Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques

Sven Verdoolaege Manjunath Kudlur Rob Schreiber Harinath Kamepalli

Cerebras Systems

January 22, 2020

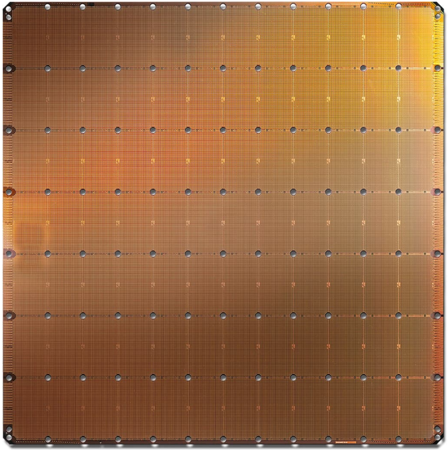
Outline

Outline

Cerebras CS-1

Largest chip ever built

- $46{,}225\ mm^2$ silicon
- 1.2 trillion transistors
- 400,000 AI optimized cores
- 18 Gigabytes of On-chip Memory
- 9 PByte/s memory bandwidth
- 100 Pbit/s fabric bandwidth
- TSMC 16nm process



Interesting Features

- Dataflow scheduling in hardware
  - Triggered by data
  - Filters out sparse zero data
  - Skips unnecessary processing

Sparse Tensor Communication

Tensor

0	42	0
0	0	0
57	0	13

Dense Communication

$\xleftarrow{\text{send}}$

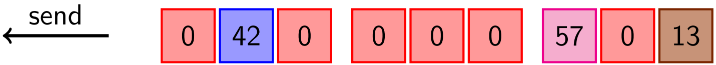
0	42	0	0	0	0	57	0	13

Sparse Tensor Communication

Tensor

0	42	0
0	0	0
57	0	13

Dense Communication



Sparse Communication

- break up tensor into chunks (e.g., rows)
- only send
  - non-zero entry + position in chunk
  - end-of-chunk

Interesting Features

- Dataflow scheduling in hardware
  - Triggered by data
  - Filters out sparse zero data
  - Skips unnecessary processing

Interesting Features

- Dataflow scheduling in hardware
  - Triggered by data
  - Filters out sparse zero data
  - Skips unnecessary processing
- Powerful SIMD Engine
  - Performs *some number* of operations per cycle
  - Mimics normalized loop nest of depth at most four
    $\Rightarrow$ removes overhead of software managed loops

SIMD Instructions

Loop code:

```
handle(uint16_t index, half data) {
  for (int c3 = 0; c3 <= 4; c3 += 1)
    for (int c4 = 0; c4 <= 4; c4 += 1)
      dx_local[2 * dy_index_0 + c3][2 * index + c4] +=
        (data) * (W_local[0][c3][c4]);
}
```

SIMD Instructions

Loop code:

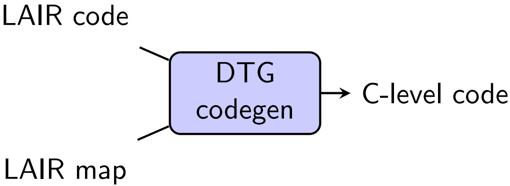
```
handle(uint16_t index, half data) {
  for (int c3 = 0; c3 <= 4; c3 += 1)
    for (int c4 = 0; c4 <= 4; c4 += 1)
      dx_local[2 * dy_index_0 + c3][2 * index + c4] +=
        (data) * (W_local[0][c3][c4]);
}
```

SIMD instruction:

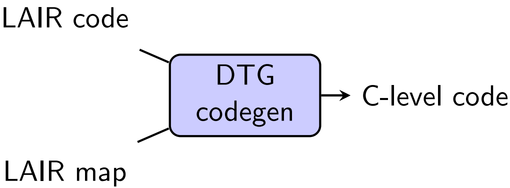
```
handle(uint16_t index, half data) {
  set_base_address(dx, &dx_local[2 * dy_index_0][2 * index]);
  invoke_simd(fmach, dx, W, data, index);
}

void main() {
  configure(/* 5,5; W_local: i,j -> 0,i,j; dx_local: i,j -> i,j */);
  set_base_address(W, &W_local[0][0][0]);
}
```

Outline

Code Generation Overview



Code Generation Overview



LAIR

⇒ DSL written by hand or extracted from `TensorFlow` (Abadi et al. 2016)

LAIR Example

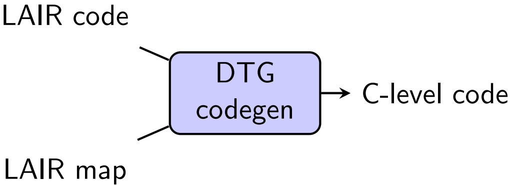
```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

`lair` node

- defines one or more output tensors in terms of input tensors
- each statement has zero-based rectangular set of instances
- LAIR is single assignment (at tensor level)
- all accesses are affine (not piecewise, not quasi-affine)
- each tensor in a statement is accessed through single index expression

Other nodes combine and/or specialize `lair` nodes
$\Rightarrow$ e.g., $M = 32$ and $N = 16$

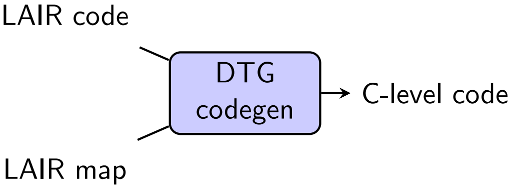
Code Generation Overview



LAIR

$\Rightarrow$ DSL written by hand or extracted from `TensorFlow` (Abadi et al. 2016)

Code Generation Overview



LAIR

$\Rightarrow$ DSL written by hand or extracted from `TensorFlow` (Abadi et al. 2016)

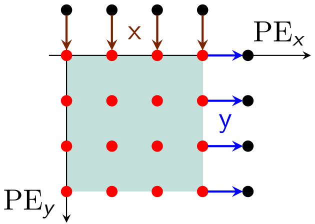
LAIR map contains information in `isl` (V. 2010) notation about

- the size of the target rectangle of PEs
- how input and output tensors are communicated
- where computations are performed

LAIR Map Example

```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

Mapping of 32×16 matrix vector multiplication to 4×4 PEs.




```
size: { PE[4, 4] }
compute_map: { ff[i, j] -> PE[j//4, i//8] }
iport_map: { x[i=0:15] -> [PE[i//4, -1] -> index[i%4]] }
oport_map: { y[i=0:31] -> [PE[4, i//8] -> index[i%8]] }
```

Task Graph Construction

Code generation consists of

- Parse LAIR and LAIR map
- Construct task graph
- Detect SIMD opportunities
- Write out code

Task Graph Construction

Code generation consists of

- Parse LAIR and LAIR map
- Construct task graph
- Detect SIMD opportunities
- Write out code

Task graph construction: split LAIR specification into

- communication tasks
- computation tasks
  Two types:
  - react to incoming tensor element
  - read in entire tensor or operate on local memory

Outline

SIMD Code Generation

$\Rightarrow$ detect sets of computation instances that can be performed by SIMD instructions

$\Rightarrow$ determine

- supported instruction
- "fixed" instance set sizes
- accesses of the form

$$\textit{offset} + \textit{linear in iterators}$$

"fixed" sizes: may depend on PE, but not on tensor element
Otherwise, configuration needs to be performed before each invocation

SIMD Code Generation

$\Rightarrow$ detect sets of computation instances that can be performed by SIMD instructions

$\Rightarrow$ determine

- supported instruction
- "fixed" instance set sizes
- accesses of the form

$$\textit{offset} + \textit{linear in iterators}$$

"fixed" sizes: may depend on PE, but not on tensor element
Otherwise, configuration needs to be performed before each invocation

SIMD Instructions

Loop code:

```
handle(uint16_t index, half data) {
  for (int c3 = 0; c3 <= 4; c3 += 1)
    for (int c4 = 0; c4 <= 4; c4 += 1)
      dx_local[2 * dy_index_0 + c3][2 * index + c4] +=
        (data) * (W_local[0][c3][c4]);
}
```

SIMD instruction:

```
handle(uint16_t index, half data) {
  set_base_address(dx, &dx_local[2 * dy_index_0][2 * index]);
  invoke_simd(fmach, dx, W, data, index);
}

void main() {
  configure(/* 5,5; W_local: i,j -> 0,i,j; dx_local: i,j -> i,j */);
  set_base_address(W, &W_local[0][0][0]);
}
```

Challenge

Recall:

`lair` node guarantees:

- each statement has zero-based rectangular set of instances
- all accesses are affine (not piecewise, not quasi-affine)

SIMD detection requirements:

- "fixed" instance set sizes
- accesses of the form

$$\textit{offset} + \textit{linear in iterators}$$

Trivial?

Trivial Example

```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

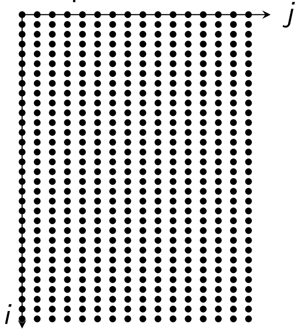
```
compute_map: { ff[i, j] -> PE[j//4, i//8] }
```

Trivial Example

```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

```
compute_map: { ff[i, j] -> PE[j//4, i//8] }
```

Computation instances:

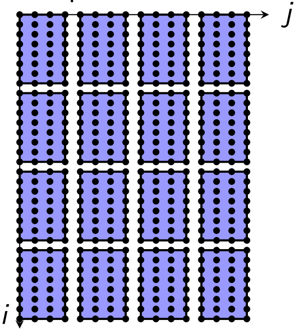


Trivial Example

```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

```
compute_map: { ff[i, j] -> PE[j//4, i//8] }
```

Computation instances:

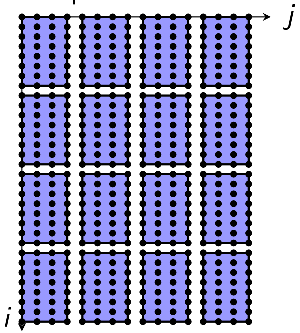


- Mapping to PEs

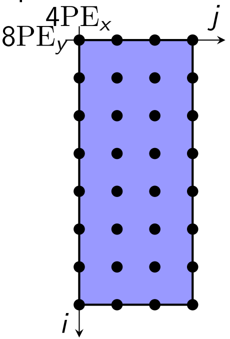
Trivial Example

```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}

compute_map: { ff[i, j] -> PE[j//4, i//8] }
```

Computation instances:



Computation instances on PE:



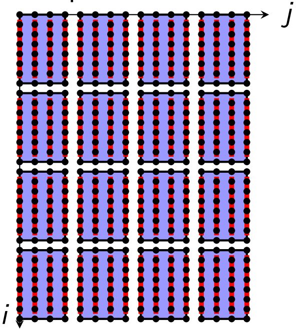
- Mapping to PEs

Trivial Example

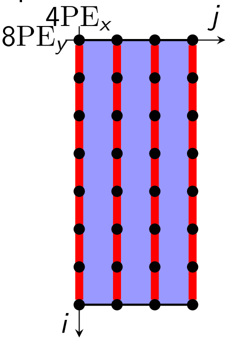
```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

```
compute_map: { ff[i, j] -> PE[j//4, i//8] }
```

Computation instances:



Computation instances on PE:



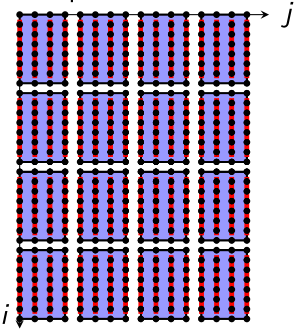
- Mapping to PEs
- Arrival of x-value

Trivial Example

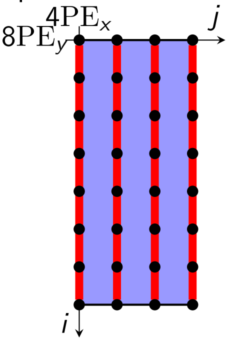
```
lair matvec<T=float16>(M, N): T W[M][N], T x[N] -> T y[M] {
    all (i, j) in (M, N)
        y[i] += W[i][j] * x[j]
}
```

```
compute_map: { ff[i, j] -> PE[j//4, i//8] }
```

Computation instances:



Computation instances on PE:



- Mapping to PEs
- Arrival of x-value

$\Rightarrow$ Size: $[8, 1]$

$\Rightarrow$ Access to y: $\mathrm{y}[8\mathrm{PE}_y + i']$

(local coordinates: i', j')

Size Computation

Input: S: set of instances executed on a PE on arrival of a tensor element

Size Computation

Input: S: set of instances executed on a PE on arrival of a tensor element

- Compute element-wise minimum and maximum of S
- Construct $\{\, \mathbf{x} : \min \leq \mathbf{x} \leq \max \,\}$
- Check equal to S
  $\Rightarrow$ S is a dense box
- Size: $\max - \min + 1$
- Check size does not depend on "index"

Convolution

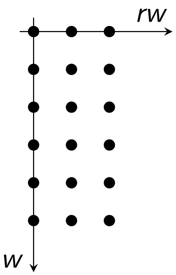
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
compute_map: { C[w, rw] -> PE[0, 0] }
```

Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



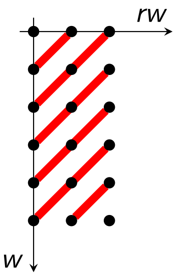
Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:

- Arrival of x-value

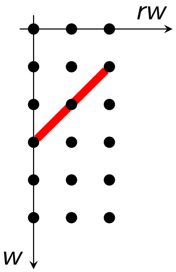


Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



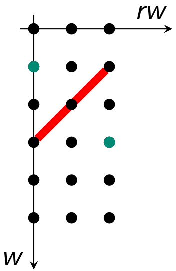
- Arrival of x-value

Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



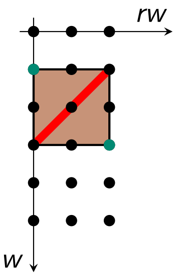
- Arrival of x-value
- Compute minimum and maximum

Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



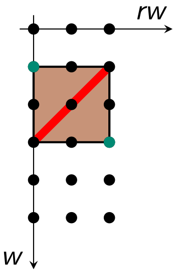
- Arrival of x-value
- Compute minimum and maximum
- Construct $\{\,\mathbf{x} : \min \le \mathbf{x} \le \max\,\}$

Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



- Arrival of x-value
- Compute minimum and maximum
- Construct $\{\,\mathbf{x} : \min \leq \mathbf{x} \leq \max\,\}$

$\Rightarrow$ not a dense box

Variable Compression

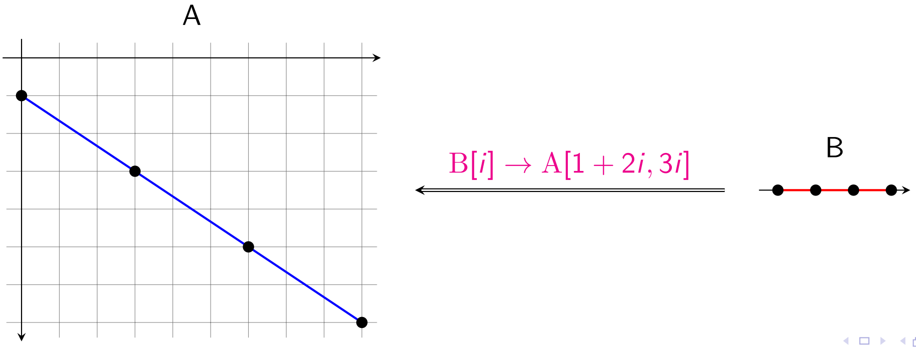
Variable compression (Meister 2004):

- pick affine transformation (with inverse) mapping
- lower-dimensional set to
- full-dimensional set (in lower-dimensional space)

Variable Compression

Variable compression (Meister 2004):

- pick affine transformation (with inverse) mapping
- lower-dimensional set to
- full-dimensional set (in lower-dimensional space)



Size Computation

Input: S: set of instances executed on a PE on arrival of a tensor element

- Compute element-wise minimum and maximum of S
- Construct $\{\, \mathbf{x} : \min \le \mathbf{x} \le \max \,\}$
- Check equal to S
  $\Rightarrow$ S is a dense box
- Size: $\max - \min + 1$
- Check size does not depend on "index"

Size Computation

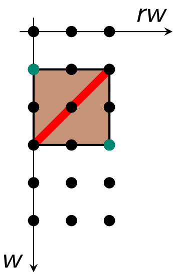
Input: S: set of instances executed on a PE on arrival of a tensor element

- Apply variable compression to S to obtain S'
- Compute element-wise minimum and maximum of S'
- Construct $\{\, \mathbf{x} : \min \le \mathbf{x} \le \max \,\}$
- Check equal to S'
  $\Rightarrow$ S' is a dense box
- Size: $\max - \min + 1$
- Check size does not depend on "index"

Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



- Arrival of x-value
- Compute minimum and maximum
- Construct $\{\, \mathbf{x} : \min \le \mathbf{x} \le \max \,\}$

$\Rightarrow$ not a dense box

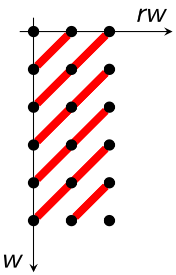
Convolution

```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:

- Arrival of x-value

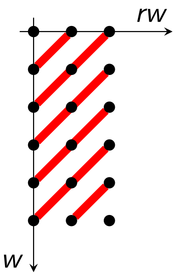


Convolution

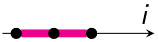
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



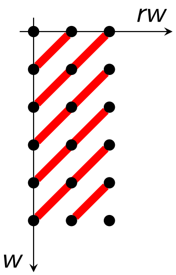
- Arrival of x-value
- Compress

Convolution

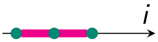
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



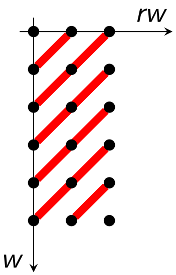
- Arrival of x-value
- Compress
- Compute minimum and maximum

Convolution

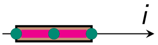
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



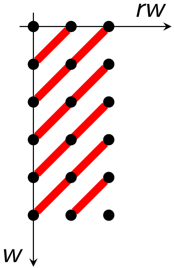
- Arrival of x-value
- Compress
- Compute minimum and maximum
- Construct $\{\, \mathbf{x} : \min \leq \mathbf{x} \leq \max \,\}$

Convolution

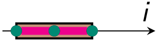
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



- Arrival of x-value
- Compress
- Compute minimum and maximum
- Construct $\{\, \mathbf{x} : \min \leq \mathbf{x} \leq \max \,\}$

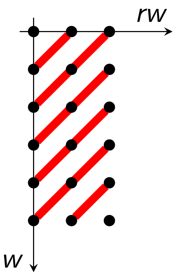
$\Rightarrow$ a dense box

Convolution

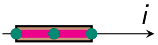
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



- Arrival of x-value
- Compress
- Compute minimum and maximum
- Construct $\{\, \mathbf{x} : \min \le \mathbf{x} \le \max \,\}$

$\Rightarrow$ a dense box

- Size: $\max - \min + 1$

$\Rightarrow$ [1], [2] or [3] depending on "index"

Fixed Size Box Hull Approximation

Fixed size box hull approximation:

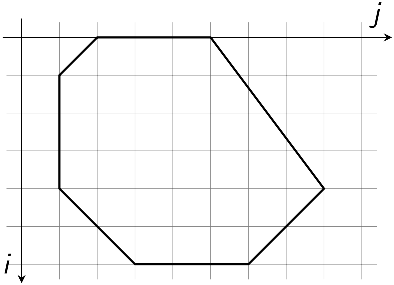
- Result: box containing the input set with
  - variable offset (in particular, may involve "index")
  - fixed size (in particular, does *not* involve "index")
- Approach: look for suitable constraints in representation of input set
- May fail to produce a result

(also used by `PPCG` (V. et al. 2013) to obtain mapping to shared memory)

Fixed Size Box Hull Approximation

Fixed size box hull approximation:

- Result: box containing the input set with
  - variable offset (in particular, may involve "index")
  - fixed size (in particular, does *not* involve "index")
- Approach: look for suitable constraints in representation of input set
- May fail to produce a result

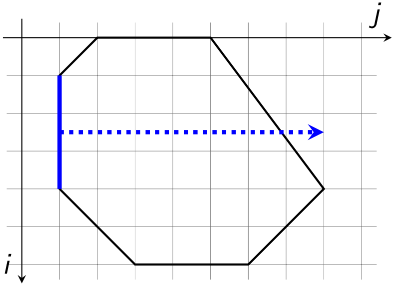


(also used by `PPCG` (V. et al. 2013) to obtain mapping to shared memory)

Fixed Size Box Hull Approximation

Fixed size box hull approximation:

- Result: box containing the input set with
  - variable offset (in particular, may involve "index")
  - fixed size (in particular, does *not* involve "index")
- Approach: look for suitable constraints in representation of input set
- May fail to produce a result

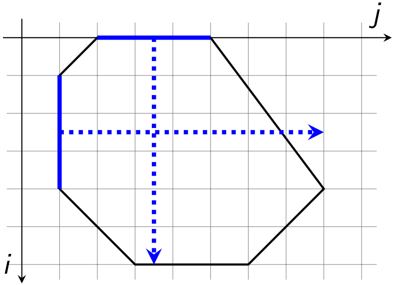


(also used by PPCG (V. et al. 2013) to obtain mapping to shared memory)

Fixed Size Box Hull Approximation

Fixed size box hull approximation:

- Result: box containing the input set with
  - variable offset (in particular, may involve "index")
  - fixed size (in particular, does *not* involve "index")
- Approach: look for suitable constraints in representation of input set
- May fail to produce a result

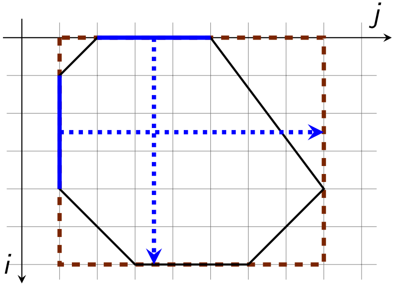


(also used by `PPCG` (V. et al. 2013) to obtain mapping to shared memory)

Fixed Size Box Hull Approximation

Fixed size box hull approximation:

- Result: box containing the input set with
  - variable offset (in particular, may involve "index")
  - fixed size (in particular, does *not* involve "index")
- Approach: look for suitable constraints in representation of input set
- May fail to produce a result



(also used by PPCG (V. et al. 2013) to obtain mapping to shared memory)

Size Computation

Input: S: set of instances executed on a PE on arrival of a tensor element

- Apply variable compression to S to obtain S'

- Compute element-wise minimum and maximum of S'
- Construct $\{ \mathbf{x} : \min \leq \mathbf{x} \leq \max \}$
- Check equal to S'
  $\Rightarrow$ S' is a dense box
- Size: $\max - \min + 1$
- Check size does not depend on "index"

Size Computation

Input: S: set of instances executed on a PE on arrival of a tensor element

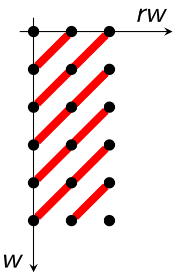
- Apply variable compression to S to obtain S'
- Try and compute fixed size box hull of S'
  If successful and extra instances write to disjoint locations, then use box size. Stop.
- Compute element-wise minimum and maximum of S'
- Construct $\{\, \mathbf{x} : \min \le \mathbf{x} \le \max \,\}$
- Check equal to S'
  $\Rightarrow S'$ is a dense box
- Size: $\max - \min + 1$
- Check size does not depend on "index"

Convolution

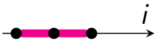
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



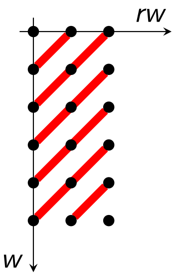
- Arrival of x-value
- Compress

Convolution

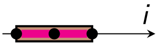
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



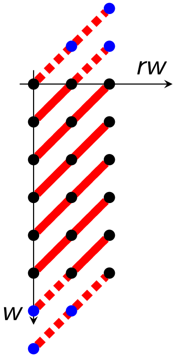
- Arrival of x-value
- Compress
- Try and compute box hull

Convolution

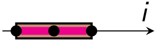
```
lair C() : float16 x[8], float16 W[3] -> float16 y[6] {
  all (w, rw) in (8 - 3 + 1, 3)
    y[w] += x[w + rw] * W[rw]
}
```

```
compute_map: { C[w, rw] -> PE[0, 0] }
```

Computation instances:



Compressed instances:



- Arrival of x-value
- Compress
- Try and compute box hull
- Extra instances write to disjoint locations

Outline

Conclusion

- achieving good performance on Cerebras CS-1 requires generation of SIMD instructions
- heuristics based approach can detect opportunities in many cases, using
  - variable compression
  - fixed size box hull approximation
- effective use of polyhedral compilation techniques (other than affine scheduling)

References I

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (Nov. 2016). "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, pp. 265–283.

Meister, Benoît (Dec. 2004). "Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization". PhD thesis. Université Louis Pasteur.

V., Sven (2010). "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

References II

V., Sven, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor (2013). "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4, p. 54. DOI: 10.1145/2400682.2400713.