Geometric data analysis

Lecture 7/7 – GPU programming

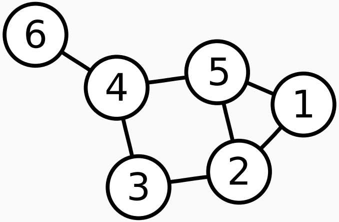
Jean Feydy
HeKA team, Inria Paris, Inserm, Université Paris-Cité

Thursday, 9am–12pm – 7 lectures

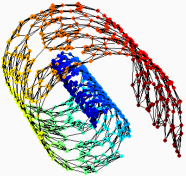
Faculté de médecine, Hôpital Cochin, rooms 2001 + 2005

Validation: project + quizz

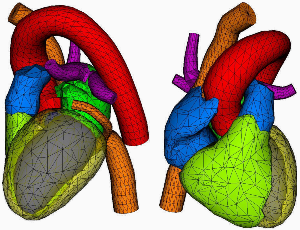
Towards a continuous analysis of large datasets [Pey11, EPW11]



Simple graph.

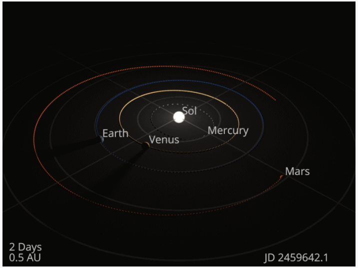


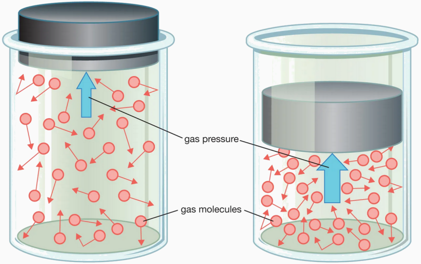
Manifold **hypothesis**.



Physical manifold.

Long history in physics [Dat18, Bri, NWRC22]





The **Solar** system. The **ideal gas** model. **Fluid** simulation.

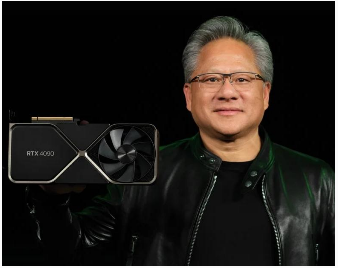
Research in **physics** $\Longleftrightarrow$ **High Performance Supercomputers**

Only available through large **institutional centers**.

Recent history around video games and movies

FFVII on the PS1 – 1997.

FFVII on the PS4 – 2020.



Jensen Huang – 2022.

Research in **graphics** $\Longleftrightarrow$ **Graphics Processing Units**

Affordable to any researcher: game-changer.

The “AI revolution” is primarily driven by hardware

Statistics and Machine Learning have been around for **decades**.

Breakthrough in 2010-15: hacking **PlayStations** for **science** became **easy**.

As AI researchers, we must understand:

1. **What is a GPU?**
    - Thousands of cores, complex **memory** management.
    - 4 rules of GPU programming.
2. **Current trends in the semiconductor industry**
    - Just-in-time compilation, custom AI chips.
    - **Supply chain** issues and their impact on our careers.

Accessible references

Coming from a **math background**:

- Chapter 2 of my PhD thesis, *Geometric data analysis, beyond convolutions*.
- Albert Chern's lecture notes at UCSD, *Introduction to computer graphics*.

Two **YouTube channels** to learn about **hardware**:

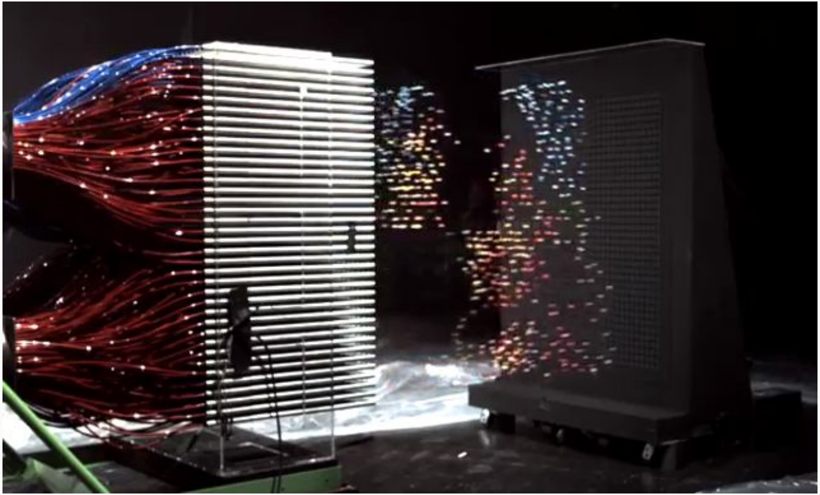
- *Branch Education* – to understand the circuits.
- *Asianometry* – to get some context on the industry.

Great software documentation – the source of Nvidia's monopoly in research:

- Mark Harris' posts on the Nvidia dev blog, GPU Gems textbooks.
- CUDA toolkit documentation, CUTLASS, CUB.

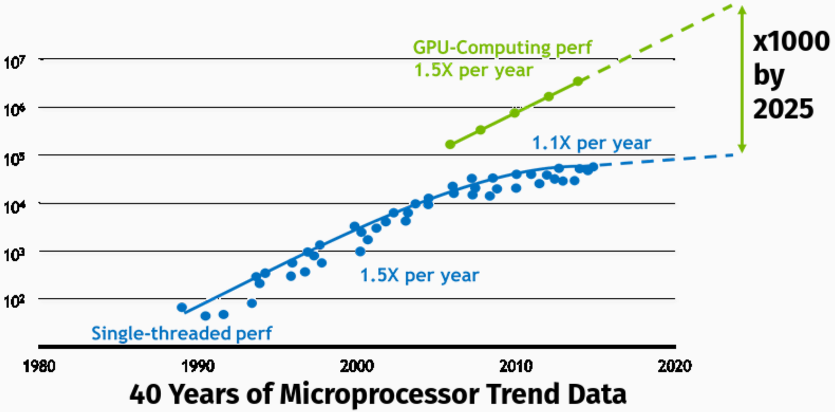
What is a GPU?

Nvidia focuses its marketing on economies of scale



Mythbusters Demo GPU versus CPU – 2009.

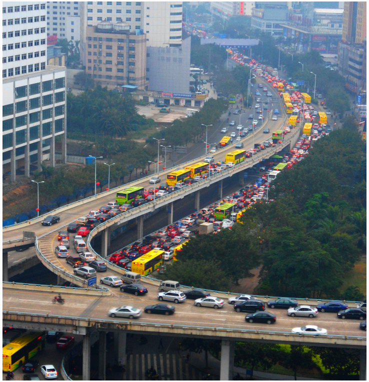
Nvidia focuses its marketing on economies of scale



Simple message: 10,000 cores $\Longrightarrow$ x1,000 acceleration vs. a 10-core CPU.

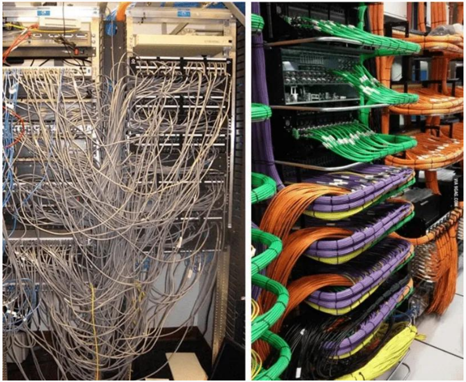
But **how did we fill those tubes** with the correct paintballs?

Scientific programs are memory bound [Fro12]



The curse of parallelism:

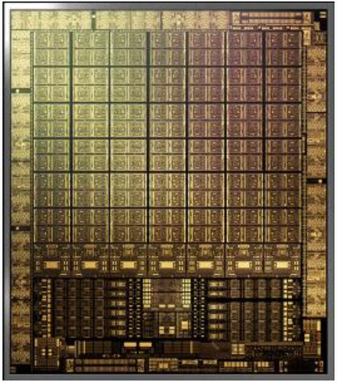
traffic jams.



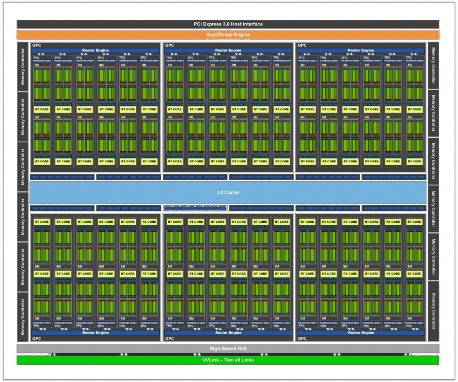
Structure is required. Design **choices** favor **“bankable”** program architectures.

Let’s open up a GPU

Let’s open up a GPU

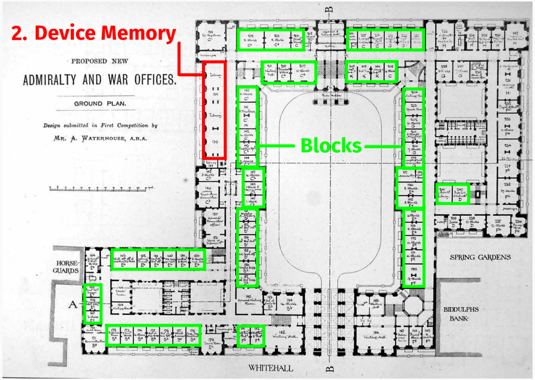


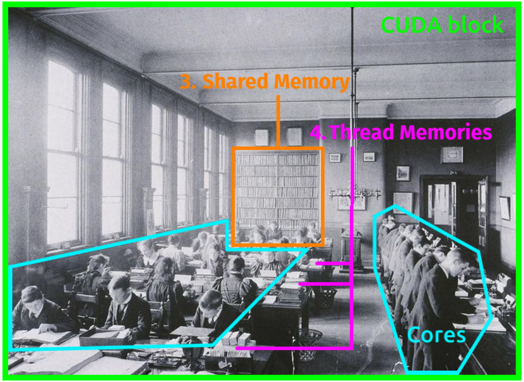
7,000 cores on a single GPU.



The Turing **architecture**.

GPUs and large administrations follow the same plan



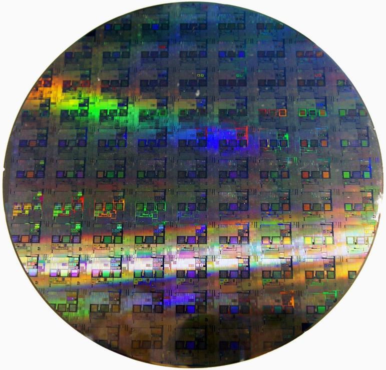


GPU $\simeq$ **100 redundant blocks.**

Inside a CUDA block: **workers and buffers.**

Redundancy is key to recover high yields in spite of defects [Dor97, Pee11]

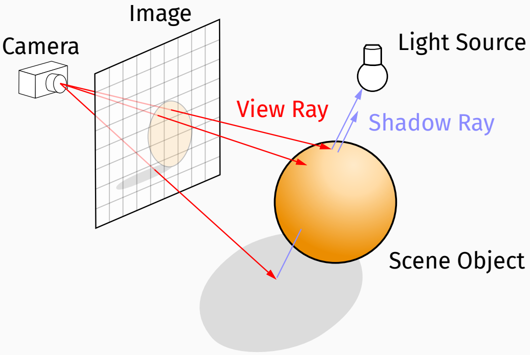
Silicon crystal.



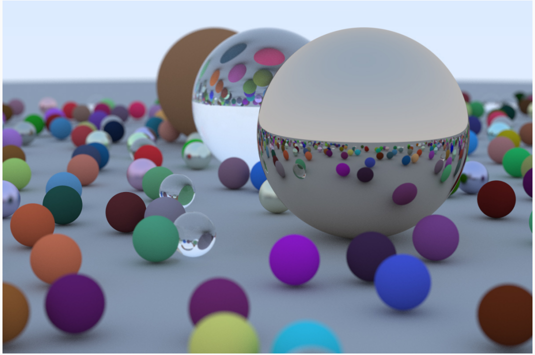
Chips are **etched onto silicon wafers**.

GeForce RTX **3090** $>$ GeForce RTX **3080** $>$ GeForce RTX **3070** $>$ …

GPUs are optimized to render 3D meshes in real time [Hen08, Shi20]



Simulating light rays.



Ray tracing in one weekend.

Nvidia GeForce **RTX** (**Ray Tracing** Texel eXtreme)

$\iff$ **Geometric** computations + **textures**, on independent **patches** of the screen.

5 main layers of memory storage

1 GPU $\simeq$ **100 blocks of 100 cores**.

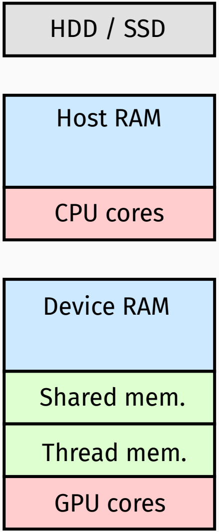
On the CPU host:

- **HDD / SSD** – 1 TB.
- **Host RAM** – 100 GB.

On the GPU device:

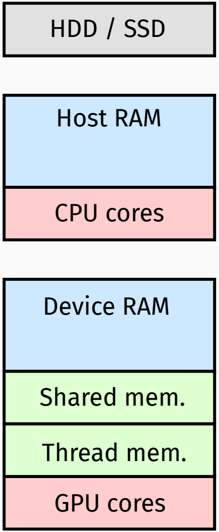
- **Device RAM** – 10 GB.
- **Shared block-wise memories** – **1 Kb/core**.
- **Thread-wise registers** – **1 Kb/core**.

Time(Device RAM $\leftrightarrow$ Core) $\simeq$ **100 arithmetic operations.**



4 rules of GPU programming

1. Promote **block-wise** parallelism.
2. Reduce **Host** $\leftrightarrow$ **Device** memory transfers.
3. Reduce **Device** $\leftrightarrow$ **Shared/Thread** memory transfers.
4. Promote **block-wise, contiguous memory accesses**.



The CUDA toolkit – a C++ dialect for GPU programming

```
__global__ void
My_CUDA_kernel(int param, float *device_data, float *device_output) {

    // We use the indices of the current thread and CUDA block
    // to assign each worker to its place in the computation plan:
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // We declare local variables as in standard C++.
    // They'll be stored in the Thread memory whenever possible:
    float some_value = 0;
    // We access the Shared memory through a raw C++ pointer:
    extern __shared__ float shared_mem[];

    // We handle transfers with a transparent interface:
    some_value    = device_data[i];  // Thread memory <- Device RAM
    shared_mem[i] = device_data[i];  // Shared memory <- Device RAM
```

The CUDA toolkit – a C++ dialect for GPU programming

```
    // Computations are written in standard C++ and executed in parallel
    // by all the threads of the CUDA block:
    for(int k = 0; k < param; k++) {
        some_value = some_value + k * shared_mem[i];
        ...
    }

    // We may create checkpoints for all threads in a CUDA block.
    // This may impact performances.
    __syncthreads();


    // We write results back to the Device RAM with:
    device_output[i] = some_value;  // Device RAM <- Thread memory
}
```

The CUDA toolkit – a C++ dialect for GPU programming

```
// The main C++ program, executed by the CPU:
int main(void) {
    int N = 1024; float *host_data, *host_out, *device_data, *device_out;

    // Allocate memory on the device - the API is a bit heavy:
    cudaMalloc((void**) &device_data,  N * sizeof(float));

    // Device RAM <- Host RAM:
    cudaMemcpy(device_data, host_data, N * sizeof(float),
               cudaMemcpyHostToDevice);

    // Set the parameters of the CUDA block:
    int block_size = 128; int grid_size  = N / block_size;
    int shared_mem_size = 2 * block_size * sizeof(float);
    // Run the GPU kernel:
    My_CUDA_kernel<<<grid_size, block_size, shared_mem_size>>>(...);
```

The CUDA toolkit – a C++ dialect for GPU programming

```
    // Wait for the GPU to finish its computations:
    cudaDeviceSynchronize();

    // Host RAM <- Device RAM:
    cudaMemcpy(host_out, device_out, N * sizeof(float),
               cudaMemcpyDeviceToHost);

    // Process and save the result "output array":
    ...

    // Don't forget to free the allocated memory:
    cudaFree(device_data);

    // And exit gracefully:
    return 0;
}
```

Recap on GPUs

1,000 € $=$ 1 GPU $=$ 100 $\times$ 100 cores with 5 main layers of memory:

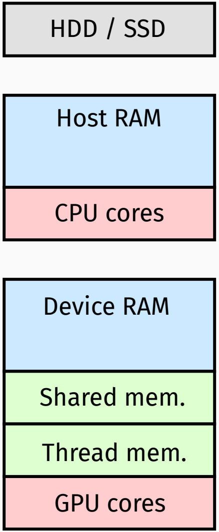
- **Large** arrays are **slow**: Memory read/write $\gg$ Arithmetics.
- **Fast** buffers are **small**: 1 KB $\simeq$ 100 float numbers per core.

To optimize the **Shared** and **Thread** memories: **C++ or Assembly.**

Most **scientists** rely on **pre-existing libraries** of CUDA kernels
and **never dig deeper than the GPU Device RAM.**

A practical example: nearest neighbor search

```python
import torch
x = torch.rand(M, D)                    # (M, D)
y = torch.rand(N, D)                    # (N, D)



diff = x.view(M,1,D) - y.view(1,N,D)    # (M, N, D)
diff2 = diff ** 2                       # (M, N, D)
sqdists = diff2.sum(dim=2)              # (M, N)
indices = sqdists.argmin(dim=1)         # (M,)
```



Bottleneck:

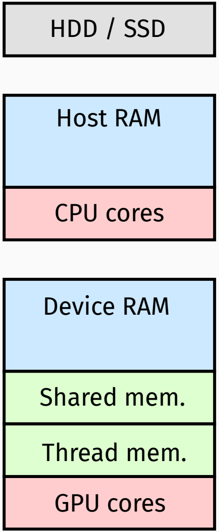
$(M \times N \times D)$ **CPU** operations and memory transfers.

A practical example: nearest neighbor search

```python
import torch
x_ = torch.rand(M, D)                     # (M, D)
y_ = torch.rand(N, D)                     # (N, D)
x = x_.cuda()                             # (M, D)
x = y_.cuda()                             # (N, D)

diff = x.view(M,1,D) - y.view(1,N,D)      # (M, N, D)
diff2 = diff ** 2                         # (M, N, D)
sqdists = diff2.sum(dim=2)                # (M, N)
indices = sqdists.argmin(dim=1)           # (M,)
```

Bottleneck:

$(M \times N \times D)$ **Device↔Thread** memory transfers.



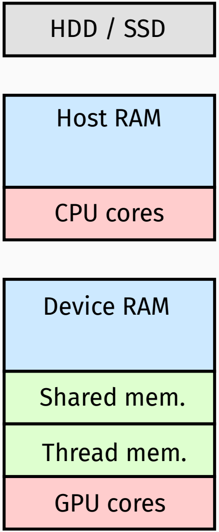
A practical example: nearest neighbor search

```python
import torch
x = torch.rand(M, D).cuda()              # (M, D)
y = torch.rand(N, D).cuda()              # (N, D)

# Use that |x-y|^2 = |x|^2 - 2 (x.y) + |y|^2:
dots = x @ y.T                           # (M, N)
sq_y = (y ** 2).sum(dim=1)               # (N,)

sqdists = - 2 * dots + sq_y.view(1,N)   # (M, N)
indices = sqdists.argmin(dim=1)          # (M,)
```

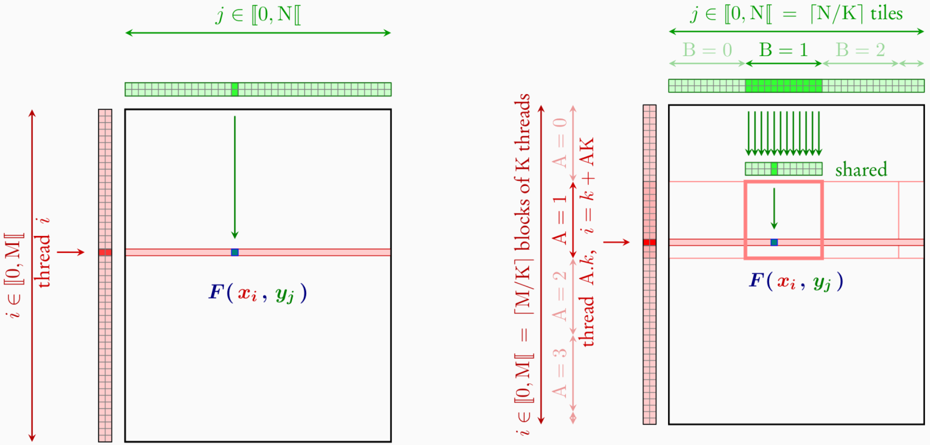
Bottleneck:

$(\mathrm{M} \times \mathrm{N} \times \mathrm{D})$ **GPU** computations if $\mathrm{D} > 100$,

$(\mathrm{M} \times \mathrm{N})$ **Device↔Thread** memory transfers otherwise.



A practical example: nearest neighbor search



On-the-fly, tiled reduction: **optimal memory management**.

Bottleneck: $(\mathrm{M} \times \mathrm{N} \times \mathrm{D})$ GPU computations.

Recap on nearest neighbor search

$$\forall\, i \in [1, \mathrm{M}]\,,\;\; \text{index}[\,i\,] \leftarrow \; \arg\min_{j=1}^{\mathrm{N}} \sum_{k=1}^{\mathrm{D}} (x[\,i,\,k\,] - y[\,j,\,k\,])^2$$

- Each **improvement** provides a $\times 10$ to $\times 100$ speed-up.
- Going even further, for **structured** data:
  - **Clusterize** the two point clouds.
  - **Sort** them to ensure that the clusters are **contiguous** in memory.
  - **Skip** whole **blocks** of the tiled distance matrix.
- **Standard benchmarks** (ann-benchmarks.com) and **libraries**: FAISS…

Compilation

Compilation is a major bottleneck in computer science

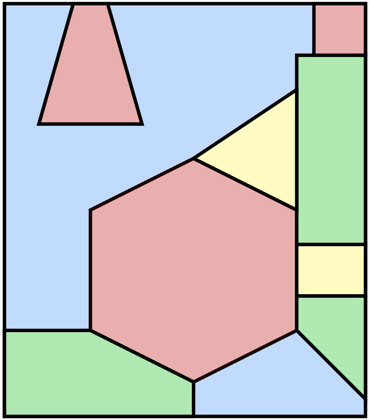
$$\forall\, i \in [\,1, \mathrm{M}\,],\;\; \text{index}[\,i\,] \leftarrow \operatorname*{arg\,min}_{j=1}^{\mathrm{N}} \sum_{k=1}^{\mathrm{D}} (x[\,i,\,k\,] - y[\,j,\,k\,])^2$$

- We have seen **4-5 different strategies**, increasingly fast but complex.
- Optimal schemes for $\mathrm{M} < 1{,}000$ look **completely different**.

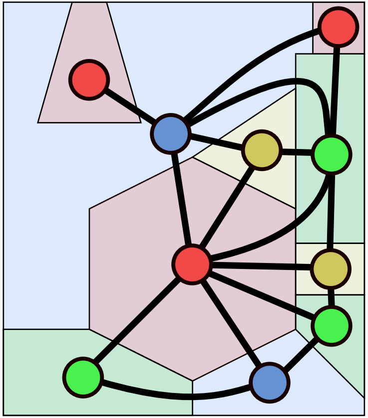
Naive GPU implementations are often **x100–x1,000 too slow**.

Reaching optimal run times is **hard**.

Compilation is a deep scientific problem



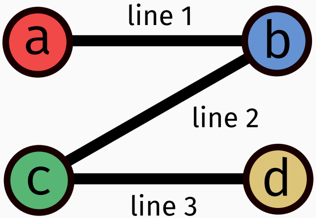
The 4 color theorem.



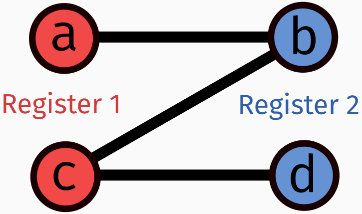
4-coloring a planar graph.

Register allocation via k-coloring of the interference graph

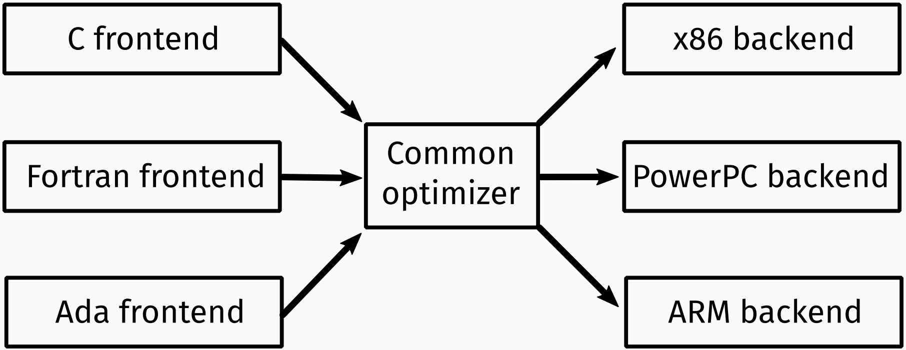
```
def f(a):
    b = a ** 2
    c = 5 * b
    d = c + 6
    return d
```




```
function(R1):
    R2 = R1 ** 2
    R1 = 5 * R2
    R2 = R1 + 6
    return R2
```



LLVM: a welcome consolidation [Lat11]



Just-in-time compilation

Dream: turn **high-level** Python code into an **optimal GPU binary**.

Reality: very hard **combinatorial** problem, **task-specific heuristics**.

Existing libraries focus on **different targets**:

- **Shaders** for 3D meshes.
- **Convolutions** on 2D and 3D grids – with varying filter sizes, channels…
- Fusion of **matrix multiplications** and **non-linearities** for MLPs, Transformers.

$\Longrightarrow$ A **critical mass** is required to attract investments.

What about **geometric ML?**

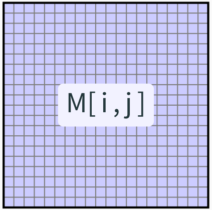
Computing libraries represent most objects as tensors

Context. Constrained **memory accesses** on the GPU:

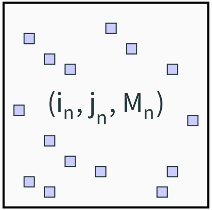
- **Long access times** to the registers penalize the use of large **dense** arrays.
- Hard-wired **contiguous** memory accesses penalize the use of **sparse** matrices.

Challenge. In order to reach optimal run times:

- **Restrict** ourselves to operations that are supported by the constructor: convolutions, FFT, etc.
- Develop new routines from scratch in C++/CUDA (FAISS, KPConv...): **several months of work**.



Dense array



Sparse matrix

The KeOps library: efficient support for symbolic matrices

Solution. KeOps – `www.kernel-operations.io`:

- For PyTorch, NumPy, Matlab and R, on **CPU and GPU**.
- **Automatic differentiation**.
- Just-in-time **compilation** of **optimized** C++ schemes, triggered for every new **reduction**: sum, min, etc.

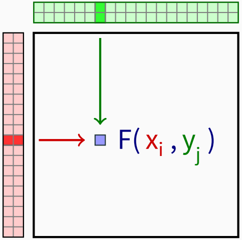
If the formula "F" is simple ($\leqslant$ 100 arithmetic operations):

"100k $\times$ 100k" computation $\to$ 10ms – 100ms,

"1M $\times$ 1M" computation $\to$ 1s – 10s.

Hardware ceiling of 10^{12} operations/s.

$\times$**10 to** $\times$**100 speed-up** vs standard GPU implementations for a wide range of problems.



Symbolic matric

Formula + data

- Distances $d(x_i,y_j)$.
- Kernel $k(x_i,y_j)$.
- Numerous transforms.

A first example: efficient nearest neighbor search in dimension 50

Create large point clouds using **standard PyTorch syntax**:

```
import torch
N, M, D = 10**6, 10**6, 50
x = torch.rand(N, 1, D).cuda()  # (1M,  1, 50) array
y = torch.rand(1, M, D).cuda()  # ( 1, 1M, 50) array
```

Turn **dense** arrays into **symbolic** matrices:

```
from pykeops.torch import LazyTensor
x_i, y_j = LazyTensor(x), LazyTensor(y)
```

Create a large **symbolic matrix** of squared distances:

```
D_ij = ((x_i - y_j) ** 2).sum(dim=2)  # (1M, 1M) symbolic
```

Use an `.argmin()` **reduction** to perform a nearest neighbor query:

```
indices_i = D_ij.argmin(dim=1)  # -> standard torch tensor
```

The KeOps library combines performance with flexibility

Script of the previous slide = efficient nearest neighbor query,
on par with the bruteforce CUDA scheme of the **FAISS** library…
And can be used with **any metric**!

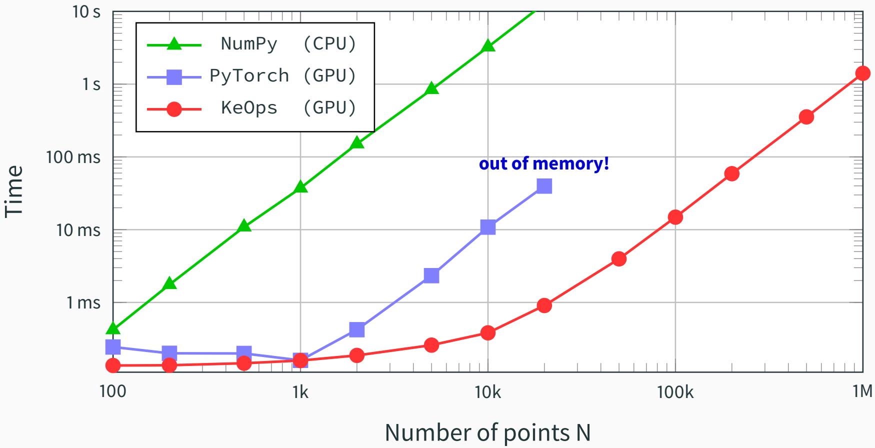
```
D_ij = ((x_i - x_j) ** 2).sum(dim=2)      # Euclidean
M_ij =  (x_i - x_j).abs().sum(dim=2)      # Manhattan
C_ij = 1 - (x_i | x_j)                    # Cosine
H_ij = D_ij / (x_i[...,0] * x_j[...,0])   # Hyperbolic
```

KeOps supports arbitrary **formulas** and **variables** with:

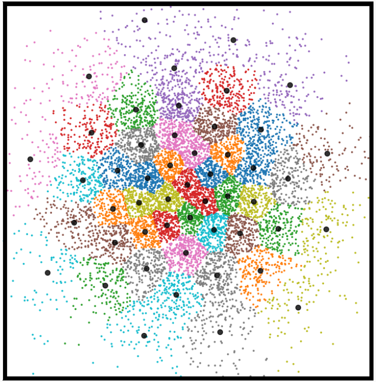
- **Reductions:** sum, log-sum-exp, K-min, matrix-vector product, etc.
- **Operations:** $+$, $\times$, sqrt, exp, neural networks, etc.
- **Advanced schemes:** batch processing, block sparsity, etc.
- **Automatic differentiation:** seamless integration with PyTorch.

KeOps lets users work with millions of points at a time

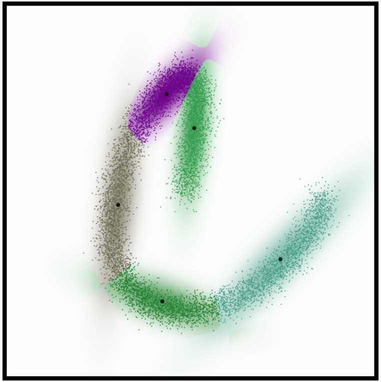
Benchmark of a Gaussian **convolution**
between **clouds of N 3D points** on a RTX 2080 Ti GPU.



KeOps is a good fit for machine learning research



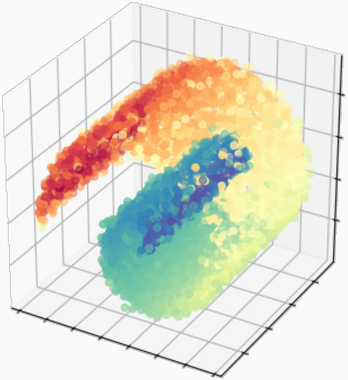
K-Means.



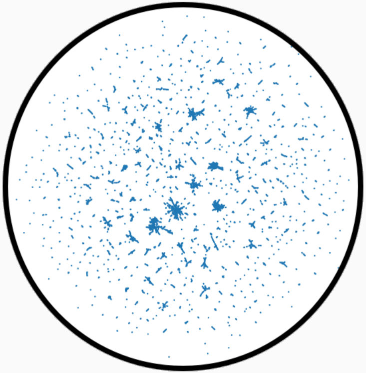
Gaussian Mixture Model.

Use **any** kernel, metric or formula **you** like!

KeOps is a good fit for machine learning research



Spectral analysis.

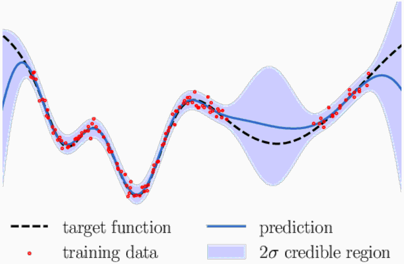


UMAP in hyperbolic space.

Use **any** kernel, metric or formula **you** like!

Applications to Kriging, spline, Gaussian process, kernel regression

A standard tool for regression [Lec18]:



Under the hood, solve a **kernel linear system**:

$$(\lambda\,\mathrm{Id} + K_{xx})\,a \;=\; b \qquad \text{i.e.} \qquad a \;\leftarrow\; (\lambda\,\mathrm{Id} + K_{xx})^{-1} b$$

where $\lambda \geqslant 0$ et $(K_{xx})_{i,j} = k(x_i, x_j)$ is a positive definite matrix.

Applications to Kriging, spline, Gaussian process, kernel regression

KeOps symbolic tensors $(K_{xx})_{i,j} = k(x_i, x_j)$ **:**

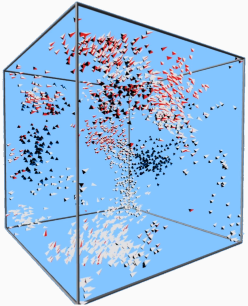
- Can be fed to **standard solvers**: SciPy, GPyTorch, etc.
- GPytorch on the 3DRoad dataset (N = 278k, D = 3):

  7h with 8 GPUs $\longrightarrow$ **15mn with 1 GPU.**

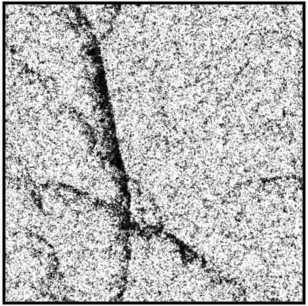
- Provide a **fast backend for research codes**:
  see e.g. *Kernel methods through the roof: handling **billions of points** efficiently*,
  by G. Meanti, L. Carratino, L. Rosasco, A. Rudi (2020).

KeOps lets researchers focus on their models, results and theorems

Some applications to **dynamical systems** [DM08, DFMAT17]
and **statistics** [CDF19] with A. Diez, G. Clarté et P. Degond:



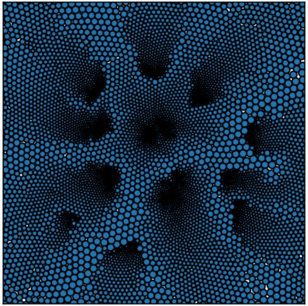
3D Vicsek model with orientation,
interactive demo with 2k **flyers**.



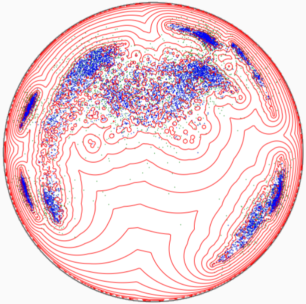
2D Vicsek model on the torus,
in real-time with 100k **swimmers**.

KeOps lets researchers focus on their models, results and theorems

$\Longrightarrow$ Scale up to **millions/billions** of agents with Python scripts.

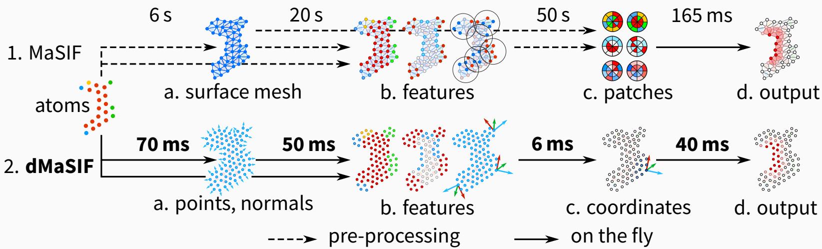


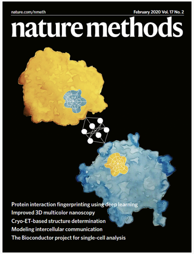
Packing problem in 2D
with 10k repulsive balls.



Collective Monte Carlo **sampling**
on the hyperbolic Poincaré disk.

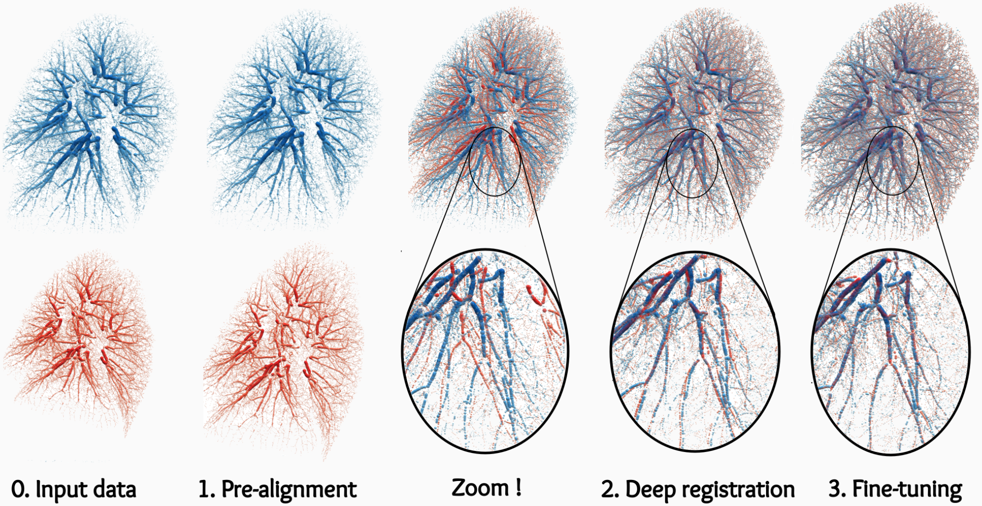
Scaling up geometric deep learning [GSM+20, SFCB20, SFS+22]





$\longrightarrow$ $\times 100$ - $\times 1{,}000$ **faster**, **lighter** and fully differentiable.

Scaling up geometric deep learning and optimal transport [SFL+21]



Recap on compilation

- Turning scientific code into optimal binaries is **an open problem**:

$\longrightarrow$ Massive **room for improvement** on the software side.

$\longrightarrow$ **Valuable and impactful skill.**

- **Symbolic** matrices are to **geometric** ML what **sparse** matrices are to **graph** processing:

$\longrightarrow$ KeOps: **x30 speed-up** vs. PyTorch, TF et JAX.

$\longrightarrow$ Useful in a wide range of settings.

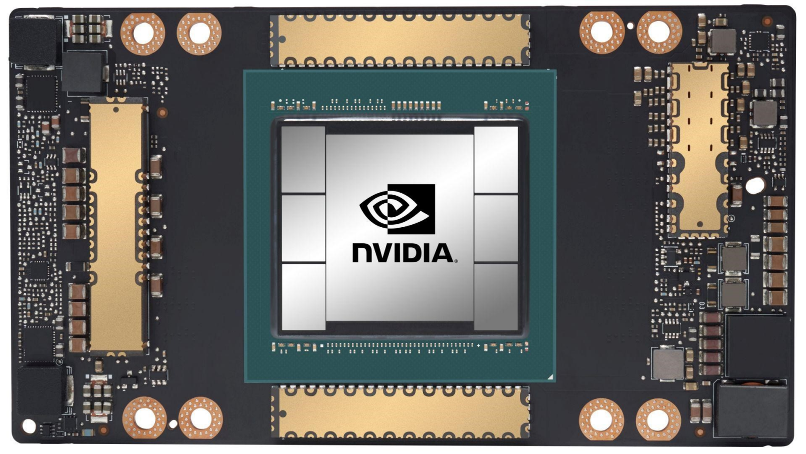
- These tools open **new paths** for geometers and statisticians:

$\longrightarrow$ GPUs are more **versatile** than you think.

$\longrightarrow$ Ongoing work to provide **fast GPU backends** to researchers, going beyond what Google and Facebook are ready to pay for.

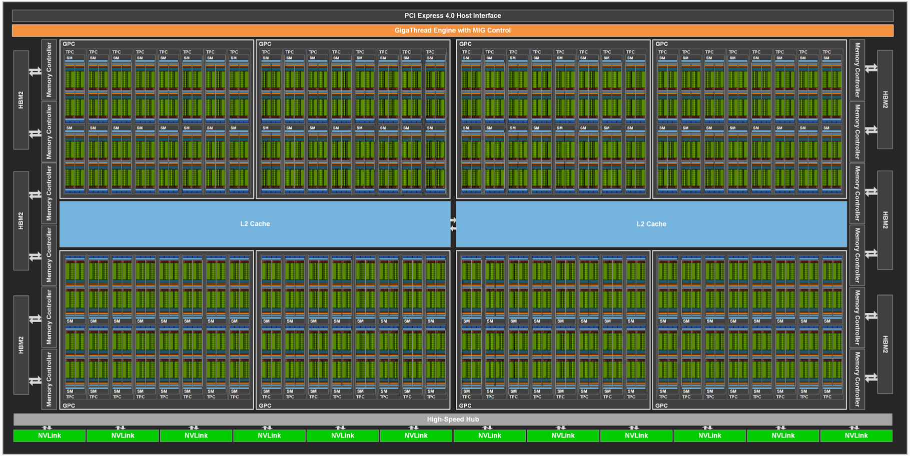
Optimized AI cores

Nvidia Ampere architecture in-depth [KGJ+20]



NVIDIA **A100 GPU** – the flagship AI chip as of 2020-22.

Nvidia Ampere architecture in-depth [KGJ+20]



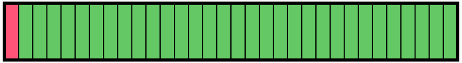
GA100 architecture with all 128 blocks. **A100 GPU = 108 functional blocks.**

Nvidia Ampere architecture in-depth [KGJ+20]

"Physical" **CUDA block** or Streaming Multiprocessor:

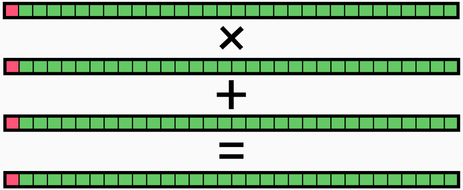
- 192 KB of **Shared** memory.
- **4 squads** of "physical threads" or warps with:
    - 64 KB of **Thread** memory.
    - **16 int-32** cores.
    - **16 float-32** cores.
    - **8 float-64** cores.
    - **1 Tensor core.**

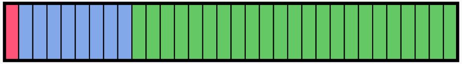
Integer cores: handle memory addresses – Float-32 cores: great for 3D geometry



1 sign bit 31 significand bits

$1\text{-}2^{-31}$ — $+2^{+31}$ $\simeq \pm 2{,}147{,}483{,}647$

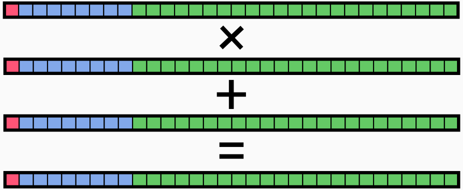




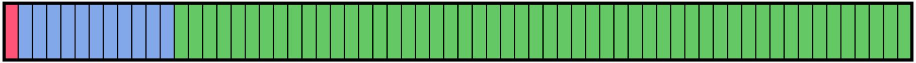
1 sign 8 exponent 23 significand

$2^{-126} - 2^{+127} \simeq 10^{-38} - 10^{+38}$

$1 + 2^{-23} \simeq 1\,.\,000\;000\;1$



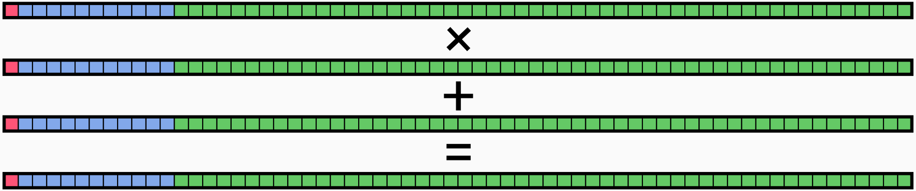
Float-64 cores: great for physics simulation



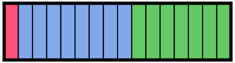
1 sign 11 exponent 52 significand

$$2^{-1022} - 2^{+1023} \simeq 10^{-308} - 10^{+308}$$

$$1 + 2^{-52} \simeq 1.000\,000\,000\,000\,000\,2$$



Tensor cores: great for CNNs and transformers

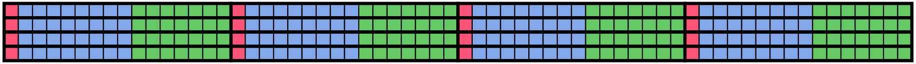


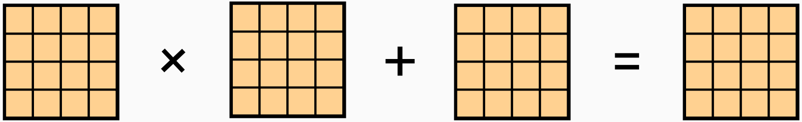
1 sign 8 exponent 7 significand

$2^{-126} - 2^{+127} \simeq 10^{-38} - 10^{+38}$

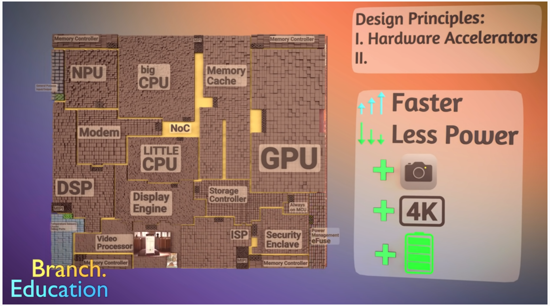
$1 + 2^{-7} \simeq 1.007$

256 bits $\simeq$ 4x4 bfloat-16

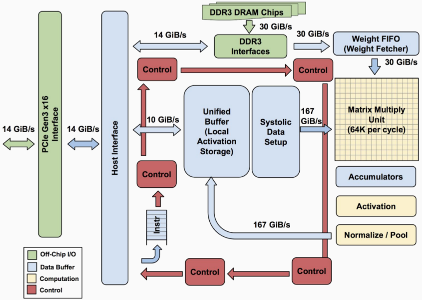




Trading speed vs. power consumption vs. versatiliy vs. manufacturing costs

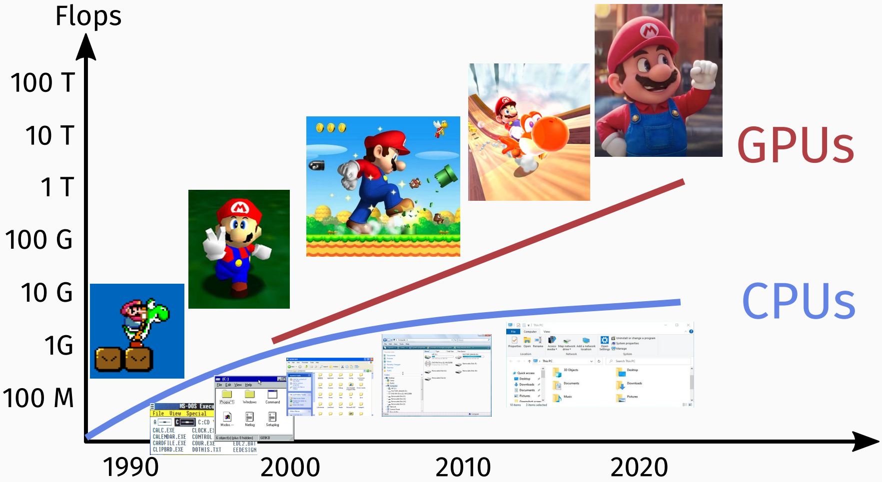


“How do **Smartphone CPUs** Work?”
by Branch Education.

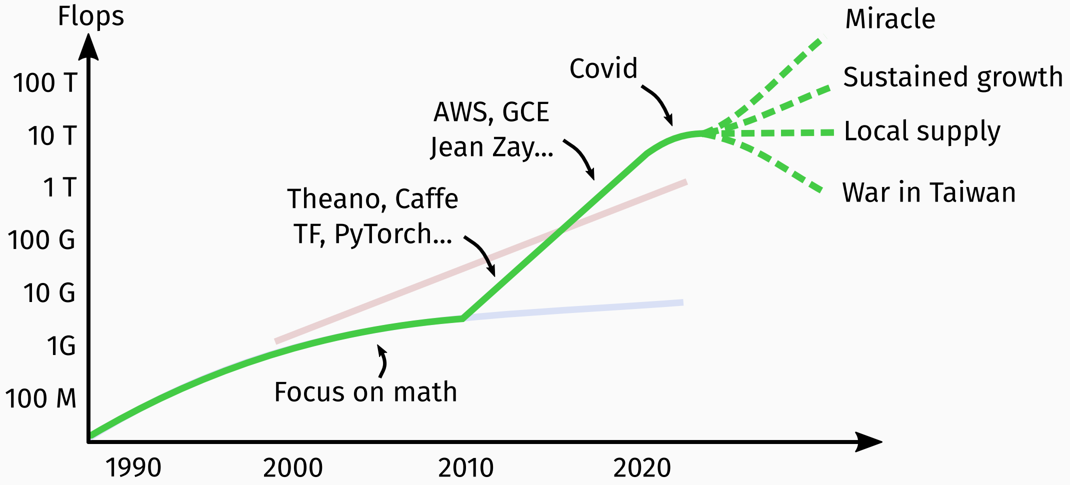


Tensor Processing Units,
by Google.

The CPU vs. GPU uncoupling occured in the early 2000’s

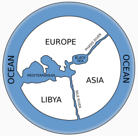


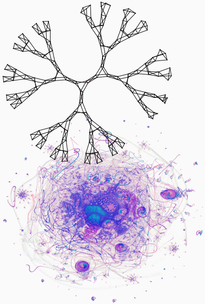
Computing power available to ML researchers

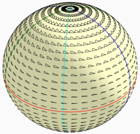


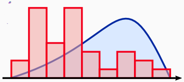
Conclusion

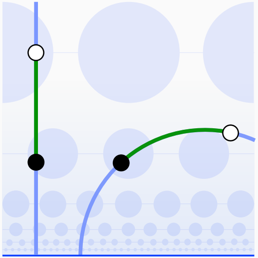
A geometric tour of data science

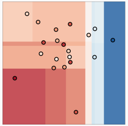


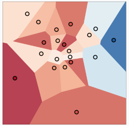


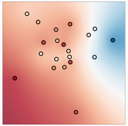


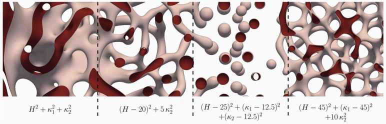


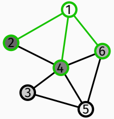


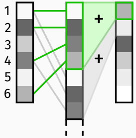












What is AI research about?

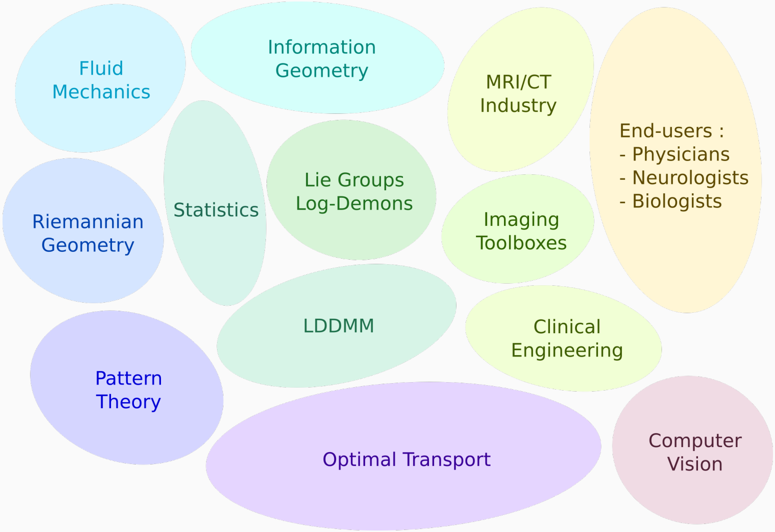
Outsider's view:
enthusiast.

Insider's view:
professional.

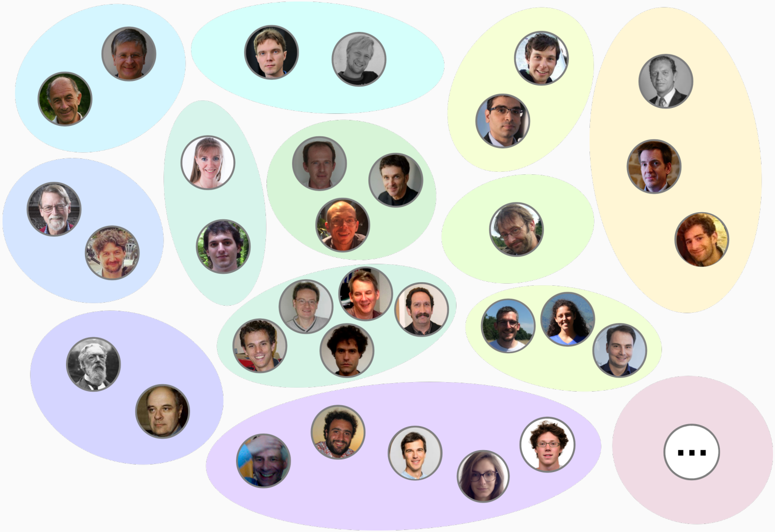
Tunnel vision on a single angle $\Longrightarrow$ **high risk** career.

Biggest success of the 1848 **gold rush**: Levi's **blue jeans**.

ML research is 100% interdisciplinary – a mind map of my own PhD experience



Research is a deeply social and diverse activity



Some early career advice

1. **You** bring more to the table than your **potential advisor**:
   - **Full-time** focus on a subject $=$ only during your PhD.
   - Your **leverage**: show that you are skilled and **reliable**.

2. **Tutoring time** + **open** research area $\gg$ Prestige:
   - Avoid **crowded** teams and topics.
   - Outstanding environments **outside** of Paris/London/Boston/SF...
   - Connect in conferences and **workshops**.

Some early career advice

3. Different **countries**, different **people**, different **perspectives**.
Who is the **"main character"** of a PhD thesis?

- I believe that it should be the **student**.
- Some people think that it is the **advisor**.

4. **Personal chemistry** + **general** research area $\gg$ Precise topic:

- A PhD that goes according to plan is a bit disappointing anyway ;-)
- **Meet** team members (including **students**!) before signing a long-term contract.
- **Internship** $\simeq$ trial period, goes both ways.

Befriend domain experts – Find your own balance

References

References i

Encyclopædia Britannica.

Ideal gas.

https://www.britannica.com/science/ideal-gas.

Grégoire Clarté, Antoine Diez, and Jean Feydy.

Collective proposal distributions for nonlinear MCMC samplers: Mean-field theory and fast implementation.

arXiv preprint arXiv:1909.08988, 2019.

Datumizer.

Solar system orrery inner planets.

https://commons.wikimedia.org/wiki/File:Solar_system_orrery_inner_planets.gif, 2018.

CC BY-SA 4.0.

Pierre Degond, Amic Frouvelle, Sara Merino-Aceituno, and Ariane Trescases.

Alignment of self-propelled rigid bodies: from particle systems to macroscopic equations.

In *International workshop on Stochastic Dynamics out of Equilibrium*, pages 28–66. Springer, 2017.

Pierre Degond and Sébastien Motsch.

Continuum limit of self-driven particles with orientation interaction.

Mathematical Models and Methods in Applied Sciences, 18(supp01):1193–1215, 2008.

Erich Dornberger.

***Prediction of OSF ring dynamics and grown-in voids in Czochralski silicon crystals*.**

PhD thesis, UCL-Université Catholique de Louvain, 1997.

Olivier Ecabert, Jochen Peters, and Matthew Walker.

Segmentation of the heart and great vessels in ct images using a model-based adaptation framework.

Medical Image Analysis, (15):863–876, 2011.

Anna Frodesiak.

Traffic jam at 17:30 downtown haikou city, hainan province, china.

https://commons.wikimedia.org/wiki/File:Traffic_jam_in_Haikou,_Hainan,_China_01.jpg, 2012.

Public domain.

Pablo Gainza, Freyr Sverrisson, Frederico Monti, Emanuele Rodola, D Boscaini, MM Bronstein, and BE Correia.

Deciphering interaction fingerprints from protein molecular surfaces using geometric deep learning.

Nature Methods, 17(2):184–192, 2020.

Henrik.

This diagram illustrates the ray tracing algorithm for rendering an image.

https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg, 2008.

CC BY-SA 4.0.

Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy.

Nvidia ampere architecture in-depth.

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/, 2020.

Chris Lattner.

The architecture of open source applications – llvm.

https://www.aosabook.org/en/llvm.html, 2011.

Florent Leclercq.

Bayesian optimization for likelihood-free cosmological inference.

Physical Review D, 98(6):063511, 2018.

Mohammad Sina Nabizadeh, Stephanie Wang, Ravi Ramamoorthi, and Albert Chern.

Covector fluids.

ACM Transactions on Graphics (TOG), 41(4):113:1–113:15, 2022.

Peellden.

A 12-inch silicon wafer.

https://commons.wikimedia.org/wiki/File:12-inch_silicon_wafer.jpg, 2011.

CC BY-SA 3.0.

Gabriel Peyré.

The numerical tours of signal processing-advanced computational signal and image processing.

IEEE Computing in Science and Engineering, 13(4):94–97, 2011.

Freyr Sverrisson, Jean Feydy, Bruno E. Correia, and Michael M. Bronstein.

Fast end-to-end learning on protein surfaces.

bioRxiv, 2020.

Zhengyang Shen, Jean Feydy, Peirong Liu, Ariel H Curiale, Ruben San Jose Estepar, Raul San Jose Estepar, and Marc Niethammer.

Accurate point cloud registration with robust optimal transport.

Advances in Neural Information Processing Systems, 34:5373–5389, 2021.

Freyr Sverrisson, Jean Feydy, Joshua Southern, Michael M Bronstein, and Bruno Correia.

Physics-informed deep neural network for rigid-body protein docking.

In *ICLR2022 Machine Learning for Drug Discovery*, 2022.

Peter Shirley.

Ray tracing in one weekend, December 2020.

`https://raytracing.github.io/books/RayTracingInOneWeekend.html`.