ECE 454
Computer Systems Programming
Optimizing for Caches

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
Content

- Cache basics and organization (last lecture)
- Optimizing for caches (this lecture)
  - Loop reordering
  - Tiling/blocking
- Prefetching (later)
- Virtual memory (later)
Optimizing for Caches
Memory Optimizations

• **Write code that has locality**
  - Spatial: access data that is contiguous as much as possible
  - Temporal: access the same data within short intervals of time

• **How to achieve locality?**
  - Proper choice of algorithm
  - Loop transformations
Background: Array Allocation

- \( T \ A[L] \);
  - Array of data type \( T \) and length \( L \)
  - Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes

```plaintext
char string[12];
int val[5];
double a[3];
char *p[3];
```

(64 bit)
Multidimensional (Nested) Arrays

- Declaration: T A[R][C];
  - 2D array of data type T
  - R rows, C columns
  - T element requires K bytes

- Arrangement
  - Row-Major Ordering (C code)

Array Size: R * C * K bytes

4*R*C Bytes
Assumed Simple Cache

- 2 (4 byte) integers per block (8 bytes)
- 2-way set associative (2 blocks per set)
- 1 set
- Total size = 2 blocks, i.e., same as fully associative
- Replacement policy: Least Recently Used (LRU)
Some Key Questions

• How many elements are there per block?
• Does the data structure fit in the cache?
• Do I reuse blocks over time?
• In what order am I accessing blocks?
Simple Array

Cache

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
</tr>
</thead>
</table>

```
A[0] = A[1];
... = A[i];
```

- Miss rate = \#misses / \#accesses = \( \frac{N/2}{N} = \frac{1}{2} = 50\% \)
Simple Array with Outer Loop, Fits in Cache

- Assume array A fits in the cache
- Miss rate = #misses / #accesses = \( \frac{N/2}{NP} = \frac{1}{2P} \)

Lesson: for sequential accesses with re-use, if data fits in the cache, first visit suffers all the misses
Simple Array with Outer Loop, Doesn’t Fit in Cache

- Assume array A does not fit in the cache
- Miss rate = \#misses / \#accesses = \((N/2) / N = 1/2 = 50\%\)

Lesson: for sequential accesses with re-use, if the data doesn’t fit, same miss rate as no-reuse
2D Array, Fits in Cache

• Assume matrix A fits in the cache

• Miss rate = #misses / #accesses = \( \frac{(N^2/2)}{N^2} = \frac{1}{2} = 50\% \)

```python
def matrix_access(A, N):
    for i in range(N):
        for j in range(N):
            ... = A[i][j];
```

```
+---+---+---+---+
| 0 | 1 | 2 | 3 |
+---+---+---+---+
| 4 | 5 | 6 | 7 |
+---+---+---+---+
```
2D Array, Doesn’t Fit in Cache

- Assume matrix A does **not** fit in the cache
- Miss rate = #misses / #accesses = \( \frac{N^2/2}{N^2} = \frac{1}{2} = 50\% \)

**Lesson:** for 2D accesses, row-order, no re-use, whether data fits or not, same rate as sequential
2D Array, Column Order, Column Fits in Cache

- Assume matrix A fits in the cache
- Miss rate = \#misses / \#accesses = \( \frac{N^2/2}{N^2} = \frac{1}{2} = 50\% \)

Lesson: for 2D accesses, column-order, no re-use, when column data fits in cache, same rate as sequential
2D Array, Column Order, Column Doesn’t Fit in Cache

- Assume matrix A does not fit in the cache
- Miss rate = #misses / #accesses = \( \frac{N^2}{N^2} = 100\% \)

Lesson: for 2D accesses, column-order, no re-use, when column data doesn’t fit in cache, 100% miss rate
Loop Reordering/Interchange

- Initial code accesses data in column order
  - 100% miss rate, when column doesn’t fit in cache
- Loop reordering allows accessing data in row order
  - 50% miss rate

```c
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        A[j][i] = i * j;
    }
}
```

```c
for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
        A[j][i] = i * j;
    }
}
```
Matrix Multiplication

for (i=0; i<N; i++){
    for (j=0; j<N; j++){
        for (k=0; k<N; k++){
            ... = A[i][k] * B[k][j];
        }
    }
}
2 2D Arrays

- Assume matrix A and matrix B do not fit in cache
- Assume a column of B does not fit at same time as a row of A
- Miss rate = \#misses / \#accesses =

```c
for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        for (k=0;k<N;k++){
            ... = A[i][k] * B[k][j];
        }
    }
}
```
2 2D Arrays

for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        for (k=0;k<N;k++){
            ... = A[i][k] * B[k][j];
        }
    }
}

The inner most loop (i=j=0):
A[0][0] * B[0][0], A[0][1] * B[1][0],
A[0][2] * B[2][0], A[0][3] * B[3][0]

Cache

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time

1 2 3 4

A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

B

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
</tr>
</tbody>
</table>
2 2D Arrays

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        for (k=0; k<N; k++) {
            ... = A[i][k] * B[k][j];
        }
    }
}
```

The inner most loop (i=j=0):

- $A[0][0] * B[0][0]$
- $A[0][1] * B[1][0]$
- $A[0][2] * B[2][0]$
- $A[0][3] * B[3][0]$

---

**Cache Time**

<table>
<thead>
<tr>
<th>Cache</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
2 2D Arrays

for (i=0; i<N; i++)
for (j=0; j<N; j++)
for (k=0; k<N; k++)
    ... = A[i][k] * B[k][j];

The inner most loop (i=j=0):

A[0][0] * B[0][0], A[0][1] * B[1][0],
A[0][2] * B[2][0], A[0][3] * B[3][0]
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        for (k=0; k<N; k++) {
            ... = A[i][k] * B[k][j];
        }
    }
}

Next time: (i=0, j=1): 
A[0][0] * B[0][1], A[0][1] * B[1][1],
for (i=0; i< N; i++){
    for (j=0; j< N; j++){
        for (k=0; k< N; k++){
            ... = A[i][k] * B[k][j];
        }
    }
}

Next time: (i=0, j=1):
A[0][0] * B[0][1], A[0][1] * B[1][1],

A  |  1  |  2  |  3  |  4  |
---|-----|-----|-----|-----|
  | 1  |  2  |  3  |  4  |
  | 5  |  6  |  7  |  8  |
  | 9  | 10  | 11  | 12  |
  | 13 | 14  | 15  | 16  |

B  |  17 |  18 |  19 |  20 |
---|-----|-----|-----|-----|
  | 21 |  22 |  23 |  24 |
  | 25 |  26 |  27 |  28 |
  | 29 |  30 |  31 |  32 |

Cache   Time
---|---
29 | 30 |
25 | 26 |
  |   |
1  |  2 |
  |   |
3  |  4 |
  |   |
### 2D Arrays

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        for (k=0; k<N; k++) {
            ... = A[i][k] * B[k][j];
        }
    }
}
```
2D Arrays

for (i=0; i<\text{N}; i++){
    for (j=0; j<\text{N}; j++){
        for (k=0; k<\text{N}; k++){
            ... = A[i][k] \times B[k][j];
        }
    }
}

Next time: (i=0, j=1):

\[ A[0][0] \times B[0][1], A[0][1] \times B[1][1], A[0][2] \times B[2][1], A[0][3] \times B[3][1] \]
2 2D Arrays

for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        for (k=0;k<N;k++){
            \[ \ldots = A[i][k] \times B[k][j]; \]
        }
    }
}

Next time: (i=0, j=1):

\[ A[0][0] \times B[0][1], A[0][1] \times B[1][1], \]
\[ A[0][2] \times B[2][1], A[0][3] \times B[3][1] \]
2 2D Arrays

for (i=0; i<N; i++){
    for (j=0; j<N; j++){
        for (k=0; k<N; k++){
            ... = A[i][k] * B[k][j];
        }
    }
}

Next time: (i=0, j=1):

A[0][0] * B[0][1], A[0][1] * B[1][1],
• Assume matrix A and matrix B do not fit in cache
• Assume a column of B does not fit at same time as a row of A
• Miss rate = #misses / #accesses = 75%

```c
for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        for (k=0;k<N;k++){
            ...
            = A[i][k] * B[k][j];
        }
    }
}
Next time: (i=0, j=1):

A[0][0] * B[0][1], A[0][1] * B[1][1],
```
Matrix Multiplication

/* Multiply N x N matrices a and b */

```c
mmm(double a[][N], double b[][N], double c[][N])
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k]*b[k][j];
}
```

![Matrix multiplication diagram]
Cache Miss Analysis, First Iteration

- **Assume:**
  - Matrix elements are doubles
  - Cache block 64B = 8 doubles
  - Cache capacity << n (much smaller than n)
    - i.e., can’t even hold an entire row in the cache!

- **On first iteration:**
  - How many misses?
  - \( \frac{n}{8} + n = \frac{9n}{8} \) misses

- **At the end of first iteration:**
  - in cache
Cache Miss Analysis, Second Iteration

- Assume:
  - Matrix elements are doubles
  - Cache block 64B = 8 doubles
  - Cache capacity << n (much smaller than n)
    - i.e., can’t even hold an entire row in the cache!

- On second iteration:
  - How many misses?
  - \( \frac{n}{8} + n = \frac{9n}{8} \) misses

- Total misses for matrix multiplication:
  - \( \left( \frac{9n}{8} \right) \times n^2 = \left( \frac{9}{8} \right) \times n^3 \)
Improving Cache Reuse

• Misses are expensive
  • L1 cache reference: 1-4 ns (L1 cache size: 32 KB)
  • Main memory reference: 100 ns (memory size: 4-256 GBs)

• Matrix multiplication has lots of data re-use
  • Key idea: Try to use entire cache block once it is loaded
  • Challenge: We need to work with both rows and columns

• Solution:
  • Operate in sub-squares of the matrices
    • One sub-square per matrix should fit in cache simultaneously
    • Heavily re-use the sub-squares before loading new ones
  • Called Tiling or Blocking (a sub-square is a tile)
Idea for Tiled Matrix Multiplication

- Perform tile based mini-multiplications
  - E.g., $TA_{00} \times TB_{00}$, $TA_{00} \times TB_{01}$
  - $C[0][0] = 1 \times 16 + 2 \times 15 + 3 \times 14 + 4 \times 13$
    - Perform as part of $TA_{00} \times TB_{00}$
    - Perform as part of $TA_{01} \times TB_{10}$
Tiled Matrix Multiplication

/* Multiply N x N matrices a and b */
void mmm(double a[][N], double b[][N], double c[][N], int T) {
    int i, j, k;
    for (i = 0; i < n; i+=T)
        for (j = 0; j < n; j+=T)
            for (k = 0; k < n; k+=T)
                /* T x T mini matrix multiplications */
                for (i1 = i; i1 < i+T; i1++)
                    for (j1 = j; j1 < j+T; j1++)
                        for (k1 = k; k1 < k+T; k1++)
                            c[i1][j1] += a[i1][k1]*b[k1][j1];
}

Tile size T x T
• First calculate $\text{TC}_{00} = C[0][0] - C[T-1][T-1]$
• Next calculate $TC_{01} = C[0][T] - C[T-1][2T-1]$
Why Does Tiling Work?

- \( C[0][0] = 1 \times 16 + 2 \times 15 \)
  - Perform as part of \( TA_{00} \times TB_{00} \)
  - 3 \times 14 + 4 \times 13
  - Perform as part of \( TA_{01} \times TB_{10} \)
- Still have to access \( B[\_] \) column-wise
- But now \( B[\_] \)'s cache blocks don't get replaced
Tiling Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache capacity << n (much smaller than n)
  - With 3 arrays, need to fit 3 tiles in cache
    - Cache capacity > $3T^2$

- **Misses per tile-iteration:**
  - $T^2/8$ misses for each tile
  - $(2n/T) \times (T^2/8) = nT/4$

- **Total misses:**
  - Tiled: $(nT/4) \times (n/T)^2 = (1/(4T)) \times n^3$
  - Untiled: $(9/8) \times n^3$