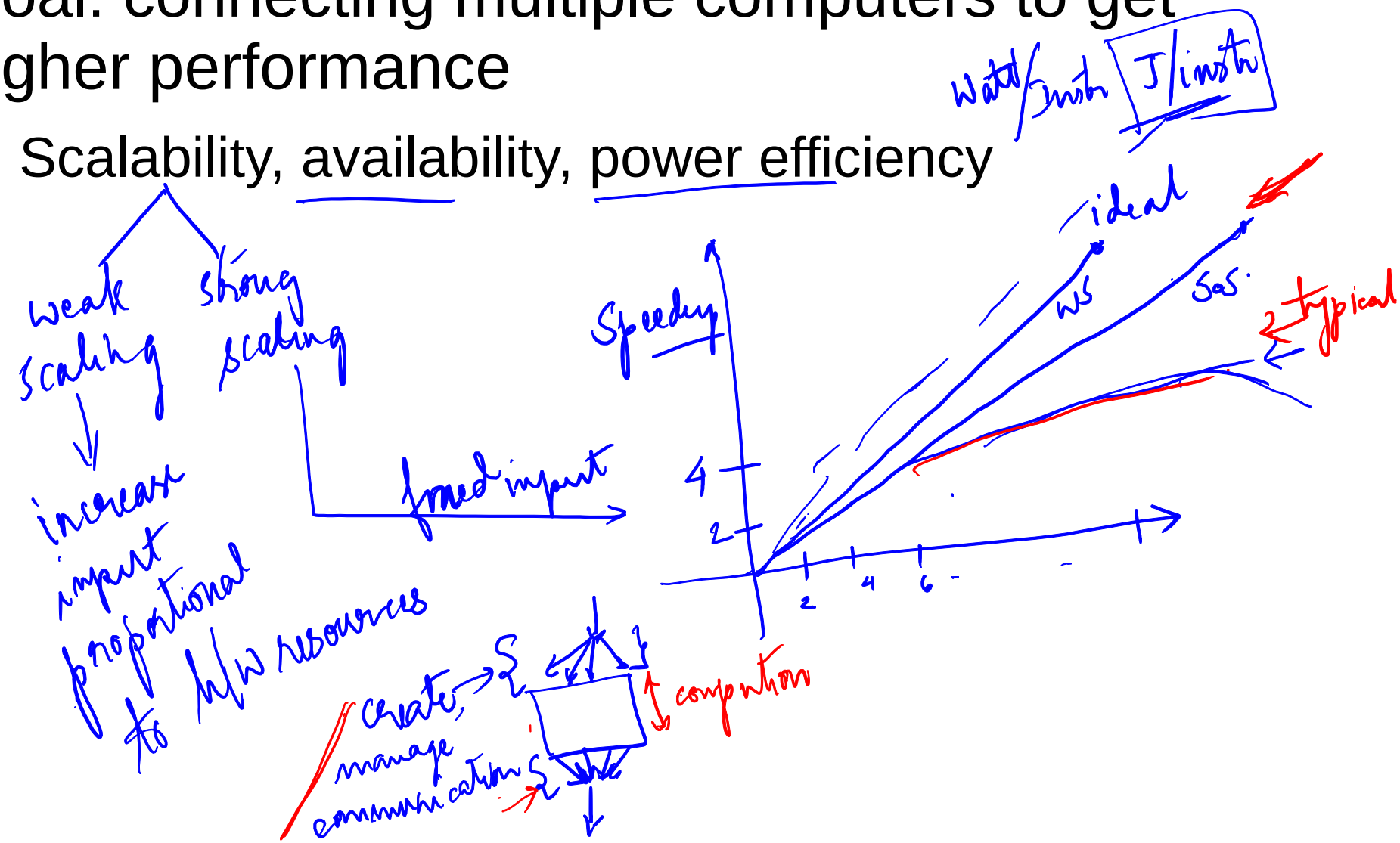
M4 – Parallelism

Outline

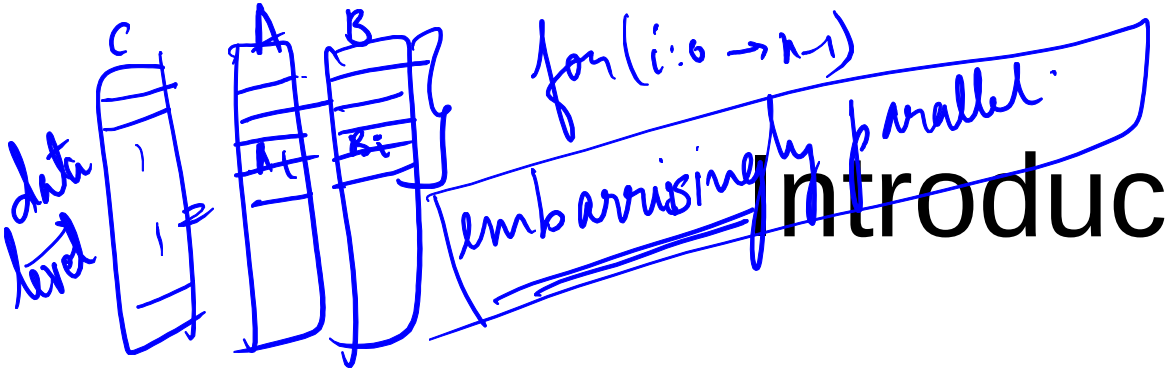
- Parallelism
- Flynn's classification
- Vector Processing
  - Subword Parallelism
- Synchronization
  - ISA
- Symmetric Multiprocessors, Distributed Memory Machines
  - Shared Memory Multiprocessing, Message Passing
- Cache coherence

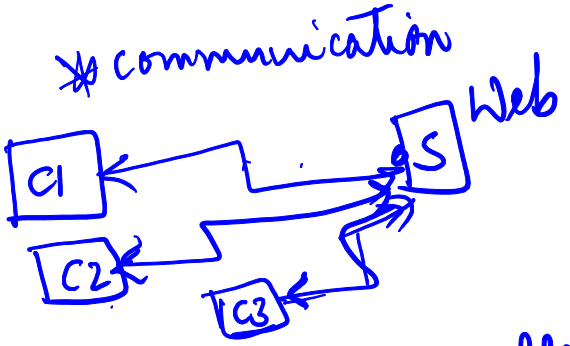
Introduction

- Goal: connecting multiple computers to get higher performance
  - Scalability, availability, power efficiency



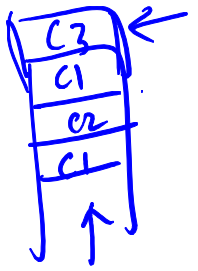
Introduction

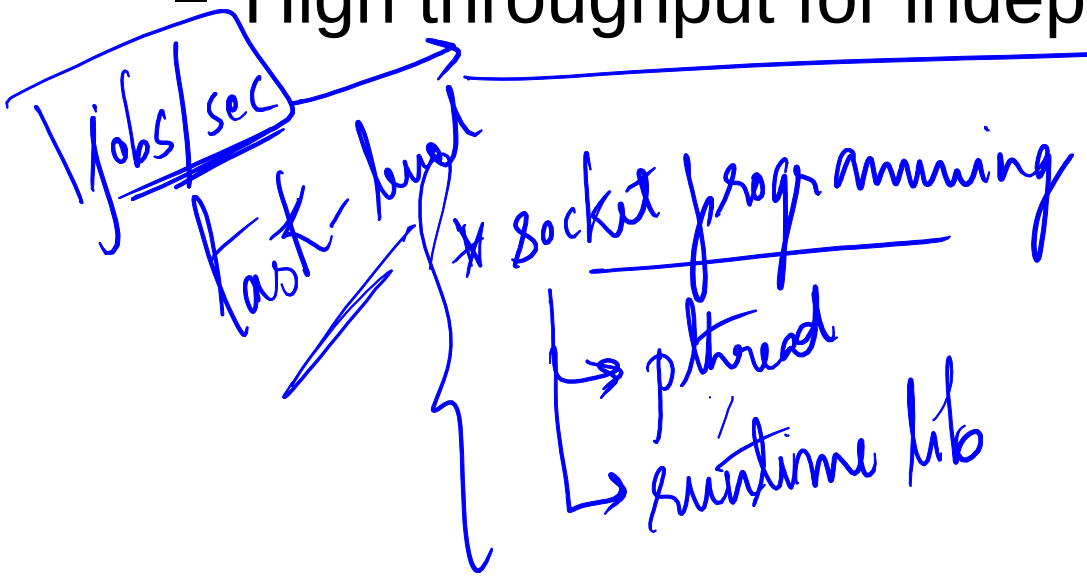


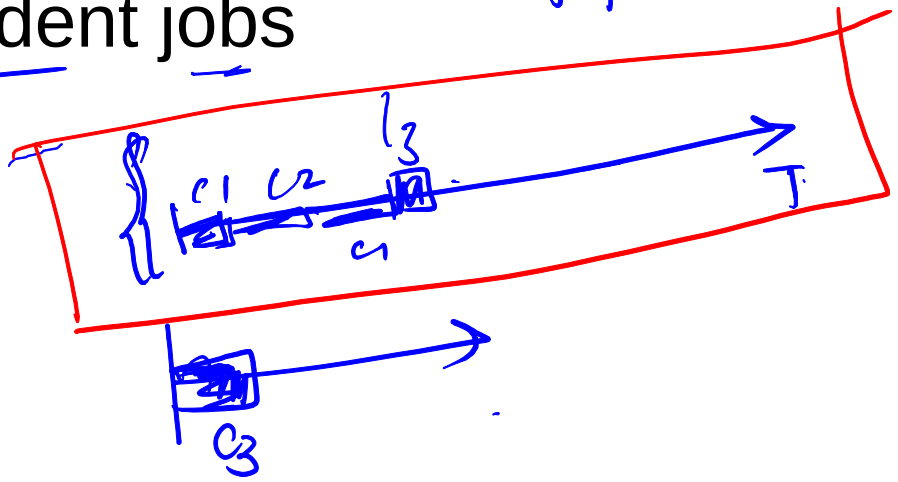


- Goal: connecting multiple computers to get higher performance
  - Scalability, availability, power efficiency
- Task-level (process-level) parallelism
  - High throughput for independent jobs

parallel





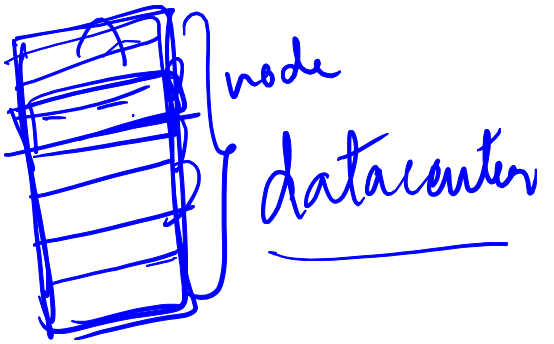


Introduction

- Goal: connecting multiple computers to get higher performance
  - Scalability, availability, power efficiency
- Task-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors

Motivation for Multiprocessing

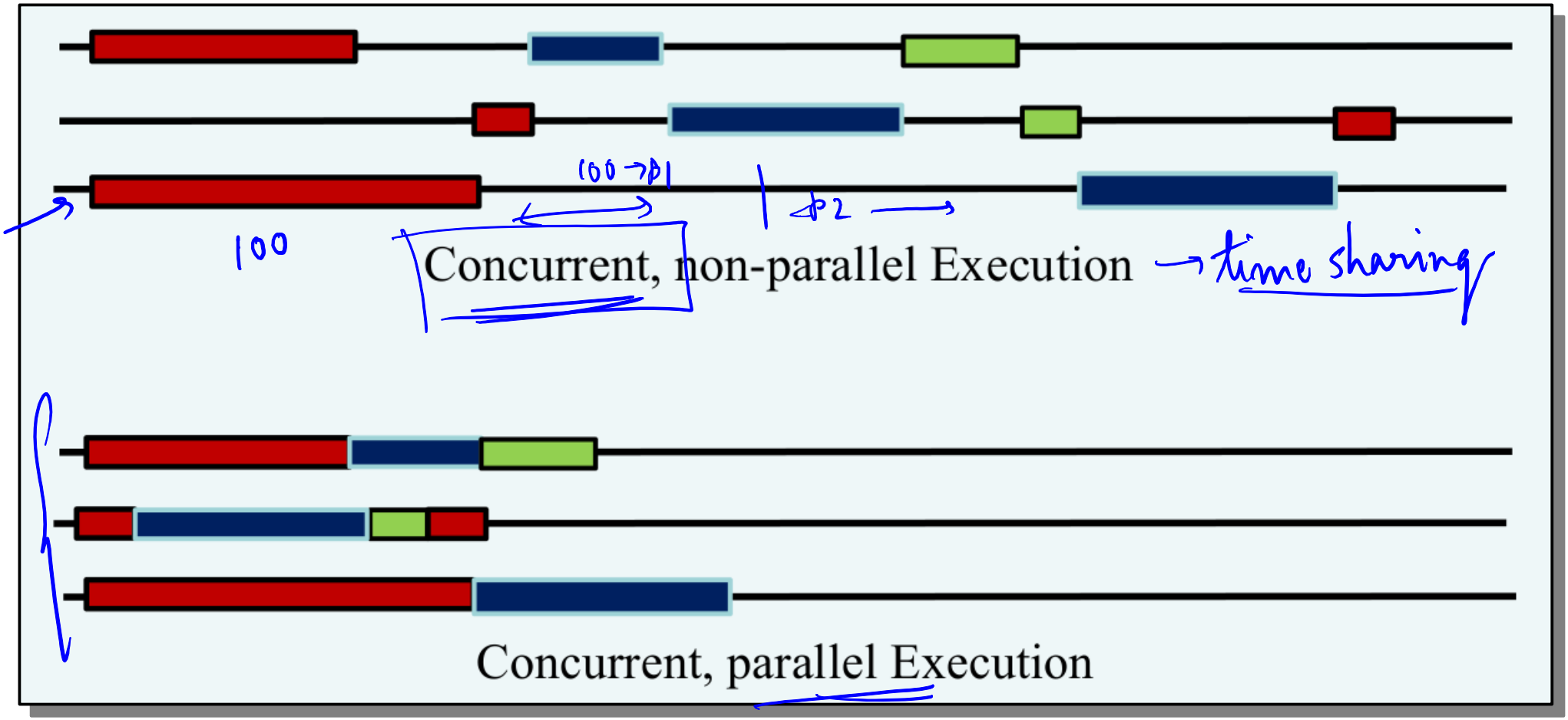
- Performance
- Clusters, Software as a Service
- Data intensive applications
- Natural parallelism in large scientific applications
- More return on investments by replicating current designs



Terms

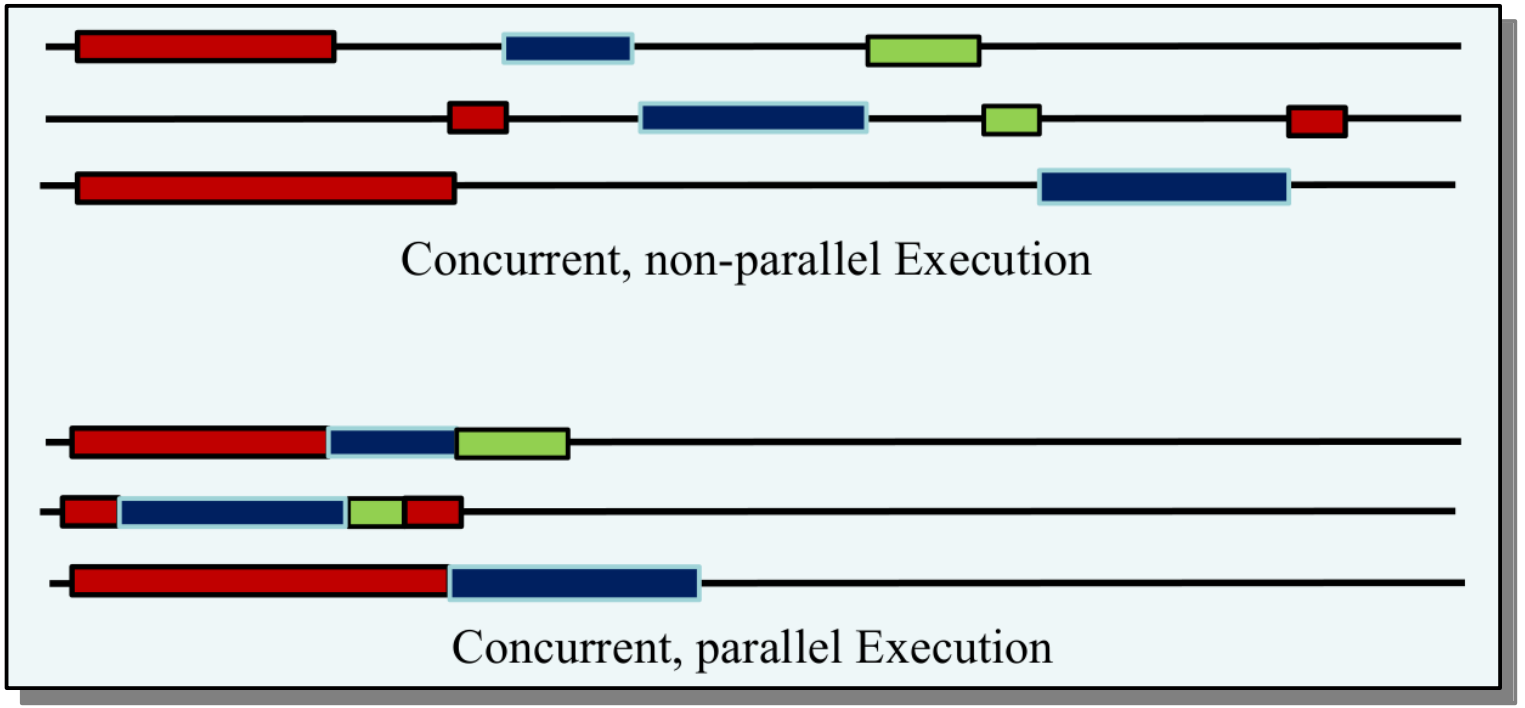
- Software
  - Sequential, Concurrent
- Hardware
  - Serial, Parallel

Concurrency vs. Parallelism



Concurrency vs. Parallelism

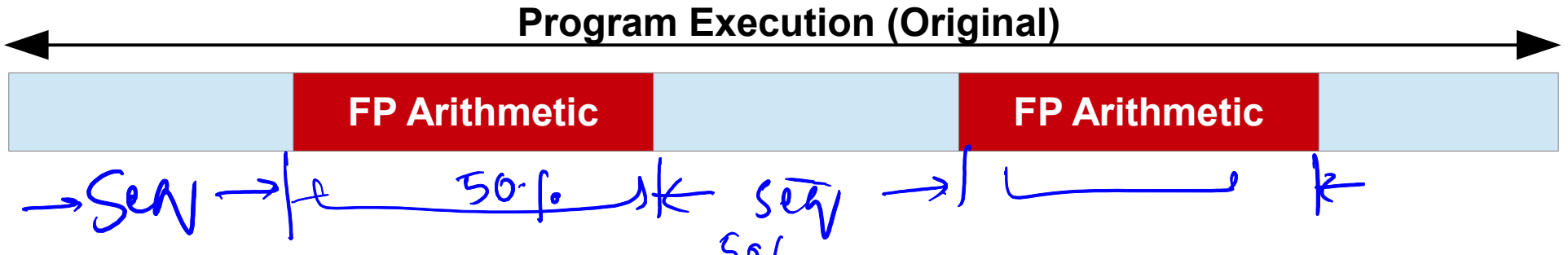
- Concurrency: A condition of a system in which multiple tasks are logically active at one time.
- Parallelism: A condition of a system in which multiple tasks are actually active at one time.



Amdahl's Law

- Gene Amdahl (1922 – 2015)
- Law quantifies a fundamental limitation of parallel computing

Amdahl's Law



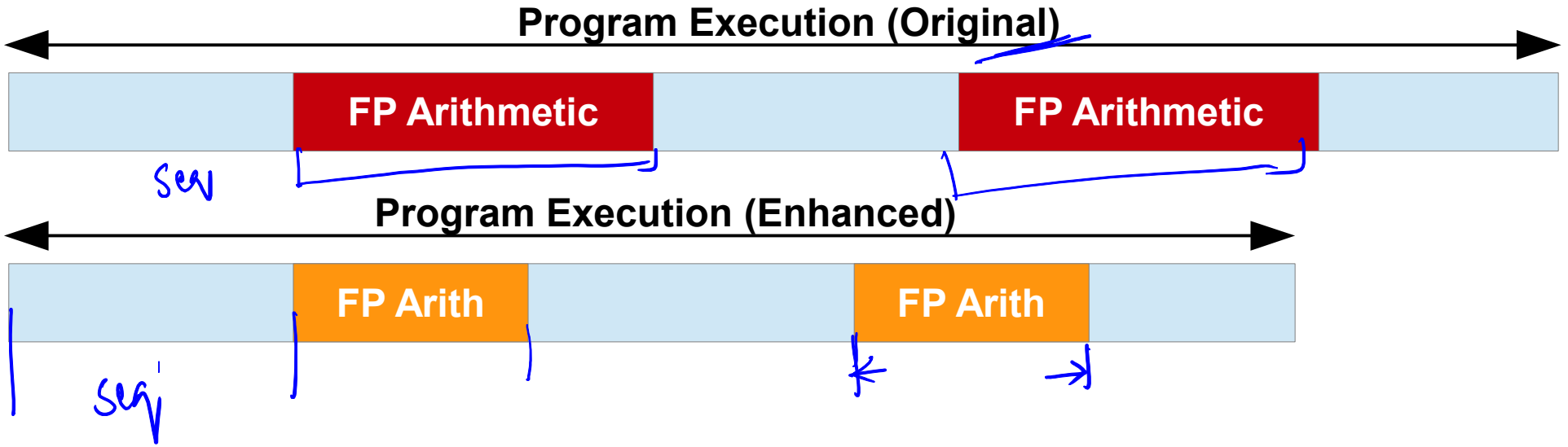
Seq → 50% ← Seq → ← 50%

max speedup ?

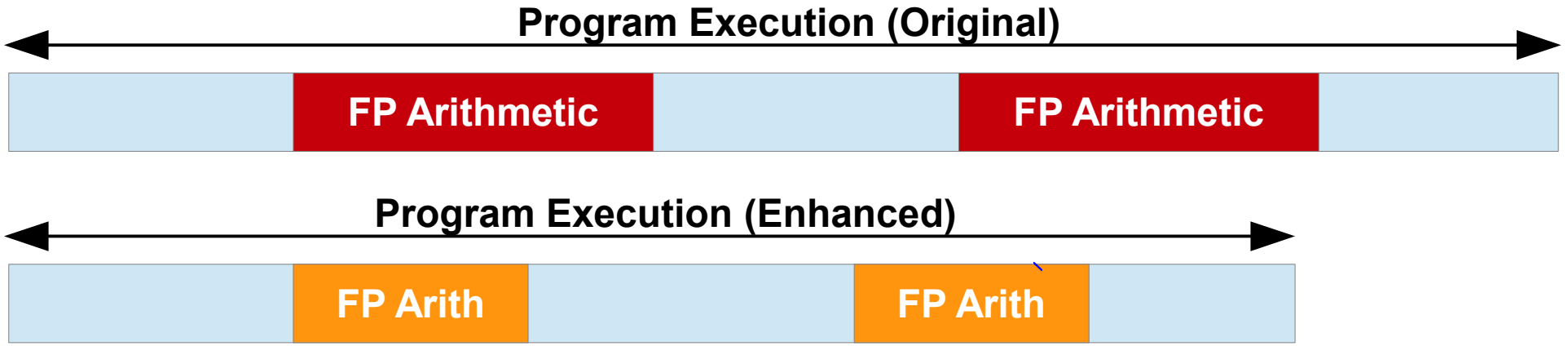
VAdd

for (i:) { } completely parallel

Amdahl's Law

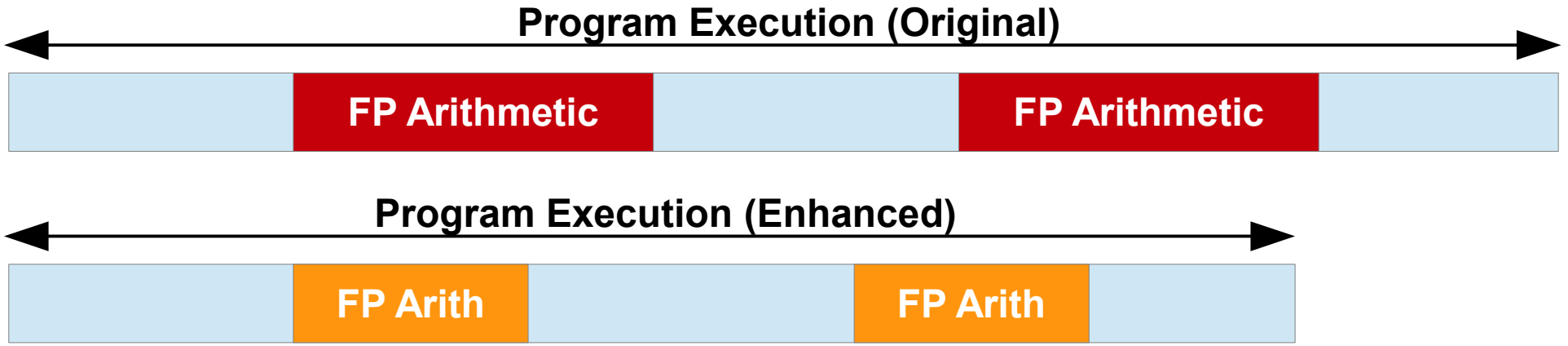

Program Execution (Original)
FP Arithmetic
FP Arithmetic
Program Execution (Enhanced)
FP Arith
FP Arith

Amdahl's Law



- What is the overall speedup by enhancing the performance of a single block?

Amdahl's Law



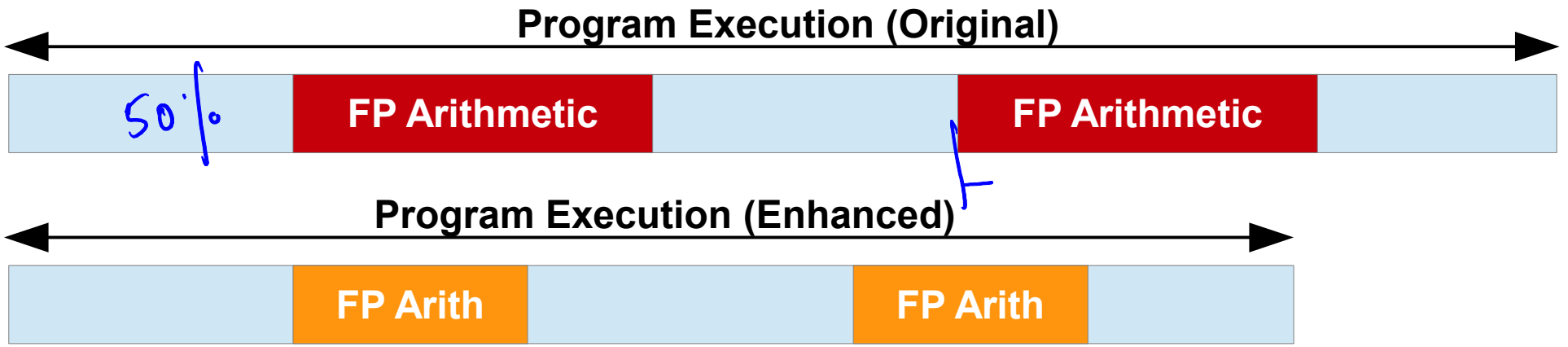
- What is the overall speedup by enhancing the performance of a single block?

$$Speedup_{enhanced} = \frac{Execution\,Time_{original}}{Execution\,Time_{enhancement}}$$

← single threaded

← not

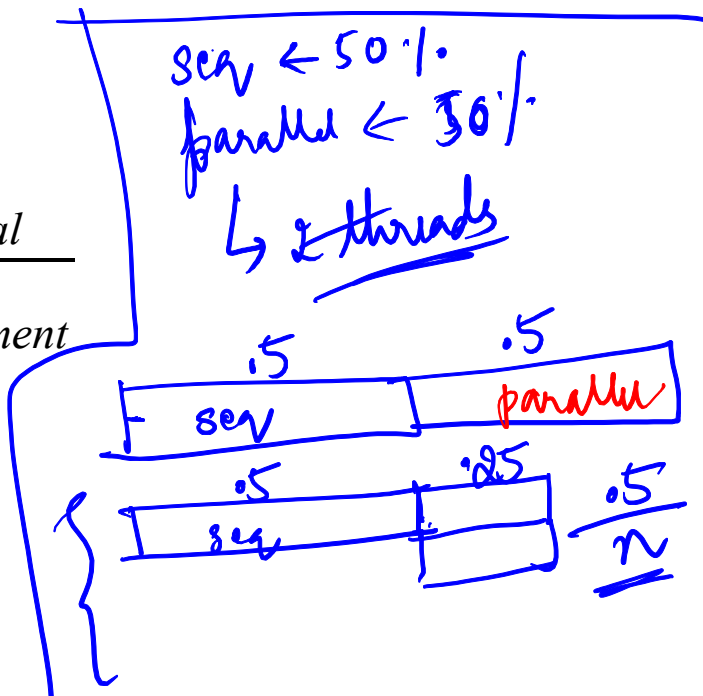
Amdahl's Law



- What is the overall speedup by enhancing the performance of a single block?

$$Speedup_{enhanced} = \frac{Execution\,Time_{original}}{Execution\,Time_{enhancement}}$$

- $Speedup_{enhanced}$ (always >1)
- $Fraction_{enhanced}$ (always <1)



Amdahl's Law

$$Time_{new} = Time_{old} * \left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)$$

Seq portion

parallel

threads

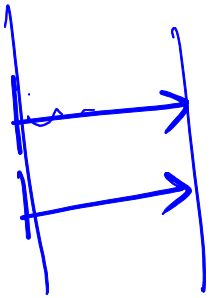
$$T_{new} = \left[(1 - 0.5) + \frac{0.5}{N} \right] T_{old}$$

Amdahl's Law

$$Time_{new} = Time_{old} * \left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)$$

In the example, FP arithmetic was used for 50% of the program. The new FP arithmetic block was 3 times faster than the previous version. What is the new performance number?

Speedup = 1.5



Amdahl's Law

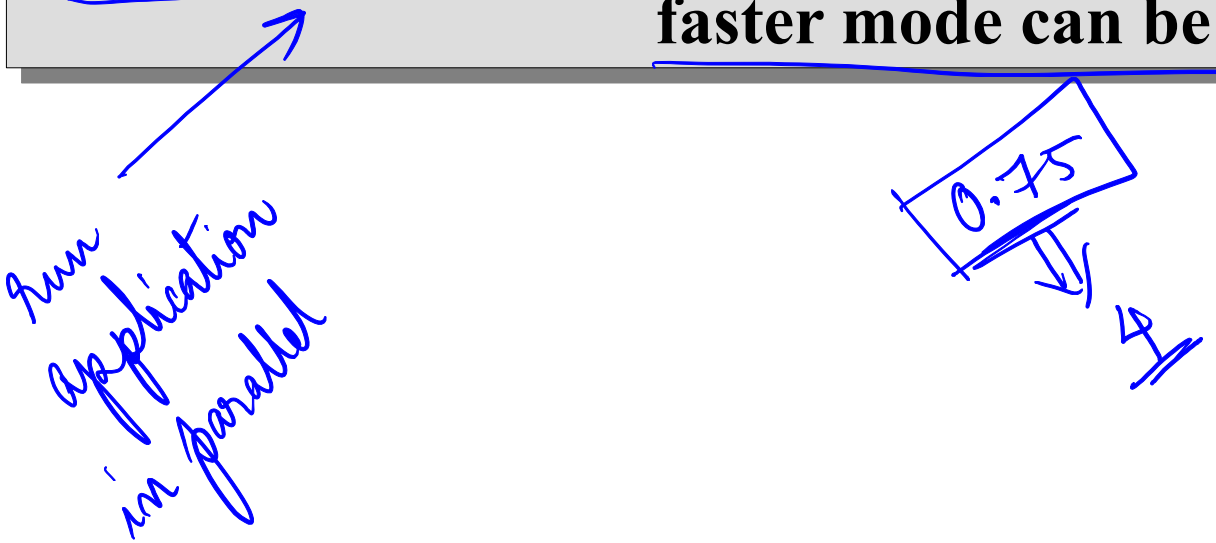
$$Time_{new} = Time_{old} * \left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)$$

In the example, FP arithmetic was used for 50% of the program. The new FP arithmetic block was 3 times faster than the previous version. What is the new performance number?

Objective: Make the program 10 times faster. Say, 25% of the program is waiting in I/O and cannot be enhanced. How much should the speedup of the enhanced computer be?

Amdahl's Law

The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used

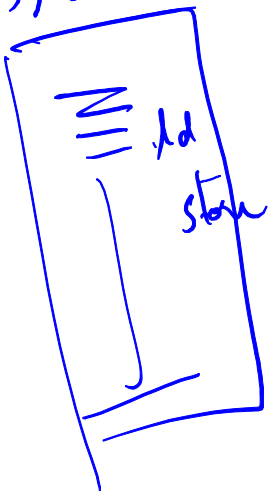


Flynn's Classification

1964/65

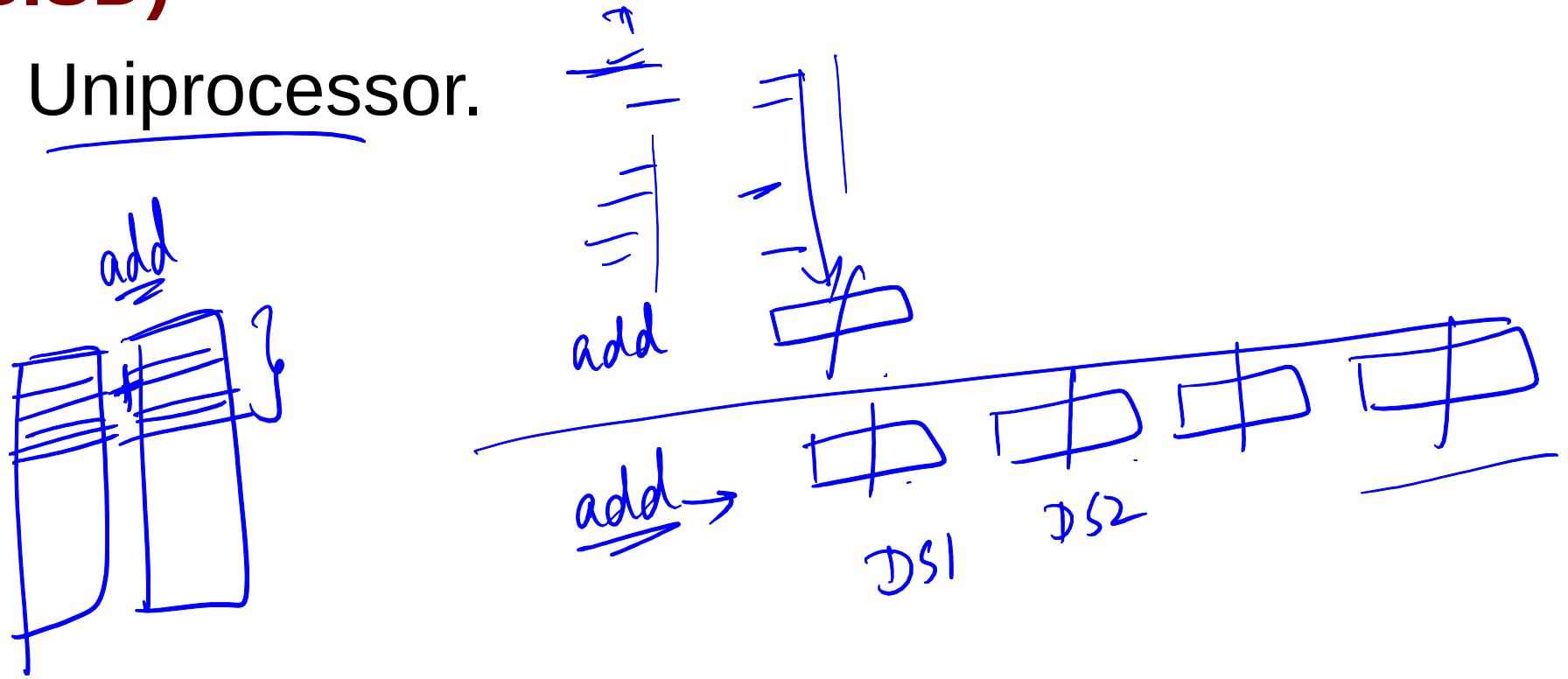
data stream

Instruction stream



Flynn's Classification

- **Single instruction stream, single data stream (SISD)**
  - Uniprocessor.



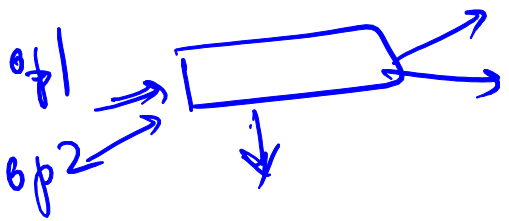
DAXPY

Flynn's Classification

- **Single instruction stream, single data stream (SISD)**
  - Uniprocessor.
- **Single instruction stream, multiple data streams (SIMD)**
  - Data-level parallelism, Subword Parallelism
  - Applying same operations to multiple items of data in parallel
  - Eg. Multimedia extensions, Vector architectures
  - Gaming, 3D, real-time VR, rendering, ...

Flynn's Classification

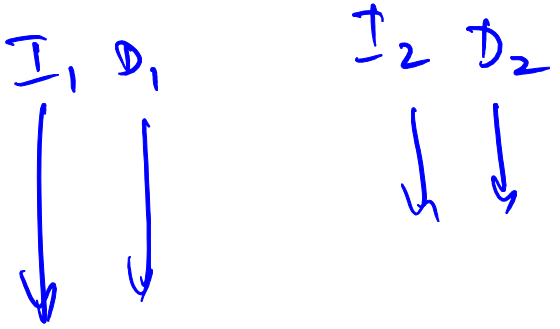
- **Multiple instruction streams, single data stream (MISD)**



This slide is intentionally left blank

Flynn's Classification

- **Multiple instruction streams, multiple data streams (MIMD)**
  - Thread-level parallelism



SIMD

add $R_1, R_2 R_3$

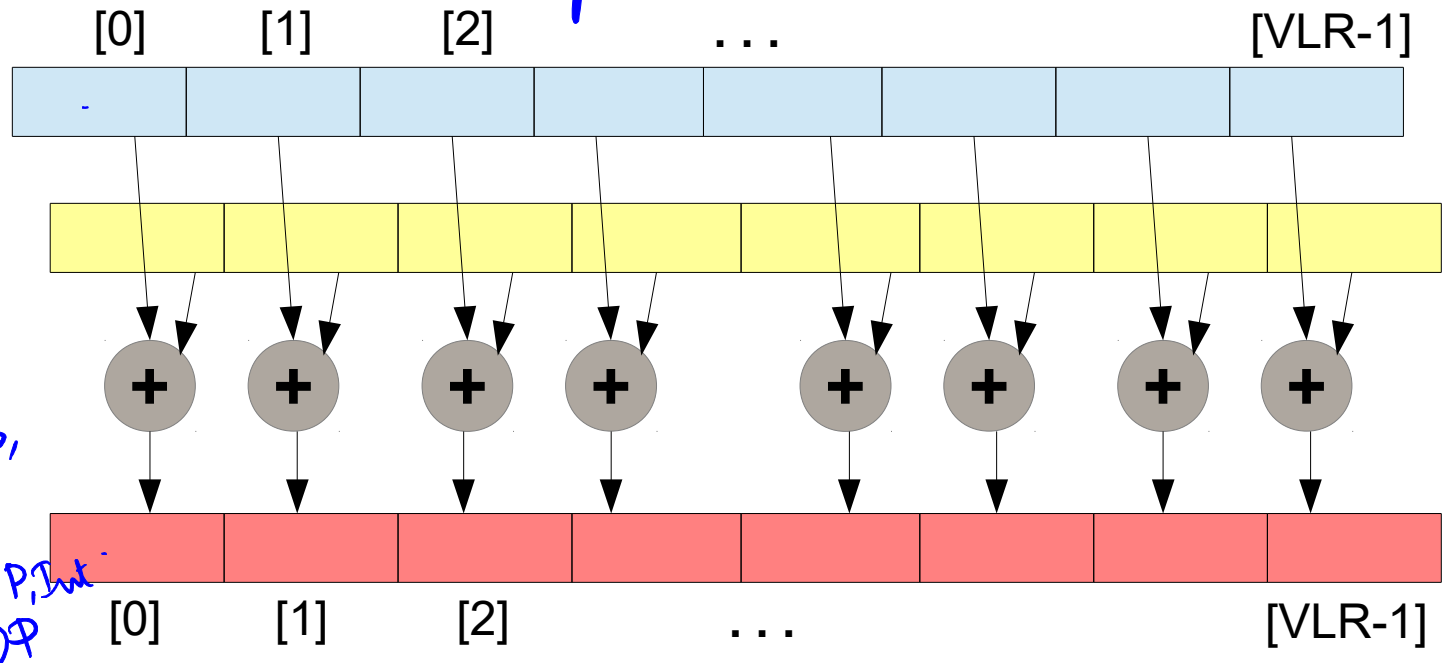
vector

SIMD Instructions

ADDVV V2, V0, V1

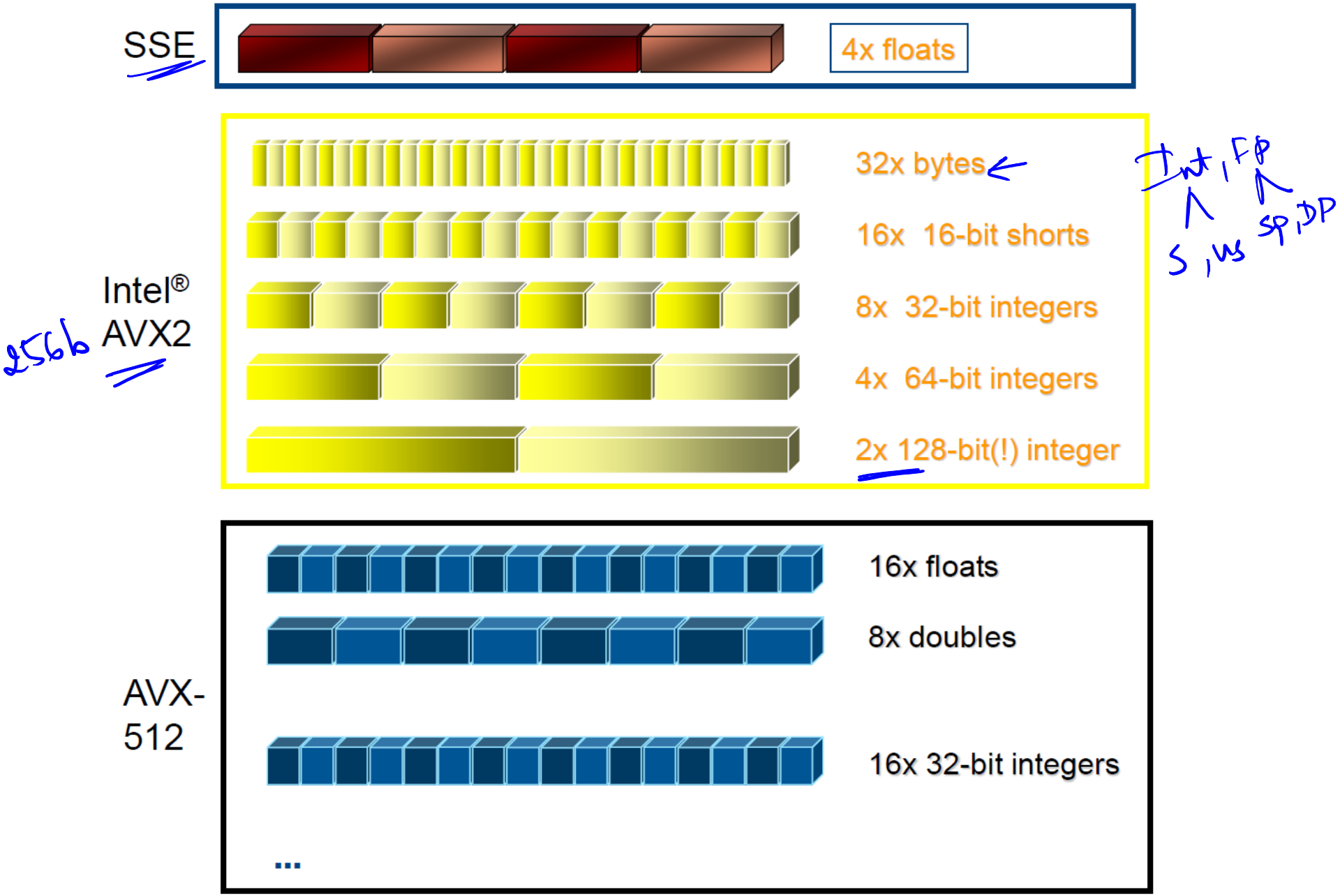
SIMD Instructions

ADDVV V2, V0, V1

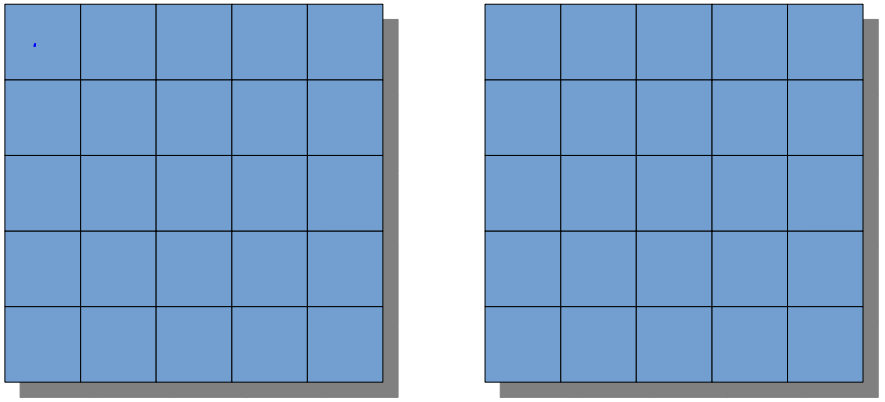


Vector Length Register **VLR**

MMX, SSE, AVX,

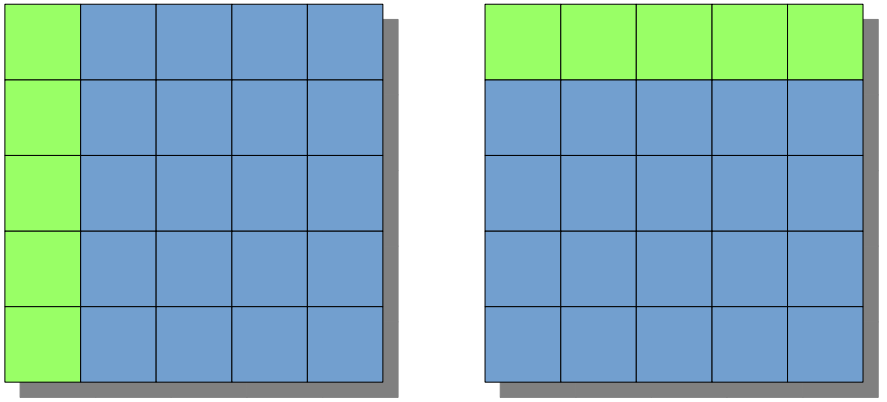

SSE
4x floats
Intel®
AVX2
32x bytes
16x 16-bit shorts
8x 32-bit integers
4x 64-bit integers
2x 128-bit(!) integer
AVX-
512
16x floats
8x doubles
16x 32-bit integers
...

DGEMM – Access Pattern




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

DGEMM – Access Pattern




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

Matrix Multiply

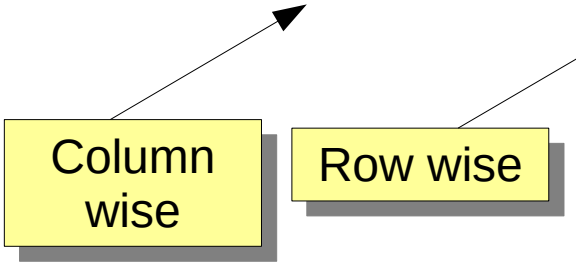
```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {



7.     for(int k = 0; k < n; k++ )



10.   }
11. }
```

Matrix Multiply

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {

7.     for(int k = 0; k < n; k++ )
8.      cij += A[i+k*n] * B[k+j*n];
                                    /* cij += A[i][k]*B[k][j] */

10.   }
11. }
```

Matrix Multiply

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {

7.     for(int k = 0; k < n; k++ )
8.      cij += A[i+k*n] * B[k+j*n];
                          /* cij += A[i][k]*B[k][j] */

10.   }
11. }
```

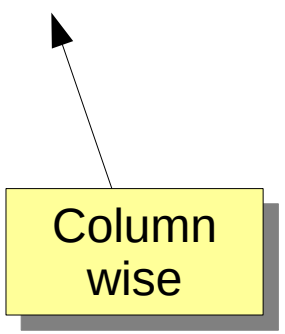


Matrix Multiply

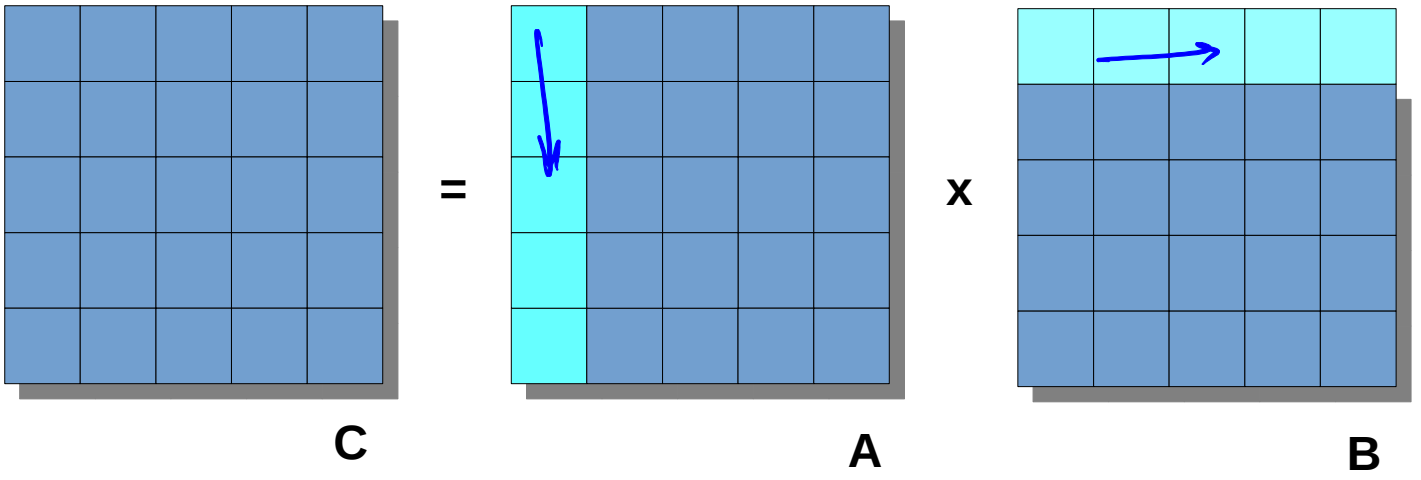
```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {
6.     double cij = C[i+j*n]; /* cij = C[i][j] */
7.     for(int k = 0; k < n; k++ )
8.      cij += A[i+k*n] * B[k+j*n];
                                /* cij += A[i][k]*B[k][j] */
9.     C[i+j*n] = cij; /* C[i][j] = cij */
10.   }
11. }
```

Matrix Multiply

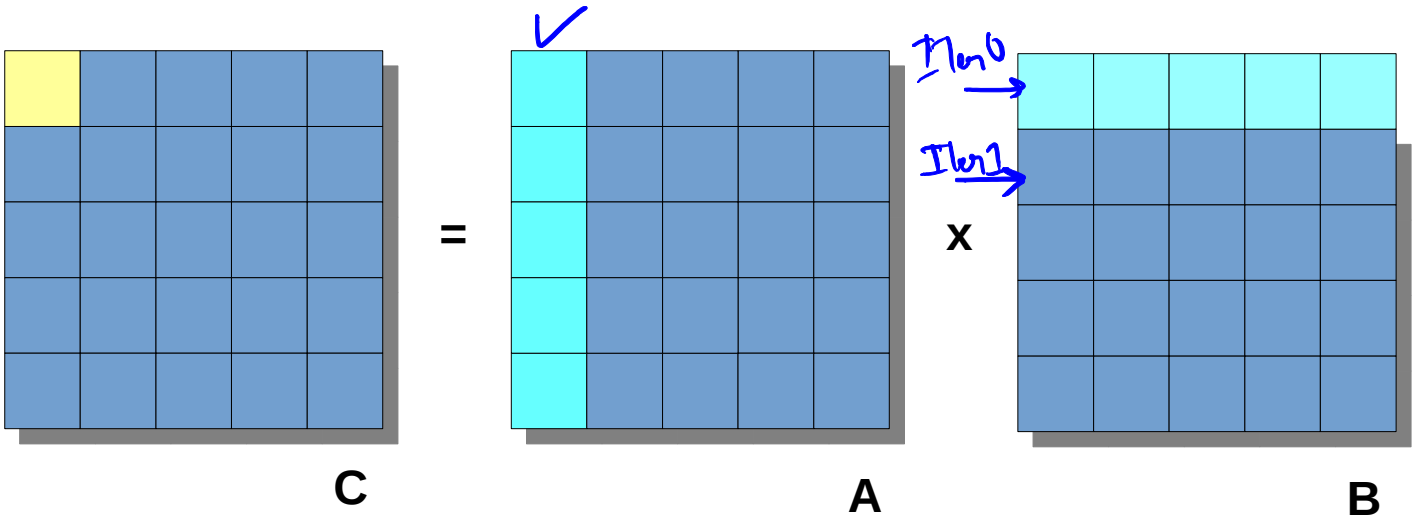
```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {
6.     double cij = C[i+j*n]; /* cij = C[i][j] */
7.     for(int k = 0; k < n; k++ )
8.      cij += A[i+k*n] * B[k+j*n];
                          /* cij += A[i][k]*B[k][j] */
9.     C[i+j*n] = cij; /* C[i][j] = cij */
10.   }
11. }
```



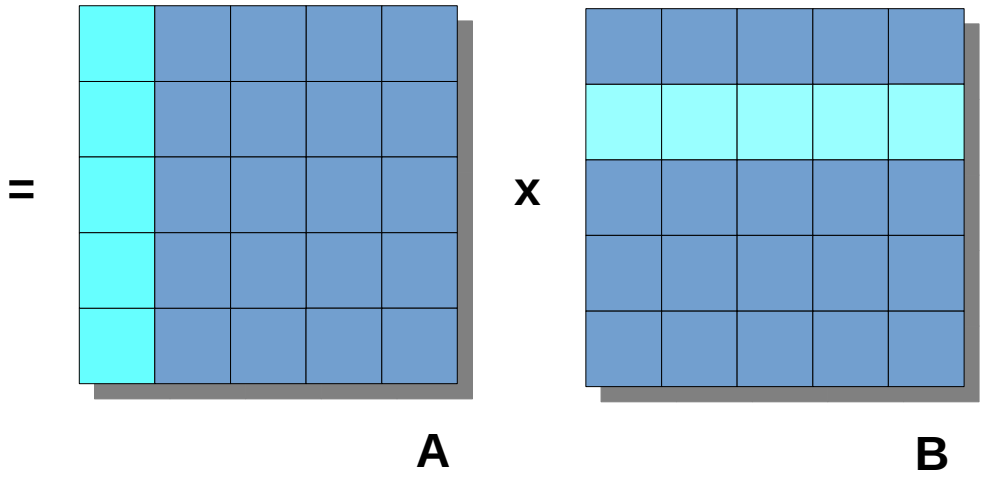
Matrix Multiply – Basic Version



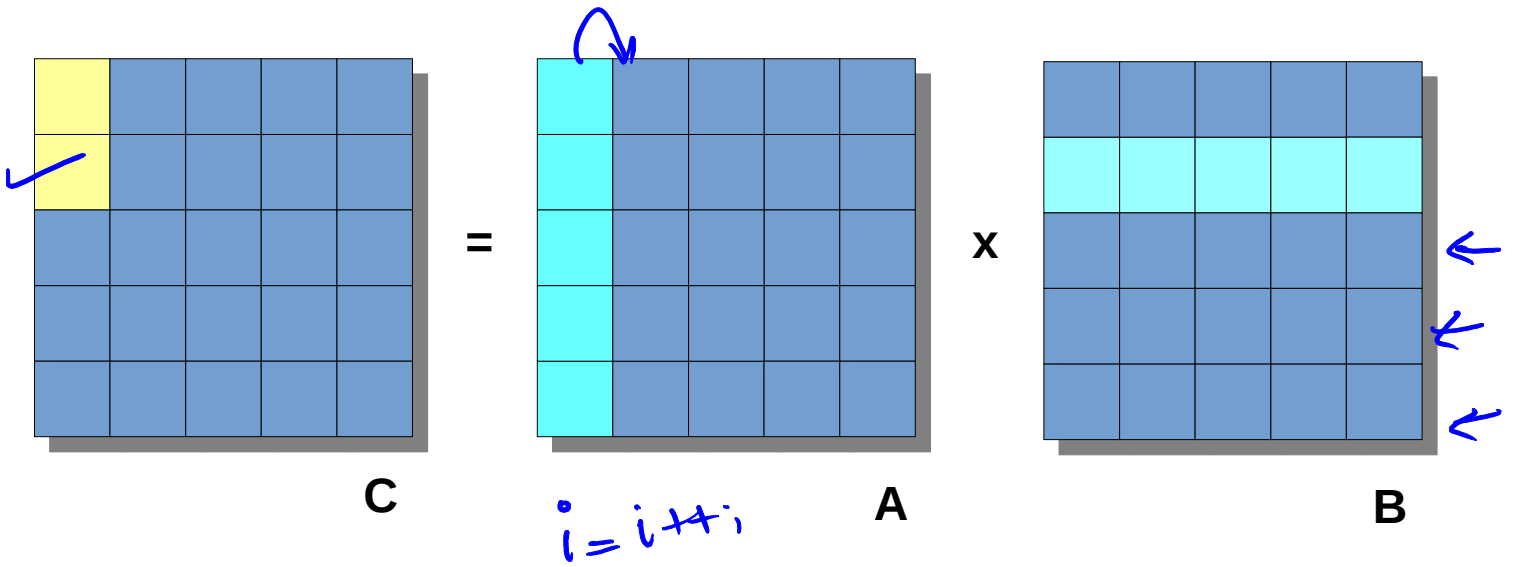
Matrix Multiply – Basic Version



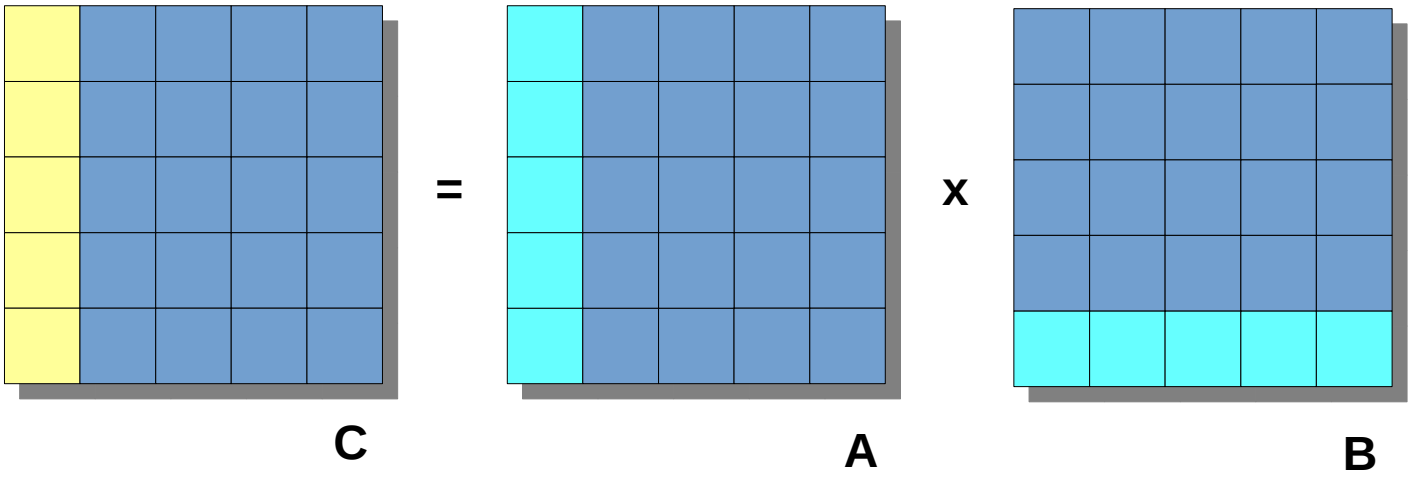
Matrix Multiply – Basic Version



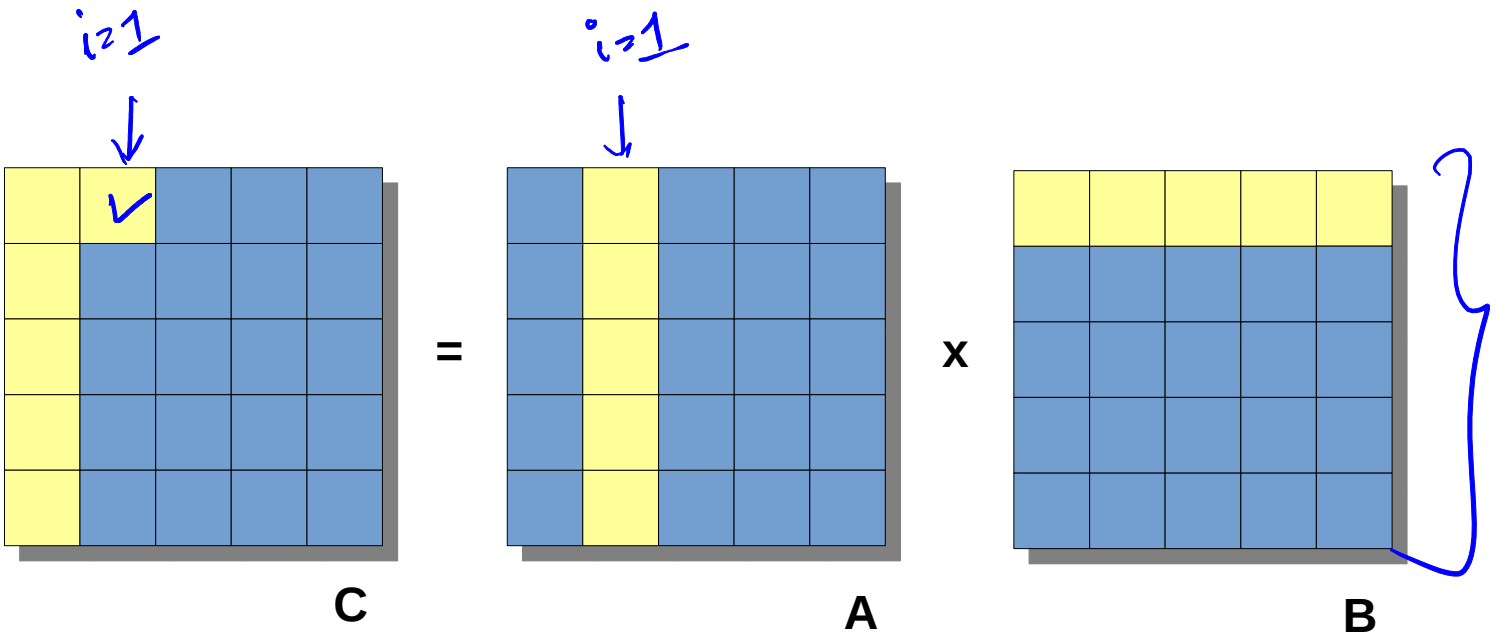
Matrix Multiply – Basic Version



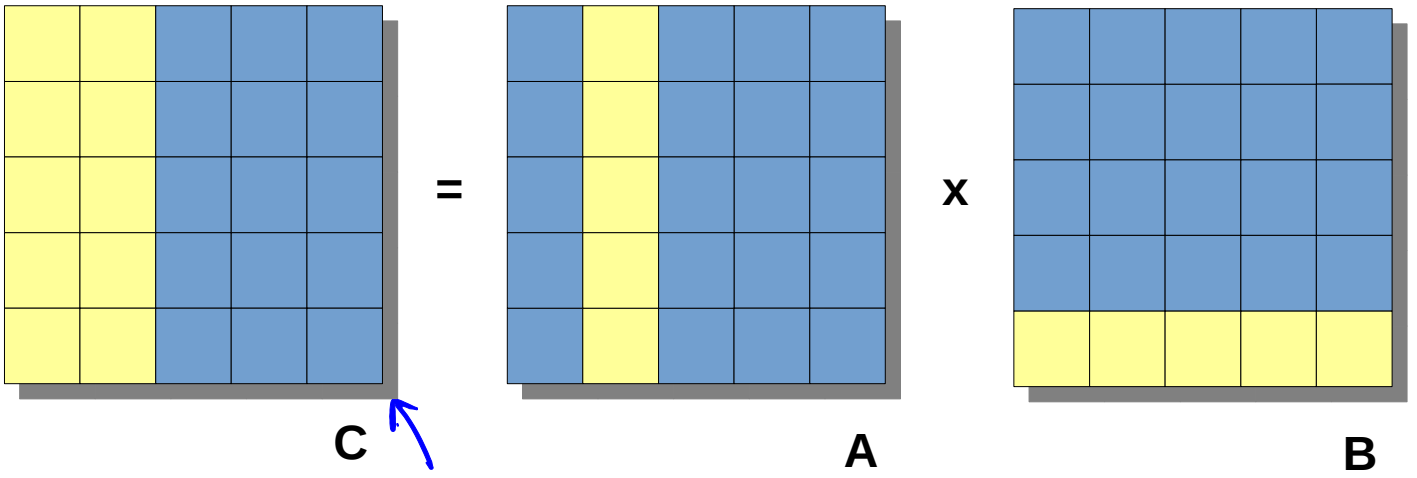
Matrix Multiply – Basic Version



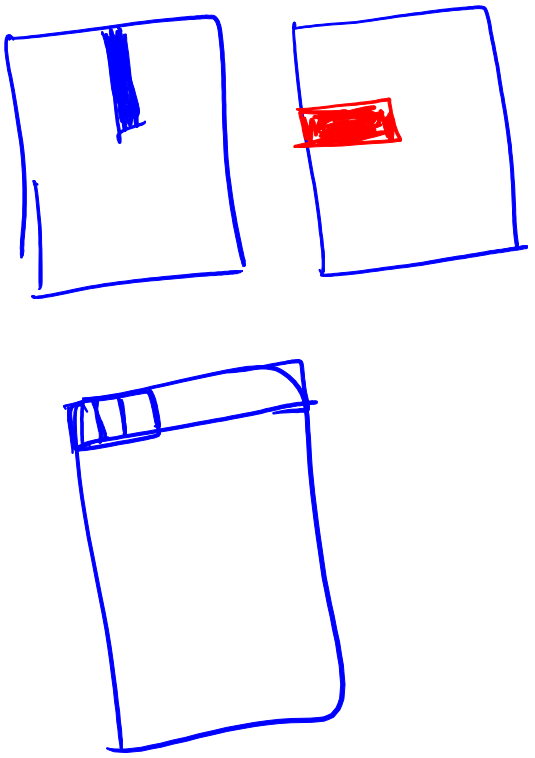
Matrix Multiply – Basic Version



Matrix Multiply – Basic Version



Matrix Multiply – SIMD Version



addv.

load.us.i *a, v1

store v.

a | a+1 | a+2 | a+3

load.us.i *a, v1, 16

a | a+16 | a+32 | a+

gather

scatter

Store

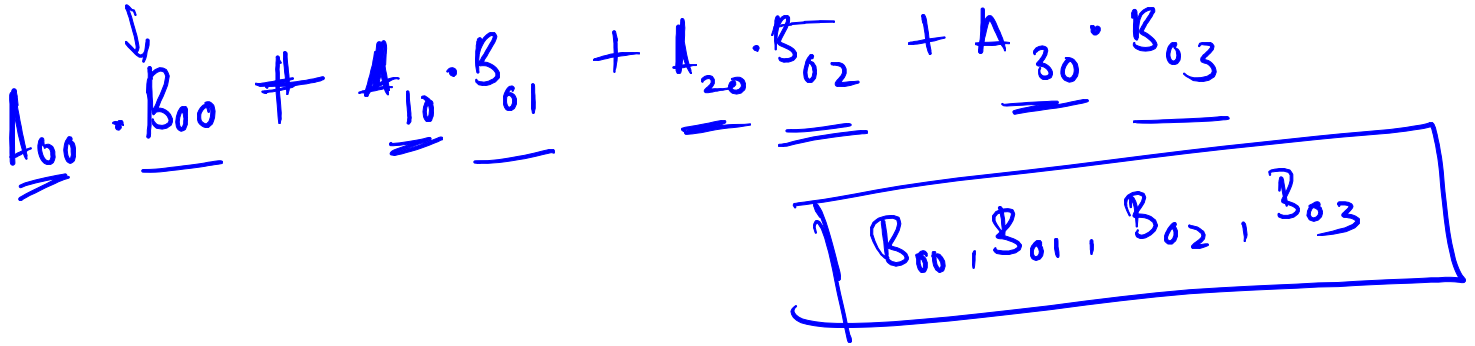
Vector

Seymour

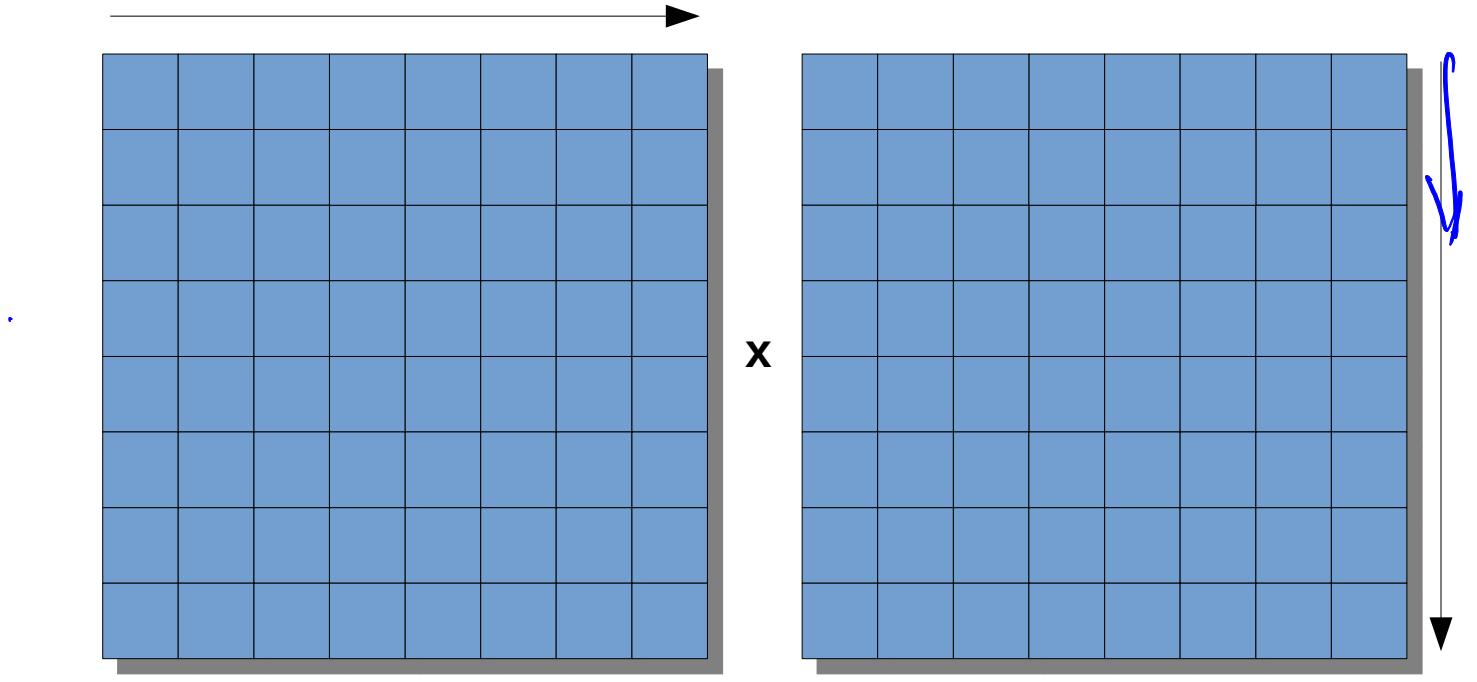
Cray

Matrix Multiply – SIMD Version

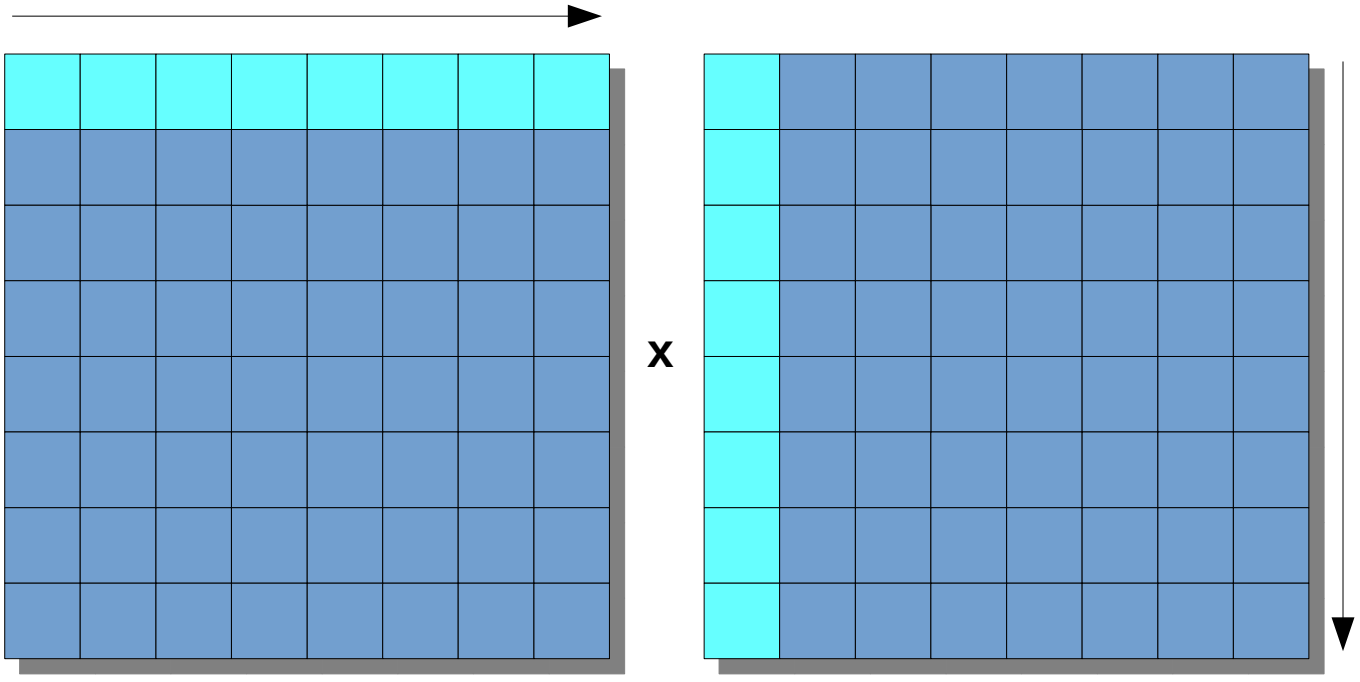
- Which are the SIMD operations?
-



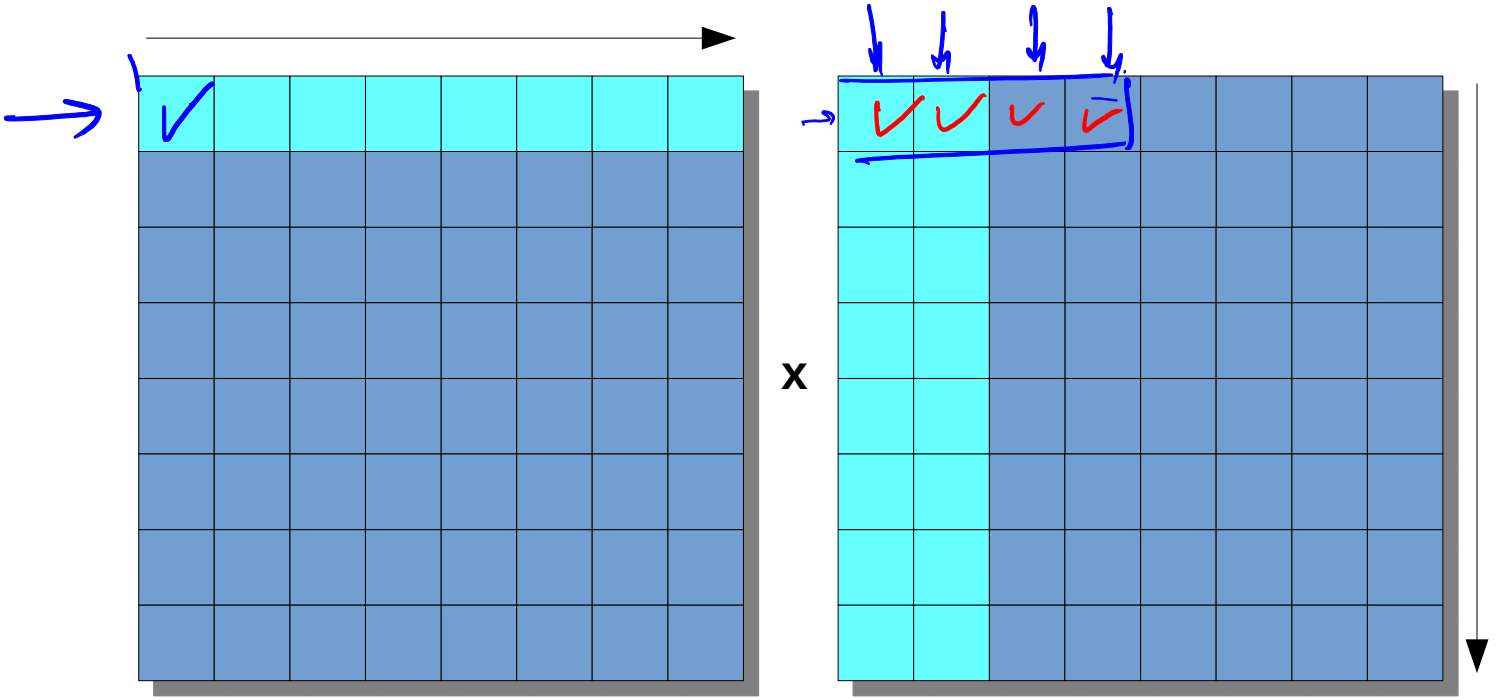
Matrix Multiply – SIMD Version

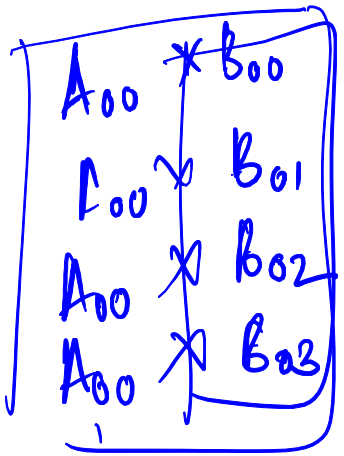


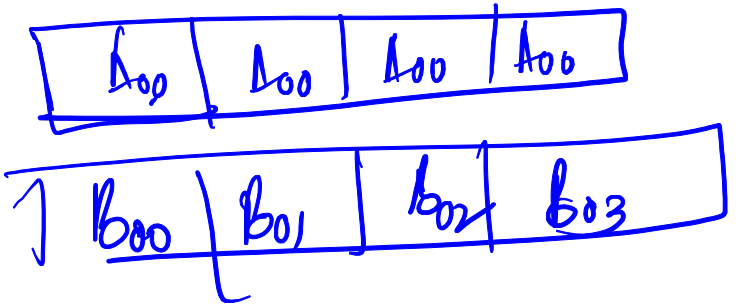
Matrix Multiply – SIMD Version



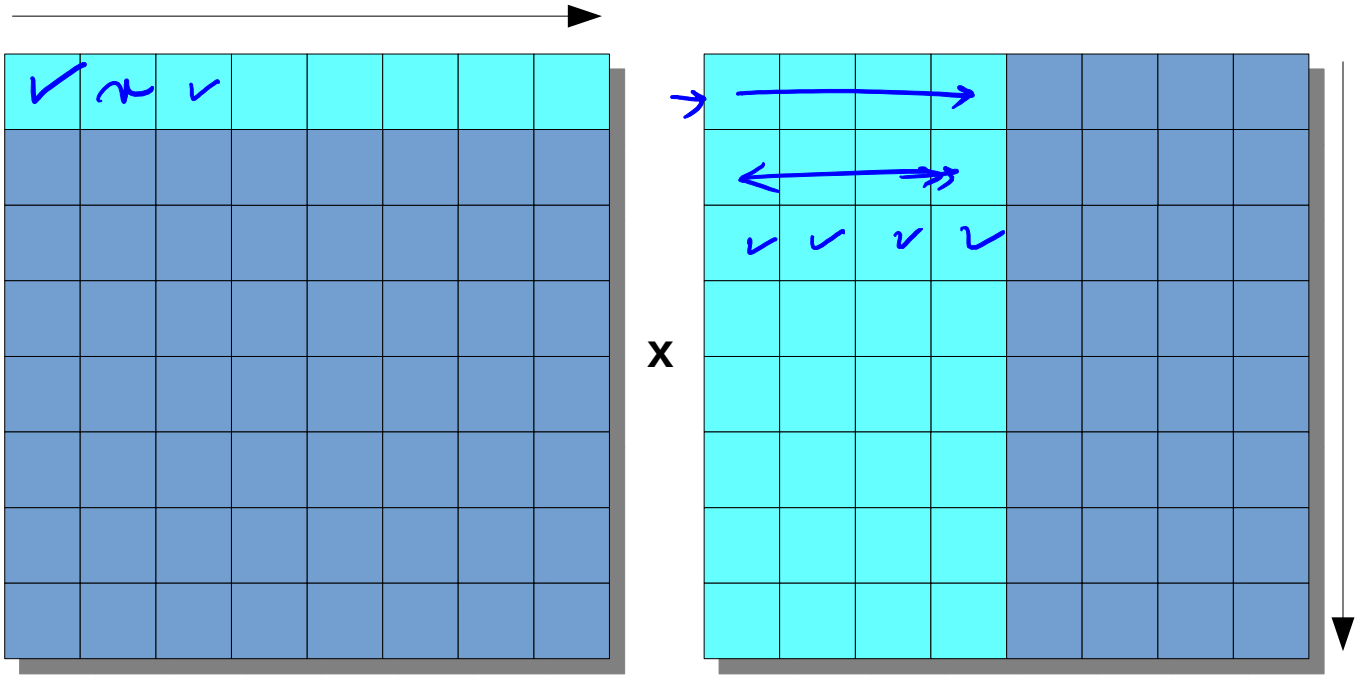
Matrix Multiply – SIMD Version



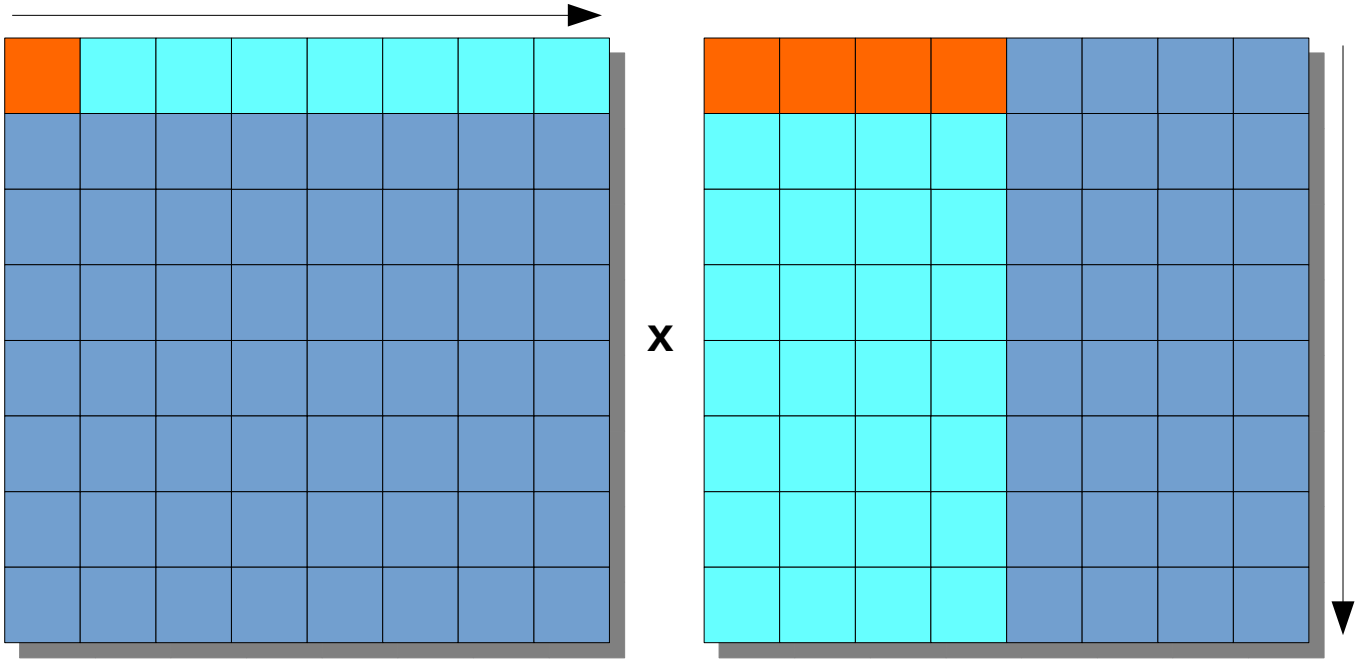




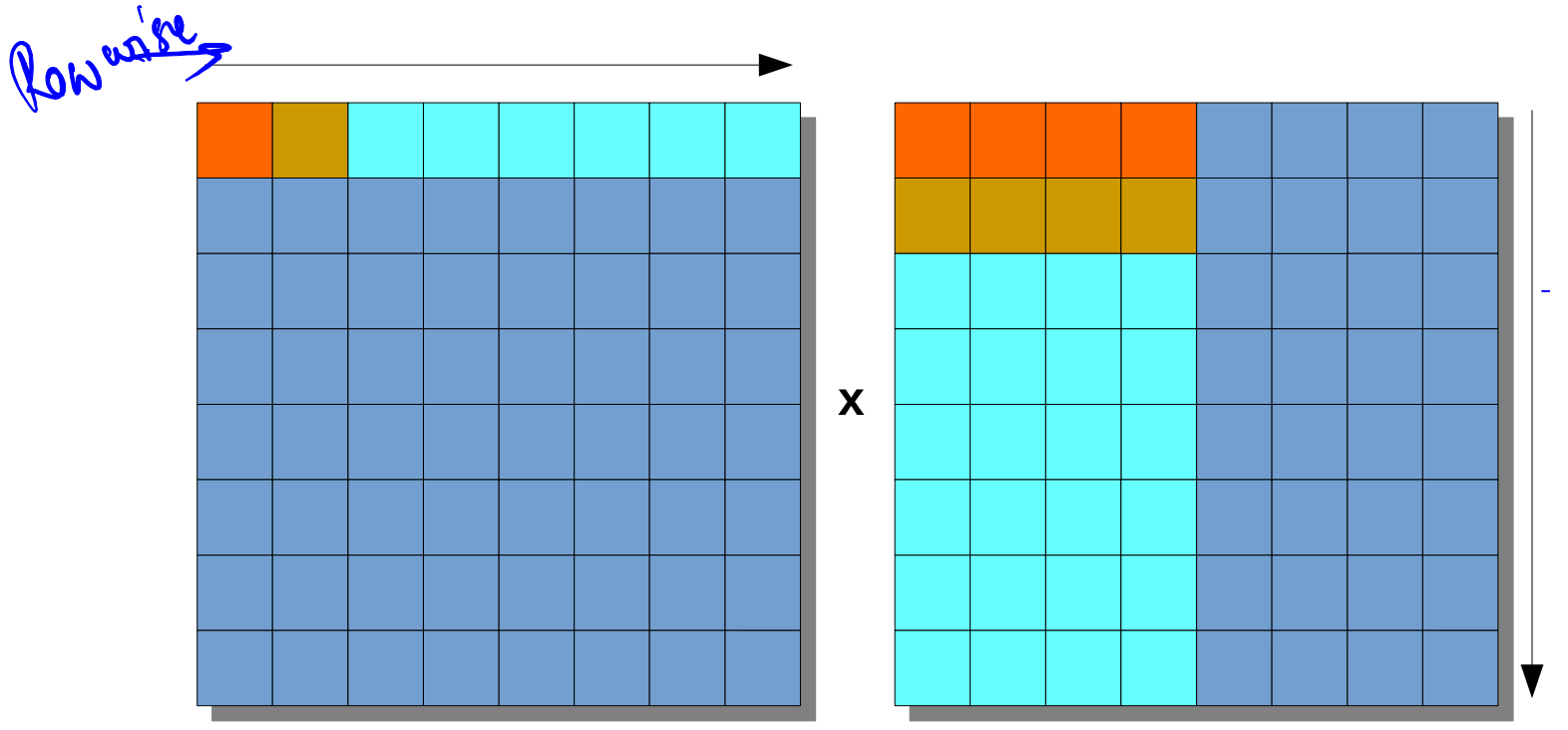
Matrix Multiply – SIMD Version



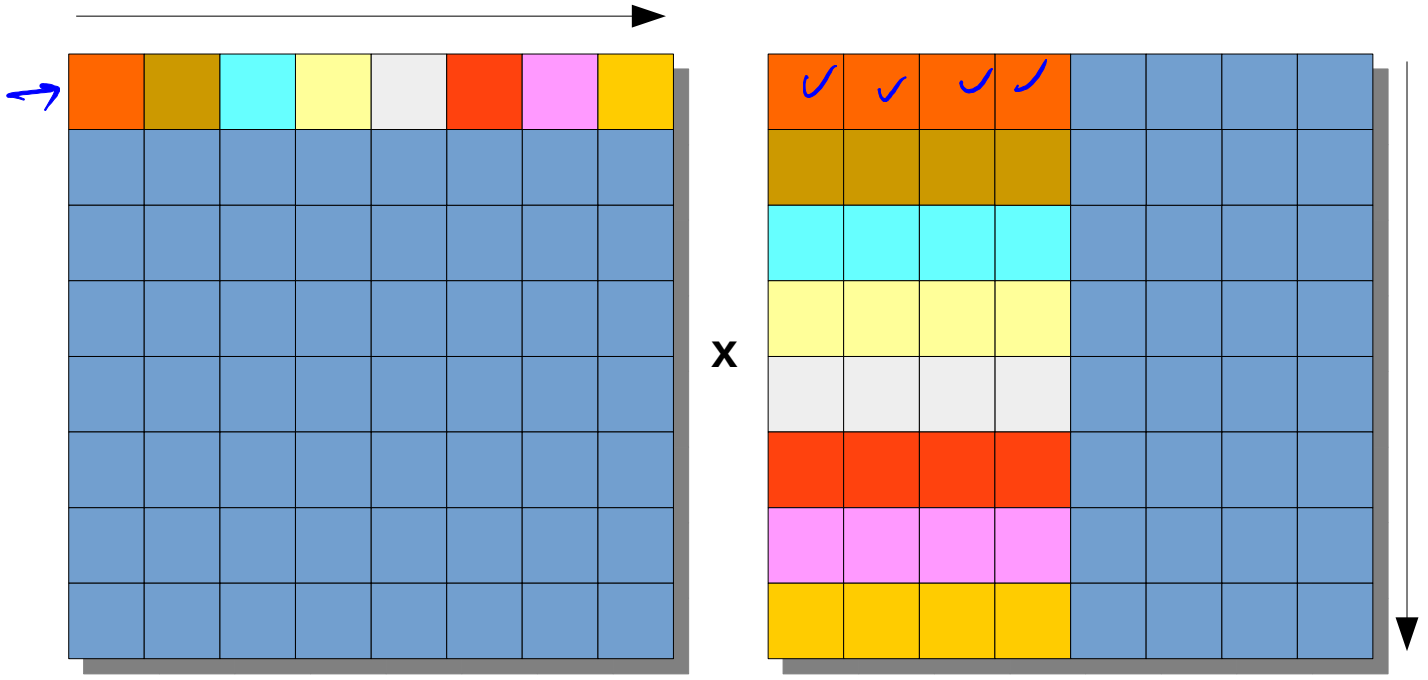
Matrix Multiply – SIMD Version



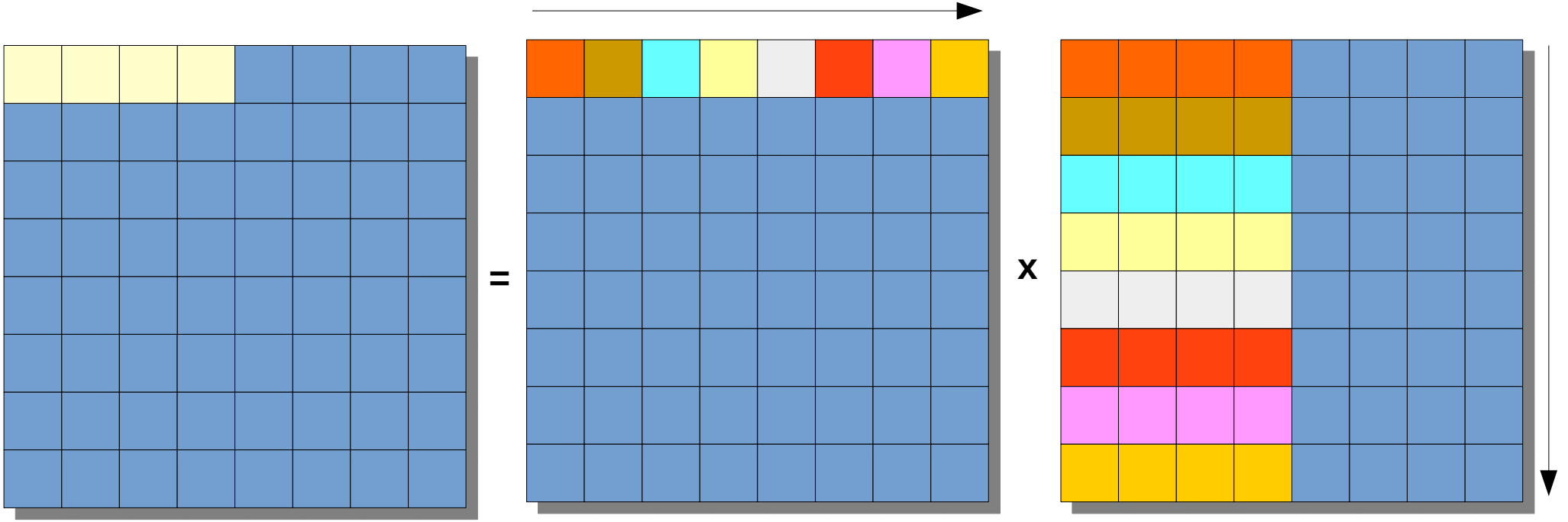
Matrix Multiply – SIMD Version



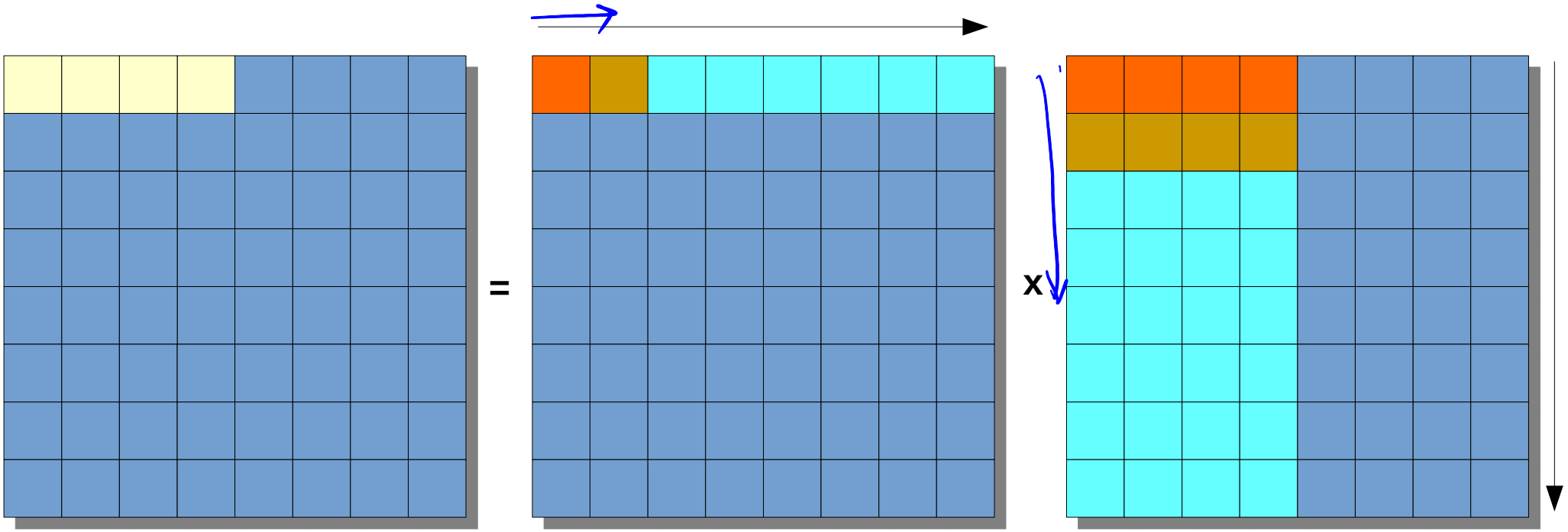
Matrix Multiply – SIMD Version



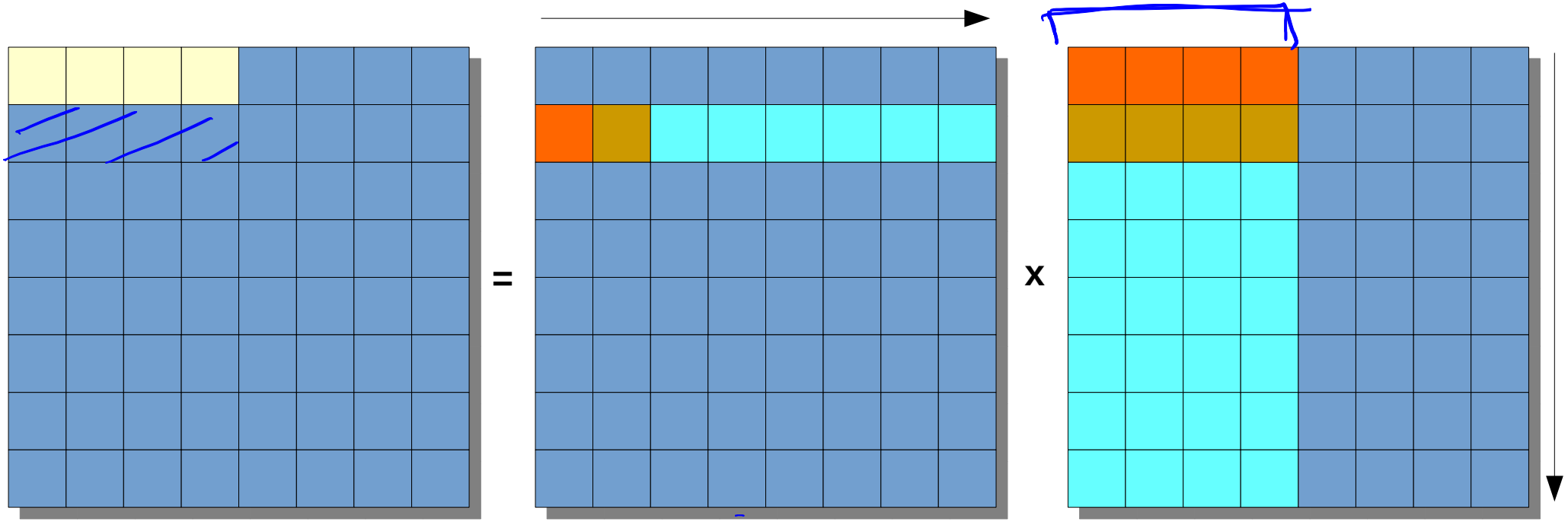
Matrix Multiply – SIMD Version



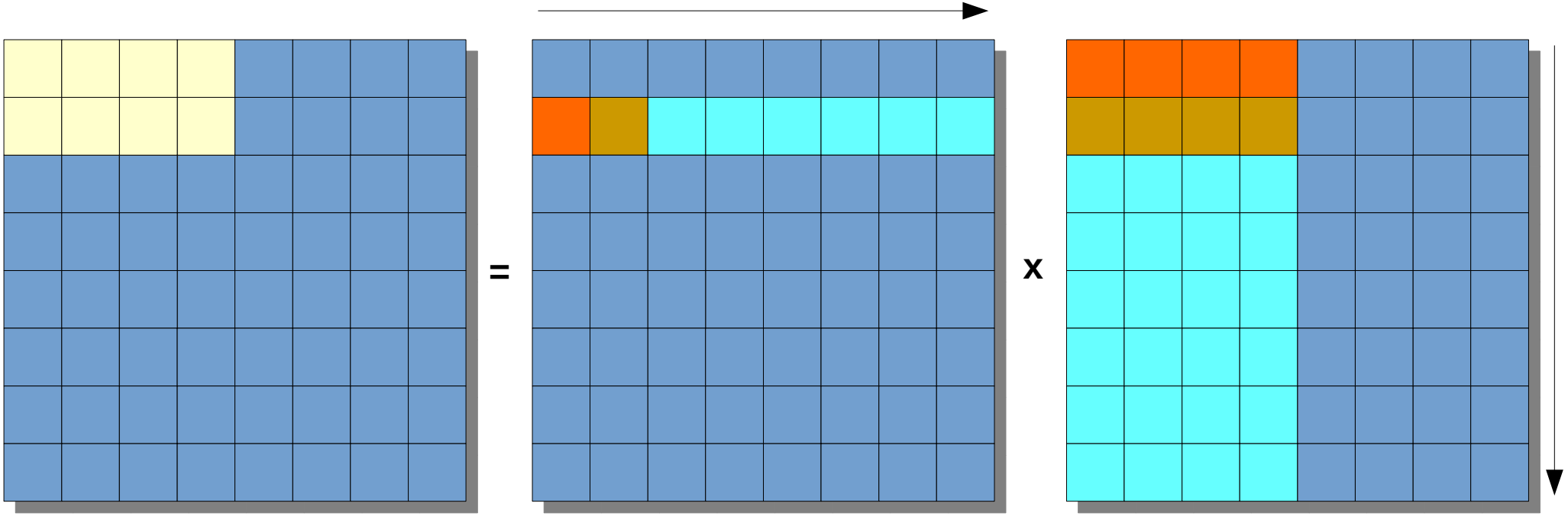
Matrix Multiply – SIMD Version



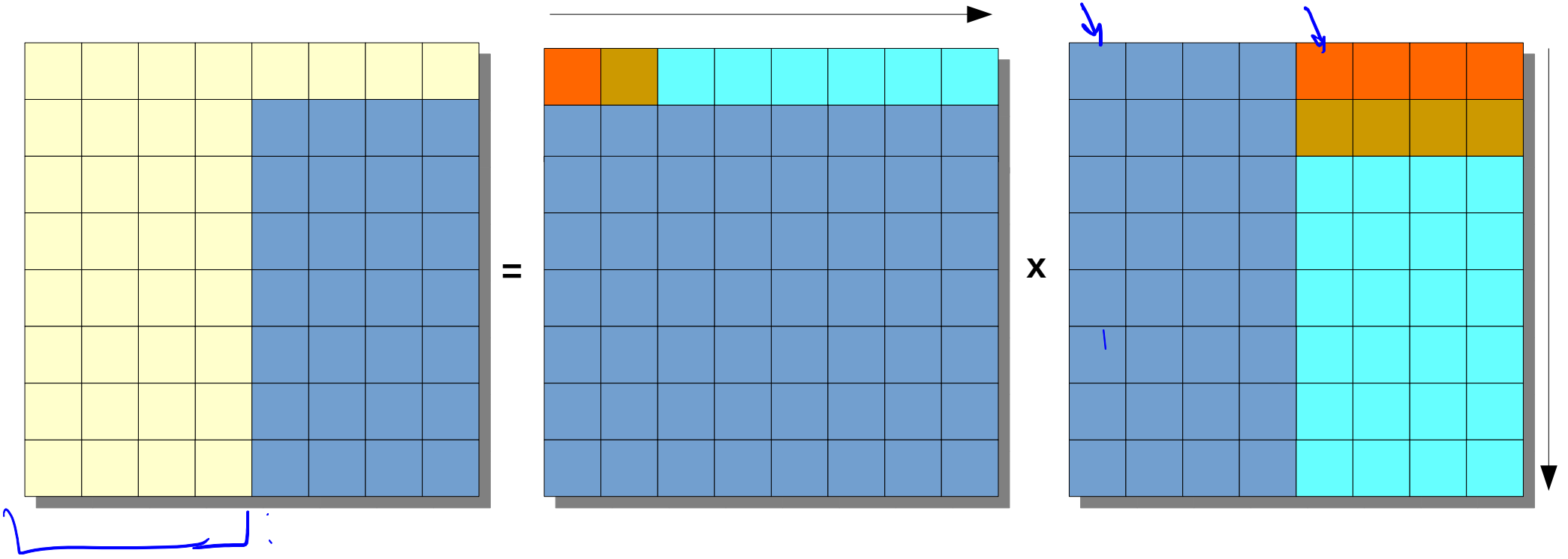
Matrix Multiply – SIMD Version



Matrix Multiply – SIMD Version



Matrix Multiply – SIMD Version



SSE Instructions

```
extern __m256d _mm256_load_pd(double const *a);
```

- *a: pointer to a memory location that can hold constant float64 values; the address must be 32-byte aligned
- moves packed double-precision floating point values from aligned memory location to a destination vector.
- returns a 256-bit vector with float64 values.

SSE Instructions

```
extern __m256d _mm256_broadcast_sd(double const *a);
```

- *a: pointer to a memory location that can hold constant scalar float64 values
- Loads scalar double-precision floating-point values from the specified address a, and broadcasts it to all four elements in the destination vector
- returns a 256-bit vector with 4 identical float64 scalar values.

SSE Instructions

```
extern __m256d _mm256_mul_pd(__m256d m1, __m256d m2);
```

- m1,m2: float64 input vectors
- adds m1 and m2 float64 vectors
- returns a float64 vector

SSE Instructions

```
extern __m256d _mm256_add_pd(__m256d m1, __m256d m2);
```

- m1,m2: float64 input vectors
- adds m1 and m2 float64 vectors
- returns a float64 vector

SSE Instructions

```
extern void _mm256_store_pd(double *a, __m256d b);
```

- *a: pointer to a memory location that can hold double-precision floating point (float64) values; the address must be 32-byte aligned
- b: float64 vector
- Performs a store. Moves the packed float64 values from vector b to a 256-bit aligned memory location, pointed to by a.

Matrix Multiply

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B,
     double* C)
3. {
```

Matrix Multiply

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B,
   double* C)
3. {
4.   for ( int i = 0; i < n; i+=4 )
```

Matrix Multiply

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B,
   double* C)
3. {
4.   for ( int i = 0; i < n; i+=4 )
5.     for ( int j = 0; j < n; j++ ) {
```

Matrix Multiply

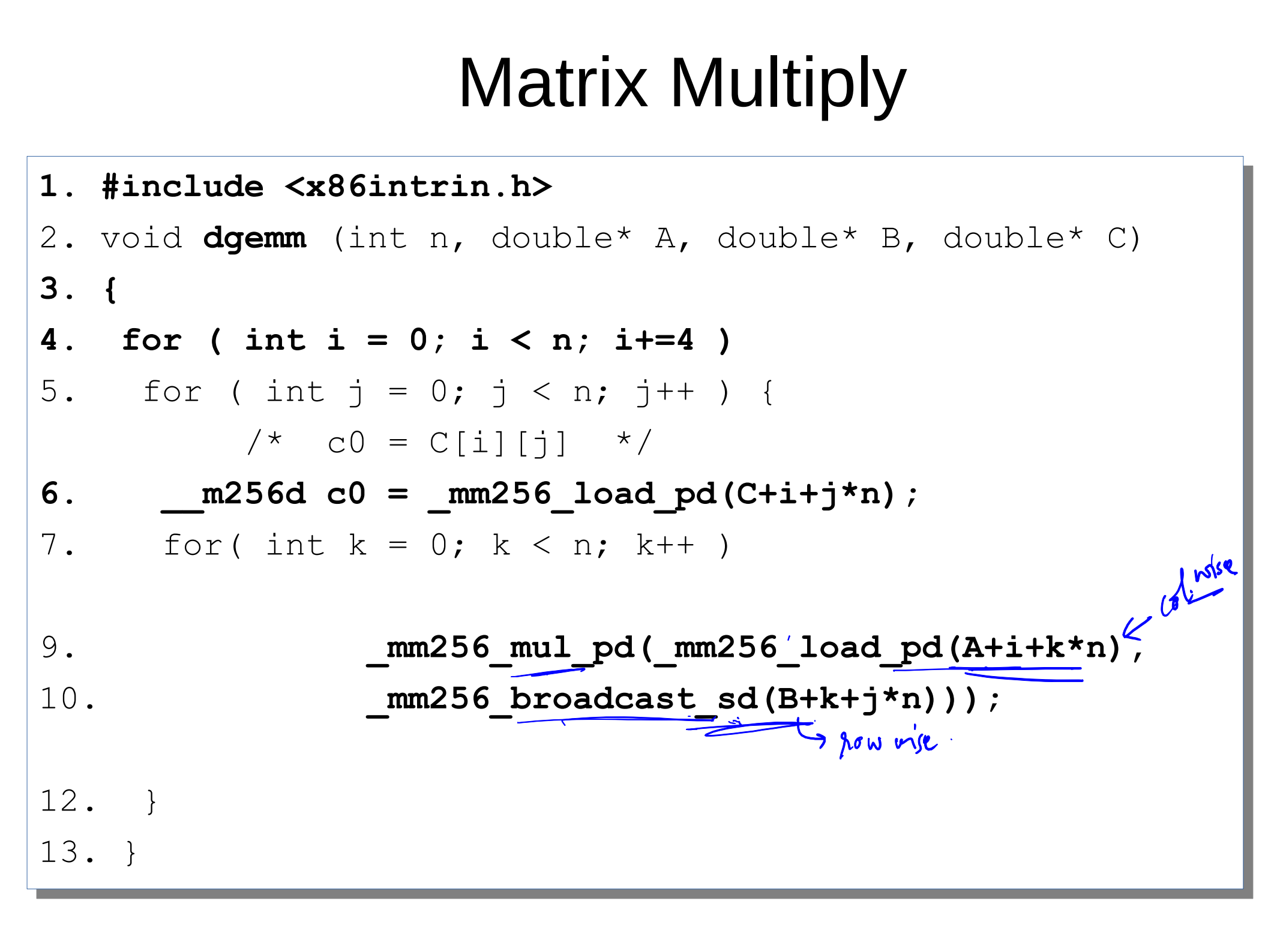
```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double*
   C)
3. {
4.   for ( int i = 0; i < n; i+=4 )
5.     for ( int j = 0; j < n; j++ ) {
           /*  c0 = C[i][j]  */
6.       __m256d c0 = _mm256_load_pd(C+i+j*n);
```

Matrix Multiply

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
         /*  c0 = C[i][j]  */
6.    __m256d c0 = _mm256_load_pd(C+i+j*n);
7.    for( int k = 0; k < n; k++ )




12.  }
13. }
```

Matrix Multiply

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
         /*  c0 = C[i][j]  */
6.    __m256d c0 = _mm256_load_pd(C+i+j*n);
7.    for( int k = 0; k < n; k++ )

9.              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.             _mm256_broadcast_sd(B+k+j*n)));

12.  }
13. }
```



Matrix Multiply

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
         /*  c0 = C[i][j]  */
6.    __m256d c0 = _mm256_load_pd(C+i+j*n);
7.    for( int k = 0; k < n; k++ )
8.    c0 = _mm256_add_pd(c0,
9.               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.              _mm256_broadcast_sd(B+k+j*n)));


12.  }
13. }
```

Matrix Multiply

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
         /*  c0 = C[i][j]  */
6.    __m256d c0 = _mm256_load_pd(C+i+j*n);
7.    for( int k = 0; k < n; k++ )
8.     c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.              _mm256_broadcast_sd(B+k+j*n)));
11.   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.  }
13. }
```