MODELING AND ANALYZING DEEP LEARNING ACCELERATOR DATAFLOWS WITH MAESTRO

Michael Pellauer*, Hyoukjun Kwon** and Tushar Krishna**

*Architecture Research Group, NVIDIA

**Georgia Institute of Technology
ACKNOWLEDGMENTS

Many slides from: Professors Joel Emer (NVIDIA, MIT) and Vivienne Sze’s (MIT) class
  - 6.082/6.888 Hardware Architectures for Deep Learning (Fall, 2017)
Also includes slides from: Angshuman Parashar, Senior Research Scientist (NVIDIA)

Also influenced by:

Steve Keckler (NVIDIA)
Jason Clemons (NVIDIA)
Sophia Shao (NVIDIA)
Cristopher Fletcher (UIUC)

Former Nvidia interns:

Yu-Hsin Chen (MIT)
Anurag Mukkara (MIT)
Animesh Jain (U Mich.)
Motivation and Background
  • Why architecture researchers should care about dataflow

Concepts in Tiling, Blocking, and Scheduling
  • Basic example: 1D-convolution
  • Adding intermediate staging buffers
  • Extending to full CNN layers

MAESTRO: Modeling Accelerator Efficiency via Spatio-Temporal Resource Occupancy
  • Analytic approach for architectural exploration
  • Hands-on lab
**MOTIVATION: DATA MOVEMENT**

**Why it’s important**

<table>
<thead>
<tr>
<th>Energy costs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit Integer Multiply</td>
<td>1x</td>
</tr>
<tr>
<td>Fetch two 8-bit operands from DRAM</td>
<td>~100x</td>
</tr>
<tr>
<td>Fetch two 8-bit operands from large SRAM</td>
<td>~10x</td>
</tr>
</tbody>
</table>

**Fortunately...**

<table>
<thead>
<tr>
<th>VGG16 conv 3_2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply Add Ops</td>
<td>1.85 Billion</td>
</tr>
<tr>
<td>Weights</td>
<td>590 K</td>
</tr>
<tr>
<td>Inputs</td>
<td>803 K</td>
</tr>
<tr>
<td>Outputs</td>
<td>803 K</td>
</tr>
</tbody>
</table>
MAPPING REUSE TO HARDWARE

7-dimensional network layer

- 7D Computation Space
  - \( R \times S \times X \times Y \times C \times K \times N \)

- 4D Operand / Result Spaces
  - Weights - \( R \times S \times C \times K \)
  - Inputs - \( X \times Y \times C \times N \)
  - Outputs - \( X' \times Y' \times K \times N \)

2D hardware array

Algorithmic Reuse

Hardware Reuse

Temporal

• DRAM
  • Buf
  • RF

Multicast

Forwarding

Millions of non-trivial mappings
Efficiency is dependent on concrete dimension sizes
LEARNING ABOUT DATAFLOWS
PEDAGOGAL EXAMPLE: 1-D CONVOLUTION

Weights $S$ * Inputs $X$ = Outputs $X' = X \cdot \text{ceil}(S/2)^\dagger$

* The terms “Output” and “Partial Sum” used interchangeably

$\dagger$ Assuming: ‘valid’ style convolution

```c
int i[X];  // Input activations
int w[S];  // Filter weights
int o[X']; // Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s] * w[s];
    }
}
```

How often does the datapath change the weight and input?
Every cycle

Output?
Every R cycles: “Output stationary”
WHAT DO WE MEAN BY “STATIONARY”?  

The datatype (and dimension) that changes most slowly

Imprecise analogy: think of data transfers as a wave with “amplitude” and “period”

• The stationary datatype has the longest period (locally held tile changes most slowly)
• Note: like waves, also can have harmful “constructive interference” (bursts)
• Often corresponds to datatype that is “done with” earliest without further reloads

Later we will see how intermediate staging buffers reduce both bandwidth and energy

Note: the “stationary” name is meant to give intuition, not to be a complete specification of all the behavior of a dataflow
FROM “LOOP NEST” TO DATAFLOW

Weights \* Inputs = Outputs

\[ S \times X = X' = X - \text{ceil}(S/2) \]

```
int i[X];  # Input activations
int w[S];  # Filter weights
int o[X']; # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

* Because we don’t care about where precision/saturation issues occur - usually choose data sizes such that it never happens

[See NVDLA’s 48-bit accumulators for 16-bit operands]
Weights $S$ \* Inputs $X$ = Outputs $X' = X - \text{ceil}(S/2)$

```
int i[X];  // Input activations
int w[S];  // Filter weights
int o[X']; // Output activations

for (s = 0; s < S; s++) {
    for (x = 0; x < X'; x++) {
        o[x] += i[x+s]*w[s];
    }
}
```

What dataflow is this? Weight stationary
MORE DATAFLOWS

Weights \( S \) * Inputs \( X \) = Outputs \( X' = X - \text{ceil}(S/2)^\dagger \)

```
int i[X];  # Input activations
int w[S];  # Filter weights
int o[X']; # Output activations

for (x = 0; x < X'; x++) {
    for (s = 0; s < S; s++) {
        o[x] += i[x+s]*w[s];
    }
}
```

How can we implement input stationary with no input index?
INPUT STATIONARY

Weights \( S \) \* Inputs \( X \) = Outputs \( X' = X - \text{ceil}(S/2) \)

```
int i[X];     # Input activations
int w[S];     # Filter weights
int o[X'];    # Output activations

for (x = 0; x < X; x++) {
    for (s = 0; s < S; s++) {
        o[x-s] += i[x]*w[s];
    }
}
```

Beware w-r must be >= 0 and <E
SIMPLE MODEL FOR MAPPING DATAFLOWS TO HARDWARE

Weights \* Inputs = Outputs

Weights

S

X

X' = X-ceil(S/2)

Common metric | Weights | Inputs | Outputs / Partial Sums
--- | --- | --- | ---
Alg. Min. accesses to backing store (MINALG) | S | X | X'
Maximum operand uses (MAXOP) | SX' | SX' | SX'
1D CONVOLUTION - SUMMARY

Weights * Inputs = Outputs

Common metric | Weights | Inputs | Outputs / Partial Sums
---|---|---|---
Size = Alg. Min. accesses | S | X | X’
Maximum operand uses | SX’ | SX’ | SX’

BUFSIZE-1D (Buffer size for zero re-fetch)

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>Weights</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-stationary</td>
<td>1</td>
<td>X’</td>
<td>X’</td>
</tr>
<tr>
<td>Input-stationary</td>
<td>S</td>
<td>1</td>
<td>S</td>
</tr>
<tr>
<td>Output-stationary</td>
<td>S</td>
<td>S</td>
<td>1</td>
</tr>
</tbody>
</table>

BUFMULT-1D (#times full buffer is accessed)

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>Weights</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-stationary</td>
<td>SX’</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Input-stationary</td>
<td>X’</td>
<td>SX’</td>
<td>X’</td>
</tr>
<tr>
<td>Output-stationary</td>
<td>X’</td>
<td>X’</td>
<td>SX’</td>
</tr>
</tbody>
</table>

BSACCESS-1D (Backing store access counts)

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>Weights</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-stationary</td>
<td>S</td>
<td>X</td>
<td>X’</td>
</tr>
<tr>
<td>Input-stationary</td>
<td>S</td>
<td>X</td>
<td>X’</td>
</tr>
<tr>
<td>Output-stationary</td>
<td>S</td>
<td>X</td>
<td>X’</td>
</tr>
</tbody>
</table>

BUFACCESS-1D (Buffer access expense) = BUFMULT * BUFSIZE

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>Weights</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-stationary</td>
<td>SX’(1)</td>
<td>S(X’)</td>
<td>S(X’)</td>
</tr>
<tr>
<td>Input-stationary</td>
<td>X’(S)</td>
<td>SX’(1)</td>
<td>X’(S)</td>
</tr>
<tr>
<td>Output-stationary</td>
<td>X’(S)</td>
<td>X’(S)</td>
<td>SX’(1)</td>
</tr>
</tbody>
</table>

Note: product always equals SX’
### 1D CONVOLUTION - SUMMARY

**Weights** \( S \) \( \times \) **Inputs** \( X \sim X' \) = **Outputs** \( X' \)

**Buffer size for zero re-fetch**

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>Weights</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-stationary</td>
<td>1</td>
<td>( X' )</td>
<td>( X' )</td>
</tr>
<tr>
<td>Input-stationary</td>
<td>( S )</td>
<td>1</td>
<td>( S )</td>
</tr>
<tr>
<td>Output-stationary</td>
<td>( S )</td>
<td>( S )</td>
<td>1</td>
</tr>
</tbody>
</table>

**Buffer accesses**

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>Weights</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight-stationary</td>
<td>( SX'(1) )</td>
<td>( S(X') )</td>
<td>( S(X') )</td>
</tr>
<tr>
<td>Input-stationary</td>
<td>( X'(S) )</td>
<td>( SX'(1) )</td>
<td>( X'(S) )</td>
</tr>
<tr>
<td>Output-stationary</td>
<td>( X'(S) )</td>
<td>( X'(S) )</td>
<td>( SX'(1) )</td>
</tr>
</tbody>
</table>

**Buffer access energy:**

\[
WS = SX'[f(1) + f(X') + 2f(X')] \\
IS = SX'[f(S) + f(1) + 2f(S)] \\
OS = SX'[f(S) + f(S) + 2f(1)]
\]

\( f(x) = \) energy cost of accessing a RAM structure of size \( x \)

---

**Significant difference in buffer access energy cost based on dataflow**

**But what if the provisioned buffering is smaller than required?**
GETTING MORE REALISTIC
MULTI-LAYER BUFFERING
1-D CONVOLUTION - BUFFERED

Weights * Inputs = Outputs

\[ X' = X - \text{ceil}(X/2) \]

\[
\begin{align*}
\text{i}[X]; & \quad \# \text{Input activations} \\
\text{w}[S]; & \quad \# \text{Filter Weights} \\
\text{o}[X']; & \quad \# \text{Output activations}
\end{align*}
\]

// Level 1
for (x1 = 0; x1 < X'1; x1++) {
    for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                x = x1 * X'0 + x0;
                s = r1 * R0 + r0;
                o[x] += i[x+s] * w[s];
            }
        }
    }
}

Note X' and S are factored so:
\[ X'0 \times X'1 = X' \]
\[ S0 \times S1 = S \]
Energy of a buffer access is a function of the size of the buffer.

Each buffer level’s energy is proportional to the number of accesses at that level.

For level 0 that is all the operands to the Datapath.

For level L>0 there are three components:
- Data arriving from level L+1
- Data that needs to be transferred to level L-1
- Data that is returned from level L-1

```
// Level 1
for (x1 = 0; x1 < X’1; x1++) {
    for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X’0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0] * w[s1*S0+s0];
            }
        }
    }
}
```
// Level 1
for (x1 = 0; x1 < X’1; x1++) {
    for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X’0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0] * w[s1*S’0+s0];
            }
        }
    }
}

Weights

s1=0

s1++

s1++

s1=0

Next e1 iteration
MAPPING - WEIGHT ACCESS COSTS

Level 0 reads
Per level 1 iteration -> $X'0*S0$ weight reads
Times $X'1*S1$ level 1 iterations
Total reads = $(X'0*S0)*(X'1*S1) = (X'0*X'1)*(S0*S1) = SX'$ reads

Level 1 to 0 transfers
Per level 1 iteration -> $S0$ weights transferred
Times same number of level 1 iterations = $X'1 * S1$
Total transfers -> $S0*(X'1*S1) = X'1*(S0*S1) = SX'1$

Disjoint/partitioned reuse pattern
MAPPING - INPUT ACCESS COSTS

// Level 1
for (x1 = 0; x1 < X'1; x1++) {
    for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
            }
        }
    }
}

s1=0
X'0+S0

s1++
S0
X'0+S0

s1++
S0
X'0+S0

Next x1 iteration

Input halo!

Sliding window
MAPPING - INPUT ACCESS COSTS

Level 0 reads
Per level 1 iteration -> X’0+S0 inputs reads
Times X’1*S1 level 1 iterations
Total reads = X’1*S1*(X’0+S0) = ((X’1*X’0)*S1)+(X’1*(S1*S0)) = X’*S1+X’1*S reads

Level 1 to 0 transfers
For s=0, X’0+S0 inputs transferred
For each of the following S1-1 iterations another S0 inputs transferred
So total per x1 iteration is: X’0+S0*S1 = X’0+S inputs
Times number of x1 iterations = X’1
So total transfers = X’1*(X’0+S) = (X’1*X’0)+X’1*S = X’+X’1*S

Sliding window/partitioned reuse pattern
MAPPING - OUTPUT ACCESS COSTS

// Level 1
for (x1 = 0; x1 < X’1; x1++) {
    for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X’0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                o[x1*X’0+x0] += i[x1*X’0+x0 + s1*S0+s0] * w[s1*S0+s0];
            }
        }
    }
}

Outputs

s1=0

s1+=

Next x1 iteration
MAPPING - OUTPUT ACCESS COSTS

Level 0 writes
Due to level 0 being ‘output stationary’ only X’0 writes per level 1 iteration
Times X’1*S1 level 1 iterations
Total writes = X’0*(X’1*S1) = (X’0*X’1)*S1 = X’*S1 writes

Level 0 to 1 transfers
After each S1 iterations a completed partial sum for X’0 outputs are transferred
There are X’1 chunks of S1 iterations
So total is X’1*X’0 = X’ transfers
MAPPING DATA COST SUMMARY

```
// Level 1
for (x1 = 0; x1 < X'1; x1++) {
    for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0]* w[s1*S0+s0];
            }
        }
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Level 0</th>
<th>Level 1 to 0 transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weight Reads</strong></td>
<td>SX’</td>
<td>SX’1</td>
</tr>
<tr>
<td><strong>Input Reads</strong></td>
<td>X’ * S1+ X’1 * S</td>
<td>X’+X’1*S</td>
</tr>
<tr>
<td><strong>Output Reads</strong></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Output Writes</strong></td>
<td>X’ * S1</td>
<td>X’</td>
</tr>
</tbody>
</table>
How will this be reflected in the loop nest?  

New ‘level’ of loops
INTERMEDIATE BUFFERING

L1 Weights
L1 Inputs
L1 Outputs

L0 Weights
L0 Inputs
L0 Outputs

PE
PE

nvidia
1-D CONVOLUTION - SPATIAL

Weights * Inputs = Outputs

\[ X' = X - \text{ceil}(S/2) \]

```
int i[X];  // Input activations
int w[S];  // Filter Weights
int o[X']; // Output activations

// Level 1
parallel-for (x1 = 0; x1 < X'1; x1++) {
    parallel-for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
            }
        }
    }
}
```

Note:
- \( X'0 \times X'1 = X' \)
- \( S0 \times S1 = S \)
- \( X'1 = 2 \)
- \( S1 = 1 \Rightarrow s1 = 0 \)
1-D CONVOLUTION - SPATIAL

```
int i[X];  // Input activations
int w[S];  // Filter Weights
int o[X']; // Output activations

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
    }
  }
}
```
Implementation opportunity? Yes, single fetch and multicast
1-D CONVOLUTION - SPATIAL

// Level 1
parallel-for (x1 = 0; x1 < 2; x1++) {
// Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
        }
    }
}

How do we recognize multicast opportunities?

Indices independent of spatial index
SPATIAL PES

L0 Weights
L0 Inputs
L0 Outputs

PE0
>>w[0]
>>i[0]
>>w[1]
>>i[1]
>>w[2]
>>i[2]
<<o[0]

PE1
>>w[0]
>>i[0]
>>w[1]
>>i[1]
>>w[2]
>>i[2]
<<o[0]

Implementation opportunity? Parallel fetch

Assuming S=3
SPATIAL PARTITIONING WEIGHTS

Weights \(*\) Inputs = Outputs

\(X' = X - \text{ceil}(S/2)\)

```c
int i[W];    // Input activations
int w[R];    // Filter Weights
int o[E];    // Output activations

// Level 1
parallel for (x1 = 0; x1 < X'1; x1++) {
    parallel-for (s1 = 0; s1 < S1; s1++) {
        // Level 0
        for (x0 = 0; x0 < X'0; x0++) {
            for (s0 = 0; s0 < S0; s0++) {
                o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
            }
        }
    }
}
```

Note:
- \(X'0*X'1 = X'\)
- \(S0*S1 = S\)
- \(X'1 = 1 \Rightarrow x1 = 0\)
- \(S0 = 1, S1 = 2\)
**SPATIAL PARTITIONING WEIGHTS**

Weights * Inputs = Outputs

\[ S \times X = X' = X - \text{ceil}(S/2) \]

```
int i[X];    // Input activations
int w[S];    // Filter Weights
int o[X'];   // Output activations

// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
    // Level 0
    for (x0 = 0; x0 < X'0; x0++) {
        for (s0 = 0; s0 < S0; s0++) {
            o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
        }
    }
}
```

Note:
- \(X'0 \times X'1 = X'\)
- \(S0 \times S1 = S\)
1-D CONVOLUTION - SPATIAL

```
// Level 1
parallel-for (s1 = 0; s1 < 2; s1++) {
  // Level 0
  for (x0 = 0; x0 < X'0; x0++) {
    for (s0 = 0; s0 < S0; s0++) {
      o[x1*X'0+x0] += i[x1*X'0+x0 + s1*S0+s0] * w[s1*S0+s0];
    }
  }
}
```

How do we handle same index for output in multiple PEs?  Spatial summation…

Other multicast opportunities?  No
SPATIAL PES

What if hardware cannot do a spatial sum?

Illegal mapping!
2D CONVOLUTION - ADDING DIMENSIONS

- Weights: $R \times S$
- Inputs: $X \times Y$
- Partial Sums: $X' = X - \text{ceil}(S/2)$, $Y' = Y - \text{ceil}(R/2)$
CNN PROBLEM FORMULATION - INPUT CHANNELS

\[ X' = X - \text{ceil}(S/2) \]

\[ Y' = Y - \text{ceil}(R/2) \]
CNN PROBLEM FORMULATION - INPUT CHANNELS

\[ X' = X - \text{ceil}(S/2) \]

\[ Y' = Y - \text{ceil}(R/2) \]
CNN PROBLEM FORMULATION - OUTPUT CHANNELS

Weights

Inputs

Partial Sums

\[ X' = X - \text{ceil}(S/2) \]

\[ Y' = Y - \text{ceil}(R/2) \]
Naïve 6-layer* for-loop implementation:

```c
for(int ki=0; ki<K; ki++) //Loop_K
   for(int ci=0; ci<C; ci++) //Loop_C
      for(int yi=0; yi<Y; yi++) //Loop_Y
         for(int xi=0; xi<X; xi++) //Loop_X
            for(int ri=0; ri<R; ri++) //Loop_R
               for(int si=0; si<S; si++) //Loop_S
                  O[k][x][y] += W[k][c][r][s] * I[c][y+r][x+s]
```

Gigantic space of potential tilings and dataflows – how to explore?

- Cycle-accurate modeling of realistic dimensions and fabric sizes too slow
- Solution: use an analytic model specialized for CNNs

*Batch size would be a 7th loop (if available)
MAESTRO: A DATAFLOW COST MODEL
OVERVIEW OF MAESTRO APPROACH
Modeling Accelerator Efficiency via Spatio-Temporal Resource Occupancy

Analytic model:
- Does not execute or simulate
- Answers obtained via mathematical equations
- First-cut estimate for focusing later simulation “deep dives”
- Validation is ongoing work

Not a general cost model:
- By specializing workloads and execution substrate we gain efficiency and productivity
ARCHITECTURE AND CNN LAYER DESCRIPTION

- Architecture assumptions:
  - Spatial array of PEs, each with L1 buffer
  - Shared L2 buffer, via on-chip network

- Model Input Parameters:
  - Number of PEs
  - Vector width
  - On-chip network bandwidth cap
  - Average number of hops
  - Does OCN support multicast?

- Layer:
  - Domain-specific convolutional layer description:
DESCRIBING DATAFLOWS IN MAESTRO

Combine 5 possible pragmas to represent CNN loop nest tiling

- **Temporal_map** (SIZE, OFFSET) Var
  - T0
  - T1
  - Tiles
  - Ta=0
  - Ta=1

- **Spatial_map** (SIZE, OFFSET) Var
  - T0
  - T1
  - Tiles
  - Ta=0
  - Ta=1

- **Tile** (SIZE) Var
  - T0
  - T1
  - T2
  - T3
  - T4
  - T5
  - Ta0
  - Ta1
  - Ta2
  - Ta3
  - Ta4
  - Tile structure before Tile pragma
  - Tile structure at loop A

- **Merge** Var
  - for(b=0; b<3; b++)
  - for(a=0; a<2; a++)

- **Unroll** Var
  - for(b=0; b<B; b++)
  - for(a=0; a<1; a++)
  - f(a,b)

**NOTE:** all SIZE and OFFSET are in terms of tiles
Var = {K, C, X, Y, R, S}
# DATAFLOW EXAMPLES

Our best effort to reproduce the representations in the literature

<table>
<thead>
<tr>
<th>Accelerator</th>
<th>Dataflow Strategy</th>
<th>Dataflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example for this work</td>
<td>No Local Reuse (NLR)</td>
<td><code>TEMPORAL_MAP (4,4) \(K\) \rightarrow TEMPORAL_MAP (1,1) \(C\) \rightarrow TEMPORAL_MAP (1,1) \(Y\) \rightarrow TEMPORAL_MAP (1,1) \(R\) \rightarrow TEMPORAL_MAP (1,1) \(S\) \rightarrow SPATIAL_MAP (1,1) \(X\)</code></td>
</tr>
<tr>
<td>Example for this work</td>
<td>Weight Stationary (WS)</td>
<td><code>TEMPORAL_MAP (1,1) \(K\) \rightarrow TEMPORAL_MAP (1,1) \(C\) \rightarrow TEMPORAL_MAP (3,3) \(Y\) \rightarrow SPATIAL_MAP (1,1) \(X\) \rightarrow UNROLL \(R\) \rightarrow UNROLL \(S\)</code></td>
</tr>
<tr>
<td>ShiDiannao [9]</td>
<td>Output Stationary (OS)</td>
<td><code>TEMPORAL_MAP (1,1) \(K\) \rightarrow TEMPORAL_MAP (1,1) \(C\) \rightarrow TEMPORAL_MAP (1,1) \(Y\) \rightarrow SPATIAL_MAP (3,1) \(X\) \rightarrow UNROLL \(R\) \rightarrow UNROLL \(S\)</code></td>
</tr>
<tr>
<td>Eyeriss [3]</td>
<td>Row-stationary</td>
<td><code>TEMPORAL_MAP (1,1) \(K\) \rightarrow TEMPORAL_MAP (1,1) \(C\) \rightarrow TEMPORAL_MAP (3,1) \(Y\) \rightarrow SPATIAL_MAP (1,1) \(X\) \rightarrow TEMPORAL_MAP (1,1) \(R\) \rightarrow TILE (SZ(S)) \(S\) \rightarrow TEMPORAL_MAP (SZ(S),SZ(S)) \(S\)</code></td>
</tr>
<tr>
<td>NVIDIA [6]</td>
<td>Weight Stationary</td>
<td><code>TEMPORAL_MAP (1,1) \(R\) \rightarrow TEMPORAL_MAP (1,1) \(S\) \rightarrow TEMPORAL_MAP (64,64) \(C\) \rightarrow TEMPORAL_MAP (1,1) \(Y\) \rightarrow TEMPORAL_MAP (1,1) \(X\) \rightarrow SPATIAL_MAP (1,1) \(K\)</code></td>
</tr>
</tbody>
</table>
ANALYSIS MODEL

Ignore this “eyechart” slide - Just intuition as to how it works

Definitions
S-tile: Spatial tile, tiles at the innermost spatially mapped loop
NumTile: Total number of S-tiles
NumTempFold: Number of iterations of the entire innermost loop with spatial_map
NumSpatialFold: Number of implicit folding (due to insufficient number of tiles for a spatial mapping) within the innermost loop with spatial_map
NumSpatialClass: Number of data points of the data class accessed in a S-tile
NumSpatialVar: The entities size of the Var dimension in the given neural network layer
TSzVari: Number of assigned variable Var within a S-tile
UTSzVari: Number of assigned unique variable Var within a S-tile (each considered)

TSzVari := match getPragVar
            [Spatial_Map(5, 0), Temporal_Map(5, 0), Tc
             Uref := getUpperBound(var)
             |Merge := getSze(var) / getSize(getUplCopVar(var))

UTSzVari := match getPragVar
            [Spatial_Map(5, 0), Temporal_Map(5, 0), Tc = 0 T + 1
default := TSzVari];
EXAMPLES USING MAESTRO OUTPUT

Combining with an energy model:

NOTE: this represents the performance of the dataflow on normalized PE substrate
- Not representative of the performance of the original architecture
MAESTRO command-line parameters:

```
$naestro <layer>.m <dataflow>.m <vector_width> <max_bandwidth> <support_multicast?> <avg_hops> <num_peq>
```

Pre-provided layers:
```
pwc_fpe_conv1.m pwc_fpe_conv6.m vgg16_conv1.m vgg16_conv4.m vgg16_conv9.m
pwc_fpe_conv4.m vgg16_conv11.m vgg16_conv2.m vgg16_conv6.m
```

Pre-provided dataflows:
```
bal.m dla.m nlr.m nsd.m prs.m rs.m sd.m ws2.m ws.m
```

Output example:
```
<< Buffer Analysis >>
L1BufferReq: 66.000 Byte
L1BufferReq: 2.060 KB
L1RdCount: 59305623552
L1RdWeight: 29595009024
L1RdInput: 29595009024
L1RdSum: 115605504
L1WrCount: 3814981632
L1WrWeight: 1849688064
L1WrInput: 1849688064
L1WrSum: 115605504

<< NoC Analysis >>
L1 to L2 NoCBW: 16 elements per cycle
L2 to L1 NoCBW: 512 elements per cycle
<< Performance Analysis >>
Roofline throughput: 1.000000 GFLOPS with 1GHz clock
L1 to L2 sum: 16.000000
L2 to L1 delay: 0.000000
L1 to L2 delay: 10.000000
Compute delay: 426.000000
total latency: 3150249984 cycles
```
MAESTRO HANDS-ON LAB
(Don’t forget about the 10:00-10:30 coffee break!)

Exercise #1: Sweep throughput as a function of the number of PEs 1-256 (doubling)
- e.g. ./maestro_layerDescription/vgg16_convt2.m dataflowDescription/dla.m 1 8 1 8 <N>
- At what point do you see diminishing returns? Why? Can you fix it?

Exercise #2: Sweep numeric parameter sizes of a dataflow for a fixed architecture
- e.g. tmap (4, 4) K => tmap (8, 8) K => tmap (16, 16) K, etc.
- What effect does this have on results? Any differences for early/late layers?

Exercise #3: Sweep dataflows for a fixed layer and architecture
- e.g. ./maestro_layerDescription/vgg16_convt2.m <dataflow>.m 1 8 1 8 64
- Which dataflow has min/max reads for L1 weights/inputs/sums? Min/max transfers?

Challenge: Design a new dataflow with the following properties:
- No more than 16 PEs, bandwidth 4, L2 to L1 BW < 0.04 per cycle, 1 sMap and 5 tMaps
- Better than all provided dataflows on late layers, but can be worse on early layers
CONCLUSIONS

Understanding CNN dataflow and tiling concepts is critical for computer architecture researchers focusing on domain-specific accelerators.

Caches are generally considered too expensive (area+power) but we can easily make up the difference using workload knowledge.

For an example of exploiting advanced dataflow knowledge, come see:

UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition - Kartik Hegde (UIUC), Jiyong Yu (UIUC), Rohit Agrawal (UIUC), Mengjia Yan (UIUC), Michael Pellauer (NVIDIA), Christopher Fletcher (UIUC)

Session 8A: Machine Learning Systems 1 (Wednesday, 9:30 AM)