Recitation 6: Cache Access Patterns

Andrew Faulring

15213 Section A

14 October 2002

Andrew Faulring

- faulring@cs.cmu.edu
- Office hours:
  - NSH 2504 (lab) / 2507 (conference room)
  - Wednesday 5-6

- Lab 4
  - due Thursday, 24 Oct @ 11:59pm

Today's Plan

- Optimization
  - Amdahl's law
- Cache Access Patterns
  - Practice problems 6.4, 6.15–17
- Lab 4
  - Horner's Rule, including naïve code

Amdahl's law

Old program (unenhanced)

Old time: $T = T_1 + T_2$

New program (enhanced)

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

T_1 = time that can NOT be enhanced.

T_2 = time that can be enhanced.

T_2' = time after the enhancement.

Speedup: $S_{overall} = T \;/\; T'$

Key idea: Amdahl's law quantifies the general notion of diminishing returns. It applies to any activity, not just computer programs.

Example: Amdahl's law

- You plan to visit a friend in Normandy France and must decide whether it is worth it to take the Concorde SST ($3,100) or a 747 ($1,021) from NY to Paris, assuming it will take 4 hours Pgh to NY and 4 hours Paris to Normandy.

	time NY->Paris	total trip time	speedup over 747
747	8.5 hours	16.5 hours	1
SST	3.75 hours	11.75 hours	1.4

- Taking the SST (which is 2.2 times faster) speeds up the overall trip by only a factor of 1.4!

Amdahl's law (cont)

- Trip example: Suppose that for the New York to Paris leg, we now consider the possibility of taking a rocket ship (15 minutes) or a handy rip in the fabric of space-time (0 minutes):

	time NY->Paris	total trip 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 hours	2.1

Moral: It is hard to speed up a program.
Moral++ : It is easy to make premature optimizations.

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

```
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;
}
```

Answer

```c
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 Access Patterns

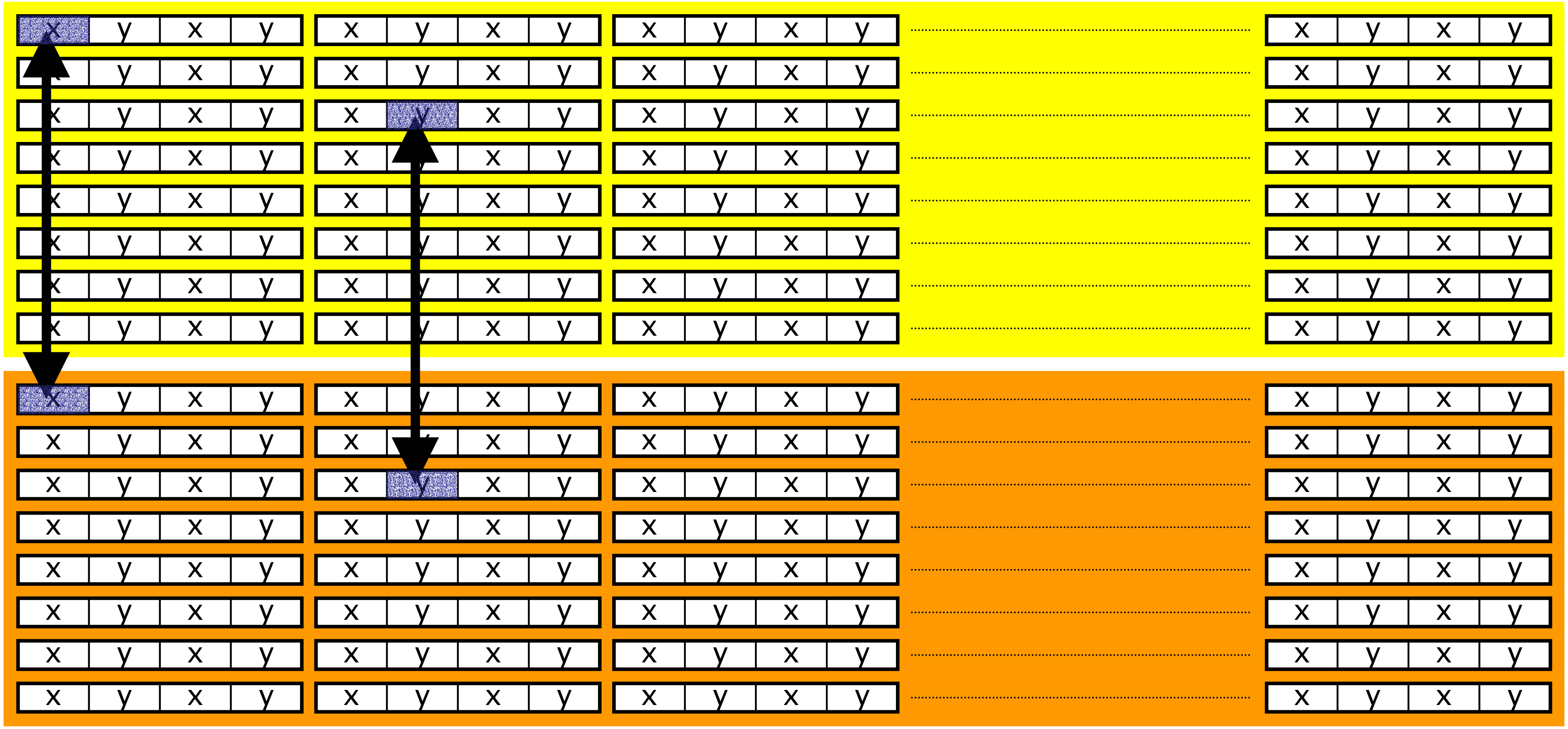
- Spend the next fifteen minutes working on Practice Problems 6.15–17
- Handout is a photocopy from the text

Practice Problem 6.15-17

- **`sizeof(algae_position) = 8`**
- Each block (16 bytes) holds two **`algae_position`** structures
- The 16×16 array requires 2048 bytes of memory
  - Twice the size of the 1024 byte cache

Practice Problem 6.15-17

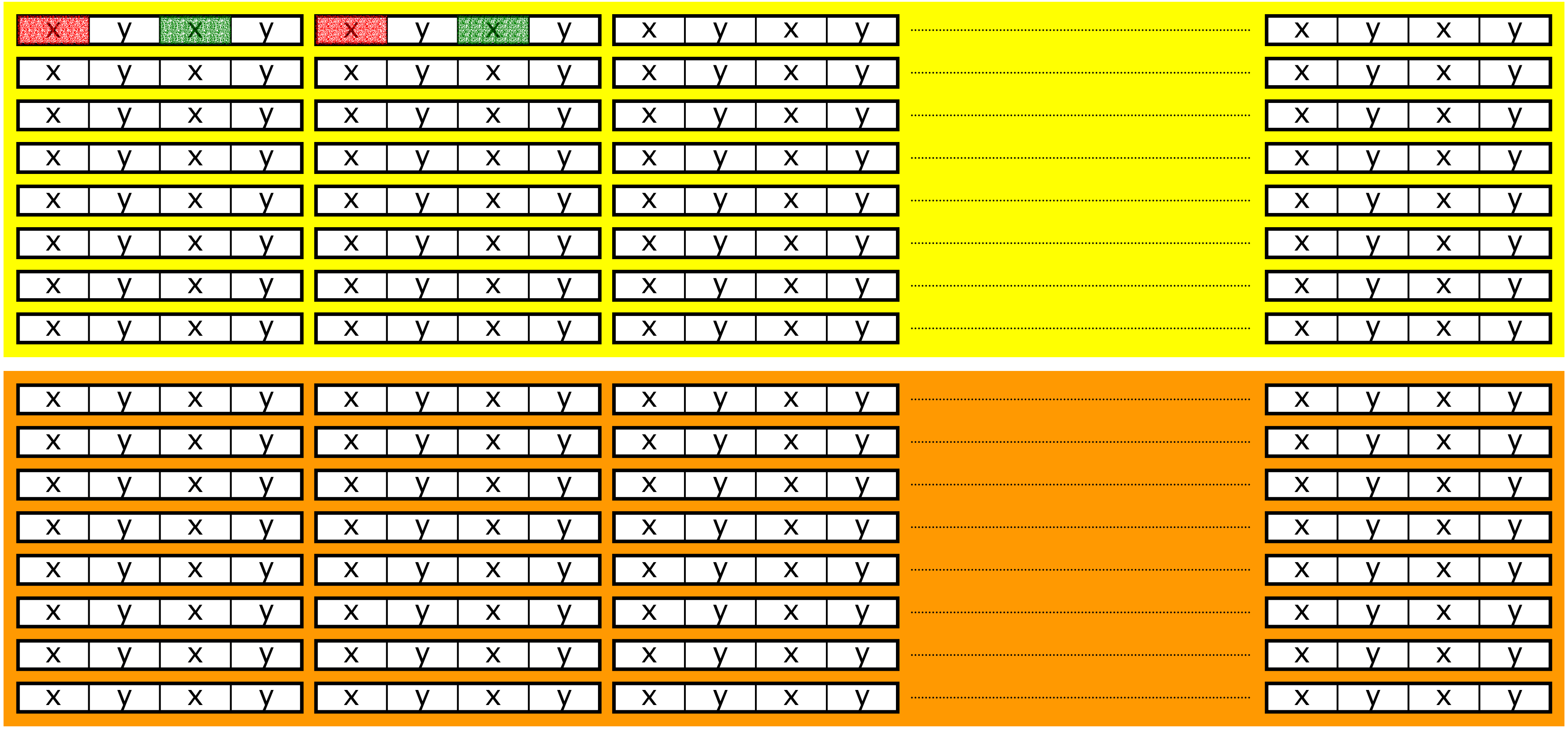
- Rows: 16 items (8 blocks, 128 bytes)
- Columns: 16 items
- Yellow block: 1k; Orange block 1k



6.15: Row major access pattern

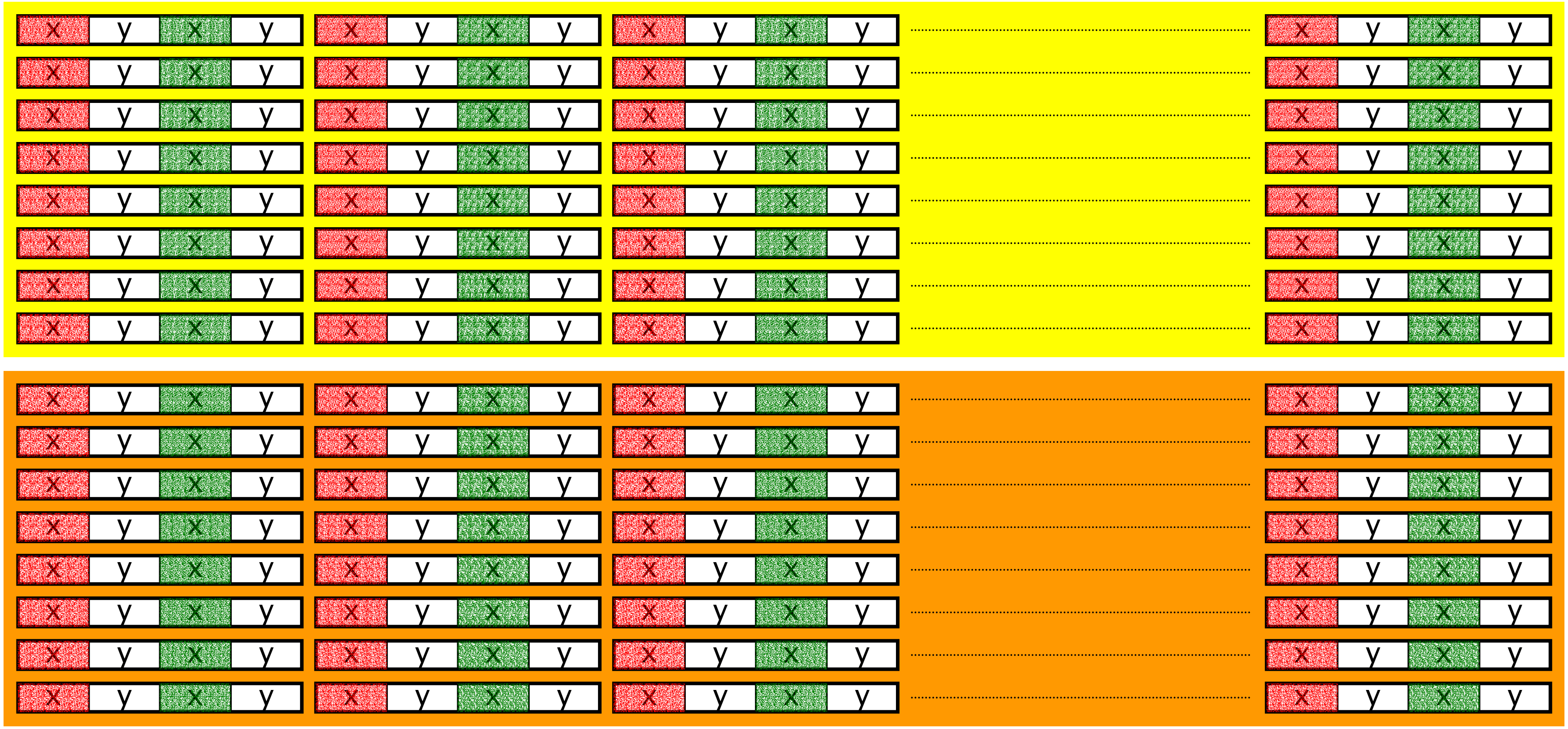
6.15: Stride of 2 words

- First loop, accessing just x’s



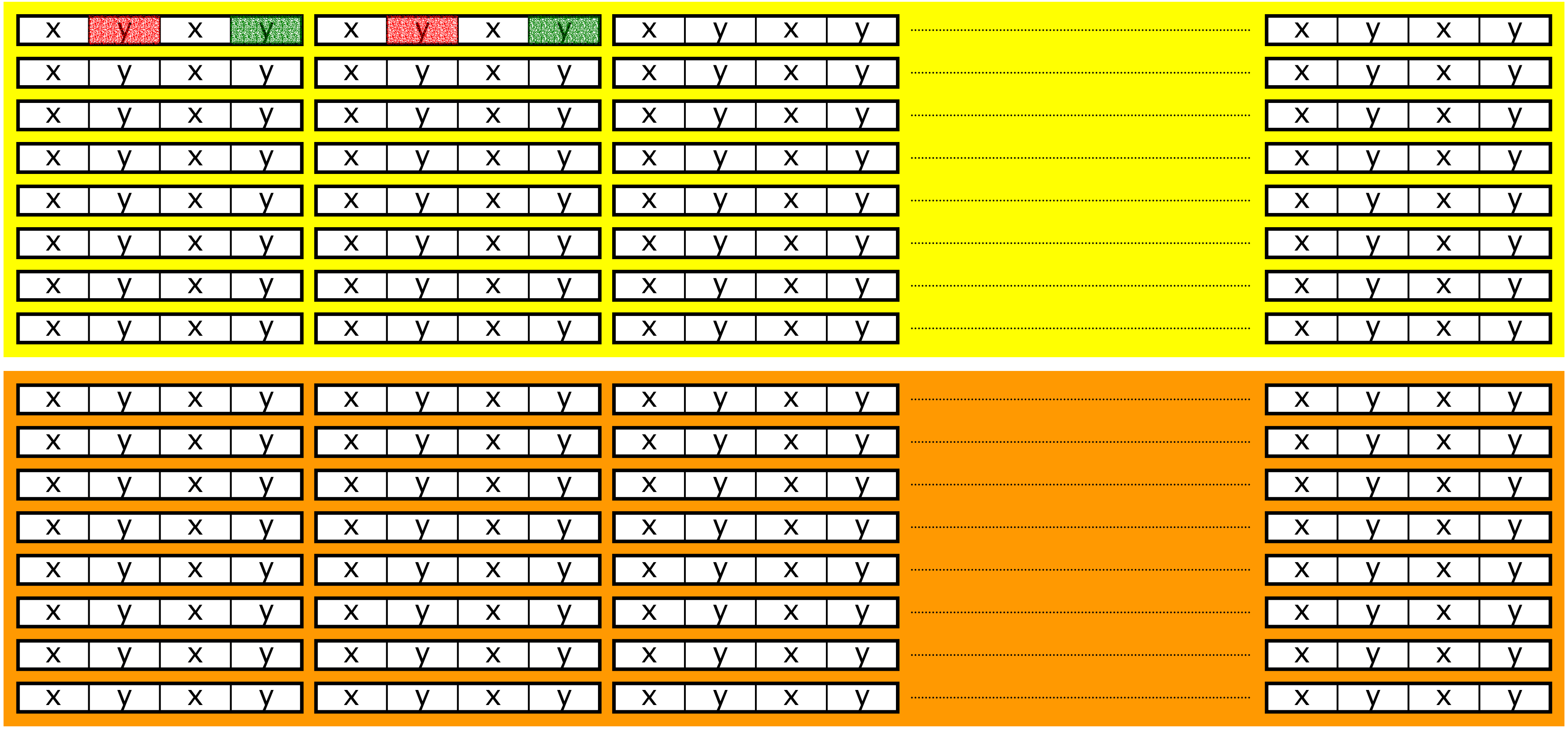
6.15: Stride of 2 words

- First loop, accessing just x's



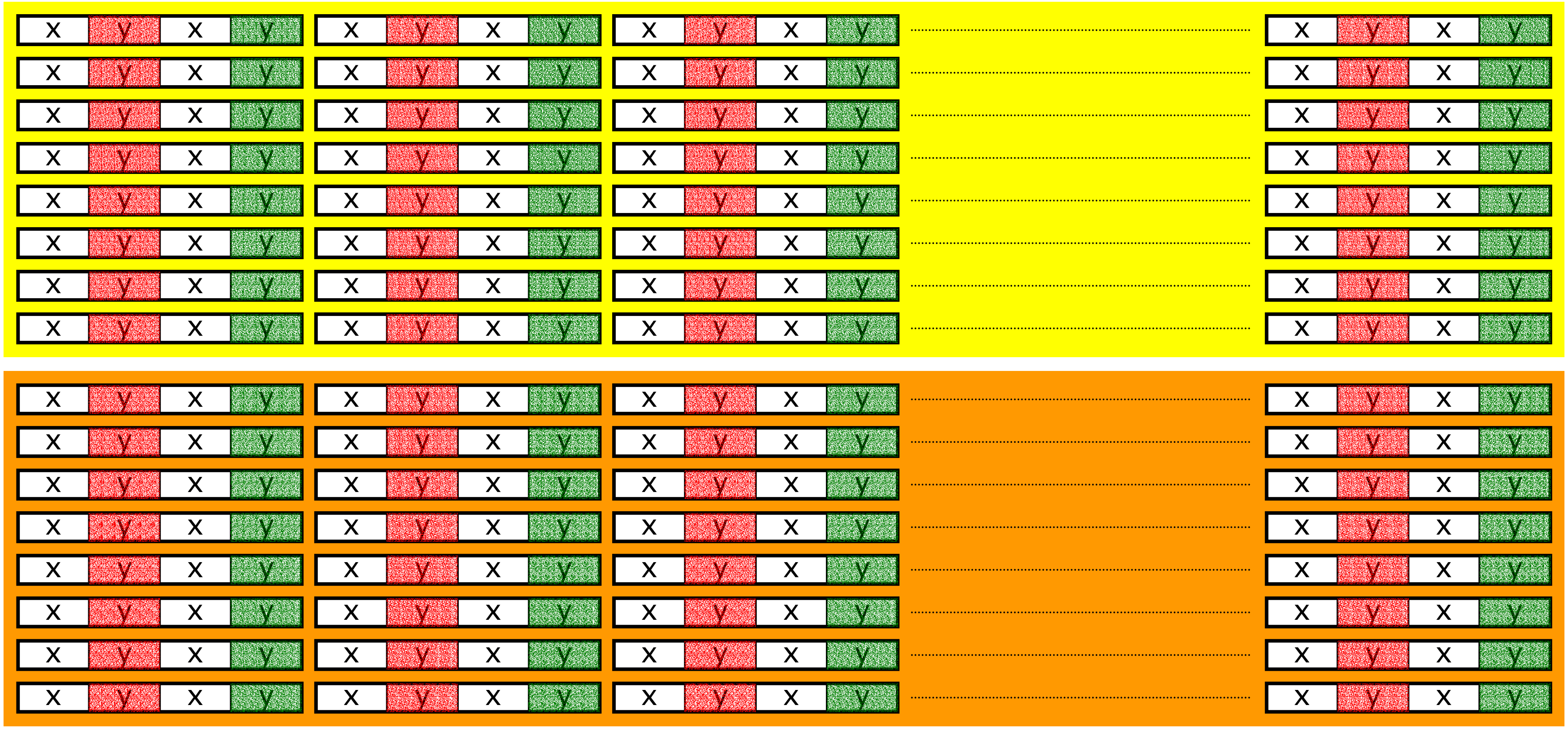
6.15: Stride of 2 words

- Second loop, accessing just the y's
- Same miss pattern because accessing the orange area flushed blocks from the yellow area



6.15: Stride of 2 words

- Second loop, accessing just the y's
- Same miss pattern because accessing the orange area flushed blocks from the yellow area

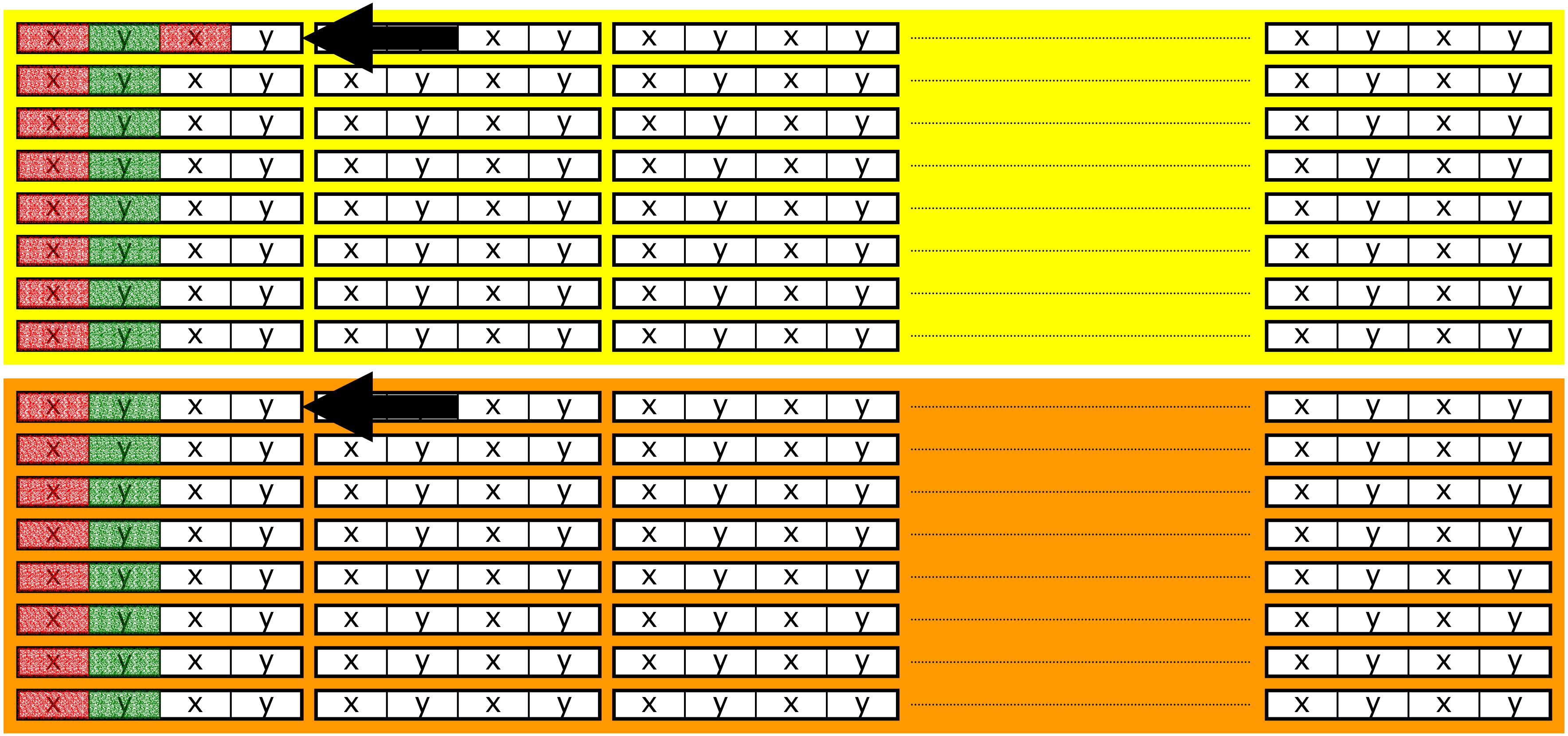


Answers to 6.15

- A: 512
  - 2 for each of 256 array elements
- B: 256
  - Every other array element experiences a miss
- C: 50%

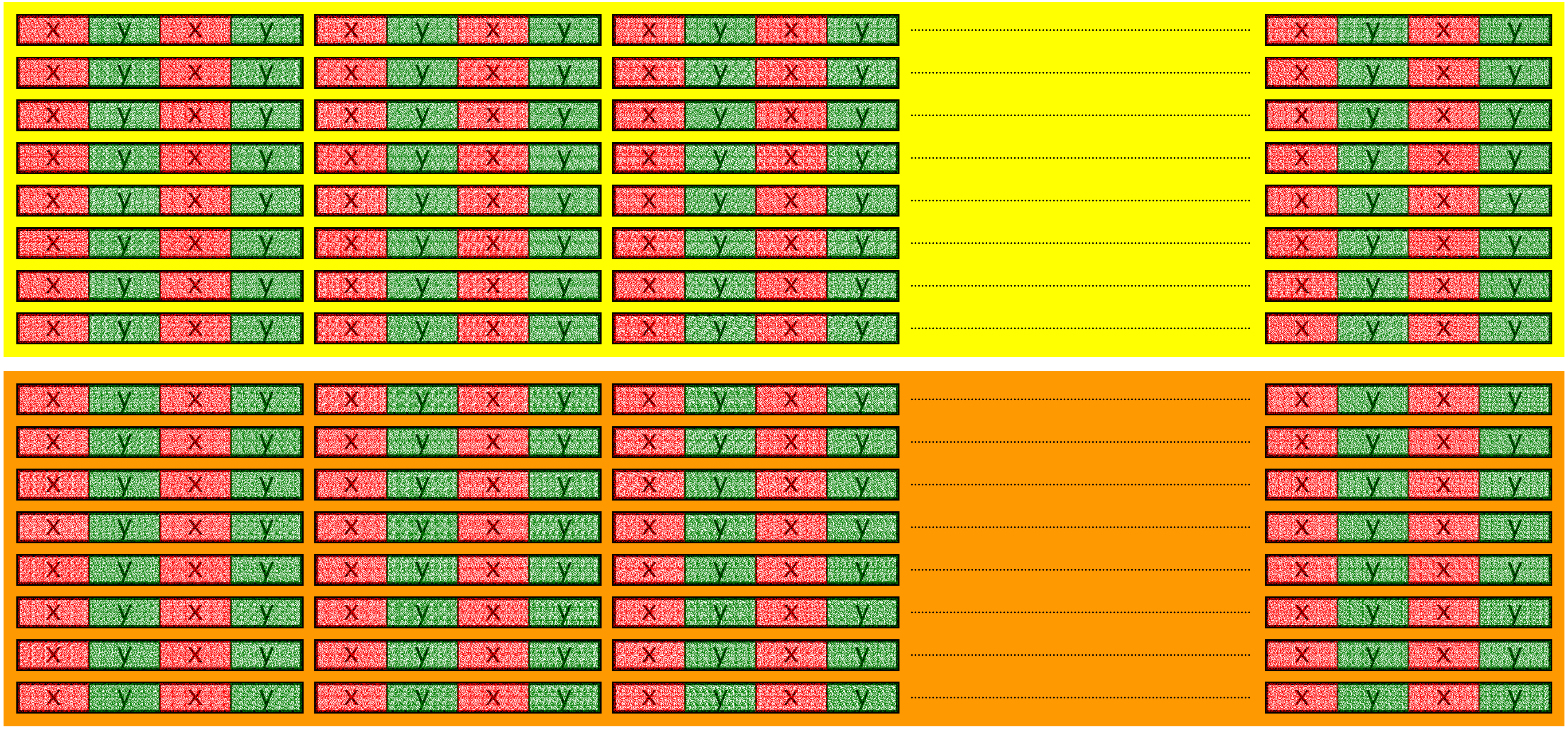
Column major access pattern

New access removes first cache line contents before its were used



Column major access pattern

New access removes first cache line contents before its were used

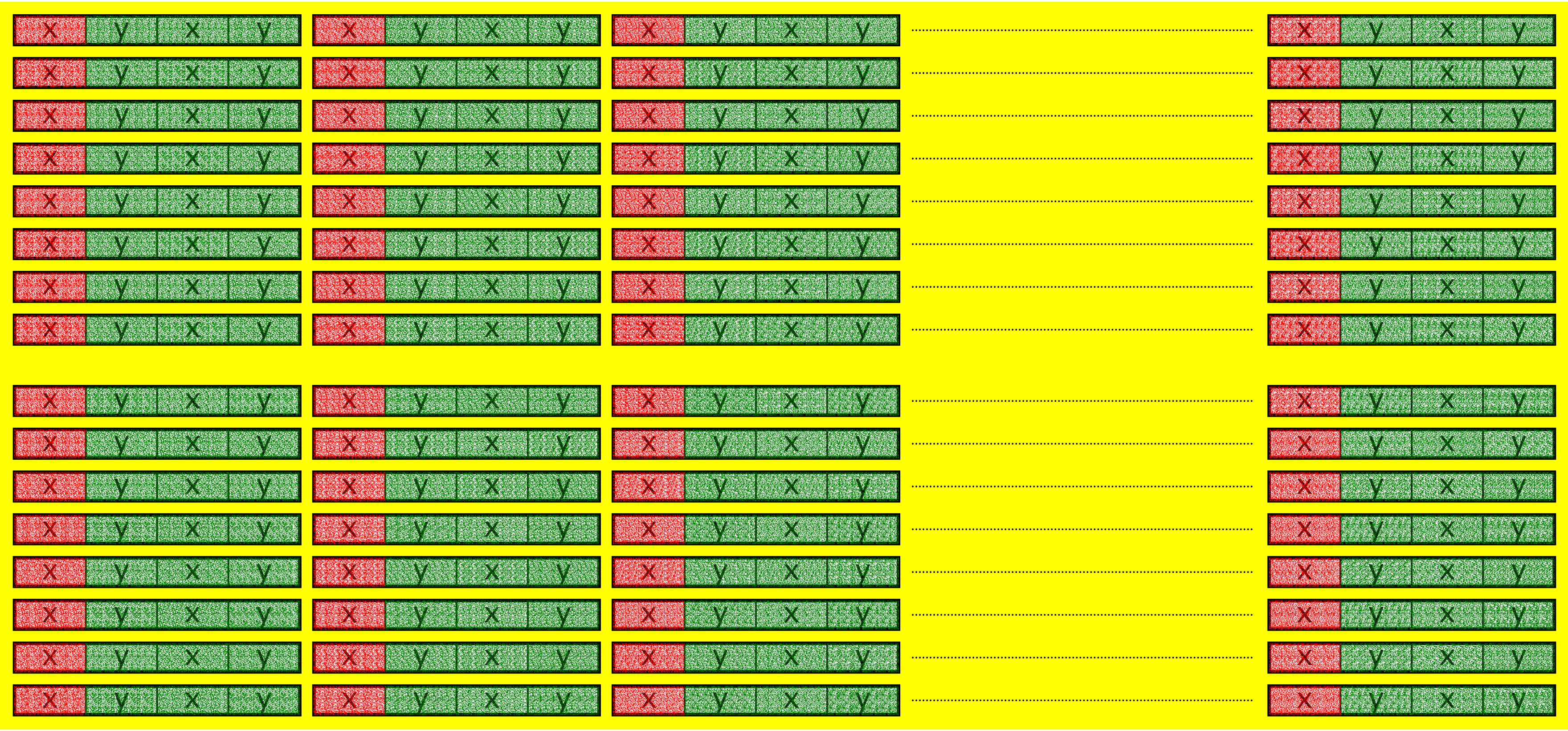


Answers to 6.16

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

Column major access pattern

No misses on second access to each block, because the entire array fits in the cache.

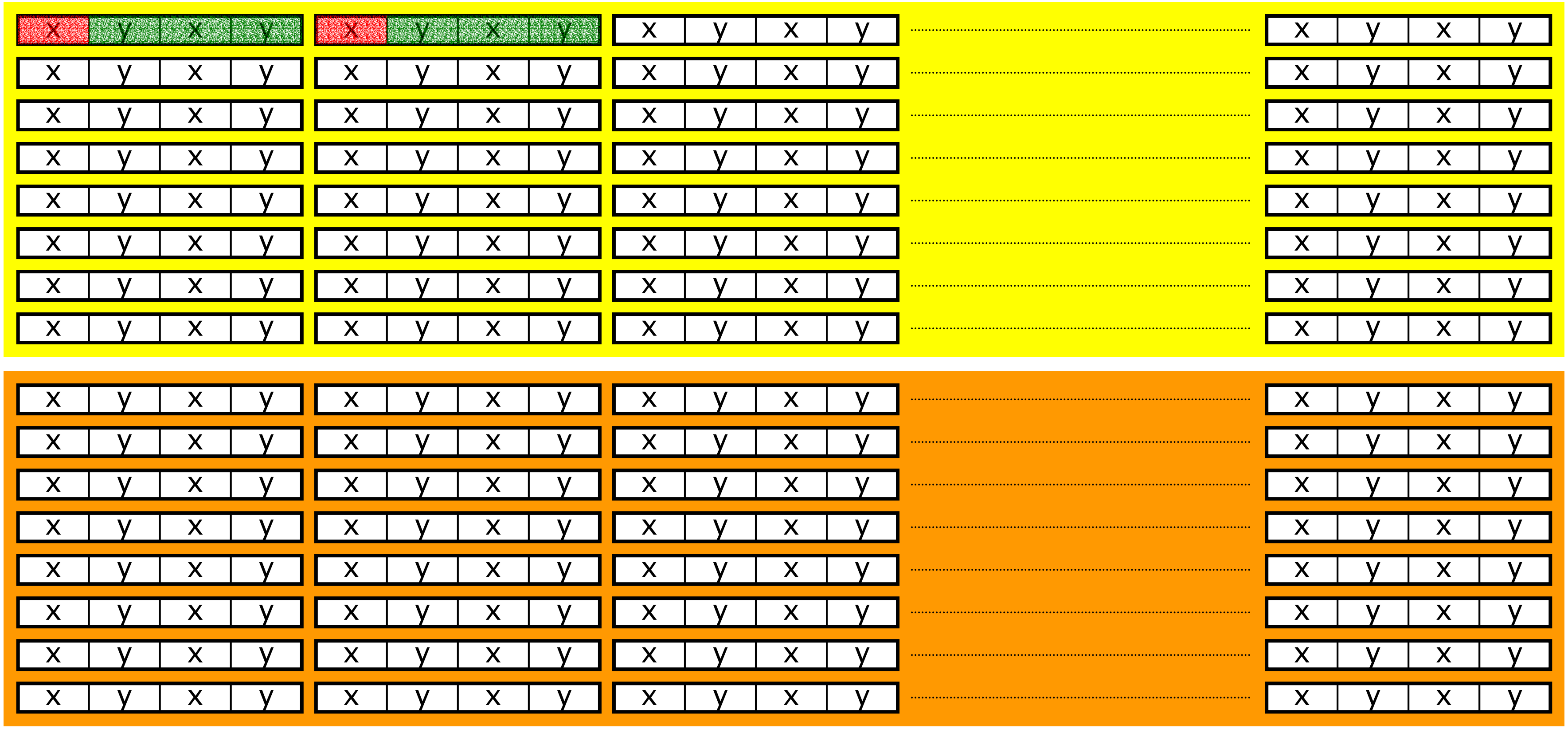


Answers to 6.16

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

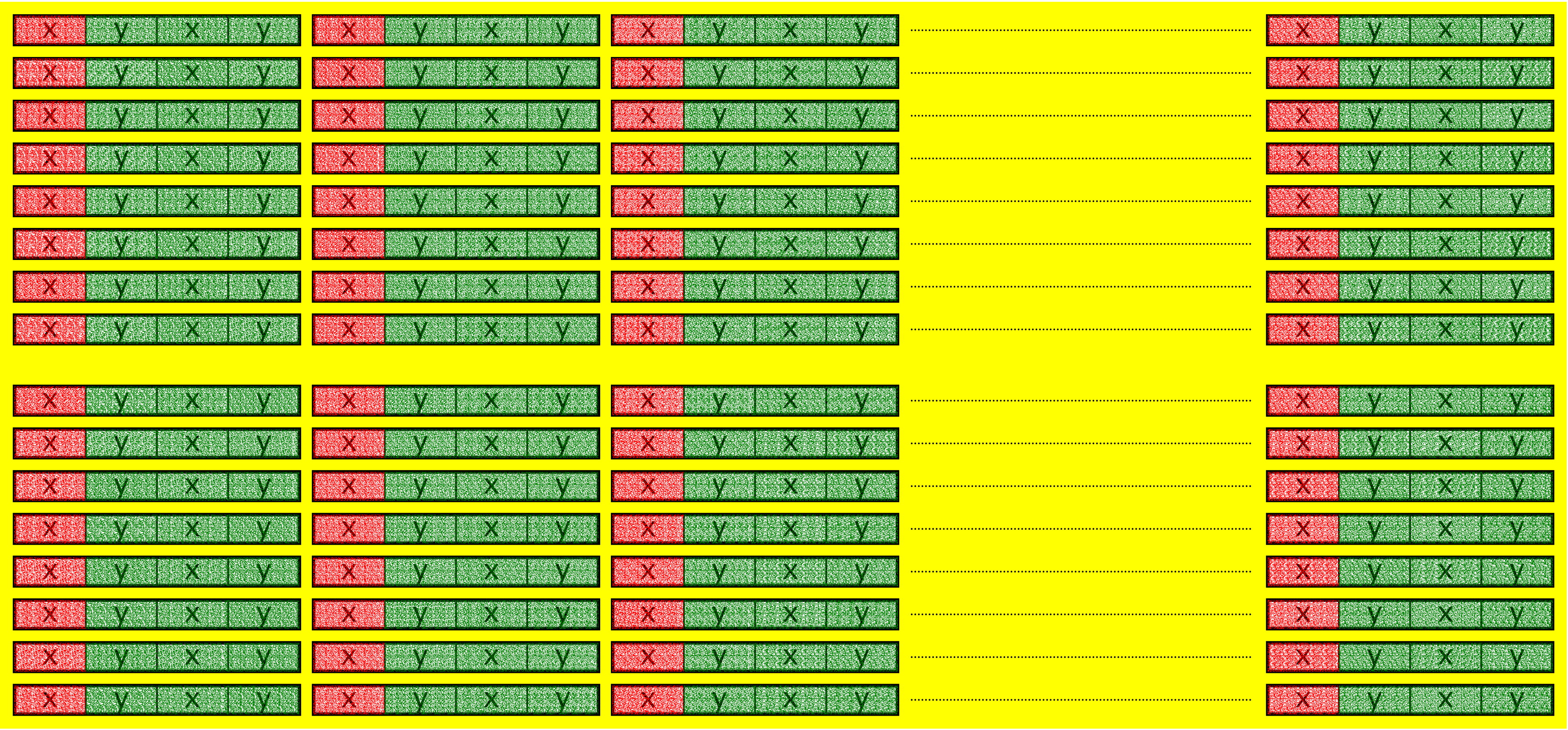
Stride of 1 word

- Access both x and y in row major order



Stride of 1 word

- Access both x and y in row major order



Answers to 6.17

- A: 512
- B: 128
  - All are compulsory misses
- C: 25%
- D: 25%
  - Cache size does not matter since all misses are compulsory
  - Though the block size does matter

Lab 4: Horner's Rule

Polynomial of degree d (d+1 coefficients)

$$P(x)=a_0+a_1x+a_2x^2+\cdots+a_dx^d$$

$$P(x)=a_0+(a_1+(a_2+(\cdots+(a_{d-1}+a_dx)x\cdots)x)x)x$$

Naïve code for Horner's Rule

```c
/* Horner's rule */
int poly_evalh(int *a, int degree, int x)
{
    int result = a[degree];
    int i;
    for (i = degree-1; i >= 0; i--)
        result = result*x+a[i];
    return result;
}
```