CMPE 200
Computer Architecture & Design

Lecture 4. Memory Hierarchy (4)

Haonan Wang

SJSU SAN JOSÉ STATE UNIVERSITY

Cache Policies: Cases

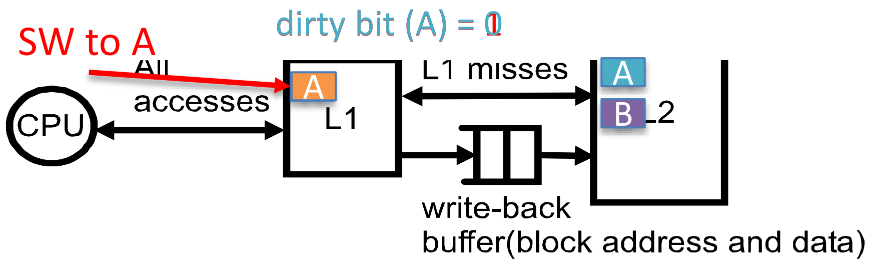
- **Allocation policy: do we allocate a block in cache for the missed data?**

- **Read policies:**
  - Read Hit: this is what we want. Only one data read from the cache.
  - Read Miss: needs to fetch from lower level, but just write to the register once after that
    - read-allocate (with replacement policy) vs. no-read-allocate (i.e., cache bypassing)

- **Write policies (only for the data cache): consistency & performance tradeoffs**
  - Write Hit: <u>behavior and number of writes depends on write policy</u>
    - Write-through vs. write-back vs. write-evict
  - Write Miss: <u>needs to first read from lower level, then apply write policies</u>
    - Write-allocate (with replacement policy): Write-through vs. Write-back
    - No-write-allocate (bypassing): Write-evict

Write Policy: Write-back

- **Write-back: inconsistent with lower level**
  - The value is written only to the cache line.
  - The modified (dirty) cache line is written to the lower level only when it is evicted.
    - 1 dirty bit is needed for each cache line.
  - A write-back buffer to update lower level with evicted dirty blocks
    - Must write a full block at this point since we do not know which word is modified
  - Example: assume cache block containing word address A is initially in the cache

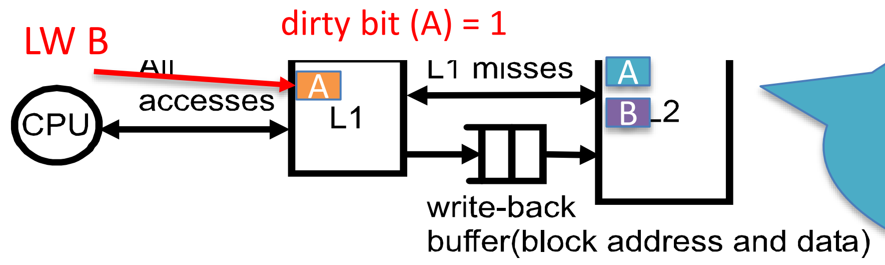


SJSU SAN JOSÉ STATE UNIVERSITY

Write Policy: Write-back

- **Write-back: inconsistent with lower level**
  - The value is written only to the cache line.
  - The modified (dirty) cache line is written to the lower level <u>only when it is evicted</u>.
    - 1 dirty bit is needed for each cache line.
  - A write-back buffer to update lower level with evicted dirty blocks
    - Must write <u>a full block</u> at this point since we do not know which word is modified
  - Example: assume cache block containing word address A is initially in the cache

A few cycles later...

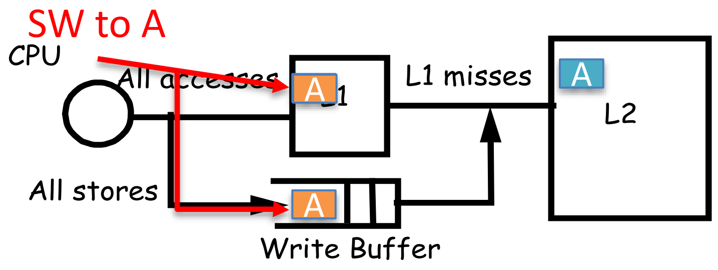


L1 is full and A is selected to be replaced by B

If A was not modified (dirty bit = 0) when replaced, A is not updated to lower mem.

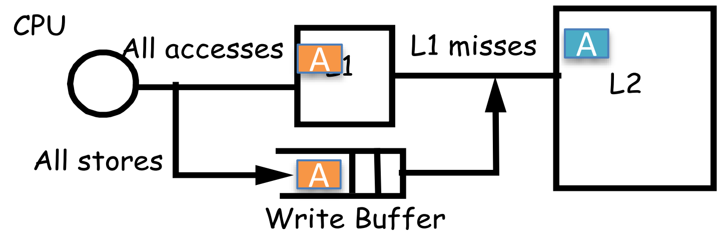
Write Policy: Write-through

- **Write-through: consistent with lower level**
  - The value is written to both the cache line and the lower-level memory.
    - No need to write a full block to the lower level (e.g., only write 4B for a SW)
  - Write buffer can be used so cache does not need to wait for the write to lower level
    - stall only if the write buffer is full
  - Example: assume cache block containing word address A is initially in the cache



Write Policy: Write-through

- **Write-through: consistent with lower level**
  - The value is written to both the cache line and the lower-level memory.
    - No need to write a full block to the lower level (e.g., only write 4B for a SW)
  - Write buffer can be used so cache does not need to wait for the write to lower level
    - stall only if the write buffer is full
  - Example: assume cache block containing word address A is initially in the cache



SJSU SAN JOSÉ STATE UNIVERSITY

Write Policy: Write-evict

- **Write-evict (no-write): no consistency issue**
  - The value is written only to the lower level (i.e., bypassing).
  - No need to fetch the data first if it is missing in the cache
  - If the block containing the write address exists in the cache, evict it.
  - The write only needs to send the data specified by the store instruction (e.g., 4 Bytes).
  - Write buffer can be used so cache does not need to wait for the write to lower level
  - More expensive when there is read after write

Cache Policies: Cases - Revisit

- **Allocation policy: do we allocate a block in cache for the missed data?**

What else must be considered?

- **Read policies:**
  - Read Hit: this is what we want. Only one data read from the cache.
  - Read Miss: needs to fetch from lower level, but just write to the register once after that
    - read-allocate (with replacement policy) vs. no-read-allocate (i.e., cache bypassing)

      + write policy of evicted data
- **Write policies (only for the data cache): consistency & performance tradeoffs**
  - Write Hit: behavior and number of writes depends on write policy
    - Write-through vs. write-back vs. write-evict
  - Write Miss: needs to first read from lower level, then apply write policies
    - Write-allocate (with replacement policy): Write-through vs. Write-back

      + write policy of evicted data
    - No-write-allocate (bypassing): Write-evict

Cache Miss Behavior Analysis

- **Read miss:**
  - **+Write-through**: evict victim block + fetch block from lower level
  - **+Write-back**: evict victim block + write back evicted block if dirty + fetch block from lower level
  - **+No-write**: find victim block + fetch block from lower level
- **Write miss:**
  - Write-allocate:
    - **+Write-through**: evict victim block + fetch block from lower level + store word to block + <u>store word to lower level</u>
    - **+Write-back**: evict victim block + <u>write back evicted block if dirty</u> + fetch block from lower level + store word to block
  - No-write-allocate: **+Write-evict**: store word to lower level directly

Other Cache Policy details

- **Some caches support multiple policies**
  - Can achieve different delay/bandwidth tradeoffs or even dynamically change
- **Write hit and write miss can use different policies**
  - E.g., write-hit write-evict + write-miss write-allocate-write-back
- **Different cache levels can also use different policies**
  - E.g., L1 use write-evict + L2 use write-back
- **Cache can also be completely bypassed (not the same as write-evict)**
  - E.g., bypassing L1

Reduce Miss Rate (1): Code Optimization

- **Misses occur if sequentially accessed array elements come from different cache blocks**

- **Code optimizations → No hardware change**
  - Rely on programmers or compilers

- **Examples:**
  - Loop interchange: In nested loops, outer loop becomes inner loop and vice versa
  - Loop blocking: partition large array into smaller blocks, thus fitting the accessed array elements into cache size

Loop Interchange

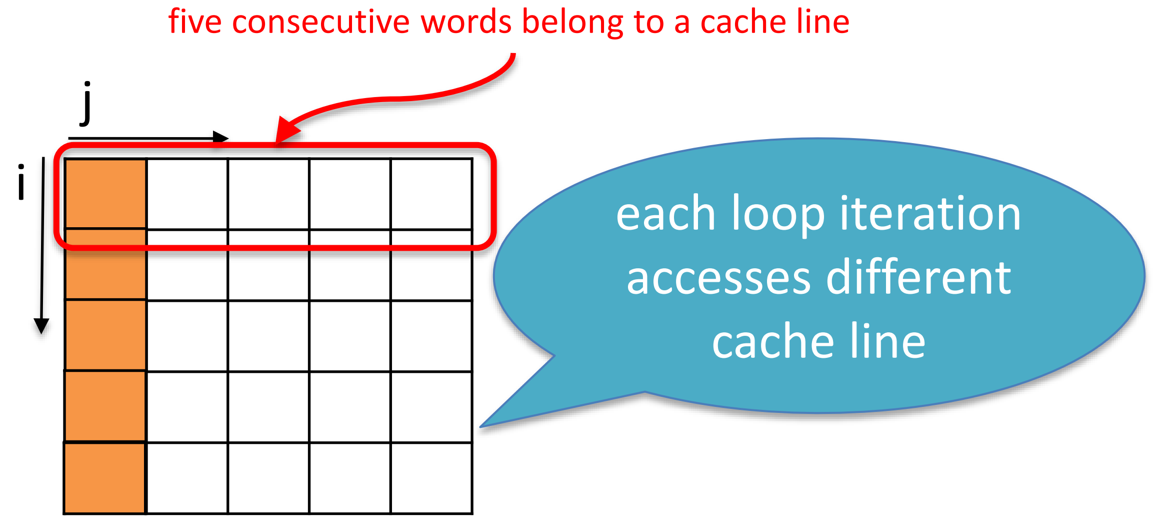
Assume: cache line size = 5 words
What addresses are accessed in each iteration of inner loop?

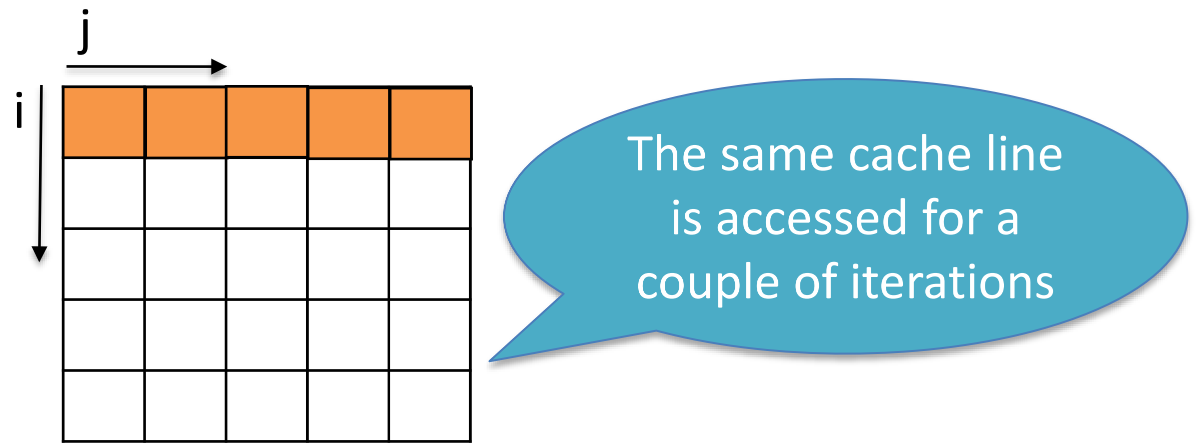
```
/* Before */
for (j=0; j<5; j++)
 for (i=0; i<5; i++)
   x[i][j] = 2*x[i][j]
```

Column-major ordering

Row-major ordering

```
/* After */
for (i=0; i<5; i++)
 for (j=0; j<5; j++)
   x[i][j] = 2*x[i][j]
```


five consecutive words belong to a cache line
j
i
each loop iteration accesses different cache line

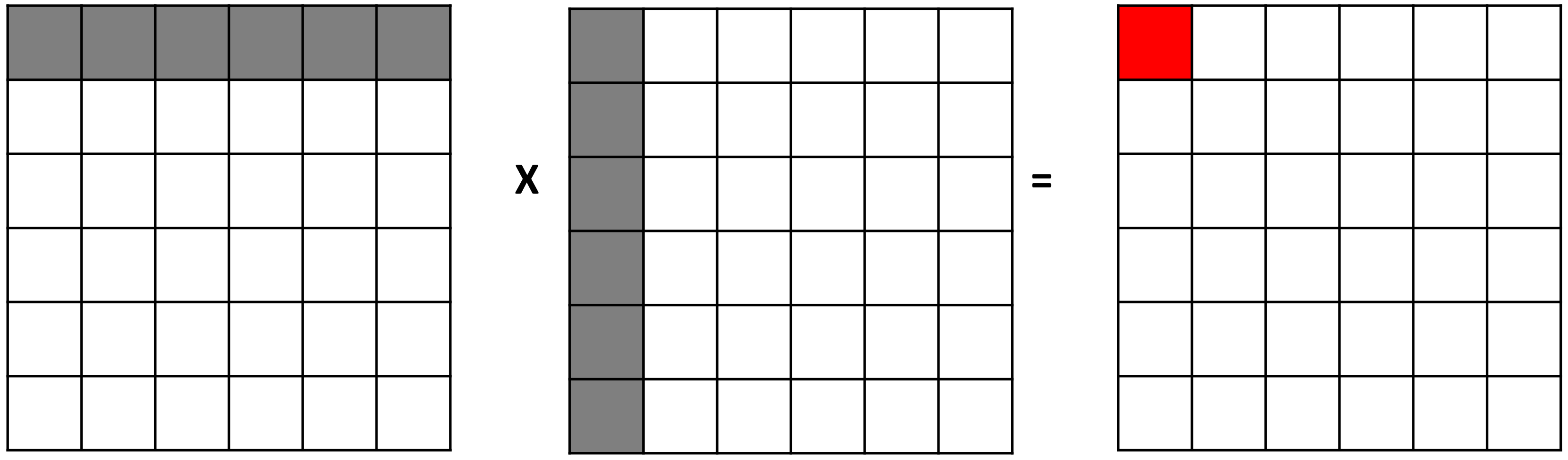


SJSU SAN JOSÉ STATE UNIVERSITY

Loop Blocking

In the traditional matrix multiplication:

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

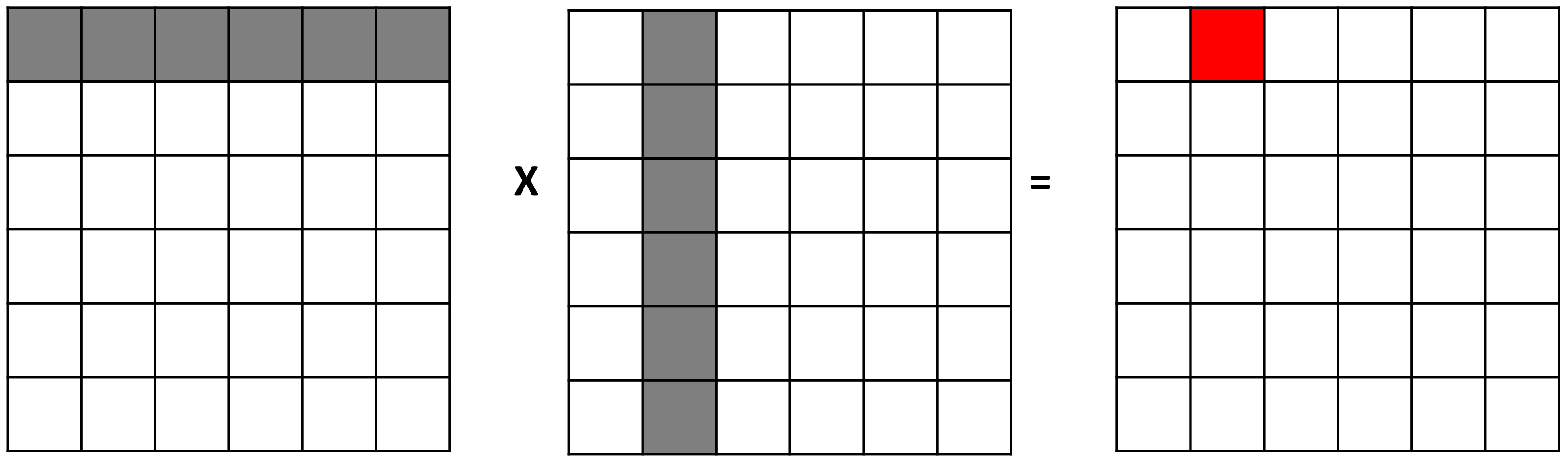


To generate one output element (red color), the 6 elements of each input matrix (gray color) will be accessed.

Loop Blocking

In the traditional matrix multiplication:

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

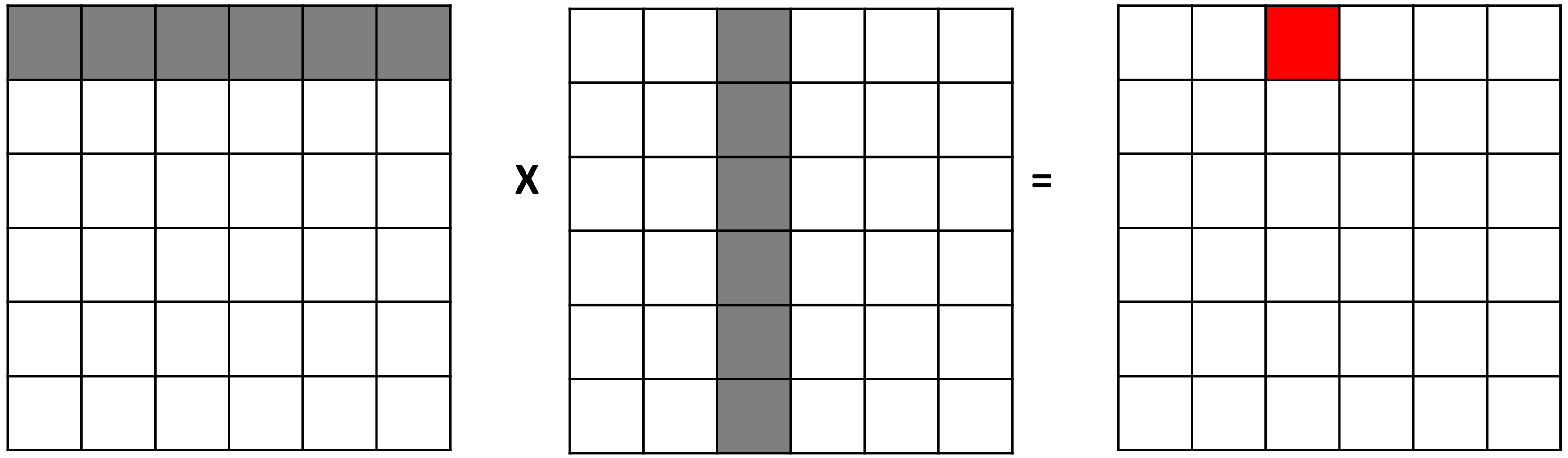


To generate one output element (red color), the 6 elements of each input matrix (gray color) will be accessed.

Loop Blocking

In the traditional matrix multiplication:

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

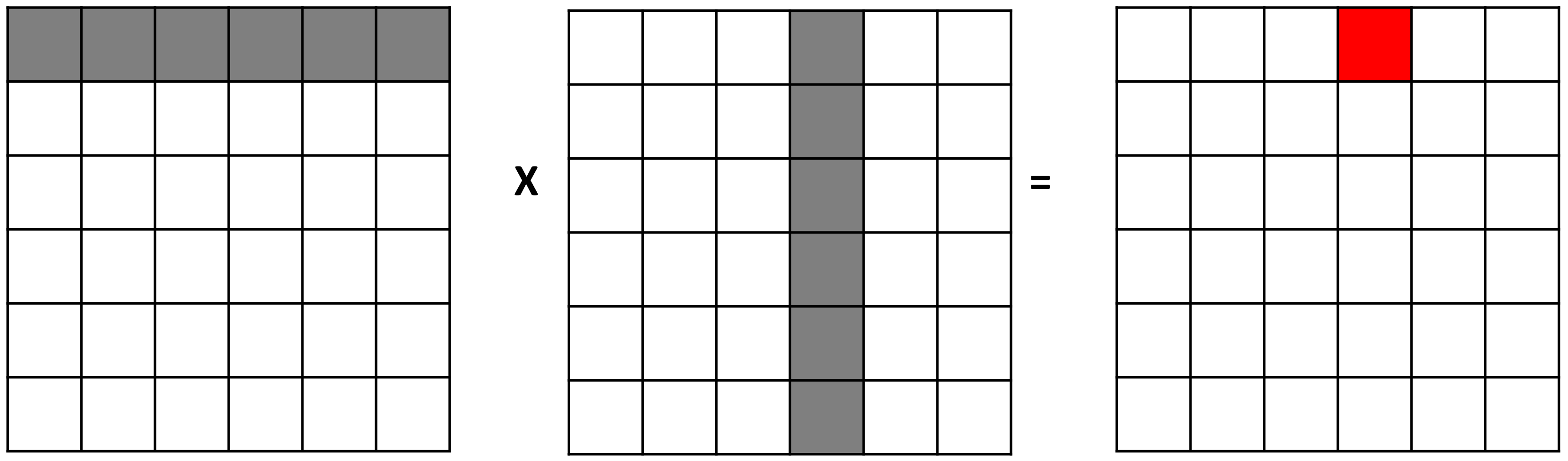


To generate one output element (red color), the 6 elements of each input matrix (gray color) will be accessed.

Loop Blocking

In the traditional matrix multiplication:

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

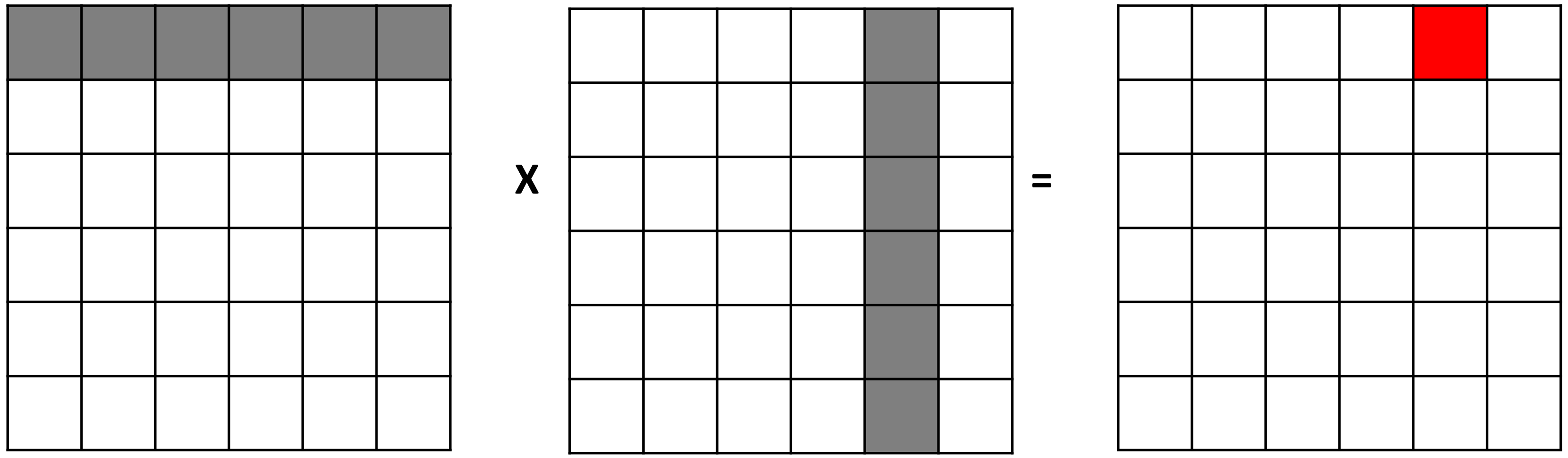


To generate one output element (red color), the 6 elements of each input matrix (gray color) will be accessed.

Loop Blocking

In the traditional matrix multiplication:

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



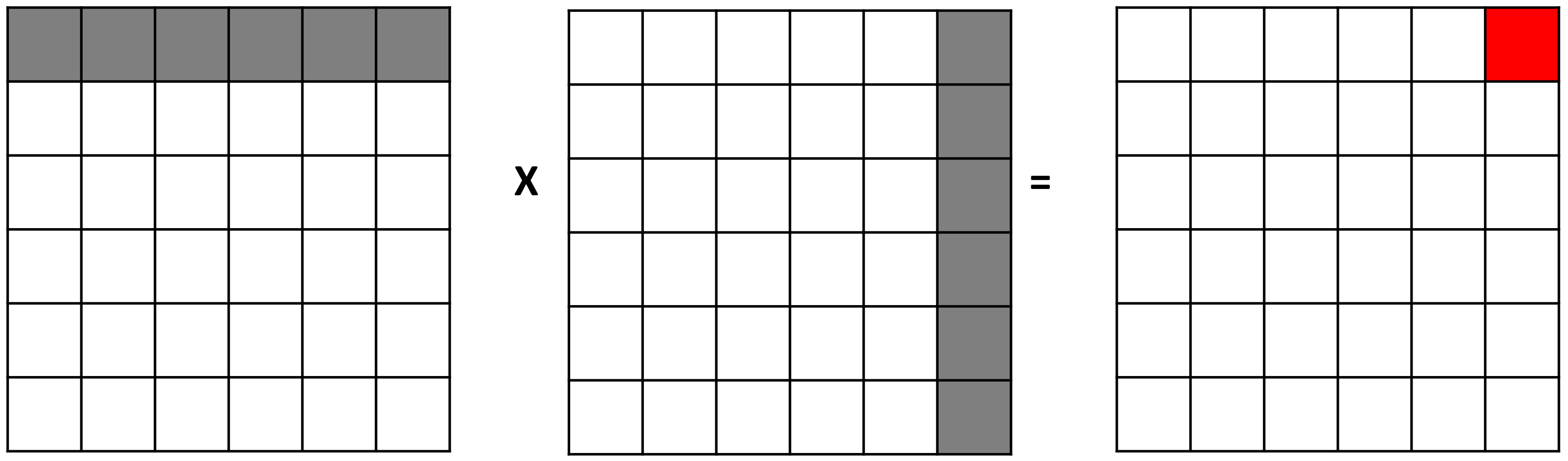
To generate one output element (red color), the 6 elements of each input matrix (gray color) will be accessed.

Loop Blocking

In the traditional matrix multiplication:

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

The second matrix may always encounter misses because individual data are from different cache blocks



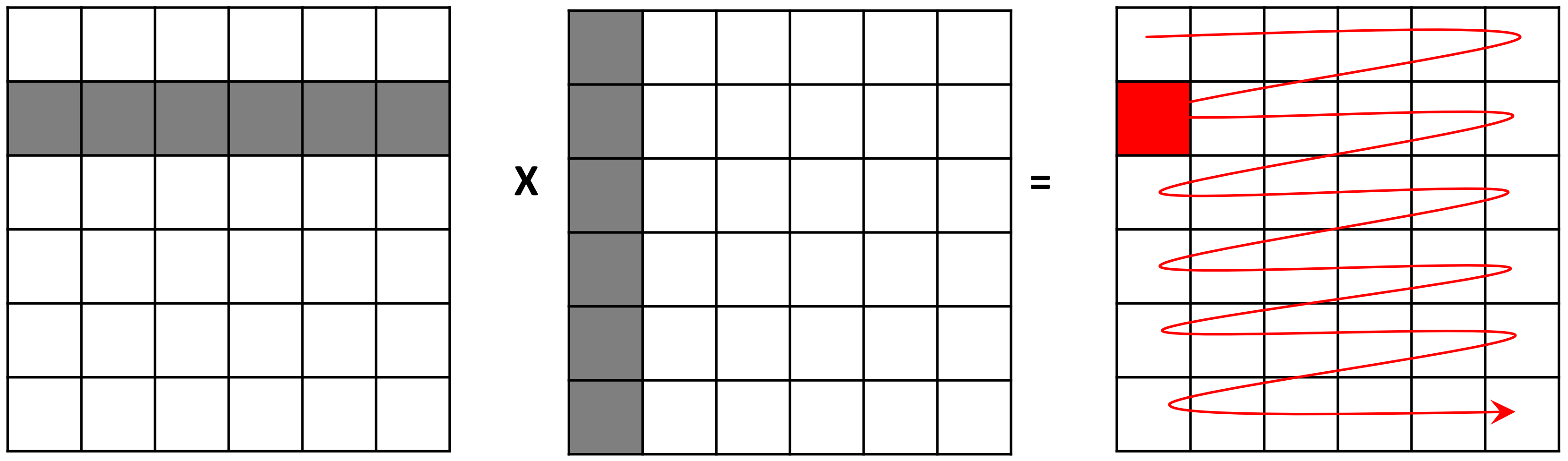
To generate one output element (red color), the 6 elements of each input matrix (gray color) will be accessed.

Loop Blocking

In the traditional matrix multiplication:

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

If N is large, the cache line that has been accessed is likely to be replaced before next access



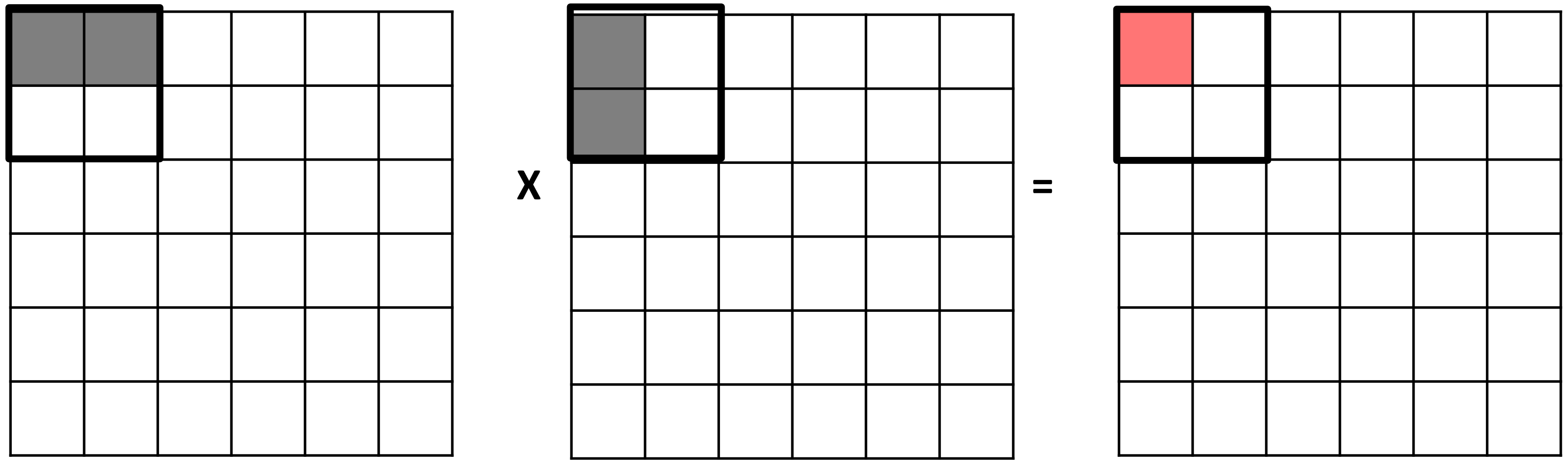
To generate one output element (red color), the 6 elements of each input matrix (gray color) will be accessed.

Loop Blocking

In the block-based matrix multiplication:

```
/* After (block size = 4) */
for (i=0; i<N; i=i+b)
 for (j=0; j<N; j=j+b)
  for (k=0; k<N; k=k+b)
   for (ii=i; ii<i+b; ii++)
        for (jj=j; jj<j+b; jj++)
         for (kk=k; kk<k+b; kk++)
                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```

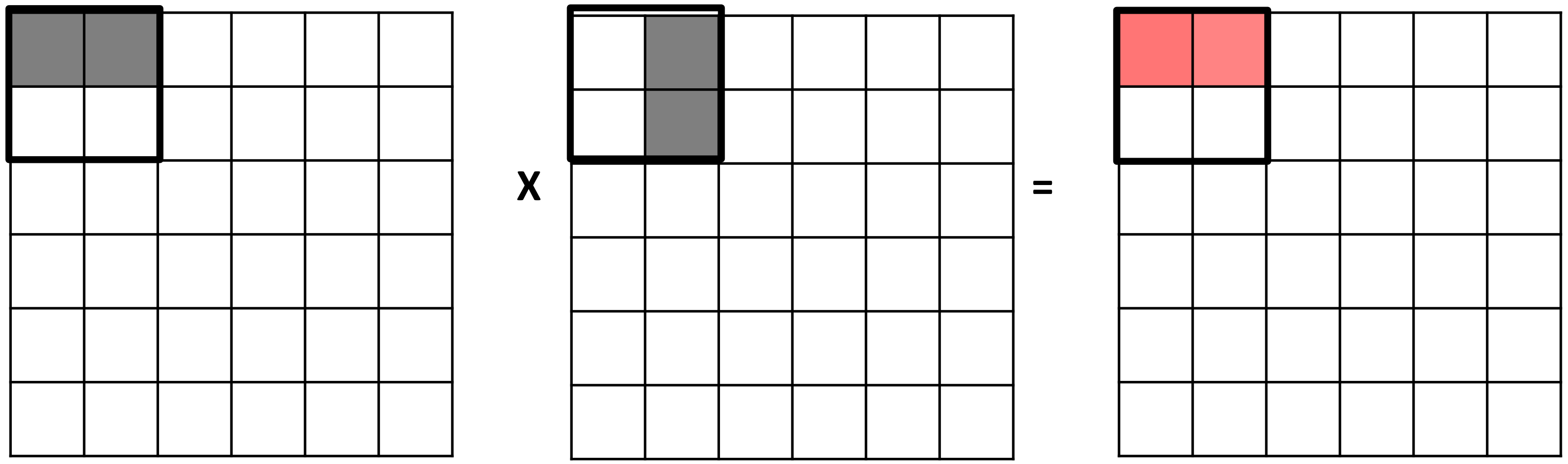
Matrix is computed in a smaller block unit → higher locality



Loop Blocking

In the block-based matrix multiplication:

```
/* After (block size = 4) */
for (i=0; i<N; i=i+b)
 for (j=0; j<N; j=j+b)
  for (k=0; k<N; k=k+b)
   for (ii=i; ii<i+b; ii++)
        for (jj=j; jj<j+b; jj++)
         for (kk=k; kk<k+b; kk++)
               C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```

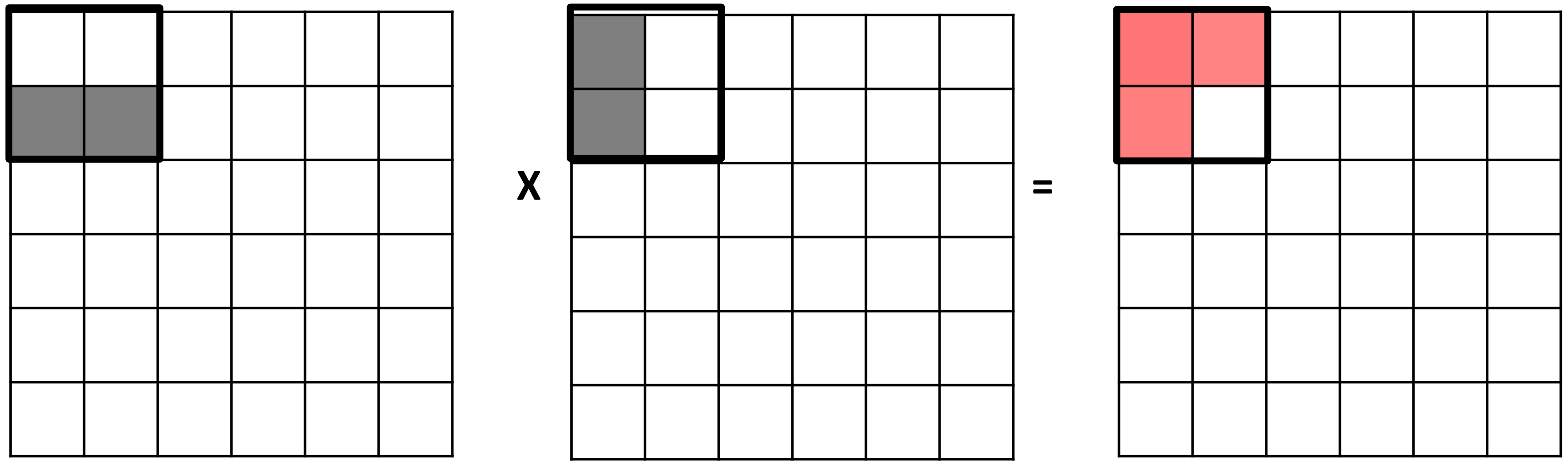


SJSU SAN JOSÉ STATE UNIVERSITY

Loop Blocking

In the block-based matrix multiplication:

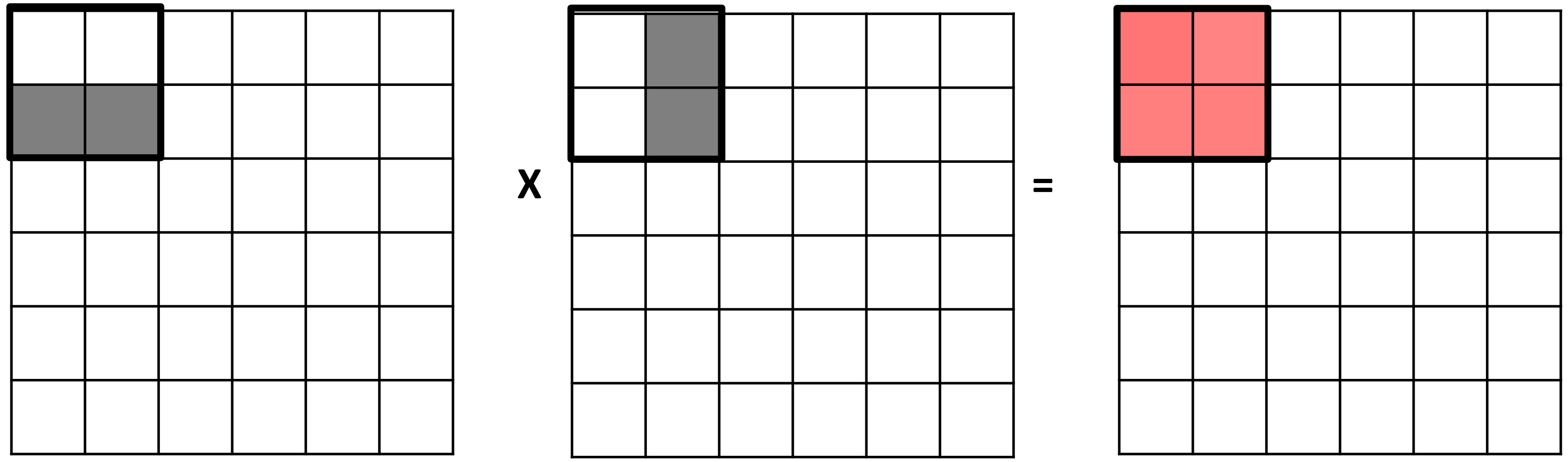
```
/* After (block size = 4) */
for (i=0; i<N; i=i+b)
 for (j=0; j<N; j=j+b)
  for (k=0; k<N; k=k+b)
   for (ii=i; ii<i+b; ii++)
        for (jj=j; jj<j+b; jj++)
         for (kk=k; kk<k+b; kk++)
               C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```



Loop Blocking

In the block-based matrix multiplication:

```
/* After (block size = 4) */
for (i=0; i<N; i=i+b)
 for (j=0; j<N; j=j+b)
  for (k=0; k<N; k=k+b)
   for (ii=i; ii<i+b; ii++)
        for (jj=j; jj<j+b; jj++)
         for (kk=k; kk<k+b; kk++)
               C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```

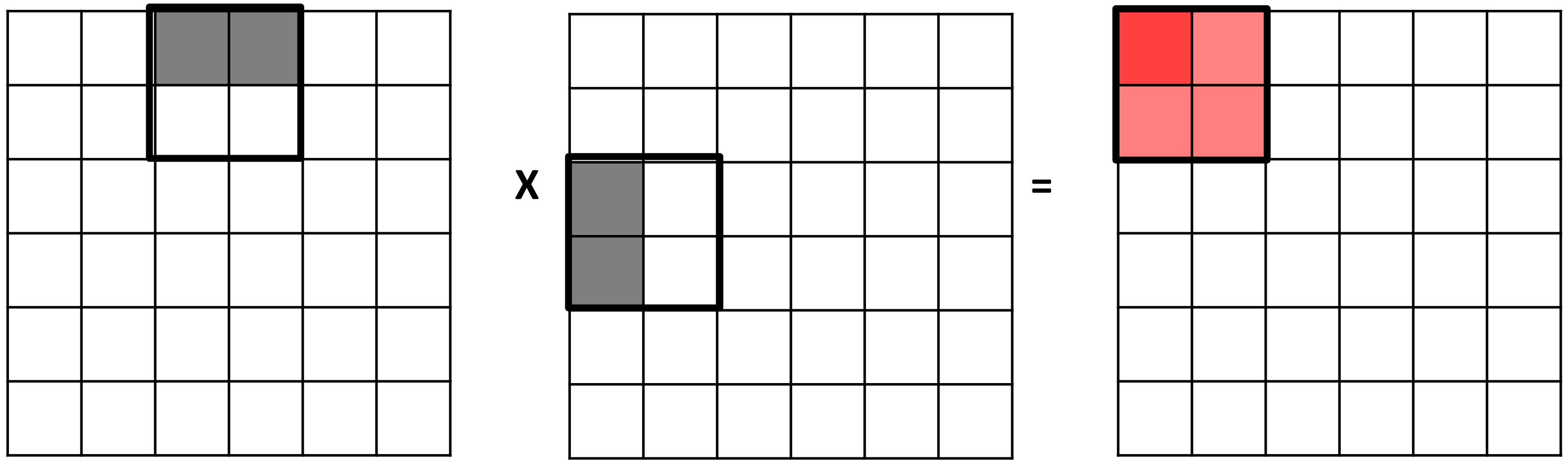


SJSU SAN JOSÉ STATE UNIVERSITY

Loop Blocking

In the block-based matrix multiplication:

```
/* After (block size = 4) */
for (i=0; i<N; i=i+b)
 for (j=0; j<N; j=j+b)
  for (k=0; k<N; k=k+b)
   for (ii=i; ii<i+b; ii++)
        for (jj=j; jj<j+b; jj++)
         for (kk=k; kk<k+b; kk++)
               C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```

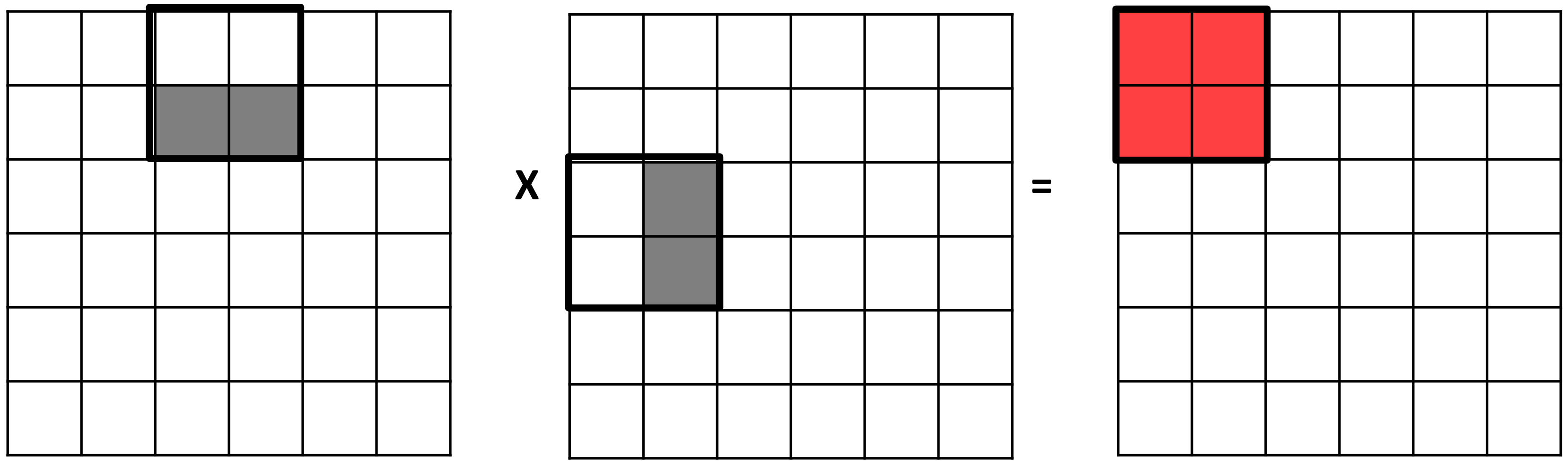


SJSU SAN JOSÉ STATE UNIVERSITY

Loop Blocking

In the block-based matrix multiplication:

```
/* After (block size = 4) */
for (i=0; i<N; i=i+b)
 for (j=0; j<N; j=j+b)
  for (k=0; k<N; k=k+b)
   for (ii=i; ii<i+b; ii++)
        for (jj=j; jj<j+b; jj++)
         for (kk=k; kk<k+b; kk++)
                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```

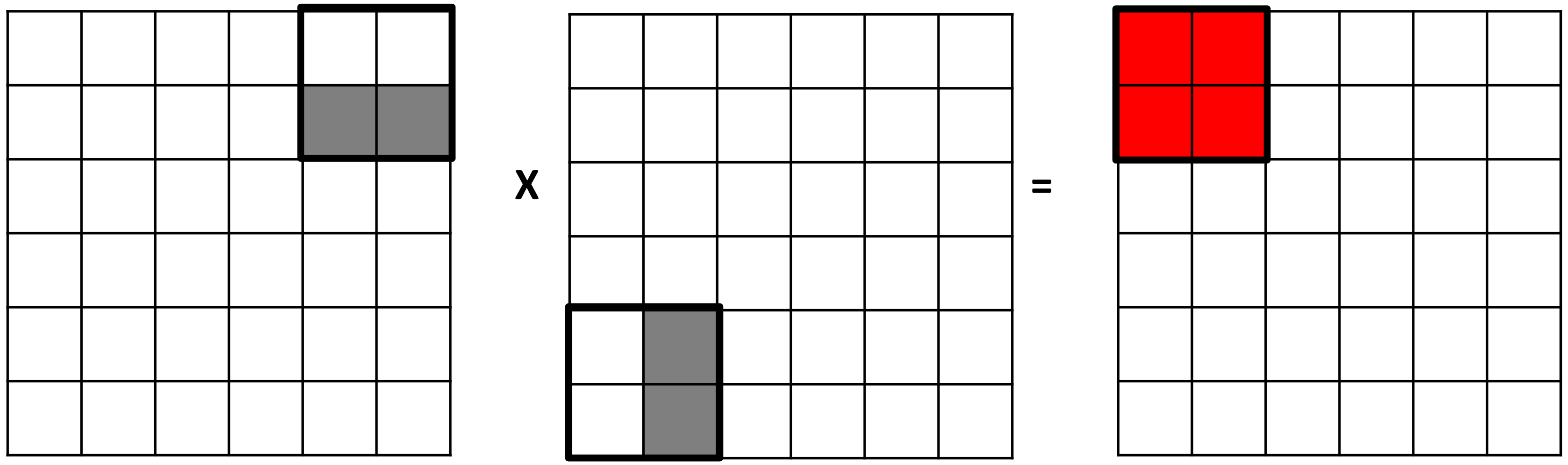


SJSU SAN JOSÉ STATE UNIVERSITY

Loop Blocking

In the block-based matrix multiplication:

```
/* After (block size = 4) */
for (i=0; i<N; i=i+b)
 for (j=0; j<N; j=j+b)
  for (k=0; k<N; k=k+b)
   for (ii=i; ii<i+b; ii++)
        for (jj=j; jj<j+b; jj++)
         for (kk=k; kk<k+b; kk++)
                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```



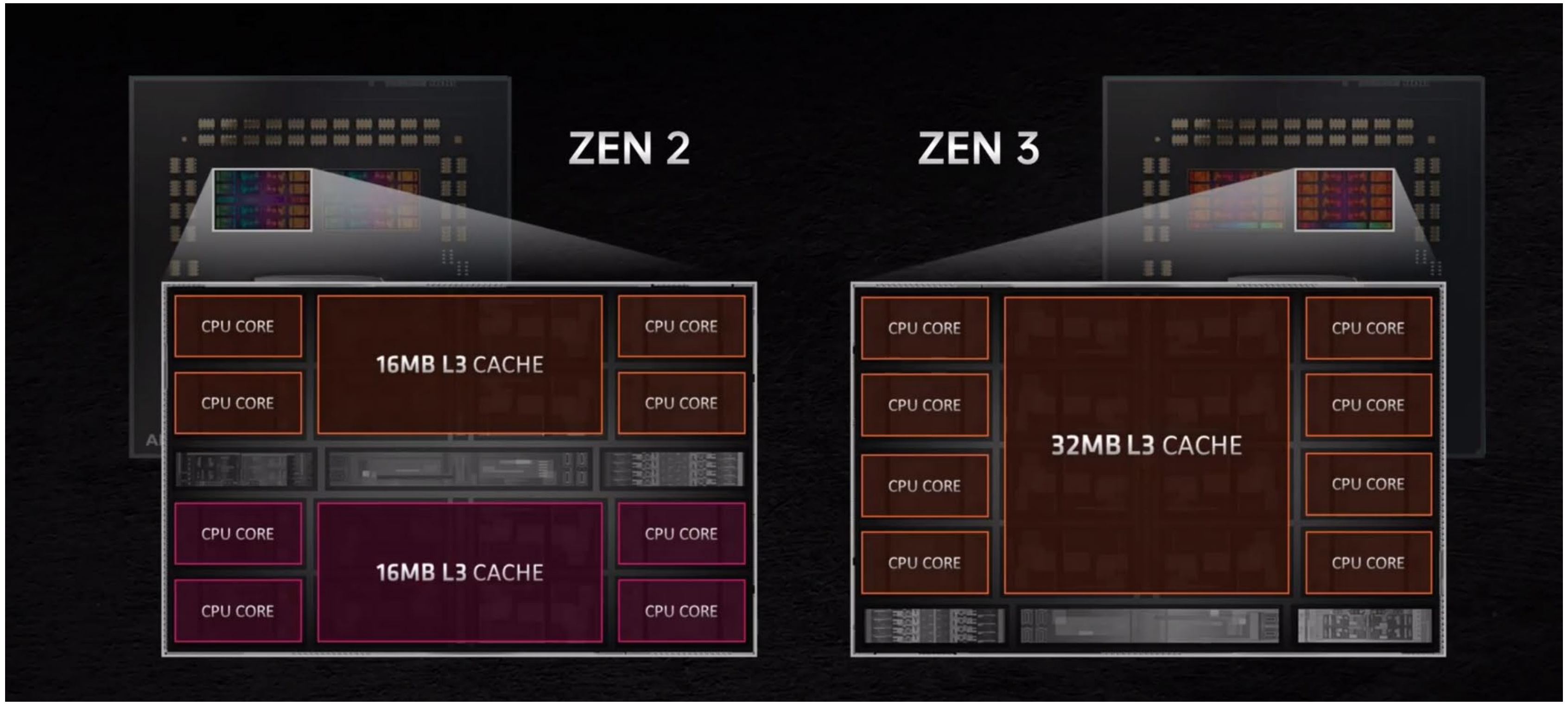
Cache Miss Classification: The 3 C's

- **Compulsory (cold) Misses**
  - On the 1st reference to a block
  - Related to # blocks accessed by a code, not related to the configuration of a cache
- **Capacity Misses**
  - The program's working set size exceeds the cache capacity
- **Conflict Misses**
  - Multiple memory blocks map to the same set in set-associative caches

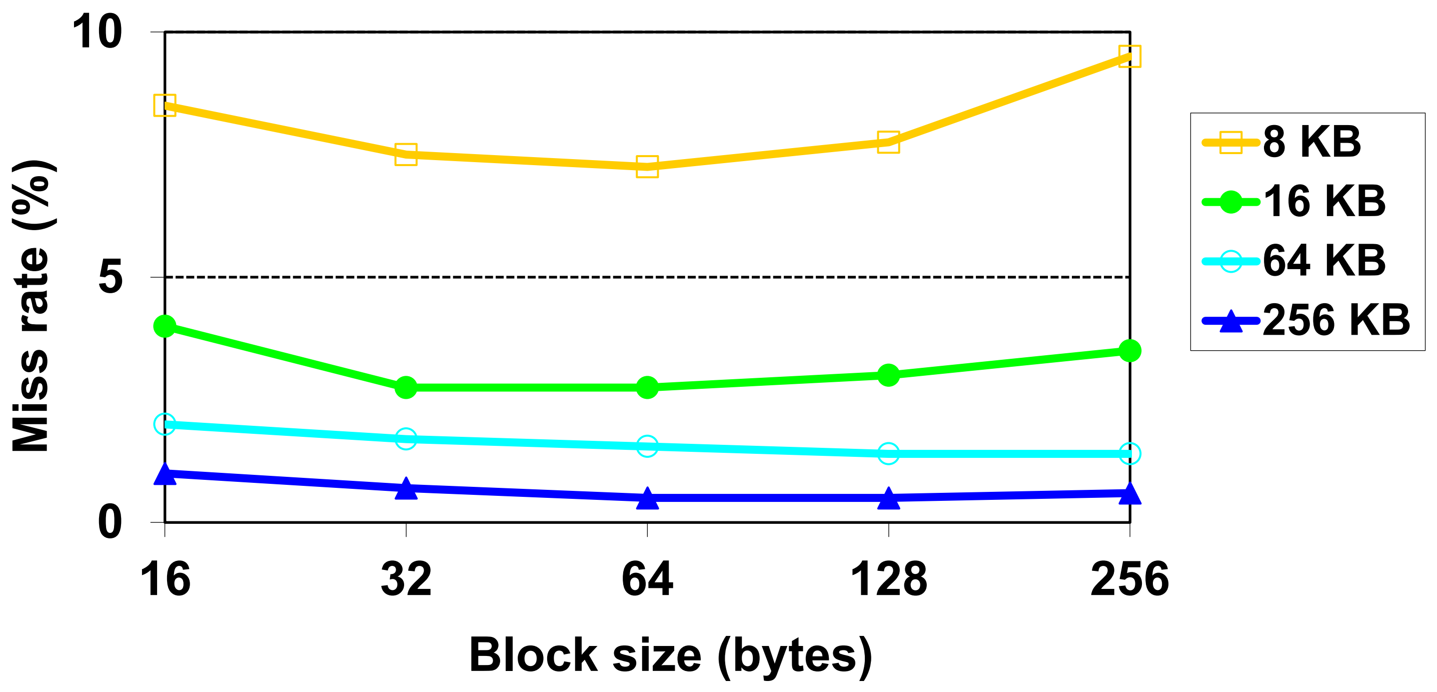
Reduce Miss Rate (2): Reduce the 3 C’s

- **Increase Cache Size**
  - Reduce miss for: capacity miss, conflict miss
  - But has many limitations
- **Increase Associativity (cache size unchanged)**
  - Reduce miss for: conflict miss
  - But may increase access latency
- **Increase Cache Block Size (cache size unchanged)**
  - Reduce miss for: compulsory
  - But may increase miss penalty (more data will be evicted and fetched)
  - Very large blocks could increase miss rate (i.e., increase capacity miss)

Increase Cache Size

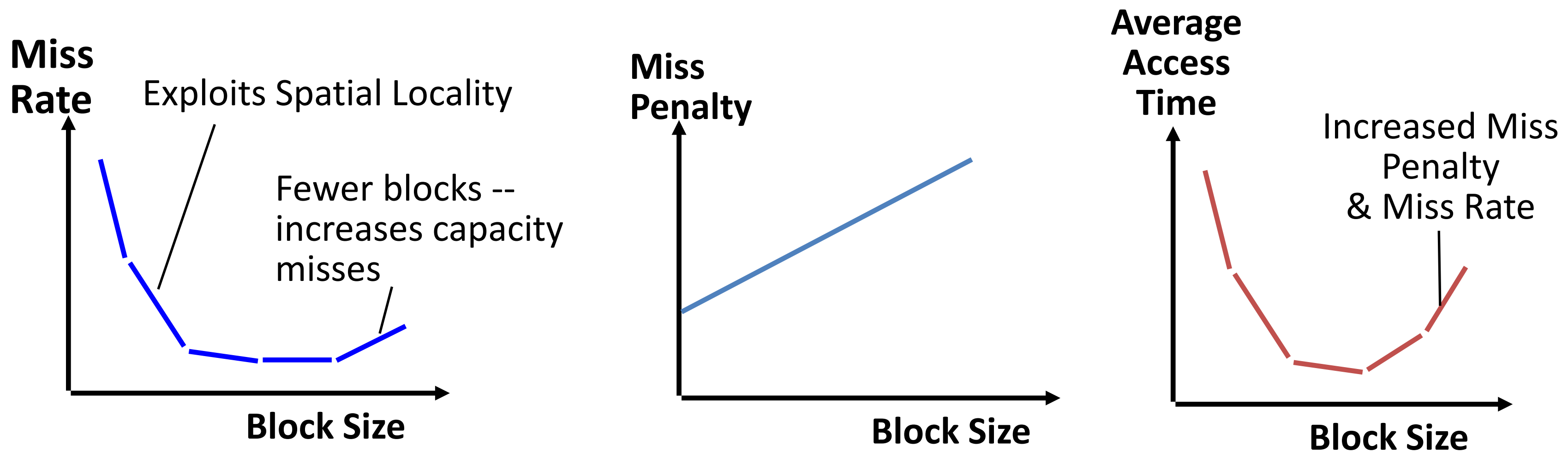


Miss Rate vs Block Size vs Cache Size



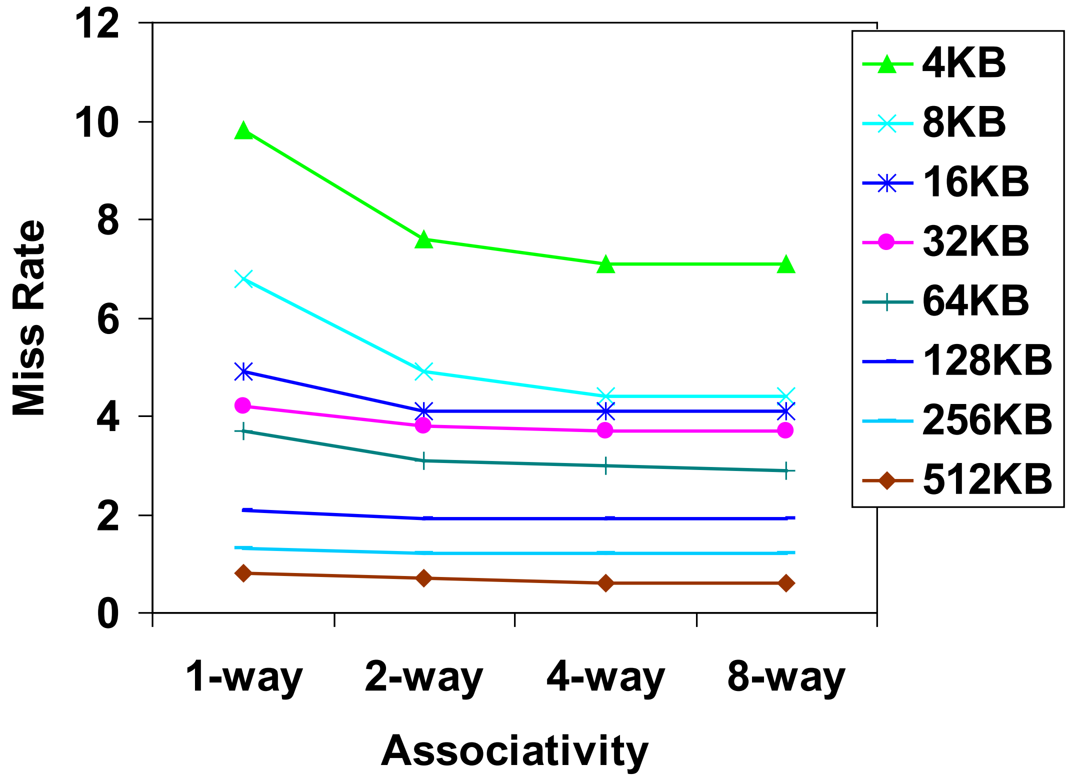
- Cache size – the larger the better
- Block size – tradeoffs **Why?**

Block Size Tradeoff



Average Memory Access Time = Hit Time + Miss Penalty x Miss Rate

Benefits of Set Associative Caches



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

SJSU SAN JOSÉ STATE UNIVERSITY

Data from Hennessy & Patterson, *Computer Architecture*

Reduce Miss Rate (3): Policies

- **Use appropriate policies**
  - LRU + write-back works for most applications
  - Many GPUs use write-hit write-evict + write-miss no-write-allocate
  - Can even dynamically change policy according to profiling
- **Use more advanced policies**
  - E.g., ML based

Reduce Miss Rate (4): Multi-level Caches

- Having a unified L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache

- L1 cache should focus on minimizing hit time in support of a shorter clock cycle

- Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times

Split vs. Unified Caches

- **Split L1I$ and L1D$: to optimize t_{hit}**
  - Low capacity/associativity/block size (reduce t_{hit})
  - Minimize structural hazards
  - Can optimize L1I$ for wide output and no writes

- **Unified L2: to optimize hit rate**
  - t_{hit} is less important due to less frequent accesses & long delay already
  - High capacity/associativity/block size (reduce $\%_{miss}$)
    - Unused instr capacity can be used for data (fewer capacity misses)
    - Instr / data conflicts (small to no increase for conflict misses in a large cache)
  - Instr / data structural hazards are rare (would take a simultaneous L1I$ and L1D$ miss)

Reduce Miss Rate (5): Prefetching

- **Hardware prefetching**
  - Fetch blocks into the cache proactively (speculatively)
  - Key is to anticipate the upcoming miss addresses accurately
  - Relies on having unused memory bandwidth available
- **A simple case is to use next block prefetching**
  - Miss on address X → anticipate next reference miss on X + block-size
  - Works well for instructions (sequential execution) and for arrays of data
- **Need to initiate prefetches sufficiently in advance**
- **If prefetched instruction/data is not used, the cache is polluted with unnecessary data (possibly evicting useful data)**

Other than Miss Rates

- **Cache bypassing (e.g., L1D)**
  - Reduce latency if L1 misses
- **Sector cache**
  - Fetch only a portion of the cache line
  - Not the same as reducing cache line size
- **Value prediction**
  - Using history value to predict future values
  - Roll back if predict wrong
  - Or accept a certain quality loss if the application is error tolerant

Let us Conclude

What are the operations in cache for write-miss when using write-back?

evict victim block + write back evicted block if dirty + fetch block from lower level + store word to block

List two ways of code optimizations to improve locality?

Loop interchange, Loop blocking

Conclusion Time

What are some ways to reduce cache misses?

Software methods, reduce 3C’s, changing policies, multi-level caches, prefetching, value prediction …

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY