Compiling For High Performance

Issues
- parallelism
- locality

Classical compilation
- focuses on individual operations
- uses of values
- unstructured code

High-performance compilation
- focuses 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 vs reuse

Locality
- temporal locality
- spatial locality

Reuse
- self-reuse
- group-reuse

What reuse occurs in this loop nest?

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

<table>
<thead>
<tr>
<th>Ref</th>
<th>Reuse on loop i</th>
<th>Reuse on loop j</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>spatial</td>
<td>temporal</td>
</tr>
<tr>
<td>B</td>
<td>temporal, group</td>
<td>spatial, group</td>
</tr>
</tbody>
</table>

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
2. compute the cost in cache lines accessed
3. rank the loops based on their loop cost

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 \( l \)
1. select representative from reference group
2. find cost in (cache lines) with \( k \) innermost
a. invariant stride
   otherwise \( \frac{(L_k - L_k + 1)}{c_{ls}} \)
   otherwise \( L_k - L_k + 1 \)
3. multiply by trip counts of outer loops

Loop cost = sum of costs for reference groups

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

Matrix Multiply (exec time in seconds)

![Graph showing execution time for matrix multiplication with different matrix sizes and processors]
Matrix Multiply

Example

\[
\begin{align*}
\text{do } i = 1, n \\
\text{do } j = 1, n \\
\text{do } k = 1, n \\
A(i,j) &= A(i,j) + B(i,k) \cdot C(k,j)
\end{align*}
\]

Question

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

Reusing Cache Lines

Matrix multiple example

\[
\begin{align*}
\text{do } ii = 1, 500, T \\
\text{do } jj = 1, 500, T \\
\text{do } i = ii, ii+T-1 \\
\text{do } j = jj, jj+T-1 \\
\text{do } k = 1, 500 \\
A(i,j) &= A(i,j) + B(i,k) \cdot C(k,j)
\end{align*}
\]

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 \(\rightarrow\) conflict misses
- too many addresses mapped to same cache line
- common for power-of-two array sizes


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 Replacement

Array references

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

\[
\begin{align*}
\text{do } i = 1, 100 \\
A(0) &= A(0) + A(i) \\
\text{enddo}
\end{align*}
\]

Scalar replacement transformation

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

\[
\begin{align*}
t &= A(0) \\
\text{do } i = 1, 100 \\
t &= t + A(i) \\
\text{enddo}
\end{align*}
\]

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

Original:
\[
\text{do } i=1,100 \text{ do } j=1,100 \text{ } A(i)+=B(j)
\]

Unrolled:
\[
\text{do } i=1,100,2 \text{ do } j=1,100 \text{ } A(i)+=B(j)
\]

Fuse:
\[
\text{do } i=1,100,2 \text{ do } j=1,100 \text{ } A(i)+=B(j)
\]

Replace:
\[
\text{do } i=1,100,2 \text{ do } j=1,100 \text{ } t = B(j) \text{ } A(i)+=t
\]

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
- nonblocking caches
- hardware prefetching
- software prefetching

Software Prefetching

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

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

Software Prefetching

Example
- two elements per cache line
- latency of six iterations

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 \( i = 1, 100 \)
\[
\begin{align*}
A(i) &= A(i-1) \\
&= A(i)
\end{align*}
\]

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

do i1 = L1, U1
... do in = Ln, Un
S1 \ A(f1(i1, ..., in), ..., fn(i1, ..., in)) = ...
S2 \ A(g1(i1, ..., in), ..., gm(i1, ..., in)) = ...

A dependence between statement S1 and S2, denoted \( S1 \rightsquigarrow S2 \), indicates that S1, the source, must be executed before S2, the sink, on some iteration of the nest.

Let \( \alpha \leq \beta, \exists t \) such that

\[ f_\alpha = g_\beta \quad \forall k, 1 \leq k \leq m \]

Distance Vectors

\[
\begin{align*}
do I = 1, N 
d J = 1, N 
S1: A(I,J) &= A(I-1,J-1) 
enddo 

do I = 1, N 
d J = 1, N 
S2: B(I,J) &= B(I-1,J+1) 
enddo 

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

S1 \( S_1 \)
S2 \( S_2 \)
S1S2 \( S_1S_2 \)
(0,1)
(1,1)
(1,-1)

Loop Interchange

\[
\begin{align*}
do I = 1, N 
d J = 1, N 
S1: A(I,J) &= A(I-1,J-1) 
enddo 

do J = 1, N 
do I = 1, N 
S2: B(I,J) &= B(I-1,J+1) 
enddo 

\text{Loop interchange is safe if} \quad \text{it does not create a lexicographically negative direction vector} \quad (1,-1) \rightarrow (-1,1)

\text{Benefits}
\text{expression parallelism, increase granularity}
\text{reordering iterations may improve reuse}

Forms of Parallelism

Instruction-level parallelism
- for superscalar and VLIW architectures
- examine dependences between statements
- very fine grain parallelism
  \( A = 1 \quad B = C \)

Task-level parallelism
- for multiprocessors
- examine dependences between tasks
- parallelism is not scalable
  \( \text{do i = 1,10} \quad \text{do i = 1,10} \quad A(i) = A(i+1) \quad B(i) = B(i+1) \)

Loop-level parallelism
- for vector machines and multiprocessors
- examine dependences between loop iterations
- parallelism is scalable
  \( \text{doall i = 1,10} \quad A(i) = A(i+10) \quad B(i) = B(i+10) \)

Loop-level Parallelism

Basic approach
- execute loop iterations in parallel
- safe if no loop-carried data dependences
  (i.e., no access to same memory location)
  \( \text{do i = 1,10} \quad \text{doall i = 1,10} \quad A(i) = A(i+1) \quad B(i) = B(i+1) \)

Several parallel architectures
- vector processors
- multiprocessors
  \( \text{doall i = 1,10} \quad A(i) = B(i+1) \)
- message-passing machines
  \( \text{if ... send B(1)} \quad \text{if ... recv B(1)} \quad \text{do i = L to B} \quad A(i) = B(i+1) \)
Which Loops are Parallel?

do I = 1, N
do J = 1, N
S_1: A(I,J) = A(I,J-1)
do I = 1, N
do J = 1, N
S_2: A(I,J) = A(I-1,J-1)
do I = 1, N
do J = 1, N
S_3: B(I,J) = B(I-1,J+1)

• a dependence \(D = (d_1, \ldots, d_k)\) is carried at level \(i\), if \(d_1\) is the first nonzero element of the distance vector
• a loop \(l\) is parallel, if \(D_l\) carried at level \(i\)

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 j = 1,10
\(A(k) = A(i+j)\)
• constant propagation
do i = 1,10
do j = 1,10
\(A(i+j) = A(i+j)\)
• induction variable recognition
do i = 1,10
do j = 1,10
\(A(k) = A(i+1)\)

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
\(A(i) = A(i+1)\)

Exposing Parallelism

Reductions
• loop-carried true (flow) dependences
• operations are associative (can commute)
do i = 1,10
do i = 1,10
\(t = t + A(i)\)
• roundoff error for floating point arithmetic

Solution techniques
• vector reduction operation
do i = 1,10
do i = 1,10
\(t = VADD(A[i:10])\)
• parallelize reduction
do i = 1,10
do i = 1,10
\(t = t + A(i)\)

Vectorization

Vector processors
• operations on vectors of data
• overlap iterations of inner loop
do all i = 1,10
\(A(1:10) = 1.0\)
\(A(i) = A(i+1)\)
\(B(1:10) = B(i)\)
\(B(i) = A(i)\)
• 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 - reduce loops to reduce overhead
• control flow - convert conditions into explicit data

Parallization

Multiprocessors
• multiple independent processors (IMM)
• assign iterations to different processors
do all i = 1,10
do i = 1,10
\(A(i) = 1.0\)
\(A(i) = A(i+1)\)
\(B(i) = 1.0\)
\(B(i) = A(i)\)
• 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 - reduce 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