Compile-time Composition of Run-time Data and Iteration Reordering (PLDI ’03)

Michelle Mills Strout  
UC, San Diego

Larry Carter  
UC, San Diego

Jeanne Ferrante  
UC, San Diego
Introduction

• Data locality and parallelism can be enhanced by data and loop transformation.

• Recent compile-time frameworks restricted to affine loop bonds and affine array references.

• Most compile-time frameworks assume dependence when faced with non-affine memory references (such as A[B[i]]).

• Kelly and Pugh framework

• Presburger arithmetic for non-affine memory references
  • Data-mapping between loop iterations and data locations  
  • Dependence between loop iterations
Introduction

• Advantages of compile-time framework
  • Both run-time and compile-time transformations are uniformly described
  • Transformation legality checks provides constraints on run-time reordering functions
  • Overhead involved in generating the run-time reordering functions can be reduced
Introduction

• Contributions made:
  • Experimental results shows significant performance improvements
  • How to use an existing compile-time framework to describe a number of existing run-time data and iteration-reordering transformations that improve data locality
  • Sparse tiling technique
  • Experimental results shows moving data to new location only once reduces the overhead of composed run-time reordering transformations.
Motivation for Compositions

• A framework that allows compile-time composition of run-time reordering transformations by showing that benchmarks with hand-coded composed transformations result in improved performance.
Motivation for Compositions

Outer loop $S$ makes amortization of run-time reordering overhead possible.

left, right as access or index array

$\text{x, } \text{vx, } \text{fx as base or data array}$

Loop $j$, non-affine memory references

```
do $s = 1 \text{ to } \text{num\_steps}$
  do $i = 1 \text{ to } \text{num\_nodes}$
    $x[i] = x[i] + \text{vx}[i] + \text{fx}[i]$
    enddo
  S1
  do $j = 1 \text{ to } \text{num\_inter}$
    $\text{fx}[\text{left}[j]] += g(x[\text{left}[j]], x[\text{right}[j]])$
    $\text{fx}[\text{right}[j]] += g(x[\text{left}[j]], x[\text{right}[j]])$
    enddo
  S2
  do $k = 1 \text{ to } \text{num\_nodes}$
    $\text{vx}[k] += \text{fx}[k]$
    enddo
  S3
  enddo
S4
```

Figure 1: Simplified moldyn example
Run-time Data Reordering

• Improve spatial locality in the loop by reordering the data based on the order in which it is referred in the loop
• Consecutive packing (CPACK) and graph partitioning (Gpart)

Figure 2: Example mapping of iterations in the j loop to locations in the data arrays x and fx. Here, circles represent iterations of the j loop inside one iteration of the outer time-stepping loop.

Figure 3: Example of Figure 2 mapping after the CPACK data reordering.
Run-time Iteration Reordering

• Often beneficial follow a data reordering.
• Reorder iterations based on the order in which the loop touched the data
• Lexicographical grouping (lexGroup). Same or adjacent data locations execute consecutively.

Figure 4: Example of Figure 2 mapping after the CPACK data reordering followed by a lexGroup iteration reordering.
Sparse Tiling

- Sparse tiling results in run-time generated tiles or slices that cut between loops or across outer loop.
- Improve locality between loops or iterations of an outer loop.
- Improve locality when there are data dependence.
- Existing run-time iteration reordering inspectors traverse data mapping, sparse tiling inspectors traverse dependences.
- Only applied to Gauss-Seidel.
- Provided coarser granularity of parallelism.

Figure 5: We highlight the iterations of one sparse tile for the code in Figure 1. The j loop has been blocked to provide a seed partitioning. In the full sparse tiled executor code, the iterations within a tile are executed atomically.
Experimental Results

Figure 6: Normalized execution time without overhead on the Power3.

Figure 7: Normalized execution time without overhead on the Pentium 4.
Experimental Results

Figure 8: Amortization of the overhead on the Power3 in number of outer loop iterations based on the savings per iteration of the outer loop.

Figure 9: Amortization of the overhead on the Pentium 4 in number of outer loop iterations based on the savings per iteration of the outer loop.
Framework Terminology

• Kelly-Pugh iteration reordering framework terminology

• Loop, Statement, and Data

\[
I_0 = \{ [s, 1, i, 1] \mid (1 \leq s \leq \text{num\_steps}) \\
\quad \quad \quad \quad (1 \leq i \leq \text{num\_nodes}) \} \cup \\
\{ [s, 2, j, q] \mid (1 \leq s \leq \text{num\_steps}) \\
\quad \quad \quad \quad (1 \leq j \leq \text{num\_inter}) \\
\quad \quad \quad \quad (1 \leq q \leq 2) \} \cup \\
\{ [s, 3, k, 1] \mid (1 \leq s \leq \text{num\_steps}) \\
\quad \quad \quad \quad (1 \leq k \leq \text{num\_nodes}) \}
\]

\[
M_{I_0 \rightarrow x_0} = \{ [s, 1, i, 1] \rightarrow [i] \}
\quad \cup \{ [s, 2, j, q] \rightarrow [\text{left}(j)] \}
\quad \cup \{ [s, 2, j, q] \rightarrow [\text{right}(j)] \}
\quad \cup \{ [s, 3, k, 1] \rightarrow [k] \}
\]

\[
x_0 = \{ [m] \mid 1 \leq m \leq \text{num\_nodes} \}
\]

\[
v_{x_0} = \{ [m] \mid 1 \leq m \leq \text{num\_nodes} \}
\]

\[
f_{x_0} = \{ [m] \mid 1 \leq m \leq \text{num\_nodes} \}
\]

\[
\text{left}_0 = \{ [m] \mid 1 \leq m \leq \text{num\_inter} \}
\]

\[
\text{right}_0 = \{ [m] \mid 1 \leq m \leq \text{num\_inter} \}
\]

\[
M_{I_0 \rightarrow v_{x_0}} = \{ [s, 1, i, 1] \rightarrow [i] \}
\quad \cup \{ [s, 3, k, 1] \rightarrow [k] \}
\]

\[
M_{I_0 \rightarrow \text{left}_0} = \{ [s, 2, j, q] \rightarrow [j] \}
\]

\[
M_{I_0 \rightarrow \text{right}_0} = M_{I_0 \rightarrow \text{left}_0}
\]
Framework Terminology

• Dependences
  • $x$ and $f_x$ array dependences between $S_1$ ([s,1,i,1]), $S_2$ ([s,2,j,1]), $S_3$ ([s,2,j,2])
    \[ d_{24} \cup d_{34} = \{ [s, 2, j, q] \rightarrow [s', 3, k, 1] \mid (s \leq s') \]
    \[\wedge (1 \leq q \leq 2)\]
    \[\wedge (k = left(j) \lor k = right(j))\}\]
  • $f_x$ array dependences between $S_1$ ([s,2,j,1]), $S_2$ ([s,2,j,2]), $S_3$ ([s,3,k,1])
    \[ d_{12} \cup d_{13} = \{ [s, 1, i, 1] \rightarrow [s', 2, j, q] \mid (s \leq s') \]
    \[\wedge (1 \leq q \leq 2)\]
    \[\wedge (i = left(j) \lor i = right(j))\}\]
Run-time Reordering Transformations

• Data reordering \( M_{I \rightarrow a'} = \{ \vec{p} \rightarrow R_{a \rightarrow a'}(m) | m \in M_{I \rightarrow a}(\vec{p}) \land \vec{p} \in I \} \)
  • No legality, will not change dependences

• Iteration reordering \( \forall \vec{p}, \vec{q} : \vec{p} \rightarrow \vec{q} \in D_{I \rightarrow I} \Rightarrow T_{I \rightarrow I'}(\vec{p}) < T_{I \rightarrow I'}(\vec{q}) \)
  • New iteration order must respect all the dependences of the origin

\[
D_{I' \rightarrow I'} = \{ T_{I \rightarrow I'}(\vec{p}) \rightarrow T_{I \rightarrow I'}(\vec{q}) | \vec{p} \rightarrow \vec{q} \in D_{I \rightarrow I} \}
\]

• New data mapping

\[
M_{I' \rightarrow a} = \{ T_{I \rightarrow I'}(\vec{p}) \rightarrow M_{I \rightarrow a}(\vec{p}) | \vec{p} \in I \}
\]

• Sparse tiling: partition a portion of the subspace and grow tiles that respect data dependences through the rest of the subspace
Composing Transformations – Run-time Data Reordering

- CPACK data reordering
- Can be specified at compile-time by changing all the data mappings which involve the array being reordered

```c
CPACK_M_I0_to_x0(left,right)
// initialize alreadyOrdered bit vector
// to all false
count = 0
do j=1 to num_inter
  mem_loc1 = left[j]
  mem_loc2 = right[j]
  if not alreadyOrdered(mem_loc1)
    sigma.cp_inv[count] = mem_loc1
    alreadyOrdered(mem_loc1) = true
    count = count + 1
  endif
  if not alreadyOrdered(mem_loc2)
    sigma.cp_inv[count] = mem_loc2
    alreadyOrdered(mem_loc2) = true
    count = count + 1
  endif
endo
do i=1 to num_nodes
  if not alreadyOrdered(i)
    sigma.cp_inv[count] = i
    count = count + 1
  endif
endo
return sigma.cp_inv
```

$M_{i_0 \rightarrow x_1} = \{[s, 1, i, 1] \rightarrow [\sigma_{cp}(i)]\}$

$\cup \{[s, 2, j, q] \rightarrow [\sigma_{cp}(left(j))]|j \neq i\}$

$\cup \{[s, 2, j, q] \rightarrow [\sigma_{cp}(right(j))]|j \neq i\}$

$\cup \{[s, 3, k, 1] \rightarrow [\sigma_{cp}(k)]\}$

Figure 10: First CPACK inspector for moldyn called from composed inspector in Figure 11.
Composing Transformations – Run-time Iteration Reordering

• Iteration reordering transformations after CPACK data reordering

\[
T_{t_0 \rightarrow t_1} = \{ [s, 1, i, 1] \rightarrow [s, 1, i_1, 1] \mid i_1 = \sigma_{c_p}(i) \} \\
\cup \{ [s, 2, j, q] \rightarrow [s, 2, j_1, q] \mid j_1 = \delta_{i_g}(j) \} \\
\cup \{ [s, 3, k, q] \rightarrow [s, 3, k_1, 1] \mid k_1 = \sigma_{c_p}(k) \}
\]

• Data mapping and dependences after iteration reordering

\[
M_{t_1 \rightarrow x_1} = \{ [s, 1, \sigma_{c_p}(i), 1] \rightarrow [\sigma_{c_p}(i)] \} \\
\cup \{ [s, 2, \delta_{i_g}(j), q] \rightarrow [\sigma_{c_p}(\text{left}(j))] \} \\
\cup \{ [s, 2, \delta_{i_g}(j), q] \rightarrow [\sigma_{c_p}(\text{right}(j))] \} \\
\cup \{ [s, 3, \sigma_{c_p}(k), q] \rightarrow [\sigma_{c_p}(k)] \}
\]

\[
d'_{12} \cup d'_{13} = \{ [s, 1, \sigma_{c_p}(i), 1] \rightarrow [s', 2, \delta_{i_g}(j), q] \mid (s \leq s') \land (1 \leq q \leq 2) \land (i = \text{left}(j) \lor i = \text{right}(j)) \}
\]

\[
d'_{24} \cup d'_{34} = \{ [s, 2, \delta_{i_g}(j), q] \rightarrow [s', 3, \sigma_{c_p}(k), 1] \mid (s \leq s') \land (1 \leq q \leq 2) \land (k = \text{left}(j) \lor k = \text{right}(j)) \}
\]
Subsequent Transformations - Inspector

// First application of CPACK
sigma.cp.inv = CPACK.M.I0.to.x0(left,right)
sigma.cp = calcInverse(sigma.cp.inv)

// First application of lexGroup
delta.lg = lexGroup.M.I0.to.x1(left,right, sigma.cp)
delta.lg.inv = calcInverse(delta.lg)

// Second application of CPACK
sigma.cp2.inv = CPACK.M.I1.to.x1(left,right, sigma.cp,delta.lg.inv)
sigma.cp2 = calcInverse(sigma.cp2.inv)

// Second application of lexGroup
delta.lg2 = lexGroup.M.I1.to.x2(left,right, sigma.cp,delta.lg.inv,sigma.cp2)
delta.lg2.inv = calcInverse(delta.lg2)

// Reorder data arrays to reflect
// final data mapping
x2 = remapArray_R.x0.to.x2(x, sigma.cp, sigma.cp2)

// Adjust values in index arrays to
// reflect final data mapping
left = adjustIndexArray_R.x0.to.x2(left, sigma.cp,sigma.cp2)
right = adjustIndexArray_R.x0.to.x2(right, sigma.cp,sigma.cp2)

// Reorder index arrays to implement
// final iteration reordering
left2 = remapArray_T.I0.to.I2(left, delta.lg,delta.lg2)
right2 = remapArray_T.I0.to.I2(right, delta.lg,delta.lg2)

Figure 11: Composed inspector for CPACK, lexGroup, CPACK, and lexGroup composition.
Subsequent Transformations - Executor

```
    do s = 1 to num_steps
        do i2 = 1 to num_nodes
            x2[i2] = x2[i2] + vx2[i2] + fx2[i2]
        enddo

        do j2 = 1 to num_inter
            fx2[left2[j2]] += g(x2[left2[j2]],
                                x2[right2[j2]])
            fx2[right2[j2]] += g(x2[left2[j2]],
                                 x2[right2[j2]])
        enddo

        do k2 = 1 to num_nodes
            vx2[k2] += fx2[k2]
        enddo
    enddo
```

Figure 13: Executor for simple moldyn example when inspector applies CPACK, lexGroup, CPACK, lexGroup composition.
Sparse Tiling

Figure 14: Sparse tiled executor when the composed inspector performs CPACK, lexGroup, CPACK, lexGroup, full sparse tiling, and tilePack.

• Sparse tiling inspector traverse the data dependences, others only traverse data mapping

\[
T_{l_2 \rightarrow l_3} = \begin{cases}
[s, 1, i_2, 1] \rightarrow [s, \theta(1, i_3), 1, i_3, 1] \mid i_3 = i_2 \\
\cup \{[s, 2, j_2, g] \rightarrow [s, \theta(2, j_3), 2, j_3, g] \mid j_3 = j_2 \} \\
\cup \{[s, 3, k_2, 1] \rightarrow [s, \theta(3, k_3), 3, k_3, 1] \mid k_3 = k_2 \}
\end{cases}
\]
Reducing The Overhead

• Whenever two sets of data dependences satisfy the same constraints, it is only necessary to traverse one set at run-time

• Remapping and adjusting the index array after each transformations and remapping data arrays after all run-time reordering functions have been done
Reducing The Overhead

// First application of CPACK
sigma_cp_inv = CPACK_M_I0_to_x0(left, right)
sigma_cp    = calcInverse(sigma_cp_inv)
// Reorder data arrays to reflect data mapping
x1          = remapArray_R_x0_to_x1(x, sigma_cp)
// Adjust values in index arrays
left         = adjustIndexArray_R_x0_to_x1(left, sigma_cp)
right        = adjustIndexArray_R_x0_to_x1(right, sigma_cp)

// Second application of lexGroup
delta_lg     = lexGroup_M_I0_to_x1_B(left, right)
// Reorder index arrays to implement lexGroup
left1        = remapArray_T_I0_to_I1(left, delta_lg)
right1       = remapArray_T_I0_to_I1(right, delta_lg)

// Second application of CPACK
sigma_cp2_inv= CPACK_M_I1_to_x1_B(left1, right1)
sigma_cp2    = calcInverse(sigma_cp2_inv)

// Reorder data arrays to reflect data mapping
x2          = remapArray_R_x1_to_x2(x1, sigma_cp2)
// Adjust values in index arrays
left1       = adjustIndexArray_R_x1_to_x2(left1, sigma_cp2)
right1      = adjustIndexArray_R_x1_to_x2(right1, sigma_cp2)

// Second application of lexGroup
delta_lg2    = lexGroup_M_I1_to_x2_B(left1, right1)
// Reorder index arrays to implement
// final iteration reordering
left2        = remapArray_T_I1_to_I2(left1, delta_lg2)
right2       = remapArray_T_I1_to_I2(right1, delta_lg2)

Figure 15: Composed inspector for CPACK, lexGroup, CPACK, and lexGroup where data remapping and index array updates are done immediately after the relevant reordering function is generated.
Reducing the Overhead

Figure 10: Percent reduction in inspector overhead for compositions with two or more data reorderings and data is only remapped once.
Future Work

• Automatically generation of specialized run-time inspectors.
  • Automatically generating specialized and optimized versions of data and iteration reordering algorithm such as CPACK, lexGroup, and FST is challenging

• How to choose between various compositions and the parameters for compositions are always challenging

Figure 17: Normalized execution time without overhead on the Pentium 4 as the Gpart and full sparse tiling parameters are selected to target half of L1 cache, L1 cache, 1/2 of L2 cache, and L2 cache.
Conclusion

- Experimental results shows significant performance improvements for moldyn, nbf and irreg benchmark
- Use an existing compile-time framework to describe changes in dependences and data mappings
- Sparse tiling technique can be represented in their framework
- Using two optimizations to improve inspectors, reducing the overhead of composed run-time reordering transformations