ask. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 631–643. http://dl.acm.org/citation.cfm?id=3154690.3154750

[6] 2017. Neural Network Distiller by Intel AI Lab. "https://github.com/NervanaSystems/distiller".

[7] T. M. Aamodt, W. W. L. Fung, T. G. Rogers, and M. Martonosi. 2018. *General-Purpose Graphics Processor Architecture*. Morgan & Claypool. https://ieeexplore.ieee.org/document/8363085

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[9] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 105–117. https://doi.org/10.1145/2749469.2750386

[10] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 1–13. https://doi.org/10.1109/ISCA.2016.11

[11] Krste Asanović and David A Patterson. 2014. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[12] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. 2011. GPU acceleration for support vector machines. In *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*.

[13] 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. https://doi.org/10.1145/2228360.2228584

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011).

[15] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[16] 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. https://doi.org/10.1109/MC.2004.65

[17] Bingyi Cao, Kenneth A. Ross, Stephen A. Edwards, and Martha A. Kim. 2017. Deadlock-free joins in DB-mesh, an asynchronous systolic array accelerator. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*. 5:1–5:8. https://doi.org/10.1145/3076113.3076118

[18] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)* 2, 3 (2011), 27.

[19] 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. https://doi.org/10.1145/2541940.2541967

[20] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.

[21] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[22] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 609–622. https://doi.org/10.1109/MICRO.2014.58

[23] Yu-Ting Chen, Jason Cong, Jie Lei, and Peng Wei. 2015. A novel high-throughput acceleration engine for read alignment. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 199–202.

[24] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[25] 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

[26] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 9–16.

[27] NVIDIA Corp. [n. d.]. GeForce GTX 1080 Whitepaper.

[28] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (April 2019), 640–653. https://doi.org/10.1109/TCAD.2018.2821565

[29] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104. https://doi.org/10.1145/2749469.2750389

[30] A. Duran and M. Klemm. 2012. The Intel Many Integrated Core Architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*.

[31] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 533–545. https://doi.org/10.1145/2830772.2830809

[32] Yuanwei Fang, Chen Zou, Aaron Elmore, and Andrew Chien. 2016. UDP: A Programmable Accelerator for Extract-Transform-Load Workloads and More. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[33] J. Fowers, J. Y. Kim, D. Burger, and S. Hauck. 2015. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 52–59. https://doi.org/10.1109/FCCM.2015.46

[34] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. 2014. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 36–43.

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

[36] Daichi Fujiki, Aran Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A genome sequencing accelerator. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 69–82.

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

[38] Paul Grigoraş, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. 2016. Optimising Sparse Matrix Vector multiplication for large scale FEM problems on FPGA. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 1–9.

[39] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[40] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[41] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 1135–1143. http://dl.acm.org/citation.cfm?id=2969239.2969366

[42] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro,

et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 620–629.

[43] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 674–687. https://doi.org/10.1109/ISCA.2018.00062

[44] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. AyguadÃĬ, and M. Valero. 2014. Advanced Pattern based Memory Controller for FPGA based HPC applications. In *2014 International Conference on High Performance Computing Simulation (HPCS)*. 287–294. https://doi.org/10.1109/HPCSim.2014.6903697

[45] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[46] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[47] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783722

[48] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. 2015. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–13.

[49] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 3146–3154. http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf

[50] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO*.

[51] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[52] 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–. https://doi.org/10.1145/1028176.1006736

[53] 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. https://doi.org/10.1145/2000064.2000080

[54] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. In *ASPLOS*.

[55] Andrea Lottarini, João P. Cerqueira, Thomas J. Repetti, Stephen A. Edwards, Kenneth A. Ross, Mingoo Seok, and Martha A. Kim. 2019. Master of None Acceleration: A Comparison of Accelerator Architectures for Analytical Query Processing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 762–773. https://doi.org/10.1145/3307650.3322220

[56] L. McMurchie and C. Ebeling. 1995. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Third International ACM Symposium on Field-Programmable Gate Arrays*. 111–117. https://doi.org/10.1109/FPGA.1995.242049

[57] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*. Springer, 61–70.

[58] X. Mei and X. Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (Jan 2017), 72–86. https://doi.org/10.1109/TPDS.2016.2549523

[59] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr. 2017. Fine-grained accelerators for sparse machine learning workloads. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 635–640. https://doi.org/10.1109/ASPDAC.2017.7858395

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

[61] M Naumov, LS Chien, P Vandermersch, and U Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

[62] Chris Nicol. 2017. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing. *WaveComputing WhitePaper* (2017).

[63] Tony Nowatzki. 2017. Stream-dataflow public release. *URL: https://github.com/PolyArch/stream-dataflow* (2017).

[64] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid Optimization/Heuristic Instruction Scheduling for Programmable Accelerator Codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/3243176.3243212

[65] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 416–429. https://doi.org/10.1145/3079856.3080255

[66] 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. https://doi.org/10.1109/HPCA.2016.7446051

[67] 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. https://doi.org/10.1145/2491956.2462163

[68] E. Nurvitadhi, A. Mishra, and D. Marr. 2015. A sparse matrix vector multiply accelerator for support vector machine. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 109–116. https://doi.org/10.1109/CASES.2015.7324551

[69] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE: THE WORLD'S MOST ADVANCED DATA CENTER GPU. *NVIDIA WhitePaper* (2017). http://www.nvidia.com/object/volta-architecture-whitepaper.html

[70] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. https://doi.org/10.1109/HPCA.2018.00067

[71] 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. https://doi.org/10.1145/2485922.2485935

[72] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/3079856.3080254

[73] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. 2008. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08)*. 166–176. https://doi.org/10.1145/1454115.1454140

[74] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/2594291.2594339

[75] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 651–665.

https://doi.org/10.1145/2872362.2872415

[76] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[77] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. 2018. Plasticine: A Reconfigurable Accelerator for Parallel Patterns. *IEEE Micro* 38, 3 (May 2018), 20–31. https://doi.org/10.1109/MM.2018.032271058

[78] Benoıt Pradelle, Benoıt Meister, Muthu Baskaran, Jonathan Springer, and Richard Lethin. [n. d.]. Polyhedral Optimization of TensorFlow Computation Graphs.

[79] 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. https://doi.org/10.1109/MM.2015.42

[80] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. HernÃąndez-Lobato, G. Y. Wei, and D. Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 267–278. https://doi.org/10.1109/ISCA.2016.32

[81] Thomas J. Repetti, João P. Cerqueira, Martha A. Kim, and Mingoo Seok. 2017. Pipelining a Triggered Processing Element. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, New York, NY, USA, 96–108. https://doi.org/10.1145/3123939.3124551

[82] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. 2006. Vector Lane Threading. In *2006 International Conference on Parallel Processing (ICPP'06)*. 55–64. https://doi.org/10.1109/ICPP.2006.74

[83] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. 1998. A Bandwidth-efficient Architecture for Media Processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 31)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3–13. http://dl.acm.org/citation.cfm?id=290940.290946

[84] Alec Roelke and Mircea R. Stan. 2017. RISC5: Implementing the RISC-V ISA in gem5. (2017).

[85] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26. https://doi.org/10.1109/ISCA.2016.12

[86] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[87] Yujia Shen, Arthur Choi, and Adnan Darwiche. 2016. Tractable Operations for Arithmetic Circuits of Probabilistic Models. In *Advances in Neural Information Processing Systems 29 (NIPS)*.

[88] 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. https://doi.org/10.1109/12.859540

[89] Rekha Singhal, Yaqi Zhang, Jeffrey D Ullman, Raghu Prabhakar, and Kunle Olukotun. 2019. Efficient Multiway Hash Join on Reconfigurable Hardware. *Technology Conference on Performance Evaluation & Benchmarking (TPCTC)* (2019).

[90] Avinash Sodani. 2015. Knights landing (knl): 2nd generation intel® xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 1–24.

[91] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[92] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.

[93] 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–. https://doi.org/10.1109/MICRO.2003.1253203

[94] Tuan Ta, Lin Cheng, and Christopher Batten. 2018. Simulating Multi-Core RISC-V Systems in gem5. (2018).

[95] Yatish Turakhia, Gill Bejerano, and William J Dally. 2018. Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 199–213.

[96] Yatish Turakhia, Sneha D Goenka, Gill Bejerano, and William J Dally. 2019. Darwin-WGA: A co-processor provides increased sensitivity in whole genome alignments with high speedup. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 359–372.

[97] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[98] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. 2017. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/3079856.3080244

[99] 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. https://doi.org/10.1109/ISCA.2014.6853234

[100] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An overview of micron's automata processor. In *2016 international conference on hardware/software codesign and system synthesis (CODES+ ISSS)*. IEEE, 1–3.

[101] Zhengrong Wang and Tony Nowatzki. 2019. Stream-based Memory Access Specialization for General Purpose Processors. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. ACM, New York, NY, USA, 736–749. https://doi.org/10.1145/3307650.3322229

[102] 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. https://doi.org/10.1109/FPL.2015.7294017

[103] NVIDIA Whitepaper. 2019. Cuda C Best Practices Guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[104] WikiChip. 2019. Configurable Spatial Accelerator. https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator.

[105] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2013. Navigating Big Data with High-throughput, Energy-efficient Data Partitioning. In *ISCA*.

[106] 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. https://doi.org/10.1145/2541940.2541961

[107] Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. 2018. Ganax: A unified mimd-simd acceleration for generative adversarial networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 650–661.

[108] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.

[109] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable interconnects for reconfigurable spatial architectures. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 615–628. https://doi.org/10.1145/3307650.3322249

[110] Y. Zhu and V. J. Reddi. 2014. WebCore: Architectural support for mobile Web browsing. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 541–552. https://doi.org/10.1109/ISCA.2014.6853239

[111] Ling Zhuo and Viktor K Prasanna. 2005. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM, 63–74.