Review: Cache Configurations

- Direct-mapped cache:
  - Index bits determined by the number of cache blocks

- Set-associative cache
  - Index bits determined by the number of sets

- Fully-associative cache
  - No index

http://csillustrated.berkeley.edu/illustrations.php

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
  - (Block number) modulo (number of sets)
- Direct mapped
  - Number of sets = 4

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

- 2-way set associative: (Block number) modulo (number of sets)
  - Number of sets = 4/2 = 2

Block address	Cache (set) index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

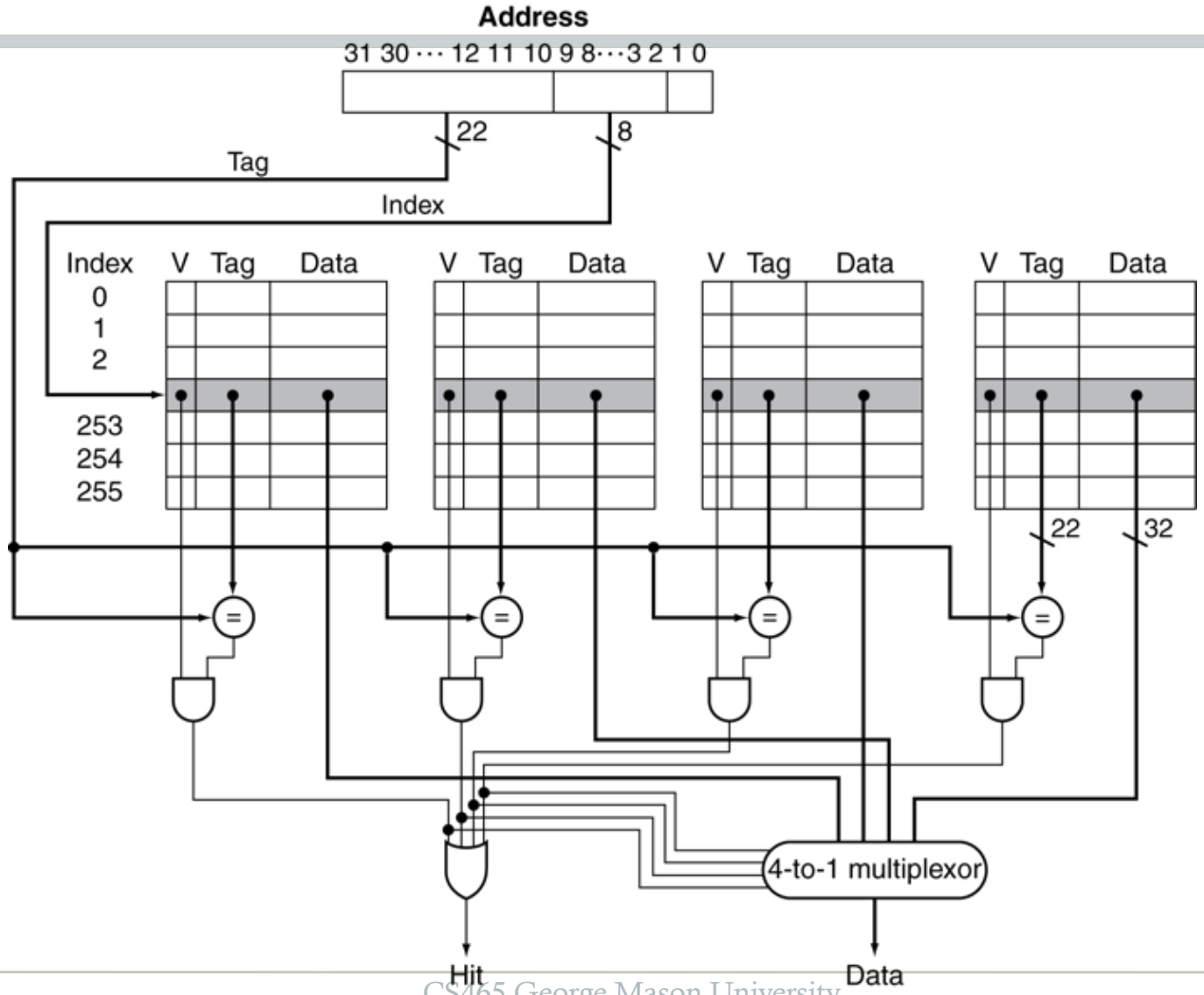
How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Set Associative Cache

- What's so great about this?
  - Even a 2-way set assoc cache avoids a lot of conflict misses
  - Hardware cost isn't that bad: only need N comparators (N usually a small integer)

Set Associative Cache Organization



Roadmap

- Cache basics
- Cache performance
  - Evaluation
  - Improvement
    - Set-associative caches
    - Block replacement policy ←
    - Multilevel caches
    - Program optimizations

Replacement Policy

- Placement policy: where to place an incoming block?
  - Direct mapped: mod determines only one choice
  - Set (fully) associative: anywhere within the set is allowed
    - Any location with valid bit off (empty) → use that entry
    - All possible locations already have a valid block → we must pick a block to replace
- Replacement policy
  - Rule by which we determine which block gets "cached out" on a miss

Replacement Policy

- Ideal/best (Belady's algorithm)
  - Keep the one with a closer future access
  - Need to know the future (impossible!)
- LRU (Least Recently Used)
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality ⇒ recent past use implies likely future use
  - Con: hardware and time cost to keep track of the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, hard beyond that

Replacement Example

- Fully associative cache with only 2 slots

Block address	LRU Replacement		
	Hit/Miss	Cache content after access	
5	Miss	Mem[5]	
1	Miss	Mem[5]	Mem[1]
2	Miss	**Mem[2]**	**Mem[1]**
1	Hit	Mem[2]	Mem[1]
2	Hit	Mem[2]	Mem[1]

Mem[1] is more recently accessed

Replacement Example

- Fully associative cache with only 2 slots

Block address	LRU			Belady		
	Hit/Miss	Cache content after access		Hit/Miss	Cache content after access	
0	Miss	Mem[0]		Miss	Mem[0]	
8	Miss	Mem[0]	Mem[8]	Miss	Mem[0]	Mem[8]
0	Hit	Mem[0]	Mem[8]	Hit	Mem[0]	Mem[8]
6	Miss	Mem[0]	**Mem[6]**	Miss	**Mem[6]**	Mem[8]
8	**Miss**	Mem[8]	Mem[6]	**Hit**	Mem[6]	Mem[8]

Replacement Policy

- Ideal/best (Belady's algorithm)
  - Need to know the future (impossible!)
- LRU (Least Recently Used)
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality ⇒ recent past use implies likely future use
  - Con: hardware and time cost to keep track of the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, hard beyond that
- Many other policies
  - FIFO, LFU, Random, …

Block Replacement Example

- We have a 2-way set associative cache with a four word *total* capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):

  0, 2, 0, 1, 4, 0, 2, 3, 5, 4

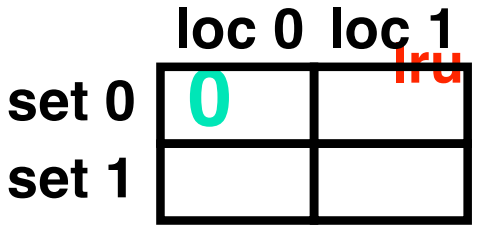
  - How many misses will there be for the LRU block replacement policy?

  - Note: for 2-way cache, we add 1 LRU bit to keep track of the one unused longer

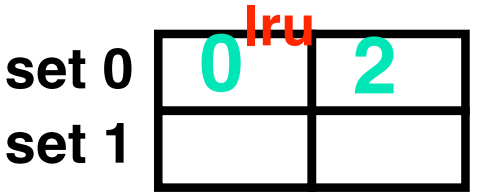
Block Replacement Example: LRU

- 2-way set associative cache, 4 blocks total
  - Number of sets: 4/2 = 2
- Adresses 0, 2, 0, 1, 4, 0, ...
  - Mod 2 to map to sets: 0, 0, 0, 1, 0, 0, …

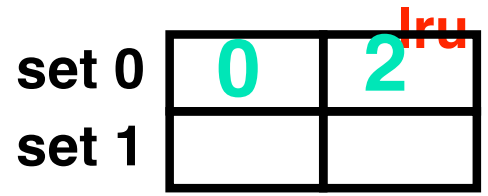
0: miss, bring into set 0 (loc 0)



2: miss, bring into set 0 (loc 1)



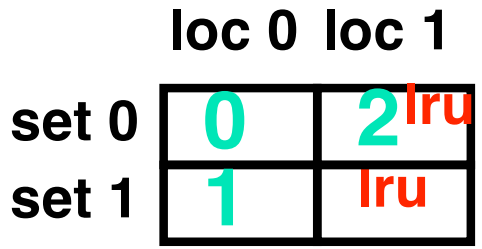
0: hit



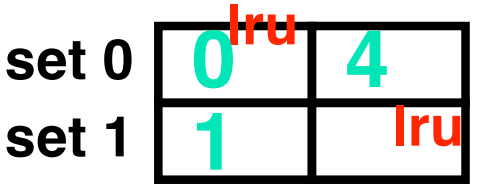
Block Replacement Example: LRU

- Adresses 0, 2, 0, 1, 4, 0, ...
  - Mod 2 to map to sets: 0, 0, 0, 1, 0, 0, …

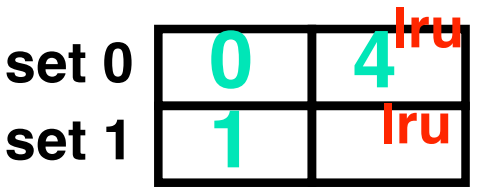
1: miss, bring into set 1 (loc 0)



4: miss, bring into set 0 (loc 1, replace 2)



0: hit



LRU Variations

- True LRU is costly for caches with a high associativity
- Variations based LRU
  - NRU (Not Recently Used)
    - Setting/clearing an access bit: similar to VM algorithm
    - Used in Intel Itanium, Sparc T2
  - Tree PLRU (Tree Pseudo-LRU)
    - Binary tree with an "LRU" flag
    - Used in Intel 486, Power PC
  - SLRU (Segmented/protected LRU)
    - Divide cache into two or more segments based on lifetime/locality features
    - Used in Facebook Content Distribution Network (FBCDN)

Roadmap

- Cache basics
- Cache performance
  - Evaluation
  - Improvement
    - Set associative cache Block
    - Replacement policy
    - Multilevel caches
    - Program optimizations
- Virtual memory
- Concluding remarks

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some systems include L-3 cache

Multi-level Cache Performance

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate * L1 Miss Penalty
L1 Miss Penalty =
L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate *
(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction (L1 cache) = 10%
  - Main memory access time = 25ns
- With just L-1 cache
  - Effective CPI = ?
  - Miss penalty = 25ns/0.25ns = 100 cycles
  - CPI_{actual} = CPI_{base} + memory stalls per instruction
    = 1 + 10% * 100 = 11

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - L2 miss rate = 2%
  - Effective CPI = ?
- L-1 miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- L-1 miss with L-2 miss
  - Extra penalty = 100 cycles (same as before)

Example (cont.)

- L-1 miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
  - L1 miss rate = 10%
- L-1 miss with L-2 miss
  - L2 miss rate = 2%, Extra penalty = 100
  - % insn to main memory = 10% * 2% = 0.2%
- CPI_{actual} = CPI_{base} + memory stalls per instruction
  = 1 + 10% * 20 + 0.2% * 100 = 3.2
- Compared with L1-cache only:
  - CPI_{actual} = CPI_{base} + memory stalls per instruction
    = 1 + 10% * 100 = 11

Multilevel Cache Considerations

- Primary/L-1 cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache in L1-L2 is usually smaller than a cache with L-1 only
  - L-1 block size smaller than L-2 block size

Outline

- Cache basics
- Cache performance
  - Evaluation
  - Improvement
    - Set-associative caches
    - Block replacement policy
    - Multilevel caches
    - Program optimizations
- Virtual memory
- Concluding remarks

Program Optimizations

- Many transformations targeting cache performance performed by compilers
- Many ways to rewrite loops
  - Loop fusion

```
for (i=0; i<n; i++)
    A[i] = C[i] + D[i]
for (i=0; i<n; i++)
    B[i] = C[i] - D[i]
```

```
for (i=0; i<n; i++)
    A[i] = C[i] + D[i]
    B[i] = C[i] - D[i]
```

  - Loop interchange
    - Assume row-major storage

```
for (j=1; j<n; j++)
    for (i=0; i<n; i++)
        X[i][j] = 2 * X[i][j-1]
```

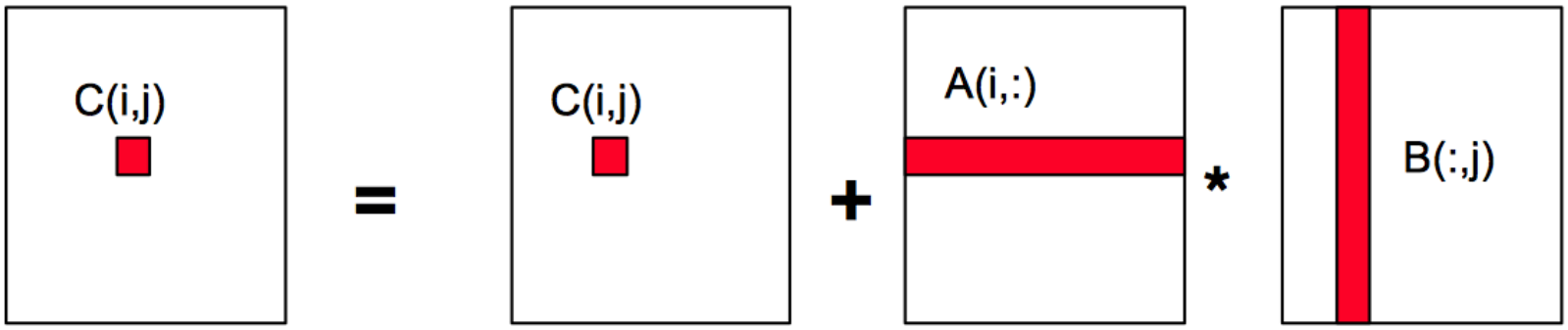
```
for (i=0; i<n; i++)
    for (j=1; j<n; j++)
        X[i][j] = 2 * X[i][j-1]
```

Blocking

- Matrix multiplication: C = C+A*B

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

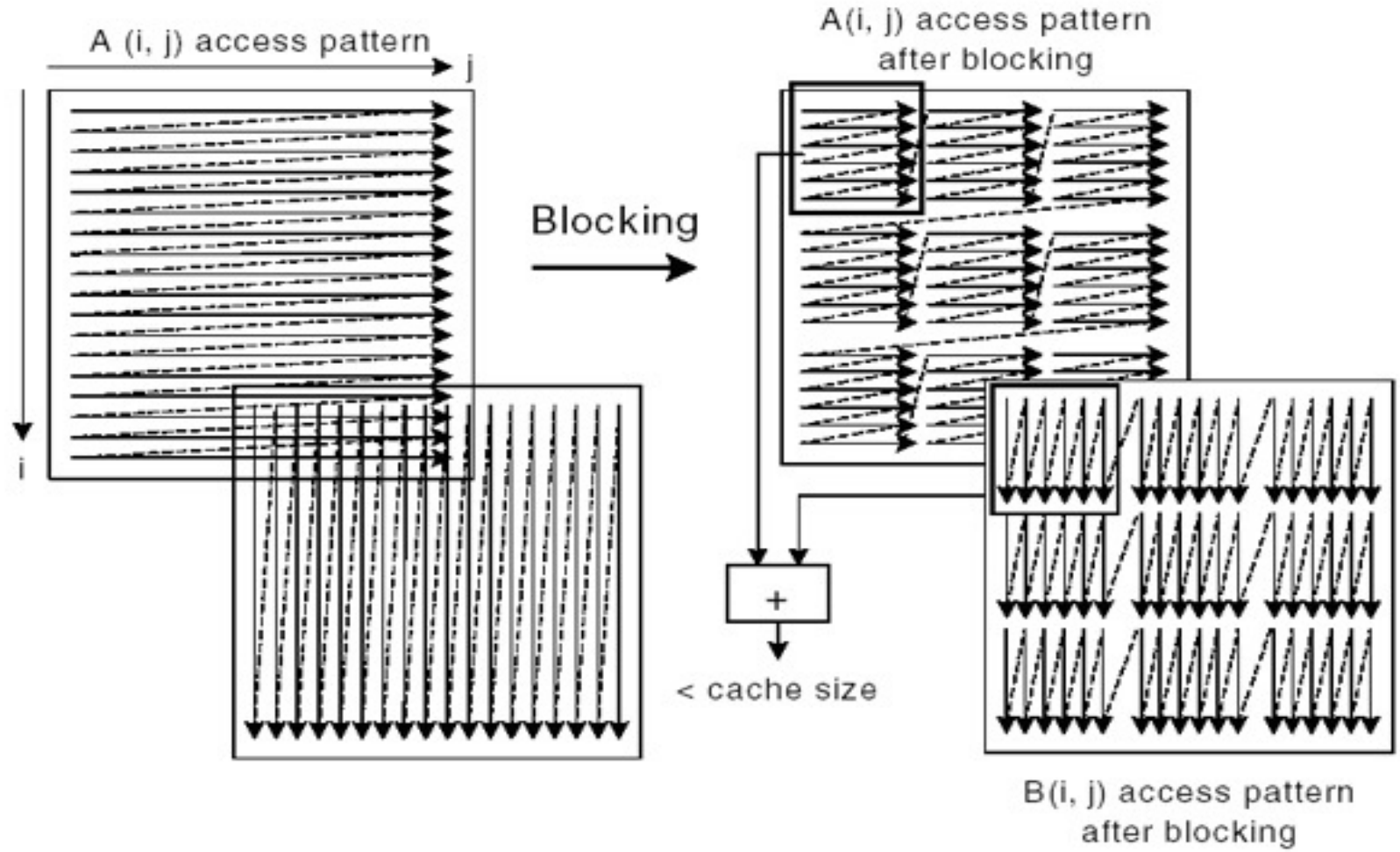
$$C_{ij} \mathrel{+}= \sum_k A_{ik} \times B_{kj}$$



Blocking

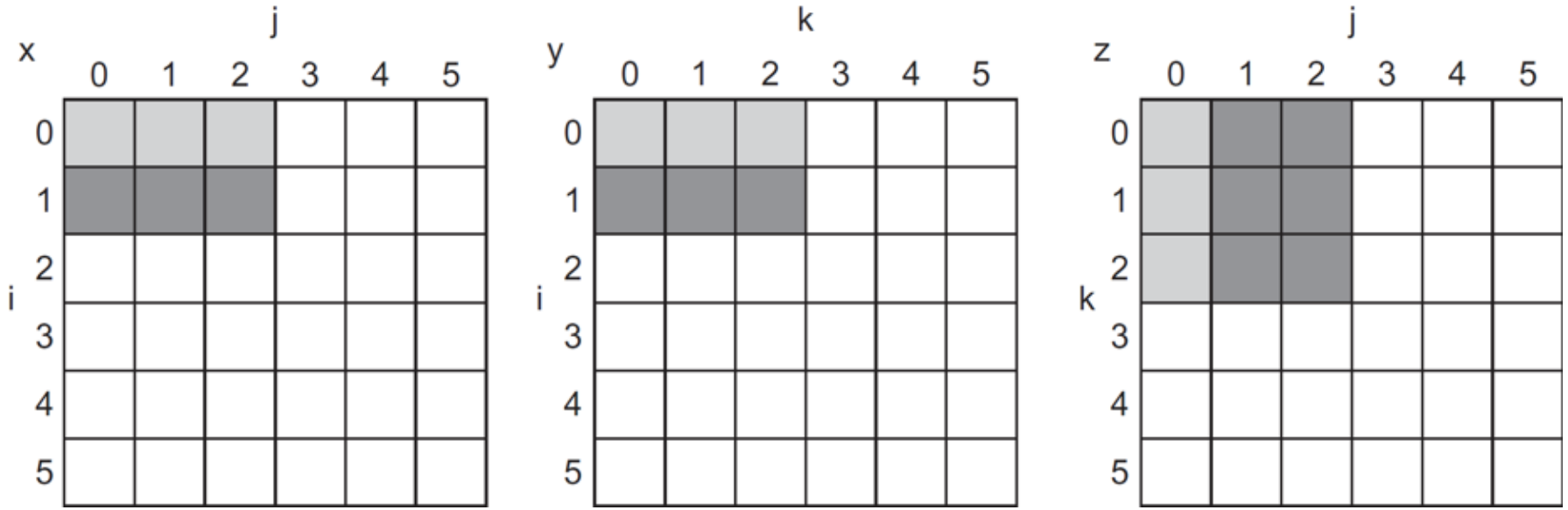
- Optimization for matrix calculations, e.g. matrix multiplication
- Idea:
  - Divide the original matrix into multiple sub-matrices
  - Work on one sub-matrix in each step: lower requirements on cache size
  - Maximize work performed on each sub-matrix
- Check Ch5.4 (pp.415) of textbook for details of transformation

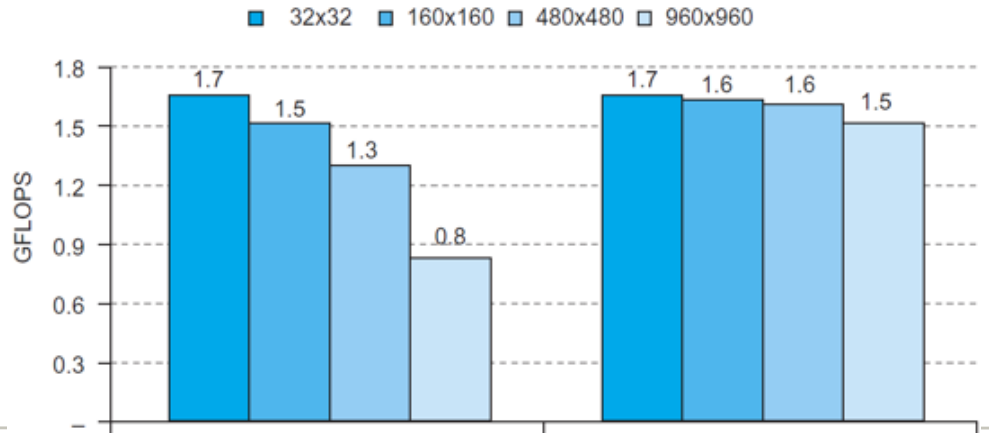
Updated Access Pattern



https://software.intel.com/en-us/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture

Blocked DGEMM Access Pattern





Outline

- Cache basics
- Cache performance
  - Configurations
  - Evaluation
  - Improvement
- Virtual memory←
  - Four memory hierarchy questions
  - Page table and TLB
- Concluding remarks