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

Why locality?

- memory accesses are expensive
- exploit higher levels of memory hierarchy by using registers, cache lines, TLB, etc.

Locality

- temporal locality
- reuse of a specific location
- self-reuse (caused by same reference)

- spatial locality
- reuse of adjacent locations
- group-reuse (caused by multiple references)

Reuse

- reuse on loop i
- reuse on loop j

Data Locality

Loop transformations
- dependence analysis
- unstructured code
- memory access patterns

High-performance compilation
- loop on nestable operations (loops)
- scalar variables
- loop of values
- scalar variables

Classical compilation
- locality
- parallelism
- issues

Compiler for High Performance

- dependence analysis
- unstructured code
- memory access patterns

Issues

- dependence analysis
- unstructured code
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 ⇒ reference groups
2. compute the cost in cache lines accessed ⇒ loop cost
3. rank the loops based on their loop cost ⇒ 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

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

\[ \text{cost of reference group for loop } k = \frac{1 + \frac{\gamma_k - \frac{u_k}{2}}{p/(1 + \frac{\gamma_k - \frac{u_k}{2})}}}{1} \]

Matrix multiplication example

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

RefGroups

Cost of reference group for loop \( k \):

\[ \text{cost of reference group for loop } k = \frac{1 + \frac{\gamma_k - \frac{u_k}{2}}{p/(1 + \frac{\gamma_k - \frac{u_k}{2})}}}{1} \]

Selecting a Loop Permutation

Avoids evaluating many permutations:

1. select representative from reference group
2. compute the cost in cache lines accessed
3. rank the loops based on their loop cost

If loop \( l \) promotes more reuse than loop \( k \) at the innermost position, then it

Key insight

Memory order ⇒

1. partition references with group-reuse
2. compute the cost in cache lines accessed
3. rank the loops based on their loop cost
4. select memory order

For each loop \( l \) in a nest, consider \( l \) innermost:

To calculate temporal and spatial reuse
Matrix Multiply (exec time in seconds)
Matrix Multiply

Example

\[
A(i', j') = A(i', j') + B(i', k') \cdot C(k', j')
\]

Question

- Suppose arrays do not fit in cache. Can we exploit more reuse?

Tiled version

\[
\begin{aligned}
do & \ i = 1, 500 \\
do & \ j = 1, 500 \\
do & \ k = 1, 500
\end{aligned}
\]

\[
A(i, j) = A(i, j) + B(i, k) \cdot C(k, j)
\]

Reusing Cache Lines

Matrix multiply example

Question

- Can we exploit more reuse?

Example

Matrix Multiply
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
- needs information about cache size
- needs information about cache size
- sets associativity → conflict misses

**Cache Conflicts**

potentially large improvement, but sensitive to cache / array size

**Results**

**Performance of Tiling**

**Question**

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


performance of Tiling

IBM RS6000

DEC 3100

Performance of Tiling

**Results**

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

**Question**

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

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)

Array References

Scalar replacement transformation

- Replace array reference with scalar
- Eliminates aliases through renaming
- Relies on dependence analysis
- Simplifies scalar compiler back end
- Explicitly chosen to avoid cache conflict
- May result in long thin tiles

Scalar replacement transformation

- Replace array reference with scalar
- Eliminates aliases through renaming
- Relies on dependence analysis
- Simplifies scalar compiler back end
- Explicitly chosen to avoid cache conflict
- May result in long thin tiles
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

\[
\begin{align*}
&\text{do } i=1,100,2 \text{ do } i=1,100,2 \\
&\text{do } j=1,100 \text{ do } j=1,100 \\
&A(i) += B(j) \quad A(i) += B(j) \\
&\text{do } j=1,100 \\
&A(i+1) += B(j) \quad A(i) += t \\
&\text{do } i=1,100,2 \text{ do } i=1,100,2 \\
&\text{do } j=1,100 \text{ do } j=1,100 \\
&A(i) += B(j) \quad t = B(j) \\
&A(i+1) += t
\end{align*}
\]

CMSC 430
Lecture 24, Page 13

Memory Latency

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

Latency
- time between data request and receipt
- needed to move data to address of processor
- needed to move data at address and receipt

Approaches to reducing memory latency
- can overlap memory load with computation or other memory requests
- can overlap memory load with computation or other memory requests

Approaches to improving 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
- translate into prefetch predicate
- 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

Example

- two elements per cache line

Prefetching side effects

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

Software Prefetching

Example

- two elements per cache line

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

---

Loop-carried dependences occur on different loop iterations. The source and sink are on the same loop iteration.

Loop-independent dependences occur on different loop iterations. The source and sink are on the same loop iteration.

---

**Dependence Analysis**

```plaintext
loopp I = 1, 100
A(I) = A(I-1)
enddo

loopp I = 1, 100
A(I) = A(I)
enddo
```
Dependence Testing

Given

\[
\begin{align*}
\forall I > I' & \quad (\gamma_I \cdot \cdots \cdot \gamma_{I' - 1}) = (\gamma_{I'} \cdot \cdots \cdot \gamma_{I - 1}) \\
\forall I > I' & \quad S_I \cap S_{I'} = \emptyset \\
\end{align*}
\]

Exceptional (sequential) order for the above iteration space is

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
I & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
J & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

Iteration Space

```
do I = 1, 5
  do J = I, 6
    \ldots
  enddo
enddo
```

A dependence between statement \( S_I \) and \( S_{I'} \) on some iteration of the nest, \( I < I' \), indicates that \( S_I \), the source, must be executed before \( S_{I'} \), the sink, on some iteration of the nest.

Lett \( a \) and \( b \) be a vector of integers within the ranges of the lower and upper bounds of the \( I \) loop.

Dependence Testing

\[
\begin{align*}
\forall \alpha \geq \beta & \quad \gamma_I = \gamma_I' \\
\end{align*}
\]

Given

\[
\begin{align*}
((\gamma_I \cdot \cdots \cdot \gamma_{I' - 1})) & = \cdots \\
\forall I > I' & \quad S_I \cap S_{I'} = \emptyset \\
\end{align*}
\]
Distance Vectors = number of iterations between accesses to the same location

\[ \delta S^1 = (0,1) \]

\[ \delta S^2 = (1,1) \]

\[ \delta S^3 = (1,-1) \]

Loop interchange is safe iff

- it does not create a lexicographically negative direction vector

Loop interchange is safe iff

- it does not create a lexicographically negative direction vector

[Diagram]

Loop interchange is safe iff

- it does not create a lexicographically negative direction vector

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

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

Several parallel architectures
• vector processors
• doall i = 1,10
A(i) = B(i+1)
• multiprocessors
• message-passing machines
• if (...) send B(1)
• if(...) recv B(11)

Loop-level parallelism
• execute loop iterations in parallel
• safe if no loop-carried data dependences

(i.e., no access to same memory location)
• Vector processors
• Basic approach

Task-level parallelism
• very fine grain parallelism

B = C
• examine dependences between statements

Instruction-level parallelism
• for superscalar and VLIW architectures
Which Loops are Parallel?

\[
\begin{align*}
\text{do } i = 1, N \\
\text{do } j = 1, N \\
S_1 & = A(i, j-1) \\
\text{do } i = 1, N \\
\text{do } j = 1, N \\
S_2 & = A(i-1, j-1) \\
\text{do } i = 1, N \\
\text{do } j = 1, N \\
S_3 & = B(i, j+1)
\end{align*}
\]

\[D = (d_1, \ldots, d_k)\]

• a dependence carried at level \(i\), if \(d_i\) is the first nonzero element of the distance vector

- \(\forall\) a loop \(l_i\) is parallel, if \(\not\exists\) a dependence \(D_j\) carried at level \(i\)

\[d_i = (d_1, \ldots, d_i-1 > 0 \text{ OR } d_1, \ldots, d_i = 0)\]

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
  - constant propagation
  - induction variable recognition

Solution techniques based on data-flow analysis

Scalar analysis

CMSC 430 Lecture 24, Page 25

- \(0 = \begin{pmatrix} p^{i+1} & \cdots & p^1 \end{pmatrix} \) OR \(D\)
- a loop \(l_i\) is parallel, if \(\not\exists\) a dependence \(D\) carried at level \(i\)
- a dependence carried at level \(i\), if \(d_i\) is the first nonzero element

\[\text{Distance vector} = (d_1, \ldots, d_i-1) \]

Which loops are parallel?
Exposing Parallelism

### Storage-related dependences
- **anti** and output dependences
- Caused by reusing storage
- No flow of values (not inherently sequential)

### Solution techniques
- **Renaming**
  ```fortran
do i = 1,10
da = A(i + 1)
t = t + A(i)
```
- **Scalar/array expansion**
  ```fortran
do i = 1,10
t = t(i)
```
- **Scalar/array privatization**
  ```fortran
do j = 1,10
doi = 1,10
A(i) =
B(i,j) = A(i)
```

### Reductions
- **Loop-carried true (flow) dependences**
- Operations are associative (can commute)

### Solution techniques
- **Vector reduction operation**
  ```fortran
do i = 1,10
t = VADD(A[1:10])
t = t + A(i)
```
- **Parallelize reduction**
  ```fortran
do i = 1,10
A(t) = t + A(i)
```
- **Roundoff error for floating-point arithmetic**
  ```fortran
do i = 1,10
if (t < A(i))
t = A(i)
```

### Solution techniques for shared and output dependences
- **Vector reduction operation**
  ```fortran
do i = 1,10
A(t) = t = A(i)
```
- **Roundoff error for floating-point arithmetic**
  ```fortran
do i = 1,10
if (t < A(i))
t = A(i)
```

---

Exposing Parallelism

### Reductions
- **Loop-carried true (flow) dependences**
- Operations are associative (can commute)

### Solution techniques
- **Vector reduction operation**
  ```fortran
do i = 1,10
t = VADD(A[1:10])
t = t + A(i)
```
- **Parallelize reduction**
  ```fortran
do i = 1,10
A(t) = t + A(i)
```
- **Roundoff error for floating-point arithmetic**
  ```fortran
do i = 1,10
if (t < A(i))
t = A(i)
```

---

Exposing Parallelism

### Reductions
- **Loop-carried true (flow) dependences**
- Operations are associative (can commute)

### Solution techniques
- **Vector reduction operation**
  ```fortran
do i = 1,10
t = VADD(A[1:10])
t = t + A(i)
```
- **Parallelize reduction**
  ```fortran
do i = 1,10
A(t) = t + A(i)
```
- **Roundoff error for floating-point arithmetic**
  ```fortran
do i = 1,10
if (t < A(i))
t = A(i)
```
Vectorization

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

```
A[1:10] = 1.0
A(i) = 1.0
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 parallel operations
  - vector length - coalesce loops to reduce overhead
  - control flow - convert conditions into explicit data

Issues

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

Practice problems

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

Multiprocessors

- shorthand for assigning iterations
  - AVxM

Parallelization

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

Issues

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

Execution model

- expressed in vector languages (AVxM, Fortran 90)

```
B[1:10] = 0.0
```

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

Vector 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
- switch sets of registers
- HEFT
- shared caches

Software support

Hardware support
- requires threads, different context switch
- overhead computation with event
- High latency event

- uncover parallelism for multiple threads
- reduce context switch overhead

CMSC 430
Lecture 24, Page 31