CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Caches II

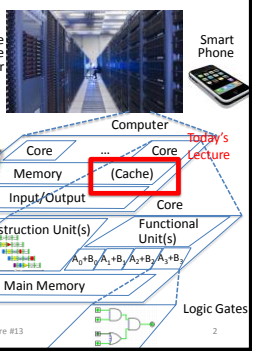
Instructor:
Michael Greenbaum

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

Harness Parallelism & Achieve High Performance

Hardware



Review: Direct Mapped Cache Layout

- 8 bit address space, 32 byte cache with 8 byte (2 word) blocks.
- Offset – 3 bits, Index – 2 bits, Tag – 3 bits

Offset

Index	V	Tag	000	001	010	011	100	101	110	111
00										
01										
10										
11										

- Remember, MIPS operates with words (4 bytes), so the only offsets we'll see in a MIPS system are multiples of 4. Entire words will be transferred to and from the CPU.

Agenda

- Cache Reads and Writes, Consistency
- More Cache Performance
- Administrivia
- Set Associative Caches
- Break
- Set Associative Caches (Cont'd)

Handling Cache Misses (Single Word Blocks)

- Read misses (I$ and D$)
  - Stall execution, fetch the block from the next level in the memory hierarchy, install it in the cache, send requested word to processor, and then let execution resume
- Write misses (D$ only)
  - Write allocate: Stall execution, fetch the block from next level in the memory hierarchy, install it in cache, write the word from processor to cache, also update memory, then let execution resume

  or
  - No-write allocate: skip the cache write and just write the word to memory

Cache-Memory Consistency? (1/2)

- Need to make sure cache and memory are consistant (know about all updates)

1) Write-Through Policy: write cache and write *through* the cache to memory
   - Every write eventually gets to memory
   - Too slow, so include Write Buffer to allow processor to continue once data in Buffer, Buffer updates memory in parallel to processor

Cache-Memory Consistency? (2/2)

- Need to make sure cache and memory are consistant (know about all updates)

2) Write-Back Policy: write only to cache and then write cache block *back* to memory when evict block from cache
  - Writes collected in cache, only single write to memory per block
  - Include bit to see if wrote to block or not, and then only write back if bit is set
    - Called "Dirty" bit (writing makes it "dirty")

Agenda

- Cache Reads and Writes, Consistency
- More Cache Performance
- Administrivia
- Set Associative Caches
- Break
- Set Associative Caches (Cont'd)

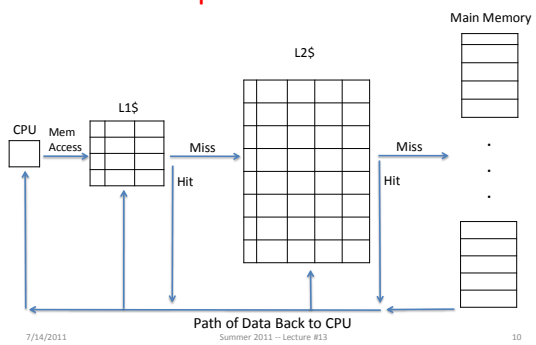
Recall: Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses

AMAT = Time for a hit + Miss rate x Miss penalty

- How reduce Miss Penalty?

Multiple Cache Levels



Multiple Cache Levels

- With advancing technology, have more room on die for bigger L1 caches and for second level cache – normally a unified L2 cache (i.e., it holds both instructions and data,) and in some cases even a unified L3 cache
- New AMAT Calculation:

AMAT = L1 Hit Time + L1 Miss Rate * **L1 Miss Penalty**
L1 Miss Penalty = L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

and so forth (final miss penalty is Main Memory access time)

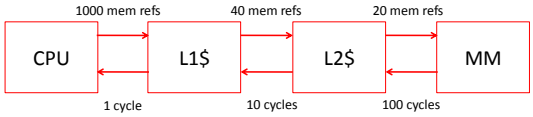
New AMAT Example

- 1 cycle L1 Hit Time, 2% L1 Miss Rate, 5 cycle L2 Hit Time, 5% L2 Miss Rate.
- 100 cycle Main Memory access time
- No L2 Cache:
  AMAT = 1 + .02*100 = 3
- With L2 Cache:
  AMAT = 1 + .02*(5 + .05*100) = 1.2!

Local vs. Global Miss Rates

- ***Local miss rate*** – the fraction of references to one level of a cache that miss
- Local Miss rate L2$ = $L2 Misses / L1$ Misses
- ***Global miss rate*** – the fraction of references that miss out of all accesses in system.
  - L2$ local miss rate >> than the global miss rate
- Global Miss rate = L2$ Misses / Total Accesses
  = L2$ Misses / L1$ Misses x L1$ Misses / Total Accesses
  = Local Miss rate L2$ x Local Miss rate L1$
- AMAT Calculation used **Local Miss Rate**.

Memory Hierarchy with Two Cache Levels



- For every 1000 CPU to memory references
  - 40 will miss in L1$; what is the miss rate?
  - 20 will miss in L2$; what is the miss rate?
  - Global vs. local miss rate?

CPI_{stalls} Calculation

- Assume
  - CPI_{ideal} of 2
  - 100 cycle miss penalty to main memory
  - 25 cycle miss penalty to Unified L2$
  - 36% of instructions are load/stores
  - 2% L1 I$ miss rate; 4% L1 D$ miss rate
  - 0.5% **global** U(nified)L2$ miss rate
- CPI_{Total} = 2 + 1x.02×25 + .36×.04×25 (L1)
  + 1x.005×100 + .36×.005×100 (L2)
  = 3.54 (vs. 5.44 with no L2$)

IFetch
Ld/St

Design Considerations

- Different design considerations for L1$ and L2$
  - L1$ focuses on fast access: minimize hit time to achieve shorter clock cycle, e.g., smaller $
  - L2$, L3$ focus on low miss rate: reduce penalty of long main memory access times: e.g., Larger $ with larger block sizes/higher levels of associativity
- Miss penalty of L1$ is significantly reduced by presence of L2$, so can be smaller/faster even with higher miss rate
- For the L2$, fast hit time is less important than low miss rate
  - L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate >> than the global miss rate

Agenda

- Cache Reads and Writes, Consistency
- More Cache Performance
- Administrivia
- Set Associative Caches
- Break
- Set Associative Caches (Cont'd)

Administrivia

- Midterm
  - Friday 7/15, 9am-12pm, 2050 VLSB
  - How to study:
    - Studying in groups can help.
    - Take old exams for practice (link at top of main webpage)
    - Look at lectures, section notes, projects, hw, labs, etc.
  - Will cover up to today's material.
- Mid-Session Survey
  - Short survey to complete as part of Lab 7.
  - Let us know how we're doing, and what we can do to improve!

Agenda

- Cache Reads and Writes, Consistency
- More Cache Performance
- Administrivia
- Set Associative Caches
- Break
- Set Associative Caches (Cont'd)

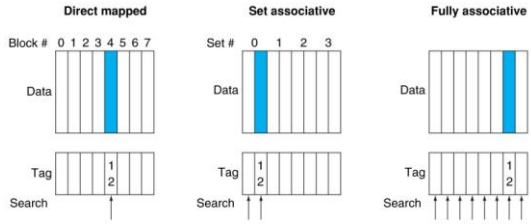
Sources of Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - **Solution 2: increase associativity (may increase access time)**

Reducing Cache Misses

- Allow more flexible block placement in cache
- *Direct mapped $*: memory block maps to exactly one cache block
- *Fully associative $*: allow a memory block to be mapped to any cache block
- Compromise: divide $ into sets, each of which consists of n "ways" (*n-way set associative*) to place memory block
  - Memory block maps to unique set determined by index field and is placed in any of the n-ways of that set
  - Calculation: (block address) modulo (# sets in the cache)

Alternative Block Placement Schemes



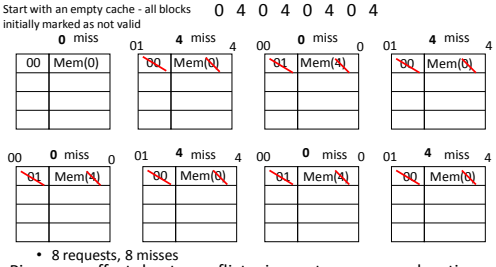
- DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- FA placement: mem block 12 can appear in any cache blocks

Example: 4-Word Direct-Mapped $ Worst-Case Reference String

- Consider the sequence of memory accesses

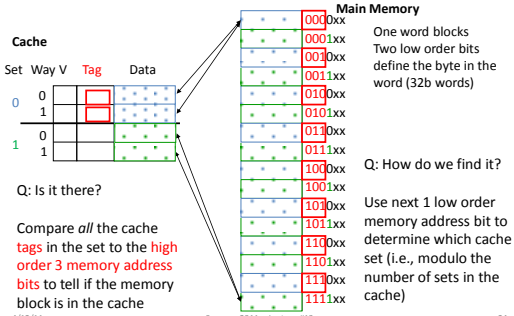
Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4



- 8 requests, 8 misses
- Ping pong effect due to conflict misses - two memory locations that map into the same cache block

Example: 2-Way Set Associative $ (4 words = 2 sets x 2 ways per set)



One word blocks
Two low order bits define the byte in the word (32b words)

Q: How do we find it?

Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

Q: Is it there?

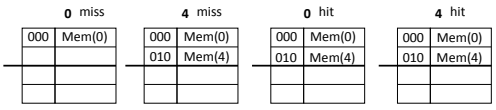
Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

Example: 4-Word 2-Way SA $ Same Reference String

- Consider the sequence of memory accesses

Start with an empty cache - all blocks initially marked as not valid

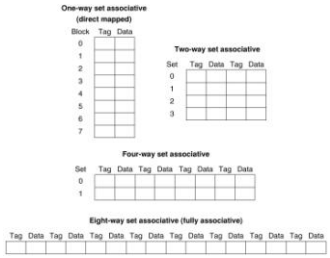
0 4 0 4 0 4 0 4



- 8 requests, 2 misses

- Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

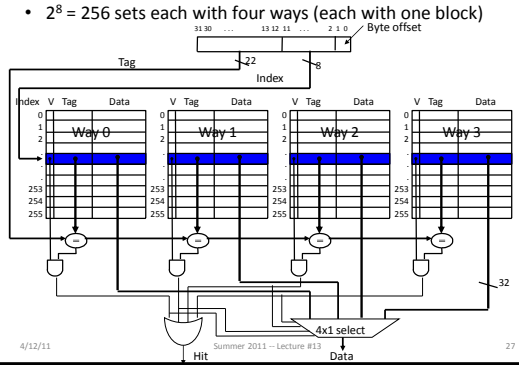
Example: Eight-Block Cache with Different Organizations



Total size of $ in blocks is equal to *number of sets* x *associativity*. For fixed $ size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $ is same as a fully associative $.

Four-Way Set-Associative Cache

- 2^8 = 256 sets each with four ways (each with one block)



Peer Instruction

- 32 bit address space, 32KB 4-way set associative cache with 8 word blocks. What is the TIO breakdown?

Blue)	**T - 21**	**I - 8**	**O - 3**
Red)	**T - 19**	**I - 10**	**O - 3**
Yellow)	**T - 17**	**I - 12**	**O - 3**
Purple)	**T - 19**	**I - 8**	**O - 5**
Green)	**T - 17**	**I - 10**	**O - 5**
White)	**T - 15**	**I - 12**	**O - 5**

Peer Instruction

- 32 bit address space, 32KB 4-way set associative cache with 8 word blocks. What is the TIO breakdown?

8 word blocks => 32 bytes / block => O = 5
32 KB / (32 bytes / block) = 2^10 blocks total
2^10 blocks / (4 blocks / set) = 2^8 sets total
Index bits will index into sets => I = 8

Blue)	**T - 21**	**I - 8**	**O - 3**
Red)	**T - 19**	**I - 10**	**O - 3**
Yellow)	**T - 17**	**I - 12**	**O - 3**
Purple)	**T - 19**	**I - 8**	**O - 5**
Green)	**T - 17**	**I - 10**	**O - 5**
White)	**T - 15**	**I - 12**	**O - 5**

Agenda

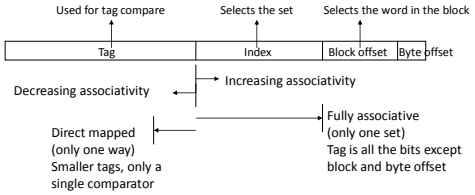
- Cache Reads and Writes, Consistency
- More Cache Performance
- Administrivia
- Set Associative Caches
- Break
- Set Associative Caches (Cont'd)

Agenda

- Cache Reads and Writes, Consistency
- More Cache Performance
- Administrivia
- Set Associative Caches
- Break
- Set Associative Caches (Cont'd)

Range of Set-Associative Caches

- For a fixed-size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

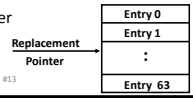


Costs of Set-Associative Caches

- When miss occurs, which way's block selected for replacement?
  - Least Recently Used (LRU): one that has been unused the longest
    - Must track when each way's block was used relative to other blocks in the set
    - For 2-way SA $, one bit per set → set to 1 when a block is referenced; reset the other way's bit (i.e., "last used")
- N-way set-associative cache costs
  - N comparators for tag comparisons
  - Must choose appropriate set (multiplexer) before data is available

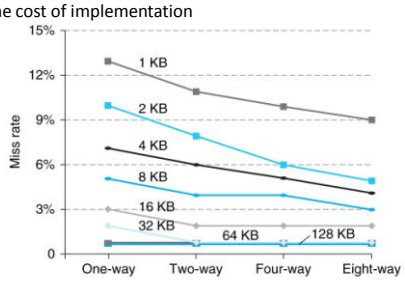
Cache Block Replacement Policies

- Random Replacement
  - Hardware randomly selects a cache item and throw it out
- Least Recently Used
  - Hardware keeps track of access history
  - Replace the entry that has not been used for the longest time
  - For 2-way set-associative cache, need one bit for LRU replacement
- Example of a Simple "Pseudo" LRU Implementation
  - Assume 64 Fully Associative entries in a set.
  - Hardware replacement pointer points to one cache entry
  - Whenever access is made to the entry the pointer points to:
    - Move the pointer to the next entry
  - Otherwise: do not move the pointer



Benefits of Set-Associative Caches

- Choice of DM $ or SA $ depends on the cost of a miss versus the cost of implementation

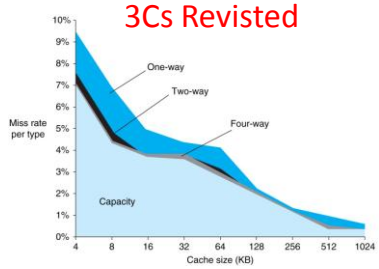


- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

How to Calculate 3C's using Cache Simulator

1. *Compulsory*: set cache size to infinity and fully associative, and count number of misses
2. *Capacity*: Change cache size from infinity, usually in powers of 2, and count misses for each reduction in size
   - 16 MB, 8 MB, 4 MB, … 128 KB, 64 KB, 16 KB
3. *Conflict*: Change from fully associative to n-way set associative while counting misses
   - Fully associative, 16-way, 8-way, 4-way, 2-way, 1-way

3Cs Revisted



- Three sources of misses (SPEC2000 integer and floating-point benchmarks)
  - Compulsory misses 0.006%; not visible
  - Capacity misses, function of cache size
  - Conflict portion depends on associativity and cache size

Improving Cache Performance: Summary

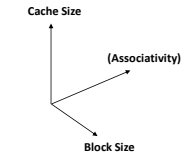
1. Reduce the time to hit in the cache
   - Smaller cache
   - 1 word blocks (no multiplexor/selector to pick word)
2. Reduce the miss rate
   - Bigger cache
   - Larger blocks (16 to 64 bytes typical)
   - (Later in semester: More flexible placement by increasing associativity)

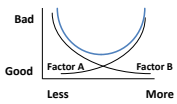
Improving Cache Performance: Summary

3. Reduce the miss penalty
   - Smaller blocks
   - Use multiple cache levels
     - L2 cache not tied to processor clock rate
   - Use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
   - Check write buffer on read miss – may get lucky
   - Faster backing store/improved memory bandwidth
     - (Later in lecture)

The Cache Design Space

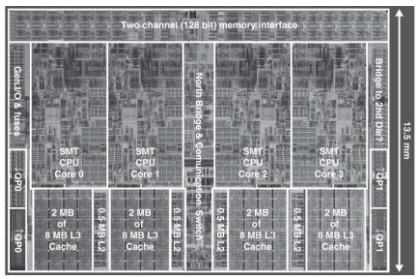
- Several interacting dimensions
  - Cache size
  - Block size
  - Write-through vs. write-back
  - Write allocation
  - Later Associativity
  - Replacement policy
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins





Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

Intel Nehalem Die Photo



- 4 cores, 32KB I$/32-KB D$, 512KB L2$
- Share one 8-MB L3$

Summary

- Multi-level caches - Reduce Cache Miss Penalty
  - Optimize first level to be fast!
  - Optimize 2nd and 3rd levels to minimize the memory access penalty
- Set-associativity - Reduce Cache Miss Rate
  - Memory block maps into more than 1 cache block
  - N-way: n possible places in cache to hold a memory block
- Lots and lots of cache parameters!
  - Write-back vs. write through, write allocation, block size, cache size, associativity, etc.

Bonus slides

- Note: These slides will be very useful for understanding lab 7 and especially project 2!
- The slides will appear in the order they would have in the normal presentation

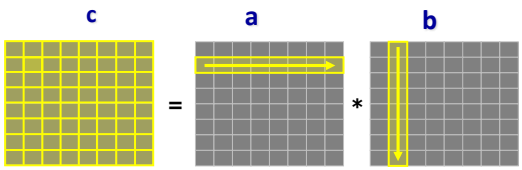
Performance Programming: Adjust software accesses to improve miss rate

- Now that understand how caches work, can revise program to improve cache utilization
  - Cache size
  - Block size
  - Multiple levels

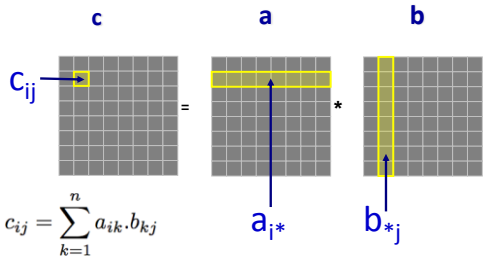
Performance of Loops and Arrays

- Array performance often limited by memory speed
- OK if access memory 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
- One approach called *Cache Blocking*: "shrink" problem by performing multiple iterations within smaller cache blocks
- Use Matrix Multiply as example: Next Lab and Project 3

Matrix Multiplication



Matrix Multiplication



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

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

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];
```

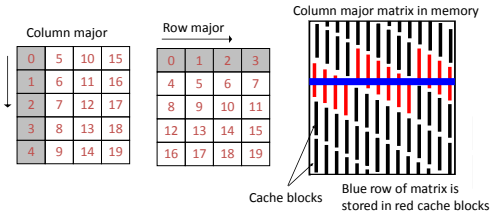
Advantage: code simplicity

Disadvantage: Marches through memory and caches

49

Note on Matrix in Memory

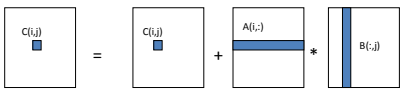
- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout
  - by column, or "column major" (Fortran default); A(i,j) at A+i+j*n
  - by row, or "row major" (C default) A(i,j) at A+i*n+j



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}
```



Linear Algebra to the Rescue!

- Instead of Multiplying two, say, 6 x 6 matrices

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}$$

where $A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$. From this "blocked" representation, we can now calculate the matrix product as such

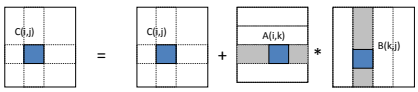
$$AB = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}) & (A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32}) & (A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33}) \\ (A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}) & (A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32}) & (A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33}) \\ (A_{31}B_{11} + A_{32}B_{21} + A_{33}B_{31}) & (A_{31}B_{12} + A_{32}B_{22} + A_{33}B_{32}) & (A_{31}B_{13} + A_{32}B_{23} + A_{33}B_{33}) \end{bmatrix}$$

- Thus, can get same result as multiplication of a set of submatricies

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

Another View of "Blocked" Matrix Multiplication

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

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_{144}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

N = 4 *r

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

- **Main Point:** each multiplication operates on small "block" matrices, whose size may be chosen so that they fit in the cache.

54

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

55

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 ≈ √M

Simple Matrix Multiply Whole Thing - www.youtube.com/watch?v=f3-z6t_xlyw

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; other things