Memory Optimization

Saman Amarasinghe
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Outline

• Issues with the Memory System
• Loop Transformations
• Data Transformations
• Prefetching
• Alias Analysis
Memory Hierarchy

1 - 2 ns  Registers  32 – 512 B

3 - 10 ns  L1 Private Cache  16 – 128 KB

8 - 30 ns  L2/L3 Shared Cache  1 – 16 MB

60 - 250 ns  Main Memory (DRAM)  1 GB – 128 GB

5 - 20 ms  Permanent Storage (Hard Disk)  250 MB – 4 TB
## Cache Architecture

<table>
<thead>
<tr>
<th></th>
<th>Pentium D</th>
<th>Core Duo</th>
<th>Core 2 Duo</th>
<th>Athelon 64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 code</strong></td>
<td>size</td>
<td>12 K uops</td>
<td>32 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td></td>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
</tr>
<tr>
<td></td>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L1 data</strong></td>
<td>size</td>
<td>16 KB</td>
<td>32 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td></td>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
</tr>
<tr>
<td></td>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L1 to L2</strong></td>
<td>Latency</td>
<td>4 cycles</td>
<td>3 cycles</td>
<td>3 cycles</td>
</tr>
<tr>
<td><strong>L2 shared</strong></td>
<td>size</td>
<td>4 MB</td>
<td>4 MB</td>
<td>4 MB</td>
</tr>
<tr>
<td></td>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>16 way</td>
</tr>
<tr>
<td></td>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L2 to L3(off)</strong></td>
<td>Latency</td>
<td>31 cycles</td>
<td>14 cycles</td>
<td>14 cycles</td>
</tr>
</tbody>
</table>

Saman Amarasinghe

6.035 ©MIT Fall 2006
Cache Misses

- Cold misses
  - First time a data is accessed

- Capacity misses
  - Data got evicted between accesses because a lot of other data (more than the cache size) was accessed

- Conflict misses
  - Data got evicted because a subsequent access fell on the same cache line (due to associativity)

- True sharing misses (multicores)
  - Another processor accessed the data between the accesses

- False sharing misses (multicores)
  - Another processor accessed different data in the same cache line between the accesses
Data Reuse

- **Temporal Reuse**
  - A given reference accesses the same location in multiple iterations

- **Spatial Reuse**
  - Accesses to different locations within the same cache line

- **Group Reuse**
  - Multiple references access the same location

```plaintext
for i = 0 to N
  for j = 0 to N
    A[j] =

for i = 0 to N
  for j = 0 to N
    B[i, j] =

for i = 0 to N
  A[i] = A[i-1] + 1
```
Outline

• Issues with the Memory System
• Loop Transformations
• Data Transformations
• Prefetching
• Alias Analysis
Loop Transformations

• Transform the iteration space to reduce the number of misses
• Reuse distance – For a given access, number of other data items accessed before that data is accessed again
• Reuse distance > cache size
  – Data is spilled between accesses
Loop Transformations

for $i = 0$ to $N$
  for $j = 0$ to $N$
    for $k = 0$ to $N$
      $A[k,j]$

Reuse distance = $N^2$

If Cache size < 16 doubles?
A lot of capacity misses

©MIT Fall 2006
Loop Transformations

for i = 0 to N
    for j = 0 to N
        for k = 0 to N
            A[k,j]

Loop Interchange

for j = 0 to N
    for i = 0 to N
        for k = 0 to N
            A[k,j]
Loop Transformations

for \( j = 0 \) to \( N \)
  for \( i = 0 \) to \( N \)
    for \( k = 0 \) to \( N \)
      A[k,j]

Cache line size > data size
Cache line size = \( L \)
Reuse distance = \( LN \)

If cache size < 8 doubles?
  Again a lot of capacity misses
Loop Transformations

for j = 0 to N
  for i = 0 to N
    for k = 0 to N
      A[k,j]

Loop Interchange

for k = 0 to N
  for i = 0 to N
    for j = 0 to N
      A[k,j]
Loop Transformations

for $i = 0$ to $N$
  for $j = 0$ to $N$
    for $k = 0$ to $N$

- No matter what loop transformation you do one array access has to traverse the full array multiple times
Example: Matrix Multiply

\[
\begin{align*}
1 & \quad 1024 \\
1 & \quad 1024 \\
\times & \\
1 & \quad 1024 \\
1 & \quad 1024
\end{align*}
\]

\[
\begin{align*}
1 & \\
1 & \\
\times & \\
1 & \\
1 &
\end{align*}
\]

Data Accessed

1,050,624
for $i = 0$ to $N$
for $j = 0$ to $N$

for $ii = 0$ to ceil($N/b$)
for $jj = 0$ to ceil($N/b$)
for $i = b*ii$ to min($b*ii+b-1, N$)
for $j = b*jj$ to min($b*jj+b-1, N$)
Example: Matrix Multiply

Data Accessed

1,050,624

66,560
Outline

• Issues with the Memory System
• Loop Transformations
• **Data Transformations**
• Prefetching
• Alias Analysis
False Sharing Misses

for J =
for all I =
X(I, J) = ...
Conflict Misses

for \( J = \)

forall \( I = \)

\( X(I, J) = \ldots \)
Eliminating False Sharing and Conflict Misses

\[\text{for } J = \]
\[\text{forall } I = \]
\[X(I, J) = \ldots\]
Data Transformations

• Similar to loop transformations

• All the accesses have to be updated
  – Whole program analysis is required
Strip-Minding
Create two dims from one

With blocksize=4

Storage

\[
\begin{bmatrix}
N
\end{bmatrix} \xrightarrow{\text{}} \begin{bmatrix}
4 \\
\frac{N}{4}
\end{bmatrix}
\]

Array

\[
\begin{bmatrix}
i
\end{bmatrix} \xrightarrow{\text{}} \begin{bmatrix}
i \mod 4 \\
i/4
\end{bmatrix}
\]

Memory Layout

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]

\[
\begin{array}{cccccccc}
0,0 & 1,0 & 2,0 & 3,0 & 4,0 & 5,1 & 6,1 & 7,1
\end{array}
\]
Strip-Minding
Create two dims from one

With blocksize=4

Permutation
Change memory layout

With permutation matrix
\[
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
\]
Data Transformation Algorithm

- Rearrange data: Each processor’s data is contiguous
- Use data decomposition
  - *, block, cyclic, block-cyclic
- Transform each dimension according to the decomposition
- Use a combination of strip-mining and permutation primitives
Example I: (Block, Block)
Example I: (Block, Block)

\[ i_1 \mod \left\lfloor \frac{d_1}{P} \right\rfloor \]

\[ i_1 / \left\lfloor \frac{d_1}{P} \right\rfloor \]

Strip-Mine

\[
\begin{pmatrix}
i_1 \\
i_2
\end{pmatrix}
\]

\[
\begin{pmatrix}
i_1 \mod \left\lfloor \frac{d_1}{P} \right\rfloor \\
i_1 / \left\lfloor \frac{d_1}{P} \right\rfloor \\
i_2
\end{pmatrix}
\]
Example I: (Block, Block)

Strip-Mine

\[
\begin{align*}
    i_1 &\mod \left\lceil \frac{d_1}{P} \right\rceil \\
    i_1 &\div \left\lfloor \frac{d_1}{P} \right\rfloor \\
    i_2 &\mod \left\lceil \frac{d_1}{P} \right\rceil \\
    i_2 &\div \left\lfloor \frac{d_1}{P} \right\rfloor
\end{align*}
\]

Permute

\[
\begin{align*}
    i_1 &\mod \left\lceil \frac{d_1}{P} \right\rceil \\
    i_2 &\mod \left\lceil \frac{d_1}{P} \right\rceil \\
    i_1 &\div \left\lfloor \frac{d_1}{P} \right\rfloor \\
    i_2 &\div \left\lfloor \frac{d_1}{P} \right\rfloor
\end{align*}
\]
Example I: (Cyclic, *)
Example 1: (Cyclic, *)

Strip-Mine

\[
\begin{align*}
i_1 &\mod P \\
i_1 / P \\
i_2 &
\end{align*}
\]
Example 1: (Cyclic, *)

Strip-Mine

\[ i_1 \mod P \]

\[ i_1 / P \]

\[ i_2 \]

Permute

\[ i_1 \mod P \]

\[ i_2 \]

\[ i_1 / P \]
Performance

LU Decomposition (256x256)

5 point stencil (512x512)

LU Decomposition (1Kx1K)

- Parallelizing outer loop
- Best computation placement
- + data transformations
Optimizations

• Modulo and division operations in the index calculation
  - Very high overhead

• Use standard techniques
  - Loop invariant removal, CSE
  - Strength reduction exploiting properties of modulo and division
  - Use knowledge about the program
Outline

• Issues with the Memory System
• Loop Transformations
• Data Transformations
• Prefetching
• Alias Analysis
Prefetching

• Cache miss stalls the processor for hundreds of cycles
  – Start fetching the data early so it’ll be available when needed

• Pros
  – Reduction of cache misses → increased performance

• Cons
  – Prefetch contents for fetch bandwidth
    • Solution: Hardware only issue prefetches on unused bandwidth
  – Evicts a data item that may be used
    • Solution: Don’t prefetch too early
  – Pretech is still pending when the memory is accessed
    • Solution: Don’t prefetch too late
  – Prefetch data is never used
    • Solution: Prefetch only data guaranteed to be used
  – Too many prefetch instructions
    • Prefetch only if access is going to miss in the cache
Prefetching

• Compiler inserted
  – Use reuse analysis to identify misses
  – Partition the program and insert prefetches

• Run ahead thread (helper threads)
  – Create a separate thread that runs ahead of the main thread
  – Runahead only does computation needed for control-flow and address calculations
  – Runahead performs data (pre)fetched
Outline

• Issues with the Memory System
• Loop Transformations
• Data Transformations
• Prefetching
• Alias Analysis
Alias Analysis

- Aliases destroy local reasoning
  - Simple, local transformations require global reasoning in the presence of aliases
  - A critical issue in pointer-heavy code
  - This problem is even worse for multithreaded programs

- Two solutions
  - Alias analysis
    - Tools to tell us the potential aliases
  - Change the programming language
    - Languages have no facilities for talking about aliases
    - Want to make local reasoning possible

Courtesy of Alex Aiken. Used with permission.
Aliases

- Definition

*Two pointers that point to the same location are* **aliases**

- Example

```c
Y = &Z
X = Y
*X = 3  /* changes the value of *Y */
```

Courtesy of Alex Aiken. Used with permission.
Example

```c
foo(int * A, int * B, int * C, int N)
    for i = 0 to N-1
```

• Is this loop parallel?

• Depends

```c
int X[1000];
int Y[1000];
int Z[1000]
foo(X, Y, Z, 1000);
```

```c
int X[1000];
foo(&X[1], &X[0], &X[2], 998);
```
Points-To Analysis

• Consider:
  
  \[
  \begin{align*}
  P &= & Q \\
  Y &= & Z \\
  X &= & Y \\
  *X &= & P \\
  \end{align*}
  \]

• Informally:
  
  - P can point to Q
  - Y can point to Z
  - X can point to Z
  - Z can point to Q

Courtesy of Alex Aiken. Used with permission.
Points-To Relations

- A graph
  - Nodes are program names
  - Edge (x, y) says x may point to y

- Finite set of names
  - Implies each name represents many heap cells
  - Correctness: If *x = y in any state of any execution, then (x, y) is an edge in the points-to graph

Courtesy of Alex Aiken. Used with permission.
Sensitivity

• *Context sensitivity*
  - Separate different uses of functions
  - Different is the key – if the analysis think the input is the same, reuse the old results

• *Flow sensitivity*
  - For insensitivity makes any permutation of program statements gives same result
  - Flow sensitive is similar to data-flow analysis
Conclusion

• Memory systems are designed to give a huge performance boost for “normal” operations

• The performance gap between good and bad memory usage is huge

• Programs analyses and transformations are needed

• Can off-load this task to the compiler
6.035 Computer Language Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.