A Practical Automatic Polyhedral Parallelizer and Locality Optimizer

JIAHUAN HE (JACK)
APRIL 9TH
The Goal

This paper presents an automatic optimization that aiming to find good ways of tiling for parallelism and locality through affine transformations framework.
Tiling for Parallelization

Tiling for parallelism **partition** the **iteration** space into tiles that can be **concurrently** executed with a reduced frequency and volume of inter-processor communication.

A tile is atomically executed on a processor with communication required only before and after execution.
Tiling for Locality Optimization

Tiling for locality requires grouping points in an iteration space into smaller blocks (tiles).

These blocks allow data reuse in multiple directions when the block fits in a faster memory (registers, L1, or L2 cache).
Streamline
Dependence Test

The dependence polyhedron is in the sum of the dimensionalities of the source and target statement’s polyhedral.

Let $s$ represent the source iteration and $t$ be the target iteration pertaining to a dependence edge $e$. 

$$\langle \vec{s}, \vec{t} \rangle \in P_e$$
Dependence Test

Dependence satisfaction:

An affine dependence with polyhedron $P_e$ is satisfied at a level $l$ iff the following condition is satisfied:

$$\forall k (1 \leq k \leq l-1): \phi_{s_j}^k (\vec{t}) - \phi_{s_i}^k (\vec{s}) \geq 0, \langle \vec{s}, \vec{t} \rangle \in P_e$$

and

$$\phi_{s_j}^l (\vec{t}) - \phi_{s_i}^l (\vec{s}) \geq 1, \langle \vec{s}, \vec{t} \rangle \in P_e$$
Legality of Tiling Multiple Domains With Affine Dependences

**Lemma 1.** Let $\phi_{s_i}$ be a one-dimensional affine transform for statement $S_i$. For $\{\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_k}\}$, to be a legal (statement-wise) tiling hyperplane, the following should hold for each edge $e \in E$:

$$\phi_{s_j}(t) - \phi_{s_i}(\vec{s}) \geq 0, \quad \langle \vec{s}, \vec{t} \rangle \in P_e$$

(2)

Let $\{\phi^1_{s_1}, \phi^1_{s_2}, \ldots, \phi^1_{s_k}\}, \{\phi^2_{s_1}, \phi^2_{s_2}, \ldots, \phi^2_{s_k}\}$ be two statementwise 1-d affine transforms that satisfy (2). Then these two sets represent tileable loops in the transformed space.
Requirement for Tiling

One tile can be formed by aggregating a group of hyperplane instances along $\phi_{s_i}^1$ and $\phi_{s_i}^2$.

That means any solution for LEMMA 1 represents a common dimension for all statements in the transformed space with intra-statement affine dependences.
Cost Function

\[ \delta_e(\vec{s}, \vec{t}) = \phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}), \quad (\vec{s}, \vec{t}) \in \mathcal{P}_e \quad (3) \]

- This function is the number of hyperplanes the dependence \( e \) traverses along the hyperplane normal.
- If \( \phi \) is used as a space loop to generate tiles for parallelization, this function is a factor in the communication volume.
- On the other hand, if \( \phi \) is used as a sequential loop, it gives us a measure of the reuse distance.
- Therefore, minimize the function will reduce the cache miss or cache load to register.
Apply Farkas Lemma to Minimize

\[ \phi_{s_j}(\bar{t}) - \phi_{s_i}(\bar{s}) \leq v(\bar{p}), \quad (\bar{s}, \bar{t}) \in \mathcal{P}_e, \quad \forall e \in E \]
\[ v(\bar{p}) - \delta_e(\bar{s}, \bar{t}) \geq 0, \quad (\bar{s}, \bar{t}) \in \mathcal{P}_e, \quad \forall e \in E \quad (4) \]
\[ v(\bar{p}) - \delta_e(\bar{s}, \bar{t}) \equiv \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek} \mathcal{P}_e^k, \quad \lambda_{ek} \geq 0 \]

Because (2) is the basic constraint of (4), we can apply Farkas Lemma.
Then, coefficients of iterators can be gathered and equated from both LHS and RHS.
Get Constraints from Farkas Lemma

Since the loop variables themselves can be bounded by affine functions of the parameters, one can always find an affine form in the program parameters, \( p \), that bounds the function for every dependence edge.

\[
v(p) = u \cdot \bar{p} + w, \quad u = (u_1, u_2, \ldots, u_k)
\]
More Considerations

Since we have a number of possible solutions based on the constraints given by Farkas Lemma, finding a lexicographic minimal solution with $u$ and $w$ is the way to solve this problem.

For RAR (Read after Read) dependence, we still need to preserve it by applying the constraint:

$$|\phi_{s_j}(t) - \phi_{s_i}(s)| \leq v(p), \quad \langle s, t \rangle \in \mathcal{P}^R$$
More Considerations

We need at least as many independent solutions (for a statement) as the dimensionality of its domain.

To secure the linear independence with exist solutions, we add new constraints to find new solution in orthogonal direction and force a non-zero component for that solution.
An example

for (t=0; t<T; t++) {
    for (i=2; i<N-1; i++) {
        b[i] = 0.333*(a[i-1] + a[i] + a[i+1]);
    }
}

for (j=2; j<N-1; j++){
    a[j] = b[j];
}

\[
\begin{bmatrix}
1 & 0 \\
2 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
t_i \\
0
\end{bmatrix}
+ 
\begin{bmatrix}
0 \\
0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
2 & 1 \\
0 & 0
\end{bmatrix}
\begin{bmatrix}
t_j \\
0
\end{bmatrix}
+ 
\begin{bmatrix}
0 \\
1
\end{bmatrix}
\]
Next Step, Tiling
Tiling for multiple statements under transformations

specify a modified higher dimensional domain and specify transformations for what would be the tile space loops in the transformed space.
Tiling for multiple statements under transformations

Algorithm 1 Tiling for multiple stmts under transformations

INPUT Hyperplanes (statement-wise) belonging to a tilable band of width 
k: $\phi^0_S, \phi^1_S, \ldots, \phi^{i+k-1}_S$, expressed as affine functions of corresponding original iterators, $\vec{i}_S$; Original domains: $\mathcal{D}_S$; Tile sizes: $\tau_i, \tau_{i+1}, \ldots, \tau_{i+k-1}$

1: /* Update the domains */
2: for each statement $S$ do
3: for each $\phi^j_S = f^j(\vec{i}_S) + f_0$ do
4: Increase the domain ($\mathcal{D}_S$) dimensionality by creating supernodes for all original iterators that appear in $\phi^j_S$
5: Let the supernode iterators be $\vec{\tau}^j$
6: Add the following two constraints to $\mathcal{D}_S$:
7: $\tau_j \bullet f^j(\vec{i}_S) \leq f^j(\vec{i}_S) + f_0^j \leq \tau_j \bullet f^j(\vec{i}_S) + \tau_j - 1$
8: end for
Tiling for multiple statements under transformations

9: /* Update the transformation matrices */
10: for each statement $S$ do
11:     Add $k$ new rows to the transformation of $S$ at level $i$
12:     Add as many columns as the number of supernodes added to $D_{S}$ in Step 4
13:     for each $\phi_{S}^{j} = f^{j}(i_{S}) + f_{0}^{j}, j = i, \ldots, i + k - 1$ do
14:         Add a supernode for this hyperplane: $\phi T_{S}^{j} = f^{j}(i_{\bar{T}_{S}})$
15:     end for
16: end for
OUTPUT Updated domains ($D_{S}$) and transformations
Tiling for multiple statements under transformations

Example: \( c_1 = i \), \( c_2 = i + j \)

<table>
<thead>
<tr>
<th>Domain</th>
<th>Scattering</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 \leq i \leq N - 1 )</td>
<td>( c_{1T} = i_T )</td>
</tr>
<tr>
<td>( 0 \leq j \leq N - 1 )</td>
<td>( c_{2T} = i_T + j_T )</td>
</tr>
<tr>
<td>( 0 \leq i - 32i_T \leq 31 )</td>
<td>( c_1 = i )</td>
</tr>
<tr>
<td>( 0 \leq (i + j) - 32(i_T + j_T) \leq 31 )</td>
<td>( c_2 = i + j )</td>
</tr>
<tr>
<td>( c_{1T}, c_{2T}, c_1, c_2 )</td>
<td>( \leftarrow \text{scatter}(i_T, j_T, i, j) )</td>
</tr>
</tbody>
</table>
Algorithm 2 Tiled pipelined parallel code generation

INPUT Given that Algorithm 1 has been applied, a set of $k$ (statement-wise) supernodes in the transformed space belonging to a tilable band: $\phi T^1_S, \phi T^2_S, \ldots, \phi T^k_S$

1: To extract $m$ ($< k$) degrees of pipelined parallelism:
2: /* Update transformation matrices */
3: for each statement $S$ do
4: Perform the following unimodular transformation on only the scattering supernodes: $\phi T^1 \rightarrow \phi T^1 + \phi T^2 + \ldots + \phi T^{m+1}$
5: Mark $\phi T^2, \phi T^3, \ldots, \phi T^{m+1}$ as parallel
6: Leave $\phi T^1, \phi T^{m+2}, \ldots, \phi T^k$ as sequential
7: end for

OUTPUT Updated transformation matrices/scatterings
Tiled pipelined parallel code generation

Why we can do the sum in line 4?

The sum of rows in transformation satisfies all affine dependences that satisfied that certain, and gives a legal schedule of tiles.
Tiled pipelined parallel code generation

Example:

Before parallelized, just locality tiled

\[
\begin{bmatrix}
c_1 \\
c_2 \\
c_3 \\
c_4 \\
c_5 \\
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \begin{bmatrix}
t_T \\
_i_T \\
t \\
_i \\
1 \\
\end{bmatrix}
\]

After parallelized, with locality tiled

\[
\begin{bmatrix}
c_1 \\
c_2 \\
c_3 \\
c_4 \\
c_5 \\
\end{bmatrix} = \begin{bmatrix}
3 & 1 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix} \begin{bmatrix}
t_T \\
_i_T \\
t \\
_i \\
1 \\
\end{bmatrix}
\]
The Whole Process - LU decomposition

```
for (k=0; k<N; k++)
    for (j=k+1; j<N; j++)
        a[k][j] = a[k][j] / a[k][k];

for (i=k+1; i<N; i++)
    { for (j=k+1; j<N; j++)
        a[i][j] = a[i][j] - a[i][k] * a[k][j];
    }
```

---

Figure 9. LU decomposition
First Step, Find the Transformation

\begin{align*}
\text{(a) Original code} \\
\text{for } (k=0; k<N; k++) \\
\quad \text{for } (j=k+1; j<N; j++) \\
\quad \quad a[k][j] = a[k][j]/a[k][k] \\
\text{for } (i=k+1; i<N; i++) \\
\quad \quad \text{for } (j=k+1; j<N; j++) \\
\quad \quad \quad a[i][j] = a[i][j] - a[i][k]*a[k][j] \\
\end{align*}

S1: \[
\begin{bmatrix}
c_1 \\ c_2 \\ c_3 
\end{bmatrix} = 
\begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
k \\ j 
\end{bmatrix}
\]

S2: \[
\begin{bmatrix}
c_1 \\ c_2 \\ c_3 
\end{bmatrix} = 
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
k \\ i \\ j 
\end{bmatrix}
\]
Second Step, Tiles under Transformation

<table>
<thead>
<tr>
<th>Domains</th>
<th>Scatterings</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S1 )</td>
<td>( S1 )</td>
</tr>
<tr>
<td>( 0 \leq k \leq N - 1 )</td>
<td>( c_1 = k )</td>
</tr>
<tr>
<td>( k + 1 \leq j \leq N - 1 )</td>
<td>( c_2 = j )</td>
</tr>
<tr>
<td>0 \leq k - 32k_T \leq 31</td>
<td>( c_3 = k )</td>
</tr>
<tr>
<td>0 \leq j - 32j_T \leq 31</td>
<td>(( c_1, c_2, c_3, c_1, c_2, c_3 ))</td>
</tr>
<tr>
<td></td>
<td>( \leftarrow \text{scatter}(k_T, j_T, k, j) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( S2 )</th>
<th>( S2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 \leq k \leq N - 1 )</td>
<td>( c_1 = k )</td>
</tr>
<tr>
<td>( k + 1 \leq i \leq N - 1 )</td>
<td>( c_2 = j )</td>
</tr>
<tr>
<td>( k + 1 \leq j \leq N - 1 )</td>
<td>( c_3 = i )</td>
</tr>
<tr>
<td>0 \leq k - 32k_T \leq 31</td>
<td>(( c_1, c_2, c_3, c_1, c_2, c_3 ))</td>
</tr>
<tr>
<td>0 \leq i - 32i_T \leq 31</td>
<td>( \leftarrow \text{scatter}(k_T, j_T, i_T, k, j, i) )</td>
</tr>
</tbody>
</table>
Third Step, Generate Parallel Transformation
Final Step, Code Generation

```c
#define S1(z,T0,T1,k,j) {a[k][j]=a[k][j]/a[k][k];}
#define S2(z,T0,T1,T2,k,j) {a[i][j]=a[i][j] - a[i][k]*a[k][j];}

/* Generated by CLoopG v0.14.1 64 bits in 0.02s. */
for (c1=1; c1<=floord(2*N-3,32); c1++)
  lb = max(max(ceild(16*c1-15,32),ceild(32*c1-N+2,32)),0);
  ub = min(floord(32*c1+31,32), floord(N-1,32));
#pragma omp parallel for shared(c1,lb,ub,a) private (c2,c3,c4,c5,c6,i,j,k,l,m,n)
for (c2=lb; c2<=ub; c2++)
  for (c3=max(ceild(16*c1-16*c2-465,496),ceild(16*c1-16*c2-15,16)); c3<=floord(N-1,32); c3++)
    if (c1 <= c2+c3) {
      for (c4=max(0,32*c3); c4<=min(min(32*c3+30,N-2),32*c2+30); c4++)
        for (c5=max(32*c2,c4+1); c5<=min(N-1,32*c2+31); c5++)
          S1(c1-c2,c1,c4,c5);
      for (c6=c4+1; c6<=min(32*c3+31,N-1); c6++)
        S2(c1-c2,c1,c2,c4,c6,c5);
    }
  }
for (c4=max(0,32*c1-32*c2); c4<=min(min(32*c1-32*c2+31,32*c1-31),32*c2+30); c4++)
  for (c5=max(32*c2,c4+1); c5<=min(N-1,32*c2+31); c5++)
    for (c6=32*c3; c6<=min(32*c3+31,N-1); c6++)
      S2(c1-c2,c1,c2,c4,c6,c5);
if ((c1 == -c2-c3) && (c1 <= min(floord(32*c2+N-33,32), floord(64*c2-1,32)))) {
  for (c5=max(32*c1-32*c2+32,c1-32*c2+31); c5<=min(32*c2+31,N-1); c5++)
    S1(c1-c2,c2+32,c1-32*c2+31,c5);
}

(c) LU (1-d pipelined parallel + L1 tiled) (tile size 32) cloog -f 4 -17
```
Experimental Evaluation

Figure 6. Imperfectly nested Jacobi stencil
Experimental Evaluation

Figure 10. LU performance

(a) Single core (L1 and L2 tiled)

(b) On a quad core: N=8000
Analysis

Significant Speedup: ranging from 2x to 5x are obtained over previous automatic transformation approaches.

Linear to super-linear speedups are seen for almost all compute-intensive kernels due to optimization for locality as well as parallelism.
Thank you!