Programmable Hardware Acceleration

Vinay Gangadhar

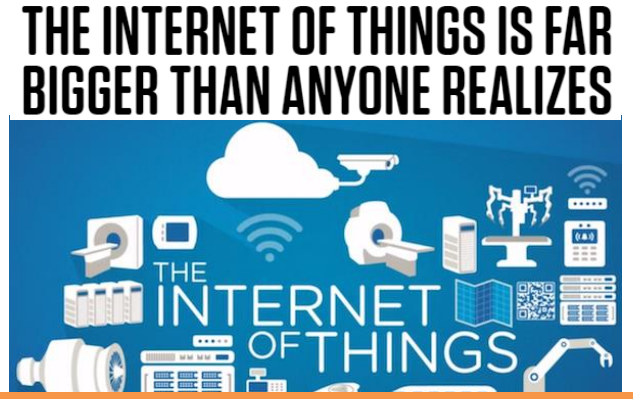
PhD Final Examination

Thursday, Nov 16th, 2017

Advisor: Karu Sankaralingam

Committee: Mark Hill, Mikko Lipasti, David Wood, Dimitris Papailiopoulos

Computing Trends



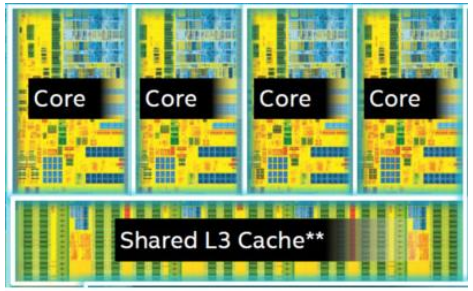
Device scaling slowdown (or dead)
&
Dark silicon problem

Emerging applications driving computing with new demands

Era of Specialization

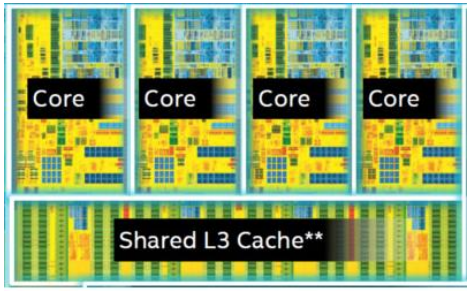
VERTICAL

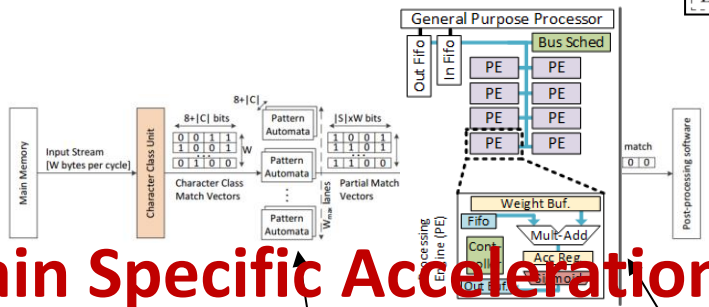
Traditional Multicore



Application domain specialization

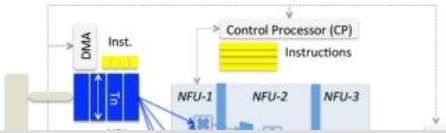
Domain Specific Acceleration

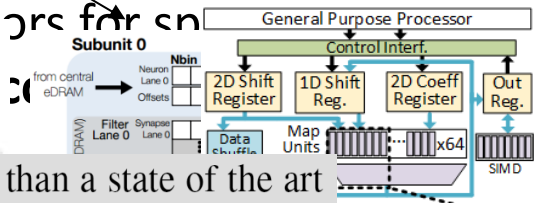




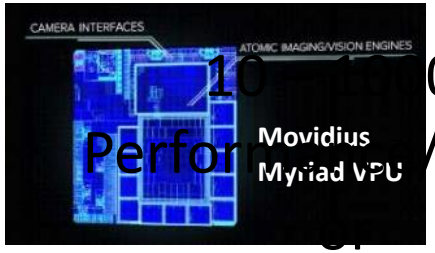
Reg Expr. | AI | Neural Approx. | Scan | Graph Traversal | Image Processing | Deep Neural | Stencil | Sort

Fixed-function Accelerators for sp





+ High Efficiency



10 0x

Perfor /Power

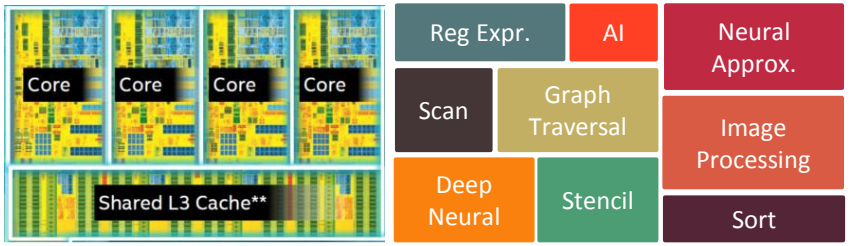
NVIDIA DGX-1 AI Accele & NVDLA Architectu

Performance/Area

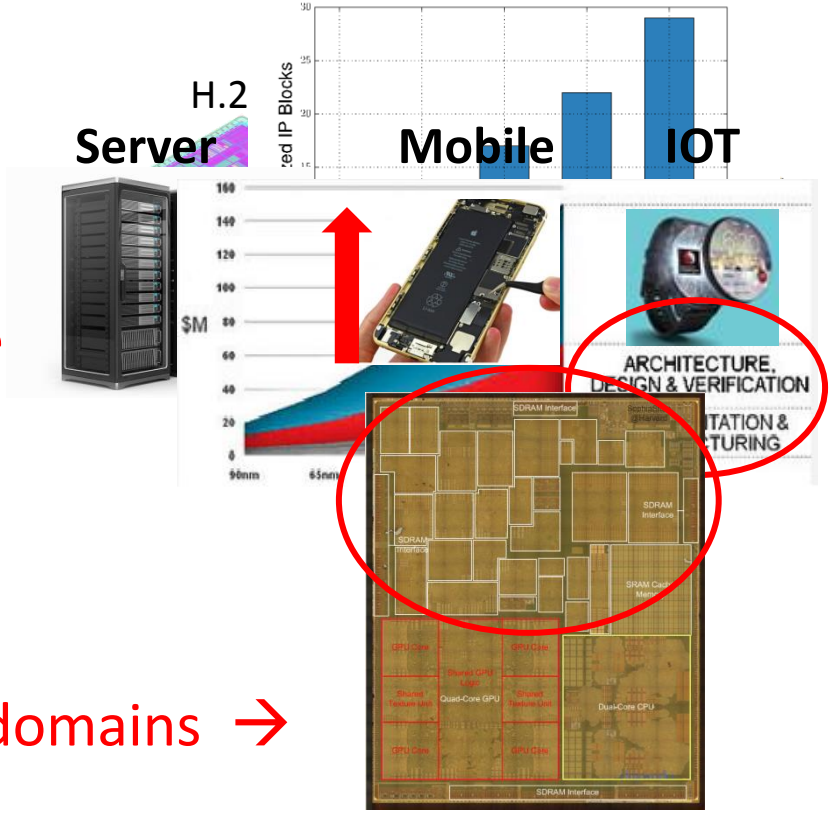
three orders of magnitude less energy than a state of the art software DBMS, while the performance-oriented design out-performs the same DBMS by **70X**

sor, the accelerator is **117X** faster, and it can reduce the total energy by **21X** The accelerator characteristics are obtained after layout at 65nm. Such a high throughput in

Caveats of Domain-Specific Accelerators (DSAs)

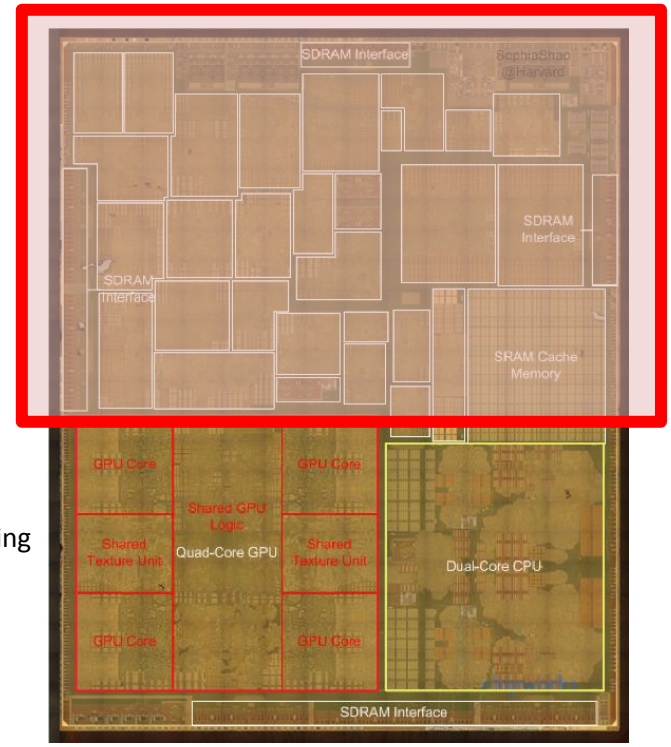


- Minimally programmable/ Not Re-configurable
- Obsoletion prone
- Domains targeting each device type
- Architecture, design, verification and fabrication cost
- Multi-DSA chip for "N" application domains → Area and cost inefficient



Source: Malitel Consulting

The Universal Accelerator Dream...



Source:
Malitel Consulting

Deep Neural

Image Processing

Automated Driving

Compression

Regex Matching

Query Processing

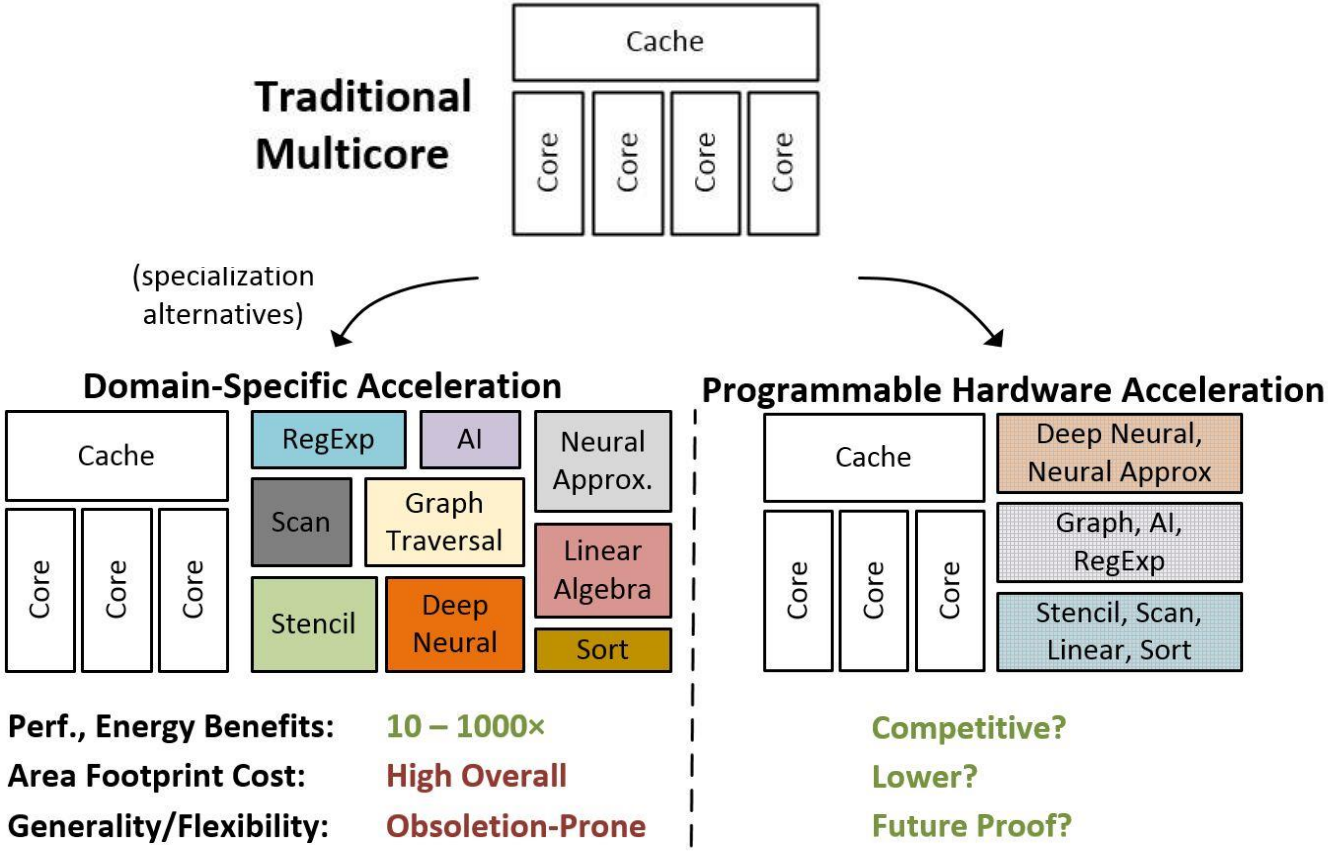
Convert 100+ Accelerators

1 Programmable Accelerator Fabric

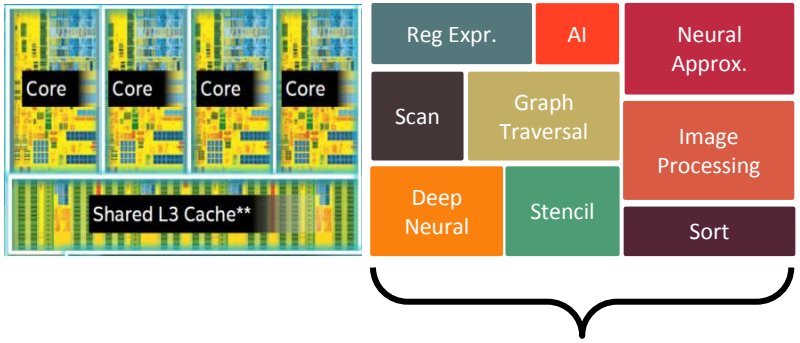
Standard programming and threading interface

A generic programmable hardware accelerator matching the efficiency of Domain Specific Accelerators (DSAs) with an efficient hardware-software interface

Specialization Paradigms



Research Overview



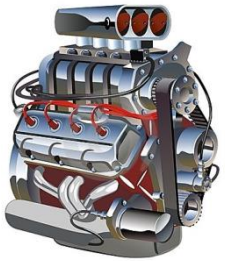
Domain-Specific Accelerators (DSAs)

Commonality in DSAs ?

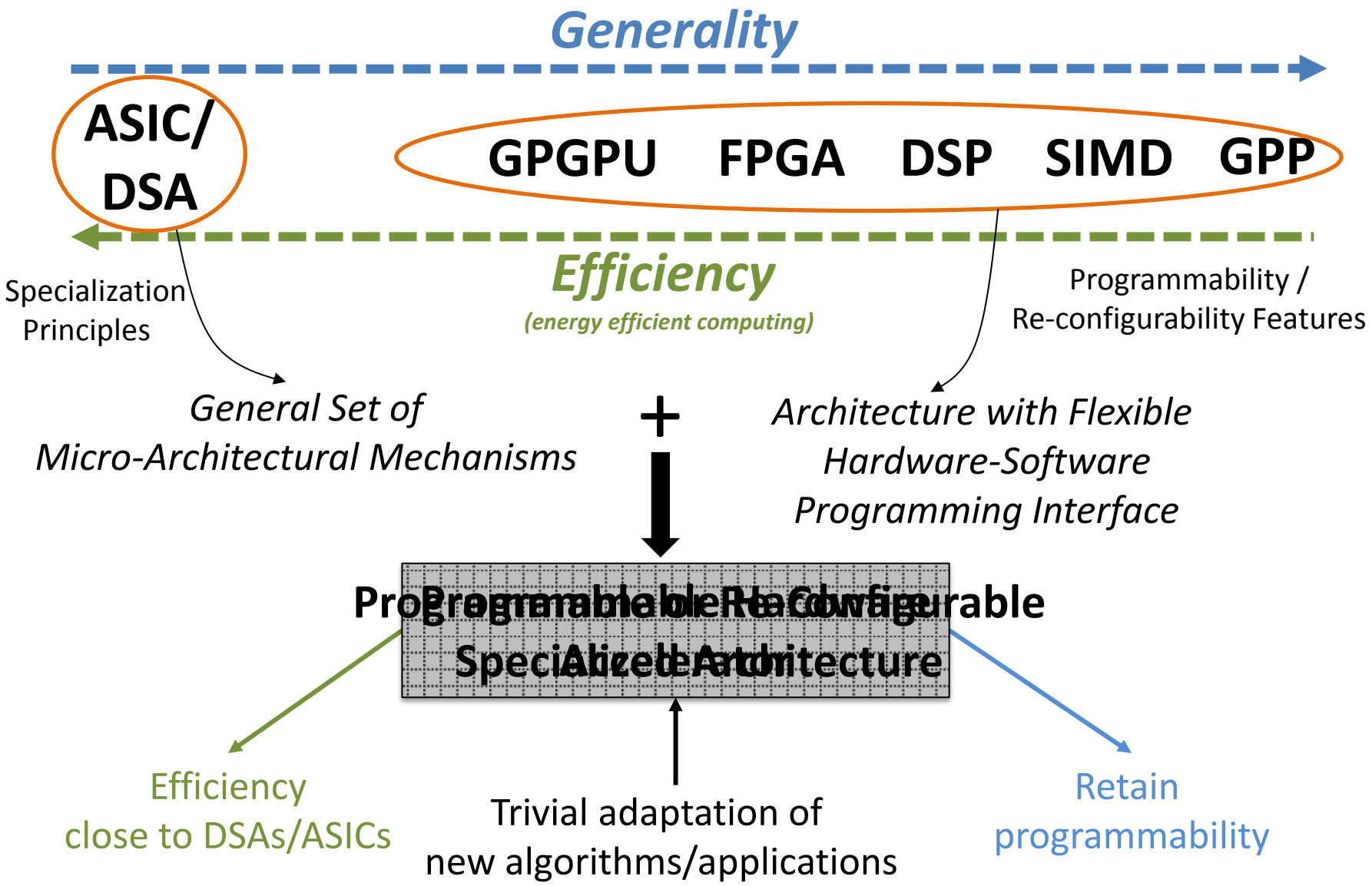
Specialization Principles

Micro-Architectural Mechanisms

Programmable Hardware Accelerator Architecture



Research Overview



Dissertation Research Goal

Programmable Hardware Acceleration

1. Explore the commonality in the way the DSAs specialize – ***Specialization Principles***

2. **General Mechanisms** for the design of a generic programmable hardware accelerator matching the efficiency of DSAs

3. A programmable/re-configurable accelerator architecture with an efficient **accelerator hardware-software (ISA) interface**

4. **Easy adaptation** of new acceleratable algorithms in a domain-agnostic way

Dissertation Statement

Programmable Hardware Acceleration

A ***programmable hardware accelerator*** nearing the efficiency of a domain-specific accelerator (DSA) is feasible to build by:

- *Identifying the common principles of architectural specialization*
- *Applying general set of micro-architectural mechanisms for the identified principles*
- *Having an efficient hardware-software interface to be able to express any typical accelerator application*

Contributions

Modeling Programmable Hardware Acceleration

- Exploring the common principles of architectural specialization
- Modeling a general set of mechanisms to exploit the specialization principles – GenAccel Model
- Quantitative evaluation of GenAccel Model with four DSAs
- System-Level Tradeoffs of GenAccel Model vs. DSAs

Architectural Realization with Stream-Dataflow Acceleration

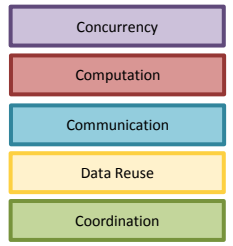
- Stream-Dataflow programmable accelerator architecture with:
  - Programming abstractions and execution model
  - ISA interface
- Detailed micro-architecture with an efficient architectural realization of stream-dataflow accelerator – *Softbrain*
- Quantitative evaluation of Softbrain with state-of-the-art DSA solutions

Modeling Programmable Hardware Acceleration*

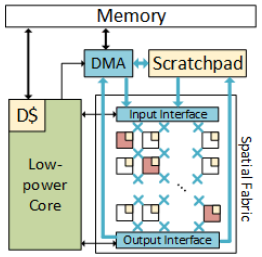
*Published in HPCA 2016, IEEE Micro Top Picks 2017

Outline

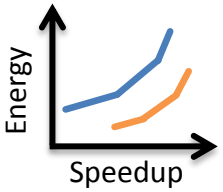
- Principles of architectural specialization
  - Embodiment of principles in DSAs



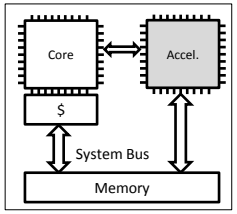
- Modeling mechanisms exploiting specialization principles for a generic programmable accelerator (GenAccel Model)



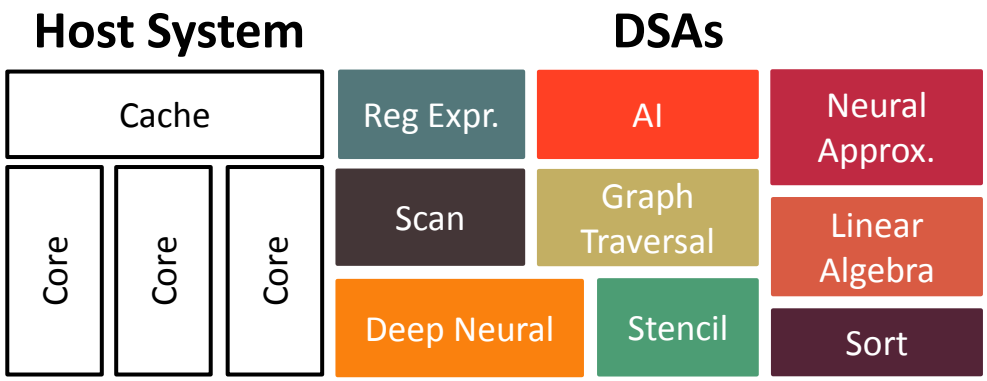
- Evaluation of GenAccel with 4 DSAs (Performance, power & area)



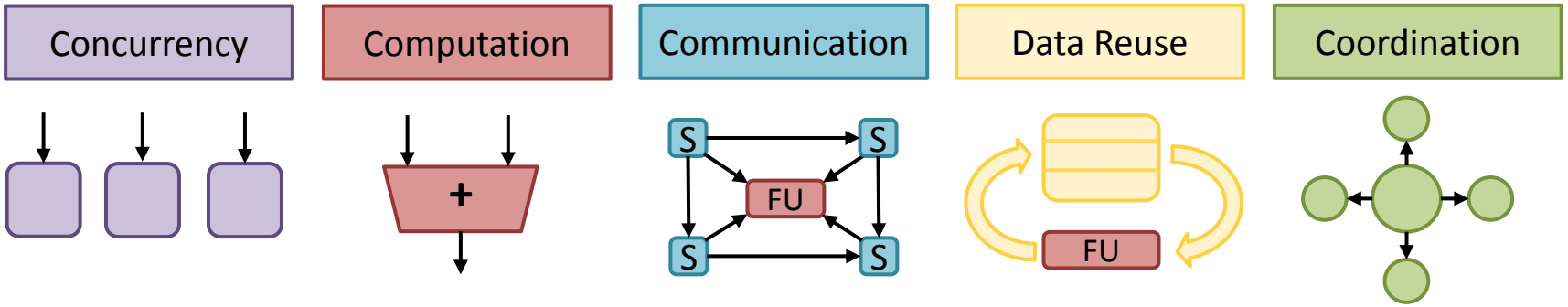
- System-level energy efficiency tradeoffs with GenAccel and DSA



Key Insight: Commonality in DSAs' Specialization Principles

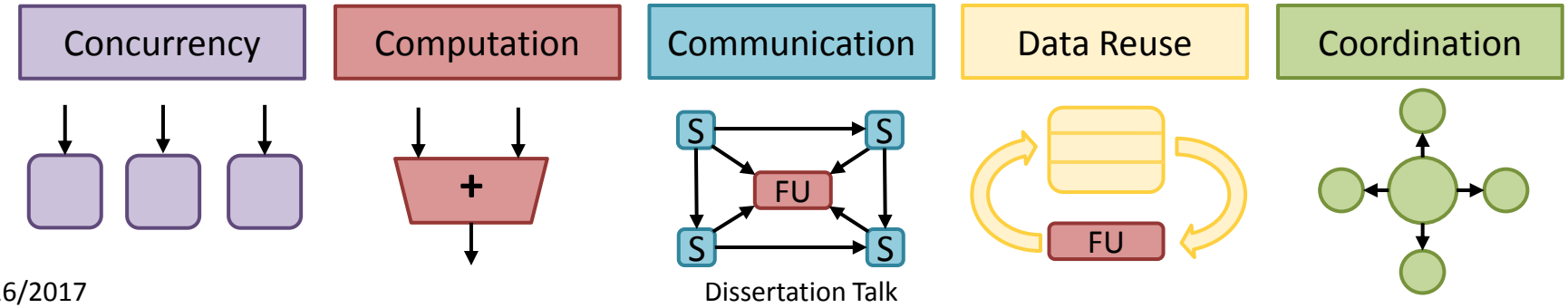


Most DSAs employ 5 common Specialization Principles

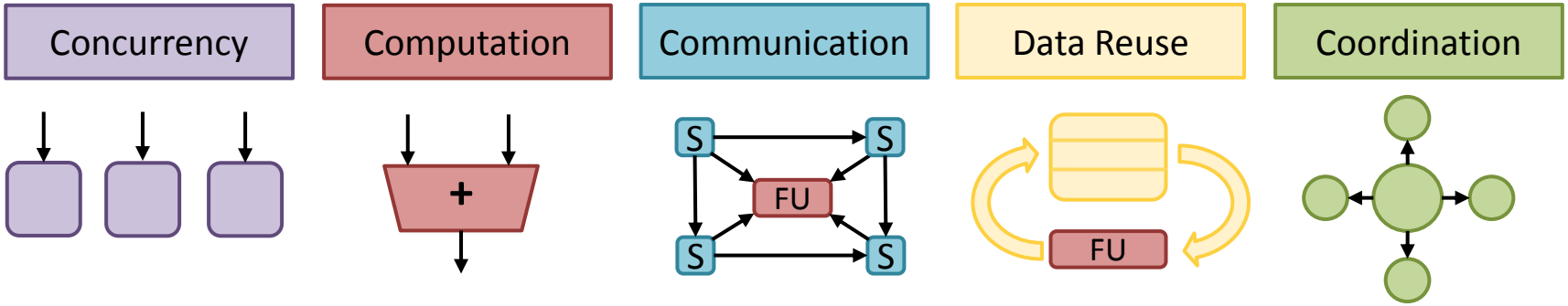


Principles of Architectural Specialization

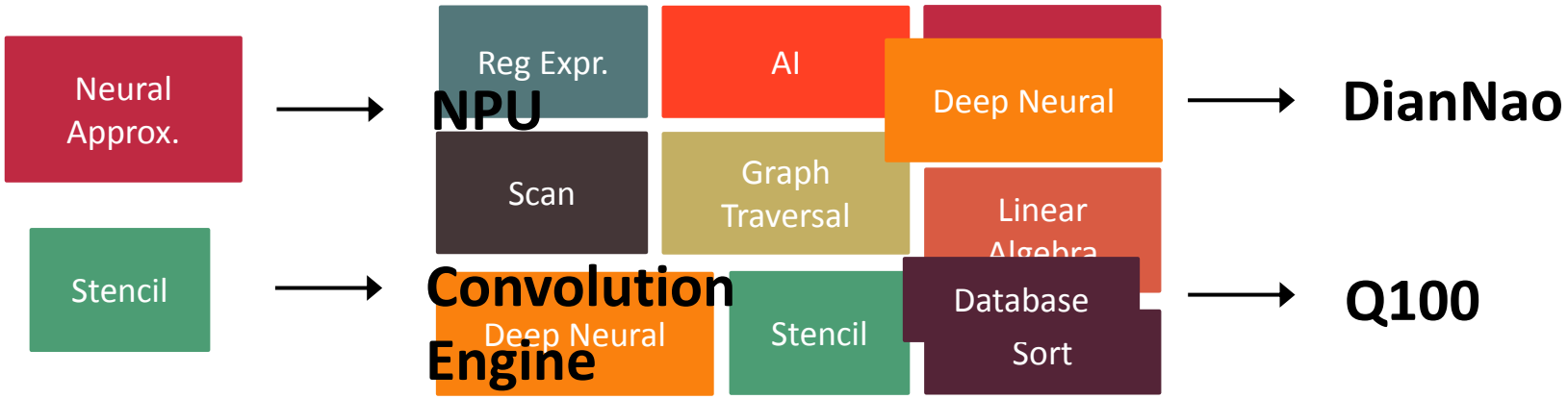
- Match hardware **concurrency** to that of algorithm
- Problem-specific **computation** units
- Explicit **communication** as opposed to implicit communication
- Customized structures for **data reuse**
- Hardware **coordination** using simple low-power control logic



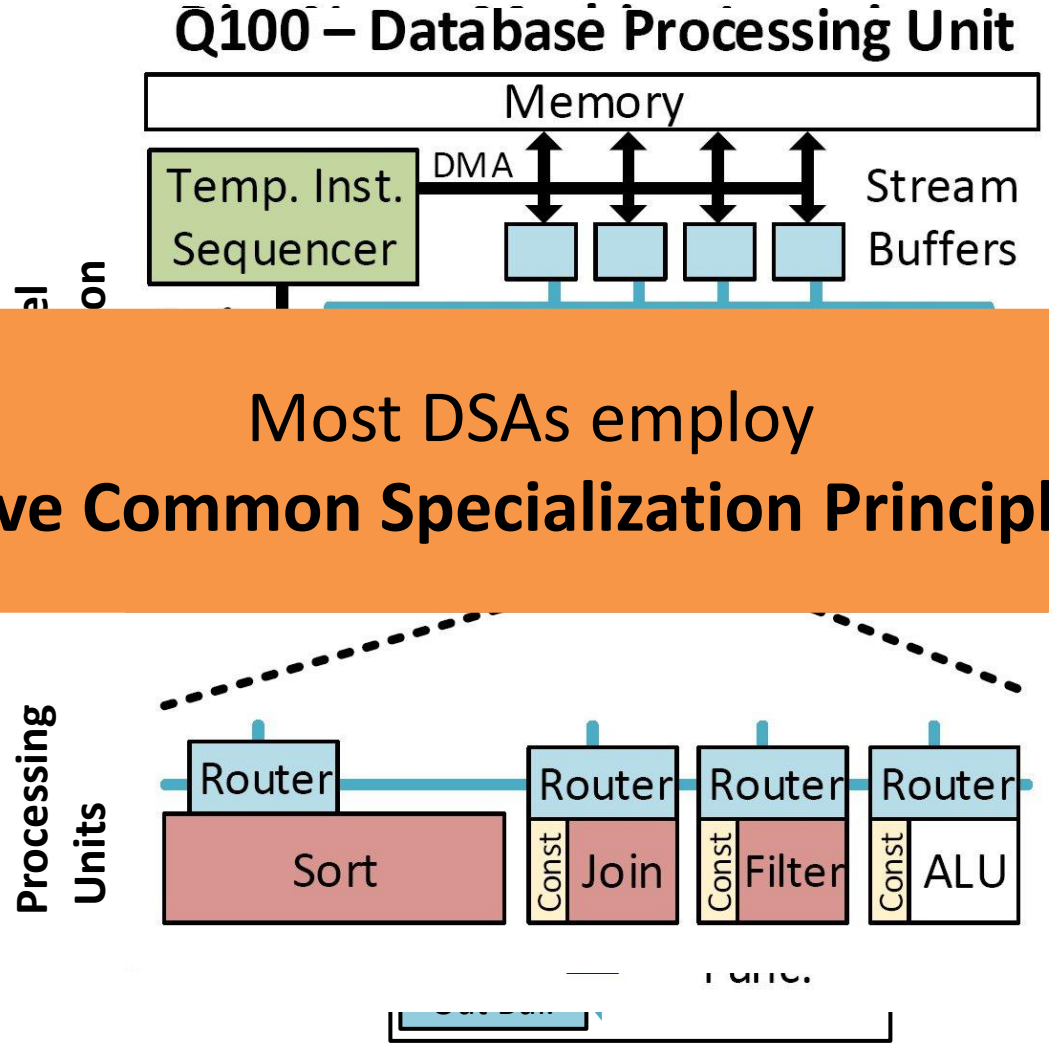
5 Specialization Principles



How do DSAs embody these principles in a domain specific way ?



Principles in DSAs



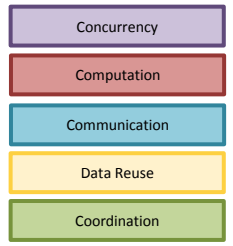
Most DSAs employ

Five Common Specialization Principles

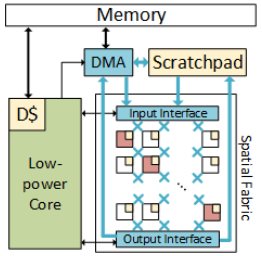
Concurrency | Computation | Communication | Data Reuse | Coordination

Outline

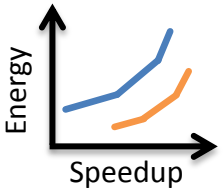
- Principles of architectural specialization
  - Embodiment of principles in DSAs



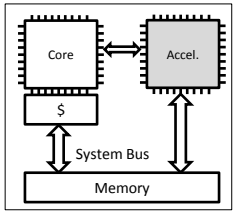
- Modeling mechanisms exploiting specialization principles for a generic programmable accelerator (GenAccel Model)



- Evaluation of GenAccel with 4 DSAs (Performance, power & area)



- System-level energy efficiency tradeoffs with GenAccel and DSA



Implementation of Principles in a General Way

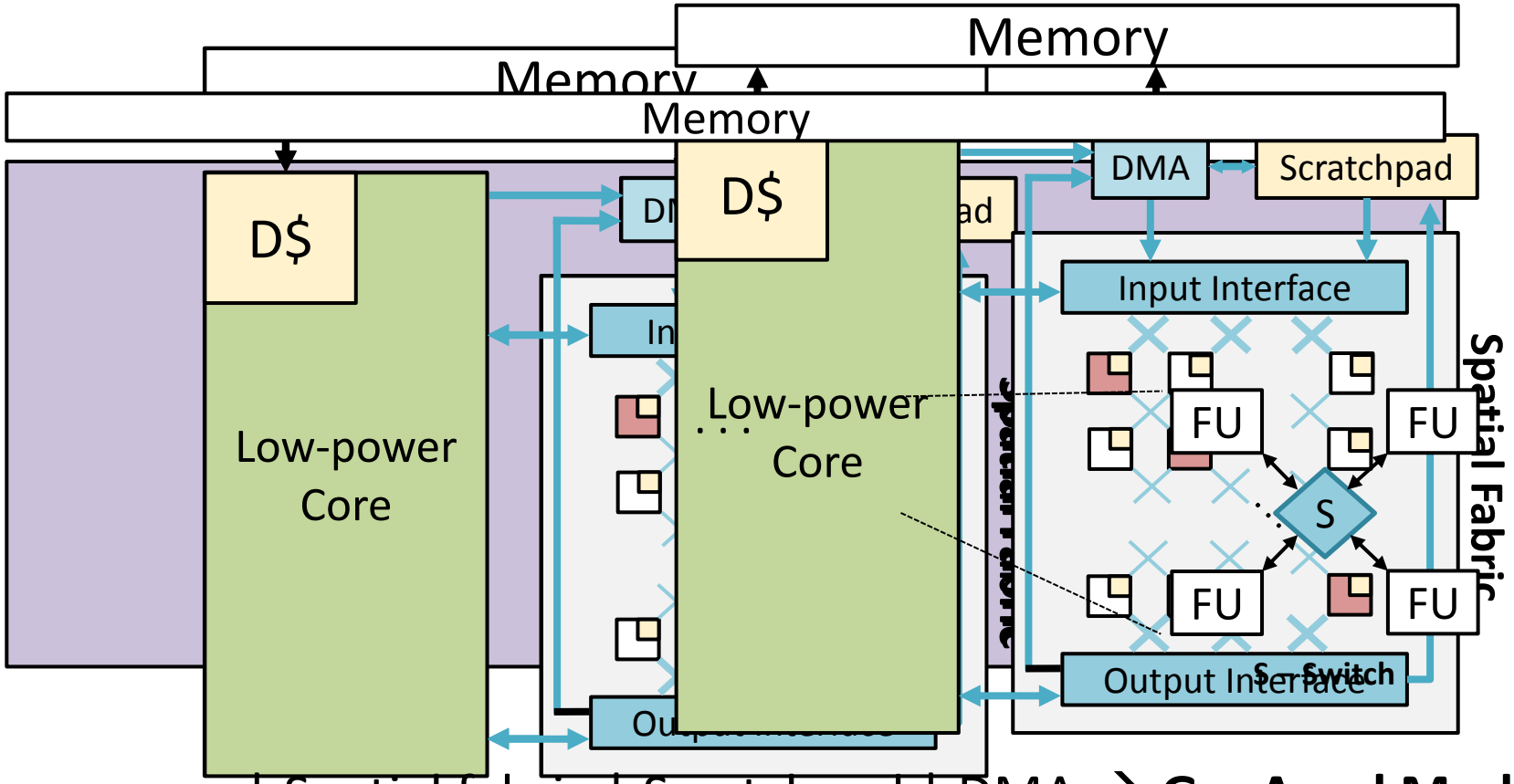
Composition of simple micro-architectural mechanisms

- **Concurrency:** Multiple tiles (Tile – hardware for coarse grain unit of work)
- **Computation:** Special FUs in spatial fabric
- **Communication:** Dataflow + spatial fabric
- **Data Reuse:** Scratchpad (SRAMs)
- **Coordination:** Low-power simple core

Each Tile

Concurrency | Computation | Communication | Data Reuse | Coordination

Modeling the Generic Programmable Accelerator Design

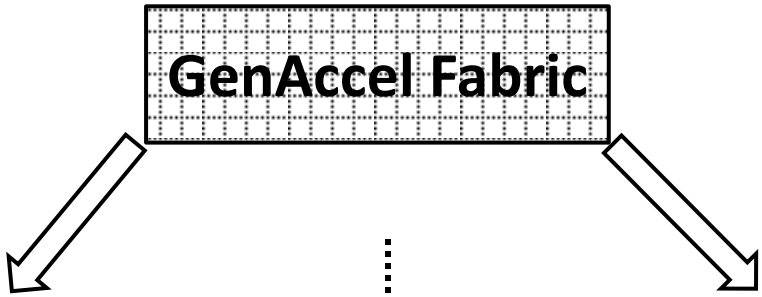


Low power core | Spatial fabric | Scratchpad | DMA → **GenAccel Model**

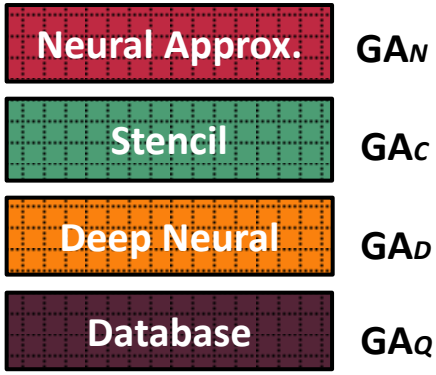
Concurrency | Computation | Communication | Data Reuse | Coordination

Instantiating GenAccel

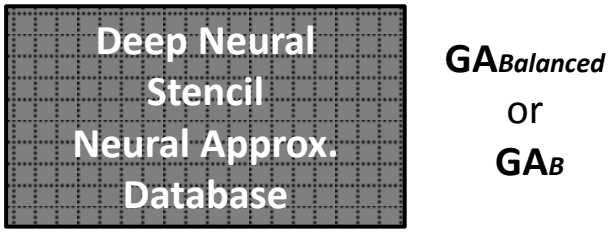
Programmable hardware template for specialization



Provisioned for
one single application domain



Provisioned for
multiple application domains

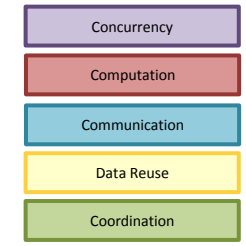


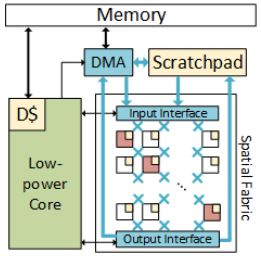
GenAccel Usage, Design point selection & Synthesis etc.
More details in backup…..

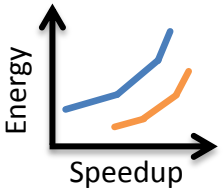
*Figures not to scale

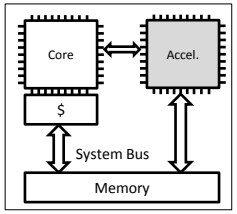
Outline

- Principles of architectural specialization
  - Embodiment of principles in DSAs
- Modeling mechanisms exploiting specialization principles for a generic programmable accelerator (GenAccel Model)
- Evaluation of GenAccel with 4 DSAs (Performance, power & area)
- System-level energy efficiency tradeoffs with GenAccel and DSA



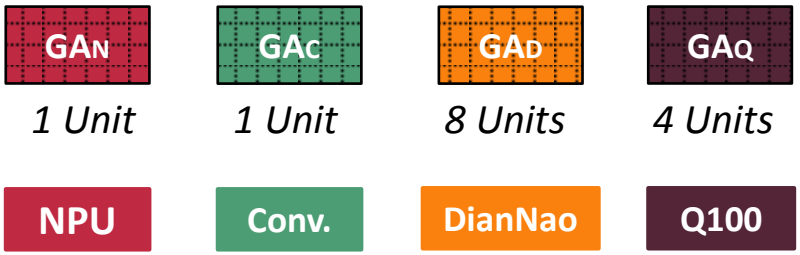




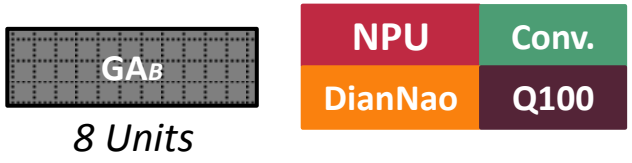


Methodology

- Modeling framework for GenAccel
  - Performance: Trace driven simulator + application specific modeling
  - Power & Area: Synthesized modules, CACTI and McPAT
- Compared to four DSAs (published perf., area & power)
- Four parameterized GenAccels

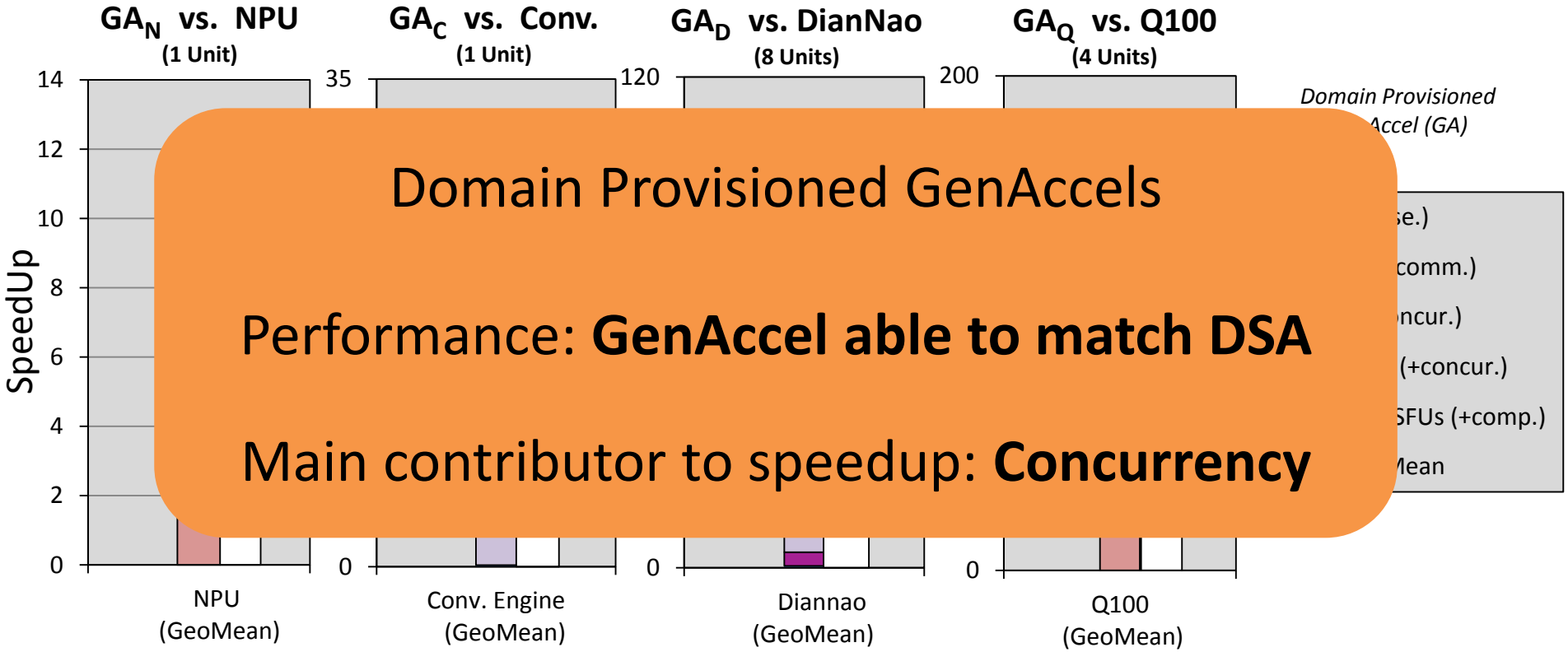


One combined balanced GenAccel



- Provisioned to *match* performance of DSAs
  - Other tradeoffs possible (power, area, energy etc.)

Performance Analysis
GenAccel vs DSAs

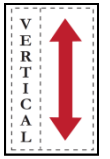


Domain Provisioned GenAccels

Performance: **GenAccel able to match DSA**

Main contributor to speedup: **Concurrency**

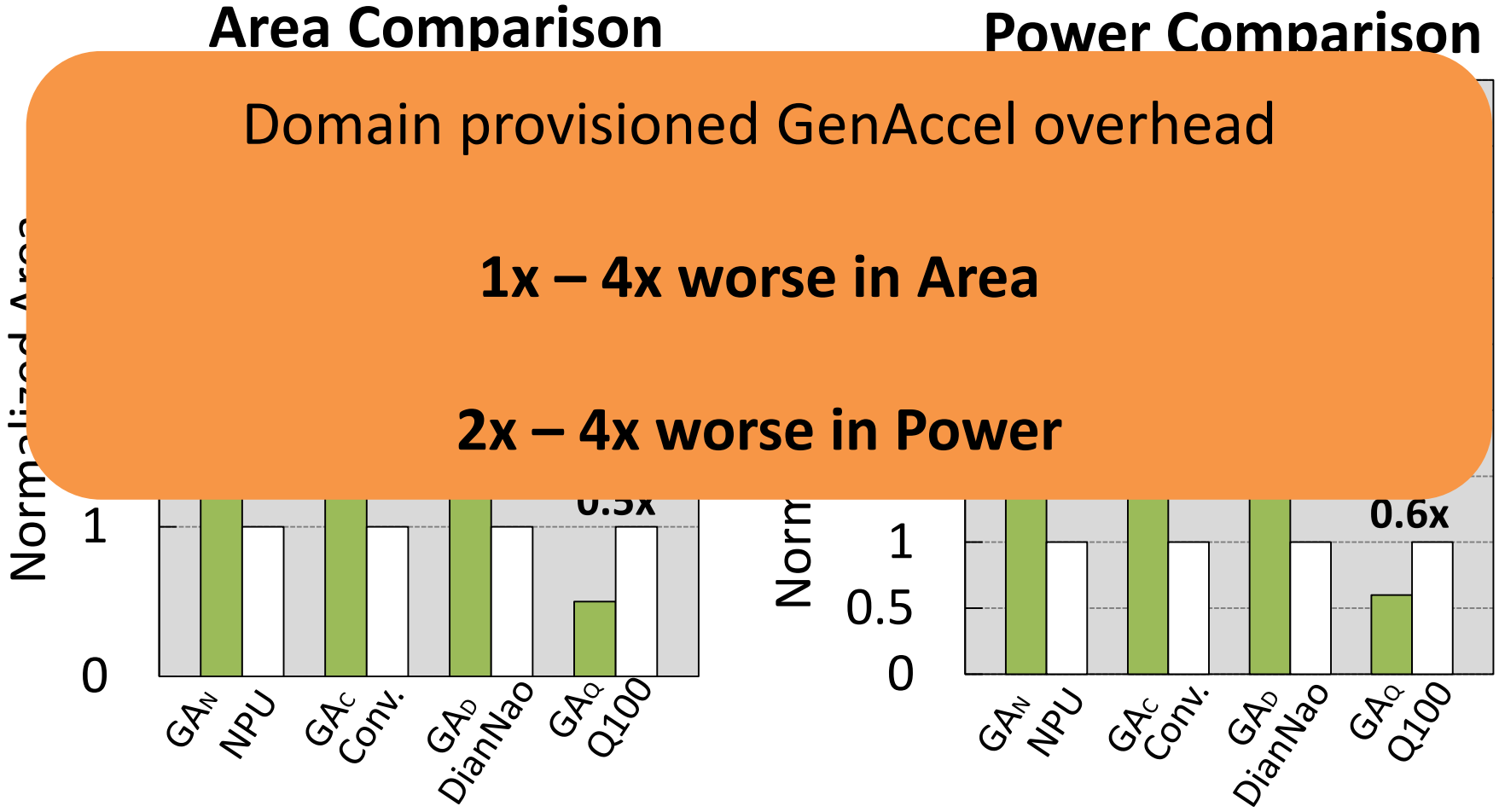
Baseline – 4 wide OOO core (Intel 3770K)



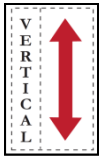
Domain Provisioned GenAccels

GenAccel area & power compared to a single DSA ?

Domain Provisioned GenAccels Area and Power Analysis



*Detailed area breakdown in backup

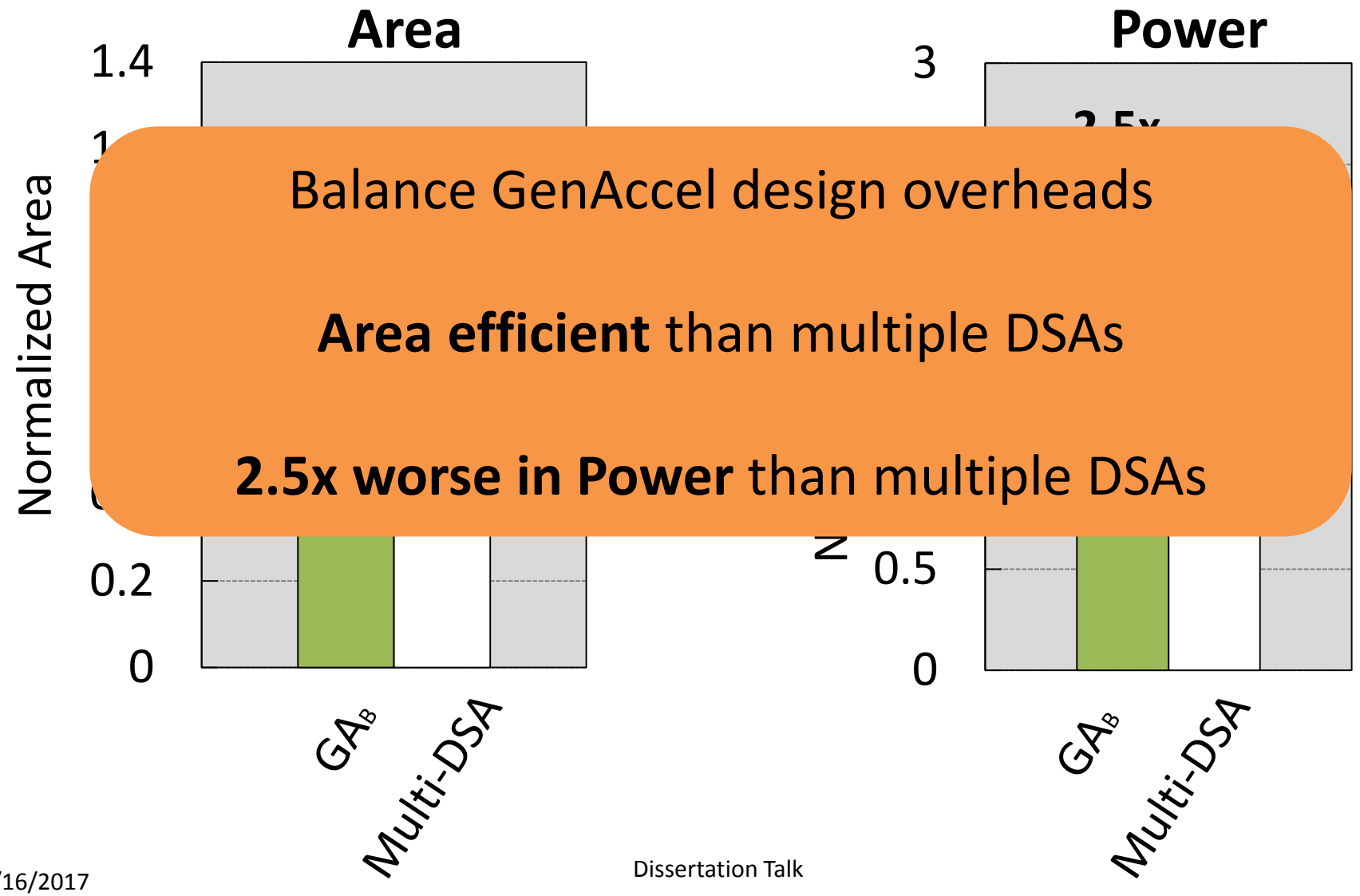


Balanced GenAccel design

Area and power of GenAccel Balanced design, when multiple domains mapped* ?

* Still provisioned to match the performance of each DSA

GenAccel Balanced Design Area-Power Analysis



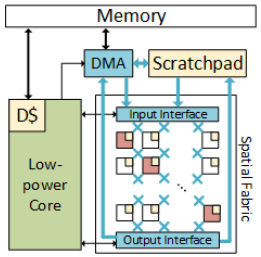
Balance GenAccel design overheads

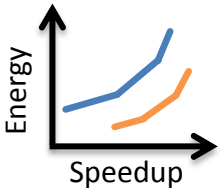
Area efficient than multiple DSAs

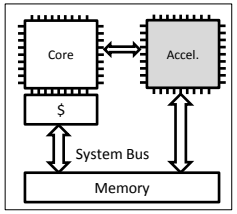
2.5x worse in Power than multiple DSAs

Outline

- Introduction
- Principles of architectural specialization
  - Embodiment of principles in DSAs
- Modeling mechanisms exploiting specialization principles for a generic programmable accelerator (GenAccel Model)
- Evaluation of GenAccel with 4 DSAs (Performance, power & area)
- System-level energy efficiency tradeoffs with GenAccel and DSA

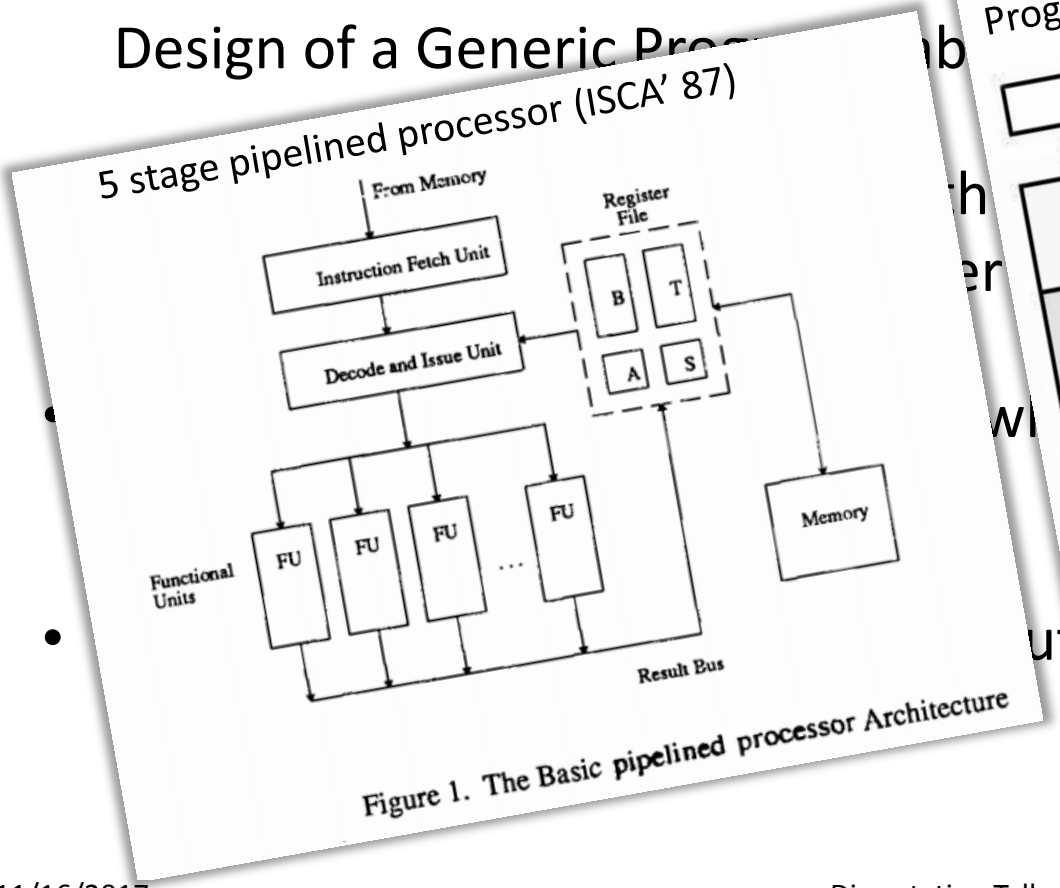


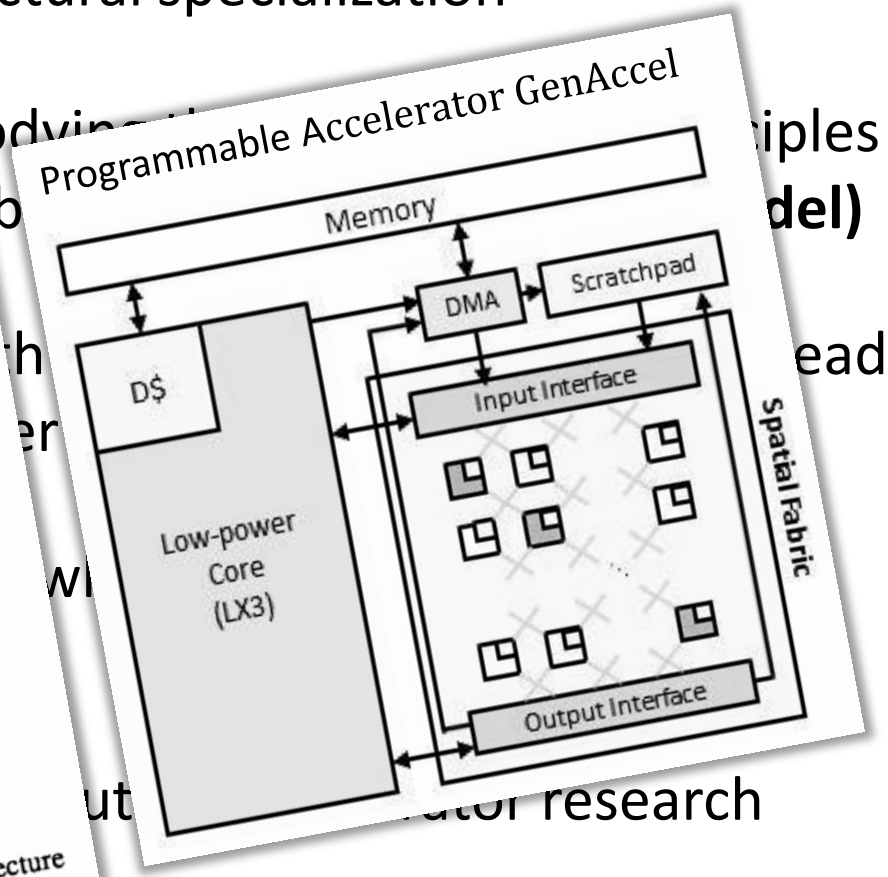




Conclusion – Modeling Programmable Hardware Acceleration

- 5 common principles for architectural specialization
- Modeled the mechanisms embodyi… …iples – Design of a Generic Prog… …b… …**del)**
- …h… …eads
- …er
- …wi…
- …ut… …ator research





Dissertation Research Goal

Programmable Hardware Acceleration

1. Explore the commonality in the way the DSAs specialize – ***Specialization Principles*** ✓

2. **General Mechanisms** for the design of a generic programmable hardware accelerator matching the efficiency of DSAs ✓

3. A programmable/re-configurable accelerator architecture with an efficient **accelerator hardware-software (ISA) interface**

4. **Easy adaptation** of new acceleratable algorithms in a domain-agnostic way

Contributions

Modeling Programmable Hardware Acceleration

- Exploring the common principles of architectural specialization
- Modeling a general set of mechanisms to exploit the specialization principles – GenAccel Model
- Quantitative evaluation of GenAccel Model with four DSAs
- System-Level Tradeoffs of GenAccel Model vs. DSAs

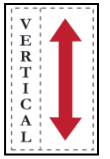
Architectural Realization with Stream-Dataflow Acceleration

- Stream-Dataflow programmable accelerator architecture with:
  - Programming abstractions and execution model
  - ISA interface
- Detailed micro-architecture with an efficient architectural realization of stream-dataflow accelerator – *Softbrain*
- Quantitative evaluation of Softbrain with state-of-the-art DSA solutions

Stream-Dataflow Acceleration*

*Published in ISCA 2017, Submitted to IEEE Micro Top-Picks 2018

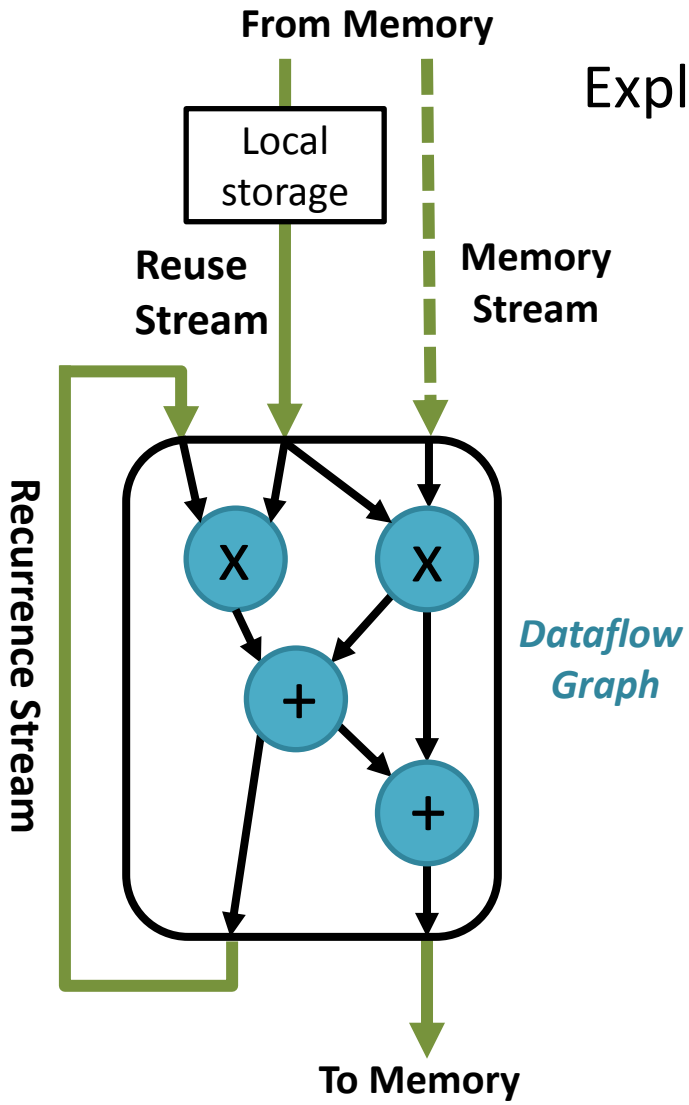
Architectural Realization of Programmable Hardware Acceleration



- Workloads characteristics:
  - Regular streaming memory accesses with straightforward patterns
  - Computationally intensive with long execution phases
  - Ample data-level parallelism with large datapath
  - Small instruction footprints with simple control flow

- Accelerator architecture to accelerate data-streaming applications
  - Instantiates the hardware primitives from GenAccel model
    - Exploit all the five specialization principles
  - Stream-Dataflow high-performance compute substrate with ***Dataflow*** and ***Stream*** specialization components
  - Exposes a novel stream-dataflow ISA interface for programming the accelerator

Stream-Dataflow Acceleration



Exploit common accelerator application behavior:

Dataflow Computation

- Stream-Dataflow **Execution model** – Abstracts typical accelerator computation phases

Stream Patterns and Interface

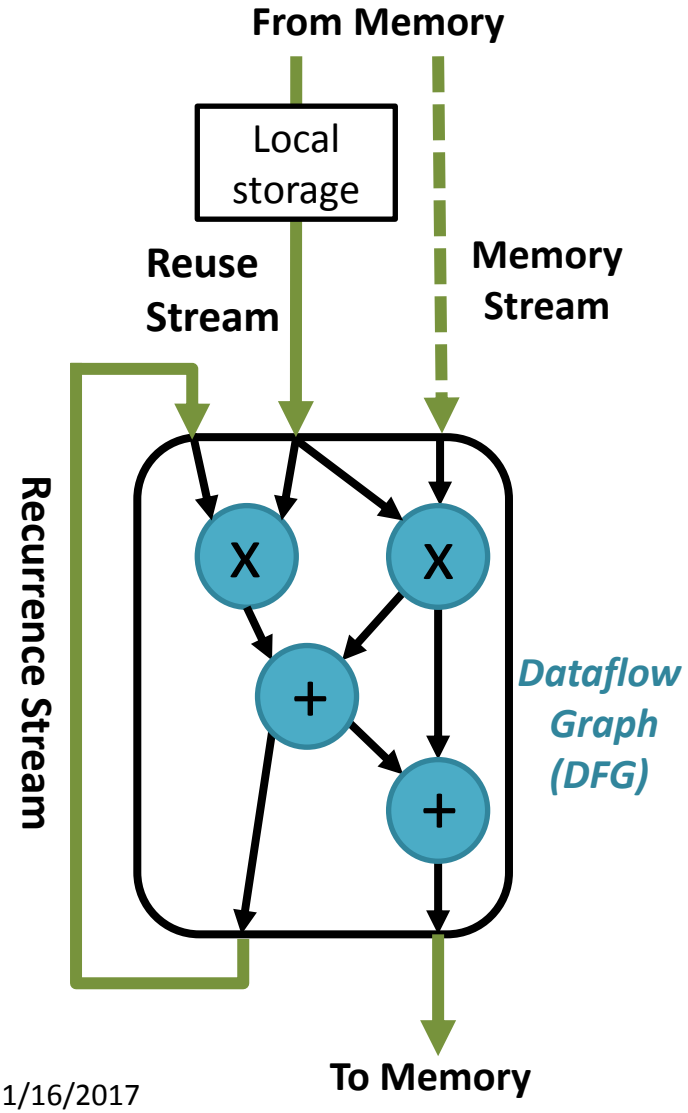
- Stream-Dataflow **ISA encoding** and **Hardware-Software interface** – Exposes parallelism available in these phases

Synchronization Primitives

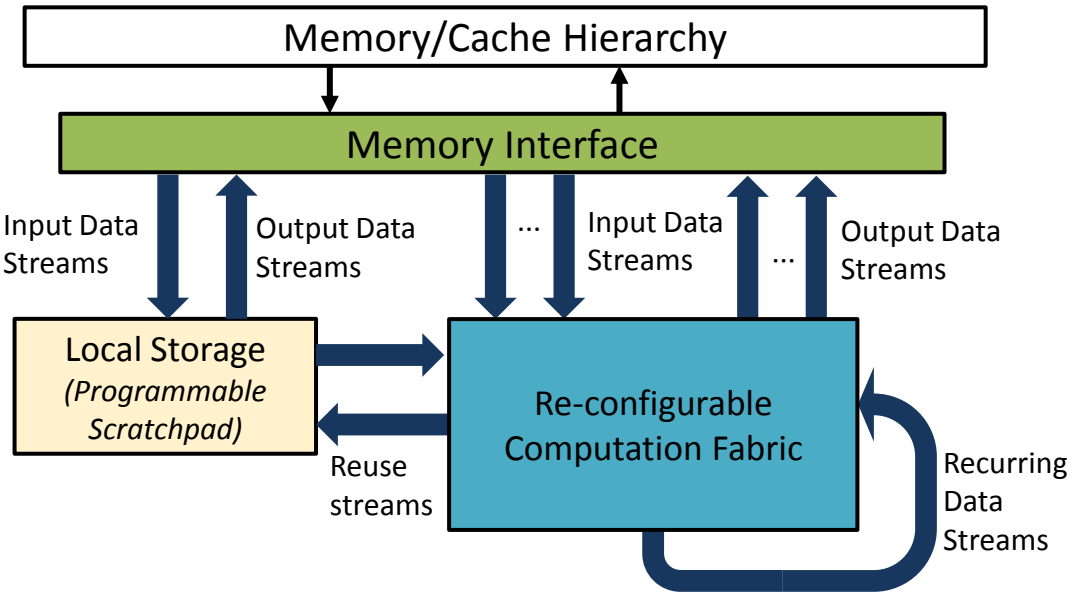
- Barrier commands to facilitate data coordination and data consistency

Stream-Dataflow Acceleration

Stream-Dataflow Model



Programmable Stream-Dataflow Accelerator

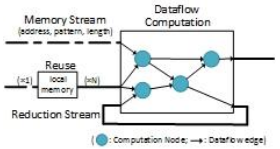


- Data-parallel program kernels streaming data from memory
- Dataflow computation fabric operates on data streams iteratively
- Computed output streams stored back to memory

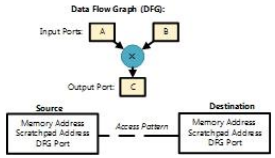
Outline

VERTICAL

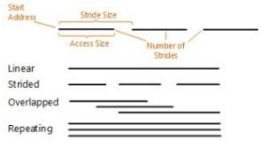
- Overview



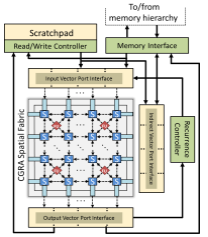
- Stream-Dataflow Execution Model



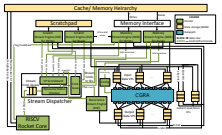
- Hardware-Software (ISA) Interface for Programmable Hardware Accelerator



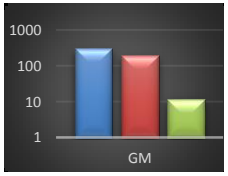
- Stream-Dataflow Accelerator Architecture and Example program



- Stream-Dataflow Micro-Architecture – *Softbrain*

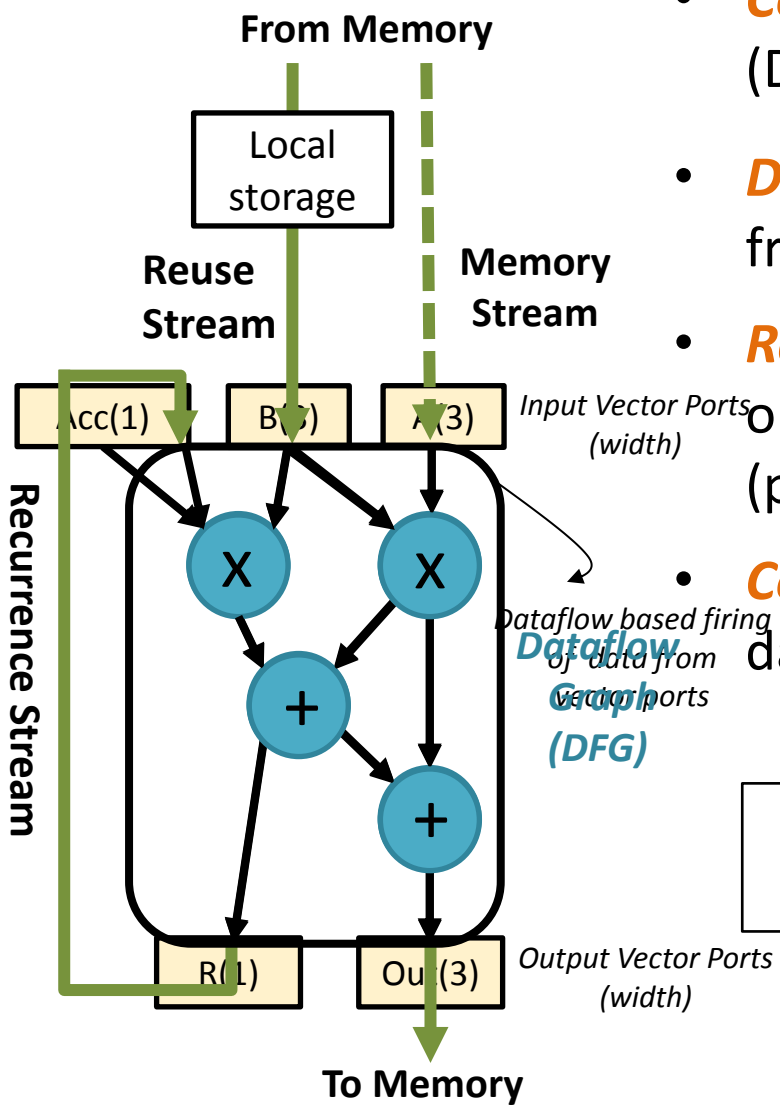


- Evaluation and Results

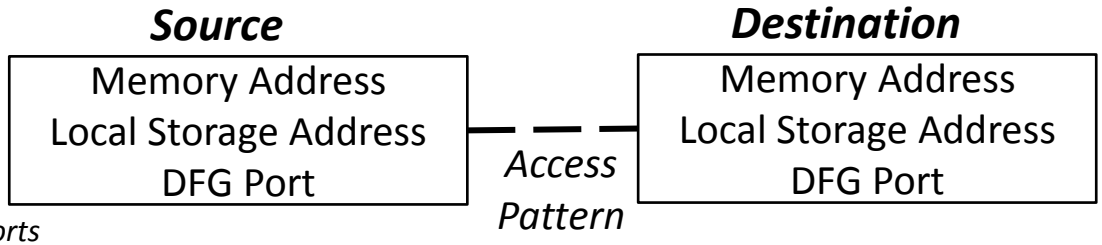


Stream-Dataflow Execution Model

Architectural Abstractions for Stream-Dataflow Model



- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused again
- ***Communication abstraction*** – Stream-Dataflow data movement commands and barriers

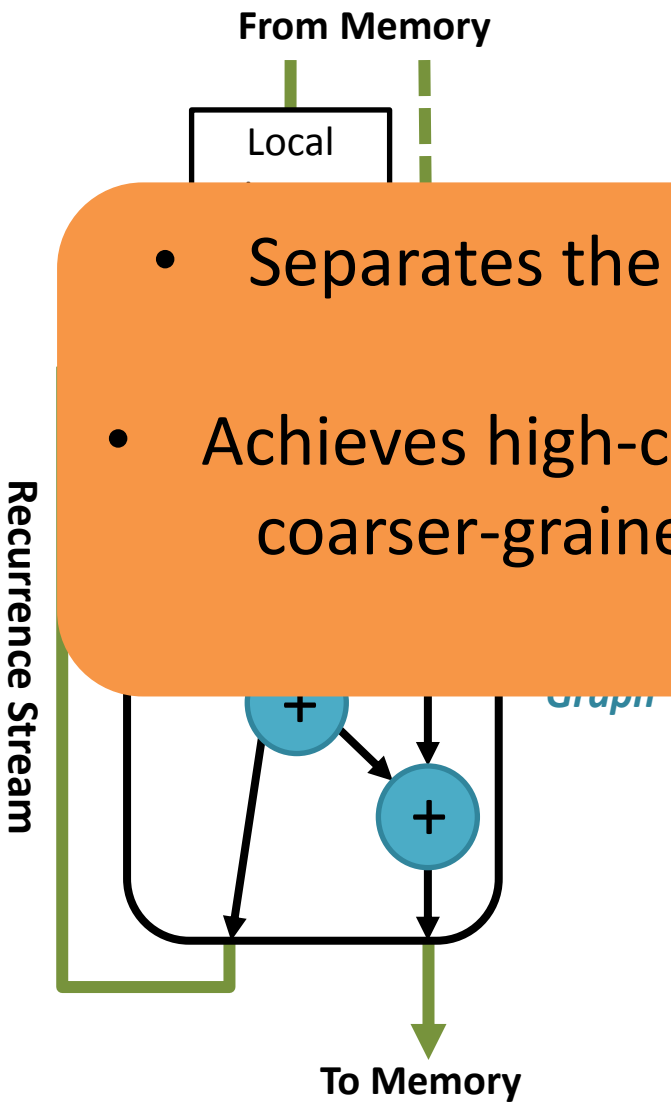


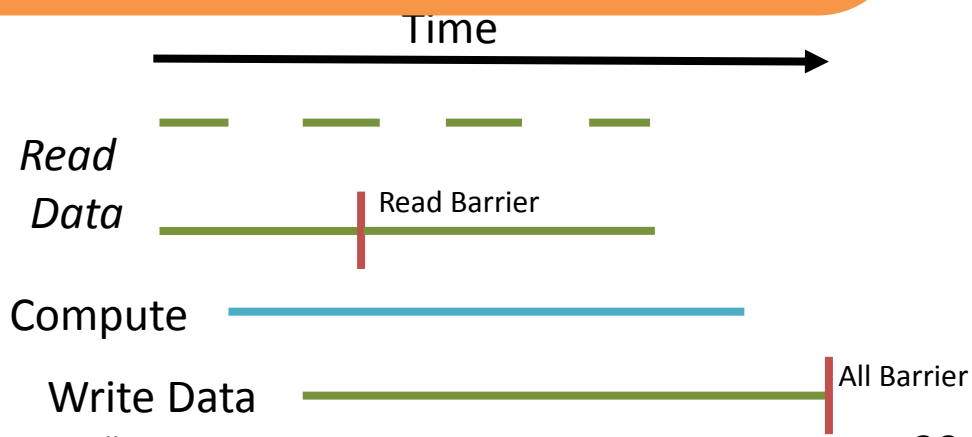
Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched

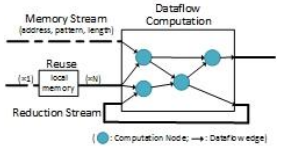
- Separates the data-movement from computation
- Achieves high-concurrency through the execution of coarser-grained data streams alongside dataflow computation

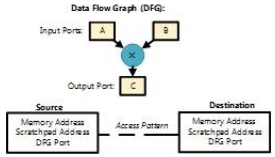


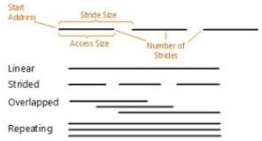


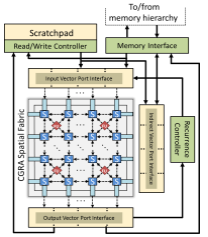
Outline

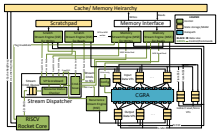
- Overview
- Stream-Dataflow Execution Model
- Hardware-Software (ISA) Interface for Programmable Hardware Accelerator
- Stream-Dataflow Accelerator Architecture and Example program
- Stream-Dataflow Micro-Architecture – *Softbrain*
- Evaluation and Results

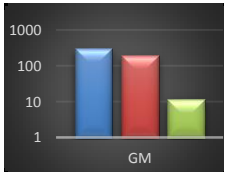












Traditional Arch.

Progammable Hardware Accelerator

Accelerator (DSA)

Programs

General Language

Compiler

General ISA

General Purpose Hardware

Programs ("Specialized")

H/W Parameters

H/W-S/W Interface

Re-Configurable Hardware

Domain-Specific Programs

Tiny H/W-S/W Interface

Application/Domain Specific Hardware

Can the specialized programs be adapted in a domain-agnostic way with this interface?

ance/Area
ability)

Stream-Dataflow ISA Interface

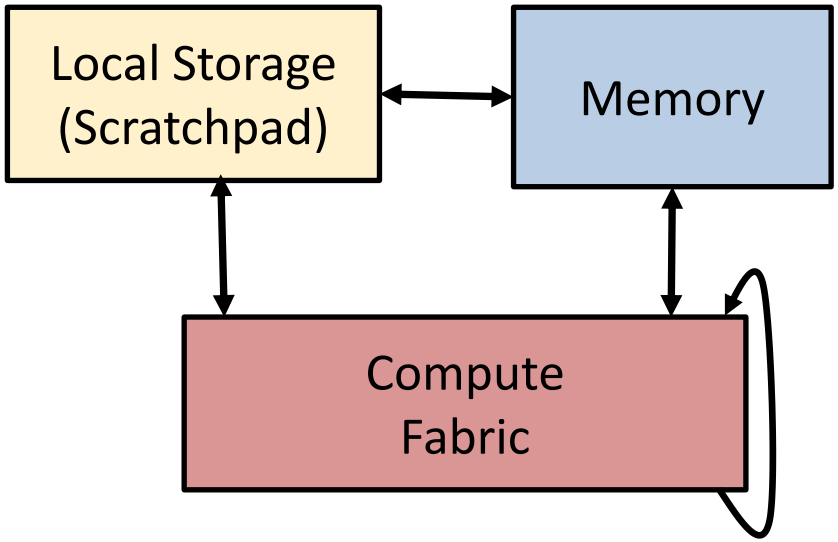
Express any data-stream pattern of accelerator applications using simple, flexible and yet efficient encoding scheme

Stream-Dataflow ISA

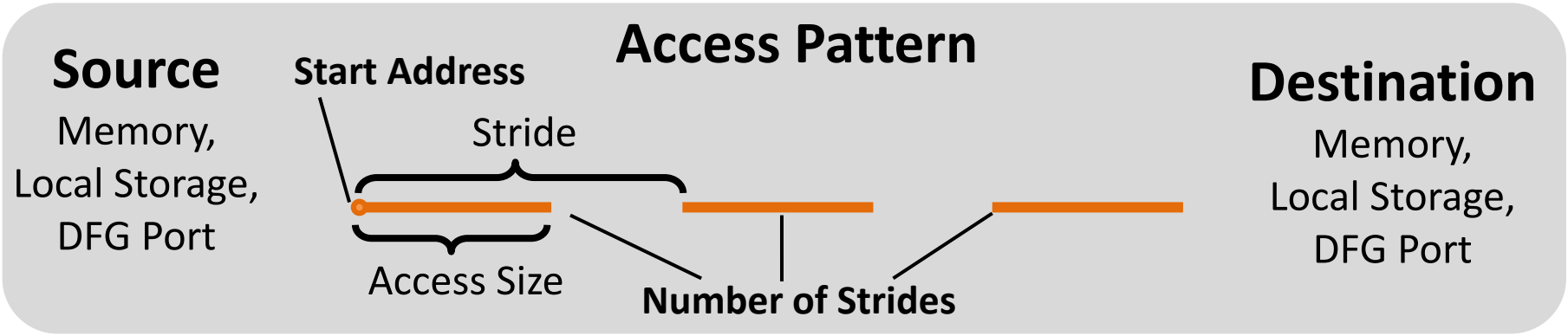
- Set-up Interface:
  `SD_Config` – Configuration data stream for dataflow computation fabric (CGRA)

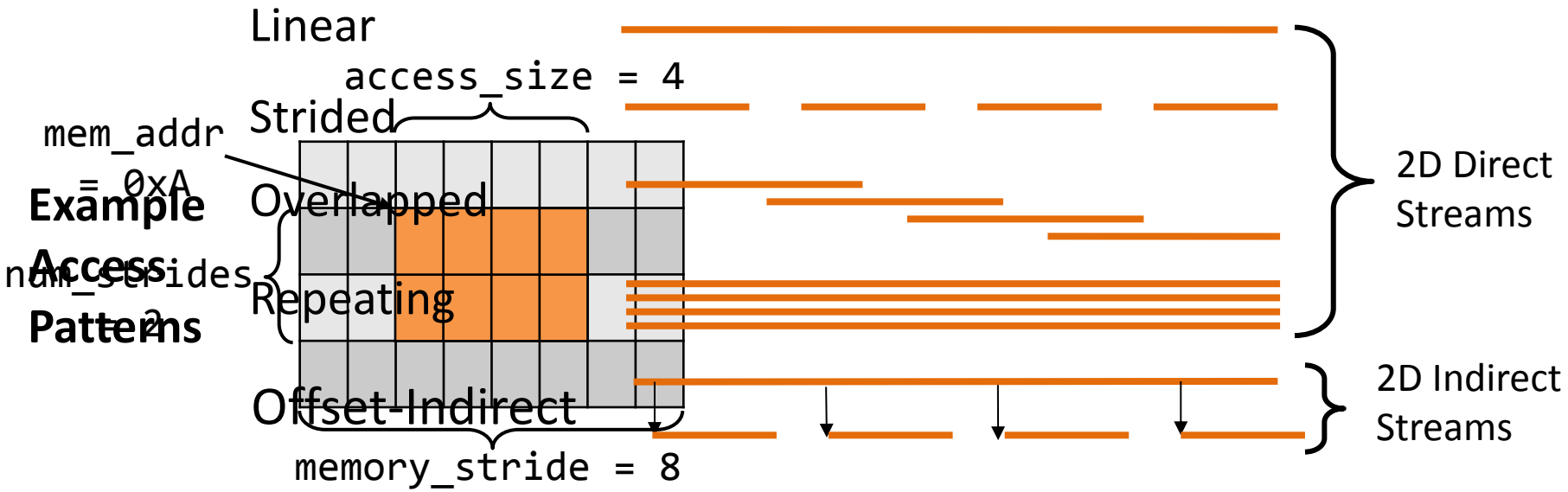
- Control Interface:
  `SD_Barrier_Scratch_Rd`, **`SD_Barrier_Scratch_Wr`**, **`SD_Barrier_All`**

- Stream Interface → **`SD_[source]_[dest]`**
  Source/~~Dest Parameters~~: *Address (memory or local_storage), DFG Port number*
  Pattern Parameters: *access_size, stride_size, num_strides*



Stream-Dataflow Programming Interface

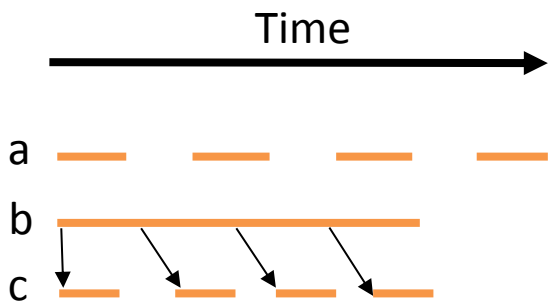




Stream-Dataflow ISA Encoding

Stream:

```
for i = 1 to 100:
  ... = a[2*i];
  ... = b[i];
  c[b[i]] = ...
```



Stream Encoding

<address, access_size, stride_size, length>

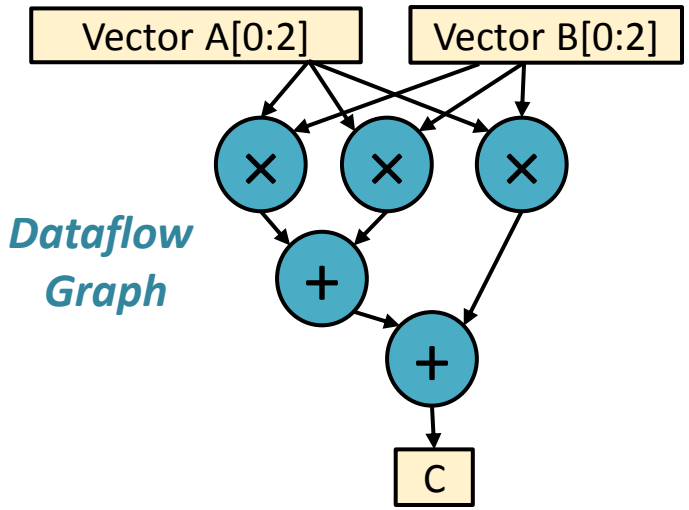
Eg: <a, 1, 2, 100>

<b, 1, 1, 100>

IND<[prev], c, 100>

<stream_start, offset_address>

Dataflow:



Dataflow Graph

Specified in a Domain Specific Language (DSL)

Example Pseudo-Code: Dot Product

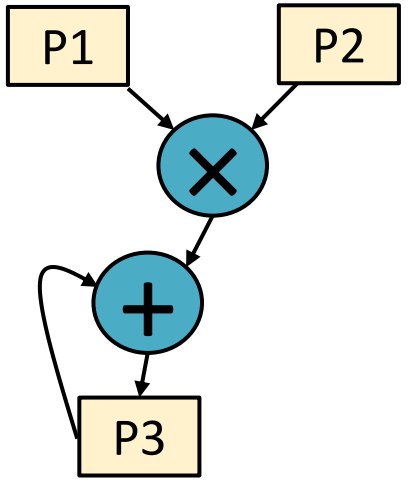
Original Program

```
for(int i = 0 to N) {
  c += a[i] * b[i];
}
```

Stream ISA Encoding

```
Put     a[0: N] → P1
Put     b[0: N] → P2
Recur   P3, N - 1
Get P3 → c
```

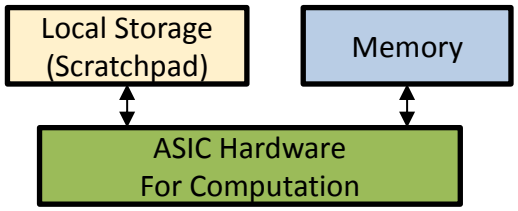
Dataflow Encoding



New ISA Class for Programmable Hardware Acceleration

Stream-Dataflow ISA

- Expresses long memory streams and access patterns efficiently
  - – Address generation hardware becomes much simpler
- Decouples access and execute phases
- Reduces instruction overheads
- Dependences are explicitly encoded
- Reduces cache requests and pressure by encoding alias-free memory requests
  - – Implicit coalescing for concurrent memory accesses
- Separates architecture abstractions from the implementation details

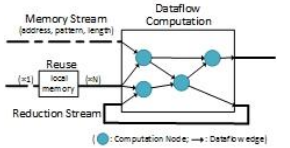


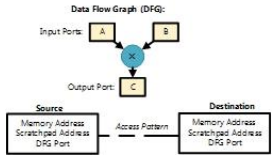
A New ISA Paradigm for Acceleration

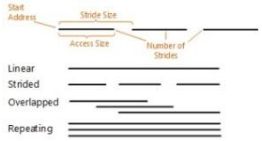
- Need to embody common accelerator principles and execution model
- Need to represent programs without requiring complex micro-architecture techniques for performance
  - – VLIW, SIMT and SIMD have their own drawbacks for accelerators
- Micro-Architecture for C-programmable ASICs
  - – Enables 'hardened' ASIC compute substrate implementation
  - – Separates the memory interface primitives and interaction

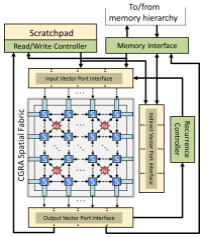
Outline

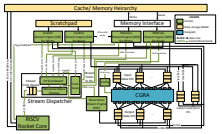
- Overview
- Stream-Dataflow Execution Model
- Hardware-Software (ISA) Interface for Programmable Hardware Accelerator
- Stream-Dataflow Accelerator Architecture and Example program
- Stream-Dataflow Micro-Architecture – *Softbrain*
- Evaluation and Results

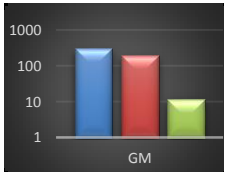






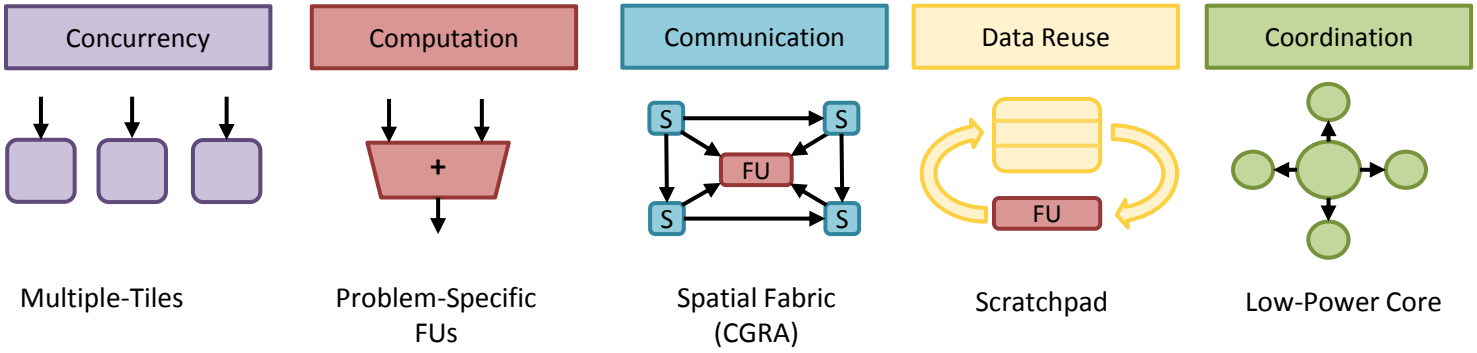






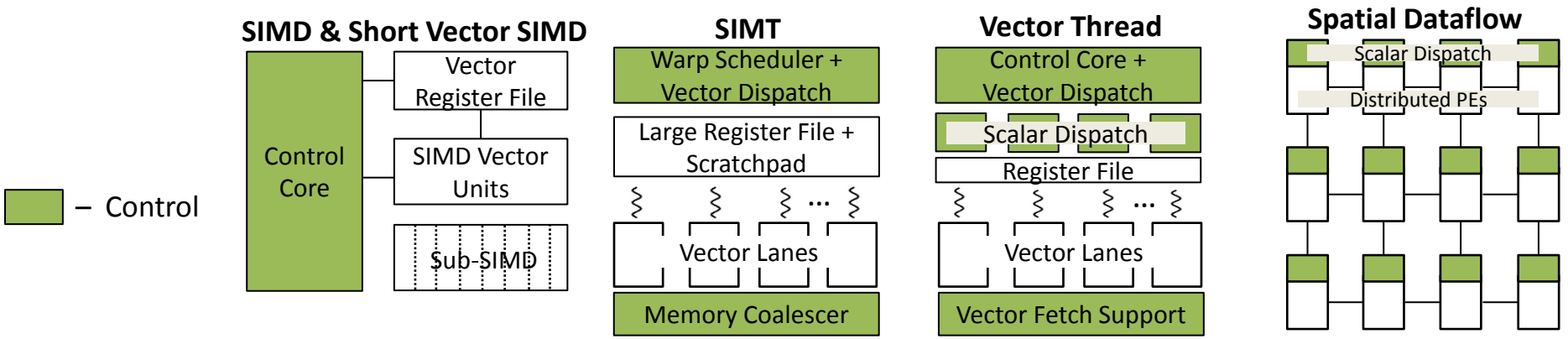
Requirements for Stream-Dataflow Accelerator Architecture

1. Should employ the common specialization principles and hardware mechanisms explored in GenAccel model
   (*IEEE Micro Top-Picks 2017: ***Domain Specialization is Generally Unnecessary for Accelerators***)



2. Programmability features without the inefficiencies of existing data-parallel architectures (with less power, area and control overheads)

Inefficiencies in Data-Parallel Architectures

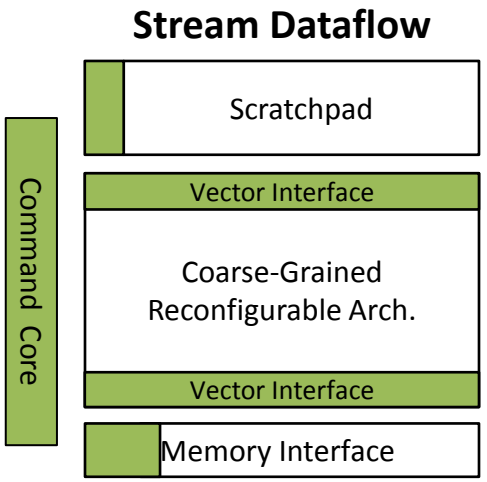


- ***Vector architectures*** – Efficient parallel memory interface
- ***Spatial Architectures*** – Efficient parallel computation interface
- ***Application/Domain Specific Architectures*** – Efficient datapath for pipelined concurrent execution

Irregular execution support	• Inefficient general pipeline	• Warp divergence hardware support	• Re-convergence for diverged vector threads	-

Stream-Dataflow Accelerator Architecture Opportunities

- Reduce address generation & duplication overheads
- Distributed control to boost pipelined concurrent execution
- High utilization of execution resources w/o massive multi-threading, reducing cache pressure or using multi-ported scratchpad
- Decouple access and execute phases of programs
- Simplest hardware fallback mechanism for irregular memory access support
- Able to be easily customizable/configurable for new application domain

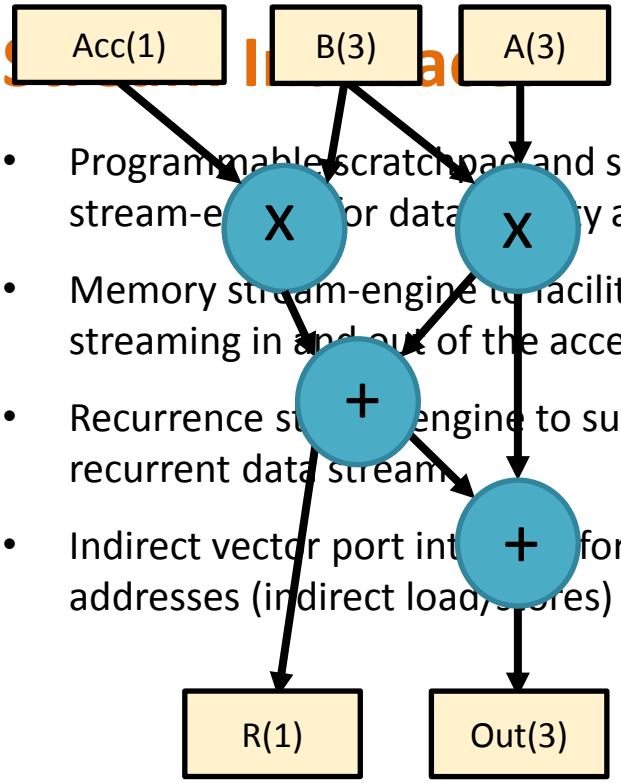


Stream-Dataflow Accelerator Architecture

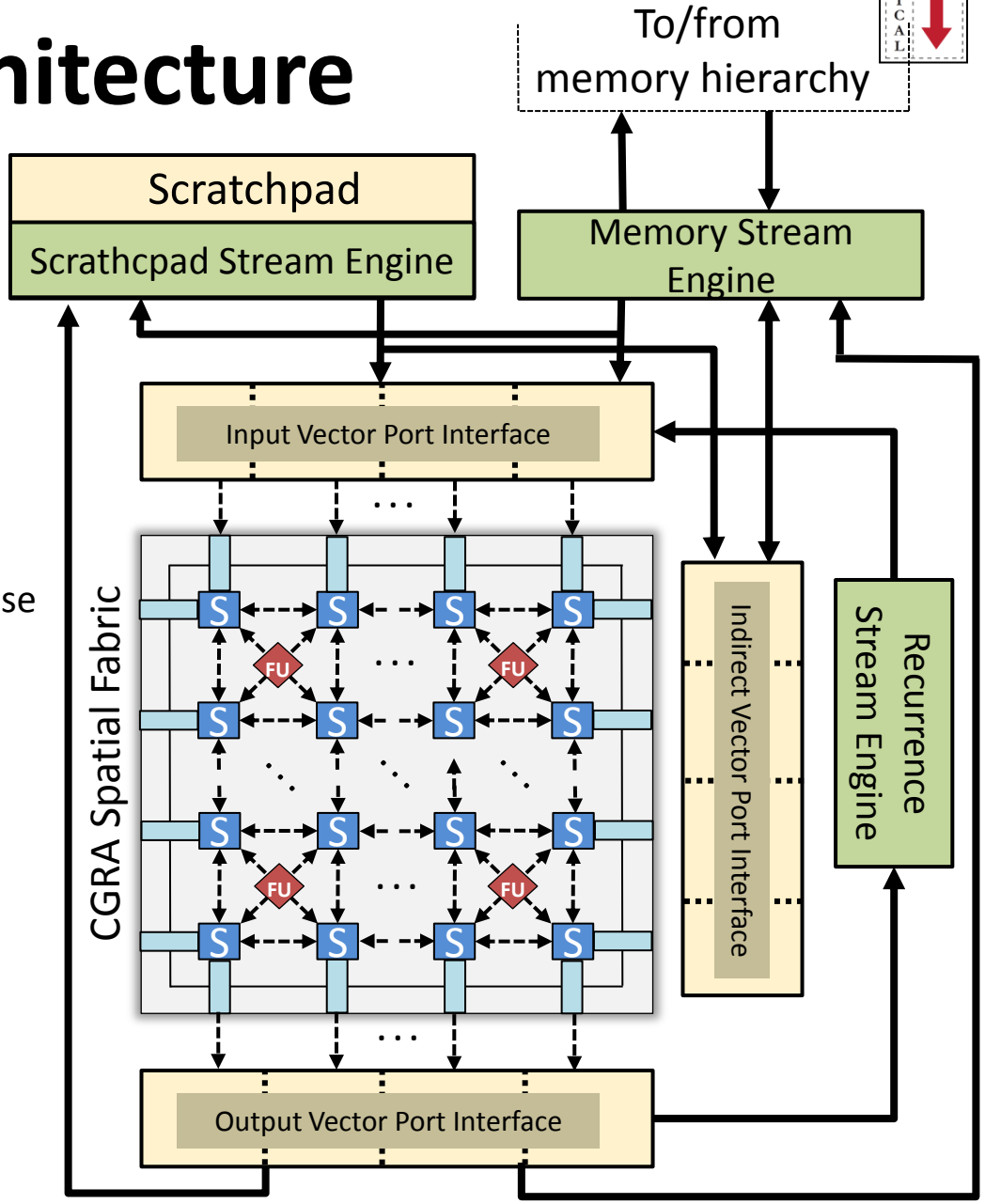
512b 64b

Dataflow:

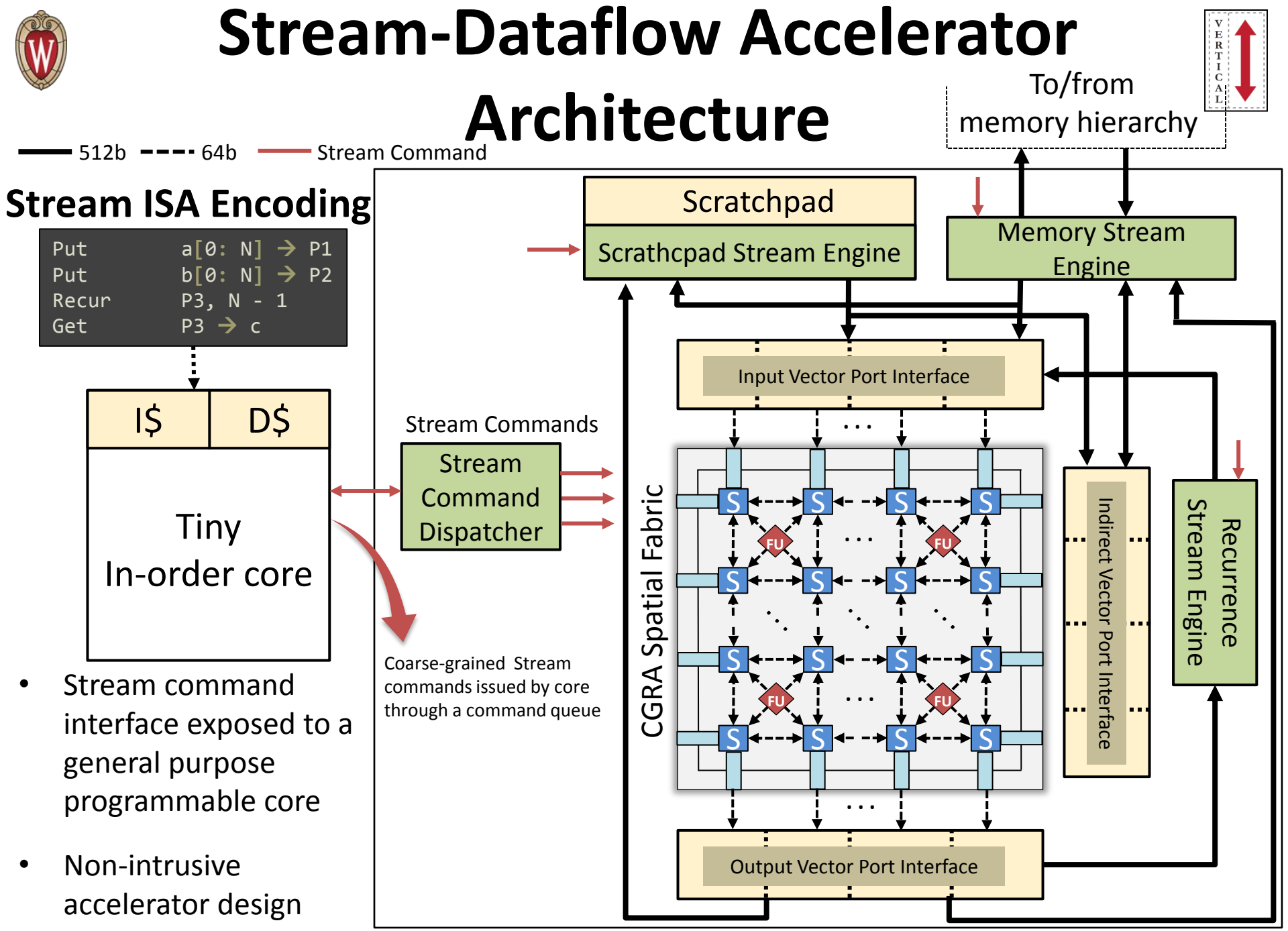
- Coarse grained reconfigurable architecture (CGRA) for data parallel execution
- Direct vector port interface into and out of CGRA for vector execution



- Programmable scratchpad and supporting stream-engine for data-locality and data-reuse
- Memory stream-engine to facilitate data streaming in and out of the accelerator
- Recurrence stream-engine to support recurrent data streams
- Indirect vector port interface for streaming addresses (indirect load/stores)



Stream-Dataflow Accelerator Architecture



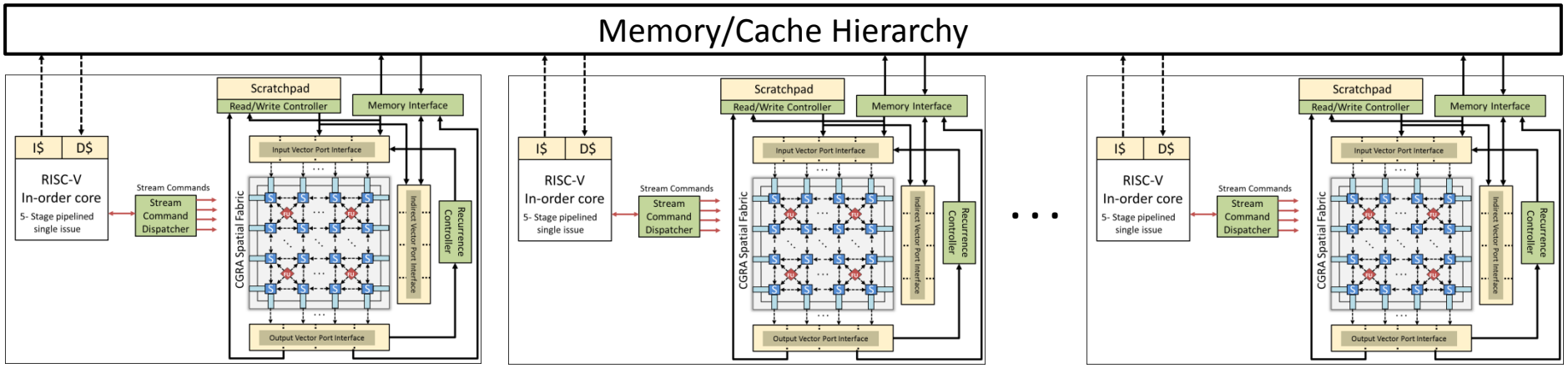
Stream ISA Encoding

```
Put     a[0: N] → P1
Put     b[0: N] → P2
Recur   P3, N - 1
Get     P3 → c
```

- Stream command interface exposed to a general purpose programmable core
- Non-intrusive accelerator design

Stream-Dataflow Accelerator Architecture Integration

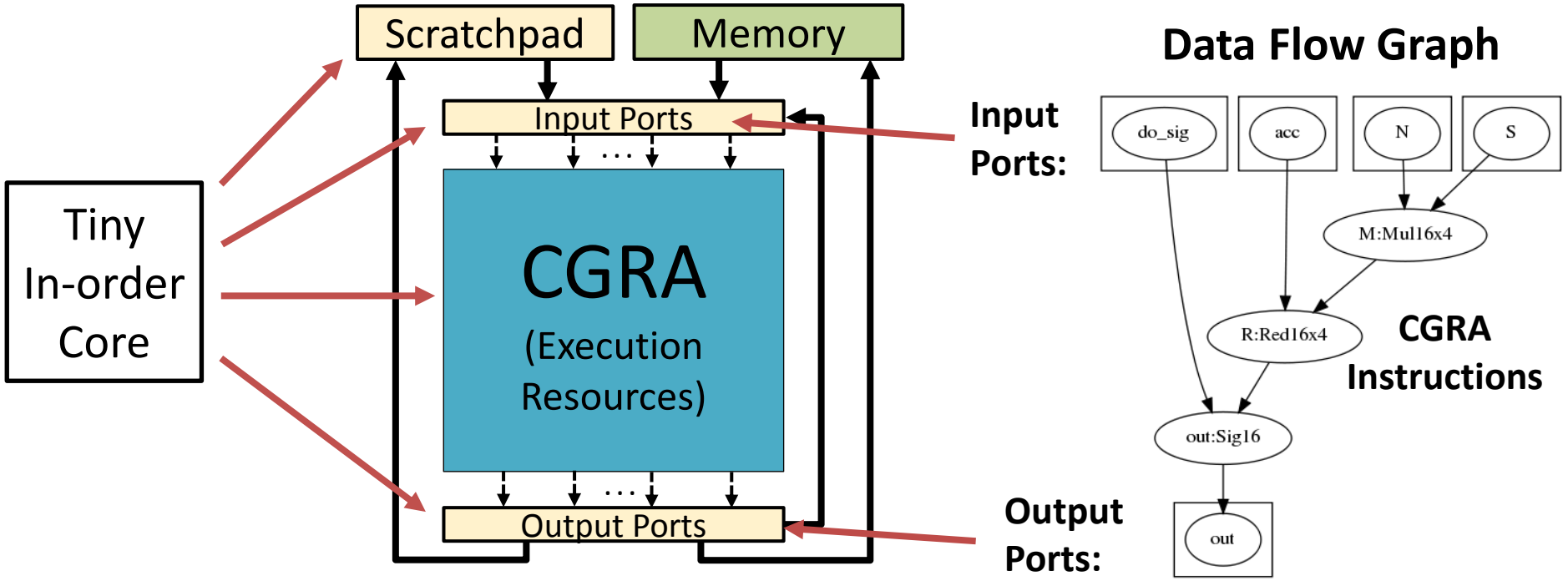
Multi-Tile Stream-Dataflow Accelerator



- Each tile is connected to higher-L2 cache interface
- Need a simple scheduler logic to schedule the offloaded stream-dataflow kernels to each tile

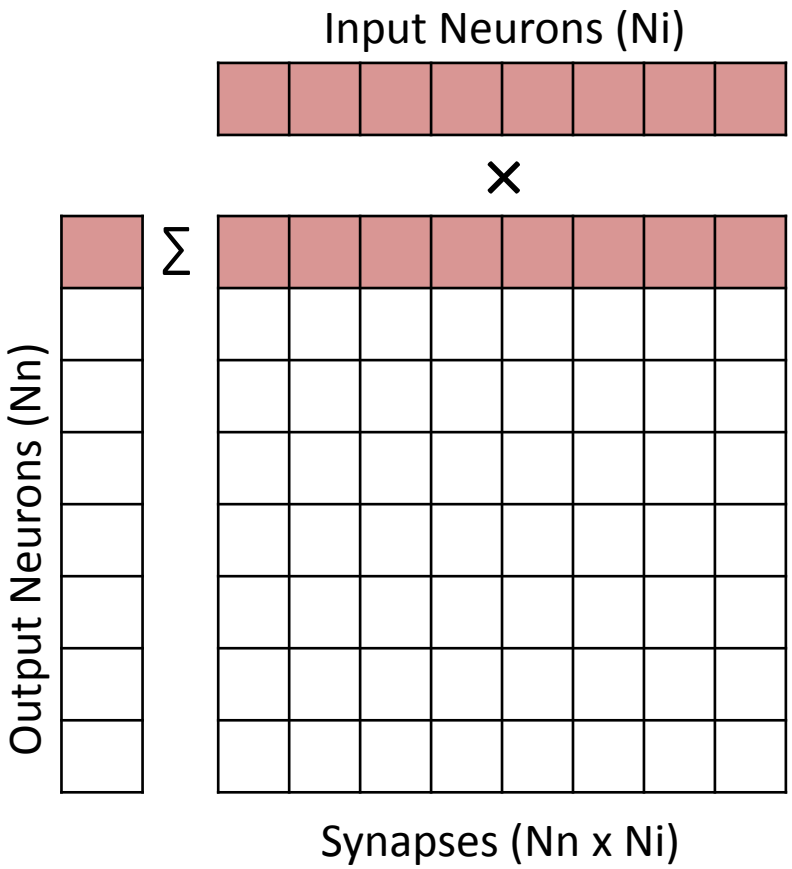
Programming Stream-Dataflow Accelerator

1. Specify Datapath for the CGRA
   - Simple Dataflow Language for DFG
2. Orchestrate the parallel execution of hardware components
   - Coarse-grained stream commands using the stream-interface



Classifier Layer (Original)

```
#define Ni 8
#define Nn 8

// synapse and neurons - 2 bytes
uint16_t synapse[Nn][Ni];
uint16_t neuron_i[Ni];
uint16_t neuron_n[Nn];

for (n = 0; n < Nn; n++) {
  sum = 0;

  for (i = 0; i < Ni; i++) {
    sum += synapse[n][i] * neuron_i[i];
  }

  neuron_n[n] = sigmoid(sum);
}
```



Dataflow Graph (DFG) for CGRA: *Classifier Kernel*

Computation DFG for

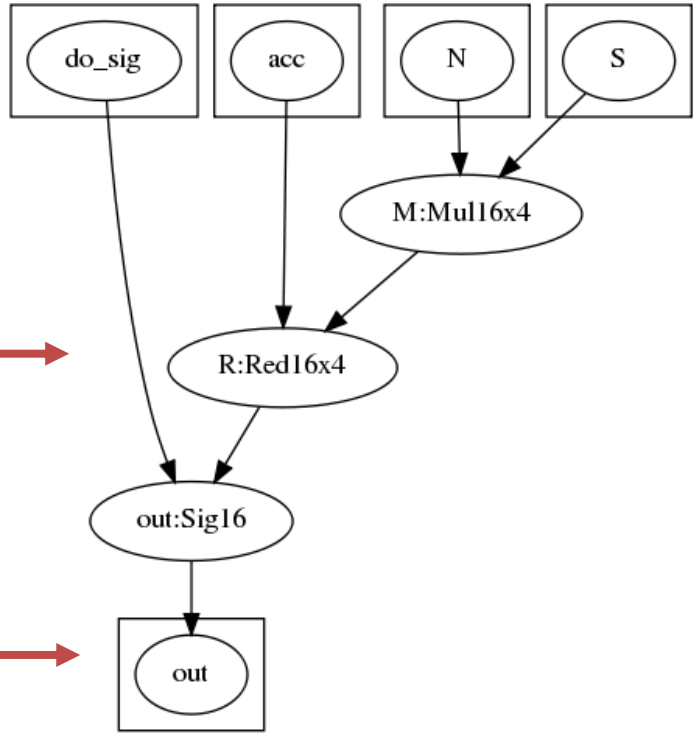
```
sum += synapse[n][i] * neuron_i[i];
neuron_n[n] = sigmoid(sum);
```

```
Input: do_sig
Input: acc
Input: N
Input: S

M = Mul16x4(N, S)
R = Red16x4(M, acc)
out = Sig16(R, do_sig)


Output: out
```

Input Ports:

CGRA Instructions

Output Ports:



N – *Input neuron (Ni) port*
S – *Synapses (synapse) port*
do_sig – *Input sigmoid predicate port*
acc – *Input accumulate port*
out – *Output neurons (No) port*

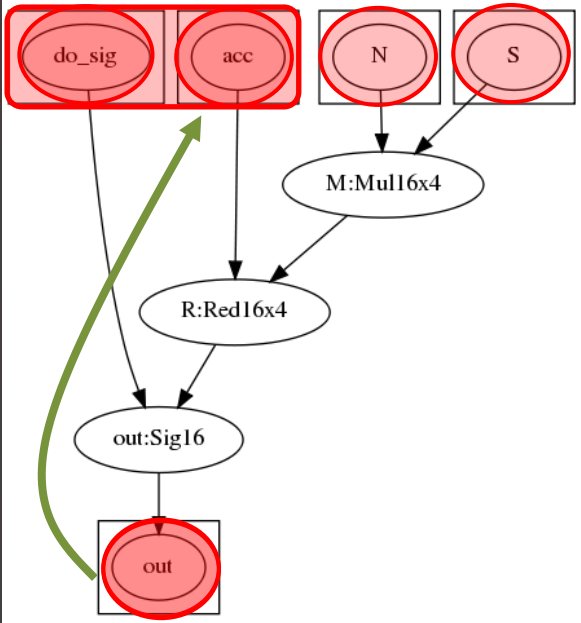
Compilation + Spatial scheduling

class_cfg

(Configuration data for CGRA)

Stream Dataflow Program: *Classifier Kernel*

```
// Configure the CGRA
SD_CONFIG(class_cfg, sizeof(class_cfg));

// Stream the data from memory to ports
SD_MEM_PORT(synapse,  8, 8, Ni * Nn/ 4, Port_S);
SD_MEM_PORT(neuron_i, 8, 8, Ni/4,       Port_N);

for (n = 0; n < Nn/nthreads; n++) {
  // Stream the constant values to constant ports
  SD_CONST(Port_acc,    0, 1);
  SD_CONST(Port_do_sig, 0, Ni - 1);

  // Recur the computed data back for accumulation
  SD_PORT_PORT(Port_out, N - 1, Port_acc);

  // Sigmoid computation and output neuron written
  SD_CONST(Port_do_sig, 1, 1);
  SD_PORT_MEM(Port_out, 2, 2, 1, &neuron_n[n]);
}

SD_BARRIER_ALL();
```



Compilation + Spatial scheduling

class_cfg *(Configuration data for CGRA)*

Performance Considerations

- ***Goal:*** Fully pipeline the largest dataflow graph
  - Increase performance [**CGRA Instructions / Cycle**]
  - Increase throughput [**Graph computation instances per cycle**]

- Primary Bottlenecks:
  - Computations per Size of Dataflow Graph

    Increase through Loop Unrolling/Vectorization
  - General Core (for Issuing Streams)

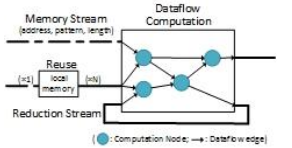
    Increase "length" of streams
  - Memory/Cache Bandwidth

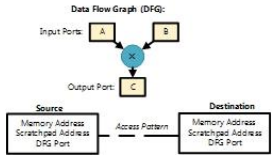
    Use Scratchpad for data-reuse
  - Recurrence Serialization Overhead

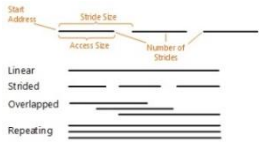
    Increase Parallel Computations (tiling)

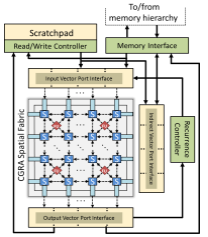
Outline

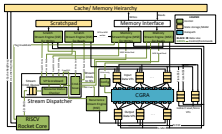
- Overview
- Stream-Dataflow Execution Model
- Hardware-Software (ISA) Interface for Programmable Hardware Accelerator
- Stream-Dataflow Accelerator Architecture and Example program
- Stream-Dataflow Micro-Architecture – *Softbrain*
- Evaluation and Results

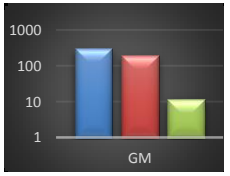


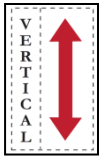












Micro-Architecture Design Principles

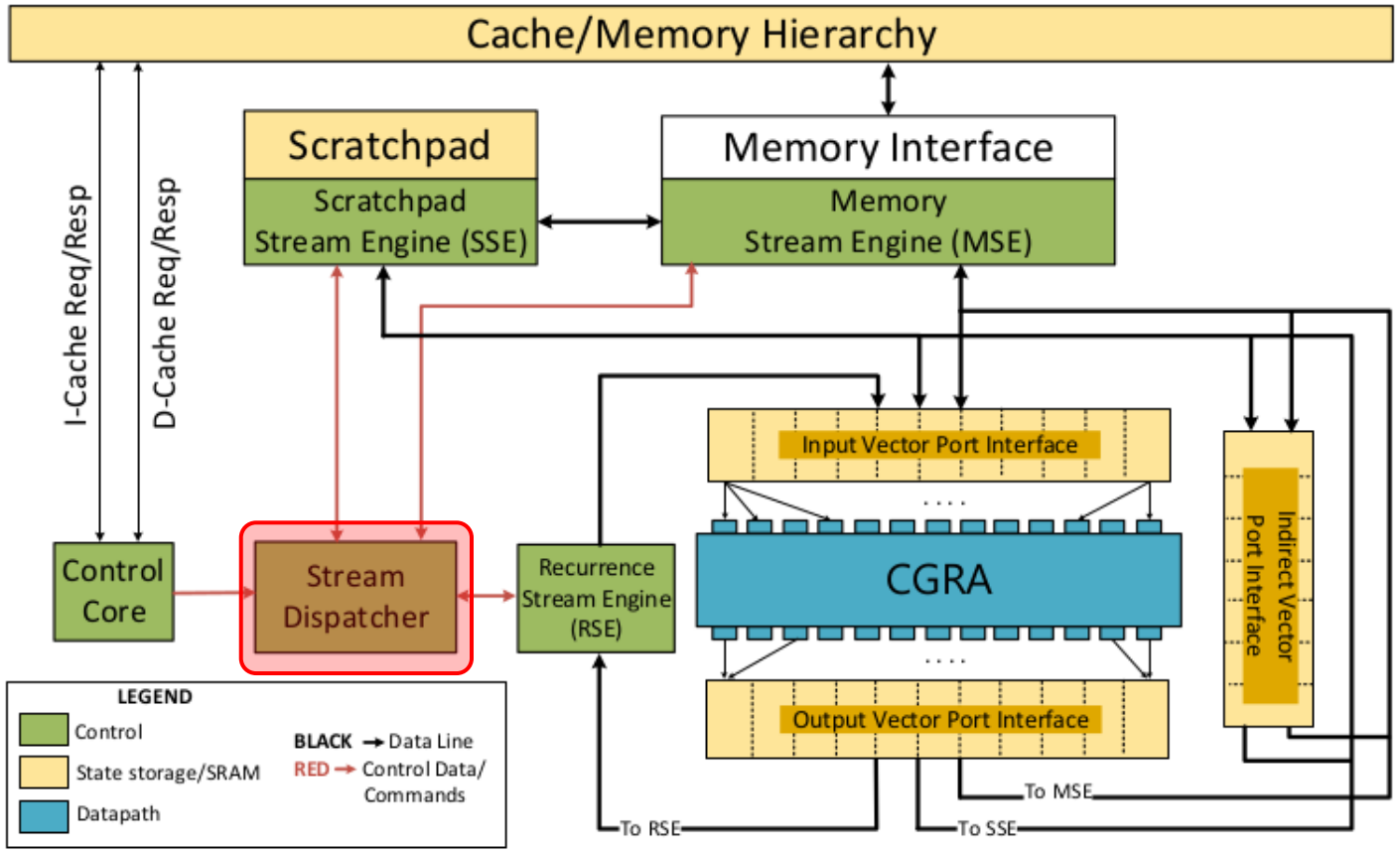
1. Low-overhead control structures

2. Efficient execution of concurrent stream commands with simple resource dependency tracking

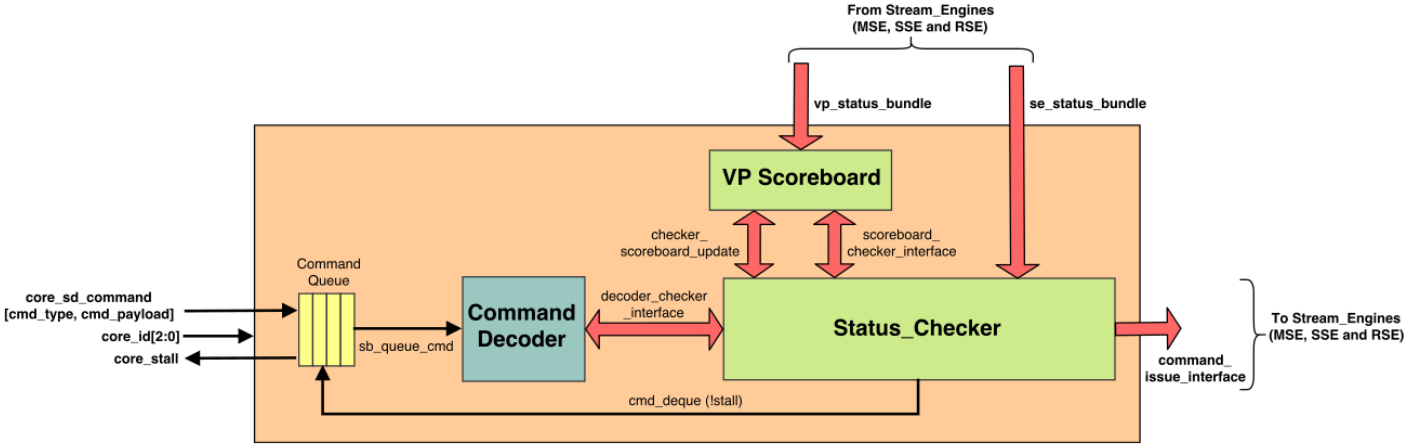
3. Not introduce power hungry or large CAM-like structures

4. Parameterizable design

Micro-Architecture of Stream-Dataflow Accelerator – *Softbrain*

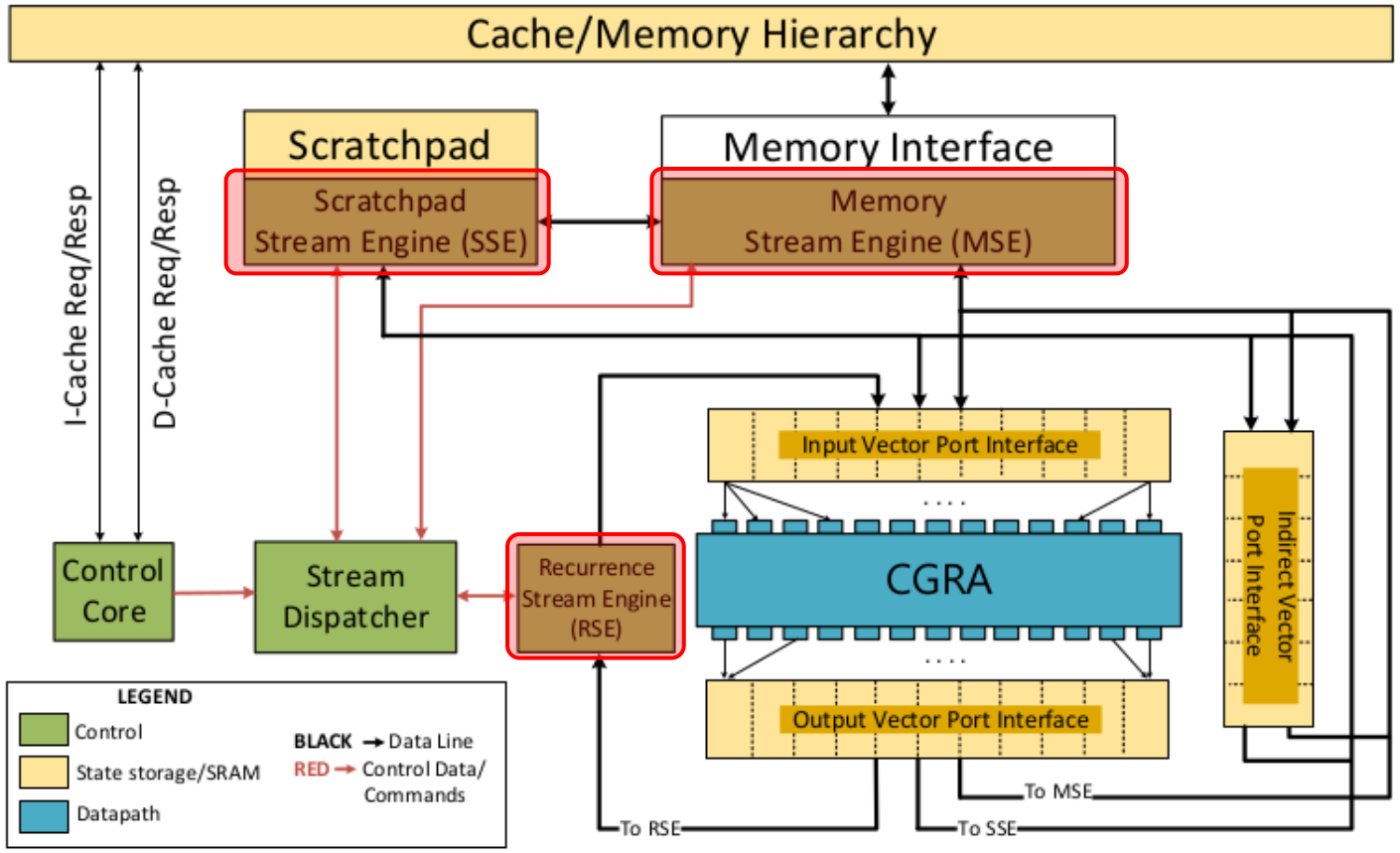


Stream-Dispatcher of *Softbrain*

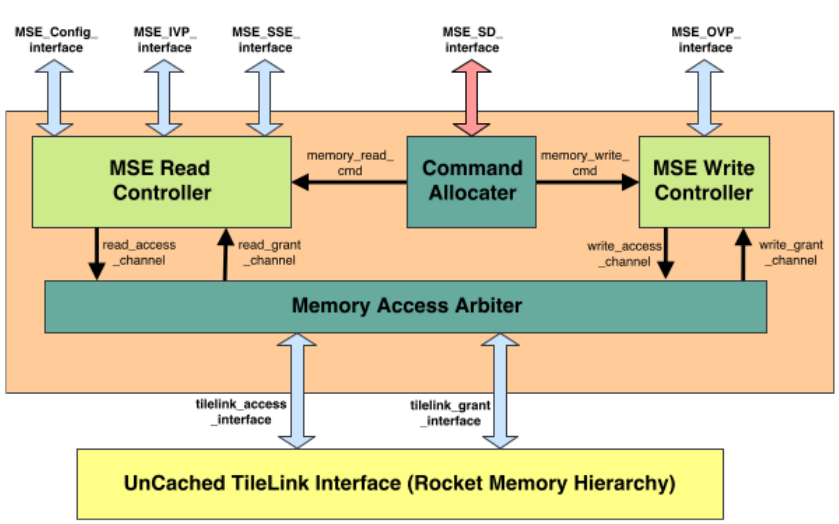


- Issues the stream commands to stream-engines
- Resource dependency tracking
  - Simple vector-port to stream-engine scoreboard mechanism
- Barriers – Enforces the explicit stream-barriers for data-consistency in scratchpad as well as memory state
- Interfaces to the low-power core using a simple queue-based custom accelerator logic

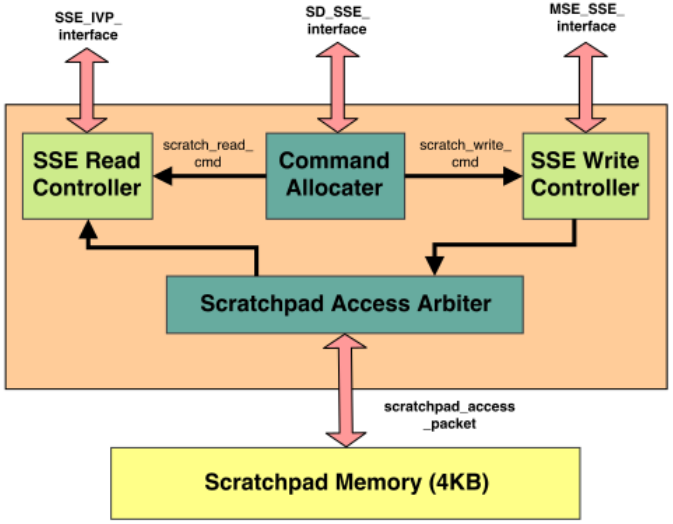
Micro-Architecture of Stream-Dataflow Accelerator – *Softbrain*



Stream-Engine of *Softbrain*



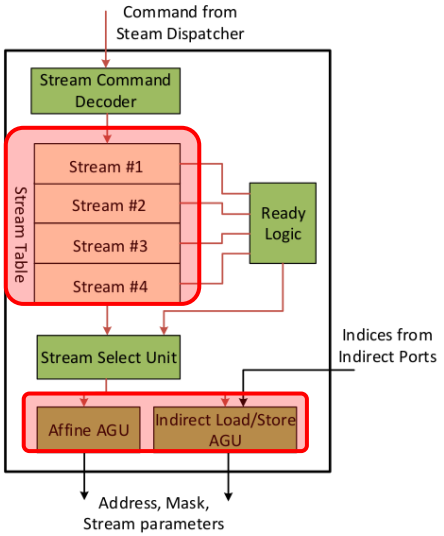
Memory Stream-Engine (MSE)



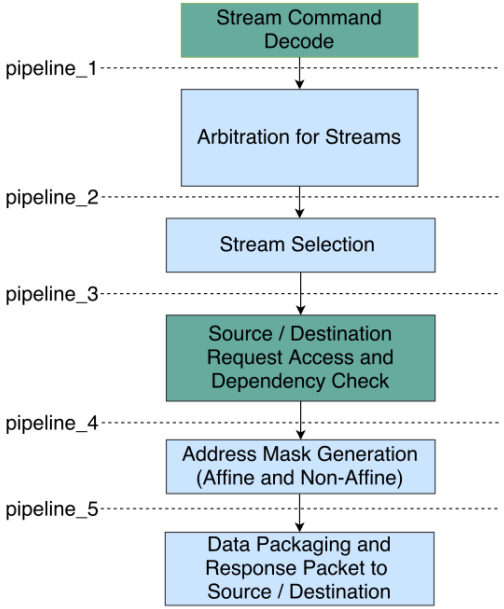
Scratchpad Stream-Engine (SSE)

- Arbitration of multiple stream command requests
- Responsible for address generation for various data-stream access patterns
- Manages concurrent accesses to vector ports, scratchpad and the cache/memory hierarchy
- Dynamic switching of streams to account for L2 cache misses and maintain the high-bandwidth memory accesses

Softbrain Stream-Engine Controller Request Pipeline



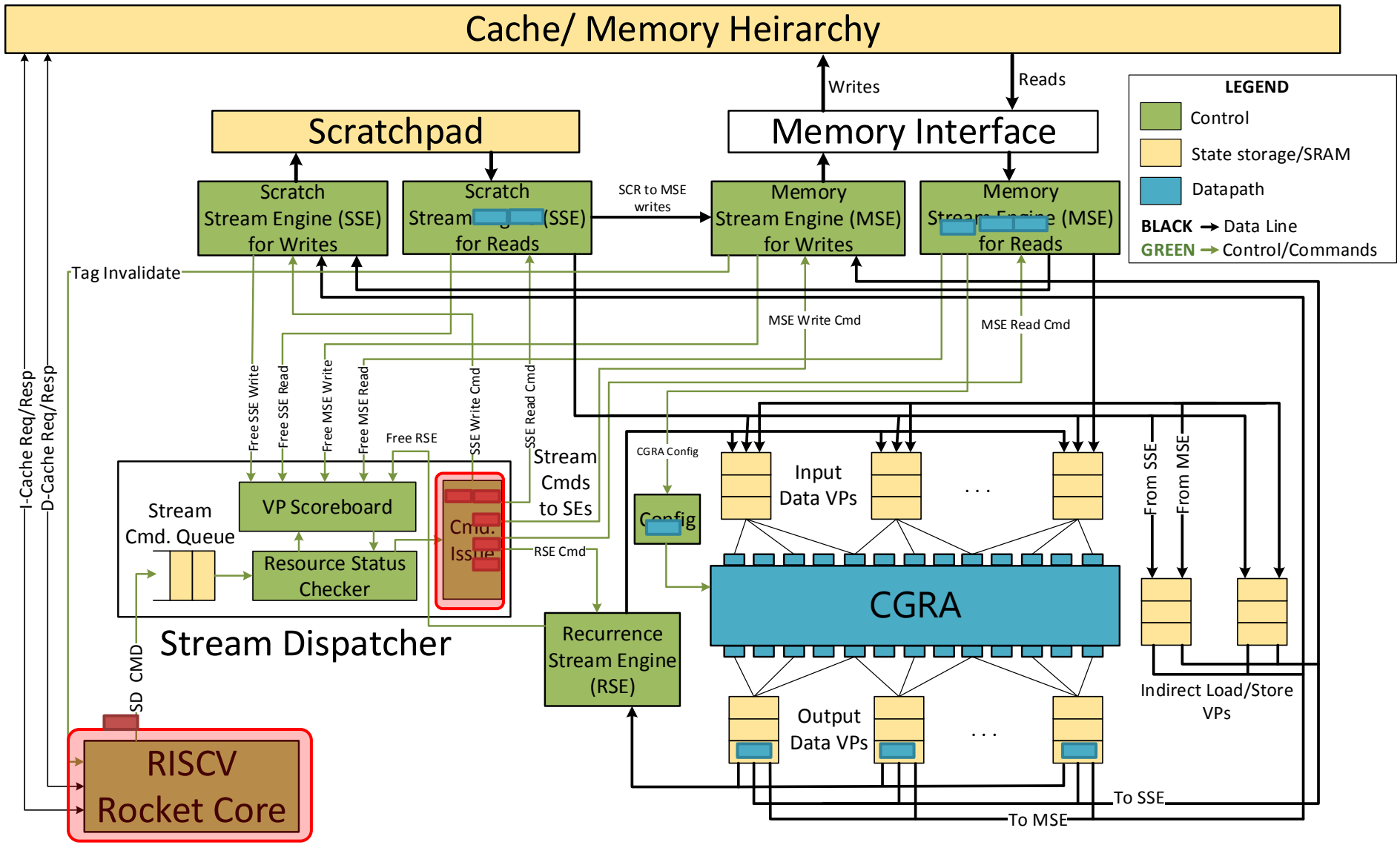
Stream-Engine Controller



Stream Request Pipeline

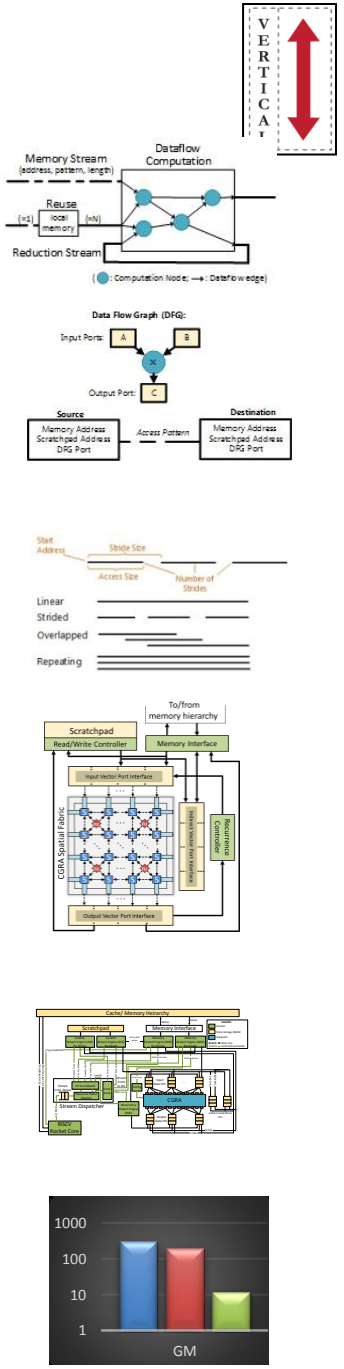
- Responsible for address generation for both *direct* and *indirect* data-streams
- Priority based selection among multiple queued data-steams
- Direct streams – Affine Address Generation Unit (AGU) generates memory addresses
- Indirect Streams – Non-affine AGU gets addresses, offsets from indirect vector ports

Micro-Architecture Flow of *Softbrain*

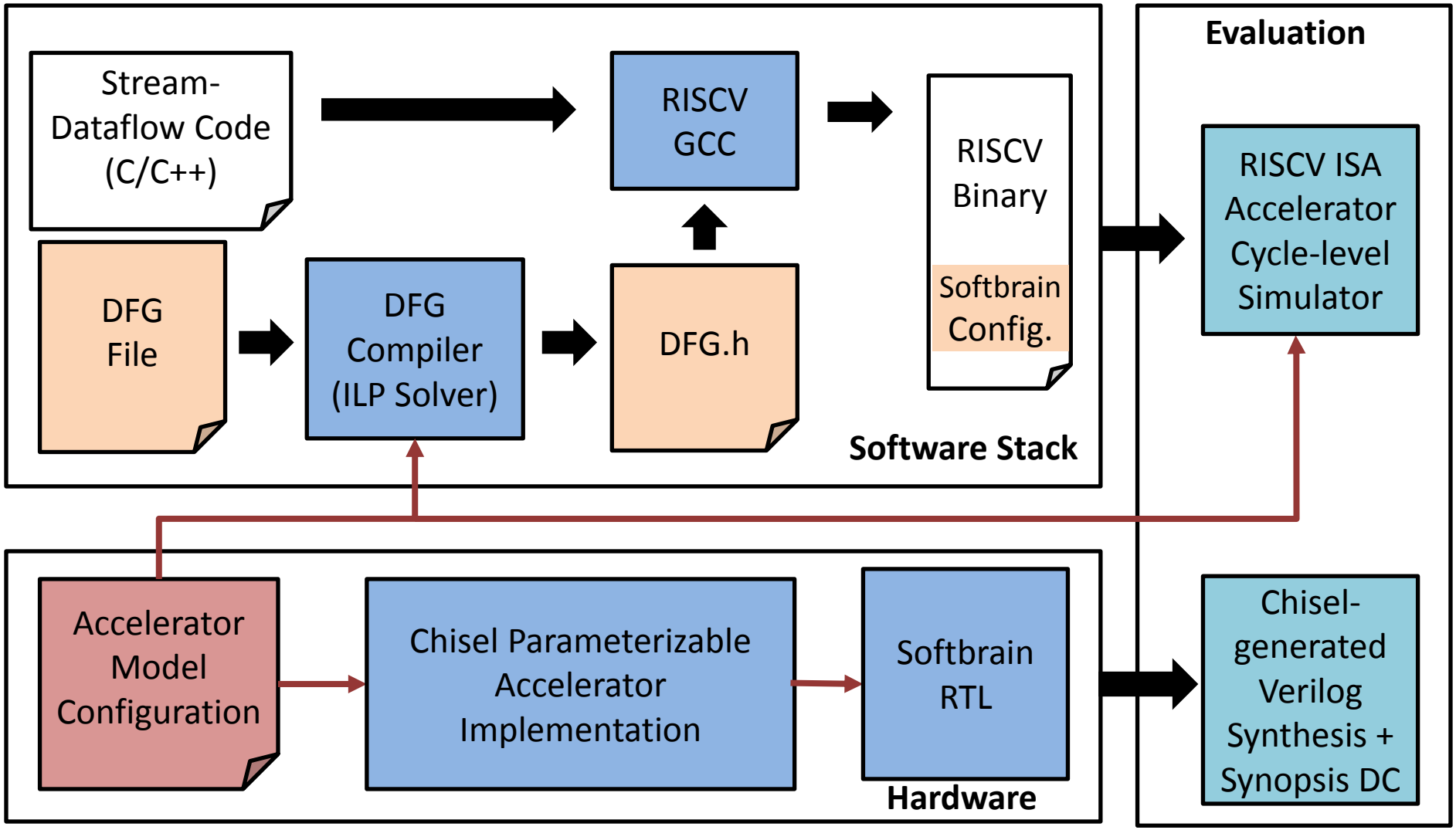


Outline

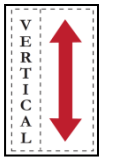
- Overview
- Stream-Dataflow Execution Model
- Hardware-Software (ISA) Interface for Programmable Hardware Accelerator
- Stream-Dataflow Accelerator Architecture and Example program
- Stream-Dataflow Micro-Architecture – *Softbrain*
- Evaluation and Results



Stream-Dataflow Implementation: *Softbrain*



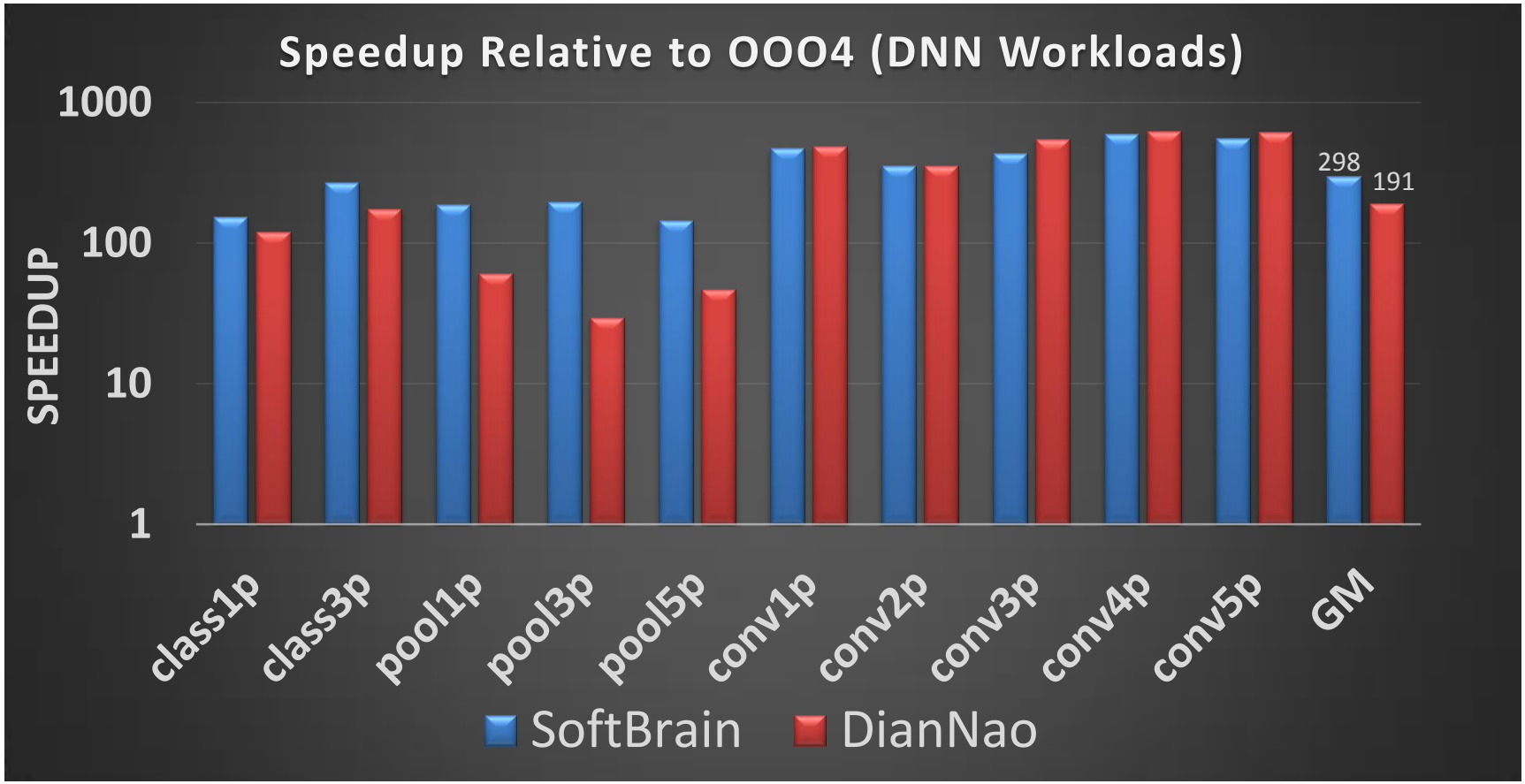
Evaluation Methodology



- Workloads
  - Deep Neural Networks (DNN) – For **domain provisioned** comparison
  - Machsuite Accelerator Workloads – For comparison with **application specific** accelerators
- Comparison
  - **Domain Provisioned Softbrain** vs. DianNao DSA
  - **Broadly provisioned Softbrain** vs. ASIC design points – ***Aladdin**** generated performance, power and area
- Area and Power of Softbrain
  - Synthesized area, power estimates
  - CACTI for cache and SRAM estimates

*Sophia, Shao et al. – *Aladdin: a Pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures*

Domain-Specific Comparison (Softbrain vs DianNao DSA)



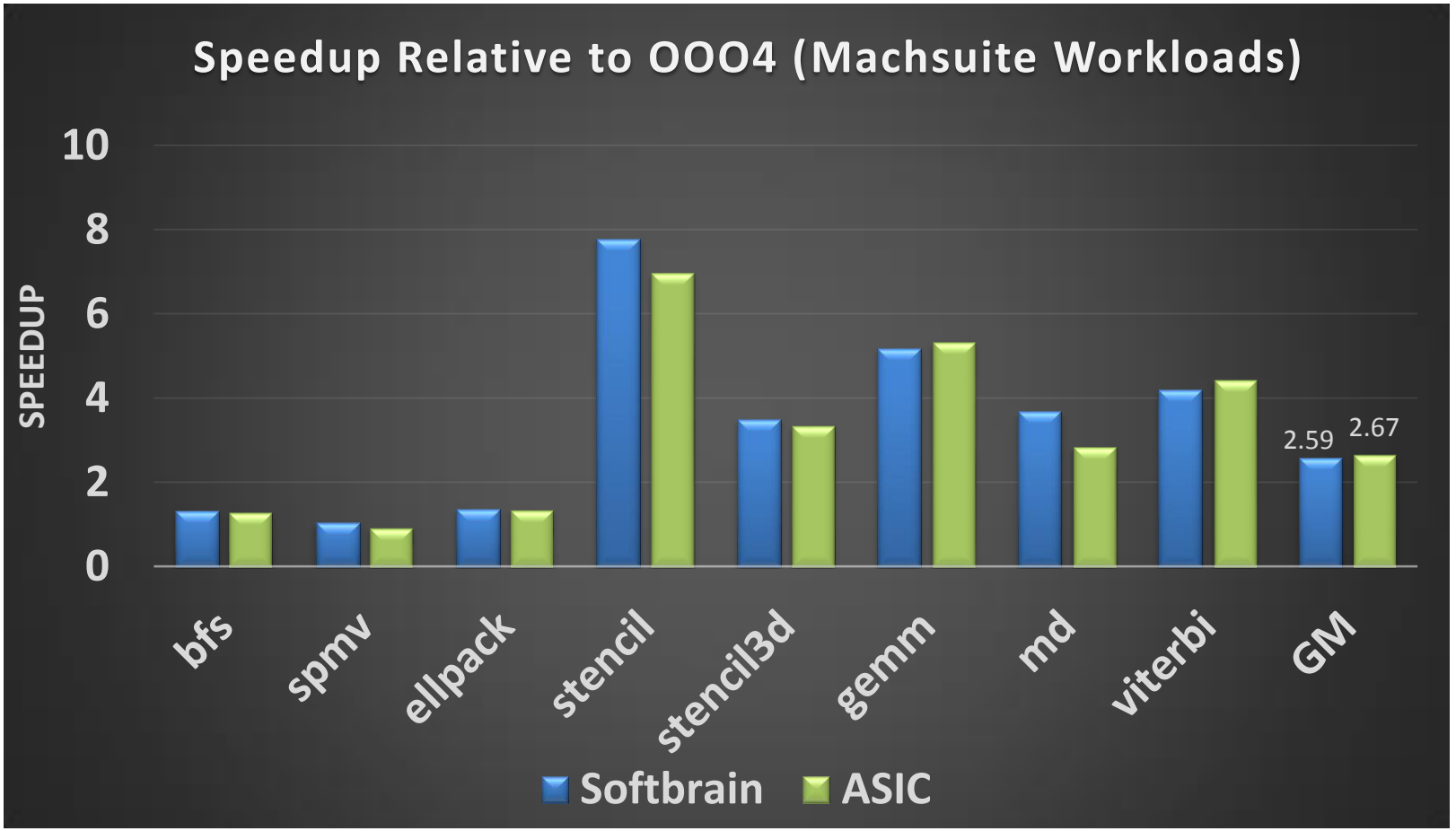
Area-Power Estimates of Domain Provisioned Softbrain

Components		Area (mm2) @ 28nm	Power (mW)
1 Softbrain Unit		0.47	119.3
8 Softbrain Units		3.76	954.4
DianNao DSA		2.16	418.3
Softbrain / DianNao Overhead		**1.74**	**2.28**

Softbrain vs Diannao (DNN DSA)

- *Perf. – Able to match the performance*
- *Area – **1.74x** Overhead*
- *Power – **2.28x** Overhead*

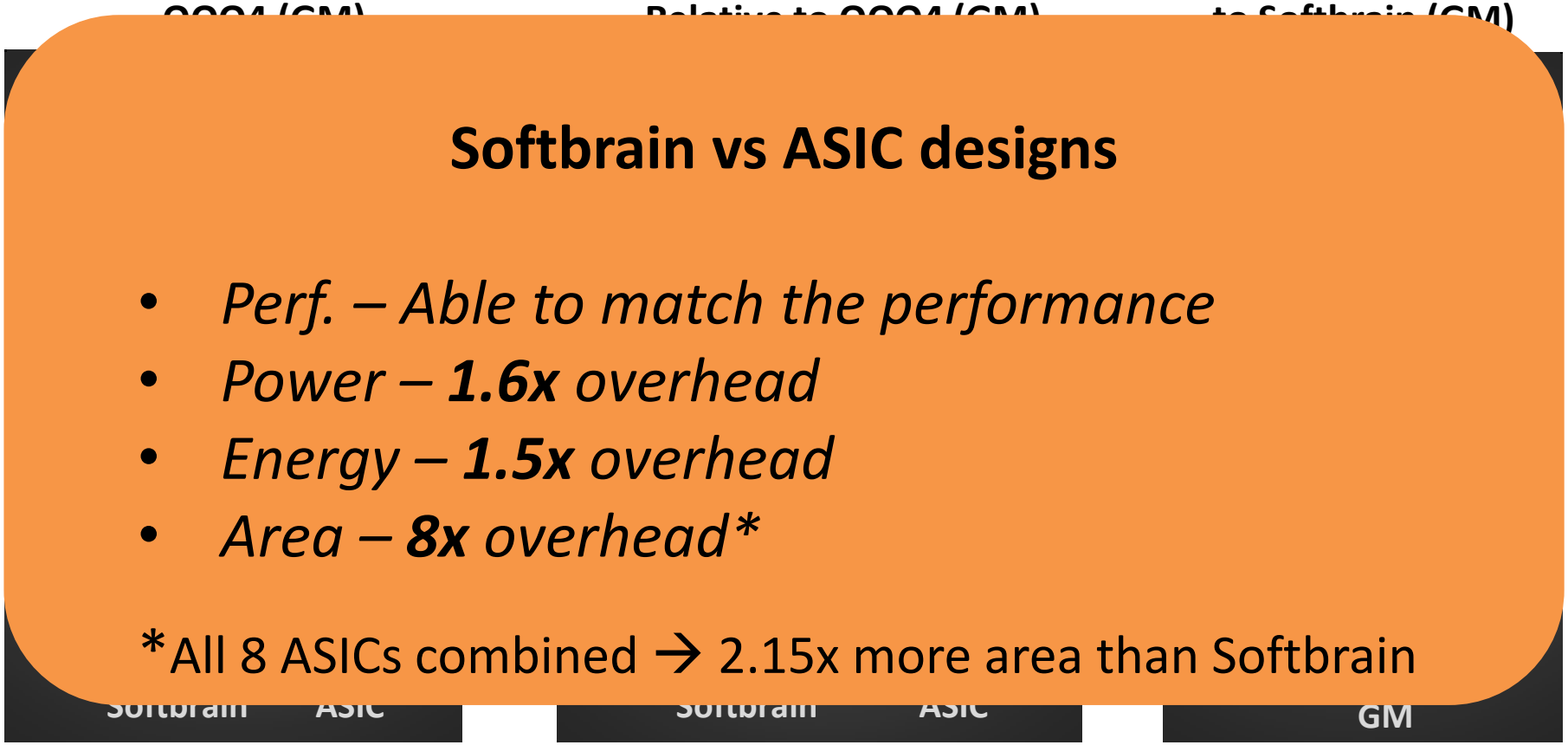
Broadly Provisioned Softbrain vs ASIC Performance Comparison



Aladdin* generated ASIC design points – Resources constrained to be in ~15% of Softbrain Perf. to do iso-performance analysis

*Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. Sophia Shao , .et. al

Broadly Provisioned Softbrain vs ASIC Area & Power Comparison



Softbrain vs ASIC designs

- *Perf. – Able to match the performance*
- *Power – **1.6x** overhead*
- *Energy – **1.5x** overhead*
- *Area – **8x** overhead**

*All 8 ASICs combined → 2.15x more area than Softbrain

Conclusion – Stream-Dataflow Acceleration

- Stream-Dataflow Acceleration
  - Stream-Dataflow **Execution Model** – Abstracts typical accelerator computation phases using a dataflow graph
  - Stream-Dataflow **ISA Encoding** and **Hardware-Software Interface** – Exposes parallelism available in these phases
- Stream-Dataflow Accelerator Architecture
  - CGRA and vector ports for pipelined vector-dataflow computation
  - Highly parallel stream-engines for low-power stream communication
- Stream-Dataflow Prototype & Implementation – Softbrain
  - Matches performance of domain provisioned accelerator (DianNao DSA) with **~2x** overheads in area and power
  - Compared to application specific designs (ASICs), Softbrain has **~2x** overheads in power and **~8x** in area

Dissertation Research Goal

Programmable Hardware Acceleration

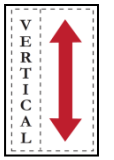
1. Explore the commonality in the way the DSAs specialize – ***Specialization Principles*** ✓

2. **General Mechanisms** for the design of a generic programmable hardware accelerator matching the efficiency of DSAs ✓

3. A programmable/re-configurable accelerator architecture with an efficient **accelerator hardware-software (ISA) interface** ✓

4. **Easy adaptation** of new acceleratable algorithms in a domain-agnostic way ✓

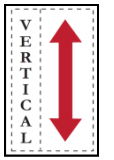
Conclusion – *Programmable Hardware Acceleration*



- New acceleration paradigm in specialization era
  - Programmable Hardware Acceleration breaking the limits of acceleration
- F n
- d,
- **Getting There !!**

  A good enabler for exploring general purpose programmable hardware acceleration ….
- Reduce the orders of magnitude overheads of programmability and generality compared to ASICs
- Drives future accelerator research and innovation

Future Work



- Multiple DFG executions
  - Configuration cache for CGRA to switch between DFGs
- Further distribute the control into vector ports
  - Dynamic deadlock detection for buffer overflow
  - Concurrent execution of different set of streams (of different DFGs)
- Low-power dynamic credit-based CGRA schedule
  - Allow vector ports to run out-of-order reducing the overall latency
- 3D support for streams in ISA
- Partitioned scratchpad to support data dependent address generation
- Support for fine-grained configuration through FPGA slices (along with SRAM mats) next to CGRA for memory-dependent algorithm acceleration

Related Work

- Programmable specialization architectures:
  - *Smart memories, Charm, Camel, Mosphosys, XLOOPS, Maven-VT*
- Principles of Specialization
  - GPPs inefficient and need specialization – *Hameed. et. Al*
  - Trace processing – *Beret*
  - Transparent Specialization – *CCA, CRIB etc,*
- Heterogeneous Cores – GPP + Specialized engines
  - *Composite cores, DySER, Cambricon*
- Streaming Engines:
  - *RSVP arch, Imagine, Triggered instructions, MAD, CoRAM++*

Other Works

- Open Source GPGPU – MIAOW
  - Lead developer and contributor to open source hardware GPGPU – MIAOW
  - AMD Southern Island based RTL implementation of GPGPU able to execute unmodified AMDAPP OpenCL kernels
  - Published in [***ACM TACO 2015, HOTCHIPS' 2015, COOLCHIPS' 2015, HiPEAC' 2016***]
- Von-Neumann/Dataflow Hybrid Architecture
  - A hybrid architecture aimed to exploit ILP in irregular applications
  - Lead developer of the micro-architecture of the dataflow offload engine – Specialized Engine for Explicit Dataflow (SEED)
  - Published in [***ISCA' 2015, IEEE MICRO Top Picks 2016***]
- Open-source Hardware: Opportunities and Challenges
  - A position article on the advantages of open-source hardware for hardware innovation
  - Huge believer in open-source hardware and contribution
  - To be published in ***IEEE Computer' 17***

Back Up

Programmable Hardware Acceleration

Idea 1: Specialization principles can be exploited in a ***general way***

Idea 2: Composition of known ***Micro-Architectural mechanisms*** embodying the specialization principles

Programmable Hardware Accelerator (GenAccel)

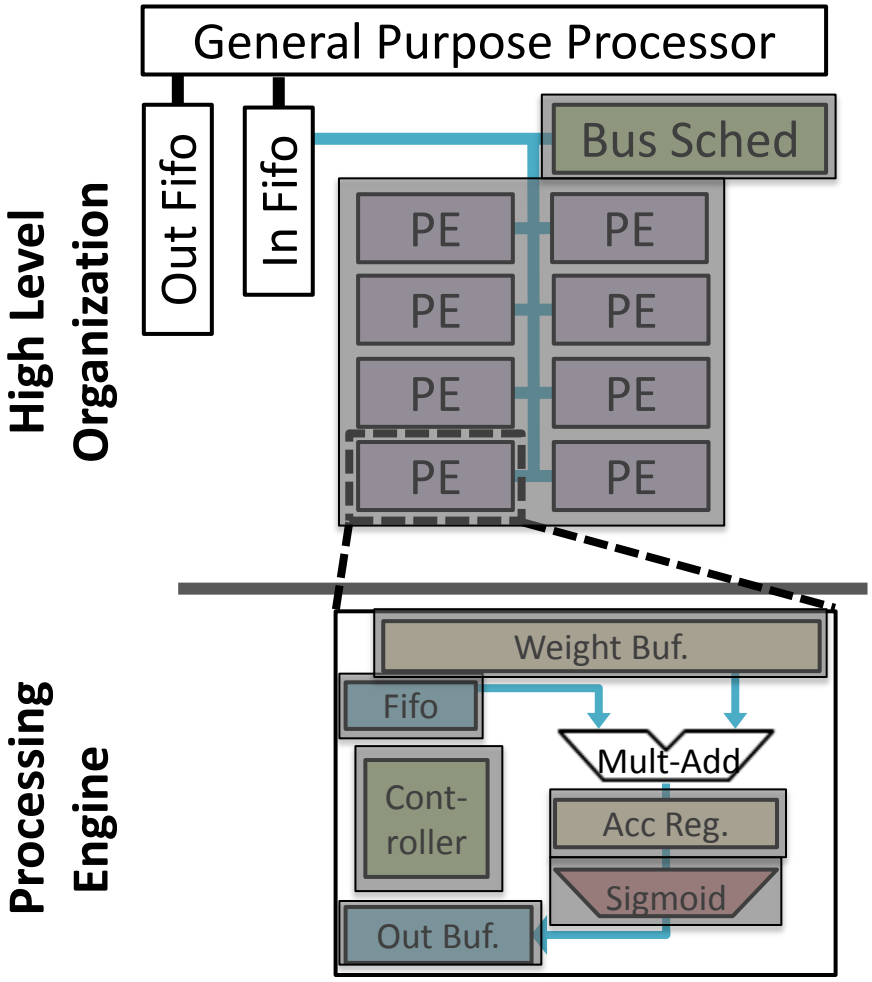
GenAccel as a programmable hardware design template to map **one** or **many** application domains

Domain provisioned GenAccel **Balanced GenAccel**

*Figures not to scale

Principles in DSAs

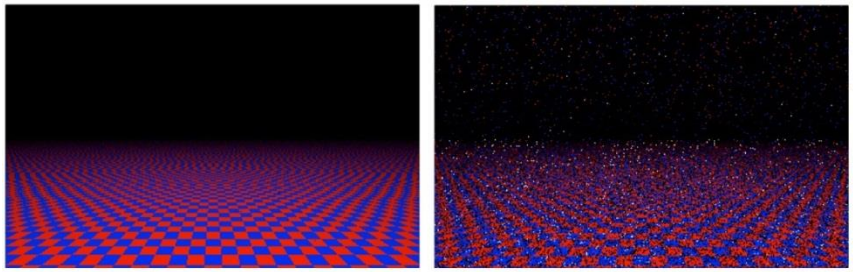
NPU – Neural Proc. Unit



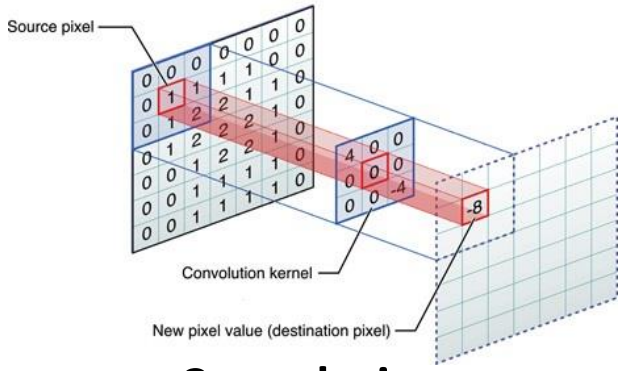
- Match hardware **concurrency** to that of algorithm
- Problem-specific **computation** units
- Explicit **communication** as opposed to implicit communication
- Customized structures for **data reuse**
- Hardware **coordination** using simple low-power control logic

Concurrency | Computation | Communication | Data Reuse | Coordination

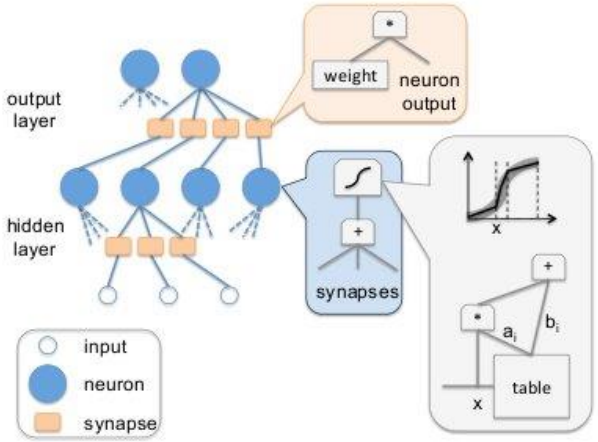
Accelerator Workloads



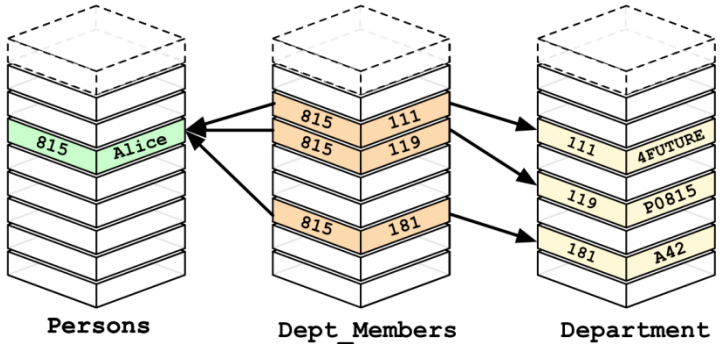
Neural Approx.



Convolution



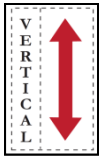
DNN



Database Streaming

1. Ample Parallelism
2. Regular Memory
3. Large Datapath
4. Computation Heavy

GenAccel Modeling Strategy



- Phase 1. Model Single-Core with PIN + Gem5 based trace simulation
  - The algorithm to specialize in the form of c-code/binary
  - Potential Core Types, CGRA sizes, any specialized instructions
  - Degree of memory customization (which memory accesses to be specialized, either with DMA or scratchpad)
  - Output: single-core perf./energy for “Pareto-optimal” designs

- Phase 2. Model coarse-grained parallelism
  - Use profiling information to determine parallel portion of the algorithm (or tell user to indicate or estimate)
  - Use simple Amdahl's law to get performance estimate
  - Use execution time, single-core energy estimate, and static power estimate to get overall energy estimate

GenAccel in Practice

1. Design Synthesis

Hardware Architect/Designer

Performance Requirements

	Perf.
App. 1:	...
App. 2:	...
App. 3:	...

Hardware Constraints

Area goal:	...
Power goal:	...

- ✓ **FU Types**
- ✓ **No. of FUs**
- ✓ **Spatial fabric size**
- ✓ **No. of GenAccel tiles**

Design decisions

Synthesis

Programmable Accelerator (GenAccel)

2. Programming

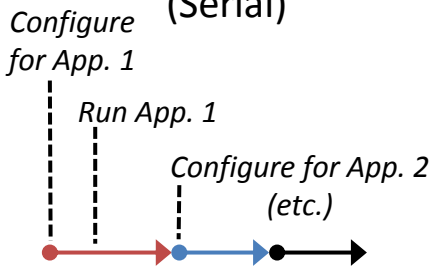
```
#pragma genaccel cores 2
#pragma reuse-scratchpad weights
void nn_layer(int num_in, int num_out,
              const float* weights,
              const float* in, float* out)
{
  for (int j = 0; j < num_out; ++j) {
    for (int i = 0; i < num_in; ++i) {
      out[j] += weights[j][i] *in[i];
    }
    out[j] = sigmoid(out[j]);
  }
}
```

For each application:

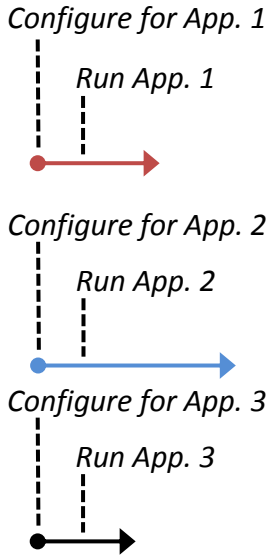
- ✓ **Write Control Program** (C Program + Annotations)
- ✓ **Write Datapath Program** (spatial scheduling)

3. Runtime

Runtime configuration (Serial)



Runtime configuration (Parallel)

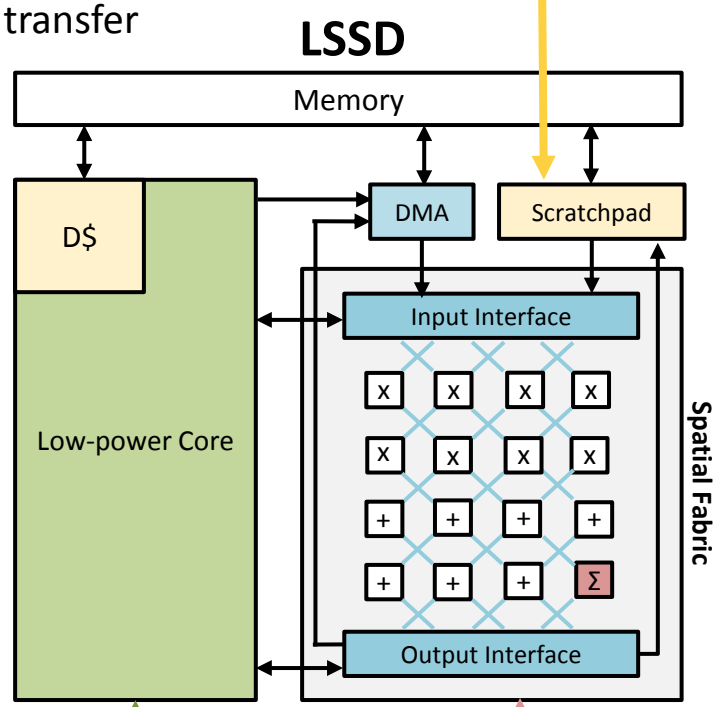


Programming GenAccel

Pragmas

```
#pragma genaccel cores 2
#pragma reuse-scratchpad weights

void nn_layer(int num_in, int num_out,
              const float* weights,
              const float* in,
              const float* out )
{
    for (int j = 0; j < num_out; ++j)
    {
        for (int i = 0; i < num_in; ++i)
        {
            out[j] += weights[j][i] *in[i];
        }
        out[j] = sigmoid(out[j]);
    }
}
```

Insert data transfer



Loop Parallelize, Insert Communication, Modulo Schedule

Resize Computation (Unroll), Extract Computation Subgraph, Spatial Schedule

GenAccel Design Point Selection

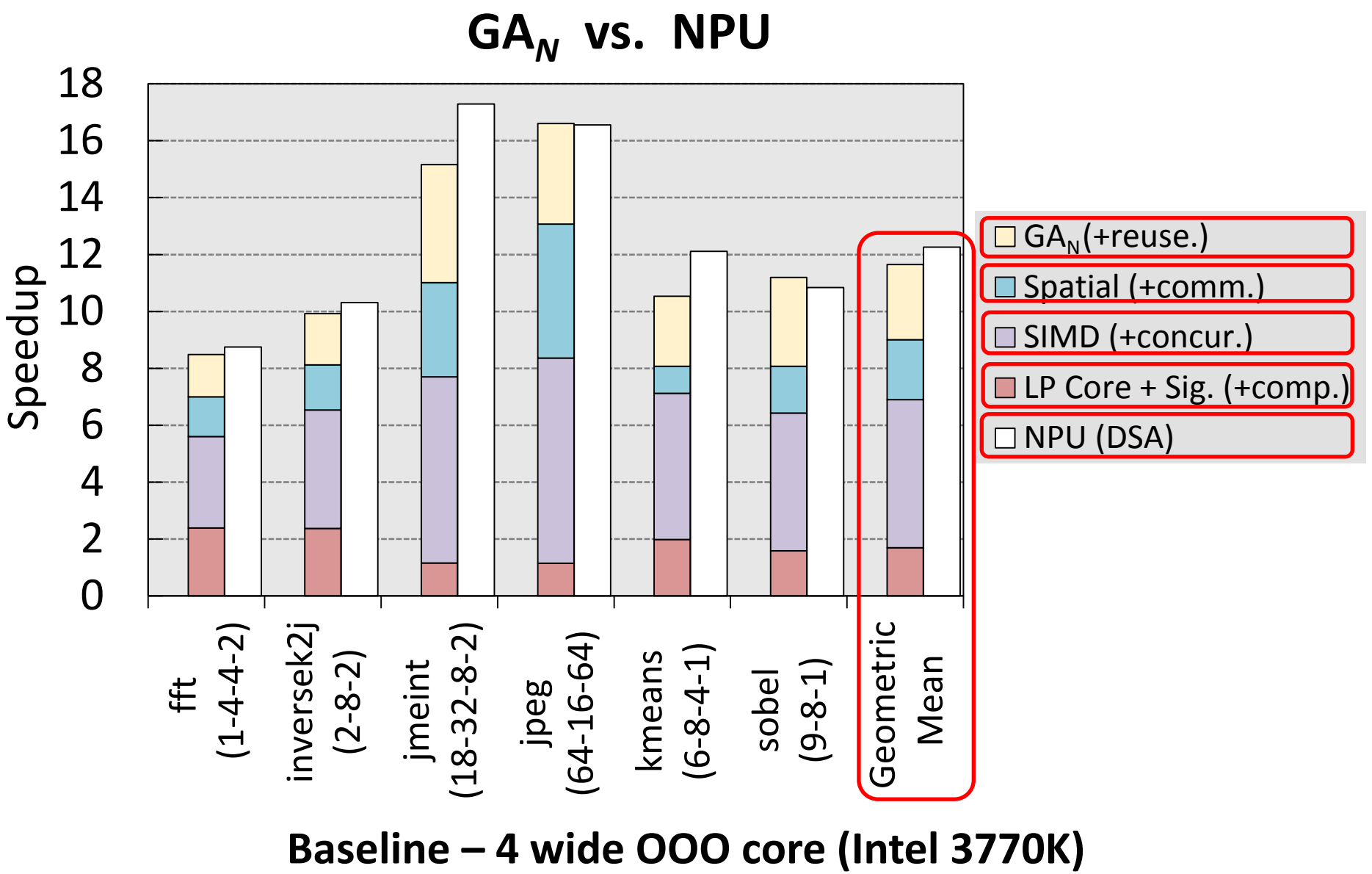
Design	Concurrency	Computation	Communication	Data Reuse	No. of GenAccel Units
GA_N	24-tile CGRA (8 Mul, 8 Add, 1 Sigmoid)	2k x 32b sigmoid lookup table	32b CGRA; 256b SRAM interface	2k x 32b weight buffer	1
GA_C	64-tile CGRA (32 Mul/Shift, 32 Add/logic)	Standard 16b FUs	16b CGRA; 512b SRAM interface	512 x 16b SRAM for inputs	1
GA_D	64-tile CGRA (32 Mul, 32 Add, 2 Sigmoid)	Piecewise linear sigmoid unit	32b CGRA; 512b SRAM interface	2k x 16b SRAMs for inputs	8
GA_Q	32-tile CGRA (16 ALU, 4 Agg, 4 Join)	Join + Filter units	64b CGRA; 256b SRAM interface	SRAMs for buffering	4
GA_B	32-tile CGRA (Combination of above)	Combination of above FUs	64b CGRA; 512b SRAM interface	4KB SRAM	8

Mul: Multiplier, Add: Adder

Design-Time vs. Runtime Decisions

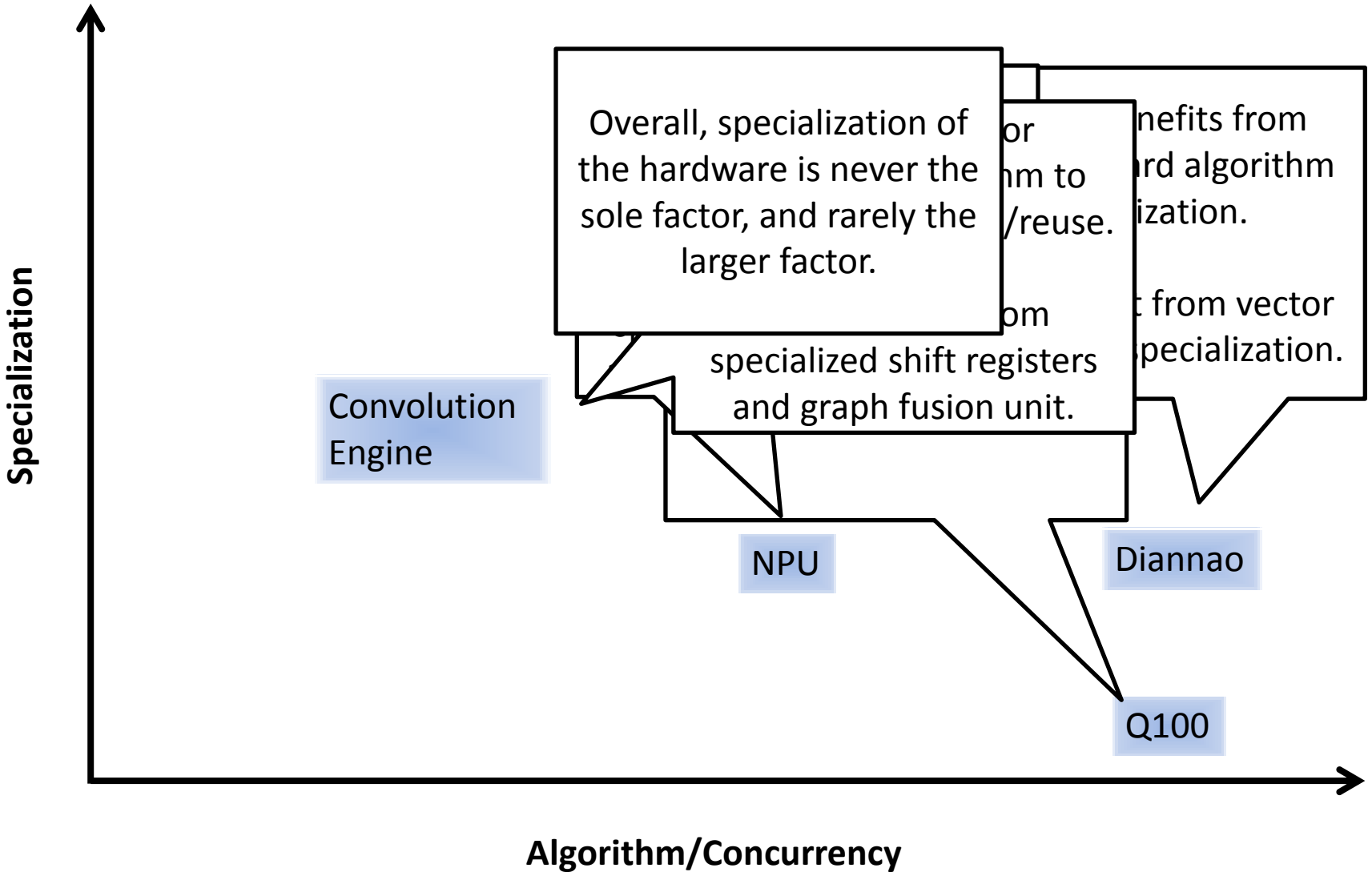
	Synthesis – Time	Run – Time
Concurrency	No. of GenAccel Units	Power-gating unused GenAccel Units
Computation	Spatial fabric FU mix	Scheduling of spatial fabric and core
Communication	Enabling spatial datapath elements, & SRAM interface widths	Configuration of spatial datapath, switches and ports, memory access pattern
Data Reuse	Scratchpad (SRAM) size	Scratchpad used as DMA/reuse buffer

Performance Analysis (1)

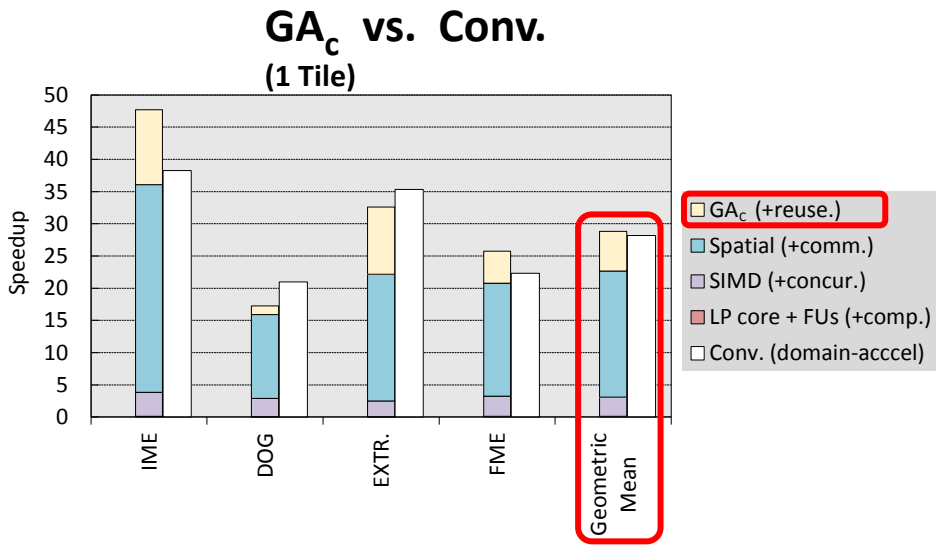


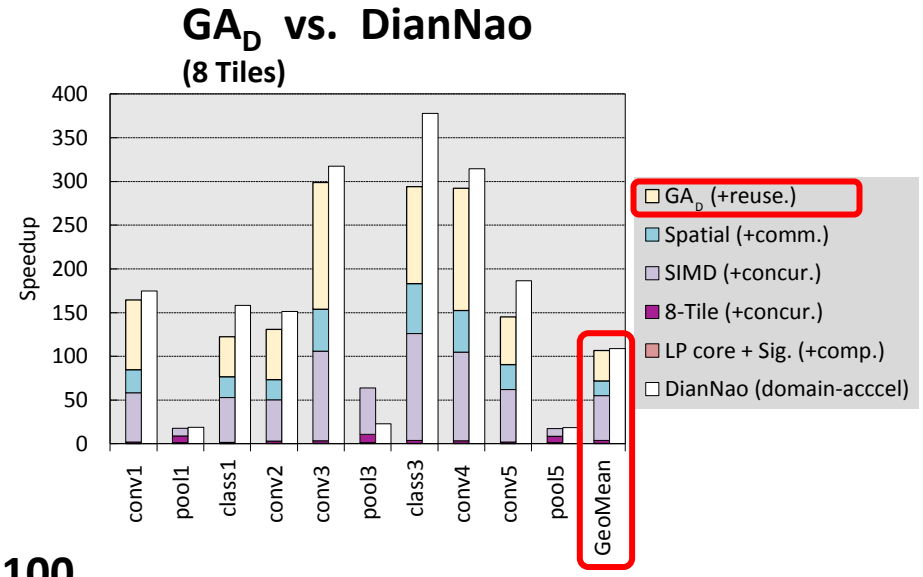
Baseline – 4 wide OOO core (Intel 3770K)

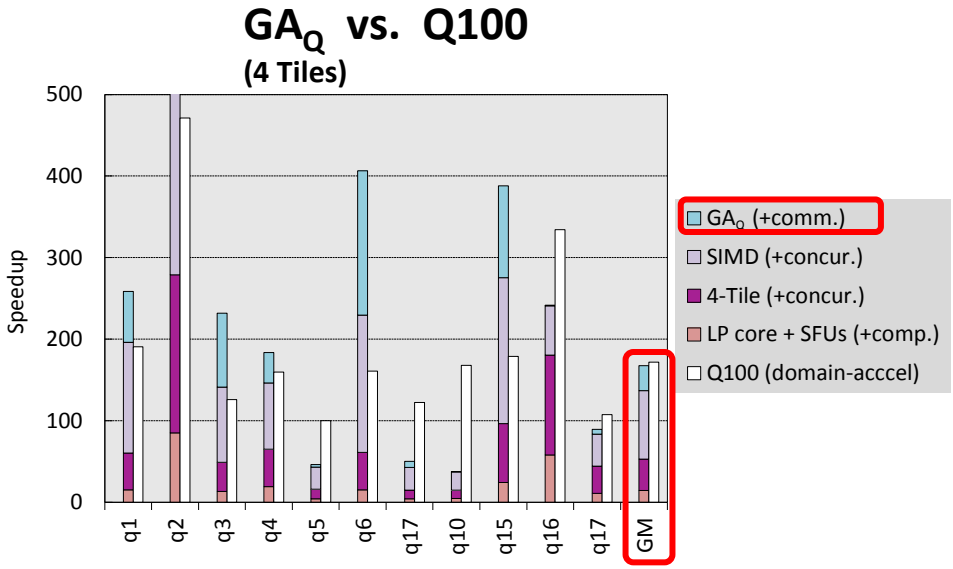
Source of Accelertion Benefits



Performance Analysis (2)







Baseline – 4 wide OOO core (Intel 3770K)

GenAccel Area & Power Numbers

		Area (mm^2)	Power (mW)
Neural Approx.	GA_N	0.37	149
	NPU	0.30	74
Stencil	GA_C	0.15	108
	Conv. Engine	0.08	30
Deep Neural.	GA_D	2.11	867
	DianNao	0.56	213
Database Streaming	GA_Q	1.78	519
	Q100	3.69	870
	$GA_{Balanaced}$	**2.74**	**352**

*Intel Ivybridge 3770K CPU 1 core Area – **12.9mm²** | Power – **4.95W**

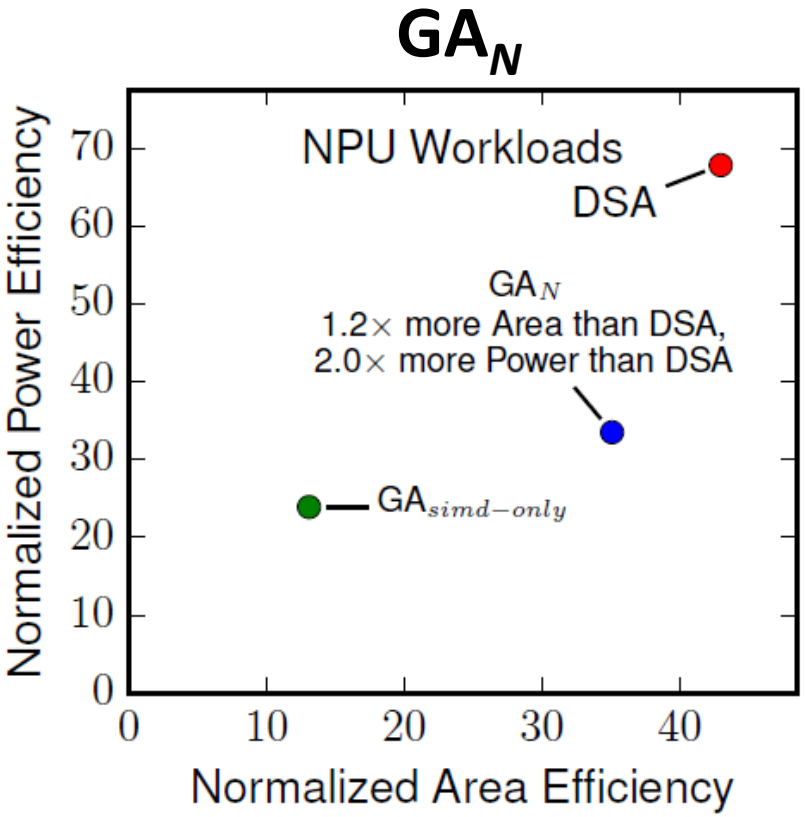
*Intel Ivybridge 3770K iGPU 1 execution lane Area – **5.75mm²**

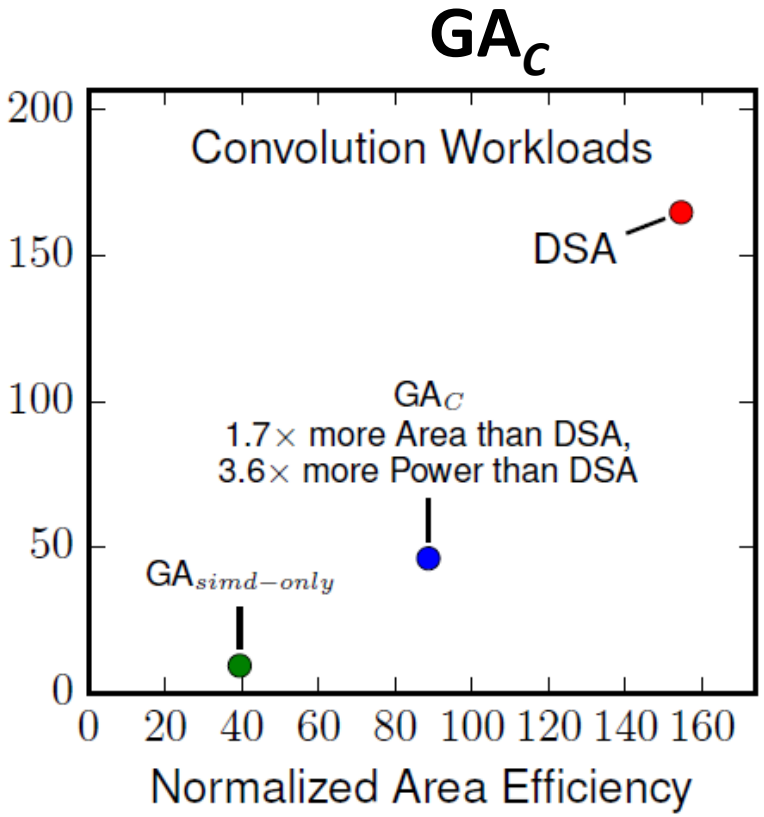
+AMD Kaveri APU Tahiti based GPU 1CU Area – **5.02mm²**

*Source: http://www.anandtech.com/show/5771/the-intel-ivy-bridge-core-i7-3770k-review/3

+Estimate from die-photo analysis and block diagrams from wccftech.com

Power & Area Analysis (1)

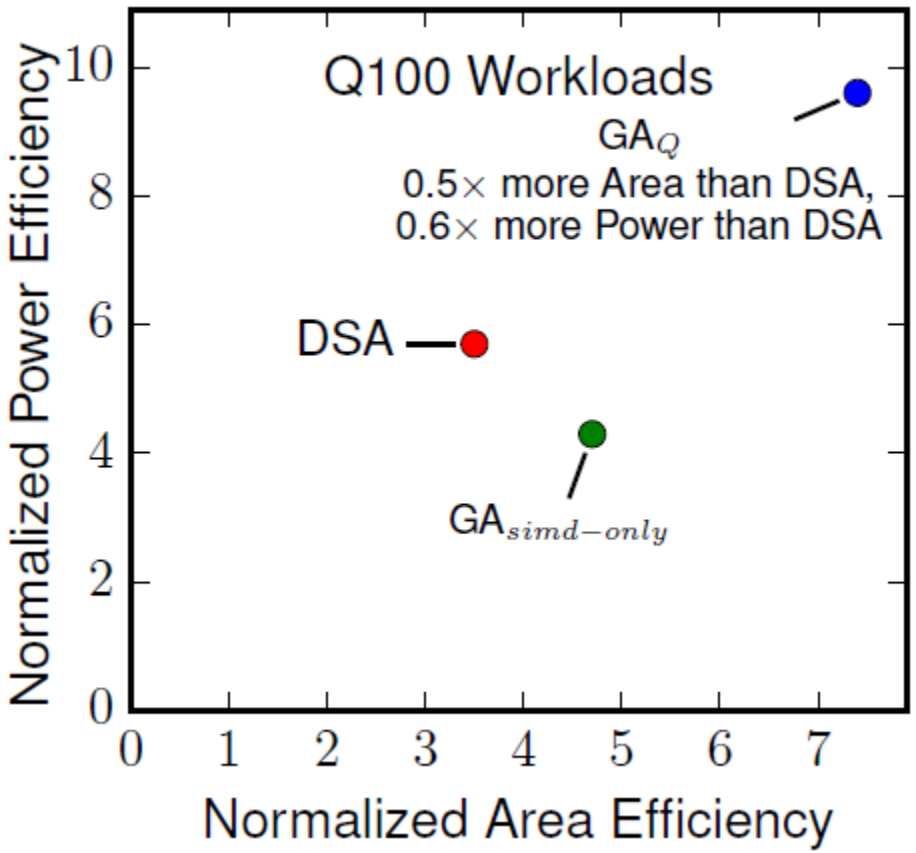


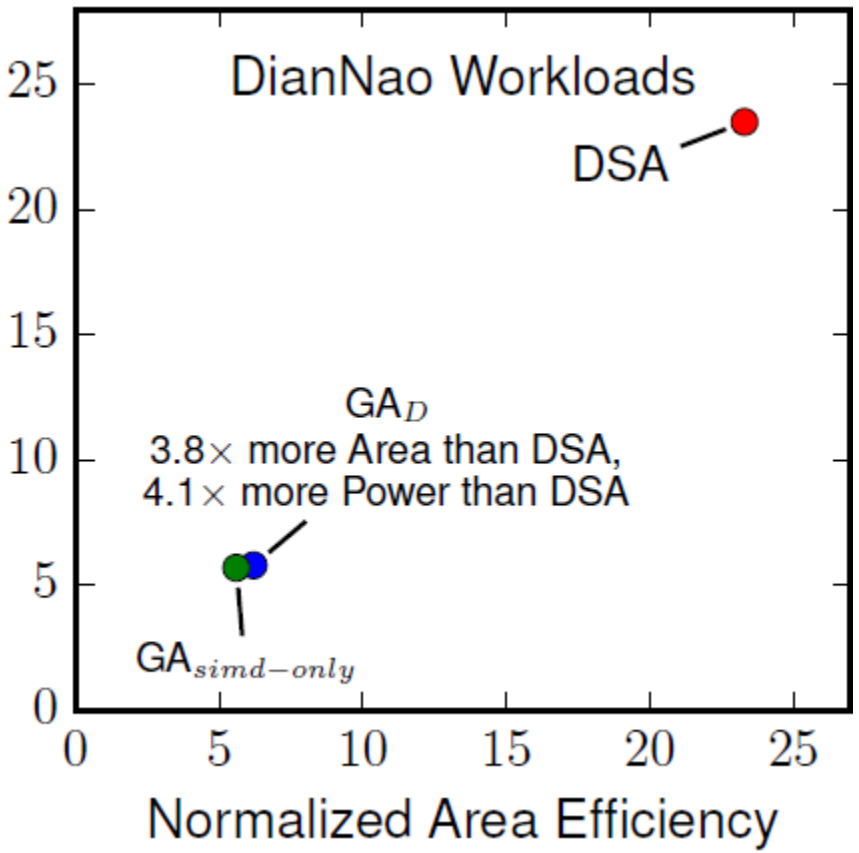


1.2x more Area than DSA
2x more Power than DSA

1.7x more Area than DSA
3.6x more Power than DSA

Power & Area Analysis (2)



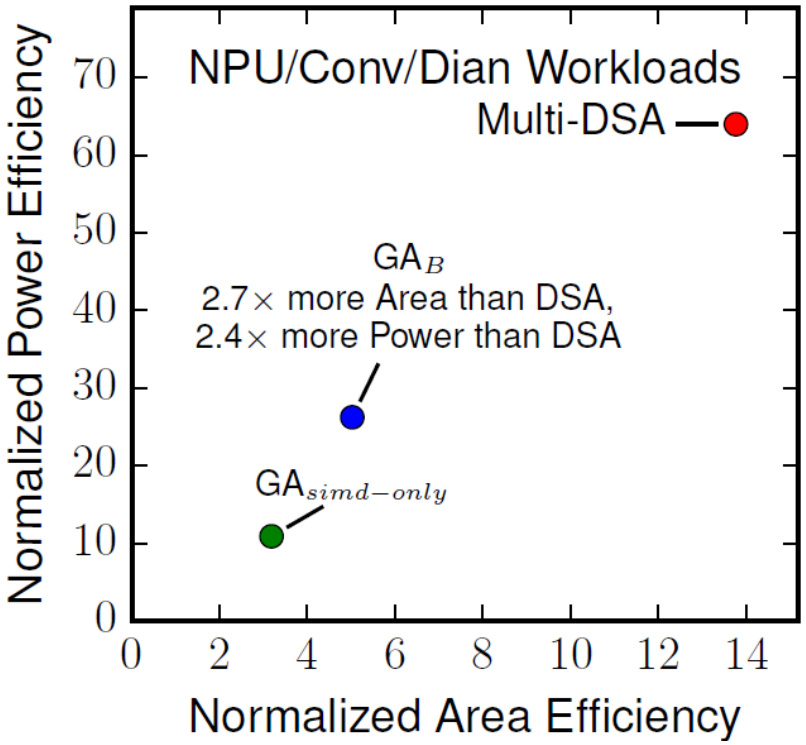


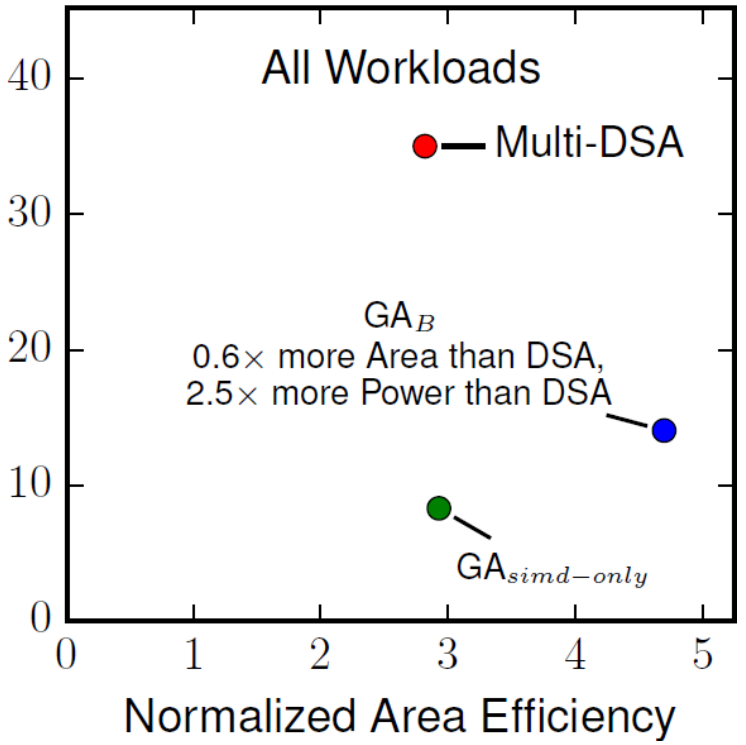
0.5x more Area than DSA
0.6x more Power than DSA

3.8x more Area than DSA
4.1x more Power than DSA

Power & Area Analysis (3)

$LSSD_B$ → Balanced LSSD design





2.7x more Area than DSAs
2.4x more Power than DSAs

0.6x more Area than DSA
2.5x more Power than DSA

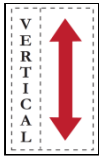
Unsuitable Workloads for GenAccel /Stream-Dataflow

- Memory-dominated workloads
- Specifically small-memory footprint, but “irregular”
- Heavily serialized data dependent address generation
- Memory compression for example
  - A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs, Fower et. al
- Other examples:
  - IBM PowerEN Regular Expression
  - DFA based codes

GenAccel vs. FPGA

Domain Kernel	Number of States
NPU	17
Convolution Engine	14
Diannao	86

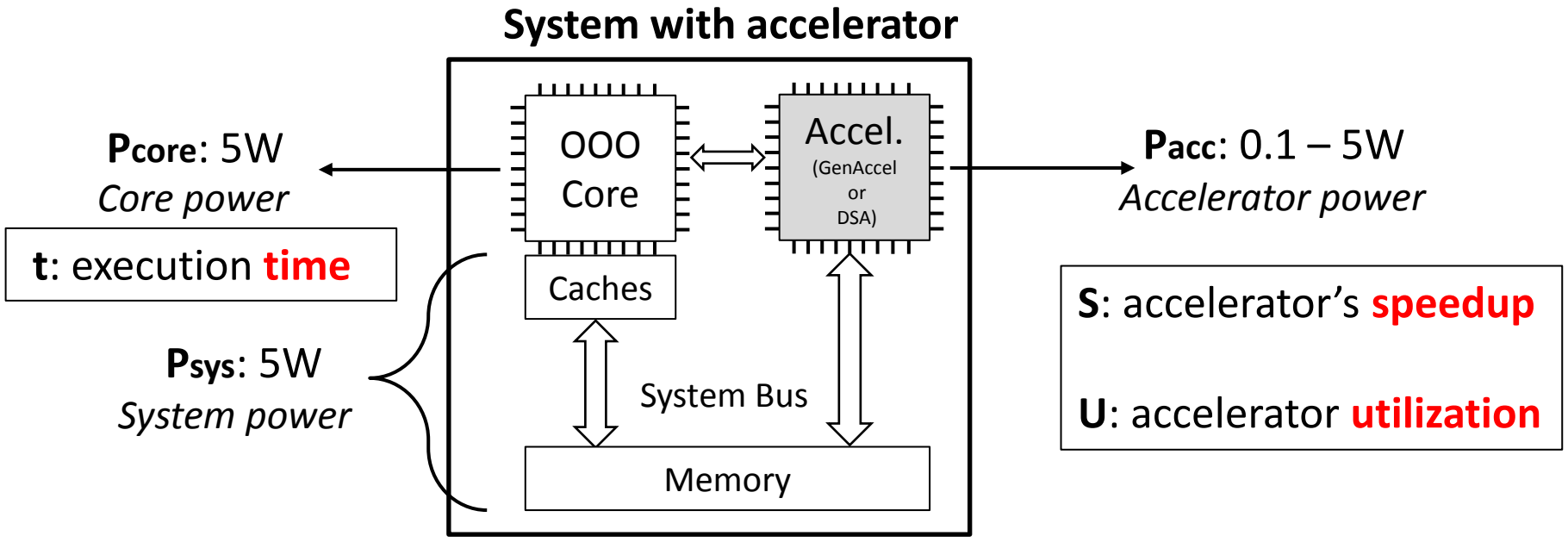
- FPGAs are much lower frequency (global-routing and too fine-grained)
- BlockRAMs too small to gang-up
- Logical Multi-ported Register File needed to pass values between DSP slices to match high operand-level concurrency
- Altera's Stratix 10 seems headed exactly this direction



GenAccel's power overhead of
2x - 4x matter in a system with accelerator?

In what scenarios you want to build
DSA over GenAccel?

Energy Efficiency Tradeoffs



Overall **energy** of the computation executed on system

$$E = P_{acc} * (U/S) * t + P_{sys} * (1 - U + U/S) * t + P_{core} * (1 - U) * t$$

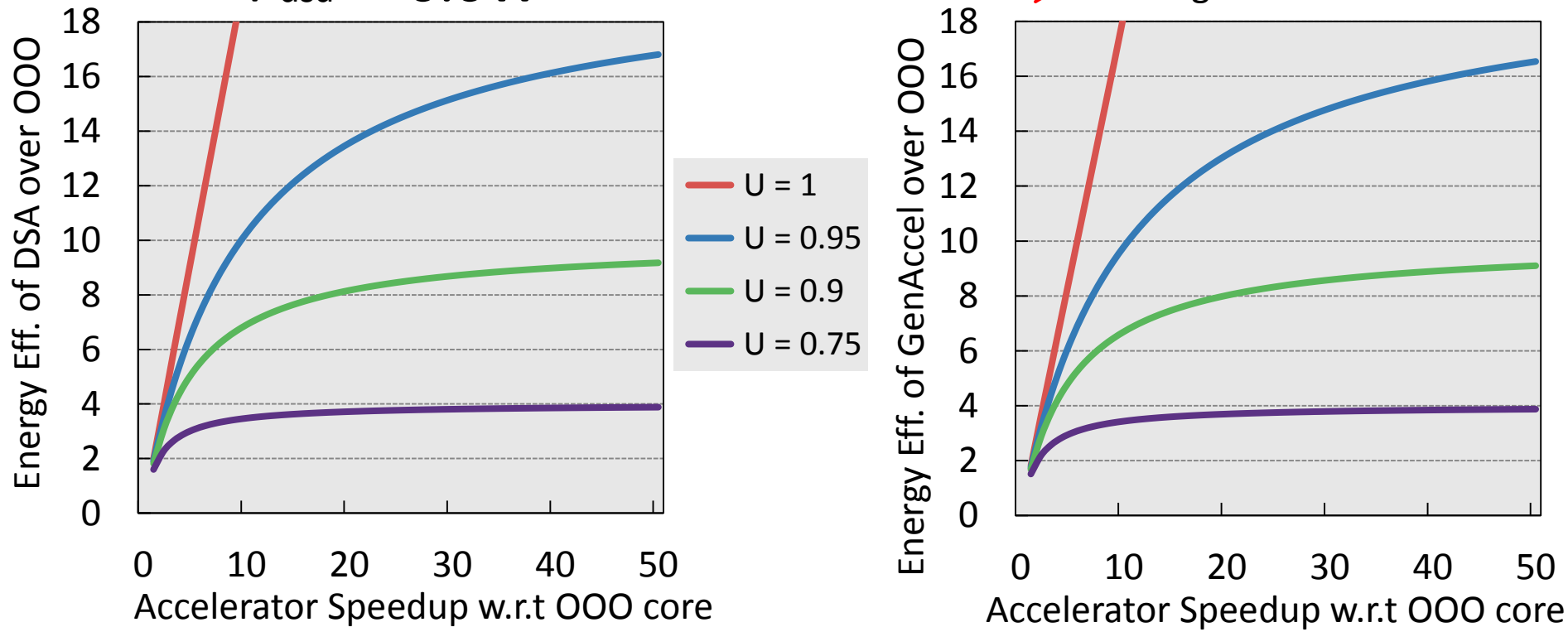
Accel. energy | System energy | Core energy

*Power numbers are example representation

Energy Efficiency Gains of GenAccel & DSA over OOO core

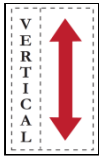
$Speedup_{ga} = Speedup_{dsa}$ (Speedup w.r.t OOO)

$P_{dsa} \approx 0.0W$ **500mW (5x)Power overhead** $P_{ga} = 0.5W$



Baseline – 4 wide OOO core

Efficiency gains of both GenAccel and DSA are almost similar &
At higher speedups both get "capped" due to large system power

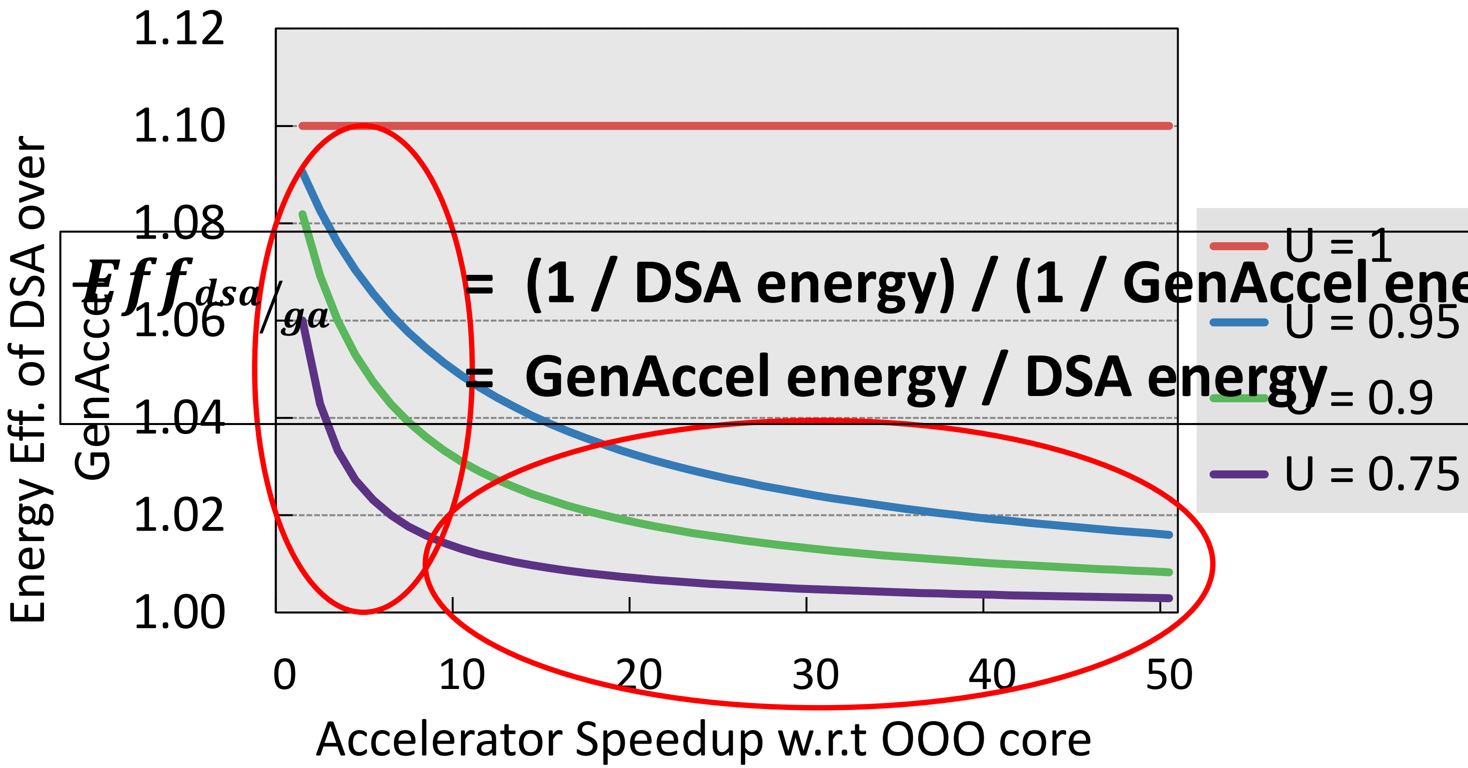


GenAccel's power overhead of
2x - 4x matter in a system with accelerator?

When $\mathbf{P_{sys} >> P_{ga}}$, 2x - 4x power overheads of GenAccel become inconsequential

Energy Efficiency Gains of DSA over GenAccel

Speedup$_{ga}$ = Speedup$_{dsa}$ (Speedup w.r.t OOO)

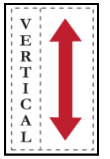


$Eff_{dsa/ga}$ = (1 / DSA energy) / (1 / GenAccel energy)

= GenAccel energy / DSA energy

Baseline – GenAccel

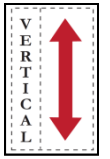
$Eff_{dsa/ga}$ is no more than 10% even at 100% utilization



In what scenarios you want to build DSA over GenAccel?

Only when application speedups are small & small energy efficiency gains too important

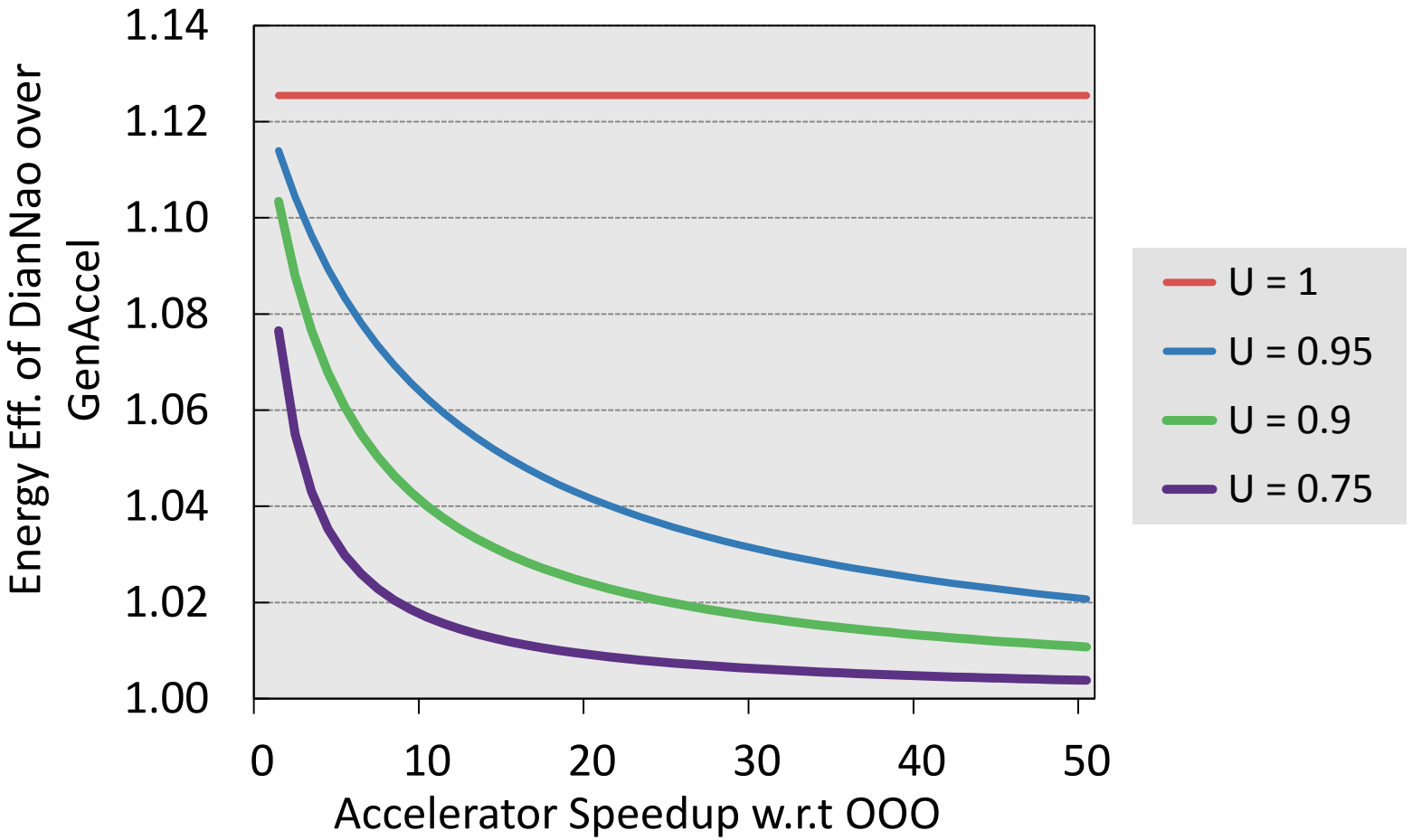
When does accelerator power or DSA matter?



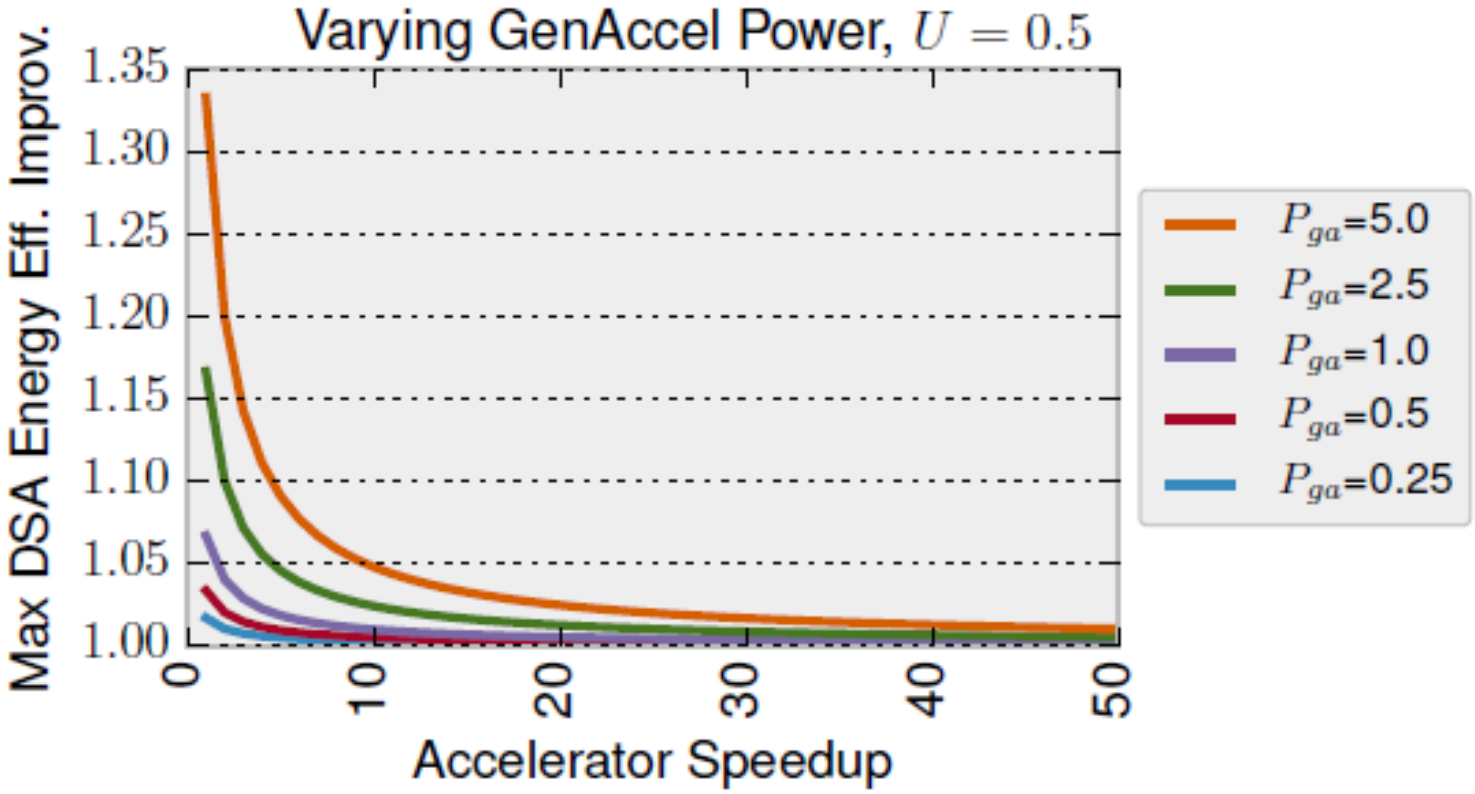
- GenAccel cannot match DSA for performance
- Accelerator is a "vertically-integrated" accelerator
  - Logic attached to memory or IO, that P_{sys} is affected
  - ShiDianNao for example (DNN attached to image sensor)
- Speedups are "small" and 10% energy difference is "valuable"

Energy Efficiency Gains of DianNao over GenAccel

$Speedup_{GA} = Speedup_{DianNao}$ (Speedup w.r.t OOO)



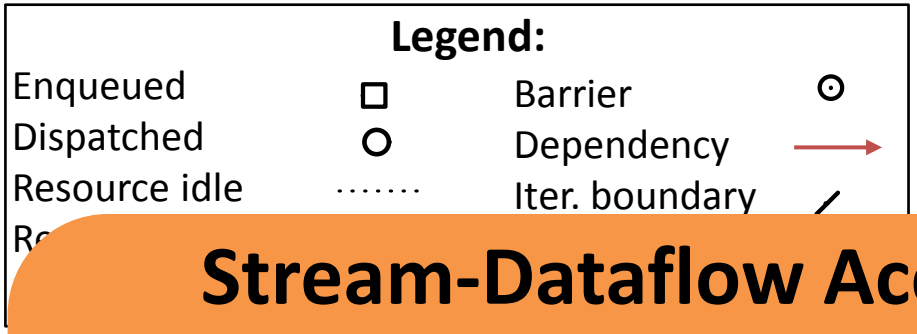
Does Accelerator power matter?



- At Speedups > 10x, DSA eff. is around 5%, when accelerator power == core power

- At smaller speedups, makes a bigger difference, up to 35%

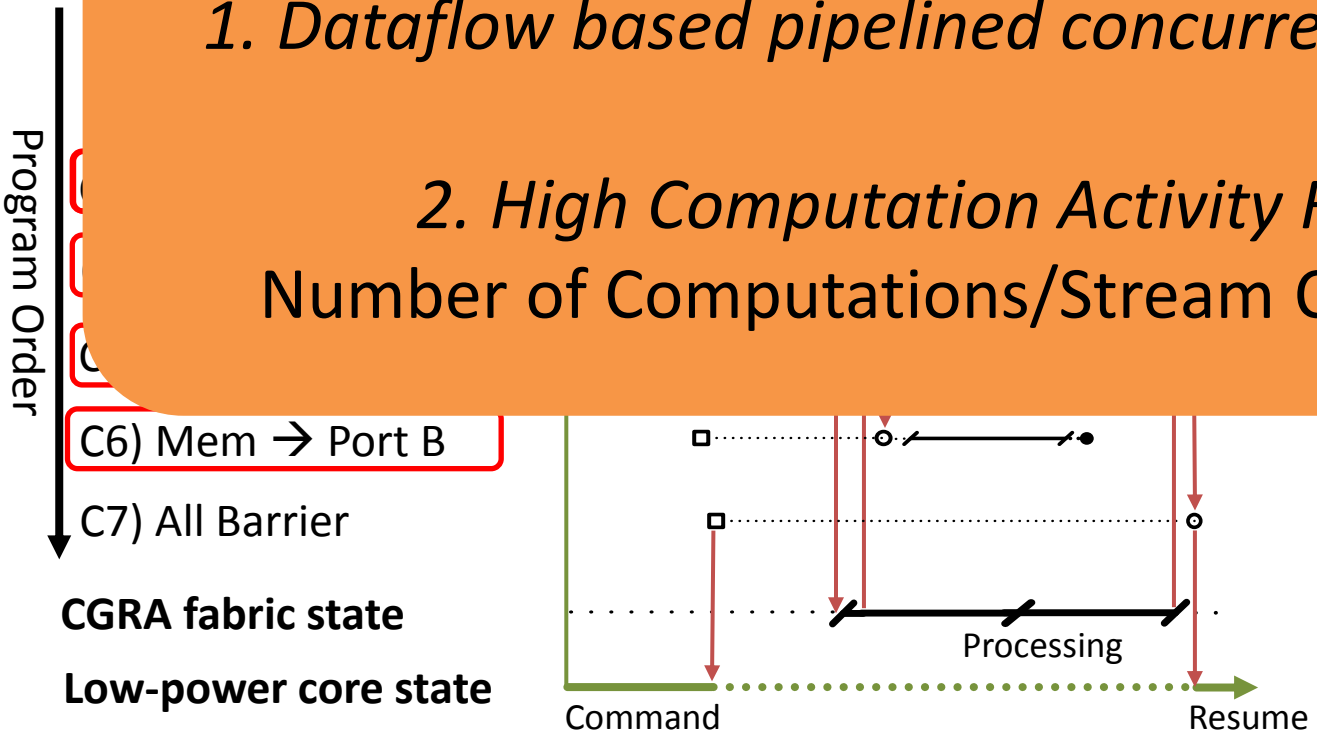
Detailed Example of Stream-Dataflow Execution Model



Stream-Dataflow Accelerator Potential

1. Dataflow based pipelined concurrent execution

2. High Computation Activity Ratio:
Number of Computations/Stream Commands

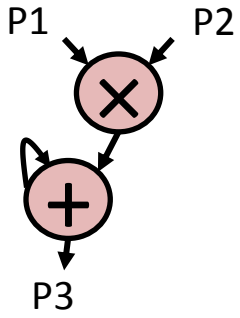


Example Code: Dot Product (Instruction Comparisons)

Original Program

```
for(int i = 0 to N) {
  dot_prod += a[i] * b[i]
}
```

Computation Graph:



Scalar

```
for(i = 0 to N) {
  Send a[i] -> P1
  Send b[i] -> P2
}
Get P3 -> result
```

~2N Instructions

Vector

```
for(i = 0 to N, i+=vec_len) {
  Send a[i:i+vec_len] -> P1
  Send b[i:i+vec_len] -> P2
}
Get P3 -> result
```

~2N/vec_len Instructions

Stream-Dataflow

```
Send a[i:i+N] -> P1
Send b[i:i+N] -> P2
Get P3 -> result
```

~3 Instructions

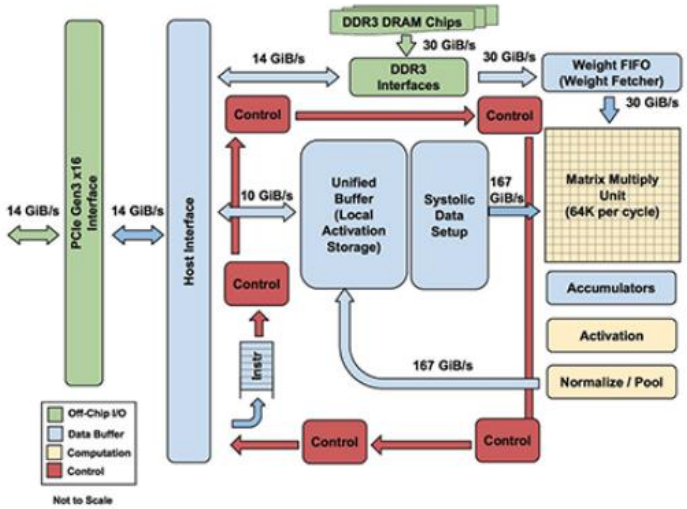
Stream-Dataflow ISA vs. TPU ISA

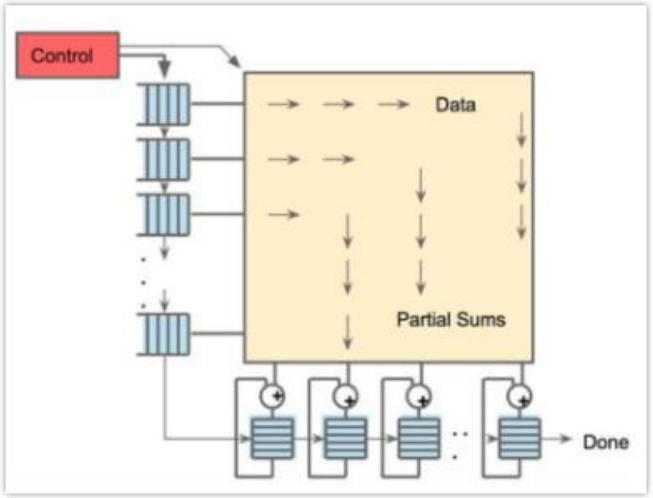
Google TPU ISA

- Design goal of TPU ISA
  - To be a programmable ISA with less instruction overheads
- Restricted to neural networks domain only → More of programmable ISA for NN domain
- CISC principle to run complex tasks → To run fast multiple-add accumulations
- Uses matrix as a primitive instead of vector or scalar
  - Huge performance benefit for neural network applications
  - Reduced latency for inference [< 7ms]
  - ISA restricted heavily for certain type of computations

    [*Read_Host_Memory, Read_Weights, MatrxMultiply/Convolve, Activate, Write_Host_Memory*]
- Heavily relies on host processor to send the instructions. Host software will be a bottleneck
- Does not decouple the memory and computation phases

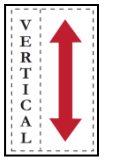
TPU Compute Capability





- 700 Mhz target frequency with 40W TDP. External accelerator and PCIe based interconnect to host – 12.5GB/s effective bandwidth
- An inference chip for MLPs, CNN and LSTM → ***Matrix-Matrix*** multiplication support – 65K operations per cycle using a 256 x 256 systolic array 2D pipeline
- Quantization helps performance to operate on 8-bit integers only

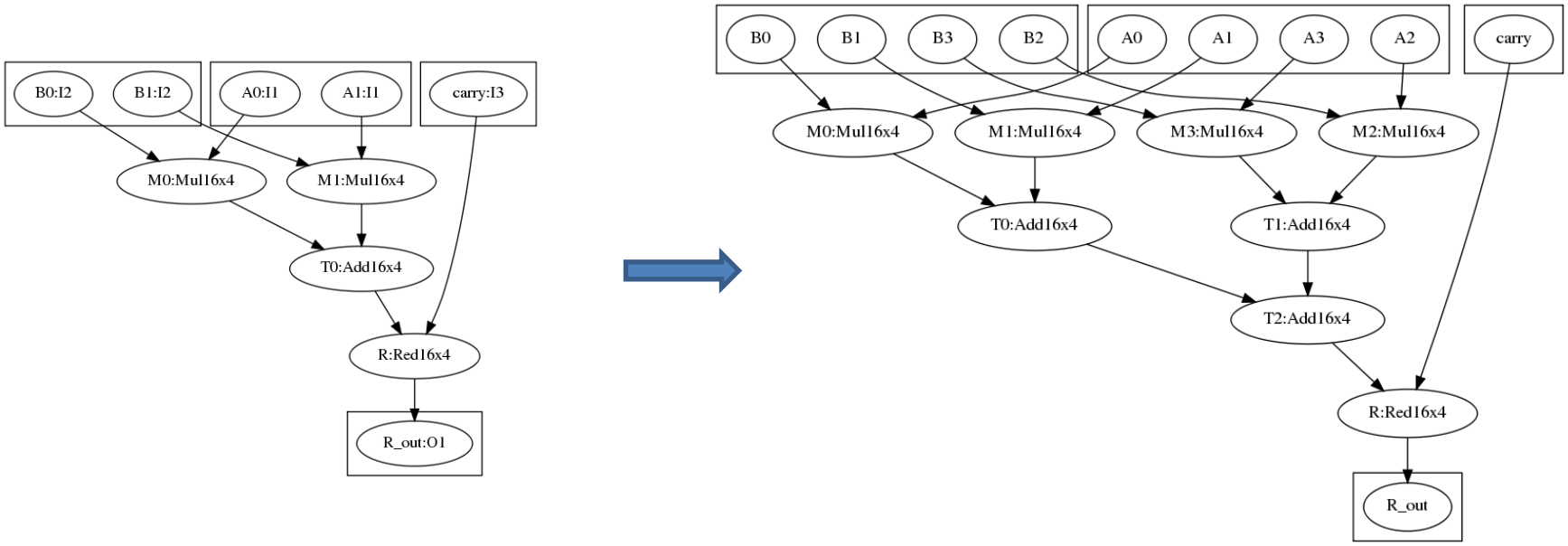
Potential Performance Bottlenecks



1. Computations Per CGRA Instance
2. General Core Instructions
3. Cache → GRA Bandwidth
4. Initialization/Draining Latency (Memory & CGRA)
5. Length of Recurrence through CGRA

1. Computations Per CGRA Instance

Principle: Few instructions control many computation instances



HINT: This usually involves unrolling a loop – but not necessarily the inner loop.

2. General Core Instructions

- ***Principle:*** Few core instructions control many computation instances
  - Use as **long** streams as possible
  - Computation Instances > 2 * Number of Commands

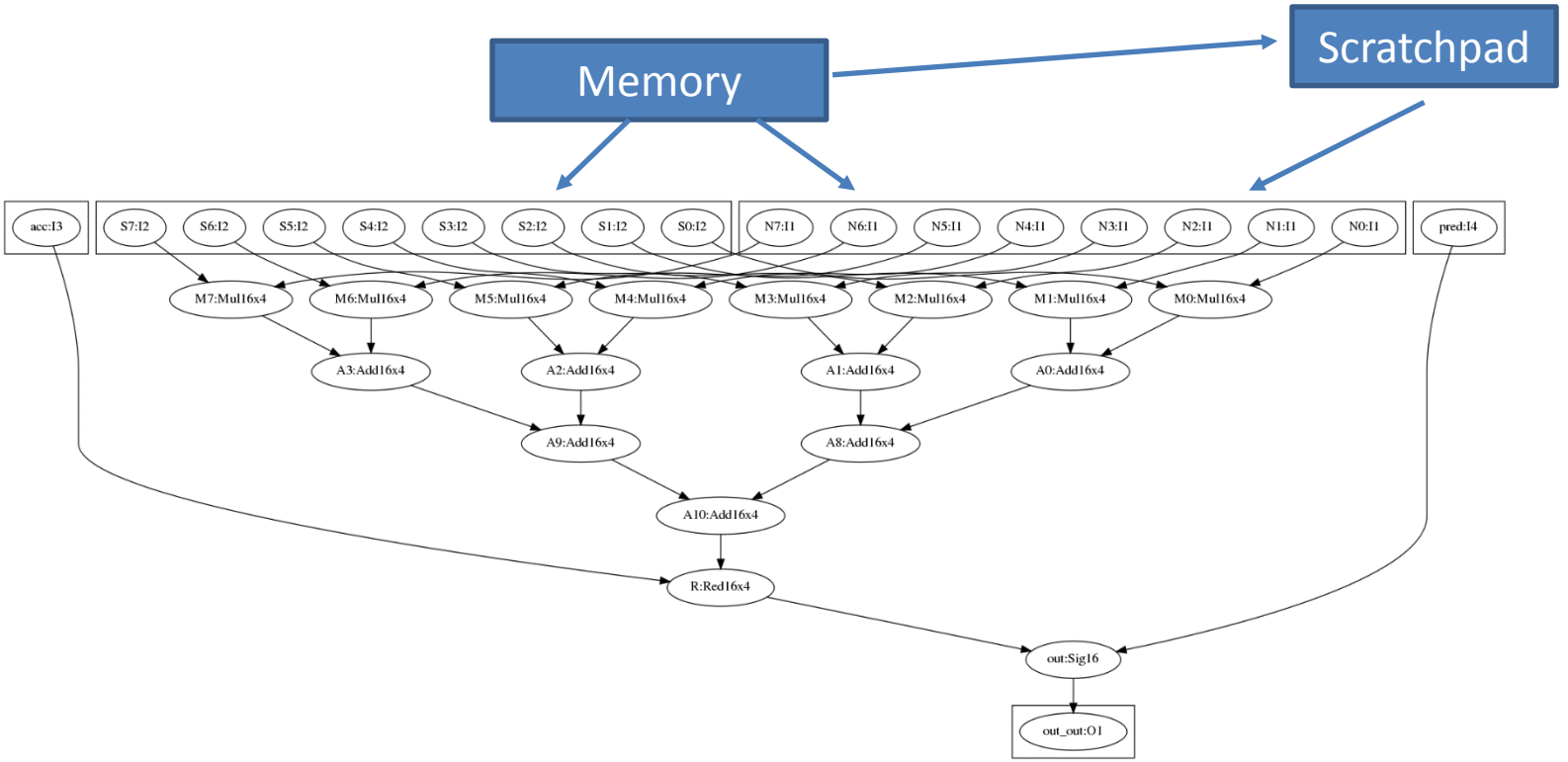
```
for(int i = 0; i < 128; ++i) {
  SB_MEM_PORT(array[i], stride_size,
        acc_size, num_times, Port);
   …
}
```

```
for(int i = 0; i < 128; i+=2) {
  SB_MEM_PORT(array[i], stride_size,
       acc_size, num_times*2, Port);
   …
}
```

```
 SB_MEM_PORT(array[0], stride_size,
       acc_size, num_times*128, Port);

for(int i = 0; i < 128; ++i) {
  …
}
```

3. Cache → CGRA Bandwidth (1)

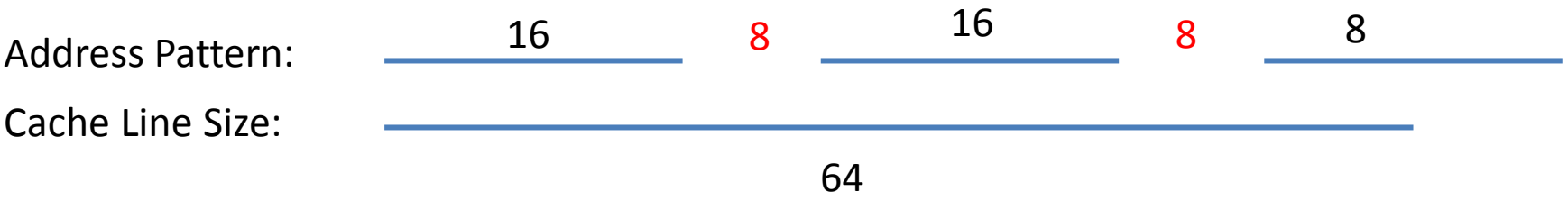
- ***Principle 1:*** Only 64-bytes per cycle can come from memory
  - Can feed One 8-wide port, Two 4-wide ports, Four 2-wide ports
  - Use scratch streams to supplement memory streams



3. Cache → CGRA Bandwidth (2)

- ***Principle 2:*** Not-accessed elements within a 64-byte cache line **COUNT** towards bandwidth

Stream:
access_size = 16 bytes
stride_size = 24 bytes

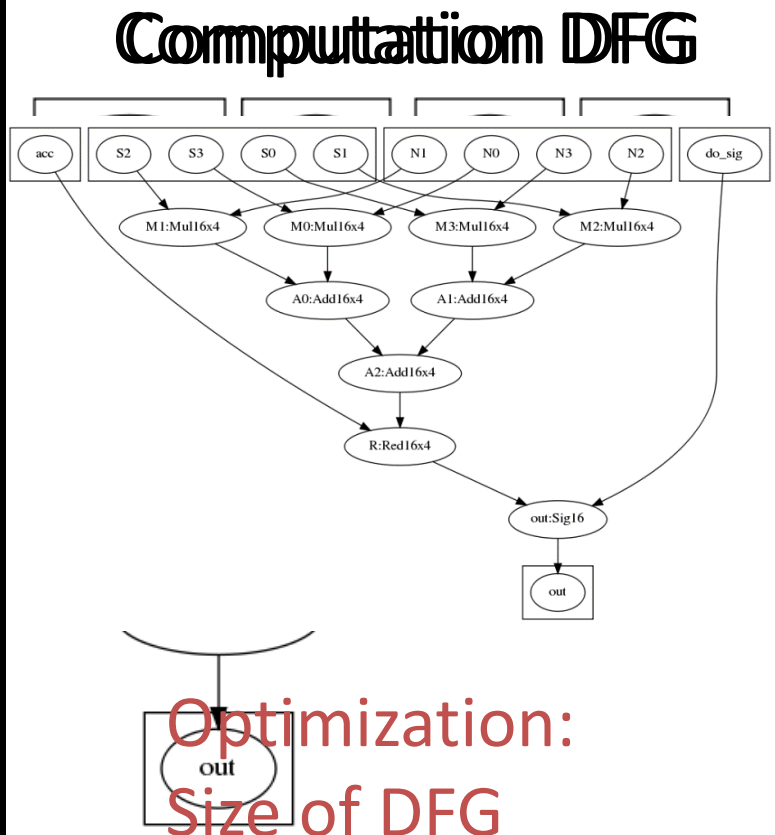


HINT 1: Don't use access patterns with "gaps" smaller than the cache line size.

HINT 2: Try to align accesses with cache line boundaries

Optimizing Classifier Layer

```
SD_Config(classifier_cfg, sizeof(cfg));

SD_Mem_Port(synapse, 8,
            8, Ni * Nn/4,Port_S);

SD_Mem_Scratch(neuron_i, Ni * 2,
               Ni * 2, 1, 0);
SD_Barrier_Scratch_Wr();
SD_Scratch_Port(0, Ni * 2,
                Ni * 2, 1, Port_N);

for (n = 0; n < Nn; n++) {
  SD_Const_Port(0, 1, Port_acc);
  SD_Const_Port(0, Ni/4 - 1, Port_do_sig);
  SD_Const_Port(1, 1, Port_do_sig);
  SD_Port_Port(Port_out, Ni/4 - 1, Port_acc);
  SD_Port_Mem(Port_out, 1, &neuron_n[i])
}

SD_Barrier_All;
```

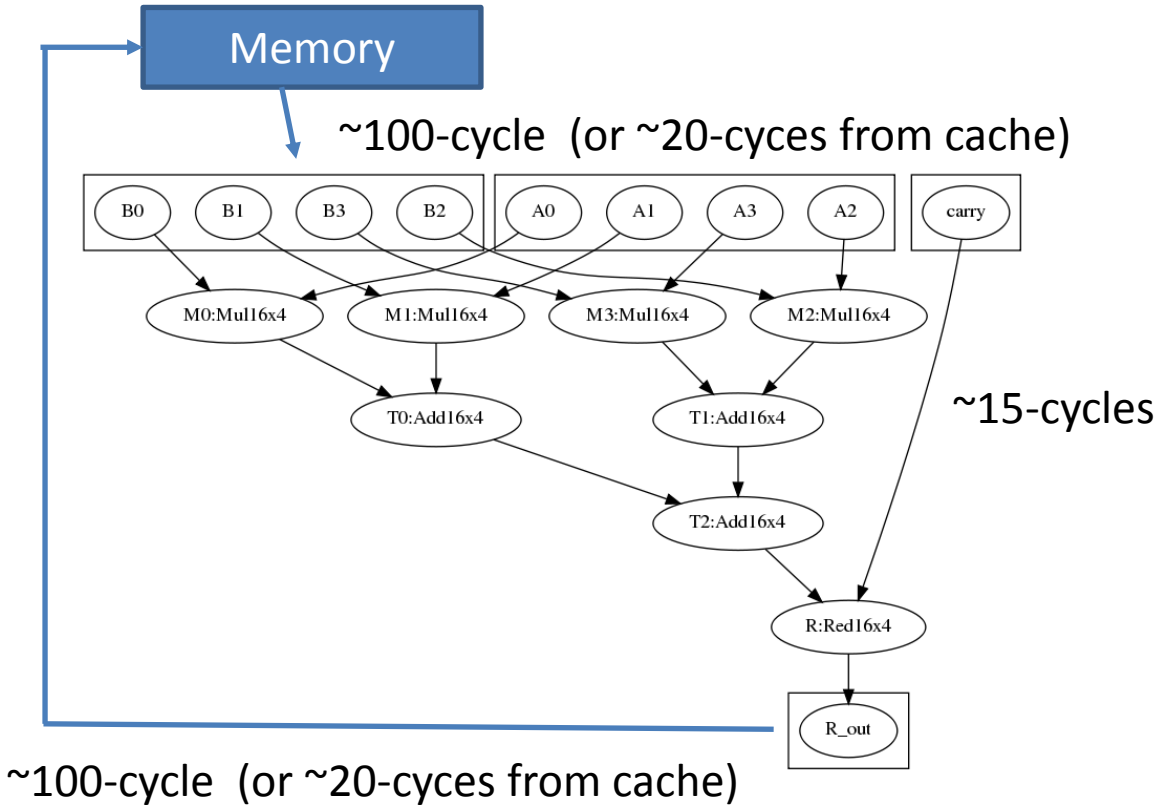


Optimization: Size of DFG

Optimization: Scratch for Memory B/W

6. Initialization/Draining Latency (Memory & CGRA)

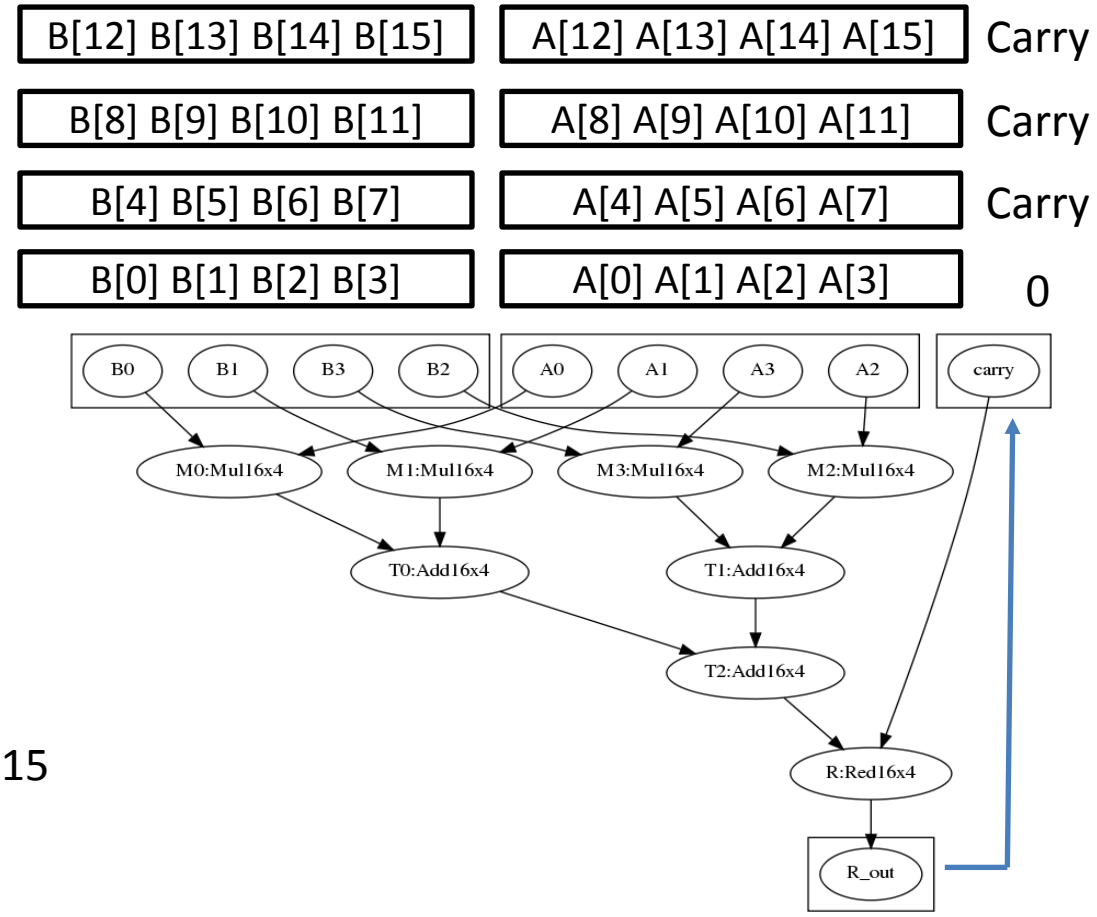
- ***Principle:*** Hide memory latency by having "longer pipelined phases"



7. Length of Recurrence through CGRA

- ***Principle:*** Number of independent instances should be > the length of the longest recurrence.

Dot Product of arrays B and A

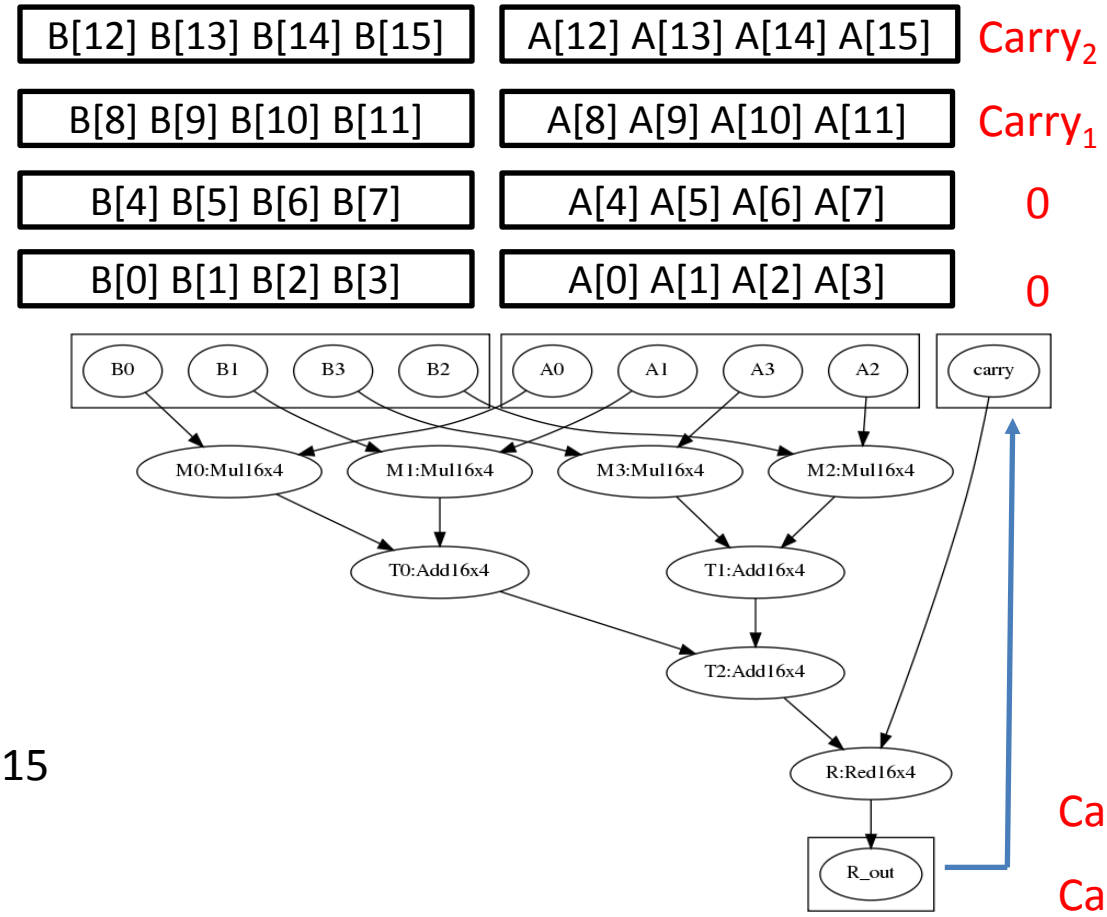


Latency = 15 Cycles

Instances / Cycle = 1 / 15

7. Length of Recurrence through CGRA (2)

Dot Product of arrays B and A

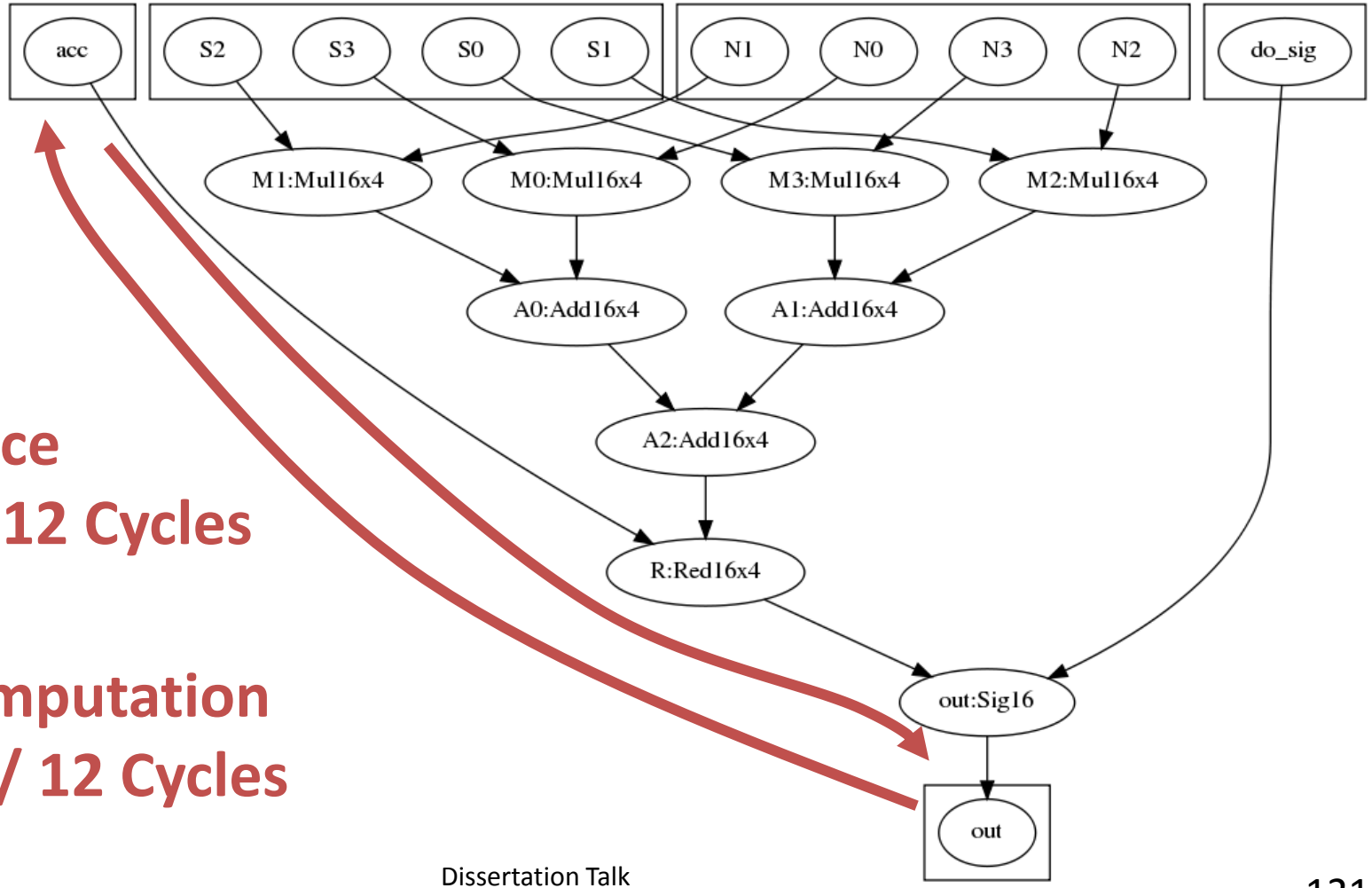


Latency=15 Cycles

Instances / Cycle = 2 / 15

Recurrence Serialization Overhead

Maximum Computation Rate =
 # Pipelinable Instances / Recurrence Length

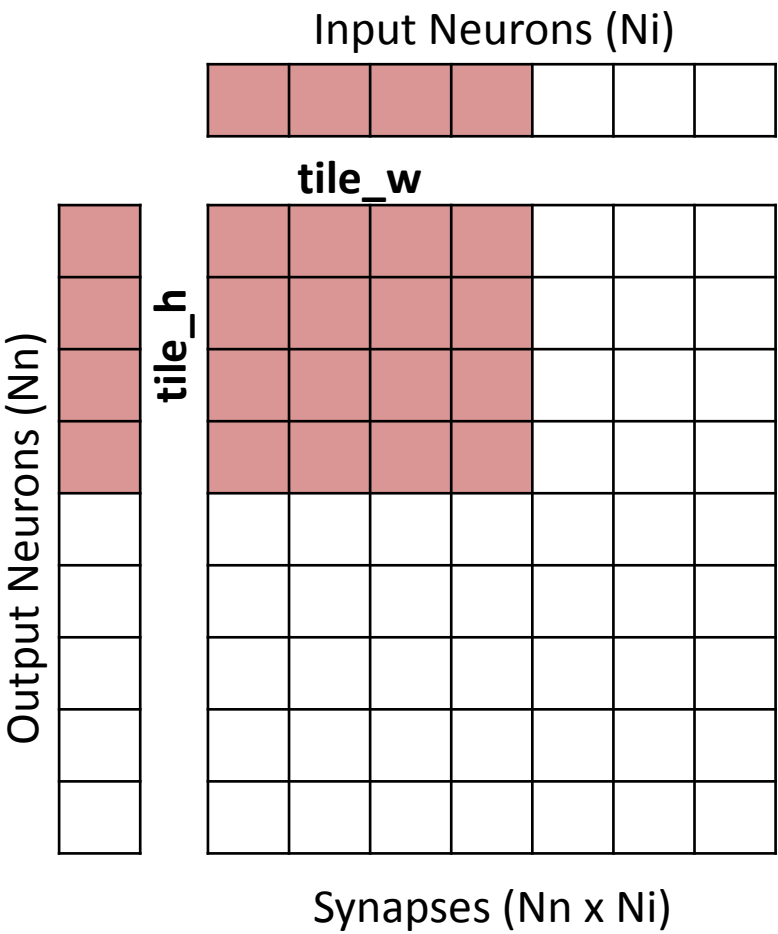


Recurrence Length = 12 Cycles

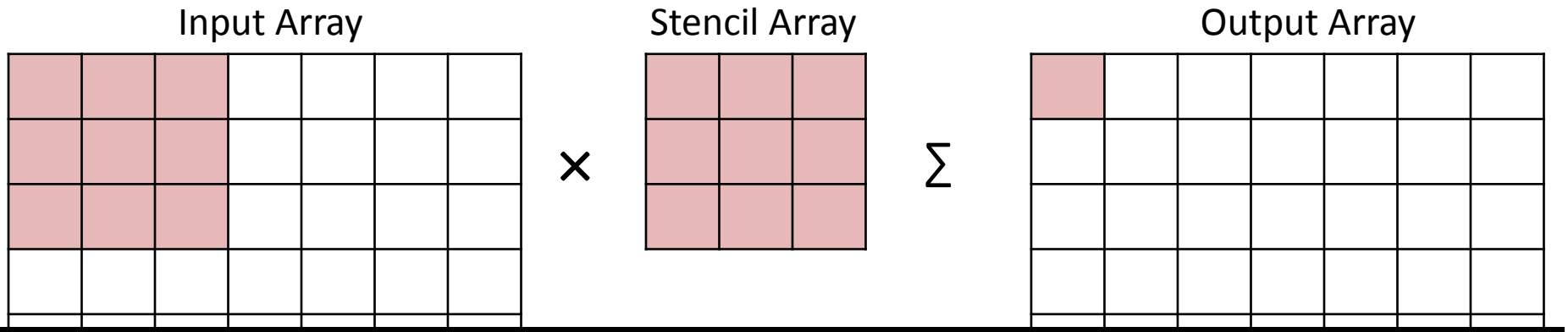
Max. Computation Rate = 1 / 12 Cycles

Pipelining Classifier Layer

```
SD_Config(classifier_cfg)

SD_Mem_Scratch(neuron_i, 0,Ni*2,1, 0)
SD_Barrier_Scratch_Write()

for (n = 0; n < Nn; n+=tile_h) {
  SD_Constant(0, tile height, Port acc)
  for(i = 0; i < Ni; i+=tile_w) {
    if(not last_iter) {
      SD_Constant(0,   tile_h,P_do_sig)
      SD_Port_Port(P_out, tile_h,P_acc)
    } else {
      SD_Constant(0,      tile_h,P_do_sig)
      SD_Port_Mem(Port_out, 1, &neuron_n[i])
    }
    SD_Scratch_Port(i*2, 0, 8*tile_w, 1,
                    Port_N)
    SD_Mem_Port(&synapse[n][i],
                2*Ni, 8*tile_w, tile_h, Port_S)
  }
}
SD_Barrier_All();
```

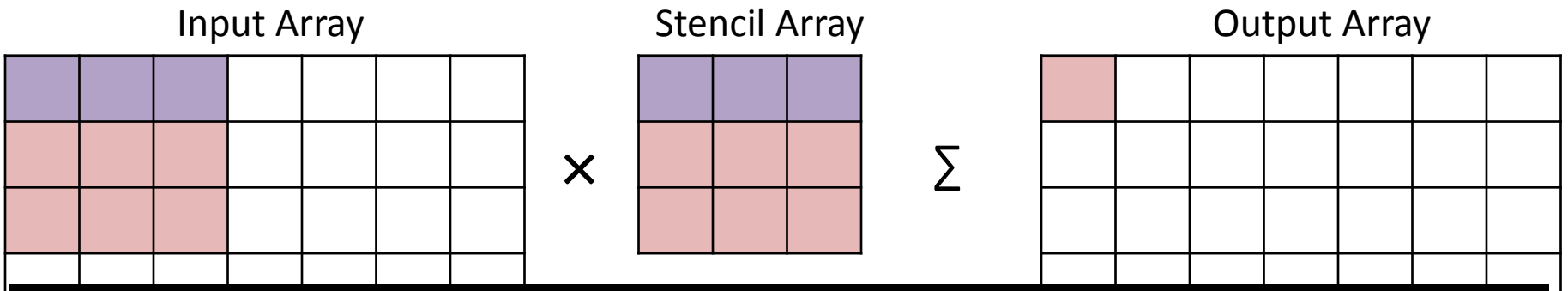


2D Stencil Example




```
for (r=0; r<row_size-2; r++) {
  for (c=0; c<col_size-2; c++) {
    temp = (TYPE)0;
    for (k1=0;k1<3;k1++) { //Row access
      for (k2=0;k2<3;k2++) { //column access
        mul = filter[k1*3 + k2] * orig[(r+k1)*col_size + c+k2];
        temp += mul;
      }
    }
    sol[(r*col_size) + c] = temp;
  }
}
```

"Easy" Approach



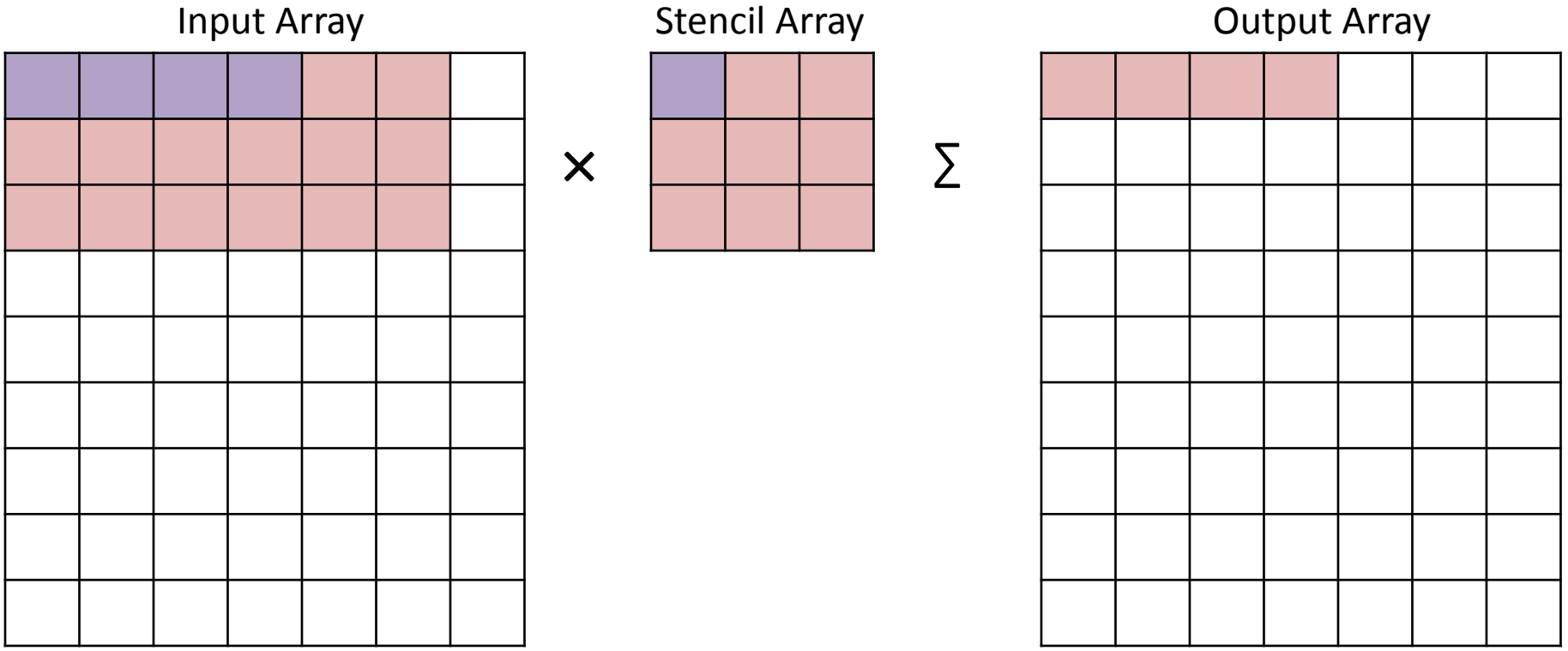

```
for (r = 0; r < row_size - 2; r++) {
  for (c = 0; c < col_size - 2; c++) {
    SD_Constant(P_stencil_sb_carry, 1, 1);
    for (k1 = 0; k1 < 3; k1++) {
      SD_Mem_Port((orig + (r + k1) * col_size + c),
                  sizeof(TYPE), sizeof(TYPE), 4, P_stencil_sb_I);
      SD_Mem_Port(filter + (k1 * 3),
                  sizeof(TYPE), sizeof(TYPE), 4, P_stencil_sb_F);
    }
    SD_port_Port(P_stencil_sb_R, P_stencil_sb_carry, 2);
    SB_Port_Mem(P_stencil_sb_R, sizeof(TYPE),
                sizeof(TYPE), 1, sol + (r * col_size) + c);
  }
}
SB_Barrier_All();
```

Easy Approach's Bottlenecks

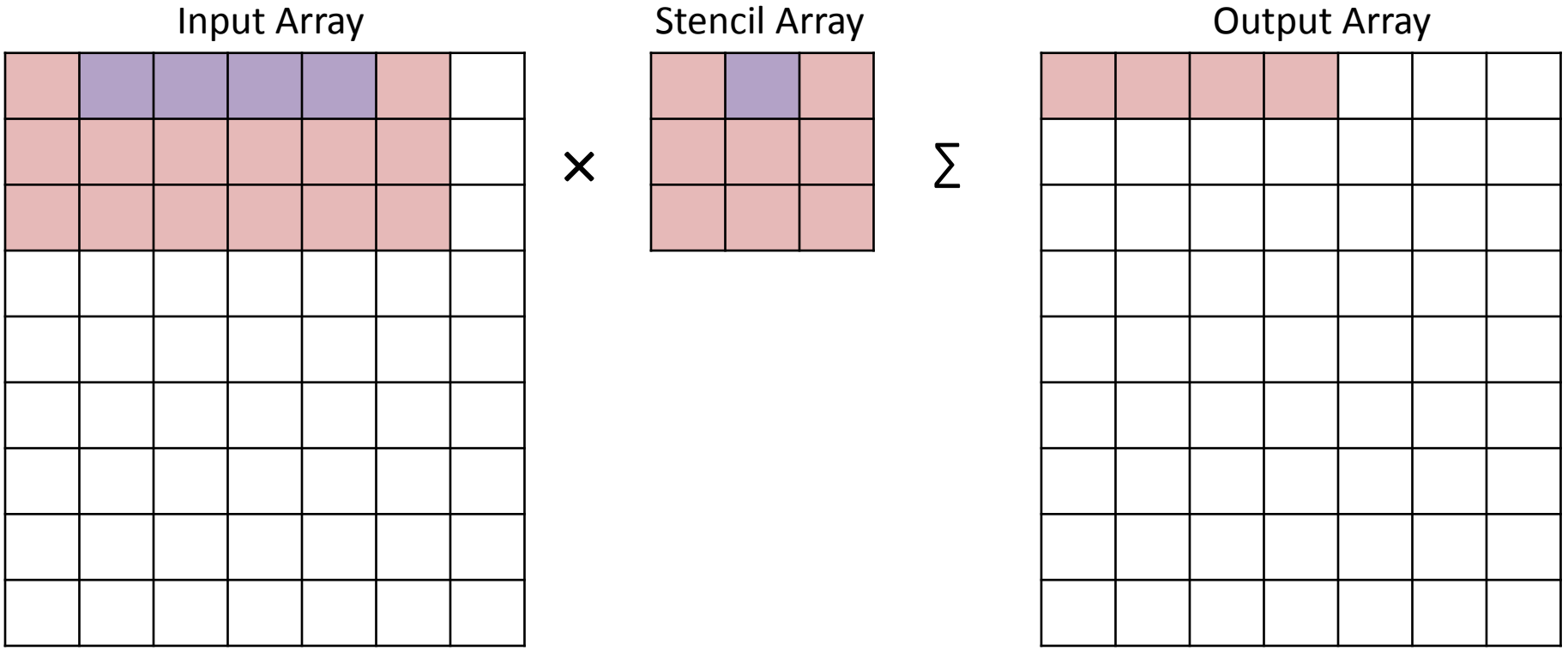
1. Computations Per CGRA Instance (only 3 mults!)
2. General Core Instructions (core insts == CGRA insts)
3. Cache → CGRA Bandwidth (wasted b/c of acc_size)
4. Initialization/Draining Latency
5. Length of Recurrence through CGRA

(no independent computations through CGRA)

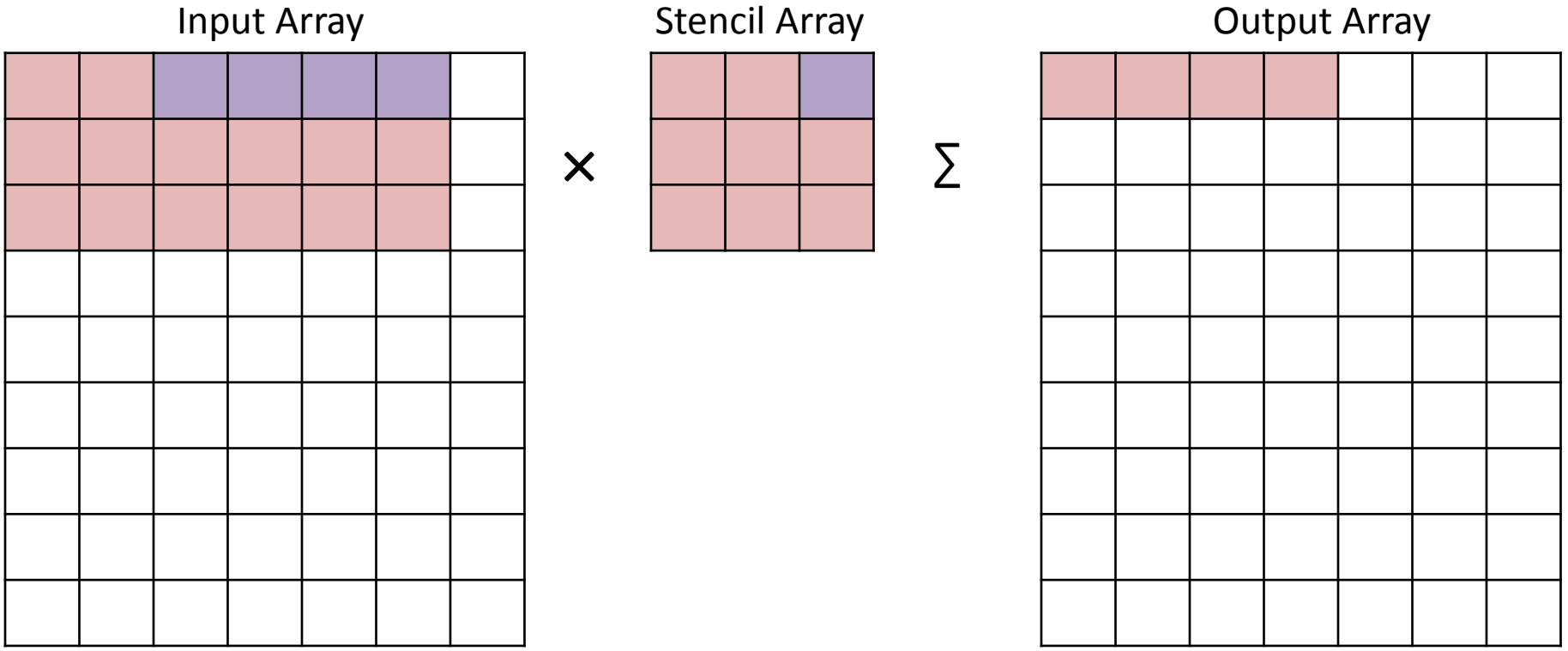
Better Approach (probably not best)



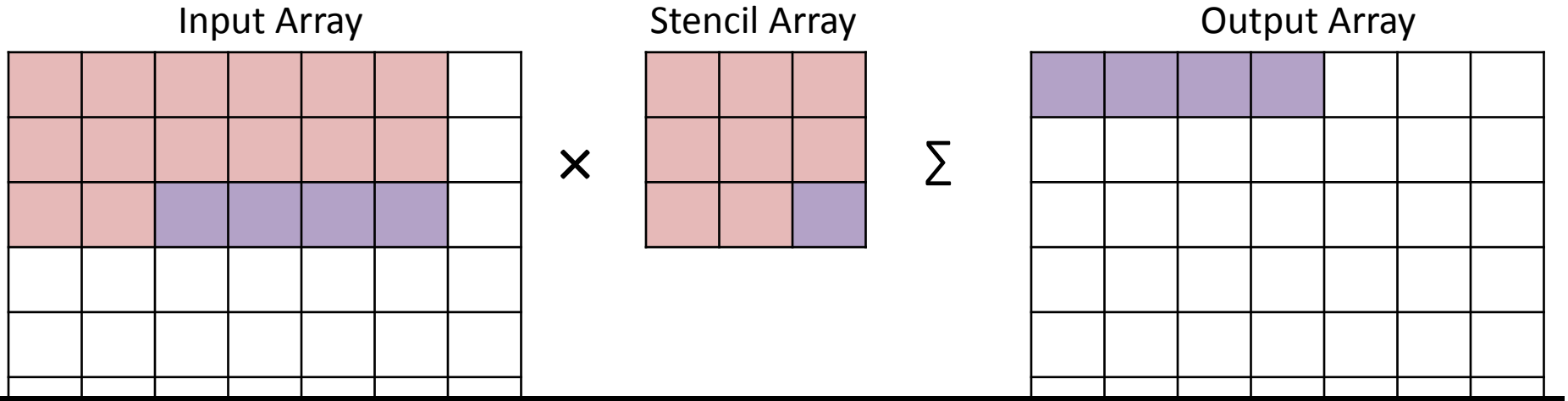
Better Approach (probably not best)



Better Approach (probably not best)



Better Approach (probably not best)

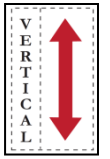



```
for (r=0; r<row_size-2; r++) {
  for (c=0; c<col_size-2; c++) {
    temp = (TYPE)0;
    for (k1=0;k1<3;k1++) { //Row access
      for (k2=0;k2<3;k2++) { //column access
        mul = filter[k1*3 + k2] * orig[(r+k1)*col_size + c+k2];
        temp += mul;
      }
    }
    sol[(r*col_size) + c] = temp;
  }
}
```

Better Approach's Bottlenecks

1. Computations Per CGRA Instance (up to 8 mults!)
2. General Core Instructions (core insts << CGRA insts)
3. Cache → CGRA Bandwidth (acc_size > cache_size)
4. Scratchpad → CGRA Bandwidth
5. Memory → Cache Bandwidth
6. Initialization/Draining Latency
7. Length of Recurrence through CGRA (if you stripmine the c-loop past the DFG width, you can stream multiple independent computations through the CGRA!)

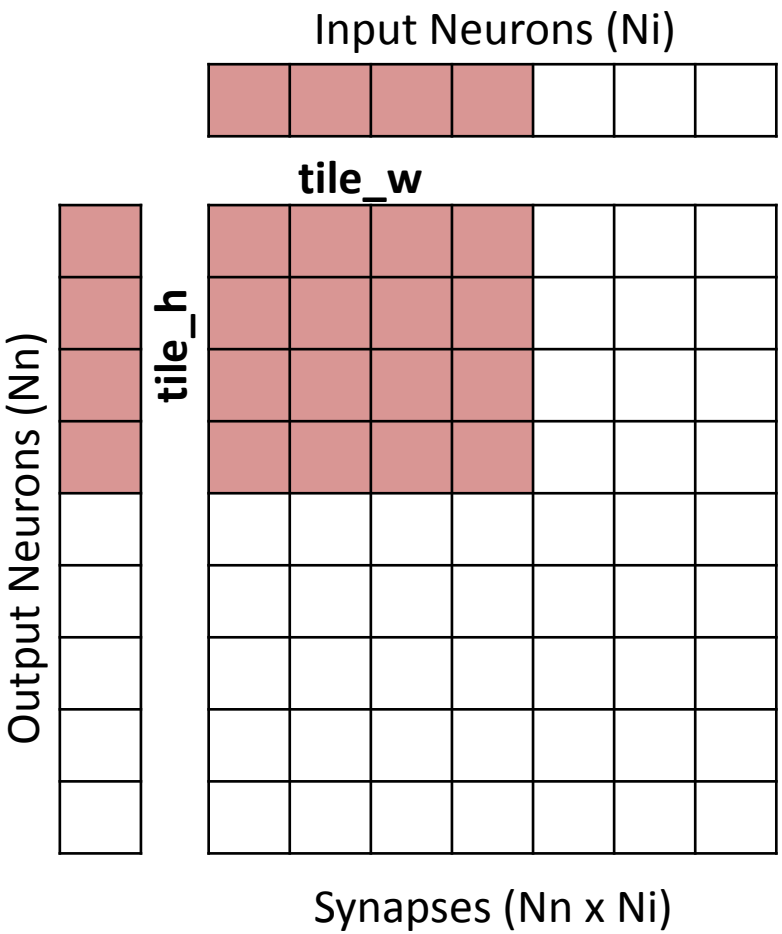
Programming Restrictions



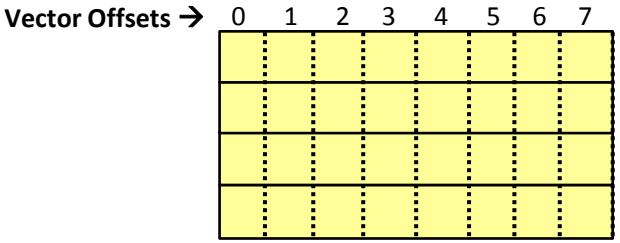
- CGRA Instruction Types & Data-width
- Shape of the stream (strided)
- Width of input/output ports
- Number of simultaneous streams
- Issue to free-port (data always balanced)

Pipelining Classifier Layer

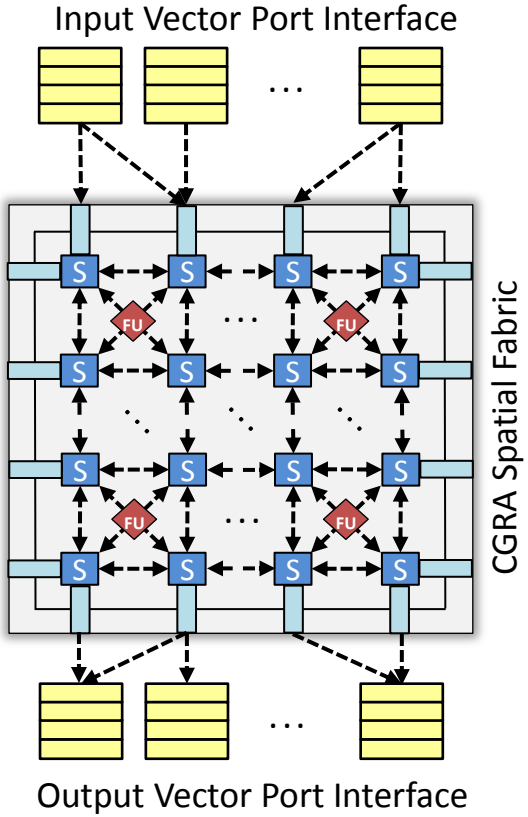
```
SD_Config(classifier_cfg, sizeof(cfg))

SD_Mem_Scratch(neuron_i, Ni * 2,
               Ni * 2, 1, 0);
SB_Barrier_Scratch_Wr();

for (n = 0; n < Nn; n += tile_h) {
  SD_Const_Port(0, tile_h, Port_acc);

  for(i = 0; i < Ni; i += tile_w) {
    if(not last_iter) {
      SD_Const-Port(0, tile_h, Port_do_sig);
      SD_Port_Port(P_out, tile_h, Port_acc);
    } else {
      SD_Const_Port(0, tile_h, Port_do_sig);
      SD_Port_Mem(Port_out, 1, &neuron_n[i]);
    }
    SB_Scracth_Port(i * 2, 8 * tile_w,
                    8 * tile_w, 1, Port_N);
    SB_Mem_Port(&synapse[n][i], 2 * Ni,
                8 * tile_w, tile_h, Port_S);
  }
}
SD_Barrier_All;
```



CGRA – Vector Port Interface



4 Entry Vector Port (512b or 64B wide) –
Each element 8B or 64b)



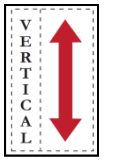
- Vector ports facilitate "vector/SIMD execution and can store entire cache-line in a cycle (8 wide)
- Vector ports' offsets are connected to CGRA input links – Mapping done by hardware architects recorded as ***Softbrain Hardware Parameter Model***
- Hardware parameter model is passed to scheduler/compiler for mapping software DFG ports to hardware vector ports
- Enable flexible hardware-software interface for variable width SIMD-execution

Example vector port to CGRA links mapping
[VPORT_Num]: [Offset]:[CGRA Link Num]

VPORT_IN 0: 0:2, 1:5, 2:8, 3:11, 4:17, 5:20, 6:23, 7:26
VPORT_IN 1: 0:4, 1:7, 2:10, 3:16, 4:19, 5:22, 6:25, 7:31

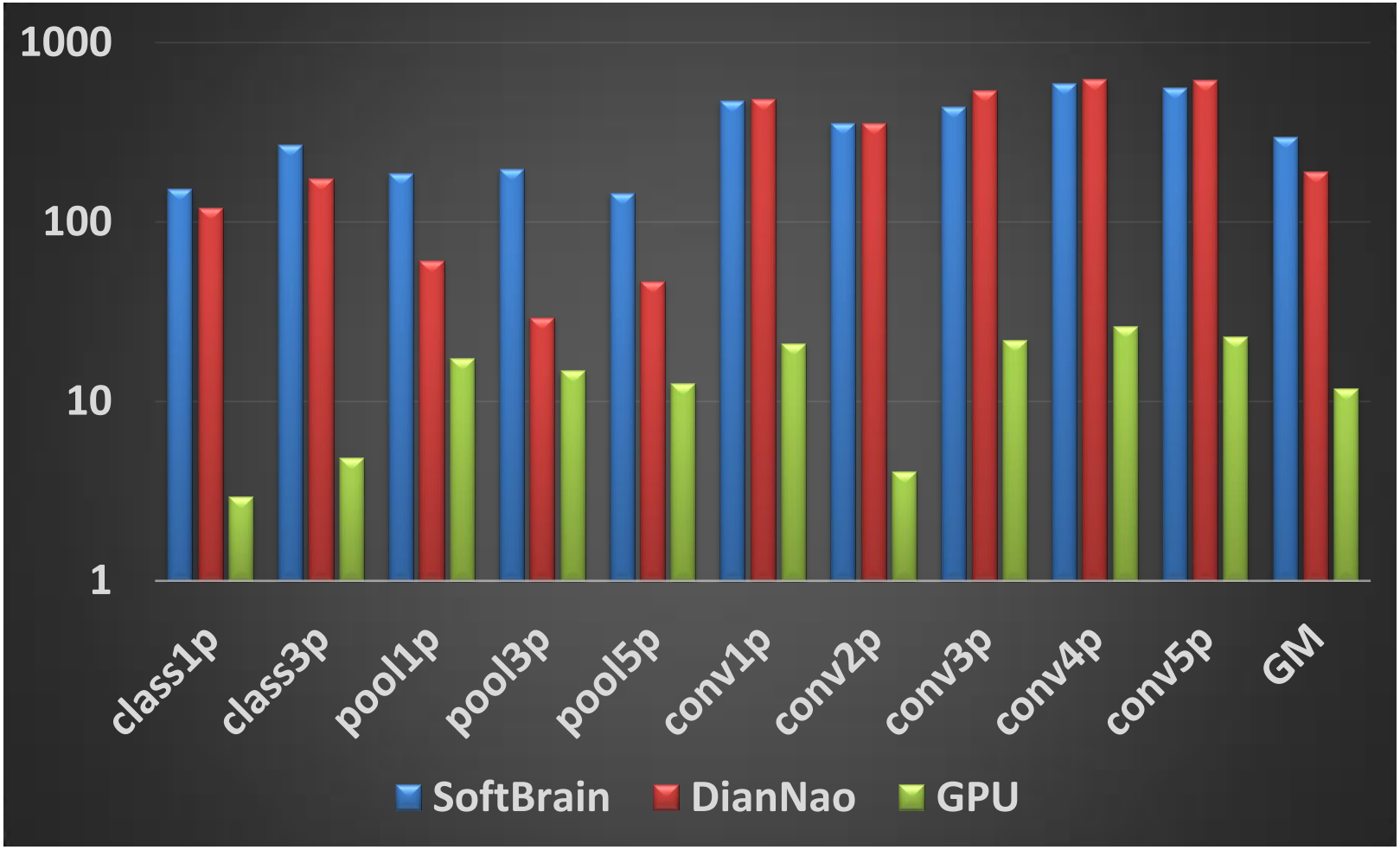
VPORT_OUT 0: 0:1, 1:3, 2:5, 3:6, 4:8, 5:9, 6:11, 7:12

Workload Characterization for Application Specific Softbrain

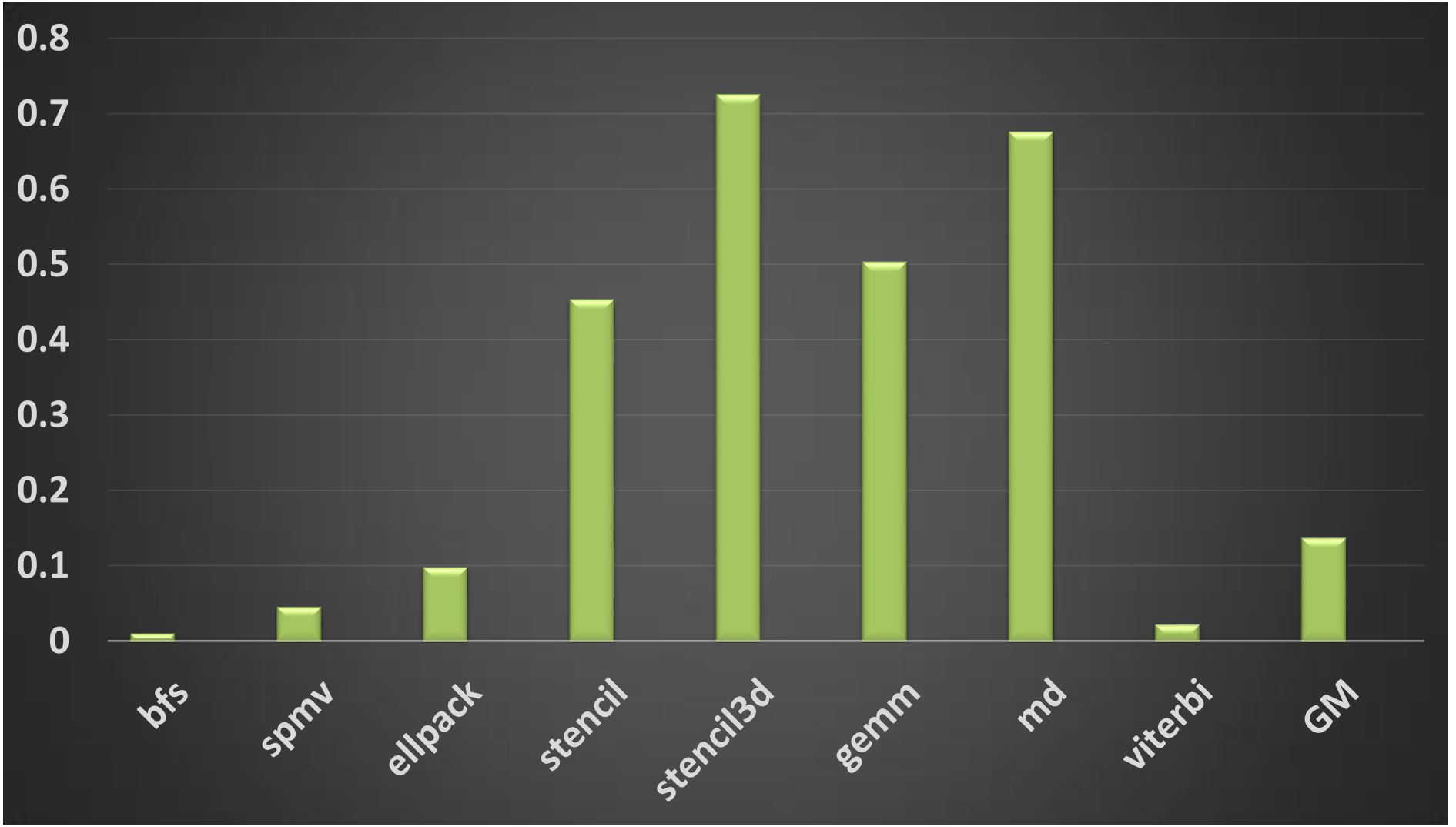


Implemented Codes	Stream Patterns	Datapath
bfs	Indirect Loads/Stores, Recurrence	Compare/Increment
gemm	Affine, Recurrence	8-Way Multiply-Accumulate
md-knn	Indirect Loads, Recurrence	Large Irregular Datapath
spmv-crs	Indirect, Linear	Single Multiply-Accumulate
spmv-ellpack	Indirect, Linear, Recurrence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

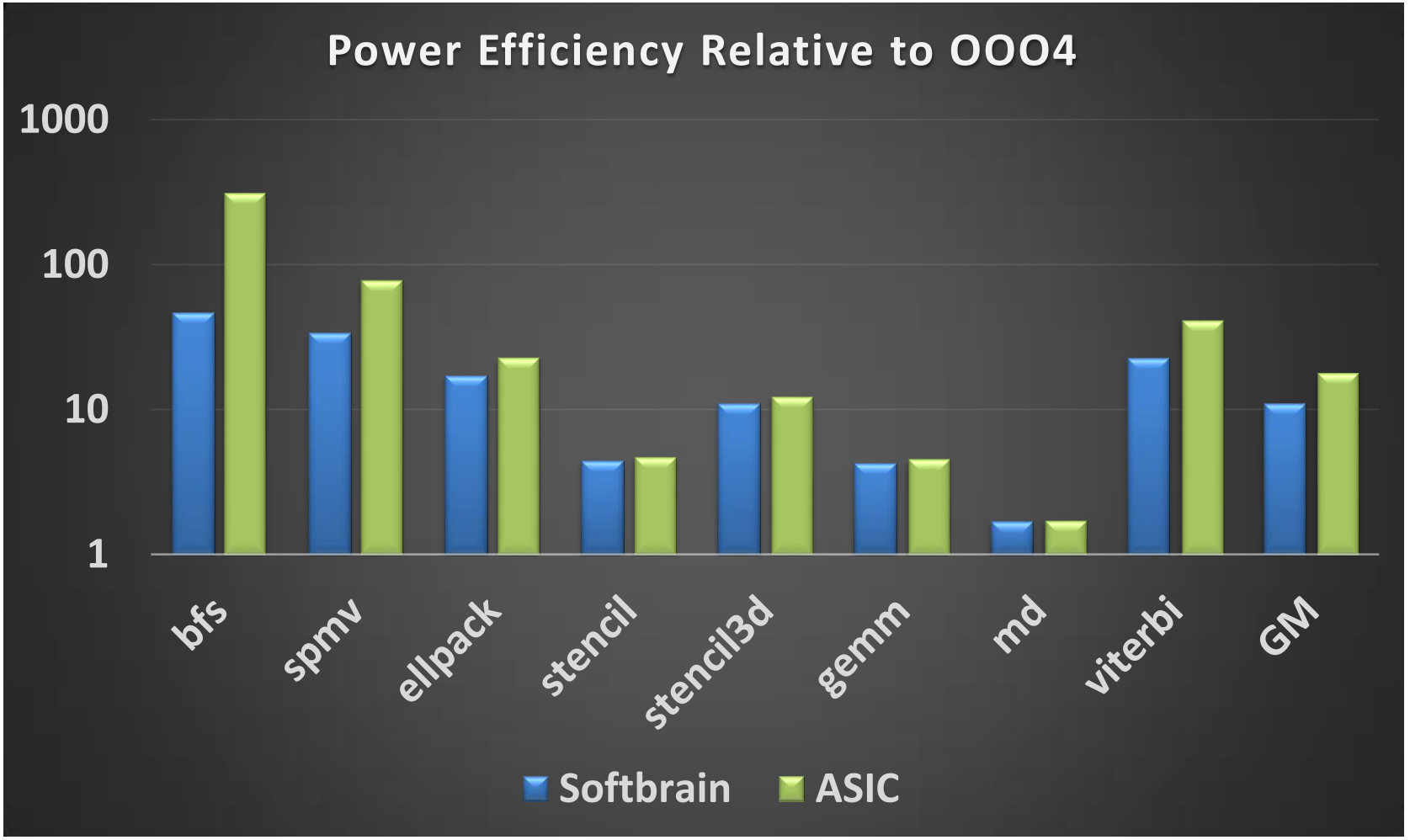
Softbrain vs. DianNao vs. GPU



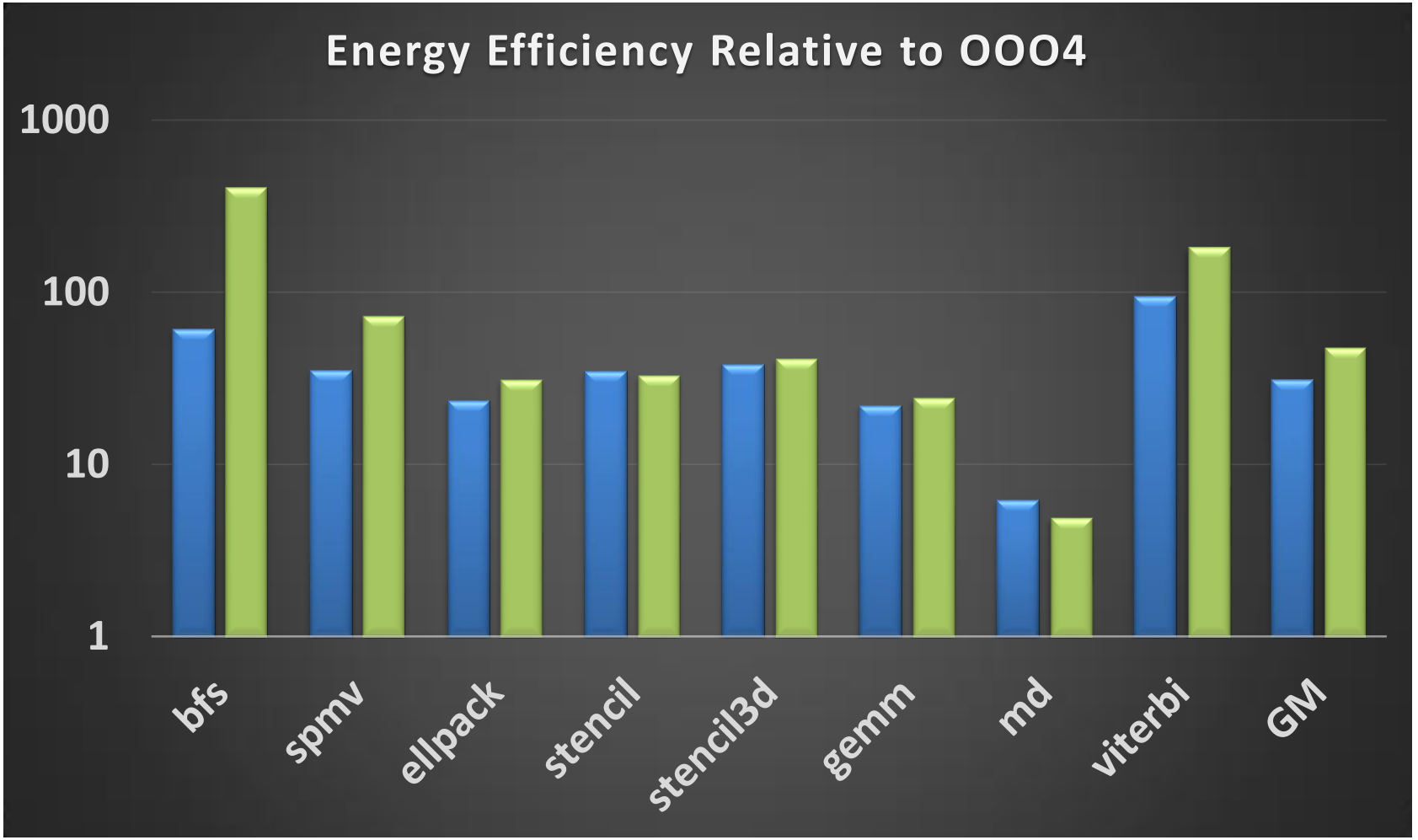
ASIC Area Relative to Softbrain



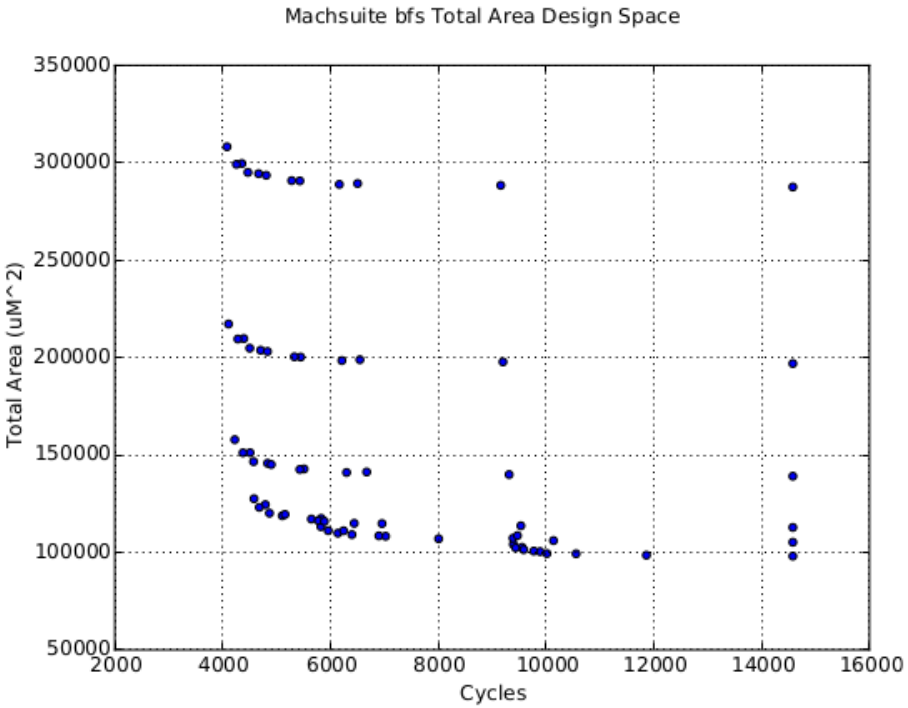
Softbrain vs. ASIC
Power Efficiency Comparison



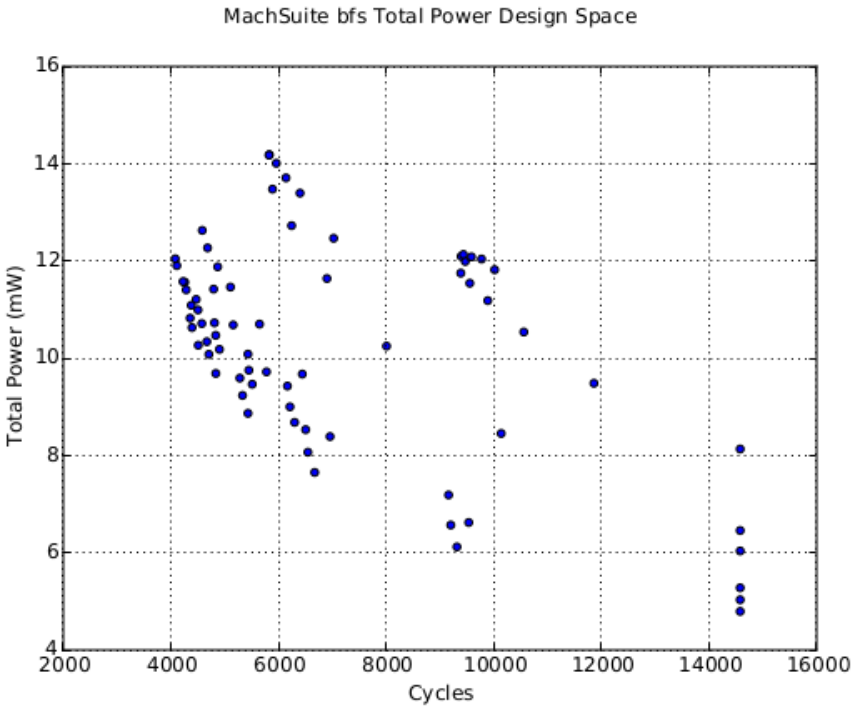
Softbrain vs. ASIC
Energy Efficiency Comparison



Design Space Exploration for ASIC Comparison

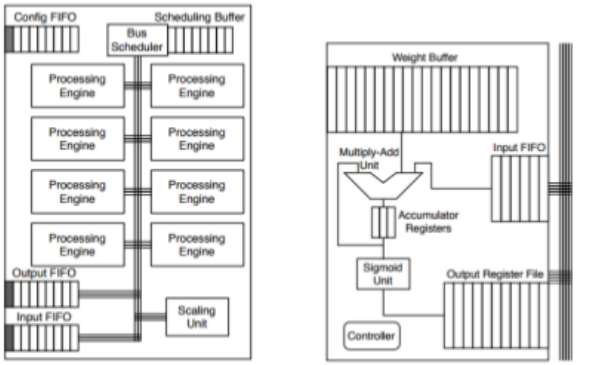


(a) Total Cycles of bfs ASIC for Various Area Optimized Design Points



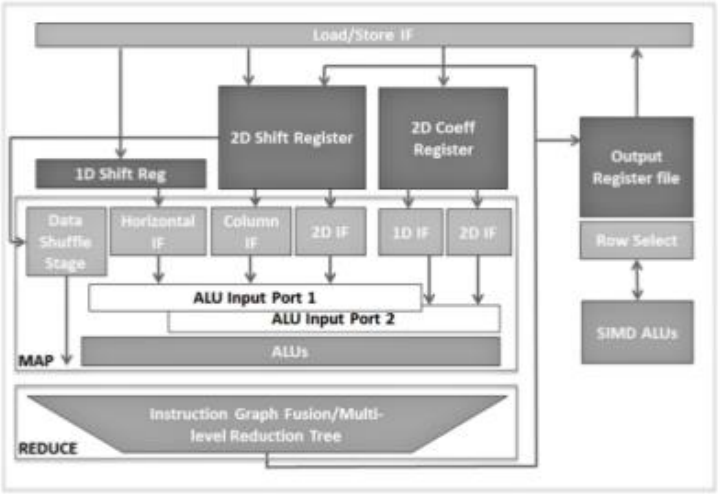
(b) Total Cycles of bfs ASIC for Various Power Optimized Design Points

DSA Architectures

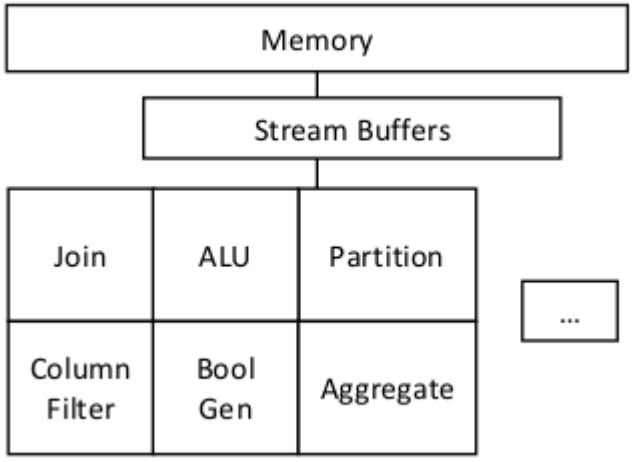


a) 8-PE NPU b) Single processing engine (PE)

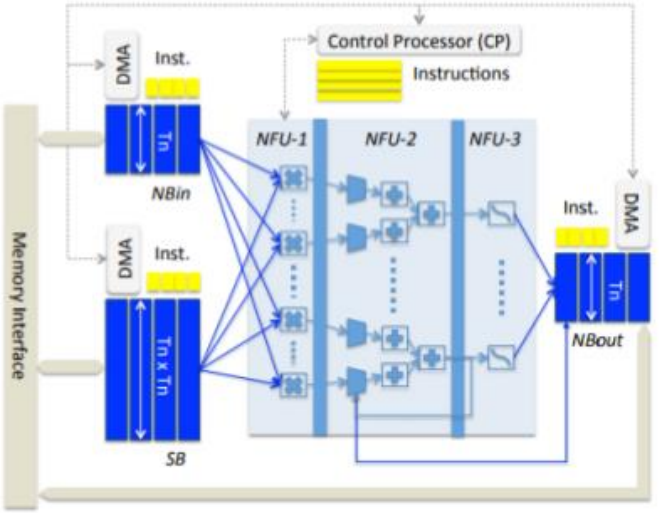
NPU



Convolution Engine

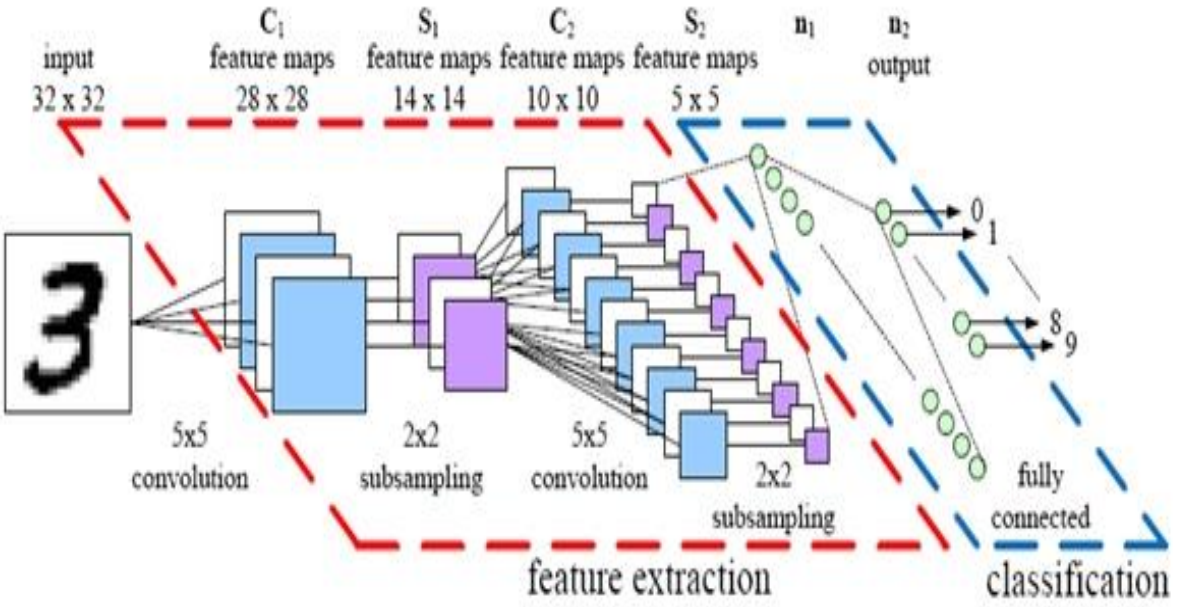


Q100

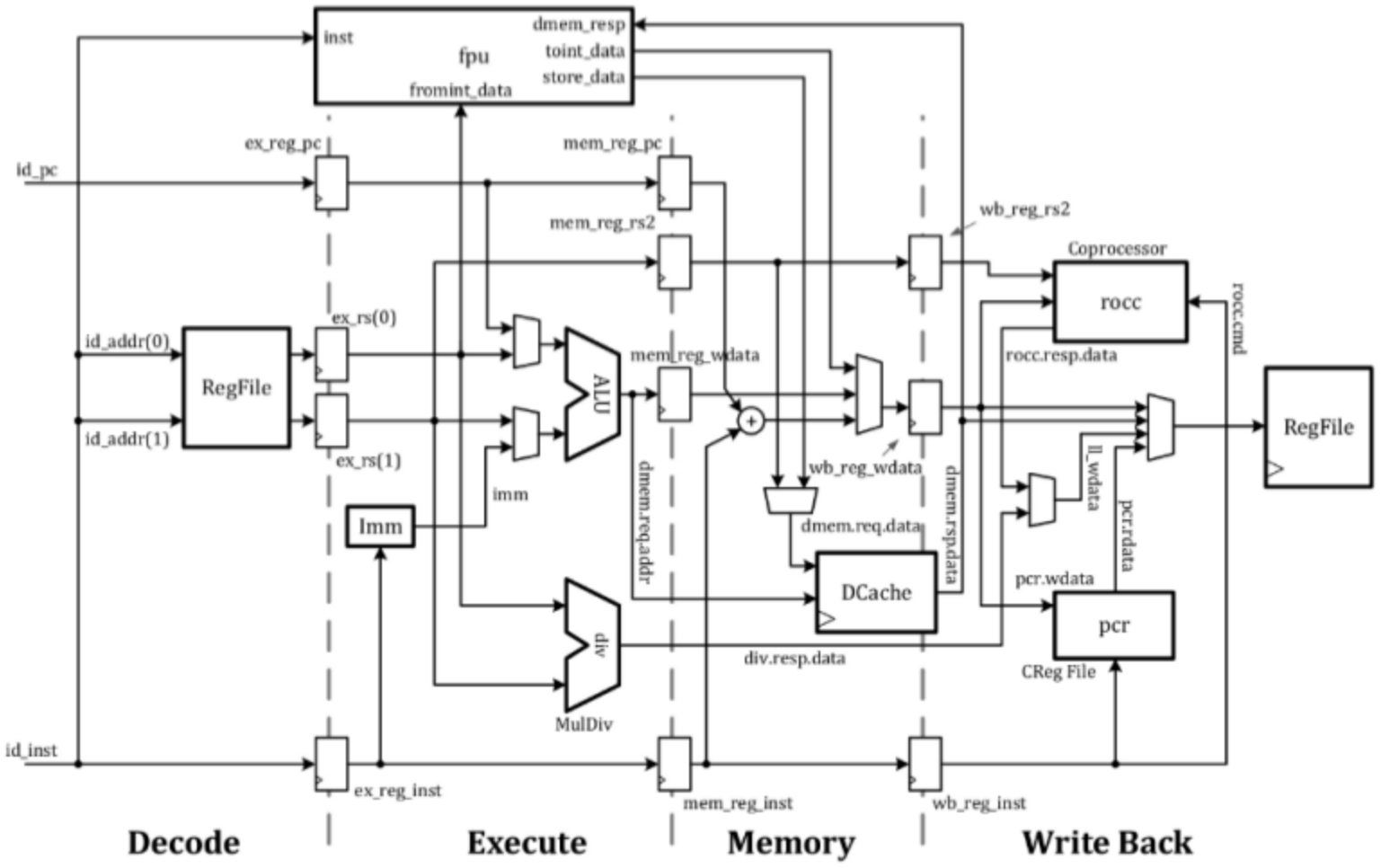


DianNao

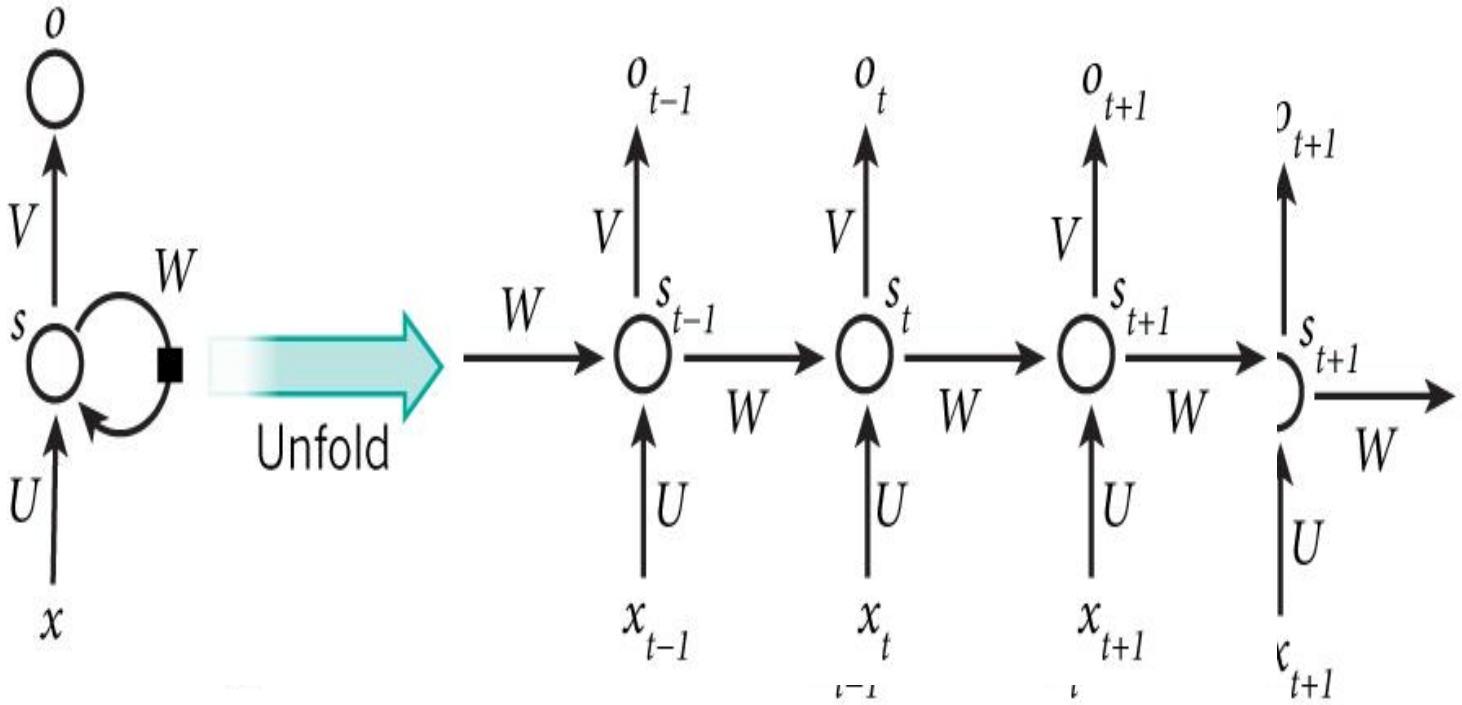
Convolutional Neural Network



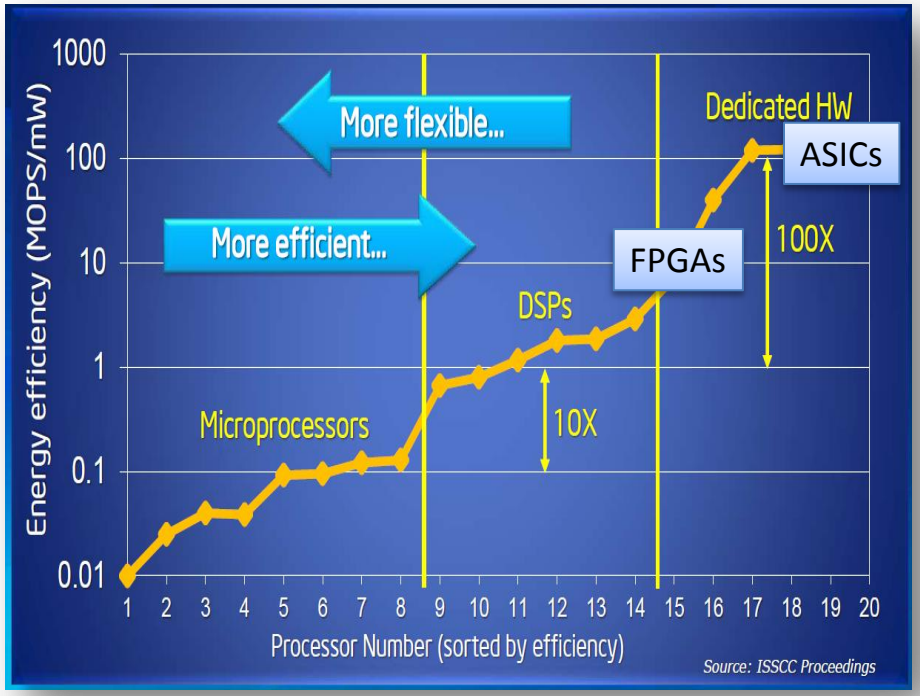
Rocket Core RoCC Interface



Recurrent Neural Network



Specialization Spectrum



Source: Bob Broderson, Berkeley Wireless group

More gains the lower you go

Code specialization	10x
Logic specialization	100x
Circuit specialization	1000x
Device specialization	10000x