DSAGEN: Democratizing the Research on Spatial Accelerators

Jian Weng, Sihao Liu, Vidushi Dadu, Tony Nowatzki
University of California, Los Angeles
Oct. 17\textsuperscript{th}, 2020
Outline

• Background: Decoupled Spatial Architectures
• Decoupled-Spatial Architecture Programming

• Advanced DSA Programming
  • Warm up: Support for data-dependent control/memory
  • Exercise 3: Sparse vector-vector dot-product (Hands-on-exercise)
  • Full SpMSSpV with Multi-core implementation

• Compiling High-Level Lang. to DSA
• Composing a Spatial Architecture
Outline

• Background: Decoupled Spatial Architectures
• Decoupled-Spatial Architecture Programming

• Advanced DSA Programming
  • Warm up: Support for data-dependent control/memory
  • Exercise 3: Sparse vector-vector dot-product (Hands-on-exercise)
  • Full SpMSpV with Multi-core implementation

• Compiling High-Level Lang. to DSA
• Composing a Spatial Architecture
Challenges with Irregular Workloads: SpMSpV Example

<table>
<thead>
<tr>
<th>Input Vector A (N)</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>3</th>
<th>0</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Vector C (N)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Input Vector A (N) = [0, 2, 0, 3, 0, 4, 0]
Input Matrix B (NxN) =

\[
\begin{bmatrix}
1 & 0 & 4 & 1 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 7 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 9 \\
0 & 3 & 3 & 1 & 1 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 0 & 3 \\
2 & 0 & 5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 6 & 0 & 0 \\
2 & 3 & 4 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
Challenges with Irregular Workloads: SpMSpV

Example

**Dot product in CSR format**

<table>
<thead>
<tr>
<th>A</th>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B[0]</th>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Output of conditional

\[
\text{Input Vector A (N)} = \begin{bmatrix} 0 & 2 & 0 & 3 & 0 & 4 & 0 \end{bmatrix}
\]

\[
\text{Input Matrix B (NxN)} = \begin{bmatrix}
1 & 0 & 4 & 1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 7 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 \\
0 & 3 & 3 & 1 & 1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 2 & 0 & 5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\
2 & 3 & 4 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\text{Output Vector C (N)} = \sum \begin{bmatrix}
3 & 1 & 2 & 4 & 5 & 1 & 3 & 4 \end{bmatrix}
\times
\begin{bmatrix}
1 & 0 & 4 & 1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 7 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 \\
0 & 3 & 3 & 1 & 1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 2 & 0 & 5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\
2 & 3 & 4 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
Challenges with Irregular Workloads: SpMSpV Example

**Dot product in CSR format**

**Input Vector A (N)**

\[
\begin{array}{ccccccc}
0 & 2 & 0 & 3 & 0 & 4 & 0 \\
\end{array}
\]

**Output Vector C (N)**

\[
\begin{array}{ccccccc}
3 & 1 & 0 & 4 & 1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 7 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 \\
0 & 3 & 3 & 1 & 1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 0 & 3 & 0 \\
2 & 0 & 5 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\
2 & 3 & 4 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[A\] \[\times\] \[B[0]\] = \[Output of conditional\]
Challenges with Irregular Workloads: SpMSpV

Example

Dot product in CSR format

Input Vector A (N)

Output Vector C (N)

Input Matrix B (NxN)

Conditional output 0 means no multiplication
Challenges with Irregular Workloads: SpMxSpV Example

**Dot product in CSR format**

<table>
<thead>
<tr>
<th>A</th>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B[0]</th>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Output of conditional: 0 means no multiplication

<table>
<thead>
<tr>
<th>Input Vector A (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 0 3 0 4 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Matrix B (NxN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 4 1 0 0 0</td>
</tr>
<tr>
<td>0 2 0 0 0 7 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 9</td>
</tr>
<tr>
<td>0 3 3 1 1 1 0 0</td>
</tr>
<tr>
<td>0 2 0 0 0 0 0 3</td>
</tr>
<tr>
<td>0 2 0 5 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 6 0 0</td>
</tr>
<tr>
<td>2 3 4 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Output Vector C (N):

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Diagram showing the computation and the output vector.
Challenges with Irregular Workloads: SpMSpV

Example

Dot product in CSR format

Input Vector A (N)

Output Vector C (N)

Input Matrix B (NxN)

Conditional output 0 means no multiplication
Challenges with Irregular Workloads: SpMSpV Example

**Dot product in CSR format**

- **Input Vector A (N)**: 0 2 0 3 0 4 0
- **Input Matrix B (NxN)**:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>4</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Output Vector C (N)**: 0 0 0 1

  - **Output of conditional**: 0 0 0 1
  - **Conditional output 0 means no multiplication**

- **Example**: total+=3*1
Challenges with Irregular Workloads: SpMSpV Example

**Dot product in CSR format**

<table>
<thead>
<tr>
<th>A</th>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B[0]</th>
<th>idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Output of conditional: 0 0 0 1 0

Conditional output 0 means no multiplication

Input Vector A (N)

\[
\begin{bmatrix}
0 & 2 & 0 & 3 & 0 & 4 & 0
\end{bmatrix}
\]

Input Matrix B (NxN)

\[
\begin{bmatrix}
1 & 0 & 4 & 1 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 7 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 9 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 3 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Output Vector C (N)

\[
\begin{bmatrix}
3 \\
6 \\
\end{bmatrix}
\]
Challenges with Irregular Workloads: SpMSpV Example

**Dot product in CSR format**

<table>
<thead>
<tr>
<th>A idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 5</td>
<td>2 3 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B[0] idx</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 3</td>
<td>1 4 1</td>
</tr>
</tbody>
</table>

Output of conditional: 0 0 0 1 0
Conditional output 0 means no multiplication

Input Vector A (N)

```
0 2 0 3 0 4 0
```

Input Matrix B (NxN)

```
<table>
<thead>
<tr>
<th></th>
<th>1 0 4 1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 0 0 0 7 0 0</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 9</td>
</tr>
<tr>
<td>9</td>
<td>0 3 3 1 1 0 0</td>
</tr>
<tr>
<td>4</td>
<td>0 2 0 0 0 0 3</td>
</tr>
<tr>
<td>0</td>
<td>2 0 5 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>1 0 0 0 6 0 0</td>
</tr>
<tr>
<td>6</td>
<td>2 3 4 0 0 0 0</td>
</tr>
</tbody>
</table>
```

Total+=3*1
Dot Product Implementation with Existing Features

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Dot Product Implementation with Existing Features

```c
// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
```
Dot Product Implementation with Existing Features

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Dot Product Implementation with Existing Features

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[+iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Dot Product Implementation with Existing Features

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++]out] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j

1. Indirect memory access

matchedInd[0:N]
Dot Product Implementation with Existing Features

```c
// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
```
Feature 1: Data-dependent memory access

- Semantics: $a[b[i]]$

```
[2 7 3 4]
```

```
```
Feature 1: Data-dependent memory access

• Semantics: $a[b[i]]$

Stream-dataflow intrinsic

SS_INDIRECT($\text{ind\_port}$, $\text{addr}$, $\text{num\_elem}$, $\text{input\_port}$);

for $data$ in $\text{ind\_port}$:
    calc_addr = $\text{addr} + data$
    send $data$ to $\text{input\_port}$
Feature 2: Data-dependent length

`SS_RECV(port, mem_addr)`

Acts as a barrier until a value from a `port` is written to a `memory location`
Feature 2: Data-dependent length

**SS_RECV**(port, mem_addr)

Acts as a barrier until a value from a port is written to a memory location.

**SS_RESET()**

 Resets all pending read requests and active streams except live data.
Dot product Impl. Using Indirect Access (manual/04_bad_join with N=4096, DENSITY=0.1)
Dot product Impl. Using Indirect Access (manual/04_bad_join with N=4096, DENSITY=0.1)

Simulator Time: 0.05903 sec
**Cycles:** 1798
Number of coalesced SPU requests: 0
Control Core Insts Issued: 37
Control Core Discarded Insts Issued: 17
Control Core Discarded Inst Stalls: 0.4595
Control Core Config Stalls: 0.07731
Control Core Wait Stalls:

**ACCEL 0 STATS ***
Commands Issued: 5
CGRA Instances: 868 -- **Activity Ratio:** 0.2286, DFGs / Cycle: 0.4828
For backcgra, Average throughput of all ports (overall): 0.1207, CGRA outputs/cgra busy cycles: 2.112
CGRA Insts / Computation Instance: 4.368
CGRA Insts / Cycle: 2.108 (overall activity factor)
**Mapped DFG utilization:** 0.1917
Dot product Impl. Using Indirect Access (manual/04_bad_join with N=4096, DENSITY=0.1)

Simulator Time: 0.05903 sec  
**Cycles: 1798**
Number of coalesced SPU requests: 0
Control Core Insts Issued: 37
Control Core Discarded Insts Issued: 17
Control Core Discarded Inst Stalls: 0.4595
Control Core Config Stalls: 0.07731
Control Core Wait Stalls:

ACCEL 0 STATS ***
Commands Issued: 5
CGRA Instances: 868 -- **Activity Ratio: 0.2286**, DFGs / Cycle: 0.4828
For backcgra, Average throughput of all ports (overall): 0.1207, CGRA outputs/cgra busy cycles: 2.112
CGRA Insts / Computation Instance: 4.368
CGRA Insts / Cycle: 2.108 (overall activity factor)
**Mapped DFG utilization: 0.1917**

• Low activity ratio

Why?
Cyclic Dependence Loop Limits the Throughput However Unnecessary
Cyclic Dependence Loop Limits the Throughput However Unnecessary

Cyclic dependence loop. Low throughput!!
Cyclic Dependence Loop Limits the Throughput However Unnecessary

Cyclic dependence loop. Low throughput!!
Cyclic Dependence Loop Limits the Throughput However Unnecessary

Cyclic dependence loop.
Low throughput!!
Cyclic Dependence Loop Limits the Throughput However Unnecessary

Cyclic dependence loop.
Low throughput!!
Cyclic Dependence Loop Limits the Throughput However Unnecessary

Cyclic dependence loop. Low throughput!!
Cyclic Dependence Loop Limits the Throughput However Unnecessary

Cyclic dependence loop. Low throughput!!
Feature 3: Data-dependent control in a PE

CLT $\rightarrow$ $f$ (inst_out, control_in) $\rightarrow$ reuse1, reuse2, discard, reset
Feature 3: Data-dependent control in a PE

CLT $\Rightarrow f\ (\text{inst\_out}, \text{control\_in}) \rightarrow \text{reuse1, reuse2, discard, reset}$
Feature 3: Data-dependent control in a PE

CLT \( \rightarrow \) \( f (\text{inst\_out}, \text{control\_in}) \rightarrow \text{reuse1, reuse2, discard, reset} \)
Feature 3: Data-dependent control in a PE

Example func unit: Compare

```c
if(inp1==INF && inp2==INF) return 3;
if (inp1 == inp2) return 0;
else if (inp1 > inp2) return 1;
else return 2;
```

Example LUT config:

<table>
<thead>
<tr>
<th>Self Ctrl i/p</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reuse1</td>
</tr>
<tr>
<td>2</td>
<td>Reuse2</td>
</tr>
<tr>
<td>3</td>
<td>Reset</td>
</tr>
</tbody>
</table>
Feature 3: Data-dependent control in a PE

**Example func unit: Compare**

```
if(inp1==INF && inp2==INF) return 3;
if (inp1 == inp2) return 0;
else if (inp1 > inp2) return 1;
else return 2;
```

**Example LUT config:**

<table>
<thead>
<tr>
<th>Self Ctrl i/p</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reuse1</td>
</tr>
<tr>
<td>2</td>
<td>Reuse2</td>
</tr>
<tr>
<td>3</td>
<td>Reset</td>
</tr>
</tbody>
</table>

**CGRA Processing Element**
Feature 3: Data-dependent control in a PE

Example func unit: Compare

if(inp1==INF && inp2==INF) return 3;
if (inp1 == inp2) return 0;
else if (inp1 > inp2) return 1;
else return 2;

Example LUT config:

<table>
<thead>
<tr>
<th>Self Ctrl i/p</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reuse1</td>
</tr>
<tr>
<td>2</td>
<td>Reuse2</td>
</tr>
<tr>
<td>3</td>
<td>Reset</td>
</tr>
</tbody>
</table>
Feature 3: Data-dependent control in a PE

Example func unit: Compare

```c
if(inp1==INF && inp2==INF) return 3;
if (inp1 == inp2) return 0;
else if (inp1 > inp2) return 1;
else return 2;
```

Example LUT config:

<table>
<thead>
<tr>
<th>Self Ctrl i/p</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reuse1</td>
</tr>
<tr>
<td>2</td>
<td>Reuse2</td>
</tr>
<tr>
<td>3</td>
<td>Reset</td>
</tr>
</tbody>
</table>
**Feature 3: Data-dependent control in a PE**

**Example func unit: Compare**

```
if(inp1==INF && inp2==INF) return 3;
if (inp1 == inp2) return 0;
else if (inp1 > inp2) return 1;
else return 2;
```

**Example LUT config:**

<table>
<thead>
<tr>
<th>Self Ctrl i/p</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reuse1</td>
</tr>
<tr>
<td>2</td>
<td>Reuse2</td>
</tr>
<tr>
<td>3</td>
<td>Reset</td>
</tr>
</tbody>
</table>

**CGRA Processing Element**
Feature 3: Data-dependent control in a PE

Example func unit: Compare

```c
if(inp1==INF && inp2==INF) return 3;
if (inp1 == inp2) return 0;
else if (inp1 > inp2) return 1;
else return 2;
```

Example LUT config:

<table>
<thead>
<tr>
<th>Self Ctrl</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reuse1</td>
</tr>
<tr>
<td>2</td>
<td>Reuse2</td>
</tr>
<tr>
<td>3</td>
<td>Reset</td>
</tr>
</tbody>
</table>
Feature 3: Data-dependent control in a PE

Example func unit: Compare

```
if(inp1==INF && inp2==INF) return 3;
if (inp1 == inp2) return 0;
else if (inp1 > inp2) return 1;
else return 2;
```

Example LUT config:

<table>
<thead>
<tr>
<th>Self Ctrl i/p</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reuse1</td>
</tr>
<tr>
<td>2</td>
<td>Reuse2</td>
</tr>
<tr>
<td>3</td>
<td>Reset</td>
</tr>
</tbody>
</table>
Outline

• Background: Decoupled Spatial Architectures
• Decoupled-Spatial Architecture Programming
• Advanced DSA Programming
  • Warm up: Support for data-dependent control/memory
  • Exercise 3: Sparse vector-vector dot-product (Hands-on-exercise)
  • Full SpMSpV and Multi-core implementation
• Compiling High-Level Lang. to DSA
• Composing a Spatial Architecture
Hands-on-Exercise: (Find index when index-data matched!!)

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Hands-on-Exercise: (Find index when index-data matched!!)

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Hands-on-Exercise: (Find index when index-data matched!!)

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++iout] = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Hands-on-Exercise: (Find index when index-data matched!!)

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++]out = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Hands-on-Exercise: (Find index when index-data matched!!)

// vanilla C code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        matchInd[++]iout = indA
        ++i; ++j
    elif (indA < indB):
        ++i
    else:
        ++j
Hands-on-Exercise: (Find index when index-data matched!!)
Hands-on-Exercise: (Find index when index-data matched!!)

Step 1: Write the DFG (manual/04_join/join.dfg)
Hands-on-Exercise: (Find index when index-data matched!!)

Step 1: Write the DFG (manual/04_join/join.dfg)

Step 2: Write control code (manual/04_join/main.cc)
Step 1: Append the DFG
(manual/04_join/join.dfg)

// 04_join/join.dfg
Input: indA
Input: indB

/* Write Compare operation here (Definition below:)
* if (op1 > op2) out = 1
* else if (op2 > op1) out = 2
* else if (op1 == INF) out = 3
* else out = 1
*/
match = Min64(indA, indB, ctrl=pred{1:b1|d, 2:b2|d})
is_end = ICmpEQ(pred, 3, ctrl=pred{1:d, 2:d})
matchedInd = Keep(match, 1, ctrl=is_end{1:d})
done = Keep(match, is_end)

Output: matchedInd, done
Step 1: Append the DFG
(manual/04_join/join.dfg)

// 04_join/join.dfg
Input: indA
Input: indB
/* Write Compare operation here (Definition below:)
* if (op1 > op2) out = 1
* else if (op2 > op1) out = 2
* else if (op1 == INF) out = 3
* else out = 1
*/
match = Min64(indA, indB, ctrl=pred{1:b1|d, 2:b2|d})
is_end = ICmpEQ(pred, 3, ctrl=pred{1:d, 2:d})
matchedInd = Keep(match, 1, ctrl=is_end)
done = Keep(match, is_end)

Output: matchedInd, done
Step 1: Append the DFG
(manual/04_join/join.dfg)

// 04_join/join.dfg
Input: indA
Input: indB
pred = Compare64(indA, indB, self={1:b1, 2:b2})
match = Min64(indA, indB, ctrl=pred{1:b1|d, 2:b2|d})
is_end = ICmpEQ(pred, 3, ctrl=pred{1:d, 2:d})
matchedInd = Keep(match, 1, ctrl=is_end{1:d})
done = Keep(match, is_end)

Output: matchedInd, done
Step 1: Append the DFG
(manual/04_join/join.dfg)

// 04_join/join.dfg
Input: indA
Input: indB
pred = Compare64(indA, indB,
self={1:b1, 2:b2})
match = Min64(indA, indB,
ctrl=pred{1:b1|d, 2:b2|d})
is_end = ICmpEQ(pred, 3,
ctrl=pred{1:d, 2:d})
matchedInd = Keep(match, 1,
ctrl=is_end{1:d})
done = Keep(match, is_end)

Output: matchedInd, done
Step 2: Append control code (manual/04_join/main.cc)

Useful intrinsics

// Read data from memory
SS_LINEAR_READ(addr, bytes, port);

// Write constant to the port
SS_CONST(port, data, times);

// Write output data to memory
SS_LINEAR_WRITE(port, bytes, addr);
Step 2: Append control code
(manual/04_join/main.cc)

```
// 04_join/main.cc
SS_LINEAR_READ(&indA[0], an*8, P_join_indA);
SS_CONST(P_join_indA, sentinel, 1);
SS_LINEAR_READ(&indB[0], bn*8, P_join_indB);
SS_CONST(P_join_indB, sentinel, 1);
SS_LINEAR_WRITE(P_join_matchedInd, &output[0], N*8);
SS_RECV(P_join_done, &done_flag);
SS_RESET();
SS_WAIT_ALL();
```
Run it!! ($ ./run.sh main.out)  
N=4096, DENSITY=0.1

Simulator Time: 0.02005 sec  
Cycles: 799  
Number of coalesced SPU requests: 0  
Control Core Insts Issued: 35  
Control Core Discarded Insts Issued: 17  
Control Core Discarded Inst Stalls: 0.4857  
Control Core Config Stalls: 0  
Control Core Wait Stalls:

ACCEL 0 STATS ***  
Commands Issued: 5  
CGRA Instances: 40 -- Activity Ratio: **0.9787**, DFGs / Cycle: 0.05006  
For backcgra, Average throughput of all ports (overall): 0.02503, CGRA outputs/cgra busy cycles: 0.05115  
CGRA Insts / Computation Instance: 60.65  
CGRA Insts / Cycle: 3.036 (overall activity factor)  
**Mapped DFG utilization: 0.6073**
Run it!! ($ ./run.sh main.out)
N=4096, DENSITY=0.1

Simulator Time: 0.02005 sec
Cycles: 799
Number of coalesced SPU requests: 0
Control Core Insts Issued: 35
Control Core Discarded Insts Issued: 17
Control Core Discarded Inst Stalls: 0.4857
Control Core Config Stalls: 0
Control Core Wait Stalls:

ACCEL 0 STATS ***
Commands Issued: 5
CGRA Instances: 40 -- Activity Ratio: **0.9787**, DFGs / Cycle: 0.05006
For backcgra, Average throughput of all ports (overall): 0.02503, CGRA outputs/cgra busy cycles: 0.05115
CGRA Insts / Computation Instance: 60.65
CGRA Insts / Cycle: 3.036 (overall activity factor)
Mapped DFG utilization: **0.6073**

• Index-match is done at a line rate
Outline

• Background: Decoupled Spatial Architectures
• Decoupled-Spatial Architecture Programming
• Advanced DSA Programming
  • Warm up: Support for data-dependent control/memory
  • Exercise 3: Sparse vector-vector dot-product (Hands-on-exercise)
  • Full SpMSpV and Multi-core implementation
• Compiling High-Level Lang. to DSA
• Composing a Spatial Architecture
Example 3: SpMSpV (Dot product kernel)

// vanilla code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        out += valA[i]* valB[j]
        ++i; ++j
    elif (indA[i] < indB[j]):
        ++i
    else:
        ++j
// vanilla code
while (i < an && j < bn): 
    if (indA[i] == indB[j]):
        out += valA[i] * valB[j]
        ++i; ++j
    elif (indA[i] < indB[j]):
        ++i
    else:
        ++j

matchIndA[0:N] matchIndB[0:N]
valA[0:N] valB[0:N]
indA[0:an] indB[0:bn]
cntIndA[0:N] cntIndB[0:an]
indA[0:an] indB[0:bn]

Recv (O)
Example 3: SpMSpV (Dot product kernel)

// vanilla code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        out += valA[i]* valB[j]
        ++i; ++j
    elif (indA[i] < indB[j]):
        ++i
    else:
        ++j
Example 3: SpMSpV (Dot product kernel)

// vanilla code
while (i < an && j < bn):
    if (indA[i] == indB[j]):
        out += valA[i]*valB[j]
        ++i; ++j
    elif (indA[i] < indB[j]):
        ++i
    else:
        ++j

Indirect access to values at matched indices
Example 3: SpMSpV (manual/05_spmv/)
(density=1 N=4096 DT=64)

Simulator Time: 0.2533 sec
Cycles: 4155

ACCEL 0 STATS ***
Commands Issued: 8
CGRA Instances: 8193 -- Activity Ratio: 0.9894, DFGs / Cycle: 1.972
For backcgra, Average throughput of all ports (overall): 0.6573, CGRA
outputs/cgra busy cycles: 1.993
CGRA Insts / Computation Instance: 3.5
CGRA Insts / Cycle: 6.902 (overall activity factor)
Mapped DFG utilization: 0.986
Example 3: SpMSpV (manual/05_spmv/) (density=0.1 N=4096 DT=64)

<table>
<thead>
<tr>
<th>Simulator Time: 0.04866 sec</th>
<th>• Can exploit sparsity at a near-line rate index match</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cycles:</strong> 924</td>
<td></td>
</tr>
</tbody>
</table>

**ACCEL 0 STATS ***

- Commands Issued: 8
- CGRA Instances: 87 -- **Activity Ratio:** 0.8452, DFGs / Cycle: 0.09416
- For backcgra, Average throughput of all ports (overall): 0.03139, CGRA outputs/cgra busy cycles: 0.1114
- CGRA Insts / Computation Instance: 37.29
- CGRA Insts / Cycle: 3.511 (overall activity factor)

**Mapped DFG utilization:** 0.5015
Example 4: SpMSpV (Parallelizing Dot products across cores)
Example 4: SpMSpV (Parallelizing Dot products across cores)

```
dotp (matrix[0], vector)
barrier
dotp (matrix[16], vector)
barrier
...```

Diagram:
```
Main Memory

Global Stream Engine

DSA Core

DSA Core

DSA Core

DSA Core

DSA Core

...```
Example 4: SpMSpV (Parallelizing Dot products across cores)

```c
// vanilla C code
int row = tid;
while (row < N):
    out_arr[row] = dotp (mat[row], vec)
    row += core_cnt
pthread_barrier_wait(&barr);
```

```
dotp (matrix[0], vector)
barrier
dotp (matrix[16], vector)
barrier
...```

Main Memory

Global Stream Engine

- DSA Core
- DSA Core
- DSA Core
- DSA Core
- DSA Core
- DSA Core

...
Example 4: SpMSpV (Parallelizing Dot products across cores)

// vanilla C code
int row = tid;
while (row < N):
    out_arr[row] = dotp (mat[row], vec)
    row += core_cnt
pthread_barrier_wait(&barr);

1. Core-to-core communication
Example 4: SpMSpV (Parallelizing Dot products across cores)

// vanilla C code
int row = tid;
while (row < N):
    out_arr[row] = dotp (mat[row], vec)
    row += core_cnt
pthread_barrier_wait(&barr);

1. Core-to-core communication

2. Synchronization
Example 4: SpMSpV (Parallelizing Dot products across cores)

// vanilla C code
int row = tid;
while (row < N):
    out_arr[row] = dotp (mat[row], vec)
    row += core_cnt
pthread_barrier_wait(&barr);

1. Core-to-core communication
2. Synchronization
3. Mapping data to scratchpad space
Multicore Explicit Communication Primitives

A) Explicit Multicast from a port to masked cores/ports

```
SS_REM_PORT(out_port, num_elem, mask, rem_port);
```
Multicore Explicit Communication Primitives

A) Explicit Multicast from a port to masked cores/ports

Core 1
- Ctrl
- Spad
- Vector ports

Core 2
- Ctrl
- Spad
- Vector ports

Core 3
- Ctrl
- Spad
- Vector ports

Core 4
- Ctrl
- Spad
- Vector ports

SS_REM_PORT(out_port, num_elem, mask, rem_port);

B) Implicit communication using global address space

Core 1
- Ctrl
- Spad
- Vector ports

Core 2
- Ctrl
- Spad
- Vector ports

Core 3
- Ctrl
- Spad
- Vector ports

Core 4
- Ctrl
- Spad
- Vector ports

SS_SCRATCH_READ(scr_addr, n_bytes, port)
... and other cmd as DMA
Multicore Configurable Data Mapping

SS_CFG_MEM_MAP(part_size, active_core_bv, map_type)

Example 1: cyclic

Part_size

Active_core_bv = “1111...”
Multicore Configurable Data Mapping

SS_CFG_MEM_MAP(part_size, active_core_bv, map_type)

Example 1: cyclic

Example 2: Subset of cores

Part_size

Active_core_bv = “1111...”

Active_core_bv = “001100...”
Multicore Configurable Data Mapping

SS_CFG_MEM_MAP(part_size, active_core_bv, map_type)

Example 1: cyclic
Part_size
Active_core_bv = “1111...”

Example 2: Subset of cores
Currently, map_type is default cyclic but we would like to study replication.
Active_core_bv = “001100...”
Example 4: SpMSpV multicore implementation

```c
// manual/07_spmv_multicore/main.cc
if(tid==0) {
    SS_LINEAR_READ(&indA[0], an, P_IND_1);
    uint64_t bdcast_mask=pow(2,num_cores)-1;
    SS_REM_PORT(P_IND_1, an*8, bdcast_mask, P_dotp_indA);
}
dotp_impl(); // no need to load vector here
SS_GLOBAL_WAIT(16);
```

Read at core 1 (i.e. tid=0) and broadcast to all cores
Results with Basic pthread Support (num_threads=4 density=0.1 N=8192)

*** ROI STATISTICS for CORE ID: 1 ***
Cycles: 7312
GRA Instances: 647 -- Activity Ratio: 0.4546, DFGs / Cycle: 0.08848
**Mapped DFG utilization: 0.25**

*** ROI STATISTICS for CORE ID: 2 ***
Cycles: 7328
CGRA Instances: 661 -- Activity Ratio: 0.4615, DFGs / Cycle: 0.0902
**Mapped DFG utilization: 0.2493**

*** ROI STATISTICS for CORE ID: 4 ***
Cycles: 7286
CGRA Instances: 641 -- Activity Ratio: 0.4547, DFGs / Cycle: 0.08798
**Mapped DFG utilization: 0.251**

*** ROI STATISTICS for CORE ID: 3 ***
Cycles: 7346
CGRA Instances: 635 -- Activity Ratio: 0.4597, DFGs / Cycle: 0.08644
**Mapped DFG utilization: 0.249**

- Lower DFG utilization due to network traffic (25% vs 45% for a single core)
Results: manual/07_spmv_multicore/ (num_threads=4 density=0.1 N=8192)

*** ROI STATISTICS for CORE ID: 1 ***
Cycles: 5316
CGRA Instances: 647 -- Activity Ratio: 0.585, DFGs / Cycle: 0.1217
Mapped DFG utilization: 0.3439

*** ROI STATISTICS for CORE ID: 2 ***
Cycles: 5384
CGRA Instances: 661 -- Activity Ratio: 0.5914, DFGs / Cycle: 0.1228
Mapped DFG utilization: 0.3393

*** ROI STATISTICS for CORE ID: 4 ***
Cycles: 5286
CGRA Instances: 641 -- Activity Ratio: 0.6035, DFGs / Cycle: 0.1213
Mapped DFG utilization: 0.3459

*** ROI STATISTICS for CORE ID: 3 ***
Cycles: 5281
CGRA Instances: 635 -- Activity Ratio: 0.6095, DFGs / Cycle: 0.1202
Mapped DFG utilization: 0.3463

- DFG utilization improves with multicast (34% vs 25%)
BACKUP SLIDES
Example 3: SpMV (Dot product kernel)

```plaintext
// 05_spmv/merge.dfg
Input: indA, indB, matchValA, matchValB

pred = Compare64(indA, indB, self={1:b1, 2:b2})
cntIndA = Acc64(1, 0, ctrl=pred{1:a})
cntIndB = Acc64(1, 0, ctrl=pred{2:a})
matchInd = Min64(cntIndA, cntIndB, ctrl=pred{1:b1|d, 2:b2|d, 3:d})
mult = Mul64(matchValA, matchValB)
done = Keep(pred, 1, ctrl=pred{1:d, 2:d})
O = Acc64(mult, ctrl=done{0:d, 3:r})

Output: matchIndA, matchIndB, O
```
Example 3: SpMV (Dot product kernel)

```c
// 05_spmv/main.cc
SS_LINEAR_READ(indA, an*8, P_dotp_indA);
SS_CONST(P_dotp_indA, sentinel, 1);
SS_LINEAR_READ(indB, bn*8, P_dotp_indB);
SS_CONST(P_dotp_indB, sentinel, 1);
SS_INDIRECT(P_dotp_matchIndA, valA, N, P_dotp_valA);
SS_INDIRECT(P_dotp_matchIndB, valB, N, P_dotp_valB);
SS_RECV(P_dotp_O, output[row]);
SS_RESET();
SS_WAIT_ALL();
```
Example 2: SpMV ($ ./run.sh main.out DENSITY=0.1 N=4096 DT=64)

Simulator Time: 2.59 sec
Cycles: 95651

ACCEL 0 STATS ***
Commands Issued: 512
CGRA Instances: 5276 -- Activity Ratio: 0.543, DFGs / Cycle: 0.05516
For backcgra, Average throughput of all ports (overall): 0.01839, CGRA outputs/cgra busy cycles: 0.1016
CGRA Insts / Computation Instance: 39.38
CGRA Insts / Cycle: 2.172 (overall activity factor)
Mapped DFG utilization: 0.3103

• Matrix occupies more space, needs double buffer for better hit rate
Example 2: Dot product
($=./run.sh main.out DENSITY=0.1 N=1024 DT=16)

<table>
<thead>
<tr>
<th>Simulator Time</th>
<th>0.04666 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>225</td>
</tr>
<tr>
<td>Number of coalesced SPU requests</td>
<td>0</td>
</tr>
<tr>
<td>Control Core Insts Issued</td>
<td>53</td>
</tr>
<tr>
<td>Control Core Discarded Insts Issued</td>
<td>18</td>
</tr>
<tr>
<td>Control Core Discarded Inst Stalls</td>
<td>0.3396</td>
</tr>
<tr>
<td>Control Core Config Stalls</td>
<td>0.04889</td>
</tr>
<tr>
<td>Control Core Wait Stalls:</td>
<td></td>
</tr>
</tbody>
</table>

ACCEL 0 STATS ***
| Commands Issued | 8                   |
| CGRA Instances  | 5 -- Activity Ratio | 0.9244, DFGs / Cycle | 0.02222 |
| For backcgra, Average throughput of all ports (overall): | 0.007407, CGRA outputs/cgра busy cycles: | 0.02404 |
| CGRA Insts / Computation Instance | 165.8               |
| CGRA Insts / Cycle | 3.684 (overall activity factor) |
| Mapped DFG utilization: | 0.5263             |