PyPhy, 2011-08-29

Multi-(nodes|cores|shaders) On Python For Dummies

A (very) pragmatic point of view

Emmanuel QUEMENER
Centre Blaise Pascal, ENS-Lyon

From science to simulations : causality

- Inside Science principles: causality
  - *“Relation between 2 events: cause & consequence”*
    - Definition from Wikipedia
    - Strangely: no page in french (but Descartes is french) !
- Corollary : reproducibility
  - *“Same causes, same consequences”*
- Paradigm of computing simulations (only ?):
  - `Process(Input) → Output` like `Cause → Consequence`
  - Is observational (slave?) or experimental (master?) science ?

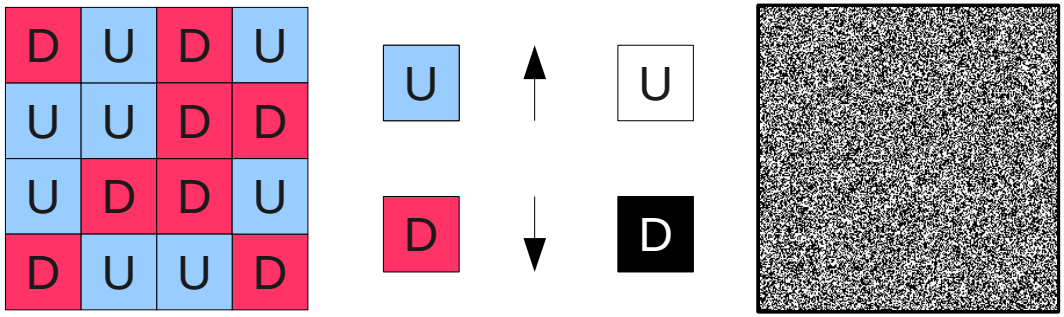
From science to simulations : causality

- What to do : Redo & Explore phase space
  - For done `Process`, if `Input==Input'` then `Output=Output'`
    - in fact '=' seems '~' (with mean & standard deviation defined)
  - For `Input[n]` , get `Output[n]` for `n` → {the largest}
- Example : Ising 2D model
- Python and parallelism (to explore tiny phase space)
  - Multi-cores : distributing “locally”
  - Multi-nodes : distributing “network-ly”
  - Multi-shaders : distributing “onboard-ly”

Ising 2D as “physical problem” Spin glass

- A random distributed square spin glass

4x4



256x256

- A simple interaction based on 4 neighbors
  - `LocalEnergy=2*J*s(x,y)*[s(x-1,y)+s(x-1,y)+s(x,y-1)+s(x,y+1)]`
- A simple rule to flip (1 → -1 or -1 → 1) site of lattice:
  - 1) If `LocalEnergy` negative : converted by neighbors
  - 2) If RNG lower than Boltzmann factor (@ `LocalEnergy` and T°)
  - (1) is *“Panurge cheep law”*, (2) is *“french spirit law”*

Ising 2D model : now, simulation
7 questions for the project

- **Why** ? Estimate phase transition T°
- **What** ? Lattice and Energy evolutions for T° ~ [0-5]
- **Where** : Lattice of size 256^2 with 256^3 random access
- **Who** ? A Random Number Generator
  - 2 RND to get a (x,y) coordinate of a spin site
  - 1 RND to be compared with Bolzmann factor
- **When** ? Retro-planning for PyPhy conference
- **How much** ? Nothing (in first approximation !)
- **How** ? Python/Numpy implementation

Ising 2D model : serial simulation

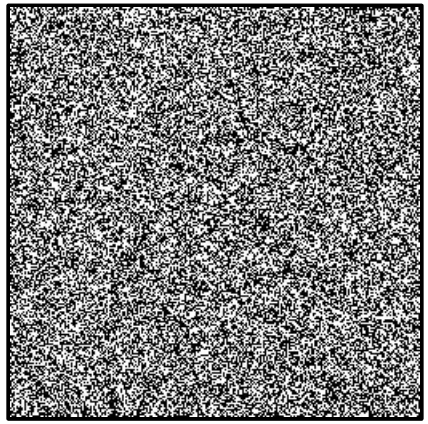
- Function Metropolis : **Metropolis(sigma,J,T,step,iterations)**
  - sigma : lattice of spin
  - J : coupling factor (equals 1 in all following simulations)
  - T : temperature to explore
  - Step : iterations between 2 laps (to save an output image)
  - Iterations : total explorations of lattice (randomly explored)
- Exploring temperatures : from 0.1 to 5 step of 0.1
- Outputs :
  - Energy : based on interaction coupling estimation
  - OutputImage : lattice distribution of spins

Ising 2D model : serial simulation Main Loop

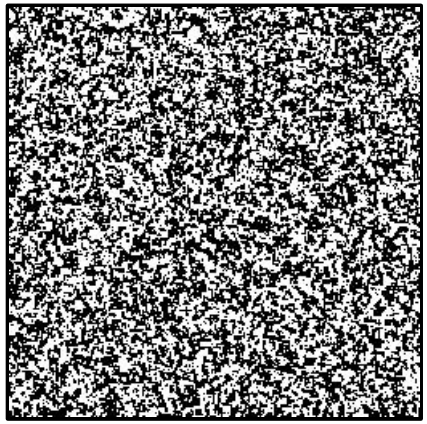
```
def MainLoop(sigma,J,T,iterations):
    for p in range(0,iterations):
        X=numpy.random.randint(SIZE),
        Y=numpy.random.randint(SIZE)
        DeltaE=2.*J*sigma[X,Y]*(
                        sigma[X,(Y+1)%SIZE]+
                        sigma[X,(Y-1)%SIZE]+
                        sigma[(X-1)%SIZE,Y]+
                        sigma[(X+1)%SIZE,Y])
        if DeltaE < 0. or random() < exp(-DeltaE/T):
            sigma[X,Y]=-sigma[X,Y]
```

- RNG retreives
  - X,Y coordonates
  - Between 0 and 1
- Boundary conditions
  - “cylindrical universe”
- Spin Interaction
  - Energy estimation
- Boltzmann factor
  - Random get & check

Ising 2D model : serial simulation
Iterations as parameter of Evolution

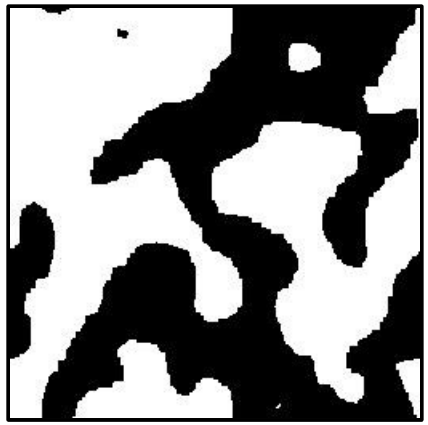


Initial : T=0.1

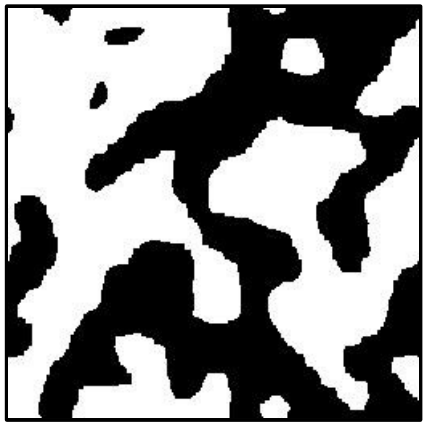


i=N²

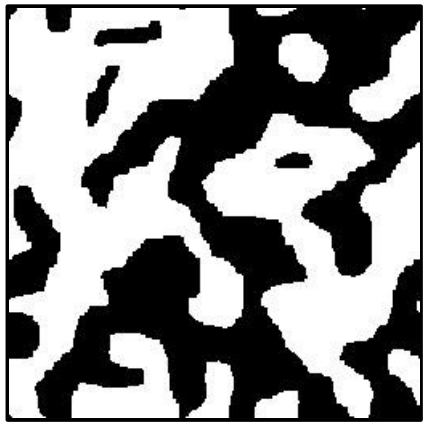
i=N³/4



Final : i=N³

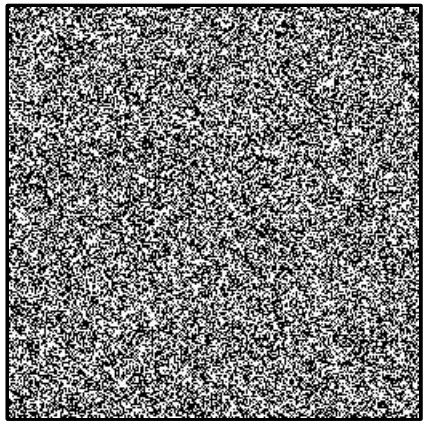


i=3N³/4



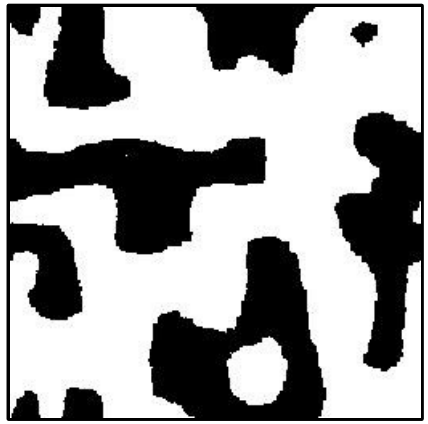
i=N³/2

Ising 2D model : serial simulation
T° as parameter of Evolution

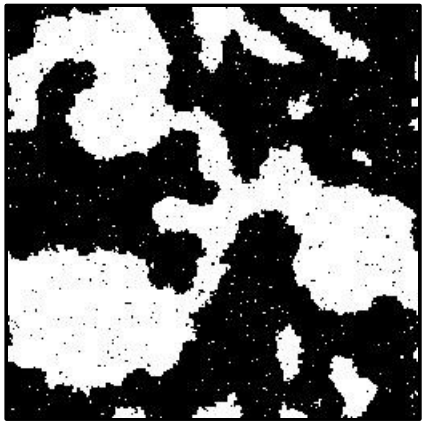


Initial

T=0.1

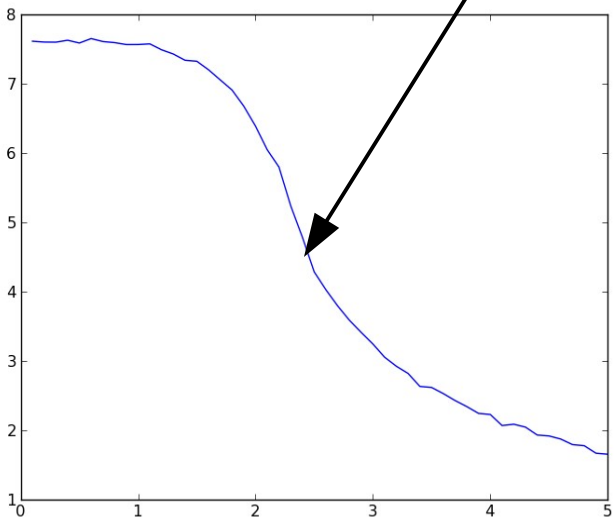


T=0.8

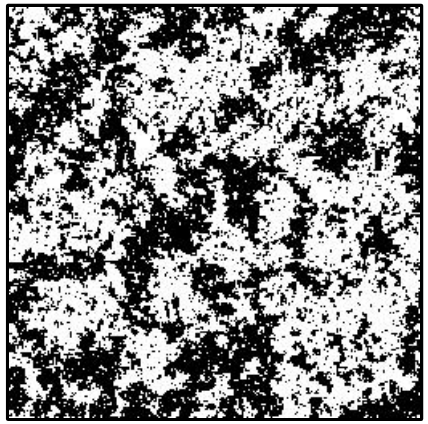


T=1.6

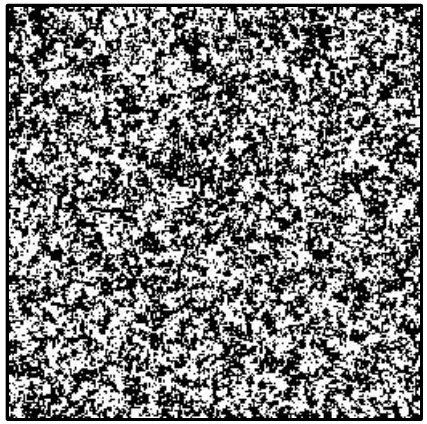
Tc~2.3



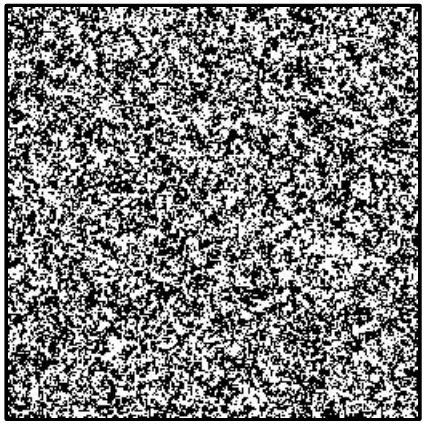
Energy=f(T)



T=2.4



T=3.2



T=4.0

Ising 2D : from serial simulation … to parallel simulation

- A goal : estimate phase transition T° !
- From T° ~ 0.1K to T° ~ 5K with step of 0.1K :
  - 50 Times the same operations !
- Idea : distribute pools of T° on processors
- How to do it ?
  - First : “thread” module
  - Second : “multiprocessing” module
  - Third : “MPI4Py” module

Ising2D : “Thread” module

- Define process : Metropolis function call
- Define job : each temperature between 0.1 and 5
- Launch 1 thread of process per job (each T°)
- Wait (a long long time !)

Ising 2D : “Thread” module

```
import threading
from threading import Thread
…
sigma={}
Trange=numpy.arange(0.1,5.1,0.1)
for T in Trange:
    sigma[t]=numpy.copy(sigmaIn)
    thread=threading.Thread(target=Metropolis,
                            args=(sigma[t],J,T,step,iterations))
    thread.start()
    threadList.append(thread)
for t in threadList:
   t.join()
```

Ising2D : “Thread” module

- Behavior :
  - Expensive cost for context switching
- Bad ! Most efficient in serial process.
  - Reason : GIL : Global Interrupt Link
  - But can be used in outside processes
- Solution : “Multiprocessing” module

Ising 2D : “Multiprocessing” module Simple distribution locally

- Import multiprocessing into Python
  - (in *“batteries included”*)
- Define process : Metropolis
- Define pool & jobs
- Redefine main core simulation function
  - With only 1 parameter: T°
- Launch pool
- Wait (no so much time !)

Ising2D : “Multiprocessing” module
Main procedure

```
from multiprocessing import Pool
PROCS=12
Trange=numpy.arange(Tmin,Tmax+Tstep,Tstep)
def MetropolisStrip(T):
        sigma=numpy.copy(sigmaIn)
        Metropolis(sigma,J,T,step,iterations)
        return(T,Energy(sigma,J))
pool=Pool(processes=PROCS)
Results=pool.map(MetropolisStrip,Trange)
```

Ising 2D : “Multiprocessing” module
Simple distribution locally

- Behavior :
  - Efficient distribution of jobs on cores
  - Distributing process low cost
- Advantages : very simple implementation
- Inconvenients :
  - limited to the number of SMP cores
  - 2 to 4 : cheaper
  - >4 to 48 : can become very expensive

Ising 2D : “mpi4py” module MPI implementation

- Why “mpi4py” instead on “mpi”
  - Try to use python-mpi : 1/1example crash
  - Mpi4py : new implementation
    - Not in Squeeze
    - But easy backport
- Two types of jobs to define : master & slave
  - On master : define pool & jobs, send data
  - On slave : receive data, compute, send result

Ising2D :"mpi4py" module
Main procedure

- MPI initialization
- Definition of Master work : `rank==0` (in MPI, chief is 0 !)
  - Define Lattice with random conditions
  - Define jobs to achieve based on T°
  - Split global work to nodes
  - Send each node Lattice and work cases
- Definition of Slave work : `rank!=0`
  - Receive Lattice and test cases
  - Compute all test cases
  - Send Master results

Ising2D :"mpi4py" module
Main for Master (rank is 0)

```
sigmaIn=numpy.where(numpy.random.randn(SIZE,SIZE)>0,1,-1).astype(numpy.int8)
T=numpy.arange(Tmin,Tmax+Tstep,Tstep)
if len(T)%NODES==0:
   SizePerNode=len(T)/NODES
else:
   SizePerNode=len(T)/NODES+1
for i in range(NODES):
   if i==NODES-1:
      Input=T[(i+1)*SizePerNode+1:]
   else:
      Input=T[i*SizePerNode:(i+1)*SizePerNode]
ToSend=sigmaIn,J,Input
comm.send(ToSend, dest=i+1, tag=11)
for i in range(NODES):
   Output=comm.recv(source=i+1,tag=11)
   ToSave+=Output
```

Initial Conditions

Distribute Work

Send Work to Slaves

Receive from Slaves

Ising2D :"mpi4py" module
Main for Slave (rank different of 0)

```
ToSplit=comm.recv(source=0, tag=11)
sigmaIn,J,Input=ToSplit
Output=[]
for T in Input:
    sigma=numpy.copy(sigmaIn)
    duration=Metropolis(sigma,J,T,step,iterations)
    E=Energy(sigma,J)
    Output.append([T,E,M])
comm.send(Output, dest=0, tag=11)
```

Retrieve from Master

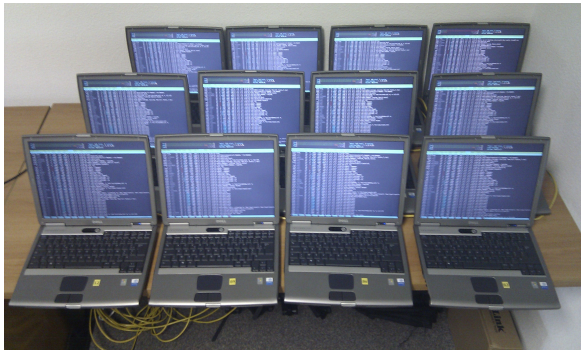
Split data

Compute

Each test case

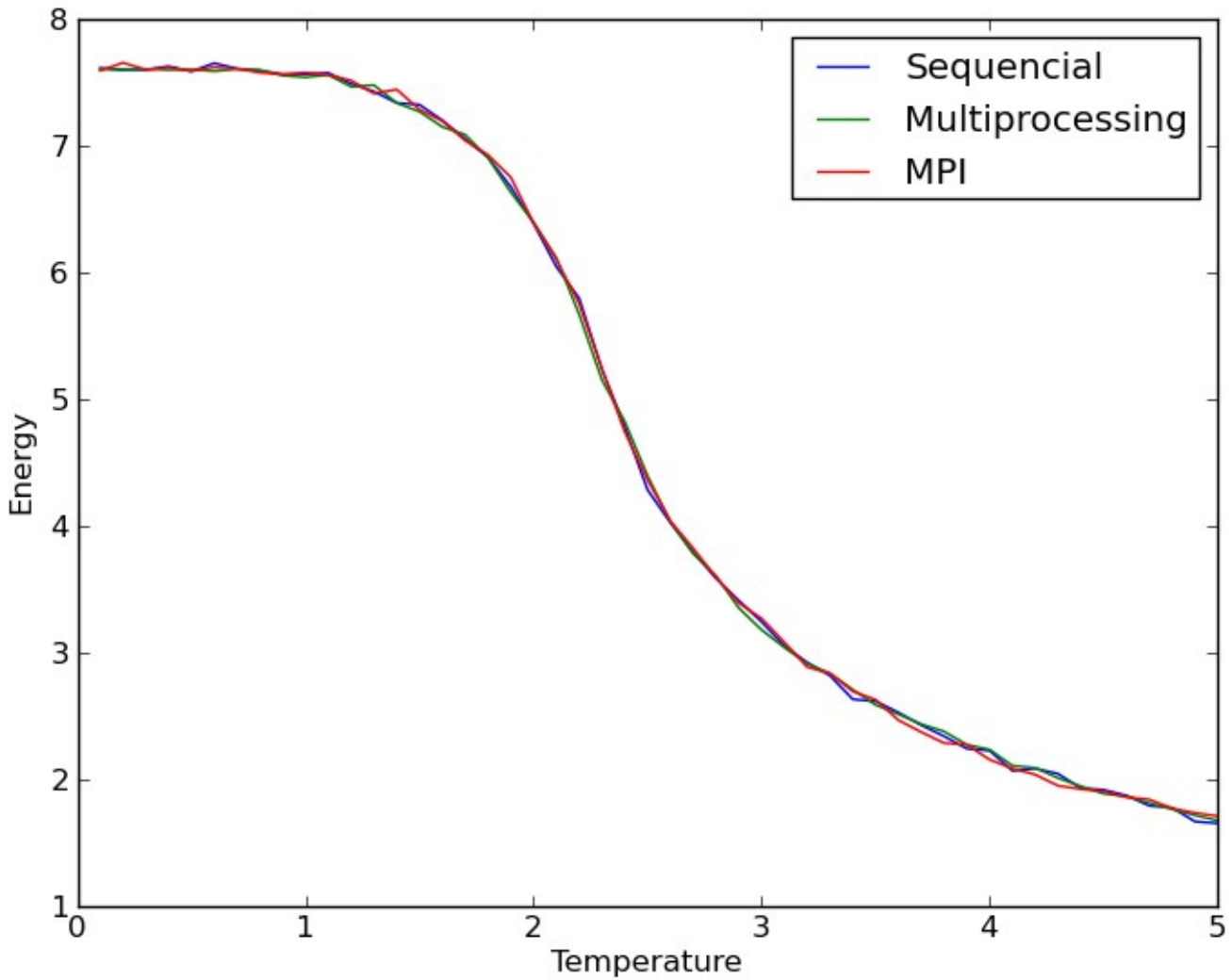
Send results to Master

Ising 2D : MPI “mpi4py” module Main program



- Behavior :
  - Efficient distribution of jobs on nodes
- Advantages :
  - Large distribution over network (on several nodes)
  - Local distribution possible (on several cores)
- Inconvenients :
  - “tiny” MPI learn
  - Need program on nodes (by shared folder for example)

Ising 2D on Python
Are my results coherent ?



Pretty Match...

Isn't it ?

Normal...
Same program

Ising 2D on Python
First conclusions on CPU

- **Sequential process** :
  - each T° ~150s, Real 126m, User 126m
- **Thread** : Gain : **~0.29** (best of 12 or 1 because of GIL)
  - each T° ~25000s, Real 7h11m, User 7h11m
- **MultiProcessing** : Gain : **~7.6** (best of 12)
  - each T° ~163s, Real 15m37, User 132m
- **MPI** : Gain : **~9** (best of 10 : 1 master and 1 nothing)
  - each T° ~164s, Real 13m54, User 149m
- And the winner is... **MP(I)** (no, that's programmer ;-))

Python & GPU : from OpenGL to Cuda & OpenCL

- Why using GPU shader for computing ? Because...
  - It's “fashion” !
  - It's cheap : good performance/price **41 times cheaper** !
    - 200€ : 520 Gflops (yes, in SP, but 100 in DP !) : GTX 560Ti
    - 4k€ : 240 Gflops (in SP, but 120 in DP) : 2x X5650 with 6 cores
- A brief history (of mistime ?)
  - @ the beginning : OpenGL and vector functions
  - In december 2002, Nvidia-CG
  - In june 2007, CUDA
  - In august 2009, OpenCL (In MacOSX Leopard) !
  - In 2011 : 3 implementations (Nvidia, **AMD** and Intel)

Python & GPU : Prerequisites

- A **recent GPU** supporting OpenCL (not necessary) and/or CUDA :
  - Nvidia video board
  - AMD/ATI video board (but not necessary)
- A **specific** video **driver** (not necessary)
  - On Nvidia, last one is 270.41.19 for CUDA 4
  - On AMD/ATI, last one is 11.8
- A package with **librairies** (and examples !) :
  - On Nvidia, cudatoolkit & gpucomputingsdk 4.0.17 of last may 2011
  - On AMD, AMD-APP-SDK-2.5 of last august 2011 (can be used on CPU)
- **Python API** (thousands thanks to Andreas Klöckner)
  - PyOpenCL, last one is 2011.1.2 (for ATI & Nvidia boards)
  - PyCUDA, last one is 2011.1.2 (for Nvidia specific)

“OpenCL” : a brief introduction

- OpenCL : Open Computation Language
  - Pushed by Apple (why ?)
  - Sponsored by AMD, Nvidia, IBM, etc...
  - Extension of OpenGL for classical computing operations
- good :
  - To be used by both GPU and CPU
  - 3 implementations : Nvidia, AMD (and Intel)
- bad :
  - Learning not a peace of cake

OpenCL : warnings & why Python

- OpenCL is difficult to learn (and difficult to code)
  - And Kernel is not exactly the same on Nvidia and AMD
- API to code OpenCL in C from Nvidia and AMD/ATI
  - And not compatible one to the other !
- But Why using Python ?
- A small example : add 2 vectors (& print HW), `wc test`
  - In C : 75 lines, 262 words, 2848 bytes
  - In Python : 51 lines, 137 words, 1551 bytes
  - Factors : **0.68** on lines, **0.52** on words and **0.54** on bytes
- Not yet convinced why to use Python : **LOOK** !

OpenCL : simple program "Hello OpenCL World!"

- Define two Ascii vectors
- Duplicate them as 2 large vectors
- Add them by OpenCL kernel
- Print the result

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

“OpenCL” : a simple program in 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” : a simple program in 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");
```

Ising 2D : “OpenCL” module

- Warnings :
  - no RNG implemented : MWC sufficient
  - Data locality : “global” & “local” approaches
- Possible implementations
  - “global” : GPU as ALU → brutal
  - “local” : GPU as multi-ALU → distribution
- **ENORMOUS Warnings !**
  - The Cast of variables !
  - The memory spaces (specifically as local ones)

Ising2D : PyOpenCL & global memory

- Only One Space & Only One program
- Same as : Python/Numpy
  - But need to program in C-like !
- Speeding process on tests
- Let's have a quick look ?
  - On OpenCL code
  - On Python API calls

Ising2D : “OpenCL” module
Global OpenCL Kernel

```
#define znew (z=36969*(z&65535)+(z>>16))
#define wnew (w=18000*(w&65535)+(w>>16))
#define MWC ((znew<<16)+wnew )
#define MWCfp (MWC + 1.0f) * 2.32830643545449 4e-10f
__kernel void MainLoop(__global char *s,
    float J,float T,uint size,uint iterations,
    uint seed_w,uint seed_z)
{
   uint sizex=size;
   uint sizey=size;
   uint z=seed_z;
   uint w=seed_w;
   float DeltaE=0.0f;
   for (uint i=0;i<iterations;i++) {
      uint x=MWCfp*sizex ;
      uint y=MWCfp*sizey ;
      int p=s[x+sizex*y];
      int d=s[x+sizex*((y+1)%sizey)];
      int u=s[x+sizex*((y-1)%sizey)];
      int l=s[((x-1)%sizex)+sizex*y];
      int r=s[((x+1)%sizex)+sizex*y];
      DeltaE=2.0f*J*p*(u+d+l+r);
      int factor=((DeltaE < 0.0f) ||
                  (MWCfp <  exp(-DeltaE/T))) ? -1:1;
      s[x%sizex+sizex*(y%sizey)] = factor*p;
      barrier(CLK_GLOBAL_MEM_FENCE);
   }
}
```

Ising 2D : “PyOpenCL” module Global Main Loop

```
    for platform in cl.get_platforms():
        for device in platform.get_devices():
            if cl.device_type.to_string(device.type)
==='GPU':
                GPU=device
    ctx = cl.Context([GPU])
    queue =
cl.CommandQueue(ctx,properties=cl.command_q
ueue_properties.PROFILING_ENABLE)
    mf = cl.mem_flags
    sigmaCL = cl.Buffer(ctx, mf.WRITE_ONLY |
mf.COPY_HOST_PTR,hostbuf=sigma)
    MetropolisCL =
cl.Program(ctx,KERNEL_CODE).build(options =
"-cl-mad-enable -cl-fast-relaxed-math")
```

```
    i=0
    step=lap
    duration=0.
    while (step*i < iterations):
    CLLaunch=MetropolisCL.MainLoop(queue,(1,
),(1,), sigmaCL,numpy.float32(J),numpy.float32(T
), numpy.uint32(SIZE), numpy.uint32(step),
numpy.uint32(nprnd(2**8)),
numpy.uint32(nprnd(2**8)))
    CLLaunch.wait()
    elapsed = 1e-9*(CLLaunch.profile.end -
CLLaunch.profile.start)
    cl.enqueue_copy(queue, sigma, sigmaCL
).wait()
    i=i+1
    duration=duration+elapsed
    sigmaCL.release()
```

Ising2D : PyOpenCL & local memory

- Multi Space & multi thread
- Splitting lattice into sub-lattices (local domains)
- For each local domain, one thread
- Boundaries conditions on local domains
- RNG seeds on local domains
- Thread synchronization needed
- Let's have a quick look ?
  - On OpenCL code
  - On Python API calls

Ising 2D : “PyOpenCL” Local OpenCL Kernel

```
#define BSZ 16
#define znew (z=36969*(z&65535)+(z>>16))
#define wnew (w=18000*(w&65535)+(w>>16))
#define MWC ((znew<<16)+wnew )
#define MWCfp (MWC + 1.0f) * 2.328306435454494e-10f
__kernel void MainLoop(__global int *s,float J,float T,uint size, uint iterations,uint seed_w,uint seed_z)
  __local int Sg[BSZ*BSZ]
  int base_idx=BSZ*get_group_id(0);
  int base_idy=BSZ*get_group_id(1);
  int base_id=base_idx+base_idy*size;
  uint glob_id=base_id+get_local_id(0)+get_local_id(1)*size;
  Sg[get_local_id(1)*BSZ+get_local_id(0)]=s[glob_id];
  barrier(CLK_LOCAL_MEM_FENCE);
  __local uint z,w,x,y;
  z=seed_z;
  w=seed_w;
  __local float DeltaE;
  int i,p,u,d,l,r;
for (i=0;i<iterations;i++)  {
    x=MWCfp*BSZ ;
    y=MWCfp*BSZ ;
    p=Sg[x+BSZ*y];
    u=Sg[x+BSZ*((y-1)%BSZ)];
    d=Sg[x+BSZ*((y+1)%BSZ)];
    l=Sg[((x-1)%BSZ)+BSZ*y];
    r=Sg[((x+1)%BSZ)+BSZ*y];
    d= (y==BSZ-1) ? s[((base_idx+x)%size)+size*((base_idy+y+1)%size)]:d;
    u= (y==    0) ? s[((base_idx+x)%size)+size*((base_idy+y-1)%size)]:u;
    l= (x==    0) ? s[((base_idx+x-1)%size)+size*((base_idy+y)%size)]:l;
    r= (x==BSZ-1) ? s[((base_idx+x+1)%size)+size*((base_idy+y)%size)]:r;
    DeltaE=2.0f*J*p*(u+d+l+r);
    int factor= ((DeltaE < 0.0f) || (MWCfp < exp(-DeltaE/T))) ? -1:1;
    Sg[x+BSZ*y] = factor*p;
    s[base_id+x+size*y]= factor*p;
    barrier(CLK_LOCAL_MEM_FENCE);
    barrier(CLK_GLOBAL_MEM_FENCE);
  }
}
```

Ising 2D : “PyOpenCL” Local Main Loop

```
    for platform in cl.get_platforms():
        for device in platform.get_devices():
            if cl.device_type.to_string(device.type)
==='GPU':
                GPU=device
    ctx = cl.Context([GPU])
    queue =
cl.CommandQueue(ctx,properties=cl.comman
d_queue_properties.PROFILING_ENABLE)
   mf = cl.mem_flags
   sigmaCL = cl.Buffer(ctx, mf.WRITE_ONLY |
mf.COPY_HOST_PTR, hostbuf=sigma)
    MetropolisCL =
cl.Program(ctx,KERNEL_CODE).build(options
= "-cl-mad-enable -cl-fast-relaxed-math")
```

```
    step=lap/BSZ/BSZ
    i=0
    duration=0.
    while (step*i < iterations/BSZ/BSZ):
    CLLaunch=MetropolisCL.MainLoop(queue,
(SIZE,SIZE),(BSZ, BSZ), sigmaCL,numpy.float32(J),
numpy.float32(T),numpy.uint32(SIZE),numpy.uint32(step
),numpy.uint32(nprnd(2**8)),numpy.uint32(nprnd(2**8)))
    CLLaunch.wait()
    elapsed = 1e-9*(CLLaunch.profile.end -
CLLaunch.profile.start)
    cl.enqueue_copy(queue, sigma, sigmaCL).wait()
    i=i+1
    duration=duration+elapsed
    sigmaCL.release()
```

PyOpenCL : what's important !

- Implement your own RNG...
  - And be careful that RNG is pseudo one (seeds !)
- Cast the variables
  - At the creation for Numpy Arrays
  - At Kernel call for other ones
- Beware that memory is hierarchical
  - Global memory is low and rather large
  - Local memory is quick but very tiny !
- Thread call are implicit
  - And synchronization is necessary, time to time...

Ising2D : PyCUDA

- Almost same as PyOpenCL : simpler
  - GPU initialization process simplification
    - From X lines to Y lines
  - Find/Replace memory call statements
  - Use of AMD cookbook for migration CUDA → OpenCL

Ising 2D : “CUDA” module Global Kernel

```
#define znew (z=36969*(z&65535)+(z>>16))
#define wnew (w=18000*(w&65535)+(w>>16))
#define MWC ((znew<<16)+wnew )
#define MWCfp (MWC + 1.0f) * 2.32830643545449 4e-10f
__global__ void MainLoop(char *s,float J,float T,uint size, uint iterations,uint seed_w,uint seed_z)
{
   uint sizex=size;
   uint sizey=size;
   uint z=seed_z;
   uint w=seed_w;
   float DeltaE=0.0f;
   for (uint i=0;i<iterations;i++) {
      uint x=MWCfp*sizex ;
      uint y=MWCfp*sizey ;
      int p=s[x+sizex*y];
      int d=s[x+sizex*((y+1)%sizey)];
      int u=s[x+sizex*((y-1)%sizey)];
      int l=s[((x-1)%sizex)+sizex*y];
      int r=s[((x+1)%sizex)+sizex*y];
      DeltaE=2.0f*J*p*(u+d+l+r);
      int factor=((DeltaE < 0.0f) || (MWCfp < exp(-DeltaE/T))) ? -1:1;
      s[x%sizex+sizex*(y%sizey)] = factor*p;
      __syncthreads();
   }
}
```

Ising 2D : “CUDA” module
Global Main Loop

```
import pycuda.driver as cuda
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
from pycuda.compiler import SourceModule
sigmaCU = cuda.InOut(sigma)
mod = SourceModule(KERNEL_CODE)
MetropolisCU=mod.get_function(
    "MainLoop")
start = pycuda.driver.Event()
stop = pycuda.driver.Event()
i=0
duration=0.
step=lap
    while (step*i < iterations):
    cuda.Context.synchronize()
    start.record()
MetropolisCU(sigmaCU,J,T,numpy.uint32(
SIZE),numpy.uint32(step),
numpy.uint32(nprnd(2**8)
),numpy.uint32(nprnd(2**8)),block=(1,1,1))
    stop.record()
    stop.synchronize()
    elapsed = stop.time_since(start)*1e-3
    i=i+1
    duration=duration+elapsed
```

Ising 2D : “CUDA” module
Local Kernel

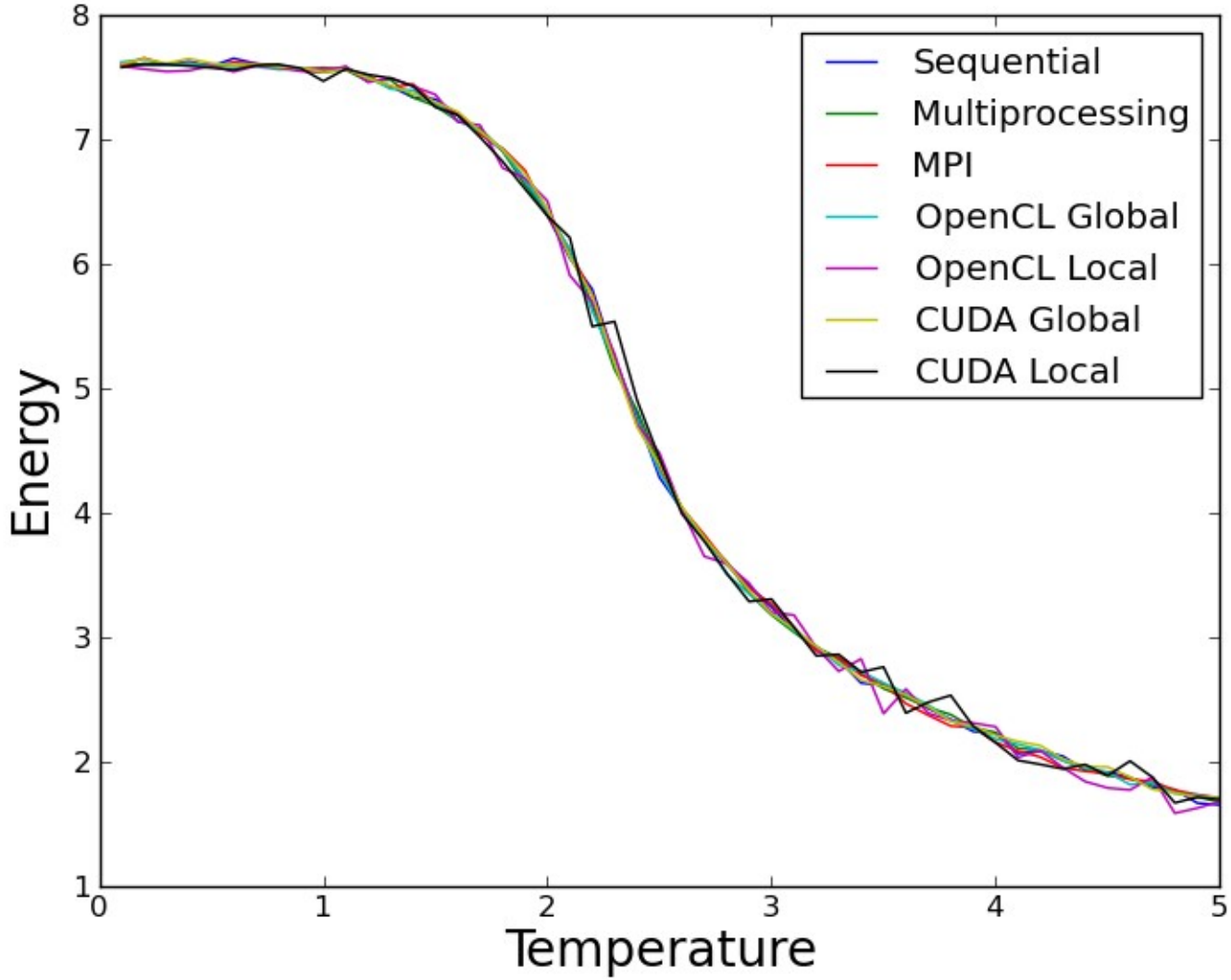
```
#define BSZ 16
#define znew (z=36969*(z&65535)+(z>>16))
#define wnew (w=18000*(w&65535)+(w>>16))
#define MWC ((znew<<16)+wnew )
#define MWCfp (MWC + 1.0f) * 2.328306435454494e-10f
__global__ void MainLoop(char *s,float J,float T,uint size,
                         uint iterations,uint seed_w,uint seed_z)
{
   __shared__ char Sg[BSZ*BSZ];
   uint base_idx=BSZ*blockIdx.x;
   uint base_idy=BSZ*blockIdx.y;
   uint base_id=base_idx+base_idy*size;
   uint glob_id=base_id+threadIdx.x+threadIdx.y*size;
   Sg[threadIdx.y*BSZ+threadIdx.x]=s[glob_id];
   __syncthreads();
   uint z=seed_z;
   uint w=seed_w;
   float DeltaE=0.0f;
   for (int i=0;i<iterations;i++)
   {
      uint x=MWCfp*BSZ ;
      uint y=MWCfp*BSZ ;
      int p=Sg[x+BSZ*y];
      int u=Sg[x+BSZ*((y-1)%BSZ)];
      int d=Sg[x+BSZ*((y+1)%BSZ)];
      int l=Sg[((x-1)%BSZ)+BSZ*y];
      int r=Sg[((x+1)%BSZ)+BSZ*y];
      d= (y==BSZ-1) ? s[((base_idx+x)%size)+size*((base_idy+y+1)%size)]:d;
      u= (y==    0) ? s[((base_idx+x)%size)+size*((base_idy+y-1)%size)]:u;
      l= (x==    0) ? s[((base_idx+x-1)%size)+size*((base_idy+y)%size)]:l;
      r= (x==BSZ-1) ? s[((base_idx+x+1)%size)+size*((base_idy+y)%size)]:r;
      DeltaE=2.0f*J*p*(u+d+l+r);
      int factor= ((DeltaE < 0.0f) || (MWCfp < exp(-DeltaE/T))) ? -1:1;
      Sg[x+BSZ*y] = factor*p;
      s[base_id+x+size*y]= factor*p;
      __syncthreads();
   }
}
```

Ising 2D : “CUDA” module
Local Main Loop

```
import pycuda.driver as cuda
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
from pycuda.compiler import SourceModule
sigmaCU = cuda.InOut(sigma)
mod = SourceModule(KERNEL_CODE)
MetropolisCU=mod.get_function("MainLoop")
start = pycuda.driver.Event()
stop = pycuda.driver.Event()
step=lap/BSZ/BSZ
i=0
duration=0.
```

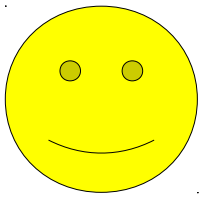
```
while (step*i < iterations/BSZ/BSZ):
    cuda.Context.synchronize()
    start.record()
    MetropolisCU(sigmaCU,J,T,
            numpy.uint32(SIZE),
            numpy.uint32(step),
            numpy.uint32(nprnd(2**8)),
            numpy.uint32(nprnd(2**8)),
            grid=(SIZE/BSZ,SIZE/BSZ),
            block=(BSZ,BSZ,1))
    stop.record()
    stop.synchronize()
    elapsed = stop.time_since(start)*1e-3
    i=i+1
    duration=duration+elapsed
```

Ising 2D : are Energy curves MP(I) & GPU coherent ?



Reproducibility with random processes...

Not so bad, Isn't it ?



Ising 2D : Results with GPU

- **Sequential process** : Each T° ~150s, Real 126m, User 126m

  Machine used : Precision 7500 & Nvidia Quadro 4000

- **OpenCL Global** : Gain : **~5**
  - each T° ~27s, Real 23m57, User 13m54, Sys 10m
- **OpenCL Local** : Gain : **~55 or ~71**
  - each T° ~2.11s, Real 2m17, User 54s, Sys 1m22
- **CUDA Global** : Gain : **~4 to ~5**
  - each T° ~30s, Real 33m31, User 33s23, Sys 3s
- **CUDA Local** : Gain **~47 to ~48**
  - each T° ~3.10s, Real 2m39, User 2m37, Sys 1s
- And the winner is ... **Local OpenCL** (no, its programmer...)

Filtering : From Numpy to PyFFT

- Abbe et Porter set : a Python Matplotlib
- Fraunhofer diffraction → Double FFT
  - Main core computing : FFT and FFTshift
- Realtime Numpy/FFT impossible large arrays
- Use PyFFT based on PyOpenCL & PyCUDA
- Speed up depending of board !

PyFFT : Piece of c(ake|ode)...

- **Standard one : def fraunhofer(source)**

```
return(factor*fftshift(fft2(fftshift(source))))
```

- **OpenCL one :**

```
gpuplan = Plan((dim_x, dim_y), normalize=True, fast_math=True, queue=queue)
gpu_data = cl_array.to_device(ctx,queue,fftshift(source))
gpuplan.execute(gpu_data.data)
return(factor*fftshift(gpu_data.get()))
```

- **CUDA one :**

```
gpuplan = Plan((dim_x, dim_y), normalize=True, stream=stream)
gpu_data = gpuarray.to_gpu(fftshift(source))
gpuplan.execute(gpu_data)
return(factor*fftshift(gpu_data.get()))
```

PyFFT : Initialization

- **On CUDA implementation :**

```
from pyfft.cuda import Plan
import pycuda.driver as cuda
from pycuda.tools import make_default_context
 import pycuda.gpuarray as gpuarray
cuda.init()
context = make_default_context()
stream = cuda.Stream()
```

- **On OpenCL implementation :**

```
from pyfft.cl import Plan
 import pyopencl as cl
 import pyopencl.array as cl_array
ctx = cl.create_some_context(interactive=False)
queue = cl.CommandQueue(ctx)
```

Multi-X on Python
What can you expect ?

- **Context :** large initial sets & simplistic simulation
- **X=(nodes|cores)** : Good scaling (gain of 9/10 on example)
  - 5 operations : define pool, create jobs, distribute, compute, collect
  - Quick gain : minimum effort (slightly greater for MPI)
- **X=shaders** : Good performances (gain 50 to 70 on example)
  - Need recoding → need “learn GPU”
  - Large gain : maximum effort
  - Having (or Waiting for) APIs to perform operations : FFT (CuBLAS)
  - GPU MUST be compared as a analog computer (not a black box)

Python : Best Way to learn (and use) !

Multi-X on Python

How do you call people speak 3 languages ?
Trilingual people !

How do you call people speak 2 languages ?
Bilingual people !

How do you call people speak 1 language ?
French people !

I'm french : if questions, speak slowly !