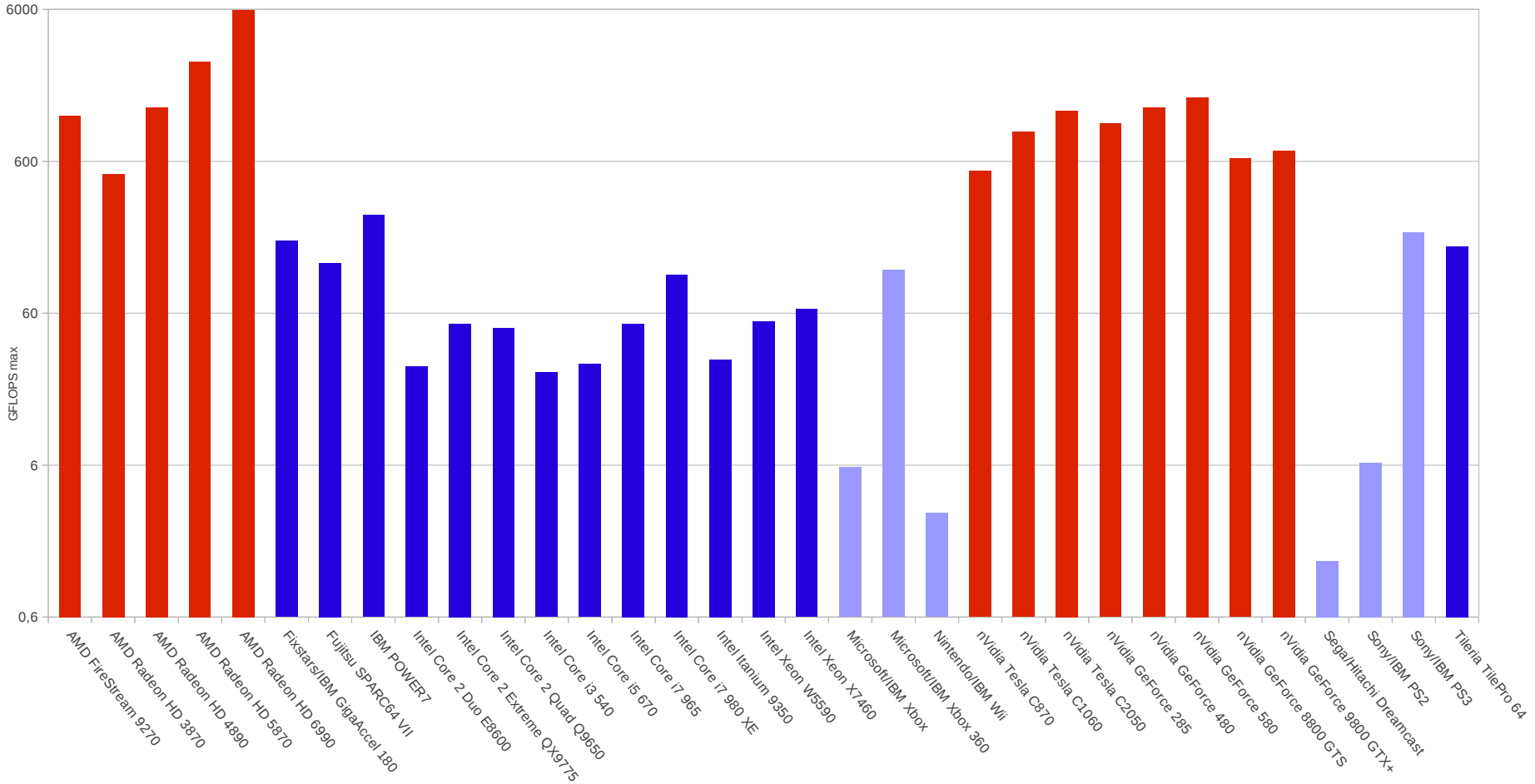
INVENTEURS DU MONDE NUMÉRIQUE

ATELIER GPU
JDEV 2011

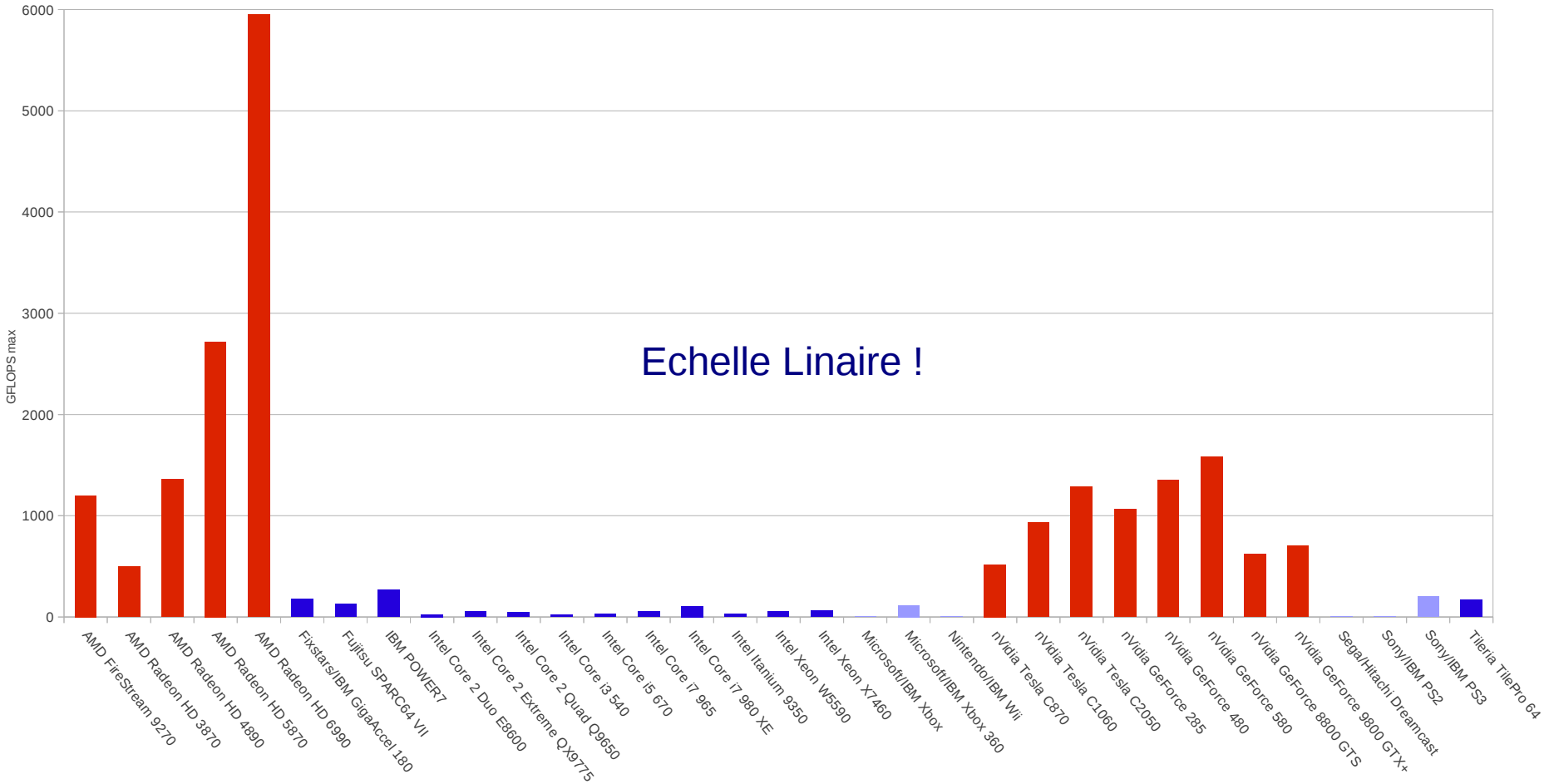

```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf)

print la.norm(a_plus_b - (a+b))
```

INTRODUCTION : pourquoi le GPU ?



Source : http://jonon.gs/blog/computers/gflop-comparison-table-of-cpus-and-gpus/

INTRODUCTION : pourquoi le GPU ?

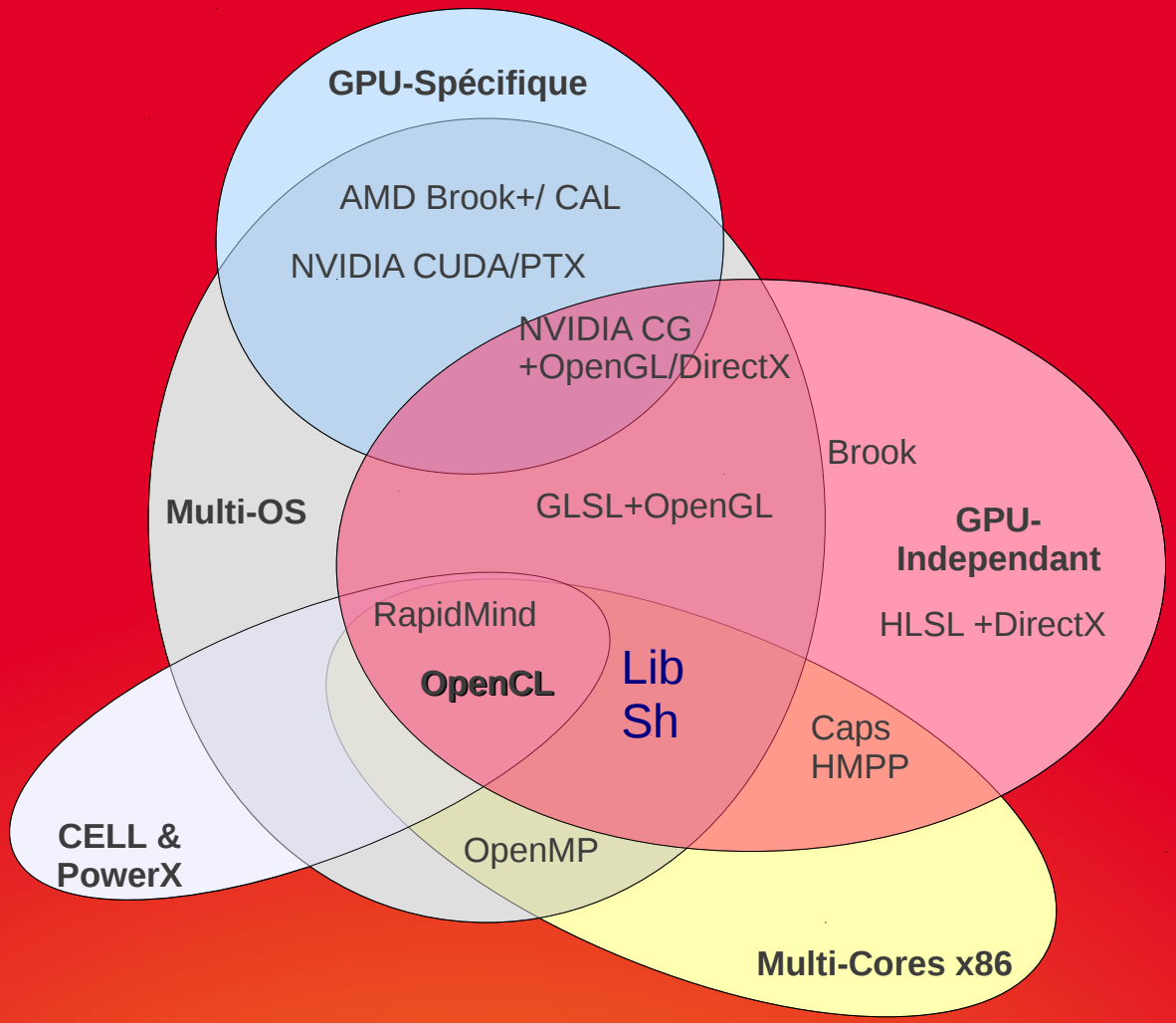


Source : http://jonon.gs/blog/computers/gflop-comparison-table-of-cpus-and-gpus/

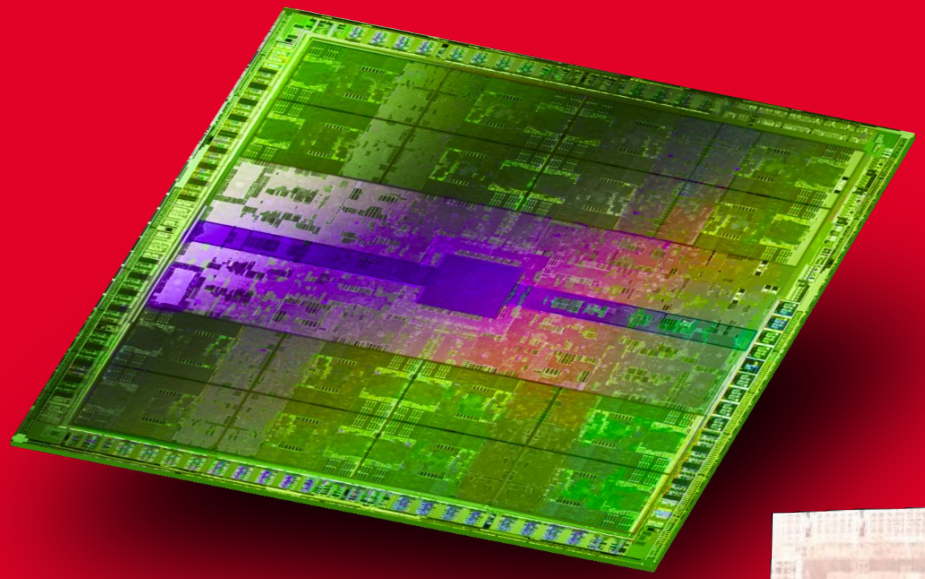
INTRODUCTION : pourquoi le GP-GPU ?

- GPUs **massivement parallèles**
- Bande passante interne très importante
- Calcul en virgule flottante 32 bits (IEEE 754, depuis NVIDIA G200)
- Langages de haut et bas niveau pour programmation générique:
  - NVIDIA CUDA / PTX sur GF1X0, G2XX, G9X et G80
  - AMD Brook+ / CAL sur RV870, RV770, R7xx et R600
  - OpenCL
- Grande série => coût réduits
- Efficacité énergétique: plus de 6 Gflops / watts

INTRODUCTION : pourquoi OpenCL?



SOMMAIRE



1

Architectures de GPU

(mais pas seulement)

Architecture NVIDIA (dans le commerce)

NVIDIA GeForce GTX 580 (GF110)

- 16 = 4 x 4 Streaming Multiprocessors (SM)
  - 512 ALUs scalaires = 16 x 32
  - 64 SFU = 16 x 4
  - 1544 Mhz

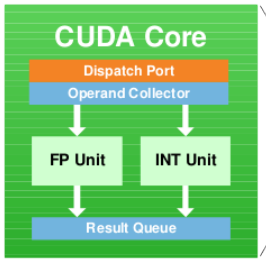
  => **1,581** TFLOPS (32 bits)

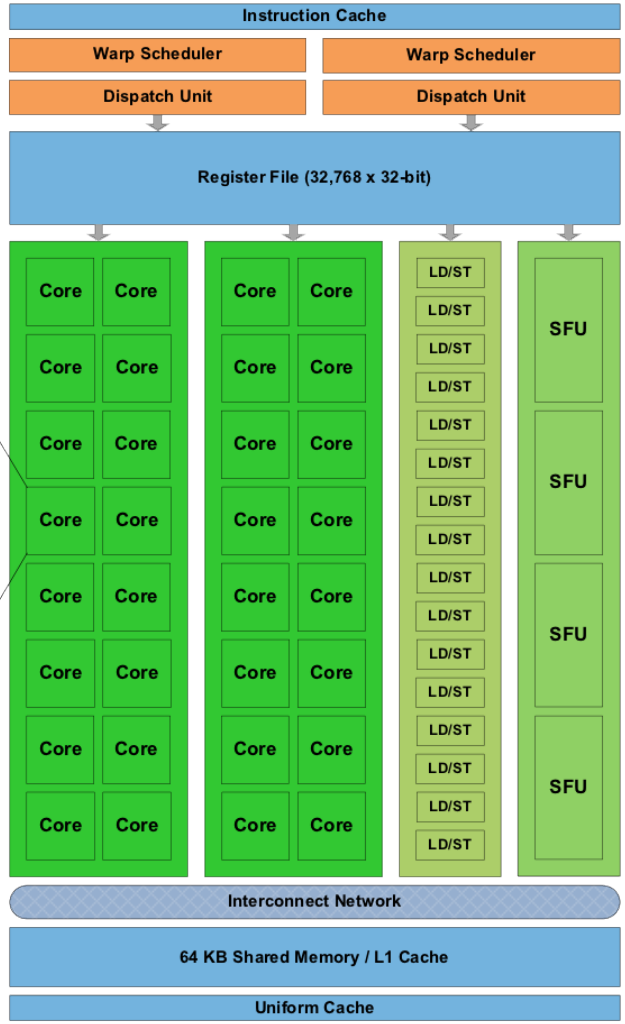
- Memoire
  - Jusqu'à 3 Go RAM (GDDR5)
  - Bus mémoire 384 bits
  - 2004 Mhz

  => Bande passante **192,4** GB/s

L'unités de traitement NVIDIA: Streaming Multiprocessor (SM)

- 32 Cœurs Cuda
  - 1 Cœur Cuda (1 scalaire)
    - 1 Flottant IEEE 754-2008 (32 bits) ou
    - 1 Entier (32 bits)
    - 1 unité de branchement
- 16 LD/ST (Load/Store units) 64 bits
- 4 SFU (Special Function Units) :
  - Sin, cos , sqrt, rsqrt
- 32x2 (FMA) FLOPS 32 bits
- 16x2 (FMA) FLOPS 64 bits
- 64 Ko mémoire partagée ou Cache L1 (48/16 ou 16/48)
- 32K registres de 32 bits.
- Dual Warp Scheduler (jusqu'à deux warps en //)

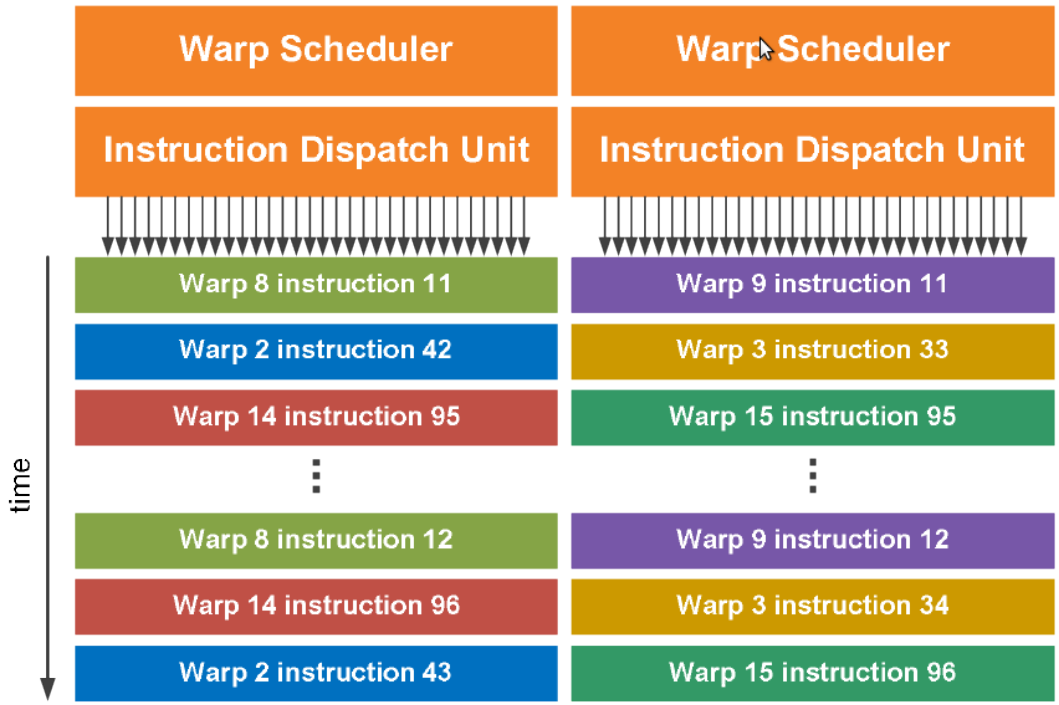




Fermi Streaming Multiprocessor (SM)

Source : http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

Modèle d'exécution NVIDIA



- Warps = 32 Threads
- 2 warps indépendants si :
  - Opérations sur entiers : 2 cycles sur 16 cœurs
  - Opérations flottants 32 bits: 2 cycles sur 16 cœurs
  - Opérations sur SFU : 8 cycles sur 4 SFU.
  - Opérations LD/ST : 2 cycle sur 16 LD/ST
- Un seul Warp si Opérations sur flottants ou entiers 64 bits

Source : http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

Architecture AMD (dans le commerce)

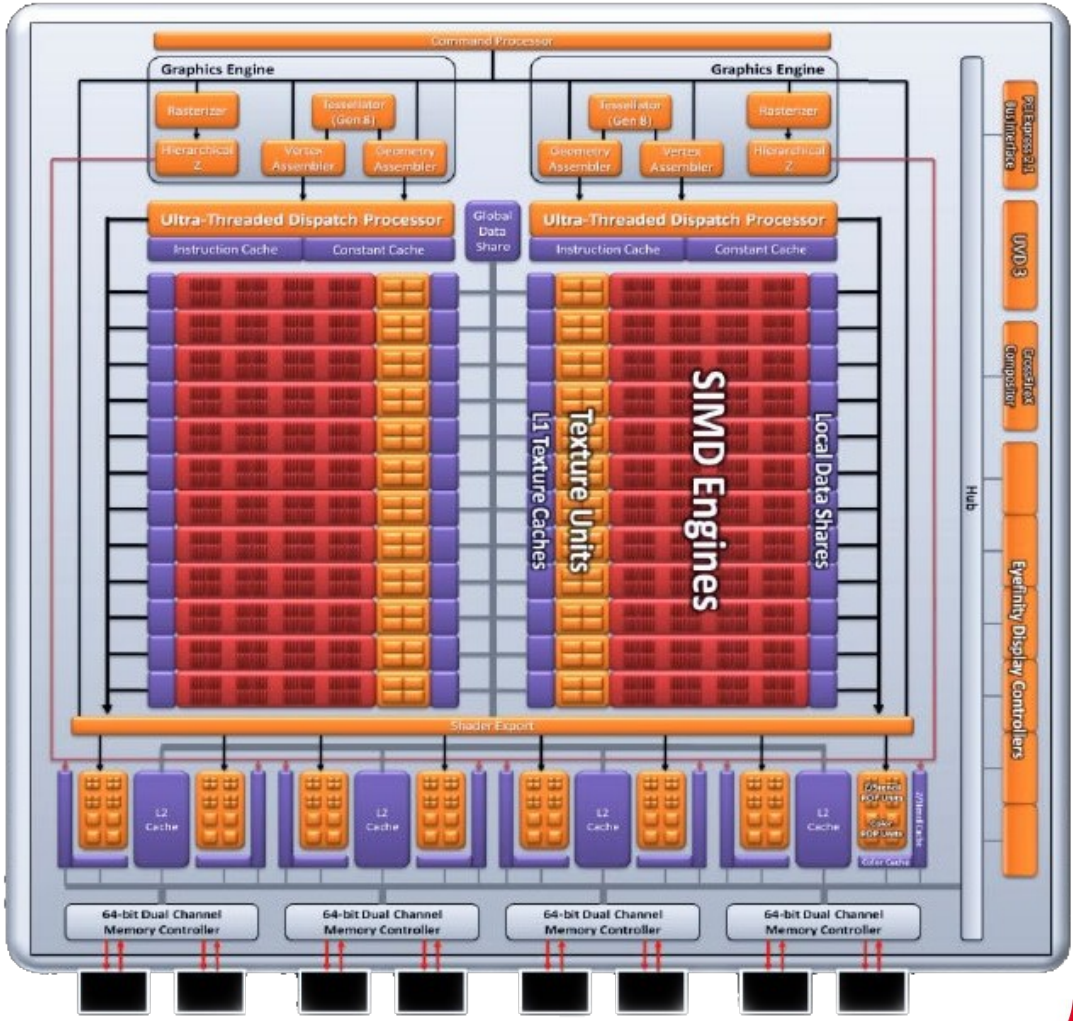
ATI Radeon HD 6970 (Cayman XT)

- 24 Cœurs SIMD
  - 16 VLIW4 :
    - [3 ALU ou 1 SFU] + 1 ALU
  - 1536 ALUs = 24 x 16 x 4
  - 880 Mhz

  => **2,703** TFLOPS

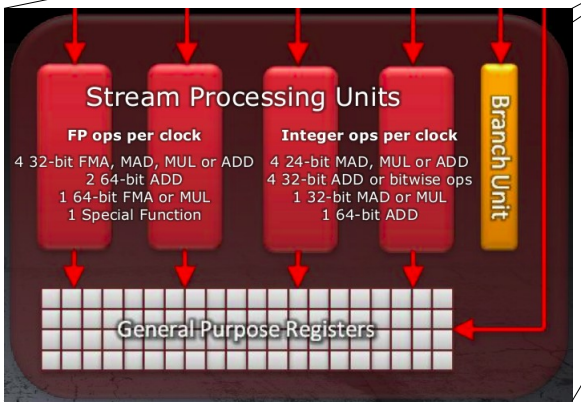
- Mémoire :
  - 2 Go RAM (GDDR5)
  - 1375 Mhz
  - Bus 256 bits

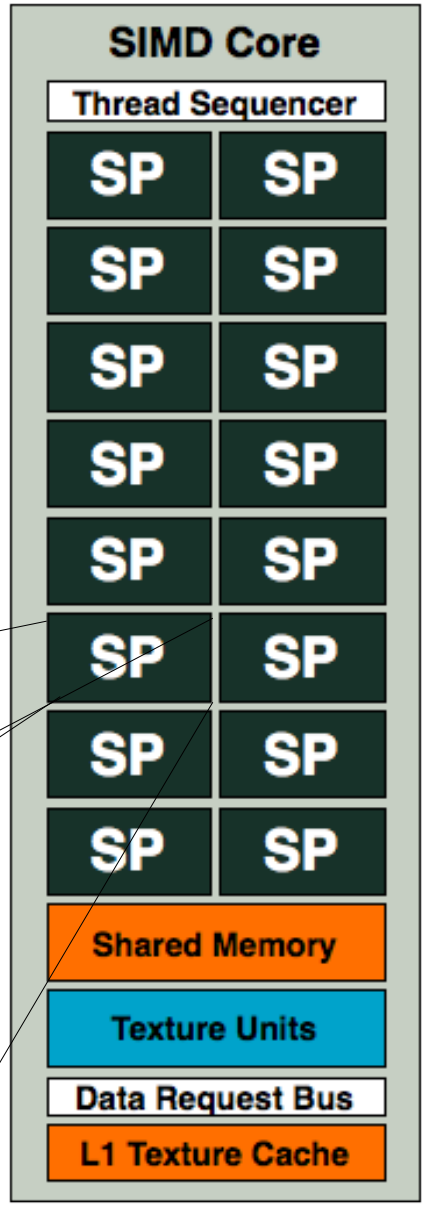
  => Bande passante **176** GB/s



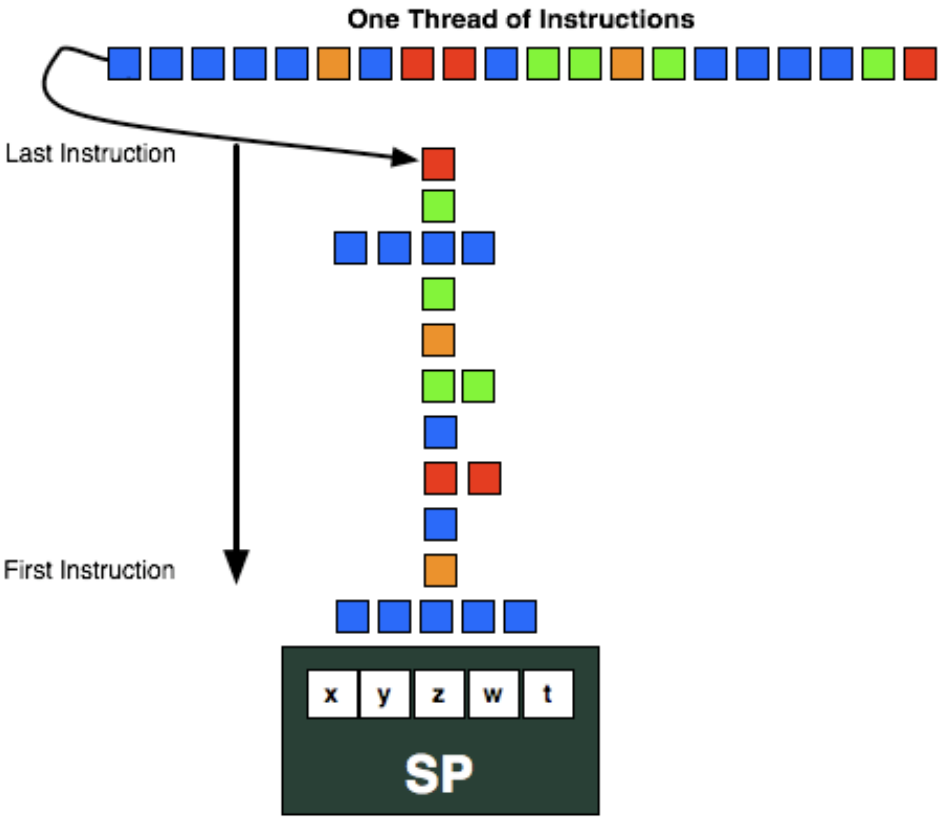
L'unités de traitement AMD: SIMD Engine 16 voies

- 16 SP (Stream multi-processor)
- 32Ko Mémoire Partagée
- 8ko Texture L1 Cache
- Wavefront 4 cycles x 16
- 1 SP :
  - VLIW4
  - Registres :
    - 256 x 128 bits





Execution sur un VLIW



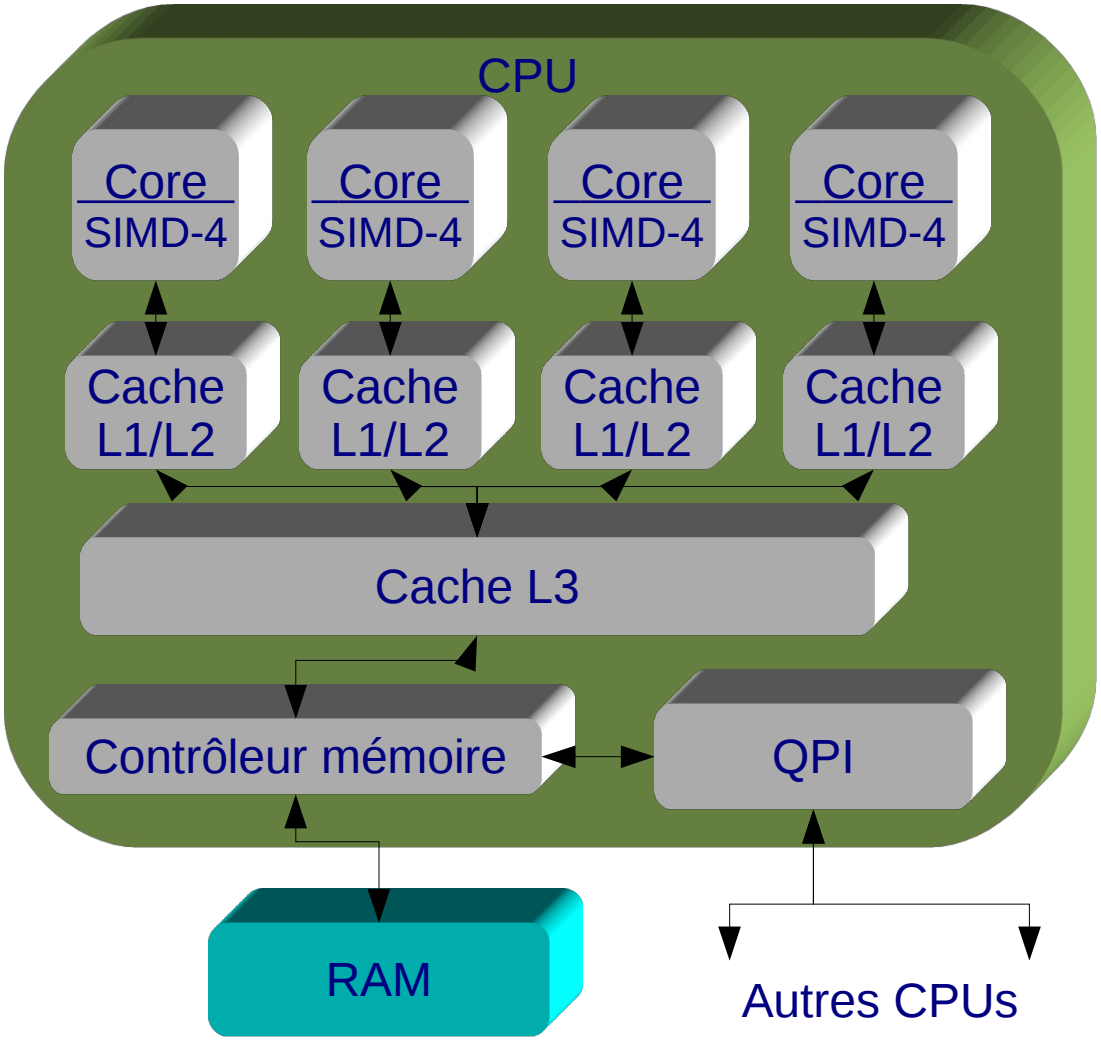
- Une série d'instructions sont partagée entre les differents composants du VLIW (ici VLIW5) à la compilation.

✔ Jussqu'à 4 (ou 5) instructions en parallele !

✗ La dépendance entre les instructions entraine une perte de performance.

Intel Core i7

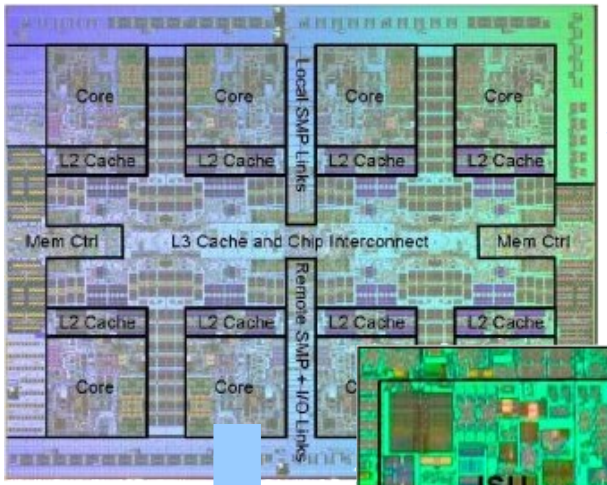
- 4 Cœurs, pour chaque cœur :
  - SIMD-4 Flottants 32 bits
  - *Out-Of-Order*
  - Prediction de branche
- Hyper-threading
  - 8 threads en simultanée
- QPI : Inter-connection point à point entre CPUs
  - Jusqu'à **4 CPU** soit :
    - **4**x4x2=32 threads en simultanée!
    - **4**x4x(4+1) = 80 Opérations Flottantes 32 bits par cycle => 256 GFLOPS @ 3.2 GHz

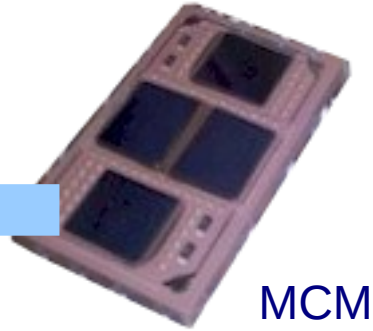


IBM Power 7

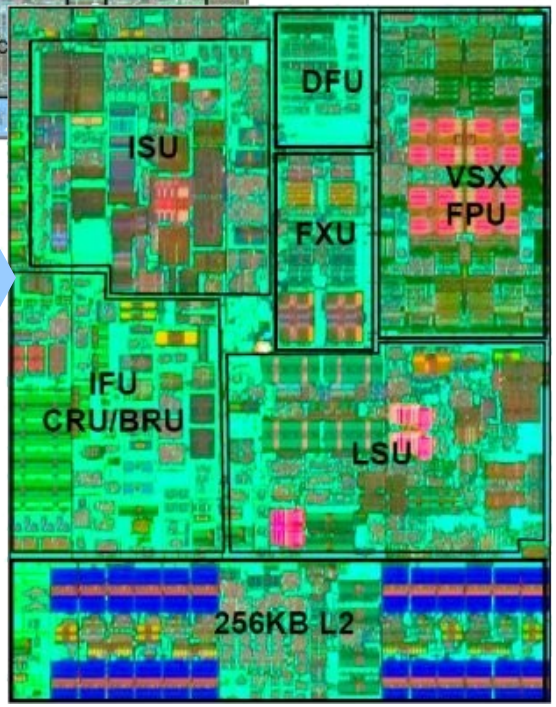
- 1 Multi-Chip Module (MCM)
  - 100 GB/s bande passante DDR3 RAM
  - 4 puces
  - Chaque puce :
    - 32 Mb Cache L3
    - 8 Cœurs
    - Chaque Cœur :
      - VSX = SIMD-4 Flottants 32 bits ou SIMD-2 Flottants 64 bits
      - 2 Flottants 64 bits x2 (FMA)
      - 1 Decimal Floating Point Unit 128 bits
      - 256 Kb Cache L2
      - *Out-Of-Order*
      - 4 way SMT= 4 threads en simultanée
  - **Par MCM :**
    - **4**x8x4=128 threads en simultanée!
    - **4**x8x(4+2x2) = 256 Opérations Flottantes 32 bits par cycle => **1059.8** GFLOPS @ 4.14 Ghz (**264,96** GFLOPS par puce)

puce





MCM

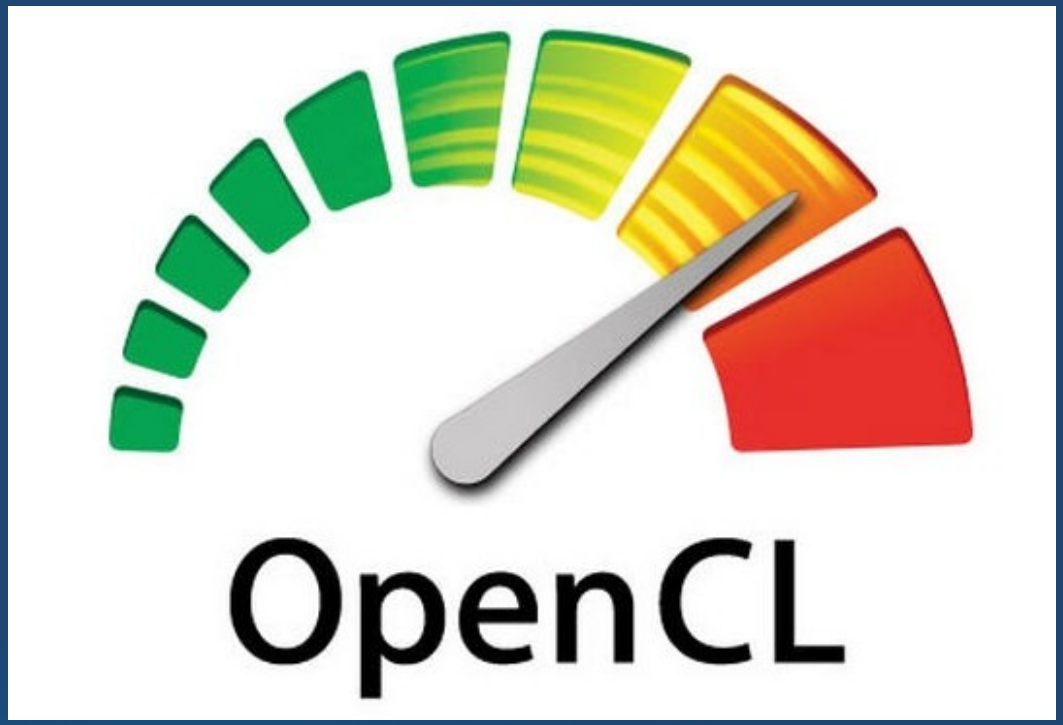


Cœur

Source: http://www.spscicomp.org/ScicomP16/presentations/Power7_Performance_Overview.pdf

2

OpenCL



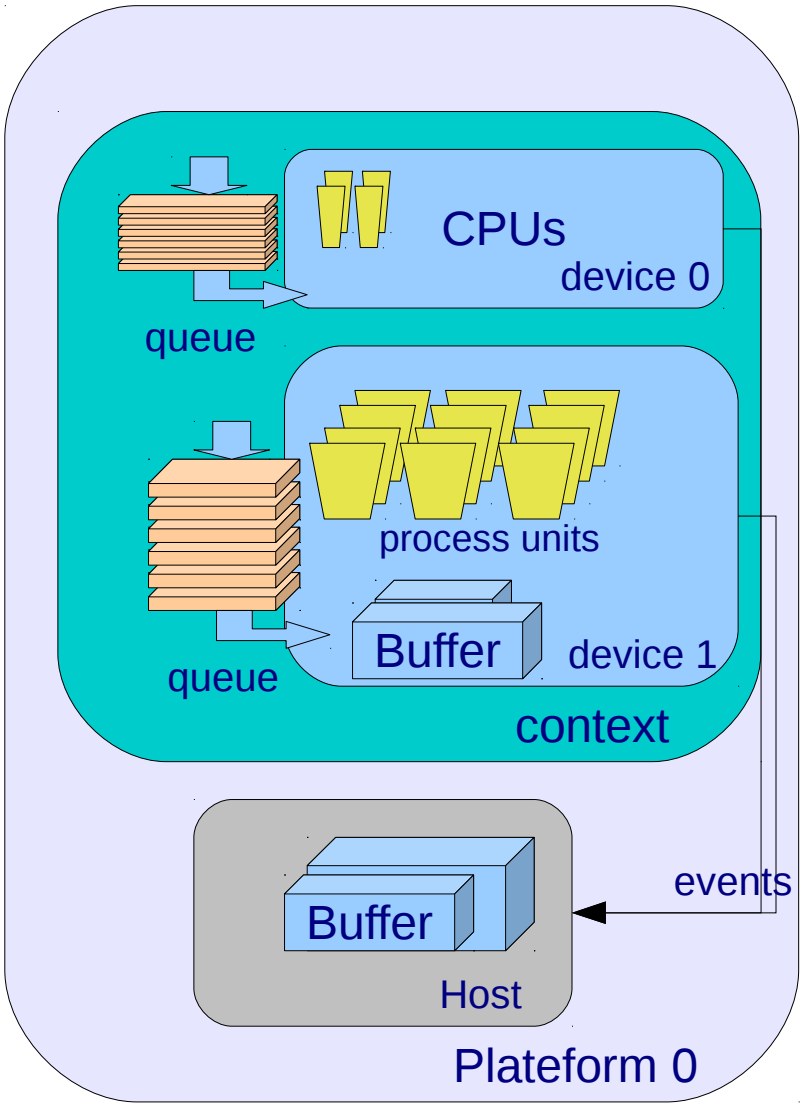
S'abstraire du matériel (ou presque)

OpenCL

- Standard ouvert promu à l'origine par Apple
  - Géré par KHRONOS Group, un consortium d'entreprises : AMD, Intel, Apple, NVIDIA, ARM, IBM...
- API commune en C et noyaux en C99
  - Exécutables en parallèle sur plusieurs CPUs,GPUs ou CELL/BE
  - Gestion des contextes multiples
  - Gestion explicite du transfert mémoire
  - Exécution synchrone ou asynchrone

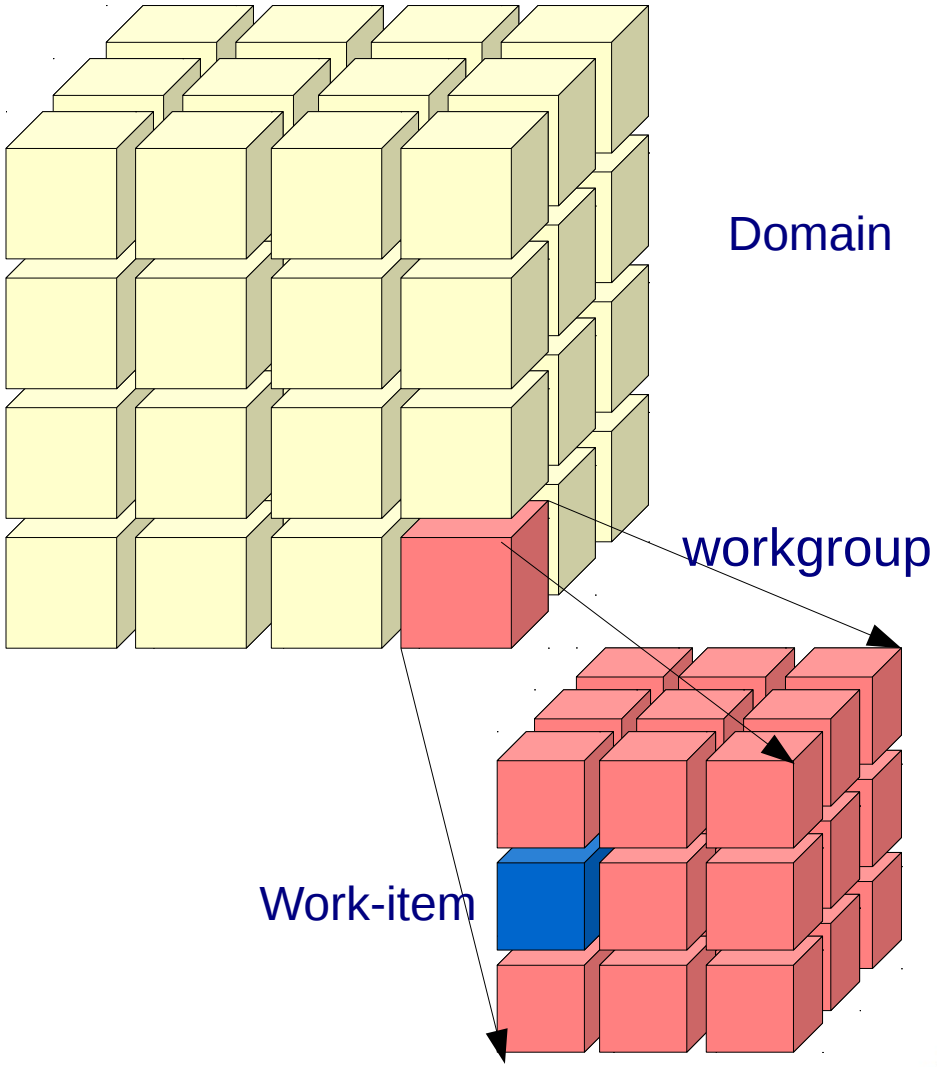
Abstraction du Matériel

- *contexte* : collection de plusieurs dispositif de calcul (*devices*)
  - GPUs
  - CPUs (cores + SSE)
  - CELL BE
- *program* : Collection de noyaux *kernels*
  - *Kernel : function s'exécutant dans un process unit*
- Mémoires :
  - *Buffers* :1D
  - *Images* : 2D ou 3D
- Une *queue* associée à chaque *device* des ordres :
  - d'exécution de noyaux
  - copie de mémoires
- *events* : synchronisation des exécutions entre *devices*



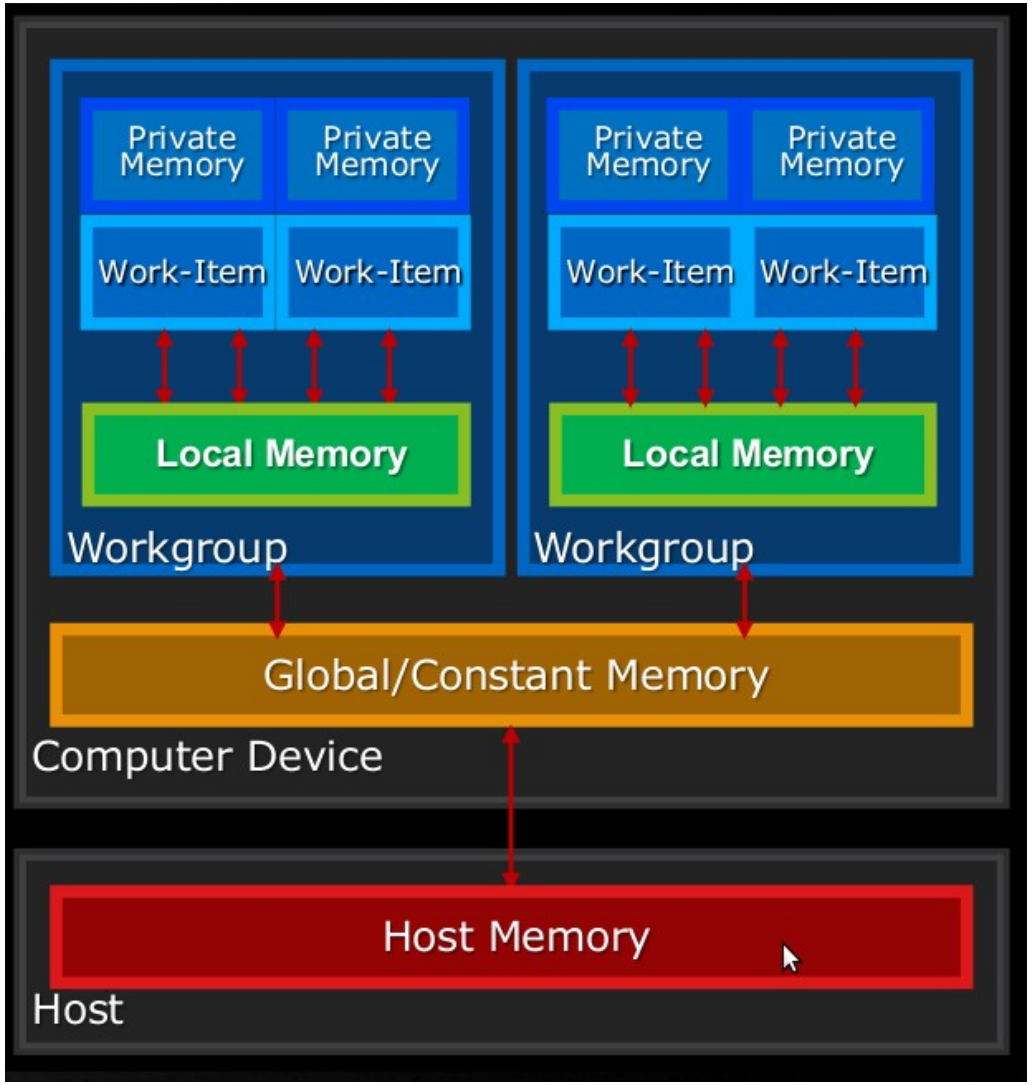
Décomposition du traitement

- Domaine du traitement à N-dimension (Nmax=3)
  - *global dimensions*
  - Décomposition en groupes *workgroups*
- Décomposition de chaque groupe en éléments de traitement
  - *Work-items*
  - *Local dimensions*



Modèle de Mémoire

- Mémoire privée (*private memory*) à chaque *work-item*
- Mémoire partagée (*local memory*) entre *work-items* d'un même *workgroup*
- Mémoire globale du *device*
- Mémoire du hôte



Le « Hello Word ! » du calcul avec OpenCL

- Addition de deux tableaux **A** et **B** de taille n et stocker le résultat dans un tableau C de même taille.

$$C_i = A_i + B_i$$
$$i \in [0, n-1], i \in \mathbb{N}$$

1. Réserver la mémoire dans le GPU pour recevoir les données A et B et le résultat C
2. Copier les données A et B de la RAM du PC dans le GPU
3. Lancer le calcul sur le GPU de la somme
4. Transférer le résultat dans le tableau C dans la RAM du PC

Le noyeau

```
// OpenCL Kernel Code
__kernel void
vectorAdd(__global const float * a,
          __global const float * b,
          __global float * c)
{
  // Vector element index
  int nIndex = get_global_id(0);
  c[nIndex] = a[nIndex] + b[nIndex];
}
```

Obtenir le contexte et la liste des GPUs

```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,CL_DEVICE_TYPE_GPU,
0, 0, 0);

// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
0, 0, &nContextDescriptorSize);

cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
nContextDescriptorSize, aDevices, 0);
```

Créer la queue sur le premier GPU et compiler le noyau

```
// create a command queue for first device the context
reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0],
0, 0);
// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,
                                     sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0) ;
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vectorAdd", 0);
```

Créer A, B et C et copier les données dans le GPU

```
// allocate host vectors
float * A = new float[n];
float * B = new float[n];
float * C = new float[n];
...
// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             n * sizeof(cl_float),
                             A,
                             0) ;
hDeviceMemB = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             n * sizeof(cl_float),
                             B,
                             0) ;
hDeviceMemC = clCreateBuffer(hContext,
                             CL_MEM_WRITE_ONLY,
                             n * sizeof(cl_float),
                             0, 0);
```

Appeler le noyau et transférer le résultat à la RAM du PC

```
// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void
*)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void
*)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void
*)&hDeviceMemC);
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                       &n, 0, 0, 0, 0);
// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
                    n * sizeof(cl_float),C, 0, 0, 0);
```

Faire le ménage en partant ...

```
delete[] A;
delete[] B;
delete[] C;
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
```

Utilisation de la mémoire locale

```
__kernel void transpose(
          __global float *a_t, __global float *a,
          unsigned a_width, unsigned a_height,
          __local float *a_local)
        {
            int base_idx_a   =
              get_group_id(0) * get_local_size(0) +
              get_group_id(1) * a_width;
            int base_idx_a_t =
              get_group_id(1) * get_local_size(1) +
              get_group_id(0) * a_height;

          int glob_idx_a   = base_idx_a + get_local_id(0) + a_width *
get_local_id(1);
          int glob_idx_a_t = base_idx_a_t + get_local_id(0) + a_height *
get_local_id(1);

          a_local[get_local_id(1)*get_local_size(1)+get_local_id(0)] =
a[glob_idx_a];

          barrier(CLK_LOCAL_MEM_FENCE);

          a_t[glob_idx_a_t] =
a_local[get_local_id(0)*get_local_size(0)+get_local_id(1)];
        }
```

3

PyOpenCL

Soyez le chef de l'orchestre !

Pourquoi Python ?

- Langage généraliste
  - Orienté Objet
  - Typé dynamiquement
  - Gestion automatique de la mémoire
- Haut niveau :
  - Interprété
  - librairies natives de haute performance
    - Numpy (aussi rapide qu'en C)
    - ... (milliers d'autres)
- Multi-OS
- OpenSource et communauté très active
- **http://docs.python.org/tutorial/introduction.html**

Pourquoi PyOpenCL ?

- Interface d'OpenCL pour le langage Python
- Simplification de l'interface mais garde la totalité de possibilités
- Ajout de fonctions de haut-niveau :
  - Map/ Reduce
  - Générateur de nombres aléatoires
  - FFT via pyfft
- OpenSource licence type MIT

http://documen.tician.de/pyopencl/

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(num
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Ajouter la librairie à python

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOS
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Initialisation des données côté PC (host) avec numpy

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Initialisation et
choix interactif des *devices*
à utiliser

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Accéder à la queue associée au contexte

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Réserver et copier les données A et B dans le *device*

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Réserver la mémoire pour loger le résultat C dans le *device*

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_b
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).
print la.norm(a_plus_b - (a+b))
```

Déclaration et compilation du noyau de calcul

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Appel au noyau dans le *device*

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Transfert du résultat dans la mémoire du PC (host)

4

Exercice

Mettre les mains à la pâte

Exercice 0 : Ce connecter au serveur

```
ssh -XC jdev{a-y}@soroban.irisa.fr
```

- Serveur HP Z800 :
  - Ubuntu 11.4
  - 2 CPU Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
    - Soit 8 cores hyper-threads
    - SSE 4.2
  - 2 Cartes graphiques AMD ATI FirePro V7800 (FireGL)
    - **Aimablement mises à disposition pour cet atelier par AMD france**
    - Unités de traitetement : 1440
    - Mémoire :
      - 2GB GDDR5
      - bande passante : 128.0 Go/s

Exercice 0.1 : voir ce qu'on a sous le capot

Dans le terminal de calcul :

```
export DISPLAY=:0.0
python /opt/pyopencl/examples/dump-properties.py | less
```

```
===========================================================================
<pyopencl.Platform 'ATI Stream' at 0x7f870b003880>
===========================================================================
EXTENSIONS: cl_khr_icd cl_amd_event_callback cl_amd_offline_devices
NAME: ATI Stream
PROFILE: FULL_PROFILE
VENDOR: Advanced Micro Devices, Inc.
VERSION: OpenCL 1.1 ATI-Stream-v2.3 (451)
---------------------------------------------------------------------------
<pyopencl.Device 'Cypress' at 0x2822510>
---------------------------------------------------------------------------
...
```

Exercice 1 : Modifier le banc d'essai

Dans un **AUTRE** terminal :

```
ssh -XC jdev{a-y}@soroban.irisa.fr

cp /opt/pyopencl/examples/benchmark-all.py tp1.py

gedit tp1.py
```

Objectif :

- Remplacer le code natif python par du code numpy
  - Optimal pour un seul cœur sans SSE dans la version standard de Ubuntu
- Paramétrer la taille des données avec une variable `numData`=100000
- Enlever la boucle dans le noyau (variable `loop`) et la boucle en CPU (variable `j`)

Exercice 2 : Problème à N corps

- Application
  - Astronomie
  - Physique des matérieaux
  - Simulation mécanique
  - Simulation de foules

Définition

- Particules

$$n \gg 1$$

- Calcul de l'interaction pour chaque particule

$$i \in [0, n-1]$$

- Avec (ici pour simplifier) :

$$F(i) = \sum_{j=0}^{j=n-1} f(x_i, x_j)$$

$$f(x_i, x_j) = x_i - x_j$$

Version C sur CPU

```
inline float f(float xi, float xj)
   { return xi - xj; }
void calcul( float* F, float* X, unsigned int n)
   {
      for( unsigned int i = 0; i < n; i++)
         {
            F[i]=0.0f;
            for( unsigned int j = 0; j < n; ++j)
               F[i] = F[i] + f(X[i],X[j]);
         }
   }
```

Version Python (courte) sur CPU

```
def f(x_j,x_i) :
    return x_j - x_i
import numpy
def calcul(X) :
   return numpy.float32([numpy.sum(f(X,X_i))
for X_i in X])
```

Version 0 OpenCL (noyau)

```
float f(float x_i, float x_j)
{return x_i - x_j;}
__kernel void sum(__global const float *X,
__global float *F, uint n)
{
        int gid = get_global_id(0);
        F[gid] = 0.0f;
        for (uint j=0;j<n ; j++)
            F[gid] = F[gid] + f(X[gid],X[j]);
}
```

5

Performance

Ah !

Résultats

Pour n=128 * 250 * 4 soit 128 000 particules sur le même PC nous avons les résultats suivants :

n=128000	Code C (référence)	Python+ numpy	PyOpenCL sur CPU (SDK AMD)	PyOpenCL sur AMD ATI FirePro F7800	PyOpenCL sur AMD ATI Radeon HD 5870
Temps total (Sec.)	19,64	40,96	4,02	3,91	3,24
Sans temps de transfert (Sec.)			3,95	3,83	3,15
Accéleration (transfert compris)	1	0,49	4,88	5,01	6,07
Puissance avec transfert (GFLOPS)	0,83	0,40	4,08	4,19	5,05

Premières observations

- Le code OpenCL « simple » exploitant la carte graphique permet de gagner un facteur 5 en vitesse par rapport au code « simple » en C
- Le rapport de performance entre les cartes graphiques est linéairement proportionnel au nombre d'unités de calcul et à leur fréquence (1400@800 Mhz et 1600@850Hz).
- Le code OpenCL sur CPU via le SDK d'AMD permet d'obtenir un facteur d'accélération proche de 5 par rapport au code « simple » en C en exploitant les multiples cœurs disponibles.
- La version numpy qui manipule des vecteurs de données est moins performante que la version native en C, mais reste compétitive dans un contexte de prototypage rapide.
- La performance obtenue dans cet exemple est accessible au regard du coût de portage relativement faible sur OpenCL et PyOpenCL.

Mais la puissance maximale de calcul obtenue est de 5 Gflops, pouvons nous aller plus loin ?

Exploiter mieux le parallélisme

- Utiliser tous les cœurs disponibles
  - Pour le code C/C++ :
    - Directives OpenMP
    - Librairies multi-threads : Intel TBB, pthread, etc.
- Utiliser les unités de traitement parallèle
  - Instructions SSE2 sur CPU
  - VLIW(4-5) sur cartes ATI AMD

=> Utiliser float4 pour OpenCL

=> Utiliser SSE Intrinsics pour C/C++

Version initiale C/C++

```
inline float f(float xi, float xj)
   { return xi - xj; }
void calcul( float* F, float* X, unsigned int n)
   {

      for( unsigned int i = 0; i < n; i++)
         {
            float f_i=0.0f;
            for( unsigned int j = 0; j < n; ++j)
               f_i = f_i + f(X[i],X[j]);
            F[i]=f_i ;
         }
   }
```

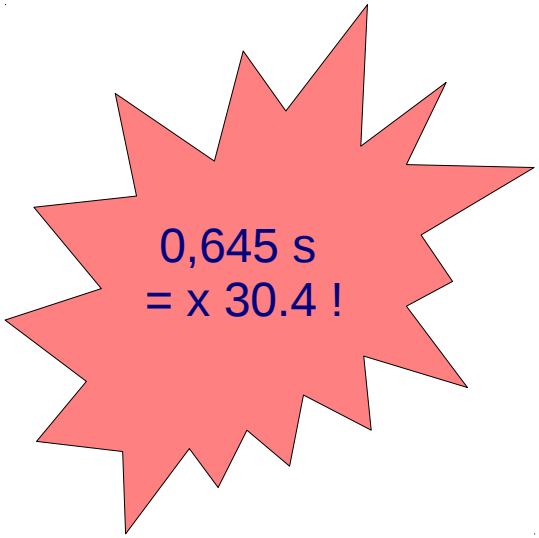
19,64 s

OpenMP pour le code C/C++

```
inline float f(float xi, float xj)
   { return xi - xj; }
void calcul( float* F, float* X, unsigned int n)
   {
      #pragma omp parallel for
      for( unsigned int i = 0; i < n; i++)
         {
            float f_i=0.0f;
            for( unsigned int j = 0; j < n; ++j)
               f_i = f_i + f(X[i],X[j]);
            F[i]=f_i ;
         }
   }
```

OpenMP + SSE pour le code C/C++

```
#include<xmmintrin.h>
inline __m128 f(__m128 x_i, __m128 x_j)
{ return x_i-x_j; }
void calcul( float* F, float* X, unsigned int n)
  {
    #pragma omp parallel for
    for( unsigned int i = 0; i < n; i+=4) {
      __m128 f_i=_mm_set_ps1(0.0f);
      __m128 x_i=_mm_load_ps(X+i);
      for( unsigned int j = 0; j < n; ++j) {
        __m128 x_j=_mm_set_ps1(X[j]);
        f_i = f_i + f(x_i,x_j);
      }
      _mm_store_ps((F_result+i), f_i );
    }
  }
```



Version 1 OpenCL avec Float4 (noyau)

```
inline float4 f(float4 x_i, float x_j)

{return x_i - x_j;}

__kernel void sum(__global const float4 *X, __global float4 *F,
uint n_4)

{

    int gid = get_global_id(0);

    F[gid] = 0.0f;

    for (uint j=0;j<n_4 ; j++) {

       F[gid] = F[gid] + f(X[gid],X[j].x) + f(X[gid],X[j].y)

                       + f(X[gid],X[j].z)+ f(X[gid],X[j].w);

     }

}
```

Nouveaux Résultats

n=128000	Code C + OpenMP+SSE	PyOpenCL float4 sur CPU (SDK AMD)	PyOpenCL float4 sur CPU (SDK AMD) & hyper-threading	PyOpenCL float4 sur AMD ATI FirePro F7800	PyOpenCL float4 sur AMD ATI Radeon HD 5870
Temps total (Sec.)	0,645	0,769	0,632	0,459	0,472
Sans temps de transfert (Sec.)		0,699	0,559	0,381	0,384
Accéleration (transfert compris)	30,44	25,54	31,08	42,80	41,60
Puissance avec transfert (GFLOPS)	25,39	21,31	25,97	35,7	34,7

Nouvelles observations

- L'utilisation des types vectoriels float4 permet un bien meilleur rendement des cartes ATI et des CPUs avec les SDK OpenCL AMD.
  - L'accélération sur OpenCL sur CPU (correspond pratiquement à 32 = 8 cœurs x 4 Flop par SSE avec l'hyper-threading

    => masquage du temps d'accès à la mémoire par du calcul
- La carte ATI FirePro F7800 est légèrement plus rapide que la carte ATI Radeon HD 5870 en contradiction du ratio de puissance brute.

  => la limite ne réside plus dans la puissance de calcul développée mais dans des goulots d'étranglement comme l'accès à la mémoire

Peut on réduire le nombre d'accès à la mémoire et gagner d'avantage de performance ?

Version 1 OpenCL avec Float4 (noyau)

```
__kernel void sum(__global const float4 *X, __global
float4 *F, uint n_4)
{
    int gid = get_global_id(0);
    F[gid] = 0.0f;
    for (uint j=0;j<n_4 ; j++) {
        F[gid] = F[gid] + f(X[gid],X[j].x)
            + f(X[gid],X[j].y) + f(X[gid],X[j].z)
            + f(X[gid],X[j].w);
    }
}
```

n*n accès en lecture

n*n accès en écriture

Version 2 OpenCL avec un registre (noyau)

```
__kernel void sum(__global const float4 *X,
__global float4 *F, uint n_4)
{
    int gid = get_global_id(0);
    Float4 F_i = 0.0f ;
    for (uint j=0;j<n_4 ; j++){
        F_i = F_i + f(X[gid],X[j].x) + f(X[gid],X[j].y)
                  + f(X[i],X[j].z)+ f(X[i],X[j].w);
     }
    F[gid]=F_i ;
}
```

n accès en écriture

Résultats avec l'utilisation d'un registre

n=128000	Code C + OpenMP+SSE	PyOpenCL avec registres sur CPU (SDK AMD)	PyOpenCL avec registres sur CPU (SDK AMD) & hyper-threading	PyOpenCL sur registres sur AMD ATI FirePro F7800	PyOpenCL avec registres sur AMD ATI Radeon HD 5870
Temps total (Sec.)	0,645	0,711	0,450	0,159	0,142
Sans temps de transfert (Sec.)		0,641	0,429	0,0968	0,0793
Accéleration (transfert compris)	30,44	27,61	39,29	123,1	138,1
Puissance avec transfert (GFLOPS)	25,39	23,03	32,77	102,7	115,2

Conclusions

- **L'utilisation des registres permettent d'éviter des réécritures dans la mémoire globale.**
- **Certains accès en lecture sont transformées en registres automatiquement (Cas du X[i]) par le compilateur**
- **L'impact de l'utilisation des registres est moins important sur CPU qui dispose d'un cache en lecture/écriture.**
- **La performance globale est sensiblement améliorée par itérations successives**
- **Il est possible de réduire les accès à la mémoire globale (Cas du X[j]) par l'utilisation de la mémoire locale.**

  => Dans une prochaine version de ce document nous traiterons de ce point.

merci

Toulouse

INSEEHT

http://devlog.cnrs.fr/