Programmazione di Sistemi ~~Embedded~~ e Multicore

Teacher: Daniele De Sensi

Teaser Trailer

AI Boom

- Neural networks date back to the 60s
- What are the resons of the current AI boom?

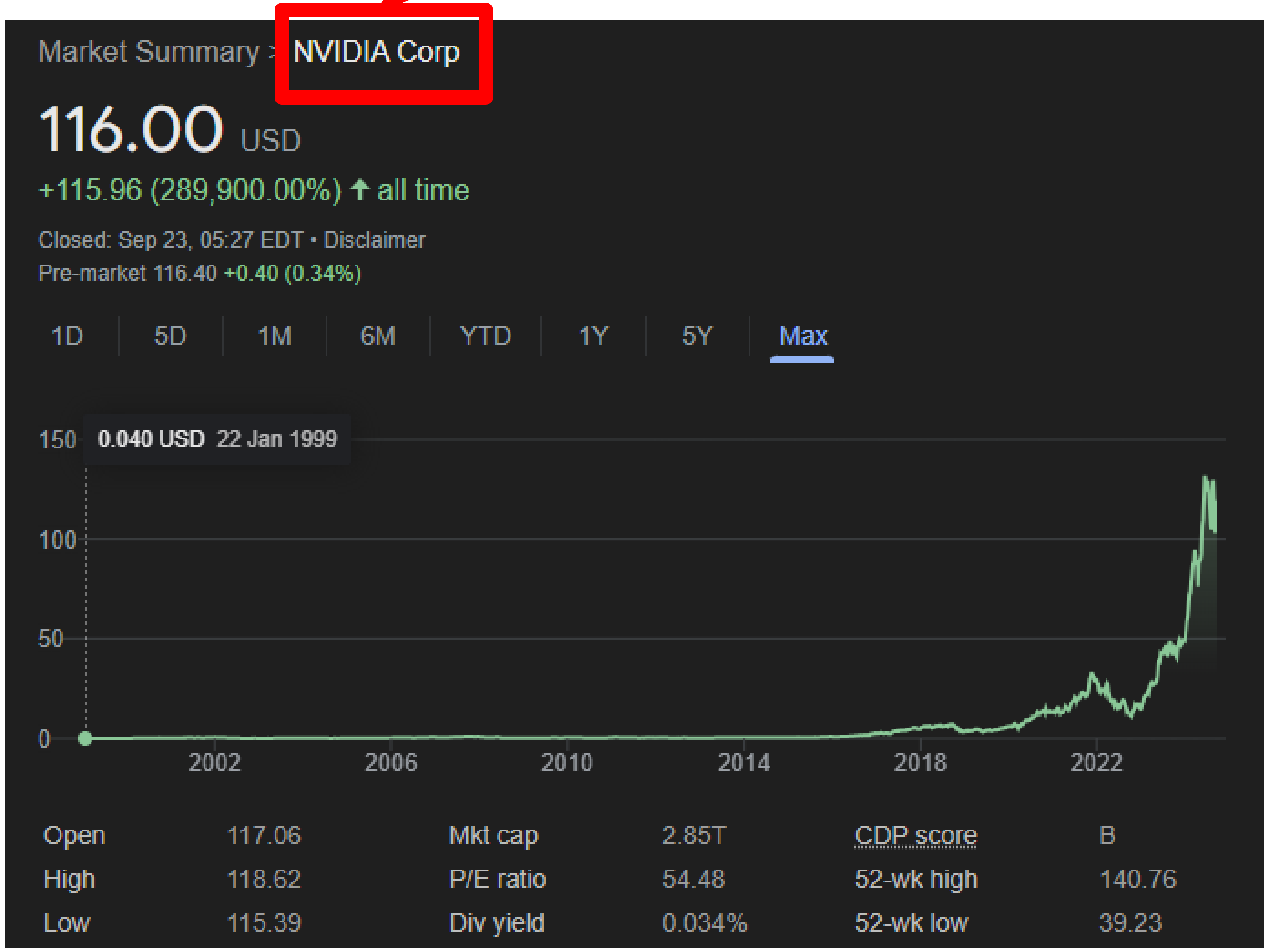
Data Availability

More Complex Models

Computational Power Availability

AI Boom

It is an hardware company (they build GPUs),
not an AI company such as OpenAI, etc..


Market Summary > NVIDIA Corp

116.00 USD

+115.96 (289,900.00%) ↑ all time

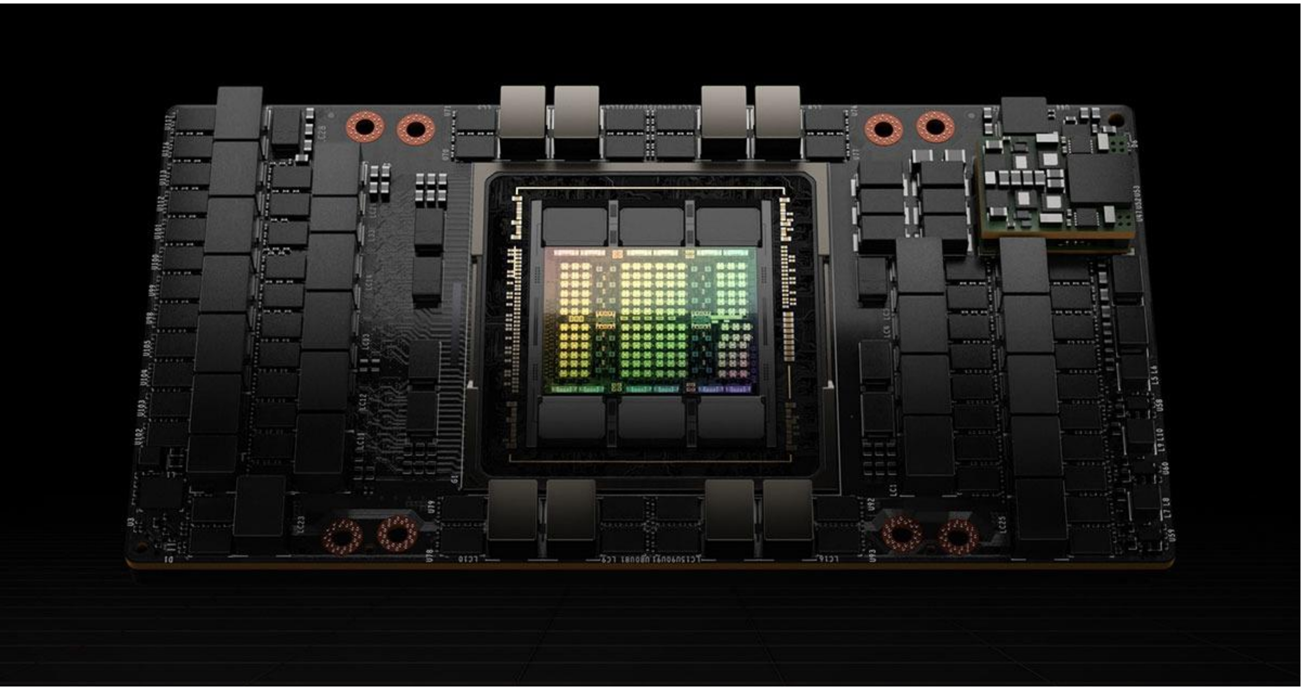
Closed: Sep 23, 05:27 EDT • Disclaimer

Pre-market 116.40 +0.40 (0.34%)

Open	117.06	Mkt cap	2.85T	CDP score	B
High	118.62	P/E ratio	54.48	52-wk high	140.76
Low	115.39	Div yield	0.034%	52-wk low	39.23

Computational Power Availability

GPU



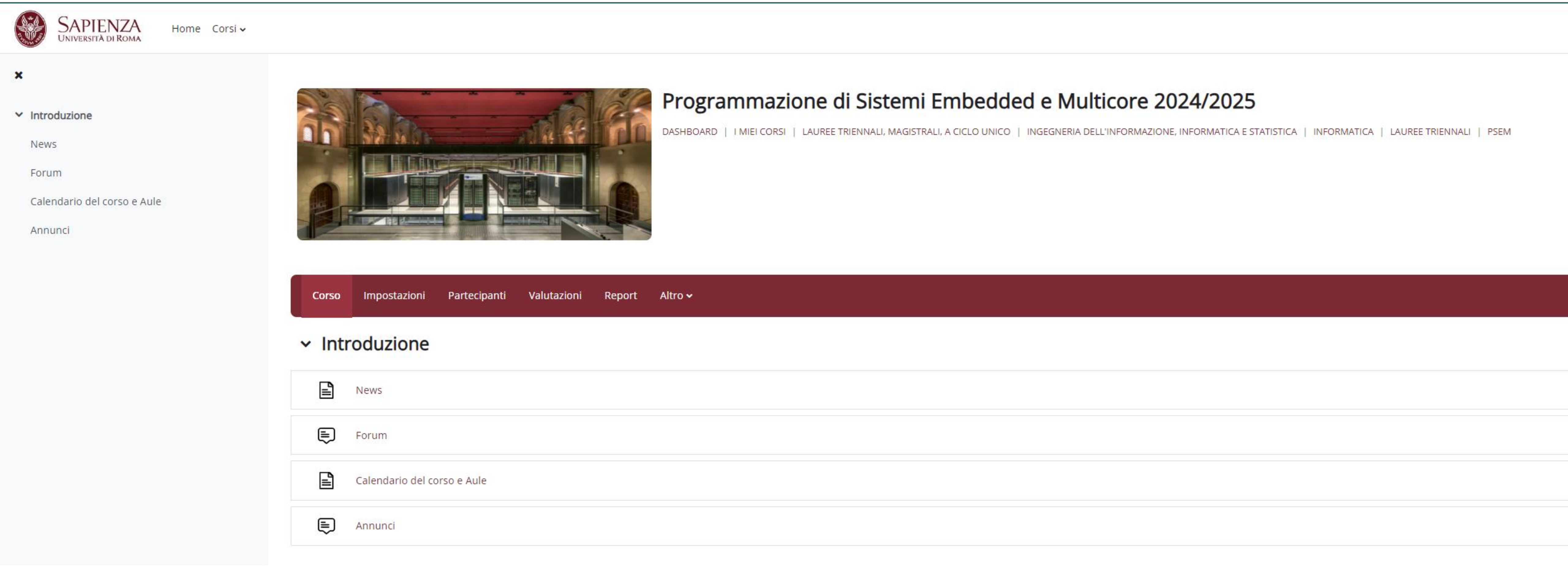
- 80 billion transistors
- 30,000 USD
- How many of those do you need to train something like ChatGPT?
  - > 200,000
  - Cost: 6 billion USD
- How much will the electicity bill cost?
  - 1 million USD per month for around 10,000 GPUs
  - Last week news: Oracle is going to power its datacenter with 3 small nuclear reactors, Microsoft is going to re-open a nuclear reactor

- How do you program these systems efficiently?

Questions?

General Info – Part I

- **Moodle** page with slides, forum, news, exercises, etc…
- *https://elearning.uniroma1.it/course/view.php?id=**18515***



SAPIENZA UNIVERSITÀ DI ROMA

Home Corsi

Introduzione

News

Forum

Calendario del corso e Aule

Annunci

Programmazione di Sistemi Embedded e Multicore 2024/2025

DASHBOARD | I MIEI CORSI | LAUREE TRIENNALI, MAGISTRALI, A CICLO UNICO | INGEGNERIA DELL'INFORMAZIONE, INFORMATICA E STATISTICA | INFORMATICA | LAUREE TRIENNALI | PSEM

Corso Impostazioni Partecipanti Valutazioni Report Altro

Introduzione

News

Forum

Calendario del corso e Aule

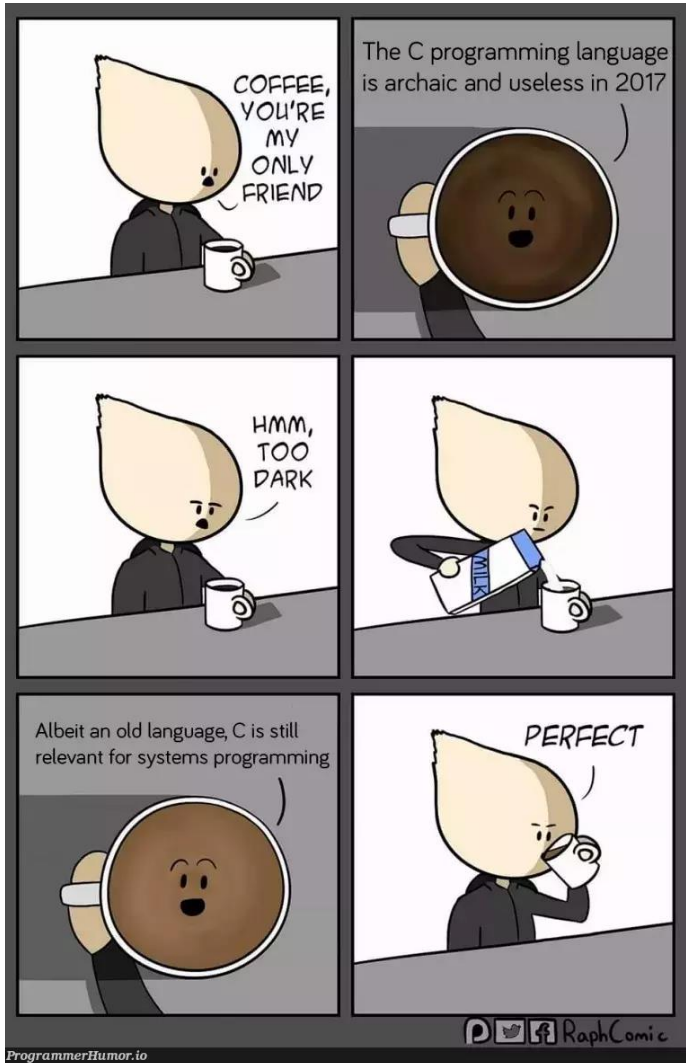
Annunci

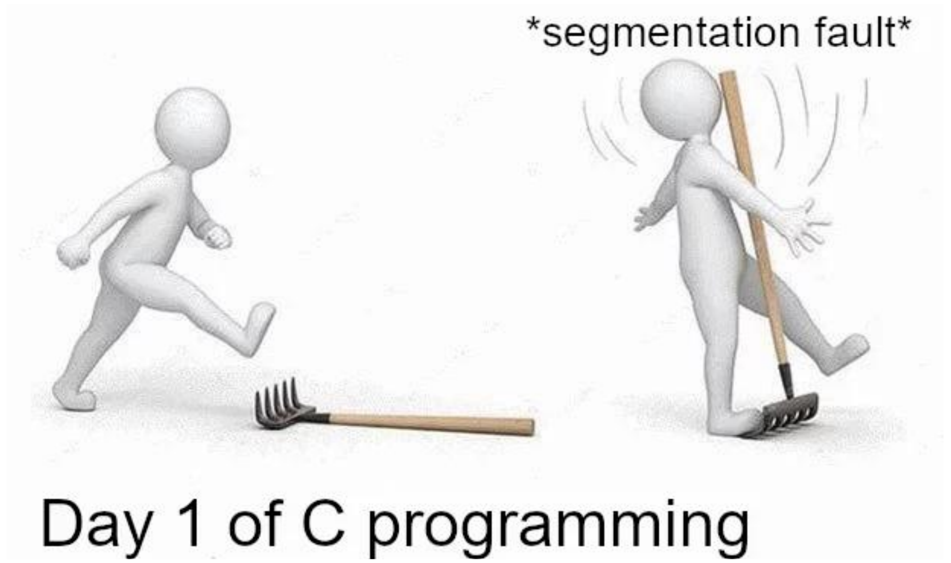
General Info – Part II

- **Email:** desensi@di.uniroma1.it
- **Schedule:**
  - Tuesday from 13:00 till 15:00 (Aula 1 Castro Laurenziano)
  - Thursday from 13:00 till 16:00 (Aula 1 Castro Laurenziano)
- **Question Time:** You can book question time slots by emailing me (we will agree on a day/time/place).
- Book: ***An Introduction to Parallel Programming*, 2nd ed., Pacheco & Malensek**
  - **Extra resources will be linked on Moodle**
- Interactions/questions during lectures are welcome
- **Exam:** Project + oral exam
  - **Possibility of extra points for intermediate evaluations (TBD)**

General Info – Part III

Reference programming language: C (we said we want to do things efficiently!)


COFFEE, YOU'RE MY ONLY FRIEND
The C programming language is archaic and useless in 2017
HMM, TOO DARK
MILK
Albeit an old language, C is still relevant for systems programming
PERFECT
RaphComic
ProgrammerHumor.io



Program

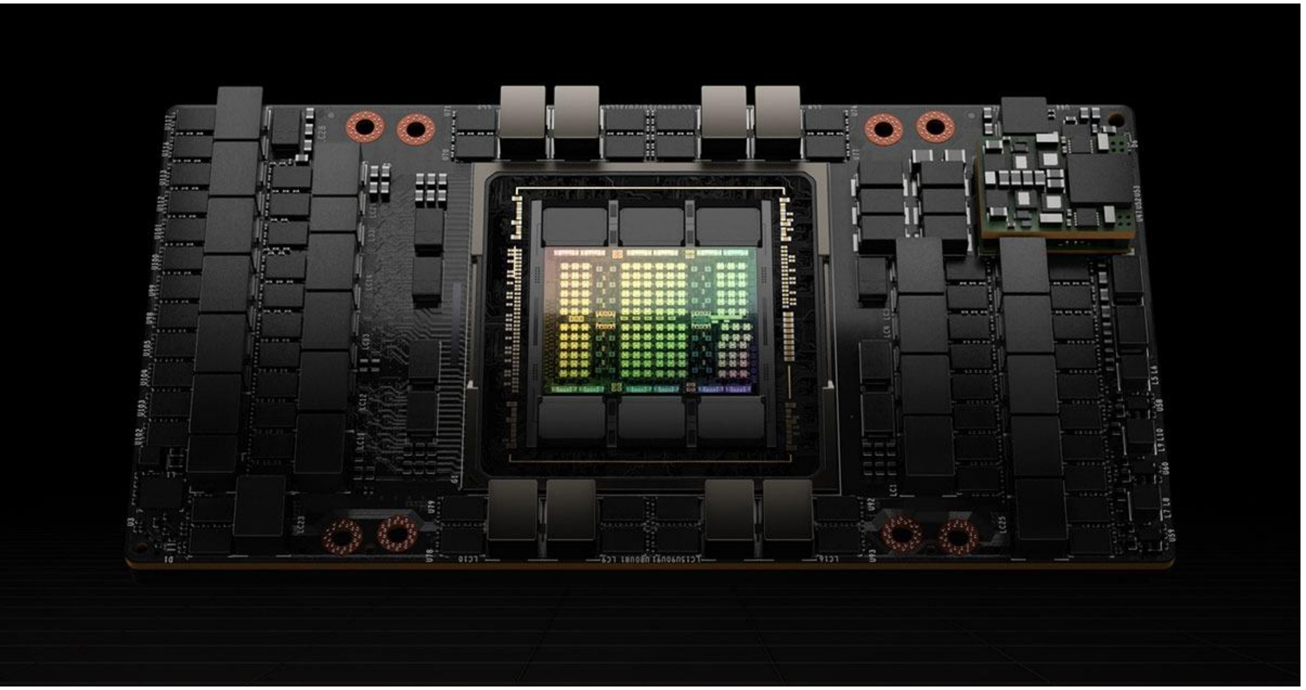
- Chapter 1: Why parallel computing
- Chapter 2: Parallel hardware and parallel software
- Chapter 3: Distributed programming with MPI
- Chapter 4: Shared-memory programming with Pthreads
- Chapter 5: Shared-memory programming with OpenMP
- $\boxed{\text{Chapter 6: GPU programming with CUDA}}$

Might be on another book

Questions?

Why parallel computing

GPU



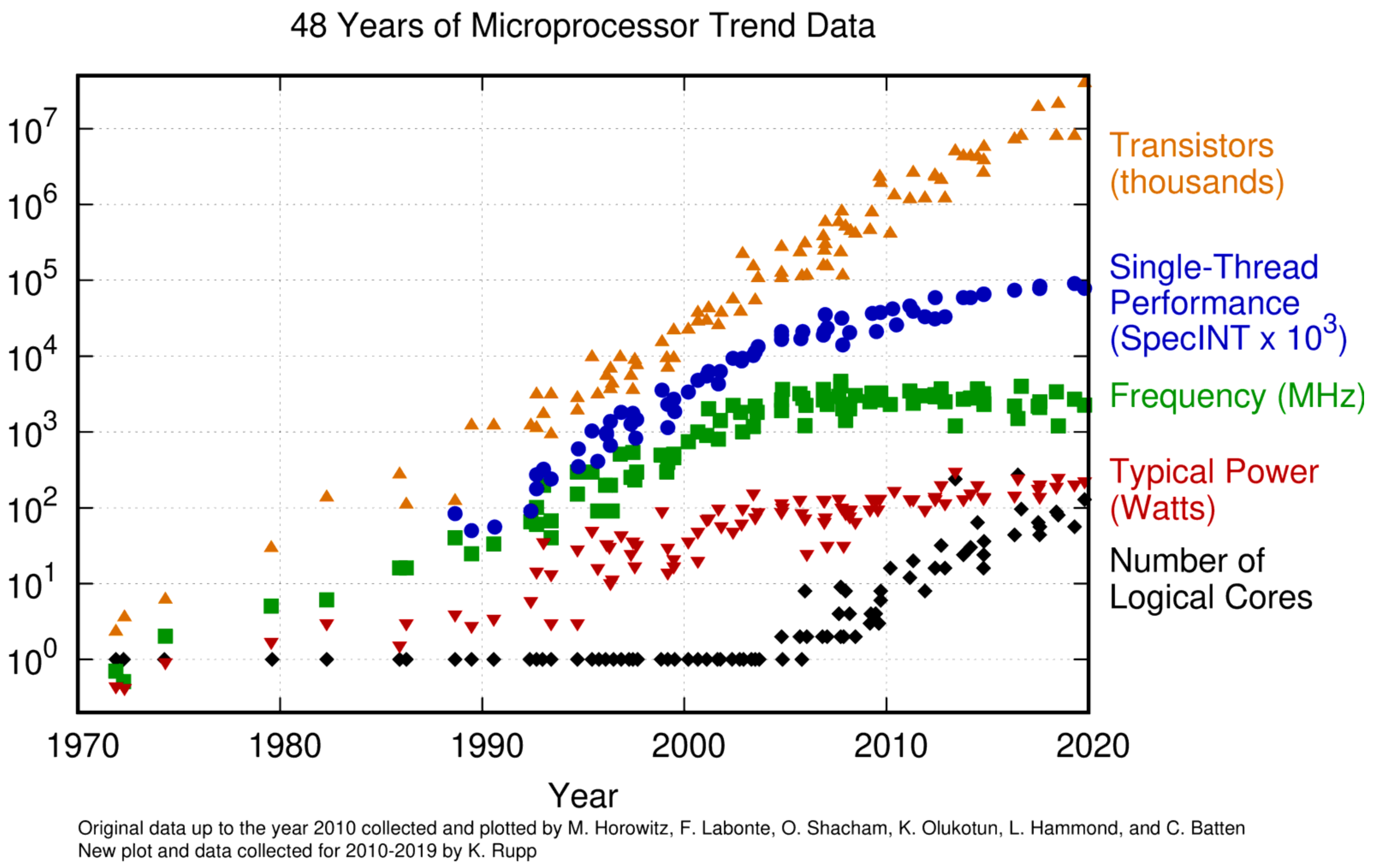
- 80 billion transistors
- 30,000 USD
- How many of those do you need to train something like ChatGPT?
  - > 200,000
  - Cost: 6 billion USD
- How much will the electicity bill cost?
  - 1 million USD per month for around 10,000 GPUs
  - Last week news: Oracle is going to power its datacenter with 3 small nuclear reactors, Microsoft is going to re-open a nuclear reactor

- How do you program these systems <u>efficiently</u>?
- How did we reach this point? Wouldn't it be simpler to have a huge CPU?

Why parallel computing

- From 1986 to 2003, the performance of microprocessors increased, on average, by **50% per year (60x increase in 10 years)**
- Since 2003 this increase slowed down. From 2015 to 2017, by **4% a year (1.5x increase in 10 years)**

- For this reason, instead of trying to have more powerful monolithic processors, we are putting **multiple processors on a single integrated circuit**


48 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Programmers

- Serial programs don't benefit from this approach. In most cases, they **will still run on a single processor** (but you can have tens/hundreds of them in a single CPU)
- The programmer must be aware of them and **know how to use them**
- But wait, why do we need all that computational power? **Can't we just keep running sequential code?**

Why this need for performance

- Humanity always finds more complex problems to address:
  - LLMs
  - Decoding the human genome
  - Climate modeling
  - Protein folding
  - Drug discovery
  - Energy research
  - Fundamental physics (e.g., experiments at CERN etc...)
- I might invite some of those scientists to give seminars (additionally to lectures)

Why we build parallel systems

- Traditionally, performance increases caused by increase in the **density of transistors**
- However:
  - Smaller transistors -> Faster processors
  - Faster processors -> Increased power consumption
  - Increased power consumption -> Increased heat
  - Increased heat -> Unreliable processors
- I.e., there are **physical limits** that make parallel computers necessary

Why do we need to write parallel programs

- Running multiple instances of a serial program is often not useful
  - Think about running multiple instances of your favorite game (**it will not run faster!**)
- Programs must be written to explicitly exploit the parallelism
- Can't this be done automatically?
  - Difficult to do so
  - Limited success

Why we build parallel systems

- Some coding constructs can be **recognized and converted** to a parallel construct
- However, this is **often inefficient**
- Sometimes, the best solution is to **step back and design an entirely new algorithm**
- I.e., often, it is **not enough to just parallelize the available sequential code**

Example

- Compute n values and add them together
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

Example – Parallel solution

- We have *p* cores, *p* << *n*
- Each core computes the sum of n/p values

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
    my_x = Compute_next_value(. . .);
    my_sum += my_x;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores.

- After each core completes execution, *my_sum* contains the sum of n/p values

Example – Parallel solution

- Ex. p=8 cores, n=24

1, 4, 3,   9, 2, 8,   5, 1, 1,   6, 2, 7,   2, 5, 0,   4, 1, 8,   6, 5, 1,   2, 3, 9,

then the values stored in `my_sum` might be

Core	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
`my_sum`	8	19	7	15	7	13	12	14

Example – Parallel solution

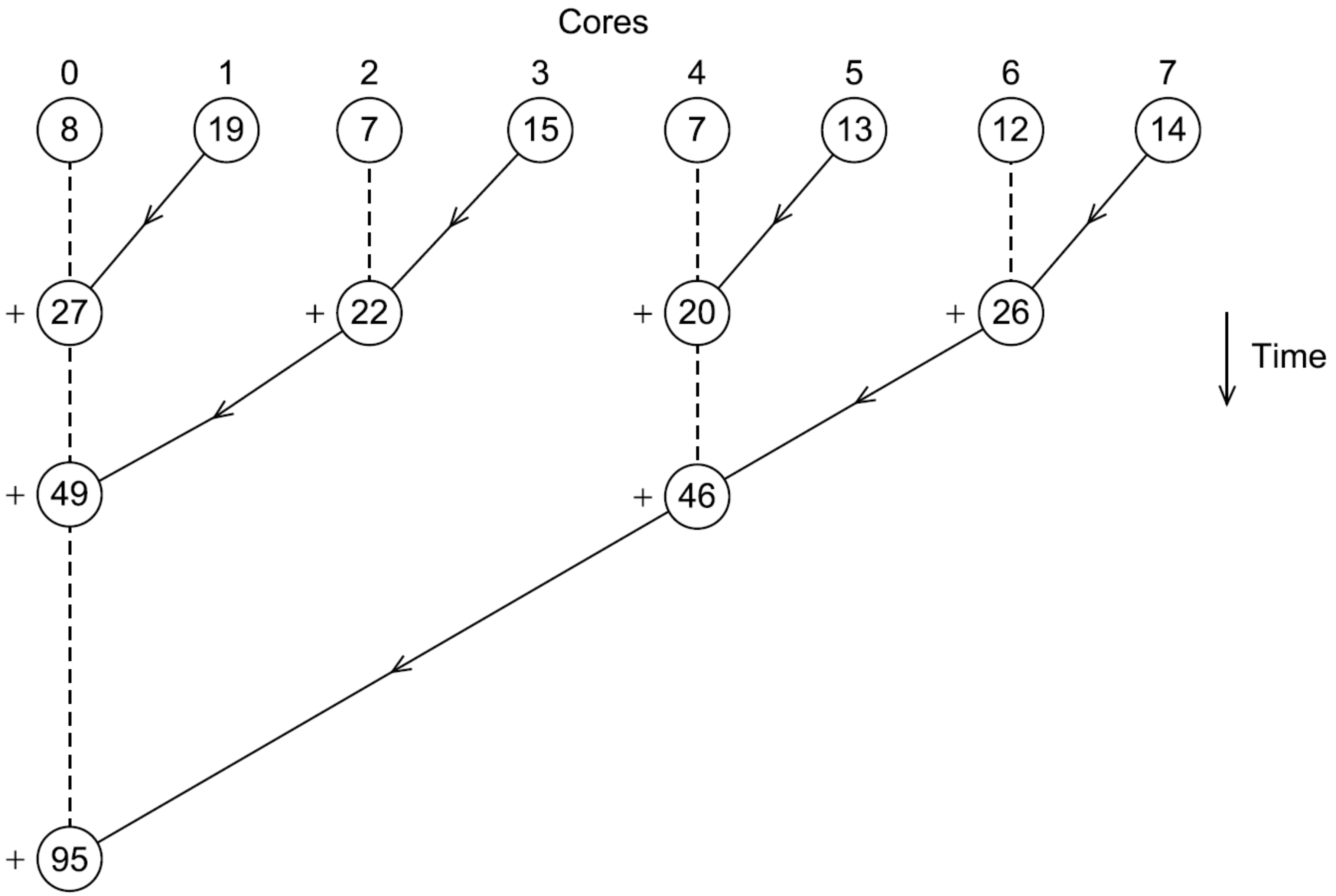
- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.

```
if (I'm the master core) {
    sum = my_sum;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_sum to the master;
}
```

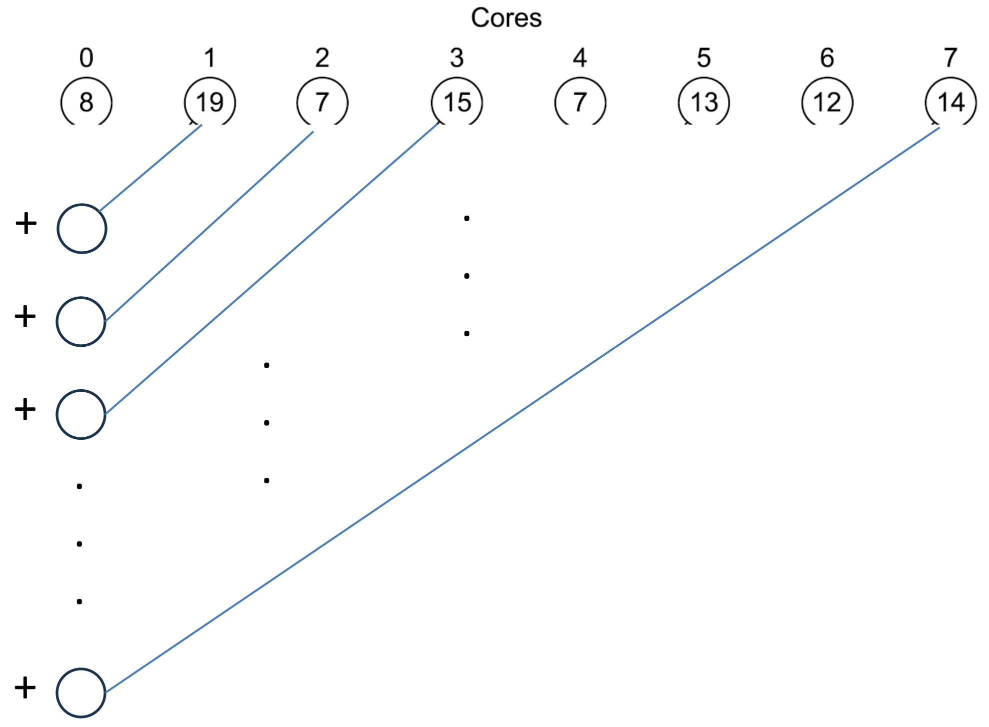
Example – Better parallel solution

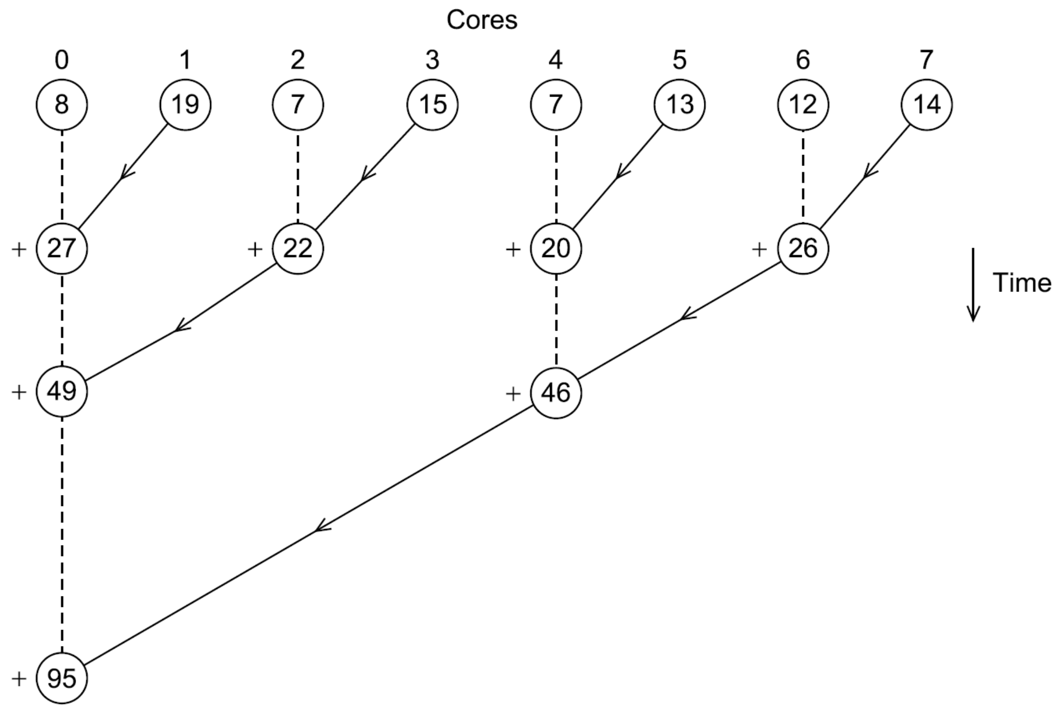
- Issue with the previous solution: the **master is doing all the work**, while the other p-1 cores are doing nothing
- Pair the cores so that core 0 adds its result with core 1's result, core 2 adds its result with core 3's result, etc…
- Repeat the process only with 0, 2, 4, ….
- Repeat the process only with 0, 4, …
- Done

Example – Better parallel solution



Example – Solutions comparison





7 receive + 7 additions

In general, p-1 receive/additions

3 receive + 3 additions

In general, log2(p) receive/additions

If p=1000, 999 receive/additions vs. 10 receive/additions

Example – Solutions comparison

- We could have an optimized implementation of this global sum, and a compiler could replace the serial code with the parallel code
- Things get complex easily

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

- What if *compute_next_value* has an internal state which gets updated at each iteration?

Questions?

How do we write parallel programs?

- Task parallelism
  - Partition various tasks among the cores
  - The "temporal parallelism" we have seen in circuits is a specific type of task parallelism
- Data parallelism
  - Partition the data used in solving the problem among the cores
  - Each core carries out similar operations on it's part of the data
  - Similar to "spatial parallelism" in circuits

How do we write parallel programs? Example

- I need to grade 300 exams, each with 15 questions, and I have 3 teaching assistants
- Data parallelism:
  - Each assistant grades 100 exams
- Task parallelism
  - Assistant 1 grades all the exams, but only questions 1-5
  - Assistant 2 grades all the exams, but only questions 6-10
  - Assistant 3 grades all the exams, but only questions 11-15

Sum example – $\boxed{\text{Data}}$ or task?

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

Sum example – Data or $\boxed{\text{task?}}$

```
if (I'm the master core) {
    sum = my_sum;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_sum to the master;
}
```

Task 1: Receive and sum

Task 2: Send partial sum to master

How do we write parallel programs?

- If each core can work independently, it is quite similar to write a serial program
  - E.g., you want to compress 100 files and you have 10 cores: each core compresses 10 files
- In practice, cores need to coordinate, for different reasons:
  - **Communication:** e.g., one core sends its partial sum to another core
  - **Load balancing:** share the work evenly so that one is not heavily loaded (what if some file to compress is much bigger than the others?)
    - If not, p-1 one cores could wait for the slowest (wasted resources & power!)
  - **Synchronization:** each core works at its own pace, make sure some core does not get too far ahead
    - E.g., one core fills the list of files to compress. If the other ones start too early they might miss some files

Questions?

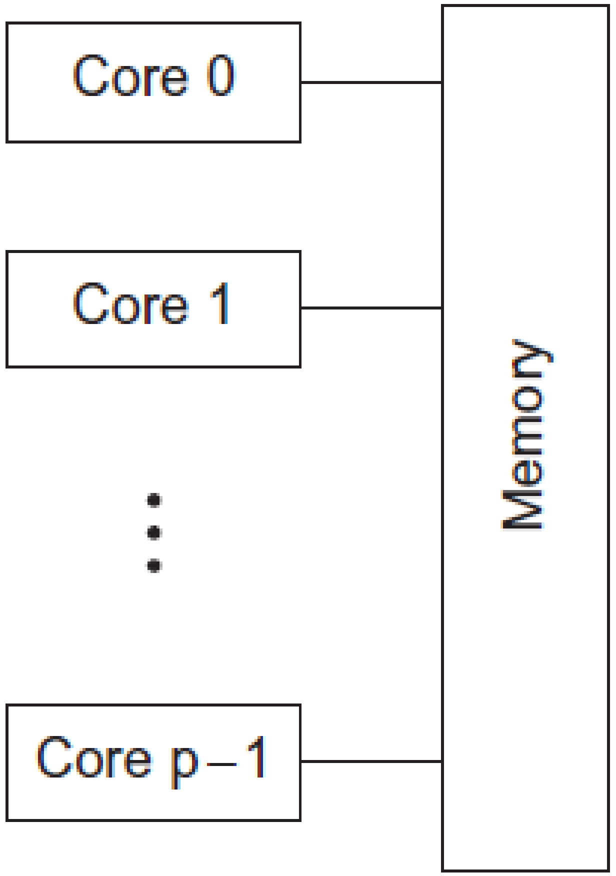
How will we write parallel programs?

- We will write programs that are explicitly parallel
- Using four different extensions of the C APIs:
  - Message-Passing Interface (MPI) [Library]
  - Posix Threads (Pthreads) [Library]
  - OpenMP [Library + Compiler]
  - CUDA [Library + Compiler]
- Higher-level libraries exist, but they trade-off ease-of-use for performance (i.e., the more performing/efficient you want to be, the more you need to suffer)
- Why do we need four different libraries?

Type of parallel systems

- Shared-memory
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- Distributed-memory
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

Type of parallel systems



(a)

Shared-memory

Type of parallel systems

- Multiple-Instruction Multiple-Data (MIMD)
  - Each core has its own control units and can work independently from the others
- Single-Instruction Multiple-Data (SIMD) (e.g., GPUs)
  - Cores share the control units (they must all execute the same instruction – or stay idle)

Time	First core	Second core
1	Addition	Idle
2	Idle	Multiply

How will we write parallel programs?

	Shared Memory	Distributed Memory
SIMD	CUDA	
MIMD	Pthreads/ OpenMP/ CUDA	MPI

Questions?

Concurrent vs. Parallel vs. Distributed

- There isn't a complete agreement on the definition, but:
- Concurrent: Multiple tasks can be in progress at any time
- Parallel: Multiple tasks cooperate closely to solve a problem
- Distributed: A program might need to cooperate with other programs to solve a problem

- Parallel and distributed are concurrent
- Concurrent programs can be serial (e.g., multitasking operating system running on a single core)
- Parallel: tightly coupled (cores share the memory or are connected through a fast network)
- Distributed: more loosely coupled (e.g., servicesconnected through the Internet)

Remarks & Conclusions

- The laws of physics have brought us to the doorstep of multicore technology. Existing applications can use >200,000 GPUs at once
- Serial programs typically don't benefit from multiple cores.
- Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers.
- Learning to write parallel programs involves learning how to coordinate the cores.
- Several APIs depending on the type of parallel system
- Parallel programs are more complex than their serial counterpart and therefore, require sound program techniques and development.