Python pour l'exploration numérique

De la gestion de clusters ...

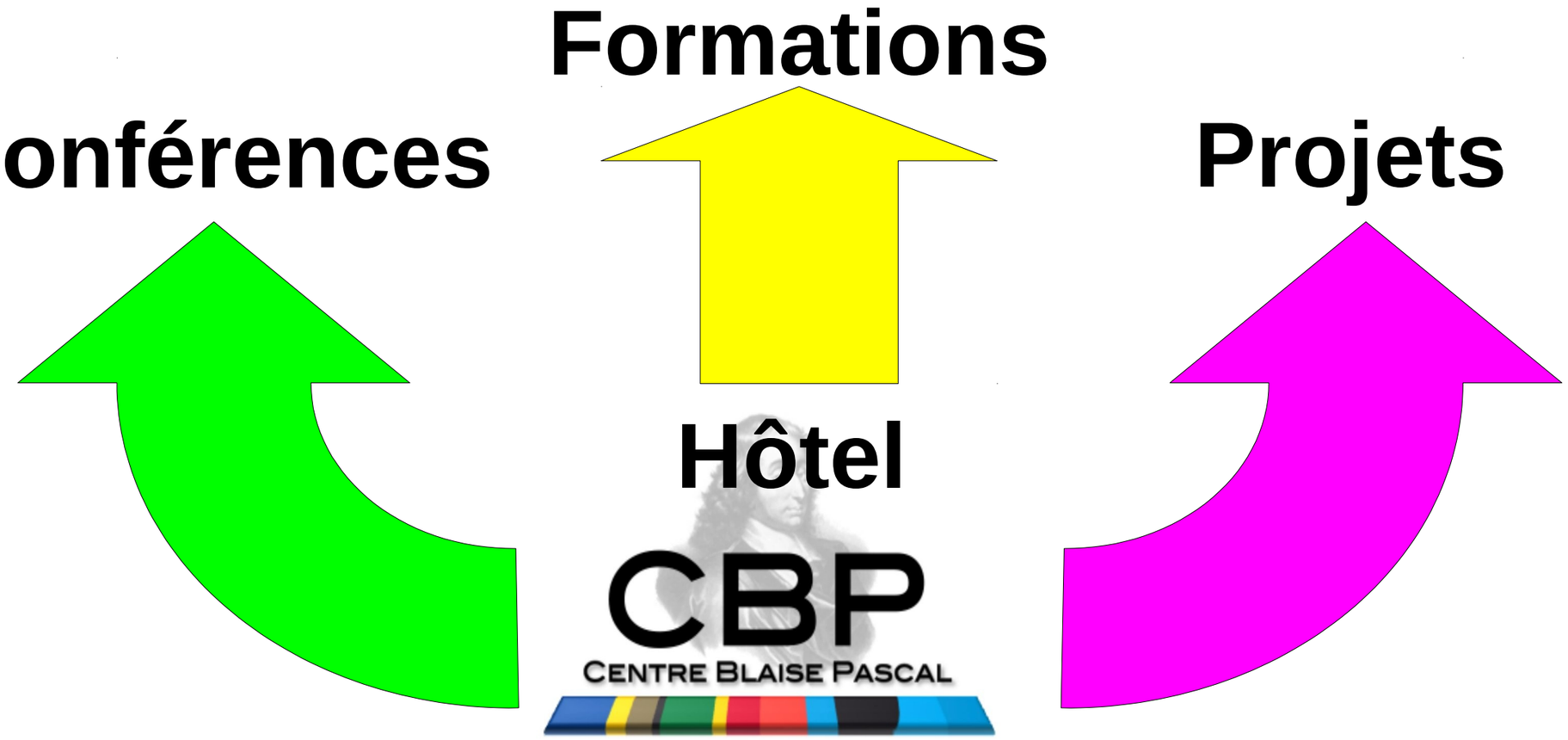
À l'examen in silico des processeurs :

la polyvalence de Python

au travers de deux exemples...

« L'expérimentateur pose des questions à la nature ; mais, dès qu'elle parle, il doit se taire ; il doit constater ce qu'elle répond, l'écouter jusqu'au bout, et, dans tous les cas, se soumettre » Claude Bernard

Qu'est-ce que le Centre Blaise Pascal ?
Hôtel à projets & Maison de la modélisation
Avant tout, « outil » de recherche...



« Catalyser » l'informatique scientifique :
CBP : Maison de la Simulation & Centre d'essais

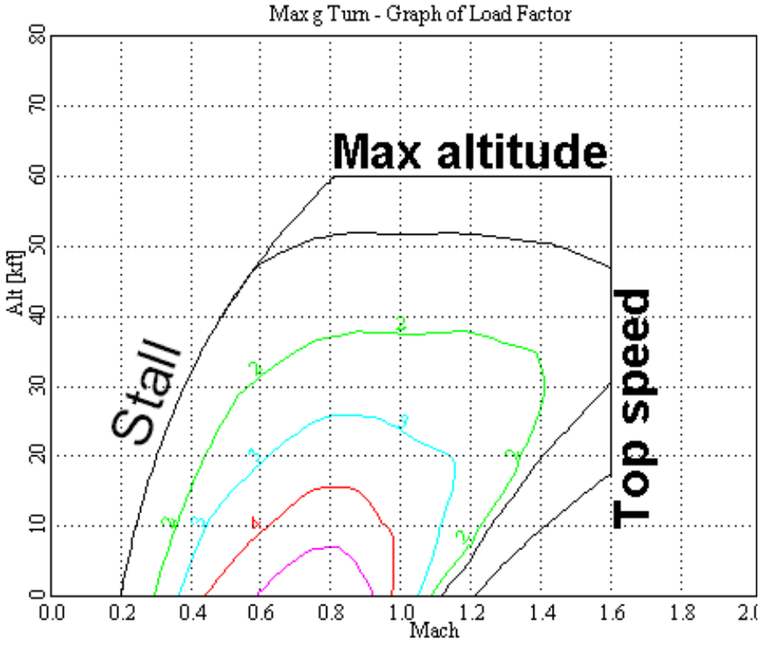
Nasa X29

- Cellule de F5
- Moteur de F18
- Servos de F16
- Études
  - Flèche inversée
  - Incidence >50°
  - *« Fly-By-Wire »*

Le CBP (via son pilote d'essais) réutilise et explore...

Domaines de vol/fonctionnement : la fin du « ça marche/marche pas »

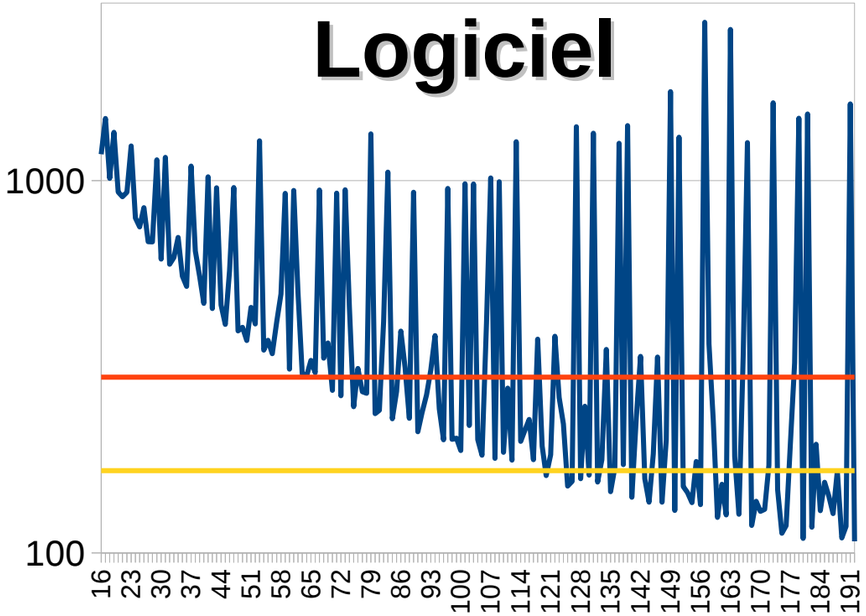
Enveloppe de vol

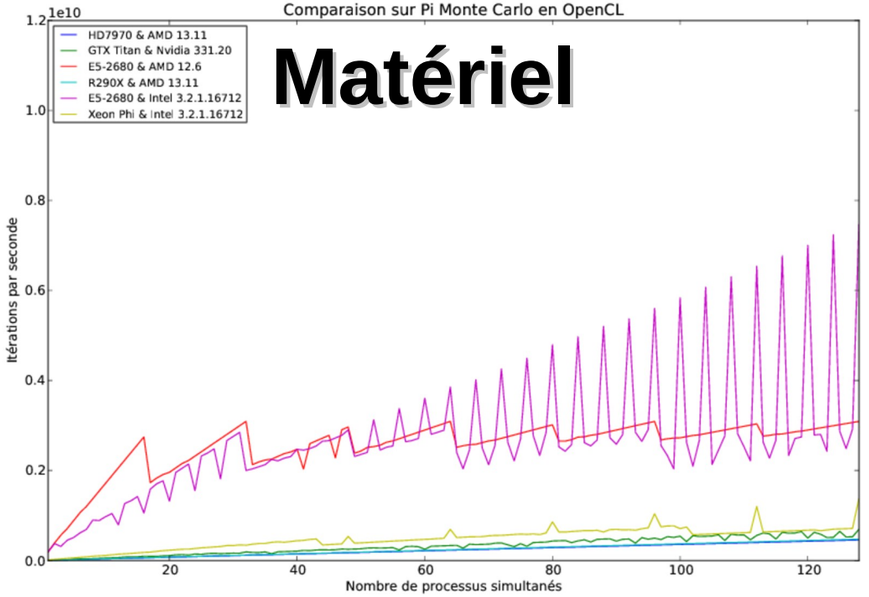


Vitesse/altitude

facteur de charge

Enveloppes de parallélisation





Parallélisation/Mémoire/CPU/GPU

Entrées/Sorties/Contextes/...

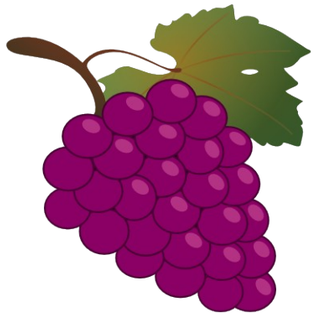
Une Plate-Forme Expérimentale : Des plateaux techniques

- **Multi-nœuds : 76 permanents de 4, 24 et 48 nœuds**
- **Multi-cœurs : 100 stations/serveurs/nœuds de 2 à 20 cœurs**
- **Multi-shaders : 28 stations/nœuds avec 22 (GP)GPU différents**
- Intégration logicielle (versions de distribution Debian, Ubuntu, CentOS)
- Paillasses numériques : expérience/démonstrateur/prototype
- Intégration matérielle (Sparc, PowerPC, … et ARM)
- Visualisation scientifique : plateau 3D
- Plateau COMOD : « *Compute On My Own Device* »

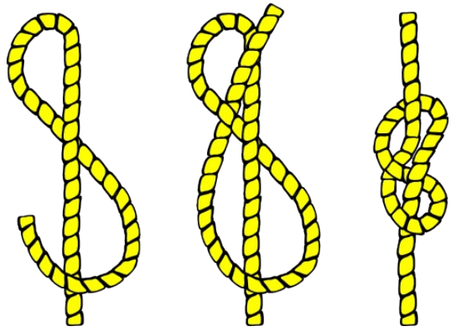
Pour étudiants, enseignants, chercheurs, ingénieurs

Calcul Haute Performance
Petit (et nécessaire?) rappel de vocabulaire

Grappe



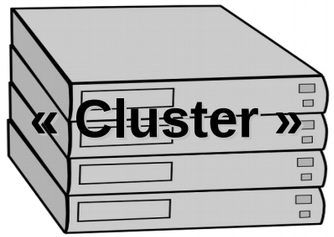
Nœud

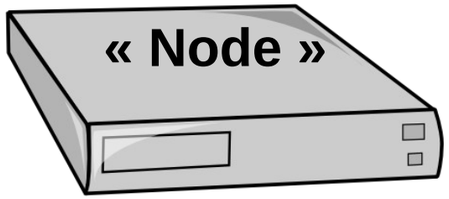


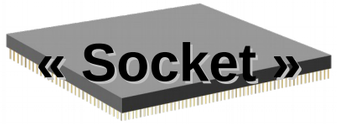
Socquette

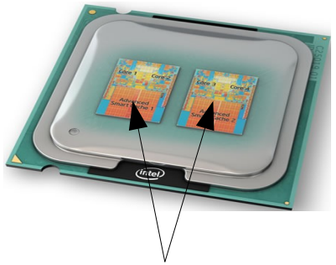
Cœur … Jarretière !

Voilà ce que cela signifie dans le langage des informaticiens !

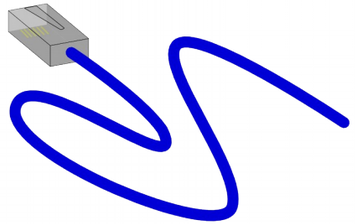






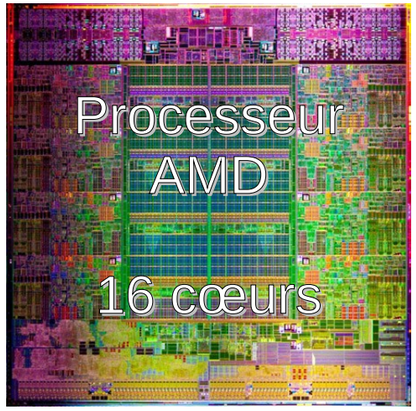


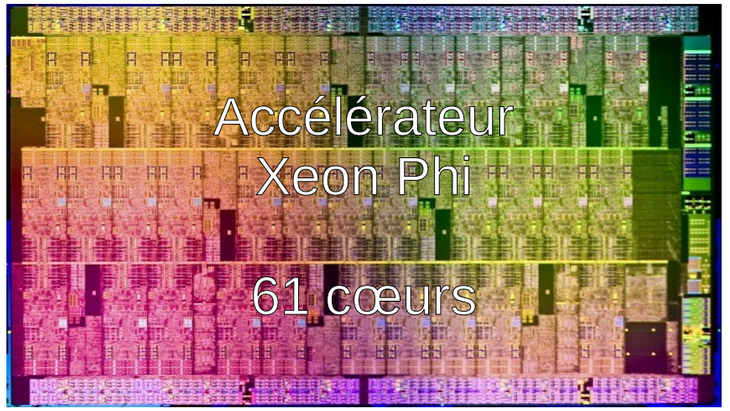
« Cores »



Après l'inflation en fréquence, la multiplication des cœurs...

- La fréquence : le compte-tour de l'ordinateur
  - 1989-1999 : x100 (4 à 400 MHz), 1999-2004 : x10 (0.4 à 3.8 GHz)
  - 2004-2009 : x0.5 (3.8 à 2 GHz), 2009-2014 : x 1.5 (2 GHz à 3.5 GHz)
- La multiplication des cœurs : timide démarrage avant l'explosion
  - 2005 : 2 cœurs, 2010 : 4 cœurs, 2013 : 16 cœurs

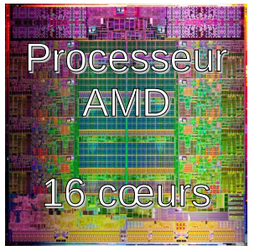


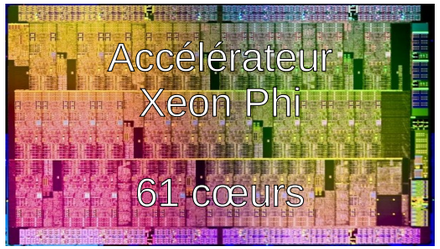


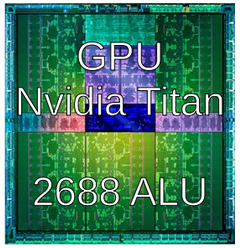
Comment exploiter efficacement ces architectures matérielles ?

Paralléliser, c'est distribuer les tâches
Comment le faire en Python ?

- Sur une **même machine** : des tâches & un pool de cœurs
  - Module multiprocessing
  - Module **PyOpenCL**
- Sur **plusieurs machines** : 1 maître, X esclaves, des messages
  - Module mpi4py
- Sur un **accélérateur Intel** : une carte, des centaines de threads
  - Module multiprocessing & Module mpi4py
  - Module **PyOpenCL**
- Sur un **processeur graphique** : une carte, des milliers d'UAL
  - Module PyCUDA pour les cartes Nvidia
  - Modules **PyOpenCL** pour les cartes Nvidia, AMD/ATI, (& Intel)







Comment ?
Un « Hello World » en OpenCL...

- Définir deux vecteurs en « Ascii »
- Les dupliquer en deux gros vecteurs
- Les ajouter avec un noyau OpenCL
- Imprimer à l'écran le résultat (oui, c'est tout...)

Addition de
2 vecteurs a+b =c
pour tout n :
c[n] = a[n] + b[n]

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

The End

ÉCOLE NORMALE SUPÉRIEURE DE LYON

Comment ?
« Hello World » OpenCL en C...

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

Overkill !

Noyau OpenCL

Nombre de Lignes De OpenCL

Appel Noyau

ÉCOLE NORMALE SUPÉRIEURE DE LYON

Comment ? « Hello World » OpenCL en Python

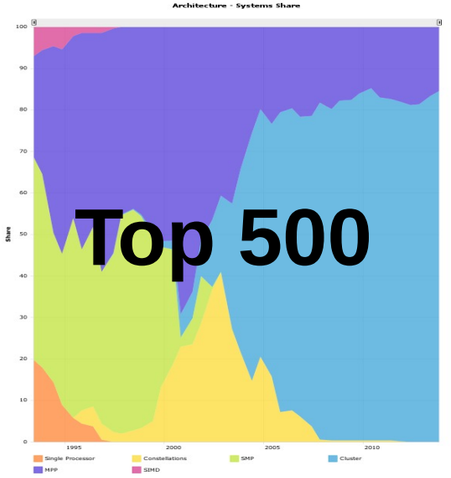
```
import pyopencl as cl
import numpy
import numpy.linalg as la
import sys
OpenCLSource = """
__kernel void VectorAdd(__global int* c, __global int* a,__global int* b)
{
      // Index of the elements to add
      unsigned int n = get_global_id(0);
      // Sum the n th element of vectors a and b and 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)
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");
```

ÉCOLE NORMALE SUPÉRIEURE DE LYON

Comment ?
« Hello World » OpenCL : la pesée

- Sur l'implémentation OpenCL précédente :
  - En C : 75 lignes, 262 mots, 2848 octets
  - En Python : 51 lignes, 137 mots, 1551 octets
  - Facteurs : **0.68**, **0.52, 0.54** en lignes, mots et caractères...
- Programmation OpenCL :
  - Contexte de programmation difficile :
    - *Ouvrir le carton est plus difficile que de monter le meuble...*
    - Exigence d'une API simplifiant l'appel d'OpenCL
  - Pas de compatibilité entre les SDK de AMD et Nvidia
    - Chacun a développé son API : portabilité des codes difficile...
  - Une solution, sinon « LA » solution : Python

Le contexte *High Performance Computing*
Des nœuds, du réseau et du stockage...



- Des évidences, d'autres moins...
  - La majorité des « machines » de HPC sont des grappes :
    - Pas de mémoire partagée, espace de stockage locale non partagé
  - Un stockage local assure une performance maximale
    - Disque local : 150 MB/s, Stockage sur Gigabit Ethernet, 125 MB/s
- Pour Equip@Meso au PSMN une question : quel *scratch* distribué ?
  - *Scratch* : espace de stockage temporaire pour les calculs
  - Une **nécessité** : partager le scratch entre tous les nœuds indépendants
  - Une **contrainte** : le scratch le plus rapide & le plus réactif possible
  - Un **vœu** : une solution facile à déployer, administrer, étendre

Exemple de paillasse numérique
Qualification de GlusterFS & usage de ClusterShell

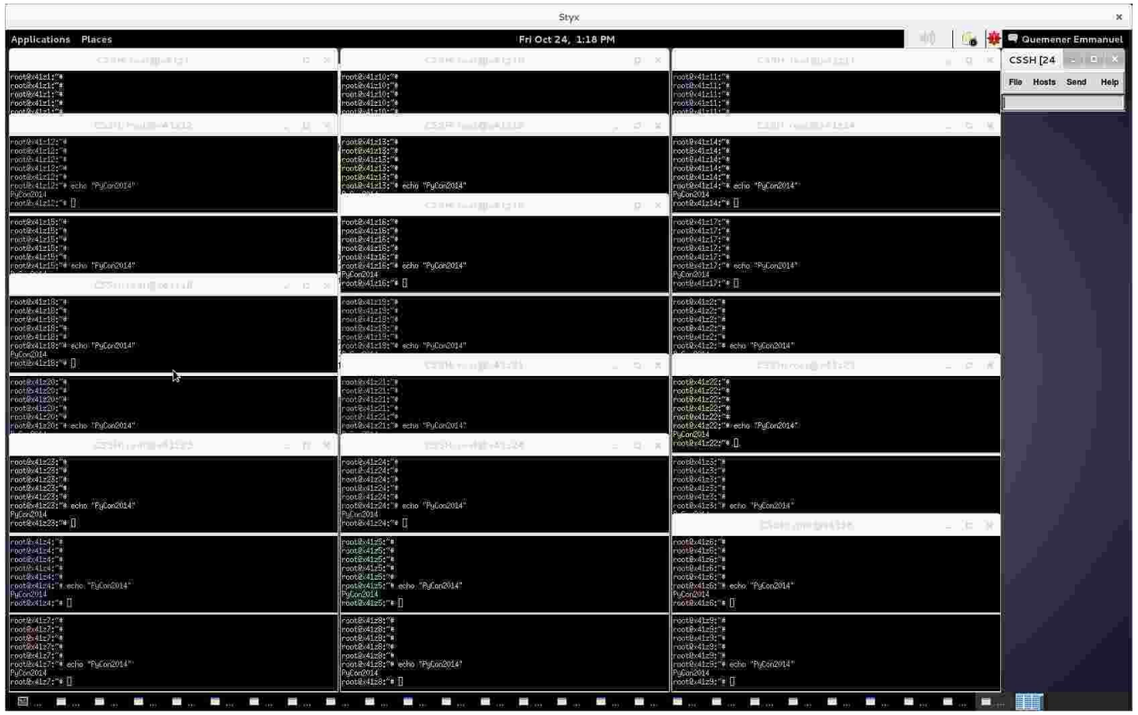
- **Objectif** :
  - Évaluation de GlusterFS comme /scratch de haute performance
- **Plate-forme d'expérimentation** : 20 nœuds + infrastructure
  - 20 nœuds Sandy Bridge 2x8 cœurs avec 64 GB de RAM
  - Un système **SIDUS** Debian Wheezy
  - Interconnexion InfiniBand FDR 56 Gb/s
  - **Pas de latence disque : RamDisk BRD/Ext2 et TMPFS de 60 GB**
  - 10 paires GlusterFS : 1 serveur sur RamDisk, 1 client
  - Usage de IOZone3 : 13 tests de lecture/écriture
  - 20 expériences pour un échantillon statistique représentatif

Comment configurer son infrastructure

- Processus :
  - Configuration des cibles GlusterFS
    - Création des volumes TMPFS locaux
    - Création du volume GlusterFS
  - Accrochage des prédateurs GlusterFS
    - Montage du volume distant
  - Lancement du test IOZone
  - Décrochage des prédateurs GlusterFS
  - Décrochage des cibles GlusterFS
- Au début, un script Bash, tournant rapidement à la catastophe
  - Exploitation d'un outil

Plusieurs méthodes pour la configuration
La « moche », la « brute », la « bonne »

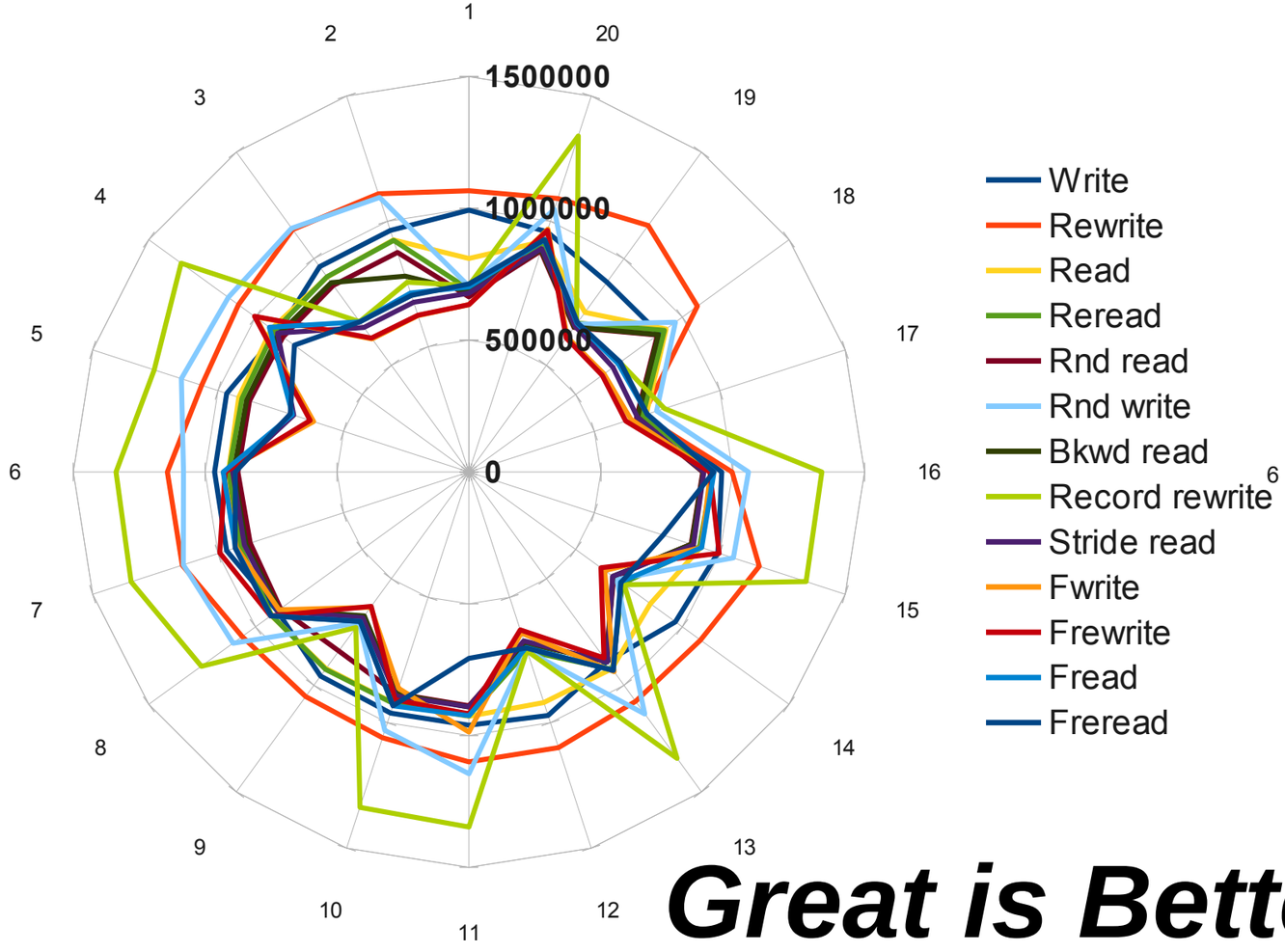
- La « Brute » : **ClusterSSH**
  - Un terminal par serveur, Un terminal par client
  - Des copier/coller pour tout !



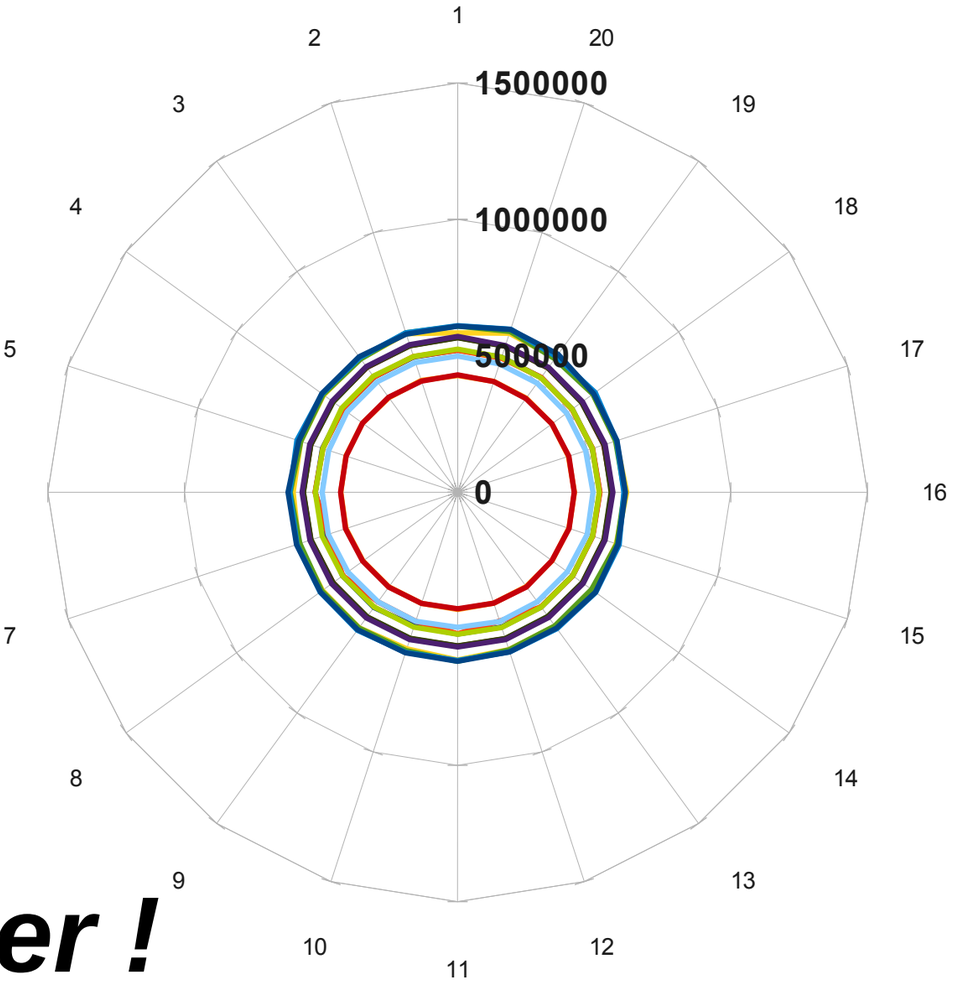
- La « moche » : **ClusterShell** Bash
  - Des « pools » serveurs & clients
  - Des commandes par ligne préfixées
  - **Problème** : passage de variables
- La « bonne » : **ClusterShell** Python
  - Des « Instances » serveurs & clients
  - Des méthodes pour les commandes
  - **Avantage** :
    - Scénarii de test
    - Récupération directe des sorties

Jour 1 : lancement des tests & surprises !
Sur les vitesses d'exécution I/O

Nœud 11 vers Nœud 1



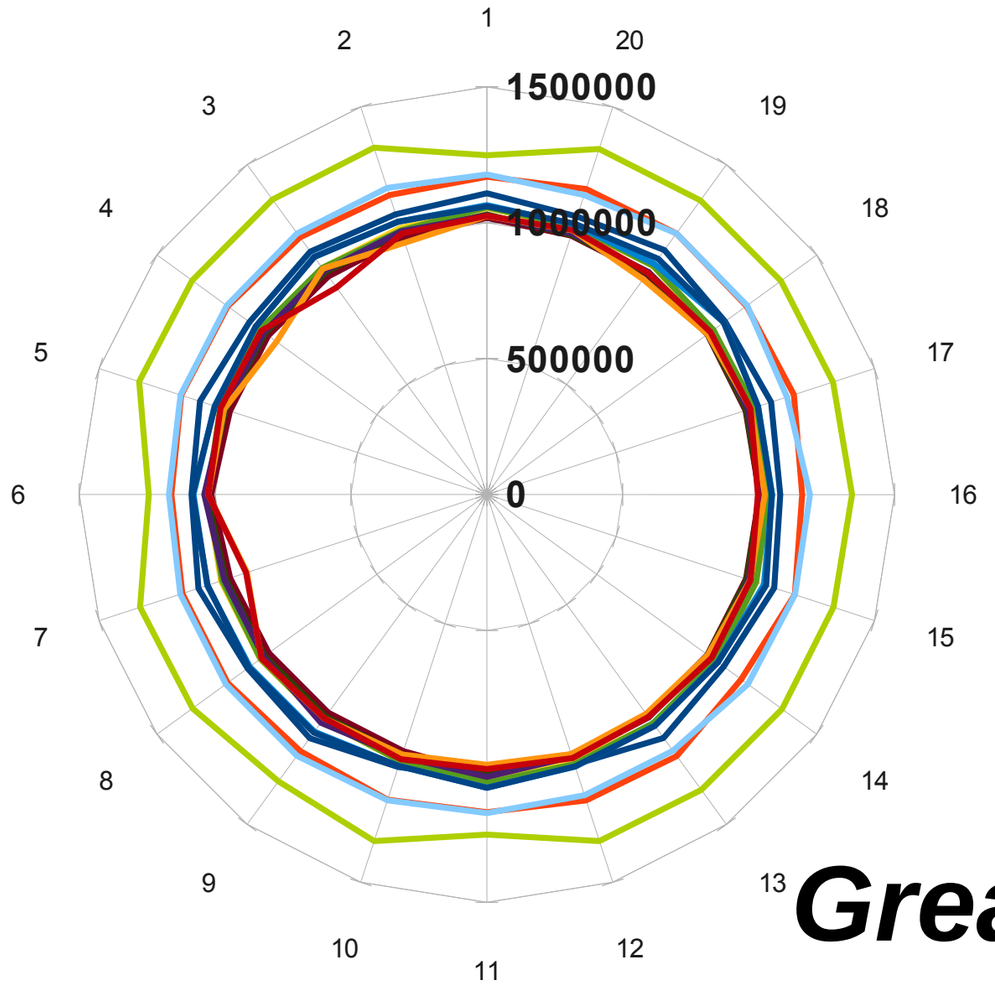
Nœud 13 vers Nœud 3

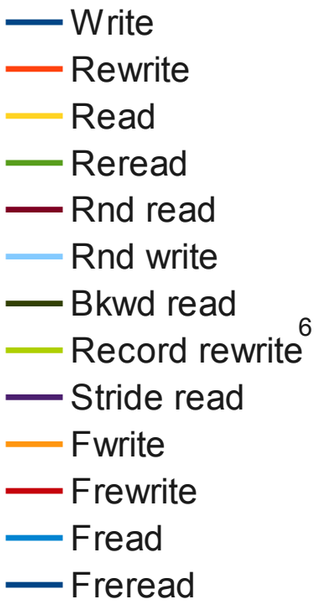


Great is Better !

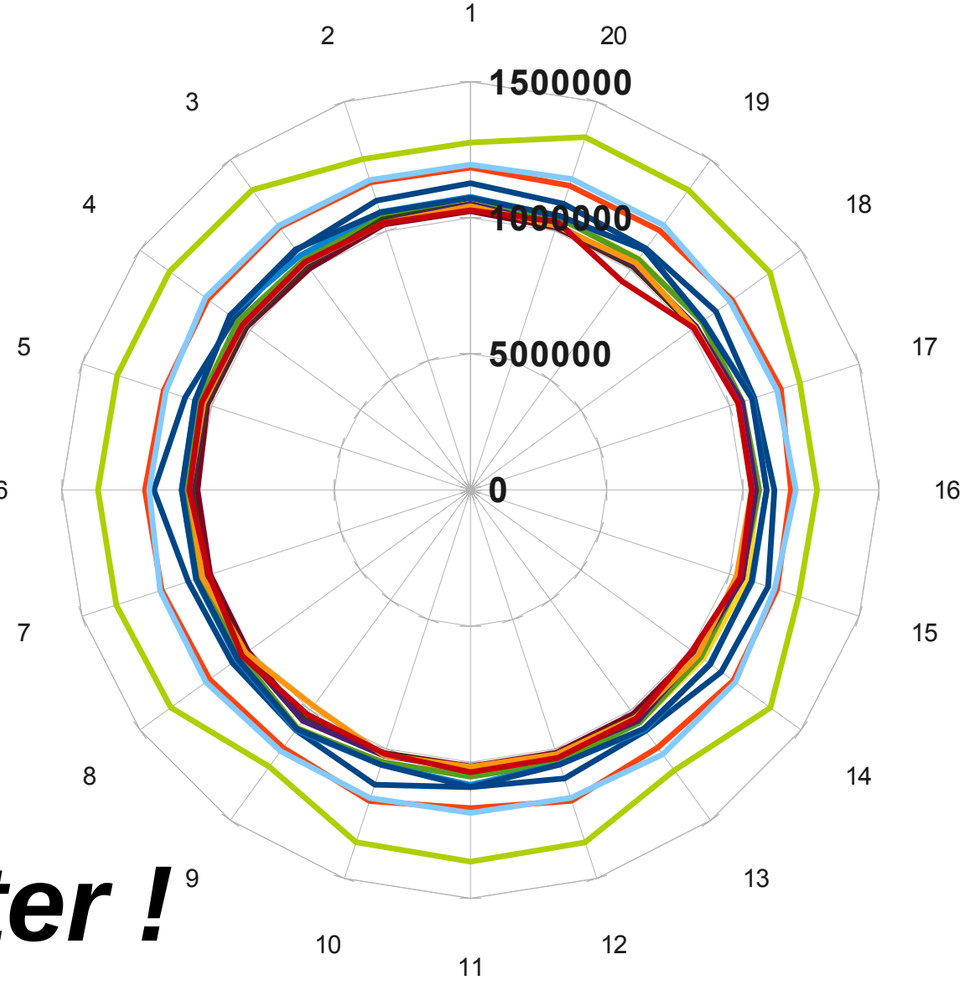
Jour 2 : et sur les deux couples du début ? Sur les vitesses d'exécution

Nœud 11 sur Nœud 1





Nœud 13 sur Nœud 3



Great is Better !

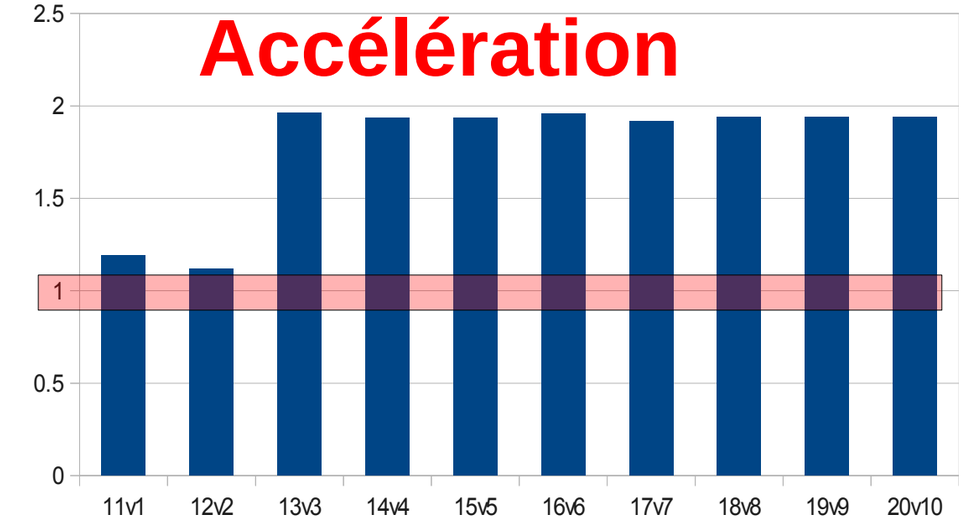
Quel miracle entre jours 1 & 2 ?

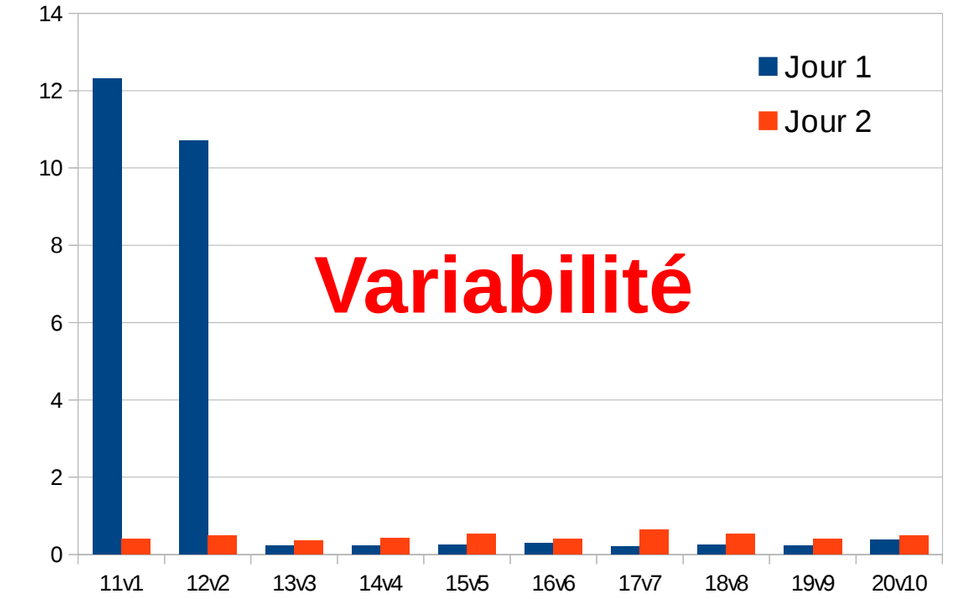
Deux questions : Comment ...

- ... multiplier par 2 la vitesse ?
- ... diviser entre 20/30 sa variabilité ?

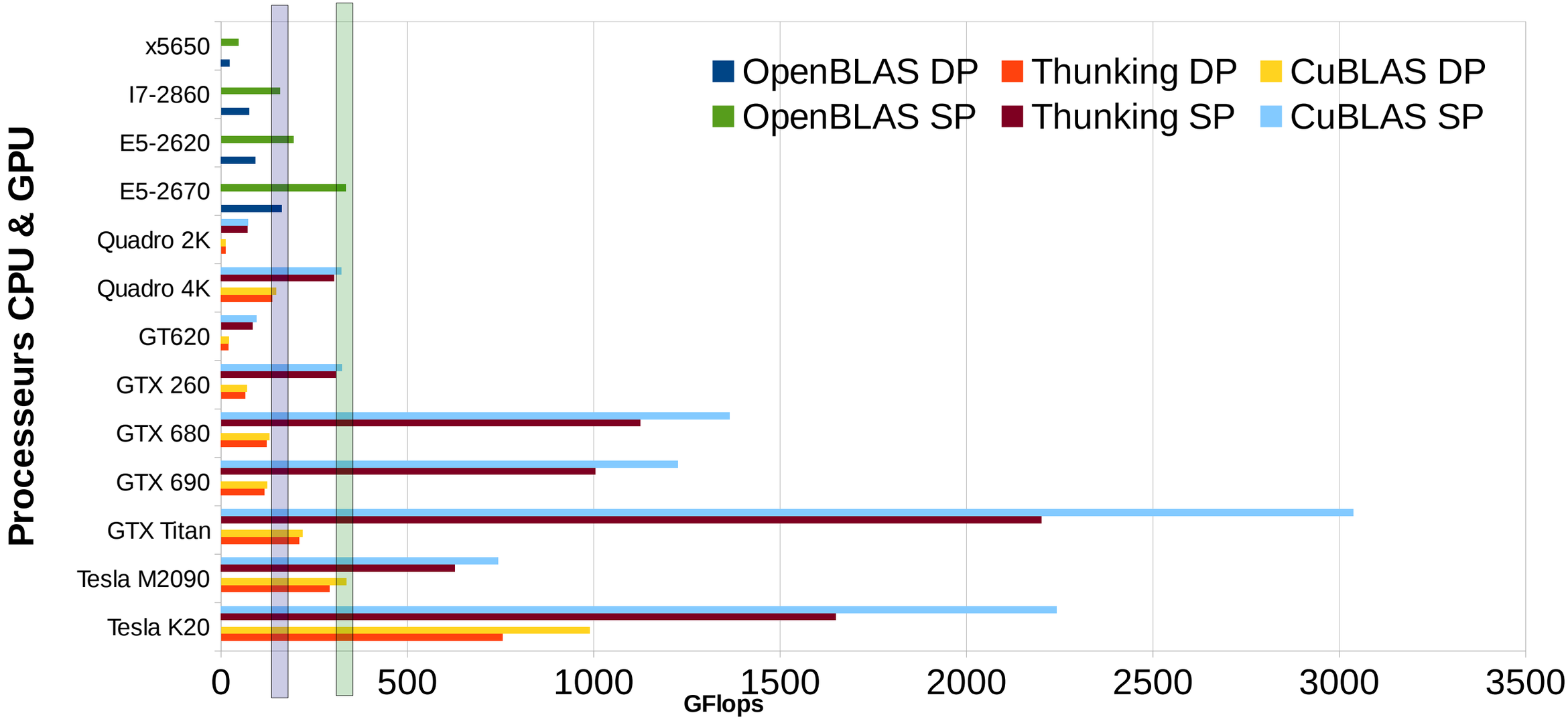
La réponse :

- Optimiser le réseau ? Non
- Optimiser les noyaux des OS ? Non
- Changer le BIOS ? **OUI !!!**
  - BIOS de 1 & 2 en Max Performance
  - BIOS de 3 à 20 par défaut
- Solution : BIOS en Max Perf !



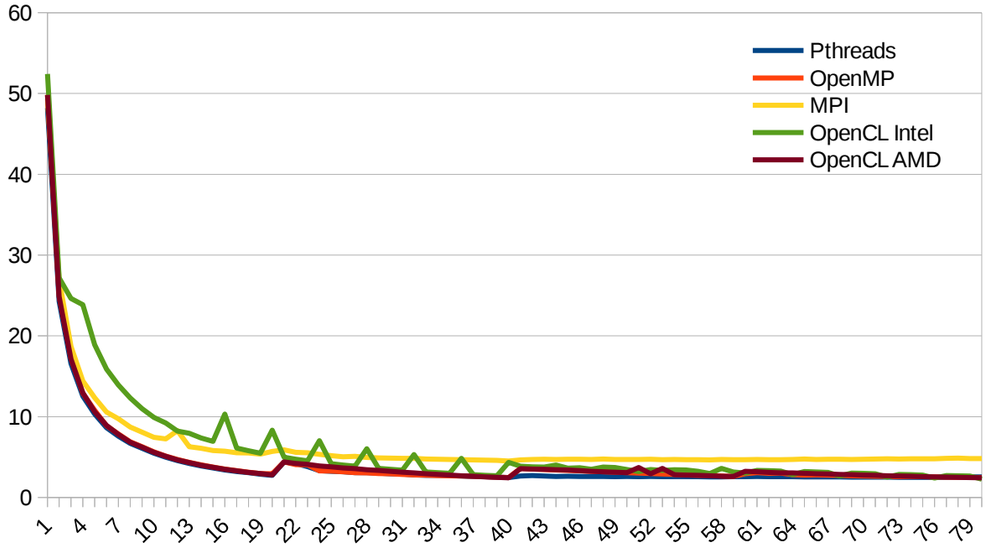


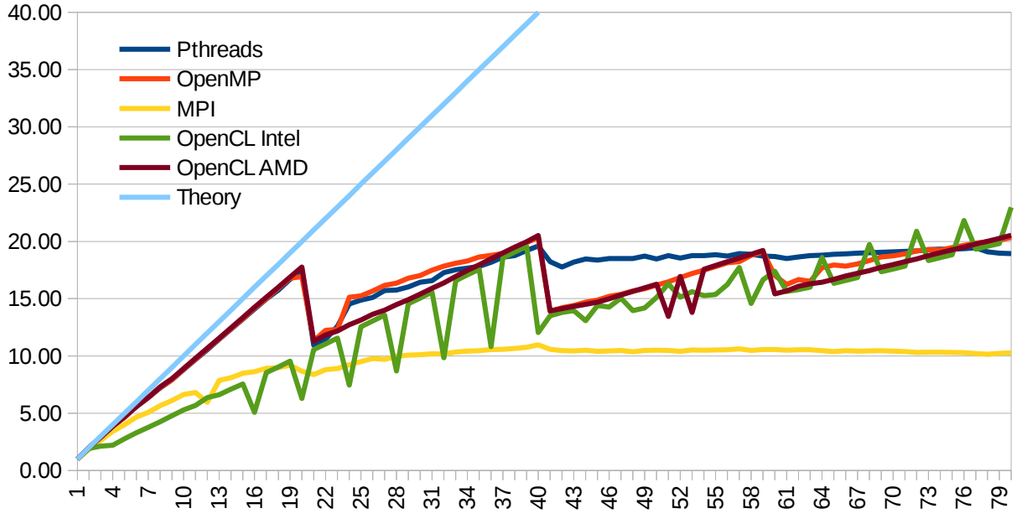
Pourquoi un test Pi Monte Carlo ?
En sortir de l'hégémonie du BLAS !



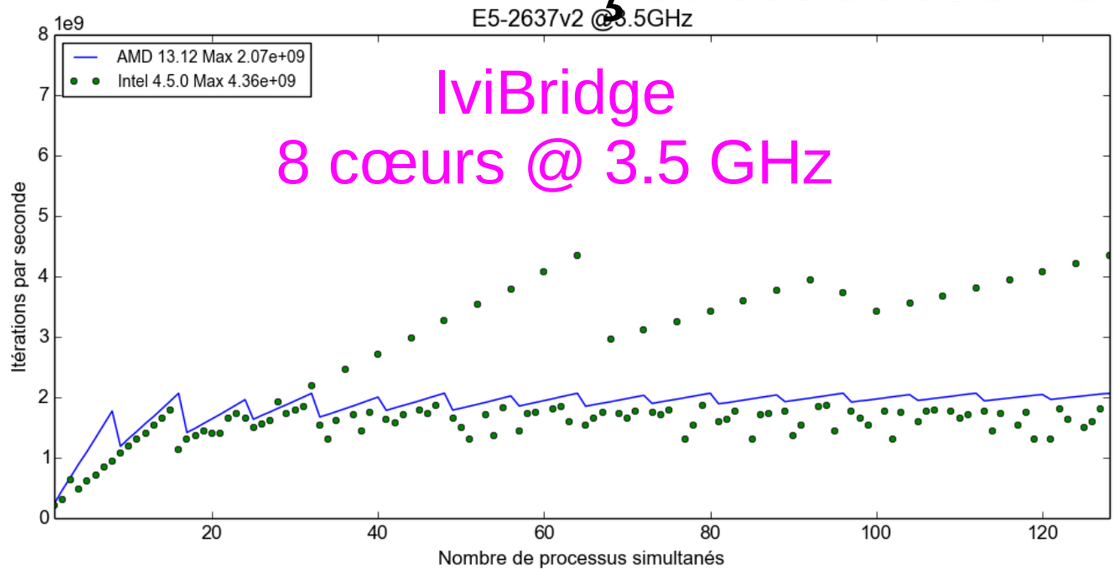
Le parallélisme dans tous ses Au commencement, il y avait le CPU

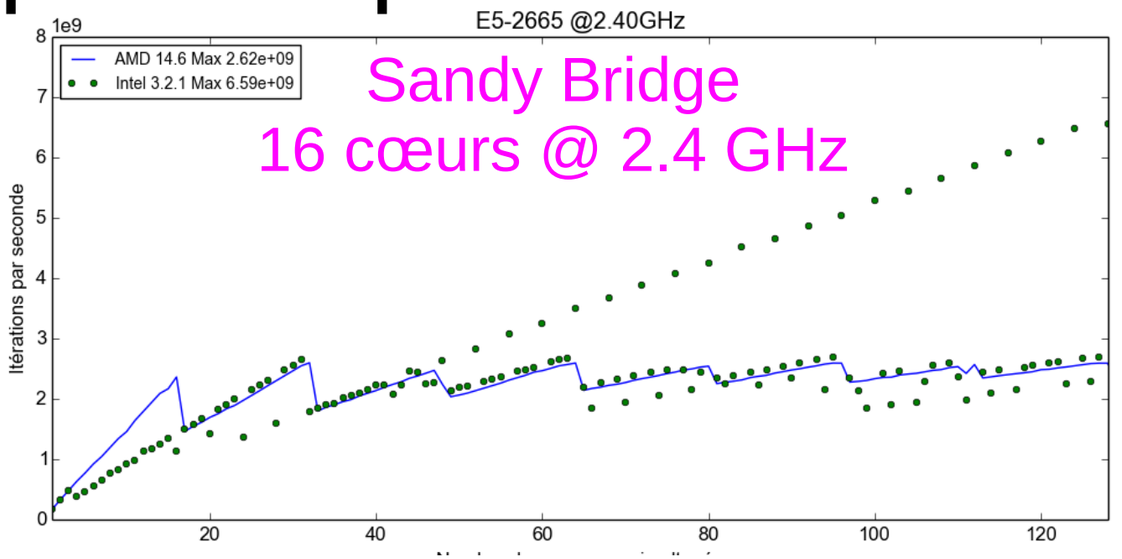
- Un serveur : Dell R620
  - Bi-socket, décacœur : 20
  - Mode HyperThreadé activé : 40
- Implémentations parallèles
  - MPI en C
  - OpenMP en C
  - Pthreads en C
  - OpenCL en Python
    - OpenCL par AMD
    - OpenCL par Intel
- Finalement, pas mal OpenCL !

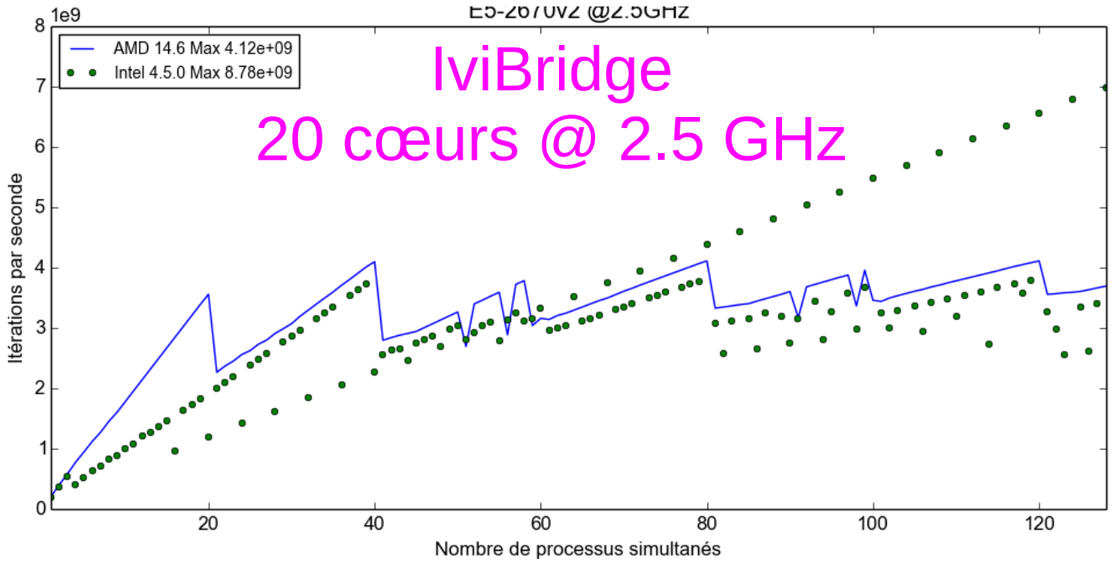


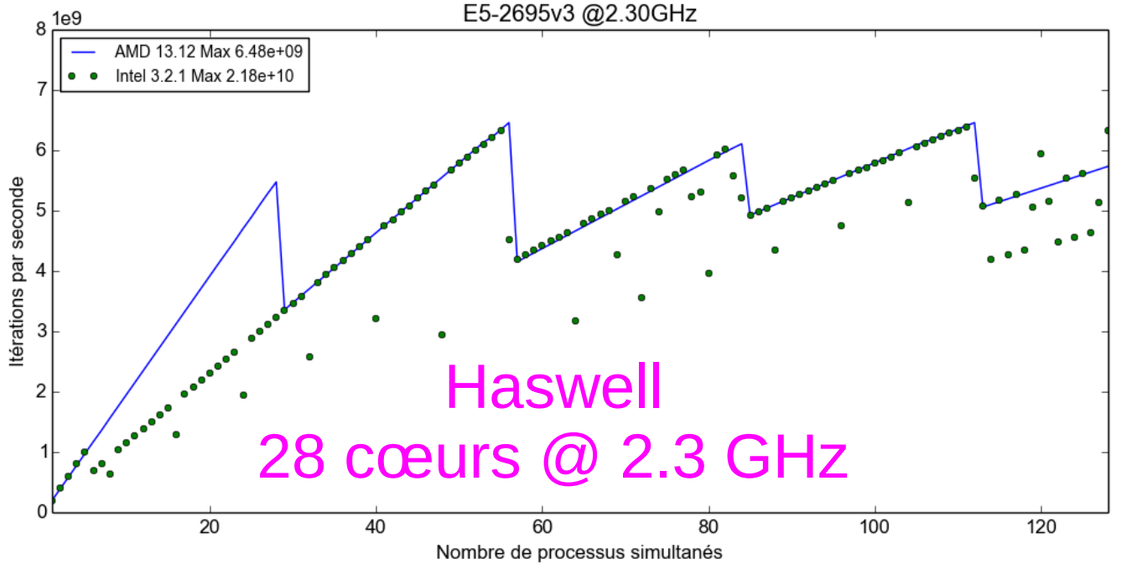


Et au delà de 80 processus, ça monte encore pour l'OpenCL Intel

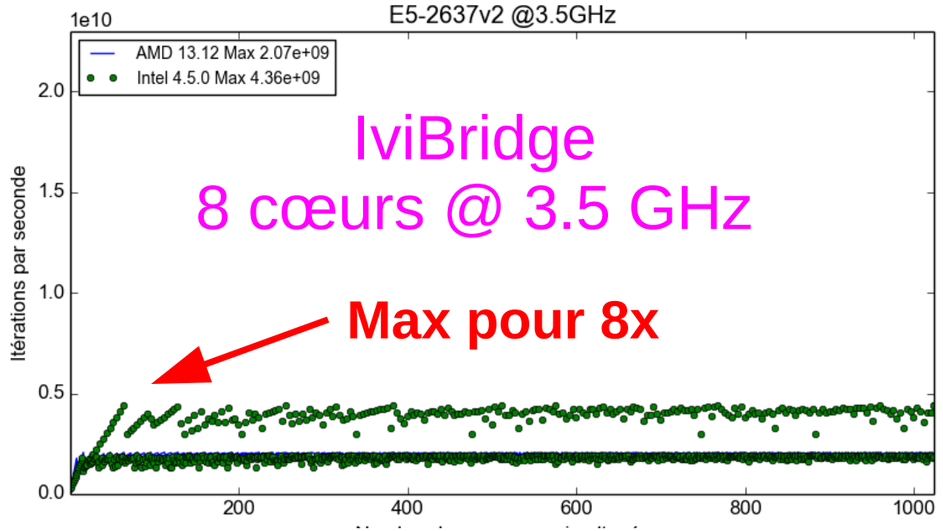


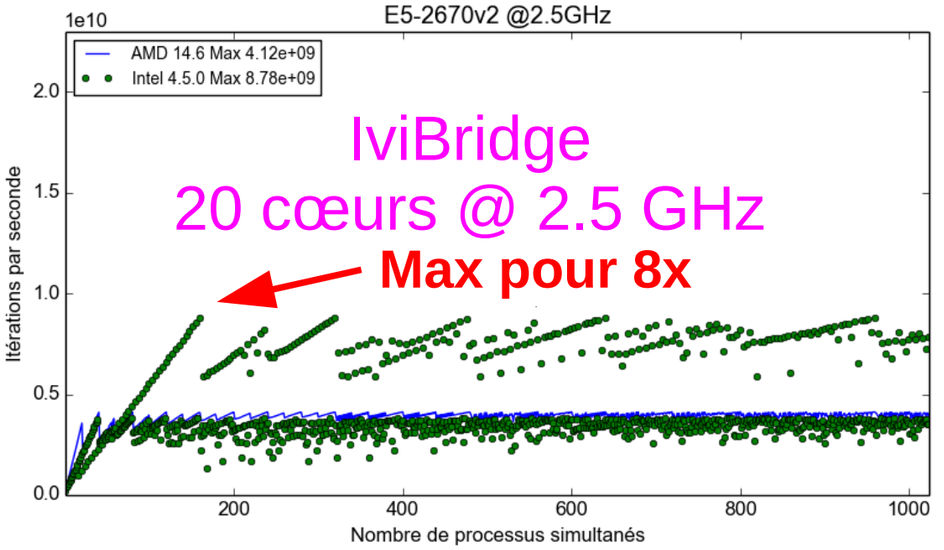


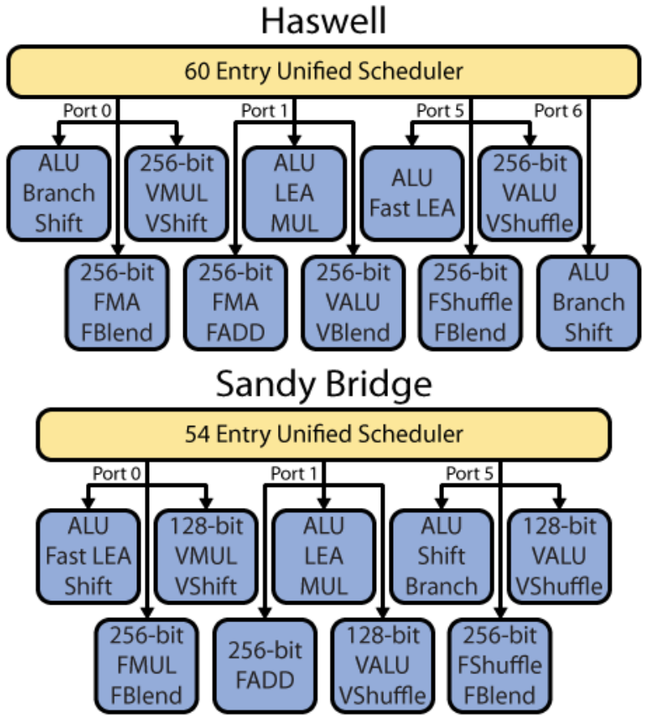




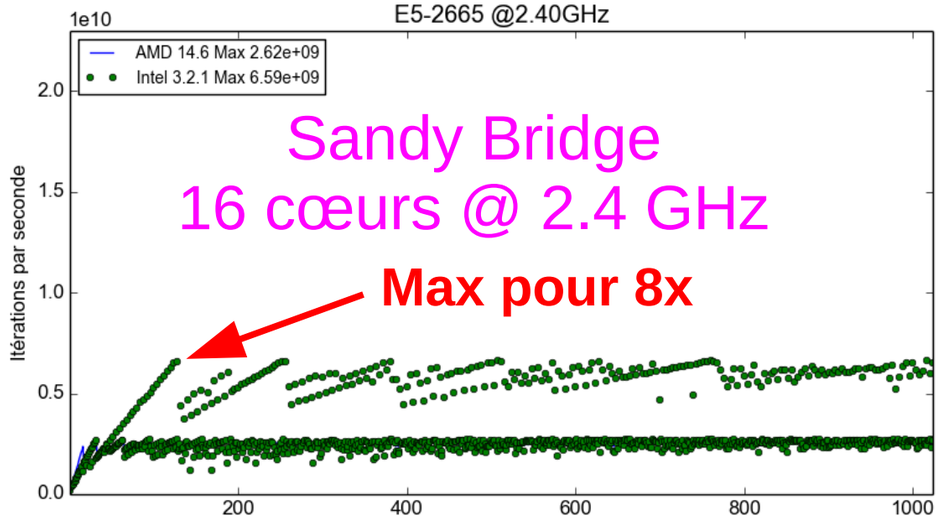
Mais jusqu'où cela va-t-il monter ? Jusqu'à 1024 processus simultanés

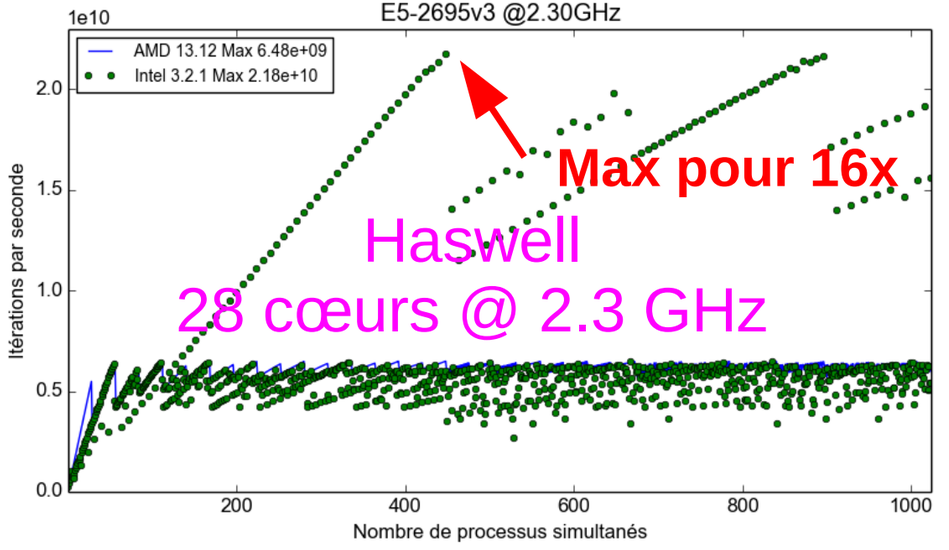




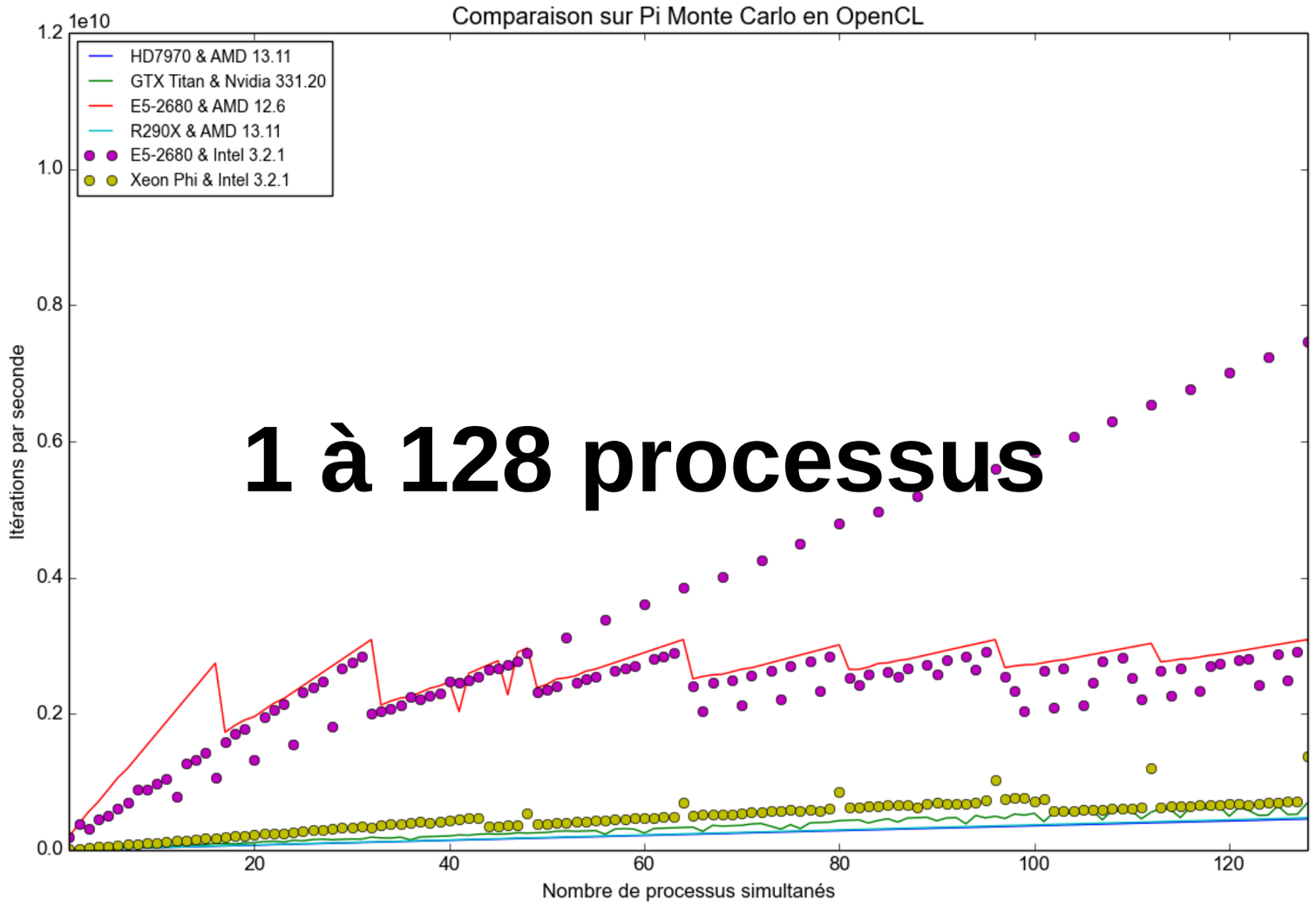


- Intel x2 à x3 vs AMD
- Périodicité de 4
- Performance Maximale :
  - x8 pour (Sandy|Ivi)Bridge
  - x16 pour Haswell

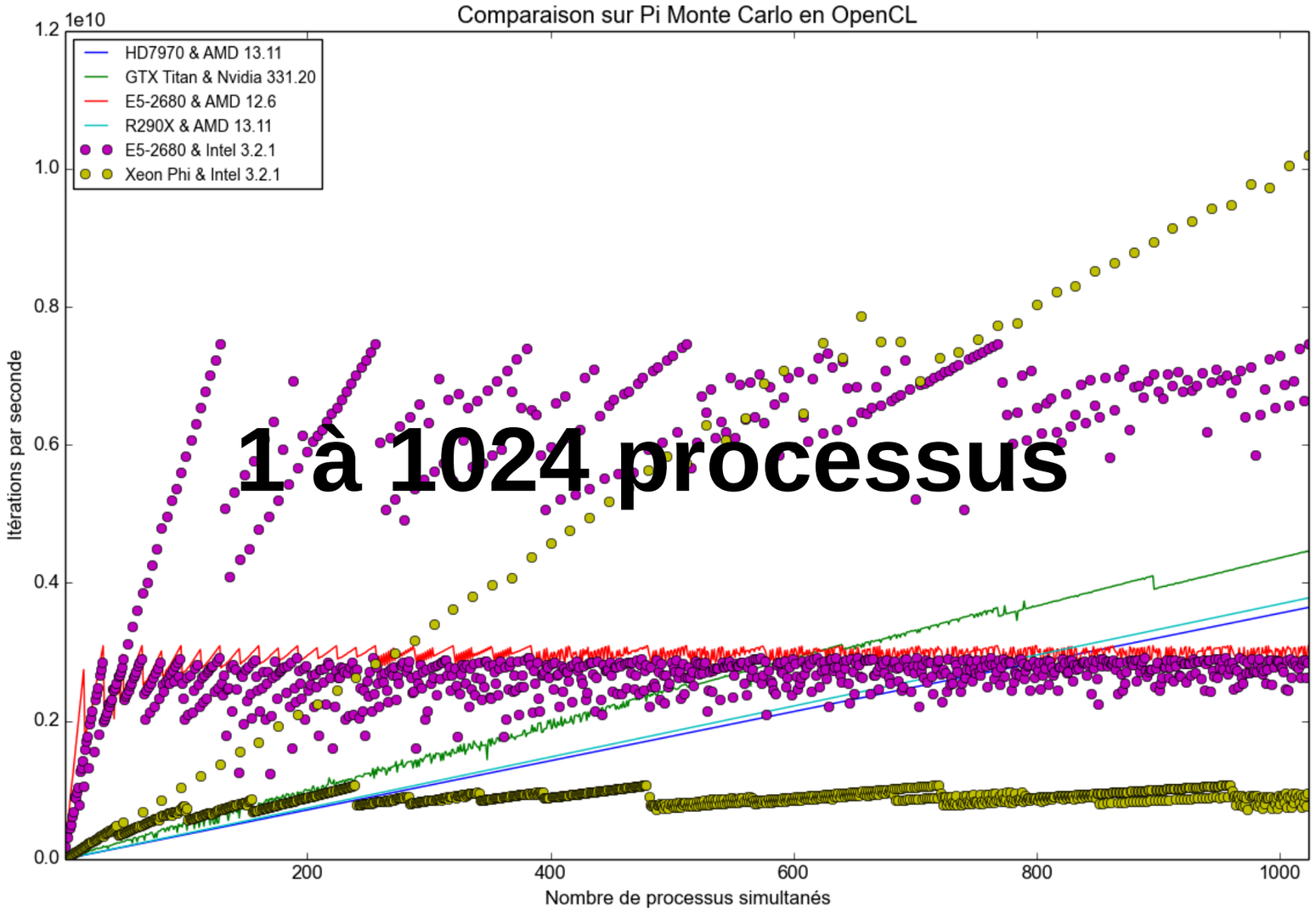




Exploration de large domaine de parallélisme Entre CPU/GPU et accélérateur !



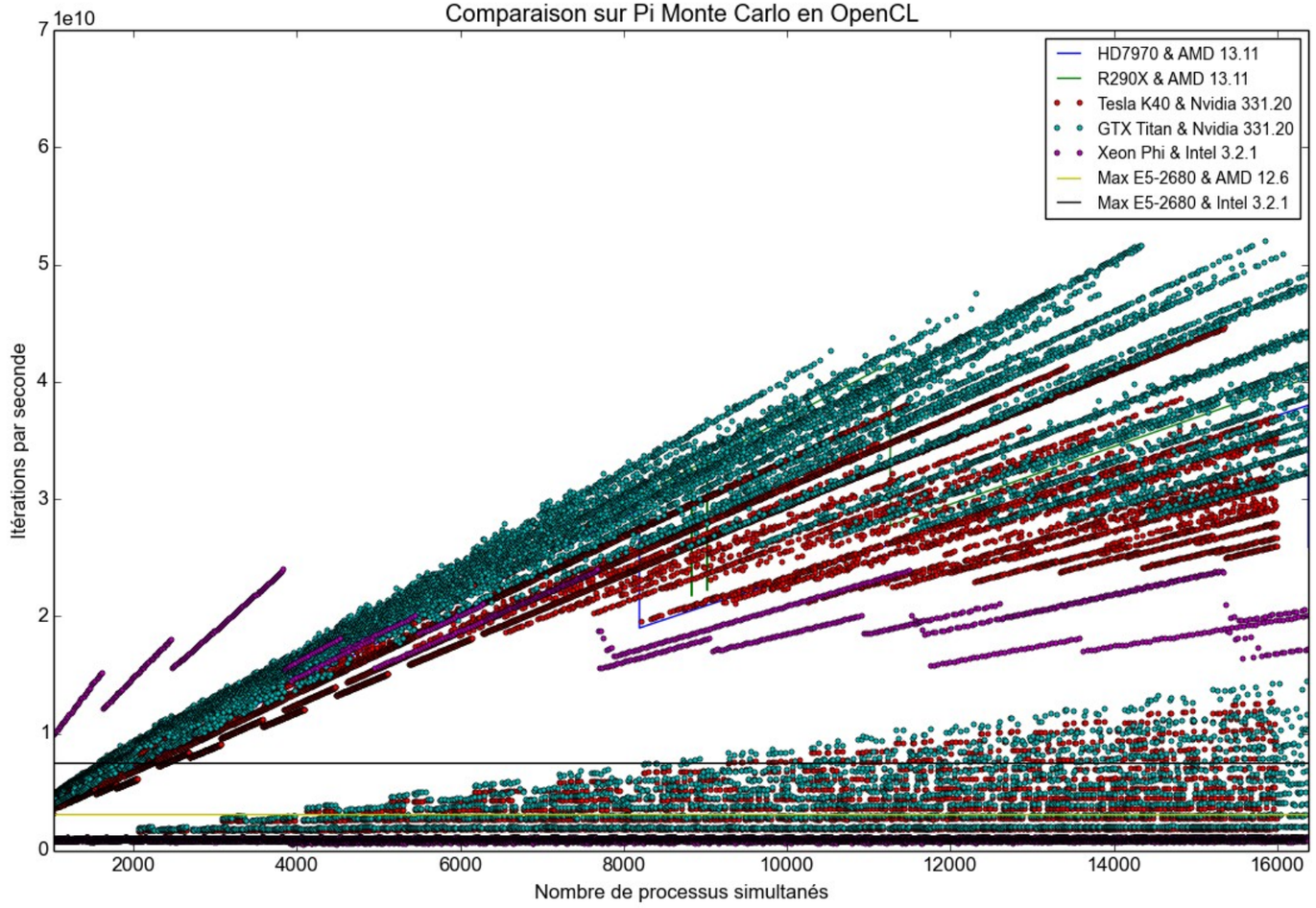
Exploration de large domaine de parallélisme Entre CPU/GPU et accélérateur !

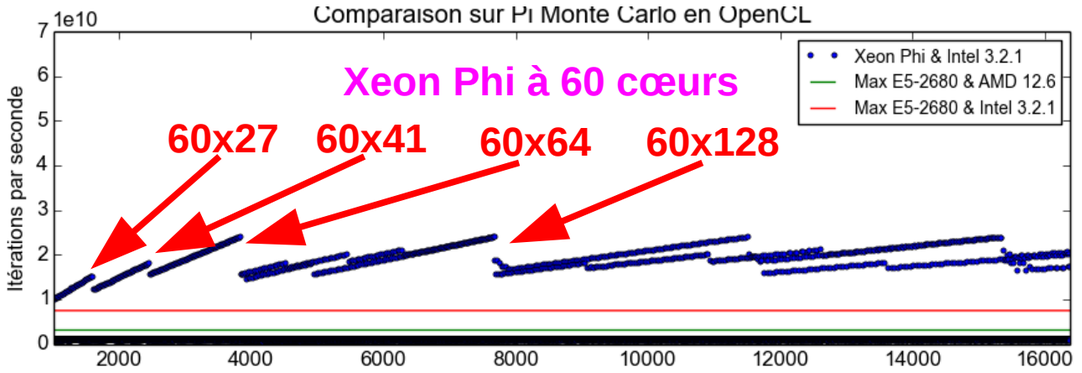


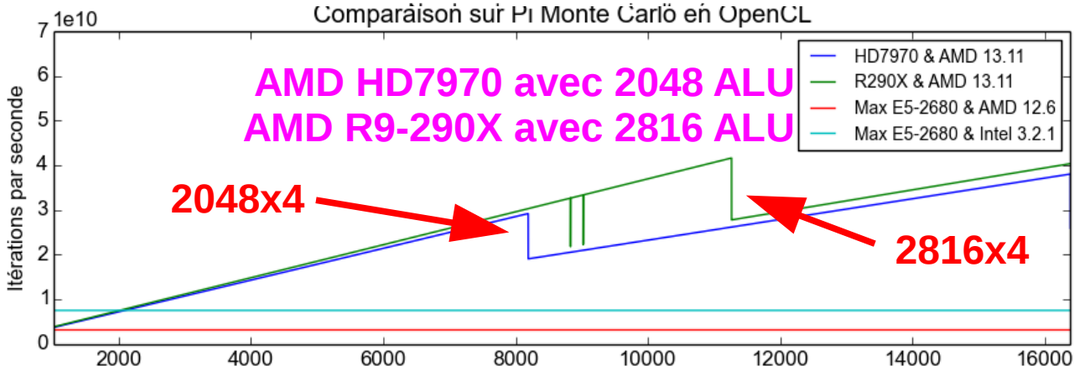
1 à 1024 processus

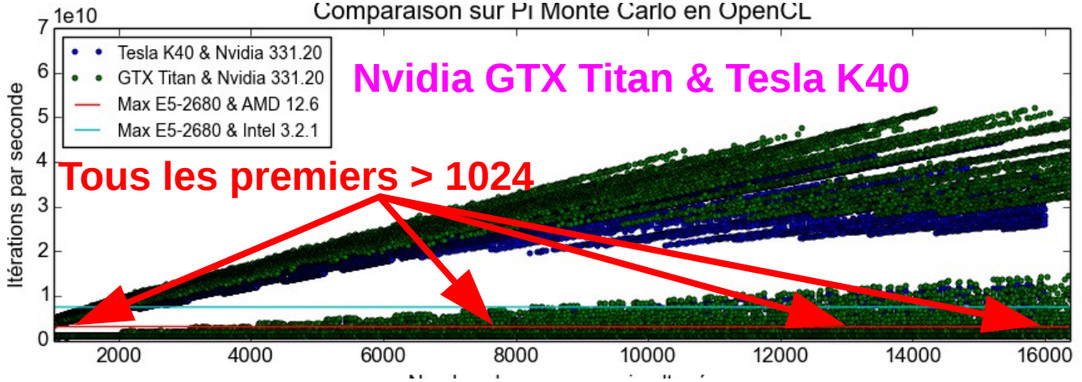
Exploration de large domaine de parallélisme Entre CPU/GPU et accélérateur !

1024 à 16384 processus



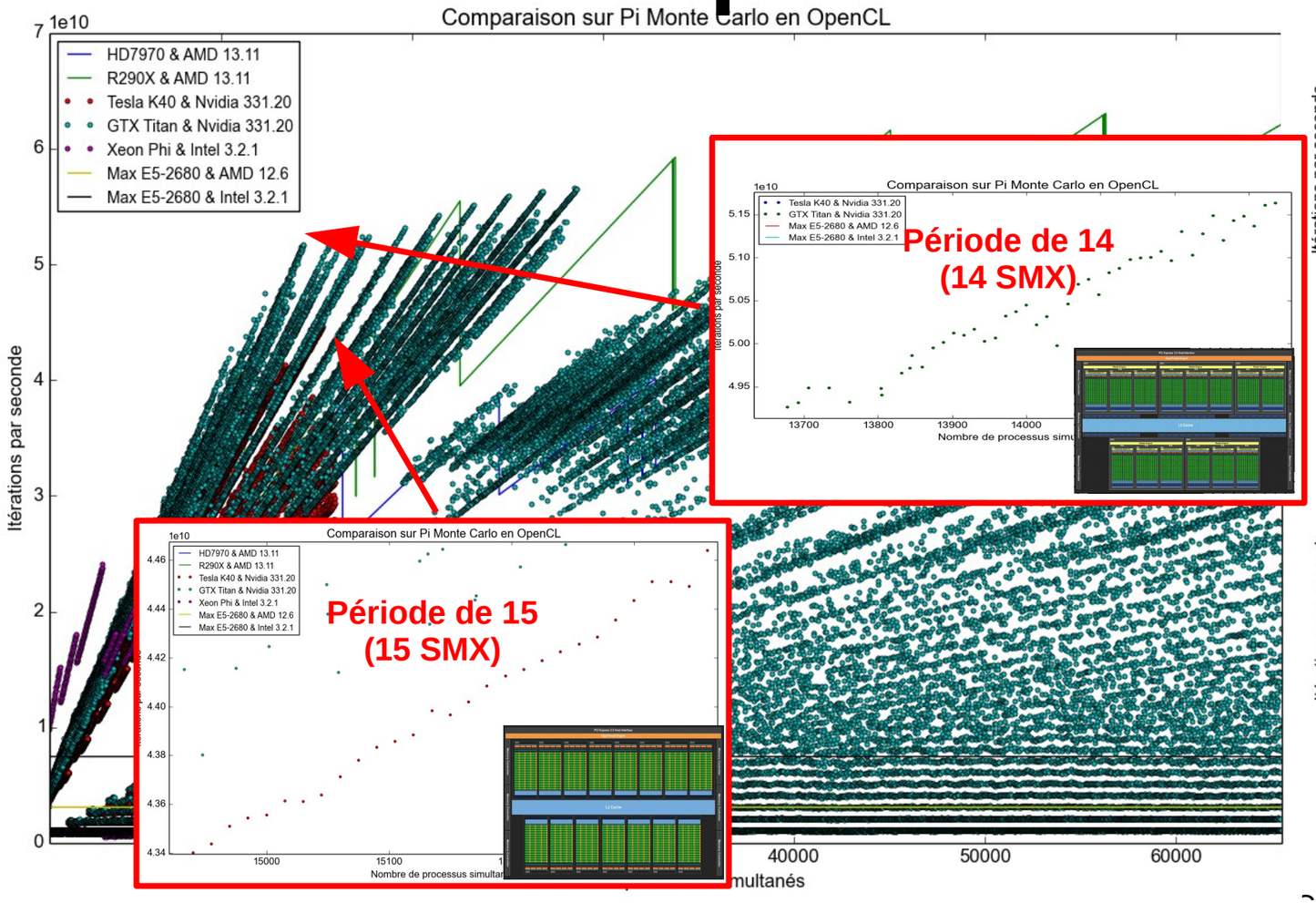


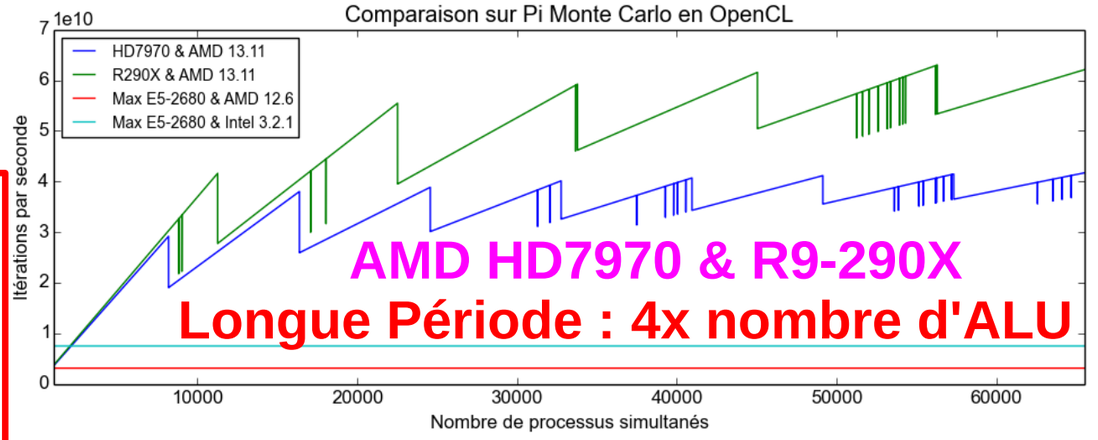


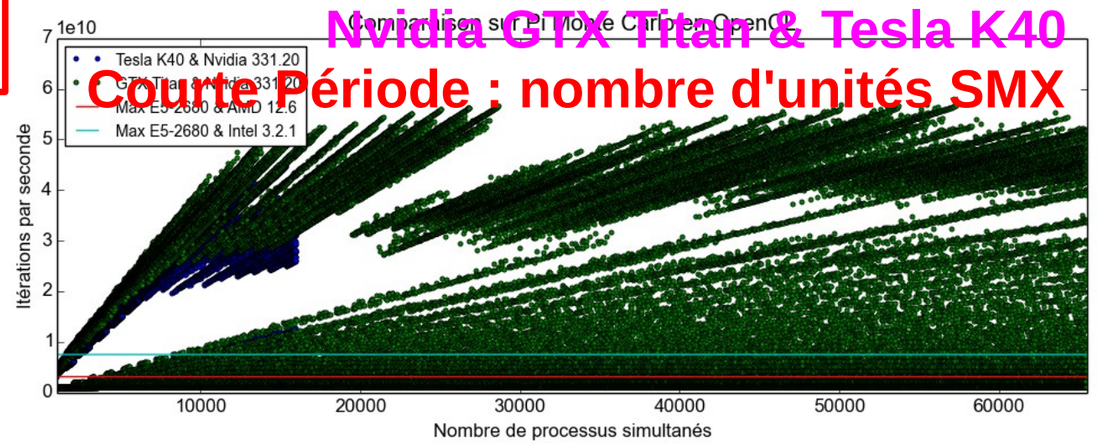


Exploration de large domaine de parallélisme Entre CPU/GPU et accélérateur !

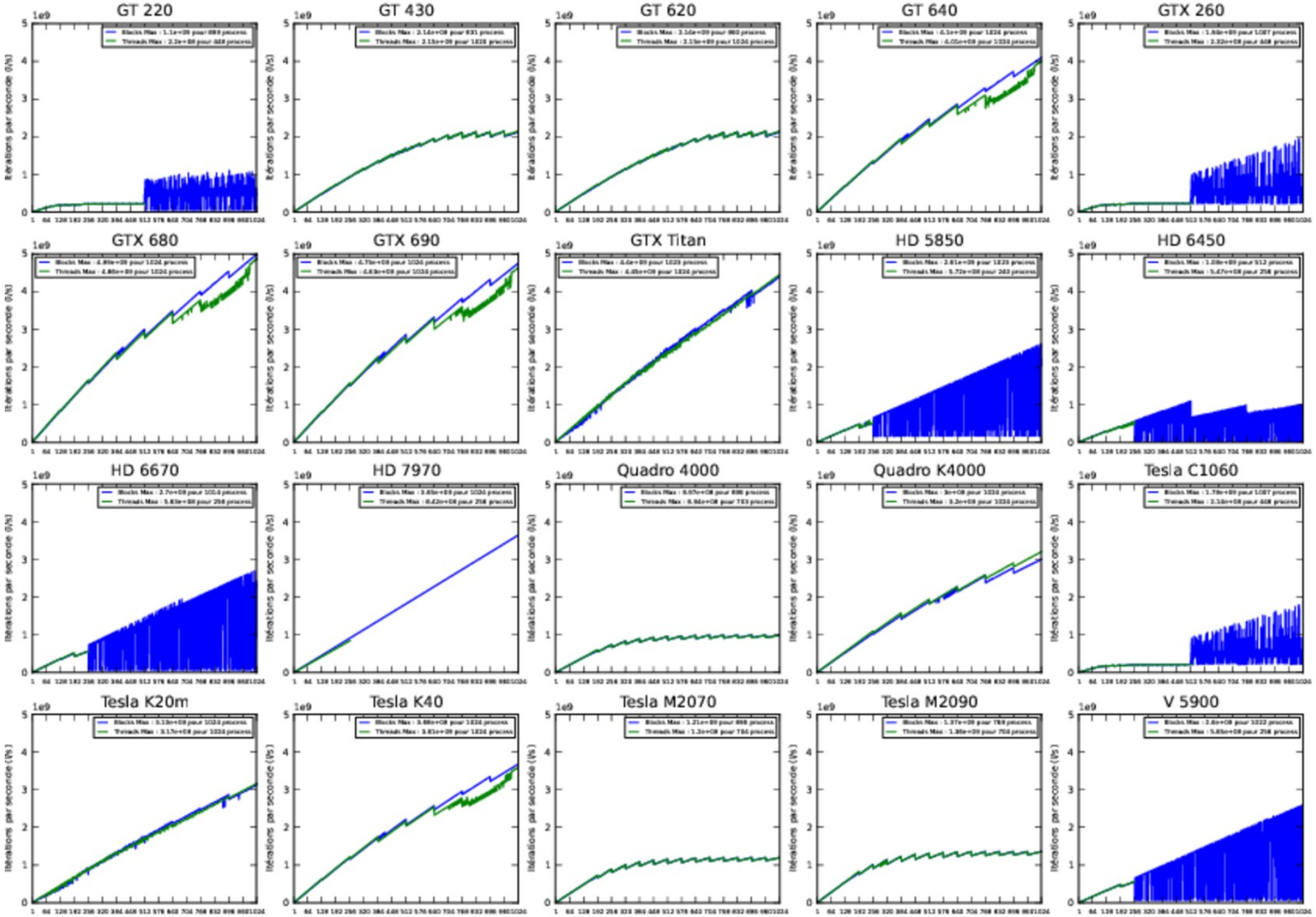
1024 à 65536 processus







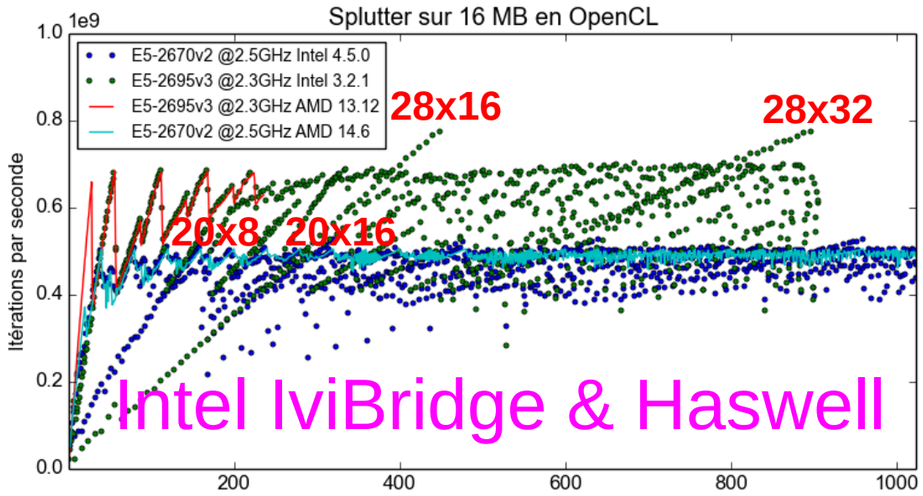
Comparaison « visuelle » de performances
Enveloppe de Parallélisme des GPUs

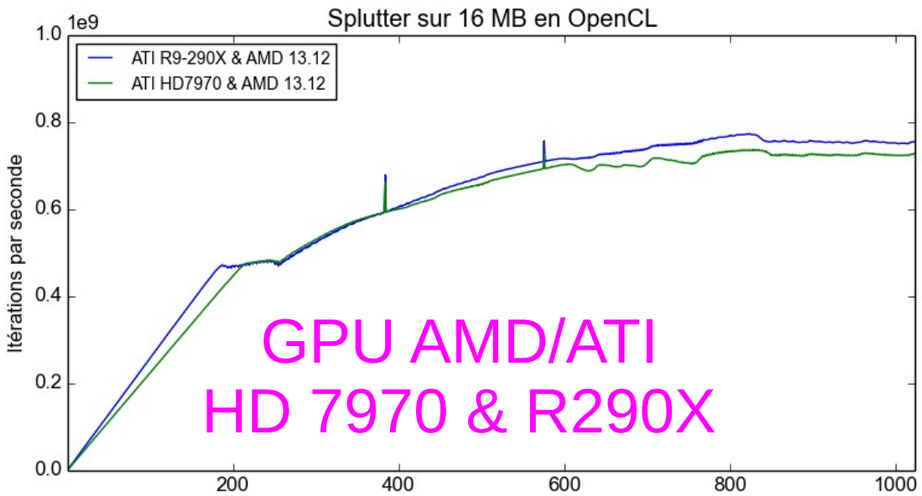


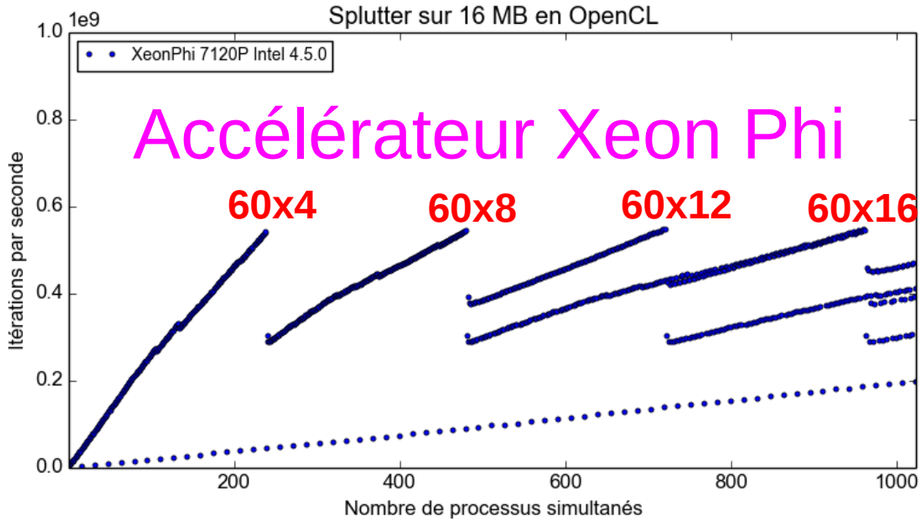
Ok, c'était des tests ALU. Et la mémoire ? Le code *Splutter* !

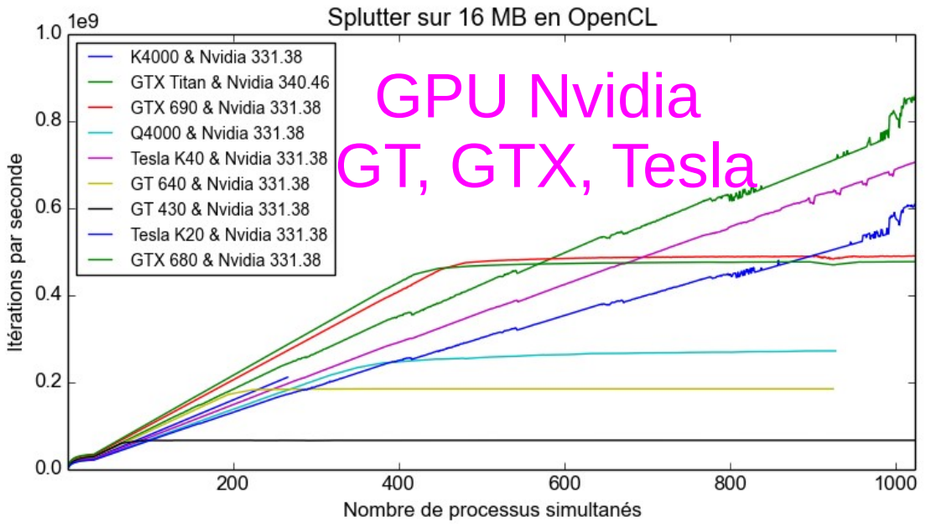
- Objectif : bombarder un espace avec des processus indépendants
  - Attention à la cohérence sur l'accès mémoire : si aucun contrôle, perte...
  - Attention à la limitation des « threads » : 256 ou 1024, coût synchronisation
  - Tests trop restrictifs et peu « linéaires » : autre voie, pire scénario
- Choix : exploitation d'opérations « atomiques »
  - Algorithme de postillonnage :
    - Réservation d'un espace mémoire de X entiers 64 bits
    - Pour chaque processus : boucle de Z itérations
      - Tirage aléatoire d'un entier Y sur 32 bits non signé par la méthode MWC :
      - Incrément de 1 à l'adresse de Y % X
    - A la fin, statistique de distribution et temps de calcul

Splutter ou le stress mémoire en parallèle CPUs, Accélérateur, GPUs









Quel intérêt de ces expériences ? Quel intérêt de Python

- **Pour les expériences :** la nécessité du prototypage matériel
  - Valider GlusterFS comme scratch & lever l'importance des réglages BIOS
    - Approche réexploitable de la fusion d'espaces RAM pour *scratch* très haut débit
  - Étudier des hauts degrés de parallélisme sur tous les MyriALU
    - Pertinence de l'utilisation de codes *KISS* : PiMC (ALU) & Splutter (RAM)
  - Appréhender des conditions d'exploitation optimales et ... contre-intuitives
  - Recycler les protocoles expérimentaux
- **Pour l'utilisation de Python :** l'universalité d'un langage polyvalent
  - **ClusterShell** : du l'OS atomique à l'OS distribué, bonne extension de SIDUS
  - **PyOpenCL** : excellent pour l'exploration, très bon pour l'exploitation

Perspectives
Vers l'arbre des possibles...

- Et si la variabilité (temporelle) ...
  - était LE critère discriminant des *PU ?
- Et si l'approche « intégrateur » …
  - L'emportait sur l'approche « développeur » ?
- Et si la biologie et ses exigences …
  - modifiait profondément le HPC ?
- Et si Python devenait au HPC ...
  - ce que les maths sont à la science