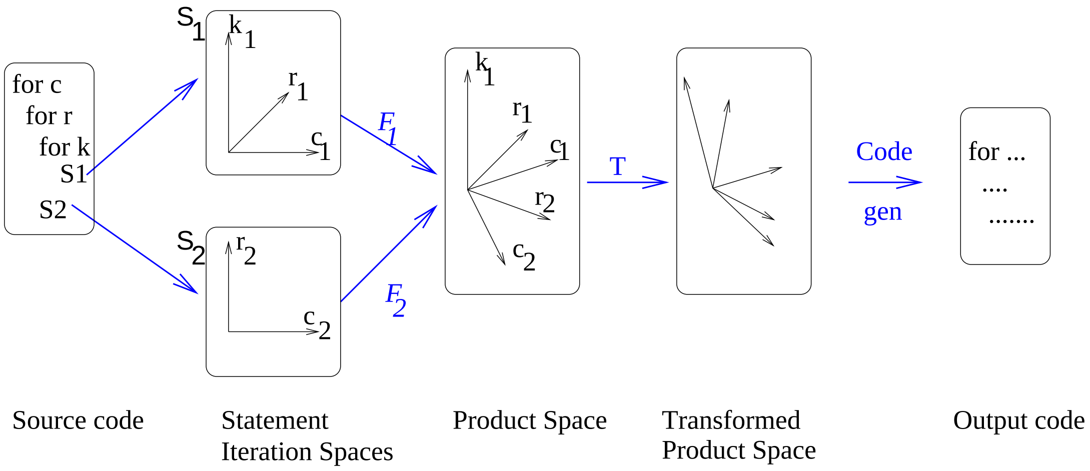
Locality Enhancement for Imperfectly-nested Loops

General approach



- Each statement has a statement iteration space.
- Product space: Cartesian product of individual stater interation spaces.
- Each statement iteration space is embedded into prod using affine embedding functions F_i.
- Product space is transformed using linear loop transfo to enhance locality.
- Code is produced to scan points in final space.

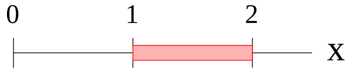
Key result required to compute embeddings:

Farkas's Lemma: Any affine function $f(x)$ which is non-n
everywhere in a polyhedron $Ax + b \geq 0$ can be represente
follows:

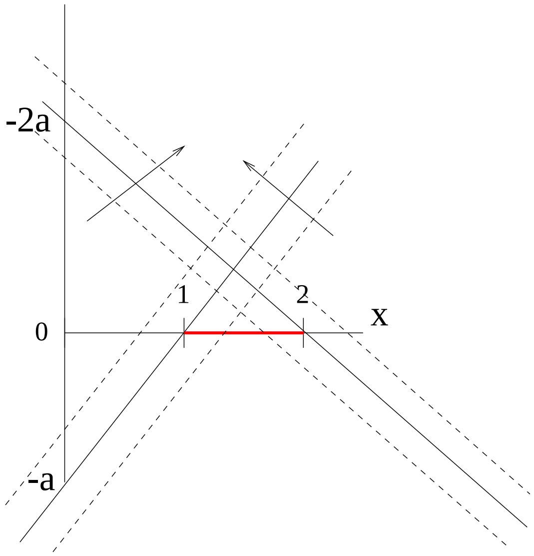
$$f(x) = \lambda_0 + \Lambda^T(Ax + b) \text{ where } \lambda_0 \geq 0, \Lambda \geq 0$$

In words: any function that is positive everywhere in a po
$Ax + b \geq 0$ can be expressed as a positive linear combinat
rows of the vector $Ax + b$.

Example for Farkas's Lemma: Let $f(x) = ax + b$ be non-n
domain



What are constraints on a and b?



It is easy to see geometrically that
if a is +ve, then $b \geq -a$
if a is -ve, then $b \geq -2a$

How do we deduce this algebraically?

Domain:

$x - 1 \geq 0$

$2 - x \geq 0$

Function: $f(x) = ax + b$

From Farkas's lemma, we can write

$f(x) = \lambda_0 + \lambda_1(x-1) + \lambda_2(2-x)$

Equating coefficients for the two expressions for f, we see

$\lambda_0 - \lambda_1 + 2\lambda_2 = b$

$\lambda_1 - \lambda_2 = a$

$\lambda_0 \geq 0$

$\lambda_1 \geq 0$

$\lambda_2 \geq 0$

Use Fourier-Motzkin elimination to eliminate λ's from sys

$\lambda_0 - \lambda_1 + 2\lambda_2 = b$

$\lambda_1 - \lambda_2 = a$

$\lambda_0 \geq 0$

$\lambda_1 \geq 0$

$\lambda_2 \geq 0$

to get

$(b + a) \geq max(-a, 0)$

which is equivalent to what we determined geometrically.

Determining embeddings for legality

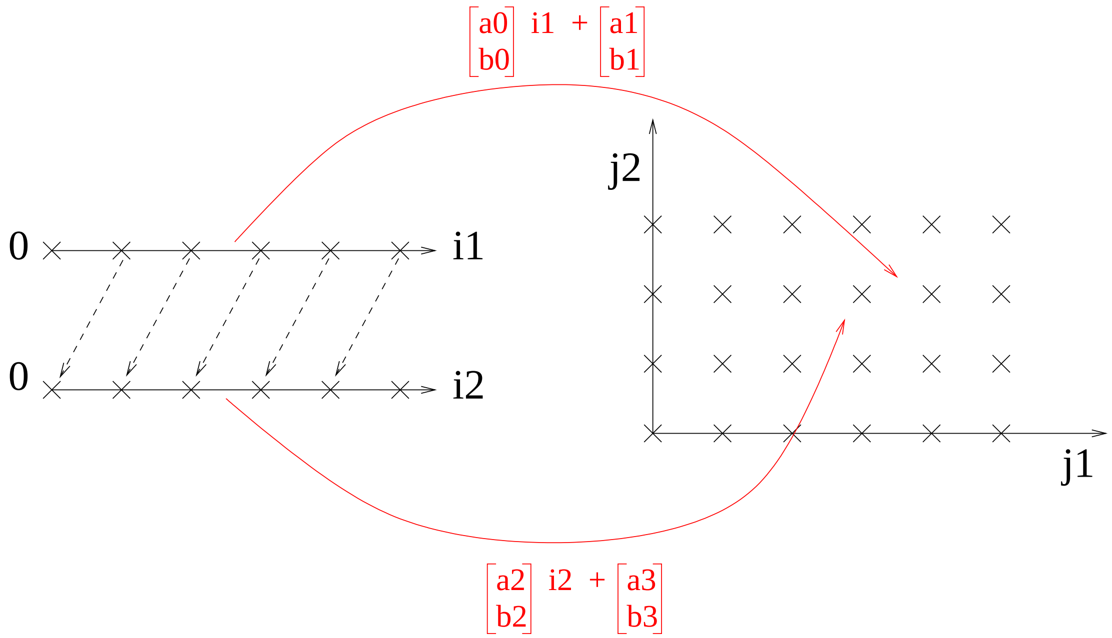
Let us consider a simpler problem than locality enhancem

Given an imperfectly nested loop,

find embeddings into product space to generate a legal pr
(lexicographic order of execution in product space is legal

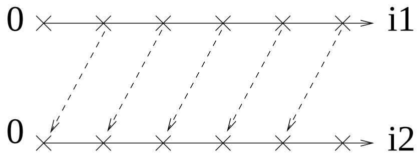
Example for embeddings:

```
S1: for i1 = 0, N-1           for j1 = -inf, inf
      A(i1) = ....             for j2 = -inf, inf
                          =>    if (S1(m) is mapped to
S2: for i2 = 0, N-1                execute S1(m);
      sum = sum + A(i2+1)       if (S2(n) is mapped to
                                   execute S2(n);
```



Dependence polyhedron:

```
S1: for i1 = 0, N-1
      A(i1) = ....
S2: for i2 = 0, N-1
      sum = sum + A(i2+1)
```



$i1 = i2 + 1$

$0 \leq i1 \leq N - 1$

$0 \leq i2 \leq N - 1$

which can be written as follows:

$$\begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} i1 \\ i2 \\ N \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \\ 0 \\ -1 \\ 0 \\ -1 \end{pmatrix} \geq 0$$

Let us write this as:

$$D \begin{pmatrix} i1 \\ i2 \\ N \end{pmatrix} + \underline{d} \geq 0$$

Constraint on embeddings:

$$\begin{pmatrix} a0 \\ b0 \end{pmatrix} i1 + \begin{pmatrix} a1 \\ b1 \end{pmatrix} \preceq \begin{pmatrix} a2 \\ b2 \end{pmatrix} i2 + \begin{pmatrix} a3 \\ b3 \end{pmatrix}$$

Let us first determine embeddings that make the first dim
difference vector between dependent iterations +ve.

We want $f(x) = a2 * i2 + a3 - a0 * i1 - a1 > 0$

which can be written as $f(x) = a2 * i2 + a3 - a0 * i1 - a1$

Using Farkas's lemma, we can write this as follows:

$$f(x) = a2 * i2 + a3 - a0 * i1 - a1 - 1 \geq 0 - - - - - - (1)$$

$$f(x) = \lambda_0 + \Lambda^T (D * \begin{pmatrix} i1 \\ i2 \\ N \end{pmatrix} + \underline{d}) - - - - - - (2)$$

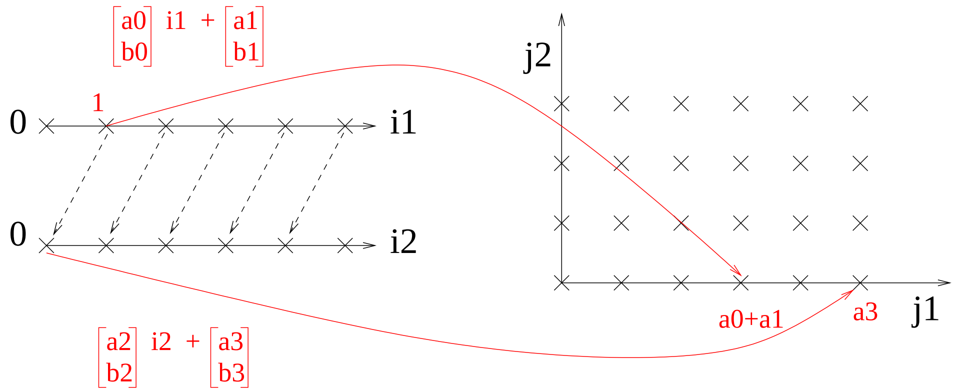
Equating coefficients, we get:

$$\begin{aligned} -a0 &= -\lambda_1 + \lambda_2 + \lambda_3 - \lambda_4 \\ a2 &= \lambda_1 - \lambda_2 + \lambda_5 - \lambda_6 \\ \lambda_4 + \lambda_6 &= 0 \\ a3 - a1 &= \lambda_0 + \lambda_1 - \lambda_2 - \lambda_4 - \lambda_6 \end{aligned}$$

Using Fourier-Motzkin elimination to eliminate the λ's, w
following constraints on the coefficients of the embedding

$$a0 + a1 < a3$$
$$a0 \leq a2$$

Let us see geometrically why this makes sense!



$$a0 + a1 < a3$$

$$a0 \leq a2$$

First constraint ensures that first two dependent points ar
correct order.

Second constraint ensures that "jumps" to successive poi
always larger for S2 than S1.

Other choices for embedding functions:

$$\begin{pmatrix} a0 \\ b0 \end{pmatrix} i1 + \begin{pmatrix} a1 \\ b1 \end{pmatrix} \preceq \begin{pmatrix} a2 \\ b2 \end{pmatrix} i2 + \begin{pmatrix} a3 \\ b3 \end{pmatrix}$$

- Make first dimension of difference vector $= 0$ within dependence polyhedron
  $a0 * i1 + a1 = a2 * i2 + a3$
  This can be expressed as two inequalities, and two ap
  of Farkas's lemma gives $a0 + a1 = a3, a0 = a2$.
- Make second dimension of difference vector positive w
  dependence polyhedron
  $g(x) = (b2 * i2 + b3) - (b0 * i1 + b1) > 0$
  This gives $b0 + b1 < b3, b0 \leq b2$.

Complete solution for legal embeddings:

- Dependence vector of form $(+, *)^{\mathrm{T}}$

$$a0 + a1 < a3$$
$$a0 \leq a2$$

- Dependence vector of form $(0, +)^{\mathrm{T}}$

$$a0 + a1 = a3$$
$$a0 = a2$$
$$b0 + b1 < b3$$
$$b0 \leq b2$$

- We can also get a dependence vector of form $(0, 0)^{\mathrm{T}}$

General picture for determining embeddings into product

Statement iteration spaces: $I_1, I_2, ..., I_n$

Product space: $I_1 \times I_2 ... \times I_n$

Embeddings: $F_1, F_2, ..., F_n$

Dependence polyhedra: $(D_1, d_1), (D_2, d_2), ..., (D_k, d_k)$

Legality:

$\forall (D_m, d_m) \forall (i_j \rightarrow i_k) \in (D_m, d_m) . F_k(i_k) - F_j(i_j) \succeq 0$

Solving for embeddings: solve for each dimension of P

1. first dimension: all difference vector entries must be p zero
2. remaining dimensions: satisfied dependences can be d

Small caveat: we want to avoid a solution in which all sta
instances get mapped to a single point in product space!!

This is a trivial solution and is not very useful.

Our solution:

Restrict F_i to act like the identity in the subspace I_i of th
space.

There may be other ways to solve this problem, but this s
work fine in practice.

Determining embeddings that promote locality:

Reuse polyhedra: formulate similar to dependence polyhe

One strategy:

- find legal embeddings F_i
- for each legal embedding, find best transformation T
- pick best one

Too many possibilities....

One approach: starting with first dimension, determine en dimension by dimension, choosing embeddings for a dimer before going on to next one.

Seems to work fine in practice, but in principle, it may fai legal embeddings....

Sketch of greedy algorithm: [Ahmed,Mateev,Pingali (ICS'

Go dimension by dimension trying to

- height reduce reuse classes
- make entries of dependence vectors positive to enable

To avoid combinatorial explosion, pick embeddings for eac
dimension before looking at succeeding dimensions.

```
DU = set of all unsatisfied dependence classes;
DS = set of all satisfied dependence classes;
RS = ordered set of all reuse classes sorted by priority;
for each dimension p of product space P do
  { L = Legality Constraints(p,DU,DS);
    if system L has solutions
     {Embedding coefficients for dimension p =
        PromoteReuse(p,L,RS);
      Update DS and DU;
     }
    else abort;
  }
```

```
ALGORITHM LegalityConstraints($q$, $DU$, $DS$ ) {
 /*
   $q$ is dimension being processed.
   $DU$ is set of unsatisfied dependence classes.
   $DS$ is set of satisfied dependence classes.
 */

  Construct system $Temp$ constraining the $qth$ dimension
    of every embedding function as follows:

  for each unsatisfied dependence class $u \in DU$
     Add constraints so that each entry in dimension $q$
       of all difference vectors of $u$ is non-negative;
  //enable tiling by considering satisfied dependence classes as wel
  for each satisfied dependence class $s \in DS$
     Add constraints so that each entry in dimension $q$
       of all difference vectors of $s$ is non-negative;

  Use Farkas' lemma to convert system $Temp$ into
    a system $L$ constraining unknown embedding
    coefficients;

  Return L; }
```

```
ALGORITHM PromoteReuse($q$,$L$,$RS$) {
 /*
   $q$ is dimension being processed.
   $L$ is a system constraining unknown embedding coefficients.
   $RS$ is set of prioritized reuse classes.
 */


$L'$:= $L$
for every reuse class $R$ in $RS$ in priority order
 {
    Z := System constraining unknown embedding function
         coefficients so $qth$ dimension entries of
         all reuse vectors of class $R$ is zero

    if ($L' \cap Z \neq \emptyset$)
     {
         $L'$ := $L' \cap Z$
     }
 }
 return any set of coefficients satisfying $L'$;
}
```

Limitation of this algorithm: does not consider T, so we c
consider illegal embeddings that can be "fixed" by choosin
appropriately.

Solution: determine T and embeddings simultaneously.

See paper for details.

Next slide shows the kind of modifications that need to b

Modification to permit skewing T:

```
ALGORITHM LegalityConstraints($q$, $DU$, $DS$ ) {
 /*
   $q$ is dimension being processed.
   $DU$ is set of unsatisfied dependence classes.
   $DS$ is set of satisfied dependence classes.
 */
  Construct system $Temp$ constraining the $qth$ dimension
    of every embedding function as follows:

  for each unsatisfied dependence class $u \in DU$
     Add constraints so that each entry in dimension $q$
       of all difference vectors of $u$ is non-negative;
  //enable tiling by considering satisfied dependence classes as wel
  //permit skewing to enable tiling
  for each satisfied dependence class $s \in DS$
     Add constraints so that each entry in dimension $q$
       of all difference vectors of $s$ + positive $\alpha$
       is non-negative; //skewing later will eliminate -ve entries

  Use Farkas' lemma to convert system $Temp$ into
    a system $L$ constraining unknown embedding
    coefficients;
  Return L; }
```

```
ALGORITHM LocalityEnhancement {
$Q$  := Set of dimensions of product space;
$DU$ := Set of unsatisfied dependence classes
             (initialized to all dependence classes);
$DS$ := Set of satisfied dependence classes
             (initialized to empty set);
$RS$ := Set of reuse classes of the program
             (sorted by priority);
$j$  := Current dimension in transformed product space
             (initialized to 1);
while ($Q$ is non-empty)
 {for each $q$ in $Q$
    {$L$ = LegalityConstraints($q$, $DU$, $DS$);
       if system $L$ has solutions
        {   Embedding coefficients for dimension j =
               PromoteReuse($q$,$L$,$RS$);
            Update $DS$ and $DU$;
            Delete $q$ from $Q$;
            j = j + 1;
        }
     }
   // No more dimensions $q$ can be added to current band.
   // Start a new band of fully permutable loops.
   $DS$ := empty set;  }
```

```
Apply Algorithm  DimensionOrdering to the dimensions;
Eliminate redundant dimensions;
Tile permutable dimensions with non-zero $ReusePenalty$;
}
```

```
ALGORITHM DimensionOrdering

$RPO$  =  $\{i1,\ i2,\ \ldots\ ip\}$  // $ReusePenalty$ order
$NRPO$ =  $\emptyset$                // nearby permutation

$m$ = $p$  // number of dimensions left to process
$k$ = $0$  // number of dimensions processed
while  $RPO \neq \emptyset$
 {
    for dimension $j$ = 1,$m$
     {
        $l$ = $ij \in RPO$
        Let $NRPO$ = $\{i1',\ i2',\ \ldots\ ik'\}$
        if $\{i1',\ i2',\ \ldots\ ik',\ l\}$ is legal
         {  $NRPO$ = $\{i1',\ i2',\ \ldots\ ik',\ l\}$
            $RPO$  = $RPO - \{l\}$
            $m$ = $m - 1$
            $k$ = $k + 1$
            continue while loop
         }
     }
 }
```

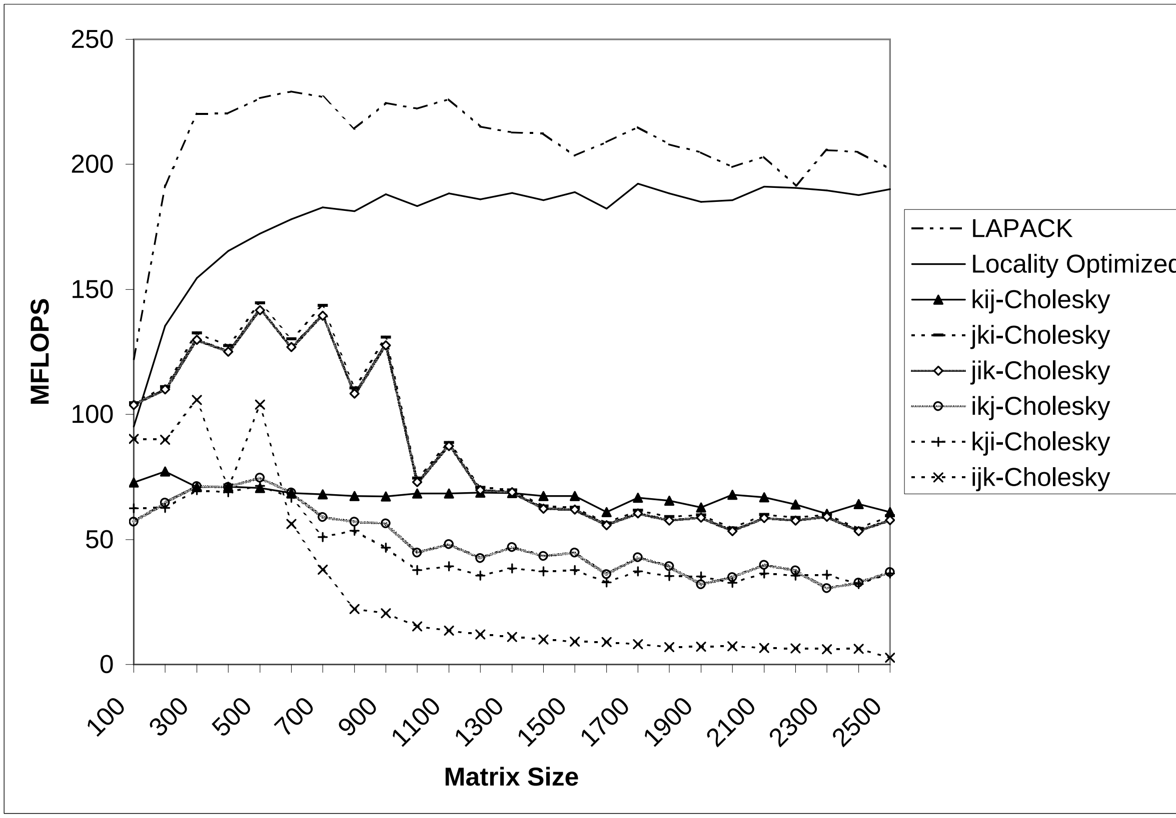
Eliminating redundant dimensions:

$\mathcal{P}$: a Cartesian space

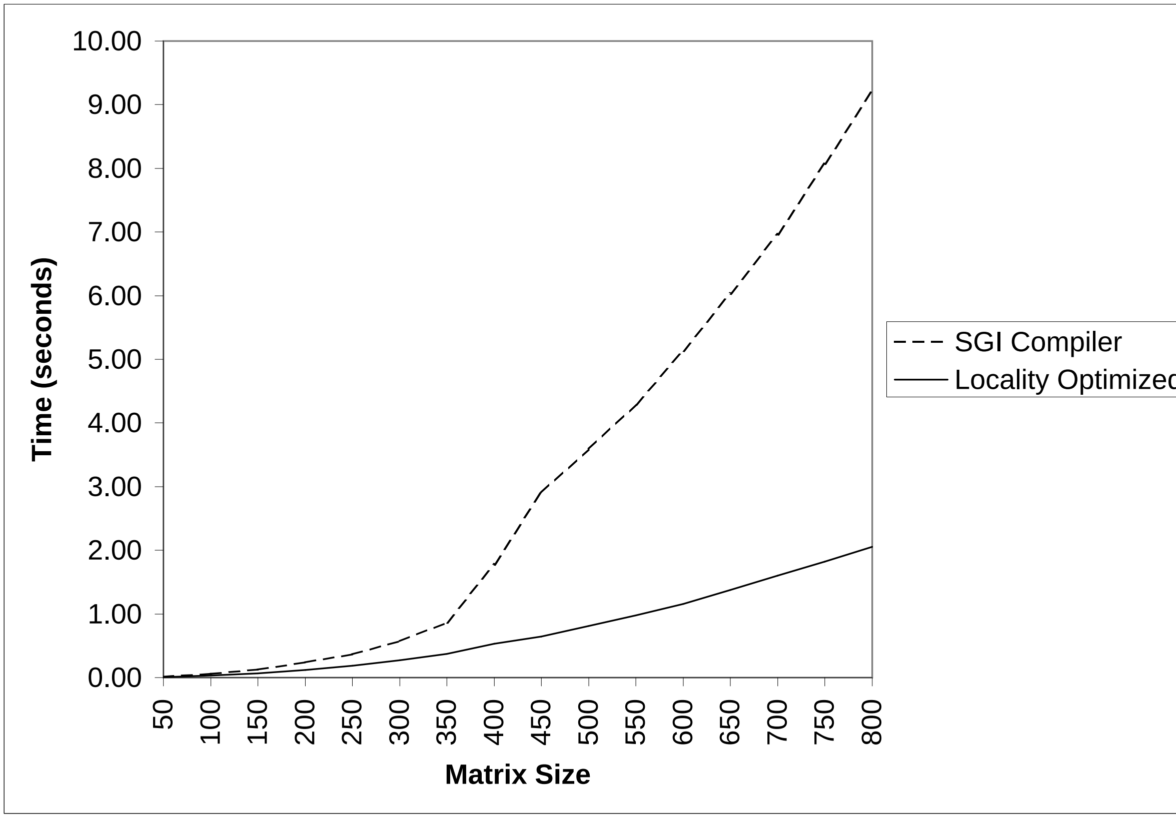
$\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$: a set of affine embedding functions
$F_k(\vec{\imath}_k) = G_k\vec{\imath}_k + g_k$.

Number of independent dimensions of $\mathcal{P}$ =
number of independent rows of matrix $G = [G_1 G_2 \ldots G_n]$

Experimental results:



Cholesky factorization on SGI Octane



Jacobi on SGI Octane

Summary

- We have seen a polyhedral framework for imperfectly- loop transformations.
- We can do a reasonable job of locality enhancement i
  - BLAS: inner product, MVM, MMM, triangular so
  - Cholesky factorization
  - Relaxation codes like Jacobi and Gauss-Seidel
- Accurate tile size determination is a problem. Empirical optimization might be one solution.
- Block-recursive codes: we can generate block-recursiv from iterative codes using this approach.
- Is product space tractable for large programs? Perhaps we can treat basic blocks as single statement

- LU factorization with pivoting: requires fractal symbo
  analysis
- QR factorization: not clear what a compiler can do