

# Accelerators and Architectural Specialization

---

15-740 FALL'19

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

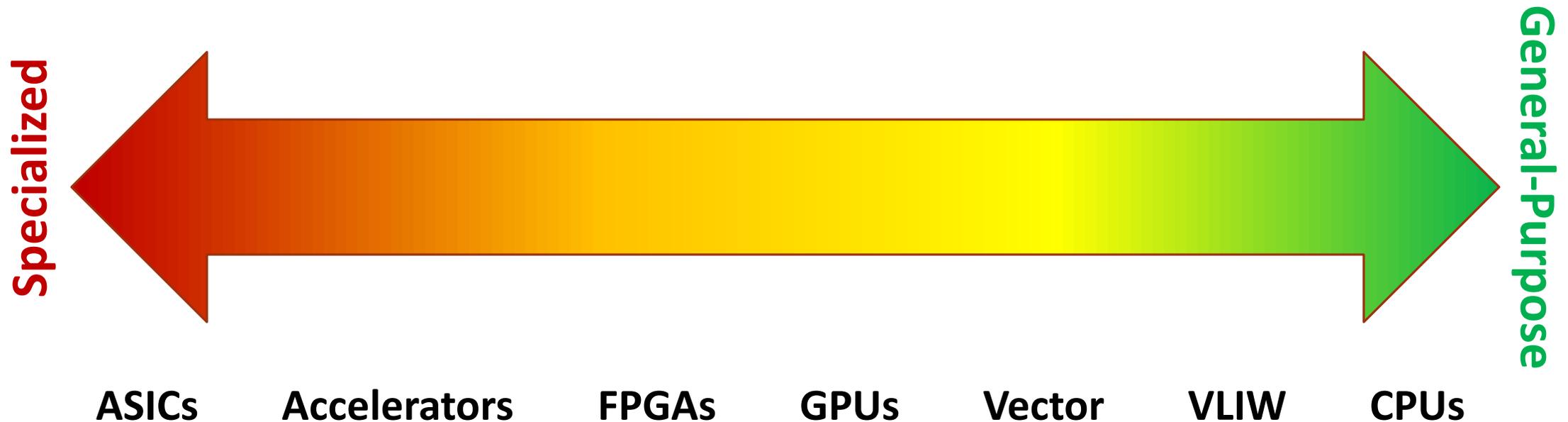
# Why specialization?

---

# What is specialization?

Architectures designed with a specific class of computations in mind

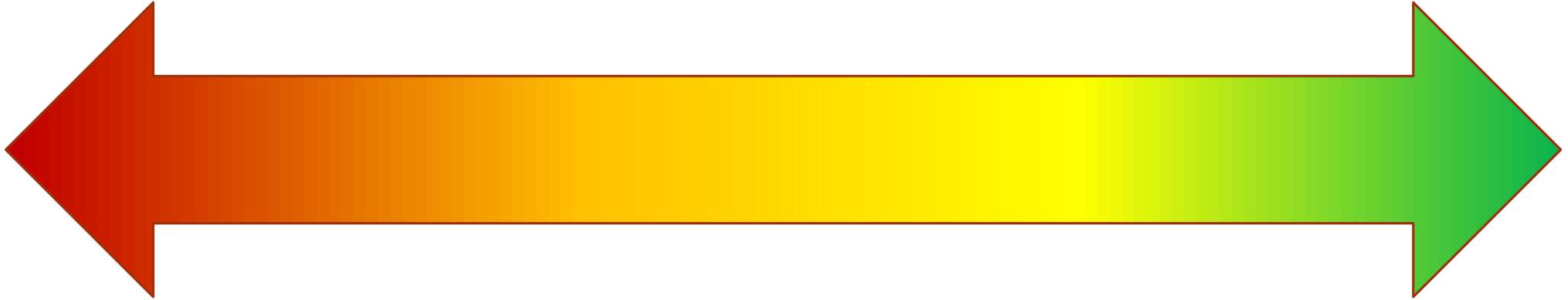
- Optimizations that only make sense for the target applications
- Sometimes “fixed-function” – i.e., only run target applications – but usually some configurability / programmability
- Fuzzy where exactly to draw the line



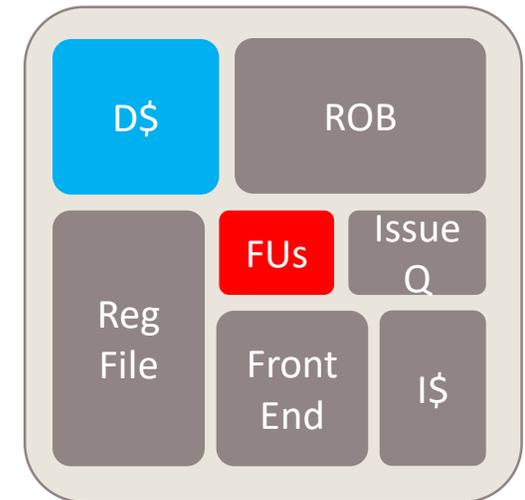
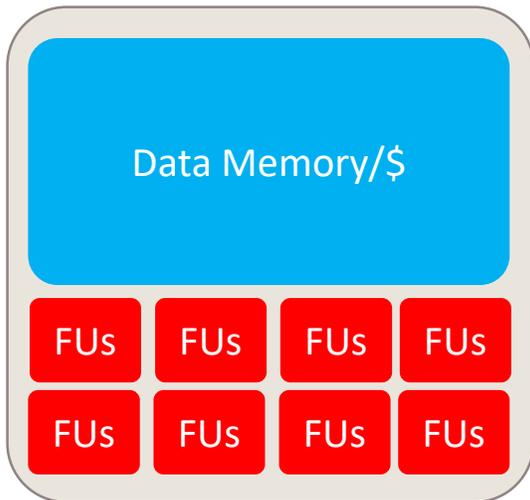
*\* — Actual order will depend on application*

# Why specialize?

Specialized



General-Purpose



# How much does specialization help?

---

Only  $\approx 5\%$  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 ×
- VLIW/Vector – 7 ×
- Custom, “fused” FUs – 10 ×
- “Magic” super-instructions – 180 ×
- ASIC – 500 ×

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

Wait a second – GPUs get  $\gg 10 \times$  efficiency gains!

- Limitation: Only considers 16-wide “SIMD extension”-style vectors

# What are the benefits of specialization?

---

The “five Cs” of specialization:

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

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

- (This paper claims that 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...more on this later)

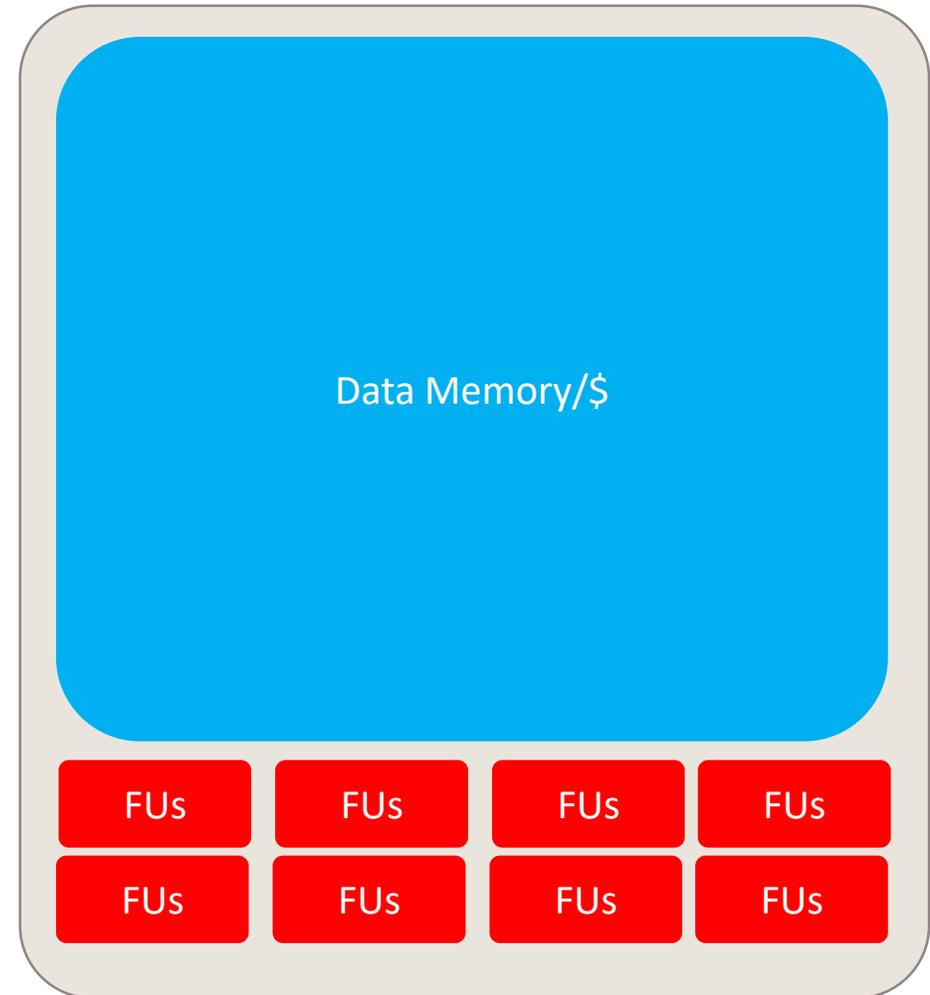
# Sketch of an accelerator

*Accelerator  $\approx$  Giant on-chip memory with a dash of custom logic on the side.*

1. Custom memory that dominates area
  - Optimized for application access patterns
  - Banking, word size, addressing all highly tuned
  - Idea: Keep all data on-chip, directly accessed
2. Array of simple PEs
  - Simple, hard-wired control – only solve one problem!
  - Parallelism across many PEs
  - Each PE directly implements required ops & little else

Important theme: Algorithm-accelerator **co-design**

- Fit data into on-chip scratchpad
- Map algorithm onto array of simple PEs



# Why is specialization relevant now?

## Technology trends & “dark silicon” [Esmailzadeh et al, ISCA’11]

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

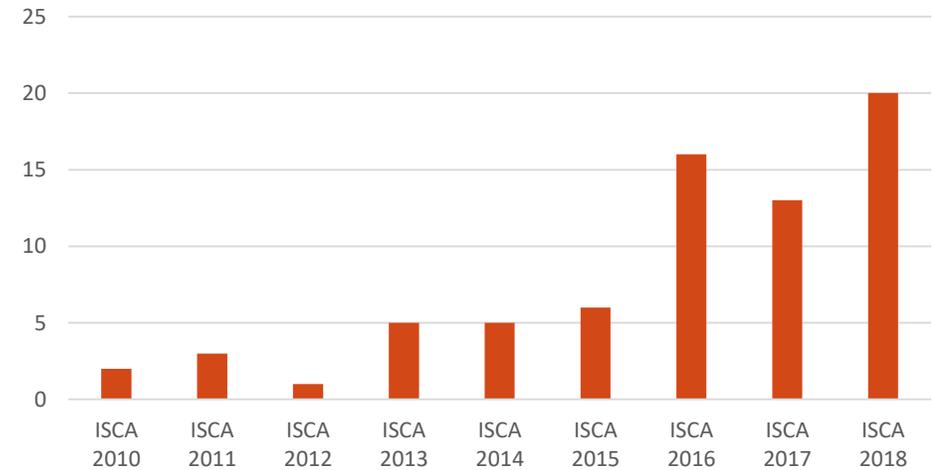
## Limits of parallelism & Amdahl’s Law

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

## Important emerging workloads

- Especially deep learning!

Number of accelerator papers at ISCA



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

# A 100B transistor chip in 2025:

CPUs for non-performance-critical code

Multicore for task/thread-parallel code

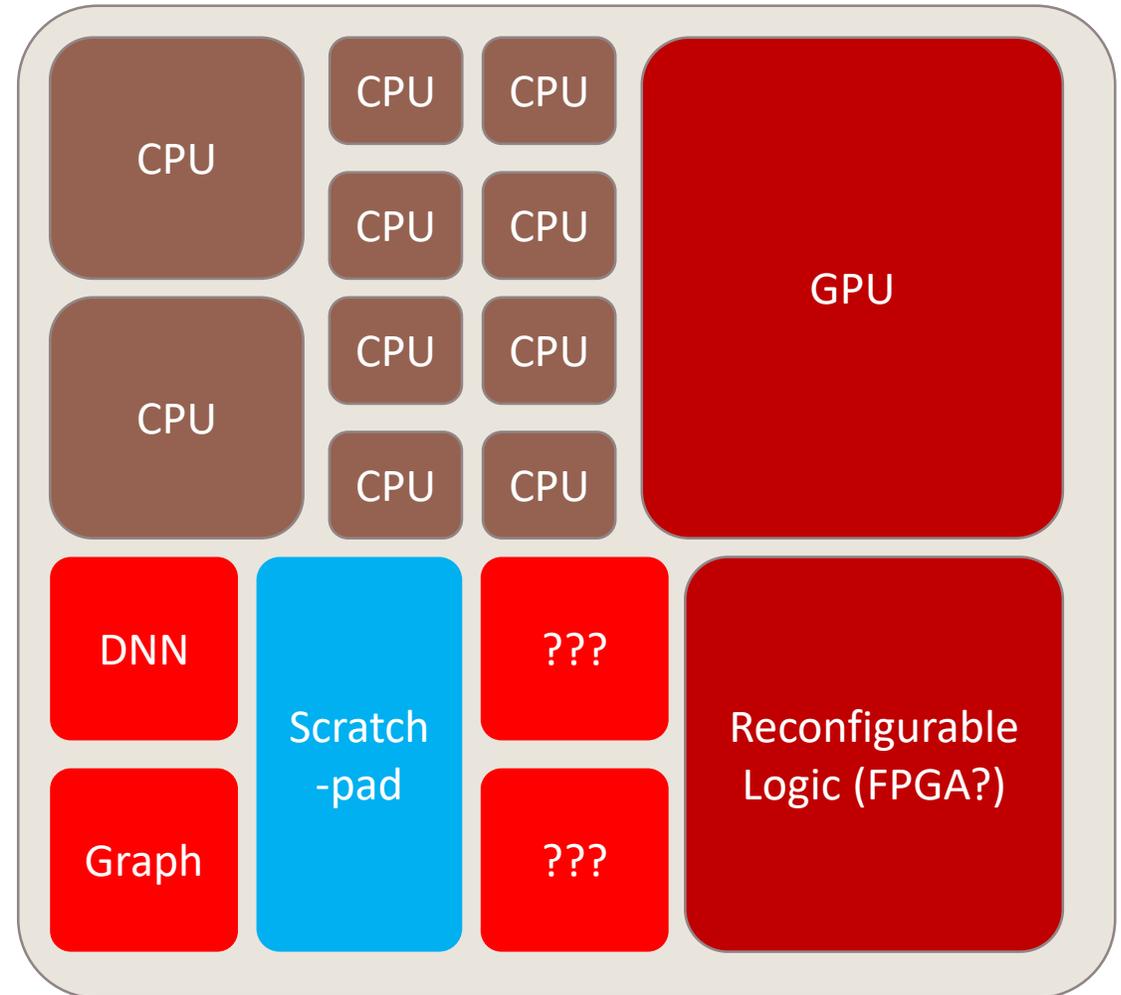
Beefy CPUs for sequential code

Accelerators for important applications

- Share memory that dominates area?

GPUs / FPGAs for other highly-parallel code

*System design is an optimization puzzle over the (weighted) application space*

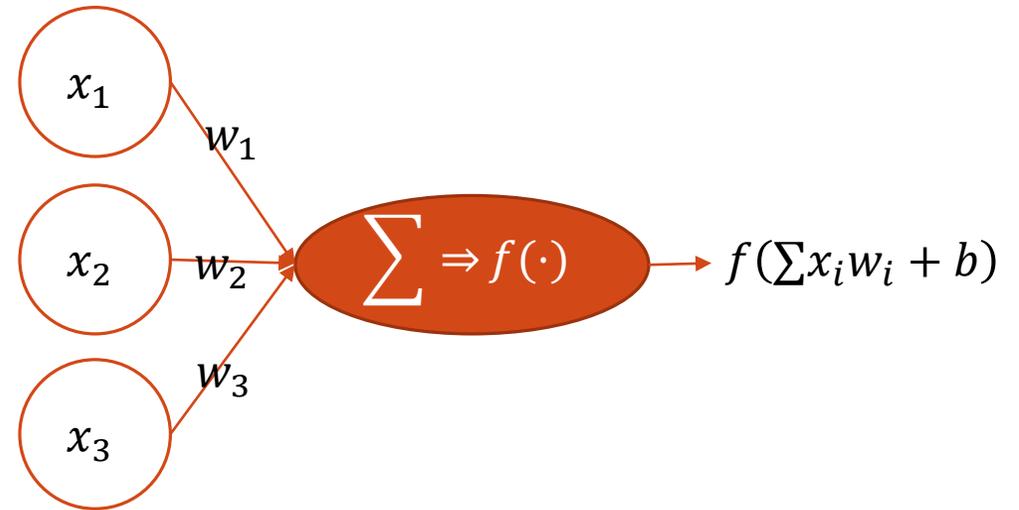
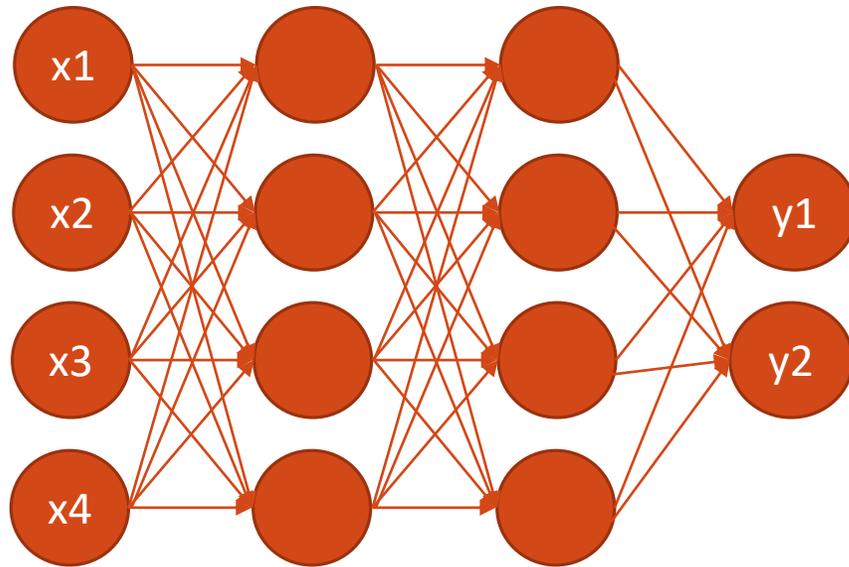


# Case study: Deep learning (1/2)

---

# 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

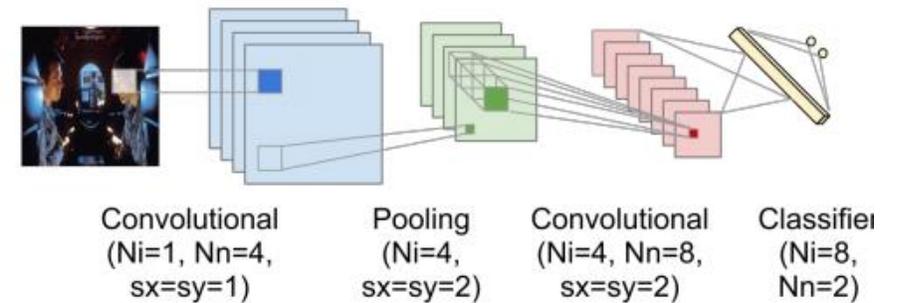
# DianNao: An early DNN accelerator

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.

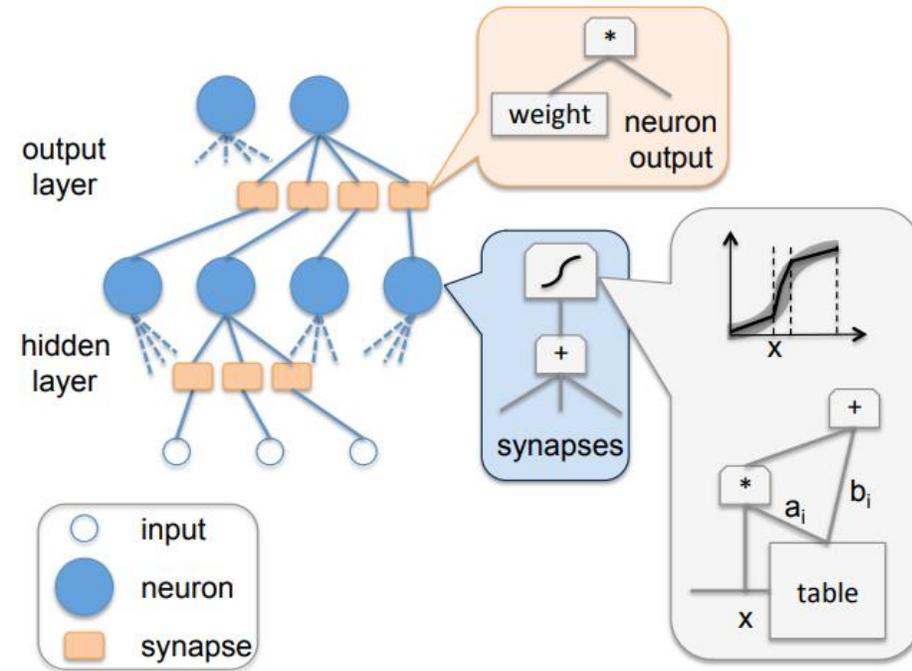
# Accelerating NNs

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.

# Sketch of deep learning inference

---

## FULLY-CONNECTED LAYER

```
for (int n = 0; n < N; n++) {
    sum := 0;
    for (int i = 0; i < N; i++) {
        sum += synapse[n][i] * neuron[n];
    }
    neuron[i] = sigmoid(sum);
}
```

## CONVOLUTION

```
for (int x = 0; x < width - filter_w; x++) {
    for (int y = 0; y < height - filter_y; y++){
        sum := 0;
        for (int fx = 0; fx < filter_w; fx++) {
            for (int fy = 0; fy < filter_y; fy++) {
                sum += filter[fx,fy]*data[x+fx][y+fy];
            }
        }
    }
}
```

# DianNao architecture

Treat DNNs as dense linear algebra

Simple, microcoded control

- Highly specialized “instructions”

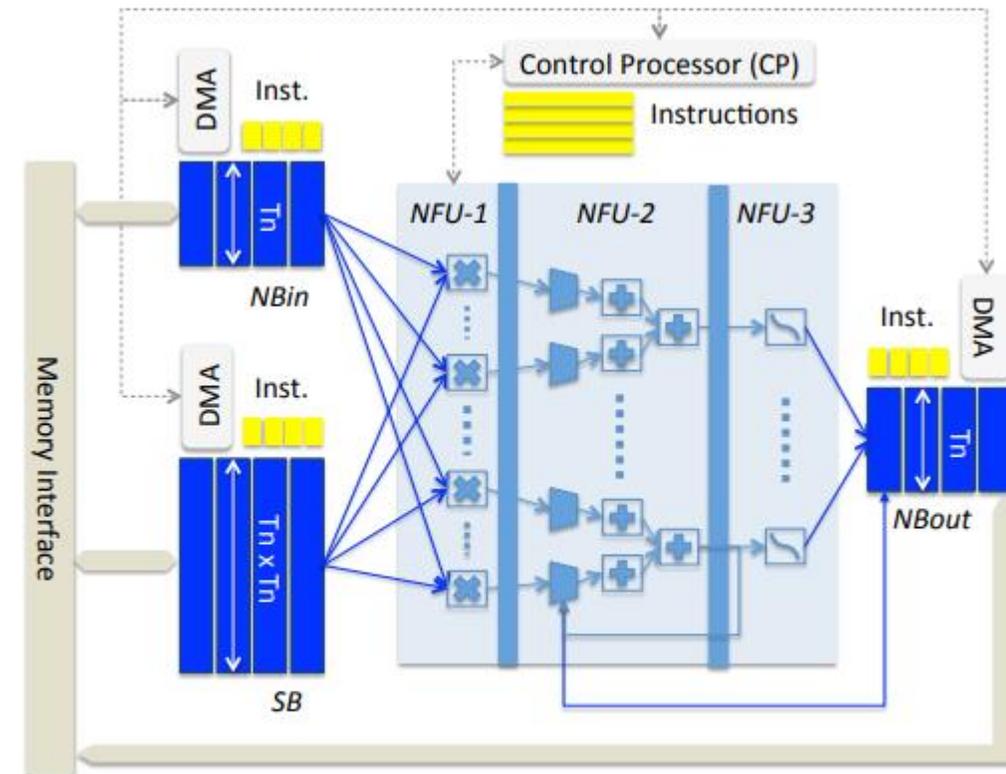
Custom datapath for multiply + add + sigmoid

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

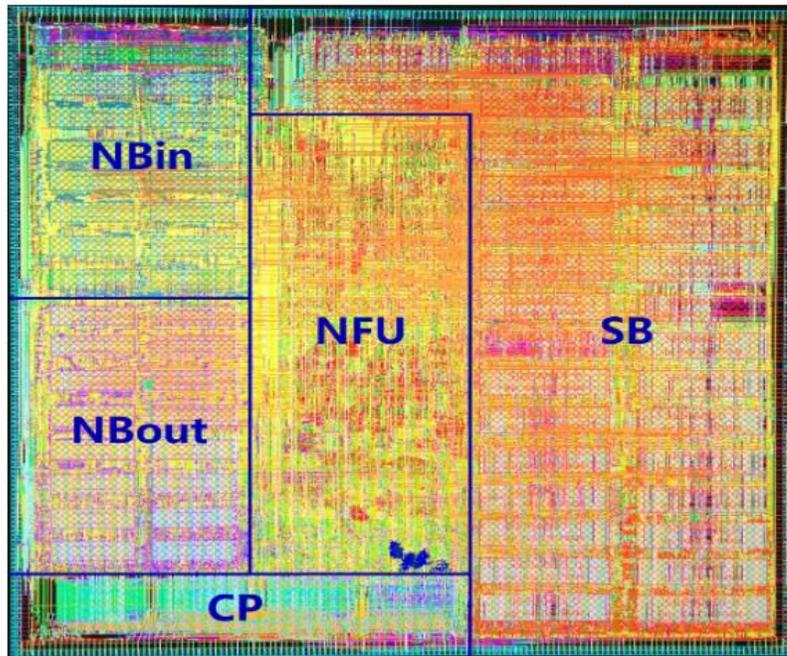
110 × avg performance improvement vs SIMD

21 × avg energy improvement vs SIMD

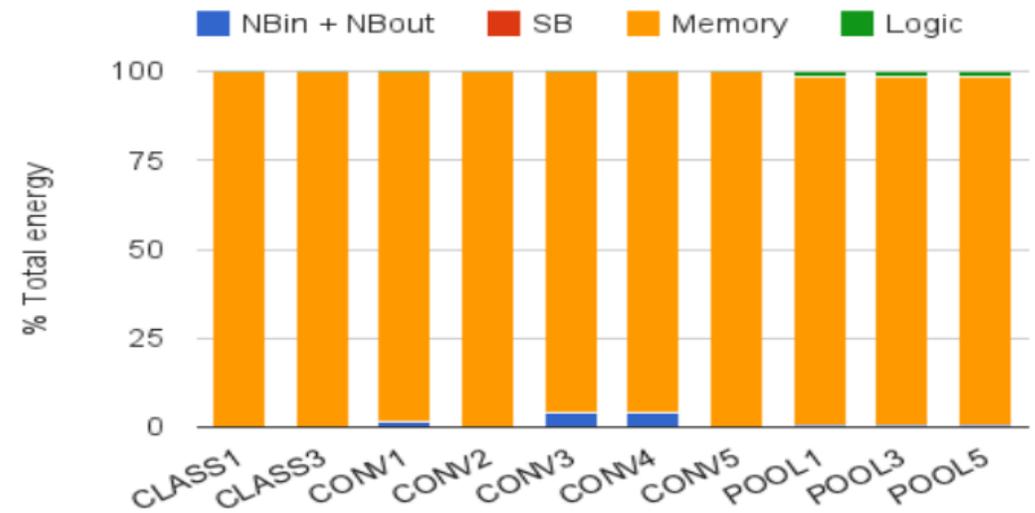
- Much smaller improvement than other studies!
- **Memory dominates energy**

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

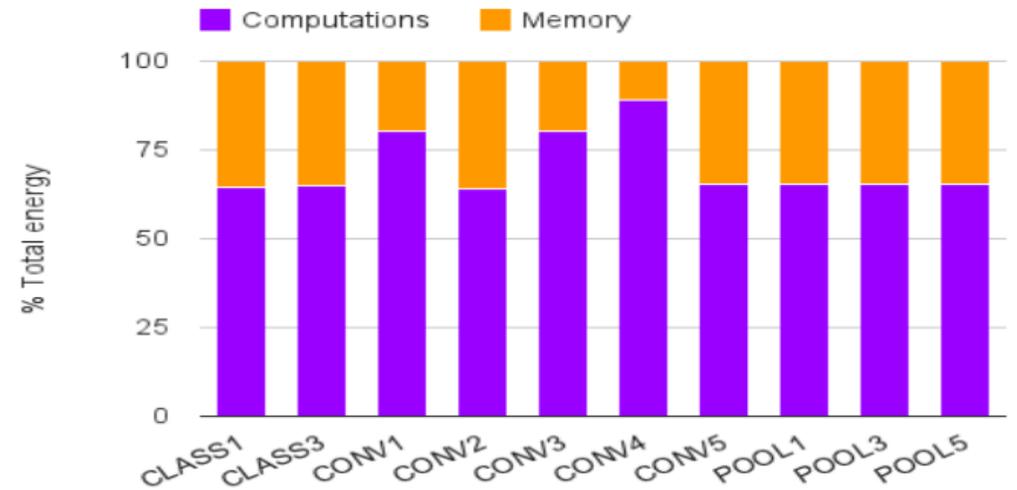
3mm<sup>2</sup>



**Figure 15.** Layout (65nm).



**Figure 18.** Breakdown of accelerator energy.



**Figure 19.** Breakdown of SIMD energy.

# Case study: Deep learning (2/2)

---

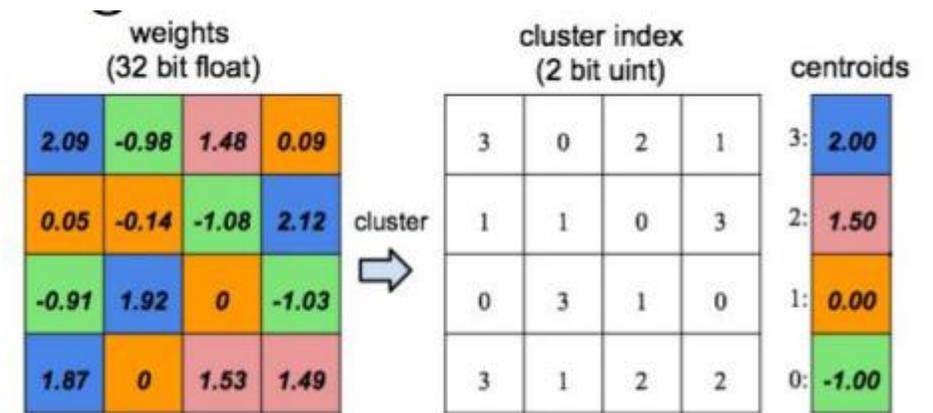
# EIE: Sparse Neural Networks

[Han et al, ISCA'16]

Also focuses on memory bottleneck

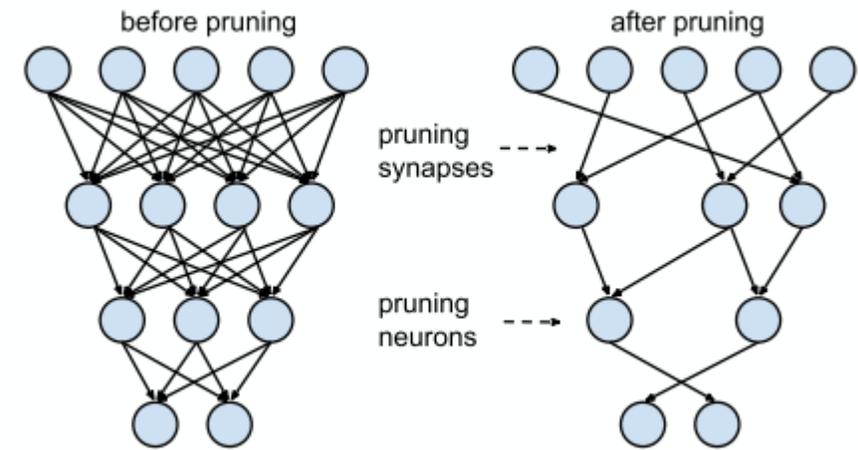
Observation: Weights are concentrated at a few values

- 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



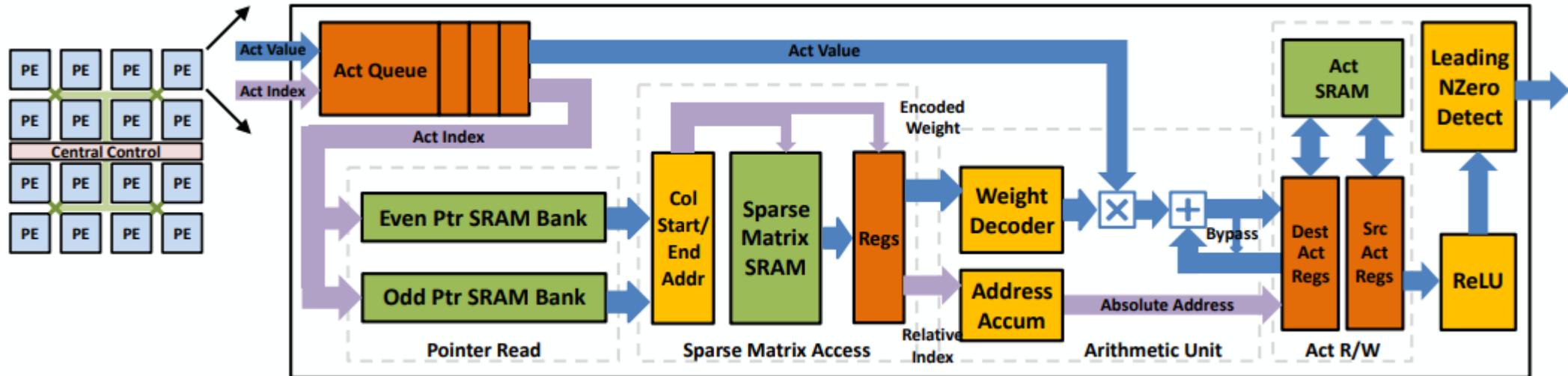
10-49× reduction in memory footprint

# 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



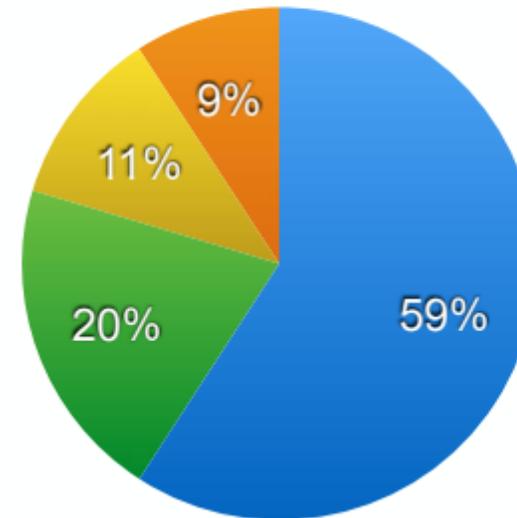
# EIE results

41mm<sup>2</sup> @ 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**

- memory
- clock network
- register
- combinational



...but memory still dominates, even with data kept in on-chip SRAM

# Other work in deep learning

---

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

- Same architectural principles as DianNao
- Gigantic “2-dimensional vector unit”

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...

- Theme: Co-design of algorithm + hardware

# Case study: Graphs

---

# Sketch of a graph algorithm

---

```
while (!frontier.empty()) {
    for (Vertex src in frontier) {
        for (Vertex dst in src.neighbors) {
            changed := processEdge(src, dst);
            if (changed) { nextFrontier.insert(dst); }
        }
    }
    for (Vertex v) {
        update(v);
    }
    frontier, nextFrontier = nextFrontier, new Frontier();
}
```

# 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...

# 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

```

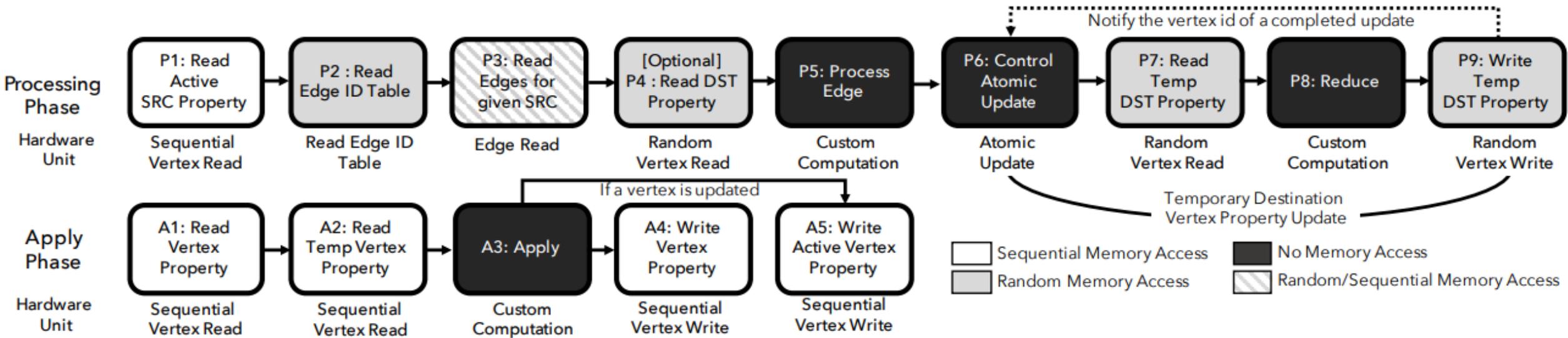
1 For each Vertex V
2   For each incoming edge E(U,V) from active vertex U
3     Res ← Process_Edge ( $E_{weight}$ ,  $U_{prop}$ , [OPTIONAL] $V_{prop}$ )
4      $V_{temp}$  ← Reduce( $V_{temp}$ , Res)
5   End
6 End
7 For each Vertex V,
8    $V_{prop}$  ← Apply( $V_{temp}$ ,  $V_{prop}$ ,  $V_{const}$ )
9 End
    
```

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

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 (literally just steps of the loop)



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**

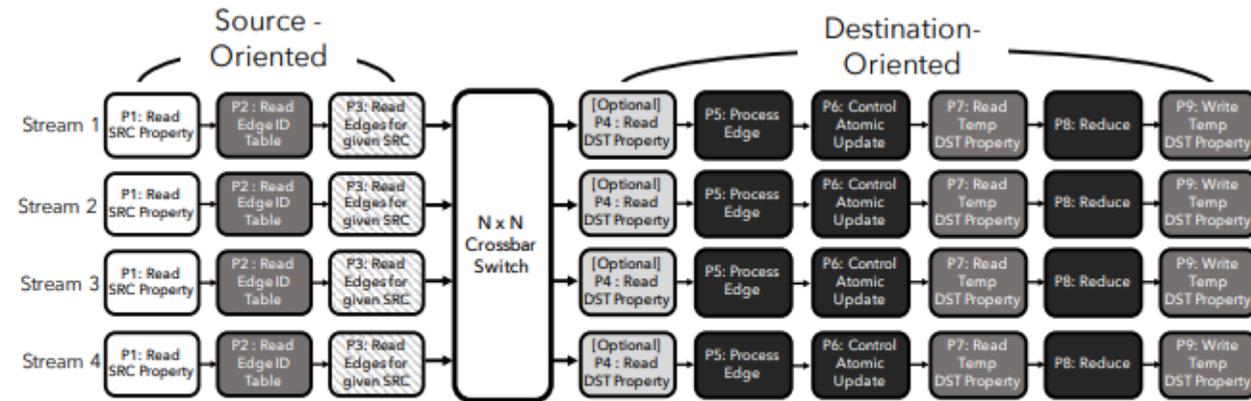


Fig. 7: Parallel implementation of Graphicianado. This diagram omits the Apply phase pipeline which is parallelized in a similar manner.

# Graphicianado memory system

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

- Assumed done via preprocessing

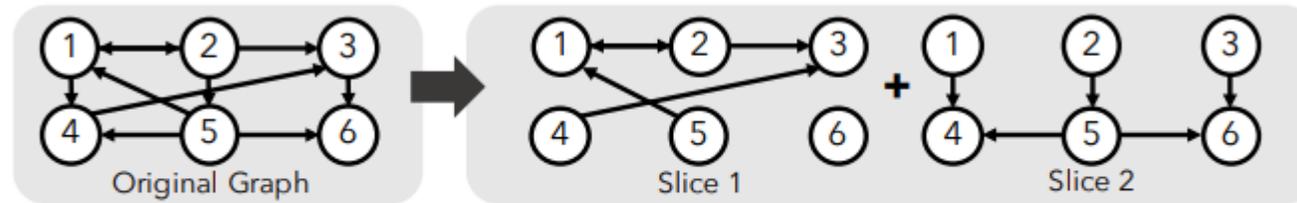


Fig. 10: Graph slicing Example.

(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

# Graphicianado evaluation

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

- This is unfortunately common for some fabs

1.75 – 6.5 × speedup vs CPU

50 – 100 × energy improvement

All energy goes into the scratchpad

**Evaluation does not include main memory!**

- ...or cost of preprocessing

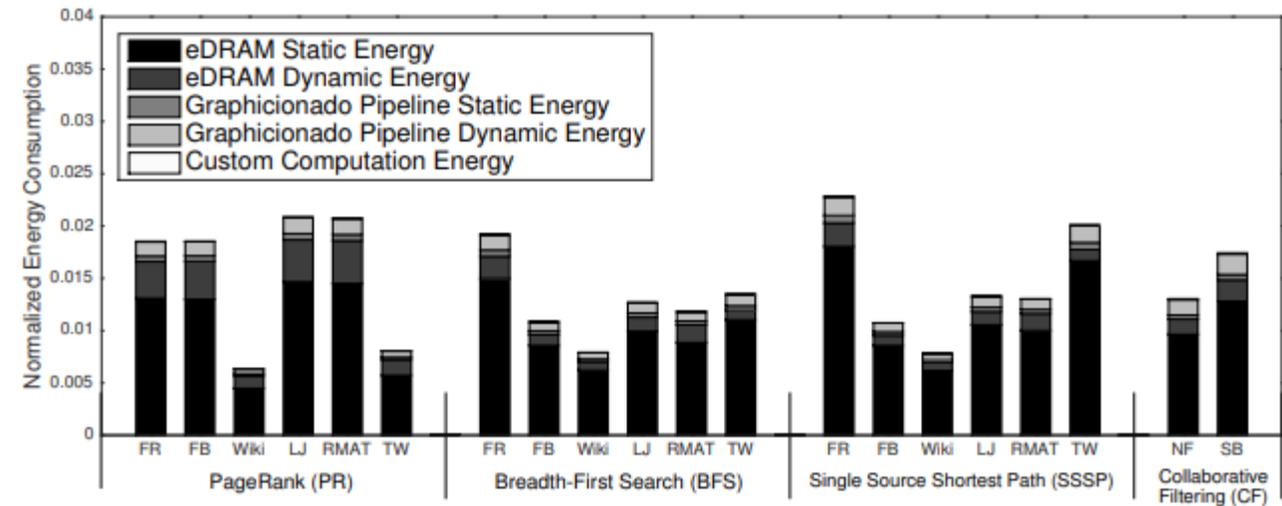


Fig. 17: Graphicianado energy consumption normalized to the energy consumption of Xeon processor running software graph processing framework.

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

# 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

# Specialization pros & cons

---

## BENEFITS / OPPORTUNITIES

Hardwired, low-power control

Custom functional units

Custom & **direct** communication

- Not through registers / cache

Custom memory system

**Extreme parallelism using app knowledge**

➔ ~All 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!

# 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!*

# Reconfigurable Architectures

---

# 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

```
1 val CUTOFF: Int = Date("1998-12-01")
2 val lineItems: Array[LineItem] = ...
3 val before = lineItems.filter{ item => item.date < CUTOFF }
4
5 val query = before.hashReduce{ item =>
6   // Key function (k)
7   (item.returnFlag, item.lineStatus)
8 }{ item =>
9   // Value function (v)
10  val quantity = item.quantity
11  val price = item.extendedPrice
12  val discount = item.discount
13  val discountPrice = price * (1.0 - discount)
14  val charge = price * (1.0 - discount) * (1.0 + item.tax)
15  val count = 1
16  (quantity, price, discount, discountedPrice, count)
17 }{ (a,b) =>
18   // Combine function (r) - combine using summation
19   val quantity = a.quantity + b.quantity
20   val price = a.price + b.price
21   val discount = a.discount + b.discount
22   val discountPrice = a.discountPrice + b.discountPrice
23   val count = a.count + b.count
24   (quantity, price, discount, discountPrice, count)
25 }
```

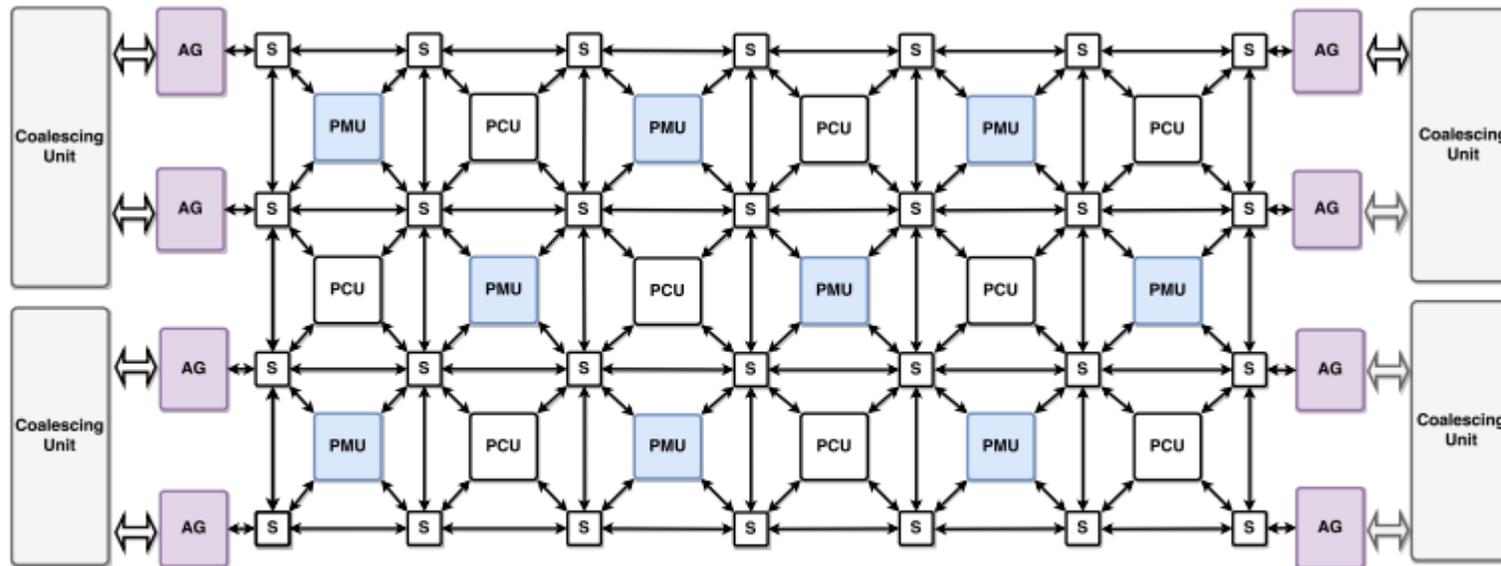
Figure 2: Example of using filter (FlatMap) and HashReduce in a Scala-based language, inspired by TPC-H query 1.

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

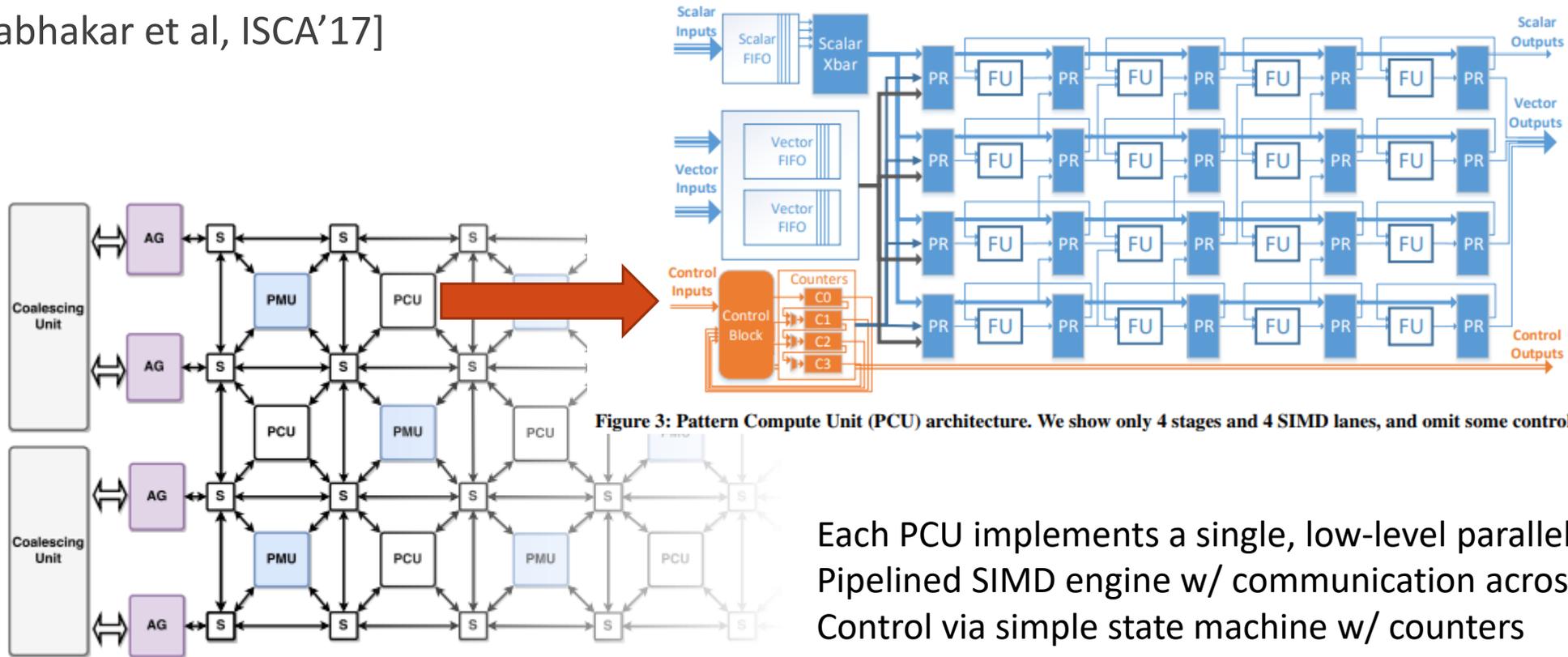


Figure 3: Pattern Compute Unit (PCU) architecture. We show only 4 stages and 4 SIMD lanes, and omit some control signals.

Each PCU implements a single, low-level parallel pattern  
 Pipelined SIMD engine w/ communication across lanes  
 Control via simple state machine w/ counters

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

# Plasticine

[Prabhakar et al, ISCA'17]

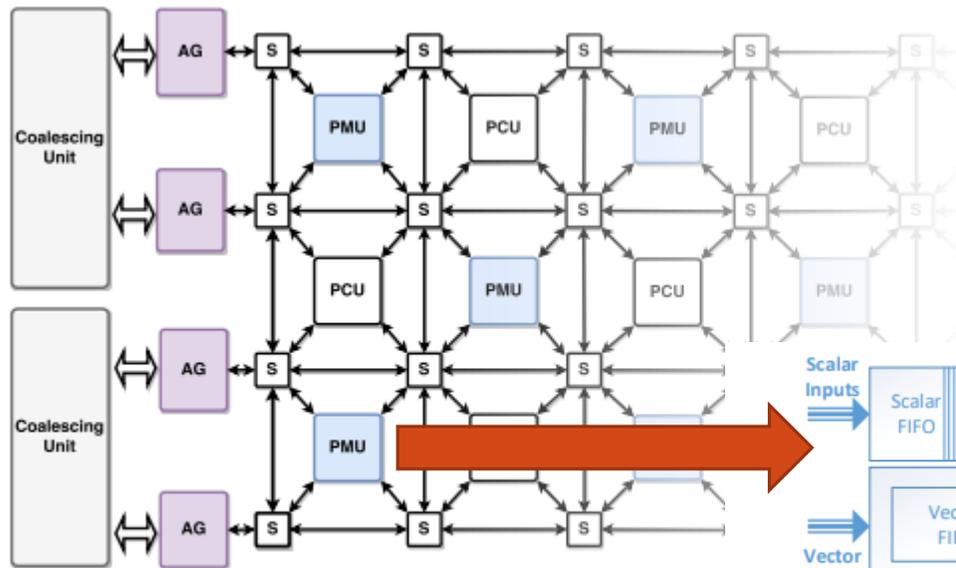


Figure 5: Plasticine chip-level architecture (actual organization PCU:Pattern Compute Unit, PMU: Pattern Memory Unit, AG:

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

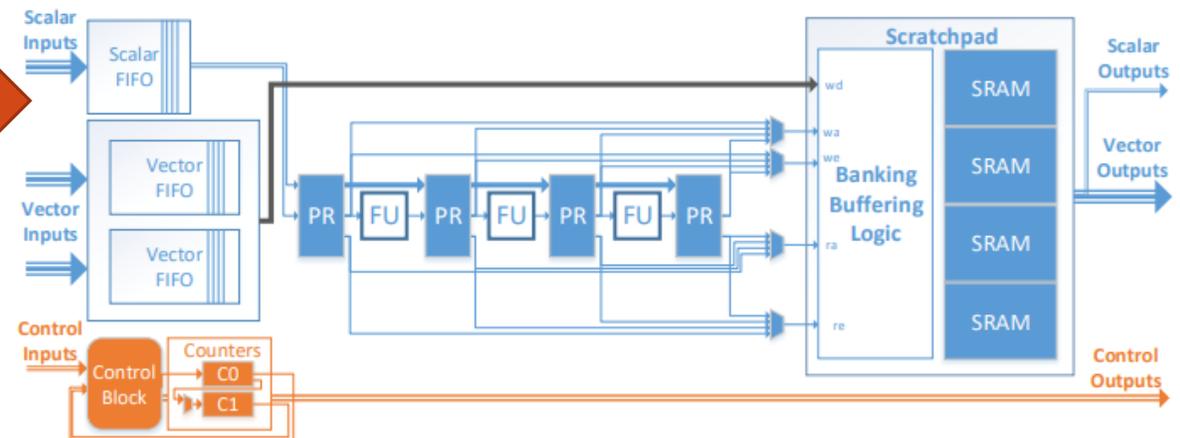


Figure 4: Pattern Memory Unit (PMU) architecture: configurable scratchpad, address calculation datapath, and control.

# Plasticine Evaluation

---

112mm<sup>2</sup> 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 × area vs. ASIC design, on average

1.5 – 77 × energy improvement vs. FPGA (depends heavily on how well app maps to FPGA)

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