Åbo Akademi

Parallel Computing

IT00CD91-3005, Spring 2025

Lecture 1: Course Overview and Parallel Hardware

by Alireza Olama

Hello World!

Teacher: A l i r e z a O l a m a

Lecturer & Postdoctoral Researcher | Information Technology Department | Åbo Akademi University (ÅAU)

Office: Vaasa, Finland

Email: Alireza.Olama@abo.fi

Github: https://github.com/Alirezalm

Research Areas

- Parallel & High – Performance Computing (HPC)
- Distributed Machine Learning
- Distributed Optmization

Projects:

- **PsFIT**: A framework for distributed sparse training
- **SCOT**: A distributed optimization solver tailored for sparse convex optimization.

SCOT

Sparse Convex Optimization Toolkit

PSFIT

Course Logistics

Course Info

The course materials are available on **Moodle** as ‘Parallel Programming Programming 2025’.

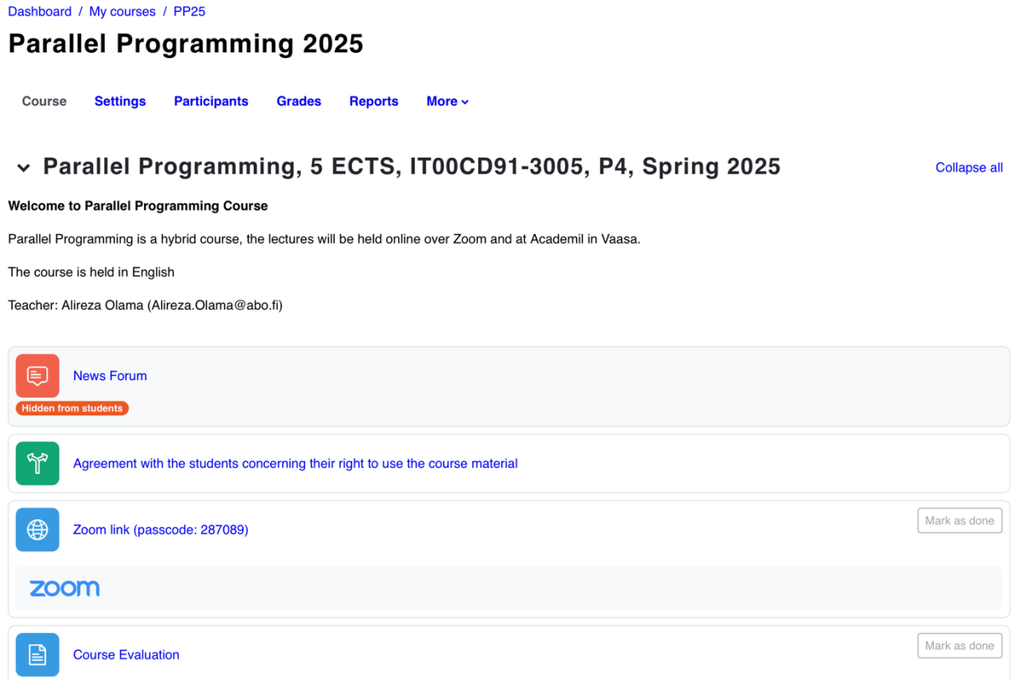
Enroll at: https://moodle.abo.fi/course/view.php?id=12998 .

Contains lecture slides, videos, assignments, and final projects. projects.

The course is held in English.

Send me an email if you are not registered for the course in Peppi. Peppi.

The lectures will take place at Academil in Vaasa in room B0215/216 B0215/216



Course Info

The classes start on **March 24th** and ends on **May 26th** (10 Lectures)

Lecture times: **Mondays 12.30 to 14.00.**

The lectures are streamed over **Zoom** and will be recorded and published on Moodle and YouTube.

The course materials will be open-sourced and published on on the course website (to be informed later)

Course Goals

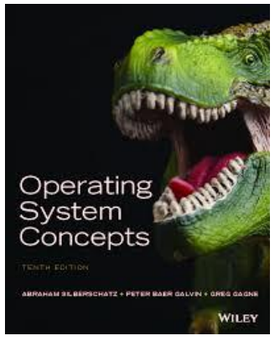
Main goal: Learning to write code that runs very fast on **modern architectures**

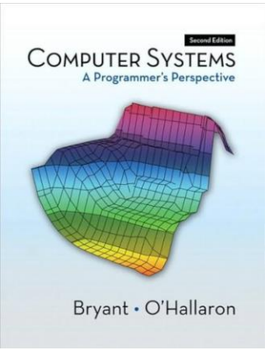
How? Design and develop programs that do lots of **independent things** in parallel

Apply parallel computing techniques to modern applications, such as **optimization**, **machine learning**, and **learning**, and **numerical simulations**

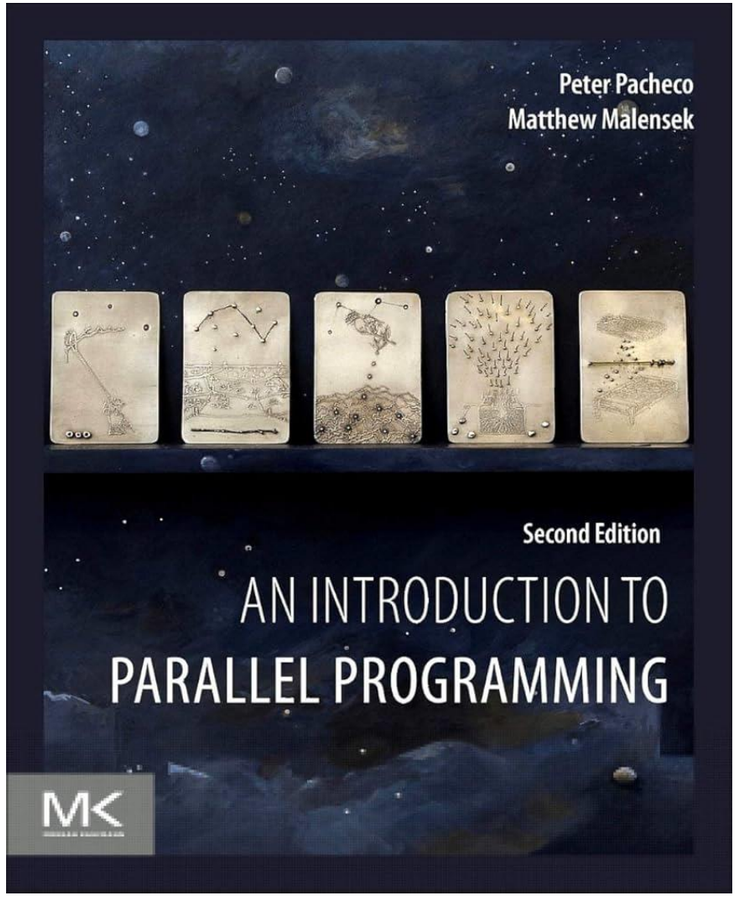
Prerequisites

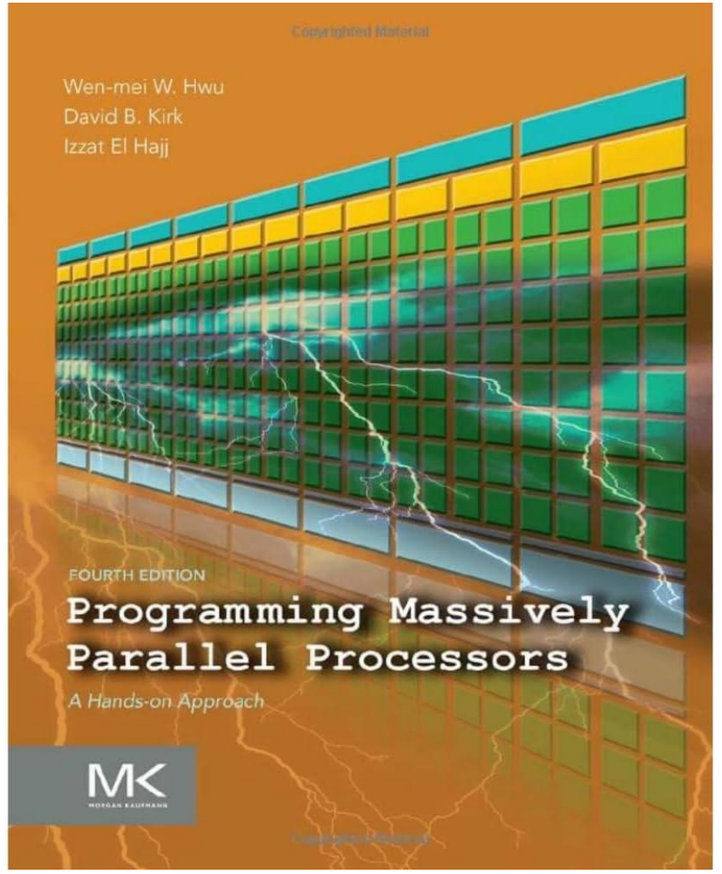
- **Solid C/C++ programming background**
  - Basic data structures
  - Pointers
  - Memory Management
- **Familiarity with Operating System (OS) concepts**
  - Processes
  - Threads
  - Inter-process communication
  - Networking
- **Machine Structure**
  - CPU– Memory model
  - Program execution





Recommended Textbooks





Grading

- **Homework assignments (40 points)**
  - Five homework assignments each is worth 8 point
  - Students must submit homework two weeks after release.
  - Late assignments may incur penalties unless prior approval is granted.
  - **Individually**

- **Final Project (40 points)**
  - Project Proposal
  - Project Implementation
  - A 10-minute presentation
  - **Can be done in groups**
  - Github

- **Exam (20 points)**

Course Schedule

- **Lecture 1:** Introduction to The Course and Parallel Hardware Architecture
- **Lecture 2:** C++ Refresher and std::threads (Tentative)
- **Lecture 3:** OpenMP – Part I
- **Lecture 4:** OpenMP – Part II
- **Lecture 5:** Introduction to GPU Computing & CUDA (Architecture & Basics)
- **Lecture 6:** CUDA I
- **Lecture7:** CUDA II
- **Lecture 8:** MPI – Part I
- **Lecture 9:** MPI – Part II
- **Lecture 10:** Distributed Machine Learning

Why Parallel Computing?

Why Parallel Computing?

It is all about the **need for speed**

What is a parallel computer?

A parallel computer is a **collection of processing elements** that cooperate to solve solve problems **quickly**

What is a parallel computer?

We’re going to use multiple processing elements to get it

A parallel computer is a **collection of processing elements** that cooperate to solve problems **quickly**

What is a parallel computer?

We're going to use multiple processing elements to get it

A parallel computer is a **collection of processing elements** that cooperate to solve problems **quickly**

We care about performance, and we care about efficiency

How to speed up programs?

Vertical Scaling:

Increasing CPU clock frequency

Horizontal Scaling:

Increasing the number of processors – enabling parallelism

What is a computer program?

Here is a program written in C

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 1;

    for (int i = 0; i < 10; i++) {
        x = x + x;
    }

    printf("%d\n", x);
    return 0;
}
```

What is a program? (from processor’s perspective)

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 1;

    for (int i = 0; i < 10; i++) {
        x = x + x;
    }

    printf("%d\n", x);
    return 0;
}
```

What is a program? (from processor’s perspective)

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 1;

    for (int i = 0; i < 10; i++) {
        x = x + x;
    }

    printf("%d\n", x);
    return 0;
}
```

What is a program? (from processor's perspective)

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 1;

    for (int i = 0; i < 10; i++) {
        x = x + x;
    }

    printf("%d\n", x);
    return 0;
}
```

Compiler

```
_main:
100000f10:    push    %rbp
100000f11:    mov     %rsp, %rbp
100000f14:    sub     $32, %rsp               ; allocate 32 bytes on stack

100000f18:    mov     $0, -4(%rbp)            ; argc = 0 (dummy, not used here)
100000f1f:    mov     %edi, -8(%rbp)          ; store argc
100000f22:    mov     %rsi, -16(%rbp)         ; store argv

100000f26:    mov     $1, -20(%rbp)           ; int x = 1;
100000f2d:    mov     $0, -24(%rbp)           ; int i = 0;

.L_loop_start:
100000f34:    cmp     $10, -24(%rbp)          ; compare i with 10
100000f38:    jge     0x100000f5f             ; if i >= 10, jump to end of loop

100000f3e:    mov     -20(%rbp), %eax         ; eax = x
100000f41:    add     -20(%rbp), %eax         ; eax = x + x
100000f44:    mov     %eax, -20(%rbp)         ; x = eax

100000f47:    mov     -24(%rbp), %eax         ; eax = i
100000f4a:    add     $1, %eax                ; i++
100000f4d:    mov     %eax, -24(%rbp)         ; store updated i

100000f50:    jmp     0x100000f34             ; repeat loop

.L_loop_end:
100000f55:    leaq    0x100000f7b(%rip), %rdi ; load format string "%d\n"
100000f5c:    mov     -20(%rbp), %esi         ; 2nd argument: x
100000f5f:    mov     $0, %al                 ; clear AL before printf (varargs)
```

What is a program? (from processor's perspective)

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 1;

    for (int i = 0; i < 10; i++) {
        x = x + x;
    }

    printf("%d\n", x);
    return 0;
}
```

Compiler

```
_main:
100000f10:    push    %rbp
100000f11:    mov     %rsp, %rbp
100000f14:    sub     $32, %rsp               ; allocate 32 bytes on stack

100000f18:    mov     $0, -4(%rbp)                rgc = 0 (dummy, not used here)
100000f1f:    mov     %edi, -8(%rbp)              re argc
100000f22:    mov     %rsi, -16(%r                argv

100000f26:    mov     $1,
100000f2d:    mov

.L_loop_start:
100000f34:                                        are i with 10
100000f                                           if i >= 10, jump to end of loop

                                                  ; eax = x
1                                                 ; eax = x + x
100                                   rbp)        ; x = eax

100000f                 (%rbp), %eax              ; eax = i
100000f4a               $1, %eax                  ; i++
100000f4d:              %eax, -24(%rbp)           ; store updated i

100000f50:    jmp     0x100000f34               ; repeat loop

.L_loop_end:
100000f55:    leaq    0x100000f7b(%rip), %rdi ; load format string "%d\n"
100000f5c:    mov     -20(%rbp), %esi           ; 2nd argument: x
100000f5f:    mov     $0, %al                   ; clear AL before printf (varargs)
```

Who runs this instruction sequence?

What is a program? (from processor's perspective)

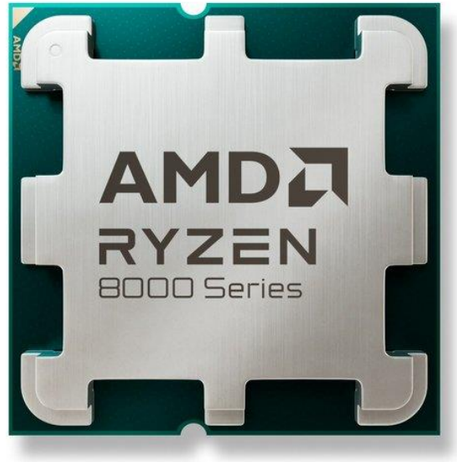
```c
#include <stdio.h>

int main(int argc, char** argv) {
    int x = 1;

    for (int i = 0; i < 10; i++) {
        x = x + x;
    }

    printf("%d\n", x);
    return 0;
}
```

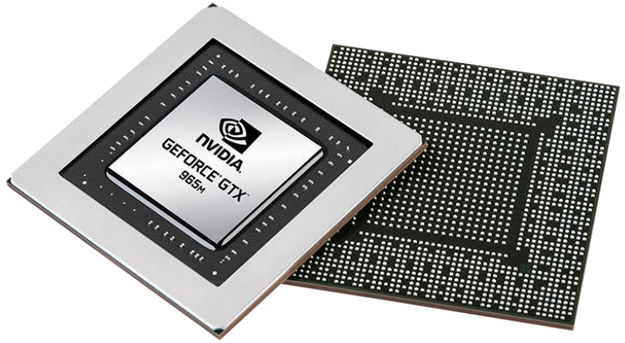
Compiler


```
_main:
100000f10:    push    %rbp
100000f11:    mov     %rsp, %rbp
100000f14:    sub     $32, %rsp               ; allocate 32 bytes on stack

100000f18:    mov     $0, -4(%rbp)                rgc = 0 (dummy, not used here)
100000f1f:    mov     %edi, -8(%rbp)              re argc
100000f22:    mov     %rsi, -16(%r                argv

100000f26:    mov     $1,
100000f2d:    mov

.L_loop_start:
100000f34:                                        are i with 10
100000f                                           if i >= 10, jump to end of loop

                                                  ; eax = x
1                                                 ; eax = x + x
100                                   rbp)        ; x = eax

100000f                (%rbp), %eax               ; eax = i
100000f4a              $1, %eax                   ; i++
100000f4d:             %eax, -24(%rbp)            ; store updated i

100000f50:    jmp     0x100000f34                ; repeat loop

.L_loop_end:
100000f55:    leaq    0x100000f7b(%rip), %rdi ; load format string "%d\n"
100000f5c:    mov     -20(%rbp), %esi         ; 2nd argument: x
100000f5f:    mov     $0, %al                 ; clear AL before printf (varargs)
```

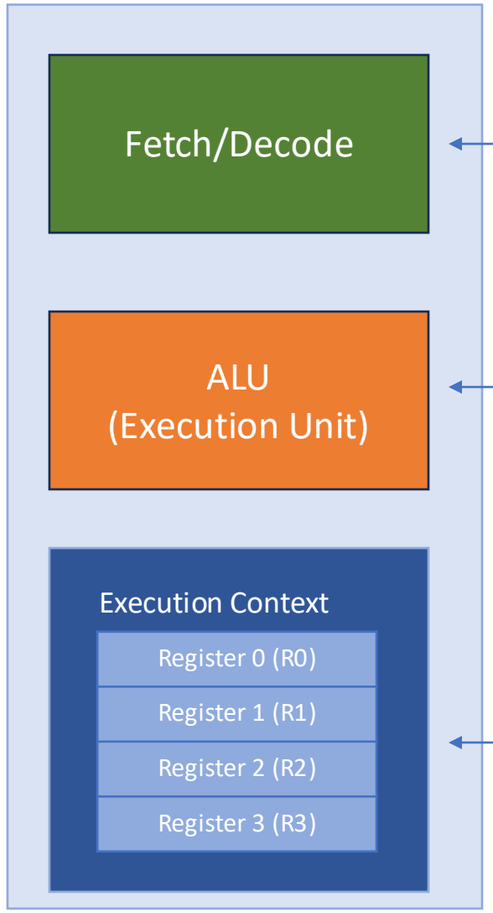
Who runs this instruction sequence?

What does a processor do?





A processor executes instructions



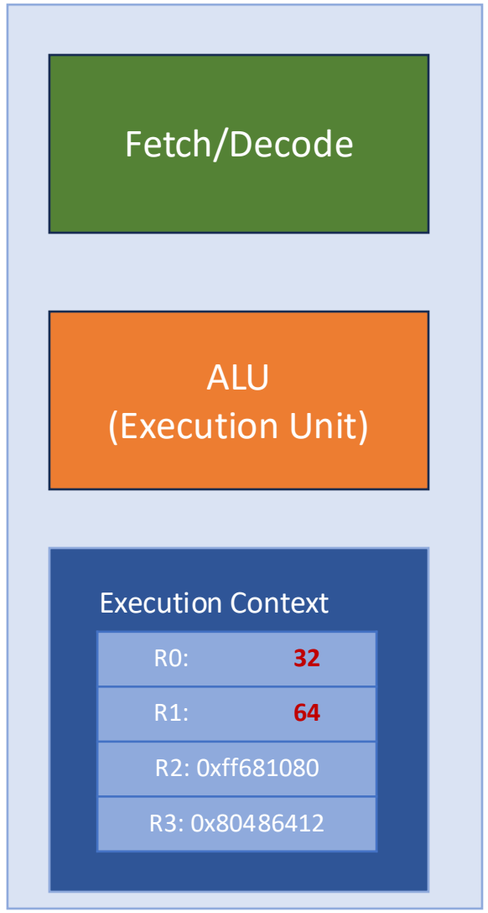
A Simple Processor

Determine what **instruction** to run next

Execution Unit: Performs the operation described by an instruction, which may modify values in the processor's registers or the computer's memory

Registers: Maintain program state – store value of variables used as inputs and outputs to operations

Example: add two numbers



A Simple Processor

Step 1:

Processor gets next program instruction from memory

```
add R0 <-- R0, R1
```

Step 2:

Get the operation inputs from registers

Content of R0 input to execution unit: **32**

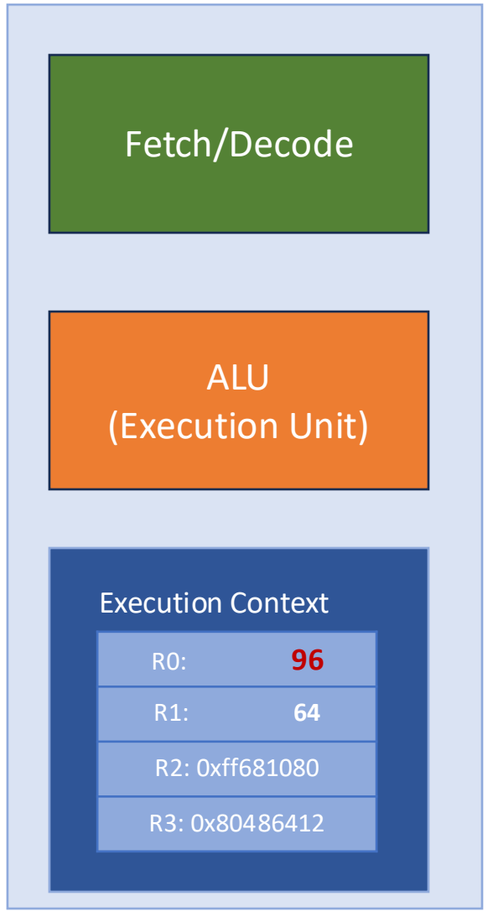
Content of R1 input to execution unit: **64**

Step 3:

Perform addition operation

Execution unit performs arithmetic, the result is **96**.

Example: add two numbers



A Simple Processor

Step 1:

Processor gets next program instruction from memory

```
add R0 <-- R0, R1
```

Step 2:

Get the operation inputs from registers

Content of R0 input to execution unit: **32**

Content of R1 input to execution unit: **64**

Step 3:

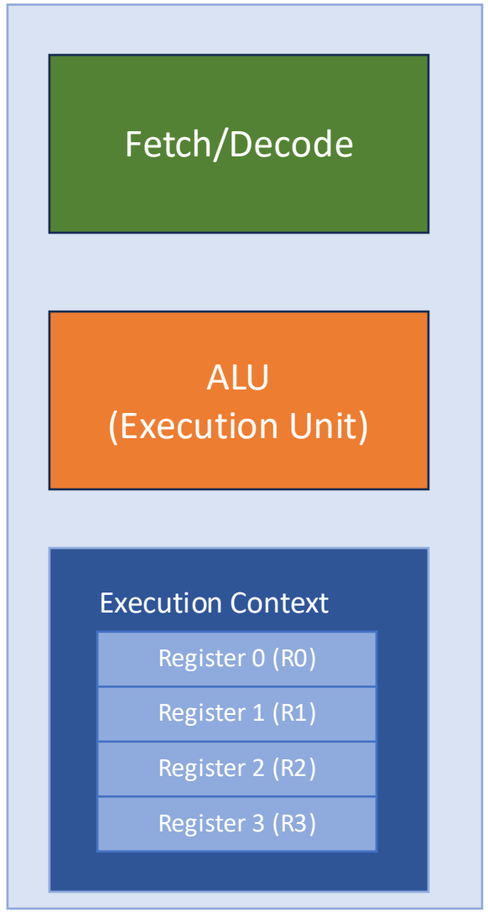
Perform addition operation

Execution unit performs arithmetic, the result is **96**.

Step 4:

Store result **96** to register R0

A processor executes instructions

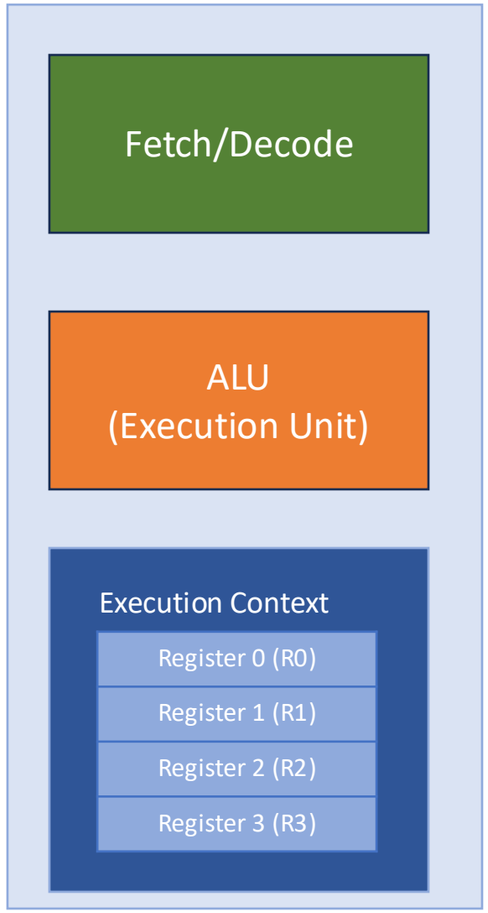


A Simple Processor

This simple Processor: **executes ONE instruction per clock**

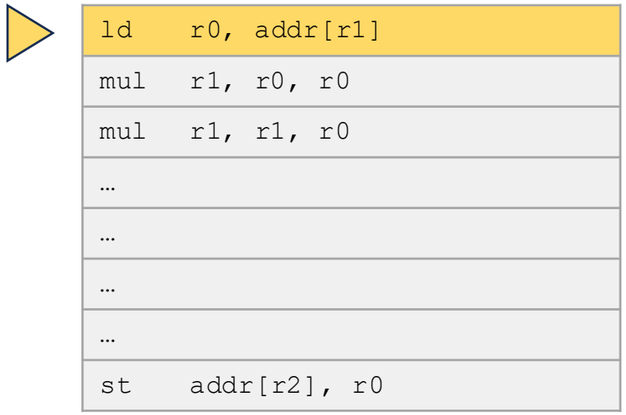
```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
…
…
…
…
st    addr[r2], r0
```

A processor executes instructions

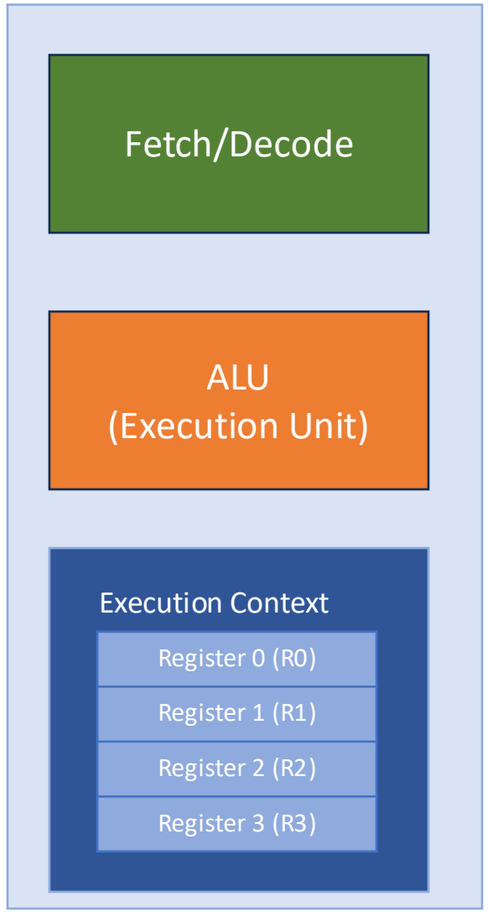


A Simple Processor

This simple Processor: executes **ONE** instruction per clock

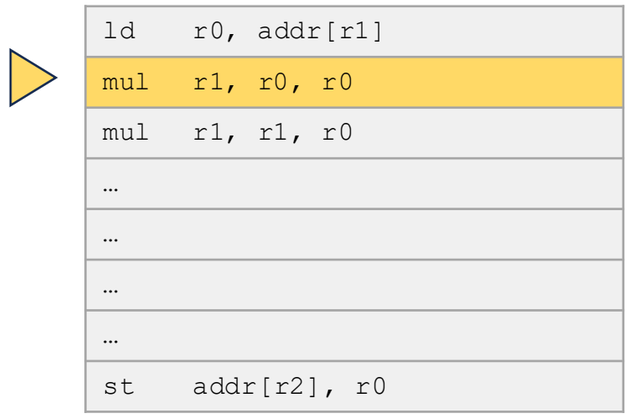


A processor executes instructions

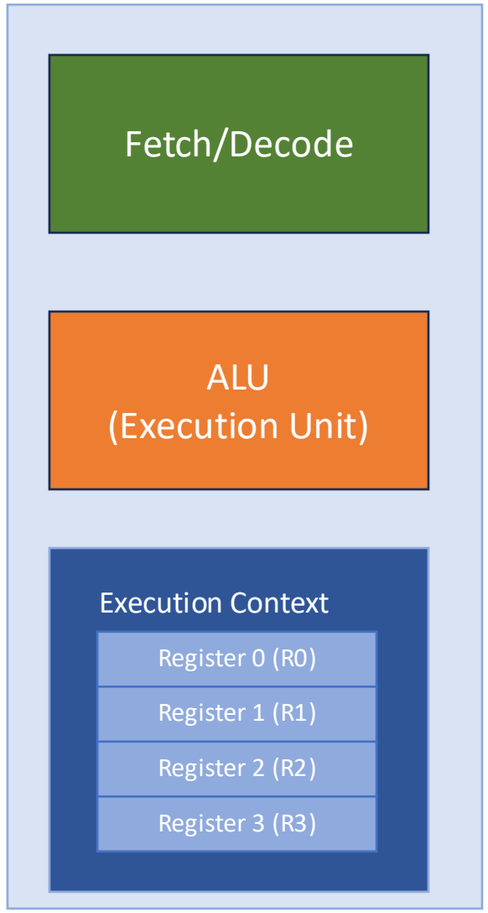


A Simple Processor

This simple Processor: executes **ONE** instruction per clock

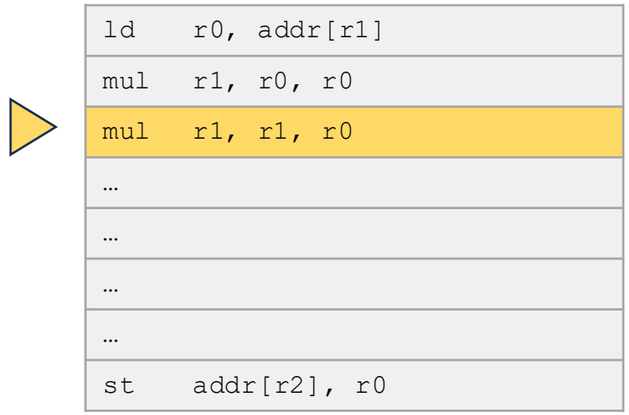


A processor executes instructions



A Simple Processor

This simple Processor: executes **ONE** instruction per clock



Review of how computers work

What is a computer program?

it is a list of instructions to execute!

Review of how computers work

What is a computer program?

it is a list of instructions to execute!

What is an instruction?

It describes an operation for a processor to perform
Executing an instruction typically modifies the computer's sate

Review of how computers work

What is a computer program?

it is a list of instructions to execute!

What is an instruction?

It describes an operation for a processor to perform
Executing an instruction typically modifies the computer's sate

What do we mean when we talk about a computer's "state"?

The values of program data, which are stored in a processor's registers or in memory.

Example

```
a = x*x + y*y + z*z
```

How many instructions is this line of code compiled to?

Example

```
a = x*x + y*y + z*z
```

How many instructions is this line of code compiled to?

```
R0 = x,      R1 = y,      R2 = z
```

```
mul   R0, R0, R0
mul   R1, R1, R1
mul   R2, R2, R2
add   R0, R0, R1
add   R3, R0, R2
```

This program has five instructions, so it will take five clocks to execute.

Can we do better?

Example

```
a = x*x + y*y + z*z
```

How many instructions is this line of code compiled to?

```
R0 = x,     R1 = y,     R2 = z
```

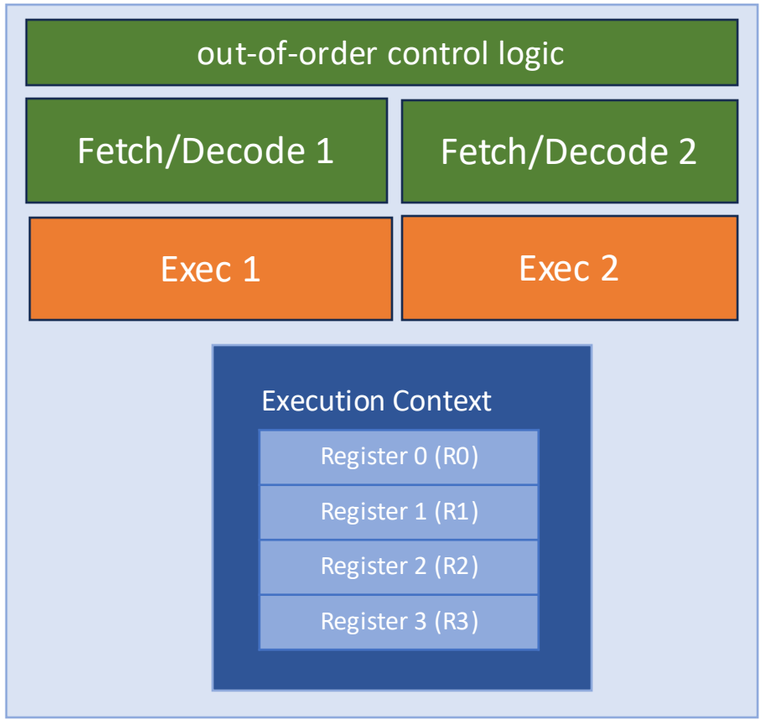
```
mul   R0, R0, R0
mul   R1, R1, R1
mul   R2, R2, R2
add   R0, R0, R1
add   R3, R0, R2
```

This program has five instructions, so it will take five clocks to execute.

Can we do better?

Superscalar execution: processor automatically finds **independent instructions** in an instruction sequence and **executes them in parallel** on multiple execution units!

Superscalar processor



This program has five instructions, so it will take five clocks to execute.

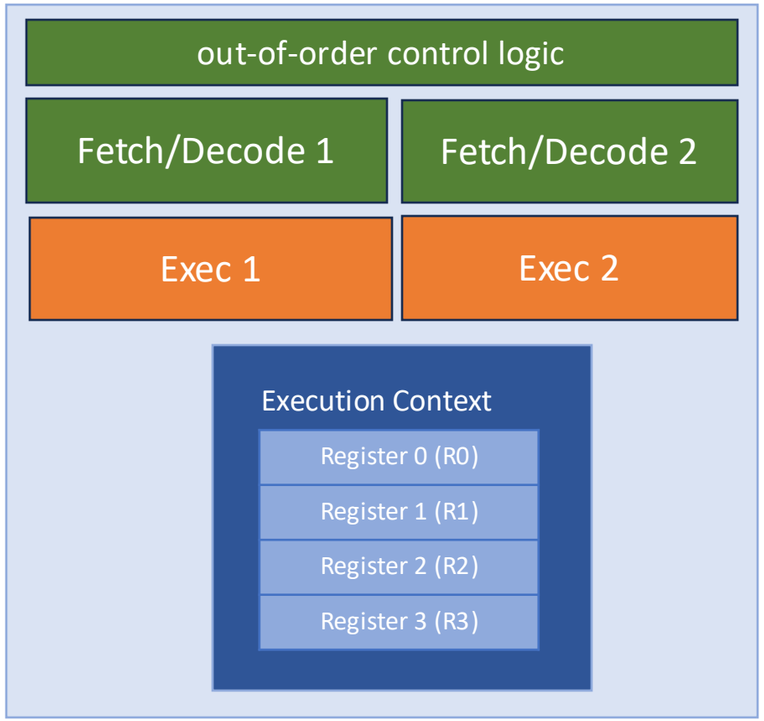
Can we do better?

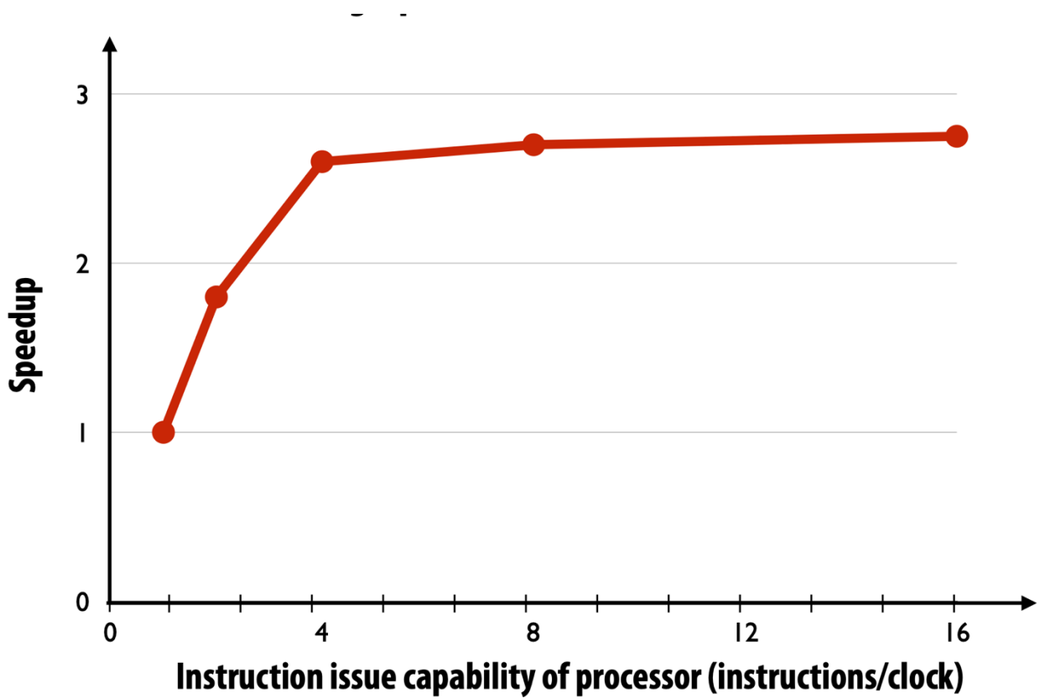
Superscalar execution: processor automatically finds **independent instructions** in an instruction sequence and **executes them in parallel** on multiple execution units!

This processor can decode and execute up to two instructions per clock.

Instruction Level Parallelism (ILP)

Instruction Level Parallelism



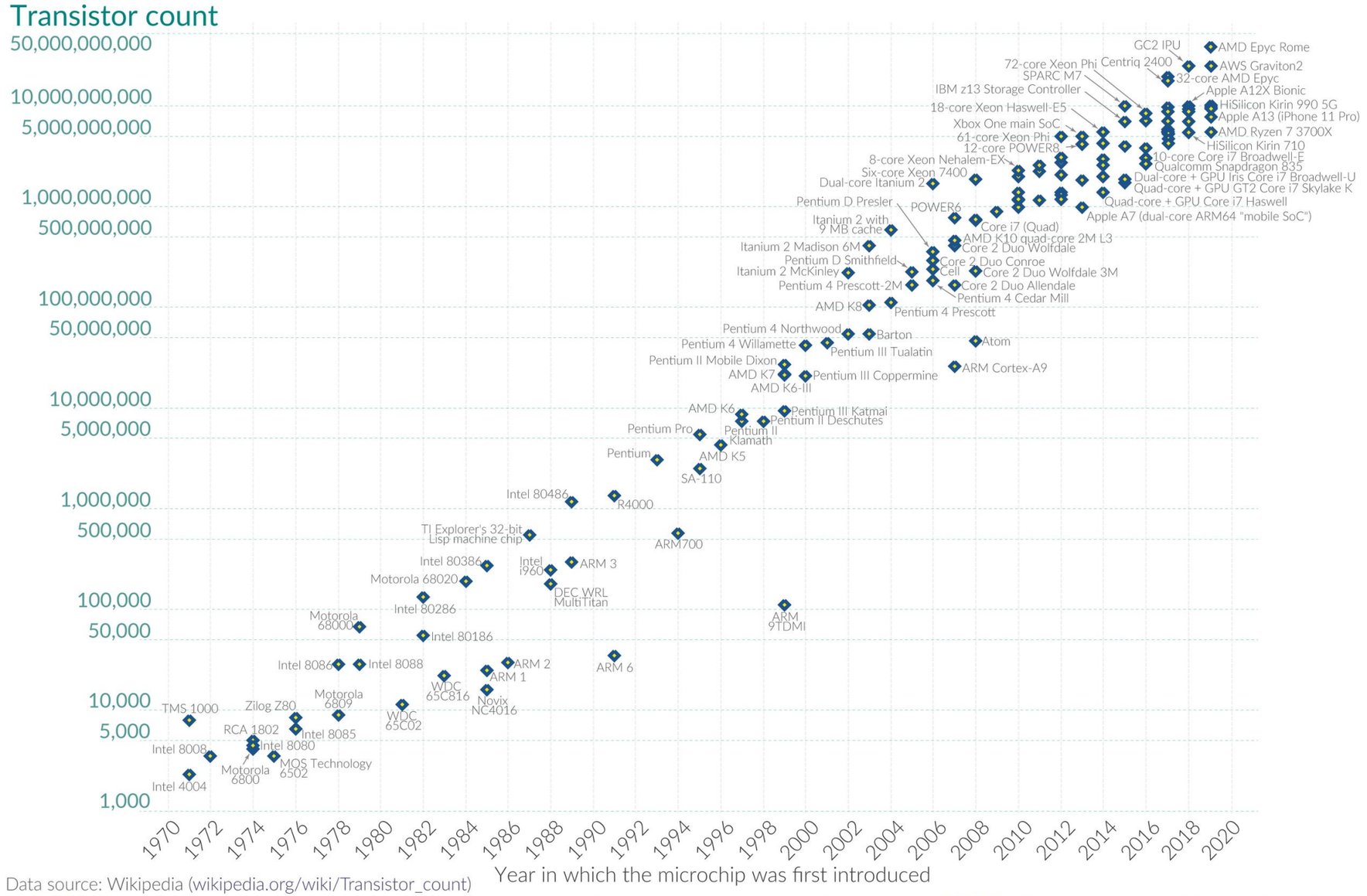


Why? Programs do not have lots of parallel instructions

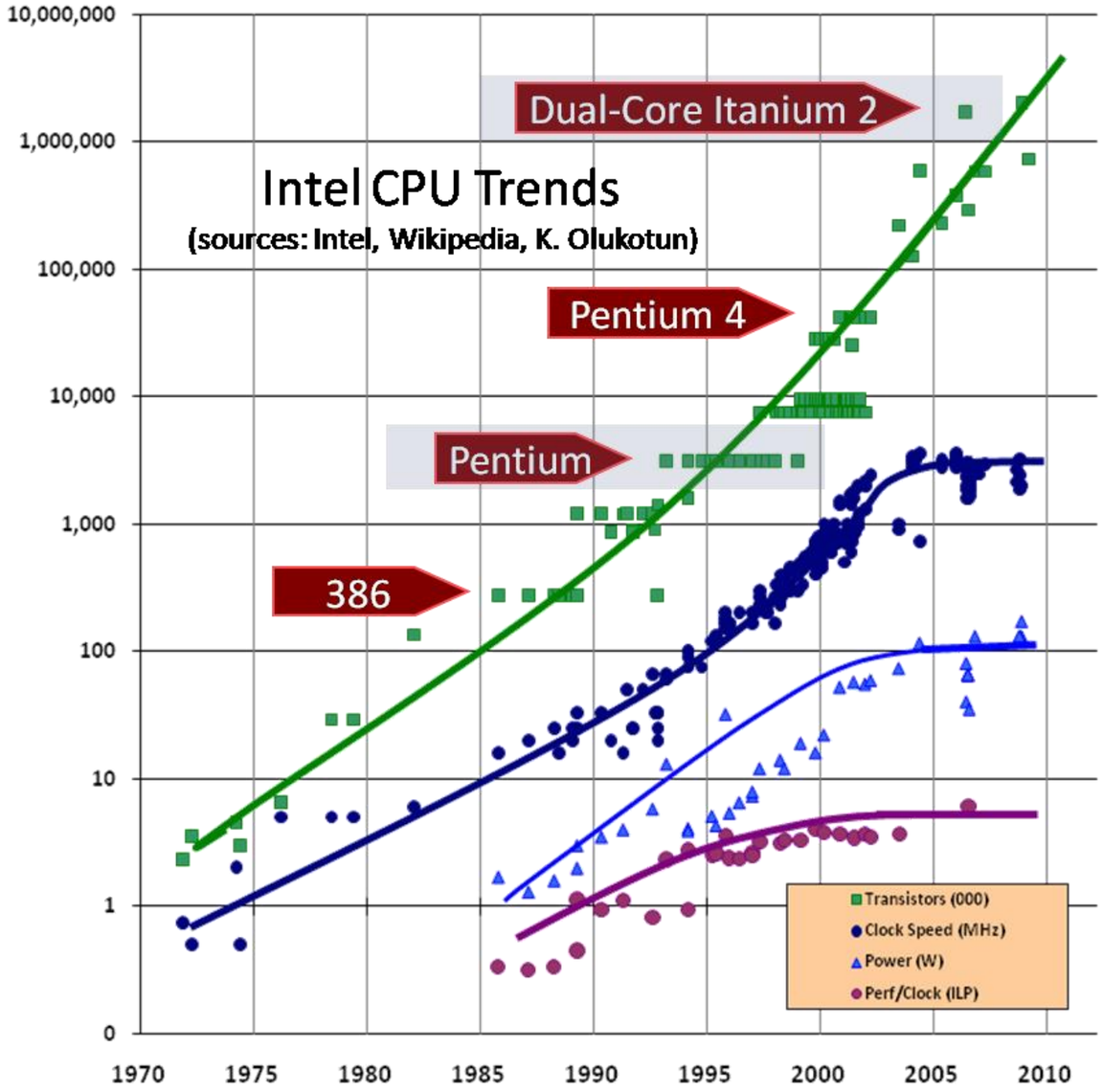
Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

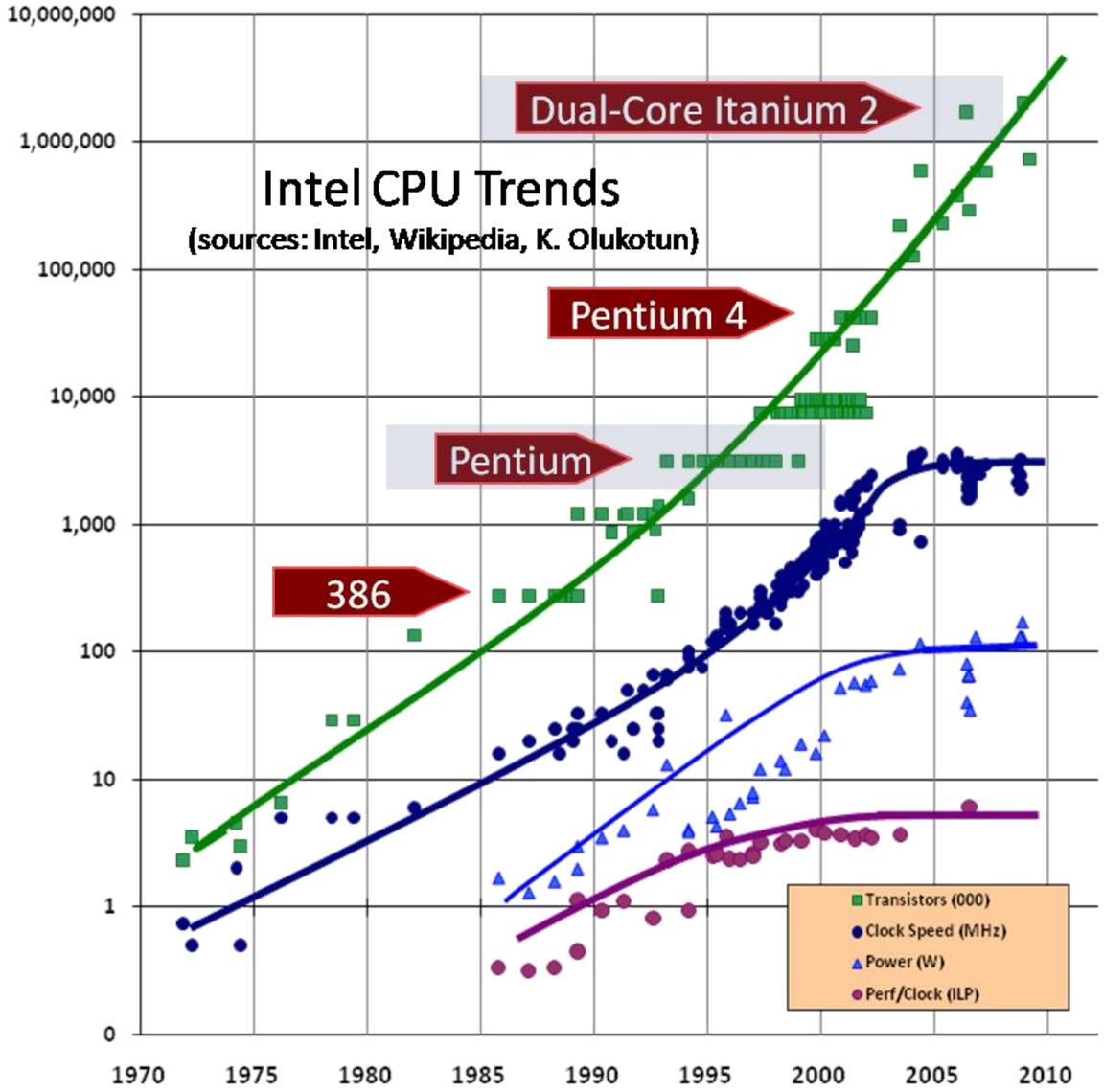


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)



We cannot turn more transistors into free parallelism.

We cannot make the instructions stream go faster, because we cannot increase the frequency of the machine.



We cannot turn more transistors into free parallelism.

We cannot make the instructions stream go faster, because we cannot increase the frequency of the machine.

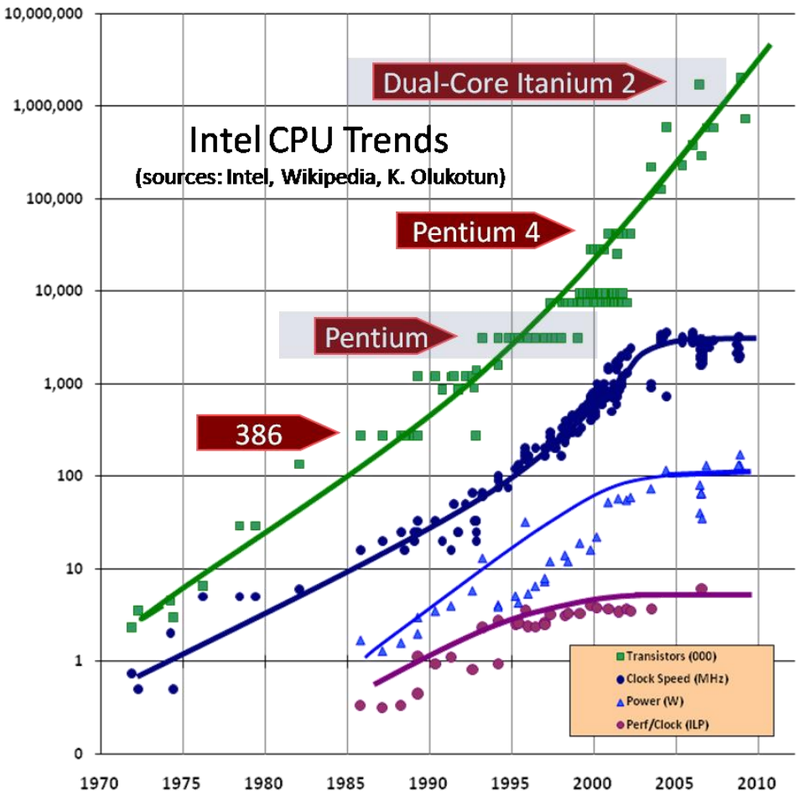
The power wall

Power consumed by a transistor is proportional to the frequency.

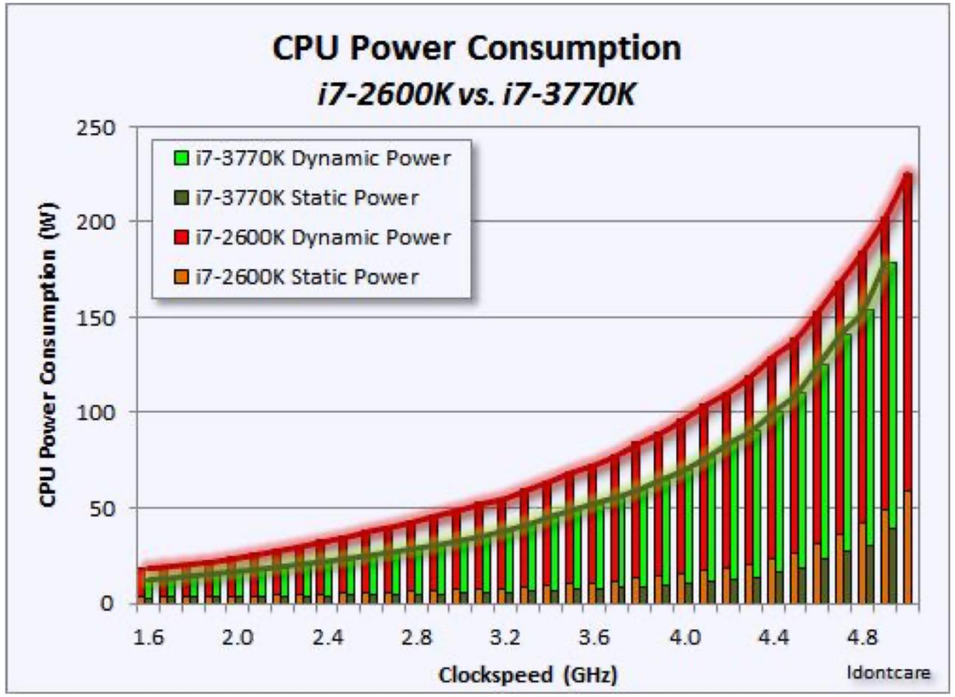
High power = high heat = the need for cooling

Power is a critical design constraint in modern processors.

NVIDIA RTX 4090 GPU: 450W
Standard microwave oven: 900W



Power as a function of clock frequency



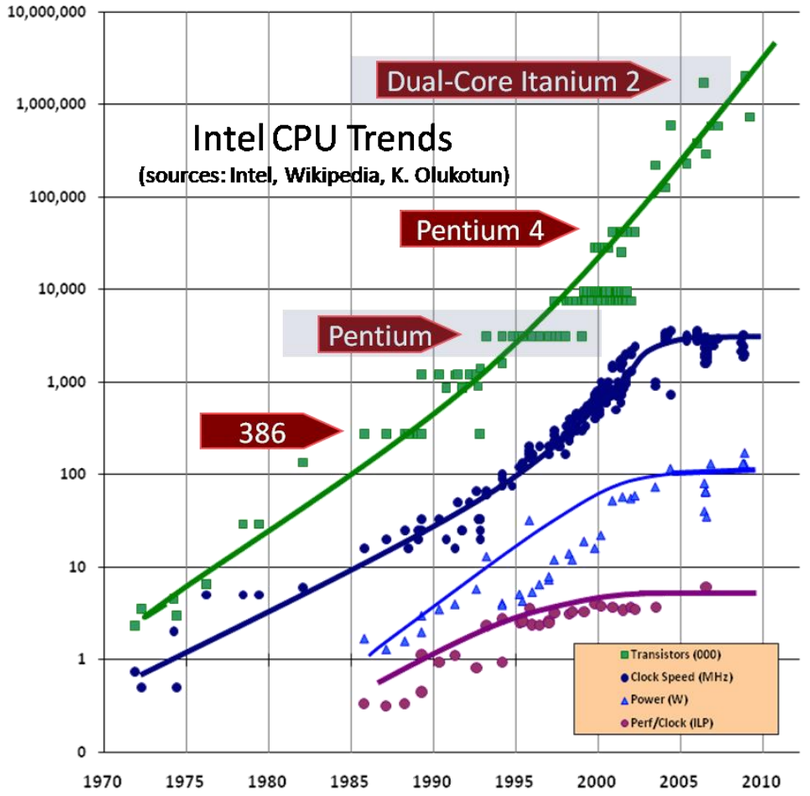
Single-core performance scaling

The rate of single-instruction stream performance scaling has decreased (almost to zero)

1. **Frequency scaling limited by power**
2. **ILP scaling tapped out**

Hardware architects are now building faster processors by adding more execution units that run in parallel.

Software must be written to be parallel to see performance gains. No more free lunch for software developers.



Parallel Computing: Faster Solutions

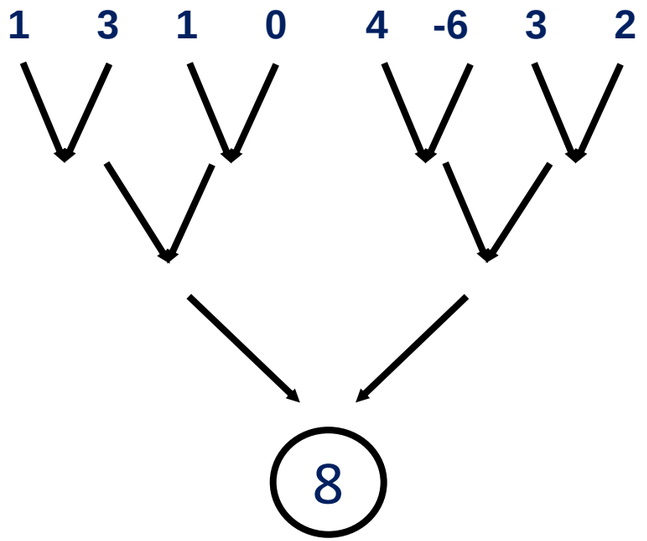
Using multiple processors in parallel to solve problems more quickly than with a single processor

Compute the prime factors of 1 billion numbers:

If we had 1 million processors…

Parallel Sum (Reduction)

Add n values



Serial: O(n)

Parallel: ?

Uses n processors!

Takes advantage of **associativity** in +

Modern Multi-Core processors

Multi-core processor

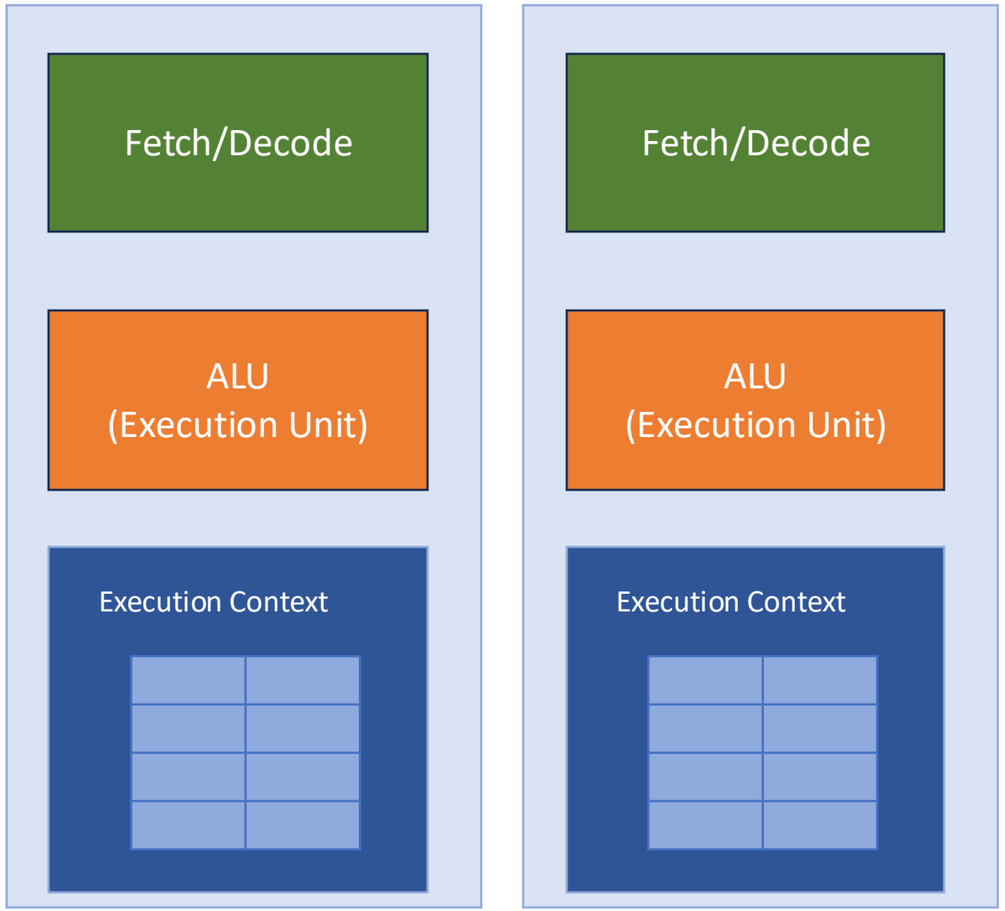
Rather than using transistors to increase clock speed to accelerate a single instruction stream

Use increasing transistor count to add more cores to the processor.

Multi-core processor

Rather than using transistors to increase clock speed to accelerate a single instruction stream

Use increasing transistor count to add more cores to the processor.



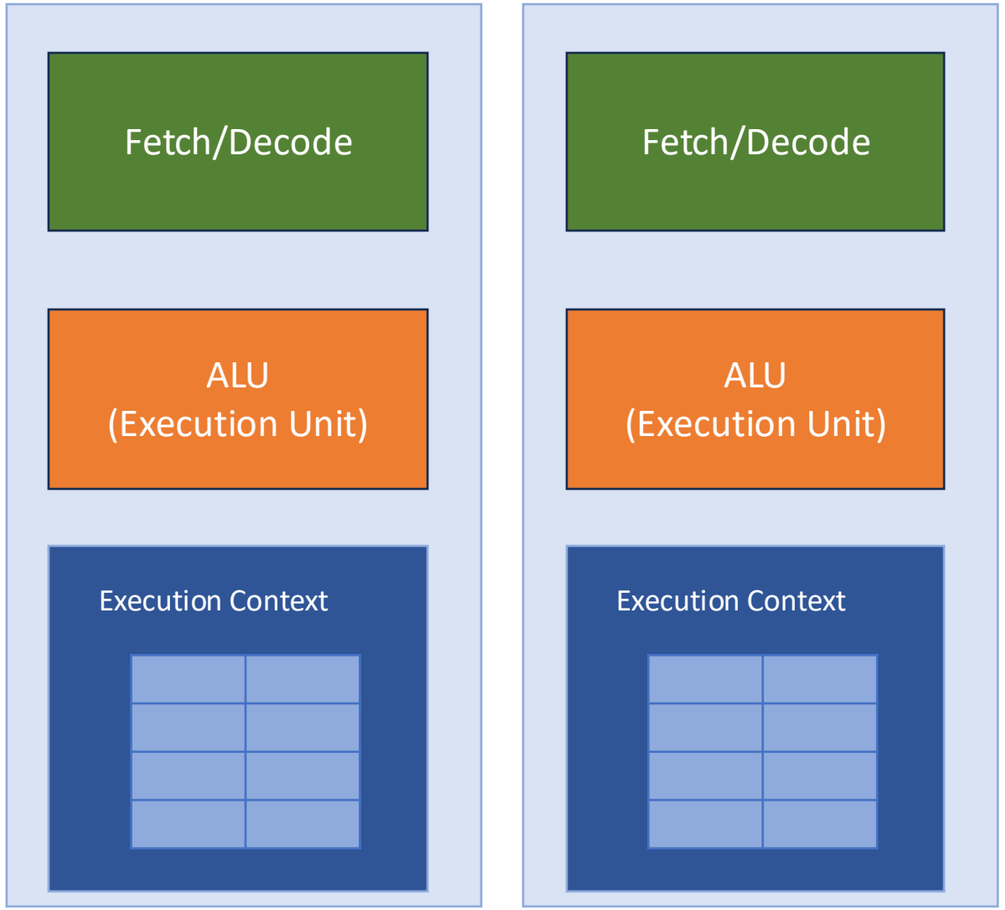
A dual-core processor: compute two elements in parallel (two instruction streams)

Multi-core processor

Rather than using transistors to increase clock speed to accelerate a single instruction stream

Use increasing transistor count to add more cores to the processor.

Simpler cores: each core may be slower at running a single instruction stream than original "fancy" core (e.g. 25% slower)



A dual-core processor: compute two elements in parallel (two instruction streams)

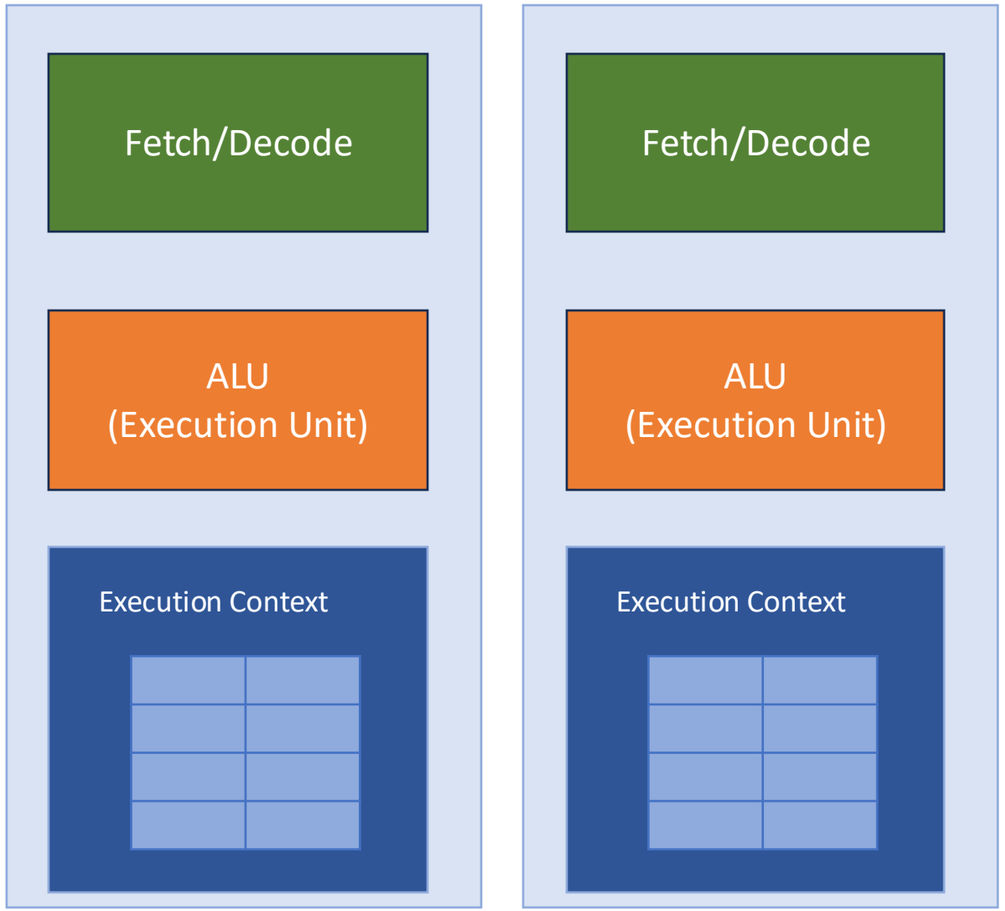
Multi-core processor

Rather than using transistors to increase clock speed to accelerate a single instruction stream

Use increasing transistor count to add more cores to the processor.

Simpler cores: each core may be slower at running a single instruction stream than original "fancy" core (e.g. 25% slower)

Programs by default will compile to an instruction stream that runs as one thread on one processor core.



A dual-core processor: compute two elements in parallel (two instruction streams)

Multi-core processor

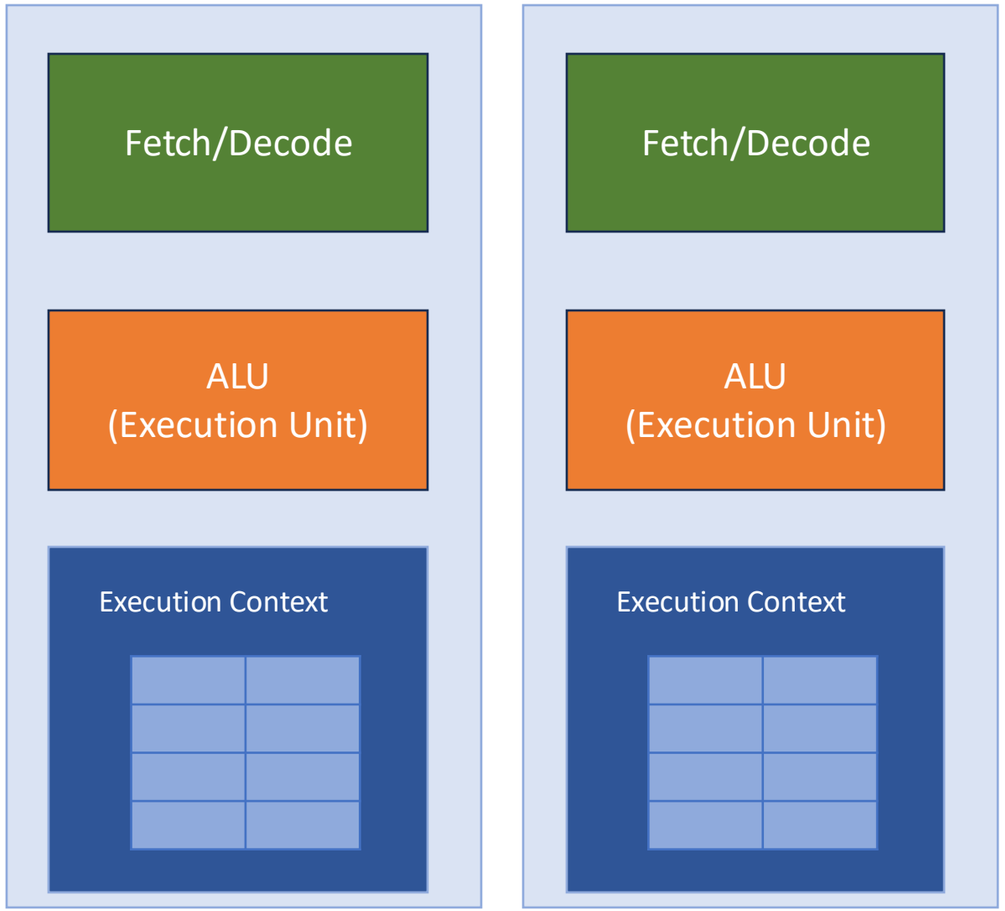
Rather than using transistors to increase clock speed to accelerate a single instruction stream

Use increasing transistor count to add more cores to the processor.

Simpler cores: each core may be slower at running a single instruction stream than original "fancy" core (e.g. 25% slower)

Programs by default will compile to an instruction stream that runs as one thread on one processor core.

If each of the simpler cores was 25% slower, our program runs 25% slower than before.



A dual-core processor: compute two elements in parallel (two instruction streams)

Multi-core processor

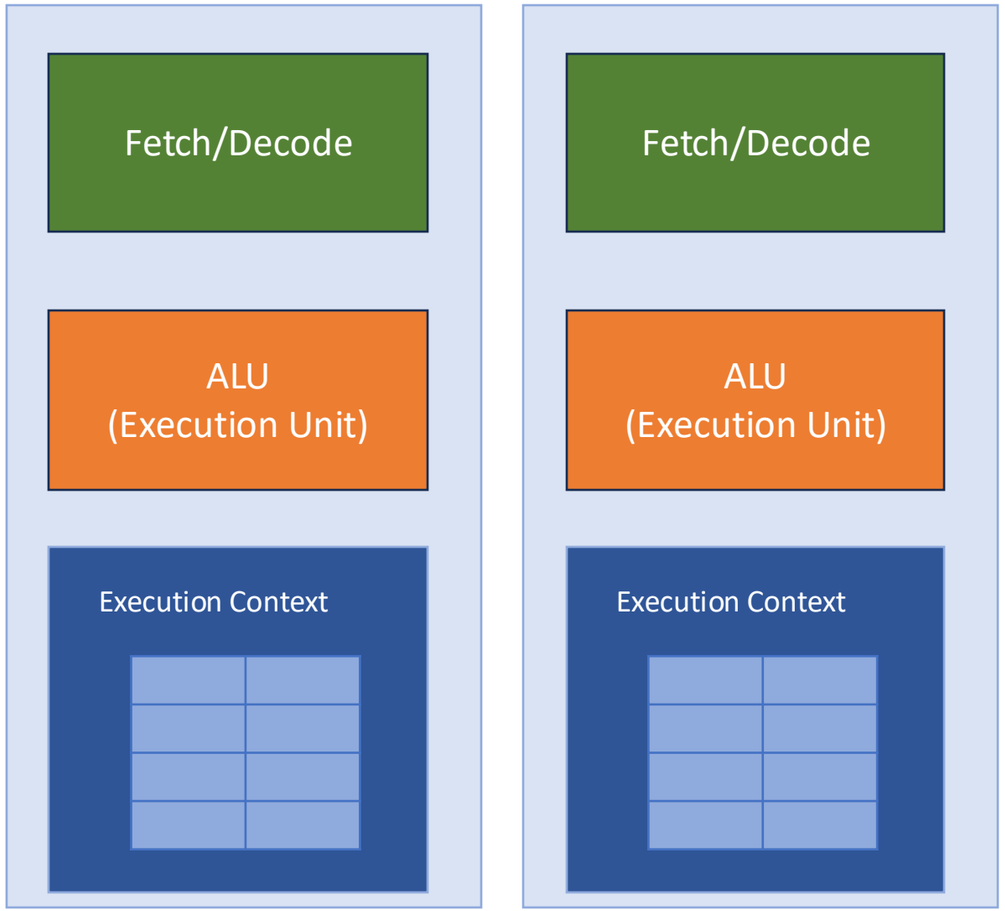
It's the responsibility of programmers to design and develop code that uses processor cores efficiently.

Data Parallelism

The data is divided across processors, and each processor performs the same task.

Task Parallelism

Different tasks run in parallel, often on the same or different data.



A dual-core processor: compute two elements in parallel (two instruction streams)

Single Instruction Multiple Data (SIMD) Parallelsim

Same instruction broadcast to all ALUs

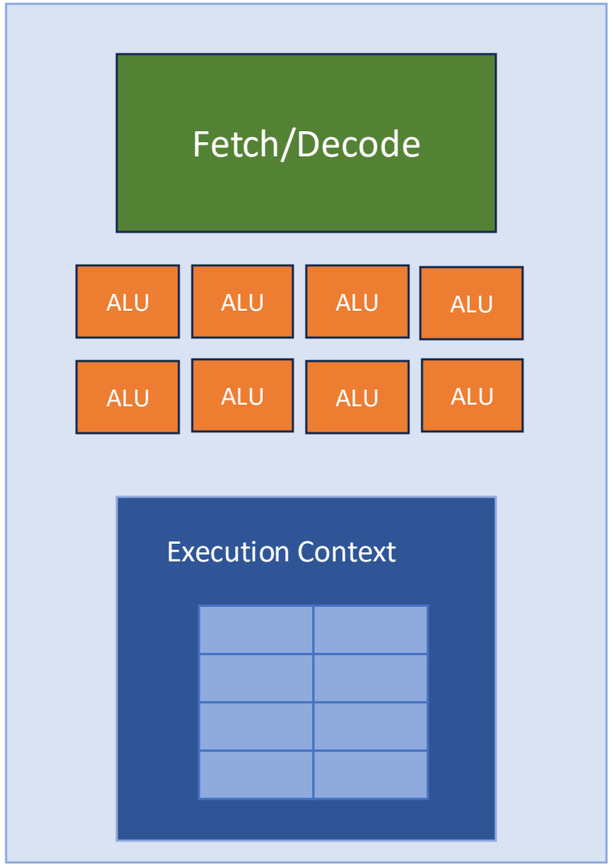
This operation is executed in parallel on all ALUs

Usually done by compilers

Modern processors are multicore and SIMD

A 16 core with 8 ALU each can compute 128 elements in parallel

16 instruction streams



Example

Compute sin(x) using Taylor expansion:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

```
void sinx(int N, int terms, float* x, float* y) {
    for (int i = 0; i < N; i++) {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j = 1; j <= terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2 * j + 2) * (2 * j + 3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```

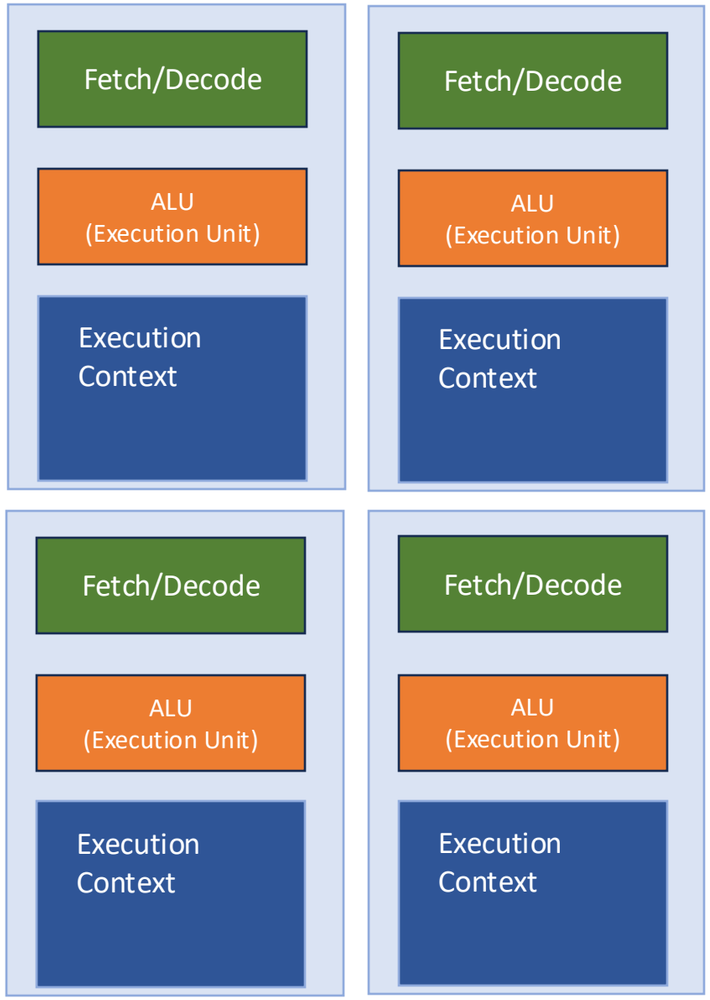
Example

```
typedef struct {
    int N;
    int terms;
    float* x;
    float* y;
} my_args;

void my_thread_func(my_args* args) {
    sinx(args->N, args->terms, args->x, args->y); // do work in thread
}

void parallel_sinx(int N, int terms, float* x, float* y) {
    std::thread my_thread;
    my_args args;

    // Setup arguments for thread
    args.N = N / 2;
    args.terms = terms;
    args.x = x;
    args.y = y;

    // Launch thread
    my_thread = std::thread(my_thread_func, &args);

    // Do second half of the work in the main thread
    sinx(N - args.N, terms, x + args.N, y + args.N);

    // Wait for the thread to complete
    my_thread.join();
}
```

```
void sinx(int N, int terms, float* x, float* y) {
    for (int i = 0; i < N; i++) {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j = 1; j <= terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2 * j + 2) * (2 * j + 3);
            sign *= -1;
        }

        y[i] = value;
    }
}
```

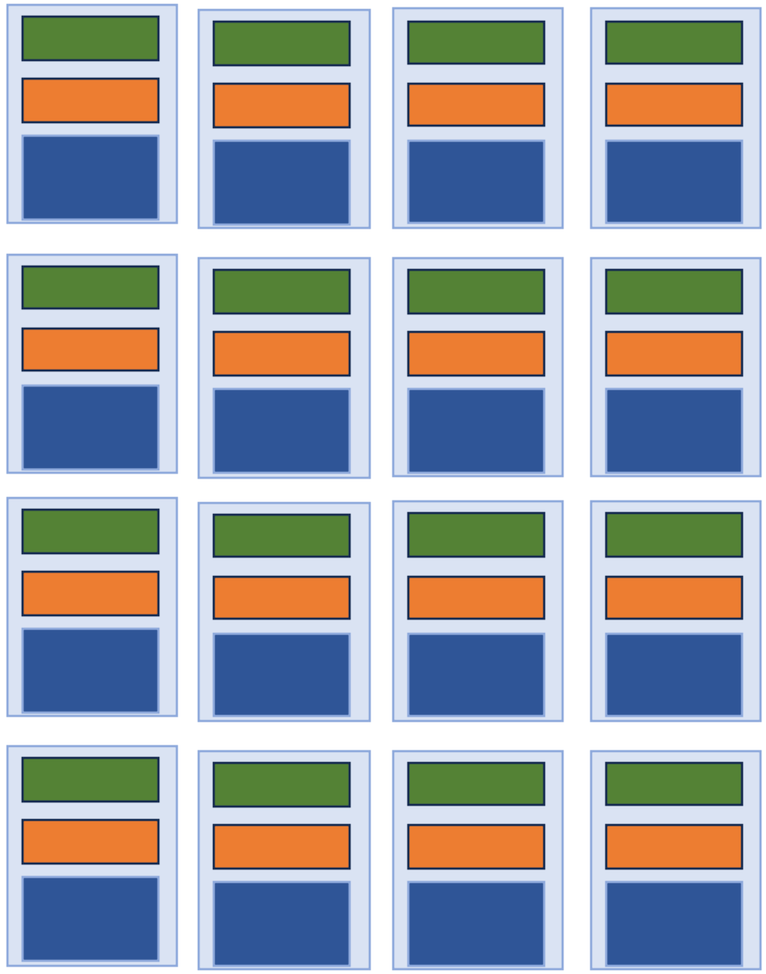
Data parallel or Task parallel?

Four-cores



A quad-core processor: how many instruction streams in parallel?

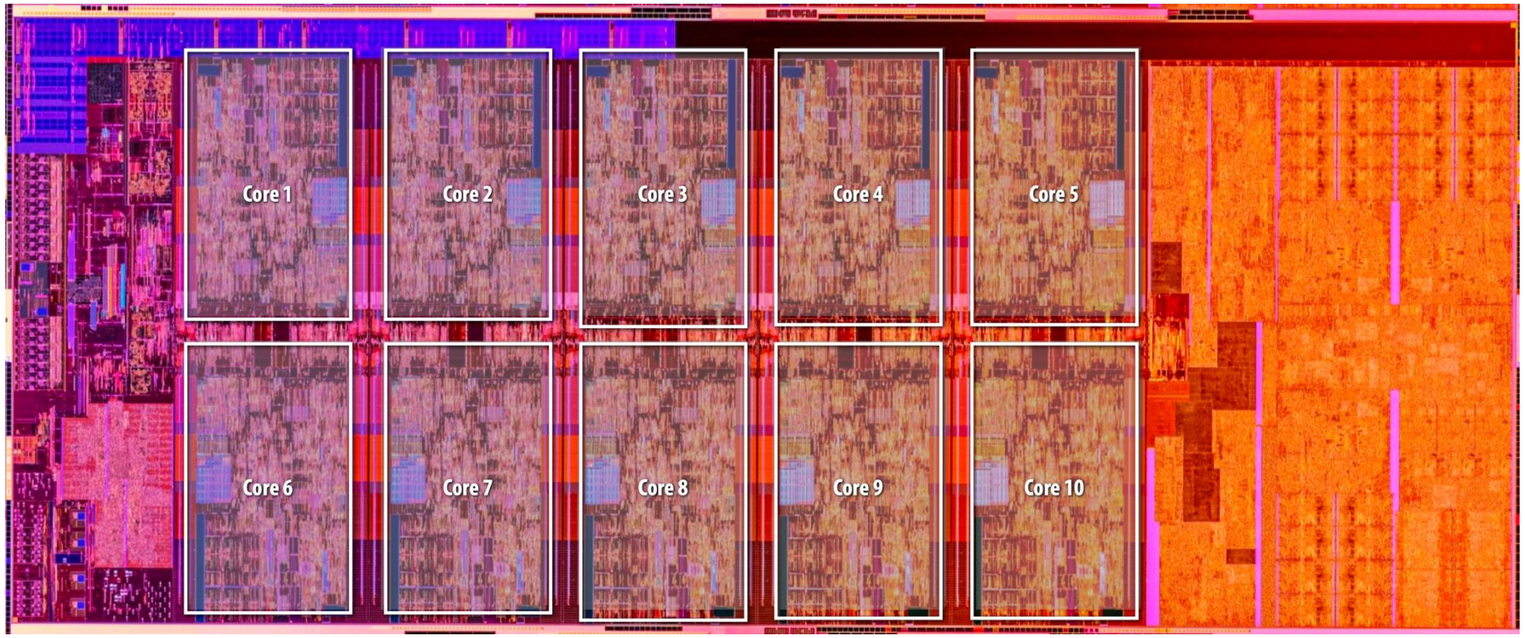
Sixteen cores



A 16-core processor: how many instruction streams in parallel?

Intel Comet Lake

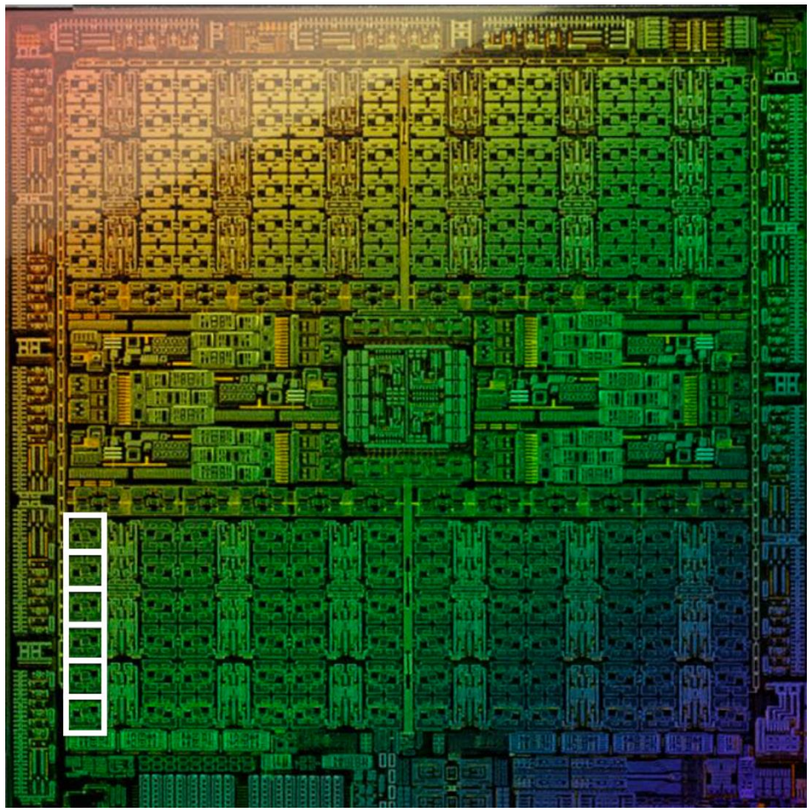
Inter comet lake 10th generation Core i9 10-core CPU (2020)



Multi-core GPU

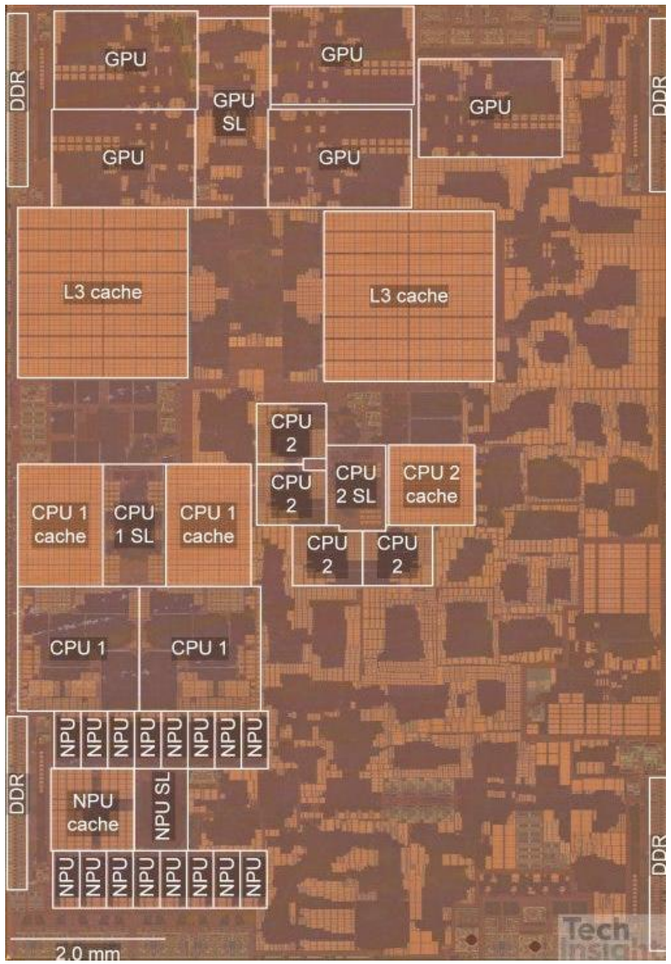
NVIDIA RTX 4090 (2022)

144 processing blocks (called SMs)



Apple A15 Bionic

Bionic



Three different forms of parallel execution

Superscalar: exploit ILP within an instruction stream. Process different instructions from the same instruction stream in parallel (within) a core.

Parallelism automatically discovered by the hardware during execution

SIMD: Multiple ALUs controlled by the same instruction (within a core)

Efficient for data-parallel workloads
vectorization done by compiler or at runtime by hardware

Multicore: use multiple processing cores

Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core

Software creates threads to expose parallelism to hardware (e.g. via threading API)

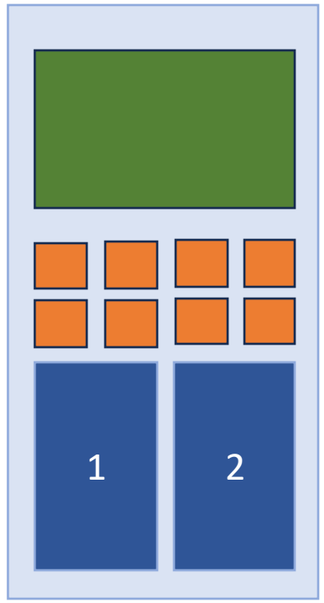
A multi-threaded core

A processor with multiple hardware threads has the ability to avoid stalls by performing instructions from other threads when one thread must wait for a long latency operation to complete.

A multi-threaded processor hides memory latency by performing arithmetic from other threads.

Core manages execution contexts for multiple threads

Threads are interleaved

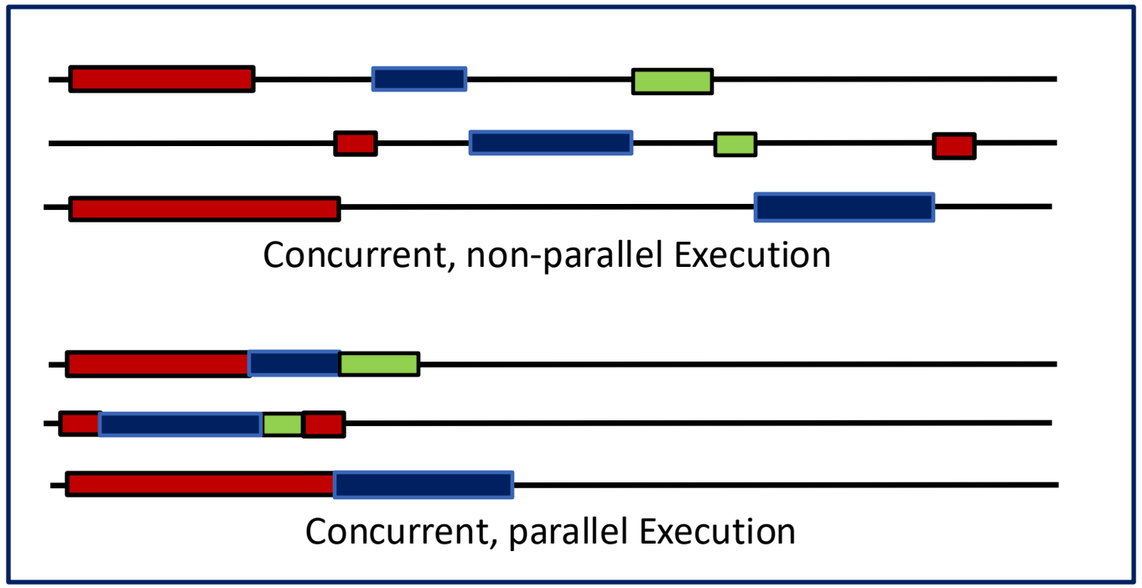


Single core processor, multithreaded core (2 threads).

Can run SIMD instruction per clock from one of the hardware threads

Concurrency vs Parallelism

- Concurrency: multiple tasks are ***logically*** active at one time.
- Parallelism: multiple tasks are ***actually*** active at one time.



A modern multi-core chip

16 cores

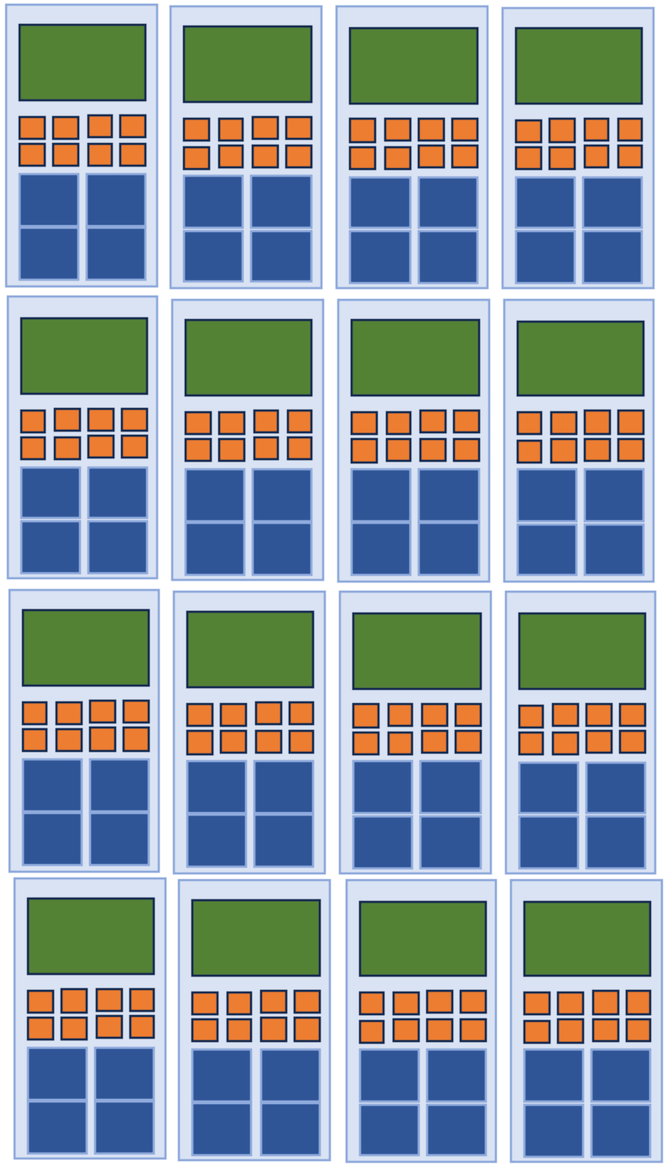
8 SIMD ALUs per core (128 total)

4 threads per core

16 simultaneous instruction stream

64 total concurrent instruction stream

512 independent pieces of work are needed to run chip with maximal latency hiding ability



Shared Memory vs Distributed Memory Architecture

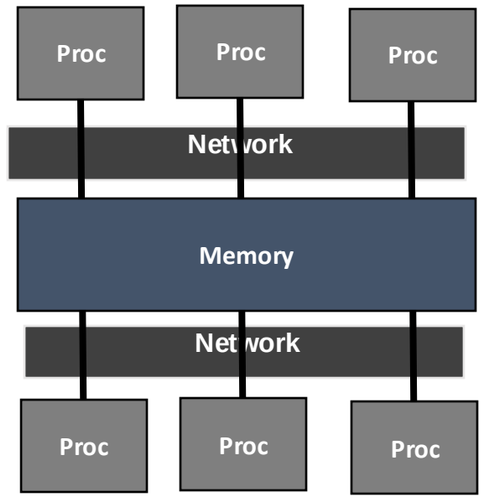
So far, we studied shared memory architecture

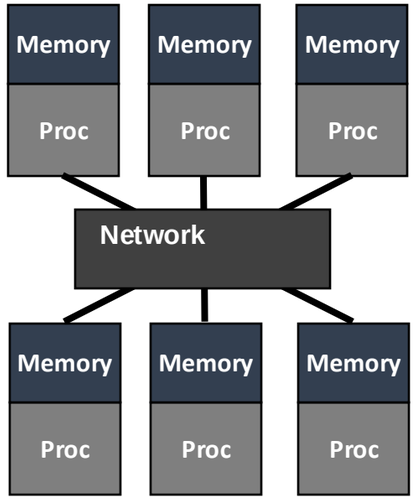
System's main memory is shared between cores.

A **distributed memory multiprocessor** has processors with their own memories connected by a high speed **network**

Also called a **cluster**

A **high performance computing (HPC)** system contains 100s or 1000s of such processors (nodes)





Shared Memory (SMP) or Multicore	High Performance Computing (HPC) or Distributed Memory

Questions?