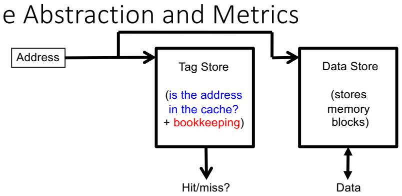
Cache Performance

Samira Khan
March 28, 2017

Agenda

- Review from last lecture
  - Cache access
  - Associativity
- Replacement
- Cache Performance

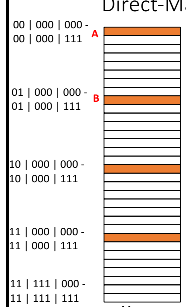
Cache Abstraction and Metrics



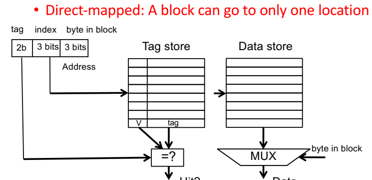
- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)

3

Direct-Mapped Cache: Placement and Access

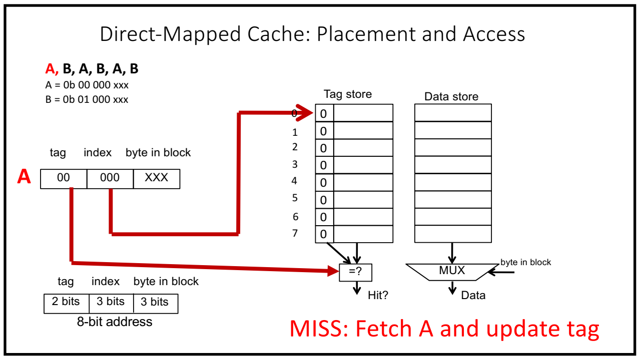


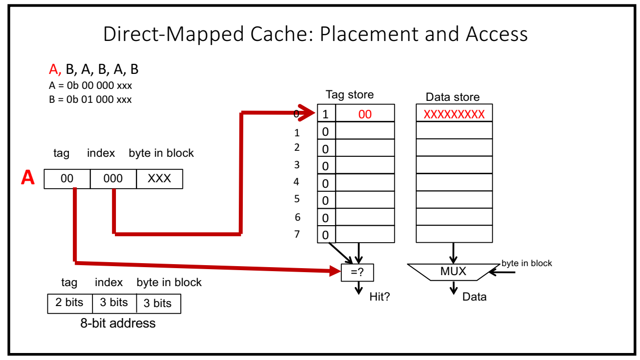
- Assume byte-addressable memory: 256 bytes, 8-byte blocks
  → 32 blocks
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location

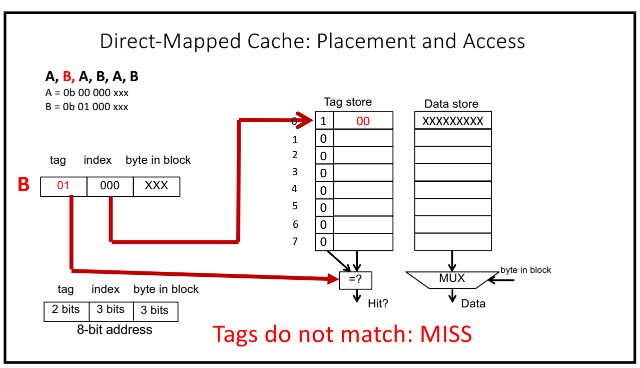


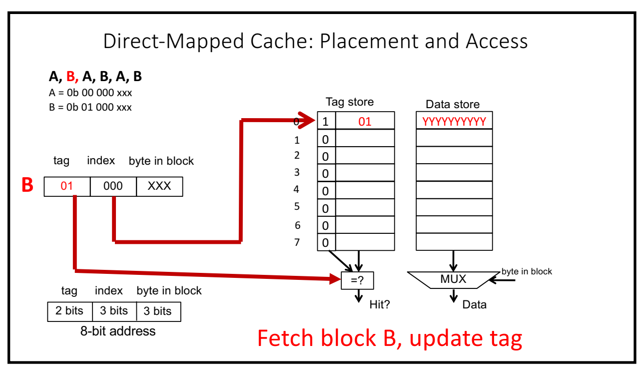
- Addresses with same index contend for the same location
  - Cause conflict misses

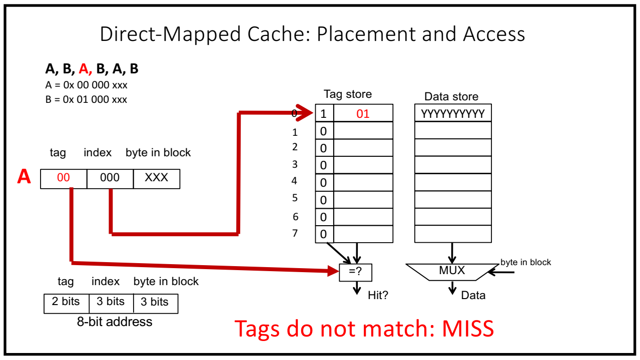
4

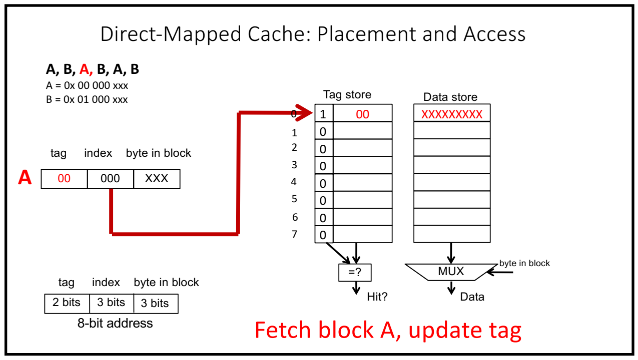

Direct-Mapped Cache: Placement and Access
A, B, A, B, A, B
A = 0b 00 000 xxx
B = 0b 01 000 xxx
Tag store
Data store
tag index byte in block
A
00 000 XXX
=?
MUX
byte in block
Hit?
Data
tag index byte in block
2 bits 3 bits 3 bits
8-bit address
MISS: Fetch A and update tag

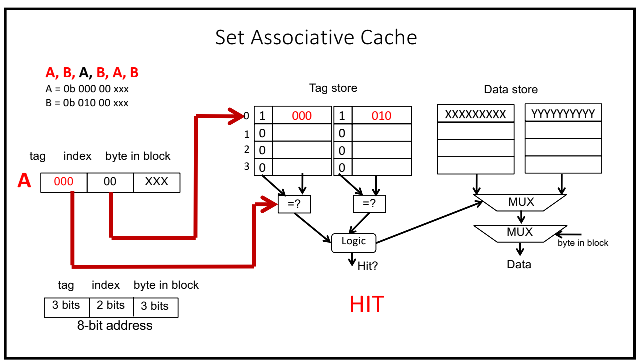

Direct-Mapped Cache: Placement and Access
A, B, A, B, A, B
A = 0b 00 000 xxx
B = 0b 01 000 xxx
Tag store
Data store
XXXXXXXXX
tag index byte in block
A
00 000 XXX
=?
MUX
byte in block
Hit?
Data
tag index byte in block
2 bits 3 bits 3 bits
8-bit address


Direct-Mapped Cache: Placement and Access
A, B, A, B, A, B
A = 0b 00 000 xxx
B = 0b 01 000 xxx
Tag store
Data store
XXXXXXXXX
tag index byte in block
B
01 000 XXX
=?
MUX
byte in block
Hit?
Data
tag index byte in block
2 bits 3 bits 3 bits
8-bit address
Tags do not match: MISS


Direct-Mapped Cache: Placement and Access
A, B, A, B, A, B
A = 0b 00 000 xxx
B = 0b 01 000 xxx
Tag store
Data store
YYYYYYYYYY
tag index byte in block
B
01 000 XXX
=?
MUX
byte in block
Hit?
Data
tag index byte in block
2 bits 3 bits 3 bits
8-bit address
Fetch block B, update tag

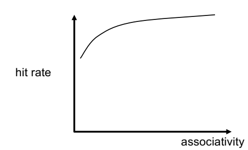






Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can map to the same index (or set)?
- Higher associativity
  - ++ Higher hit rate
  - -- Slower cache access time (hit latency and data access latency)
  - -- More expensive hardware (more comparators)
- Diminishing returns from higher associativity



12

Issues in Set-Associative Caches

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)
- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

13

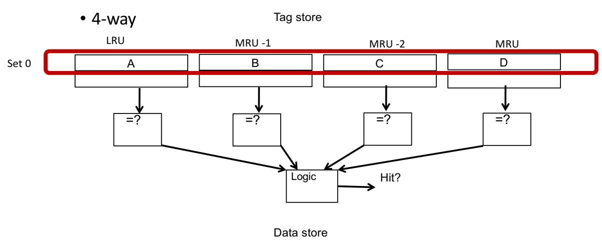
Eviction/Replacement Policy

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used
    - Hybrid replacement policies

14

Least Recently Used Replacement Policy

- 4-way

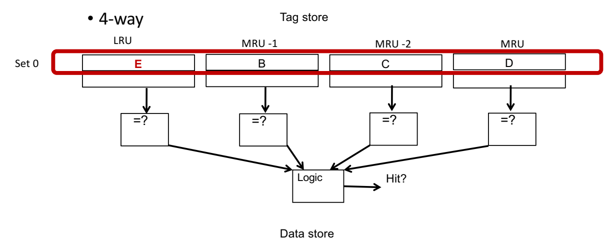


ACCESS PATTERN: ACBD

15

Least Recently Used Replacement Policy

- 4-way

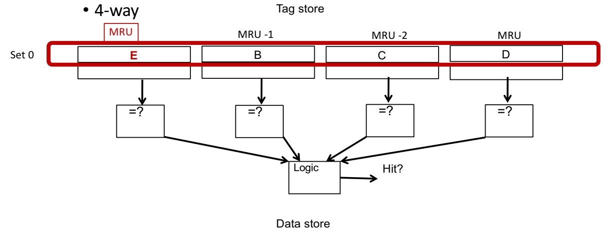


ACCESS PATTERN: ACBDE

16

Least Recently Used Replacement Policy

- 4-way

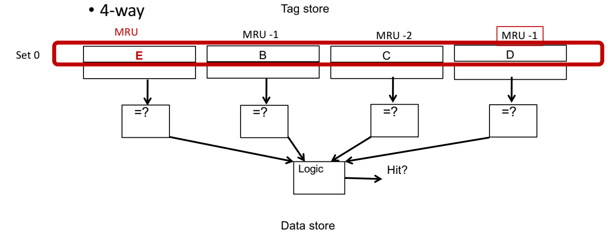


ACCESS PATTERN: ACBDE

17

Least Recently Used Replacement Policy

- 4-way

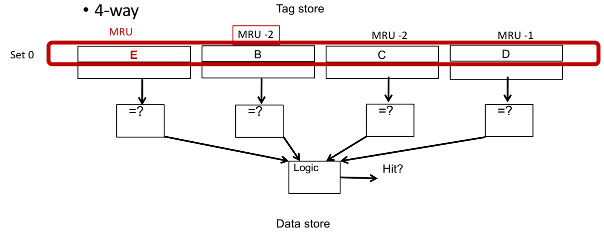


ACCESS PATTERN: ACBDE

18

Least Recently Used Replacement Policy

- 4-way

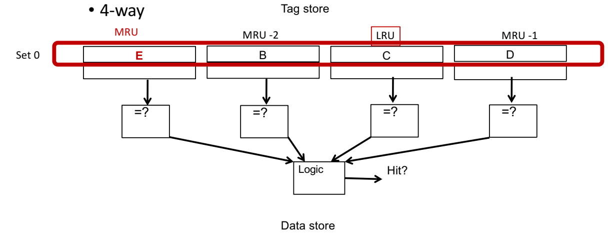


ACCESS PATTERN: ACBDE

19

Least Recently Used Replacement Policy

- 4-way

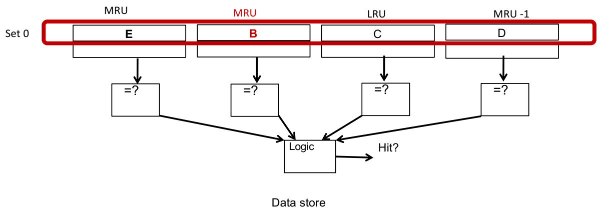


ACCESS PATTERN: ACBDE

20

Least Recently Used Replacement Policy

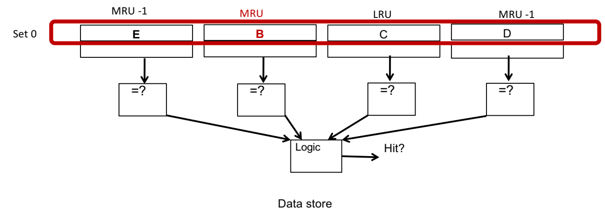
- 4-way



ACCESS PATTERN: ACBDEB

Least Recently Used Replacement Policy

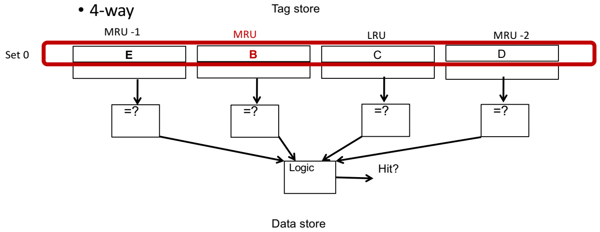
- 4-way



ACCESS PATTERN: ACBDEB

Least Recently Used Replacement Policy

- 4-way



ACCESS PATTERN: ACBDEB

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 16-way set associative cache:
  - What do you need to implement LRU perfectly?
  - What is the logic needed to determine the LRU victim?

Approximations of LRU

- Most modern processors do not implement "true LRU" (also called "perfect LRU") in highly-associative caches
- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)
- Examples:
  - Not MRU (not most recently used)

25

Cache Replacement Policy: LRU or Random

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar
- Best of both Worlds: Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

26

What's In A Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits
- Dirty bit?
  - Write back vs. write through caches

27

Handling Writes (I)

- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the block is evicted
- Write-back
  + Can consolidate multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy

  -- Need a bit in the tag store indicating the block is "dirty/modified"
- Write-through
  + Simpler
  + All levels are up to date. Consistent
  -- More bandwidth intensive; no coalescing of writes

28

Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss
  - No-allocate on write miss

- Allocate on write miss
  + Can consolidate writes instead of writing each of them individually to next level
  + Simpler because write misses can be treated the same way as read misses
  -- Requires (?) transfer of the whole cache block

- No-allocate
  + Conserves cache space if locality of writes is low (potentially better cache hit rate)

29

Instruction vs. Data Caches

- Separate or Unified?

- Unified:
  + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split I and D caches)
  -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- First level caches are almost always split
  - Mainly for the last reason above
- Second and higher levels are almost always unified

30

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
  - Tag store and data store accessed in parallel
- Second-level, third-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency less critical
  - Tag store and data store accessed serially

- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

31

Cache Performance

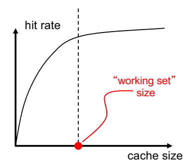
Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
  - Insertion/Placement policy

33

Cache Size

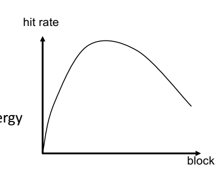
- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- Working set: the whole set of data the executing application references
  - Within a time interval



34

Block Size

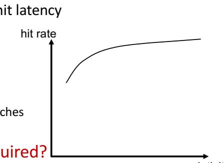
- Block size is the data that is associated with an address tag
- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead
- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high
  - Will see more examples later



35

Associativity

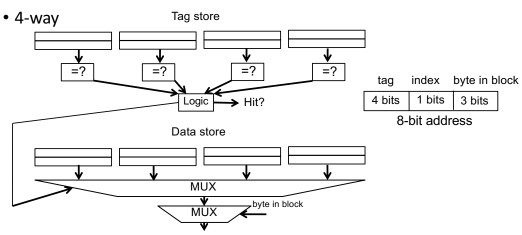
- How many blocks can map to the same index (or set)?
- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns, higher hit latency
- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches
- Power of 2 associativity required?



36

Higher Associativity

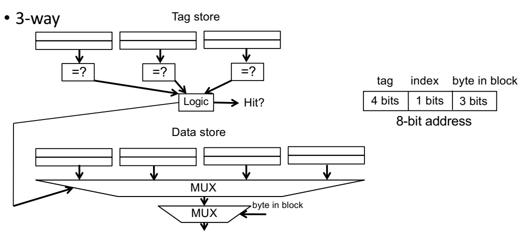
- 4-way



37

Higher Associativity

- 3-way



38

Classification of Cache Misses

- Compulsory miss
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below
- Capacity miss
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
- Conflict miss
  - defined as any miss that is neither a compulsory nor a capacity miss

39

How to Reduce Each Miss Type

- Compulsory
  - Caching cannot help
  - Prefetching
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Hashing
    - Software hints?
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set such that each "phase" fits in cache

40

Cache Performance with Code Examples

Matrix Sum

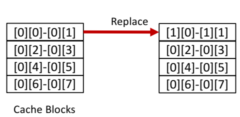
```
int sum1(int matrix[4][8]) {
    int sum = 0;
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 8; ++j) {
            sum += matrix[i][j];
        }
    }
}
```

access pattern:
matrix[0][0], [0][1], [0][2], …, [1][0] …

Exploiting Spatial Locality

8B cache block, 4 blocks, LRU, 4B integer
Access pattern matrix[0][0], [0][1], [0][2], …, [1][0] …

[0][0] → **miss**
[0][1] → hit
[0][2] → **miss**
[0][3] → hit
[0][4] → **miss**
[0][5] → hit
[0][6] → **miss**
[0][7] → hit
[1][0] → **miss**
[1][1] → hit



Exploiting Spatial Locality

- block size and spatial locality
- larger blocks — exploit spatial locality
- … but larger blocks means fewer blocks for same size
- less good at exploiting temporal locality

Alternate Matrix Sum

```
int sum2(int matrix[4][8]) {
        int sum = 0;
        // swapped loop order
        for (int j = 0; j < 8; ++j) {
                for (int i = 0; i < 4; ++i) {
                        sum += matrix[i][j];
                }
        }
}
```

access pattern:

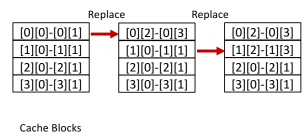
- matrix[0][0], [1][0], [2][0], [3][0], [0][1], [1][1], [2][1], [3][1],…, …

Bad at Exploiting Spatial Locality

8B cache block, 4B integer

Access pattern matrix[0][0], [1][0], [2][0], [3][0], [0][1], [1][1], [2][1], [3][1],…, …

[0][0] → **miss**
[1][0] → **miss**
[2][0] → **miss**
[3][0] → **miss**
[0][1] → hit
[1][1] → hit
[2][1] → hit
[3][1] → hit
[0][2] → **miss**
[1][2] → **miss**



A note on matrix storage

- A —> N X N matrix: represented as an 2D array
- makes dynamic sizes easier:
- float A_2d_array[N][N];
- float *A_flat = malloc(N * N);
- A_flat[i * N + j] === A_2d_array[i][j]

Matrix Squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} * A_{kj}$$

```
/* version 1: inner loop is k, middle is j */
for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
                for (int k = 0; k < N; ++k)
                        B[i*N+j] += A[i * N + k] * A[k * N + j];
```

Matrix Squaring

i →

j ↓

$\mathbf{B_{00}}$ B_{01} B_{02} B_{03}
B_{10} B_{11} B_{12} B_{13}
B_{20} B_{21} B_{22} B_{23}
B_{30} B_{31} B_{32} B_{33}

A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}

$$B_{00} = \sum_{k=0}^{n} A_{0k} * A_{k0}$$

$$B_{00} = (A_{00} * A_{00}) + (A_{01} * A_{10}) + (A_{02} * A_{20}) + (A_{03} * A_{30})$$

Matrix Squaring

i →

j ↓

$\mathbf{B_{00}}$ B_{01} B_{02} B_{03}
B_{10} B_{11} B_{12} B_{13}
B_{20} B_{21} B_{22} B_{23}
B_{30} B_{31} B_{32} B_{33}

$\mathbf{A_{00}}$ A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}

$$B_{00} = \sum_{k=0}^{n} A_{0k} * A_{k0}$$

$$B_{00} = (\mathbf{A_{00}} * \mathbf{A_{00}}) + (A_{01} * A_{10}) + (A_{02} * A_{20}) + (A_{03} * A_{30})$$

Matrix Squaring

i →

j ↓

$\mathbf{B_{00}}$ B_{01} B_{02} B_{03}
B_{10} B_{11} B_{12} B_{13}
B_{20} B_{21} B_{22} B_{23}
B_{30} B_{31} B_{32} B_{33}

$\mathbf{A_{00}}$ $\mathbf{A_{01}}$ A_{02} A_{03}
$\mathbf{A_{10}}$ A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}

$$B_{00} = \sum_{k=0}^{n} A_{0k} * A_{k0}$$

$$B_{00} = (A_{00} * A_{00}) + (\mathbf{A_{01}} * \mathbf{A_{10}}) + (A_{02} * A_{20}) + (A_{03} * A_{30})$$

Matrix Squaring

i →

j ↓

$\mathbf{B_{00}}$ B_{01} B_{02} B_{03}
B_{10} B_{11} B_{12} B_{13}
B_{20} B_{21} B_{22} B_{23}
B_{30} B_{31} B_{32} B_{33}

$\mathbf{A_{00}}$ $\mathbf{A_{01}}$ $\mathbf{A_{02}}$ A_{03}
$\mathbf{A_{10}}$ A_{11} A_{12} A_{13}
$\mathbf{A_{20}}$ A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}

$$B_{00} = \sum_{k=0}^{n} A_{0k} * A_{k0}$$

$$B_{00} = (A_{00} * A_{00}) + (A_{01} * A_{10}) + (\mathbf{A_{02}} * \mathbf{A_{20}}) + (A_{03} * A_{30})$$

Matrix Squaring

i →

j ↓

B_{00} B_{01} B_{02} B_{03}
B_{10} B_{11} B_{12} B_{13}
B_{20} B_{21} B_{22} B_{23}
B_{30} B_{31} B_{32} B_{33}

A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}

A_{ik} has spatial locality

$$B_{00} = \sum_{k=0}^{n} A_{0k} * A_{k0}$$

$$B_{00} = (A_{00} * A_{00}) + (A_{01} * A_{10}) + (A_{02} * A_{20}) + (A_{03} * A_{30})$$

Matrix Squaring

i →

j ↓

B_{00} B_{01} B_{02} B_{03}
B_{10} B_{11} B_{12} B_{13}
B_{20} B_{21} B_{22} B_{23}
B_{30} B_{31} B_{32} B_{33}

A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}

A_{ik} has spatial locality

$$B_{01} = \sum_{k=0}^{n} A_{0k} * A_{k1}$$

$$B_{01} = (A_{00} * A_{01}) + (A_{01} * A_{11}) + (A_{02} * A_{21}) + (A_{03} * A_{31})$$

Matrix Squaring

i →

j ↓

B_{00} B_{01} B_{02} B_{03}
B_{10} B_{11} B_{12} B_{13}
B_{20} B_{21} B_{22} B_{23}
B_{30} B_{31} B_{32} B_{33}

A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}

A_{ik} has spatial locality

$$B_{02} = \sum_{k=0}^{n} A_{0k} * A_{k2}$$

$$B_{02} = (A_{00} * A_{02}) + (A_{01} * A_{12}) + (A_{02} * A_{22}) + (A_{03} * A_{32})$$

Conclusion

- A_{ik} has spatial locality
- B_{ij} has temporal locality

Matrix Squaring

$$B_{ij} = \sum_{k=1}^{n} A_{ik} * A_{kj}$$

```
/* version 2: outer loop is k, middle is j */
for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
                for (int j = 0; j < N; ++j)
                        B[i*N+j] += A[i * N + k] * A[k * N + j];
```

Access pattern k = 0, i = 0

```
B[0][0]  = A[0][0] * A[0][0]
B[0][1] = A[0][0] * A[0][1]
B[0][2] = A[0][0] * A[0][2]
B[0][3] = A[0][0] * A[0][3]
```

Access pattern k = 0, i = 1

```
B[1][0]  = A[1][0] * A[0][0]
B[1][1] = A[1][0] * A[0][1]
B[1][2] = A[1][0] * A[0][2]
B[1][3] = A[1][0] * A[0][3]
```

Matrix Squaring: kij order

i →

j ↓

$$\begin{matrix} \mathbf{B_{00}} & \mathbf{B_{01}} & \mathbf{B_{02}} & \mathbf{B_{03}} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{matrix} \qquad \begin{matrix} \mathbf{A_{00}} & \mathbf{A_{01}} & \mathbf{A_{02}} & \mathbf{A_{03}} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{matrix}$$

$$\mathbf{B_{00}} = (\mathbf{A_{00}} * \mathbf{A_{00}}) + (A_{01} * A_{10}) + (A_{02} * A_{20}) + (A_{03} * A_{30})$$
$$\mathbf{B_{01}} = (\mathbf{A_{00}} * \mathbf{A_{01}}) + (A_{01} * A_{11}) + (A_{02} * A_{21}) + (A_{03} * A_{31})$$
$$\mathbf{B_{02}} = (\mathbf{A_{00}} * \mathbf{A_{02}}) + (A_{01} * A_{12}) + (A_{02} * A_{22}) + (A_{03} * A_{32})$$
$$\mathbf{B_{03}} = (\mathbf{A_{00}} * \mathbf{A_{03}}) + (A_{01} * A_{13}) + (A_{02} * A_{23}) + (A_{03} * A_{33})$$

Matrix Squaring: kij order

B_{ij} , A_{kj} have spatial locality
A_{ik} has temporal locality

i →

j ↓

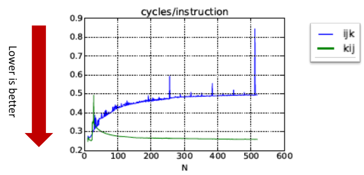
$$\begin{matrix} B_{00} & B_{01} & B_{02} & B_{03} \\ \mathbf{B_{10}} & \mathbf{B_{11}} & \mathbf{B_{12}} & \mathbf{B_{13}} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{matrix} \qquad \begin{matrix} \mathbf{A_{00}} & \mathbf{A_{01}} & \mathbf{A_{02}} & \mathbf{A_{03}} \\ \mathbf{A_{10}} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{matrix}$$

$$\mathbf{B_{10}} = (\mathbf{A_{10}} * \mathbf{A_{00}}) + (A_{11} * A_{10}) + (A_{12} * A_{20}) + (A_{13} * A_{30})$$
$$\mathbf{B_{11}} = (\mathbf{A_{10}} * \mathbf{A_{01}}) + (A_{11} * A_{11}) + (A_{12} * A_{21}) + (A_{13} * A_{31})$$
$$\mathbf{B_{12}} = (\mathbf{A_{10}} * \mathbf{A_{02}}) + (A_{11} * A_{12}) + (A_{12} * A_{22}) + (A_{13} * A_{32})$$
$$\mathbf{B_{13}} = (\mathbf{A_{10}} * \mathbf{A_{03}}) + (A_{11} * A_{13}) + (A_{12} * A_{23}) + (A_{13} * A_{33})$$

Matrix Squaring

- kij order
- B_{ij} , A_{kj} have spatial locality
- A_{ik} has temporal locality
- ijk order
- A_{ik} has spatial locality
- B_{ij} has temporal locality

Which order is better?



Order kij performs much better