Houches 2016 : 6th School

Computational Physics

GPU : the disruptive technology of 21th century

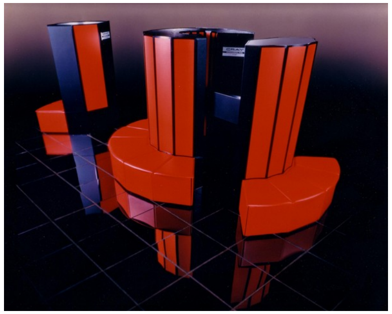
Little comparison between CPU, MIC, GPU

Emmanuel Quémener

A (human) generation before...
A serial B film in 1984

- 1984 : The Last Starfighter
  - 27 minutes of synthetic images
  - 22.5 10^9 operations per image
  - Use of Cray X-MP (130 kW)
  - 66 days (in fact, 1 year needed)

- 2016: on GTX 1080Ti (250 W)
  - 3.4 secondes
  - Comparison GTX1080Ti / Cray
    - Performance : 1 600 000 !
    - Consommation ~ 900 000 000 !



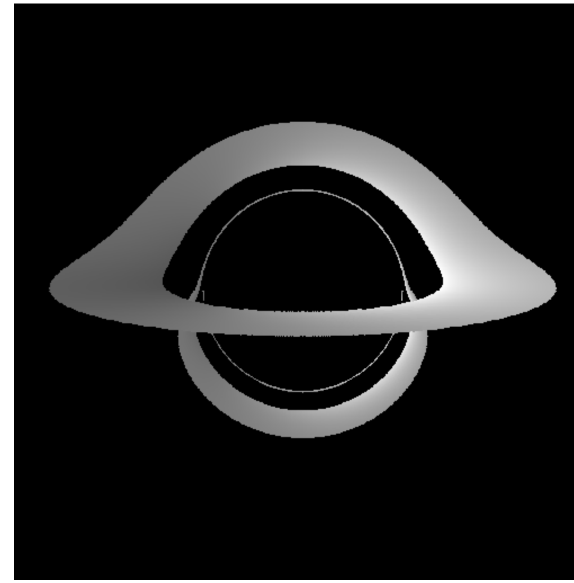
« The beginning of all things »

- « Nvidia Launches Tesla Personal Supercomputer »
- When : on 19/11/2008 Qui : Nvidia
- Where : sur Tom's Hardware
- What : a PCIe card C1060 PCIe with 240 cores
- How much : 933 Gflops SP (but 78 Gflops DP)

What position for GPUs ? The others « accelerators »

- Accelerator or the old history of coprocessors...
  - 1980 : 8087 (on 8086/8088) for floating points operations
  - 1989: 80387 (on 80386) and the respect of IEEE 754
  - 1990 : 80486DX and the integration of FPU inside the CPU
  - 1997 : K6-3DNow ! & Pentium MMX : SIMD inside the CPU
  - 1999 : SSE functions and the birth of a long serie (SSE4 & AVX)
- When chips stand out from CPU
  - 1998 : DSP style TMS320C67x as tools
  - 2008: Cell inside the PS3, IBM inside Road Runner and Top1 of Top500
- Business of compilers & very accurate programming model

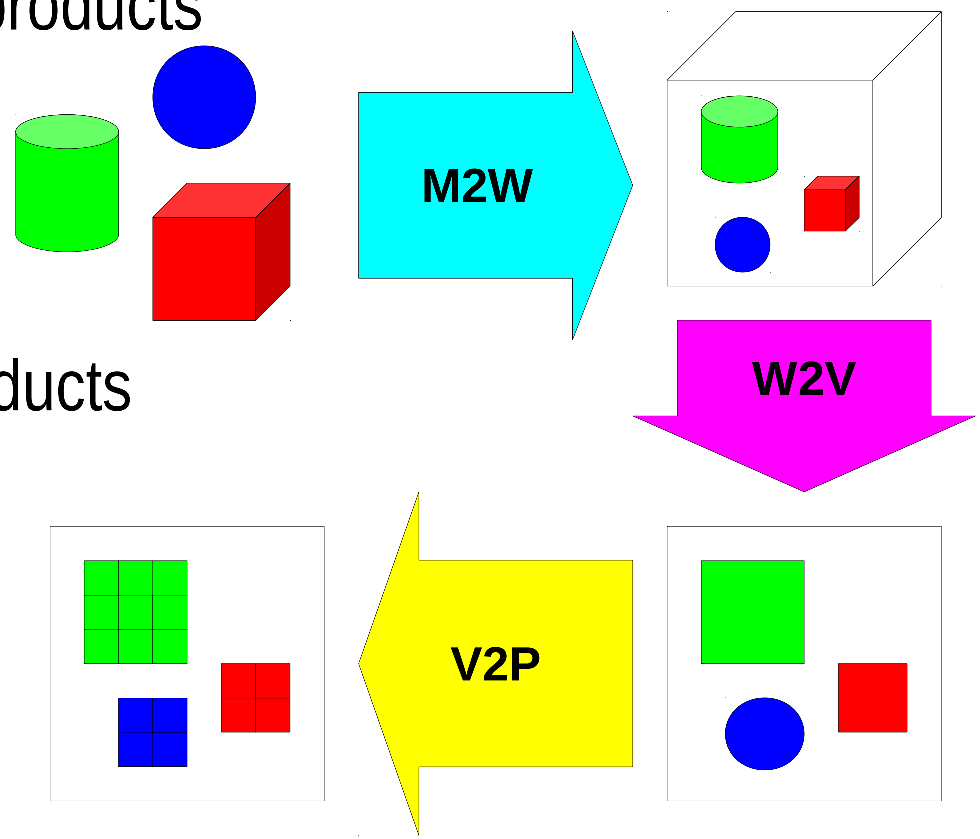
Why a GPU is so powerful ? To construct 3D scene !



- 2 approaches:
  - Raytracing : PovRay
  - Shadering : 3 operations
- Raytracing :
  - Starting from the eye to the objects of scene
- Shadering
  - Model2World : vectorial objects placed in the scene
  - World2View : projection of objects on a plan of view
  - View2Projection : pixelisation of vectorial plan of view

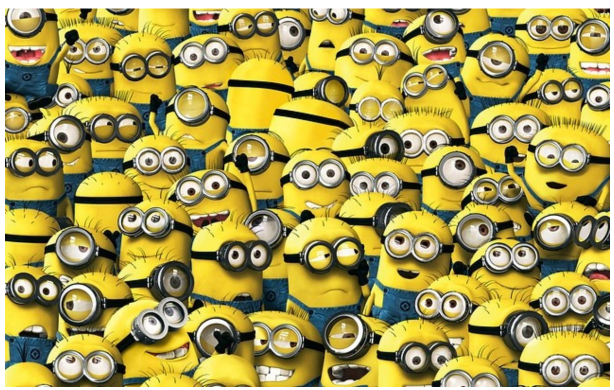
Why a GPU is so powerful Shadering & Matrix computing

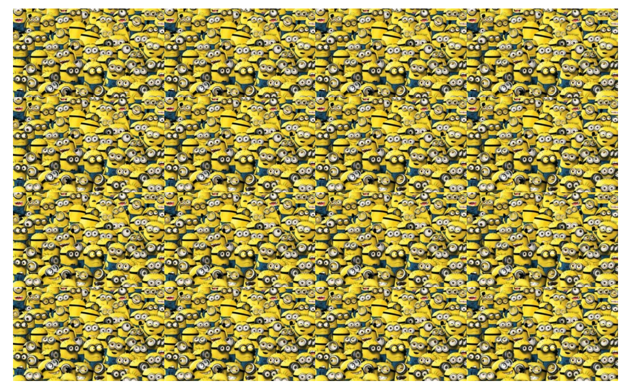
- **Modele 2 World** : 3 matrix products
  - Rotation
  - Translation
  - Scaling
- **World 2 View** : 2 matrix products
  - Camera position
  - Direction de l'endroit pointé
- **View 2 Projection**
  - Pixellisation
- A GPU : a « huge » Matrix Multiplier



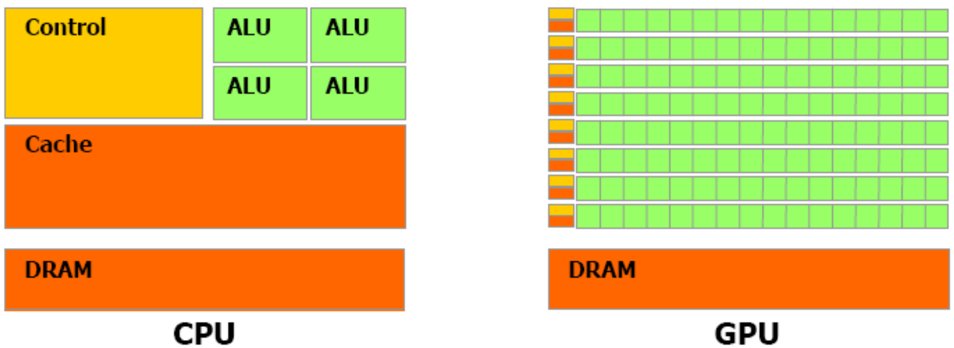
How a GPU must be considered
A laboratory of parallelism

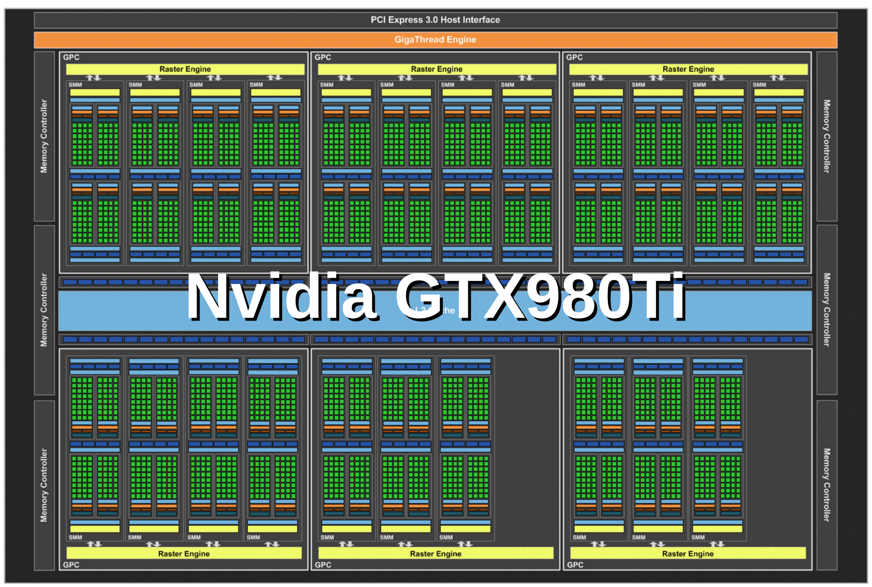
- You’ve got dozen of thousands of Equivalent PU !
- You have the choice between : **CPU MIC GPU**





How many elements inside ? Difference between GPU & CPU





- Operations
  - Matrix Multiply
  - Vectorization
  - Pipelining
  - Shader (multi)processor
- Programmation : 1993
  - OpenGL, Glide, Direct3D,
- Généricité : 2002
  - CgToolkit, CUDA, OpenCL

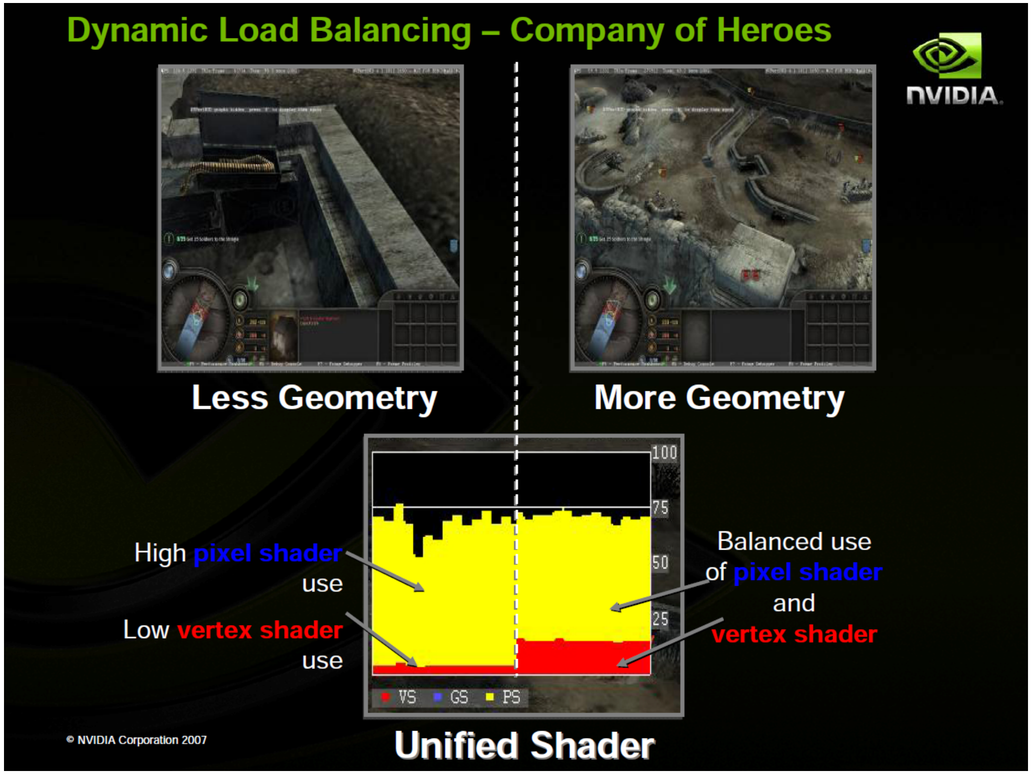
Why GPU is so powerful ? Come back generic processing

Inside the GPU

- Specialized processing units
- Pipelining efficiency

Adaptation Loss

- Changing scenes
- Different nature of details



The Solution

More generalist processing units !

Why is GPU so powerful All « M » Flynn taxonomy included

- Vectorial : SIMD (Simple Instruction Multiple Data)
  - Addition of 2 positions (x,y,z) : 1 uniq command
- Parallel : MIMD (Multiple Instructions Multiple Data)
  - Several executions in parallel with the (almost) same data
- In fact, SIMT : Simple Instruction Multiple Threads
  - All processing units share the Threads
  - Each processing unit can work independently from others
- Need to the Threads

Important dates

- 1992-01 : OpenGL and the birth of a standard
- 1998-03 : OpenGL 1.2 and interesting functions
- 2002-12 : Cg Toolkit (Nvidia) and the extensions of language
  - Wrappers for all languages (Python)
- 2007-06 : CUDA (Nvidia) or the arrival of a real langague
- 2008-06 : Snow Leopard (Apple) integrates OpenCL
  - La volonté d'utiliser au mieux sa machine ?
- 2008-11 : OpenCL 1.0 and its first specifications
- 2011-04 : WebCL and its first version by Nokia

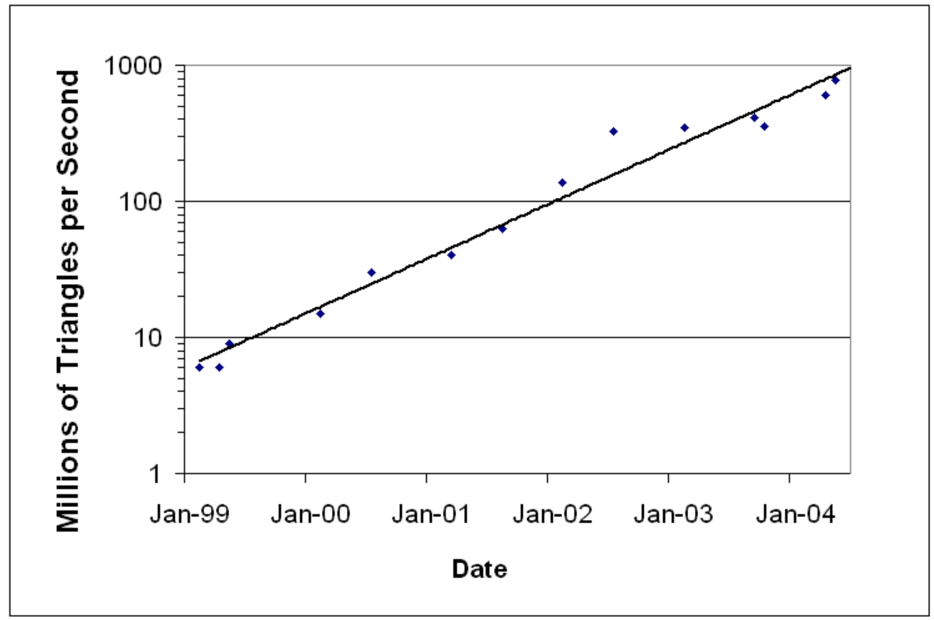
Who use OpenCL As applications

- OS : Apple in MacOSX
- « Big » applications :
  - Libreoffice
  - Ffmpeg
- Known applications
  - Graphical ones : photoscan,
  - Scientific : OpenMM, Gromacs, Lammps, Amber, …
- Hundreds of softwares, libraries ! But checking is essentiel...
  - https://www.khronos.org/opencl/resources/opencl-applications-using-opencl

Why GPU is a disruptive technology

How to reshuffle the cards in Scientific Computing

- In a conference in february 2006, this…
  - x100 in 5 years



- Between 2000 and 2015
  - GeForce 2 Go/GTX 980Ti : from 286 to 2816000 MOperations/s : x10000
  - Classical CPU: x100

Just a remind of yesterday

Parallel programming models

	Cluster	**Node CPU**	Node GPU	Node Nvidia	Accelerator
MPI	Yes	Yes	No	No	Yes*
PVM	Yes	Yes	No	No	Yes*
OpenMP	No	Yes	No	No	Yes*
Pthreads	No	Yes	No	No	Yes*
OpenCL	No	Yes	Yes	Yes	Yes
CUDA	No	No	No	Yes	No
TBB	No	Yes	No	No	Yes*

- OpenCL is the most versatile one
- CUDA can ONLY be used with Nvidia GPUs
- Accelerator seems to be the most universal, but...

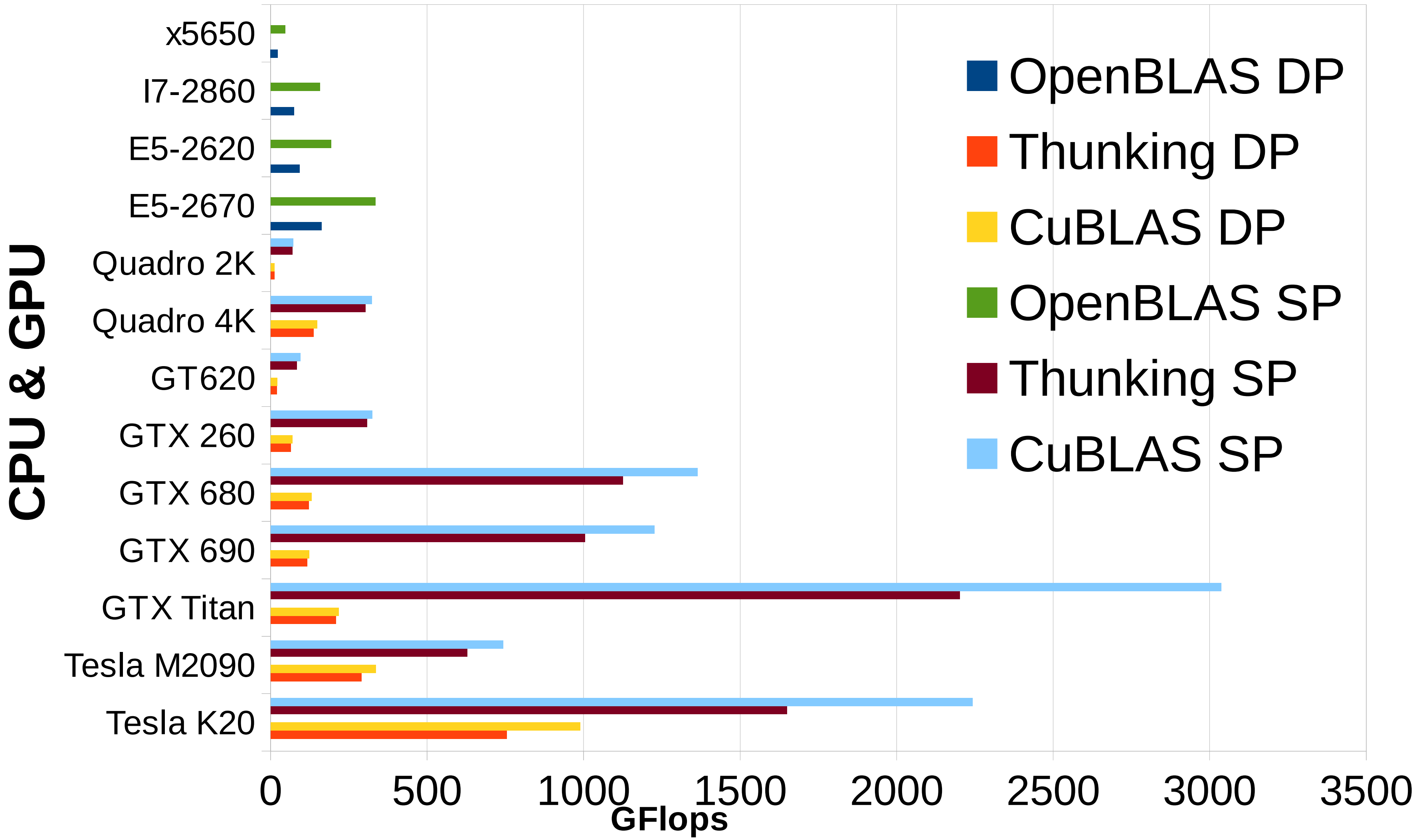
Act as integrator : do not reinvent the wheel !

Parallel programming librairies

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

- Classic librairies can be used under GPU
  - Nvidia provides lots of implementations
  - OpenCL provides several on them, but not so complete

BLAS match : CPU vs GPU xGEMM when x=D(P) or S(P)



Why OpenCL
Non Politically Correct History

- In the middle of 2000, Apple needs power for MacOSX
  - Some processing can be done by Nvidia chipsets
  - CUDA exists as a good successor of CgToolkit
- But Apple did not want to be jailed (as their users :-))
- Apple promotes the Khronos consortium
  - AMD, IBM, ARM,
  - … and Nvidia & Intel

What OpenCL offers
10 implementations on x86

- 3 implementations for CPU :
  - AMD one : the original
  - Intel one : efficient for specific parallel rate
  - PortableCL (POCL) : OpenSource
- 3 implementations for GPU :
  - AMD one : for AMD/ATI graphical boards
  - Nvidia one : for Nvidia graphical boards
  - « Beignet » one : Open Source one for Intel Graphical Chipset
- 1 implementation for Accelerator :
  - Intel one : for Xeon Phi
- On other platforms, it seems to be possible

Goal : replace loop by distribute 2 types of distributions

- Blocks/WorkItems
  - Domain « global », large but slow amount of memory available
- Threads
  - Domain « local », small but quick amount of memory available
  - Needed for synchronisation of process
- Different access to memory :
- « clinfo » to get the properties of CL devices : Max work items
  - Max work items : 1024x1024x1024 for CPU so 1 billion distribution
  -

How ?
A « Hello World » in OpenCL...

- Define two vectors in ASCII
- Duplicate them in 2 huge vectors
- Add them using a OpenCL kernel
- Définir deux vecteurs en « Ascii »
- Print them on screen

Add of

2 vectors a+b =c

For each 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!		

Differences between CUDA/OpenCL

```
__device__ ulong MainLoop(ulong iterations,uint seed_w,uint seed_z,size_t work)
{
  uint jcong=seed_z+work;
  ulong total=0;
  for (ulong i=0;i<iterations;i++) {
     float x=CONGfp ;
     float y=CONGfp ;
    ulong inside=((x*x+y*y) <= THEONE) ? 1:0;
    total+=inside;
  }
__global__ void MainLoopBlocks(ulong *s,ulong iterations,uint seed_w,uint seed_z)
{
  ulong total=MainLoop(iterations,seed_z,seed_w,blockIdx.x);
  s[blockIdx.x]=total;
  __syncthreads();
}
```

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

Kernel Code & Build and Call Add a vector in 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)
```

How to OpenCL ? Write « Hello World ! » in C

```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;
}
```

OpenCL kernel

Kernel Call

Number of lines
Of OpenCL kernel

How to OpenCL ? Write « Hello World ! » in Python

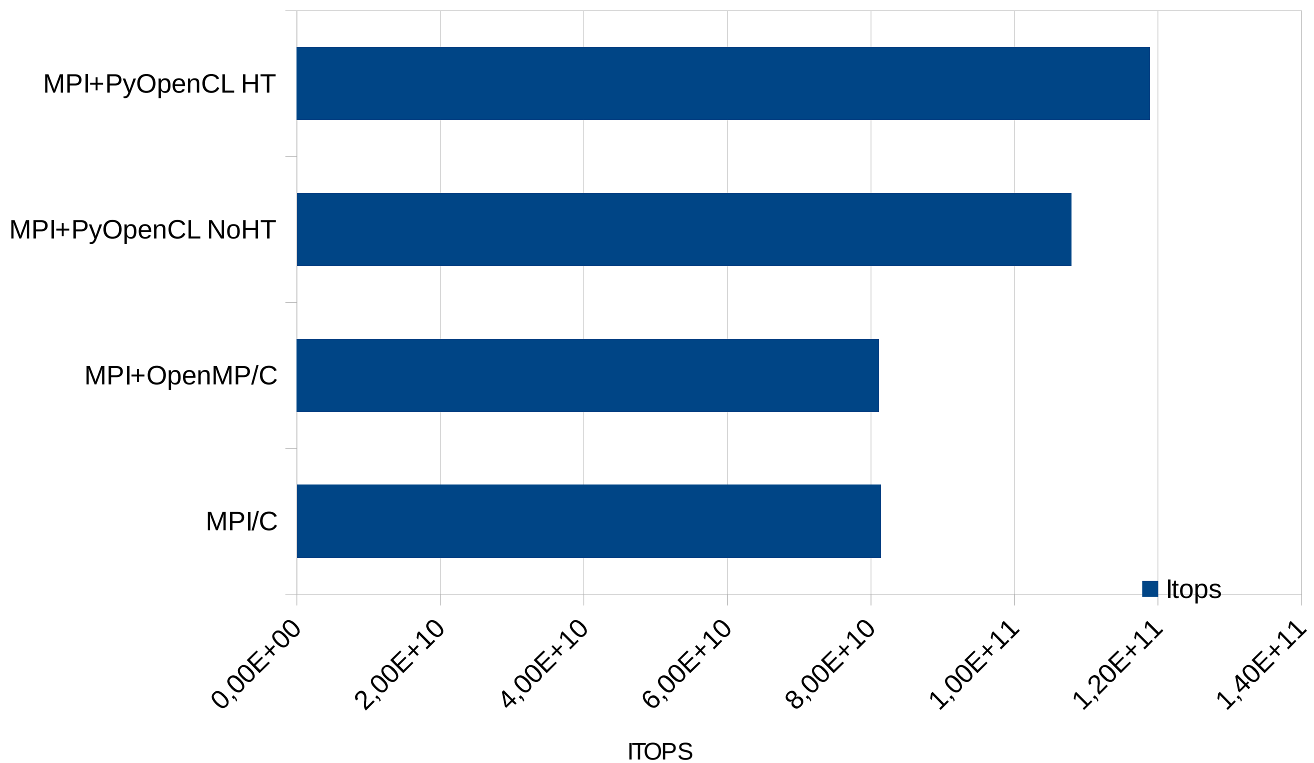
OpenCL kernel

```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)
```

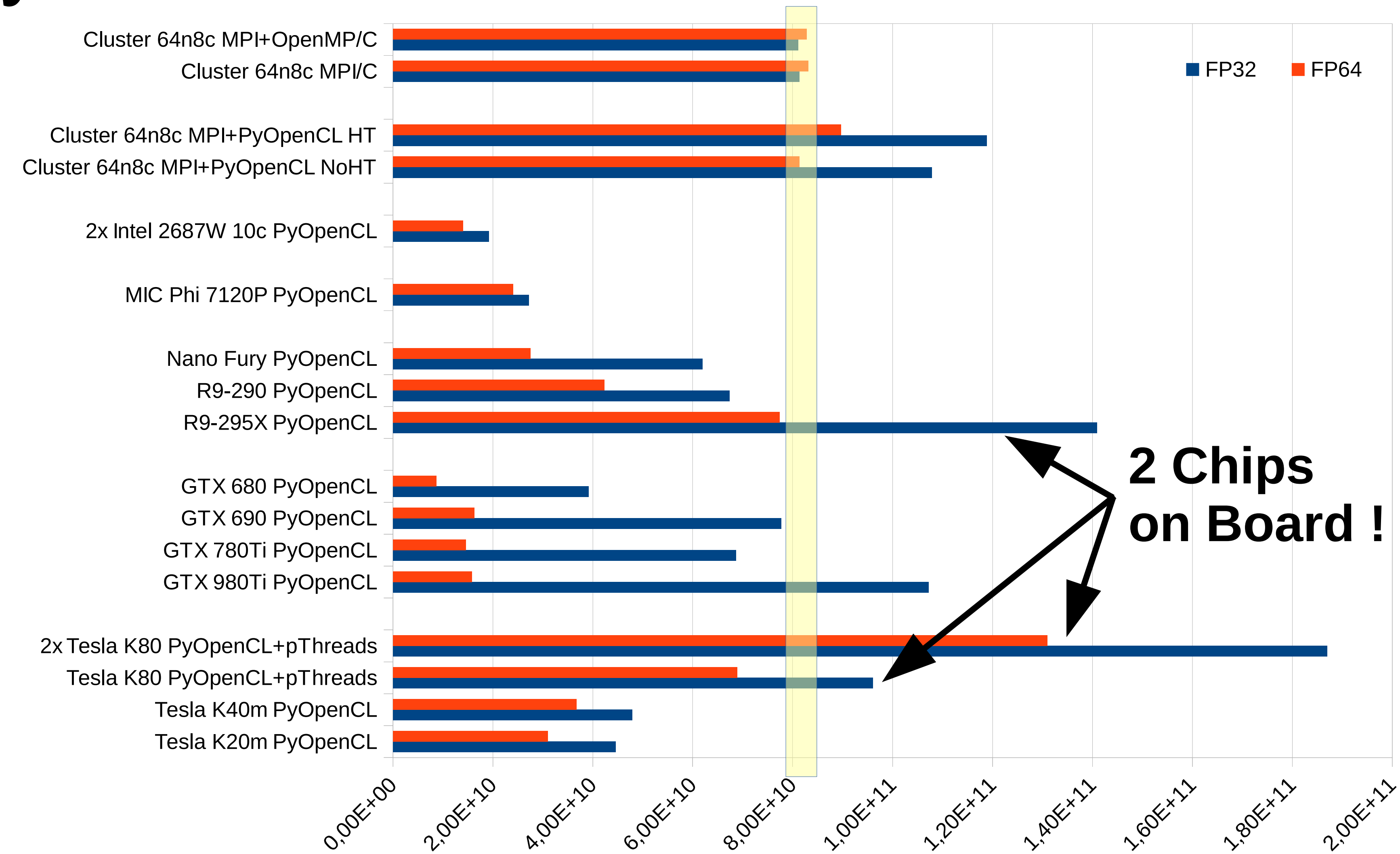
```python
mf = cl.mem_flags
GPUVector1 = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=HostVector1)
GPUVector2 = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=HostVector2)
GPUOutputVector = cl.Buffer(ctx, mf.WRITE_ONLY, HostVector1.nbytes)
OpenCLProgram = cl.Program(ctx, OpenCLSource).build()
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");
```

Kernel Call

You think Python is not efficient ? Let's have a quick comparison...



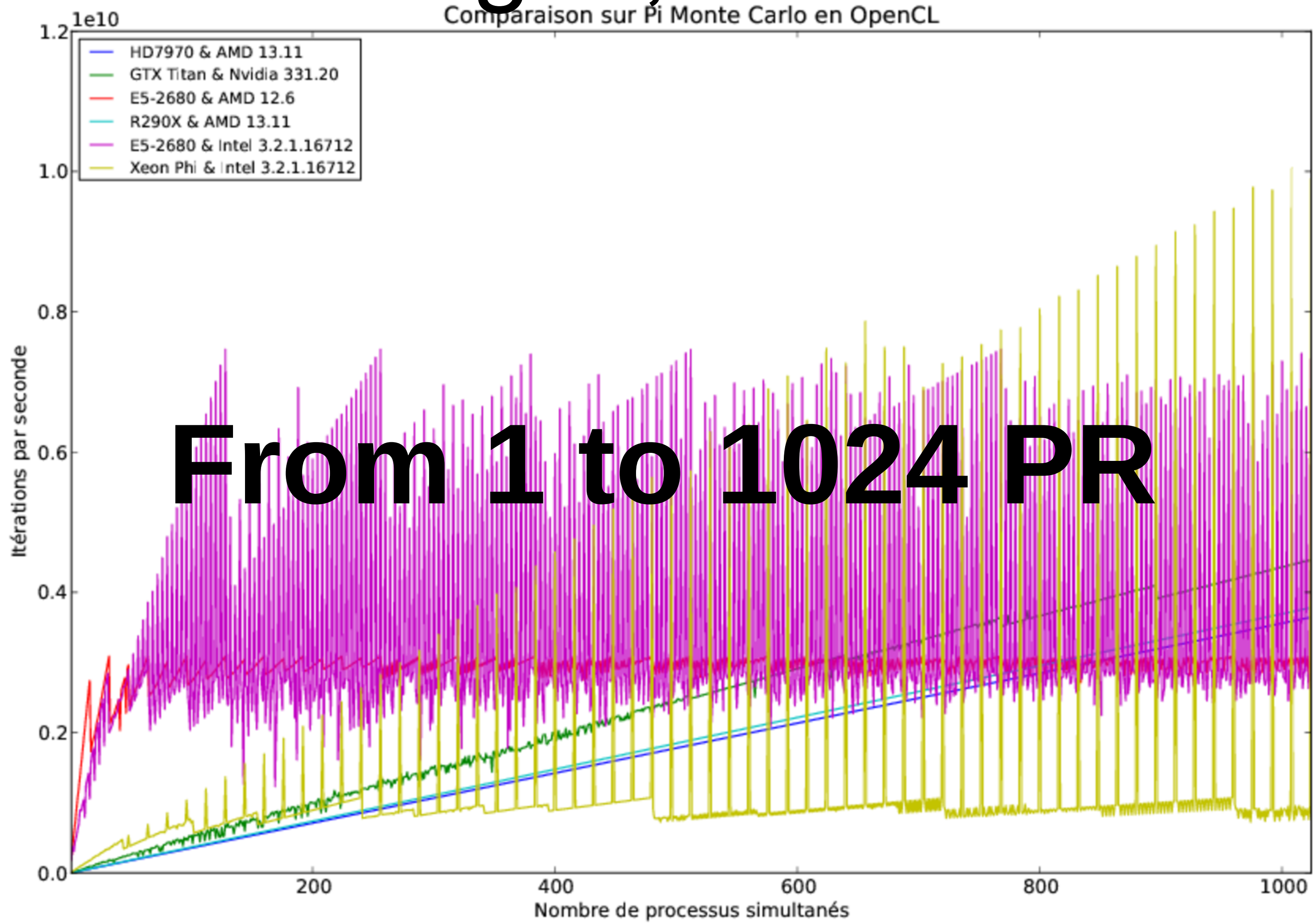
Pi Dart Board match (yesterday) : Python vs C & CPU vs GPU vs Phi



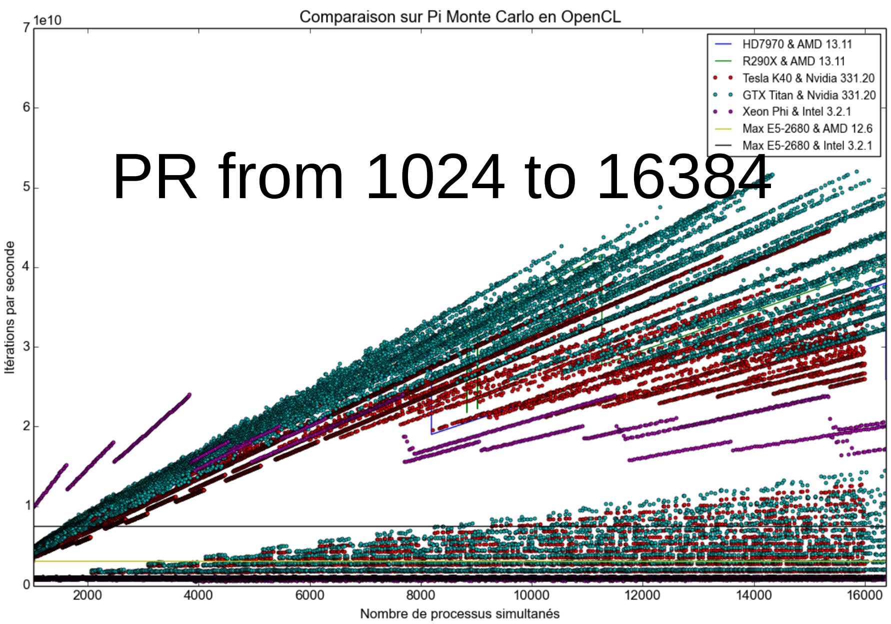
How to OpenCL ?
« Hello World » C/Python : to weight

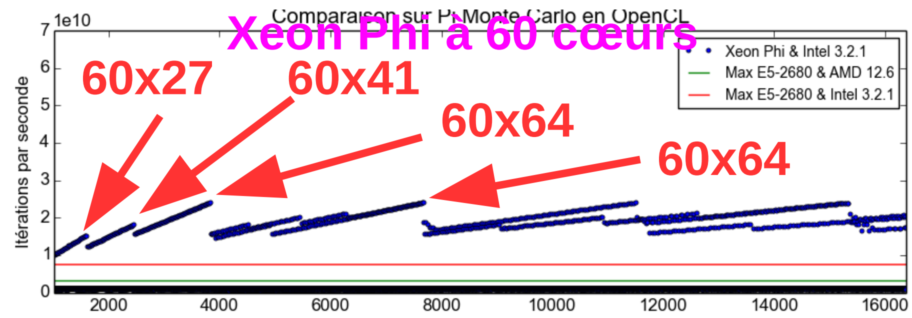
- On previous OpenCL implementations :
  - In C : 75 lines, 262 words, 2848 bytes
  - In Python : 51 lines, 137 words, 1551 bytes
  - Factors : 0.68, 0.52, 0.54 in lines, words and bytes.
- Programming OpenCL :
  - Difficult programming context
    - « To open the box in more difficult than to assemble the furniture unit ! »
    - Requirement to simplify the calls by an API
  - No compatibility between SDK of AMD and Nvidia
    - Everyone rewrite their own API
- One solution, however « The » solution : Python

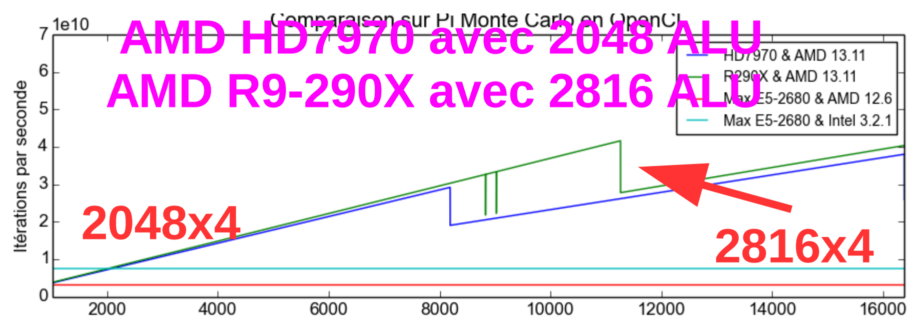
Small comparison between *PU
MIC Phi emerges, GPU ambushed

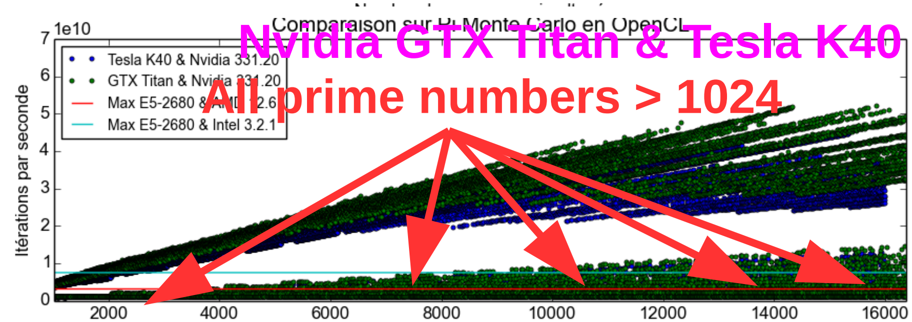


Deep exploration of Parallel Rates
Parallel Rate from 1024 to 16384

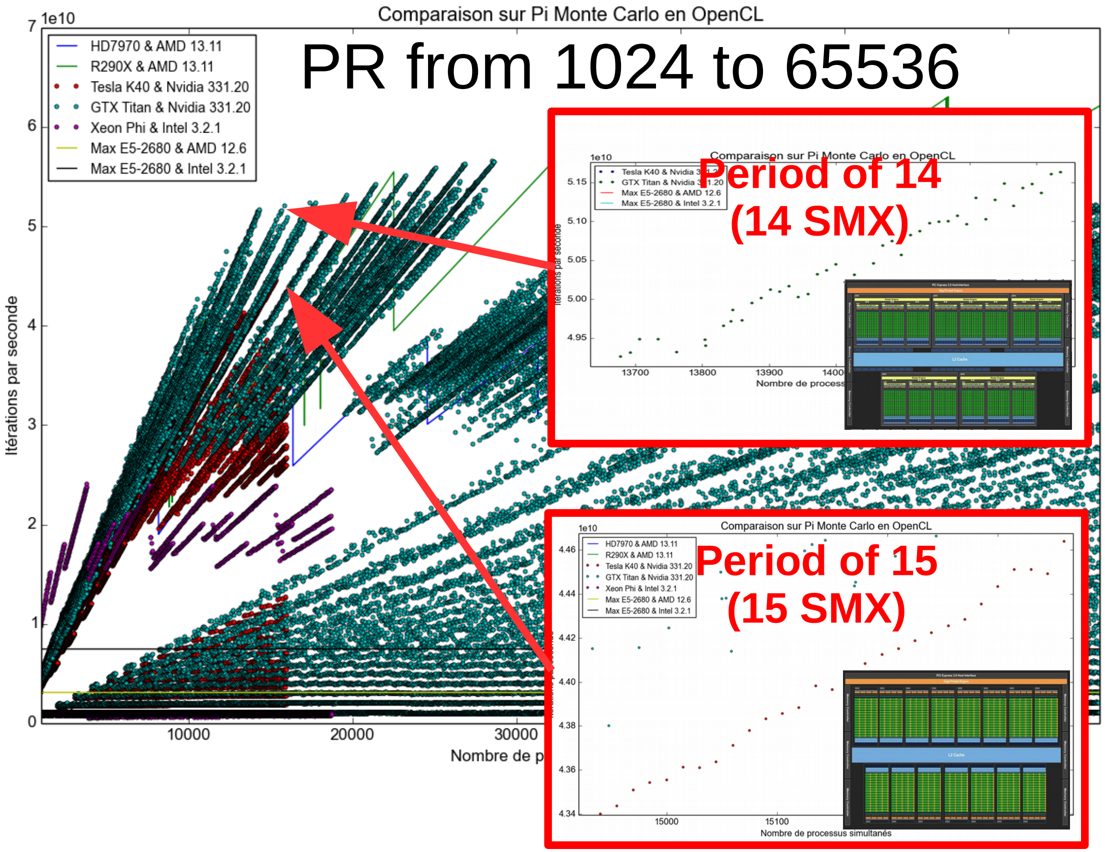




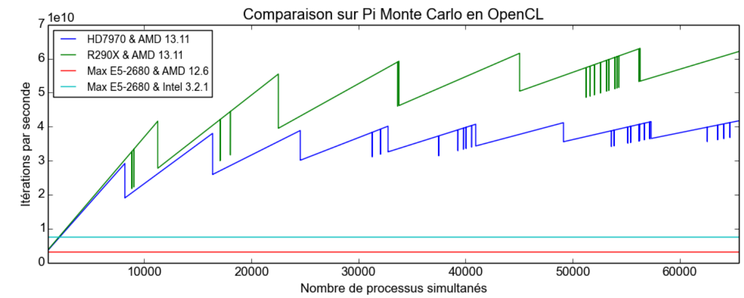




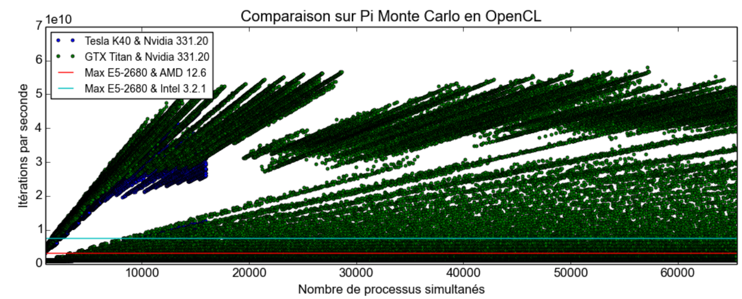
Deep exploration of Parallel Rates
Parallel Rate from 1024 to 65536



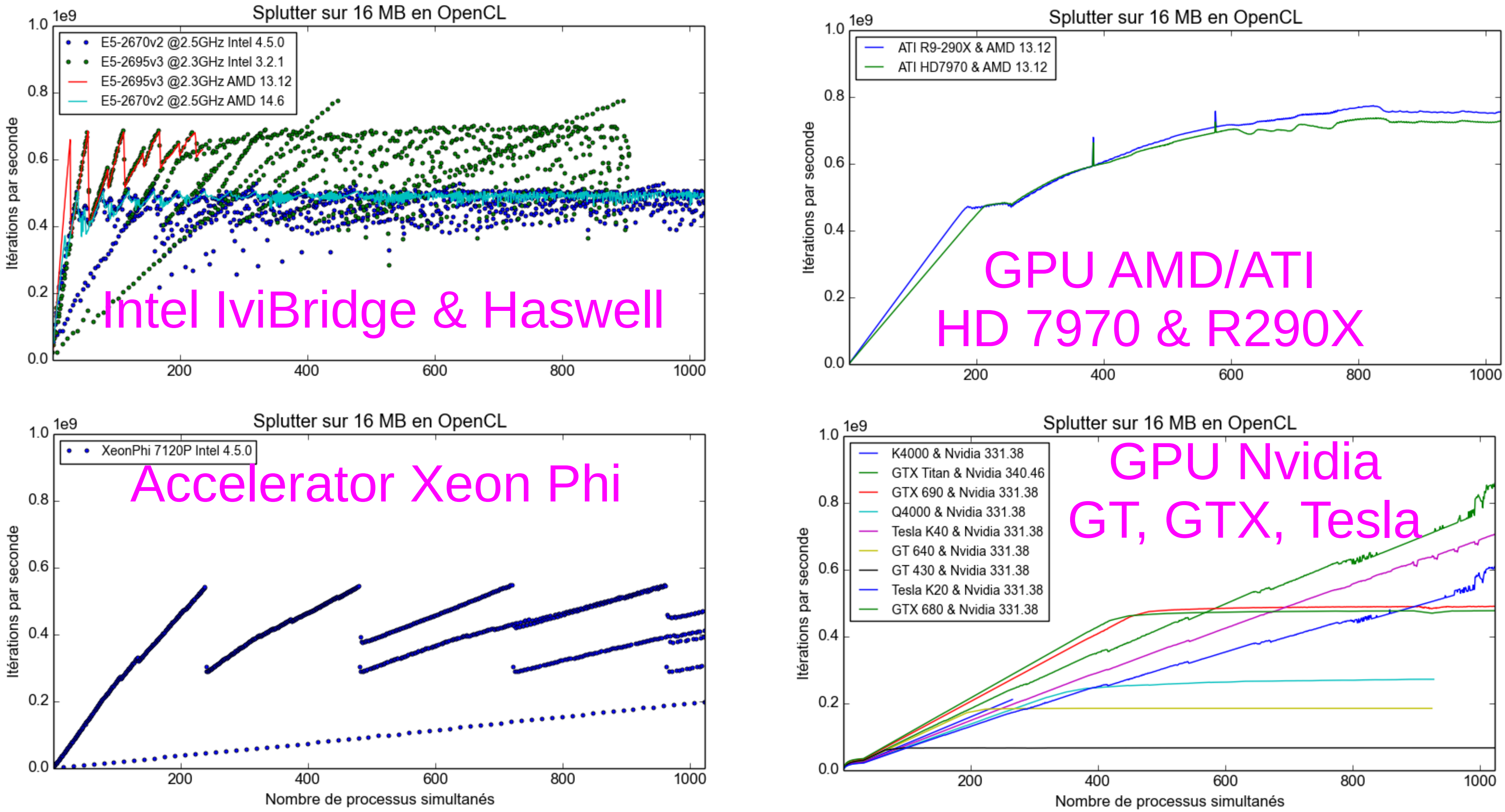
AMD HD7970 & R9-290
Long Period : 4x number of ALU



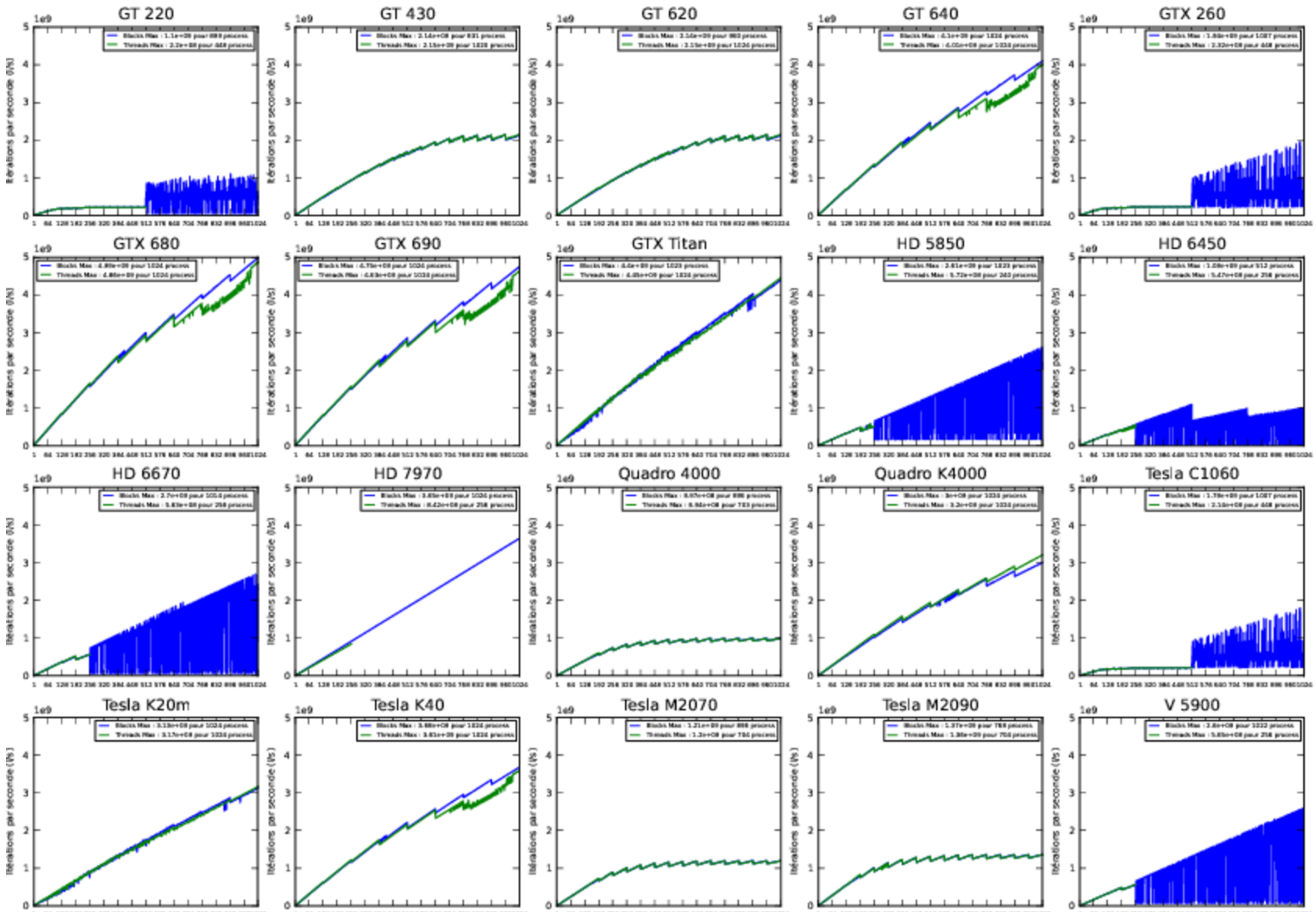
Nvidia GTX Titan & Tesla K40
Short Period : number of SMX units



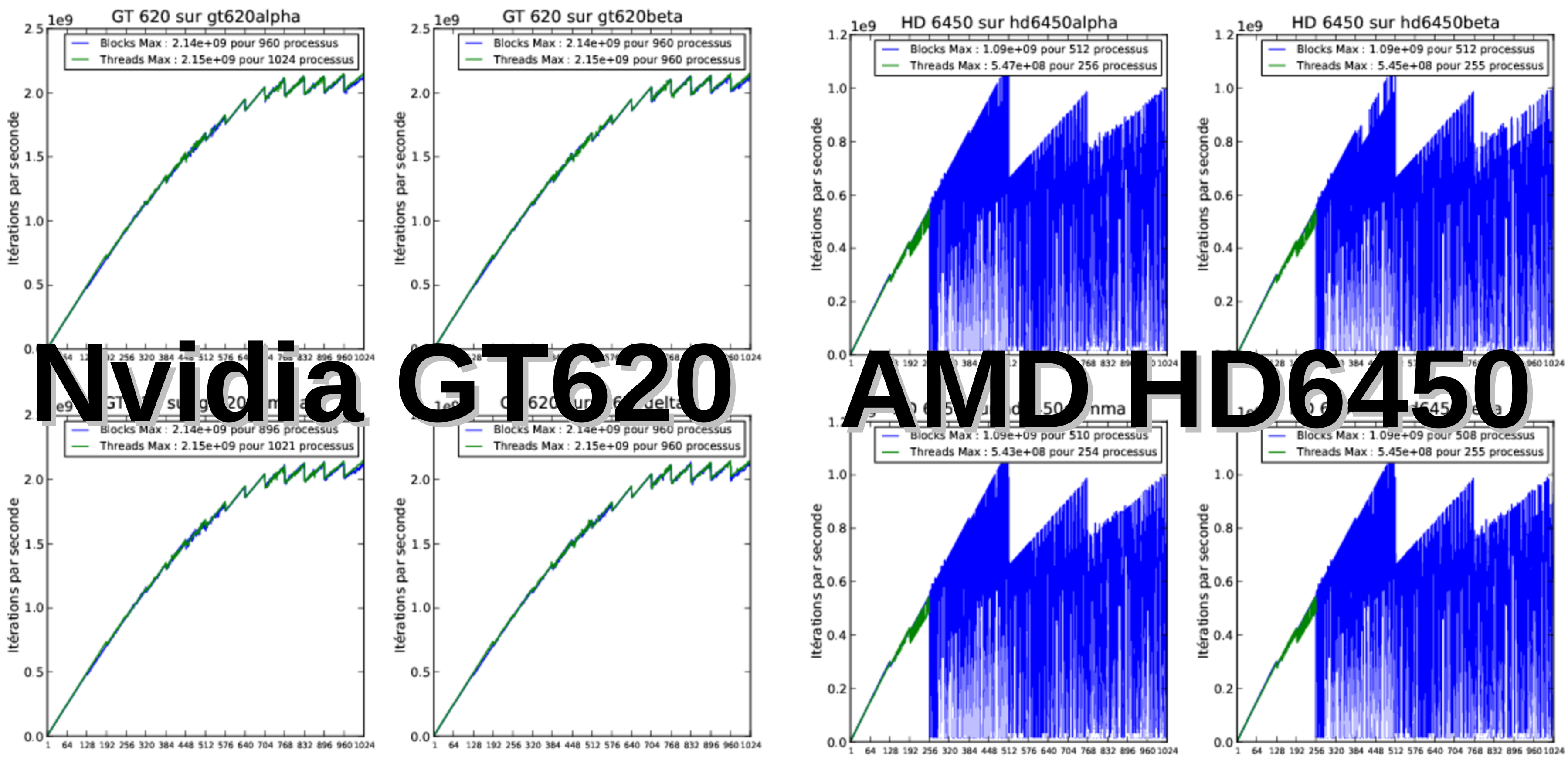
Splutter code on OpenCL devices



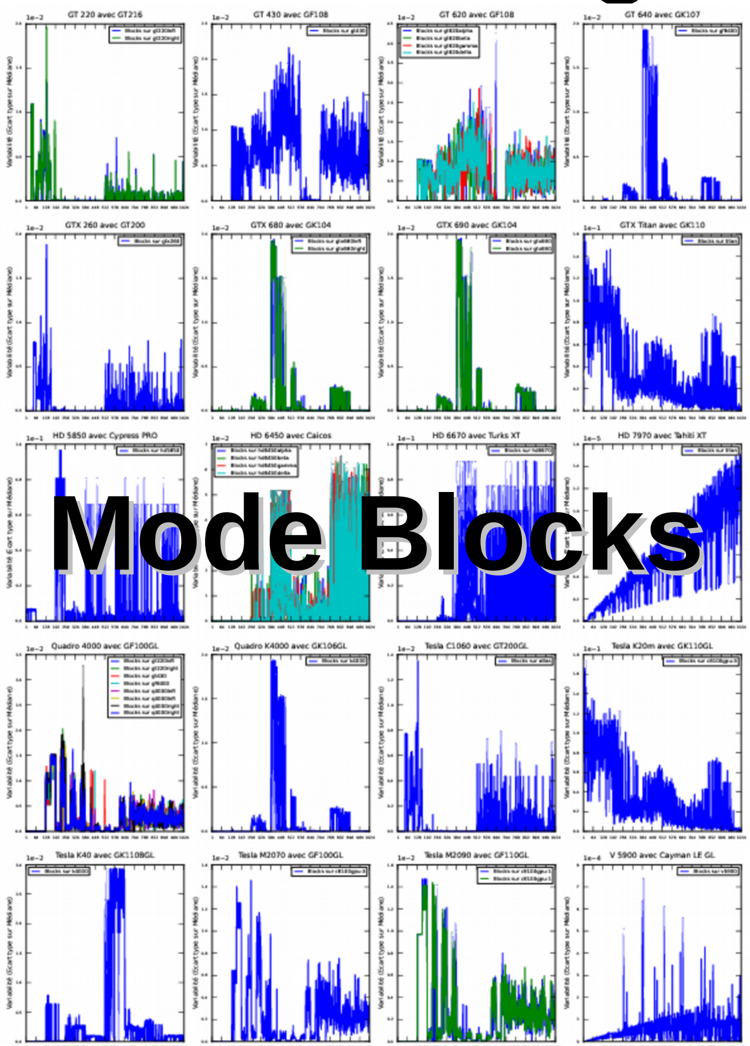
Behaviour of 20 (GP)GPU vs PR

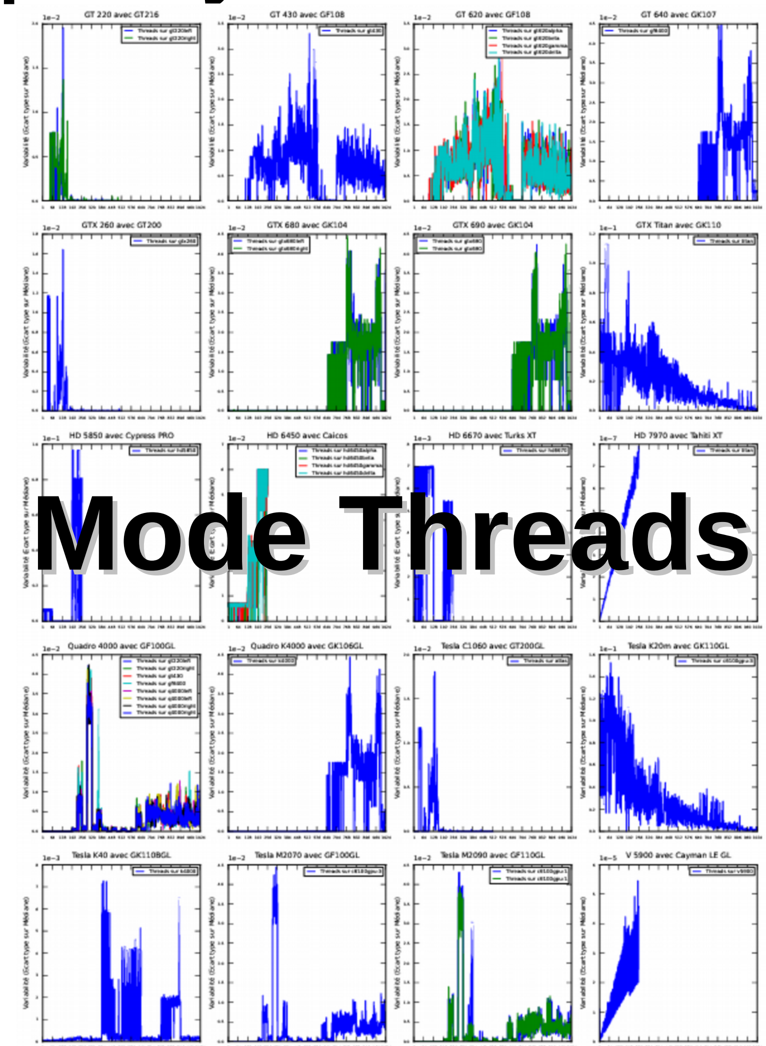


Is Parallel Envelop reproducible ? For 2 specific boards...



Variability : A distinctive criterion ! What a strange property :-/ ...





Inside your VirtualBox OpenCL CPU implementations

- What's inside :
  - The 3 CPU implementations for OpenCL : AMD, Intel and Open Source
- How to get OpenCL devices ?
  - clinfo
- How to launch a small example in Python
  - python /usr/share/doc/python-pyopencl-doc/examples/benckmark.py
- How to launch my test codes :
  - http://houches2016.no-ip.org/redmine/projects/parallelism/documents

What does it look like ?
Lets have a look to the codes !

- MPI implementation in C
- MPI+OpenMP implementation in C
- OpenCL implementation in Python
- Threads+OpenCL implementation in Python
- MPI+OpenCL implementation in Python

Conclusion

- Now, the « brute » power is inside the GPUs
- But, QPU (PR=1) of GPU is 50x slower than CPU
- To exploit GPU, parallel rate MUST be over 1000
- To program GPU, prefer :
  - Developer approach : Python with PyOpenCL (or PyCUDA)
  - Integrator approach : external libraries optimized
- In all cases, instrument your launches !