

## Master 2

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

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

Emmanuel Quémener

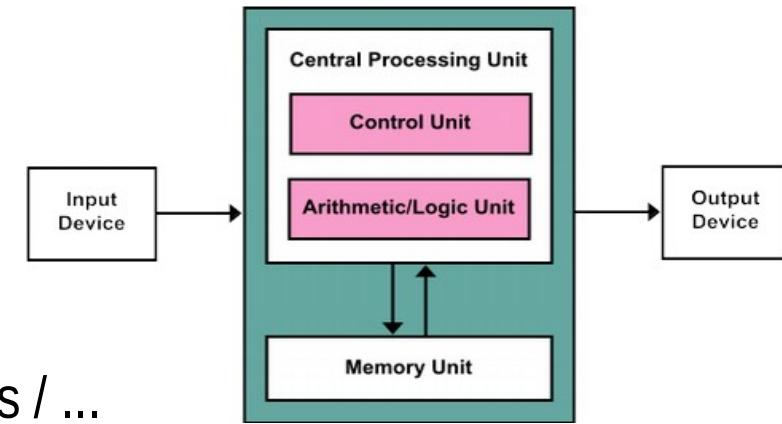
# Conseils avant écoute...

- Je suis le produit de l'université française d'il y a 25 ans...
- Je ne suis pas diplômé en informatique...
  - Mais j'utilise les ordinateurs depuis 1/3 de siècle et Linux depuis 22 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 2018
- 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 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)

# 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
  - Paillasses 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



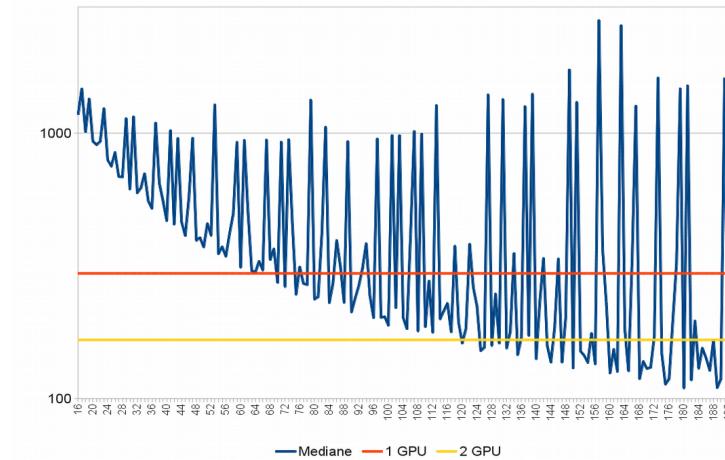
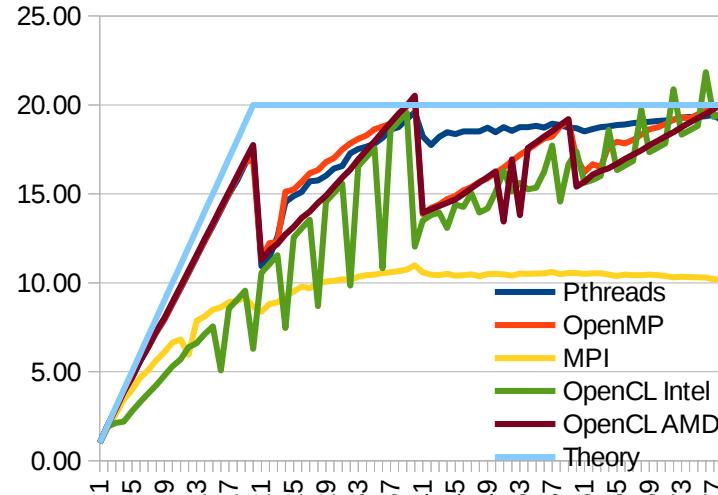
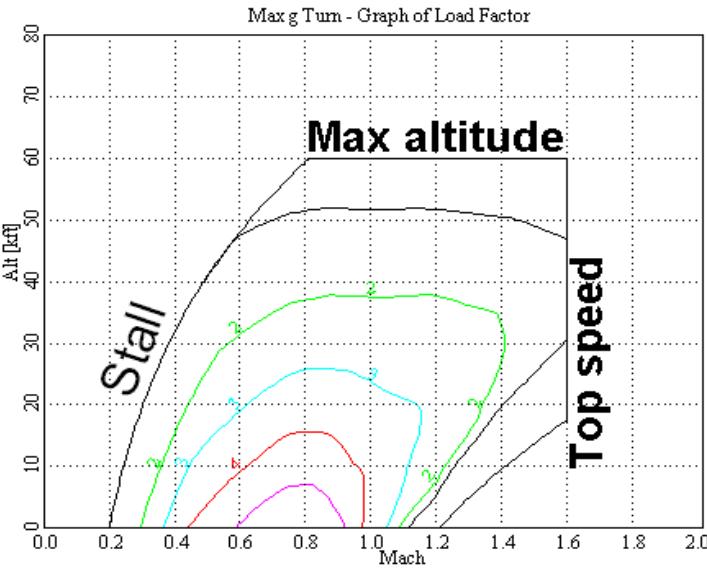
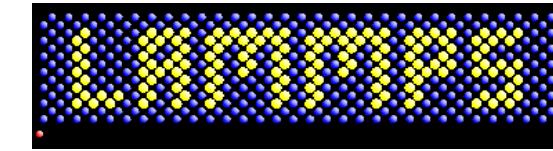
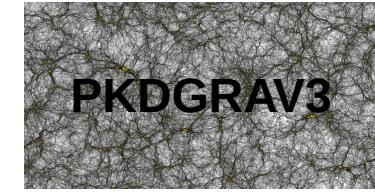
- Nasa X-29
- Cellule de F-5
- Moteur de F-18
- Train de F-16
- Etudes
  - Plans « canard »
  - Incidence  $>50^\circ$
  - « Fly-By-Wire »

Recycler, réutiliser, explorer de nouveaux domaines...

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



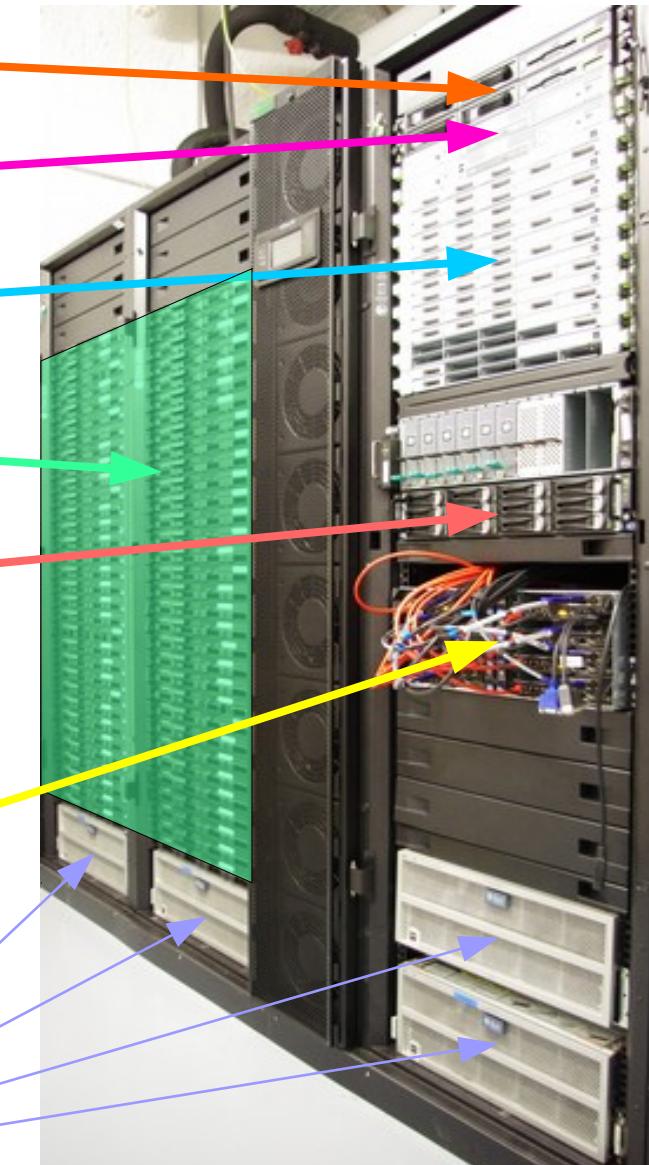
CBP



# Plateau multi-nœuds: 9 clusters 116 nœuds, 4 vitesses IB

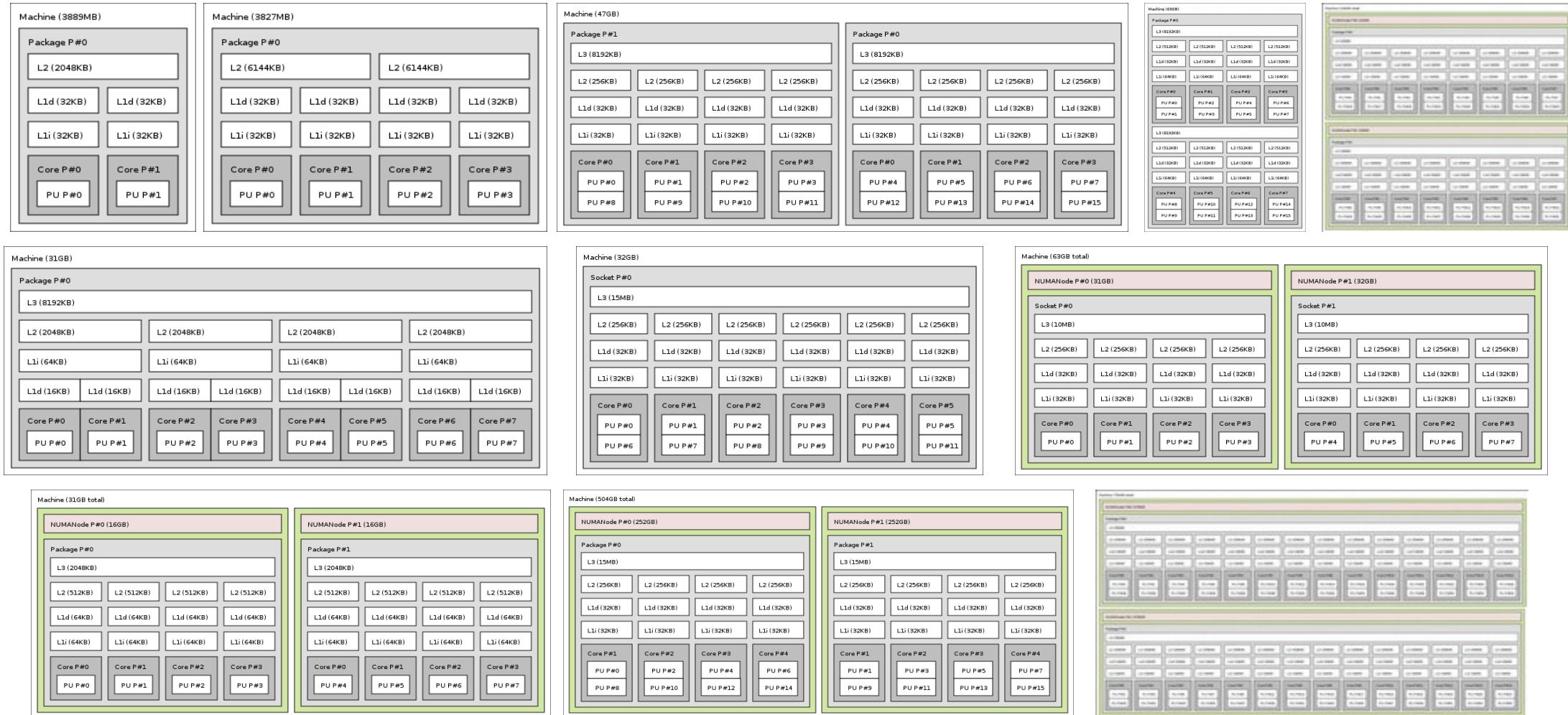


2 nodes Sun V20Z with AMD 250 4 physical cores @2400MHz Interconnection Infiniband SDR 10 Gb/s
2 nodes Sun X2200 with AMD 2218HE 8 physical cores @2600MHz Interconnection Infiniband DDR 20 Gb/s
8 nodes Sun X4150 with Xeon E5440 64 physical cores @2833MHz Interconnection Infiniband DDR 20 Gb/s
64 nodes Dell R410 with Xeon X5550 512 physical cores HT @2666MHz Interconnection Infiniband QDR 40 Gb/s
4 nodes Dell C6100 with Xeon X5650 48 physical cores HT @2666MHz Interconnection Infiniband QDR 40 Gb/s + C410X with 4 GPGPU
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
8 nodes Dell R410 with Xeon X5550 64 physical cores HT @2666MHz Interconnection Infiniband DDR 20 Gb/s
4 nodes Sun X4500 with AMD 285 16 physical cores @2400MHz Interconnection Infiniband SDR 10 Gb/s



# Plateau multi-cœurs

## De 2 à 28 coeurs: exemples...



# 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



# Plateau multi-shaders (GP)GPU

## 72 modèles différents...

**GPU Gamer : 18**

- 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 2080 Ti



**GPGPU : 9**

- 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

**GPU desktop & pro : 27**

- 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 620
- Nvidia GT 640
- Nvidia GT 710
- Nvidia GT 730
- Nvidia GT 1030
- Nvidia Quadro 2000M
- Nvidia Quadro K4000M
- Nvidia Quadro M2200
- Nvidia MX150



**GPU AMD Gamer & al : 18**

- 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

# Sur les Machines du CBP : SIDUS

## *Je n'installe pas, je démarre !*

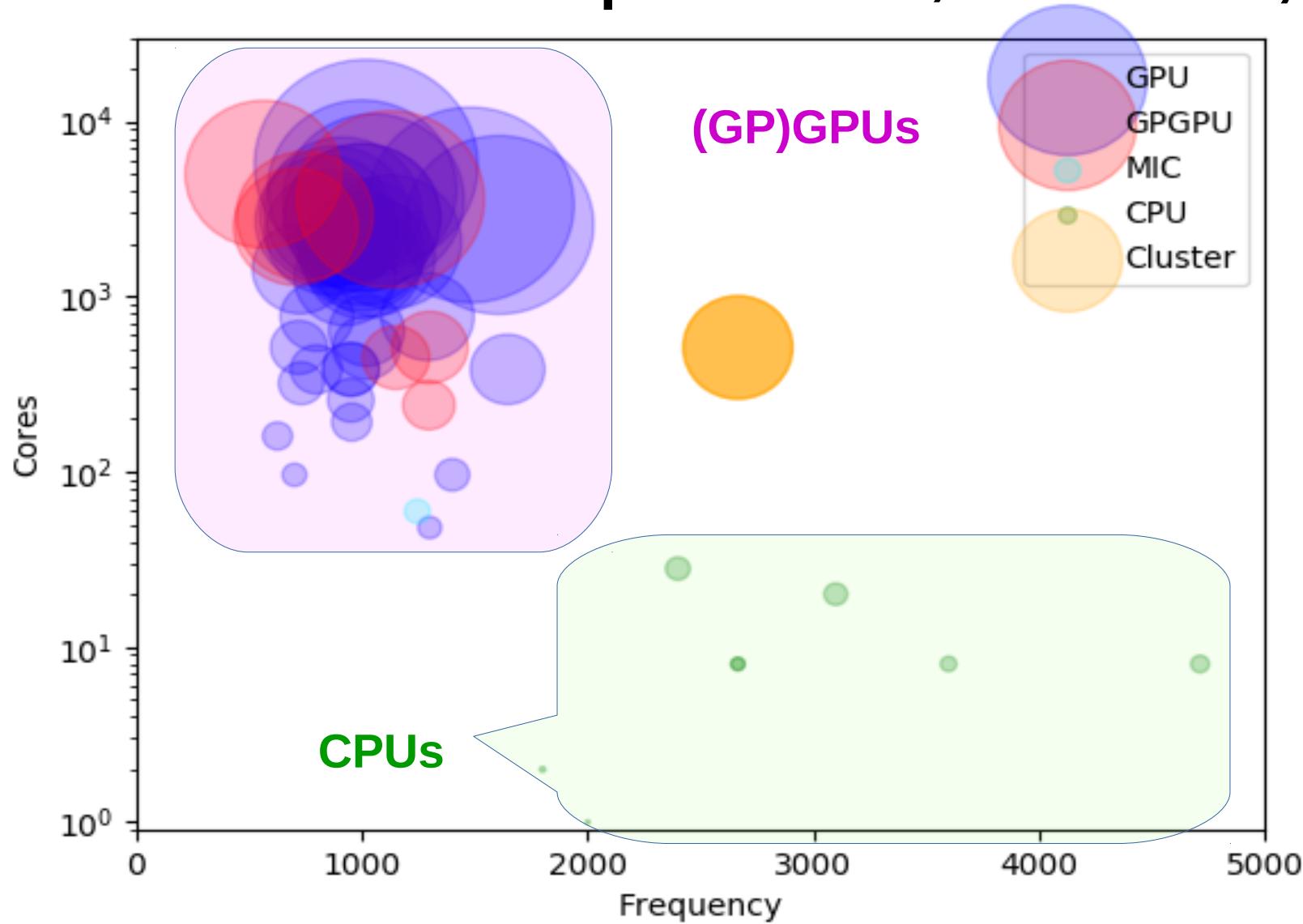
- Quoi ?
  - Déployer un système simplement sur un parc de machines
- Pourquoi ?
  - Assurer l'unicité des configurations
  - Limiter l'empreinte du système sur les disques
  - Assurer la reproductibilité dans le temps & l'espace
- Pour qui ?
  - Étudiants, enseignants, chercheurs, ingénieurs, ...
- Quand & Où ?
  - Centre Blaise Pascal : depuis 2010, près de 200 machines
  - PSMN : depuis 2011, plus de ~500 nœuds (sa propre instance)
  - Laboratoires : Chimie, UMPA, LBMC, IGFL, ISA, CRAL
- Comment ?
  - Utiliser un partage en réseau d'une arborescence
  - Détourner une astuce de LiveCD



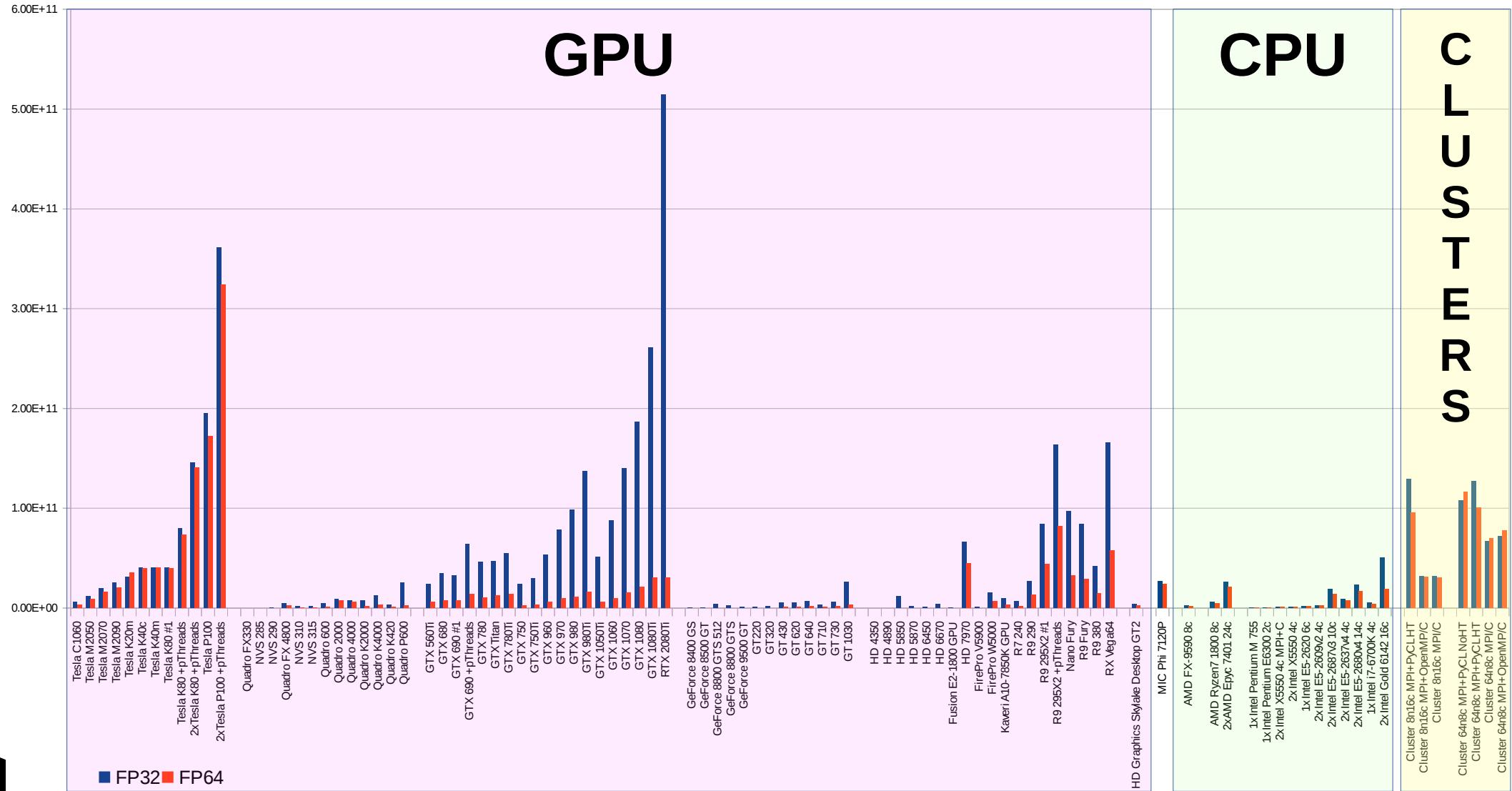
« Deux machines ayant démarré SIDUS ne peuvent pas ne pas avoir le même système ! »

# Comment les représenter ?

## Question de fréquences, cœurs, ...



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



# 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 ?



# Performance : Conditionnée par les objectifs

- « Vitesse » : Le temps écoulé (ou *Elapsed*) (mais seulement ?)
- Travail : immobilisation de ressources
- Efficacité : exploitation optimale des ressources
- Scalabilité : capacité à passer à une échelle supérieure
- Portabilité : intégration à d'autres infrastructures informatiques
- Maintenabilité : temps humain passé à maintenir le système
- Approche générale :
  - Définir un critère
  - Rechercher les valeurs extrêmales sur un ensemble de tests pertinent

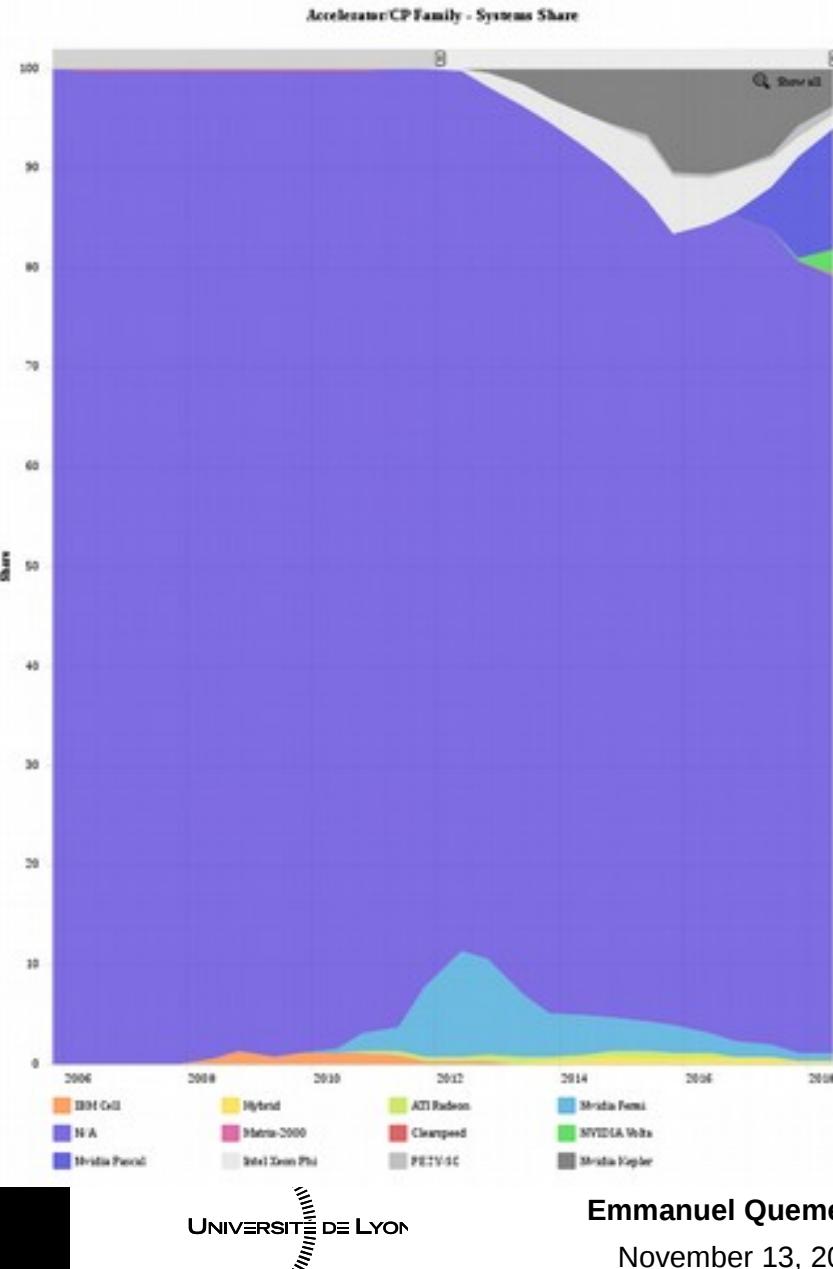
# La vitesse comme critère

## « Speed, I'm Speed... »

- Toutes les durées, et pas seulement le temps d'exécution
- Dans l'utilisation d'un code, les 3 coûts :
  - Coût d'entrée : apprendre à l'utiliser, l'intégrer à l'infrastructure, ...
  - Coût d'exploitation : le maintenir, l'utiliser
  - Coût de sortie : le remplacer par un autre code équivalent, ou une technologie équivalente
- Optimisation (et son biais) :  $DD/DE > 1$  est-il pertinent ?
  - DE : Durée totale de toutes mes exécutions
  - DD : temps passé à tenter de minimiser la durée d'exécution
- Pour estimer ces valeurs :
  - Outils système, outils de métrologie dans les langages, les codes, les matériels, ...
- « Et après moi ? Le déluge ? » : quel avenir pour le code ?

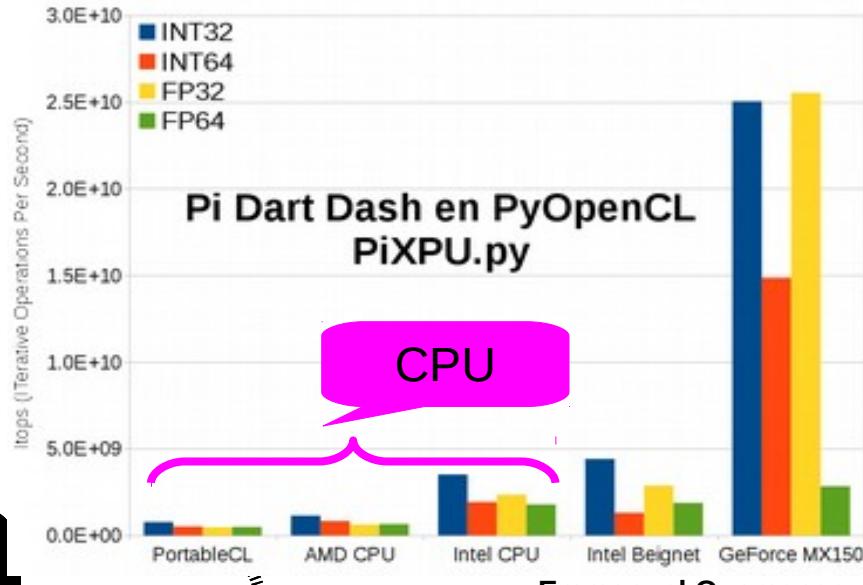
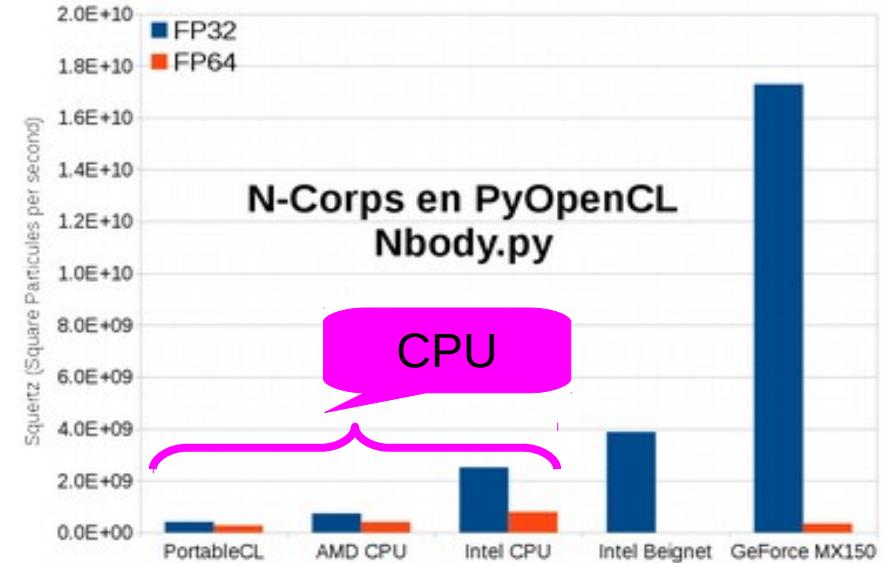
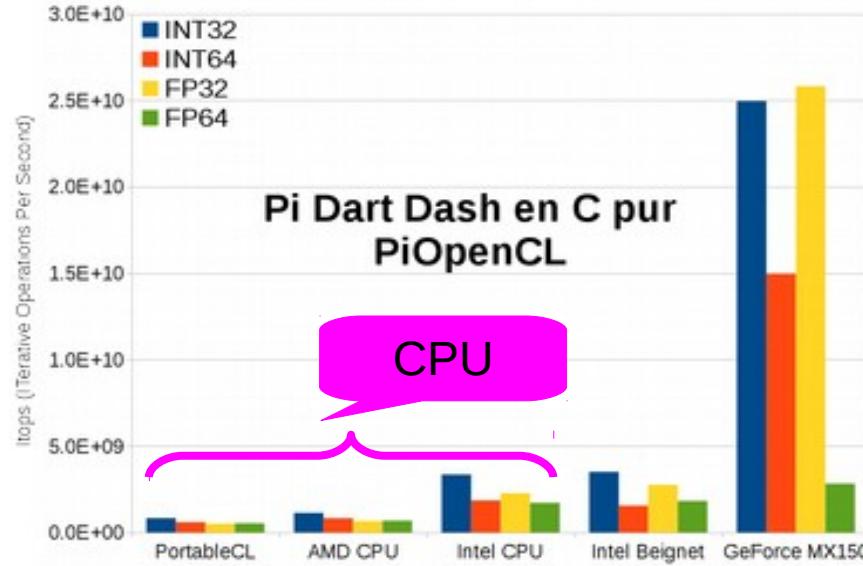


# Où en sommes-nous ? TOP 500 & les accélérateurs



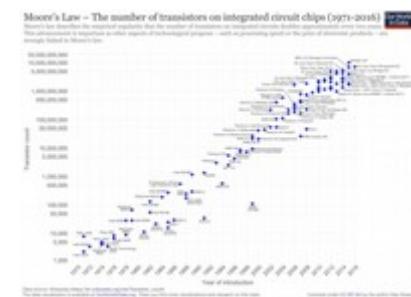
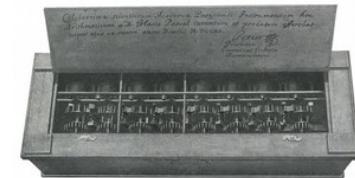
- Pourquoi : comparer les machines
- Quoi : le « linpack »
- Où : sur le Top 500
- Quand : juin 2018
- Combien :
  - ~1/4 avec accélérateurs :
    - MIC (Intel & copies)
    - GPGPU (Nvidia, AMD)
  - 7/10 dans le Top 10

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



# 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...



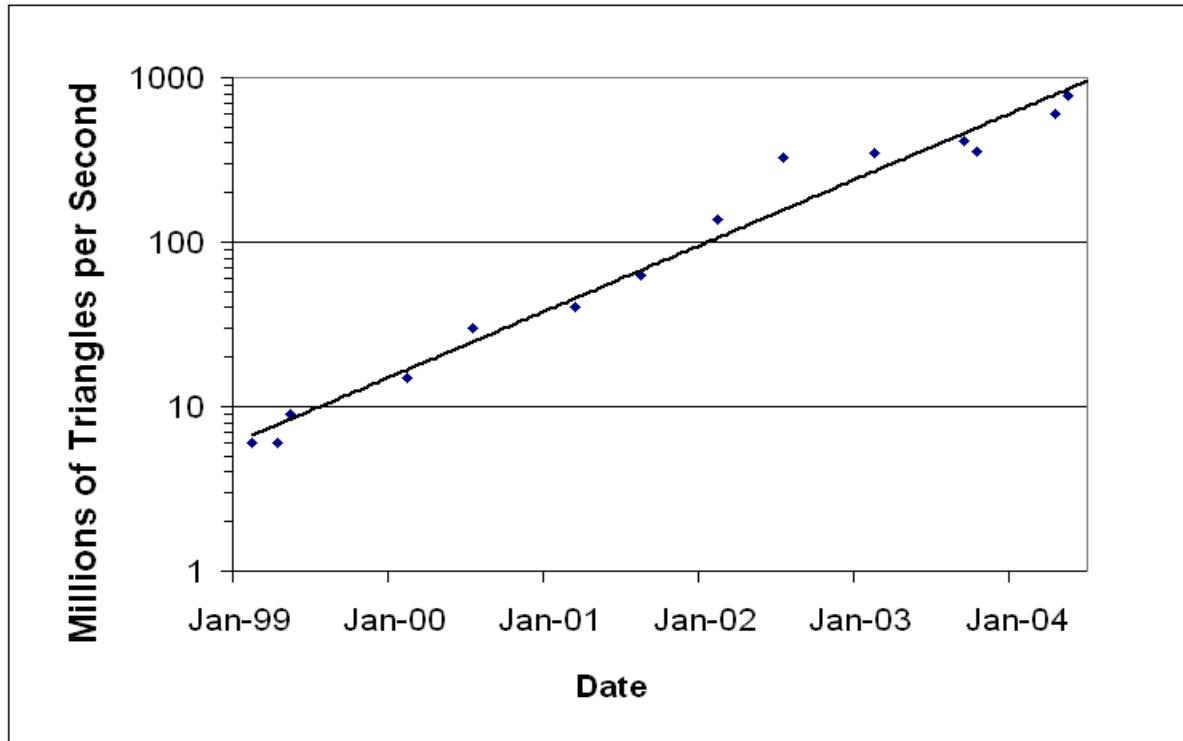
# Il y a une génération (humaine)... Un film de série B en 1984

- 1984 : The Last Starfighter
  - 27 minutes d'images synthétiques
  - $\sim 30.10^9$  opérations par image
  - Utilisation d'un Cray X-MP (130 kW)
  - 68 jours (en fait, 1 année nécessaire)
- 2018 : RTX 2080Ti (250 W)
  - 99 secondes
  - Comparison RTX 2080 Ti / Cray
    - Performance : 59 000 !
    - Consommation  $\sim 30\,000\,000$  !



# 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



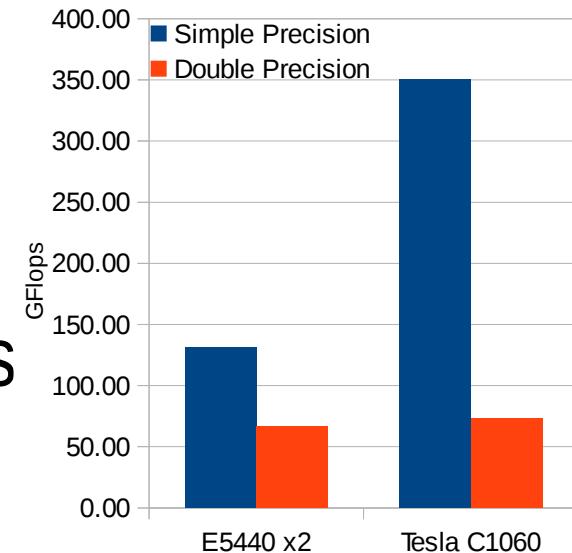
- Entre 2000 et 2015
  - GeForce 2 Go/GTX 980Ti : de 286 à 2816000 MOpérations/s : x10000
- Pour un CPU « classique » : x100

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

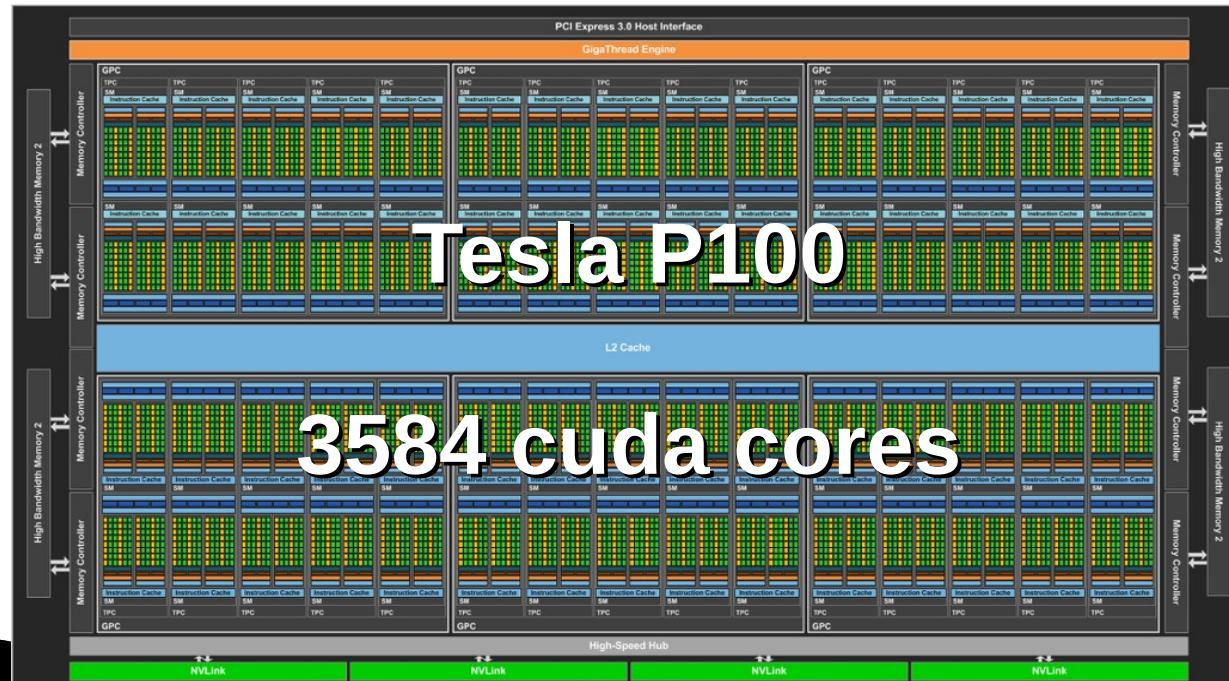
- Accélérateur ou la vieille histoire des coprocesseurs...
  - 1980 : 8087 (sur 8086/8088) pour les opérations en virgule flottante
  - 1989: 80387 (sur 80386) et son respect du IEEE 754
  - 1990 : 80486DX et l'intégration du FPU dans le CPU
  - 1997 : K6-3DNow ! & Pentium MMX : SIMD dans le CPU
  - 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

  - Cg Toolkit, CUDA, OpenCL

# Programmation : 1993

- OpenGL, Glide, Direct3D, ...

# Généricité : 2002

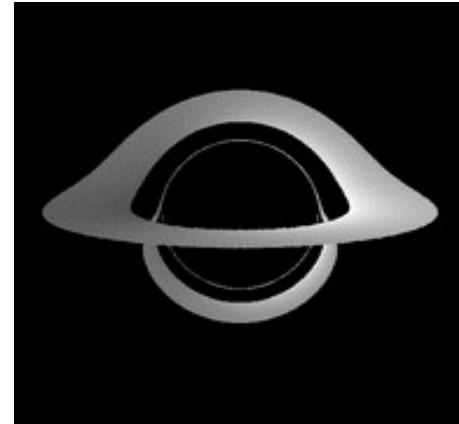
- CgToolkit, CUDA, OpenCL



# 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

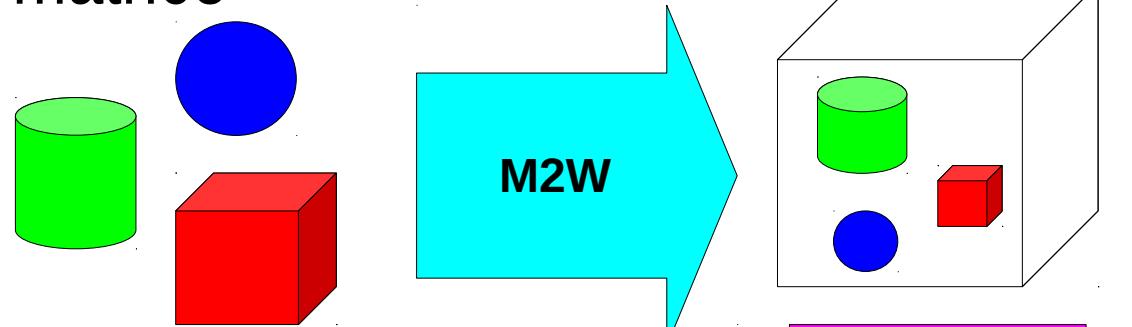


# Pourquoi le GPU est-il si puissant ?

## Shadering & Calcul matriciel

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

- Rotation
- Translation
- Mise à l'échelle



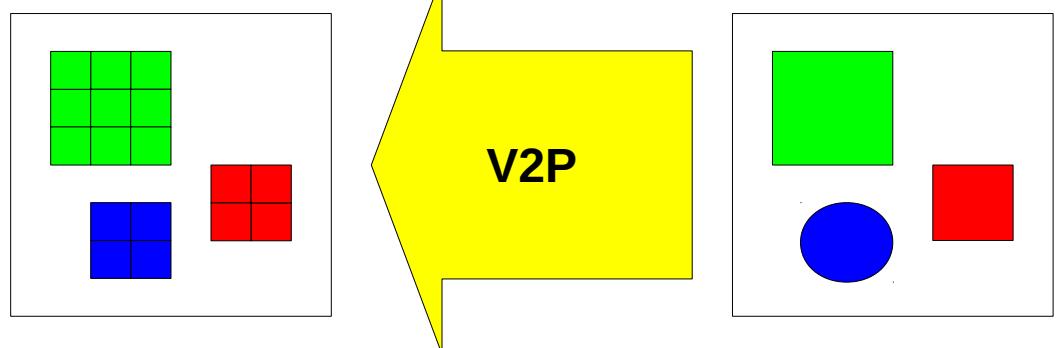
- **World 2 View** : 2 produits de matrice

- 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 » ?

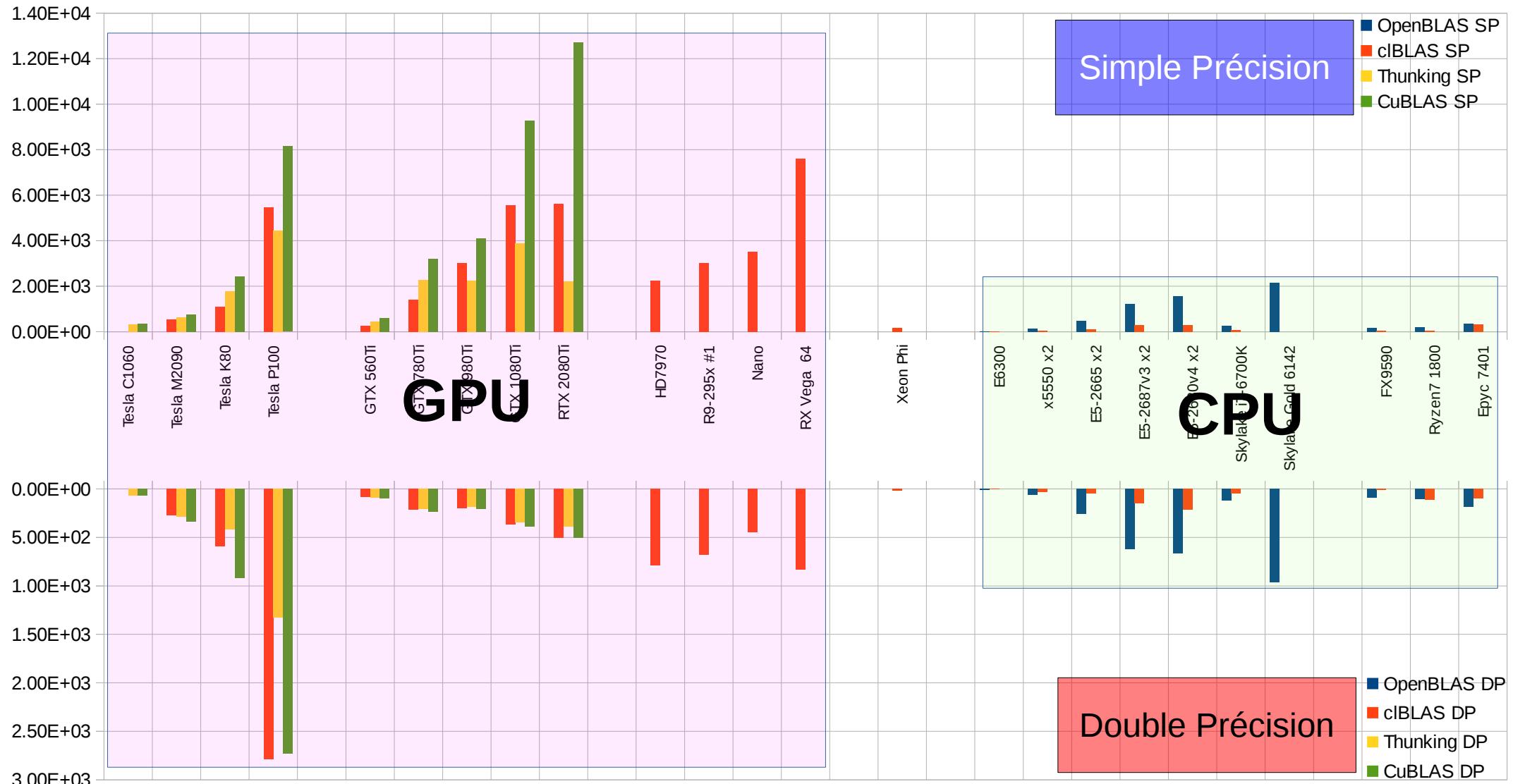
- Soit 2 matrices A et B de dimensions NxP et PxM
- Le produit matriciel de A par B donne C
- Chaque élément de C,  $C_{ij}$  pour i de 1 à N et j de 1 à M :

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

- 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 ?



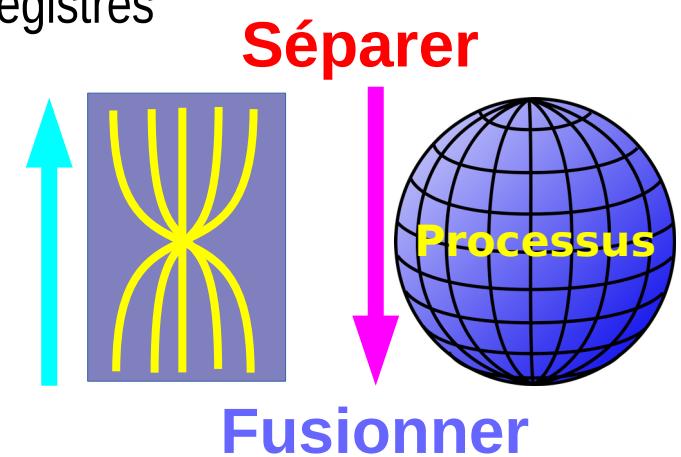
# Pourquoi le GPU est-il si puissant ? Toute la taxonomie de Flynn en «M»

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

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



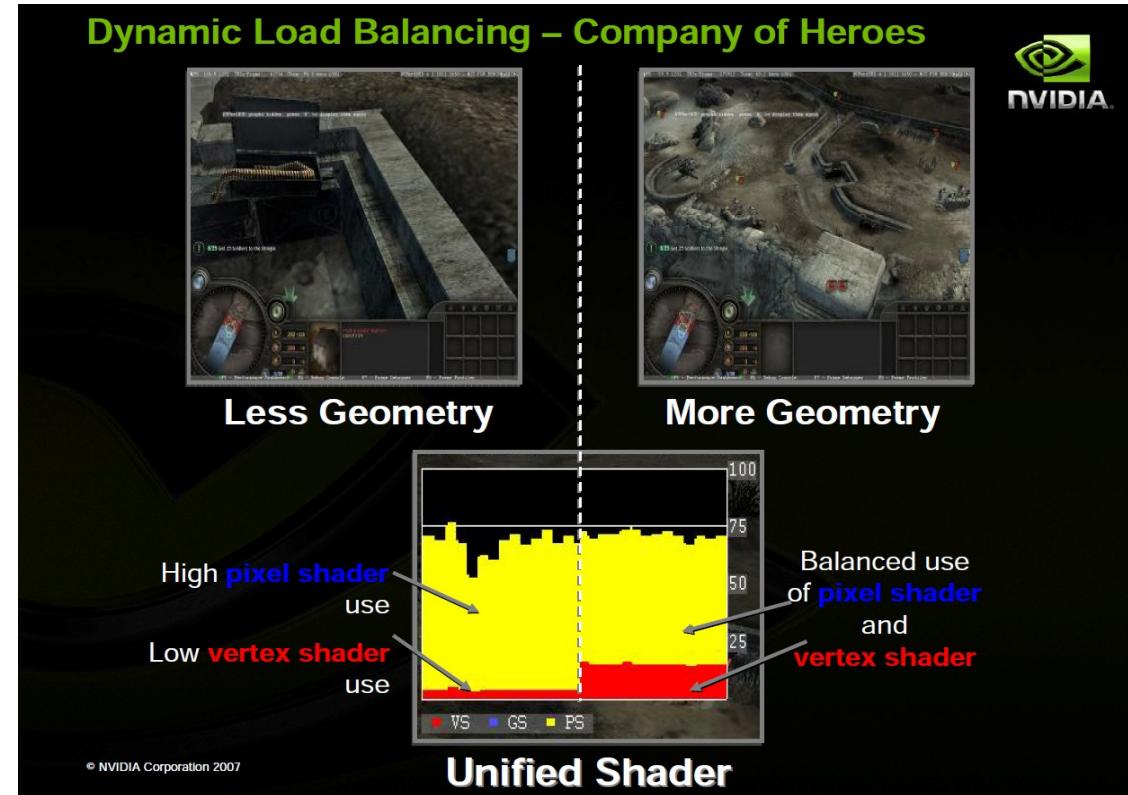
# Mais pas seulement un MM... Retour à un traitement générique

## 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
  - Des wrappers pour tous les langages (Python)
- 2007-06 : CUDA (Nvidia) ou l'arrivée d'un premier langage
- 2008-06 : Snow Leopard (Apple) intègre OpenCL
  - La volonté d'utiliser au mieux sa machine ?
- 2008-11 : OpenCL 1.0 et ses premières spécifications
- 2011-04 : WebCL et sa première version par Nokia (morts...)

# Développer ou intégrer ?

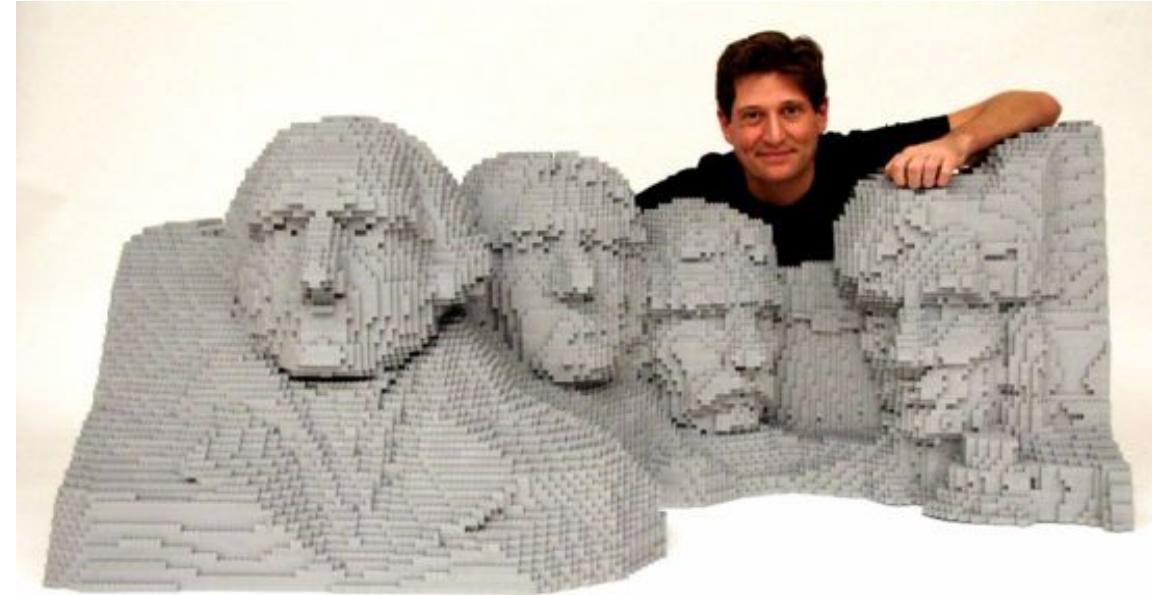


## Développer (de zéro)

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
<b>MPI</b>	Oui	Oui	Non	Non	Oui*
<b>PVM</b>	Oui	Oui	Non	Non	Oui*
<b>OpenMP</b>	Non	Oui	Non	Non	Oui*
<b>Pthreads</b>	Non	Oui	Non	Non	Oui*
<b>OpenCL</b>	Non	Oui	Oui	Oui	Oui
<b>CUDA</b>	Non	Non	Non	Oui	Non
<b>TBB</b>	Non	Oui	Non	Non	Oui*
<b>OpenACC</b>	Non	Oui	Non	Oui	Oui
<b>Kokkos</b>	Non	Oui	Oui	Oui	Oui

- OpenCL & Kokkos semblent les plus « universels »
- CUDA n'est seulement utilisable 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	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

# Des usages communs...

- De l'algèbre linéaire « basique » ou « avancée »
  - Basique avec BLAS
  - Avancée avec LAPACK
- De l'algèbre linéaire sur des systèmes « creux »
  - UMFPack
- De l'analyse spectrale avec des TF
  - FFTPack
- Des résolutions d'équations différentielles
  - PetSC
  - Trilinos

# Des adaptations logicielles aux évolutions technologiques

- En algèbre linéaire :
  - BLAS avec OpenBLAS, MKL (OpenMP), BLACS (MPI)
  - LAPACK avec ACML (OpenMP), Scalapack (MPI)
- En algèbre linéaire sur des systèmes creux
  - MUMPS (OpenMP, MPI) mais dépend de BLAS
- En analyse spectrale :
  - FFT avec FFTw3 (OpenMP), FFTw2 (MPI)

**Cohérent de voir « apparaître » des versions GPU !**

# Pour l'algèbre linéaire : la profusion

- Fonctions BLAS :
  - CuBLAS : complètes (mais d'efficacité diverses)
  - CIBlas : incomplètes
- Fonctions « creuses »
  - CuSparse
- Fonctions LAPACK :
  - Cula : recarrossage de LAPACK avec CuBLAS (mais CUDA 6)
  - Magma : encore actif (version 2.3.0 de 2017Q4)
- API Python : tout pour le langage « glue »...

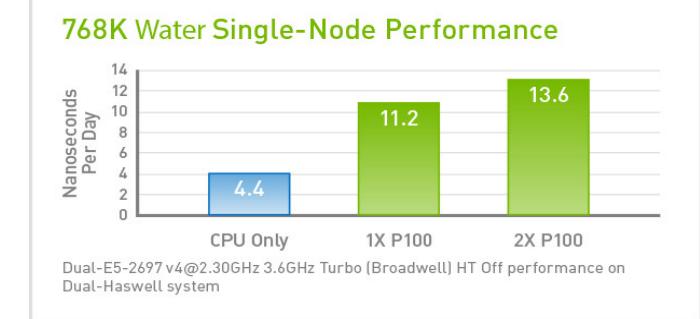
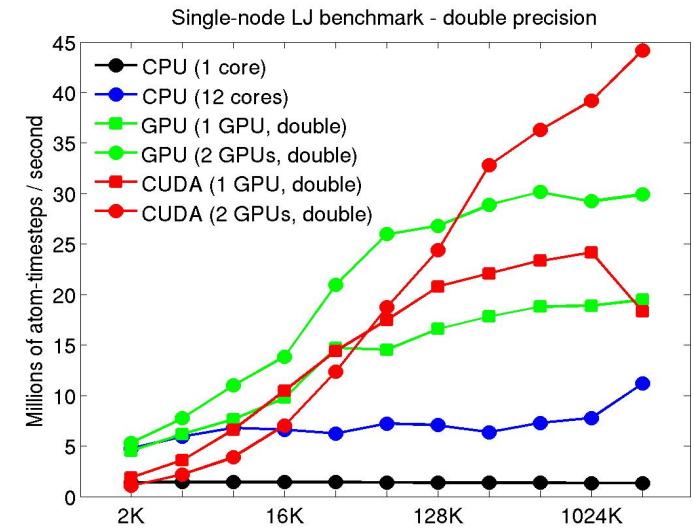
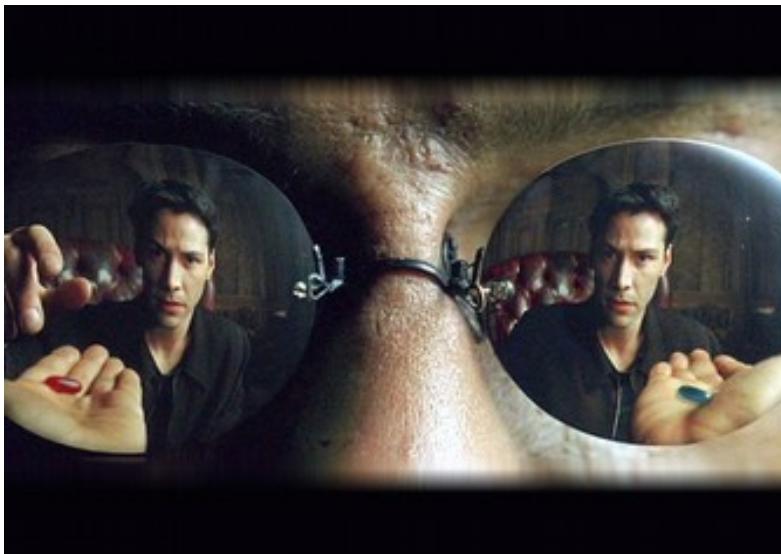
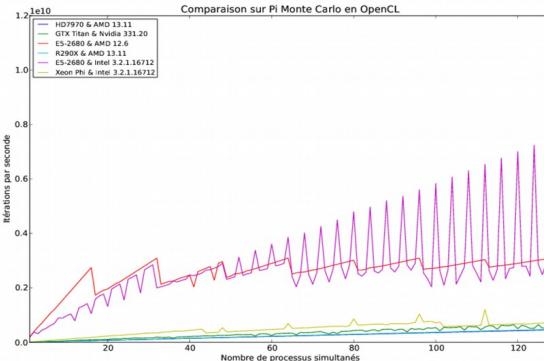
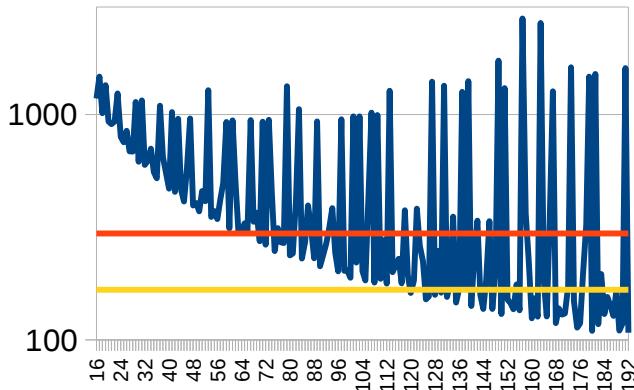
# Pour la Transformée de Fourier

- CuFFT : pour le C, C++, Fortran
  - Float sur 32, Double sur 64, Complex en 32 et en 64
  - 1D, 2D, 3D
  - Mixte sur les sorties : S2C, D2Z,
- clFFT : pour le C, C++
  - Float sur 32, Double sur 64, Complex en 32 et en 64
  - 1D, 2D, 3D
- Reikna pour Python

# Et mon code « pur » ?

- Approche ACC :
  - Une approche « pragma »tique à la OpenMP
  - OpenACC : une initiative (seulement une initiative hors PGI...)
  - 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 » ?

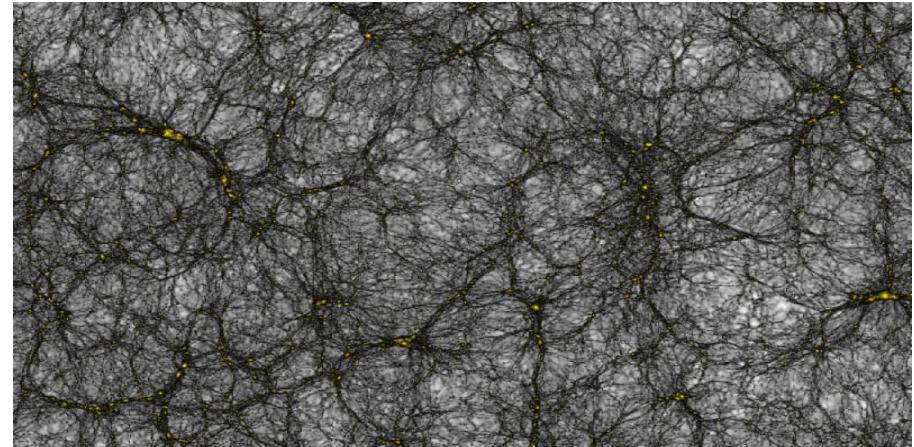


# Cadre posé... Quels codes à étudier ?

- Code « Ikea » : à compiler soi-même
  - PKDGRAV3 : astrophysique
  - Gromacs : chimie
- Code « à dépendances » :
  - Approche « intégrateur » :
    - BLAS avec xGEMM et les autres
  - Approche « développeur » :
    - Pi Dart Dash, Nbody, Splutter...

# PKDGRAV3 : un client prometteur ... et une bonne exposition média !

- Caractéristiques :
  - Open Source
  - Hybride : MPI, OpenMP, Cuda
- Compilation « facile »
  - Dans la doc (README) :
    - Quickstart : ./configure ; make
    - Pour le support CUDA :
      - ./configure --with-fftw --enable-integer-positions --with-cuda
      - make -j 16



# PKDGRAV3 : compilation facile ?

## Un peu plus compliqué sur Debian 9

- Pas de config (donc nécessité de le générer) : ./autogen.sh
- Nécessité de compiler avec fPIC (donc nécessité de modifier) :
  - sed -i 's/compute\_35/compute\_35\\ \\-Xcompiler\\ \\-fPIC/g' configure.ac
- Donc en fait, compiler PKDGRAV3 avec le support CUDA, c'est :
  - cd /local ; git clone https://bitbucket.org/dpotter/pkdgrav3.git
  - PKDGRAV3=/local/pkdgrav3-\$\{date "+%Y%m%d"\} ; mv pkdgrav3 \\$PKDGRAV3
  - cd \\$PKDGRAV3
  - sed -i 's/compute\_35/compute\_35\\ \\-Xcompiler\\ \\-fPIC/g' configure.ac
  - ./autogen.sh
  - ./configure --with-fftw –enable-integer-positions --with-cuda
  - make -j 16

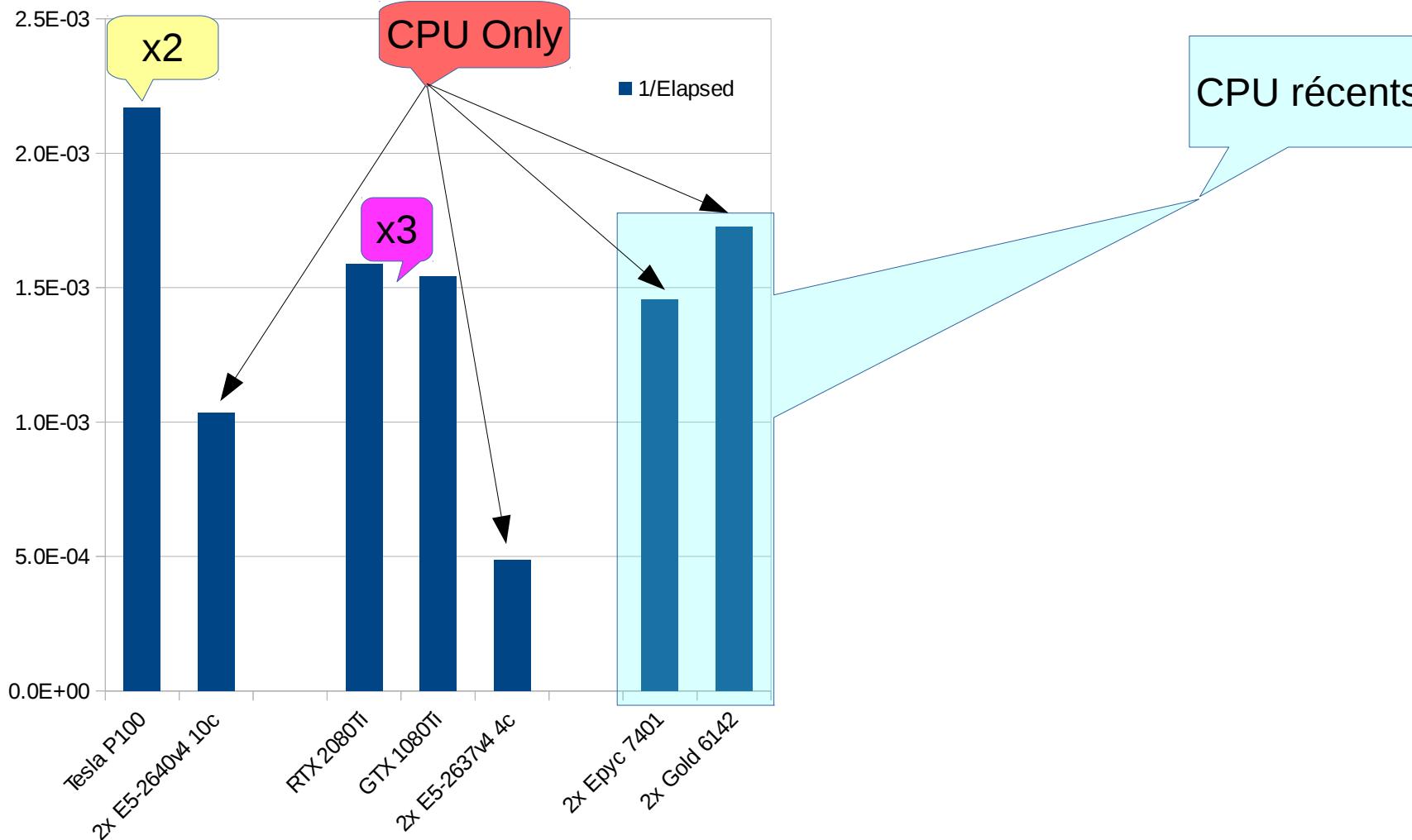
# PKDGRAV3 : on veut comparer !

## Donc, pour une pure « MPI »

- On ajoute aux commandes précédentes :
  - mv pkdgrav3\_mpi pkdgrav3\_mpi\_cuda
  - make distclean-recursive
  - ./autogen.sh
  - ./configure --with-fftw --enable-integer-positions
  - make -j 16
- Et comme test ?
  - cosmology.par dans examples...

# PKDGRAV3 en FP64

## Comparaison CPU & GPU



# Gromacs : le bon « client » de la chimie

- Socle matériel : les plates-formes du CBP
  - 51 GPU différents, 20 types de CPU
- Socle logiciel : Debian 9.4 dite « stretch » en AMD64
- Caractéristiques :
  - Open Source
  - Hybride : multi-nœuds, multi-cœurs, multi-shaders
  - Possibilité d'exécution CPU (MPI+OpenMP), CPU+GPU
  - Paramétrage facile du parallélisme et du choix du GPU

# Gromacs : phase préparatoire

## En lisant la documentation

- Expansion archive, déplacement, création dossier, déplacement
  - tar xfz gromacs-2018.1.tar.gz ; cd gromacs-2018.1 ; mkdir build ; cd build
- Paramétrage compilation avec options :
  - cmake .. -DGMX\_BUILD\_OWN\_FFTW=ON -DREGRESSIONTEST\_DOWNLOAD=ON
- Compilation et installation
  - make ; make check ; sudo make install
- Chargement de l'environnement :
  - source /usr/local/gromacs/bin/GMXRC

# Gromacs : phase préparatoire

## Nos contraintes & nos 1<sup>ers</sup> déboires

- Plusieurs contextes : à chaque processeur/GPU, sa compilation
  - Plusieurs dossiers d'installation des exécutables
- Compilateurs standards Stretch : gcc, g++ version 6.3
- Lancement cmake : KO

```
CMake Error at cmake/gmxManageGPU.cmake:289 (message):
NVCC/C compiler combination does not seem to be supported. CUDA frequently
does not support the latest versions of the host compiler, so you might
want to try an earlier C/C++ compiler version and make sure your CUDA
compiler and driver are as recent as possible.

Call Stack (most recent call first):
CMakeLists.txt:582 (gmx_gpu_setup)

-- Configuring incomplete, errors occurred!
See also "/local/Gromacs/gromacs-2018.1/build_CUDA/CMakeFiles/CMakeOutput.log".
See also "/local/Gromacs/gromacs-2018.1/build_CUDA/CMakeFiles/CMakeError.log".
```

- Trouver un autre compilateur !

# Gromacs : phase préparatoire

## Les déboires continuent

- Passage sur clang version 3.8, mais...

```
-- Adding work-around for issue compiling CUDA code with glibc 2.23 string.h
CMake Warning at cmake/gmxManageGPU.cmake:272 (message):
  To use GPU acceleration efficiently, mdrun requires OpenMP multi-threading.
  Without OpenMP a single CPU core can be used with a GPU which is not
  optimal. Note that with MPI multiple processes can be forced to use a
  single GPU, but this is typically inefficient. You need to set both C and
  C++ compilers that support OpenMP (CC and CXX environment variables,
  respectively) when using GPUs.
Call Stack (most recent call first):
  CMakeLists.txt:582 (gmx_gpu_setup)
```

- Exécution dégradée (pas de OpenMP)
- Phases cmake & compilation effroyablement lente (plus de 10x)
- Finalement crash à la compilation... :-(
- Passage sur compilateur plus ancien 4.9

# Gromacs : phase préparatoire

## Importation d'un vieux compilateur

- Exploitation des « snapshots » : tout le script d'installation...
  - wget <http://snapshot.debian.org/package/gcc-4.9/4.9.4-2/>
  - mkdir gcc-4.9.4
  - cd gcc-4.9.4
  - wget -O DebianSnapshot-gcc-4.9.4.html <http://snapshot.debian.org/package/gcc-4.9/4.9.4-2/>
  - egrep '(\_amd64.deb|\_all.deb)' DebianSnapshot-gcc-4.9.4.html | awk -F\" '{ print \$2 }' | grep -v kfreebsd | grep -v lib32 | grep -v libx32 | sort -u | grep -v multilib | xargs -I '{}' wget [http://snapshot.debian.org/{}'](http://snapshot.debian.org/{})
  - sudo dpkg -i \*deb
  - sudo apt-get -f install
- Ca y est ! Un vieux compilateur prêt pour la route...

# Gromacs : enfin la compilation !

## Tout le script, pour un CUDA

- mkdir -p /local/Gromacs/ ; cd /local/Gromacs/
- wget <ftp://ftp.gromacs.org/pub/gromacs/gromacs-2018.1.tar.gz>
- export GPU=GTX1080Ti
- export CC=/usr/bin/gcc-4.9 ; export CXX=/usr/bin/g++-4.9
- export GMXINSTALL=/scratch/Gromacs/2018.1
- export GMXSRC=/local/Gromacs/gromacs-2018.1
- tar xzf gromacs-2018.1.tar.gz ; cd gromacs-2018.1
- mkdir -p \$(dirname \$GMXINSTALL) ; mkdir \$GMXSRC/build\_\${GPU}
- cd \$GMXSRC/build\_\${GPU}
- cmake .. -DGMX\_BUILD\_OWN\_FFTW=ON -DGMX\_GPU=on  
-DREGRESSIONTEST\_DOWNLOAD=ON -DCMAKE\_INSTALL\_PREFIX=\${GMXINSTALL}\_\$  
{GPU}
- make -j 16 ; make check ; make install

# Gromacs : cas d'usage Nvidia

## Paramètres d'exécution

- Tâche coûteuse (et parallélisée) : mdrun
  - -ntmpi : contrôle du nombre de processus MPI concurrents
  - -ntomp : contrôle du nombre de processus OpenMP concurrents
  - -gpu\_id : contrôle l'exécution sur un (ou plusieurs) GPU
  - -nb cpu : contrôle l'exécution sur uniquement le CPU
- Si aucun paramètre donné : mdrun explore la machine
  - Autant de processus OpenMP que de cœurs détectés (si aucun OMP\_NUM\_THREADS)
  - Autant de processus MPI que de cœurs détectés (si OMP\_NUM\_THREADS mis à 1)
- Dans une exécution multigpu : -gpu\_id 01 active le 0 et le 1
  - Pour les Nvidia, préférez la variable d'environnement : CUDA\_VISIBLE\_DEVICES

# Le banc de test... D'abord les 9 (GP)GPUs

- GPU de gamer Nvidia : pilote 375.82
  - GTX 1080Ti : architecture Pascal, 3584 cudacores, 1582 GHz
  - GTX 980Ti : architecture Maxwell, 2816 cudacores, 1075 MHz
  - GTX 780Ti : architecture Kepler, 2880 cudacores, 875 MHz
- GPGPU de Nvidia : pilote 375.82
  - Tesla P100 : architecture Pascal, 3584 cudacores, 1126 GHz
  - Tesla K80 : architecture Kepler, 2x 2496 cudacores, 560 MHz
  - Tesla K40m : architecture Kepler, 2880 cudacores, 745 MHz
- GPU de gamer AMD : pilote 17.40 (2482.3)
  - Vega 64 : architecture GFX900, 4096 streamprocessors, 1406 GHz
  - R9-Fury : architecture Fiji, 3584 streamprocessors, 1000 GHz
  - R9-295X2 : architecture Hawaii, 2x 2816 streamprocessors, 1018 GHz

# Le banc de Test : Ensuite leur socle, les 7 CPU

- Ryzen7 1800 : 8 cœurs, 16HT, 3.6 GHz
  - Socle du R9 Fury
- Skylake i7-6700K : 4 cœurs, 8HT, 4 GHz
  - Socle du Vega64 et du R9-295X2
- E5-2637v4 : 2x4 cœurs, 16HT, 3.5 GHz
  - Socle du GTX1080 Ti
- E5-2640v4 : 10 cœurs virtualisés, 2.4 GHz
  - Socle du Tesla P100
- E5-2637v2 : 4 cœurs virtualisés, 3.5 GHz
  - Socle du Tesla K40m
- E5-2607v2 : 2x4 cœurs, 2.5 GHz
  - Socle du GTX980Ti
- E5- 2620 : 6 cœurs, 12 HT, 2 GHz
  - Socle du GTX 780Ti

# Gromacs : cas d'usage #1

## « Test » Nvidia

- Configuration des chemins et dossiers d'exécution
  - export GPU=GTX1080Ti
  - mkdir -p /scratch/Gromacs/Test/\$(hostname)\_\${GPU}
  - cd /scratch/Gromacs/Test/\$(hostname)\_\${GPU}
  - GRODIR=/scratch/Gromacs/2018.1\_\${GPU}
  - source \$GRODIR/bin/GMXRC
- Chargement & expansion de l'archive de paramètres d'entrée
  - wget ftp://ftp.gromacs.org/pub/benchmarks/water\_GMX50\_bare.tar.gz
  - [ -d water-cut1.0\_GMX50\_bare ] && rm -r water-cut1.0\_GMX50\_bare
  - tar -zxvf water\_GMX50\_bare.tar.gz ; cd water-cut1.0\_GMX50\_bare/1536
- Exécution des tâches
  - \$GRODIR/bin/gmx grompp -f pme.mdp
  - /usr/bin/time \$GRODIR/bin/gmx mdrun -resethway -noconfout -nsteps 4000 -v -gpu\_id 0

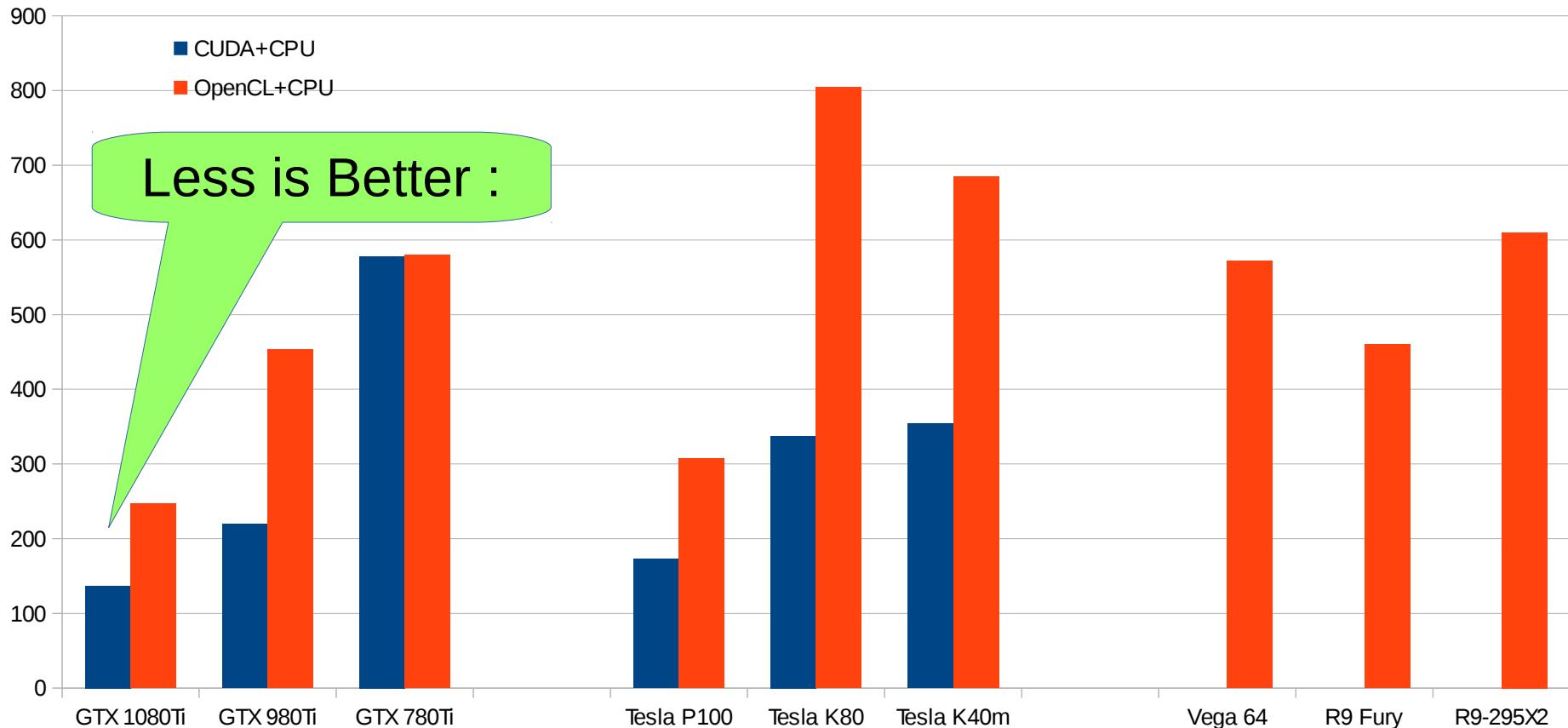
# Gromacs : cas d'usage #1

## « Test » Nvidia : sortie « time »

- TIME Command being timed: "/scratch/Gromacs/2018.1\_GTX980Ti\_OpenCL/bin/gmx mdrun -resethway -noconfout -nsteps 4000 -v -gpu\_id 0"
- TIME User time (seconds): 2767.16
- TIME System time (seconds): 25.98
- [REDACTED]
- TIME Percent of CPU this job got: 616%
- TIME Average shared text size (kbytes): 0
- TIME Average unshared data size (kbytes): 0
- TIME Average stack size (kbytes): 0
- TIME Average total size (kbytes): 0
- TIME Maximum resident set size (kbytes): 4820892
- TIME Average resident set size (kbytes): 0
- TIME Major (requiring I/O) page faults: 39
  - TIME Minor (reclaiming a frame) page faults: 1248968
  - TIME Voluntary context switches: 288413
  - TIME Involuntary context switches: 447953
  - TIME Swaps: 0
  - TIME File system inputs: 191072
  - TIME File system outputs: 0
  - TIME Socket messages sent: 0
  - TIME Socket messages received: 0
  - TIME Signals delivered: 0
  - TIME Page size (bytes): 4096
  - TIME Exit status: 0

# Cas d'usage #1 : test Nvidia

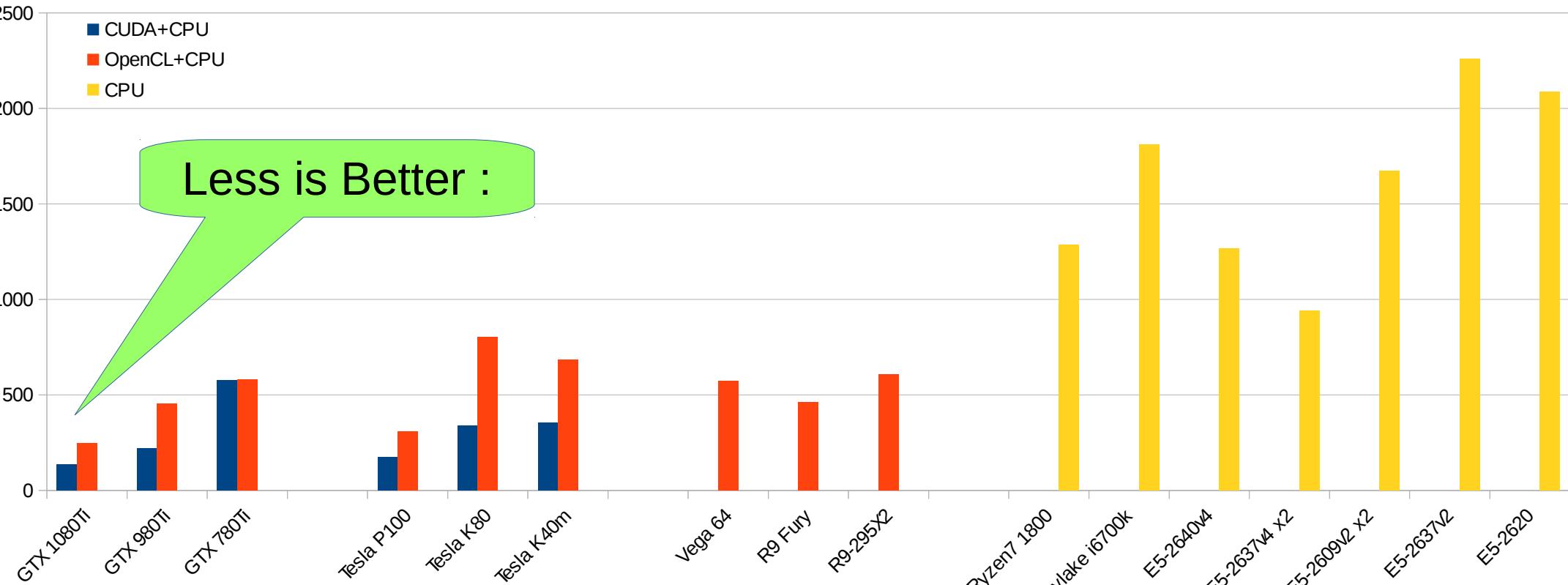
## D'abord pour les GPU...



- Les implémentations OpenCL sont 2x plus lentes
- Les GPU AMD sont au moins 4x plus lentes

# Cas d'usage #1 : test Nvidia

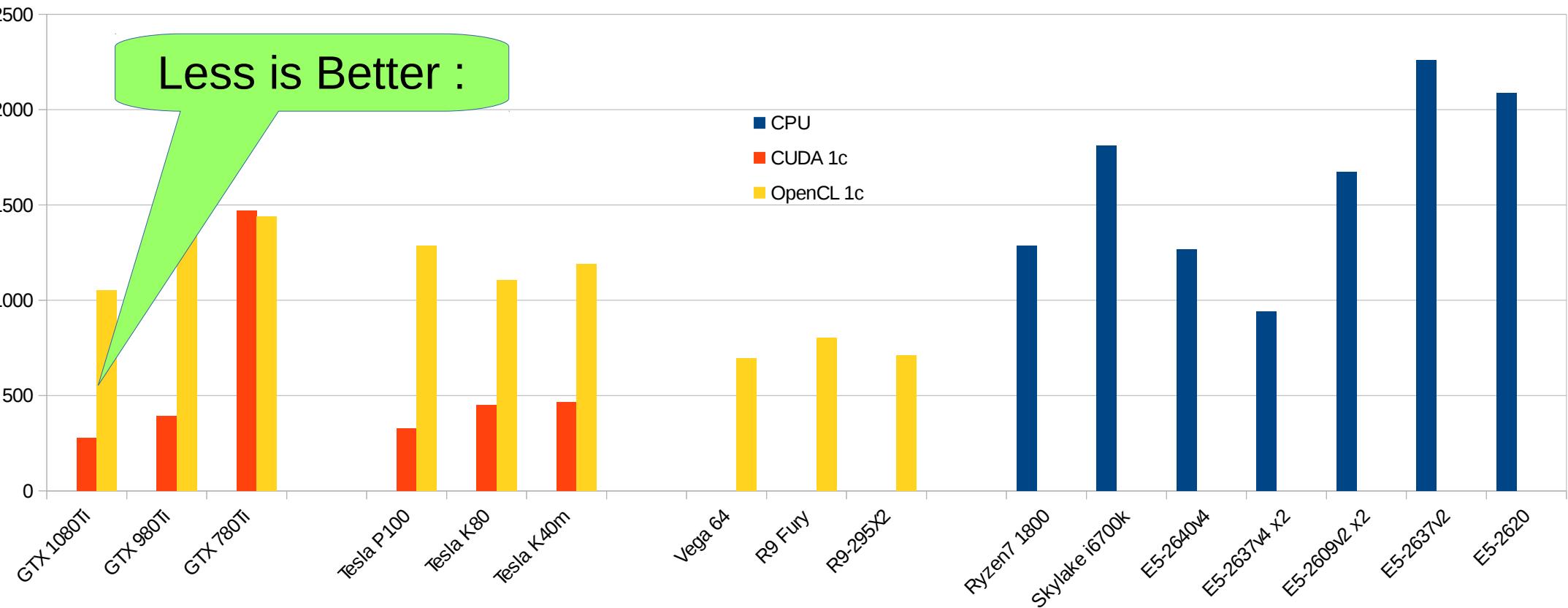
## Ensuite on rajoute les CPU



- Les processeurs sont 10x plus lents que le GPU le plus rapide
- Les GPU sont au pire 2x plus rapides que le CPU le plus rapide
- Escroquerie ! Le programme est hybride (CPU/GPU)

# Cas d'usage #1 : test Nvidia

## Renormaliser : GPU sur 1 cœur...



- Le GPU le plus puissant sur 1 cœur CPU est 4x plus rapide que le CPU
- Les GPU sont dans tous les cas plus rapides que les assemblages de CPU
- Donc, utiliser des GPU pour Gromacs, c'est pertinent...

# Gromacs : la conclusion

- GPU efficaces, MAIS dans un BON environnement
  - Le nombre de cœurs & la vitesse de la mémoire importants
- Implémentation CUDA toujours plus efficace
- Implémentation OpenCL en progrès
  - Mais 2x moins performante que la CUDA
- Exploitation de cartes de Gamer pertinente
  - Mais dans le cas de calculs en simple précision
- En gros, sans expérimentation, pas d'utilisation optimale...

**Mais où est passé le facteur 100 de la pub ?**

# 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 »

# 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 9 (GPU)GPU et 7 processeurs...



- E5-2680v4 2x : GTX1080Ti x14 en SP, Tesla P100 x16
- Oui, la puissance MxM est là, mais pour toute taille ?

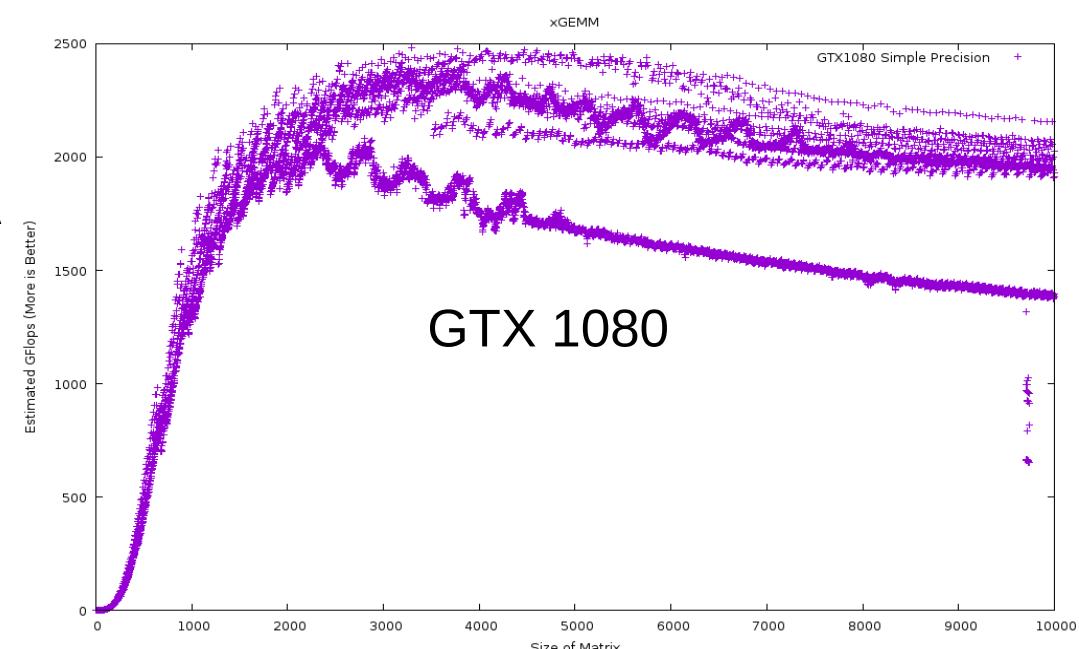
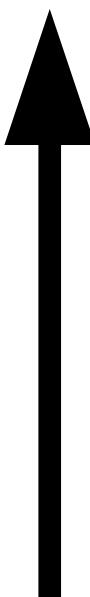
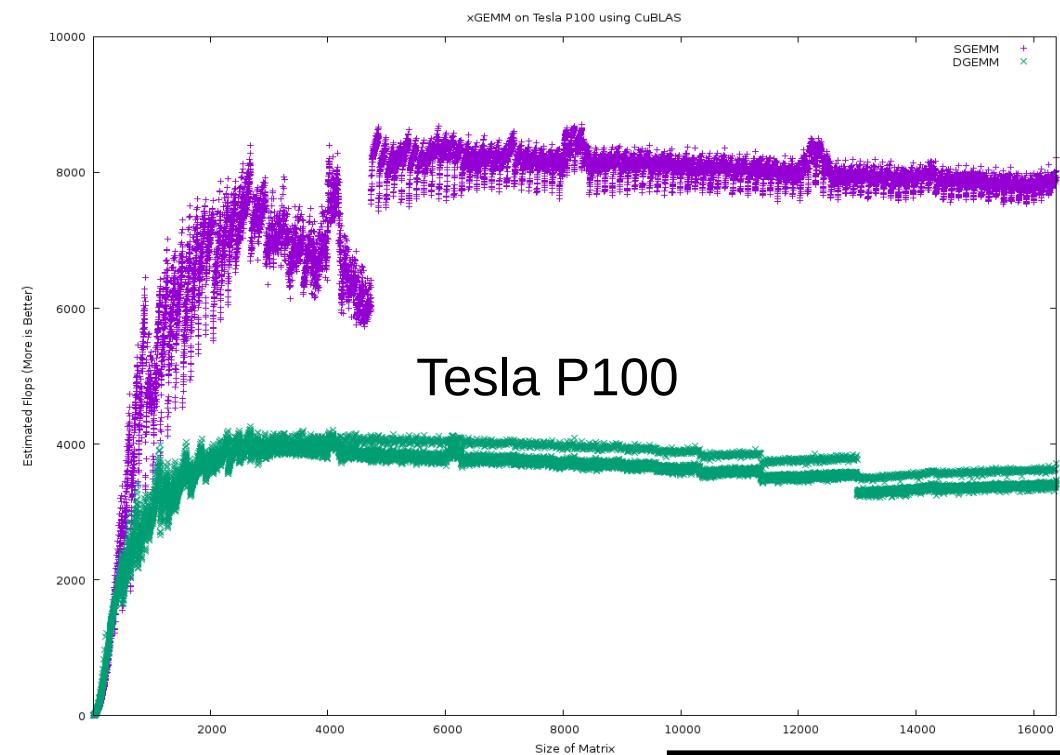
# Déjà en 2010, étrange...

## Le comportement Tesla C1060...

- Uniquement produit matriciel : xGEMM
- Propriété : Transposé ( $A * B$ )=Transposé( $A$ ) \* Transposé( $B$ )
- Résultats (en Gflops) : Yessss !
  - 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

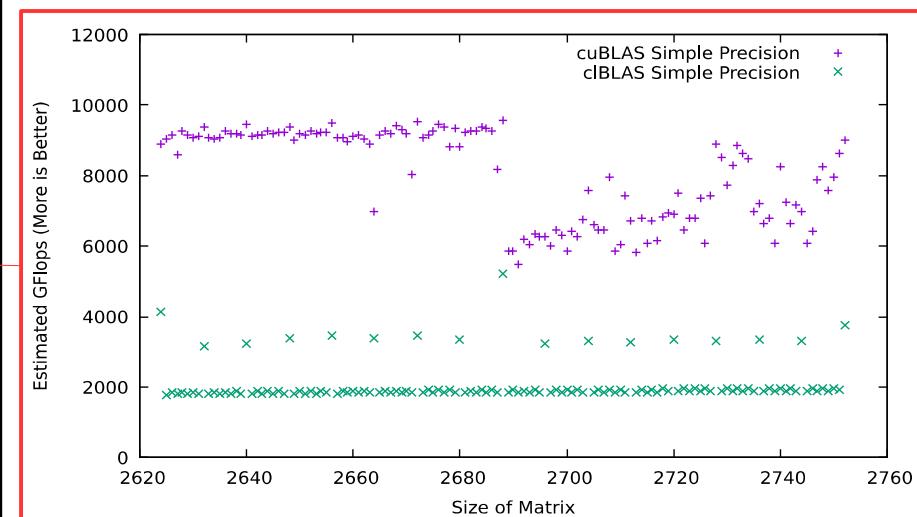
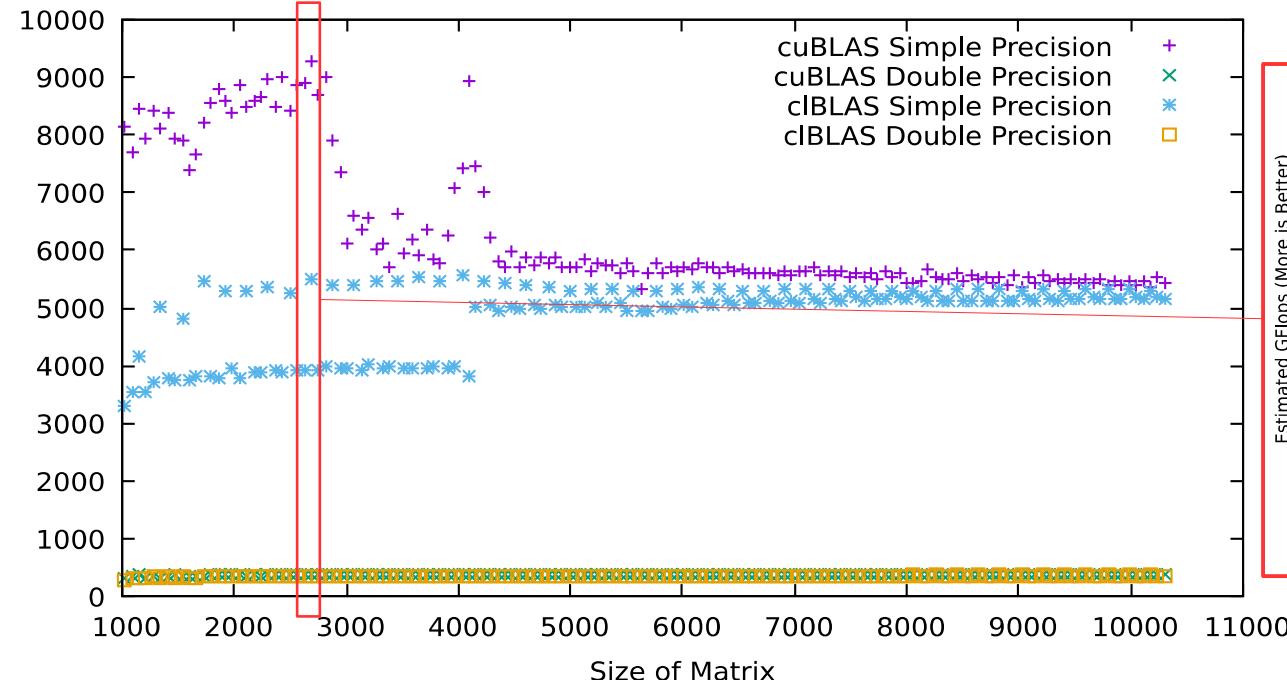


→ Taille

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

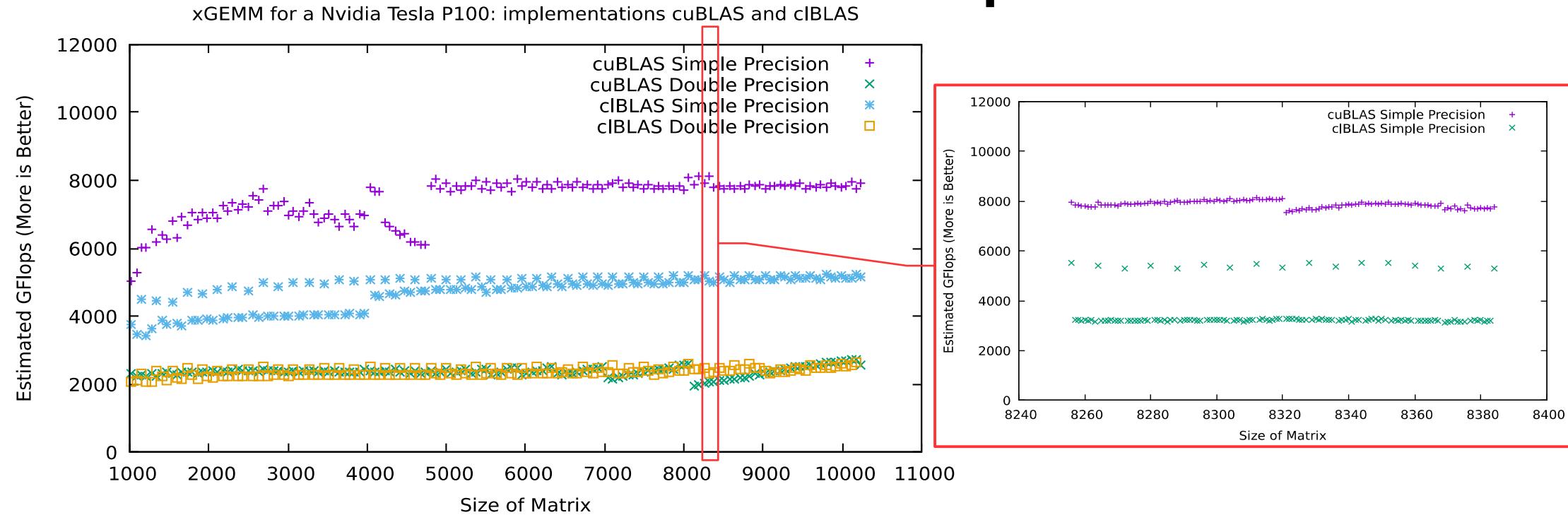
# Et pour la « bombe Gamer » Nvidia GTX 1080Ti, un max pour 2688

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



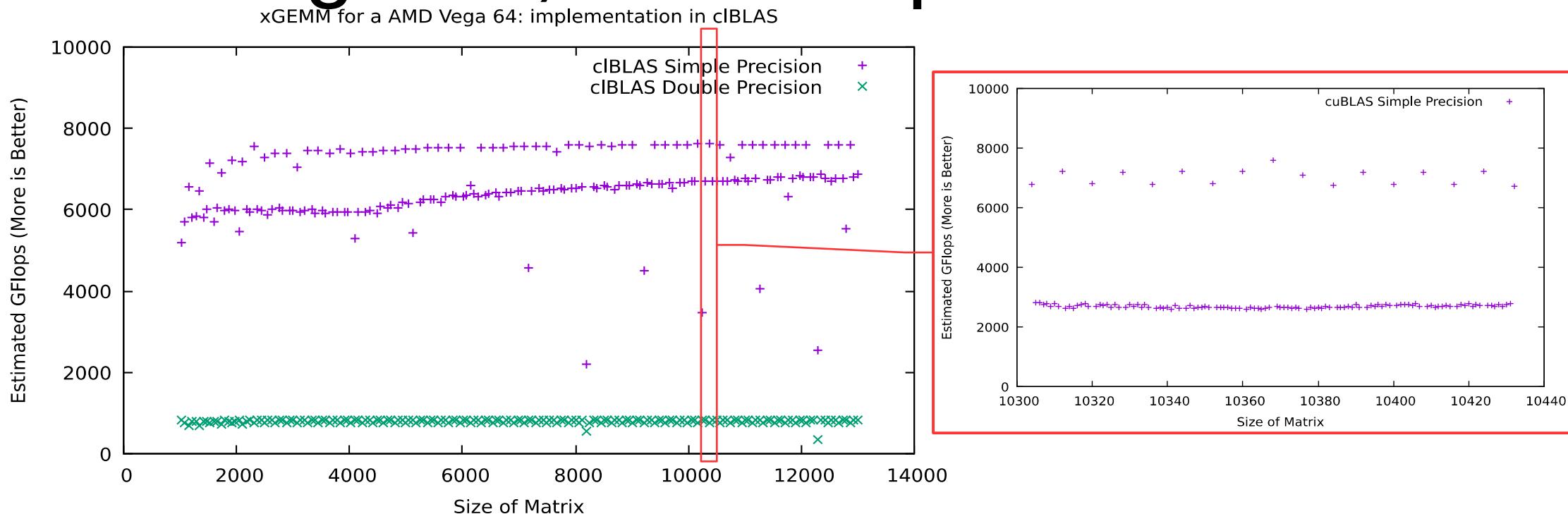
- Un facteur entre 20 et 30 entre simple & double précision
- Une implémentation clBLAS autour de la moitié pour les petites tailles, équivalente après
- Des effets de seuil (>2688 par exemple)
- Des effets arithmétiques (nombres premiers, par exemple 2671 ou 2713)

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



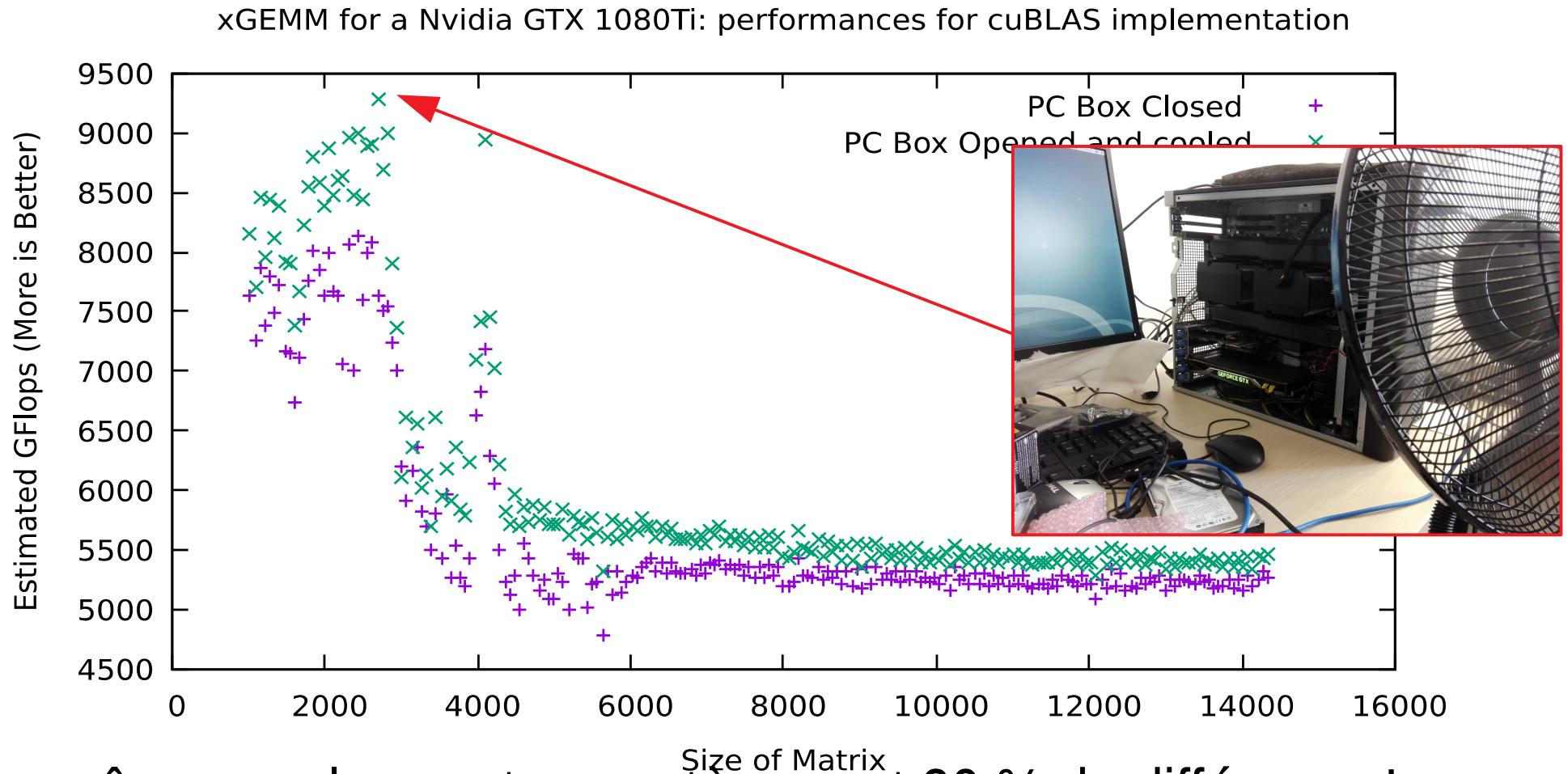
- Un facteur autour de 4 entre simple & double précision
- Une implémentation cIBLAS avec des périodicités étranges
- Quelques effets de seuil (>4096 par exemple)
- Moins d'effets arithmétiques

# Et pour la plus « grosse » AMD Vega 64, un max pour 10368



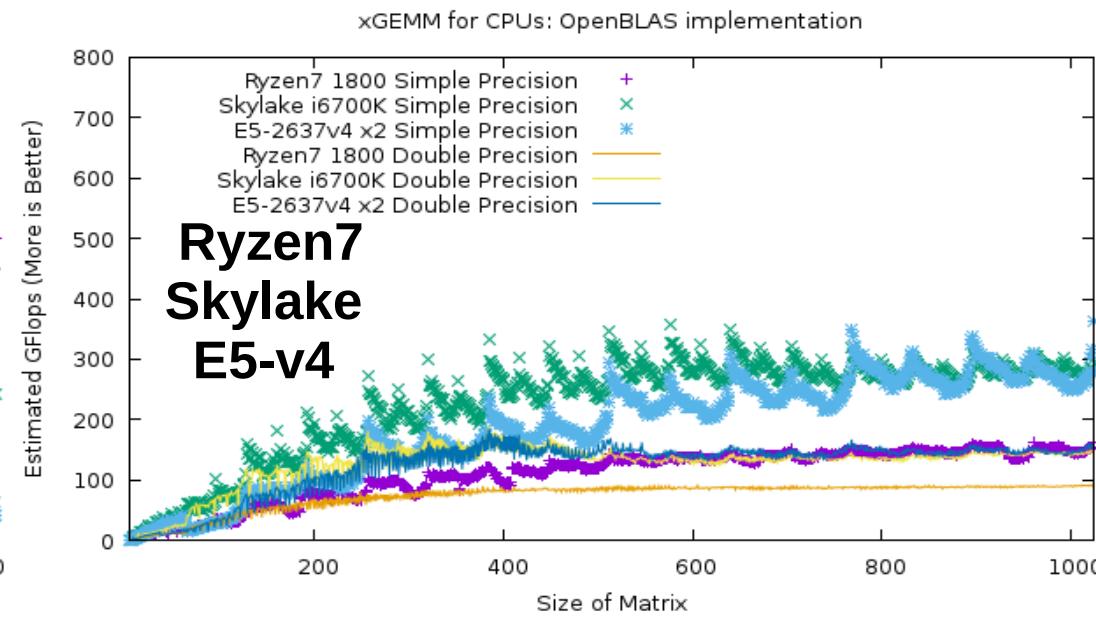
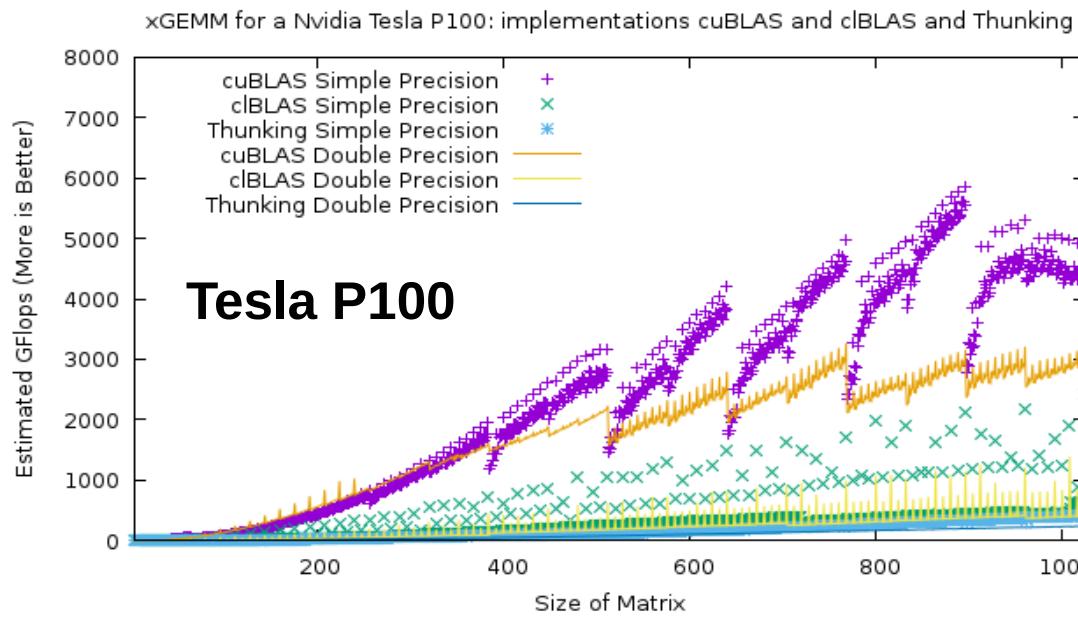
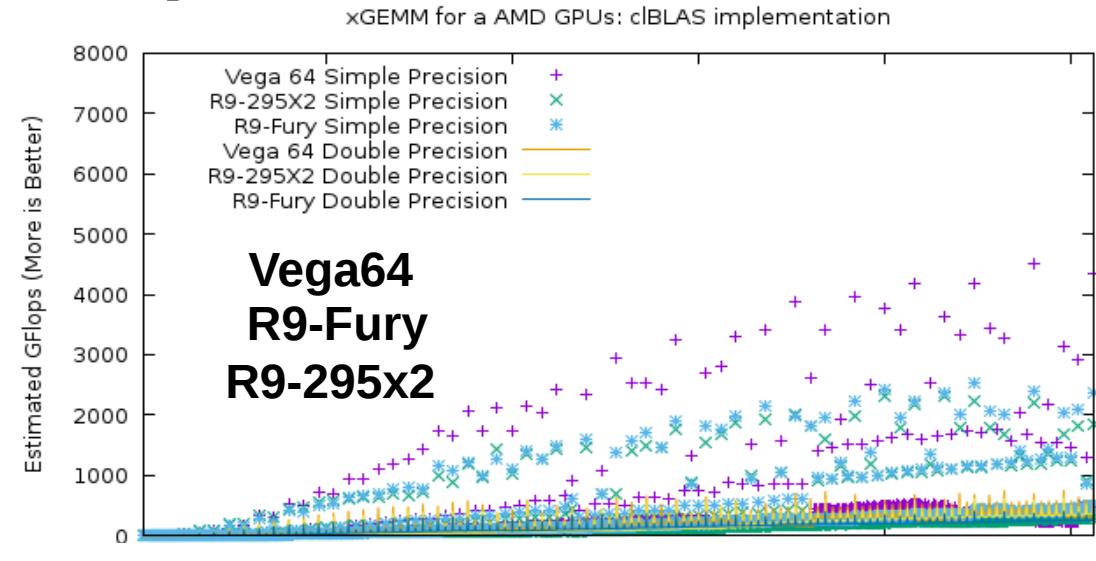
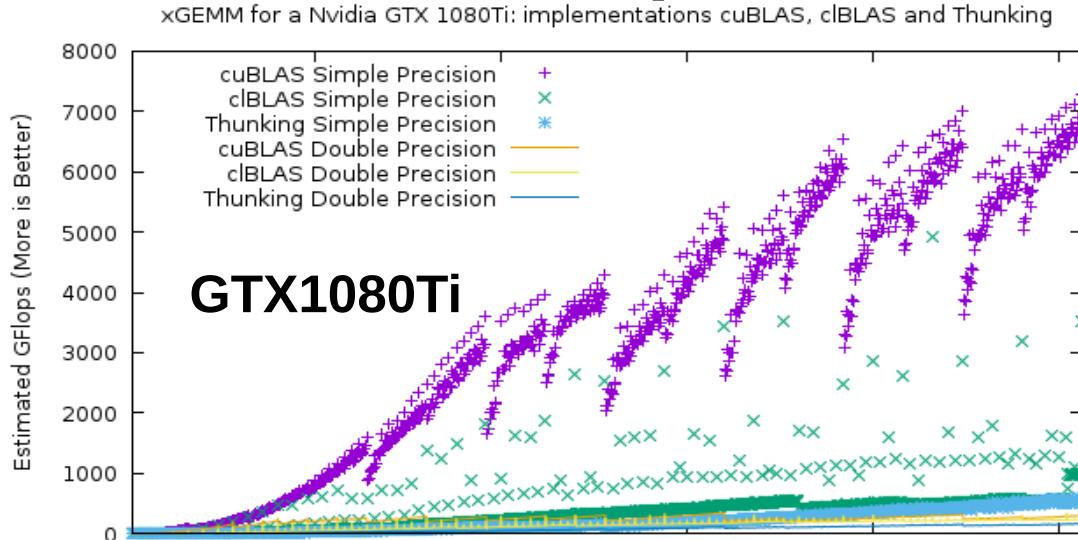
- Un facteur entre 4 et 10 (médiane à 8) entre simple & double précision
- Périodicité bimodale
- Moins d'effets arithmétiques

# 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  $C[i] = \text{somme normée de } y$  : 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  $A \cdot X$  qui donne Y
      - Je résous le système pour retrouver  $X'$  tel que :  $A \cdot X = Y$
      - 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

```
blas_dgemv(CblasRowMajor,CblasNoTrans,dim,dim,alpha,A,dim,X,incx,beta,Y,incx);  
blas_dtrsv(CblasRowMajor,CblasUpper,CblasNoTrans,CblasNonUnit, dim,A,dim,Y,incx);  
blas_daxpy(dim,beta2,Y,incx,X,incx);  
checksA[i]=(double)blas_dnrm2(dim,X,incx);  
blas_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

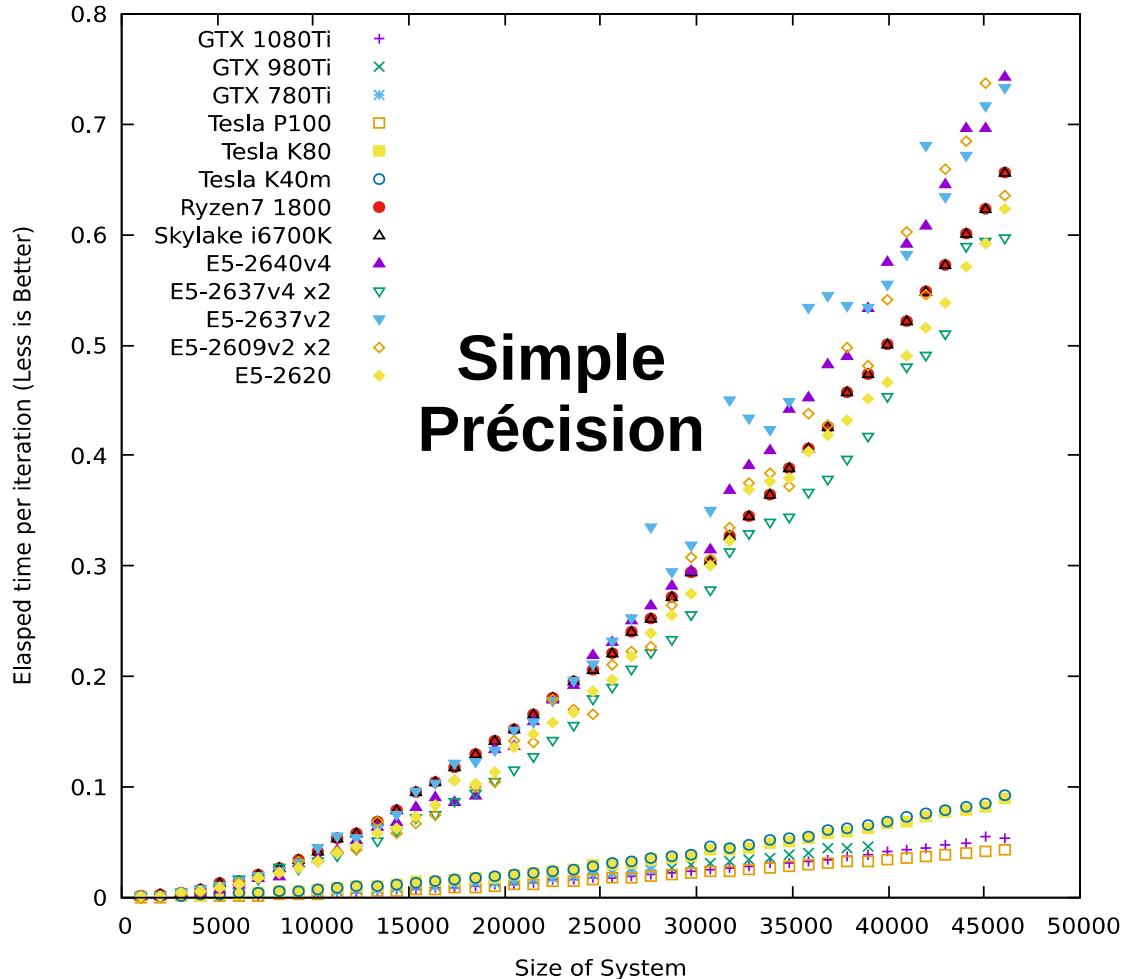
```
cublasDgemv(trans,dim,dim,alpha,devPtrA,dim,
            devPtrX,incx,beta,devPtrY,incx);
cublasDtrsv(uplo,trans,diag,dim,devPtrA,dim,
            devPtrY,incx);
cublasDaxpy(dim,beta2,devPtrY,incx,devPtrX,incx);
checksA[i]=(double)cublasDnrm2(dim,devPtrX,incx);
cublasDswap(dim,devPtrX,incx,devPtrY,incx);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

## Copie résultats, libération GPU

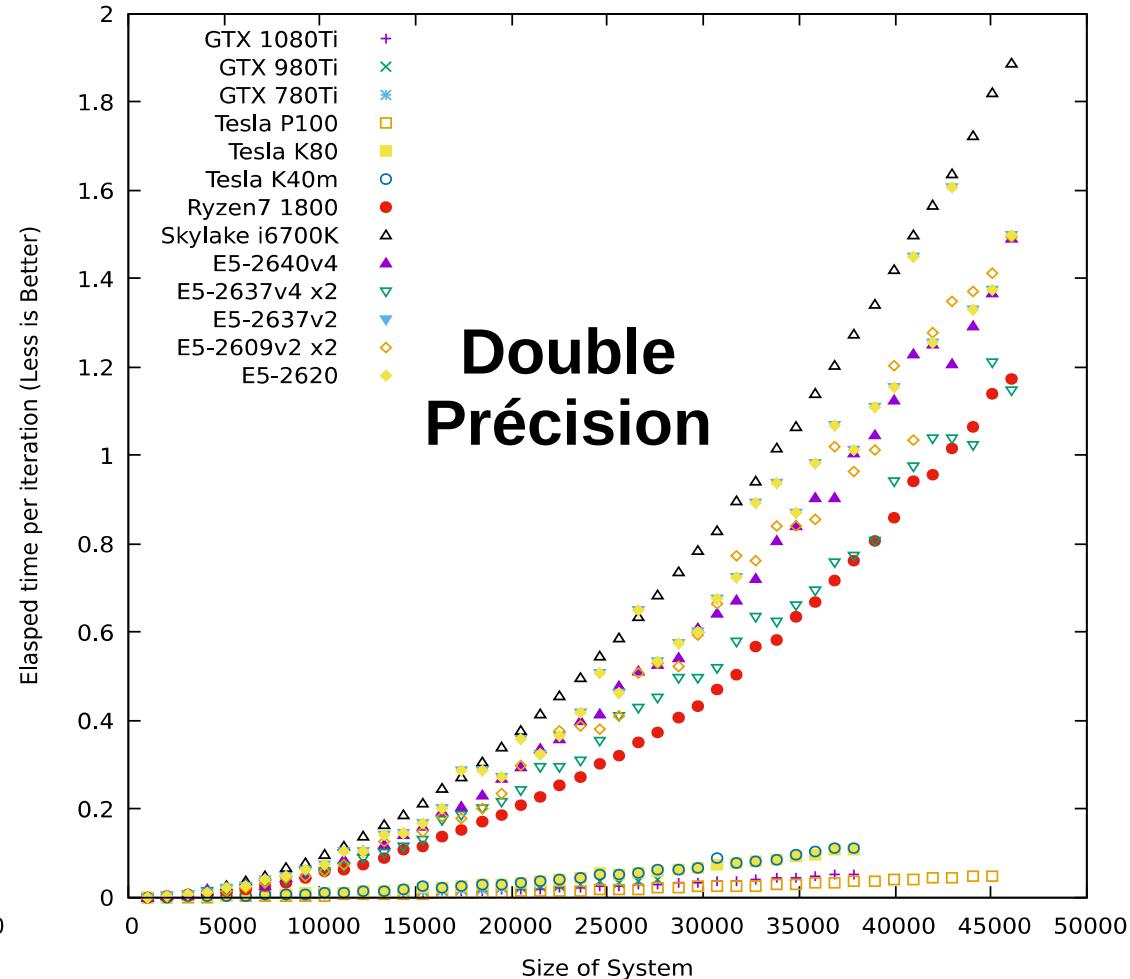
```
stat1=cublasGetVector(dim,sizeof(X[0]),devPtrX,
                      incx,X,incx);
stat2=cublasGetVector(dim,sizeof(Y[0]),devPtrY,
                      incx,Y,incx);
cublasFree (devPtrA);
cublasFree (devPtrX);
cublasFree (devPtrY);
if ((stat1 != CUBLAS_STATUS_SUCCESS) ||
    (stat2 != CUBLAS_STATUS_SUCCESS))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

# Les premiers résultats... En temps écoulé par itération

BLAS bench in Simple Precision : native cuBLAS and OpenBLAS



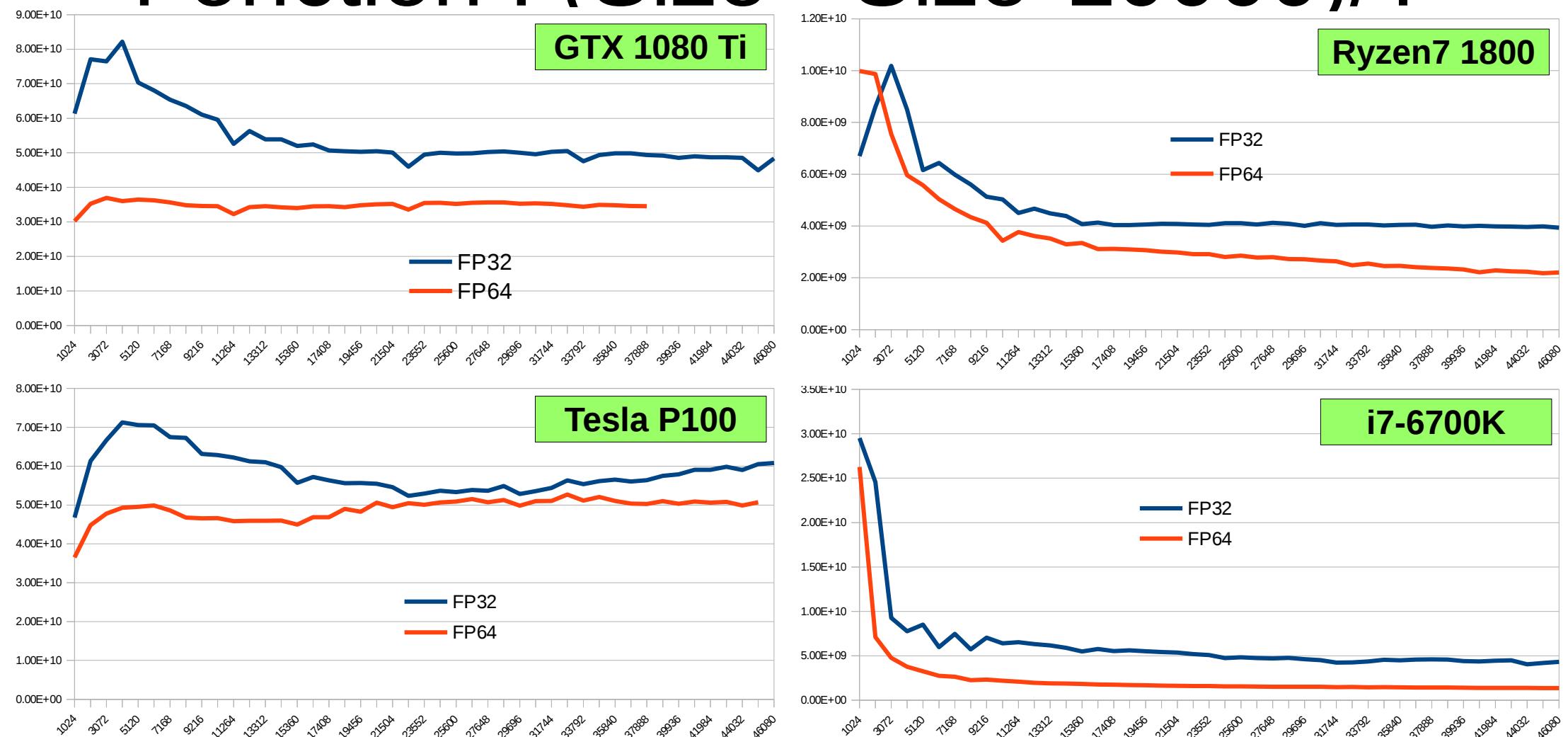
BLAS bench in Double Precision : native cuBLAS and OpenBLAS



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

# Résultats « bench BLAS »

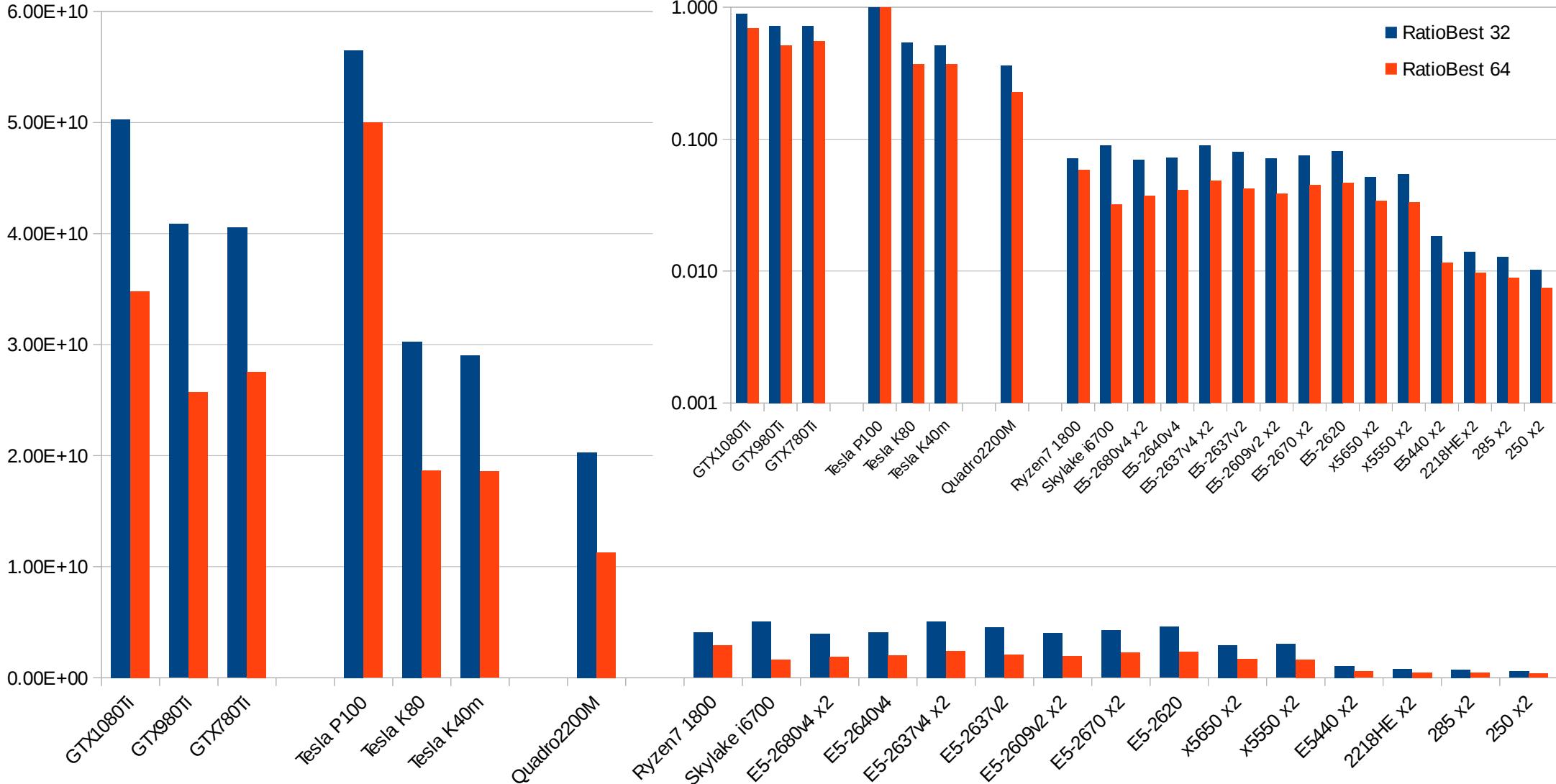
## Fonction : $(\text{Size}^2 + \text{Size} * 1000) / T$



- 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 16GB pour l'instant
- Préconisation : construire 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**
  - Un code « grain mémoire » : **Splutter**
- Deux buts : comparer GPU vs CPU et GPU vs GPU

# Qui utilise OpenCL ?

## Dans les applications

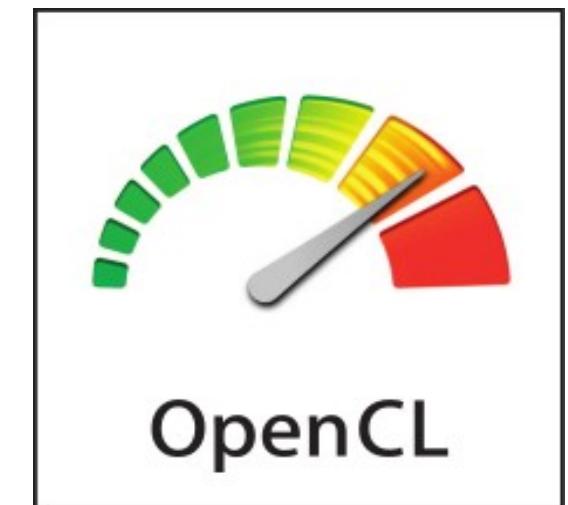
- OS : Apple dans MacOSX
- « Grosses » applications :
  - Libreoffice
  - Ffmpeg
- Applications connues
  - Les Graphiques : photoscan,
  - Les Scientifiques : OpenMM, Gromacs, Lammps, Amber, ...
- Des centaines de logiciels, librairies ! Mais un test indispensable...
  - <https://www.khronos.org/opencl/resources/opencl-applications-using-opencl>

# 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

# 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  $n$  :

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

The End



# 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] = {35,51,54,58,55,32,36,69,27,39,35,10,16,44,55,14,58,75,18,15};

#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

```
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)
```

Noyau OpenCL

```
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 ?

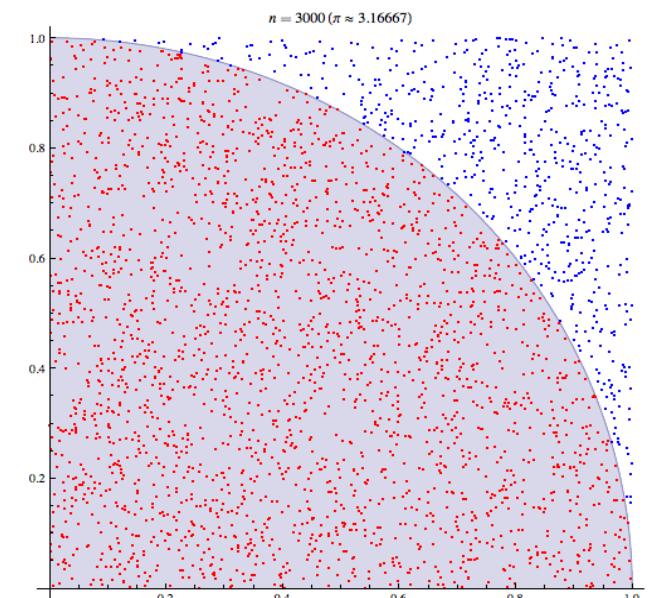
## « Hello World » C/Python : la pesée

- Sur les implémentations OpenCL précédentes :
- On previous OpenCL implementations :
  - En C :  lignes,  mots,  bytes
  - En Python :  lignes,  mots,  bytes
  - Factors : , ,  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

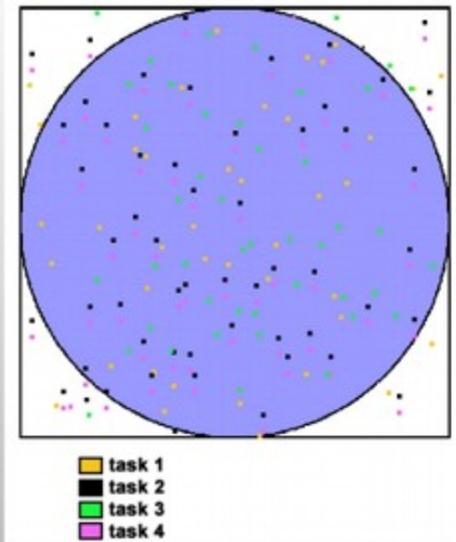
Lawrence Livermore National Laboratory



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



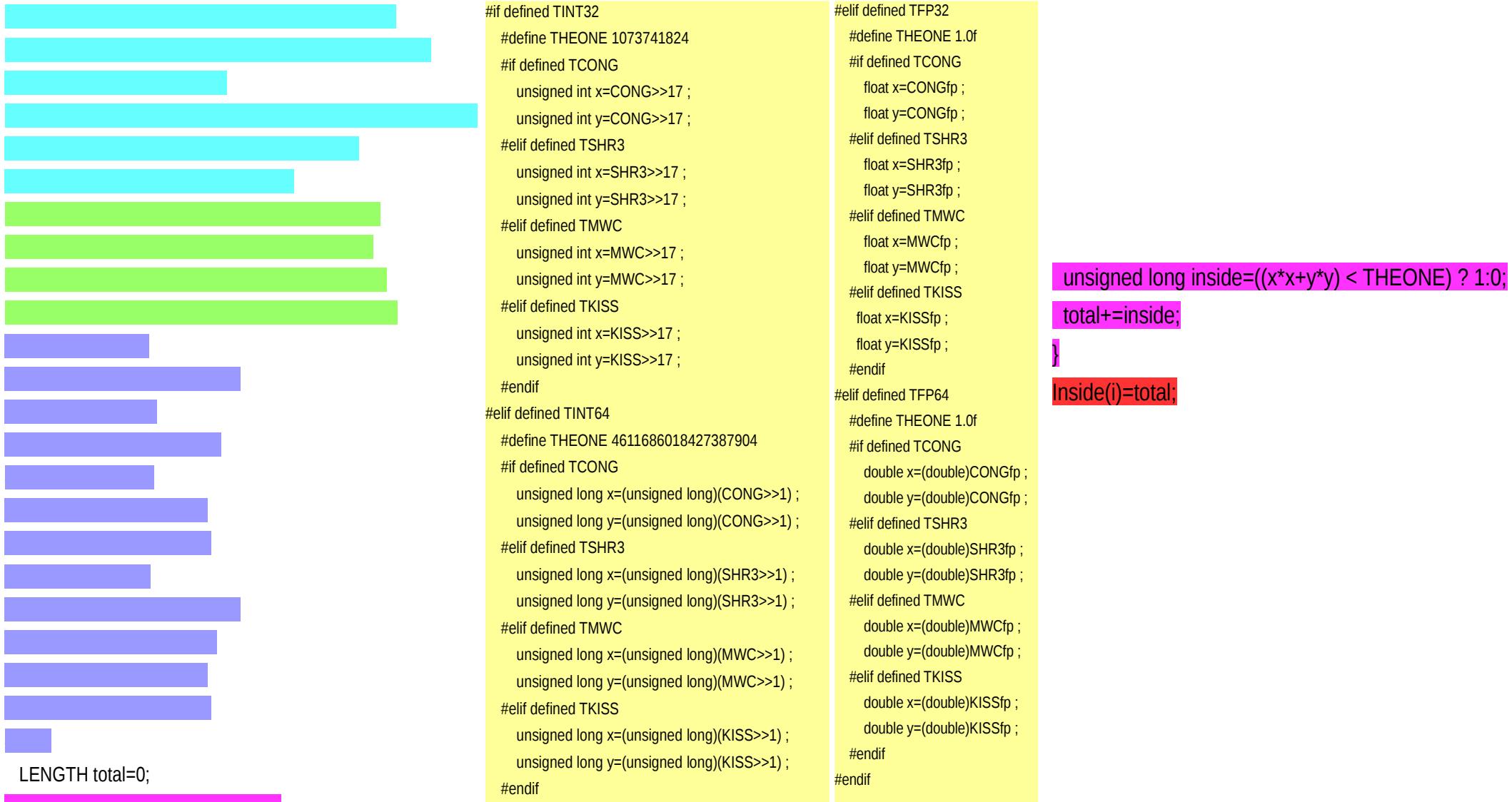
Lawrence Livermore National Laboratory

LLNL-PRES-XXXXXX

# D'une version très monolithique... Variable & RNG fixés...



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



# Différences entre CUDA/OpenCL

## Les noyaux des calculs GPU

```
__device__ ulong MainLoop(ulong iterations,uint seed_w,uint seed_z,size_t work)
{
    ulong total=0;
    for (int i=0;i<iterations;i++)
        total+=seed_z;
    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)
{
    ulong total=0;
    for (int i=0;i<iterations;i++)
        total+=seed_z;
    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 ?

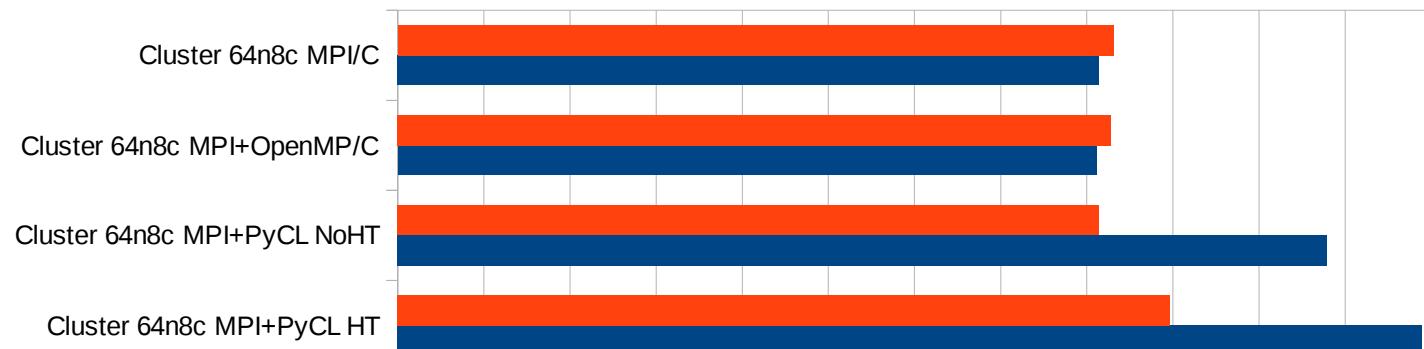
## Très simple implémentation...

- 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); }`

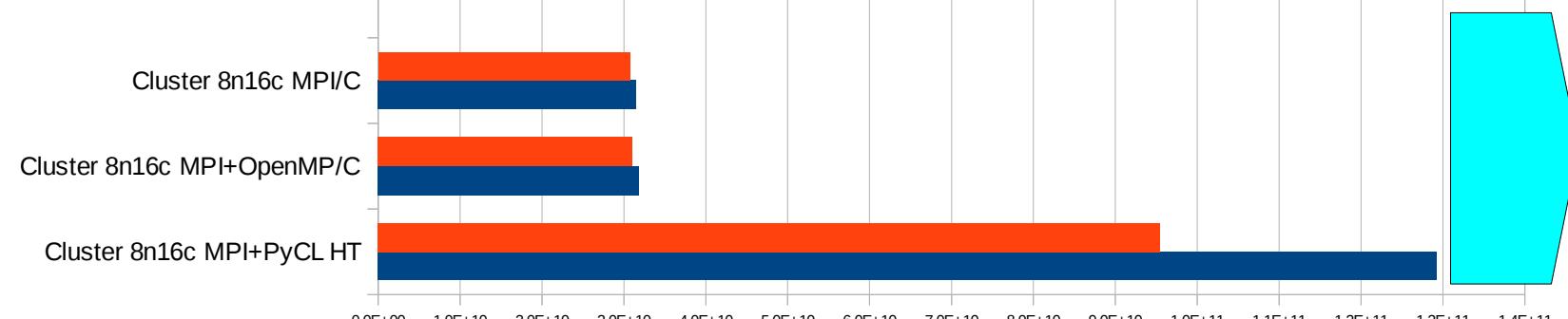
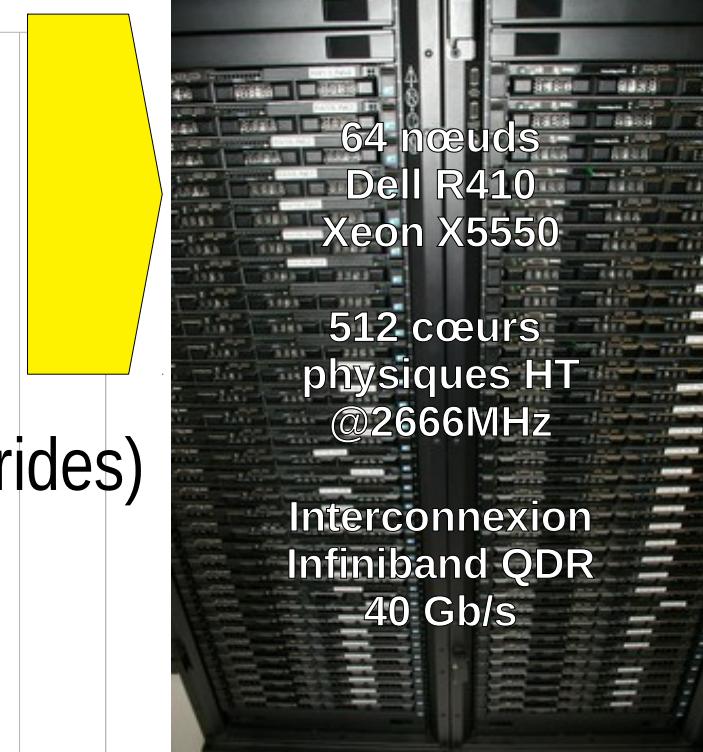
# Et en KOKKOS, ça donne quoi ?

- 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

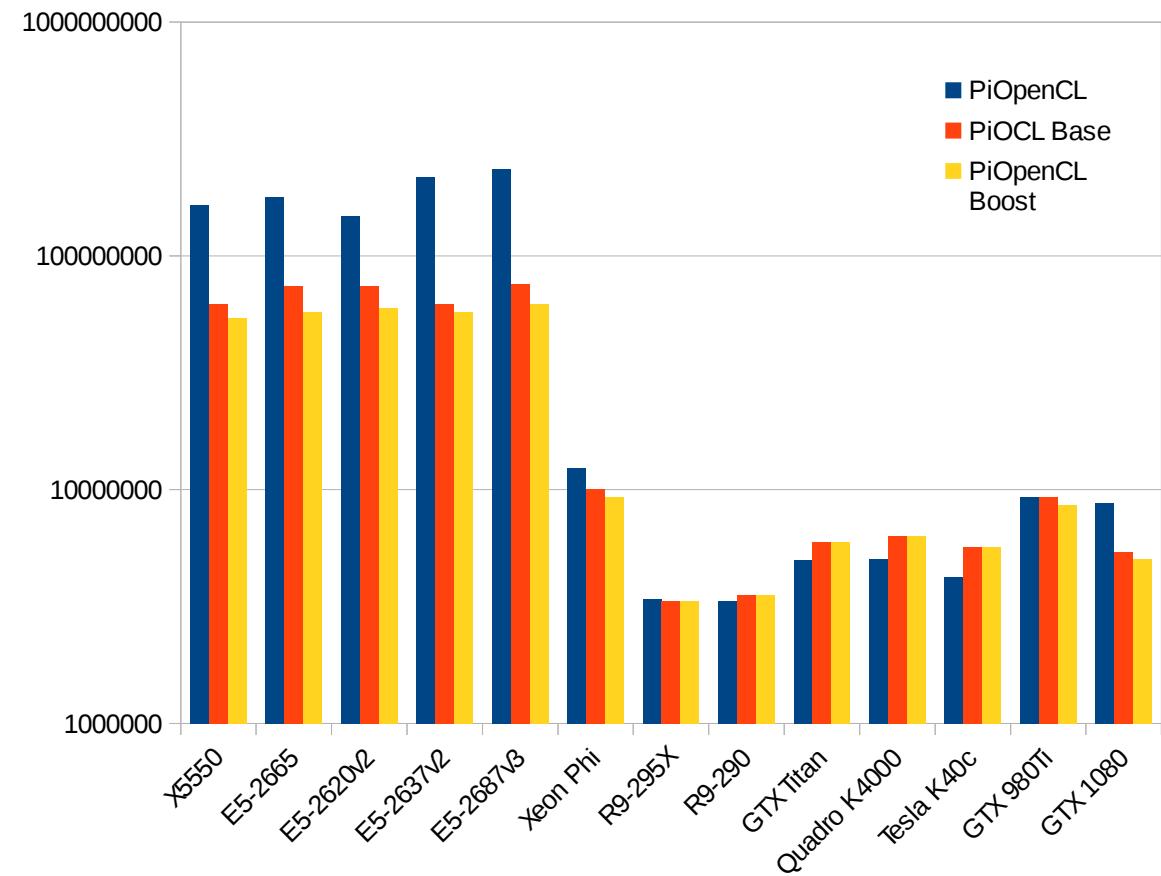
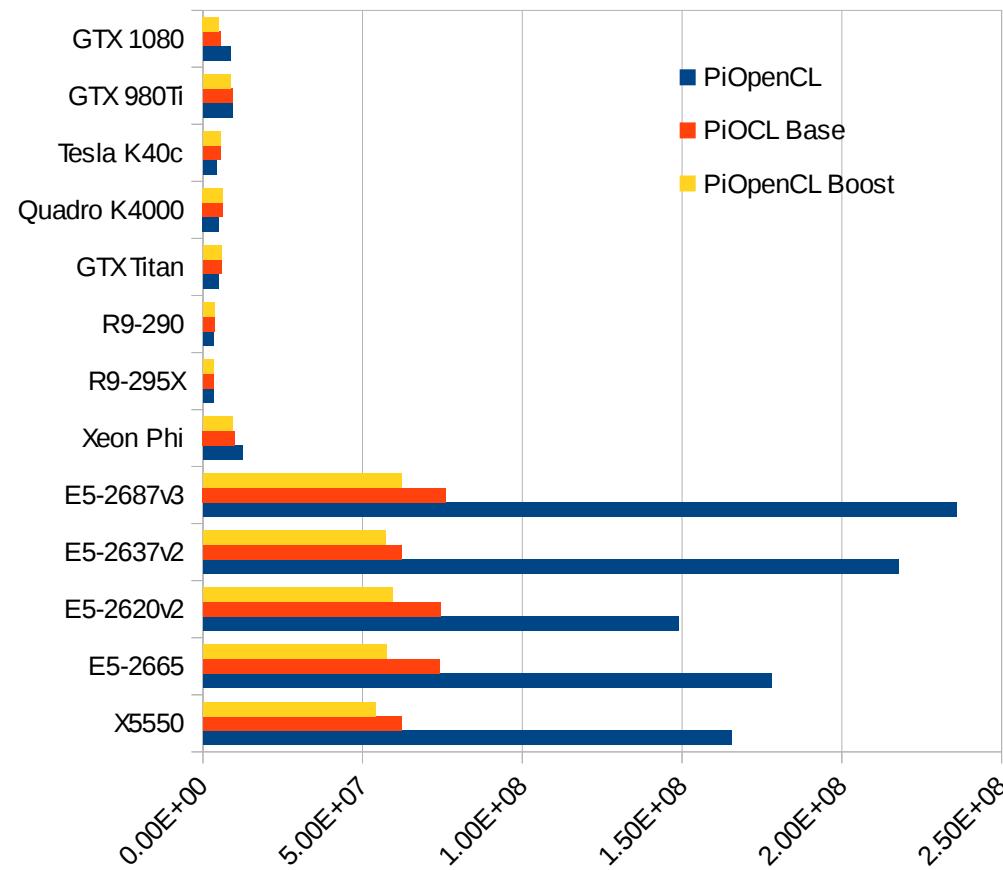


- Code Pi Dart Dash : 3 implémentations (2 hybrides)
- Cluster #1 : 64 nœuds avec 512 cœurs Gen
- Cluster #2 : 8 nœuds avec 128 cœurs Gen+5



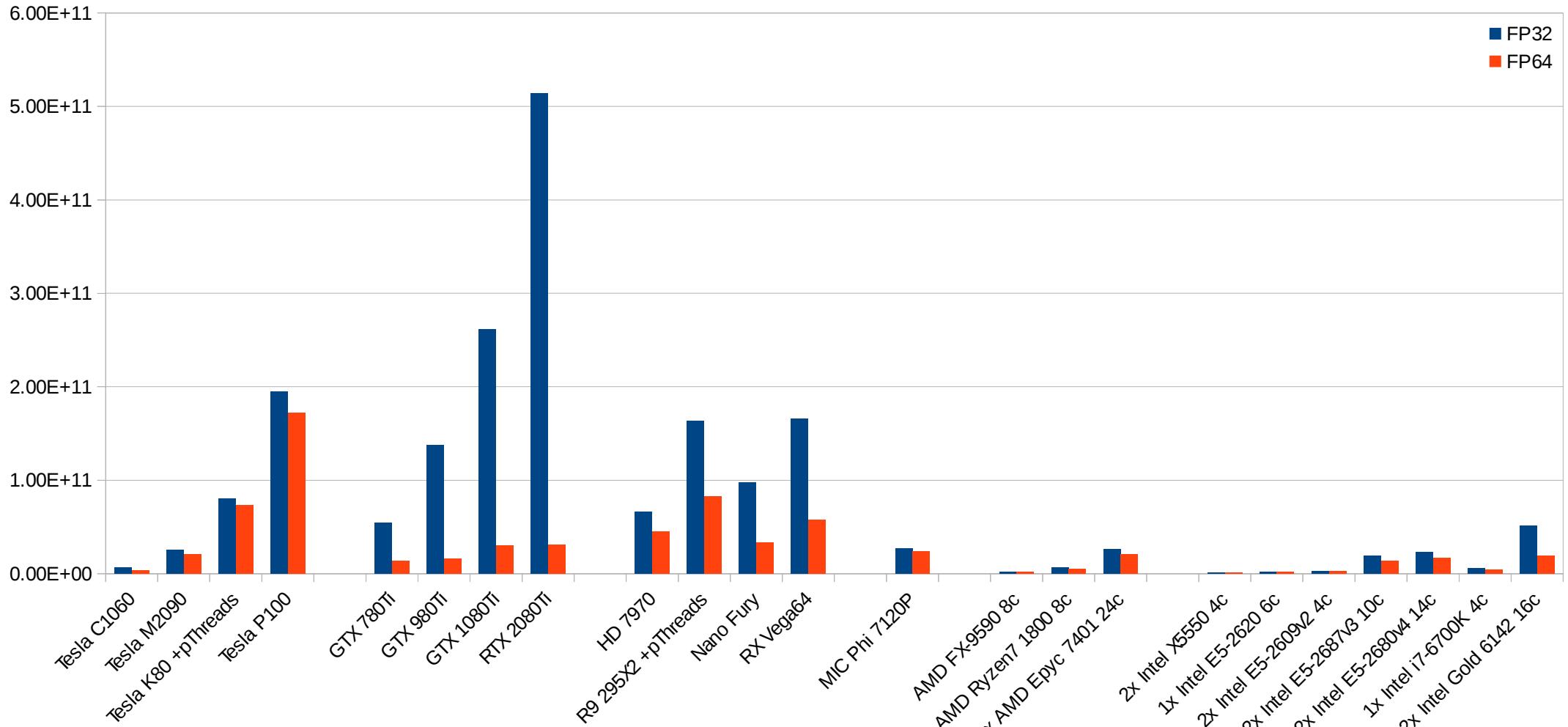
# Pour 1 de PR, bas régime, pour Pi Le CPU explose le GPU!

- De 20 à 50x plus lent !



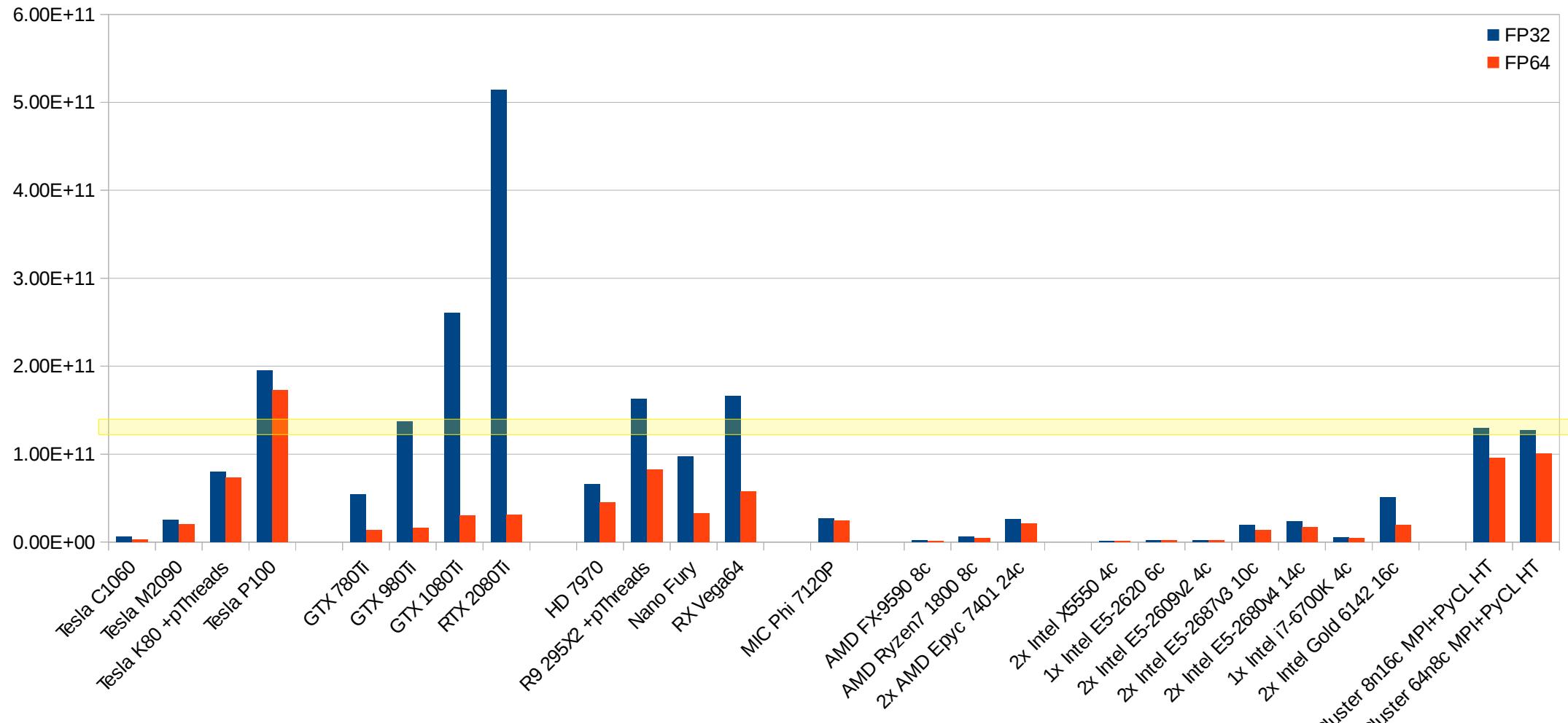
- 1.5 ordre de magnitude

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



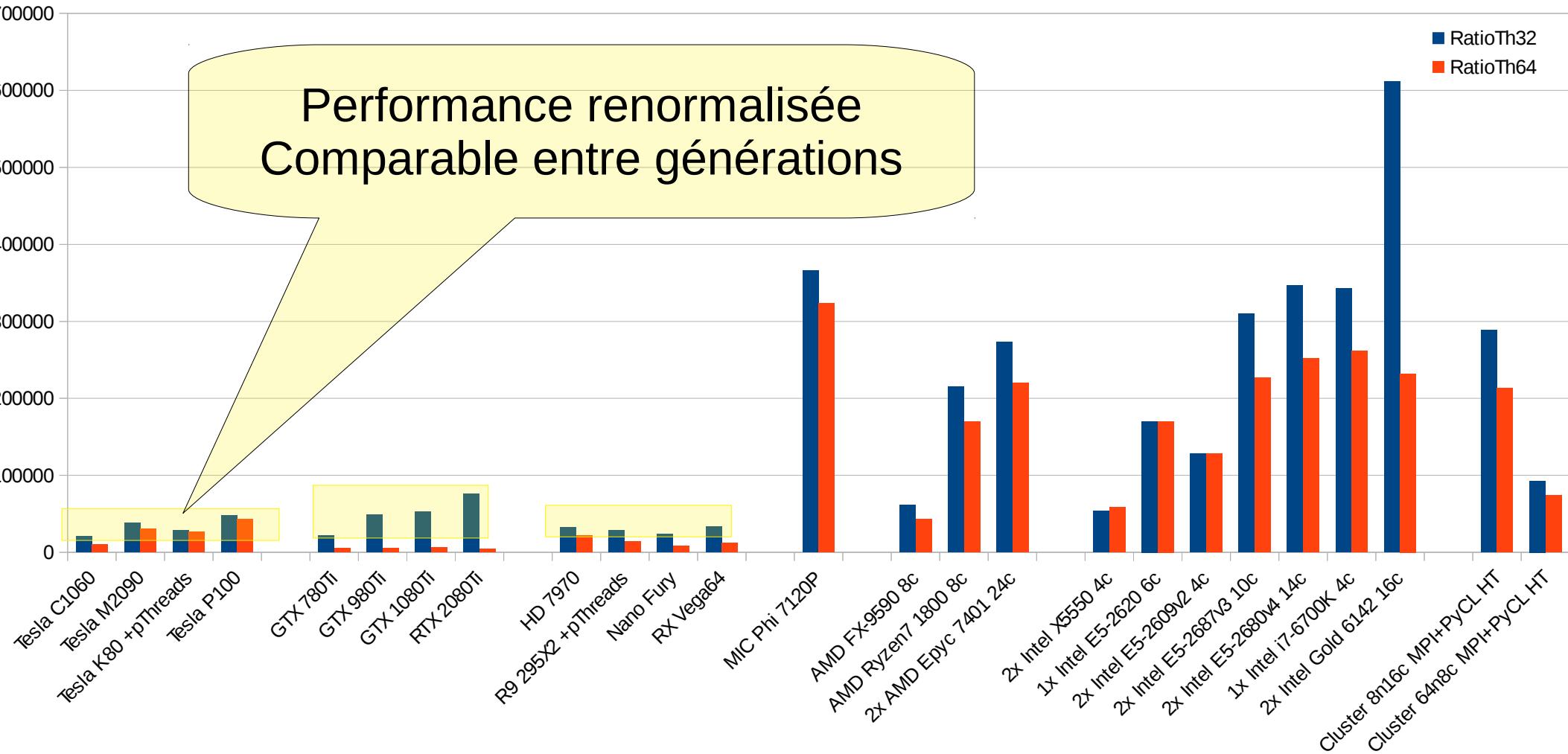
Mais en intégrant les clusters, ça donne quoi ?

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



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

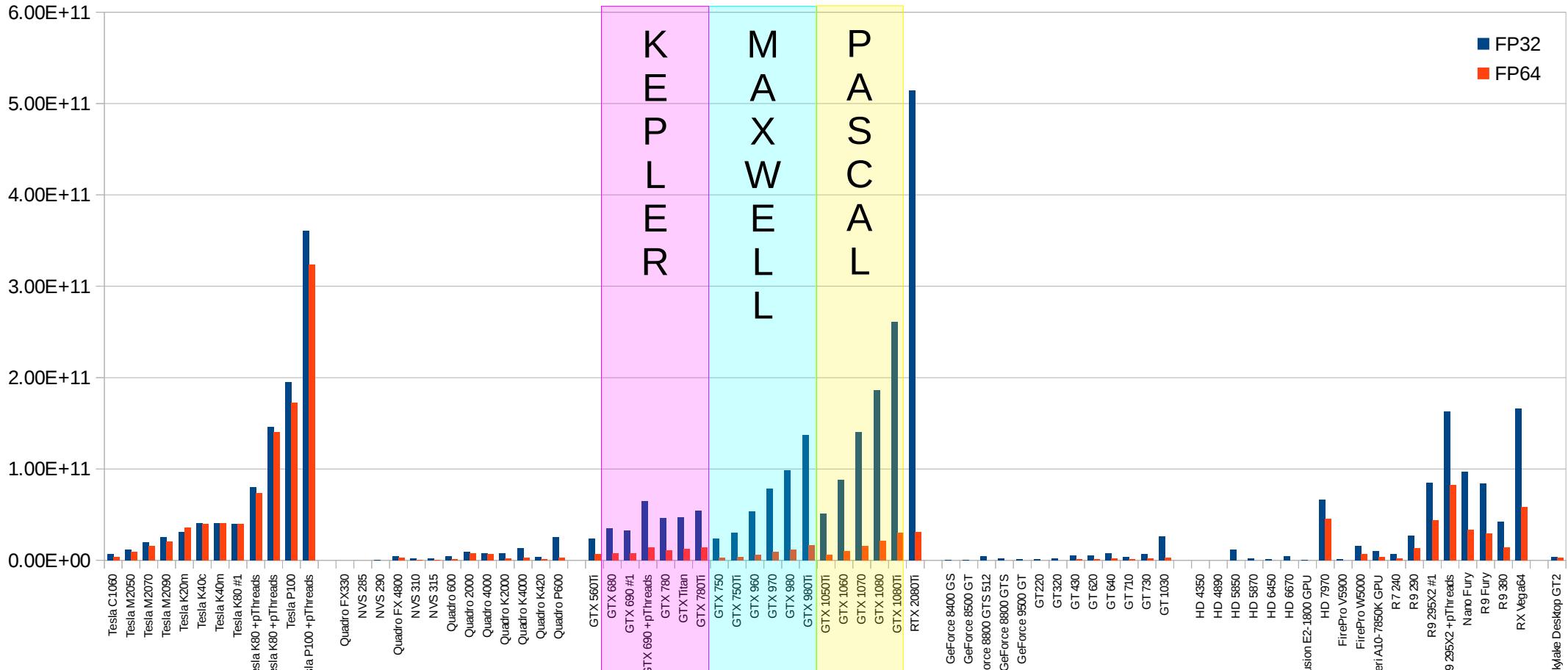
# Le rapport à la puissance « théorique » ? Itops sur coeurs\*fréquence



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

# Sur le bestiaire du CBP

## De génération en génération

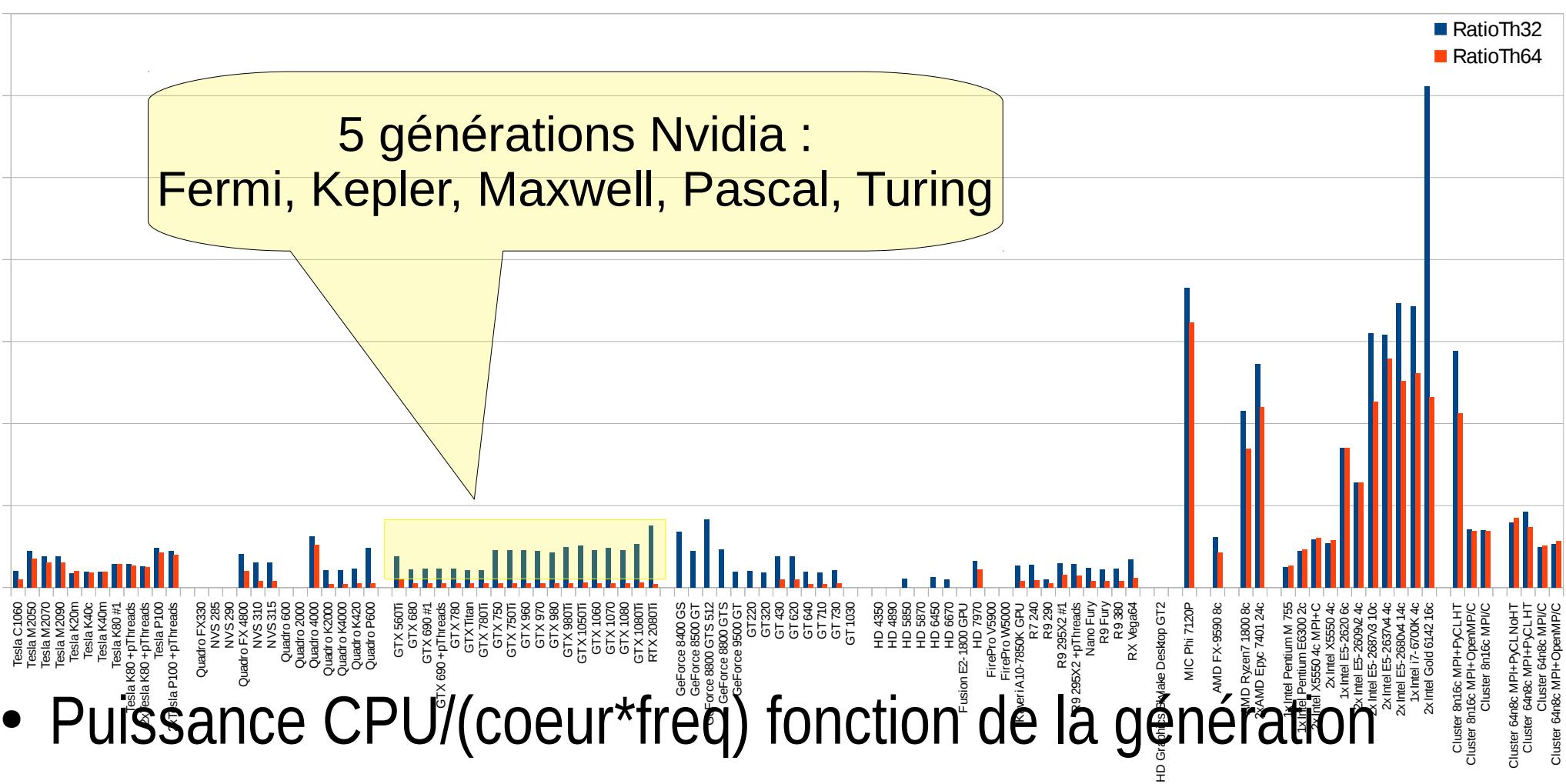


A chaque génération, toute une gamme de performances !



# Puissance renormalisée CPU/GPU Assertion généralisable ?

5 générations Nvidia :  
Fermi, Kepler, Maxwell, Pascal, Turing

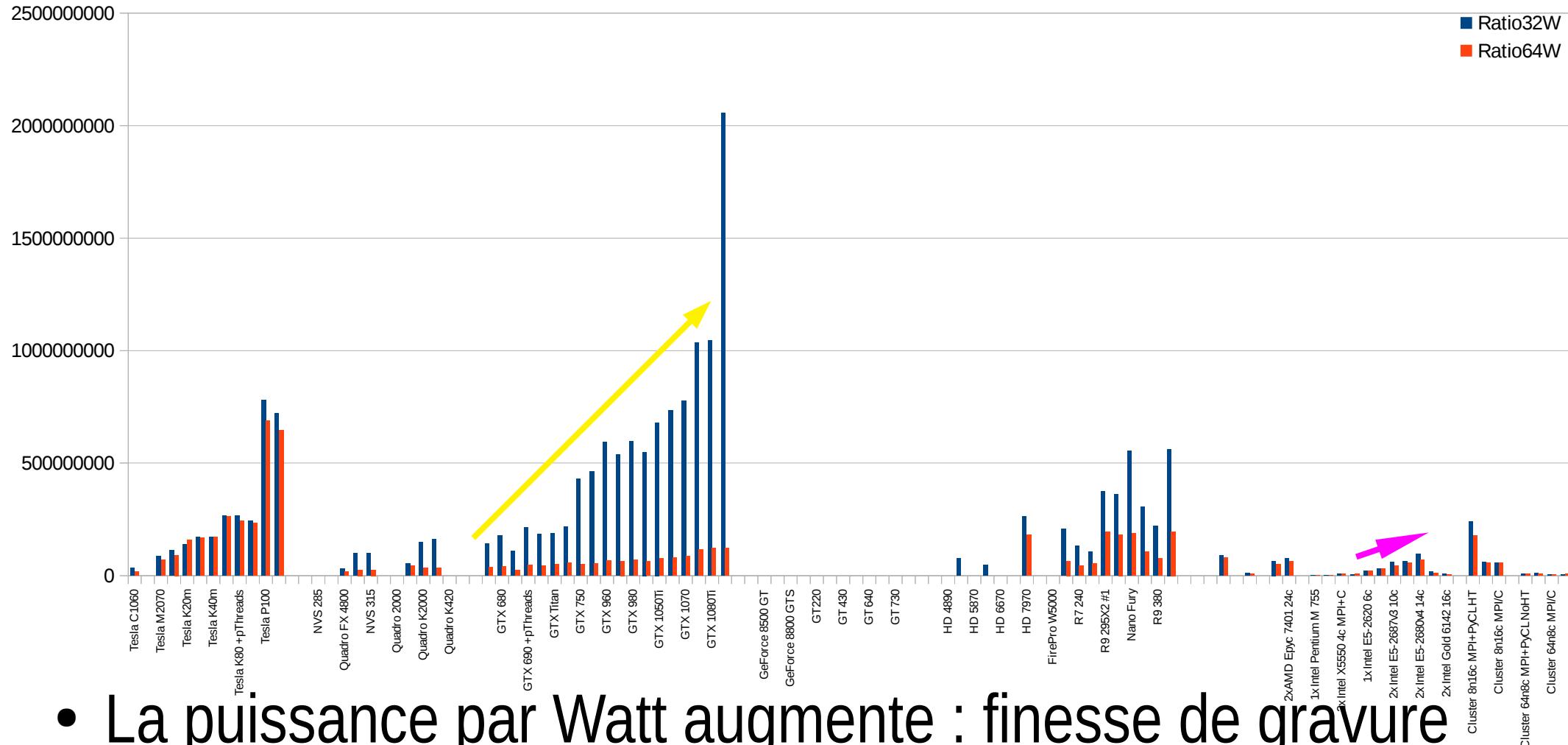


- Puissance CPU/(coeur\*freq) fonction de la génération
- Puissance GPU/(coeur\*freq) indépendante...



# La puissance pour l'écogiste

## Renormalisation à la TDP

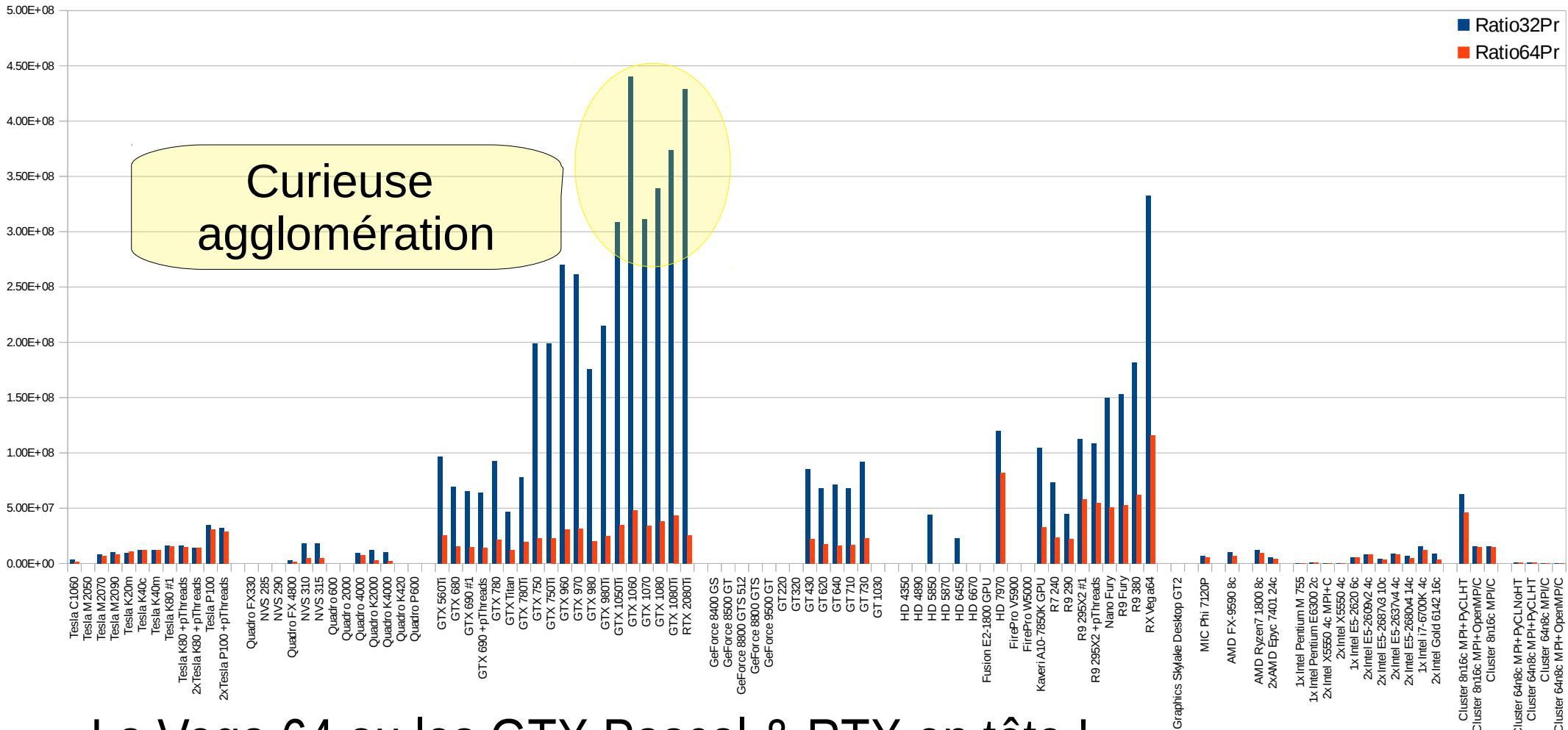


- La puissance par Watt augmente : finesse de gravure
- GPU : de 55 nm à 14 nm ; CPU : de 65 nm à 14 nm



# La puissance pour le financier

## Renormalisation à la MSRP

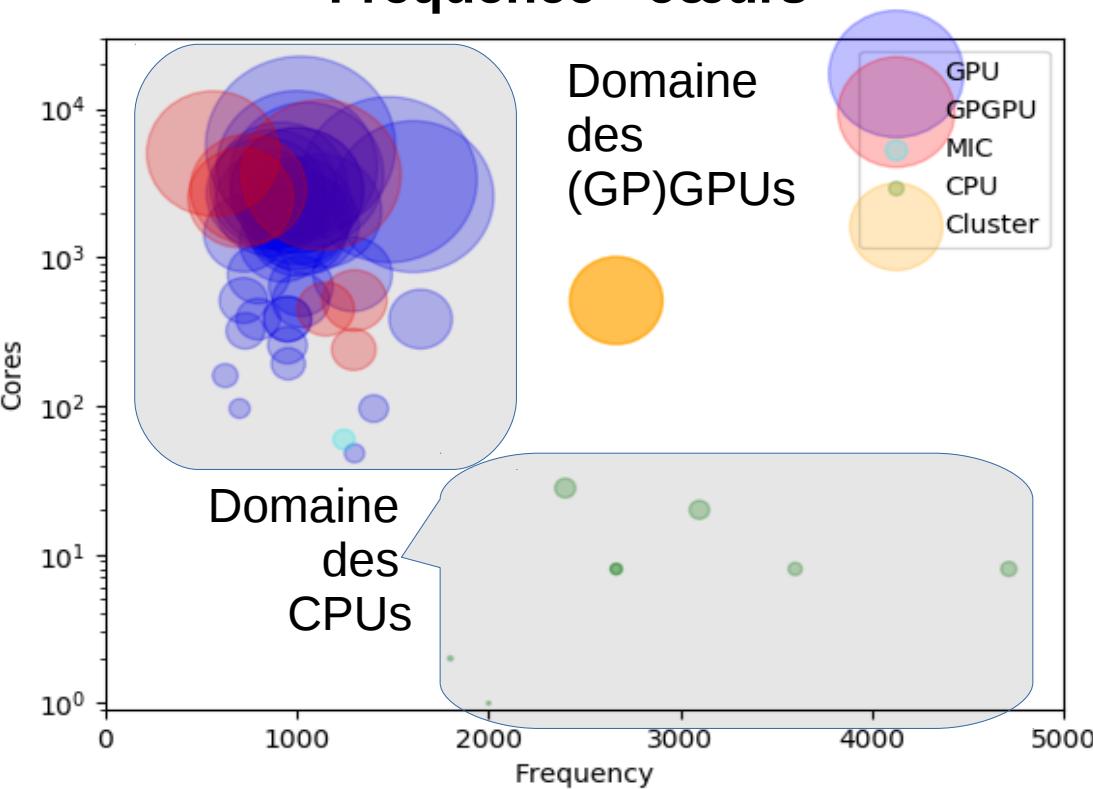


- La Vega 64 ou les GTX Pascal & RTX en tête !
- A croire qu'ils utilisent ce « test » pour leurs prix ;-)

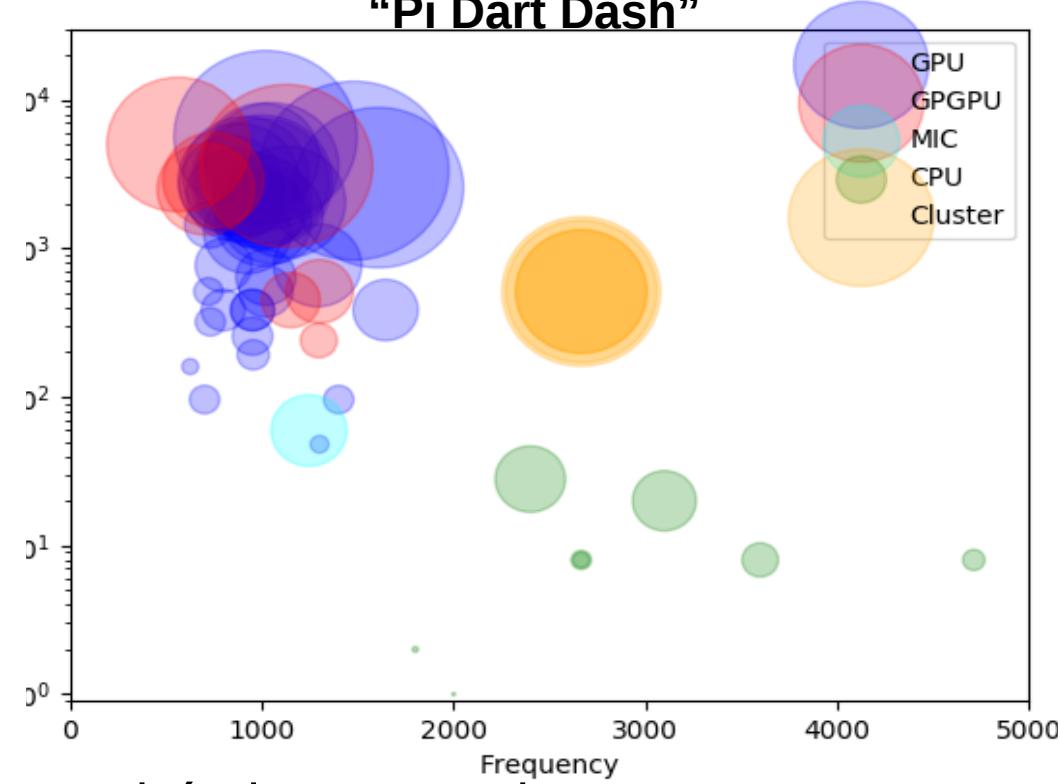


# Représenter les performances... Question de battement, de cœurs ?

Performance Théorique  
Fréquence \* cœurs



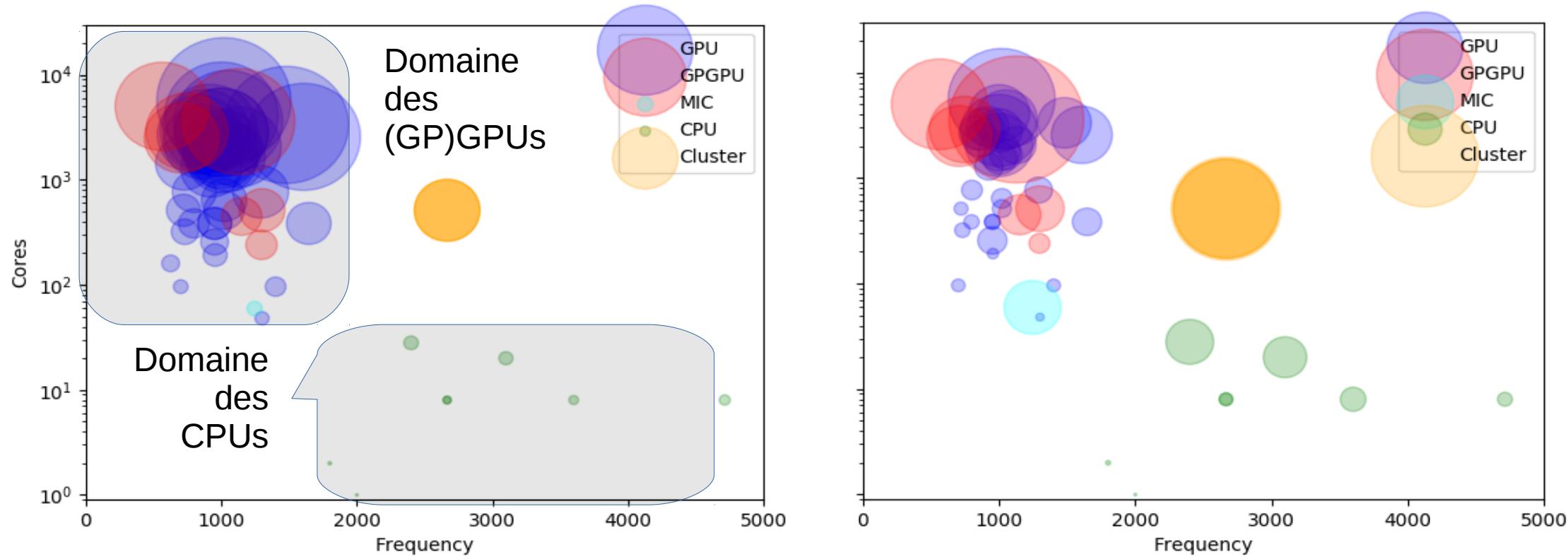
Performance en 32 bits  
“Pi Dart Dash”



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

# La performance en informatique

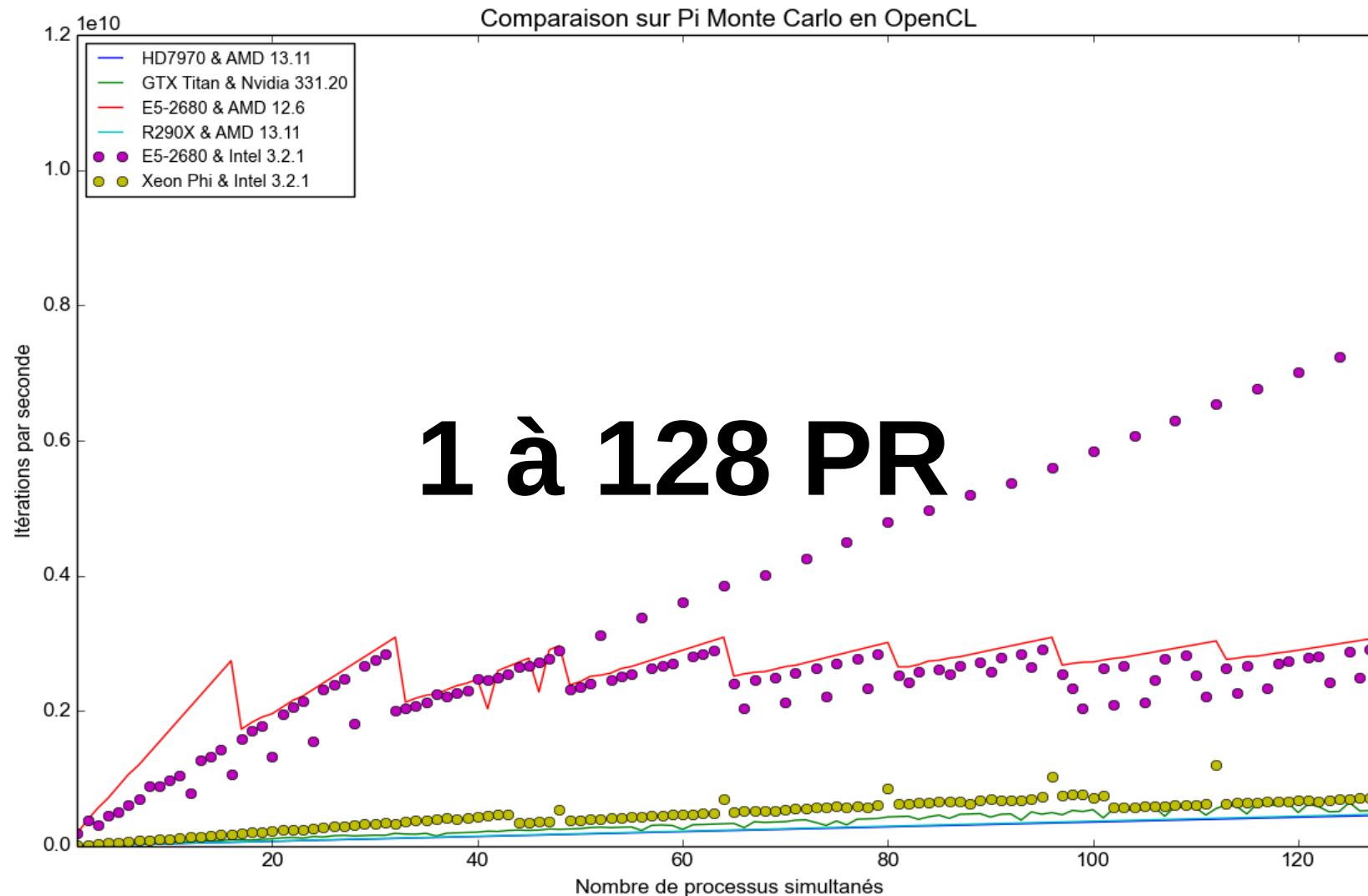
## Question de battement, de cœurs, et de bits !



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

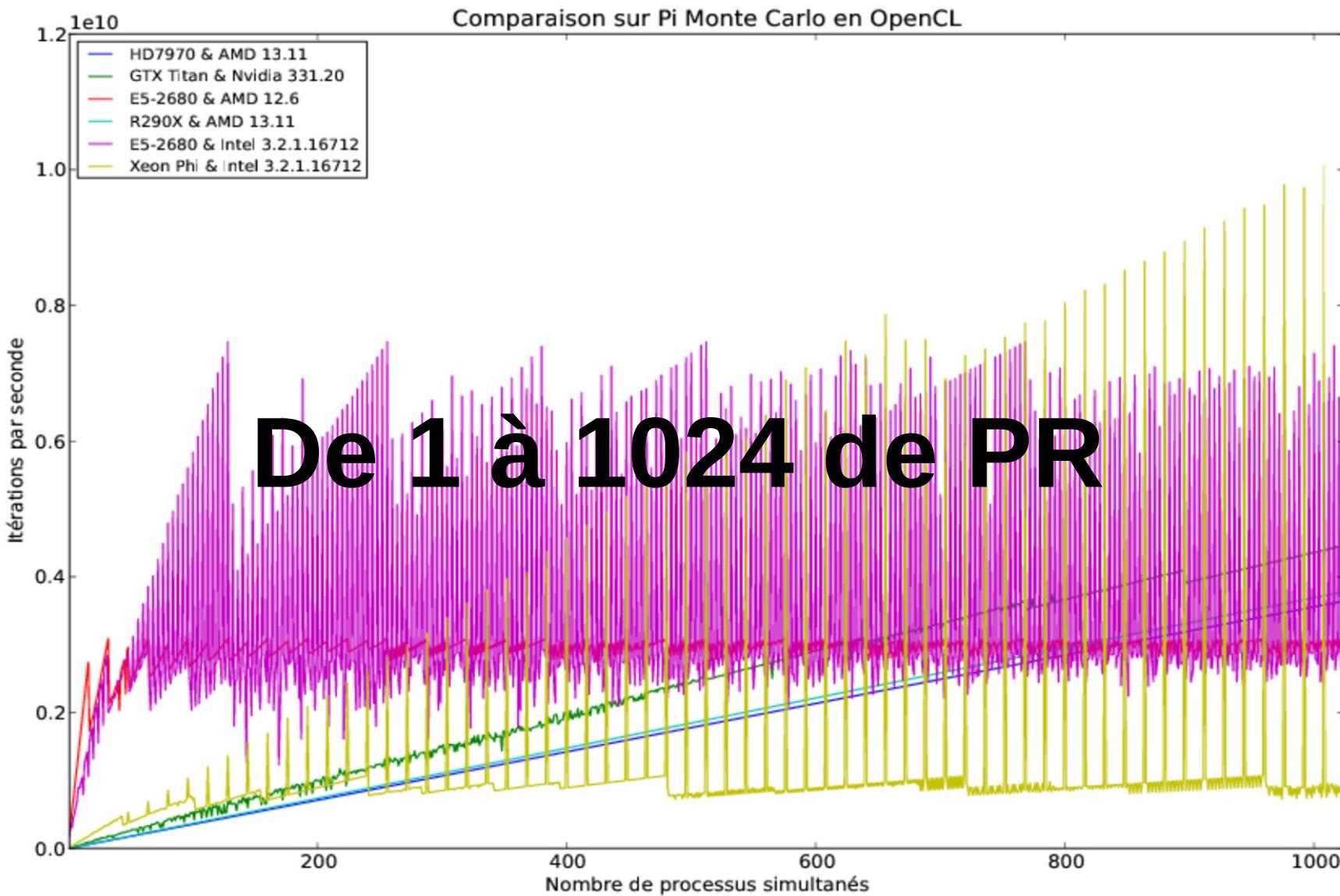
# Petite comparaison entre \*PU

## Bas pararégime : le règne du CPU

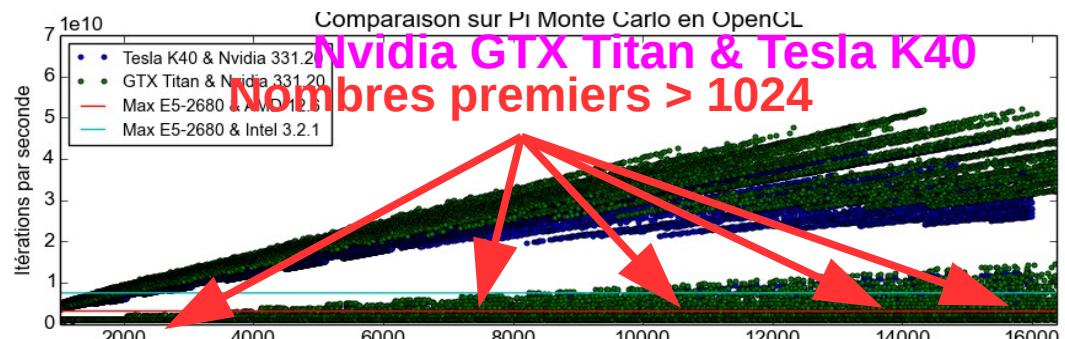
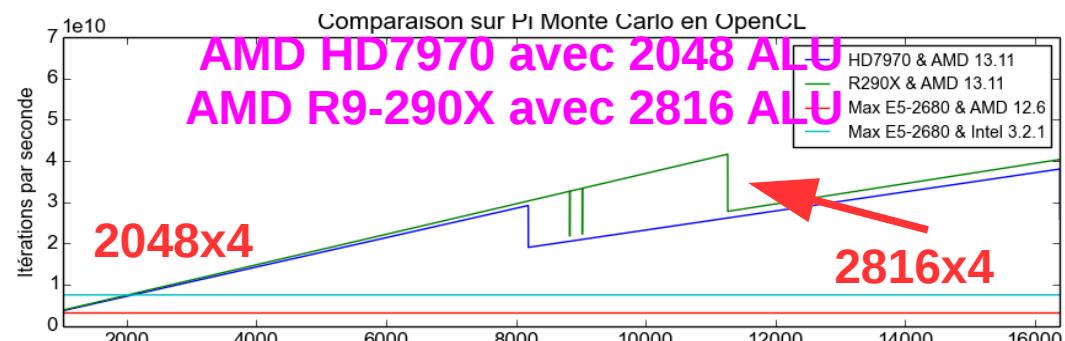
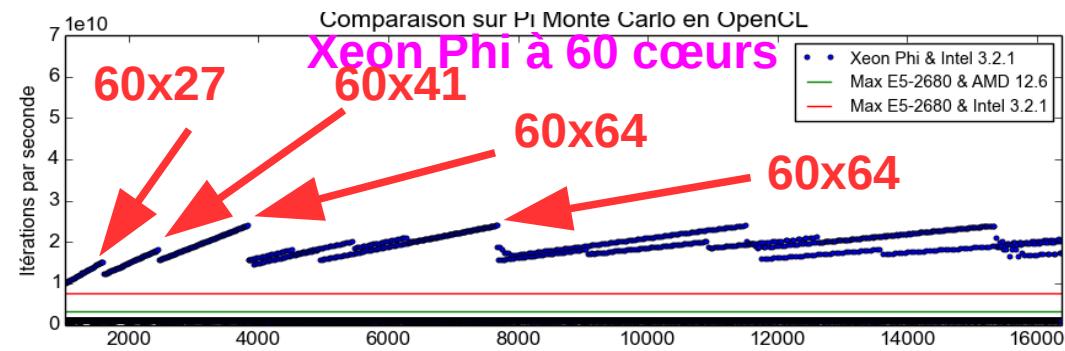
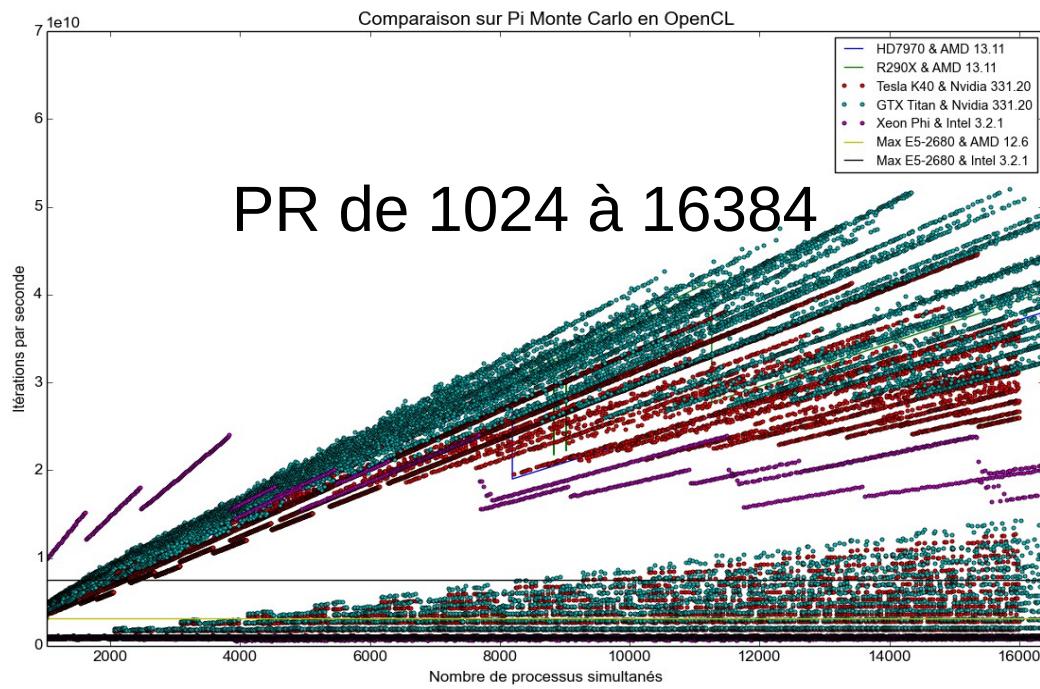


# Les processeurs toujours efficaces

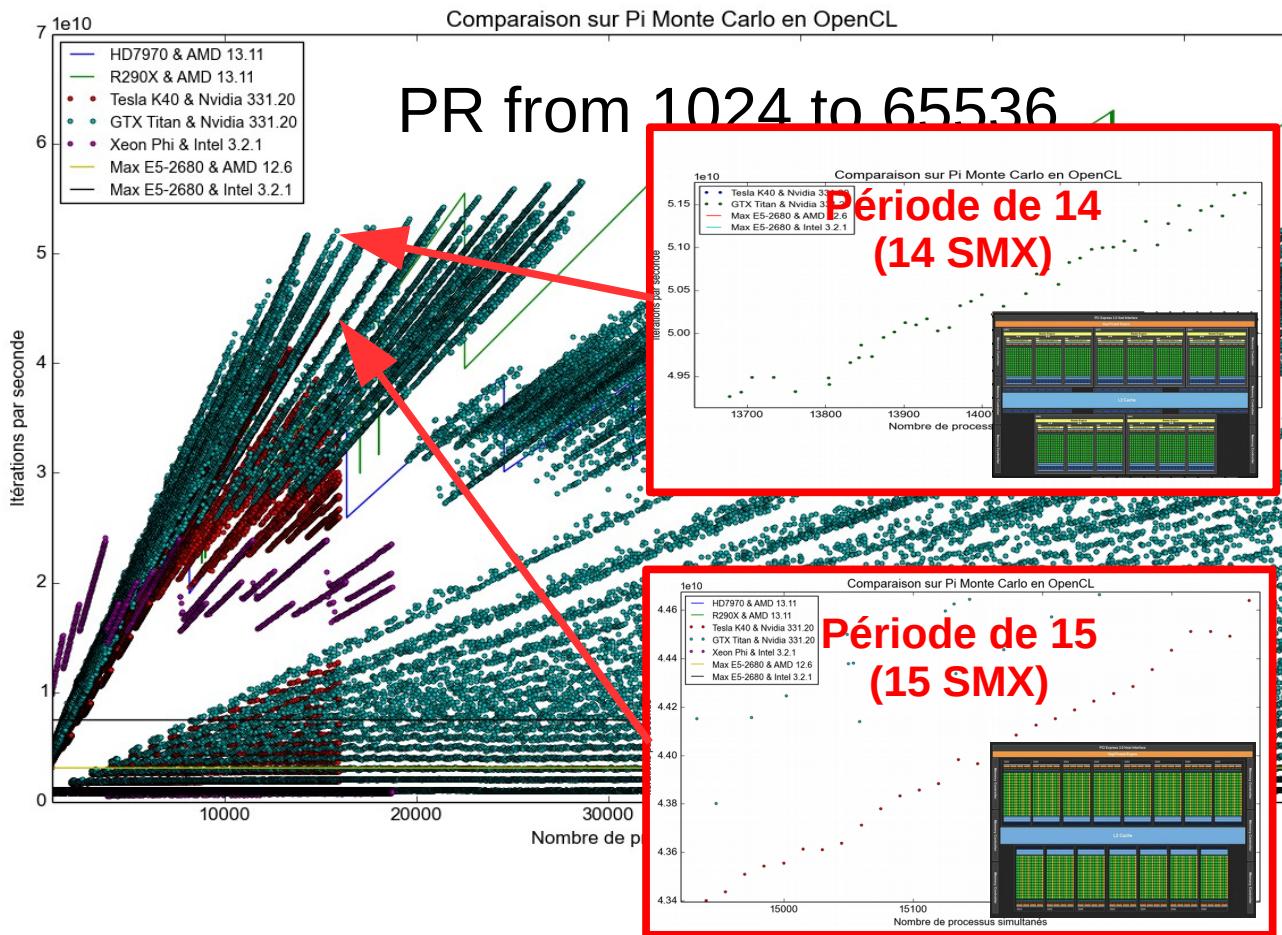
## Le Xeon Phi « sort du bois »



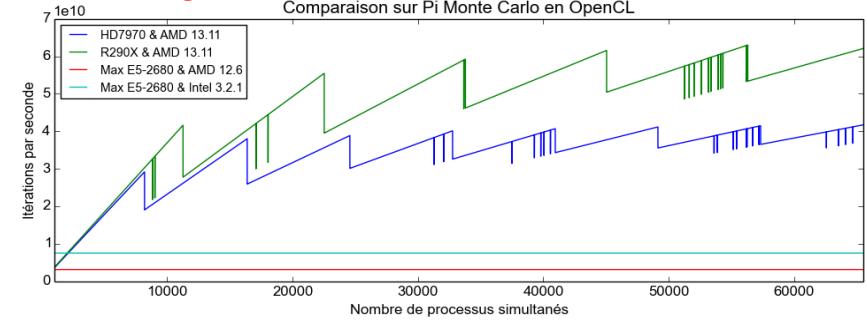
# Exploration profonde de PR ParaRégime de 1024 à 16384



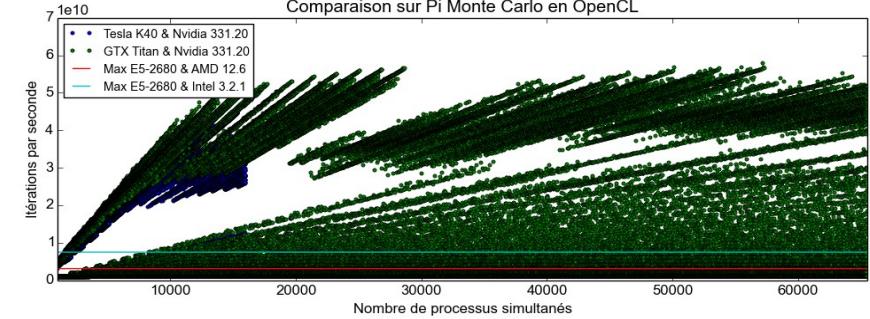
# Investiguer la structure interne Par l'exécution de Pi Dart Dash



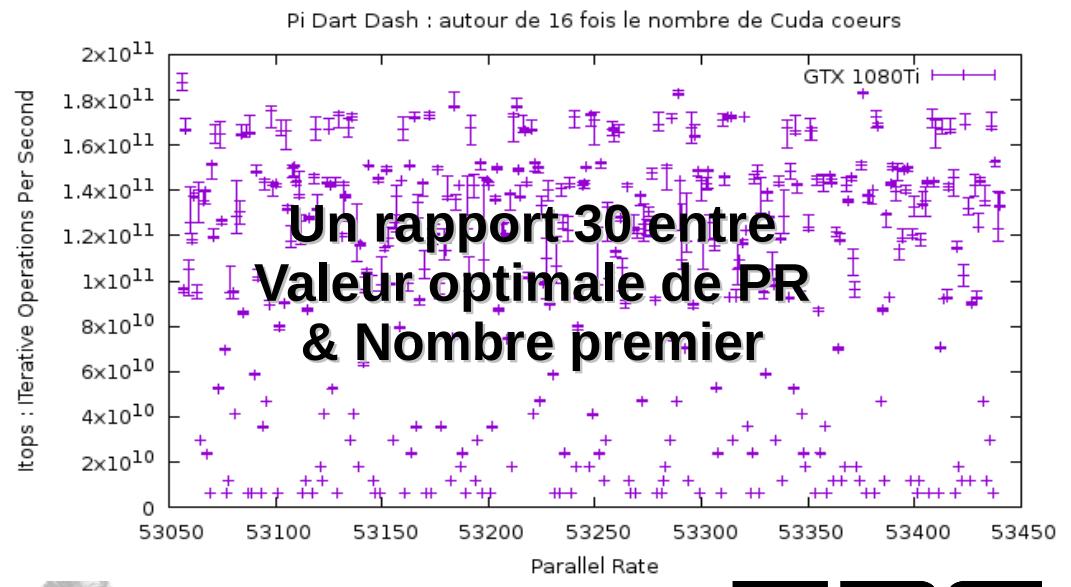
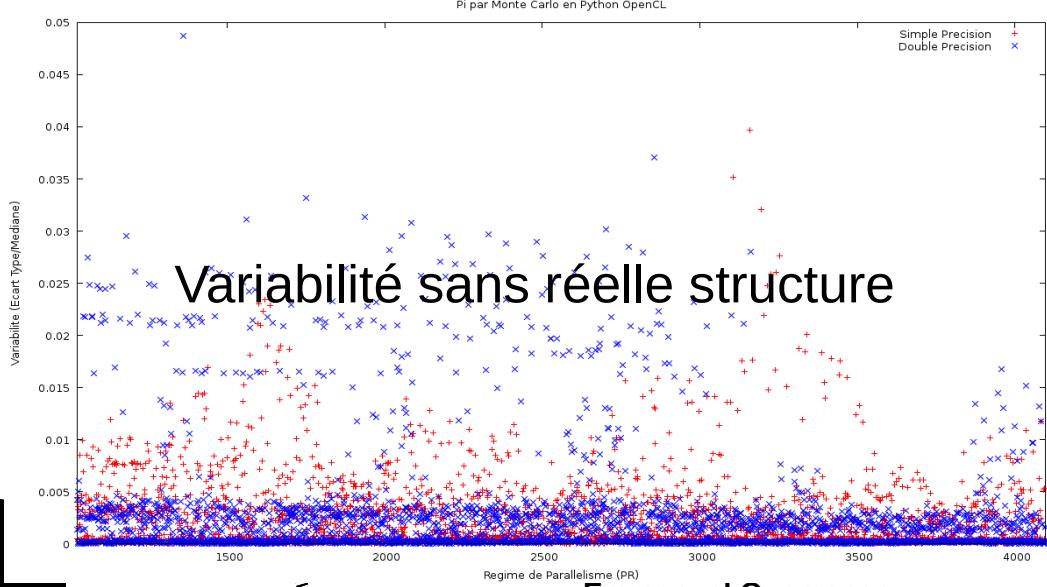
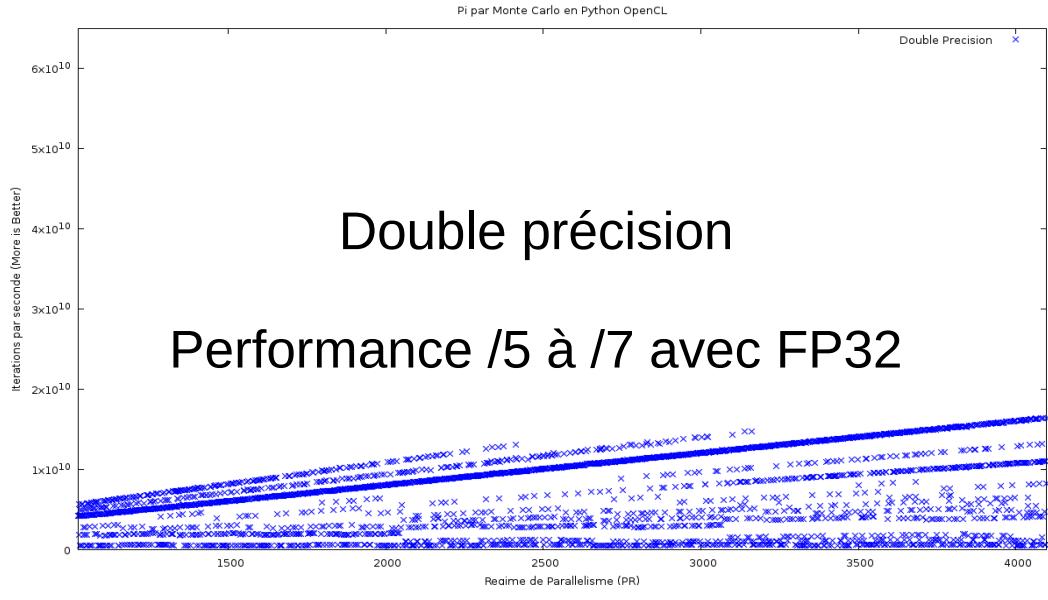
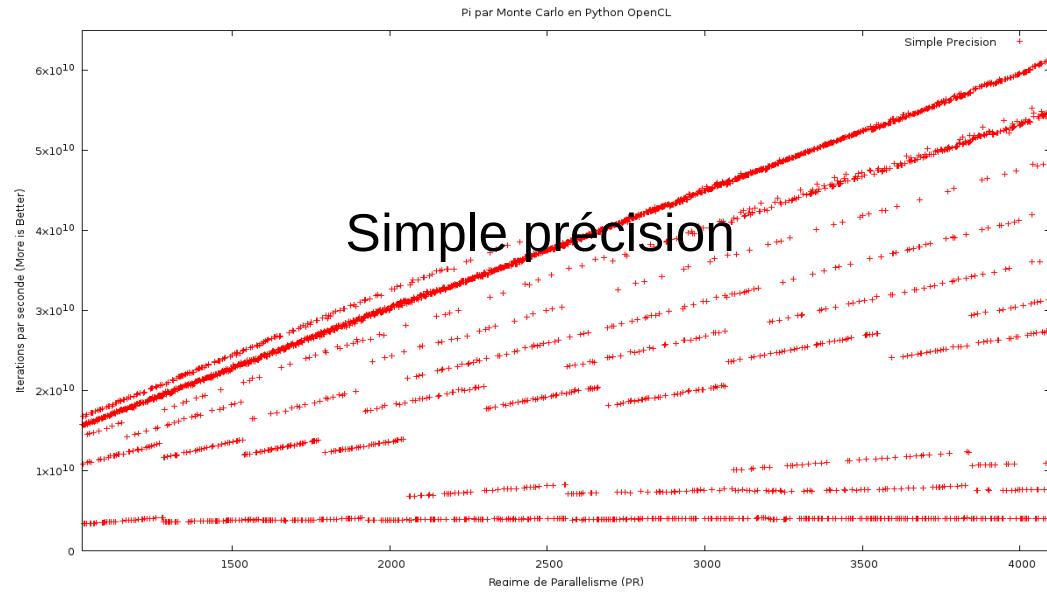
AMD HD7970 & R9-290  
Longue Période : 4x nombre d'ALU



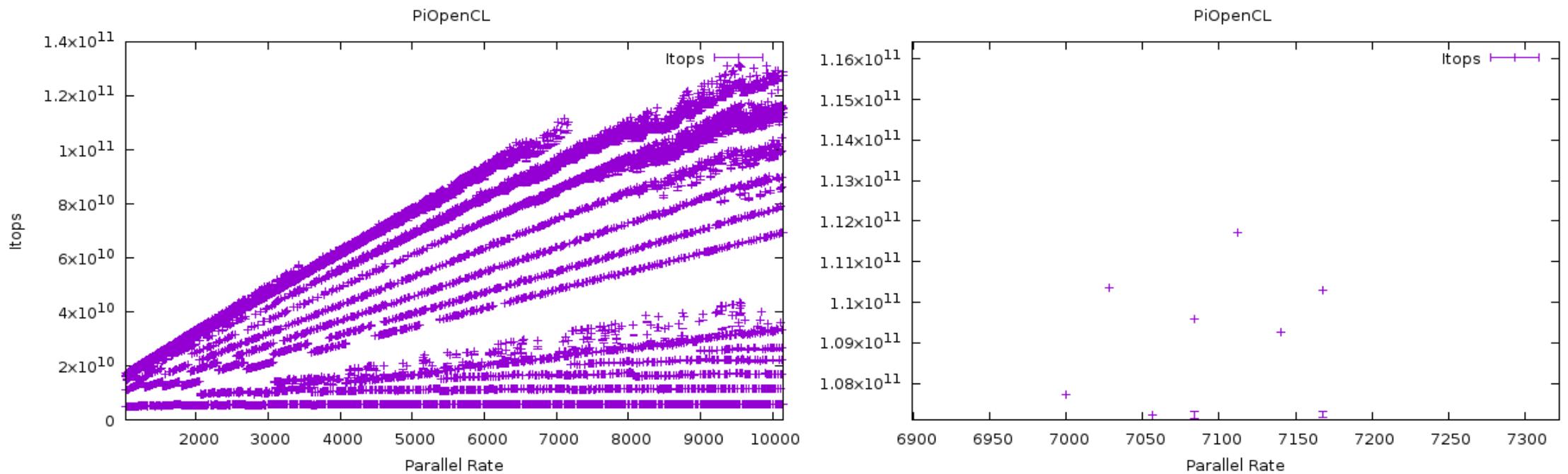
Nvidia GTX Titan & Tesla K40  
Courte Période : nombre d'unités SMX



# Et les dernières Nvidia GTX1080(Ti) Toujours affectées de bizarries ?



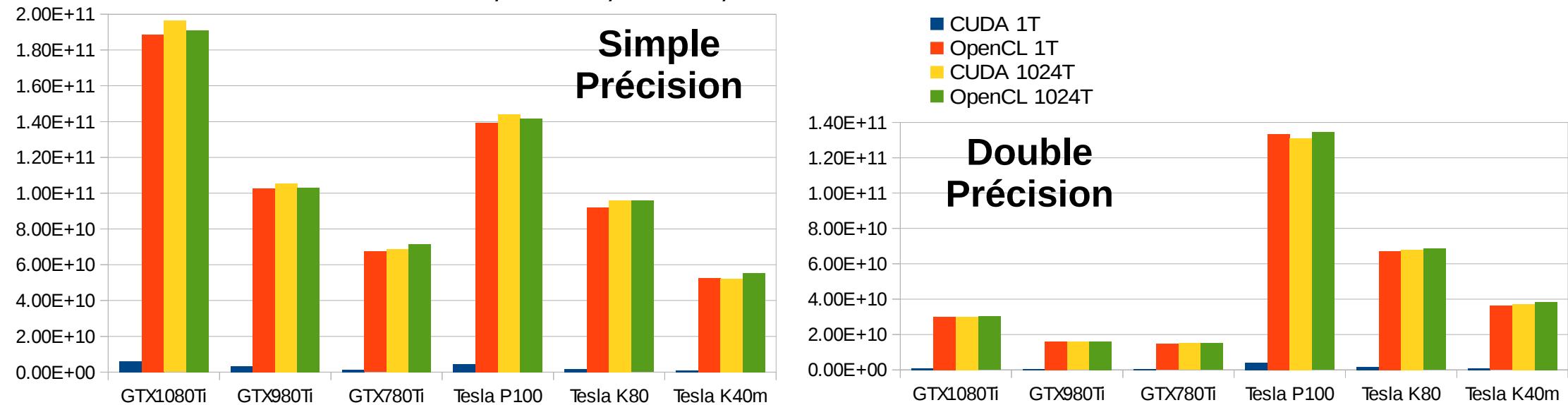
# Et en passant sur un PiOpenCL en C pur ?



- Les mêmes détections de nombres premiers
- Les mêmes max (x2 le nombre de shaders)

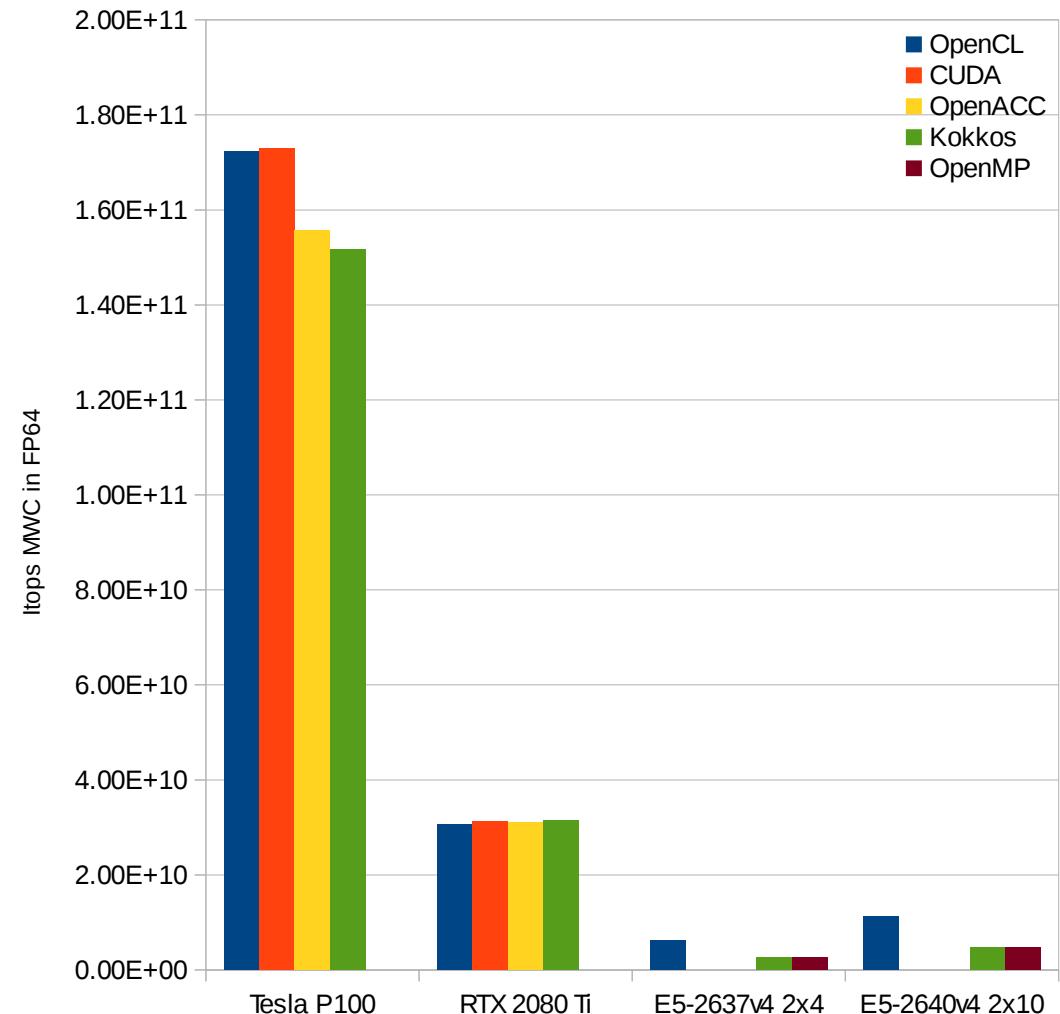
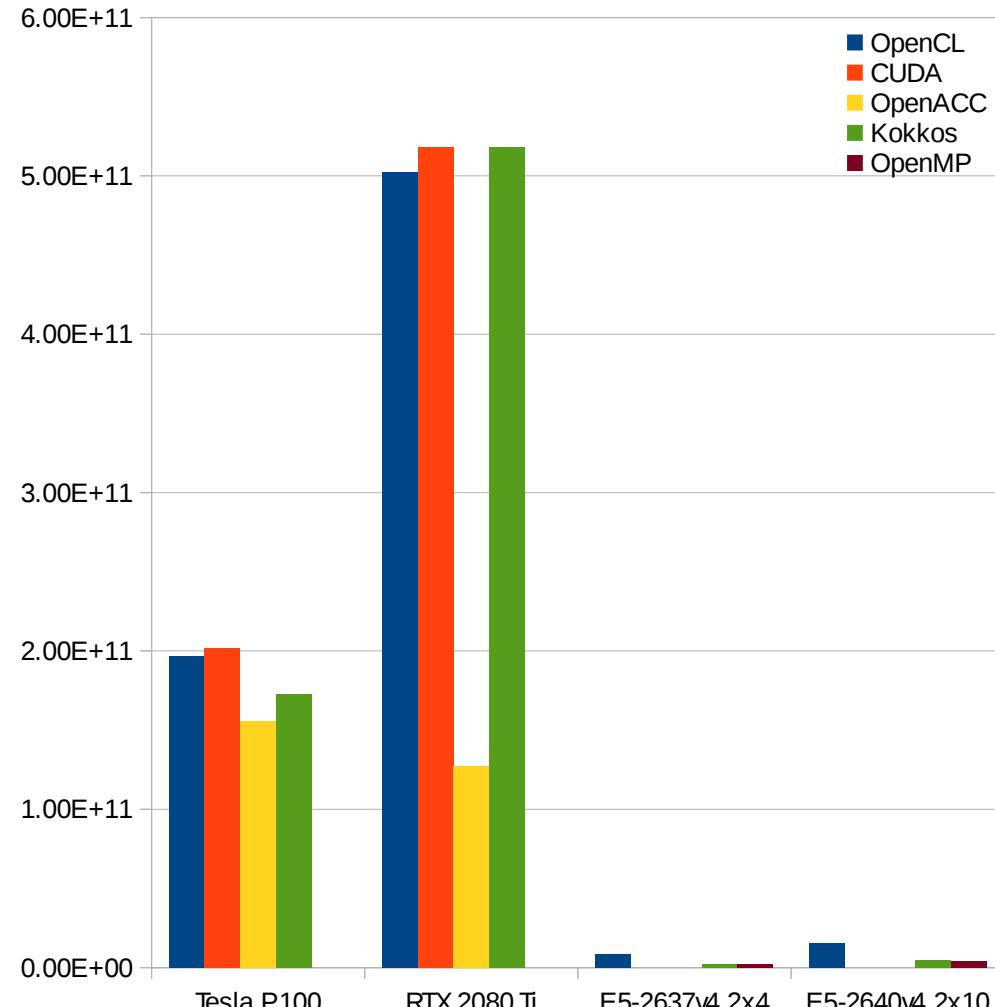
# CUDA vs OpenCL : le match

- 6 cartes : 3 GPGPU, 3 GPU, 3 générations
- Un régime de parallélisme mixte :
  - Entre Blocks & Threads ou entre Work Items & Threads
  - (Blocks,Threads)=(CudaCores,1024) ou (CudaCores\*1024,1)
  - CudaCores : 2816, 2880, 3584, 2x2496



Bref, pour ce code, CUDA~OpenCL, même mieux !

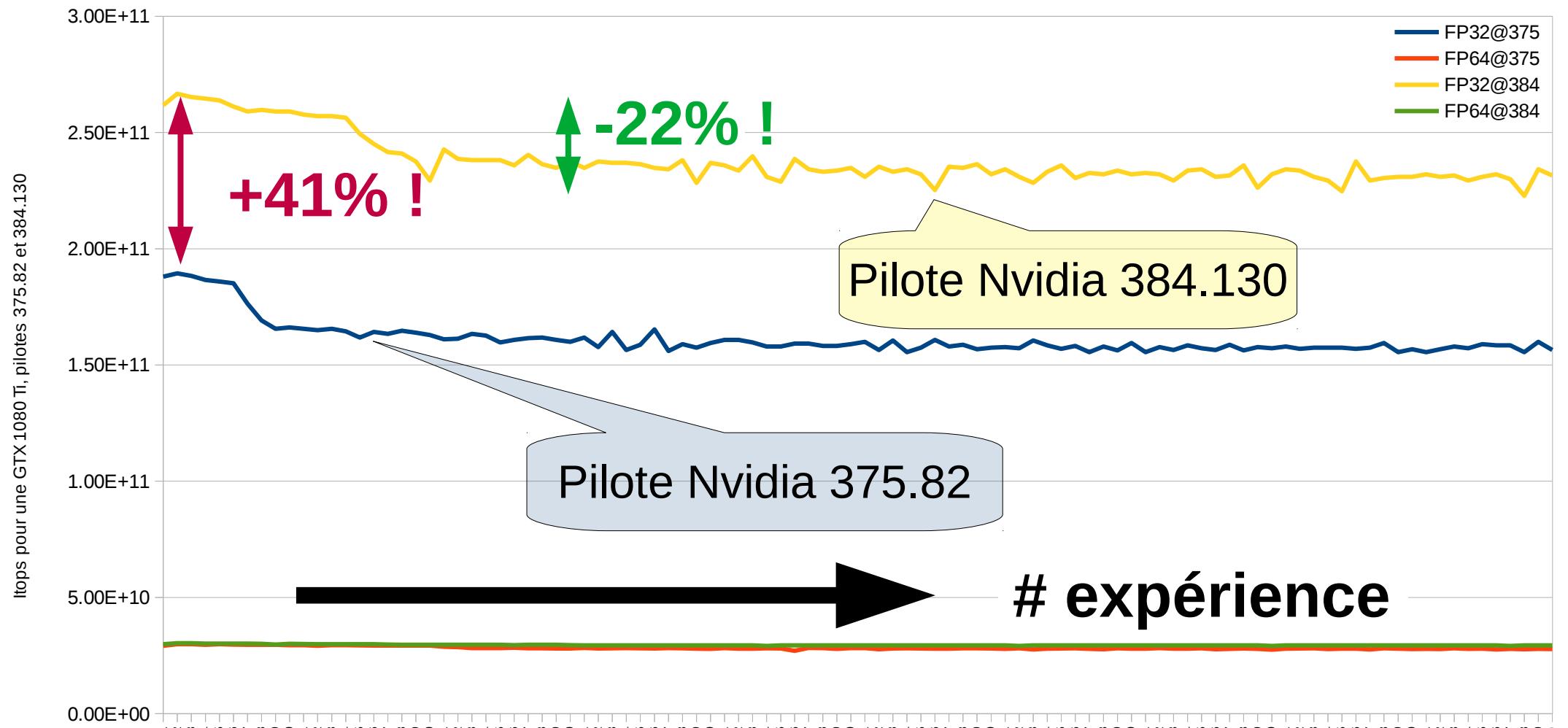
# Et pour les autres « approches » CUDA, OpenACC & Kokkos ?



Finalelement, 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 !

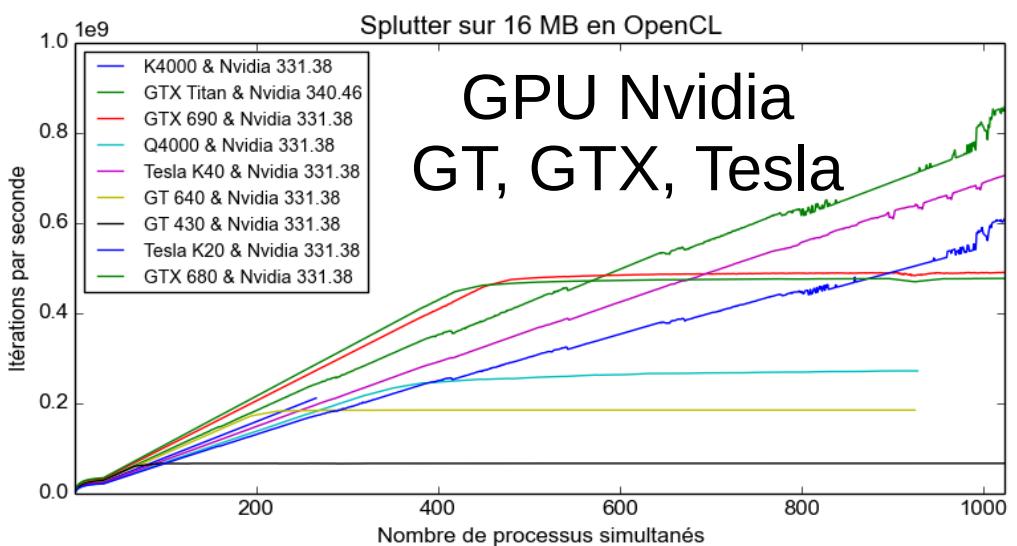
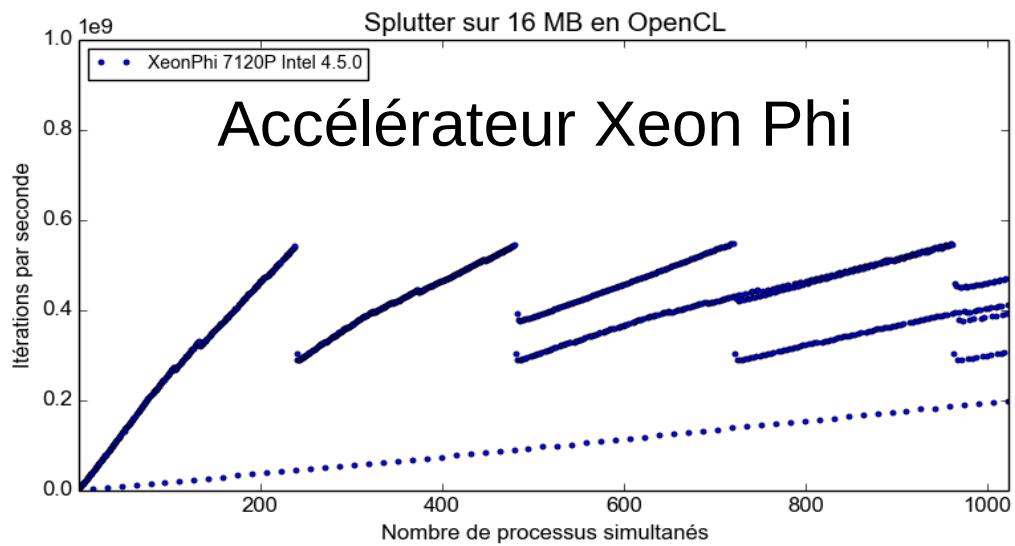
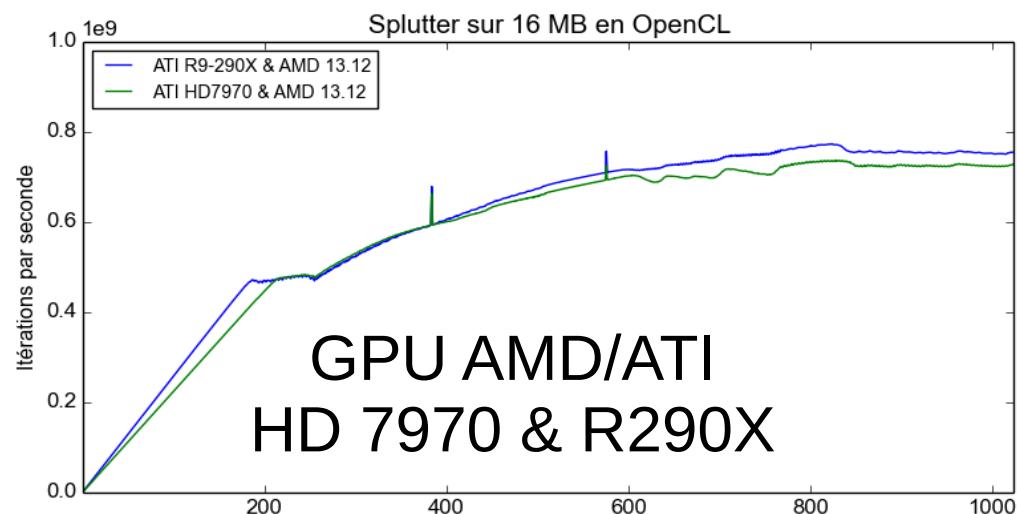
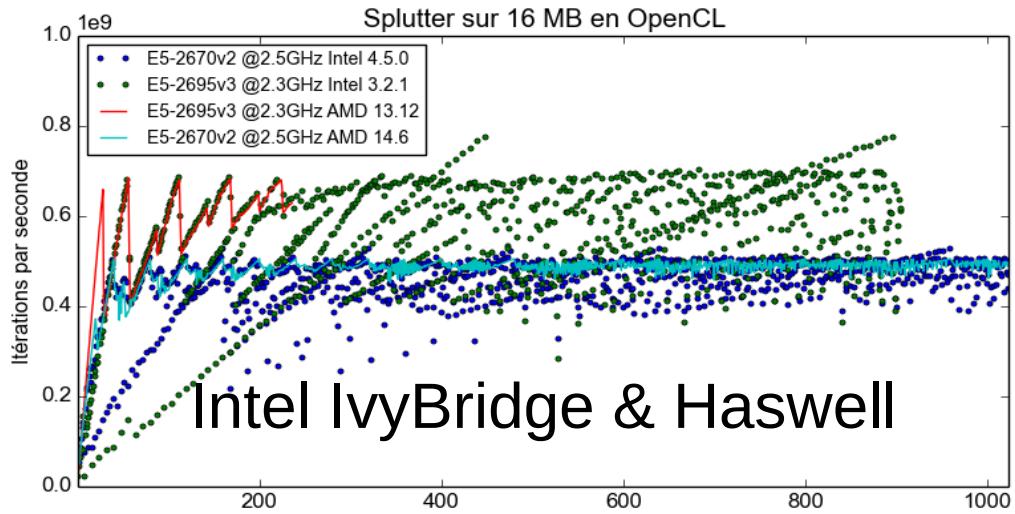
# Un test « Memory Bound »

## Le « postillonneur »...

- Le programme :
  - Je prends un espace mémoire
  - Je le transfère sur le périphérique (GPU ou CPU) : H2D
  - Pour chaque tâche parallèle, nombre équivalent d'itérations
    - Je tire un nombre aléatoire modulo l'espace mémoire
    - J'incrémente avec atomic\_inc l'espace mémoire
  - Je récupère l'espace mémoire du périphérique : D2H
- La vérification : je somme les éléments de l'espace
  - Je dois retomber sur le nombre d'itérations sélectionnés
- Les observables : les 3 temps...
  - Le temps H2D, le temps de postillonnage, le temps D2H

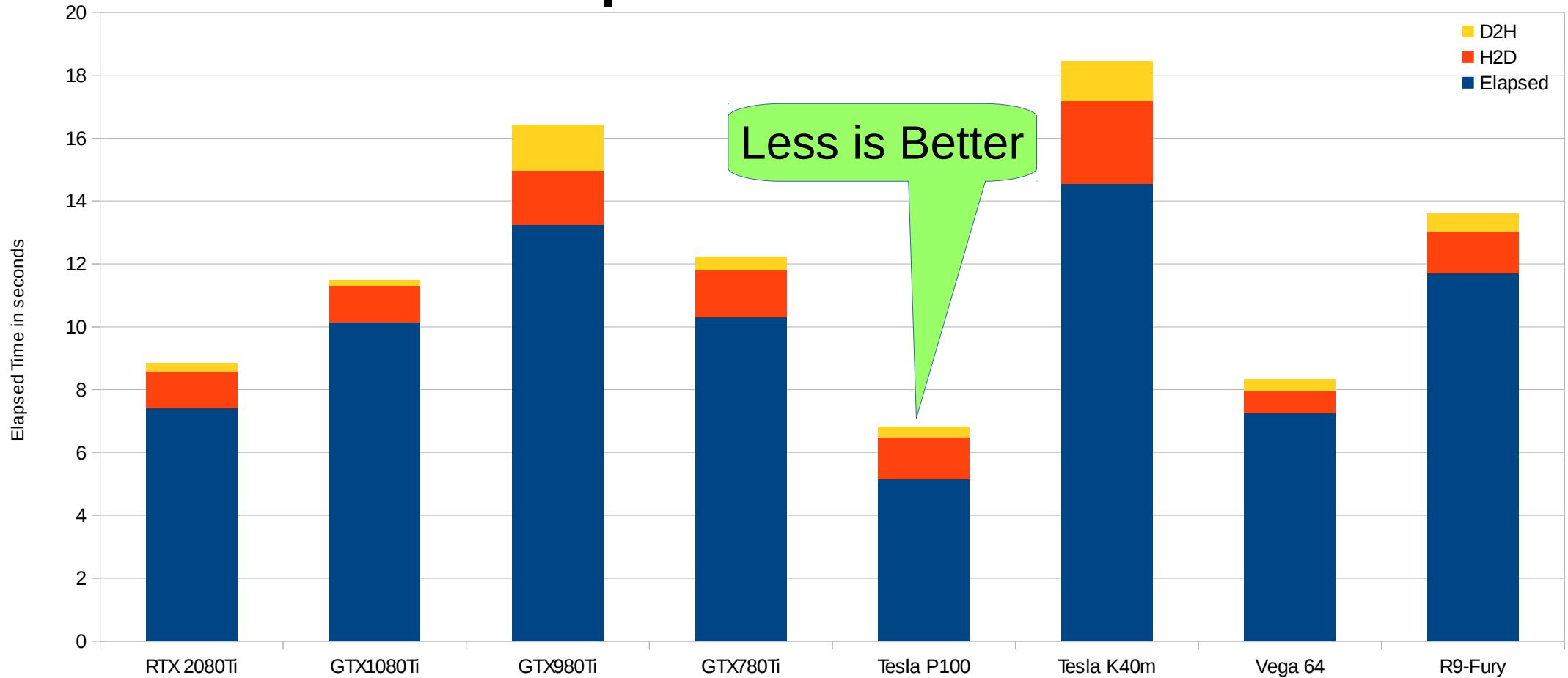
# Le « postillonneur » en OpenCL

## Il y a 4 ans...



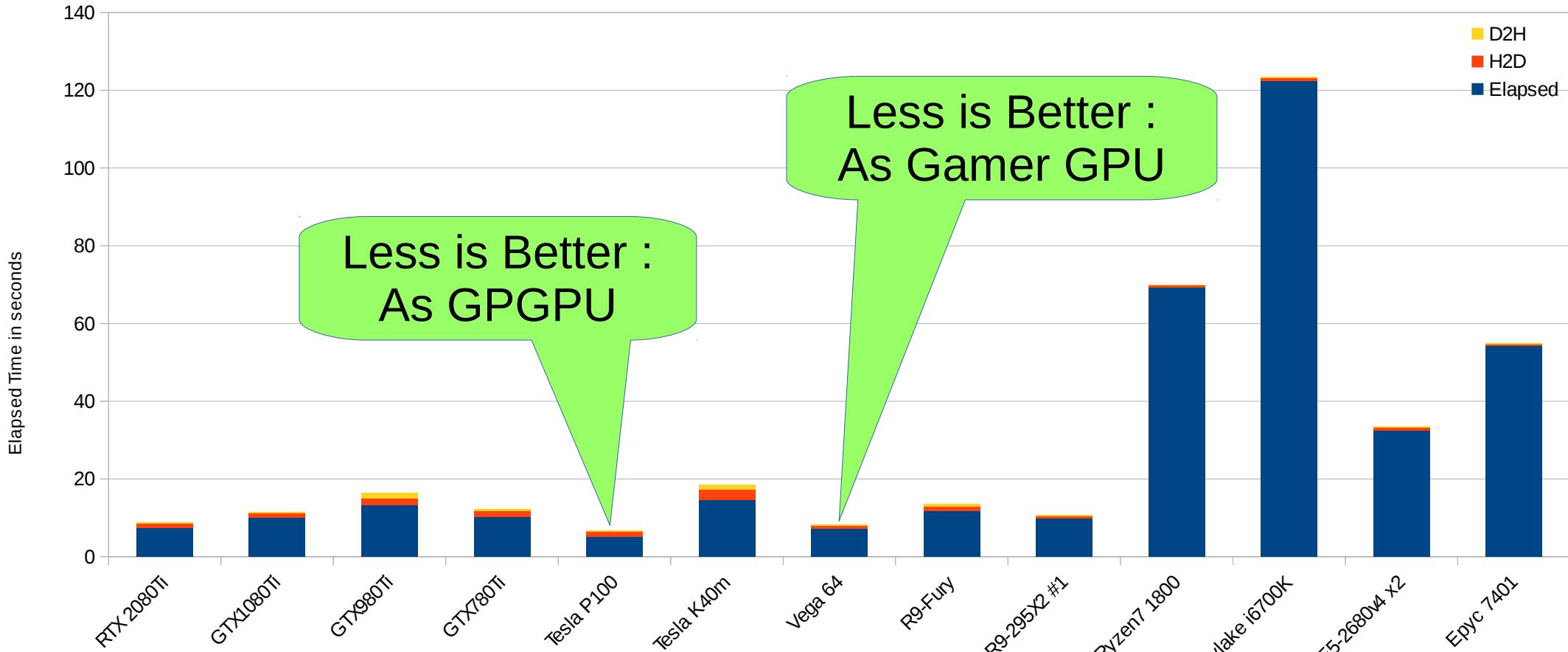
# Et pour les GPU & CPU utilisés ?

## D'abord pour les 9 GPU...



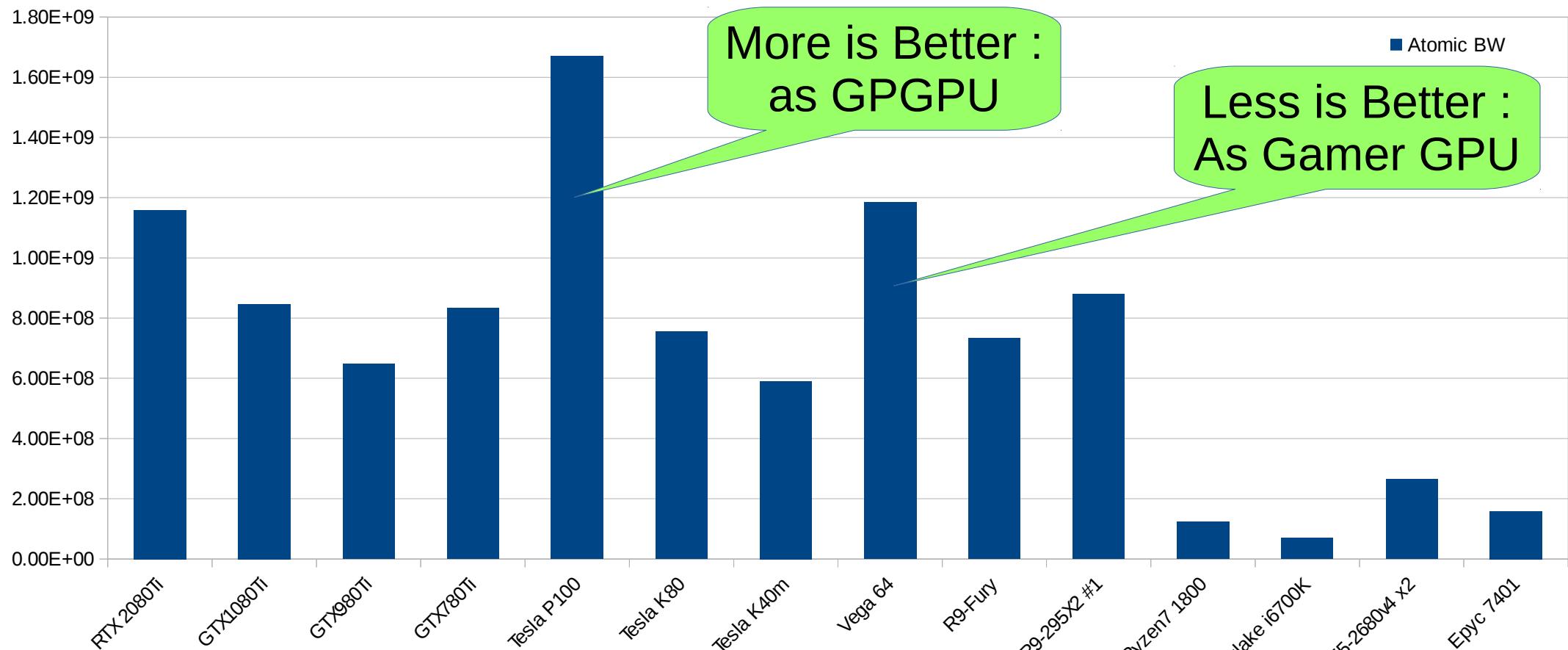
- Un espace de 2GB ( $2^{29}$  de 32 bits), 32768 tâches
- 8.5 Milliards d'itérations (16 par tâche)

# Et pour les GPU & CPU utilisés ? Les 9 GPU + les 3 meilleurs CPU



- Un espace de 2GB ( $2^{29}$  de 32 bits), 32768 tâches
- 8.5 Millards d'itérations (16 par tâche)

# Et pour les GPU & CPU utilisés ? Les 10 GPU + les 4 meilleurs CPU

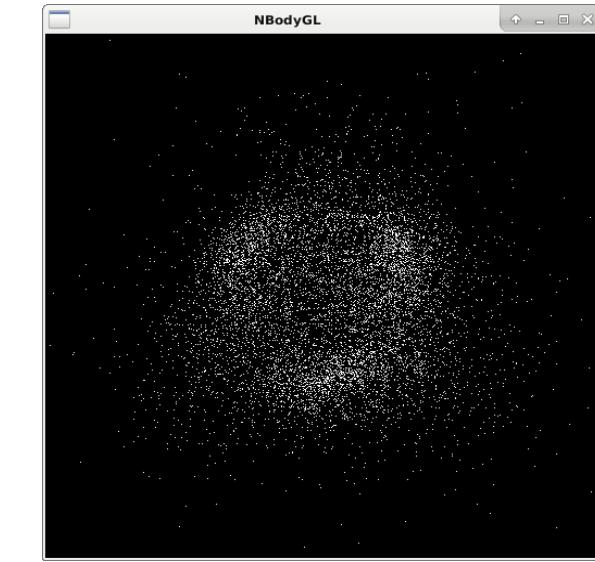


- Performance des GPU : pas uniquement dans le calcul
- Accès mémoire beaucoup plus rapide !

# 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 Implicit, Euler Explicit, Heun, Runge Kutta
- Données d'entrées :
  - Physique : nombre de particules
  - Numérique : pas d'intégration, nombre de pas
  - Architecture : influence de la nature du flottant FP32, FP64



```
numa@vivobook: ~/bench4gpu/NBody
File Edit View Search Terminal Help
('Platform selected: ', 'NVIDIA CUDA')
All particles superimposed.
All particles distributed
Center Of Mass estimated: (-0.00047948415, 0.07945487, 0.05831058)
All particles stressed
Energy estimated: Viriel=-0.125 Potential=-888202.5 Kinetic=444101.2

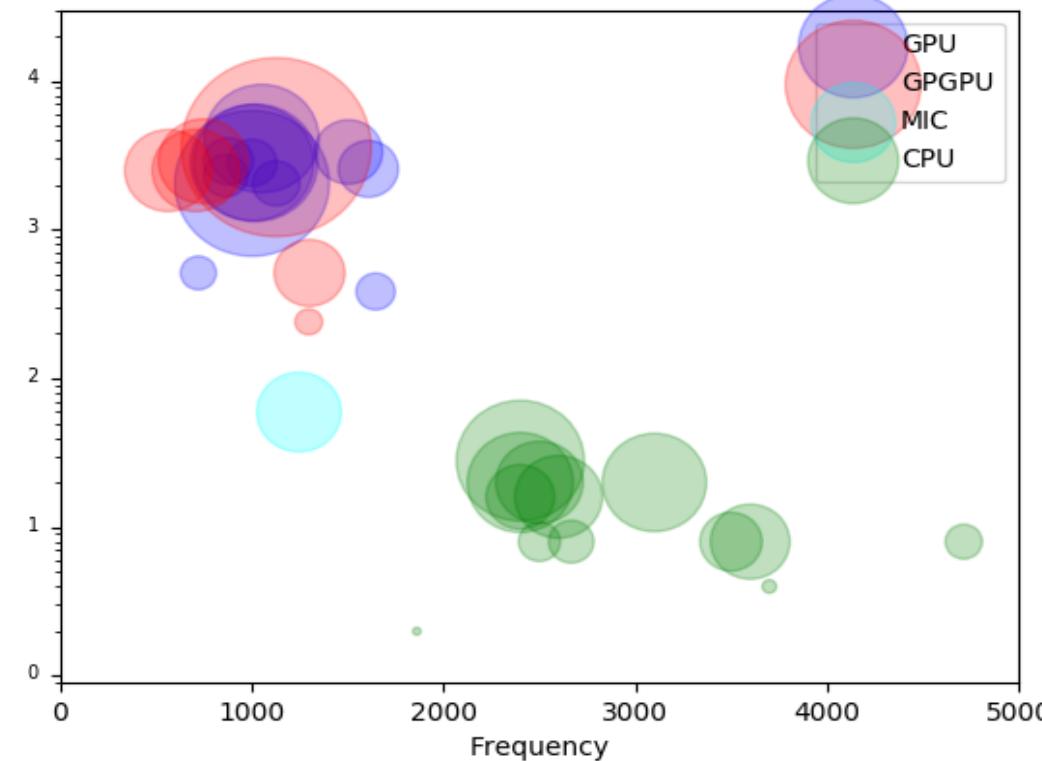
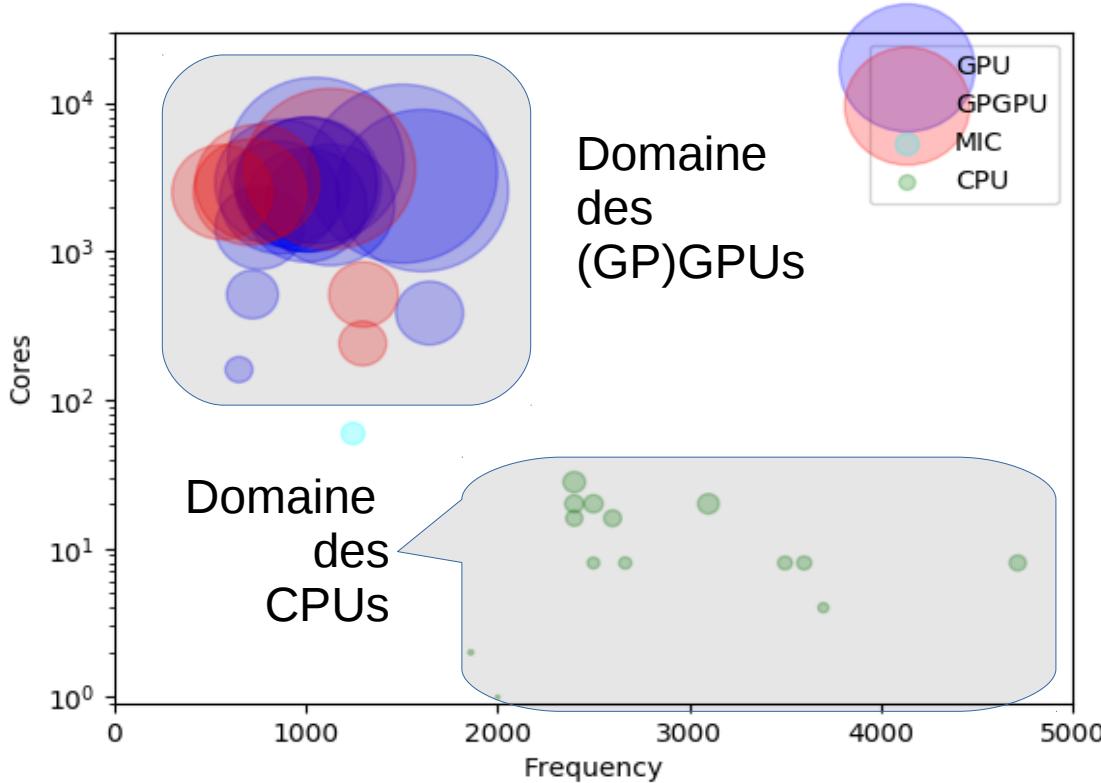
Tiny documentation to interact OpenGL rendering:
<Left|Right> Rotate around X axis
<Up|Down> Rotate around Y axis
<z|z> Rotate around Z axis
<-|+> Unzoom/Zoom
<s> Toggle to display Positions or Velocities
<Esc> Quit

Starting!
```

A screenshot of a terminal window showing the output of the NBodyGL application. It displays various system information and a tiny documentation for interacting with the OpenGL rendering. The terminal window has a title bar and a scroll bar on the right side.

# La performance en informatique

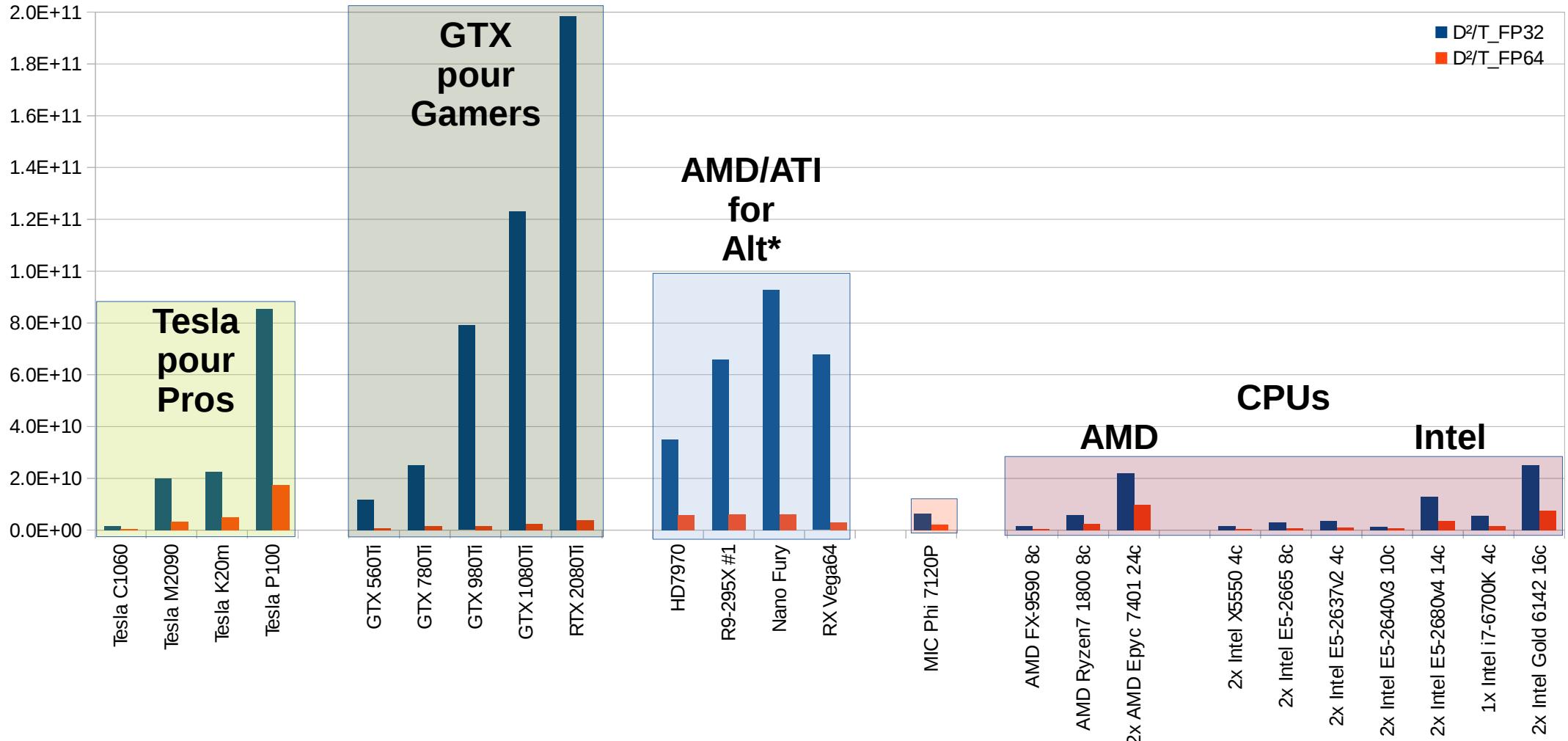
## Pour un code plus physique...



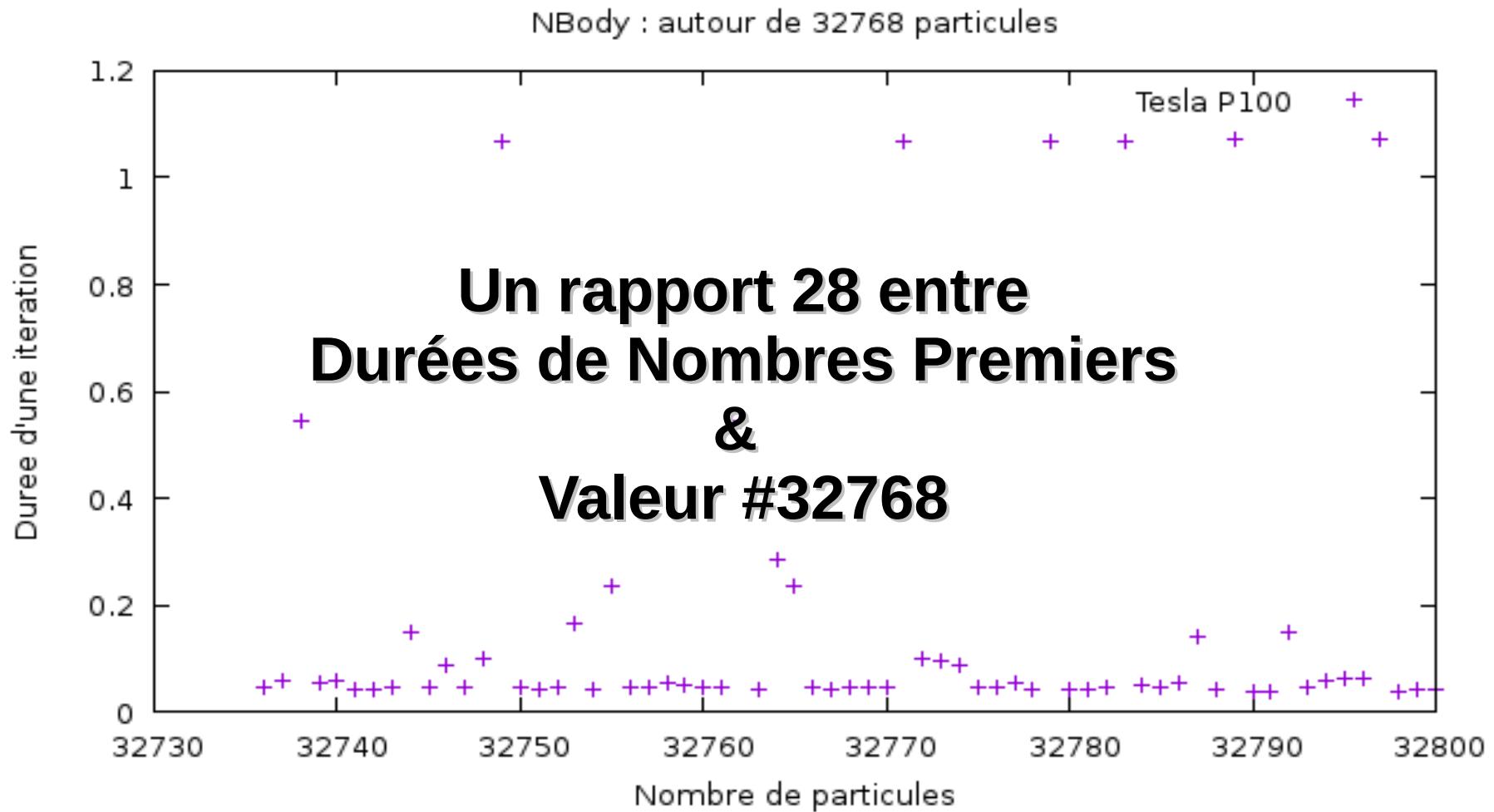
- Sur un GPGPU, performance stable
- Sur un GPU, baisse drastique de la performance (division par 20 en DP)
- Sur un CPU, performance relative meilleure

# Modèle N-corps « naïf »

## Une comparaison classique...



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

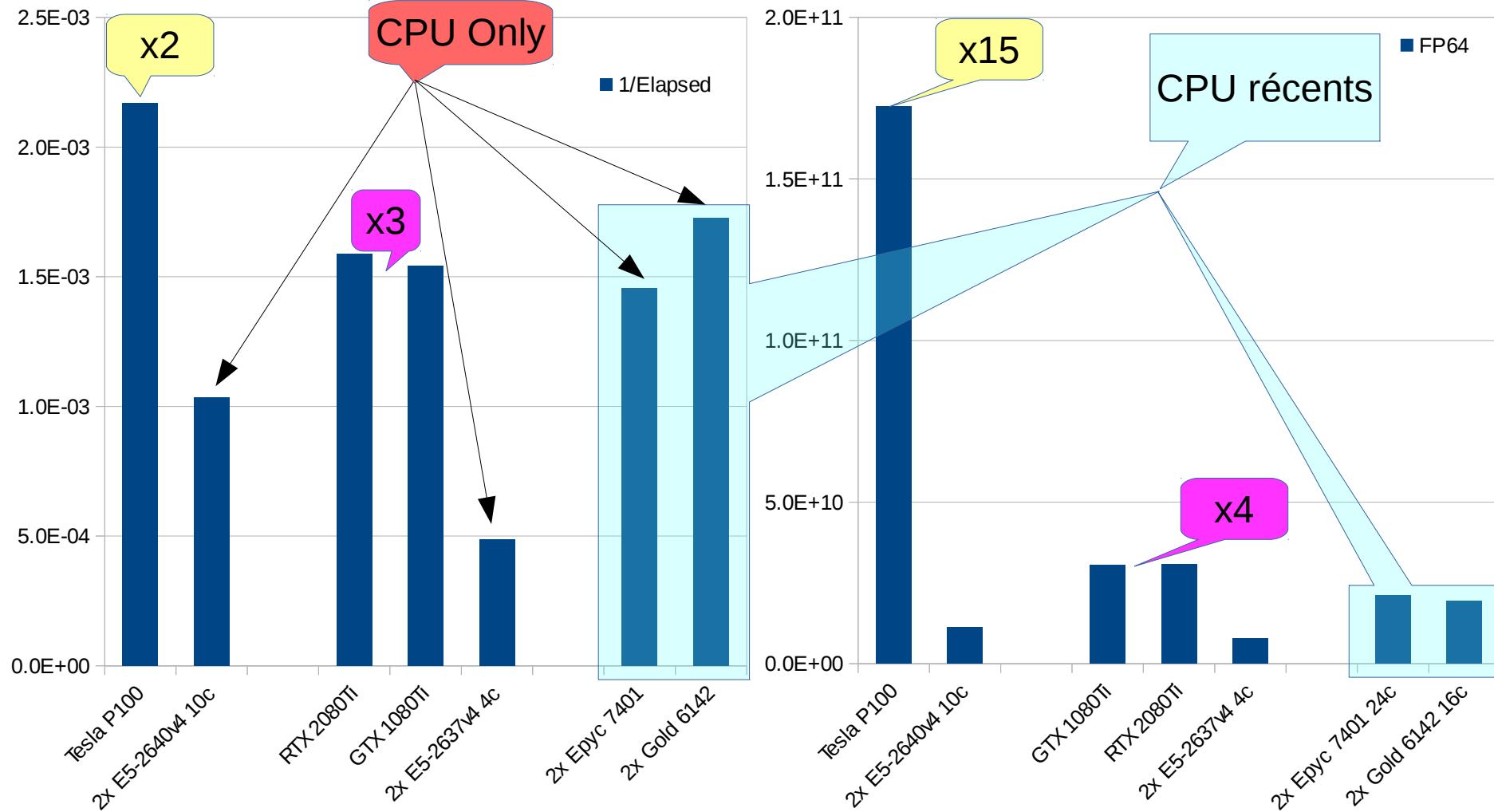


# Une petite démonstration de Nbody ?

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

# Le futur, c'est vous... Et il y a du taff !

## PKDGRAV3 & PiDartDash en FP64

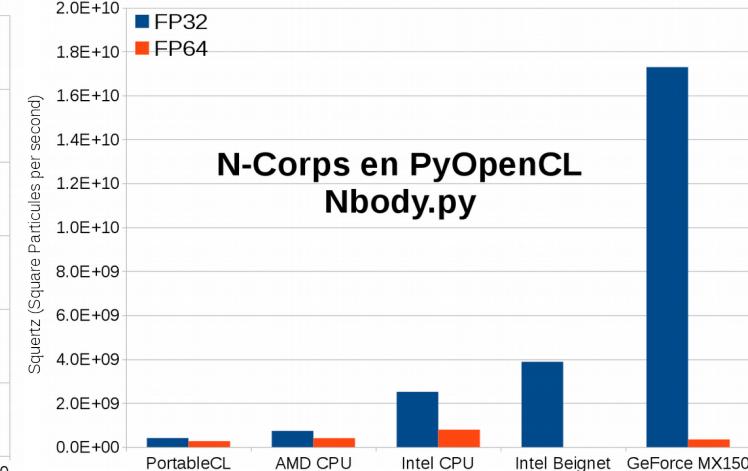
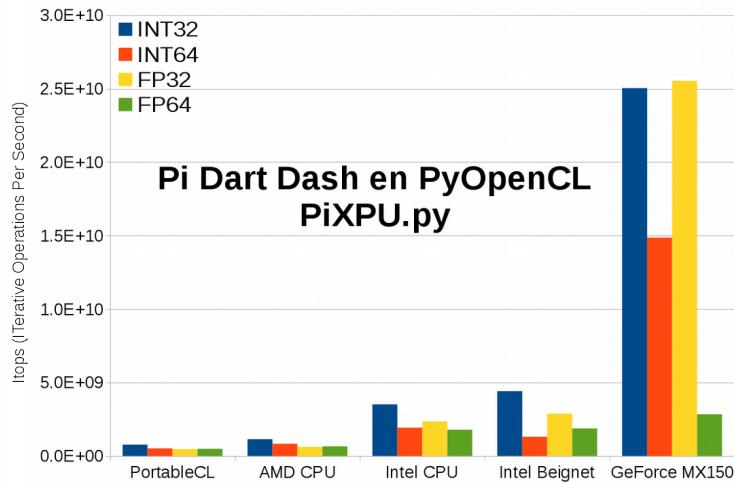
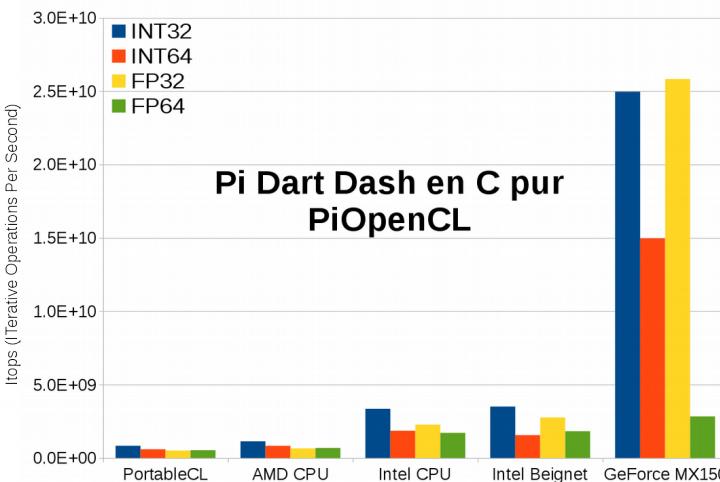


# Questions à se poser...

- 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 < 16GB ?
- 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 2018, 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%20B0016%20-%20Page%2013%20%28Octobre%201984%29.jpg](http://download.abandonware.org/magazines/Tilt/tilt_numero016/%5BMag%20-%20Fr%5D%20Tilt%20n%20B0016%20-%20Page%2013%20%28Octobre%201984%29.jpg)