# 15-213 Recitation 6: 10/14/02

## Outline

- Optimization
  - Amdahl's Law
- Cache
  - Performance Metrics
  - Access Patterns

## Reminder

- L4 due: Thursday 10/24, 11:59pm


**Annie Luo**

**e-mail:**

luluo@cs.cmu.edu

**Office Hours:**

Thursday 6:00 – 7:00

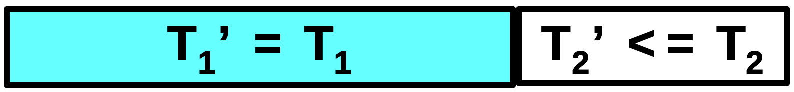
Wean 8402

# Amdahl's Law

**Old program**


**Old time: $T = T_1 + T_2$**

**New program (improved)**




**New time: $T' = T_1' + T_2'$**

**$T_1$ = time that can NOT be improved.**

**$T_2$ = time that can be improved.**

**$T_2'$ = time after the improvement.**

## Speedup = T / T'

- Amdahl's Law describes a general principle for improving any process, not only for speeding up computer systems.

# Amdahl's Law: Example

- Planning a trip *PGH –>NY –> Paris –> Metz*
- Suppose both *PGH –>NY* and *Paris –> Metz* take 4 hours
- For *NY –> Paris* take 8.5 hours by a Boeing 747
- Total travel time:

**What if we choose faster methods?**

| | NY->Paris | Total time | Speedup over 747 |
|---|---|---|---|
| 747 | 8.5 hours | 16.5 hours | 1 |
| SST | 3.75 hours | 11.75 hours | 1.4 |
| rocket | 0.25 hours | 8.25 hours | 2.0 |
| rip | 0.0 hours | 8.0 hours | 2.1 |

- It's hard to gain significant improvement.
- Larger speedup comes from improving larger fraction of the whole system.

# Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses/references)
- Hit Time
  - Time to deliver a line in the cache to the processor (including determining time)
- Miss Penalty
  - Additional time required because of a miss

# Locality

- Temporal locality:
  - a memory location that is referenced once is likely to be *reference again multiple times* in the near future
- Spatial locality:
  - if a memory location is referenced once, then the program is likely to *reference a nearby memory location* in the near future

# Practice Problem 6.4

- Permute the loops so that it scans the 3-dimensional array *a* with a stride-1 reference pattern:

```
int summary3d(int a[N][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; k < N; j++ ) {
            for (k = 0; k < N; k++ ) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

# Array Organization in Memory

a[0][0][0], a[0][0][1], … …, a[0][0][N-1],

a[0][1][0], a[0][1][1], … …, a[0][1][N-1],

a[0][2][0], a[0][2][1], … …, a[0][2][N],

… …

a[1][0][0], a[1][0][1], … …, a[1][0][N],

… …

a[N-1][N-1][0],a[N-1][N-1][1],……,a[N-1][N-1][N-1]

# Solution

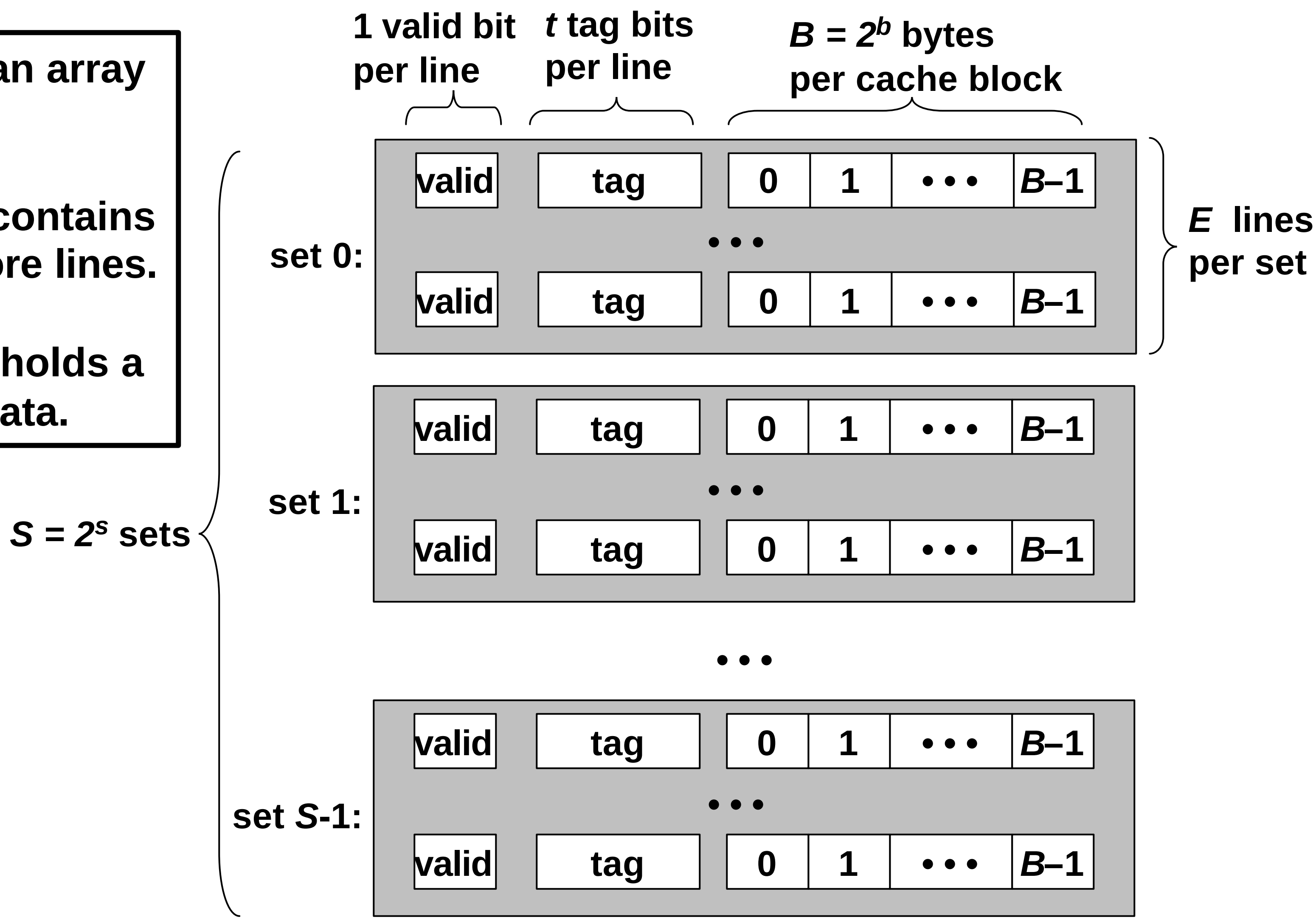
```
int summary3d(int a[N][N][N])
{
    int i, j, k, sum = 0;
    for (k = 0; k < N; k++) {
        for (i = 0; i < N; i++ ) {
            for (j = 0; j < N; j++ ) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

# Cache Organization (review)

Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.



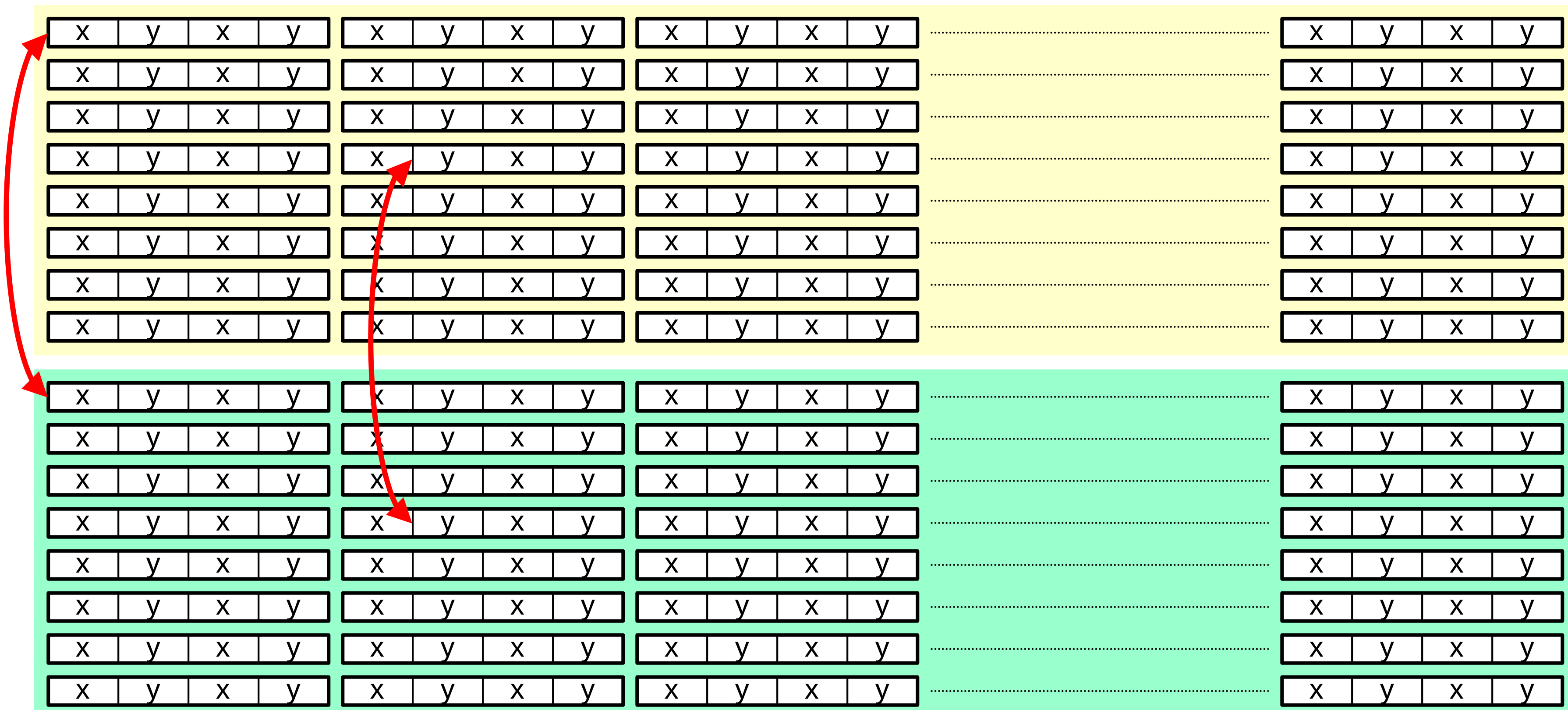
# Cache Access Patterns

- Now it's your turn to spend 15 minutes working on Practice Problems 6.15-6.17 ☺
- Handout is a photocopy from the text book

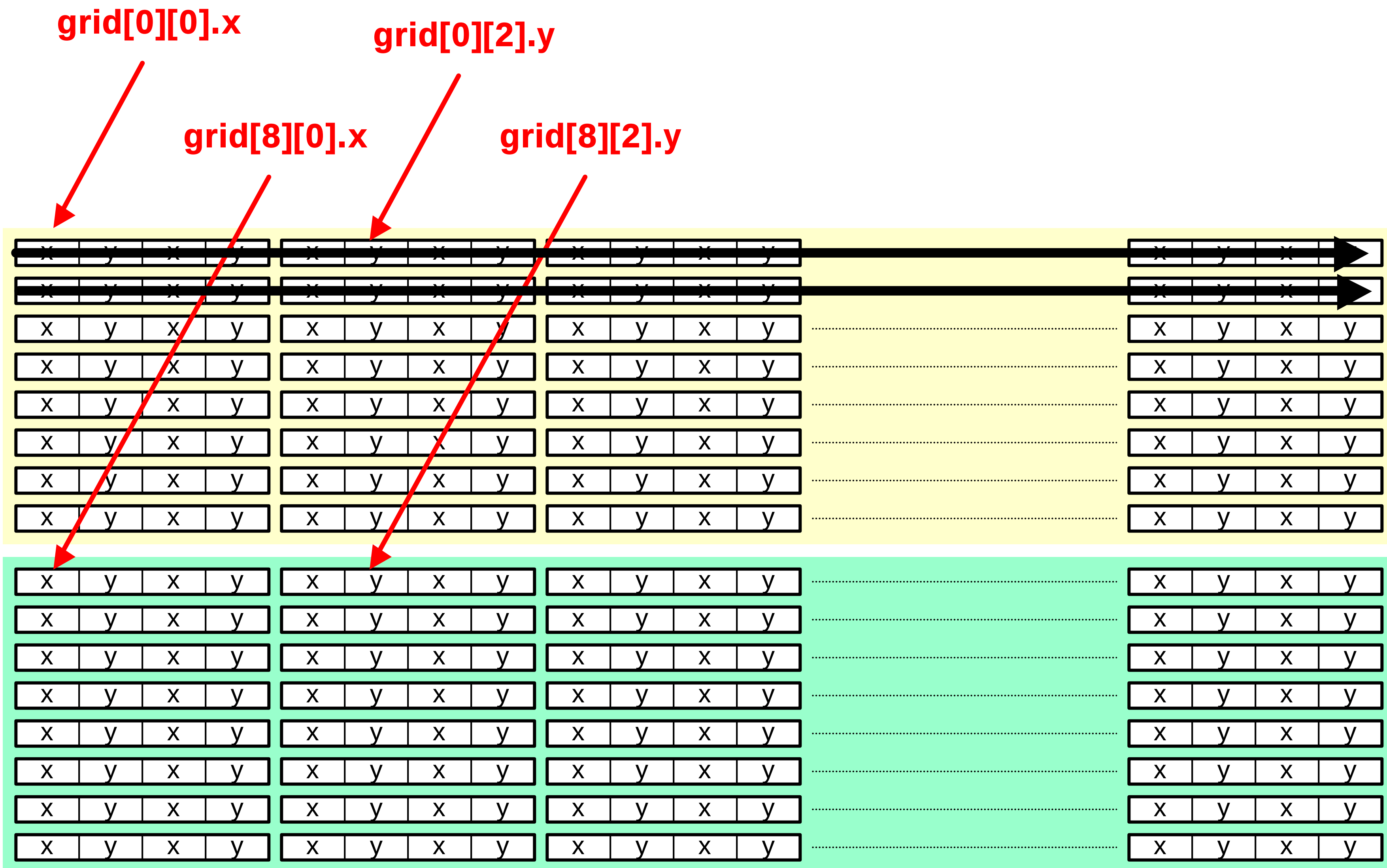
- Note that:
  - The size of struct **`algae_position`** is 8 bytes
  - Each cache block (16 bytes) holds two algae_position structs
  - The 16×16 array requires 2048 bytes of memory
  - Twice the size of the 1024 byte cache

# Practice Problem 6.15-17

- Each row: 16 struct items, 8 cache blocks, 128 bytes
- Each column: 16 struct items
- Yellow area: 1024 bytes, green area: 1024 bytes

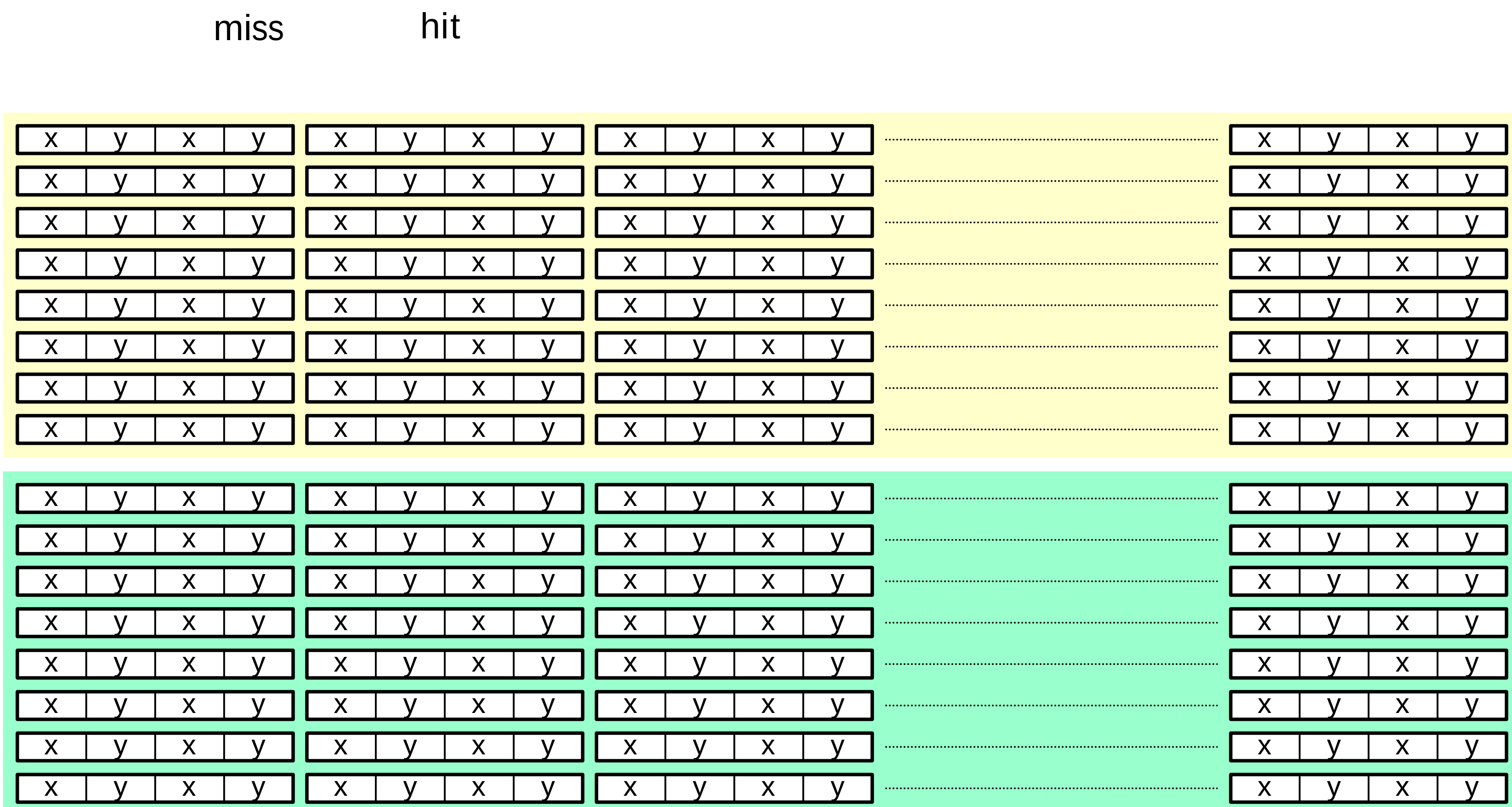


# 6.15 – Row Major Access Pattern



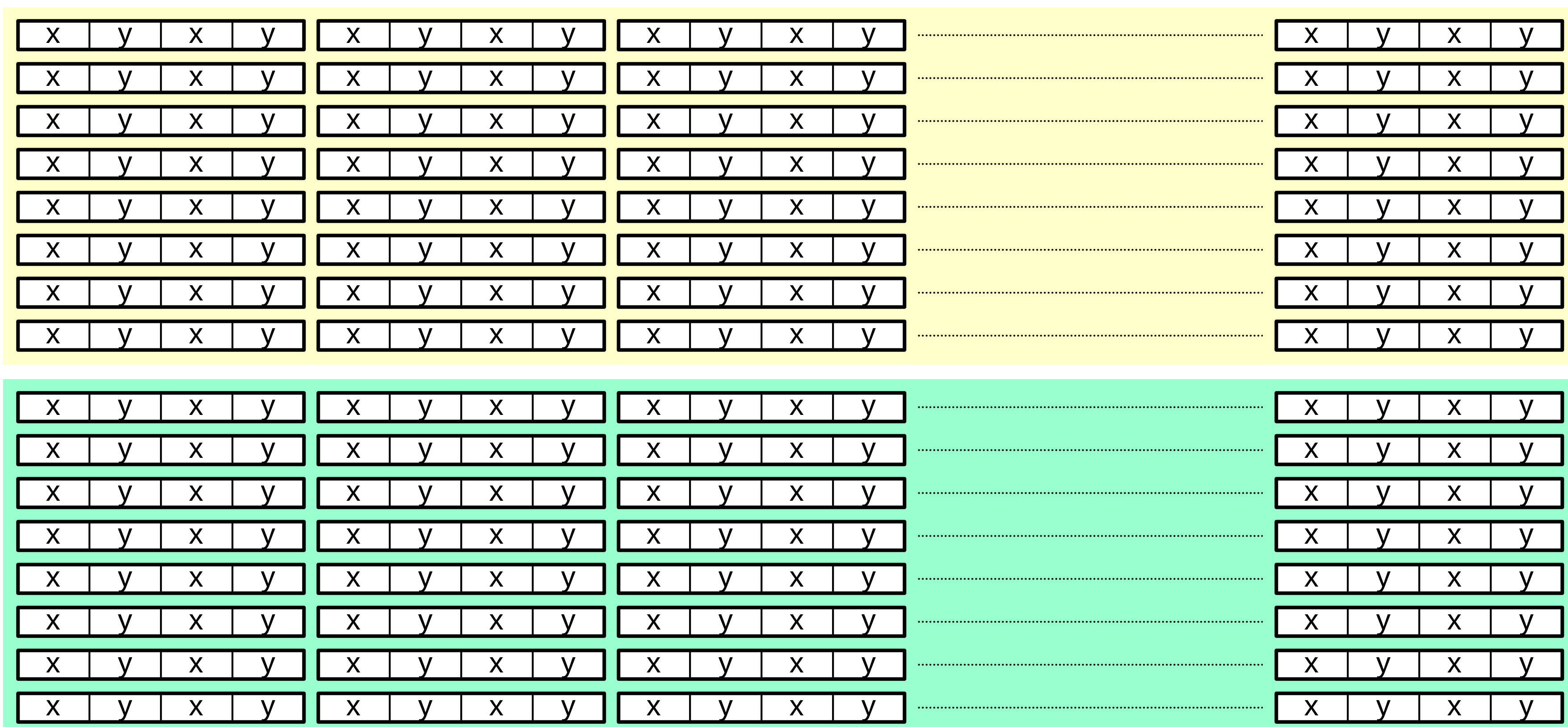
# 6.15 – Stride of two words

- First loop, accessing all x's
- When a cache miss happens, load a block from memory



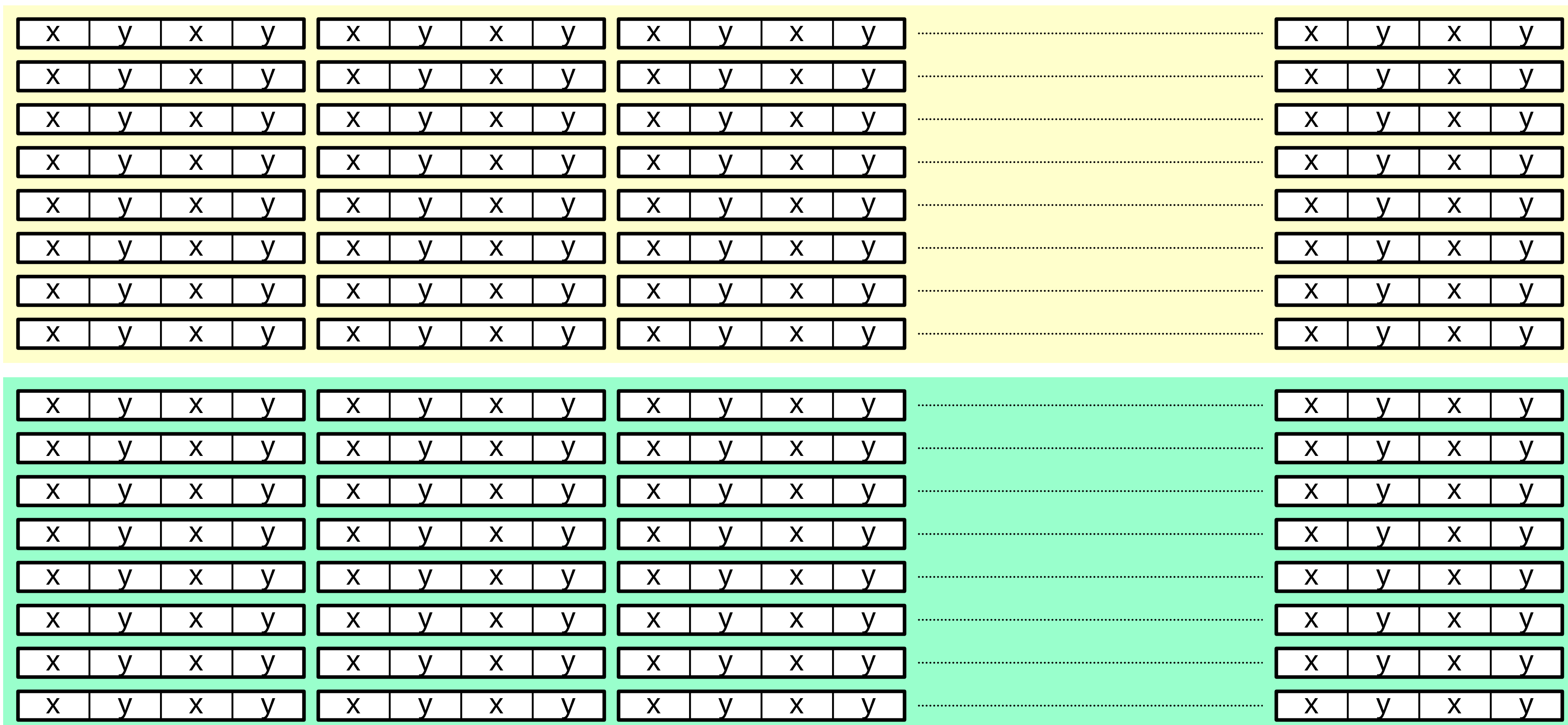
# 6.15 – Stride of two words

- First loop, accessing all x's
- When a cache miss happens, load a block from memory



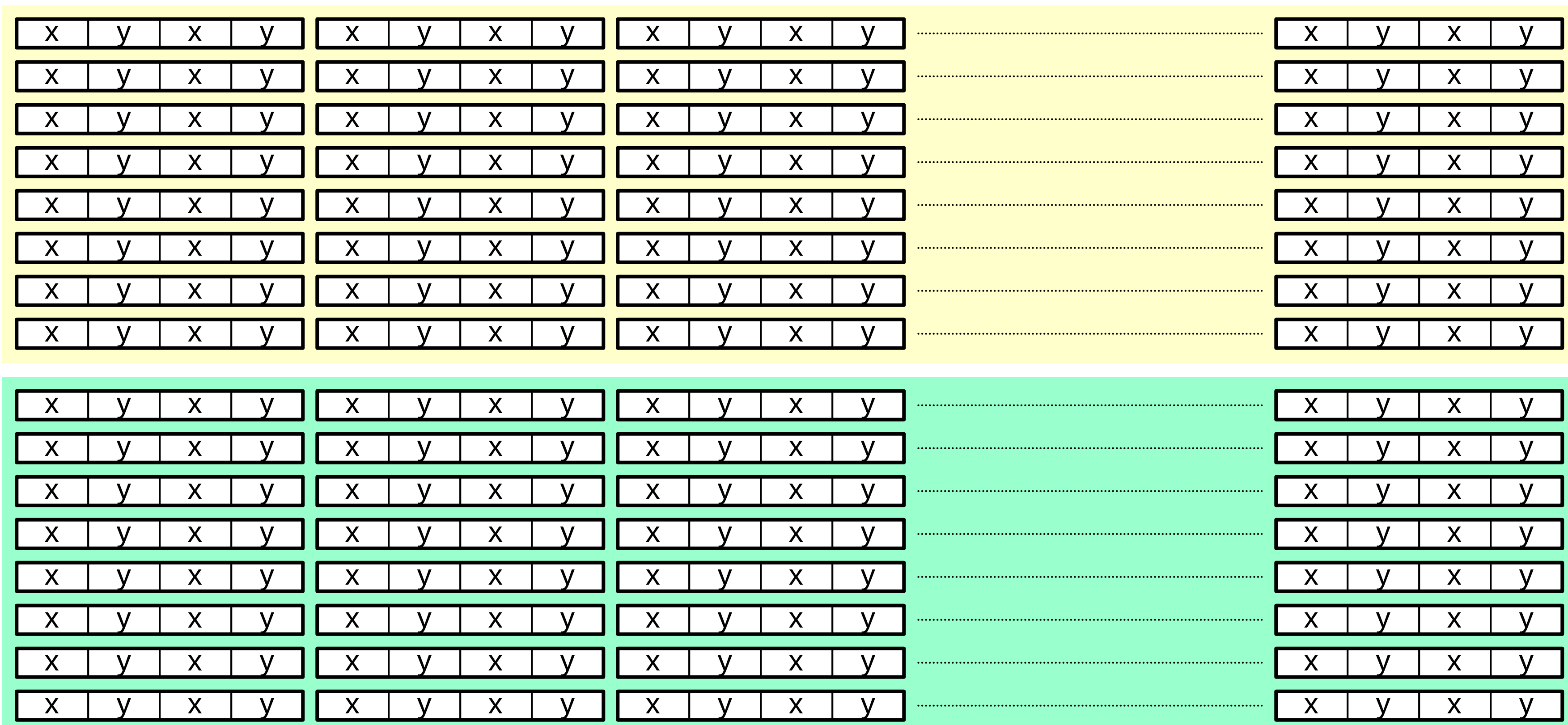
# 6.15 – Stride of two words

- Second loop, accessing all y’s
- Same missing pattern, the green area flushes blocks from the yellow area



# 6.15 – Stride of two words

- Second loop, accessing all y's
- Same missing pattern, the green area flushes blocks from the yellow area

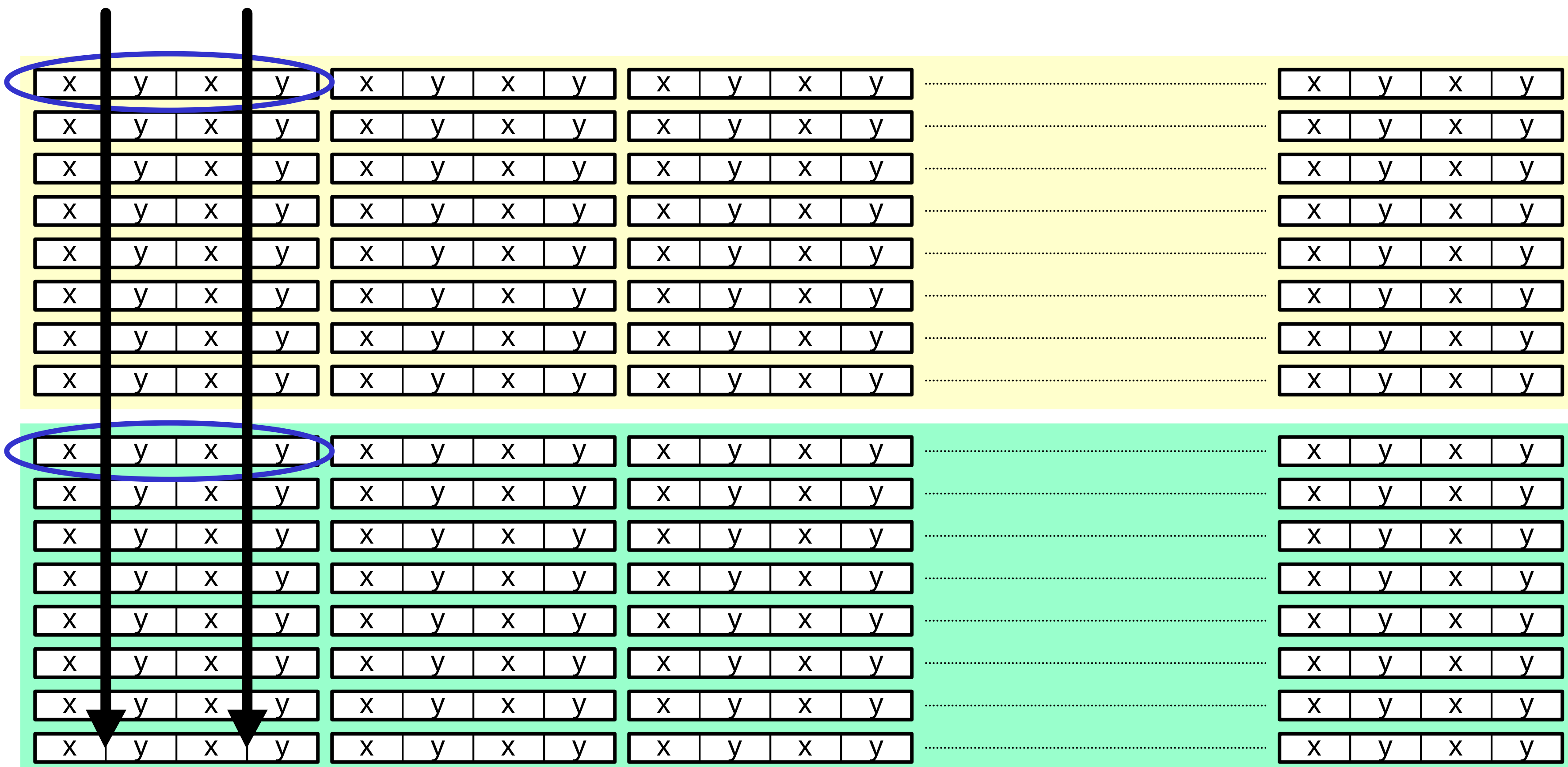


# Answer to Problem 6.15

- A: 512
  - 16x16 = 256 array elements in total
  - twice for each element
- B: 256
  - every other array element experiences a miss
- C: 50%

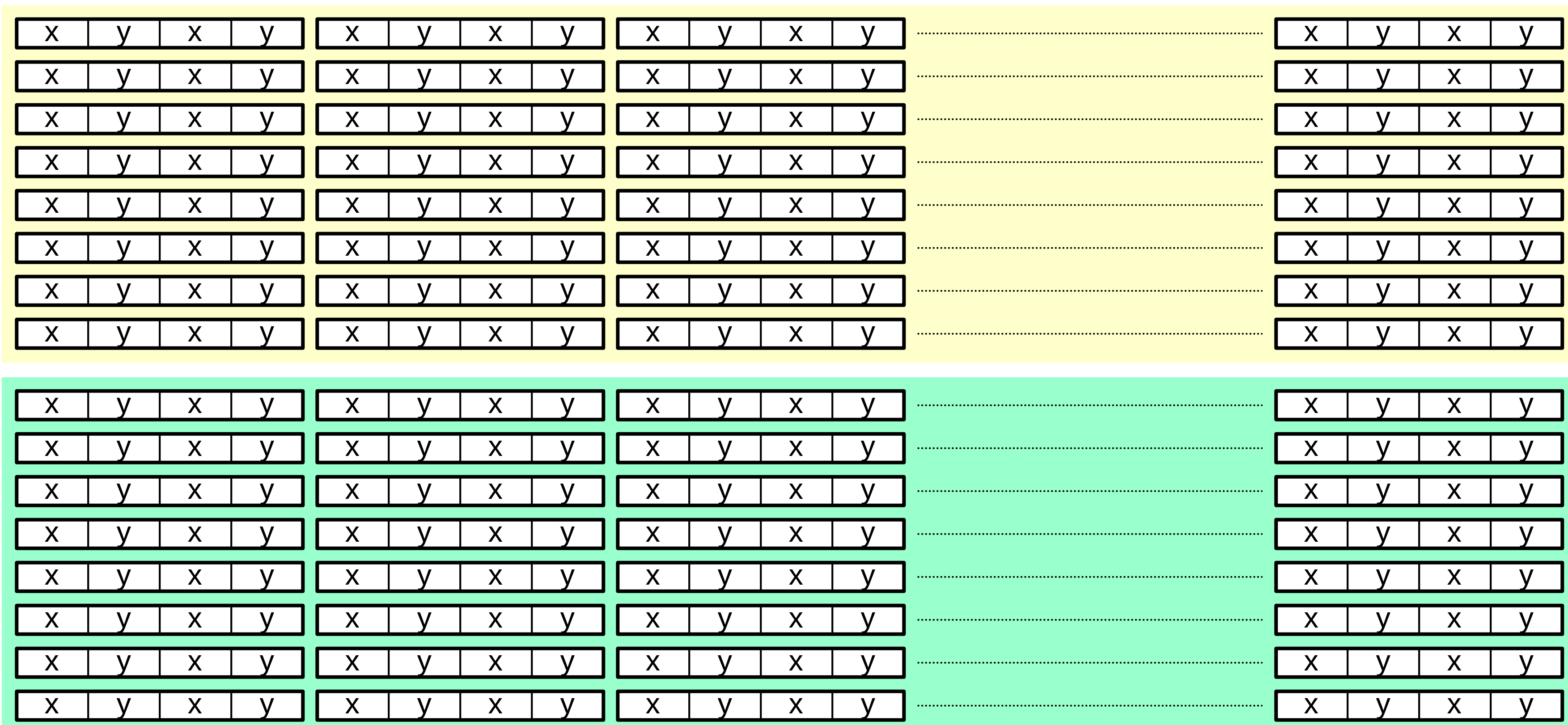
# 6.16 – Column Major Access Pattern

- New access removes first cache line contents before it is used



# 6.16 – Column Major Access Pattern

- New access removes first cache line contents before it is used

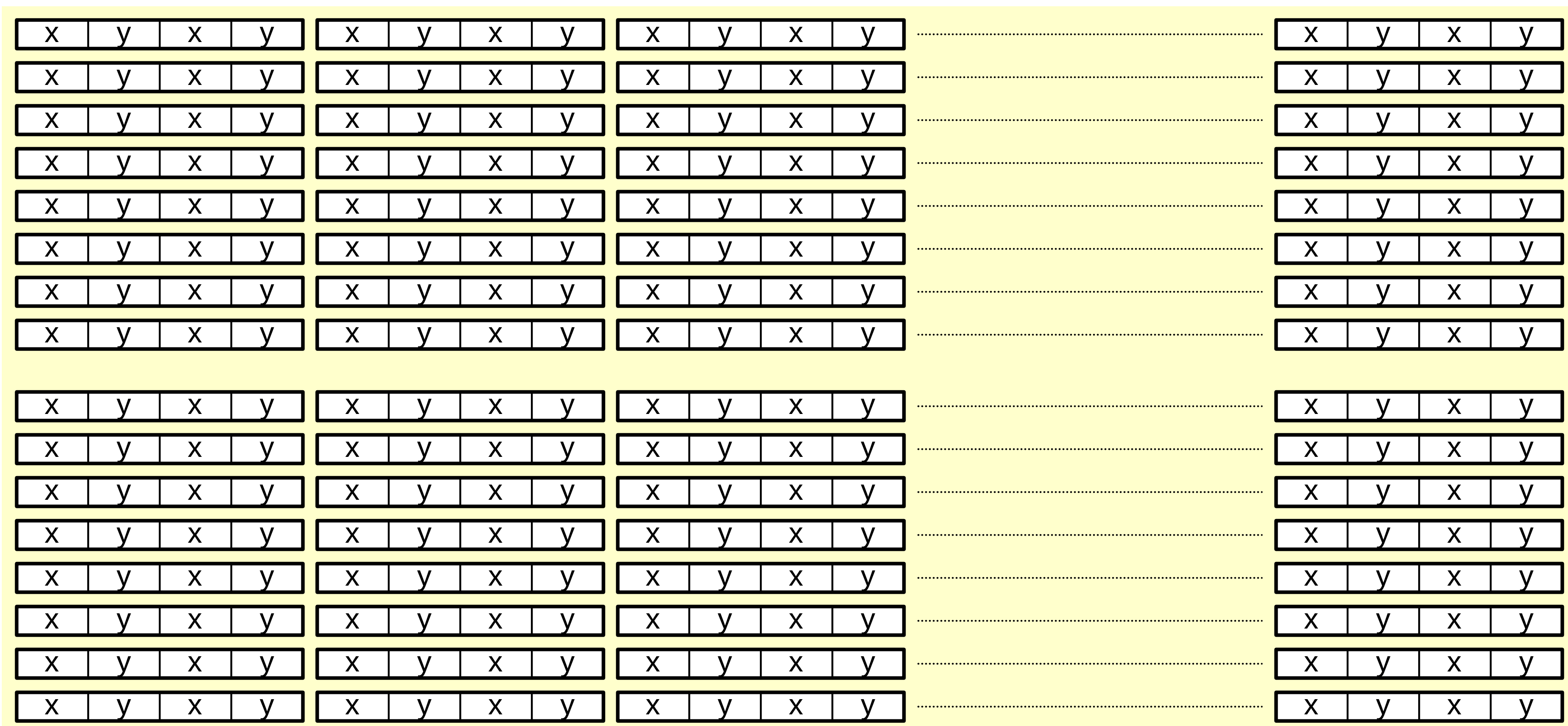


# Answer to Problem 6.16

- A: 512
- B: 256
- C: 50%

# 6.16 – Column Major Access Pattern

- What if the cache was 2048 bytes?
- No misses on second access to each block, since the entire array fits in the cache

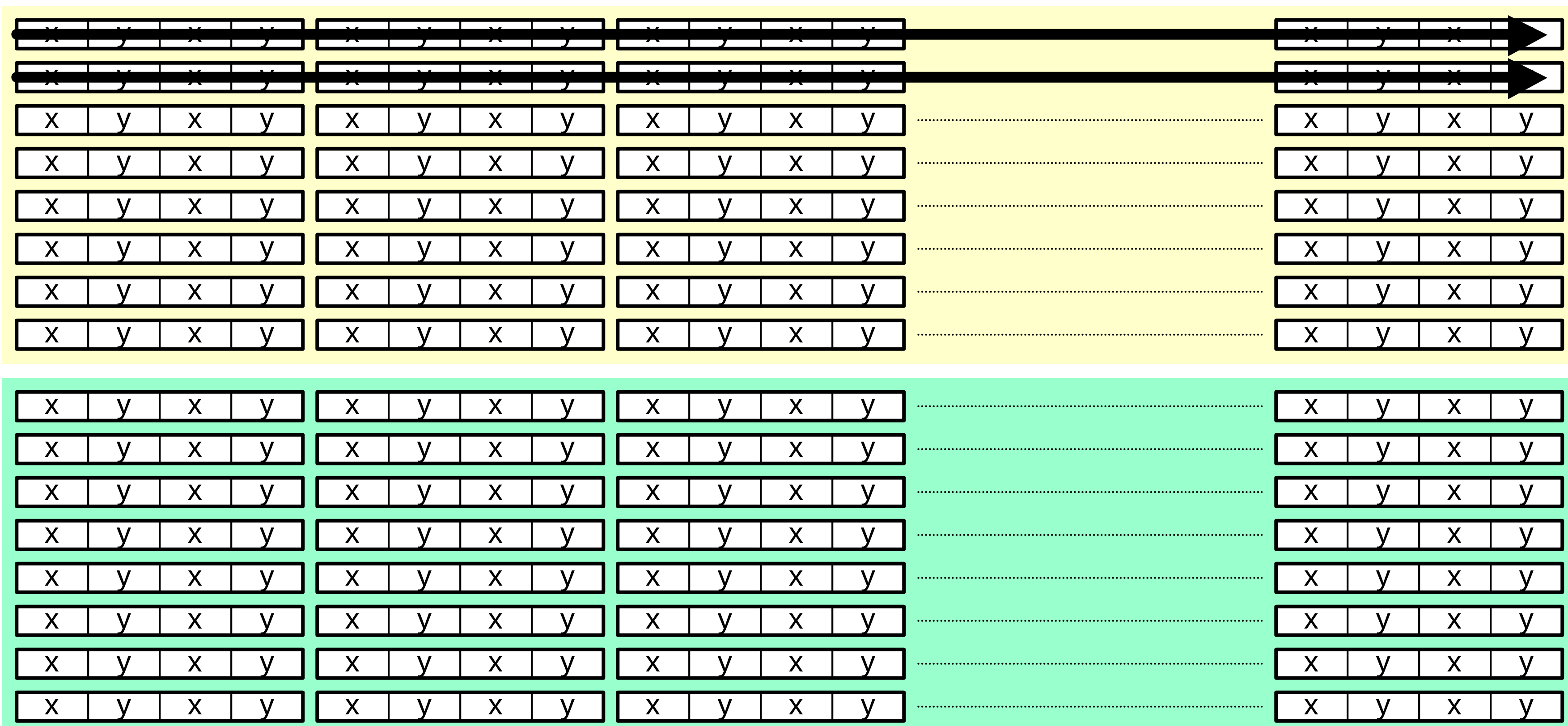


# Answer to Problem 6.16

- A: 512
- B: 256
- C: 50%
- D: 25%

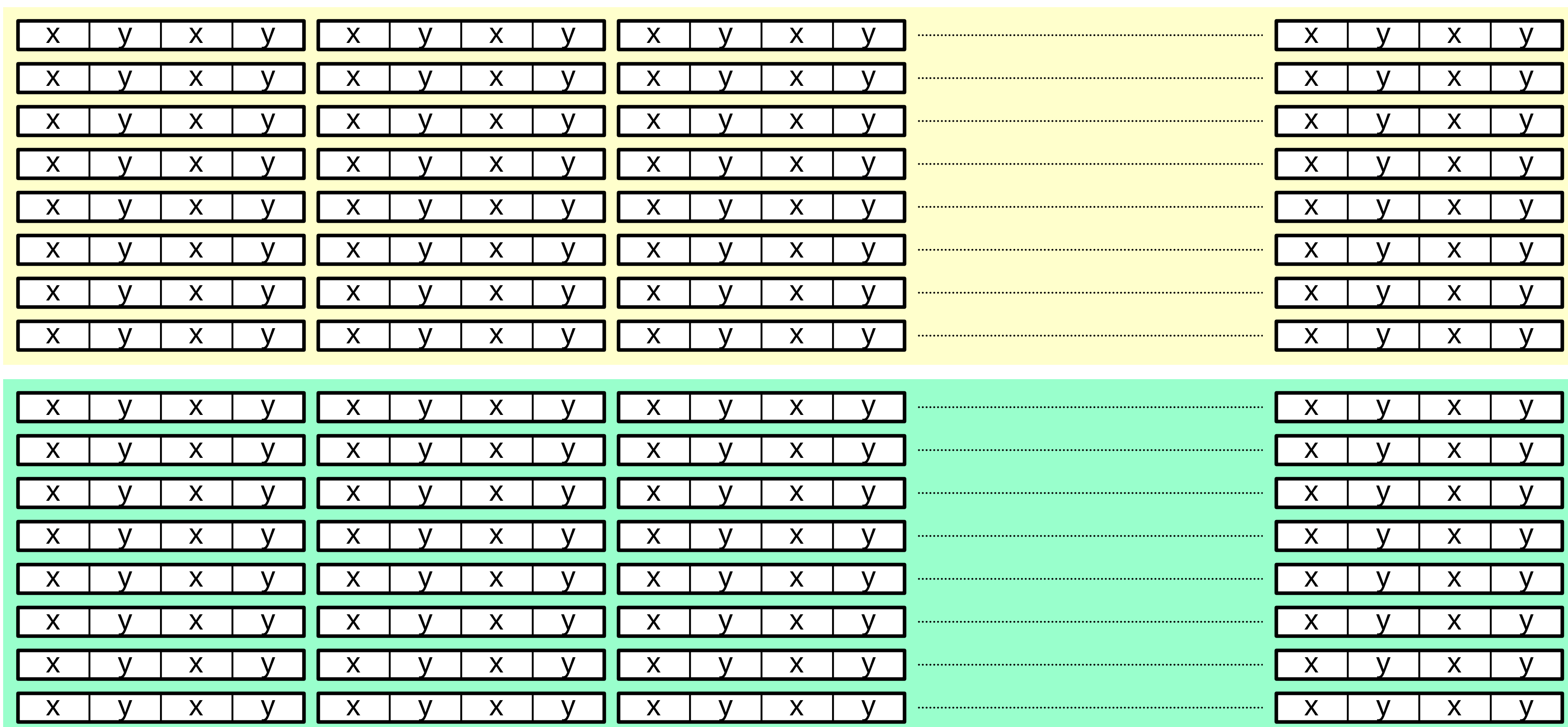
# 6.17 – Stride of One Word

- Access both x and y in row major order



# 6.17 – Stride of One Word

- Access both x and y in row major order



# Answer to Problem 6.17

- A: 512
- B: 128
  - All are compulsory misses
- C: 25%
- D: 25%
  - Cache size doesn’t matter since all misses are *must*
  - The block size does matter though