Stream-Dataflow Acceleration

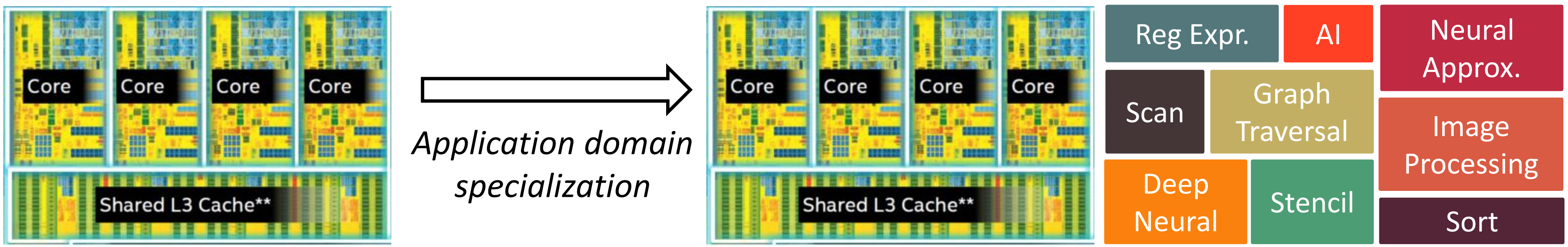
Tony Nowatzki[+], **Vinay Gangadhar***,
Newsha Ardalani*, Karu Sankaralingam*

44th ISCA, Toronto, ON, Canada
Accelerator Session (6A-4)
Tuesday June 27th, 2017

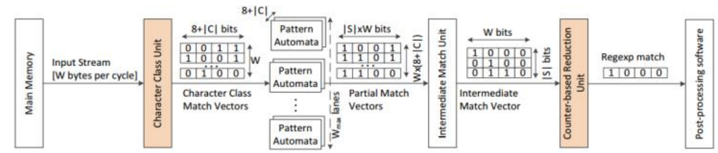
*University of Wisconsin-Madison
[+]University of California, Los Angeles

Era of Specialization

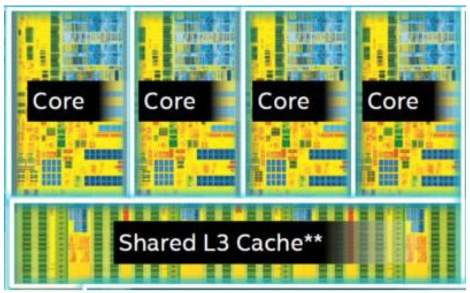
Traditional Multicore



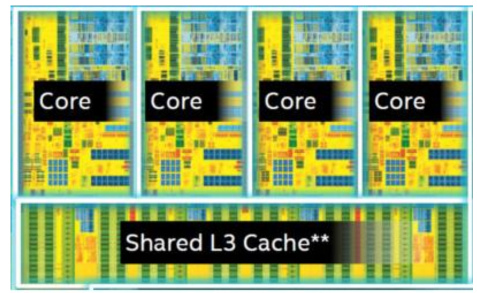
Era of Specialization



Traditional Multicore



Application domain specialization



Reg Expr.

AI

Neural Approx.

Scan

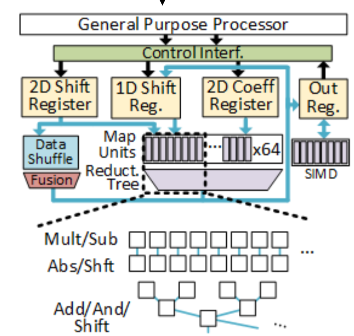
Graph Traversal

Image Processing

Deep Neural

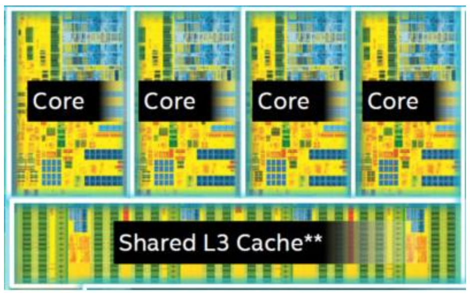
Stencil

Sort

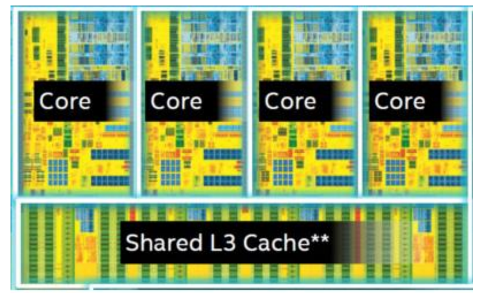


Era of Specialization

Traditional Multicore



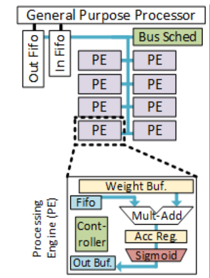
Application domain specialization

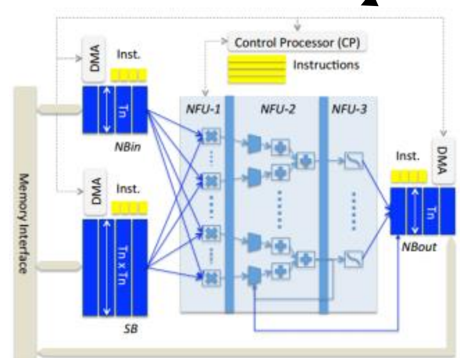


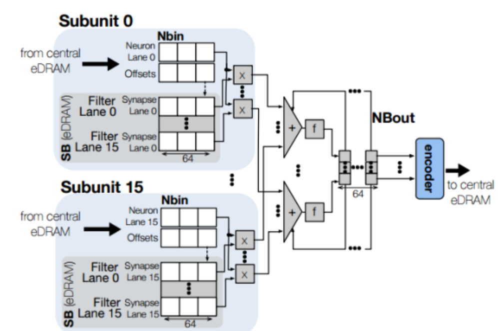
Reg Expr. | AI | Neural Approx.

Scan | Graph Traversal | Image Processing

Deep Neural | Stencil | Sort

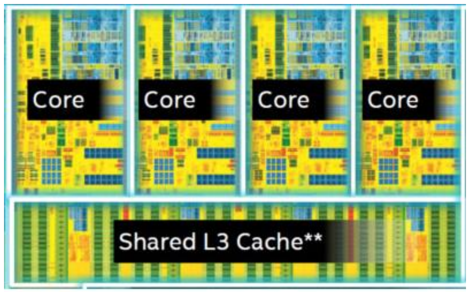




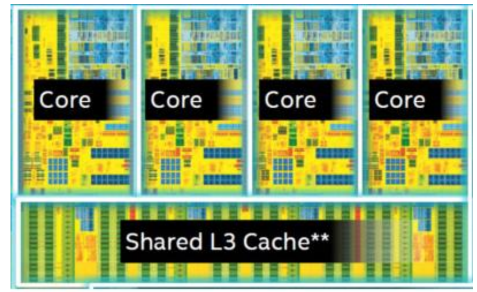


Era of Specialization

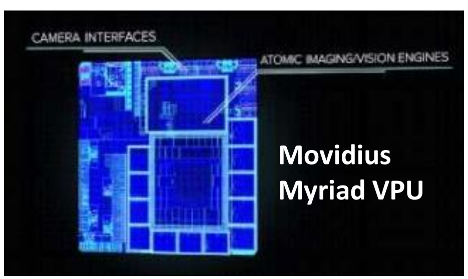
Traditional Multicore



Application domain specialization

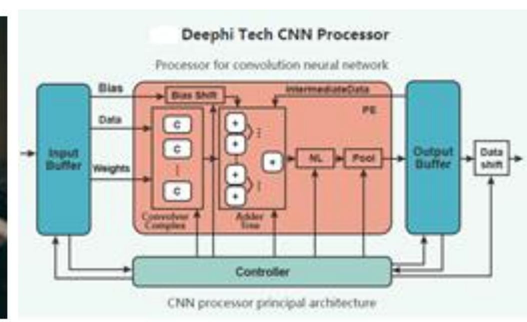


Reg Expr.	AI	Neural Approx.
Scan	Graph Traversal	Image Processing
Deep Neural	Stencil	Sort



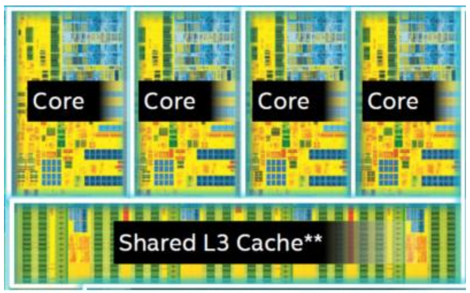
NVIDIA DGX-1 AI Accelerator

Catapult FPGA Accelerator



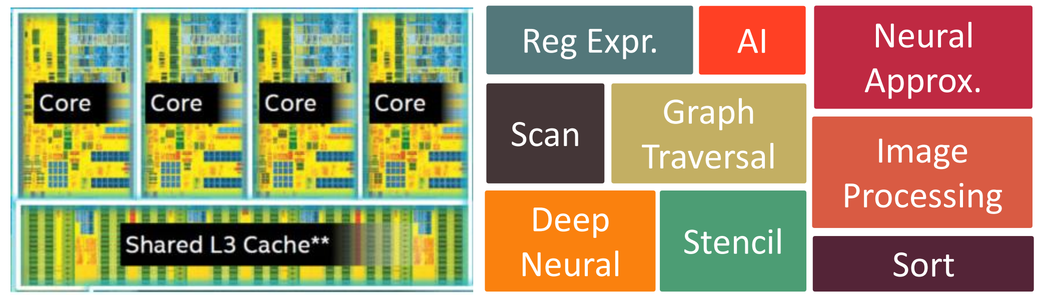
Era of Specialization

Traditional Multicore



Application domain specialization

Domain Specific Acceleration



Fixed-function Accelerators for specific domain:
Domain Specific Accelerators (DSAs)

+ High Efficiency

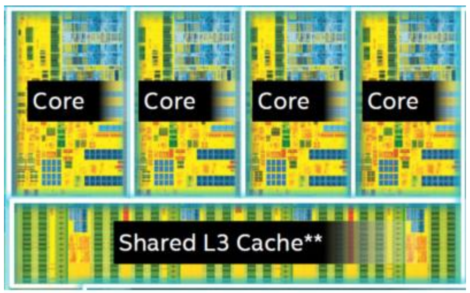
10 – 100x
Performance/Power
or
Performance/Area

three orders of magnitude less energy than a state of the art software DBMS, while the performance-oriented design outperforms 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

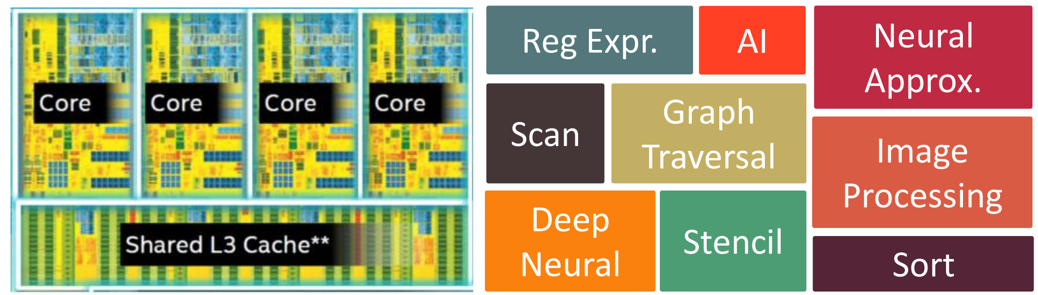
Era of Specialization

Traditional Multicore



Application domain specialization

Domain Specific Acceleration



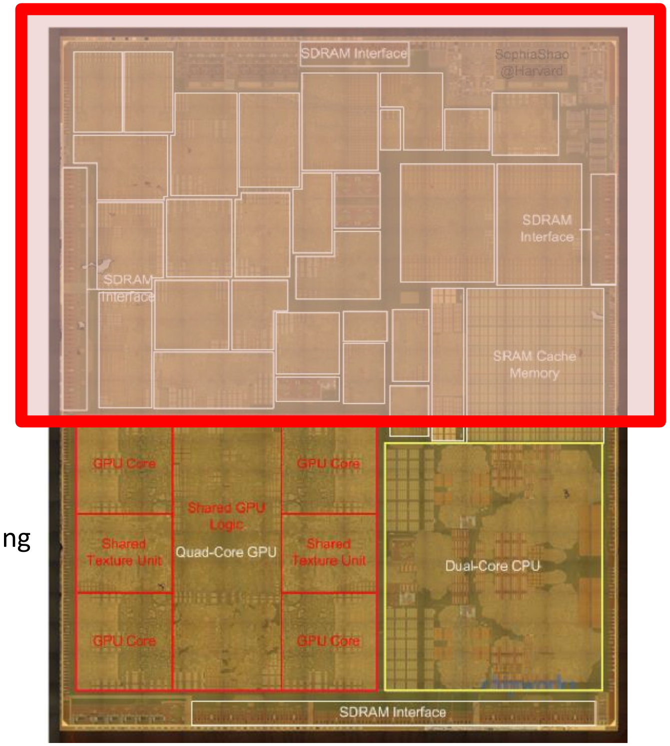
Fixed-function Accelerators for specific domain:
Domain Specific Accelerators (DSAs)

+ High Efficiency

10 – 100x
Performance/Power
or
Performance/Area

- Not programmable/re-configurable & Obsoletion prone
- Architecture, design, verification and fabrication cost
- Multi-DSA chip for "N" application domains → Area and cost inefficient

The Universal Accelerator Dream...



Source:
Malitel Consulting

Convert 100+ Accelerators

1 Programmable Accelerator Fabric

- Deep Neural
- Image Processing
- Automated Driving
- Compression
- Regex Matching
- Query Processing

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 Spectrum

Generality →

ASIC/ DSA | GPGPU | FPGA | DSP | SIMD | GPP

← *Efficiency*

(energy efficient computing)

Specialization Spectrum

Generality

ASIC/ DSA GPGPU FPGA DSP SIMD GPP

Efficiency

(energy efficient computing)

Specialization Principles

General Set of Micro-Architectural Mechanisms

UCLA

Specialization Spectrum

Generality

ASIC/ DSA

GPGPU FPGA DSP SIMD GPP

Efficiency

(energy efficient computing)

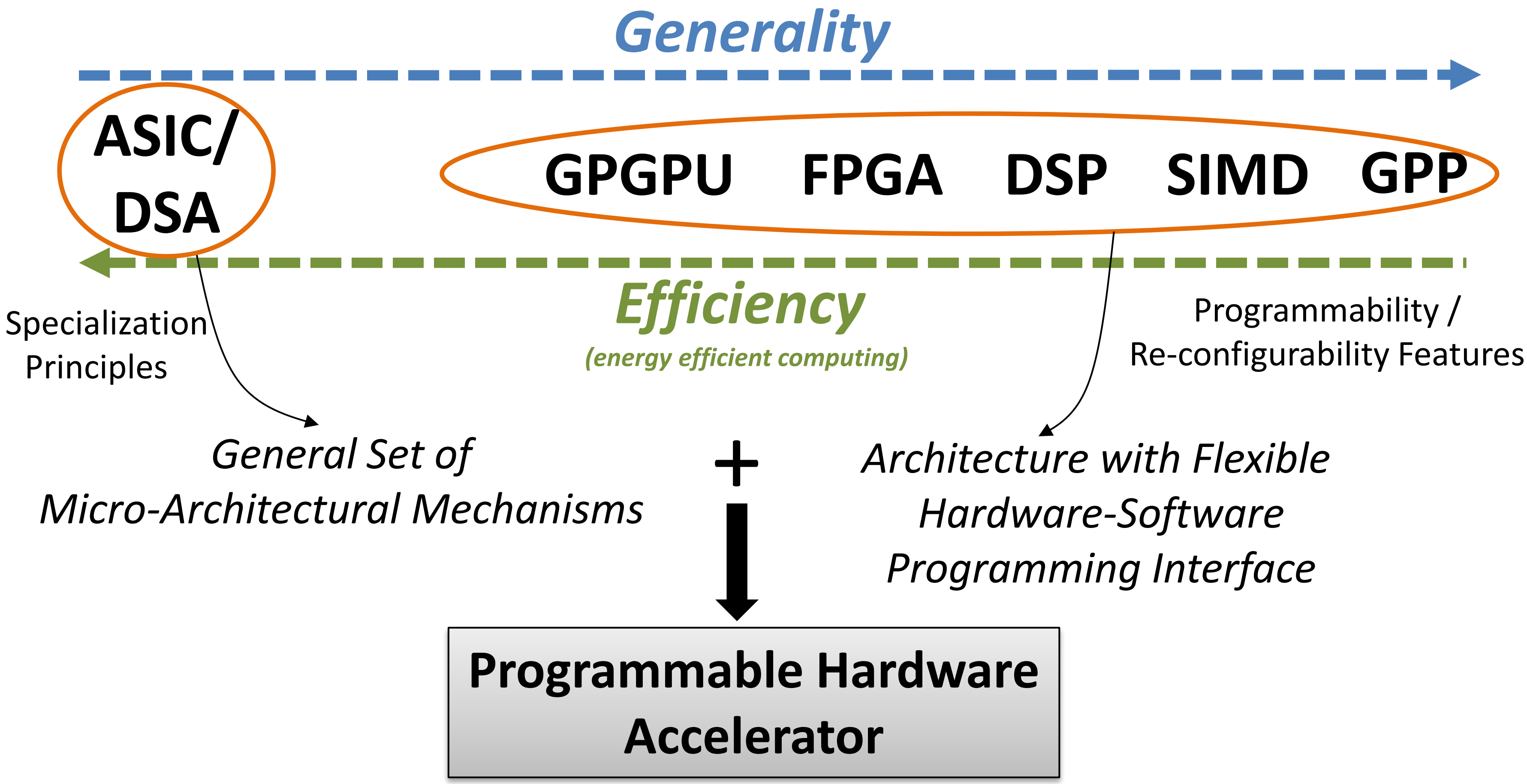
Specialization Principles

Programmability / Re-configurability Features

General Set of Micro-Architectural Mechanisms

Architecture with Flexible Hardware-Software Programming Interface

Specialization Spectrum



UCLA

Specialization Spectrum

Generality

ASIC/ DSA

GPGPU FPGA DSP SIMD GPP

Efficiency

(energy efficient computing)

Specialization Principles

Programmability / Re-configurability Features

General Set of Micro-Architectural Mechanisms

\+

Architecture with Flexible Hardware-Software Programming Interface

Programmable Hardware Accelerator

Efficiency close to DSAs/ASICs

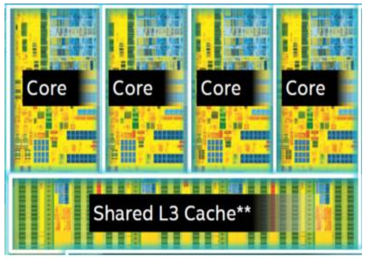
Trivial adaptation of new algorithms/applications

Retain programmability

UCLA

Background Work*

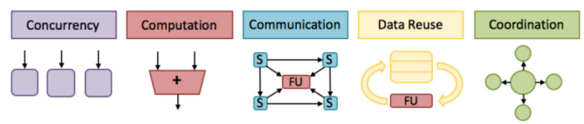
*IEEE Micro Top-Picks 2017: ***Domain Specialization is Generally Unnecessary for Accelerators***



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

Domain-Specific Accelerators (DSAs)

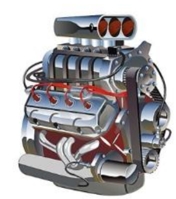
Commonality in DSAs ?



FIVE Specialization Principles

Micro-Architectural Mechanisms

Programmable Hardware Accelerator Architecture



Our Work:
Stream-Dataflow Acceleration

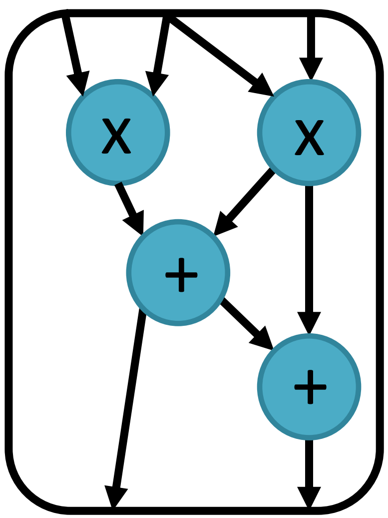
Exploit common accelerator application behavior:

Our Work: Stream-Dataflow Acceleration

Exploit common accelerator application behavior:

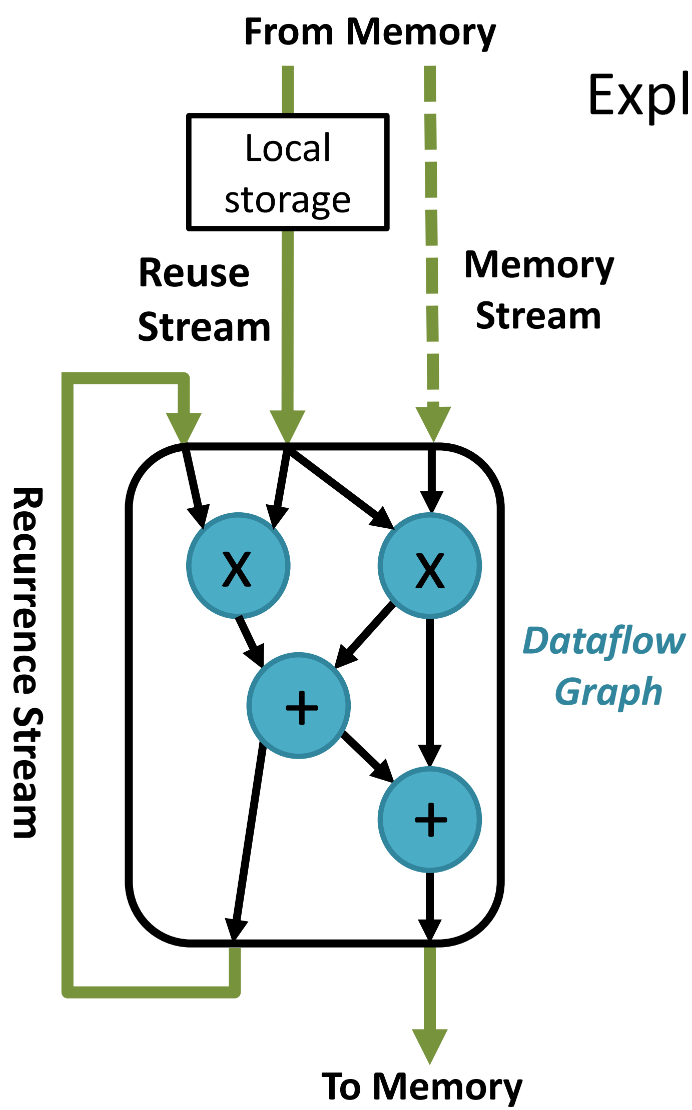
Dataflow Computation

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



Dataflow Graph

Our Work:
Stream-Dataflow Acceleration



Exploit common accelerator application behavior:

Dataflow Computation

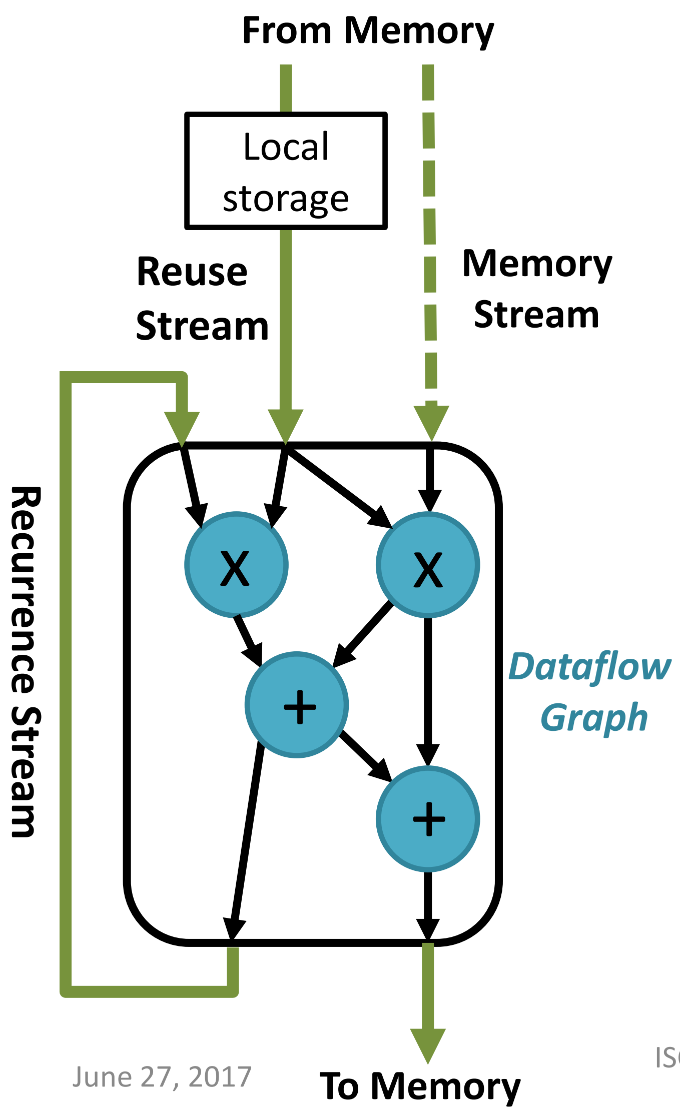
- Stream-Dataflow **Execution model**
  – Abstracts typical accelerator computation phases

Stream Patterns and Interface

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

Stream-Dataflow Acceleration

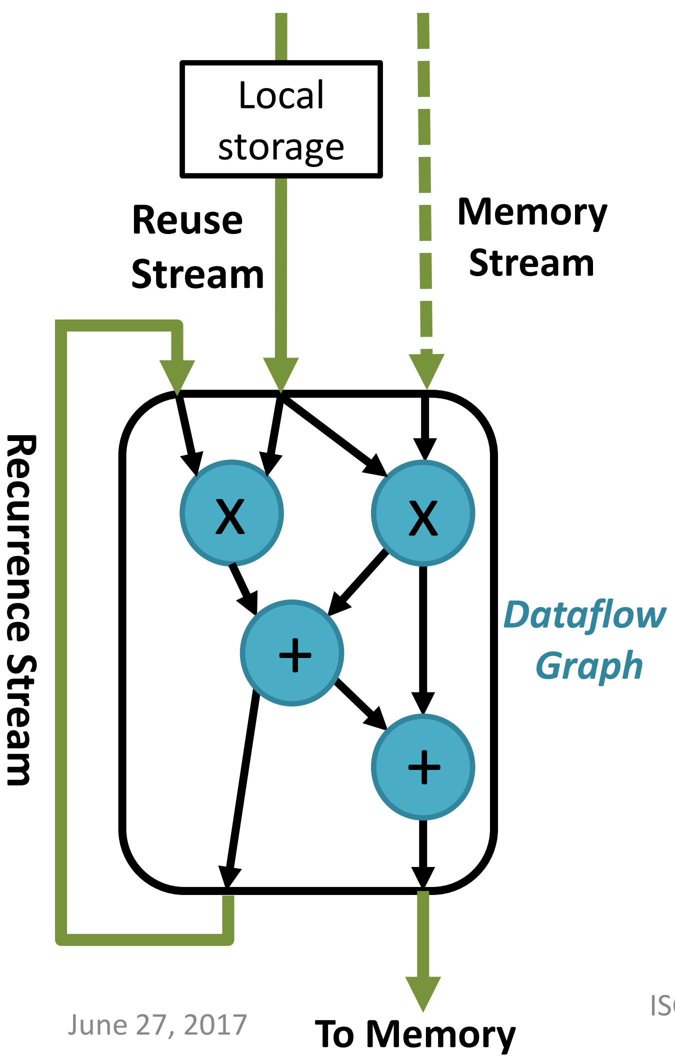
Stream-Dataflow Model



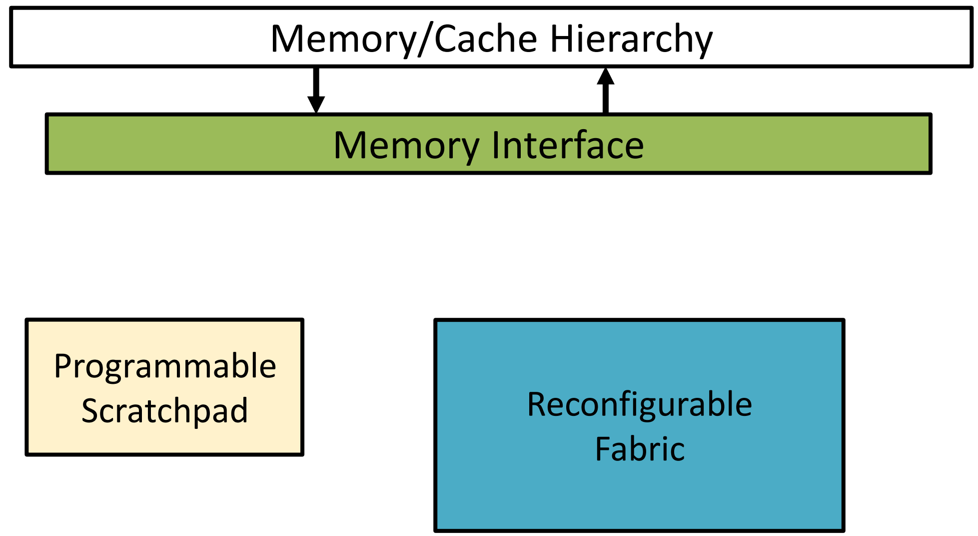
Programmable Stream-Dataflow Accelerator

Stream-Dataflow Acceleration

Stream-Dataflow Model

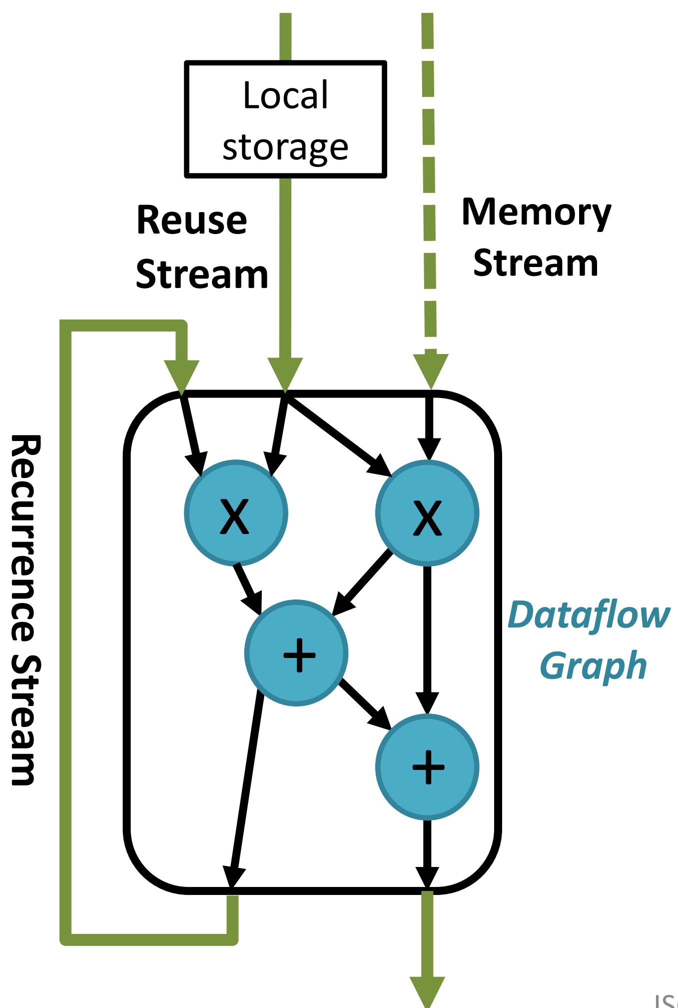


Programmable Stream-Dataflow Accelerator

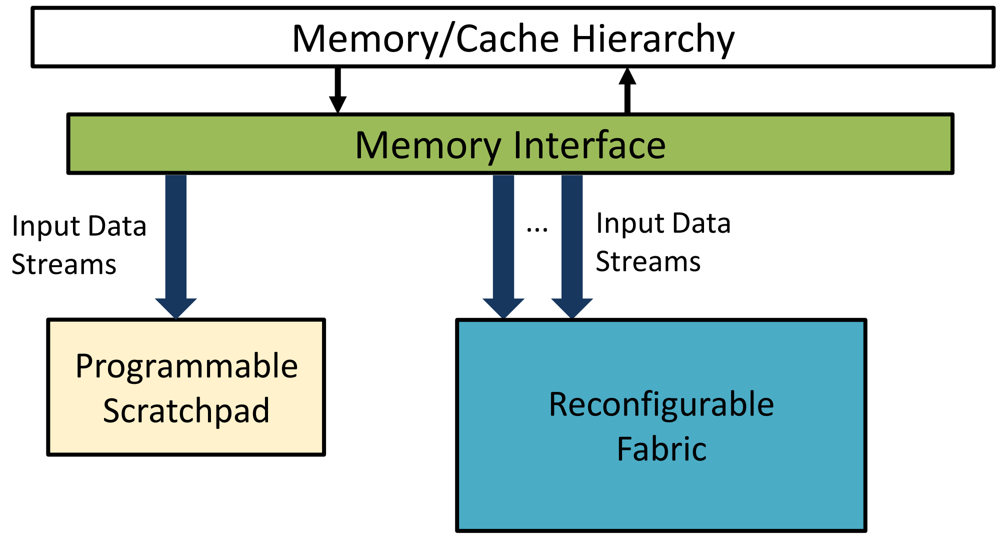


Stream-Dataflow Acceleration

Stream-Dataflow Model



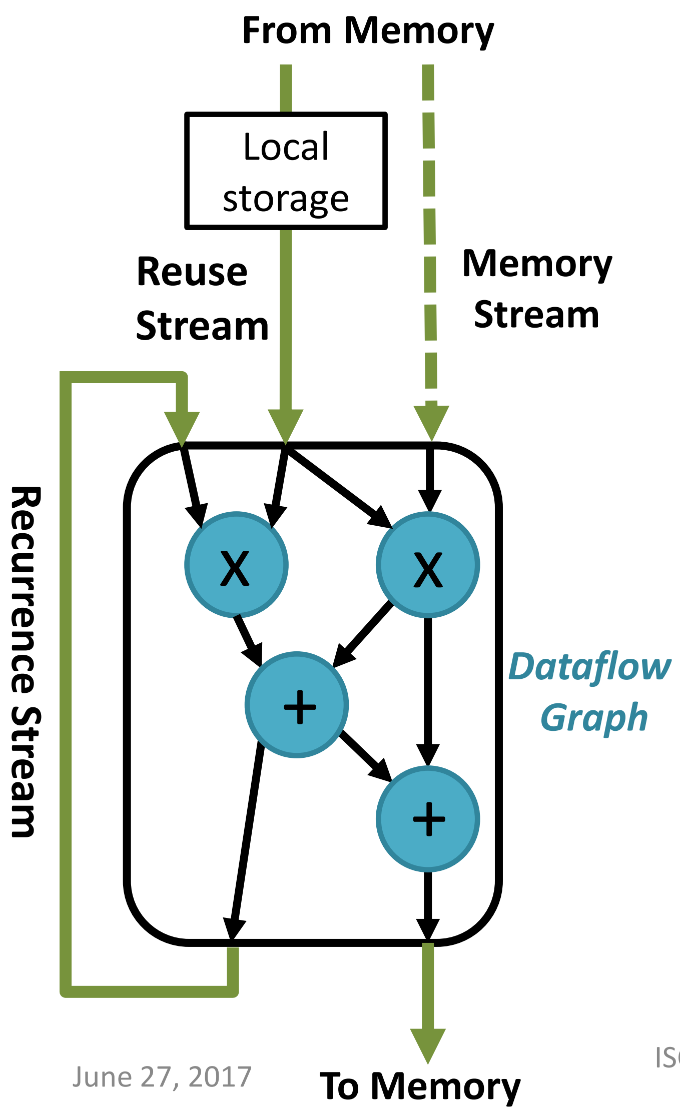
Programmable Stream-Dataflow Accelerator



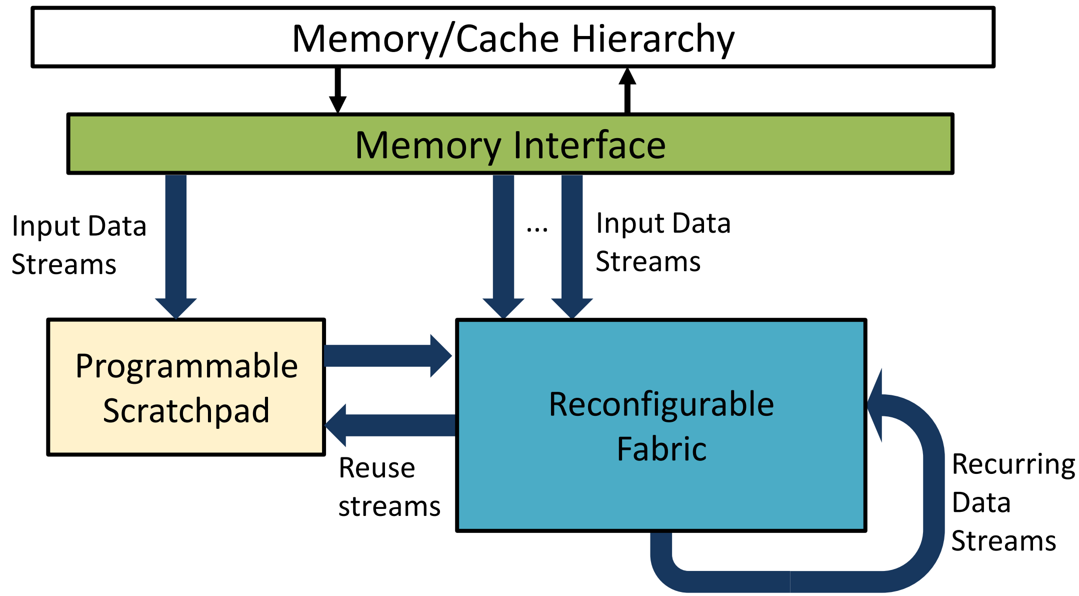
- Data-parallel program kernels streaming data from memory

Stream-Dataflow Acceleration

Stream-Dataflow Model



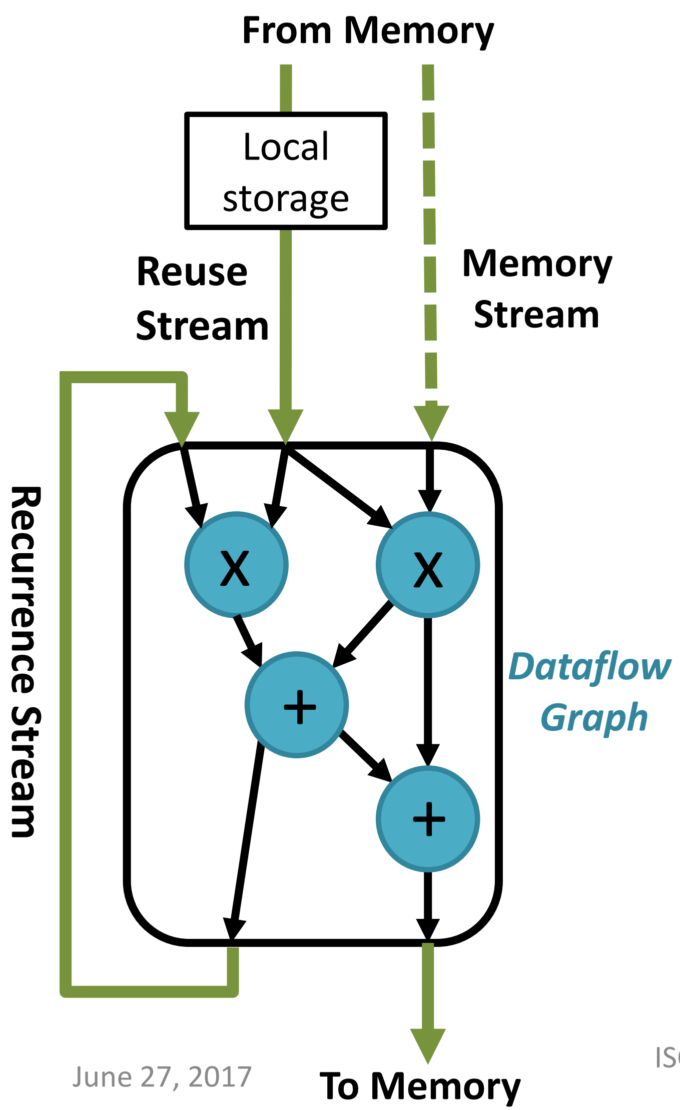
Programmable Stream-Dataflow Accelerator



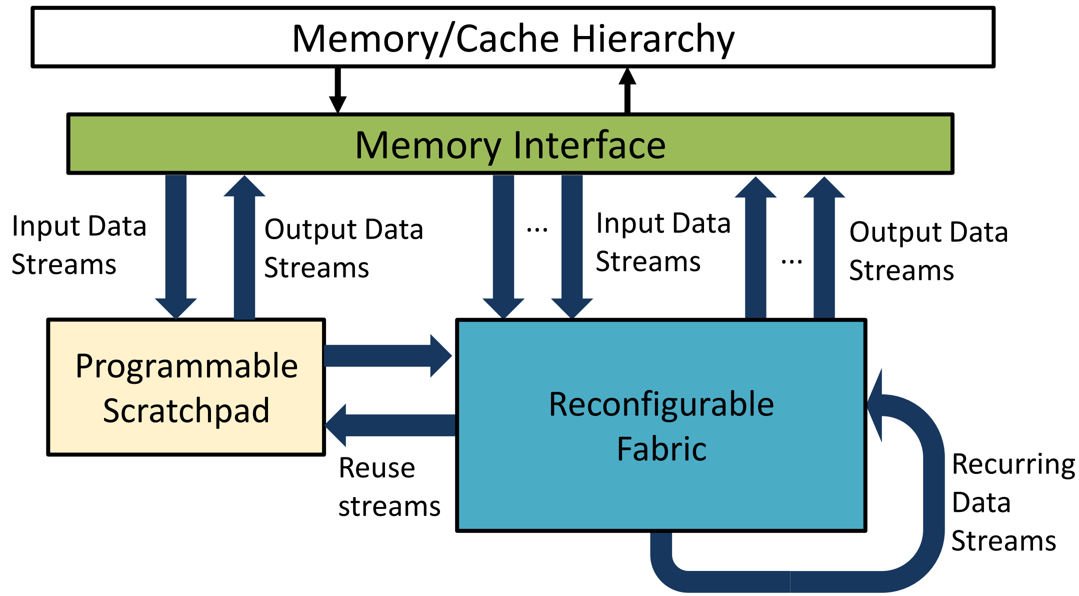
- Data-parallel program kernels streaming data from memory
- Dataflow computation fabric operates on data streams iteratively

Stream-Dataflow Acceleration

Stream-Dataflow Model



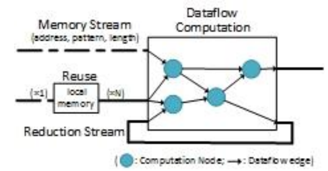
Programmable Stream-Dataflow Accelerator



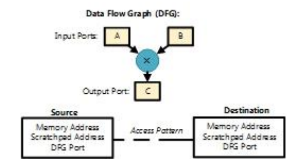
- Data-parallel program kernels streaming data from memory
- Dataflow computation fabric operates on data streams iteratively
- Computed output streams stored back to memory

Outline

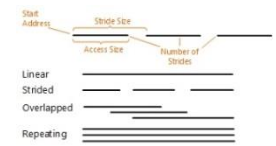
- Motivation and Overview



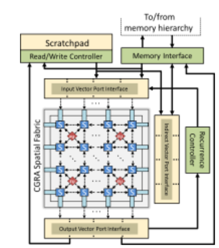
- Stream-Dataflow Execution Model



- Hardware-Software Interface and Example program



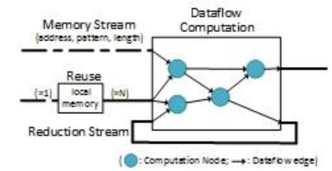
- Stream-Dataflow Accelerator Architecture



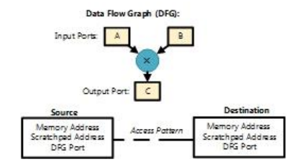
- Evaluation and Results

Outline

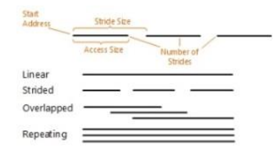
- Motivation and Overview



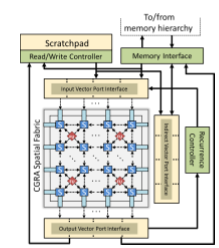
- **Stream-Dataflow Execution Model**



- Hardware-Software Interface and Example program



- Stream-Dataflow Accelerator Architecture



- Evaluation and Results

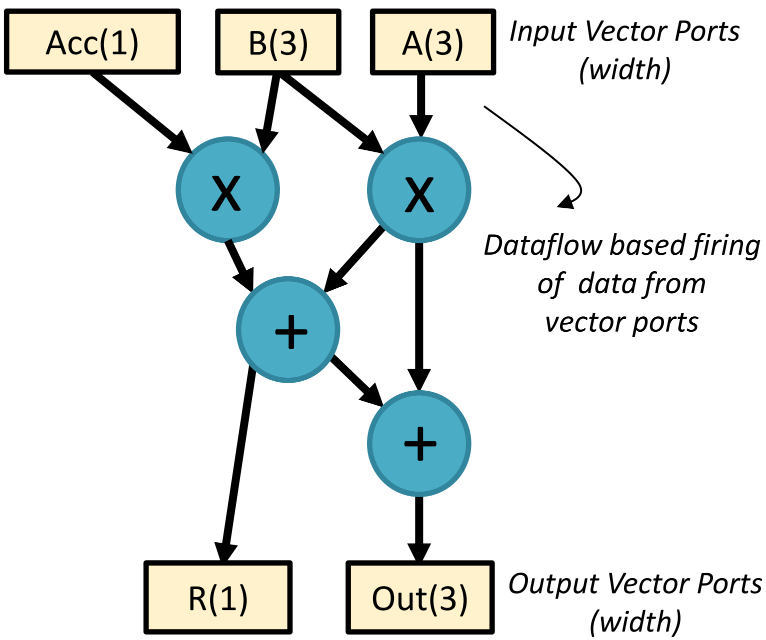
Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

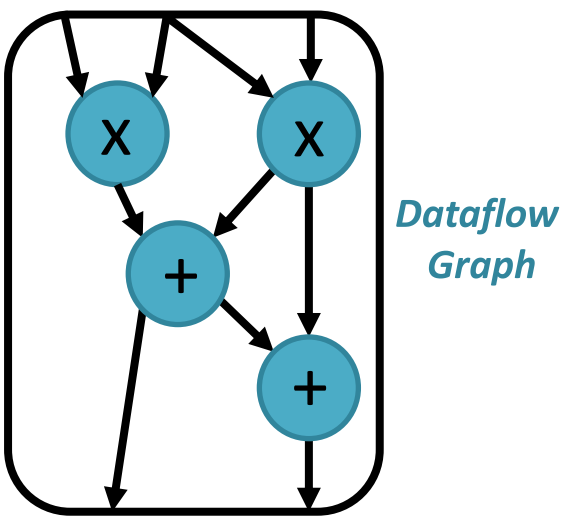
- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports



Stream-Dataflow Execution Model

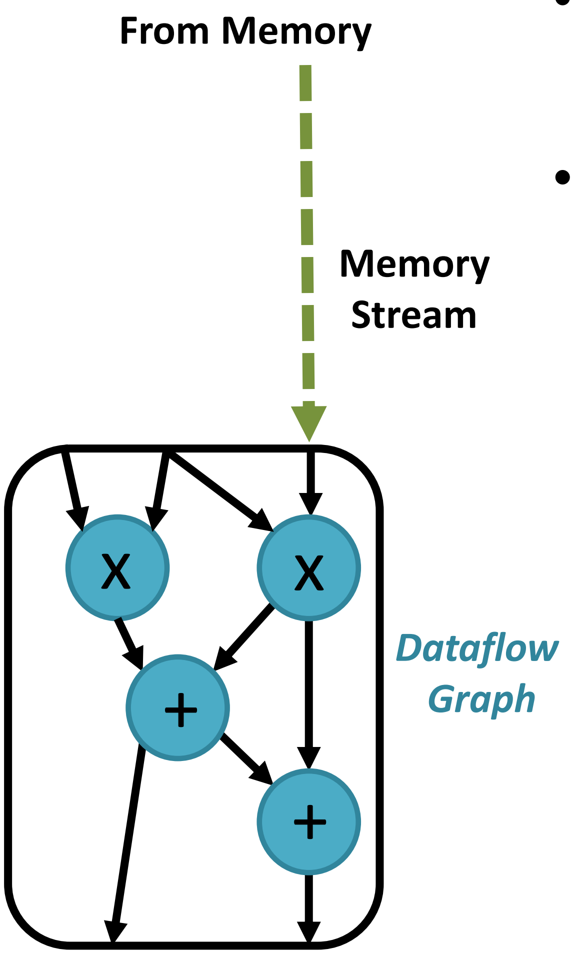
Programmer Abstractions for Stream-Dataflow Model

- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports



Stream-Dataflow Execution Model

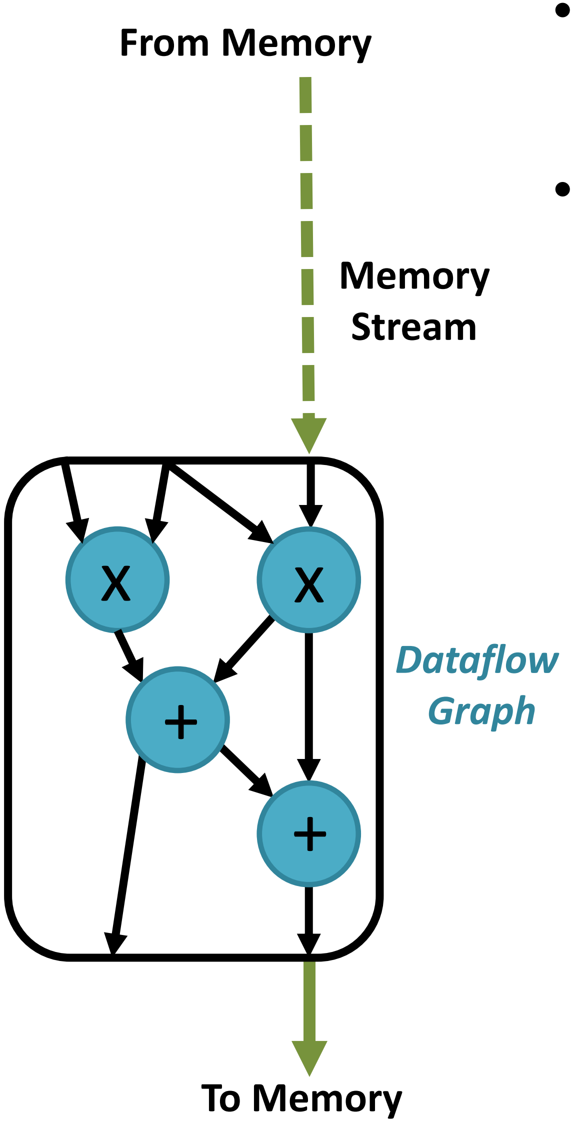
Programmer Abstractions for Stream-Dataflow Model



- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory

Stream-Dataflow Execution Model

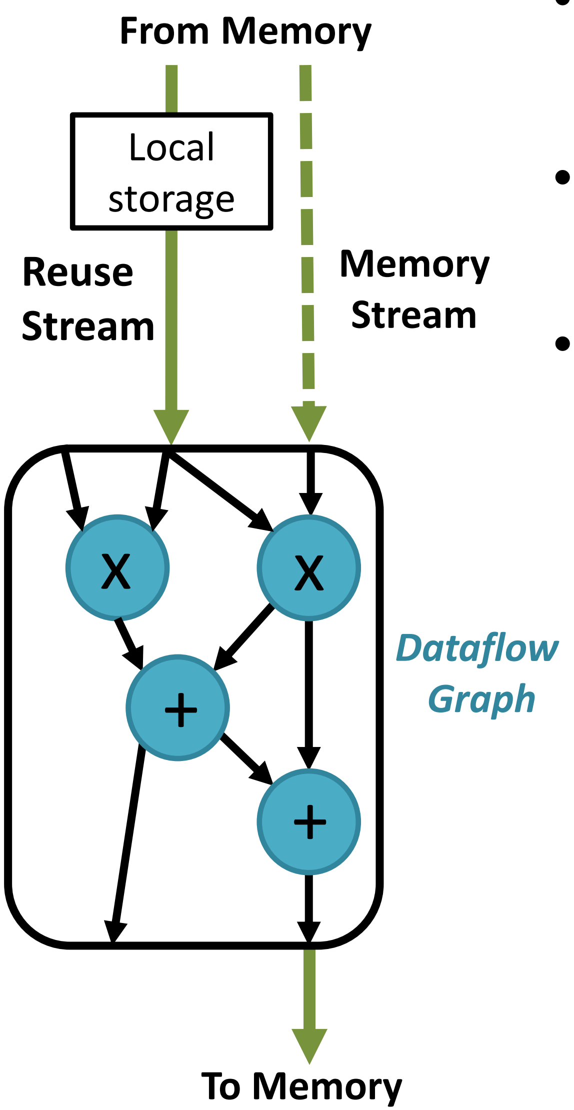
Programmer Abstractions for Stream-Dataflow Model



- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory

Stream-Dataflow Execution Model

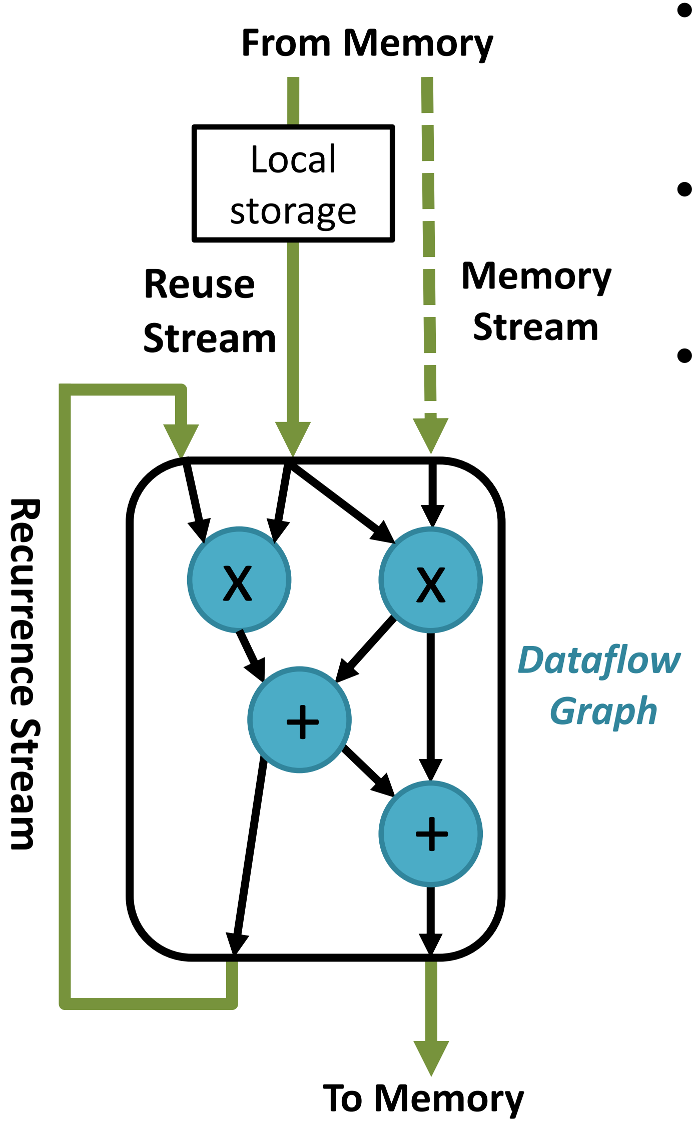
Programmer Abstractions for Stream-Dataflow Model



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

Stream-Dataflow Execution Model

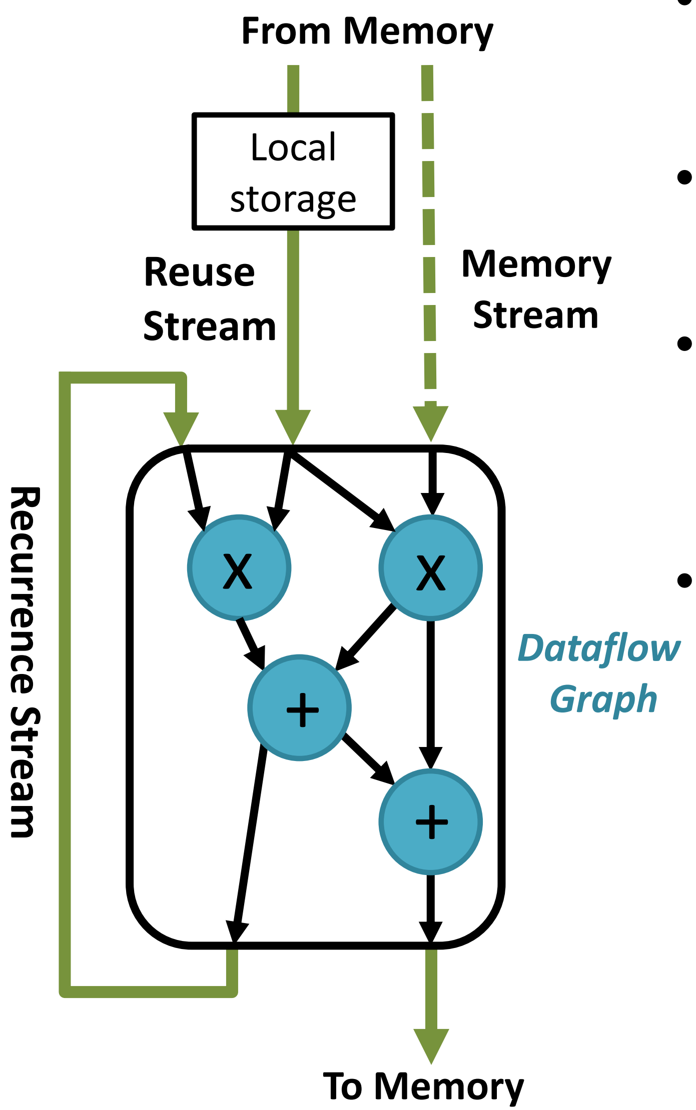
Programmer Abstractions for Stream-Dataflow Model



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

Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

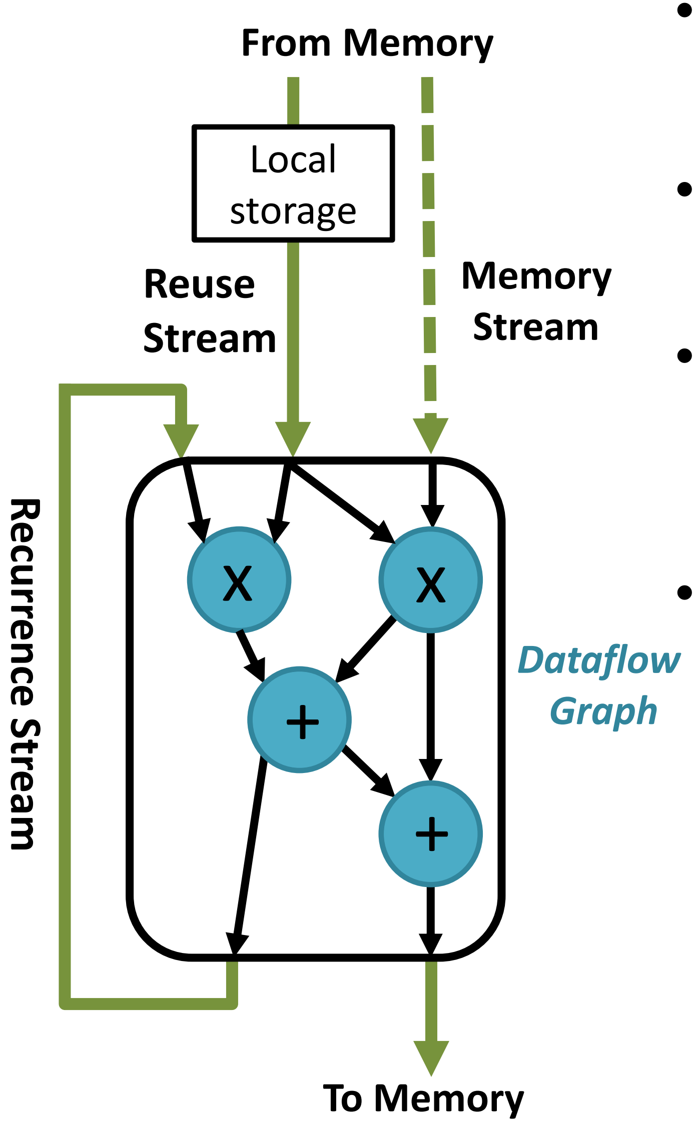


- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

Source		***Destination***
Memory Address Local Storage Address DFG Port	*Access Pattern*	Memory Address Local Storage Address DFG Port

Stream-Dataflow Execution Model

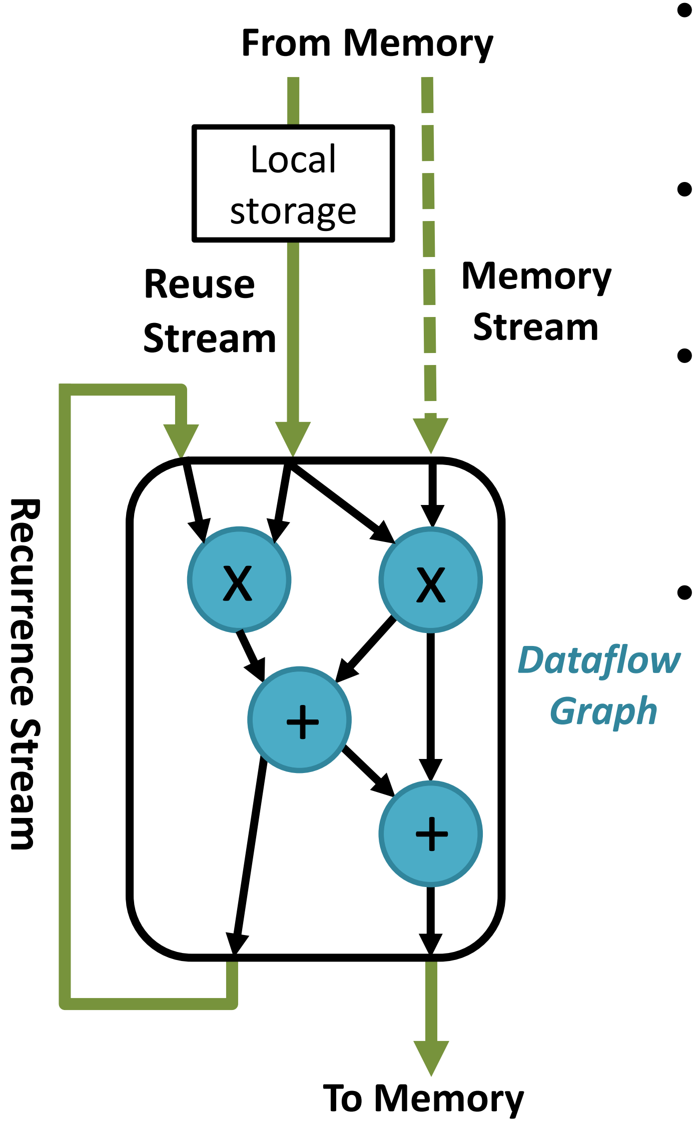
Programmer Abstractions for Stream-Dataflow Model



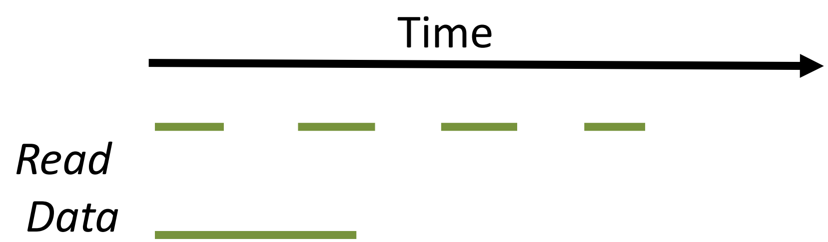
- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

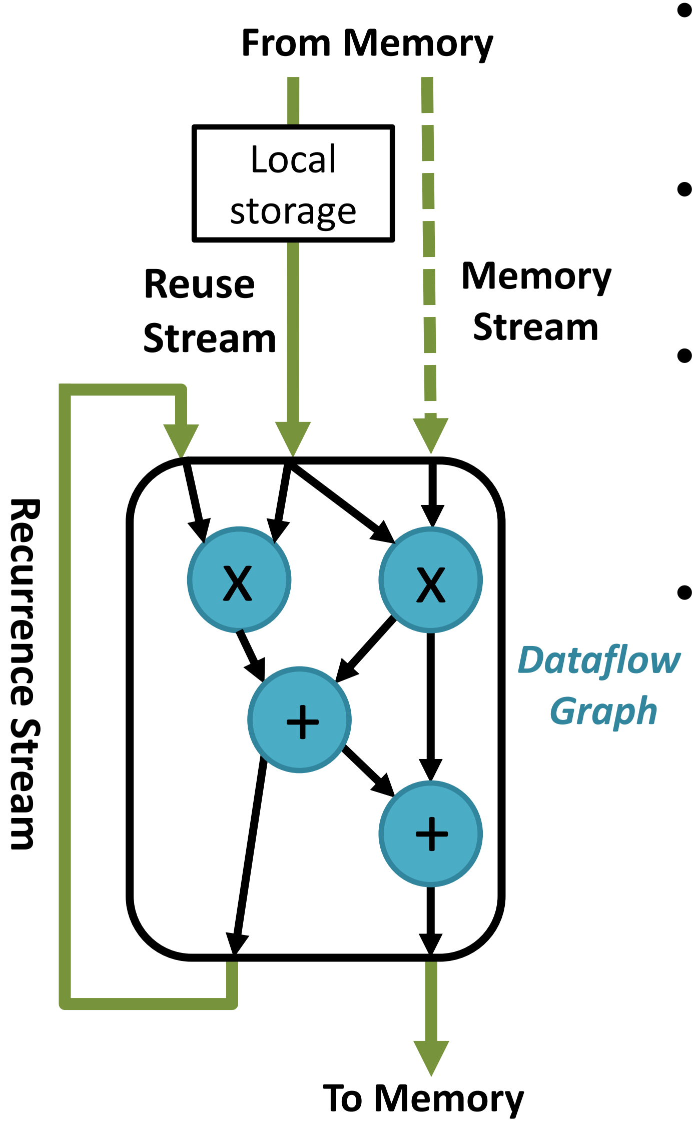


- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

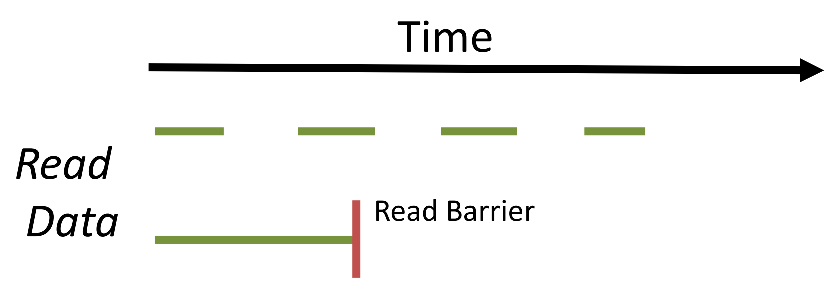


Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

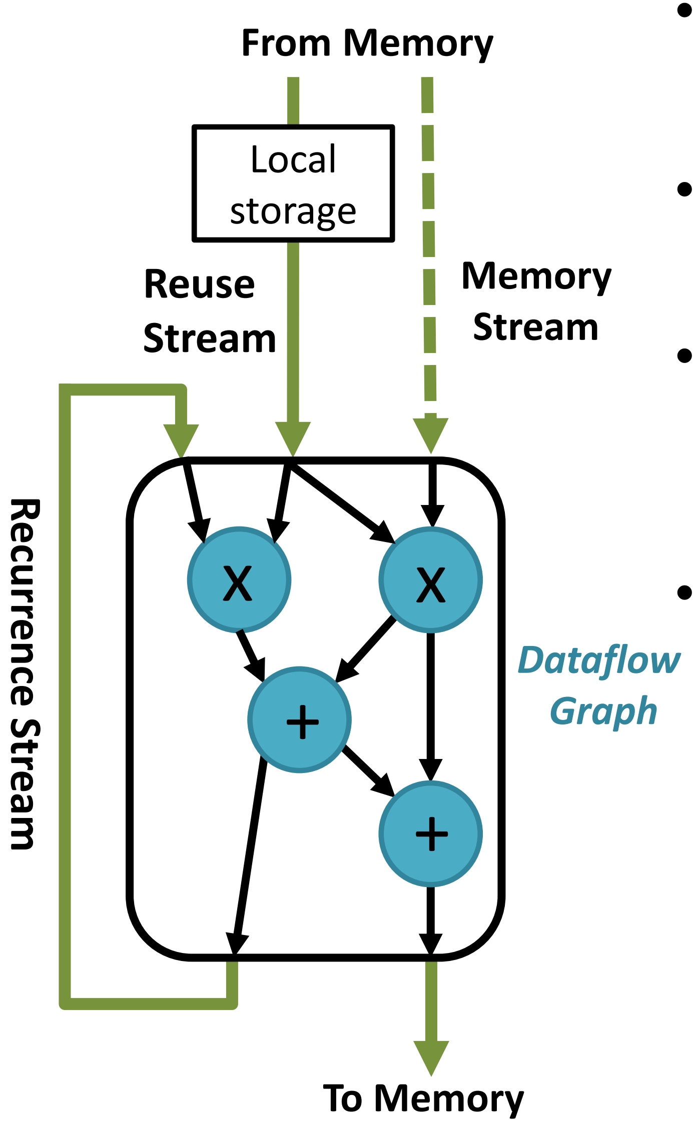


- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

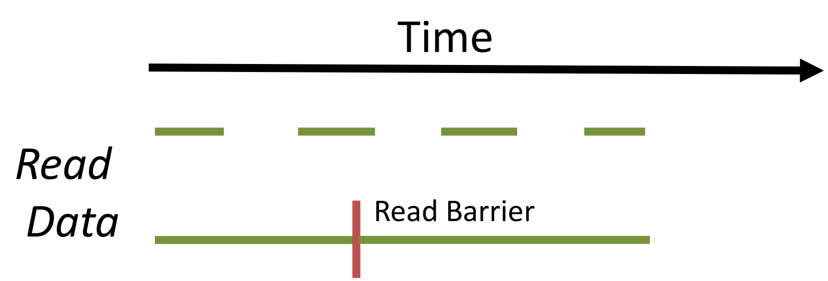


Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

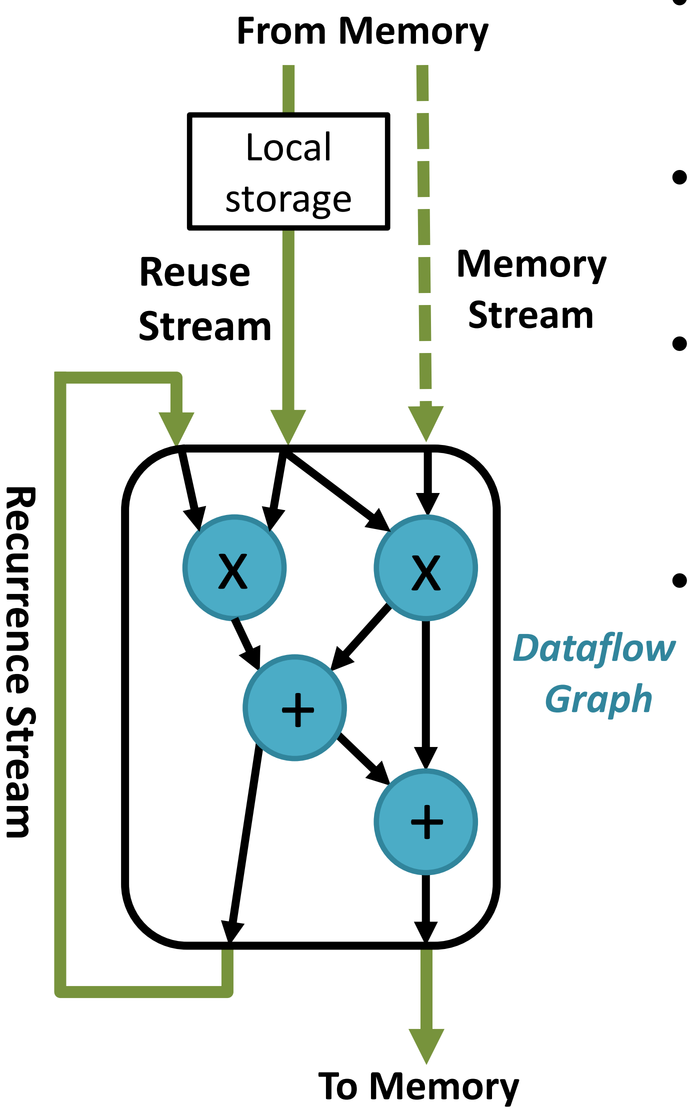


- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

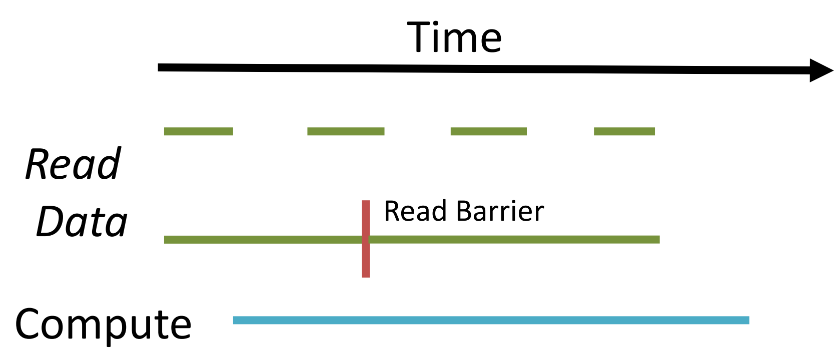


Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

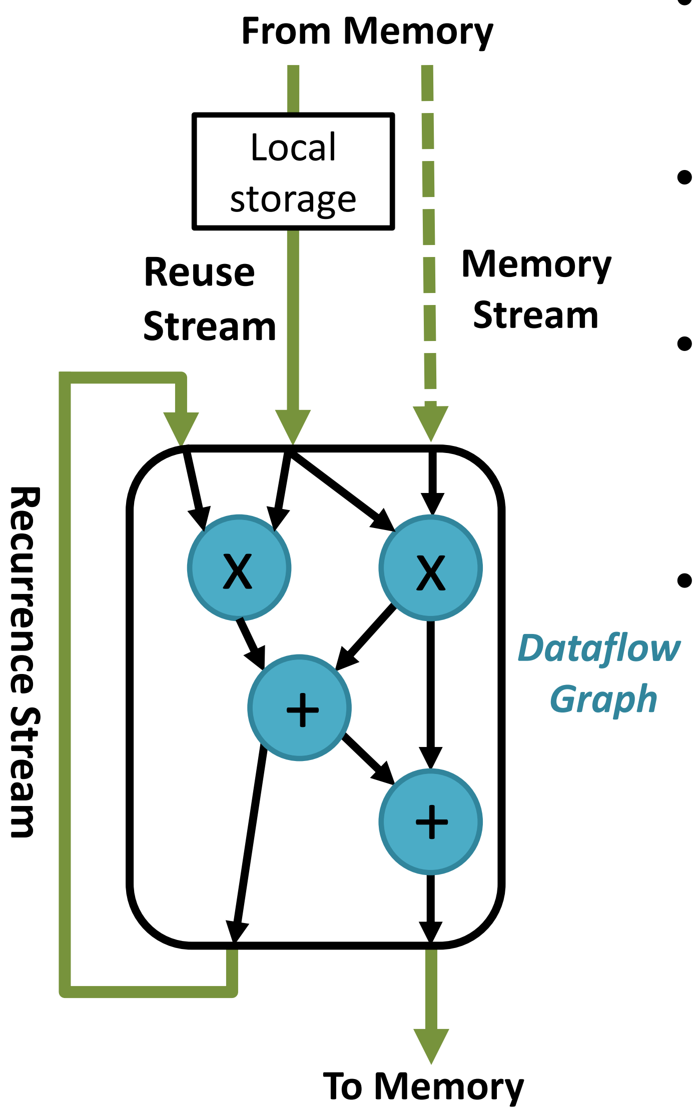


- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

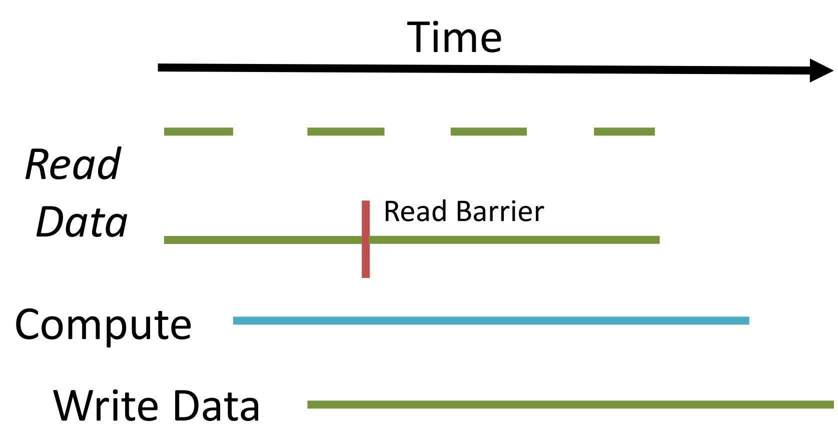


Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

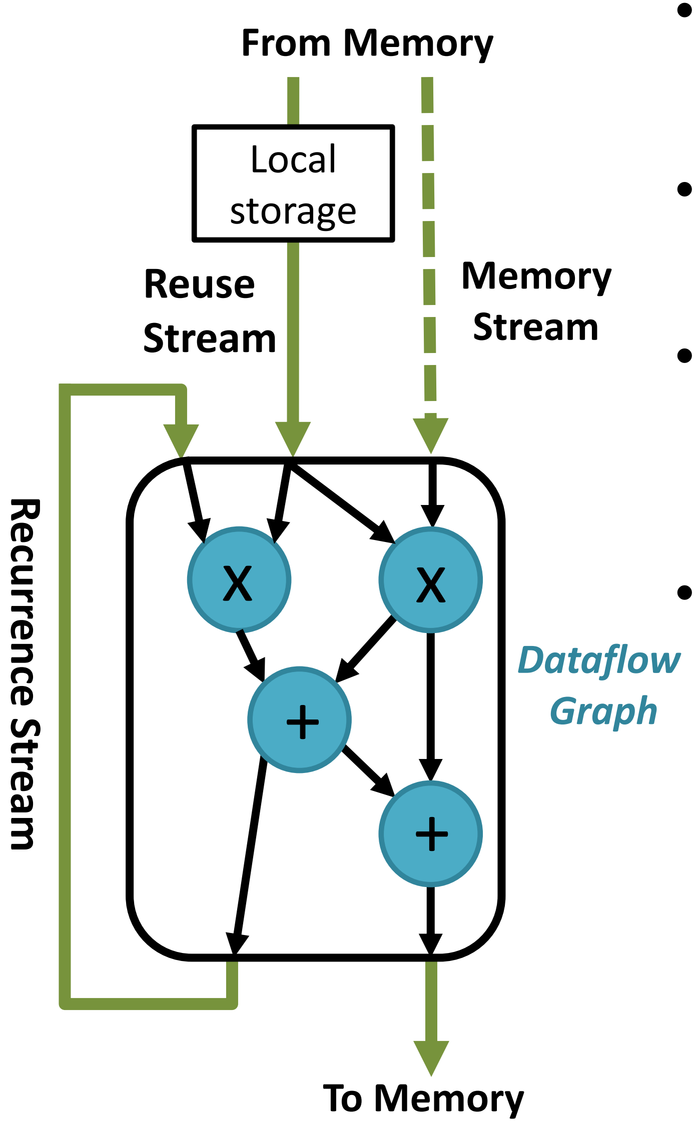


- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

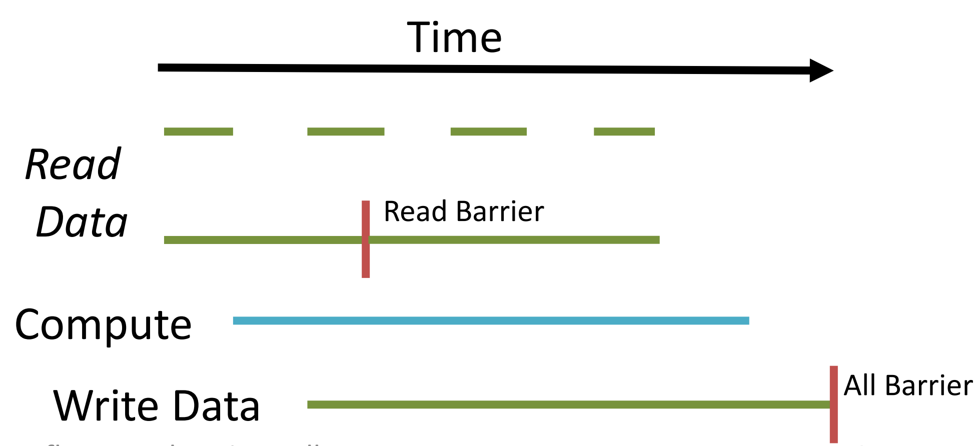


Stream-Dataflow Execution Model

Programmer Abstractions for Stream-Dataflow Model

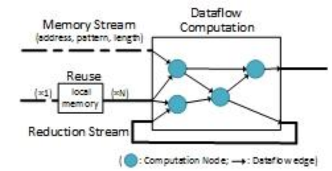


- ***Computation abstraction*** – Dataflow Graph (DFG) with input/output vector ports
- ***Data abstraction*** – Streams of data fetched from memory and stored back to memory
- ***Reuse abstraction*** – Streams of data fetched once from memory, stored in local storage (programmable scratchpad) and reused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

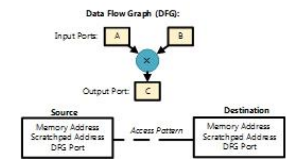


Outline

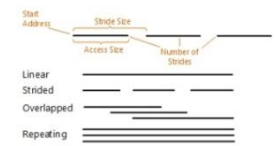
- Motivation and Overview



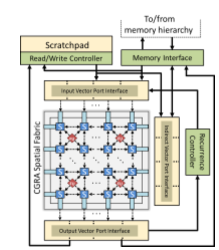
- Stream-Dataflow Execution Model



- Hardware-Software Interface and Example program



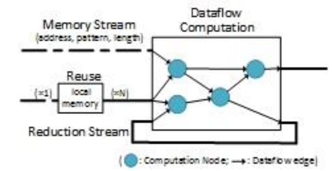
- Stream-Dataflow Accelerator Architecture



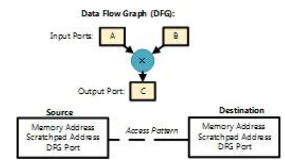
- Evaluation and Results

Outline

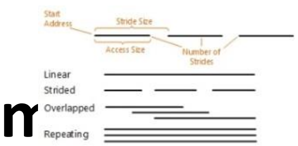
- Motivation and Overview



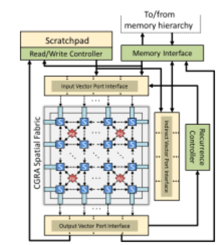
- Stream-Dataflow Execution Model



- **Hardware-Software Interface and Example program**



- Stream-Dataflow Accelerator Architecture



- Evaluation and Results

Stream-Dataflow ISA Interface

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

Stream-Dataflow ISA

Stream-Dataflow ISA

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

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-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 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*

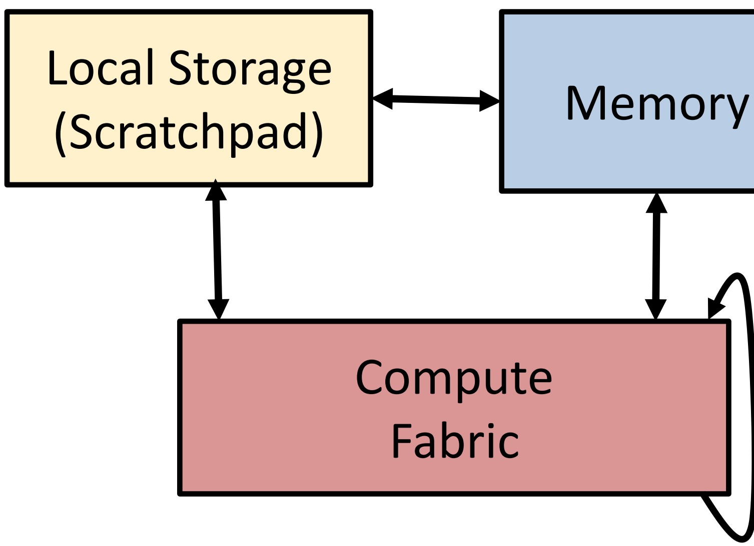
Command Name	Parameters	Description
SD_Config	Address, Size	Stream CGRA configuration from given address
SD_Mem_Scratch	Source Mem Address, Stride, Access Size, Num Strides, Dest. Scratch Address	Read from memory with pattern to scratchpad
SD_Scratch_Port	Source Scratch Address, Stride, Access Size, Strides, Input Port #	Read from scratchpad with pattern to input port
SD_Mem_Port	Source Mem Address, Stride, Access Size, Num Strides, Input Port #	Read from memory with pattern to input port
SD_Const_Port	Constant Value, Num Elements, Input Port #	Send constant value to input port
SD_Clean_Port	Num Elements, Output Port #	Throw away some elements from output port
SD_Port_Port	Output Port #, Num Elements, Input Port #	Issue recurrence between input-output port pairs
SD_Port_Scratch	Output Port #, Num Elements, Scratch Address	Write from port to scratchpad
SD_Port_Mem	Output Port #, Stride, Access Size, Num Strides, Dest. Mem Address	Write from port to memory with pattern
SD_Mem_IndPort	Source Mem Address, Stride, Access Size, Num Strides, Indirect Port #	Read the addresses from memory with pattern to indirect port
SD_IndPort_Port	Indirect Port #, Offset Address, Input Port #	Indirect load from addresses present in indirect port
SD_IndPort_Mem	Indirect Port #, Output Port #, Dest. Offset Address	Indirect store to addresses present in indirect port
SD_Barrier_Scratch_Rd	-	Barrier for scratchpad reads
SD_Barrier_Scratch_Wr	-	Barrier for scratchpad writes
SD_Barrier_All	-	Barrier to wait for all commands completion

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*

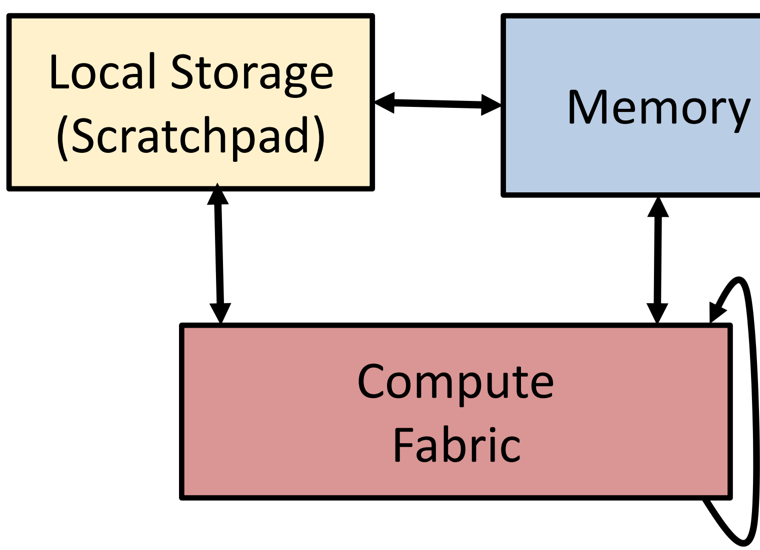


Stream-Dataflow ISA

- 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*



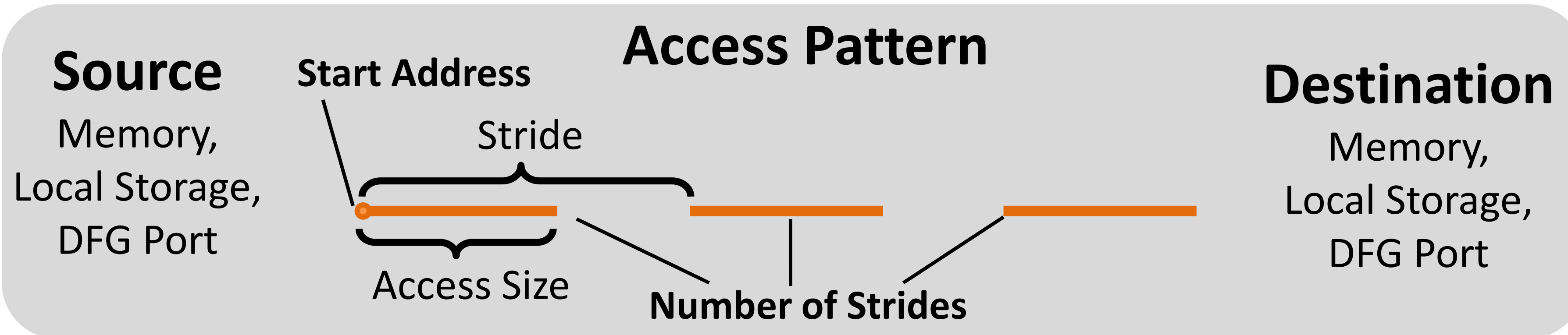
Stream-Dataflow Hardware-Software Interface

Source
Memory,
Local Storage,
DFG Port

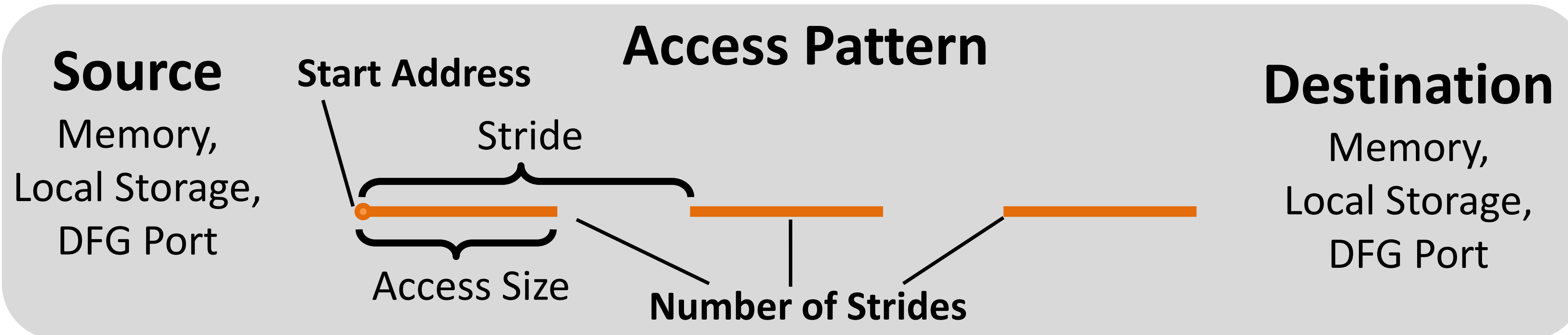
Access Pattern

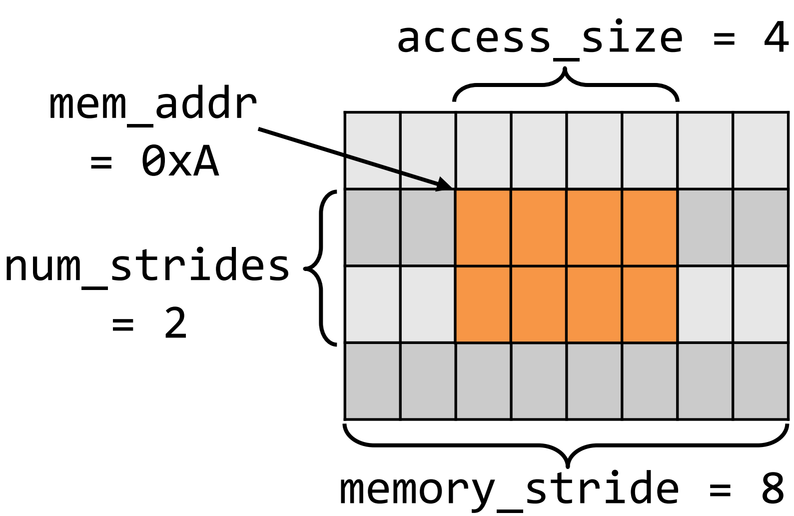
Destination
Memory,
Local Storage,
DFG Port

Stream-Dataflow Hardware-Software Interface

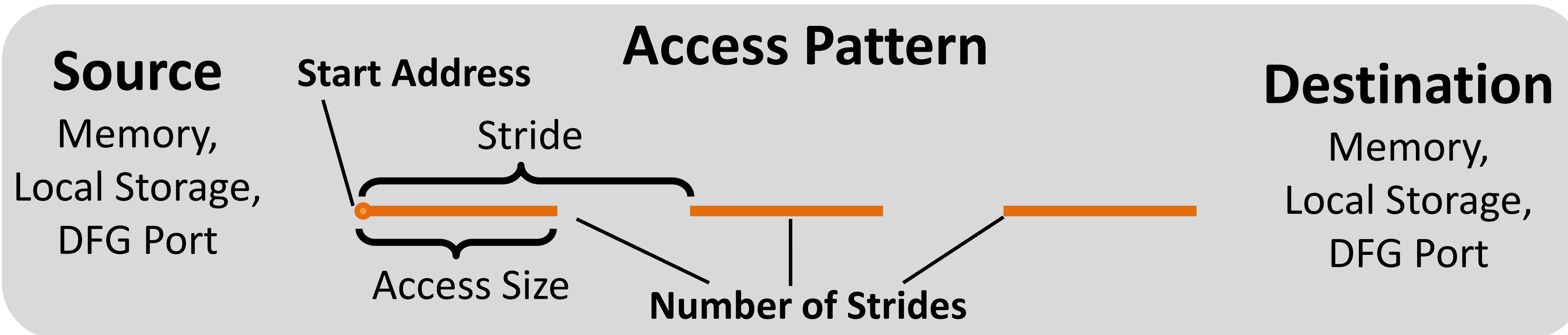


Stream-Dataflow Hardware-Software Interface





Stream-Dataflow Hardware-Software Interface



Example Access Patterns

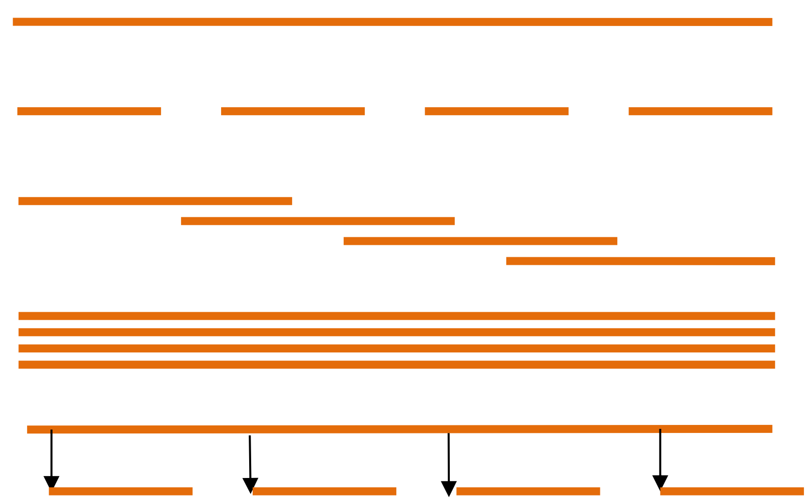
Linear

Strided

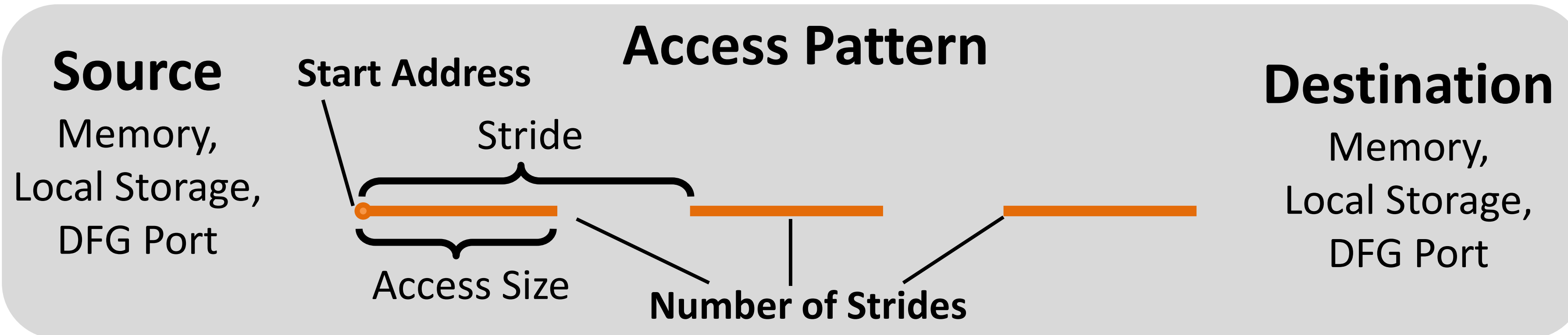
Overlapped

Repeating

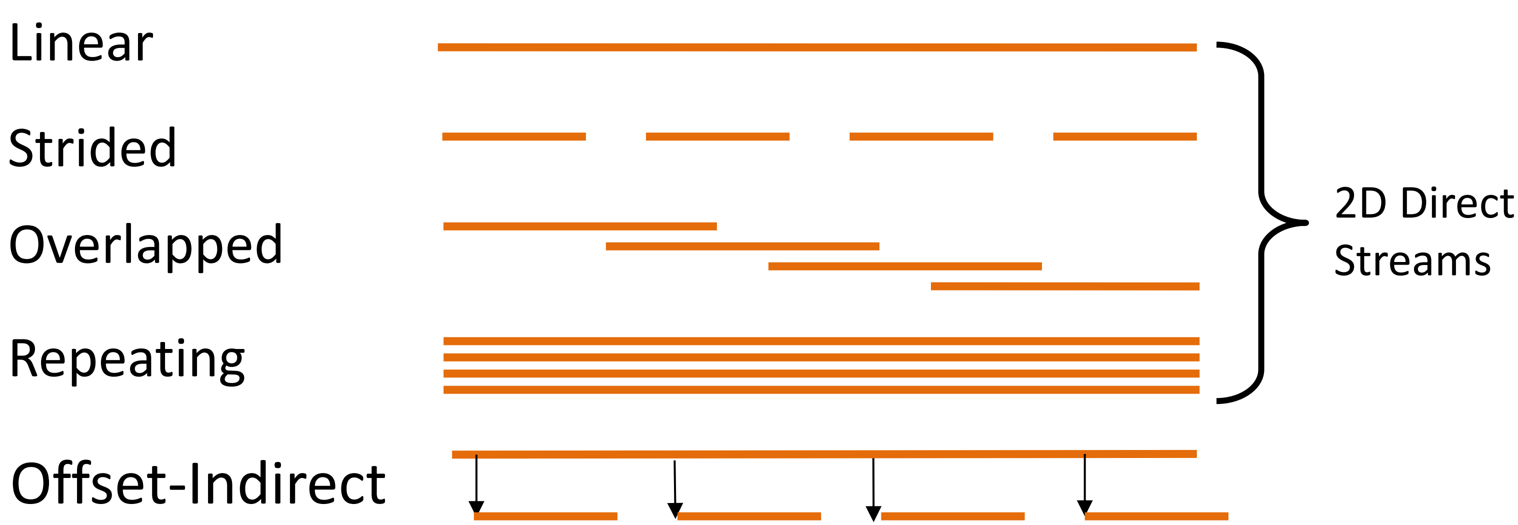
Offset-Indirect



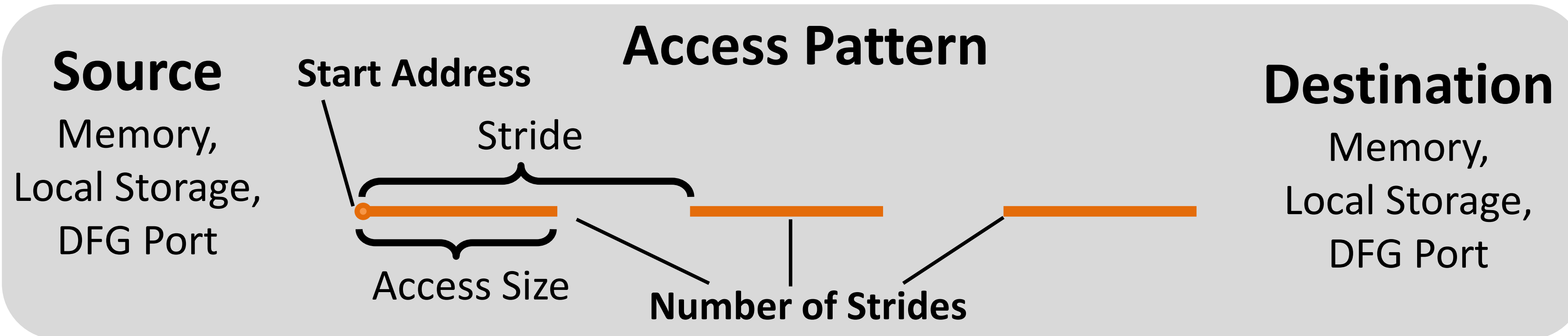
Stream-Dataflow Hardware-Software Interface

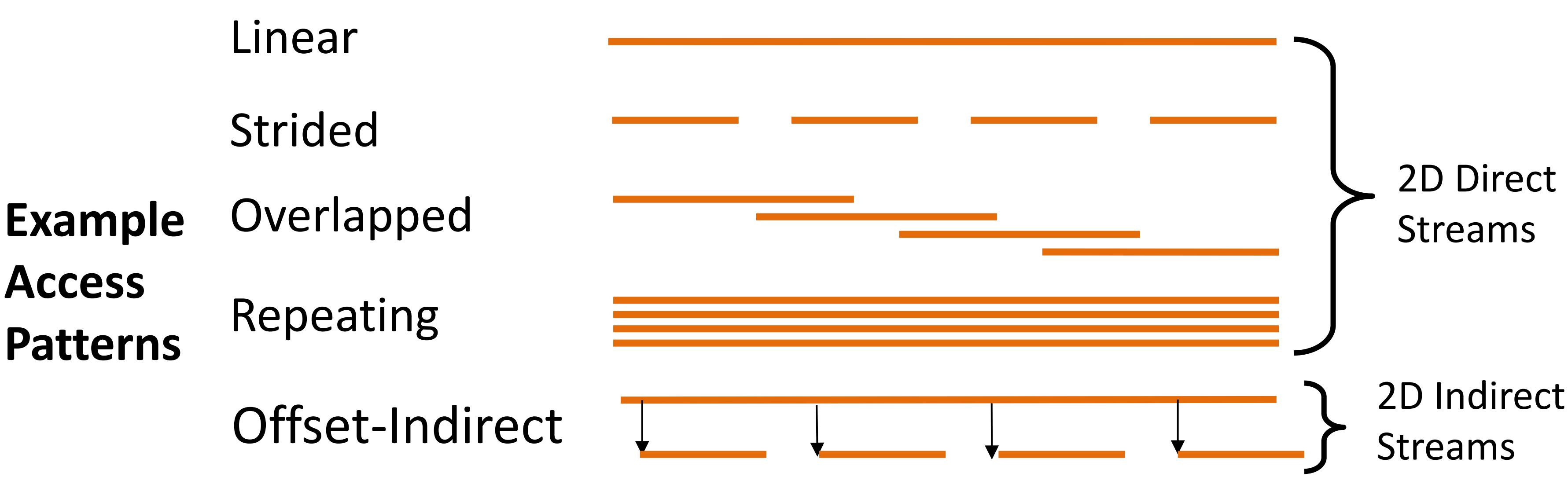


Example Access Patterns



Stream-Dataflow Hardware-Software Interface





Stream-Dataflow ISA Encoding

Stream:

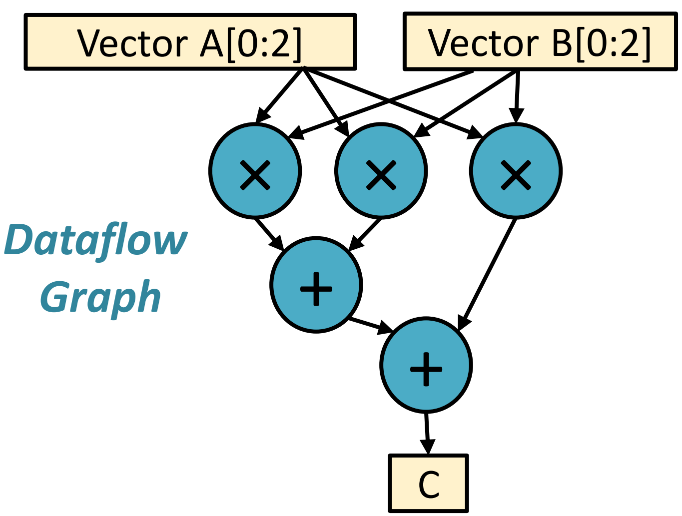
Dataflow:

Stream-Dataflow ISA Encoding

Stream:

Dataflow:

Dataflow Graph



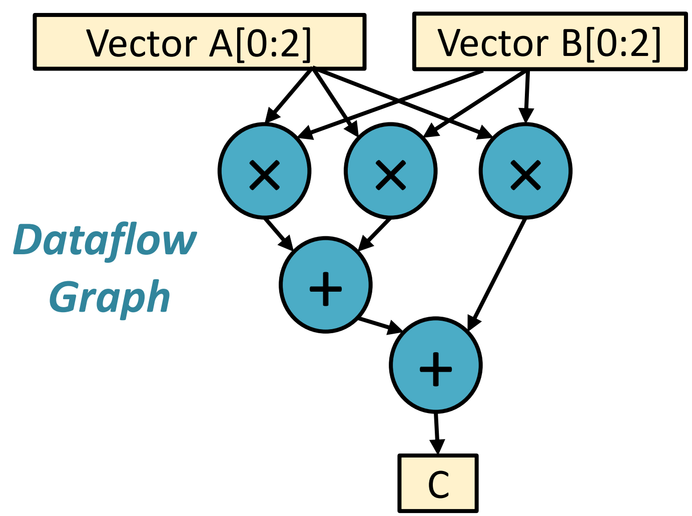
Specified in a Domain Specific Language (DSL)

Stream-Dataflow ISA Encoding

Stream:

Stream Encoding
<address, access_size, stride_size, length>

Dataflow:



Specified in a Domain Specific Language (DSL)

Stream-Dataflow ISA Encoding

Stream:

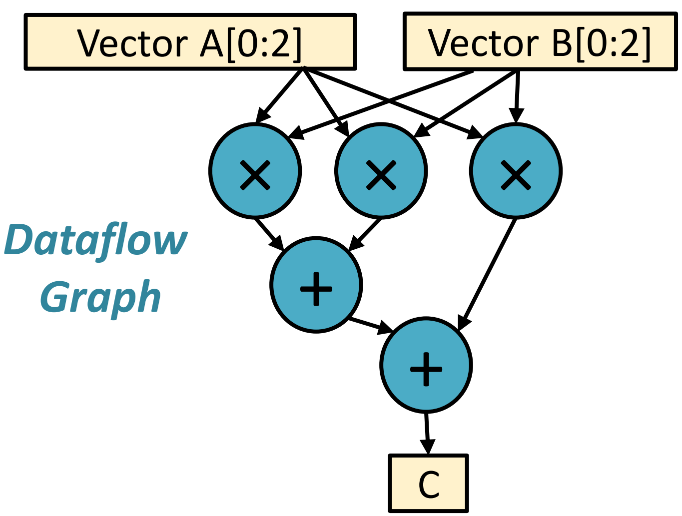
```
for i = 1 to 100:
  ... = a[2*i];
```

Time

Stream Encoding

<address, access_size, stride_size, length>

Dataflow:



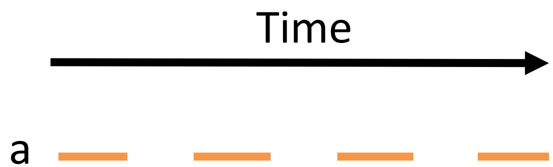
Dataflow Graph

Specified in a Domain Specific Language (DSL)

Stream-Dataflow ISA Encoding

Stream:

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

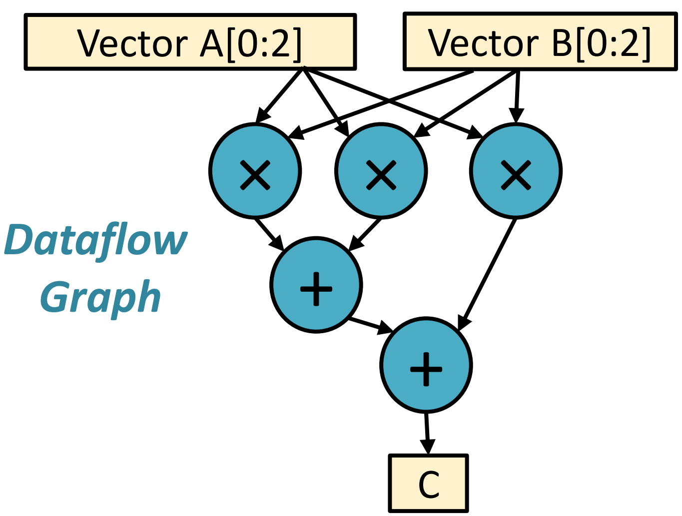


Stream Encoding

<address, access_size, stride_size, length>

Eg: <a, 1, 2, 100>

Dataflow:



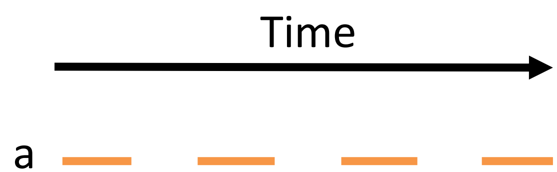
Dataflow Graph

Specified in a Domain Specific Language (DSL)

Stream-Dataflow ISA Encoding

Stream:

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

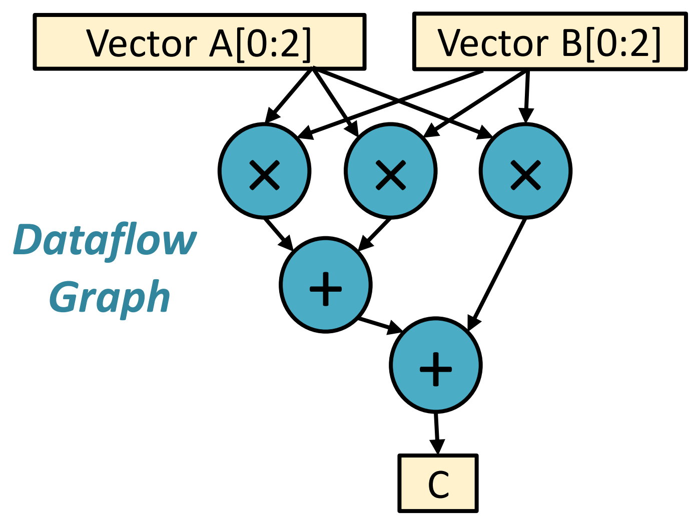


Stream Encoding

<address, access_size, stride_size, length>

Eg: <a, 1, 2, 100>

Dataflow:

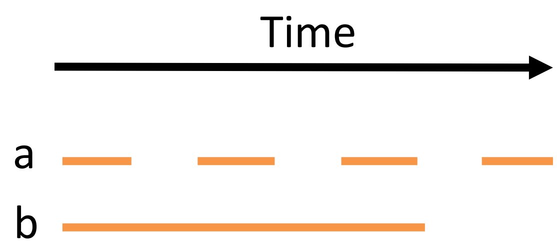


Specified in a Domain Specific Language (DSL)

Stream-Dataflow ISA Encoding

Stream:

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



Stream Encoding

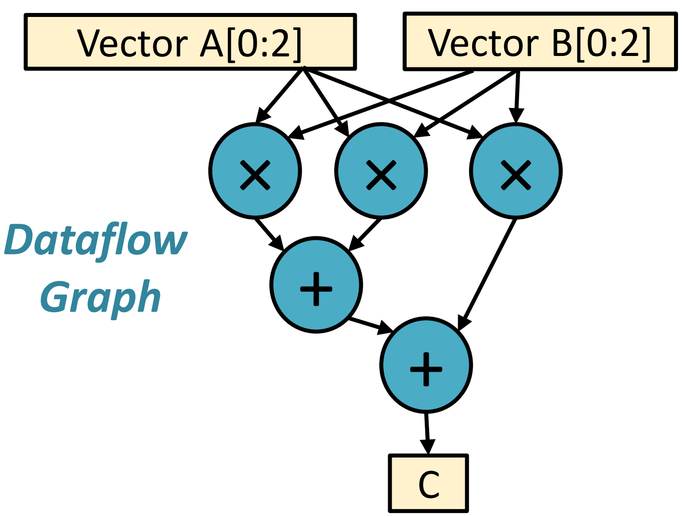
<address, access_size, stride_size, length>

Eg: <a, 1, 2, 100>

<b, 1, 1, 100>

Dataflow:

Dataflow Graph

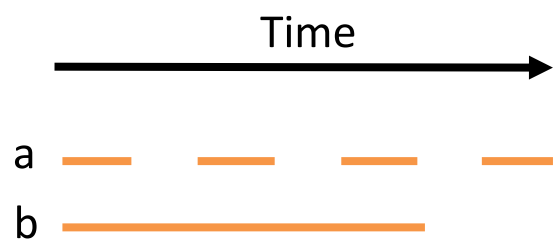


Specified in a Domain Specific Language (DSL)

Stream-Dataflow ISA Encoding

Stream:

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

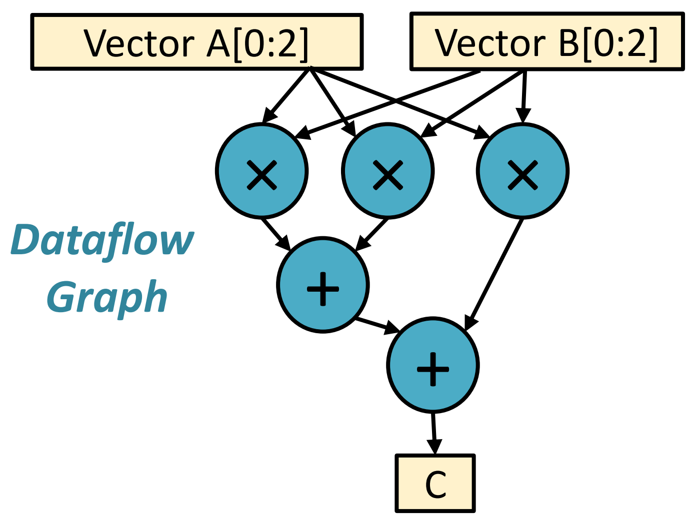


Stream Encoding
<address, access_size, stride_size, length>

Eg: <a, 1, 2, 100>

<b, 1, 1, 100>

Dataflow:



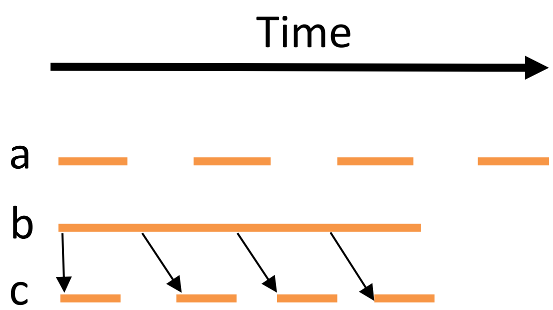
Specified in a Domain Specific Language (DSL)

Stream-Dataflow ISA Encoding

UCLA

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>

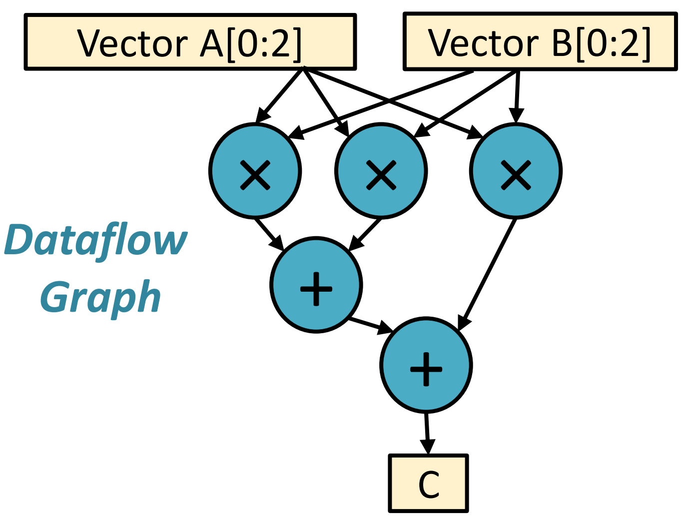
<b, 1, 1, 100>

<stream, start, offset_address>

IND<[prev], c, 100>

Dataflow:

Dataflow Graph



Specified in a Domain Specific Language (DSL)

Example Code: Dot Product

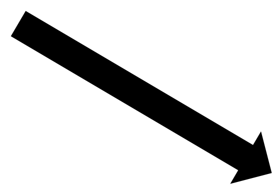
Original Program

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

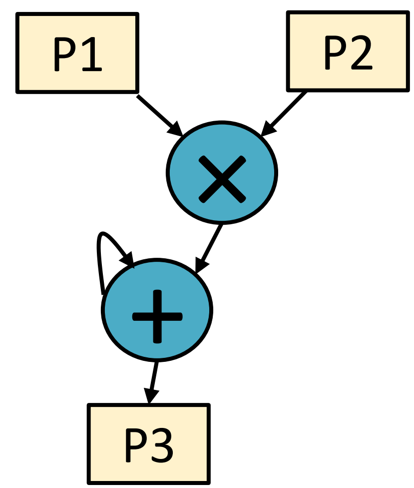
Example Code: Dot Product

Original Program

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



Dataflow Encoding



Example Code: Dot Product

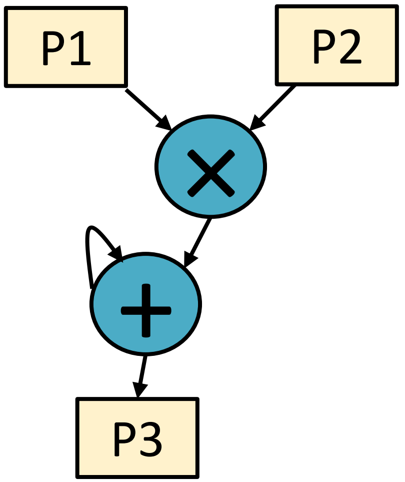
Original Program

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

Stream ISA Encoding

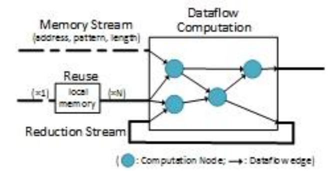
```
Send a[0: N] → P1
Send b[0: N] → P2
Get P3 → c
```

Dataflow Encoding

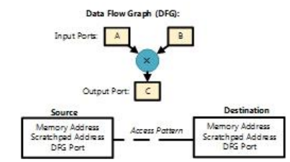


Outline

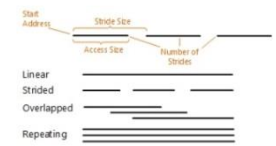
- Motivation and Overview



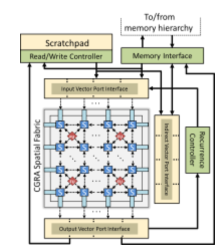
- Stream-Dataflow Execution Model



- Hardware-Software Interface and Example program



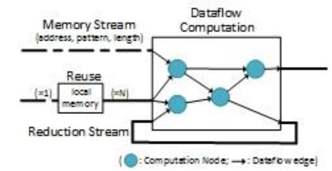
- Stream-Dataflow Accelerator Architecture



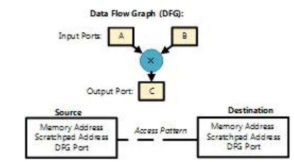
- Evaluation and Results

Outline

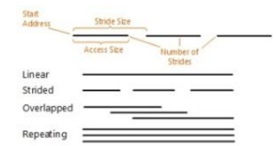
- Motivation and Overview



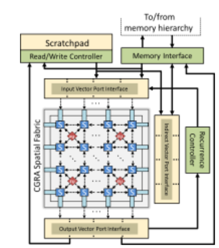
- Stream-Dataflow Execution Model



- Hardware-Software Interface and Example program



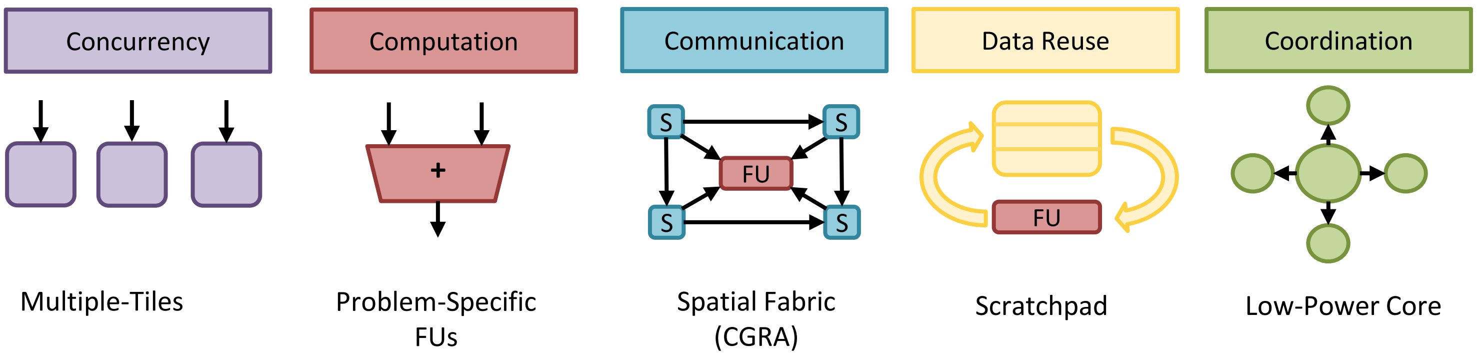
- **Stream-Dataflow Accelerator Architecture**



- Evaluation and Results

Requirements for Stream-Dataflow Accelerator Architecture

1. Should employ the common specialization principles and hardware mechanisms
   (*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)

**More detailed analysis contrasting data-parallel architectures and stream-dataflow architecture in paper*

Stream-Dataflow Accelerator Architecture

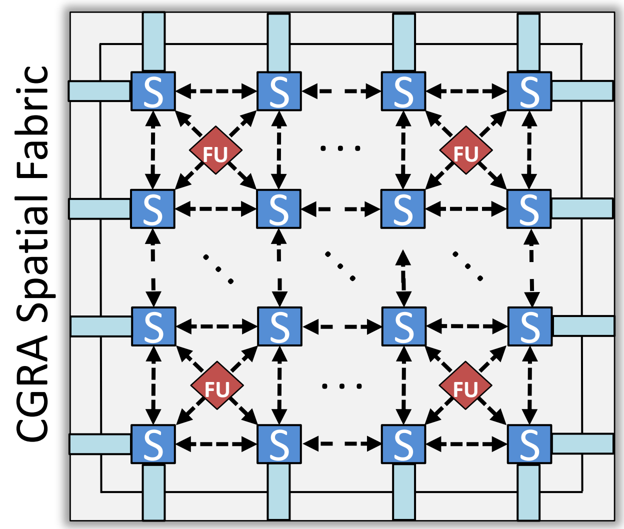
512b 64b

Stream-Dataflow Accelerator Architecture

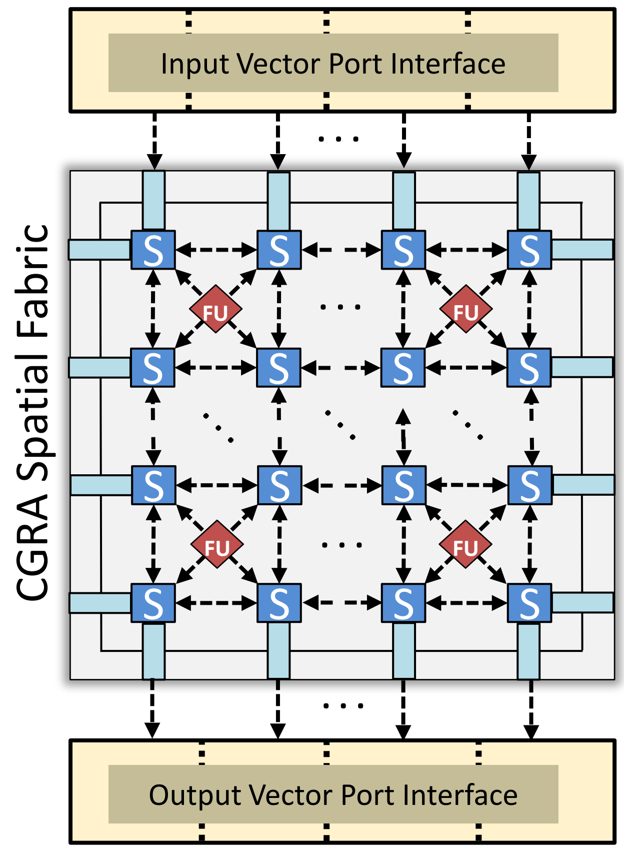
512b 64b

Dataflow:

- Coarse grained reconfigurable architecture (CGRA) for data parallel execution



Stream-Dataflow Accelerator Architecture



Dataflow:

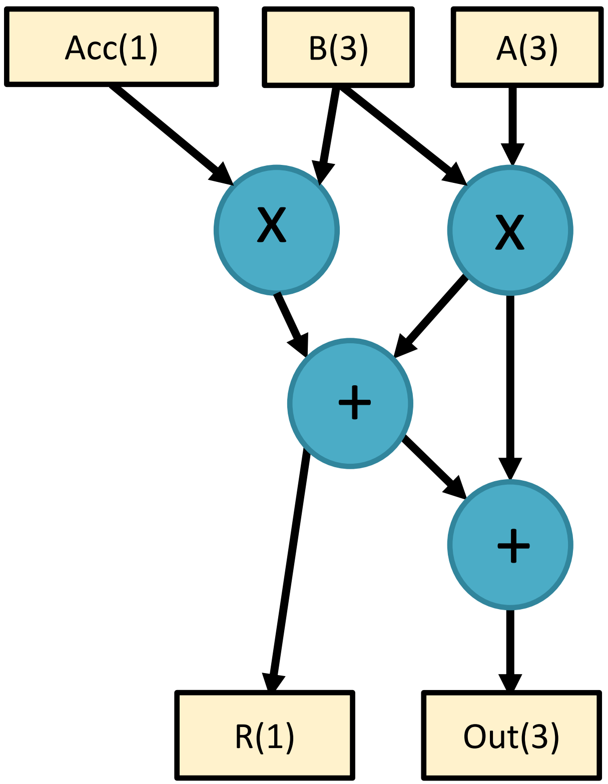
- Coarse grained reconfigurable architecture (CGRA) for data parallel execution
- Direct vector port interface into and out of CGRA for vector execution

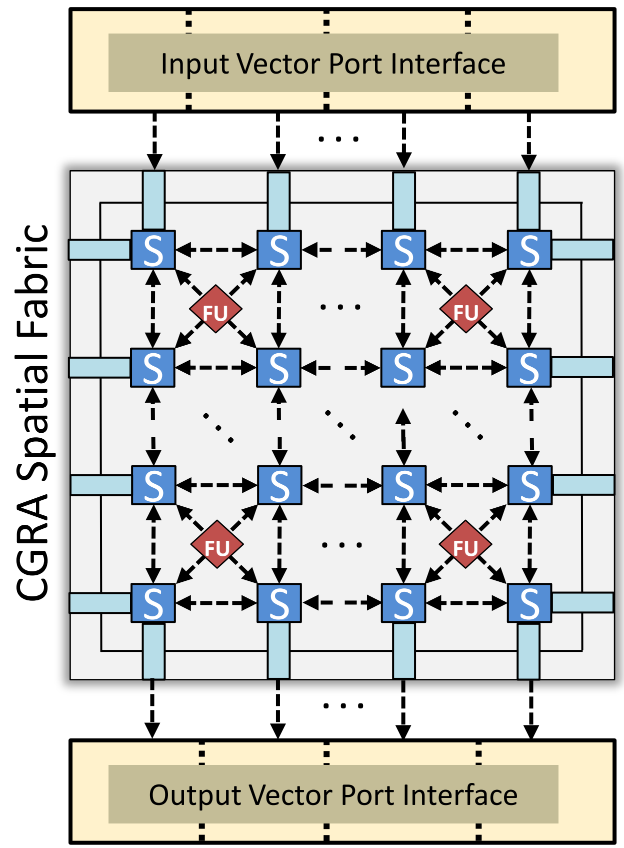
Stream-Dataflow Accelerator Architecture

512b 64b

Dataflow:

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



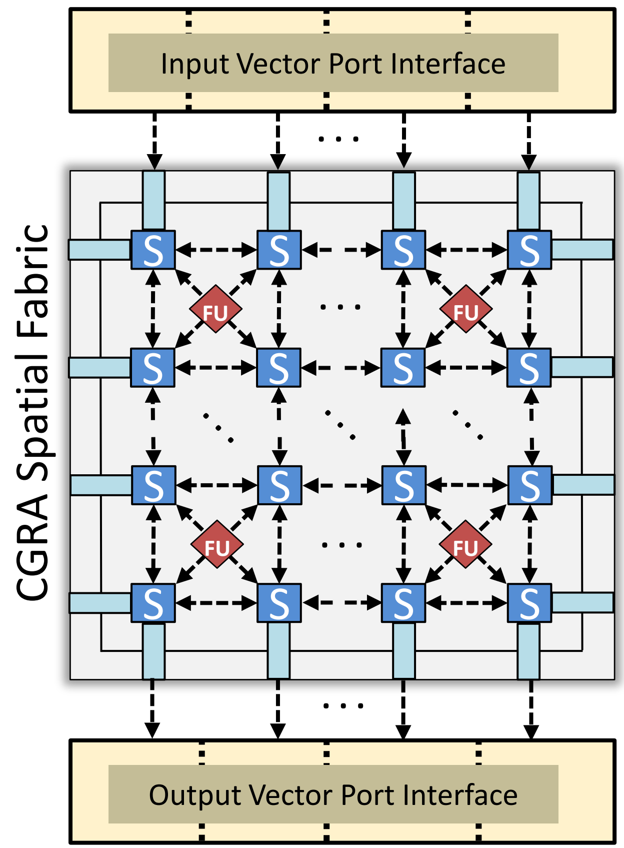


Stream-Dataflow Accelerator Architecture

512b 64b

Dataflow:

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



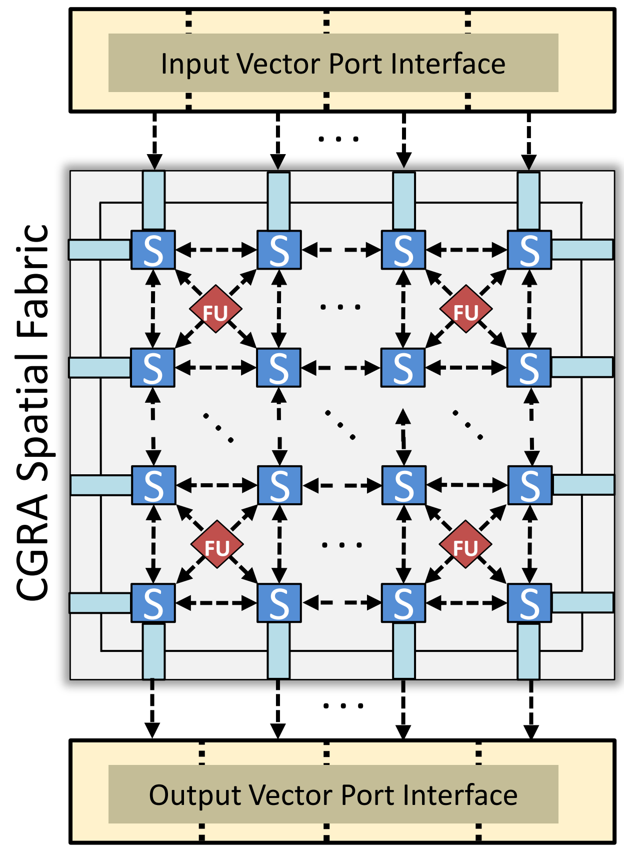
Stream-Dataflow Accelerator Architecture

512b 64b

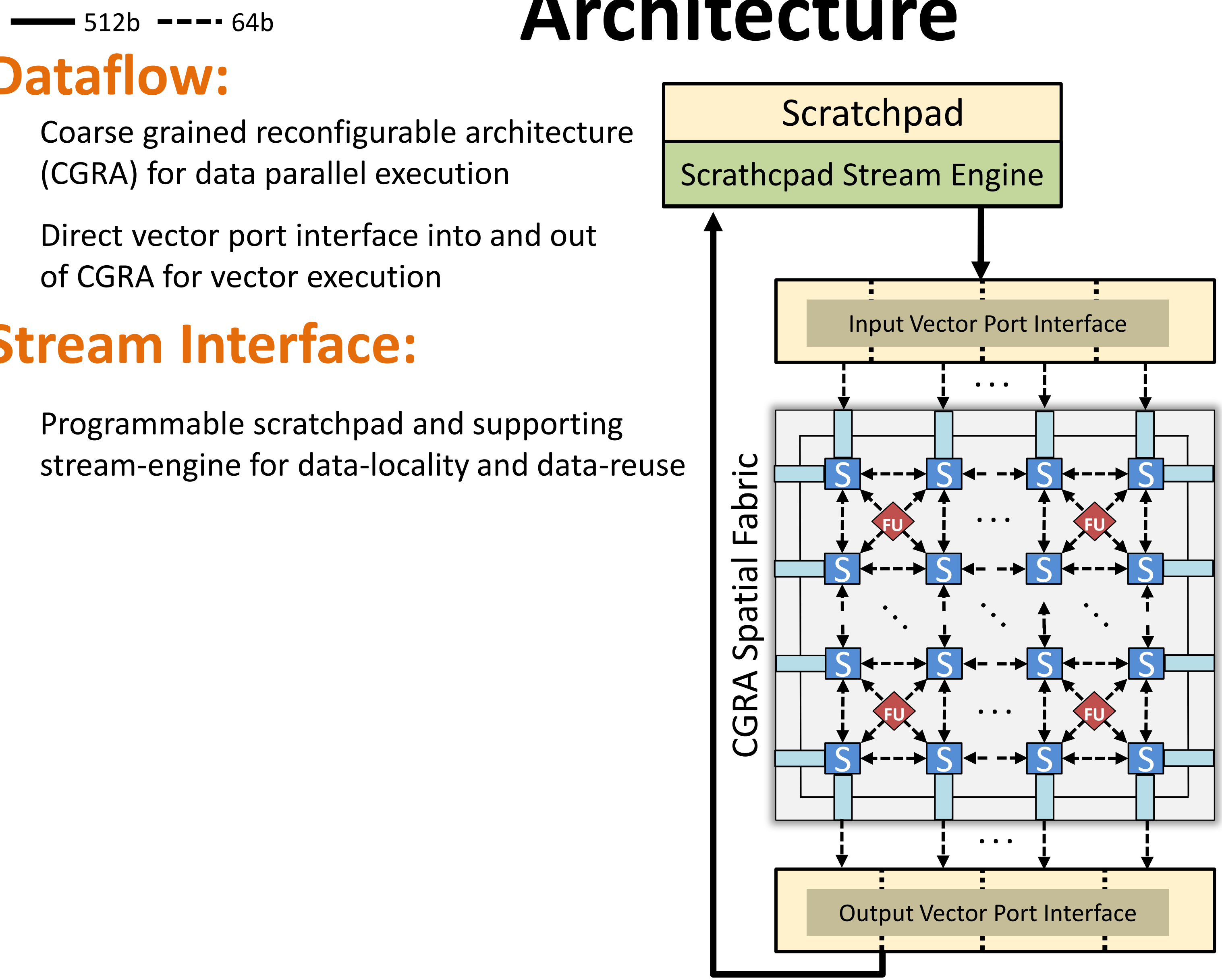
Dataflow:

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

Stream Interface:



Stream-Dataflow Accelerator Architecture



Dataflow:

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

Stream Interface:

- Programmable scratchpad and supporting stream-engine for data-locality and data-reuse

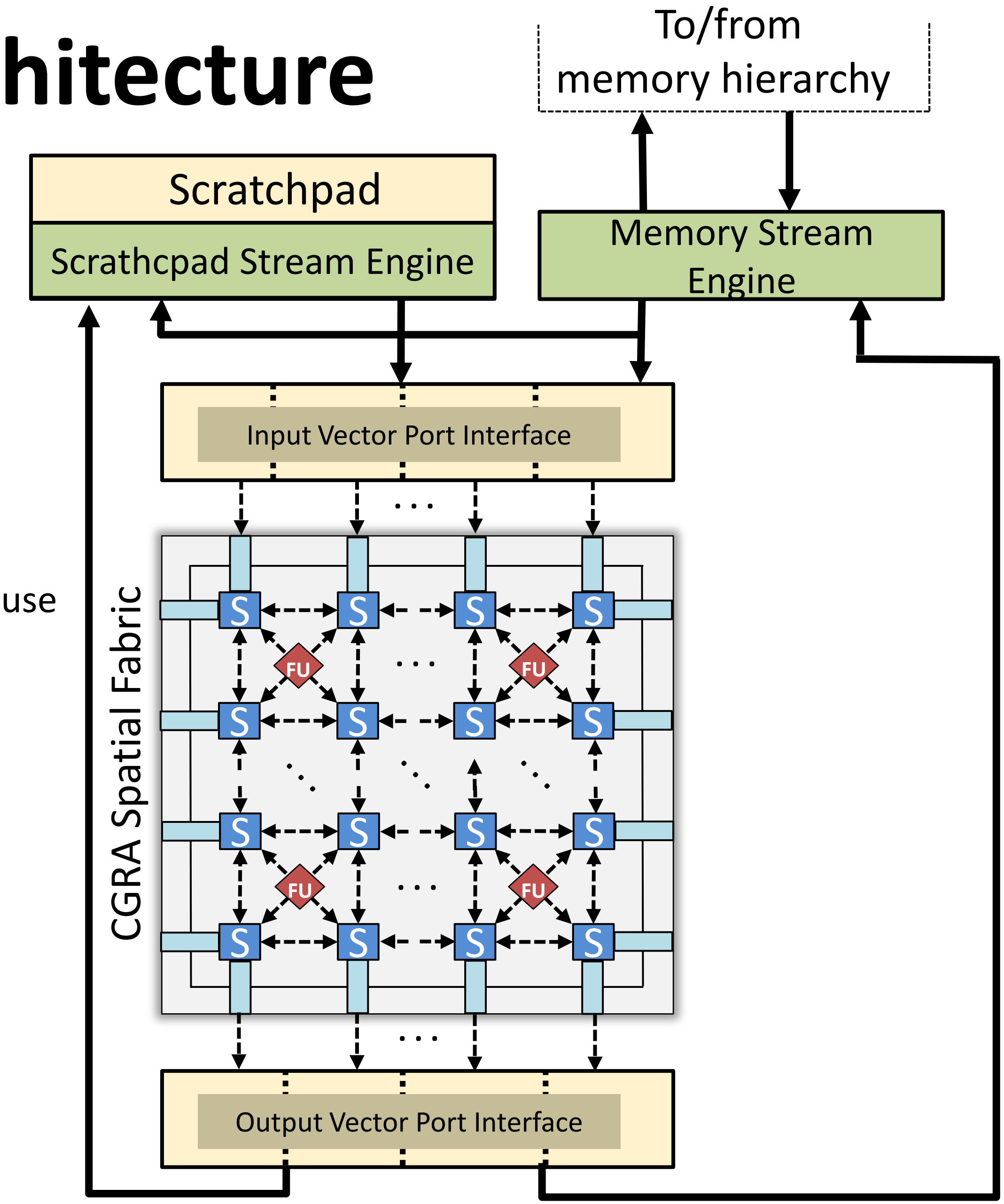
Stream-Dataflow Accelerator Architecture

Dataflow:

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

Stream Interface:

- 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



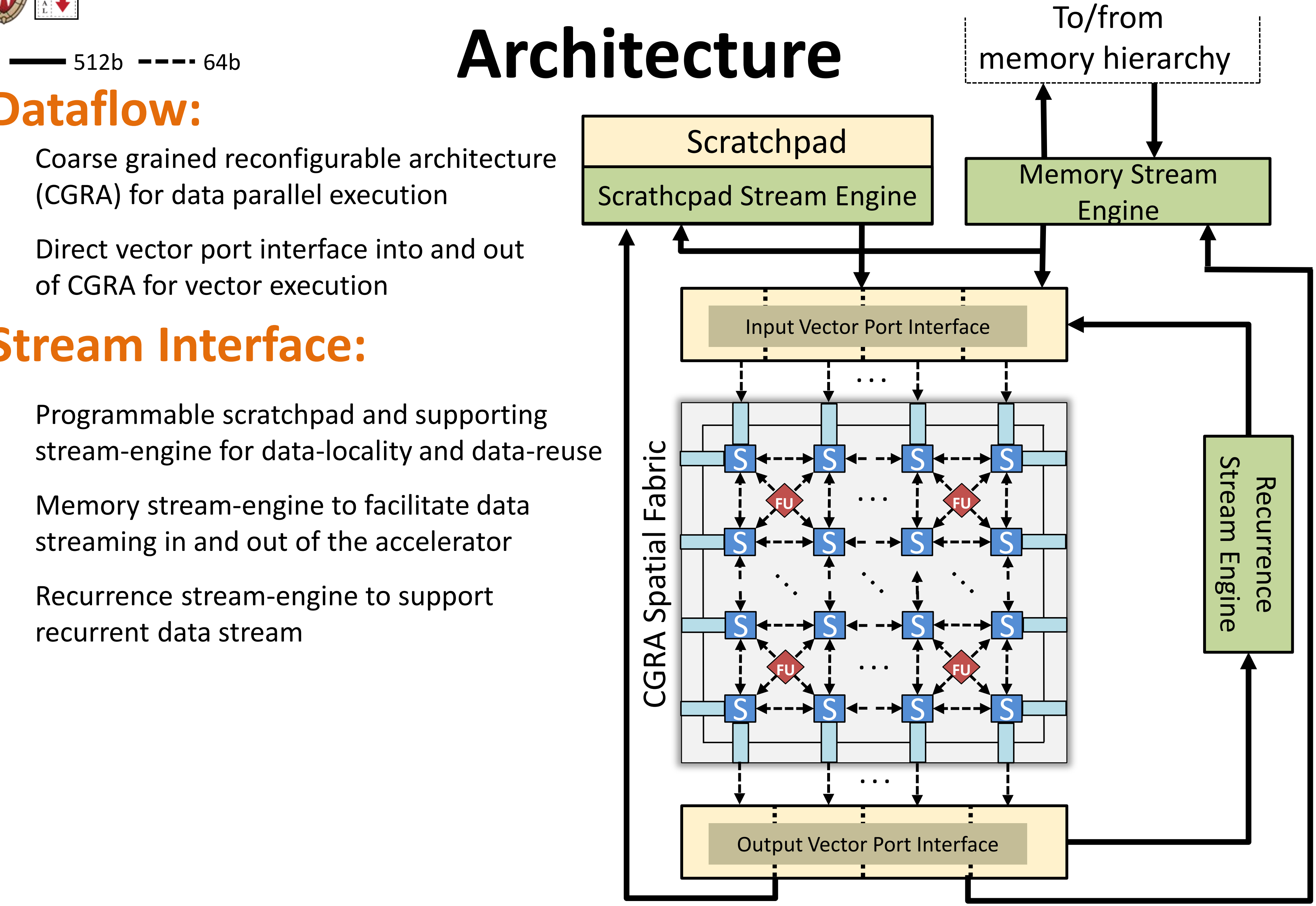
Stream-Dataflow Accelerator Architecture

Dataflow:

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

Stream Interface:

- 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 stream



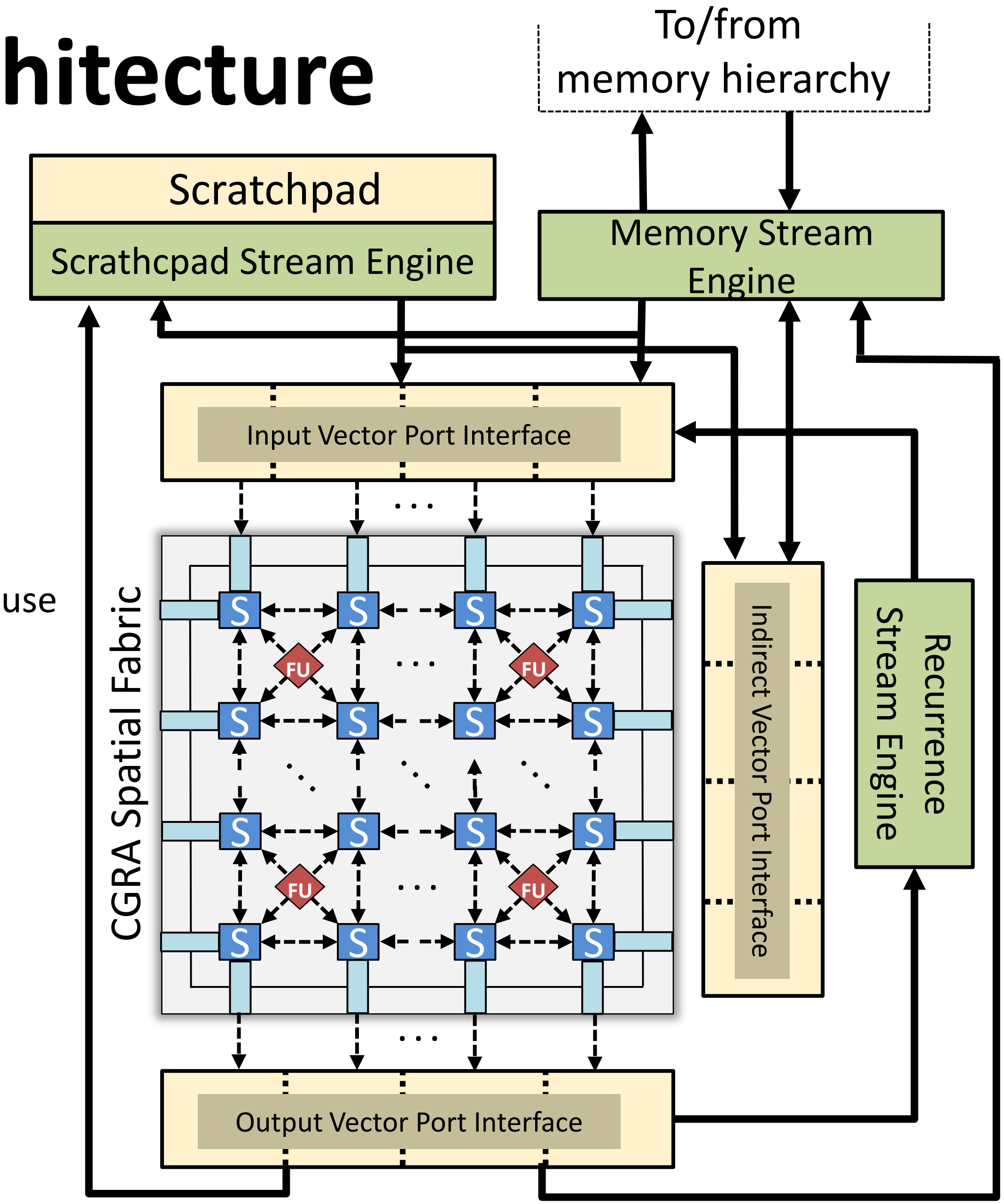
Stream-Dataflow Accelerator Architecture

Dataflow:

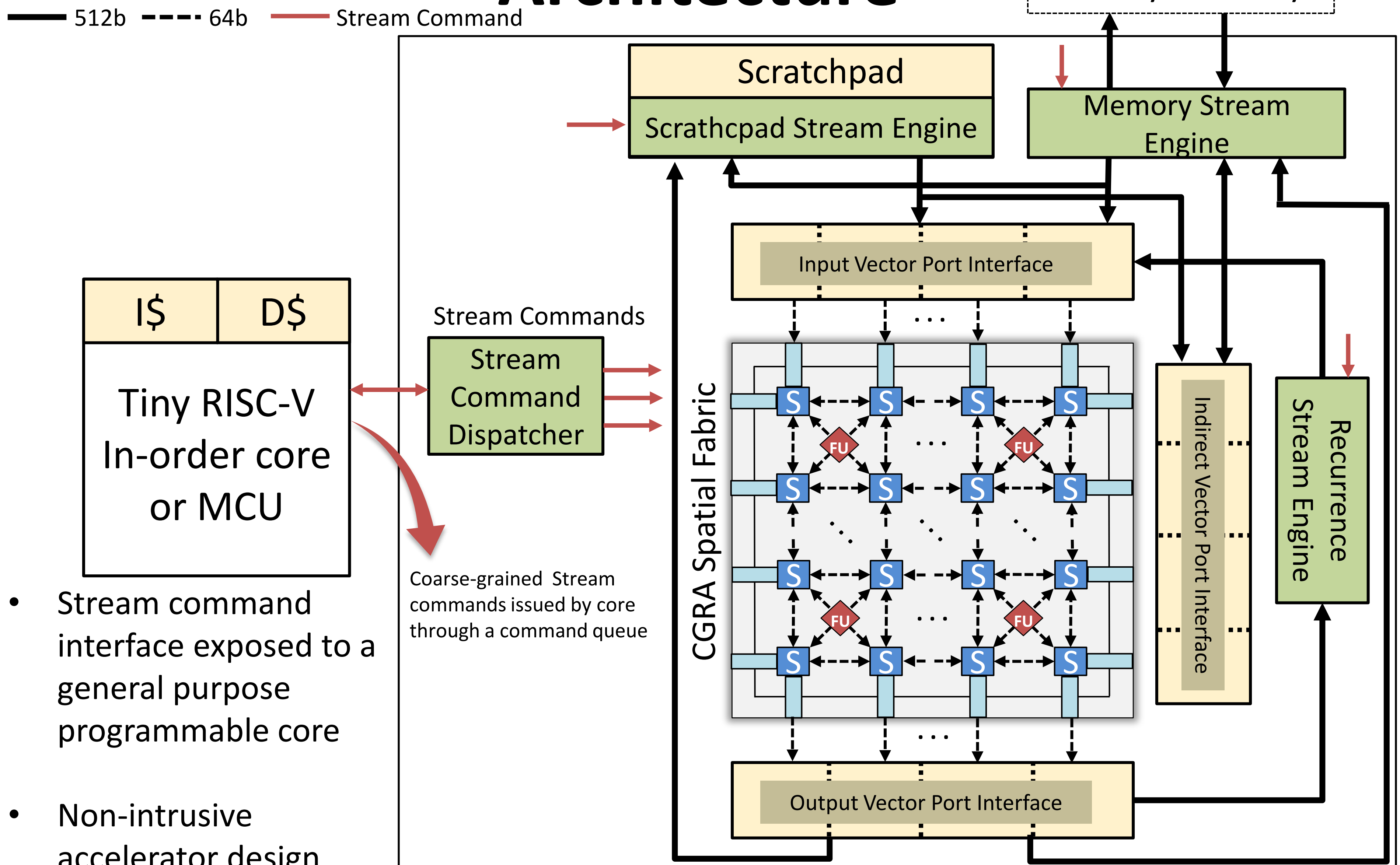
- Coarse grained reconfigurable architecture (CGRA) for data parallel execution
- Direct vector port interface into and out of CGRA for vector execution

Stream Interface:

- 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 stream
- Indirect vector port interface for streaming addresses (indirect load/stores)



Stream-Dataflow Accelerator Architecture



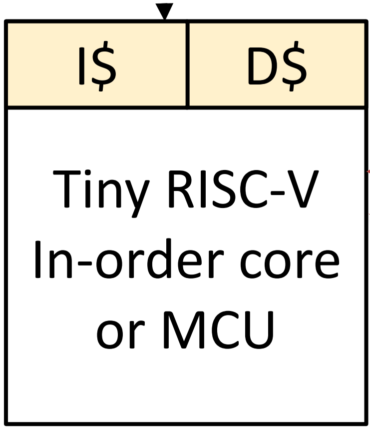
- Stream command interface exposed to a general purpose programmable core
- Non-intrusive accelerator design

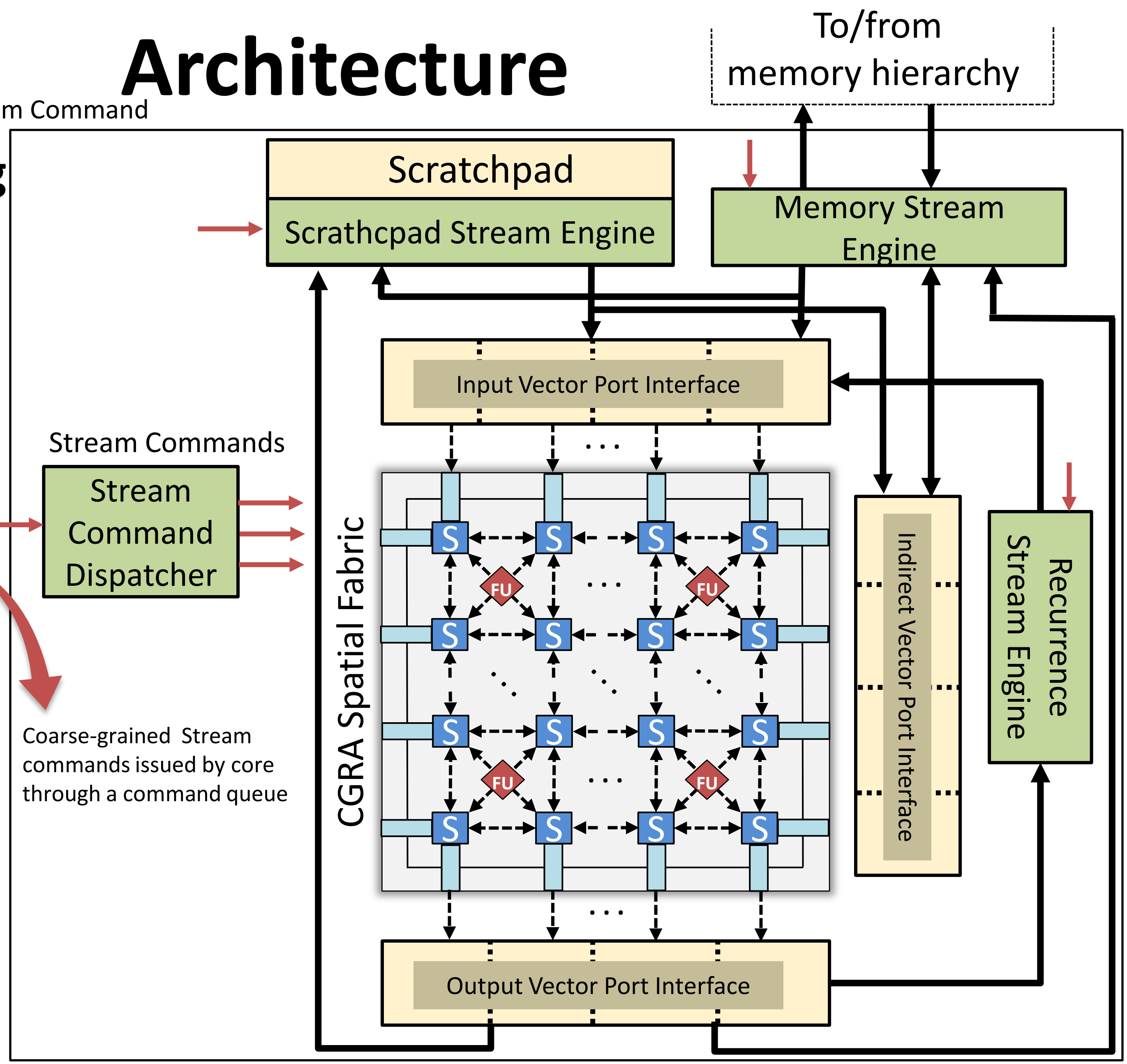
Stream-Dataflow Accelerator Architecture

512b 64b Stream Command

Stream ISA Encoding

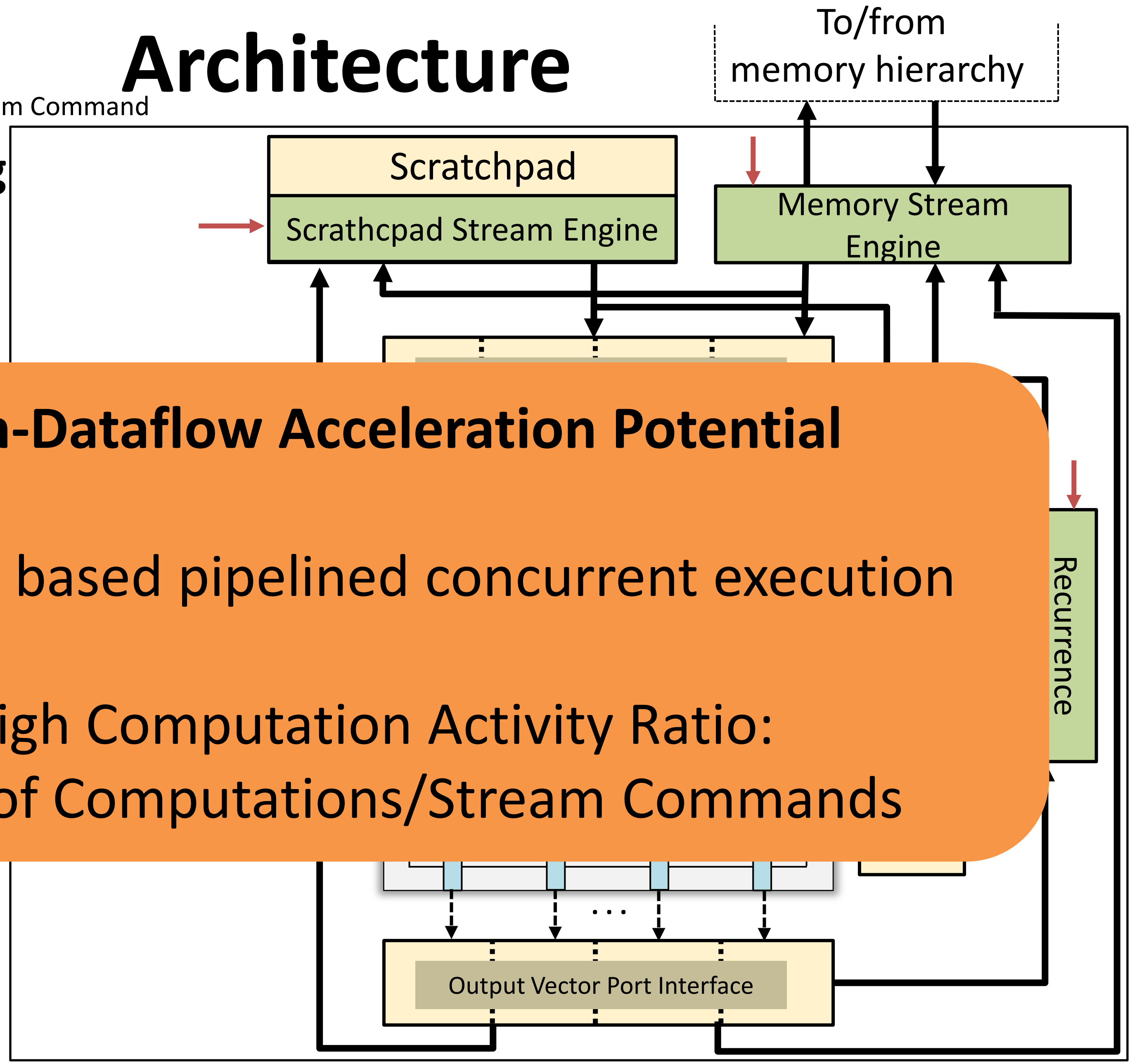
```
Send a[0: N] → P1
Send b[0: N] → P2
Get P3 → c
```





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

Stream-Dataflow Accelerator Architecture



Stream ISA Encoding

```
Send a[0: N] → P1
Send b[0: N] → P2
Get P3 → c
```

Stream-Dataflow Acceleration Potential

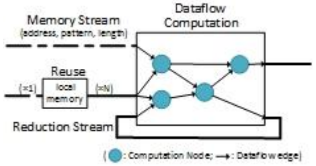
1. Dataflow based pipelined concurrent execution

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

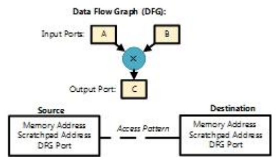
- general purpose programmable core
- Non-intrusive accelerator design

Outline

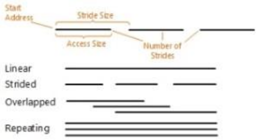
- Motivation and Overview



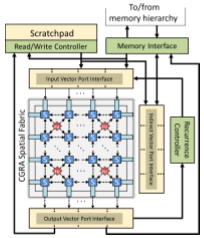
- Stream-Dataflow Execution Model



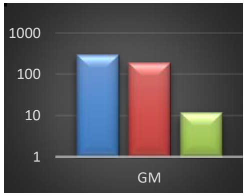
- Hardware-Software Interface and Example program



- Stream-Dataflow Accelerator Architecture

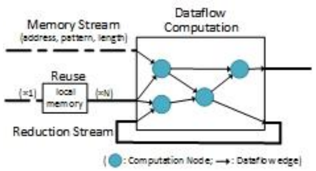


- Evaluation and Results

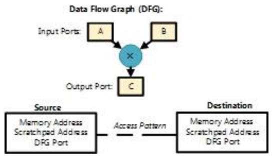


Outline

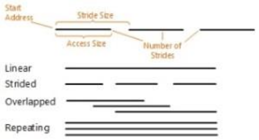
- Motivation and Overview



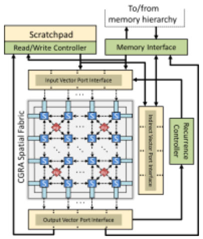
- Stream-Dataflow Execution Model



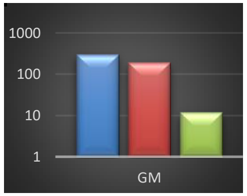
- Hardware-Software Interface and Example program



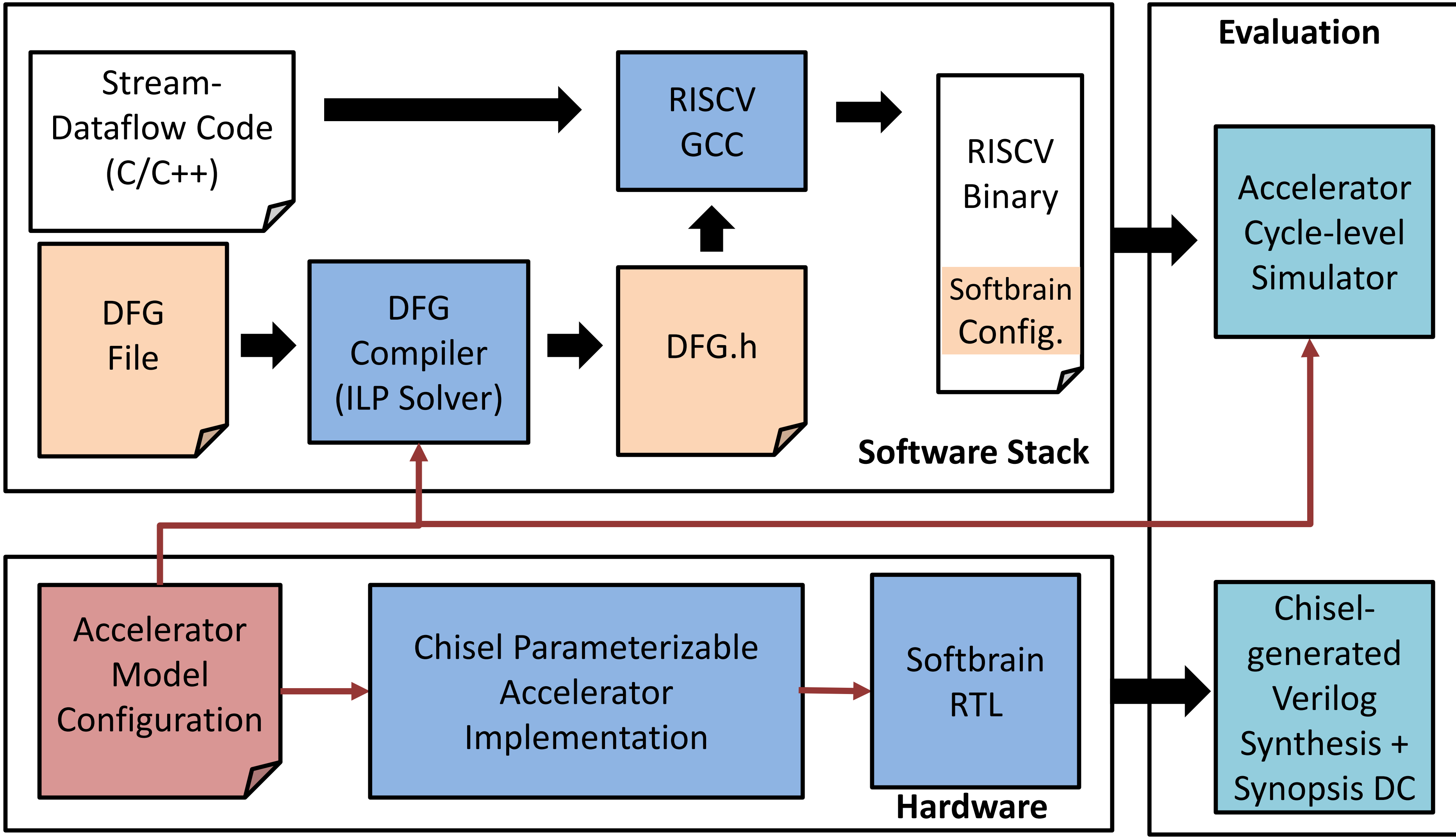
- Stream-Dataflow Accelerator Architecture



- **Evaluation and Results**



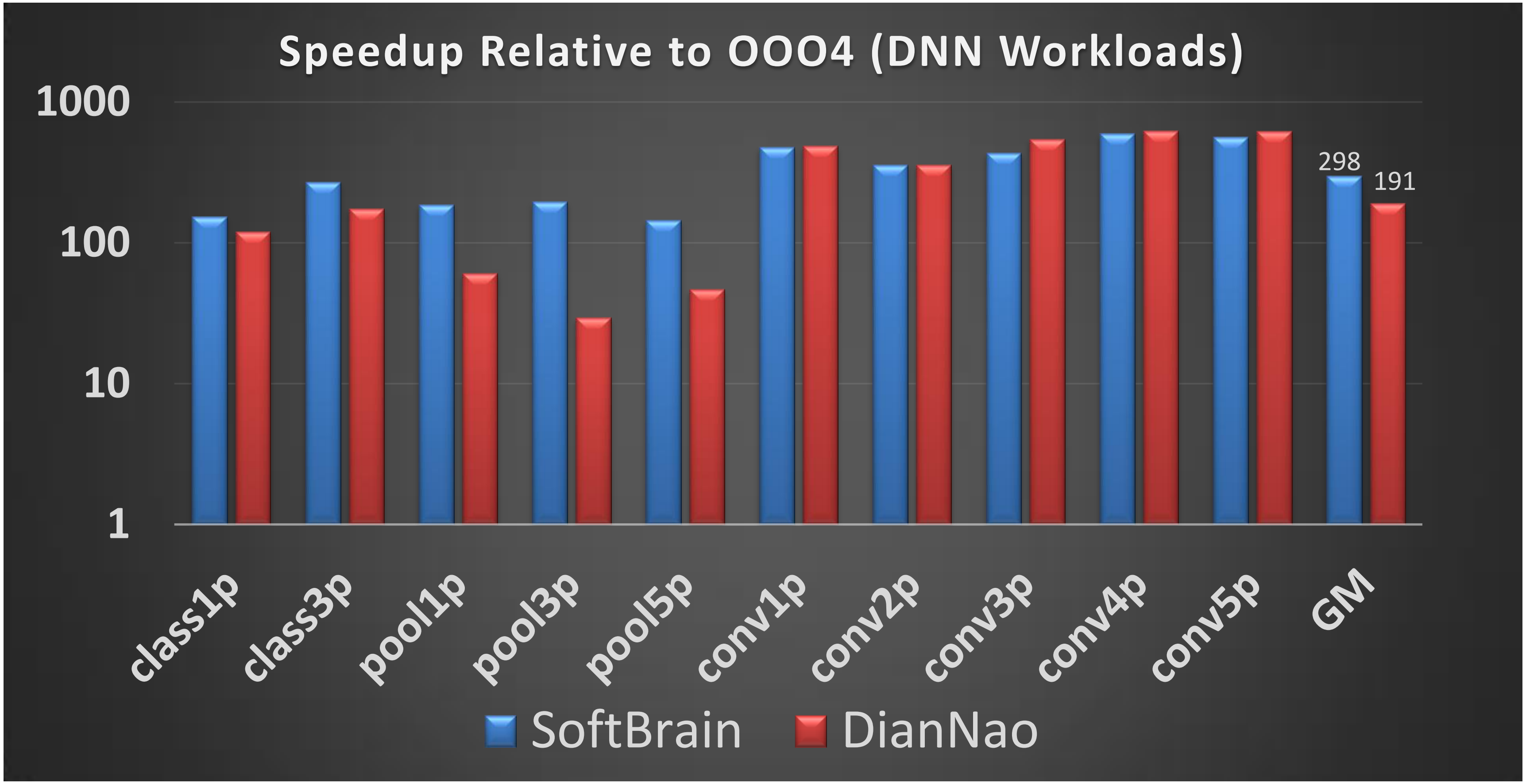
Stream-Dataflow Implementation: *Softbrain*



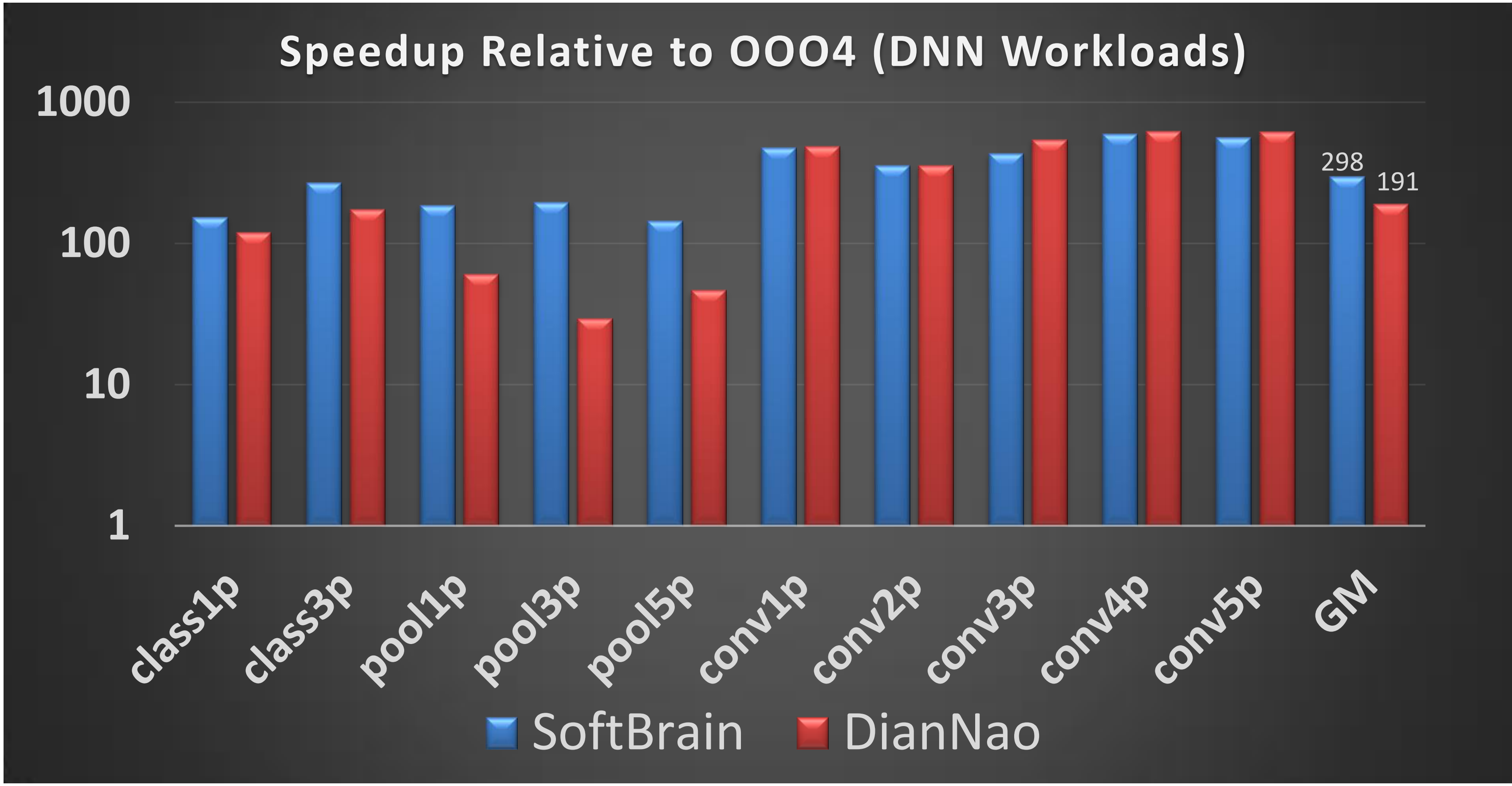
Evaluation Methodology

- 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

Domain-Specific Accelerator Comparison (Softbrain vs DianNao)



Domain-Specific Accelerator Comparison (Softbrain vs DianNao)



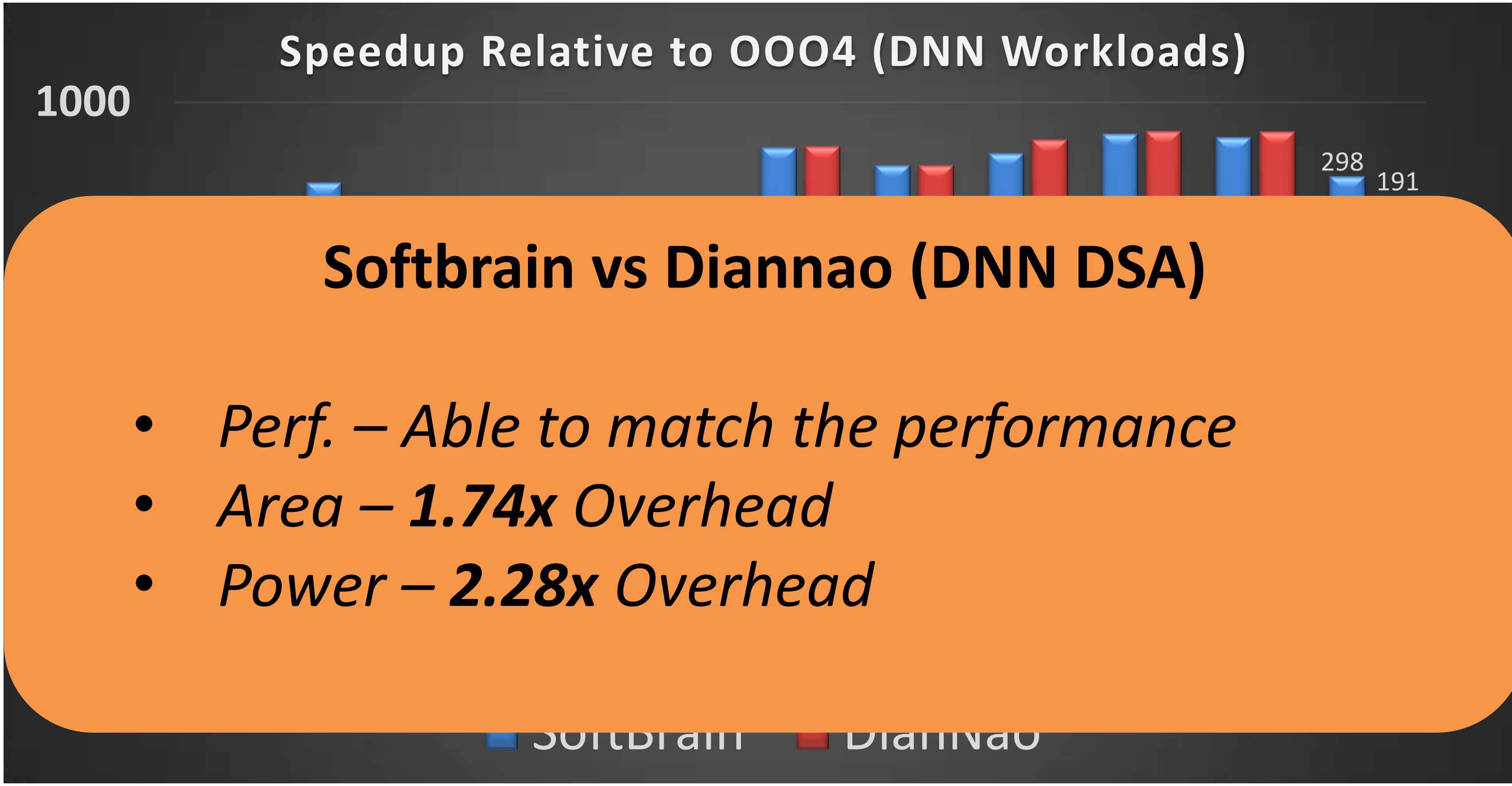
DianNao Area: **2.16 mm²**

DianNao Power: **420 mW**

Softbrain Area: **3.76 mm²**

Softbrain Power: **950 mW**

Domain-Specific Accelerator Comparison (Softbrain vs DianNao)



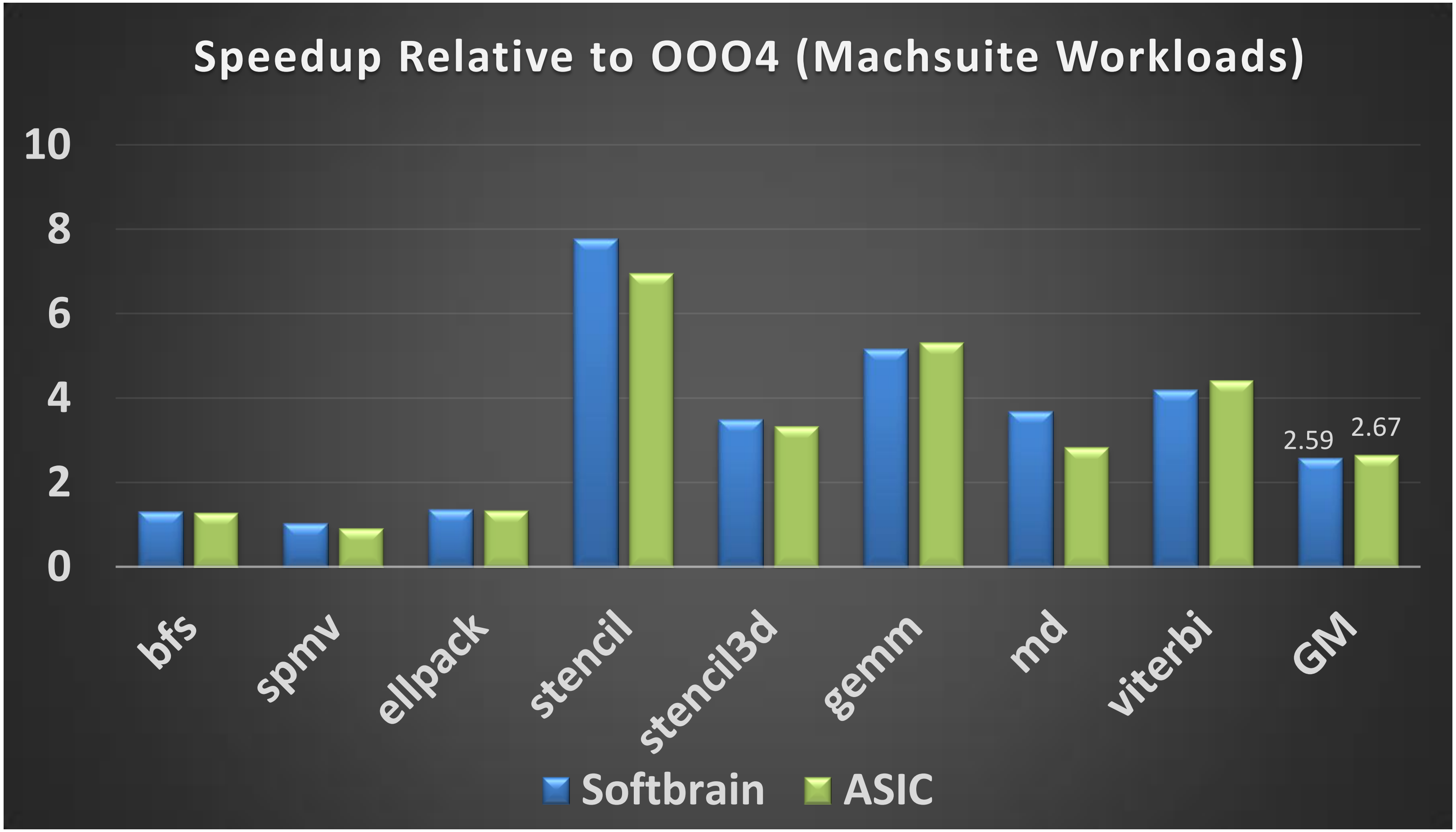
Softbrain vs Diannao (DNN DSA)

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

DianNao Area: **2.16 mm^2**
DianNao Power: **420 mW**

Softbrain Area: **3.76 mm^2**
Softbrain Power: **950 mW**

Softbrain vs ASIC Designs Comparison

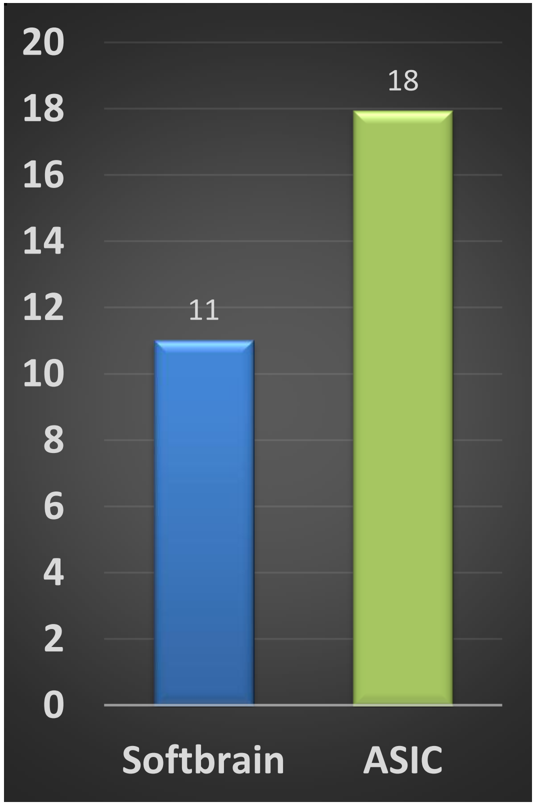


Aladdin* generated ASIC design points – Resources constrained to be in ~15% of Softbrain Perf. to do iso-performance analysis

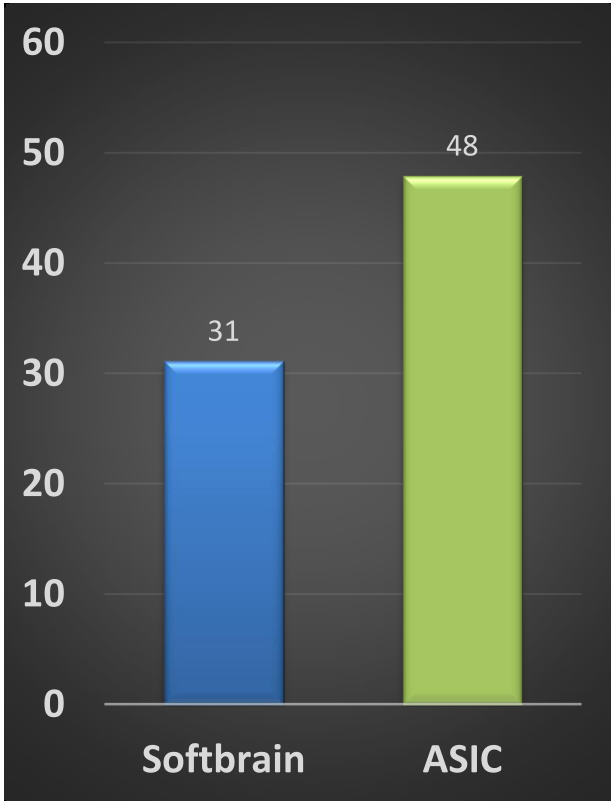
*Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. Sophia Shao , .et. al

Softbrain vs ASIC Comparison

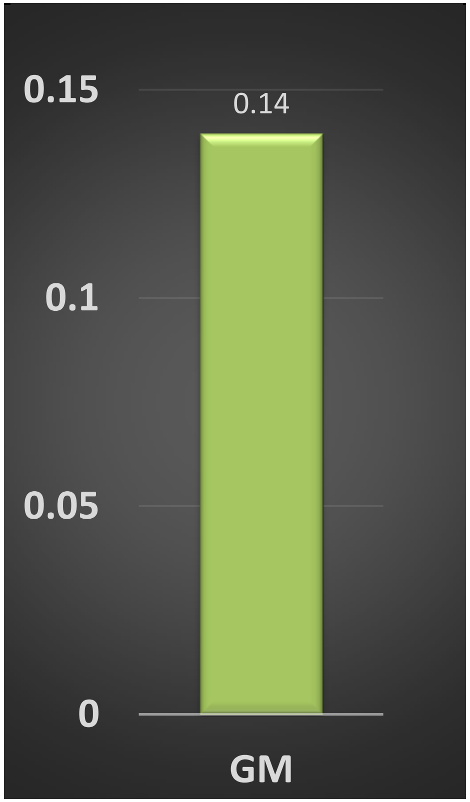
Power Efficiency Relative to OOO4 (GM)



Energy Efficiency Relative to OOO4 (GM)



ASIC Area Relative to Softbrain (GM)



Softbrain vs ASIC Comparison

Power Efficiency Relative to OOO4 (GM)

Energy Efficiency Relative to OOO4 (GM)

ASIC Area Relative to Softbrain (GM)

Softbrain vs ASIC designs

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

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

Conclusion

Conclusion

- 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

Conclusion

- 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

Conclusion

- 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

Conclusion

- Stream-Dataflow Acceleration
  - Stream-Dataflow **Execution Model** – Abstracts typical accelerator computation phases using a dataflow graph
  - Stream-Dataflow **ISA Encoding** and **Hardware-Software Interface** –

Getting There !!

A good enabler for exploring general purpose programmable hardware acceleration ….

-
-
  - Compared to application specific designs (ASICs), Softbrain has **~2x** overheads in power and **~8x** in area

Backup

UCLA

Traditional Arch.

Programs

General Language

Compiler

General ISA

General Purpose Hardware

Accelerator (DSA)

Domain-Specific Programs

Tiny H/W-S/W Interface

Application/Domain Specific Hardware

UCLA

Traditional Arch.

Accelerator (DSA)

Programs

General Language

Compiler

General ISA

General Purpose Hardware

Domain-Specific Programs

Tiny H/W-S/W Interface

Application/Domain Specific Hardware

10-100x Performance/Power or Performance/Area
(completely lose generality/programmability)

UCLA

Traditional Arch. | Progammable Hardware Accelerator | Accelerator (DSA)

Traditional Arch.

Programs

General Language

Compiler

General ISA

General Purpose Hardware

Progammable Hardware Accelerator

Programs ("Specialized")

H/W Parameters

Re-Configurable Hardware

Accelerator (DSA)

Domain-Specific Programs

Tiny H/W-S/W Interface

Application/Domain Specific Hardware

10-100x Performance/Power or Performance/Area
(completely lose generality/programmability)

UCLA

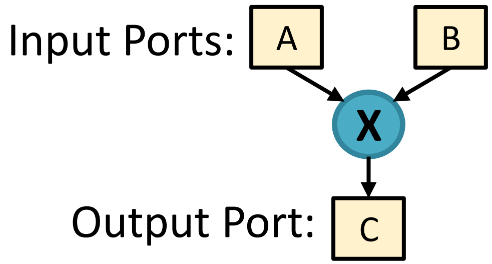
Traditional Arch.	**Progammable Hardware Accelerator**	Accelerator (DSA)
Programs	Programs ("Specialized")	Domain-Specific Programs
General Language	H/W Parameters	Tiny H/W-S/W Interface
Compiler		
General ISA		
General Purpose Hardware	Re-Configurable Hardware	Application/Domain Specific Hardware

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

Stream-Dataflow Execution Model Detailed Example

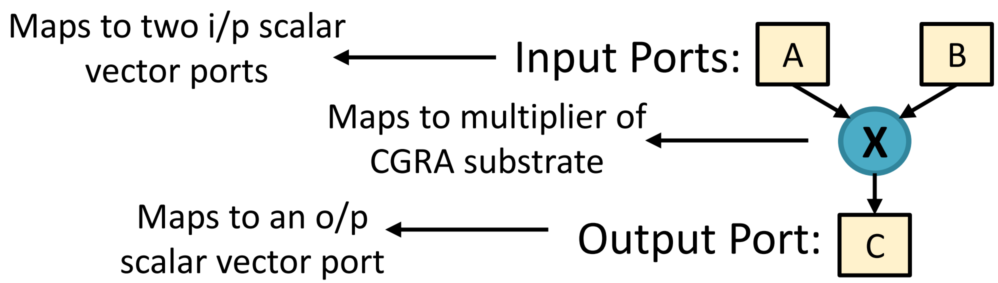
Stream-Dataflow Execution Model Detailed Example

C[i] = A[i] * B[i]



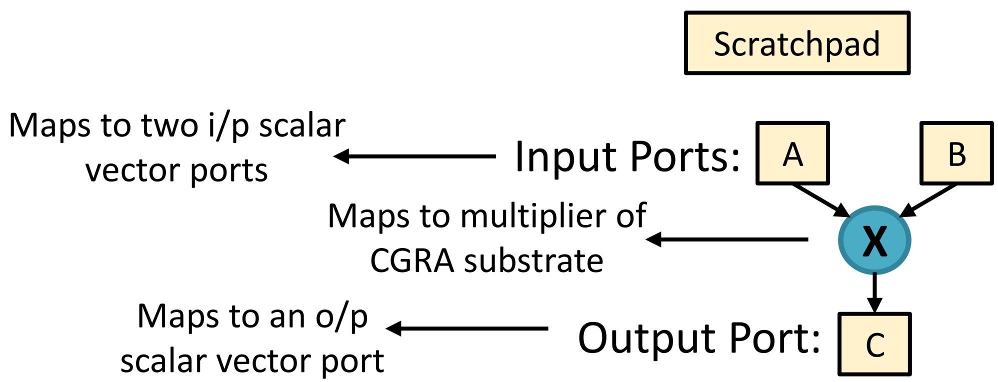
Stream-Dataflow Execution Model Detailed Example

C[i] = A[i] * B[i]



Stream-Dataflow Execution Model Detailed Example

C[i] = A[i] * B[i]



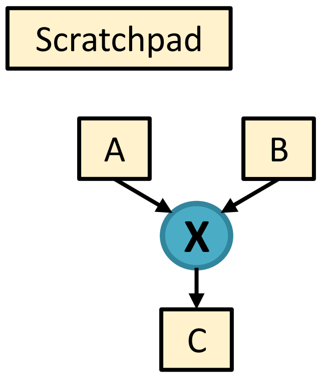
Stream-Dataflow Execution Model Detailed Example

C[i] = A[i] * B[i]

Stream Commands

Time

Program Order



Stream-Dataflow Execution Model Detailed Example

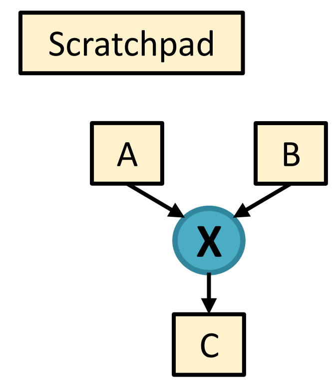
Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]

Stream Commands Time

Program Order



CGRA fabric state

Low-power core state

Stream-Dataflow Execution Model Detailed Example

Legend:

- Enqueued □
- Dispatched ○
- Resource idle ·······
- Resource in use ——
- All data at dest. ●
- Barrier ⊙
- Dependency →
- Iter. boundary ⁄

C[i] = A[i] * B[i]

Stream Commands

Time

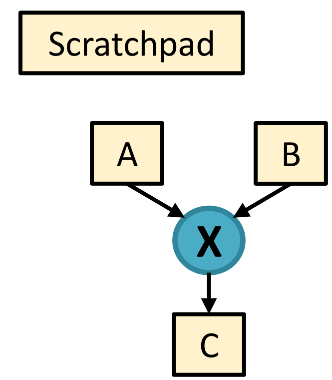
C1) Mem → Scratch

Program Order

CGRA fabric state

Low-power core state

Command generation



Stream-Dataflow Execution Model Detailed Example

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]

Stream Commands

C1) Mem → Scratch

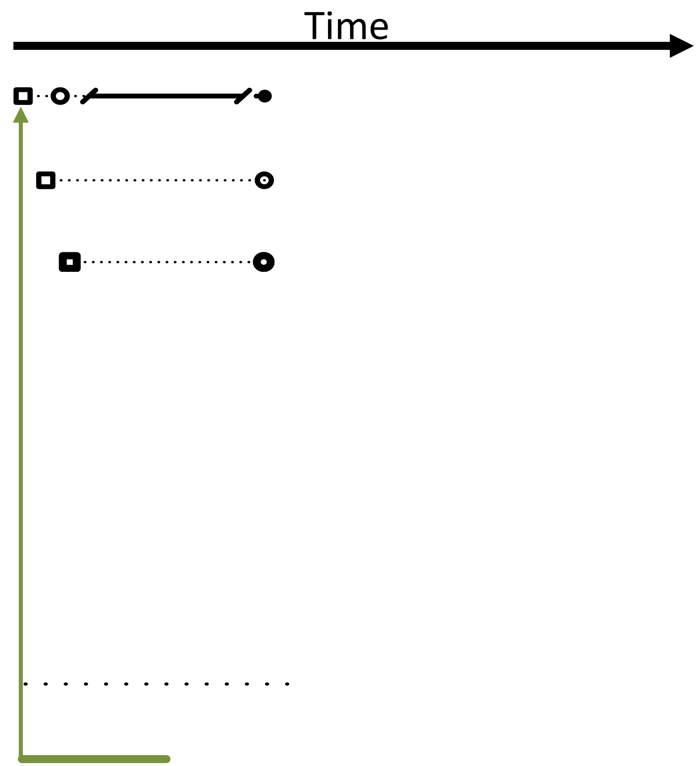
C2) Scratch Wr Barrier

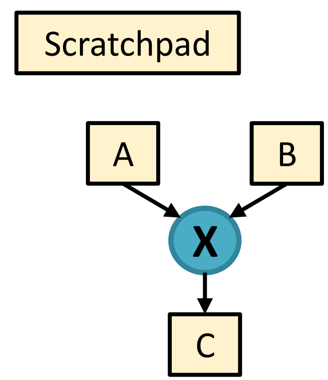
C3) Scratch → Port A

Program Order

CGRA fabric state

Low-power core state





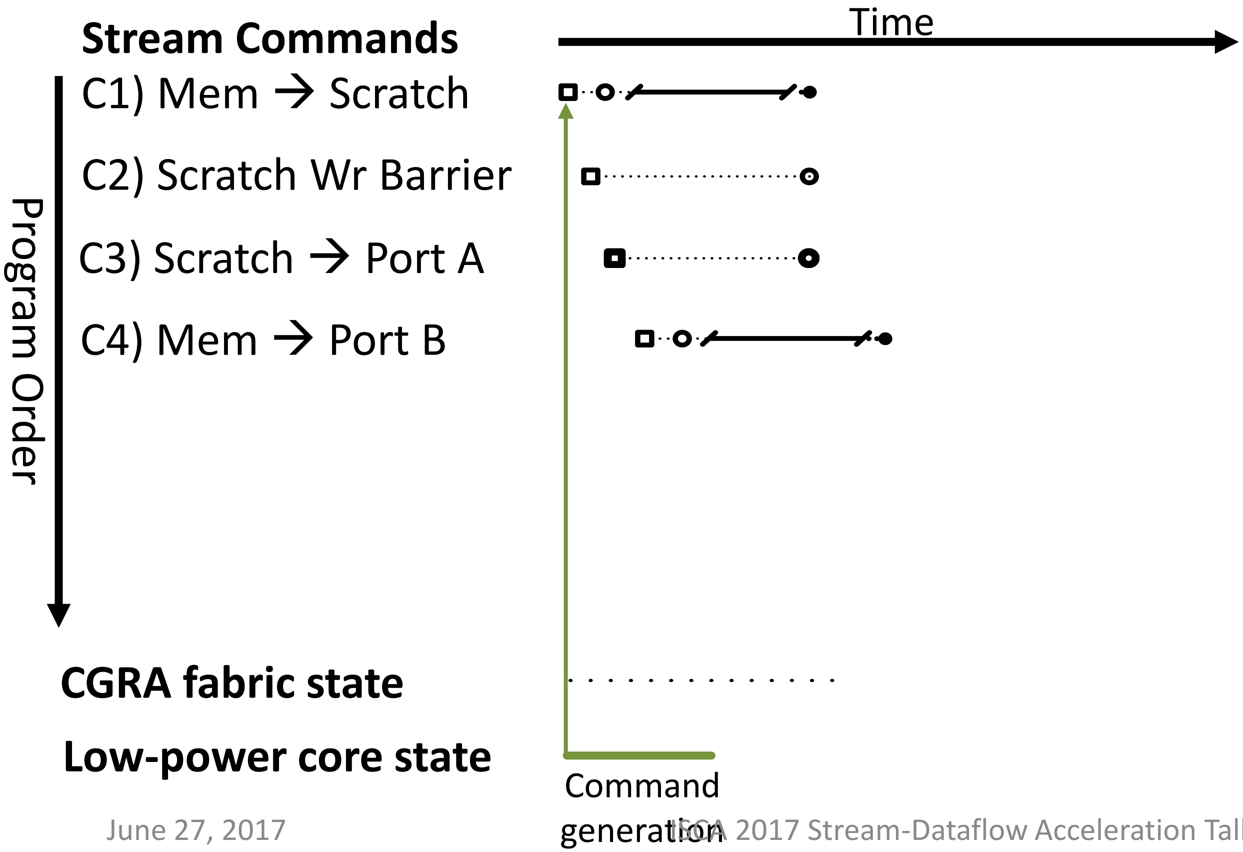
Stream-Dataflow Execution Model Detailed Example

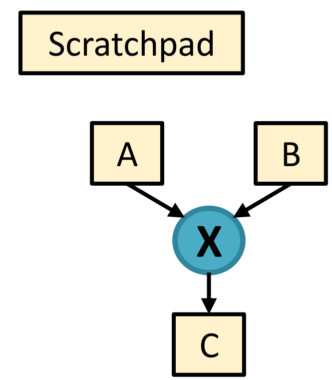
UCLA

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]





Stream-Dataflow Execution Model Detailed Example

UCLA

Legend:

- Enqueued
- Dispatched
- Resource idle
- Resource in use
- All data at dest.
- Barrier
- Dependency
- Iter. boundary

$C[i] = A[i] * B[i]$

Stream Commands

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

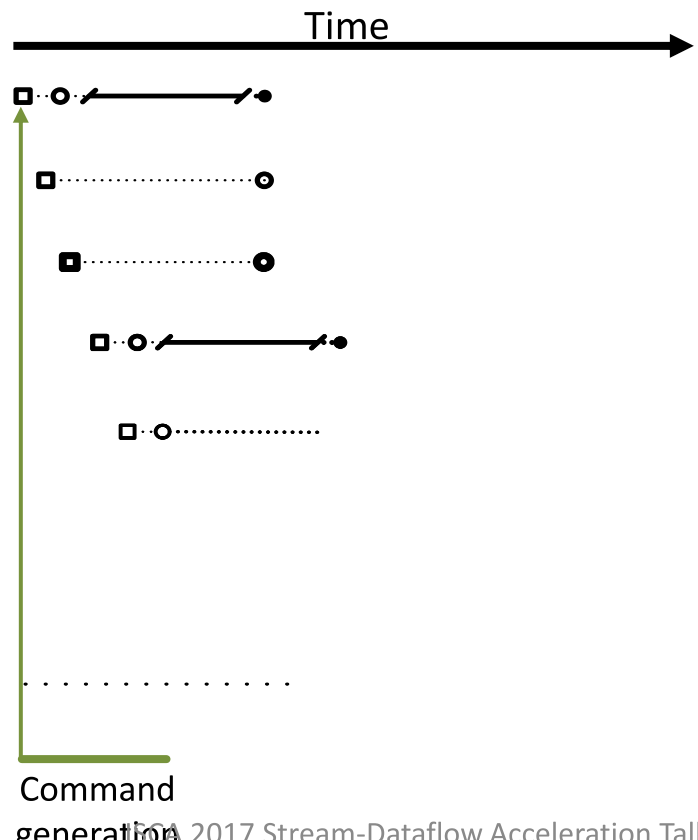
C4) Mem → Port B

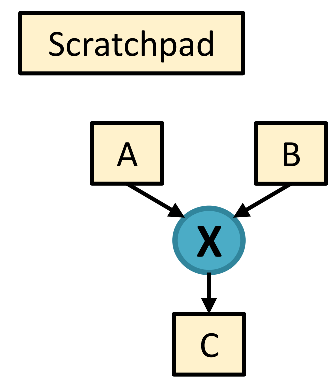
C5) Port C → Mem

Program Order

CGRA fabric state

Low-power core state





Stream-Dataflow Execution Model Detailed Example

UCLA

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]

Stream Commands

Program Order

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

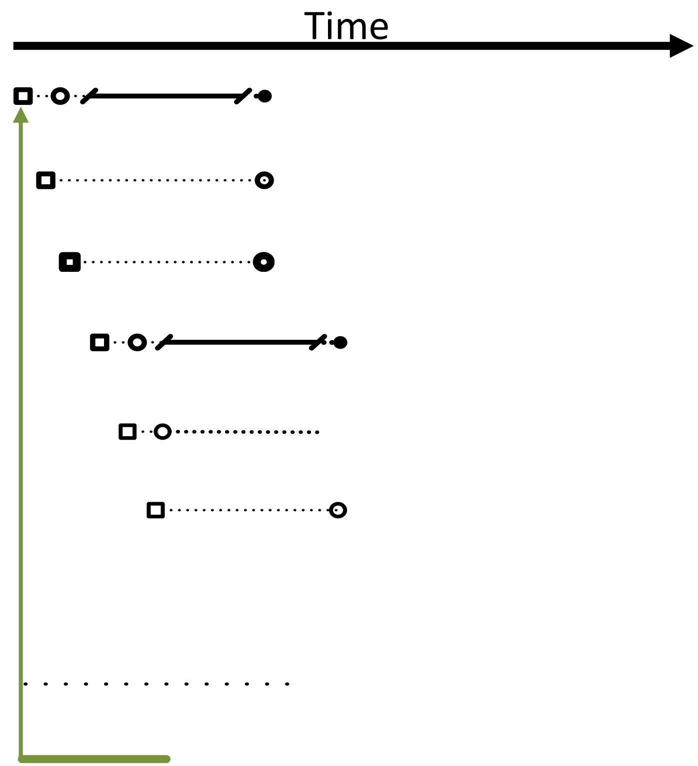
C4) Mem → Port B

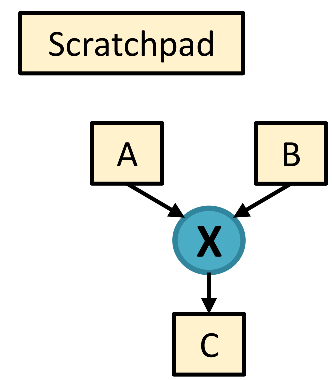
C5) Port C → Mem

C6) Mem → Port B

CGRA fabric state

Low-power core state

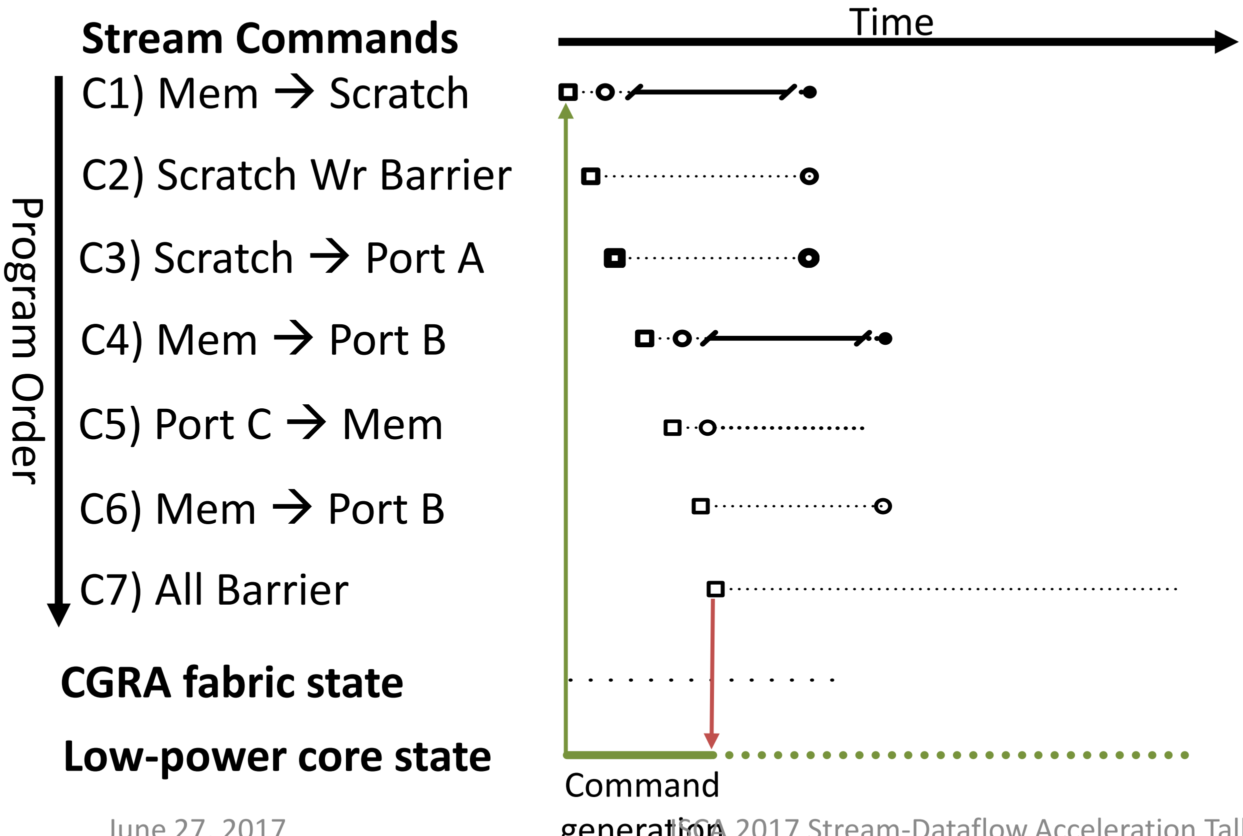




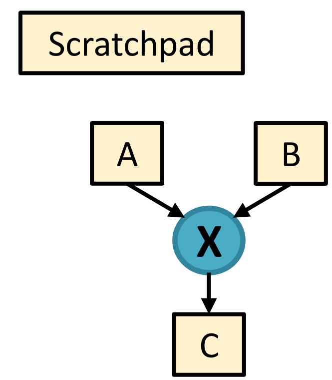
Stream-Dataflow Execution Model Detailed Example

Legend:

- Enqueued
- Dispatched
- Resource idle
- Resource in use
- All data at dest.
- Barrier
- Dependency
- Iter. boundary



C[i] = A[i] * B[i]



Stream-Dataflow Execution Model Detailed Example

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]

Stream Commands

Program Order

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

C4) Mem → Port B

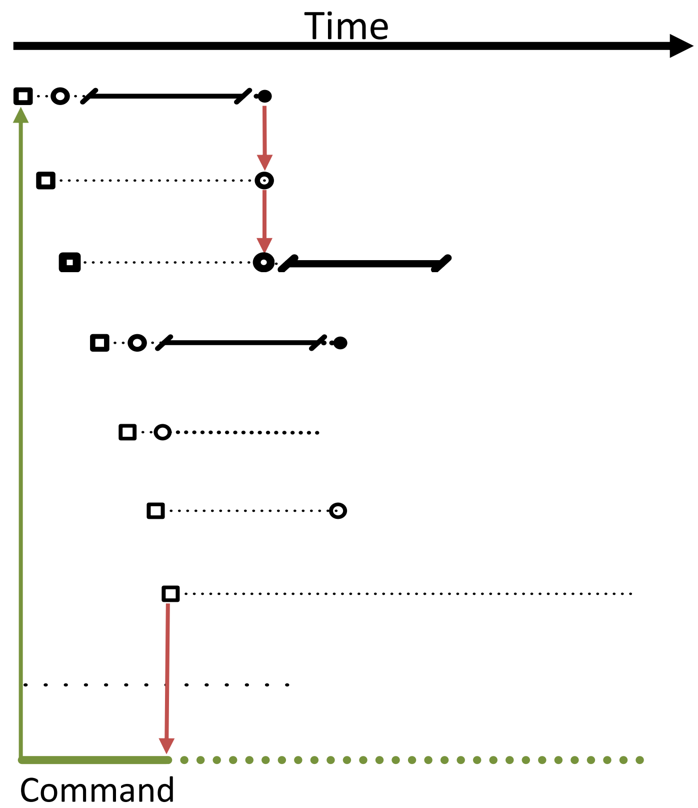
C5) Port C → Mem

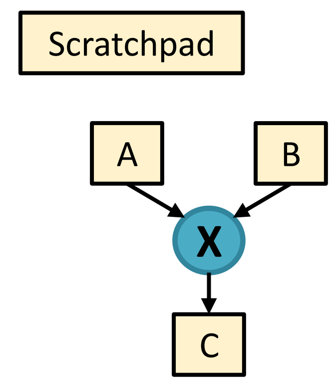
C6) Mem → Port B

C7) All Barrier

CGRA fabric state

Low-power core state





Stream-Dataflow Execution Model Detailed Example

Legend:

- Enqueued
- Dispatched
- Resource idle
- Resource in use
- All data at dest.
- Barrier
- Dependency
- Iter. boundary

$C[i] = A[i] * B[i]$

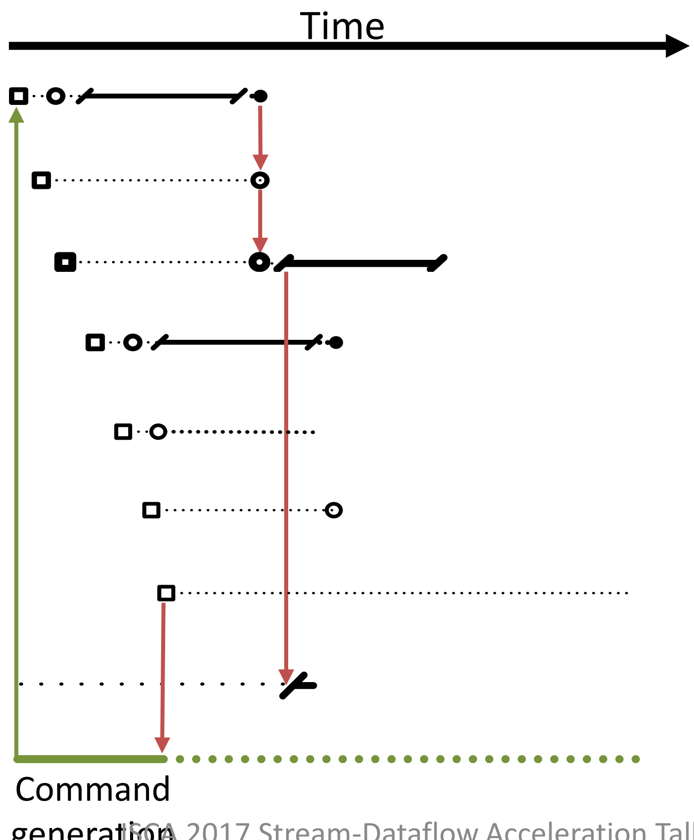
Stream Commands

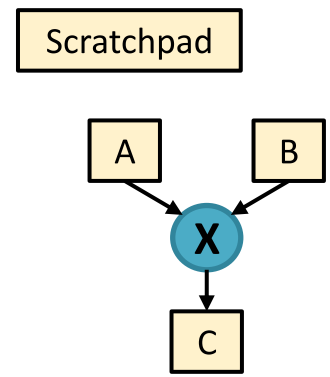
Program Order

C1) Mem → Scratch
C2) Scratch Wr Barrier
C3) Scratch → Port A
C4) Mem → Port B
C5) Port C → Mem
C6) Mem → Port B
C7) All Barrier

CGRA fabric state

Low-power core state





Stream-Dataflow Execution Model Detailed Example

UCLA

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]

Stream Commands

Program Order

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

C4) Mem → Port B

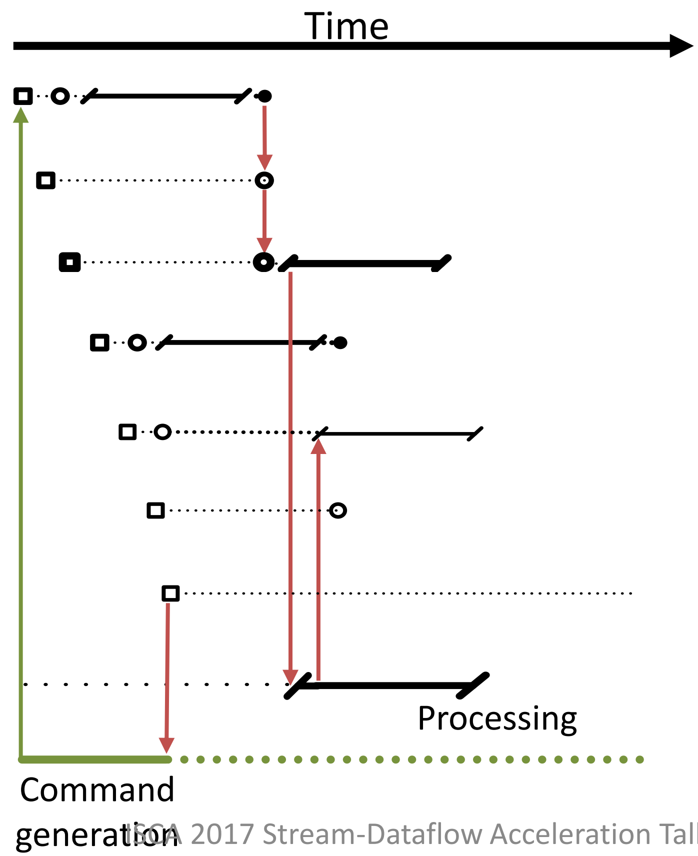
C5) Port C → Mem

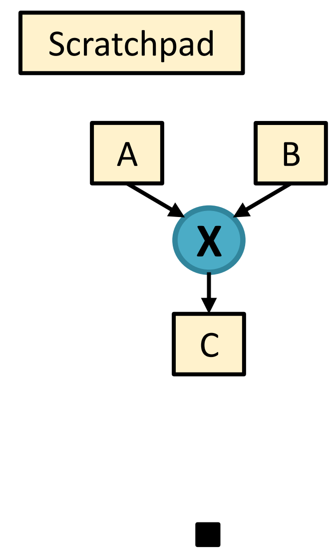
C6) Mem → Port B

C7) All Barrier

CGRA fabric state

Low-power core state





Stream-Dataflow Execution Model Detailed Example

UCLA

Legend:

Enqueued	Barrier
Dispatched	Dependency
Resource idle	Iter. boundary
Resource in use	
All data at dest.	

C[i] = A[i] * B[i]

Stream Commands

Program Order

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

C4) Mem → Port B

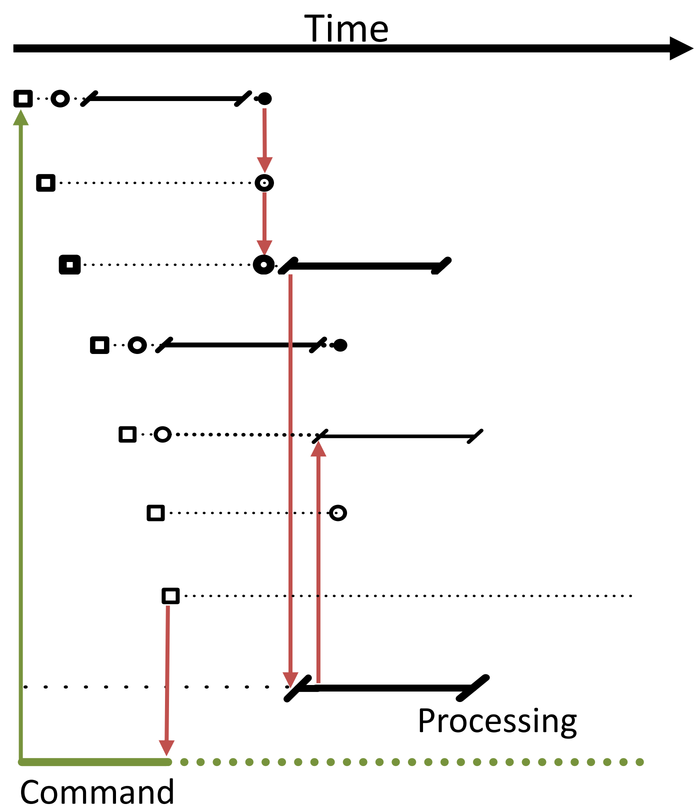
C5) Port C → Mem

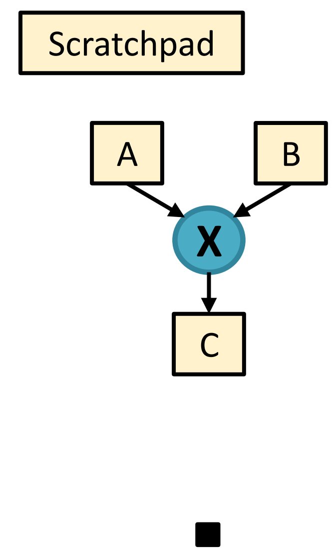
C6) Mem → Port B

C7) All Barrier

CGRA fabric state

Low-power core state



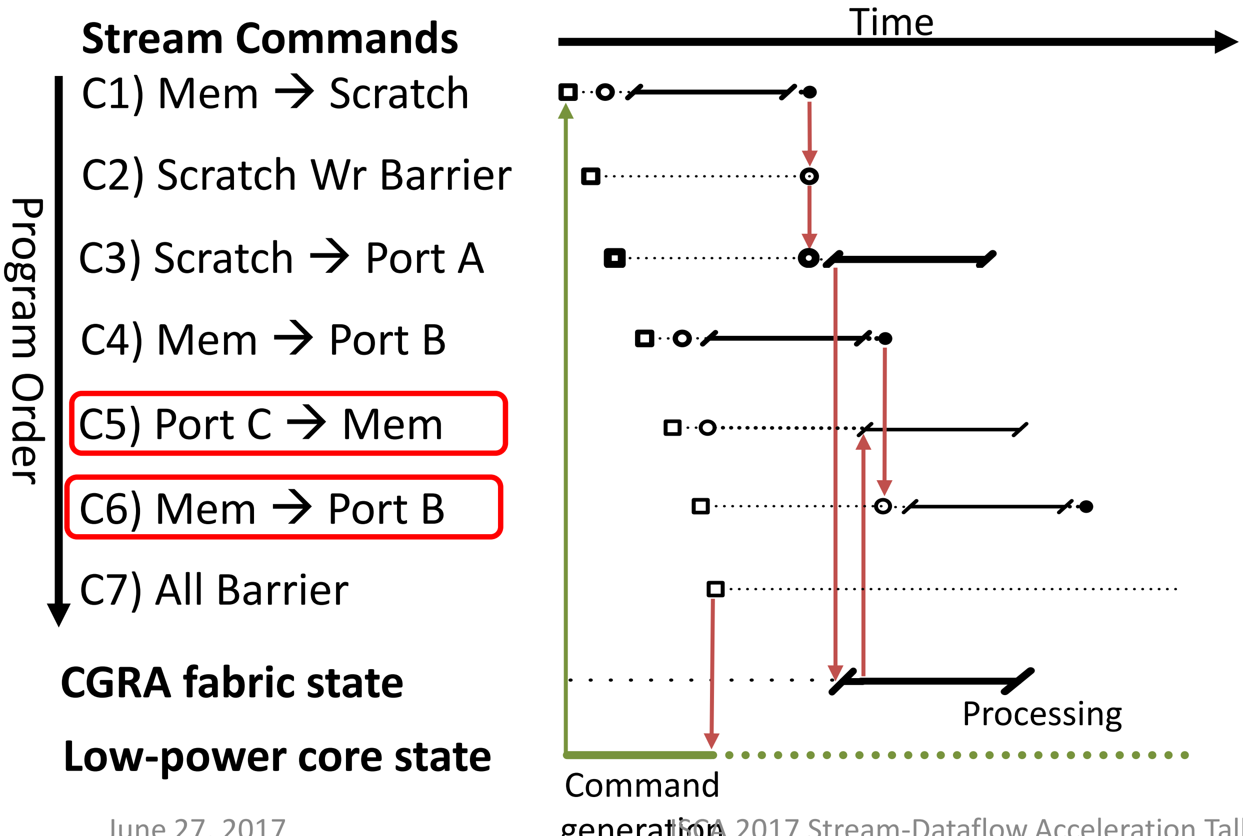


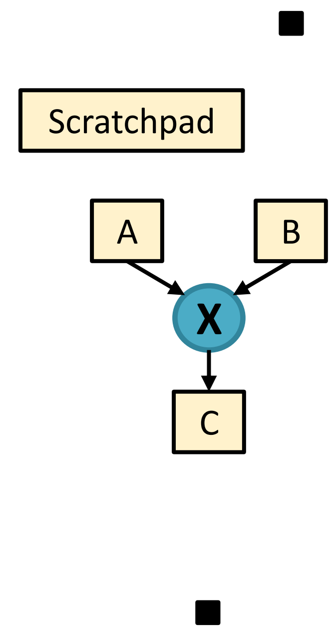
Stream-Dataflow Execution Model Detailed Example

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]



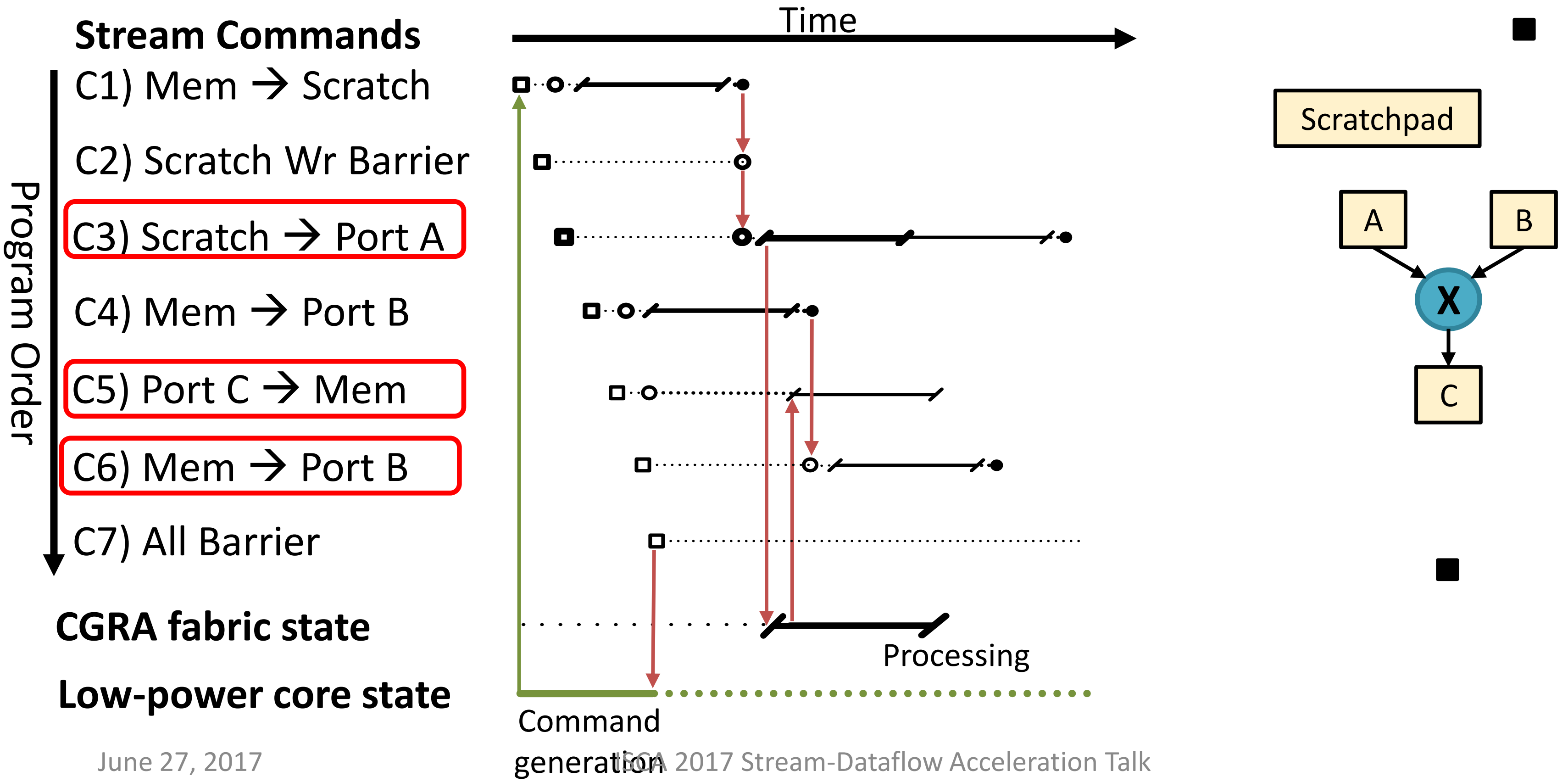


Stream-Dataflow Execution Model Detailed Example

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]



Stream-Dataflow Execution Model Detailed Example

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	⁄
Resource in use	——		
All data at dest.	●		

Stream Commands

Program Order

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

C4) Mem → Port B

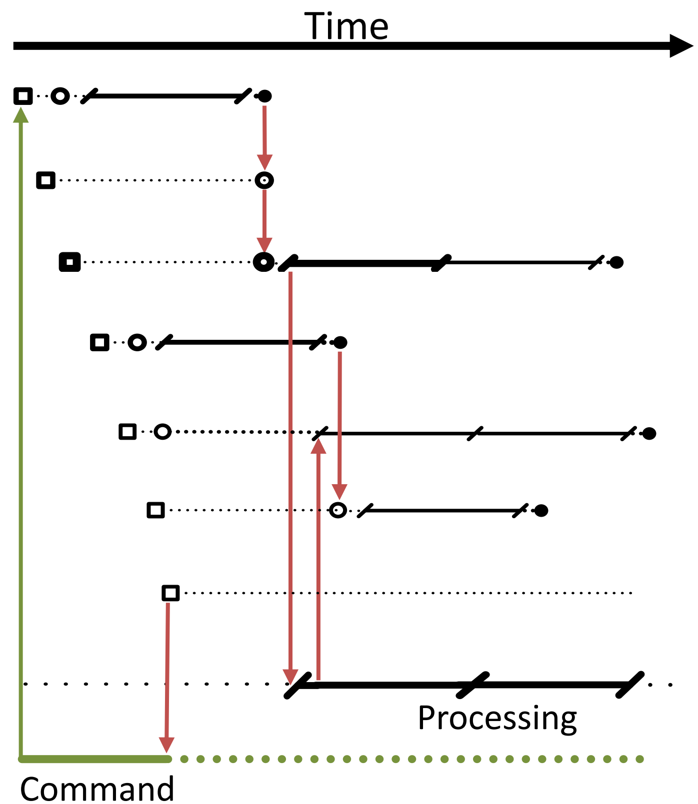
C5) Port C → Mem

C6) Mem → Port B

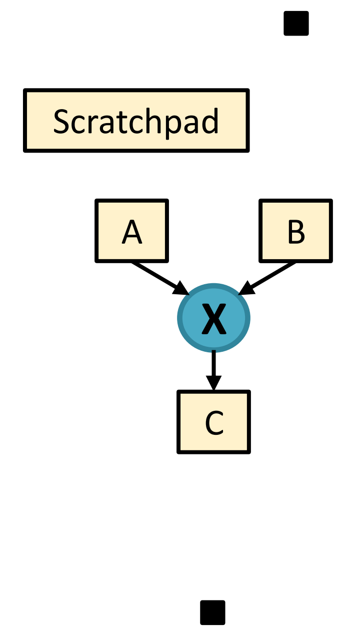
C7) All Barrier

CGRA fabric state

Low-power core state



C[i] = A[i] * B[i]



Stream-Dataflow Execution Model Detailed Example

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]

Program Order

Stream Commands

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

C4) Mem → Port B

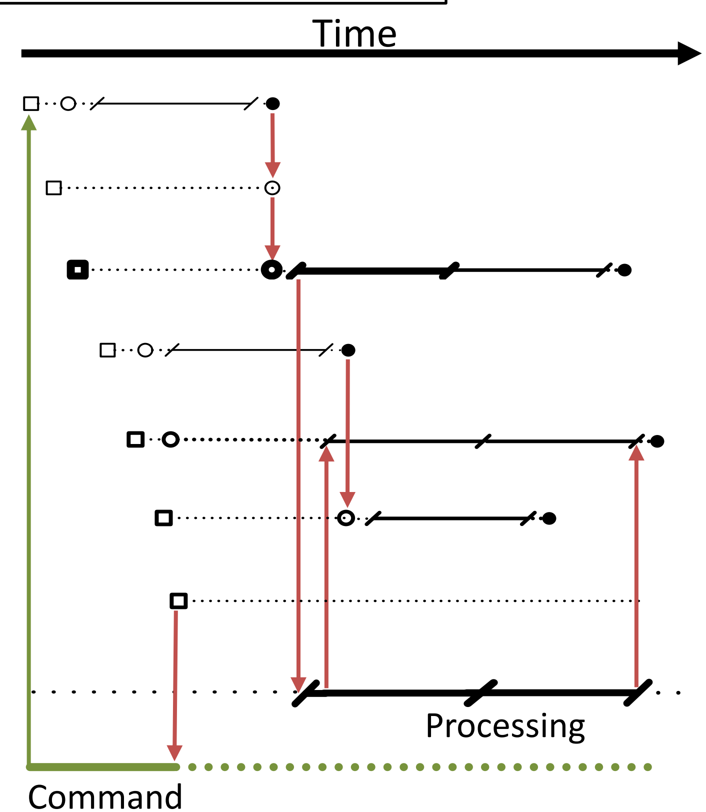
C5) Port C → Mem

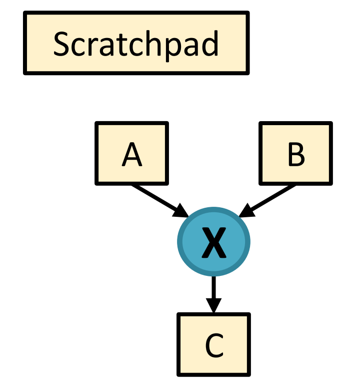
C6) Mem → Port B

C7) All Barrier

CGRA fabric state

Low-power core state





Stream-Dataflow Execution Model Detailed Example

UCLA

Legend:

Enqueued	□	Barrier	⊙
Dispatched	○	Dependency	→
Resource idle	·······	Iter. boundary	/
Resource in use	——		
All data at dest.	●		

C[i] = A[i] * B[i]

Stream Commands

Program Order

C1) Mem → Scratch

C2) Scratch Wr Barrier

C3) Scratch → Port A

C4) Mem → Port B

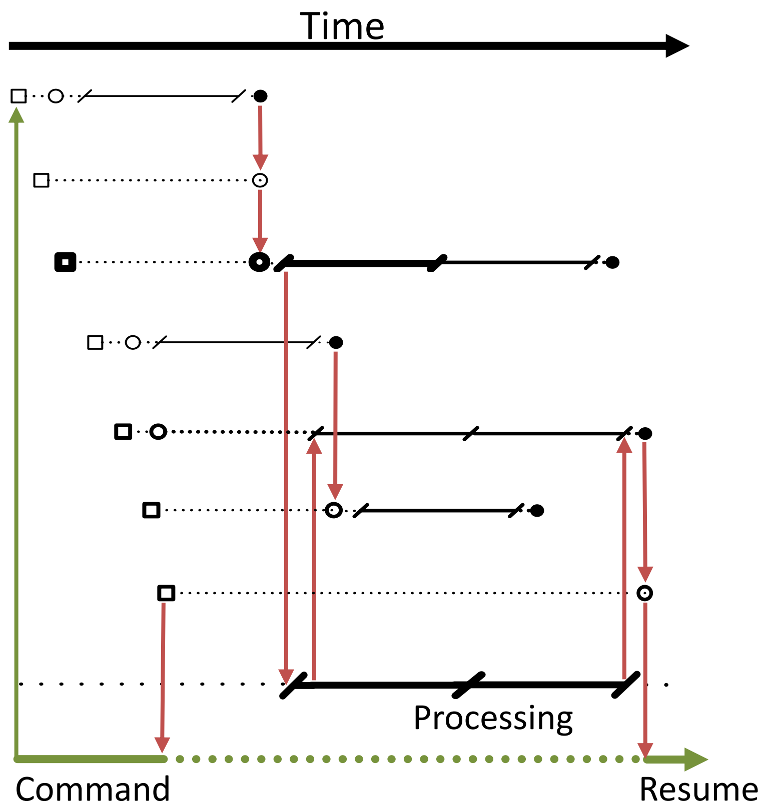
C5) Port C → Mem

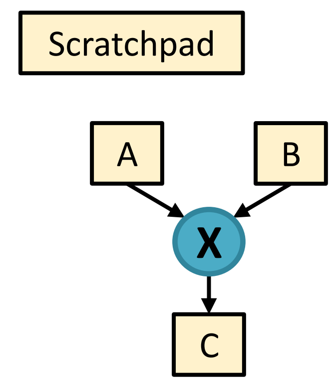
C6) Mem → Port B

C7) All Barrier

CGRA fabric state

Low-power core state





Stream-Dataflow Execution Model Detailed Example

Legend:

Enqueued □ | Barrier ⊙
Dispatched ○ | Dependency →
Resource idle ······· | Iter. boundary /

Stream-Dataflow Accelerator Potential

1. Dataflow based pipelined concurrent execution

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

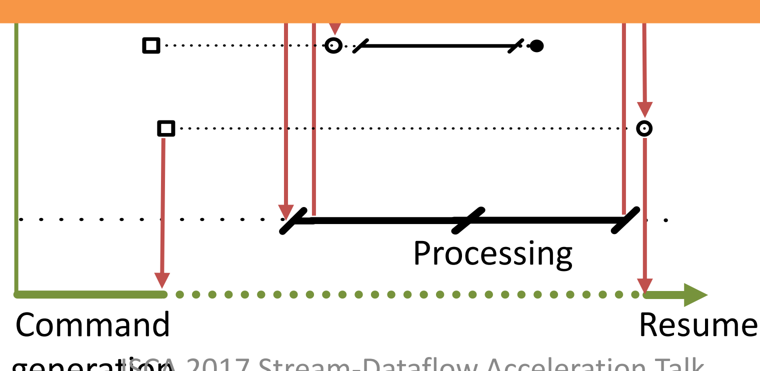
Program Order

C6) Mem → Port B

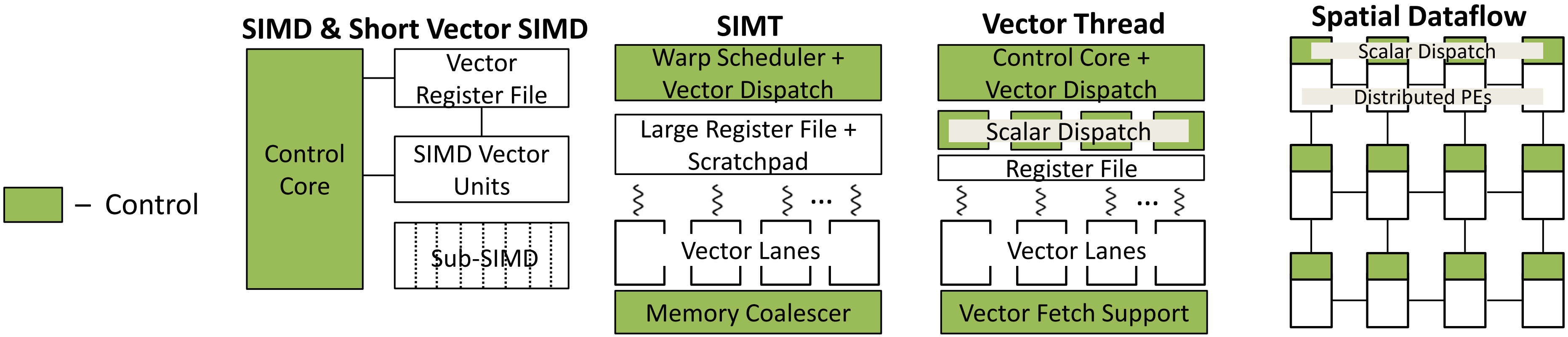
C7) All Barrier

CGRA fabric state

Low-power core state



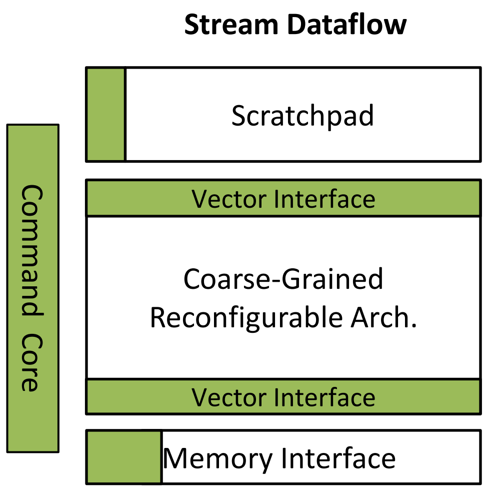
Inefficiencies in Data-Parallel Architectures



Addressing & Communication	• Unaligned addressing • Complex scatter-gather • Mask & merge instructions	• Redundant address generation • Address coalescing across threads • Non-decoupled access-execute phases	• Redundant address generation	• Redundant address generation • Inefficient memory b/w for local accesses
Resource Utilization & Latency hiding	• Core-issue width • Fixed vector width • Core to reorder instructions	• Thread scheduling • Multi-ported large register file & cache pressure	• Redundant dispatchers • Core issue width and re-ordering	• Redundant dispatch
Irregular execution support	• Inefficient general pipeline	• Warp divergence hardware support	• Re-convergence for diverged vector threads	-

Stream-Dataflow Accelerator Architecture Opportunities

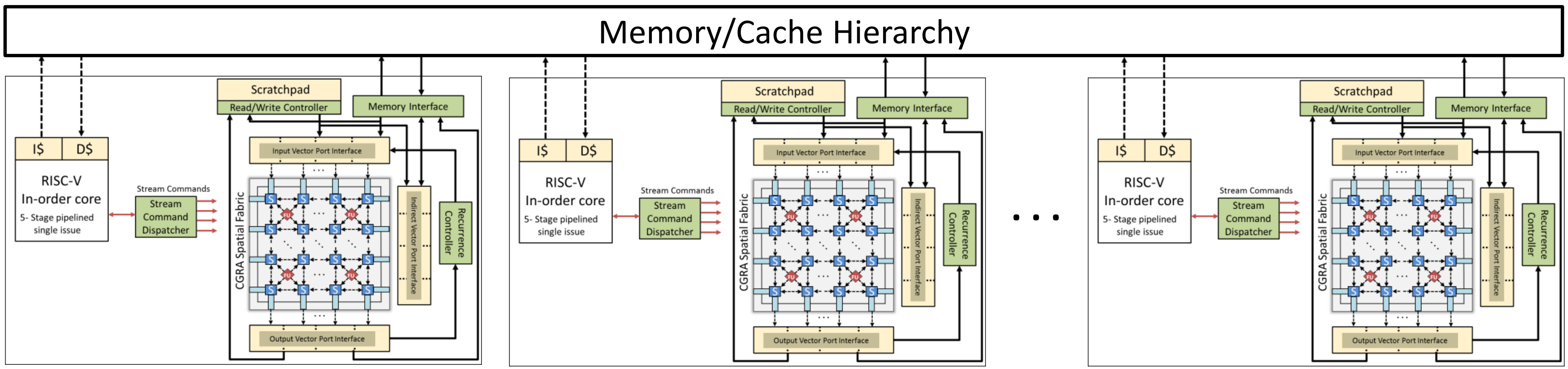
- Reduce address generation & duplication overheads
- Distributed control to boost pipelined concurrent execution
- High utilization of execution resources w/o massive multi-threading, reducing cache pressure or using multi-ported scratchpad
- Decouple access and execute phases of programs
- Able to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



Stream-Dataflow Accelerator Architecture

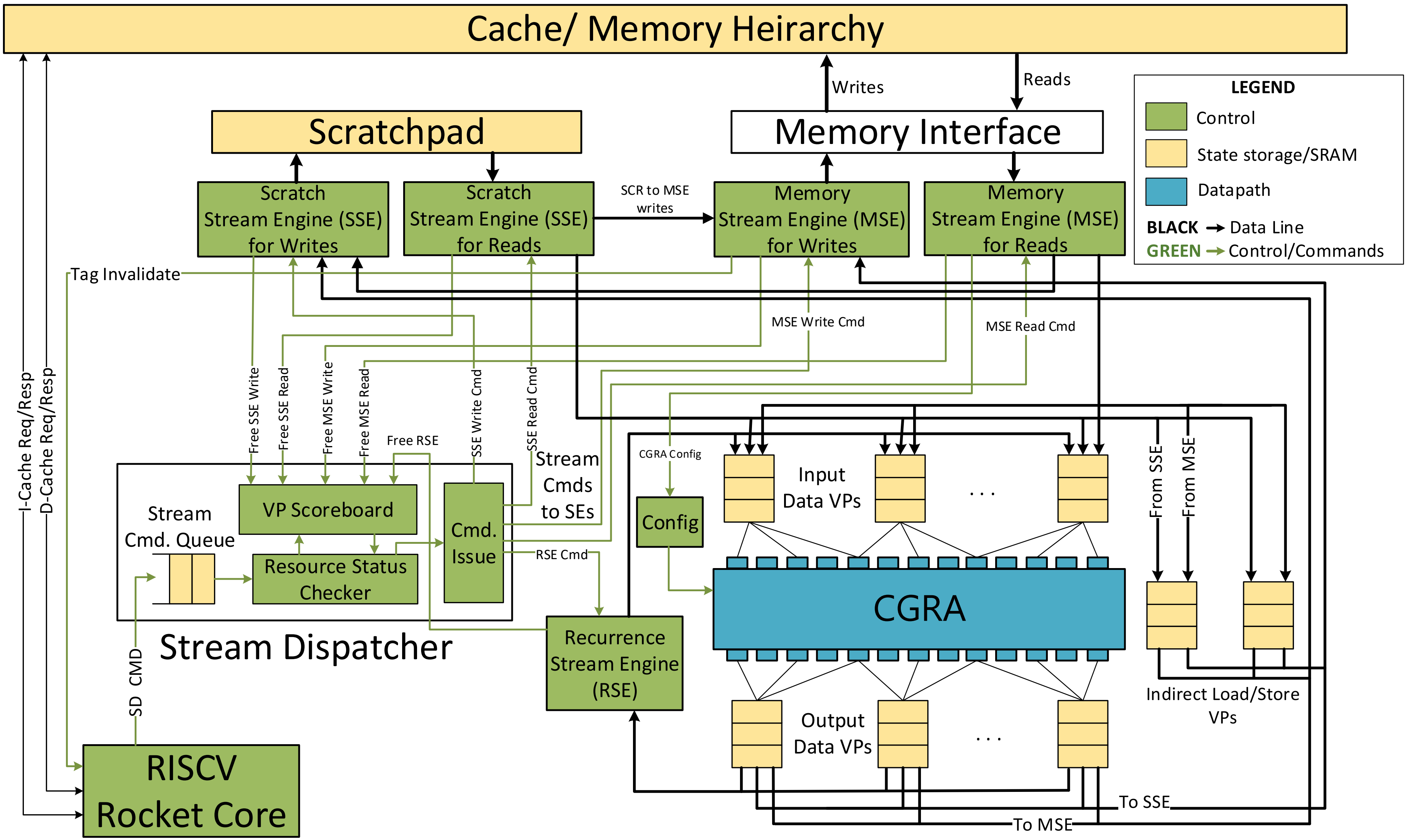
512b 64b Stream Command

Multi-Tile Stream-Dataflow Accelerator

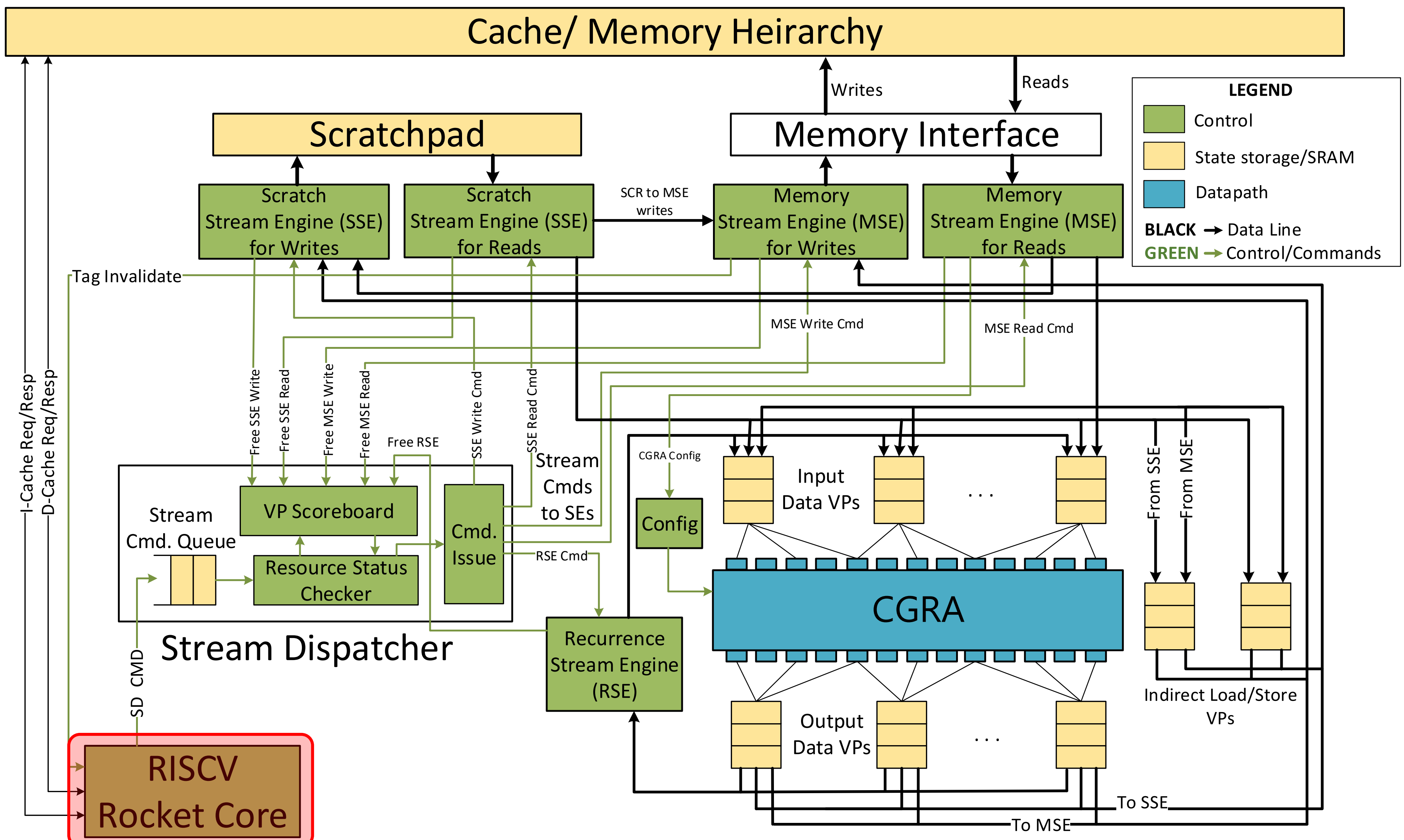


- Each tile is connected to higher-L2 cache interface
- Need a simple scheduler logic to schedule the offloaded stream-dataflow kernels to each tile

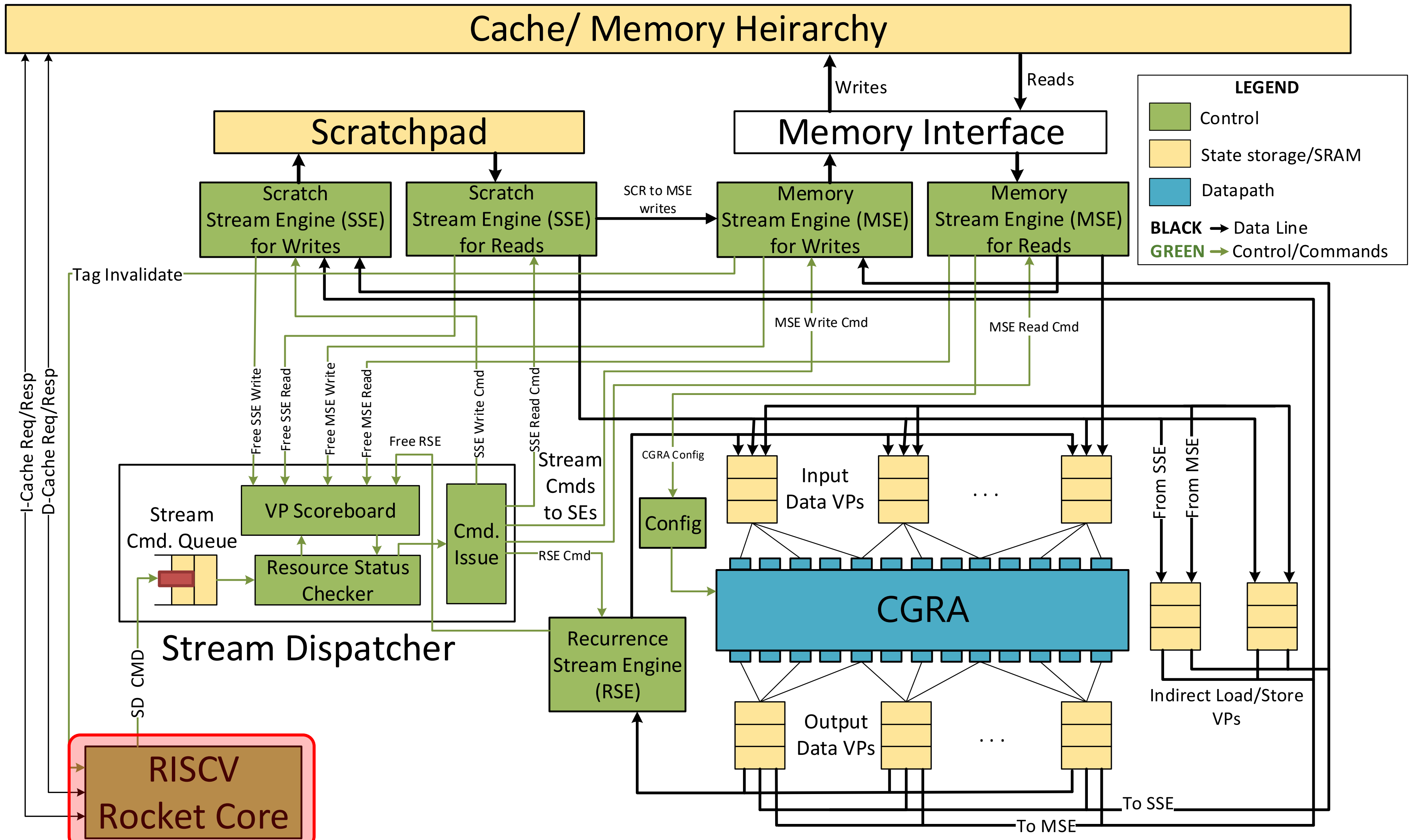
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



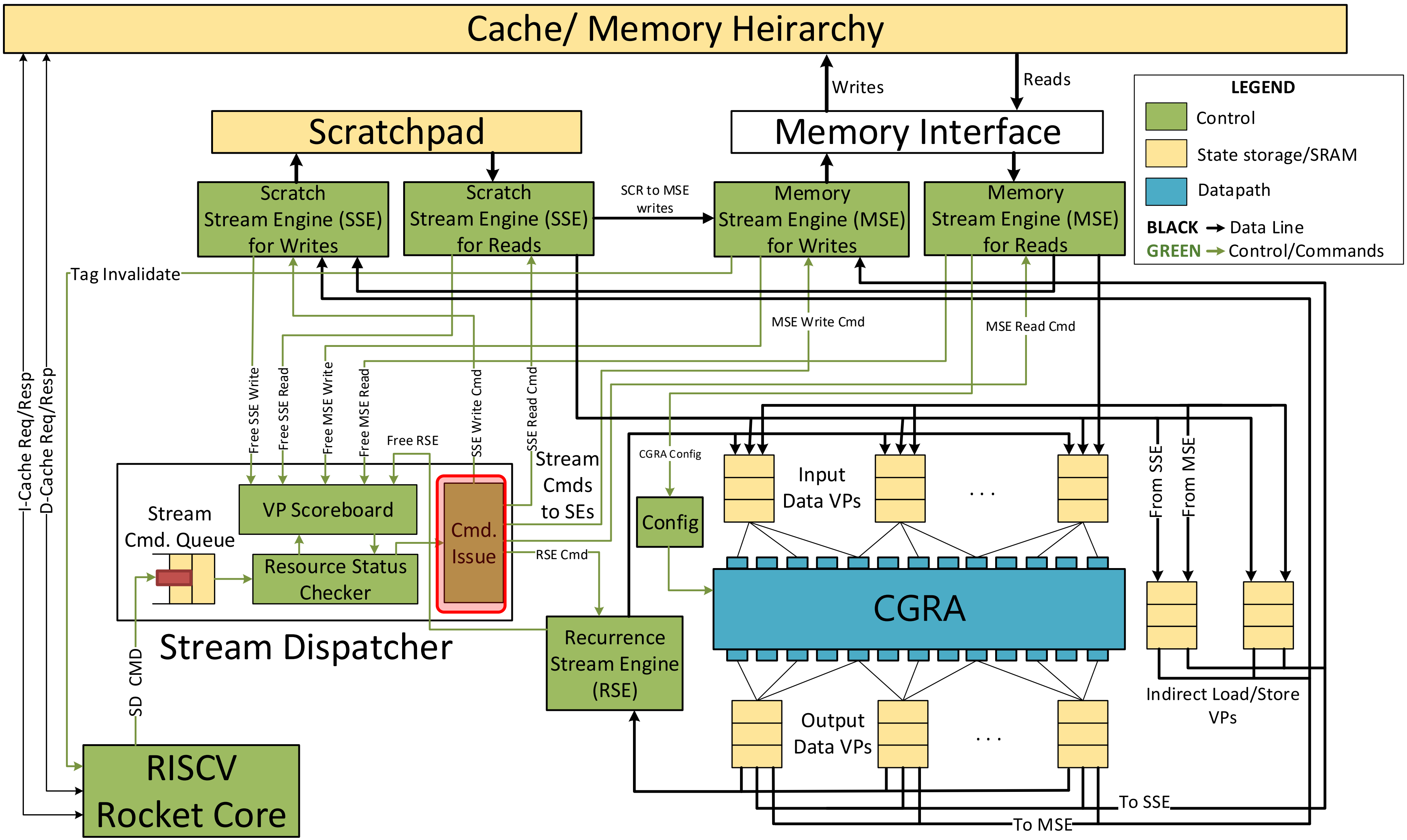
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



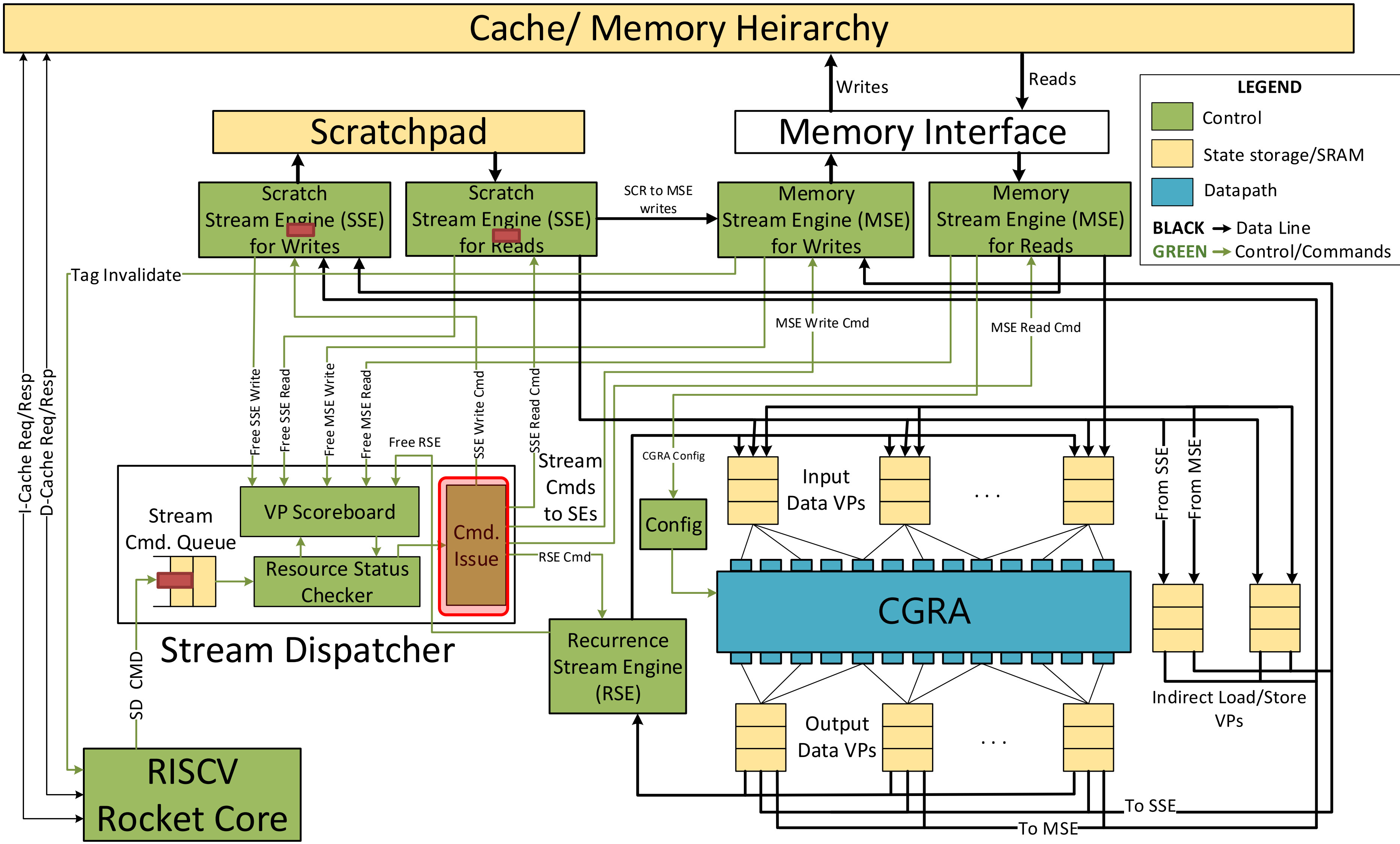
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



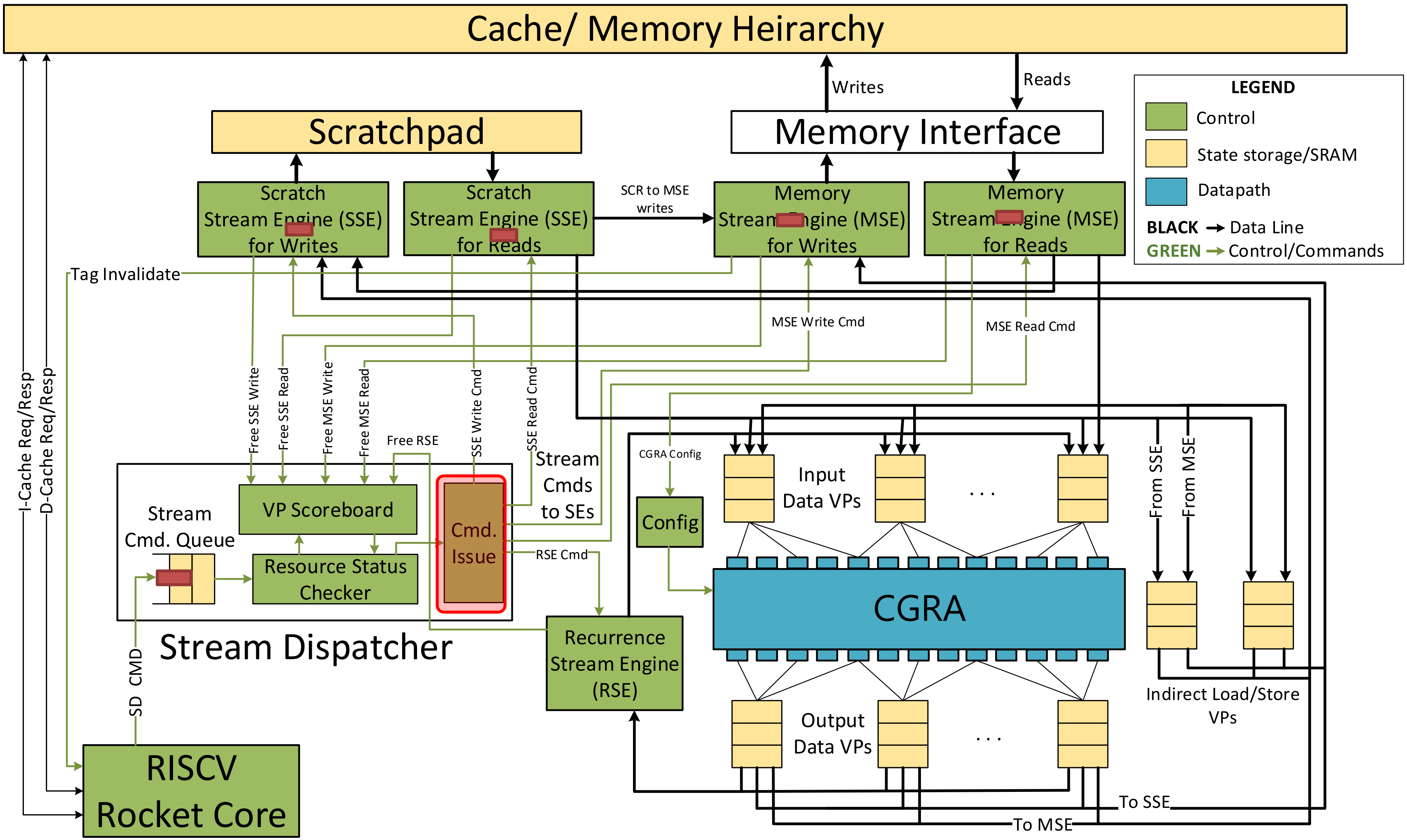
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



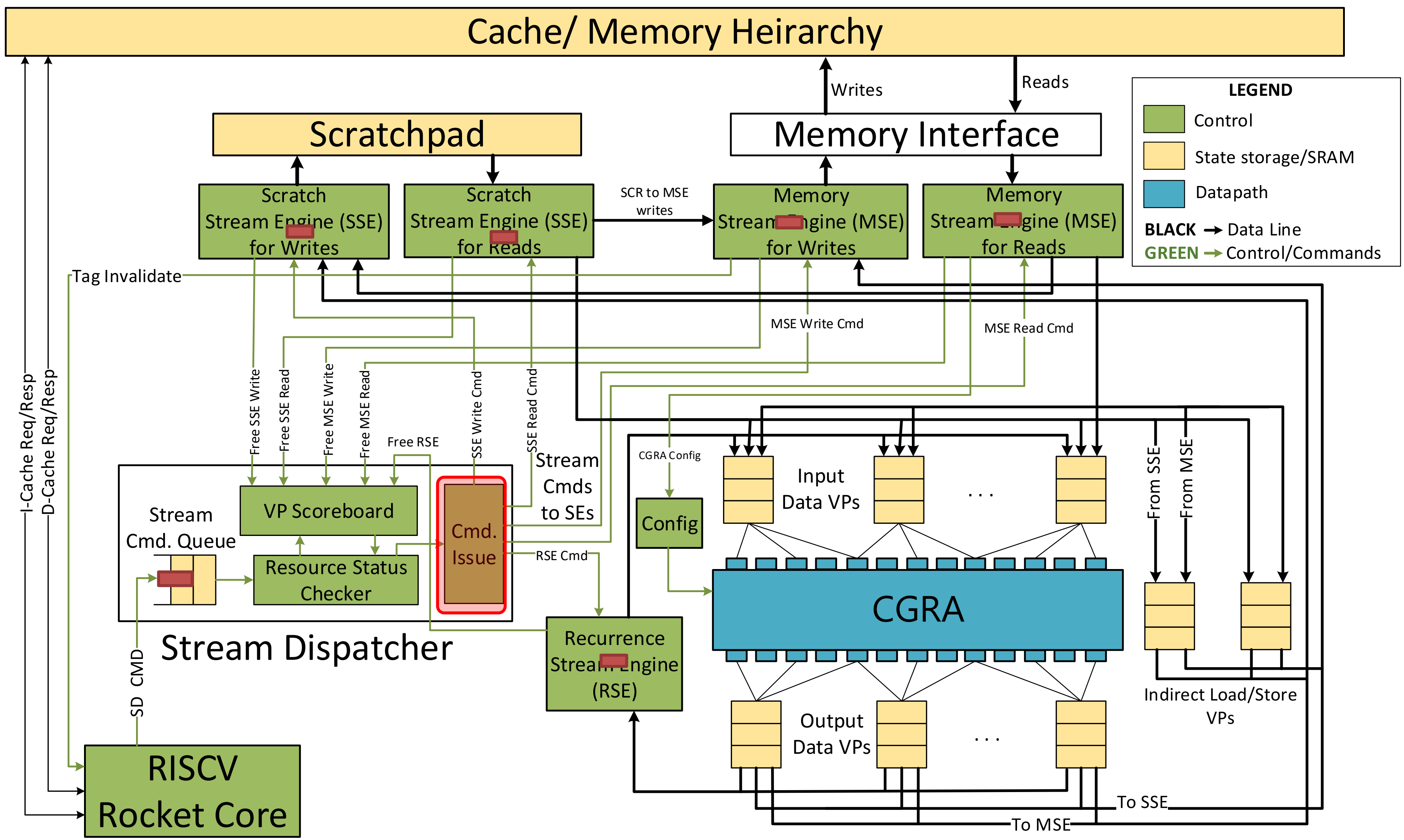
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



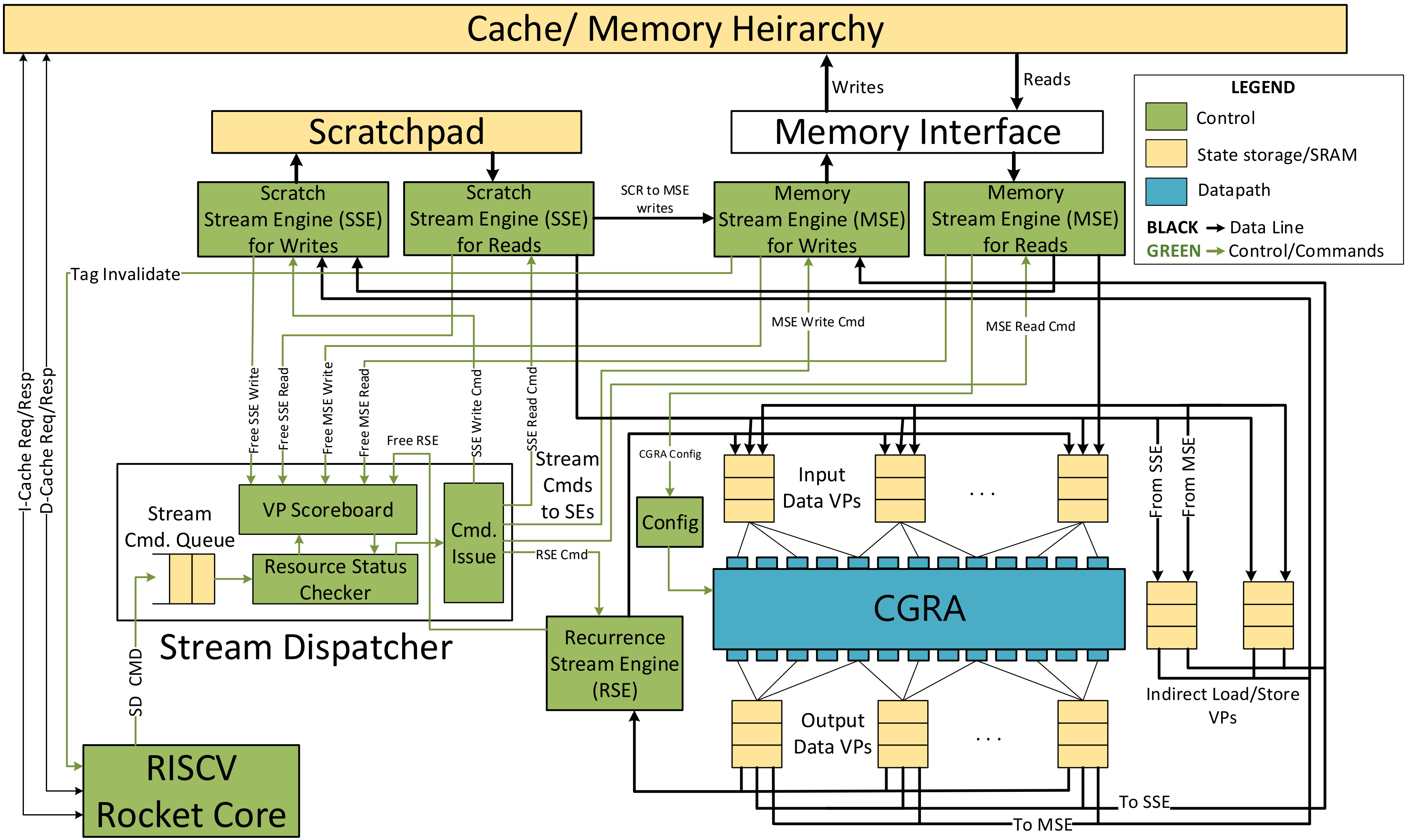
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



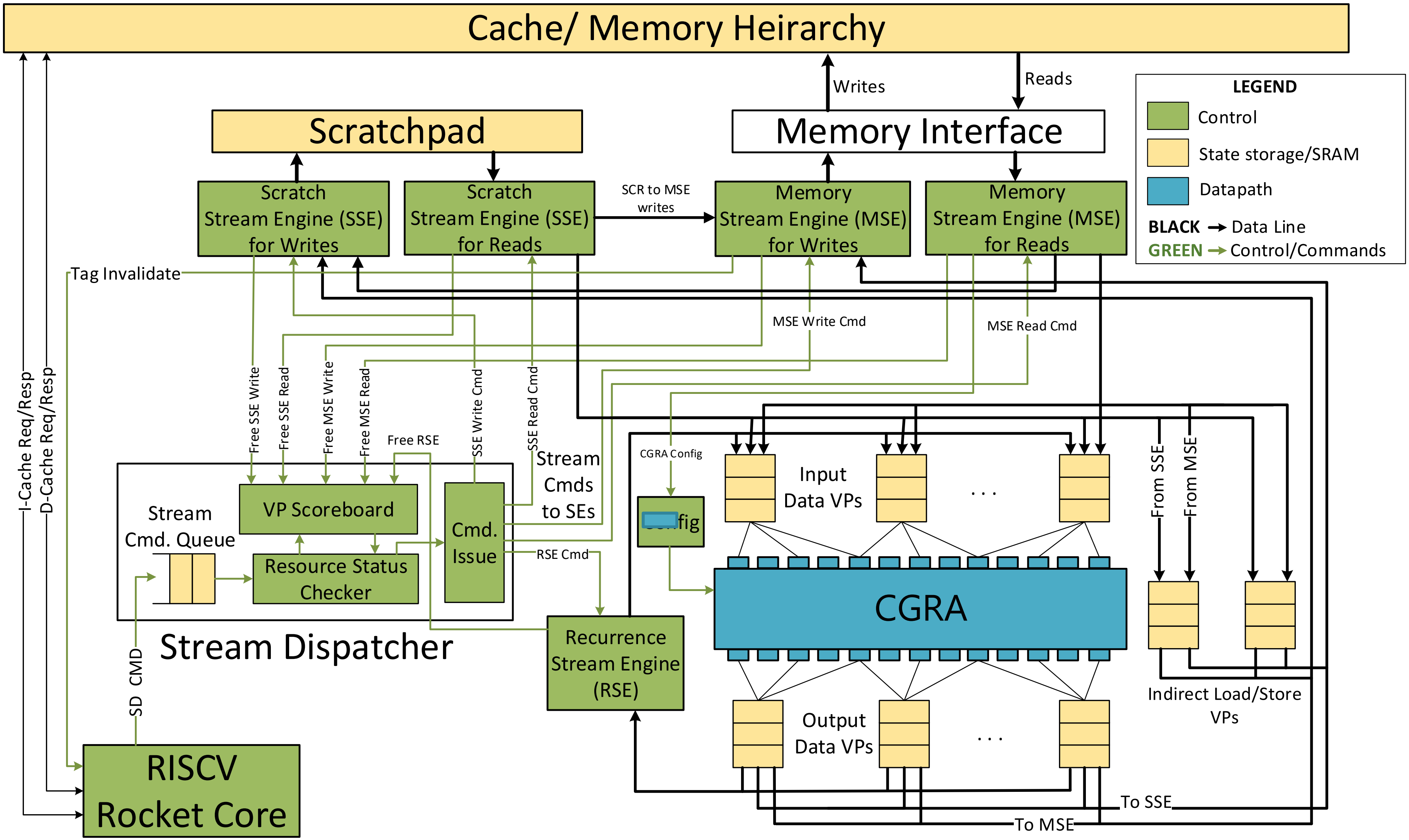
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



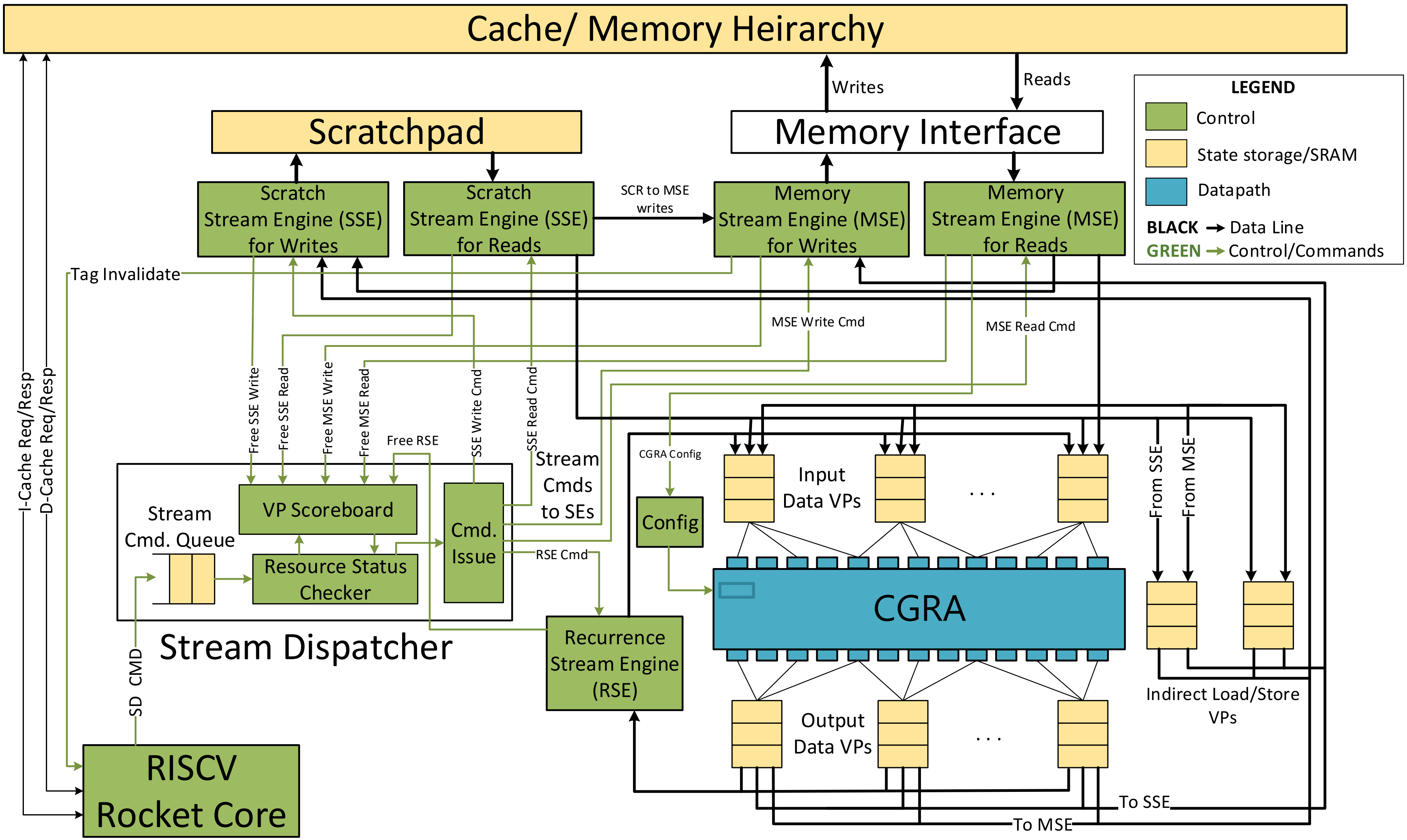
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



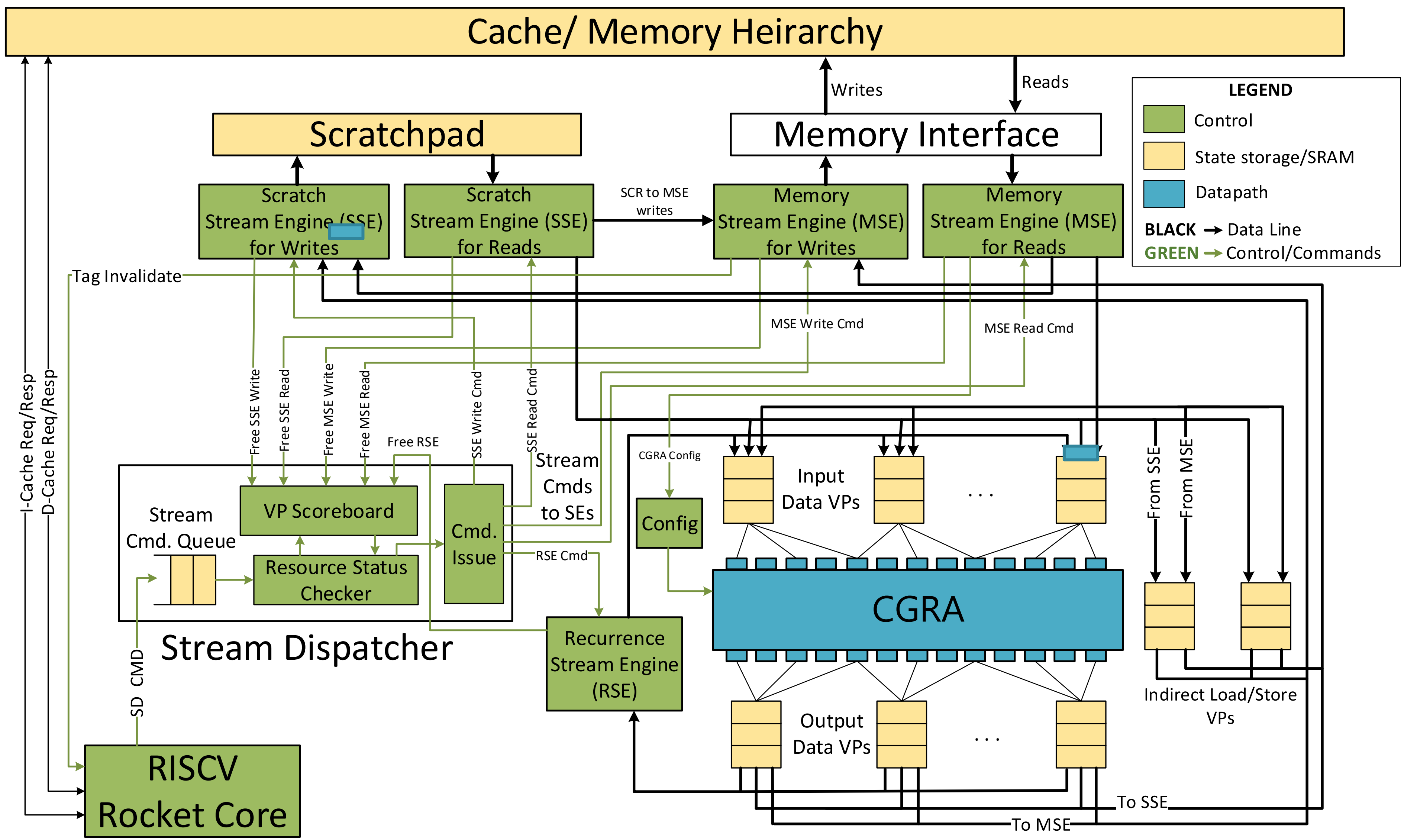
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



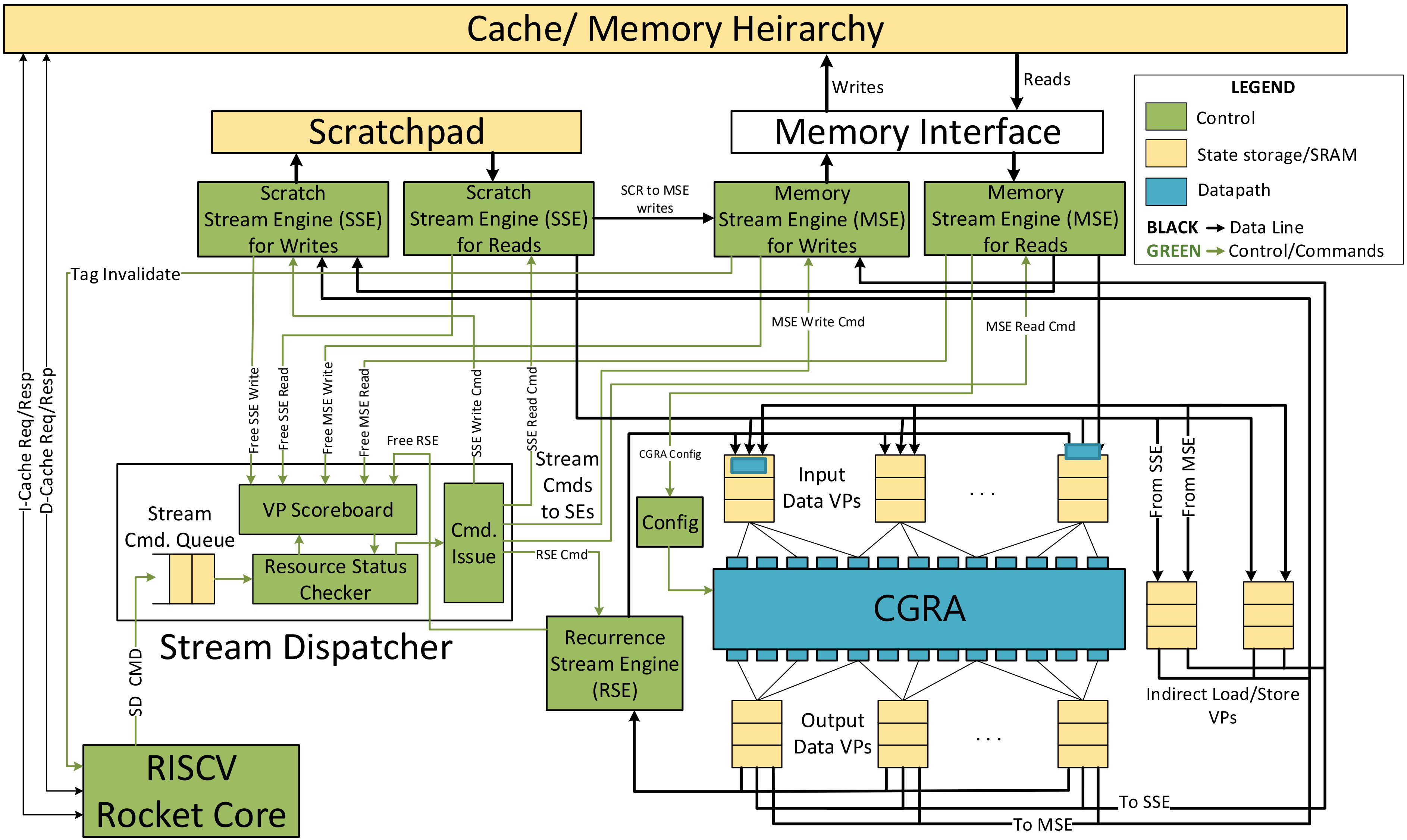
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



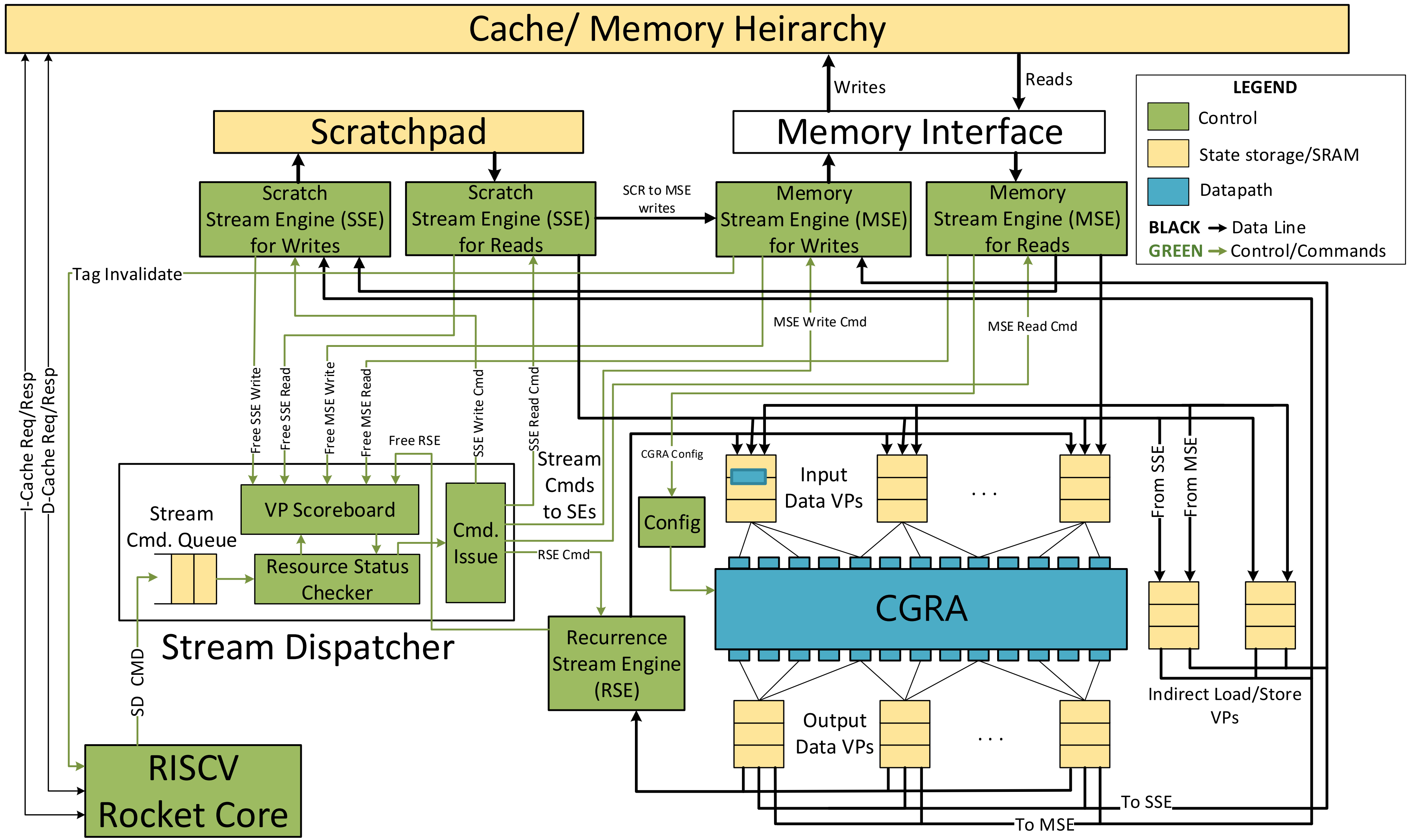
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



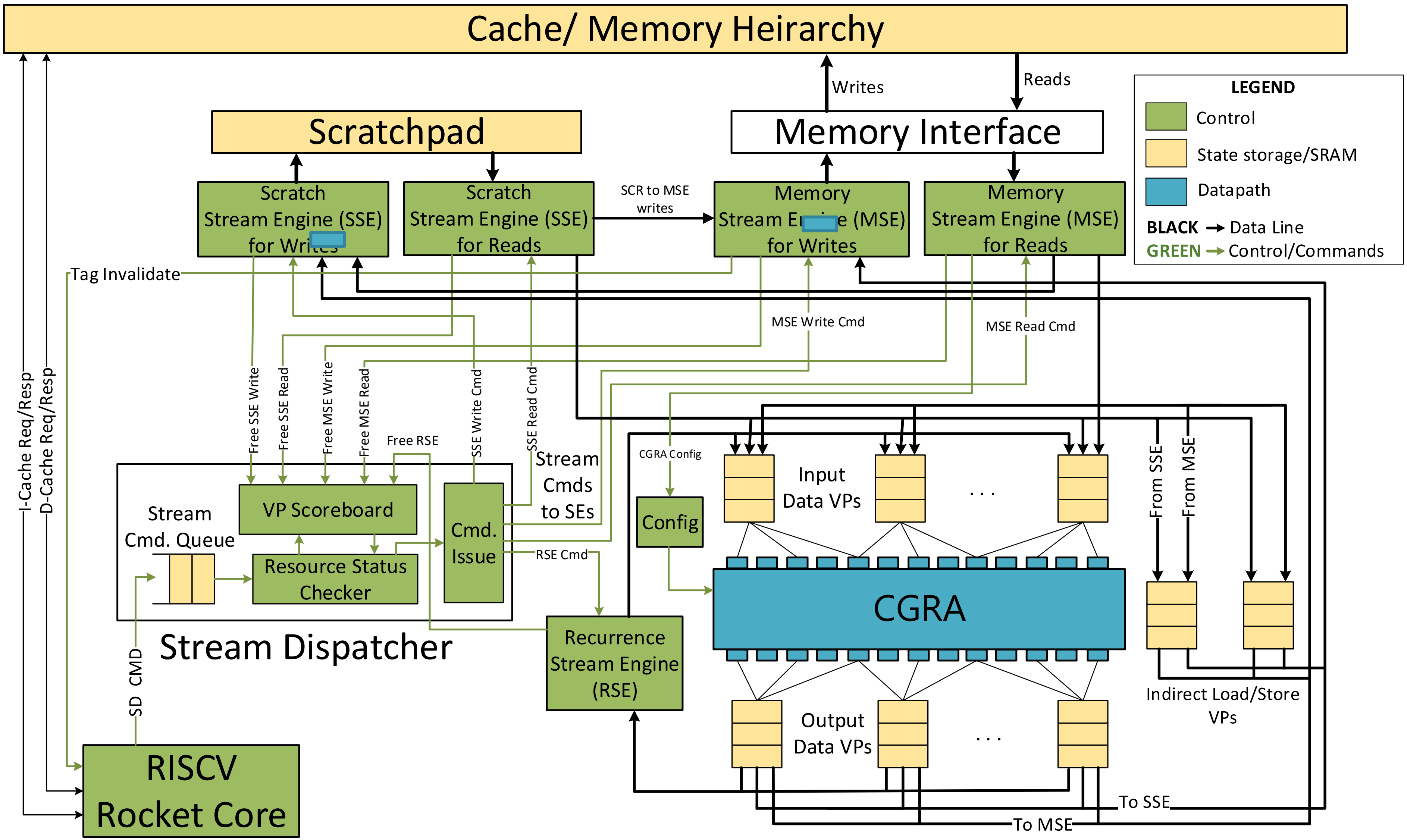
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



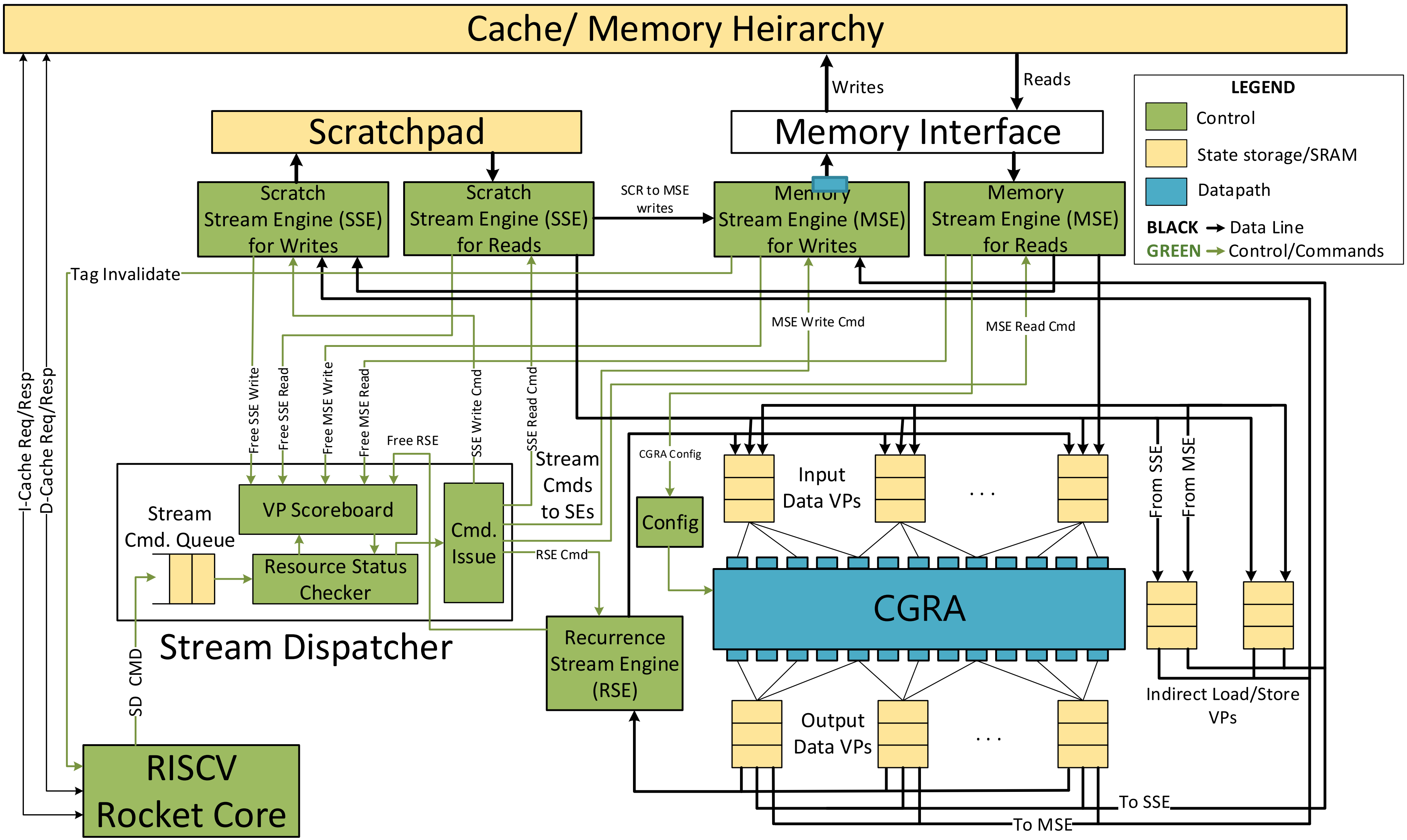
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



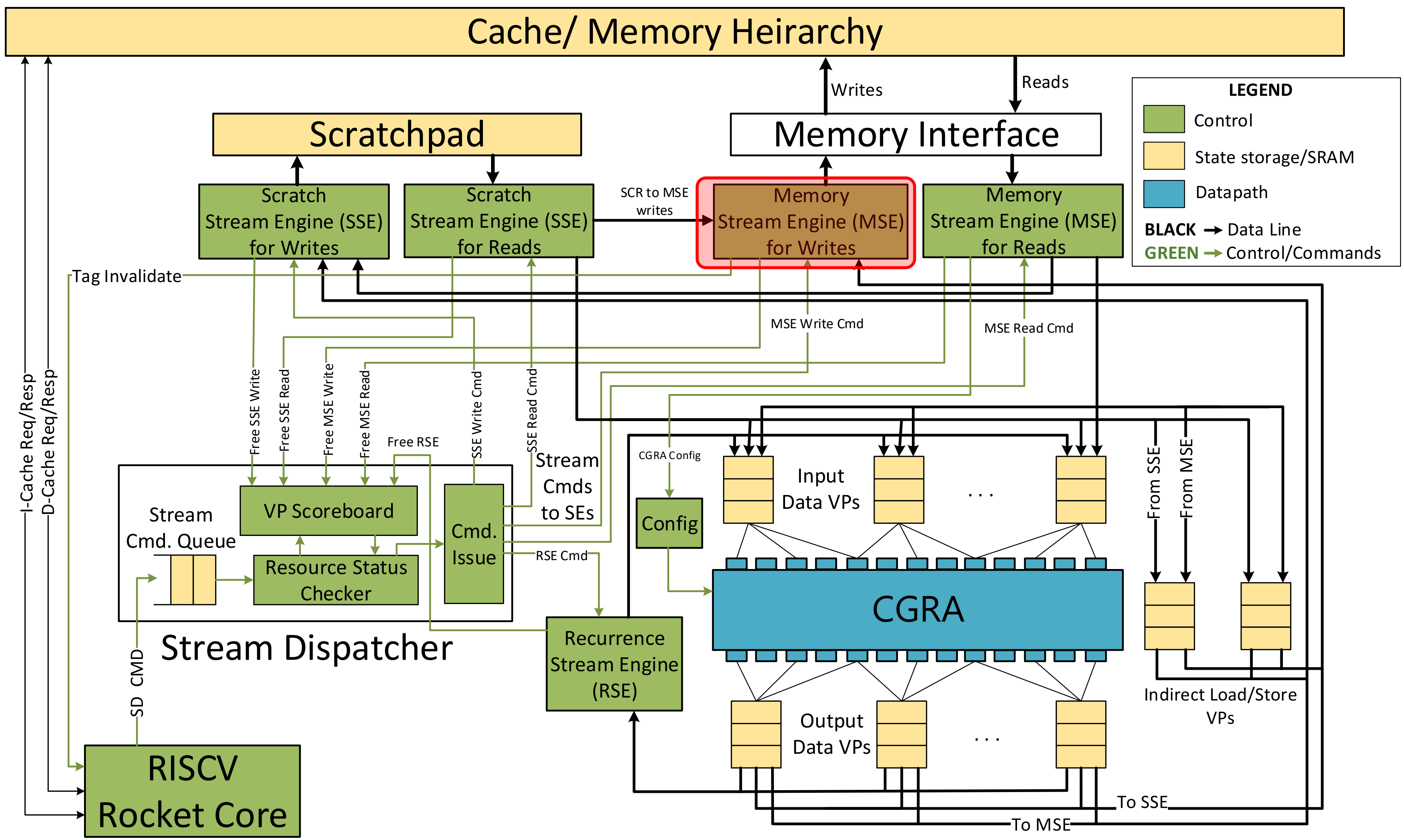
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



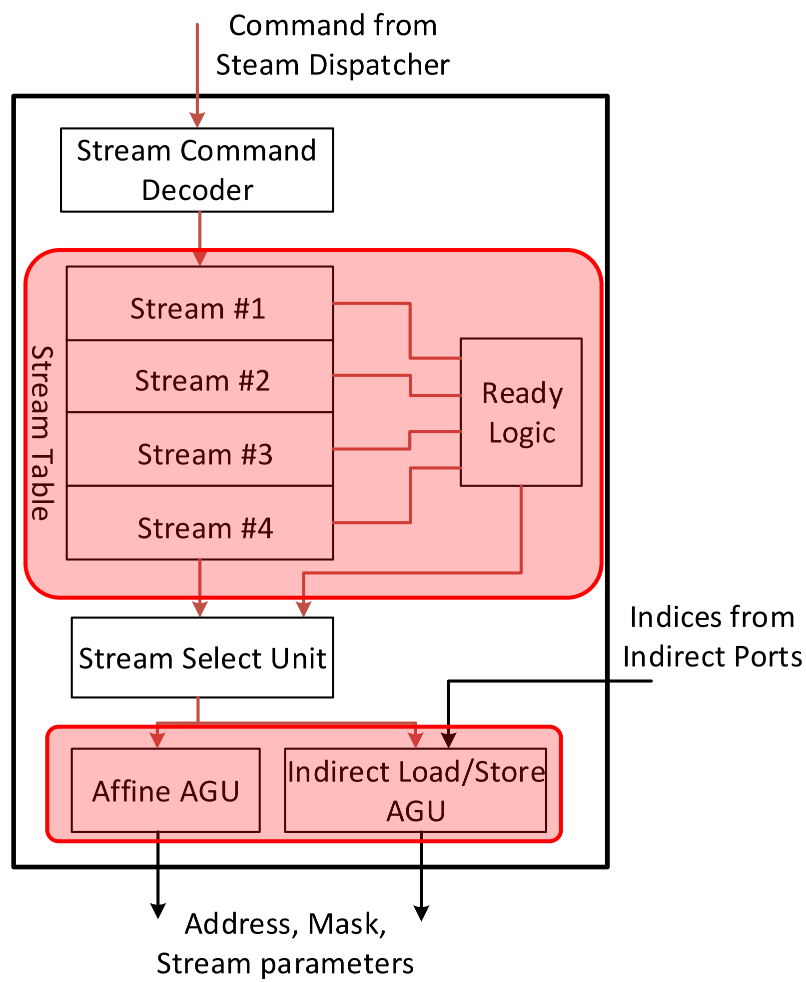
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)



Softbrain Stream Engine Request Pipeline

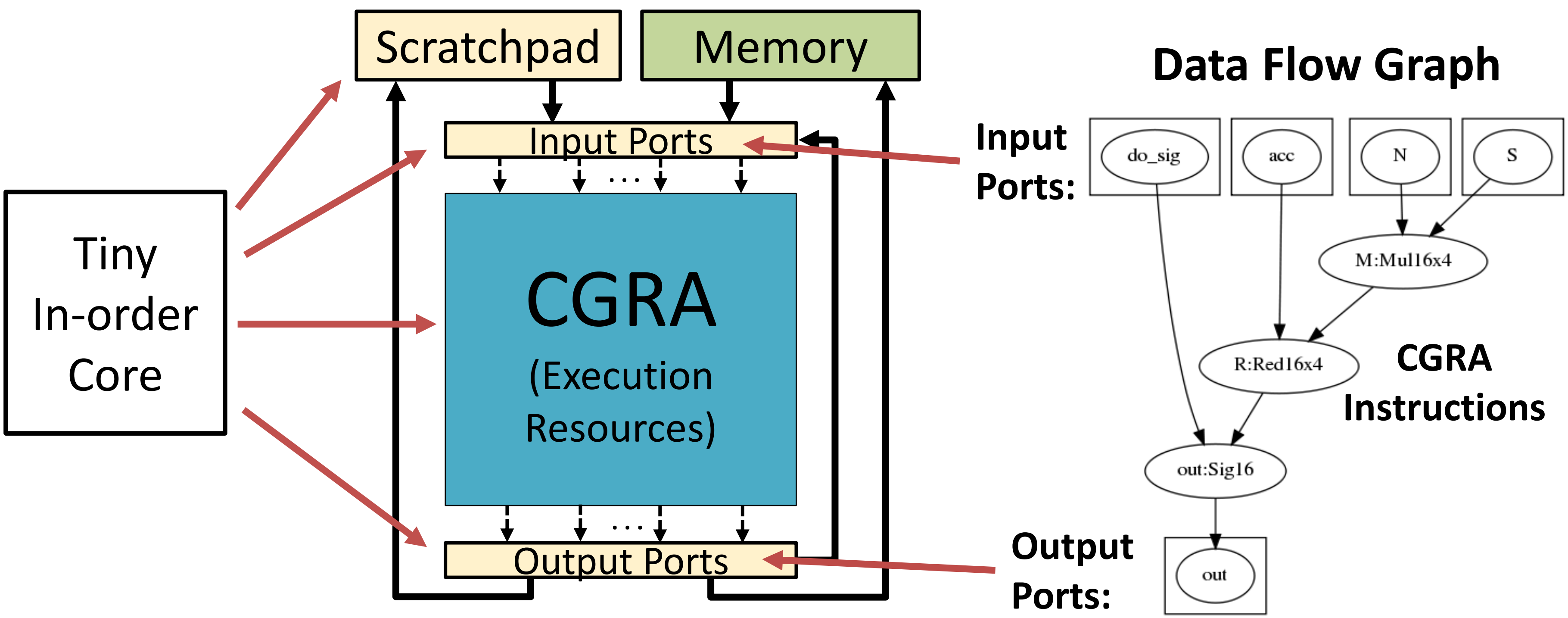
- Responsible for address generation for both *affine* and *non-affine* data-streams
- Priority based selection among multiple queued data-steams
- Affine streams – Affine Address Generation Unit (AGU) generates memory addresses
- Non-affine AGU gets addresses and offsets from indirect vector ports
- Similar stream request pipeline is used for scratchpad stream-engines with minimal changes

Stream Request Pipeline



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

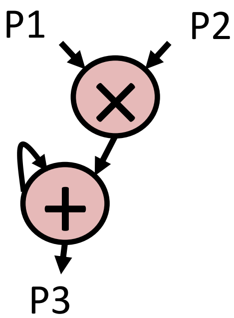


Example Code: Dot Product

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
```

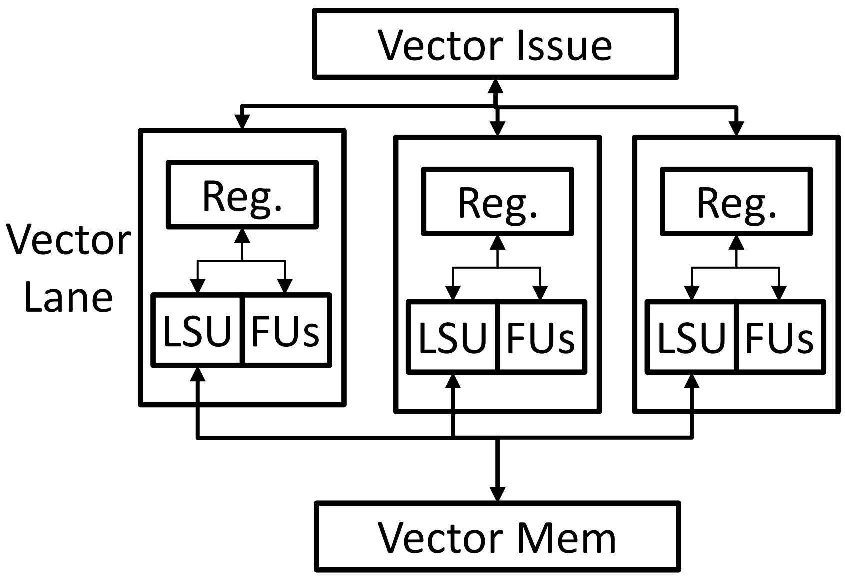
~3 Instructions

Existing Architectures for Data Parallel

UCLA

Vector Processor

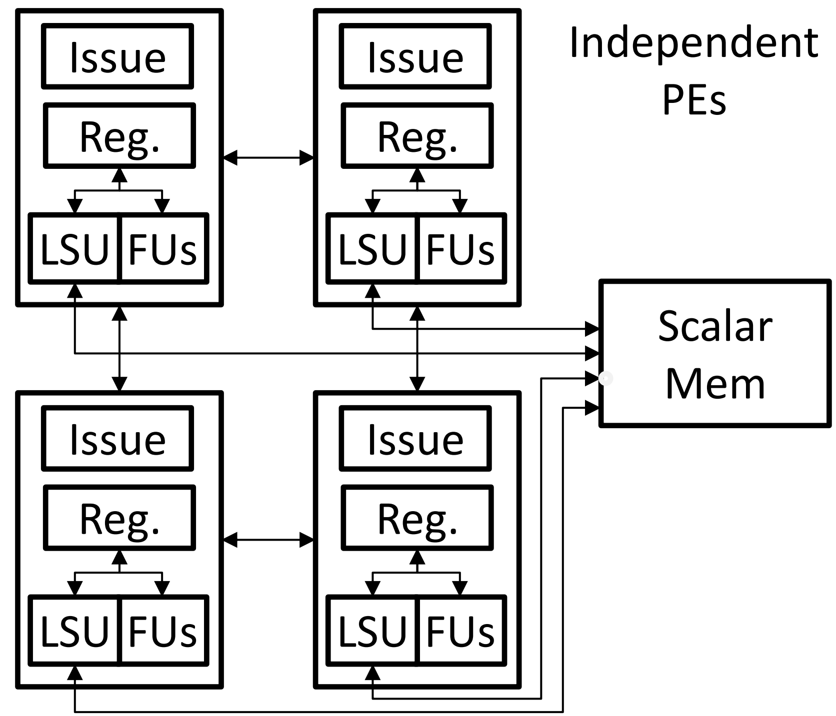
(eg. ARM Neon, X86 SSE)



- Amortized Instruction Issue
- Efficient Vector-Memory

Spatial Processor

(eg. Tilera, TRIPS, Wavescalar)



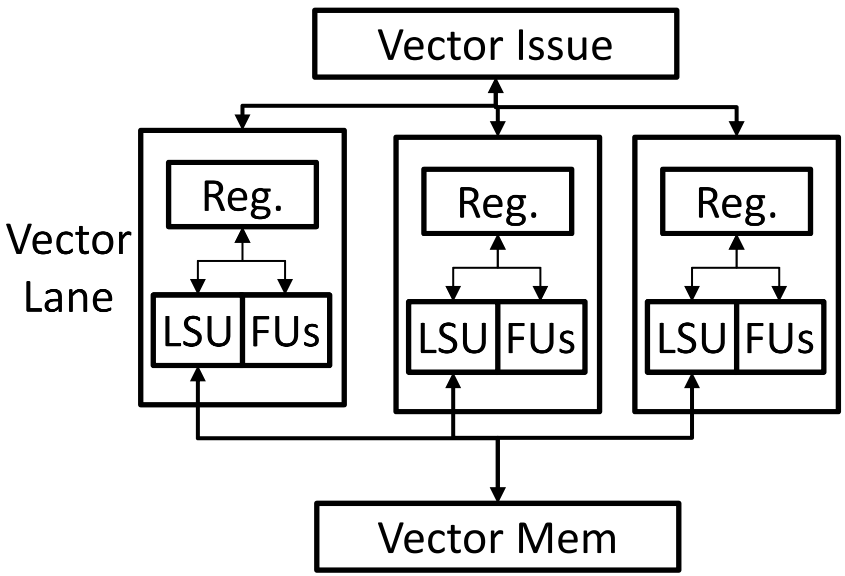
- Efficient Dataflow b/t Units
- Flexible Computation Patterns

Existing Architectures for Data Parallel

UCLA

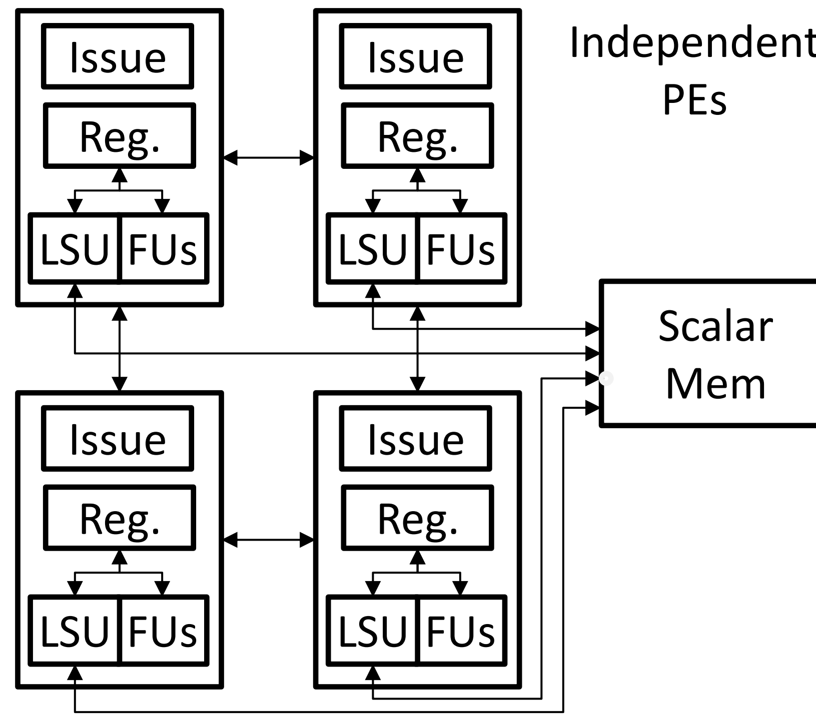
Vector Processor

(eg. ARM Neon, X86 SSE)



Spatial Processor

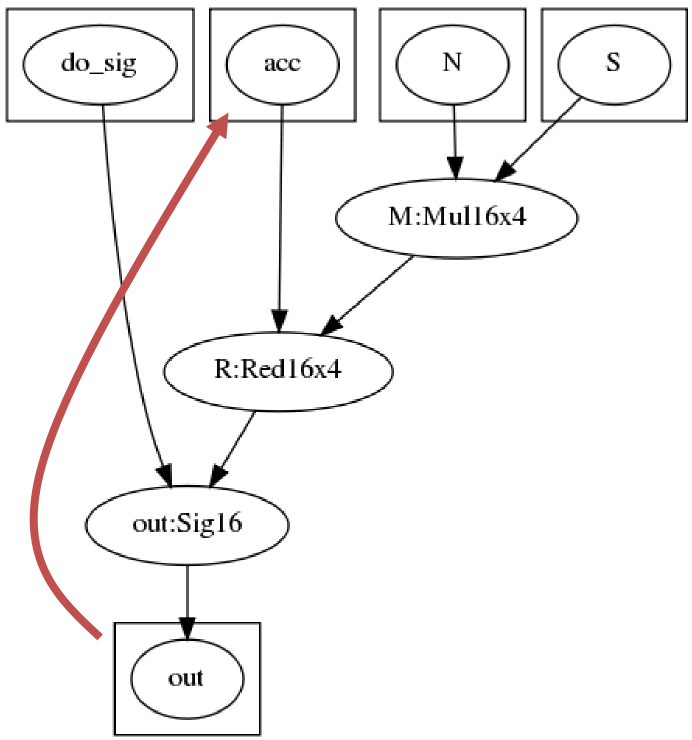
(eg. Tilera, TRIPS, Wavescalar)



Vectorized memory interface + Spatial Datapath + Amortized Issue

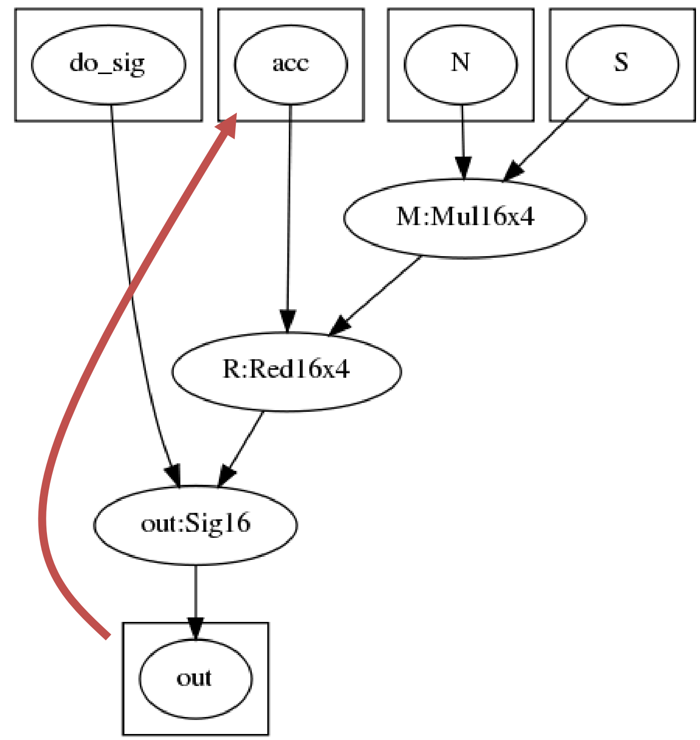
Dataflow Graph (DFG) for CGRA

```
Input: do_sig
Input: acc
Input: N
Input: S
M = Mul16x4(N, S)
R = Red16x4(M, acc)
out = Sig16(R, do_sig)
Output: out
```



Stream Dataflow Program:

```
uint16_t synapse[Nn][Ni];
uint16_t neuron_i[Ni];
uint16_t neuron_n[Nn];

SD_CONFIG(dfg_config, dfg_size);

SD_DMA_READ(synapse,  8,   8,Ni*Nn/4,P_dfg_S);
SD_DMA_READ(neuron_i, 0,Ni*2,Nn,      P_dfg_N);

for (n = 0; n < Nn/nthreads; n++) {
  SD_CONST(P_dfg_acc,0,1);
  SD_RECURRENCE(P_dfg_out,Ni/4-1,Port_acc);
  SD_CONST(P_dfg_do_sig,0,Ni/4-1);
  SD_CONST(P_dfg_do_sig,1,1);
  SD_DMA_WRITE(P_dfg_out,2,2,1,&neuron_n[n]);
}

SD_WAIT_ALL();
```



Performance Considerations

- Goal: Fully Pipeline the Largest Data Flow Graph!
- Primary Bottlenecks:
  - Size of Data Flow Graph

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

    Increase "length" of streams
  - Memory/Cache Bandwidth

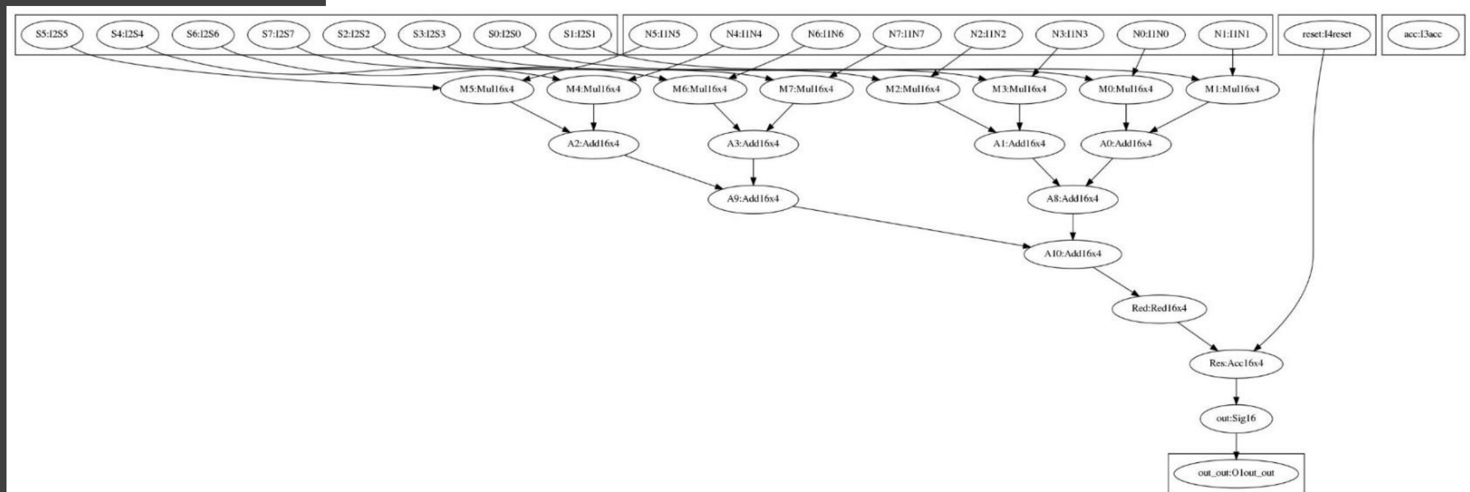
    Use Scratchpad for reused Data
  - Recurrence Serialization Overhead

    Either: 1. Increase Parallel Computations (tiling)

    2. Use internal accumulation

Optimized DFG

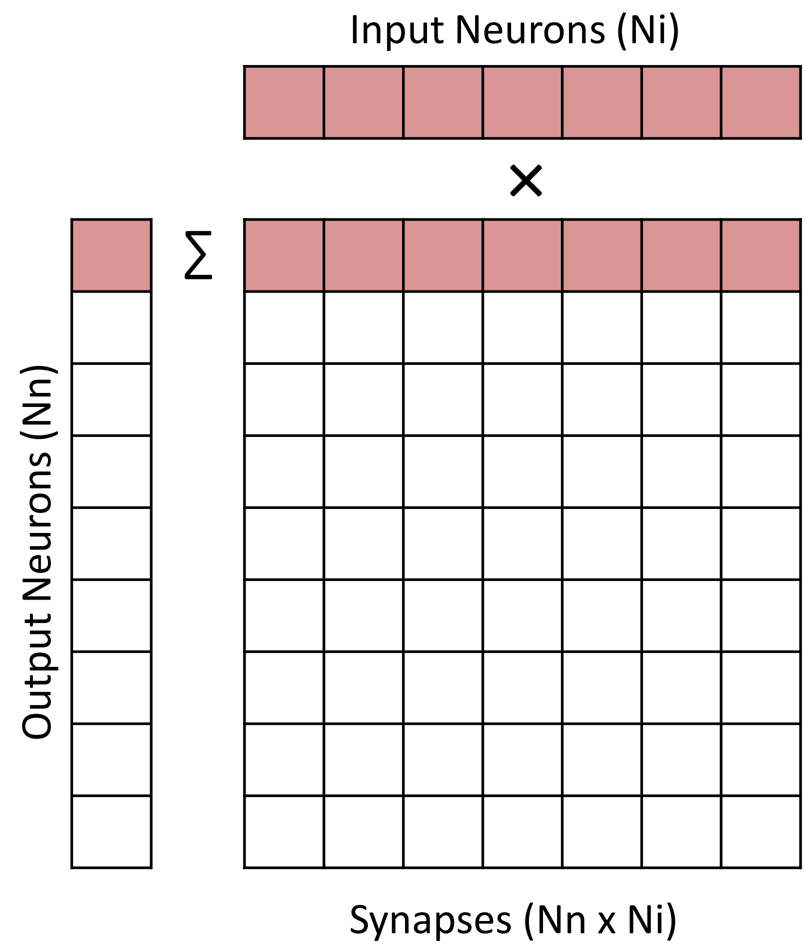

```
InputVec: N [0, 1, 2, 3, 4, 5, 6, 7]
InputVec: S [0, 1, 2, 3, 4, 5, 6, 7]
Input: reset

M0 =Mul16x4(N0, S0)
M1 =Mul16x4(N1, S1)
M2 =Mul16x4(N2, S2)
M3 =Mul16x4(N3, S3)
M4 =Mul16x4(N4, S4)
M5 =Mul16x4(N5, S5)
M6 =Mul16x4(N6, S6)
M7 =Mul16x4(N7, S7)

A0 =Add16x4(M0, M1)
A1 =Add16x4(M2, M3)
A2 =Add16x4(M4, M5)
A3 =Add16x4(M6, M7)

A8 =Add16x4(A0, A1)
A9 =Add16x4(A2, A3)

A10 = Add16x4(A8, A9)

Red = Red16x4(A10)

Res = Acc16x4(Red, reset)

out=Sig16(Res)

Output: out
```



Two optimizations:

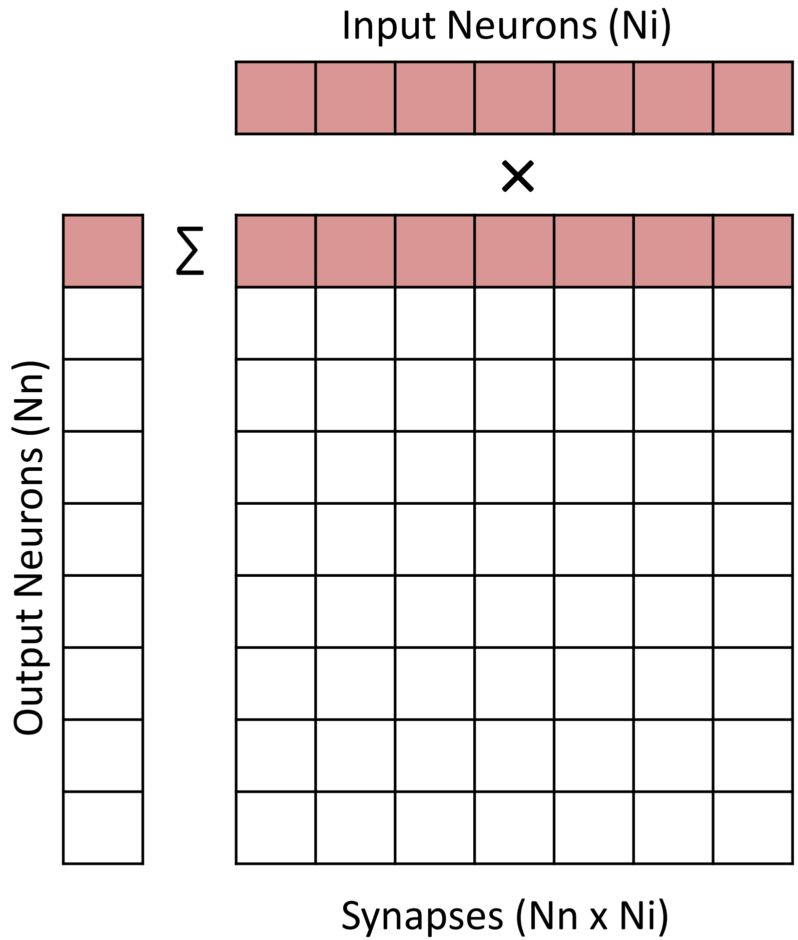
1. Increased the size of the DFG
2. Add an accumulation step and remove recurrence accumulation.

Optimized Classifier Layer



Optimized Classifier Layer

```
SD_CONFIG(dfg_config,dfg_size);

SD_DMA_READ(synapse,8,8,Ni*Nn/4,P_dfg_S);
SD_DMA_SCRATCH_LOAD(neuron_i, 0,Ni*2,1,0);
SD_WAIT_SCR_WR();

SD_SCR_PORT_STREAM(0, 0, Ni*2,1, P_dfg_N);
for (n = 0; n < Nn/nthreads; n++) {
  SD_CONST(P_dfg_reset,0,Ni/4-1);
  SD_CONST(P_dfg_reset,1,1);
  SD_GARBAGE(P_dfg_out,Ni/4-1);
  SD_DMA_WRITE(P_dfg_out,2,2,1,&neuron_n[n]);
}

SD_WAIT_ALL();
```



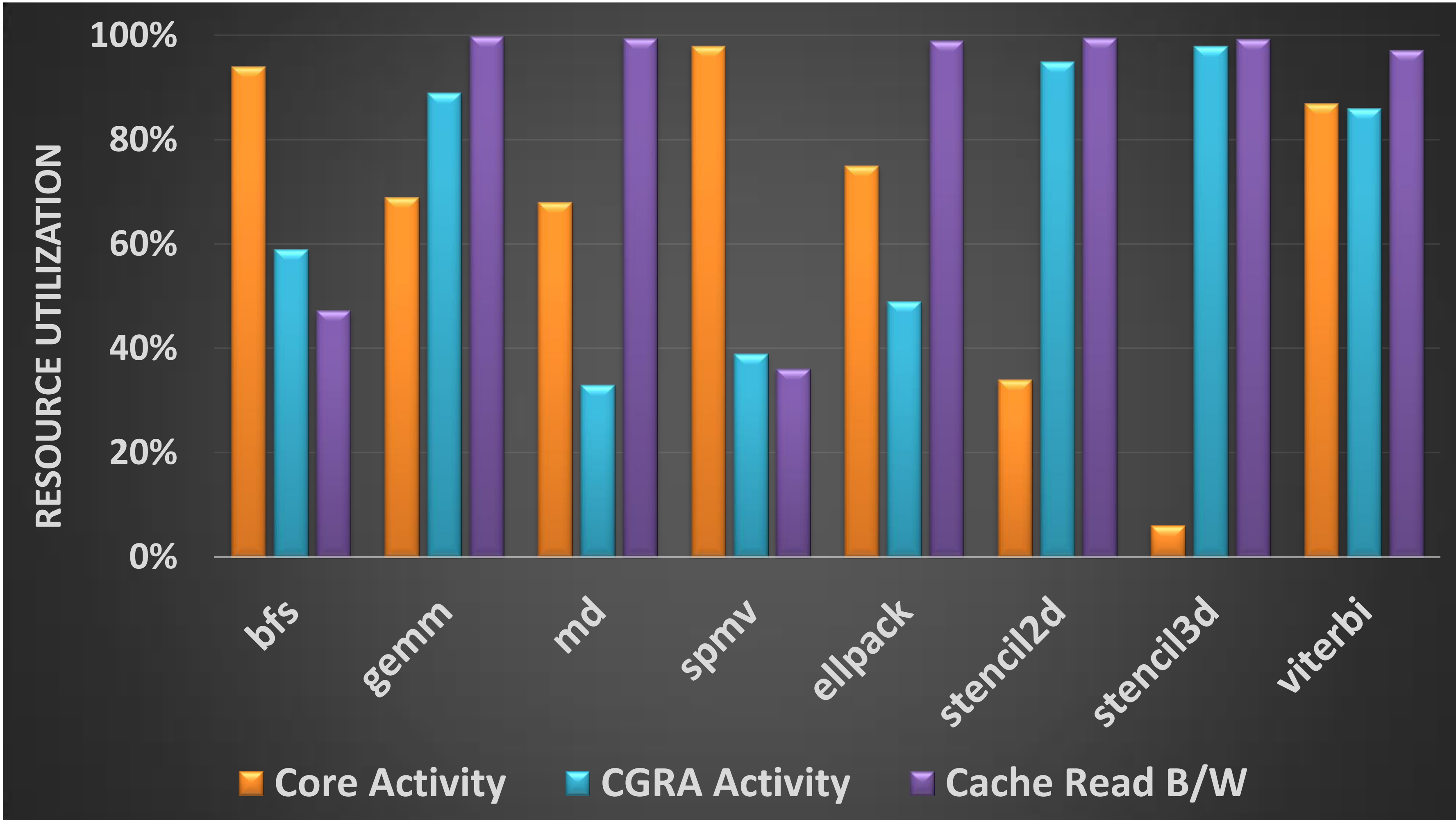
DianNao Power/Area Comparison

		area(mm^2)	power (mw)
Control Core + 16kB I & D$		0.16	39.1
CGRA	Network	0.12	31.2
	FUs (4×5)	0.04	24.4
	Total CGRA	0.16	55.6
5×Stream Engines		0.02	18.3
Scratchpad (4KB)		0.1	2.6
Vector Ports (Input & Output)		0.03	3.6
1 Softbrain Total		**0.47**	119.3
8 Softbrain Units		3.76	954.4
DianNao		2.16	418.3
Softbrain / DianNao Overhead		1.74	2.28

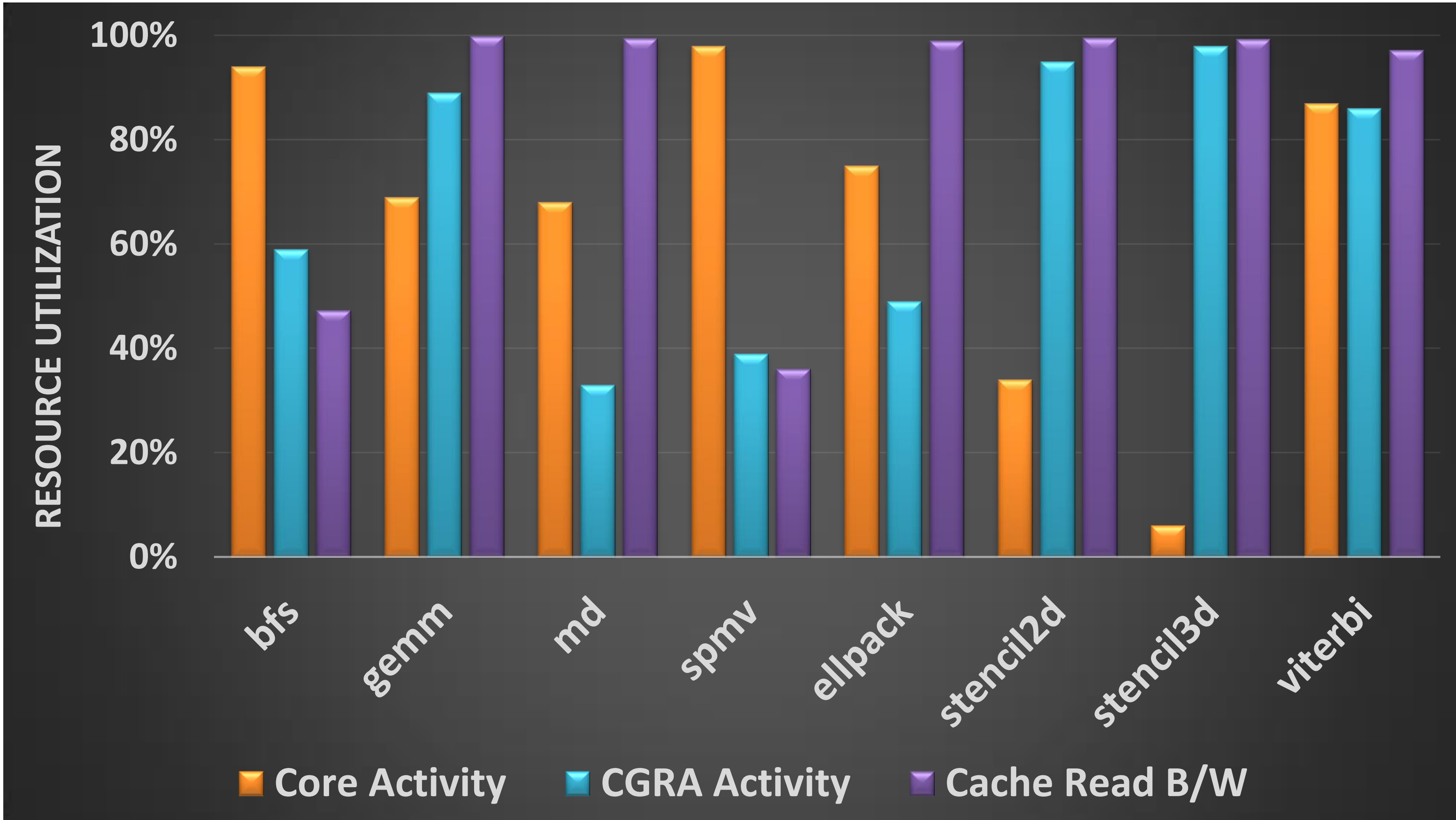
Table 3: Area and Power Breakdown / Comparison
(All numbers normalized to 55nm process technology)

Softbrain Resource Utilization

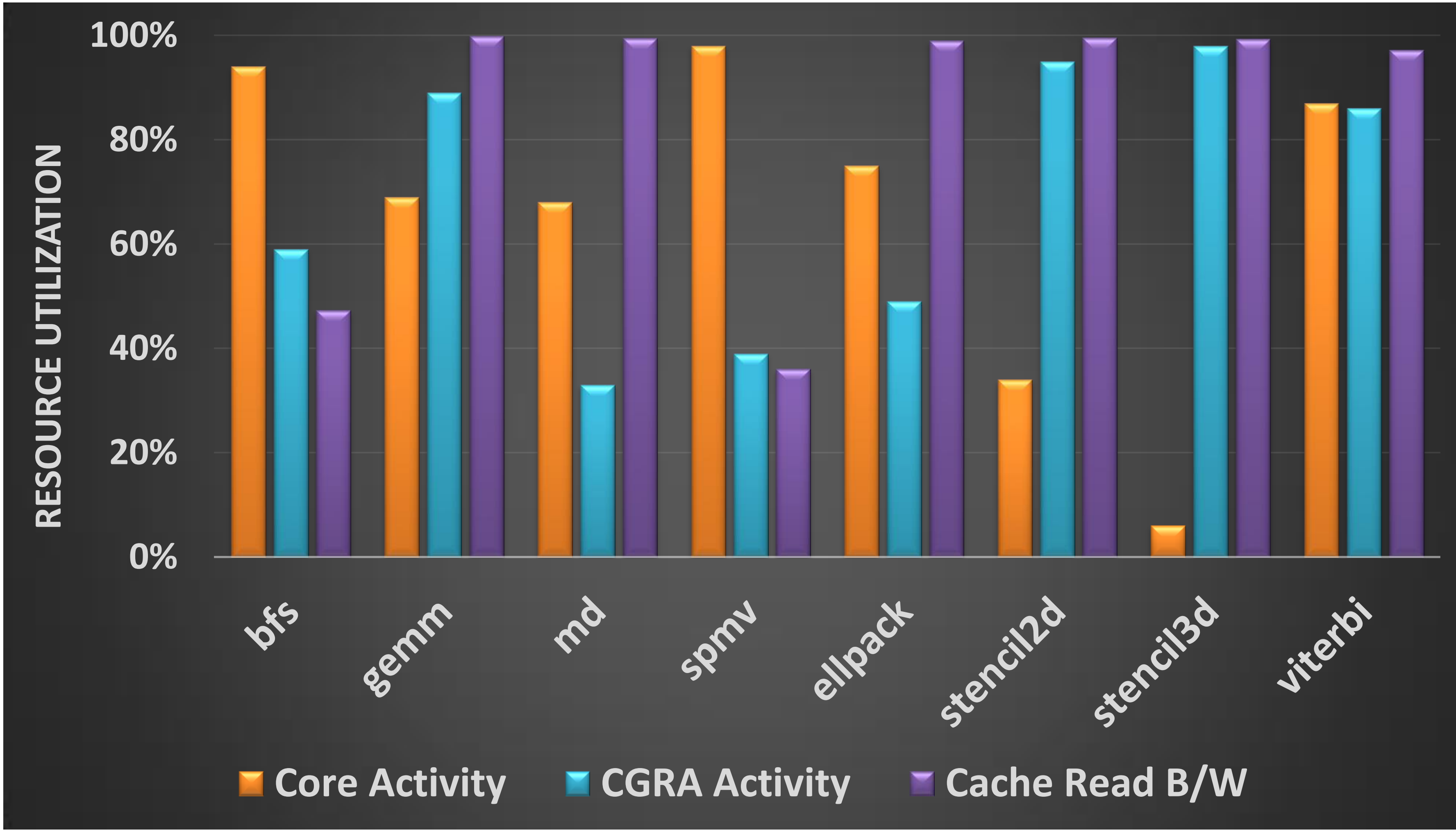
Softbrain Resource Utilization



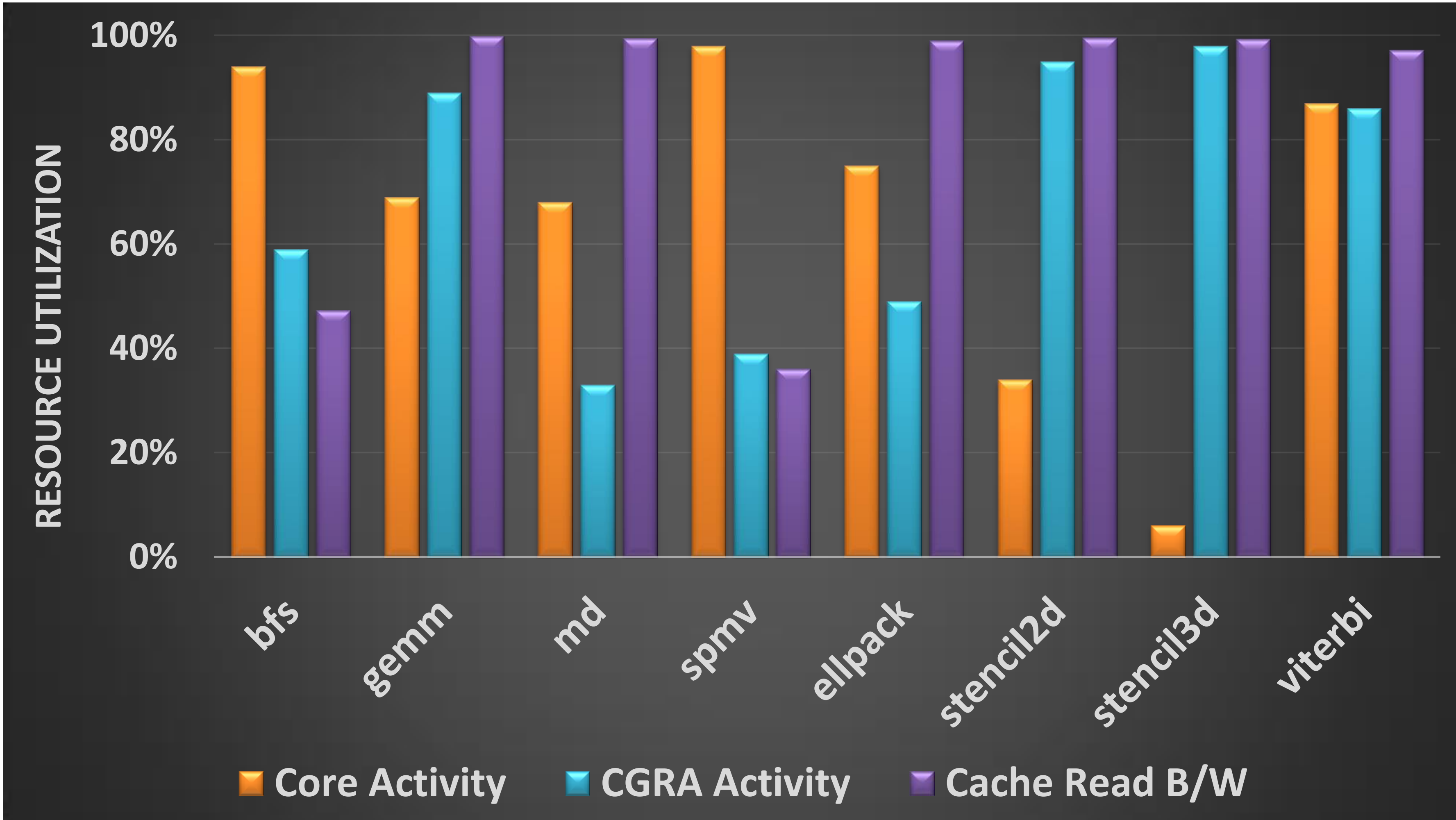
Softbrain Resource Utilization



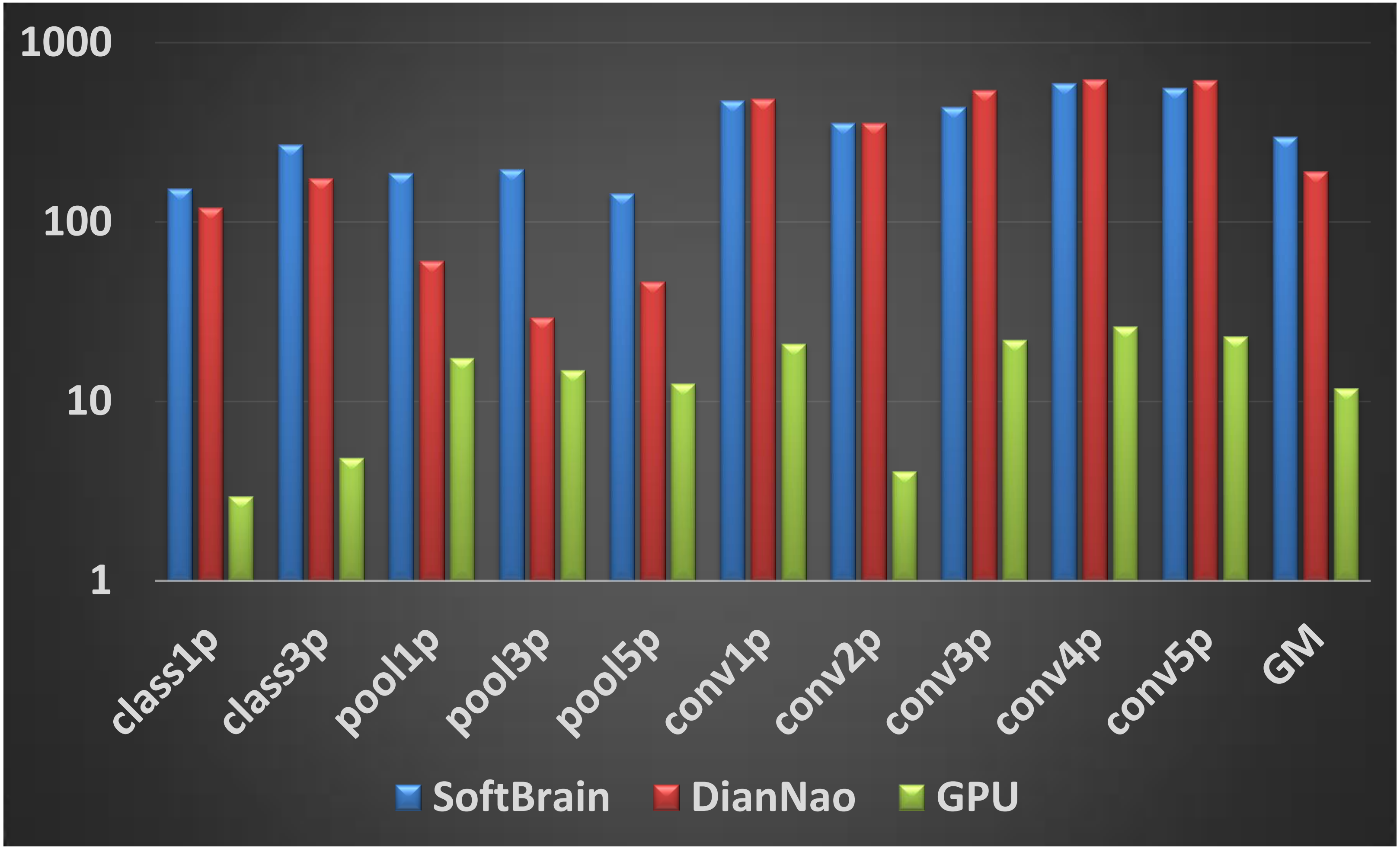
Softbrain Resource Utilization



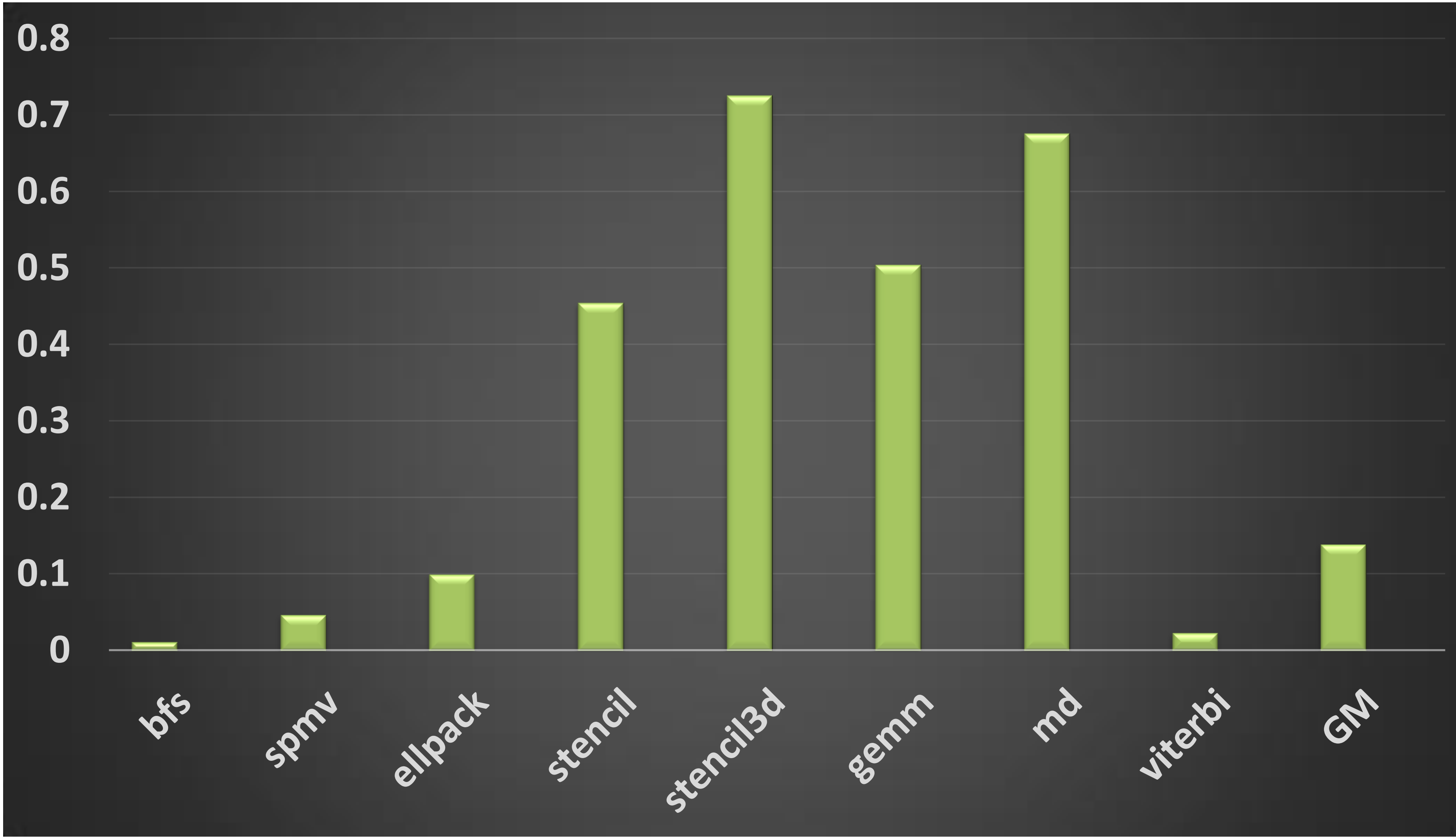
Softbrain Resource Utilization



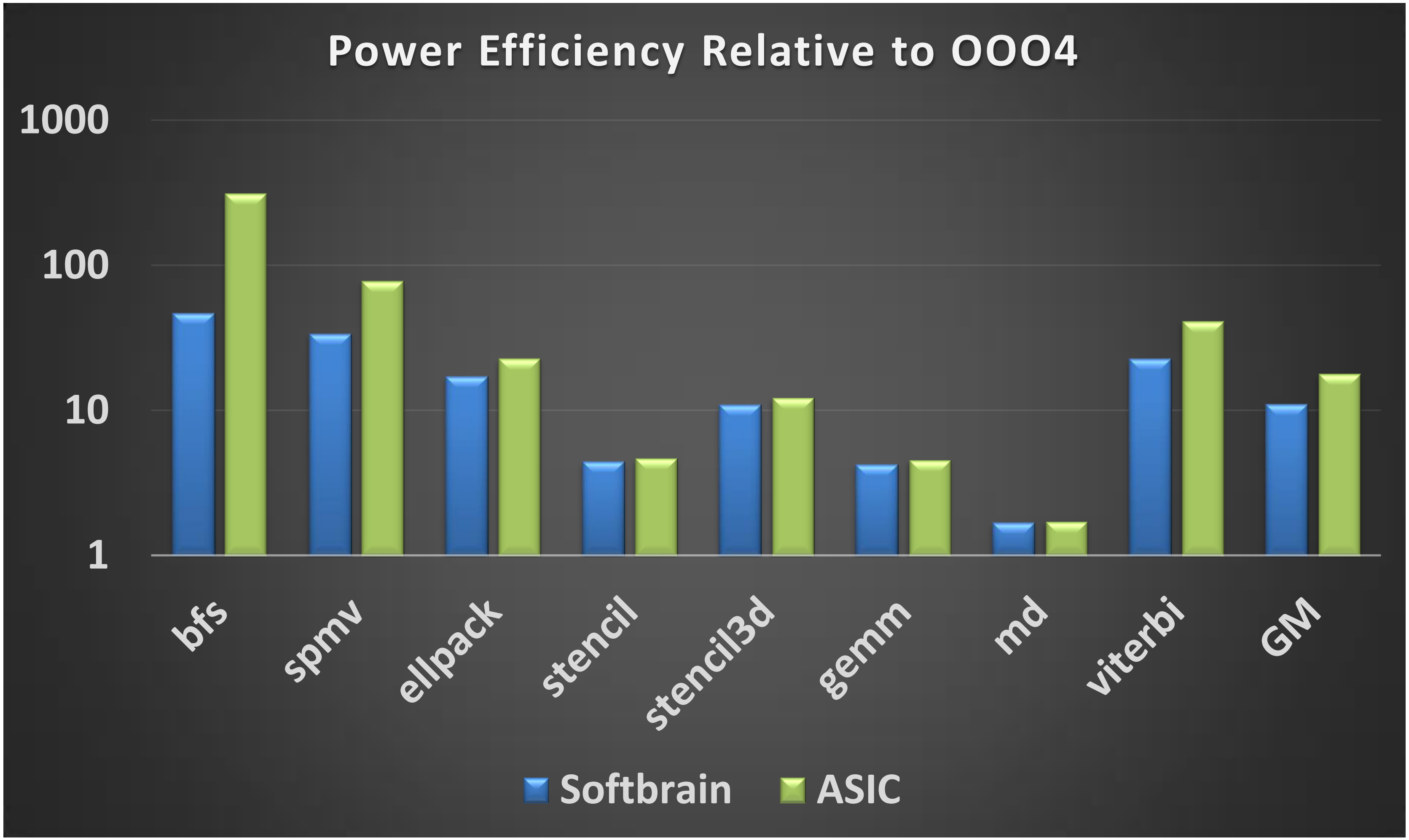
Softbrain vs. DianNao vs. GPU



ASIC Area Relative to Softbrain



Softbrain vs. ASIC Power Efficiency Comparison



Softbrain vs. ASIC Energy Efficiency Comparison