Towards General-Purpose Acceleration by Exploiting Common Data-Dependence Forms

Vidushi Dadu, Jian Weng, Sihao Liu, Tony Nowatzki

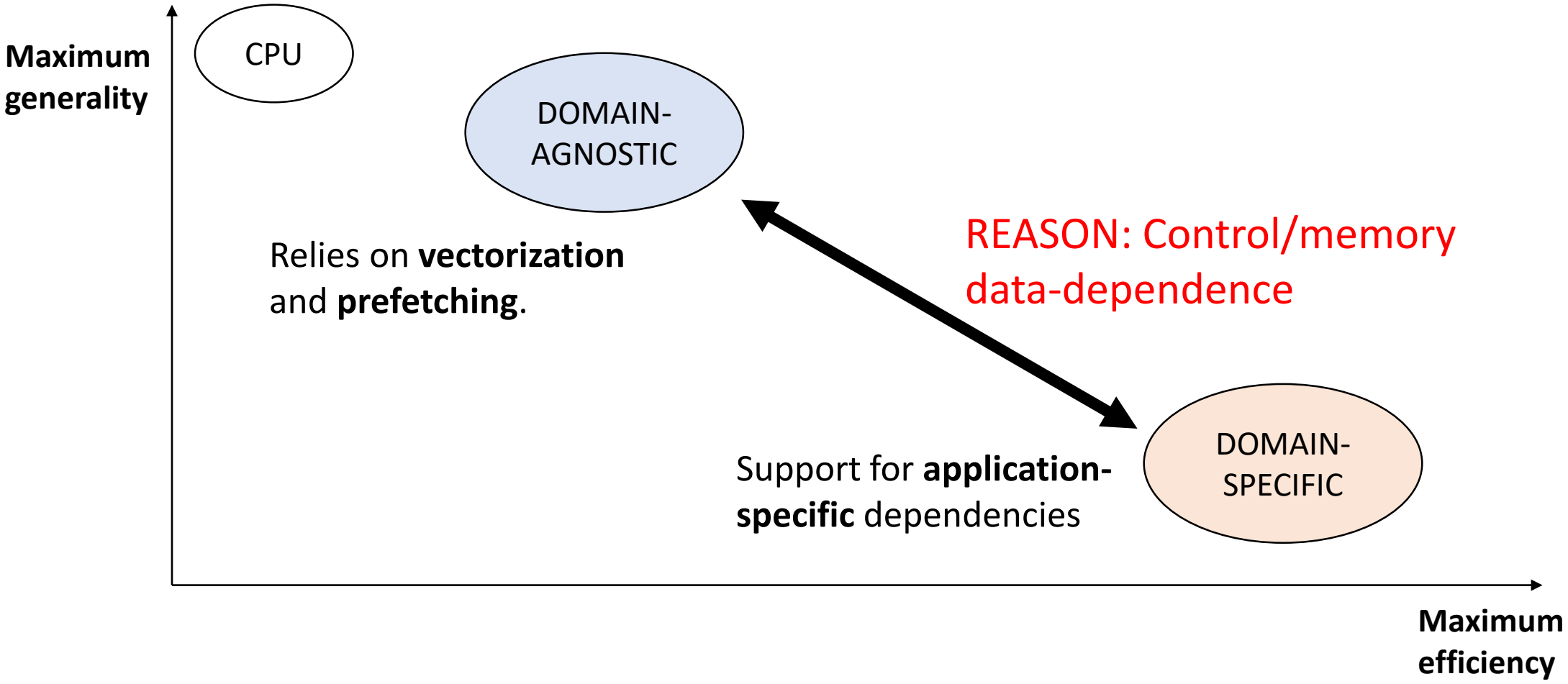
UCLA

MICRO 2019

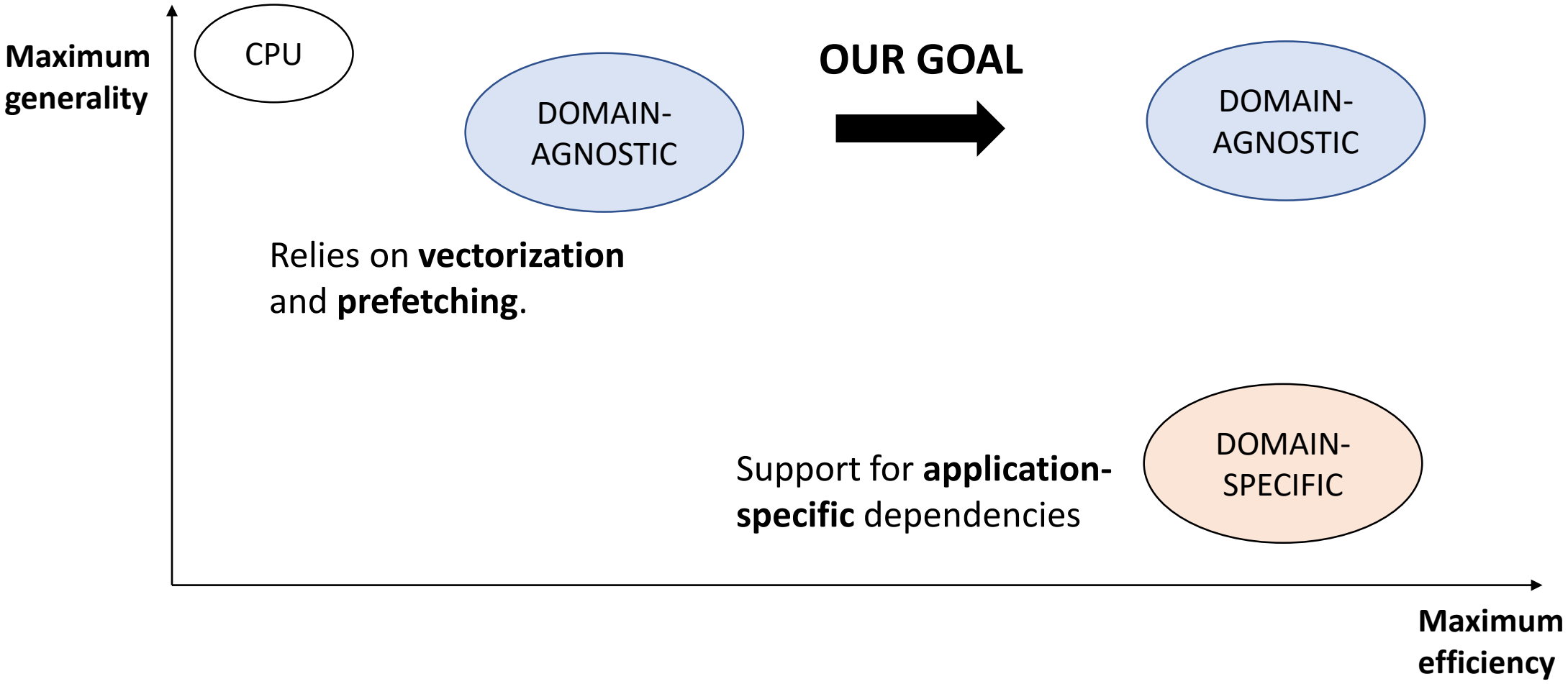
UCLA

POLY ARCH

Challenging trade-off in domain-specific and domain-agnostic acceleration



Challenging trade-off in domain-specific and domain-agnostic acceleration

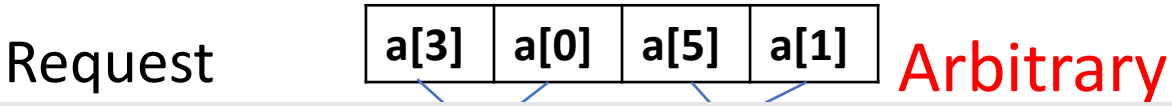


Programmable Accelerators (eg. GPUs) Fail to Handle *Arbitrary* Control/Memory Dependence

Control Dependence

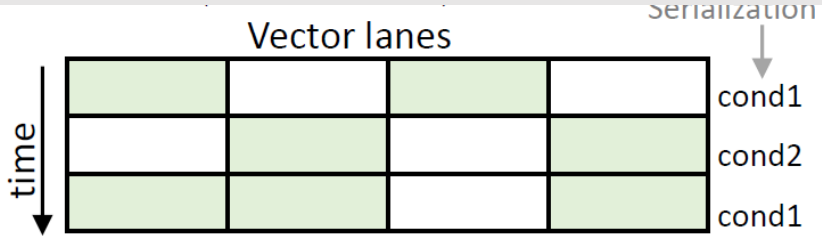
Arbitrary code

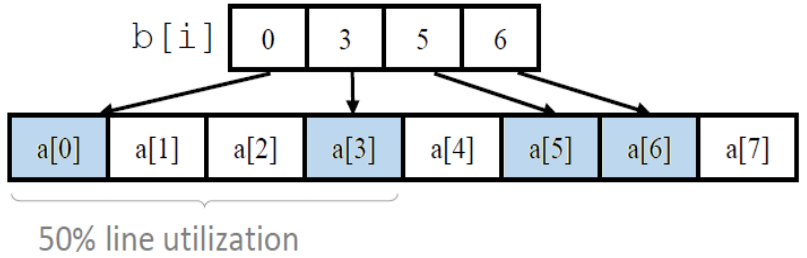
Memory Dependence



Arbitrary

Insight: *Restricted* control and memory dependence is sufficient for *many* data-processing algorithms.



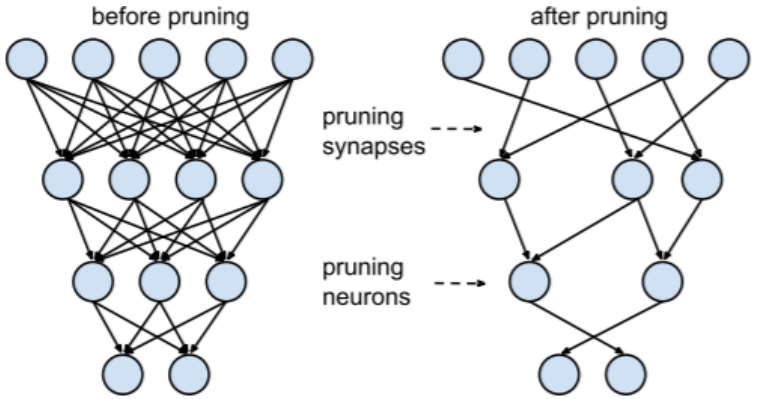


Outline

- Irregularity is ubiquitous
  - *Sufficient* and *Exploitable* forms of Control and Memory dependence
  - Example Workload: Matrix Multiply
- Exploiting data-dependence with SPU accelerator
  - uArch: Stream-join Dataflow & Compute-enabled Scratchpad
  - SPU Multicore Design
- Evaluating SPU
- Conclusion

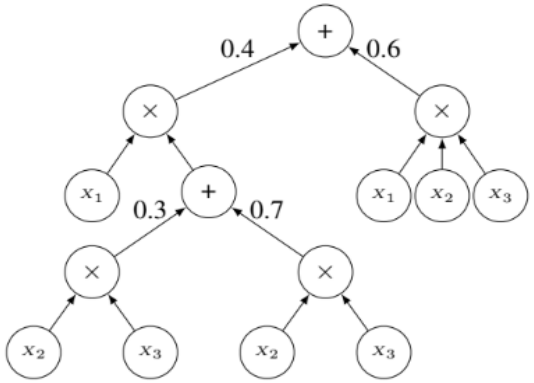
Irregularity is Ubiquitous

Sparsity within dataset (Machine Learning)



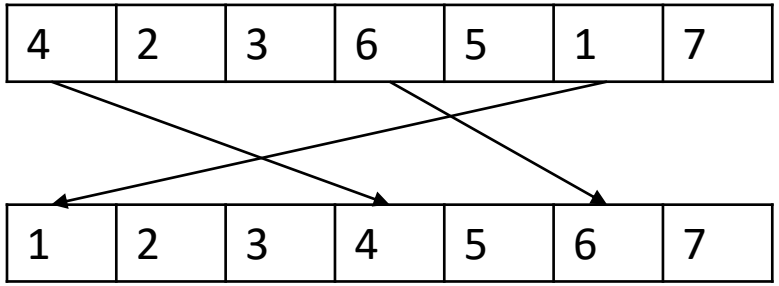
Pruned Neural Network

Data-structures representing relationships (Graphs)

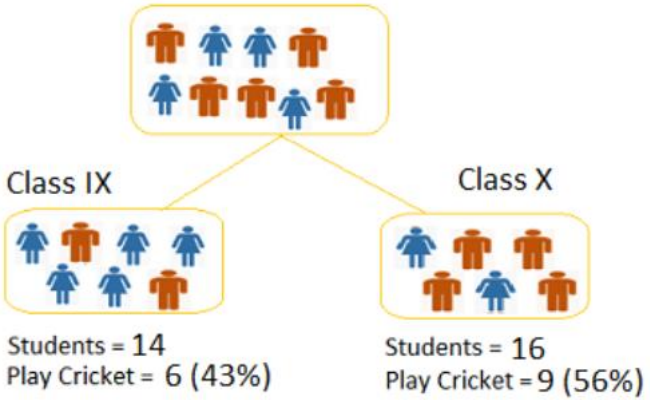


Bayesian Networks

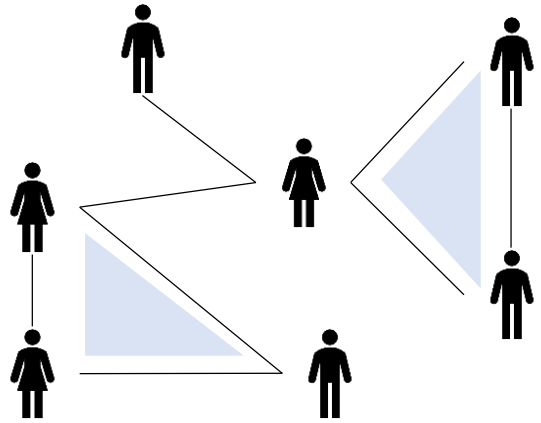
Purpose to reorder data (Databases)



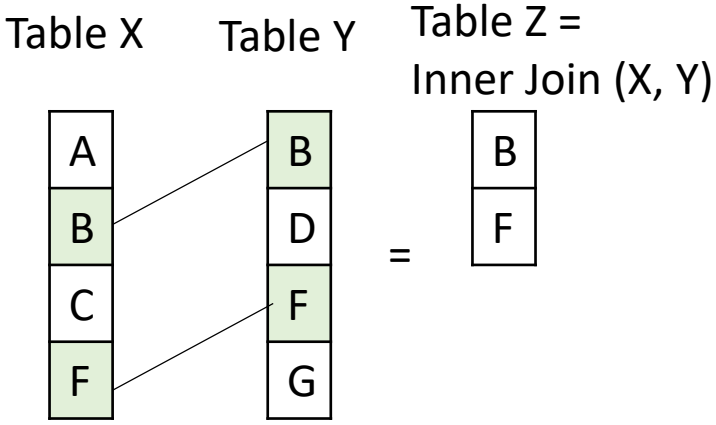
Sorting



Decision tree building



Triangle Counting

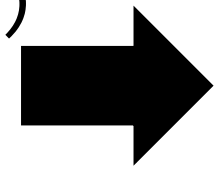


Database Join

Irregularity Stems from Data-dependence

Data-dependent aspects of execution

1. Control flow: `if(`*f*`(a[i]))`
2. Memory Access: `b[a[i]]`



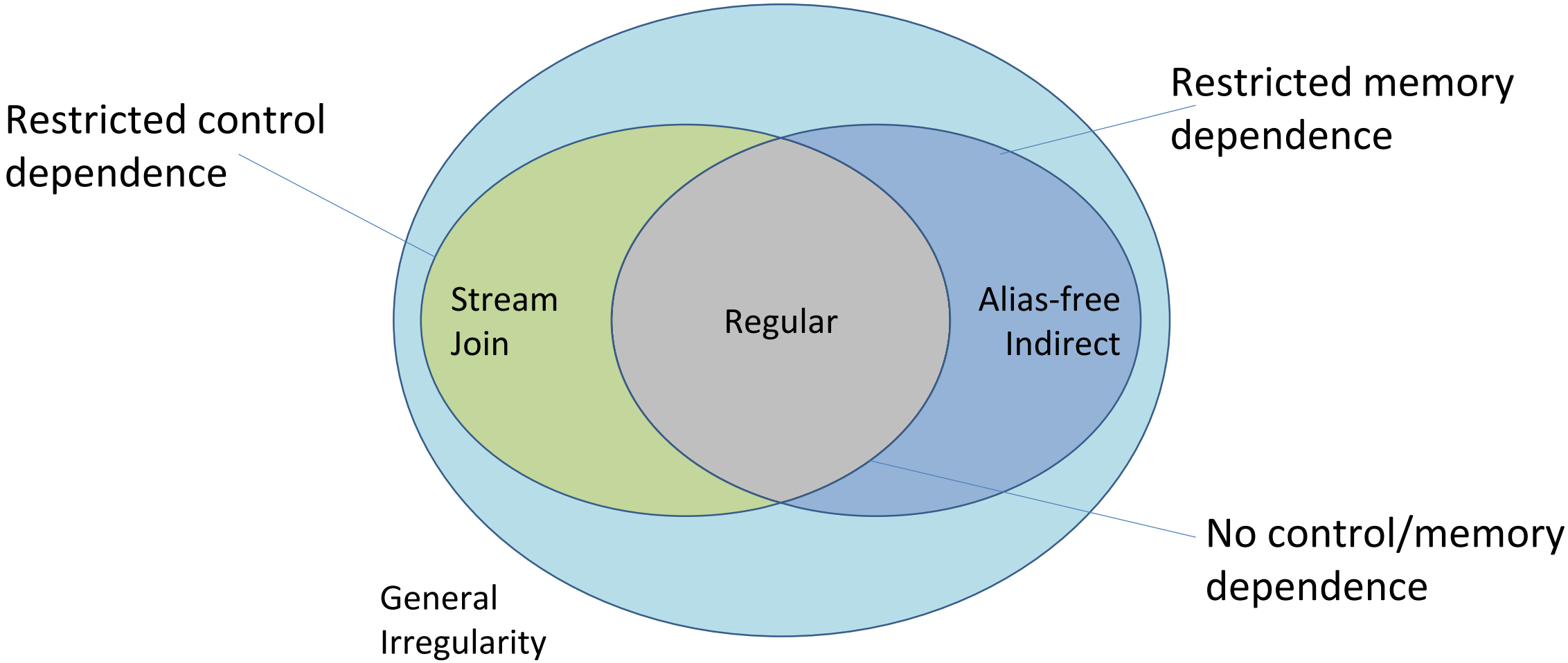
Restricted Control flow: Stream-Join

Restricted Memory Access: Alias-Free Indirection

Main-Insight: There are narrow forms of dependence which are:

- *Sufficient* to express many algorithms (from **ML, graph analytics, databases**)
- *Exploitable* with minimal hardware overhead

Algorithm Classification

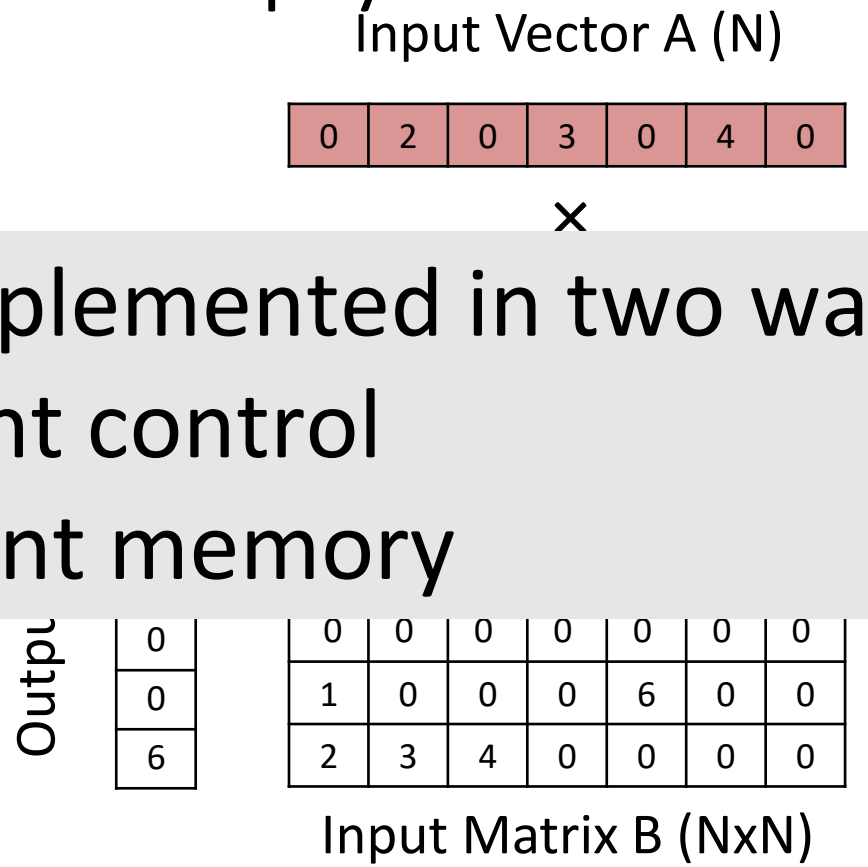


Regular Example: Dense Matrix Multiply

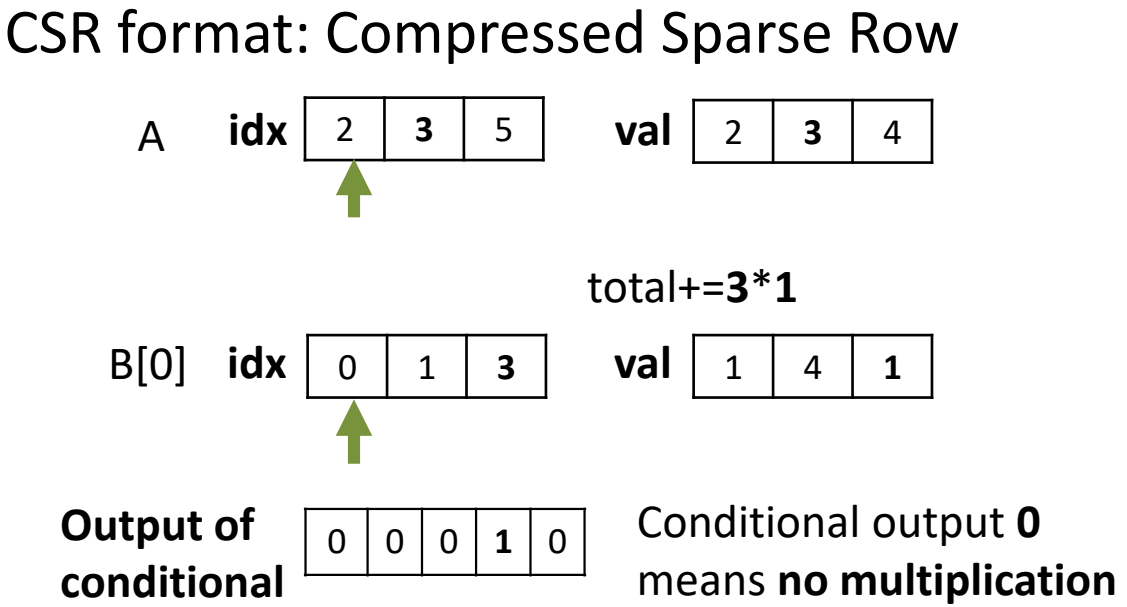
- No data-dependence;

Sparse matrix-multiply can be implemented in two ways:

1. Inner product: Data-dependent control
2. Outer product: Data-dependent memory



Sparse Inner Product Multiply (stream-join)



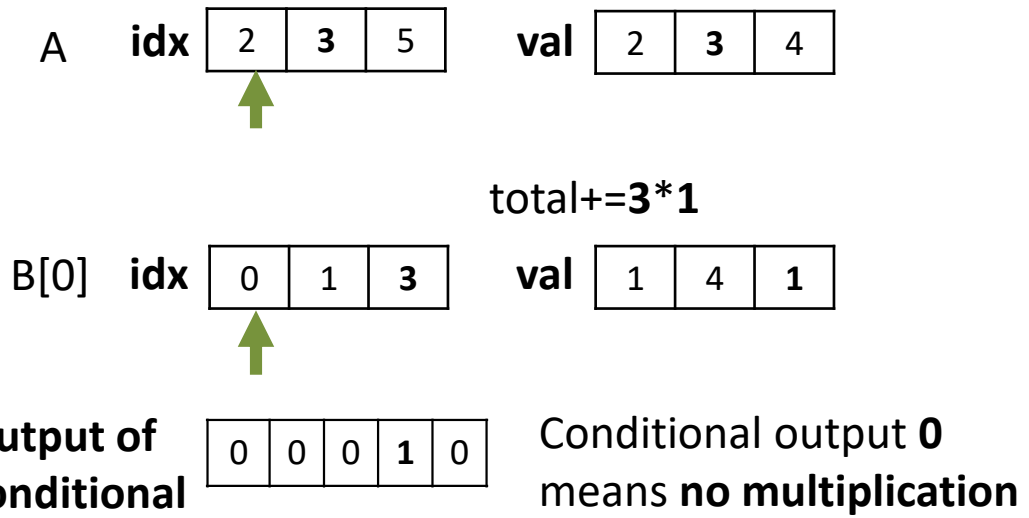
- Known memory access pattern, but unpredictability in control

Sparse Inner Product Multiply (stream-join)

```
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++
  ...
```

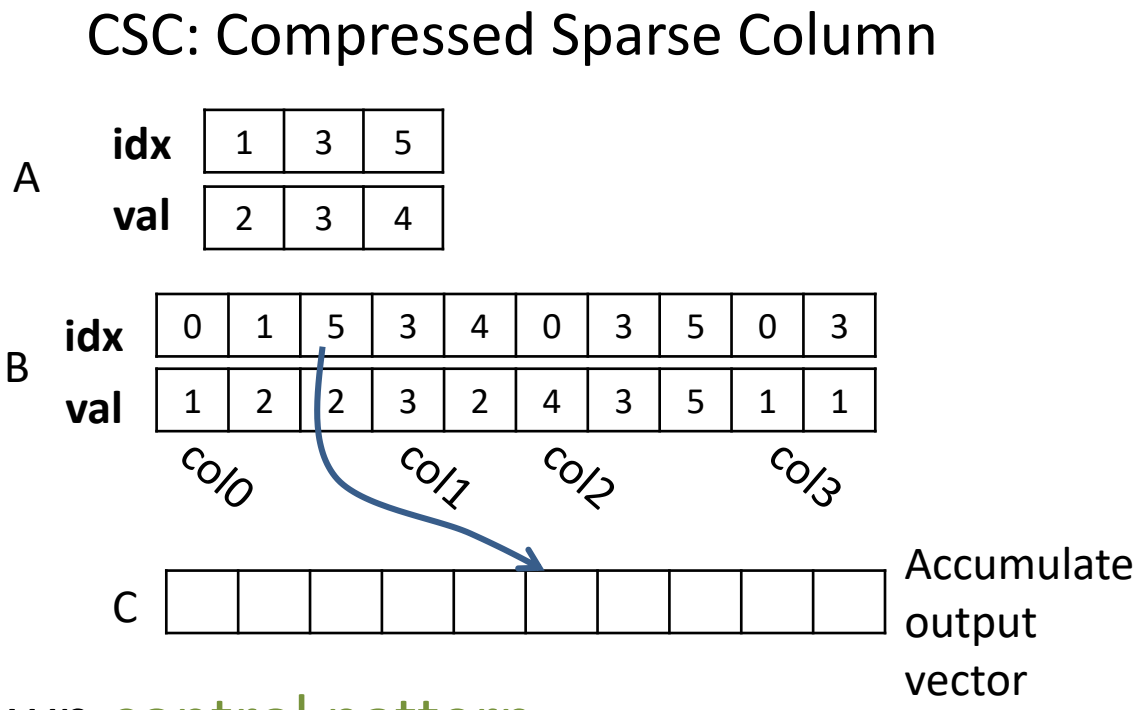
Indicative of Stream-Join

CSR format: Compressed Sparse Row



- Known memory access pattern, but unpredictability in control
- Stream Join:
  - Memory read can be independent of data*
  - Order that we consume *streams* of data is data-dependent

Sparse Outer Product Multiply (Alias-free Indirection)

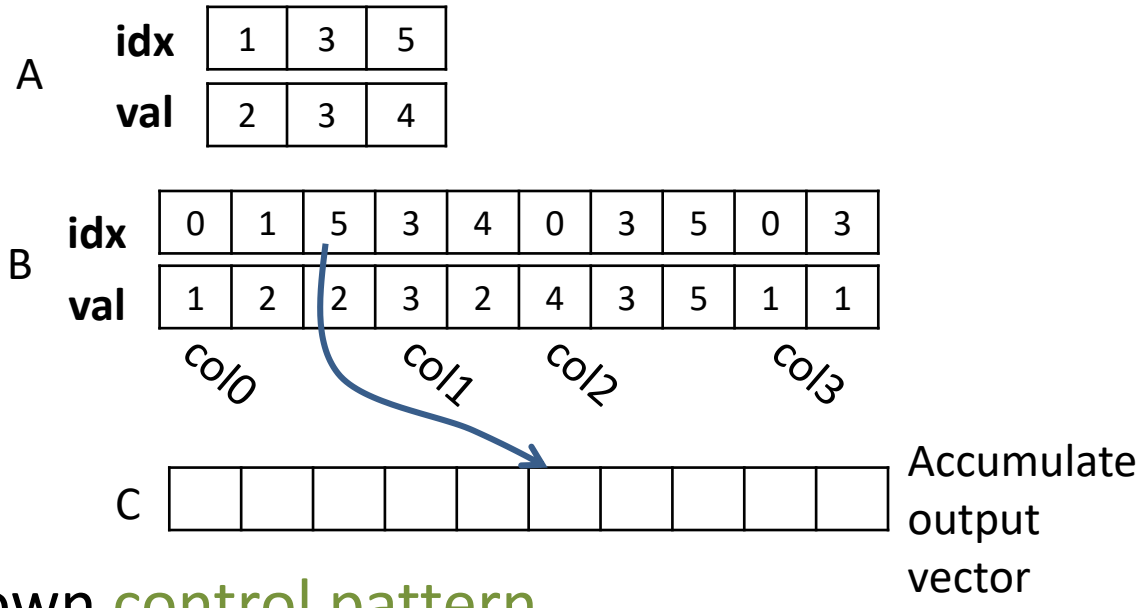


- High memory unpredictability, but known control pattern
- No unknown dependencies (only **atomic** updates: `out[i]=out[i]+prod[i]`)

Sparse Outer Product Multiply (Alias-free Indirection)

```
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++                          Indirection
```

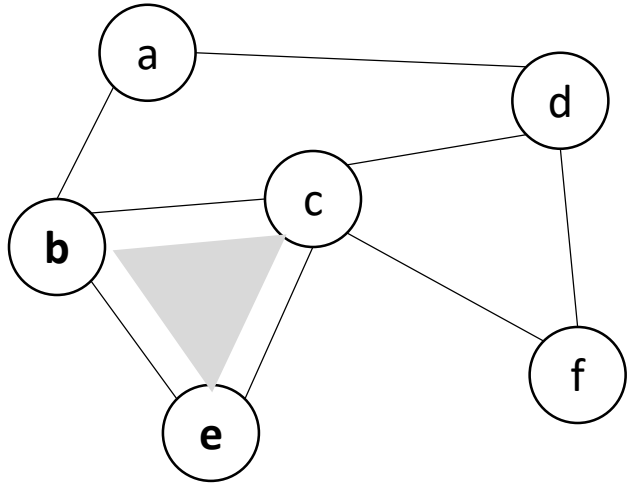
CSC: Compressed Sparse Column

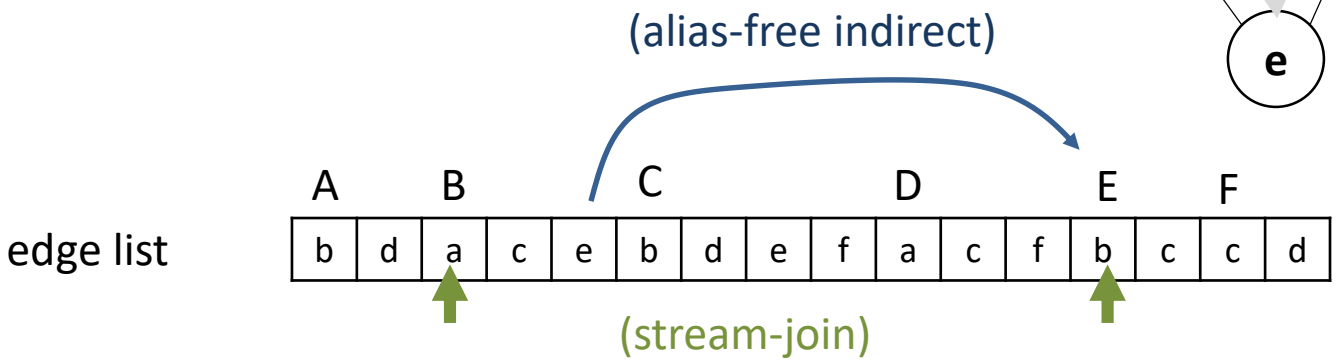


- High memory unpredictability, but known control pattern
- No unknown dependencies (only **atomic** updates: `out[i]=out[i]+prod[i]`)
- Alias-free Indirect:
  - Produce addresses depending on other data
  - Memory dependences, but no unknown (data-dependent) aliases

Graph Mining (e.g. Triangle Counting)

- For every pair of connected nodes, find if they have a common neighbor



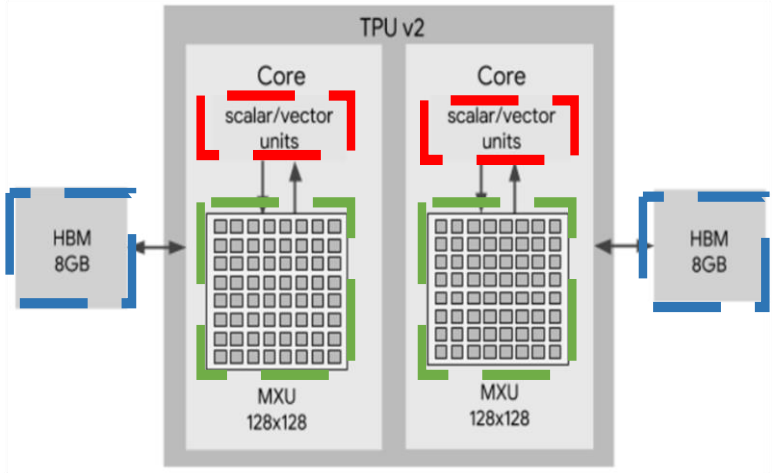


		Stream Join (irreg. control)	Alias-free Indirection (irregular memory)
Machine Learning	Neural Net (FC + Conv)	Inner Product Mult.	Outer Product Mult.
	Supp. Vector (SVM)	""	""
	Decision Trees (GBDT)	Sparse data access	+ Histogramming
	Bayesian Networks	Condition on node type	+ DAG Access
Graph	Page Rank & BFS	Sparse join of active list	+ Indirect acc. for edges
	Triangle Counting	Find common neighbor edges	+ Indirect acc. for edges
Databases	Join (inner)	Sort-Join	Hash-Join
	Sort	Merge-Sort	Radix-Sort
	Filter	Generate Filtered Col.	Generate Column Ind.

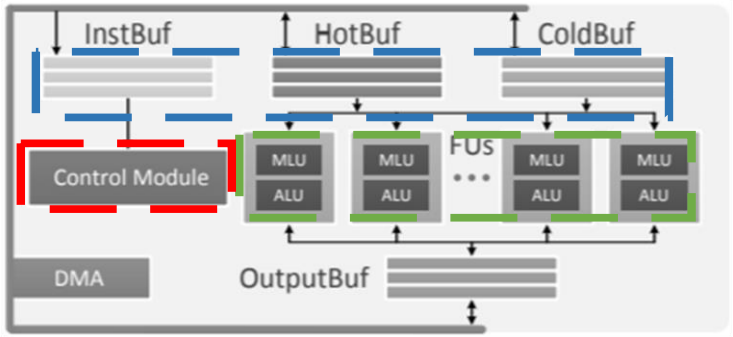
Outline

- Irregularity is ubiquitous
  - *Sufficient* and *Exploitable* forms of Control and Memory dependence
  - Example Workload: Matrix Multiply
- **Exploiting data-dependence with SPU accelerator**
  - **uArch: Stream-join Dataflow & Compute-enabled Scratchpad**
  - **SPU Multicore Design**
- Evaluating SPU
- Conclusion

Approach: Start with a Dense Programmable Accelerator



Google TPU v2
ISCA'17



PuDianNao (ASPLOS'15)

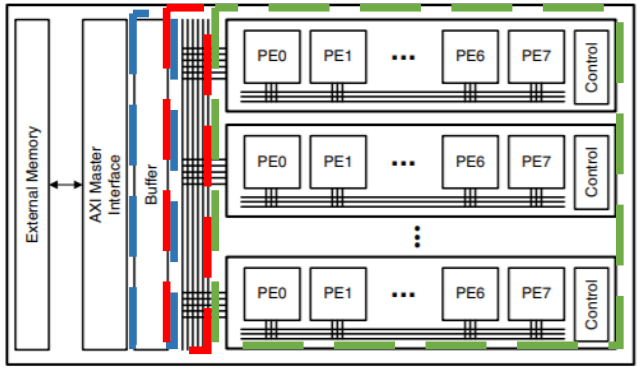
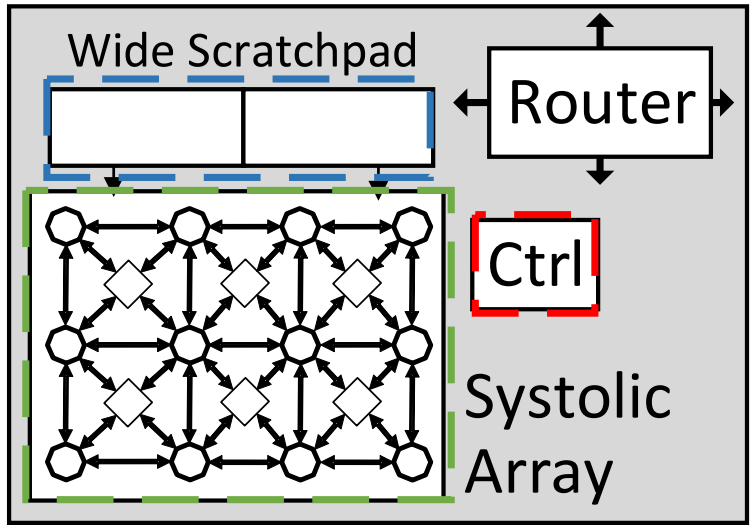


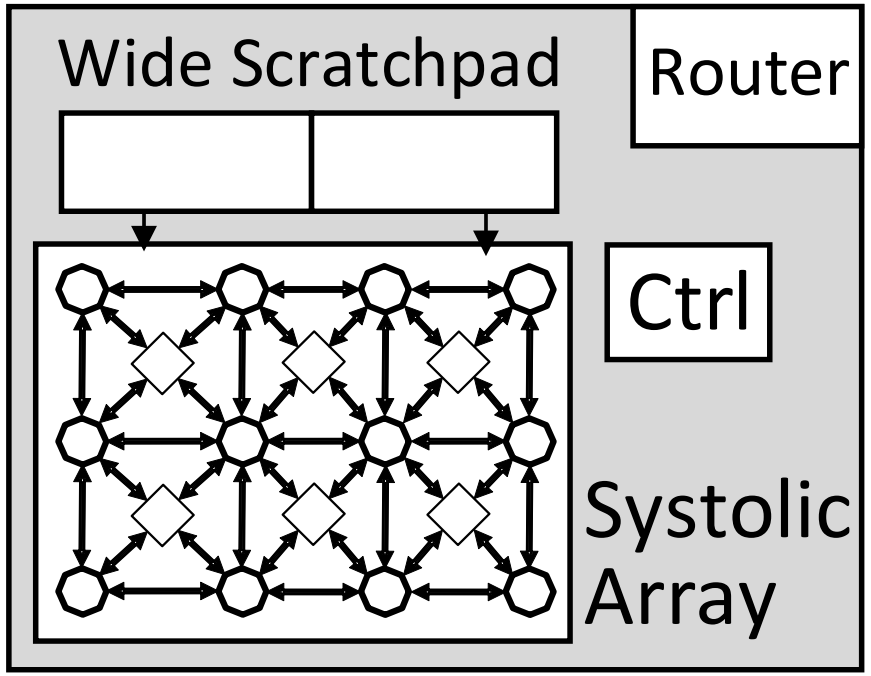
Tabla (HPCA'16)



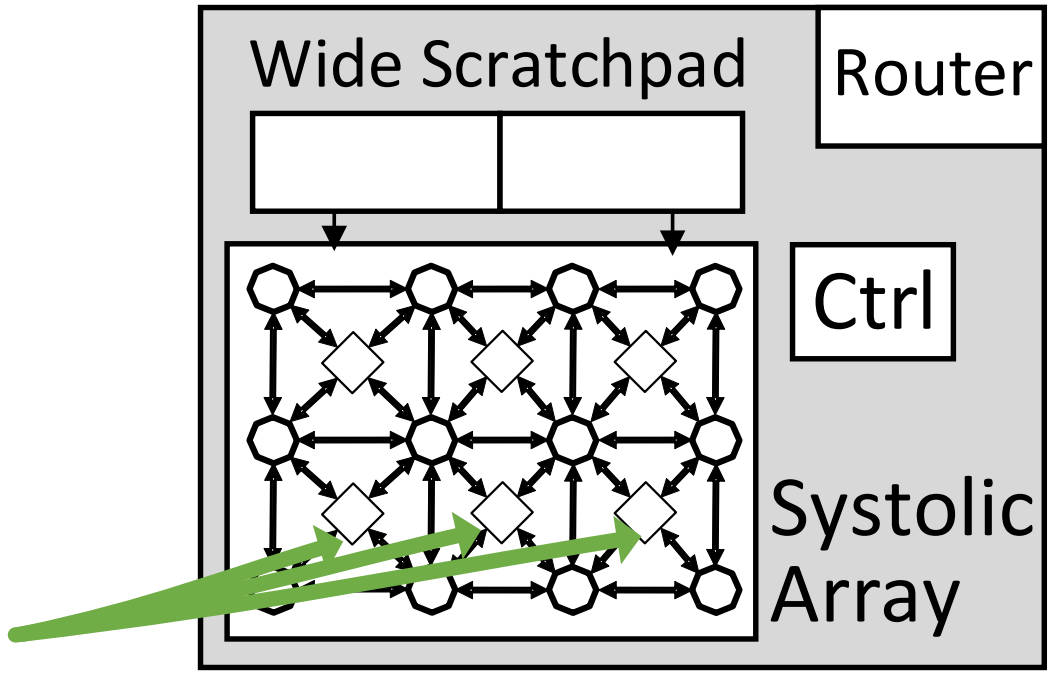
Stereotypical Dense Accelerator Core

Systolic Array
Wide Scratchpad
Control

Approach: Start with a Dense Programmable Accelerator



Approach: Start with a Dense Programmable Accelerator

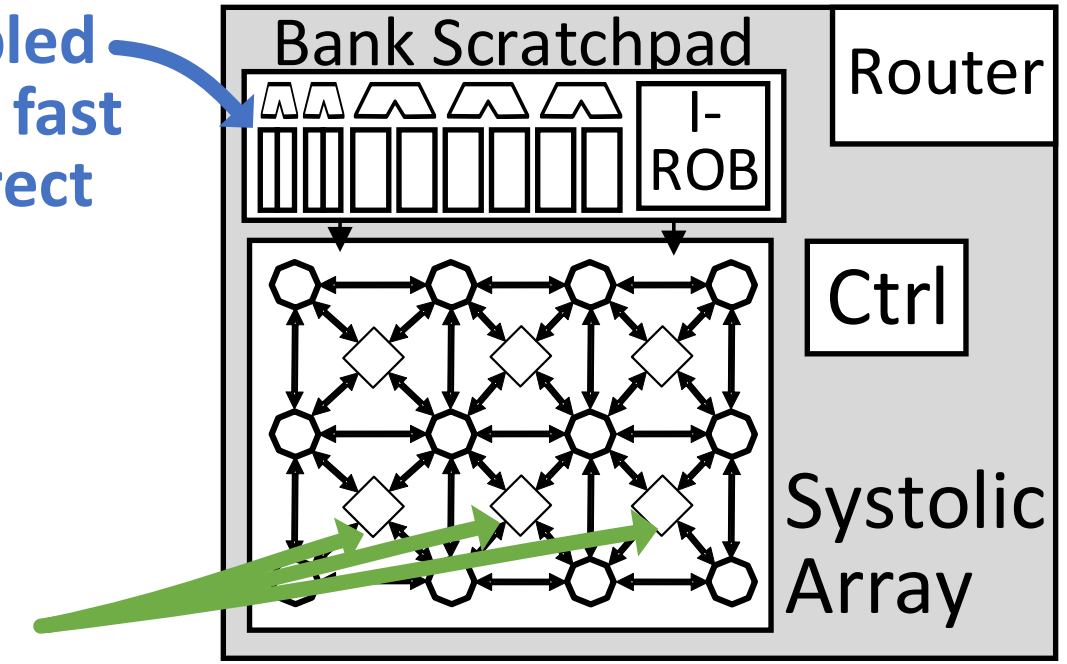


Systolic array supporting stream-join control

Approach: Start with a Dense Programmable Accelerator

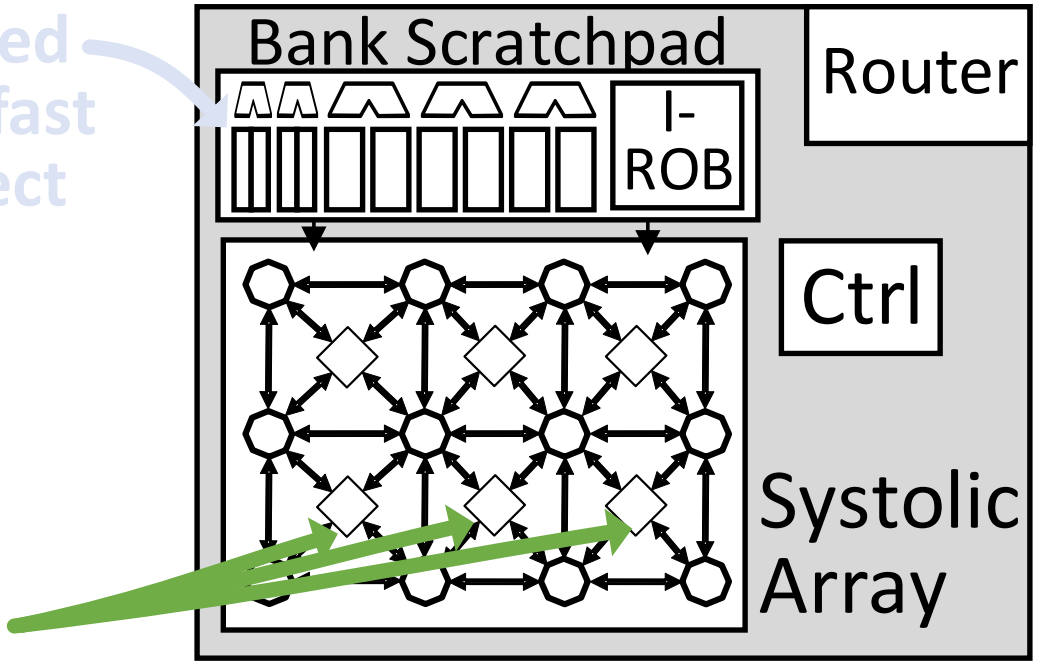
Compute-Enabled Scratchpad for fast Alias-free indirect access

Systolic array supporting stream-join control



Specializing for Stream Join

Compute-Enabled Scratchpad for fast Alias-free indirect access



Systolic array supporting stream-join control

Novel Dataflow for Stream Join

Sparse MM Example

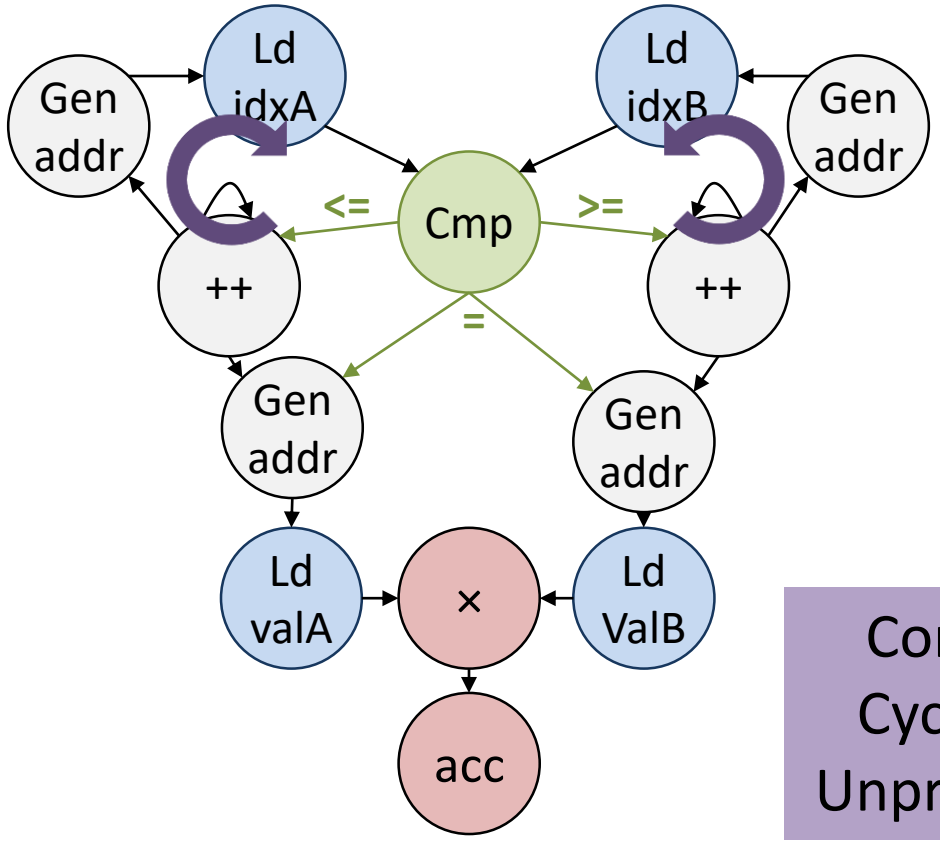
A

idx	2	**3**	5
val	2	3	4

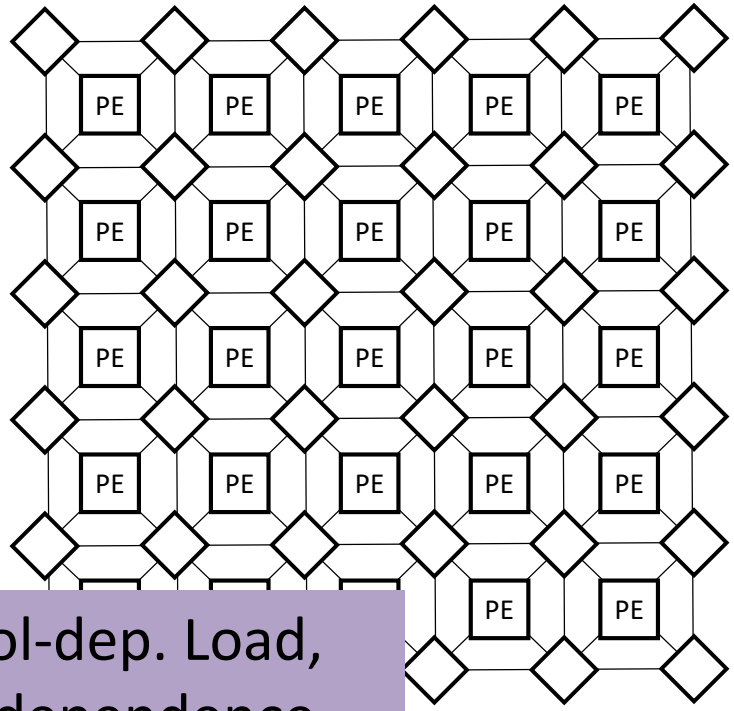
B[0]

idx	0	1	**3**
val	1	4	1

Traditional Dataflow



Systolic array



Control-dep. Load,
Cyclic dependence,
Unpredictable branch!

Novel Dataflow for Stream Join

Sparse MM Example

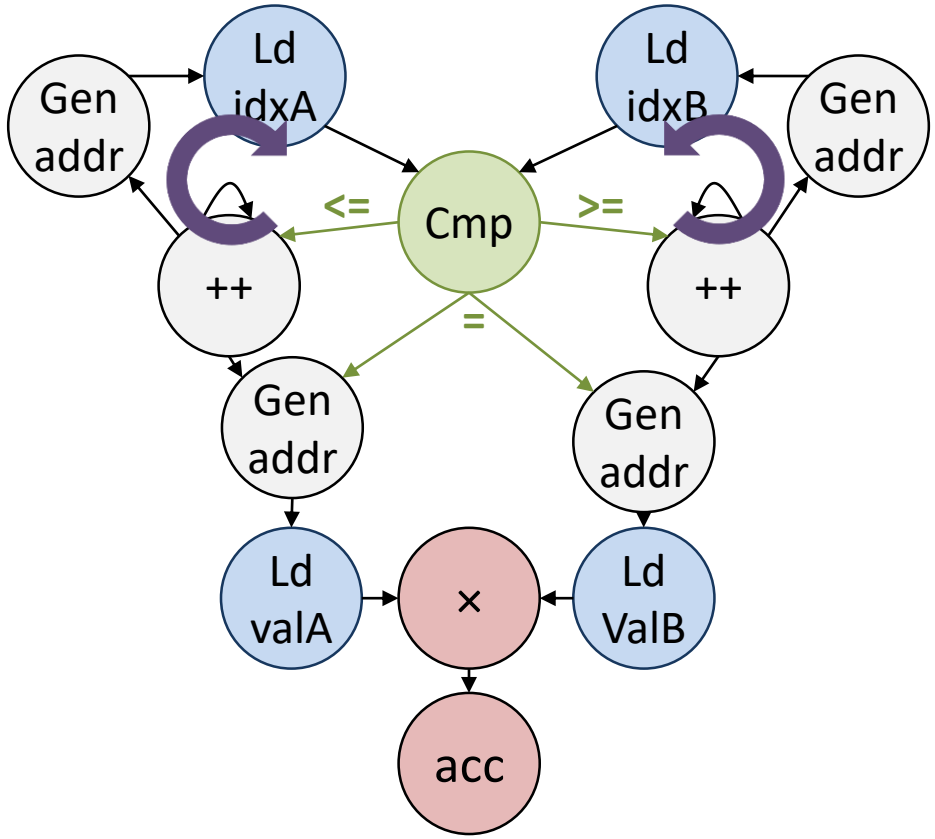
A

idx	2	**3**	5
val	2	3	4

B[0]

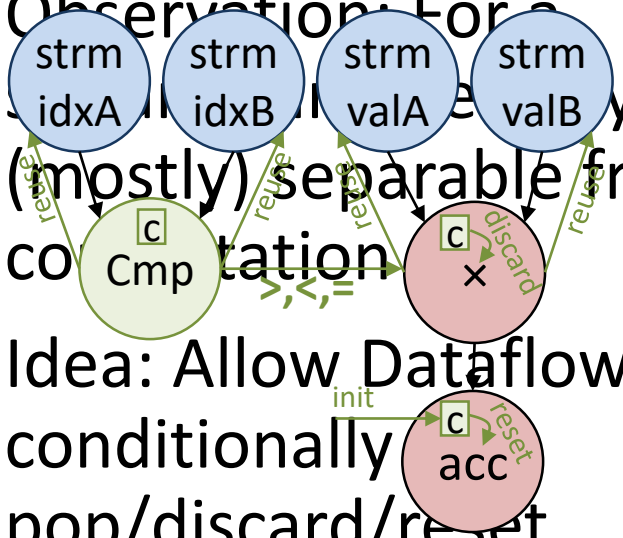
idx	0	1	**3**
val	1	4	1

Traditional Dataflow



Novel Stream Join Dataflow

- Observation: For a [illegible] is (mostly) separable from computation
- Idea: Allow Dataflow to conditionally pop/discard/reset values based on control decisions.



Novel Dataflow for Stream Join

Sparse MM Example

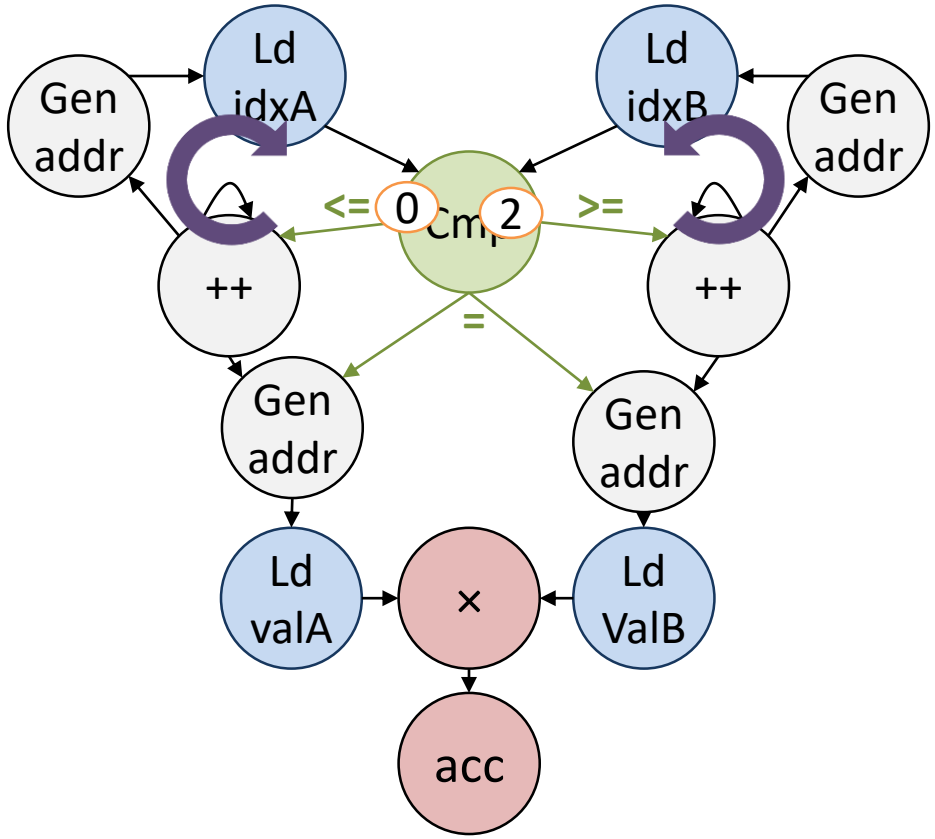
A

idx	2	**3**	5
val	2	3	4

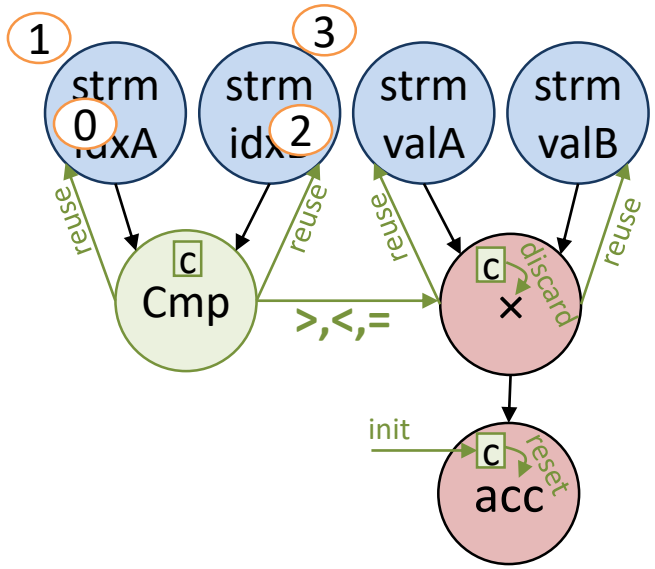
B[0]

idx	0	1	**3**
val	1	4	1

Traditional Dataflow



Novel Stream Join Dataflow



Novel Dataflow for Stream Join

Sparse MM Example

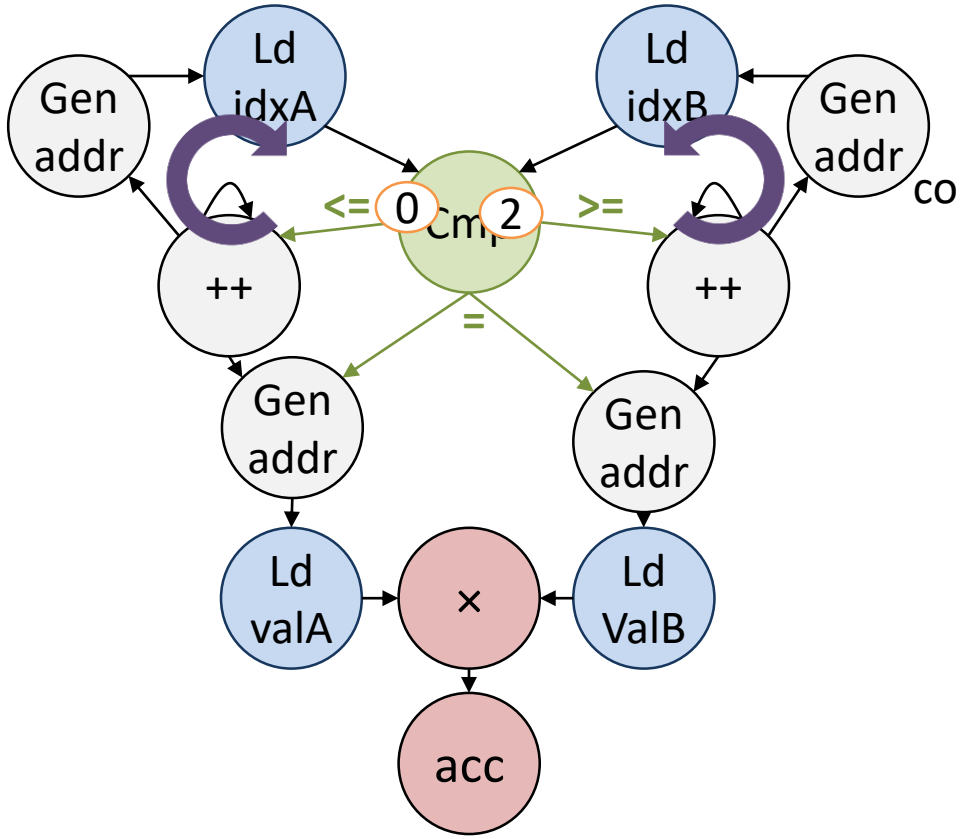
A

idx	2	**3**	5
val	2	3	4

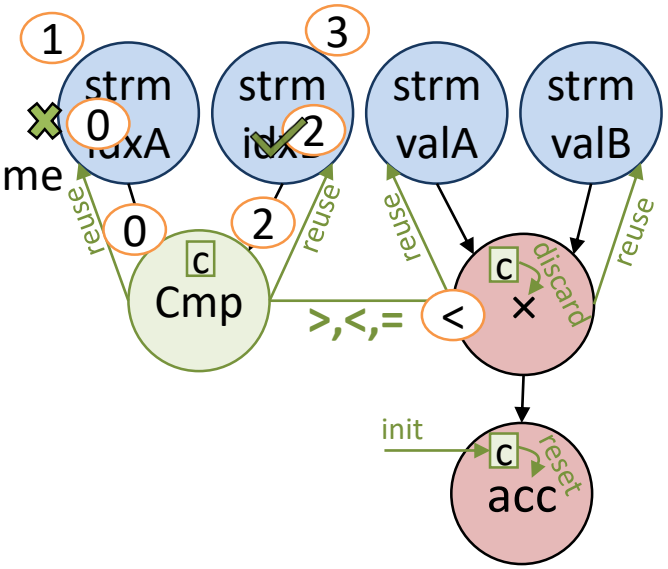
B[0]

idx	0	1	**3**
val	1	4	1

Traditional Dataflow



Novel Stream Join Dataflow



Novel Dataflow for Stream Join

Sparse MM Example

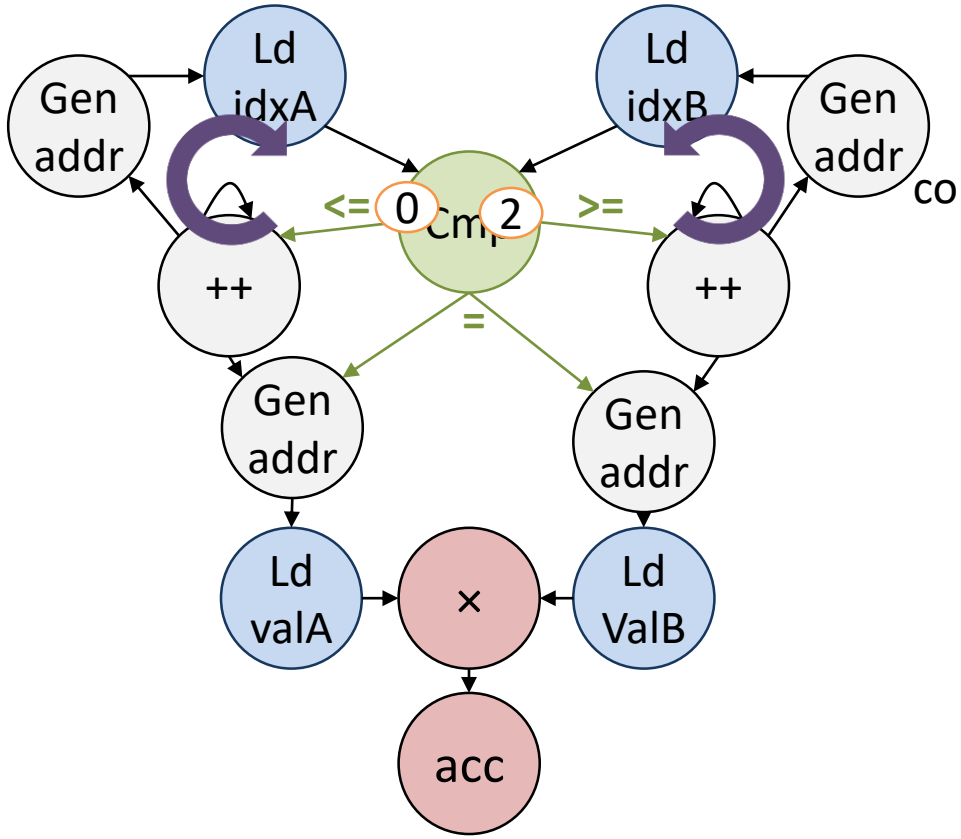
A

idx	2	**3**	5
val	2	3	4

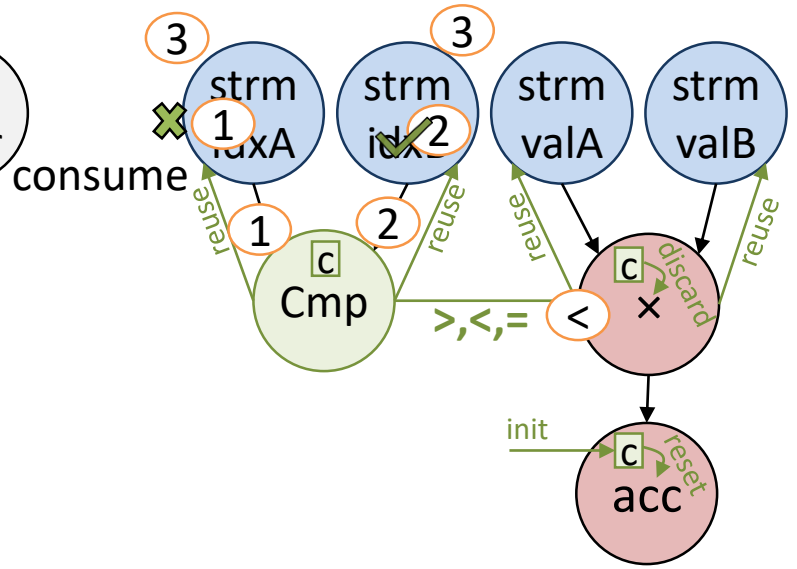
B[0]

idx	0	1	**3**
val	1	4	1

Traditional Dataflow

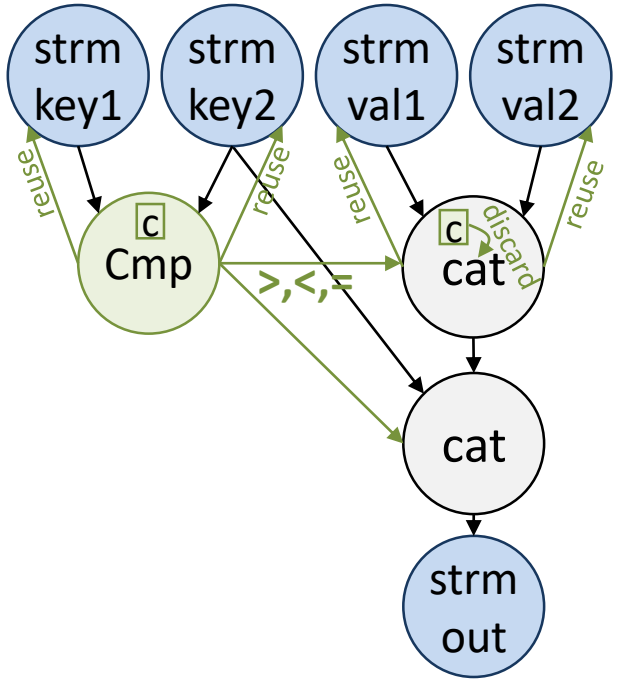


Novel Stream Join Dataflow

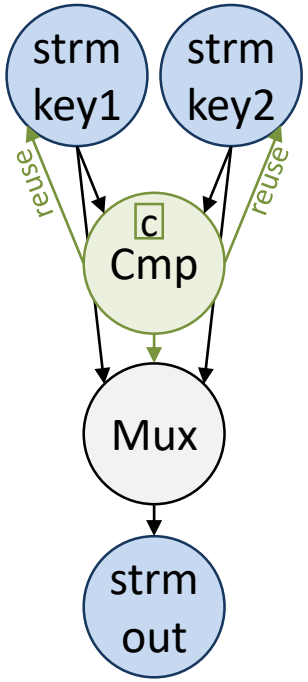


Other Kernels as Stream Join

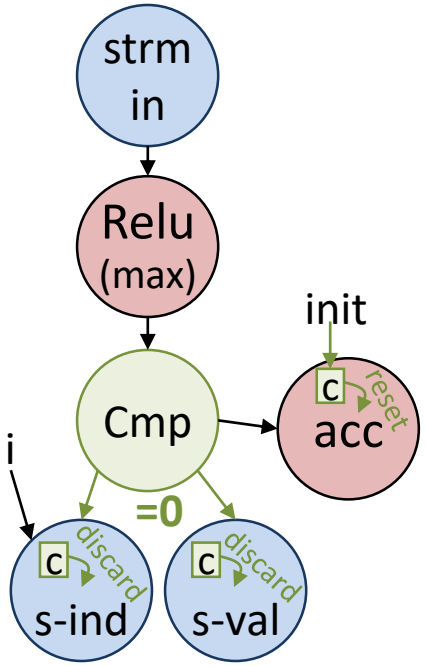
Database Join



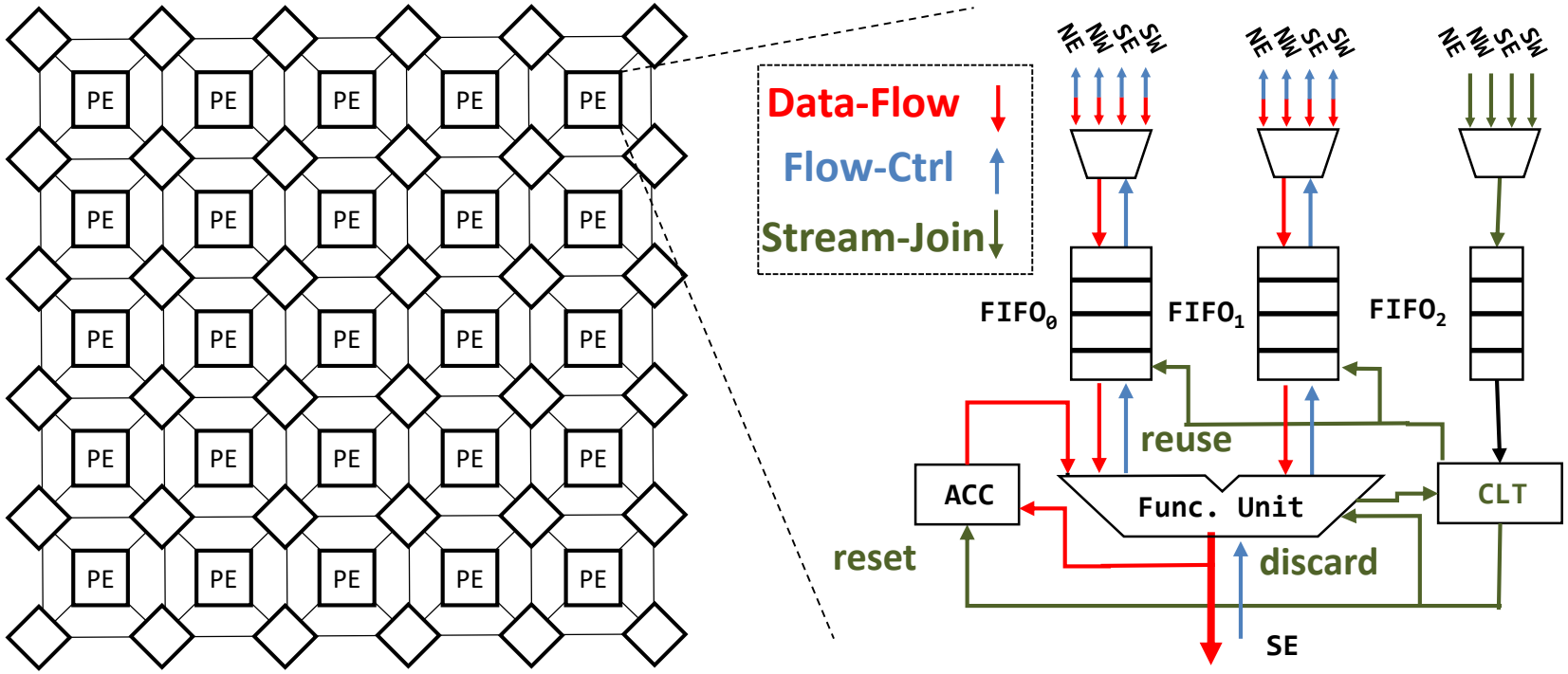
Merge (sort)



Resparsify (filter)



Supporting Stream-Join in Hardware

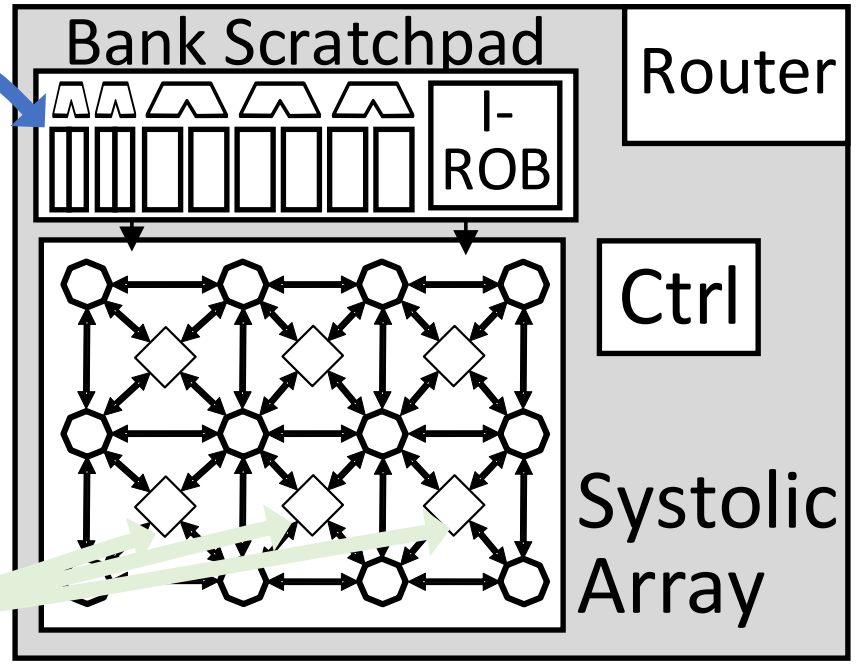


“Systolic” CGRA

CGRA Processing Element

Specializing for Indirection

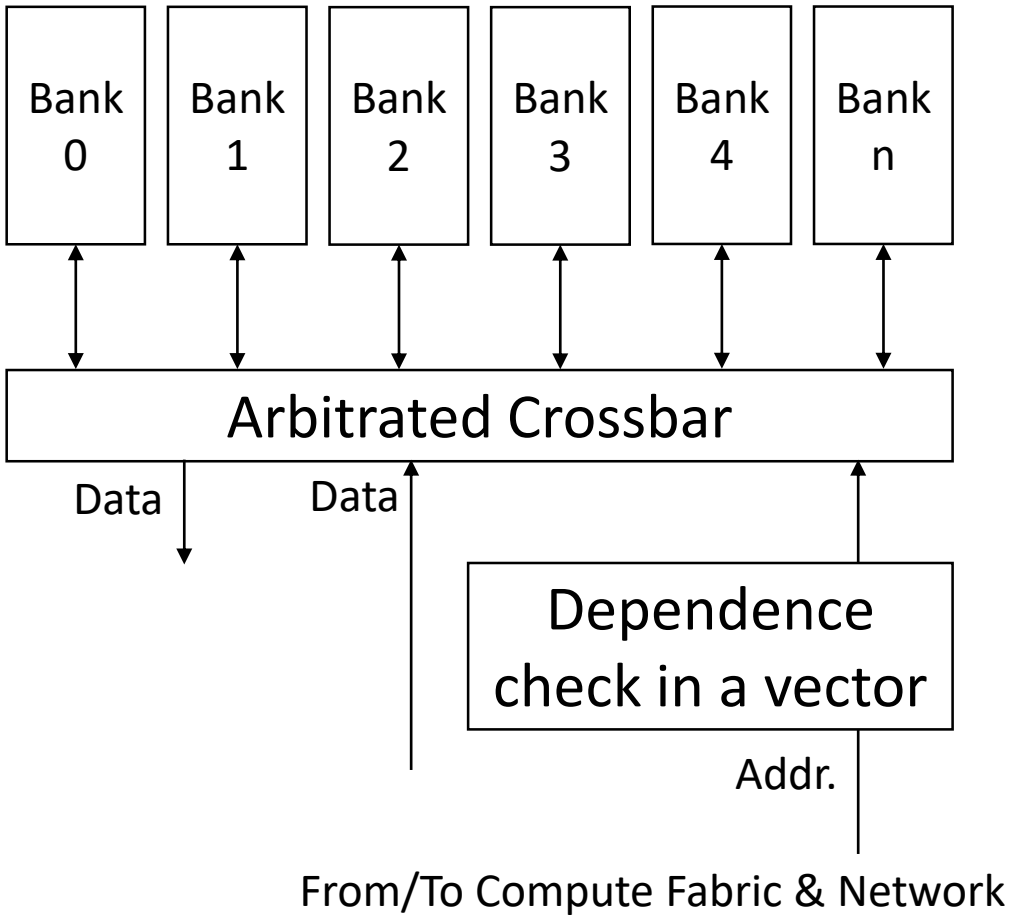
Compute-Enabled Scratchpad for fast Alias-free indirect access



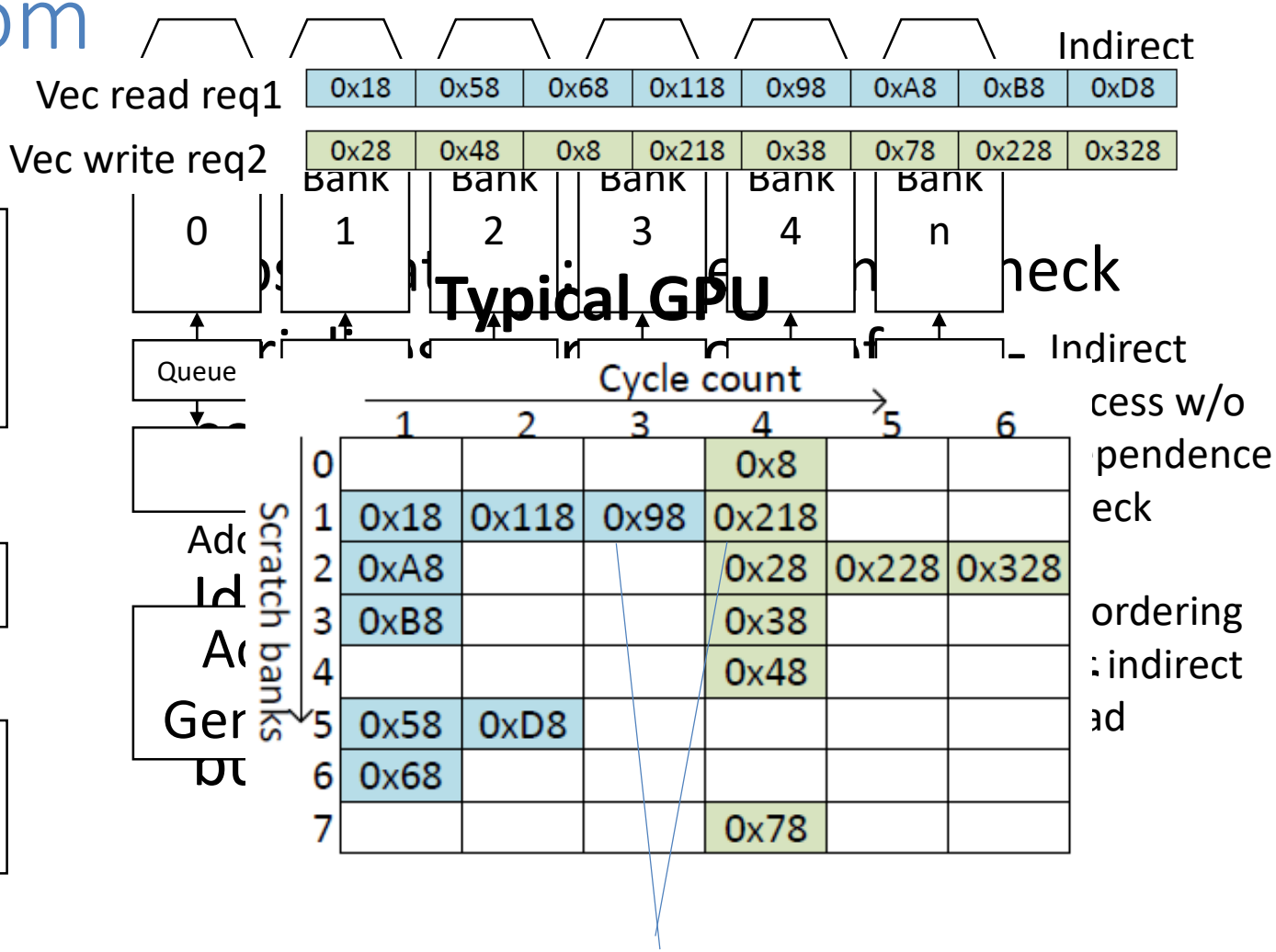
Systolic array supporting stream-join control

Indirection with guaranteed alias-freedom

Traditional Banked Memory



SPU Banked Memory

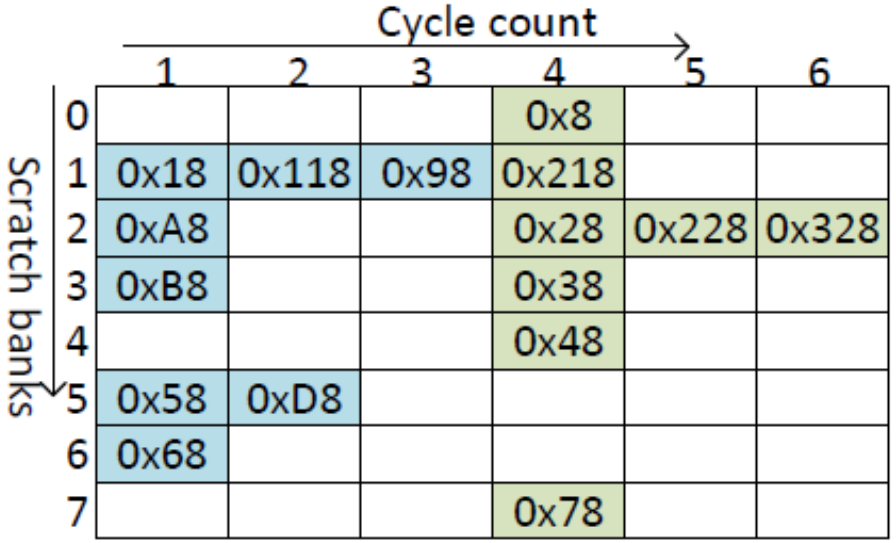


Known apriori that they won't alias. Serialization unrequired.

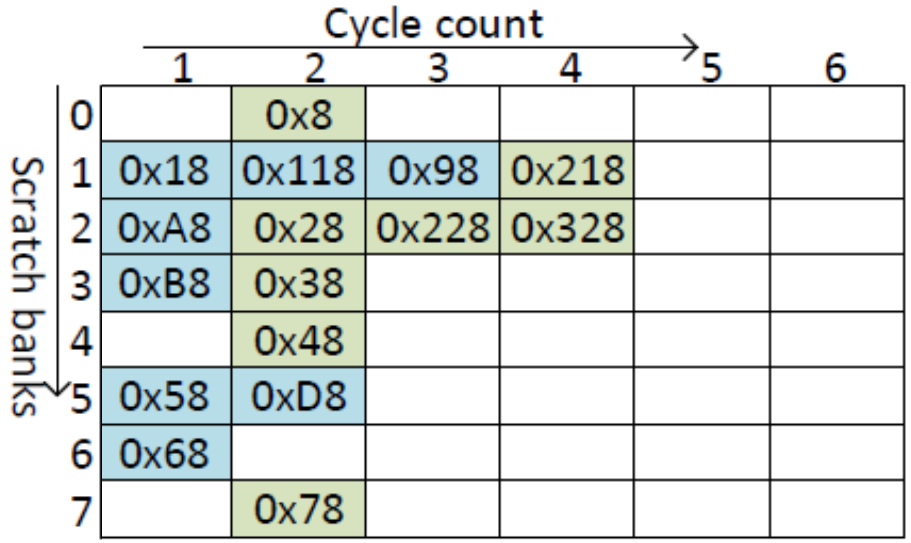
Indirect Access Reordering in Scratchpad

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

Typical GPU



SPU



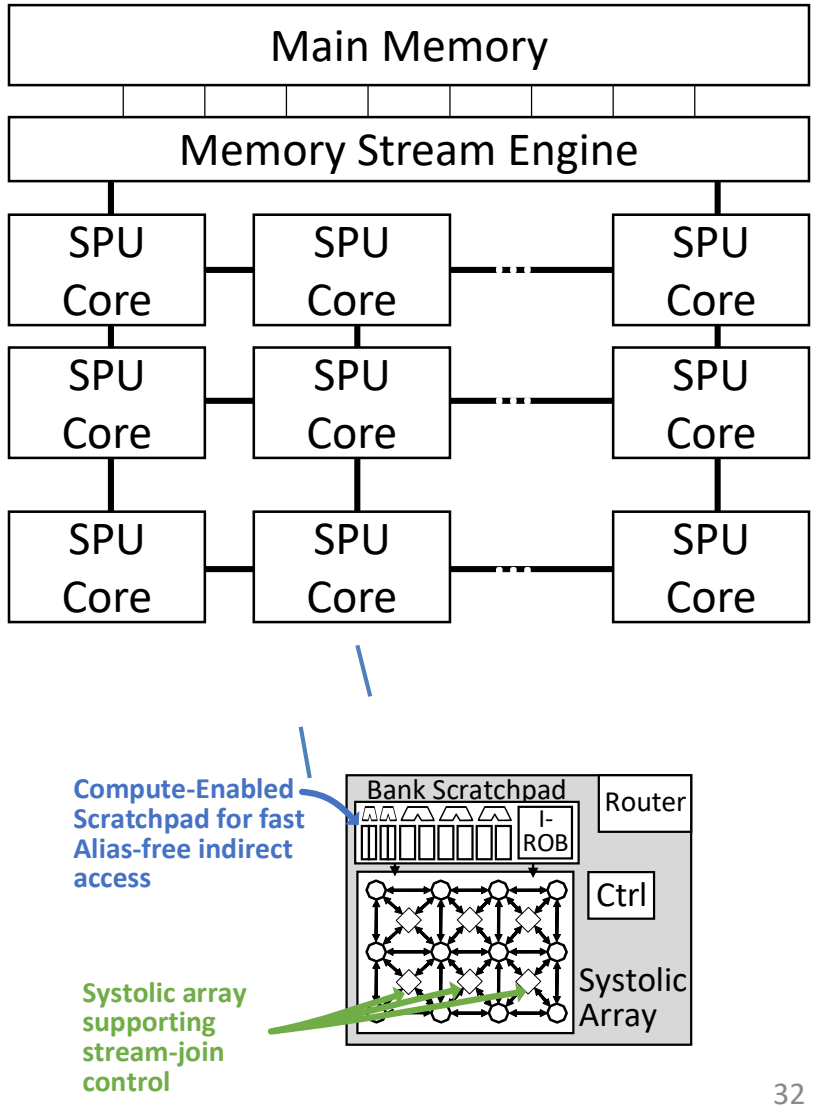
SPU: Sparse Processing Unit

Network: Traditional mesh NoC

Scratchpads in global address space: for nearby core communication

Broadcast: using memory stream engine

Synchronization: Dataflow Counters (sync. On SPAD write)



Outline

- Irregularity is ubiquitous
  - *Sufficient* and *Exploitable* forms of Control and Memory dependence
  - Example Workload: Matrix Multiply
- Exploiting data-dependence with SPU accelerator
  - uArch: Stream-join Dataflow & Compute-enabled Scratchpad
  - SPU Multicore Design
- **Evaluating SPU**
- Conclusion

Methodology

- Programming: C + Intrinsics + Dataflow Graphs
- SPU Simulation: Gem5+Ruby (RISCV inorder control core)

Benchmarks

Workloads	CPU	GPU
GBDT	LightGBM	LightGBM
Kernel-SVM	LibSVM	Hand-coded
AC	Hand-coded	Hand-coded
FC	MKL SPBLAS	cuSparse
Conv layer	MKL-DNN	cuDNN
Graph Alg.	Graphmat	-
TPCH	MonetDB	-

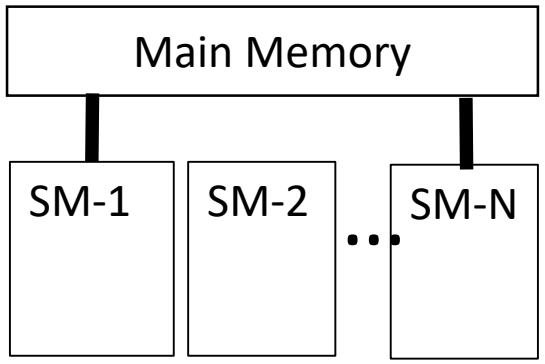
Datasets (with varying sparsity)

Wkld				
GBDT	Cifar10-bin (1)	Higgs-bin (0.28)	Yahoo-bin (0.05)	Ltrc-bin (0.008)
KSVM	Connect (0.33)	Higgs (0.92)	Yahoo (0.59)	Ltrc (0.24)
CONV	VGG-3 (0.34)	VGG-4 (0.1)	ALEX-2 (0.14)	RES-1 (0.05)
FC	VGG-12 (0.04)	VGG-13 (0.09)	ALEX-6 (0.16)	RES-1 (0.22)
AC	Pigs	Munin	Andes	Mildew
Graph	Flickr	Fb-artist	NY-road	

Domain-agnostic comparison points

P4000 Pascal GPU

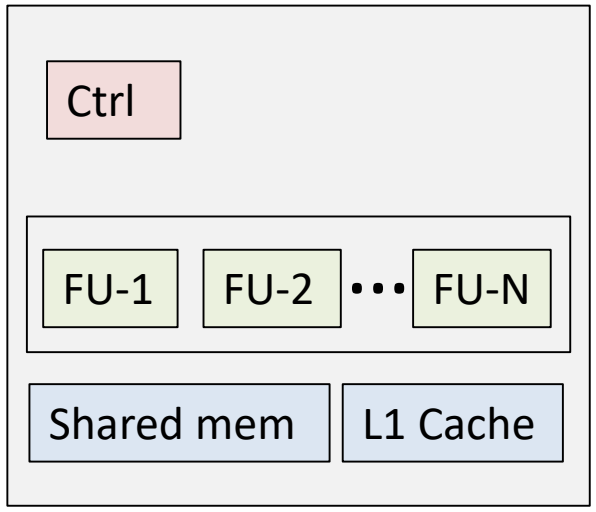
Overall Design



Mem bw = 243 GB/s

Core Design

SIMD unit

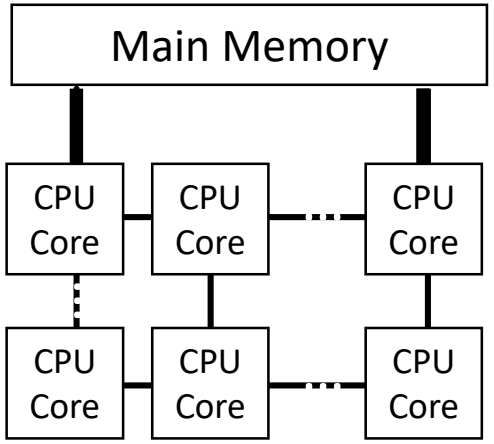


FP units: 3696

On-chip mem: 4MB

SPU-inorder

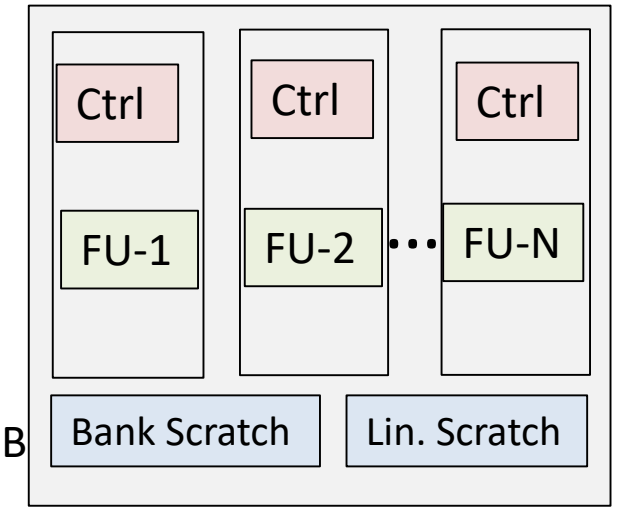
Overall Design



Mem bw = 256 GB/s

Core Design

Array of in-order cores

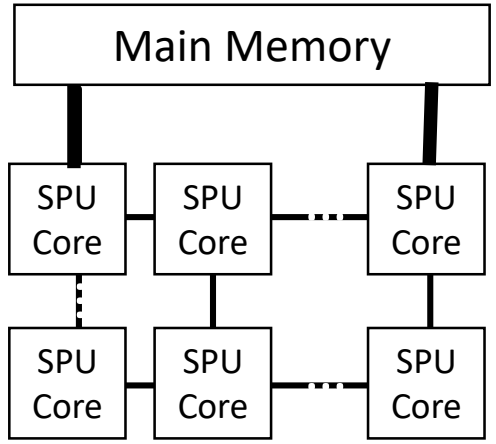


FP units: 2560

On-chip mem: 3MB

SPU

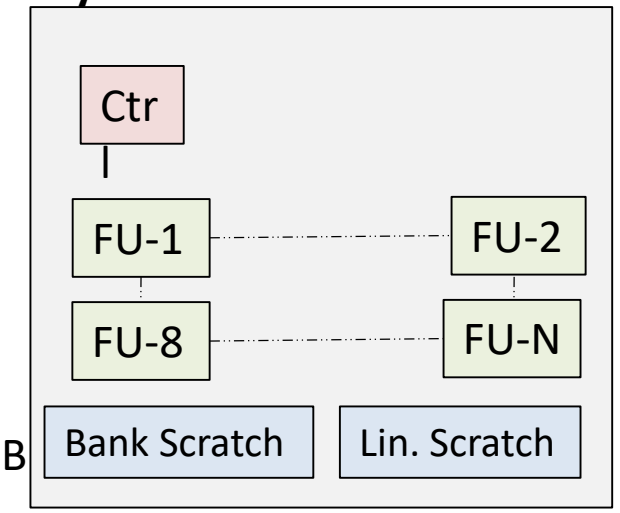
Overall Design



Mem bw = 256 GB/s

Core Design

Systolic CGRA

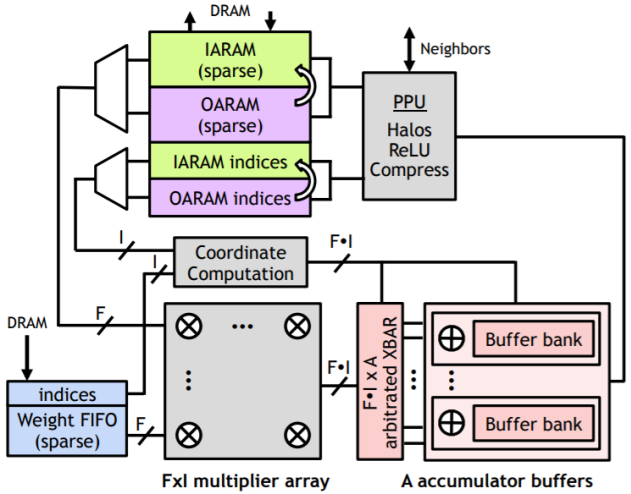


FP units: 2632

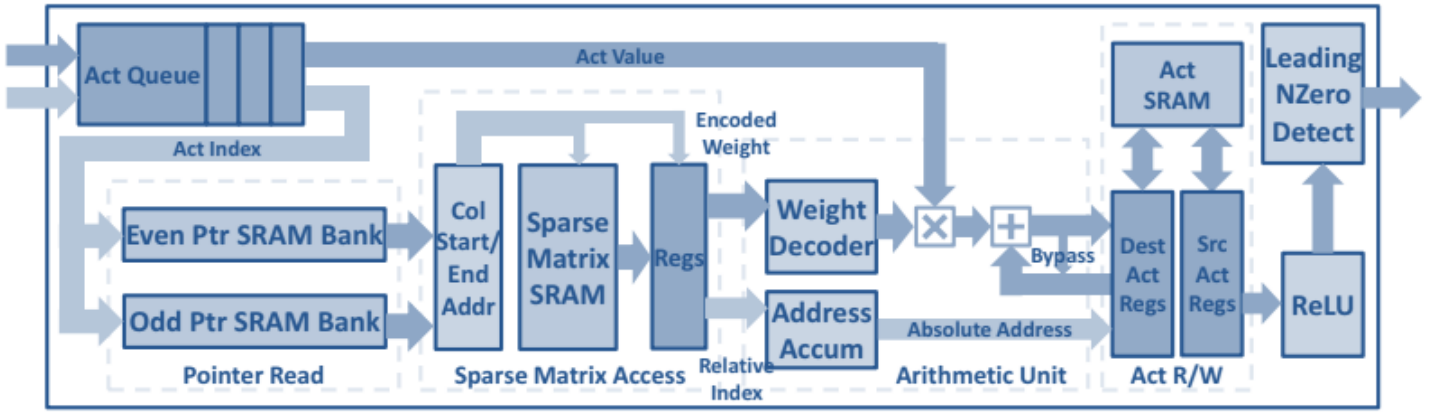
On-chip mem: 3MB

Domain-specific comparison points

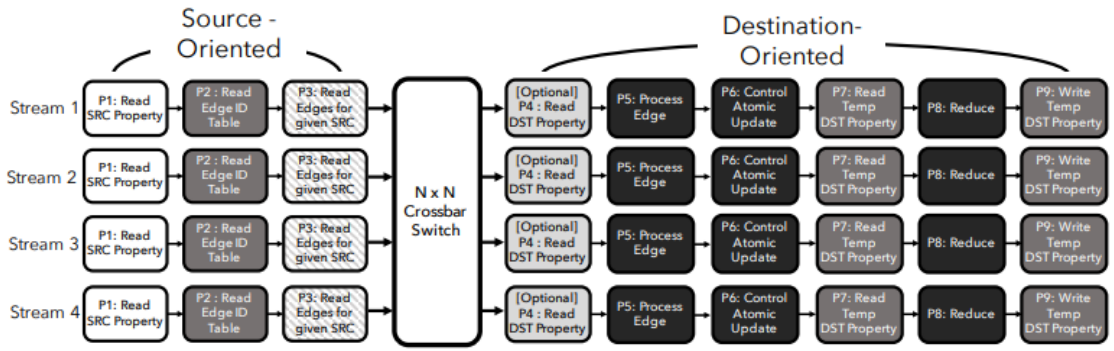
SCNN (Sparse convolution): ISCA'17



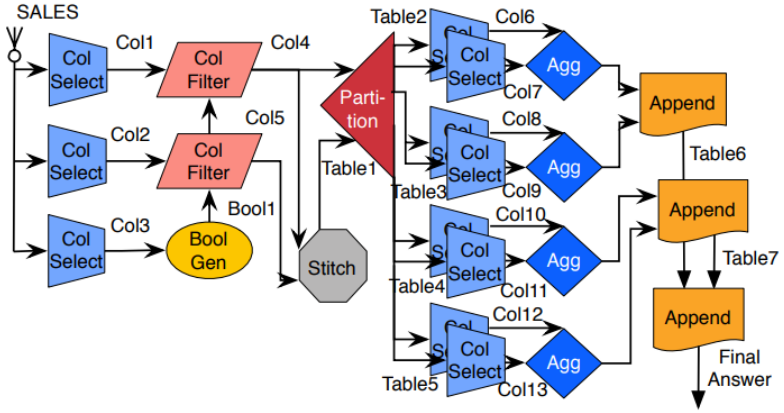
EIE (Sparse fully connected): ISCA'16



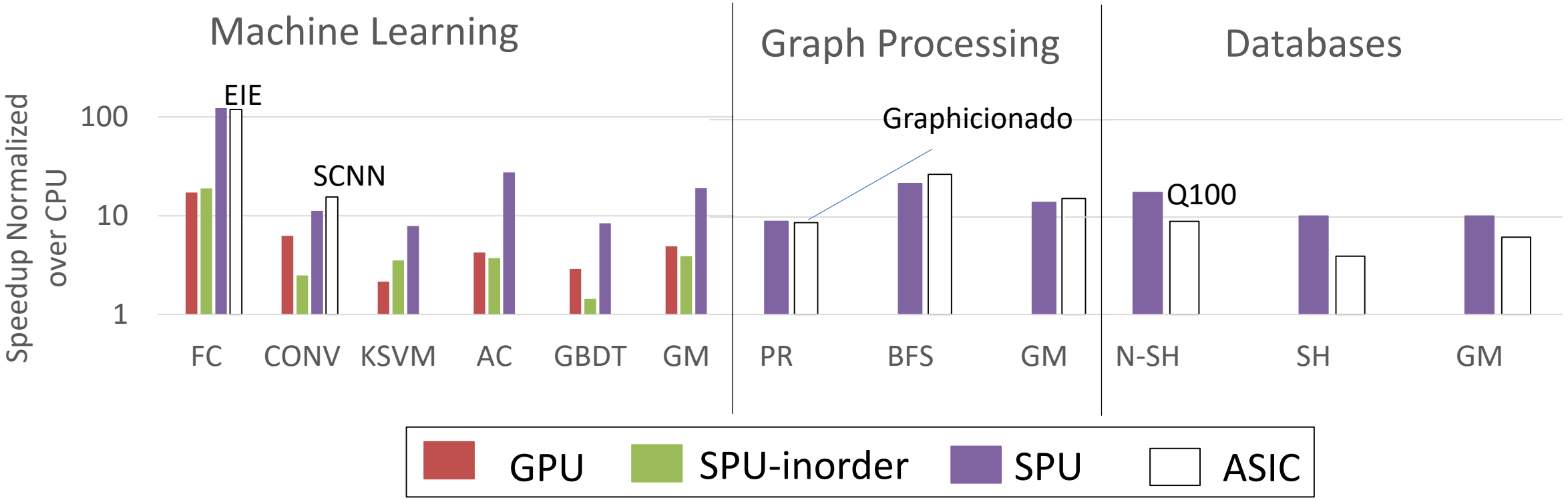
Graphicionado (Graph Analytics): MICRO'16



Q100 (Databases): ASPLOS'14



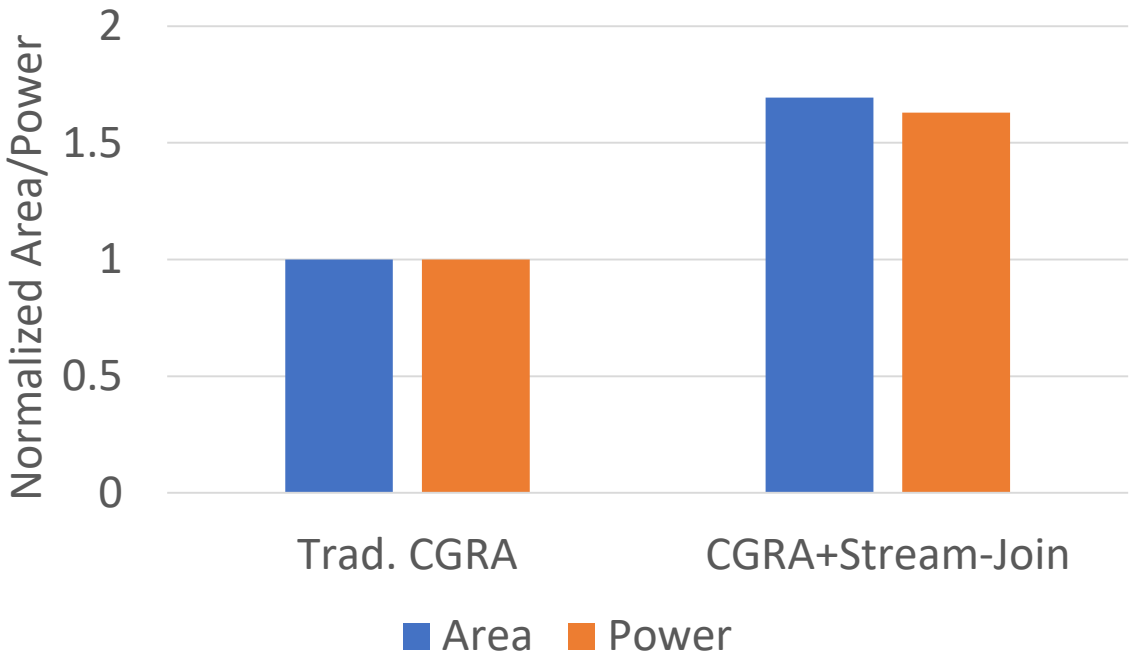
Overall Results



Cost of adding stream-join in systolic CGRA

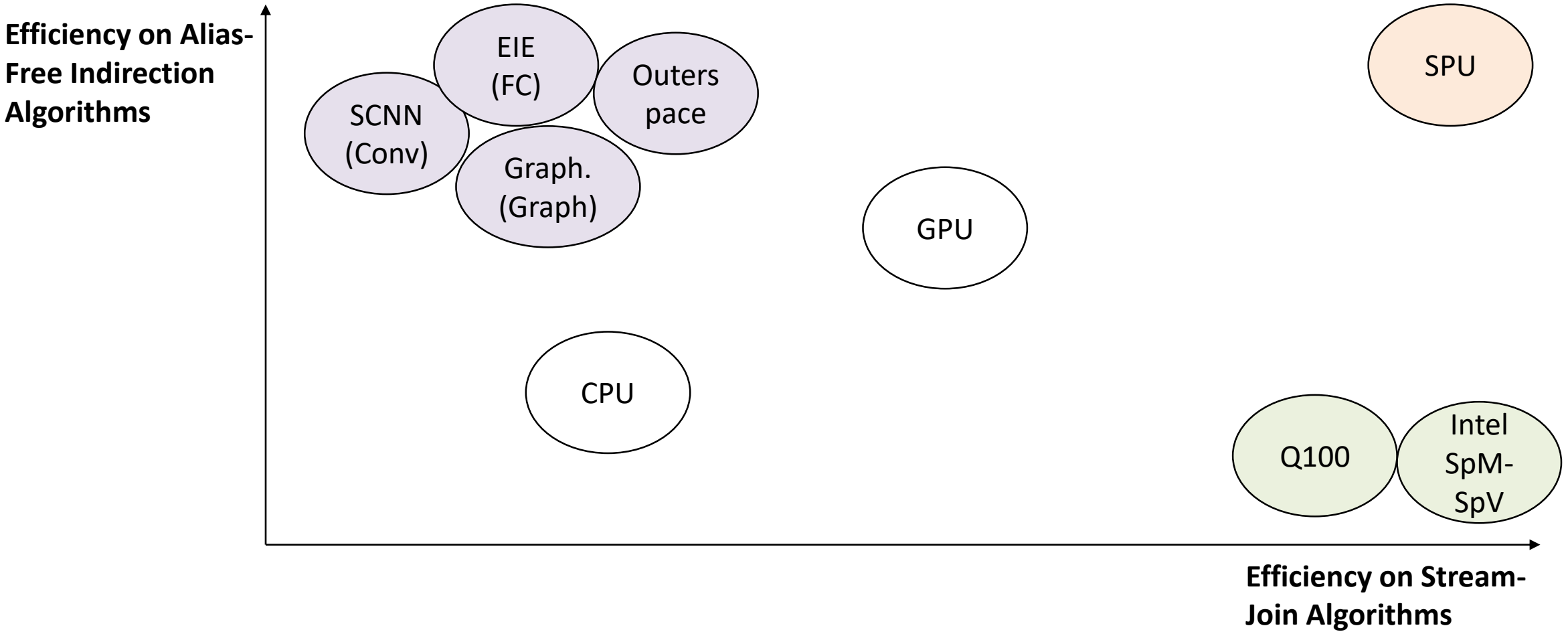
- 1.69x area overhead due to addition of **flow control**.
- 1.63x power overhead.

Compared to whole design, it is 6.9% area overhead and 14.2% power overhead.



Methodology: SPU's DGRA is implemented in Chisel and synthesized using Synopsis DC with a 28nm UMC technology library.

Conclusion

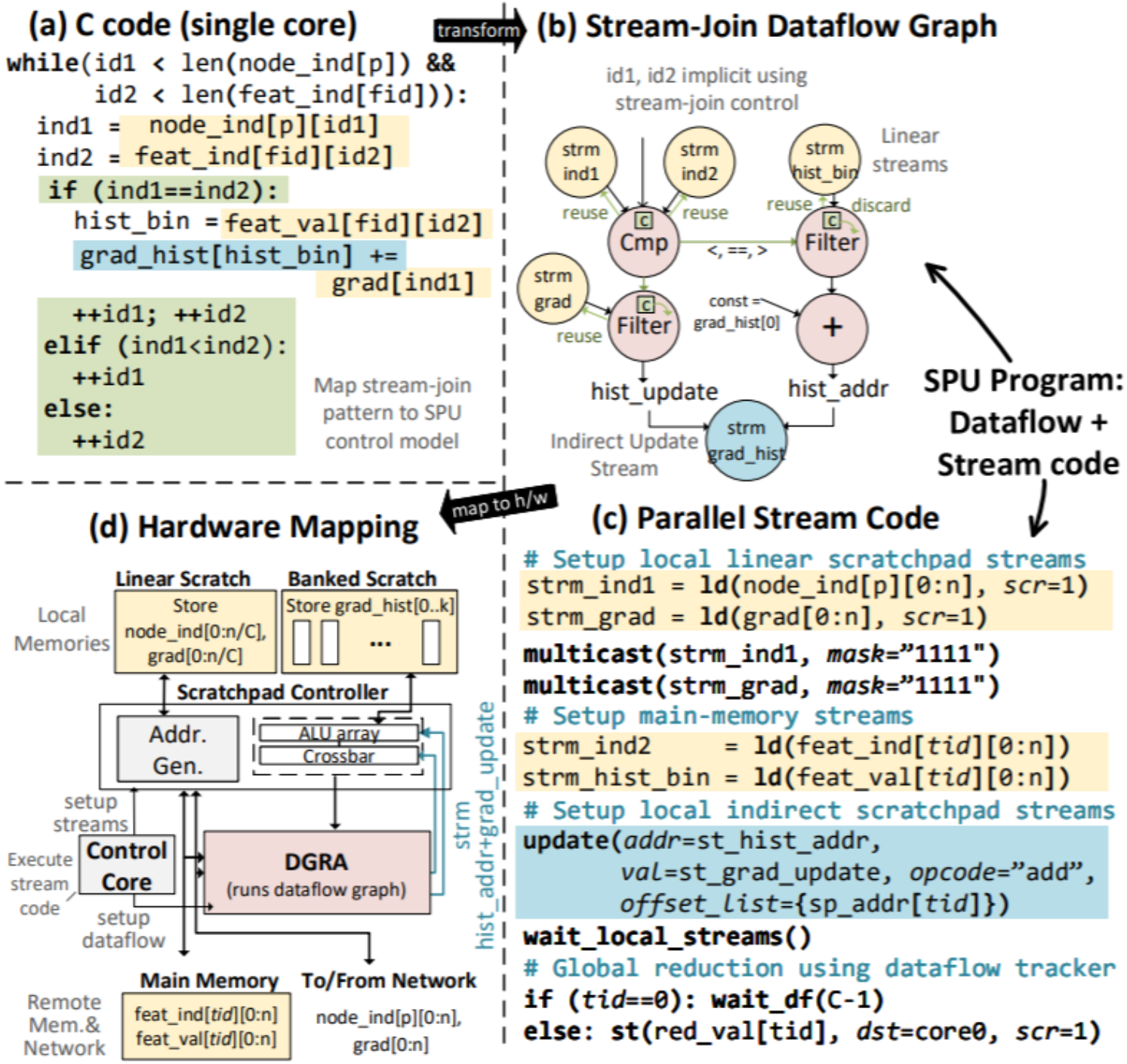

Efficiency on Alias-Free Indirection Algorithms
EIE (FC)
Outers pace
SCNN (Conv)
Graph. (Graph)
SPU
GPU
CPU
Q100
Intel SpM-SpV
Efficiency on Stream-Join Algorithms

EXTRA SLIDES

Programming SPU

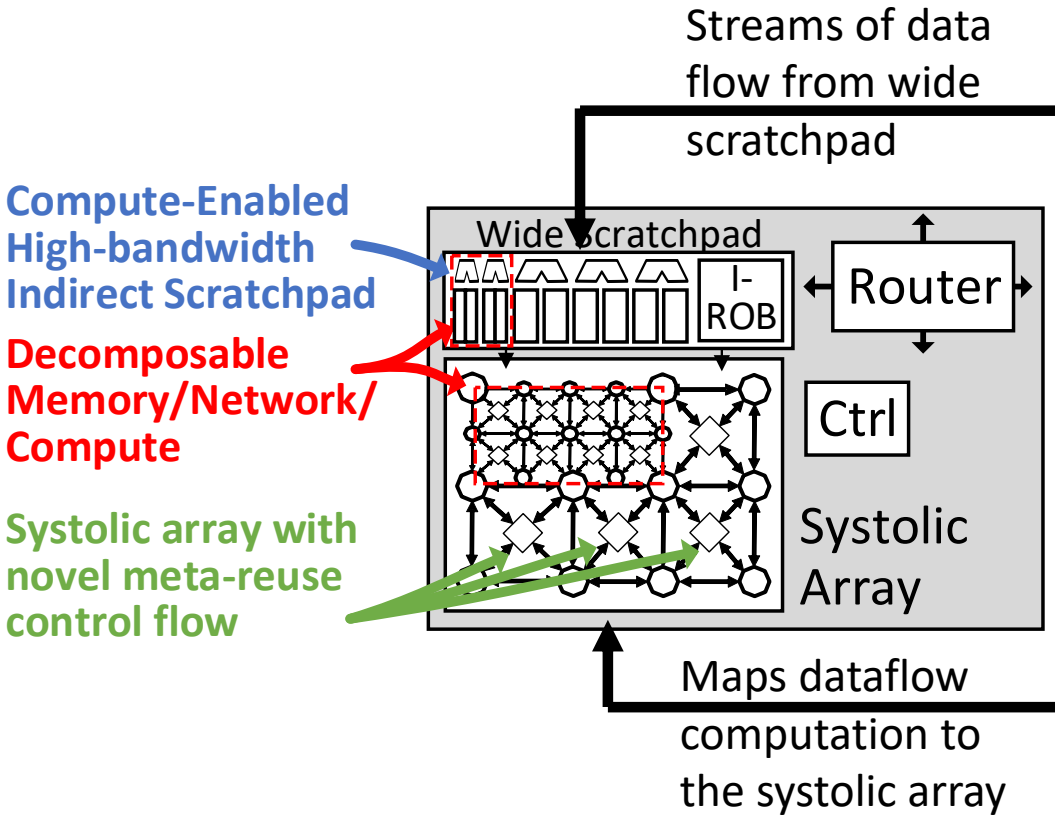
Example of gradient boosting decision trees (GBDT)

- Stream join control expressed in the dataflow graph
- Alias-Free Indirection expressed as update stream

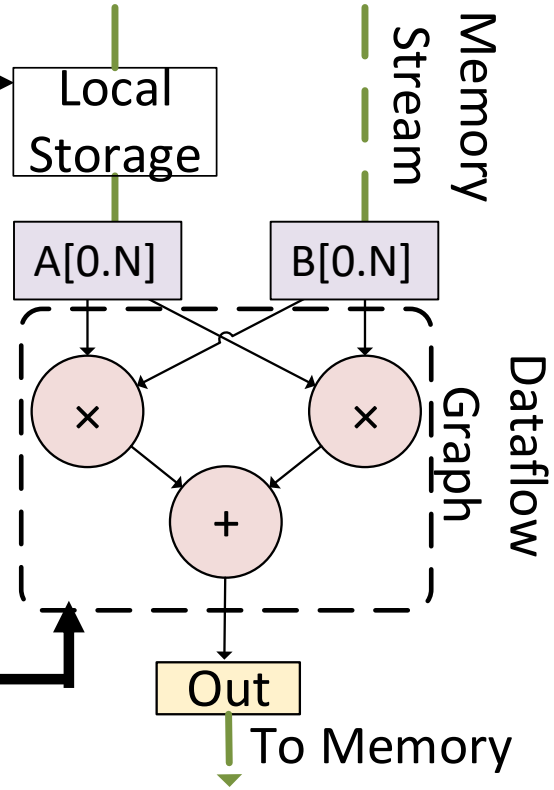


Approach Overview

Sparsity-Enabled SPU Core



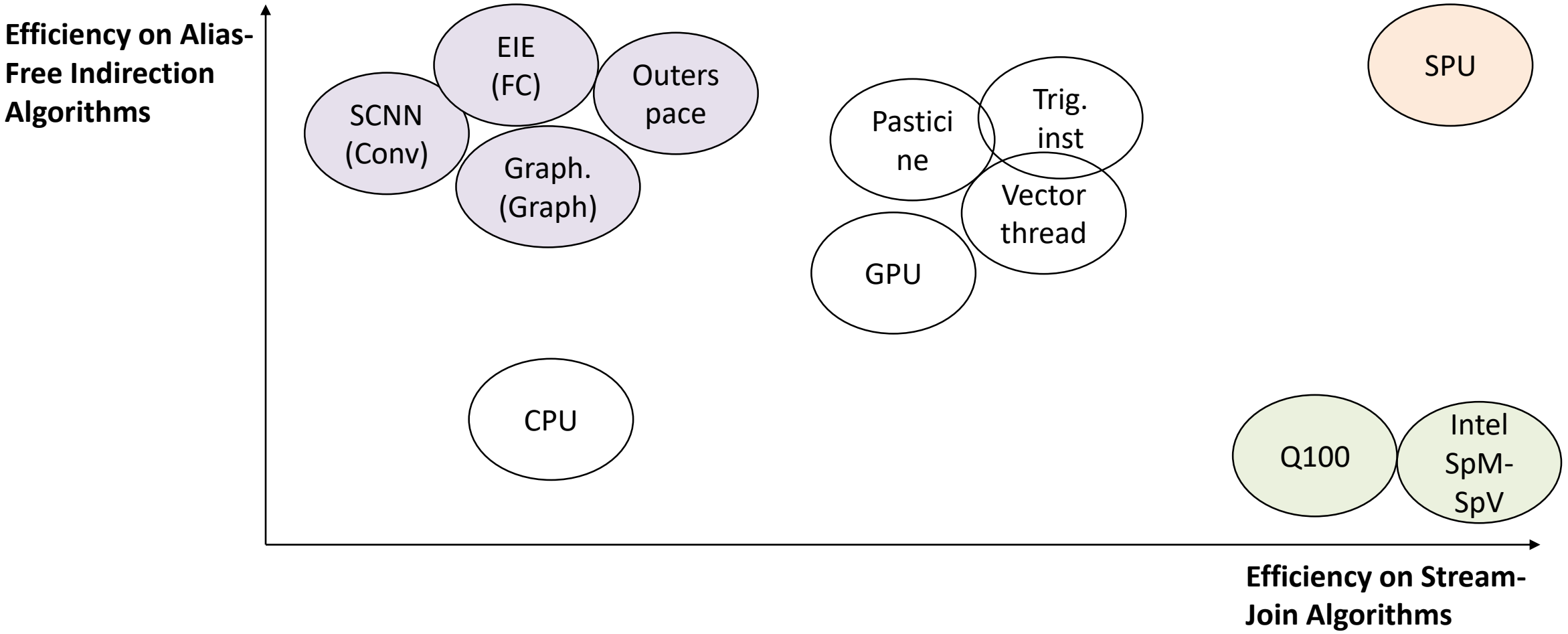
Stream-Dataflow ISA



Dataflow Computation: Dependence graph (DFG) with input/output vector ports.

Streaming Memory: Streams of data fetched from memory and stored back to memory.

Note: The relative position are the best to our knowledge



Indirect Access Reordering in Scratchpad

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

Typical GPU

Cycle count

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		

SPU

Cycle count

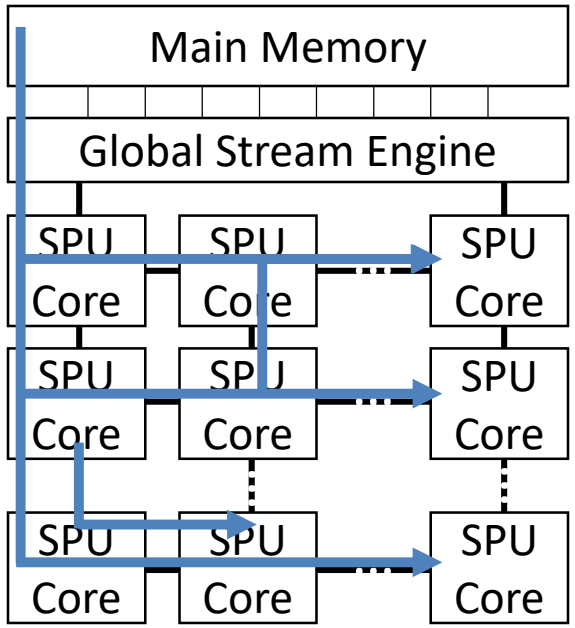
Scratch banks	1	2	3	4	5	6
0		0x8				
1	0x18	0x118	0x98	0x218		
2	0xA8	0x28	0x228	0x328		
3	0xB8	0x38				
4		0x48				
5	0x58	0xD8				
6	0x68					
7		0x78				

IROB Buffer at Cycle 2

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

SPU: Sparse Processing Unit

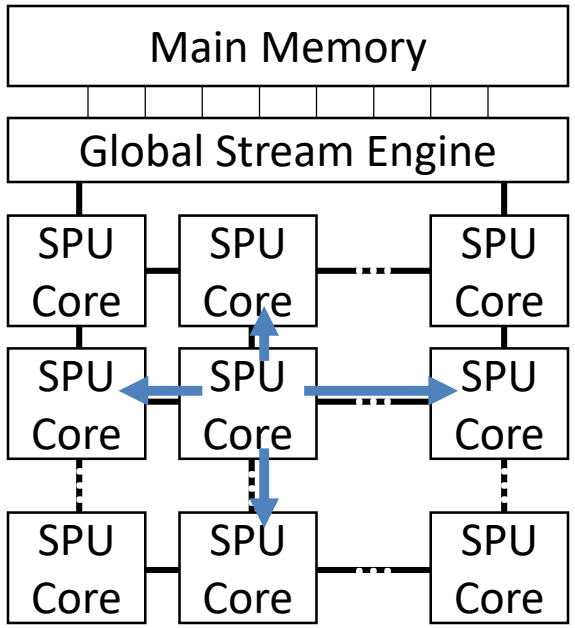
Independent Lanes (with/without Communication)



Fully Connected Layer
(broadcast row)

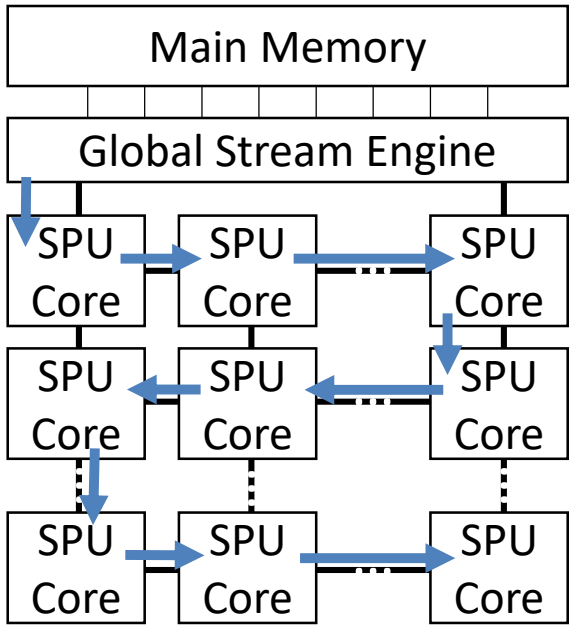
Graph Processing
(core-core communication)

Local Spatial Communication



Sparse Convolution
(communication with neighbors for halos)

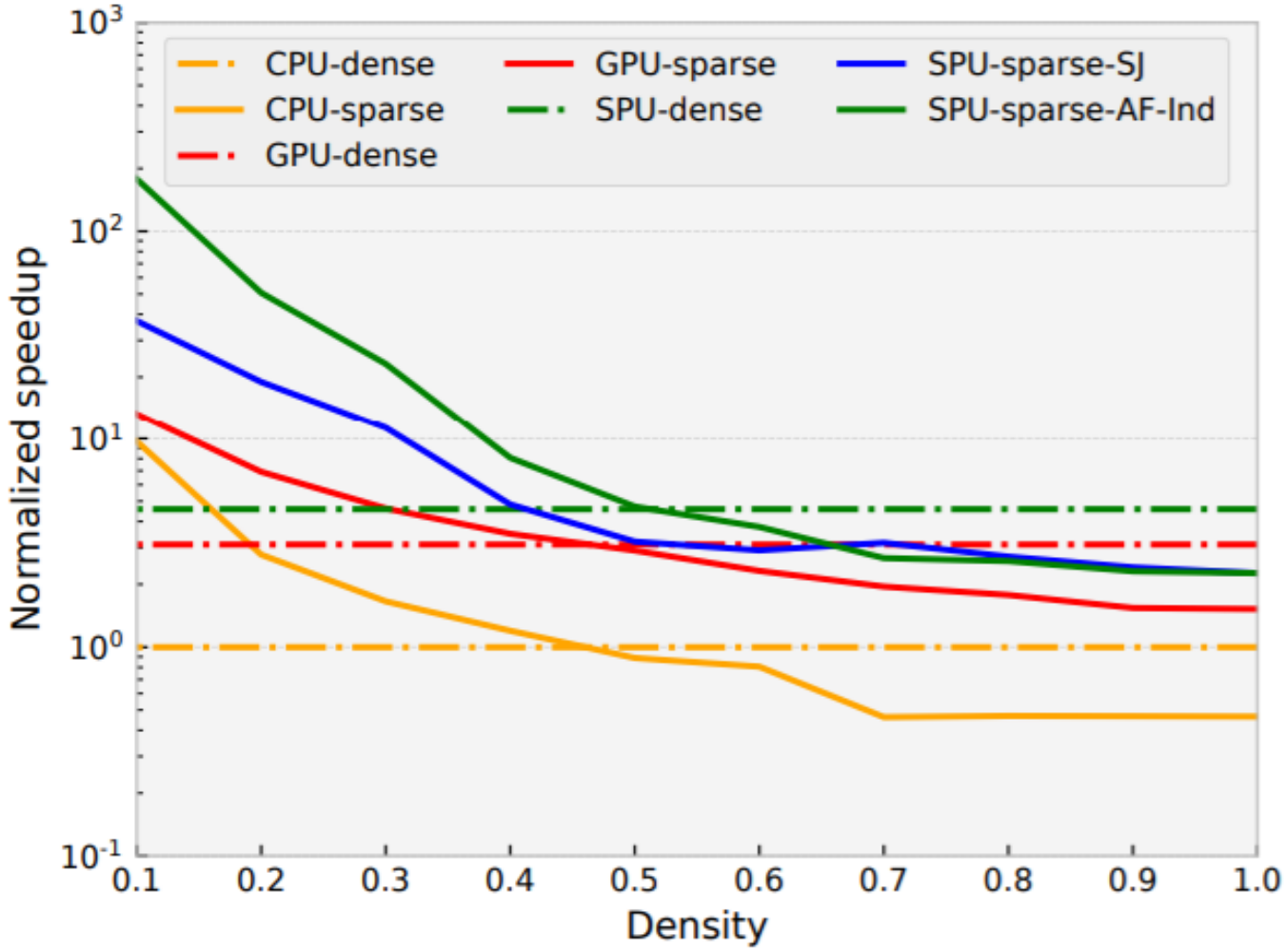
Pipelined Communication



Arithmetic Circuits –
Pipelined DAG Traveral
(pipeline node updates)

How much density is exploitable?

- Bounded by memory bandwidth, sparse versions are better at less than 50% density.
- SPU-sparse has exponential gain with sparsity.



Alias-free Indirection Abstractions

1: Indirect Memory -- `d = a[b[i]]`

- Allow to specify indirect loads or stores using an input stream as address values.
- Offset list for array-of-structs organization.

Example C code

```
struct {int f1, f2} a[N]
for i=0 to N
  ind = b[i]
  .. = a[ind].f1
offset_list={0,4})
  .. = a[ind].f2
```

->

Stream code

```
str1=load(b[0..n])
ind_load(addr=str1,
```

Update stream

2: Histogram -- `a[hist_bin] += c`

- Enhance ISA with compute-enabled semantics for the access stream
- Add update stream for common reduction operations

```
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})
```

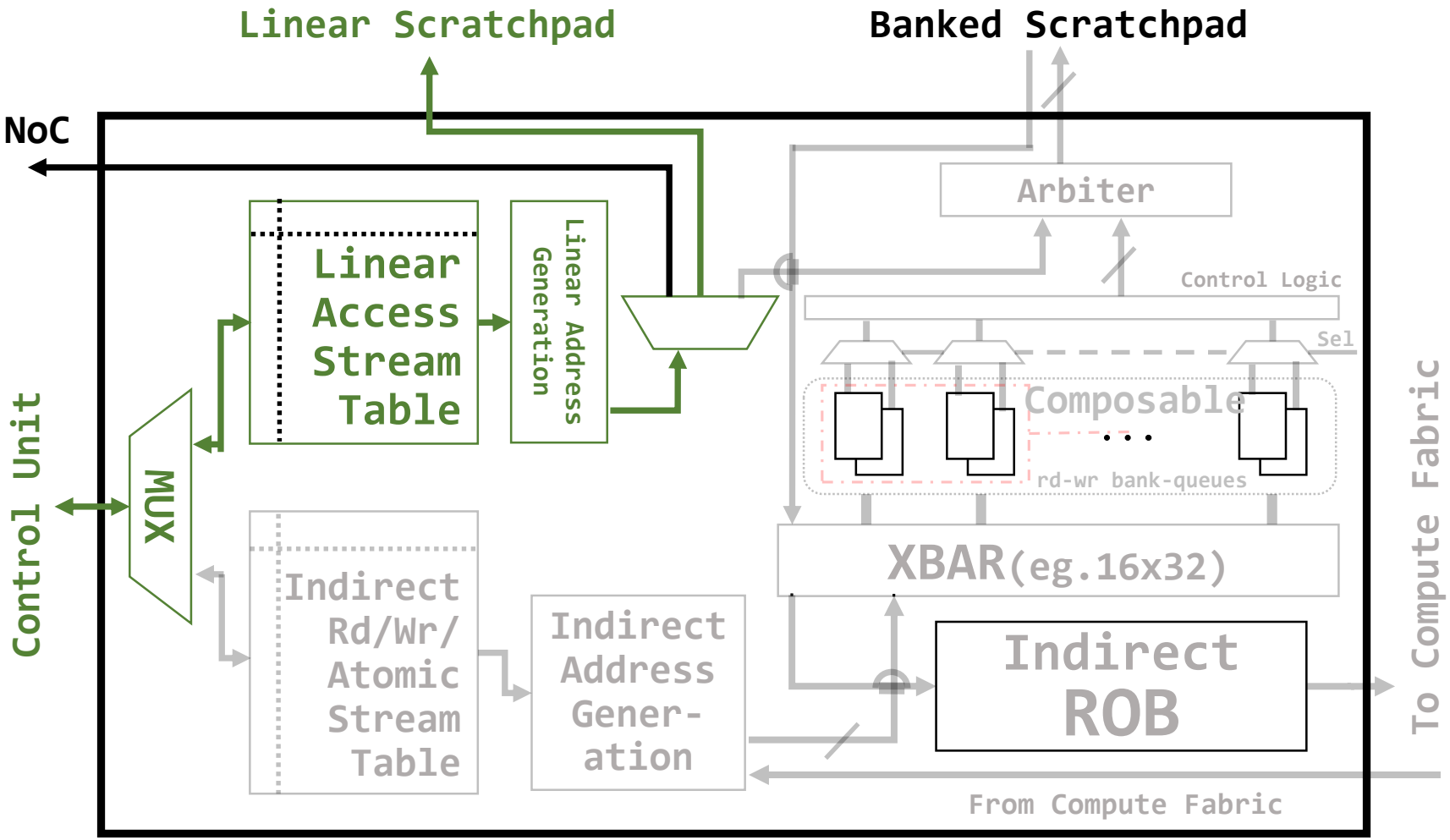
Listing 3: Indirect Update Stream

Sparsity-Enhanced Memory Micro-architecture

We keep 2 logical scratchpad memories: banked and linear.

Linear Memory

- reads/writes



Sparsity-Enhanced Memory Micro-architecture

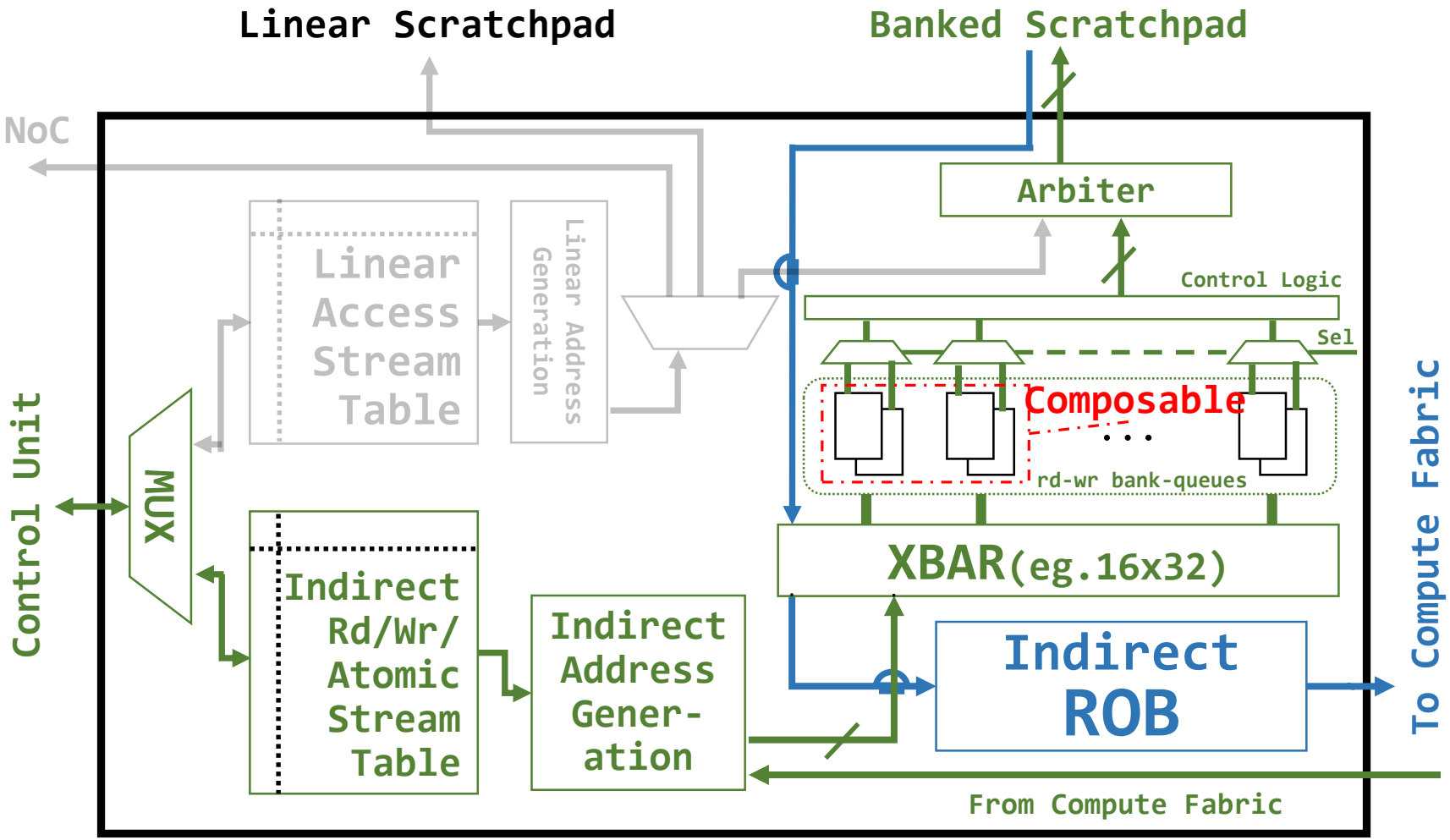
We keep 2 logical scratchpad memories: banked and linear.

Linear Memory

- reads/writes

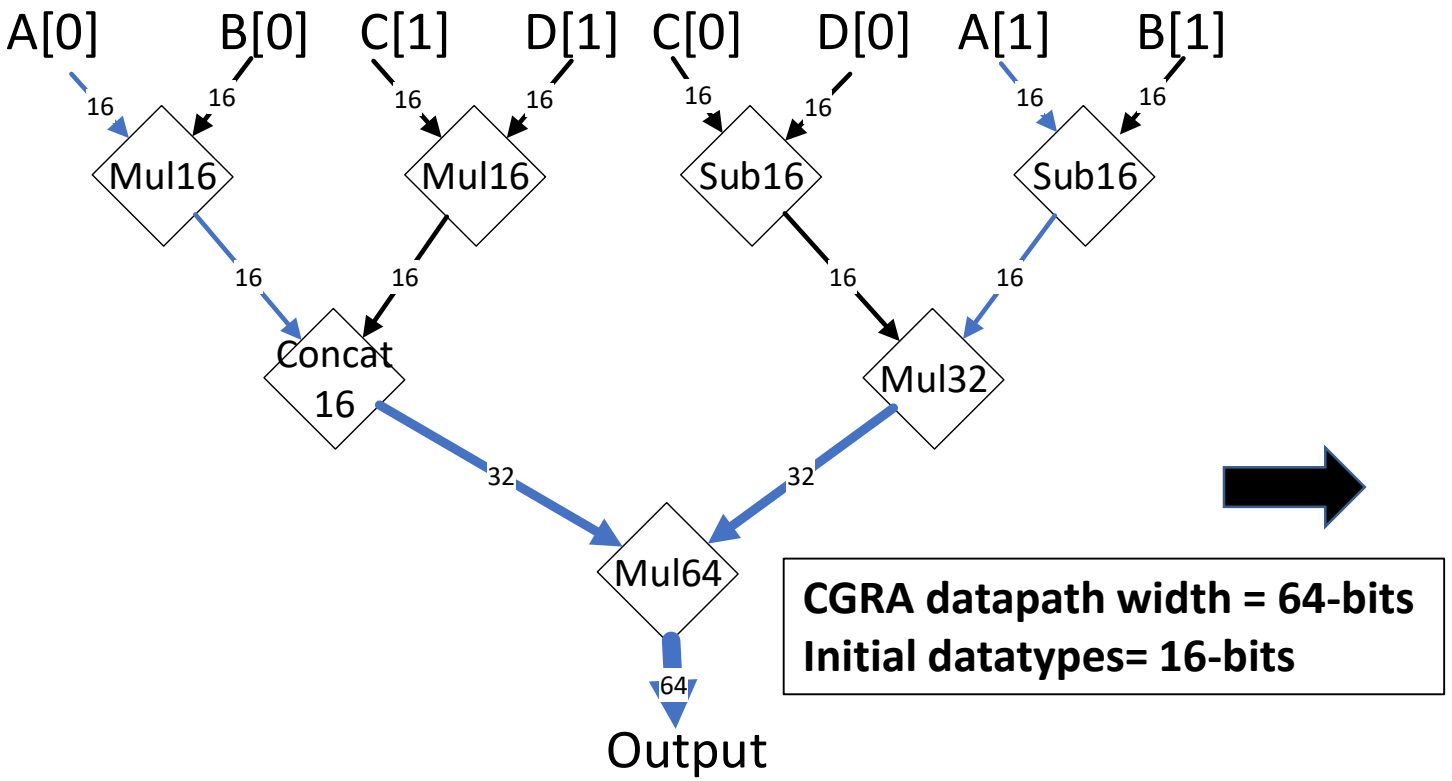
Banked Memory

- Indirect writes/updates
- Indirect reads



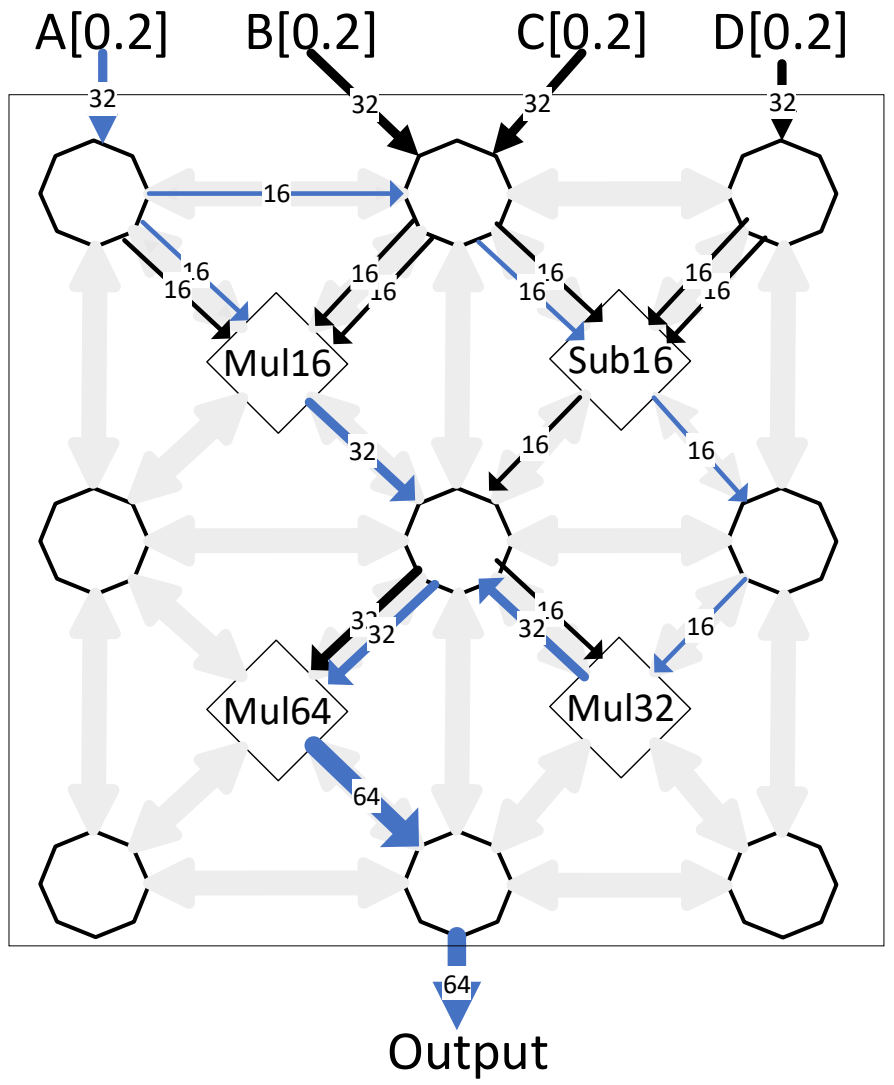
Benefits of Heterogeneity on CGRA

Example DFG



CGRA datapath width = 64-bits
Initial datatypes= 16-bits

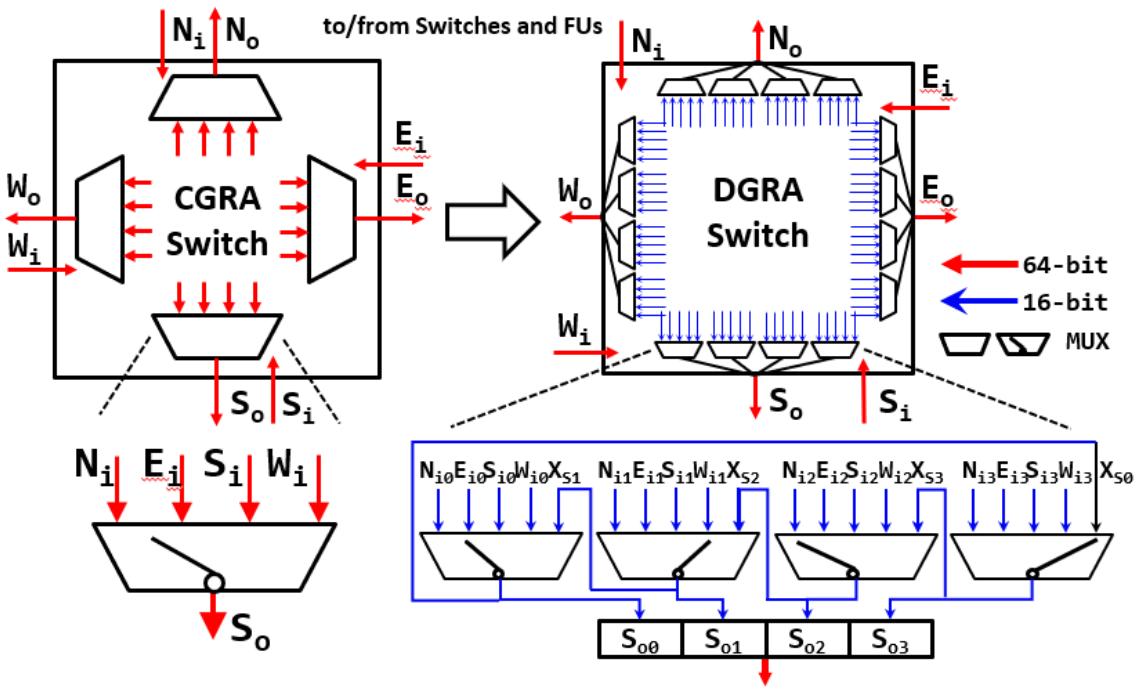
Mapping to DGRA



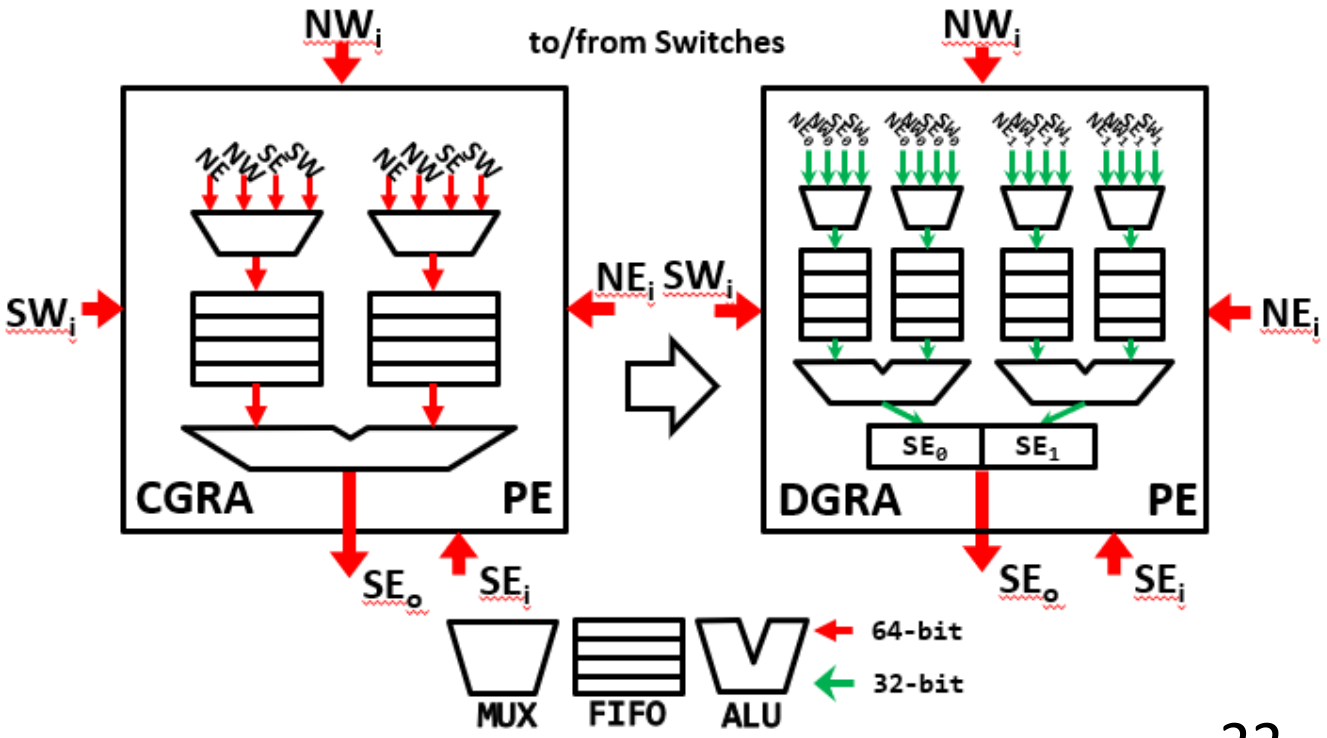
- Effective vectorization width is increased by 64/data-width.
- DGRA supports Concat and Extract using sub-networks.

Sparsity-Enhanced Computation Micro-architecture

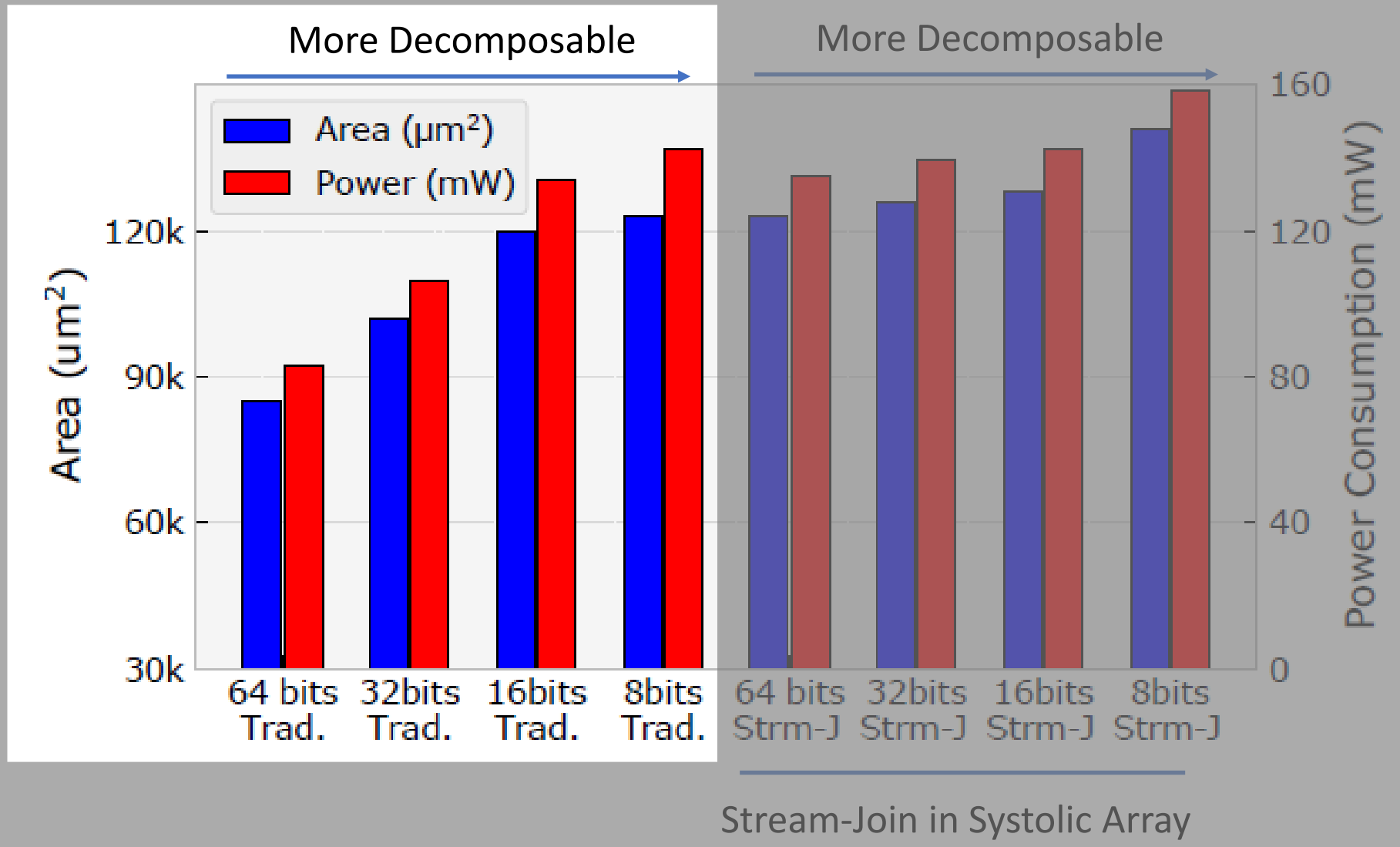
DGRA Switch: same external interface but splits i/p and o/p.



DGRA Processing Element: Decomposed to fine-grained PEs



Cost of Decomposability & Stream-Join



Remaining Challenges

- Generality
  - What about other forms of irregularity? (task-based?)
- Programmability Challenges
- Workload balance (1) same amount of work in each core 2) efficient use of available on-chip memory (global addressing helps in this case)
  - Partitioning of Computation/Memory (Locality & Parallelism)
  - Programming in low-level intrinsic (dataflow compute & stream mem)
- Virtualization/integration with CPU

Domain-agnostic comparison points