# *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

```
for I1 := 0 to 5 do
  for I2 := 0 to 6 do
    A[I2+1] := 1/3 * (A[I2] + A[I2+1] + A[I2+2]);
```

$$D = \{(0,1),(1,0),(1,-1)\}.$$

# *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

# Unimodular Transformation Theorem

**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* $\forall \vec{d} \in D : T\vec{d} \succ \vec{0}$.

# *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*

- Represent memory reference A[v] = A[H*i + c] (an indexing function)
- 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[$\vec{f}(\vec{\imath})$] *and* A[$\vec{g}(\vec{\imath})$], *where $\vec{f}$ and $\vec{g}$ are indexing functions $Z^n \rightarrow Z^d$, are called* uniformly generated *if*

$$\vec{f}(\vec{\imath}) = H\vec{\imath} + \vec{c}_f \text{ and } \vec{g}(\vec{\imath}) = H\vec{\imath} + \vec{c}_g$$

*where $H$ is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ 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 – span$\{v_1, \ldots, v_k\}$
  - 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 – 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
- Consider two memory references $A[v] = A[H*i + c_1]$ and $A[u] = A[H*i + c_2]$
- 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
- $R_{gt} = \text{span}\{r_2, \ldots, r_g\} + \ker(H)$ 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 $H*r_i = c_{s,1} - c_{s,2}$ 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:

$$\frac{g_S + (g_T - g_S)/l}{l^e\, S^{\dim(R_{SS} \cap L)}}$$

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 unimodular 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 $O(n^2 * d)$ 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

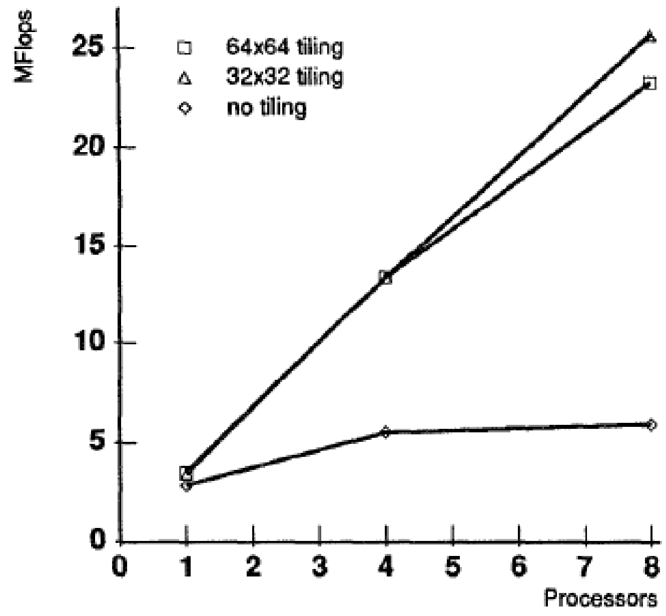


Figure 5: Performance of $500 \times 500$ double precision LU factorization without pivoting on the SGI 4D/380. No register tiling was performed.

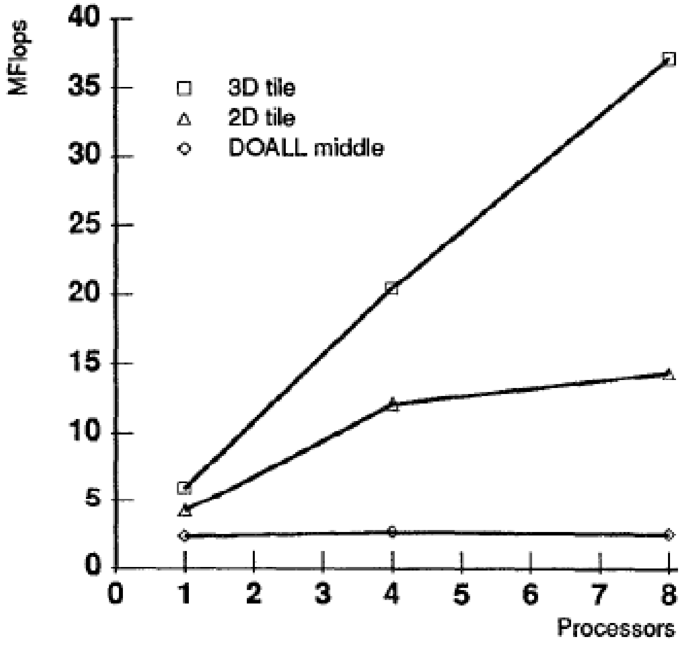


Figure 6: Behavior of 30 iterations of a $500 \times 500$ double precision SOR step on the SGI 4D/380. The tile sizes are $64 \times 64$ iterations. No register tiling was performed.

# 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

| Different Scheme | | Matrix Size for R5K | | | Matrix Size for R10K | | |
|---|---|---|---|---|---|---|---|
| | | 869 | 1024 | 1279 | 869 | 1024 | 1279 |
| Original | Time | 46 | 68 | 102 | 24.43 | 44.65 | 52.93 |
| | Speedup | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Peel-and-fusion | Time | 37 | 50 | 82 | 17.08 | 27.23 | 35.76 |
| | Speedup | 1.24 | 1.36 | 1.24 | 1.43 | 1.64 | 1.48 |
| Tiling w/ Array Dup. | Time | 25 | 38 | 60 | 9.25 | 17.23 | 20.14 |
| | Speedup | 1.84 | 1.79 | 1.70 | 2.64 | 2.59 | 2.63 |

# References


- Allen, Randy and K. Kennedy. Optimizing Compilers for Modern Architectures. Morgan Kaufman: San Francisco, 2002.
- M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI), 1991, pp. 30–44.
- Song, Y. and Li, Z. New tiling techniques to improve cache temporal locality. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1999.
- M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," Proc. IEEE, vol. 93, no. 2, pp. 232–275, Feb. 2005.