Lecture 9. Memory Hierarchy, Locality, and Caches

Introductions to Data Systems and Data Design

Ce Zhang

Journey So Far

We have learned how to make the CPU go fast:

- **ILP:** Keep the pipeline full, avoid stalls, help the branch predictor
- **SIMD:** Process multiple data elements per instruction

Journey So Far

We have learned how to make the CPU go fast:

- **ILP:** Keep the pipeline full, avoid stalls, help the branch predictor
- **SIMD:** Process multiple data elements per instruction

If every memory access took 1 cycle, we would be done.

Journey So Far

We have learned how to make the CPU go fast:

- **ILP:** Keep the pipeline full, avoid stalls, help the branch predictor
- **SIMD:** Process multiple data elements per instruction

If every memory access took 1 cycle, we would be done.

But it doesn't. A main memory access costs ~200 cycles.

The next question: **how do we keep the memory system from slowing us down?**

The Memory Wall

Typical Memory Hierarchy

Level	Name	Size	Latency (cycles)
L0	Registers	~1 KB	0
L1	L1 Cache	32 KB	~4
L2	L2 Cache	256 KB	~12
L3	L3 Cache	8 MB	~42
-	Main Memory	64 GB	~200
-	SSD	1 TB	~10,000

Typical Memory Hierarchy

Level	Name	Size	Latency (cycles)
L0	Registers	~1 KB	0
L1	L1 Cache	32 KB	~4
L2	L2 Cache	256 KB	~12
L3	L3 Cache	8 MB	~42
-	Main Memory	64 GB	~200
-	SSD	1 TB	~10,000

Key insight: Smaller = Faster = More expensive per byte

Each level caches data from the level below.

Locality

Why Caches Work: Locality

Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

Why Caches Work: Locality

Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

Temporal locality:

- Recently referenced items are likely to be referenced again
- Example: Loop counter variable `i`

Why Caches Work: Locality

Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

Temporal locality:

- Recently referenced items are likely to be referenced again
- Example: Loop counter variable `i`

Spatial locality:

- Items with nearby addresses tend to be referenced close together in time
- Example: Iterating through an array sequentially

Locality Example

```
double sum = 0;
for (int i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Locality Example

```
double sum = 0;
for (int i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Data locality:

- `sum`: High temporal locality (accessed every iteration)
- `a[i]`: High spatial locality (consecutive access)

Instruction locality:

- Temporal: Loop body executed repeatedly
- Spatial: Instructions stored contiguously

Row-Major vs Column-Major Access

```
// Row-major traversal (good locality)
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        sum += a[i][j];

// Column-major traversal (poor locality)
for (int j = 0; j < N; j++)
    for (int i = 0; i < M; i++)
        sum += a[i][j];
```

Row-Major vs Column-Major Access

```
// Row-major traversal (good locality)
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        sum += a[i][j];

// Column-major traversal (poor locality)
for (int j = 0; j < N; j++)
    for (int i = 0; i < M; i++)
        sum += a[i][j];
```

In C/C++, 2D arrays are stored in **row-major order**:

`a[0][0], a[0][1], ..., a[0][N-1], a[1][0], a[1][1], ...`

Row-major traversal has stride 1; column-major has stride N.

Measuring the Impact

```
$ cd examples && make locality
```

```
// locality_test.cpp
#include <iostream>
#include <chrono>
using namespace std;

const int M = 4096, N = 4096;
double a[M][N];

double sum_rows() {
    double sum = 0;
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}

double sum_cols() {
    double sum = 0;
    for (int j = 0; j < N; j++)
        for (int i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

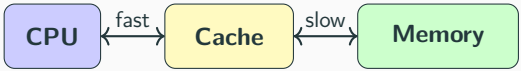
Run It Yourself

```
int main() {
    // Initialize
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            a[i][j] = 1.0;

    auto t1 = chrono::high_resolution_clock::now();
    volatile double r1 = sum_rows();
    auto t2 = chrono::high_resolution_clock::now();
    volatile double r2 = sum_cols();
    auto t3 = chrono::high_resolution_clock::now();

    cout << "Row-major: "
         << chrono::duration<double,milli>(t2-t1).count() << " ms\n";
    cout << "Col-major: "
         << chrono::duration<double,milli>(t3-t2).count() << " ms\n";
    return 0;
}
```

Cache Mechanics

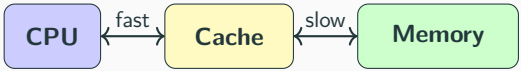
What is a Cache?

Cache: Small, fast memory that stores copies of frequently accessed data.



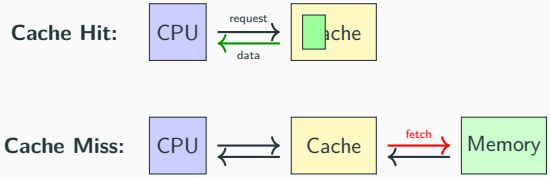
What is a Cache?

Cache: Small, fast memory that stores copies of frequently accessed data.



- **Temporal locality:** Keep recently accessed data in cache
- **Spatial locality:** Transfer data in **blocks** (64 bytes = 8 doubles)

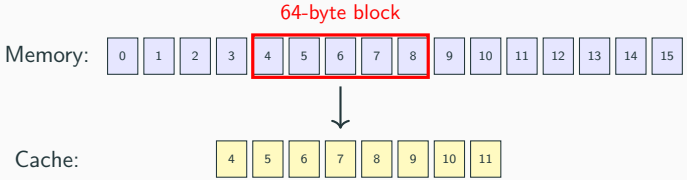
Cache Hit and Miss



- **Hit:** Data found in cache (fast, ~4 cycles for L1)
- **Miss:** Data not in cache, must fetch from memory (~200 cycles)

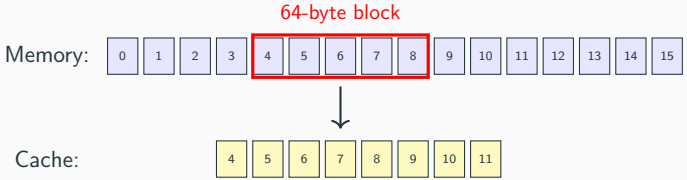
Cache Block and Spatial Locality

When a cache miss occurs, an entire **block** is loaded:



Cache Block and Spatial Locality

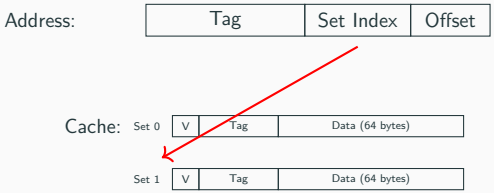
When a cache miss occurs, an entire **block** is loaded:



Block size: 64 bytes = 8 doubles on modern CPUs

Accessing `a[4]` loads `a[0]` through `a[7]` into cache.

Cache Organization: Direct Mapped

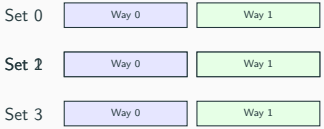


Direct mapped: Each memory address maps to exactly one cache location.

Set index = (address / block_size) mod num_sets

Cache Organization: Set Associative

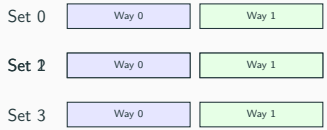
2-way set associative:



Each address can go in 2 locations within its set

Cache Organization: Set Associative

2-way set associative:



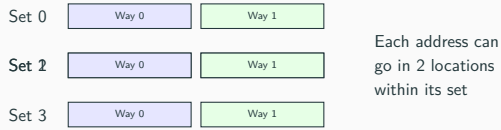
Each address can go in 2 locations within its set

E-way set associative: Each address can map to E different locations.

- E = 1: Direct mapped
- E = S: Fully associative (any location)
- Typical: L1 is 8-way

Cache Organization: Set Associative

2-way set associative:



E-way set associative: Each address can map to E different locations.

- E = 1: Direct mapped
- E = S: Fully associative (any location)
- Typical: L1 is 8-way

Why higher associativity helps: With direct mapped, two addresses that map to the same set evict each other — even if the rest of the cache is empty. More ways per set = fewer conflict misses.

The Three C's of Cache Misses

Compulsory (Cold) Miss:

- First access to a block
- Unavoidable

The Three C's of Cache Misses

Compulsory (Cold) Miss:

- First access to a block
- Unavoidable

Capacity Miss:

- Working set larger than cache
- Solution: Reduce working set, improve locality

The Three C's of Cache Misses

Compulsory (Cold) Miss:

- First access to a block
- Unavoidable

Capacity Miss:

- Working set larger than cache
- Solution: Reduce working set, improve locality

Conflict Miss:

- Cache has space, but blocks map to same set
- Solution: Increase associativity, change data layout

Operational Intensity

Operational Intensity

$$I(n) = \frac{W(n)}{Q(n)}$$

- $W(n) =$ Number of floating-point operations
- $Q(n) =$ Bytes transferred between cache and memory

Operational Intensity

$$I(n) = \frac{W(n)}{Q(n)}$$

- $W(n) =$ Number of floating-point operations
- $Q(n) =$ Bytes transferred between cache and memory

Operation	$W(n)$	$Q(n)$	$I(n)$
Vector sum: $y = x + y$	$O(n)$	$O(n)$	$O(1)$
Matrix-vector: $y = Ax$	$O(n^2)$	$O(n^2)$	$O(1)$
Matrix-matrix: $C = AB$	$O(n^3)$	$O(n^2)$	$O(n)$

Compute Bound vs Memory Bound

Memory bound: Low operational intensity

- Performance limited by memory bandwidth
- Example: Vector operations, most BLAS Level 1 & 2

Compute Bound vs Memory Bound

Memory bound: Low operational intensity

- Performance limited by memory bandwidth
- Example: Vector operations, most BLAS Level 1 & 2

Compute bound: High operational intensity

- Performance limited by compute throughput
- Example: Matrix multiplication, convolutions

Compute Bound vs Memory Bound

Memory bound: Low operational intensity

- Performance limited by memory bandwidth
- Example: Vector operations, most BLAS Level 1 & 2

Compute bound: High operational intensity

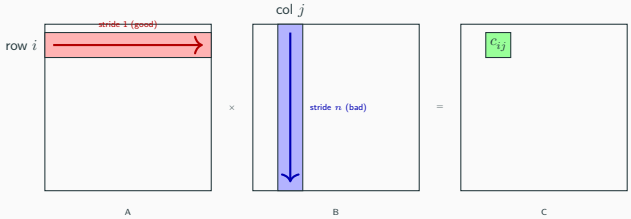
- Performance limited by compute throughput
- Example: Matrix multiplication, convolutions

Key insight: MMM has $O(n)$ operational intensity – we can hide memory latency with computation if we're clever about data reuse.

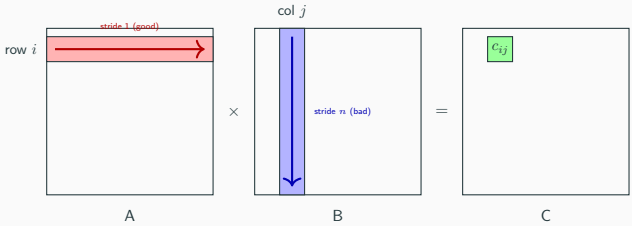
Blocking for Cache

Matrix Multiplication: Naive

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```



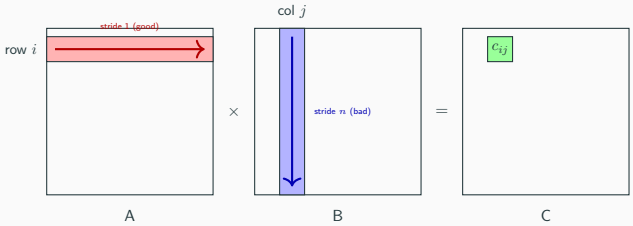
Naive MMM: Cache Analysis



Cache analysis (assuming cache « matrix size):

- For each of n^2 output elements:
  - Read n elements from row of A (n/8 misses)
  - Read n elements from column of B (n misses - poor locality!)
- Total: $\approx 9n^3/8$ cache misses

Naive MMM: Cache Analysis



Cache analysis (assuming cache « matrix size):

- For each of n^2 output elements:
  - Read n elements from row of A (n/8 misses)
  - Read n elements from column of B (n misses - poor locality!)
- Total: $\approx 9n^3/8$ cache misses

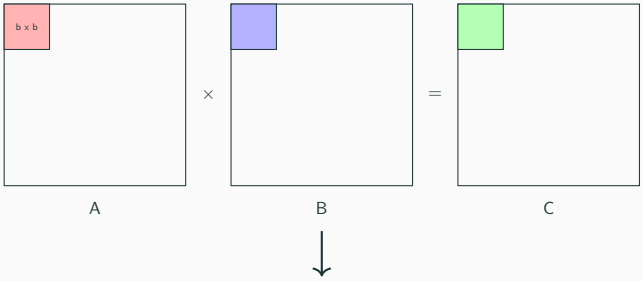
Operational intensity (naive):

$$I = \frac{W}{Q} = \frac{2n^3}{(9n^3/8) \times 64} = \frac{2}{72} \approx \frac{1}{36}$$

Only ~0.03 FLOPs per byte transferred – **heavily memory bound!**

The Blocking Idea

Instead of computing one element at a time, compute **blocks**:

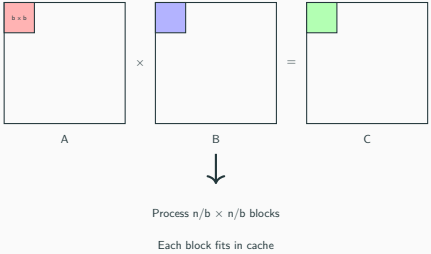


Goal: Keep working set in cache to exploit locality.

Blocked Matrix Multiplication

```
for (int ii = 0; ii < n; ii += b)
    for (int jj = 0; jj < n; jj += b)
        for (int kk = 0; kk < n; kk += b)
            // Multiply b x b blocks
            for (int i = ii; i < ii + b; i++)
                for (int j = jj; j < jj + b; j++)
                    for (int k = kk; k < kk + b; k++)
                        C[i][j] += A[i][k] * B[k][j];
```

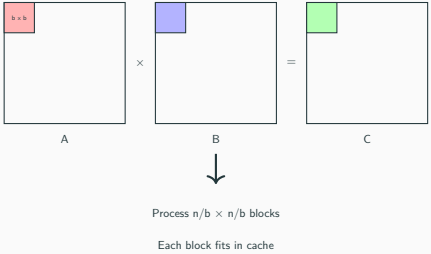
Blocked MMM: Cache Analysis



Cache analysis with blocking: - One block multiply: $b \times b$ block of A + $b \times b$ block of B + $b \times b$ block of C - If $3b^2 \leq \gamma$ (cache size in elements), all three blocks fit in cache - Loading one $b \times b$ block $= b^2/8$ misses (64-byte lines, 8 doubles each) - Three blocks $= 3 \times b^2/8$ misses per block multiply - There are $(n/b)^3$ block multiplies in total - **Total misses:** $(n/b)^3 \times 3b^2/8 = 3n^3/(8b)$ . . .

Speedup over naive (in cache misses): $\frac{9n^3/8}{3n^3/(8b)} = 3b$

Blocked MMM: Cache Analysis



Cache analysis with blocking: - One block multiply: $b \times b$ block of A + $b \times b$ block of B + $b \times b$ block of C - If $3b^2 \leq \gamma$ (cache size in elements), all three blocks fit in cache - Loading one $b \times b$ block $= b^2/8$ misses (64-byte lines, 8 doubles each) - Three blocks $= 3 \times b^2/8$ misses per block multiply - There are $(n/b)^3$ block multiplies in total - **Total misses:** $(n/b)^3 \times 3b^2/8 = 3n^3/(8b)$ . . .

Speedup over naive (in cache misses): $\frac{9n^3/8}{3n^3/(8b)} = 3b$

Operational intensity (blocked):

$$I = \frac{W}{Q} = \frac{2n^3}{(3n^3/(8b)) \times 64} = \frac{b}{12} = O(b)$$

With $b = 32$: $I \approx 2.7$ FLOPs/byte – ~**96x improvement over naive!**

Choosing the Block Size

Constraint: Three $b \times b$ blocks must fit in cache

$$3b^2 \leq \gamma \quad \Rightarrow \quad b \leq \sqrt{\gamma/3}$$

Choosing the Block Size

Constraint: Three $b \times b$ blocks must fit in cache

$$3b^2 \leq \gamma \quad \Rightarrow \quad b \leq \sqrt{\gamma/3}$$

Example: L1 cache = 32 KB = 4096 doubles

$$b \leq \sqrt{4096/3} \approx 37$$

Use $b = 32$ (power of 2 for alignment).

Choosing the Block Size

Constraint: Three $b \times b$ blocks must fit in cache

$$3b^2 \leq \gamma \quad \Rightarrow \quad b \leq \sqrt{\gamma/3}$$

Example: L1 cache = 32 KB = 4096 doubles

$$b \leq \sqrt{4096/3} \approx 37$$

Use $b = 32$ (power of 2 for alignment).

Complete Blocking Example

```
$ cd examples && make mmm
```

```
// mmm_blocked.cpp
#include <iostream>
#include <chrono>
#include <cstring>
using namespace std;

const int N = 1024;
const int BLOCK = 32;
double A[N*N], B[N*N], C[N*N];

void mmm_naive() {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i*N + j] += A[i*N + k] * B[k*N + j];
}

void mmm_blocked() {
    for (int ii = 0; ii < N; ii += BLOCK)
      for (int jj = 0; jj < N; jj += BLOCK)
        for (int kk = 0; kk < N; kk += BLOCK)
          for (int i = ii; i < ii + BLOCK; i++)
            for (int j = jj; j < jj + BLOCK; j++)
              for (int k = kk; k < kk + BLOCK; k++)
                C[i*N + j] += A[i*N + k] * B[k*N + j];
}
```

Run It Yourself

```
int main() {
    // Initialize
    for (int i = 0; i < N*N; i++) {
        A[i] = 1.0; B[i] = 1.0; C[i] = 0.0;
    }

    auto t1 = chrono::high_resolution_clock::now();
    mmm_naive();
    auto t2 = chrono::high_resolution_clock::now();

    memset(C, 0, sizeof(C));

    auto t3 = chrono::high_resolution_clock::now();
    mmm_blocked();
    auto t4 = chrono::high_resolution_clock::now();

    double gflops = 2.0 * N * N * N / 1e9;
    cout << "Naive:   " << gflops / chrono::duration<double>(t2-t1).count()
         << " GFLOPS\n";
    cout << "Blocked: " << gflops / chrono::duration<double>(t4-t3).count()
         << " GFLOPS\n";
}
```

Conflict Misses and Power-of-2 Strides

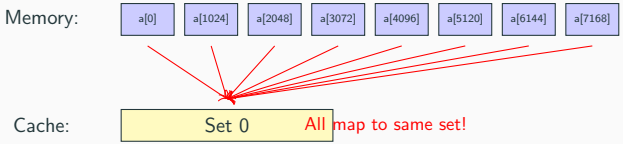
The Power-of-2 Problem

```
// Accessing every 1024th element
for (int i = 0; i < n; i += 1024)
    sum += a[i];
```

The Power-of-2 Problem

```
// Accessing every 1024th element
for (int i = 0; i < n; i += 1024)
    sum += a[i];
```

Problem: If stride is a power of 2, elements may map to same cache set!



Where Conflict Misses Occur

Common scenarios:

1. Column access in power-of-2 sized matrices
2. FFT with power-of-2 sizes
3. Stencil computations on power-of-2 grids

Where Conflict Misses Occur

Common scenarios:

1. Column access in power-of-2 sized matrices
2. FFT with power-of-2 sizes
3. Stencil computations on power-of-2 grids

Example: 2D array column access

```
double a[1024][1024];  // Power of 2!

// Column access - all elements map to same cache sets
for (int i = 0; i < 1024; i++)
    sum += a[i][0];  // Stride = 1024 doubles = 8KB
```

Solutions to Conflict Misses

1. **Padding:** Add extra elements to break alignment

```
double a[1024][1024 + 8];  // +8 padding
```

Solutions to Conflict Misses

1. **Padding:** Add extra elements to break alignment

```
double a[1024][1024 + 8];  // +8 padding
```

2. **Use non-power-of-2 sizes:**

```
double a[1000][1000];  // Avoid power of 2
```

Solutions to Conflict Misses

1. **Padding:** Add extra elements to break alignment

```
double a[1024][1024 + 8];  // +8 padding
```

2. **Use non-power-of-2 sizes:**

```
double a[1000][1000];  // Avoid power of 2
```

3. **Blocking with appropriate sizes:**

Choose block size that doesn't create power-of-2 strides.

Swizzling: A Smarter Solution

Problem with padding: Wastes memory, changes array dimensions.

Swizzling: Remap indices to spread accesses across cache sets without changing array size.

Swizzling: A Smarter Solution

Problem with padding: Wastes memory, changes array dimensions.

Swizzling: Remap indices to spread accesses across cache sets without changing array size.

Key idea: Use XOR to transform column index based on row index.

$$j_{\text{swizzled}} = j \oplus (i \wedge \text{mask})$$

where $\oplus$ is XOR and $\wedge$ is bitwise AND.

Why Does XOR Work?

XOR properties:

- $a \oplus 0 = a$ (identity)
- $a \oplus a = 0$ (self-inverse)
- $a \oplus b \oplus b = a$ (reversible!)

Why Does XOR Work?

XOR properties:

- $a \oplus 0 = a$ (identity)
- $a \oplus a = 0$ (self-inverse)
- $a \oplus b \oplus b = a$ (reversible!)

Key insight: XOR with different row indices produces different results:

Row i	i AND 3	0 XOR (i AND 3)	Result
0	0	0 XOR 0	**0**
1	1	0 XOR 1	**1**
2	2	0 XOR 2	**2**
3	3	0 XOR 3	**3**

Column 0 now maps to 4 different physical columns!

Swizzling Example: Reading Column 0

Setup: 8x8 matrix, 4 cache sets, each row maps to one set.

```
double a[8][8];  // Row i maps to cache set (i % 4)

// Want to read column 0: a[0][0], a[1][0], a[2][0], ...
```

Swizzling Example: Reading Column 0

Setup: 8x8 matrix, 4 cache sets, each row maps to one set.

```
double a[8][8];  // Row i maps to cache set (i % 4)

// Want to read column 0: a[0][0], a[1][0], a[2][0], ...
```

Without swizzling: All accesses go to column 0

Access	Location	Cache Set
a[0][0]	Row 0, Col 0	Set 0
a[1][0]	Row 1, Col 0	Set 1
a[2][0]	Row 2, Col 0	Set 2
a[3][0]	Row 3, Col 0	Set 3
a[4][0]	Row 4, Col 0	Set 0 (conflict!)
a[5][0]	Row 5, Col 0	Set 1 (conflict!)

Swizzling Example: The Transformation

With swizzling: $j' = j \oplus (i \wedge 3)$

To read "logical column 0", we access different physical columns:

Row i	Want col	$i \wedge 3$	$j' = 0 \oplus (i \wedge 3)$	Access
0	0	0	0	a[0][**0**]
1	0	1	1	a[1][**1**]
2	0	2	2	a[2][**2**]
3	0	3	3	a[3][**3**]
4	0	0	0	a[4][**0**]
5	0	1	1	a[5][**1**]

Swizzling Example: The Transformation

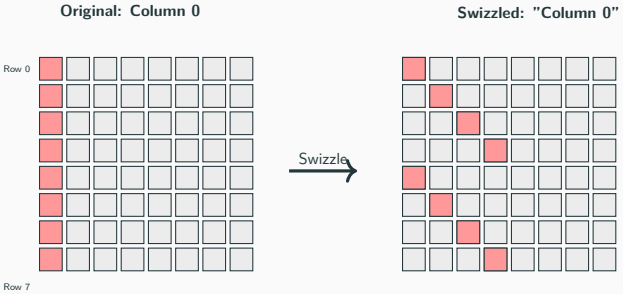
With swizzling: $j' = j \oplus (i \wedge 3)$

To read "logical column 0", we access different physical columns:

Row i	Want col	$i \wedge 3$	$j' = 0 \oplus (i \wedge 3)$	Access
0	0	0	0	a[0][**0**]
1	0	1	1	a[1][**1**]
2	0	2	2	a[2][**2**]
3	0	3	3	a[3][**3**]
4	0	0	0	a[4][**0**]
5	0	1	1	a[5][**1**]

Result: Accesses spread across columns, avoiding conflicts!

Visualizing Swizzled Layout



All same column
= conflict misses

Diagonal pattern
= no conflicts

Swizzling Code: Read Column

```c
// Reading "logical column c" from swizzled array
double read_column_swizzled(double a[N][N], int c) {
    double sum = 0;
    for (int i = 0; i < N; i++) {
        int j_physical = c ^ (i & 0x3);  // Swizzle formula
        sum += a[i][j_physical];
    }
    return sum;
}
```

Swizzling Code: Read Column

```c
// Reading "logical column c" from swizzled array
double read_column_swizzled(double a[N][N], int c) {
    double sum = 0;
    for (int i = 0; i < N; i++) {
        int j_physical = c ^ (i & 0x3);  // Swizzle formula
        sum += a[i][j_physical];
    }
    return sum;
}
```

The data must be stored in swizzled layout:

```c
// Store value at logical position (i, j)
void store_swizzled(double a[N][N], int i, int j, double val) {
    int j_physical = j ^ (i & 0x3);
    a[i][j_physical] = val;
}

// Read value from logical position (i, j)
double read_swizzled(double a[N][N], int i, int j) {
    int j_physical = j ^ (i & 0x3);
    return a[i][j_physical];
}
```

Complete Example: Swizzled Matrix Storage

```
$ cd examples && make swizzle
```

```cpp
// swizzle_demo.cpp
#include <iostream>
using namespace std;

const int N = 8;
double A[N][N];  // Physical storage

// Swizzle function
int swizzle(int i, int j) { return j ^ (i & 0x3); }

// Store in swizzled layout
void store(int i, int j, double val) { A[i][swizzle(i,j)] = val; }
```

Example Output

```
Logical column 0:
  logical[0][0] = 0 (physical col 0)
  logical[1][0] = 8 (physical col 1)
  logical[2][0] = 16 (physical col 2)
  logical[3][0] = 24 (physical col 3)
  logical[4][0] = 32 (physical col 0)
  logical[5][0] = 40 (physical col 1)
  logical[6][0] = 48 (physical col 2)
  logical[7][0] = 56 (physical col 3)
```

Example Output

```
Logical column 0:
  logical[0][0] = 0 (physical col 0)
  logical[1][0] = 8 (physical col 1)
  logical[2][0] = 16 (physical col 2)
  logical[3][0] = 24 (physical col 3)
  logical[4][0] = 32 (physical col 0)
  logical[5][0] = 40 (physical col 1)
  logical[6][0] = 48 (physical col 2)
  logical[7][0] = 56 (physical col 3)
```

Physical columns accessed: 0, 1, 2, 3, 0, 1, 2, 3

Instead of all mapping to column 0, accesses are distributed!

Swizzling for Matrix Transpose

Matrix transpose has conflict misses when writing columns:

```
// Naive: writes to B are column-wise (conflicts!)
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        B[j][i] = A[i][j];
```

Swizzling for Matrix Transpose

Matrix transpose has conflict misses when writing columns:

```
// Naive: writes to B are column-wise (conflicts!)
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        B[j][i] = A[i][j];
```

Solution: Store B in swizzled layout

```
// B stored swizzled: writes become distributed
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
        int j_swiz = i ^ (j & 0x3);  // Swizzle the write
        B[j][j_swiz] = A[i][j];
    }
```

Swizzling: The Key Insight

Swizzling trades logical simplicity for physical distribution

Swizzling: The Key Insight

Swizzling trades logical simplicity for physical distribution

Without swizzling:

- Logical column j = Physical column j
- Simple indexing
- Cache conflicts on column access

Swizzling: The Key Insight

Swizzling trades logical simplicity for physical distribution

Without swizzling:

- Logical column j = Physical column j
- Simple indexing
- Cache conflicts on column access

With swizzling:

- Logical column j = Physical column $j \oplus (i \wedge \text{mask})$
- Must swizzle on every access
- No cache conflicts!

Choosing the Swizzle Mask

The mask determines how many cache sets to distribute across:

Mask	Binary	Sets distributed
0x1	001	2 sets
0x3	011	4 sets
0x7	111	8 sets
0xF	1111	16 sets

Choosing the Swizzle Mask

The mask determines how many cache sets to distribute across:

Mask	Binary	Sets distributed
0x1	001	2 sets
0x3	011	4 sets
0x7	111	8 sets
0xF	1111	16 sets

Rule of thumb: Mask should cover enough bits to span cache associativity.

```
// For 8-way associative cache, use mask 0x7
int j_swizzled = j ^ (i & 0x7);
```

Swizzling is especially important in GPU programming

Swizzling is especially important in GPU programming:

- Shared memory has 32 banks
- Power-of-2 strides cause bank conflicts
- NVIDIA uses swizzling extensively in cuBLAS, cuDNN

Write Policies

What Happens on a Write?

Write-hit policies:

- **Write-through:** Write to both cache and memory immediately
- **Write-back:** Write only to cache; write to memory when evicted

What Happens on a Write?

Write-hit policies:

- **Write-through:** Write to both cache and memory immediately
- **Write-back:** Write only to cache; write to memory when evicted

Write-miss policies:

- **Write-allocate:** Load block into cache, then write
- **No-write-allocate:** Write directly to memory

What Happens on a Write?

Write-hit policies:

- **Write-through:** Write to both cache and memory immediately
- **Write-back:** Write only to cache; write to memory when evicted

Write-miss policies:

- **Write-allocate:** Load block into cache, then write
- **No-write-allocate:** Write directly to memory

Modern CPUs: Write-back + Write-allocate (most efficient for repeated writes)

Write-Back Implications

```
// This pattern benefits from write-back
for (int i = 0; i < n; i++)
    a[i] = a[i] * 2;  // Read and write same location
```

Write-Back Implications

```
// This pattern benefits from write-back
for (int i = 0; i < n; i++)
    a[i] = a[i] * 2;  // Read and write same location
```

With write-back:

1. First access: Load block into cache (miss)
2. Subsequent accesses: All in cache (hits)
3. Eviction: Write modified block back to memory

Traffic: Read once + Write once per block (not per element)

Summary

Key Takeaways

1. **Locality is crucial** for cache performance
   - Temporal: Reuse data while in cache
   - Spatial: Access data sequentially

Key Takeaways

1. **Locality is crucial** for cache performance
   - Temporal: Reuse data while in cache
   - Spatial: Access data sequentially

2. **Cache organization** (S sets, E ways, B bytes/block)
   - Direct mapped vs set-associative
   - Block size typically 64 bytes

Key Takeaways

1. **Locality is crucial** for cache performance
   - Temporal: Reuse data while in cache
   - Spatial: Access data sequentially
2. **Cache organization** (S sets, E ways, B bytes/block)
   - Direct mapped vs set-associative
   - Block size typically 64 bytes
3. **Three C's of misses:** Compulsory, Capacity, Conflict

Key Takeaways

1. **Locality is crucial** for cache performance
   - Temporal: Reuse data while in cache
   - Spatial: Access data sequentially
2. **Cache organization** (S sets, E ways, B bytes/block)
   - Direct mapped vs set-associative
   - Block size typically 64 bytes
3. **Three C's of misses:** Compulsory, Capacity, Conflict
4. **Blocking** improves locality for matrix operations
   - Choose block size based on cache size
   - Increases operational intensity

Try It Yourself

```
# Run all examples
$ cd examples && make run

# Build with -O3 for comparison
$ cd examples && make fast && make run

# Use perf to measure cache misses
$ cd examples && make perf
```

Practical Guidelines

Do:

- Access arrays sequentially (stride 1)
- Use blocking for matrix operations
- Keep working set small enough to fit in cache
- Measure with `perf stat` to verify

Practical Guidelines

Do:

- Access arrays sequentially (stride 1)
- Use blocking for matrix operations
- Keep working set small enough to fit in cache
- Measure with `perf stat` to verify

Avoid:

- Column-major access in row-major languages (C/C++)
- Power-of-2 array dimensions when accessing columns
- Random access patterns when sequential is possible