Compiling For High Performance

Issues

- parallelism
- locality

Classical compilation

- focus on individual operations
- scalar variables
- flow of values
- unstructured code

High-performance compilation

- focus on aggregate operations (loops)
- array variables
- memory access patterns
- structured code

Issues

- dependence analysis
- loop transformations

Data Locality

Why locality?

- memory accesses are expensive
- exploit higher levels of memory hierarchy by reusing registers, cache lines, TLB, etc.
- locality of reference ⇔ reuse

Locality

- temporal locality *reuse of a specific location*
- spatial locality *reuse of adjacent locations (cache lines, pages)*

Reuse

- self-reuse *caused by same reference*
- group-reuse *caused by multiple references*

What reuse occurs in this loop nest?

```
do i = 1, N
  do j = 1, N
      A(i) += B(j) + B(j+2)
```

Refs	Reuse on loop i	Reuse on loop j
A	spatial	temporal
B	temporal, group spatial	spatial, group temporal

Loop Transformations to Improve Reuse

To calculate *temporal* and *spatial* reuse

For each loop l in a nest, consider l innermost

1. partition references with group-reuse
   $\Rightarrow$ reference groups
2. compute the cost in cache lines accessed
   $\Rightarrow$ loop cost
3. rank the loops based on their *loop cost*
   $\Rightarrow$ memory order

Key insight

> *If loop l promotes more reuse than loop k at the innermost position, then it probably promotes more reuse at any outer position*

Selecting a loop permutation

- select *memory order* if legal
- if not, find a nearby legal permutation
- avoids evaluating many permutations

Selecting a Loop Permutation

Cost of reference group for loop k

1. select representative from reference group
2. find cost (in cache lines) with k innermost
3. multiply by trip counts of outer loops

invariant	1
unit-stride	$(U_k - L_k + 1)/cls$
otherwise	$U_k - L_k + 1$

Loop cost = sum of costs for reference groups

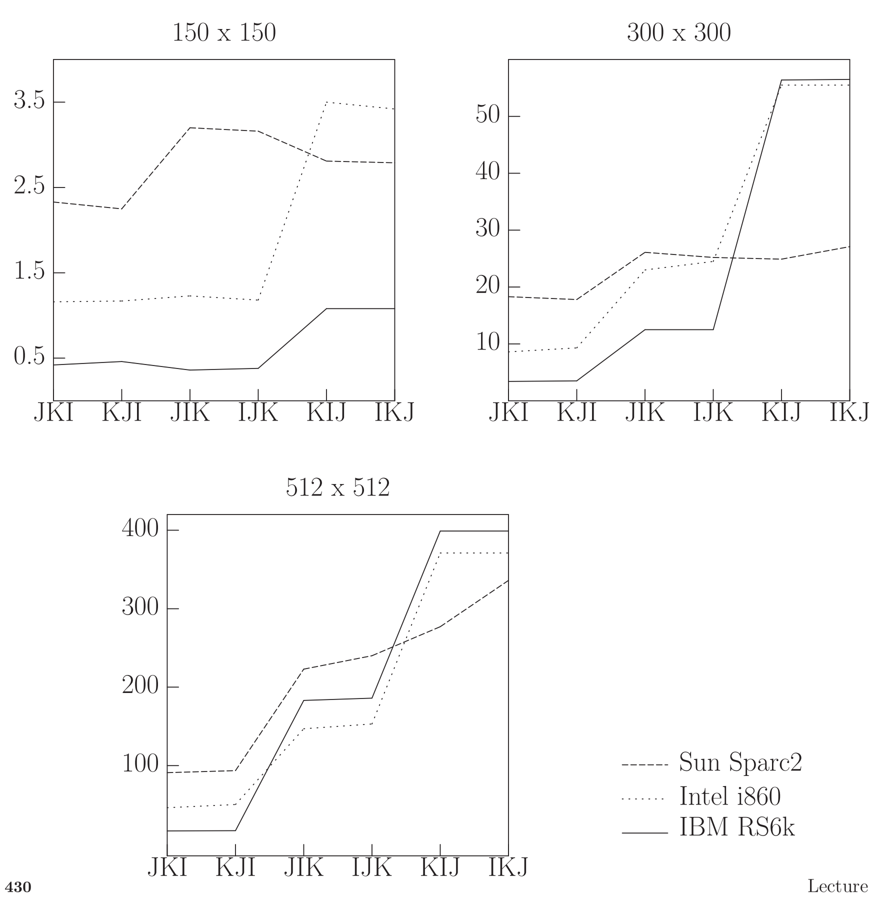
Matrix multiplication example

```
do j = 1, N
   do k = 1, N
     do i = 1, N
       C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

RefGroups	J	K	I
C(i,j)	$n * n^2$	$1 * n^2$	$\frac{1}{4}n * n^2$
A(i,k)	$1 * n^2$	$n * n^2$	$\frac{1}{4}n * n^2$
B(k,j)	$n * n^2$	$\frac{1}{4}n * n^2$	$1 * n^2$
total	$2n^3 + n^2$	$\frac{5}{4}n^3 + n^2$	$\frac{1}{2}n^3 + n^2$

LoopCost $(with\ cls = 4)$

Matrix Multiply (exec time in seconds)



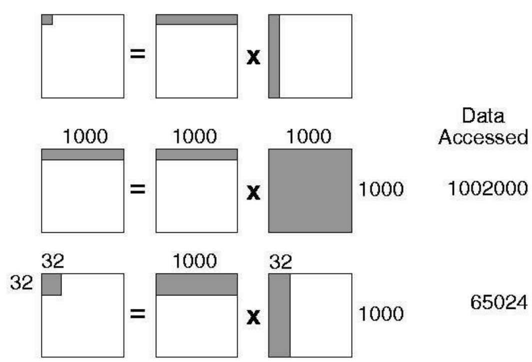
Matrix Multiply

Example

```
do i = 1, n
  do j = 1, n
    do k = 1, n
      A(i,j) = A(i,j) + B(i,k) * C(k,j)
```

Question

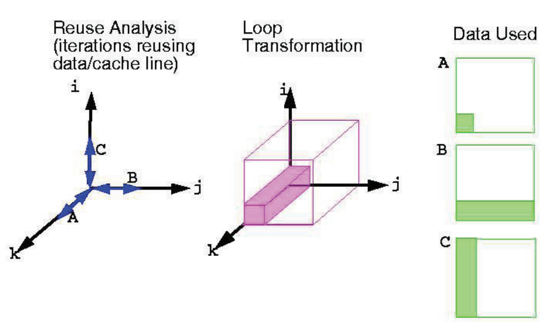
- suppose arrays do not fit in cache
- can we exploit more reuse?



Reusing Cache Lines

Matrix multiple example

```
DO i = 1, 500
  DO j = 1, 500
    DO k = 1, 500
      A[i,j] = A[i,j]+B[i,k]*C[k,j]
```



Tiled version

```
do ii = 1, 500, T
  do jj = 1, 500, T
    do i = ii, ii+T-1
      do j = jj, jj+T-1
        do k = 1, 500
          A(i,j) = A(i,j) + B(i,k) * C(k,j)
```

Tiling

Transformation

- strip-mine loops, then interchange

Benefits

- increases size of localized iteration set
- exploits reuse among multiple loops
- uses larger portion of cache

Problems

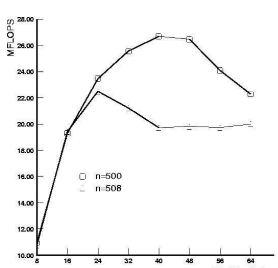
- less reuse per loop
- higher loop overhead
- needs information about cache size
- need to avoid conflict misses

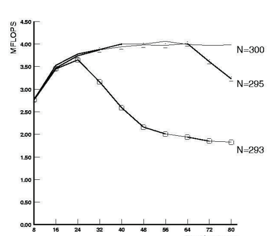
Question

- how to choose tile dimensions, size?
- is it simply question of cache size?

Performance of Tiling

IBM RS6000 DEC 3100





Results

- potentially large improvement, but sensitive to cache / array size

Cache Conflicts

- set associativity → conflict misses
- too many addresses mapped to same cache line
- common for power-of-two array sizes

M. Lam, E. Rothberg, M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," ASPLOS'91

Avoiding Cache Conflict

Use less cache

- choose smaller tile sizes
- reduces portion of cache used
- lowers chance of cache conflict
- empirically, 20–30% seems to work
- higher loop overhead

Copy optimization

- copy each tile to contiguous locations
- modify code to access new location
- amortize overhead for high reuse
- explicit copy code, overhead

Rectangular tiles

- calculate rectangular tile size
- explicitly chosen to avoid cache conflict
- may result in long thin tiles
  (equivalent to no tiling)

Scalar Replacment

Array references

- difficult to allocate to registers
- need to identify potential aliases
- dependences point out reuse

```
do i = 1,100
  A(0) = A(0)+A(i)
enddo
```

Scalar replacement transformation

- replace array reference with scalar
- eliminates aliases through renaming
- relies on dependence analysis
- simplifies scalar compiler back end

```
t = A(0)
do i = 1,100
  t = t+A(i)
enddo
A(0) = t
```

D. Callahan, S. Carr, K. Kennedy, "Improving Register Allocation for Subscripted Variables," PLDI'90

Unroll-and-Jam

Reusing registers on outer loops

- temporal locality (deps) at outer loop
- need to move reuse to loop body

Unroll-and-jam transformation

- unroll outer loop
- fuse (jam) inner loops
- results in multiple outer loop bodies

```
{1:original}        {2:unroll i}
do i=1,100          do i=1,100,2
  do j=1,100          do j=1,100
    A(i)+=B(j)          A(i)  +=B(j)
                      do j=1,100
                        A(i+1)+=B(j)

{3:fuse j}          {4:replace B}
do i=1,100,2        do i=1,100,2
  do j=1,100          do j=1,100
    A(i)  +=B(j)        t = B(j)
    A(i+1)+=B(j)        A(i)  += t
                        A(i+1)+= t
```


Memory Latency

Data locality

- loop permutation, tiling increase reuse
- accesses more likely to be in cache
- but still some cache misses

Latency

- time between data request and receipt
- needed to move data at address to processor
- can overlap memory fetch with computation or other memory fetches!

Approaches to reducing memory latency

- faster memory
  (get data to processor faster)
- nonblocking caches
  (continue execution after miss)
- hardware prefetching
  (fetch rest of data on cache line)
- software prefetching
  (instructions to prefetch data)

Software Prefetching

Indiscriminate prefetching

- insert prefetch for every reference
- high instruction overhead
- 60–95% of prefetches redundant in study

```
do i=1,100
   ...
   s += A(i)
```

```
do i=1,100
  prefetch A(i)
  ...
  s += A(i)
```

Selective prefetching

- reuse analysis → whether data in cache
- translate into *prefetch predicate* $\mathcal{P}$
  - temporal locality: i = 0
  - spatial locality: i mod *cls* = 0
  - group reuse: prefetch leader of group
- issue prefetch if predicates satisfied
- loop transformations to avoid conditionals
  - peel or split if $\mathcal{P}$ is i = 0
  - strip-mine or unroll if $\mathcal{P}$ is i mod *cls* = 0
- software pipeline fetches across iterations

Software Prefetching

Example

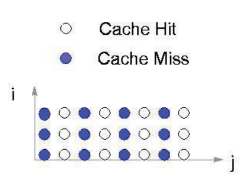
- two elements per cache line
- latency of six iterations

Original Code

```
for (i=0; i<3; i++)
  for (j=0; j<100;j++)
    A[i][j] = ...;
```

With Prefetching

```
for (i=0; i<3; i++) {
  for (j=0; j<6; j+=2) {
    prefetch(&A[i][j]);
  }
  for (j=0; j<94; j+=2) {
    prefetch(&A[i][j+6]);
    A[i][j] = ... ;
    A[i][j+1] = ...;
  }
  for (j=94; j<100; j++){
    A[i][j] = ...;
  }
```



Prefetching side effects

- higher instruction overhead
- increases application memory bandwidth
- increases lifetime of cache line,
  may cause more misses

Dependence Analysis

Question

Do two references never/maybe/always access the same memory location?

Benefits

- improves alias analysis
- enables loop transformations

Motivation

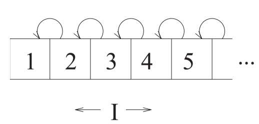
- classic optimizations
- instruction scheduling
- data locality (register/cache reuse)
- vectorization, parallelization

Obstacles

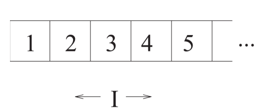
- array references
- pointer references

Dependence Analysis

```
do I = 1, 100
   A(I) =
      = A(I-1)
enddo
```




```
do I = 1, 100
   A(I) =
      = A(I)
enddo
```



A **loop-independent** dependence exists regardless of the loop structure. The source and sink of the dependence occur on the same loop iteration.

A **loop-carried** dependence is induced by the iterations of a loop. The source and sink of the dependence occur on different loop iterations.

Loop-carried dependences can inhibit parallelization and loop transformations

Dependence Testing

Given

$$
\begin{array}{ll}
 & \texttt{do } i_1 \texttt{ = } L_1, U_1 \\
 & \quad \ldots \\
 & \qquad \texttt{do } i_n \texttt{ = } L_n, U_n \\
S_1 & \qquad\quad \texttt{A(}f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)\texttt{)} \;\; \texttt{=} \;\; \ldots \\
S_2 & \qquad\quad \ldots \;\; \texttt{=} \;\; \texttt{A(}g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)\texttt{)}
\end{array}
$$

A *dependence* between statement S_1 and S_2, denoted $S_1 \delta S_2$, indicates that S_1, the *source*, must be executed before S_2, the *sink* on some iteration of the nest.

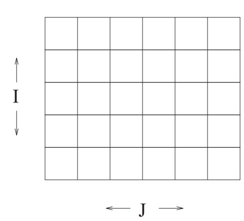
Let α & β be a vector of n integers within the ranges of the lower and upper bounds of the n loops.

Does $\exists\ \alpha \leq \beta$, s.t.

$$f_k(\alpha) \;=\; g_k(\beta) \quad \forall k,\ 1 \leq k \leq m\ ?$$

Iteration Space

```
do I = 1, 5
   do J = I, 6
      ...
   enddo
enddo
```



1 $\leq$ I $\leq$ 5
I $\leq$ J $\leq$ 6

- lexicographical (sequential) order for the above iteration space is

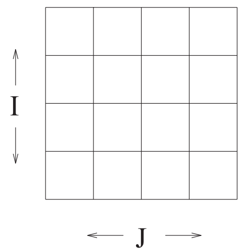
  (1,1), (1,2), …, (1,6)
  (2,2), (2,3), … (2,6)
  …
  (5,5), (5,6)

- given $I = (i_1, \ldots i_n)$ and $I' = (i'_1, \ldots, i'_n)$,

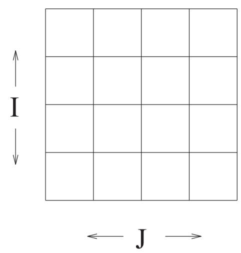
  $I \;<\; I'$ *iff*
  $(i_1, i_2, \ldots i_k) = (i'_1, i'_2, \ldots i'_k)\ \ \&\ \ i_{k+1} < i'_{k+1}$

Distance Vectors

```
 do I = 1, N
  do J = 1, N
S1 A(I,J) = A(I,J-1)
  enddo
 enddo
```




```
 do I = 1, N
  do J = 1, N
S2 A(I,J) = A(I-1,J-1)
S3 B(I,J) = B(I-1,J+1)
  enddo
 enddo
```

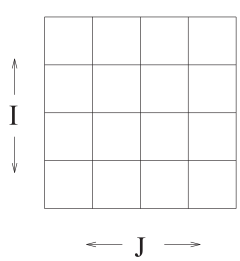


Distance Vector = number of iterations between accesses to the same location

	distance vector
$S_1 \delta S_1$	(0,1)
$S_2 \delta S_2$	(1,1)
$S_3 \delta S_3$	(1,-1)

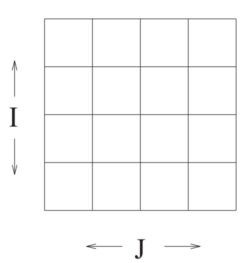
Loop Interchange

```
  do I = 1, N
    do J = 1, N
S1    A(I,J) = A(I,J-1)
S2    B(I,J) = B(I-1,J-1)
    enddo
  enddo
```



$\Longrightarrow$ loop interchange $\Longrightarrow$

```
  do J = 1, N
    do I = 1, N
S1    A(I,J) = A(I,J-1)
S2    B(I,J) = B(I-1,J-1)
    enddo
  enddo
```



Loop interchange is safe *iff*

- it does not create a lexicographically negative direction vector (1,–1) $\rightarrow$ (–1,1)

$\Rightarrow$ Benefits

- may expose parallel loops, increase granularity
- reordering iterations may improve reuse

Forms of Parallelism

Instruction-level parallelism

- for superscalar and VLIW architectures
- examine dependences between statements
- very fine grain parallelism

```
A = 1                 B = C
```

Task-level parallelism

- for multiprocessors
- examine dependences between tasks
- parallelism is not scalable

```
do i = 1,10           do i = 1,10
  A(i) = A(i+1)         B(i) = B(i+1)
```

Loop-level parallelism

- for vector machines and multiprocessors
- examine dependences between loop iterations
- parallelism is scalable

```
doall i = 1,10
  A(i) = B(i+1)
```


Loop-level Parallelism

Basic approach

- execute loop iterations in parallel
- safe if no loop-carried data dependences
  (i.e., no accesses to same memory location)

```
do i = 1,10           doall i = 1,10
  A(i) = A(i+1)         A(i) = A(i+10)
```

Several parallel architectures

- vector processors

```
A[1:10] = B[2:11]
```

- multiprocessors

```
doall i = 1,10
  A(i) = B(i+1)
```

- message-passing machines

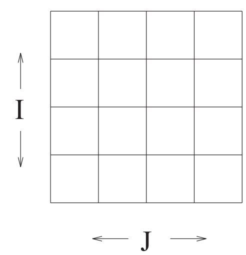
```
if (...) send B(1)
if (...) recv B(11)
do i = L,B
  A(i) = B(i+1)
```

Which Loops are Parallel?

```
 do I = 1, N
  do J = 1, N
S1 A(I,J) = A(I,J-1)

 do I = 1, N
  do J = 1, N
S2 A(I,J) = A(I-1,J-1)

 do I = 1, N
  do J = 1, N
S3 B(I,J) = B(I-1,J+1)
```



- a dependence $D = (d_1, \ldots, d_k)$ is *carried* at *level* i, if d_i is the first nonzero element of the distance vector
- a loop l_i is *parallel*, if $\nexists$ a dependence D_j carried at level i

	distance vector
$\forall D_j$	$d_1, \ldots, d_{i-1} > 0$
OR	$d_1, \ldots, d_i = 0$

Exposing Parallelism

Scalar analysis

- improve precision of dependence tests
- eliminate unnecessary scalar statements
- based on data-flow analysis

Solution techniques

- forward propagation
  (expose value of scalar variables)

```
do i = 1,10          do i =1,10
  k = i+1              k = i+1
  A(k) =               A(i+1) =
```

- constant propagation

```
k = 1                k = 1
do i = 1,10          do i =1,10
  A(i+k) =             A(i+1) =
```

- induction variable recognition

```
k = 1                k = 1
do i = 1,10          do i =1,10
  k = k+1              A(i+1) =
  A(k) =             k = 11
```

Exposing Parallelism

Storage-related dependences

- anti and output dependences
- caused by reusing storage
- no flow of values (not inherently sequential)

Solution techniques

- renaming

```
do i = 1,10          do i = 1,10
  A(i) = A(i+1)        T(i) = A(i+1)
                     do i = 1,10
                       A(i) = T(i)
```

- scalar/array expansion

```
do i = 1,10          do i =1,10
  t =                  t(i) =
    = t                  = t(i)
```

- scalar/array privatization

```
do j = 1,10          do j = 1,10
  do i = 1,10          private A(10)
    A(i) =             do i = 1,10
    B(i,j) = A(i)        A(i) = t
                         B(i,j) = A(i)
```


Exposing Parallelism

Reductions

- loop-carried true (flow) dependences
- operations are associative (can commute)

```
do i = 1,10          do i = 1,10
  t = t + A(i)         if (t < A(i))
                         t = A(i)
```

- roundoff error for floating point arithmetic

Solution techniques

- vector reduction operation

```
do i = 1,10          t = VADD(A[1:10])
  t = t + A(i)
```

- parallelize reduction

```
do i = 1,10          private tt
  t = t + A(i)       tt = 0
                     do i = L,U
                       tt = tt + A(i)
                     lock
                     t = t + tt
                     unlock
```

Vectorization

Vector processors

- operations on vectors of data
- overlap iterations of inner loop

```
doall i = 1,10      A[1:10] = 1.0
  A(i) = 1.0        B[1:10] = A[1:10]
  B(i) = A(i)
```

- exploits fine-grain parallelism
- expressed in vector languages (APL, Fortran 90)

Execution model

- single thread of control
  single instruction, multiple data (SIMD)
- load data into vector registers
- efficiently execute pipelined operations

Issues

- vector length - coalesce loops to reduce overhead
- control flow - convert conditions into explicit data

Parallelization

Multiprocessors

- multiple independent processors (MIMD)
- assign iterations to different processors

```
doall j = 1,10      do j = L,U
  do i = 1,10         do i = 1,10
    A(i,j) = 1.0        A(i,j) = 1.0
```

- exploits coarse-grain parallelism

Execution model

- fork-join parallelism
- master executes sequential code
- workers (and master) execute parallel code
- master continues after workers finish

Issues

- granularity - larger computation partitions to reduce overhead
- scheduling - policy for assigning iterations to processors

Multithreading

High latency event

- I/O
- interprocessor communication
- page miss
- cache miss

Multiple threads of execution

- switch to new thread after event
- overlaps computation with event
- requires threads, efficient context switch

Hardware support

- switch sets of registers
- shared caches
- HEP, Tera

Software support

- uncover parallelism for multiple threads
- reduce context switch overhead