DNN Dataflow Choice Is Overrated

Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qioyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz.
Stanford University, Tsinghua University
2018

Presenter : Derrick Blakely

https://qdata.github.io/deep2Read

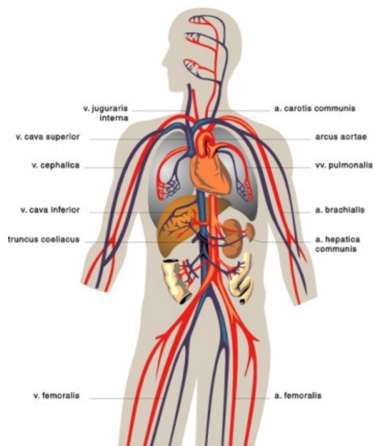
Outline

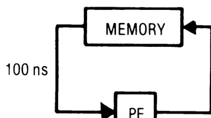
1. Background: Spatial Architectures
2. Background: Halide
3. DNN Dataflow is Overrated
4. Generating Hardware Designs Using Halide
5. Experiments
6. Conclusion

Outline

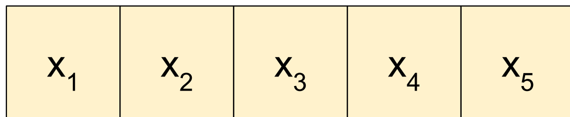
1. Background: Spatial Architectures
2. Background: Halide
3. DNN Dataflow is Overrated
4. Generating Hardware Designs Using Halide
5. Experiments
6. Conclusion

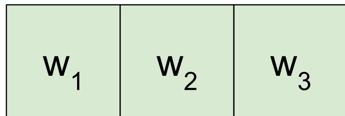
Systolic Arrays - Dataflow Analogous to Blood Circulation

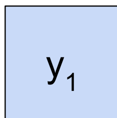




Convolution - A Use Case for Systolic Arrays

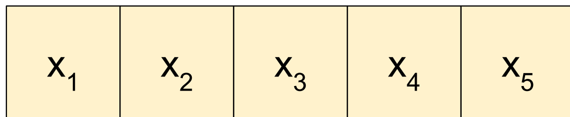


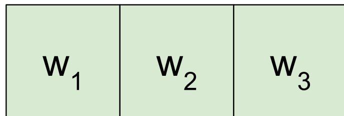


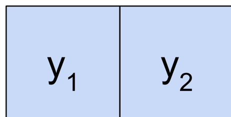


$w_1x_1+w_2x_2+w_3x_3$

Convolution - A Use Case for Systolic Arrays



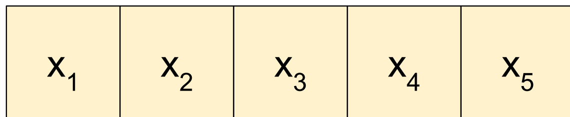


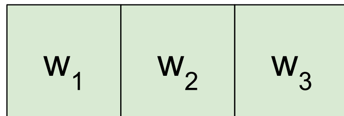


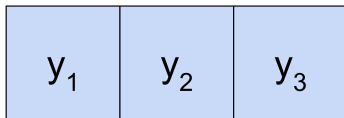
$w_1x_1+w_2x_2+w_3x$

3 $w_1x_2+w_2x_3+w_3x$

Convolution - A Use Case for Systolic Arrays



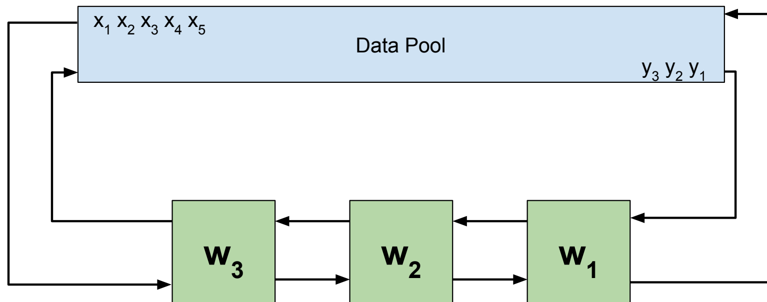




$w_1x_1+w_2x_2+w_3x_3$ $w_1x_3+w_2x_4+w_3x_5$

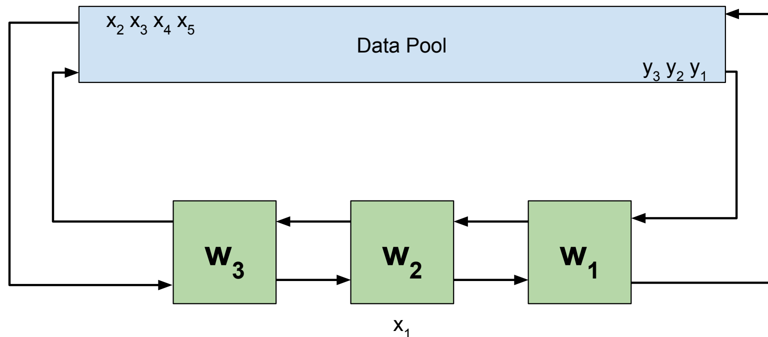
$w_1x_2+w_2x_3+w_3x_4$

1D Systolic Convolution

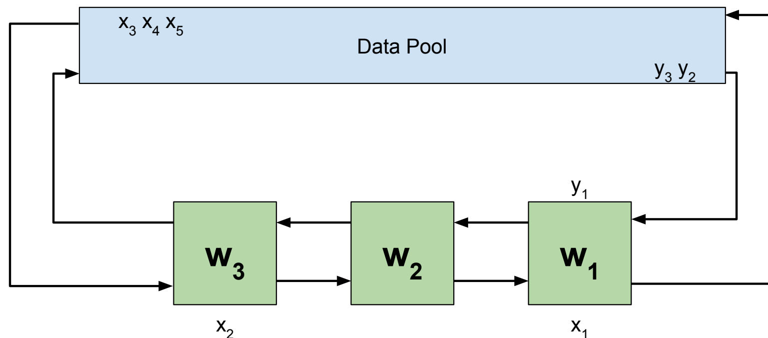


Reads: 0
Writes: 0

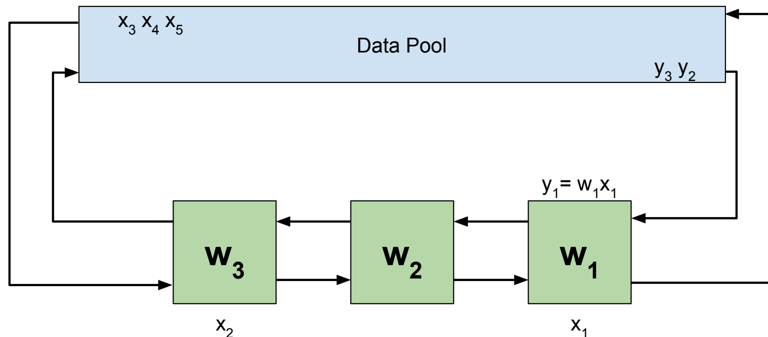
1D Systolic Convolution



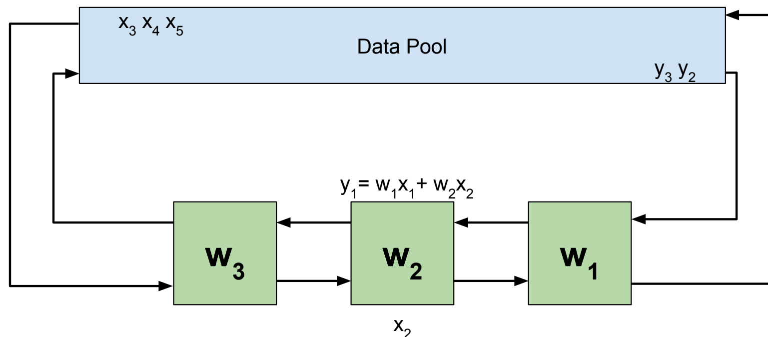
1D Systolic Convolution



1D Systolic Convolution

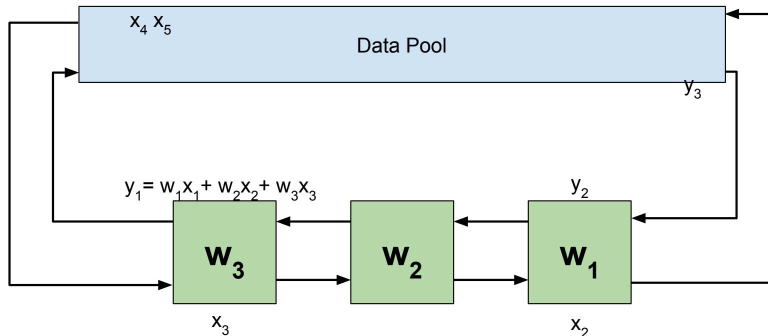


1D Systolic Convolution

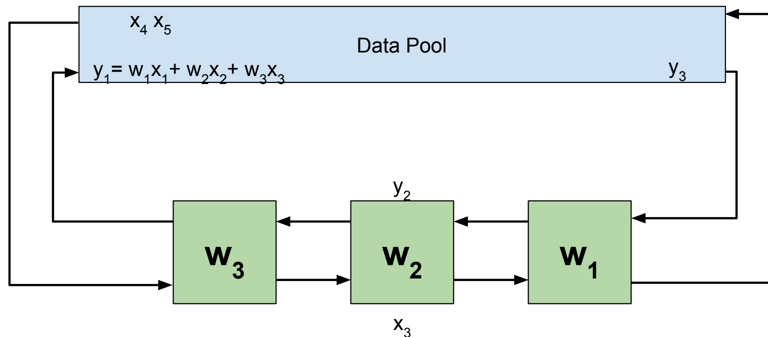


Reads: 3
Writes: 0

1D Systolic Convolution



1D Systolic Convolution



Why is this good?

- Avoids the von Neumann Bottleneck: number of memory accesses proportional to the number of inputs, not the number of computations
- Once finished: 8 reads, 3 writes
- Compare with SIMD: 18 reads, 3 writes

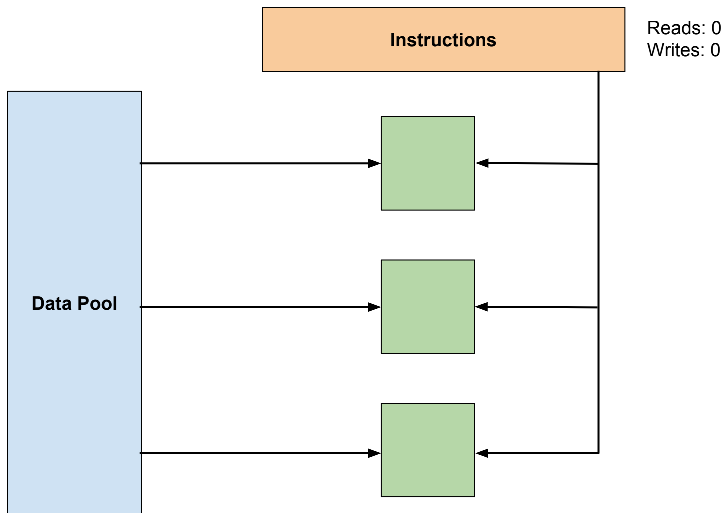
Why is this good?

- Avoids the von Neumann Bottleneck: number of memory accesses proportional to the number of inputs, not the number of computations
- Once finished: 8 reads, 3 writes
- Compare with SIMD: 18 reads, 3 writes

Why is this good?

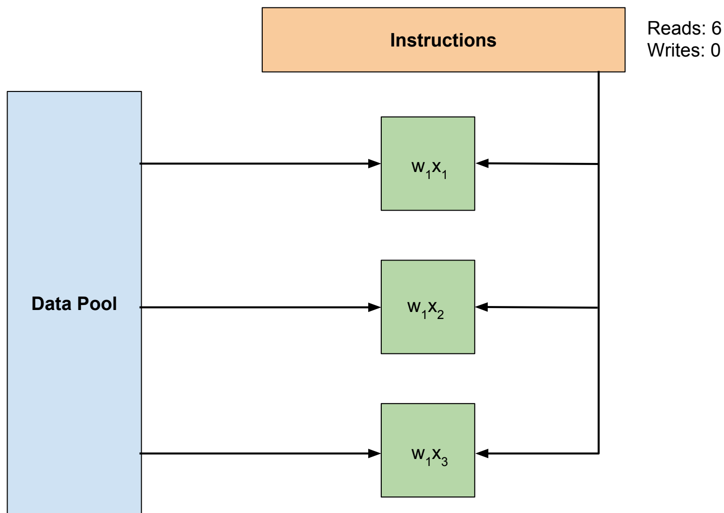
- Avoids the von Neumann Bottleneck: number of memory accesses proportional to the number of inputs, not the number of computations
- Once finished: 8 reads, 3 writes
- Compare with SIMD: 18 reads, 3 writes

1D SIMD Convolution

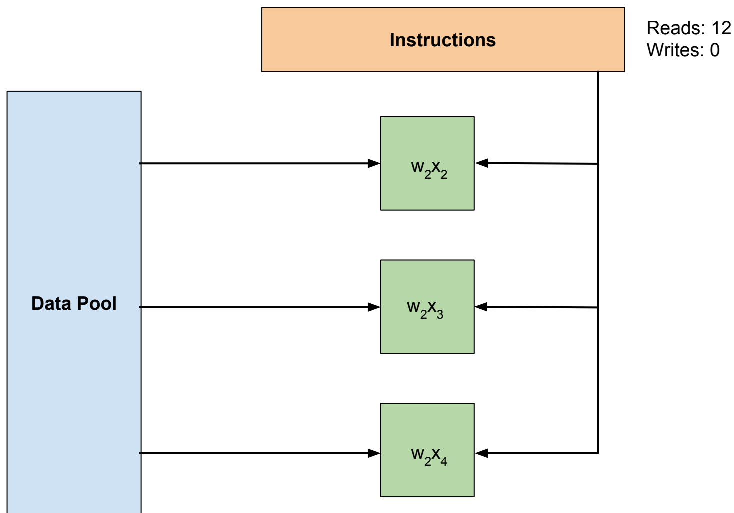


Reads: 0
Writes: 0

1D SIMD Convolution

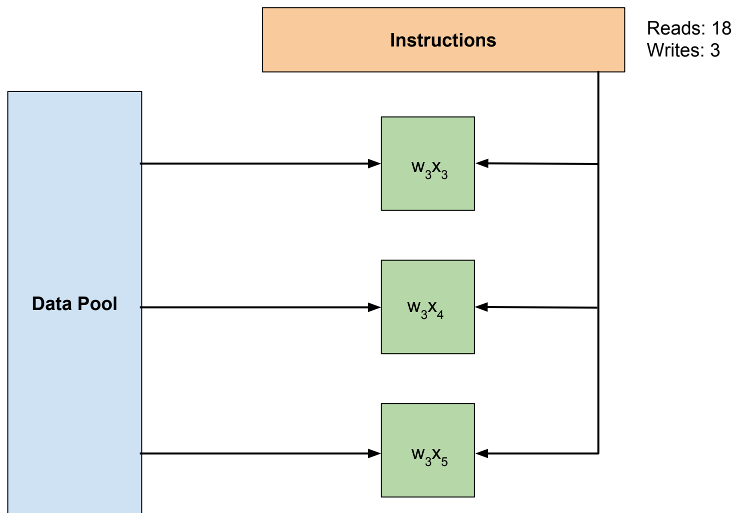


1D SIMD Convolution



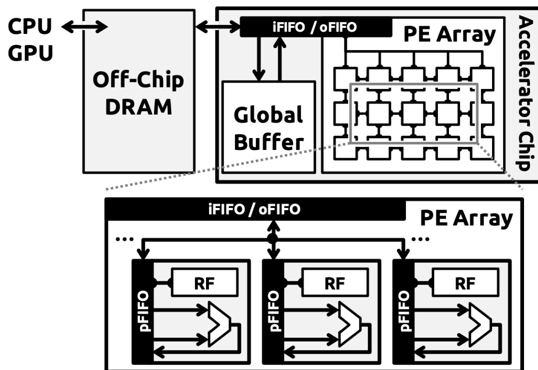
Reads: 12
Writes: 0

1D SIMD Convolution



Reads: 18
Writes: 3

CNN Accelerator Overview

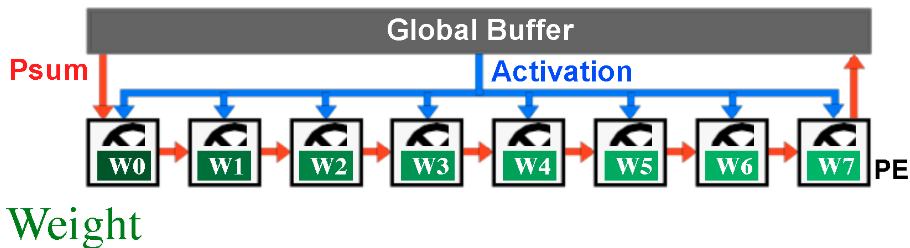


Convolution Layer Design

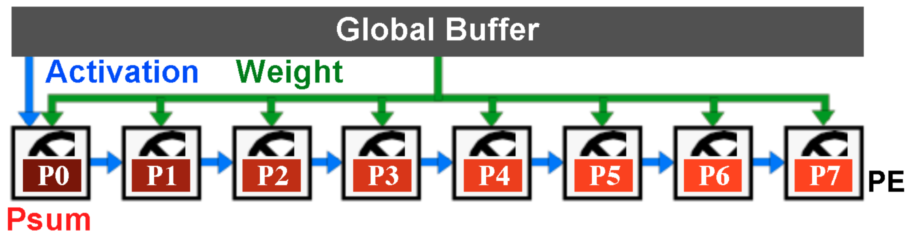
Want to:

- Maximize data reuse
- Minimize memory latency
- Minimize energy use
- Multiple things can be reused - balance reuse opportunities

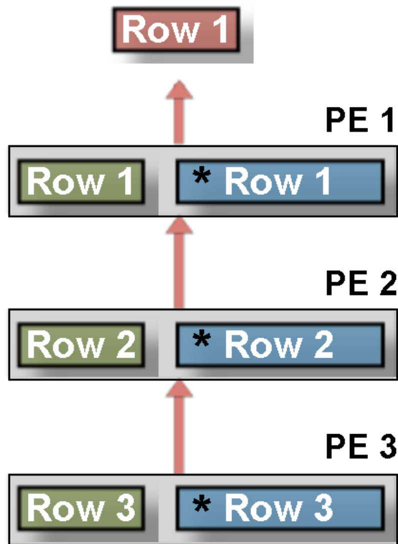
Weight Stationary



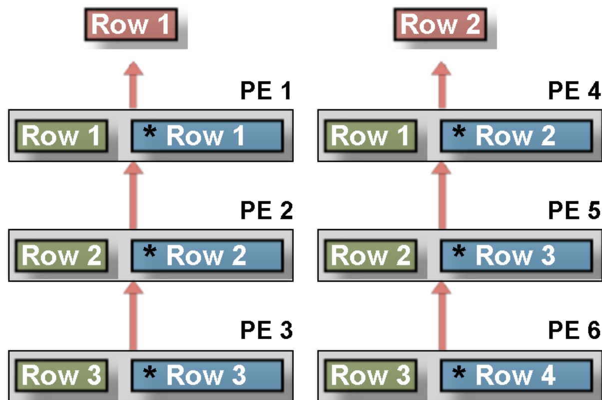
Output Stationary

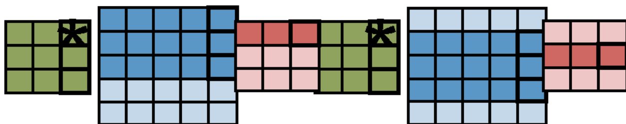


Row Stationary (Eyeriss)

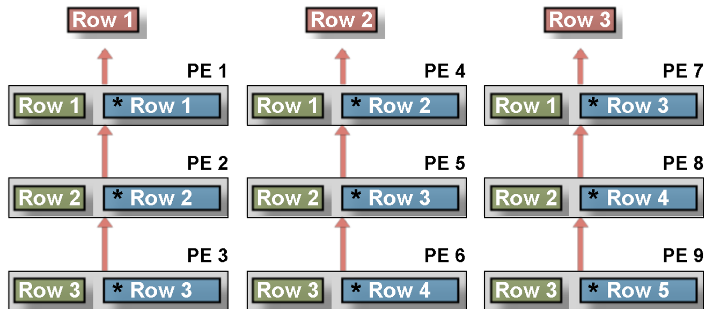


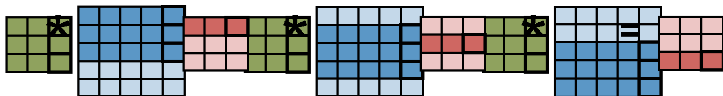
Row Stationary (Eyeriss)





Row Stationary (Eyeriss)





Outline

1. Background: Spatial Architectures
2. Background: Halide
3. DNN Dataflow is Overrated
4. Generating Hardware Designs Using Halide
5. Experiments
6. Conclusion

Motivation

- Images are ubiquitous
- Optimizing image processing code yields high ROI
- Hard to do, creates unreadable code

Motivation

- Images are ubiquitous
- Optimizing image processing code yields high ROI
- Hard to do, creates unreadable code

Motivation

- Images are ubiquitous
- Optimizing image processing code yields high ROI
- Hard to do, creates unreadable code

Motivation

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate blurx array

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

Motivation

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
         a = _mm_loadu_si128((__m128i*)(inPtr-1));
         b = _mm_loadu_si128((__m128i*)(inPtr+1));
         c = _mm_load_si128((__m128i*)(inPtr));
         sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
         avg = _mm_mulhi_epi16(sum, one_third);
         _mm_store_si128(blurxPtr++, avg);
         inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
```

Motivation

3x3 blur as a Halide *algorithm*:

```
Var x, y; Func blurx, blury;
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
```

Halide

- A language/compiler for image processing
- Key idea: decouple algorithm from computation schedule
- Algorithm: what is computed
- Schedule: where/when it's computed
- Key idea: Don't mix algorithm design with scheduling
- Allow compiler to safely optimize and schedule algorithm

Halide

- A language/compiler for image processing
- Key idea: decouple algorithm from computation schedule
- Algorithm: what is computed
- Schedule: where/when it's computed
- Key idea: Don't mix algorithm design with scheduling
- Allow compiler to safely optimize and schedule algorithm

Halide

- A language/compiler for image processing
- Key idea: decouple algorithm from computation schedule
- Algorithm: what is computed
- Schedule: where/when it's computed
- Key idea: Don't mix algorithm design with scheduling
- Allow compiler to safely optimize and schedule algorithm

Halide

- A language/compiler for image processing
- Key idea: decouple algorithm from computation schedule
- Algorithm: what is computed
- Schedule: where/when it's computed
- Key idea: Don't mix algorithm design with scheduling
- Allow compiler to safely optimize and schedule algorithm

Halide

- A language/compiler for image processing
- Key idea: decouple algorithm from computation schedule
- Algorithm: what is computed
- Schedule: where/when it's computed
- Key idea: Don't mix algorithm design with scheduling
- Allow compiler to safely optimize and schedule algorithm

Halide

- A language/compiler for image processing
- Key idea: decouple algorithm from computation schedule
- Algorithm: what is computed
- Schedule: where/when it's computed
- Key idea: Don't mix algorithm design with scheduling
- Allow compiler to safely optimize and schedule algorithm

Outline

1 Background: Spatial Architectures

2 Background: Halide

3 DNN Dataflow is Overrated

4 Generating Hardware Designs Using Halide

5 Experiments

6 Conclusion

Proposal: A 3-Dimensional Design Space

1. Loop optimizations
2. Dataflow
3. Hardware resource allocation

Dimension 1: Loop Optimizations

A Single Convolutional Layer:

- input feature maps of size $X \times Y$
- C channels
- K filters of size $C \times F_X \times F_Y$
- Batch size of B

$$O[b][k][x][y] = \sum_{c=0}^{C-1} \sum_{f_y=0}^{F_Y-1} \sum_{f_x=0}^{F_X-1} I[b][c][x+f_x][y+f_y] \times W[k][c][f_x][f_y]$$

Dimension 1: Loop Optimizations

```
1 for b = 0 : B do                                                      (Batch)
2   for k = 0 : K do                                           (Output channel)
3     for c = 0 : C do                                          (Input channel)
4       for y = 0 : Y do                                          (fmap height)
5         for x = 0 : X do                                         (fmap width)
6           for f_y = -(F_Y - 1)/2 : (F_Y - 1)/2 do              (Filter height)
7             for f_x = -(F_X - 1)/2 : (F_X - 1)/2 do             (Filter width)
8               O[k][x][y] +=
                    I[c][x + f_x][y + f_y] × W[k][c][f_x][f_y]
```

1. **for** $b = 0 : B$ **do** *(Batch)*
2. **for** $k = 0 : K$ **do** *(Output channel)*
3. **for** $c = 0 : C$ **do** *(Input channel)*
4. **for** $y = 0 : Y$ **do** *(fmap height)*
5. **for** $x = 0 : X$ **do** *(fmap width)*
6. **for** $f_y = -\frac{F_Y - 1}{2} : \frac{F_Y - 1}{2}$ **do** *(Filter height)*
7. **for** $f_x = -\frac{F_X - 1}{2} : \frac{F_X - 1}{2}$ **do** *(Filter width)*
8. $O[k][x][y] \mathrel{+}= I[c][x + f_x][y + f_y] \times W[k][c][f_x][f_y]$

Dimension 1: Loop Optimizations

```
1 for b = 0 : B do                                                (Batch)
2   for k = 0 : K do                                              (Output channel)
3     for c = 0 : C do                                            (Input channel)
4       for y = 0 : Y do                                          (fmap height)
5         for x = 0 : X do                                        (fmap width)
6           for f_y = -(F_Y - 1)/2 : (F_Y - 1)/2 do               (Filter height)
7             for f_x = -(F_X - 1)/2 : (F_X - 1)/2 do             (Filter width)
8               O[k][x][y] +=
                    I[c][x + f_x][y + f_y] × W[k][c][f_x][f_y]
```

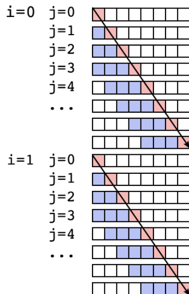
Not how anyone does it!

Dimension 1: Loop Optimizations - Tiling

LOOP TILING: REGISTER BLOCKING

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```



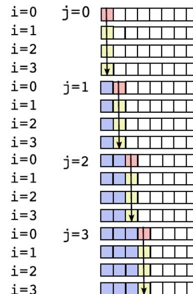
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

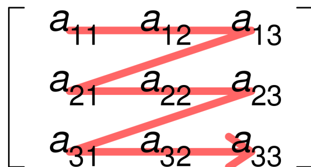
Tiled:

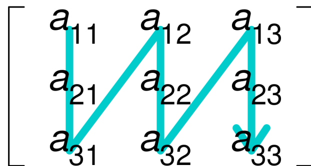
```
for (ii=0; ii<m; ii+=TILE)
  for (j=0; j<n; j++)
    for (i=ii; i<ii+TILE; i++)
      ...=...*b[j];
```



© Colfax International, 2013–2017

Dimension 1: Loop Optimizations - Reordering





Dimension 2: Dataflow

Some heuristics:

- Maximize filter reuse: "weight stationary" (WS)
- Maximize partial sum reuse: "output stationary" (OS)
- No local reuse (NLR)
- Row stationary (RS)

Dimension 2: Dataflow

Some heuristics:

- Maximize filter reuse: “weight stationary” (WS)
- Maximize partial sum reuse: “output stationary” (OS)
- No local reuse (NLR)
- Row stationary (RS)

Dimension 2: Dataflow

Some heuristics:

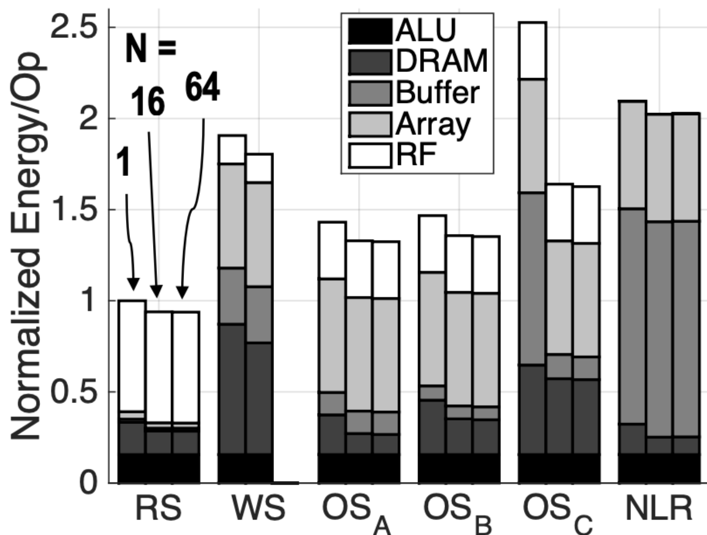
- Maximize filter reuse: “weight stationary” (WS)
- Maximize partial sum reuse: “output stationary” (OS)
- No local reuse (NLR)
- Row stationary (RS)

Dimension 2: Dataflow

Some heuristics:

- Maximize filter reuse: “weight stationary” (WS)
- Maximize partial sum reuse: “output stationary” (OS)
- No local reuse (NLR)
- Row stationary (RS)

Performance of Different Dataflows



Dimension 3: Hardware Resource Allocation

- Dimensions of PE array
- Size of different layers of the memory hierarchy
- Energy cost and latency of accesses

Outline

1 Background: Spatial Architectures

2 Background: Halide

3 DNN Dataflow is Overrated

4 Generating Hardware Designs Using Halide

5 Experiments

6 Conclusion

Halide Primitives

- Two new Halide primitives: Accelerate and Systolic
- One modified primitive: Unroll

Dimensions	Scheduling primitives
Overall scope	`accelerate`
Loop blocking	`tile, reorder`
Dataflow	`unroll, systolic`
Resource allocation	`in, compute_at`

Halide Pseudocode

```
1 // Define (minX, extentX, minY, extentY, minK, extentK)
2 RDom r(-2, 5, -2, 5, 0, 3);
3
4 output(x, y, k) += input(x + r.x, y + r.y, r.z)
5                  * w(r.x + 2, r.y + 2. r.z, k);
```

Halide Pseudocode

```
1 d = output.in()
2
3 output.tile(x, y, xo, yo, xi, yi, 28, 28)
4         .reorder(xi, yi, xo, yo)
5         .accelerate({input, w});
6
7 input.in().compute_at(output, xo);
8 w.in().compute_at(output, xo);
9 output.compute_at(d, xo);
```

Halide Pseudocode

```
1  for (k, 0, 64)
2
3    for (yo, 0, 4)
4      for (xo, 0, 4)
5        // Allocate local buffer for output.
6        alloc obuf[28, 28, 1]
7
8        // Allocate local buffer for input.
9        alloc ibuf[28 + 5 - 1, 28 + 5 - 1, 3]
10       // Copy input to buffer.
11       ibuf[...] = input[...]
12
13       // Allocate local buffer for w.
14       alloc wbuf[5, 5, 3, 1]
15       // Copy w to buffer.
16       wbuf[...] = w[...]
17
18       for (yi, 0, 28)
19         for (xi, 0, 28)
20
21           for (r.z, 0, 3)
22             for (r.y, -2, 5)
23               for (r.x, -2, 5)
24                 obuf(xi, yi, 0) +=
25                     ibuf(xi + r.x, yi + r.y, r.z)
26                     * wbuf(r.x + 2, r.y + 2, r.z, 0)
27
28       // Copy buffer to output.
29       output[...] = obuf[...]
```

New Primitive: Accelerate

- Accelerate: "Defines scope and interface to the rest of the system"
- Marks for transformation to some dataflow IR
- Dataflow IR is some kind of special C++ program
- IR compiled into Verilog using High-Level Synthesis (HLS) (probably involves additional optimizations)
- Nothing about latency or throughput

New Primitive: Accelerate

- Accelerate: "Defines scope and interface to the rest of the system"
- Marks for transformation to some dataflow IR
- Dataflow IR is some kind of special C++ program
- IR compiled into Verilog using High-Level Synthesis (HLS) (probably involves additional optimizations)
- Nothing about latency or throughput

New Primitive: Accelerate

- Accelerate: “Defines scope and interface to the rest of the system”
- Marks for transformation to some dataflow IR
- Dataflow IR is some kind of special C++ program
- IR compiled into Verilog using High-Level Synthesis (HLS) (probably involves additional optimizations)
- Nothing about latency or throughput

New Primitive: Accelerate

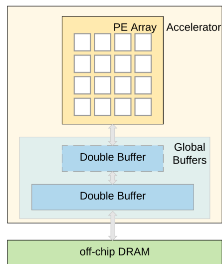
- Accelerate: "Defines scope and interface to the rest of the system"
- Marks for transformation to some dataflow IR
- Dataflow IR is some kind of special C++ program
- IR compiled into Verilog using High-Level Synthesis (HLS) (probably involves additional optimizations)
- Nothing about latency or throughput

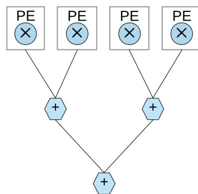
New Primitive: Accelerate

- Accelerate: “Defines scope and interface to the rest of the system”
- Marks for transformation to some dataflow IR
- Dataflow IR is some kind of special C++ program
- IR compiled into Verilog using High-Level Synthesis (HLS) (probably involves additional optimizations)
- Nothing about latency or throughput

Custom Primitive: Systolic

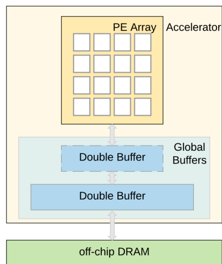
- Systolic flag: PEs communicate during loop unrolling
- No systolic flag: unrolled loop performs tree reduction

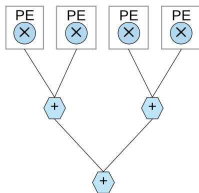




Custom Primitive: Systolic

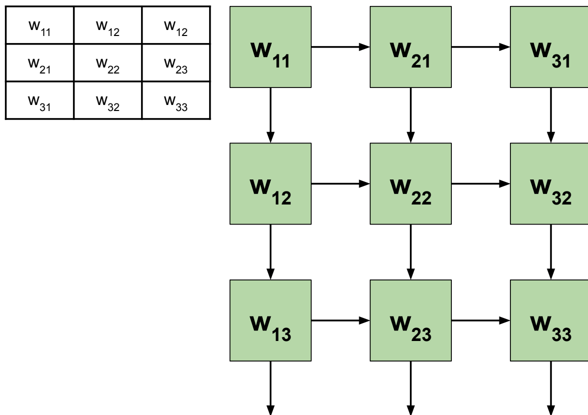
- Systolic flag: PEs communicate during loop unrolling
- No systolic flag: unrolled loop performs tree reduction





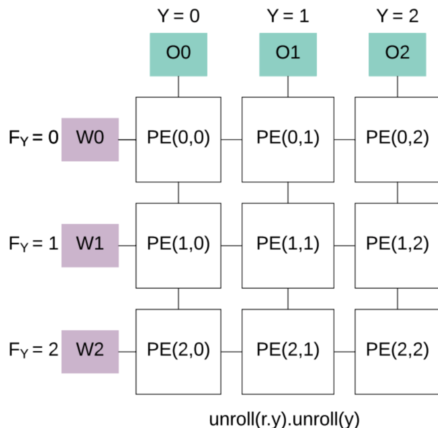
Overriden Primitive: Unroll

- Basic idea: loop unrollings correspond to different spatial architectures
- Example: unroll weights vertically and horizontally is "weight stationary" $(F_Y|F_X)$



Overridden Primitive: Unroll

- $F_Y|Y$ = Eyeriss
- $C|K$ = Google TPU



Recap

- Provide modified Halide with functional programming description of conv layer
- Modified Halide both generates a spatial architecture and schedules execution

Outline

1. Background: Spatial Architectures
2. Background: Halide
3. DNN Dataflow is Overrated
4. Generating Hardware Designs Using Halide
5. Experiments
6. Conclusion

Analysis Framework

- Use parameters from AlexNet, MobileNet, and GoogleNet
- Use CACTI to analyze performance and energy
- RTL model
- Model with 28nm technology (Eyeriss uses 65nm, TPU 28nm)
- Model memory usage energy with weighted linear model
- Use their tool to model Eyeriss, TPU, etc.

Analysis Framework

- Use parameters from AlexNet, MobileNet, and GoogleNet
- Use CACTI to analyze performance and energy
- RTL model
- Model with 28nm technology (Eyeriss uses 65nm, TPU 28nm)
- Model memory usage energy with weighted linear model
- Use their tool to model Eyeriss, TPU, etc.

Analysis Framework

- Use parameters from AlexNet, MobileNet, and GoogleNet
- Use CACTI to analyze performance and energy
- RTL model
- Model with 28nm technology (Eyeriss uses 65nm, TPU 28nm)
- Model memory usage energy with weighted linear model
- Use their tool to model Eyeriss, TPU, etc.

Analysis Framework

- Use parameters from AlexNet, MobileNet, and GoogleNet
- Use CACTI to analyze performance and energy
- RTL model
- Model with 28nm technology (Eyeriss uses 65nm, TPU 28nm)
- Model memory usage energy with weighted linear model
- Use their tool to model Eyeriss, TPU, etc.

Analysis Framework

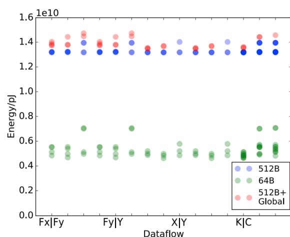
- Use parameters from AlexNet, MobileNet, and GoogleNet
- Use CACTI to analyze performance and energy
- RTL model
- Model with 28nm technology (Eyeriss uses 65nm, TPU 28nm)
- Model memory usage energy with weighted linear model
- Use their tool to model Eyeriss, TPU, etc.

Analysis Framework

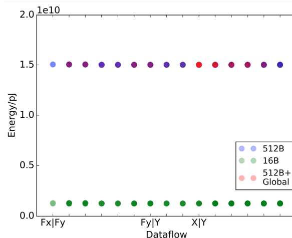
- Use parameters from AlexNet, MobileNet, and GoogleNet
- Use CACTI to analyze performance and energy
- RTL model
- Model with 28nm technology (Eyeriss uses 65nm, TPU 28nm)
- Model memory usage energy with weighted linear model
- Use their tool to model Eyeriss, TPU, etc.

Results

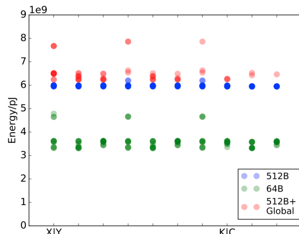
Loop unrollings/dataflows are horizontal. 1 color = 1 register file size. Same color + same dataflow means different "replications" (parallelized loops)



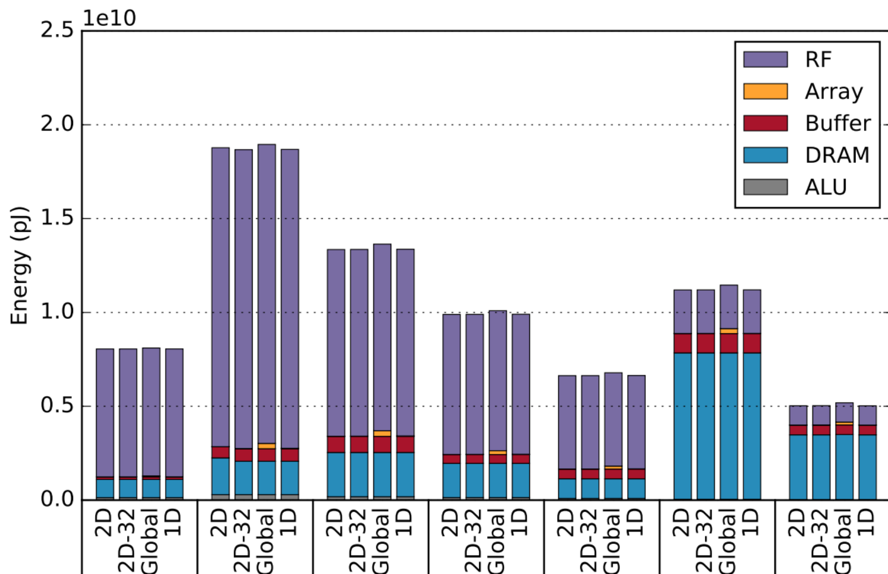
(a) AlexNet



(b) MobileNet Depthwise



With good loop optimizations, reuse is high (high RF energy use). 2d = best blue points, Global = best red points from previous fig.



Outline

1 Background: Spatial Architectures
2 Background: Halide
3 DNN Dataflow is Overrated
4 Generating Hardware Designs Using Halide
5 Experiments
6 Conclusion

Issues

- They don't successfully show that "dataflow choice is overrated"
- What about latency and throughput? Dataflow choice could be super important with respect to those
- Are they doing a good job modeling their competitors?
- Can Halide can actually model the space of "all possible systolic accelerators"?
- Uses very different nm process from Eyeriss
- Bad figures

Issues

- They don't successfully show that "dataflow choice is overrated"
- What about latency and throughput? Dataflow choice could be super important with respect to those
- Are they doing a good job modeling their competitors?
- Can Halide can actually model the space of "all possible systolic accelerators"?
- Uses very different nm process from Eyeriss
- Bad figures

Issues

- They don't successfully show that "dataflow choice is overrated"
- What about latency and throughput? Dataflow choice could be super important with respect to those
- Are they doing a good job modeling their competitors?
- Can Halide can actually model the space of "all possible systolic accelerators"?
- Uses very different nm process from Eyeriss
- Bad figures

Issues

- They don't successfully show that "dataflow choice is overrated"
- What about latency and throughput? Dataflow choice could be super important with respect to those
- Are they doing a good job modeling their competitors?
- Can Halide can actually model the space of "all possible systolic accelerators"?
- Uses very different nm process from Eyeriss
- Bad figures

Issues

- They don't successfully show that "dataflow choice is overrated"
- What about latency and throughput? Dataflow choice could be super important with respect to those
- Are they doing a good job modeling their competitors?
- Can Halide can actually model the space of "all possible systolic accelerators"?
- Uses very different nm process from Eyeriss
- Bad figures

Issues

- They don't successfully show that "dataflow choice is overrated"
- What about latency and throughput? Dataflow choice could be super important with respect to those
- Are they doing a good job modeling their competitors?
- Can Halide can actually model the space of "all possible systolic accelerators"?
- Uses very different nm process from Eyeriss
- Bad figures

Lessons Learned

- Decoupling algorithm from processing details can be valuable for accelerating ML
- Idea of using Halide to generate designs is very interesting
- Good demonstration that loop optimizations are important

Lessons Learned

- Decoupling algorithm from processing details can be valuable for accelerating ML
- Idea of using Halide to generate designs is very interesting
- Good demonstration that loop optimizations are important

Lessons Learned

- Decoupling algorithm from processing details can be valuable for accelerating ML
- Idea of using Halide to generate designs is very interesting
- Good demonstration that loop optimizations are important