ETSN 2022

Le GPU : “la” technologie disruptive du 21ème siècle

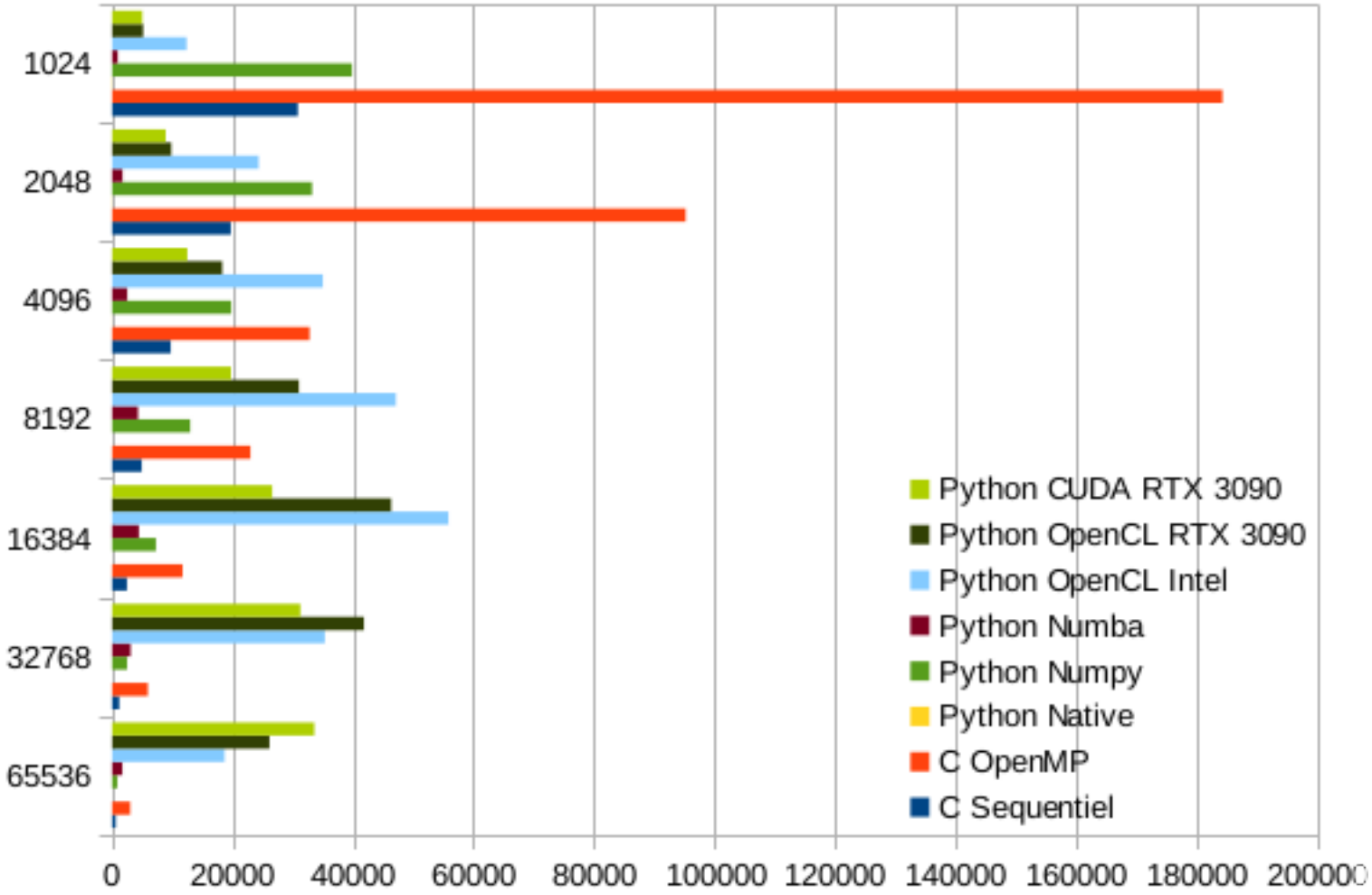
Des concepts de base du GPU aux performances comparées avec les *PU

Emmanuel Quémener

Les objectifs de cette découverte

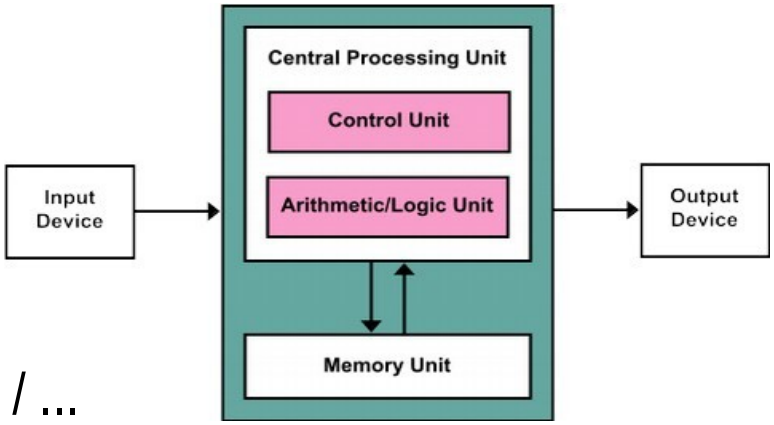
- Les GPU : des périphériques *Graphical Processors Units*
  - Puissants mais à appréhender « physiquement » en « logiquement »
- Python : un langage glue, modeste en performance
  - Mais efficace avec Numpy, Numba ou les modèles OpenCL et CUDA
- OpenCL : le langage *Open Computational Language*
  - Abrupte dans son apprentissage, universel et efficace sur ses destinations
- Les « codes matrices » : des socles
  - A exploiter pour explorer, mais aussi porter de vieux codes...

Appréhender le « cas d'usage »



Quelle vue du contexte ?
Point de vue d'un physicien...

- Approche « système » du physicien
  - Héritage des calculateurs analogiques
  - Le « système » informatique :
    - Réseau / matériel / OS / librairies / codes / usages / ...



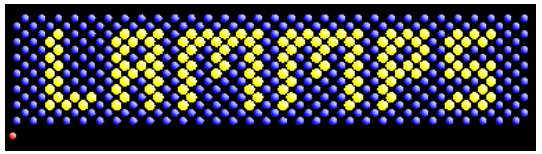
- Approche « Saint Thomas »
  - Apprentissage inductif par ma seule mesure
- Approche « pilote d'essai »
  - Caractérisation, recherche d'une exploitation optimale...

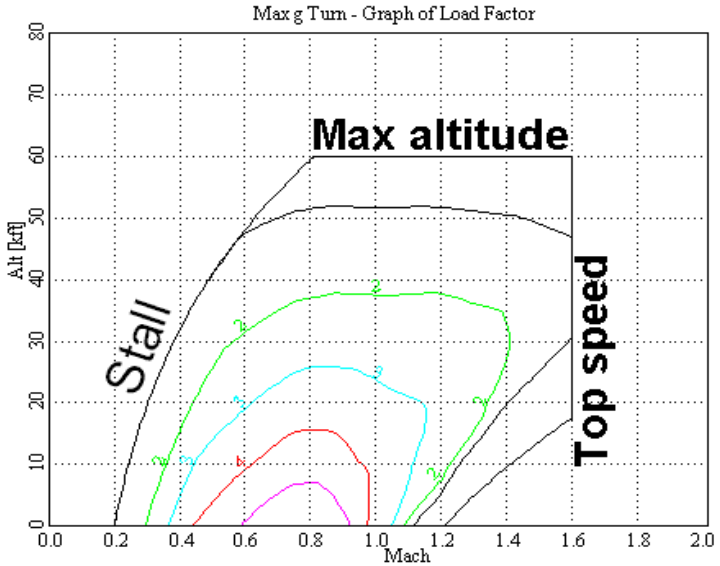
Centre Blaise Pascal ~ Dryden FR
Un petit exemple illustratif

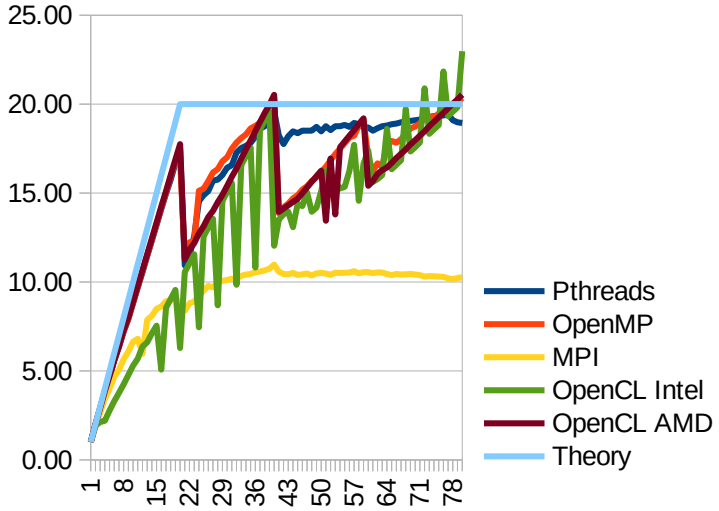
- Nasa X-29
- Cellule de F-5
- Moteur de F-18
- Train de F-16
- Etudes
  - Plans « canard »
  - Incidence >50°
  - « Fly-By-Wire »

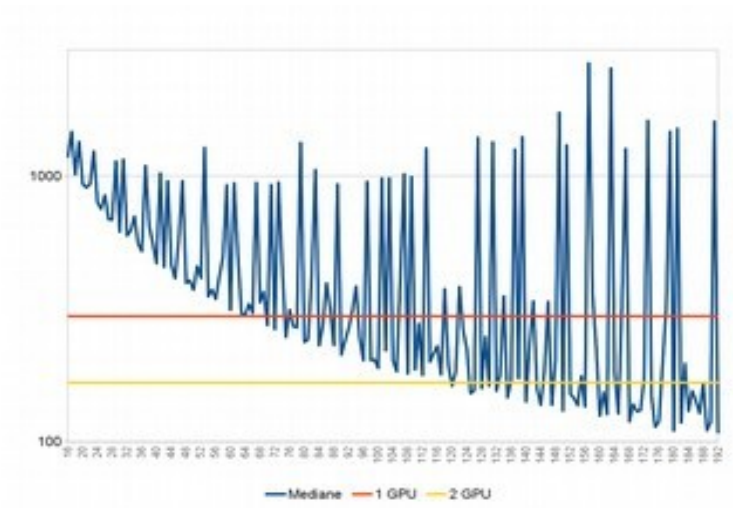
Recycler, réutiliser, explorer de nouveaux domaines...

Le but : de l'enveloppe de vol...
... aux enveloppes de parallélisme

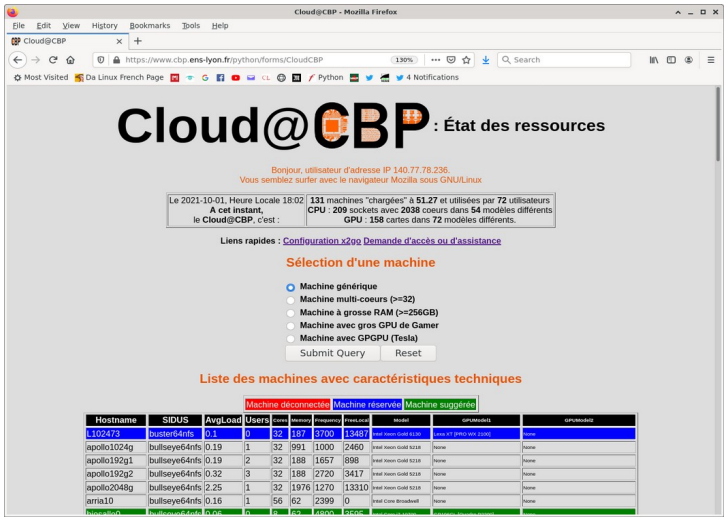


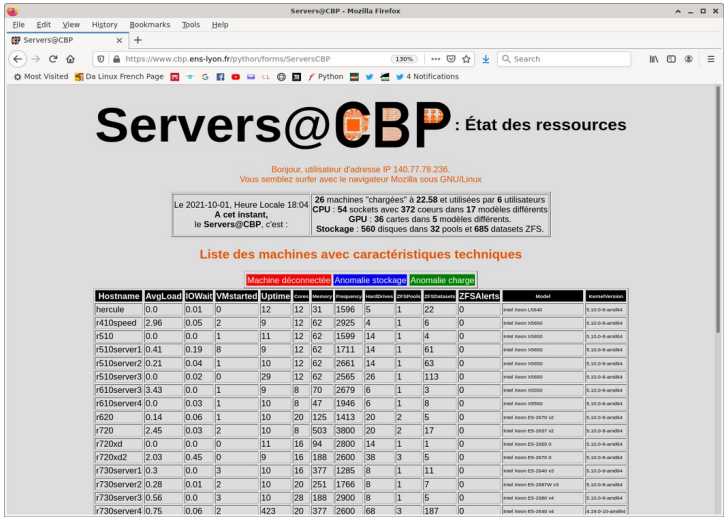


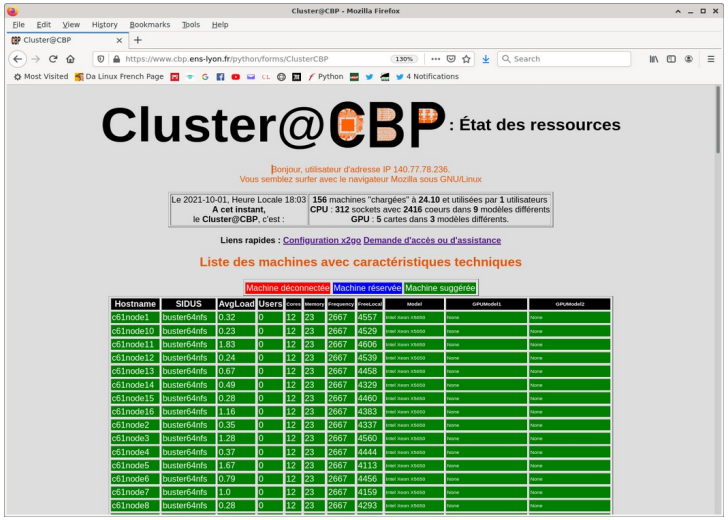




Le Centre Blaise Pascal : c'est aussi ... plus de 300 machines actives



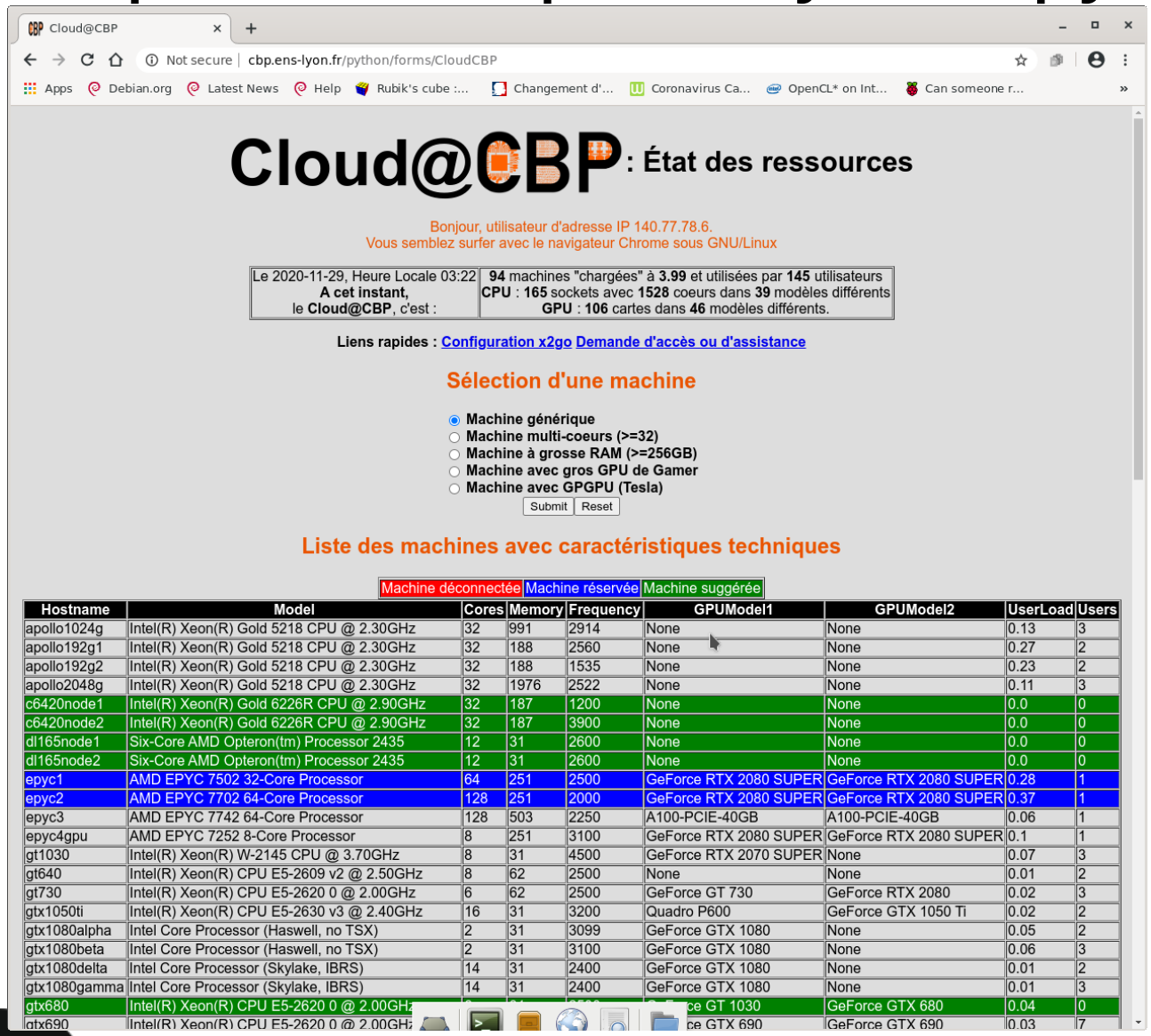




- +4600 (vrais) coeurs, 28 TiB RAM, +1200 HDD, ~4PiB
- Récupération : PSMN, DSI, labos internes & externes...
- +130 machines directement accessibles en graphique !
- Accès multiples : physique, SSH, VirtualGL, x2go...

Cloud@CBP : le point d'entrée

http://www.cbp.ens-lyon.fr/python/forms/CloudCBP



Hostname	Model	Cores	Memory	Frequency	GPUModel1	GPUModel2	UserLoad	Users
apollo1024g	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz	32	991	2914	None	None	0.13	3
apollo192g1	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz	32	188	2560	None	None	0.27	2
apollo192g2	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz	32	188	1535	None	None	0.23	2
apollo2048g	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz	32	1976	2522	None	None	0.11	3
c6420node1	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz	32	187	1200	None	None	0.0	0
c6420node2	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz	32	187	3900	None	None	0.0	0
dl165node1	Six-Core AMD Opteron(tm) Processor 2435	12	31	2600	None	None	0.0	0
dl165node2	Six-Core AMD Opteron(tm) Processor 2435	12	31	2600	None	None	0.0	0
epyc1	AMD EPYC 7502 32-Core Processor	64	251	2500	GeForce RTX 2080 SUPER	GeForce RTX 2080 SUPER	0.28	1
epyc2	AMD EPYC 7702 64-Core Processor	128	251	2000	GeForce RTX 2080 SUPER	GeForce RTX 2080 SUPER	0.37	1
epyc3	AMD EPYC 7742 64-Core Processor	128	503	2250	A100-PCIE-40GB	A100-PCIE-40GB	0.06	1
epyc4gpu	AMD EPYC 7252 8-Core Processor	8	251	3100	GeForce RTX 2080 SUPER	GeForce RTX 2080 SUPER	0.1	1
gt1030	Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz	8	31	4500	GeForce RTX 2070 SUPER	None	0.07	3
gt640	Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz	8	62	2500	None	None	0.01	2
gt730	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz	6	62	2500	GeForce GT 730	GeForce RTX 2080	0.02	3
gtx1050ti	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz	16	31	3200	Quadro P600	GeForce GTX 1050 Ti	0.02	2
gtx1080alpha	Intel Core Processor (Haswell, no TSX)	2	31	3099	GeForce GTX 1080	None	0.05	2
gtx1080beta	Intel Core Processor (Haswell, no TSX)	2	31	3100	GeForce GTX 1080	None	0.06	3
gtx1080delta	Intel Core Processor (Skylake, IBRS)	14	31	2400	GeForce GTX 1080	None	0.01	2
gtx1080gamma	Intel Core Processor (Skylake, IBRS)	14	31	2400	GeForce GTX 1080	None	0.01	3
gtx680	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz				ce GT 1030	GeForce GTX 680	0.04	0
gtx690	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz				ce GTX 690	GeForce GTX 690	0.03	7

- Etat général
  - Statistiques sommaires
- Composition
  - CPU, RAM, GPU, ...
  - Charge, utilisateurs, ...
- Liens utiles
  - Documentation x2go
  - Demande assistance
  - Sélection machine

Pour bien visualiser, ce qu'il faut : … ben une « bonne machine » !

- Dans le cloud@CBP, 5 types de machines :
  - Des postes de travail de salles de formations
  - Des stations de travail dans des bureaux ou dans la salle serveur
  - Des serveurs équipés d'équipement généralement spécifiques
  - Des machines « ouvertes » sans boitier
  - Des machines virtuelles à ressources dédiées
- La « bonne machine » :
  - C'est celle « adaptée » au « cas d'usage » : côté fonctionnel
  - C'est celle qui dispose de l'architecture spécifique : côté technique

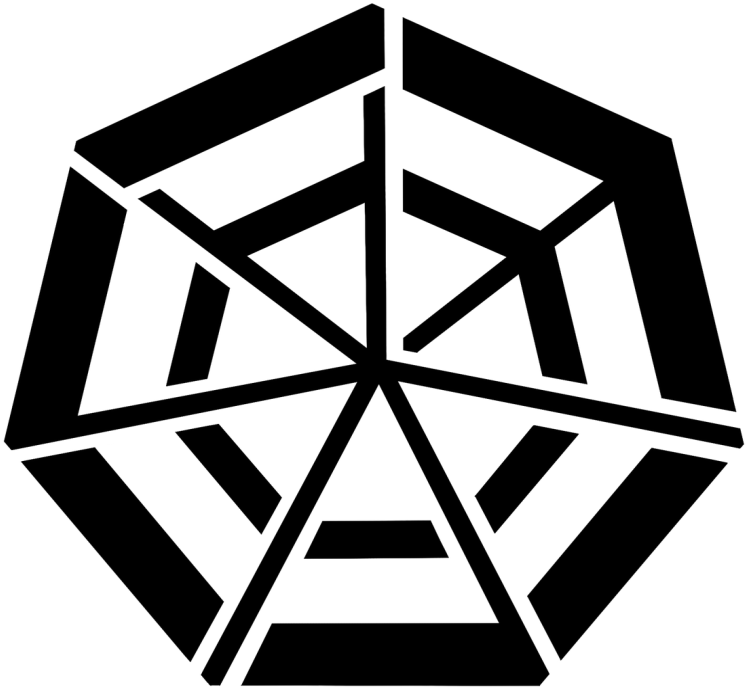
La « bonne machine » : la machine adaptée au « cas d'usage »

- Des composants : en reprenant Von Neumann
  - **Processeurs** : de 2 à 128 coeurs physiques, 55 modèles différents
  - **Mémoire vive** : de 16 GiB à 2 TiB
  - **Stockages réseau adaptés aux usages** : $HOME, projets, scratch
  - **Stockage local** : de 500 GB à 50 TB, HDD, SSD, NVMe
  - **GPU ou GPGPU : de 2008 à 2021 encore actifs,**
    - **de C1060 à A100, de HD7900 à RX6900XT, de 1 GB à 40 GB**
  - **Système unifié** : SIDUS (couramment Debian Bullseye)
- De la diversité (des équipements) vient la force :
  - Et du système unifié vient la maîtrise...

Sur les Machines du CBP : SIDUS

Je n'installe pas, je démarre !

- **Quoi ?**
  - Déployer un système simplement sur un parc de machines
- **Pourquoi ?**
  - Assurer l'unicité des configurations
  - Limiter l'empreinte du système sur les disques
- **Pour qui ?**
  - Étudiants (vous quoi!), enseignants, chercheurs, ingénieurs, ...
- **Quand & Où ?**
  - Centre Blaise Pascal : depuis 2010, plus de 280 machines aujourd'hui
  - PSMN : depuis 2011, plus de ~800 nœuds (sa propre instance) aujourd'hui
- **Comment ?**
  - Utiliser un partage en réseau d'une arborescence
  - Détourner le mécanisme de LiveCD



« Deux machines ayant démarré SIDUS ne peuvent pas ne pas avoir le même système ! »

Bureau à distance : le couple x2go/VirtualGL

- Pourquoi visualiser ?
  - Parce que la puissance d'analyse est derrière les yeux !
- Pourquoi visualiser à distance ?
  - Parce que l'accès physique n'est pas possible 7j/7, 24h/24
  - Parce que c'est plus pratique pour suivre les évolutions d'un job...
- Quelles contraintes de la visualisation à distance :
  - Sur un canal RDP : multi-plateforme, assez efficace, restreint en 3D
  - Sur un canal TeamViewer : très efficace, mais double tunnel…
  - Sur un canal SSH : lourd, passage OpenGL, multi-plateforme difficile

VirtualGL : efficace mais pénible...

- Shadering par la carte graphique
- Transport par canal SSH
- Utilisation :
  - Sous GNU/Linux, pas trop difficile :
    - Sur le poste client : vglconnect -s <MonLogin>@<MaVisu>
    - Sur le poste <MaVisu> : vglrun <MonAppli3D>
  - Sous les autres OS, bonne chance...

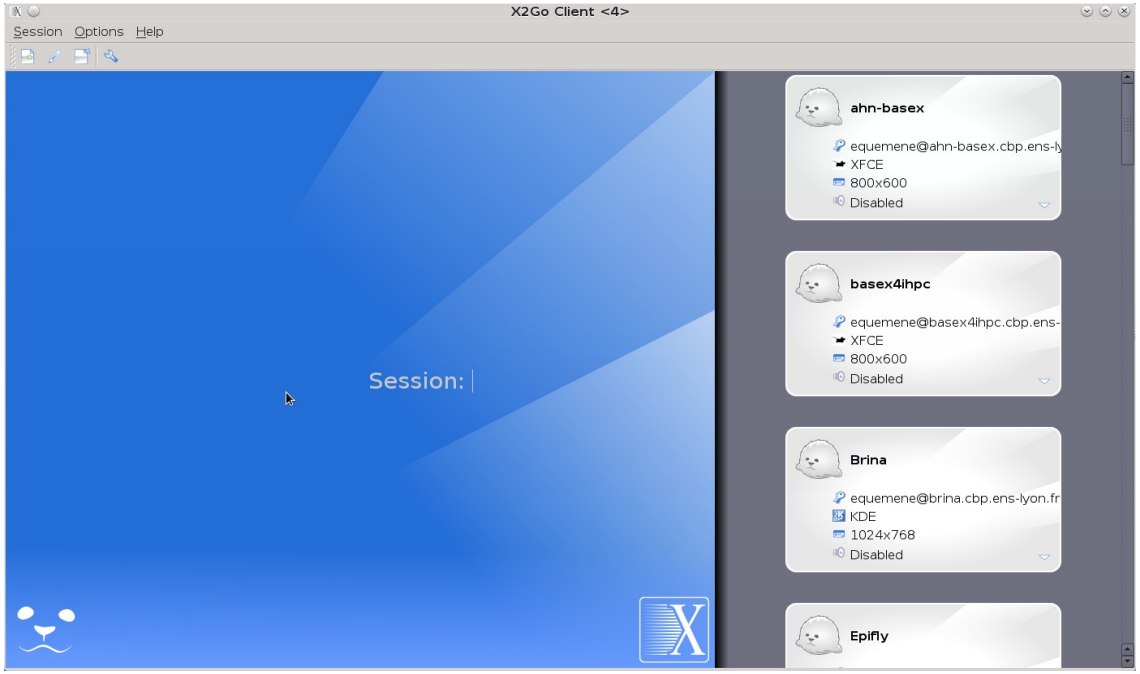
x2go : la transformation d'une session en vidéo

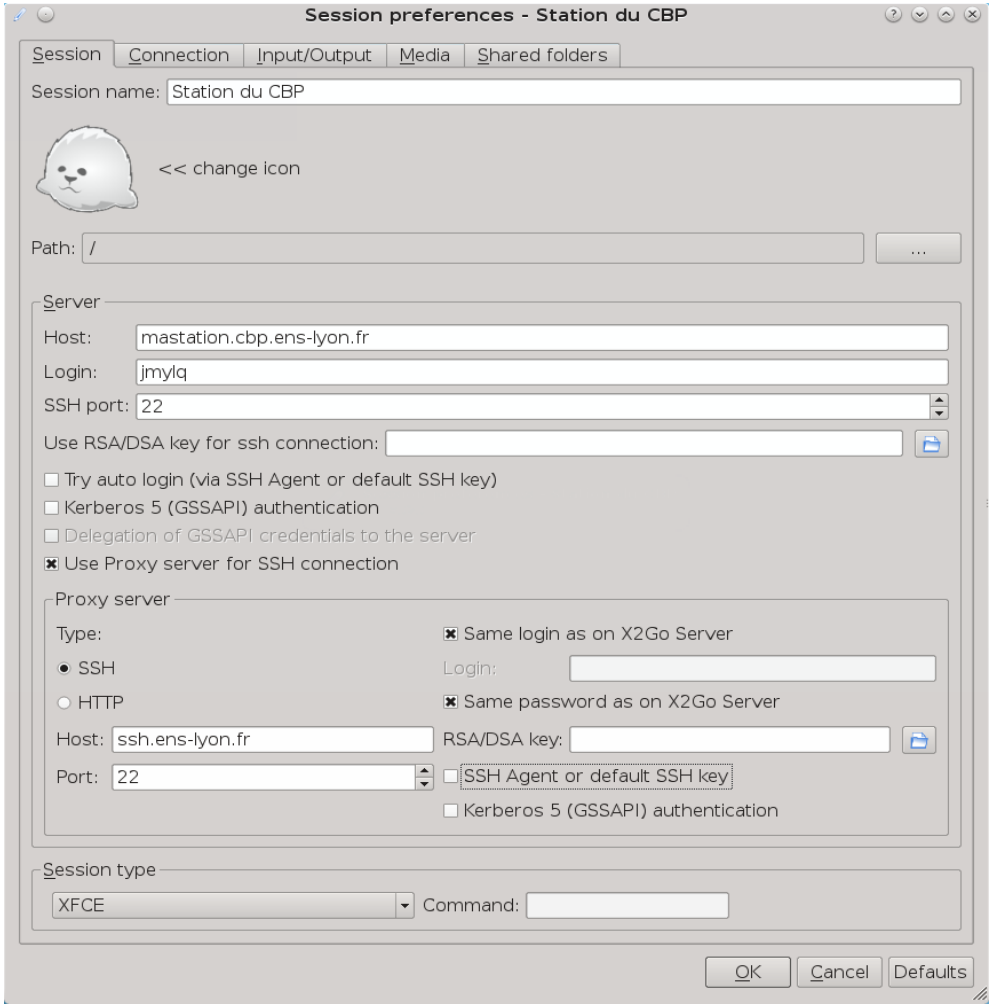
- 3 principes de base :
  - SSH, avec sa polyvalence, est le canal rêvé
  - Chaque image est transférée en jpg
  - Seule la modification de l'écran est transférée (comme MP4)
- Ses avantages :
  - Son multiplateforme : GNU/Linux, Windows, MacOSX, ChromeBook
  - Très faible bande passante utilisée : un 4G correct suffit
  - Passage de périphériques (son), espace de stockage, etc...
  - Exploitation de MesaGL pour la 3D, mais insuffisante...

x2go & VirtualGL : enfin un mariage qui marche !

- Utiliser x2go pour :
  - Son multi-plateforme : Windows, MacOSX, GNU/Linux, ChromeBook
  - Sa faible bande passante utilisée (300KB/s pour une vidéo HD)
  - Son passage de périphériques :
    - Même la carte son peut passer !
    - Mais pour MacOSX, inférieur à 10.5 pour passer les périphériques
- Utiliser VirtualGL pour :
  - Les grosses applications 3D & GPU (Cuda & OpenCL) : MorphoGraphX, Paraview, VMD, ...
  - Préfixer « juste » la commande par vglrun
- Comme ça :
  - Transfert des données primaires plus accessibles
  - Exploitation plus rationnelle des stations de travail

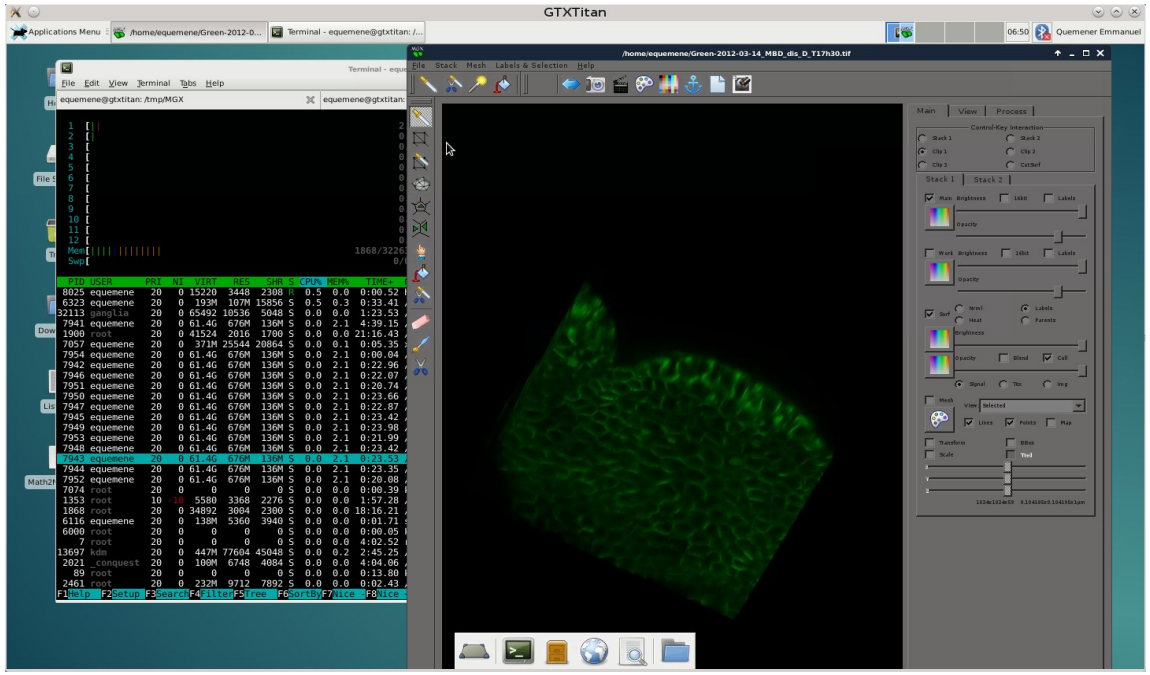
Connexion par x2go
Une configuration aisée...



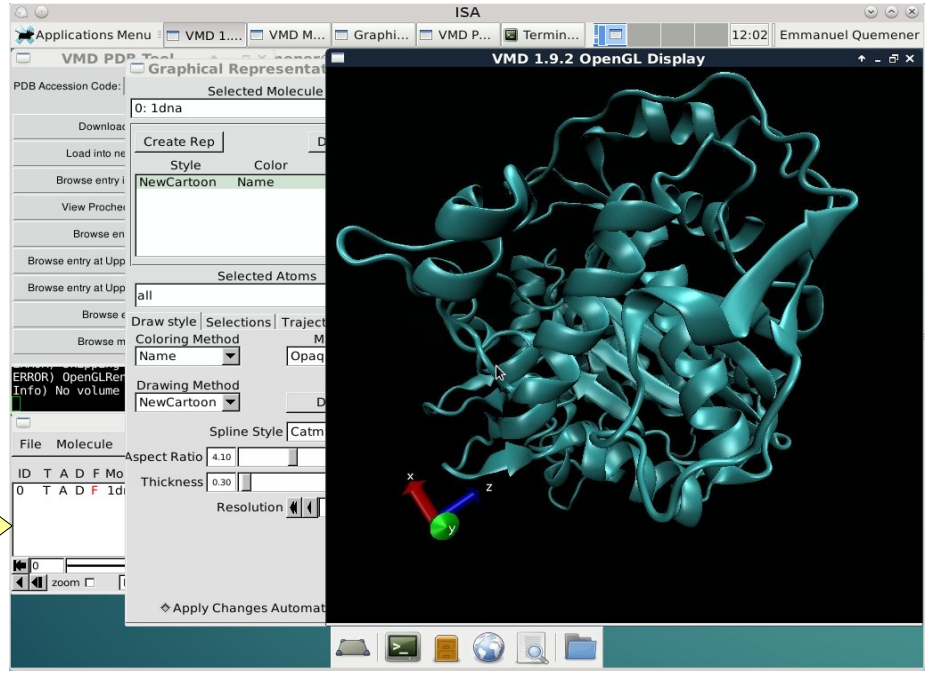


Le plus dur, c'est de suivre **scrupuleusement** la doc :-/

Exploitation x2go/VirtualGL
Pas seulement la visualisation...



Traitement avec MorphoGraphX



Visualisation avec VMD

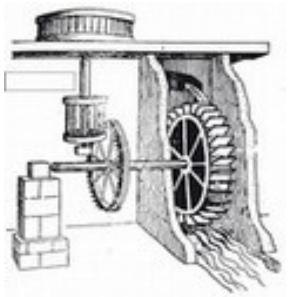
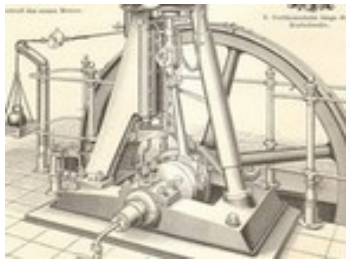
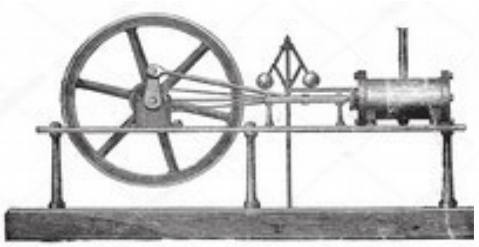
Une petite « démo » ?

De la démarche scientifique…
… À l'expérience numérique

Quelques définitions & sigles...

- ALU : Arithmetic & Logic Unit, Unité Arithmétique et Logique
- CPU : Central Processing Unit, Unité de traitement Centrale,
- Flops : Floating Point Operations Per Second, Opérations flottantes par seconde
- (GP)GPU : (General Purpose) Graphical Processing Unit, Circuit graphique
- MPI : Message Passing Interface, Interface de communication par messages
- RAM : Random Access Memory, Mémoire à accès aléatoire
- SMP : Shared Memory Processors, Processeurs à mémoire partagée
- TDP : Thermal Design Power, Enveloppe thermique
- Et quelques autres :
  - **PR** : Parallel Rate, taux de parallélisme (NP en MPI, Threads en OpenMP, Blocks, WorkItems en GPU)
  - **Itops** : Iterative Operations Per Second
  - **QPU** : Quantum Processing Unit (program exécuté avec PR=1)
  - **EPU** : Equivalent Processing Unit (PR déduit de l'optimal d'exécution du programme parallèle)

Travaux physique ou intellectuel
Moteurs & Ordinateurs :
Des auxiliaires de « puissance »

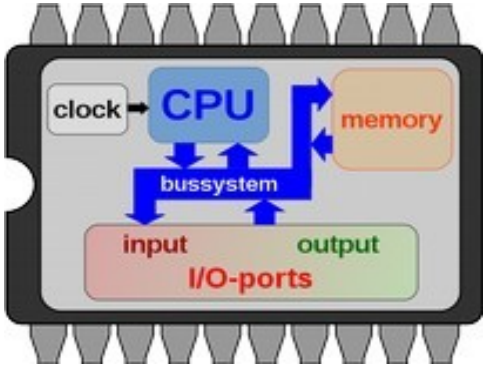






Des anciens temps aux temps modernes...

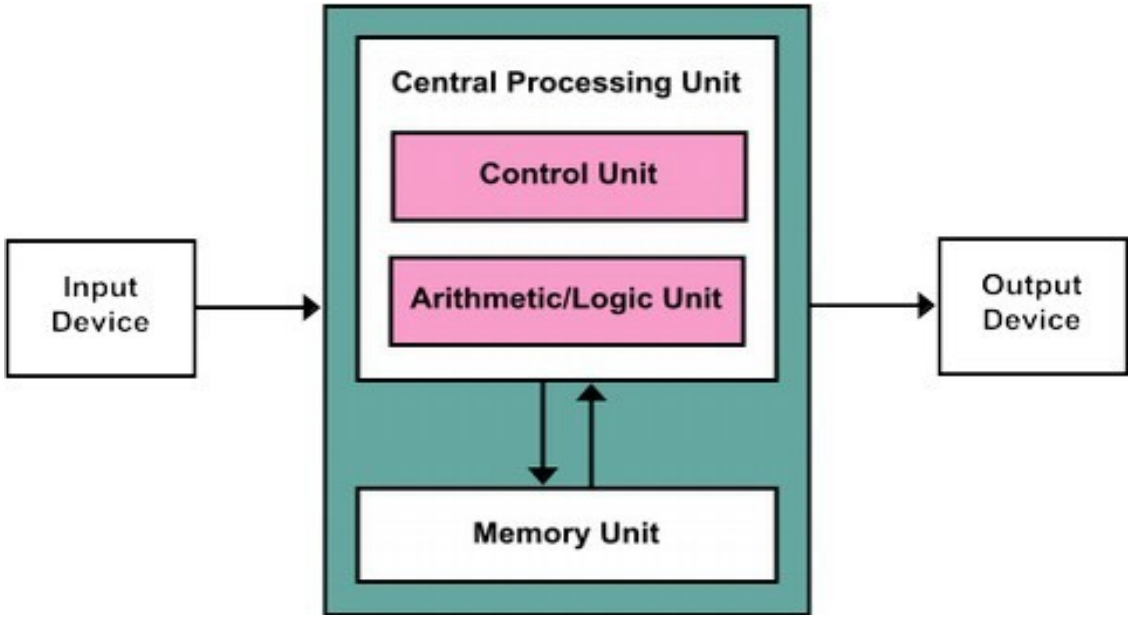

Machine
D'Anticythère
87 av JC


Pascaline
De Blaise Pascal
1642

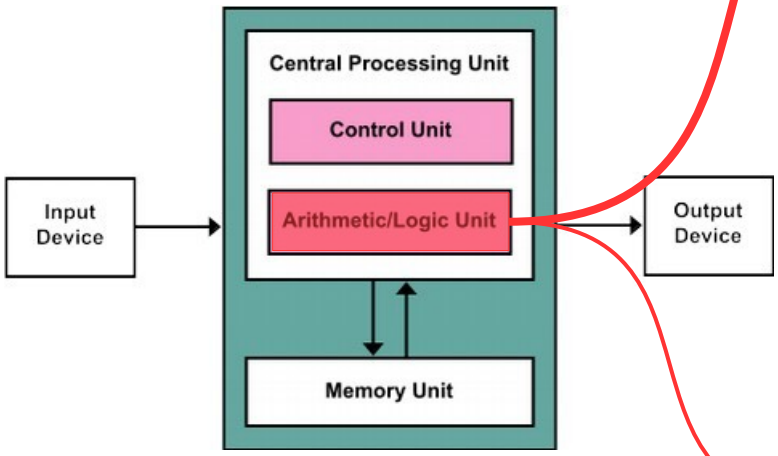

De Alan Turing
1943

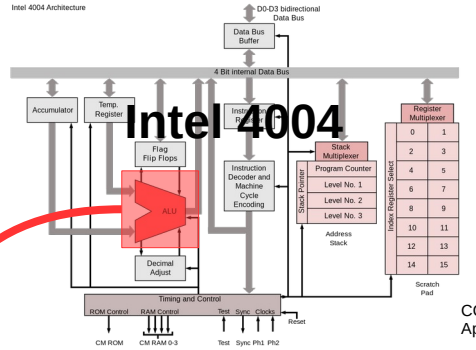


Un computeur : un métier à tisser ?
Architecture de Von Neumann

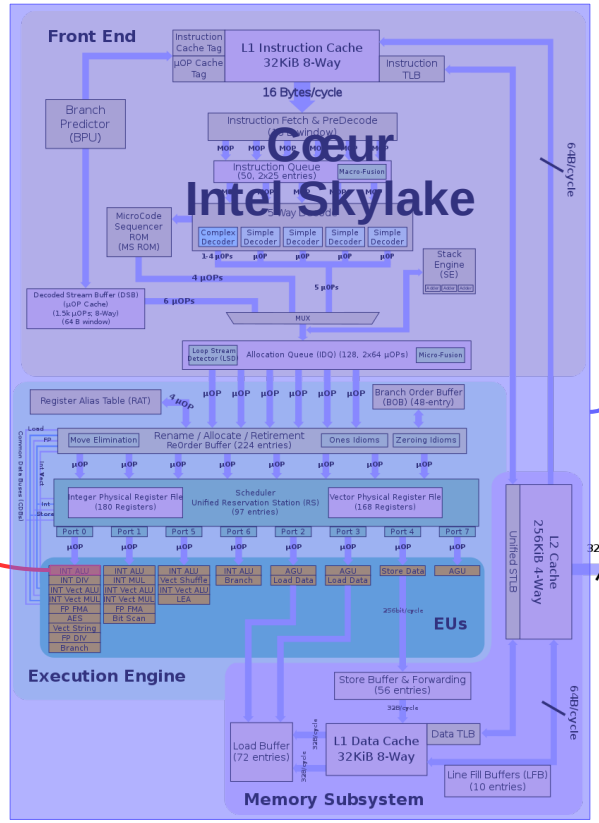


Où est le GPU ?
Modèle de Von Newmann...

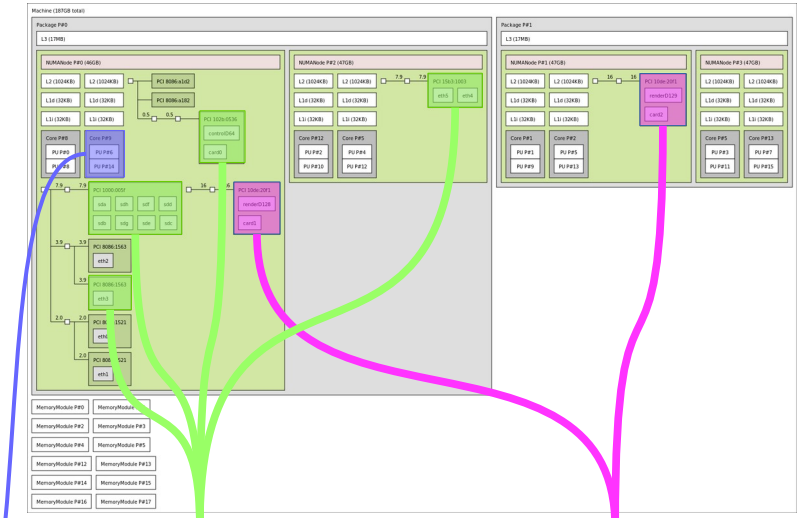




CC BY-SA
Appaloosa

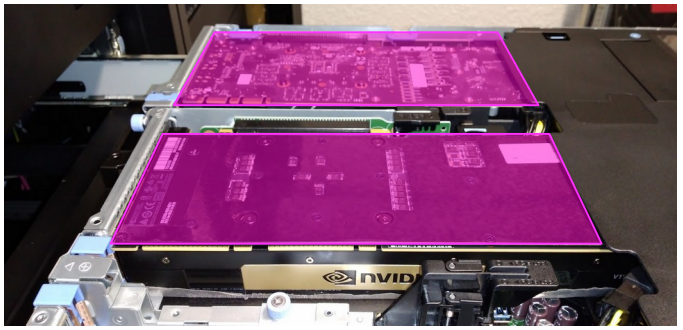


Cœur
Intel Skylake

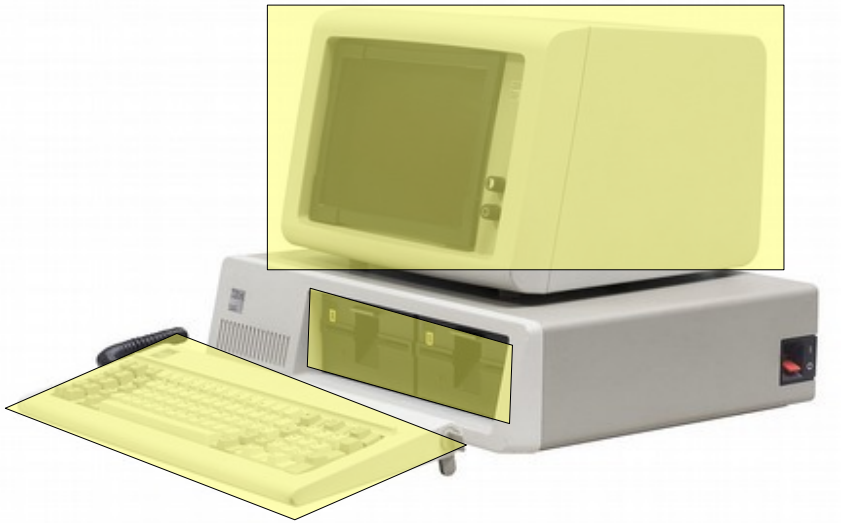


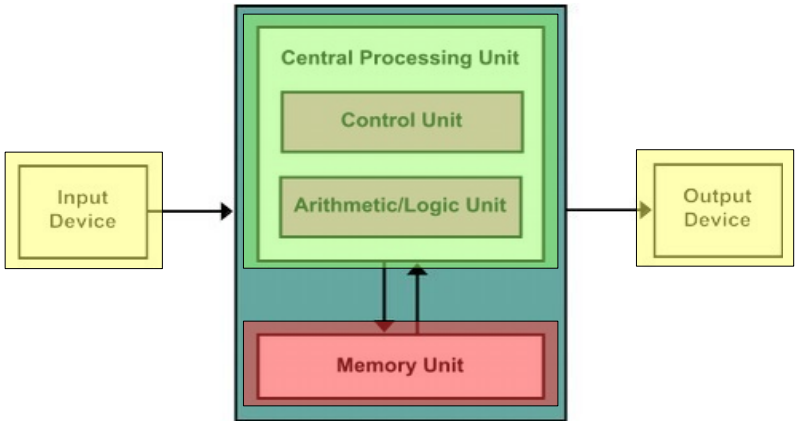
Input
/
Output

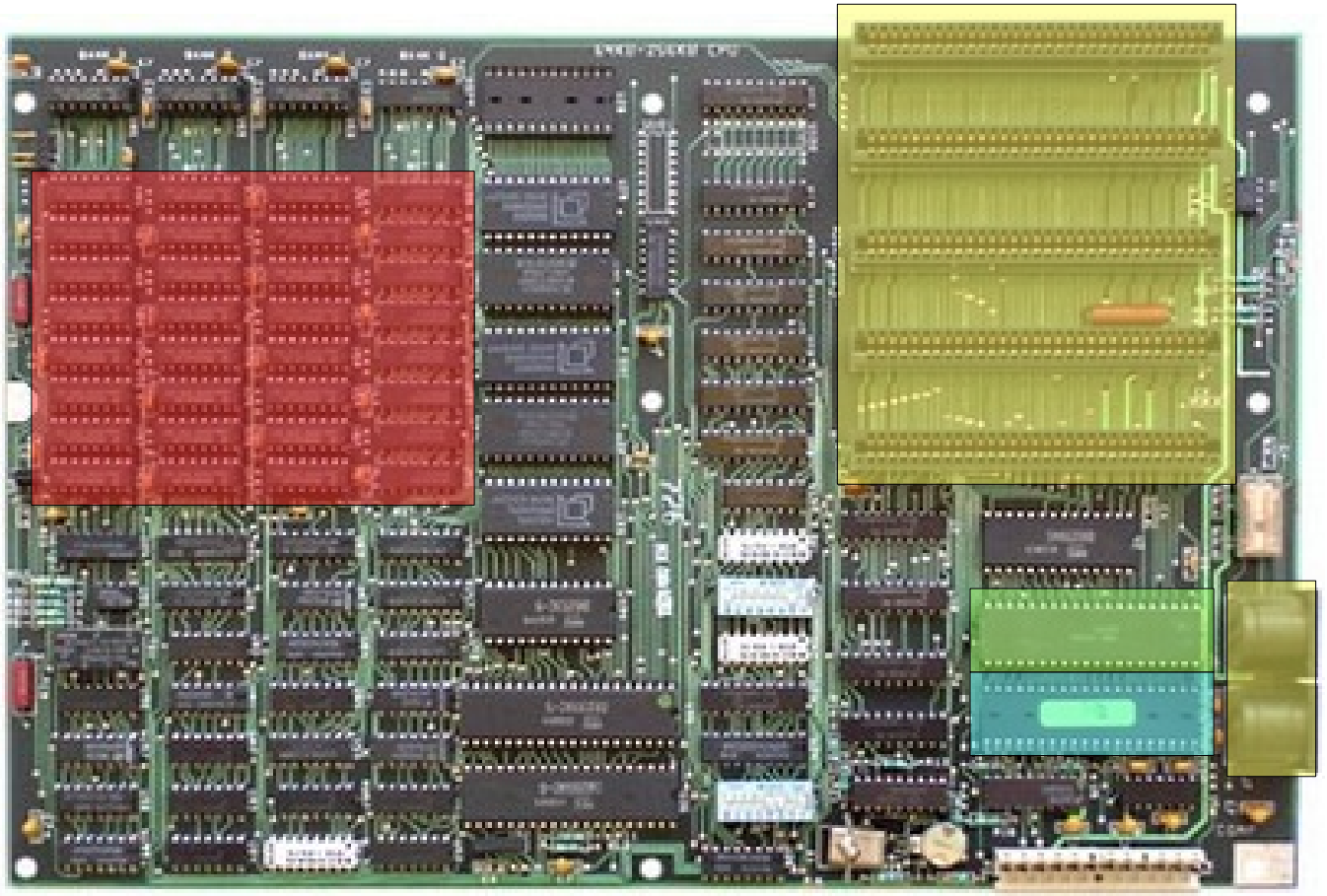
GPU
Nvidia
V100



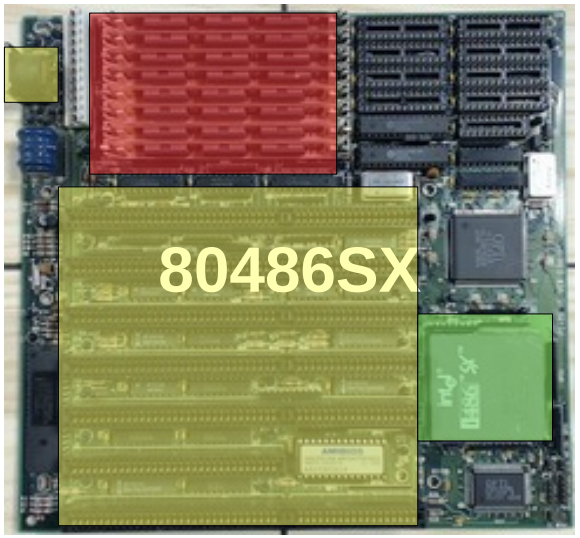
Une révolution informatique
IBM Personal Computer, le « PC »

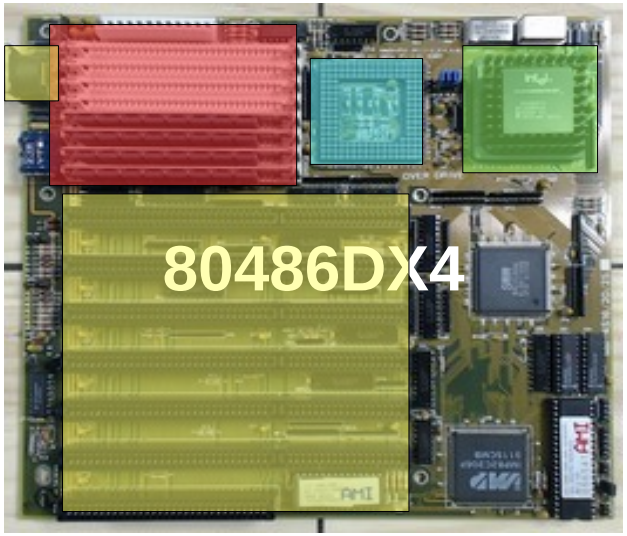


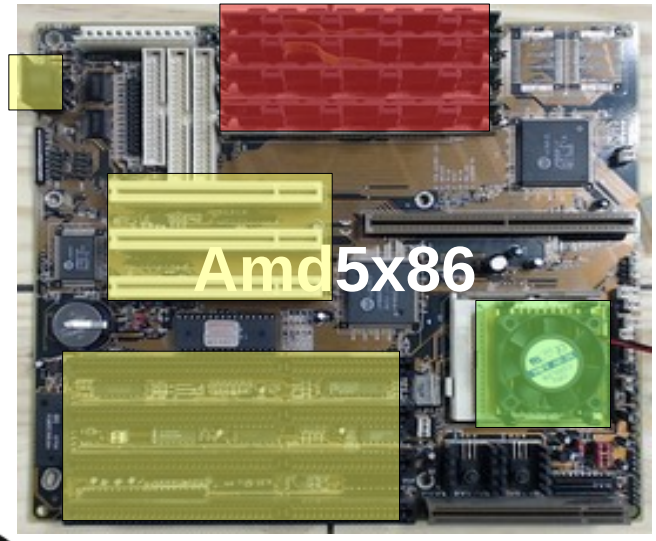


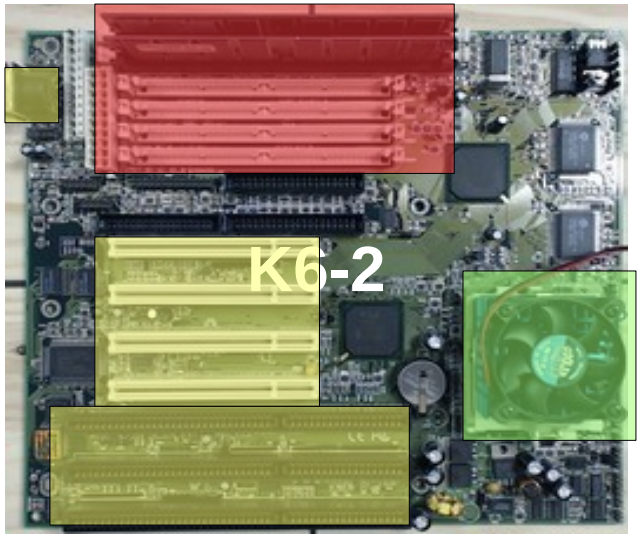


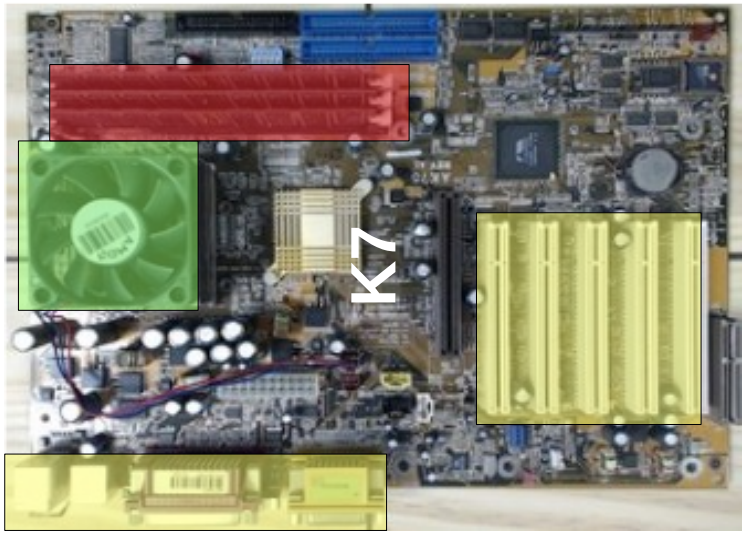
Evolution des cartes mères en un clin d'œil, de 1989 à 2002



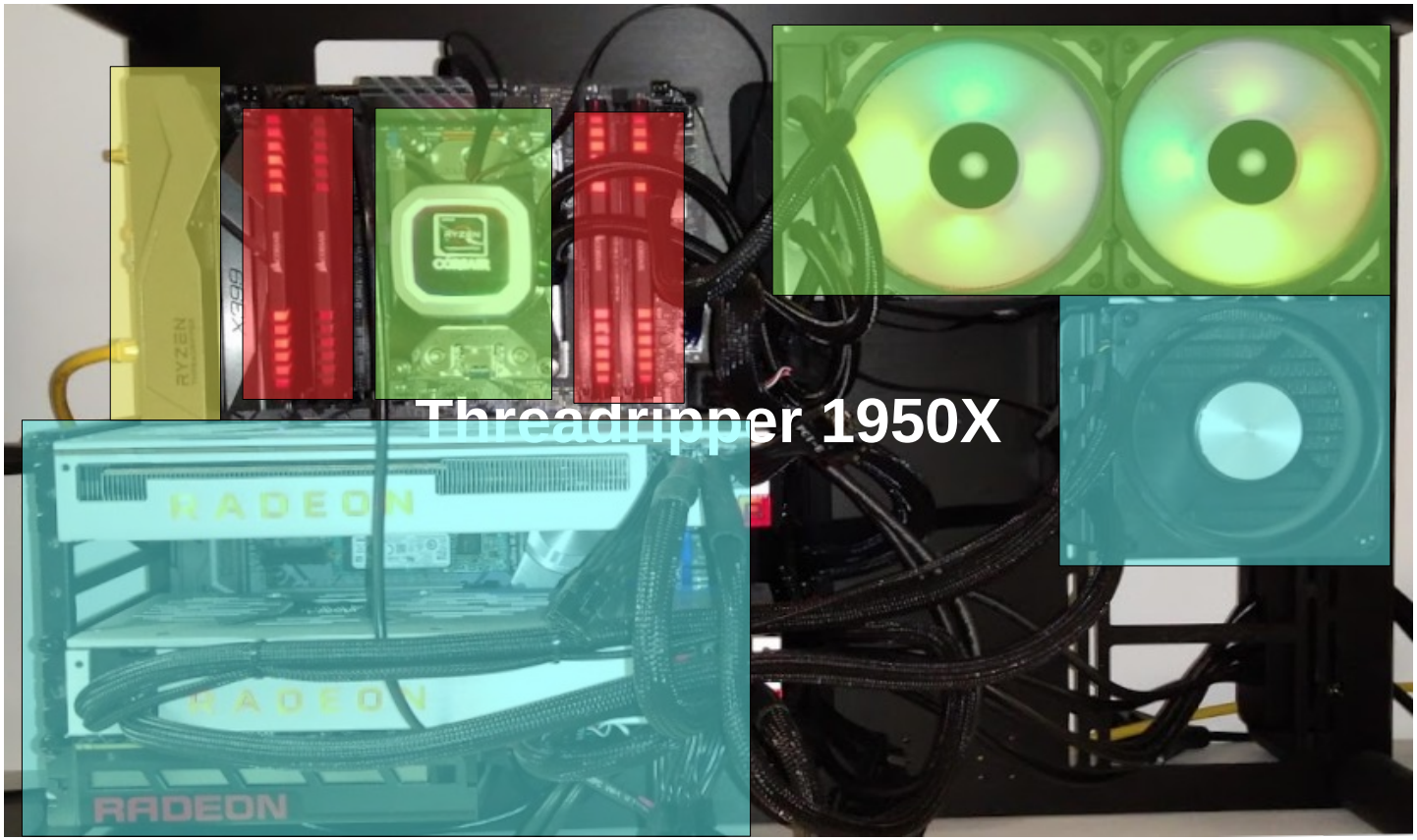




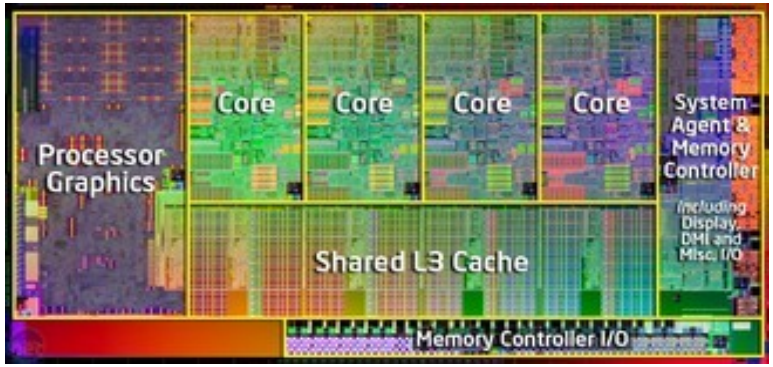


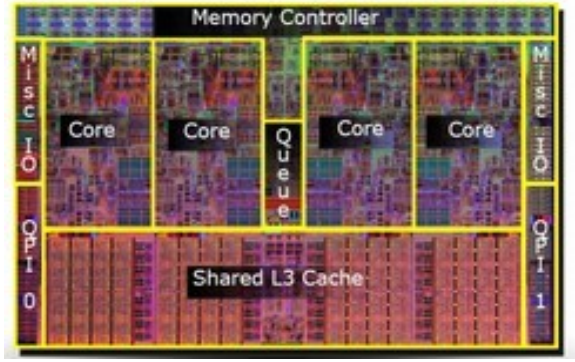


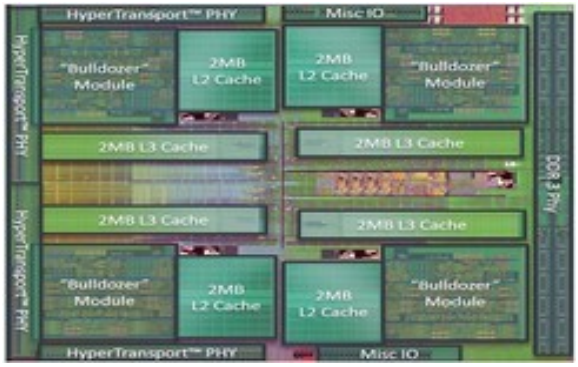
Evolution des cartes mères en un clin d'œil, de 2005 à 2018

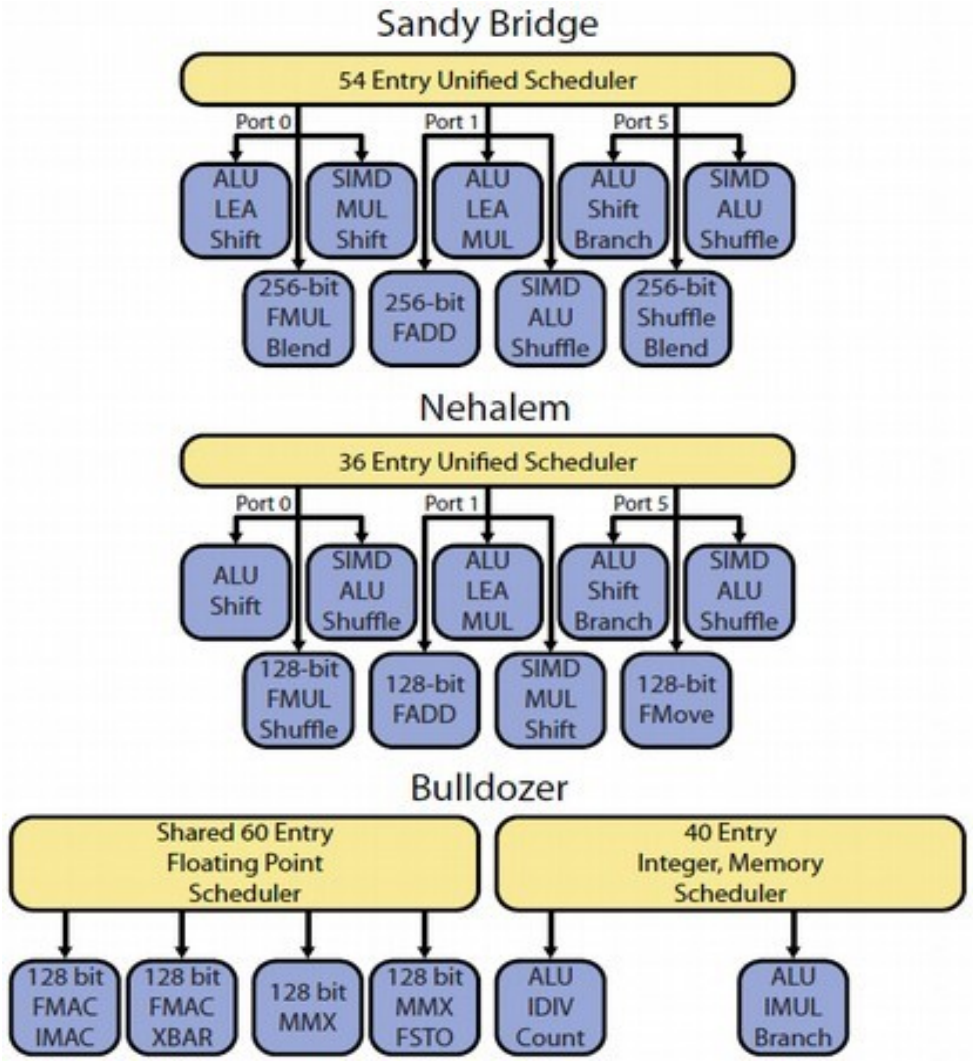


A l'intérieur d'un socket
Exemples de 3 quadri-cœurs

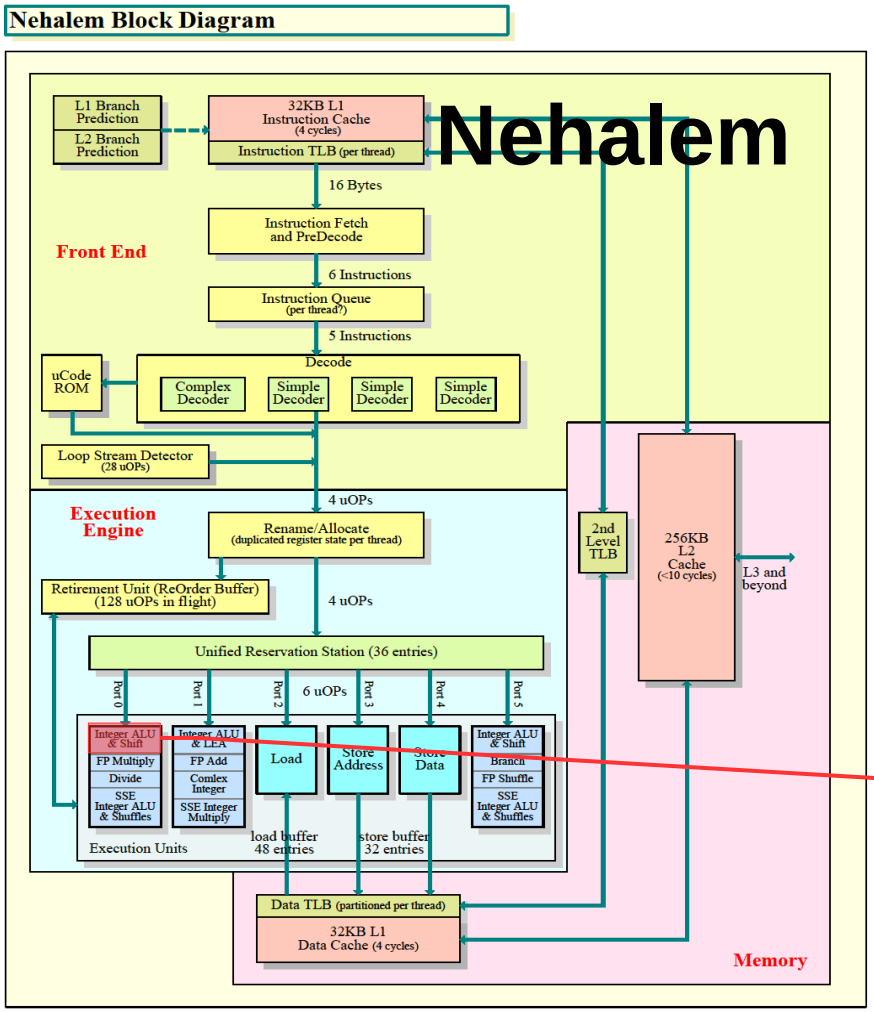


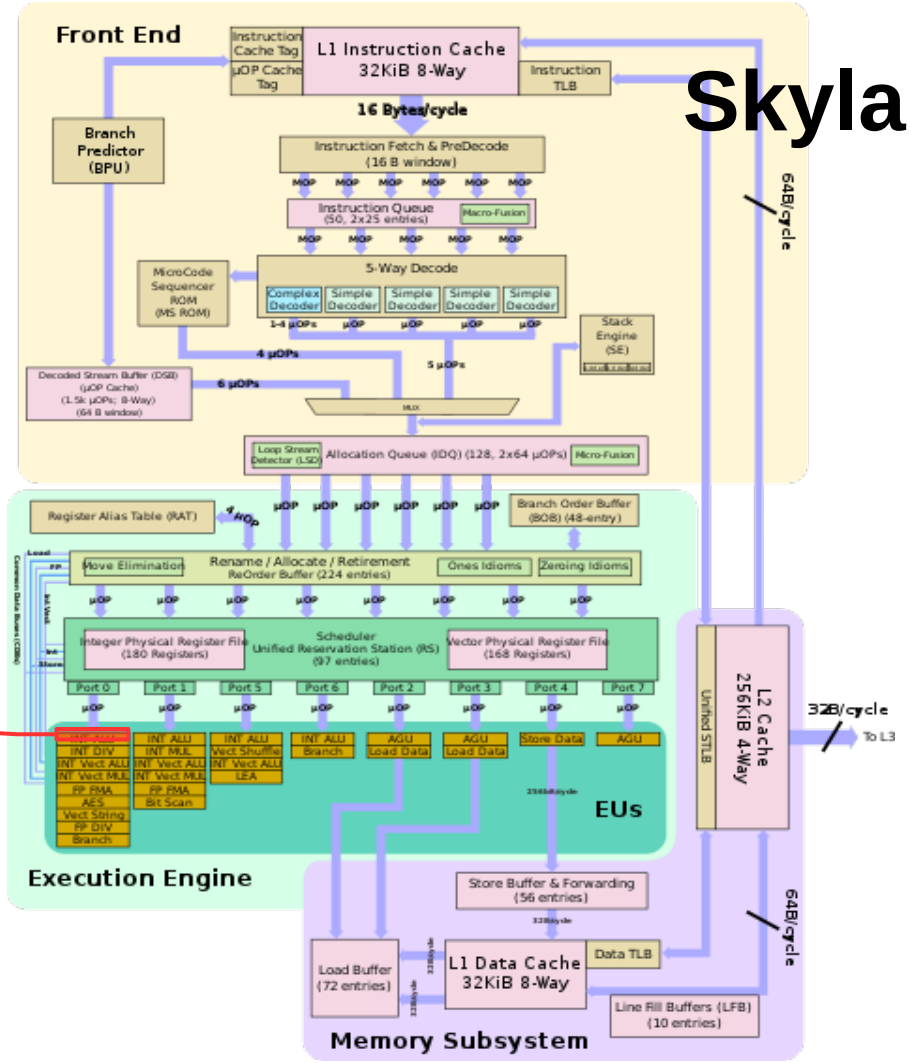




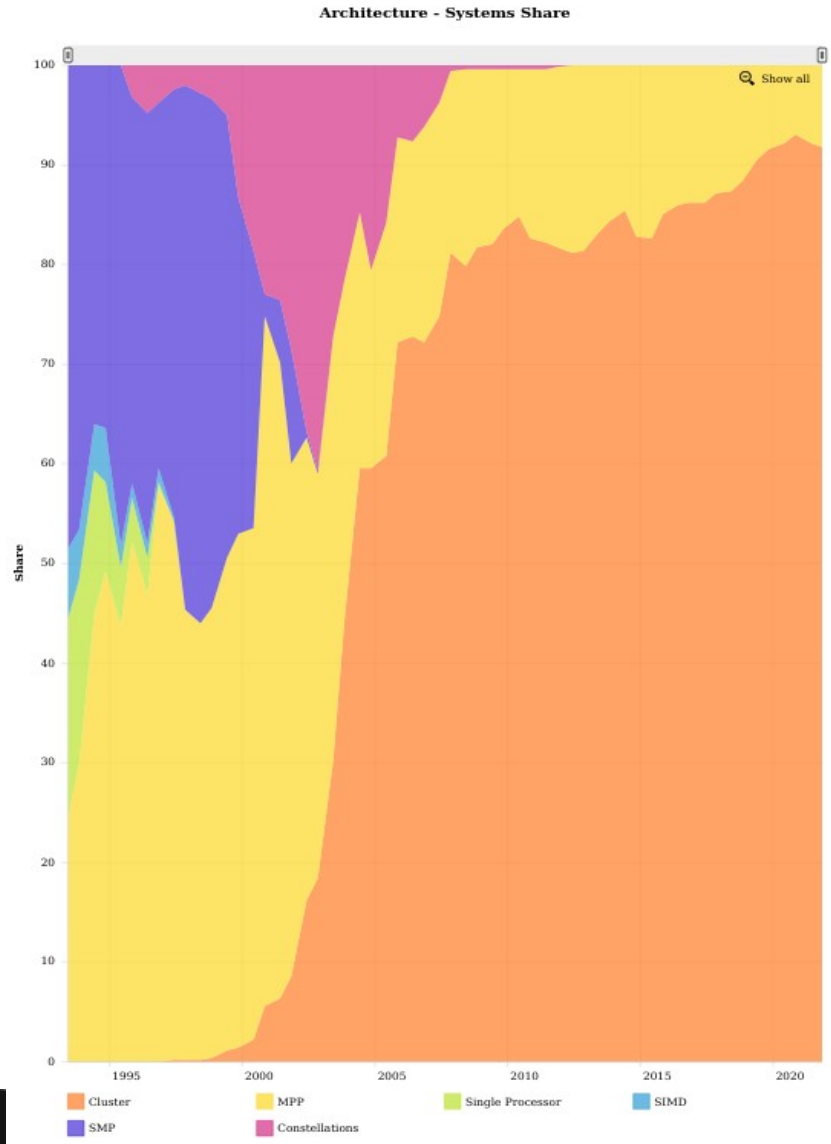


Mais maintenant, chaque cœur est un cauchemar...





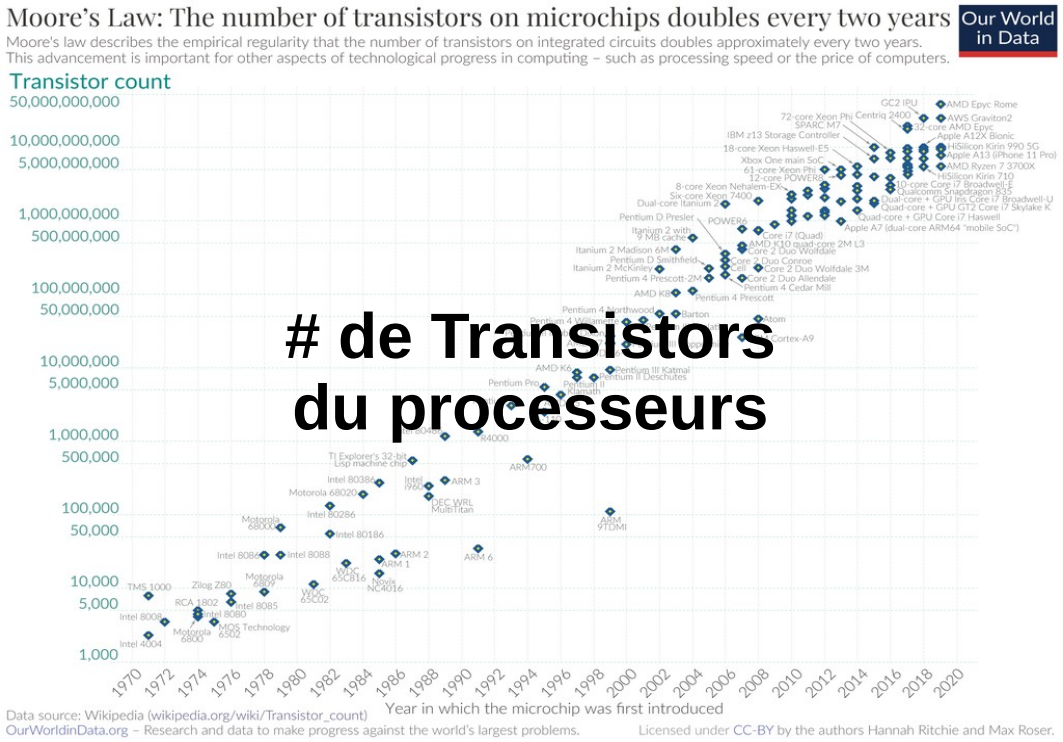
Mais tout ça, c'était du PC...
De 1993 à nos jours, le Top 500



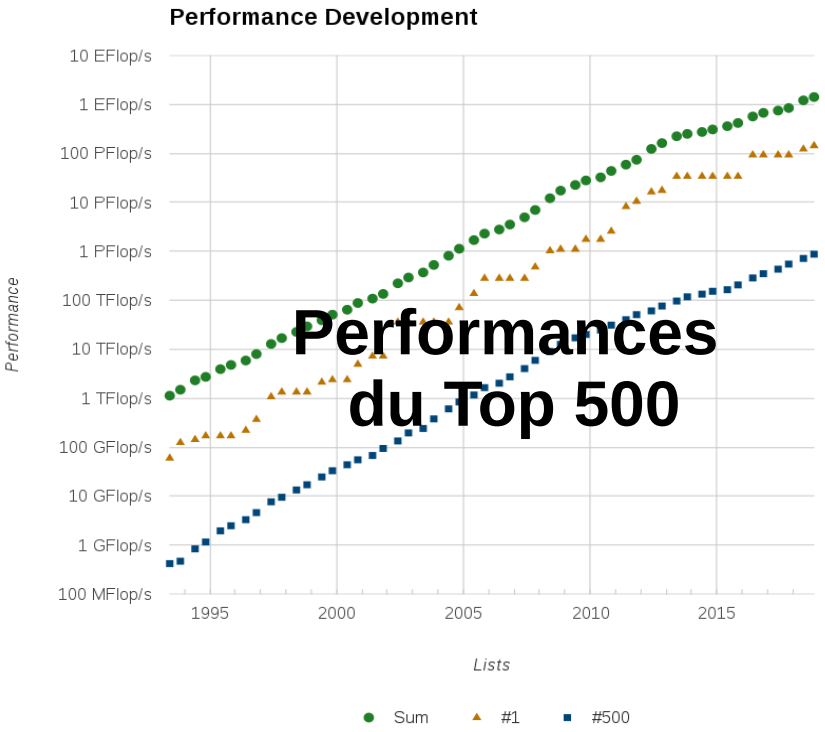
- Les catégories :
  - *Single Processor*
  - SMP : « *Symmetric Multi Processor* »
  - SIMD : « *Simple Instruction Multiple Data* »
  - MPP : « *Massive Parallel Processor* »
  - Constellation : Cœurs >> Nœuds
  - Cluster : machines « génériques »
- 4 ont « disparu »...

« Lois » en informatique
La loi de Moore (et son évolution)

- 1965 : « complexité » des transistors tous les ans
- 1975 : x2 des transistors sur processeur tous les 2 ans
- Actuelle : x2 de « performance » tous les 18 mois

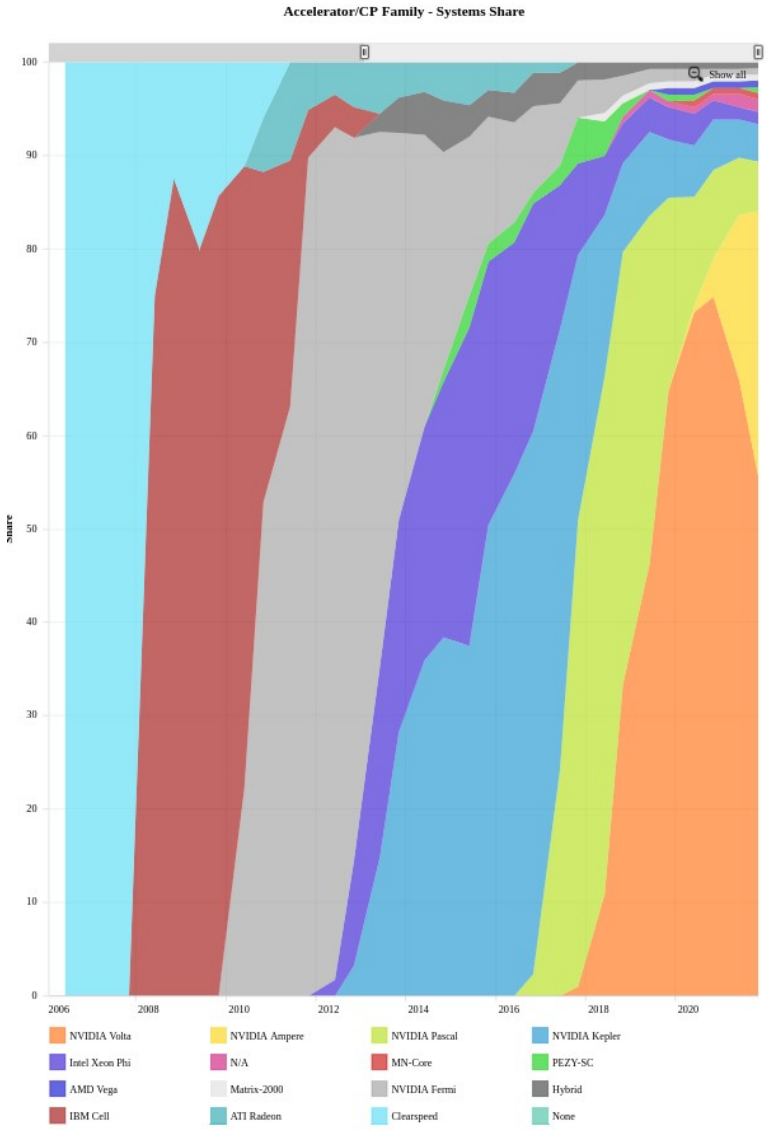


 # de Transistors du processeurs



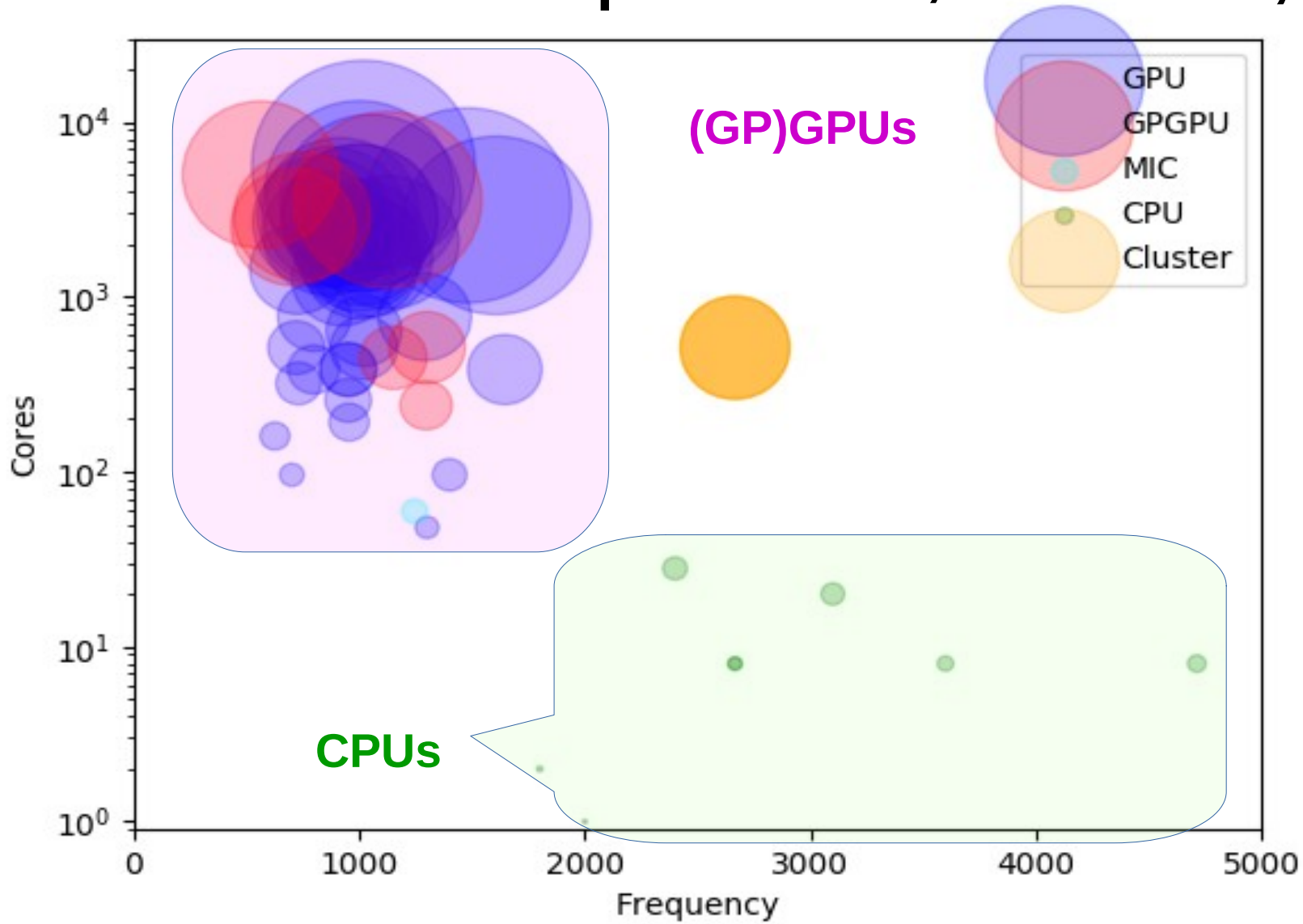
Performances du Top 500

Où en sommes-nous ?
TOP 500 & les accélérateurs

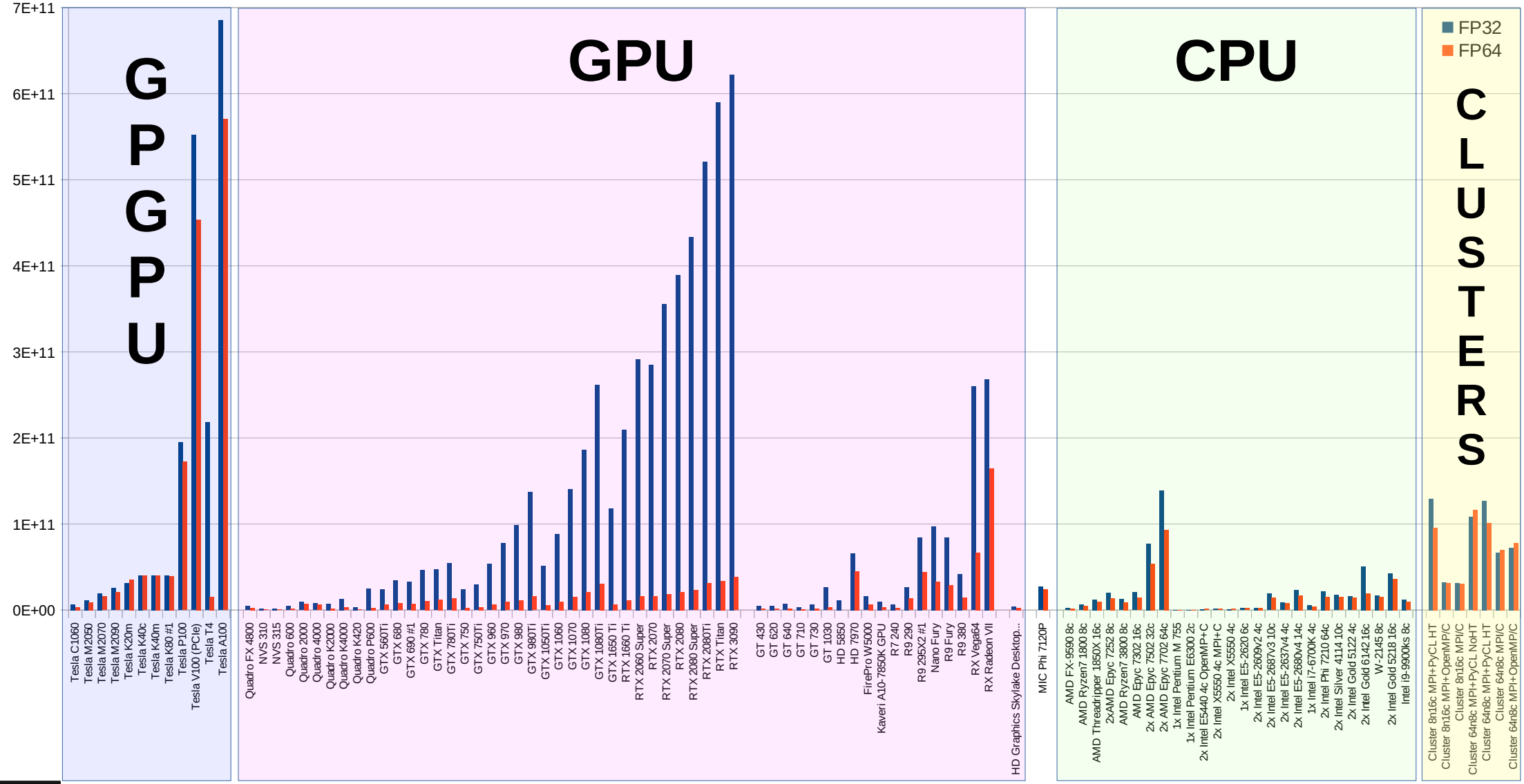


- Pourquoi : comparer les « machines »
- Quoi : le « linpack »
- Où : sur le Top 500
- Quand : juin 2022
- Combien :
  - +1/3 avec accélérateurs :
    - MIC (Intel & copies)
    - GPGPU (Nvidia, AMD)
  - 7/10 dans le Top 10

Comment les représenter ?
Question de fréquences, cœurs, ...



Tous ces plateaux techniques, Et ça donne quoi en performance ?



Mais avant tout chose... La performance, c'est quoi ?

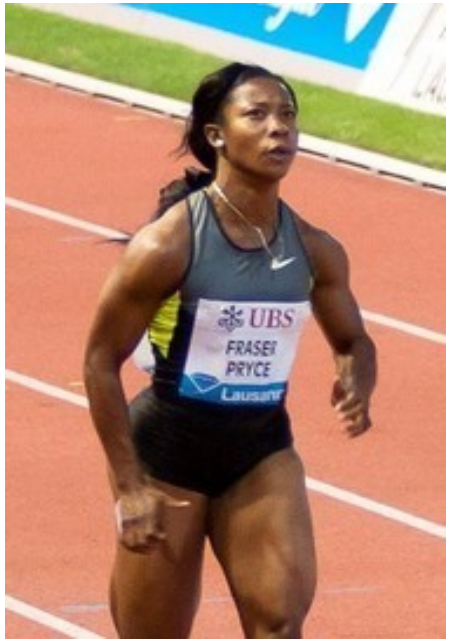
- D'où ça vient ce mot ?
  - cela vient de l'anglais : « accomplir » ou « réaliser »
  - c'est également : « manière de se comporter »
  - mais aussi : « ensemble des possibilités optimales d'un appareil »
- La « performance » :
  - c'est d'abord « faire le job »
  - c'est ensuite « le faire suivant des critères (à définir) »
  - c'est enfin « le faire le mieux possible », mais quel mieux ?

Performance : comment ?
Une question d'objectifs !

- Mettre tous les bagages et la famille dans la voiture
- Attirer l'attention des filles en sortant de boîte de nuit
- Aller d'un point A à un point B dans une ville embouteillée
- Gravir Pikes Peak aux USA



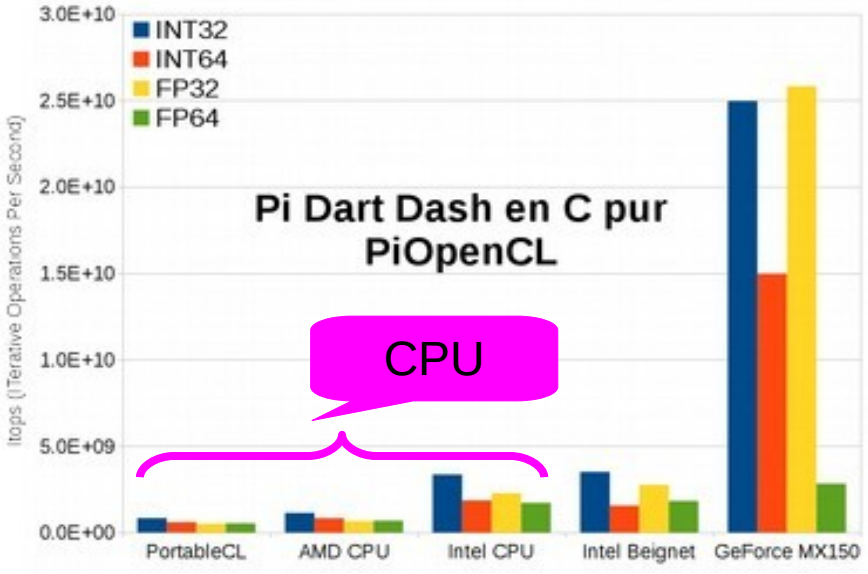

Performance : comment ?
Une question d'observables

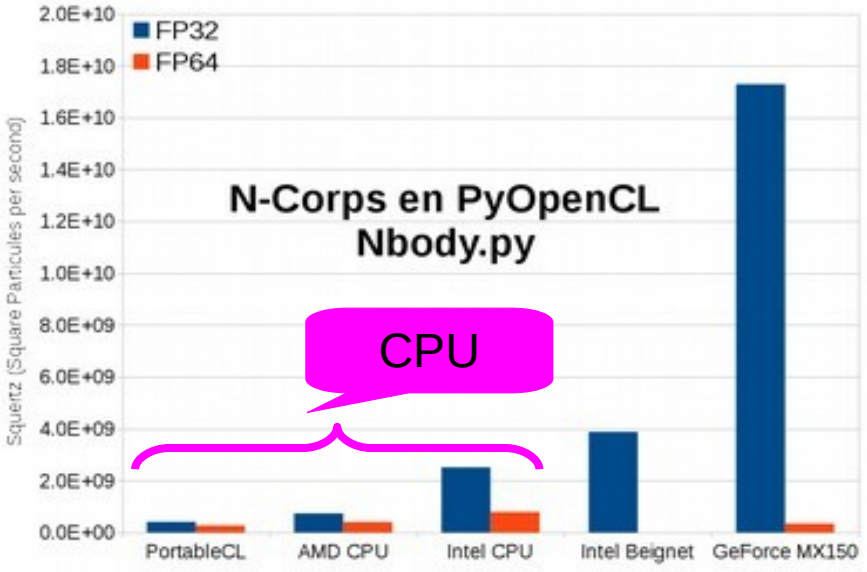


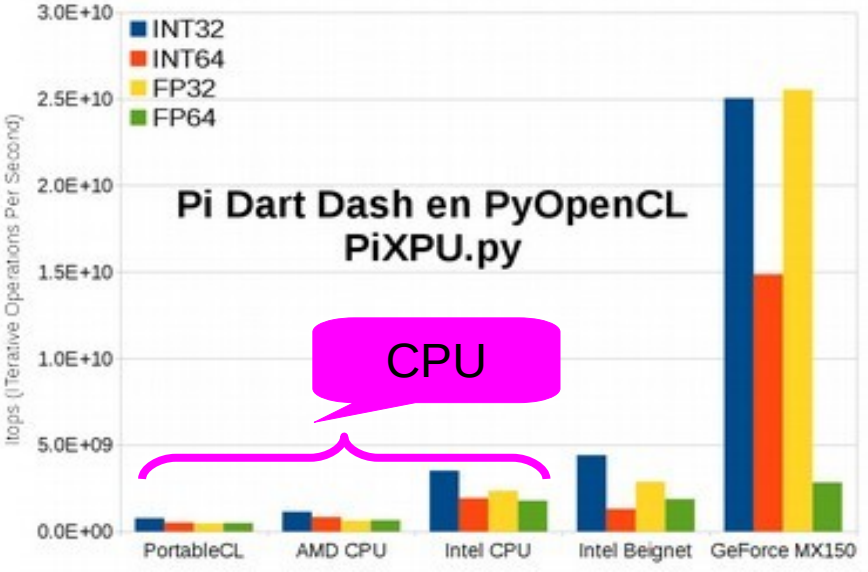
La performance en sport

- Pour faire un 100 mètres ?
- Pour boucler un marathon ?
- Pour lancer un poids ?
- Pour accomplir un heptathlon ?

Où en sommes-nous ? Qu'espérer ?
Sur un laptop, benchs (C|G)PU



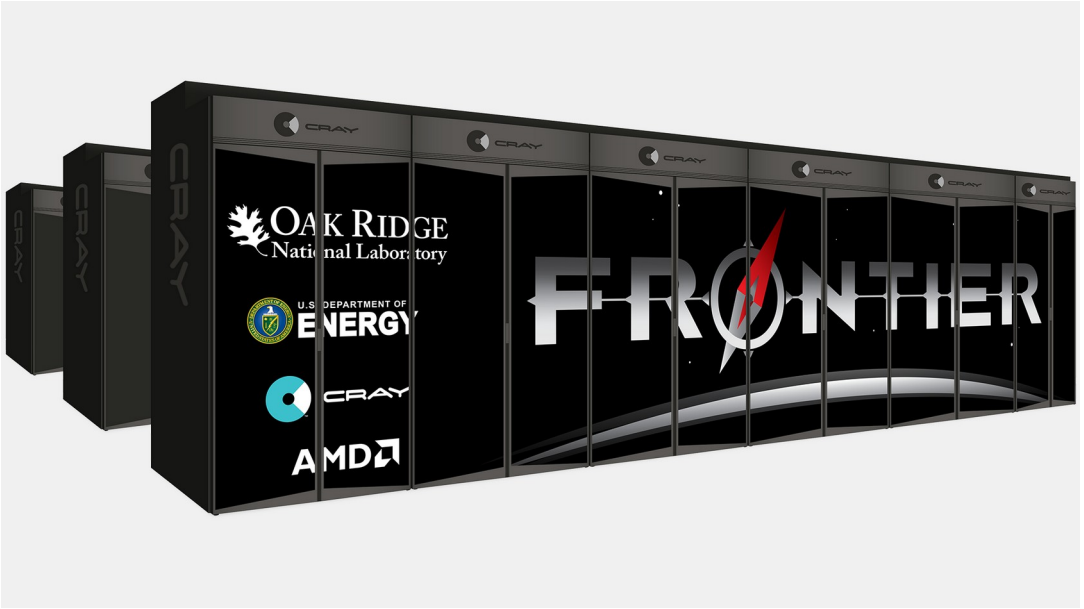




- Un « Pi Dart Dash »
  - En C/OpenCL
  - En Python OpenCL
- Un « N-Corps » naïf

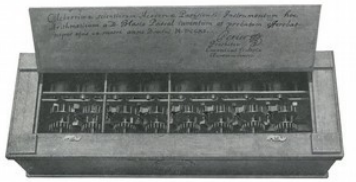
Et sur la 1ère au Top500 : Frontier ?

- Première machine exaflopique, avec 21100 kW de conso...
- 9248 nœuds avec 182 Tflops/nœud :
  - 12.3 Tflops/CPU 7713P : 64*3e9*64
  - 169 Tflops/4GPU MI250X : 4*14080*1.5e6*2
- Rpeak : 92.5 % dans GPU
- Conso : 87 % dans GPU



L’histoire d’un grand détournement ! Mais pas le premier...

- La pascaline avec les engrenages d’horloge…
  - Mais la machine d’Anticythère disposait de 26 engrenages



- La machine de Jacquart et ses « cartes mémoires »
  - Et sa ressemblance frappante avec le modèle de Von Neumann

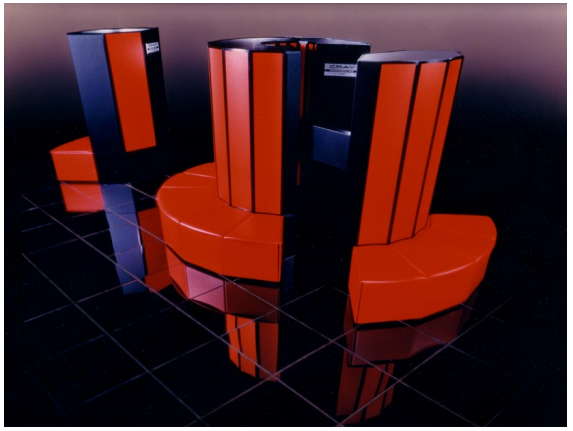
- La lampe (tube à vide) et son amplification de puissance
  - Et les 20000 tubes de la machine Eniac

- Le transistor et son amplification de courant
  - Et sa loi de Moore encore valable 40 ans après...

Il y a une génération (humaine)... Un film de série B en 1984

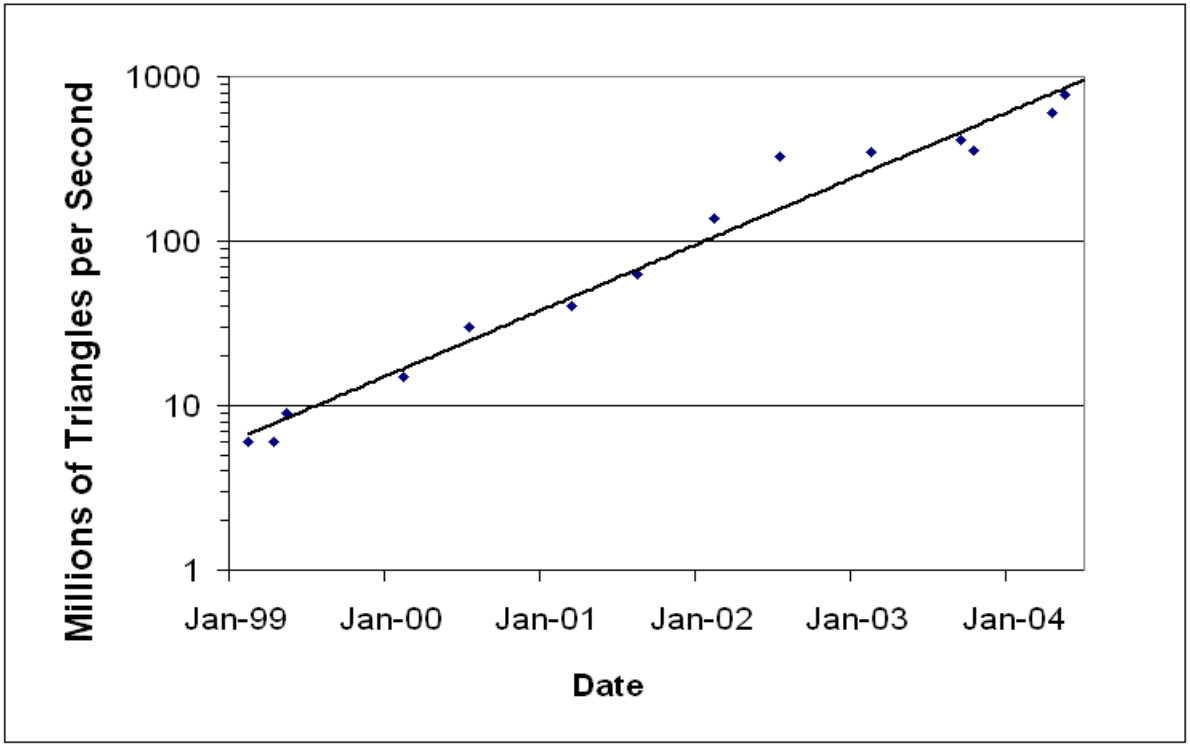
- 1984 : The Last Starfighter
  - 27 minutes d'images synthétiques
  - ~ 30.10^9 operations par image
  - Utilisation d'un Cray X-MP (130 kW)
  - 68 jours (en fait, 1 année nécessaire)

- 2020 : RTX 3090 (350 W)
  - 33 secondes
  - Comparison RTX 3090 / Cray
    - Performance : 178 000 !
    - Consommation ~ 66 000 000 !



Pourquoi le GPU est « disruptif » ? Le « grand détournement » en HPC

- Dans une conférence à l'ENS-Cachan en février 2006, ceci...
  - x100 en 5 ans



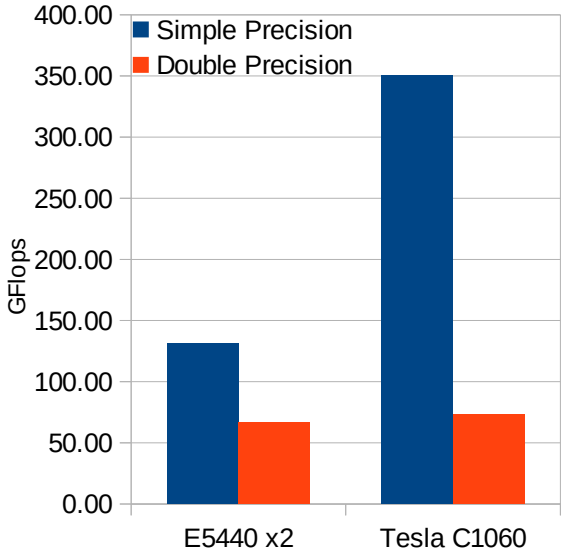
- Entre 2000 et 2015
  - GeForce 2 Go/GTX 980Ti : de 286 à 2816000 MOpérations/s : x10000
- Pour un CPU « classique » : x100

Position du GPU face aux autres?
Les autres « accélérateurs »

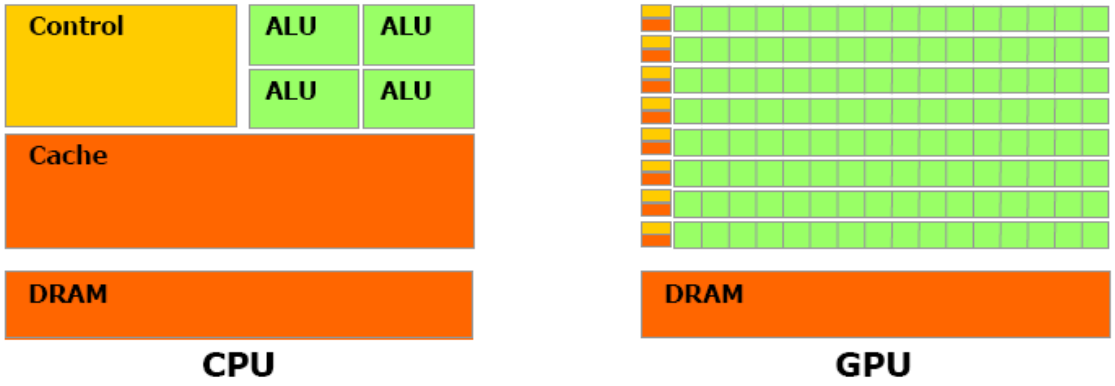
- Accélérateur ou la vieille histoire des coprocesseurs...
  - 1980 : 8087 (sur 8086/8088) pour les opérations en virgule flottante
  - 1989: 80387 (sur 80386) et son respect du IEEE 754
  - 1990 : 80486DX et l'intégration du FPU dans le CPU
  - 1997 : K6-3DNow ! & Pentium MMX : SIMD dans le CPU
  - 1999 : fonctions SSE et le début d'une longue série (SSE4 & AVX)
- Quand les circuits restent hors du CPU
  - 1998 : Les DSP de la catégorie des TMS320C67x comme outils
  - 2008: le Cell dans la PS3, IBM dans le Road Runner & un Top1 au Top500
  - 2013 : Tianhe-2 à base de Xeon Phi, Top 1 pendant 2 ans
- Travail de compilateurs & forte dépendance au modèle

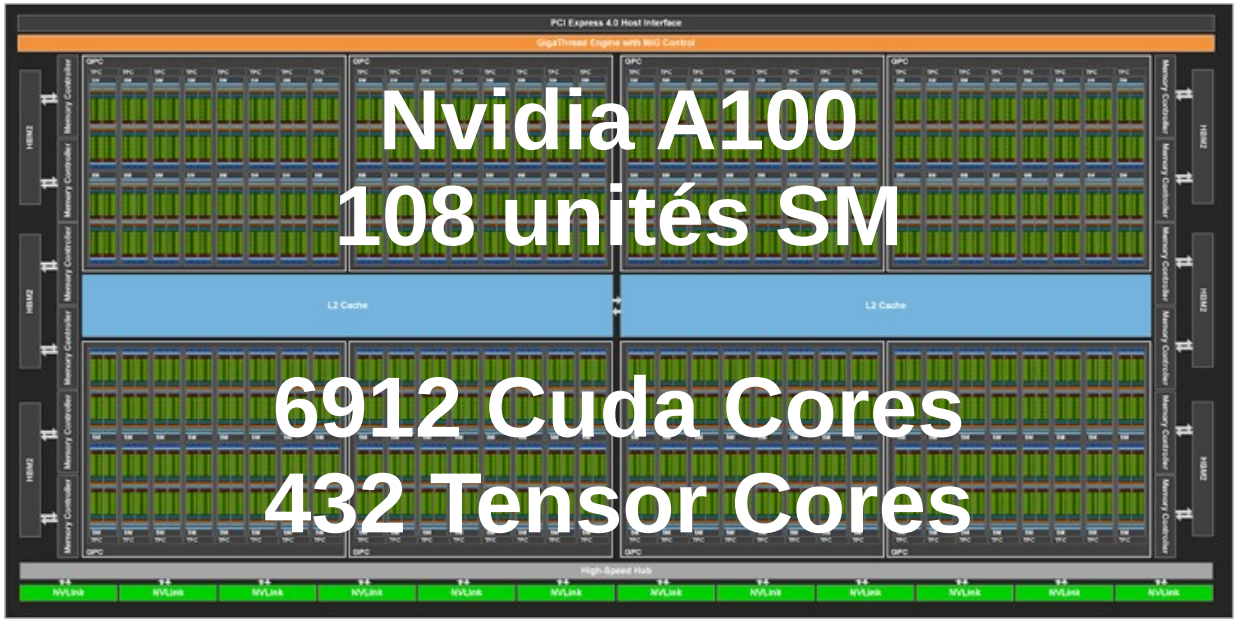
« *Let's go back to the source !* »

- « Nvidia Launches Tesla Personal Supercomputer »
- Quand : le 19/11/2008 Qui : Nvidia
- Où : sur Tom's Hardware
- Quoi : une carte PCIe C1060 PCIe avec 240 cores
- Combien : 933 Gflops SP (mais 78 Gflops DP)



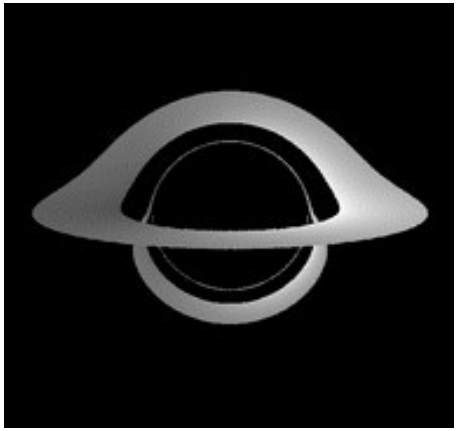
Combien de composants à l'intérieur ?
Quelle différence entre GPU & CPU





- Opérations
  - Multiplication de Matrice
  - Vectorisation
  - « Pipelining »
  - Shader (multi)processeur
- Programmation : 1993
  - OpenGL, Glide, Direct3D, ...
- Généricité : 2002
  - CgToolkit, CUDA, OpenCL

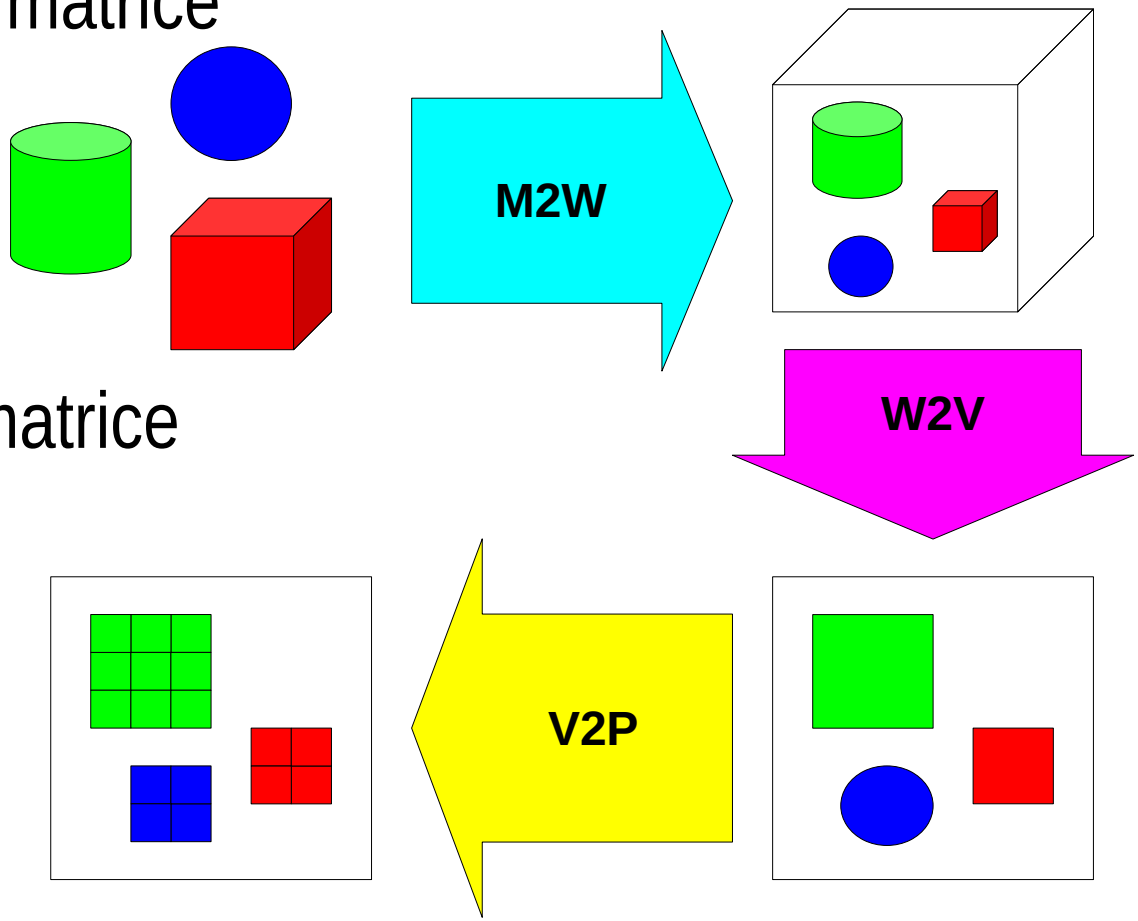
Pourquoi le GPU est-il si puissant ? Pour construire une image 3D !



- 2 approches:
  - Raytracing : PovRay (« dimensions » de l'UMPA)
  - Shadering : 3 opérations
- « Raytracing » :
  - De l'oeil vers les contacts de chaque objet
- « Shadering »
  - ***Model2World*** : objets vectoriels placés dans la scène
  - ***World2View*** : projection des objets dans un plan de vue vectoriel
  - ***View2Projection*** : pixellisation du plan de vue

Pourquoi le GPU est-il si puissant ? Shadering & Calcul matriciel

- **Model 2 World** : 3 produits de matrice
  - Rotation
  - Translation
  - Mise à l'échelle
- **World 2 View** : 2 produits de matrice
  - Positionnement de la caméra
  - Direction de l'endroit pointé
- **View 2 Projection**
  - Pixellisation



Donc, un GPU : c'est un « gros » Multiplicateur de Matrice

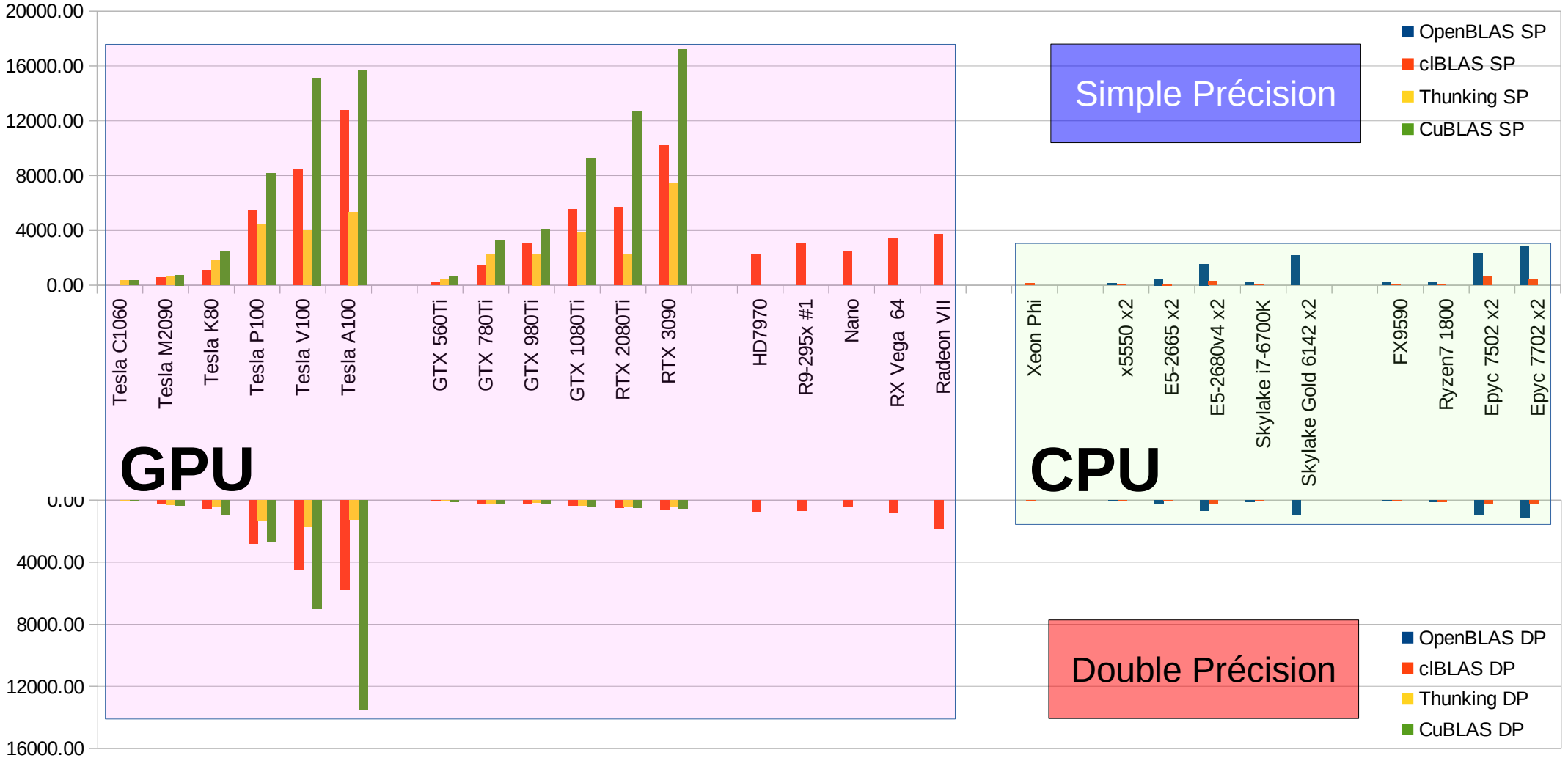
Multiplication Matrice Matrice : Pourquoi aussi « efficace » ?

- Soit 2 matrices A et B de dimensions NxP et PxM
- Le produit matriciel de A par B donne C
- Chaque élément de C, C_{ij} pour i de 1 à N et j de 1 à M :

$$C_{ij} = \sum_{k=1}^{k=P} A_{ik} B_{kj}$$

- Les C_{ij} sont indépendants : parallélisme « gros grain »
- Les $A_{ik}B_{kj}$ groupables par bloc : vecteurs & unités FMA

Multiplication Matrice Matrice
Et c'est vraiment le cas ?



Pourquoi le GPU est-il si puissant ? Toute la taxonomie de Flynn en «M»

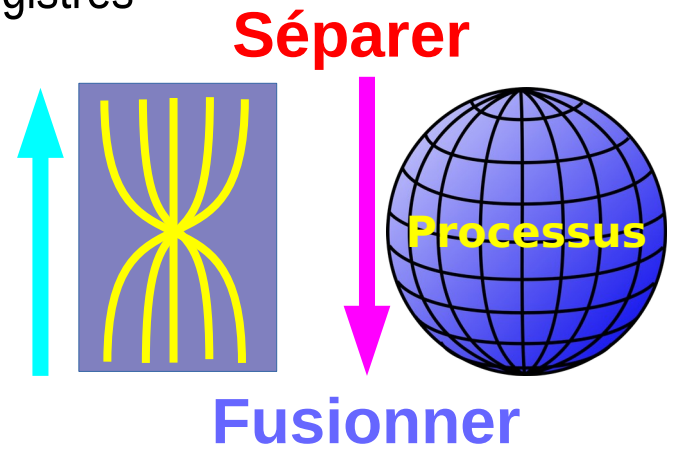
- Vectorielle : SIMD (Simple Instruction Multiple Data)
  - Addition de 2 positions (x,y,z) : 1 commande unique
- Parallèle : MIMD (Multiple Instructions Multiple Data)
  - Plusieurs executions en parallèle avec à peu près les mêmes datas
- En fait, SIMT : Simple Instruction Multiple Threads
  - Toutes les unités de traitement partagent les Threads
  - Chaque unité de traitement peut travailler indépendamment des autres
- Nécessité de synchroniser les Threads

Le GPU & l'exécution en parallèle ? Les différentes approches...

- Pipeliner à grain fin :
  - 5 instructions simples @ chaque pas
    - Chargement de l'instruction
    - Décode de l'instruction
    - Exécution
    - (Mémorisation)
    - Écriture en retour
  - 2 spécifications des RISC : 1 instruction/cycle, exploitation des registres

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

- Deux approches
  - Vectorisation : Fusion/Processus/Séparation
  - Distribution : Séparation/Processus/Fusion
- En fait, paralléliser, c'est plutôt « médianiser »



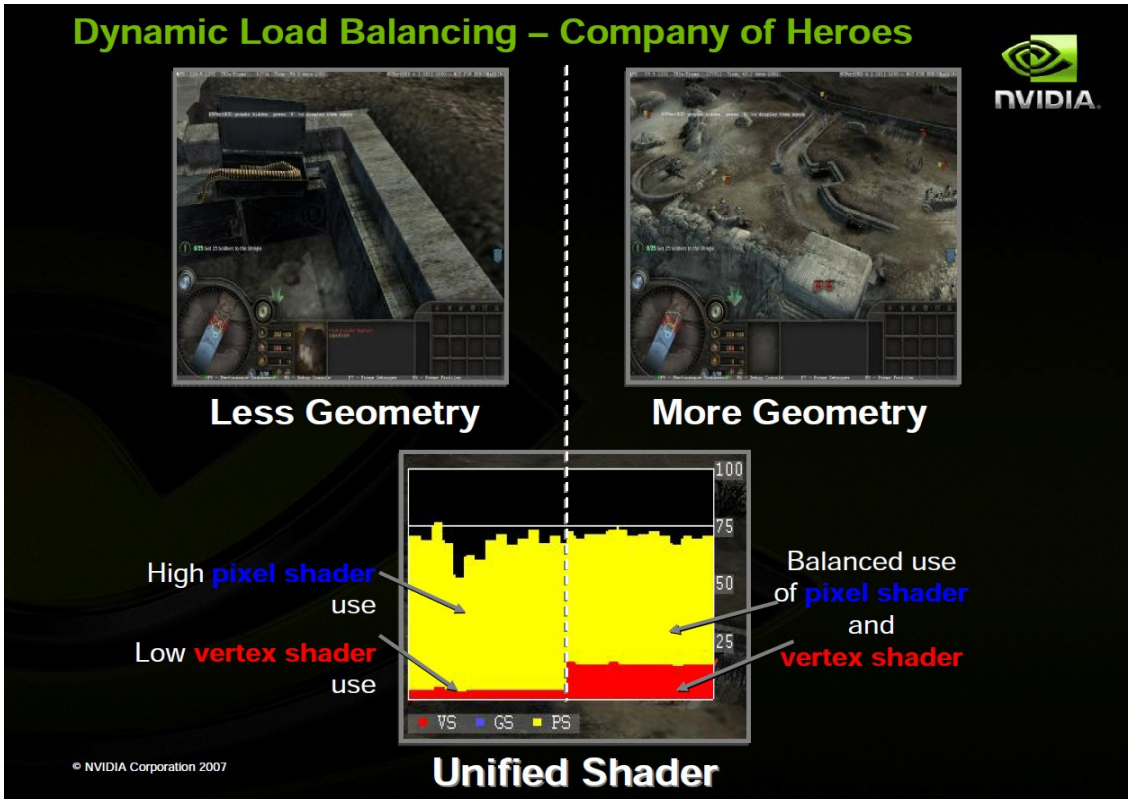
Mais pas seulement un MM…
Retour à un traitement générique

Dans le GPU

- Des unités spécialisées
- Une grande utilisation de pipelines

Perte d'adaptabilité

- Pour des scènes changeantes
- Pour différentes natures de détails



La solution

Des unités un peu plus généralistes...

Dates (importantes)

- 1992-01 : OpenGL et la naissance d'un standard
- 1998-03 : OpenGL 1.2 et des fonctions intéressantes
- 2002-12 : Cg Toolkit (Nvidia) et des extensions de langage
- 2007-06 : CUDA (Nvidia) ou l'arrivée d'un premier langage
- 2008-06 : Snow Leopard (Apple) intègre OpenCL
- 2008-11 : OpenCL 1.0 et ses premières spécifications
- 2011-04 : WebCL et sa première version par Nokia (morts…)
- 2012-11 : OpenACC, approche « OpenMP » pour GPU (tous…)
- 2014-03 : SyCL, approche « TBB » de Khronos en C++
- 2015-04 : KOKKOS, approche « OpenMP » en C++ (*PU)
- 2020-05 : OpenMP « offloading » sur GPU
- 2020-12 : oneAPI, nouvelle approche objet d'Intel, compatible SyCL...

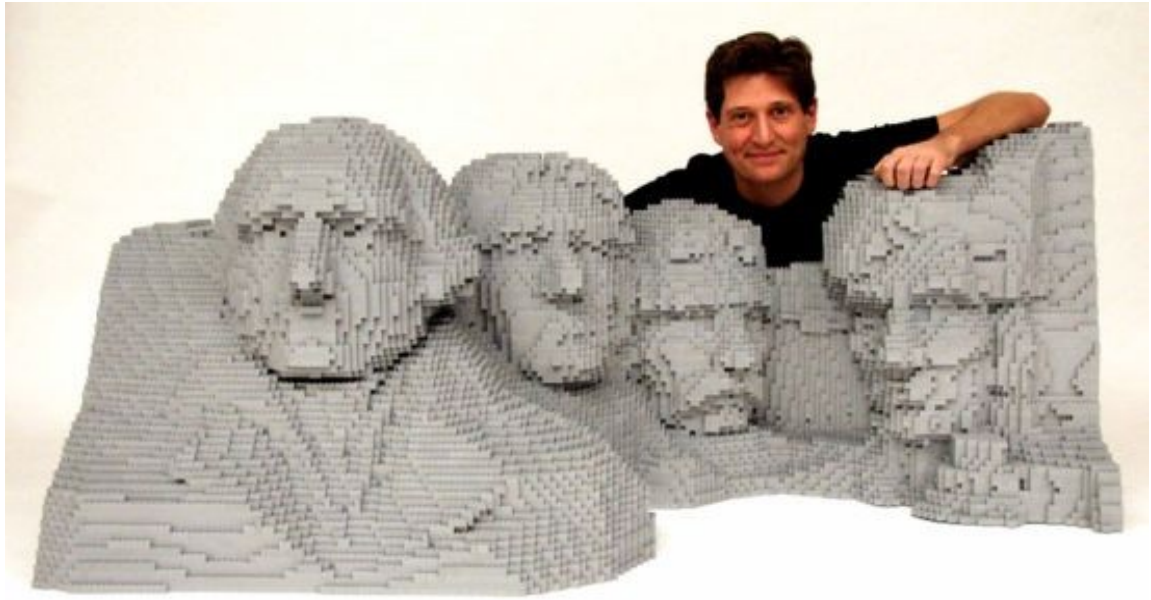
Développer ou intégrer ?

Développer (de zéro)

D'une grosse masse (projet)...
... par affinages successifs (code)...
... à un produit fini (application).

Intégrer

De composants (*framework*)...
... par assemblage...
... à un produit fini (application).



Entre développeur et intégrateur Que choisir ?

- 2 approches
  - Une approche « intégrateur »
    - Le code utilise des librairies génériques
    - Le code n'est modifié que pour remplacer ces appels
  - Une approche « développeur »
    - Le GPU est un nouveau processeur
    - Il exige un apprentissage comme tout nouveau matériel
    - Le code doit être réécrit pour l'utiliser au mieux
- 1 contrainte mais 2 manifestations : **le temps**
  - Le temps de programmation : plutôt intégrateur
  - Le temps d'exécution : plutôt développeur

Petit tour d'horizon des modèles

Modèles de programmation parallèle

	Cluster	Nœud CPU	Nœud GPU	Nœud Nvidia	Accelerator
MPI	Oui	Oui	Non	Non	Oui*
PVM	Oui	Oui	Non	Non	Oui*
OpenMP	Non	Oui	Non	Non	Oui*
Pthreads	Non	Oui	Non	Non	Oui*
OpenCL	Non	Oui	Oui	Oui	Oui
CUDA	Non	Non	Non	Oui	Non
TBB	Non	Oui	Non	Non	Oui*
OpenACC	Non	Oui	Non	Oui	Oui
Kokkos	Non	Oui	Oui	Oui	Oui

- OpenCL & Kokkos semblent les plus « universels »
- CUDA n'est utilisable QUE sur les GPUs Nvidia
- Les accélérateurs semblent les plus polyvalents, une illusion...

Agir comme un intégrateur : Pour ne pas réinventer la roue !

Librairies de programmation parallèle

	Cluster	**Node CPU**	Node GPU	Node Nvidia	Accelerator
BLAS	BLACS MKL	OpenBLAS MKL	clBLAS	CuBLAS	OpenBLAS MKL
LAPACK	Scalapack MKL	Atlas MKL	clMAGMA	MAGMA	MagmaMIC
FFT	FFTw3	FFTw3	clFFT	CuFFT	FFTw3

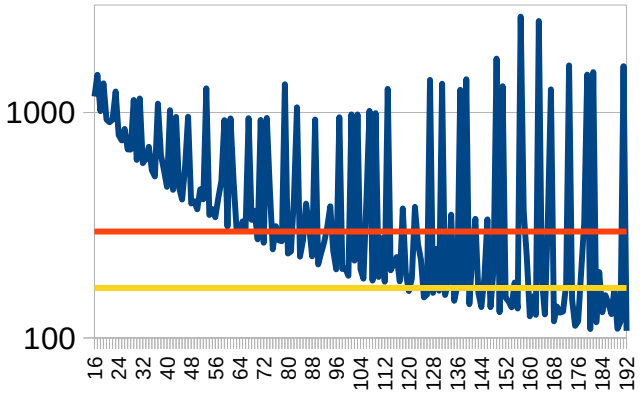
- Les librairies classiques utilisables directement sur GPU
  - Nvidia fournit plein d'implémentations
  - OpenCL n'en fournit quelques unes

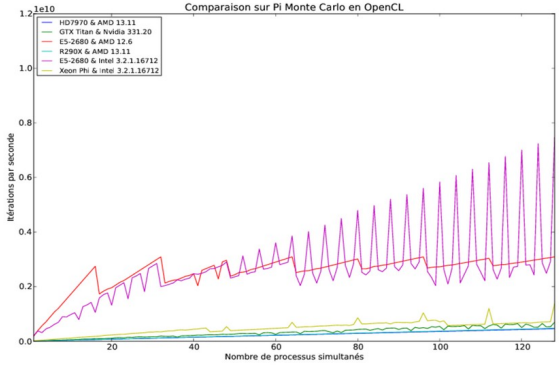
Et mon code « pur » ?

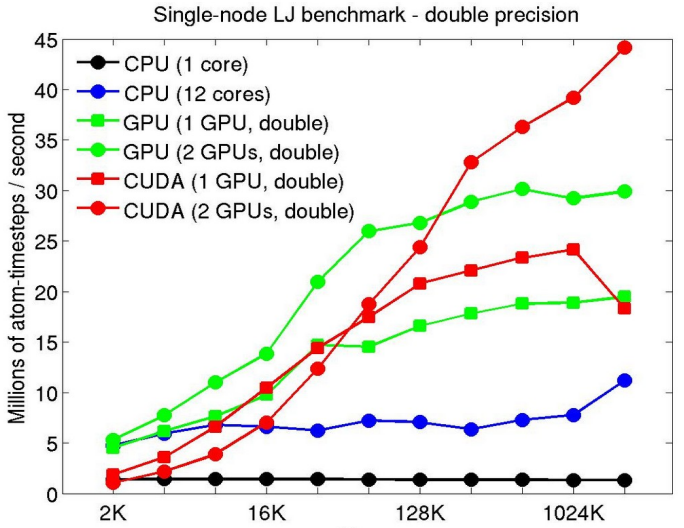
- Approche ACC :
  - Une approche « pragma »tique à la OpenMP
  - OpenACC : une initiative (seulement une initiative hors PGI ou Cray...)
  - PGI OpenACC :
    - Plutôt efficace
    - LicenceS (pour PGI ET PGI/Cuda) : à la Matlab/Toolboxes
- Approche KOKKOS :
  - Couche C++ et « spécialisation » à la compilation
- Approche Par4All (abandonné mais prometteur) :
  - Un préprocesseur analyse et le code et le « transcrit »
  - Implémentation OpenMP, Cuda (et OpenCL)
  - Projet assez « vert » (mais très pédagogique)

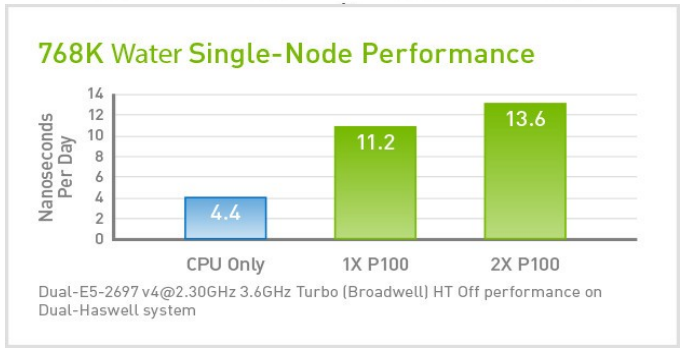
Accélération GPU ? Vérité ou mensonge

Prêt à prendre la « pilule rouge » ?





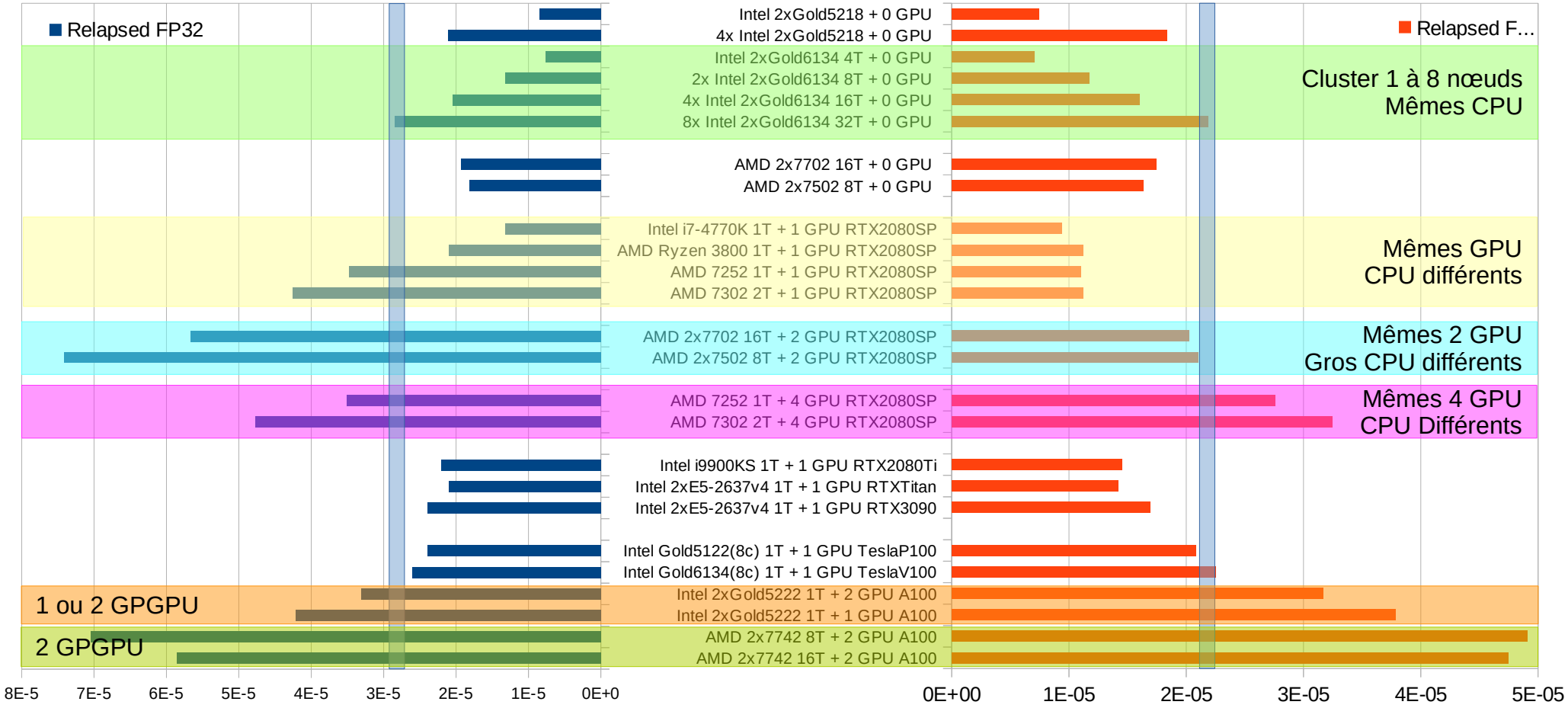




Pour les applications « métier » Modèle « Ikea » ou « Crozatier »

- Attention !
  - Aujourd'hui n'est pas demain : récupérer le contexte complet
    - Matériel : lshw, lscpu, nvidia-smi, … & logiciel : système, pilotes, etc...
  - A chaque usage (ou préparation) son contexte optimal
    - Une préparation, c'est l'assemblage entre recette, ustenciles et ingrédients
- Mais rassurez vous !
  - C'est aussi le cas pour les CPU ;-)
- Retournons aux autres codes :
  - Approche « intégrateur » ou approche « développeur »

Code « métier » trhybride : GENESIS
… et un même cas d'usage !

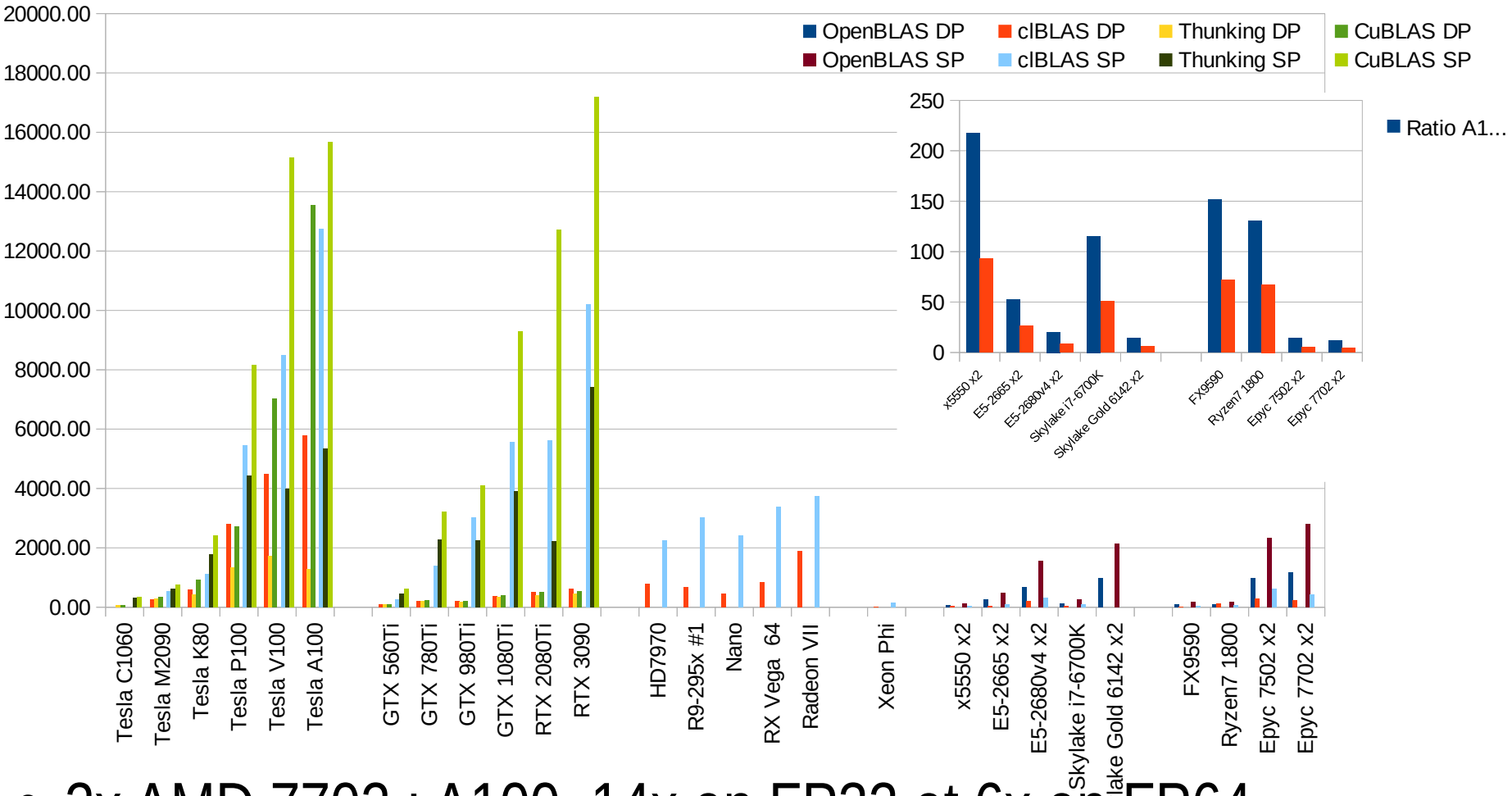


Comme quoi, difficile de se projeter sur une performance...

Première « intégration » bas niveau
A la découverte de BLAS

- Fonctions BLAS
  - 3 niveaux de fonctions :
    - Niveau 1 : rotations, normes, échanges, copies, produits scalaires, ...
      - Exemple : xSWAP pour échanger 2 vecteurs
    - Niveau 2 : produit matrice-vecteur, résolution système triangulaire,
      - Exemple : xTSRV pour résolution de système triangulaire
    - Niveau 3 : opérations simples sur les matrices
      - Exemple : xGEMM pour le produit de matrice
- Les fonctions de toutes nos attentions :
  - sGEMM & dGEMM pour les produits simple & double précision

Remise en perspective avec le matériel xGEMM pour 17 (GP)GPU et 9 processeurs...



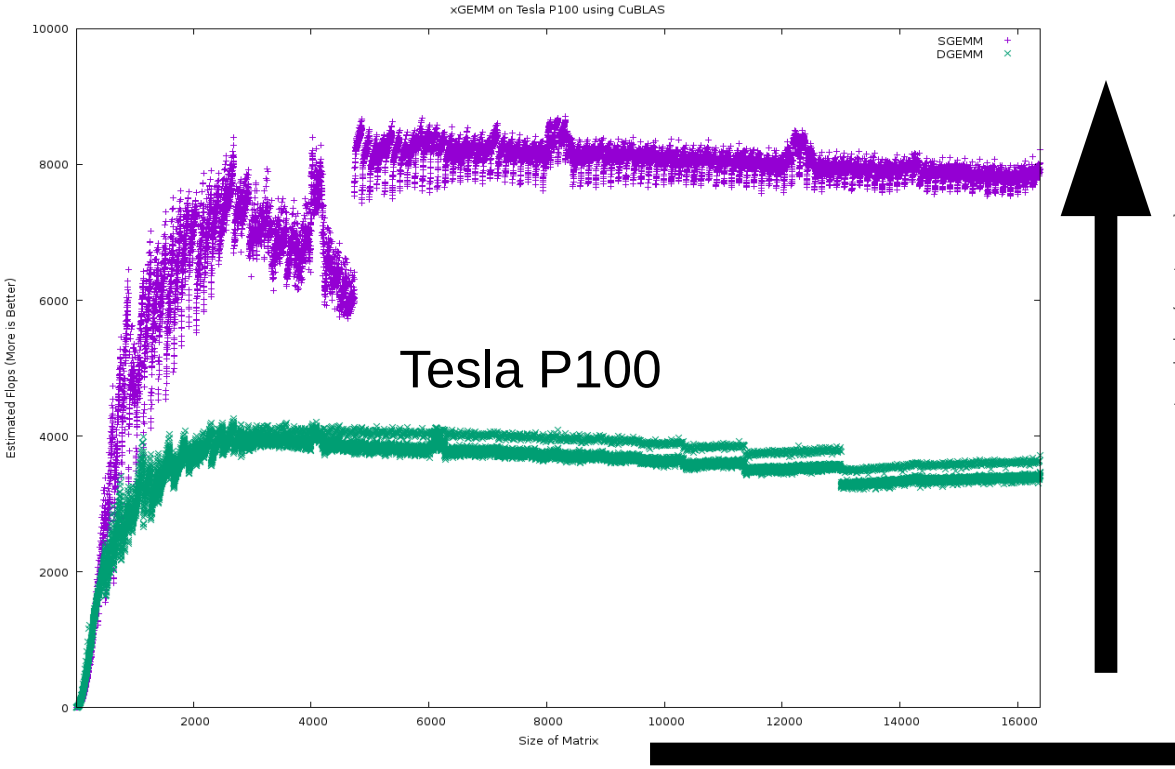
- 2x AMD 7702 : A100, 14x en FP32 et 6x en FP64
- Oui, la puissance MxM est là, mais pour toute taille ?

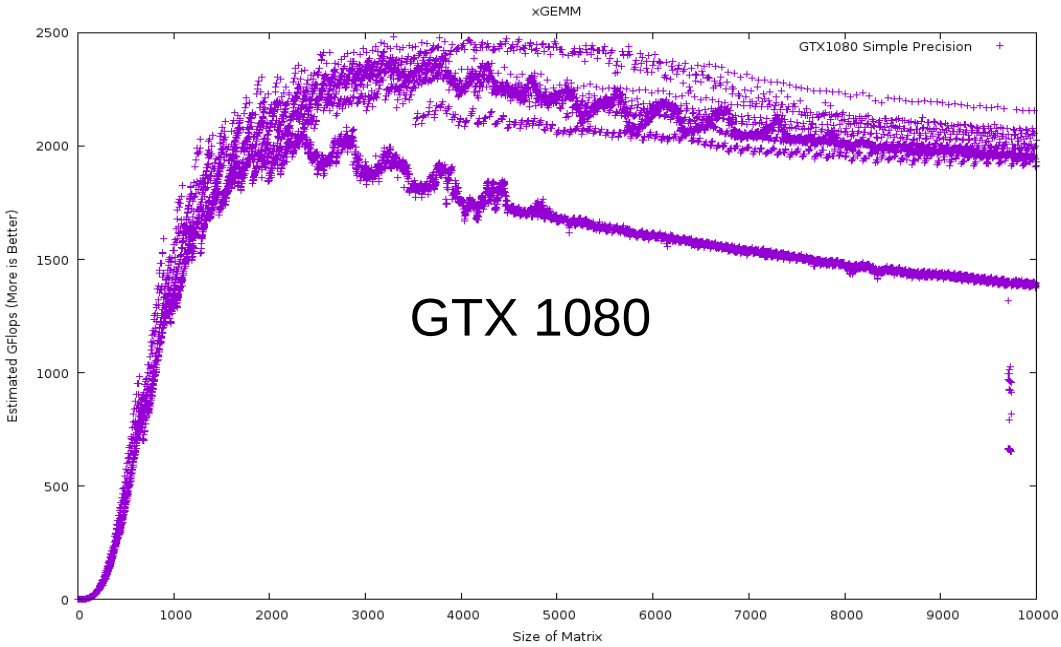
Déjà en 2010, étrange… Le comportement Tesla C1060...

- Uniquement produit matriciel : xGEMM
- Propriété : Transposé (A * B)=Transposé(A) * Transposé(B)
- Résultats (en Gflops) : Yesss !
  - SP : FBLAS/CBLAS : 12, CuBLAS : 350/327 : x27 !!!
  - DP : FBLAS/CBLAS : 6, CuBLAS : 73/70 : x11 !
- Surprise : Ah !!! CuBLAS préfère les x16 !
  - SP : 16000^2, 350, mais 15999^2 ou 16001^2, 97 : x3,6 !
  - DP : 10000^2, 73, mais 9999^2 ou 10001^2, 31 : x2,35

Ce que Nvidia ne précise jamais… xGEMM sur des tailles différentes...

Performance

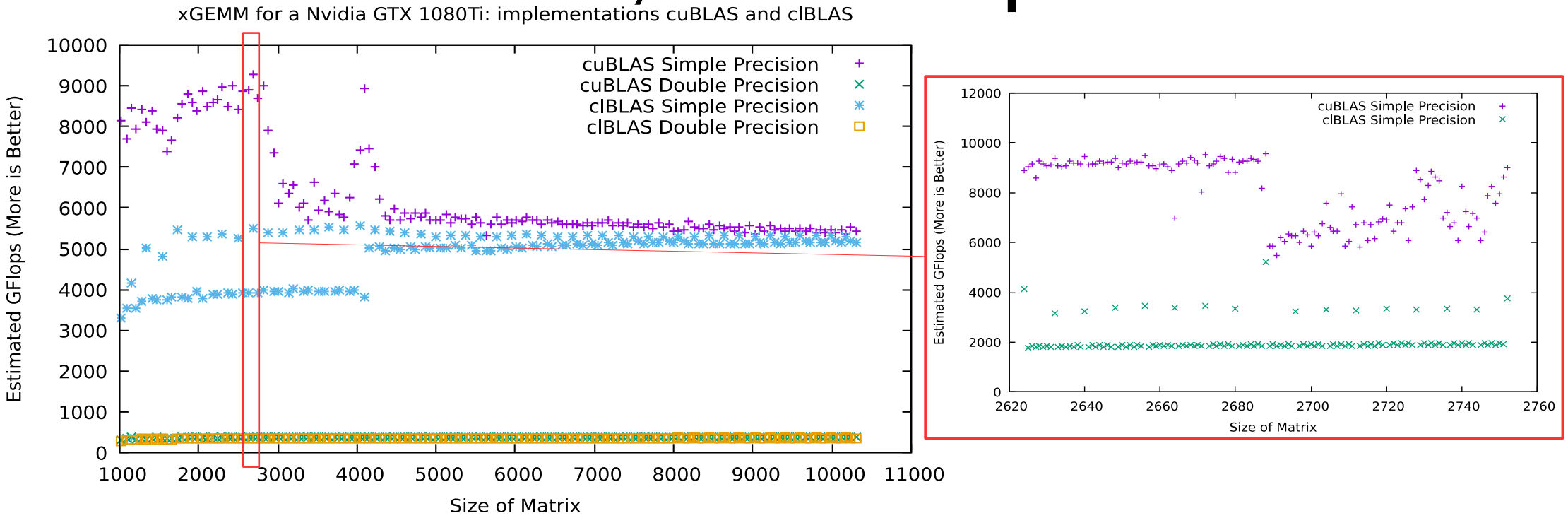




Taille

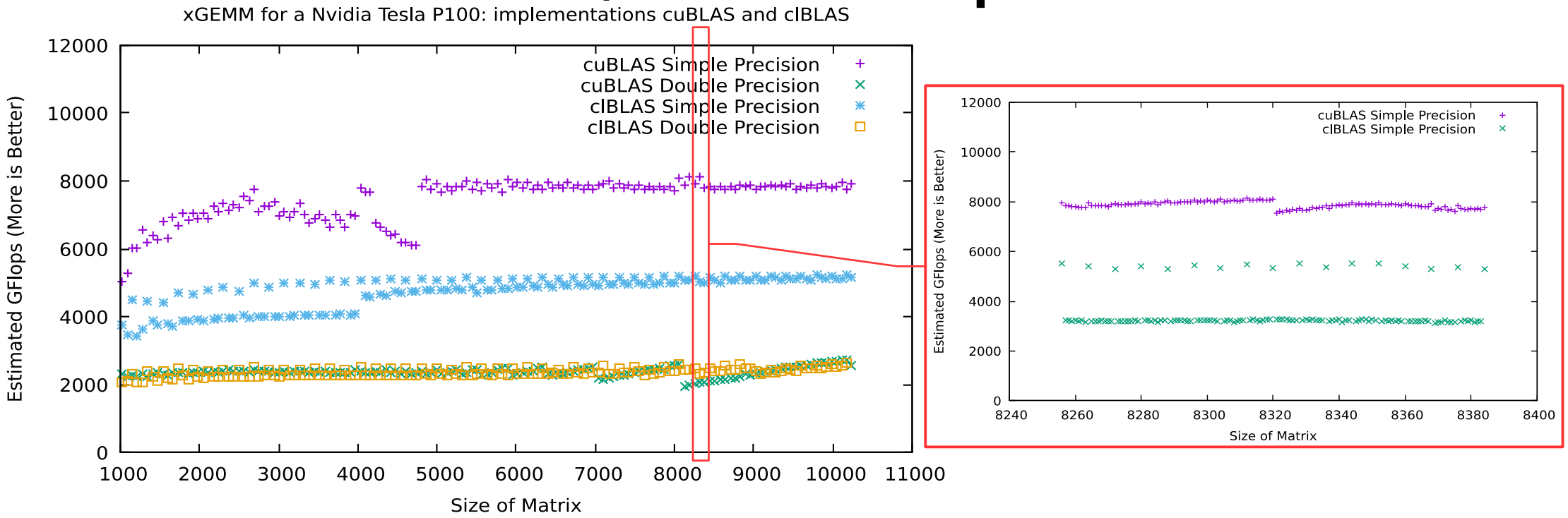
Oui, il y a la performance, mais dans certaines conditions...

Et pour l'ancienne « Gamer » Nvidia GTX 1080Ti, un max pour 2688



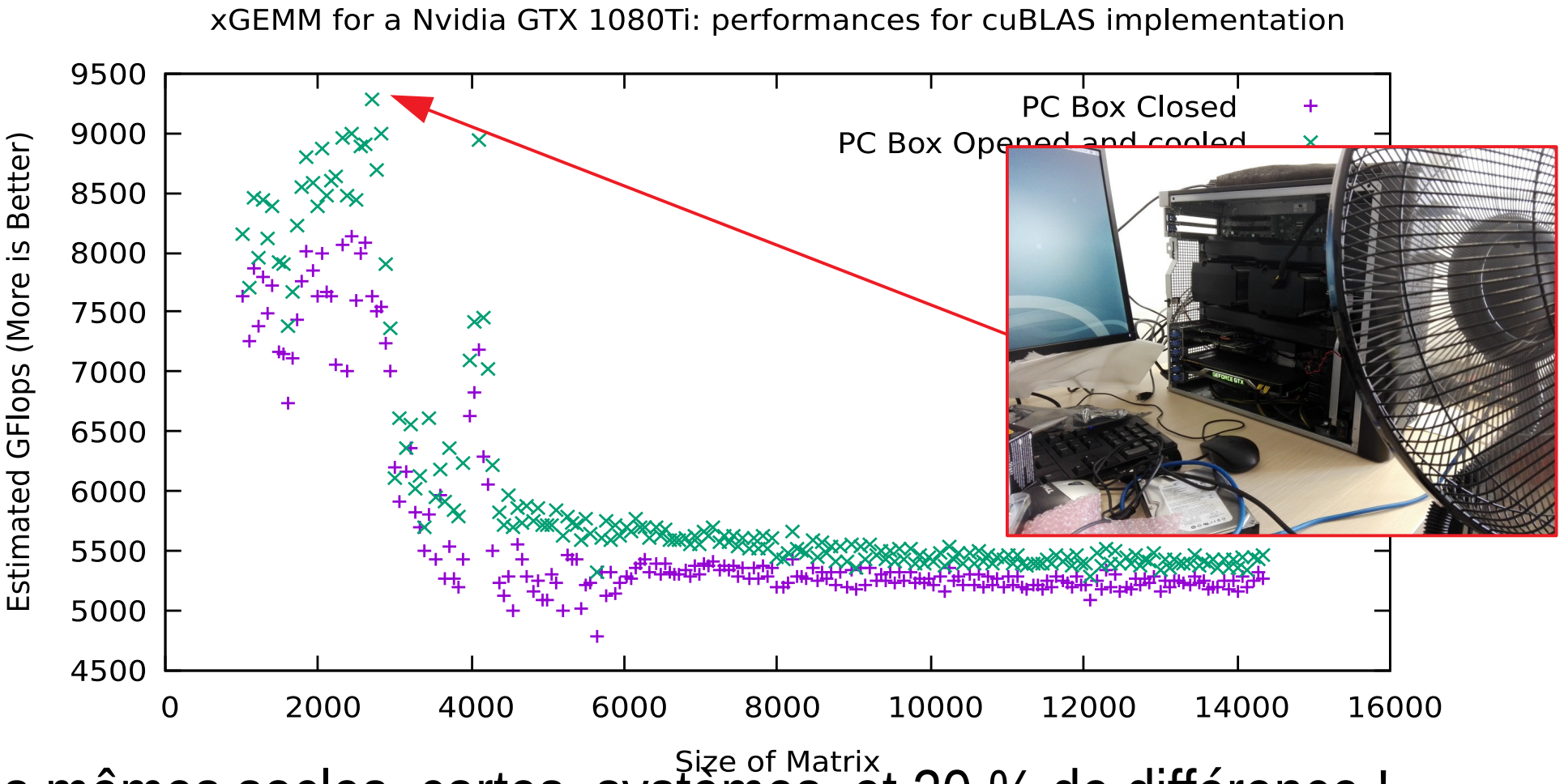
- Un facteur entre 20 et 30 entre simple & double précision
- Une implémentation clBLAS autour de la moitié pour les petites tailles, équivalente après
- Des effets de seuil (>2688 par exemple)
- Des effets arithmétiques (nombres premiers, par exemple 2671 ou 2713)

Et pour la « très chère » Nvidia Tesla P100, un max pour 8320



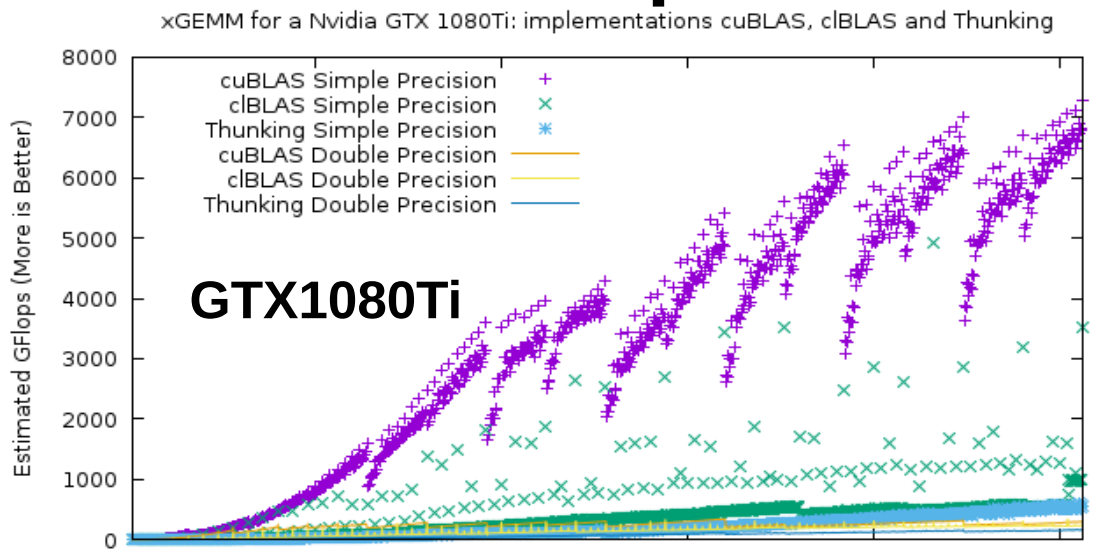
- Un facteur autour de 4 entre simple & double précision
- Une implémentation clBLAS avec des périodicités étranges
- Quelques effets de seuil (>4096 par exemple)
- Moins d'effets arithmétiques

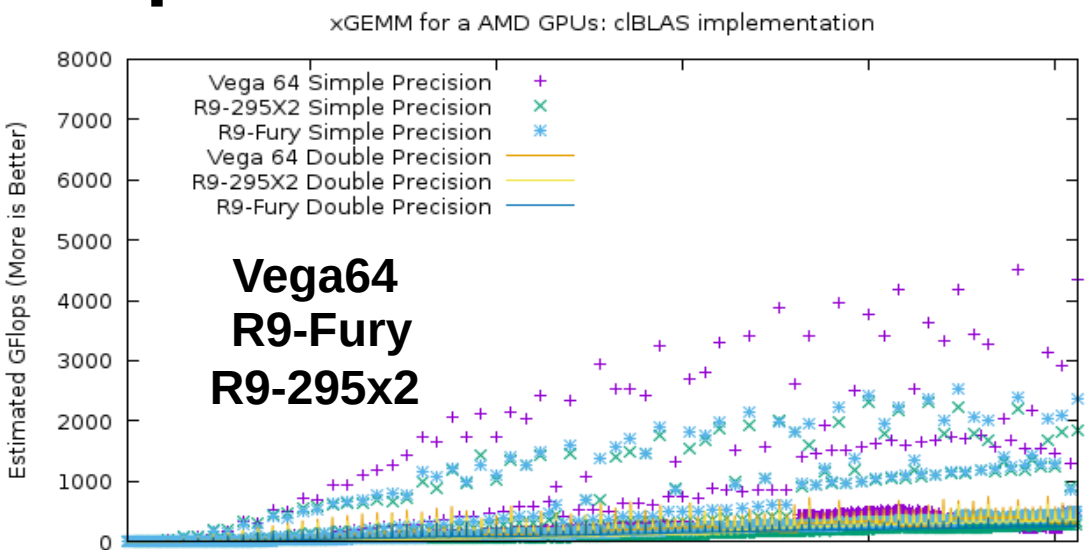
Mais il y a pire ! Pendant les tests de la GTX 1080 Ti

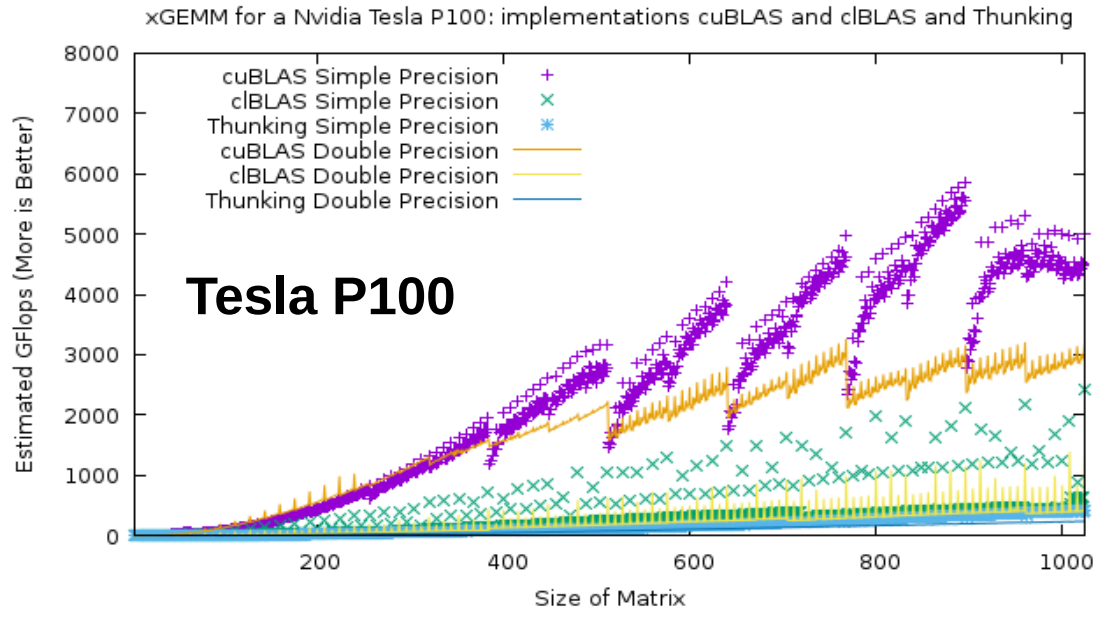


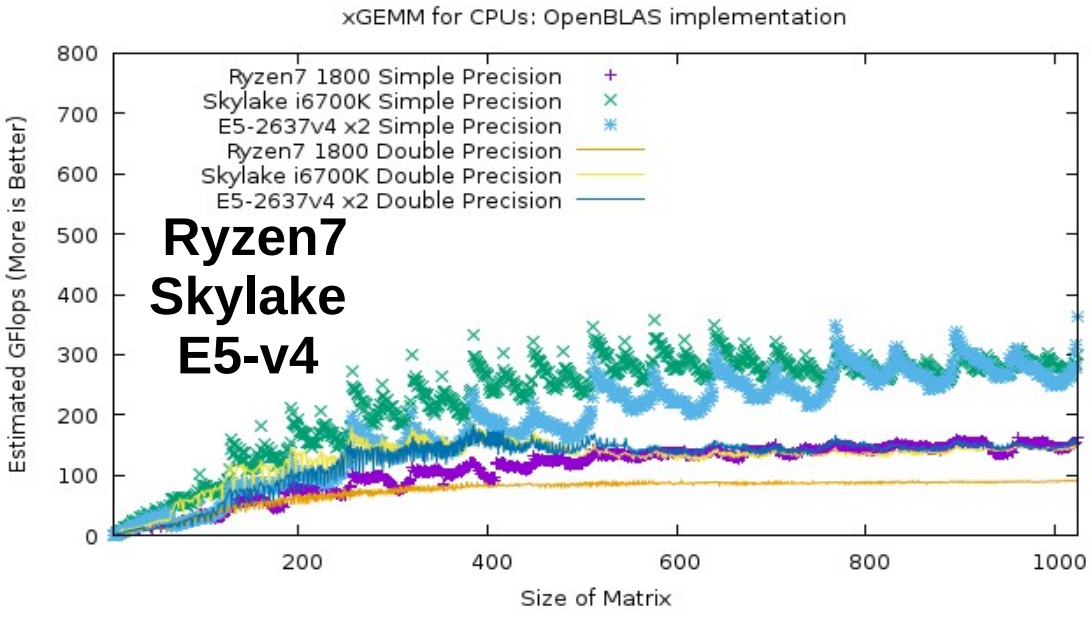
- Les mêmes socles, cartes, systèmes, et 20 % de différence !
  - De l'importance des conditions climatiques durant l'expérimentation...

xGEMM pour les « petites tailles »









Oui ! CuBLAS ou clBLAS, c'est bien, même en DP, mais...

Conclusion préliminaire de BLAS Exploitation de xGEMM

- Les GPU très supérieurs aux CPU
  - Mais attention à la non-continuité des performances
- Plus la famille du GPU est récente, mieux c'est
- L'implémentation clBLAS est crédible
  - Mais seulement pour certaines valeurs...
- Question : quid des « autres » fonctions BLAS ?

Un banc de test sans xGEMM
Assemblage de BLAS

- Banc d'essai : créer un système et le résoudre...
  - Formation d'une matrice aléatoire A conditionnée et d'un vecteur Y
  - Boucle pour chaque itération i :
    - Application $Y \leftarrow A.X$: fonction xGEMV
    - Application $Y \leftarrow A^{-1}.Y$: fonction xTRSV
    - Application $Y \leftarrow -Y+X$: fonction xAXPY
    - Application C[i] = somme normée de y : fonction xNRM2
    - Application $X \leftrightarrow Y$: fonction xSWAP
- Bon maintenant il fait l'intégrer !

Construction d'une approche par « intégration »

- « Bottom-up » : apprentissage // des xBLAS
  - J'ai des fonctions élémentaires d'algèbre linéaire
  - Je les utilise pour établir mon bench :
    - Je génère un vecteur X de dimension N
    - Je génère une matrice triangulaire A de dimension NxN
    - Je commence ma boucle
      - J'applique l'opération A.X qui donne Y
      - Je résouds le système pour retrouver X' tel que : A.X=Y
      - Je compare X à X' par leur différence et je stocke la somme normée (l'erreur cumulée) comme résultat
      - J'échange Y à X
  - Je programme en FBLAS
  - Je généralise en CBLAS et GSL en ajoutant des directives
  - Je programme en CuBLAS « use thunking »
  - Je programme en CuBLAS natif

Banc de test BLAS
A quoi ça ressemble dans les faits

Avec CBLAS

```
cblas_dgemv(CblasRowMajor,CblasNoTrans,dim,dim,alpha,A,dim,X,incx,beta,Y,incx);
cblas_dtrsv(CblasRowMajor,CblasUpper,CblasNoTrans,CblasNonUnit, dim,A,dim,Y,incx);
cblas_daxpy(dim,beta2,Y,incx,X,incx);
checksA[i]=(double)cblas_dnrm2(dim,X,incx);
cblas_dswap(dim,X,incx,Y,incx);
```

Avec FBLAS

```
dgemv_(&trans,&dim,&dim,&alpha,A,&dim,X,&incx,&beta,Y,&incx);
dtrsv_(&uplo,&trans,&diag,&dim,A,&dim,Y,&incx);
daxpy_(&dim,&beta2,Y,&incx,X,&incx);
dnrm2_(&dim,X,&incx,&checksA[i]);
dswap_(&dim,X,&incx,Y,&incx);
```

Avec CuBLAS version « use Thunking »

```
CUBLAS_DGEMV(&trans,&dim,&dim, &alpha,A,&dim,X,&incx,&beta,Y,&incx);
CUBLAS_DTRSV(&uplo,&trans,&diag,&dim,A,&dim,Y,&incx);
CUBLAS_DAXPY(&dim,&beta2,Y,&incx,X,&incx);
checksA[i]=(double)CUBLAS_DNRM2(&dim,X,&incx);
CUBLAS_DSWAP(&dim,X,&incx,Y,&incx);
```

Et pour le CUDA natif ?

Déclaration, réservation et copie dans GPU

```
stat1=cublasAlloc(dim*dim,sizeof(devPtrA[0]),(void**)&devPtrA);
stat2=cublasAlloc(dim,sizeof(devPtrX[0]),(void**)&devPtrX);
stat3=cublasAlloc(dim,sizeof(devPtrY[0]),(void**)&devPtrY);
if ((stat1 != CUBLAS_STATUS_SUCCESS) ||
    (stat2 != CUBLAS_STATUS_SUCCESS) ||
    (stat3 != CUBLAS_STATUS_SUCCESS)) {
  wrapperError ("Strsv", CUBLAS_WRAPPER_ERROR_ALLOC);
  cublasFree (devPtrA);
  cublasFree (devPtrX);
  cublasFree (devPtrY);
  return;
}
stat1=cublasSetMatrix(dim,dim,sizeof(A[0]),A,dim,devPtrA,dim);
stat2=cublasSetVector(dim,sizeof(X[0]),X,incx,devPtrX,incx);
stat3=cublasSetVector(dim,sizeof(Y[0]),Y,incx,devPtrY,incx);
if ((stat1 != CUBLAS_STATUS_SUCCESS) ||
    (stat2 != CUBLAS_STATUS_SUCCESS) ||
    (stat3 != CUBLAS_STATUS_SUCCESS)) {
  wrapperError ("Strsv", CUBLAS_WRAPPER_ERROR_SET);
  cublasFree (devPtrA);
  cublasFree (devPtrX);
  cublasFree (devPtrY);
  return;
}
```

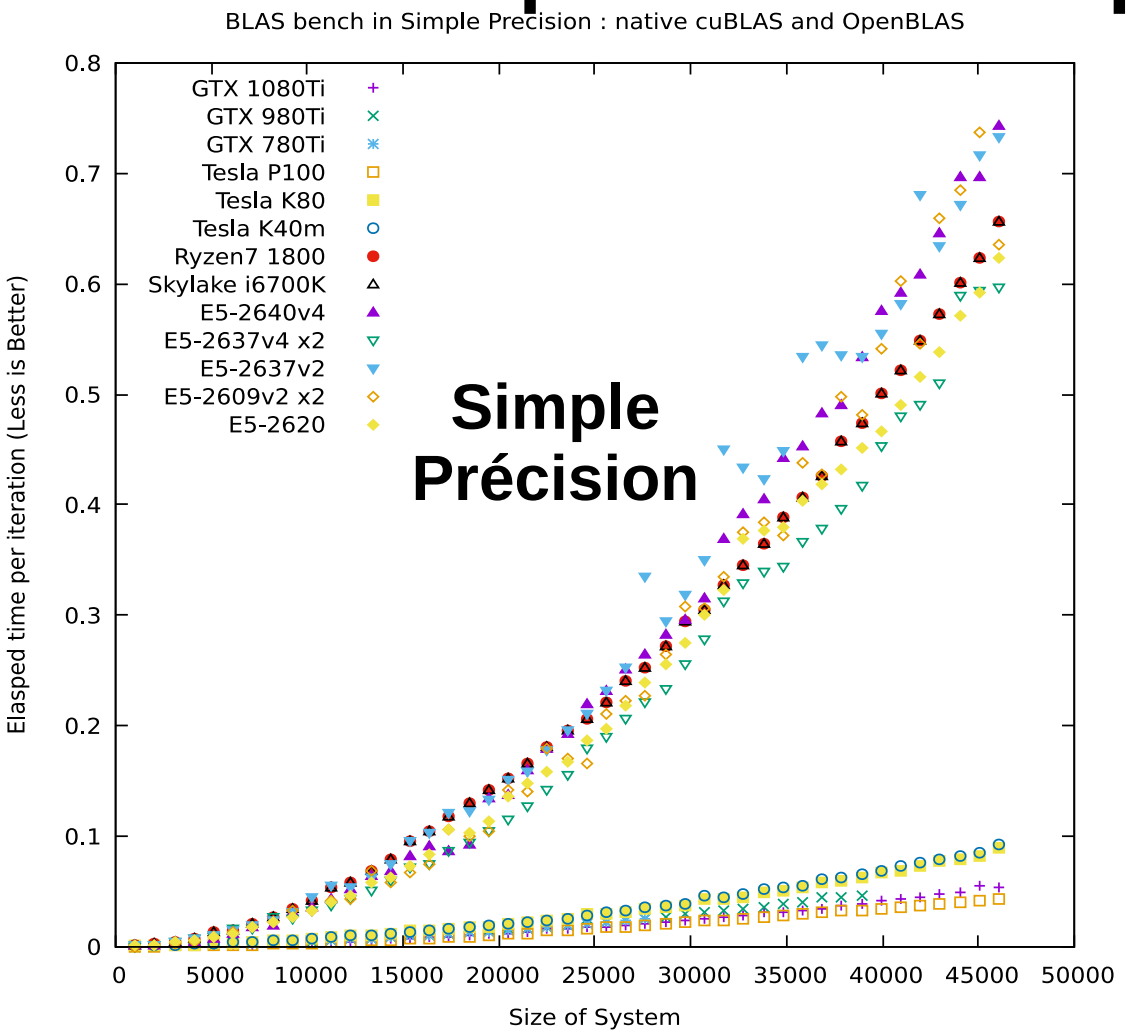
Calcul

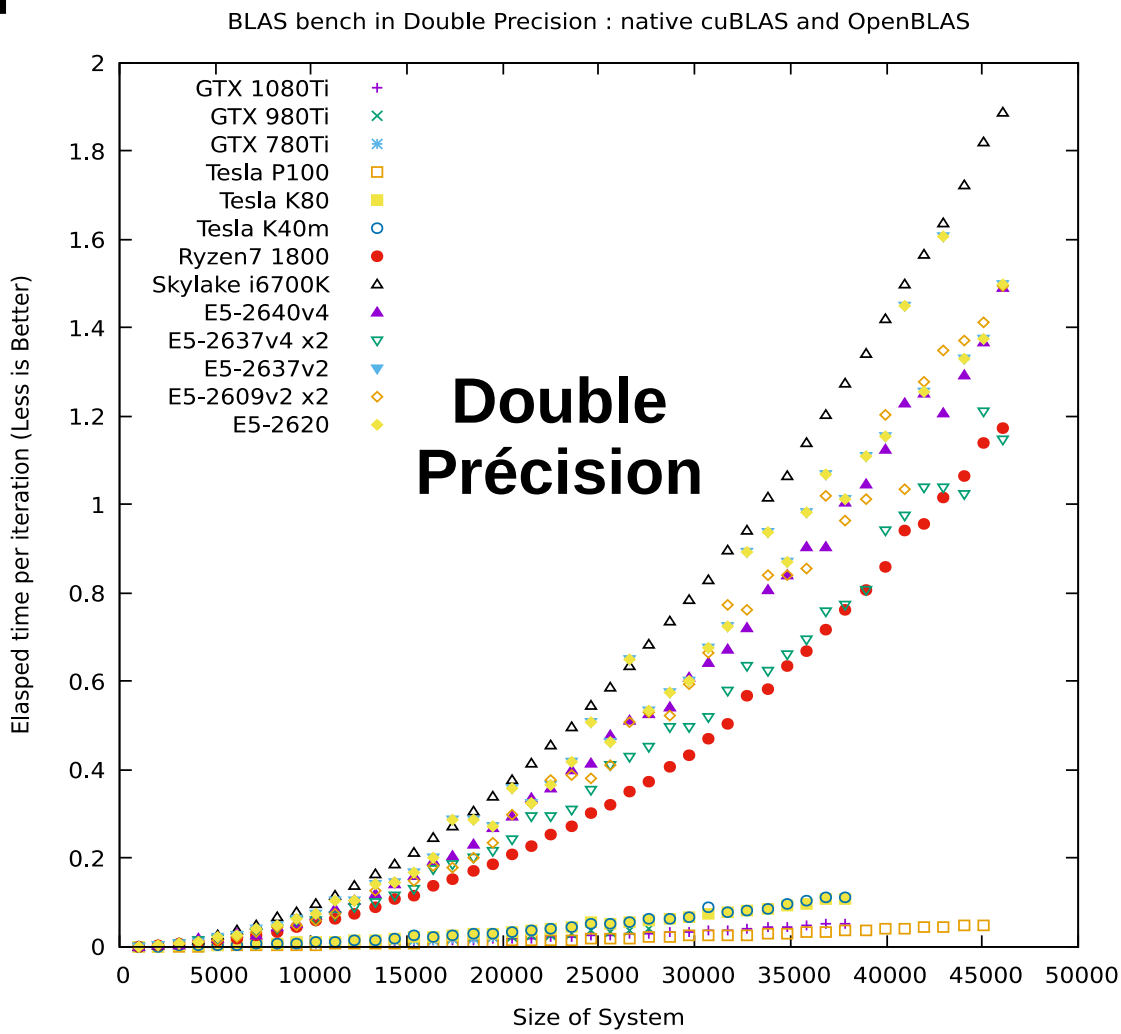
```
cublasDgemv(trans,dim,dim,alpha,devPtrA,dim,
                  devPtrX,incx,beta,devPtrY,incx);
cublasDtrsv(uplo,trans,diag,dim,devPtrA,dim,
                devPtrY,incx);
cublasDaxpy(dim,beta2,devPtrY,incx,devPtrX,incx);
checksA[i]=(double)cublasDnrm2(dim,devPtrX,incx);
cublasDswap(dim,devPtrX,incx,devPtrY,incx);
```

Copie résultats, libération GPU

```
stat1=cublasGetVector(dim,sizeof(X[0]),devPtrX,
                      incx,X,incx);
stat2=cublasGetVector(dim,sizeof(Y[0]),devPtrY,
                      incx,Y,incx);
  cublasFree (devPtrA);
  cublasFree (devPtrX);
  cublasFree (devPtrY);
  if ((stat1 != CUBLAS_STATUS_SUCCESS) ||
      (stat2 != CUBLAS_STATUS_SUCCESS)) {
    wrapperError ("Strsv",
                  CUBLAS_WRAPPER_ERROR_GET);
```

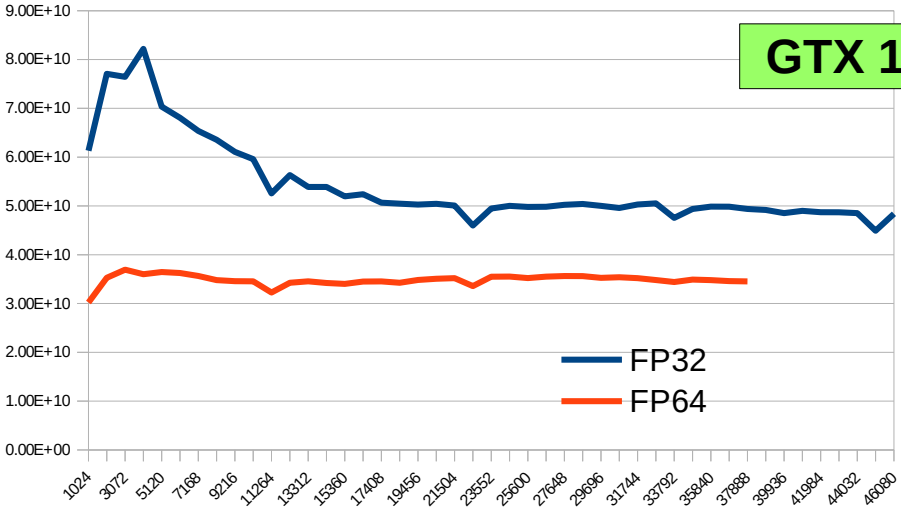
Les premiers résultats…
En temps écoulé par itération

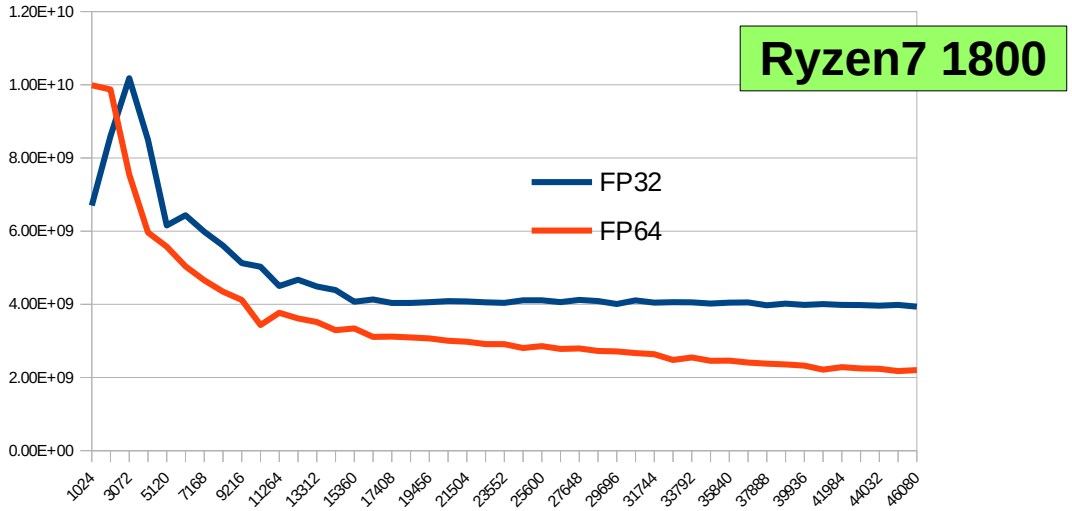


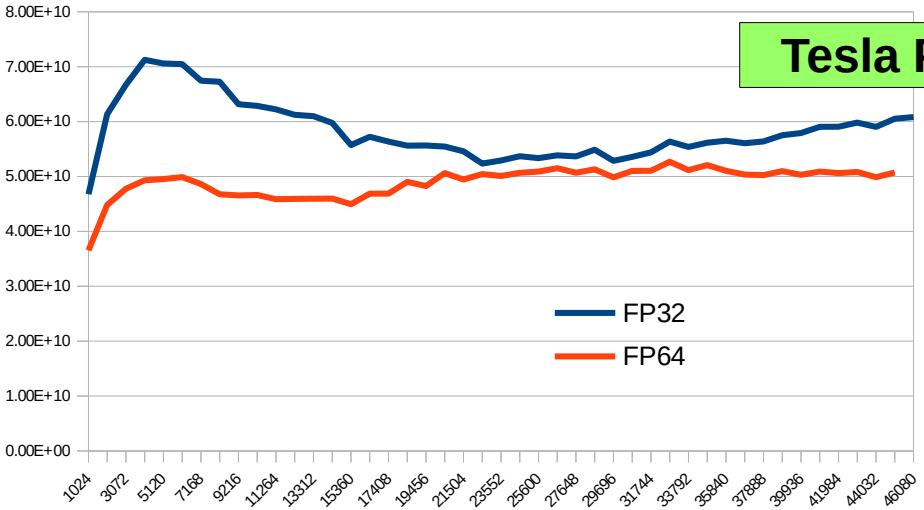


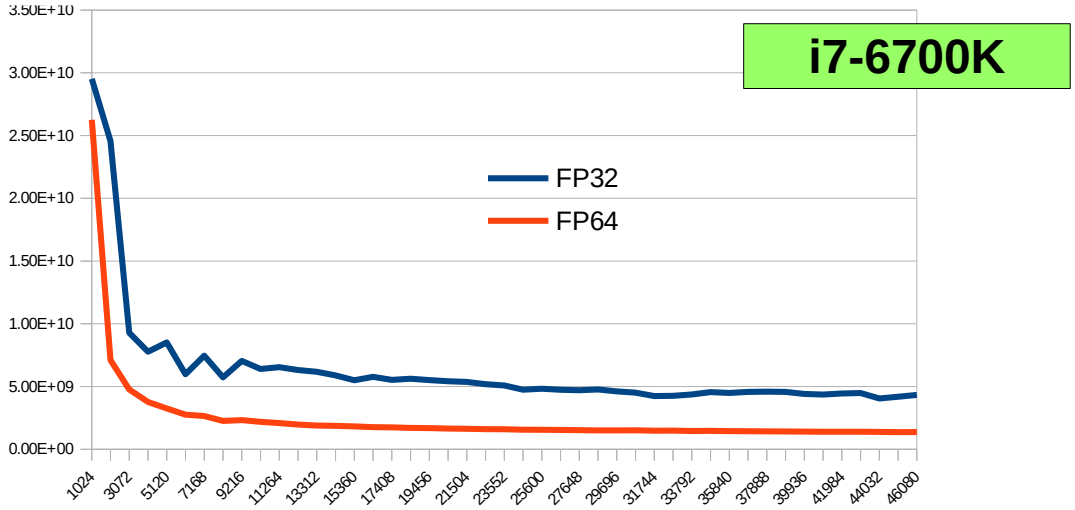
- Le problème de l'observable : réduire la courbe à une unique valeur...

Résultats « bench BLAS »
Fonction : Size²/T



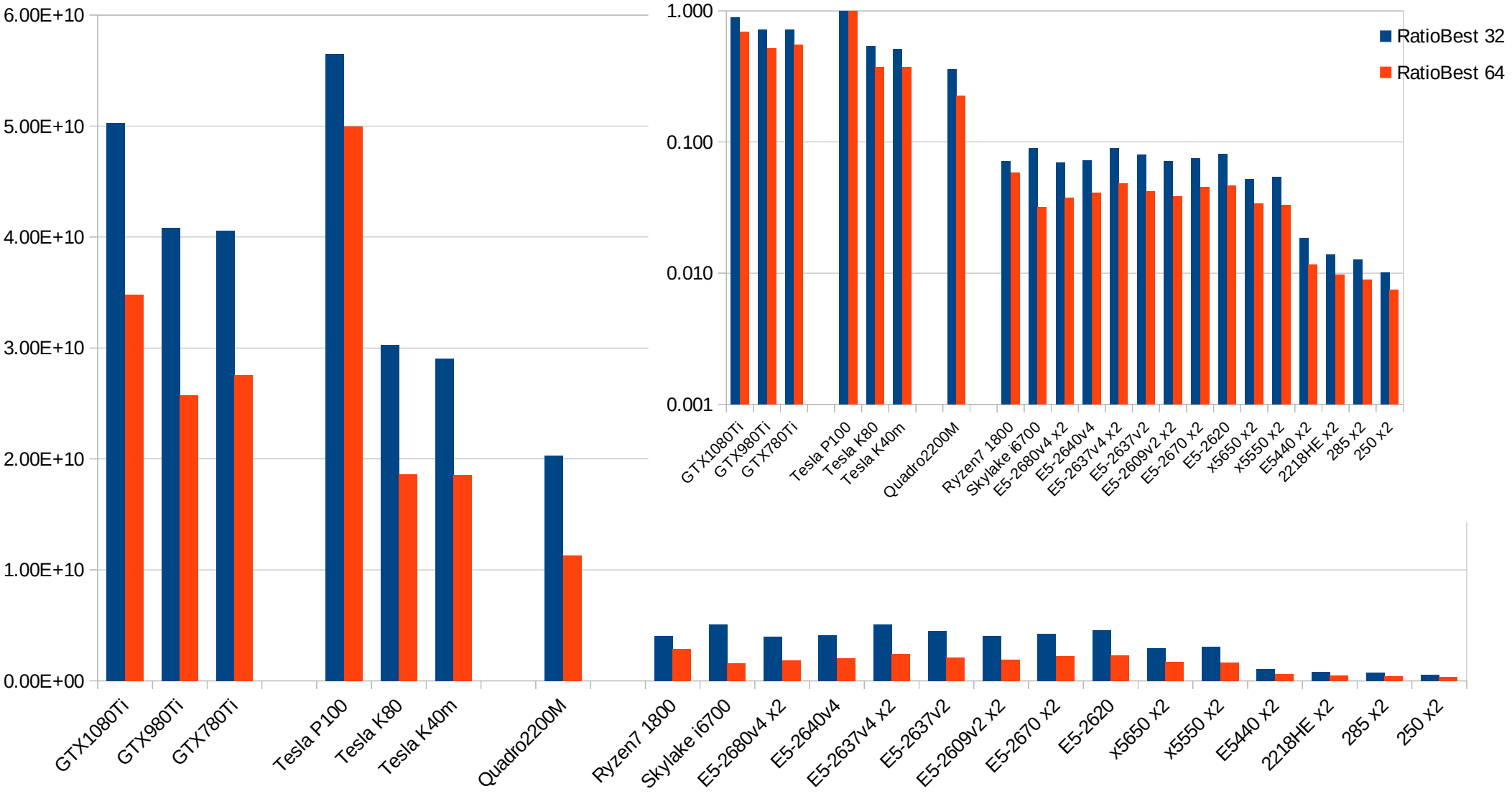






- Indicateur de « performances » : Médiane...

Bench BLAS en « more is better »
GPU vs CPU : un x10...



BLAS : conclusion

- CPU : des valeurs comparables pour tous
  - Quelque soit la fréquence ou le nombre de coeurs
- GPU autour de 10x plus rapide
  - Mais les circuits « Pascal » ont mis un coup de « boost »
- Bande passante mémoire prédominante
  - Mais RAM limitée à 40GB pour l'instant
- Préconisation : contruire son « banc d'essai »

Descendre encore un cran L'approche « développeur »

- Jusqu'à présent :
  - Code « métier » : de 2x à 5x en vitesse
  - Approche « intégrateur » : librairies optimisées facteur 10x
- Maintenant, quels codes pour « toucher la bête » :
  - Un code « gros grain », code « ALU » : **Pi Dart Dash**
  - Un code « grain fin », code « mémoire » : **Nbody**
- Deux buts : comparer GPU vs CPU et GPU vs GPU

Qui utilise OpenCL ?
Dans les applications

- OS : Apple dans MacOSX
- « Grosses » applications :
  - Libreoffice
  - Ffmpeg
- Applications connues
  - Les Graphiques : photoscan,
  - Les Scientifiques : OpenMM, Gromacs, Lammps, Amber, …
- Des centaines de logiciels, librairies ! Mais un test indispensable...
  - https://www.khronos.org/opencl/resources/opencl-applications-using-opencl

Qu'est-ce que OpenCL offre 9 implémentations sur x86

- 3 implémentations pour CPU :
  - AMD : la première, l'originale, très proche de OpenMP en performance
  - Intel : très efficace pour certains régimes de parallélisme
  - PortableCL (POCL) : celle OpenSource, son seul atout...
- 4 implémentations pour GPU :
  - AMDGPU-Pro : pour les circuits AMD/ATI très récents
  - AMDGPU & Mesa Gallium : pour les AMD/ATI par trop anciennes, mais pas récentes
  - Nvidia : pour les ciruits Nvidia
  - « Beignet » : une Open Source pour les circuits Intel
- 1 implémentation pour Accélérateur : celle d'Intel pour Xeon Phi
- 1 implémentation pour FPGA : celle d'Intel pour les FPGA Altera
- Pour d'autres plates-formes, possibles (PowerVR & Mali sur ARM)

Pourquoi OpenCL (et pas CUDA) ? Une histoire politiquement incorrecte

- Autour de 2005, Apple a besoin de puissance pour MacOSX
  - Certains calculs peuvent être « décharger » sur les circuits Nvidia
  - CUDA existe comme un bon successeur à CG Toolkit
- Mais Apple ne veut pas être prisonnier !
  - (comme ses utilisateurs)
- Apple est à l'initiative du consortium Khronos
  - AMD, IBM, ARM viennent rappidement
  - … et ensuite Nvidia, Intel Atlera les rejoignent...

Objectif : supprimer les boucles... 2 types de distributions

- Blocks/WorkItems
  - Domaine « global », de grande taille (~1 à 16GB) mais (relativement) lent !
- Threads
  - Domaine « local », de petite taille (~64KB) mais très rapide
  - Nécessité de sychronisation des processus
- Différents accès à la mémoire :
- « clinfo » pour récupérer les propriétés des devices CL devices :
  - Max work items : 1024x1024x1024 for CPU soit une distribution de 1.1 milliards

Comme l’utiliser ? Petit programme... Un « Hello World » en OpenCL...

- Définir deux vecteurs en ASCII
- Les dupliquer en 2 gros vecteurs
- Les additionner avec un noyau OpenCL
- Les imprimer à l’écran

Addition de

2 vecteurs a+b =c

Pour chaque n :

$c[n] = a[n] + b[n]$

Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL
World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello
OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World!
Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL
World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello
OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World!
Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL
World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello
OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World!
Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL
World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello
OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World! Hello OpenCL World!

The End

Code du noyau : Contruction & Appel Addition de vecteurs en OpenCL

```
__kernel void VectorAdd(__global int* c, __global int* a,__global int* b)
{
    // Index of the elements to add
    unsigned int n = get_global_id(0);
    // Sum the n th element of vectors a and b and store in c
    c[n] = a[n] + b[n];
}
```

```
OpenCLProgram = cl.Program(ctx, OpenCLSource).build()

OpenCLProgram.VectorAdd(queue, HostVector1.shape, None,GPUOutputVector ,
GPUVector1, GPUVector2)
```

Comment le faire en OpenCL ?
Écrire « Hello World ! » en C

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
const char* OpenCLSource[] = {
  "__kernel void VectorAdd(__global int* c, __global int* a,__global int* b)",
  "{",
  "       // Index of the elements to add \n",
  "       unsigned int n = get_global_id(0);",
  "       // Sum the n'th element of vectors a and b and store in c \n",
  "       c[n] = a[n] + b[n];",
  "}"
};
int InitialData1[20] = {37,50,54,50,56,0,43,43,74,71,32,36,16,43,56,100,50,25,15,17};
int InitialData2[20] = {35,51,54,58,55,32,36,69,27,39,35,40,16,44,55,14,58,75,18,15};
#define SIZE 2048
int main(int argc, char **argv)
{
  int HostVector1[SIZE], HostVector2[SIZE];
  for(int c = 0; c < SIZE; c++)
    {
      HostVector1[c] = InitialData1[c%20];
      HostVector2[c] = InitialData2[c%20];
    }
  cl_platform_id cpPlatform;
  clGetPlatformIDs(1, &cpPlatform, NULL);
  cl_int ciErr1;
  cl_device_id cdDevice;
  ciErr1 = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
  cl_context GPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
  cl_command_queue cqCommandQueue = clCreateCommandQueue(GPUContext,
cdDevice, 0, NULL);
  cl_mem GPUVector1 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,sizeof(int) *
SIZE, HostVector1, NULL);
  cl_mem GPUVector2 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,sizeof(int) *
SIZE, HostVector2, NULL);
  cl_mem GPUOutputVector = clCreateBuffer(GPUContext, CL_MEM_WRITE_ONLY,sizeof(int) * SIZE, NULL, NULL);
  cl_program OpenCLProgram = clCreateProgramWithSource(GPUContext, 7, OpenCLSource,NULL, NULL);
  clBuildProgram(OpenCLProgram, 0, NULL, NULL, NULL, NULL);
  cl_kernel OpenCLVectorAdd = clCreateKernel(OpenCLProgram, "VectorAdd", NULL);
  clSetKernelArg(OpenCLVectorAdd, 0, sizeof(cl_mem),(void*)&GPUOutputVector);
  clSetKernelArg(OpenCLVectorAdd, 1, sizeof(cl_mem), (void*)&GPUVector1);
  clSetKernelArg(OpenCLVectorAdd, 2, sizeof(cl_mem), (void*)&GPUVector2);
  size_t WorkSize[1] = {SIZE}; // one dimensional Range
  clEnqueueNDRangeKernel(cqCommandQueue, OpenCLVectorAdd, 1, NULL, WorkSize, NULL, 0, NULL, NULL);
  int HostOutputVector[SIZE];
  clEnqueueReadBuffer(cqCommandQueue, GPUOutputVector, CL_TRUE, 0,SIZE * sizeof(int), HostOutputVector,
0, NULL, NULL);
  clReleaseKernel(OpenCLVectorAdd);
  clReleaseProgram(OpenCLProgram);
  clReleaseCommandQueue(cqCommandQueue);
  clReleaseContext(GPUContext);
  clReleaseMemObject(GPUVector1);
  clReleaseMemObject(GPUVector2);
  clReleaseMemObject(GPUOutputVector);
  for (int Rows = 0; Rows < (SIZE/20); Rows++) {
      printf("\t");
      for(int c = 0; c <20; c++) {
printf("%c",(char)HostOutputVector[Rows * 20 + c]);
      }
  }
  printf("\n\nThe End\n\n");
  return 0;
}
```

Noyau OpenCL

Appel Noyau

Nombre de lignes du noyau OpenCL

Comment programmer en OpenCL ?
Écrire « Hello World ! » en Python

Noyau OpenCL

```
import pyopencl as cl
import numpy
import numpy.linalg as la
import sys
OpenCLSource = """
__kernel void VectorAdd(__global int* c, __global int* a,__global int* b)
{
      // Index of the elements to add
      unsigned int n = get_global_id(0);
      // Sum the n th element of vectors a and b and store in c
      c[n] = a[n] + b[n];
}
"""
InitialData1=[37,50,54,50,56,0,43,43,74,71,32,36,16,43,56,100,50,25,15,17]
InitialData2=[35,51,54,58,55,32,36,69,27,39,35,40,16,44,55,14,58,75,18,15]
SIZE=2048
HostVector1=numpy.zeros(SIZE).astype(numpy.int32)
HostVector2=numpy.zeros(SIZE).astype(numpy.int32)
for c in range(SIZE):
    HostVector1[c] = InitialData1[c%20]
    HostVector2[c] = InitialData2[c%20]
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
GPUVector1 = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=HostVector1)
GPUVector2 = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=HostVector2)
GPUOutputVector = cl.Buffer(ctx, mf.WRITE_ONLY, HostVector1.nbytes)
OpenCLProgram = cl.Program(ctx, OpenCLSource).build()
OpenCLProgram.VectorAdd(queue, HostVector1.shape,
None,GPUOutputVector , GPUVector1, GPUVector2)
HostOutputVector = numpy.empty_like(HostVector1)
cl.enqueue_copy(queue, HostOutputVector, GPUOutputVector)
GPUVector1.release()
GPUVector2.release()
GPUOutputVector.release()
OutputString=''
for rows in range(SIZE/20):
    OutputString+='\t'
    for c in range(20):
        OutputString+=chr(HostOutputVector[rows*20+c])
print OutputString
sys.stdout.write("\nThe End\n\n");
```

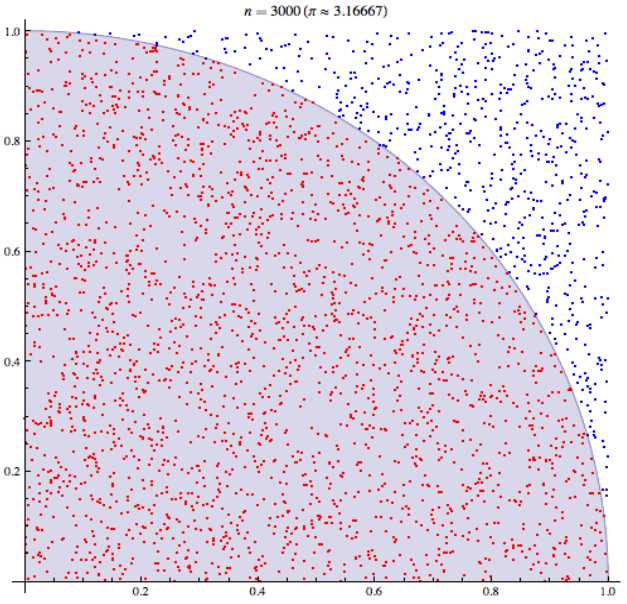
Appel Noyau

Comment faire en OpenCL ?
« Hello World » C/Python : la pesée

- Sur les implémentations OpenCL précédentes :
- On previous OpenCL implementations :
  - En C : 75 lignes, 262 mots, 2848 bytes
  - En Python : 51 lignes, 137 mots, 1551 bytes
  - Factors : 0.68, 0.52, 0.54 en lignes, mots et bytes.
- Programmer en OpenCL :
  - Principale difficulté : maîtriser le « contexte »
    - « Ouvrir la boîte est plus difficile qu'assembler le meuble ! »
    - Nécessité de simplifier les appels par une API
  - Pas de compatibilité entre les API de AMD et Nvidia
    - Chacun réécrit la sienne !
- Une solution, sinon « la » solution : Python !

Et pour des implémentations simples ?
PiMC : Pi by Dart Board Method

- Exemple classique du calcul de Monte Carlo
- Implémentation parallèle : distribution
  - De 2 à 4 paramètres
    - Nombre total d'itérations
    - Régime de Parallélisme (PR)
    - (Type de variable : INT32, INT64, FP32, FP64)
    - (RNG : MWC, CONG, SHR3, KISS)
  - 2 observables simples :
    - Estimation de Pi estimation (juste indicative, Pi n'est pas rationnel :-))
    - Temps écoulé



Pas terrible comme programme ? Regardez un cours du LLNL ;-)

Introduction to GPU Parallel Programming

Data Heroes Summer HPC Workshop
June 27, 2016

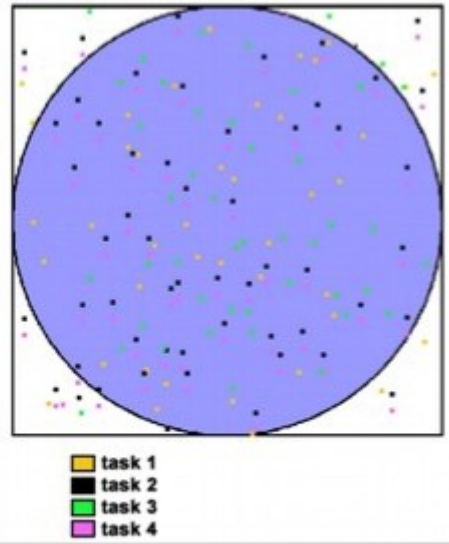
Donald Frederick,
Livermore Computing

Lawrence Livermore National Laboratory

LLNL-PRES-XXXXXX
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Approximation of Pi by Monte Carlo – Parallel Version

- Another problem that's easy to parallelize: All point calculations are independent; no data dependencies
- Work can be evenly divided; no load balance concerns
- No need for communication or synchronization between tasks
- Parallel strategy: Divide the loop into equal portions that can be executed by the pool of tasks
- Each task independently performs its work
- A SPMD model is used
- One task acts as the master to collect results and compute the value of PI



Lawrence Livermore National Laboratory

D’une version très monolithique… Variable & RNG fixés…

```
unsigned int jcong=seed_z+i;
#define CONG  (jcong=69069*jcong+1234567)
#define CONGfp CONG * 2.32830643545494e-10f
LENGTH total=0;
for (LENGTH i=0;i<iterations;i++) {
        float x=CONGfp ;
        float y=CONGfp ;
        unsigned long inside=((x*x+y*y) < 1.0f) ? 1:0;
        total+=inside;
}
Inside(i)=total;
```

Et ça donne quoi comme code ? Pour les 4x4 implémentations...

```
#define znew  ((z=36969*(z&65535)+(z>>16))<<16)
#define wnew  ((w=18000*(w&65535)+(w>>16))&65535)
#define MWC   (znew+wnew)
#define SHR3  (jsr=(jsr=(jsr=jsr^(jsr<<17))^(jsr>>13))^(jsr<<5))
#define CONG  (jcong=69069*jcong+1234567)
#define KISS  ((MWC^CONG)+SHR3)
#define MWCfp MWC * 2.328306435454494e-10f
#define KISSfp KISS * 2.328306435454494e-10f
#define SHR3fp SHR3 * 2.328306435454494e-10f
#define CONGfp CONG * 2.328306435454494e-10f
#if defined TCONG
  unsigned int jcong=seed_z+i;
#elif defined TSHR3
  unsigned int jsr=seed_w+i;
#elif defined TMWC
  unsigned int z=seed_z+i;
  unsigned int w=seed_w-i;
#elif defined TKISS
  unsigned int jcong=seed_z+i;
  unsigned int jsr=seed_w-i;
  unsigned int z=seed_z+i;
  unsigned int w=seed_w-i;
#endif
  LENGTH total=0;
  for (LENGTH i=0;i<iterations;i++) {
```

```
#if defined TINT32
  #define THEONE 1073741824
  #if defined TCONG
    unsigned int x=CONG>>17 ;
    unsigned int y=CONG>>17 ;
  #elif defined TSHR3
    unsigned int x=SHR3>>17 ;
    unsigned int y=SHR3>>17 ;
  #elif defined TMWC
    unsigned int x=MWC>>17 ;
    unsigned int y=MWC>>17 ;
  #elif defined TKISS
    unsigned int x=KISS>>17 ;
    unsigned int y=KISS>>17 ;
  #endif
#elif defined TINT64
  #define THEONE 4611686018427387904
  #if defined TCONG
    unsigned long x=(unsigned long)(CONG>>1) ;
    unsigned long y=(unsigned long)(CONG>>1) ;
  #elif defined TSHR3
    unsigned long x=(unsigned long)(SHR3>>1) ;
    unsigned long y=(unsigned long)(SHR3>>1) ;
  #elif defined TMWC
    unsigned long x=(unsigned long)(MWC>>1) ;
    unsigned long y=(unsigned long)(MWC>>1) ;
  #elif defined TKISS
    unsigned long x=(unsigned long)(KISS>>1) ;
    unsigned long y=(unsigned long)(KISS>>1) ;
  #endif
```

```
#elif defined TFP32
  #define THEONE 1.0f
  #if defined TCONG
    float x=CONGfp ;
    float y=CONGfp ;
  #elif defined TSHR3
    float x=SHR3fp ;
    float y=SHR3fp ;
  #elif defined TMWC
    float x=MWCfp ;
    float y=MWCfp ;
  #elif defined TKISS
   float x=KISSfp ;
   float y=KISSfp ;
  #endif
#elif defined TFP64
  #define THEONE 1.0f
  #if defined TCONG
    double x=(double)CONGfp ;
    double y=(double)CONGfp ;
  #elif defined TSHR3
    double x=(double)SHR3fp ;
    double y=(double)SHR3fp ;
  #elif defined TMWC
    double x=(double)MWCfp ;
    double y=(double)MWCfp ;
  #elif defined TKISS
    double x=(double)KISSfp ;
    double y=(double)KISSfp ;
  #endif
#endif
```

```
  unsigned long inside=((x*x+y*y) < THEONE) ? 1:0;
  total+=inside;
}
Inside(i)=total;
```

Différences entre CUDA/OpenCL Les noyaux des calculs GPU

```
__device__ ulong MainLoop(ulong iterations,uint seed_w,uint seed_z,size_t work)
{
  uint jcong=seed_z+work;
  ulong total=0;
  for (ulong i=0;i<iterations;i++) {
      float x=CONGfp ;
      float y=CONGfp ;
    ulong inside=((x*x+y*y) <= THEONE) ? 1:0;
    total+=inside;
  }
  return(total);
}

__global__ void MainLoopBlocks(ulong *s,ulong iterations,uint seed_w,uint seed_z)
{
  ulong total=MainLoop(iterations,seed_z,seed_w,blockIdx.x);
  s[blockIdx.x]=total;
  __syncthreads();
}
```

```
ulong MainLoop(ulong iterations,uint seed_z,uint seed_w,size_t work)
{
  uint jcong=seed_z+work;
  ulong total=0;
  for (ulong i=0;i<iterations;i++) {
      float x=CONGfp ;
      float y=CONGfp ;
    ulong inside=((x*x+y*y) <= THEONE) ? 1:0;
    total+=inside;
  }
  return(total);
}
__kernel void MainLoopGlobal(__global ulong *s,ulong iterations,uint seed_w,uint seed_z)
{
  ulong total=MainLoop(iterations,seed_z,seed_w,get_global_id(0));
  barrier(CLK_GLOBAL_MEM_FENCE);
  s[get_global_id(0)]=total;
}
```

Xeon Phi a été mentionné : Pas un GPU, mort né, mais instructif...

- 3 voies de programmation
  - Compilateur Intel, cross-compiling, exécution dans un micro-système
  - Compilateur Intel, OpenMP en mode « offload », exécution transparente
  - Implémentation de OpenCL d'Intel
- Qu'est-ce que « l'offload » en OpenMP ?

OpenMP classique sur tâches indépendantes

```
#pragma omp parallel for
  for (int i=0 ; i<process; i++) {
inside[i]=MainLoopGlobal(iterations/process,seed_w+i,seed_z+i);
  }
```

Appel *Offload* sur Xeon Phi MIC

```
#pragma omp target device(0)
#pragma omp teams num_teams(60) thread_limit(4)
#pragma omp distribute
  for (int i=0 ; i<process; i++) {
inside[i]=MainLoopGlobal(iterations/process,seed_w+i,seed_z+i);
```

Et une implémentation en C/OpenCL
Pas vraiment simple… Mais attendez !

Sélection de la plate-forme et le périphérique

- err = clGetPlatformIDs(0, NULL, &platformCount);
- platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * platformCount);
- err = clGetPlatformIDs(platformCount, platforms, NULL);
- err = clGetDeviceIDs(platforms[MyPlatform], CL_DEVICE_TYPE_ALL, 0, NULL, &deviceCount);
- devices = (cl_device_id*) malloc(sizeof(cl_device_id) * deviceCount);
- err = clGetDeviceIDs(platforms[MyPlatform], CL_DEVICE_TYPE_ALL, deviceCount, devices, NULL);
- cl_context GPUContext = clCreateContext(props, 1, &devices[MyDevice], NULL, NULL, &err);
- cl_command_queue cqCommandQueue = clCreateCommandQueue(GPUContext,devices[MyDevice], 0, &err);
- cl_mem GPUInside = clCreateBuffer(GPUContext, CL_MEM_WRITE_ONLY, sizeof(uint64_t) * ParallelRate, NULL, NULL);
- cl_program OpenCLProgram = clCreateProgramWithSource(GPUContext, 130 ,OpenCLSource,NULL,NULL);
- clBuildProgram(OpenCLProgram, 0, NULL, NULL, NULL, NULL);
- cl_kernel OpenCLMainLoopGlobal = clCreateKernel(OpenCLProgram, "MainLoopGlobal", NULL);

Définition des attributs du noyau à appeler

- clSetKernelArg(OpenCLMainLoopGlobal, 0, sizeof(cl_mem),&GPUInside);
- clSetKernelArg(OpenCLMainLoopGlobal, 1, sizeof(uint64_t),&IterationsEach);
- clSetKernelArg(OpenCLMainLoopGlobal, 2, sizeof(uint32_t),&seed_w);
- clSetKernelArg(OpenCLMainLoopGlobal, 3, sizeof(uint32_t),&seed_z);
- clSetKernelArg(OpenCLMainLoopGlobal, 4, sizeof(uint32_t),&MyType);
- size_t WorkSize[1] = {ParallelRate}; // one dimensional Range

- clEnqueueNDRangeKernel(cqCommandQueue, OpenCLMainLoopGlobal, 1, NULL, WorkSize, NULL, 0, NULL, NULL);
- clEnqueueReadBuffer(cqCommandQueue, GPUInside, CL_TRUE, 0, ParallelRate * sizeof(uint64_t), HostInside, 0, NULL, NULL);

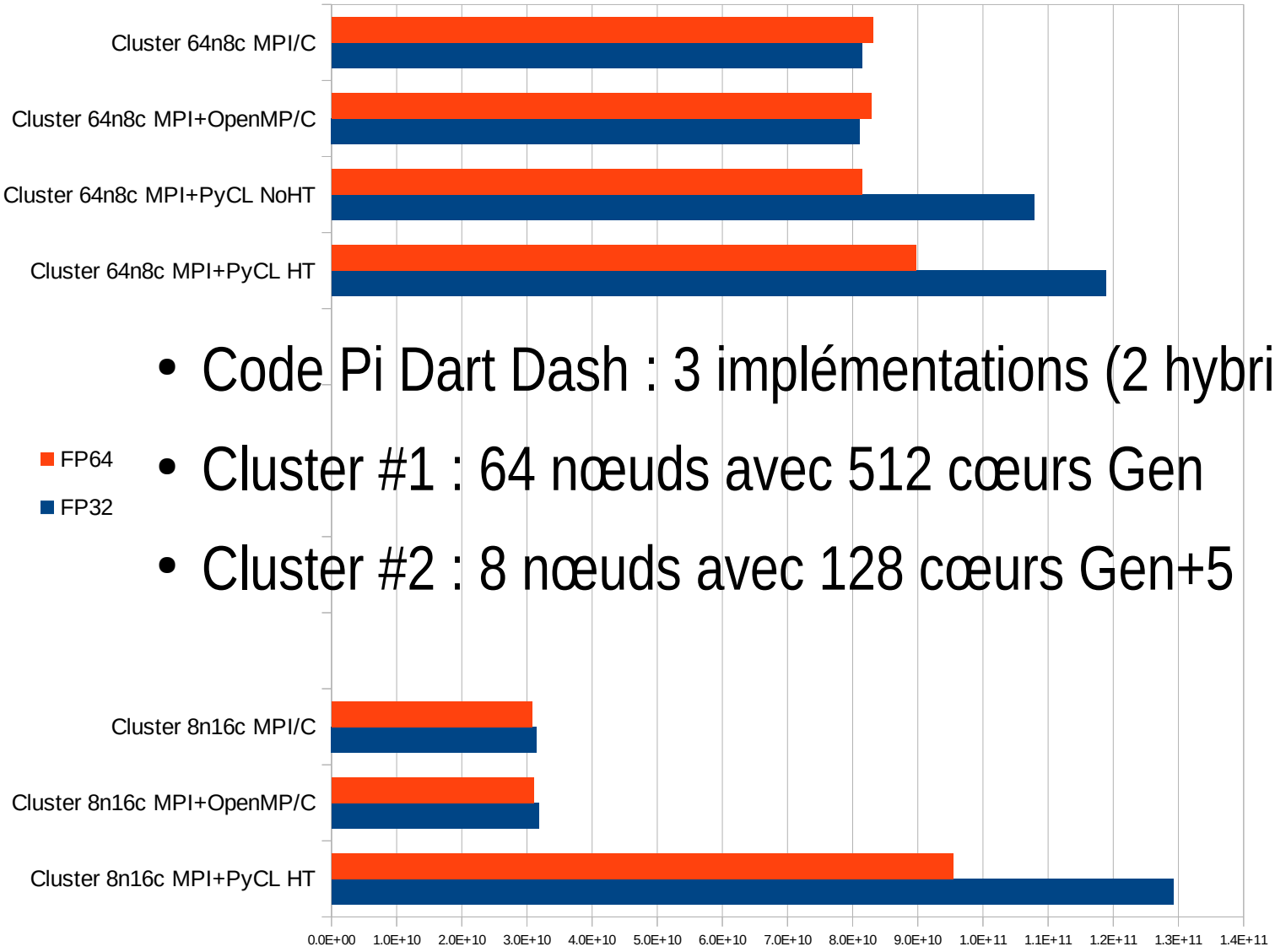
Et en OpenACC, ça donne quoi ? Très simple implémentation...

- Avant la routine « cœur » du programme :
  - #pragma acc routine
  - LENGTH MainLoopGlobal(LENGTH iterations,unsigned int seed_w,unsigned int seed_z)
- Avant la distribution des boucles
  - #pragma omp parallel for shared(ParallelRate,inside)
  - #pragma acc kernels loop
  - for (int i=0 ; i<ParallelRate; i++) {
  - inside[i]=MainLoopGlobal(IterationsEach,seed_w+i,seed_z+i); }

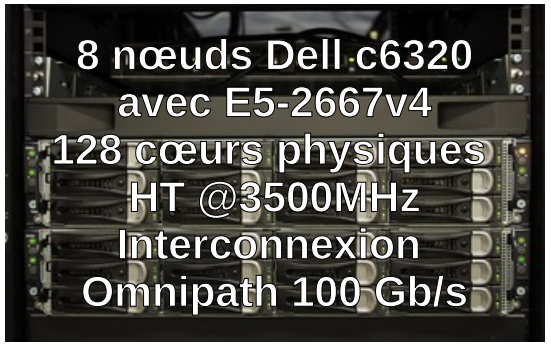
Et en KOKKOS, ça donne quoi ?

- Avant la routine « cœur » du programme :
  - struct splitter {
  - // initialisation classe
  - KOKKOS_INLINE_FUNCTION
  - void operator() (int i) const {
  - // Toute la routine MainLoopGlobal
- Initialisation de l'environnement KOKKOS
  - Kokkos::initialize (argc, argv);
- Lors de la distribution & de la réduction
  - Kokkos::parallel_for (ParallelRate,splitter(Inside,IterationsEach,seed_w,seed_z));
  - Kokkos::parallel_reduce (ParallelRate, ReduceFunctor (Inside), insides);
- Sortie de l'environnement
  - Kokkos::finalize ();

Sur des clusters ? Python efficace ? Une rapide comparaison avec GNU

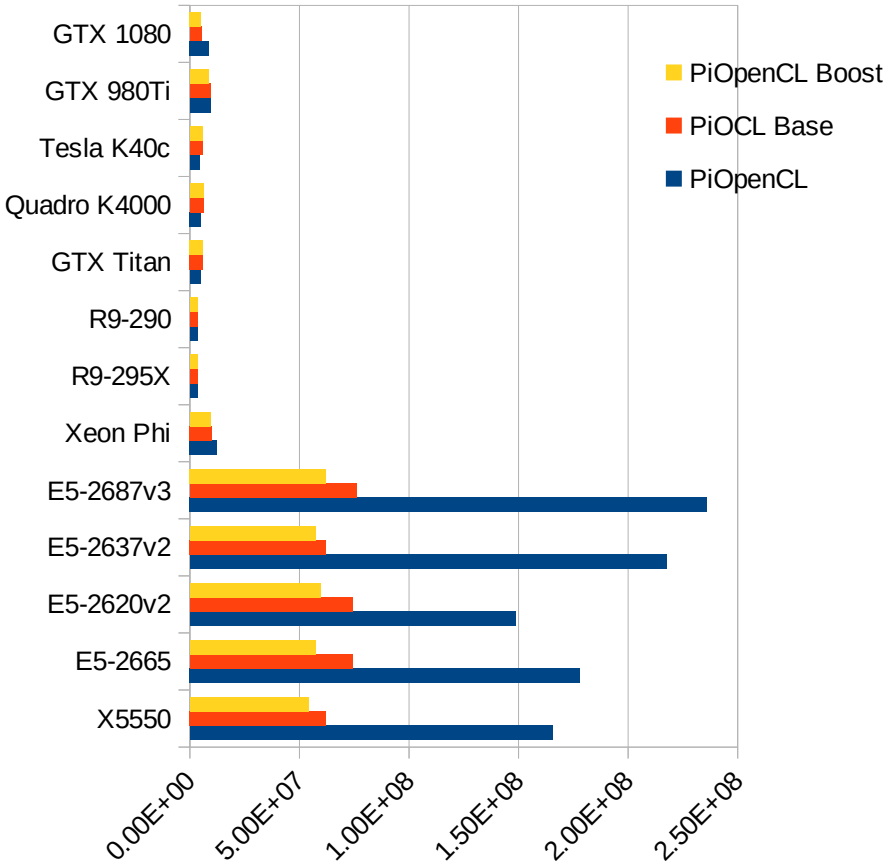


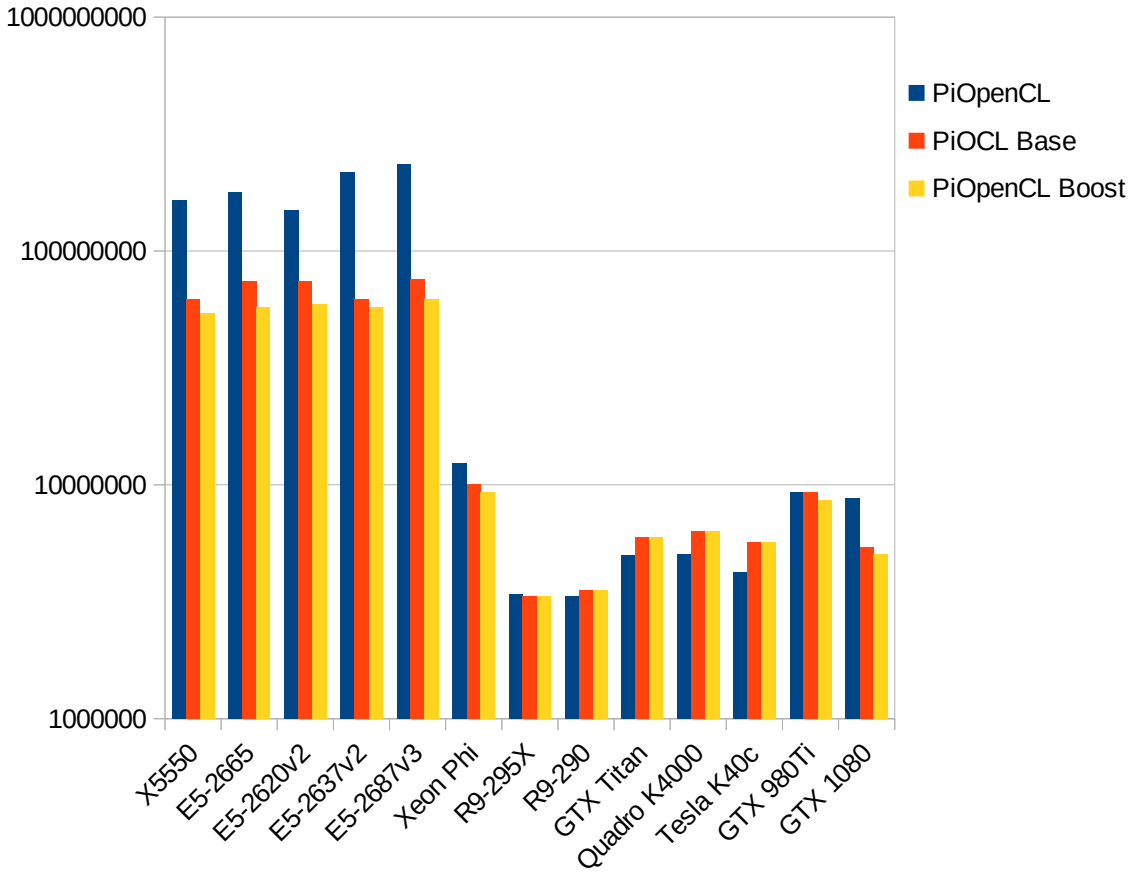
- Code Pi Dart Dash : 3 implémentations (2 hybrides)
- Cluster #1 : 64 nœuds avec 512 cœurs Gen
- Cluster #2 : 8 nœuds avec 128 cœurs Gen+5



Pour 1 de PR, bas régime, pour Pi
Le CPU explose le GPU!

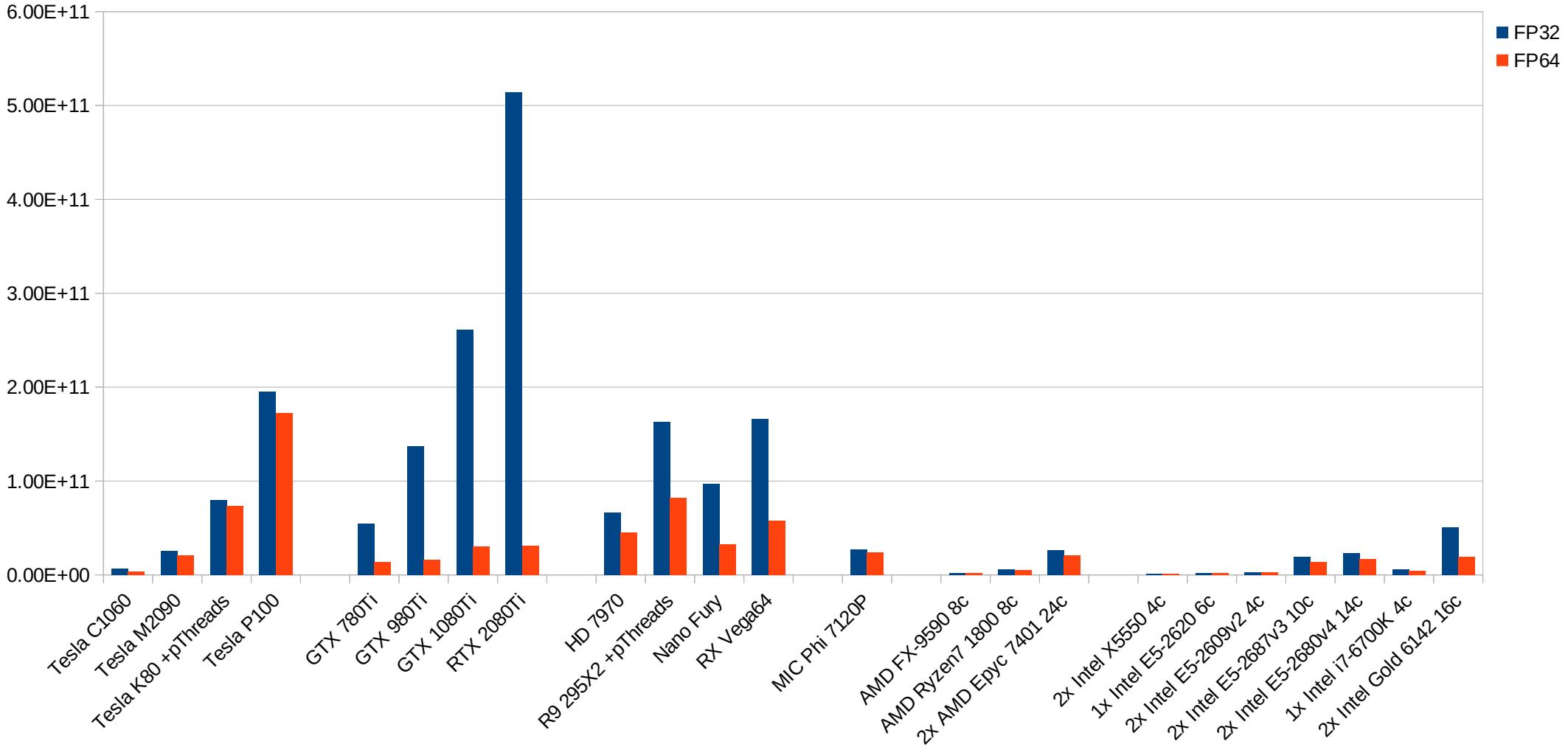
- De 20 à 50x plus lent !





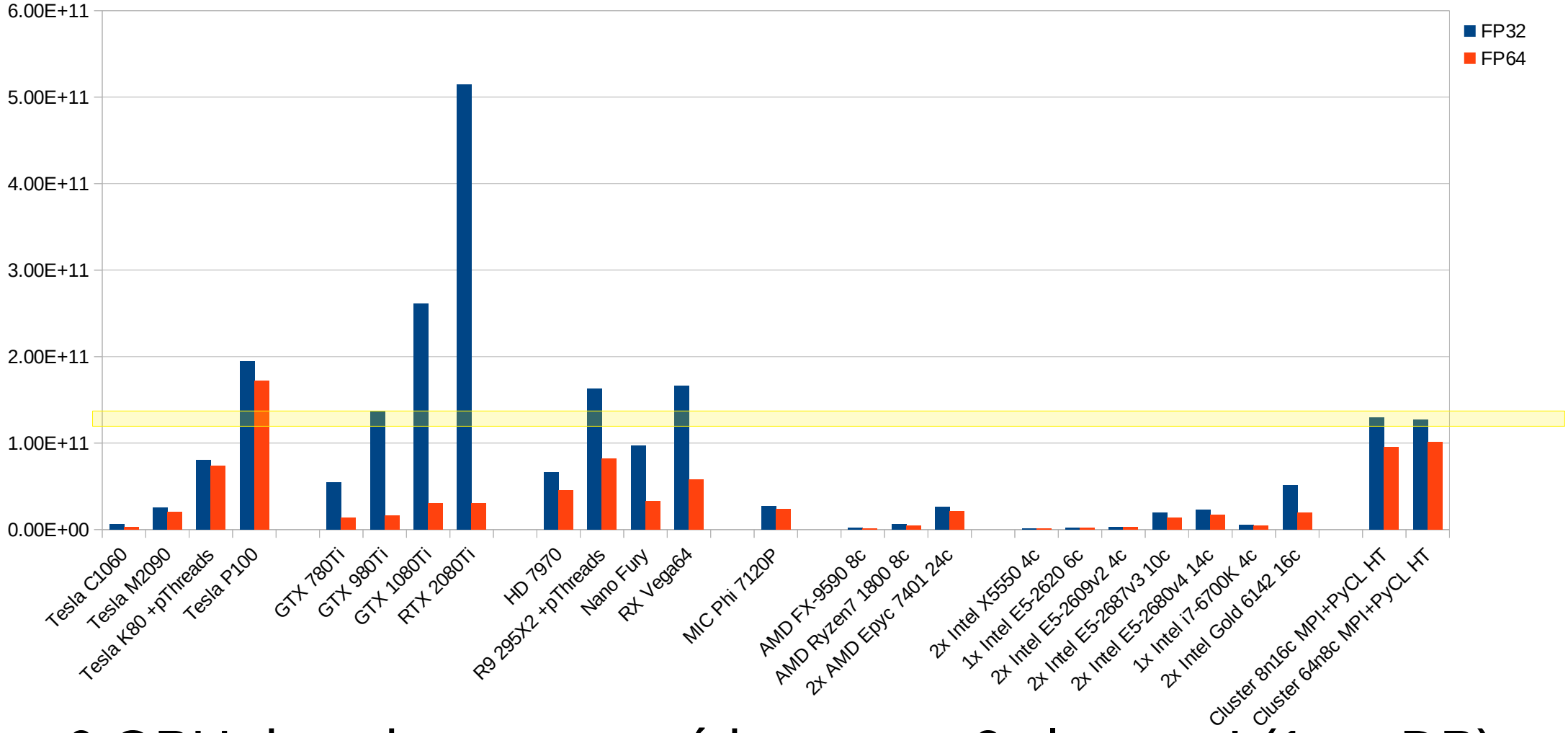
- 1.5 ordre de magnitude

Pi Dart Dash, pour nos cobayes…
Avec un code en Python multiGPU



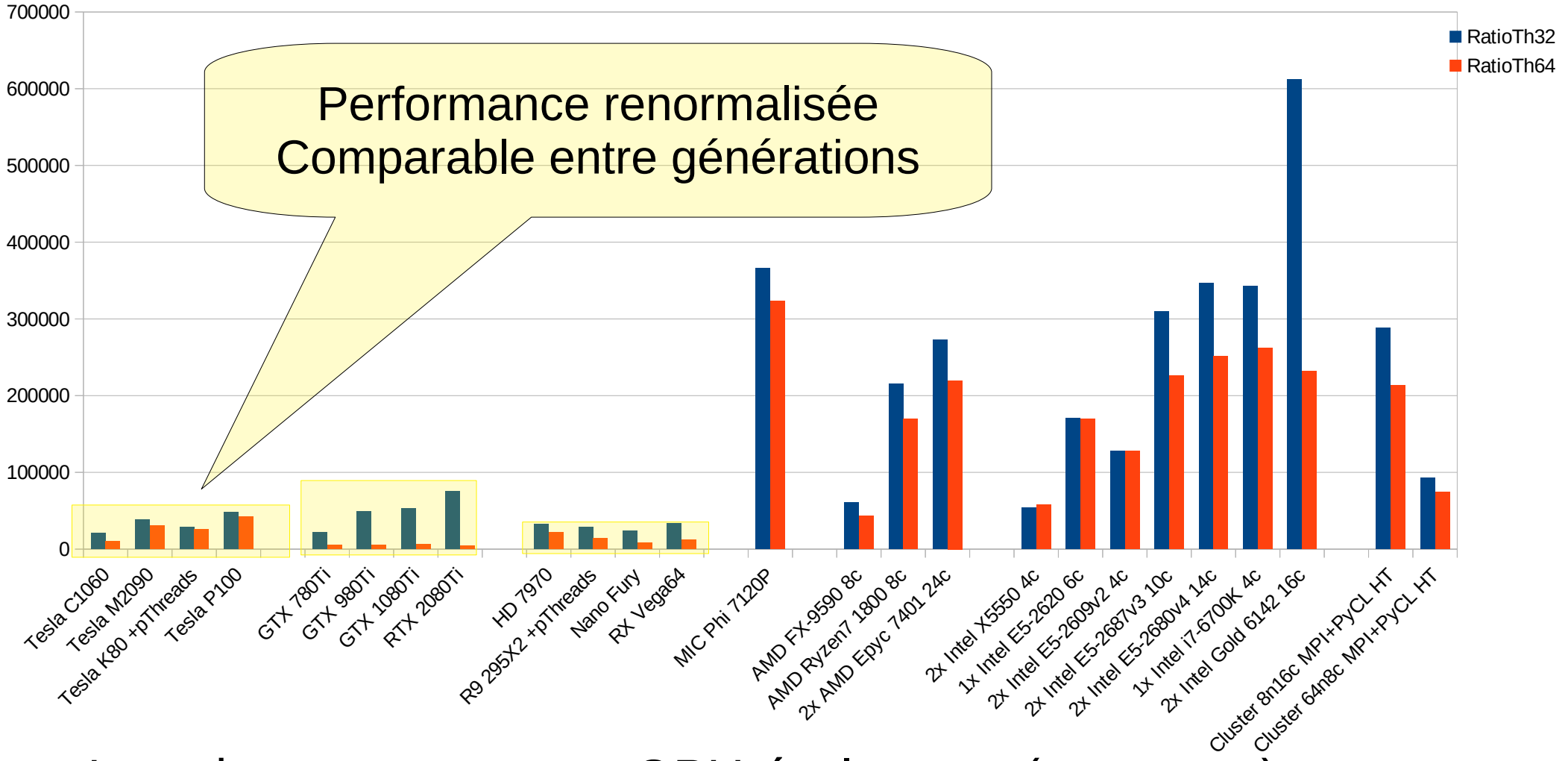
Mais en intégrant les clusters, ça donne quoi ?

Pi Dart Dash les clusters en plus…
A oui quand même...



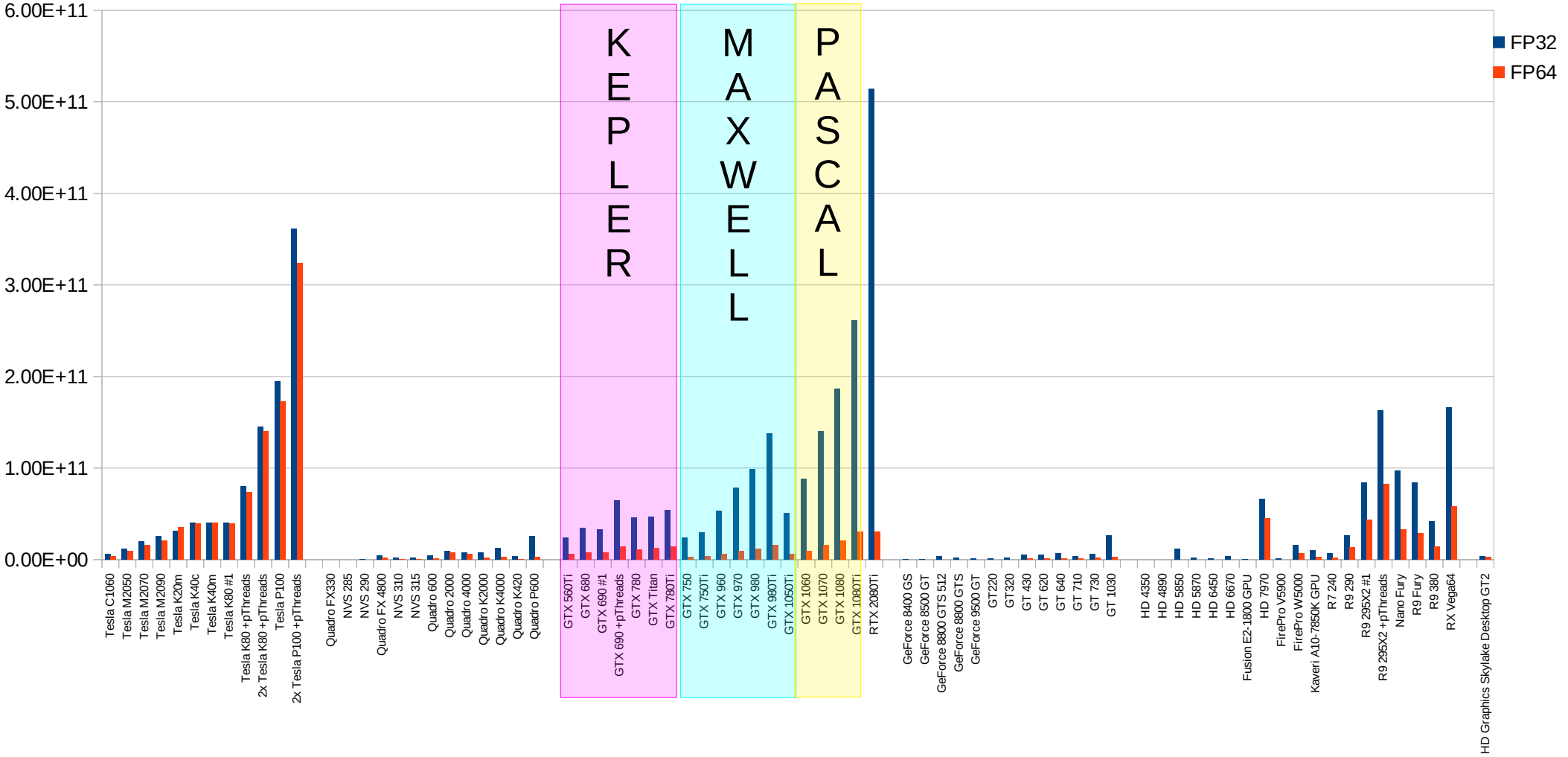
6 GPU de puissance supérieure aux 2 clusters ! (1 en DP)

Le rapport à la puissance « théorique » ? Itops sur coeurs*fréquence



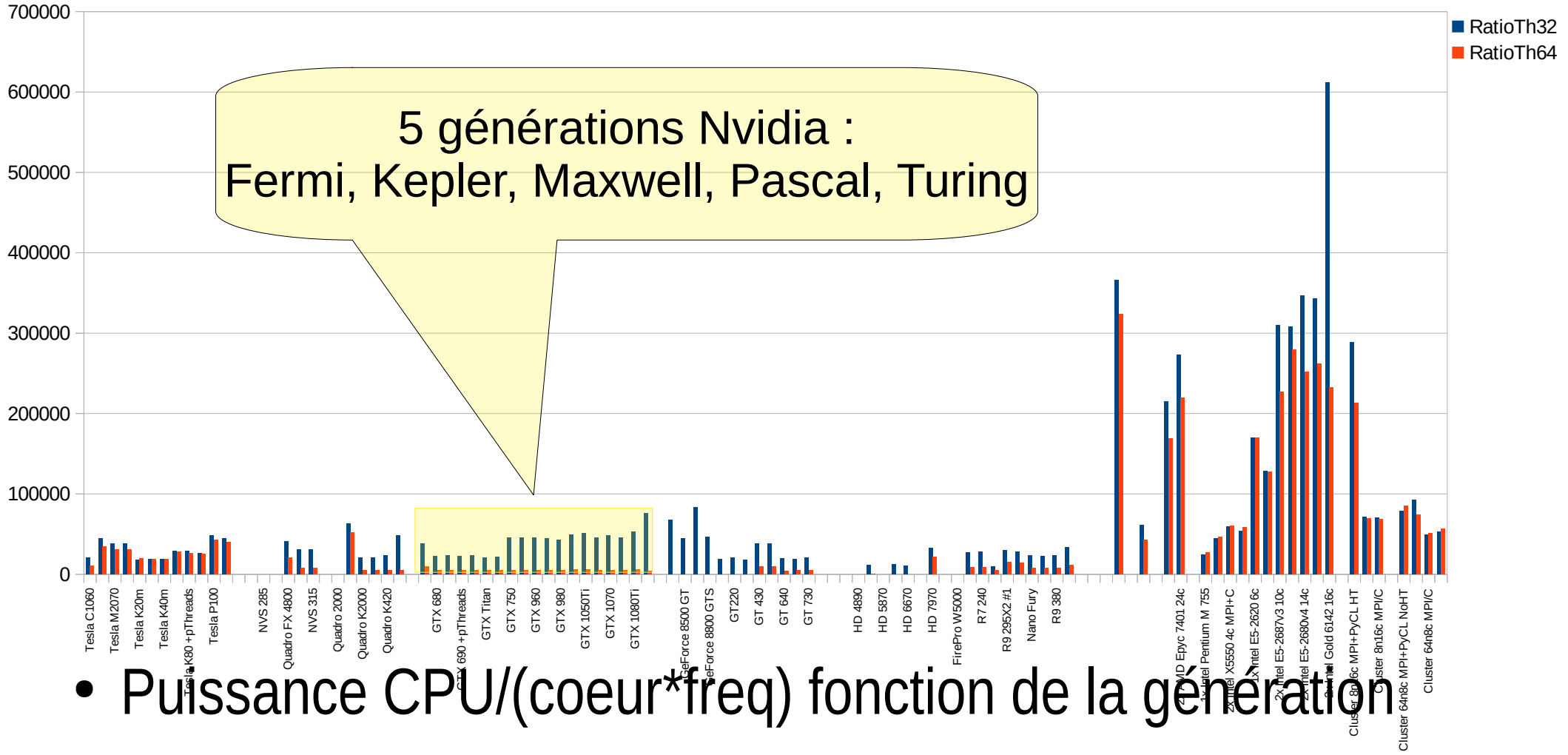
- La puissance par coeur GPU évolue peu (constante)
- La puissance par coeur CPU grandit

Sur le bestiaire du CBP
De génération en génération



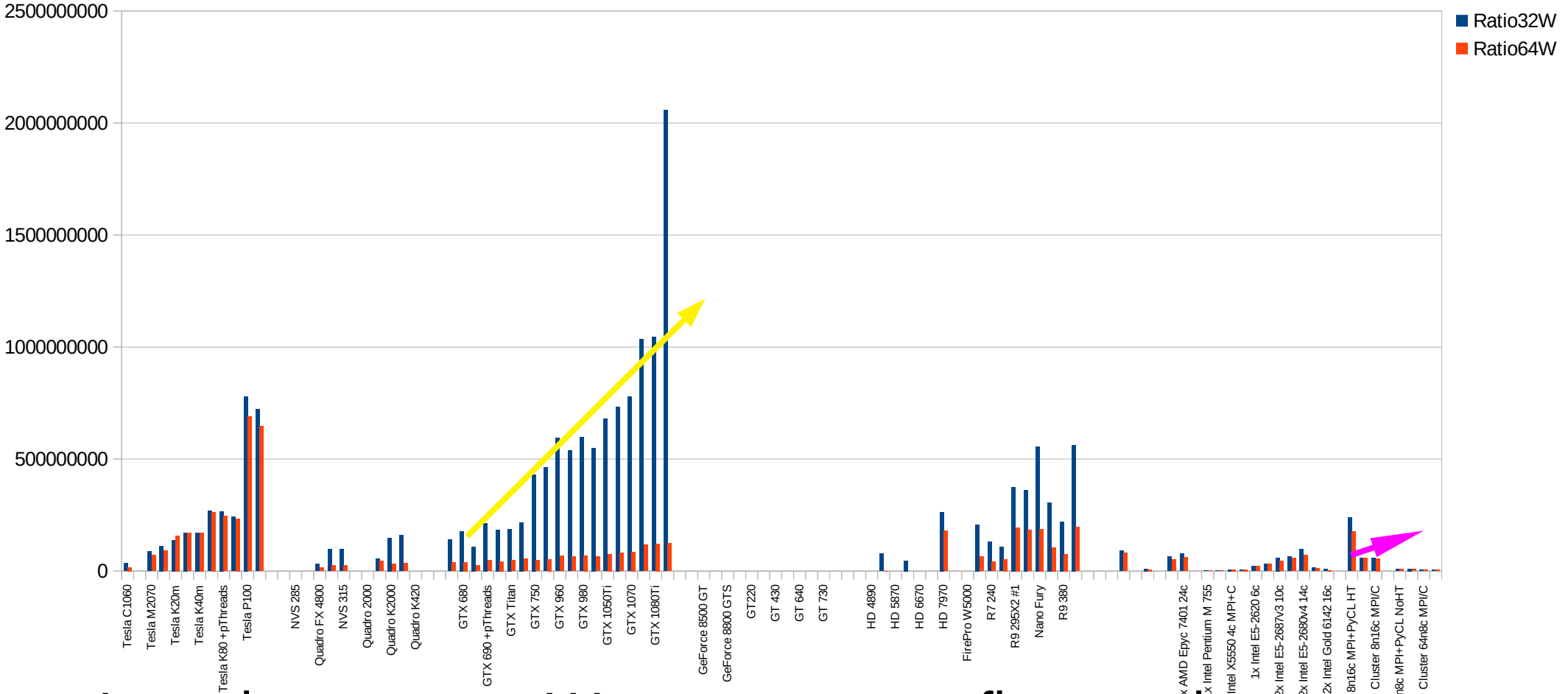
A chaque génération, toute une gamme de performances !

Puissance renormalisée CPU/GPU
Assertion généralisable ?



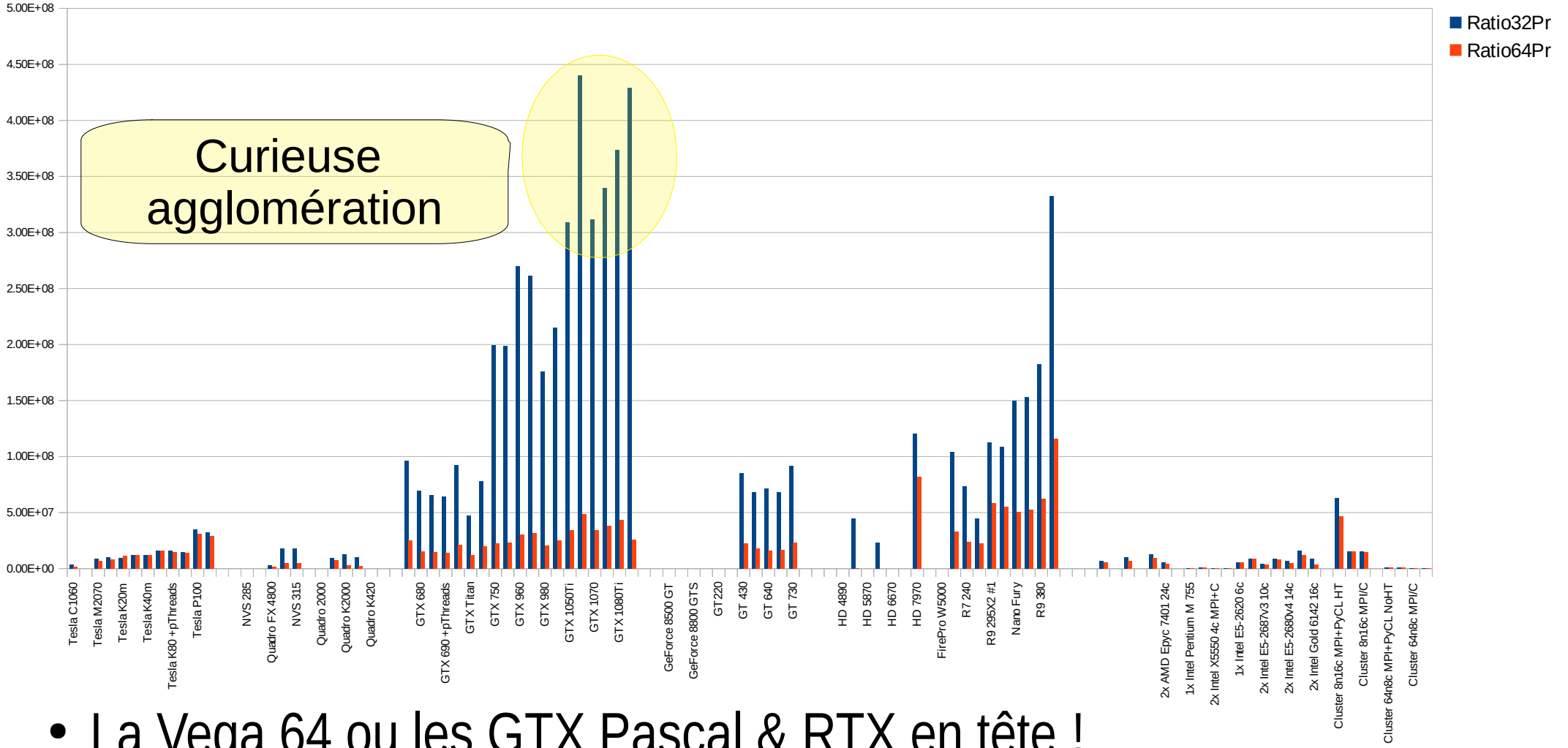
- Puissance CPU/(coeur*freq) fonction de la génération
- Puissance GPU/(coeur*freq) indépendante...

La puissance pour l'écologiste
Renormalisation à la TDP



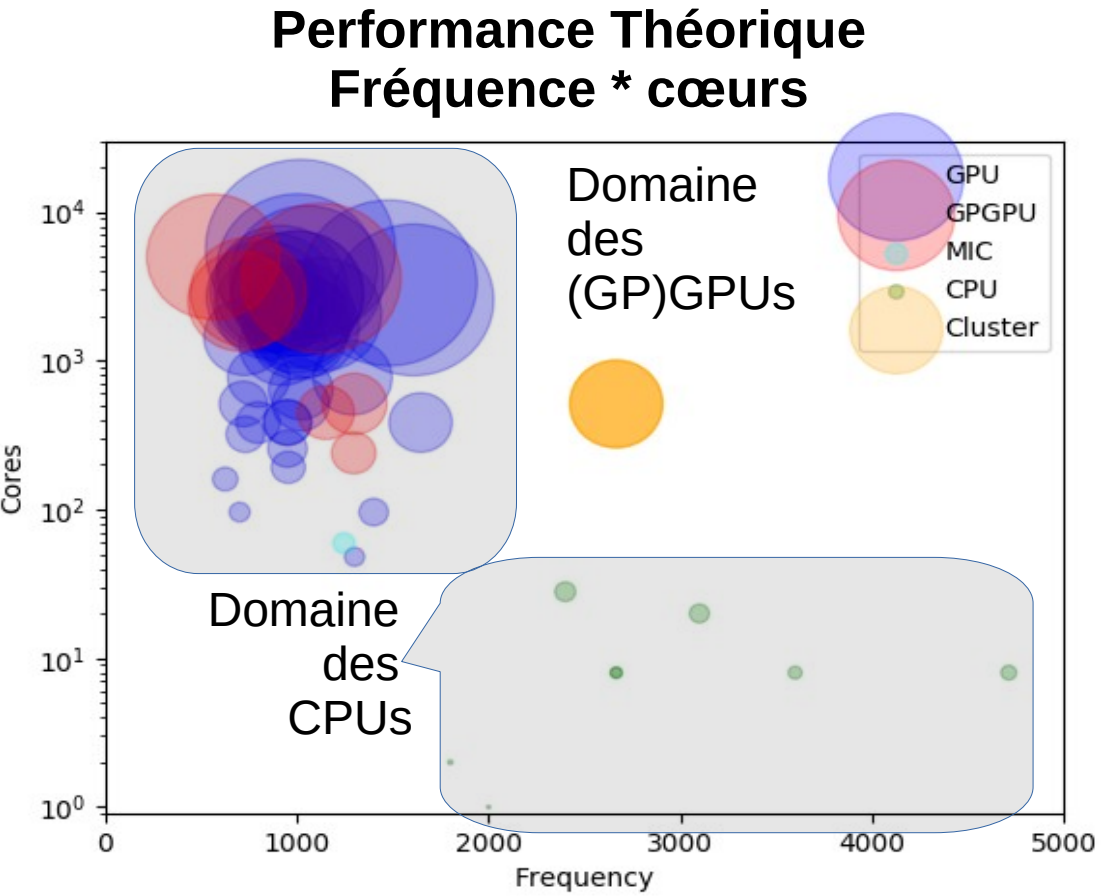
- La puissance par Watt augmente : finesse de gravure
- GPU : de 55 nm à 14 nm ; CPU : de 65 nm à 14 nm

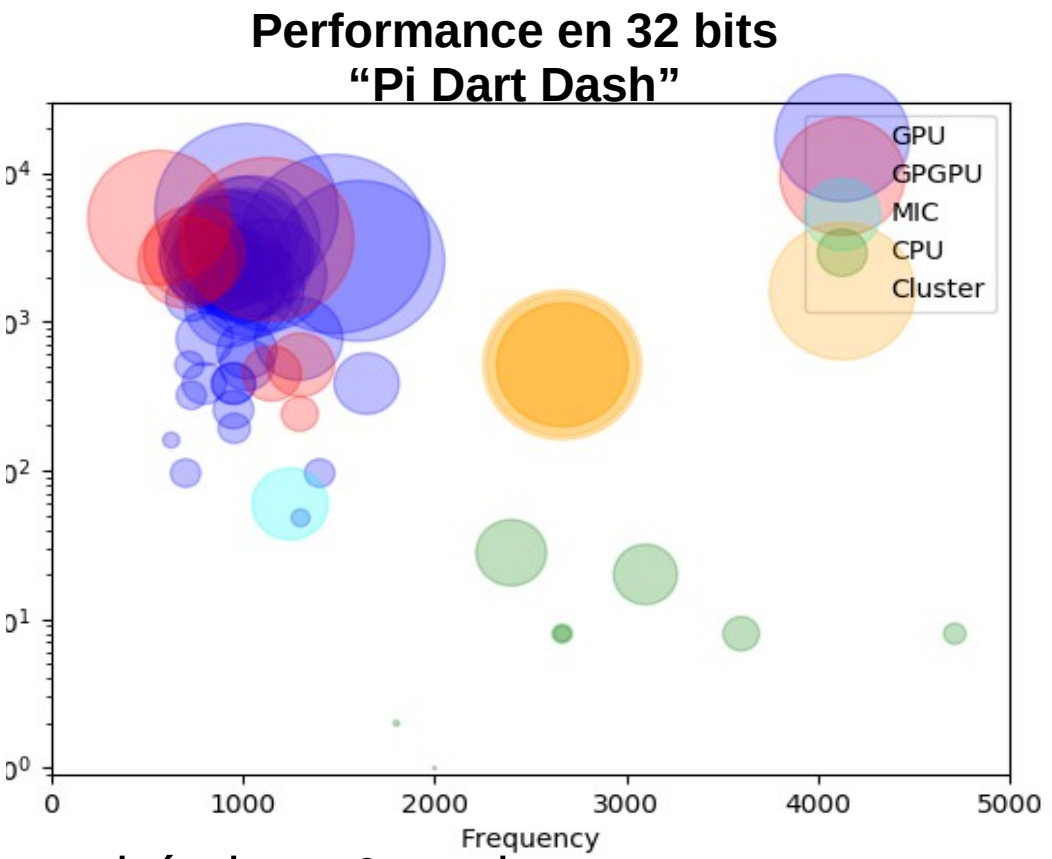
La puissance pour le financier
Renormalisation à la MSRP



- La Vega 64 ou les GTX Pascal & RTX en tête !
- A croire qu'ils utilisent ce « test » pour leurs prix ;-)

Représenter les performances... Question de battement, de cœurs ?

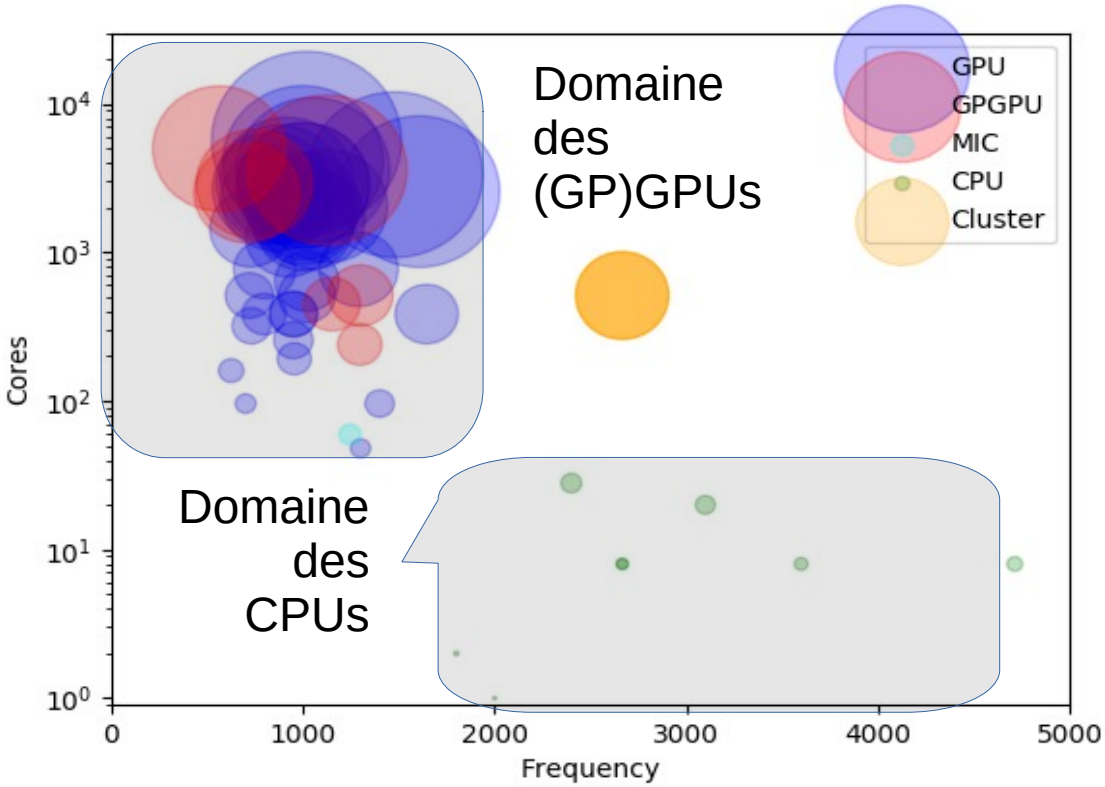


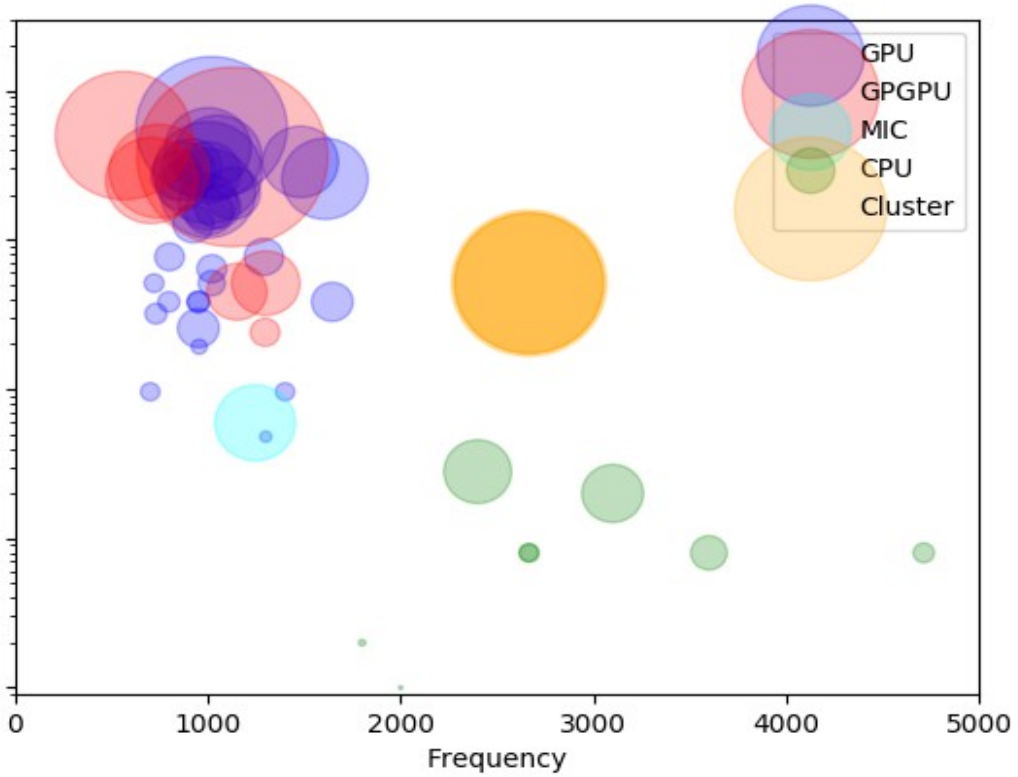


- Sur un GPU, cohérence entre performances théorique & pratique
- Sur un CPU, performance relative meilleure

La performance en informatique

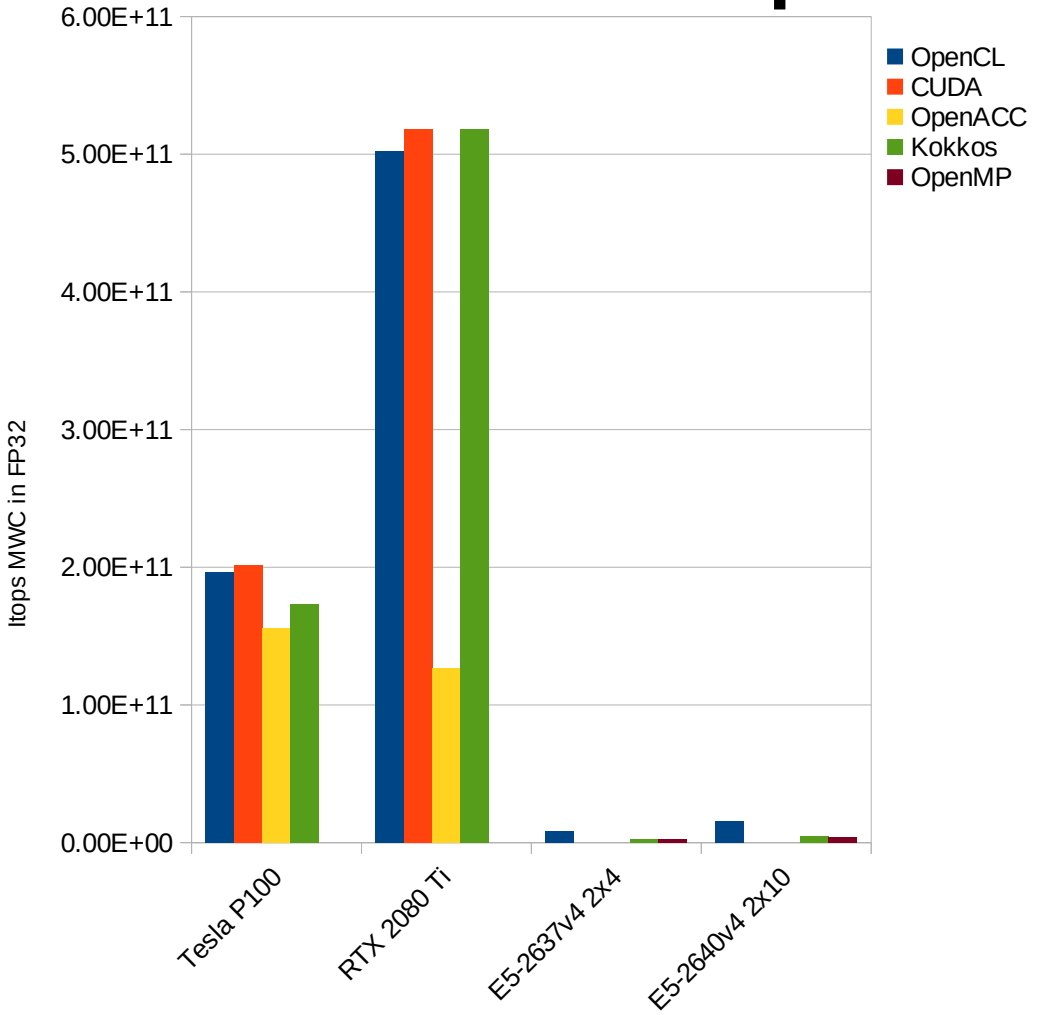
Question de battement, de cœurs, et de bits !

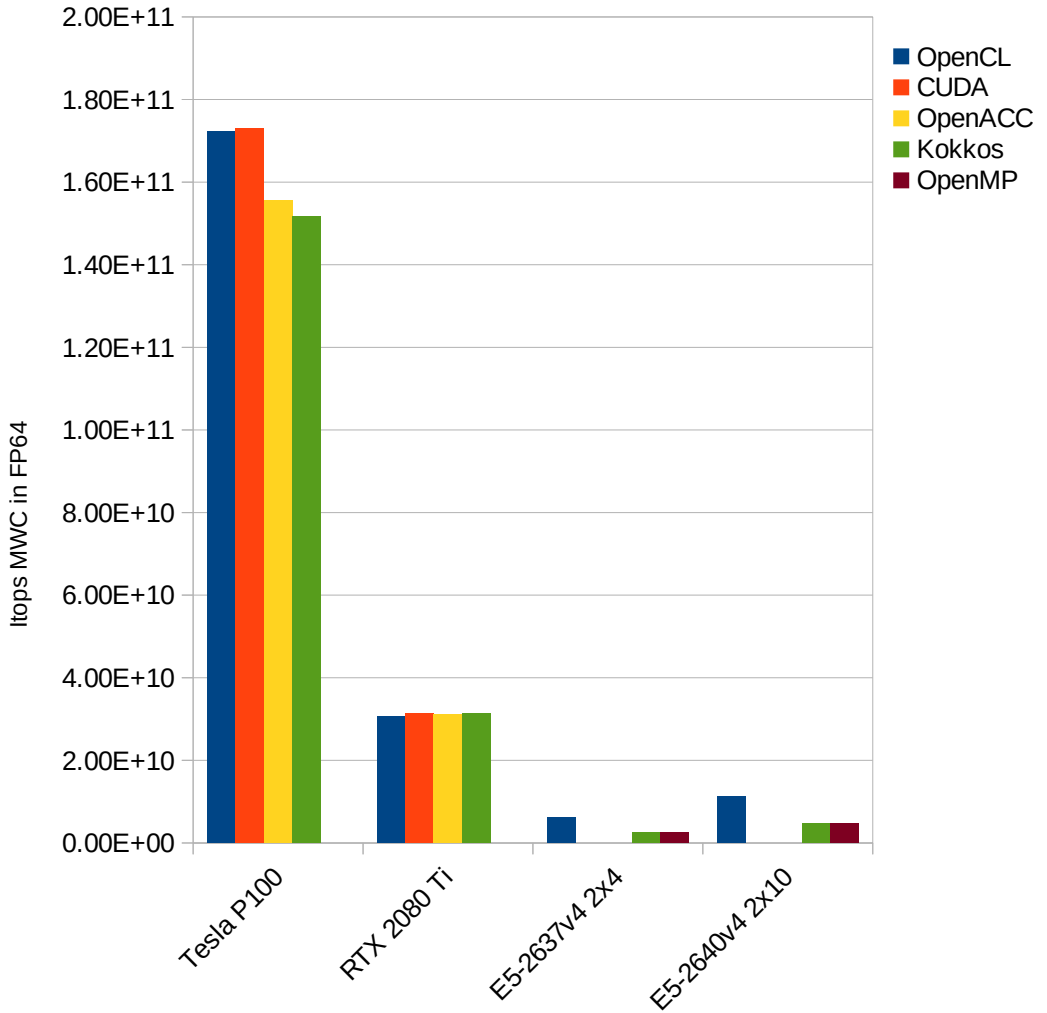




- Sur un GPU, baisse très sensible des performances pour les GPU
- Sur un CPU, performance relative encore meilleure

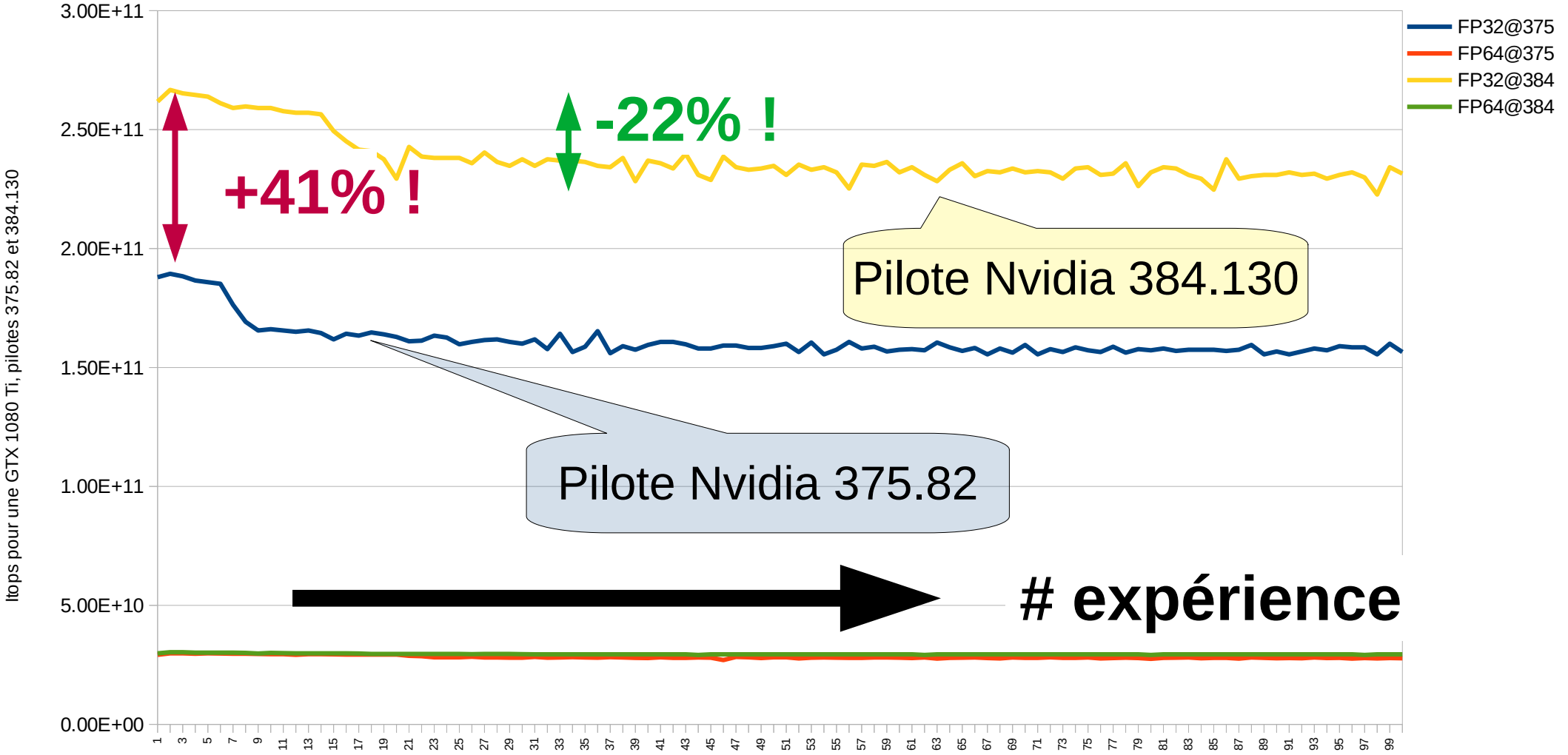
Et pour les autres « approches » CUDA, OpenACC & Kokkos ?





Finalement, ça se vaut… Critère à évaluer : le coût d'entrée..

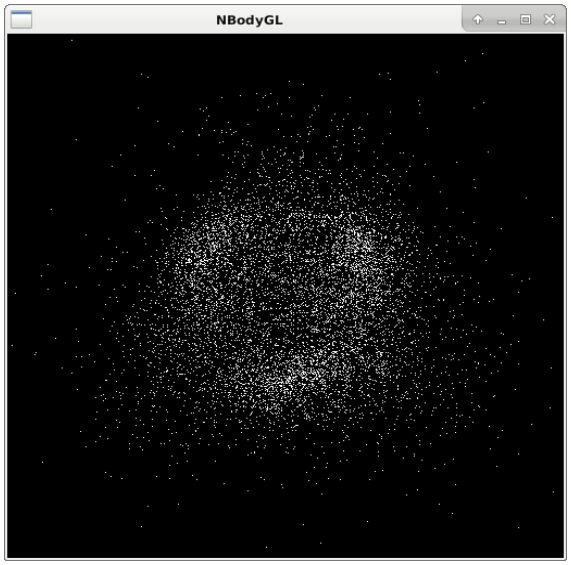
Versions de pilote & expériences
Facteurs de variabilité...

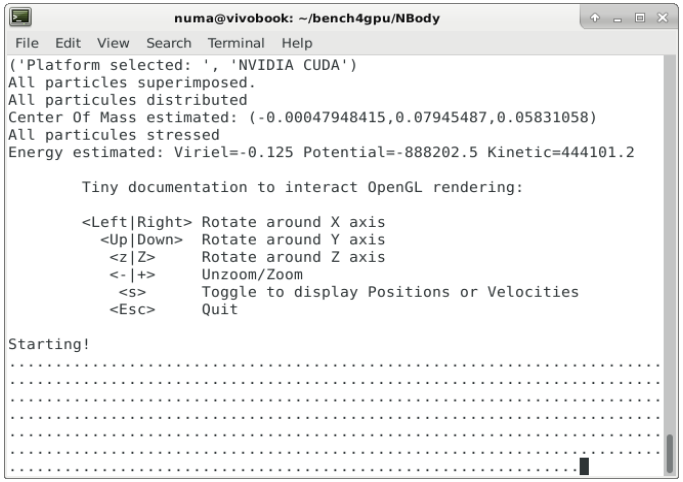


A bien prendre en compte dans l'expérience !

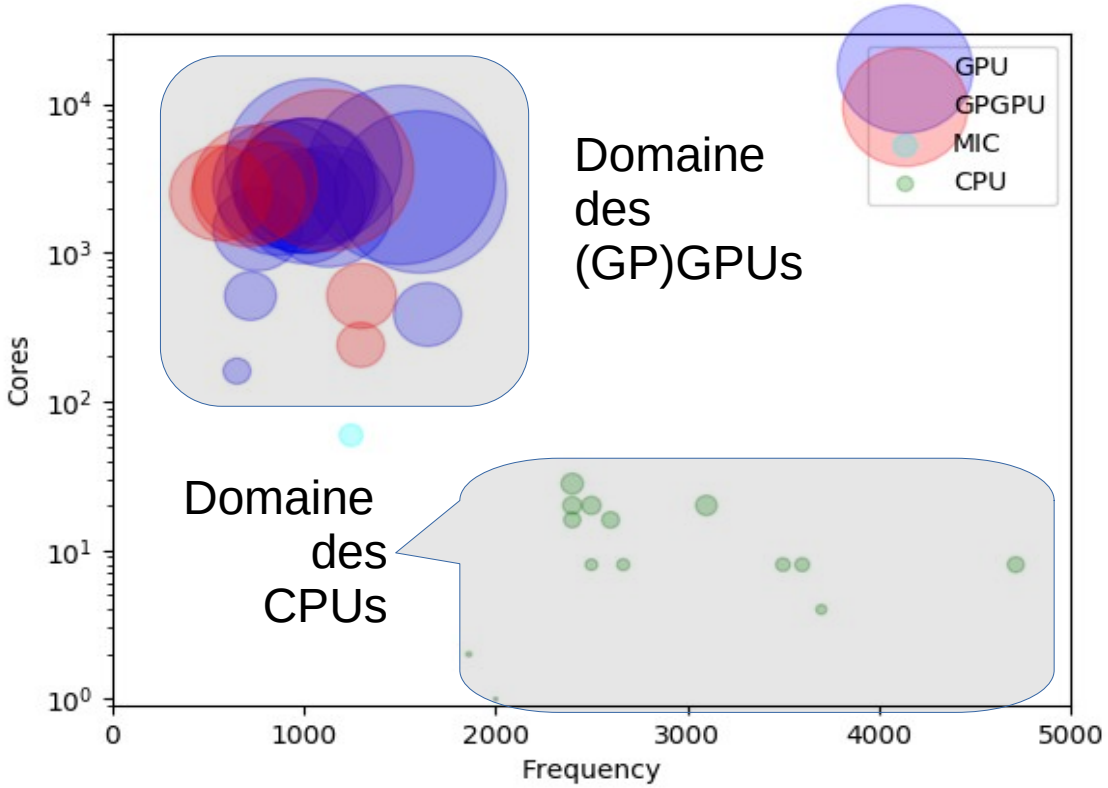
Retour à la physique
Un code newtonien N-corps...

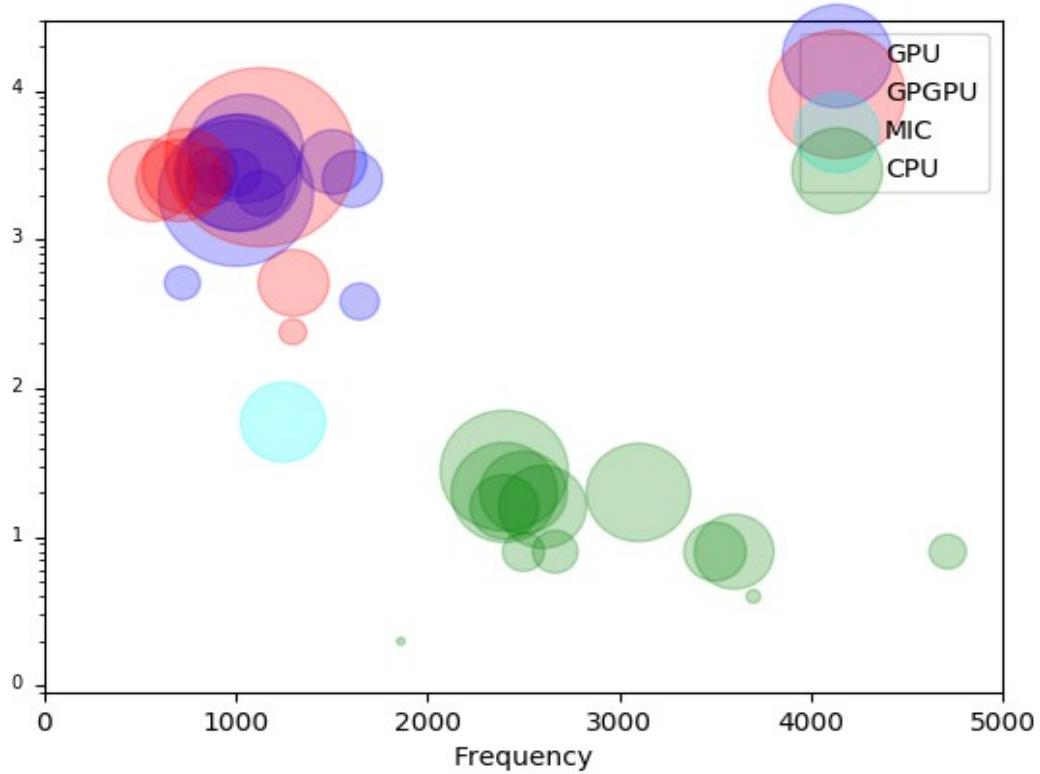
- Passage sur un code « grain fin »
- Code N-body très simplement implémenté
  - Seconde loi de Newton, système autogravitant
- Méthodes d'intégration différentielle :
  - Euler Implicite, Euler Explicite, Heun, Runge Kutta
- Données d'entrées :
  - Physique : nombre de particules
  - Numerique : pas d'intégration, nombre de pas
  - Architecture : influence de la nature du flottant FP32, FP64





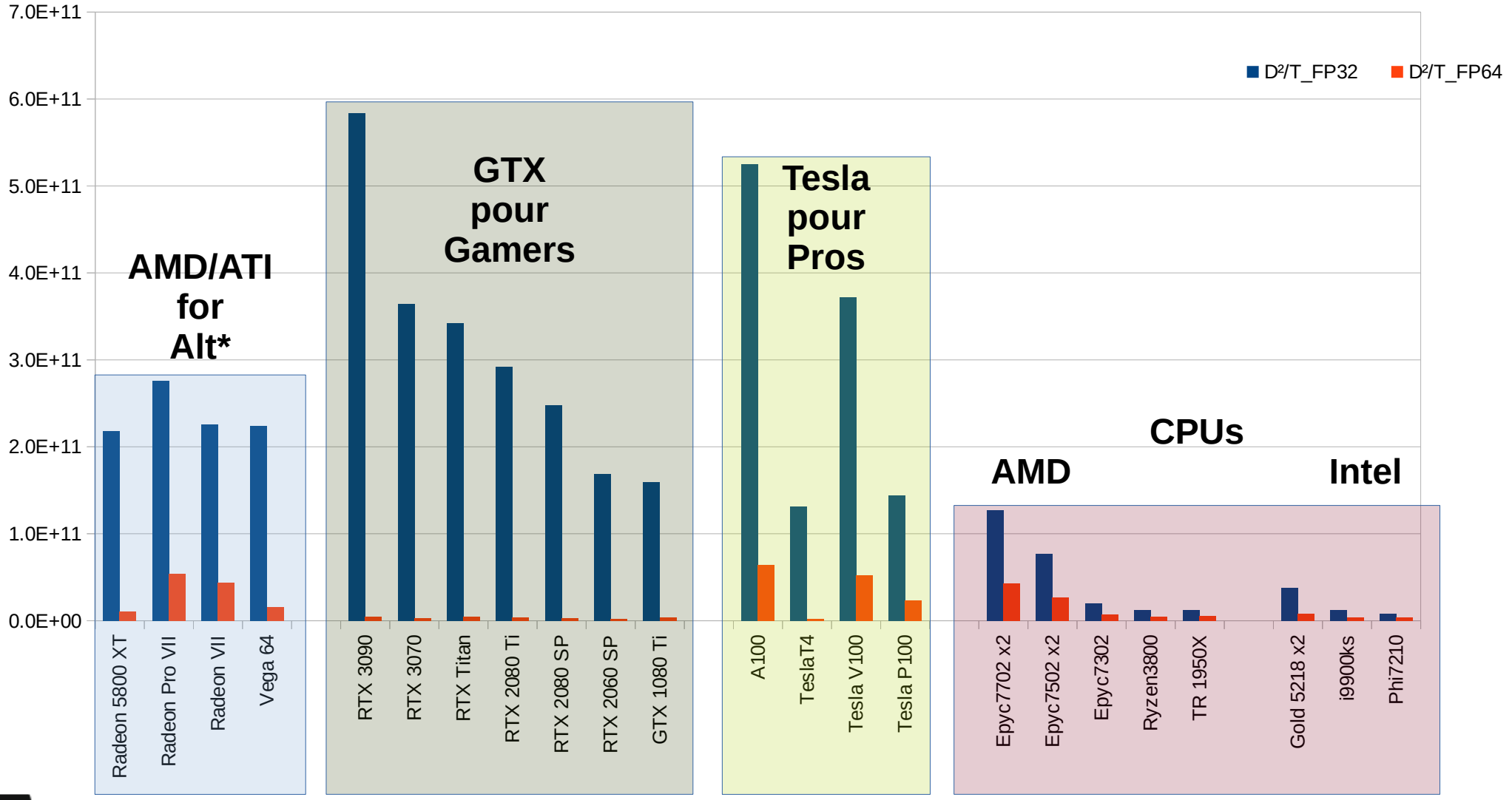
La performance en informatique
Pour un code plus physique...



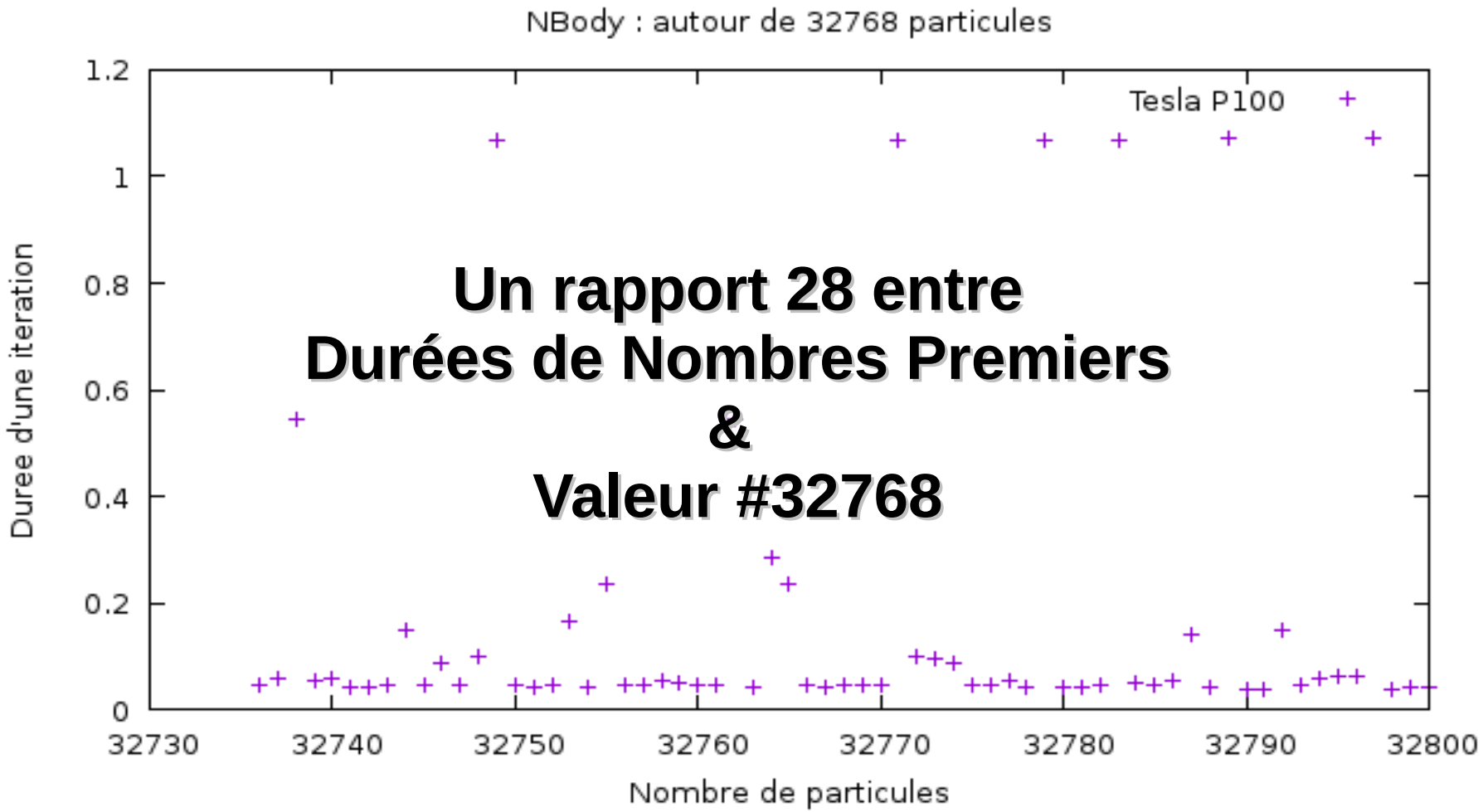


- Sur un GPGPU, performance stable
- Sur un GPU, baisse drastique de la performance (division par 20 en DP)
- Sur un CPU, performance relative meilleure

Modèle N-corps « naïf »
Une comparaison classique...



Et les effets « prime numbers » pour les cartes Nvidia ?



Une petite démonstration de Nbody ?

- Quelques cas d'usage :
  - 8192 particules en FP32
  - 8291 particules en FP32 (8191 est premier!)
  - 8193 particules en FP32
  - 8192 particules en FP64

Références :

- https://computing.llnl.gov/tutorials/dataheroes/GPUParallelProgramming.pdf
- http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf
- https://pdfs.semanticscholar.org/40c0/34f4f11831e837367d06c11b0beb83eefc6b.pdf
- http://download.abandonware.org/magazines/Tilt/tilt_numero016/%5BMag%20-%20Fr%5D%20Tilt%20n%B0016%20-%20Page%20113%20%28Octobre%201984%29.jpg