Accelerators and Architectural Specialization

15-740 SPRING'18

NATHAN BECKMANN

Today: Accelerators & specialization

Trends towards increasing architectural specialization

- Advantages
- Challenges
- Why now?

Accelerator case studies

- Deep learning x2
- Graphs

Analysis & forecasting

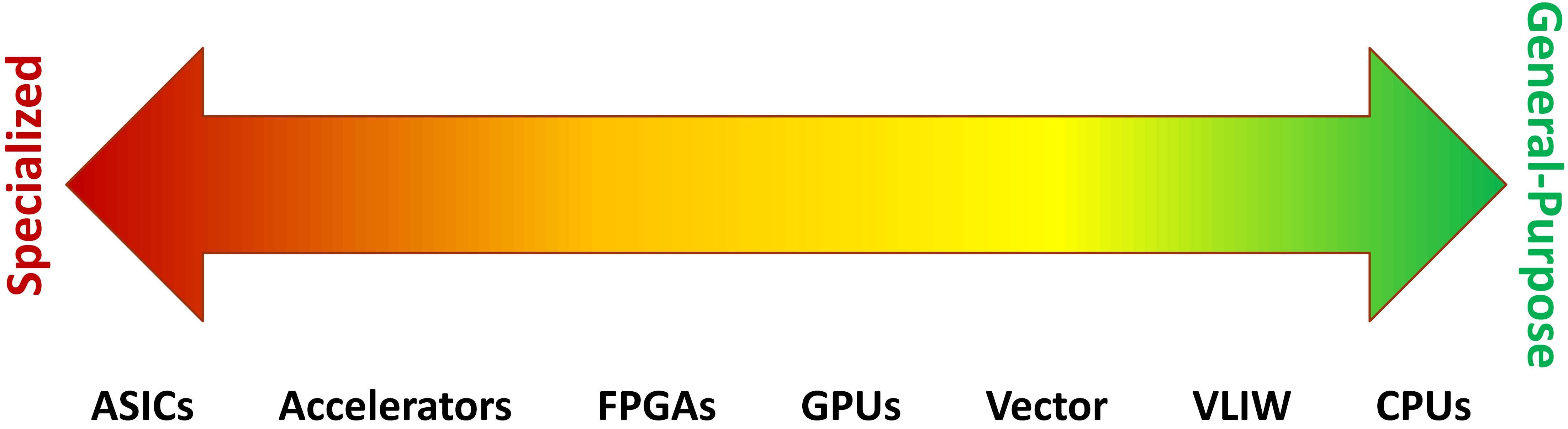
- Specialization’s real benefits & how much is needed
- What computing may look like in 5-10 years

Why specialization?

What is specialization?

Architectures designed with a specific class of computations in mind

- Optimizations that only make sense for the target applications
- Sometimes “fixed-function” – i.e., only run target applications – but usually some configurability / programmability
- Fuzzy where exactly to draw the line



Specialized

General-Purpose

ASICs **Accelerators** **FPGAs** **GPUs** **Vector** **VLIW** **CPUs**

** — Actual order will depend on application*

Ex: Modest specialization

IMP: Indirect memory prefetcher [Yu et al, MICRO'15]

```
for (i = 0; i < N; i++)
    if (A[B[i]] > 0)
        X += A[B[i]]
```

IMP prefetches indirect memory references

- Detects when LD addresses match data elsewhere in memory
- E.g., A[B[i]] is a function of the B[i] in memory
- Prefetches B[i] to then prefetch A[B[i]]

Lets indirect memory apps saturate memory bandwidth

…But complex: Requires a *reverse TLB* to detect access pattern (why?)

Is this specialization or just an optimization? Line is fuzzy…

Ex: Moderate specialization

Bespoke Processors for Applications with Ultra-Low Area and Power Constraints

[Cherupalli et al, ISCA'17]

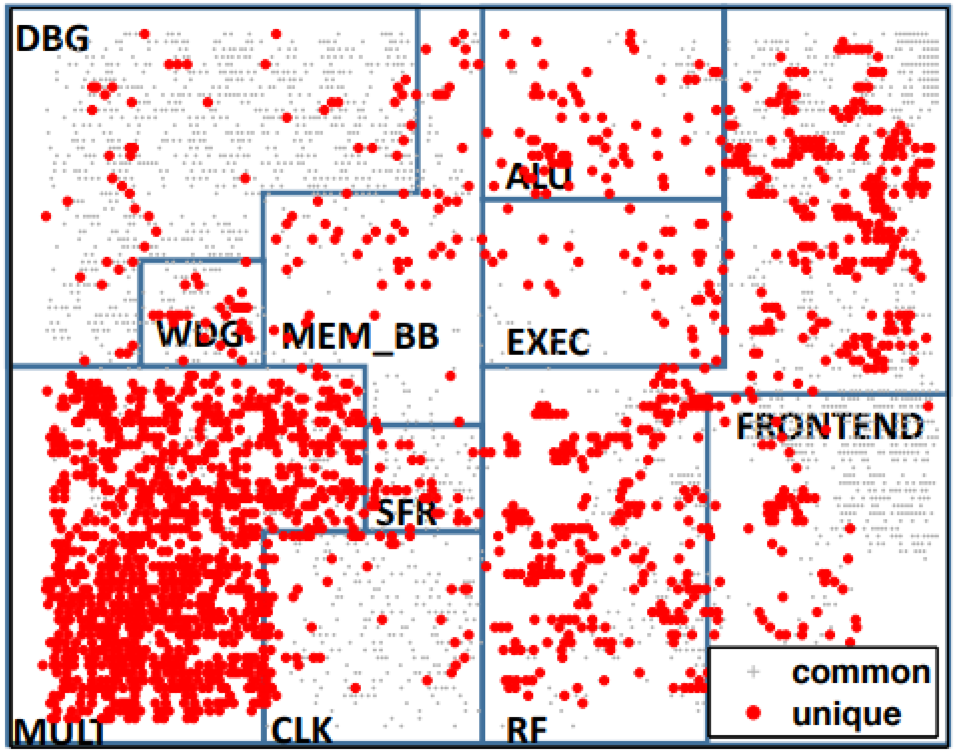
Profile applications and see which gates are used

Remove everything else from your processor

Saves 62% area and 50% power

Fully automatic

Leverages verification & design of a baseline processor



(b) binSearch

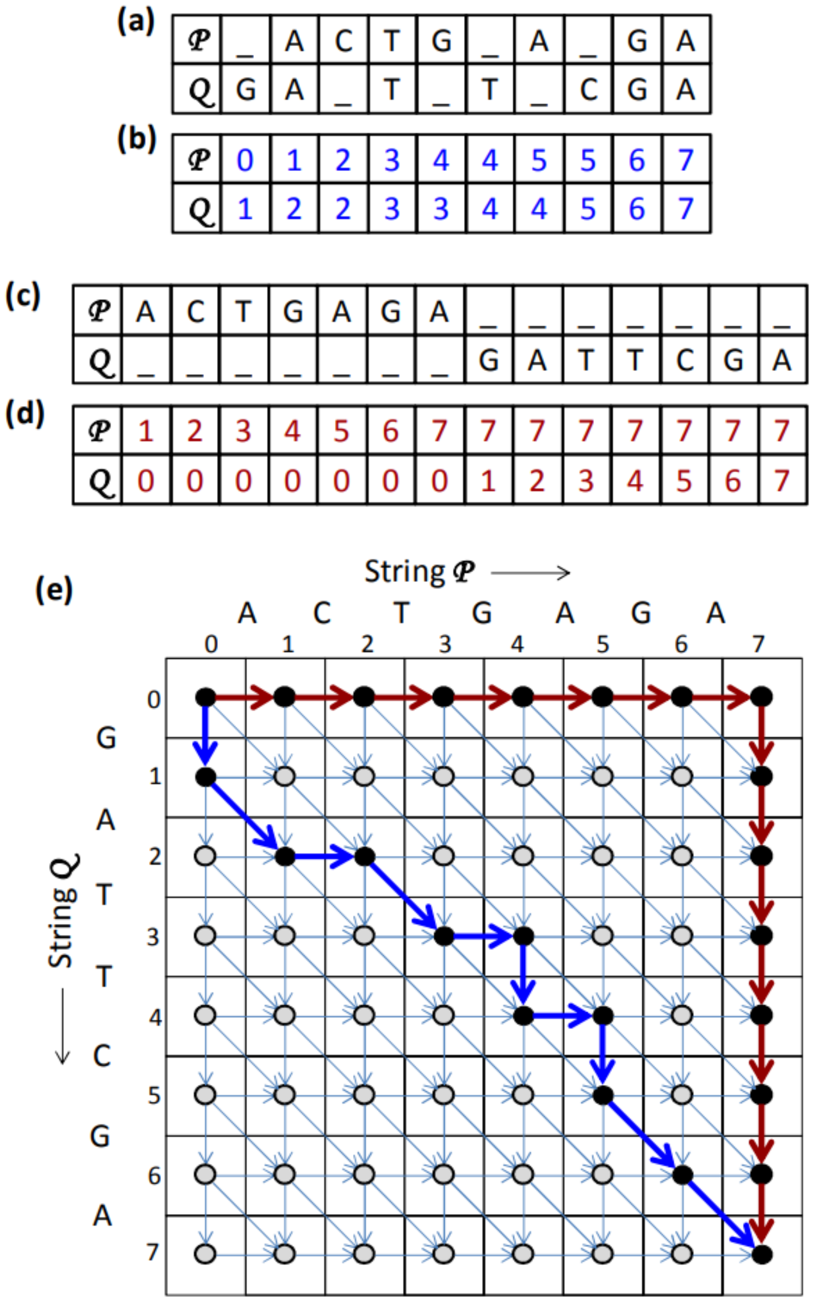
Ex: Extreme specialization

"Race logic" [Madhavan et al, ISCA'14]

Compute shortest path through a graph:

- Nodes mapped onto PEs
- PEs connected via on-chip network
- PEs signal each other, adding delay according to edge weight between source and destination
- The *delay* from source to destination gives the shortest path in the graph
- PEs very simple ➔ lots of PEs & fast

Computation primitive can solve several problems, e.g., DNA alignment:



(a)

$\mathcal{P}$	_	A	C	T	G	_	A	_	G	A
Q	G	A	_	T	_	T	_	C	G	A

(b)

$\mathcal{P}$	0	1	2	3	4	4	5	5	6	7
Q	1	2	2	3	3	4	4	5	6	7

(c)

$\mathcal{P}$	A	C	T	G	A	G	A	_	_	_	_	_	_	_
Q	_	_	_	_	_	_	_	G	A	T	T	C	G	A

(d)

$\mathcal{P}$	1	2	3	4	5	6	7	7	7	7	7	7	7	7
Q	0	0	0	0	0	0	0	1	2	3	4	5	6	7

Figure 1. (a, c) Two possible alignments between strings $\mathcal{P}$ and Q and (b, d) their corresponding alignment matrixes and (e) edit graph.

What is specialization?

BENEFITS / OPPORTUNITIES

Hardwired, low-power control

Custom functional units

Custom & **direct** communication

- Not through registers / cache

Custom memory system

Extreme parallelism using app knowledge

➔ Energy & area spent on **useful work**

WEAKNESSES / CHALLENGES

Scope – how many programs can my chip run?

- Tension between efficiency & generality
- “Amdahl’s Law of specialization” – is it better to speedup 1% of apps by 100× or all apps by 1%?

System integration

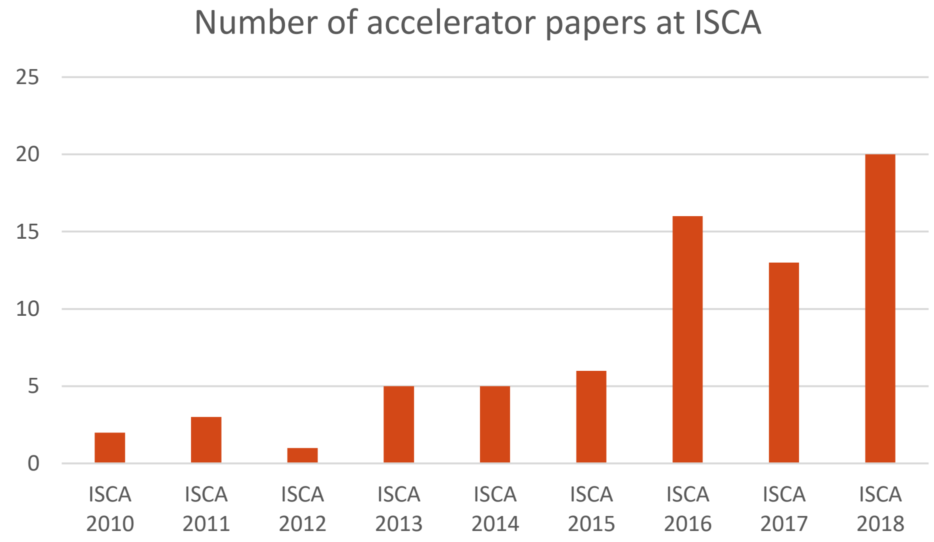
- How do users know about & use an accelerator?
- Do accelerators & cores communicate?
- One-off solutions vs. general framework

Test & design costs – hardware is hard!

Why is specialization relevant **now**?

Technology trends & “dark silicon”

- Power not decreasing, transistor counts are
- ➔ **Cannot toggle all transistors anymore**


Number of accelerator papers at ISCA

* – its becoming hard to find the real accelerator papers because everyone says they “accelerate” something now

Limits of parallelism & Amdahl’s Law

- Getting good performance out of multicore is hard
- Specialization gives 100× perf/energy for “free”

Important emerging workloads

- Especially deep learning!
- Two-three sessions on deep learning per conference in last three years

Case study:
Deep learning (1/2)

DianNao: An early DNN accelerators

Series of papers:

- DianNao [Chen et al, ASPLOS'14, Best paper]
- DaDianNao [Chen et al, MICRO'14, Best paper]
- PuDianNao [Liu et al, ASPLOS'15]
- ShiDianNao [Du et al, ISCA'15]

DNNs were becoming increasingly important & **large**

- Prior accelerators had focused on compute
- DianNao tackled **memory challenge**

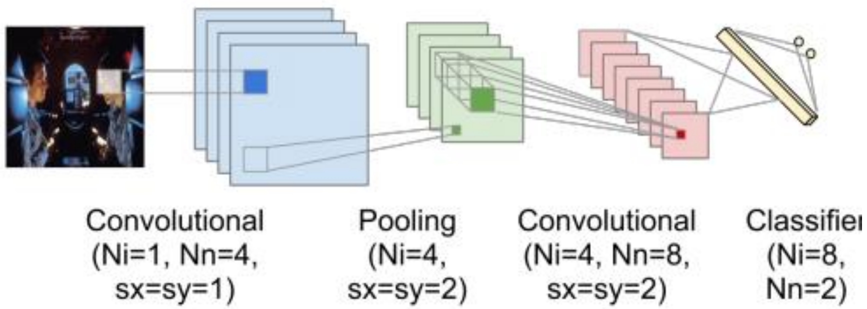
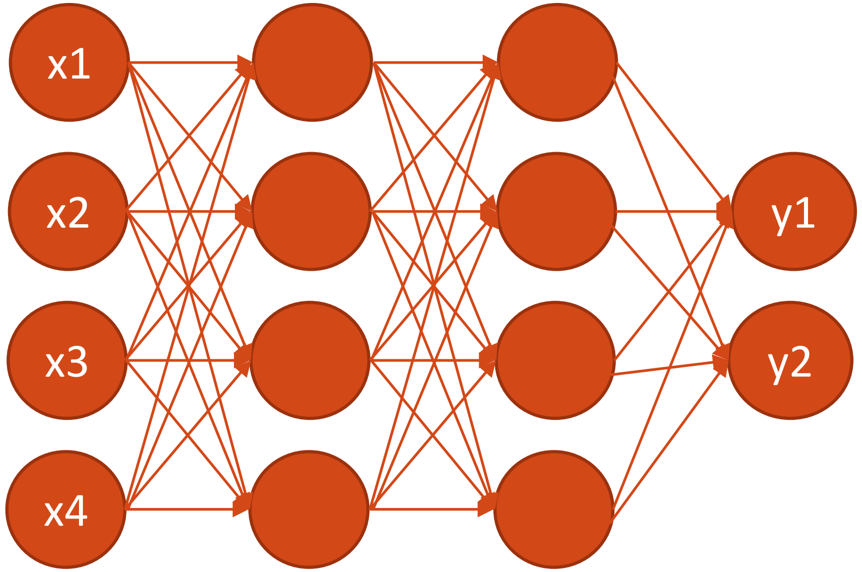
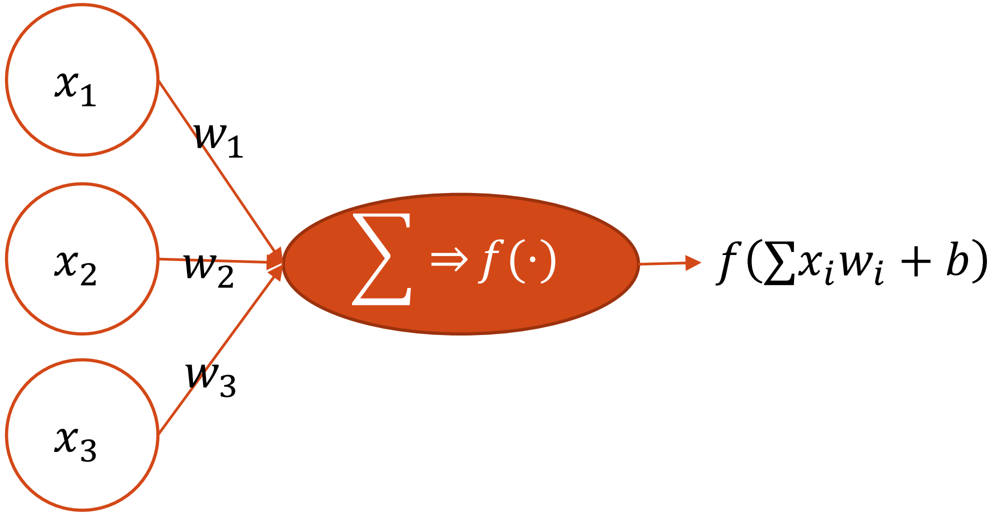


Figure 1. *Neural network hierarchy containing convolutional, pooling and classifier layers.*

Neural networks (NNs)

Simple artificial model of a brain





NN are *trained* to find the parameters w_i, b that minimize a loss function over some input set

Deep neural networks (DNNs) use many layers with different structure

Accelerating NNs

Directly represent neurons in hardware

Pros: Simplicity and performance

Cons: NN size limited by area

- Time multiplexing possible but expensive
- Only used for small perceptrons, not DNNs

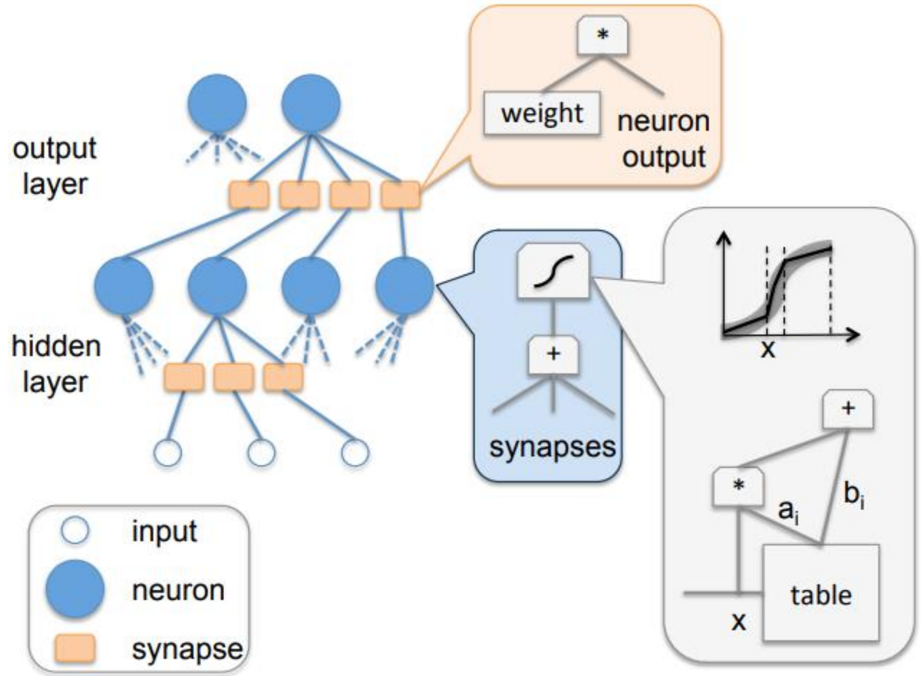


Figure 9. *Full hardware implementation of neural networks.*

DianNao architecture

Treat DNNs as dense linear algebra

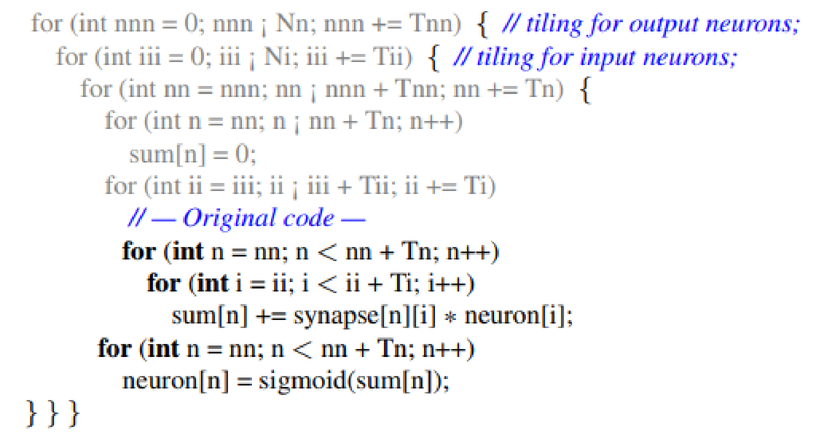
Simple, microcoded control

- Highly specialized “instructions”

Custom datapath for multiply + add + sigmoid

Custom scratchpads for inputs (NBin), outputs (NBout), and synapses (SB)

- No associative lookups, no conflicts
- Match line size to tile size for efficiency
- DMA issued as needed to rotate values in/out



```
for (int nnn = 0; nnn ¡ Nn; nnn += Tnn) { // tiling for output neurons;
  for (int iii = 0; iii ¡ Ni; iii += Tii) { // tiling for input neurons;
    for (int nn = nnn; nn ¡ nnn + Tnn; nn += Tn) {
      for (int n = nn; n ¡ nn + Tn; n++)
        sum[n] = 0;
      for (int ii = iii; ii ¡ iii + Tii; ii += Ti)
        // — Original code —
        for (int n = nn; n < nn + Tn; n++)
          for (int i = ii; i < ii + Ti; i++)
            sum[n] += synapse[n][i] ∗ neuron[i];
      for (int n = nn; n < nn + Tn; n++)
        neuron[n] = sigmoid(sum[n]);
} } }
```

Figure 5. *Pseudo-code for a classifier (here, perceptron) layer (original loop nest + locality optimization).*

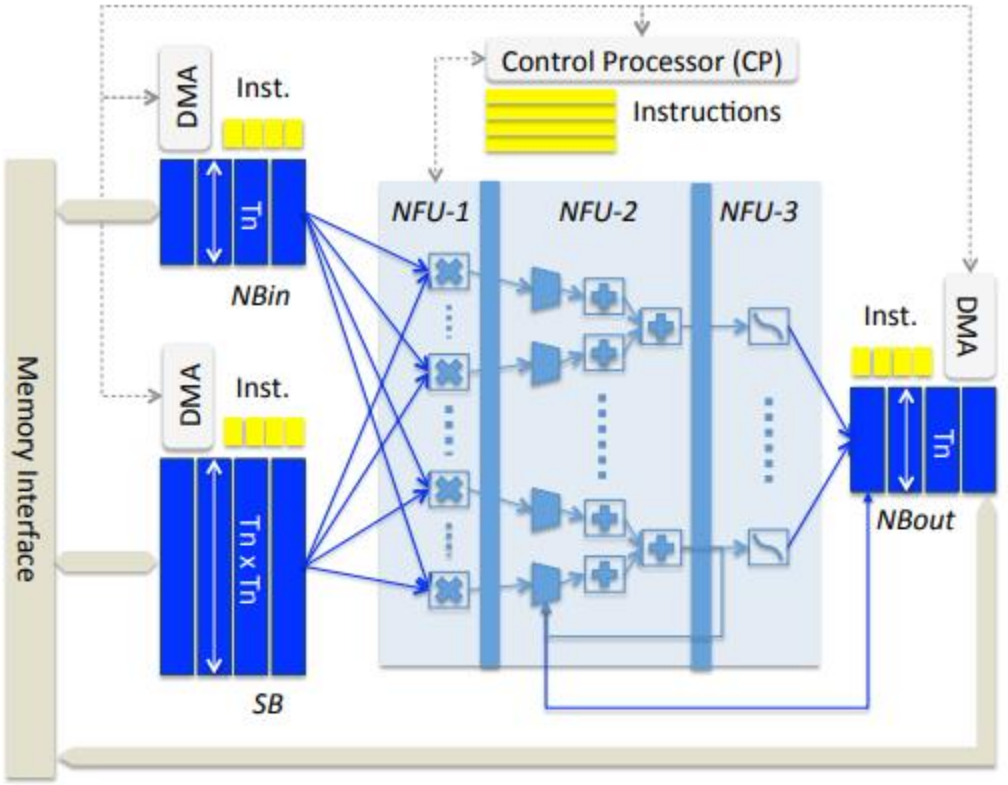


Figure 11. *Accelerator.*

DianNao results

110 × avg performance improvement vs SIMD

21 × avg energy improvement vs SIMD

- Much smaller improvement than other studies!
- **Memory dominates energy**

DaDianNao added large on-chip memories on multiple chips to improve energy by 150 ×

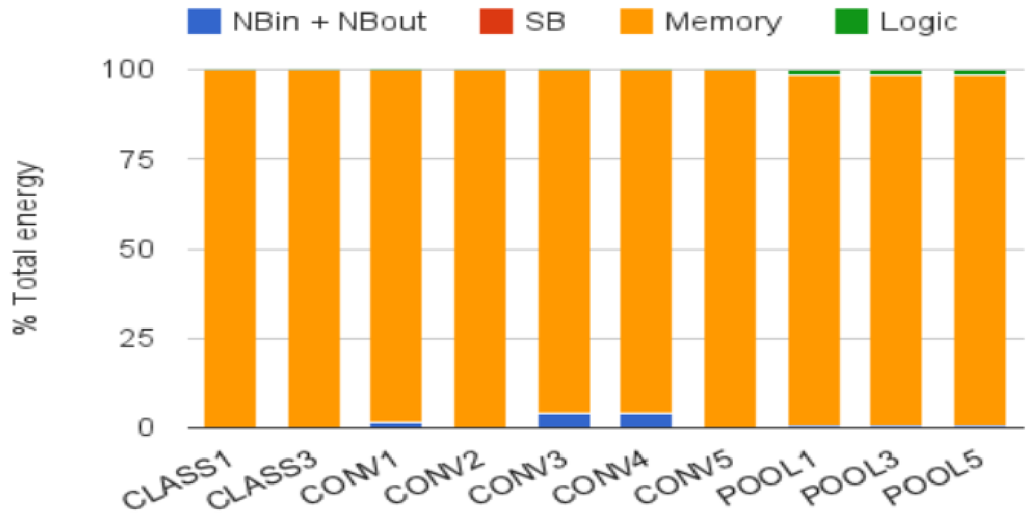


Figure 18. *Breakdown of accelerator energy.*

3mm²

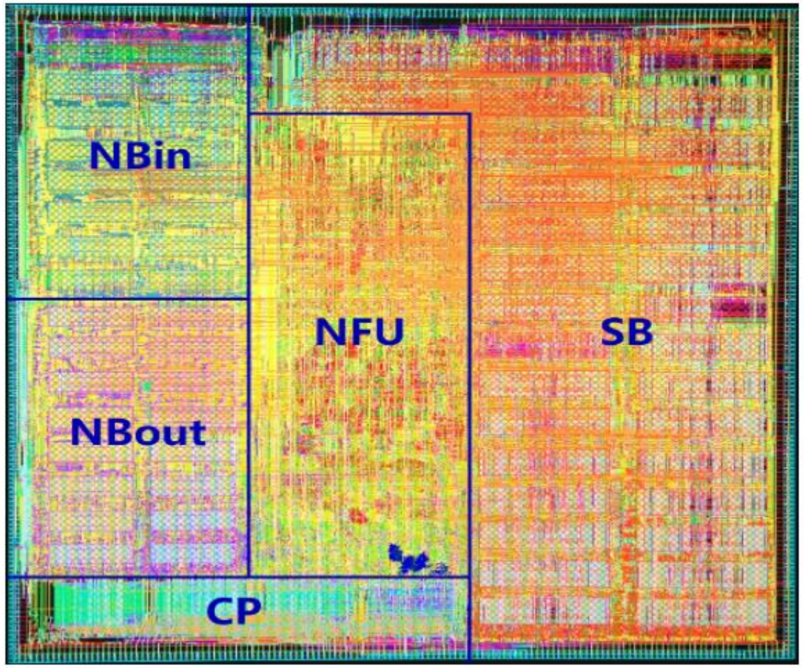


Figure 15. *Layout (65nm).*

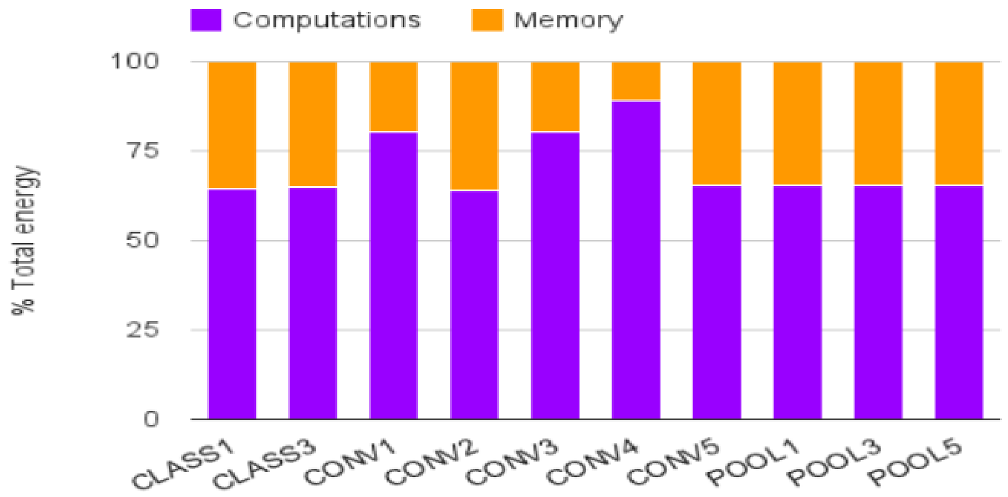


Figure 19. *Breakdown of SIMD energy.*

Case study:
Deep learning (2/2)

EIE: Sparse Neural Networks

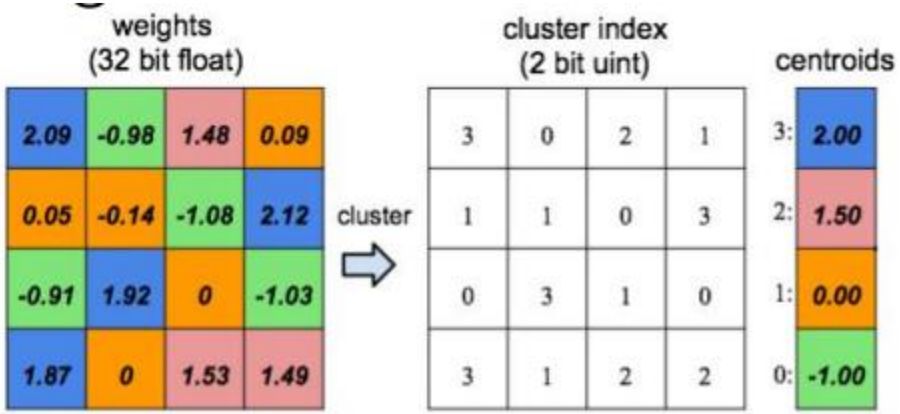
[Han et al, ISCA'16]

Also focuses on memory bottleneck

Observation: Weights are concentrated at a few values
- Use only **two bits** to represent weight
- Need higher precision than this
  ➔ use **codebook** to store $2^{2\text{ bits}} = 4$ higher-precision values

Observation: Most weights are close to zero
- **Prune** near-zero weights
  ➔ significantly less memory & compute needed!
- Turns NNs into **sparse linear algebra** computation
  ➔ irregular control & memory references

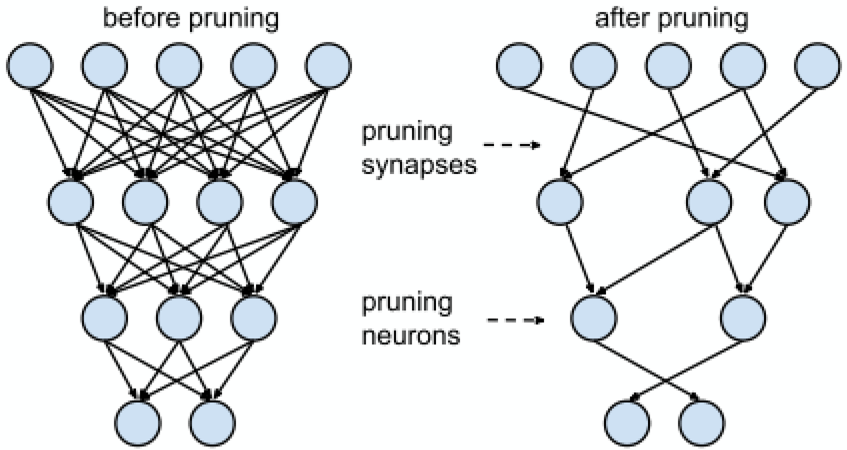
10-49× reduction in memory footprint



2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

3	0	2	1
1	1	0	3
0	3	1	0
3	1	2	2

3: 2.00
2: 1.50
1: 0.00
0: -1.00

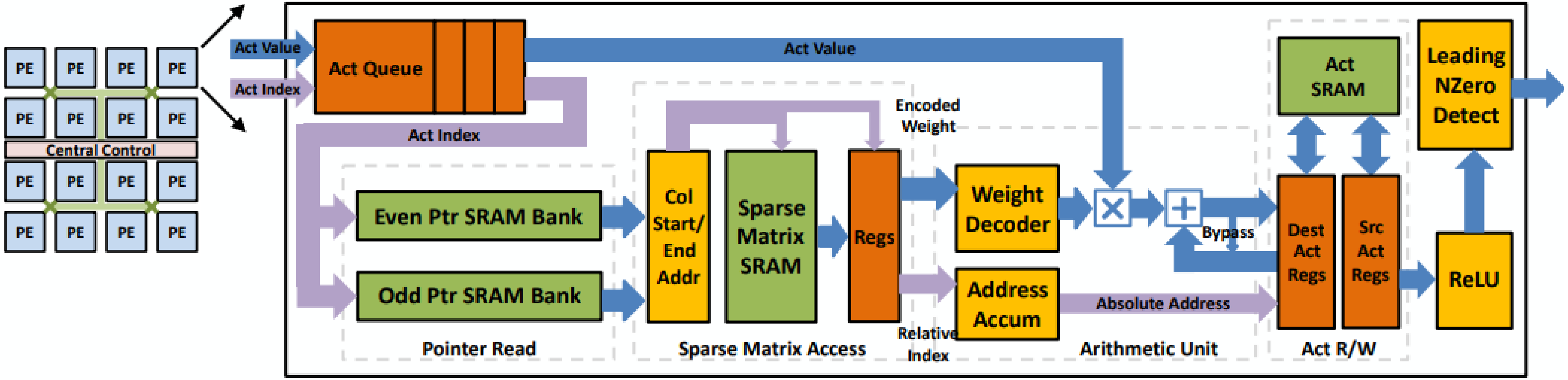


EIE architecture

Compression lets **all weights for a layer fit in on-chip SRAM**

➔ Large energy improvements

Hardware support for codebooks, run-length encoding, + sparse address calculation

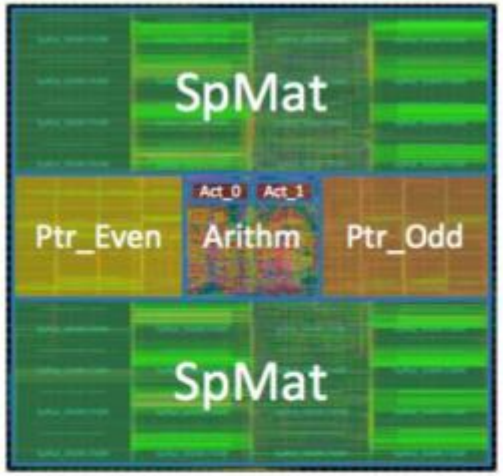


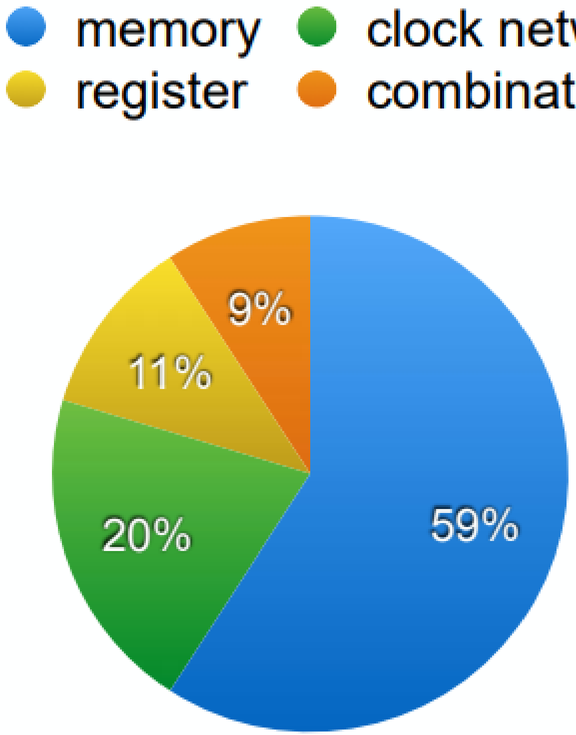
EIE results

41mm^2 @ 45nm (much bigger than DianNao)

Significantly less energy spent on memory accesses

Claims 24,000 × improvement vs CPU and 3,400 × improvement vs GPU w/out compression





Other work in deep learning

Industry – Google TPU [Jouppi et al, ISCA'16]

- Same architectural principles as DianNao

Convolutional layers: Small weights ➔ compute matters more

- Systolic arrays & detailed dataflow analysis, e.g., *Eyeriss* [Chen, Emer, and Sze; ISCA'16]
- Sparse convolutions [Parashar et al, ISCA'17]

FPGAs [Sharma et al, MICRO'16]

Much, much more…

Case study: Graphs

Graphicianado

[Ham et al, MICRO’16, Best paper]

Graph analytics is important

- E.g., PageRank for web search

Graphs have irregular memory accesses that limit their performance

- Little compute to do per vertex
- Memory latency-bound ➔ Low bandwidth utilization, cores mostly idle
- **Very inefficient on big, out-of-order cores**

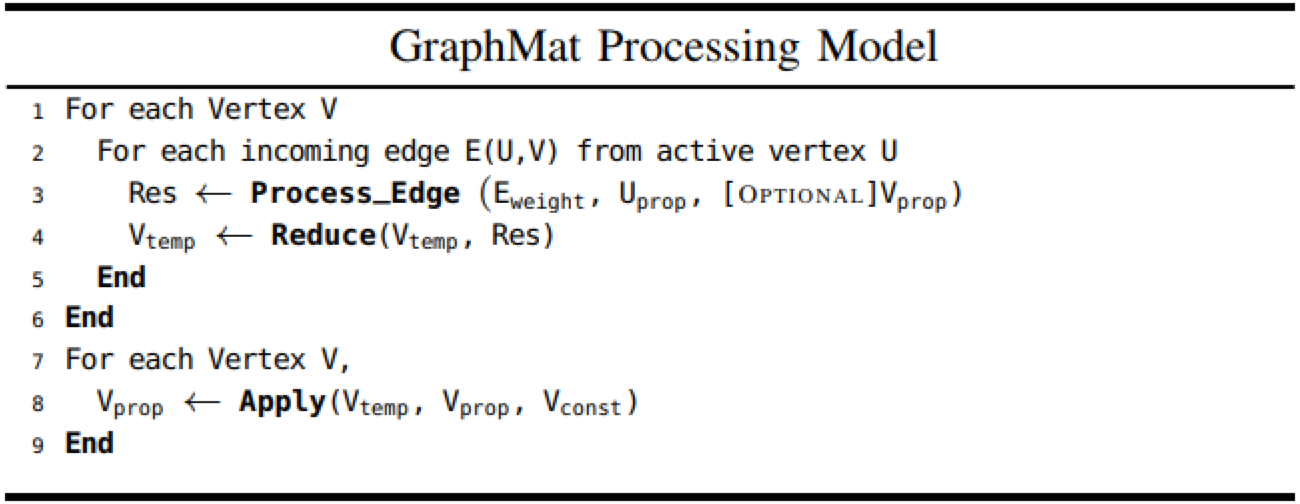
Graphicianado introduces a pipelined accelerator to keep memory busy & reduce energy spent on compute

GraphMat framework

First, loop over edges accumulating updates

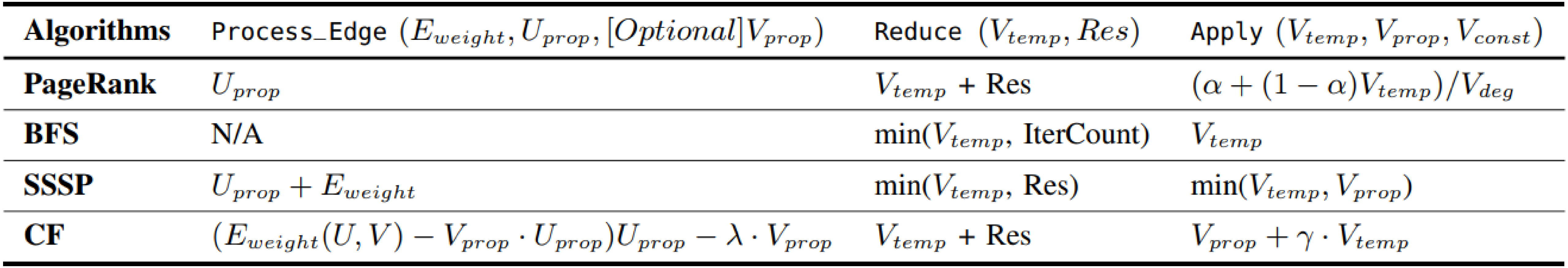
Then, loop over vertices applying the update

Covers many common algorithms



GraphMat Processing Model

```
For each Vertex V
  For each incoming edge E(U,V) from active vertex U
    Res ← Process_Edge ($E_{weight}$, $U_{prop}$, [OPTIONAL]$V_{prop}$)
    $V_{temp}$ ← Reduce($V_{temp}$, Res)
  End
End
For each Vertex V,
  $V_{prop}$ ← Apply($V_{temp}$, $V_{prop}$, $V_{const}$)
End
```

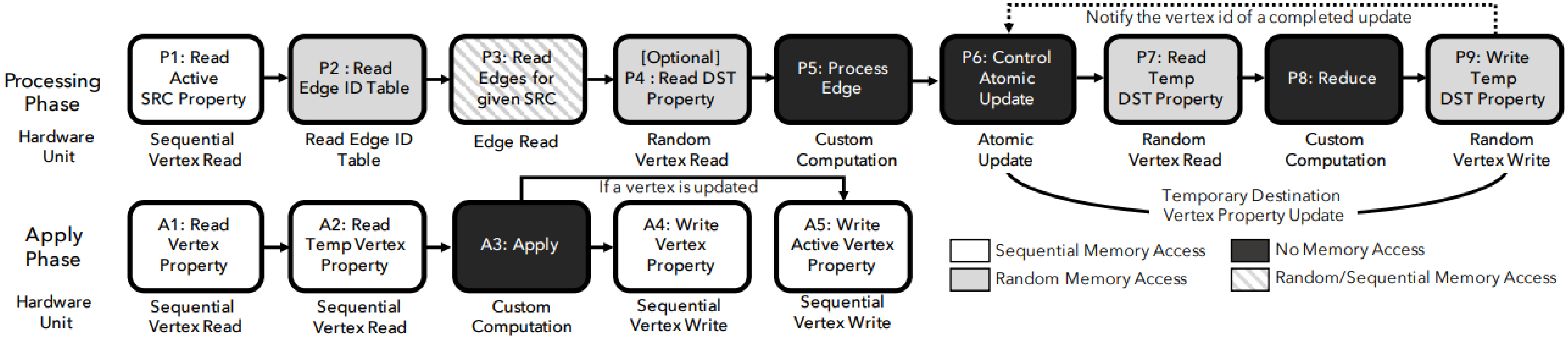


Algorithms	`Process_Edge` $(E_{weight}, U_{prop}, [Optional]V_{prop})$	`Reduce` (V_{temp}, Res)	`Apply` $(V_{temp}, V_{prop}, V_{const})$
PageRank	U_{prop}	V_{temp} + Res	$(\alpha + (1 - \alpha)V_{temp})/V_{deg}$
BFS	N/A	min(V_{temp}, IterCount)	V_{temp}
SSSP	$U_{prop} + E_{weight}$	min(V_{temp}, Res)	min(V_{temp}, V_{prop})
CF	$(E_{weight}(U,V) - V_{prop} \cdot U_{prop})U_{prop} - \lambda \cdot V_{prop}$	V_{temp} + Res	$V_{prop} + \gamma \cdot V_{temp}$

TABLE I: Example mapping of algorithms to programming model. For an edge $E = (U, V)$, U is the source vertex and V is the destination vertex.

Graphicianado datapath

Custom pipeline for each inner loop of GraphMat



Notify the vertex id of a completed update

Processing Phase

Hardware Unit

P1: Read Active SRC Property — Sequential Vertex Read

P2 : Read Edge ID Table — Read Edge ID Table

P3: Read Edges for given SRC — Edge Read

[Optional] P4 : Read DST Property — Random Vertex Read

P5: Process Edge — Custom Computation

P6: Control Atomic Update — Atomic Update

P7: Read Temp DST Property — Random Vertex Read

P8: Reduce — Custom Computation

P9: Write Temp DST Property — Random Vertex Write

Temporary Destination Vertex Property Update

Apply Phase

Hardware Unit

A1: Read Vertex Property — Sequential Vertex Read

A2: Read Temp Vertex Property — Sequential Vertex Read

A3: Apply — Custom Computation

If a vertex is updated

A4: Write Vertex Property — Sequential Vertex Write

A5: Write Active Vertex Property — Sequential Vertex Write

☐ Sequential Memory Access

▨ Random Memory Access

■ No Memory Access

▧ Random/Sequential Memory Access

Parallelized across multiple "streams"

Have to deal with some tricky issues

- E.g., hazards when vertex update is in-flight

Must re-synthesize design for each algorithm

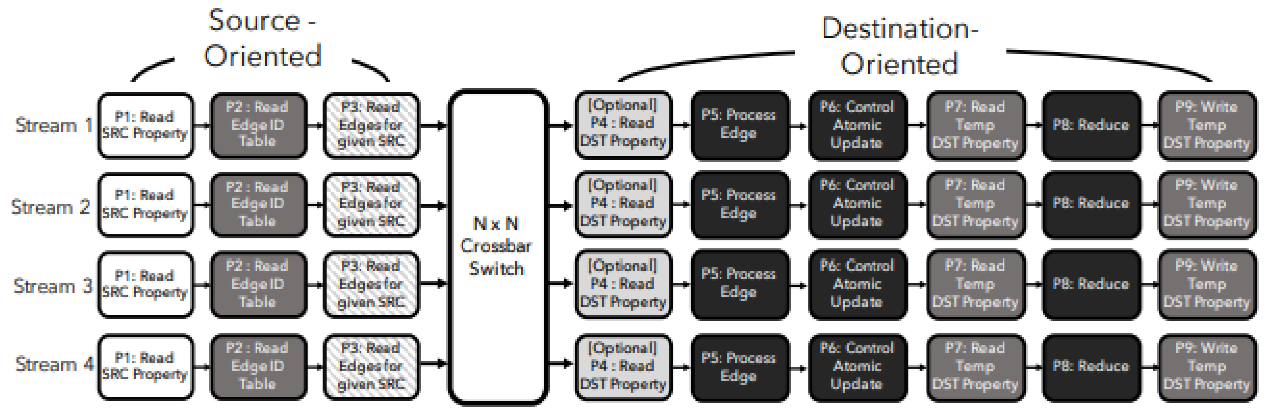


Fig. 7: Parallel implementation of Graphicionado. This diagram omits the *Apply* phase pipeline which is parallelized in a similar manner.

Graphicianado memory system

Graphicianado “slices” graph into many pieces that **fit in on-chip SRAM**

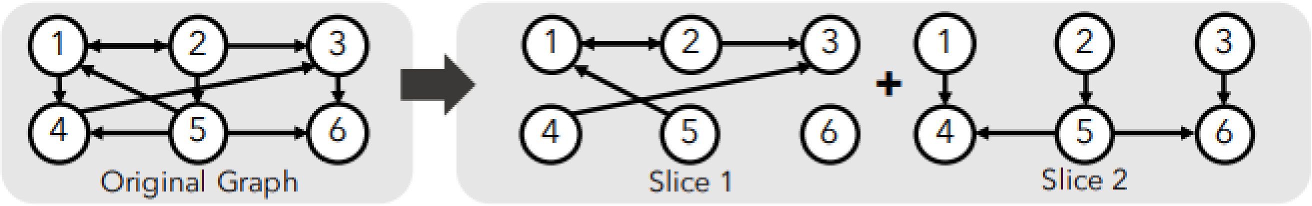


Fig. 10: Graph slicing Example.

(Hopefully) lots of reuse within a slice → most memory accesses are on-chip

Orchestrates DMA (similar to DianNao)

32MB on-chip scratchpad (eDRAM) with line size customized to algorithm

Other optimizations, e.g., perfect prefetching

Must re-synthesize design for each algorithm

Graphicianado evaluation

Synthezised in “proprietary sub-28nm” at 1GHz – no area #’s given

- This is unfortunately common for some fabs

$1.75 - 6.5 \times$ speedup vs CPU

$50 - 100 \times$ energy improvement

All energy goes into the scratchpad

Evaluation does not include main memory!

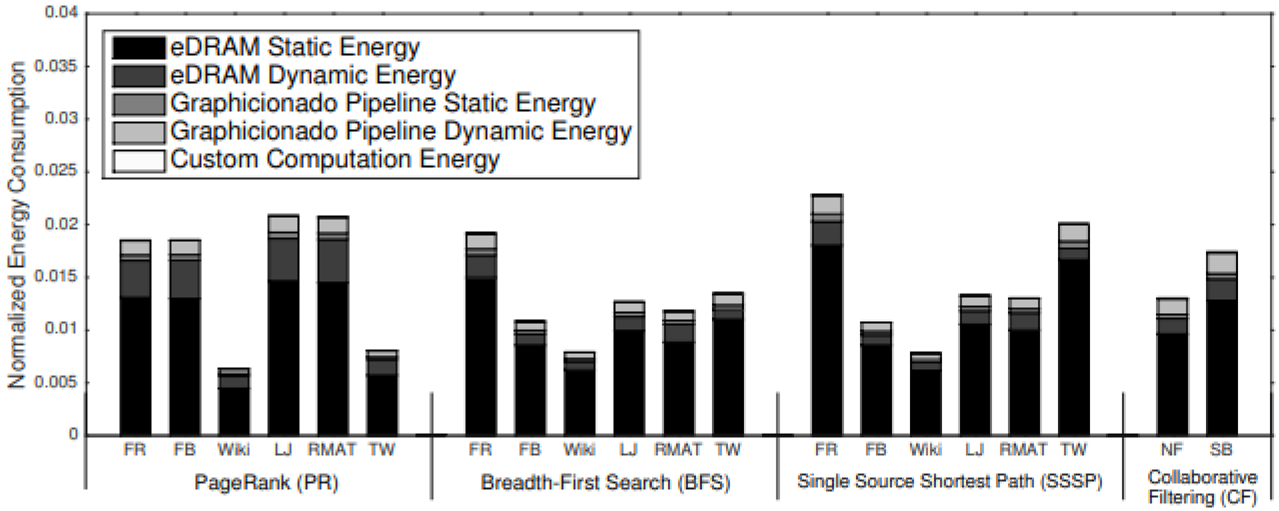


Fig. 17: Graphicionado energy consumption normalized to the energy consumption of Xeon processor running software graph processing framework.

† Custom Computation Energy contributes to less than 1% of the total energy

Do we really need specialization?

What are the benefits of specialization?

Domain specialization is generally unnecessary for accelerators [Nowatzki, IEEE Micro'17]

The "five Cs" of specialization:

1. Concurrency
2. Compute
3. Communication
4. **Caching**
5. Coordination / control

Claim: *Most of the* $\approx 100\times$ *benefit from specialization goes away when compared against a better baseline, programmable architecture that includes some of these optimizations.*

How much specialization is needed?

…but only ≈5% of energy goes towards FUs even on simple cores! [Horowitz, ISSCC’14 Keynote]

Case study: energy-efficiency of a H.264 encoder @ 720p [Hameed et al, ISCA’12]

- General-purpose core – $1\times$
- VLIW/Vector – $7\times$
- Custom, “fused” FUs – $10\times$
- “Magic” super-instructions – $180\times$
- ASIC – $500\times$

> *“The inescapable conclusion is that truly efficient designs will require application-specialized hardware.”*

Limitation: Only 16-wide vectors … not necessarily in conflict with [Nowatzki, IEEE Micro’17], or even with GPUs that get $\gg 10\times$ efficiency over CPUs

Paths forward

“Agile” hardware & “Productive” HDLs

Amdahl’s Law & Dark Silicon: The future is *not* 1000s of conventional cores

If specialized hardware is the way of the future, how do we cope?

Large research effort underway to make hardware easier to build

- Chisel from UC Berkeley
- PyMTL from Cornell
- Huge DARPA funding

OpenSource hardware movement

- RISC-V ecosystem from UC Berkeley
- OpenPiton from Princeton

…but still a long way to go

Reconfigurable architectures

Multicore's limitations does **not** necessarily imply rampant specialization

- Current SoCs are already heterogeneous: CPUs + GPUs + DSPs
- Maybe all we need is a DNN accelerator + one or two other programmable designs?

FPGAs making a comeback

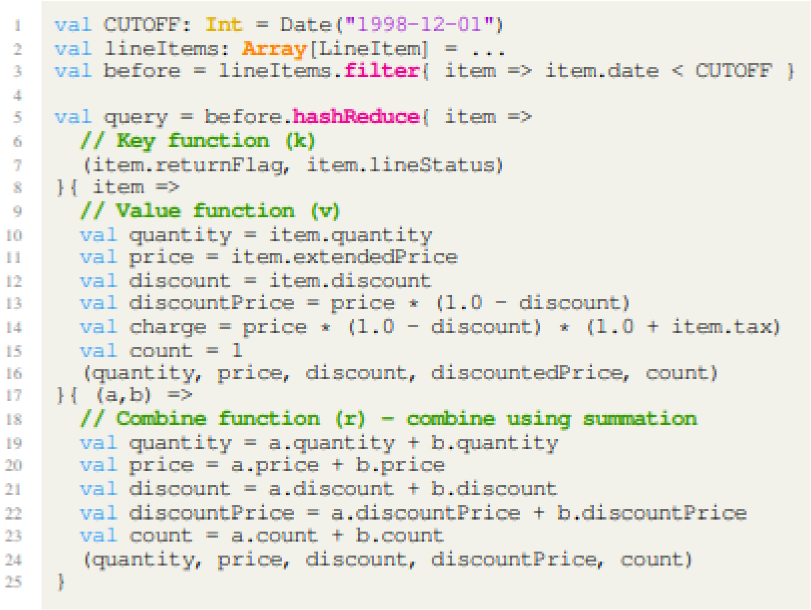
Renewed interest in **CGRAs** – coarse-grain reconfigurable arrays

- Programmable similar to FPGAs
- But with more hardened FUs / control / memories for efficiency
- E.g., *Plasticine* [Prabhakar et al, ISCA'17] and *Stream-dataflow acceleration* [Nowatzki, ISCA'17]
  - Plasticine has a nice programming story, too, building on a large body of work on domain-specific language (DSL) for parallel patterns

Plasticine

[Prabhakar et al, ISCA’17]

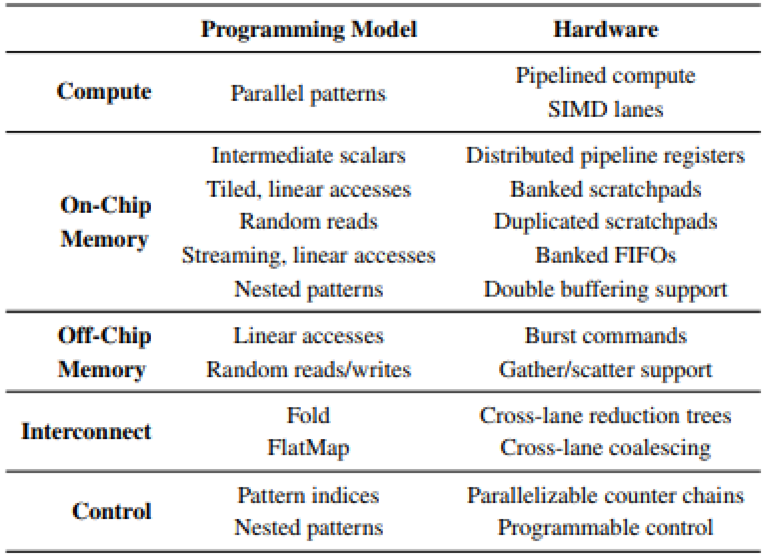
Plasticine maps high-level “parallel patterns” to hardware structures in a CGRA



```
val CUTOFF: Int = Date("1998-12-01")
val lineItems: Array[LineItem] = ...
val before = lineItems.filter{ item => item.date < CUTOFF }

val query = before.hashReduce{ item =>
  // Key function (k)
  (item.returnFlag, item.lineStatus)
}{ item =>
  // Value function (v)
  val quantity = item.quantity
  val price = item.extendedPrice
  val discount = item.discount
  val discountPrice = price * (1.0 - discount)
  val charge = price * (1.0 - discount) * (1.0 + item.tax)
  val count = 1
  (quantity, price, discount, discountedPrice, count)
}{ (a,b) =>
  // Combine function (r) - combine using summation
  val quantity = a.quantity + b.quantity
  val price = a.price + b.price
  val discount = a.discount + b.discount
  val discountPrice = a.discountPrice + b.discountPrice
  val count = a.count + b.count
  (quantity, price, discount, discountPrice, count)
}
```

Figure 2: Example of using filter (FlatMap) and HashReduce in a Scala-based language, inspired by TPC-H query 1.



	Programming Model	Hardware
Compute	Parallel patterns	Pipelined compute SIMD lanes
On-Chip Memory	Intermediate scalars	Distributed pipeline registers
	Tiled, linear accesses	Banked scratchpads
	Random reads	Duplicated scratchpads
	Streaming, linear accesses	Banked FIFOs
	Nested patterns	Double buffering support
Off-Chip Memory	Linear accesses	Burst commands
	Random reads/writes	Gather/scatter support
Interconnect	Fold	Cross-lane reduction trees
	FlatMap	Cross-lane coalescing
Control	Pattern indices	Parallelizable counter chains
	Nested patterns	Programmable control

Table 2: Programming model components and their corresponding hardware implementation requirements.

Plasticine

[Prabhakar et al, ISCA’17]

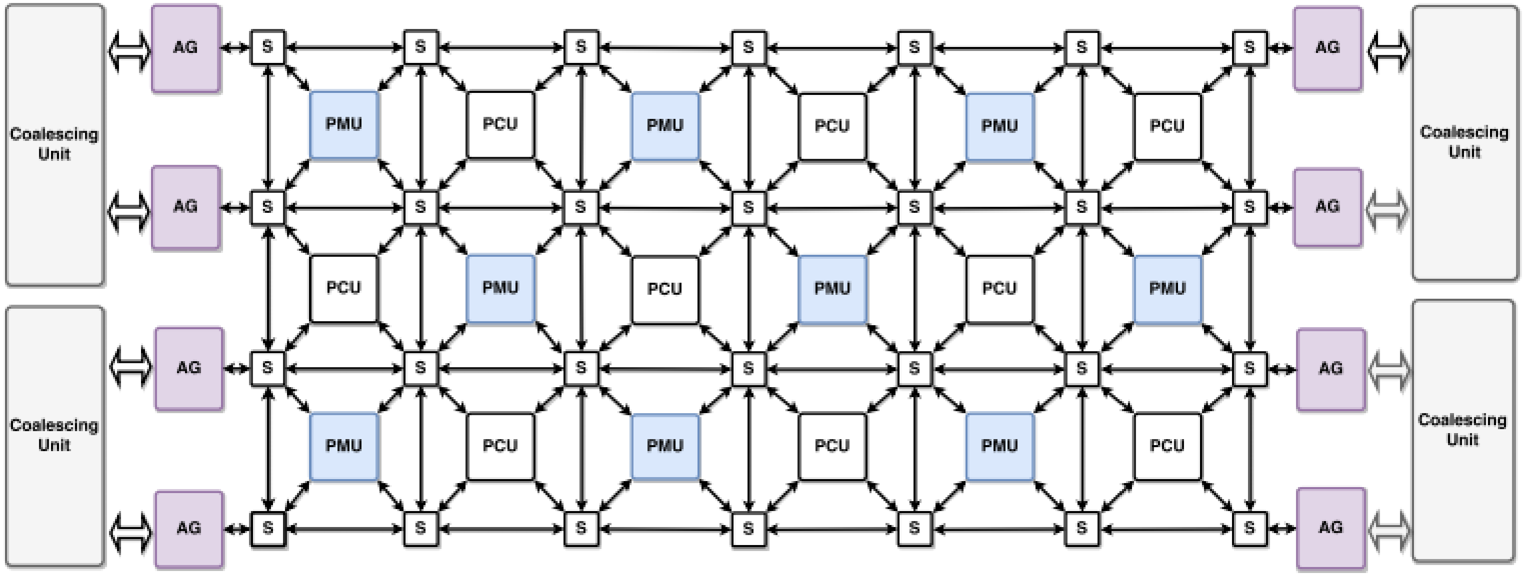


Figure 5: Plasticine chip-level architecture (actual organization 16 x 8). All three networks have the same structure. PCU:Pattern Compute Unit, PMU: Pattern Memory Unit, AG: Address Generator, S: Switch Box.

Plasticine

[Prabhakar et al, ISCA’17]

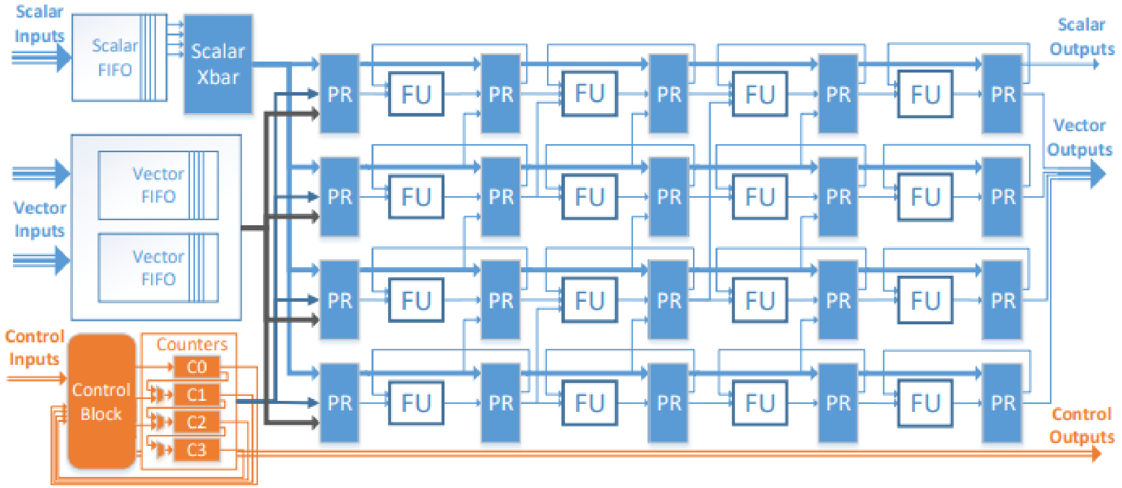


Figure 3: Pattern Compute Unit (PCU) architecture. We show only 4 stages and 4 SIMD lanes, and omit some control signals.

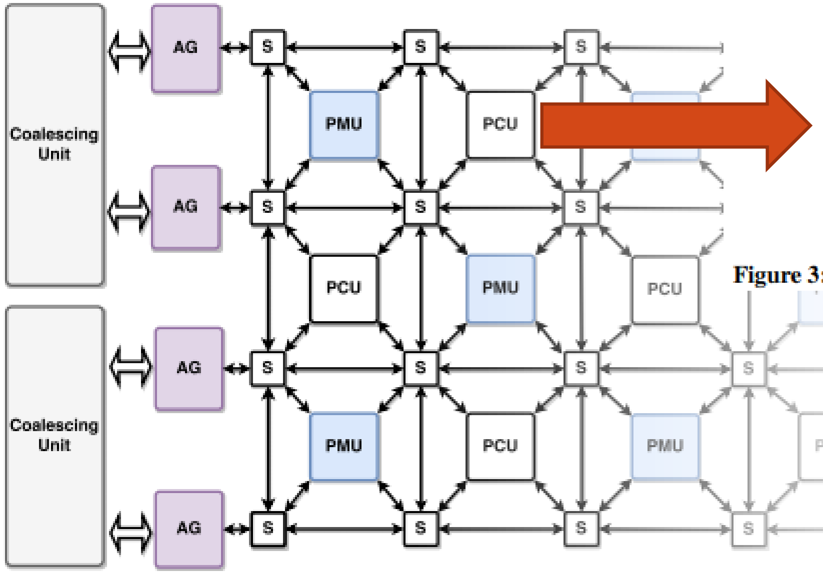


Figure 5: Plasticine chip-level architecture (actual organization 16 x 8). All three
PCU:Pattern Compute Unit, PMU: Pattern Memory Unit, AG: Address Generat

Each PCU implements a single, low-level parallel pattern

Pipelined SIMD engine w/ communication across lanes

Control via simple state machine w/ counters

Plasticine

[Prabhakar et al, ISCA’17]

Each PMU provides SRAM scratch pads & address calculation
(Address calculation in PCUs leaves lanes under-utilized)

Interconnect is statically configured (i.e., per app) to route data and control signals

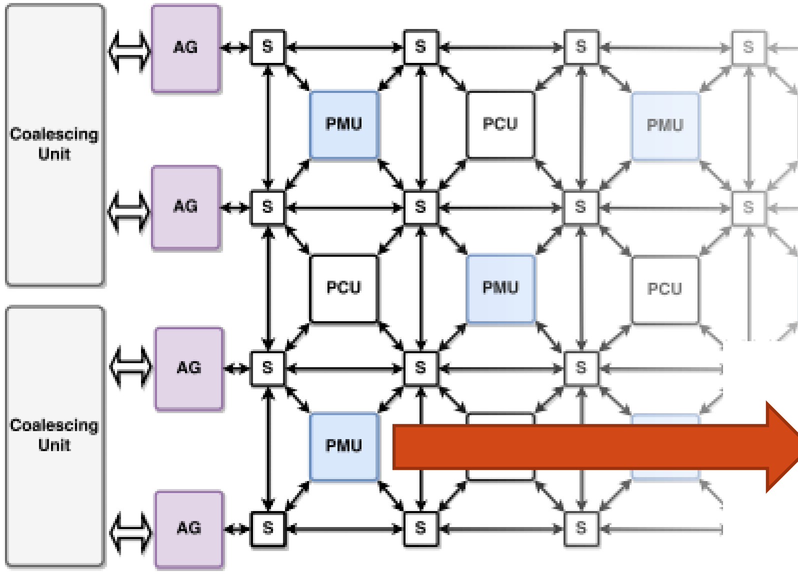


Figure 5: Plasticine chip-level architecture (actual organization
PCU:Pattern Compute Unit, PMU: Pattern Memory Unit, AG:

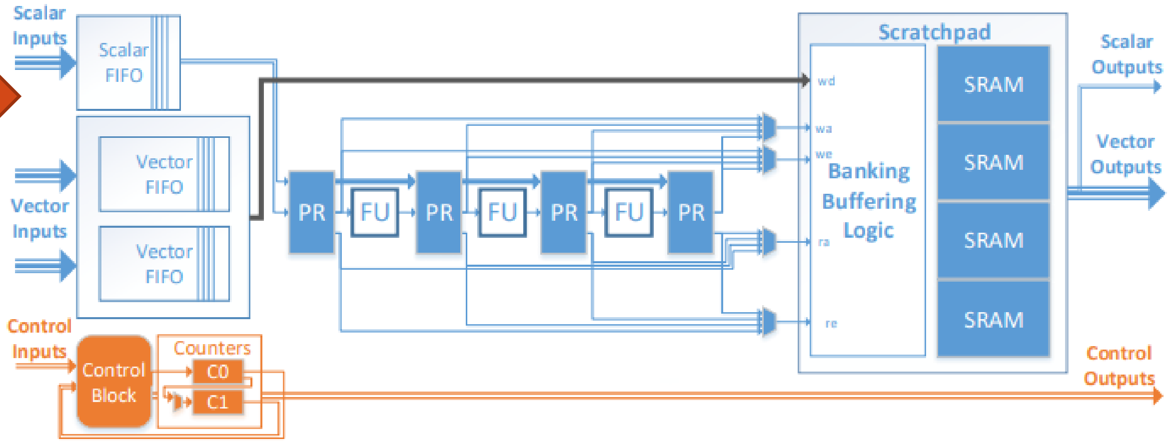


Figure 4: Pattern Memory Unit (PMU) architecture: configurable scratchpad, address calculation datapath, and control.

Plasticine Evaluation

112mm^2 in 28nm @ 1GHz

- 48% in compute units (pretty high!)
- 30% in memory units
- 16% in interconnect
- 5% in memory controller
- Can change balance of PCUs/PMUs as needed (but only at design time)

Reconfigurability costs estimated $11\times$ area vs. ASIC design, on average

$1.5-77\times$ energy improvement vs. FPGA (depends heavily on how well app maps to FPGA)

- More memory available in PMUs
- More efficient compute in PCUs

Summary: Accelerators & specialization

Several trends point to diminishing returns from general-purpose architectures

Specialization promises $\approx 100\times$ improvement in perf/energy

…But comes with major challenges

- What applications to specialize for?
- How to cope with design costs?
- Multicores were too hard to use effectively, and somehow accelerators are supposed to help?

Cynical perspective: Moore's Law made computer science complacent

"Why worry about efficiency? Everything will be $2\times$ better in a year…"

Moore's Law is over! Now the hard work begins!