CS 61C:
Great Ideas in Computer Architecture
Cache Performance

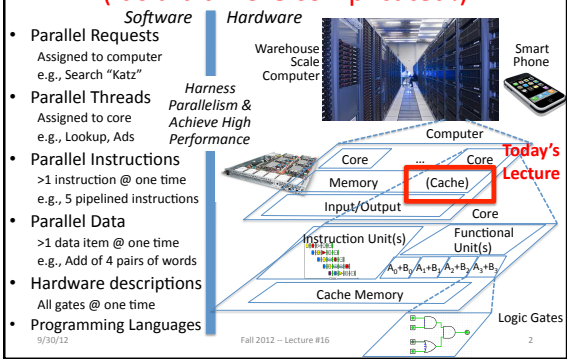
Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

New-School Machine Structures (It's a bit more complicated!)

Software
- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance

Hardware



Review

- Write-through versus write-back caches
- AMAT = Hit time + Miss rate x Miss penalty
- Larger caches reduce Miss rate via Temporal and Spatial Locality, but can increase Hit time
- Multilevel caches help Miss penalty

Caches are software-invisible

- Load and store instructions just access large memory (32-bit addresses in MIPS); hardware automatically moves data in and out of cache
- Even if programmer writes applications not knowing about caches, we observe temporal and spatial locality in memory accesses
- Performance improves (over no caches) even when programmer unaware of cache's existence

CPI/Miss Rates/DRAM Access
SpecInt2006 on AMD Barcelona (64KB L1, 512KB L2)

		Data Only	Data Only	Instructions and Data
Name	CPI	L1 D cache misses/1000 instr	L2 D cache misses/1000 instr	DRAM accesses/1000 instr
perl	0.75	3.5	1.1	1.3
bzip2	0.85	11.0	5.8	2.5
gcc	1.72	24.3	13.4	14.8
mcf	10.00	106.8	88.0	88.5
go	1.09	4.5	1.4	1.7
hmmer	0.80	4.4	2.5	0.6
sjeng	0.96	1.9	0.6	0.8
libquantum	1.61	33.0	33.1	47.7
h264avc	0.80	8.8	1.6	0.2
omnetpp	2.94	30.9	27.7	29.8
astar	1.79	16.3	9.2	8.2
xalancbmk	2.70	38.0	15.8	11.4
Median	1.35	13.6	7.5	5.4

Performance Programming:
Adjust software accesses to improve miss rate

- Now that understand how caches work, can revise program to improve cache utilization
  - Cache size
  - Block size
  - Multiple levels
- "Cache-Aware" performance optimizations
  - but code would still work even if no caches present

Performance of Loops over Arrays

- Array performance often limited by memory speed
- OK to access memory in different order as long as get correct result
- Goal: Increase performance by minimizing traffic from cache to memory
  - That is, reduce Miss rate by getting better reuse of data already in cache

Alternate Matrix Layouts in Memory

- A matrix is a 2-D array of elements, but memory addresses are "1-D" (0…MaximumMemoryAddress)
- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j*n
  - by row, or "row major" (C default) A[i][j] at A+i*n+j

Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

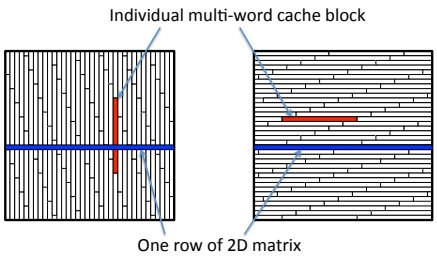
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

How a 4x5 Matrix is stored in memory, red numbers are memory addresses

Cache Blocks* in Matrix

Column Major

Row Major



*Cache Line is alternative name for Cache Entry or Block

Loop Interchange: Flashcard quiz

```
for(j=0; j < N; j++) {
  for(i=0; i < M; i++) {
     x[i][j] = 2 * x[i][j];
  }
}
```

⇩

```
for(i=0; i < M; i++) {
  for(j=0; j < N; j++) {
     x[i][j] = 2 * x[i][j];
  }
}
```

What kind of locality does this improve?

Spatial

Temporal

Both

Neither

Loop Fusion: Flashcard Quiz

```
for(i=0; i < N; i++)
     a[i] = b[i] * c[i];

for(i=0; i < N; i++)
     d[i] = a[i] * c[i];
```

⇩

```
for(i=0; i < N; i++)
{
     a[i] = b[i] * c[i];
     d[i] = a[i] * c[i];
}
```

What kind of locality does this improve?

Spatial

Temporal

Both

Neither

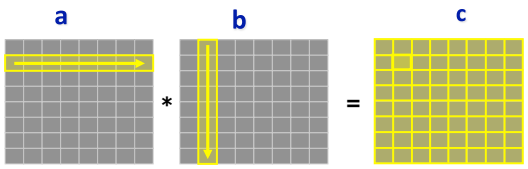
Administrivia

- Lab #6: More MIPS
- Project 2b: MIPS Emulator, due Sunday
- Midterm, a week from Tuesday

Cache Blocking (aka Cache Tiling)

- "shrink" problem by performing multiple iterations within smaller cache blocks
- Also known as cache *tiling*
- Don't confuse term "cache blocking" with:
  - cache blocks, i.e., individual cache entries or lines
  - (or later, blocking versus non-blocking caches)
- Use Matrix Multiply as example: Next Lab and Project 3

Matrix Multiplication



The simplest algorithm

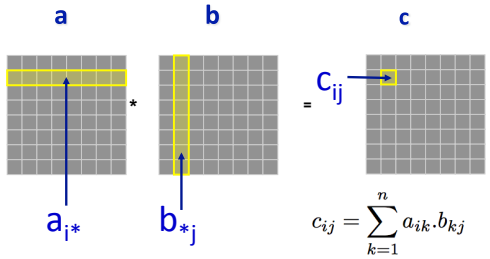
Assumption: the matrices are stored as 2-D NxN arrays

```
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: Marches through memory and caches

Matrix Multiplication

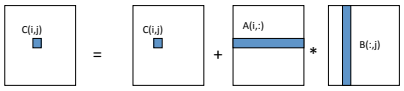


Simple Matrix Multiply - www.youtube.com/watch?v=yl0LTcDIhxc

100 x 100 Matrix, Cache 1000 blocks, 1 word/block

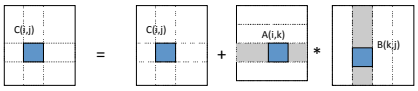
Improving reuse via Blocking: 1st "Naïve" Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
 {read row i of A into cache}
 for j = 1 to n
    {read c(i,j) into cache}
    {read column j of B into cache}
    for k = 1 to n
       c(i,j) = c(i,j) + a(i,k) * b(k,j)
    {write c(i,j) back to main memory}
```



Blocked Matrix Multiply

```
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is called
    the block size
    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) * B(k,j) {do a matrix multiply on blocks}
     {write block C(i,j) back to main memory}
```



Blocked Matrix Multiply - www.youtube.com/watch?v=IFWgwGMMrh0

100 x 100 Matrix, 1000 cache blocks, 1 word/block, block 30x30

Blocked Algorithm

- The blocked version of the i-j-k algorithm is written simply as (A,B,C are submatricies of a, b, c)

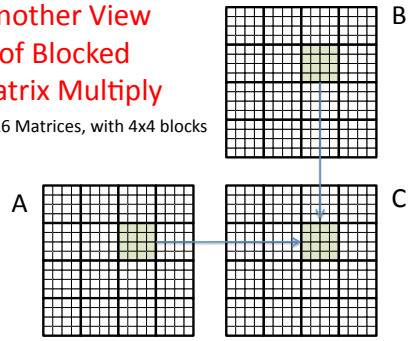
```
for (i=0;i<N/r;i++)
  for (j=0;j<N/r;j++)
    for (k=0;k<N/r;k++)
      C[i][j] += A[i][k]*B[k][j]
```

r x r matrix addition | r x r matrix multiplication

  - r = block (sub-matrix) size (Assume r divides N)
  - X[i][j] = a sub-matrix of X, defined by block row i and block column j

Another View of Blocked Matrix Multiply

16x16 Matrices, with 4x4 blocks



Maximum Block Size

- The blocking optimization works only if the blocks fit in cache.
- That is, 3 blocks of size r x r must fit in memory (for A, B, and C)
- M = size of cache (in elements/words)
- We must have: $3r^2 \approx M$, or $r \approx \sqrt{(M/3)}$
- Ratio of cache misses blocked vs. unblocked up to $\approx \sqrt{M}$

Simple Matrix Multiply Whole Thing - www.youtube.com/watch?v=f3-z6t_xIyw

1x1 blocks: 1,020,000 misses: read A once, read B 100 times, read C once

Blocked Matrix Multiply Whole Thing - www.youtube.com/watch?v=tgpmXX3xOrk

30x30 blocks: 90,000 misses = read A and B four times, read C once "Only" 11X vs 30X Matrix small enough that row of A in simple version fits completely in cache (+ few odds and ends)

Sources of Cache Misses (3 C's)

- *Compulsory* (cold start, first reference):
  - 1st access to a block, "cold" fact of life, not a lot you can do about it.
    - If running billions of instructions, compulsory misses are insignificant
- *Capacity*:
  - Cache cannot contain all blocks accessed by the program
- *Conflict* (collision):
  - Multiple memory locations mapped to the same cache location

Flashcard Quiz: With a fixed cache capacity, what effect does a larger block size have on the 3Cs?

Decreases compulsory, increases conflicts
Increases conflicts
Increases compulsory, decreases conflicts
Decreases conflicts

Flashcard Quiz: With a fixed cache block size, what effect does a larger cache capacity have on the 3Cs?

Increases compulsory, decreases conflicts
Increases conflicts, decreases capacity misses
Decreases compulsory, decreases conflicts
Decreases conflicts, decreases capacity misses

…and in Conclusion

- Although caches are software-invisible, a "cache-aware" performance programmer can improve performance by large factors by changing order of memory accesses
- Three C's of cache misses
  - Compulsory
  - Capacity
  - Conflict