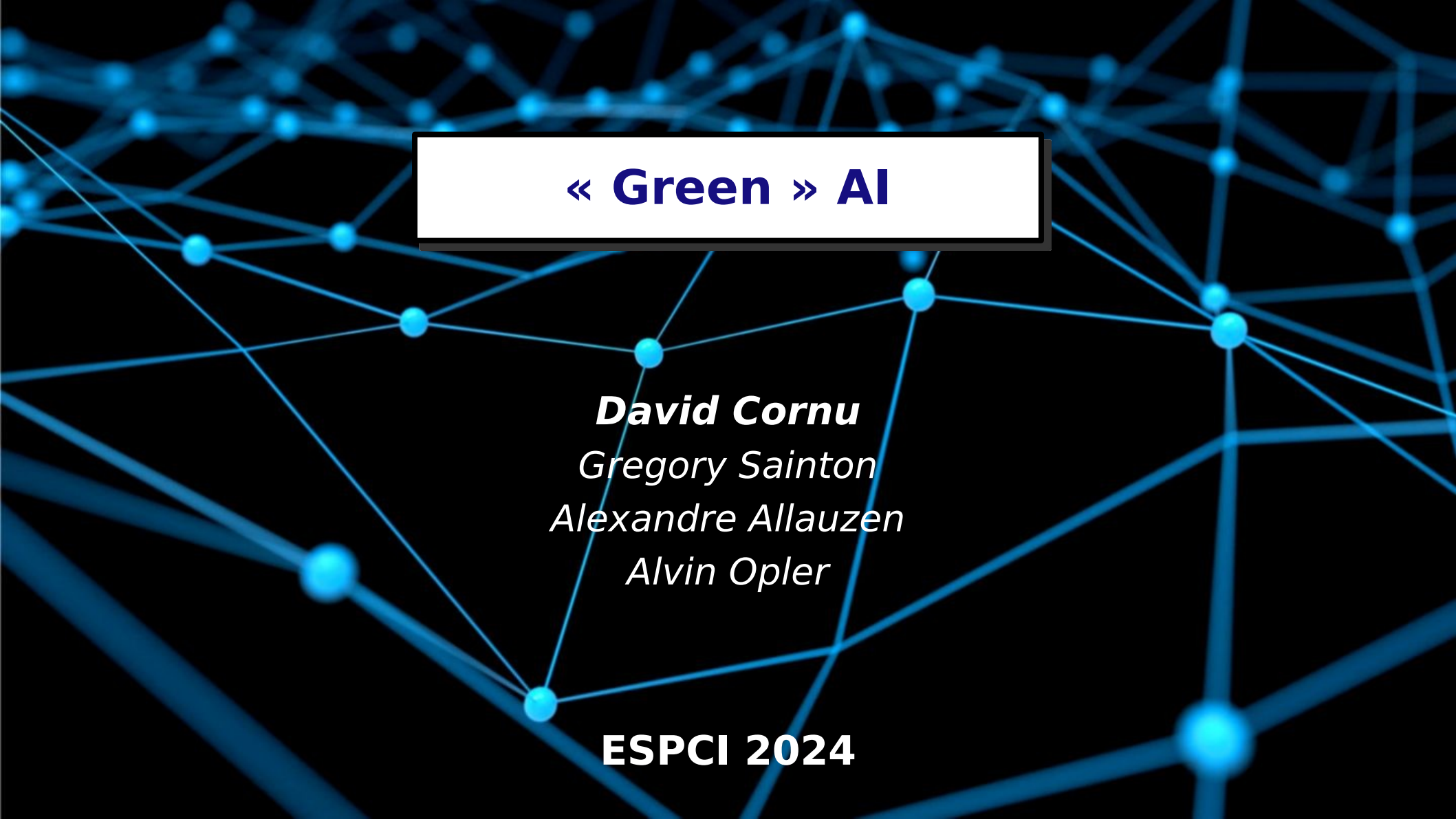


« Green » AI

David Cornu
Gregory Sainton
Alexandre Allauzen
Alvin Opler

ESPCI 2024

Lessons materials

Slides, exercises, codes, corrections and datasets are **available on GitHub** and will be updated regularly:

https://github.com/Deyht/green_ai_espci

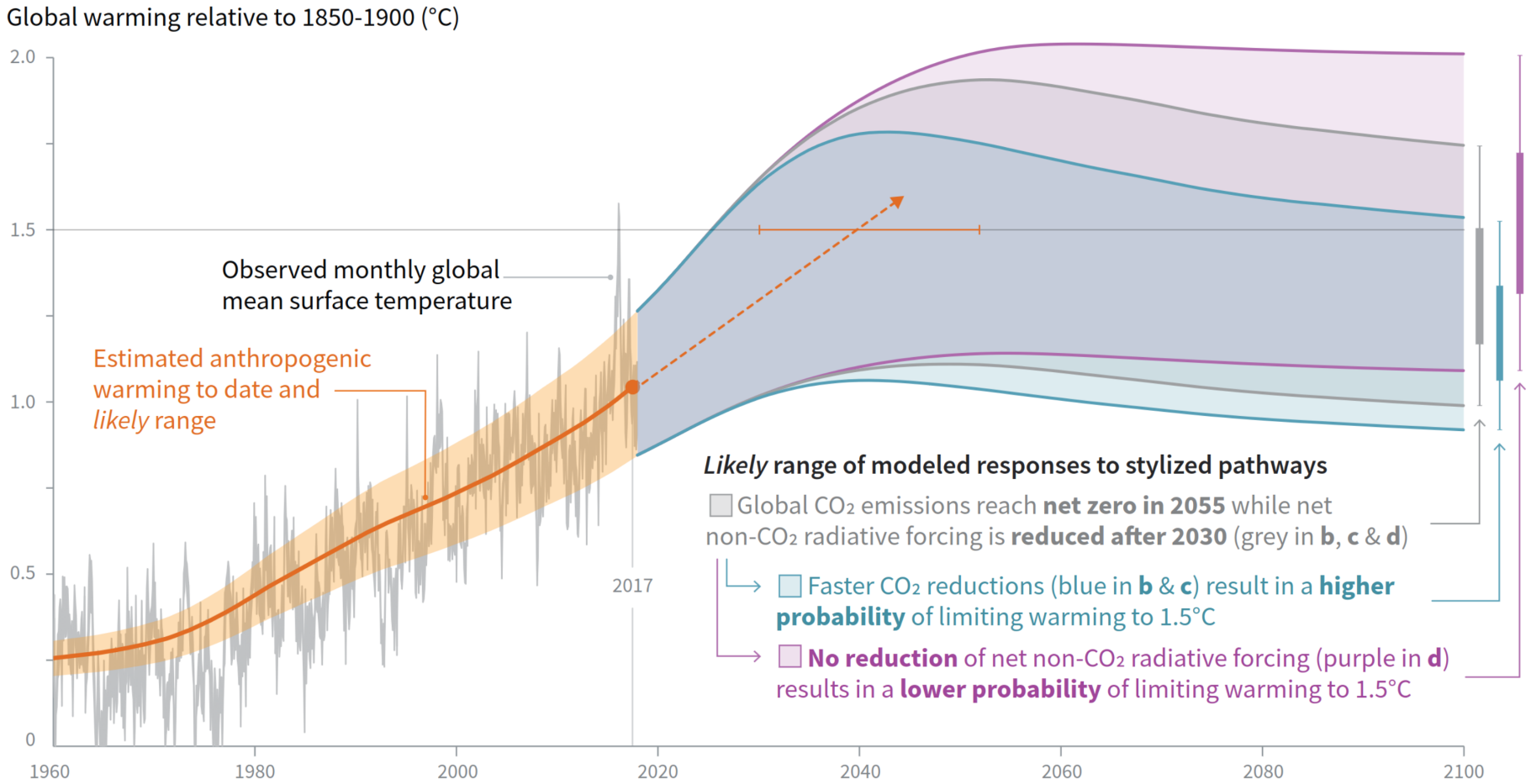
git clone https://github.com/Deyht/green_ai_espci
git pull

Or download the repository in zip file

Avoid losing your work on forced pull updates by copying all files from the cloned repository into a working directory!

Do not copy and past content from git-hub pages (lead to format errors).
Use python up to 3.10 but not more recent.

Global context



The Paris agreement require that we reduce the global CO2-e emission by 8% each year

From GIEC SPM

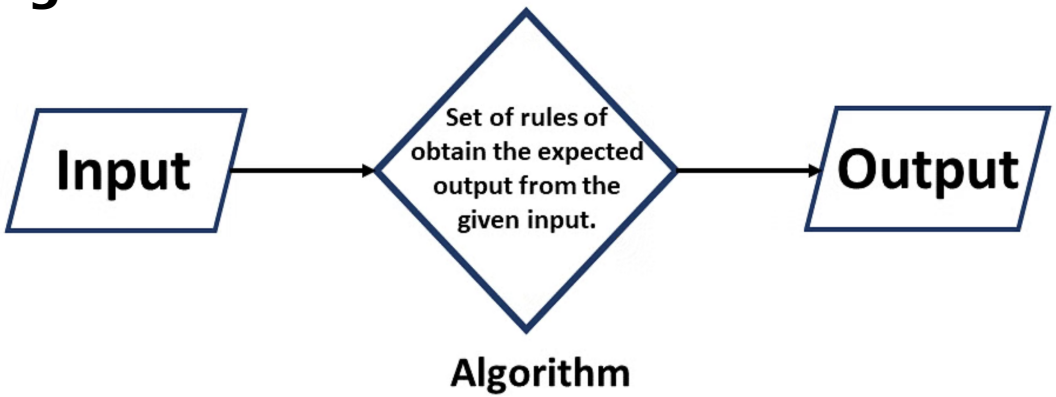
What is AI / ML and how does it impact the environment ?

AI is a computer program that provide a solution to a problem given some data.

Mostly like any other computer program!

They even use the same type of numerical infrastructures.

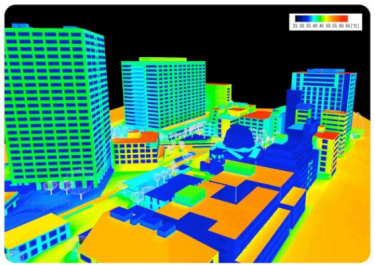
The specificity of AI models is that they learn the inner set of rules automatically through statistical learning.



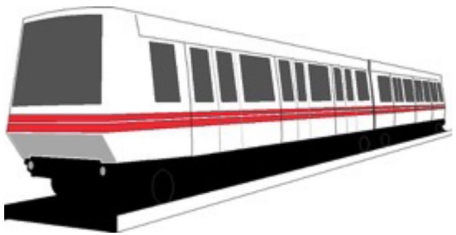
So the first question is then:

How does numerical infrastructures and technologies impact the environment ?

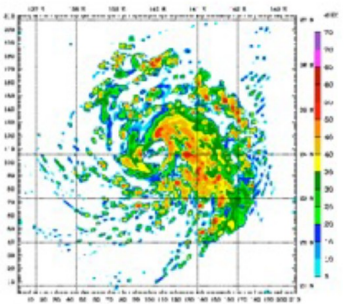
The many forms of the digital world











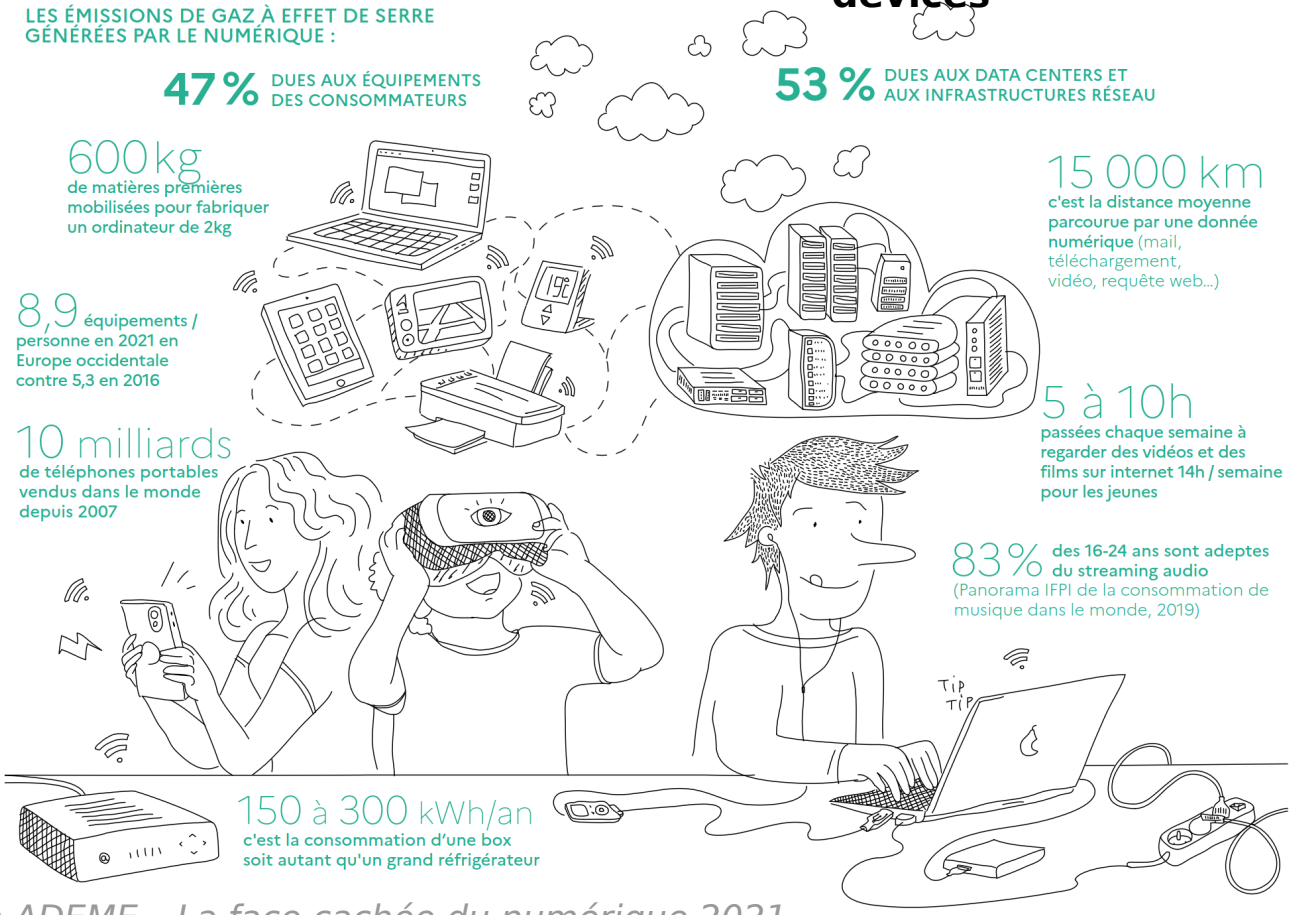
Often refer to them as **ICT = Information and communications technology**
ICT are now a part of all activity fields, so is their environmental impact

From Anne-Cecile Orgerie talk, « L'informatique verte moins branchée »

A few interesting numbers

ICT are responsible for 4 to 8% of global CO2-e emissions

=> Might seems low, but this sector is **growing fast**, especially with the development of **AI and IoT devices**



INTERNET AU NIVEAU MONDIAL

- 67 millions de serveurs
- 1,1 milliard d'équipements réseaux (routeurs, box ADSL...)
- 19 milliards d'objets connectés en 2019
- 48 milliards en 2025 selon les estimations

En 1 heure

- 8 à 10 milliards de mails échangés (hors spam)
- 180 millions de recherches Google

From ADEME – La face cachée du numérique 2021

Distribution of ICT energy consumption

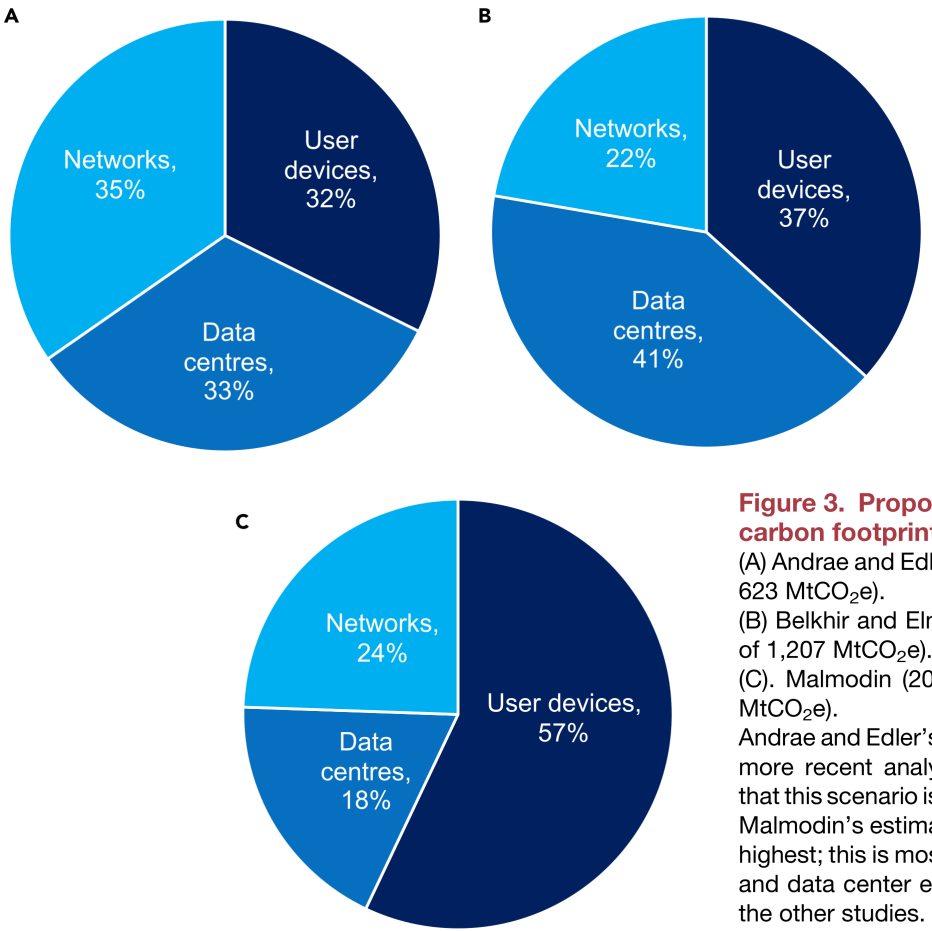


Figure 3. Proportional breakdown of ICT's carbon footprint, excluding TV
(A) Andrae and Edler (2015): 2020 best case (total of 623 $MtCO_2e$).
(B) Belkhir and Elmeligi (2018): 2020 average (total of 1,207 $MtCO_2e$).
(C). Malmodin (2020): 2020 estimate (total of 690 $MtCO_2e$).
Andrae and Edler's[3] best case is displayed because more recent analysis by the lead author suggest that this scenario is most realistic for 2020. Note that Malmodin's estimate of the share of user devices is highest; this is mostly because Malmodin's network and data center estimates are lower than those of the other studies.

ICT are usually split in three categories:

- **User devices** (your smartphone or laptop)
- **Network infrastructures**, allowing to exchange information and data
- **Data centers** that centralize the relevant data and services

All the three categories are growing fast !

Estimating the respective contributions of these parts in terms of CO2-e is difficult and depends on models with many assumptions regarding:

- The lifespan of the equipments

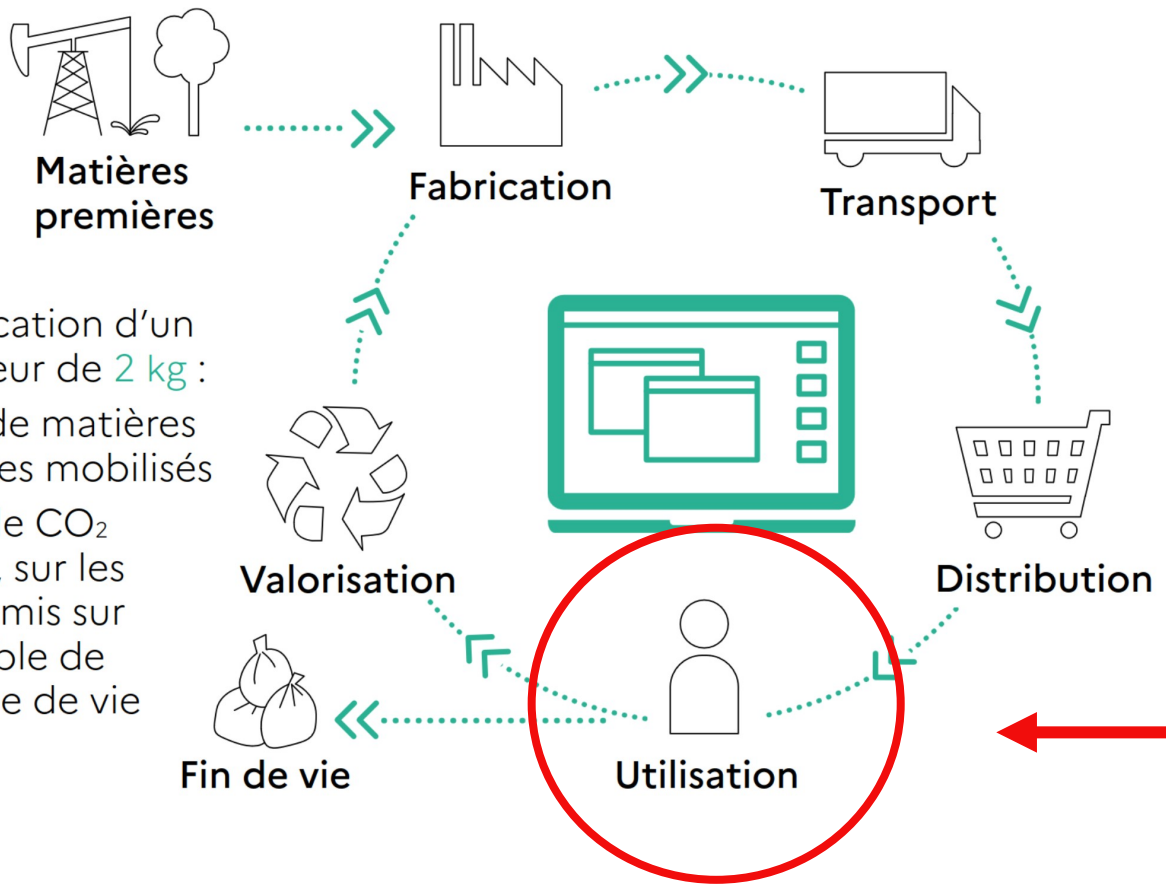
- The local electrical mix

- How and where the equipments were built and the origin of the row materials

- The exact field to which is associated a numerical activity (eg. autonomous vehicle computer count in ICT or in automobile contribution to CO2-e?)

Life cycle of an ICT

There is an environmental impact at every step of the life cycle



La fabrication d'un ordinateur de 2 kg :
600 kg de matières premières mobilisés
103 kg de CO_2 générés, sur les 156 kg émis sur l'ensemble de son cycle de vie

Most of the time, increasing the lifespan of an ICT allow to strongly reduce the CO2-e emission in comparison to replacing it.

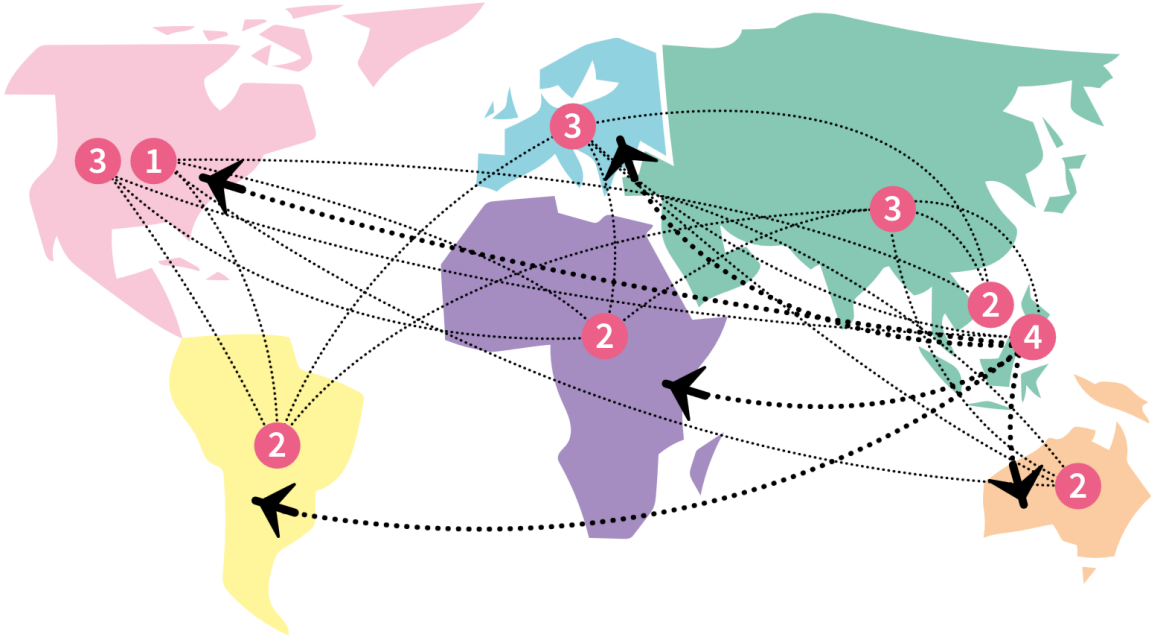
We note that planned obsolescence is responsible of many unnecessary renewal of devices.

Also, end of life can consume a lot of energy or have strong environmental impacts other than CO2-e!

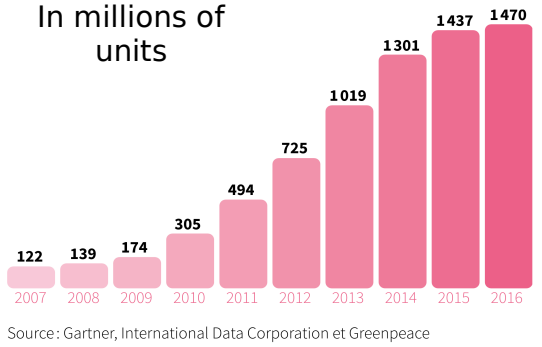
Usually have the less impact, but get most of the attention !

From ADEME – La face cachée du numérique 2021

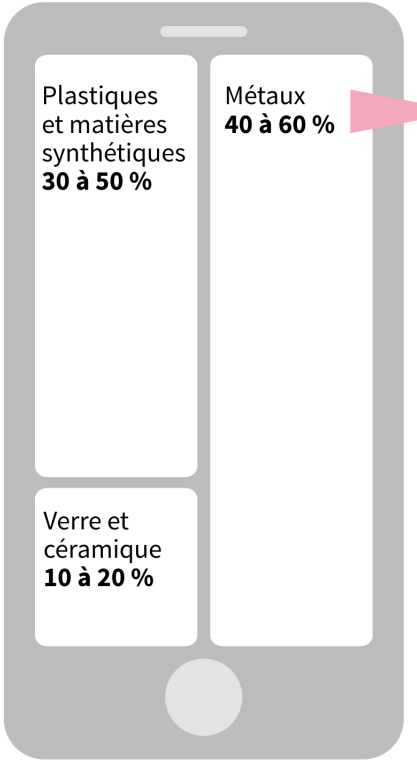
Example case of a smartphone assembly



1 – Conception
2 – Raw materials extraction
3 – Main component manufacturing
4 – Final assembly
✈ – Worldwide distribution



Source : Gartner, International Data Corporation et Greenpeace



PROPORTION DES MÉTAUX

80 à 85 % de métaux ferreux et non ferreux : cuivre, aluminium, zinc, étain, chrome, nickel…

0,5 % de métaux précieux : or, argent, platine, palladium…

0,1 % de terres rares et métaux spéciaux : europium, yttrium, terbium, gallium, tungstène, indium, tantale…

15 à 20 % d'autres substances : magnésium, carbone, cobalt, lithium…

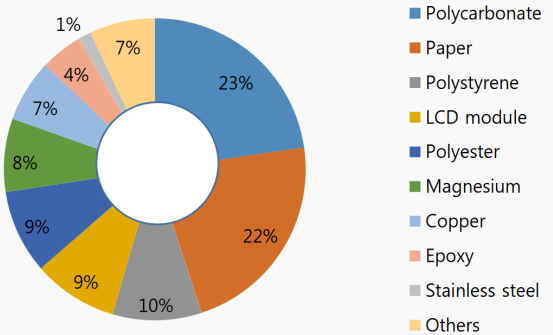
From ADEME – Les impacts du smartphone

The limits of the CO2-e measurement
Other types of impacts on the environment

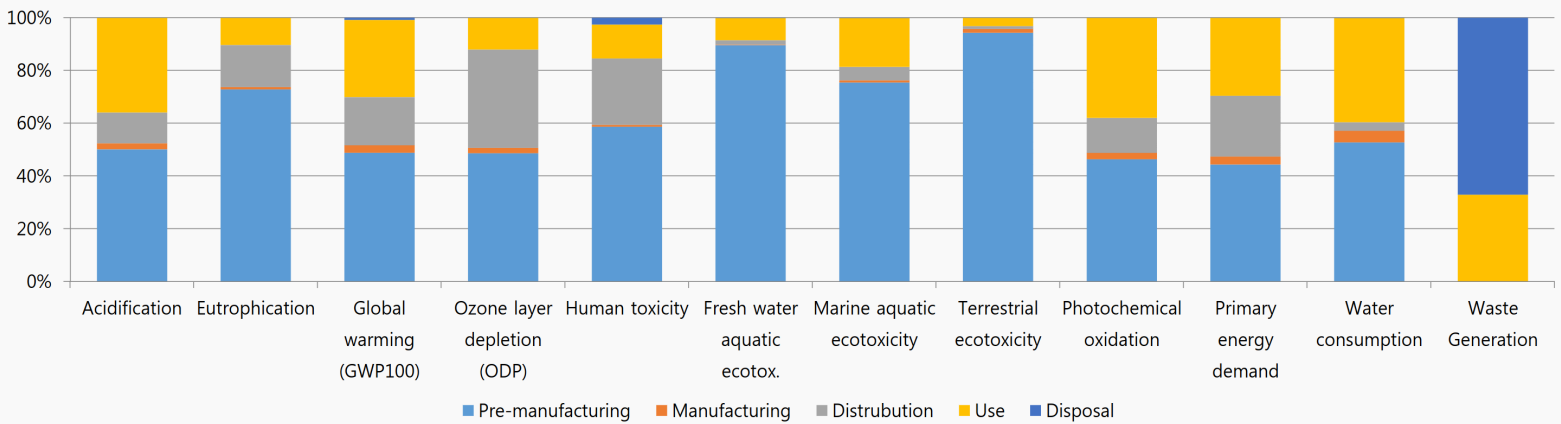
Product Features

Model name	SM-W727V (Galaxy Book)
Processor	Intel, Core i5, 3.1GHz Dual-Core 64bit
Dimension	199.8 * 291.3 * 7.4(H*W*D)
Display	AMOLED, OCTA, SDC, 2160 x 1440 (FHD+) 12.0", 303.7mm 16M
Battery	Li-Ion 5070 mAh
Camera	13 MP / 5MP
Wt.(g)	1881.9g

Material Use

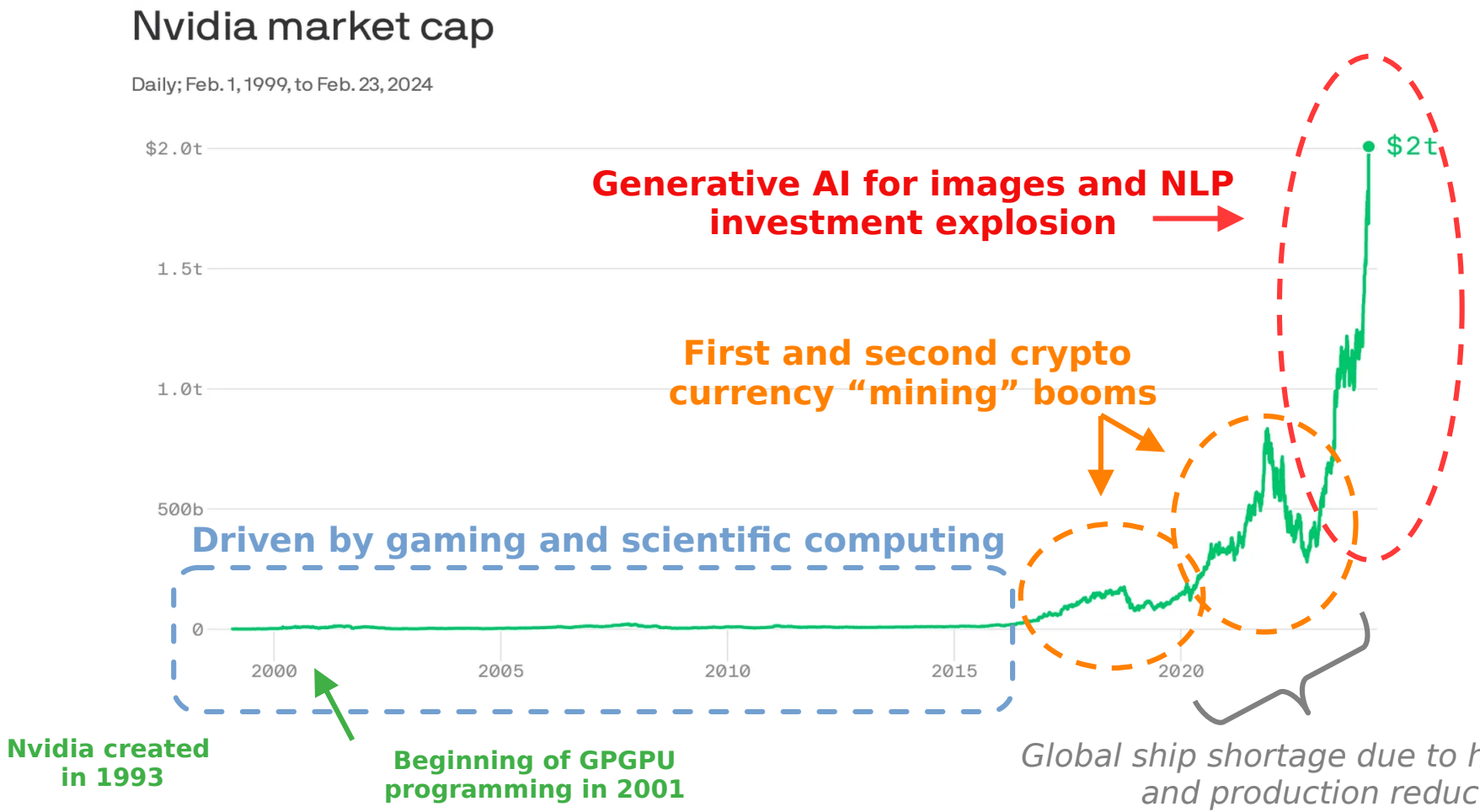


Characterized Environment Impact



From Samsung Life Cycle Assessement for Mobile Products, 2018

What about AI dedicated hardware (GPU)?



Note that most of the principles of modern ML/AI exist since 1980!

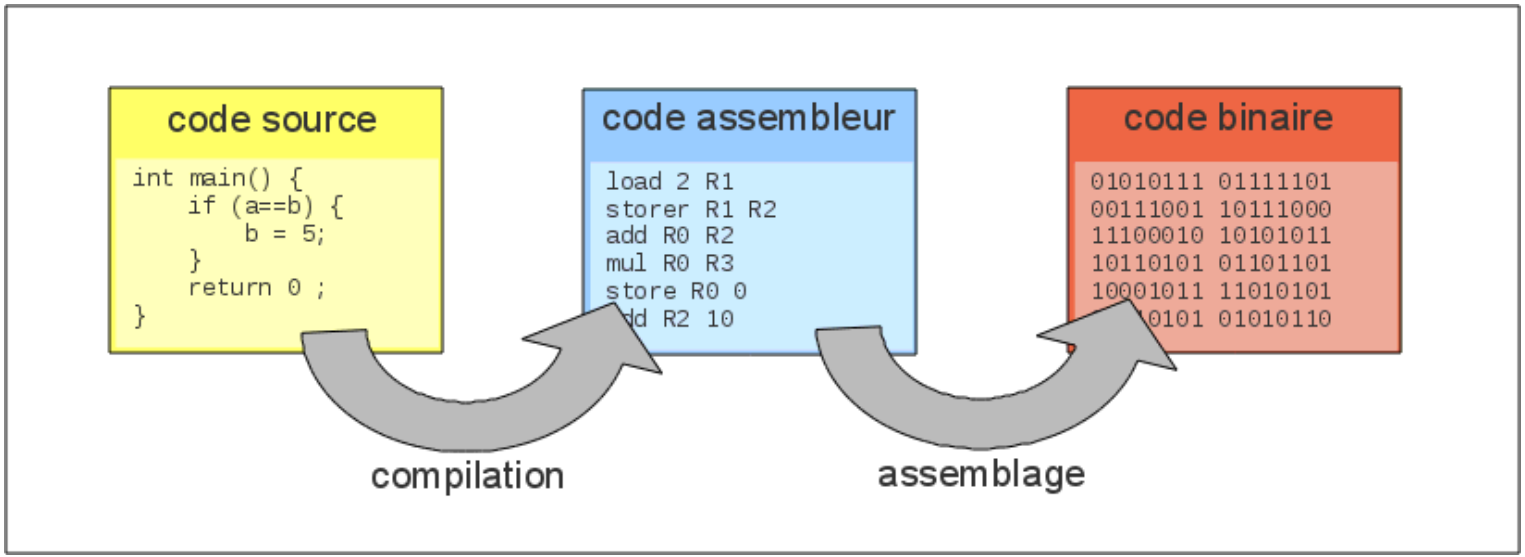
Context elements we haven’t talked about

- **Can a numerical transition accompany an environmental transition?**
  Use of numerical algorithms or devices to optimize other transitions. Is more ICT a solution?
- **The evolution of the data rate for various application is exploding and current intercontinental connections are facing strong limits**
- **Geopolitics → No country is autonomous in producing ICT. Network traffic is worldwide and require continuity of physical infrastructures.**
- **We have almost not talked about ICT end of life → recycling? E-waste ?**
- **Usage regulations? For which applications ?**

Many resources are accessible through the EcoInfo CNRS GDR. You can join the group to have interactions with an active community of researchers that try to better estimate and propose solution to the environmental impact of ICT.

Re-centering to our objective: Computation efficiency

First what is a computer program ?



1) High level code, close to natural language

3) Binary machine code that can run on a CPU

2) Low level code, series of basic instructions

What are the mains components of all computers ?

CPU

Do the computation

Storage

Keep the static data

RAM

Store the volatile data

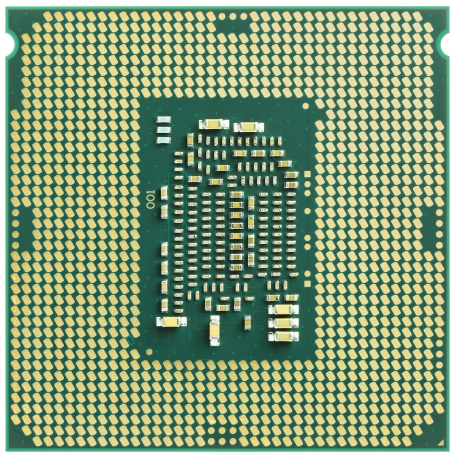
Possible other parts and peripherals include:

- Screen
- GPU
- External storage readers
- Internal power supply
- Cooling solution
- etc .

Motherboard

Link all the other elements and peripherals

The heart of the computer: the Central Processing Unit (CPU)



A CPU is an electronic circuit capable of executing instructions for a program, such as arithmetic, logic or I/O operations.

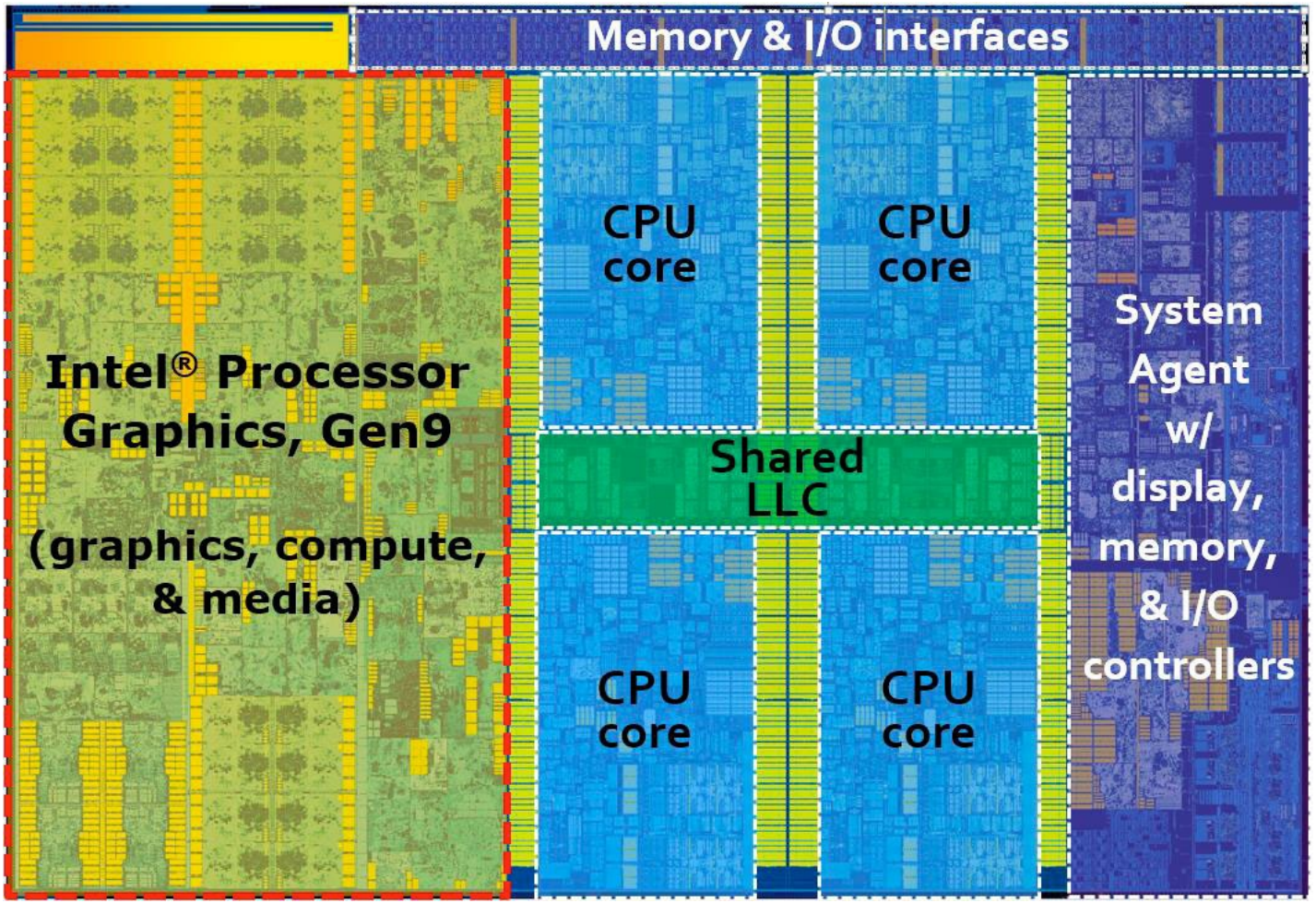
Modern CPU are implemented on integrated circuit and combined with cache memory and peripheral interfaces.

A CPU is an assembly of billions of transistors that are combined to form analogical operations that can then be regrouped to form numerical instruction sets.

The “speed” of the processor is characterized by its **frequency**, which represent how many **“cycles”** are made on the processor per second.

Modern CPU are equipped with **multiple computation cores** so they can execute several independent instruction stream in parallel. This construction allow to mutualize all the elements of the CPU that are not dedicated to compute.

The heart of the computer: the Compute Processing Unit (CPU)



What determines a CPU theoretical performances?

A CPU is characterized by its **IPC (Instruction per cycle)** capability. Increasing the IPC can be done by improving the memory hierarchy or the instruction pipeline.

The **frequency** defines how many cycle operate per second. It is limited by the physical capability of the processor to support higher power draw and to dissipate the generated heat. **Modern processor have dynamical frequency scaling that adapts to the current load on the processor (TurboBoost)!**

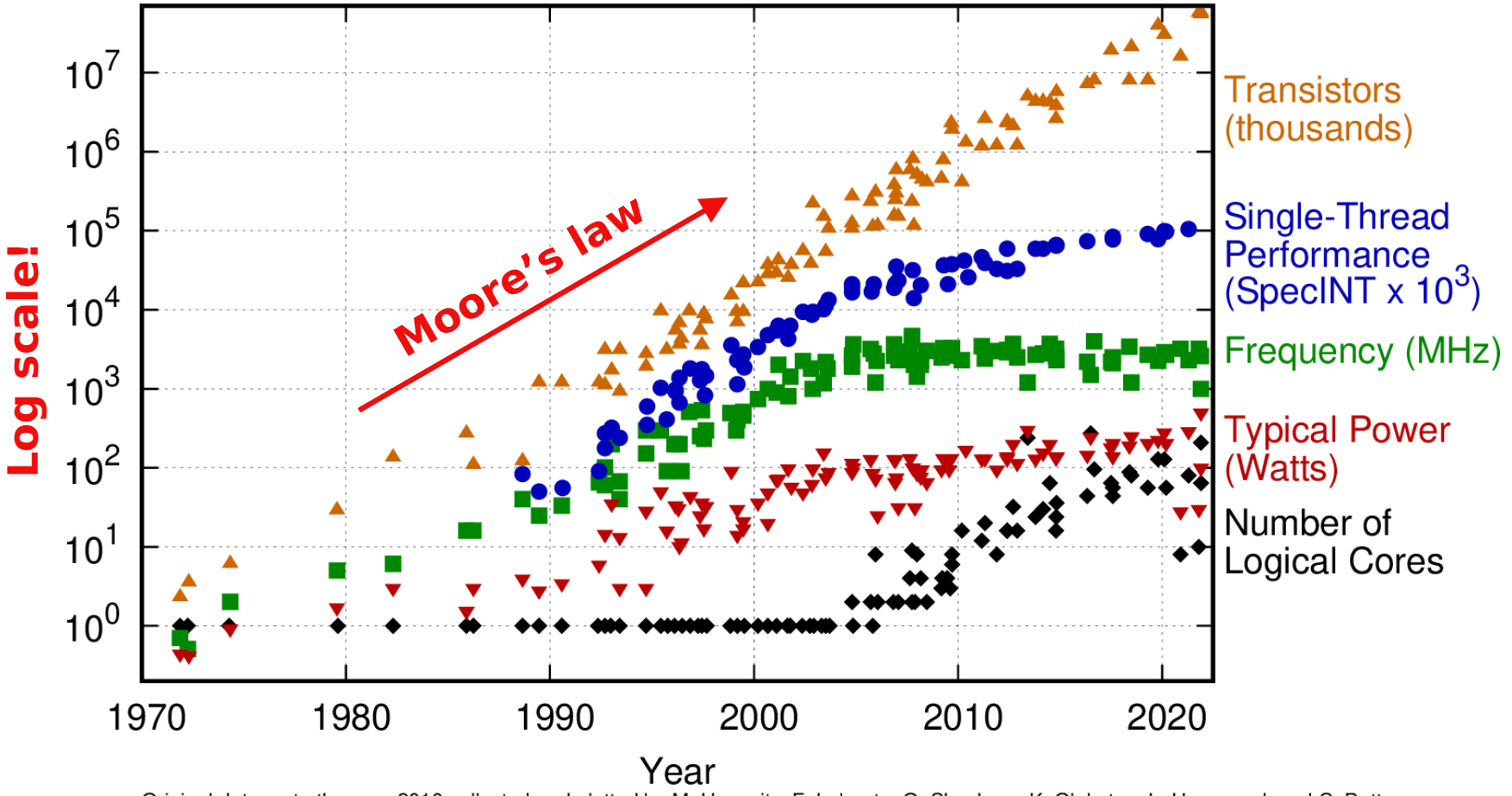
For a short load of a few seconds the frequency can go very high as it will not have the time to saturate the integrated heat spreader heat absorption. For moderate load time, the frequency can remain high depending on the cooling solution, for example heating all the water in a water cooling loop. Then, when the cooling system is saturated the CPU will stabilize to a lower frequency that match the heat dissipation capability.

The **number of cores** in a CPU can provide almost linear performance scaling depending on the application while sharing some parts of the CPU like the caches. Some CPUs have a technology called **Hyper-Threading** allowing each physical core to handle two execution steams with independent registries and low level cache.

This ensure that the actual computation units of the physical core are always saturated. Most of the time the scaling is way less good than with more physical cores. Using multiple cores will saturate the heat dissipation system much faster, so the CPU usually runs at lower speeds when all the cores are computing simultaneously

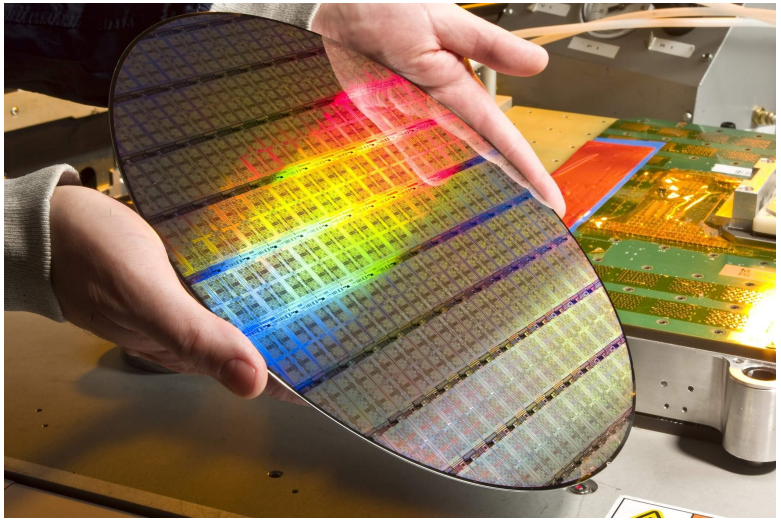
The historical mighty quest for performances

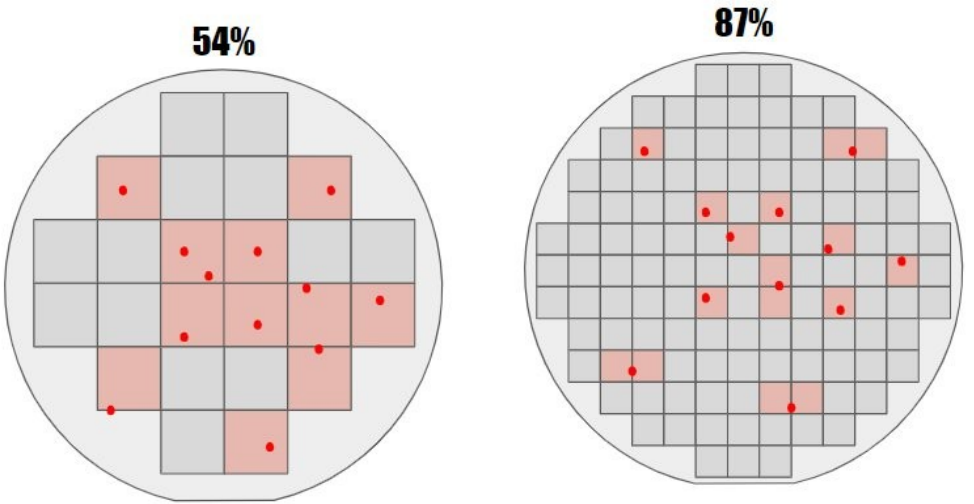
For decades, computer engineers have tried to improve the CPU performances



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 2010-2021 by K. Rupp

How to build a CPU ?





CPUs are “engraved” by batches on silicium wafers using **photolitography**.

CPU dies are getting **larger and include more cores** to increase the computing power while frequency stagnate. This increased size combined to the increase in **engraving difficulty at low finesse**, imply that the **faulty die rate increases.** This lead to higher production cost for high end processors.

The quality of the engraving can vary at the scale of the wafer, leading to a dispersion in efficiency for a given CPU chip model of a few % → **This principle is commonly reference to as « silicon lottery ».**

At the scale of a cluster the cumulated variability for each component induces a variability on the full system efficiency of up to 10%

Efficiency of a computing system or facility

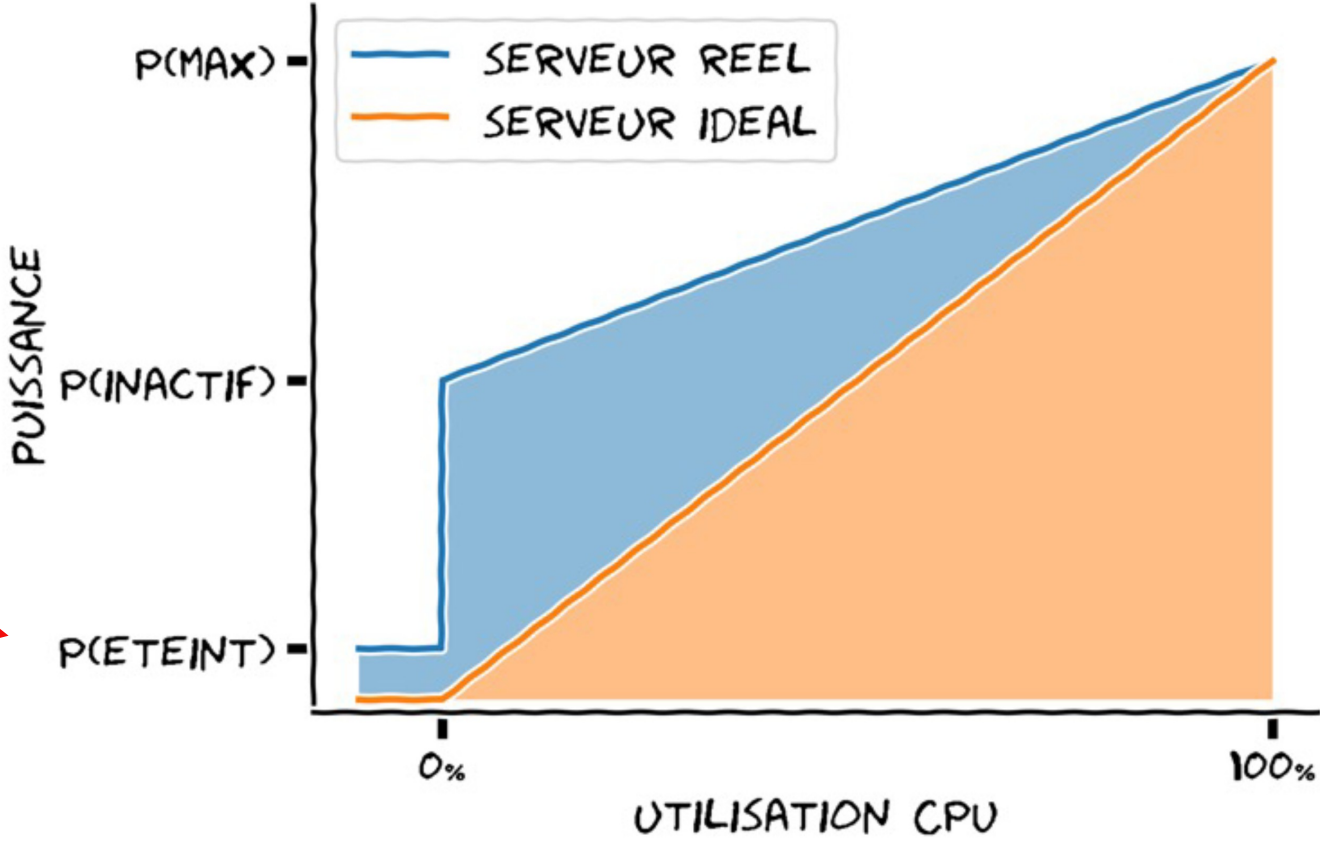
All idle systems have a baseline power draw. For individual modern systems it is often driven by the peripherals (e.g., screen, network cards, etc.), by the ram, and by the residual CPU activity coming from it being ready to handle possible incoming loads.

Not zero!

For super computing facilities this baseline power is usually quite high.

Still not zero!!!

This is more the case for servers. It remains true for almost any device, but the residual consumption of turned off modern ICT becomes negligible.



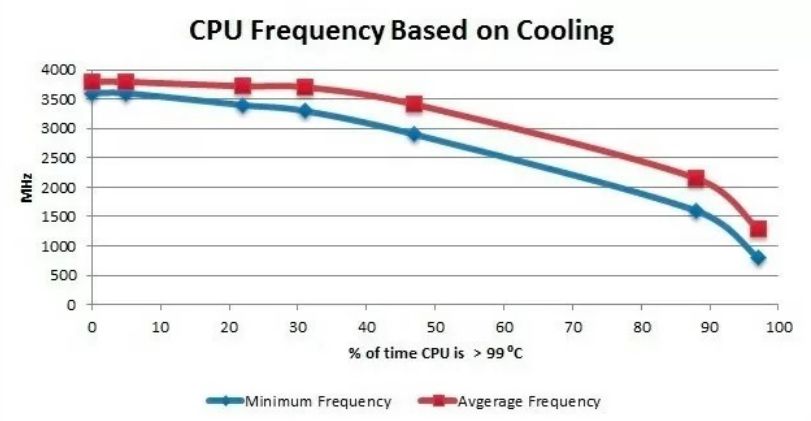
From David Guyon

When comparing **two applications**, the baseline can be subtracted.
When evaluating the **energy consumption** the total idle power must be redistributed to all the users of the facility.

Measuring the energy consumed by a computation

Before measuring the system power, we need to reduce **compute efficiency and power draw variability.**

During computation, the system will heat up, which can increase the power draw of the cooling solution, and will lower the frequency for systems that adapt automatically.



Before taking measurement, chose between the cold or hot state as your baseline.

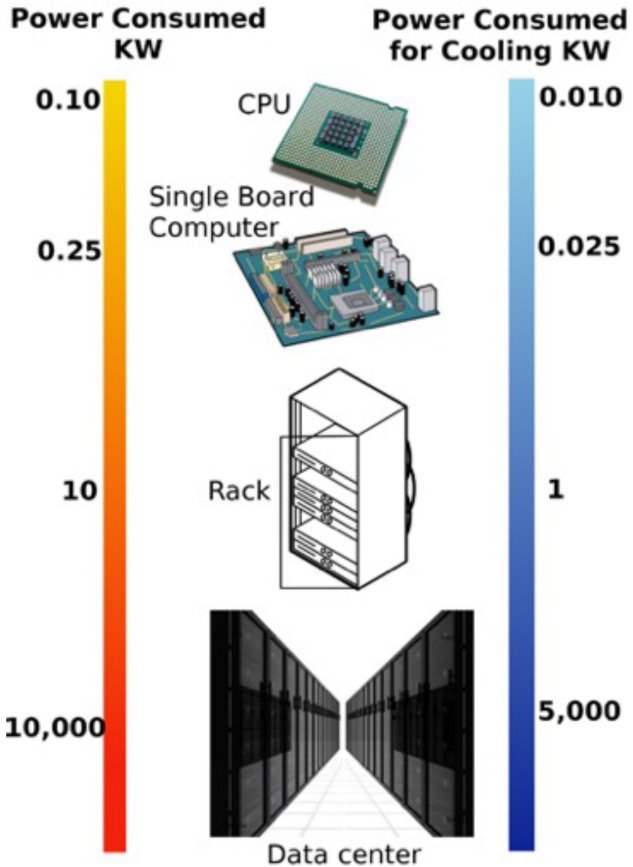
- **Cold state** is useful for comparing applications with small computation times but it will likely under-estimate the real energy consumption of the deployed software.
- **Hot state** (default) is a more accurate measurement but it increases the time and cost of the measurement as we need to « heat up » the system before.

Once the system put in the right state you can start measuring the **increase in power draw induced by the computation**. At the end the energy consumed by your computation is:

$$E(J) = \Delta P(W) \times T(s)$$

This is an approximation as ΔP can vary during complex computations.
A more complete measurement would integrate the energy consumed over small timesteps.

How to improve efficiency and at which scale?



→ More low power processors
→ Improvement of the IPC

→ More multi-core systems and improved efficiency
→ More dedicated accelerators (FPGA, GPU, NPU)

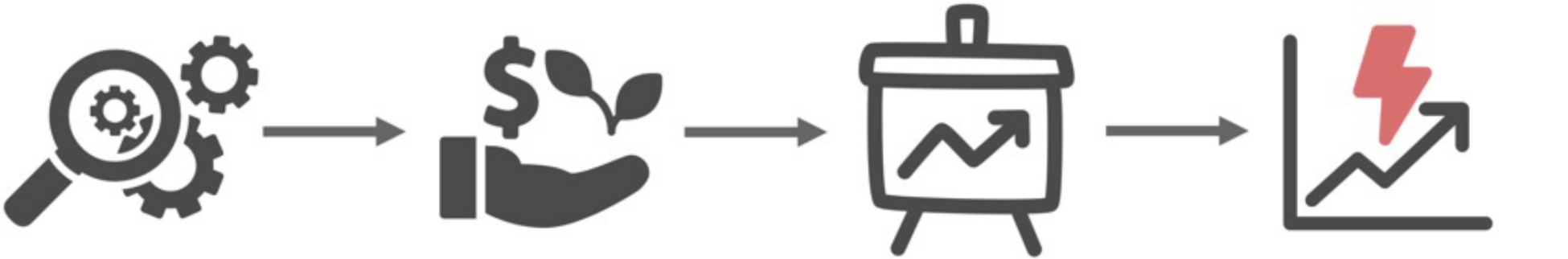
→ Optimization of data placement
→ Systematize parallel computing
→ Improvement on the communication network

→ Improving the handling of bottlenecks
→ Strategies to smooth the load
→ Automatic scaling (turning on and off parts of the cluster)

From Anne-Cecile Orgerie

Improving data centers efficiency only is unlikely to be enough,
as illustrated by the ICT distributed contribution to the global CO2-e emissions.

Watch out for the rebound effect (Jevons' Paradox)!



Some optimizations will result in a more accessible applications, increasing the number of users, filling any achieved saving and increasing the global demand.

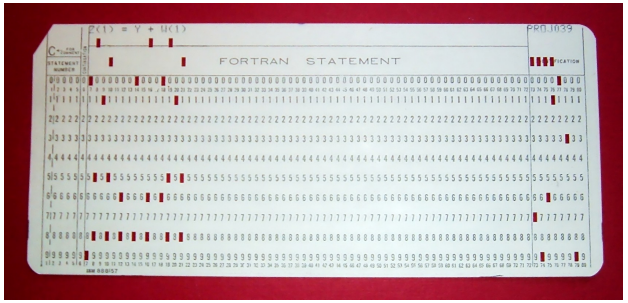
Optimizations on applications for which the demand is not changing much or that would already have room to grow but does not are the most beneficials.
→ but beware the economical rebound !

Any saving in one activity might be reinvested to other polluting applications!

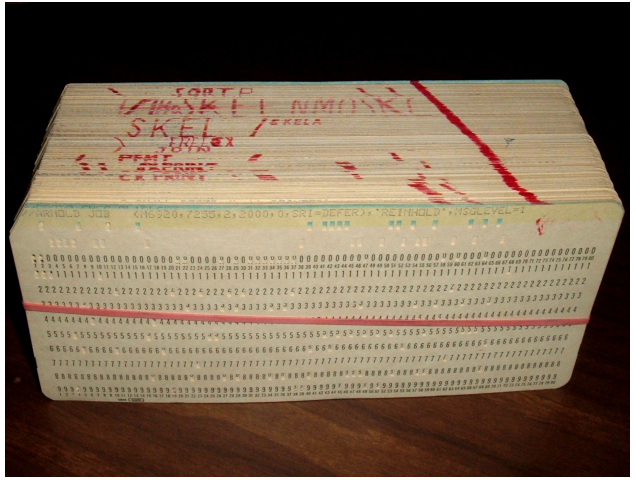
→ E.g., during the COVID pandemic, the investment in electronic devices exploded, not only due to home working, but also to the saving from the absence of other form of usual expenses !

Programming optimization in practice

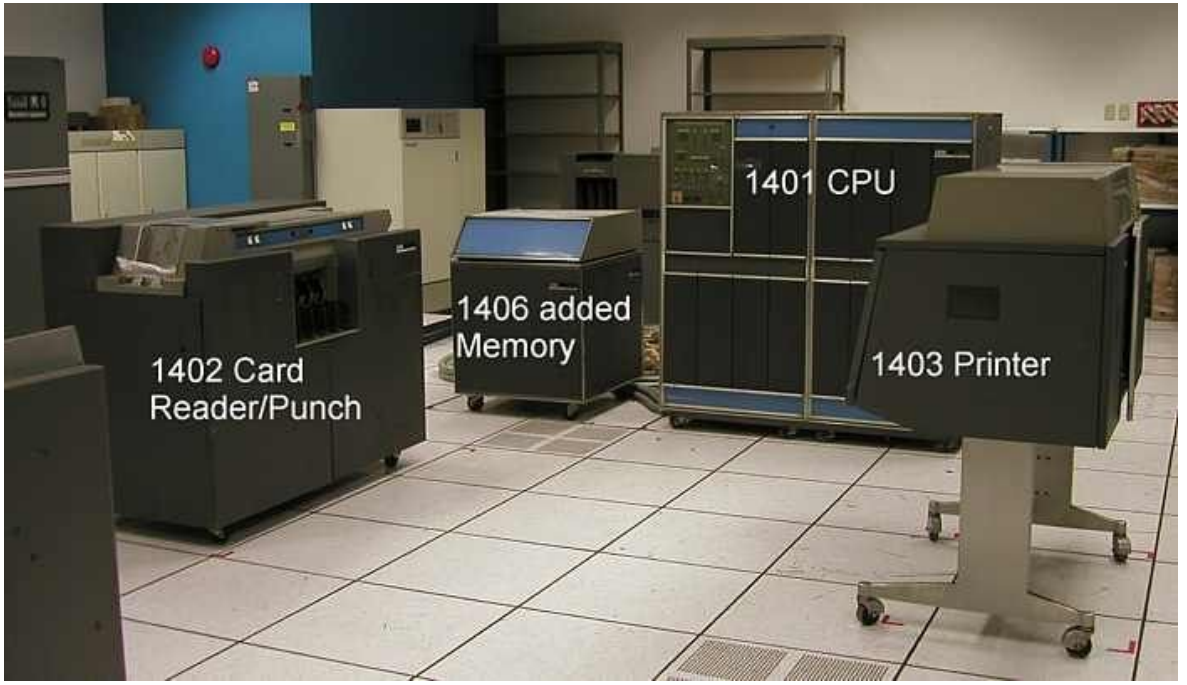
Historically, optimization and good coding practices was a necessity!



A single Fortran instruction on a punched card



The deck of card for a full Fortran program
Each card represent a simple line of code

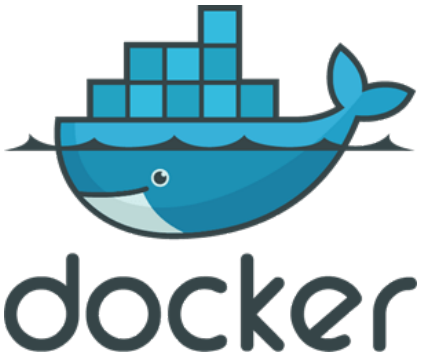


An IBM 1401 spec sheet:

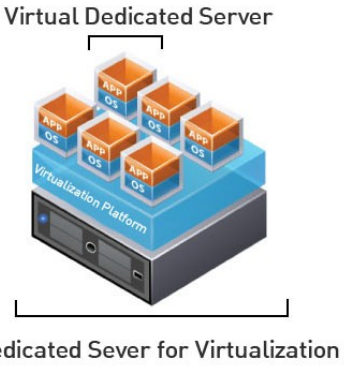
- 6-bit diode-transistor logic, with with a frequency of 90 Hz!
- 16000 Bytes of magnetic-core memory with the 1406 extension
- No storage by default, but can add a 1311 extension to support 2MB of non-volatile memory

Current programming trends are not efficiency driven

Interpreted or scripting programming languages
→ Execution overheads



Everything in containers
→ De-multiply core software



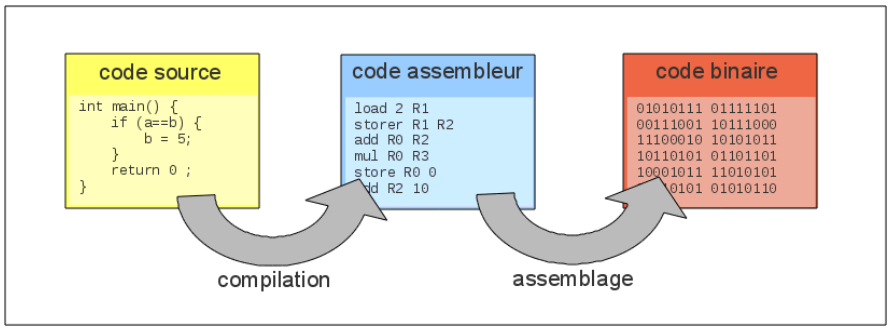
Virtual machines
→un-optimized loads

These days, the focuses are instead:

- Reducing development time (developers are costly)
- Ease of use and deployment (reduce friction)
- Improving security
- Constant accessibility (service continuity)
- Reduction of infrastructure costs (mutualize resources)

Compiled VS interpreted programming languages

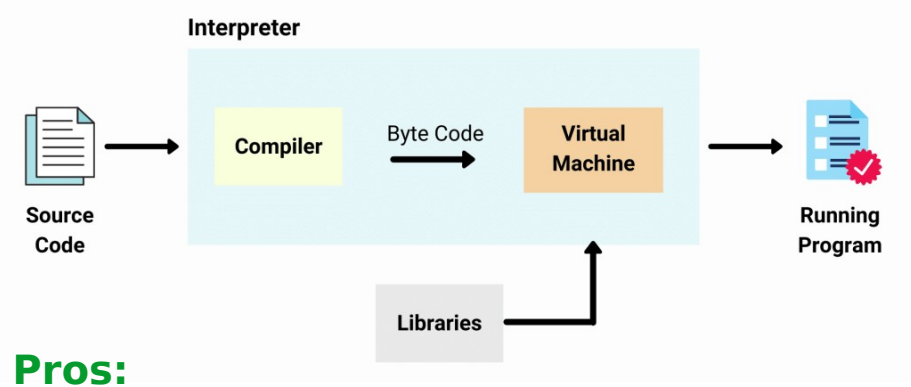
Compiled languages
(C, Fortran, Pascal, Rust, ...)



Pros:
- Very fast! Thank to the large scope of the compiler allowing strong optimization.
- Low level. closer to the machine language, expose more easily the system structure.

Cons:
- Strict syntax, and can't run if any error is detected by the compiler
- More error prone as more control also induce more responsibility (e.g., memory management)

Interpreted languages
(Python, PHP, Ruby, Javascript, ...)



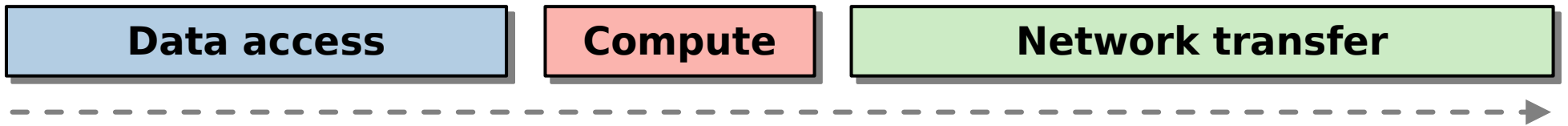
Pros:
- Compliant syntax
- Usually higher lever, do complex things in a few lines without noticing

Cons:
- Slower due to on the fly compilation overhead and to the small scope of the interpreter
- Hide the inner working of the system reducing optimization possibility and global understanding of the program requirements in terms of resources and environment.

Energy consumption comparison between programming languages → Pereira et al. 2021

Performance bottleneck

The speed of a program or a system is most of the time limited by its slower element!

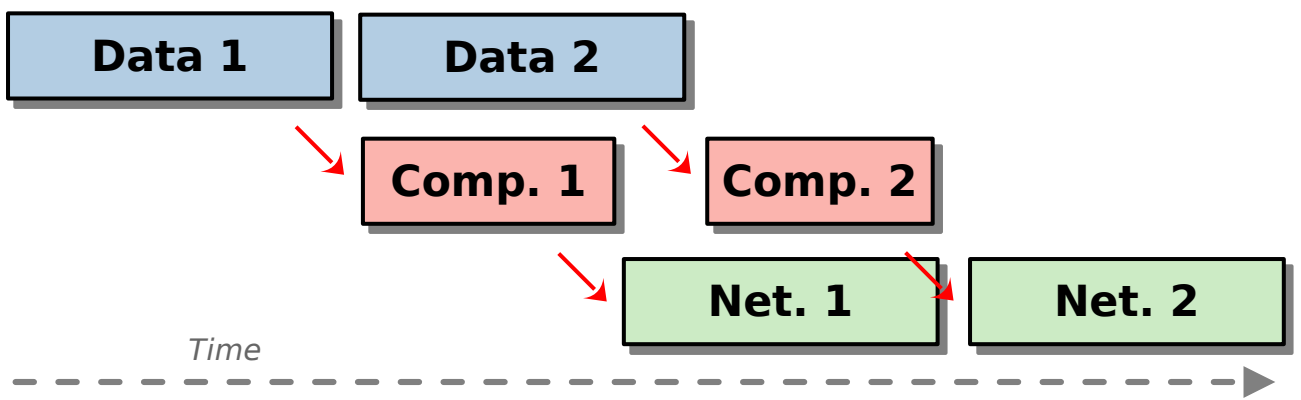


With this distribution, optimizing the “compute” part is not very useful.

Sill, it is possible to transfer stress from a sub-system to an other to a given extent.

E.g., the data to transfer through the network can be compressed first to reduce their size. The compression time now add to the compute part, and can potentially be optimized.

Also some parts can be executed concurrently !



Splitting problems into concurrent independent tasks that can overlap is one of the keystone of computing resources optimization !

All waiting time of each sub system must be reduced to the minimum

Random Access Memory

RAM = constant access time regardless of the data placement

The RAM stores data that will need fast or frequent access and must therefore be fast.

From the system standpoint, memory is **linear, continuous, and decomposed into cells,** each one referred to by **an address** (stored into the MMU directly)**.**

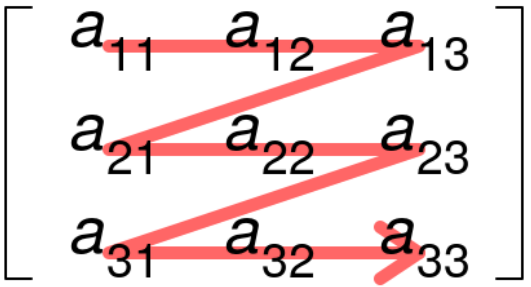
Programs reserve **chunks of memory** to work. They can be **discontinuous**.

For data arrays, memory continuity is of the outermost importance, therefore they are reserved as continuous chunks and referred to by the address of their first element.

Storing a 2D matrix of 4 by 5 elements, required an continuous chunk of 20 elements. After declaring the array we only get access to its starting position address and data type. Recovering an element is then a matter of address arithmetic.

Multidimensional data in linear storage

Row-major order

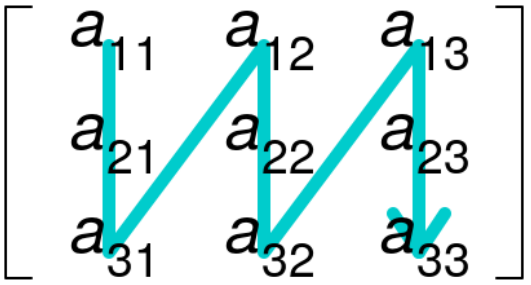


Default order for C, C++, Python, etc…

In a[i][j] the “fast” index on continuous data is the one on the **right (j)**

When flattened a[i*3+j]

Column-major order



Default order for Fortran

In a[i][j] the “fast” index on continuous data is the one on the **left (I)**

When flattened a[i+j*3]

In practice the programmer is free to use the encoding he wants by accessing directly the flattened array.

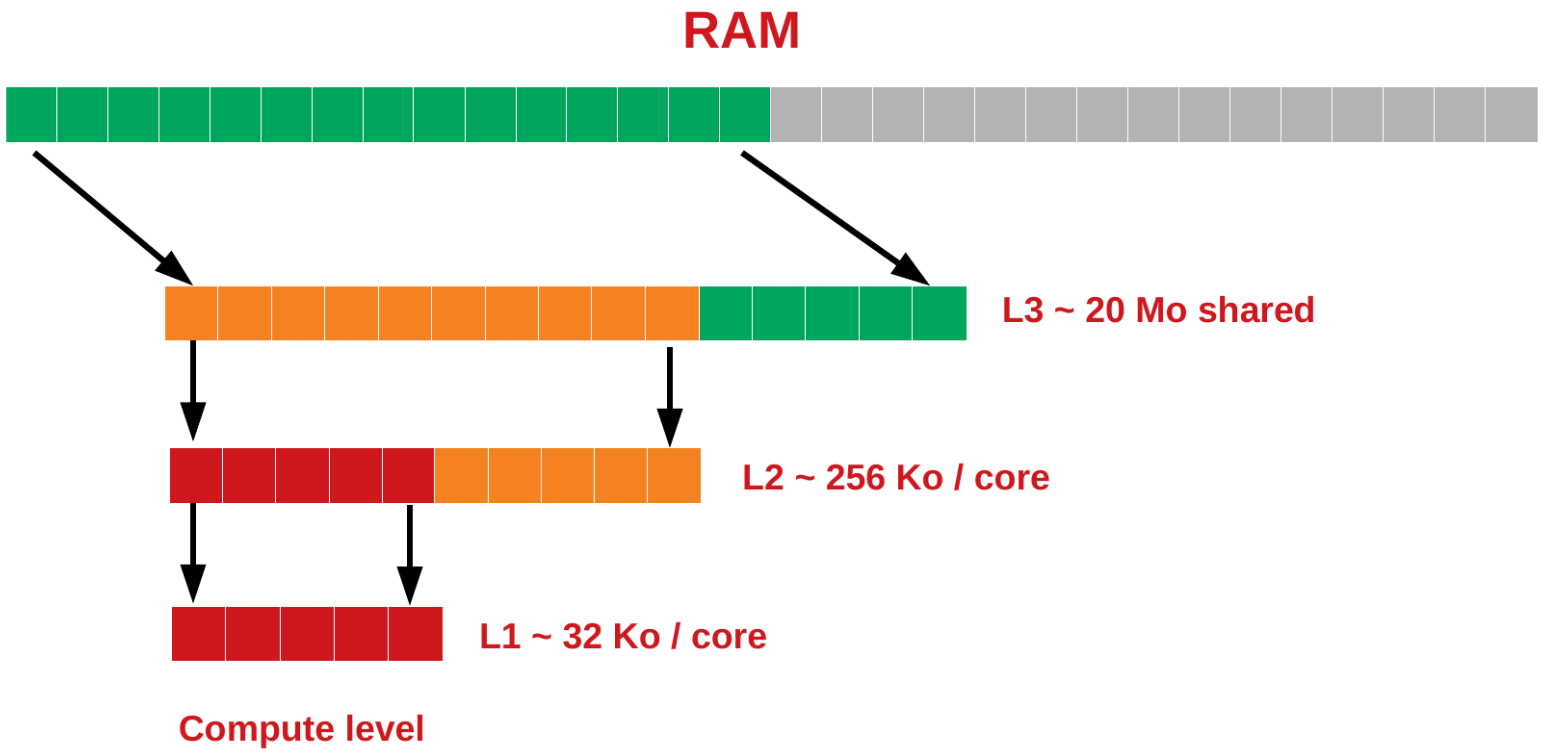
We note that it is possible to use “linked list” so the data are continuous only for one dimension. While it can ease programming it usually have strong negative effects on performances.

Data access and “cache miss”

CPUs are equipped with « caches » that are **successive levels of faster and smaller memory.**

When accessing a data, **a full chunk is loaded into the caches** so if the next operation requires the next element in memory it can simply used the cached value instead of loading it from the RAM.

Breaking this access scheme will result in **cache miss**, which mean that data has to be read from the RAM more than necessary. Typically the case when accessing a 2D array in the wrong order.



Let the compiler do the work!

The compiler is the best friend you can have regarding optimization.

CPUs are complicated and comprises **a lot of advanced instructions** (see CISC and RISC) that can perform complex operations in a **reduced amount of cycles.** Using them explicitly require meticulous programming.

Luckily, we can ask the compiler to do the work of converting simple codes to strongly optimized ones using all the available instructions. For this the compiler has to analyze large parts of the code to identify dependencies. It will then perform only the authorized optimizations.

While individual aspect of optimization can be specified, we usually rely on **the general -O flag**, with 3 different levels from -O1 to -O3.

These optimization depends on the compiler!
We only list them for GCC here

From -O1 to -O3

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion2
-fif-conversion
-finline-functions-called-once
-fipa-pure-const
-fipa-profile
-fipa-reference
-fmerge-constants
-fmove-loop-invariants
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-pta
-ftree-ter
-funit-at-a-time
```

From -O2 to -O3

```
-fthread-jumps
-falign-functions  -falign-jumps
-falign-loops  -falign-labels
-fcaller-saves
-fcrossjumping
-fcse-follow-jumps  -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-fgcse  -fgcse-lm
-fhoist-adjacent-loads
-finline-small-functions
-findirect-inlining
-fipa-cp
-fipa-bit-cp
-fipa-vrp
-fipa-sra
-fipa-icf
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fsched-interblock  -fsched-spec
-fschedule-insns  -fschedule-insns2
-fstore-merging
-fstrict-aliasing -fstrict-overflow
-ftree-builtin-call-dce
-ftree-switch-conversion -ftree-tail-merge
-fcode-hoisting
-ftree-pre
-ftree-vrp
-fipa-ra
```

-O3 only

```
-finline-functions
-funswitch-loops
-fpredictive-commoning
-fgcse-after-reload
-ftree-loop-vectorize
-ftree-loop-distribute-patterns
-fsplit-paths
-ftree-slp-vectorize
-fvect-cost-model
-ftree-partial-pre
-fpeel-loops
-fipa-cp-clone
```

First practical work Matrix multiplication optimization

$$C(i, j) = \sum_{k} A(i, k) \times B(k, j)$$

Mostly follow Algorithmica
by Sergey Slotin

Matrix operation optimization history

$$C(i, j) = \sum_k A(i, k) \times B(k, j)$$

Matrix operations are of uttermost importance

→ They appears in a lot a computing problems!

→ They represent the vast majority of the computations done by AI models.

Matrix operations have received a lot of **optimization attention** and chips manufacturer have built **dedicate hardware and instructions** for this operation!

The default algorithm **complexity scales with the cube of the matrix side size**
→ if M=N=K, it scales in $\Theta(N^3)$.

Algorithms with a lower complexity exist but they often work in a way that make them difficult to optimize for classical computing hardware.

Many libraries are dedicated to matrix **multiplication or other linear algebra operation (BLAS) acceleration** using various types of hardware:

OpenBLAS, IntelMKL, MAGMA, CuBLAS, rocBLAS, ...

Optimization: Memory continuity

$$C(i, j) = \sum_k A(i, k) \times B(k, j)$$

The naive implementation is composed of 3 loops. The first two over the i and j index specify the coordinates of a cell in the C matrix, and the last one indexed by k spans over the shared dimension between A and B. This naive implementation is memory bandwidth limited as the accessed elements are not continuous, **inducing cache miss.**

$$C(i, j) = \sum_k A^T(k, i) \times B(k, j)$$

The first main optimization that can be done is to **transpose the matrix that is responsible for the cache-miss**, the A matrix in our case (with column-major encoding).

Optimization: SIMD vectorized operations

When **accessing continuous data in memory,** the operations can be vectorized using specific instruction sets that **use the SIMD principle**.

E.g., in **matmul_v3** the main instruction **`C[j*M+i] += A`^{T}`[i*K+k] * B[j*K+k]`** can be expressed as a **SIMD operation using the FMA (Fused Multiply Add) instruction from AVX-2** set that do multiply, add, and round using a reduced number of CPU cycles.

This type of operation works on **vector register data structures**:

```
typedef float vec __attribute__ (( vector_size(32) ));
```

Using AVX-2, **the vector size is 256 bit, so it can store 8 floats of 32 bit**.

When doing operations on vector data, the compiler will automatically use the vectorized FMA operation.

An explicit vectorized matmul that uses vectorized versions of A and B and then do the accumulation on the individual vectors is presented in matmul_v4.

Optimization: CPU register re-use

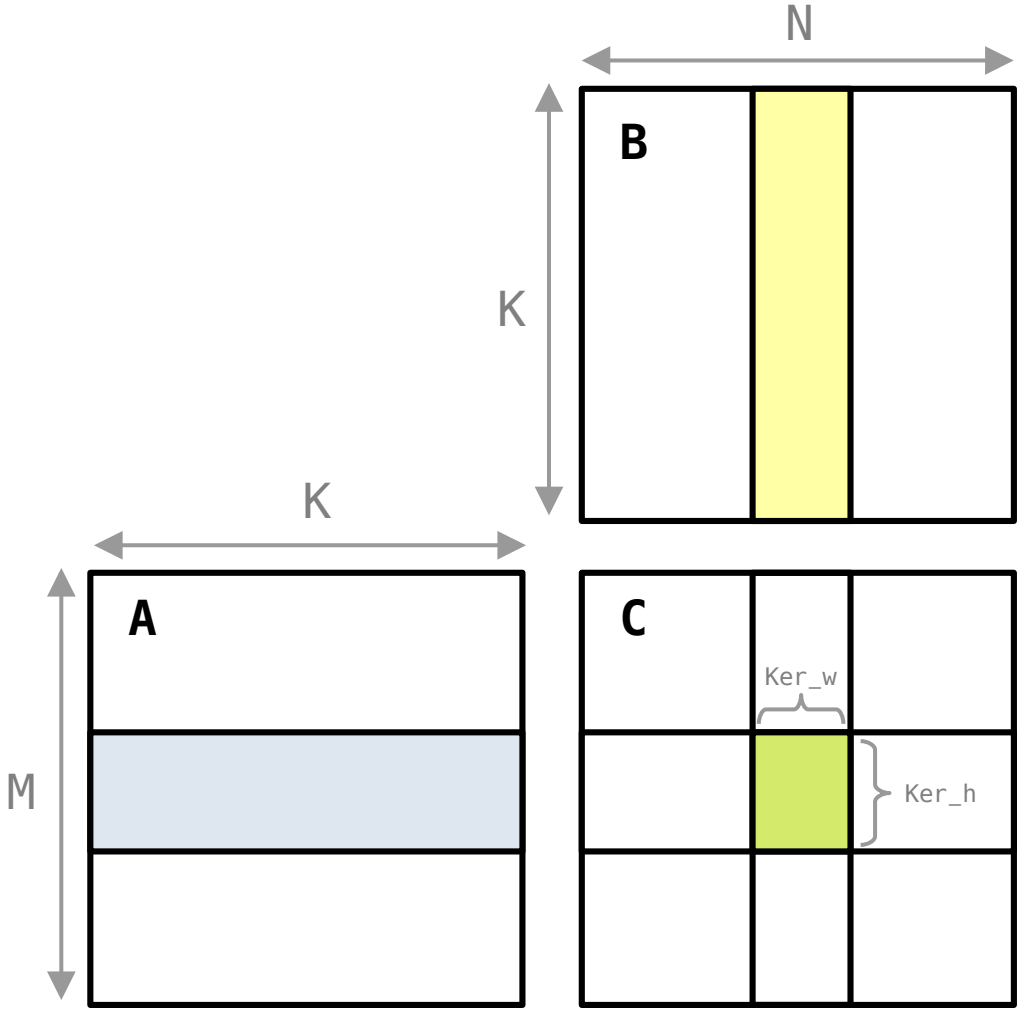
Even with continuous memory access, the implementation is likely to remain **limited by the memory bandwidth** due to the **high number of data reading and writing** that need to be done in contrast to compute operations.

We must reduce data movement and maximize the use of CPU caches while letting the compiler do the hard work as much as possible.

One approach is to ensure that data that are loaded into the **CPU register are reused as much as possible** before being replaced by others.

This can be combined with a **SIMD vectorized computation inside a kernel** that is tasked to compute the result for a sub-matrix **`C[i:i+di][j:j+dj]`.**

Optimization: SIMD vectorized kernel computation



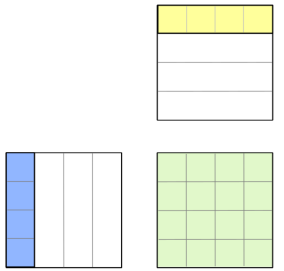
Instead of accumulating the product of all the K elements for a row in A and a column in B at once, we use a progressive accumulator.

The kernel has a size that correspond to ker_h lines from A, and ker_w lines from B. For each value of k, the kernel takes a vertical vector in A and an horizontal vector of B both of the size of the kernel.

Doing the dot product of these two vector we can recover their contributions to each cell and accumulate it into a register memory.

Accumulate version (strong re-use!)

$$
\begin{aligned}
C_{0,0} &= A_{0,0}B_{0,0} + A_{0,1}B_{1,0} + A_{0,2}B_{2,0} \\
C_{0,1} &= A_{0,0}B_{0,1} + A_{0,1}B_{1,1} + A_{0,2}B_{2,1} \\
C_{0,2} &= A_{0,0}B_{0,2} + A_{0,1}B_{1,2} + A_{0,2}B_{2,2} \\
C_{1,0} &= A_{1,0}B_{0,0} + A_{1,1}B_{1,0} + A_{1,2}B_{2,0} \\
C_{1,1} &= A_{1,0}B_{0,1} + A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
C_{1,2} &= A_{1,0}B_{0,2} + A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,0} &= A_{2,0}B_{0,0} + A_{2,1}B_{1,0} + A_{2,2}B_{2,0} \\
C_{2,1} &= A_{2,0}B_{0,1} + A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
C_{2,2} &= A_{2,0}B_{0,2} + A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{aligned}
$$



Naive version (no re-use)

Optimization: SIMD vectorized kernel computation

Inside the kernel, the operations can simply be written as:

`C[i,j] += A[i] * B[j]`

This can also be expressed as a SIMD FMA instruction.
To **saturate the execution port** of this instruction we must work on at least 10 registers.

The maximum register count is 16, so we can define a kernel with
ker_w = 16 and ker_h = 6, for total of 16/8 x 6 = 12 vector registers to keep a margin.

For each position k, the kernel must contain two loops:

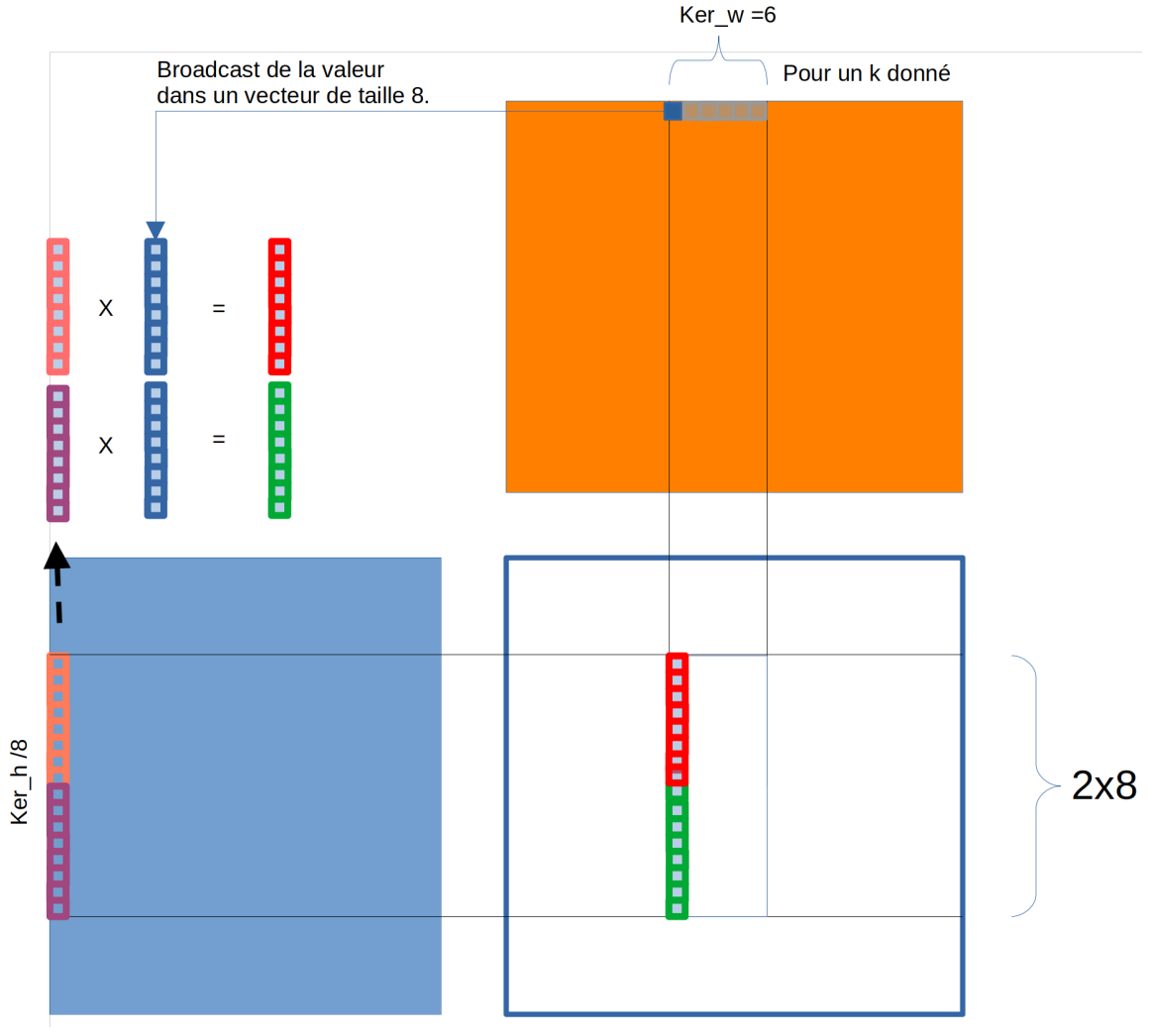
1) A loop over the 6 width kernel element of matrix B. At each step, the current value must be broadcast to a vector with 8 identical values.

2) A loop over the the height kernel element of matrix A. This time the 16 elements are decomposed over 2 vectors.

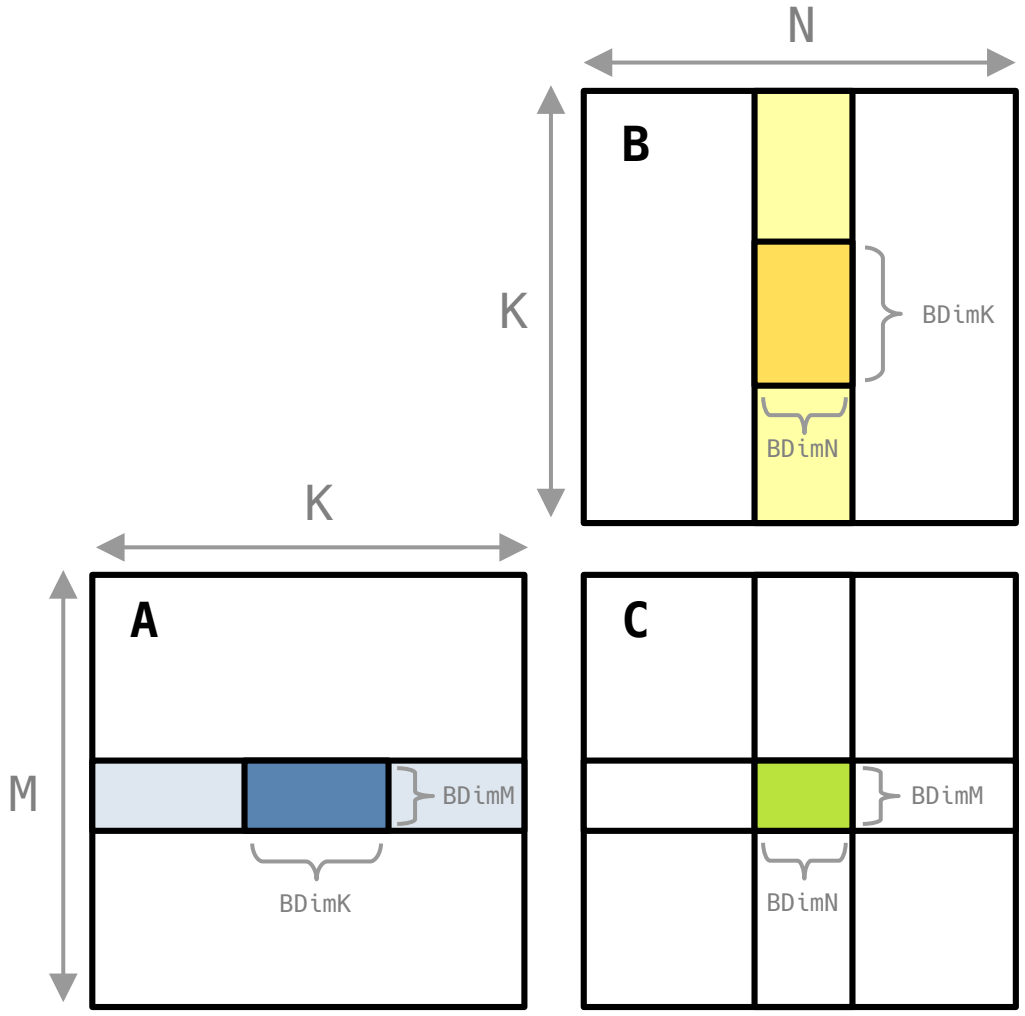
3) The two current vectors can be multiplied to accumulate 8 products in the kernel corresponding to **`C[i:i+8,j] += A[i:i+8] * B[j]`**

This kernel must be called in a double loop for all its possible positions in C

Optimization: SIMD vectorized kernel computation



Optimization: Fully blocked version with kernel



The previous version is still bandwidth limited!
To further optimize we must maximize re-use of data in the different CPU caches by working on blocks of the matrices.

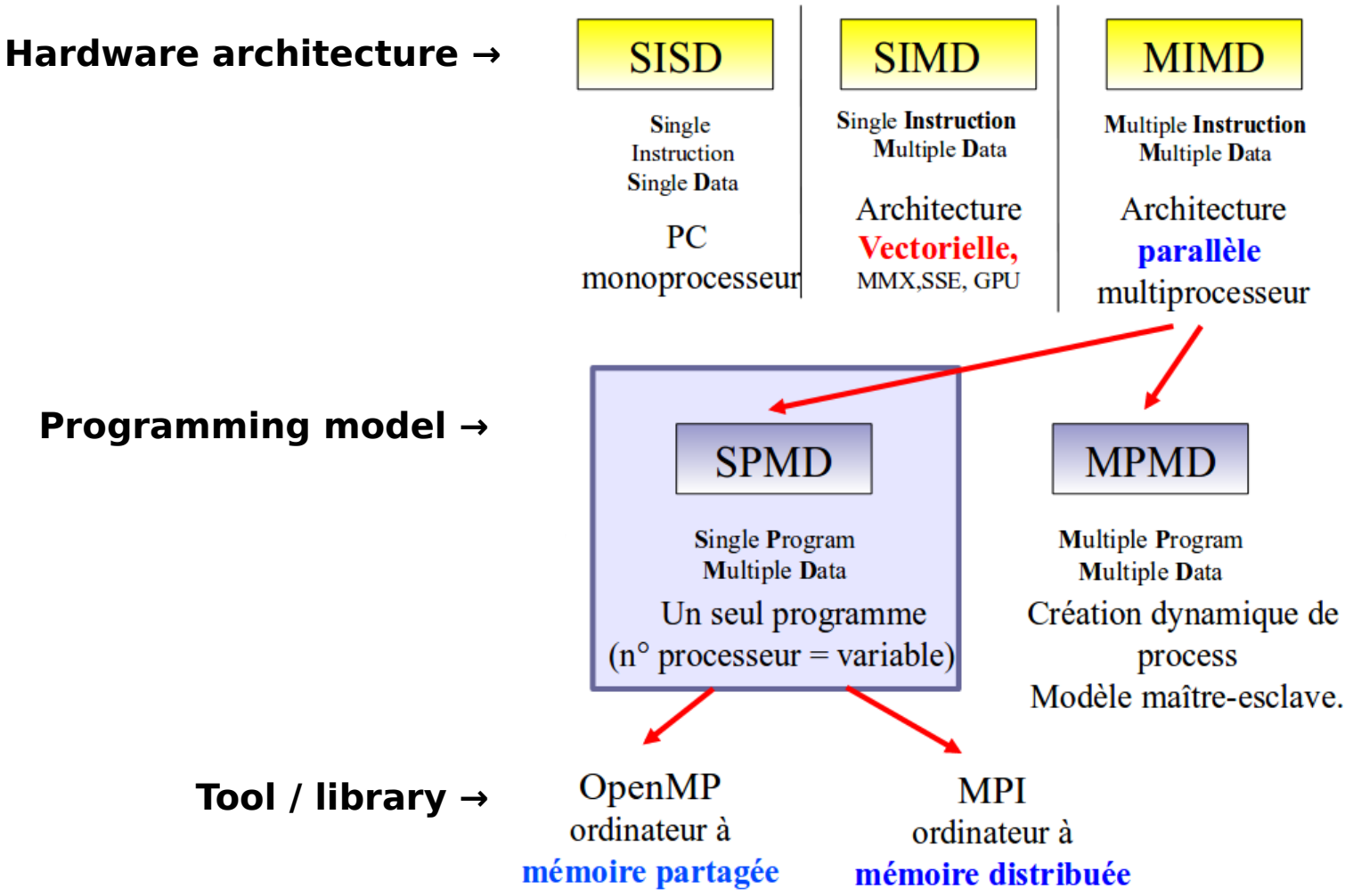
Our biggest cache miss is the frequent change of columns in A through the loop over K. Our fastest L1 cache must contain as much columns of A as possible. This will also define the number of lines in B. **We defined BDimK the number of columns of A.**

Our second cache miss is for the change in columns in B through the loop over N.
This one will mostly define our L2 cache. **BDimN is the number of columns of B.**

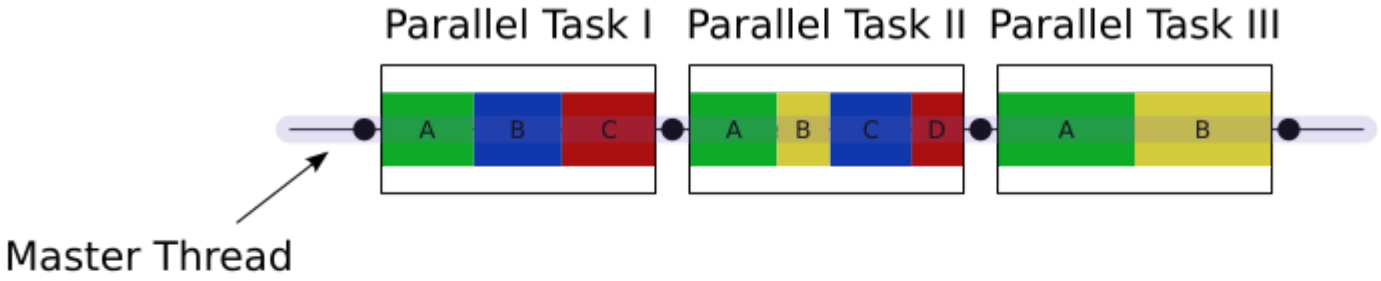
Finally, we want to cache rows of A in a way that fill the L3 cache considering the two previous dimensions. **BDimN is the number of row of A.**

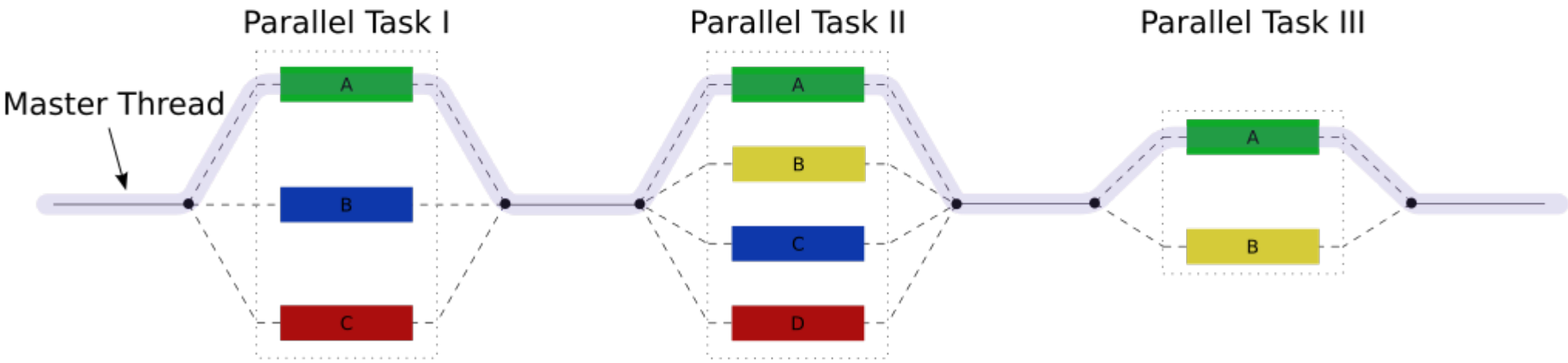
Each block must be a multiple of the kernel size, so a loop over all the kernel position in the block can be used !

CPU parallelization paradigms



OpenMP principle





OpenMP « threads » are logical entities and does not refer directly to CPU cores or CPU threads (with Hyper Threading)

→ There can be less or more OpenMP threads than cores

OpenMP syntax

OpenMP is used in the form of « pre-processing » directives.

These lines will be replaced by actual code lines at the start of the compiling.

```
#pragma omp parallel shared(…) private(…)
{
    [code inside the parallel region]
    #pragma omp for schedule(TYPE,N)
    for(i=0, i<X; i++){
        [In loop code]
    }
}
```

OpenMP directives are only taken into account when adding a specific compilation flag.

For gcc it is `-fopenmp`

The number of threads X in each parallel is defined outside of the code by setting an environment variable of the system, which can be done with:

```
export OMP_NUM_THREADS=X
```

The logical threads are distributed on the system, occupying physical core and threads.

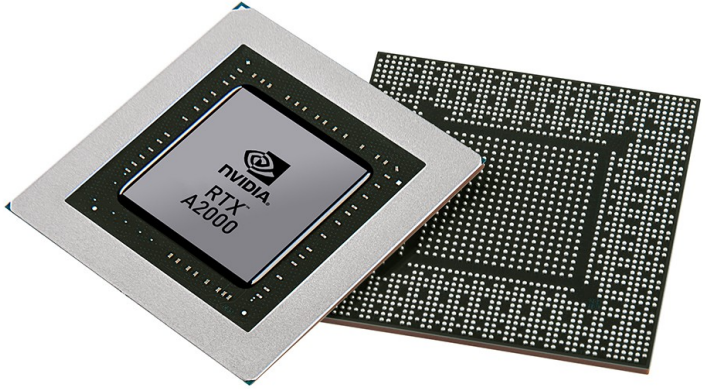
Second session

What about computing on GPU?

GPU (Graphical Processing Unit) are massively parallel computing chips dedicated to SIMD like operations (thousands of cores). Most image processing algorithm apply the same transformation to millions of pixels, hence the SIMD formalism.

GPU have the same form factor than CPU but usually come as a dedicated daughter board with their own large cooling system as they can have a much higher power draw than CPU!

GPUs are not suited for all tasks, but for those they where designed for they pack a huge amount of computing power, which include matrix multiplication!



Nvidia H100 GPU spec-sheet (AI dedicated)

Graphics Processor	
GPU Name:	GH100
Architecture:	Hopper
Foundry:	TSMC
Process Size:	4 nm
Transistors:	80,000 million
Density:	98.3M / mm²
Die Size:	814 mm²

Graphics Features	
DirectX:	N/A
OpenGL:	N/A
OpenCL:	3.0
Vulkan:	N/A
CUDA:	9.0
Shader Model:	N/A

Graphics Card	
Release Date:	Mar 21st, 2023
Generation:	Tesla Hopper (Hxx)
Predecessor:	Tesla Ada
Production:	Active
Bus Interface:	PCIe 5.0 x16

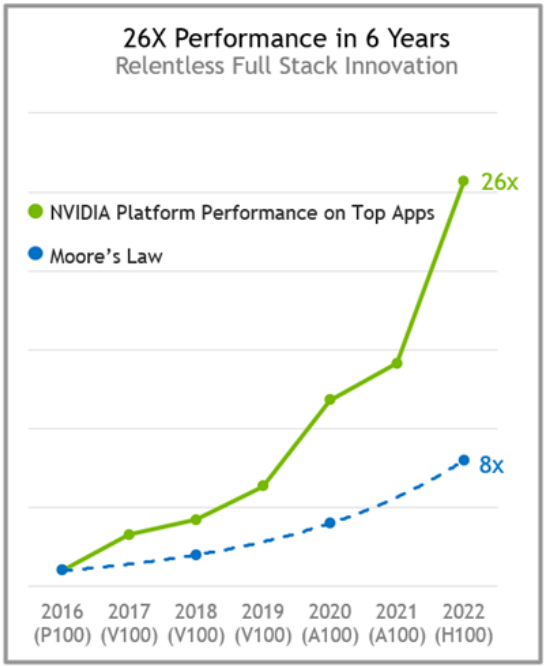
Board Design	
Slot Width:	Dual-slot
Length:	268 mm 10.6 inches
Width:	111 mm 4.4 inches
TDP:	350 W
Suggested PSU:	750 W
Outputs:	No outputs
Power Connectors:	1x 16-pin
Board Number:	P1010 SKU 200

Clock Speeds	
Base Clock:	1095 MHz
Boost Clock:	1755 MHz
Memory Clock:	1593 MHz 3.2 Gbps effective

Render Config	
Shading Units:	14592
TMUs:	456
ROPs:	24
SM Count:	114
Tensor Cores:	456
L1 Cache:	256 KB (per SM)
L2 Cache:	50 MB

Memory	
Memory Size:	80 GB
Memory Type:	HBM2e
Memory Bus:	5120 bit
Bandwidth:	2,039 GB/s

Theoretical Performance	
Pixel Rate:	42.12 GPixel/s
Texture Rate:	800.3 GTexel/s
FP16 (half):	204.9 TFLOPS (4:1)
FP32 (float):	51.22 TFLOPS
FP64 (double):	25.61 TFLOPS (1:2)



From Nvidia

From Techpowerup

GPU architecture

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The schematic Figure 1 shows an example distribution of chip resources for a CPU versus a GPU.

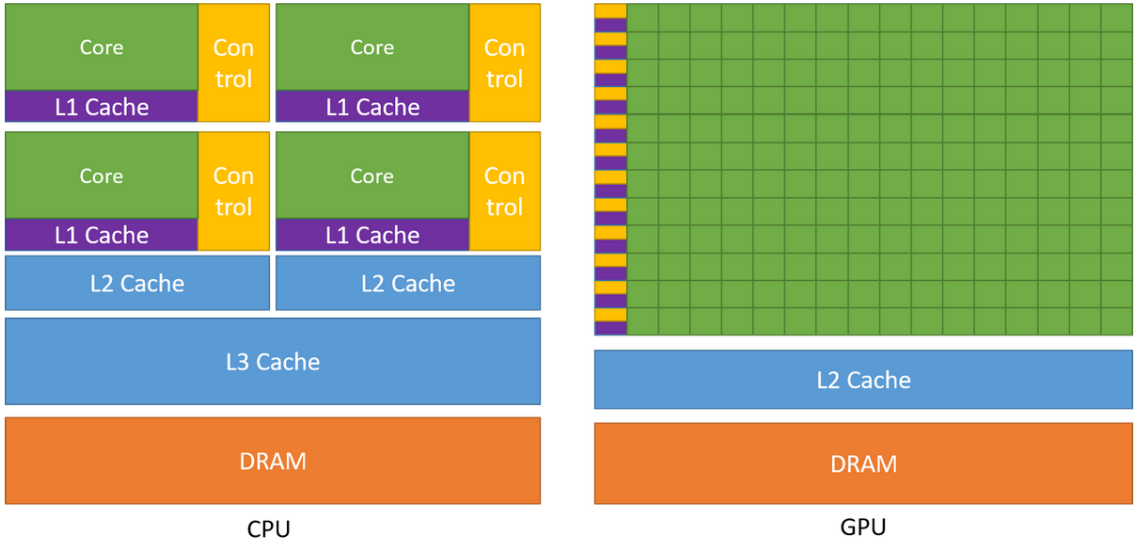
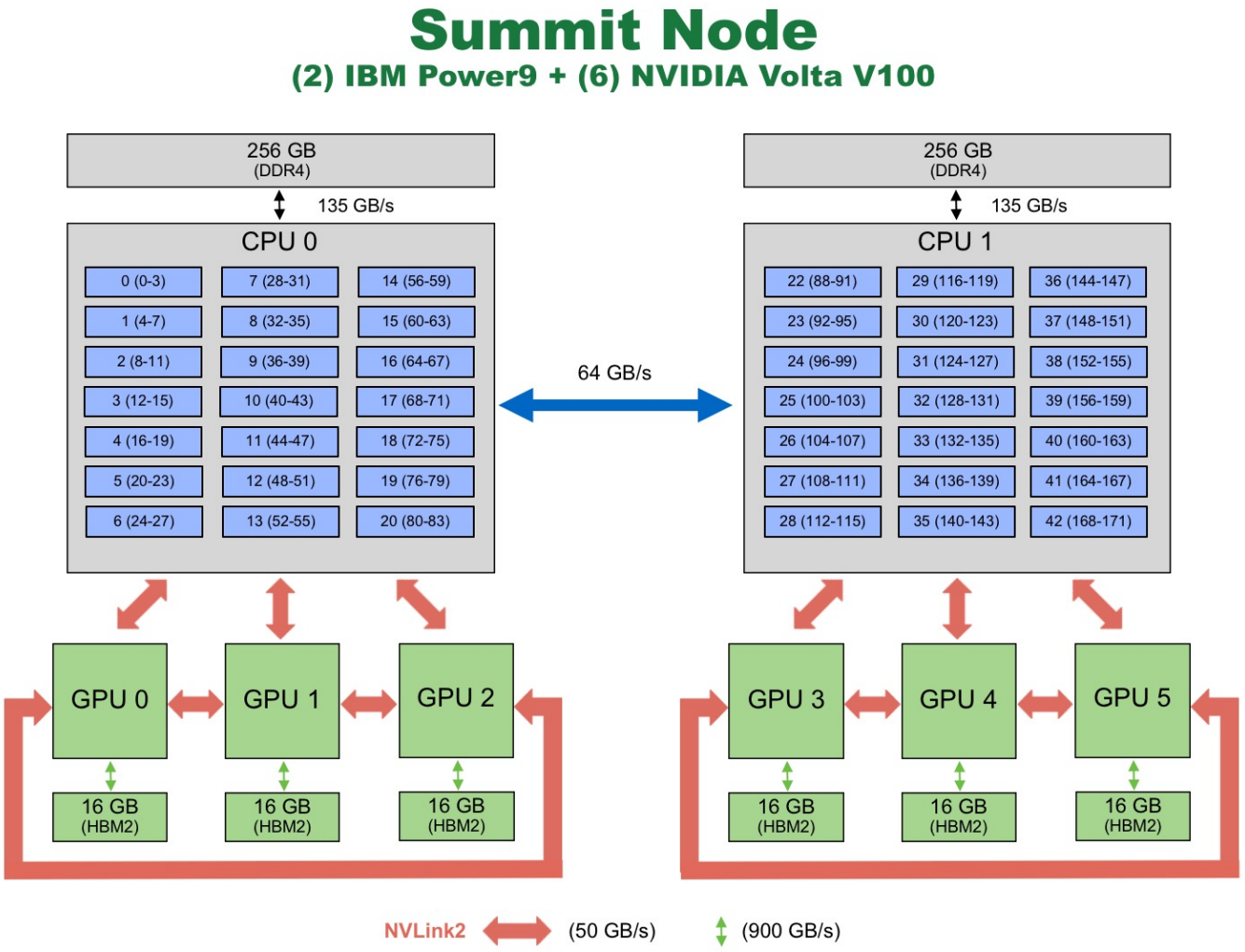


Figure 1: *The GPU Devotes More Transistors to Data Processing*

Distribution in GPU clusters



General Purpose GPU programming

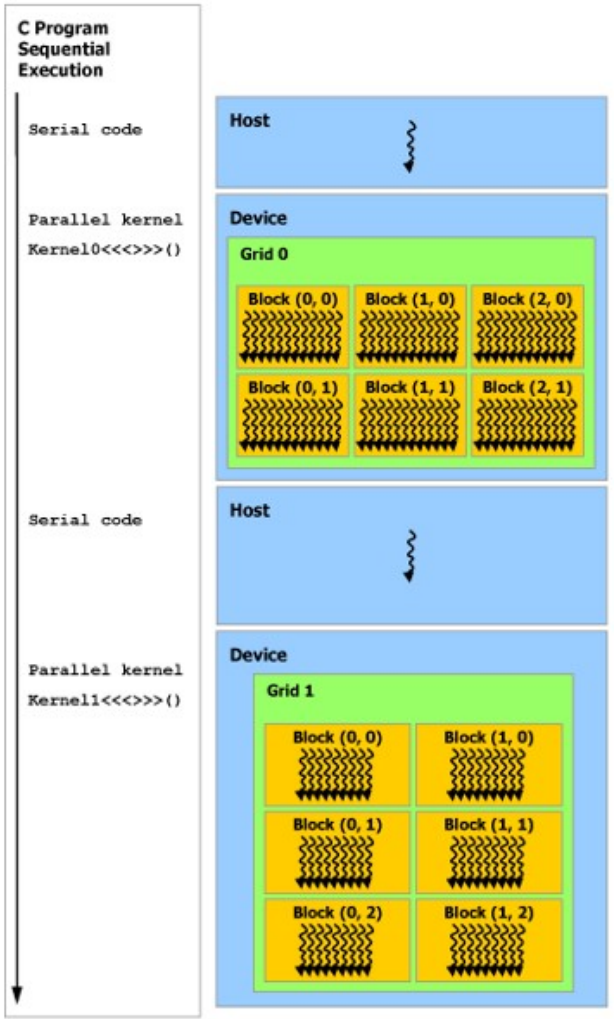
There are not many GPU designing and manufacturing companies, and even less that allow GPGPU programming. **Mainly two brands for this, Nvidia and AMD.**

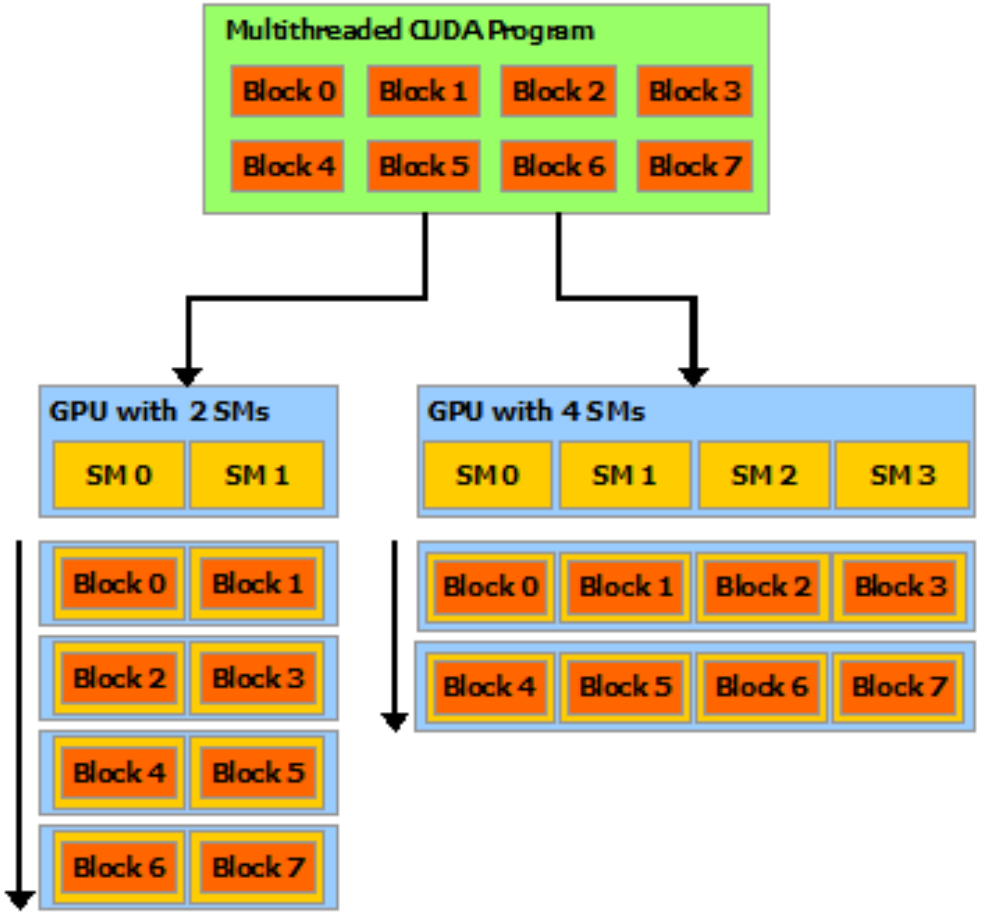
Programming on GPU usually require to use specially extended programming languages and or libraries to construct parallel kernels.

GPU programming is mostly dominated by the NVIDIA CUDA solution, however it is **limited to run only on Nvidia GPU.** Moreover, CUDA is not open source. **Nvidia is dominating the AI/ML domain with dedicated hardware and technologies.**

In contrast AMD has always worked on **open source GPGPU development tools** (OpenCL and now ROCm), and ensure that the produced code can run on a broader variety of hardware (see HIP).

CUDA programming model (SIMT)





Simple CUDA kernel example

```cpp
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

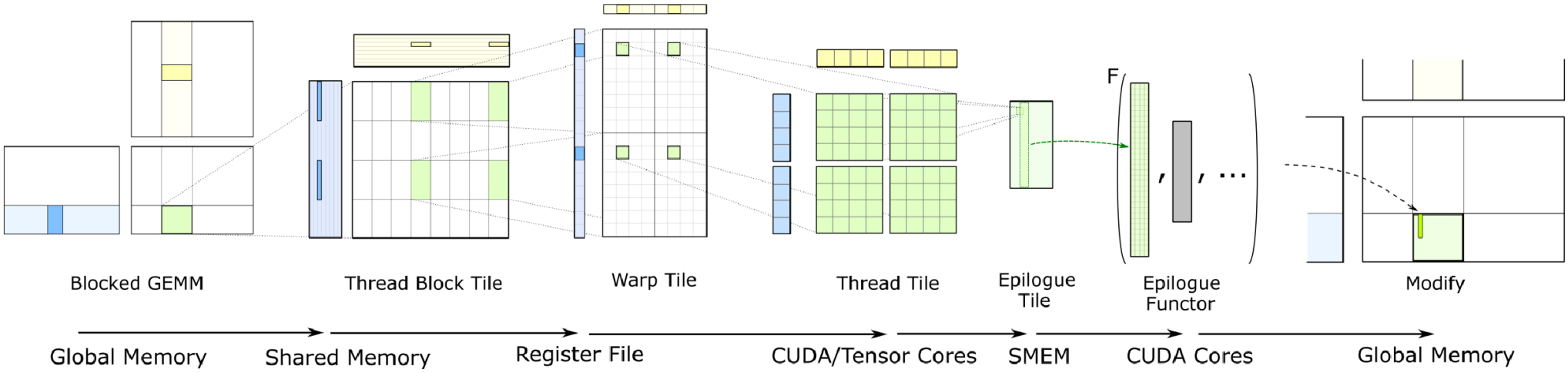
The GPU is controlled by the sequential main program, which is responsible for allocation the GPU memory, moving data to the GPU and launching kernels.

Inside a kernel, all the code is executed by each thread (SIMT). Specific variables are used to recover Ids that correspond to the position of the thread in the block and in the block grid.

With these basic elements it would be possible to design a **naive matrix multiplication kernel**. However, optimizing it to reach good performances is a much more tedious process and require more efforts for handling the memory hierarchy.

Like OpenBLAS, CUDA provide the **high level cuBLAS library** that allow to easily call **optimized matrix multiplication** operations that fully use the capabilities of Nvidia GPU.

Optimized GEMM construction for GPU!



GemmGlobalIteratorAb
Transformer
GemmSharedStoreTileAb

GemmSharedLoadTile{A,B}

fma, dp4a, hfma2
WMMA

Transformer
GemmSharedStoreTileD
GemmSharedLoadTileD

GemmGlobalIteratorC
Functor
GemmGlobalIteratorD

Global Load Stream

Shared Load Stream

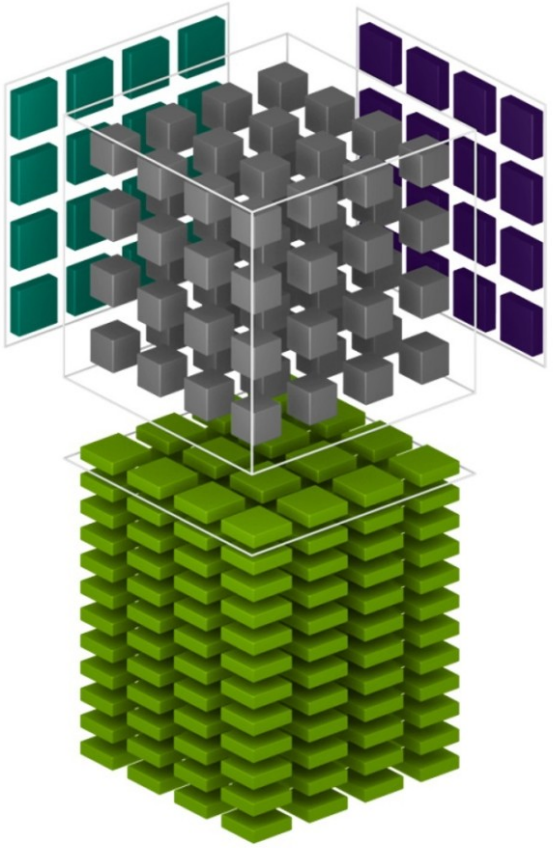
Matrix Multiply

Epilogue

From CuTLASS documentation

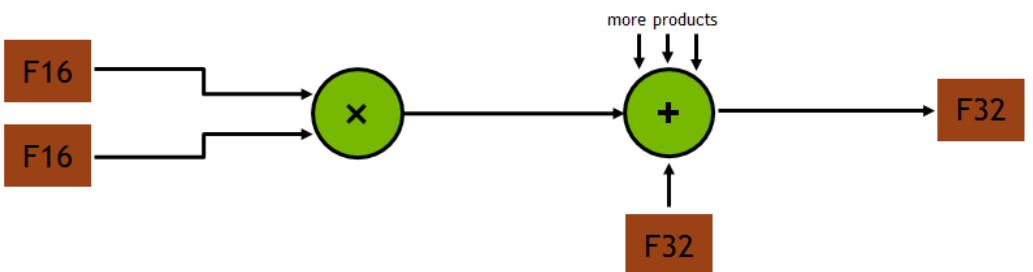
Nvidia Tensor Cores

Optimized Warp Matrix Multiply Add (WMMA) instructions !
CuBLAS can be set to used tensor core through the gemmEX function.



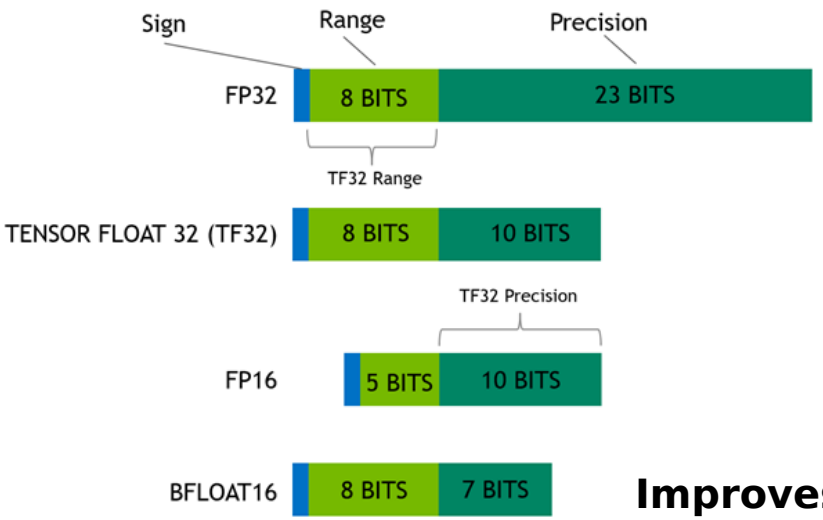
$$D = \underbrace{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}}_{\text{FP16 or FP32}} \underbrace{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}}_{\text{FP16}} + \underbrace{\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}}_{\text{FP16 or FP32}}$$

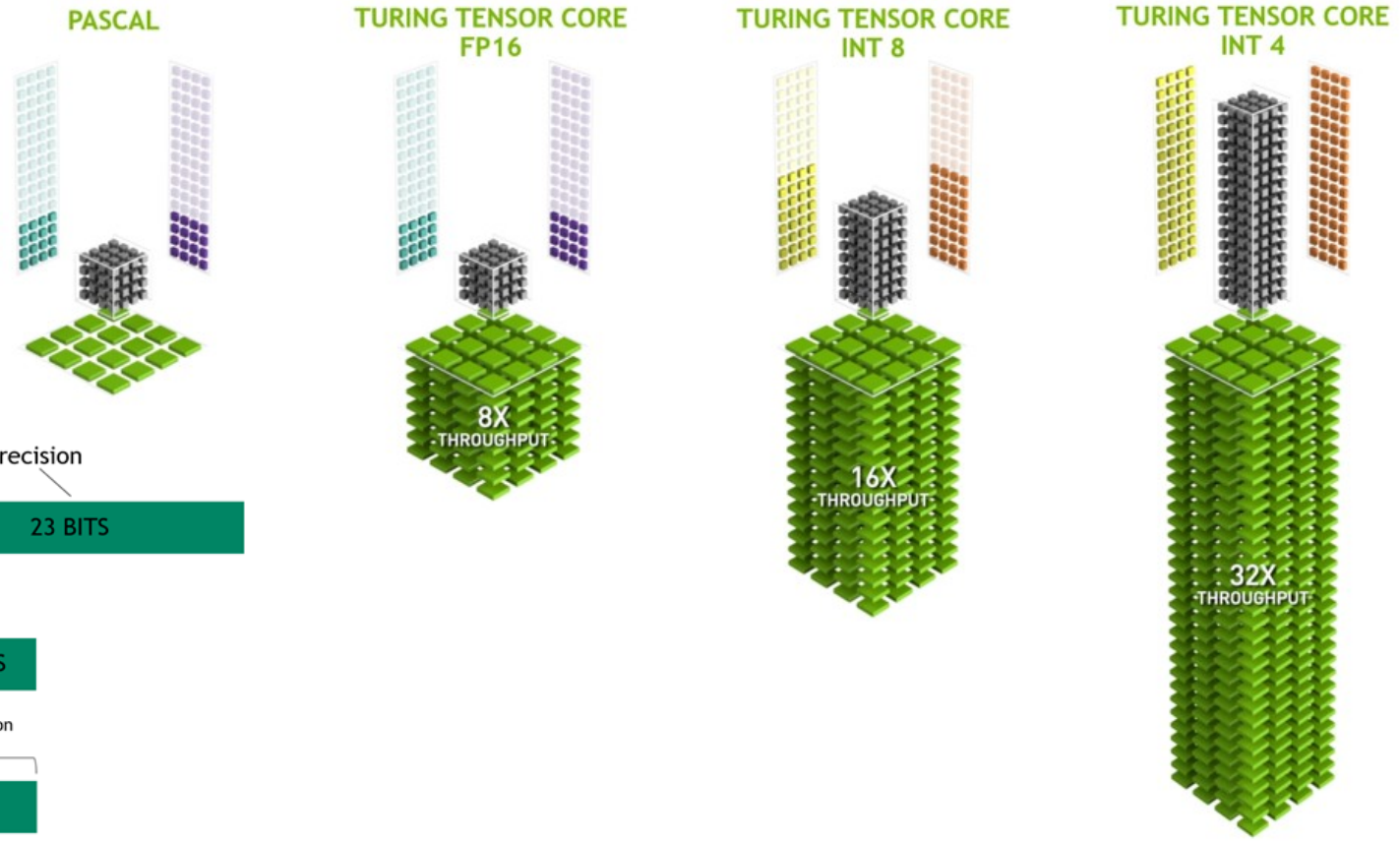
FP16 storage/input | **Full precision product** | **Sum with FP32 accumulator** | **Convert to FP32 result**



Nvidia Tensor Cores reduced quantization

$$1.2345 = \underbrace{12345}_{\text{mantissa}} \times \overbrace{10^{-4}}^{\text{exponent}}$$

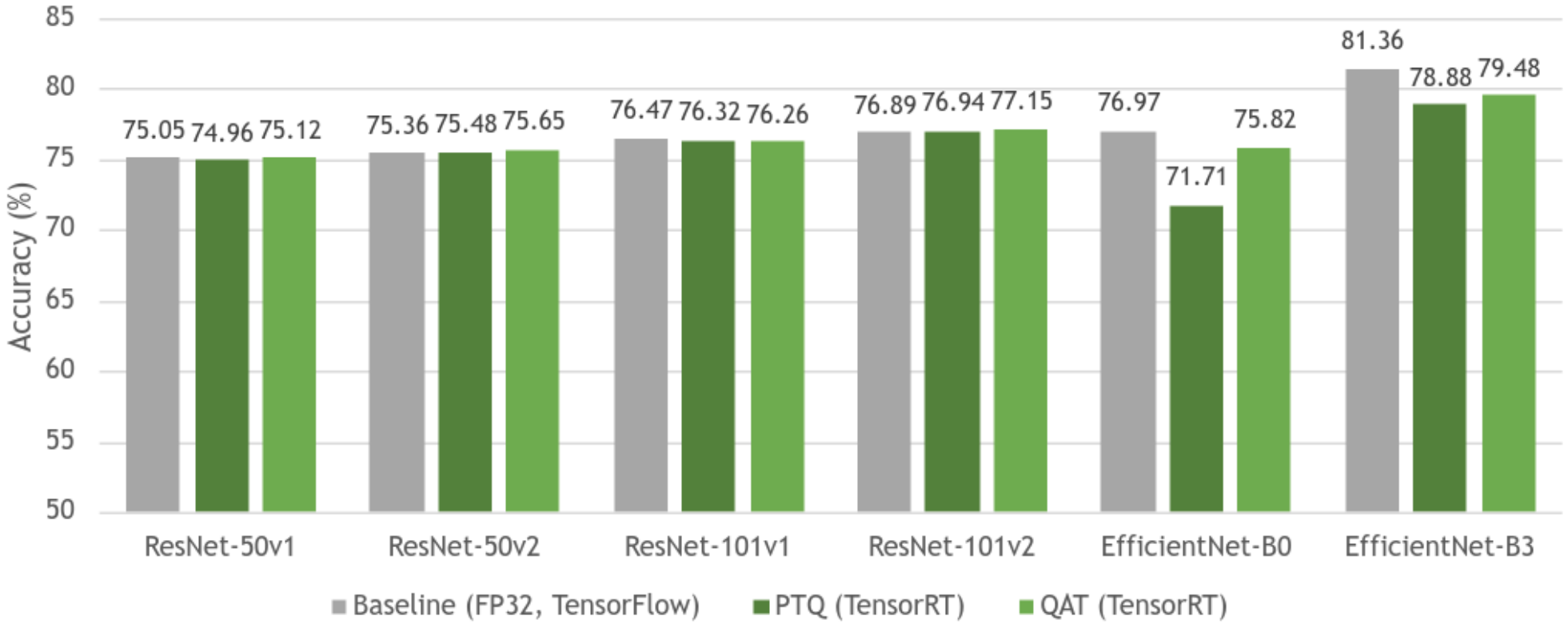




Improves compute throughput AND reduces data size,
→ reduce memory footprint and required bandwidth

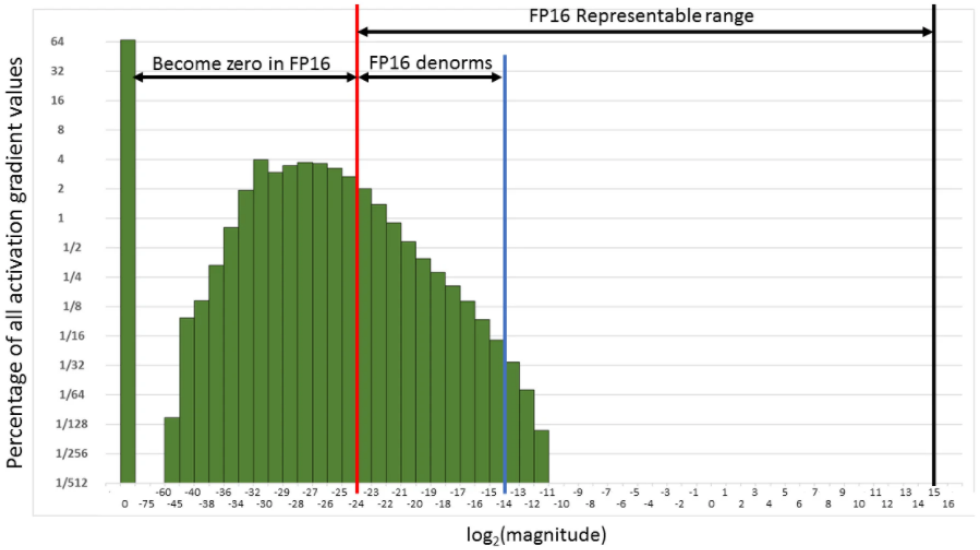
Quantization effect on AI model accuracy

All models quantized to INT8, accuracy for ImageNET-2012



PTQ = Post training quantization
QAT = Quantization aware training

Mixed precision training

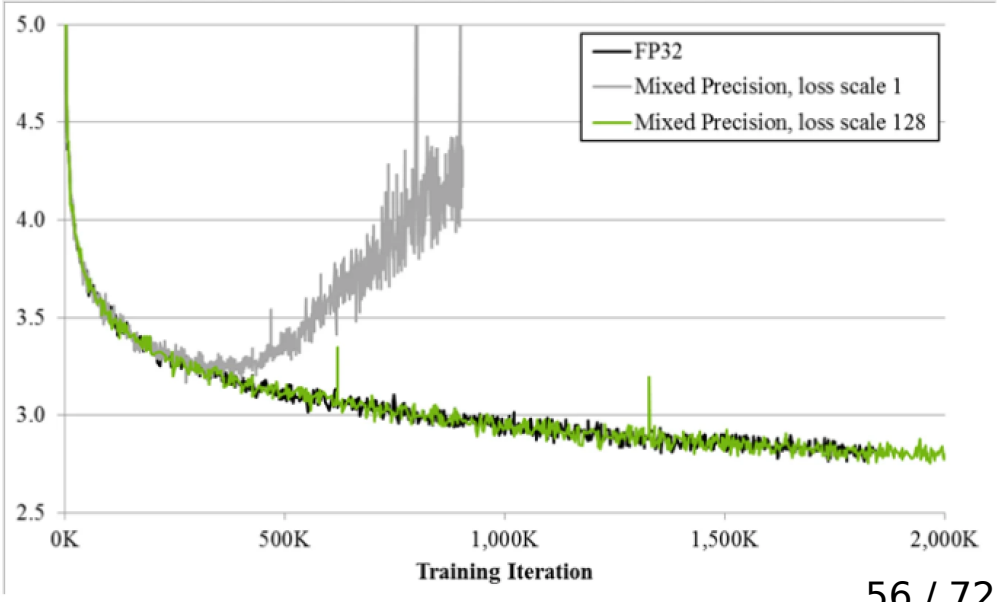


Reduced bit count variables have smaller representable ranges. This can lead to strong gradient vanishing problems.

This problem can first be mitigated by preserving an FP32 copy of the weights for accumulating the updates.

To further reduce this issue, a scaling is applied on the output loss. All propagated values are naturally scaled so they are more likely to be in the proper range.

At weight update time the correction is scaled down by the same factor.



From Nvidia « Mixed precision training »

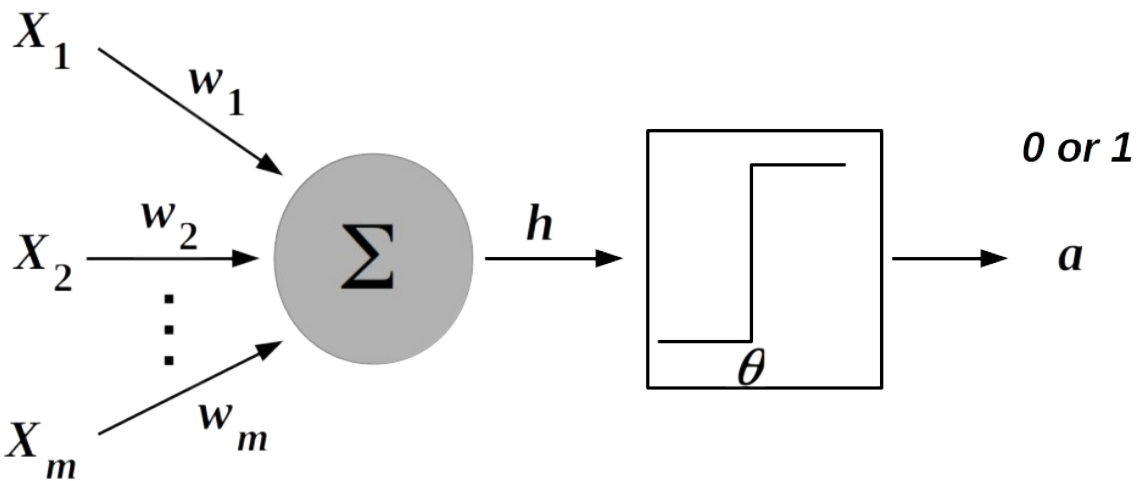
Introduction for PW part B

Single neuron operations

An single artificial neuron is characterized by vectorized operations on its input and weights.

It is also have some scalar functions like activation or loss.

However, **it must be applied on batches of data**, so the **input is in fact a 2D array.**



$$h = \sum_{i=1}^{m} X_i w_i$$

Weighted sum

$$a = g(h) = \begin{cases} 1 & \text{if} \quad h > \theta \\ 0 & \text{if} \quad h \leq \theta \end{cases}$$

Activation function

$$E = 0.5 \times (a - t)^2$$

Error/Loss function

$$\omega_i \leftarrow \omega_i - \eta \ (a - t) \ x_i$$

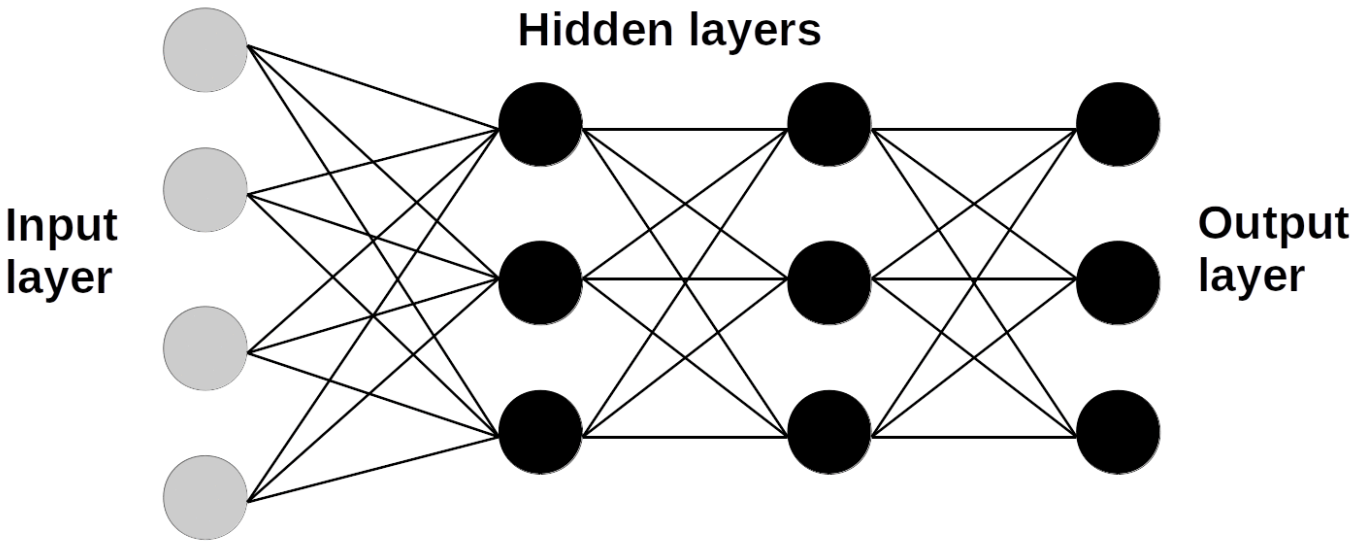
Update function

Dense network

In a « fully connected » MLP network, the weights becomes 2D arrays as well.

=> Computing the weighted sum for all input is now a matrix multiplication!

The network forward path is then a succession of matrix multiply operations.



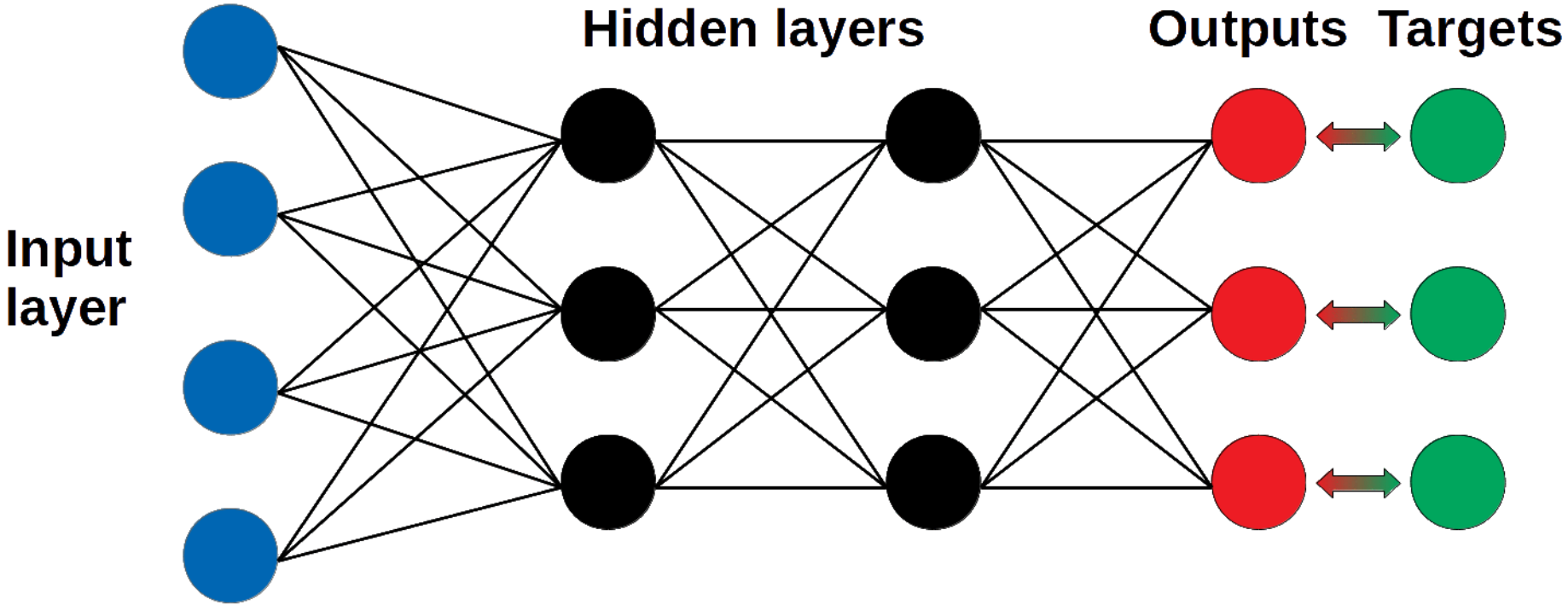
$$h_j = \sum_{i=1}^{m+1} x_i \omega_{ij}$$

$$a_j = g(h_j) = \begin{cases} 1 & \text{if} \quad h_j > \theta \\ 0 & \text{if} \quad h_j \leq \theta \end{cases}$$

$$\omega_{ij} \leftarrow \omega_{ij} - \eta \left(a_j - t_j \right) \times x_i$$

What about the error gradient back-propagation?

All operations are matrix multiplications as well !

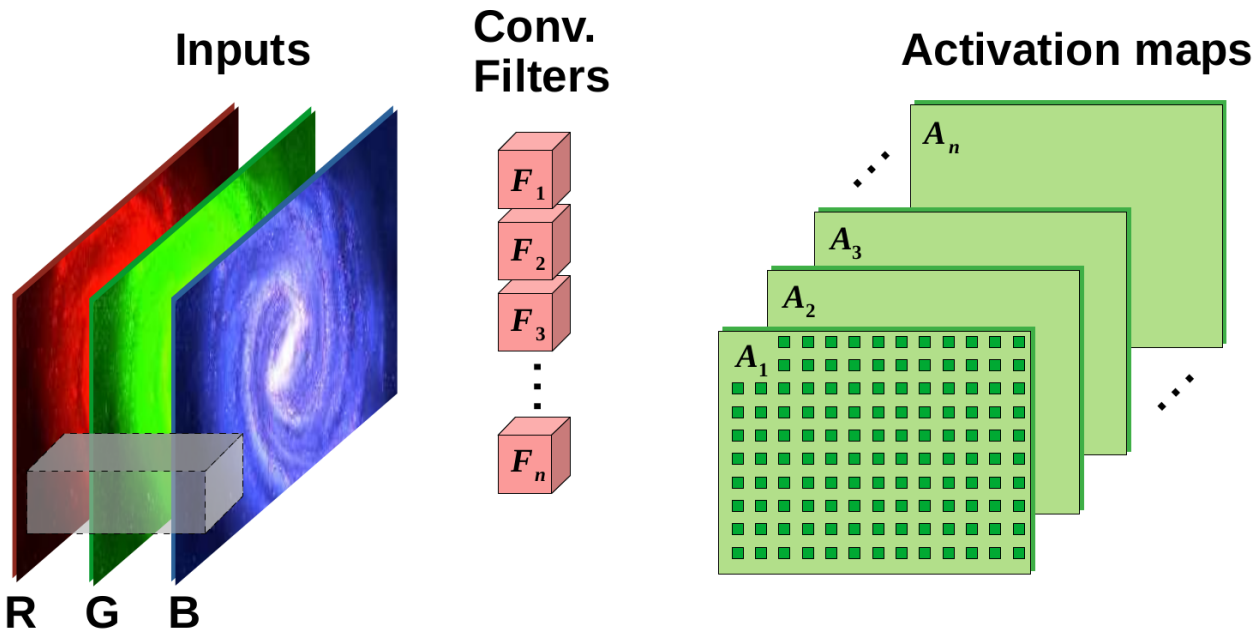


$$\omega_{ij} \leftarrow \omega_{ij} - \eta \frac{\partial E}{\partial \omega_{ij}} \quad \Rightarrow \quad \frac{\partial E}{\partial \omega_{ij}} = \delta_l(j) \frac{\partial h_j}{\partial \omega_{ij}} \quad \text{with} \quad \delta_l(j) \equiv \frac{\partial E}{\partial h_j} = \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial h_j} = \frac{\partial a_j}{\partial h_j} \sum_k \omega_{kj} \delta_{l+1}(k)$$

Single hidden layer MLP complete matrix formalism

From Cornu 2020
PhD thesis

Convolutional layer



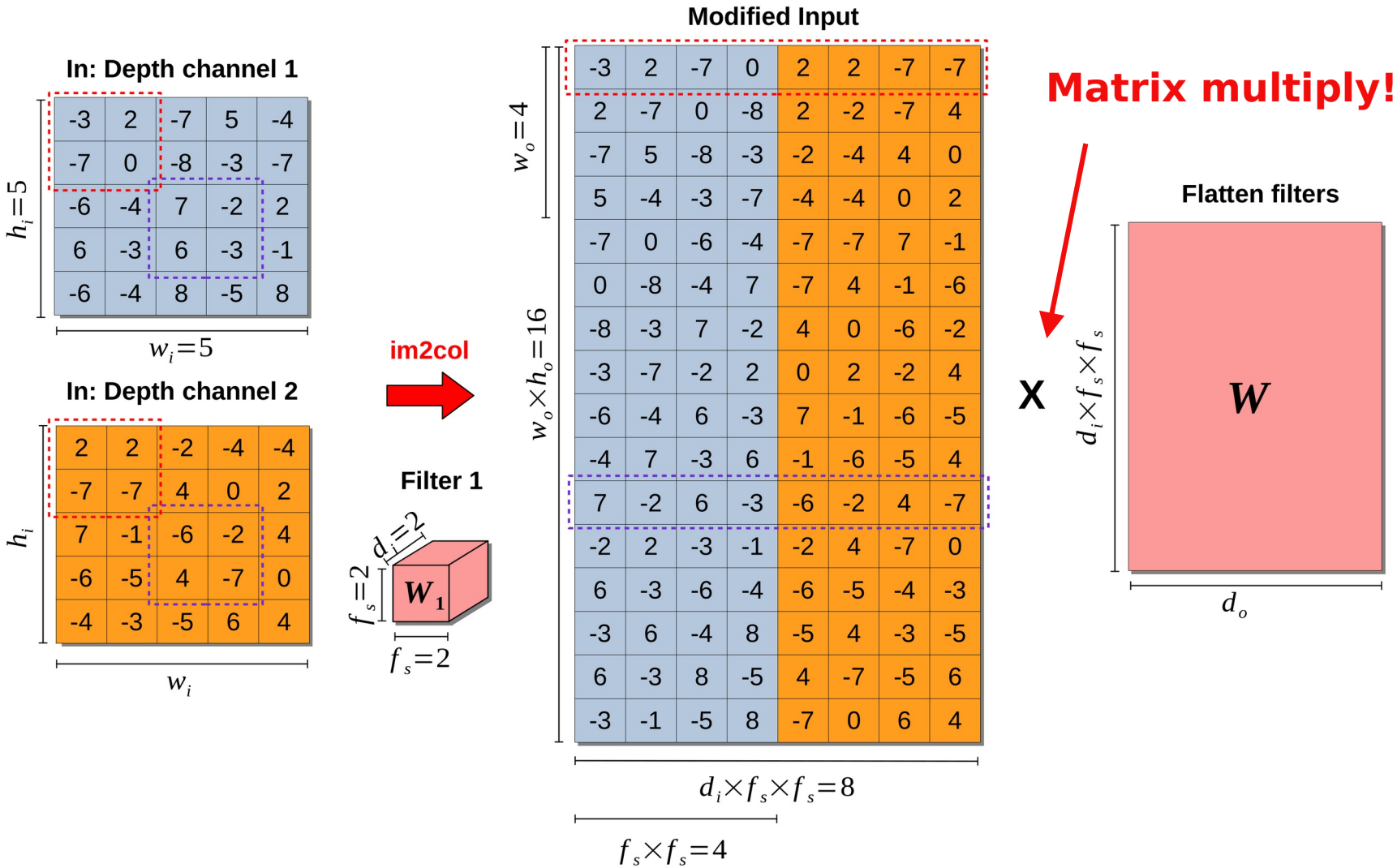
$$g\left(\sum_i X_i \circ W_i\right) = a$$

A convolution operation is not inherently a matrix operation.

However, it is equivalent to applying the weights of a single neuron to many possible positions.

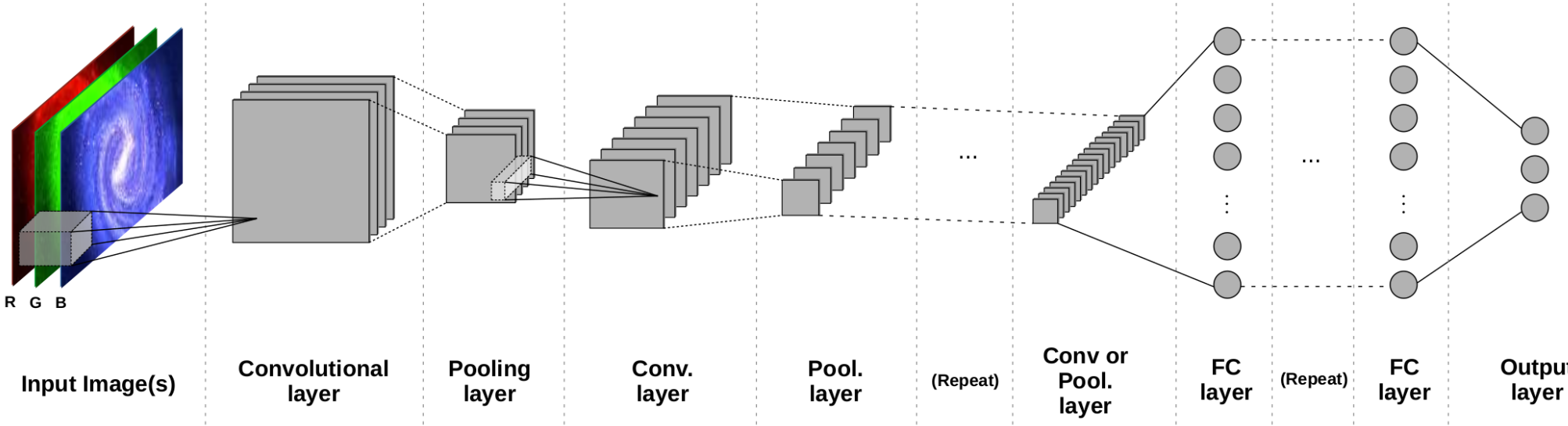
Therefore, it should be possible to **modify the input space** so the full convolution can be done through a single matrix multiply operation.

The Im2col transformation



Convolutional layer complete matrix formalism

Structural properties of ANN models



Network **expressivity** can be mostly improved in two ways

- Adding parameters (neurons, or filters) at the different layers
- Adding more layers → construct higher-level representations with less parameters, but also more difficult to train

However, the **predicted accuracy** can also be function of the **input resolution, the model quantization, or the training process ability to find the optimum.**

For a more detailed course on artificial neural networks you can have a look at
https://github.com/Deyht/ML_OSAE_M2

Computational cost

The **model size** in terms of parameters define the memory space it requires and is linked to the **amount of computation** but not in a linear way.

For a dense layer:

Forward matmul → $X[b, N_{l-1}] \times W[N_{l-1}, N_l]$

Backprop matmul → $\delta[b, N_l] \times W[N_l, N_{l-1}]$

Delta weights matmul → $X^T[N_{l-1}, b] \times \delta[b, N_l]$

For a convolutional layer:

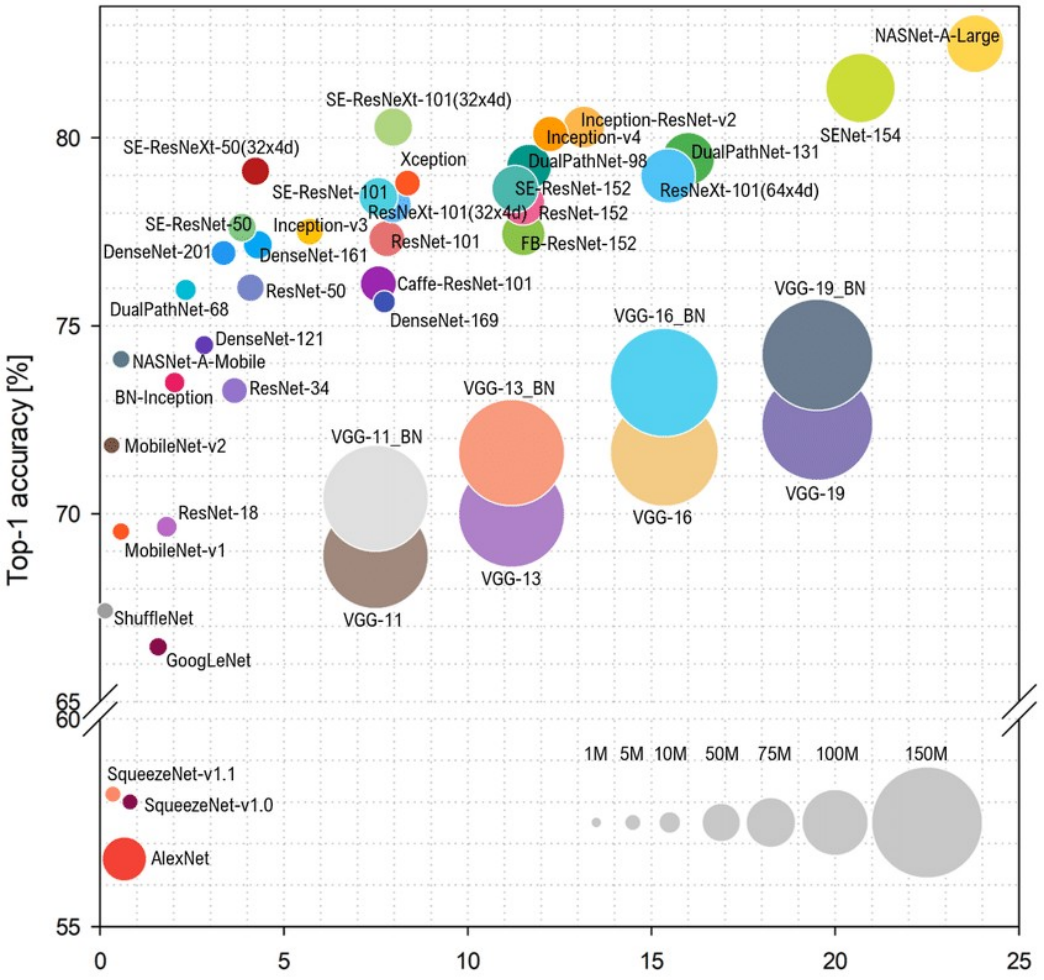
Number of convolution regions per dim. → $D_{out} = (D_{in} - f_s + 2P)/S + 1$

Forward matmul → $X_{im2col}[b.D_w.D_h, f_s.f_s.F_{l-1}] \times W[f_s.f_s.F_{l-1}, F_l]$

Backprop matmul → $\delta_{im2col}[b.D^i_w.D^i_h, f_s.f_s.F_l] \times W[f_s.f_s.F_l, F_{l-1}]$

Delta weights matmul → $X_{im2col}[f_s.f_s.F_{l-1}, b.D^i_w.D^i_h] \times \delta[b.D^i_w.D^i_h, F_l]$

Green AI vs Red AI



From Bianco et al. 2018

← Obtained form Image-Net 2012

There is a **clear relation between accuracy and computation cost and model size.**

Finding efficient models that reach high accuracy is usually possible but more difficult to design and train.

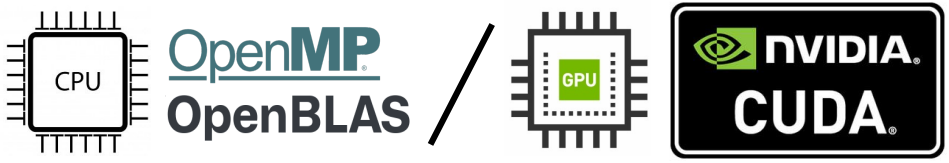
Exploring architectures and training procedures require computing time !!

- When optimizing a **single prediction**, the total impact is mostly dominated by training time. (e.g., simulation based inference in research). Therefore optimizing the architecture might be irrelevant, or counter-productive.
  → Problem: High accuracy result obtained this way encourage the use of sub-optimized model structures!
- When the model is planned to be **deployed** optimization is way more important as any % gained in efficiency scale with the size of the user base!

CIANNA

Convolutional Interactive Artificial Neural Networks by/for Astrophysicists

General purpose framework (like Keras, PyTorch, ...)
BUT developed for **astronomical applications**



x86 Intel/AMD & ARM — **6 dernière générations**

Open source – Apache 2 license

Custom YOLO implementation

Activation **Cost** **Association**

Full user interface

Successfully deployed on

- **Laptops / Workstation**
- **Local compute serveurs**
- **Mesocenteurs**
- **Large computing facilities**

And specificities:

- **Additional parameters per box**
- **Cascading loss**
- **Custom association process**
- **Custom NMS, and more ...**

Simple example

Simple modified LeNET-5 on MNIST → Images of 32x32, classification in 10 categories.

```
cnn.init(in_dim=i_ar([28,28]), in_nb_ch=1, out_dim=10,
        bias=0.1, b_size=16, comp_meth="C_CUDA", #Change to C_BLAS or C_NAIV
        dynamic_load=1, mixed_precision="FP32C_FP32A")

cnn.create_dataset("TRAIN", size=60000, input=data_train, target=target_train)
cnn.create_dataset("VALID", size=10000, input=data_valid, target=target_valid)
cnn.create_dataset("TEST" , size=10000, input=data_test, target=target_test)

#Python side datasets are not required anymore, they can be released to save RAM
#del (data_train, target_train, data_valid, target_valid, data_test, target_test)

#Used to load a saved network at a given iteration
load_step = 0
if(load_step > 0):
    cnn.load("net_save/net0_s%04d.dat"%(load_step), load_step)
else:
    cnn.conv(f_size=i_ar([5,5]), nb_filters=8 , padding=i_ar([2,2]), activation="RELU")
    cnn.pool(p_size=i_ar([2,2]), p_type="MAX")
    cnn.conv(f_size=i_ar([5,5]), nb_filters=16, padding=i_ar([2,2]), activation="RELU")
    cnn.pool(p_size=i_ar([2,2]), p_type="MAX")
    cnn.dense(nb_neurons=256, activation="RELU", drop_rate=0.5)
    cnn.dense(nb_neurons=128, activation="RELU", drop_rate=0.2)
    cnn.dense(nb_neurons=10, strict_size=1, activation="SMAX")


#To create a latex table and associated pdf with the current architecture
#cnn.print_arch_tex("./arch/", "arch", activation=1)

cnn.train(nb_iter=10, learning_rate=0.004, momentum=0.8, confmat=1, save_every=0)
cnn.perf_eval()
```

Simple example

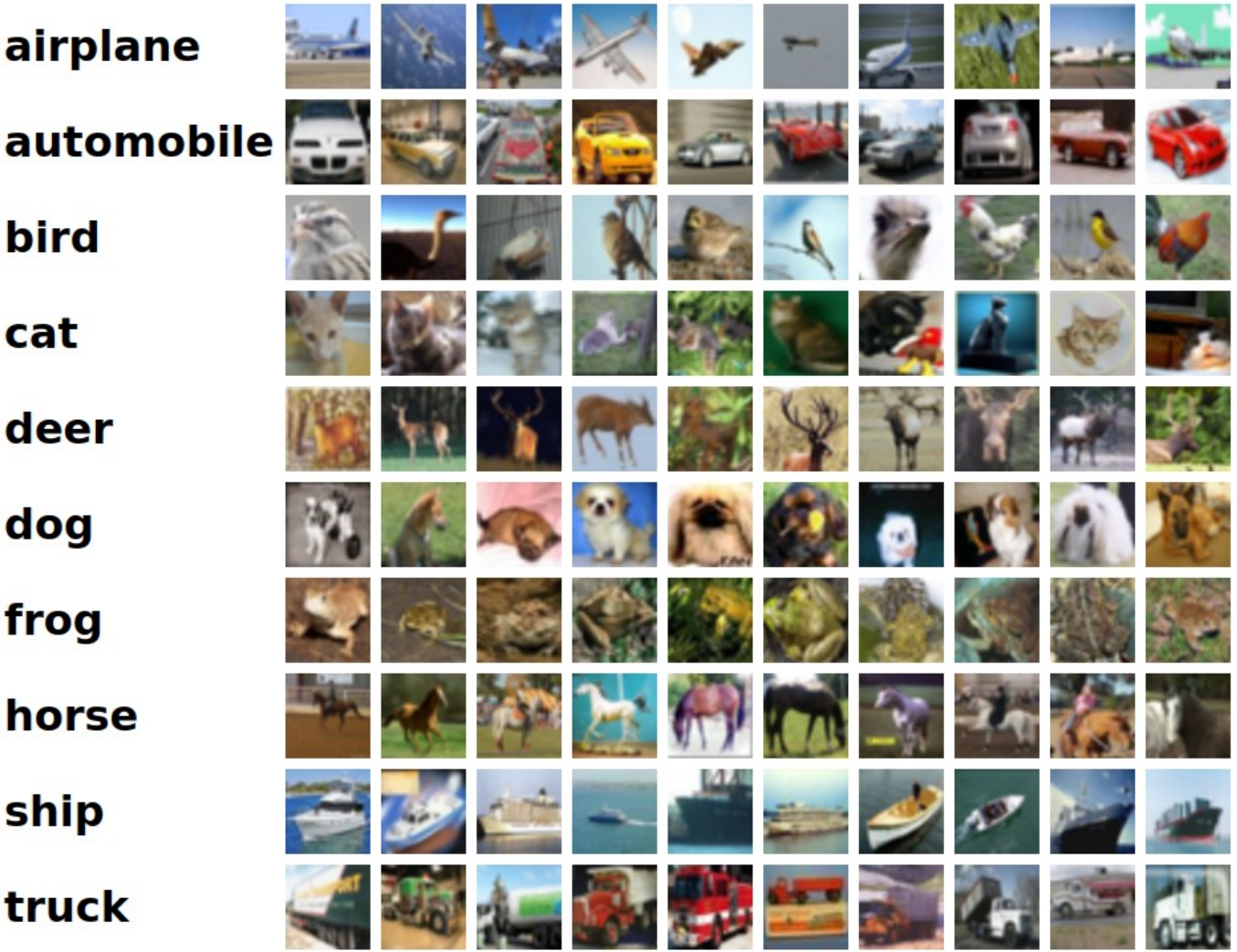
Simple modified LeNET-5 on MNIST → Images of 32x32, classification in 10 categories.

Layer N	Type T	Forward [µs] / [%]	Backprop [µs] / [%]	Cumulated [µs] / [%]
1	C	1.3 / 20.3	0.8 / 9.2	2.1 / 13.8
2	P	0.2 / 3.7	0.5 / 5.8	0.8 / 4.9
3	C	1.8 / 27.7	3.2 / 35.2	4.9 / 32.1
4	P	0.2 / 3.6	0.5 / 5.8	0.7 / 4.9
5	D	1.3 / 19.9	1.6 / 17.4	2.8 / 18.5
6	D	0.9 / 14.5	1.2 / 13.9	2.2 / 14.2
7	D	0.7 / 10.3	1.1 / 12.7	1.8 / 11.7
Total		6.4 µs	9.0 µs	15.3 µs

ConfMat

977	0	0	0	0	0	1	1	1	0	99.69%
0	1131	1	0	0	0	1	0	2	0	99.65%
1	1	1027	0	0	0	0	2	1	0	99.52%
0	0	0	1005	0	3	0	0	1	1	99.50%
0	0	0	0	977	0	1	0	1	3	99.49%
1	0	1	4	0	884	1	0	0	1	99.10%
3	2	0	0	1	2	950	0	0	0	99.16%
0	2	4	0	0	0	0	1020	1	1	99.22%
2	0	1	2	0	1	0	0	967	1	99.28%
1	1	0	1	6	3	0	1	1	995	98.61%
Prec. 99.19%	99.47%	99.32%	99.31%	99.29%	98.99%	99.58%	99.61%	99.18%	99.30%	Acc 99.33%

The CIFAR-10 dataset



60000 images of 32x32 pixels labeled into 10 classes.

50000 images for training and 10000 for testing.

Modern methods can reach 97% accuracy on this dataset.

Our reference LeNet-5 architecture reaches 75% accuracy.

Practical work: Optimize an efficiency weighted score

$$S = \left(\frac{E_r}{E}\right)^{w_E} \times \left(\frac{T_r}{T}\right)^{w_T} \times \left(\frac{P_r}{P}\right)^{w_P}$$

B	C	D	E	F	G	H
	Score parameters	Error rate (%)	Compute time (ms)	Nb. parameters		
	Ref. value	24,18	330	301821		
	Weight	1	0,95	0,1		
			(On a T4 through colab !)			
Model Name or ref.	**Participants (or team)**	**Error rate (%)**	**Compute time (ms)**	**Nb. parameters**	**Score**	**Inspiration references if any**
Ref. modififed Le-NET5 In 32x32, FP32 inference	Teaching team	24,18	330	301821	1	http://vision.stanford.edu/cs598_spring07/paper
Proposed correction arch. 1	Teaching team	19,72	150	53898	3,080836231	
					#DIV/0!	
					#DIV/0!	
					#DIV/0!	

The inference time must be estimated on Google Colab, using a T4 GPU.

More details in the practical work guide!