CS 61C: Great Ideas in Computer Architecture

*Performance Programming, Technology Trends*

*Instructor:* Justin Hsia
Review of Last Lecture

• Cache performance measured in $\text{CPI}_{\text{stall}}$ and AMAT
  – Parameters that matter: HT, MR, MP
• Multilevel caches reduce miss penalty
  – Standard to have 2-3 cache levels (and split I$/D$)
  – Makes CPI/AMAT calculations more complicated
• Set associativity reduces miss rate
  – Index field now determines set
• Cache design choices change performance parameters and cost
Question: How many total bits are stored in the following cache?

- 4-way set associative cache
- Cache size 1 KiB, Block size 16 B
- Write-back
- 16-bit address space

- $2^6 \times (2^7 + 2^3 + 2^1) = 8.625 \text{ Kib}$
- $2^4 \times (2^7 + 2^3 + 2^0) = 2.140625 \text{ Kib}$
- $2^4 \times (2^7 + 2^3 + 2^1) = 2.15625 \text{ Kib}$
- $2^4 \times (2^7 + 6 + 2^1) = 2.125 \text{ Kib}$
Question: (previous midterm question)
Which of the following cache changes will \textit{definitely} increase L1 Hit Time?
(more than one may be correct)

- Adding unified L2$, which is larger than L1 but smaller than memory
- Increasing block size while keeping cache size constant
- Increasing associativity while keeping cache size constant
- Increasing cache size while keeping block size constant
Agenda

• Performance Programming
• Administrivia
• Perf Prog: Matrix Multiply
• Technology Trends
  – The Need for Parallelism
Performance Programming

• Adjust memory accesses in *code* (software) to improve miss rate

• With understanding of how caches work, can revise programs to improve cache utilization
  – Cache size
  – Block size
  – Multiple levels
Performance of Loops and Arrays

• Array performance often limited by memory speed

• **Goal:** Increase performance by minimizing traffic from cache to memory
  – Reduce your miss rate by getting better reuse of data already in the cache
  – It is okay to access memory in different orderings as long as you still end up with the correct result

• **Cache Blocking:** “shrink” the problem by performing multiple iterations on smaller chunks that “fit well” in your cache
  – Use Matrix Multiply as an example (Lab 6 and Project 2)
Ex: Looping Performance (1/5)

• We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)

• What is will our miss rate be for a D$ with 1-word blocks? (array not in $ at start)
  – 100% MR because each array element (word) accessed just once

• Can code choices change this?
  – No
Ex: Looping Performance (2/5)

• We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)

• Now for a D$ with 2-word blocks, what are the best and worst miss rates we can achieve?
  – Best: 50% MR via standard incrementation (each block will miss then hit)
  – Code:
    
    ```
    for(int i=0; i<1024; i++) A[i]++;  
    ```
Ex: Looping Performance (3/5)

- We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)

- Now for a D$ with 2-word blocks, what are the best and worst miss rates we can achieve?
  - Worst: 100% MR by skipping elements (assuming D$ smaller than half of array size)
  - Code:
    ```
    for(int i=0; i<1024; i+=2) A[i]++;  
    for(int i=1; i<1024; i+=2) A[i]++;  
    ```
Ex: Looping Performance (4/5)

- We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)
- What does the increment operation look like in assembly?

```assembly
# A → $s0
lw $t0, 0($s0)
addiu $t0, $t0, 1
sw $t0, 0($s0)
addiu $s0, $s0, 4
```

This value may change depending on your loop in code.
Ex: Looping Performance (5/5)

• We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)

• For an I$ with 1-word blocks, what happens if we don’t use labels/loops?
  – 100% MR, as all instructions are explicitly written out sequentially

• What if we loop by incrementing `i` by 1?
  – Will miss on first pass over code, but should be found in I$ for all subsequent passes
Agenda

• Performance Programming

• Administrivia

• Perf Prog: Matrix Multiply

• Technology Trends
  – The Need for Parallelism
Administrivia

• HW3 due Wednesday
• Midterm: Friday @ 9am in 245 Li Ka Shing
  – Take old exams for practice (see Piazza post @209)
  – One-sided sheet of handwritten notes
  – MIPS Green Sheet provided; no calculators
  – Will cover up through caches
• Mid-Semester Survey (part of Lab 7)
• Justin’s OH this week: Wed 5-7pm, Room TBD
Agenda

• Performance Programming
• Administrivia
• Perf Prog: Matrix Multiply
• Technology Trends
  – The Need for Parallelism
Matrix Multiplication

\[ C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \]

\[ C = A \times B \]

\[ a_{i*} \]

\[ b_{*j} \]
Naïve Matrix Multiply

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];

**Advantage:** Code simplicity

**Disadvantage:** Blindly marches through memory and caches
Matrices in Memory (1/2)

- Matrices stored as 1-D arrays in memory
  - Column major: $A(i,j)$ at $A+i+j*n$
  - Row major: $A(i,j)$ at $A+i*n+j$
- C default is row major

<table>
<thead>
<tr>
<th>Column major:</th>
<th>Row major:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5 10 15</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>1 6 11 16</td>
<td>4 5 6 7</td>
</tr>
<tr>
<td>2 7 12 17</td>
<td>8 9 10 11</td>
</tr>
<tr>
<td>3 8 13 18</td>
<td>12 13 14 15</td>
</tr>
<tr>
<td>4 9 14 19</td>
<td>16 17 18 19</td>
</tr>
</tbody>
</table>
Matrices in Memory (2/2)

• How do cache blocks fit into this scheme?
  – Column major matrix in memory:

ROW of matrix (blue) is spread among cache blocks shown in red
Naïve Matrix Multiply (cache view)

\[
\text{# implements } C = C + A \cdot B \\
\text{for } i = 1 \text{ to } n \\
\quad \text{# read row } i \text{ of } A \text{ into cache} \\
\quad \text{for } j = 1 \text{ to } n \\
\qquad \text{# read } c(i,j) \text{ into cache} \\
\qquad \text{# read column } j \text{ of } B \text{ into cache} \\
\qquad \text{for } k = 1 \text{ to } n \\
\qquad \quad c(i,j) = c(i,j) + a(i,k) \cdot b(k,j) \\
\quad \text{# write } c(i,j) \text{ back to main memory}
\]

\[
\begin{align*}
C(i,j) &= C(i,j) + A(i,:) \cdot B(:,j)
\end{align*}
\]
Linear Algebra to the Rescue (1/2)

• Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (“blocks”)

• For example, multiply two 4×4 matrices:

\[
A = \begin{bmatrix}
    a_{11} & a_{12} & a_{13} & a_{14} \\
    a_{21} & a_{22} & a_{23} & a_{24} \\
    a_{31} & a_{32} & a_{33} & a_{34} \\
    a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
= \begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix}, \text{ with } B \text{ defined similarly.}
\]

\[
AB = \begin{bmatrix}
    (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\
    (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22})
\end{bmatrix}
\]
Matrices of size $N \times N$, split into 4 blocks of size $r$ ($N=4r$).

$$ C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} \cdot B_{k2} $$

- Multiplication operates on small “block” matrices
  - Choose size so that they fit in the cache!
Blocked Matrix Multiply

• Blocked version of the naïve algorithm:

\[
\text{for}(i=0; \ i<N/r; \ i++) \\
\quad \text{for}(j=0; \ j<N/r; \ j++) \\
\quad \quad \text{for}(k=0; \ k<N/r; \ k++) \\
\quad \quad \quad C[i][j] += A[i][k]*B[k][j]
\]

- \( r \times r \) matrix addition
- \( r \times r \) matrix multiplication

- \( r = \) block size (assume \( r \) divides \( N \))
- \( X[i][j] = \) submatrix of \( X \), defined by block row \( i \) and block column \( j \)
Blocked Matrix Multiply (cache view)

# implements $C = C + A \times B$
for $i = 1$ to $N$
  for $j = 1$ to $N$
    # read block $C(i,j)$ into cache
    for $k = 1$ to $N$
      # read block $A(i,k)$ into cache
      # read block $B(k,j)$ into cache
      $C(i,j) = C(i,j) + A(i,k) \times B(k,j)$
    # write block $C(i,j)$ back to main memory
Matrix Multiply Comparison

• Naïve Matrix Multiply
  – \( N = 100,1000 \) cache blocks, 1 word/block
  – Youtube: Slow/Fast-forward
  – \( \approx 1,020,0000 \) cache misses

• Blocked Matrix Multiply
  – \( N = 100,1000 \) cache blocks, 1 word/block, \( r = 30 \)
  – Youtube: Slow/Fast-forward
  – \( \approx 90,000 \) cache misses
Maximum Block Size

• Blocking optimization only works if the blocks fit in cache
  – Must fit 3 blocks of size $r \times r$ in memory (for A, B, and C)

• For cache of size $M$ (in elements/words), we must have $3r^2 \approx M$, or $r \approx \sqrt{M/3}$

• Ratio of cache misses unblocked vs. blocked up to $\approx \sqrt{M}$ (play with sizes to optimize)
  – From comparison: ratio was $\approx 11$, $\sqrt{M} = 31.6$
Get to Know Your Staff

• Category: **Food**
Agenda

• Performance Programming
• Administrivia
• Perf Prog: Matrix Multiply
• Technology Trends
  – The Need for Parallelism
Six Great Ideas in Computer Architecture

1. Layers of Representation/Interpretation
2. Moore’s Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy
Technology Cost over Time

What does improving technology look like?

Cost

$ 

Time

A

B

C

D
Tech Cost: Successive Generations

How Can Tech Gen 2 Replace Tech Gen 1?

- Tech Gen 1
- Tech Gen 2
- Tech Gen 2?
- Tech Gen 3

Time

Cost $
Tech Performance over Time

Performance

Time
Moore’s Law

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. ...That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.” *(from 50 in 1965)*

Gordon Moore, “Cramming more components onto integrated circuits,” *Electronics*, Volume 38, Number 8, April 19, 1965

“Integrated circuits will lead to such wonders as home computers--or at least terminals connected to a central computer--automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.”
Great Idea #2: Moore’s Law

Predicts: Transistor count per chip doubles every 2 years

Gordon Moore
Intel Cofounder
B.S. Cal 1950
Memory Chip Size

Growth in memory capacity slowing

4x in 3 years

2x in 3 years
End of Moore’s Law?

• It’s also a law of investment in equipment as well as increasing volume of integrated circuits that need more transistors per chip
• Exponential growth cannot last forever
• More transistors/chip will end during your careers
  — 2020? 2025?
  — (When) will something replace it?
Uniprocessor Performance

Improvements in processor performance have slowed. Why?
Computer Technology: Growing, But More Slowly

- **Processor**
  - Speed 2x / 1.5 years (since ’85-’05) \([slowing!]\)
  - Now +2 cores / 2 years
  - When you graduate: 3-4 GHz, 6-8 Cores in client, 10-14 in server

- **Memory (DRAM)**
  - Capacity: 2x / 2 years (since ’96) \([slowing!]\)
  - Now 2X/3-4 years
  - When you graduate: 8-16 GigaBytes

- **Disk**
  - Capacity: 2x / 1 year (since ’97)
  - 250X size last decade
  - When you graduate: 6-12 TeraBytes

- **Network**
  - Core: 2x every 2 years
  - Access: 100-1000 mbps from home, 1-10 mbps cellular
Limits to Performance: Faster Means More Power
Dynamic Power

• Power = \( C \times V^2 \times f \)
  \( \text{— Proportional to capacitance, voltage}^2, \text{and} \) frequency of switching

• What is the effect on power consumption of:
  \( \text{— “Simpler” implementation (fewer transistors)?} \quad \downarrow \)
  \( \text{— Reduced voltage?} \quad \downarrow \downarrow \)
  \( \text{— Increased clock frequency?} \quad \uparrow \)
### Multicore Helps Energy Efficiency

- **Power** = \( C \times V^2 \times f \)

---

**Table: Comparative Analysis**

<table>
<thead>
<tr>
<th></th>
<th>In the same process technology...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core</strong></td>
<td><strong>Core</strong></td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td><strong>Cache</strong></td>
</tr>
</tbody>
</table>

**Comparison**:

- **Voltage**: 1 vs. -15%
- **Freq**: 1 vs. -15%
- **Area**: 1 vs. 2
- **Power**: 1 vs. 1
- **Perf**: 1 vs. ~1.8

*From: William Holt, HOT Chips 2005*
Transition to Multicore
Parallelism - The Challenge

• Only path to performance is parallelism
  – Clock rates flat or declining

• Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  – Scheduling, load balancing, time for synchronization, overhead for communication

• Project #2: fastest matrix multiply code on 16 processor (cores) computers
Summary

• Performance programming
  – With understanding of your computer’s architecture, can optimize code to take advantage of your system’s cache
  – Especially useful for loops and arrays
  – “Cache blocking” will improve speed of Matrix Multiply with appropriately-sized blocks

• Processors have hit the power wall, the only option is to go parallel