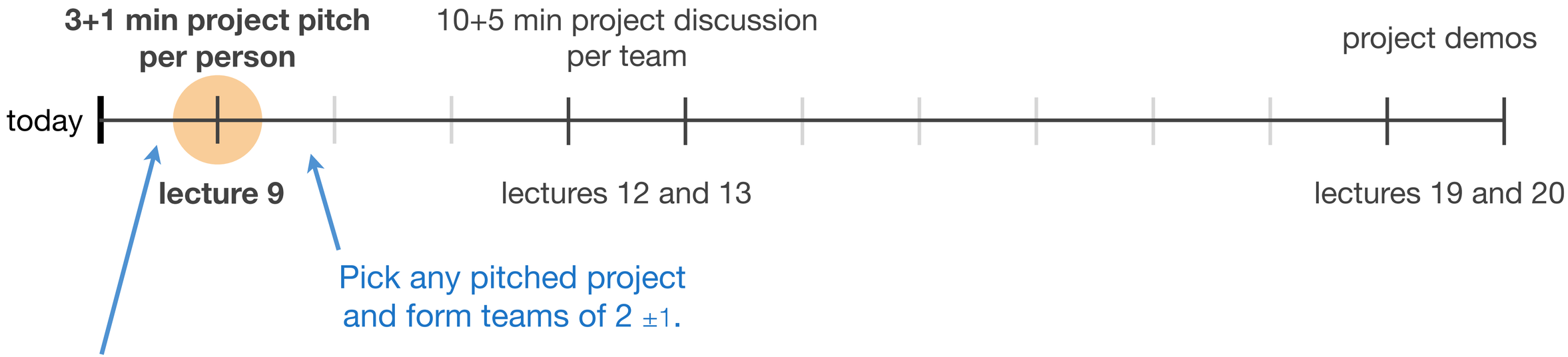
Lecture 8 — Sparse Iteration Model II

Stanford CS343D (Winter 2024)
Fred Kjolstad

Course Project



Each person contributes one pitch slide to a google slide deck. These pitches are not binding.

Overview of topics

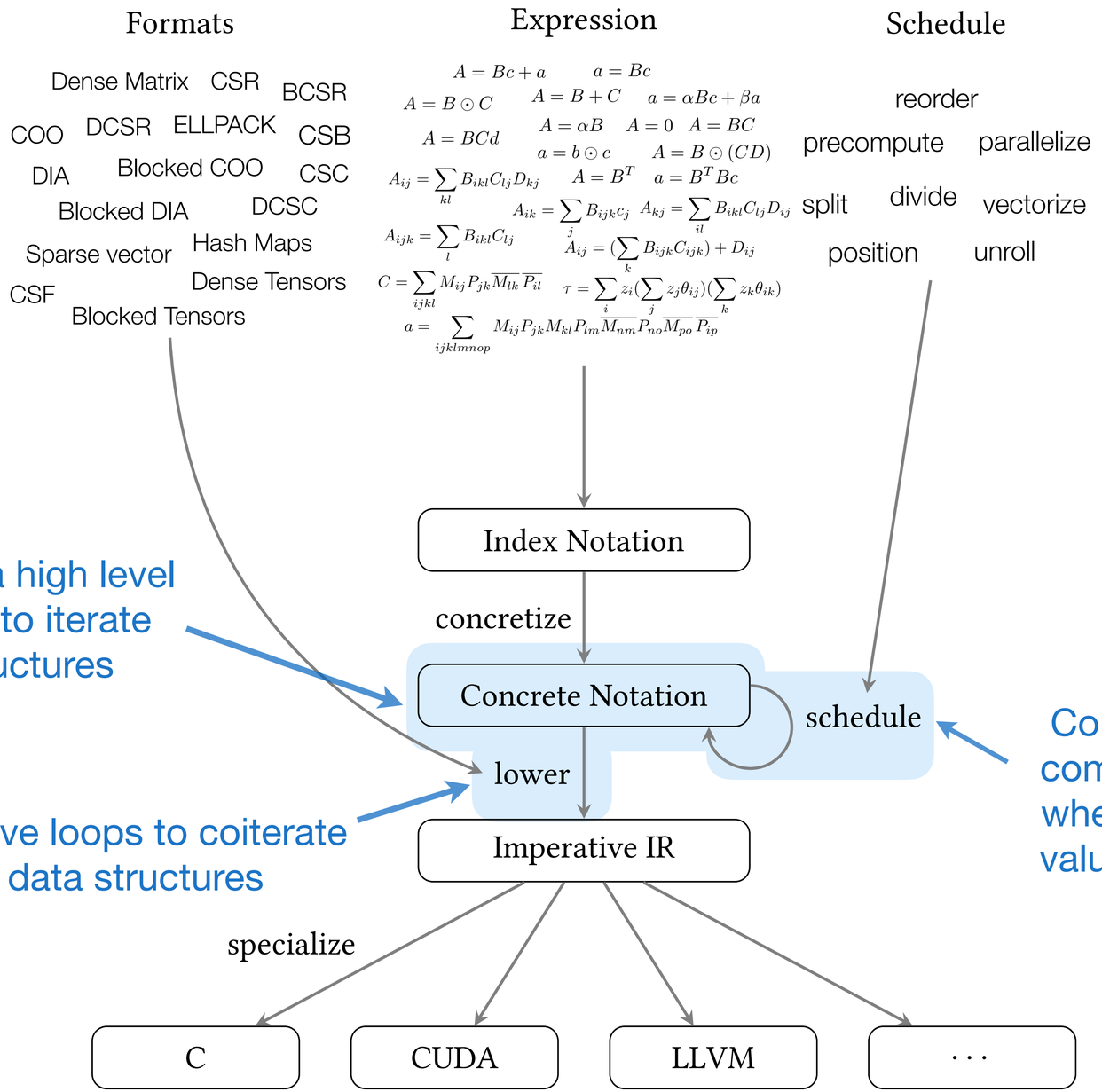
Lecture 7

- Data representation
- Iteration spaces
- Iteration graph IR
- Iteration lattices to represent coiteration

Lecture 8

- Concrete index notation IR
- Code generation algorithm
- Derived iteration spaces
- Optimizing transformations

Overview of compilation stages



Express computation at a high level before generating code to iterate over specific data structures

Generate imperative loops to coiterate over relevant data structures

Control order of computation and where temporary values are stored

Concrete index notation specifies order of computations and location of intermediate values

Index Notation		Concrete Index Notation
$A_{ij} = B_{ij} + C_{ij}$	→	$\forall_i \forall_j \; A_{ij} = B_{ij} + C_{ij}$
$\alpha = \sum_i b_i \, c_i$	→	$\forall_i \; \alpha \mathrel{+}= b_i \, c_i$
$a_i = \sum_j B_{ij} \, c_j$	→	$\forall_i \; a_i = t \; \textbf{where} \; \forall_j \; t \mathrel{+}= B_{ij} \, c_j$

Concrete index notation grammars

Assignment statement

$$A_{i\ldots} = \text{expr}$$

Forall statement

$$\forall_i \ \text{stmt}$$

Where statement

$$\text{stmt}_c \ \textbf{where} \ \text{stmt}_p$$

Environment

index index “$\xrightarrow{\text{“collapse”}}$” index

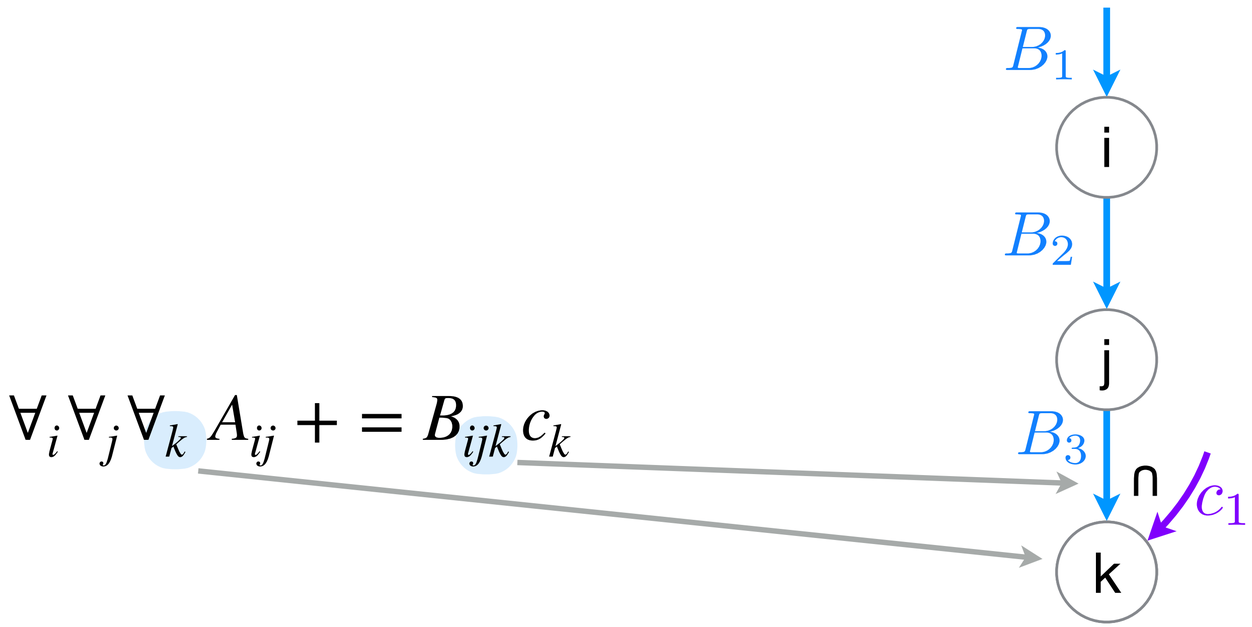
index “$\xrightarrow{\text{“split(” d “,” s “)”}}$” index index

“bound(” index “,” b “)”

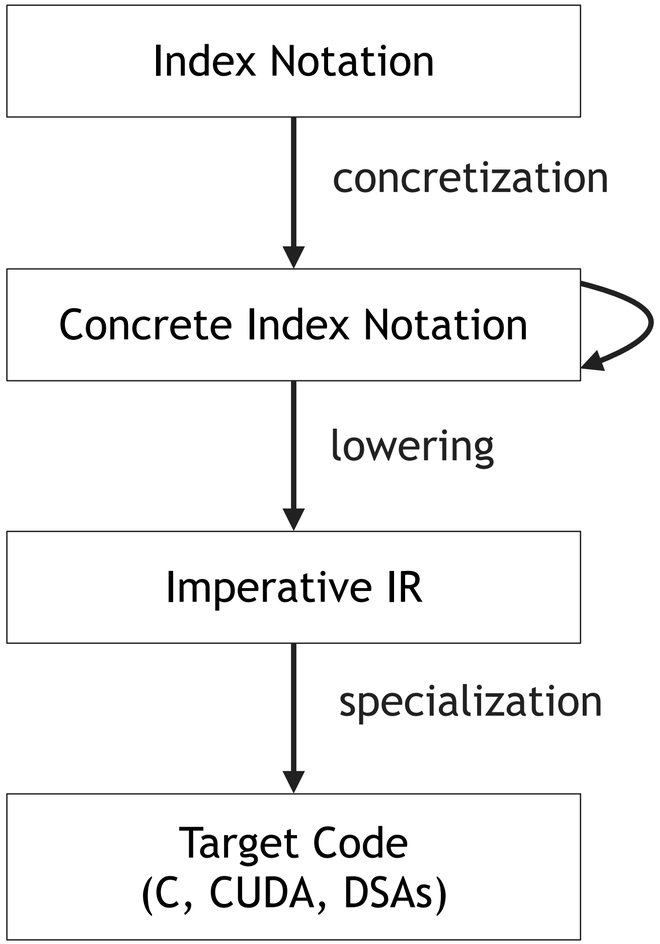
“parallelize(” index “,” p “,” r “)”

“unroll(”index “,” u“)”

Concrete index notation contains iteration graphs



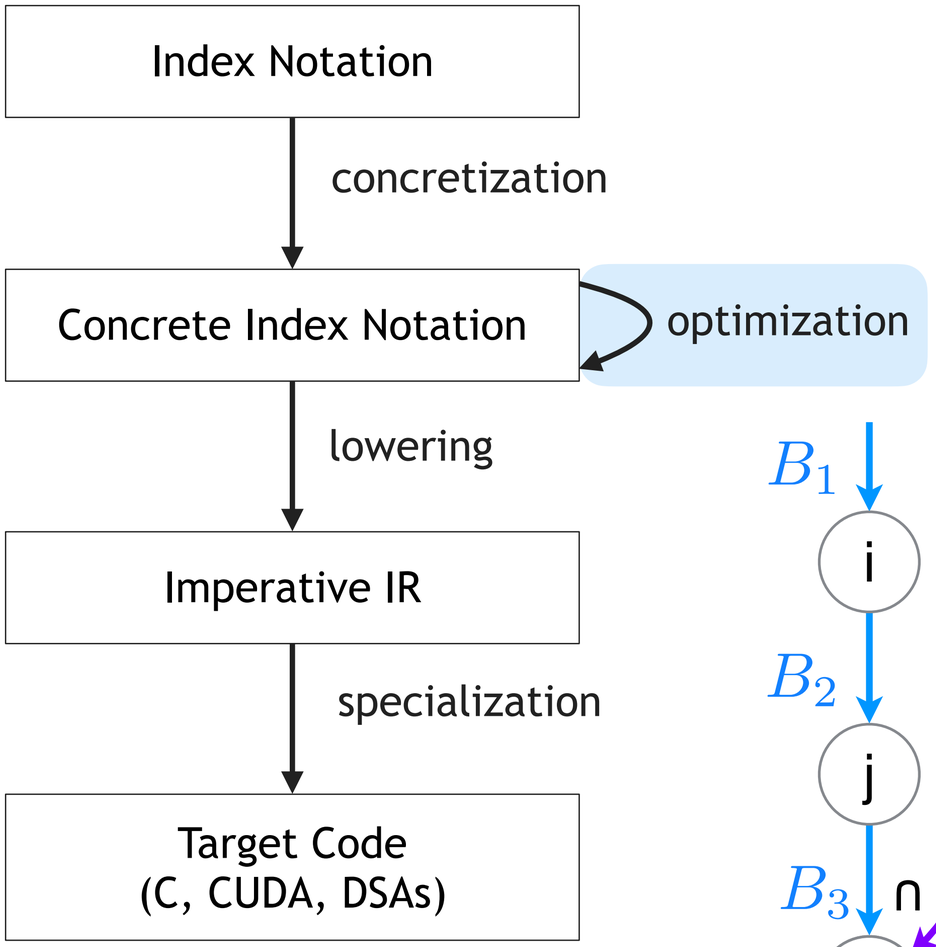
Concrete index notation as an optimization IR



scheduling

Optimizing transformations are applied before coiteration code is generated. Thus it applies equally to dense and sparse computation

Concrete index notation example



$$A_{ij} = \sum_k B_{ijk} c_k$$

Order of computation

Temporary storage

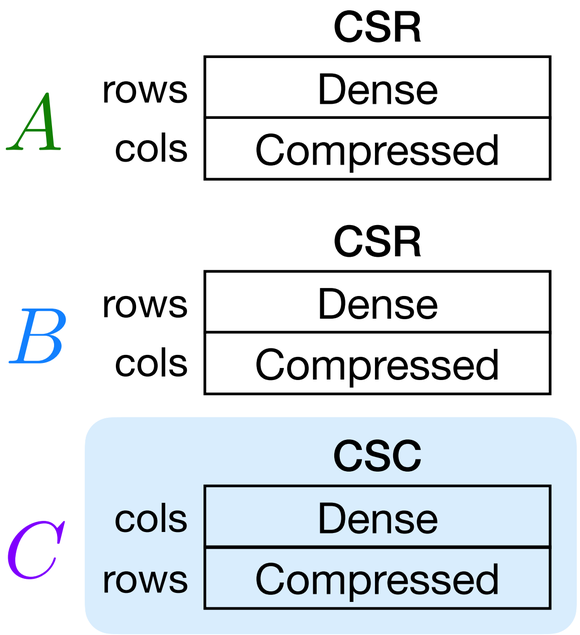
$$\forall_i \forall_j \forall_k \left(A_{ij} \mathrel{+}= t\right) \text{ where } \left(\forall_k t \mathrel{+}= B_{ijk} c_k\right)$$

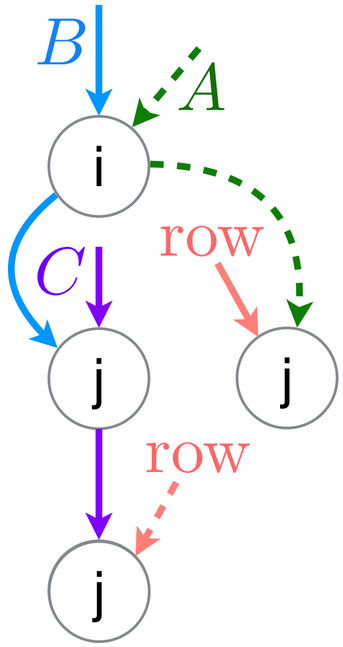

```
for (int i = 0; i < m; i++) {

  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
    int j = B2_crd[pB2];

    float t = 0.0;
    int pA2 = i*n + j;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while (pB3 < B3_pos[pB2+1] && pc1 < c1_pos[1]) {
      int kB = B3_crd[pB3];
      int kc = c1_crd[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        t += B3_vals[pB3] * c1_vals[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
    }
    A[pA2] = t;
  }
}
```

Workspace to scatter into results in sparse matrix multiplication

Linear Combination of Rows
Matrix Multiplication

$$A_{ij} = \sum_k B_{ik} C_{kj}$$





dense B_1

```
for (int i = 0; i < m; i++) {
```

compressed B_2
dense C_1

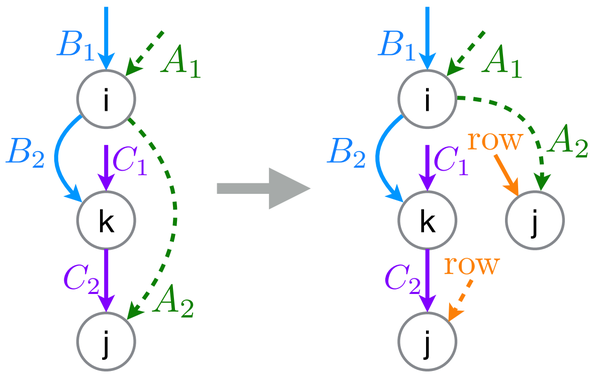
```
  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
    int k = B2_crd[pB2];
```

compressed C_2
dense row

```
    for (int pC2 = C2_pos[k]; pC2 < C2_pos[k+1]; pC2++) {
      int j = C2_crd[pC2];
      row[j] += B[pB2] * C[pC2];
    }

  }

  for (int pA2 = A2_pos[i]; pA2 < A2_pos[i+1]; pA2++) {
    int j = A2_crd[pA2];
    A[pA2] = row[j];
    row[j] = 0.0;
  }

}
```

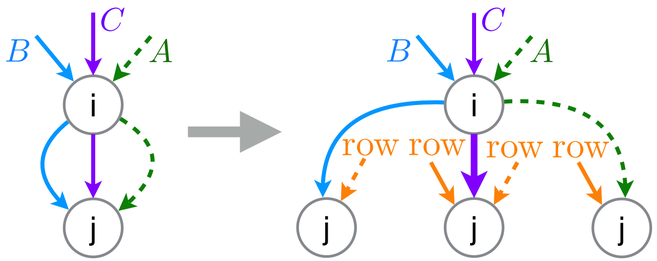
Other uses of where clauses

Scatter into results



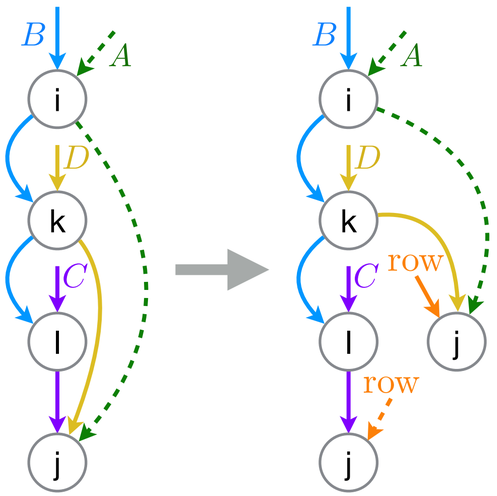
SpGEMM

Simplify merge code



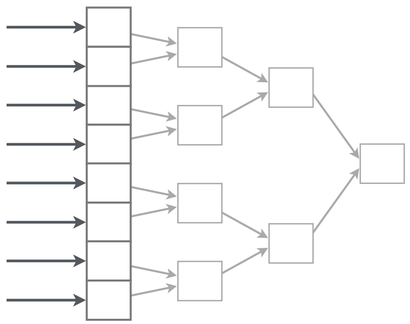
SpAdd

Loop-invariant code motion



MTTKRP

Prepare reductions



GPU shared memories

Mixed precision

```
for (int i = 0; i < m; i++) {
  double tj = 0.0;
  for (int pB2 = B2_pos[i];
           pB2 < B2_pos[i+1];
           pB2++) {
    int j = B2_crd[pB2];
    tj += B[pB2] * c[j];
  }
  a[i] = tj;
}
```

Single-precision floating point

A derived iteration space is one where dimensions have been split or collapsed

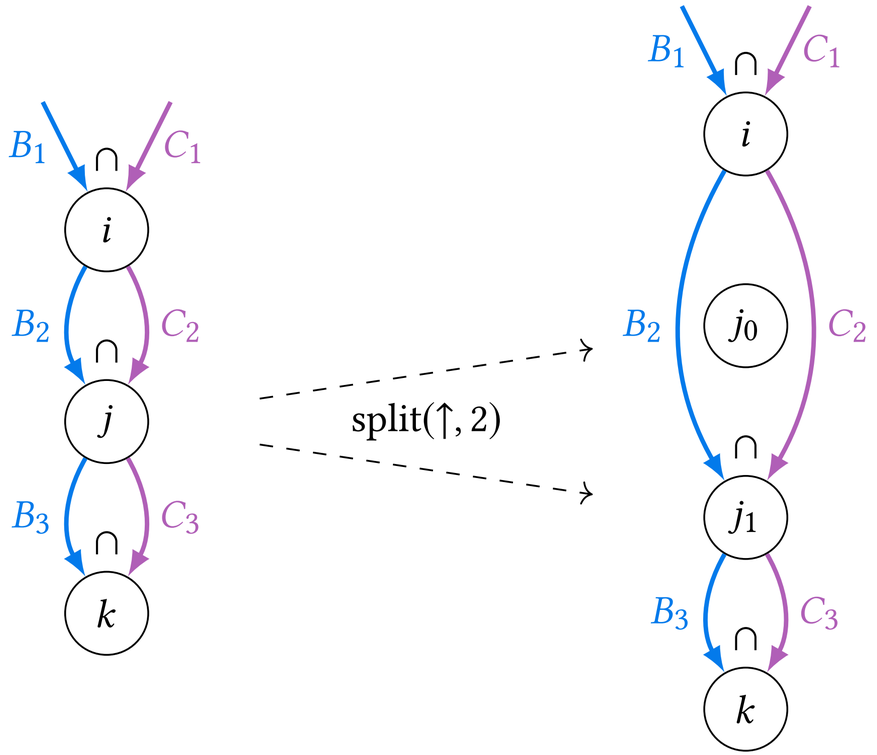
$$\forall_i\, a_i = b_i + c_i \xrightarrow{\text{split}} \forall_{i_0} \forall_{i_1}\, a_i = b_i + c_i : i \xrightarrow{\text{split}(\uparrow,4)} i_0 i_1$$

derived index variables

original index variable i

Cannot simply rewrite access expressions as with dense, so the mapping functions must be stored in the IR, so we can later emit code to remap coordinates.

The split iteration space function



$$\frac{\forall i \forall j \forall k\ B_{ijk} \cap C_{ijk}}{\begin{aligned} i &\in B_1 \cap C_1 \\ j &\in B_2 \cap C_2 \\ k &\in B_3 \cap C_3 \end{aligned}}$$

$$\frac{\forall i \forall j_0 \forall j_1 \forall k\ B_{ijk} \cap C_{ijk} \ : j \xrightarrow{\text{split}(\uparrow,2)} j_0 j_1}{\begin{aligned} i &\in B_1 \cap C_1 \\ j_0 &\in [0,2) \\ j_1 &\in B_2 \cap C_2 \cap [j_0 \cdot n/2, (j_0+1) \cdot n/2] \\ k &\in B_3 \cap C_3 \end{aligned}}$$

The split iteration space function comes in several variants

$$\text{split}(\underline{\text{direction}, \text{stride}}, [\underline{\text{tensor operand}}])$$

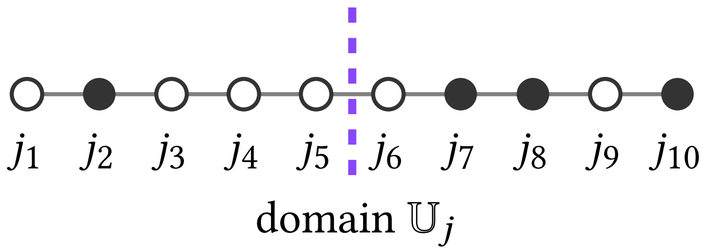
Split off a loop with stride iterations as the:

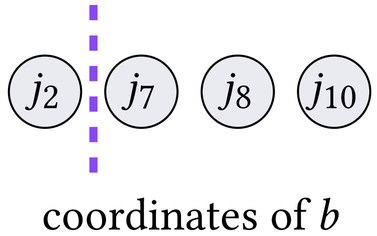
- outer loop (direction $= \uparrow$), or
- inner loop (direction $= \downarrow$)

Optional tensor operand whose nonzero coordinates the split applies to. If not given, the split applies to the universe of coordinates of the split index variable.

The split iteration space function can split with respect to the universe of coordinates or the coordinates of one operand

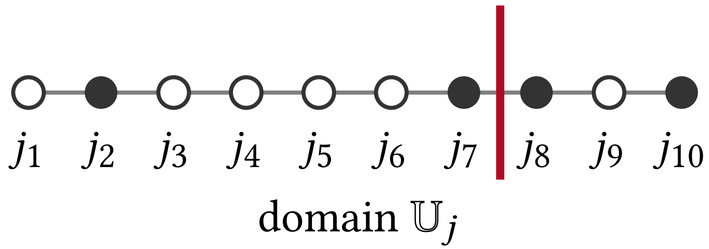
Universe split

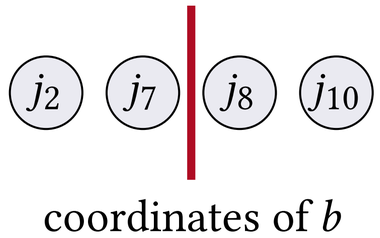




even split in coordinate universe, but
uneven split in b's nonzero coordinates

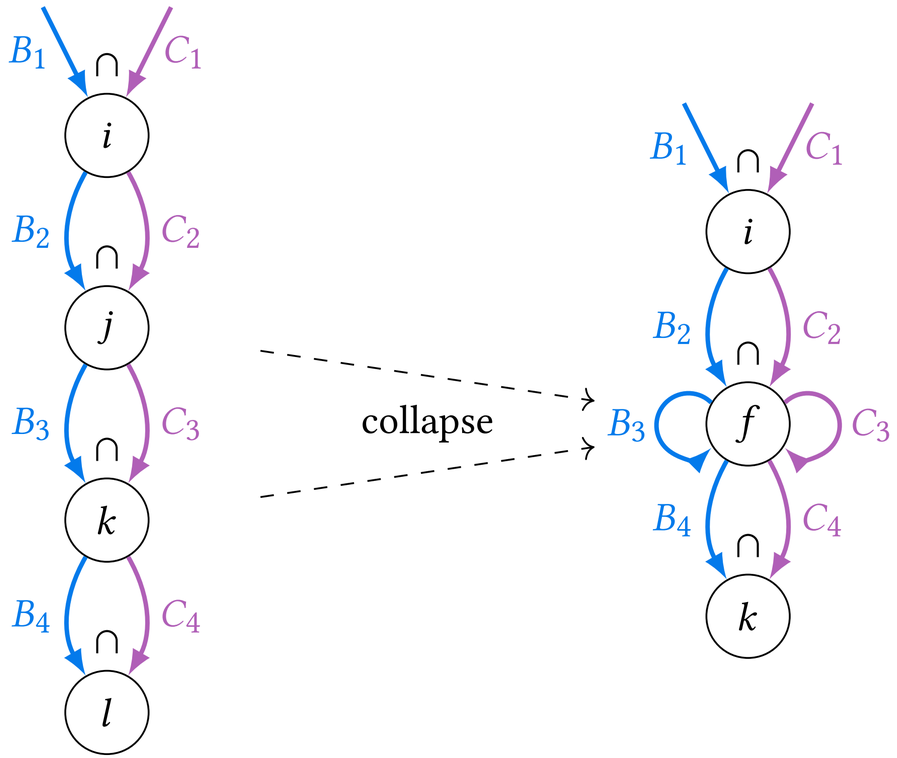
Coordinate tree split





uneven split in coordinate universe, but
even split in b's nonzero coordinates

The collapse iteration space function



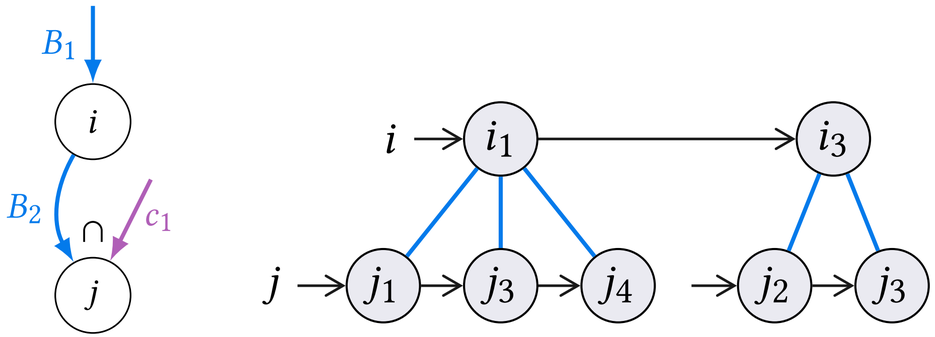
$$\frac{\forall i \forall j \forall k \forall l \; B_{ijkl} \cap C_{ijkl}}{\begin{aligned} i &\in B_1 \cap C_1 \\ j &\in B_2 \cap C_2 \\ k &\in B_3 \cap C_3 \\ l &\in B_4 \cap C_4 \end{aligned}}$$

$$\frac{\forall i \forall f \forall l \; B_{ijkl} \cap C_{ijkl} : jk \xrightarrow{\text{collapse}} f}{\begin{aligned} i &\in B_1 \cap C_1 \\ f &\in (B_2 \times B_3) \cap (C_2 \times C_3) \\ k &\in B_4 \cap C_4 \end{aligned}}$$

Iterate over Cartesian combination of coordinates in i and j.

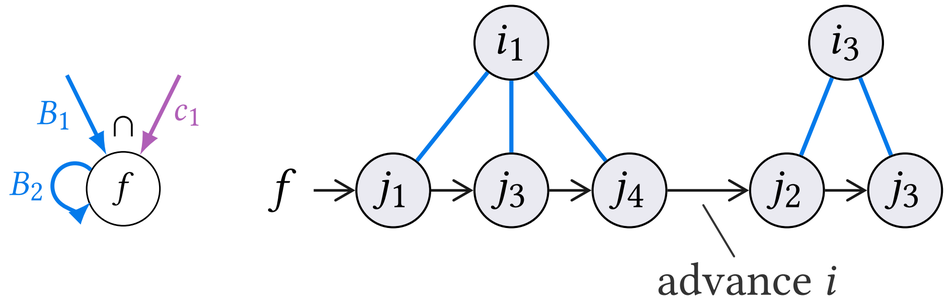
The collapse function leads to bottom-up iteration

Pre-collapse top-down iteration




```
for (int i = 0; i < m; i++) {
  for (int jB = B2_pos[i];
           jB < B2_pos[i+1]; jB++) {
    int j = B2_crd[jB];
    a[i] += B[jB] * c[j];
  }
}
```

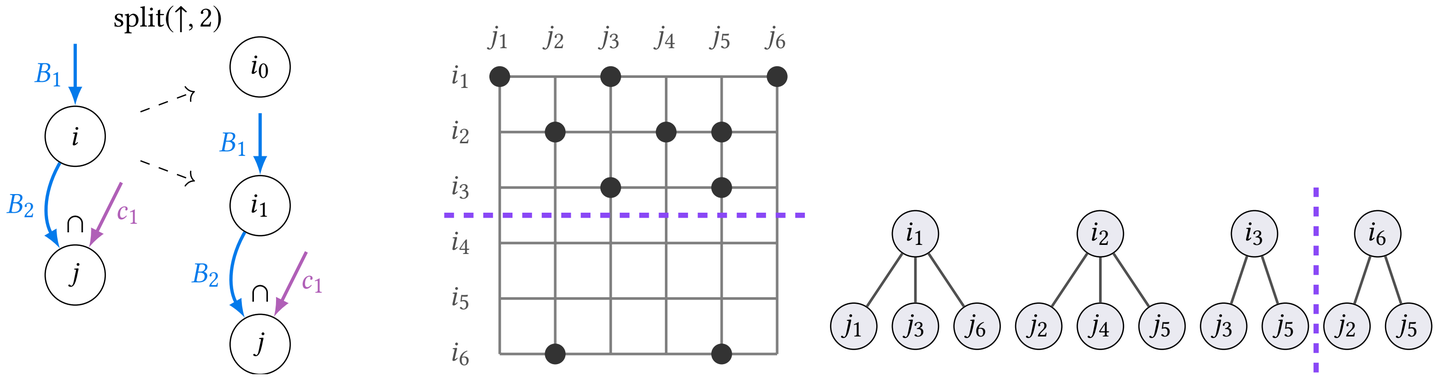
Post-collapse bottom-up iteration



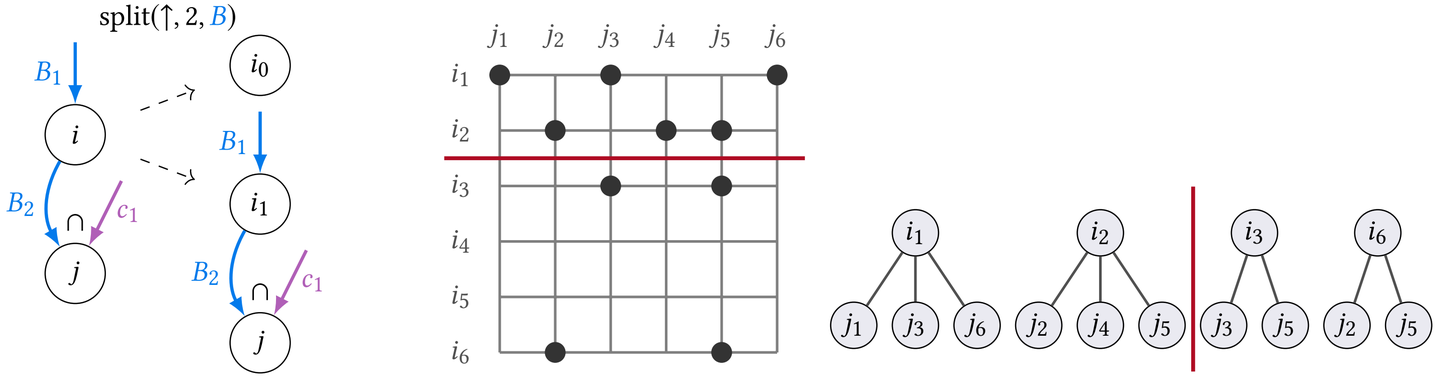

```
for (int f = 0, i = 0;
         f < B2_pos[m]; f++) {
  if (f >= B2_pos[m]) break;

  int j = B2_crd[f];
  while (f == B2_pos[i+1]) i++;

  a[i] += B[f] * c[j];
}
```

Two-dimensional tiling examples

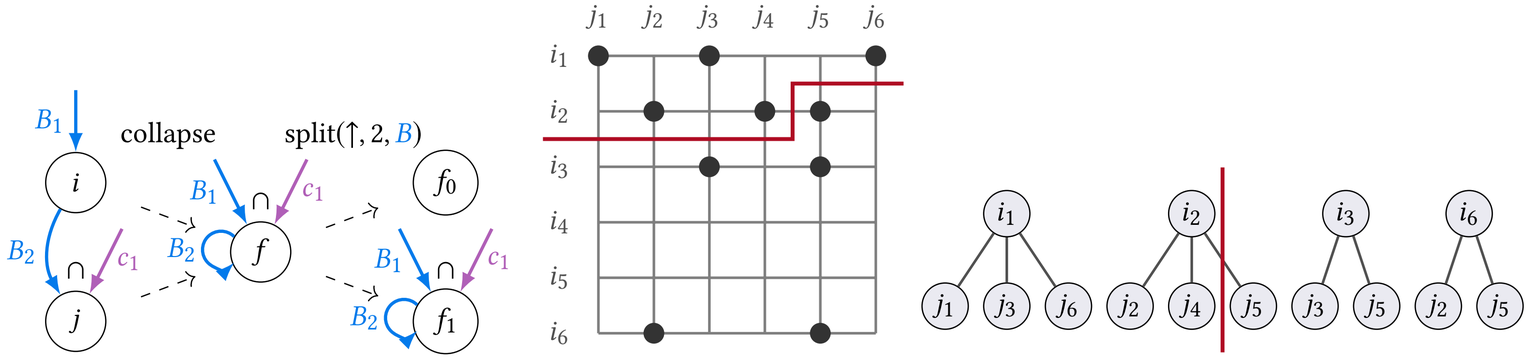
Universe split



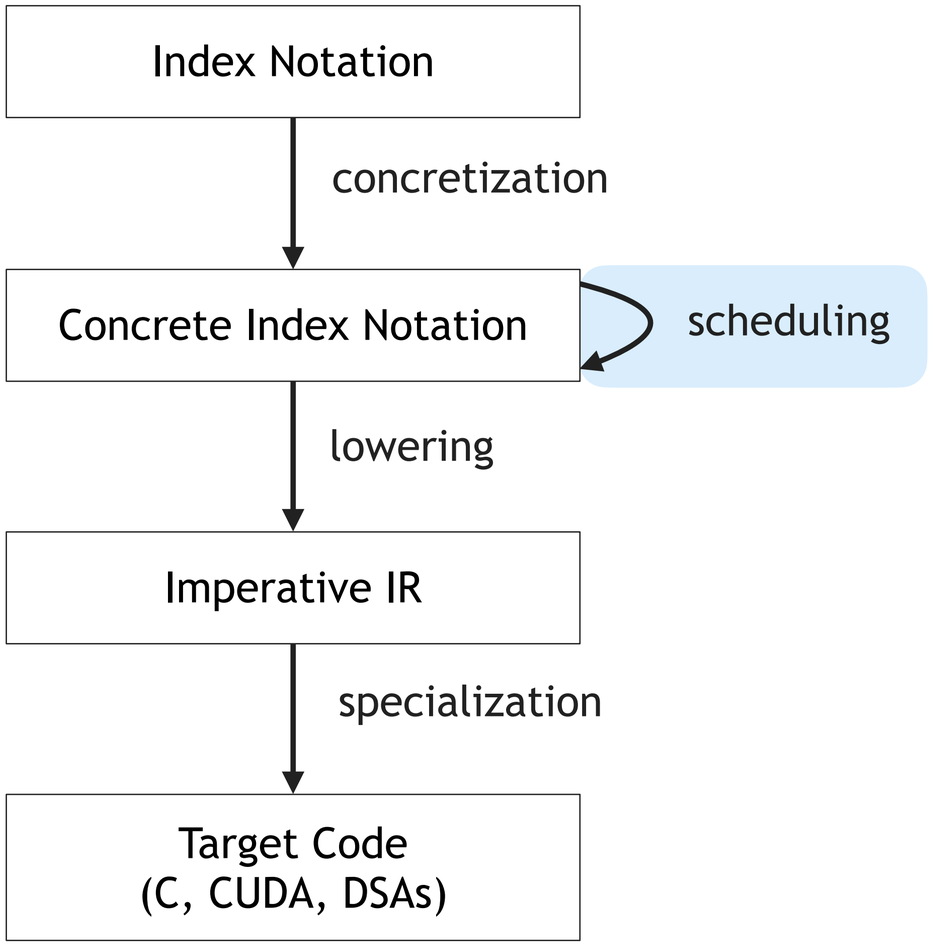
Coordinate tree split



Collapse+coordinate tree split



A sparse (tensor algebra) scheduling language



- **reorder(i, j)** interchanges loops i and j
- **split(i, i_1, i_2, d, s, t)** strip-mines i into two loops i_1 and i_2, where i_1 or i_1 is of size s depending on the direction d. The tensor t is optional and, if given, means the loop is strip-mined w.r.t. its nonzeros.
- **collapse(i, j, f)** collapses loops i and j into a new loop f, which iterates over their Cartesian combination.
- **precompute(S, e, t, I)** precomputes expression e in index statement S before the loops I and stores the results in tensor t.
- **unroll**, **parallelize**, **vectorize**, …

Lowering algorithm for code generation

function LOWER(assignment statement $S_{\text{assignment}}$)
    Emit compute code
end function

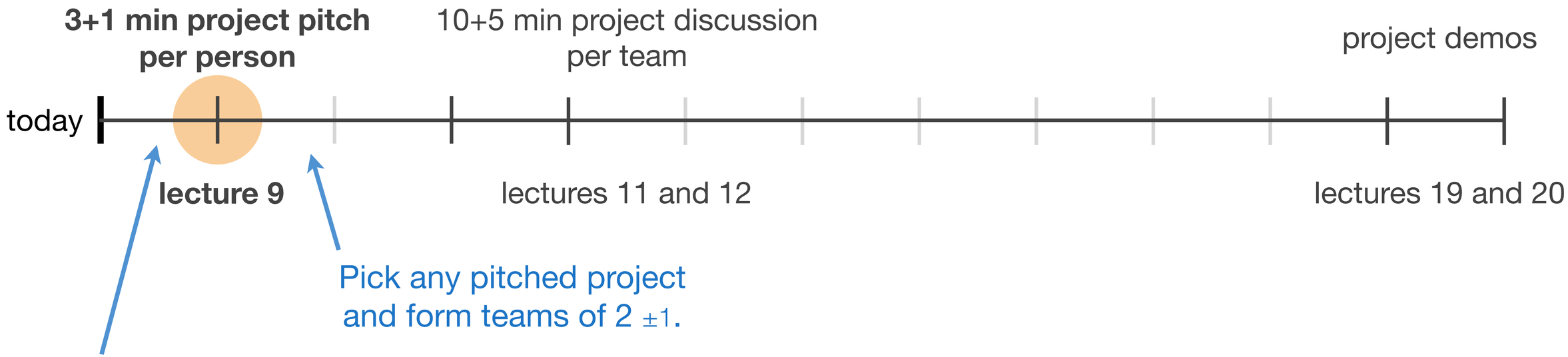
function LOWER(where statement S_{where})
    LOWER(producer statement of S_{where})
    LOWER(consumer statement of S_{where})
end function

function LOWER(sequence statement S_{sequence})
    LOWER(definition statement of S_{sequence})
    LOWER(mutation statement of S_{sequence})
end function

function LOWER(multi statement S_{multi})
    LOWER(left statement of S_{multi})
    LOWER(right statement of S_{multi})
end function

function LOWER(forall statement S_{forall} of index variable i)
    let $\mathcal{L}$ be an iteration lattice constructed from S_{forall}
    Emit initialize iterators
    for each lattice point $\mathcal{L}_p$ in $\mathcal{L}$ **do**
        Emit loop header
        Emit access iterators
        Emit map candidate coordinates to the original space
        Emit resolve the coordinate of i
        Emit map resolved coordinate to each derived space
        Emit locate from locators
        for each lattice point $\mathcal{L}_q < \mathcal{L}_p$ in $\mathcal{L}$ **do**
            Emit conditional header
            let $S_{\text{simplified}}$ be a statement constructed from
                the body of S_{forall} by removing operands
                that have run out of values in $\mathcal{L}_q$
            LOWER($S_{\text{simplified}}$)
            Emit assembly code
            Emit conditional footer
        end for
        Emit advance iterators
        Emit loop footer
    end for
end function

Course Project



Each person contributes one pitch slide to a google slide deck. These pitches are not binding.