Machine Learning in Climate Science MSc Class in *Ocean and Climate Physics*
!Open for Computer Science!



UHH Vorlesung „Hochleistungsrechnen“
GPU Hardware and Programming

Christopher Kadow & Georgiana Mania

German Climate Computing Center

1. DL/ML on HPCs — Chris
2. GPU Hardware and Programming — Chris + Georgiana
3. LLMs on HPCs — Chris
   plus Tutorial — Chris + Sebastian
4. LLMs on HPCs II — Chris + Gizem

DKRZ
DEUTSCHES KLIMARECHENZENTRUM

Class @ UHH
BSc/MSc Studies
Nov, 19 2026

FROM PIXELS TO SUPERCOMPUTERS

The surprising history of GPUs—from games and 3D graphics to CUDA, AI and climate computing.

HARDWARE CREATES POSSIBILITIES. SOFTWARE UNLOCKS THEM.

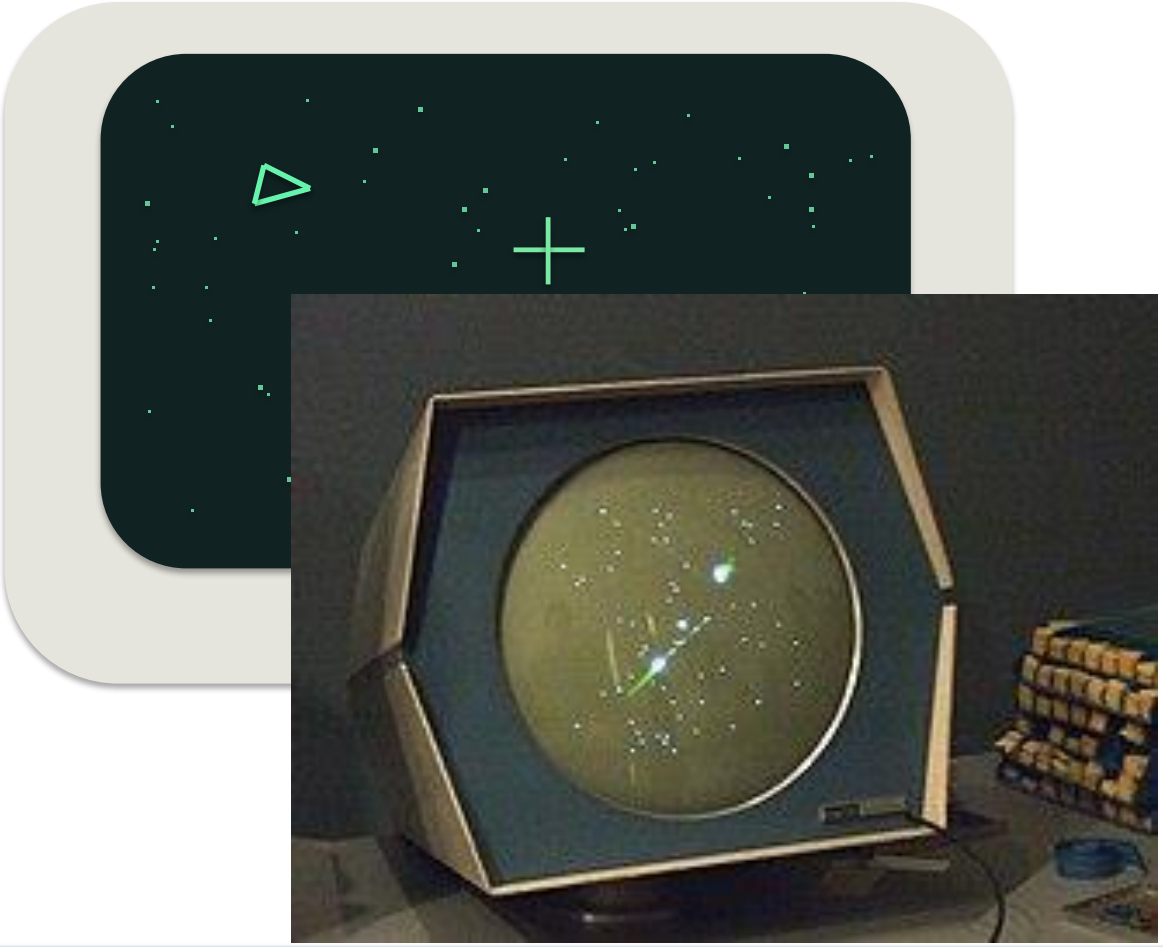


Epic Games image via Stanford CS149 (uploaded PDF)

1960S

Computers begin to draw.

Long before GPUs: interactive displays and vector graphics.



1962 · SPACEWAR!

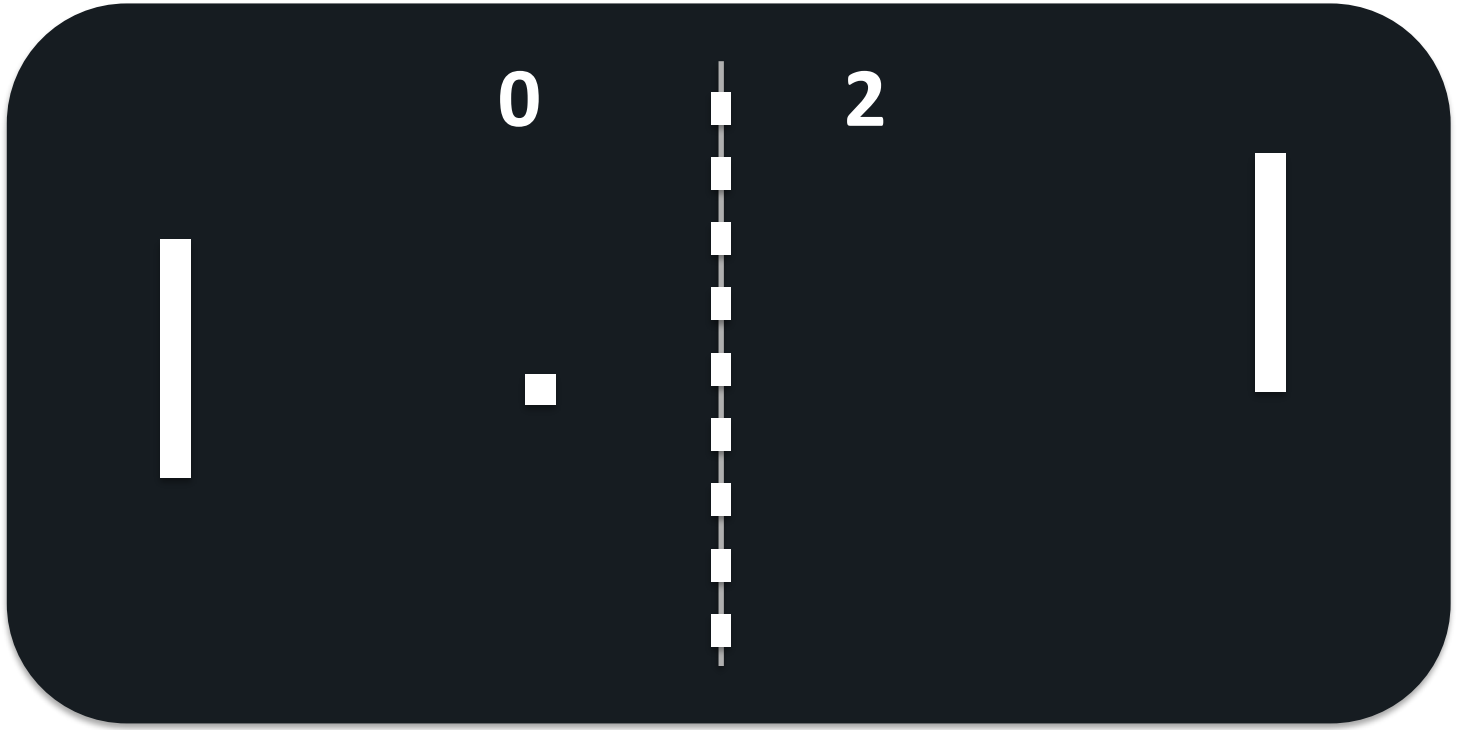
Vector displays: lines and points, not millions of shaded pixels.

Already interactive. Already computational. But still a long way from modern GPU graphics.

1972

Pong: two paddles, one moving dot.

Computer graphics become entertainment—and an enormous commercial demand driver.



1972

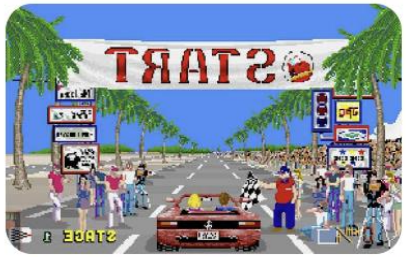
Simple graphics. Massive impact.

Every interaction must update the screen quickly. This becomes the central pressure on graphics hardware.

1980S

Games demand more: color, motion, detail.

The screen becomes a competitive battlefield for hardware innovation.



The 15 greatest video games of the 80...
The Guardian



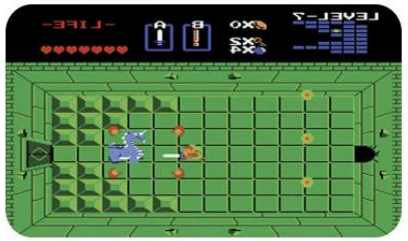
What computers did you first play gam...
Reddit



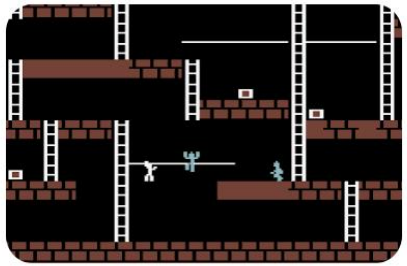
The 15 greatest video games of the 80...
The Guardian



The 15 greatest video games of the 80...
The Guardian



The 50 best games of the '80s, ranked ...
GamesRadar



GameSpy: The Greatest PC Games of A...
GameSpy



GameSpy: The Greatest PC Games of A...
GameSpy



The Classic PC Gaming Era (1977-1989) ...



20 Computer Games That Should've...
Rediscover the 80s



20 Forgotten Video Games From The 80s
TheGamer



The 50 best games of the '80s, ranked ...
GamesRadar

1980s

More objects.
More pixels.
More frames.

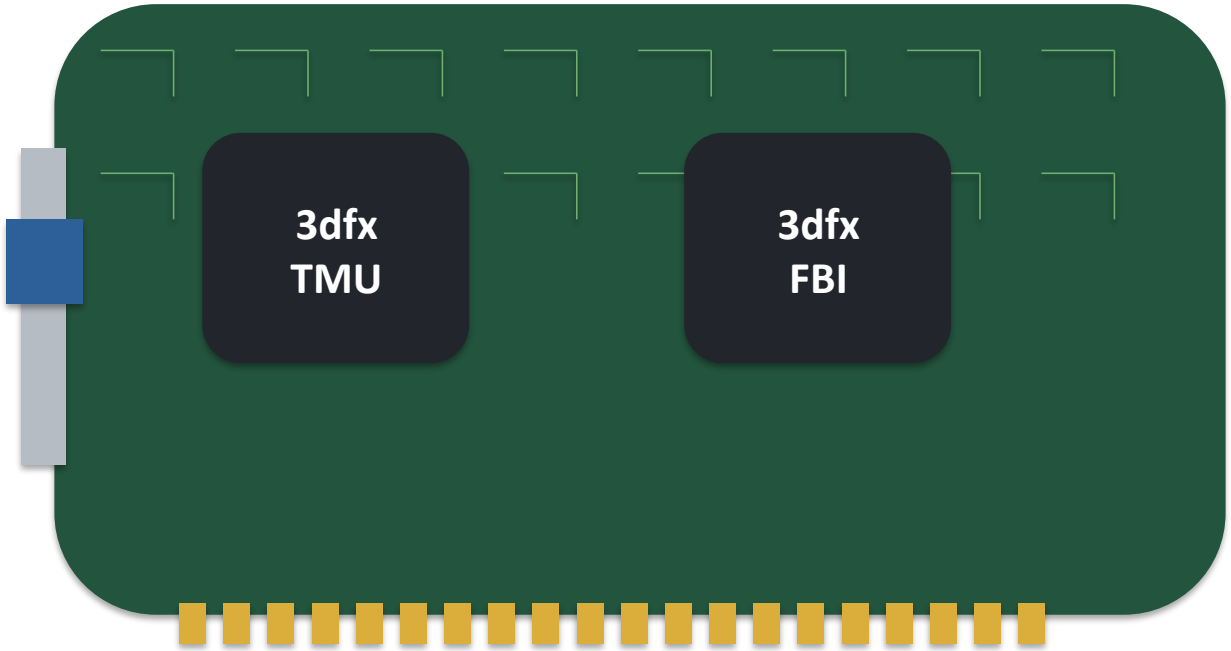
3D?

PCs, consoles and arcades make rendering speed a product differentiator.

1996

3dfx Voodoo: add a card just for 3D.

A historic idea: let specialized hardware do the graphics work.



1996 · VOODOO GRAPHICS

Not a second CPU.
A graphics coprocessor.

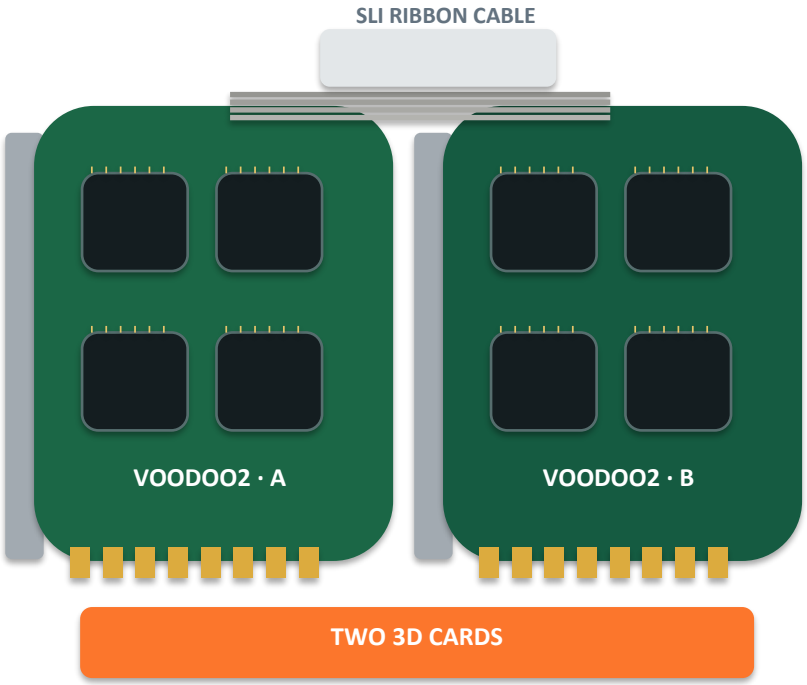
The original Voodoo was a 3D-only accelerator: it worked alongside a separate 2D graphics card.

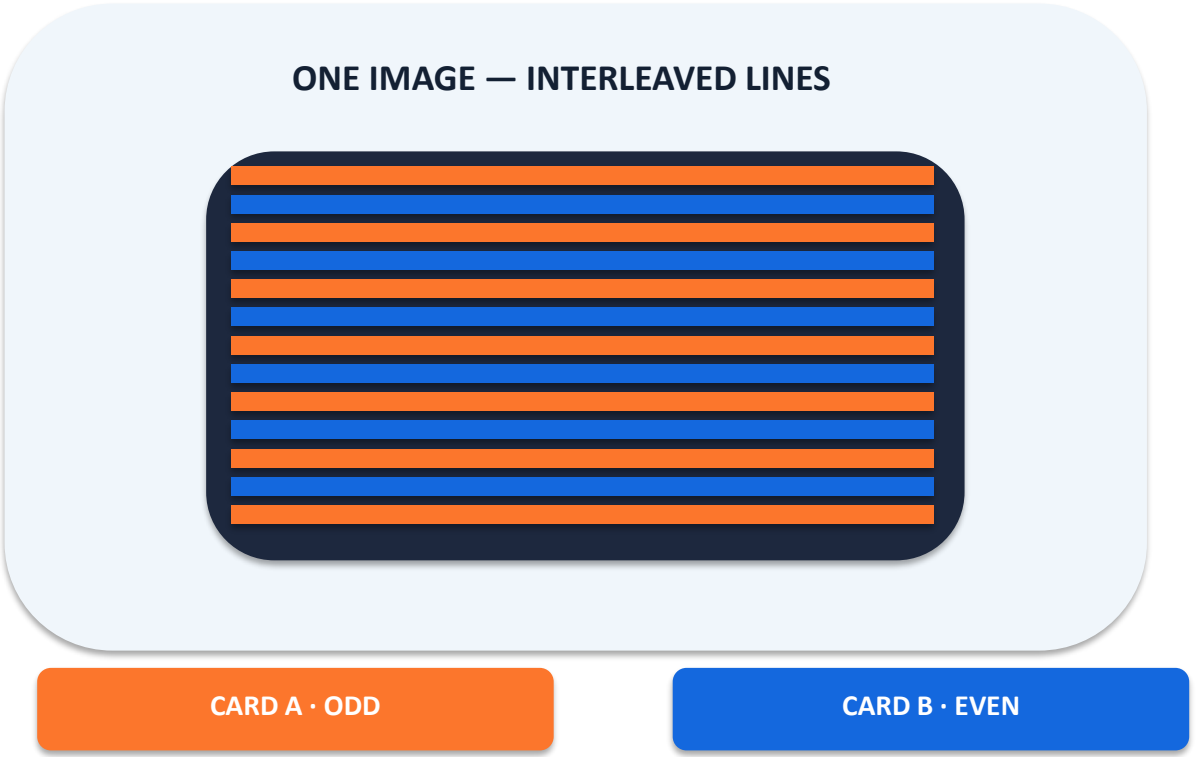
Early competitors like Nvidia's NV1 tried to cram 2D, 3D, and even sound onto a single chip, driving up costs and hurting performance. 3dfx dedicated 100% of its silicon budget strictly to 3D rendering

1998

3dfx Voodoo2: two cards, one image.

SLI = Scan-Line Interleave. A consumer PC could split rendering work across **TWO** 3D accelerators.





The original setup still required a separate 2D graphics card.

1999

NVIDIA GeForce 256: the “GPU” is born.

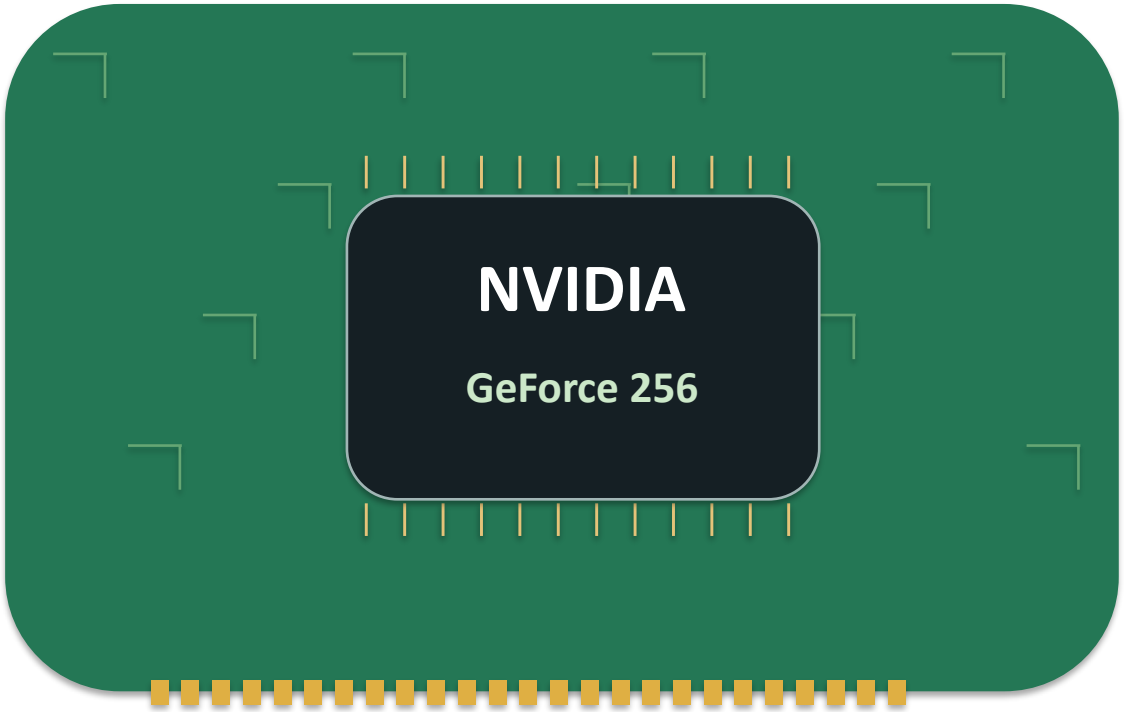
The phrase GRAPHICS PROCESSING UNIT becomes a mainstream product category.

1999 · GEFORCE 256

G P U

Graphics Processing Unit

NVIDIA marketed the GeForce 256 as the world’s first GPU—popularizing the name used ever since.



Hardware Transform & Lighting (T&L)

1990S

What that extra card made possible.

3D accelerators turn polygons into interactive worlds.



Stanford CS149 / 3D teapot illustration

GEOMETRY



Epic Games Nanite demo, via Stanford CS149

PIXELS + LIGHTING

2000–2006

The GPU industry takes shape.

The Voodoo era ends, but competing graphics lineages continue.

3dfx → **NVIDIA**

2000–01 · assets and intellectual property

ATI → **AMD**

2006 · company acquisition

Different corporate routes. Both helped shape the modern GPU ecosystem.

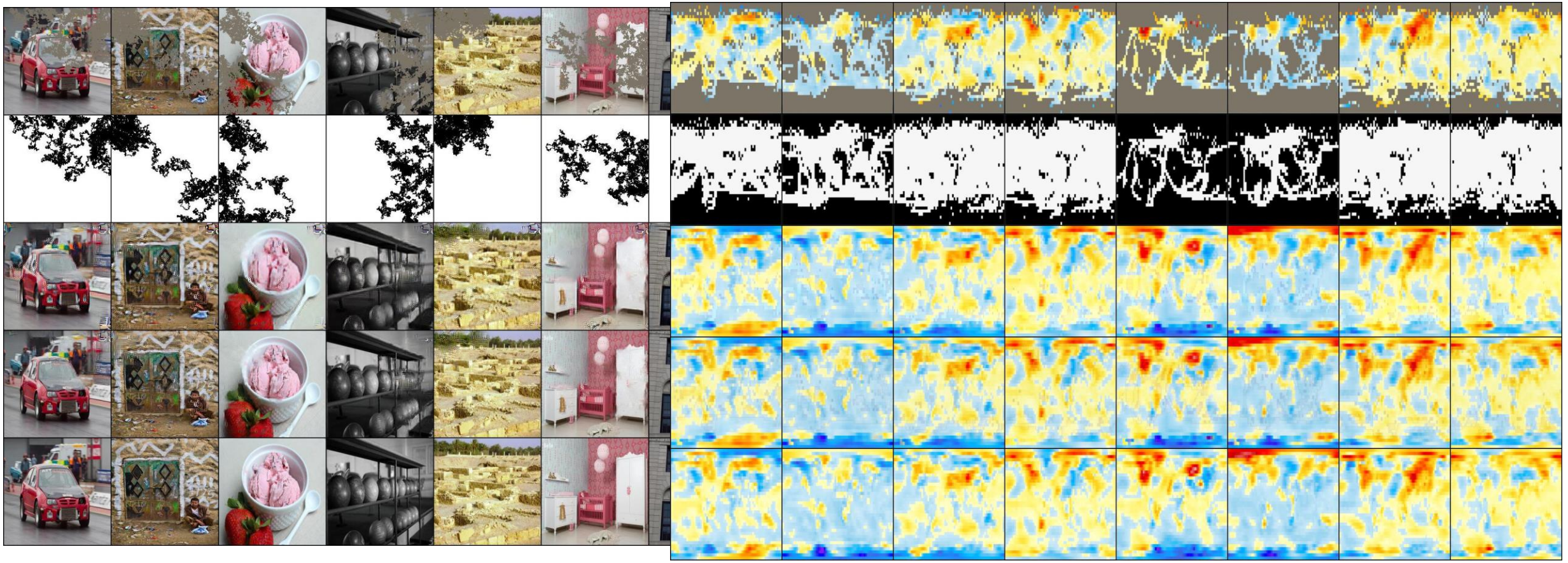
2001–2003

Someone noticed: pixels are just data.

The same recipe applied to many pixels looks a lot like scientific computing.

PIXELS

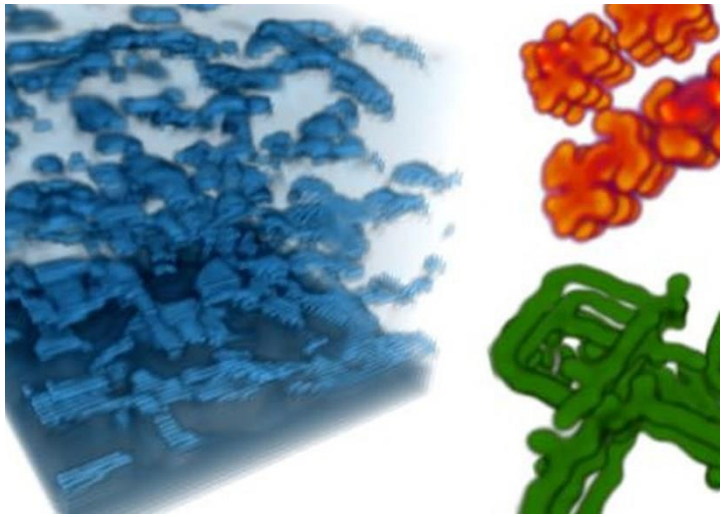
ARRAY ELEMENTS



2002–2003

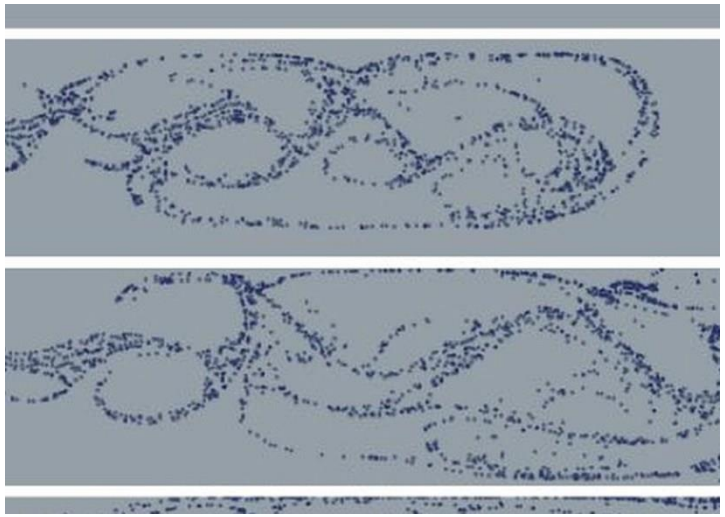
The first scientific GPU hacks.

Scientists used the graphics pipeline to do mathematics.



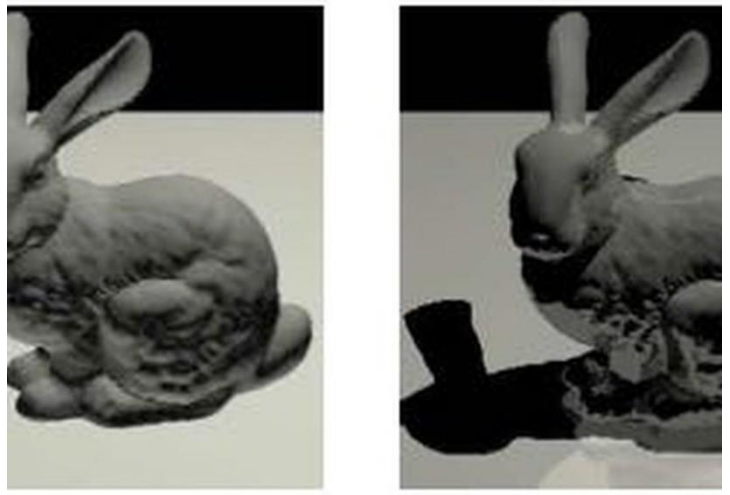
Harris 2002 / Stanford CS149

Lattice simulation



Bolz et al. 2003 / Stanford CS149

Sparse matrix solvers



Purcell et al. 2002 / Stanford CS149

Ray tracing

The accelerator was already fast. The programming interface was the problem.

2006–2007

CUDA opens the GPU to programmers.

A better programming model turns a graphics accelerator into a computing platform.

BEFORE

Graphics commands
Shader workarounds
Textures as arrays

AFTER

Launch compute kernels
Use arrays directly
Build reusable libraries

CUDA + compiler tools + libraries = the real platform shift.

2008 / JAPAN

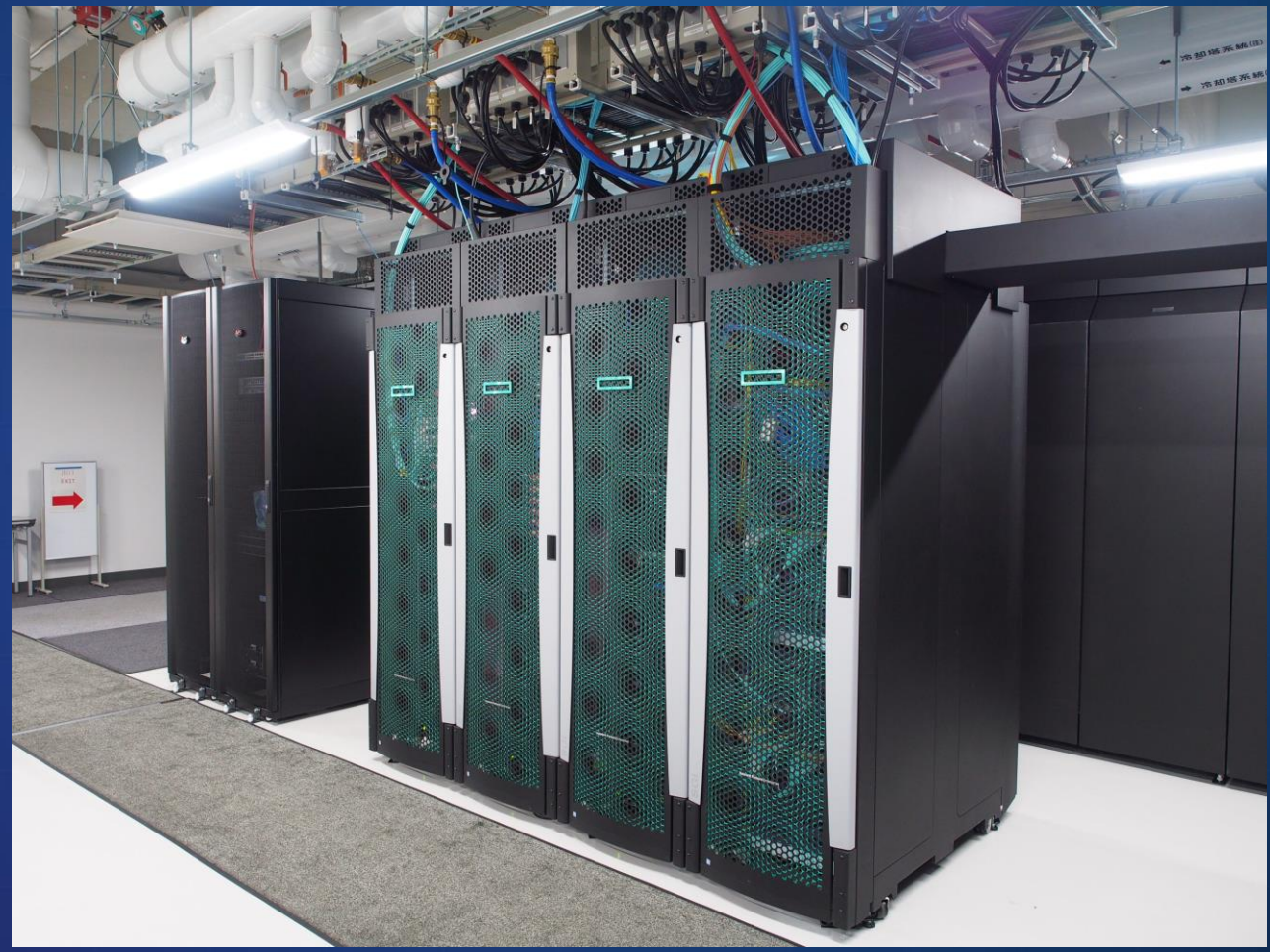
The GPU enters the supercomputer.

TSUBAME 1.2 · TOKYO TECH

680 NVIDIA TESLA GPUs

170 Tesla S1070 accelerator systems

77.48 TFLOPS LINPACK · November 2008



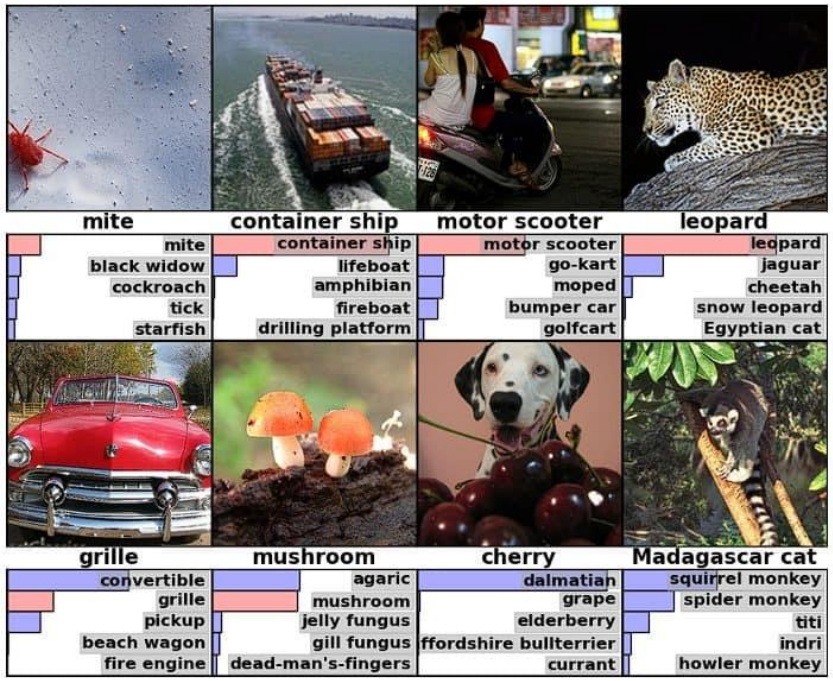
CPU + GPU HYBRID CLUSTER

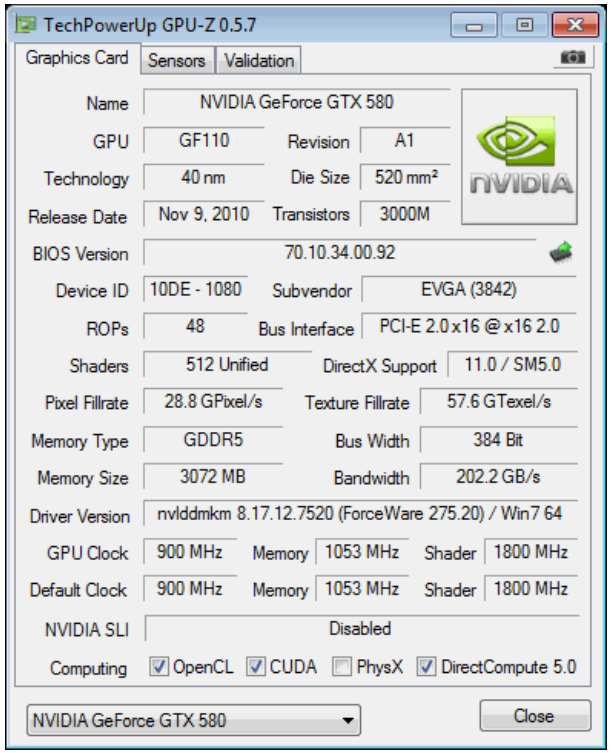
FIRST GPU-BASED HYBRID SUPERCOMPUTER TO ENTER THE TOP500

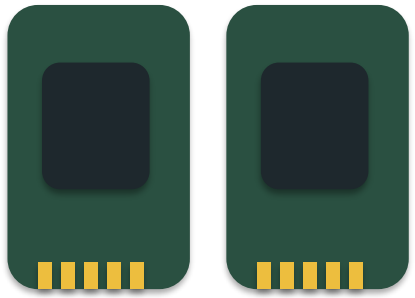
2012

AlexNet: GPUs help change AI.

The big breakthrough came from combining algorithms, large datasets, and GPU compute.







TWO GPUs

ALEXNET

2012 ImageNet win

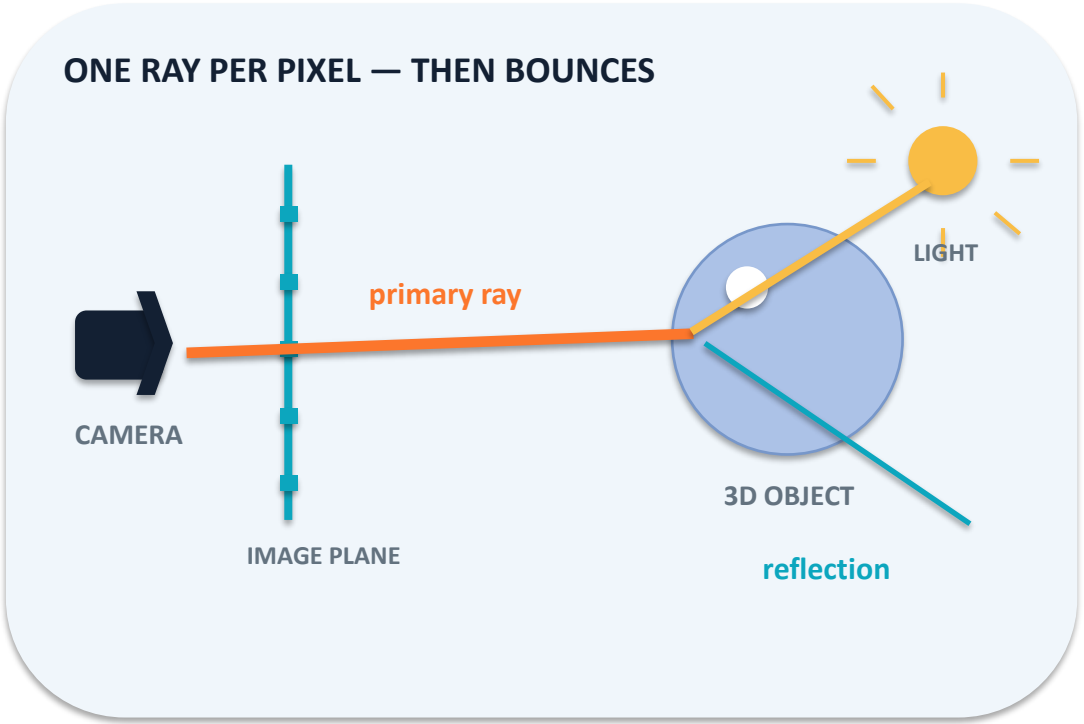
Deep learning becomes a powerful new direction for computer vision.

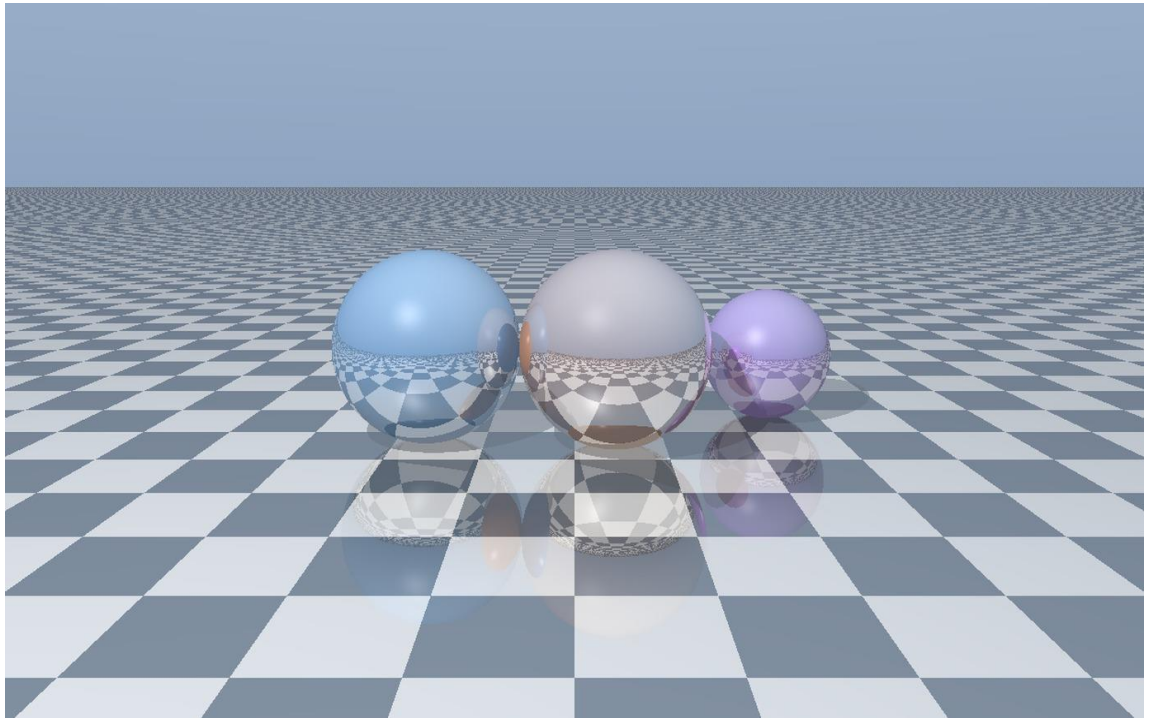
This was not just better hardware. The complete software + data + model stack mattered.

2018 · REAL-TIME RAY TRACING

Ray tracing: follow the light.

A decades-old rendering technique becomes interactive with dedicated GPU hardware.





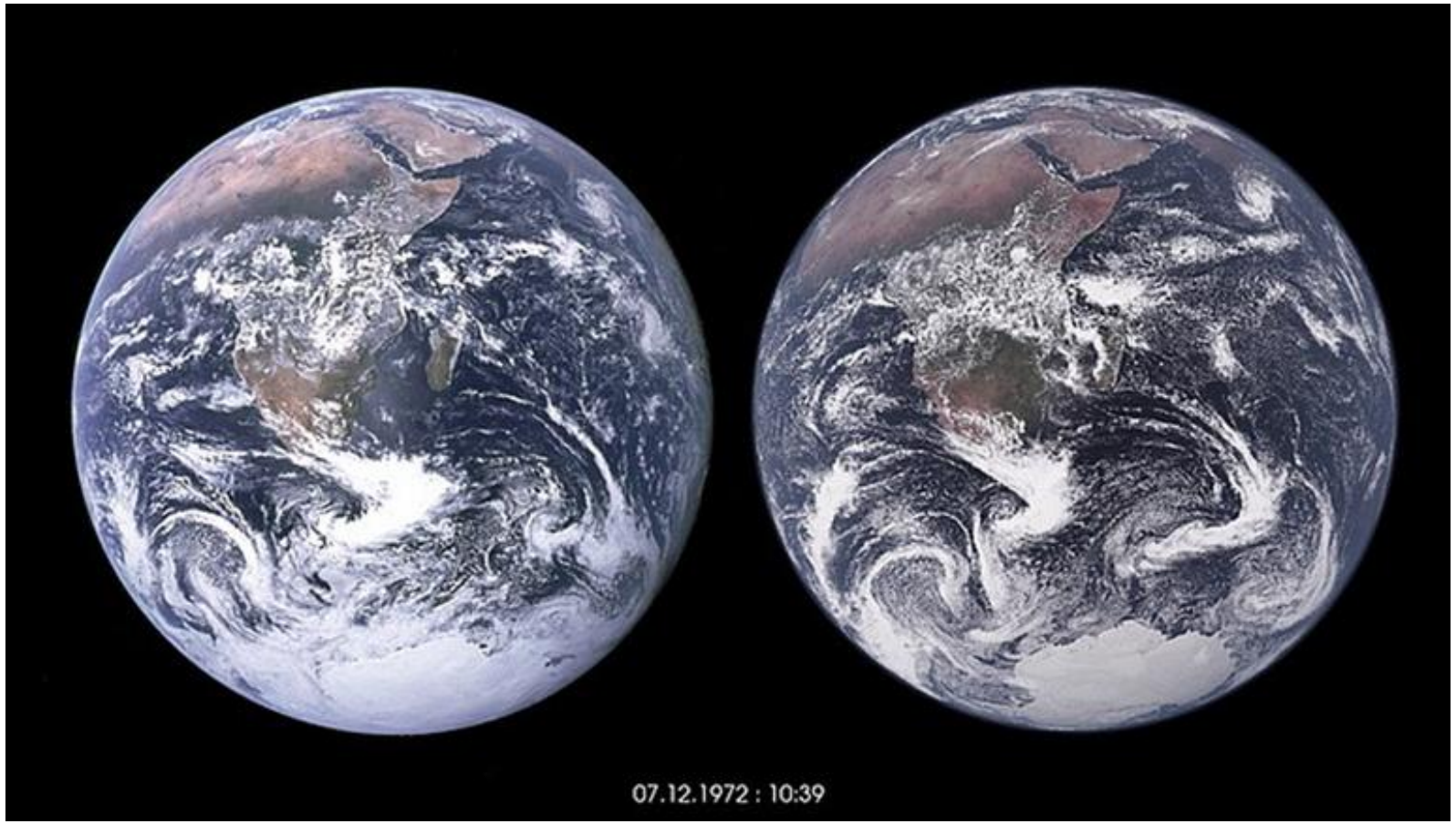
REALISTIC REFLECTIONS · SHADOWS · LIGHTING

Original ray-traced render + editable teaching diagram

TODAY

From pixels to climate simulations.

A shader for each pixel becomes an operation for each grid cell.



NASA Blue Marble image

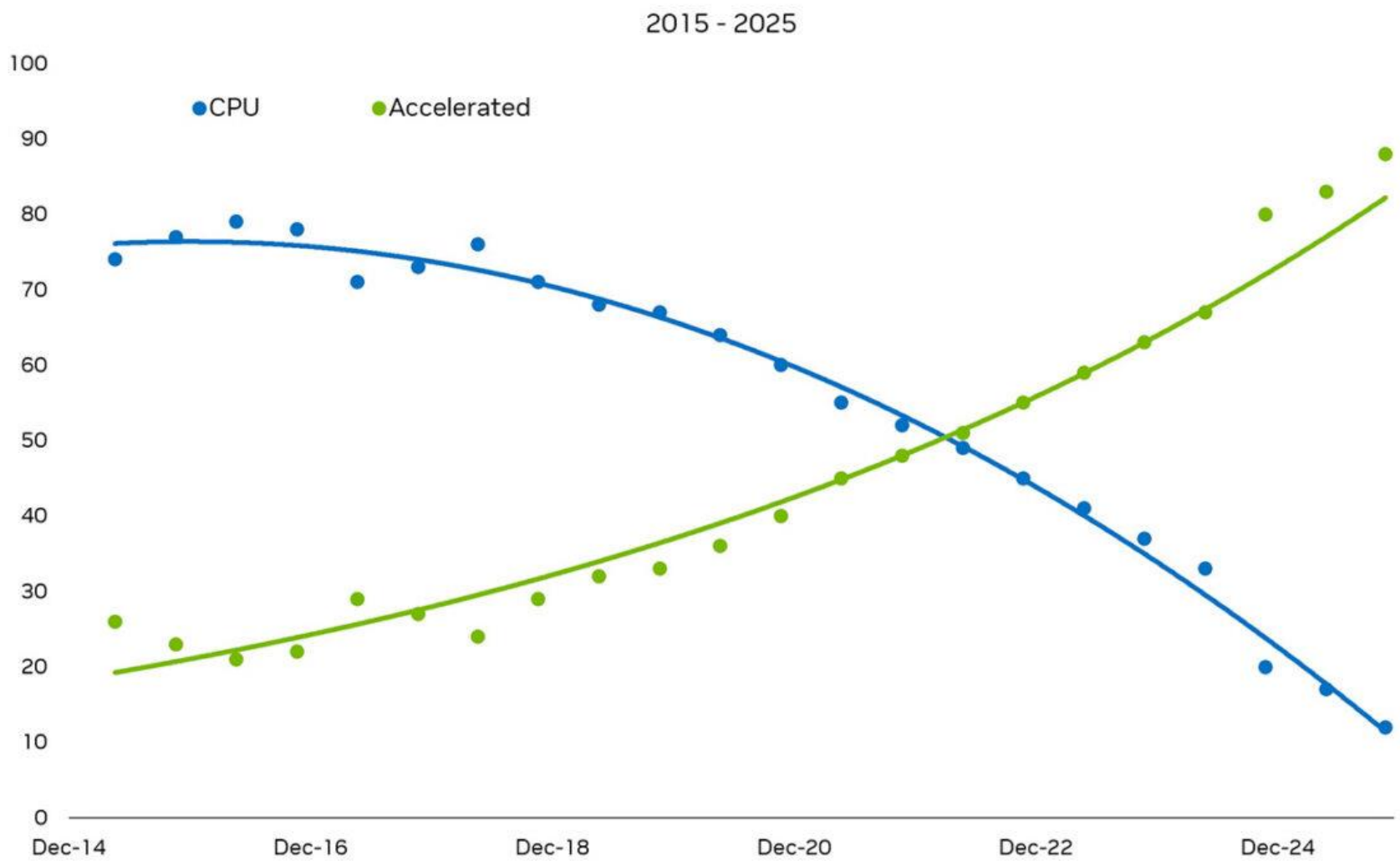
EARTH OBSERVATION

Illustrative scientific field

CLIMATE MODEL FIELDS

Top500 Transition from CPU to GPU

History of Acceleration of the Top100



Hardware & Software

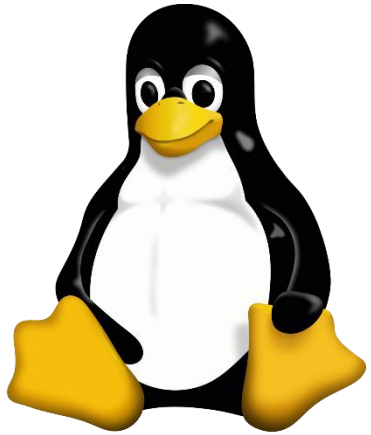
Great News: Deep Learning, Machine Learning, Artificial Intelligence is possible on **CPU, GPU and TPU**











Hardware & Software

Great News: Deep Learning, Machine Learning, Artificial Intelligence is possible on **CPU, GPU and TPU**



You can use bigger or high performance computer like DKRZ or UHH.

Or also you could use Amazon Web Services (AWS) or Google Cloud.

DKRZ DEUTSCHES KLIMARECHENZENTRUM

UHH

aws

Google Cloud

Hardware & Software





JUPITER MODULES

JUPITER Booster

- ~125 Racks BullSequana XH3000
- Node design
  - ~6000 nodes
  - 4× NVIDIA CG1 per node
- CG1: NVIDIA Grace-Hopper
  - 72 Arm Neoverse V2 cores (4×128b SVE2); 120 GB LPDDR5
  - H100 (132 SMs); 96 GB HBM3
  - NVLink C2C (900 GB/s)



JUPITER Cluster

- ~14 Racks BullSequana XH3000
- Node design
  - ~1300 nodes
  - 2× SiPearl Rhea1 per node
- Rhea1
  - 80 Arm Neoverse V1 cores (2×256b SVE)
  - 256 GB DDR5, 64 GB HBM2e



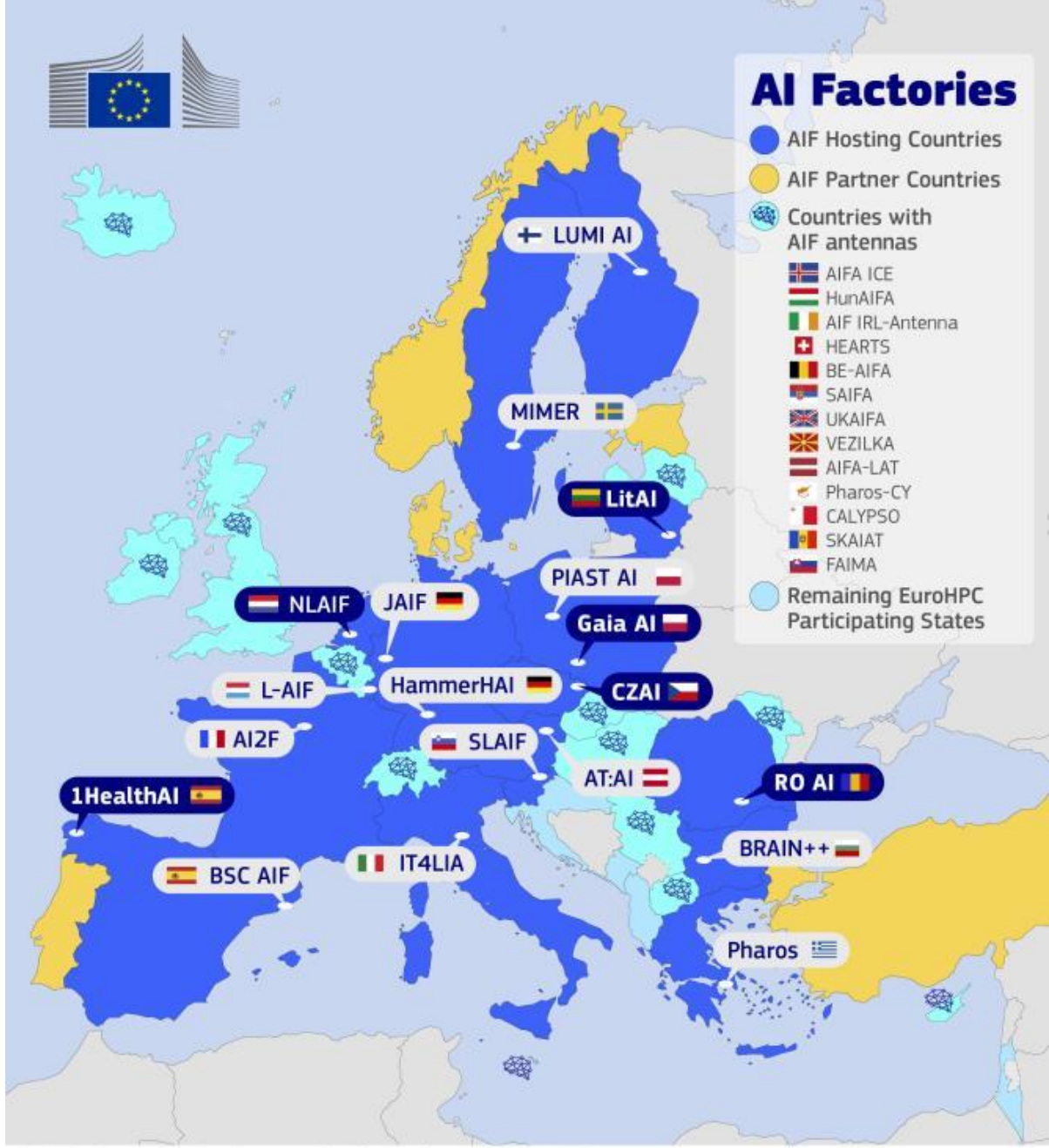
Preliminary numbers, might change during ins

Hardware & Software



JUPITER
AI FACTORY





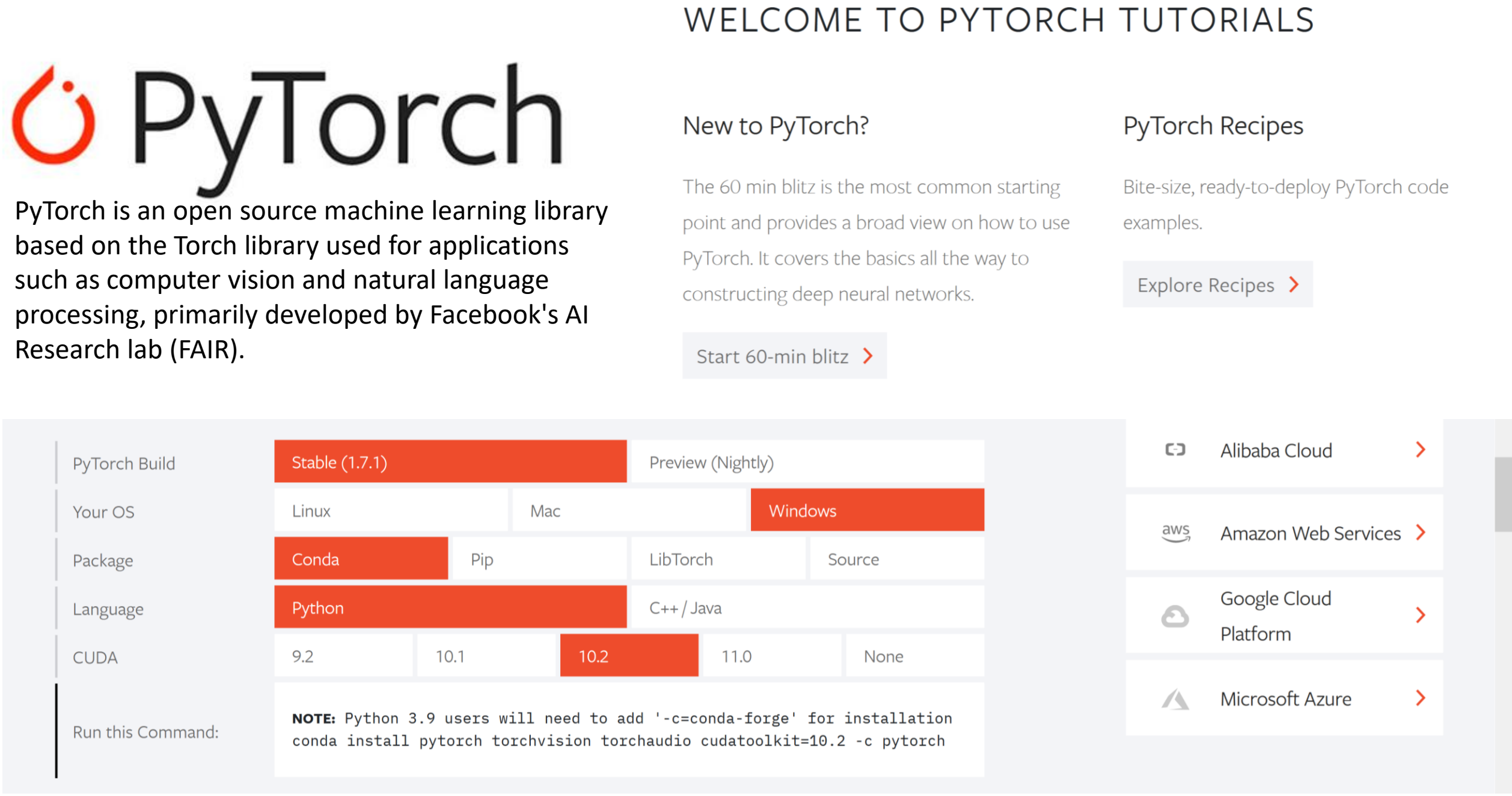
Switzerland's participation is contingent upon the ratification of its accession to Horizon Europe.

The boundaries and names shown and the designations used on this map do not imply official endorsement or acceptance by the European Union. This designation shall not be construed as recognition of a State of Palestine and is without prejudice to the individual positions of the Member States on this issue.

Administrative boundaries: © EuroGeographics © OpenStreetMap
Cartography: Eurostat – IMAGE, 05/2025

Hardware & Software

PyTorch is an open source machine learning library based on the Torch library used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR).



THE LESSON LEARNED

Hardware creates possibilities. Software unlocks them.

This is why the history of GPUs is also the history of programming models.

01

HARDWARE

Dedicated acceleration

02

SOFTWARE

CUDA, compilers, libraries

03

APPLICATIONS

AI, climate, scientific HPC

A fast accelerator is useful only when algorithms can exploit it efficiently.

NOW THE DETAILED LECTURE

What can actually happen at the same time?

Dependencies. Memory. Data movement. Communication.

A GPU can do many calculations in parallel.
But which calculations are independent?



NEXT → THE PANCAKE EXAMPLE & PARALLEL DEPENDENCIES

From parallelism to GPU programming

Scientific Computing Lecture

Georgiana Mania

19.11.2026

German Climate Computing Center (DKRZ)

Outline

- **Part 1:** Parallelism in HPC
- **Part 2:** CPU vs GPU architecture in HPC supercomputers
- **Part 3:** Programming GPUs

Part 1: Parallelism in HPC

Building intuition

Imagine that running large scientific applications on supercomputers is similar to ...

baking pancakes!

There are some steps that must happen in a specific order:



1. Prepare!



2. Whisk!



3. Bake!

Source of images 1, 2: thespruceeats.com

What can be parallelized?



the task



sequential



parallel processes



parallel threads

Images generated by Pradipta Samanta (DKRZ) with DALL-E

Precondition for parallelization

Two consecutive actions can be executed in parallel if they are ***independent***.

What does **dependence** mean?

Dependency types

- **data** dependency – the pancakes cannot be baked unless the batter is ready!
- **resource** dependency – we cannot bake 2 pancakes in the same time if we have only 1 small pan!
- **control** dependency – the time for baking a pancake can increase/decrease if the stove's temperature is lowered/raised

Now back to computer science!

Bernstein's parallelism conditions

The intersection between read-write set, write-read set and write-write set of instructions is null.

– Bernstein's condition for data independence

```
c = a + b;  // S1
d = a - b;  // S2
```

Read and write sets for S1 and S2:

$$R_1 = \{a, b\};\ W_1 = \{c\}$$
$$R_2 = \{a, b\};\ W_2 = \{d\}$$

Bernstein's conditions:

$$R_1 \cap W_2 = \emptyset$$
$$W_1 \cap R_2 = \emptyset$$
$$W_1 \cap W_2 = \emptyset$$

S1 and S2 can be executed in parallel!

Bernstein's parallelism conditions

```
c = a + b;   // S1
d = a - b;   // S2
```

Read and write sets for S1 and S2:

$$R_1 = \{a, b\};\ W_1 = \{c\}$$
$$R_2 = \{c, b\};\ W_2 = \{d\}$$

Replace a with c in statement 2

```
c = a + b;   // S1
d = c - b;   // S2
```

Bernstein's conditions:

$$R_1 \cap W_2 = \emptyset$$
$$W_1 \cap R_2 = \{c\}$$
$$W_1 \cap W_2 = \emptyset$$

S1 and S2 can NOT be executed in parallel!

Results of parallel execution

- Parallelization **should not change** the results, there are exceptions for floating-point precision!
- Potential problems
  - Faulty domain decomposition
  - Wrong communication pattern
  - Race conditions

Speedup and efficiency

Speedup

$$S(n) = \frac{T(1)}{T(n)}$$

where

- n = the number of processors
- $T(1)$ = time to execute the algorithm with 1 process
- $T(n)$ = time to execute the algorithm with n processes

Speedup efficiency

$$E(n) = \frac{S(n)}{n}$$

where

- $S(n)$ = speedup
- n = the number of processors

Strong scaling

Strong scaling refers to increasing the number of processes n times while keeping the problem size **constant**, with the goal to reduce the total execution time.

use 4 pans to bake a total of 8 pancakes $\Rightarrow$ reduce baking time



Weak scaling

Weak scaling refers to increasing the number of the processes n times while also increasing the problem size n times so that the amount of work per processor remains **constant**.

use 4 pans to bake a total of 4×8 pancakes $\Rightarrow$ bake $4\times$ more pancakes in the same time



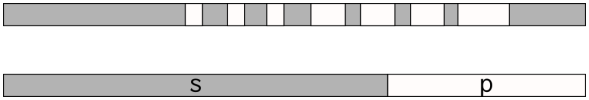
Strong vs weak scaling

Starting with 1 pan, we can (perfectly) scale this by using N pans in two ways:

Parameter	Initial	Strong scaling	Weak scaling
Resources (e.g. pans)	1	N	N
Total workload (e.g. pancake count)	8	8	$8 \times N$
Workload per worker (e.g. pancakes per pan)	8	$8/N$	8
Total time	T	T/N	T

Strong scaling: Amdahl's law

- Q: Can a parallel algorithm achieve 100% efficiency speedup (running it with n processes reduces the time n times)?



A: No, that's idealistic!

- $S(n)$ has an upper bound given by **Amdahl's law**

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

where p is the fraction of the program that can be executed in parallel

Weak scaling: Gustafson's law

Typically, the serial part of an algorithm does not grow with the problem size, while the parallel part grows with n. The benefits are described by **Gustafson's law**:

$$S(n) = s + p \times n$$

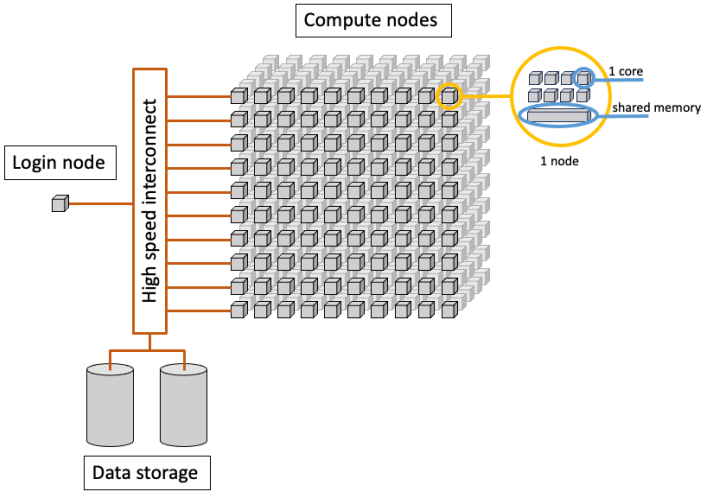
where s and $p = 1 - s$ are the serial and parallel fractions of the run time **on** n **processes**

Part 1: Summary

- Only independent tasks can run in parallel – check the dependencies (Bernstein's conditions)
- Parallelization **should not change** the results – beware of race conditions
- The speedup of a program is limited by its sequential part s (Amdahl's law); growing the problem with the resources helps (Gustafson's law)
- Assigning more resources doesn't always bring the best performance, e.g. more MPI tasks means more communication between the processes (i.e. increasing s)
- Always check results against a benchmark produced on the same hardware, using the same compiler (family) and the same compilation flags

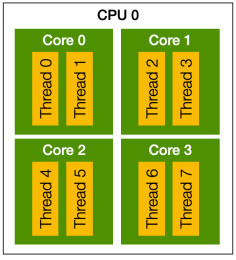
Part 2: CPU vs GPU architecture in HPC supercomputers

HPC supercomputer



Execution units in CPU nodes

Each CPU node has several **processors (CPU)**, each having multiple **cores**, with 1–2 threads/core



Schematic CPU [source: DKRZ]

e.g. Levante CPU node = 2 processors $\times$ 64 cores/processor $\times$ 2 threads/core

Execution units in GPU nodes

- Each GPU node has multiple **streaming multiprocessors (SM)**
- each SM has multiple cores
- and can run multiple of 32 threads at once



NVIDIA A100 [source: NVIDIA Ampere architecture whitepaper]

CPU vs GPU: two design philosophies

	CPU (AMD EPYC 7763)	GPU (NVIDIA A100)
Cores	64 complex cores	108 SMs $\times$ 32 = 3456 simple FP64 cores
Threads in flight	128 (2 per core)	$>$ 200 000 (2048 per SM)
Peak FP64	$\approx$ 2.5 TFLOP/s	9.7 TFLOP/s (19.5 with tensor cores)
Memory bandwidth	$\approx$ 0.2 TB/s (DDR4)	$\approx$ 1.6–2 TB/s (HBM2)
Optimized for	**latency**: large caches, branch prediction, out-of-order execution	**throughput**: hide memory latency by switching between thousands of threads

A CPU is a few skilled chefs; a GPU is a kitchen full of helpers who all flip a pancake **at the same time**.

Threads, warps and blocks

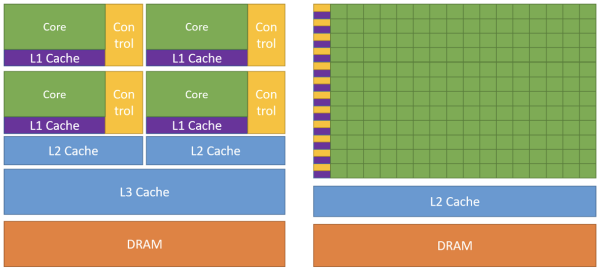
- The basic execution unit is the **thread**
  - CPU: 1–2 threads per core, organised in **teams** (e.g. OpenMP)
  - GPU: threads are grouped in **warps** (32 threads on NVIDIA, 64 on AMD) executed in **lockstep** on an SM

Parallelism execution

- **MIMD**: Multiple Instructions, Multiple Data
  - Each unit executes a **different** instruction on **different** chunks of data $\Rightarrow$ **task-level parallelism**
  - E.g. multiple cores in a CPU
- **SIMD**: Single Instruction, Multiple Data
  - Each unit executes **the same** instruction on **different** chunks of data $\Rightarrow$ **data-level parallelism**
  - E.g. a Streaming Multiprocessor in a GPU

Memory resources inside a node

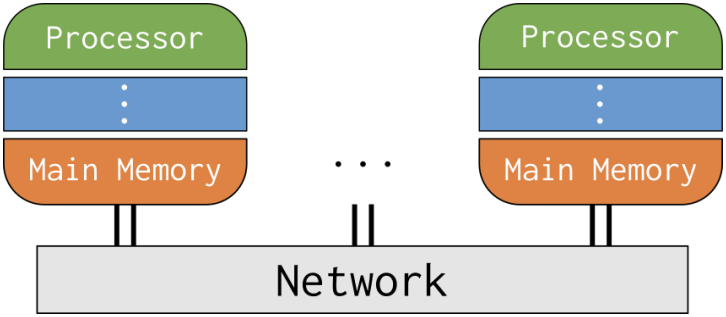
- all cores inside a node **share** the same global memory space
- there is a **memory hierarchy**: L1, L2, L3?, global memory
- better **latency** is associated with reduced storage capacity: L1 ($\sim$1 ns), L2 ($\sim$5 ns), L3 ($\sim$10 ns), DRAM ($\sim$100 ns)
- overall a high memory bandwidth is important for performance



Shared memory model [source: CUDA C programming guide]

Distributed memory resources

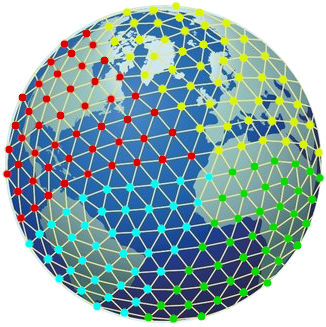
- each node has its private memory, but nodes could share data via network $\Rightarrow$ the application running on multiple nodes has **distributed memory**
- data communication between nodes via **Message Passing Interface (MPI)**



Distributed memory model [source: DKRZ]

HPC supercomputer can do parallel computing

- split a problem into smaller subtasks (or processes), which can be solved simultaneously by different resources



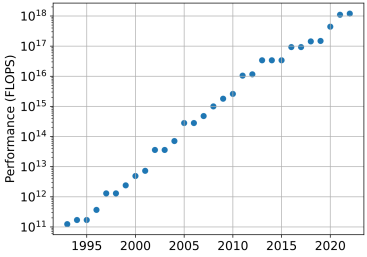
Part 2: Summary

- A supercomputer consists of many **nodes** with CPUs and/or GPUs
- CPU: few complex cores; GPU: many SMs running thousands of simple threads (SIMD)
- Inside the node, all the execution units (cores, threads) share the global memory
- Moving beyond one node implies sending/receiving data to/from neighboring domains
- Memory's size and bandwidth impact performance

Part 3: Programming GPUs

Why GPUs?

- Moore's law (1965): *the number of transistors on a chip doubles roughly every 2 years*, but single-core clock speed stopped growing around 2005
  - more performance must come from **parallelism**
- Modern supercomputers: many CPU cores **& accelerators (GPUs)**
  - 9/10 fastest supercomputers in the world use GPUs (Top500, 2025)
- GPUs deliver more FLOP/s and memory bandwidth **per Watt**



Peak performance of the fastest Top500 system [source: top500.org]

Parallel programming languages

	Intra-node	**Inter-node**
Memory	shared	distributed
Logical execution unit	thread	process
Communication	automatic, due to shared memory	explicit, via **M**essage **P**assing **I**nterface
Parallel APIs	OpenMP, OpenACC	OpenMPI (G2G)
Vendor APIs	CUDA, HIP	
Parallel programming frameworks	Kokkos, SYCL	

The APIs can require runtime libraries and/or compiler support.

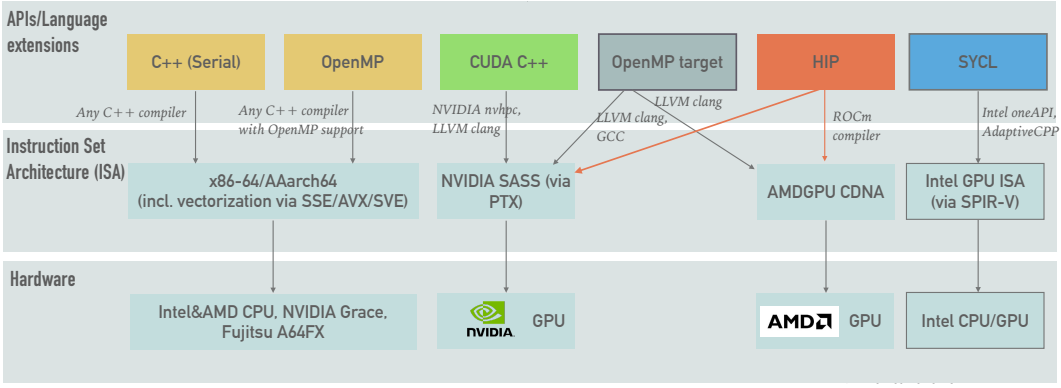
Host and device: the offload model

- CPU = **host**, GPU = **device**, each with **its own memory**
- Typical steps:

1. allocate memory on the device
2. copy input host $\rightarrow$ device
3. launch the **kernel** (runs on GPU)
4. synchronize: wait for the kernel
5. copy results device $\rightarrow$ host
6. free device memory

The host–device link (PCIe $\approx$ 32 GB/s) is $\sim$50$\times$ slower than GPU memory $\Rightarrow$ **keep the data on the GPU**

From source code to hardware: compiling for CPUs and GPUs



Kokkos supports **all these APIs as backends**: one source code, the backend is chosen at compile time.

Hello world of GPU computing: SAXPY

$$y \leftarrow a x + y$$ Single-precision A·X Plus Y

serial C (code/saxpy_serial.c)

```
void saxpy(int n, float a, const float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

- Every iteration `i` is **independent**: Bernstein's conditions hold for any pair of iterations
- 2 FLOPs per 12 bytes moved ⇒ limited by **memory bandwidth**, not compute

Q: What has to change to run this on a GPU? **A: who computes iteration i, and where x and y live.**

SAXPY with OpenMP: directives

CPU threads

```
#pragma omp parallel for
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
```

GPU offload (code/saxpy_openmp.c)

```
#pragma omp target teams distribute parallel for \
            map(to: x[0:n]) map(tofrom: y[0:n])
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
```

- The loop stays unchanged – the compiler is told **what** to parallelize via `#pragma`
- `target` offloads to the device; `teams` / `parallel for` spread the iterations over groups of GPU threads
- `map(to/from)` copies the data between host and device

```
gcc -O3 -fopenmp saxpy_openmp.c             # CPU threads
nvc -O3 -mp=gpu -gpu=cc80 saxpy_openmp.c    # GPU offload (NVIDIA HPC SDK)
```

SAXPY in CUDA: the kernel

code/saxpy_cuda.cu

```
__global__ void saxpy(int n, float a, const float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // global thread id
    if (i < n)                                      // last block may overhang
        y[i] = a * x[i] + y[i];
}
```

- `__global__`: function runs on the GPU and is launched from the host
- **One thread per element** – the `for` loop has disappeared!
- Each thread computes its global index from its block and thread id
- `if (i < n)`: we launch whole blocks, so there may be more threads than elements

SAXPY in CUDA: the host code

```
float *d_x, *d_y;
cudaMalloc(&d_x, bytes);                                  // 1. allocate
cudaMalloc(&d_y, bytes);
cudaMemcpy(d_x, x, bytes, cudaMemcpyHostToDevice); // 2. to GPU
cudaMemcpy(d_y, y, bytes, cudaMemcpyHostToDevice);

int threads = 256;                                        // per block
int blocks  = (n + threads - 1) / threads;          // cover n
saxpy<<<blocks, threads>>>(n, 3.0f, d_x, d_y);      // 3. launch
cudaDeviceSynchronize();                                  // 4. synchronize

cudaMemcpy(y, d_y, bytes, cudaMemcpyDeviceToHost); // 5. back
cudaFree(d_x); cudaFree(d_y);                             // 6. free
```

```
nvcc -O3 -arch=sm_80 saxpy_cuda.cu -o saxpy_cuda    # A100 = sm_80
```

- **explicit** memory management
- launch configuration `<<<blocks, threads>>>`
- kernel launch returns immediately ⇒ synchronize before copying back
- Q: which part is slowest?

  A: the copies!

Kokkos in a nutshell

C++ performance portability programming model which provides abstractions for both parallel execution of code and data management – *kokkos.org*

- **Same code** runs on CPUs and on GPUs from different vendors
- The **backend** is chosen at compile time: Serial, OpenMP, CUDA, HIP, SYCL
- **Data:** `Kokkos::View` – multi-dimensional array, templated on memory space, layout and traits
- **Execution:** `parallel_for`, `parallel_reduce`, `parallel_scan` – templated on execution space, pattern and policy
- `Kokkos::fence()` waits for kernels; `Kokkos::deep_copy()` synchronizes host and device data

```
cmake -DKokkos_ENABLE_OPENMP=ON -DKokkos_ENABLE_CUDA=ON ..   # one CPU + one GPU backend
```

SAXPY with Kokkos

code/saxpy_kokkos.cpp

```
Kokkos::initialize(argc, argv);
{
  Kokkos::View<float *> x("x", n), y("y", n);

  // ... initialize x and y here (needed before saxpy)
  Kokkos::parallel_for("saxpy", n, KOKKOS_LAMBDA(const int i) {
    y(i) = a * x(i) + y(i);
  });
  Kokkos::fence();                // kernels are asynchronous

  auto y_host = Kokkos::create_mirror_view_and_copy(
                    Kokkos::HostSpace(), y);
}
Kokkos::finalize();
```

Q: Where do `x` and `y` live?
A: It depends on how the code is compiled: GPU memory with CUDA, host memory with OpenMP.

Q: How is `"saxpy"` executed?
A: In parallel, by CUDA threads or OpenMP threads – same source!

Same loop, three programming models

	CUDA	OpenMP	Kokkos
What is it?	language extension	compiler directives	C++ template library
Hardware	NVIDIA only (HIP for AMD)	CPU + GPUs with compiler support	CPU + NVIDIA, AMD, Intel GPUs
Parallel unit	kernel, 1 thread / element	annotated loop	lambda in `parallel_for`
Data movement	explicit `cudaMemcpy`	`map` clauses	`View` + `deep_copy`
Code changes	rewrite loops as kernels	add pragmas	rewrite with Views + lambdas

In practice: **MPI + X** – MPI between nodes/GPUs, X inside a node.

Part 3: Summary

- GPUs deliver most of the compute power of today's fastest supercomputers
- Offload model: host and device have separate memories $\Rightarrow$ keep the data on the GPU
- Low level: **CUDA** – explicit kernels, launches, copies and synchronization
- Directives: **OpenMP** – annotate the existing loop, `map` the data
- Portable: **Kokkos** – one C++ source, the backend is chosen at compile time
- In practice: **MPI + X**

Key takeaways

Parallelism in HPC

- Dependencies & Bernstein's conditions
- Speedup and efficiency
- Strong and weak scaling
- Amdahl's and Gustafson's laws

CPU vs GPU architecture

- Supercomputer: nodes with CPUs, GPUs
- Parallelism paradigms: MIMD, SIMD
- Memory: hierarchy, shared, distributed

Programming GPUs

- Threads and warps; one thread per element
- Offload model: keep data on the device
- Same loop in CUDA, OpenMP and Kokkos