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
- On-chip Cache
  - 1 – 16 KB

25 - 50 ns
- Off-Chip Cache (SRAM)
  - 64 – 256 KB

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

5 - 20 ms
- Permanent Storage (Hard Disk)
  - 100 MB – 1 TB
Processor-Memory Gap

µProc
60%/year
(2x/1.5yr)

DRAM
9%/year
(2x/10 yrs)

Year

Performance

## 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 (per core)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>12 K uops</td>
<td>32 KB</td>
<td>32 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
<td>2 way</td>
</tr>
<tr>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L1 data (per core)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>16 KB</td>
<td>32 KB</td>
<td>32 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
</tr>
<tr>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L1 to L2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>4 cycles</td>
<td>3 cycles</td>
<td>3 cycles</td>
<td>3 cycles</td>
</tr>
<tr>
<td><strong>L2 shared</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>4 MB</td>
<td>4 MB</td>
<td>4 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>16 way</td>
<td>16 way</td>
</tr>
<tr>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L2 to L3 (off)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>31 cycles</td>
<td>14 cycles</td>
<td>14 cycles</td>
<td>20 cycles</td>
</tr>
</tbody>
</table>
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
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 $N$
      $A[k, j]$
for $j = 0$ to $N$
  for $i = 0$ to $N$
    for $k = 0$ to $N$
      $A[k,j]$

Cache line size $> \text{data size}$
Cache line size $= L$
Reuse distance $= LN$

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

\[ \begin{array}{ccc}
1 & 1024 \\
1024 & 1024 \\
\end{array} \times \begin{array}{ccc}
1 & 1024 \\
1024 & 1024 \\
\end{array} = \begin{array}{ccc}
1 & 1024 \\
1024 & 1024 \\
\end{array} \]

Data Accessed

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

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

\[
\begin{align*}
\begin{array}{ccc}
1 & \text{1024} & x \\
1 & \text{1024} & 1 \\
1 & \text{1024} & 1024 \\
\end{array}
& = \\
\begin{array}{ccc}
1 & \text{1024} & 1 \\
1 & \text{1024} & 1024 \\
32 & \text{1024} & 32 \\
\end{array}
\end{align*}
\]

Data Accessed:

\begin{align*}
1,050,624 \\
66,560
\end{align*}
Outline

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

for J =
forall I =
X(I, J) = ...

Array X
Cache Lines
Conflict Misses

for J =
forall I =
X(I, J) = ...
Eliminating False Sharing and Conflict Misses

for J =

forall I =

X(I, J) = ...
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 Declaration

\[ \begin{array}{c}
\left[ N \right] \\
\end{array} \]

Array Access

\[ \begin{array}{c}
\left[ i \right] \\
\end{array} \]

Memory Layout

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

With blocksize=4

<table>
<thead>
<tr>
<th>Storage Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [N] \rightarrow 4 \rightarrow \frac{N}{4} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [i] \rightarrow i \mod 4 \rightarrow i/4 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Layout</th>
</tr>
</thead>
</table>
| \[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 0 & 1 & 0 & 2 & 0 & 3 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\] |

With permutation matrix \(
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
\end{pmatrix}
\)

### Permutation
Change memory layout

With permutation matrix \( \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \)
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)

Strip-Mine

\[ i_1 \mod \left( \frac{d_1}{P} \right) \]

\[ i_1 / \left( \frac{d_1}{P} \right) \]

\[ i_2 \]
Example I: (Block, Block)

Strip-Mine

\[ i_1 \mod \frac{d_1}{P} \]
\[ i_1 \div \frac{d_1}{P} \]
\[ i_2 \]

Permute

\[ i_1 \mod \frac{d_1}{P} \]
\[ i_2 \]
\[ i_1 \div \frac{d_1}{P} \]
Example I: (Cyclic, *)
Example I: (Cyclic, *)

i_1 \mod P

\frac{i_1}{P}

i_2

Strip-Mine

i_1

i_2
Example I: (Cyclic, *)

Strip-Mine

\[
i_1 \mod P
\]
\[
i_1 / P
\]

Permute

\[
i_1 \mod P
\]
\[
i_2
\]
\[
i_1 / P
\]
Performance

LU Decomposition (256x256)

LU Decomposition (1Kx1K)

5 point stencil (512x512)

- 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
Aliases

- Definition

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

- Example

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

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

• Is this loop parallel?

• Depends

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

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

- Consider:
  - $P = &Q$
  - $Y = &Z$
  - $X = Y$
  - $*X = P$

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

From Prof. Aiken @ Stanford CS 294-1 Lecture 15
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
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