Mid-level Compiler Optimizations and Transformations

Uday Kumar Reddy Bondhugula

udayb@iisc.ac.in

Dept of CSA
Indian Institute of Science

OUTLINE

- Data Dependences, Transformations, Parallelization
- Locality
- Affine Transformations
- Parallelism
- Tiling, Fusion, Vectorization
- Other Complementary Transformations

Iteration Spaces and Dependences

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
      A[t+1][i] = f(A[t][i+1], A[t][i], A[t][i-1]);
```

1. **Iteration Domains**
   - Every statement has a domain or an index **set** – instances that have to be executed
   - Each instance is a vector (of loop index values from outermost to innermost)
     $D_S = \{[t, i] \mid 0 \leq t \leq T - 1,\ 1 \leq i \leq N\}$
2. **Dependences**
   - A dependence is a **relation** between domain instances that are in conflict (more on next slide)

LEXICOGRAPHIC ORDERING

- **Lexicographic ordering**: $\succ$, $\prec$, $\vec{x} \succ \vec{y}$, $\succ \vec{0}$
- **Transformations** as a way to provide multi-dimensional timestamps
- Code generation: **Scanning points in the transformed space in lexicographically increasing order**

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

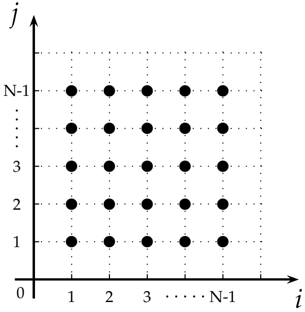


Figure: Original space (i, j)

- **Domain**: $\{[i, j] \mid 1 \leq i, j \leq N - 1\}$

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

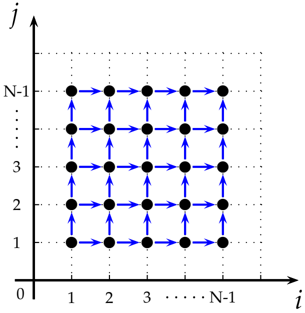


Figure: Original space (i, j)

- **Dependences**:
  1. $\{[i,j] \to [i+1,j] \mid 1 \leq i \leq N-2, 0 \leq j \leq N-1\}$ — **(1,0)**
  2. $\{[i,j] \to [i,j+1] \mid 1 \leq i \leq N-1, 0 \leq j \leq N-2\}$ — **(0,1)**

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

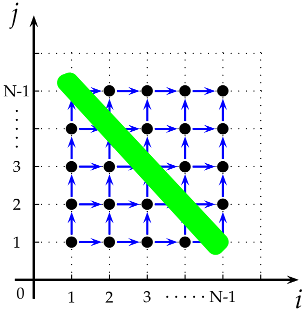


Figure: Original space (i, j)

- **Dependences**:
  1. $\{[i,j] \to [i+1,j] \mid 1 \le i \le N-2, 0 \le j \le N-1\}$ — **(1,0)**
  2. $\{[i,j] \to [i,j+1] \mid 1 \le i \le N-1, 0 \le j \le N-2\}$ — **(0,1)**

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

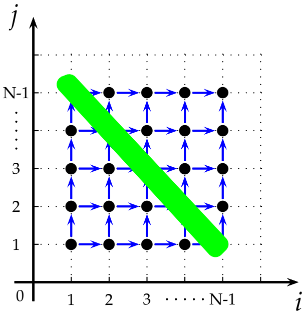


Figure: Original space (i, j)

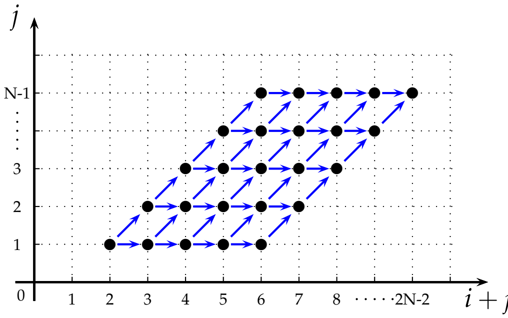


Figure: Transformed space $(i + j, j)$

- **Transformation**: $T(i, j) = (i + j, j)$
  - Dependences: (1,0) and (0,1) now become (1,0) and (1,1) resp.
  - Inner loop is now parallel

Domains, Dependences, and Transformations

```
for (i=1; i<=N-1; i++)
  for (j=1; j<=N-1; j++)
    A[i][j] = f(A[i-1][j], A[i][j-1]);
```

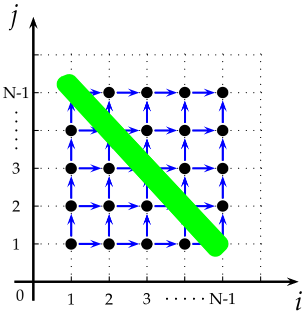


Figure: Original space (i, j)

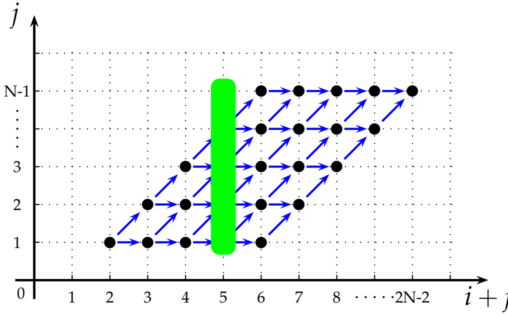


Figure: Transformed space $(i + j, j)$

- **Transformation**: $T(i, j) = (i + j, j)$
  - Dependences: (1,0) and (0,1) now become (1,0) and (1,1) resp.
  - Inner loop is now parallel

Dependences: Another Example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    A[i] = f(A[i+1], A[i], A[i-1]);
```

- Compute the dependences
- Transitivity in dependences?
- Remove transitively covered dependences.

Dependences: Another Example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    A[i] = f(A[i+1], A[i], A[i-1]);
```

- Compute the dependences
- Transitivity in dependences?
- Remove transitively covered dependences.

Dependences: Yet Another Example

```
for (i = 0; i < N; i++)
  for (j = 1; j < i; j++)
    A[j] = A[j] - A[j]/A[i];
```

- Compute the dependences.

Dependence Representations

1. Distance vectors: constant dependences
2. Dependence levels: depth at which a dependence is carried
3. Direction vectors: direction of the dependence along each dimension
4. Dependence as presburger formulae, relations on integer sets with affine constraints and existential quantifiers

Dependence Testing

- GCD test, GCD tightening of constraints
- Guassian elimination, Fourier-Motzkin elimination (super-exponential) complexity
- Omega test

Outline

- Data Dependences, Transformations, Parallelization
- Locality
- Affine Transformations
- Parallelism
- Tiling, Fusion, Vectorization
- Other Complementary Transformations

Characterizing Reuse

- Reuse through multi-dimensional array accesses
  1. Self reuse
  2. Group reuse
- In space or in time?
  1. Spatial reuse (self or group)
  2. Temporal reuse (self or group)
- Under what conditions does an access exhibit spatial or temporal reuse along a specific outer loop?
  - This topic is well-covered in the Dragon textbook.
- Degree of temporal reuse: Dimensionality of the iteration space minus rank of the access function
  Eg: *for* (i, j, k), access A[i + j][j][j] has an access function of rank two in an iteration space of dimensionality three → one degree of temporary reuse.

Representation of Array Accesses

1. Linear Algebraic representation of “regular” accesses
2. Affine access functions can be analyzed by the compiler easily for reuse, dependences, optimization, and parallelization
3. Refer to the definition of affine functions earlier
4. Handling compositions of mod and floordiv functions in accesses requires additional techniques to determine spatial and temporal reuse

Loop Nests: Some definitions

- **Perfectly nested** loop nest: A sequence of successively nested loops (from outermost to innermost) where every loop other than the innermost one has a single loop as the only statement in its body.
- **Imperfectly nested**: not perfectly nests.

```
// Perfectly nested.
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      S(t, i, j);
```

```
// (t, i, j) is imperfectly nested, but
// (t, i) is perfectly nested.
for (t = 0; t < T; t++) {
  for (i = 1; i < N+1; i++) {
    S1(t, i);
    for (j = 1; j < N+1; j++)
      S2(t, i, j);
  }
}
```

OUTLINE

- Data Dependences, Transformations, Parallelization
- Locality
- Affine Transformations
- Parallelism
- Tiling, Fusion, Vectorization
- Other Complementary Transformations

Affine Transformations

- Examples of affine functions of i, j: $i + j$, $i - j$, $i + 1$, $2i + 5$
- Not affine: ij, i^2, $i^2 + j^2$, $a[j]$

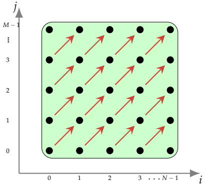


Figure: Iteration space

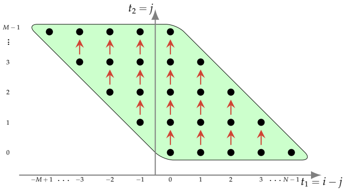


Figure: Transformed space

```
// O(N) synchronization if j is parallelized.
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
     A[i+1][j+1] = f(A[i][j]);
```

```
// Synchronization-free.
#pragma omp parallel for private(t2)
for (t1=-M+1; t1<=N-1; t1++)
  for (t2 = max(0,-t1); t2 <= min(M-1,N-1-t1); t2++)
    A[t1+t2+1][t2+1] = f(A[t1+t2][t2]);
```

- Transformation: $(i, j) \rightarrow (i - j, j)$

Affine Transformations

- Examples of affine functions of i, j: $i + j$, $i - j$, $i + 1$, $2i + 5$
- Not affine: ij, i^2, $i^2 + j^2$, $a[j]$

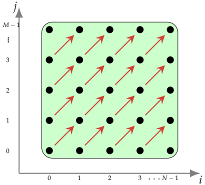


Figure: Iteration space

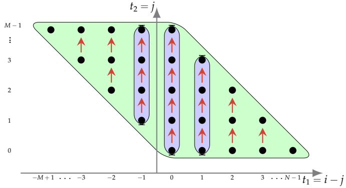


Figure: Transformed space

```
// O(N) synchronization if j is parallelized.
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    A[i+1][j+1] = f(A[i][j]);
```

```
// Synchronization-free.
#pragma omp parallel for private(t2)
for (t1=-M+1; t1<=N-1; t1++)
  for (t2 = max(0,-t1); t2 <= min(M-1,N-1-t1); t2++)
    A[t1+t2+1][t2+1] = f(A[t1+t2][t2]);
```

- Transformation: $(i, j) \rightarrow (i - j, j)$

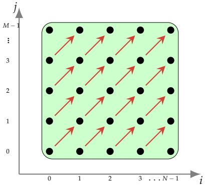


Figure: Iteration space

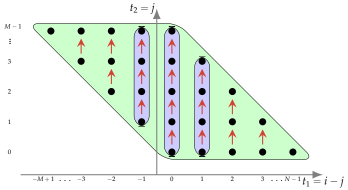


Figure: Transformed space

- Affine transformations are attractive because:
  - Preserve **collinearity** of points and **ratio of distances** between points
  - Code generation with affine transformations has thus been studied well (CLooG, ISL, OMEGA+)
  - Model a very rich class of loop re-orderings
  - Useful for several domains like dense linear algebra, stencil computations, image processing pipelines, deep learning

Finding Good Affine Transformations

(i, j)	Identity
(j, i)	Interchange
$(i + j, j)$	Skew i (by a factor of one w.r.t j)
$(i - j, -j)$	Reverse j and skew i
$(i, 2i + j)$	Skew j (by a factor of two w.r.t i)
$(2i, j)$	Scale i by a factor of two
$(i, j + 1)$	Shift j
$(i + j, i - j)$	More complex
$(i/32, j/32, i, j)$	Tile
$\ldots$	

- One-to-one functions
- Can be expressed using matrices:

$$T(i, j) = (i + j, j) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}.$$

- Unimodular and non-unimodular transformations

Finding Good Affine Transformations

(i, j)	Identity
(j, i)	Interchange
$(i + j, j)$	Skew i (by a factor of one w.r.t j)
$(i - j, -j)$	Reverse j and skew i
$(i, 2i + j)$	Skew j (by a factor of two w.r.t i)
$(2i, j)$	Scale i by a factor of two
$(i, j + 1)$	Shift j
$(i + j, i - j)$	More complex
$(i/32, j/32, i, j)$	Tile
$\ldots$	

- One-to-one functions
- Can be expressed using matrices:
  $T(i, j) = (i + j, j) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}.$
- Unimodular and non-unimodular transformations

DEPENDENCES

- Dependences are determined pairwise between conflicting accesses

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- Dependence notations
  - Distance vectors: (1,-1,0), (1,0,0), (1,1,0), (1,0,-1), (1,0,1)
  - Direction vectors
  - Dependence relations as integer sets with affine constraints and existential quantifiers or Presburger formulae — powerful
- Consider the dependence from the write to the third read: $A[(t+1)\%2][i][j] \rightarrow A[t'\%2][i'-1][j']$
  Dependence relation: $\{[t,i,j] \rightarrow [t',i',j'] \mid t' = t+1, i' = i+1, j' = j, 0 \leq t \leq T-1,\ 0 \leq i \leq N-1, 0 \leq j \leq N\}$

PRESERVING DEPENDENCES

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                            A[t%2][i][j+1], A[t%2][i][j-1]);
```

- For affine loop nests, these dependences can be analyzed and represented precisely
- **Next step:** Transform while preserving dependences
  - Find execution reorderings that **preserve** dependences and improve performance
  - Execution reordering as a function: $T(\vec{i})$
  - For all dependence relation instances $(\vec{s} \to \vec{t})$,
    $T(\vec{t}) - T(\vec{s}) \succ \vec{0}$,
    i.e., the source should precede the target even in the transformed space
- What is the structure of **T**?

Valid Transformations

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- Dependences: $(1,0,0)$, $(1,0,1)$, $(1,0,-1)$, $(1,1,0)$, (1,-1,0)
- Validity: $T(\vec{t}) - T(\vec{s}) \succ \vec{0}$, i.e., $T(\vec{t} - \vec{s}) \succ \vec{0}$
- Examples of invalid transformations
  - $T(t,i,j) = (i,j,t)$
  - Similarly, (i,t,j), (j,i,t), $(t+i,i,j)$, $(t+i+j,i,j)$ are all invalid transformations
- Valid transformations
  - (t,j,i), $(t,t+i,t+j)$, $(t,t+i,t+i+j)$
  - However, only some of the infinitely many valid ones are interesting

Generating Loops After Transformation

- Fourier-Motzkin elimination can be used to generate code
  - Successively eliminate old loop variables, and then new loop variables from innermost to outermost, generating bounds for the loop being eliminated at each step.
  - Replace old loop IVs with new ones in the loop body
- More powerful techniques exist to generate more efficient code (fewer/no redundancy in loop bound checks, conditional guards)
- Work out for this example transformation: $(i, j) \rightarrow (i + j, j)$.

Parallelism and Dependence Carrying

- Carrying or satisfying a dependence
- Loop-carried dependence
- A loop is parallel if does not carry any dependences.
- For each dependence, determine the depth at which it is carried
- For constant distance vectors, the depth of the first non-zero dependence component is the depth at which the dependence is satisfied

Synchronization-Free or Communication-free Parallelism

- Number of degrees of synchronization-free parallelim
- m: Dimensionality of the iteration space
- D: Dependence matrix – columns are distance vectors
- m - $rank(D)$ degrees of synchronization-free parallelism
- For any perfect loop nest that has only constant dependences, we can always obtain at least $m - 1$ degrees of parallelism.
- How do you determine or maximize synchronization-free parallelism? Find T (transformation matrix) that satisfies certain properties.
- Find $\vec{t} \neq \vec{0}$ such that $\vec{t}.\vec{d_i} = 0, \forall \vec{d_i}$ (dependence distance vector).

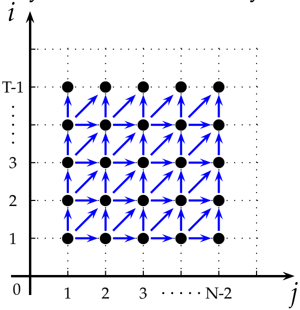
Wavefront Parallelism

- Synchronization required after execution of a parallel loop
    - A single outer sequential loop with N iterations containing all inner parallel loops will lead to O(N) synchronization
- Refer illustration earlier in this chapter: $(i + j, j)$ mapping for an example
- Connection to *DoAcross* parallelism, as opposed to *DoAll parallelism*?
- It's possible to parallelize using barrier-style synchronization or point-to-point synchronization (between specific pairs of processors)

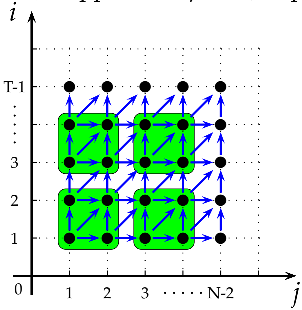
Outline

TILING (BLOCKING)

- Partition and execute iteration space in blocks
- A tile is executed atomically
- Benefits: exploits *cache locality* & improves *parallelization* in the presence of synchronization
- **Allows reuse in multiple directions**
- **Reduces frequency of synchronization** for parallelization: synchronization after you execute *tiles* (as opposed to *points*) in parallel



$(\mathbf{i}, \mathbf{j}) \rightarrow (\mathbf{i}/\mathbf{50}, \mathbf{j}/\mathbf{50}, \mathbf{i}, \mathbf{j});$



$(\mathbf{i}, \mathbf{j}) \rightarrow (\mathbf{i}/\mathbf{50} + \mathbf{j}/\mathbf{50}, \mathbf{j}/\mathbf{50}, \mathbf{i}, \mathbf{j})$

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
             A[i%2][j], A[i%2][j+1]);
```

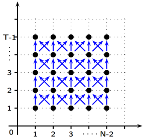


Figure: Iteration space

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
            A[i%2][j], A[i%2][j+1]);
```

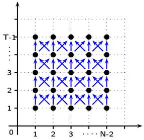


Figure: Iteration space

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
            A[i%2][j], A[i%2][j+1]);
```

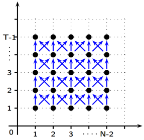


Figure: Iteration space

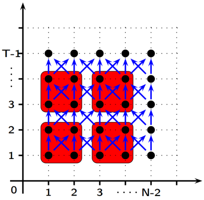


Figure: Invalid tiling

Validity of Tiling (blocking)

- Validity of tiling
  - There should be no cycle between the tiles
  - **Sufficient condition**: All dependence components should be non-negative along dimensions that are being tiled
  - Dependences: (1,0), (1,1), (1,-1)

```
for (i=1; i<T; i++)
  for (j=1; j<N-1; j++)
    A[(i+1)%2][j] = f(A[i%2][j-1],
            A[i%2][j], A[i%2][j+1]);
```

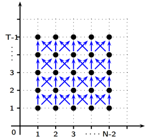


Figure: Iteration space

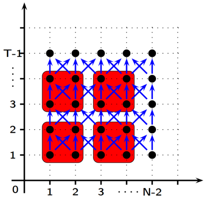


Figure: Invalid tiling

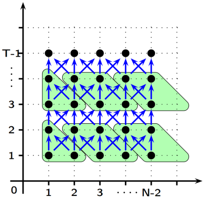


Figure: Valid tiling

Tiling (blocking)

- Affine transformations can enable tiling
  - First skew: $T(i,j) = (i, i+j)$

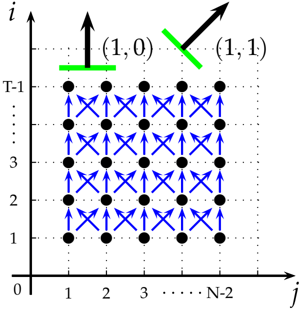


Figure: Original space (i, j)

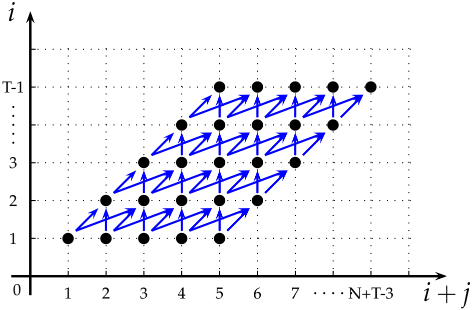


Figure: Transformed space $(i, i+j)$

- Affine transformations can enable tiling
  - First skew: $T(i,j) = (i, i+j)$
  - Then, apply (rectangular) tiling:
    $T(i,j) = (i/64, (i+j)/64, i, i+j)$
    - i and $i+j$ are also called *tiling hyperplanes*

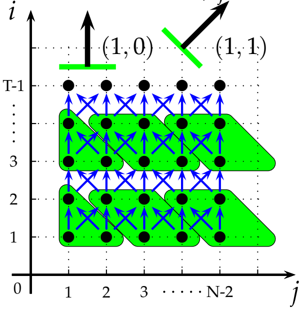


Figure: Original space (i,j)

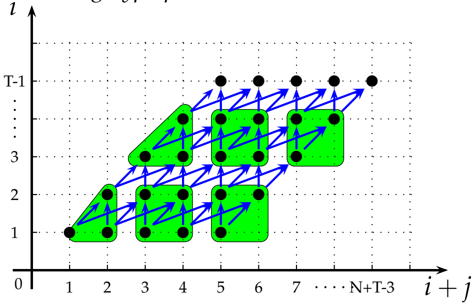


Figure: Transformed space $(i, i+j)$

Back to 3-D example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64+(t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Back to 3-D example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                           A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64+(t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Back to 3-d example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                          A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64+(t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Back to 3-D example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N+1; i++)
    for (j = 1; j < N+1; j++)
      A[(t+1)%2][i][j] = f((A[t%2][i+1][j], A[t%2][i][j], A[t%2][i-1][j],
                          A[t%2][i][j+1], A[t%2][i][j-1]);
```

- What is a good transformation here to improve parallelism and locality?
- Demo
  - Skewing: $(t, t+i, t+j)$
  - Tiling: $(t/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$
  - Tile wavefront: $(t/64+(t+i)/64,\ (t+i)/64,\ (t+j)/1000,\ t,\ t+i,\ t+j)$

Other Transformations and Optimizations

- Loop Fusion
- Loop Distribution
- Vectorization
- Explicit copying (Packing)
- Unroll-and-Jam, Register Tiling
- Complementary/enabling transformations for Parallelism
  - Privatization, Scalar expansion, Array Expansion
  - Trade-off between parallelism and memory usage
- Reductions - parallelization and vectorization

Loop Fusion: Validity

- A fine (or finer) grained interleaving of the execution of multiple loop nests
- Validity: fusion is valid if, for every loop being fused, there are no dependences from the first nest body to the second nest body that have a negative component on the loop being fused while not being carried by any outer loops
- Data Dependence Graph (DDG) needed to model "inter-statement" dependences to analyze the above conditions
  - Statements (IR operations or groups of IR operations) are nodes of this graph
  - Each edge corresponds to a dependence from the source node to the target node
  - Directed graph, can have multiple edges between nodes and self edges.
  - Each edge has information on the source and target memory accesses involved in the dependence and additional information.

Fusion: Example

```
// Original code.
// Produces B[i] using another array A.
for (i = 0; i < N - 1; i++)
  B[i] = A[i] + A[i + 1];
// Consumes B[i] to create C[i].
for (i = 0; i < N - 1; i++)
  C[i] = B[i];
```

```
// Fused code.
for (i = 0; i < N - 1; i++) {
  B[i] = A[i] + A[i + 1];
  C[i] = B[i];
}
```

```
// Fusion not valid without shifting the second nest forward by one.
for (i = 0; i < N; i++)
  B[i] = A[i];
// Consumes B[i] to create C[i].
for (i = 0; i < N - 1; i++)
  C[i] = B[i] + B[i + 1];
```

- Fusion can be enabled other transformations: shifting, permutation/interchange
- Fusion can be partial as well, i.e., not fusing all loops
- For partial fusion, consider dependence components up until the loops being fused.

Fusion: Other Examples

```
// Original code.
// Produces B using another array A.
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    B[i][j] = A[i][j];
// Consumes B to create C. Fusion is valid.
// Dependence carried on the fused 'i' loop.
for (i = 0; i < N; i++)
  for (j = 0; j < N - 1; j++)
    C[i][j] = B[i][j] + B[i - 1][j + 1];
```

```
// Original code.
// Produces B using another array A.
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    B[i][j] = A[i][j];
// Consumes B to create C.
for (i = 1; i < N; i++)
  for (j = 0; j < N - 1; j++)
    C[i - 1][j] = B[i][j] + B[i - 1][j];
```

Loop Fusion and Distribution: Costs/Benefits

- Benefits
  1. Improves cache locality: producer-consumer reuse, input reuse
  2. Improves register reuse
  3. Eliminates intermediate arrays and reduces memory consumption
  4. Reduces code size, less control overhead
- Disadvantages
  1. Reduces effective cache capacity available for each of components fused: cache capacity misses
  2. Increases the risk of conflict misses
  3. Can lead to loss of parallelism, loss of tilability, or loss of vectorizability
  4. Increases hardware prefetch stream utilization; can lead to lower prefetching performance

Loop Distribution

- Loop distribution is the inverse of fusion
- Two operations/statements part of the same strongly connected component of the data dependence graph can't be distributed
- Distribution at the inner level or partial distribution: consider only a part of the DDG, discarding dependences carried on outer loops that aren't being considered for distribution.
- Maximal distribution: distribute out all strongly connnected components of a loop nest.
- Disadvantages of fusion are the benefits of distribution

Vectorization

- A fine-grained parallelization: single instruction on multiple data (SIMD)
- Vectorization, SIMDization used synonymously today
- An efficient form of parallelization with minimal additional hardware resources
- Reduction in the number of instructions executed
- The instructions that form a vector can come from a loop body ("superword-level parallelism") or from a loop ("loop vectorization")

Loop Vectorization: Examples

```
// Vectorizable loop.
for (i = 0; i < N; i++)
  C[i] = A[i] + B[i];

// Non-vectorizable loop.
for (i = 2; i < N; i++)
  A[i] = A[i - 1] + A[i - 2];

// A loop doesn't have to be parallel to be vectorizable.
// Loop i is vectorizable despite not being parallel and despite
// carrying a short loop dependence. No dependence cycle.
for (i = 0; i < N; i++) {
  C[i + 1] = A[i] * B[i];
  D[i] = C[i] + X[i];
}
// Vectorizing a loop body like this can also be viewed as tiling by vector
// width, distributing the intra-tile loops, and vectorizing them.
```

Loop Vectorization: Validity

- A loop can be vectorized only if there is no dependence cycle betweeen the instructions that spans less than the "vector width" iterations.
- Contiguity: Data being loaded for a vector may need to be contiguous in memory; depends on hardware
- Alignment: data may have to be aligned depending on the hardware – modern general-purpose processors typically don't have an alignment requirement
- Performance of aligned vs unaligned memory operations

Vectorization: Example

```
// Original code.
affine.for %i = 0 to 4096 {
 affine.for %j = 0 to 4096 {
  affine.for %k = 0 to 4096 {
   %lhs = affine.load %A[%i, %k] : memref<4096x4096xf32>
   %rhs = affine.load %B[%k, %j] : memref<4096x4096xf32>
   %in = affine.load %C[%i, %j] : memref<4096x4096xf32>
   %product = arith.mulf %lhs, %rhs : f32
   %acc = arith.addf %in, %product : f32
   affine.store %acc, %C[%i, %j] : memref<4096x4096xf32>
  }
 }
}
```

```
// Interchanged %j to innermost and vectorized 8-way along the %j loop.
affine.for %i = 0 to 4096 {
 affine.for %k = 0 to 4096 {
  affine.for %j = 0 to 4096 step 8 {
   %lhs = affine.load %A[%i, %k] : memref<4096x4096xf32>
   %v_lhs = vector.splat %lhs : vector<8xf32>
   %v_rhs = affine.vector_load %B[%k, %j] : memref<4096x4096xf32>
   %product = arith.mulf %v_lhs, %v_rhs : vector<8xf32>
   %in = affine.vector_load %C[%i, %j] : memref<4096x4096xf32>
   %acc = arith.addf %in, %product : vector<8xf32>
   affine.vector_store %acc, %C[%i, %j] : memref<4096x4096xf32>
  }
 }
}
```

Explicit Copying or Packing

- Typically performed in conjunction with tiling
- Pack data being accessed by a 'tile' into a contiguous buffer that fits in cache/fast memory
- 'Compute' tile reads from packed input buffers and writes out to a packed buffer; unpack output buffer.
- Benefits
  1. Eliminates conflicts misses and thus improves cache locality
  2. Reduces TLB misses
  3. Improves prefetching performance (fewer hardware prefetch streams used)
- Packing involves overhead (copy-in and copy-out)
- Reference: see packing scheme for high-performance matrix-matrix multiplication in this illustration: Analytical Modeling is Enough for High Performance BLIS, Low et al., ACM TOMS 2016.

Unroll-and-jam or Register tiling

- Improves register reuse
- Multi-dimensional unroll-and-jam (multiple loops) can be performed to simultaneously exploit register reuse along multiple dimensions
- Can be thought of as tiling for register locality except that the tiles are small (variables being reused to fit in registers ideally) and the tile is fully unrolled.
- Improves the compute to load/store operation ratio – extremely important for high-performance on modern architectures
- Sufficient: if it is valid to make a loop the innermost loop, it is valid to unroll-and-jam it.
- More precise: unroll-and-jam is valid iff stripminng the loop by the unroll-and-jam factor and bringing the intra-tile loop to the innermost position is valid
- Multi-dimensional unroll-and-jam (multiple loops)

Unroll-and-jam or Register tiling (continued)

- For a matrix-matrix multiplication in the canonical *ijk* form, work out the improvement in compute to load/store ratio when unroll-and-jamming i and j loops with factors U_i and U_j respectively.
- Assume a register budget of 16 registers in one case and 32 registers in another.

REDUCTIONS

- Reductions can be parallelized
- Reductions can be vectorized

```
s = 0;
for (i = 0; i < N; i++)
  s += A[i];
```

A composition of transformations

```
for (i = 1 i < N; i++)
  // S1.
  B[i] = A[i];

for (i = 1; i < N; i++)
  // S2.
  C[i - 1] = B[i] + B[i - 1]
```

- Original ordering: $T_{S_1}(i) = (0, i)$, $T_{S_2}(i) = (1, i)$
- Fused + Tiled + Innermost loop distribution
  - Produce a chunk of A and consume it before a new chunk is produced
  - Transformation: $T_{S_1}(i) = (i/32, 0, i)\,, \quad T_{S_2}(i) = (i/32, 1, i)$.

```
for (t1=0;t1<=floord(N-1,32);t1++) {
  for (t3=max(1,32*t1;t3<=min(N-1,32*t1+31);t3++)
    B[t3] = A[t3];
  for (t3=max(1,32*t1);t3<=min(N-1,32*t1+31);t3++)
    C[t3 - 1] = B[t3] + B[t3 - 1];
}
```

  - Provides cache locality while also providing parallelism and vectorization.
  - Either locality or parallelism/vectorizability would have otherwise been lost with only fusion or only parallelizing without any fusion.

Algorithms to Find Transformations

- **The history**
  - A data locality optimizing algorithm, Wolf and Lam, PLDI 1991: Improve locality through unimodular transformations
    - Characterize self-spatial, self-temporal, and group reuse
    - Find unimodular transformations (permutation, reversal, skewing) to transform to permutable loop nests with reuse, and subsequently tile them
- Several advances on polyhedral transformation algorithms through 1990s and 2000s: Feautrier [1991–1992], Lim and Lam (Affine Partitioning) [1997–2001], Pluto [2008–2015]
- **The Present**
  - Polyhedral framework provides a powerful mathematical abstraction (away from the syntax)
  - A number of new techniques, open-source libraries and tools have been developed and are actively maintained
  - Affine abstractions and infrastructure in MLIR