Optimizing Data Locality

Presented by Chris Frisz and Janhavi Virkar
Motivation

- Memory access is slow
- Speed up computations by keeping data in cache...
  - ...but cache is small
- Utilize analysis and transformations of loops to fit data into cache appropriately
Overview

- Represent iteration space as dependence vector space
- Dependence Vectors

\[
\text{for } I_1 := 0 \text{ to } 5 \text{ do } \\
\text{for } I_2 := 0 \text{ to } 6 \text{ do } \\
D = \{(0,1), (1,0), (1,-1)\}. \]
Background

- Unimodular Transformations
  - Loops transformations represented as matrix transformations
  - Example Loop interchange

\[
\begin{bmatrix}
  0 & 1 \\
  1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  d_1 \\
  d_2 \\
\end{bmatrix}
= 
\begin{bmatrix}
  d_2 \\
  d_1 \\
\end{bmatrix}.
\]
Loop Transformation Theory

- The same transformation matrix is used to verify that a transformation is legal using dependence vectors
  - Since iteration space and dependence space are the same dimension
Theorem 2.1. Let $D$ be the set of distance vectors of a loop nest. A unimodular transformation $T$ is legal if and only if $\forall \vec{d} \in D : T\vec{d} \geq 0$. 
Fully Permutable

- Loop nests to which any arbitrary loop permutation that would make the transformed dependences lexicographically positive are called fully permutable
- Important property for tiling
Localized vector space

- Localized iteration space
  - Part of the iteration space which carries reuse

- Localized vector space
  - Abstraction of the localized iteration space
  - Given in terms of a span of vectors
Reuse vs. Locality

- Reuse inherent in computation
- Locality is obtained by exploiting reuse
Indexing function and uniformly generated sets


- Multiple memory references form a uniformly generated set if the locations they access only differ by a constant term

Definition 4.1 Let $n$ be the depth of a loop nest, and $d$ be the dimensions of an array $A$. Two references $A[\bar{f}(i)]$ and $A[\bar{g}(i)]$, where $\bar{f}$ and $\bar{g}$ are indexing functions $\mathbb{Z}^n \rightarrow \mathbb{Z}^d$, are called uniformly generated if

$$\bar{f}(i) = Hi + \bar{c}_f \text{ and } \bar{g}(i) = Hi + \bar{c}_g$$

where $H$ is a linear transformation and $\bar{c}_f$ and $\bar{c}_g$ are constant vectors.
Linear algebra background

- Matrices as linear transformations
  - Representing loop transformations

- Kernel of a matrix
  - The set of vectors whose product with the matrix is the zero vector
    - This is a subspace
  - We'll use it for quantifying reuse
Linear algebra background (cont.)

- Span of a set of vectors – \( \text{span}\{v_1, ..., v_k\} \)
  - Represents a subspace consisting of a linear combination of the given vectors
    - Example: two-dimensional Cartesian space
    - We use it to represent dependence vector spaces and quantify reuse

- Dimension of a subspace – \( \text{dim}(S) \)
  - The minimum number of linearly independent vectors in a span for a subspace
  - We use it to give the number of loops carrying reuse.
Evaluating reuse and locality

- Types of reuse
  - Self-temporal: $R_{st}$
  - Self-spatial: $R_{ss}$
  - Group-temporal: $R_{gt}$
  - Group-spatial: $R_{gs}$
Self-temporal reuse

- Accesses to the same location by one memory reference
- For memory reference $A[v] = A[H*i + c]$, self-temporal reuse is given by $\ker(H)$
- That is, all iteration vectors whose product with $H$ is the zero vector are iterations with self-temporal reuse
- $R_{st} = \ker(H)$
**Self-spatial reuse**

- Accesses along the same row of memory by a single memory reference
- \( R_{ss} = \ker(H_s) \) where \( H_s \) is \( H \) with the bottom row zeroed out
  - Zeroing out the bottom row of \( H \) treats a memory row as the same location
- \( R_{ss} \) is a superset of \( R_{st} \)
Group–temporal reuse

- Accesses to the same location by multiple memory references
- Find a particular solution to the equation $H*r_i = c_1 - c_2$ for all pairs of memory references in a uniformly generated set
- $R_{gt} = \text{span}\{r_2, ..., r_g\} + \ker(H)$ for $r_k = c_1 - c_k$
Group-spatial reuse

- Accesses to the same memory row by multiple memory references
- Find a particular solution to the equation $H \cdot r_i = c_{s,1} - c_{s,2}$ for all pairs of memory references where $c_{s,i}$ is $c_i$ with the last element set to zero
- $R_{gs} = \text{span}\{r_2, \ldots, r_g\} + \ker(H_s)$
- $R_{gs}$ is a superset of $R_{gt}$
Reference equivalence classes

- Two memory references \( A[H*i + c_1] \) and \( A[H*i + c_2] \) are in the same temporal equivalence class if there exists some \( r \) such that \( H*r = c_1 - c_2 \)
  - The number of equivalence classes is denoted by \( g_t \)

- Two memory references \( A[H*i + c_1] \) and \( A[H*i + c_2] \) are in the same spatial equivalence class if and only if there exists some \( r \) such that \( H_s * r = c_{s,1} - c_{s,2} \)
  - The number of equivalence sets is denoted by \( g_s \)
Combining reuses

- Calculate the number of memory accesses per iteration by the equation:

\[
\frac{g_S + \frac{g_T - g_S}{l}}{l e_{\text{sdim}}(R_{SS} \cap L)}
\]

where

\[
e = \begin{cases} 
0 & R_{ST} \cap L = R_{SS} \cap L \\
1 & \text{otherwise.}
\end{cases}
\]
The data locality optimization problem

Definition 4.2 For a given iteration space with

1. a set of dependence vectors, and

2. uniformly generated reference sets

the data locality optimization problem is to find the uni-modal and/or tiling transform, subject to data dependences, that minimizes the number of memory accesses per iteration.
An algorithm

- Overview: Uses unimodular transformations and reuse quantification on a loop nest to create a fully permutable loop nest that can be tiled to optimize locality
- Step 1: Identify loops that will stay outermost
- Step 2: Identify a subset of the remaining loops which will minimize memory accesses when placed innermost and tiled
- Step 3: Recursively apply skewing, reversal, and permutation transformations to the rest of the loop nest while satisfying dependences until it is fully permutable
An algorithm (cont.)

- Performance is $O(n^2 \times d)$ for $n$ number of loops in the nest and $d$ the number of dependences
- Simplified by excluding loops that carry no reuse or must be placed outermost due to legality.
- Run quickly when there is little reuse or many dependences limit the possible loop permutations.
Benchmarks

Figure 5: Performance of 500 × 500 double precision LU factorization without pivoting on the SGI 4D/380. No register tiling was performed.

Figure 6: Behavior of 30 iterations of a 500 × 500 double precision SOR step on the SGI 4D/380. The tile sizes are 64 × 64 iterations. No register tiling was performed.
Further work

- SPIRAL framework implements tiling optimization transformations along with domain-specific optimizations
- Extending the number of permutable loops

Table 1: Execution Time (in Seconds) of Different Versions of Jacobi

<table>
<thead>
<tr>
<th>Different Scheme</th>
<th>Matrix Size for R5K</th>
<th>Matrix Size for R10K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>869</td>
<td>1024</td>
</tr>
<tr>
<td>Original</td>
<td>Time</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.00</td>
</tr>
<tr>
<td>Peel-and-fusion</td>
<td>Time</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.24</td>
</tr>
<tr>
<td>Tiling w/ Array Dup.</td>
<td>Time</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.84</td>
</tr>
</tbody>
</table>
References


