

INSA Lyon 2020

Master 2

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

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

Emmanuel Quémener

# Conseils avant écoute...

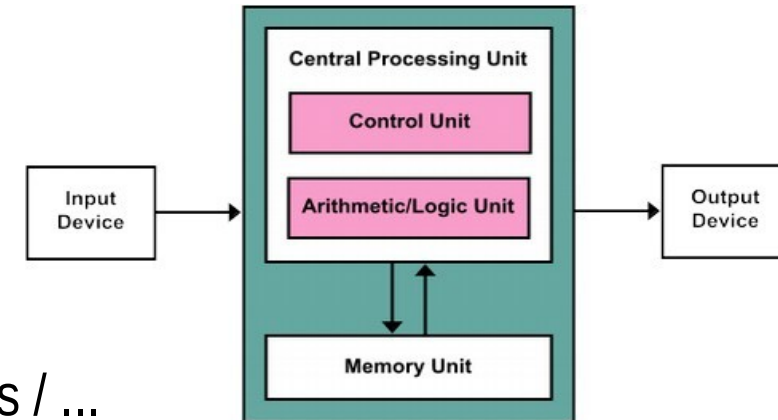
- Je suis le produit de l'université française d'il y a 30 ans...
- Je ne suis pas diplômé en informatique...
  - Mais j'utilise les ordinateurs depuis 37 ans et Linux depuis 25 ans
- Je suis physicien de formation...
  - Mais j'ai travaillé sur les calculateurs analogiques durant ma thèse
- Je suis ingénieur de recherche...
  - Mais j'améliore ma connaissance de tout le spectre IT depuis 25 ans
- Mes expériences les plus significatives d'ingénieur ?
  - KISS : pour *Keep It Simple Stupid*
  - « Si vous ne pouvez pas prouver que vous avez fait le travail, ce n'est pas la peine de l'entreprendre ! »

# 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...

# Quel objectif pour ce cours ?

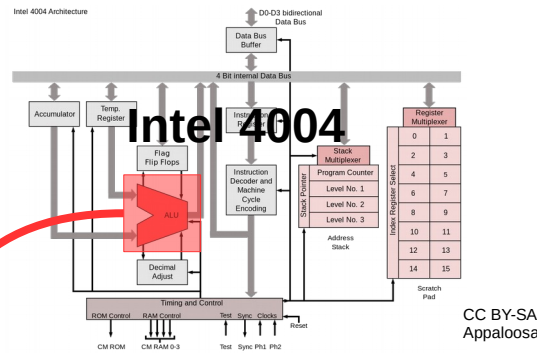
- Offrir une vision « décalée » sur les GPUs
- Présenter un tour d'horizon de leur programmation
- Attester que le GPU en calcul est incontournable en 2020
- Dégager des « comportements » à l'usage
- Montrer que le « modèle en couches » est inadapté
- Avertir que l'informatique est une science expérimentale
- Préparer la session de TP dans 1 semaine ou dans 1 mois

# Quelques définitions & sigles...

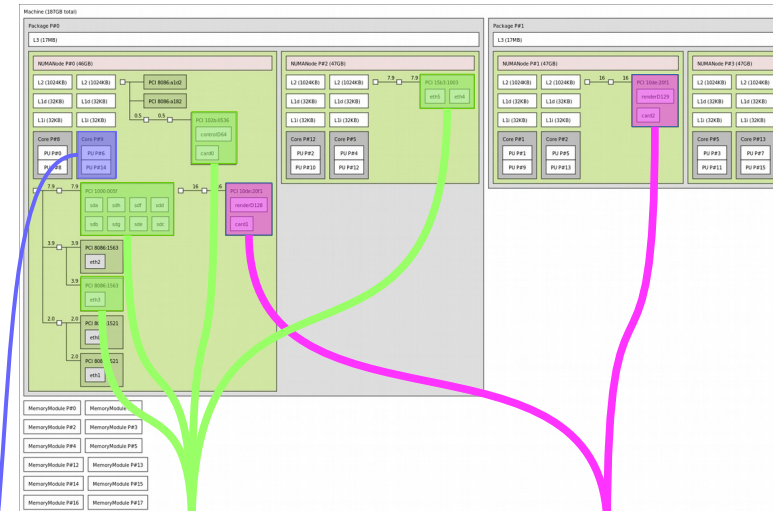
- **ALU** : *Arithmetic & Logic Unit*, Unité Arithmétique et Logique
- **CPU** : *Central Processing Unit*, Unité de traitement Centrale,
- **Flops** : *Floating Point Operations Per Second*, Opérations flottantes par seconde
- **(GP)GPU** : *(General Purpose) Graphical Processing Unit*, Circuit graphique
- **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)

# Où est le GPU ?

## Modèle de Von Neumann...

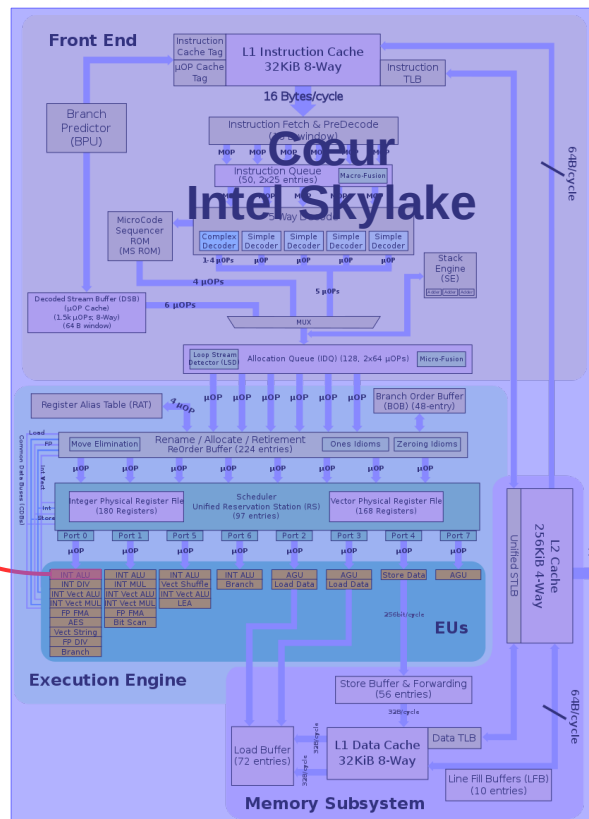
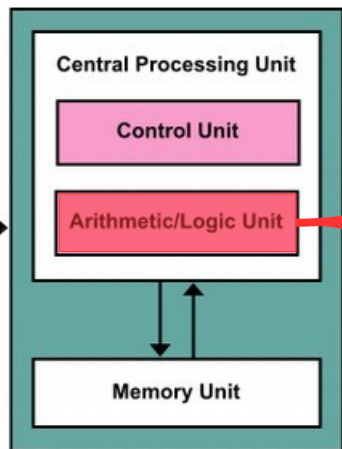
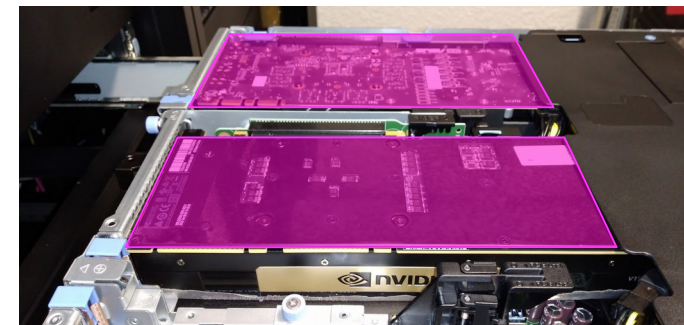


CC BY-SA Appaloosa



Input / Output

GPU Nvidia V100



# Qu'est-ce le Centre Blaise Pascal ?

Directeur : Pr Ralf Everaers

- Centre Blaise Pascal: « maison de la modélisation »
  - Hôtel à projets, conférences, formations sur toute l'informatique scientifique
- Hôtel à projets :
  - Plateaux techniques de plates-formes expérimentales pour tout le monde
  - Paillasse numériques pour les laboratoires sur besoins spécifiques
- Hôtel à formations :
  - ATOSIM (Erasmus Mundus)
  - Formation continue pour chercheurs, professeurs et ingénieurs
  - Formation avancée : M1, M2 en physique, chimie, géologie, biologie, ...
  - Ateliers 3IP : « Introduction Inductive à l'Informatique et au Parallélisme »
- **Centre d'essai : recycler, détourner, explorer en HPC & HPDA**

# 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  $>50^\circ$
  - « Fly-By-Wire »

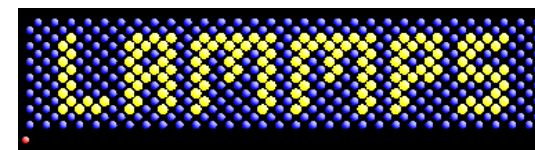
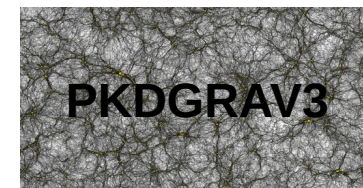
Recycler, réutiliser, explorer de nouveaux domaines...



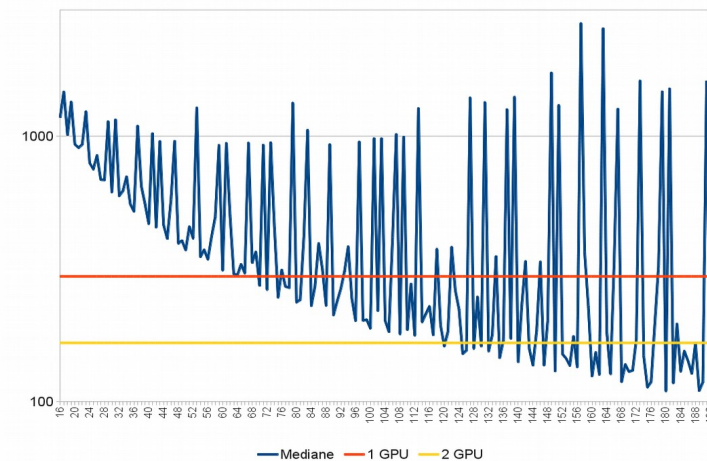
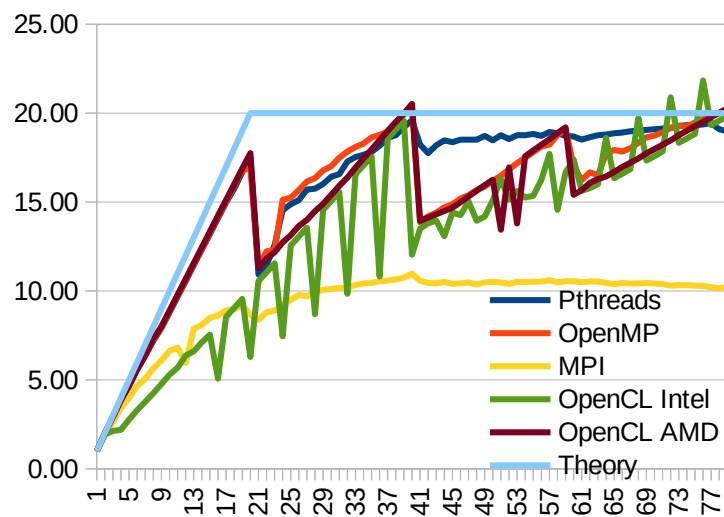
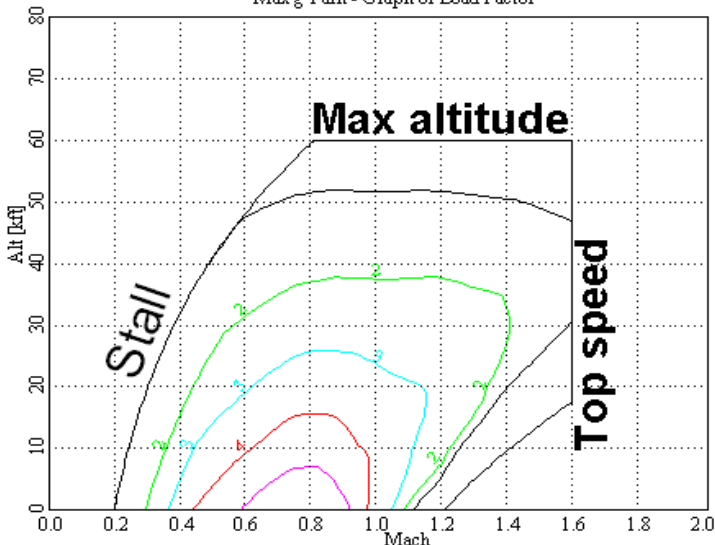
# Le but : de l'enveloppe de vol... ... aux enveloppes de parallélisme



NASA Dryden Flight Research Center EC87 0182-14 Photographed 1987 X-29



Max g Turn - Graph of Load Factor



# Cloud@CBP : le point d'entrée

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

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

**Cloud@CBP : État des ressources**

Bonjour, utilisateur d'adresse IP 140.77.78.6.  
Vous semblez surfer avec le navigateur Chrome sous GNU/Linux

Le 2020-11-29, Heure Locale 03:22 **A cet instant,** 94 machines "chargées" à 3.99 et utilisées par 145 utilisateurs  
le Cloud@CBP, c'est : CPU : 165 sockets avec 1528 cœurs dans 39 modèles différents  
GPU : 106 cartes dans 46 modèles différents.

Liens rapides : [Configuration x2go](#) [Demande d'accès ou d'assistance](#)

**Sélection d'une machine**

- Machine générique
- Machine multi-cœurs (>=32)
- Machine à grosse RAM (>=256GB)
- Machine avec gros GPU de Gamer
- Machine avec GPGPU (Tesla)

Submit Reset

**Liste des machines avec caractéristiques techniques**

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	6	62	2500	GeForce GT 1030	GeForce GTX 680	0.04	0
gtx690	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz	6	62	2500	GeForce GTX 690	GeForce GTX 690	0.03	7

# Plateau multi-nœuds: 8 clusters

## 162 nœuds, 3 vitesses IB



4 nodes Sun X4150 with Xeon E5440  
32 physical cores @2833MHz  
Interconnection Infiniband DDR 20 Gb/s

64 nodes Dell R410 with Xeon X5550  
512 physical cores HT @2666MHz  
Interconnection Infiniband QDR 40 Gb/s

64 nodes Supermicro R422 with Xeon X5650  
512 physical cores HT @2666MHz  
Interconnection Infiniband QDR 40 Gb/s

16 nodes Dell C6100 with Xeon X5650  
192 physical cores HT @2666MHz  
Interconnection Infiniband QDR 40 Gb/s

8 nodes HP SL230 with Xeon E5-2667  
64 physical cores HT @2666MHz  
Interconnection Infiniband FDR 56 Gb/s

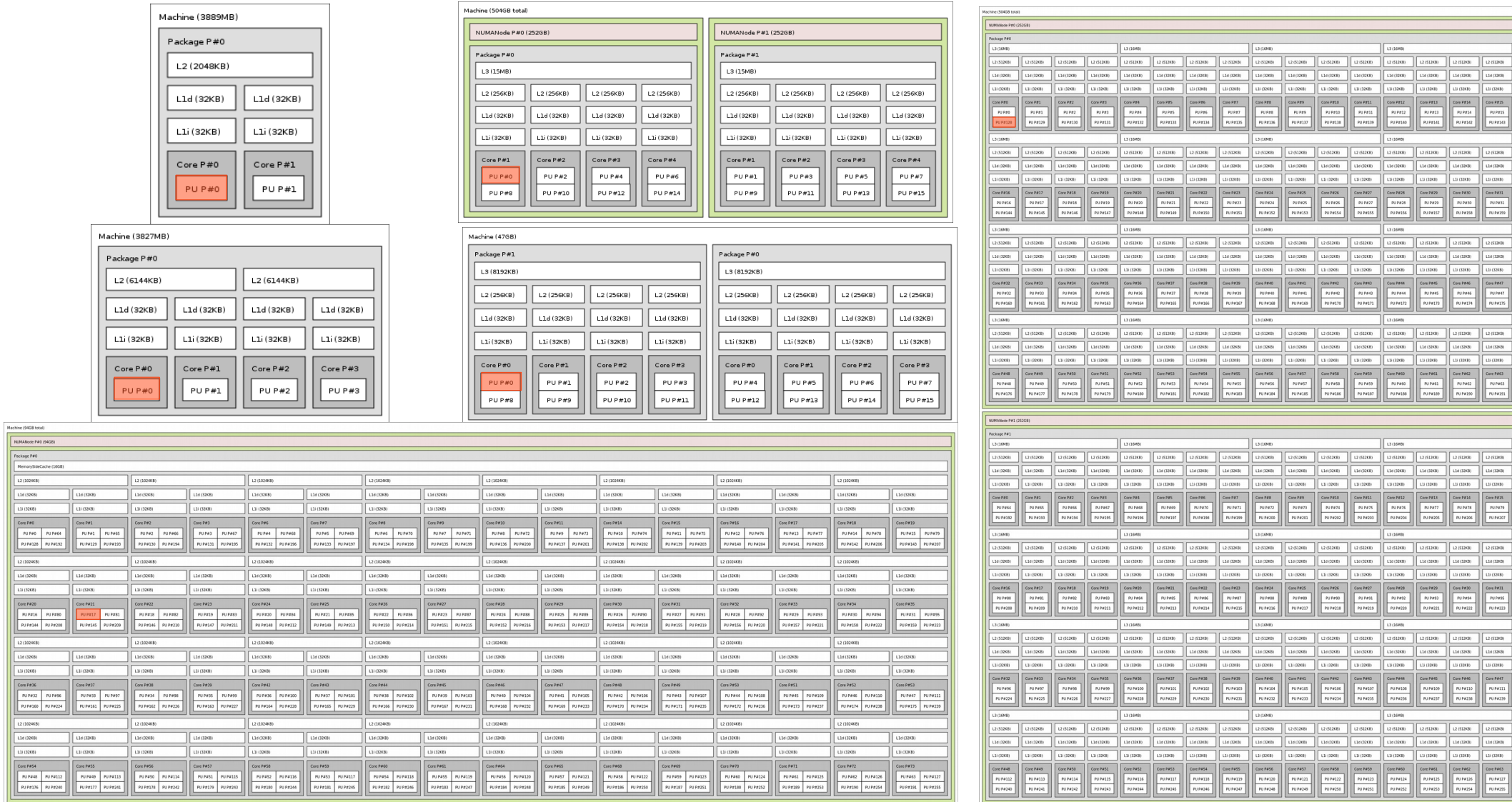
4 nodes Dell R410 with Xeon X5550  
32 physical cores HT @2666MHz  
Interconnection Infiniband DDR 20 Gb/s

2 nodes Sun X4500 with AMD 285  
4 physical cores @2400MHz  
Interconnection Infiniband SDR 10 Gb/s



# Plateau multi-cœurs

## De 2 à 128 cœurs : exemples...



# Plateau technique (GP)GPU

## 90 modèles différents...

### GPU Gamer : 28

- Nvidia GTX 560 Ti
- Nvidia GTX 680
- Nvidia GTX 690
- Nvidia GTX Titan
- Nvidia GTX 780
- Nvidia GTX 780 Ti
- Nvidia GTX 750
- Nvidia GTX 750 Ti
- Nvidia GTX 960
- Nvidia GTX 970
- Nvidia GTX 980
- Nvidia GTX 980 Ti
- Nvidia GTX 1050 Ti
- Nvidia GTX 1060
- Nvidia GTX 1070
- Nvidia GTX 1080
- Nvidia GTX 1080 Ti
- Nvidia RTX 2070
- Nvidia RTX 2080
- Nvidia RTX 2080 Ti
- Nvidia RTX 1650
- Nvidia GTX 1660 Ti
- Nvidia RTX 2060 Super
- Nvidia RTX 2070 Super
- Nvidia RTX 2080 Super
- Nvidia RTX Titan
- **Nvidia RTX 3070**
- **Nvidia RTX 3090**

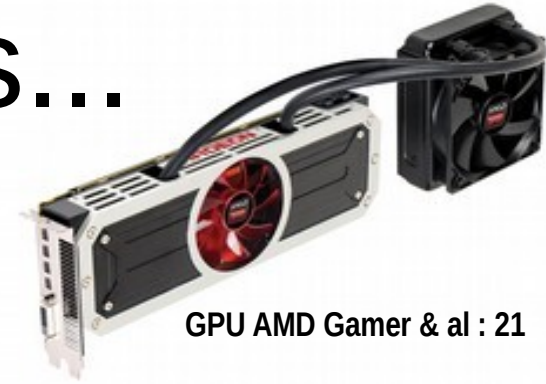


### GPGPU : 12

- Nvidia Tesla C1060
- Nvidia Tesla M2050
- Nvidia Tesla M2070
- Nvidia Tesla M2090
- Nvidia Tesla K20m
- Nvidia Tesla K40c
- Nvidia Tesla K40m
- Nvidia Tesla K80
- Nvidia Tesla P100
- Nvidia Tesla V100
- **Nvidia Tesla T4**
- **Nvidia A100**

### GPU desktop & pro : 29

- NVS 290
- Nvidia FX 4800
- NVS 310
- NVS 315
- Nvidia Quadro 600
- Nvidia Quadro 2000
- Nvidia Quadro 4000
- Nvidia Quadro K2000
- Nvidia Quadro K4000
- Nvidia Quadro K420
- Nvidia Quadro P600
- Nvidia 8400 GS
- Nvidia 8500 GT
- Nvidia 8800 GT
- Nvidia 9500 GT
- Nvidia GT 220
- Nvidia GT 320
- Nvidia GT 430
- Nvidia GT 545
- Nvidia GT 620
- Nvidia GT 640
- Nvidia GT 710
- Nvidia GT 730
- Nvidia GT 1030
- Nvidia Quadro 2000M
- Nvidia Quadro K4000M
- Nvidia Quadro M1200
- Nvidia Quadro M2200
- Nvidia MX150



### GPU AMD Gamer & al : 21

- HD 4350
- HD 4890
- HD 5850
- HD 5870
- HD 6450
- HD 6670
- Fusion E2-1800 GPU
- HD 7970
- FirePro V5900
- FirePro W5000
- Kaveri A10-7850K GPU
- R7 240
- R9 290
- R9 295X2
- Nano Fury
- R9 Fury
- R9 380
- RX Vega64
- Radeon VII
- **Radeon Pro VII**
- **Radeon 5800X**

# Plateau 3IP (prononcez “Trip”)

## “Introduction Inductive à l’Informatique et au Parallélisme”

### Computhèque

#### Atelier

- Diagnostics
- Désassemblage
- Tests unitaires
- (Re)Qualification
- Récupération supports

#### Refuge

- Machines “ouvertes”
- Machines “exotiques”
- Composants obsolètes

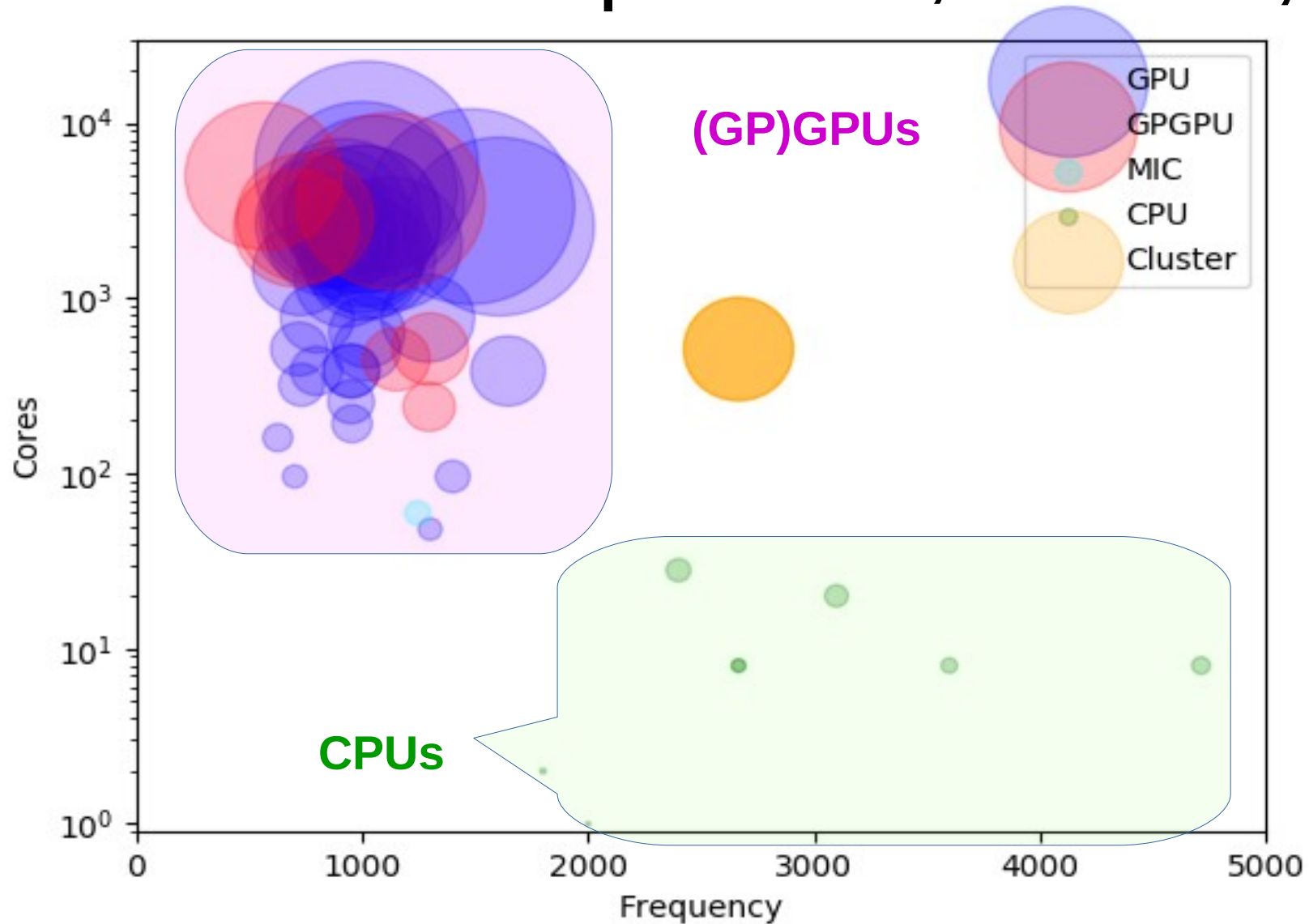
#### Salle de formation

- Ateliers 3IP
- Fête de la science

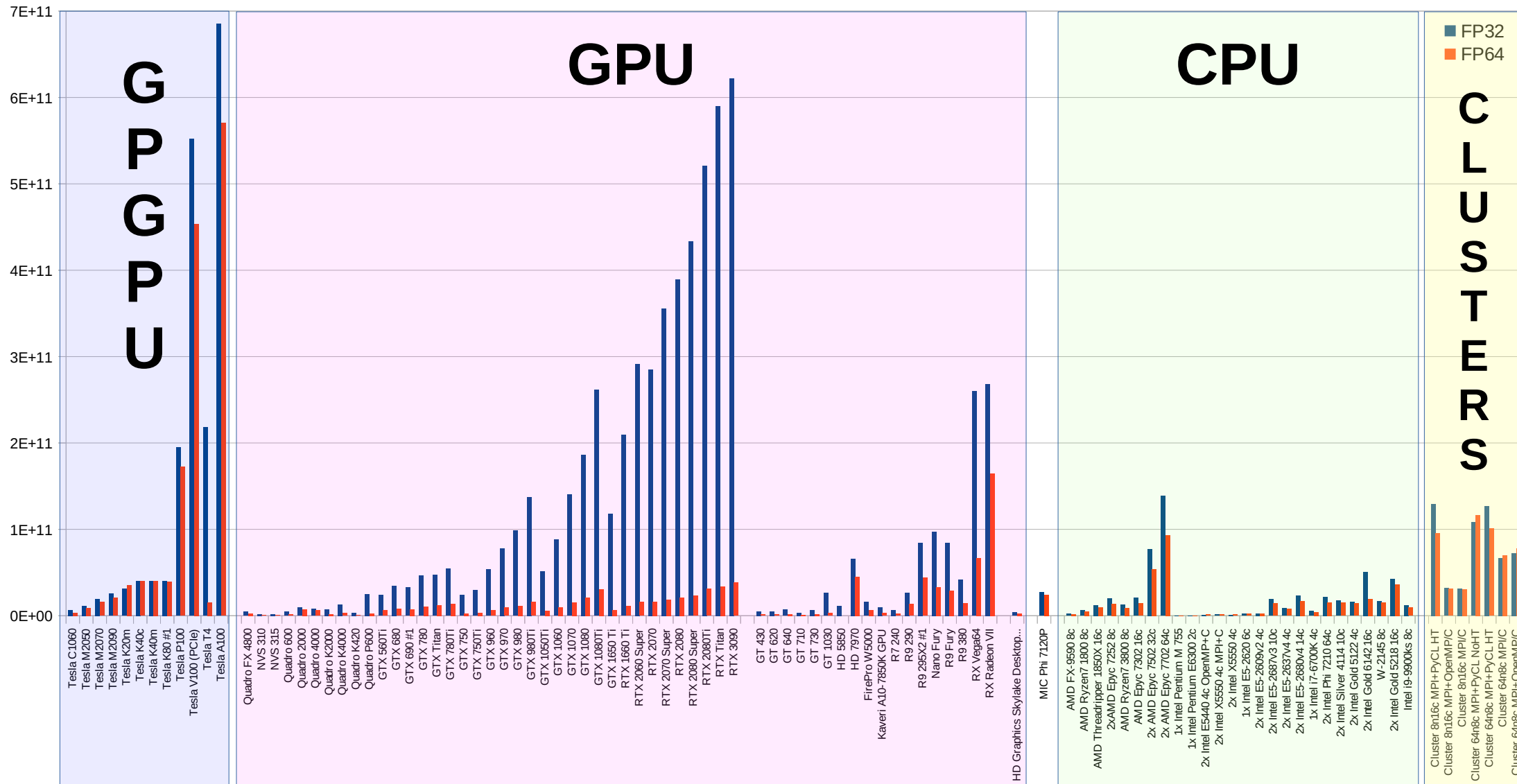


# Comment les représenter ?

## Question de fréquences, cœurs, ...



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





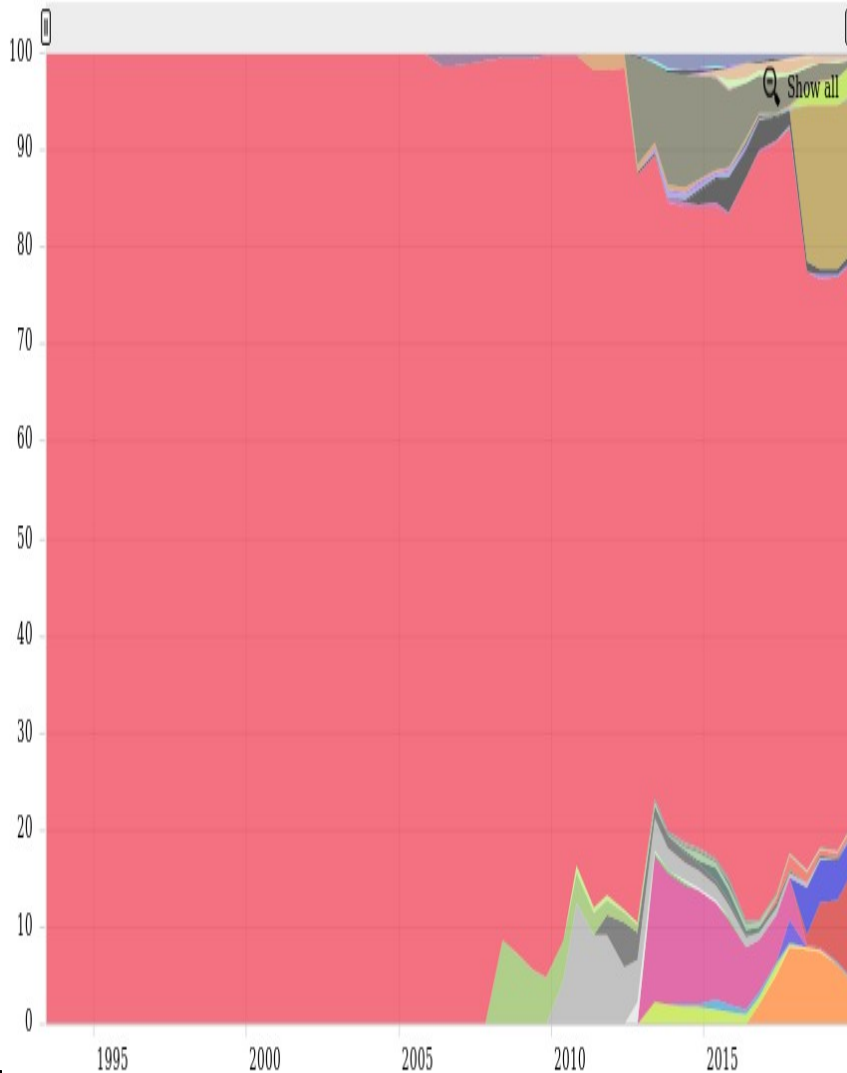
# Mais avant tout chose...

## 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 ?

# Où en sommes-nous ?

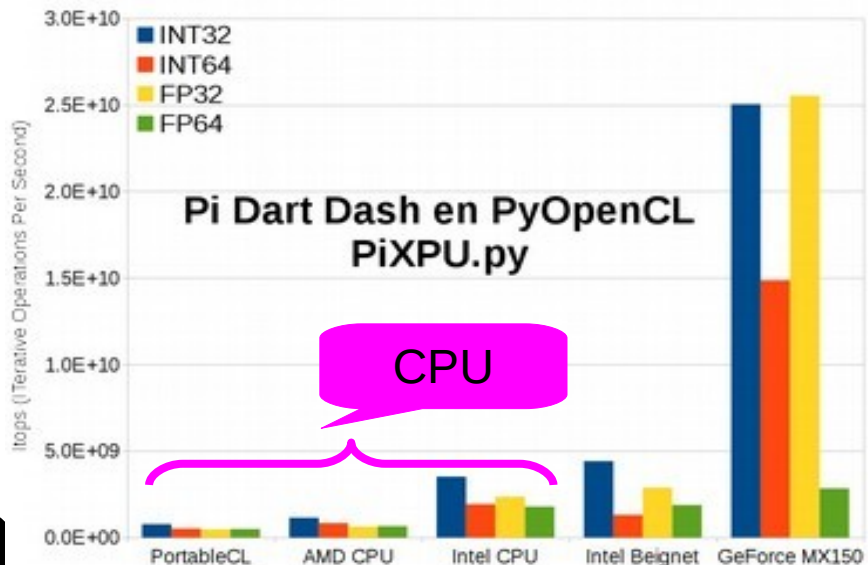
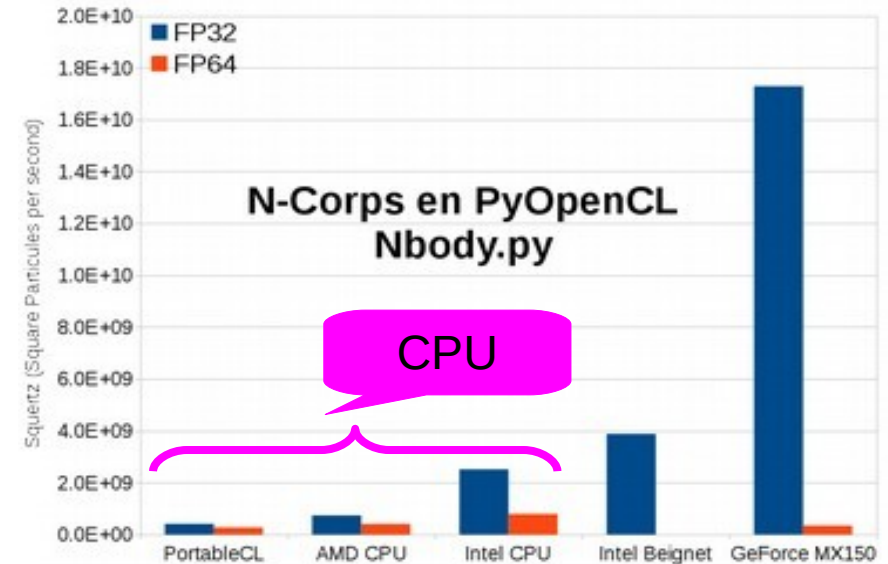
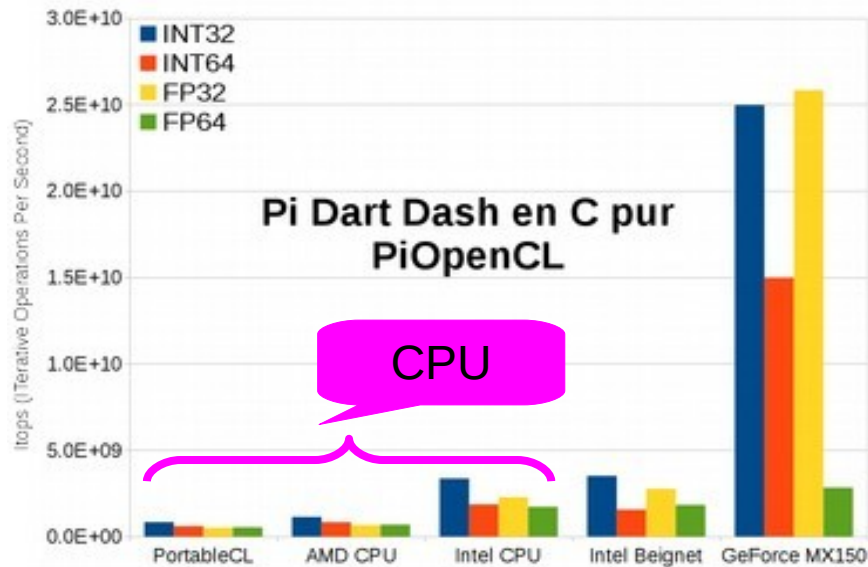
## TOP 500 & les accélérateurs



- Pourquoi : comparer les « machines »
- Quoi : le « linpack »
- Où : sur le Top 500
- Quand : novembre 2020
- Combien :
  - ~1/4 avec accélérateurs :
    - MIC (Intel & copies)
    - GPGPU (Nvidia, AMD)
  - 7/10 dans le Top 10

# Où en sommes-nous ? Qu'espérer ?

## Sur un laptop, benches (C|G)PU

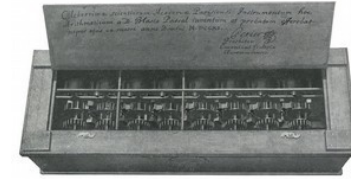


- Un « Pi Dart Dash »
  - En C/OpenCL
  - En Python OpenCL
- Un « N-Corps » naïf

# 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
  - $\sim 30 \cdot 10^9$  opérations par image
  - Utilisation d'un Cray X-MP (130 kW)
  - 68 jours (en fait, 1 année nécessaire)



- 2020 : RTX 3090 (350 W)

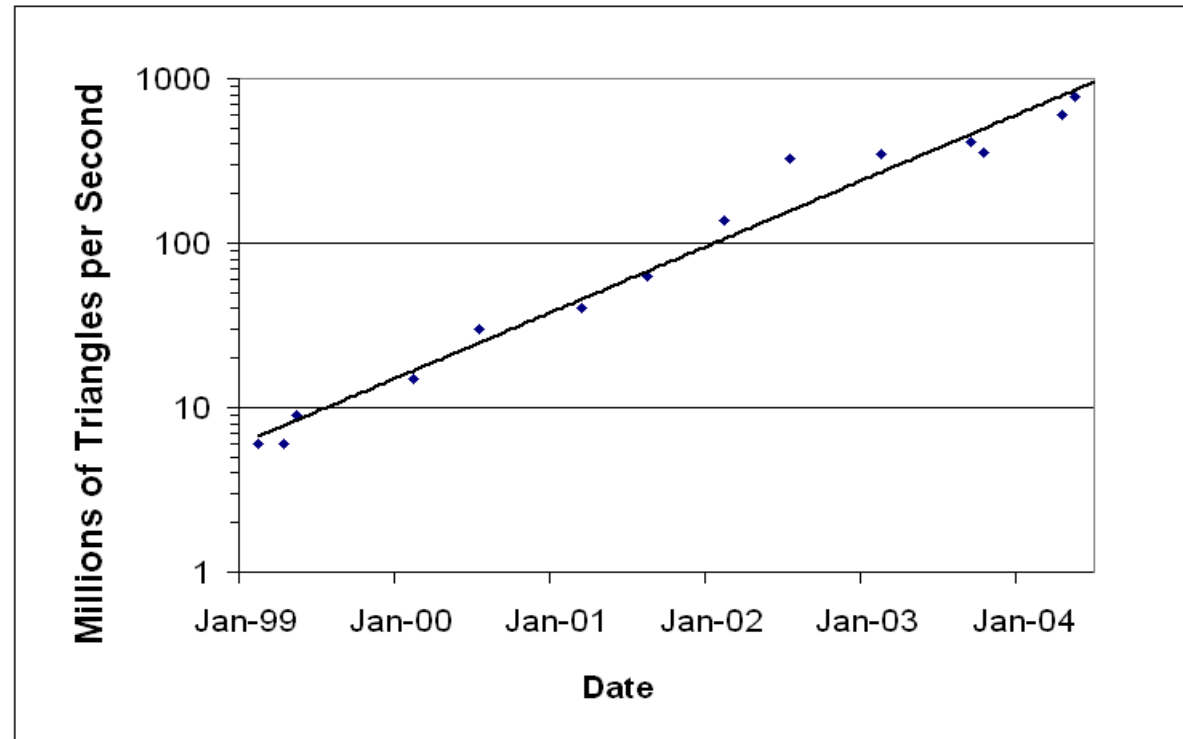
- 33 secondes
- Comparison RTX 3090 / Cray
  - Performance : 178 000 !
  - Consommation  $\sim 66\,000\,000$  !



# 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

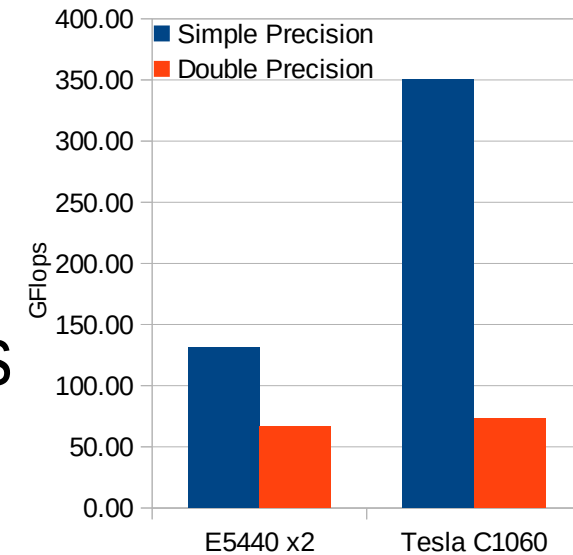
# Position du GPU face aux autres?

## 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

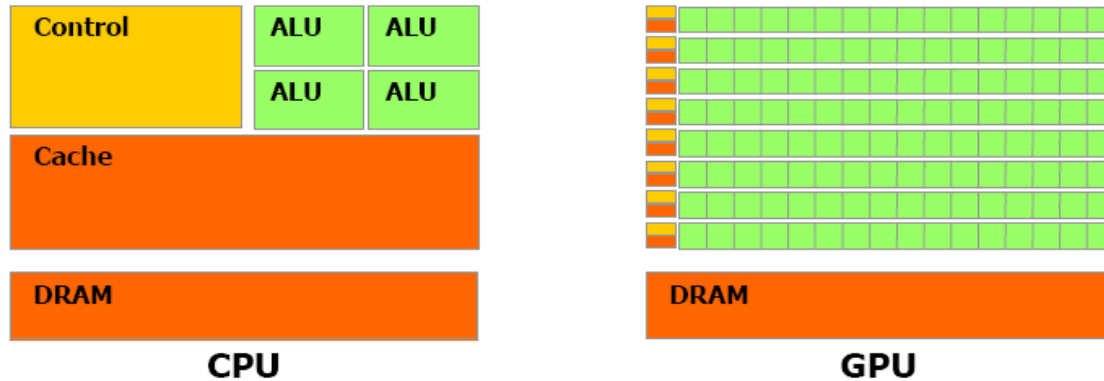
# « Let's go back to the source ! »

- « Nvidia Launches Tesla Personal Supercomputer »
- Quand : le 19/11/2008
- Où : sur Tom's Hardware
- Qui : Nvidia
- 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

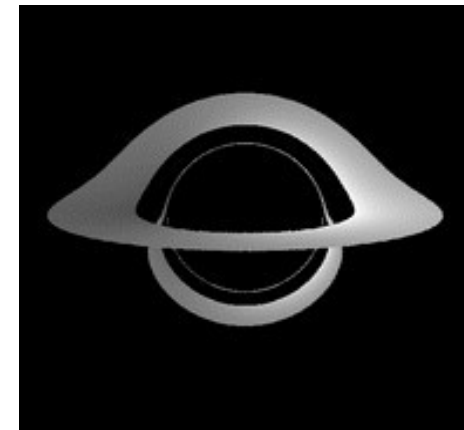
**Nvidia A100**  
**108 unités SM**

**6912 Cuda Cores**  
**432 Tensor Cores**

# Pourquoi le GPU est-il si puissant ?

## Pour construire une image 3D !

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

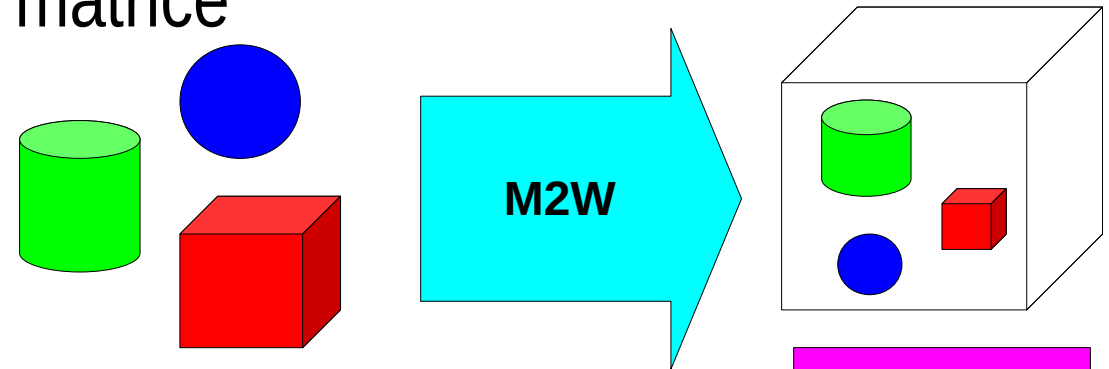


# Pourquoi le GPU est-il si puissant ?

## Shadering & Calcul matriciel

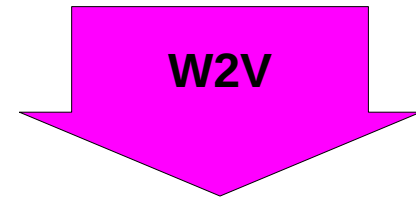
- **Model 2 World** : 3 produits de matrice

- Rotation
- Translation
- Mise à l'échelle



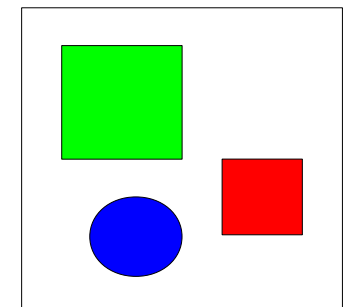
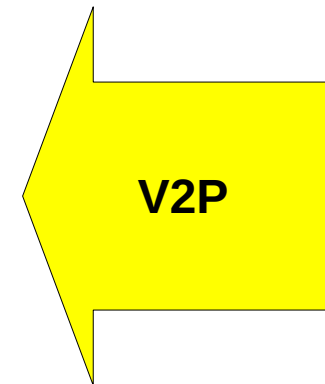
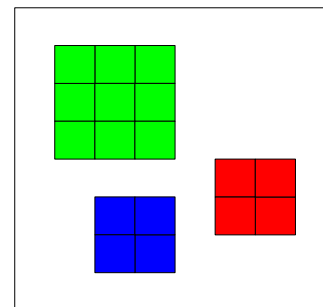
- **World 2 View** : 2 produits de matrice

- 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 » ?

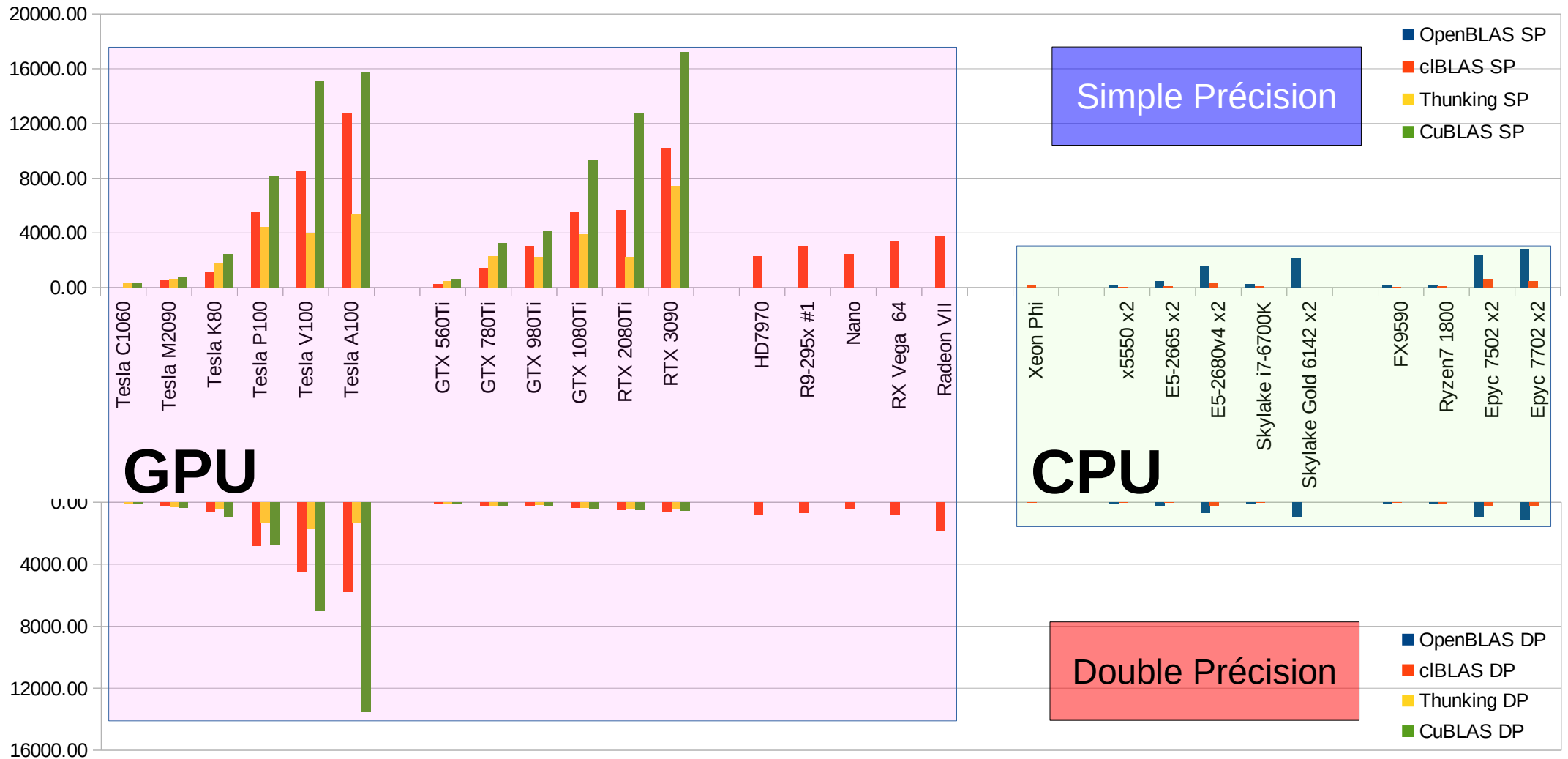
- Soit 2 matrices A et B de dimensions  $N \times P$  et  $P \times M$
- Le produit matriciel de A par B donne C
- Chaque élément de C,  $C_{ij}$  pour i de 1 à N et j de 1 à M :

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

- 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

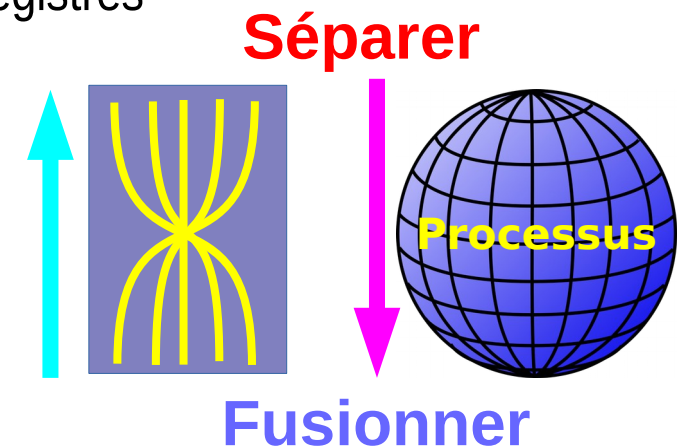
- 2 spécifications des RISC : 1 instruction/cycle, exploitation des registres

- Deux approches

- Vectorisation : Fusion/Processus/Séparation
- Distribution : Séparation/Processus/Fusion

- En fait, paralléliser, c'est plutôt « médianiser »

Instr. No.	Pipeline Stage						
	IF	ID	EX	MEM	WB		
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



# Mais pas seulement un MM...

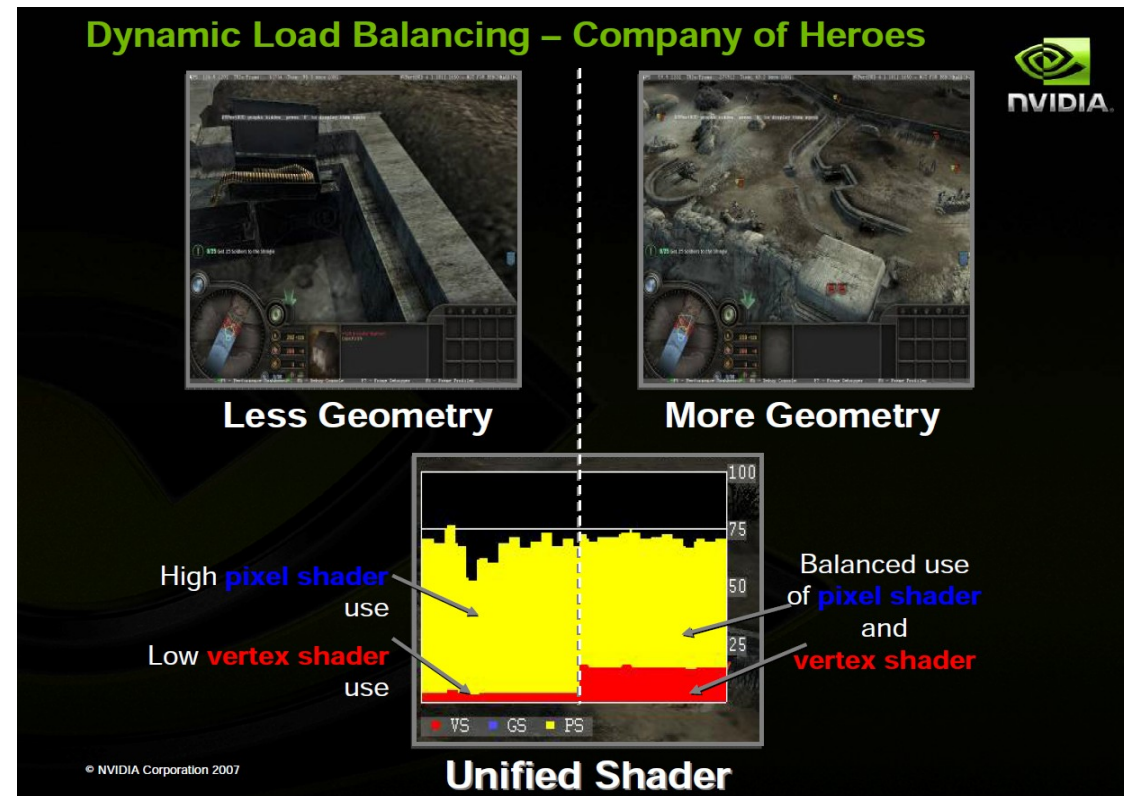
## Retour à un traitement générique

### 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 ?

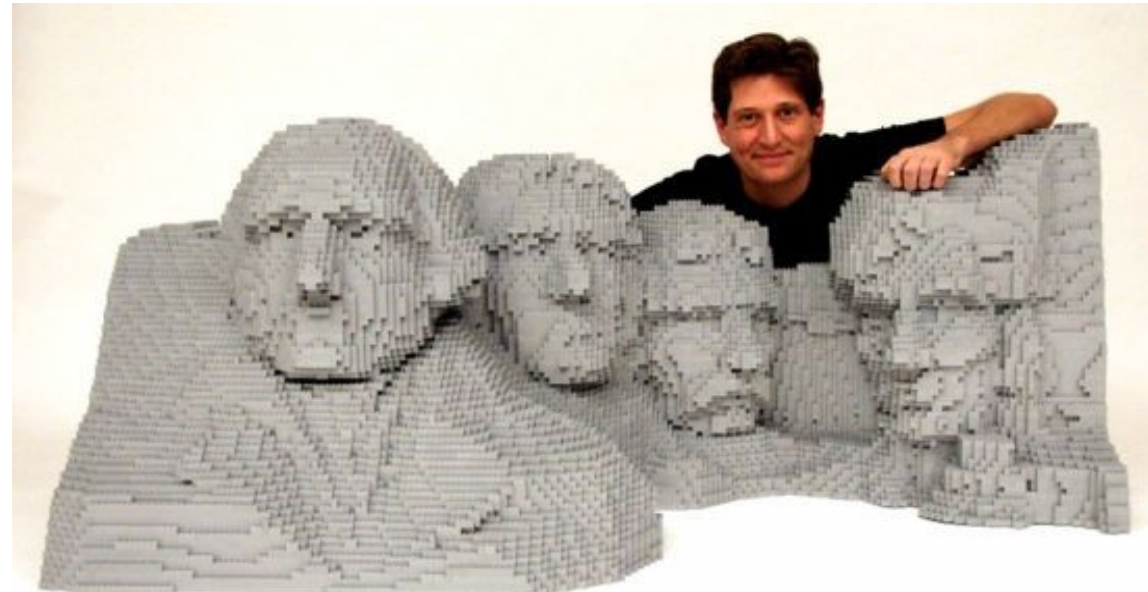


## Développer (de zéro)

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 bibliothèques 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

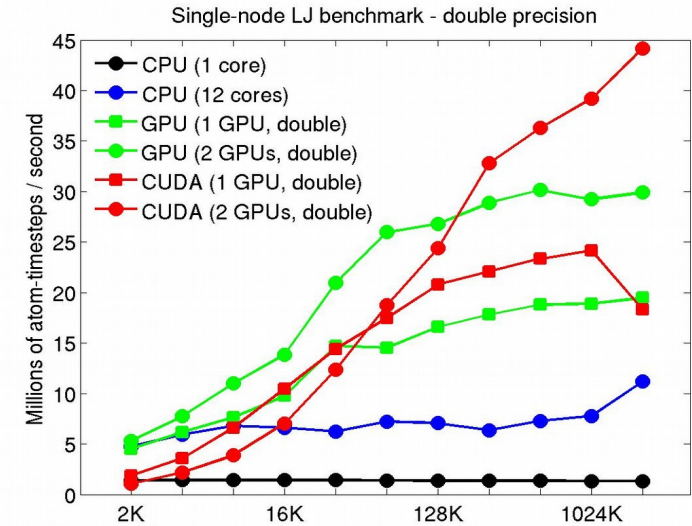
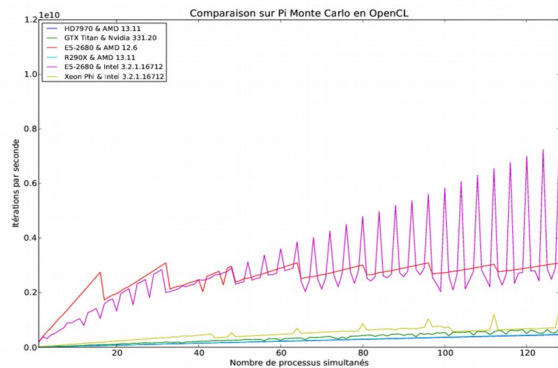
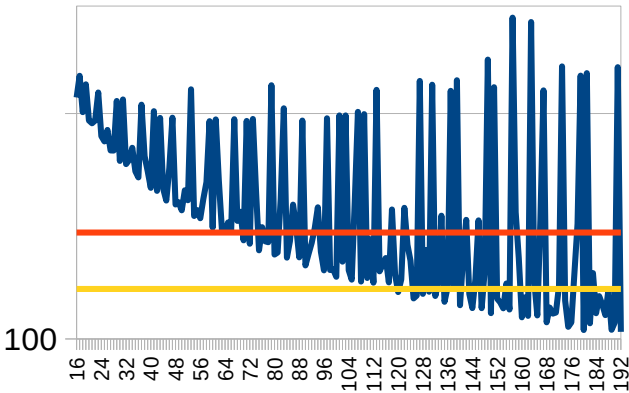
	Cluster	Node CPU	Node GPU	Node Nvidia	Accelerator
BLAS	BLACS MKL	OpenBLAS MKL	cBLAS	CuBLAS	OpenBLAS MKL
LAPACK	Scalapack MKL	Atlas MKL	cMAGMA	MAGMA	MagmaMIC
FFT	FFTw3	FFTw3	cFFT	CuFFT	FFTw3

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

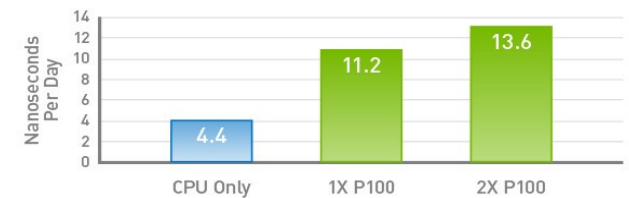
# Et mon code « pur » ?

- 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 » ?



## 768K Water Single-Node Performance



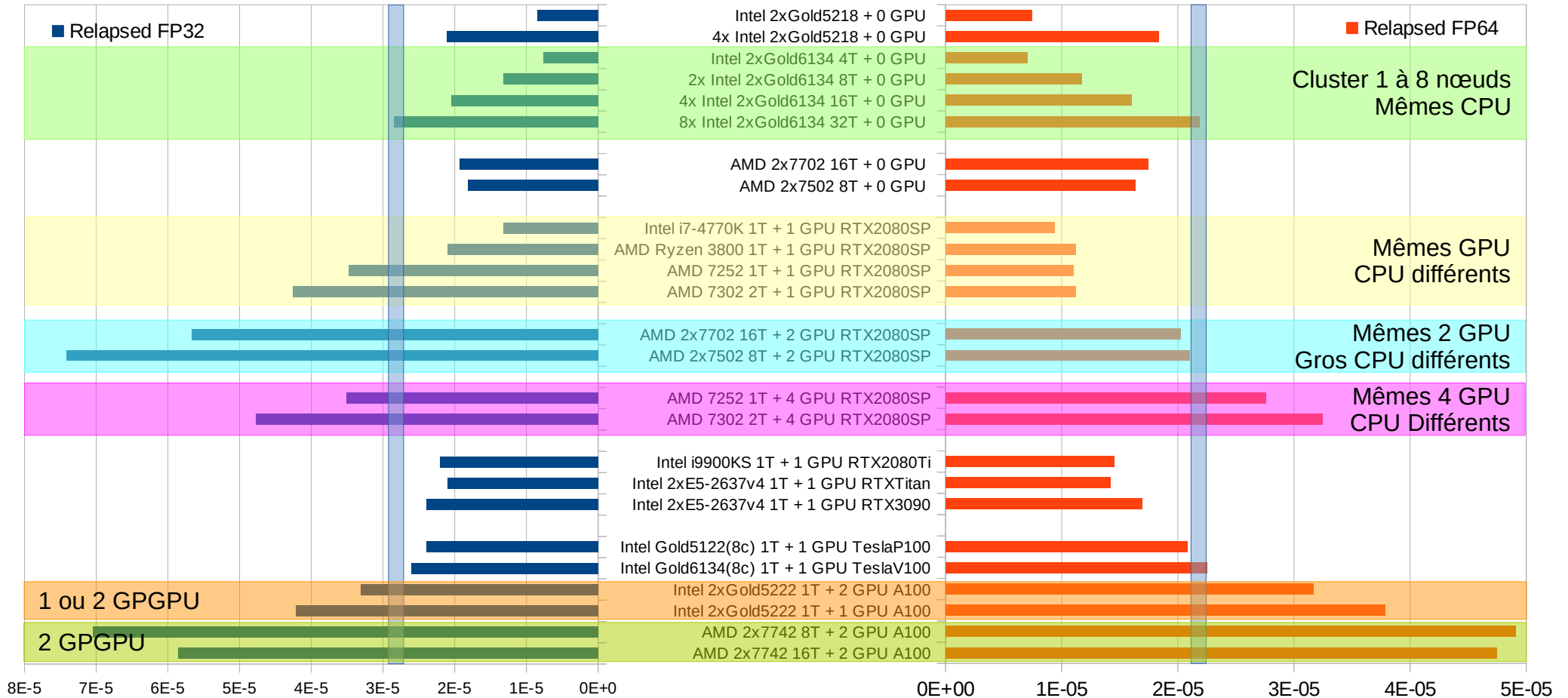
Dual-E5-2697 v4@2.30GHz 3.6GHz Turbo (Broadwell) HT Off performance on Dual-Haswell system

# 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

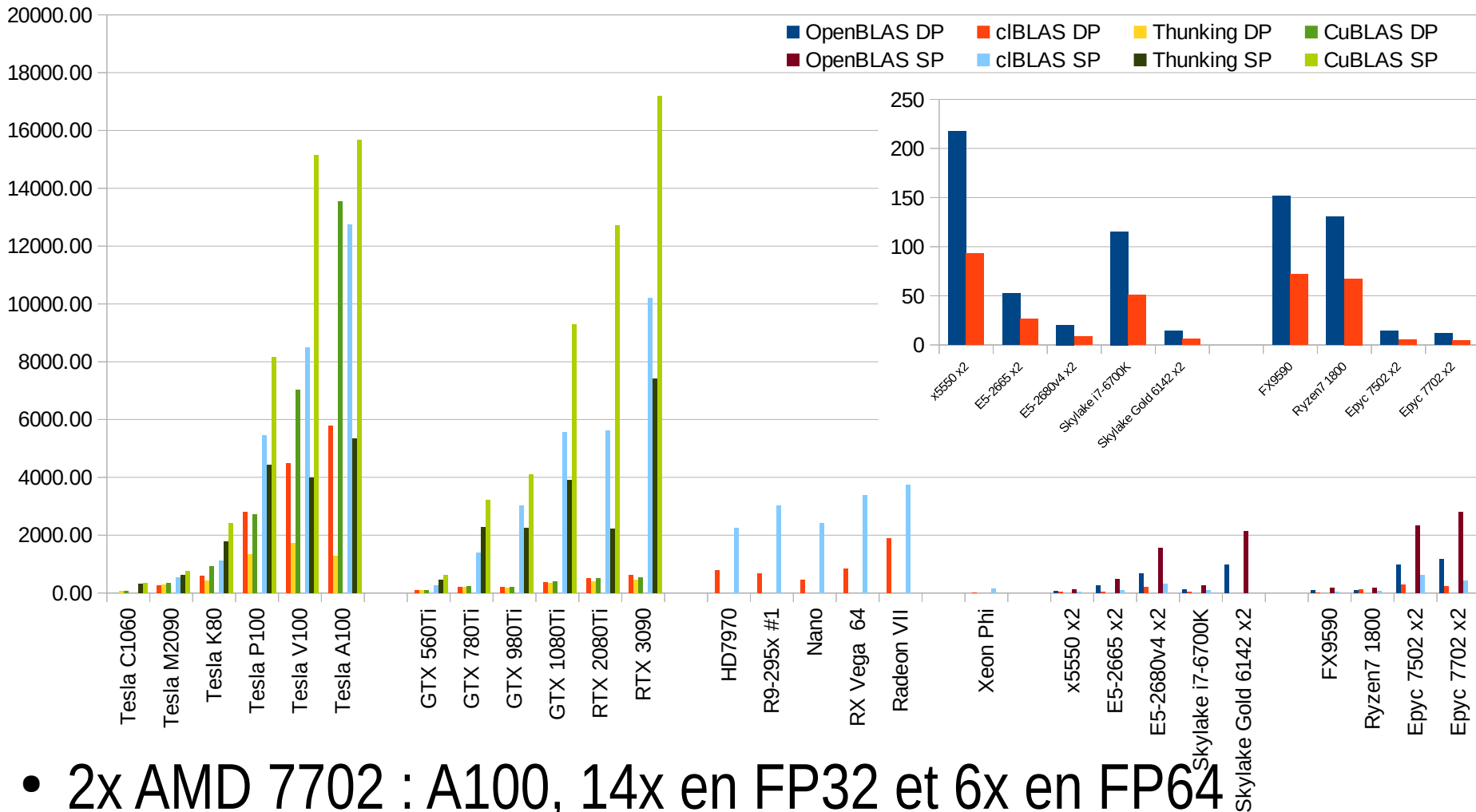
- Niveau 3 : opérations simples sur les matrices

- 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 ?

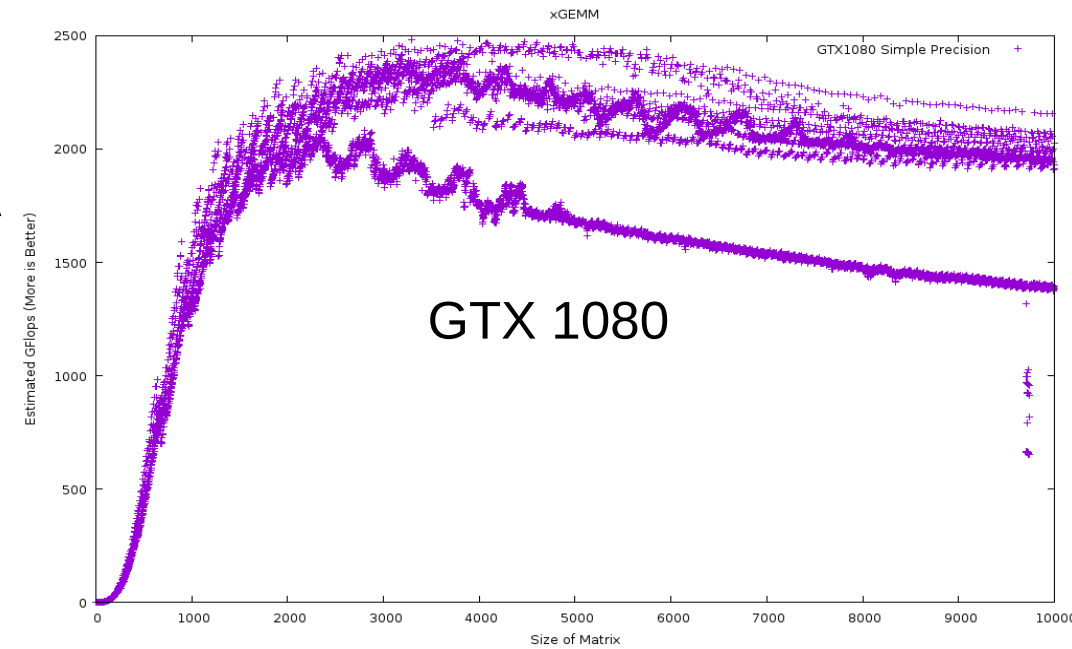
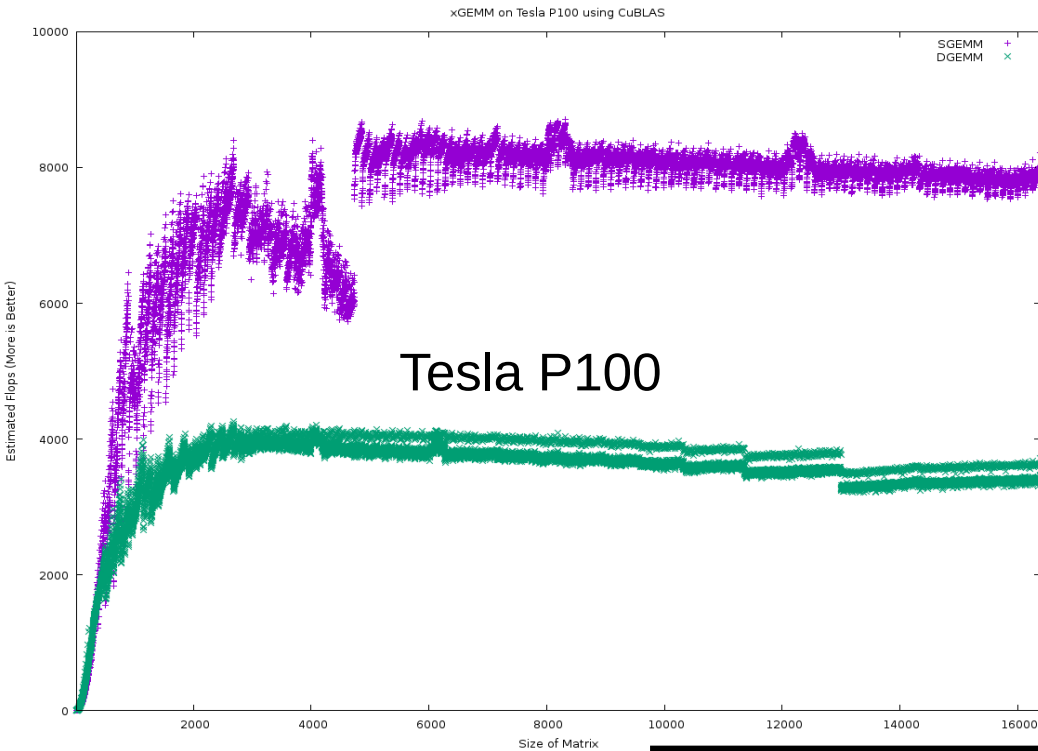
# Déjà en 2010, étrange...

## Le comportement Tesla C1060...

- Uniquement produit matriciel : xGEMM
- Propriété : Transposé  $(A * B) = \text{Transposé}(A) * \text{Transposé}(B)$
- 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

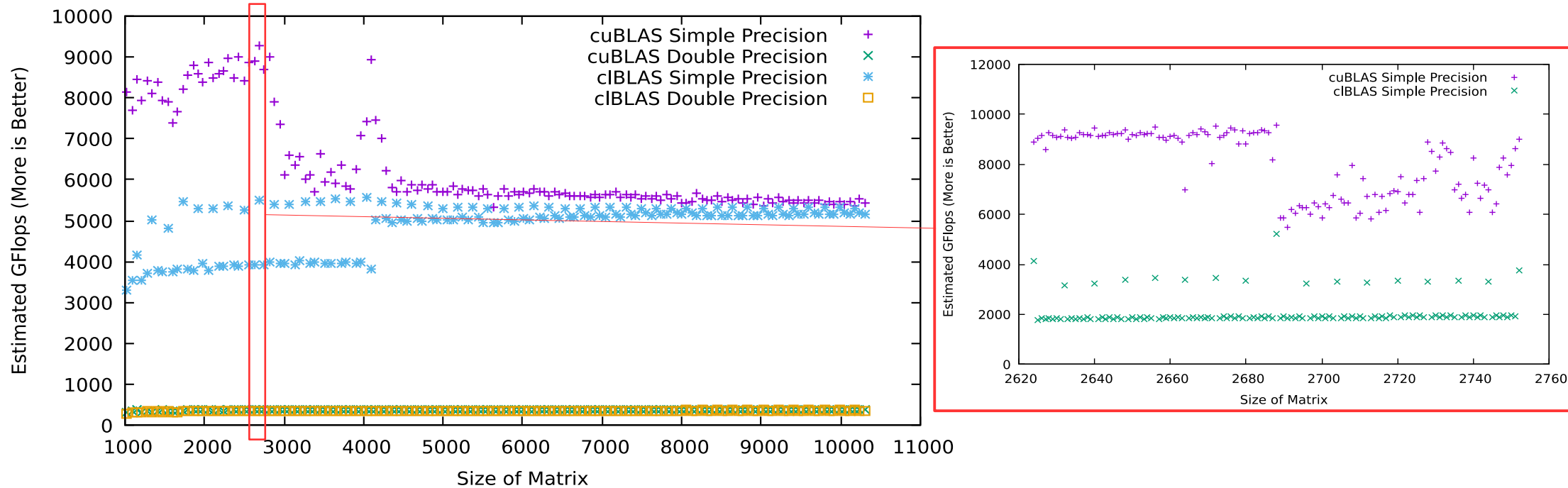


Taille

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

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

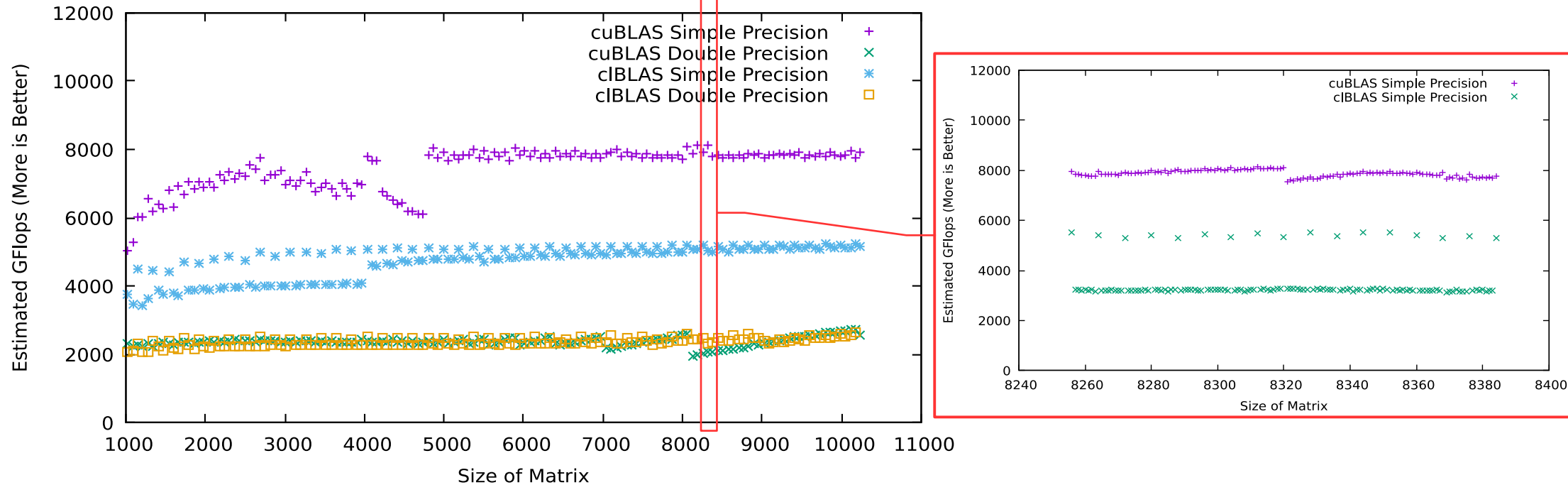
xGEMM for a Nvidia GTX 1080Ti: implementations cuBLAS and cBLAS



- Un facteur entre 20 et 30 entre simple & double précision
- Une implémentation cBLAS 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

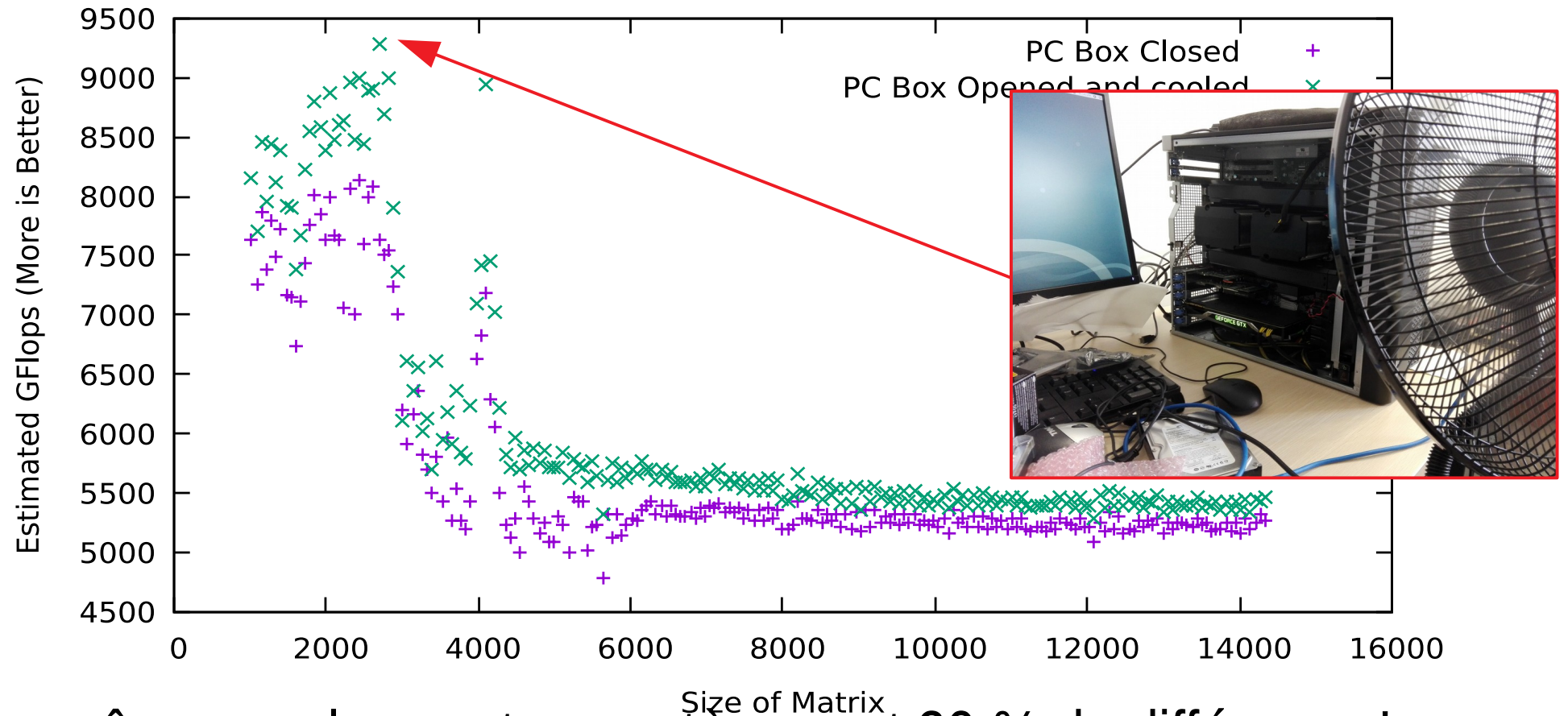
xGEMM for a Nvidia Tesla P100: implementations cuBLAS and cBLAS



- Un facteur autour de 4 entre simple & double précision
- Une implémentation cBLAS 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

xGEMM for a Nvidia GTX 1080Ti: performances for cuBLAS implementation

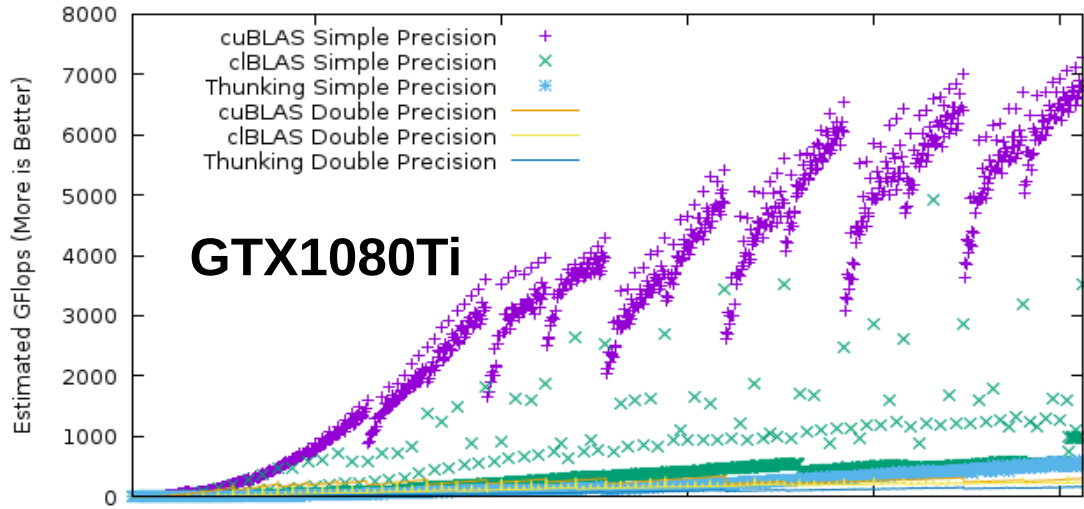


- Les mêmes socles, cartes, systèmes, et 20 % de différence !
  - De l'importance des conditions climatiques durant l'expérimentation...

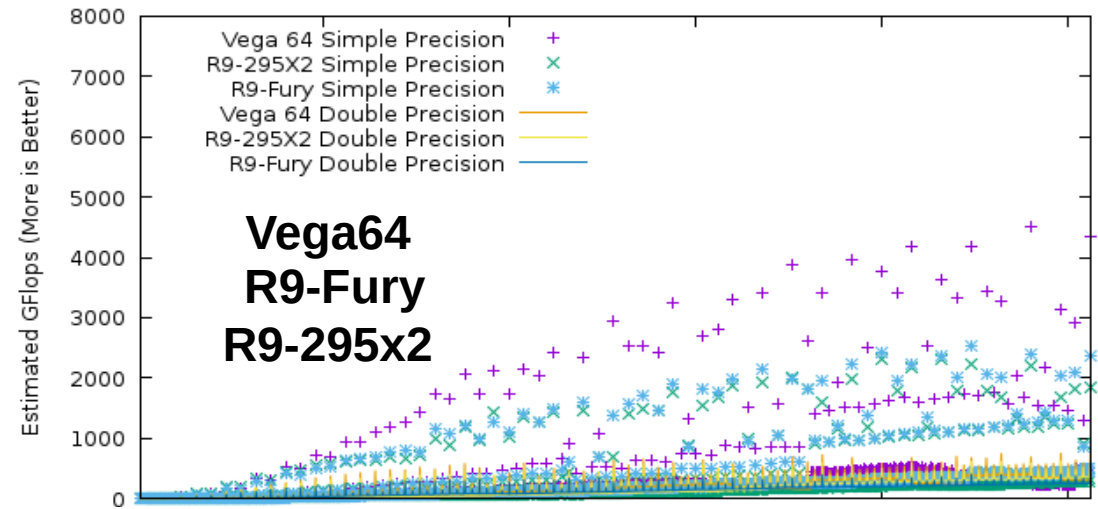


# xGEMM pour les « petites tailles »

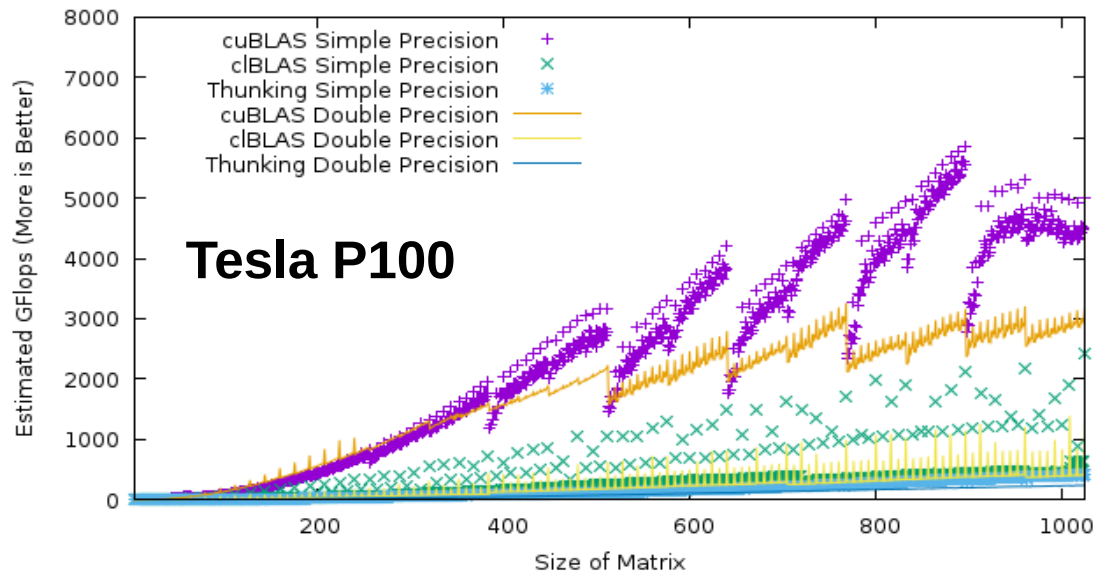
xGEMM for a Nvidia GTX 1080Ti: implementations cuBLAS, cBLAS and Thinking



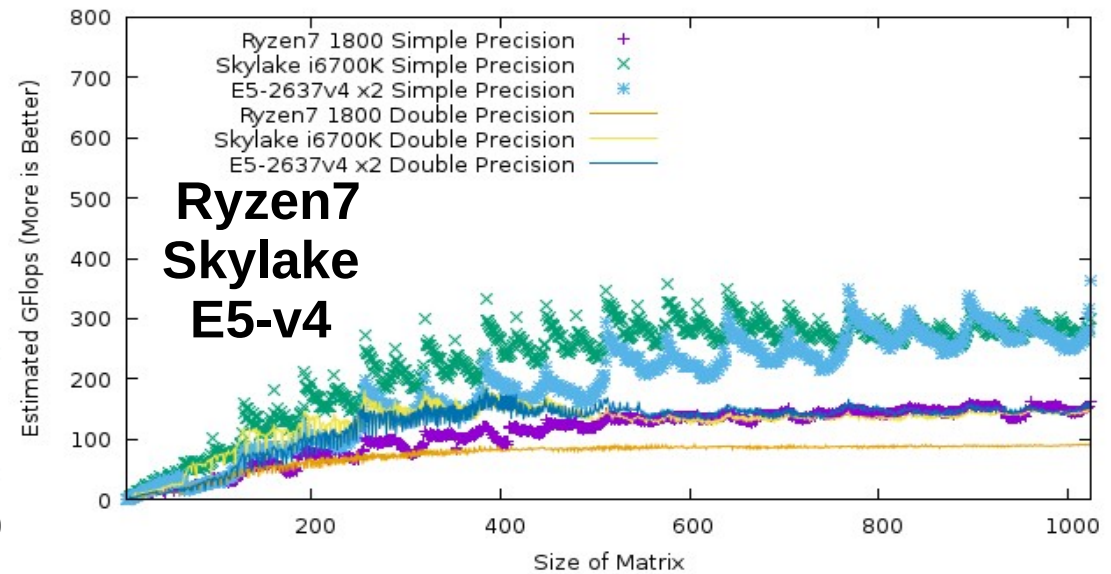
xGEMM for a AMD GPUs: cBLAS implementation



xGEMM for a Nvidia Tesla P100: implementations cuBLAS and cBLAS and Thinking



xGEMM for CPUs: OpenBLAS implementation



Oui ! CuBLAS ou cBLAS, 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 cBLAS 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  $C[i] = \text{somme normée de } y$  : fonction xNRM2
    - Application  $X \leftrightarrow Y$  : fonction xSWAP
- Bon maintenant il faut 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  $N \times N$
    - Je commence ma boucle
      - J'applique l'opération  $A.X$  qui donne  $Y$
      - Je résous 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_dnorm2(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

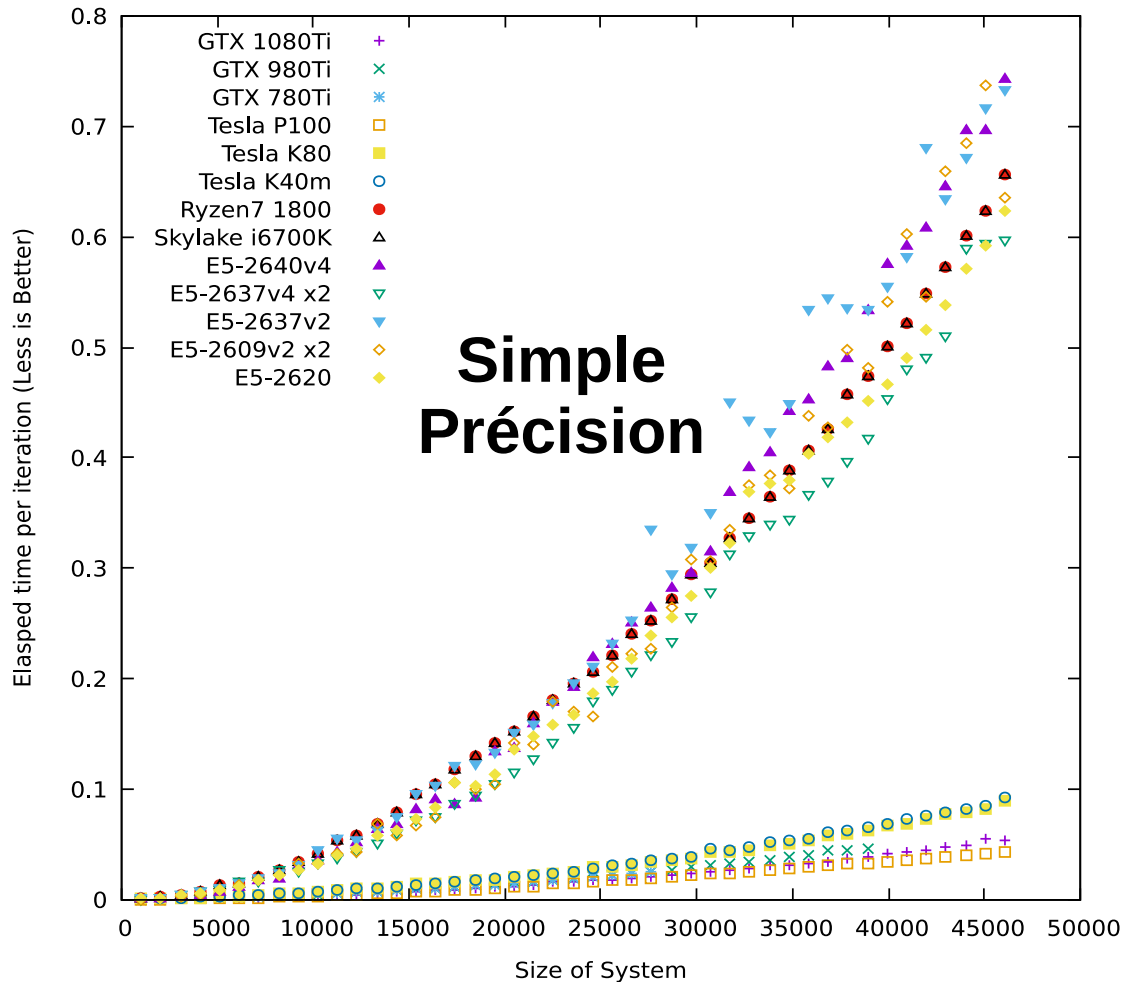
```
stat1=cublasGetVector(dim,sizeof(X[0]),devPtrX,
                    incx,X,incx);
stat2=cublasGetVector(dim,sizeof(Y[0]),devPtrY,
                    incx,Y,incx);

cublasFree (devPtrA);
cublasFree (devPtrX);
cublasFree (devPtrY);
if ((stat1 != CUBLAS_STATUS_SUCCESS) ||
    (stat2 != CUBLAS_STATUS_SUCCESS)) {
    wrapperError ("Strsv",
                CUBLAS_WRAPPER_ERROR_GET);
```

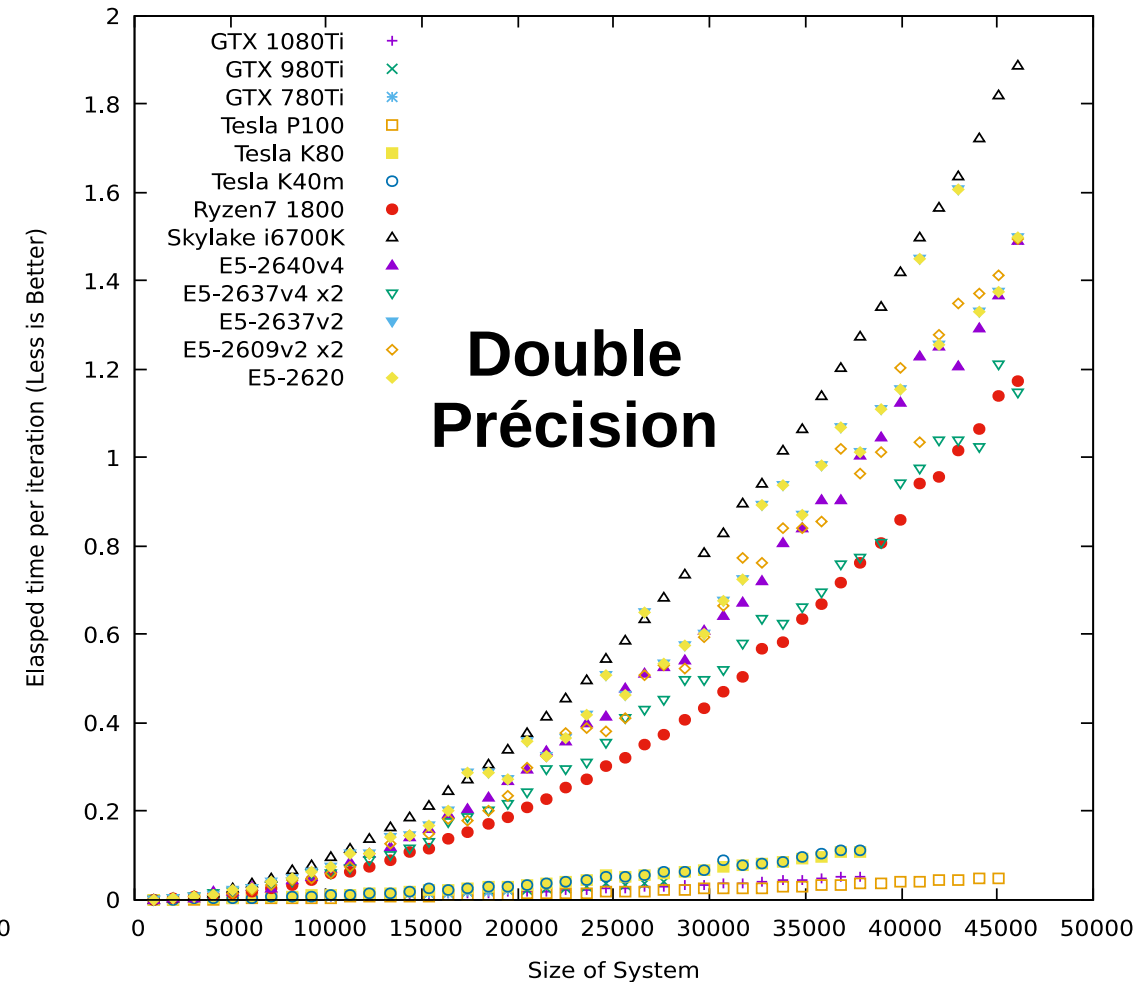
# Les premiers résultats...

## En temps écoulé par itération

BLAS bench in Simple Precision : native cuBLAS and OpenBLAS



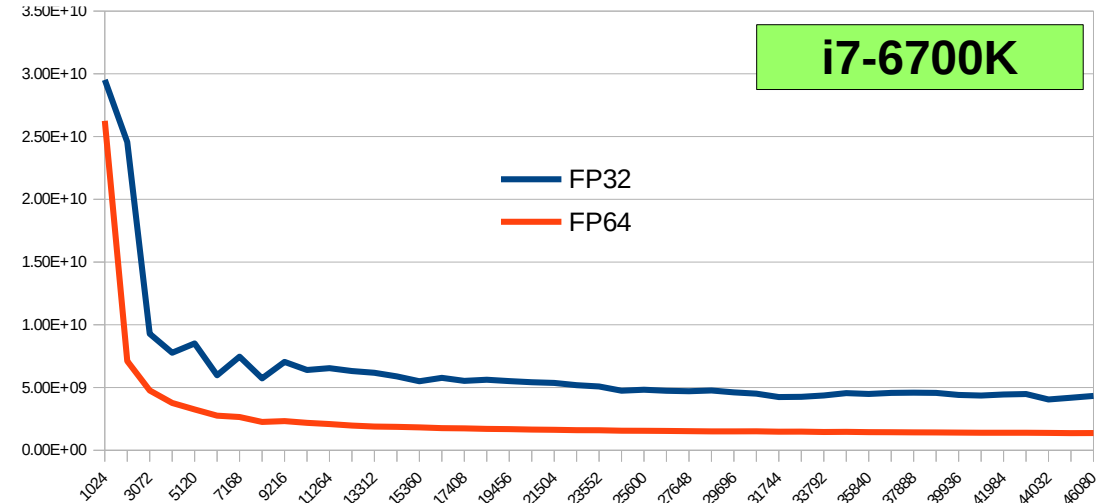
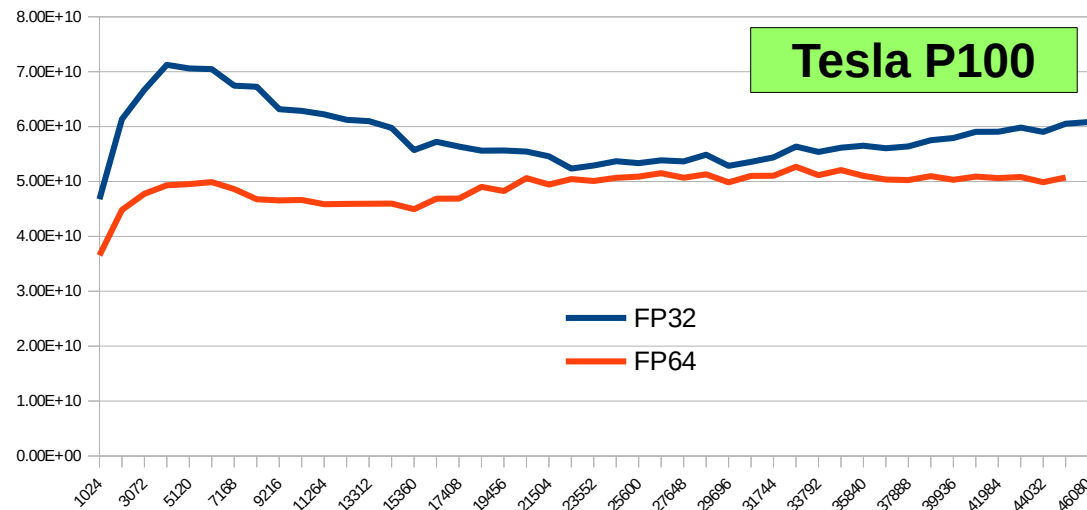
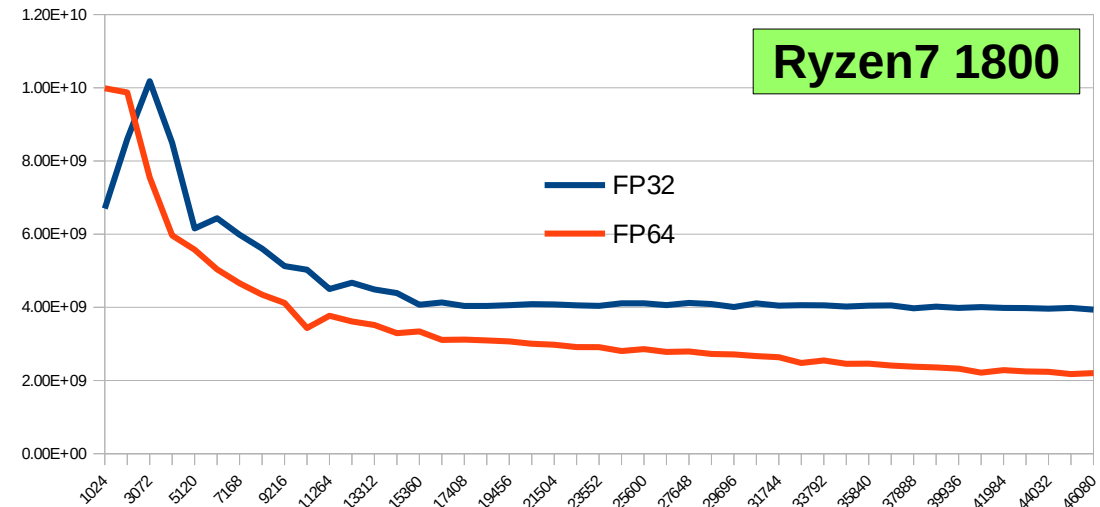
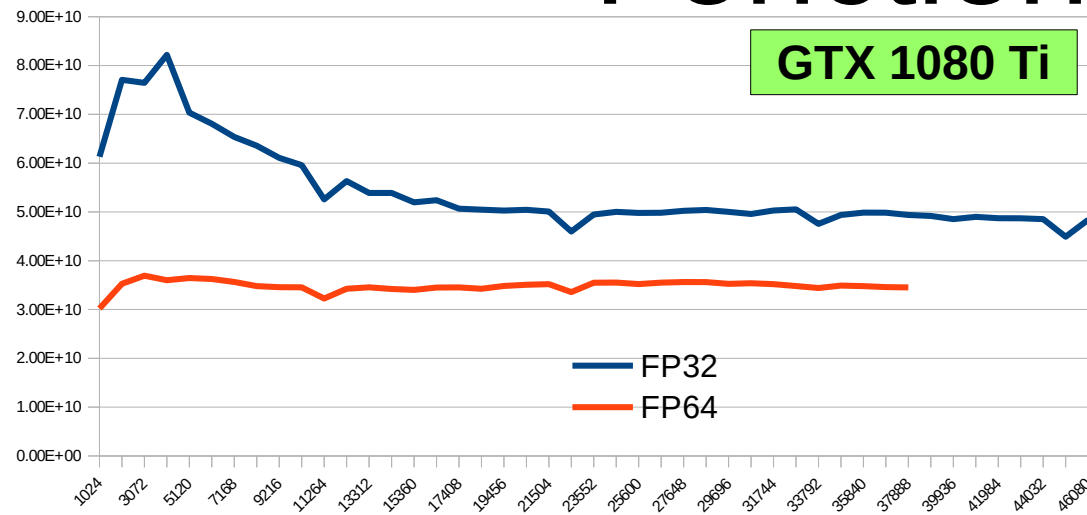
BLAS bench in Double Precision : native cuBLAS and OpenBLAS



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

# Résultats « bench BLAS »

## Fonction : $\text{Size}^2/T$

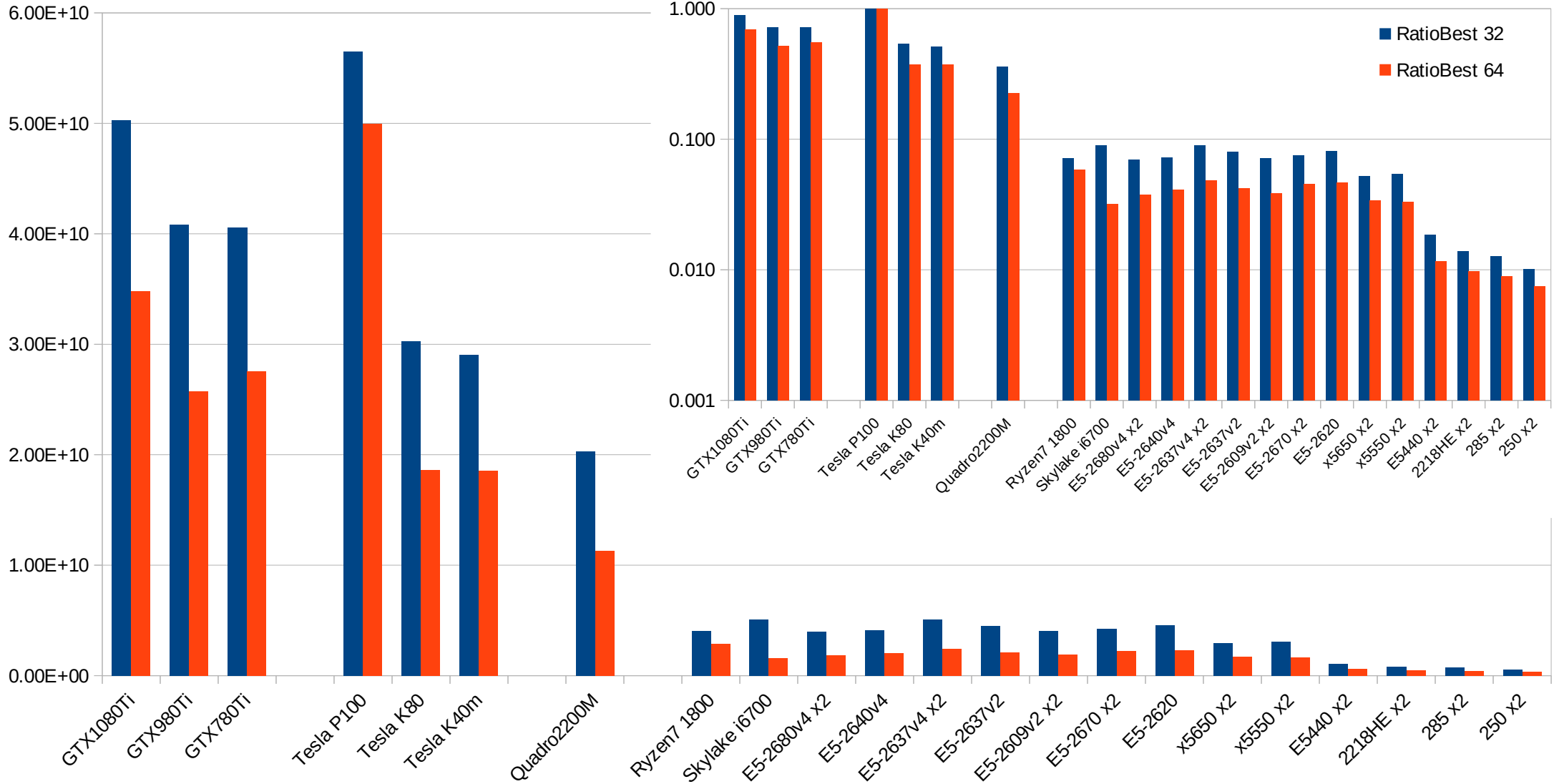


- Indicateur de « performances » : Médiane...



# 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 » : bibliothèques 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 circuits 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 rapidement
  - ... 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 synchronisation 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





# Code du noyau : Construction & 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);
```

Noyau OpenCL

```
    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;
}
```

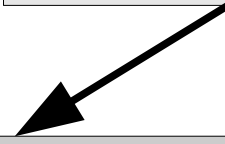
Appel Noyau

Nombre de lignes  
du noyau OpenCL

# Comment programmer en OpenCL ?

## Écrire « Hello World ! » en Python

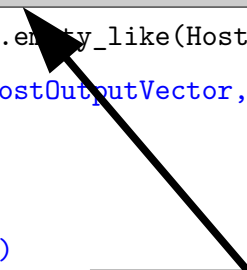
Noyau OpenCL



```
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("\n\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 :

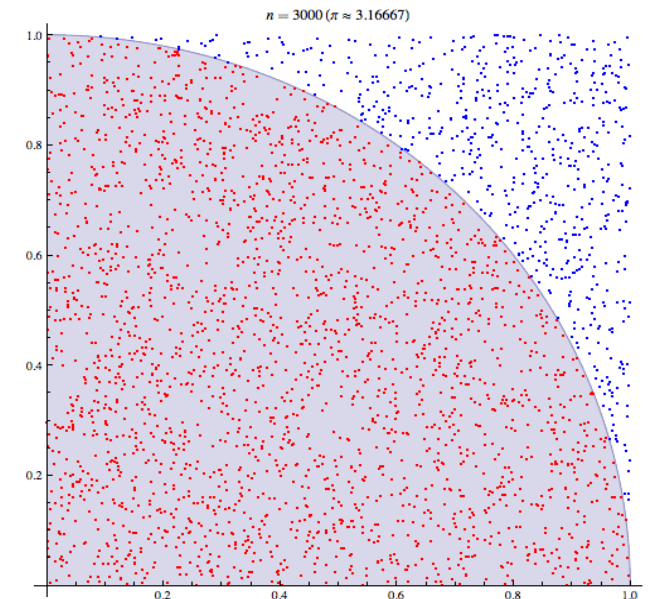
- Principale difficulté : maîtriser le « contexte »
  - « Ouvrir la boîte est plus difficile qu'assembler le meuble ! »
  - Nécessité de simplifier les appels par une API
- Pas de compatibilité entre les API de AMD et Nvidia
  - Chacun réécrit la sienne !

- 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

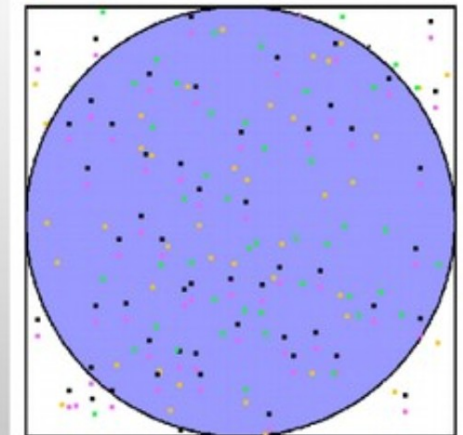


LLNL-PRES-XXXXXX

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



task 1  
task 2  
task 3  
task 4

Lawrence Livermore National Laboratory

LLNL-PRES-XXXXXX

# D'une version très monolithique...

## Variable & RNG fixés...

```
unsigned int jcong=seed_z+i;
```

```
#define CONG (jcong=69069*jcong+1234567)
```

```
#define CONGfp CONG * 2.328306435454494e-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 1.0
    #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 ?

### Appel *Offload* sur Xeon Phi MIC

#### 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);`
- `}`

- `#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

## Définition des attributs du noyau à appeler

- `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);`
- `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 ?

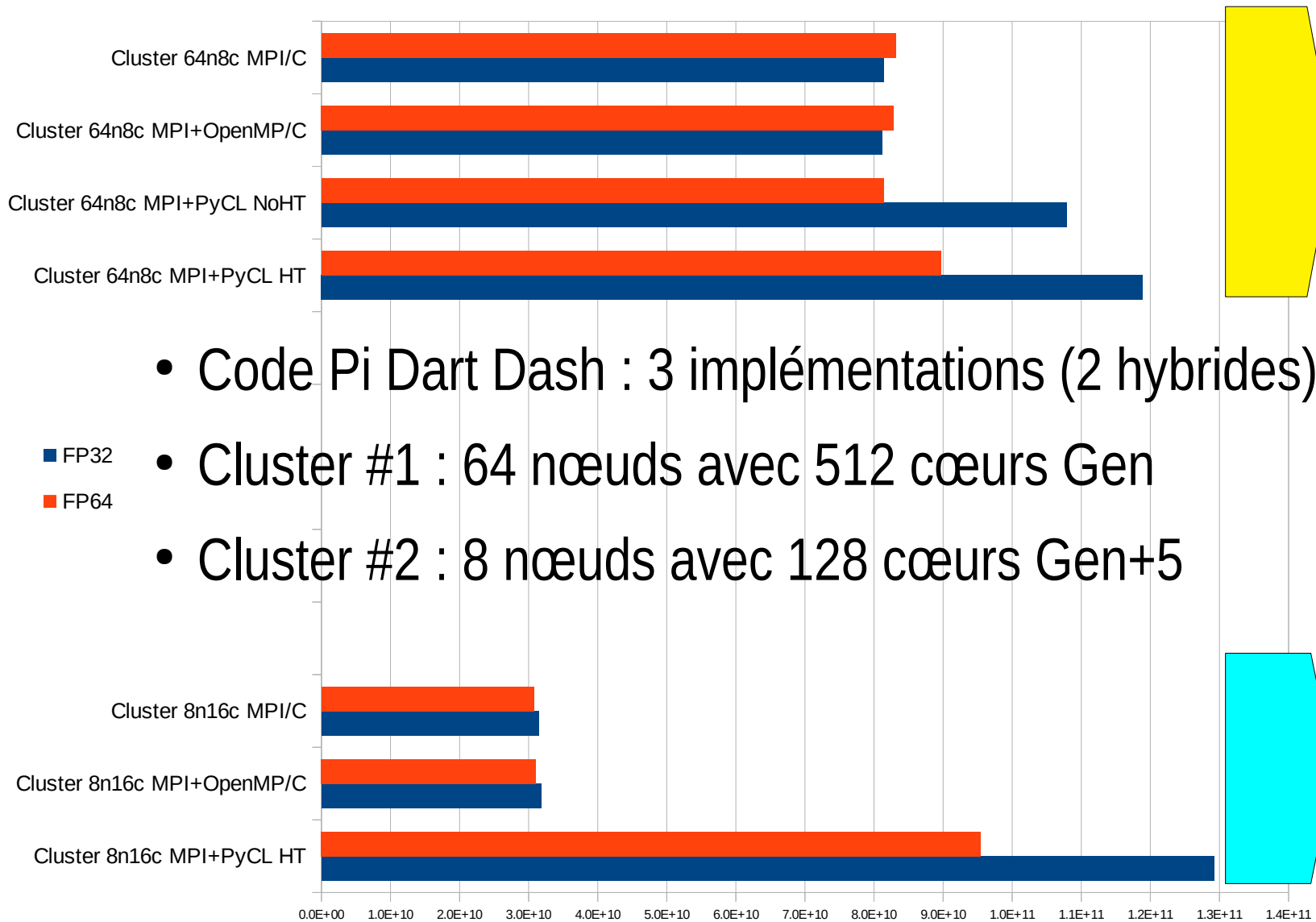
## Très simple implémentation...

- 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

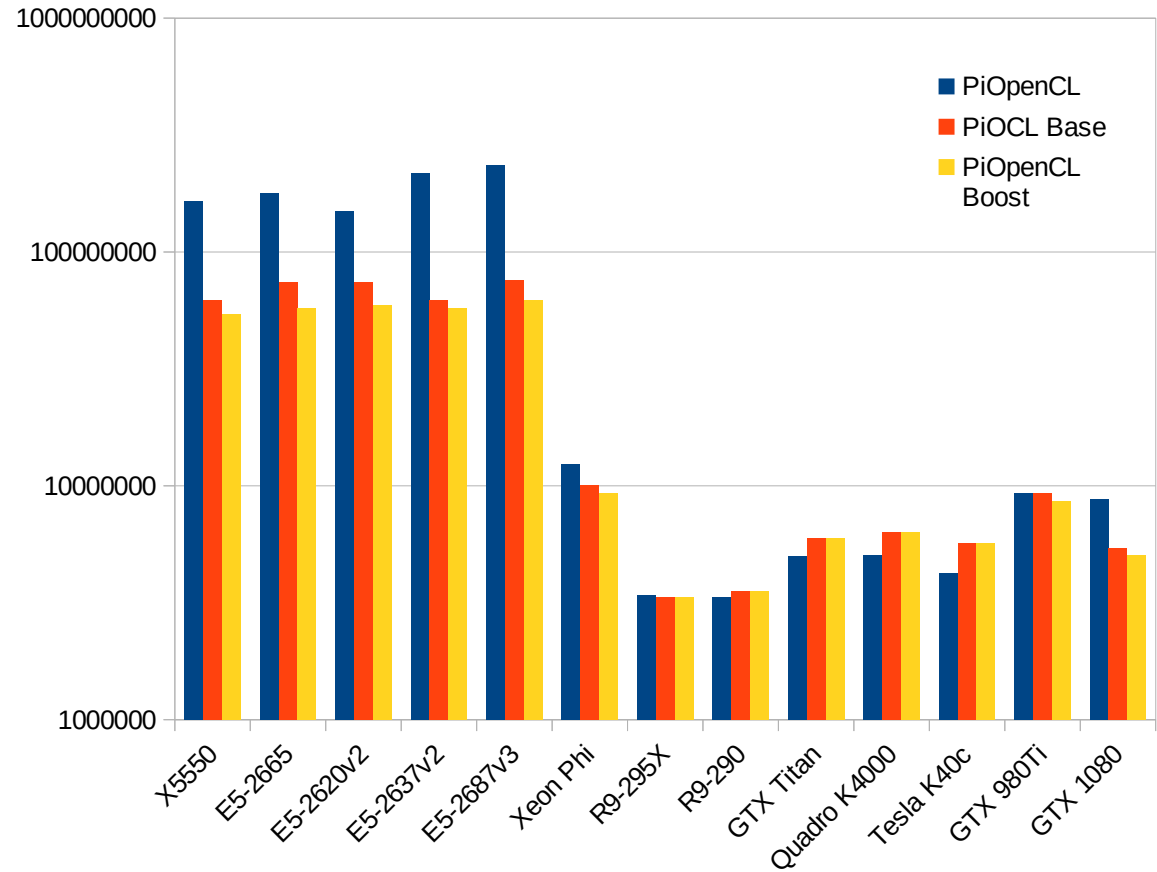


- Code Pi Dart Dash : 3 implémentations (2 hybrides)
- Cluster #1 : 64 nœuds avec 512 cœurs Gen
- Cluster #2 : 8 nœuds avec 128 cœurs Gen+5

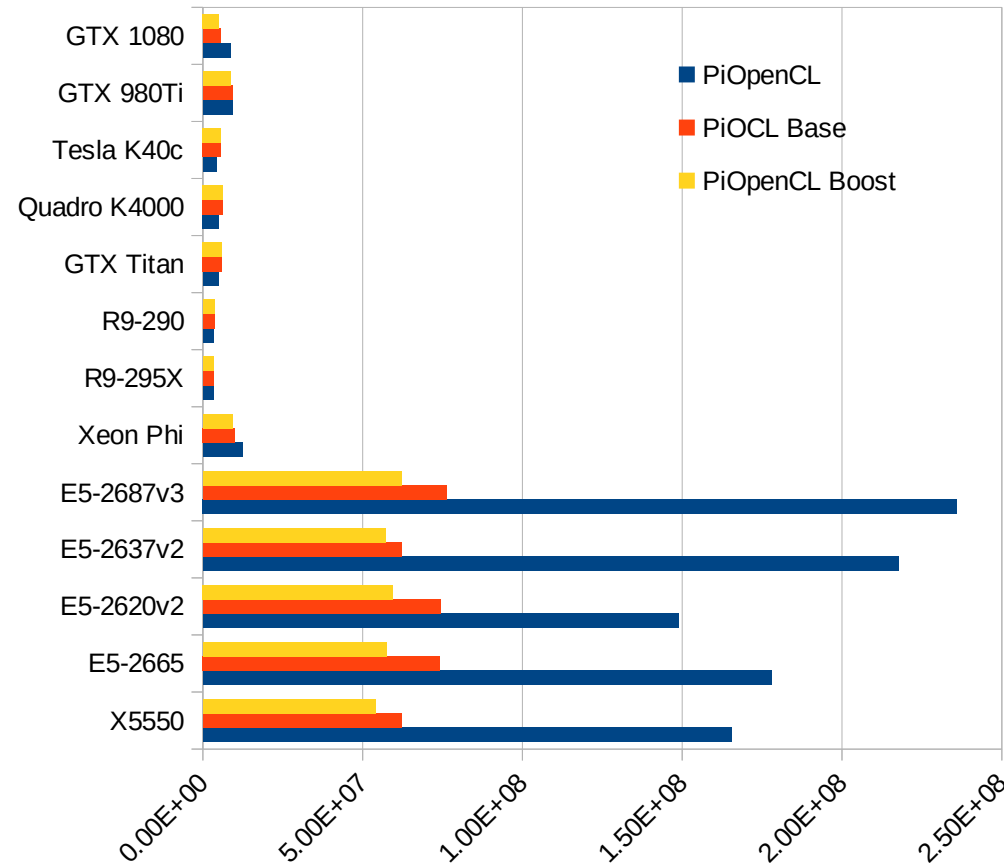


# Pour 1 de PR, bas régime, pour Pi Le CPU explose le GPU!

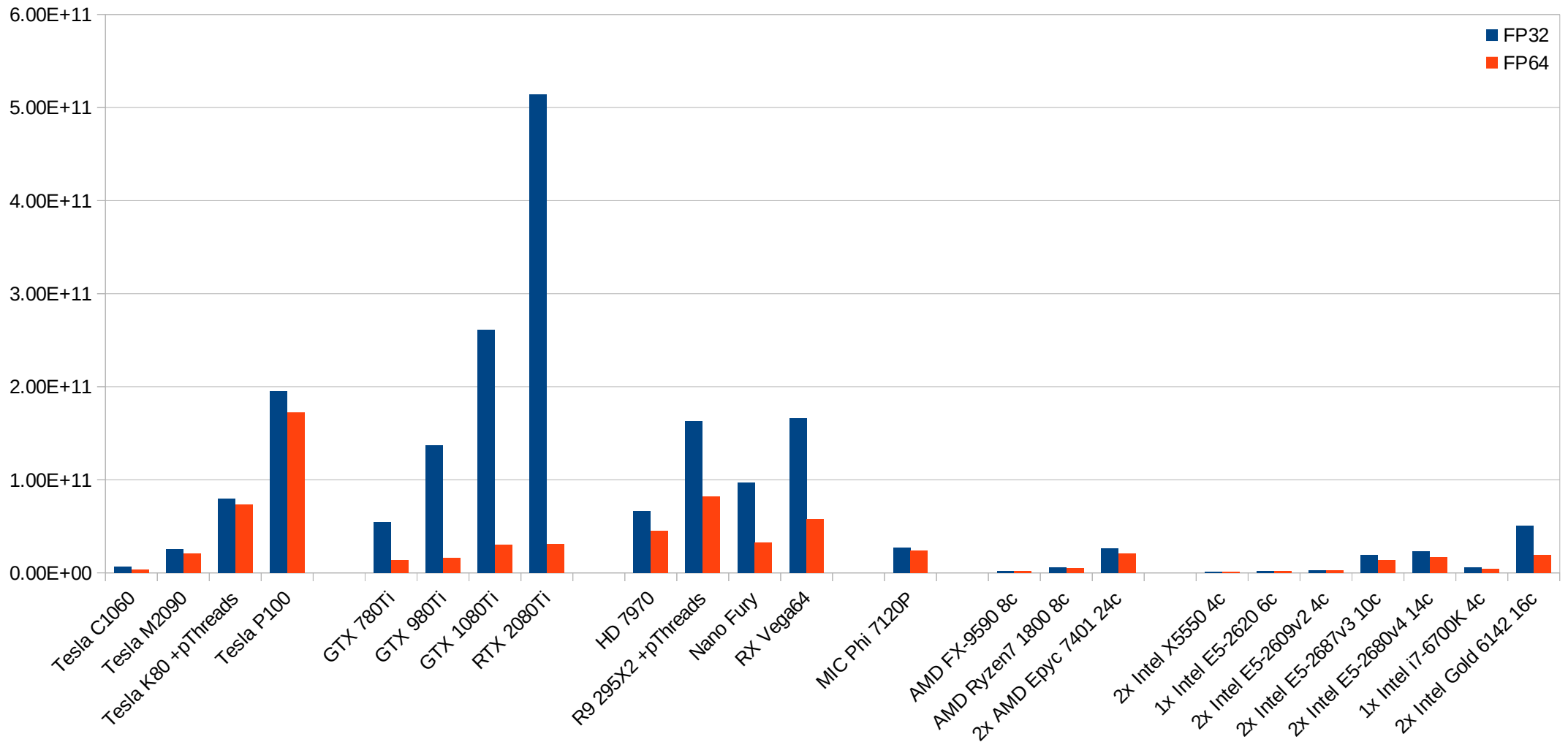
- De 20 à 50x plus lent !



- 1.5 ordre de magnitude



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

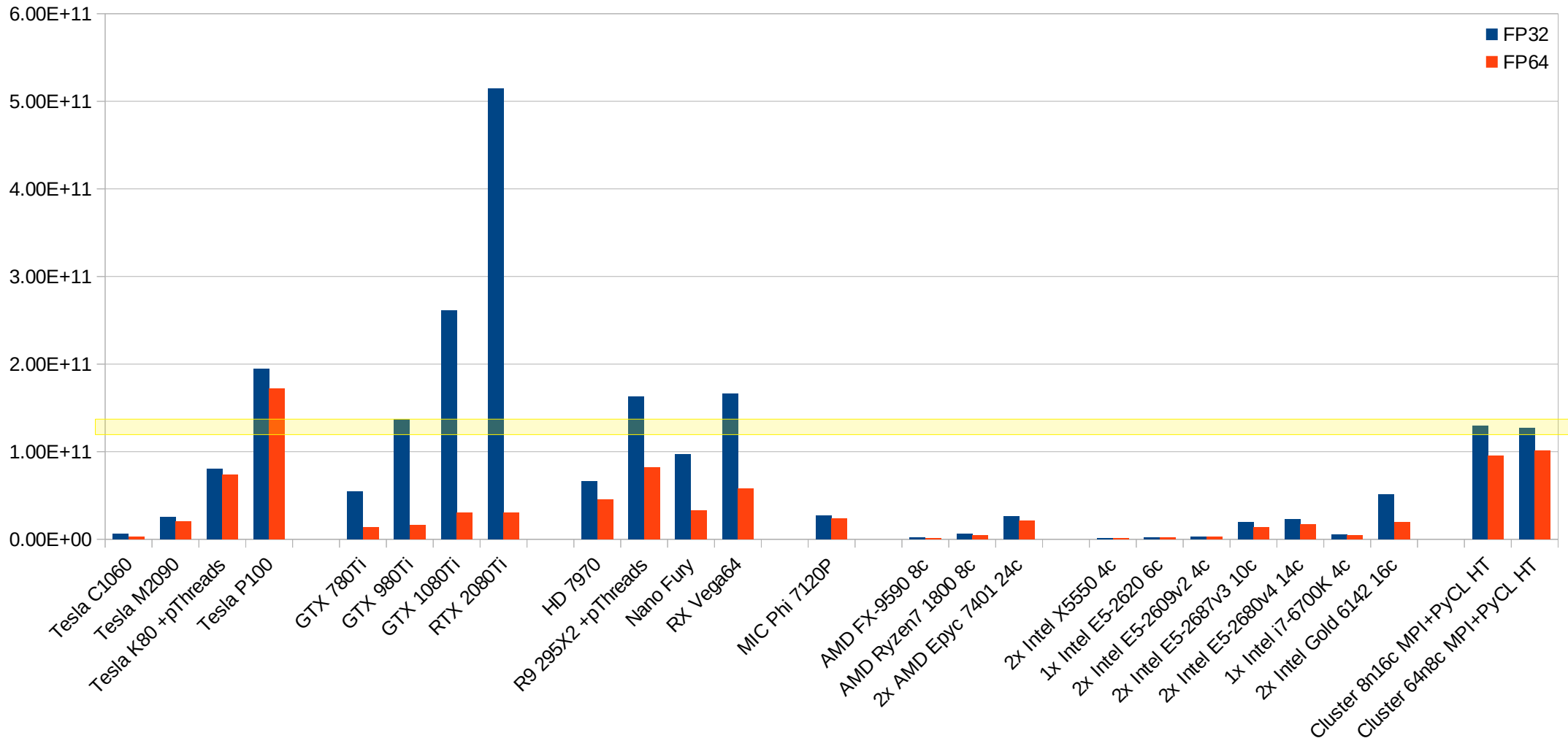


Mais en intégrant les clusters, ça donne quoi ?



# Pi Dart Dash les clusters en plus...

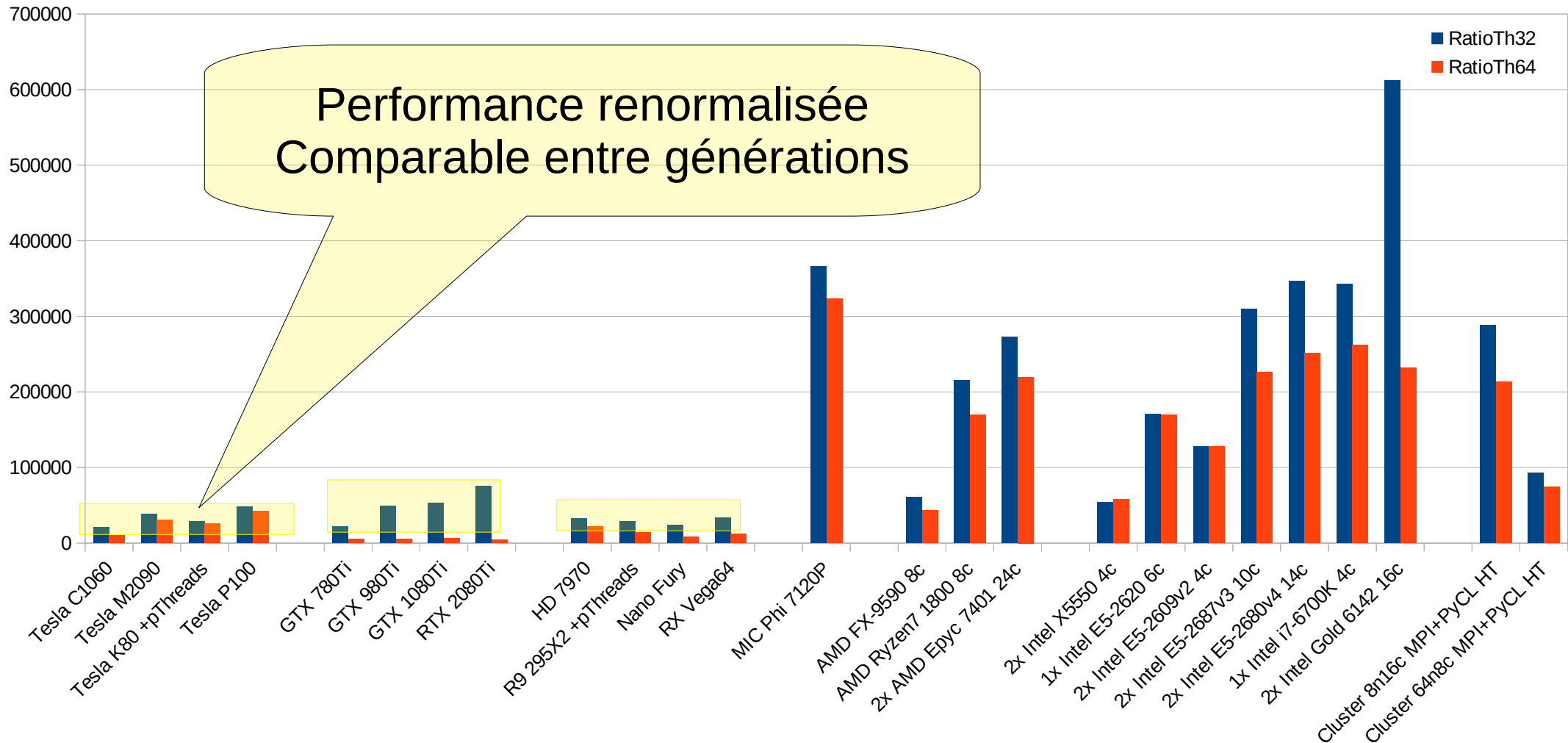
## A oui quand même...



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

# Le rapport à la puissance « théorique » ?

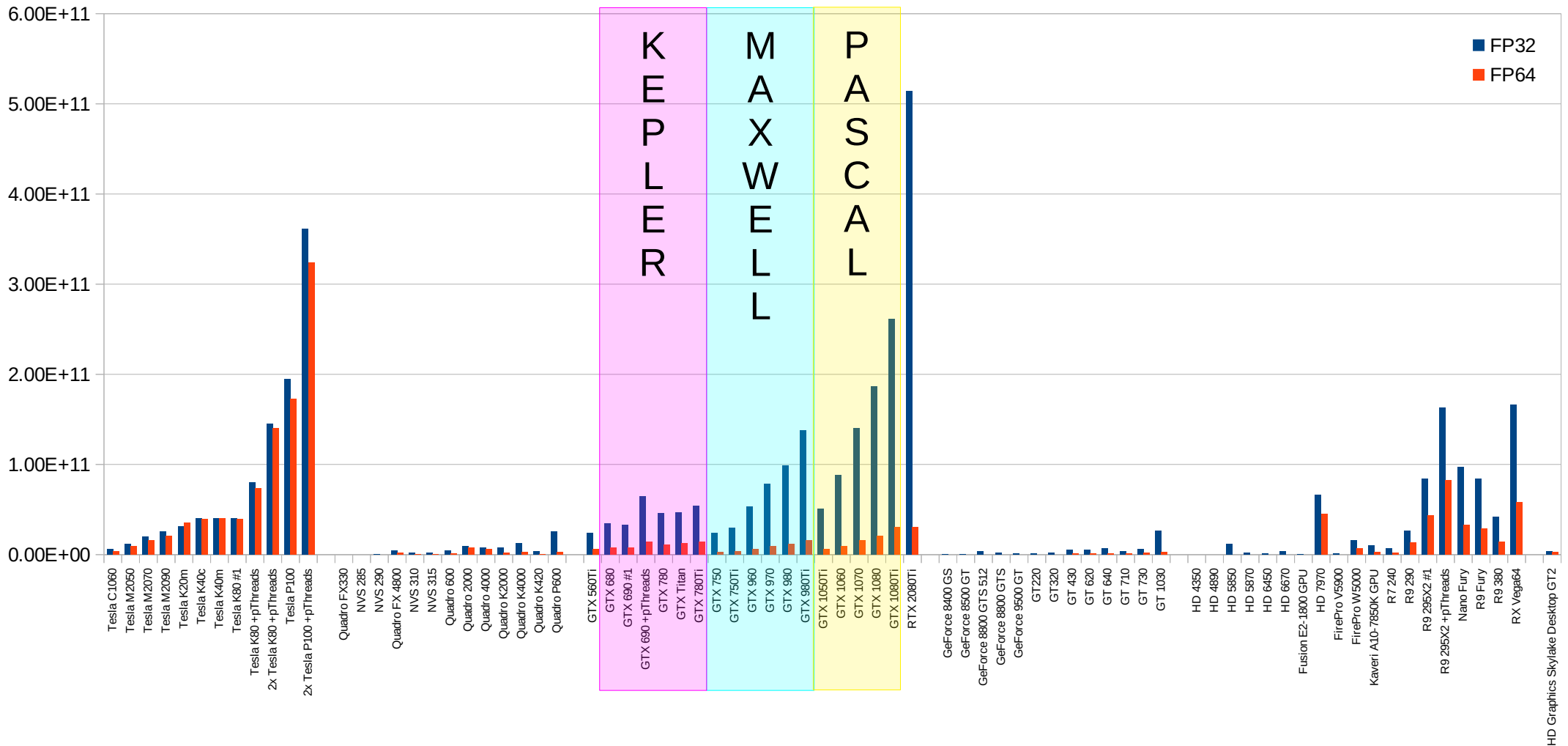
## Itops sur coeurs\*fréquence



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

# Sur le bestiaire du CBP

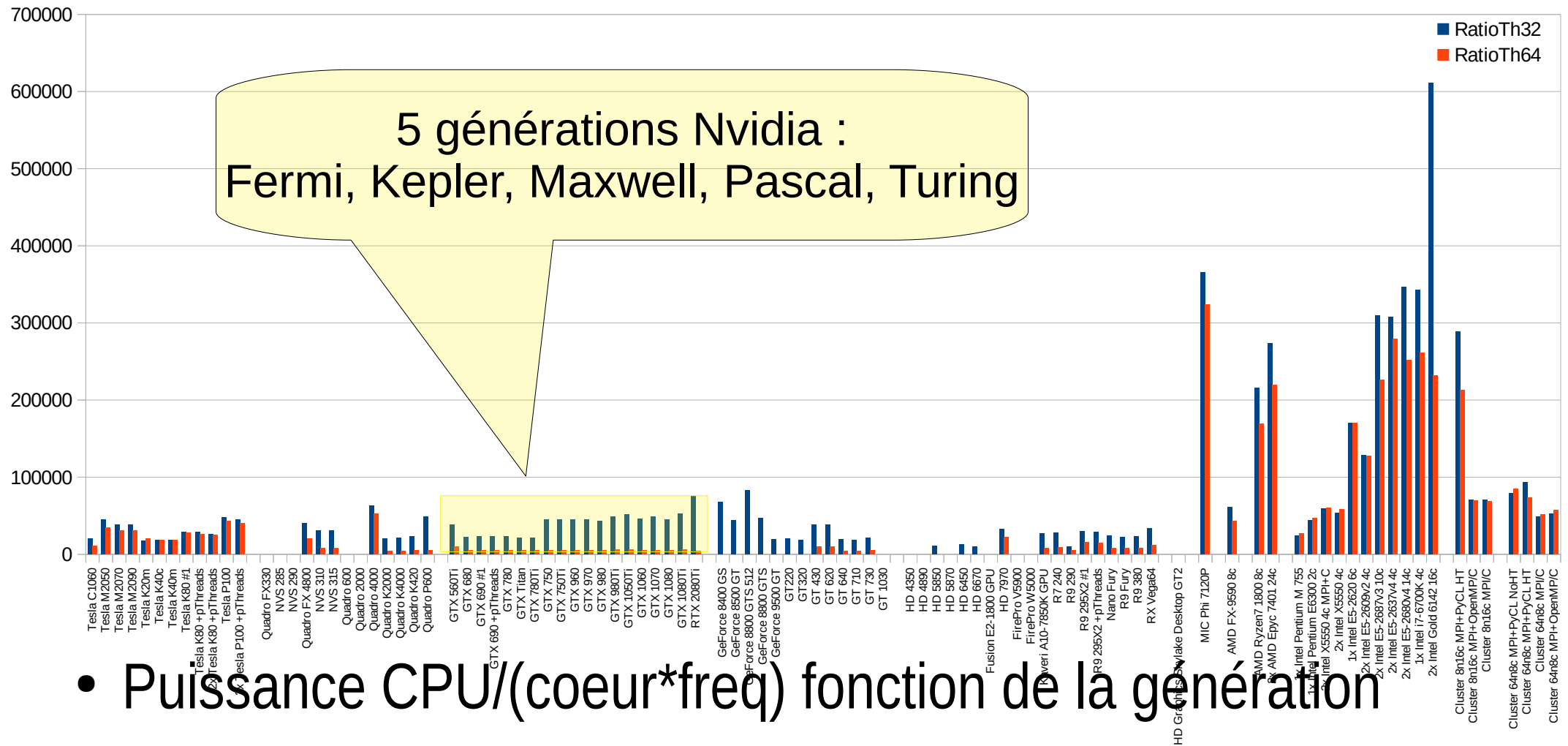
## De génération en génération



A chaque génération, toute une gamme de performances !

# Puissance renormalisée CPU/GPU

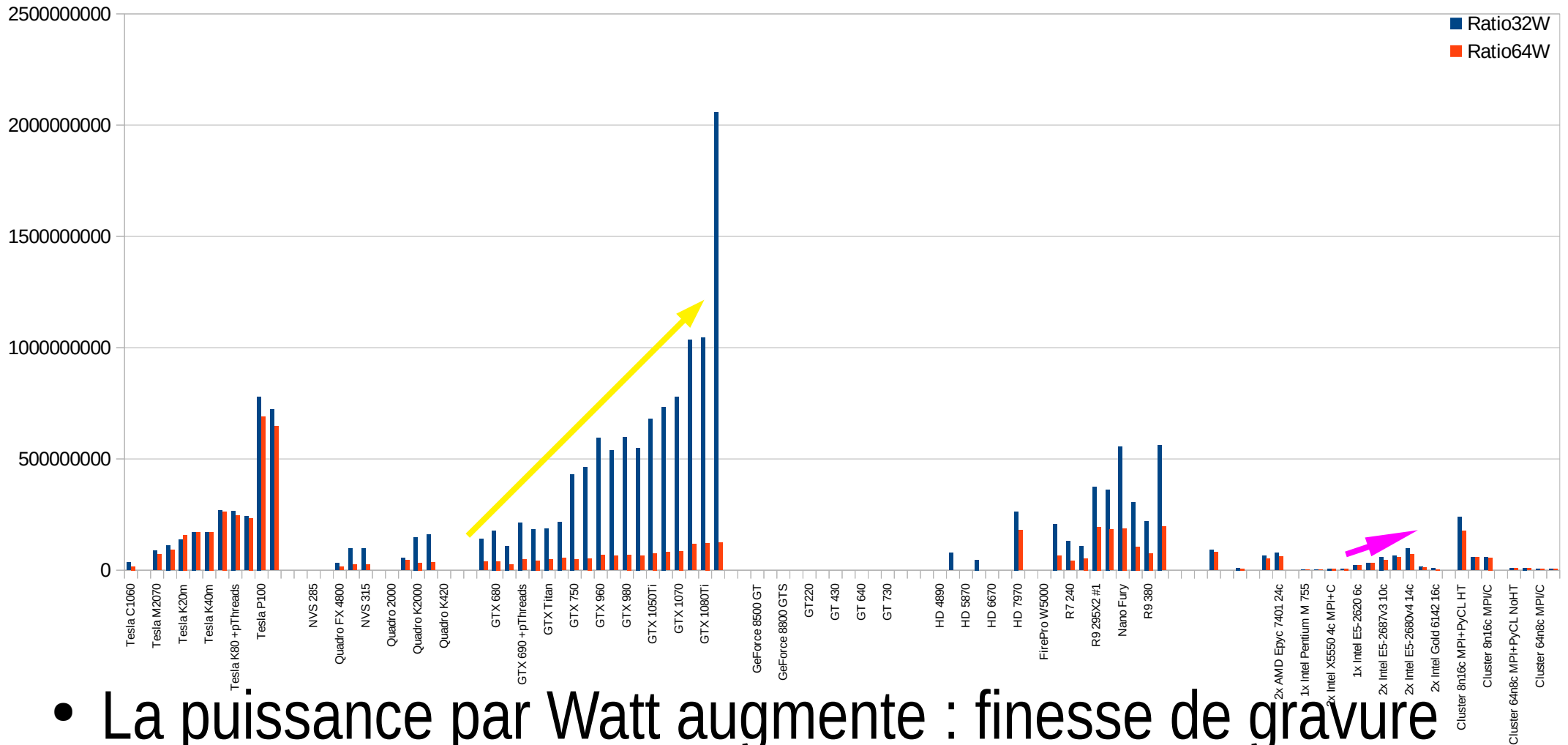
## Assertion généralisable ?



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

# La puissance pour l'écologiste

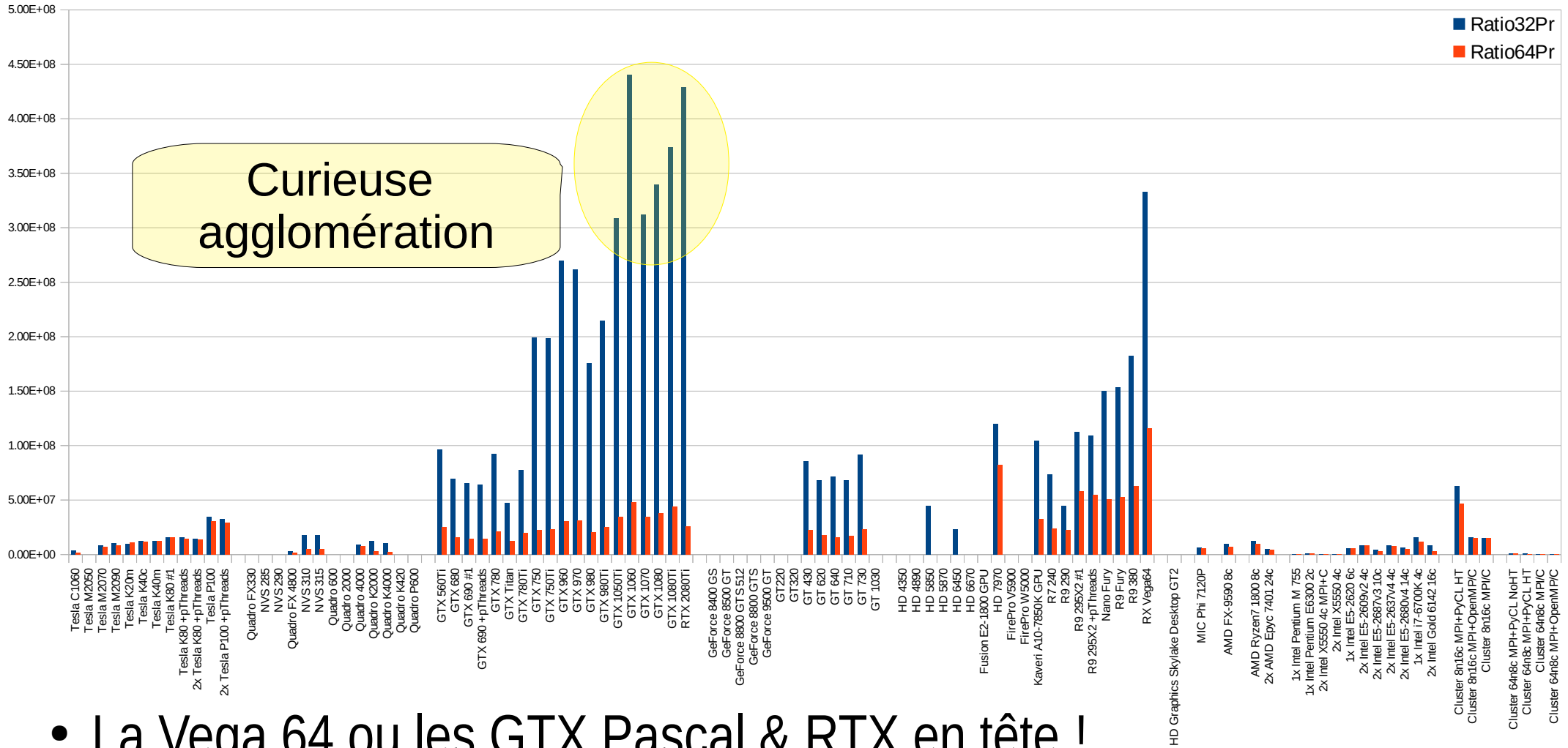
## Renormalisation à la TDP



- 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

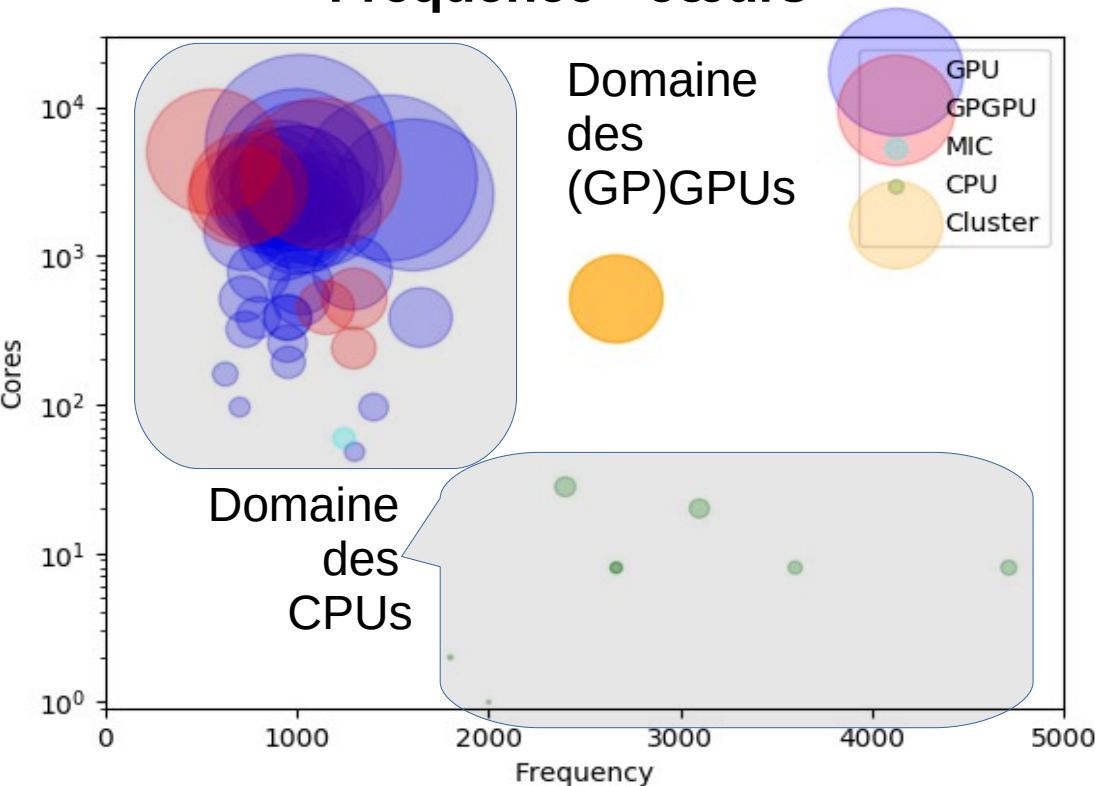


- La Vega 64 ou les GTX Pascal & RTX en tête !
- A croire qu'ils utilisent ce « test » pour leurs prix ;-)

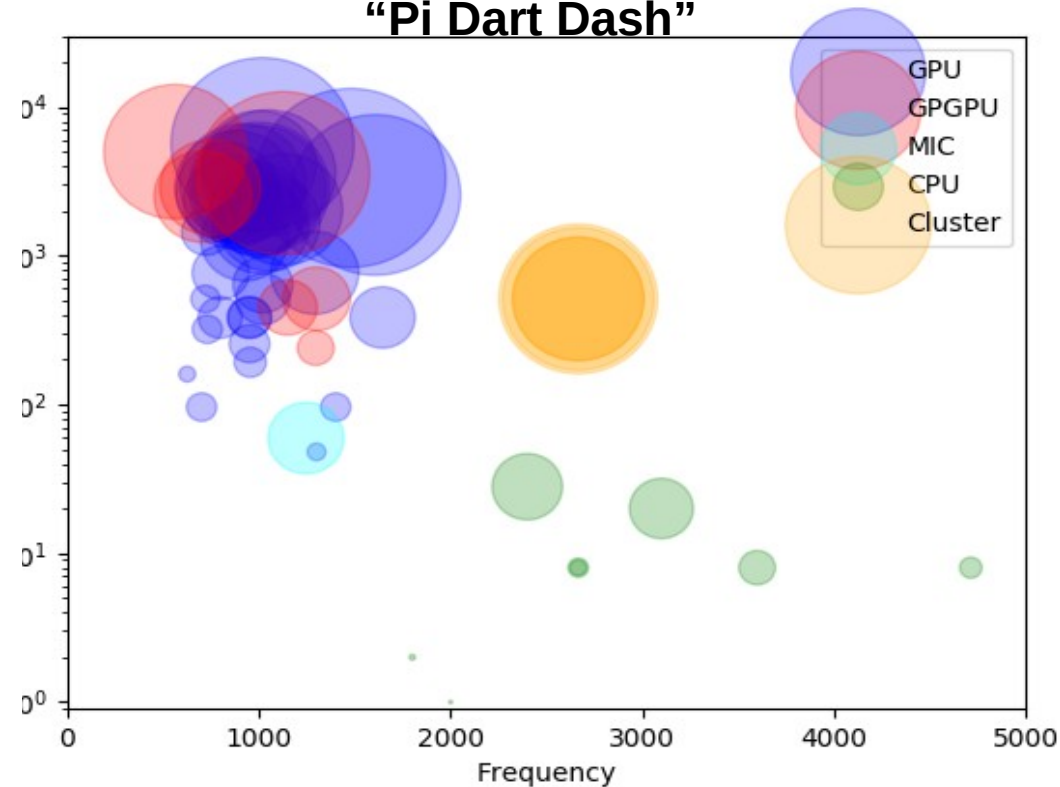
# Représenter les performances...

## Question de battement, de cœurs ?

Performance Théorique  
Fréquence \* cœurs



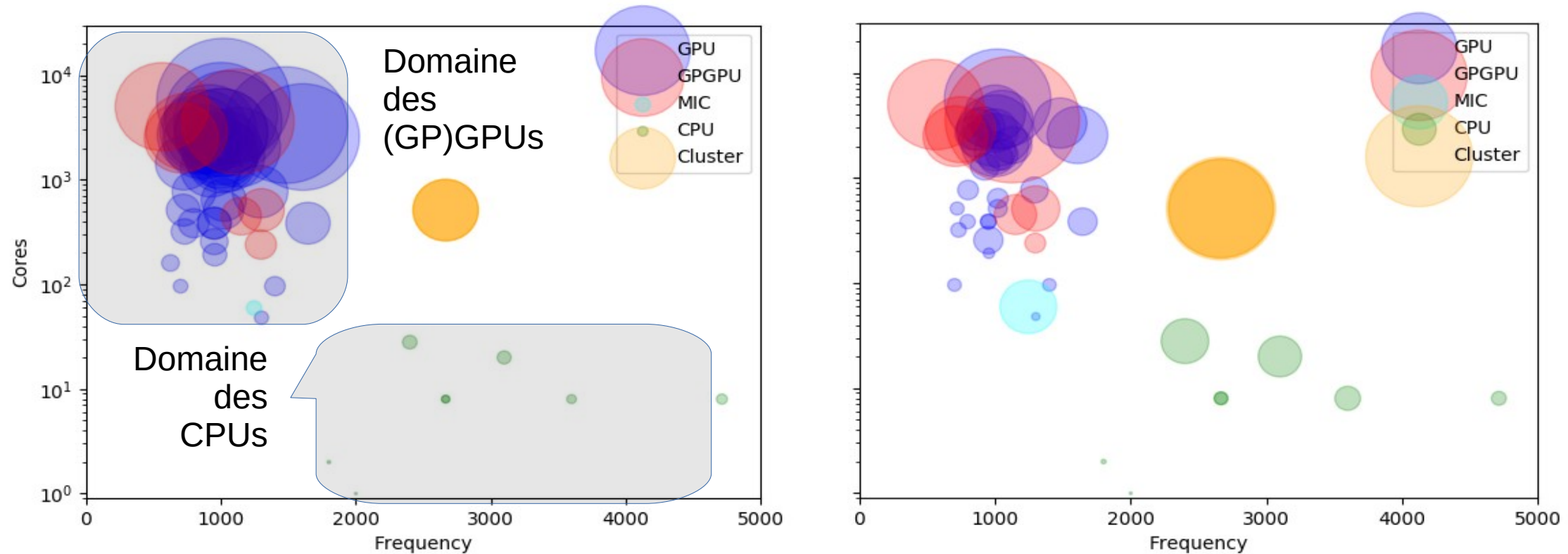
Performance en 32 bits  
"Pi Dart Dash"



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

# La performance en informatique

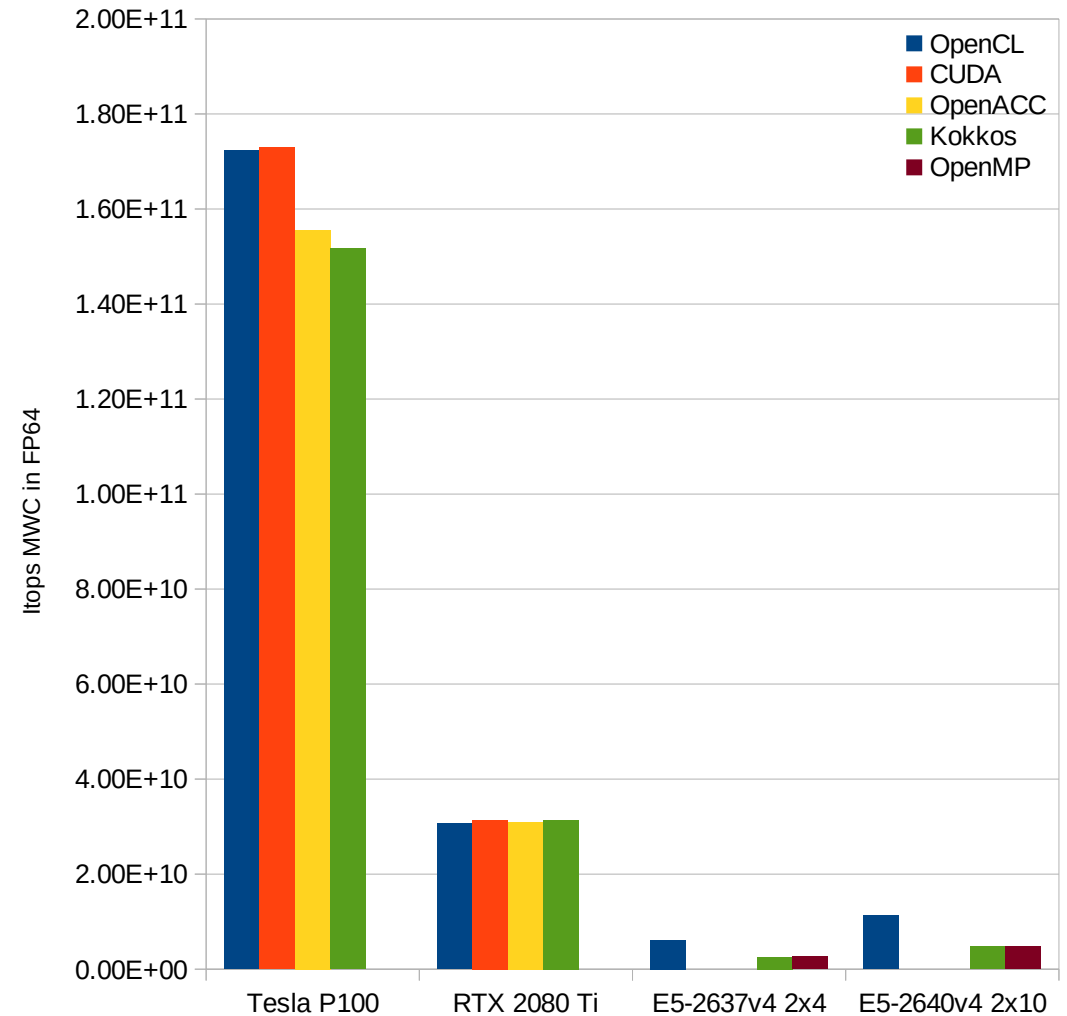
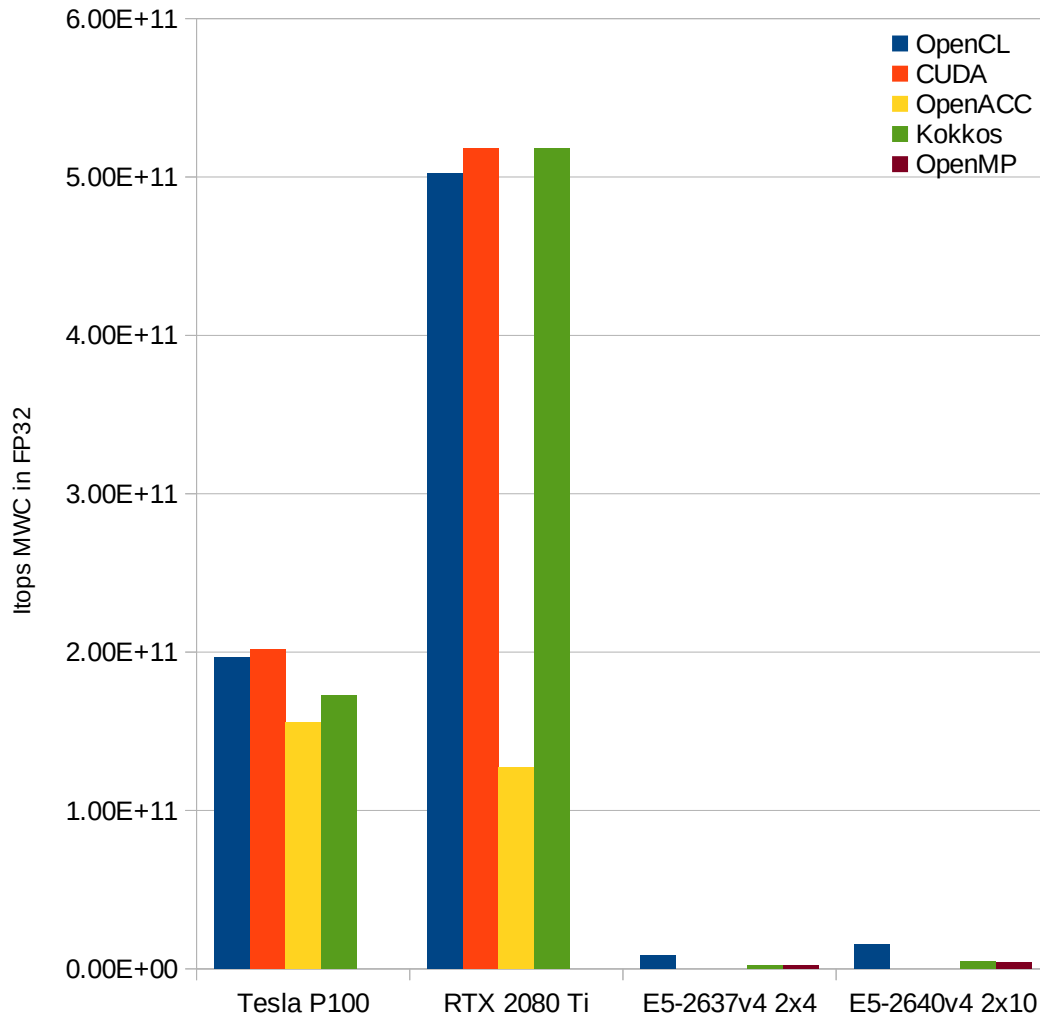
Question de battement, de cœurs, et de bits !



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



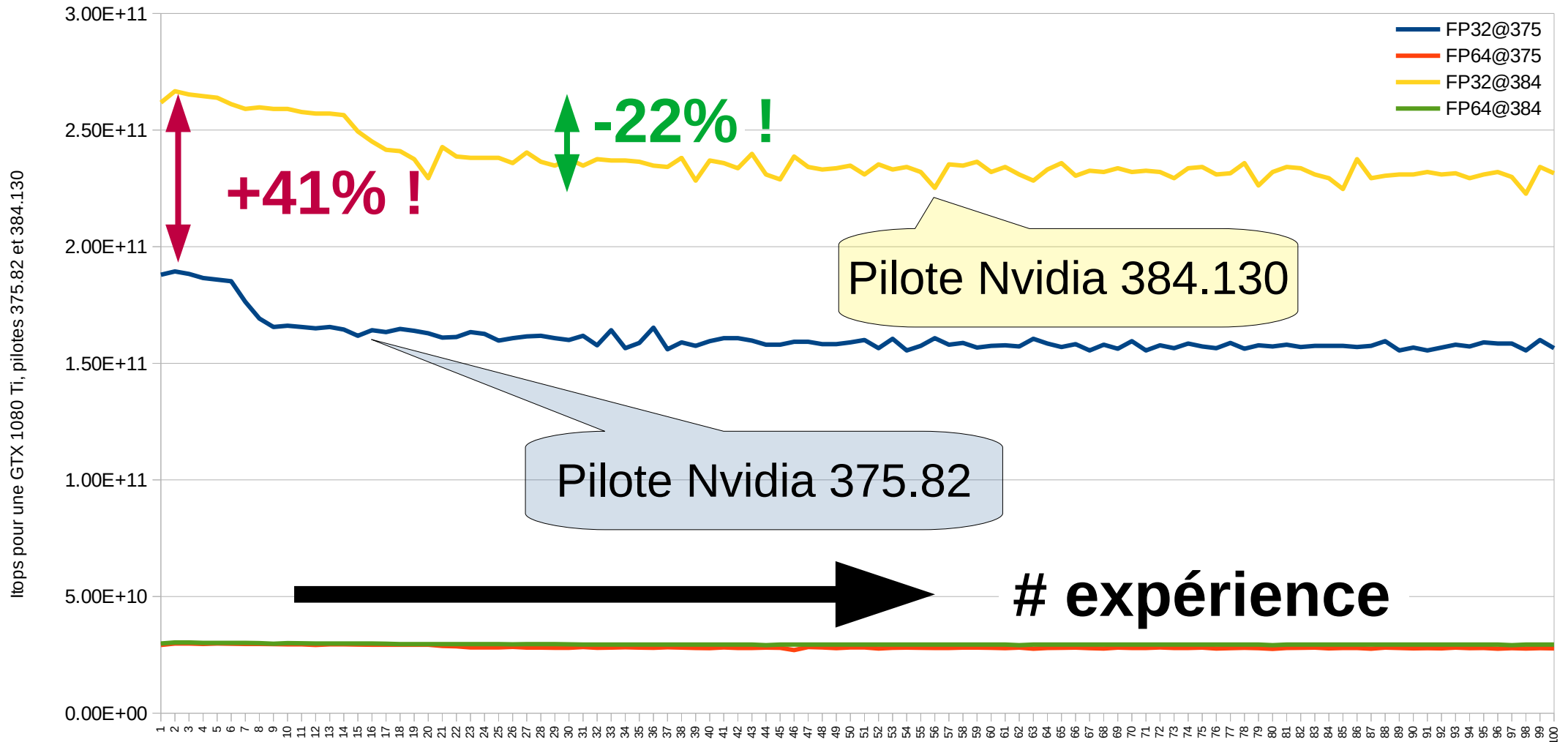
# Et pour les autres « approches » CUDA, OpenACC & Kokkos ?



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

# Versions de pilote & expériences

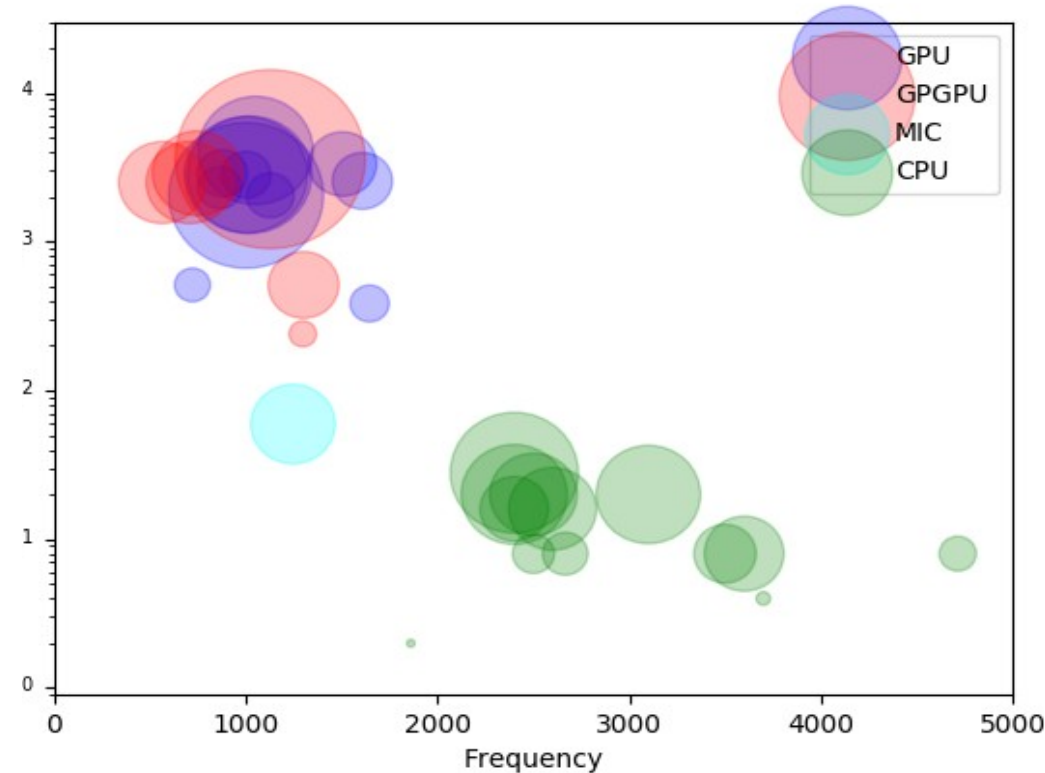
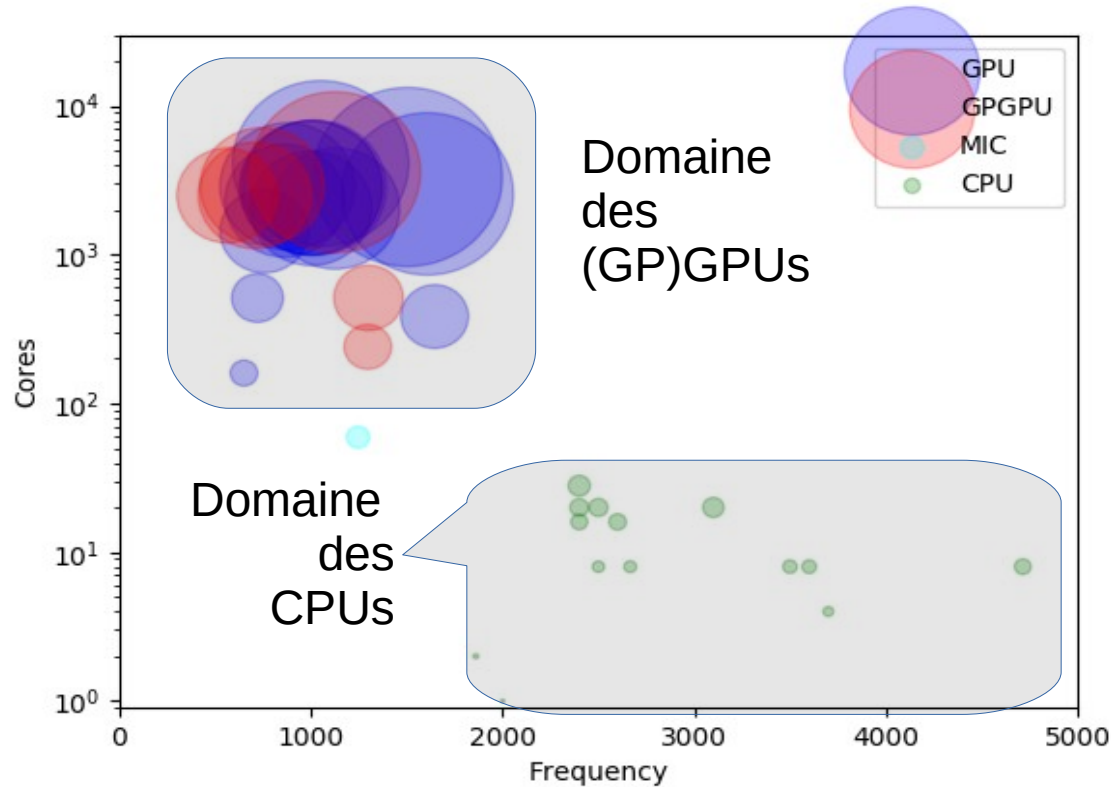
## Facteurs de variabilité...



A bien prendre en compte dans l'expérience !



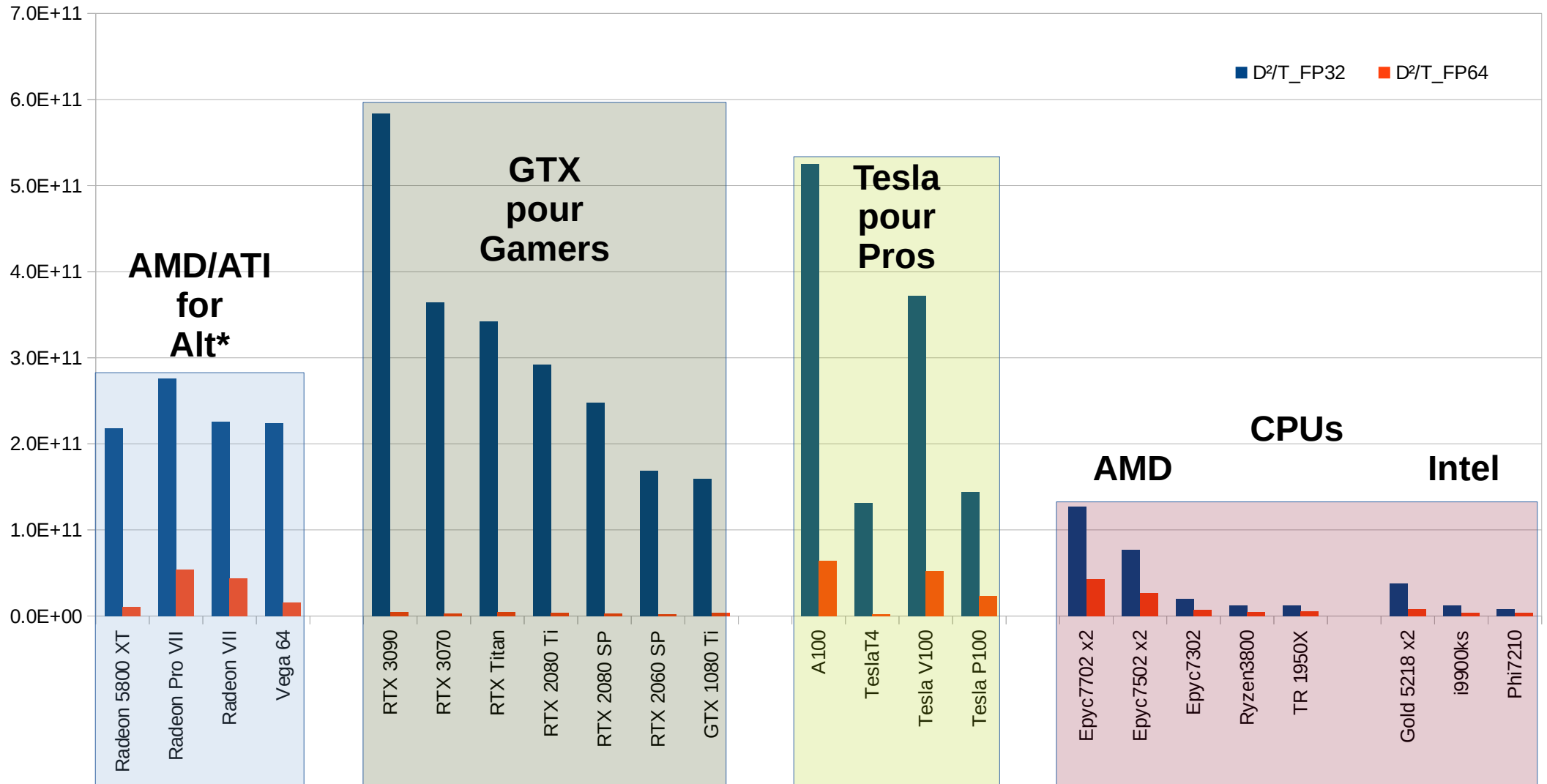
# La performance en informatique Pour un code plus physique...



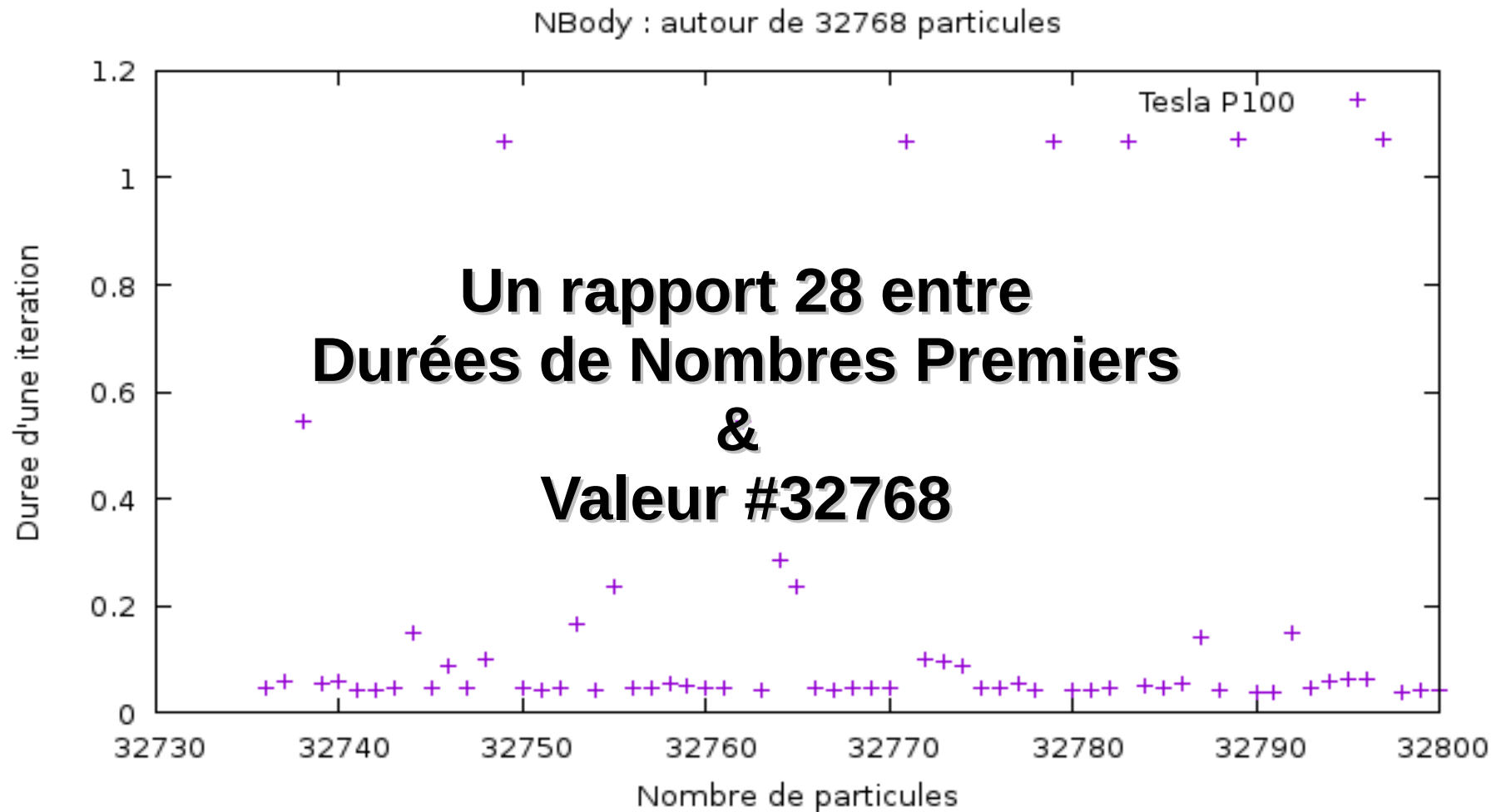
- Sur un GPGPU, performance stable
- Sur un GPU, baisse drastique de la performance (division par 20 en DP)
- Sur un CPU, performance relative meilleure

# Modèle N-corps « naïf »

## Une comparaison classique...



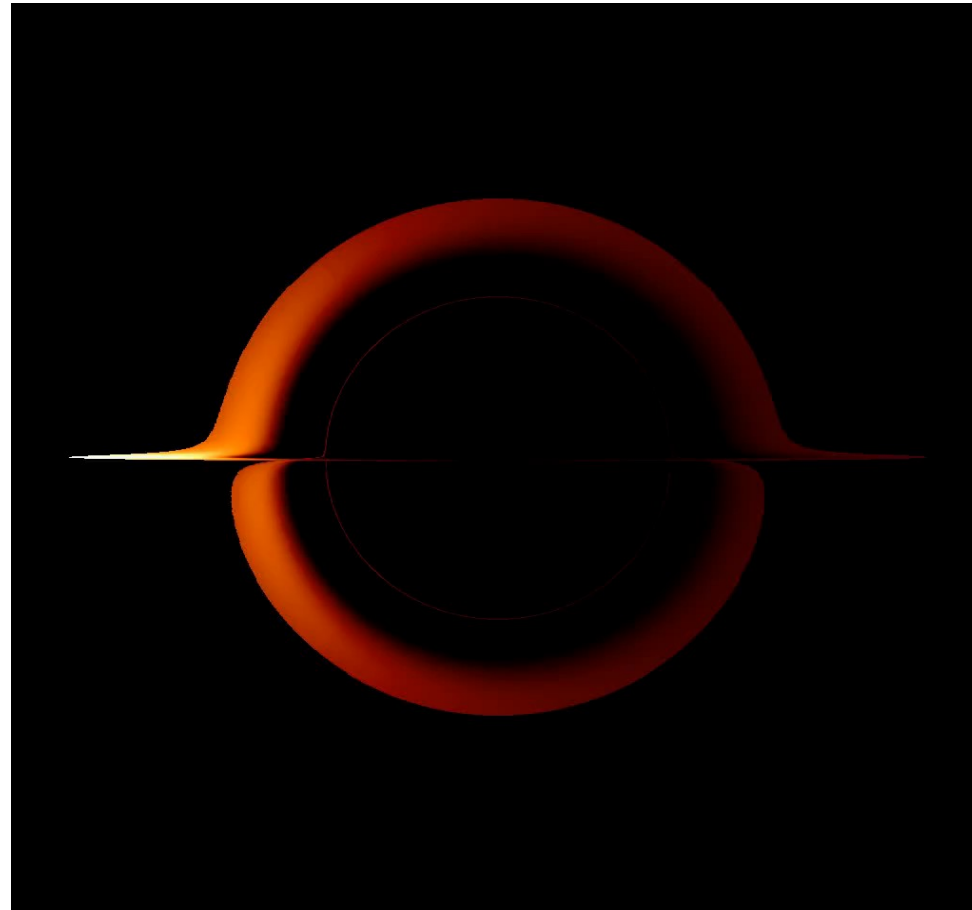
# Et les effets « prime numbers » pour les cartes Nvidia ?



# 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

# De la parallélisation à la gépufication 30 ans dévolution technologique...

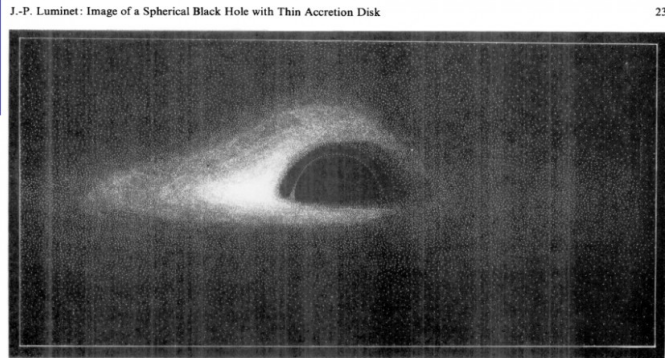




# 40 ans entre simulation et mesure

## De 2019 à 1979 : « *success story* » ?

1979 : JP Luminet A&A



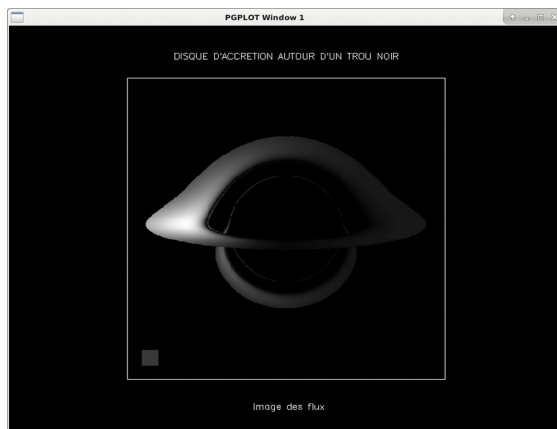
2014 : Film Interstellar



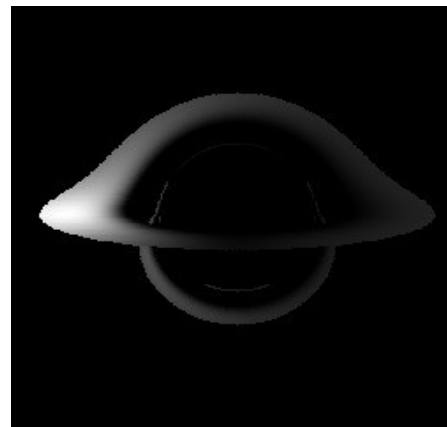
2019 : EHT, Messier 87



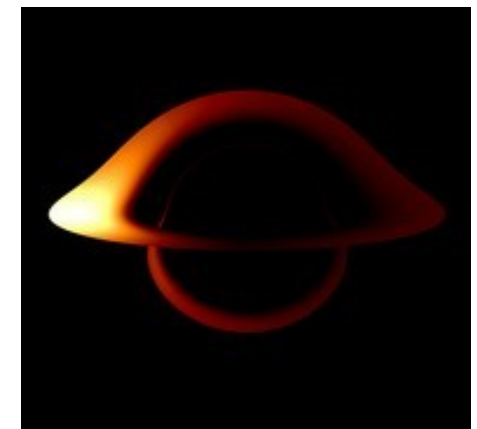
1994 : Code Fortran



1997 : Code C



2019 : OpenCL/CUDA



# La physique de base

## Tout dans un article de JP Luminet !

- Relativité générale d'Einstein
- Une métrique de Schwarzschild
- Réduction en équation polaire
- Dérivation de l'équation polaire
- Système du second ordre
- Modèle d'émission de disque
  - Raie monochromatique : cas d'école
  - Corps noir : modèle plus réaliste

Astron. Astrophys. 75, 228–235 (1979)

ASTRONOMY  
AND  
ASTROPHYSICS

### Image of a Spherical Black Hole with Thin Accretion Disk

J.-P. Luminet

Groupe d'Astrophysique Relativiste, Observatoire de Paris, Section d'Astrophysique, F-92190-Meudon, France

Received July 13, 1978

**Summary.** Black hole accretion disks are currently a topic of widespread interest in astrophysics and are supposed to play an important role in a number of high-energy situations. The present paper contains an investigation of the optical appearance of a spherical black hole surrounded by thin accretion disk. Isoradial curves corresponding to photons emitted at constant radius from the hole as seen by a distant observer in arbitrary direction have been plotted, as well as spectral shifts arising from gravitational and Doppler shifts. By the results of Page and Thorne (1974) the relative intrinsic intensity of radiation emitted by the disk at a given radius is a known function of the radius only, so that it is possible to calculate the exact distribution of observed bolometric flux. Direct and secondary images are plotted and the strong asymmetry in the flux distribution due to the rotation of the disk is exhibited. Finally a simulated photograph is constructed, valid for black holes of any mass accreting matter at any moderate rate.

**Key words:** black holes – accretion disks – geometrical optics

#### 1. Introduction

The aim of the present paper is to provide a reply to the question that many people ask themselves about the optical appearance of a black hole.

In order to be visible a black hole has of course to be illuminated, like any ordinary body. One of the simplest possibilities would be for the black hole to be illuminated by a distant localized source which in practise might be a companion star in a loosely bound binary system. A more interesting and observationally important possibility is that in which the light source is provided by an emitting accretion disk around the black hole, such as may occur in a tight binary system with overflow from the primary, and perhaps also on a much larger scale in a dense galactic nucleus. The general problem of the optical appearance of black holes is related to the analysis of trajectories in the gravitational field of black holes. For a spherical, static, electrical field-free black hole (whose external space-time geometry is described by the Schwarzschild metric) this problem is already well known (Hagihara, 1931; Darwin, 1959; for a summary, see Misner et al., 1973 [MTW]). In Sect. 2 we give only a brief outline of it with basic equations, trying to point out the major features which will appear later. All our calculations are done in the geometrical optics approximation (for a study of wave-aspects, see Sanchez, 1977). In Sect. 3 we calculate the apparent shape of circular rings orbiting a non-rotating black hole and the results are depicted in Figs. 5–6. In Sect. 4 we recall the standard analysis by Novikov and Thorne

(1973) of the problem of energy release by a thin accretion disk in a general astrophysical context, focusing attention more particularly on the analytic solution for the surface distribution of energy release that was derived by Page and Thorne (1974) in the limiting case of a sufficiently low accretion rate. In terms of this idealized (but in appropriate circumstances, realistic) model, we calculate the distribution of bolometric flux as seen by distant observers at various angles above the plane of the disk (Figs. 9–11).

#### 2. Image of a Bare Black Hole

Before analyzing the general problem of a spherical black hole surrounded by an emitting accretion disk, it is instructive to investigate a more simple case in which all the dynamics are already contained, namely the problem of the return of light from a bare black hole illuminated by a light beam projected by a distant source. It is conceptually interesting to calculate the precise apparent pattern of the reflected light, since some of the main characteristic features of the general geometrical optics problem are illustrated thereby.

The Schwarzschild metric for a static pure vacuum black hole may be written as:

$$ds^2 = -\left(1 - \frac{2M}{r}\right) dt^2 + \left(1 - \frac{2M}{r}\right)^{-1} dr^2 + r^2(d\theta^2 + \sin^2\theta d\phi^2) \quad (1)$$

where  $r$ ,  $\theta$ , and  $\phi$  are spherical coordinates and the unit system is chosen such that  $G=c=1$ .  $M$  is the relativistic mass of the hole (which has the dimensions of length). In this standard coordinate system the horizon forming the surface of the hole is located at the Schwarzschild radius  $r_s = 2M$ .

One can take advantage of the spherical symmetry to choose the "equatorial" plane  $\theta = \pi/2$  so as to contain any particular photon trajectory under consideration. The trajectories will then satisfy the differential equation:

$$\left\{ \frac{1}{r^2} \left( \frac{dr}{d\phi} \right)^2 + \frac{1}{r^2} \left( 1 - \frac{2M}{r} \right) \right\} = 1/b^2. \quad (2)$$

The second term in the left member can be interpreted as an effective potential  $V(r)$ , in analogy with the non-relativistic mechanics. The motion does not depend on the photon energy  $E$  and on its angular momentum  $L$  separately, but only on the ratio  $L/E = b$ , which is the impact parameter at infinity.

Let the observer be in a direction fixed by the polar angle  $\phi_0$  in the Schwarzschild metric, at a radius  $r_0 \gg M$ . The rays emitted by a distant source of light and deflected by the black hole intersect the observer's detector (for example a photographic plate) at a

# De l'article au rapport

- Métrique de Schwarzschild :

$$ds^2 = - \left(1 - \frac{2M}{r}\right) dt^2 + \left(1 - \frac{2M}{r}\right)^{-1} dr^2 + r^2(d\theta^2 + \sin^2\theta d\phi^2)$$

- Equation polaire :

$$\left(\frac{1}{r^2} \left(\frac{dr}{d\phi}\right)\right)^2 + \frac{1}{r^2} \left(1 - \frac{2M}{r}\right) = \left(\frac{\pi_t}{\pi_\phi}\right)^2 = \frac{1}{b^2}$$

- Changement de coordonnées :  $u=1/r$

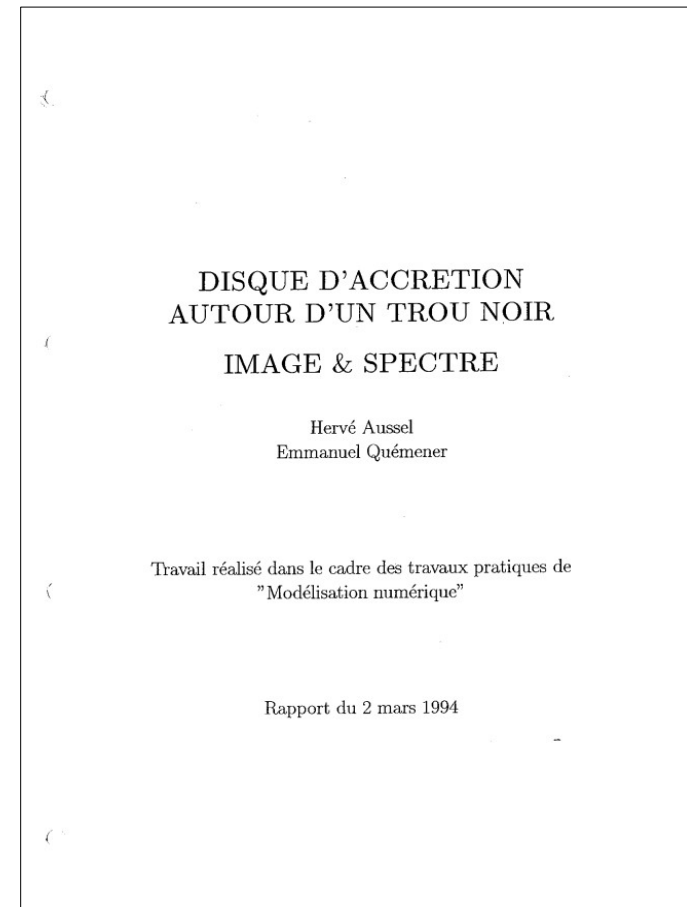
$$\left(\frac{du}{d\phi}\right)^2 + u^2 \left(1 - \frac{2Mu}{b}\right) = 1$$

- Dérivation de l'équation polaire :

$$\frac{d^2u}{d\phi^2} + u \left(1 - \frac{3Mu}{b}\right) = 0$$

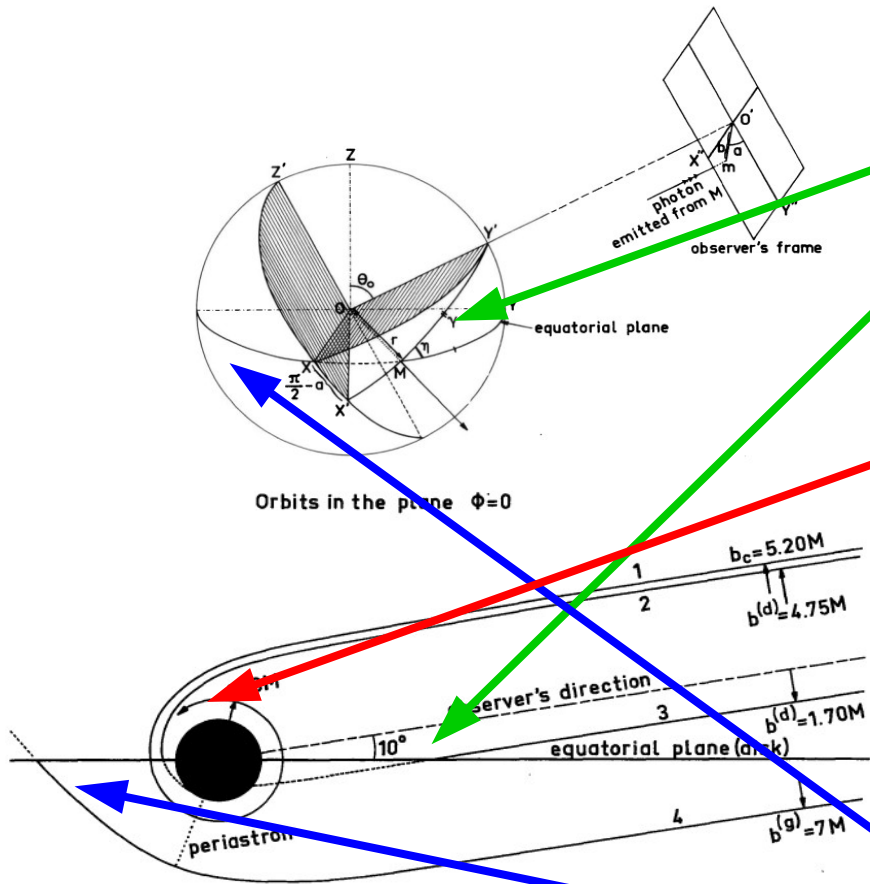
- Système d'équations à résoudre :

$$v = \frac{du}{d\phi} \quad \text{et} \quad \frac{dv}{d\phi} = 3\frac{m}{b}u^2 - u$$

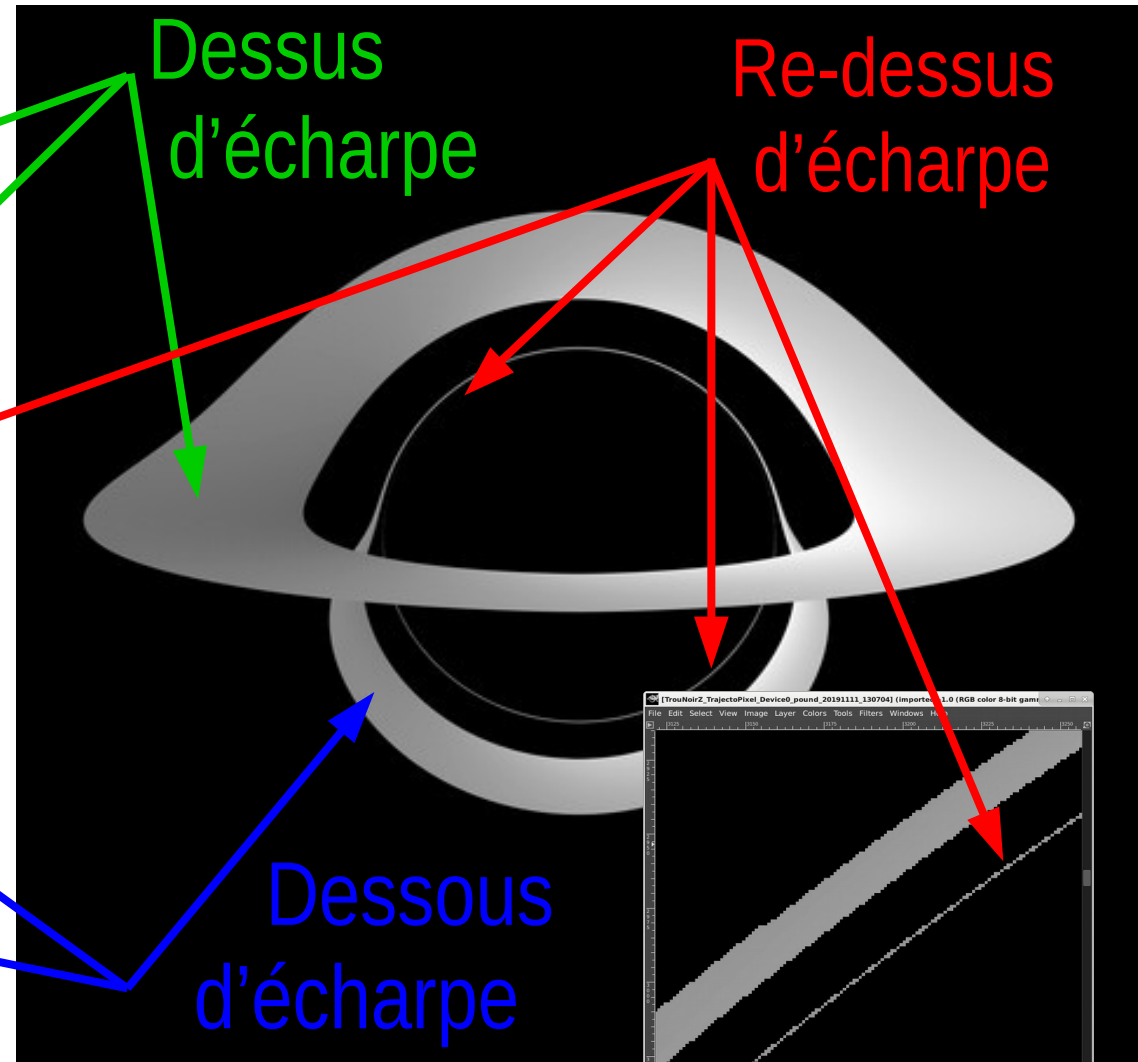


# Echarpe de plasma autour du Trou Noir

## Pas « sans » mais « avec » dessus-dessous...



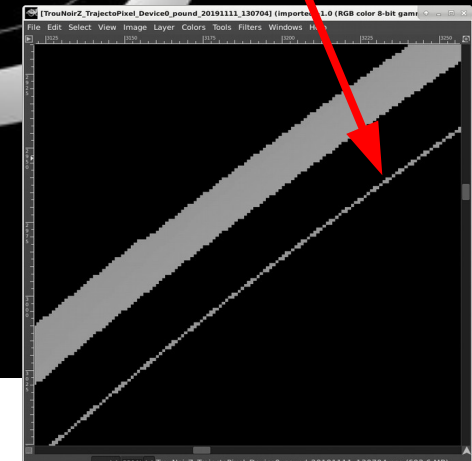
**Fig. 4.** Illustrative orbits in the plane  $\{\phi=0\}$ . Trajectory 1 has the critical impact parameter and circles infinitely around the black hole; trajectories 2 and 3 give direct images, trajectory 4 gives a secondary image



Dessus  
d'écharpe

Re-dessus  
d'écharpe

Dessous  
d'écharpe



# La méthode : « lancer des rayons » de l'oeil au disque de plasma

- Pour chaque pixel de l'image
  - Calculer la trajectoire (résoudre le système d'équations)
  - Regarder si le photon intercepte le disque
    - Si la distance du photon inférieure au rayon de Schwarzschild
      - Damage... (en même temps, c'est le principe du « trou noir »)
    - Si le photon traverse le plan du disque entre ses rayons intérieur & extérieur
      - Estimation de l'effet Doppler & Einstein
      - Estimation du flux par deux méthodes :
        - Émission monochromatique : simple mais instructive
        - Émission de « corps noir » : plus réaliste mais spectre de Planck
- Méthode systématique mais très coûteuse :
  - Aucune exploitation de la symétrie du problème physique

# Méthode « économique » : exploitation symétrie cylindrique

- Pour chaque « paramètre d'impact » (distance au centre)
  - Calcul de la trajectoire du photon en fonction de l'angle
  - Pour chacun des pixels de l'image avec ce paramètre d'impact :
    - Estimation de l'indice d'interception correspondant à l'angle du disque
    - Test si la distance au centre pour cet indice est entre les rayons interne et externe
      - Estimation de l'effet Doppler & Einstein
      - Estimation du flux par deux méthodes :
        - Émission monochromatique : simple mais instructive
        - Émission de « corps noir » : plus réaliste mais spectre de Planck
- Beaucoup plus efficace et temps de calcul  $\sim$  #pixels
  - Exploitation du PixHertz (nombre de pixels sur temps écoulé)...

# Computhèque

## 40 ans d'histoire de l'informatique

- Ordinateurs :
  - Thomson TO8, Amiga 500, Mac, ...
- Processeurs (et leur cartes mère) :
  - 80386SX, 80486SX, Overdrive DX4, Amd5x86, K6-2, K7, ...
- Périphériques de stockage :
  - Cartes, câbles, disques, lecteurs de bande, ... en SCSI, Firewire, FC
- Périphériques de communication :
  - Ethernet 10Base2, 10Base-T, ATM, Myrinet, Infiniband, ...
- OS : natifs ou Debian Buzz, Hamm, Squeeze, ...

# Le banc d'essai : échantillon

## 30 ans d'évolutions technologiques

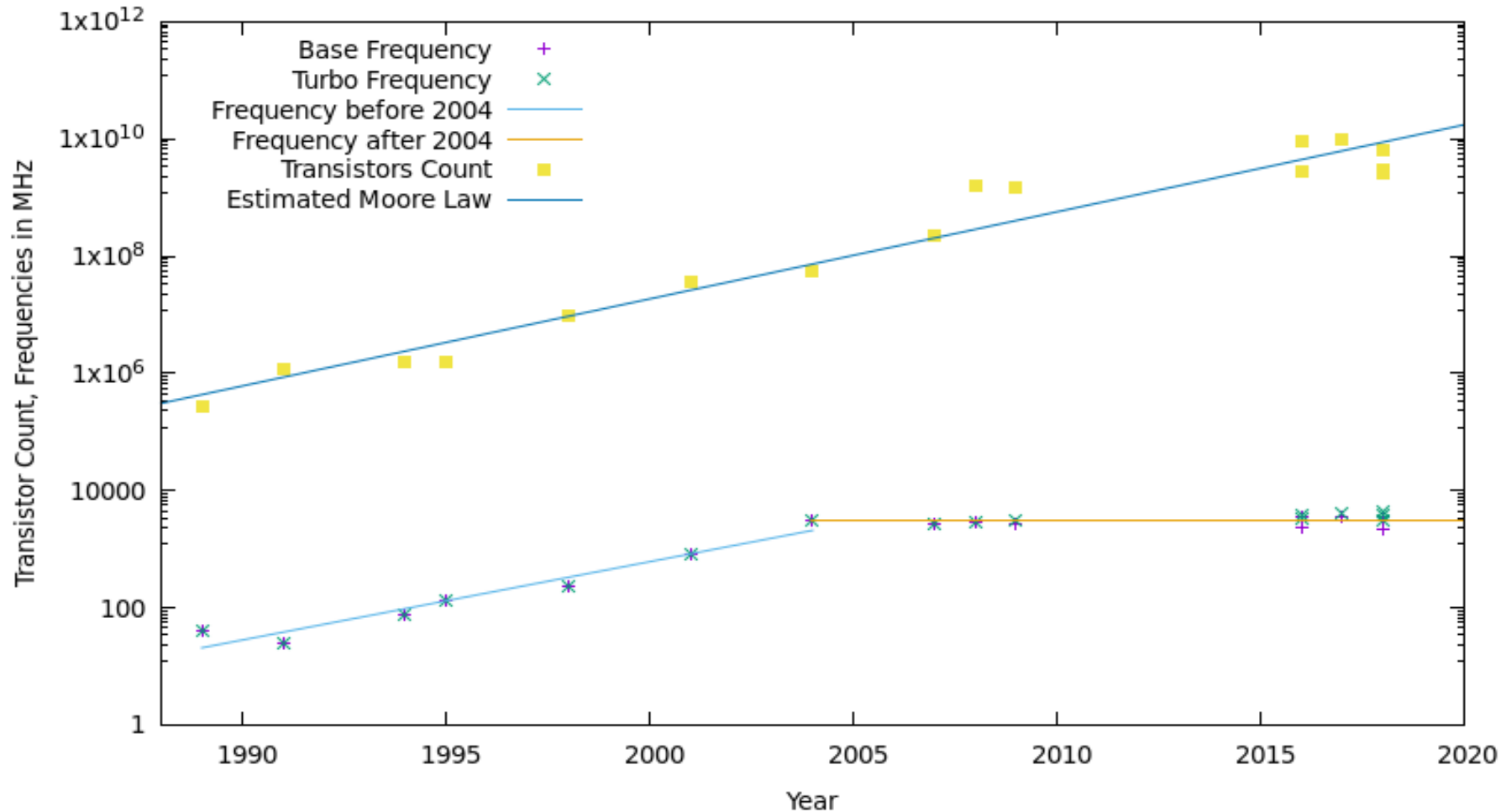
- La fréquence : de 40 MHz à 3.6 GHz
- Ces 30 dernières années : 5 (r)évolutions
  - Intégration systématique du FPU dans les processeurs : **#1**
    - Avant : 80386SX à 40 MHz (1989) et 80486SX à 25 MHz (1991)
    - Après : Overdrive DX4 à 75 MHz (1994) et Amd5x86 (1995)
  - Exploitation interne de RISC86 et intégration unité vectorielle : **#2 et #3**
    - AMD K6-2 à 233 MHz avec unité 3DNow
  - Multiplication des coeurs (d'abord virtuel) : **#4**
    - Premiers : Pentium 4 Northwood à 3 GHz avec HyperThreading et AthlonX2 à 2.6 GHz
    - Systèmes avec HarperTown, Nehalem, Broadwell, Skylake, Threadripper : 8 à 28 coeurs
  - Détournement de l'usage des GPU : **#5**
    - De la Tesla C0160 à la Tesla V100



# Distribution de CPU pertinente ?

## Transistors & Fréquences...

CPU Systems : Number of Transistors, Base Frequency, Turbo Frequency



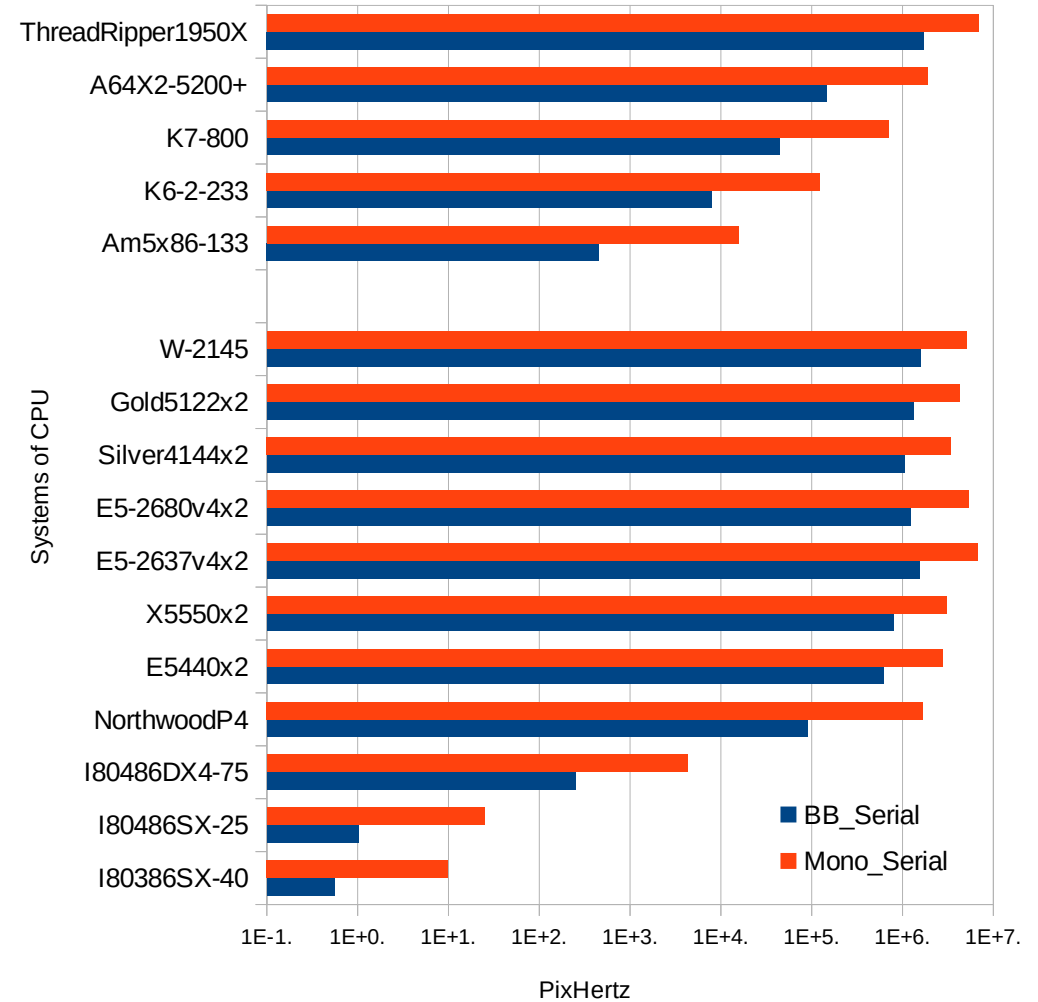
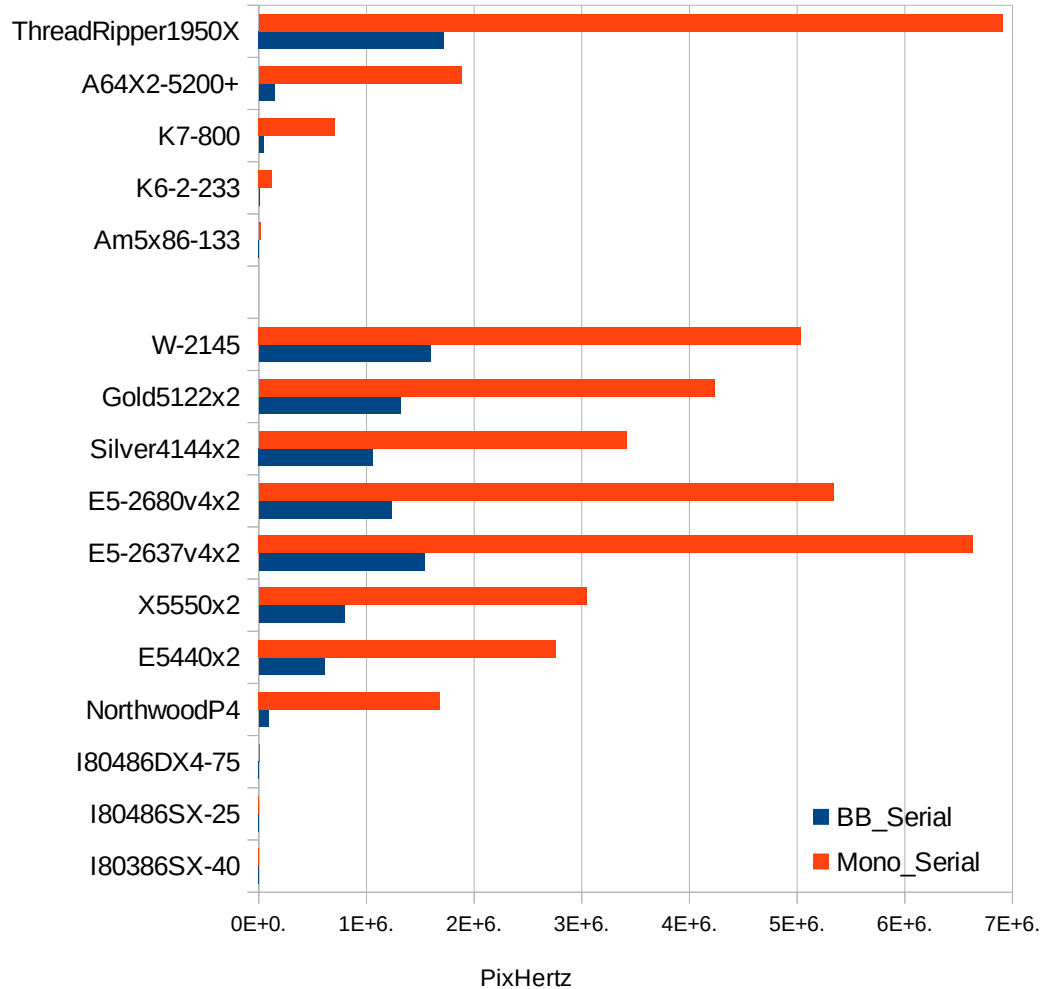
Doublement des transistors tous les 2 ans...

# Banc de test sur les 16 CPUs

- Les processeurs et leurs distributions :
  - 80386SX, 80486SX, Overdrive DX4, Amd5x86 : Debian Buzz & Hamm
  - K7, Northwood, AthlonX2 : Debian Stretch
  - E5440x2, X5550x2, E5-2637v4x2, E5-2680v4, Gold5122, Silver4144, W-2145 : Debian Stretch
  - Threadripper 1950X : Ubuntu 18.10
- Images de 64x64 à 16384x16384 pixels :  $2^6$  à 2
  - Sauf pour les très très vieux CPU : limitation à 256x256
- Méthodes : 2 à explorer avec « charges » différentes
  - Charge calculatoire faible : « Monochromatique » (ak Mono)
  - Charge calculatoire élevée : « Corps Noir » (aka BB)
- Statistiques : 10 lancements successifs
  - Exploitation de la médiane pour le « Elapsed Time »

# Exécution du code séquentiel

## On gagne (quand même) en 30 ans

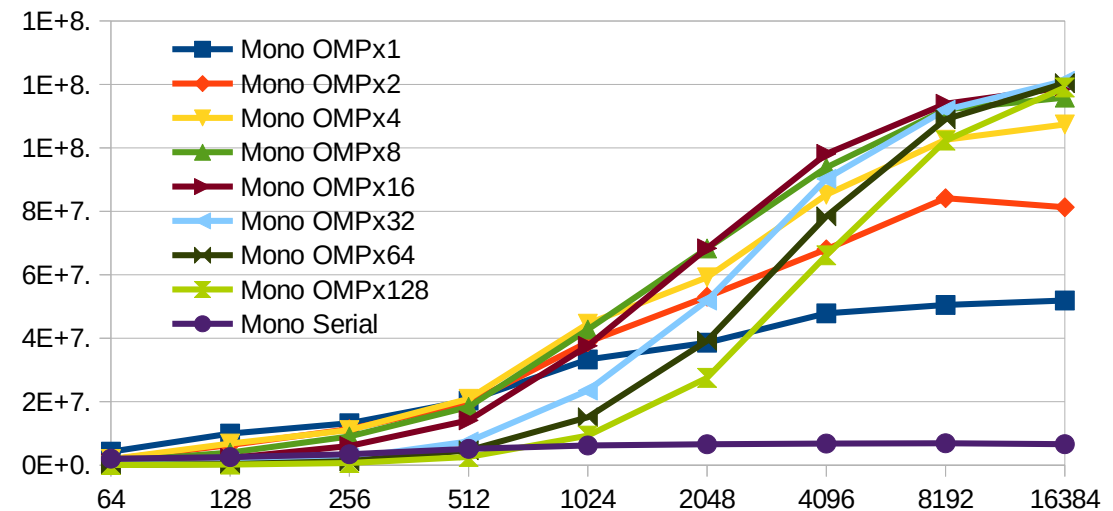
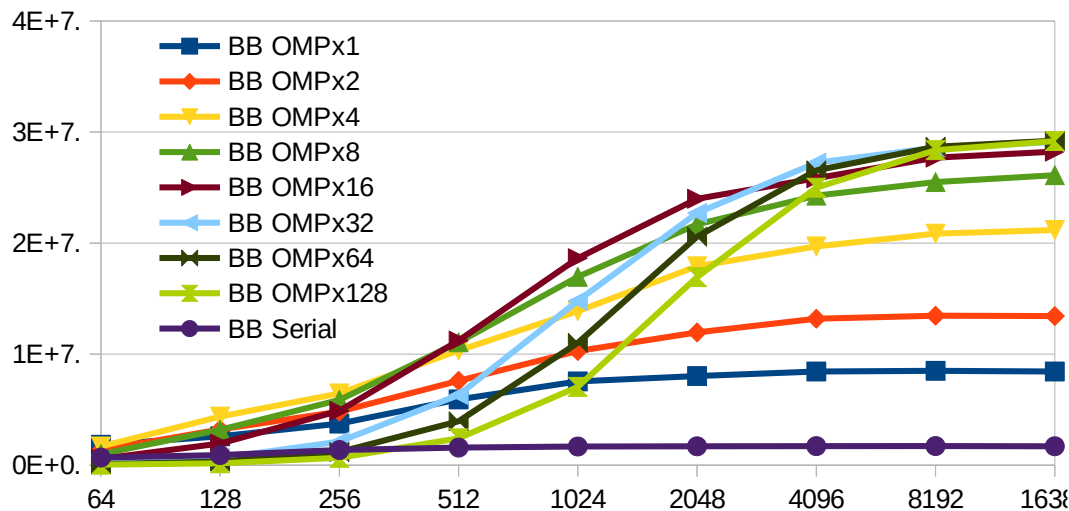


Gain Best/Worse : 3 millions en BB et 700000 en Mono...

# Parallélisation du code

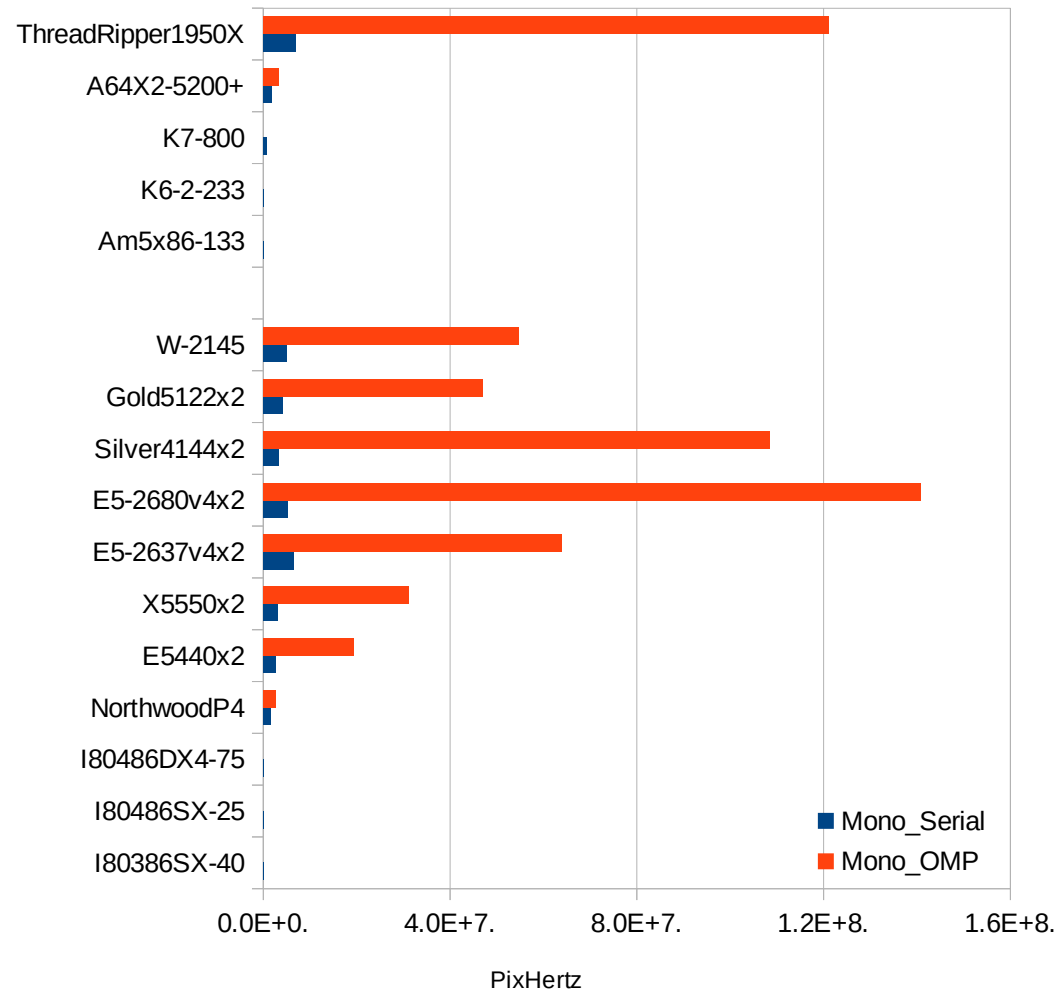
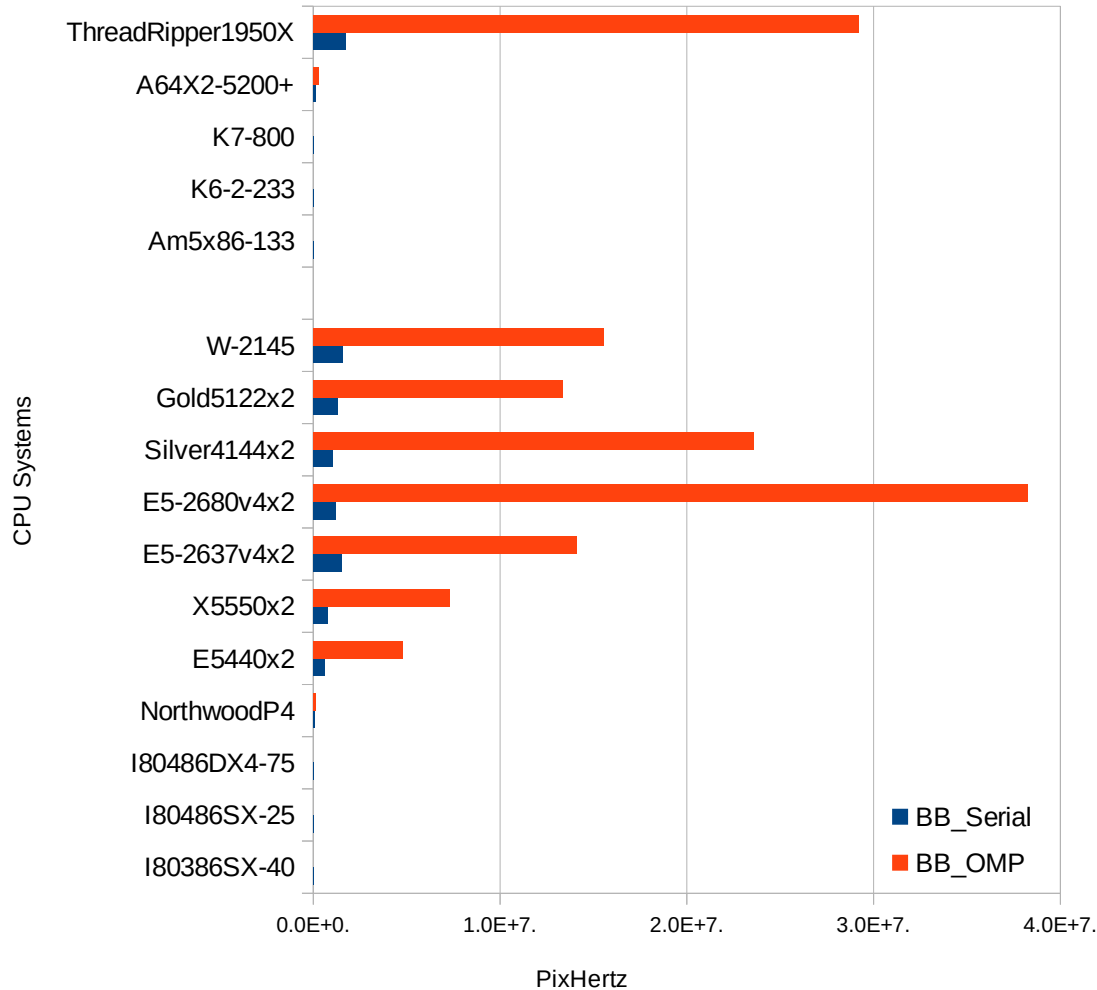
## Passage en OpenMP & étrangement

- Parallélisation « naturelle » : paramètre d'impact
  - Modification mineure du code (déplacement de déclaration de variables)
  - Crainte : charge calculatoire non équivalente pour les différents threads
  - Exploration pour différents OMP\_NUM\_THREADS : de 1x à 128x
- Pour le meilleur : le ThreadRipper 1950x, un x17-x18



# Parallélisation OpenMP

## On gagne plus que prévu en BB !



Entre x30 : x68 millions en BB et x14 millions en Mono

# Peut-on mieux faire ?

## Testons OpenCL !

- OpenCL, méconnu mais tellement polyvalent : 13 implémentations
  - GPU : Nvidia, AMD via ROCm, AMD via Mesa, Intel via Beignet et Intel
  - CPU : AMD, PortableCL, Intel
  - MIC : Intel pour Xeon Phi
  - FPGA : Altera/Intel, Xilinx
  - (DSP : Texas Instruments)
  - (GPU ARM)
- OpenCL : sa programmation...
  - Principe : des « noyaux » de calcul à distribuer à outrance !
  - Programmation « hardcore » en C, plus facile en C++
  - Programmation via API Python : « la voie » !

# OpenCL : distribuer notre calcul.

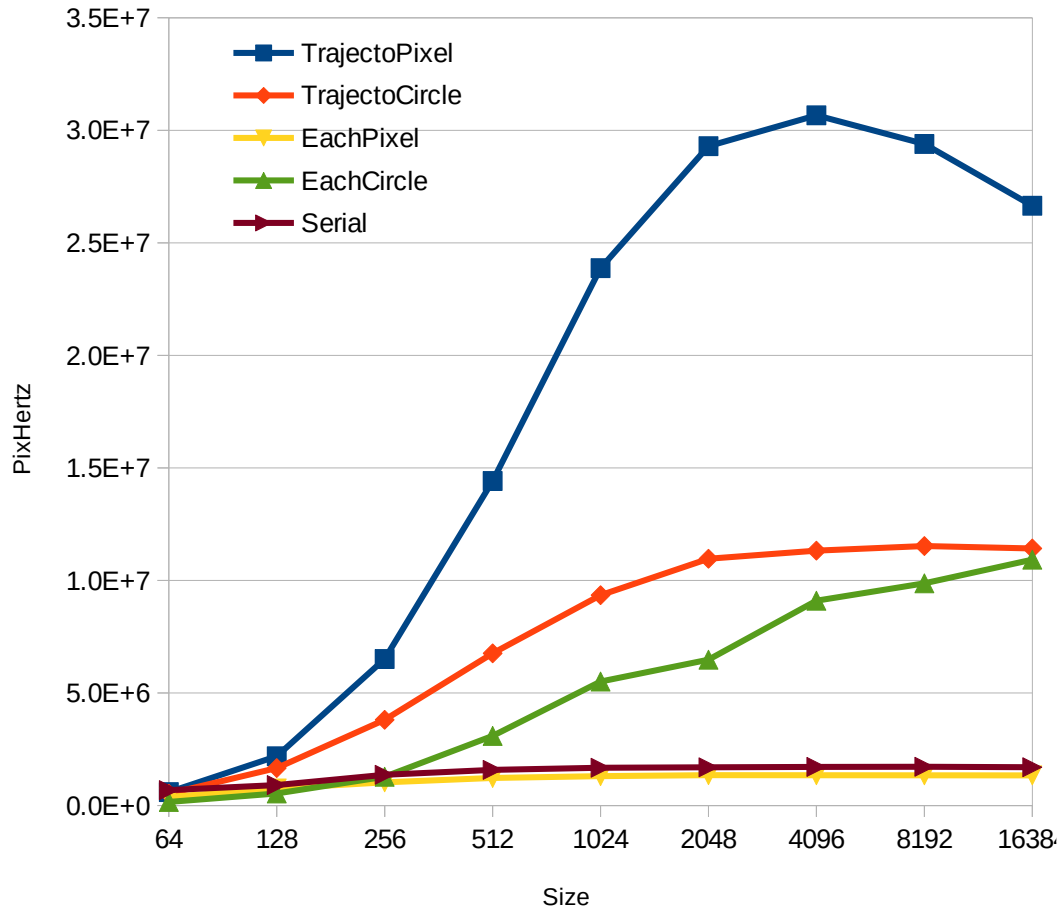
## Quel régime de parallélisme PR ?

- Approche initiale héritée du code C : **EachCircle**
  - Parallélisé suivant le paramètre d'impact :  $PR=Taille/2$
- Approche brutale : **EachPixel**
  - Parallélisé suivant le nombre de pixels :  $PR=Taille*Taille$
- Approche hybride : **TrajectoPixel**
  - D'abord parallélisé suivant les paramètres d'impact :  $PR=Taille/2$
  - Ensuite parallélisé suivant chaque pixel :  $PR=Taille*Taille$
- Approche hybride sauvage : **TrajectoCircle**
  - D'abord parallélisé suivant les paramètres d'impact :  $PR=Taille/2$
  - Ensuite parallélisé suivant chaque pixel :  $PR=4*Taille$
- Donc 4 méthodes à explorer pour tous nos CPU !

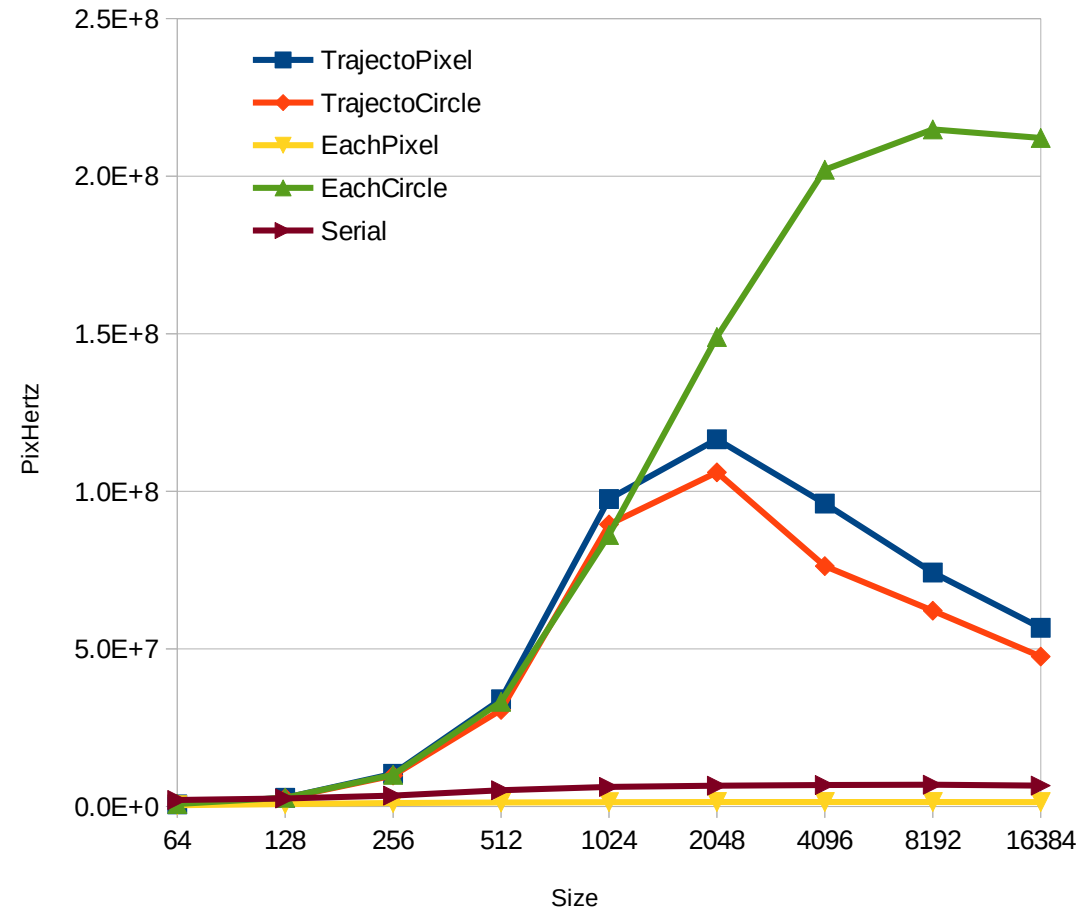
# OpenCL sur Threadripper 1950x

## La méthode de // importante...

BB on Threadripper 1950X with OpenCL



Mono on Threadripper 1950X



TrajectoPixel pour BB, EachCircle pour Mono...

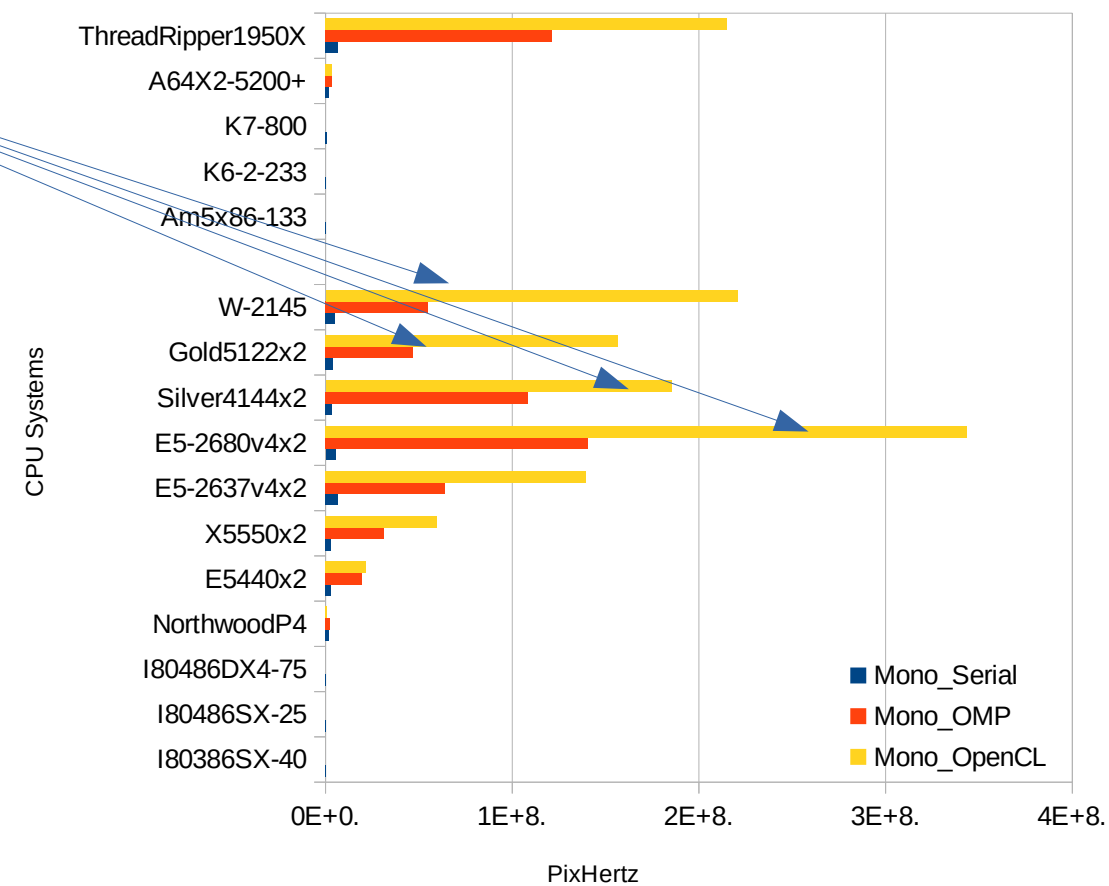
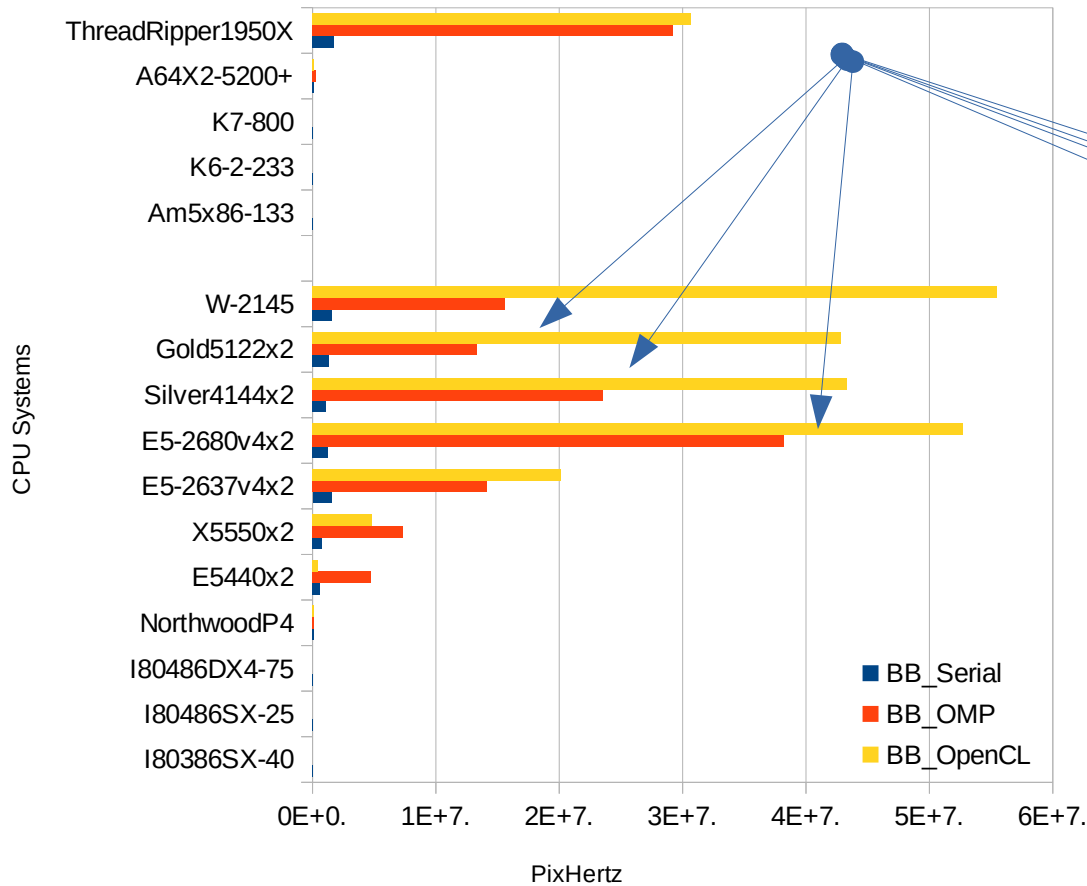


# Parallélisation OpenCL

## Pour tous les processeurs...

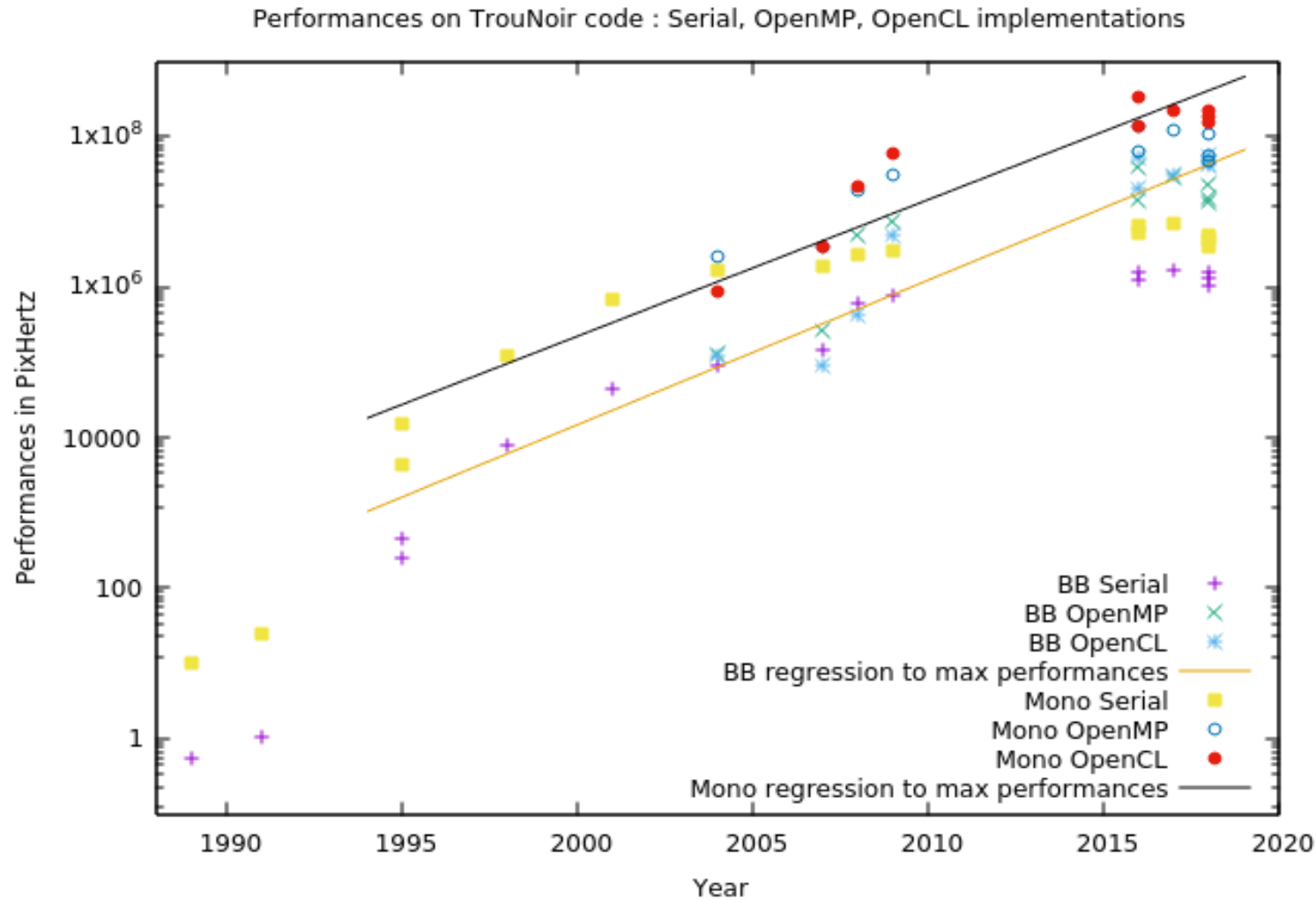
BB with Serial, OpenMP, OpenCL

Mono with Serial, OpenMP, OpenCL



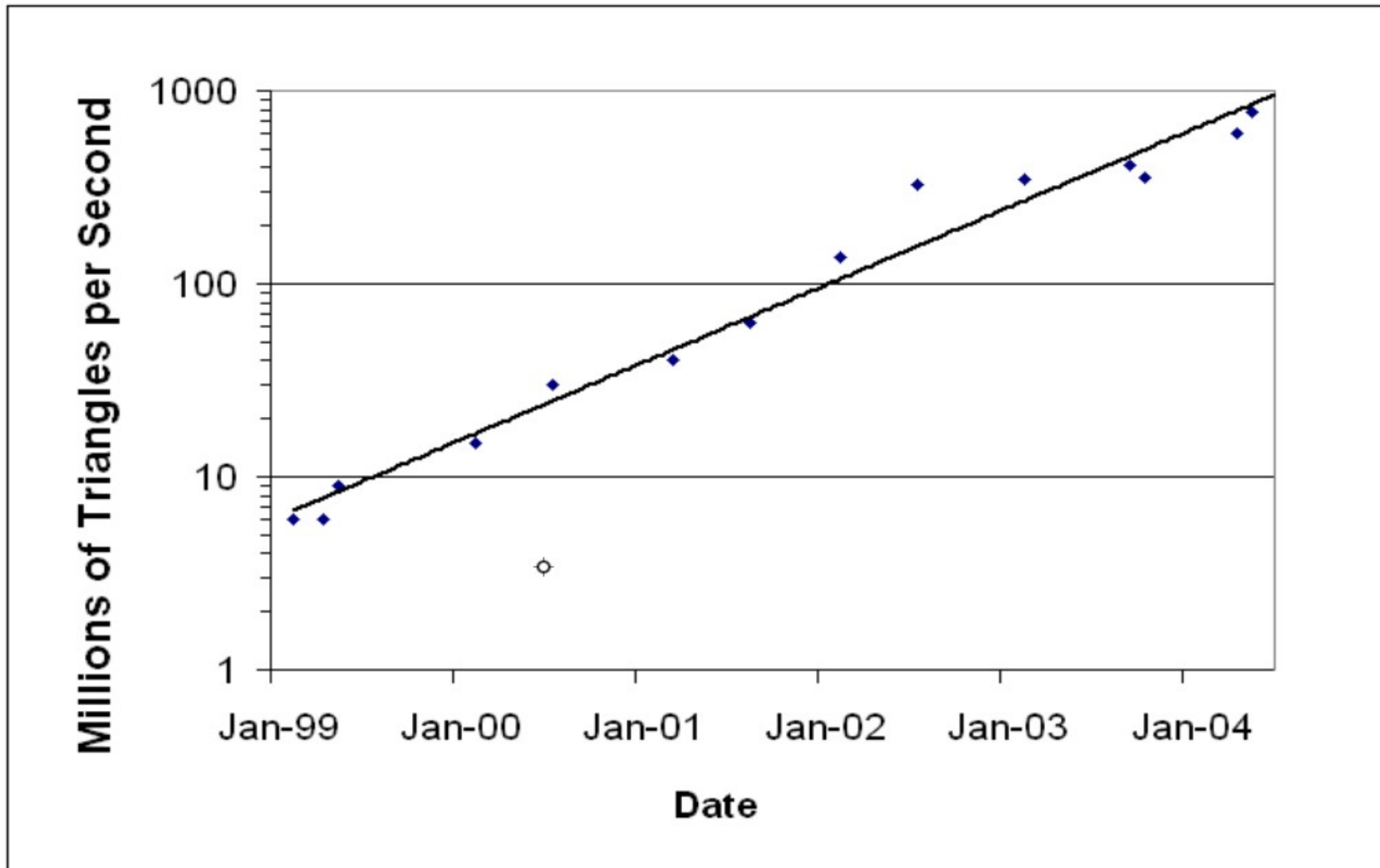
OpenCL Intel très efficace avec CPU Intel !

# Pour les processeurs Loi de Moore\* respectée ! Enfin...



Performance : x2 tous les 18 mois !

# 1999-2004 : progression CPU bof... Mais les GPU...



x100 pour les GPUs et seulement x10 pour les CPUs

# Ajoutons les GPUs & MIC

Génération Fermi, Kepler, Maxwell, Pascal, Turing

Gamer



GPGPU



Génération GT200, Fermi, Kepler, Pascal, Volta

MIC

Intel : KNC



R9-Nano, Vega 64, Radeon 7

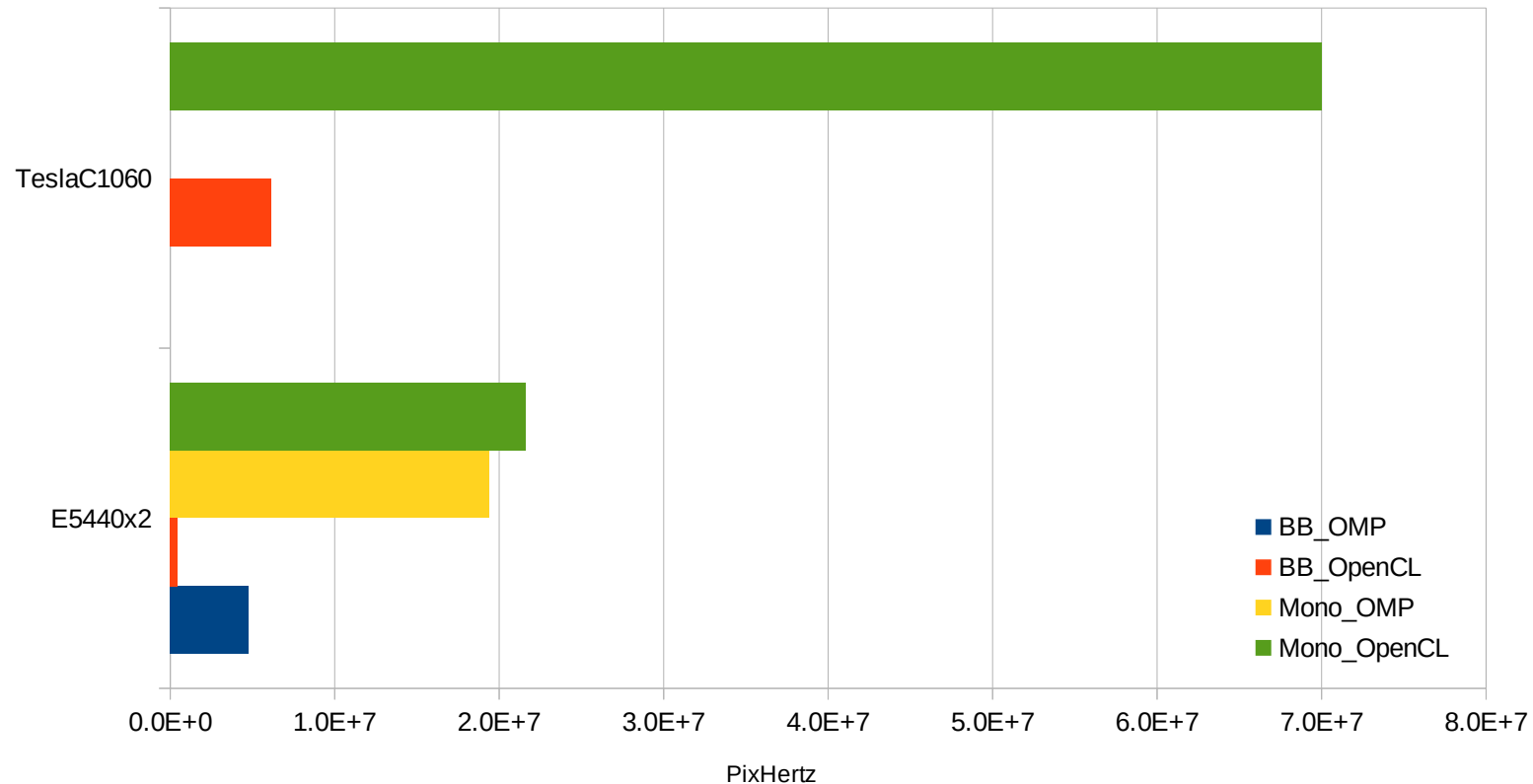


AMD GPU

Le meilleur en computèque de chaque génération...

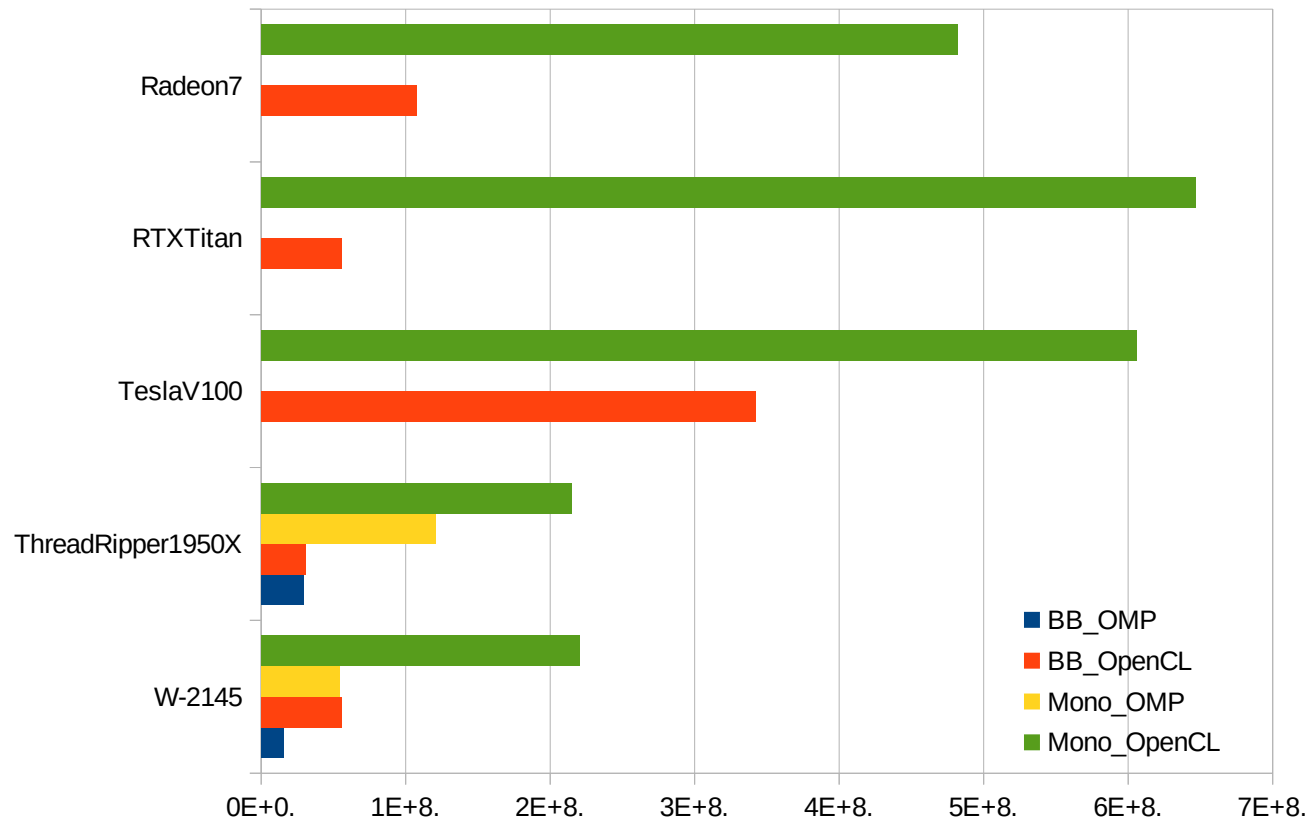
# Comparaison au fil du temps

## En 2008, E5440 & C1060



- En Mono, Un x3 en OpenCL pour le GPGPU
- En BB, à peine supérieur en OpenCL face à l'OMP

# Sortie en 2017, la Tesla V100 Seulement fin 2019 au CBP



- Tesla V100 ~ RTX Titan en Mono, x6 en BB
- Tesla V100 ~ Radeon 7 en Mono, x3 en BB
- CPU à x3 en Mono, mais x10 en BB

# Et CUDA dans tout ça ?

## d'OpenCL en CUDA avec Python

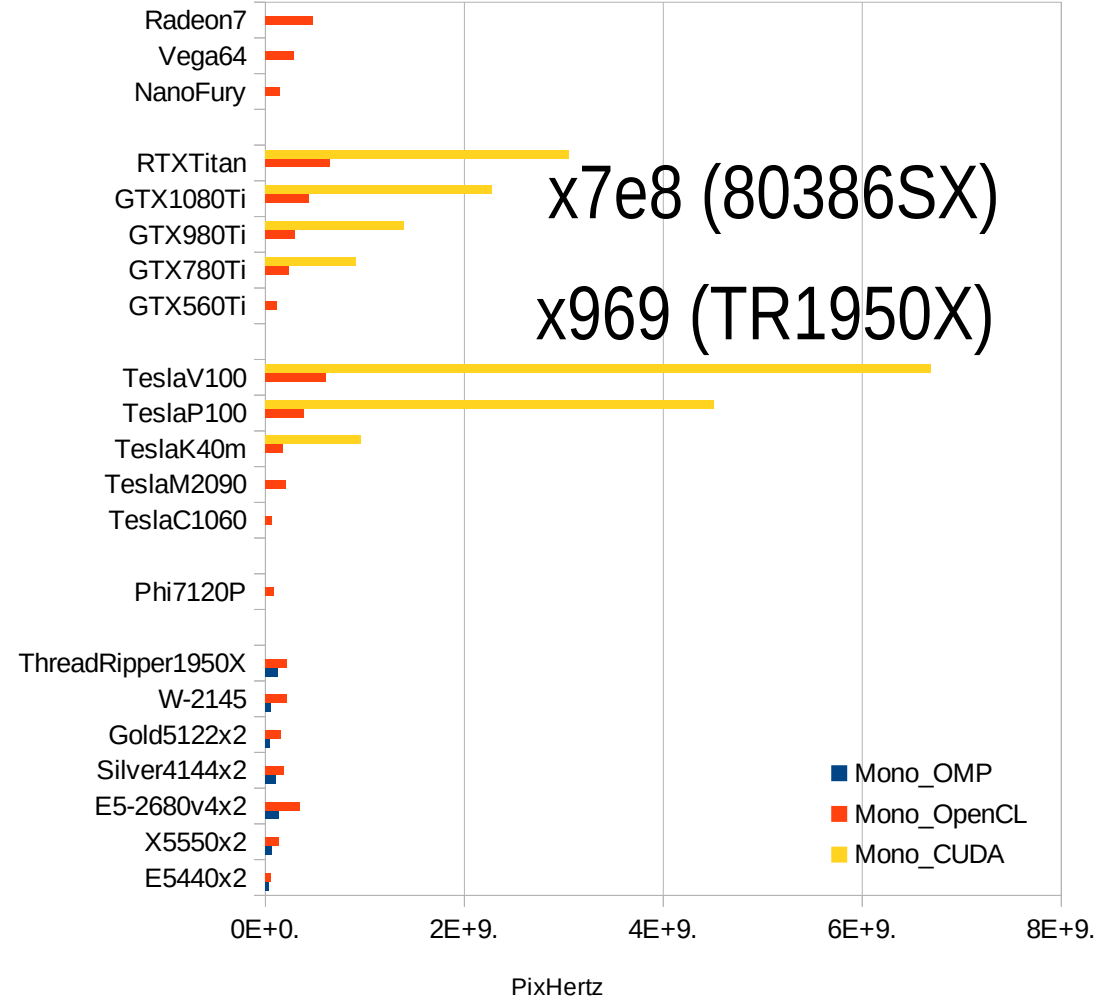
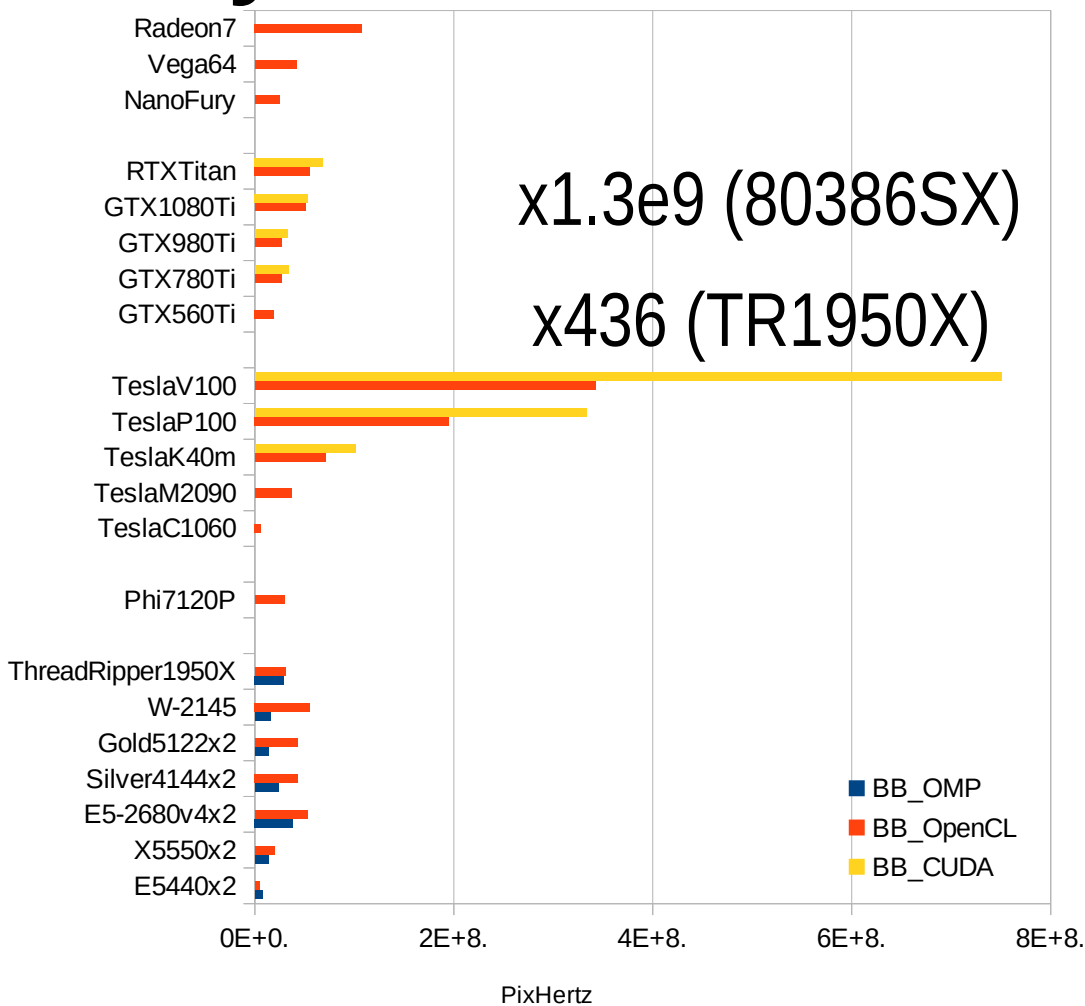
- Approche « naïve » :
  - Copier/Coller de tous les noyaux OpenCL
  - Remplacement des « workitem » par les « BlockIdx »
  - Préfixe des fonctions « internes » des noyaux par « `__device__` »
- Petit test en BB avec temps de calcul (plutôt étonnant) :
  - En OpenCL : 0.713273 s
  - En CUDA : 10.573987 s (**soit 13 fois plus lent !!!**)
- Petit test en Mono :
  - En OpenCL : 0.611651 s
  - En CUDA : 0.500251 s (**un peu meilleur mais pas terrible**)
- Bref, ça rappelle un souci déjà rencontré...

# En CUDA, l'attaque des threads...

- Exploitation de 2 étages de parallélisme
  - Définition de « grid » et de « threads »
  - Découpage dans les noyaux des tâches mode « pagination »
    - En OpenCL : `uint xi=(uint)get_global_id(0);`
    - En CUDA : `uint xi=(uint)(blockIdx.x*blockDim.x+threadIdx.x);`
- Nouveau test en BB, Mode TrajectoPixel :
  - Threads à 1 : 10.572677 s
  - Threads à 32 et  $32^2$  : 0.352543 s (soit x30)
- Nouveau test en Mono, Mode TrajectoPixel :
  - Threads à 1 : 0.504024 s
  - Threads à 32 et  $32^2$  : 0.039154 (soit x13)
- En passant en CUDA ; x2 en BB mais x15 en Mono !



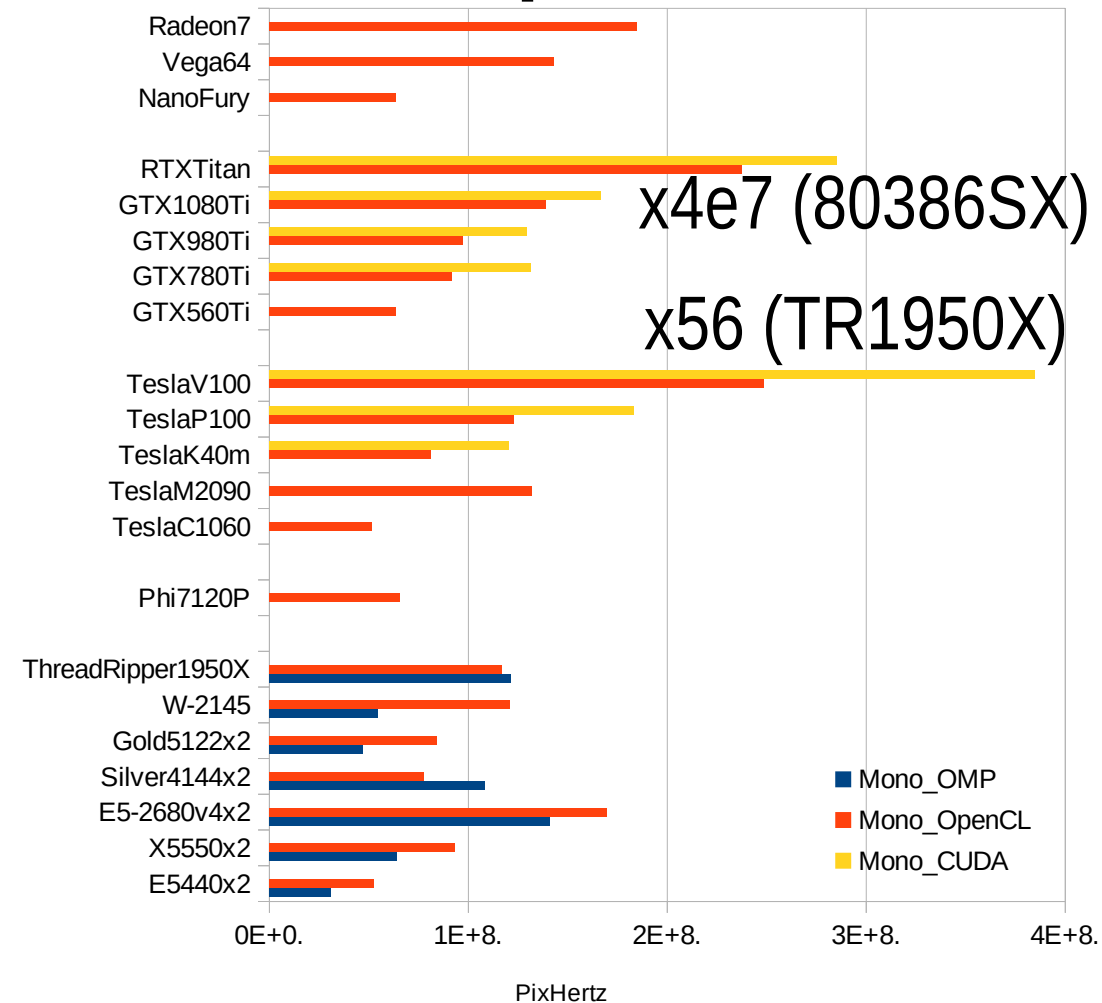
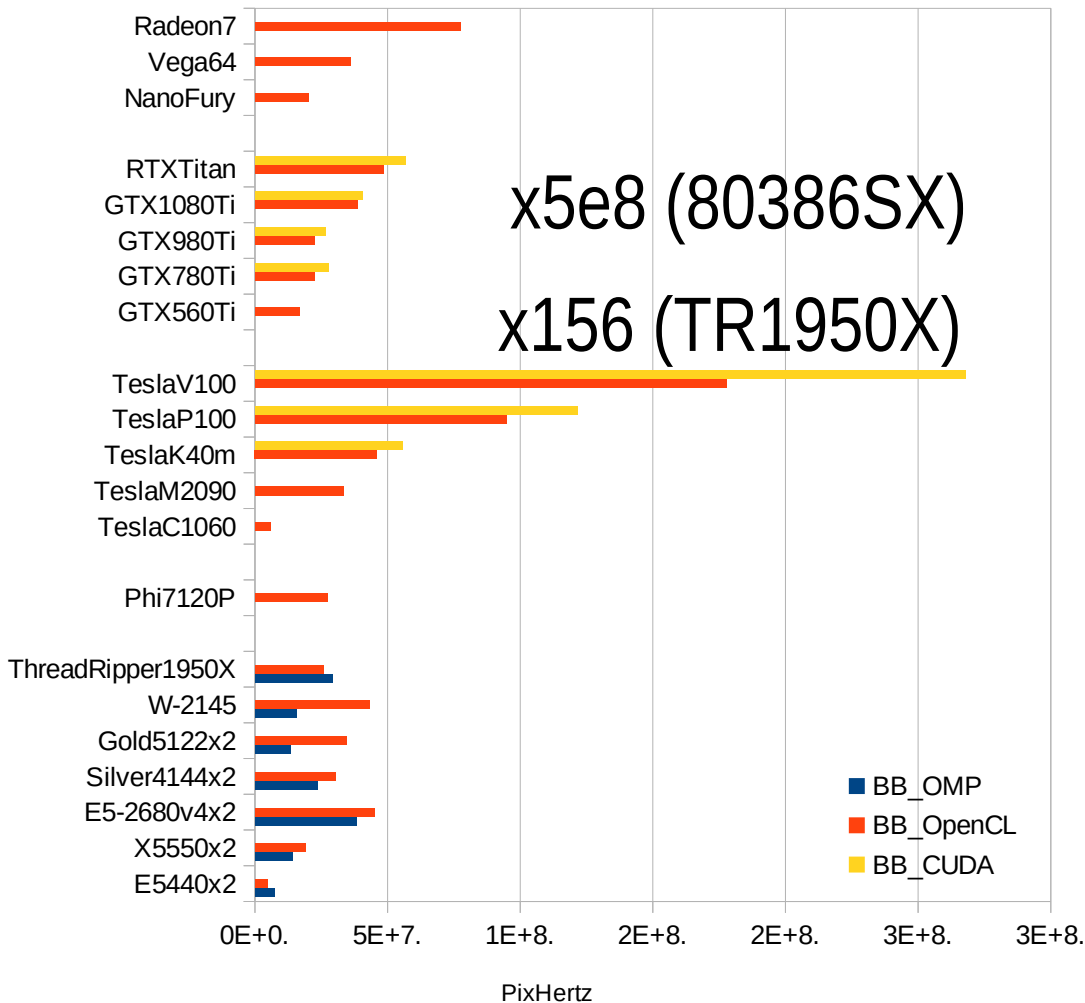
# Donc, en rajoutant CUDA, Il y a la Tesla V100 et les autres...



Face au CPU, le milliard en 30 ans, le millier en séquentiel

# Mais je serai un gros menteur...

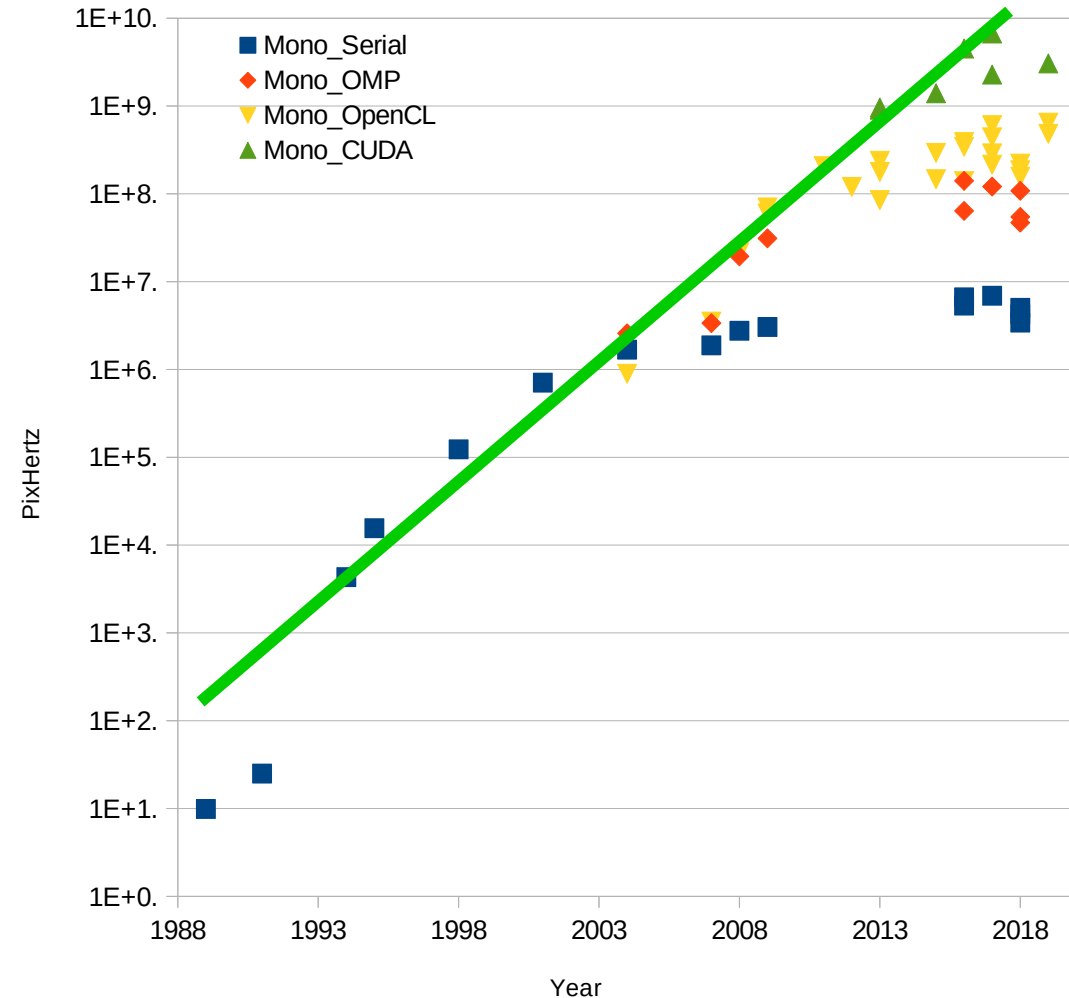
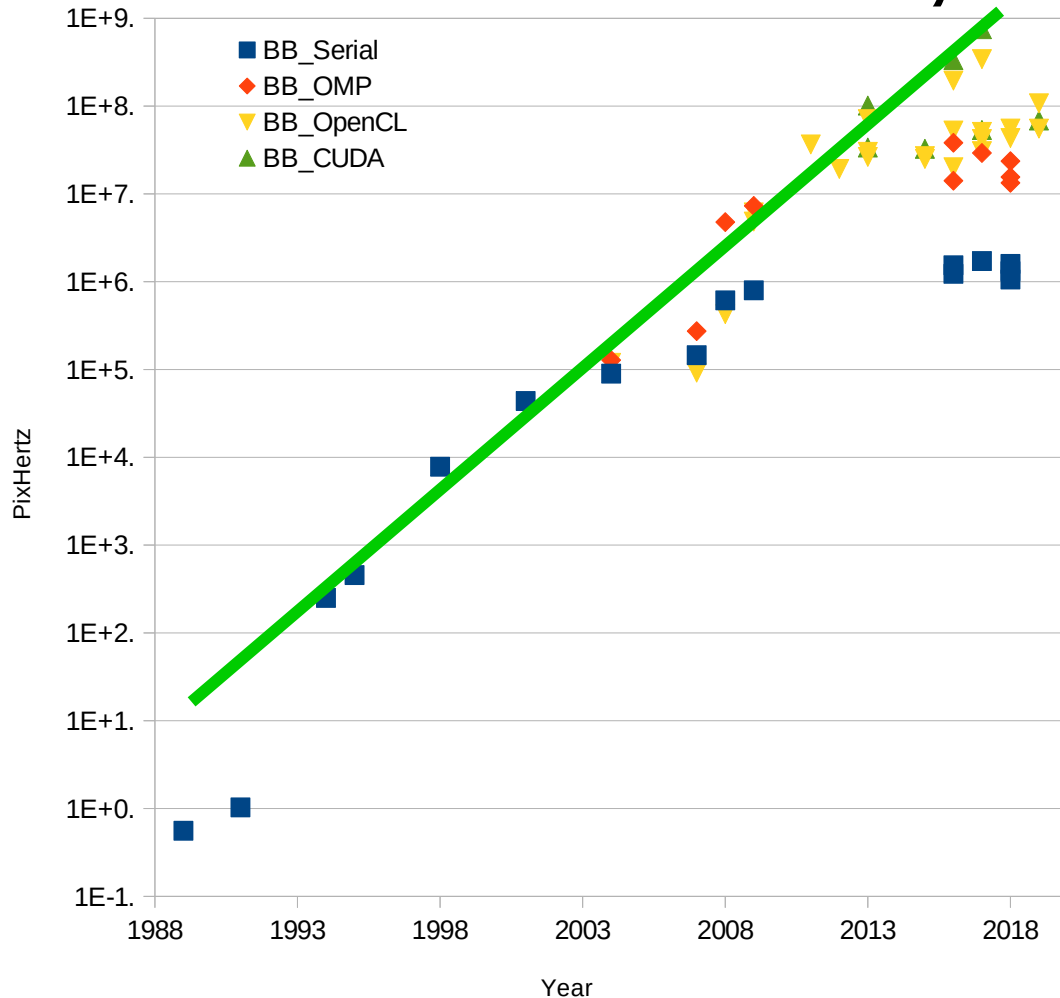
## Le transfert en CUDA et OpenCL...



Bref, dans ce cas, temps du transfert > temps de calcul

# Et la loi de Moore\* dans tout ça ?

## Grâce à CUDA, on est sur la droite !



Le (GP)GPU : même caractère « disruptif » que le FPU...

# En conclusion de ce portage... ... et son exécution sur nos \*PU

- Le Code, le chemin & les gains (temps écoulés) :
  - De Fortran à C en 1997 : x1
  - De C à C/OMP en 2019 : x22 (pour un 28 cœurs)
  - De C/OMP à Python/OpenCL sur CPU : x25 (pour un 8 cœurs)
  - Du CPU au GPU en Python/OpenCL : de x36 à x106
  - De Python/OpenCL à Python/CUDA : de x56 à x156
- Bref...
  - Il n'y a pas que CUDA ou C++ dans la vie... Il y a Python et OpenCL !
  - Vous pouvez tester :

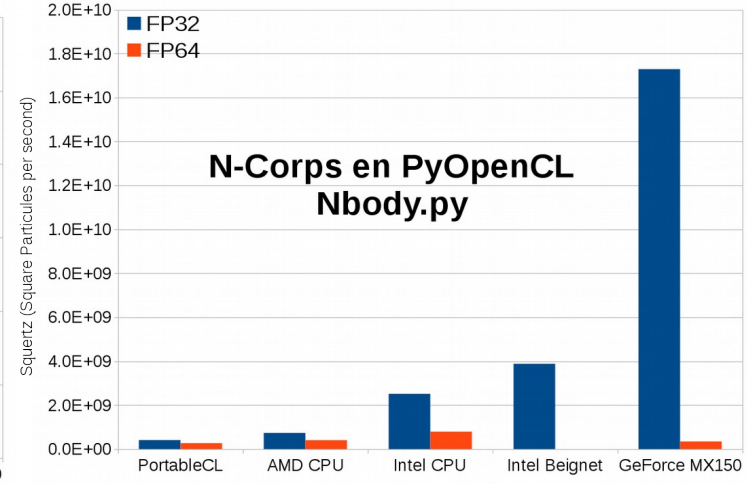
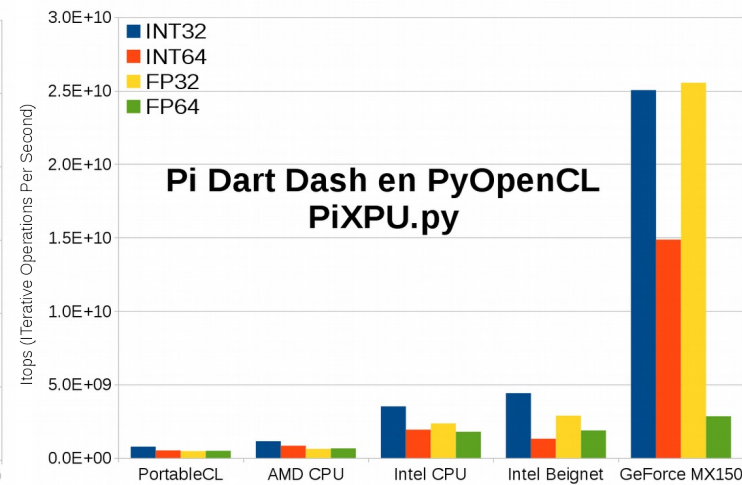
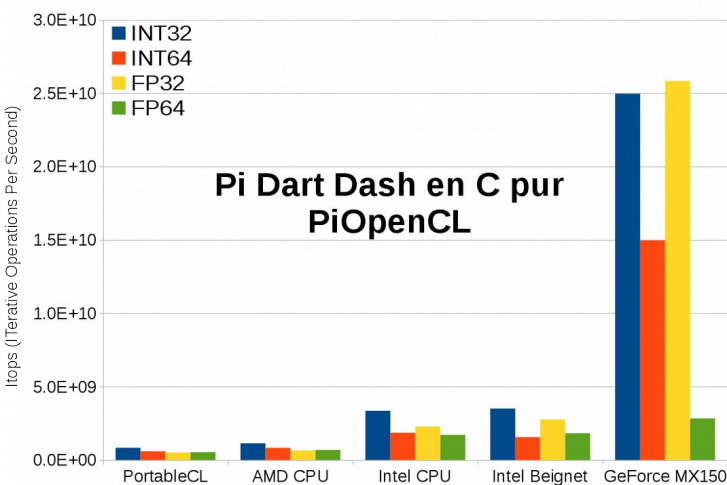
- <http://forge.cbp.ens-lyon.fr/redmine/projects/bench4gpu/repository/show/TrouNoir>

# Questions à se poser...

- Triptyque classique :
  - Où en est-on ? Le code existe-t-il ?
  - Où va-t-on ? A qui est-ce destiné ? De combien de temps dispose-t-on ?
  - Comment y va-t-on ? Quelle « approche » de programmation ?
- Le code exploite-t-il des routines « classiques » ?
  - Oui, mais de quelles dimensions sont les objets ?
- Quel est le grain de mes routines élémentaires ?
- L'espace mémoire nécessaire à mes routines  $< 40\text{GB}$ ?
- Est-ce que le 32 bits est suffisant ? (au moins partiellement)

# Et comment « commencer » ?

- Avec des exemples simples ;-)
  - Gros grain, grain fin, « ALU-bound », « memory bound »
  - Dépôt Subversion : `svn checkout https://forge.cbp.ens-lyon.fr/svn/bench4gpu/`
- Avec son portable (5 périphériques OpenCL)
  - Sur Google : `linkedin quemener gpu`
  - `https://fr.linkedin.com/pulse/cpu-vs-gpu-comment-les-comparer-ou-encore-osciller-sur-quemener`



# Mais à quoi ça sert tout ça ?

## Caractériser & éviter les regrets...

- Ce qu'il faut retenir :
  - Dans une machine, en 2020, la puissance « brute » est dans le (gros) GPU
  - Pour un régime de parallélisme bas, le CPU enfonce le GPU
  - Un GPU n'est supérieur au CPU que pour PR > 1000
  - Le GPU n'atteint son optimum QUE pour certains PR en OpenCL
  - Sans une « grosse » caractérisation, des déceptions à prévoir...
  - OpenCL est un bon compromis comme langage \*PU, mais pas que...
  - Sans Python, à chaque « système » son exécutable : recompilation nécessaire...
  - Python reste l'approche la plus rapide pour exploiter OpenCL
- Ce qu'il faut faire : expérimenter !

# Ateliers 3IP

## Donnez vos vieilles machines !

- 3IP : Introduction Inductive à L'informatique et au parallélisme
- Constat :
  - Rares sont les formations qui partent du matériel
- Objectif :
  - Mieux appréhender les usages par une connaissance du matériel
- Méthode :
  - Manipulation de composants informatiques



# 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](http://download.abandonware.org/magazines/Tilt/tilt_numero016/%5BMag%20-%20Fr%5D%20Tilt%20n%B0016%20-%20Page%20113%20%28Octobre%201984%29.jpg)