# Loop Optimizations

Riyadh Baghdadi, MIT

# Plan

» Data parallel programming model

» Loop optimizations: parallelization, vectorization, fusion, reordering, tiling

» Case study: Optimizing Matrix Multiplication

# Case Study: Matrix Multiplication

» Accelerate matrix multiplication by 565x

# Why Loops ?

- Time consuming
- Easy to optimize (parallelize, vectorize, improve locality, …)

# Data Parallel Model

- Program has many similar threads of execution
  - Each thread performs the same operation on different data
  - SIMD (Single Instruction Multiple Data)
- Main parallel programming model

# Examples

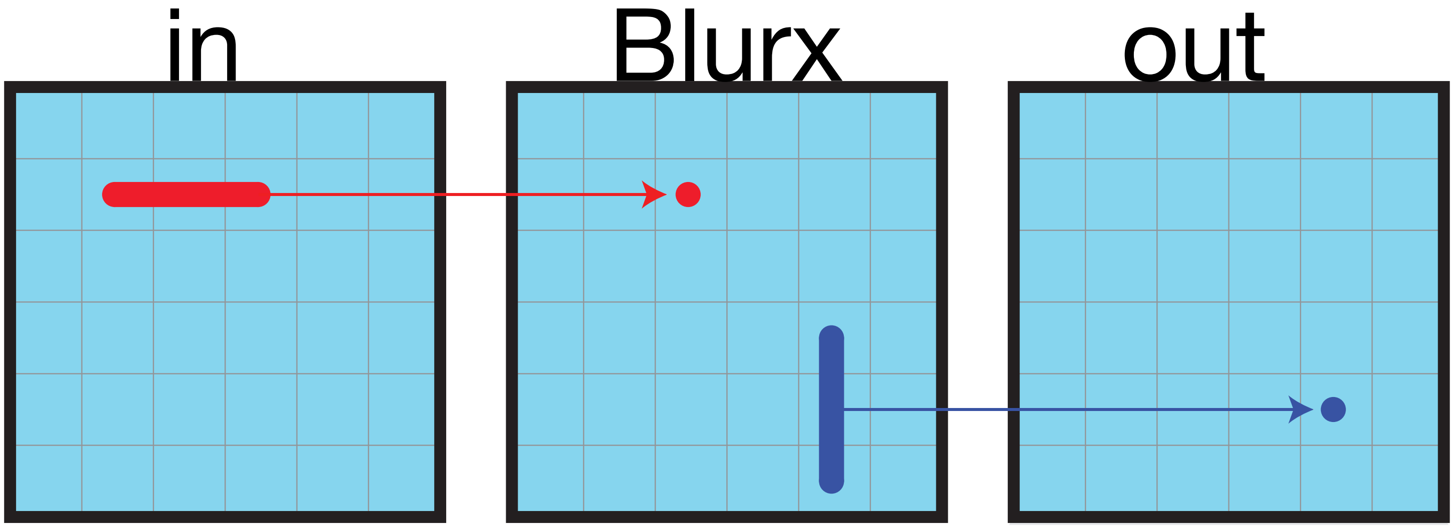
» Matrix addition: M1 + M2 = M3

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} + \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+9 & 2+8 & 3+7 \\ 4+6 & 5+5 & 6+4 \\ 7+3 & 8+2 & 9+1 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix}$$

# A simple example: 3x3 blur

# Noise reduction

# A simple example: 3x3 blur



# blur in C++

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());

  for (int y = 0; y < in.height(); y++)
      for (int x = 0; x < in.height(); x++)
          blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;


  for (int y = 0; y < in.height(); y++)
      for (int x = 0; x < in.height(); x++)
          out[x, y] = (blurx[x, y-1] + blurx[x, y] + blurx[x, y+1])/3;
}
```

# Loop Optimizations

# Loop Optimizations

- Loop parallelization: run the loop iterations in parallel

**Original**

```
for x in 0 … N
  for y in 0 … N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

**Run the loop by 2 threads**

**Thread 0**

```
for x in 0 … N/2
  for y in 0 … N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

**Thread 1**

```
for x in N/2 … N
  for y in 0 … N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

# Serial Execution

**Serial y**
**Serial x**

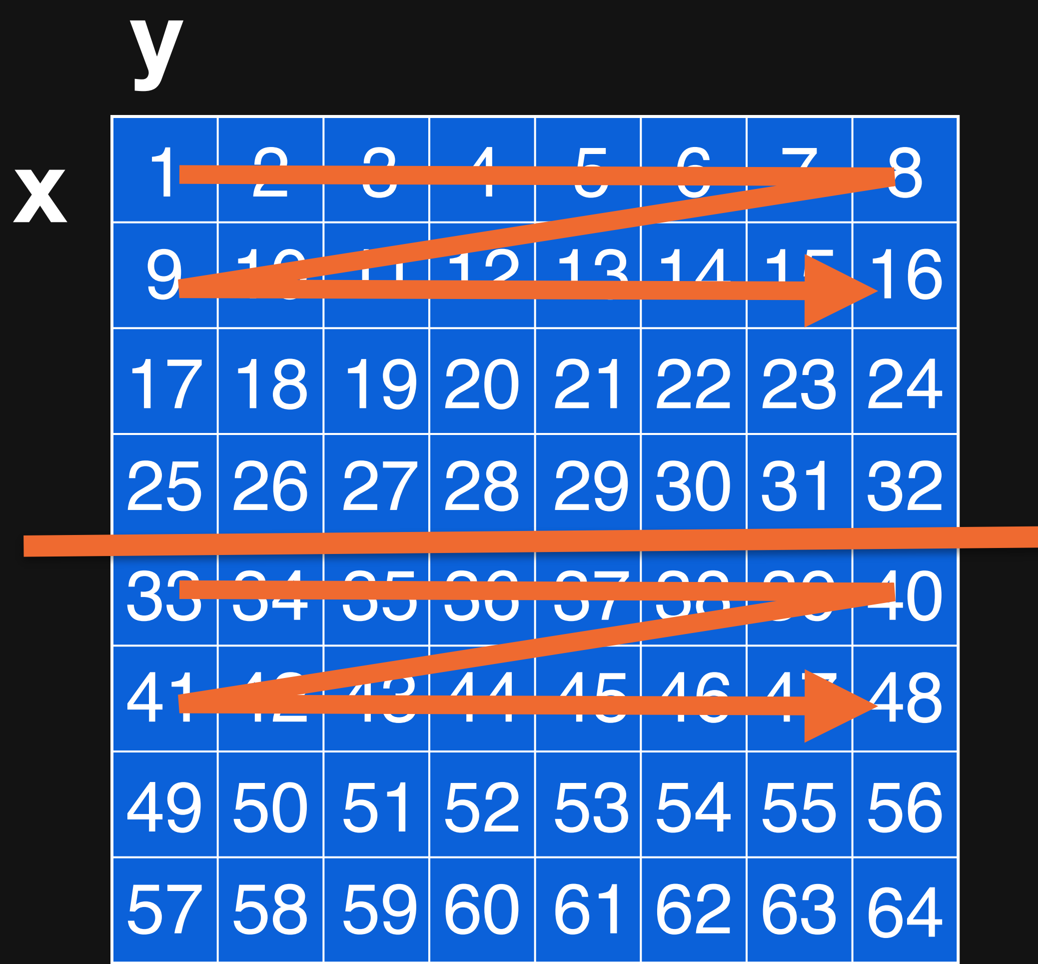
y

x

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 |
| 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |

# Parallel Execution

**Serial y**
**Parallel x**



# Loop Optimizations

- Loop parallelization: run the loop iterations in parallel

**OpenMP Parallelization**

```
#pragma omp parallel for
for x in 0 … N
  for y in 0 … N
    blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;
```

# Loop Optimizations

- Loop vectorization: use vector instructions

**Original**

```
for x in 0 … N
  for y in 0 … N
    blurx[x, y]     =  (in[x-1, y]+
                        in[x,   y]+
                        in[x+1, y])/3;
```

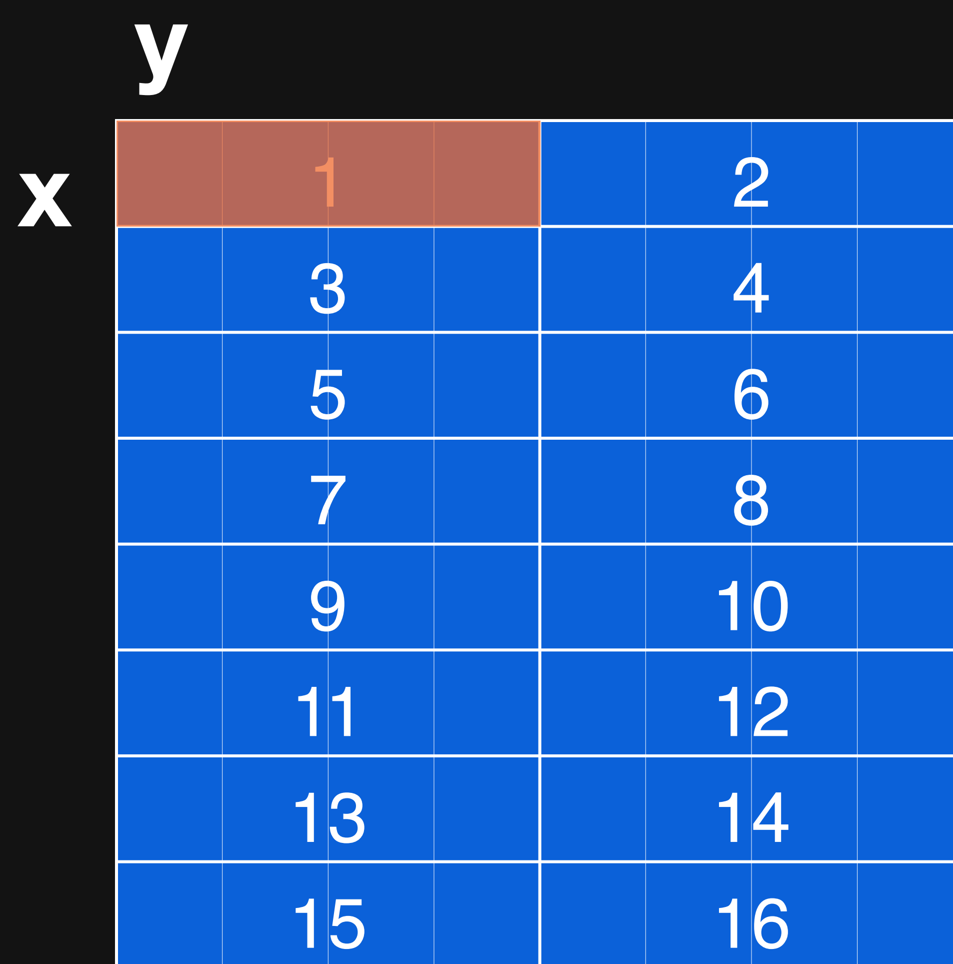
**Run the y loop using vector instructions**

**Thread 0**

```
for x in 0 … N
  for y in 0 … N/4
    blurx[x, y…y+3] =  (in[x-1, y…y+3]+
                        in[x,   y…y+3]+
                        in[x+1, y…y+3])/3;
```

# Vectorization

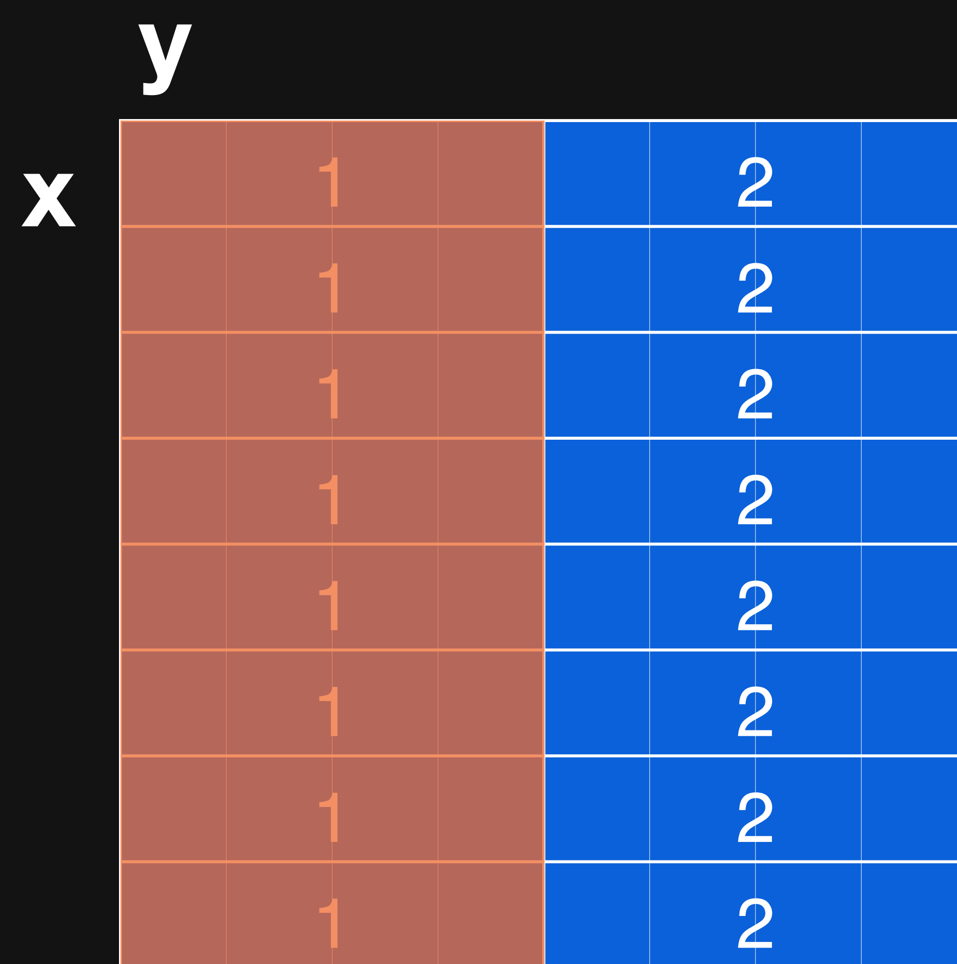
**Serial y,**
**Vectorize x**
by 4



# Parallel, Vectorized

**Parallel y, Vectorize x**

by 4



# Loop Optimizations

- Loop fusion: transform two successive loops into one loop

**Original**

```
for i in 0 … N
  for j in 0 … N
    temp(i, j) = 0;

for i in 0 … N
  for j in 0 … N
    out(i, j) = temp(i, j);
```

**Fused loops**

```
for i in 0 … N
  for j in 0 … N
    temp(i, j) = 0;
    out(i, j) = temp(i, j);
```

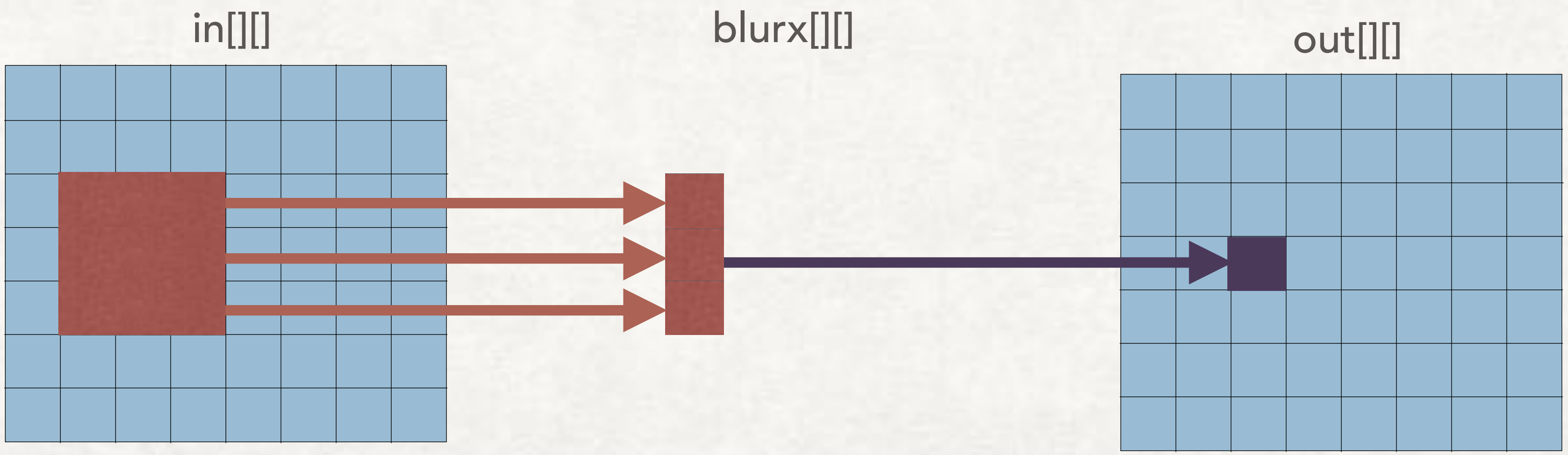
# Exercice: fuse the following two loops

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());

  for (int y = 0; y < in.height(); y++)
      for (int x = 0; x < in.height(); x++)
          blurx[x, y] = (in[x-1, y] + in[x, y] + in[x+1, y])/3;


  for (int y = 0; y < in.height(); y++)
      for (int x = 0; x < in.height(); x++)
          out[x, y] = (blurx[x, y-1] + blurx[x, y] + blurx[x, y+1])/3;
}
```

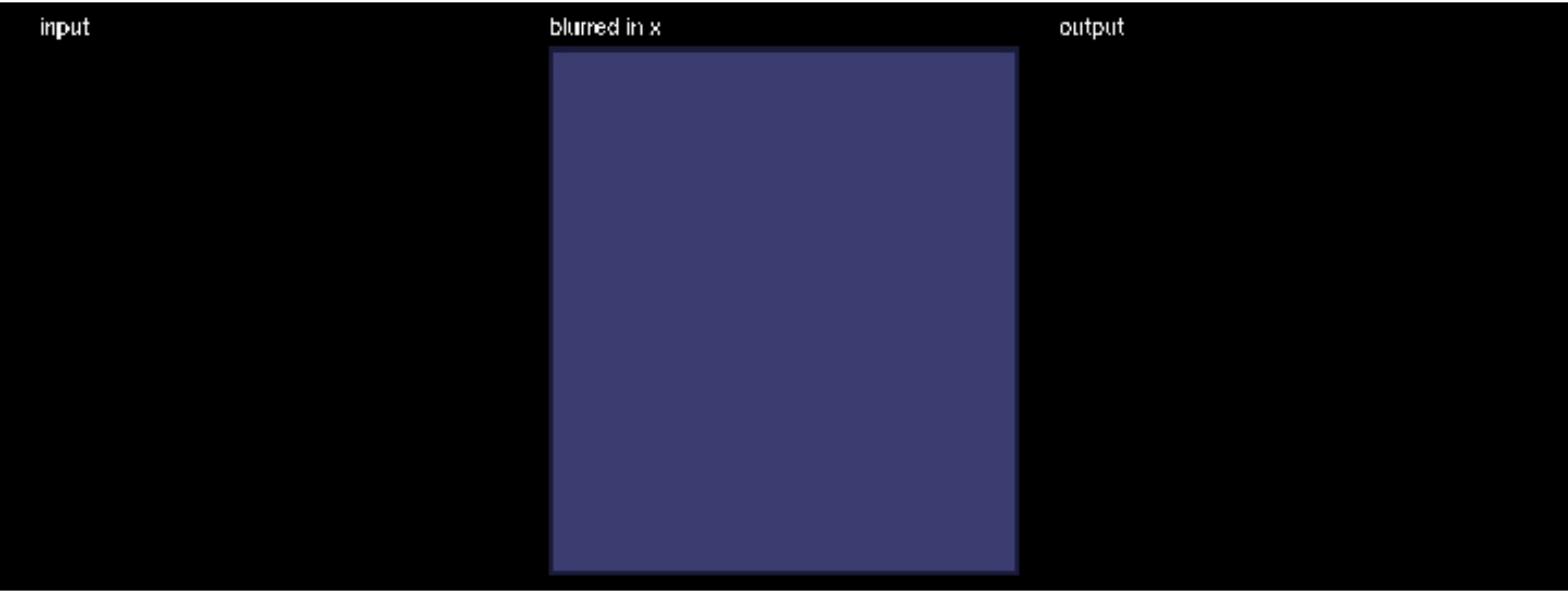
# FUSION (WITH CODE DUPLICATION)

BlurXY

```
for each y in 1..2047:
    for each x in 0..3072:
        alloc blurx[0..2]
         blurx[0]= in[y-2][x-1]+in[y-2][x]+in[y-2][x+1]
         blurx[1]= in[y-1][x-1]+in[y-1][x]+in[y-1][x+1]
         blurx[2]= in[y   ][x-1]+in[y   ][x]+in[y   ][x+1]
         out[y][x] = blurx[0] + blurx[1] + blurx[2]
```

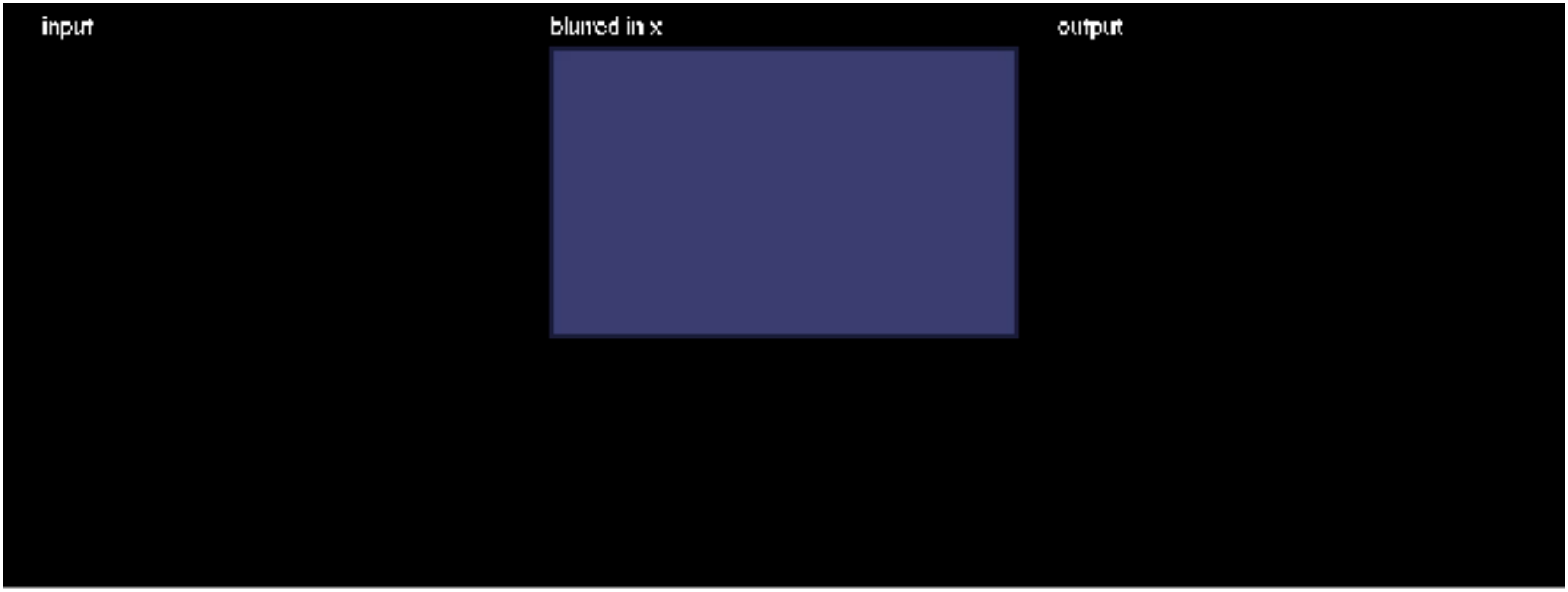


# Serial, no fusion



# Serial, fusion

# Parallel, vectorized, fused



# Loop Optimizations

- Loop reordering: reorder two loop levels to improve the data access pattern

**Original**

```
for i in 0 … N
  for j in 0 … N
    out(j, i) = A(j, i);
```

**After reordering**

```
for j in 0 … N
  for i in 0 … N
    out(j, i) = A(j, i);
```

# Loop Optimizations

- But what if we have

**Original**

```
for i in 0 … N
  for j in 0 … N
    out(j, i) = A(i, j);
```

- Loop reordering is not efficient in this case

# Loop Optimizations

- Array transposition: transpose the array A to make the access contiguous

**Original**

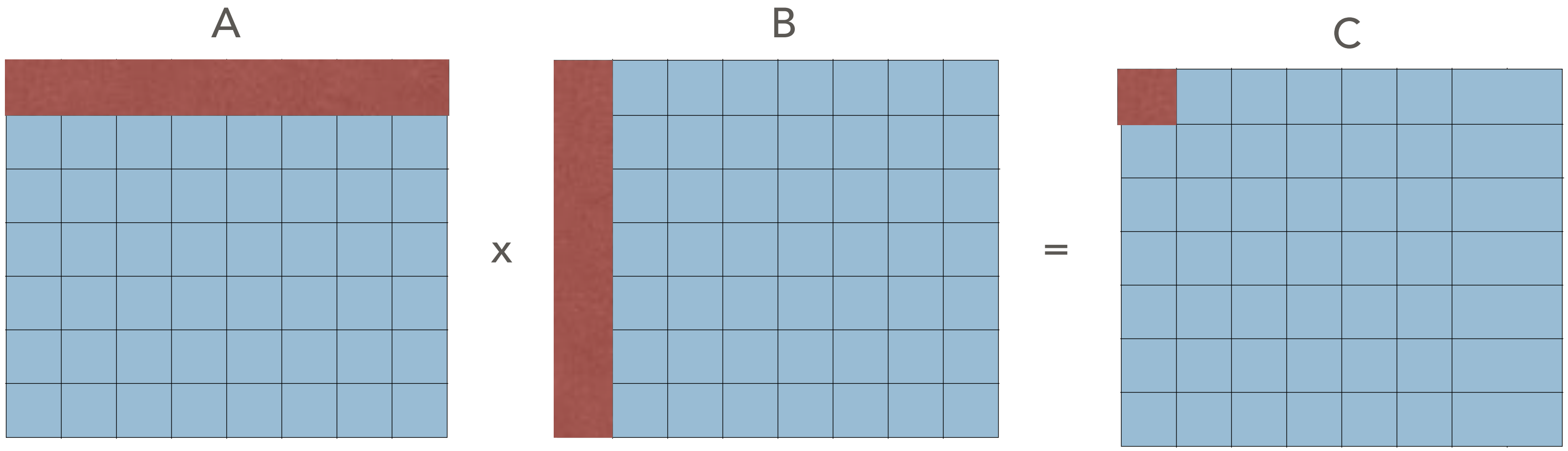
```
for i in 0 … N
  for j in 0 … N
    B(i, j) = A(j, i) + 1;
for i in 0 … N
  for j in 0 … N
    C(i, j) = A(j, i) + 2;
```

**Transposed**

```
// Transpose array
for i in 0 … N
  for j in 0 … N
    A2(i, j) = A(j, i);

for i in 0 … N
  for j in 0 … N
    B(i, j) = A2(i, j) + 1;
for i in 0 … N
  for j in 0 … N
    C(i, j) = A2(i, j) + 2;
```

# Loop Optimizations

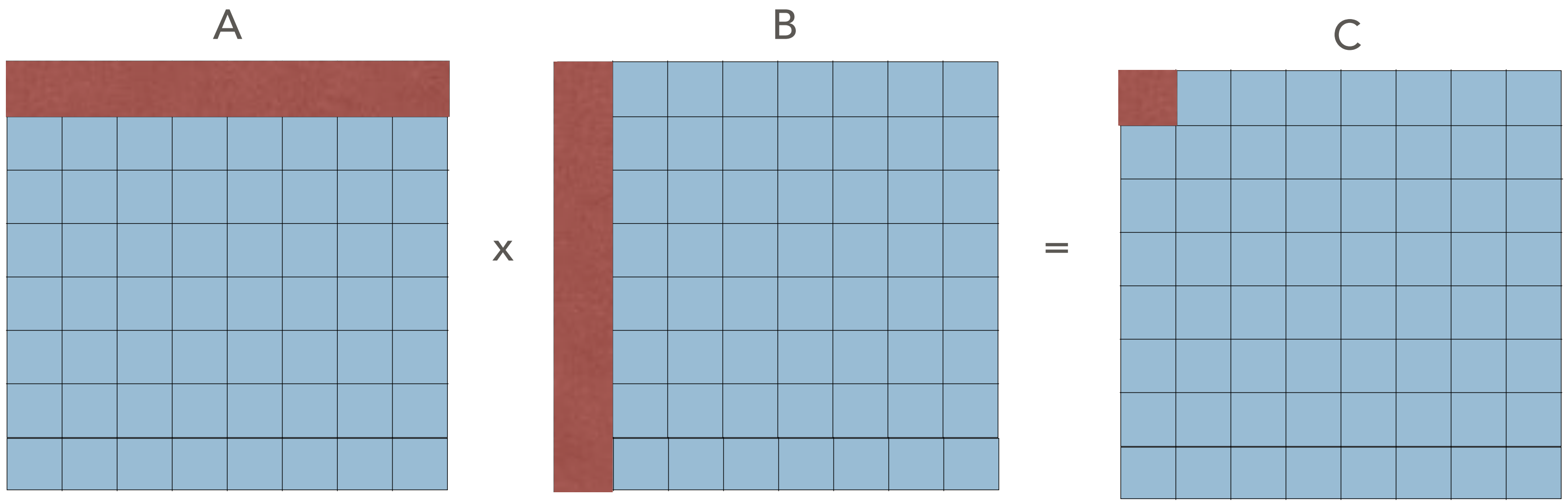
- Loop tiling: run code by tiles (blocks) to improve data locality



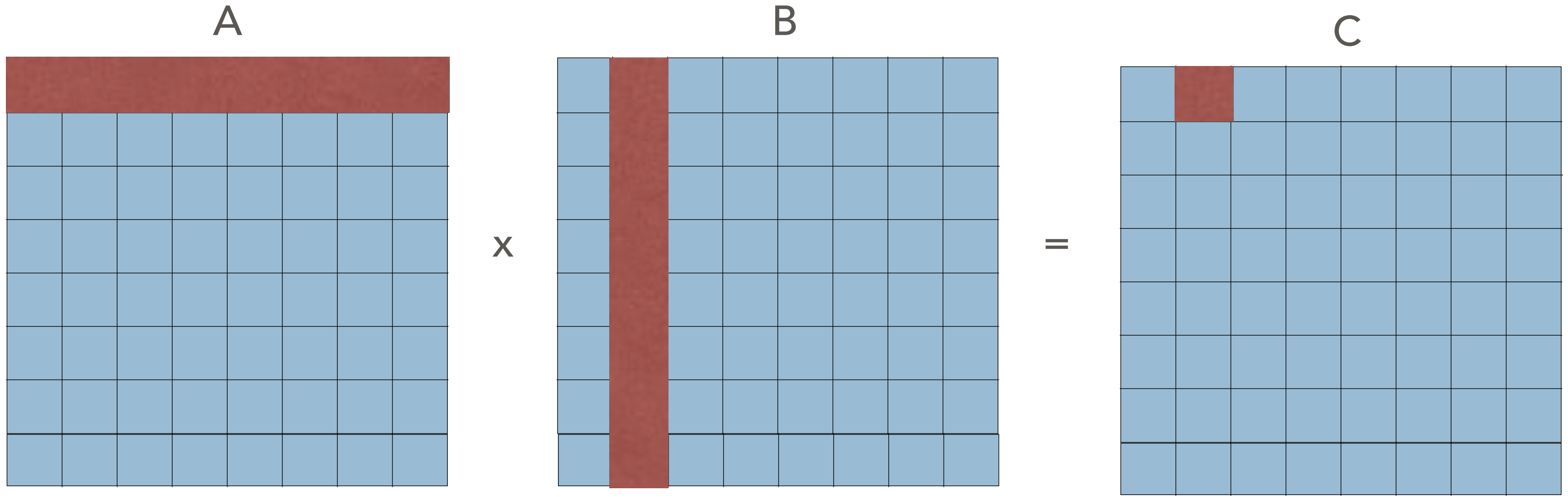

**Original**

```
for i in 0 … N
  for j in 0 … N
    C(i, j) = 0
    for k in 0 … N
      C(i, j) += A(i, k) * B(k, j)
```

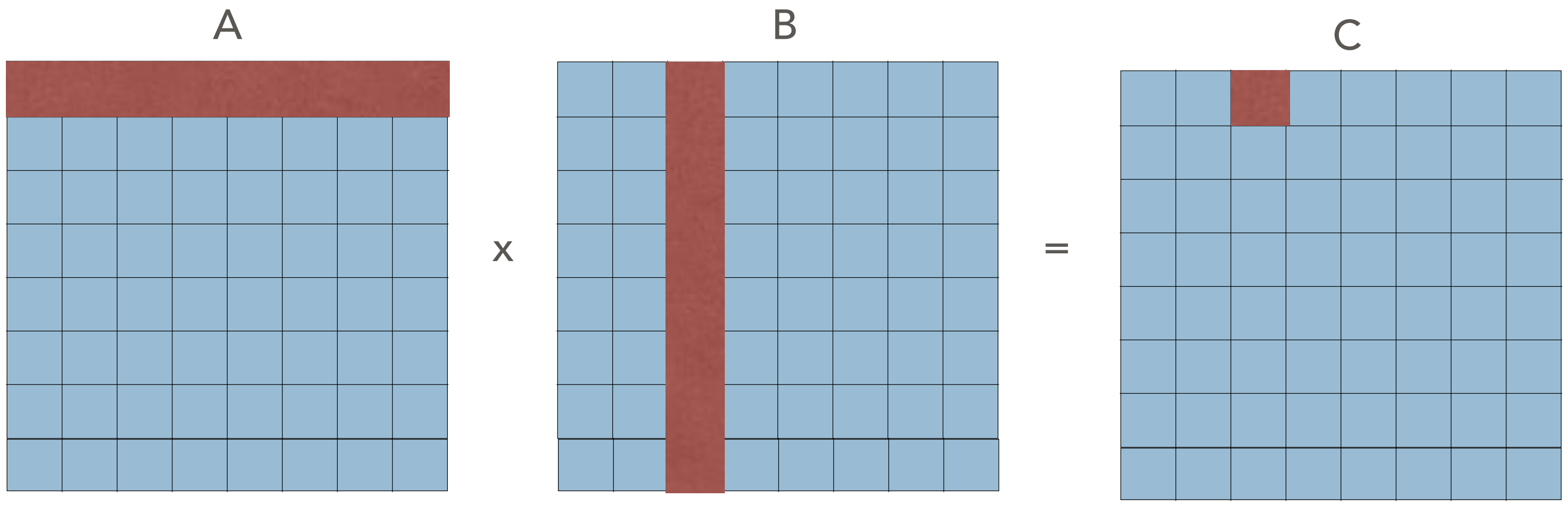
# Matrix multiplication access pattern



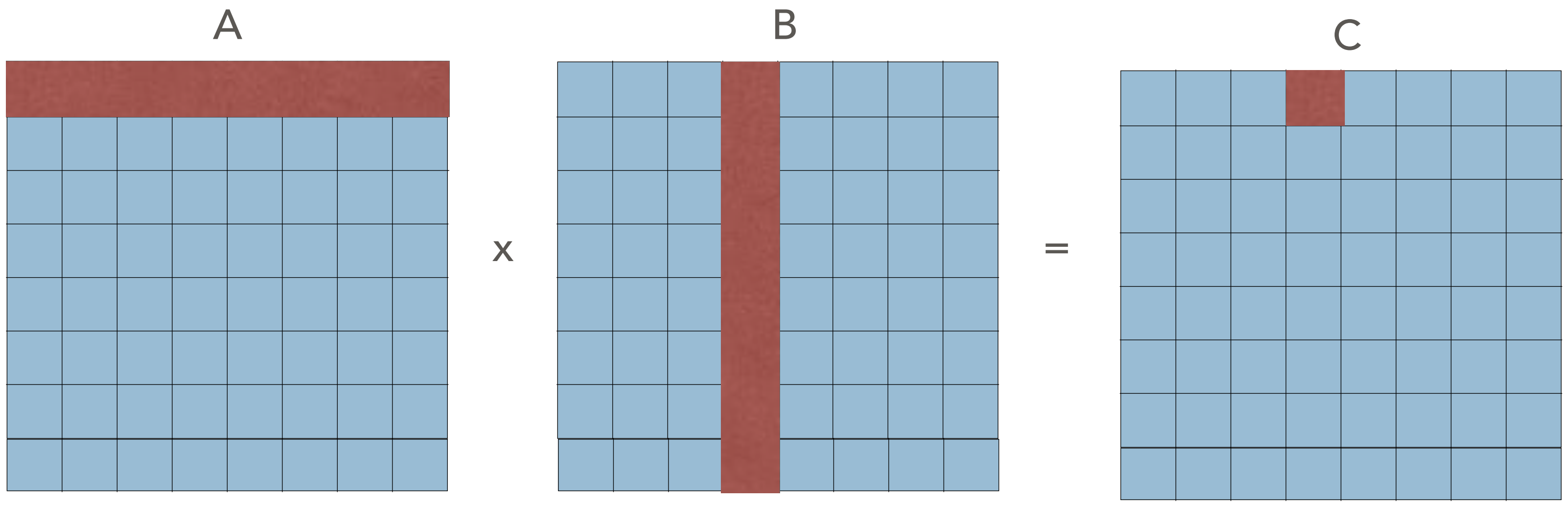
# Matrix multiplication access pattern



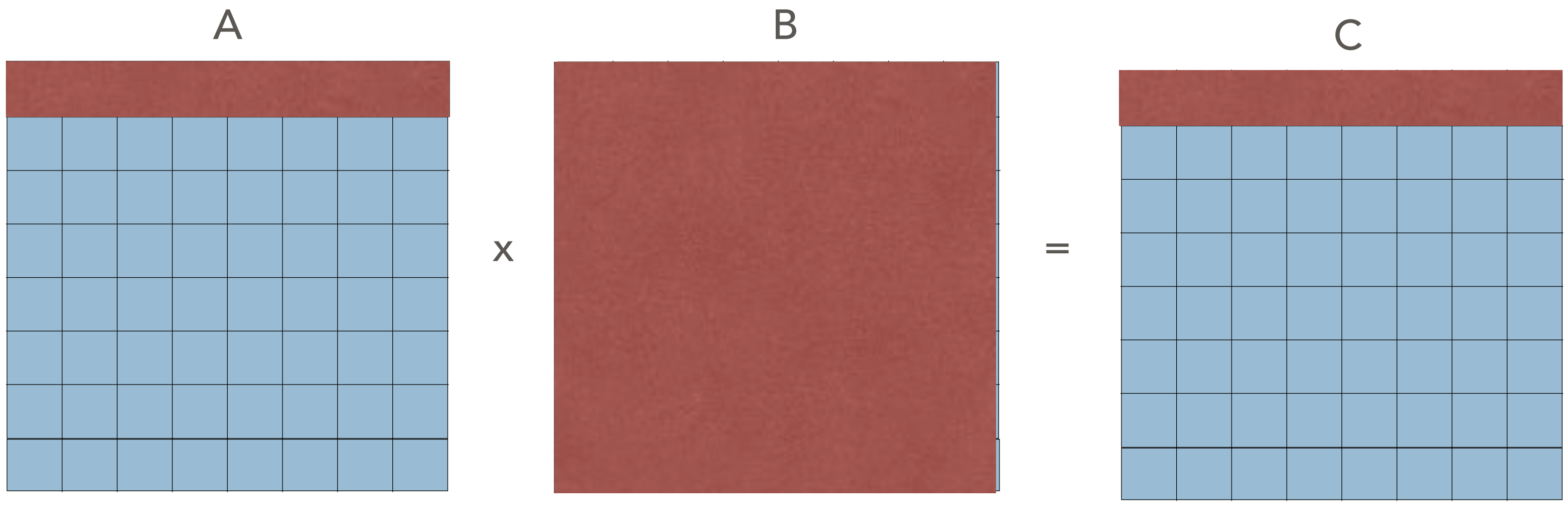
# Matrix multiplication access pattern



# Matrix multiplication access pattern

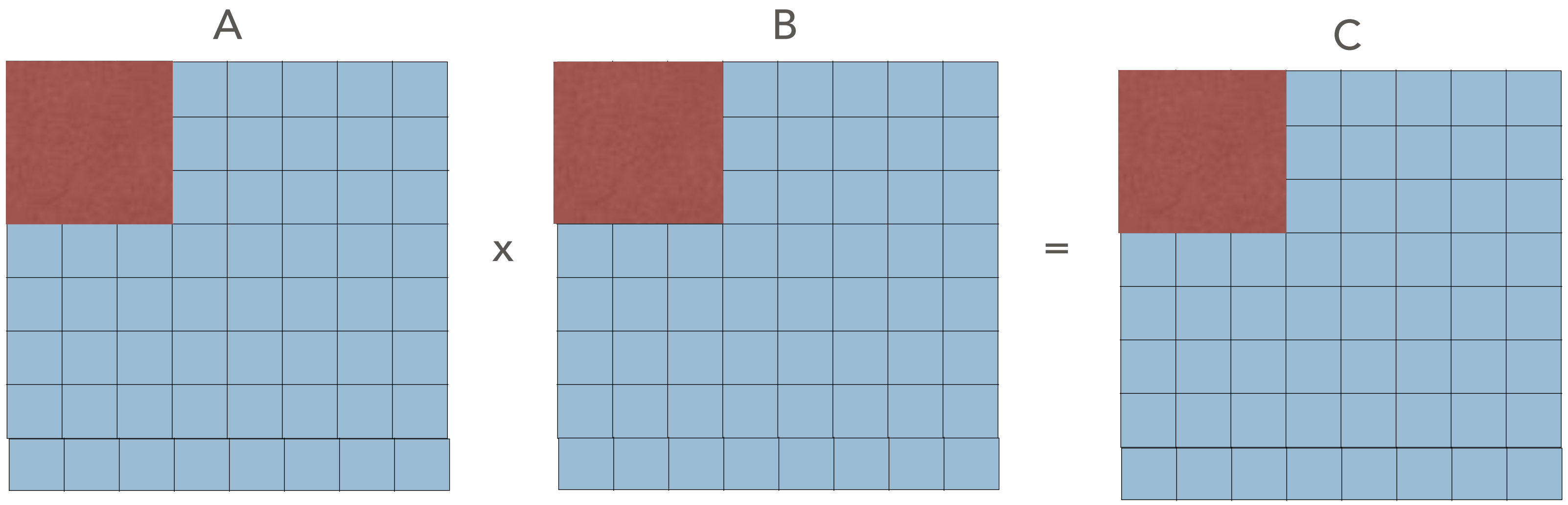


# Matrix multiplication access pattern

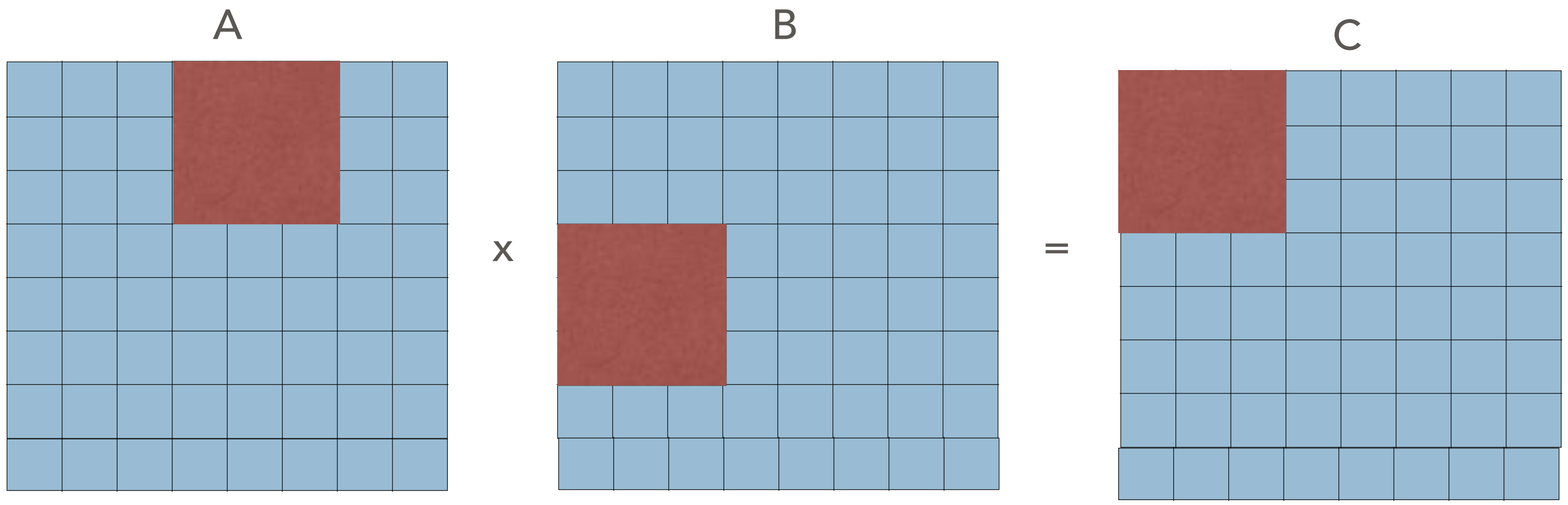



For a matrix with N rows, we read B N times

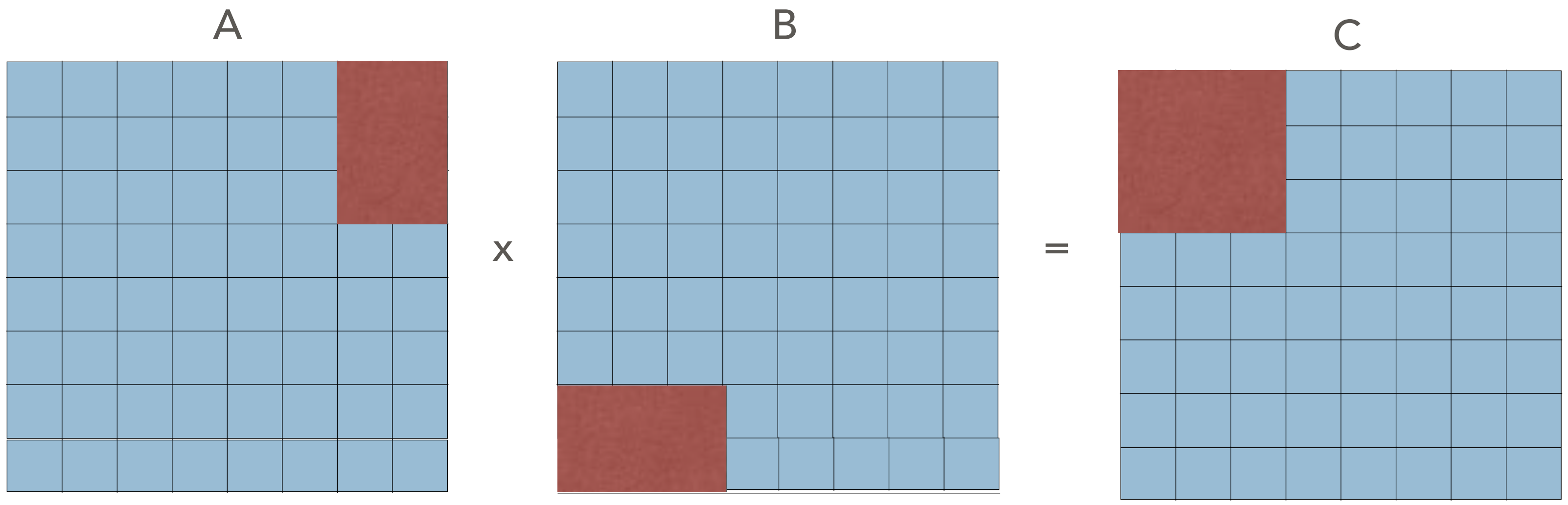
# Matrix multiplication access pattern



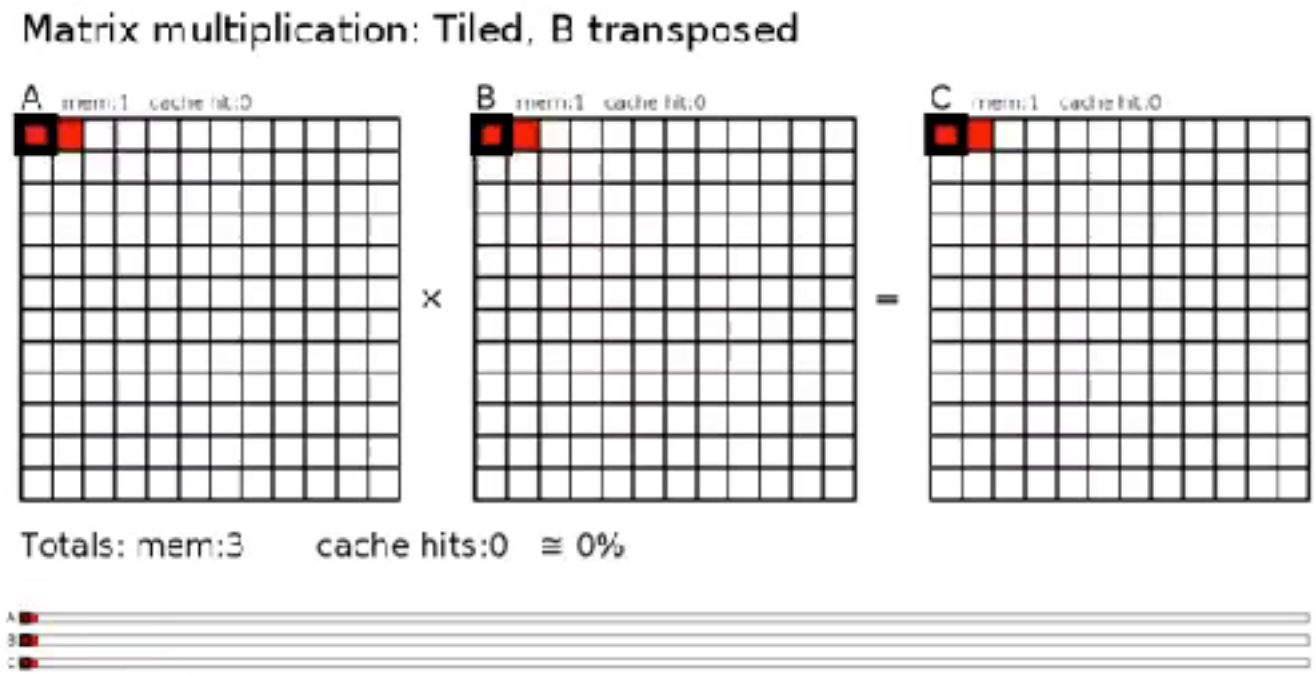
# Matrix multiplication access pattern



# Matrix multiplication access pattern



# Tiling


Matrix multiplication: Tiled, B transposed
A mem:1 cache hit:0
B mem:1 cache hit:0
C mem:1 cache hit:0
×
=
Totals: mem:3 cache hits:0 ≅ 0%

# Tiling

**Original**

```
for i in 0 … N
  for j in 0 … N
    C(i, j) = 0
    for k in 0 … N
      C(i, j) += A(i, k) * B(k, j)
```

**Tiled**

```
for (i0=0; i0<=(N-1)/32; i0++)
   for (j0=0; j0<=(N-1)/32; j0++)
      for (i1=32*i0; i1<=N-1; i1++)
         for (j1=32*j0; j1<=N-1; j1++)
            C(i1, j1) = 0
            for k in 0 … N
                C(i1, j1) += A(i1, k) * B(k, j1)
```

# Case Study: Matrix Multiplication

# Case Study: Matrix Multiplication

- Matrix multiplication written in Java (**with object oriented concepts**).
  - 32800 ms
- Problems
  - OOP requires lot of function calls (get, set …)

# Case Study: Matrix Multiplication

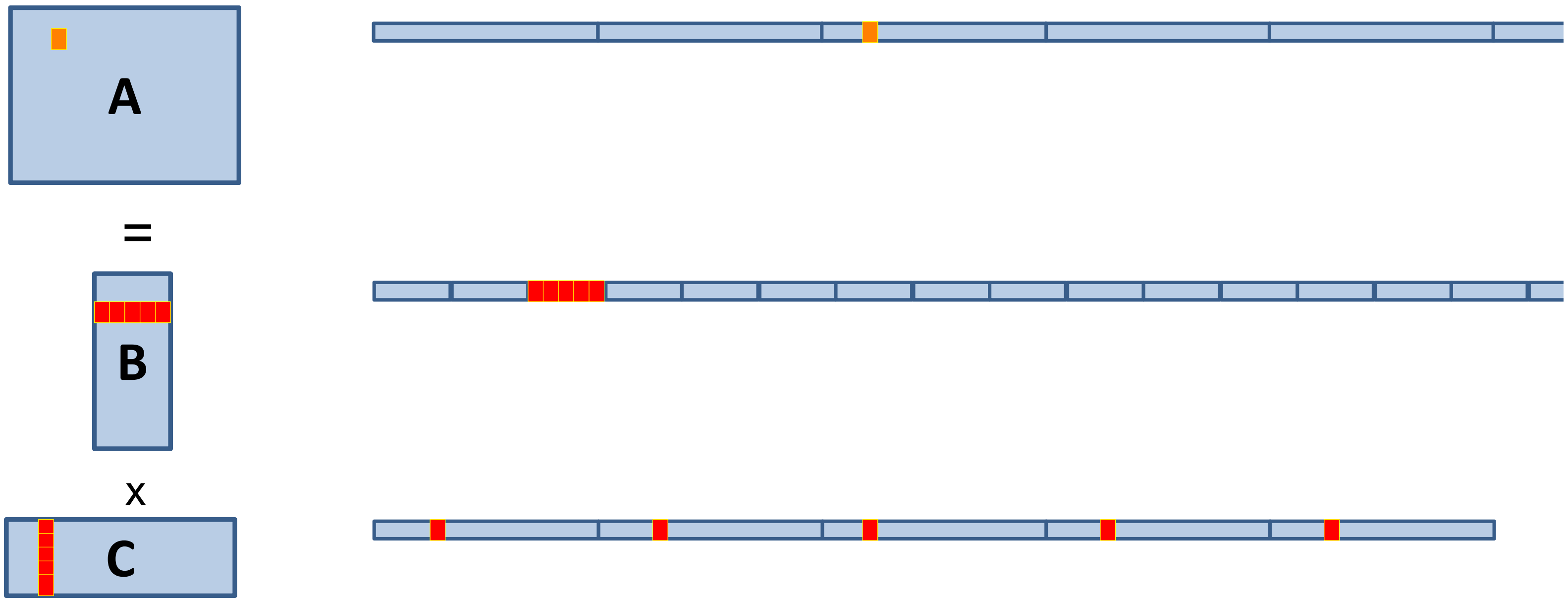
- **No object oriented concepts**
  - 15300 ms
  - 2.2x
- Problems
  - Java is interpreted
  - Java check out-of-boundary array accesses

# Case Study: Matrix Multiplication

- **In C**
  - 7500 ms
  - 2.1x
- Problems
  - Memory accesses to non-contiguous memory

# Issues with Matrix Representation

## Scanning the memory




## Contiguous accesses are better

- Data fetch as cache line (Core 2 Duo  64 byte L2 Cache line)
- Contiguous data → Single cache fetch supports 8 reads of doubles

# Case Study: Matrix Multiplication

- **Transpose B**
  - 2200 ms
  - 3.4x

# Case Study: Matrix Multiplication

- **Tiling**
  - 1388 ms
  - 1.7x

# Case Study: Matrix Multiplication

» **Vectorization**

  » 511 ms

  » 2.7x

# Case Study: Matrix Multiplication

- **BLAS**
  - 196 ms
  - 2.7x

# Case Study: Matrix Multiplication

- **Parallel BLAS**
  - 58 ms
  - 3.5x

# Case Study: Matrix Multiplication

| | Java OOP | No Object | In C | Transpose | Tiled | Vectorize | BLAS | Par BLAS |
|---|---|---|---|---|---|---|---|---|
| **ms** | 32800 | 15300 | 7500 | 2200 | 1388 | 511 | 196 | 58 |
| | | 2.2x | 2.1x | 3.4x | 1.7x | 2.8x | 2.7x | 3.5x |

**Speedup: 565x**