COL380

Introduction to
Parallel & Distributed Programming

2-0-2

Subodh Kumar

Programming Test:

Wednesday, April 23 5-6pm

Resources

- HPC
  - ➡ Login nodes (ssh hpc.iitd.ac.in) * set up password-less login *
  - ➡ Use PBS to submit jobs
  - ➡ module add compiler/gcc/11.2/openmpi/4.1.6 (also see: module list)
  - ➡ https://supercomputing.iitd.ac.in

```
$ amgr login
$ amgr checkbalance project -n col380.subodh.course
```

- css
  - ➡ ssh css.cse.iitd.ac.in
- perf, OpenMP, MPI, Cuda (Others)

OS Basics/Review

- Process & thread
  - ➡ Scheduling, Context-switching and concurrency
  - ➡ User space, Kernel space
  - ➡ System calls
- Address space & Name space
- Virtual Memory
  - ➡ Caches
- Synchronization

✦ Shell (bash)
✦ time
✦ PMU (perf)
✦ Context switch
  ✦ and scheduling
✦ System calls
✦ Interrupts

Subodh Kumar

Process, Thread

code.cpp:

```
int turn = 0;
void handler(int sig) {  turn = !turn; alarm(1); }

void func1() {
   static int i = 0;
   while(i < many1) {
       if(turn == 0) func0();
       else dothing1(i++);
   }
   turn = 0; alarm(0);
}

void func0() {
   static int i = 0;
   while(i < many0) {
       if(turn == 1) func1();
       else dothing0(i++);
   }
   turn = 1; alarm(0);
}
```

```
signal(SIGALRM, handler);
alarm(1);
```

```
g++ code.cpp -o exe
./exe &
./exe &
```

Subodh Kumar

Process, Thread

code.cpp:

```
int turn = 0;
void handler(int sig) {  turn = !turn; alarm(1); }

void func1() {
   static int i = 0;
   while(i < many1) {
       if(turn == 0) func0();
       else dothing1(i++);
   }
   turn = 0; alarm(0);
}

void func0() {
   static int i = 0;
   while(i < many0) {
       if(turn == 1) func1();
       else dothing0(i++);
   }
   turn = 1; alarm(0);
}
```

```
signal(SIGALRM, handler);
alarm(1);
```

```
g++ code.cpp -o exe
./exe
```

Subodh Kumar

First Parallel Program

```
float dot(float *a, float *b, int n)
{
    int i;
    float s0=0, s1=0, s2=0, s3=0;
    for(i=0; i<n/4*4; i+=4)
        s0 += a[i]*b[i];
        s1 += a[i+1]*b[i+1];
        s2 += a[i+2]*b[i+2];
        s3 += a[i+3]*b[i+3];
    for(; i<n; i++)
        s0 += a[i]*b[i];
    return s0+s1+s2+s3;
}
```

```
vaddps  (%rdi), %zmm1, %zmm0
```

Single Vector Instruction

Subodh Kumar

Code Execution

```c
float dot(float *a, float *b, int n)
{
    int i;
    float s0=0;
    for(i=0; i<n; i++)
        s0 += a[i]*b[i];
    return s0;
}
```

```
     test   %rdx,%rdx              // n == 0? (rdx)
     jle    29 <dot+0x29>          // exit if 0
     mov    $0x0,%eax              // i = 0 (eax/rax)
     pxor   %xmm1,%xmm1            // s0 = 0 (xmm1)
e.   movss  (%rdi,%rax,4),%xmm0    // a[i] -> tmp (xmm0)
     mulss  (%rsi,%rax,4),%xmm0    // mul b[i] to tmp (xmm0)
     addss  %xmm0,%xmm1            // add tmp to s0
     add    $0x1,%rax              //  i += 1 (rax)
     cmp    %rax,%rdx              // i == n? (rdx)
25.  jne    e <dot+0xe>            // loop if ne
     movaps %xmm1,%xmm0            // s0-> xmm0
     retq                          // return (answer in xmm0)
29.  pxor   %xmm1,%xmm1            // s0 = 0 (xmm1)
     jmp    25 <dot+0x25>          // goto exit point
```

Code Execution

```c
float dot(float *a, float *b, int n)
{
    int i;
    float s0=0;
    for(i=0; i<n; i++)
        s0 += a[i]*b[i];
    return s0;
}
```

What's in %rdi?

Virtual address of "a"

1. translate to physical
2. request memory controller

Goes through cache hierarchy

Indeterminate latency:
    blocks subsequent instructions

i not read/written (from memory)

```
     test   %rdx,%rdx                   // n == 0? (rdx)
     jle    29 <dot+0x29>               // exit if 0
     mov    $0x0,%eax                   // i = 0 (eax/rax)
     pxor   %xmm1,%xmm1
e.   movss  (%rdi,%rax,4),%xmm0
     mulss  (%rsi,%rax,4),%xmm0
     addss  %xmm0,%xmm1
     add    $0x1,%rax
     cmp    %rax,%rdx
25.  jne    e <dot+0xe>
     movaps %xmm1,%xmm0                 // s0-> xmm0
     retq                               // return (answer in xmm0)
29.  pxor   %xmm1,%xmm1                 // s0 = 0 (xmm1)
     jmp    25 <dot+0x25>               // goto exit point
```

Compiler may swap:

```
movss  (%rsi,%rax,4),%xmm0
mulss  (%rdi,%rax,4),%xmm0
```

Architecture may co-issue/swap :

```
add    $0x1,%rax
addss  %xmm0,%xmm1
```

Execution Steps

movss (%rdi,%rax,4),%xmm0

1. Map Virtual addr ➜ Physical
2. Fetch: L1, L2 .., Memory

(Multi-level) TLB-cache (CAM)
Get Physical Addr, Check protection

If TLB miss:
Walk Page Table, Add to TLB

If Page-table not resident in Memory:
Page Fault ⇒ Interrupt ⇒ Switch to Kernel
Get rescheduled after Page Table locked in

Execution Steps

`movss (%rdi,%rax,4),%xmm0`

1. Map Virtual addr ➜ Physical
2. Fetch: L1, L2 .., Memory

(Multi-level) TLB-cache (CAM)
Get Physical Addr, Check protection

If TLB miss:
Walk Page Table, Add to TLB

If Page-table not resident in Memory:
Page Fault ⇒ Interrupt ⇒ Switch to Kernel
Get rescheduled after Page Table locked in

Subodh Kumar

Execution Steps

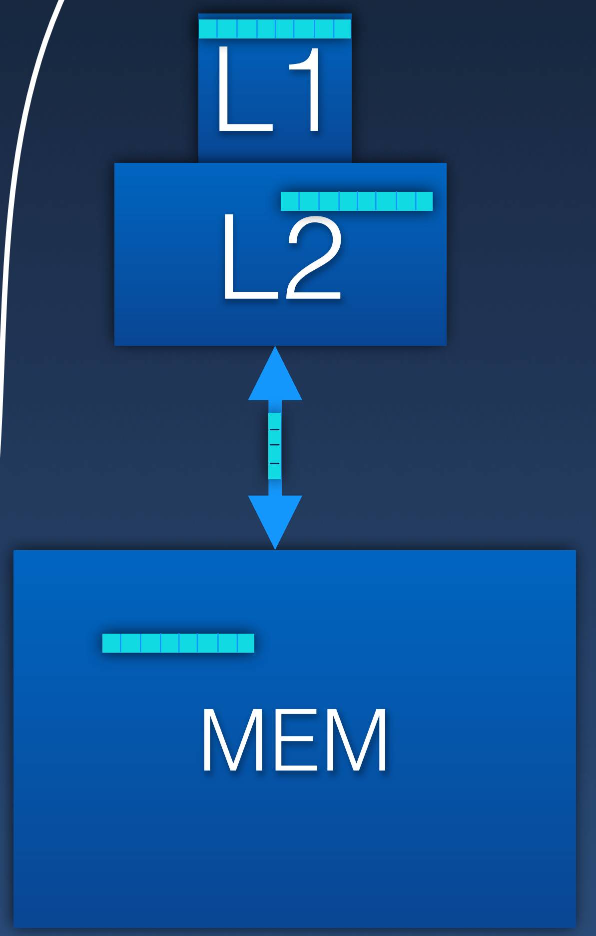
movss (%rdi,%rax,4),%xmm0

1. Map Virtual addr ➜ Physical
2. Fetch: L1, L2 .., Memory

(Multi-level) TLB-cache (CAM)
Get Physical Addr, Check protection

If TLB miss:
Walk Page Table, Add to TLB

If page not resident in Memory:
Page Fault ⇒ Interrupt ⇒ Switch to Kernel
Get rescheduled after Page Table locked in



Subodh Kumar

It's the Memory Stupid!

- Accessing variables is much 'slower' than computing
- Memory sizes are often sufficient
  - ➡ Particularly, when distributed among many nodes
    - Out-of-core computation needed sometimes (we won't focus)
- Caches are small
  - ➡ Not very helpful if traversing large swaths of memory
  - ➡ But, mind the line
    - And, prefetching is prevalent
  - ➡ Beware of sharing across caches, particularly false-sharing

Subodh Kumar

Cache Example

```
for(int i=0; i<n; i++) {
    a[i] = b[i] + c[i];
}
for(int i=0; i<n; i++) {
    d[i] = e[i] + f[i];
}
```

vs

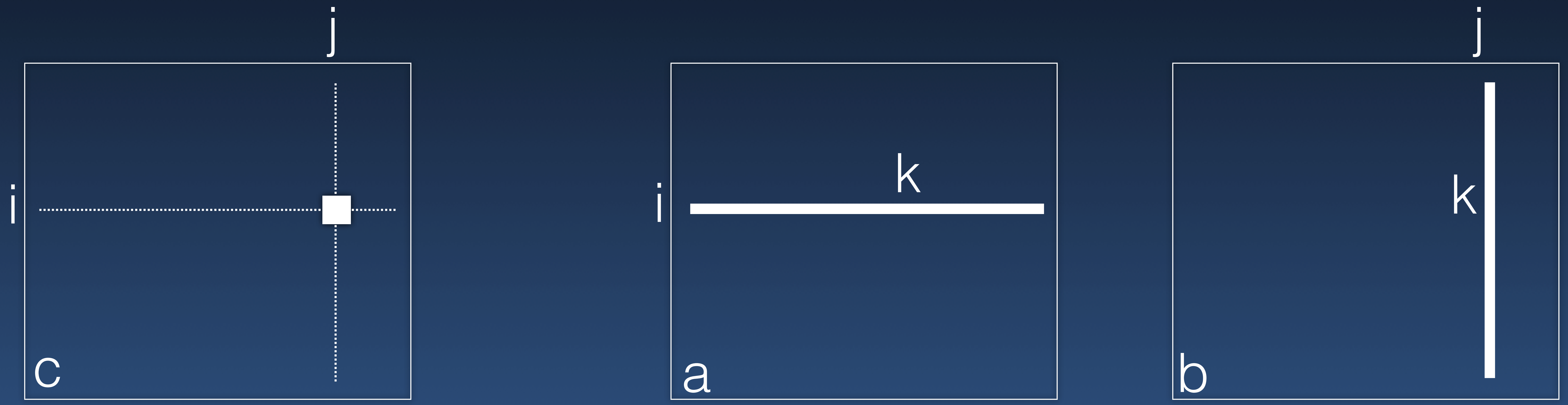
```
for(int i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    d[i] = e[i] + f[i];
}
```

```
for(int i=0; i<N; i++) {
    d[i] = x;
}
```

```
for(int i=0; i<N*STEP; i++) {
    d[(i*STEP)] = x;
}
```

Subodh Kumar

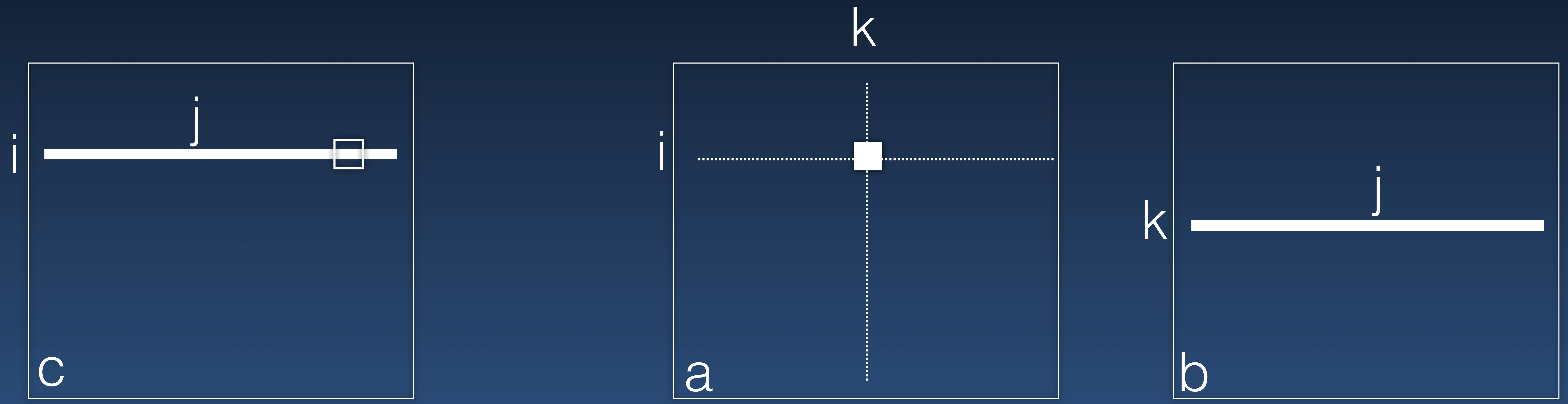
Matrix Multiplication




```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k]*b[k][j];
```

Subodh Kumar

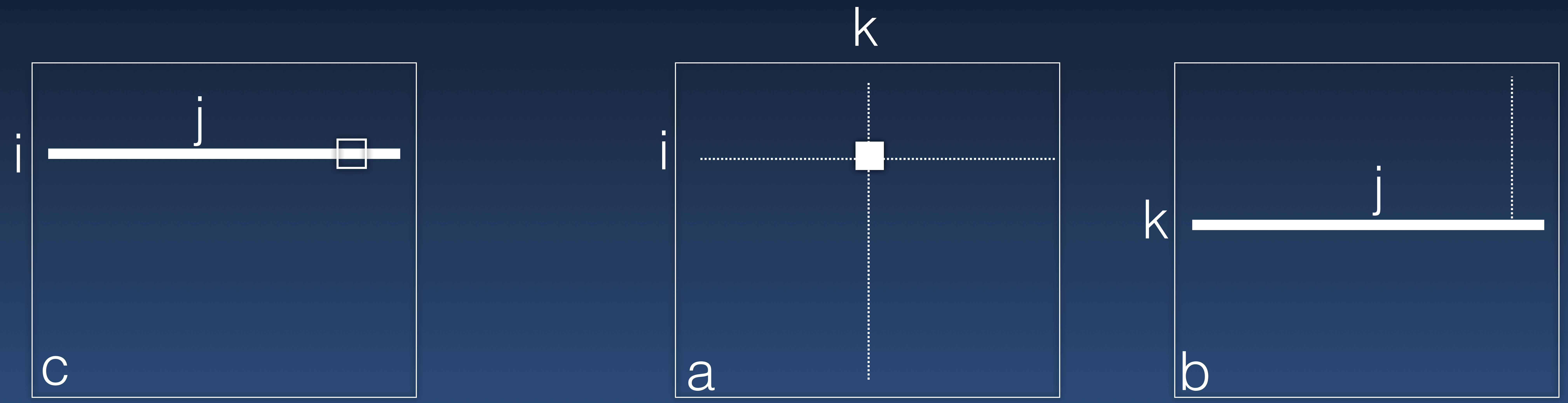
Matrix Multiplication




```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      c[i][j] += a[i][k]*b[k][j];
```

```
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++)
      c[i][j] += a[i][k]*b[k][j];
```

Matrix Multiplication

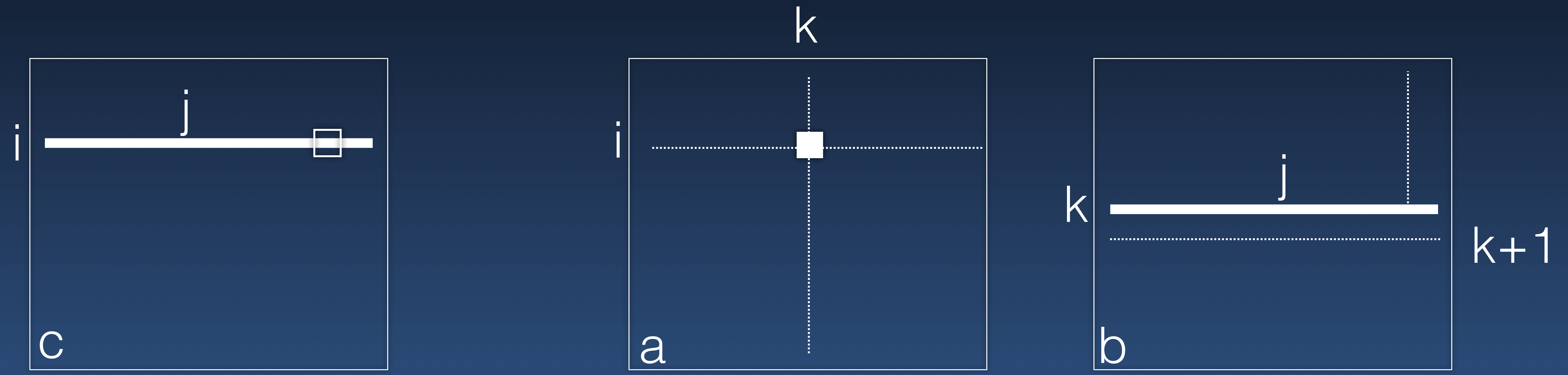



```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k]*b[k][j];
```

```
for (int i = 0; i < N; i++)
   for (int k = 0; k < N; k++)
      for (int j = 0; j < N; j++)
         c[i][j] += a[i][k]*b[k][j];
```

Subodh Kumar

Matrix Multiplication

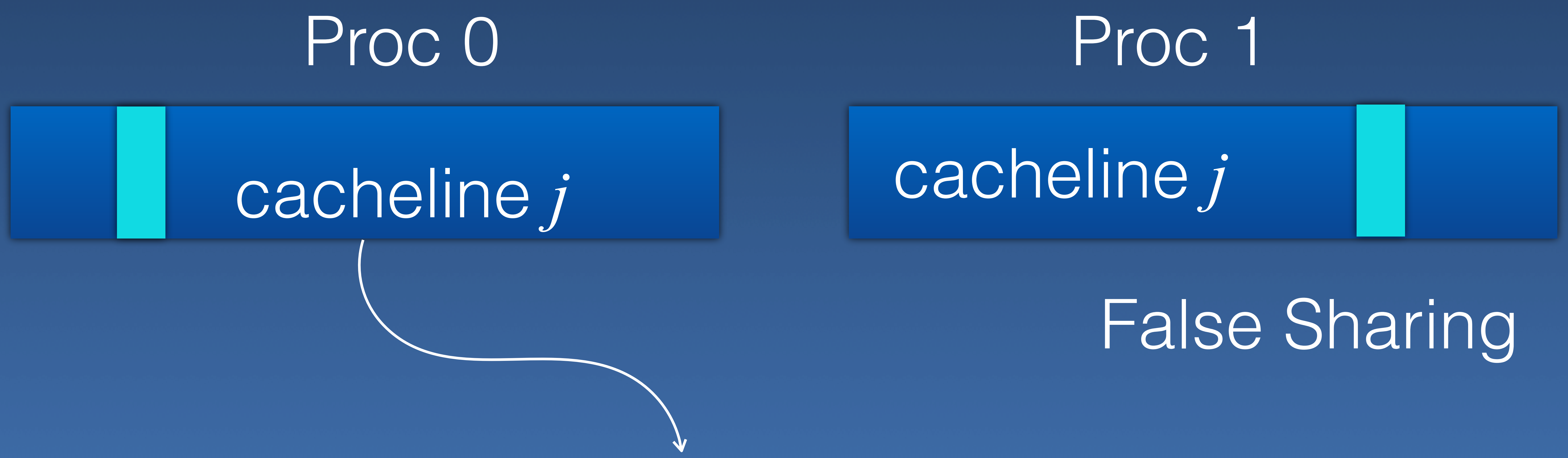



```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      c[i][j] += a[i][k]*b[k][j];
```

```
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++)
      c[i][j] += a[i][k]*b[k][j];
```

Subodh Kumar

- Coherent caches
  - ➡ Writes in on cache are updated in all cache copies
- Caches operate on line granularity
  - ➡ To write one item, may need to update the other items first



False Sharing

Recap

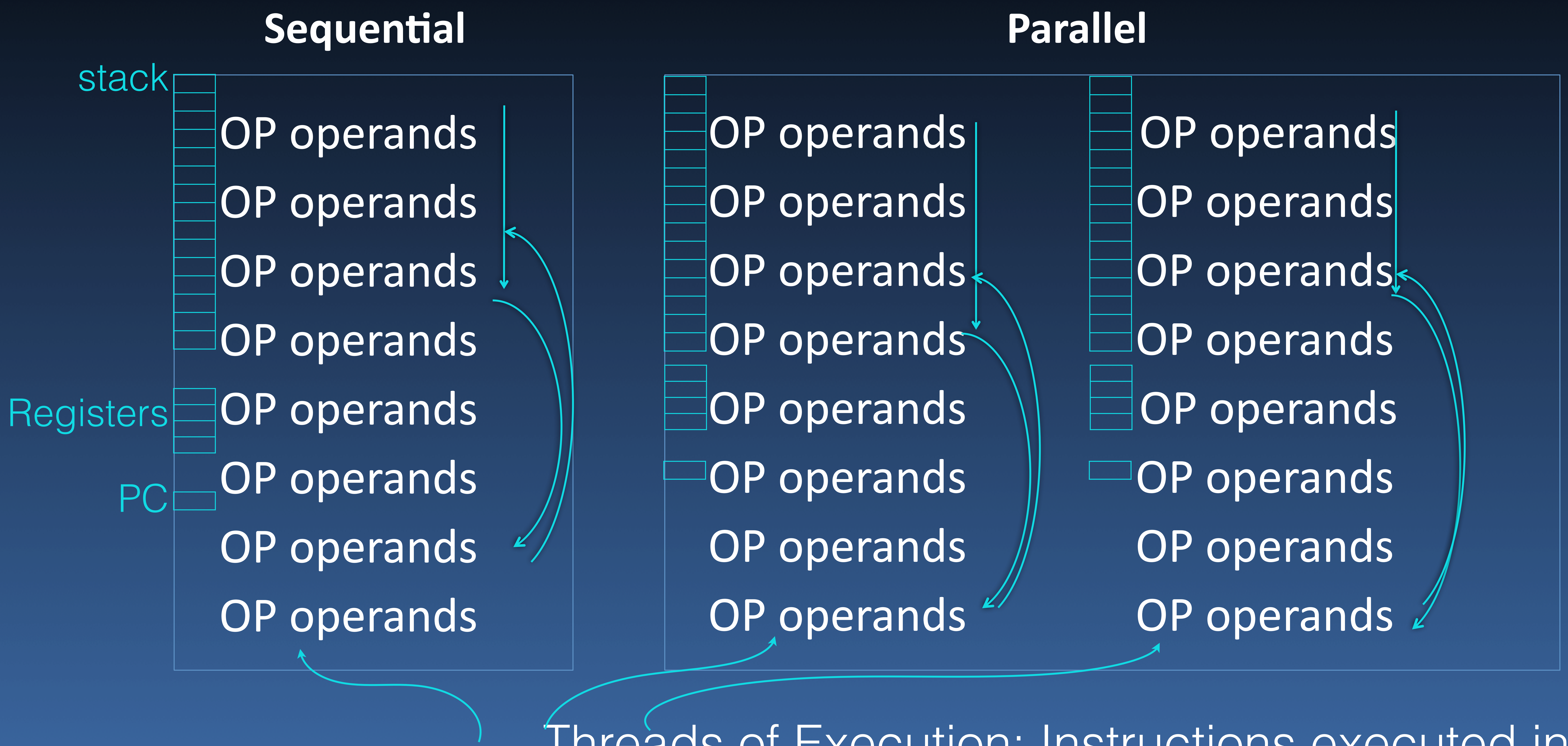
- OS Basics
  - ➡ Process, Threads, Address space, Context, System call overheads
- Role of compilers and Architecture
  - ➡ Code elimination, Memory operation elimination
  - ➡ Parallel, Pipelined, Out-of-order execution
- Memory Bottlenecks
  - ➡ Virtual address + caches, Memory latency
  - ➡ Cache lines, cache consistency, false sharing

Performance measurement

- Multiple Interacting (Sequential) Executions
  - ➡ May or may not share a common clock
  - ➡ May or may not directly share a common state
    - e.g., variables, OS
    - State may be distributed among ‘agents’
      - Inevitably have transient periods when they are ‘out of sync’

Threads of Execution

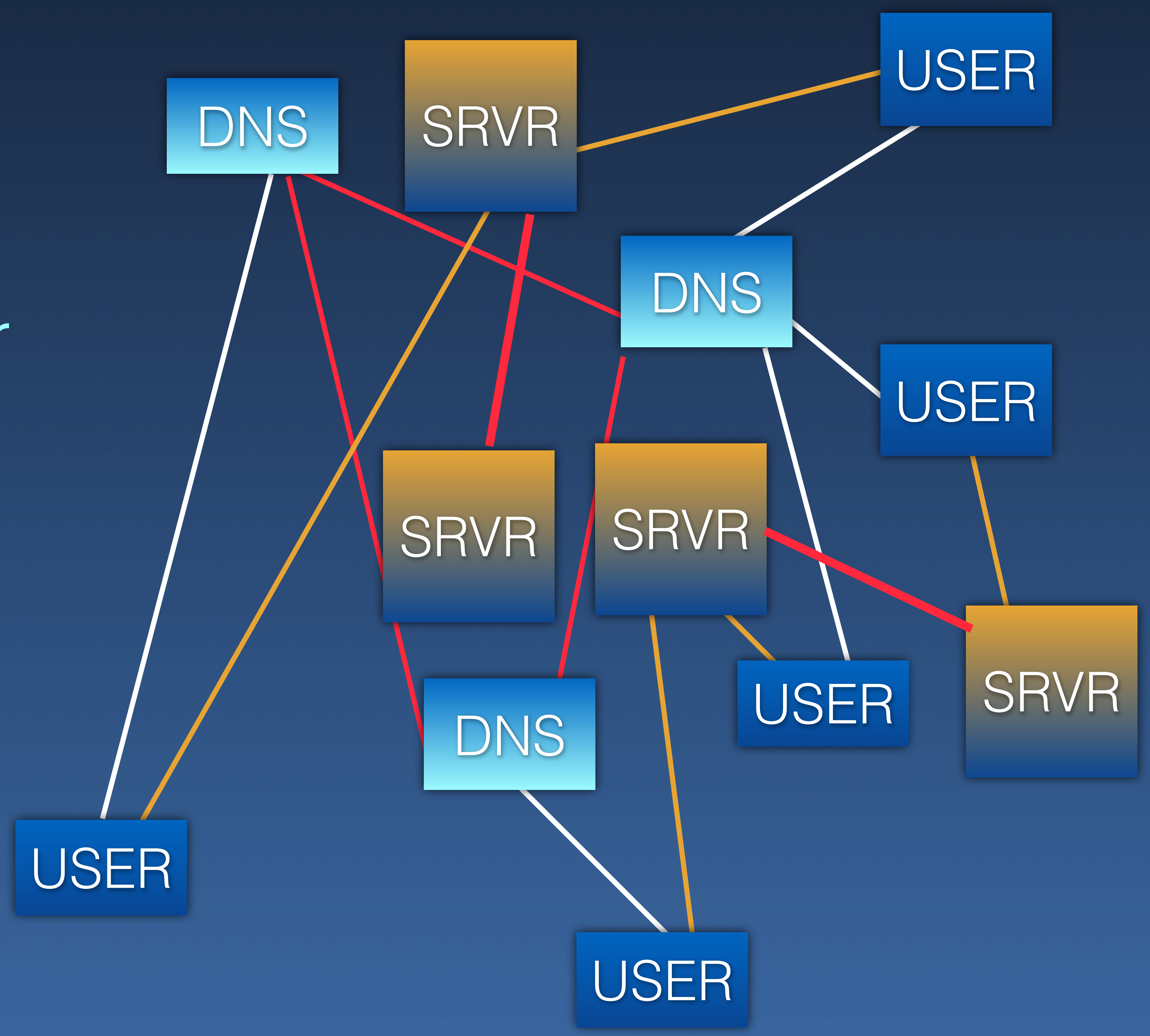
Threads of Execution



Threads of Execution: Instructions executed in order

Subodh Kumar

- Parallel
  - ➡ Focus on doing many things at the same time
- Distributed
  - ➡ How the multiple things interact with each other
- Concurrent
  - ➡ Not ordered



Parallelize Version 0

- Compiler flag
  - ➡ gcc -O2 -ftree-vectorize ### -mavx512f -fopt-info-vec-optimized
  - ➡ Or, gcc -O3
- Embarrassingly Parallel
  - ➡ Subdivide work in p equal parts, given p processors
  - ➡ forall i in [0,p)
    - Do Workpart(i)

Parallelize Version 0

- Compiler flag
  - ➡ gcc -O2 -ftree-vectorize ### -mavx512f -fopt-info-vec-optimized
  - ➡ Or, gcc -O3
- Embarrassingly Parallel
  - ➡ Subdivide work in p equal parts, given p proce
  - ➡ forall i in [0,p)
    - Do Workpart(i)

```
x = [W(i) for i in range(p)]
```

```
make -j

#!/bin/bash
for ((i=0; i< $count; i++));
do
    grep $pattern file$i &
done
```

Are files read in parallel?

Hello OpenMP

```
#include <omp.h>

#pragma omp parallel // num_threads(8)
{
    tid = omp_get_thread_num();
    dowork(input, tid);
}
```

```
> g++ file.c -fopenmp
```

Why Parallel

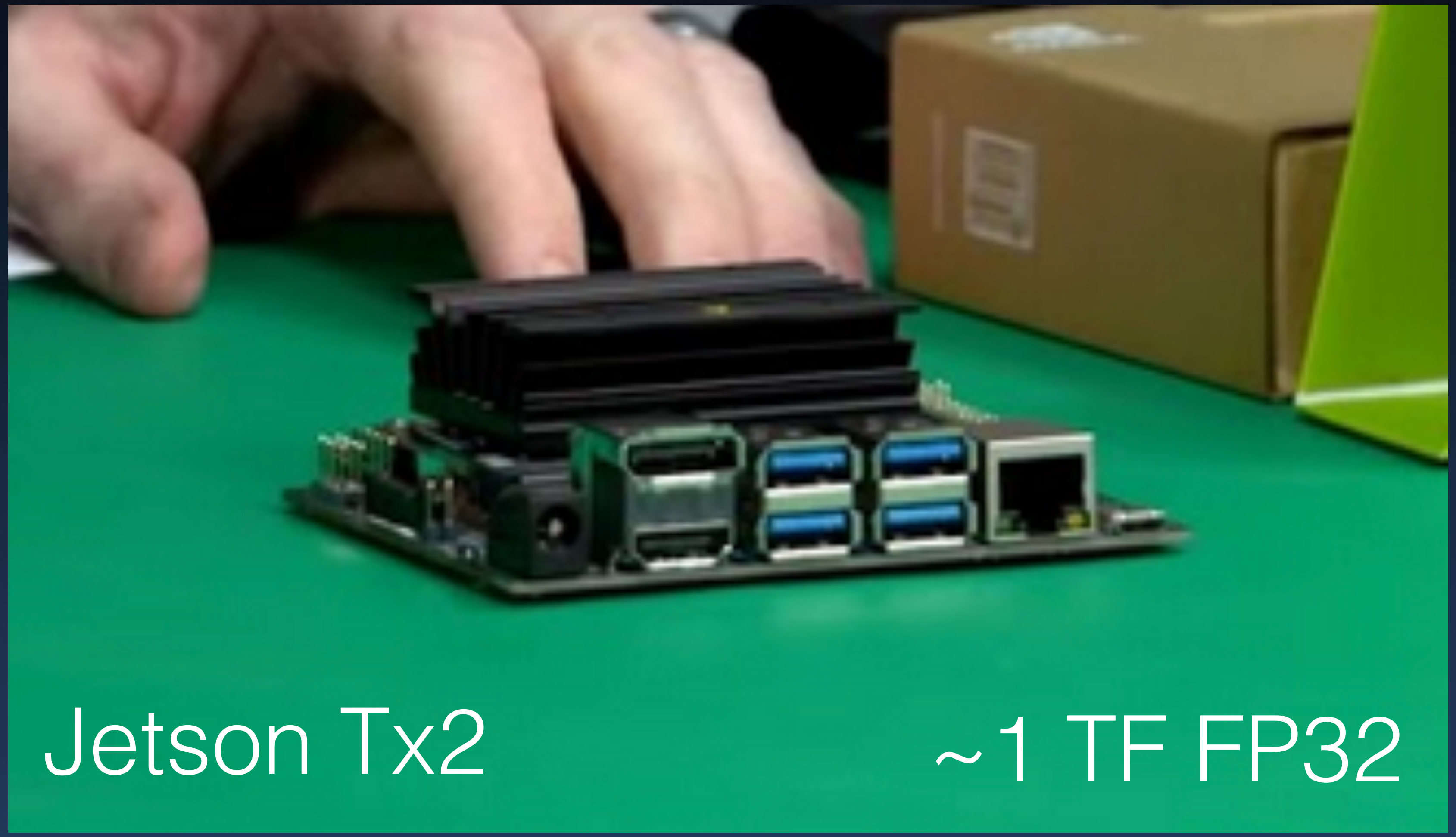
- Can't clock faster ⇒ Do more per clock
  - ➡ Execute many simple instructions on many cores
  - ➡ Simpler operations are more general
  - ➡ Complex operations require hardware coordination across the chip

$\boxed{\text{Significant software orchestration}}$

- Not just compute more things
  - ➡ Focus may be needed on parallelizing data access (Memory, IO)
  - ➡ Multiple processors can access memory in parallel, disrupt caches

Supercomputing?

~ 40 years ago



4x 64-bit Vector processor, ~117 MHz
~400 MFLOPS (peak)
128 MB RAM
(USD 20 MILLION)

The New York Times

India and U.S. Agree On Supercomputer Sale

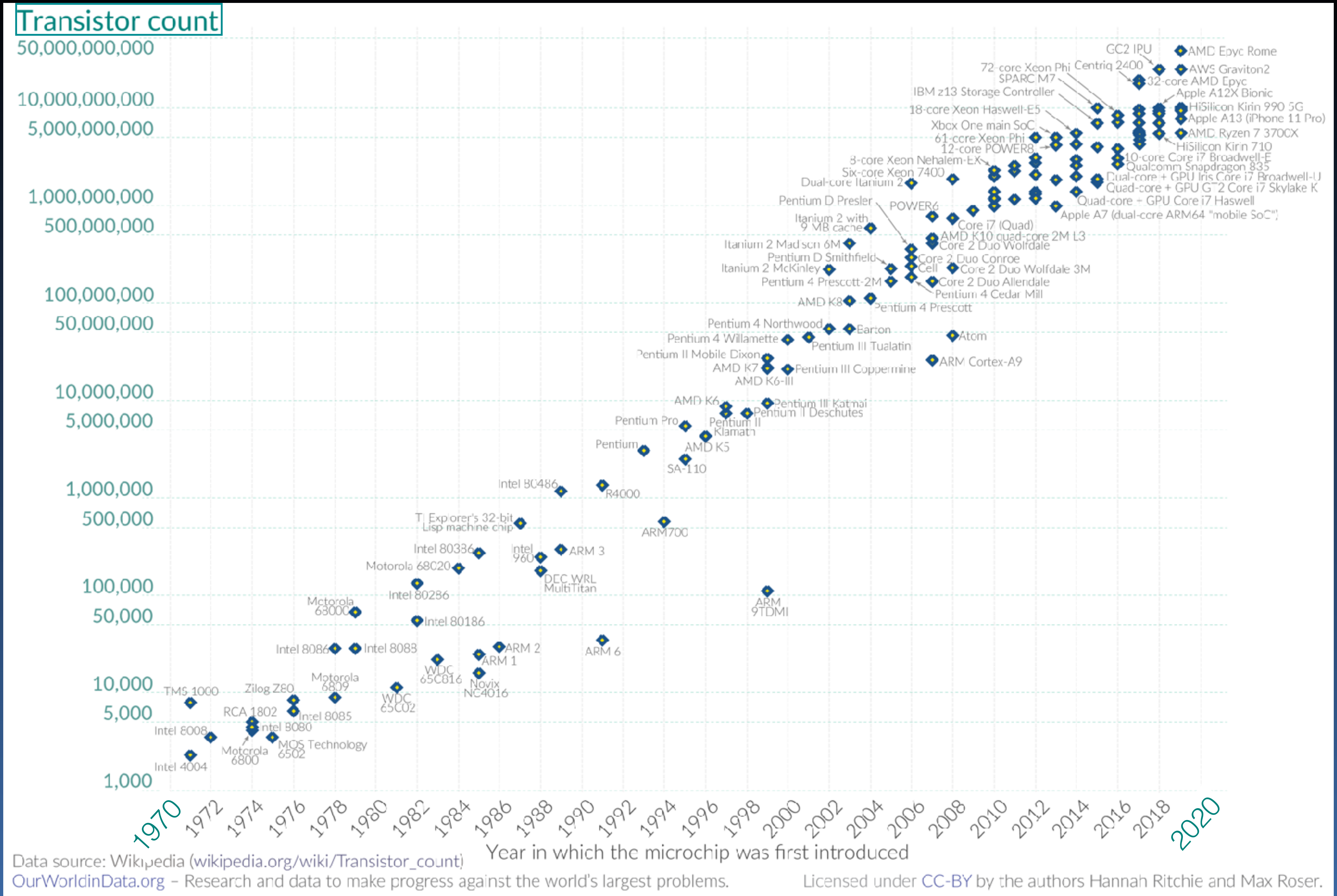
Give this article

By Steven R. Weisman, Special To the New York Times

Oct. 9, 1987

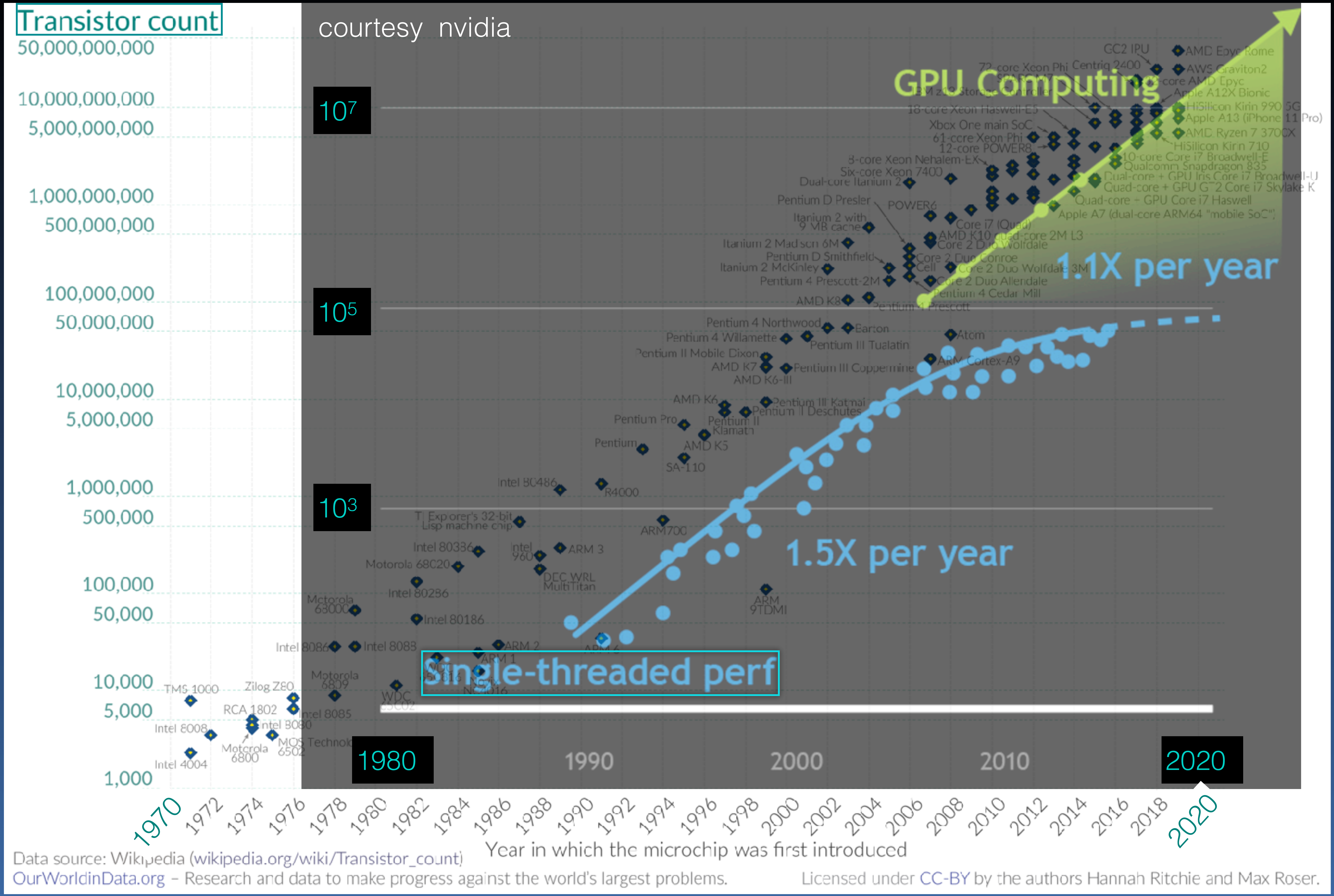
Subodh Kumar

Moore's Law



Subodh Kumar

Moore's Law



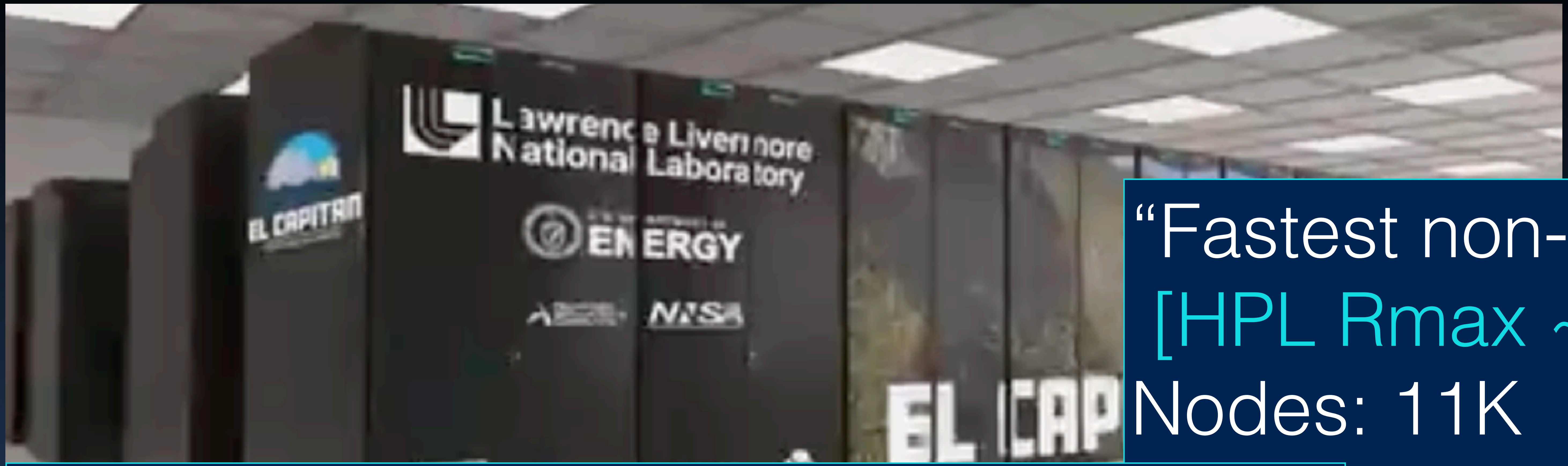
Subodh Kumar

Supercomputer

“Fastest non-distributed computer”
[HPL Rmax ~2EF: 1742000000000000000]
Nodes: 11K
Cores: 11 mil (96c CPU, 4x GPU/node)
Memory: 5.5 PB of Main memory
Interconnect: Multi-200 Gb/s port Slingshot
- Racks: 87 cabinets
- Space: 7500 SqFt
- Power consumption: 30 MW

El Capitan

Supercomputer



“Fastest non-distributed computer”

[HPL Rmax ~2EF: 1742000000000000000]

Nodes: 11K

11 mil (96c CPU, 4x GPU/node)

ry: 5.5 PB of Main memory

onnect: Multi-200 Gb/s port Slingshot

ks: 87 cabinets

pace: 7500 SqFt

ower consumption: 30 MW

El Capitan

Arm CPU (Grace)

144 core CPU (has vector processing)

Peak Flops: 7.1TF (DP)

Memory BW: 1 TB/s

Intel CPU (Granite Rapid)

2 GHz x128 cores (~16 TFlop)

+ 2x AVX-512 FMA units

Maximum Memory Speed: 8.8 GHz

Memory Channels: 12

Memory bandwidth: ~800 GB/s

Nvidia GPU (H100)

16896+ FP32 Cores: 34 TF (DP)

989 TF (FP16) with Tensor Core

GPU Memory Bandwidth: 3.35 TB/s

Network: NVLink 900 GB/s

Kumar