Stream-Dataflow Acceleration

Tony Nowatzki⁺, Vinay Gangadharar*, 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

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

- Dataflow Acceleration Talk
  - ISCA 2017 Stream
  - June 27, 2017
Era of Specialization

Traditional Multicore

Application domain specialization

- Image Processing
- Neural Approx.
- Scan
- Graph Traversal
- Deep Neural
- Stencil
- Sort

Reg Expr.
AI
Neural Approx.

June 27, 2017
ISCA 2017 Stream-Dataflow Acceleration Talk
Era of Specialization

**Traditional Multicore**

Application domain specialization

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

- Movidius Myriad VPU
- NVIDIA DGX-1 AI Accelerator
- Catapult FPGA Accelerator
Era of Specialization

Traditional Multicore

Domain Specific Acceleration

Application domain specialization

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

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

+ High Efficiency

10 – 100x Performance/Power or Performance/Area

Domain Specific Acceleration

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

- 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...

Convert 100+ Accelerators

1 Programmable Accelerator Fabric

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

Deep Neural
Image Processing
Automated Driving
Compression
Regex Matching
Query Processing

Source: Malitel Consulting

Standard programming and threading interface

June 27, 2017
ISCA 2017 Stream-Dataflow Acceleration Talk
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*
Specialization Spectrum

**Generality**

- **ASIC/DSPA**
- **GPGPU, FPGA, DSP, SIMD, GPP**

**Efficiency**

- **(energy efficient computing)**
- **Programmability / Re-configurability Features**

**Specialization Principles**
- **General Set of Micro-Architectural Mechanisms**
- **Architecture with Flexible Hardware-Software Programming Interface**
Specialization Spectrum

**Generality**

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

**Efficiency**

(energy efficient computing)

Programmability / Re-configurability Features

Architecture with Flexible Hardware-Software Programming Interface

General Set of Micro-Architectural Mechanisms

Specialization Principles

Programmable Hardware Accelerator
Specialization Spectrum

**Generality**

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

**Efficiency**

(energy efficient computing)

- Programmability / Re-configurability Features
- Architecture with Flexible Hardware-Software Programming Interface

Programmable Hardware Accelerator

- Efficiency close to DSAs/ASICs
- Trivial adaptation of new algorithms/applications
- Retain programmability

Specialization Principles

General Set of Micro-Architectural Mechanisms
Background Work*

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

**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
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 Graph

From Memory

Memory Stream

Local storage

Reuse Stream

Recurrence Stream

To Memory
Stream-Dataflow Acceleration

Stream-Dataflow Model

From Memory

Local storage

Reuse Stream

Memory Stream

Recurrence Stream

Dataflow Graph

To Memory

Programmable Stream-Dataflow Accelerator

Memory/Cache Hierarchy

Memory Interface

Programmable Scratchpad

Reconfigurable Fabric
Stream-Dataflow Acceleration

Stream-Dataflow Model

Programmable Stream-Dataflow Accelerator

- Data-parallel program kernels streaming data from memory
Stream-Dataflow Acceleration

Stream-Dataflow Model

From Memory

Local storage

Reuse Stream

Memory Stream

Recurrence Stream

To Memory

Programmable Stream-Dataflow Accelerator

Memory/Cache Hierarchy

Memory Interface

Input Data Streams

Programmable Scratchpad

Reconfigurable Fabric

Input Data Streams

Reuse streams

Recurring Data Streams

• Data-parallel program kernels streaming data from memory

• Dataflow computation fabric operates on data streams iteratively
Stream-Dataflow Acceleration

Stream-Dataflow Model

- 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

*Diagram:*
- Input Vector Ports (width)
- Dataflow based firing of data from vector ports
- Output Vector Ports (width)
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

From Memory

Memory Stream

Dataflow Graph
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

From Memory → Memory Stream → Dataflow Graph → 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

```
<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Address</td>
<td>Memory Address</td>
</tr>
<tr>
<td>Local Storage Address</td>
<td>Local Storage Address</td>
</tr>
<tr>
<td>DFG Port</td>
<td>DFG Port</td>
</tr>
</tbody>
</table>
```

![Dataflow Graph Diagram]

From Memory

Recurrence Stream

To Memory

Reuse Stream

Memory Stream
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

**Diagram:***
- From Memory
- Local storage
- Memory Stream
- Reuse Stream
- Recurrence Stream
- Dataflow Graph
- To Memory
- Time
  - Read Data
  - Read Barrier
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

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

• **Set-up Interface:**
  - \textit{SD\_Config} – Configuration data stream for dataflow computation fabric (CGRA)

• **Control Interface:**
  - \textit{SD\_Barrier\_Scratch\_Rd, SD\_Barrier\_Scratch\_Wr, SD\_Barrier\_All}

• **Stream Interface → SD\_[source]\_[dest]**
  - Source/Dest Parameters: \textit{Address (memory or local storage), DFG Port number}
  - Pattern Parameters: \textit{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
<table>
<thead>
<tr>
<th>Source</th>
<th>Access Pattern</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory, Local Storage, DFG Port</td>
<td></td>
<td>Memory, Local Storage, DFG Port</td>
</tr>
</tbody>
</table>

Stream-Dataflow
Hardware-Software Interface
Stream-Dataflow
Hardware-Software Interface

Source
Memory, Local Storage, DFG Port

Start Address

Access Pattern
Stride
Access Size
Number of Strides

Destination
Memory, Local Storage, DFG Port
Stream-Dataflow Hardware-Software Interface

Source
Memory, Local Storage, DFG Port

Destination
Memory, Local Storage, DFG Port

Access Pattern
Start Address
Stride
Access Size
Number of Strides

mem_addr = 0xA
num_strides = 2
memory_stride = 8
access_size = 4
Stream-Dataflow Hardware-Software Interface

**Source**
- Memory, Local Storage, DFG Port

**Destination**
- Memory, Local Storage, DFG Port

**Access Pattern**
- Start Address
- Stride
- Access Size
- Number of Strides

**Example Access Patterns**
- Linear
- Strided
- Overlapped
- Repeating
- Offset-Indirect
Stream-Dataflow
Hardware-Software Interface

Source
Memory, Local Storage, DFG Port

Access Pattern
Start Address
Stride
Access Size
Number of Strides

Destination
Memory, Local Storage, DFG Port

Example Access Patterns
Linear
Strided
Overlapped
Repeating
Offset-Indirect

2D Direct Streams
Stream-Dataflow
Hardware-Software Interface

Source
Memory, Local Storage, DFG Port

Start Address
Access Size

Access Pattern
Stride

Destination
Memory, Local Storage, DFG Port

Linear
Strided
Overlapped
Repeating
Offset-Indirect

Example Access Patterns

2D Direct Streams

2D Indirect Streams
Stream: 

Dataflow:
Stream-Dataflow ISA Encoding

Stream:

Dataflow:

Vector A[0:2] × Vector B[0:2]

Specified in a Domain Specific Language (DSL)

Dataflow Graph

C
Stream-Dataflow ISA Encoding

Stream:

Stream Encoding

<address, access_size, stride_size, length>

Dataflow:

Vector A[0:2]  Vector B[0:2]

Dataflow Graph

Specified in a Domain Specific Language (DSL)
Stream: 

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

Dataflow: 

Stream Encoding
<address, access_size, stride_size, length>

Dataflow Graph

Specified in a Domain Specific Language (DSL)
Stream:  

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

Stream Encoding

\(<\text{address}, \text{access}\_\text{size}, \text{stride}\_\text{size}, \text{length}>\)

Eg: \(<a, 1, 2, 100>\)

Dataflow:  

**Dataflow Graph**

- **Vector A[0:2]**
- **Vector B[0:2]**

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:

Dataflow Graph

Vector A[0:2] Vector B[0:2]

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>`

Eg: `<b, 1, 1, 100>`

Dataflow:

```
Vector A[0:2]  Vector B[0:2]
```

Dataflow Graph

```
Specified in a Domain Specific Language (DSL)
```
Stream: for $i = 1$ to $100$: 
... = $a[2\times i]$; 
... = $b[i]$; 
$c[b[i]] = ...$

Stream Encoding: $<\text{address, access\_size, stride\_size, length}>$
Eg: $<a, 1, 2, 100>$
$<b, 1, 1, 100>$

Dataflow: Specified in a Domain Specific Language (DSL)

Dataflow Graph:
- Vector $A[0:2]$
- Vector $B[0:2]$
- $\times$
- $+$
- $C$
Stream-
Dataflow ISA Encoding

Stream:

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

Dataflow:

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>

Specified in a Domain Specific Language (DSL)
Example Code: Dot Product

Original Program

```java
for(int i = 0 to N) {
    c += a[i] * b[i];
}
```
Example Code: Dot Product

Original Program

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

Dataflow Encoding

```
P1  
  └── X
      ├── P2
      │   └── +
      │       └── P3
```
Example Code: Dot Product

```java
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

1. P1
2. P2
3. P3

- Multiply
- Add
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
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
Stream-Dataflow Accelerator Architecture

**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

Diagram:

- Input Vector Port Interface
- CGRA Spatial Fabric
- Output Vector Port 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-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**

**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
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

- Tiny RISC-V In-order core or MCU

- Stream command interface exposed to a general purpose programmable core

- Non-intrusive accelerator design

Stream Commands

Coarse-grained Stream commands issued by core through a command queue

Stream-Dataflow Acceleration Talk

June 27, 2017
Stream-Dataflow Accelerator Architecture

Stream ISA Encoding

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

Tiny RISC-V In-order core or MCU

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

Coarse-grained Stream commands issued by core through a command queue

Stream Commands

In-Order core or MCU

Stream Command Dispatcher

D$ I$

Input Vector Port Interface

Scratchpad Stream Engine

Scratchpad

Memory Stream Engine

Output Vector Port Interface

Recurrence Stream Engine

FUs

CGRA Spatial Fabric

To/from memory hierarchy

512b 64b Stream Command

I$, D$

Tiny RISC-V In-order core or MCU

Coarse-grained Stream commands issued by core through a command queue

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Stream-Dataflow Accelerator Architecture

Stream ISA Encoding

- Send \(a[0:N] \rightarrow P1\)
- Send \(b[0:N] \rightarrow P2\)
- Get \(P3 \rightarrow c\)

Stream-Dataflow Acceleration Potential

1. Dataflow based pipelined concurrent execution
2. High Computation Activity Ratio: Number of Computations/Stream Commands

- Non-intrusive accelerator design
- General purpose programmable core

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
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**

**Software Stack**
- Stream-Dataflow Code (C/C++)
- DFG File
- DFG Compiler (ILP Solver)
- DFG.h
- RISCV GCC
- RISCV Binary
- Softbrain Config.

**Hardware**
- Accelerator Model Configuration
- Chisel Parameterizable Accelerator Implementation
- Softbrain RTL

**Evaluation**
- Accelerator Cycle-level Simulator
- Chisel-generated Verilog Synthesis + Synopsis DC

**Flowchart**
- Stream-Dataflow Code (C/C++) → DFG File → DFG Compiler (ILP Solver) → DFG.h → RISCV GCC → RISCV Binary → Softbrain Config. → Accelerator Model Configuration → Chisel Parameterizable Accelerator Implementation → Softbrain RTL → Chisel-generated Verilog Synthesis + Synopsis DC
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)

Speedup Relative to OOO4 (DNN Workloads)

- class1p
- class3p
- pool1p
- pool3p
- pool5p
- conv1p
- conv2p
- conv3p
- conv4p
- conv5p
- GM

- SoftBrain
- DianNao

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Domain-Specific Accelerator Comparison (Softbrain vs DianNao)

DianNao Area: 2.16 mm²  Softbrain Area: 3.76 mm²
DianNao Power: 420 mW  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

<table>
<thead>
<tr>
<th></th>
<th>Softbrain Area: 3.76 mm²</th>
<th>Softbrain Power: 950 mW</th>
</tr>
</thead>
<tbody>
<tr>
<td>DianNao</td>
<td>2.16 mm²</td>
<td>420 mW</td>
</tr>
</tbody>
</table>

June 27, 2017
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

<table>
<thead>
<tr>
<th>Power Efficiency Relative to OOO4 (GM)</th>
<th>Energy Efficiency Relative to OOO4 (GM)</th>
<th>ASIC Area Relative to Softbrain (GM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softbrain vs ASIC designs</td>
<td>Softbrain vs ASIC designs</td>
<td>Softbrain vs ASIC designs</td>
</tr>
<tr>
<td>Perf. – Able to match the performance</td>
<td>Perf. – Able to match the performance</td>
<td>Perf. – Able to match the performance</td>
</tr>
<tr>
<td>Power – ~1.6x overhead</td>
<td>Power – ~1.6x overhead</td>
<td>Power – ~1.6x overhead</td>
</tr>
<tr>
<td>Energy Efficiency – ~1.5x overhead</td>
<td>Energy Efficiency – ~1.5x overhead</td>
<td>Energy Efficiency – ~1.5x overhead</td>
</tr>
<tr>
<td>Area – ~8x overhead*</td>
<td>Area – ~8x overhead*</td>
<td>Area – ~8x overhead*</td>
</tr>
</tbody>
</table>

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

**Getting There !!**

A good enabler for exploring general purpose programmable hardware acceleration ....
Backup
Traditional Arch.

Programs
- General Language
- Compiler
- General ISA

General Purpose Hardware

Accelerator (DSA)

Domain-Specific Programs

Application/Domain Specific Hardware

Tiny H/W-S/W Interface

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

- Programs
  - General Language
  - Compiler
  - General ISA

General Purpose Hardware

- Programs
  - Programs ("Specialized")
  - Re-Configurable Hardware

H/W Parameters

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

Accelerator (DSA)

- Domain-Specific Programs
  - Application/Domain Specific Hardware

Tiny H/W-S/W Interface
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

\[ C[i] = A[i] \times B[i] \]

Input Ports: A, B
Output Port: C
Stream-Dataflow Execution Model
Detailed Example

\[ C[i] = A[i] \times B[i] \]

Maps to two i/p scalar vector ports
Maps to multiplier of CGRA substrate
Maps to an o/p scalar vector port

Input Ports: \( A \) \( B \)
Output Port: \( C \)
Stream-Dataflow Execution Model
Detailed Example

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

Maps to two i/p scalar vector ports
Maps to multiplier of CGRA substrate
Maps to an o/p scalar vector port

Input Ports: A, B

Output Port: C

Scratchpad
Stream-Dataflow Execution Model
Detailed Example

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

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

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

Stream Commands
C1) Mem → Scratch

Time

CGRA fabric state
Low-power core state

Command generation

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

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

June 27, 2017
ISCA 2017 Stream-Dataflow Acceleration Talk
Stream-Dataflow Execution Model

Detailed Example

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

Stream Commands
C1) Mem → Scratch
C2) Scratch Wr Barrier
C3) Scratch → Port A

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

CGRA fabric state

Low-power core state

June 27, 2017
Stream-Dataflow Execution Model

Detailed Example

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

Stream Commands
C1) Mem → Scratch
C2) Scratch Wr Barrier
C3) Scratch → Port A
C4) Mem → Port B

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

CGRA fabric state
Low-power core state

Program Order

Command generation

Time
Stream-Dataflow Execution Model

Detailed Example

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

Stream Commands
C1) Mem → Scratch
C2) Scratch Wr Barrier
C3) Scratch → Port A
C4) Mem → Port B
C5) Port C → Mem

Program Order

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

CGRA fabric state
Low-power core state

June 27, 2017
Stream-Dataflow Execution Model
Detailed Example

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

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

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

CGRA fabric state
Low-power core state

Program Order

Time

Command generation

June 27, 2017
Stream-Dataflow Execution Model
Detailed Example

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

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

C[i] = A[i] * B[i]
Stream-Dataflow Execution Model

Detailed Example

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

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

Critical Path:
- C3
- C4
- C5
- C6
- C7

C[i] = A[i] * B[i]
Stream-Dataflow Execution Model

Detailed Example

Legend:

<table>
<thead>
<tr>
<th>Enqueued</th>
<th>Dispatched</th>
<th>Barrier</th>
<th>Dependency</th>
<th>Iter. boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

CGRA fabric state

Low-power core state

Program Order

Time

Command generation

Scratchpad

A

B

C

June 27, 2017
Stream-Dataflow Execution Model

Detailed Example

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

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

C[i] = A[i] * B[i]
Stream-Dataflow Execution Model
Detailed Example

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

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

Time
Command generation
Processing

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

Scratchpad

June 27, 2017
Stream-Dataflow Execution Model
Detailed Example

Stream Commands
C1) Mem \(\rightarrow\) Scratch
C2) Scratch Wr Barrier
C3) Scratch \(\rightarrow\) Port A
C4) Mem \(\rightarrow\) Port B
C5) Port C \(\rightarrow\) Mem
C6) Mem \(\rightarrow\) Port B
C7) All Barrier

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

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

CGRa fabric state
Low-power core state

June 27, 2017
Stream-Dataflow Execution Model

Detailed Example

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

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

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

CGRA fabric state
Low-power core state

Program Order

Time

Command generation

Processing

Scratchpad

A

B

C

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Stream-Dataflow Execution Model
Detailed Example

<table>
<thead>
<tr>
<th>Legend:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enqueued</td>
</tr>
<tr>
<td>□</td>
</tr>
<tr>
<td>Barrier</td>
</tr>
</tbody>
</table>

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

Program Order

Time

Command generation
Processing

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

Scratchpad

A

B

C

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Stream-Dataflow Execution Model

Detailed Example

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

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

C[i] = A[i] * B[i]
Stream-Dataflow Execution Model
Detailed Example

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

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

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

Legend:
- Scratchpad
- A
- B
- C

Program Order

CGRA fabric state
Low-power core state

Time

Command generation
Processing
Resume
Stream-Dataflow Accelerator Potential

1. Dataflow based pipelined concurrent execution

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

Legend:
- Enqueued
- Dispatched
- Resource idle
- Barrier
- Dependency
- Iter. boundary

June 27, 2017
### Inefficiencies in Data-Parallel Architectures

#### SIMD & Short Vector SIMD
- Control Core
- SIMD Vector Units
- Sub-SIMD
- Vector Register File

#### SIMT
- Warp Scheduler + Vector Dispatch
- Large Register File + Scratchpad
- Vector Lanes
- Memory Coalescer

#### Vector Thread
- Control Core + Vector Dispatch
- Vector Lanes
- Vector Fetch Support

#### Spatial Dataflow
- Spatial Dataflow
- Distributed PEs

### 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
- 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
• 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)

Cache/ Memory Heirarchy

Scratchpad
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

Memory Interface
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

LEGEND
- GREEN → Data Line
- BLACK → Control/Commands
- Control
- State storage/SRAM
- Datapath

Stream Dispatcher
- SD CMD
- D-Cache Req/Resp
- Cache Req/Resp

RISCV Rocket Core

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Micro-Architecture of Stream-Dataflow Accelerator (Softbrain)

Cache/ Memory Heirarchy

Scratchpad
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

Memory Interface
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

LEGEND
- Control
- State storage/SRAM
- Datapath
- BLACK → Data Line
- GREEN → Control/Commands

Stream Dispatcher
- Stream Cmd. Queue
- VP Scoreboard
- Resource Status Checker
- Issue
- Recurrence Stream Engine (RSE)

RISCV Rocket Core
- SD CMD

Input Data VPs
- Output Data VPs

CGRA
- Config
- Indirect Load/Store VPs

From SSE
- From MSE

MSE Write Cmd
- MSE Read Cmd

Free SSE Read
- Free SSE Write

Free MSE Read
- Free MSE Write

SSE Write Cmd
- SSE Read Cmd

SCR to MSE writes

Tag Invalidate

Cache/ Memory Heirarchy

June 27, 2017
ISCA 2017 Stream-Dataflow Acceleration Talk

34
Micro-Architecture of Stream-Dataflow Accelerator (Softbrain)

Cache/ Memory Heirarchy

Scratchpad
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

Memory Interface
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

Scratchpad
- SCR to MSE writes

Memory Interface
- MSE Write Cmd
- MSE Read Cmd

Cache/ Memory Heirarchy
- Writes
- Reads

LEGEND
- GREEN → Data Line
- BLACK → Control/Commands
- Datapath
- Control
- State storage/SRAM

Stream Dispatcher
- Stream Cmd. Queue
- VP Scoreboard
- Resource Status Checker
- CGRA Config
- Stream Cmds to SEs
- RSE Cmd

RISCV Rocket Core
- SD CMD

CGRA
- Input Data VPs
- Recurrence Stream Engine (RSE)
- Output Data VPs

Resource Status Checker
- Issue
- Checker

Stream Dispatcher
- SCR to MSE writes

From MSE
- From SSE

Indirect Load/Store VPs
- To SSE
- To MSE

Cache/ Memory Heirarchy
- Tag Invalidate
- Cache Req/Resp
- D-Cache Req/Resp

June 27, 2017
ISCA 2017 Stream-Dataflow Acceleration Talk
Micro-Architecture of Stream-Dataflow Accelerator (Softbrain)

Cache/ Memory Heirarchy

Scratchpad

<table>
<thead>
<tr>
<th>Scratch Stream Engine (SSE) for Writes</th>
<th>Scratch Stream Engine (SSE) for Reads</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Memory Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Stream Engine (MSE) for Writes</td>
</tr>
<tr>
<td>Memory Stream Engine (MSE) for Reads</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes</td>
</tr>
<tr>
<td>Reads</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEGEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
</tr>
<tr>
<td>State storage/SRAM</td>
</tr>
<tr>
<td>Datapath</td>
</tr>
<tr>
<td>BLACK ➔ Data Line</td>
</tr>
<tr>
<td>GREEN ➔ Control/Commands</td>
</tr>
</tbody>
</table>

Stream Dispatcher

<table>
<thead>
<tr>
<th>Stream Dispatcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP Scoreboard</td>
</tr>
<tr>
<td>Resource Status Checker</td>
</tr>
<tr>
<td>Issue</td>
</tr>
<tr>
<td>Recurrence Stream Engine (RSE)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Data VPs</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Output Data VPs</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect Load/Store VPs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>To MSE</td>
</tr>
<tr>
<td>From MSE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>From SSE</td>
</tr>
<tr>
<td>To SSE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCR to MSE writes</td>
</tr>
<tr>
<td>MSE Write Cmd</td>
</tr>
<tr>
<td>MSE Read Cmd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Invalidate</td>
</tr>
<tr>
<td>Free SSE Write</td>
</tr>
<tr>
<td>Free SSE Read</td>
</tr>
<tr>
<td>Free MSE Read</td>
</tr>
<tr>
<td>Free MSE Write</td>
</tr>
<tr>
<td>Free RSE</td>
</tr>
<tr>
<td>Stream Cmds to SEs</td>
</tr>
<tr>
<td>CGRA Config</td>
</tr>
<tr>
<td>Config</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISCV Rocket Core</td>
</tr>
<tr>
<td>SD CMD</td>
</tr>
<tr>
<td>D-Cache Req/Resp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CGRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD CMD</td>
</tr>
<tr>
<td>D-Cache Req/Resp</td>
</tr>
</tbody>
</table>

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Micro-Architecture of Stream-Dataflow Accelerator (Softbrain)
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)

**Legend**
- **Control**: Green
- **State storage/SRAM**: Black
- **Datapath**: Blue

**Cache/Memory Heirarchy**

**Scratchpad**
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

**Memory Interface**
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

**CGRA**

**Stream Dispatcher**
- RISCV Rocket Core
- Stream Cmd. Queue
- VP Scoreboard
- Resource Status Checker
- Issue
- SCR to MSE writes
- SCR to MSE writes
- Stream Cmnds to SEs
- CGRA Config
- Recurrence Stream Engine (RSE)

**Input Data VPs**
- 

**Output Data VPs**
- 

**Indirect Load/Store VPs**
- From MSE
- To MSE
- To SSE
- To MSE
- From SSE

**Memroy Interface**
- MSE Write Cmd
- MSE Read Cmd

**Tag Invalidate**
- Tag Invalidate

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

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Micro-Architecture of Stream-Dataflow Accelerator (Softbrain)

Cache/ Memory Heirarchy

Scratchpad
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

Memory Interface
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

Legend
- BLACK -> Data Line
- GREEN -> Control/Commands

Micro-Architecture of Stream-Dataflow Accelerator (Softbrain)

Stream Dispatcher
- VP Scoreboard
- Resource Status Checker
- Cmd. Issue
- Stream Cmds to SEs
- SCR to MSE writes
- MSE Write Cmd
- MSE Read Cmd

RISCV Rocket Core
- SD CMD

Input Data VPs
- ... to CGRA

Output Data VPs
- ... from CGRA

CGRA
- ... to YSEs

Memory Interface
- Writes
- Reads

Scratchpad
- SCR to MSE writes

Legend
- GREEN -> Control/Commands
- BLACK -> Data Line

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)

Cache/ Memory Heirarchy

**Scratchpad**
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

**Memory Interface**
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

**Stream Dispatcher**
-VP Scoreboard
- Resource Status Checker
-Cmd. Issue
-Recurrence Stream Engine (RSE)

**RISCV Rocket Core**

LEGEN
- **GREEN** → Data Line
- **GREEN** → Control/Commands

**LEGEND**
- **GREEN** Control
- **State storage/SRAM**
- **Datapath**

June 27, 2017
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)

**Legend**
- Control
- State storage/SRAM
- Datapath
- BLACK → Data Line
- GREEN → Control/Commands

**Cache/Memory Heirarchy**

**Scratchpad**
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

**Memory Interface**
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

**Stream Dispatcher**
- VP Scoreboard
- Resource Status Checker
- Issue
- Stream Commands to SEs
- Recurrence Stream Engine (RSE)
- SCR to MSE writes
- SCR to MSE writes
- MSE Write Cmd
- MSE Read Cmd
- CGRA Config
- Input Data VPs
- Output Data VPs
- Recurrence Stream Engine (RSE)

**CGRA**

**Control**

**State storage/SRAM**

**Datapath**

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

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Micro-Architecture of Stream-Dataflow Accelerator (Softbrain)

Cache/ Memory Heirarchy

Scratchpad

Scratch Stream Engine (SSE) for Writes
Scratch Stream Engine (SSE) for Reads

Memory Interface

Memory Stream Engine (MSE) for Writes
Memory Stream Engine (MSE) for Reads

Scratch Stream Engine (SSE)

Scratchpad

Resource Status Checker

Command Issue

Recurrence Stream Engine (RSE)

Input Data VPs

Output Data VPs

CGRA

Stream Dispatcher

Stream Dispatcher

RISCV Rocket Core

Legend

GREEN ➔ Data Line
BLACK ➔ Control/Commands

June 27, 2017

ISCA 2017 Stream-Dataflow Acceleration Talk
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)

**Cache/ Memory Heirarchy**

**Micro-Architecture**
- **Scratchpad**
  - Scratch Stream Engine (SSE) for Writes
  - Scratch Stream Engine (SSE) for Reads
- **Memory Interface**
  - Memory Stream Engine (MSE) for Writes
  - Memory Stream Engine (MSE) for Reads
- **Stream Dispatcher**
  - VP Scoreboard
  - Resource Status Checker
  - Issue
  - Recurrence Stream Engine (RSE)
  - SCR to MSE writes
  - SCR to MSE writes
- **CGRA**
  - Input Data VPs
  - Output Data VPs
  - Indirect Load/Store VPs
  - To MSE
  - To SSE
  - From MSE
  - From SSE

**LEGEND**
- Green: Data Line
- Green: Control/Commands
- Black: Data Line
- Blue: Datapath
- Gray: Control
- Yellow: State storage/SRAM

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

**Stream Dispatcher**
- SD CMD
- D-Cache Req/ Resp
- I-Cache Req/ Resp
- Tag Invalidate

**Cache/ Memory Heirarchy**
- Writes
- Reads

**LEGEND**
- \(\text{LEGEND}\)
- Control
- State storage/SRAM
- Datapath
- BLACK \(\rightarrow\) Data Line
- GREEN \(\rightarrow\) Control/Commands

**June 27, 2017**

**ISCA 2017 Stream-Dataflow Acceleration Talk**
Micro-Architecture of Stream-Dataflow Accelerator (*Softbrain*)

Cache/ Memory Heirarchy

**Scratchpad**
- Scratch Stream Engine (SSE) for Writes
- Scratch Stream Engine (SSE) for Reads

**Memory Interface**
- Memory Stream Engine (MSE) for Writes
- Memory Stream Engine (MSE) for Reads

**Stream Dispatcher**
- Stream Cmd. Queue
- VP Scoreboard
- Resource Status Checker
- Issue
- Recurrence Stream Engine (RSE)
- SCR to MSE writes

**RISCV Rocket Core**

**LEGEND**
- GREEN = Data Line
- BLACK = Control/Commands
- CONTROL
- State storage/SRAM
- Datapath

**Cache/ Memory Heirarchy**
- Writes
- Reads

**MSE Write Cmd**
- MSE Read Cmd

**From MSE**
- To SSE
- From SSE

**From RSE**
- To MSE

**CGRA**
- Input Data VPs
- Output Data VPs

**Indirect Load/Store VPs**

**Tag Invalidate**
- I-Cache Req/Resp
- D-Cache Req/Resp

**LEGEND**
- GREEN = Data Line
- BLACK = Control/Commands

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

**June 27, 2017**

**ISCA 2017 Stream-Dataflow Acceleration Talk**
Softbrain Stream Engine Request Pipeline

- Responsible for address generation for both affine and non-affine data-streams
- Priority based selection among multiple queued data-streams
- 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
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

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

Computation Graph:

Scalar

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

Vector

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

Stream-Dataflow

```c
Send a[i:i+N] \rightarrow P1
Send b[i:i+N] \rightarrow P2
Get P3 \rightarrow result
```

~2N Instructions

~2N/vec_len Instructions

~3 Instructions
Existing Architectures for Data Parallel

Vector Processor
(eg. ARM Neon, X86 SSE)

Spatial Processor
(eg. Tilera, TRIPS, Wavescalar)

- Amortized Instruction Issue
- Efficient Vector-Memory

- Efficient Dataflow b/t Units
- Flexible Computation Patterns

June 27, 2017
ISCA 2017 Stream-Dataflow Acceleration Talk
Existing Architectures for Data Parallel

Vector Processor
(eg. ARM Neon, X86 SSE)

- Vector Issue
- Vector Lane
- Vector Mem
- Reg.
- LSU FUs

Spatial Processor
(eg. Tilera, TRIPS, Wavescalar)

- Independent PEs
- Scalar Mem
- Issue
- Reg.
- LSU FUs

Vectorized memory interface + Spatial Datapath + Amortized Issue
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:

```c
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

\[
\begin{align*}
M_0 &= \text{Mul16x4}(N_0, S_0) \\
M_1 &= \text{Mul16x4}(N_1, S_1) \\
M_2 &= \text{Mul16x4}(N_2, S_2) \\
M_3 &= \text{Mul16x4}(N_3, S_3) \\
M_4 &= \text{Mul16x4}(N_4, S_4) \\
M_5 &= \text{Mul16x4}(N_5, S_5) \\
M_6 &= \text{Mul16x4}(N_6, S_6) \\
M_7 &= \text{Mul16x4}(N_7, S_7) \\
A_0 &= \text{Add16x4}(M_0, M_1) \\
A_1 &= \text{Add16x4}(M_2, M_3) \\
A_2 &= \text{Add16x4}(M_4, M_5) \\
A_3 &= \text{Add16x4}(M_6, M_7) \\
A_8 &= \text{Add16x4}(A_0, A_1) \\
A_9 &= \text{Add16x4}(A_2, A_3) \\
A_{10} &= \text{Add16x4}(A_8, A_9) \\
\text{Red} &= \text{Red16x4}(A_{10}) \\
\text{Res} &= \text{Acc16x4}(\text{Red}, \text{reset}) \\
\text{out} &= \text{Sig16}(\text{Res})
\end{align*}
\]

Output: out

Two optimizations:
1. Increased the size of the DFG
2. Add an accumulation step and remove recurrence accumulation.
Optimized Classifier Layer

Input Neurons (Ni) × \[\sum\]

Output Neurons (Nn)

Synapses (Nn x Ni)
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();

Input Neurons (Ni)
×
∑

Output Neurons (Nn)

Synapses (Nn x Ni)
# DianNao Power/Area Comparison

<table>
<thead>
<tr>
<th>Component</th>
<th>Area (mm²)</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Core + 16kB I &amp; D$</td>
<td>0.16</td>
<td>39.1</td>
</tr>
<tr>
<td>CGRA Network</td>
<td>0.12</td>
<td>31.2</td>
</tr>
<tr>
<td>CGRA FUs (4×5)</td>
<td>0.04</td>
<td>24.4</td>
</tr>
<tr>
<td><strong>Total CGRA</strong></td>
<td><strong>0.16</strong></td>
<td><strong>55.6</strong></td>
</tr>
<tr>
<td>5×Stream Engines</td>
<td>0.02</td>
<td>18.3</td>
</tr>
<tr>
<td>Scratchpad (4KB)</td>
<td>0.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Vector Ports (Input &amp; Output)</td>
<td>0.03</td>
<td>3.6</td>
</tr>
<tr>
<td><strong>1 Softbrain Total</strong></td>
<td><strong>0.47</strong></td>
<td><strong>119.3</strong></td>
</tr>
<tr>
<td>8 Softbrain Units</td>
<td>3.76</td>
<td>954.4</td>
</tr>
<tr>
<td><strong>DianNao</strong></td>
<td><strong>2.16</strong></td>
<td><strong>418.3</strong></td>
</tr>
<tr>
<td>Softbrain / DianNao Overhead</td>
<td>1.74</td>
<td>2.28</td>
</tr>
</tbody>
</table>

**Table 3: Area and Power Breakdown / Comparison**

(All numbers normalized to 55nm process technology)
Softbrain Resource Utilization
Softbrain Resource Utilization

The chart illustrates the resource utilization for various applications, including bfs, gemm, md, spmv, ellpack, stencil2d, stencil3d, and viterbi. The y-axis represents the resource utilization percentage, ranging from 0% to 100%. The x-axis lists the application names.

- **Core Activity** is represented by orange bars.
- **CGRA Activity** is represented by teal bars.
- **Cache Read B/W** is represented by purple bars.

The data shows varying levels of resource utilization across different applications, providing insights into how efficiently resources are being allocated in each case.
Softbrain Resource Utilization

- **bfs**
- **gemm**
- **md**
- **spmv**
- **ellpack**
- **stencil2d**
- **stencil3d**
- **viterbi**

**Core Activity**, **CGRA Activity**, **Cache Read B/W**
Softbrain vs. DianNao vs. GPU

The bar chart compares SoftBrain, DianNao, and GPU across various tasks such as class1p, class3p, pool1p, pool3p, pool5p, conv1p, conv2p, conv3p, conv4p, conv5p, and GM. The y-axis represents performance measured in MIPS, ranging from 1 to 1000.
ASIC Area Relative to Softbrain

The chart above illustrates the ASIC area relative to Softbrain for various applications.

- bfs
- spmv
- ellpack
- stencil
- stencil3d
- gemm
- md
- viterbi
- GM
Softbrain vs. ASIC
Power Efficiency Comparison

Power Efficiency Relative to OOO4

- bfs
- spmv
- ellpack
- stencil
- stencil3d
- gemm
- md
- viterbi
- GM

Softbrain

ASIC
Softbrain vs. ASIC
Energy Efficiency Comparison

Energy Efficiency Relative to OOO4

- bfs
- spmv
- ellpack
- stencil
- stencil3d
- gemm
- md
- viterbi
- GM