

M4 – Parallelism

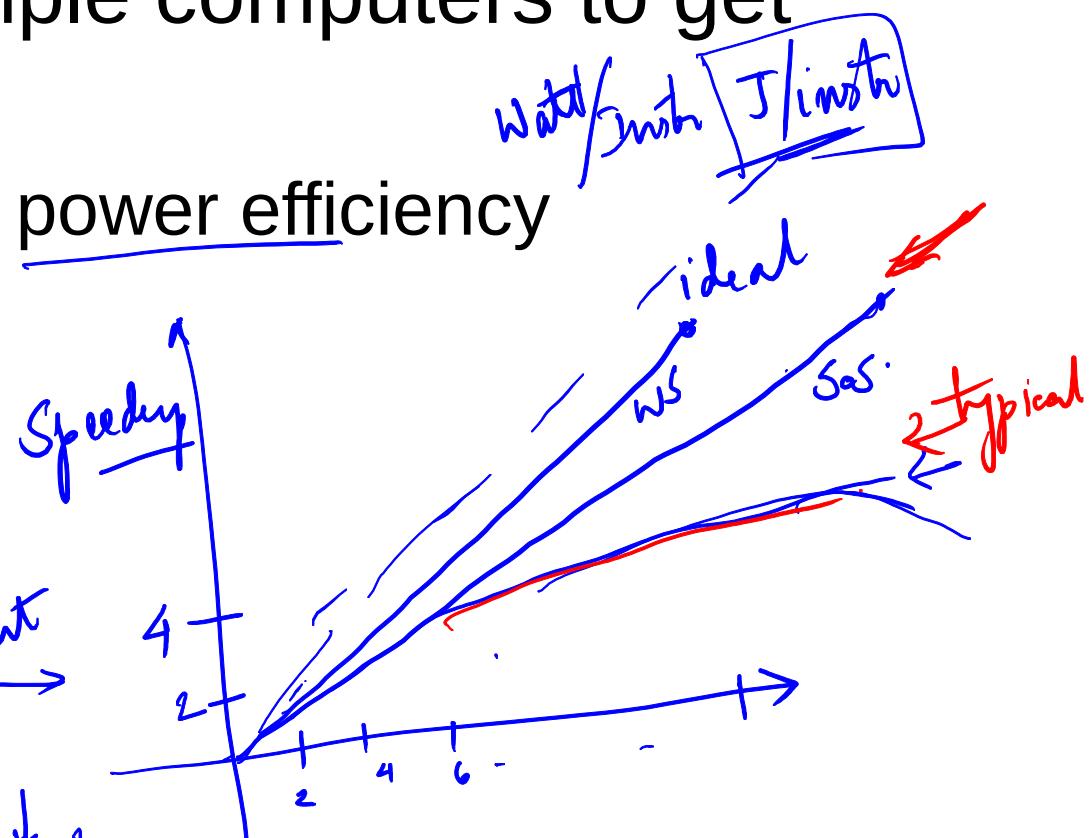
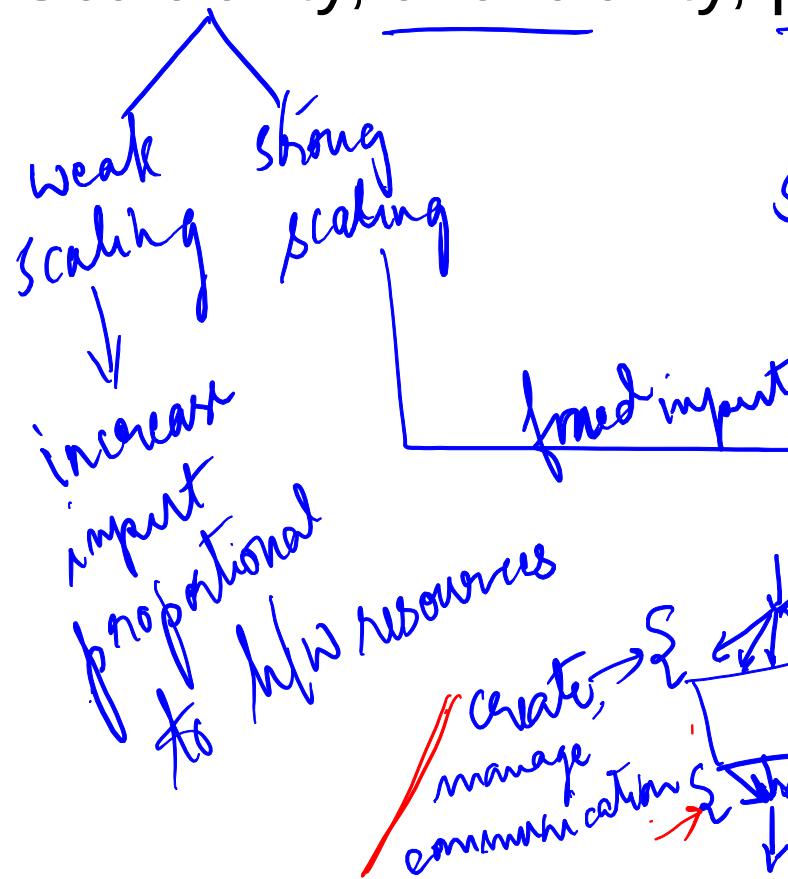
Outline

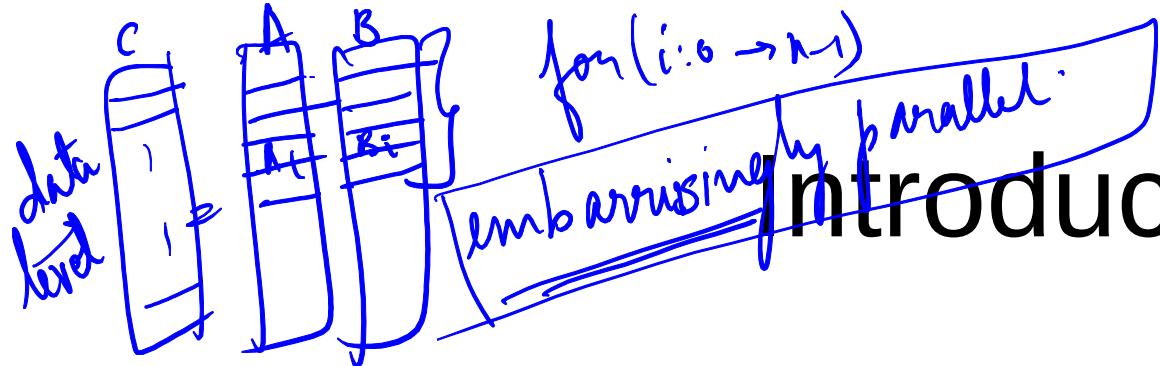
- Parallelism
- Flynn's classification
- Vector Processing
 - Subword Parallelism
- Synchronization
 - ISA
- Symmetric Multiprocessors, Distributed Memory Machines
 - Shared Memory Multiprocessing, Message Passing
- Cache coherence

100% noc

Introduction

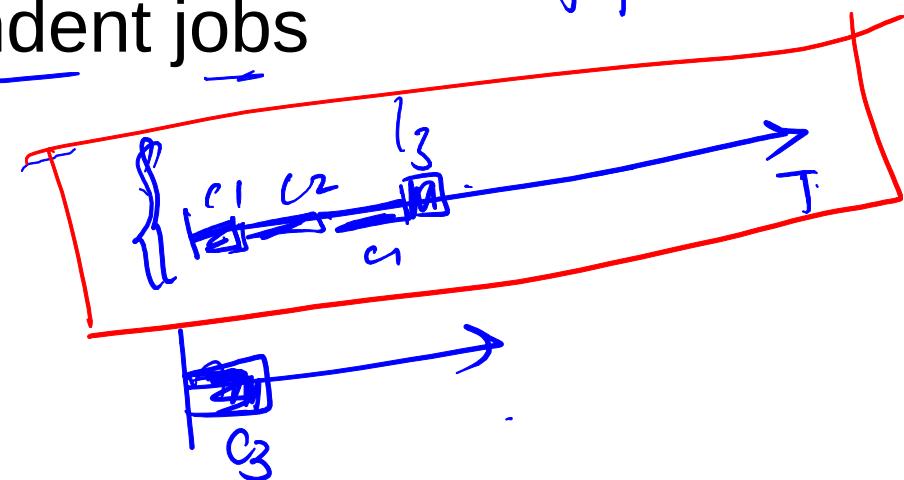
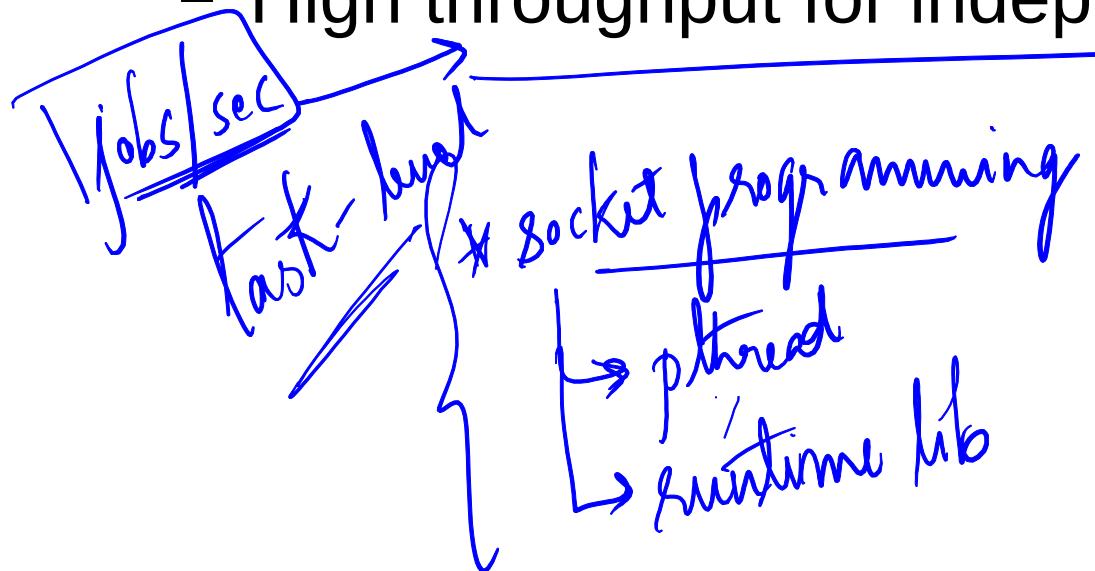
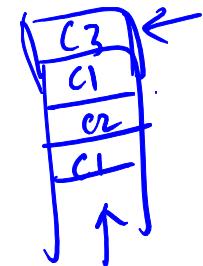
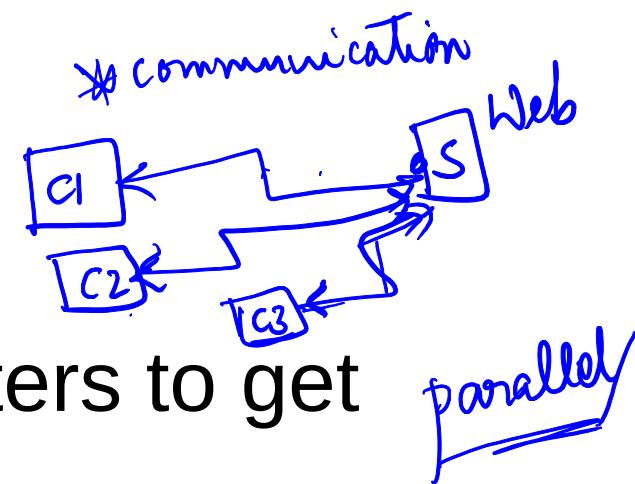
- Goal: connecting multiple computers to get higher performance
 - Scalability, availability, power efficiency





Introduction

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



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^{runt}

- Data intensive applications

- Natural parallelism in large scientific applications

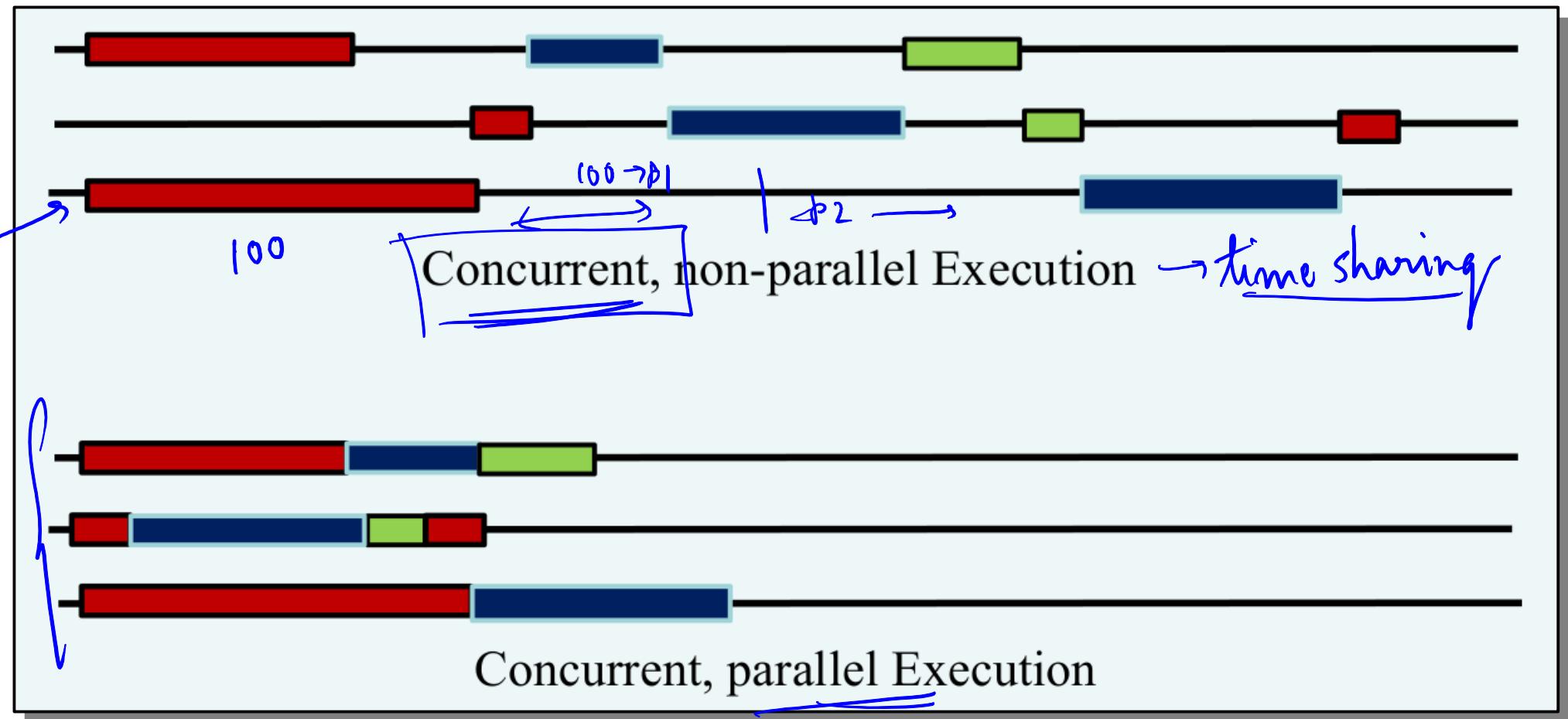
- More return on investments by replicating current designs



Terms

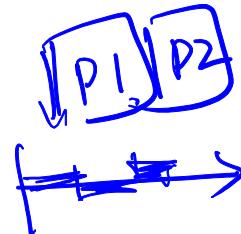
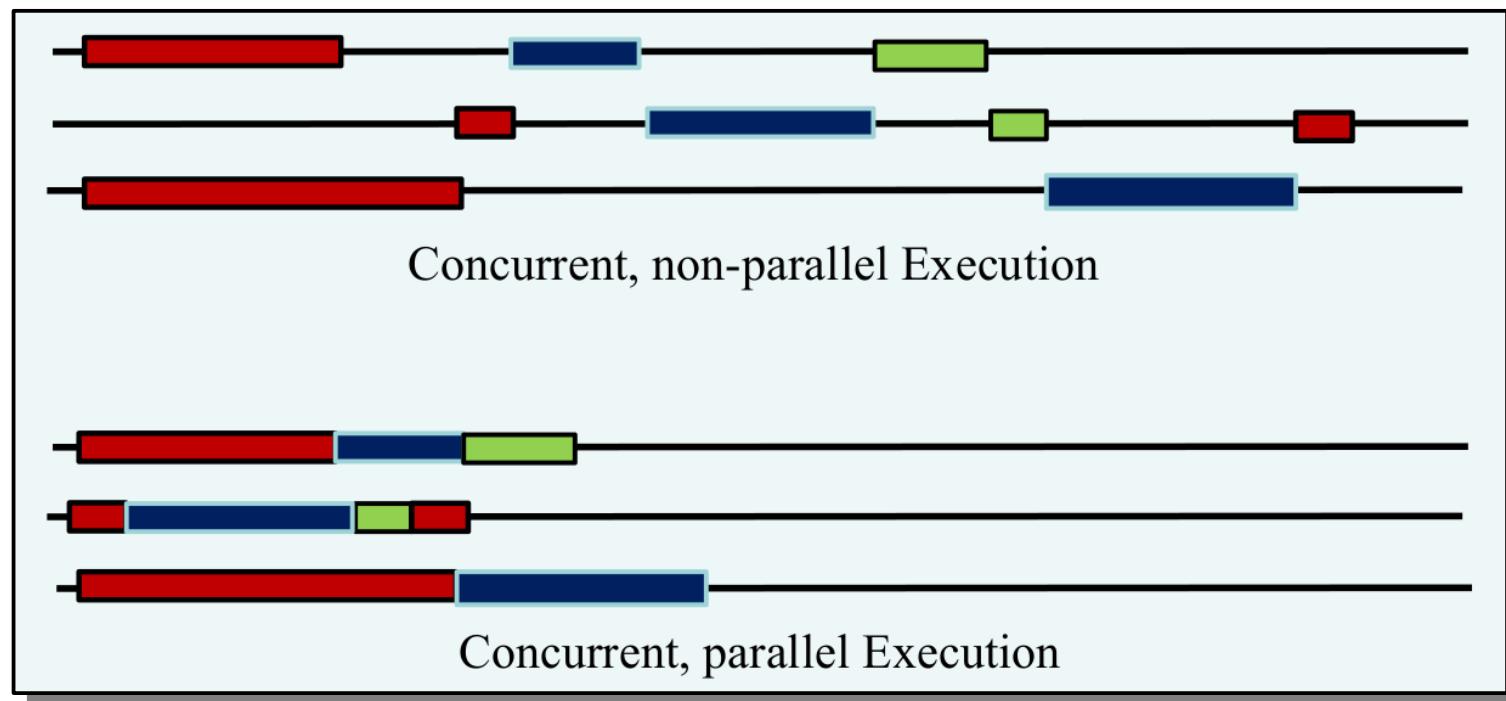
- Software
 - Sequential, Concurrent
- Hardware
 - Serial, Parallel

Concurrency vs. Parallelism



Concurrency vs. Parallelism

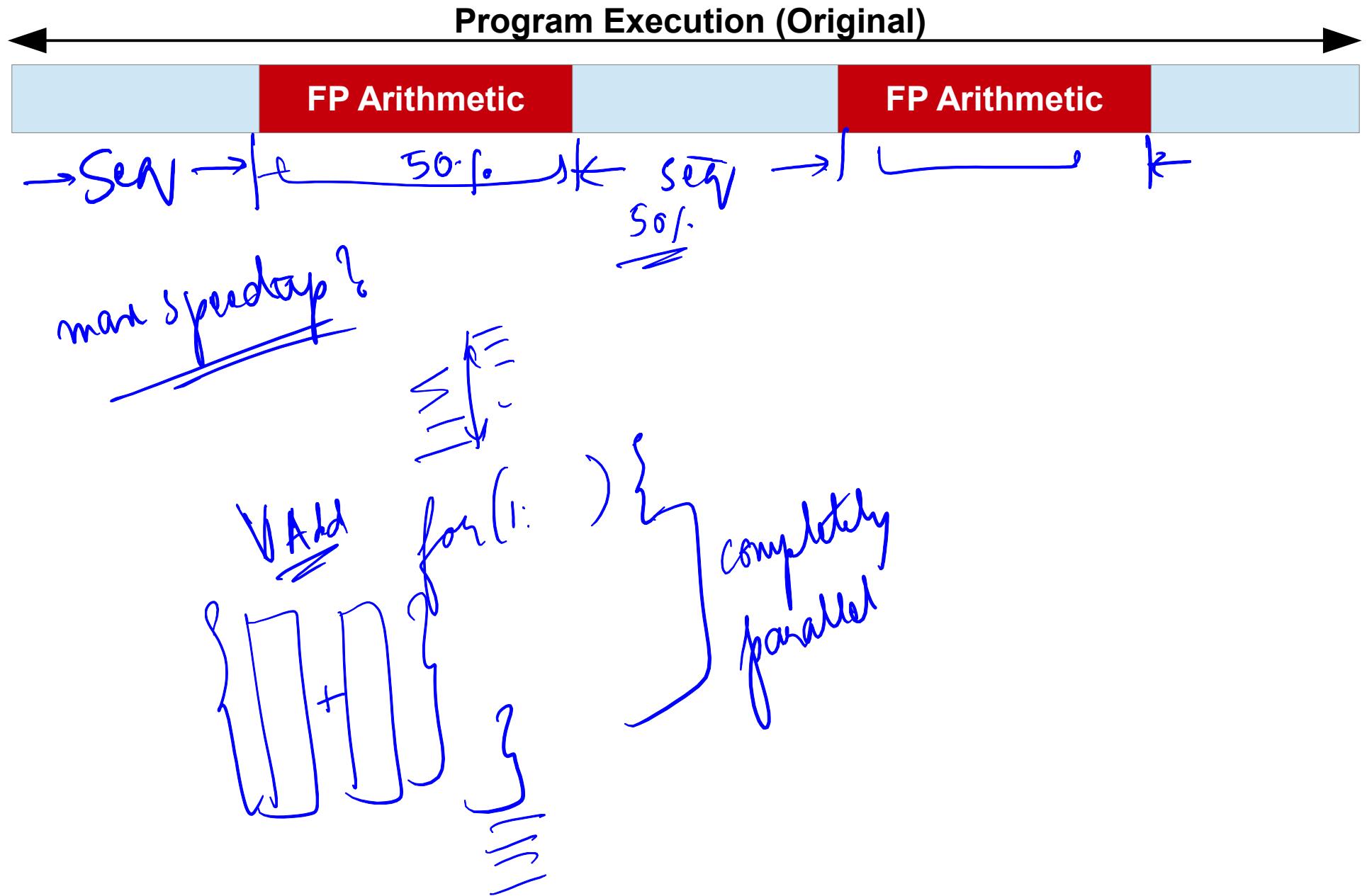
- Concurrency: A condition of a system in which multiple tasks are logically active at one time.
- Parallelism: A condition of a system in which multiple tasks are actually active at one time.



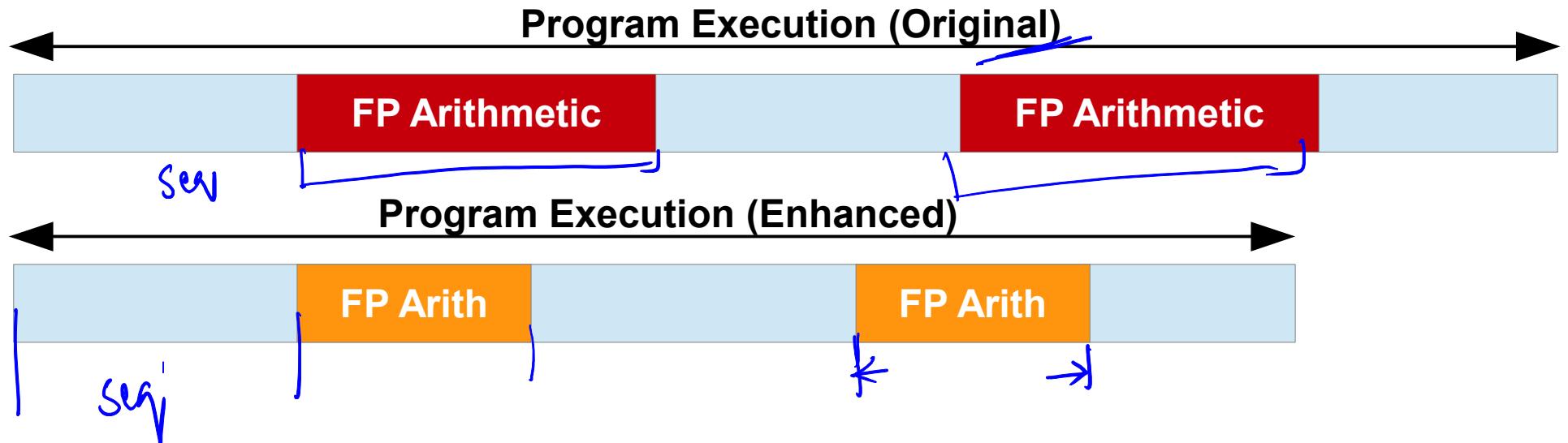
Amdahl's Law

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

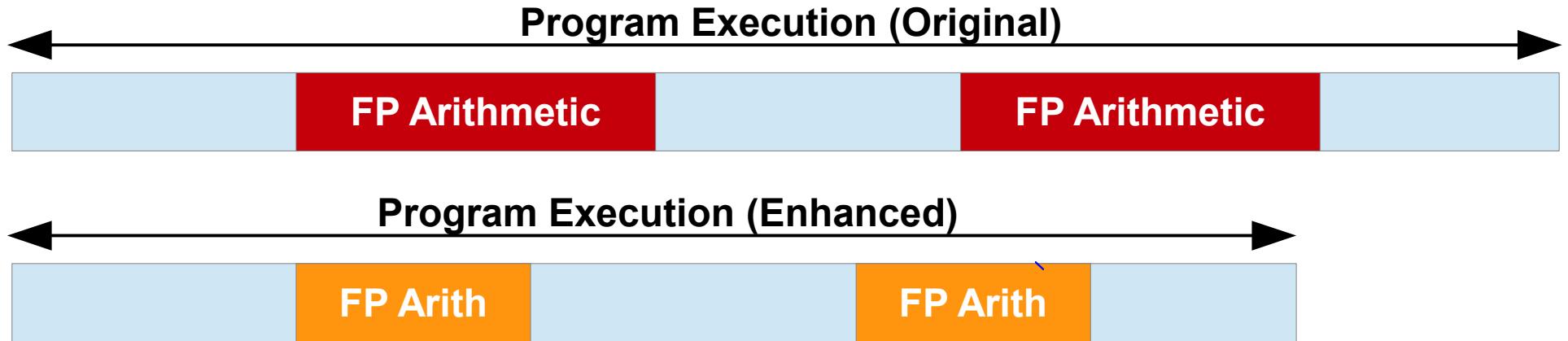
Amdahl's Law



Amdahl's Law

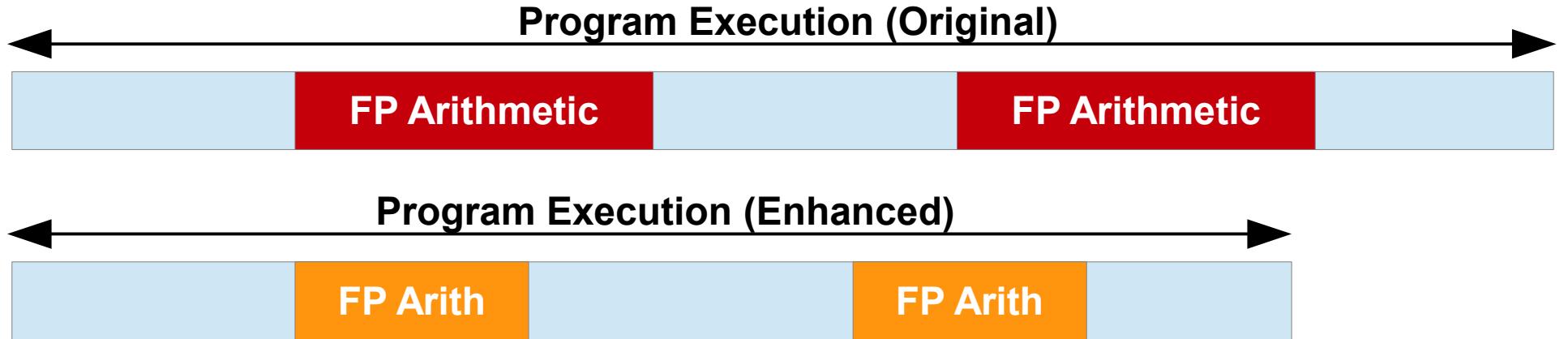


Amdahl's Law



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

Amdahl's Law



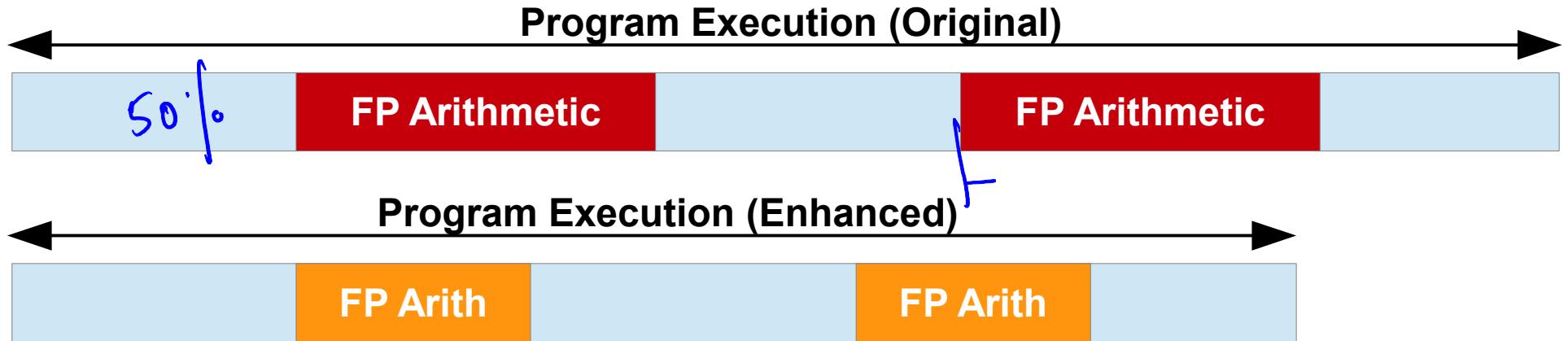
- What is the overall speedup by enhancing the performance of a single block?

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

single threaded

← original *← enhancement*

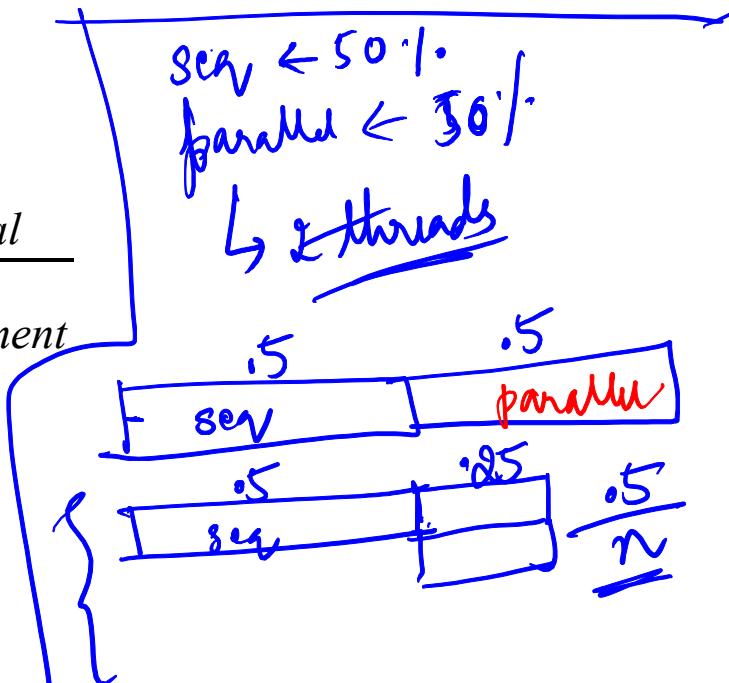
Amdahl's Law



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

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

- $\underline{\text{Speedup}_{enhanced} (\text{always } > 1)}$
- $\underline{\text{Fraction}_{enhanced} (\text{always } < 1)}$



Amdahl's Law

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

Diagram illustrating Amdahl's Law:

- The formula is enclosed in a box.
- Handwritten annotations explain the components:
 - seen portion: points to the term $(1 - Fraction_{enhanced})$.
 - parallel: points to the denominator of the first term $\frac{(1 - Fraction_{enhanced})}{parallel}$.
 - threads: points to the denominator of the second term $\frac{Fraction_{enhanced}}{\frac{Speedup_{enhanced}}{threads}}$.
- Below the box, a handwritten equation shows a specific example:
$$T_{new} = T_{old} * \left[(1 - 0.5) + \frac{0.5}{\frac{1}{w}} \right]$$

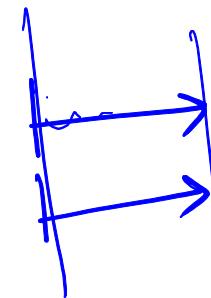
Amdahl's Law

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



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

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

parallel portion
no. of threads

↳ no. of threads

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?

~~max speedup = 8~~

~~= 0.25 + 0.75~~

~~= 6.25 x 8~~

~~min prof. 1 = T_{seq} + $T_{parallel}$~~

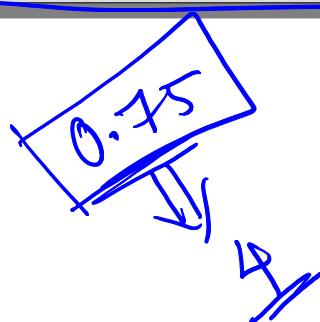
~~Acore~~

~~= T_{seq} + $\frac{T_{par}}{4}$~~

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

run application in parallel

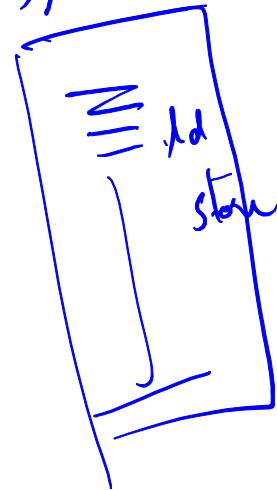


Flynn's Classification

~~1964/65~~

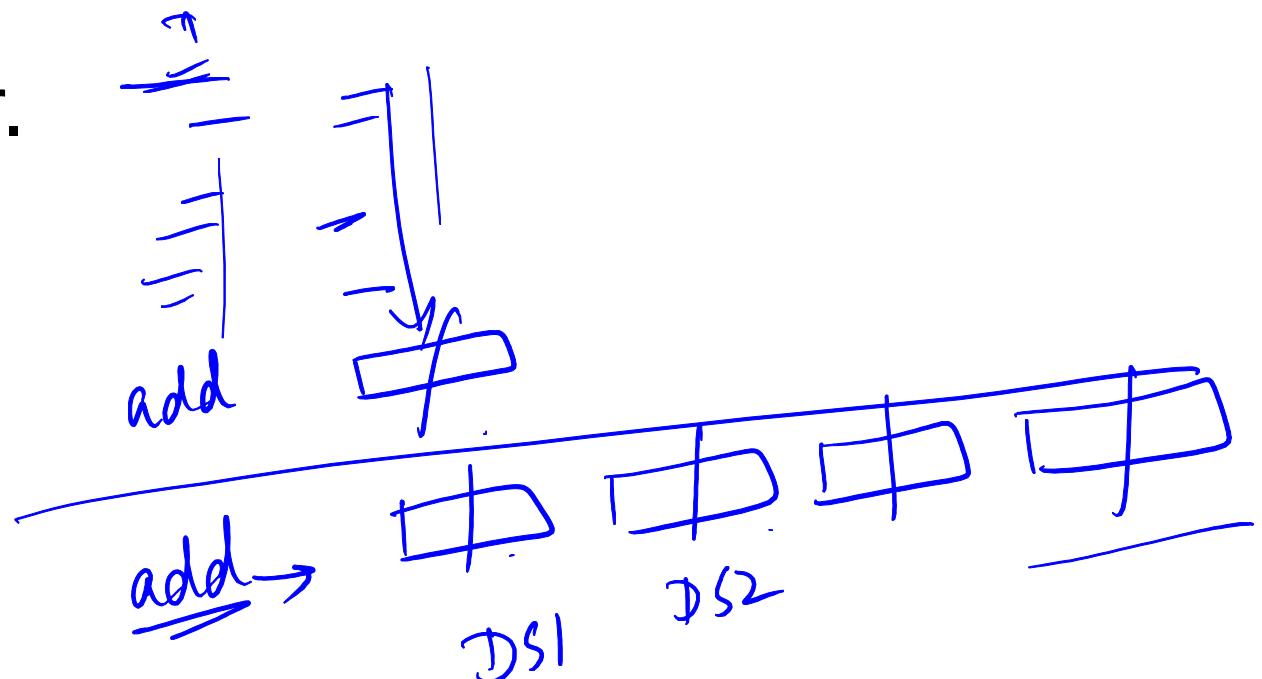
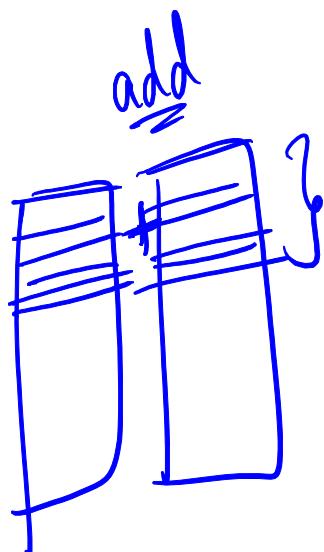
data stream

Instruction
stream



Flynn's Classification

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



~~DAXPY~~

Flynn's Classification

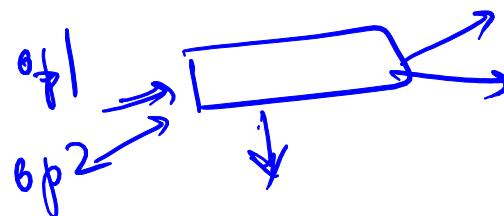
- **Single instruction stream, single data stream (SISD)**
 - Uniprocessor.
- **Single instruction stream, multiple data streams (SIMD)**
 - Data-level parallelism, Subword Parallelism
 - Applying same operations to multiple items of data in parallel
 - Eg. Multimedia extensions, Vector architectures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

→ SIMD
SIMD

Flynn's Classification

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

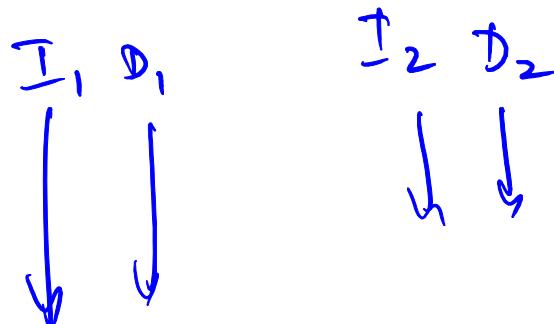
* func on same data



This slide is intentionally left blank

Flynn's Classification

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



SIMD

add R₁, R₂ R₃

vector

SIMD Instructions

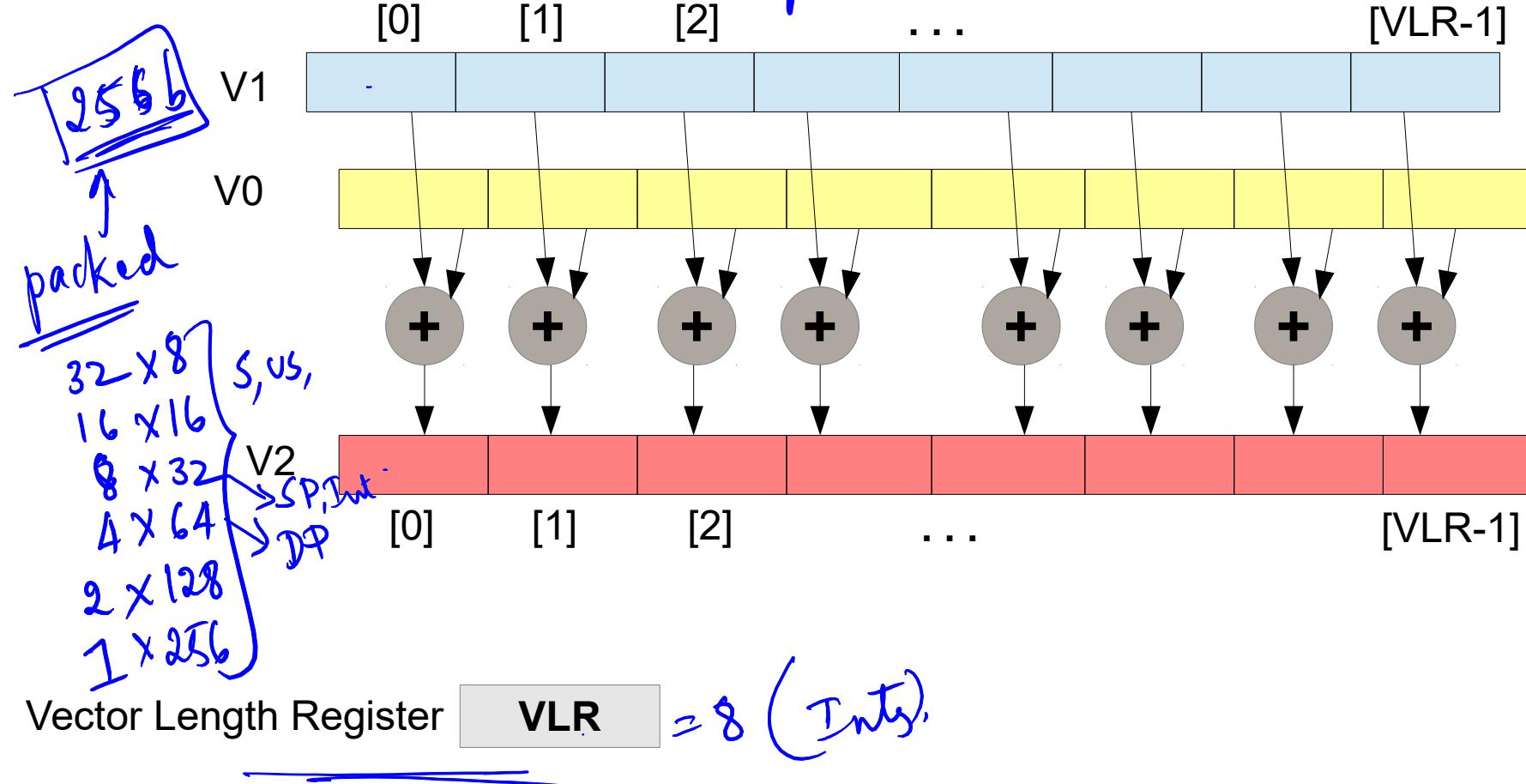
ADDVV V2, V0, V1

SIMD Instructions

5 us
1 V.
32 to int

ADDVV V2, V0, V1

← more ALUs
← Regs Vector
← CV signals all SIMD ALUs.



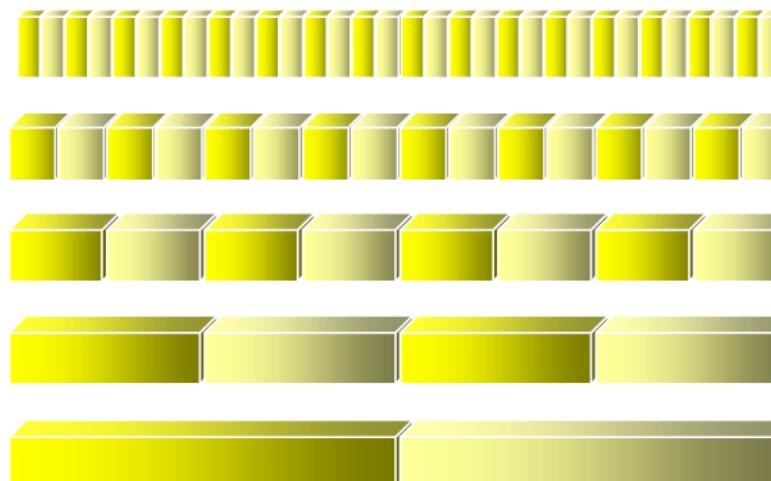
MMX, SSE, AVX,

SSE



4x floats

Intel®
AVX2



32x bytes

16x 16-bit shorts

8x 32-bit integers

4x 64-bit integers

2x 128-bit(!) integer

Int, FP
S, WS SP, DP

AVX-
512



16x floats

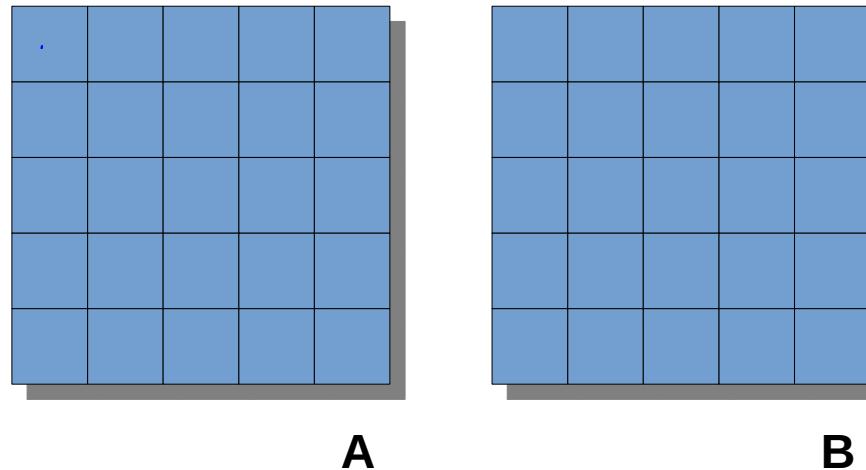
8x doubles



16x 32-bit integers

...

DGEMM – Access Pattern



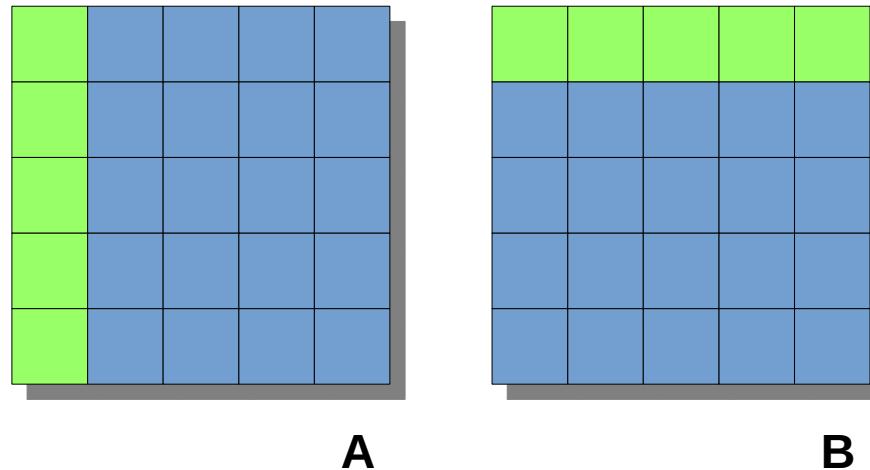
```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        DotProduct = A[i + j*n] + B[i*n + j];
```

i=0 j A₀₀
0 A₁₀
1 A₂₀
2 A₃₀
:
n-1 :

↑
Columnise

→ how wise
i=0 j=0 B₀₀
1 B₀₁
2 B₀₂
n-1 :

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.     {
6.
7.         for(int k = 0; k < n; k++ )
8.             cij += A[i+k*n] * B[k+j*n];
9.             ↑   /* 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.
7.         for(int k = 0; k < n; k++ )
8.             cij += A[i+k*n] * B[k+j*n];
9.
10.    }
11. }
```

The diagram illustrates the memory access pattern for the `dgemm` function. It shows two annotations pointing to specific parts of the code:

- A yellow box labeled "Column wise" points to the inner loop where `k` is iterated from 0 to `n`. This indicates that the computation is performed column-wise.
- A yellow box labeled "Row wise" points to the outer loop where `j` is iterated from 0 to `n`. This indicates that the computation is performed row-wise.

Below the annotations, a comment in the code is shown: `/* cij += A[i][k]*B[k][j] */`, which describes the operation being performed at the intersection of row `i` and column `j`.

Matrix Multiply

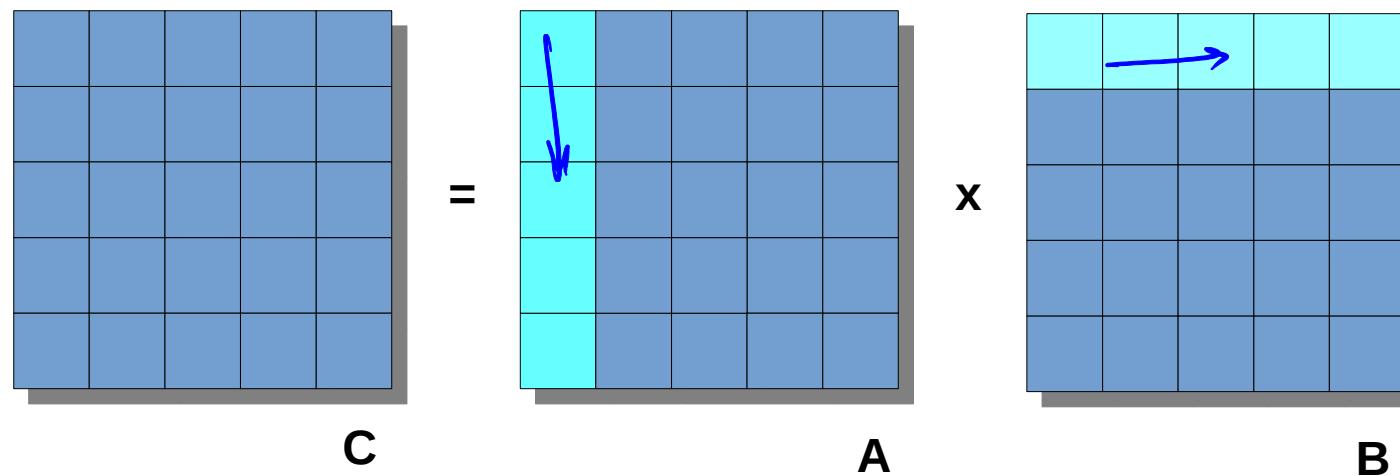
```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.     for (int i = 0; i < n; ++i)
4.         for (int j = 0; j < n; ++j)
5.     {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for(int k = 0; k < n; k++)
8.             cij += A[i+k*n] * B[k+j*n];
                  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

Matrix Multiply

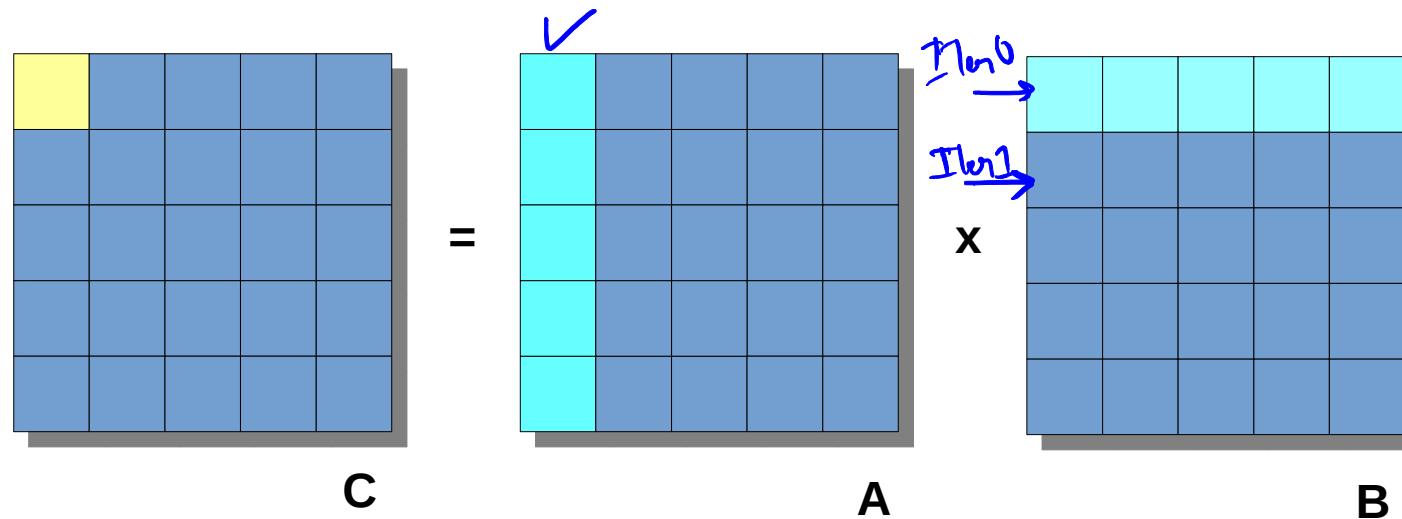
```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.     for (int i = 0; i < n; ++i)
4.         for (int j = 0; j < n; ++j)
5.     {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for(int k = 0; k < n; k++)
8.             cij += A[i+k*n] * B[k+j*n];
9.             /* cij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

Column
wise

Matrix Multiply – Basic Version



Matrix Multiply – Basic Version



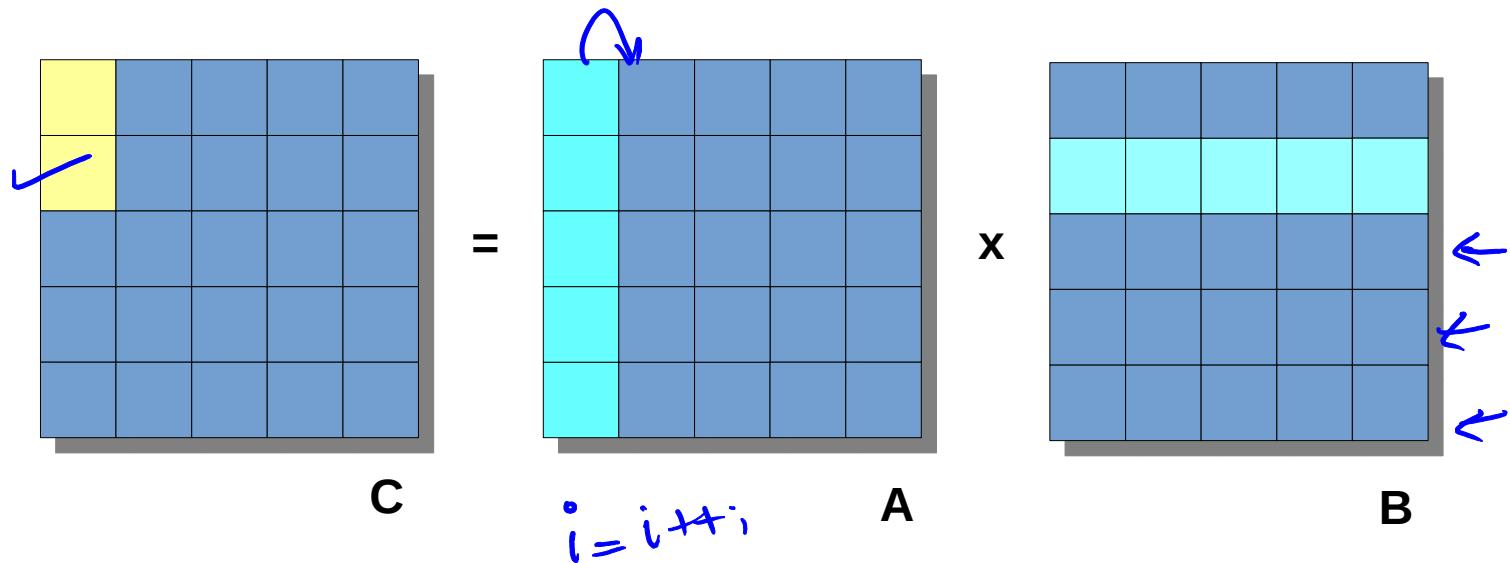
Matrix Multiply – Basic Version

$$= \begin{matrix} & & \\ & & \\ & & \\ & & \\ & & \end{matrix} \times \begin{matrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{matrix}$$

A

B

Matrix Multiply – Basic Version

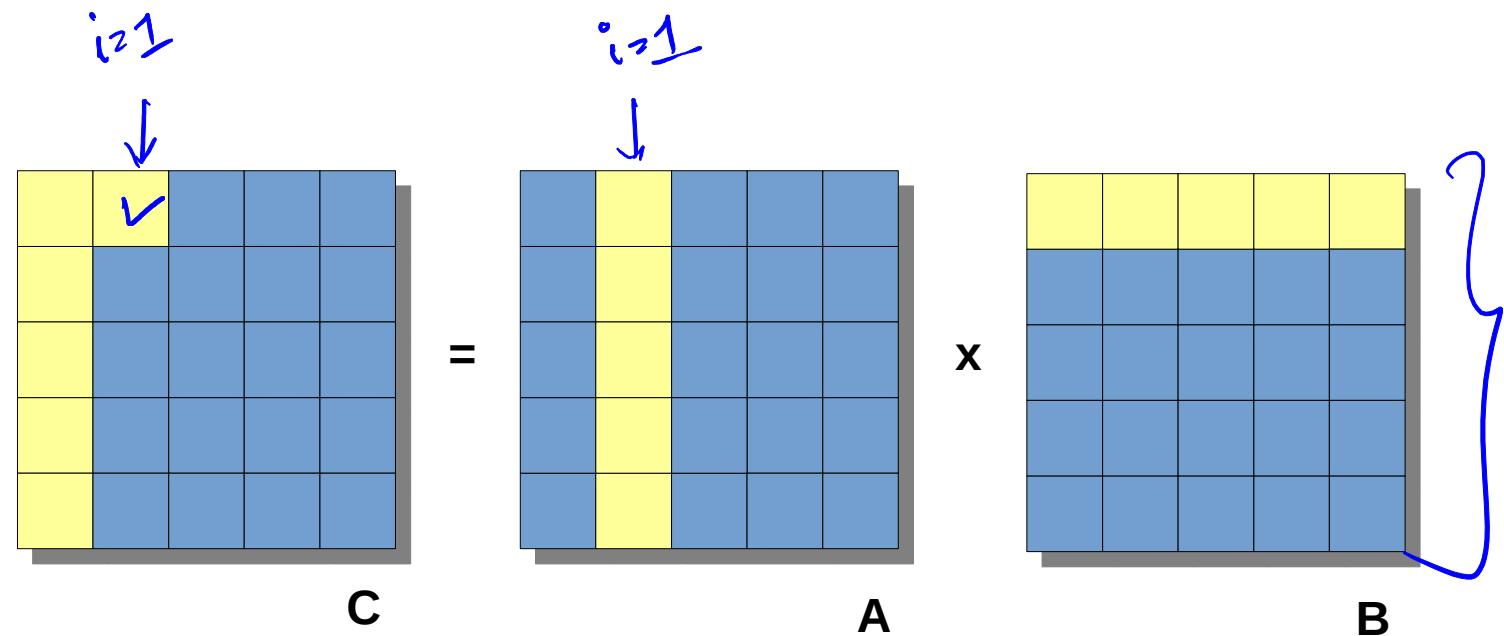


Matrix Multiply – Basic Version

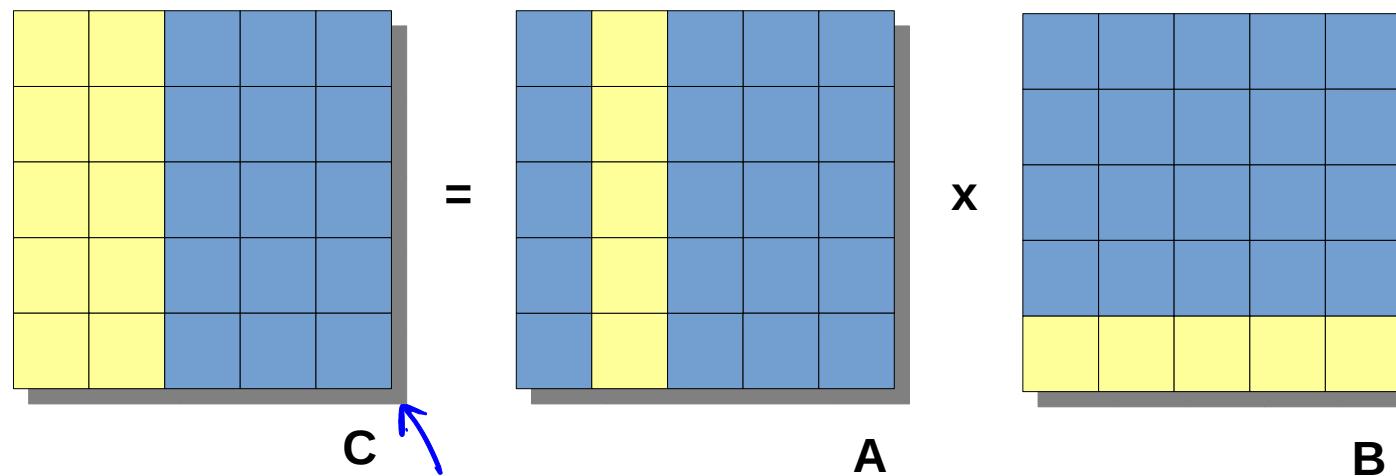
$$C = A \times B$$

The diagram illustrates the basic version of matrix multiplication. It shows three matrices: **A**, **B**, and **C**. Matrix **A** is a 5x5 matrix with columns colored yellow, cyan, blue, blue, blue. Matrix **B** is a 5x5 matrix with rows colored cyan, blue, blue, blue, blue. The result matrix **C** is a 5x5 matrix with columns colored yellow, cyan, blue, blue, blue. The multiplication is represented by the equation $C = A \times B$.

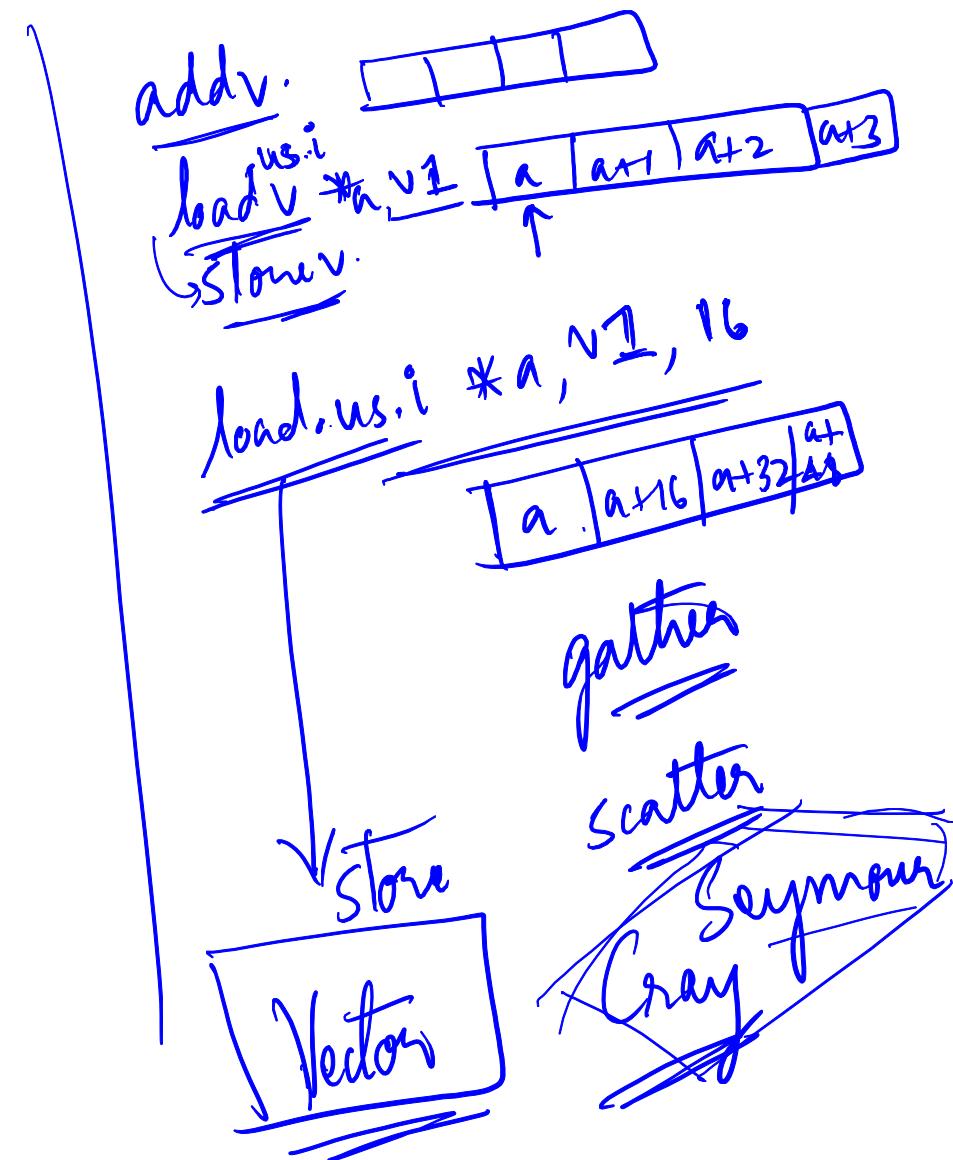
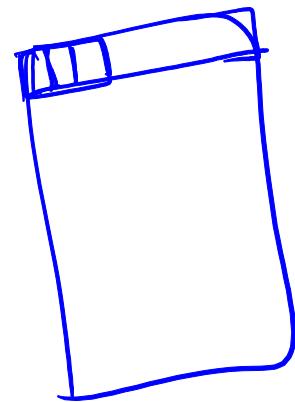
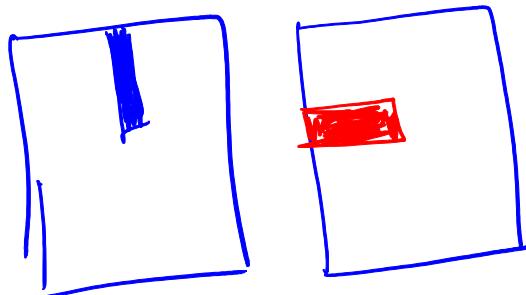
Matrix Multiply – Basic Version



Matrix Multiply – Basic Version



Matrix Multiply – SIMD Version



Matrix Multiply – SIMD Version

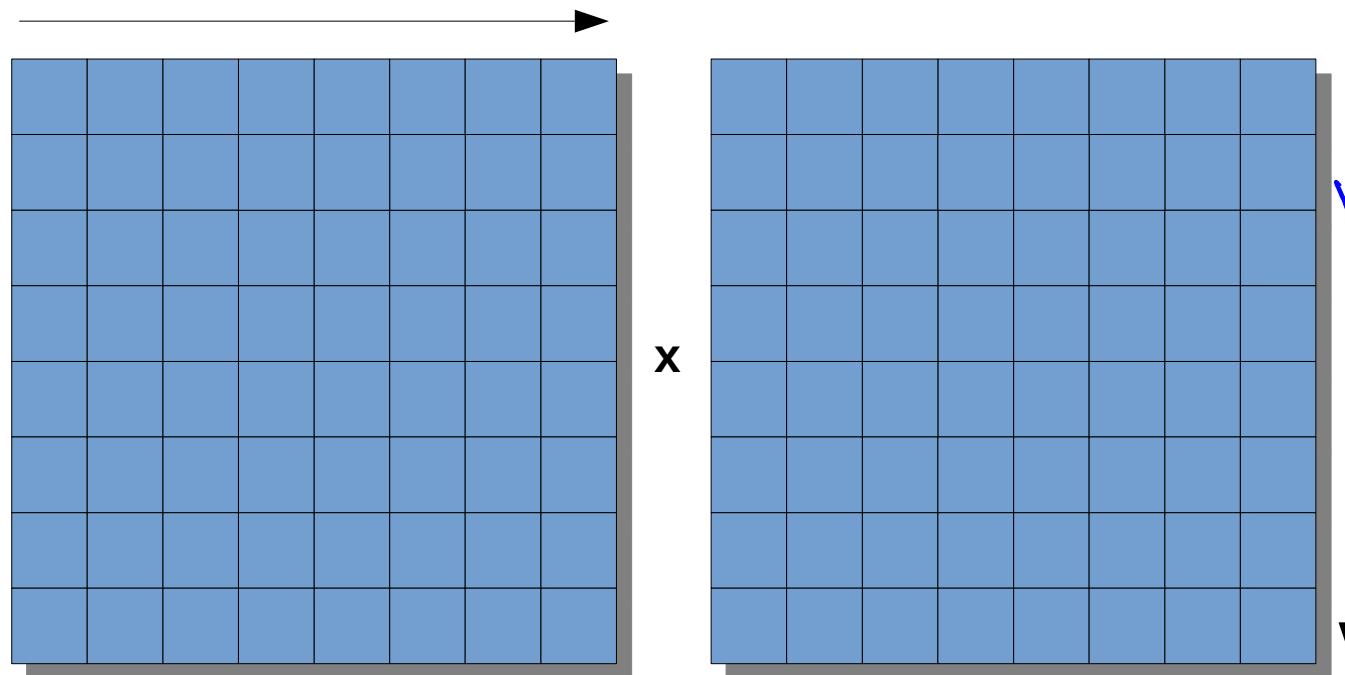
- Which are the SIMD operations?

- $$\underline{A_{00} \cdot \underline{B_{00}}} + \underline{A_{10} \cdot \underline{B_{01}}} + \underline{\underline{A_{20} \cdot B_{02}}} + \underline{A_{30} \cdot B_{03}}$$

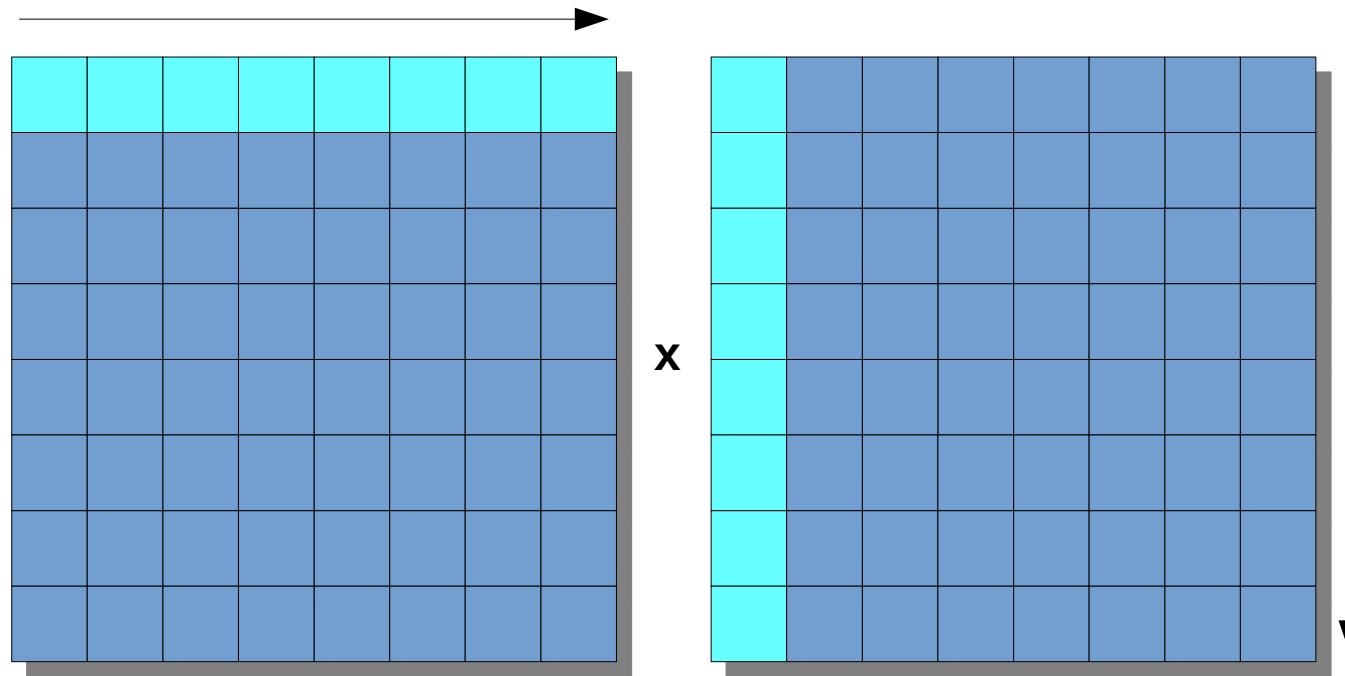
$\underline{A_{00} \cdot \underline{B_{00}}} + \underline{A_{10} \cdot \underline{B_{01}}} + \underline{\underline{A_{20} \cdot B_{02}}} + \underline{A_{30} \cdot B_{03}}$

$\underbrace{\underline{A_{00} \cdot \underline{B_{00}}}, \underline{A_{10} \cdot \underline{B_{01}}}, \underline{\underline{A_{20} \cdot B_{02}}}, \underline{A_{30} \cdot B_{03}}}$

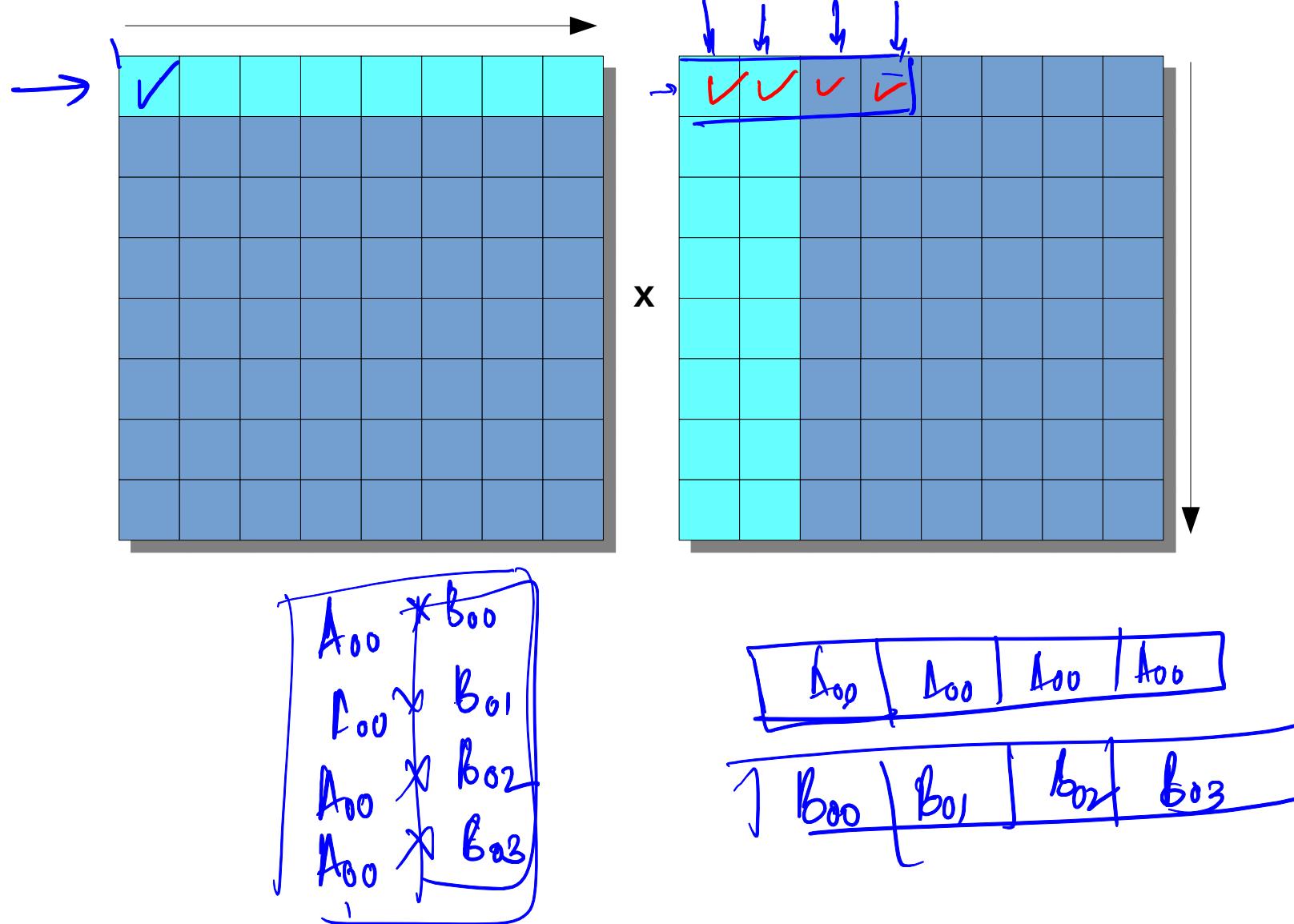
Matrix Multiply – SIMD Version



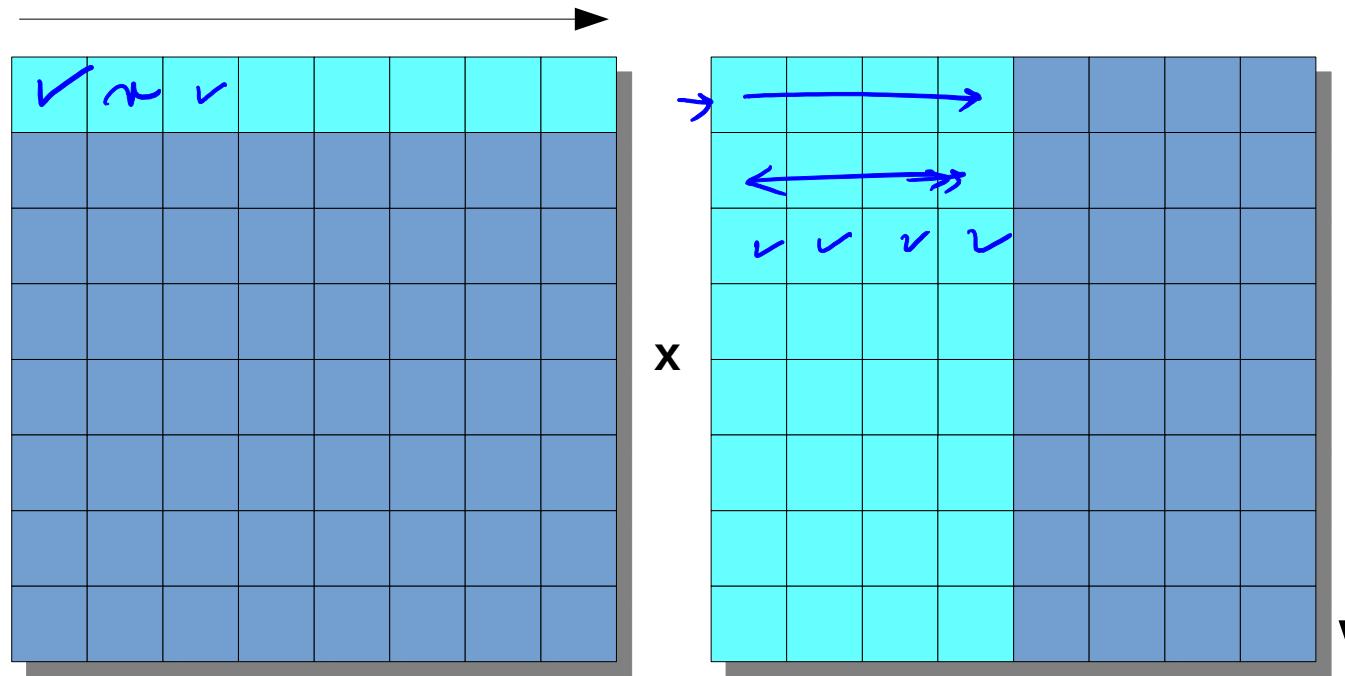
Matrix Multiply – SIMD Version



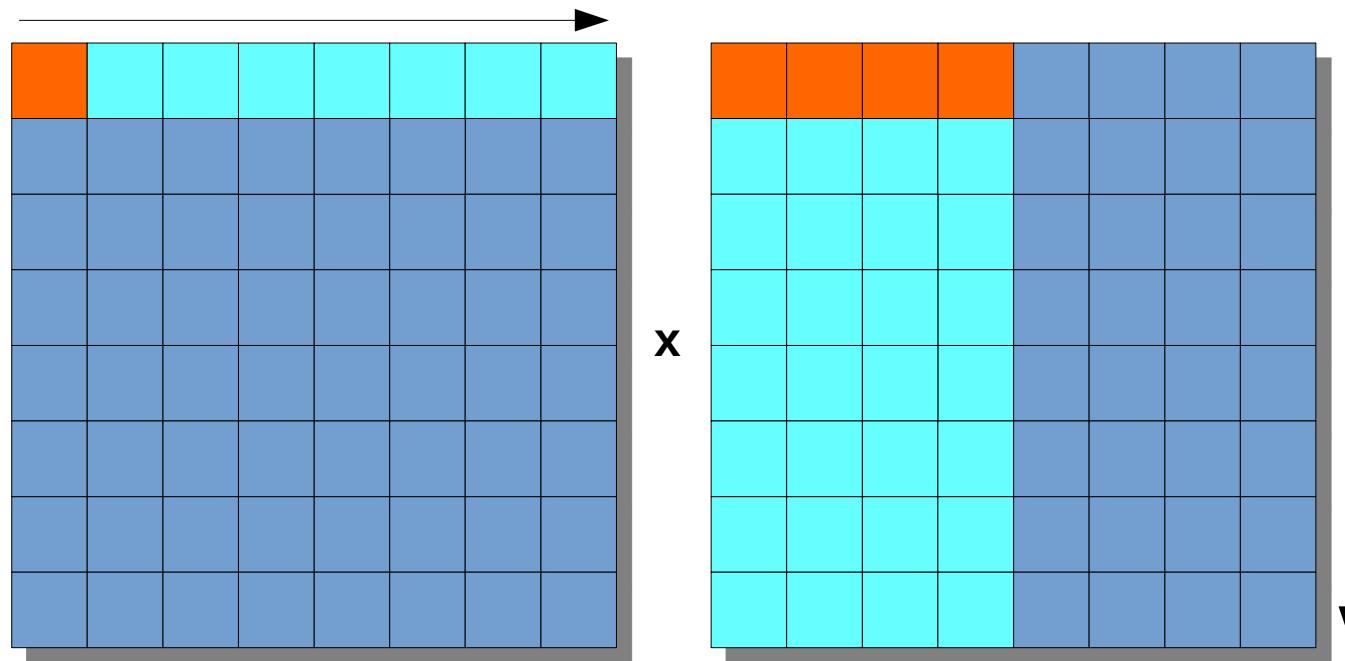
Matrix Multiply – SIMD Version



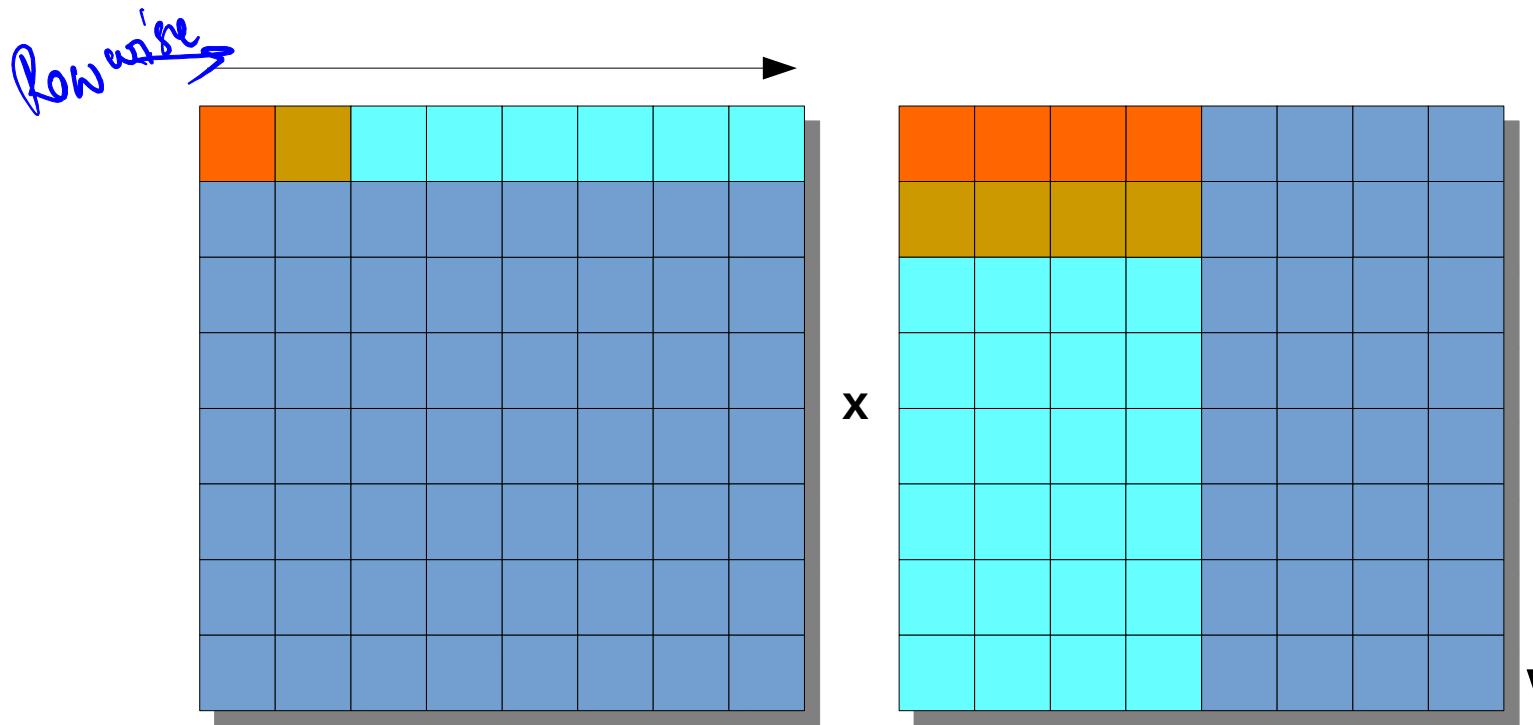
Matrix Multiply – SIMD Version



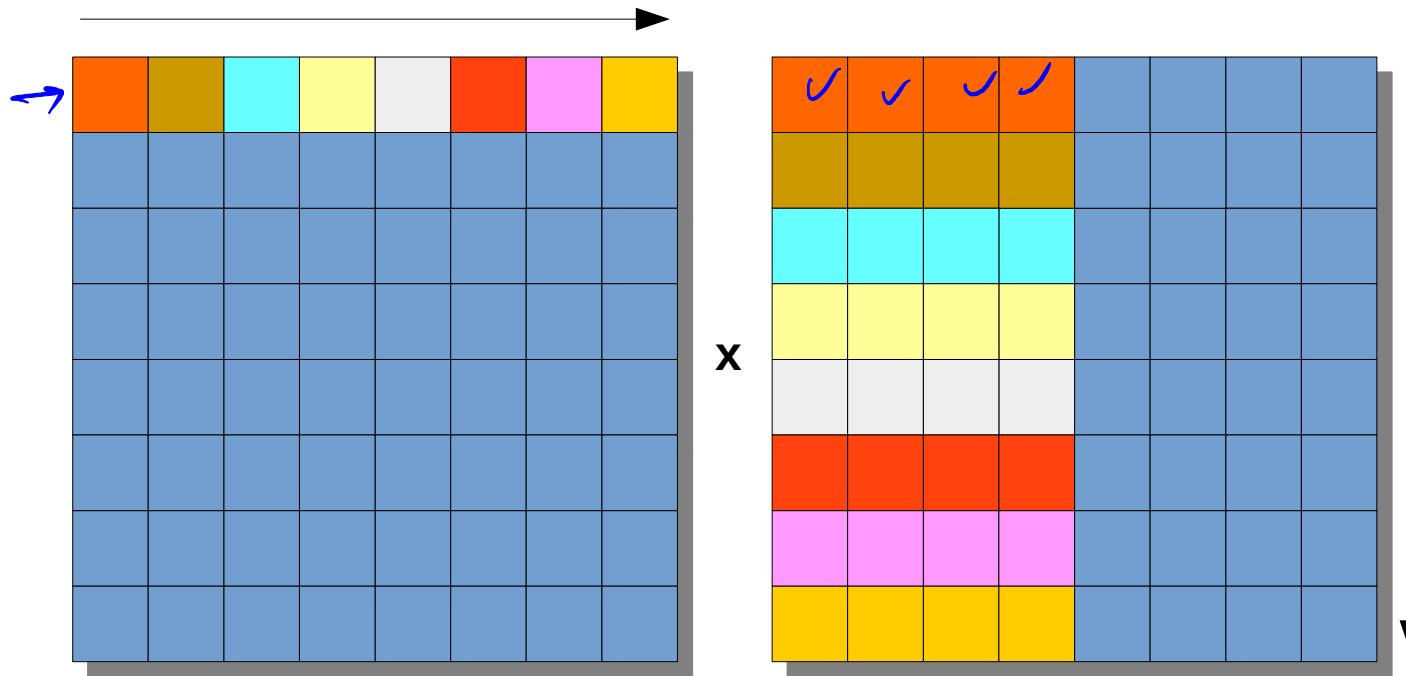
Matrix Multiply – SIMD Version



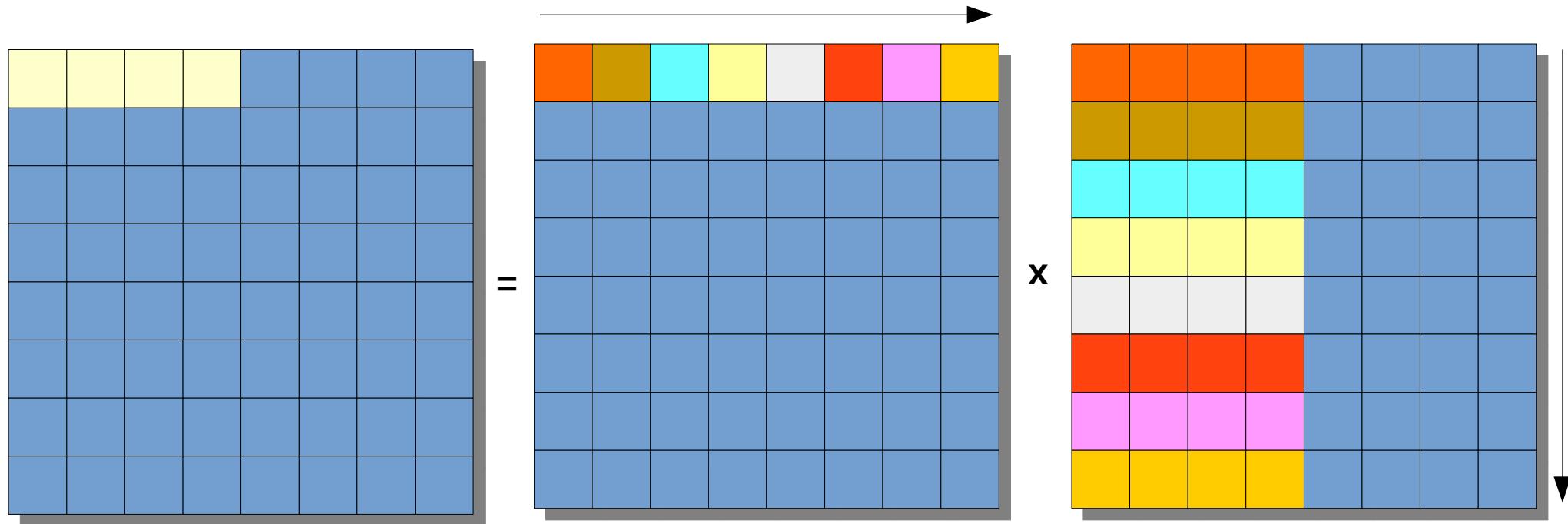
Matrix Multiply – SIMD Version



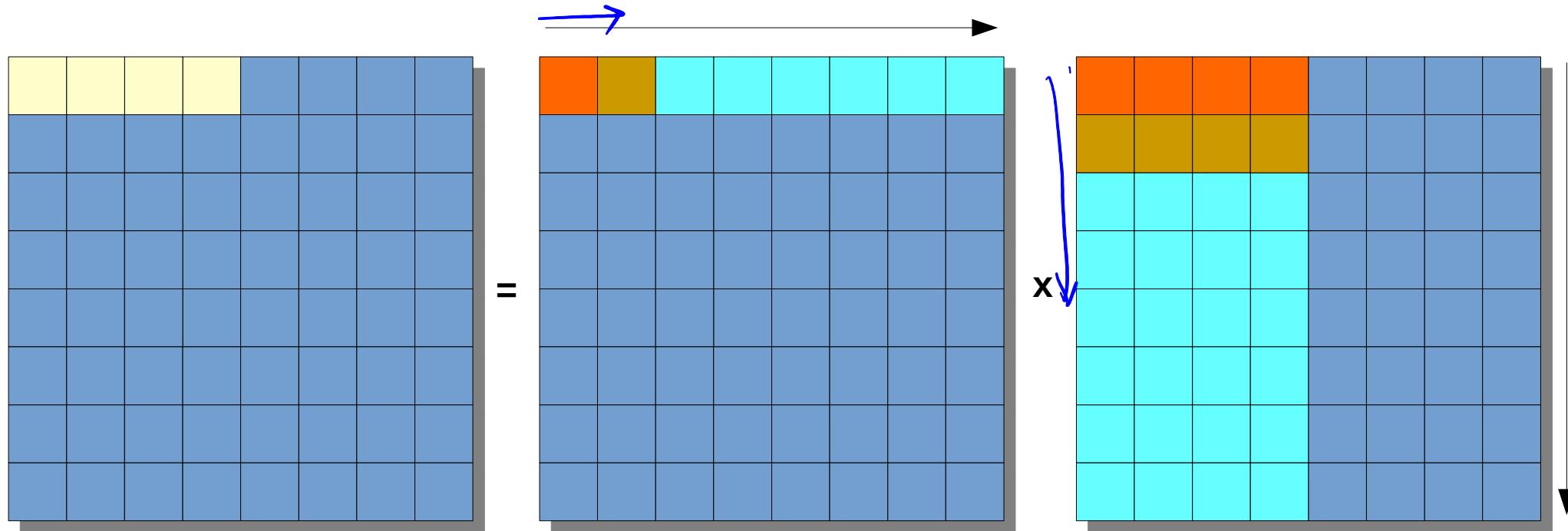
Matrix Multiply – SIMD Version



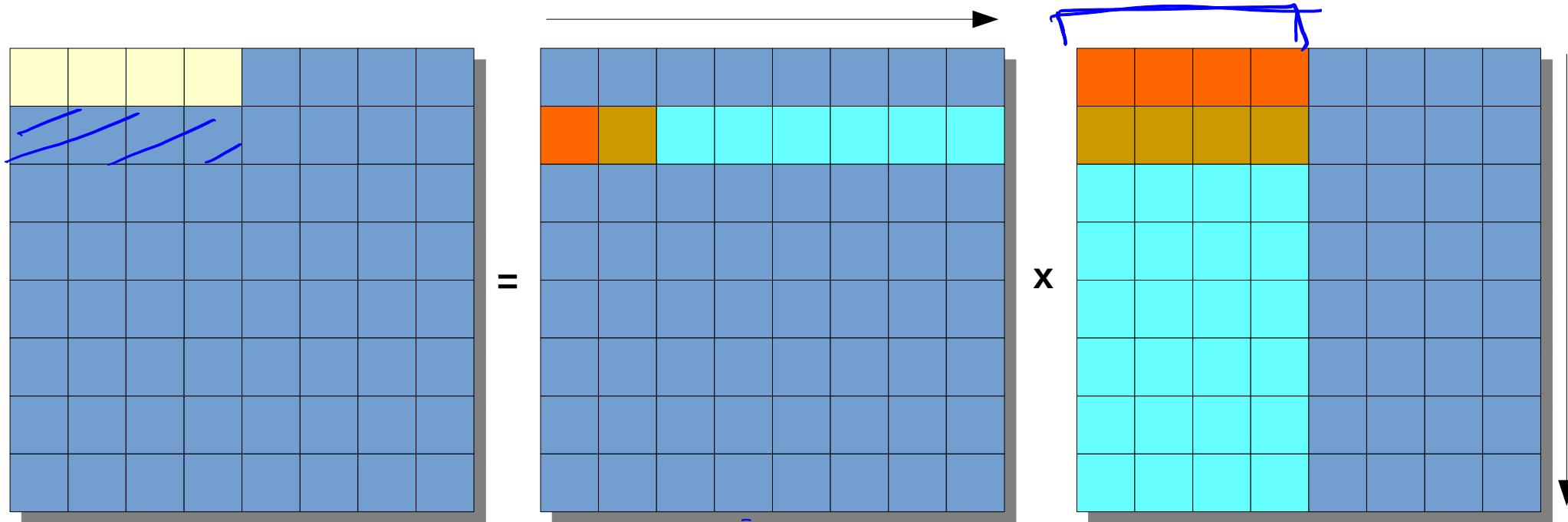
Matrix Multiply – SIMD Version



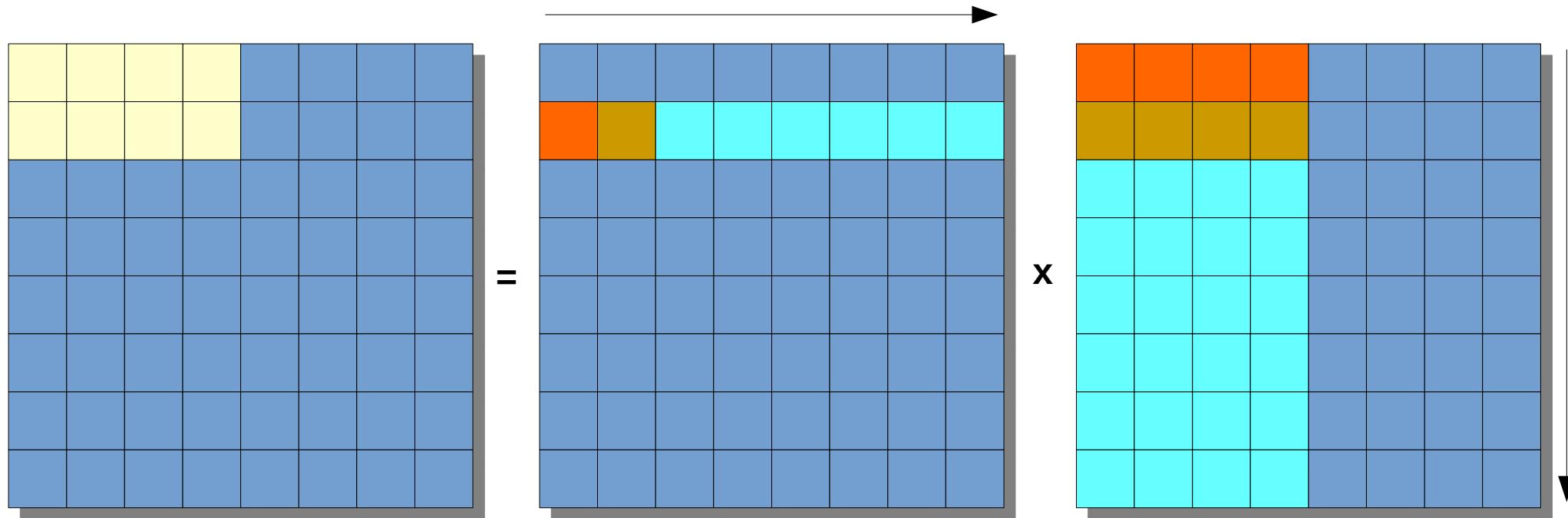
Matrix Multiply – SIMD Version



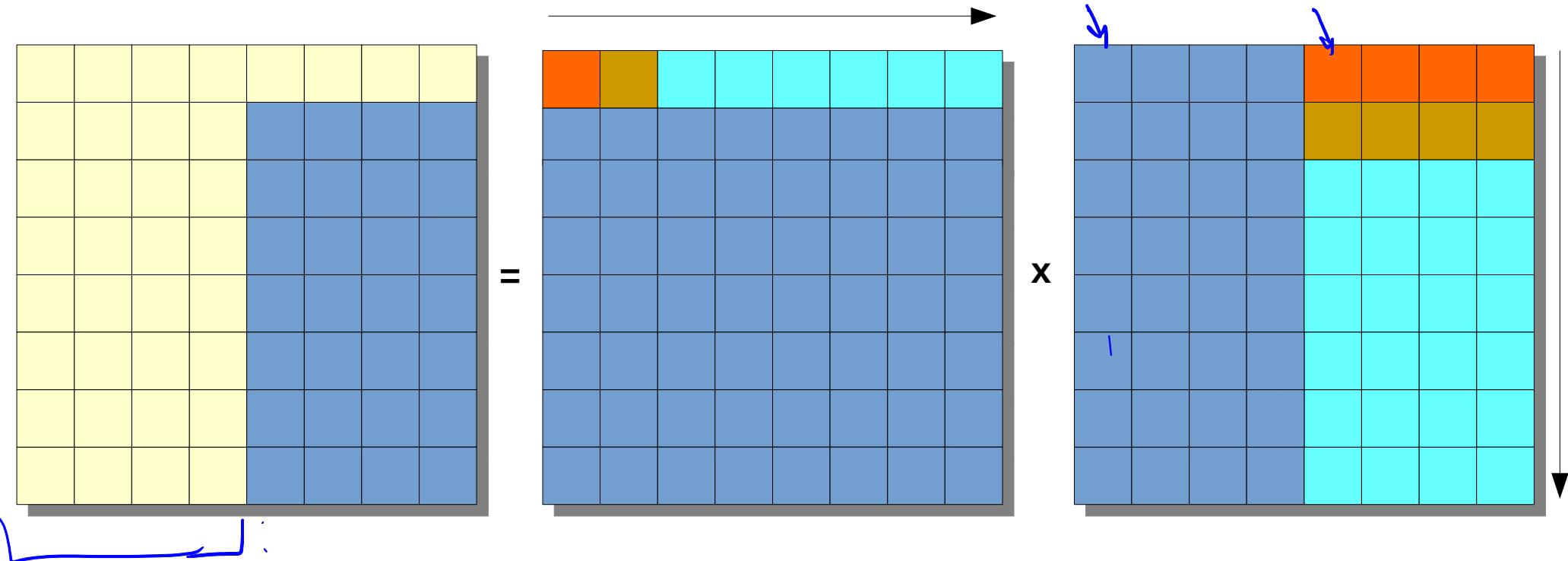
Matrix Multiply – SIMD Version



Matrix Multiply – SIMD Version



Matrix Multiply – SIMD Version



SSE Instructions

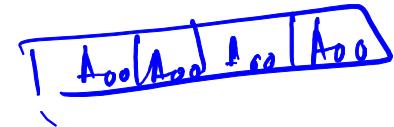
```
extern __m256d _mm256_load_pd(double const *a);
```



*↑
Packed double*

- `*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++ ) {
6.             /* c0 = C[i][j] */
7.             _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++ ) {
6.             /* c0 = C[i][j] */
7.             __m256d c0 = _mm256_load_pd(C+i+j*n);
8.             for( int k = 0; k < n; k++ )
9.                 c0 = _mm256_add_pd(c0, _mm256_load_pd(B+k*n));
10.            _mm256_store_pd(C+i+j*n, c0);
11.        }
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++ ) {
6.             /* c0 = C[i][j] */
7.             __m256d c0 = _mm256_load_pd(C+i+j*n) ;
8.             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))) ;
11.             _mm256_store_pd(c0, C+i+j*n);
12.         }
13. }
```

↑ col wise
↑ row wise

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++ ) {
6.             /* c0 = C[i][j] */
7.             __m256d c0 = _mm256_load_pd(C+i+j*n) ;
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++ ) {
6.             /* c0 = C[i][j] */
7.             __m256d c0 = _mm256_load_pd(C+i+j*n) ;
8.             for( int k = 0; k < n; k++ )
9.                 c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
10.                               _mm256_mul_pd(_mm256_load_pd(A+i+k*n) ,
11.                                             _mm256_broadcast_sd(B+k+j*n))) ;
12.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
13. }
```

