Programmation des processeurs multicoeurs

Cours théorique et
Application en OpenCL avec Matrix Studio

Pascal Ballet
pascal.ballet@univ-brest.fr

Université de Bretagne Occidentale
Laboratoire d'Informatique des Systèmes Complexes (EA3883)

Plan général

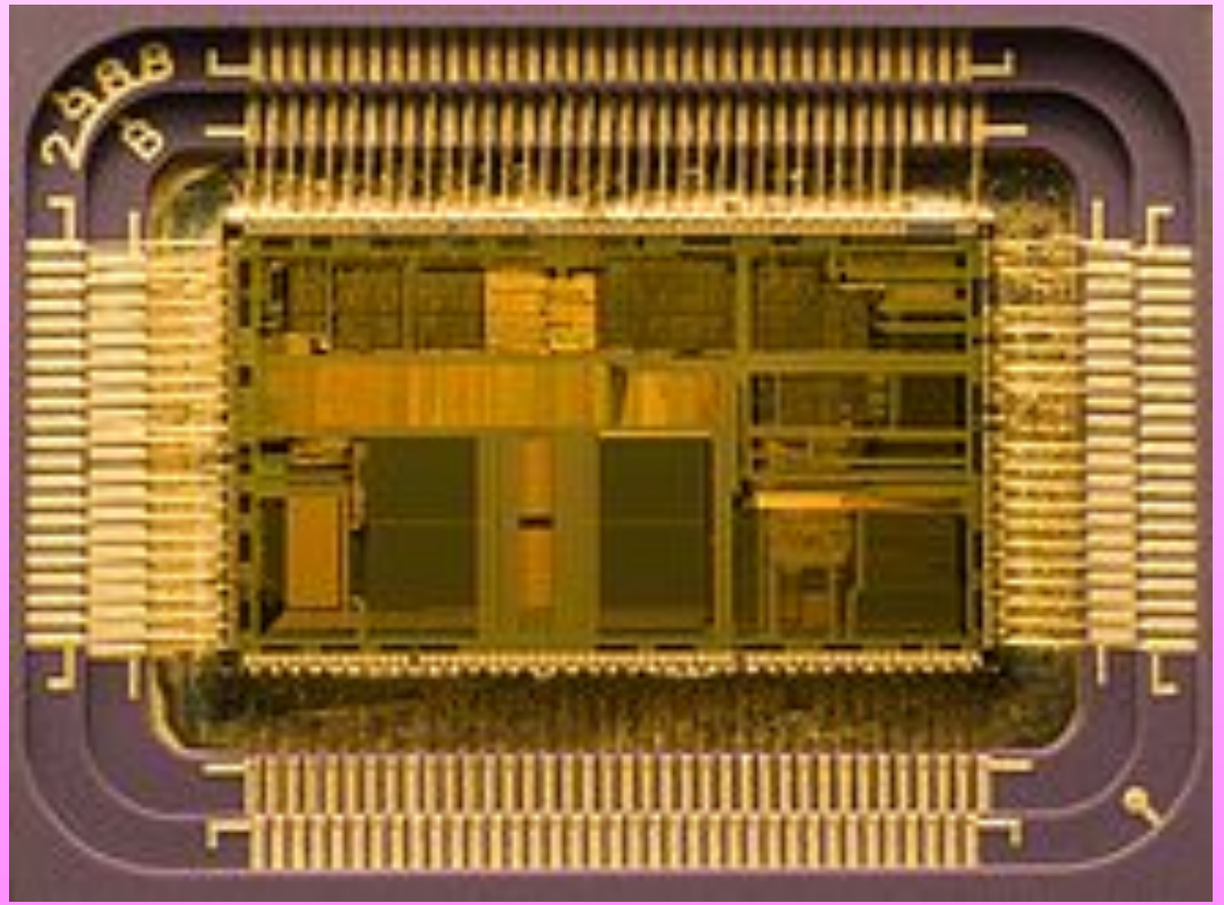
- Introduction sur les processeurs multicoeurs
- La programmation avec OpenCL
- Utilisation de Matrix Studio
- Conclusion & Perspectives

Plan de l'introduction sur les processeurs multicoeurs

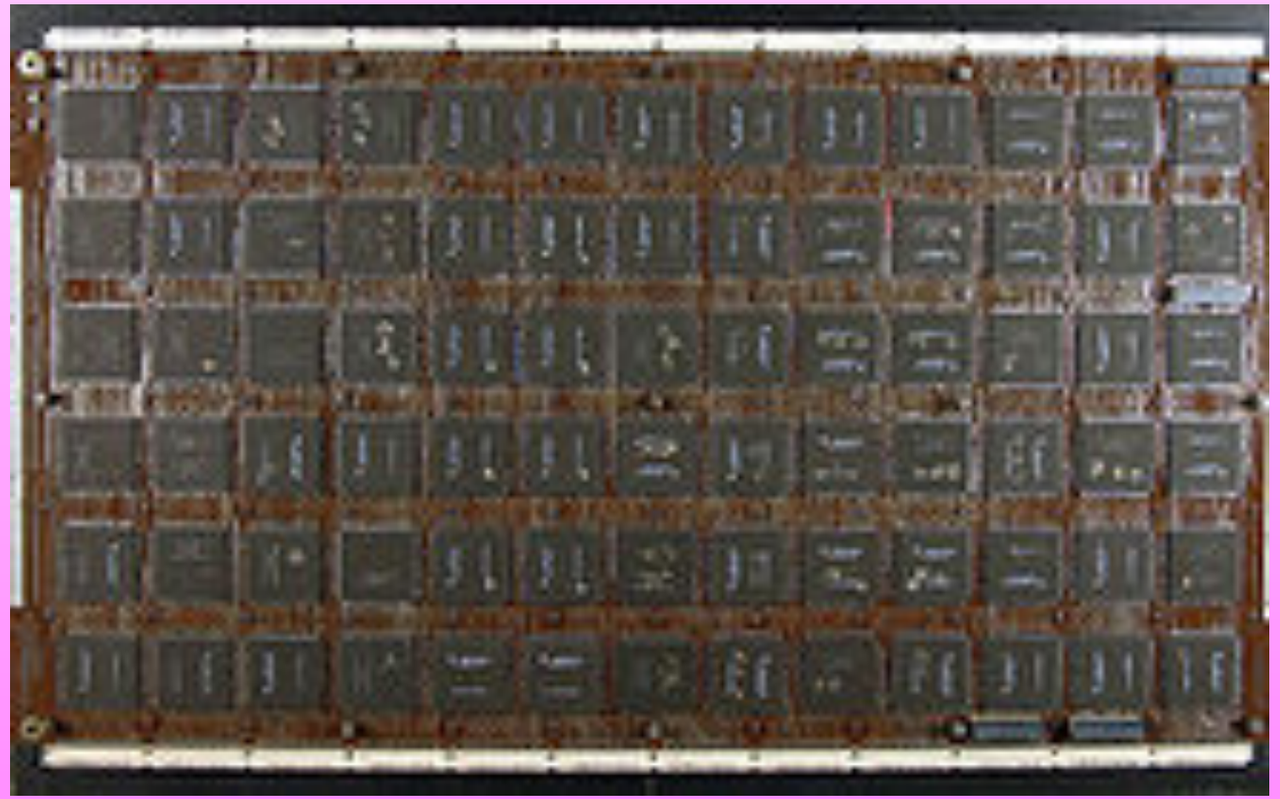
- Terminologies et matériels concernés
- Histoire, enjeux et intérêts des processeurs multicoeurs
- Architecture interne
- Notions de programmation parallèle

Terminologie

- Processeur (CPU)
  - unité centrale de traitement
  - exécute les programmes informatiques



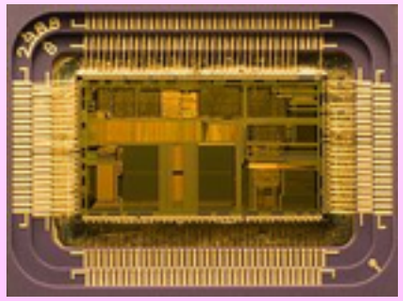
Intel 80486DX2 - Scalaire (Src wikipedia)



Processeur du Cray YMP - Vectoriel (Src wikipedia)

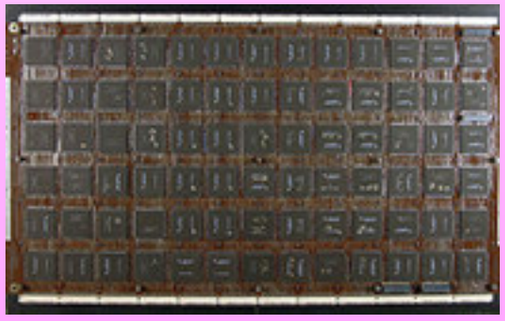
Terminologie

- 2 grands types de processeurs
  - Processeurs Scalaire
    - Travail sur des nombres simples



10 9.4

  - Processeurs Vectoriels
    - Travail sur des vecteurs de nombres



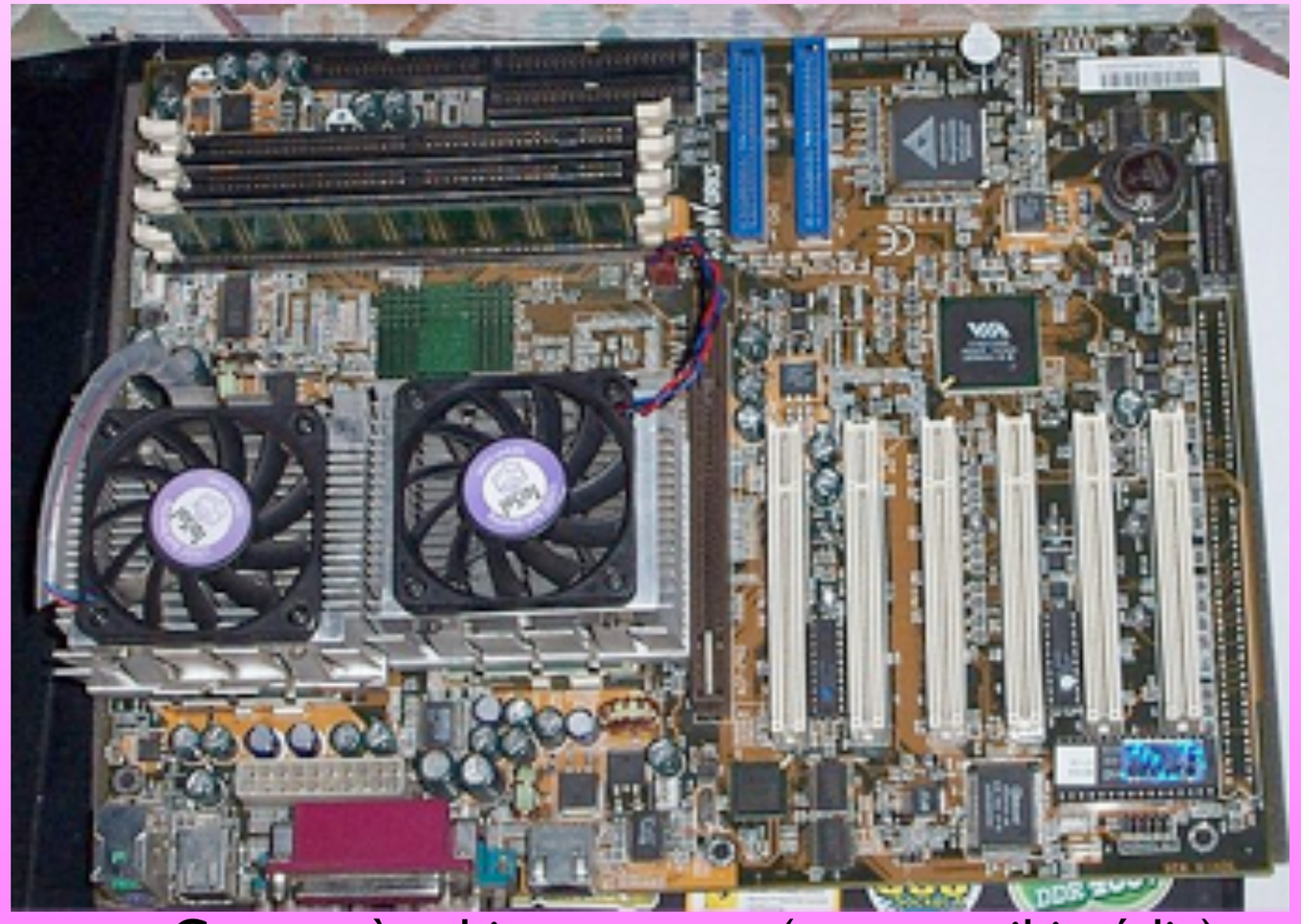
10 | 4 | 12 | 102 | 65

9.4 | 54.3 | 32.8

- Il existe d'autres classifications : hyperthreading

Terminologie

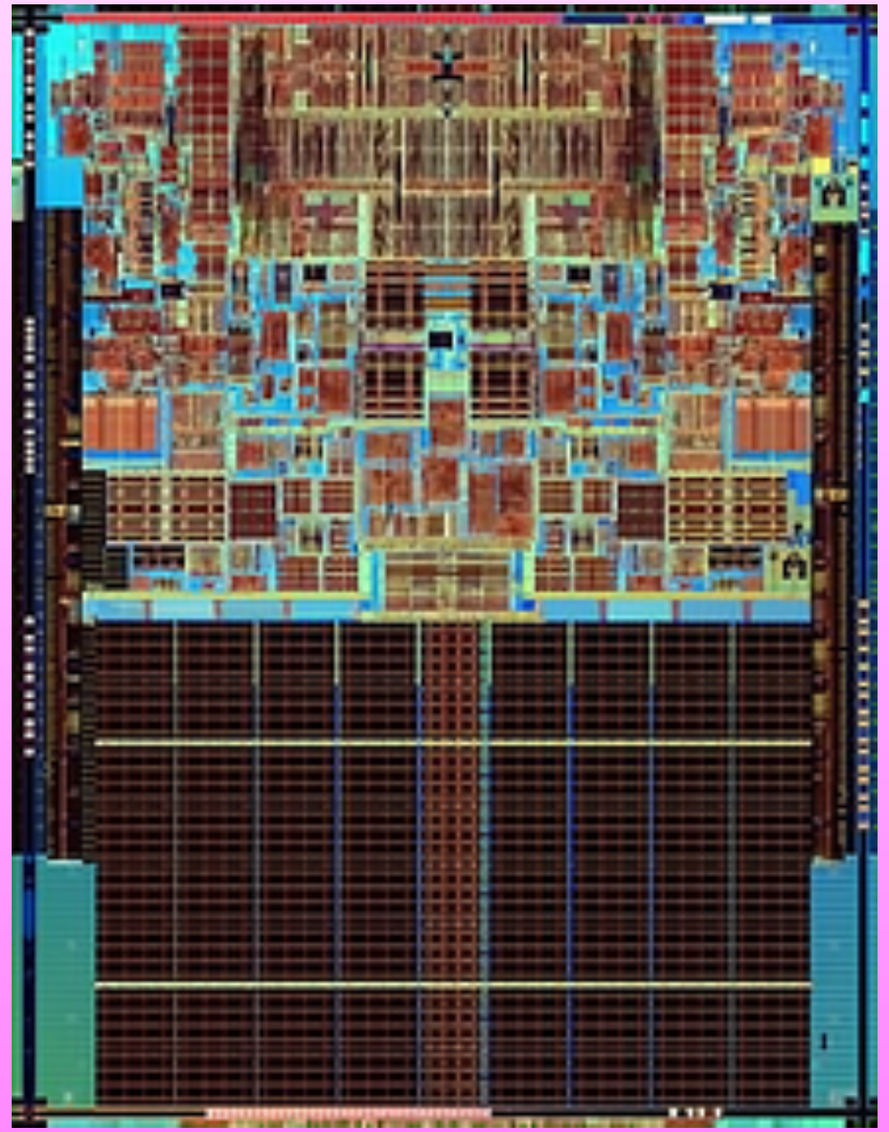
- Multiprocesseur
  - Ordinateur utilisant au moins 2 processeurs
  - Partage des ressources : mémoire, disque...
  - Utilisation en parallèle des processeurs
  - Permet d'augmenter la puissance de traitement à vitesse d'horloge constante (limitée à 3.8 GHz cause t° et 130 Watts)



Carte mère biprocesseur (source wikimédia)

Terminologie

- Processeur Multicoeurs (multicore)
  - Processeur contenant au moins 2 unités de calcul
  - gravées sur la même puce
  - Mémoire communes (RAM) + mémoires séparées (Cache)
  - Même idée que le multiproc mais solution plus élégante (vitesse et économie par ex.)

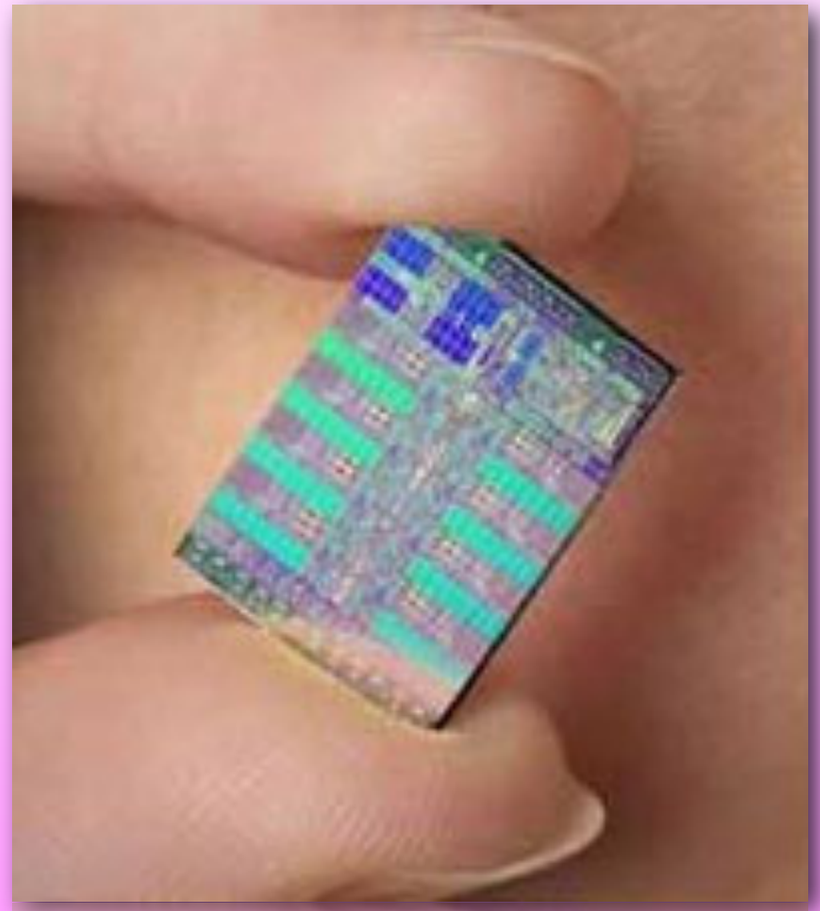


Intel Dual-core Conroe. Src Wikipedia

Terminologie

- Autre exemple de processeur Multicoeurs
  - Processeur Cell d'IBM
    - Possède 8 coeurs
    - 1 coeur sclaire
    - 7 coeurs vectoriels

Sony Play Station 3 - Utilise un Cell (Src Wikipedia)



IBM Processeur Cell
100 GFLOPS
(Src wikipedia)

Terminologie

- GPU : Graphic Processor Unit
  - Processeurs vectoriels
  - Système multicoeurs

NVidia GeForce GTX 280
933 GFLOPS
(Src wikipedia)

ATI FirePRO V9800
2.7 TFLOPS
(Src wikipedia)

Terminologie

- Cartes dédiées au calcul vectoriel
  - Cartes Tesla de NVidia

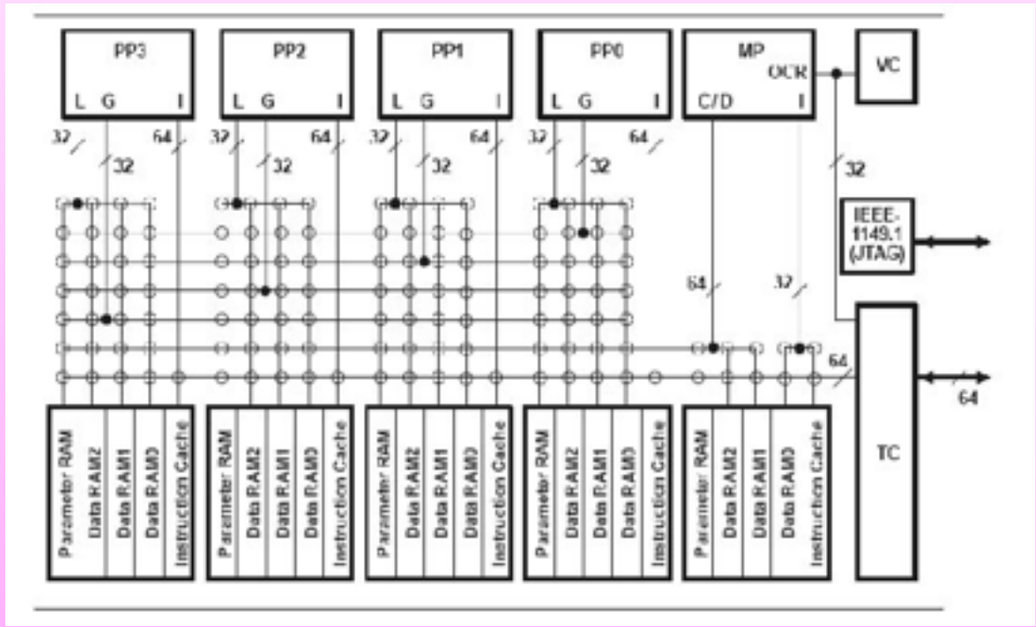
(Src wikipedia)

- Tesla série C2070
- 515 GFLOPS
- 1.6 GHz / core
- 448 cores
- Taille vecteurs = 1024
- 6 GB de mémoire

Tianhe-1A : super-ordinateur chinois (n°1)
- 2.5 PFLOPS
- Utilise des Tesla série S (2.5 TFLOPS)

Histoire

- Processeurs scalaires multicoeurs
  - 1995 pour TI avec le TMS320C80 - 4 coeurs



Src http://www.extremetech.com/

  - 1999 pour Sun avec le MAJC5200 - 2 coeurs
  - 2005 pour Intel avec le Core Duo - 2 coeurs
  - 2005 pour AMD avec l'Athlon 64 X2 - 2 coeurs

Histoire

- Processeurs scalaires multicoeurs (suite)
  - 2010, Intel Core i7-980X Extreme Edition (6 cores)
  - 2010, AMD Turion Champlain (2 cores)

Core i7 (Src wikipedia)

AMD Turion (Src wikipedia)

Histoire

- Processeurs vectoriels
  - 1960, Solomon Project (Westinghouse), plusieurs processeurs arithmétiques gérés par un processeur.
  - 1976, Cray 1
  - ...
  - 2010, Tianhe-1A

Histoire

- GPU
  - 1984, IBM Professional Graphics Controller
  - 1987, Blitter de l'Amiga (copy bloc mem)
  - 1987, IBM8514 (copy + affichage)
  - 1994, ATI Rage 1 (2D+3D)
  - 1995, NV1 de NVidia (2D+3D+Son)

Amiga 500
(Src wikipedia)

Histoire

- GPU
  - ...
  - 2010 NVidia GeForce GTX 280
  - 2010 ATI FirePRO V9800

Histoire

- Convergence / Mélange de plusieurs technologies
  - Processeurs Scalaires
  - Multicoeurs
  - Processeurs Vectoriel
  - Affichage Graphique

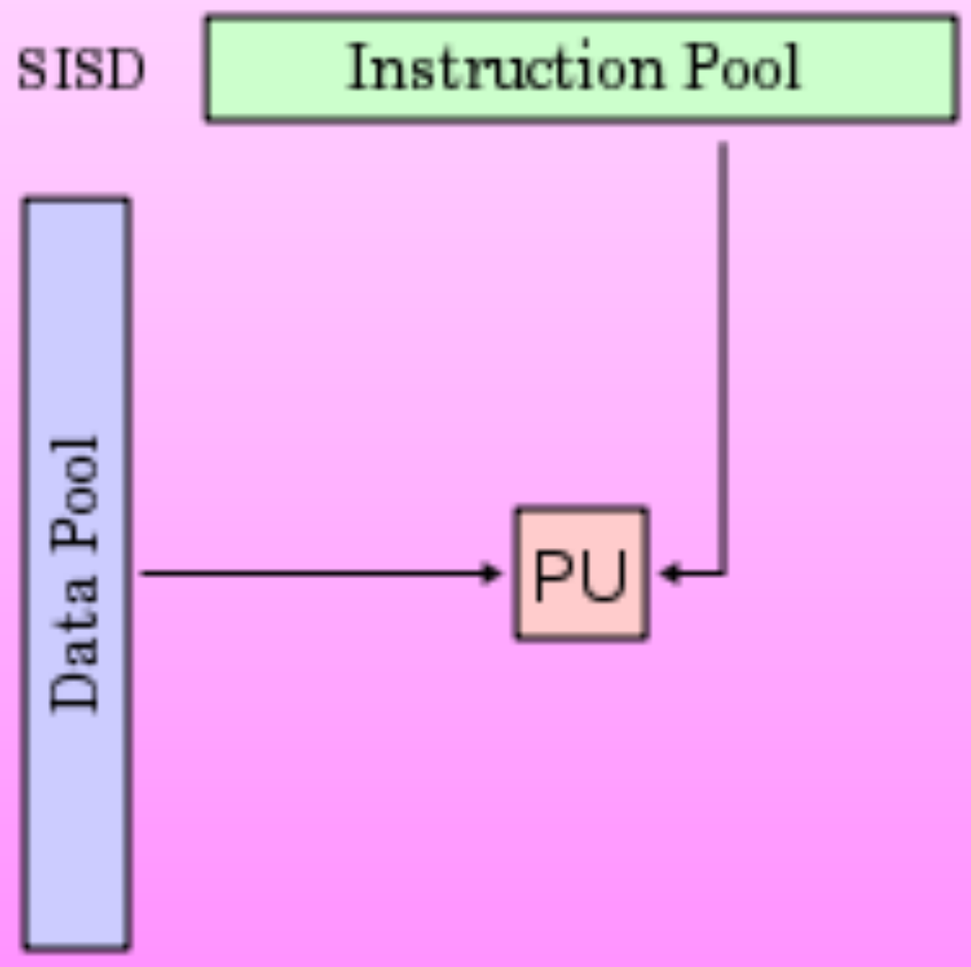
} => CPU modernes

} => GPU modernes

Programmable en OpenCL (2009)

Architecture

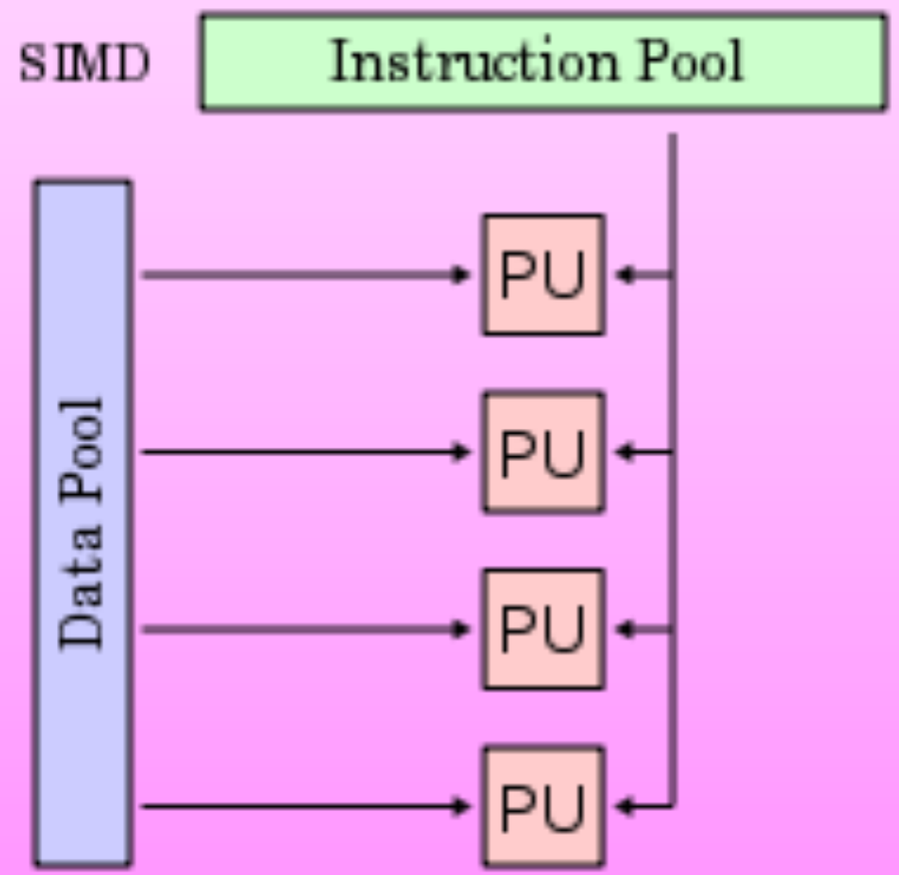
- Processeur Scalaire



(Src wikipedia)

Architecture

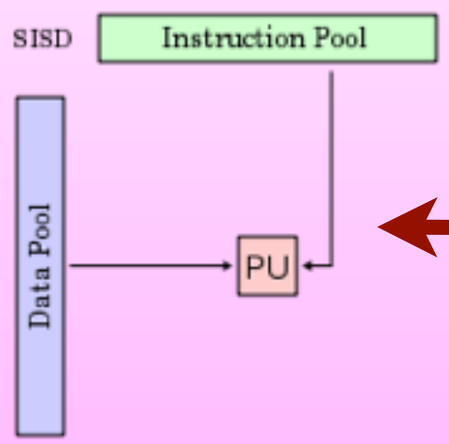
- Processeur Vectoriel



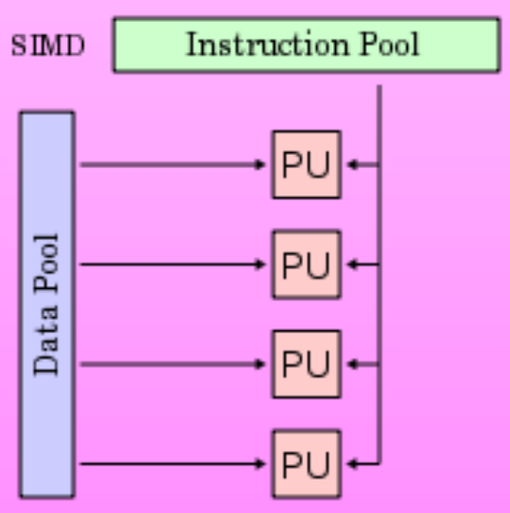
(Src wikipedia)

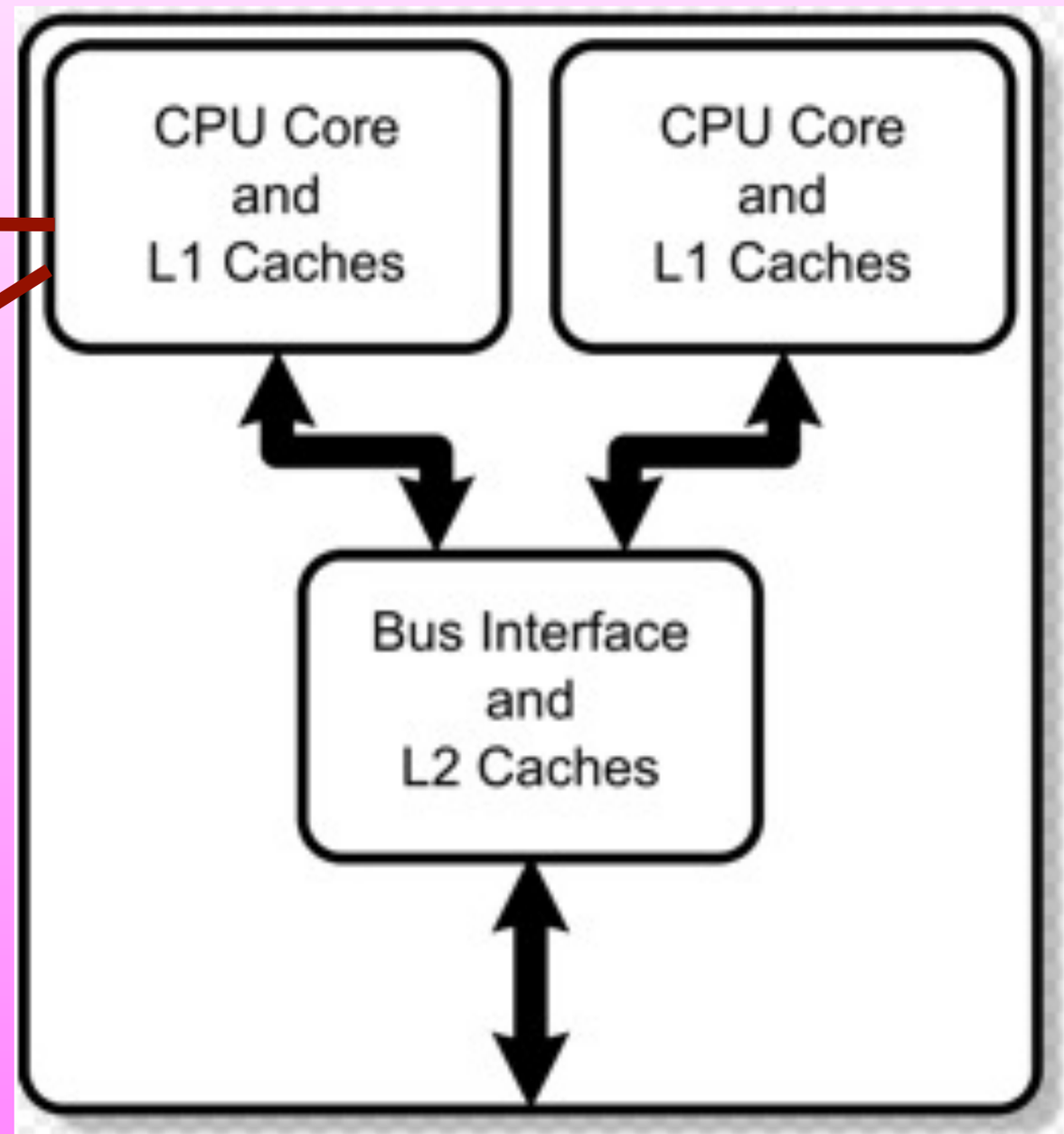
Architecture

- Processeur Multicoeurs



Peut être
Scalaire ou
Vectoriel



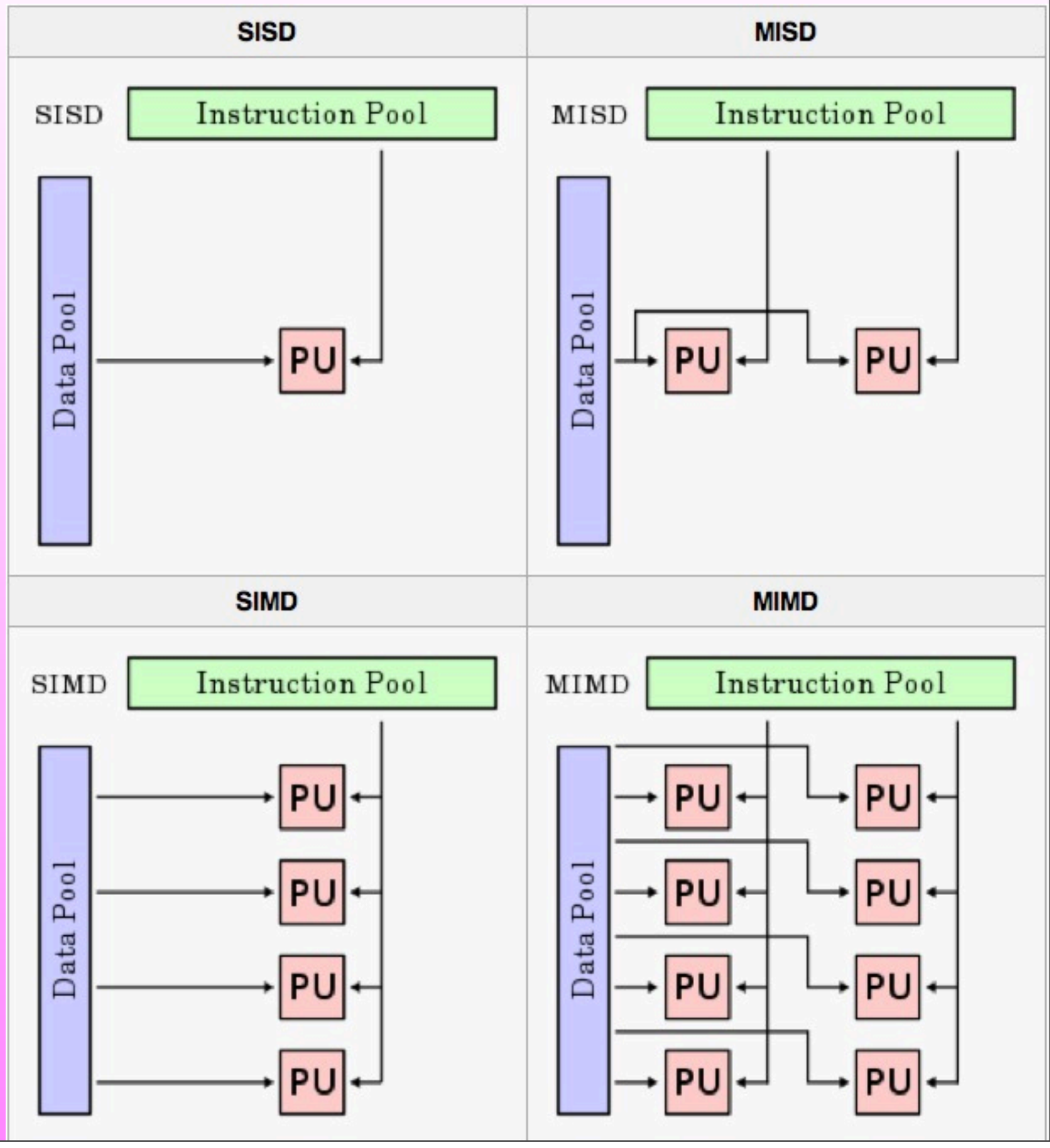


(Src wikipedia)

Architecture

- Classification de Flynn (1966)

Les 4 sont utilisables en OpenCL



Notions de programmation parallèle

- Accès concurrent à des ressources partagées
  - Protection par une section critique
  - En pratique, utilisation de Sémaphores
- Barrière et Rendez-Vous
  - Synchronisation des processus entre-eux
  - Encore à l'aide de Sémaphores

Notions de programmation parallèle

- Risques
  - Famine
  - Dead Lock
- Gestion des Priorités
  - Géré par le système d'exploitation
- Exemples : Lecteur / Rédacteur, Producteur / Consommateur, les Philosophes chinois.

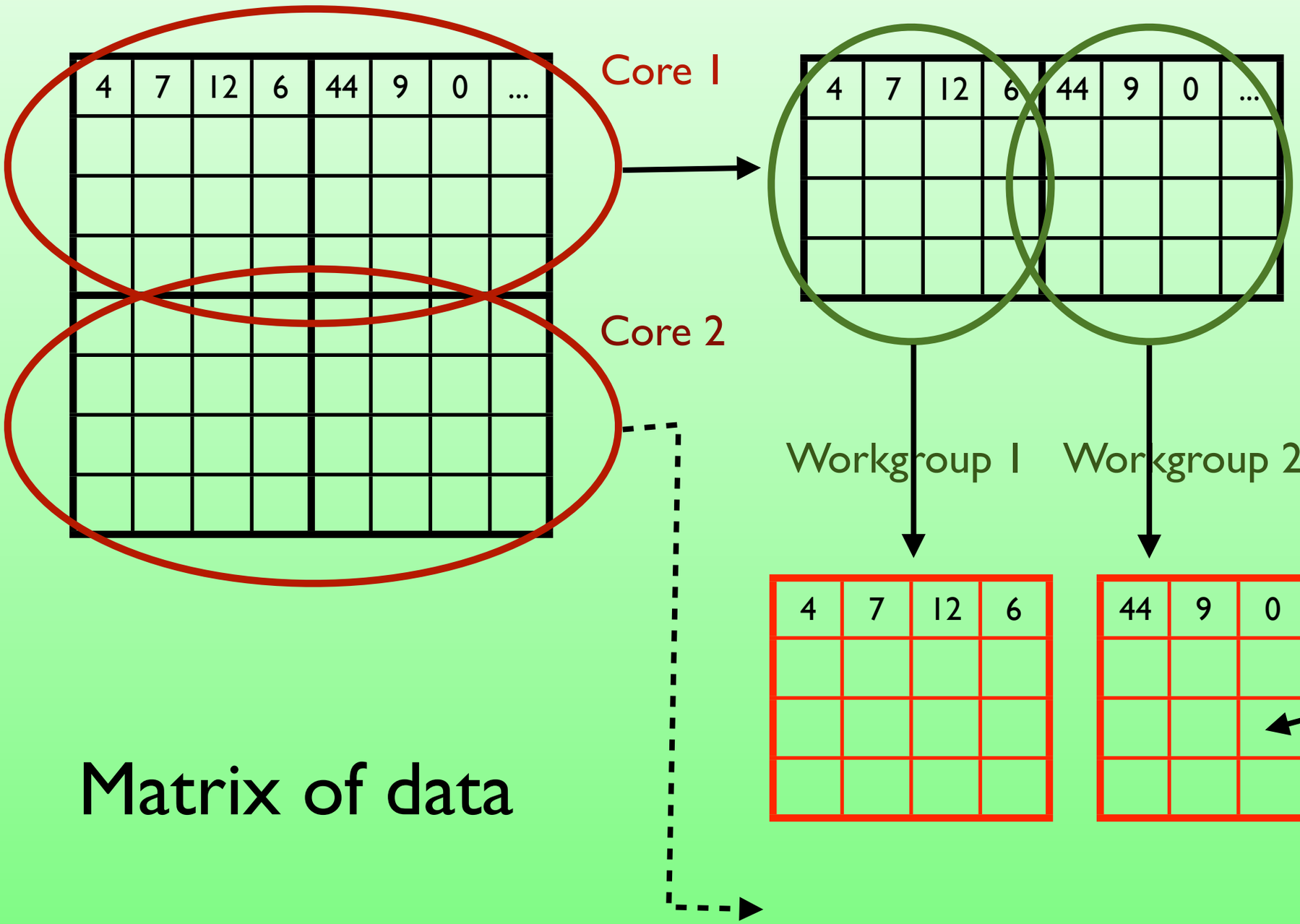
Plan de la programmation en OpenCL

- Objectives
- OpenCL
- Connecting Java and OpenCL
- Connecting OpenGL and OpenCL
- BioDynOCL
- Conclusion & Perspectives

Objectives

- Use of streaming processors inside GPU to increase computational speed
- Application to computational biology: Cellular Automata & Multiagent Systems for Cellular systems from100nm to 100µm

Streaming processors



Kernel

```
function(args) {
  treat(Matrix)
}
```

Red=//

Green=sequential

Blue=quasi //

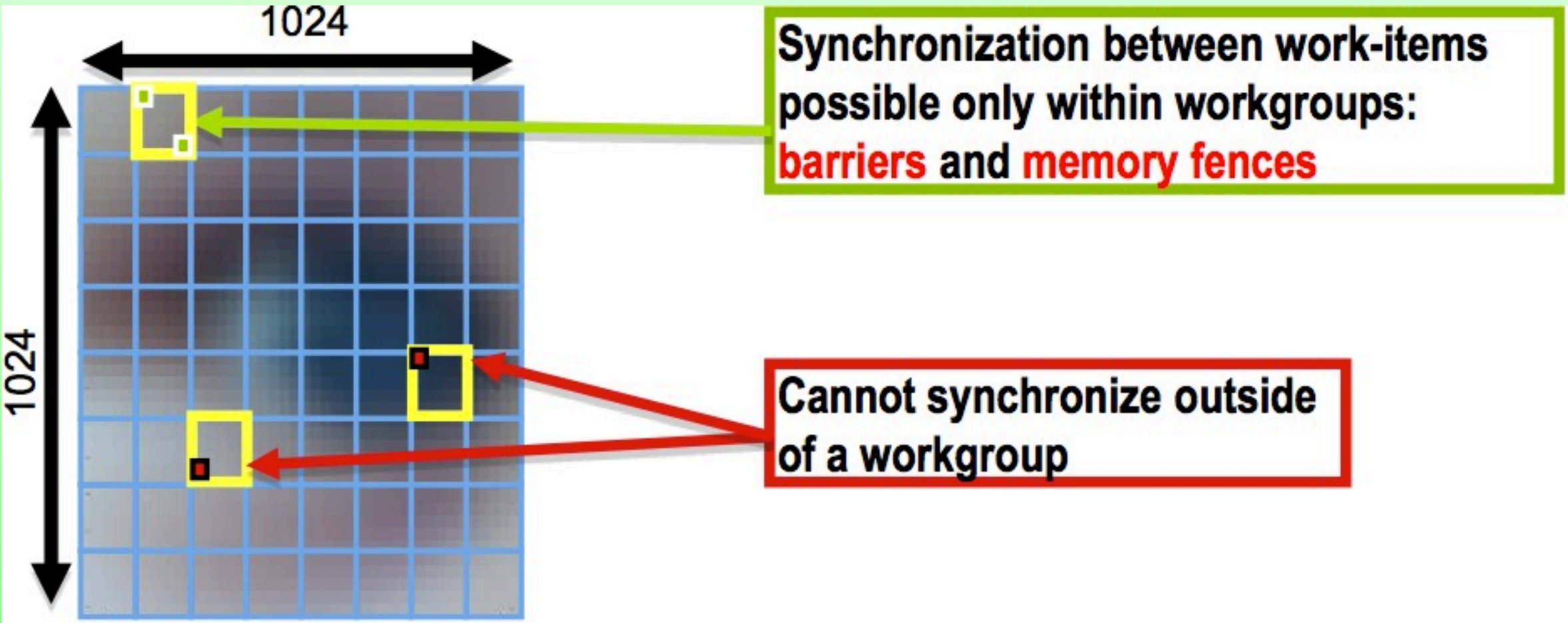
OpenCL

- Open standard (first developped by Apple, now hold by the Khronos group*)
- Version 1.0 released in may 2009
- Version 2.0 released in 2014
- Architecture + Language

*Founded in 2000 by 3DLab, ATI, Intel, NVidia, SGI, Sun to create open standard API. See http://www.khronos.org/

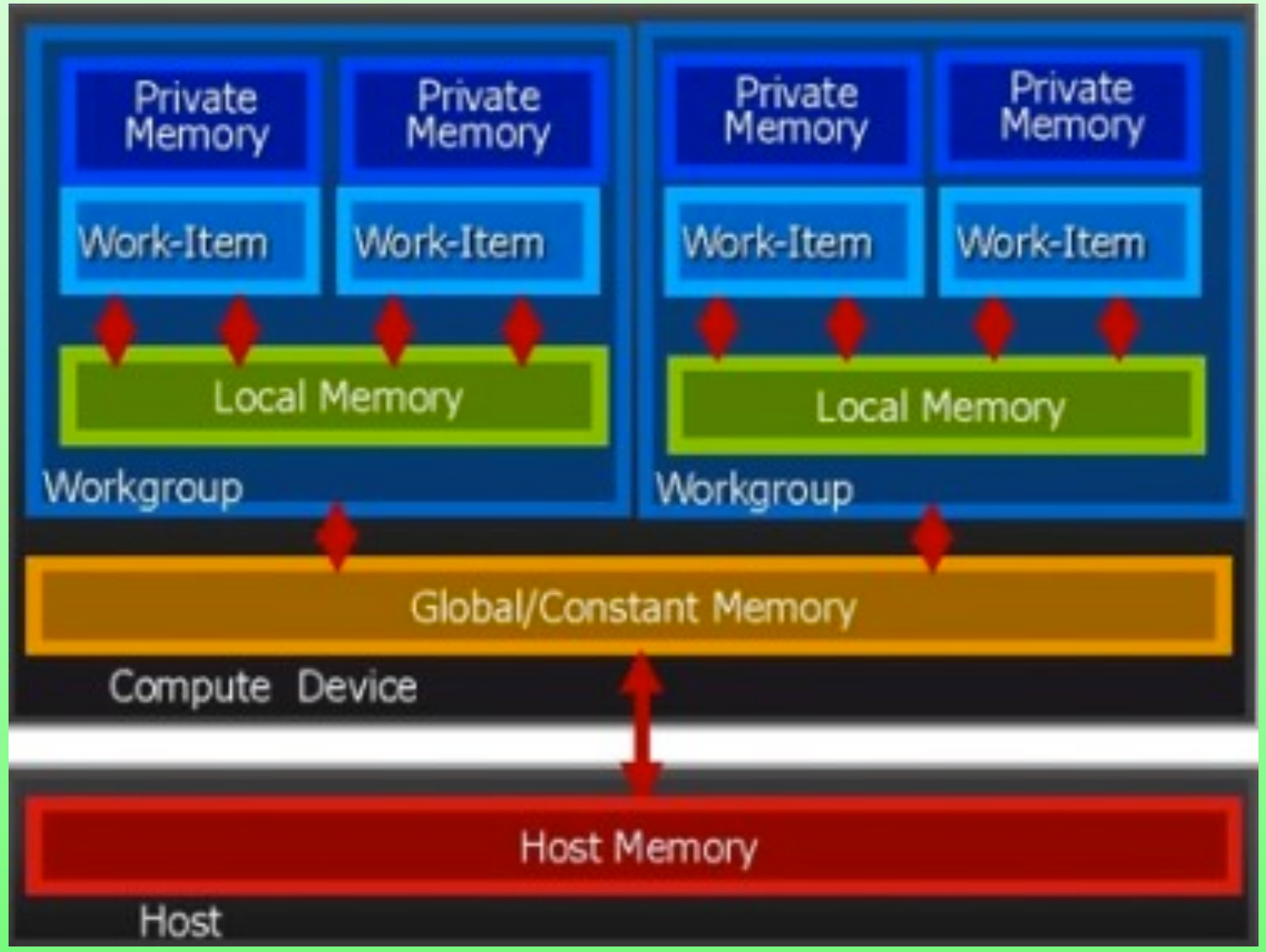
OpenCL

- Architecture



OpenCL

- Architecture



GPU / CPU

Computer

OpenCL

- Language

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];        }
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

OpenCL

- Language
  - Based on C99
    - ***without*** recursivity, function pointer nor variable length arrays
    - ***plus*** workitems, workgroups, vector types, synhronisations, adress space qualifiers (global, local, private)
  - OpenCL™ API 1.0 Quick Reference Card

Connecting Java & OCL

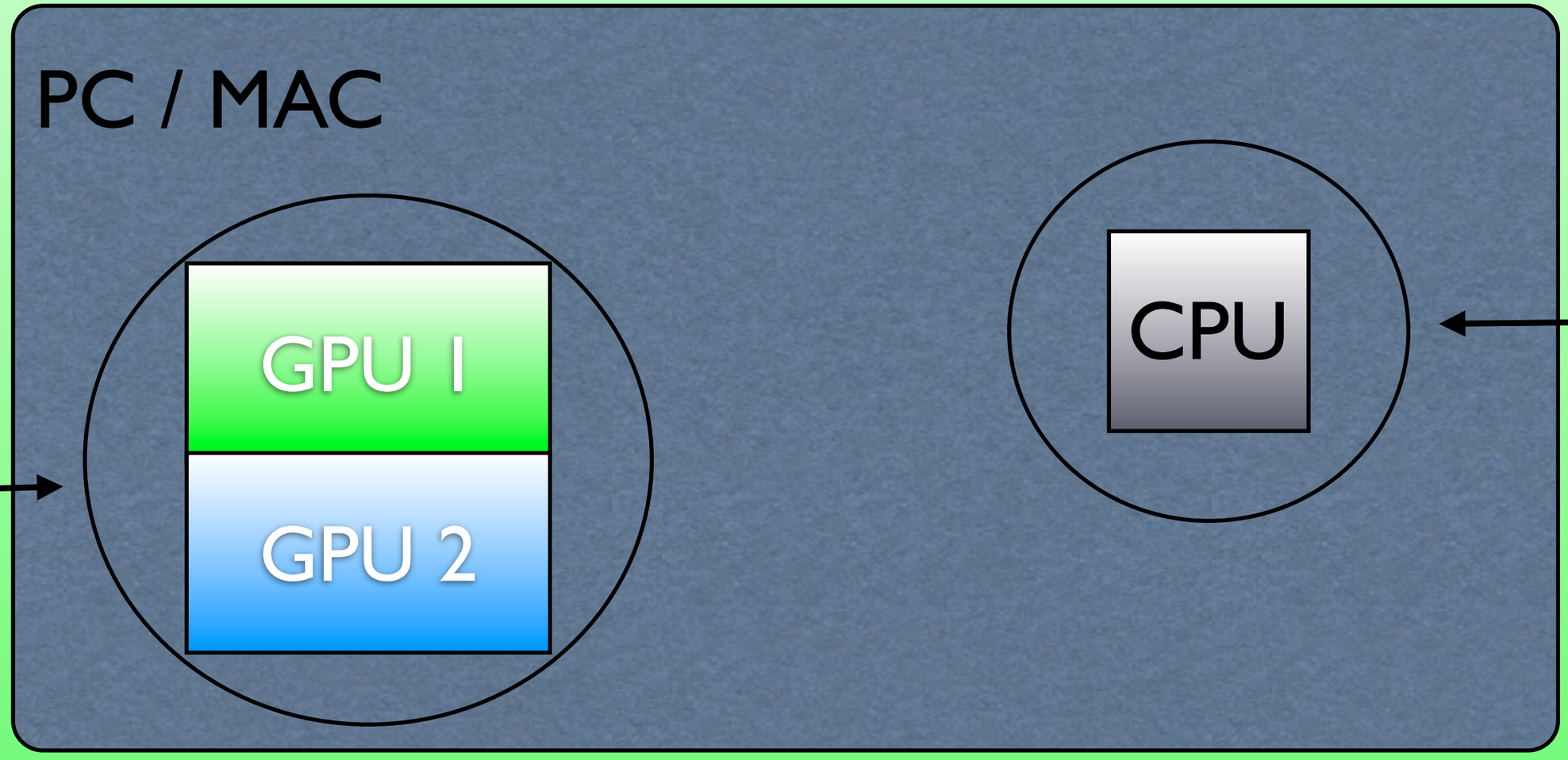
- OpenCL can be called from numerous languages: C, C++, Java, Python, SmallTalk...
- JOCL = Java bindings for OpenCL

Connecting Java & OCL

Creating a context => number of devices fct(type)

```
// Creation d'un context OpenCL sur GPU
context = clCreateContextFromType( null, CL_DEVICE_TYPE_GPU, null, null, null);
```

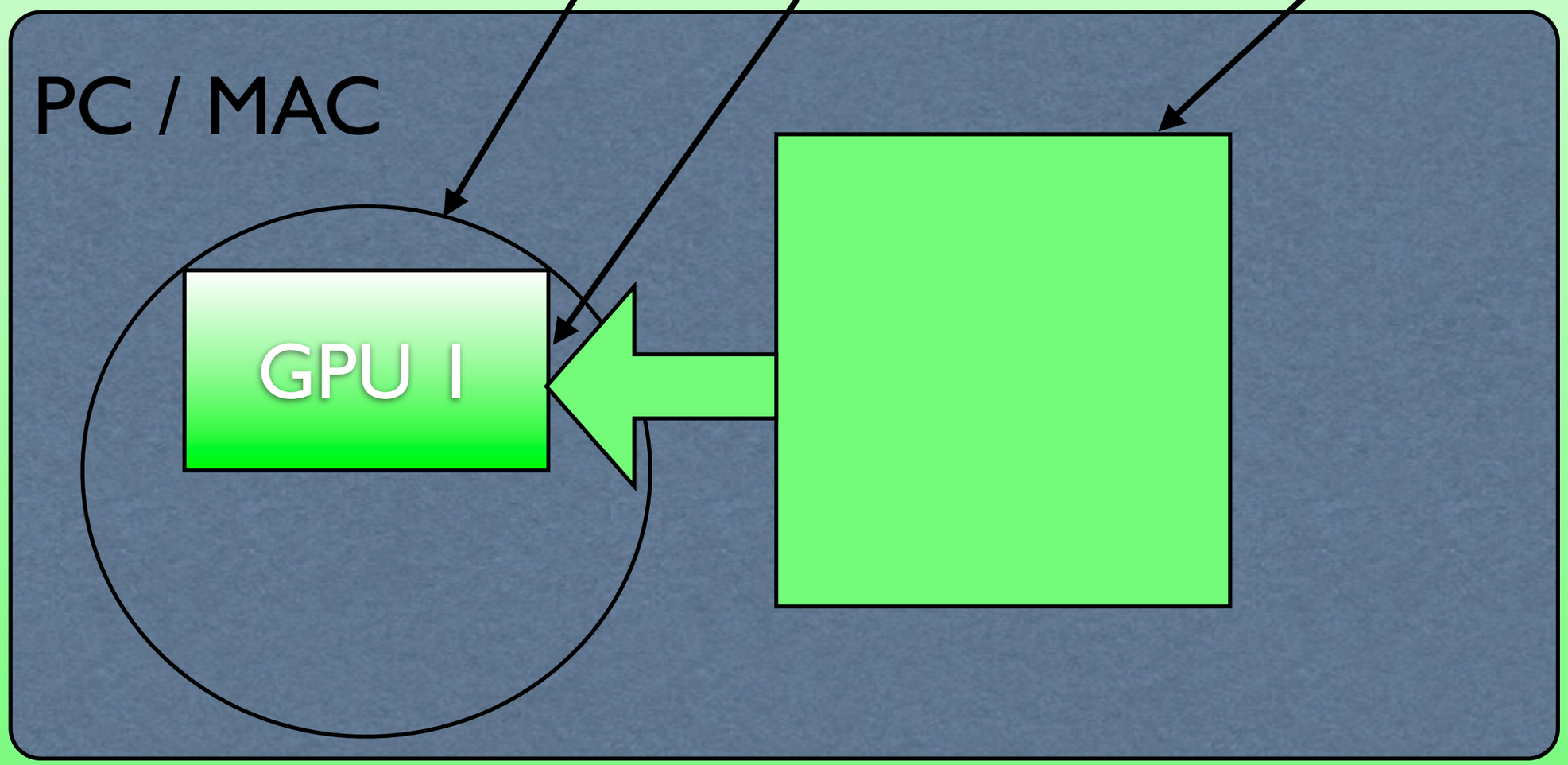
```
// Creation d'un context OpenCL sur GPU
context = clCreateContextFromType( null, CL_DEVICE_TYPE_CPU, null, null, null);
```



Connecting Java & OCL

Creating a command-queue for a specific device in a context

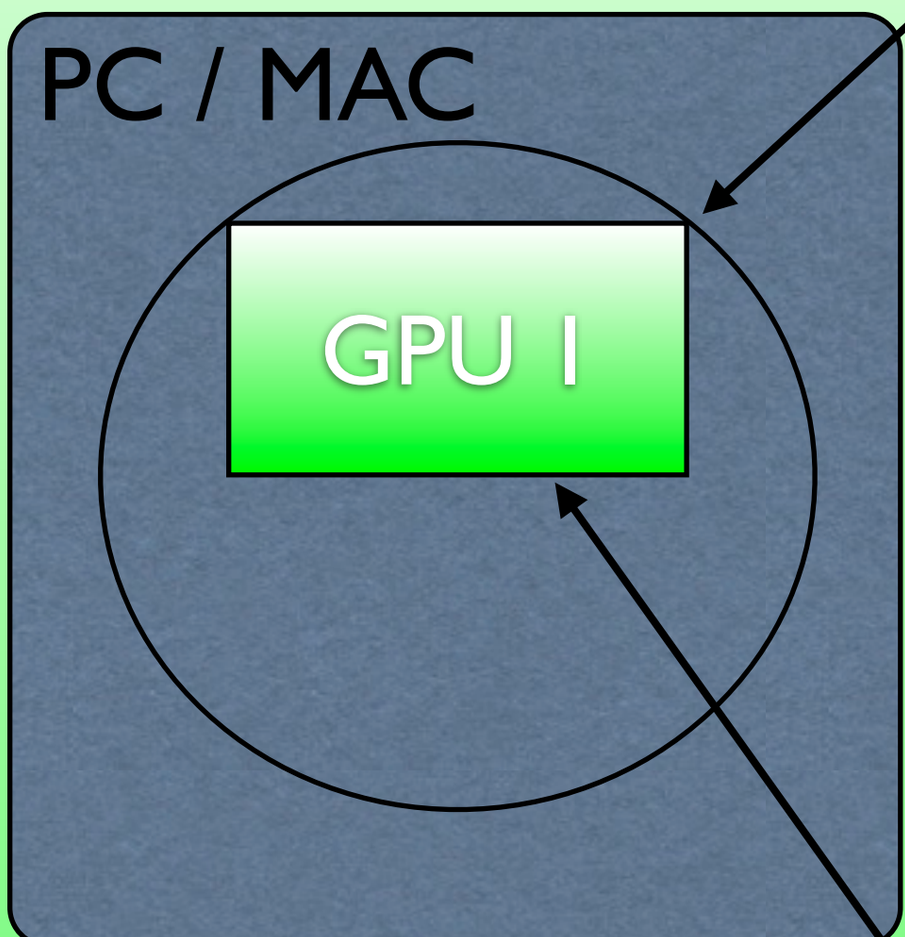
```
// Create a command-queue
commandQueue = clCreateCommandQueue(context, devices[0], 0, null);
```



Connecting Java & OCL

Creating and building an openCL program

```
program = clCreateProgramWithSource(context, 1, str_src, null, null);
```




```
uint RND_MAX = 65537; // 2^16+1
uint rnd(__global uint* Seeds)
{
  uint p = get_global_id(0);
  uint r = Seeds[p];
  Seeds[p] = (75*Seeds[p])%RND_MAX;
  return r;
}

__kernel void BioDynKernel(__global uint dx, __global uint dy
    ,__global uint t
    ,__global uint mouseX, __global uint mouseY, __global uint mouseBtn
    ,__global uint *Alea)
{
  // *** Develop your program here.
  uint h = rnd(Alea);
  // ...etc
}
```

```
err = clBuildProgram(program, 0, null, null, null, null);
```

Connecting Java & OCL

Creating a kernel function: name + args

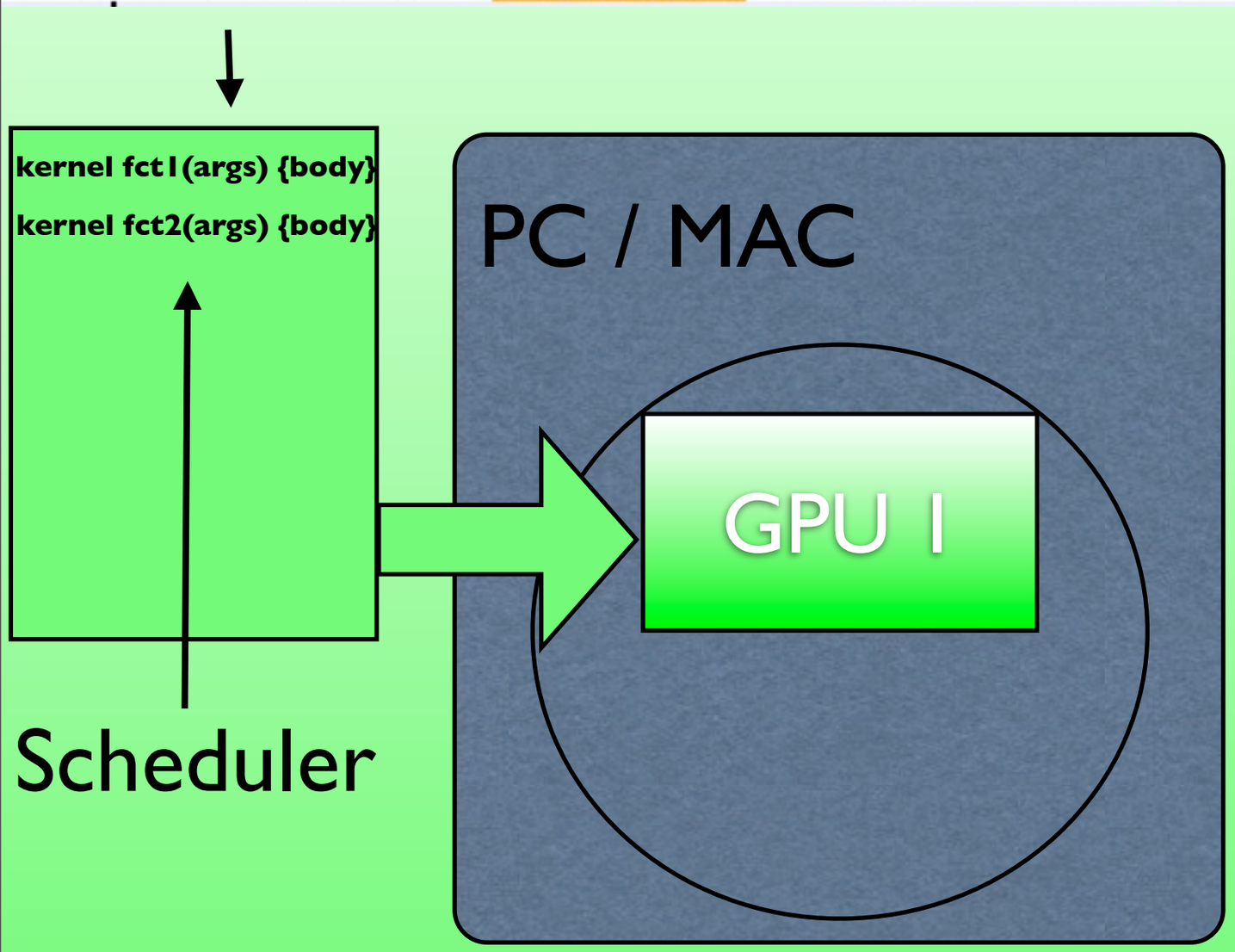
```
kernel = clCreateKernel(program, "BioDynKernel", null);
```

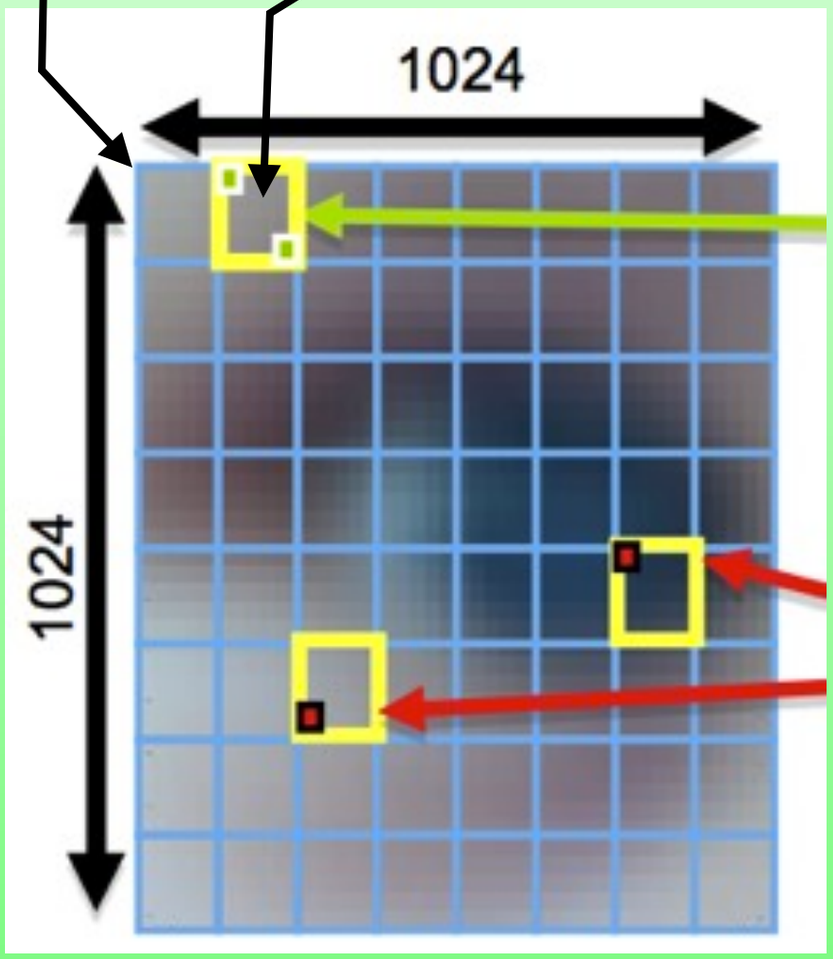
```
clSetKernelArg(kernel, 0, Sizeof.cl_int, Pointer.to(new int[]{0}));
```

Connecting Java & OCL

Adding kernel functions to the command-queue

```
clEnqueueNDRangeKernel(commandQueue, kernel, 1, null, global_work_size, local_work_size, 0, null, null);
```



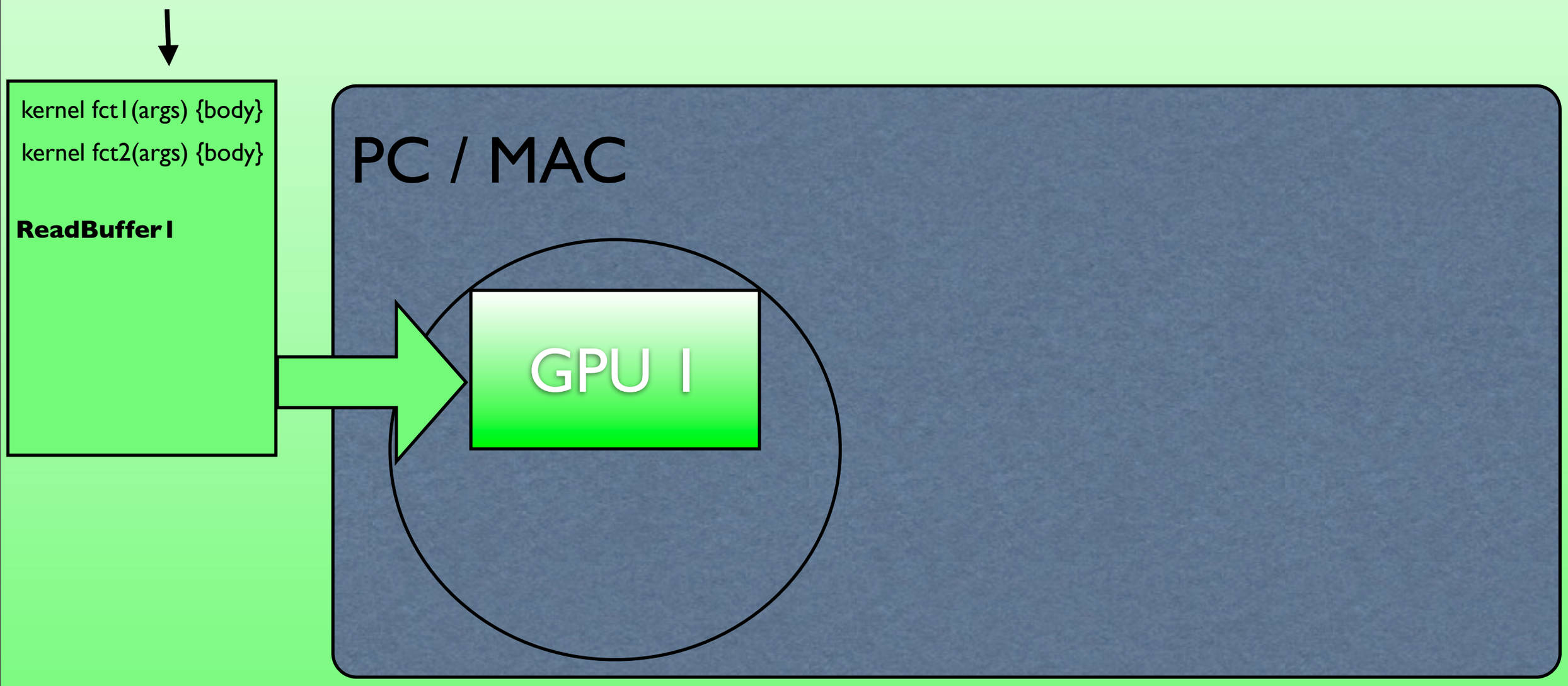


=> openCL execution

Connecting Java & OCL

Enqueue read buffers to get results from openCL to Java

```
clEnqueueReadBuffer(commandQueue, memObjects[numCalques[i]], CL_TRUE, 0, n * Sizeof.cl_int, pt_calqu
```



Connecting Java & OCL

Releasing OpenCL resources

```
clReleaseMemObject(memObjects[c]);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(commandQueue);
clReleaseContext(context);
```

Matrix Studio

Environnement de développement pour simplifier la programmation en OpenCL

Pascal Ballet
pascal.ballet@univ-brest.fr

Université de Bretagne Occidentale
Laboratoire d'Informatique des Systèmes Complexes (EA3883)

Plan

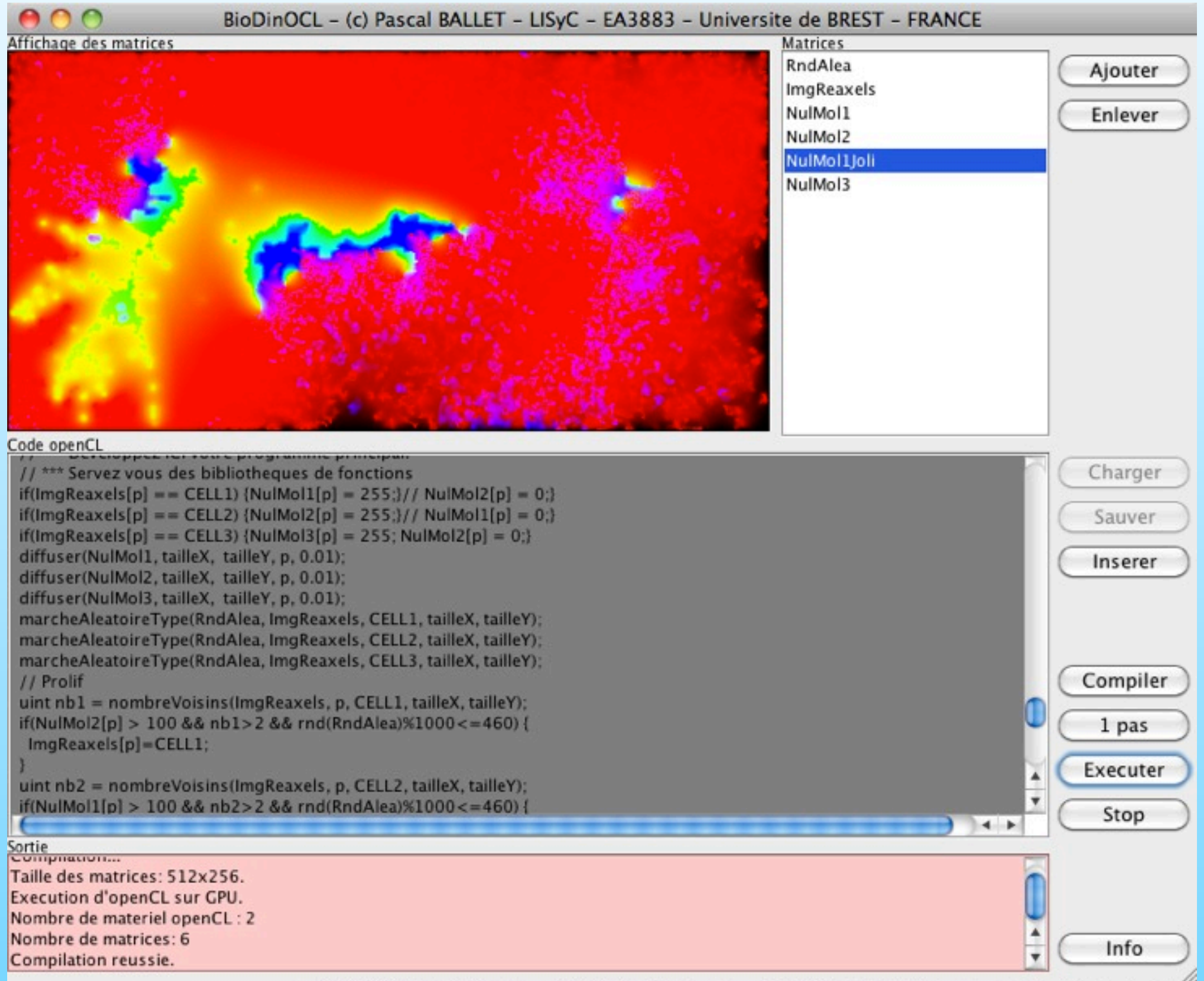
- Objectifs et Historique
- Interface
  - Matrices, Schedule, Kernels, Librairies
  - Compilation et Exécution
  - Pas à pas et Débuggage
  - Export
- Sites web, Distribution & Licence
- Classes & Librairies du code java
- Exemples
- Conclusion & Perspectives

Matrix Studio

- Objectifs
  - Masque la complexité d'OpenCL coté hôte
  - Permet de visualiser les matrices
  - Permet de créer des ordonnancements
  - Permet de coder directement en OpenCL
    - des Kernels
    - des librairies de fonction
  - Faciliter l'export des résultats / simulations

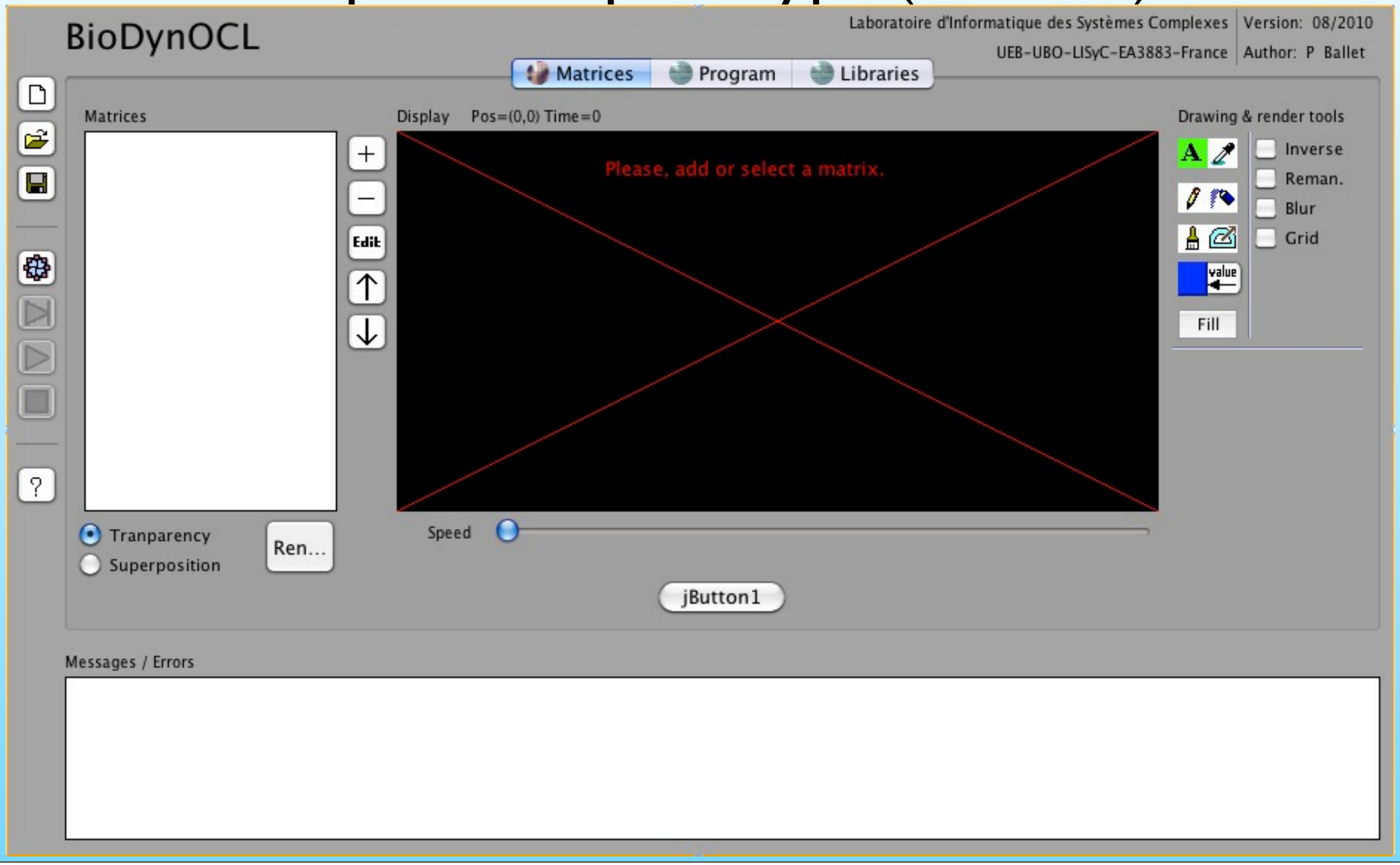
Matrix Studio

- Historique : 1er prototype (06/2010)



Matrix Studio

- Historique : 2em prototype (08/2010)



Matrix Studio

- Historique : autres contributeurs

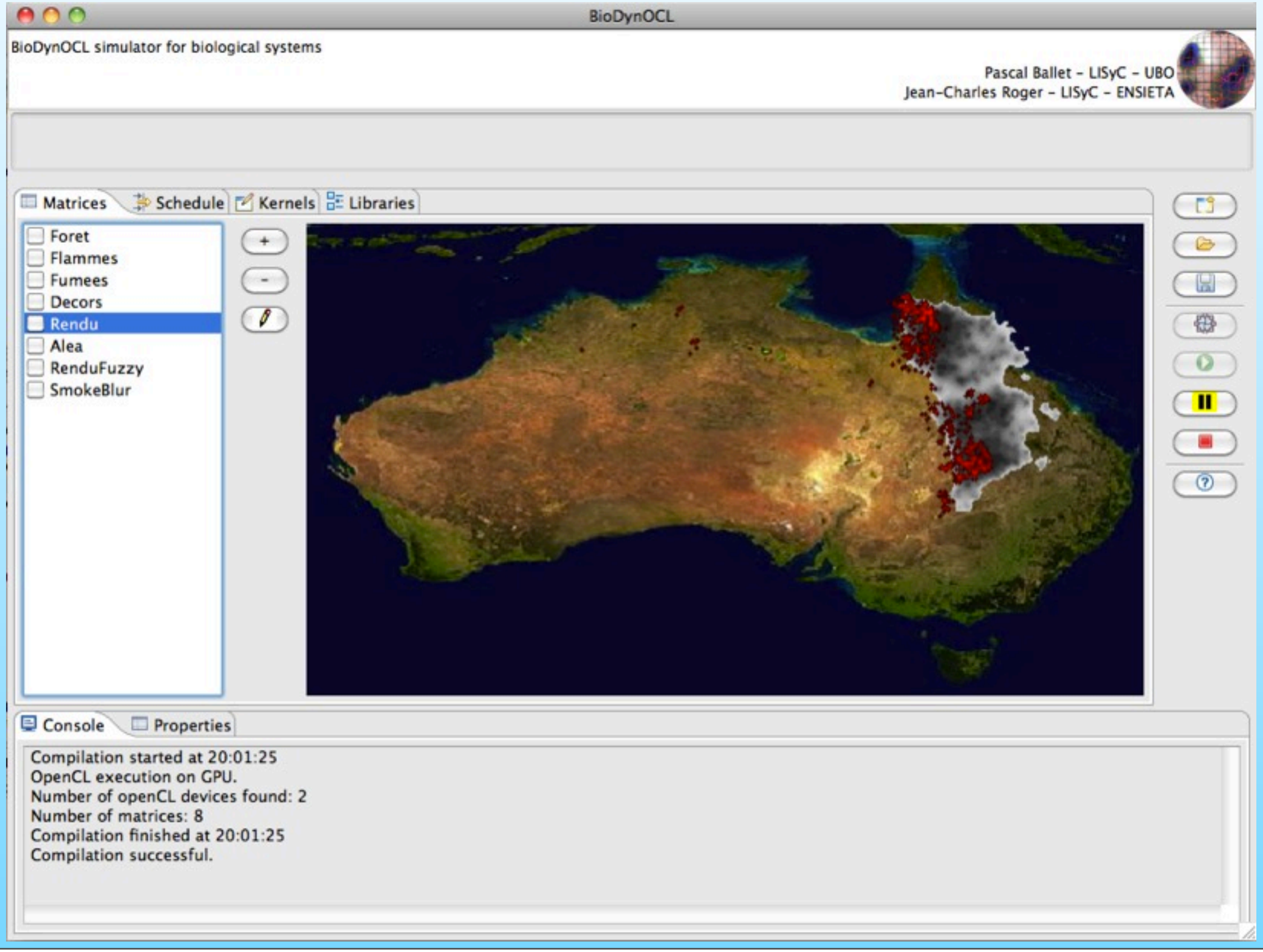
J.C Roger (LISyC - ENSTA Bretagne) :
- Architecture de classe
- Librairie Basics
- Développement

Sébastien Tripodi (LISyC - UBO) :
- Utilisateur intensif
- Détecteur de bug professionnel

Nicolas Glade (TIMC-IMAG Grenoble) :
- Générateur d'idées à haute fréquence

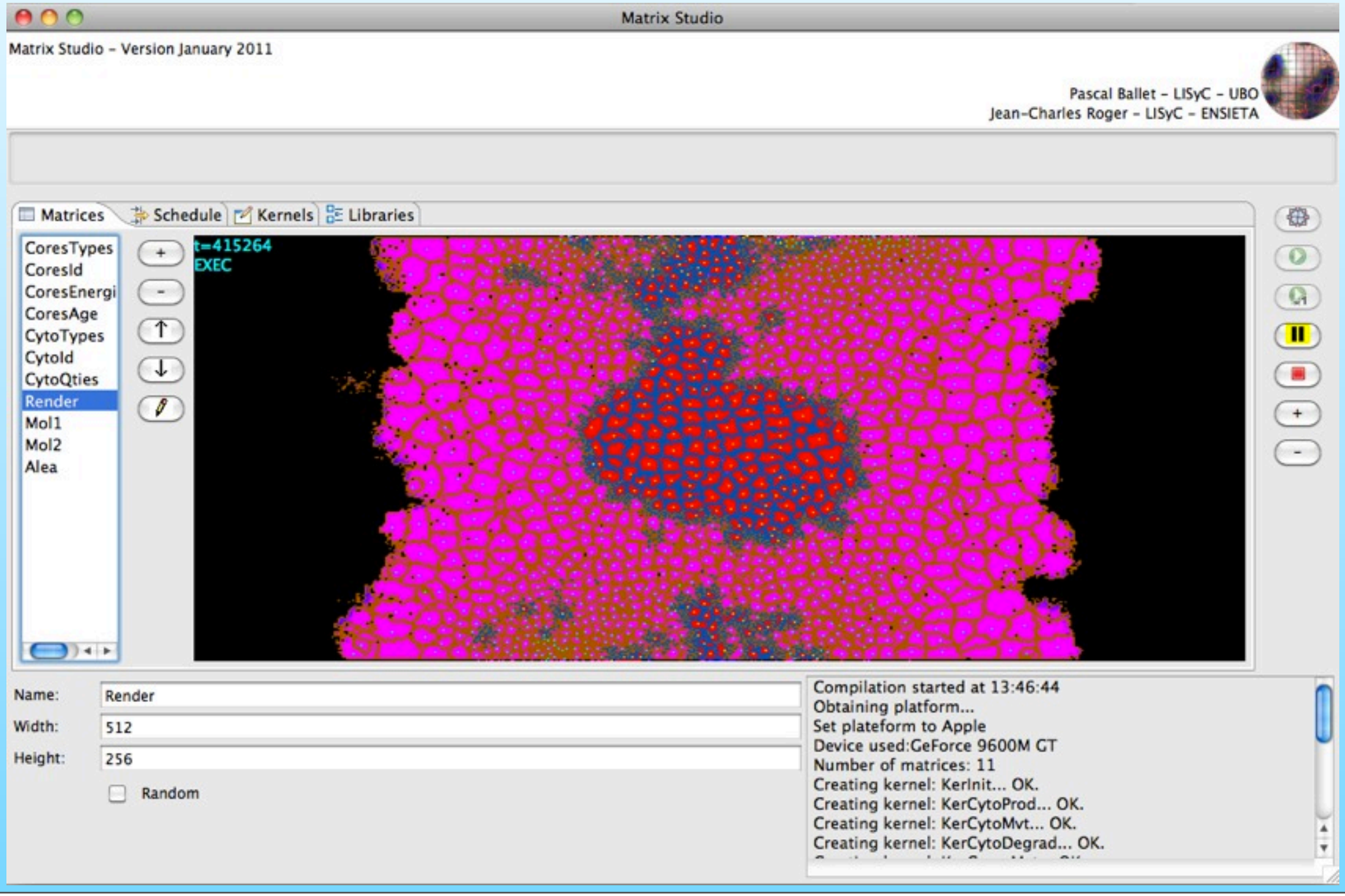
Matrix Studio

- Historique : version 1.0 (01/2011)



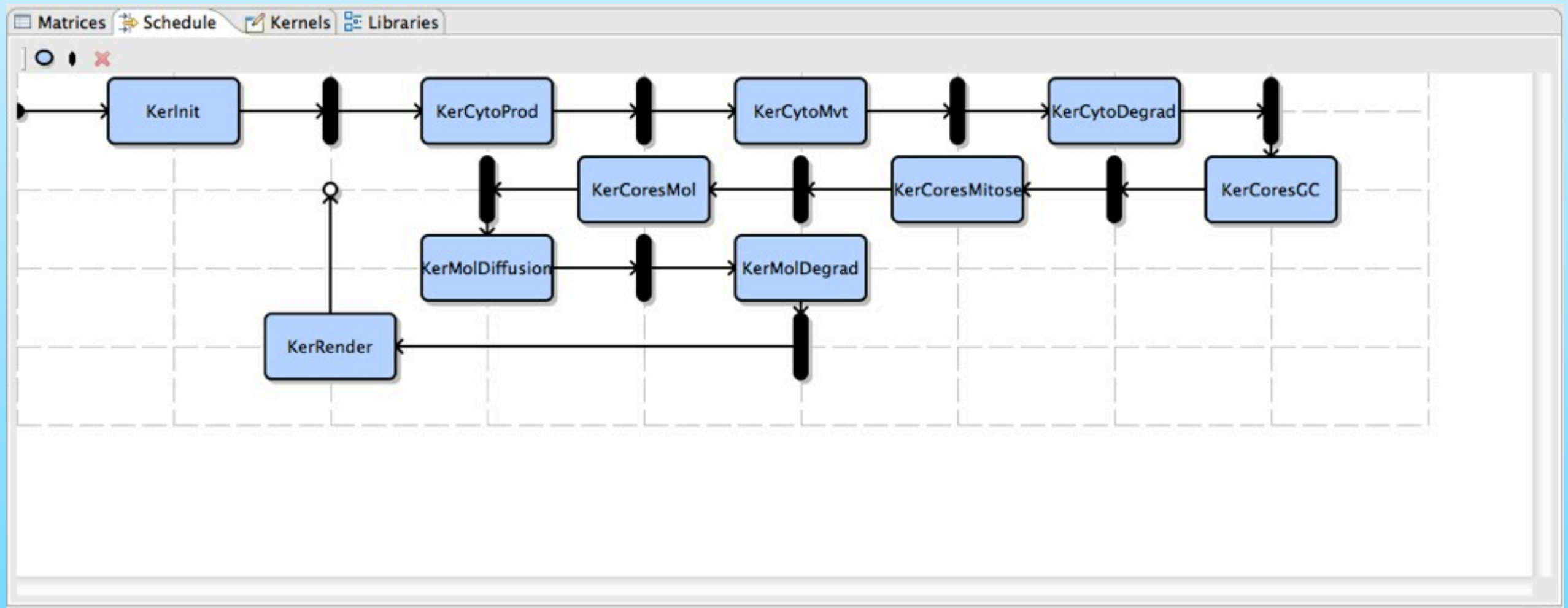
Matrix Studio

- Interface : onglet Matrices
  - Afficher / Charger / Organiser les matrices



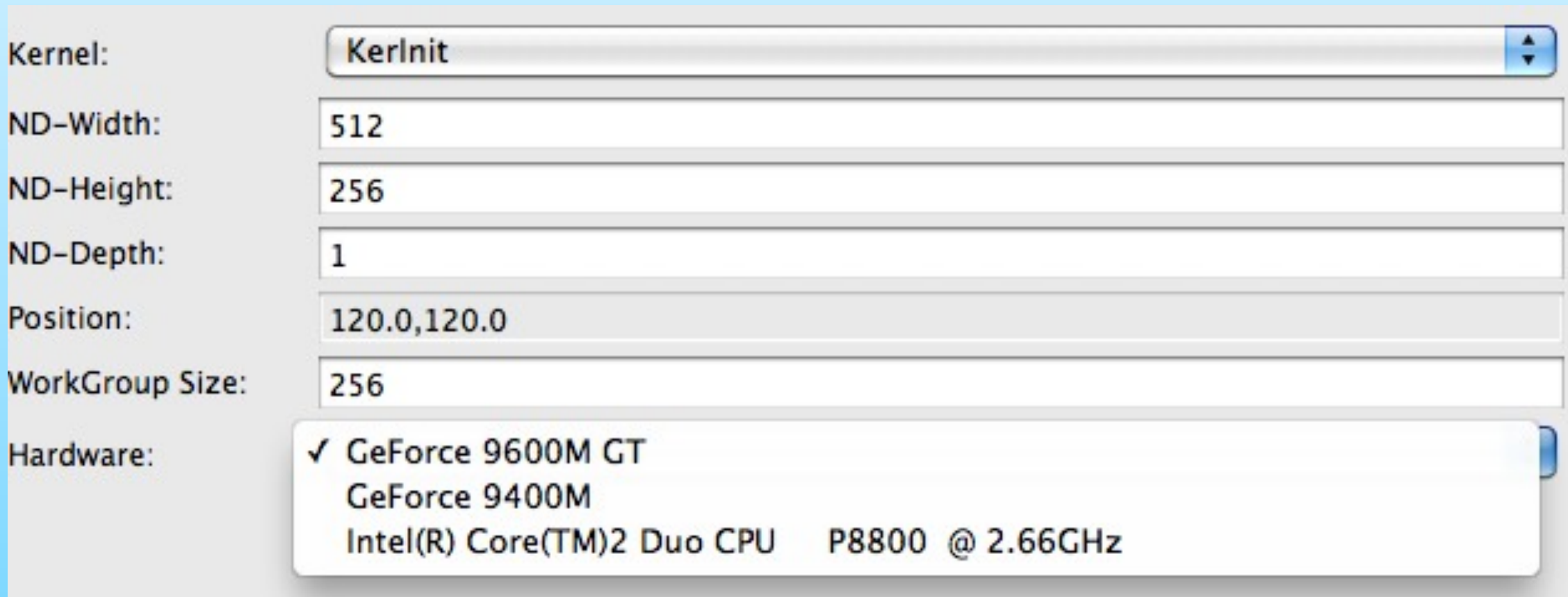
Matrix Studio

- Interface : onglet Schedule
  - Organiser l'exécution des tâches
  - Choix du découpage vectoriel
  - Choix du matériel à utiliser (CPU, GPU, etc)



Matrix Studio

- Interface : onglet Schedule (suite)
  - Configuration des tâches



Matrix Studio

- Interface : onglet Kernels
  - Ecrire du code OpenCL exécutable dans les tâches

Matrix Studio

Matrix Studio - Version 1.0 of 27th of January 2011 (OpenCL is ready)

Pascal Ballet - LISyC - UBO
Jean-Charles Roger - LISyC - ENSIETA

Matrices | Schedule | Kernels | Libraries

KerInit
KerCytoProd
KerCytoMvt
KerCytoDegrad
KerCoresMvt
KerCoresGC
KerCoresMitose
KerCoresMol
KerMolDiffusion
KerMolDegrad
KerRender

```
__global int * Render, uint Render_SX, uint Render_SY, uint Render_SZ,
__global float * Mol1, uint Mol1_SX, uint Mol1_SY, uint Mol1_SZ,
__global float * Mol2, uint Mol2_SX, uint Mol2_SY, uint Mol2_SZ,
__global int * Alea, uint Alea_SX, uint Alea_SY, uint Alea_SZ)
{
// *** Develop your code here
  uint p = get_global_id(0);
  if((rand+p)%5 != 0) return; // Not on the mask
  if(CoresId[p] == 0) return; // No core

  int x = p%CytoId_SX;
  int y = (p-x)/CytoId_SX;
  if(x <= 0 || x >= CytoId_SX-1 || y <= 0 || y >= CytoId_SY-1) return;

  int de = rnd(Alea, p)%5;
  int dp = 0;
  if(de == 0) return;
  if(de  == 1) dp = 1;
  if(de  == 2) dp = -1;
  if(de  == 3) dp = CytoId_SX;
  if(de  == 4) dp = -CytoId_SX;

  // Cannot go onto another core
  if(CoresId[p+dp] != 0) return;
```

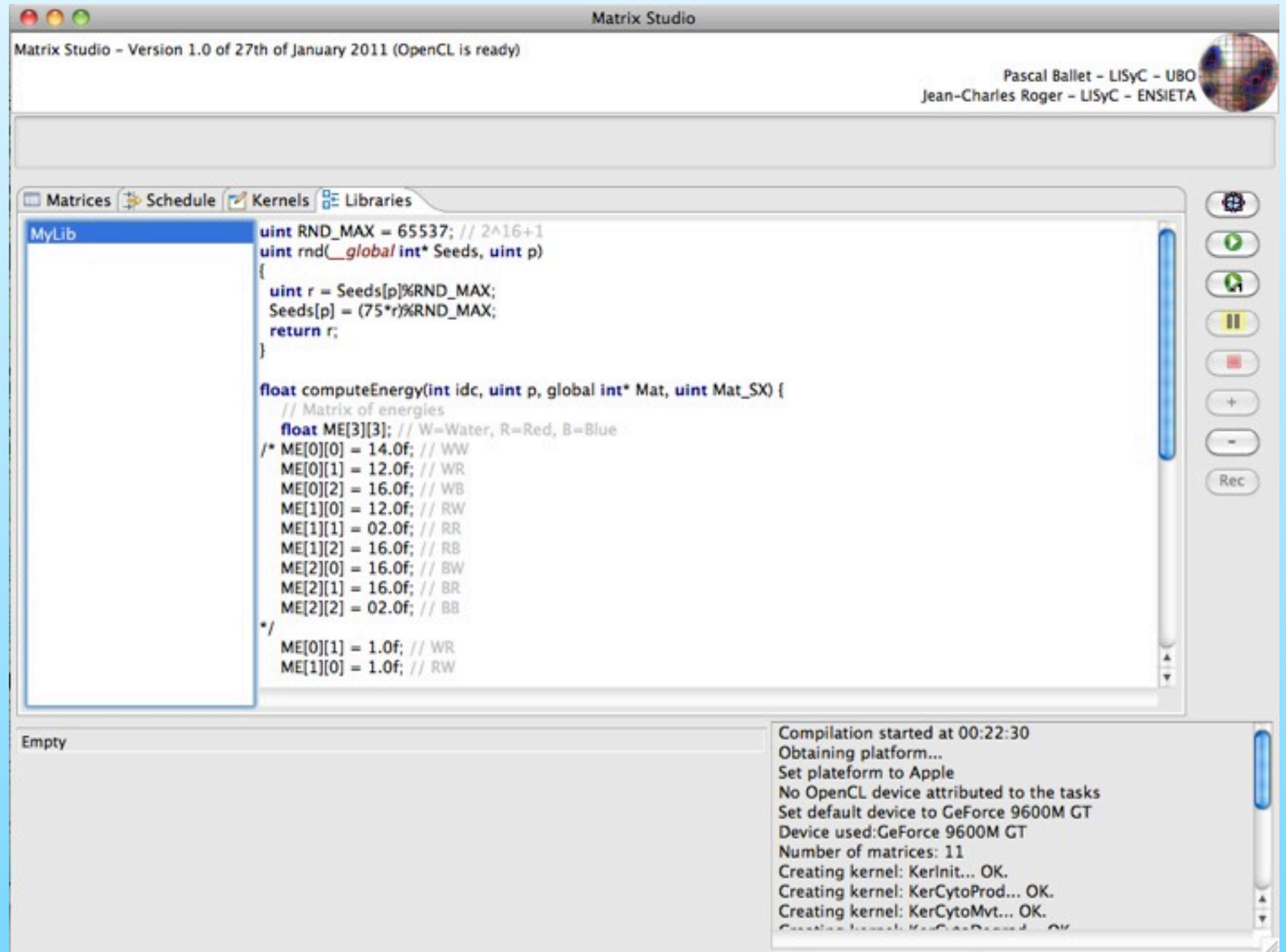
Rec

Empty

```
Compilation started at 00:22:30
Obtaining platform...
Set plateform to Apple
No OpenCL device attributed to the tasks
Set default device to GeForce 9600M GT
Device used:GeForce 9600M GT
Number of matrices: 11
Creating kernel: KerInit... OK.
Creating kernel: KerCytoProd... OK.
Creating kernel: KerCytoMvt... OK.
```

Matrix Studio

- Interface : onglet Librairies
  - Ecrire du code OpenCL exécutable dans les kernels



Matrix Studio

- Interface : compilation et exécution

Compiler

Exécuter

Exécuter 1 pas seulement

Mettre en pause l'exécution

Stopper l'exécution et revenir à l'état initial

Augmenter le taux de rafraichissement

Diminuer le taux de rafraichissement

Matrix Studio

- Interface : compilation et exécution (suite)

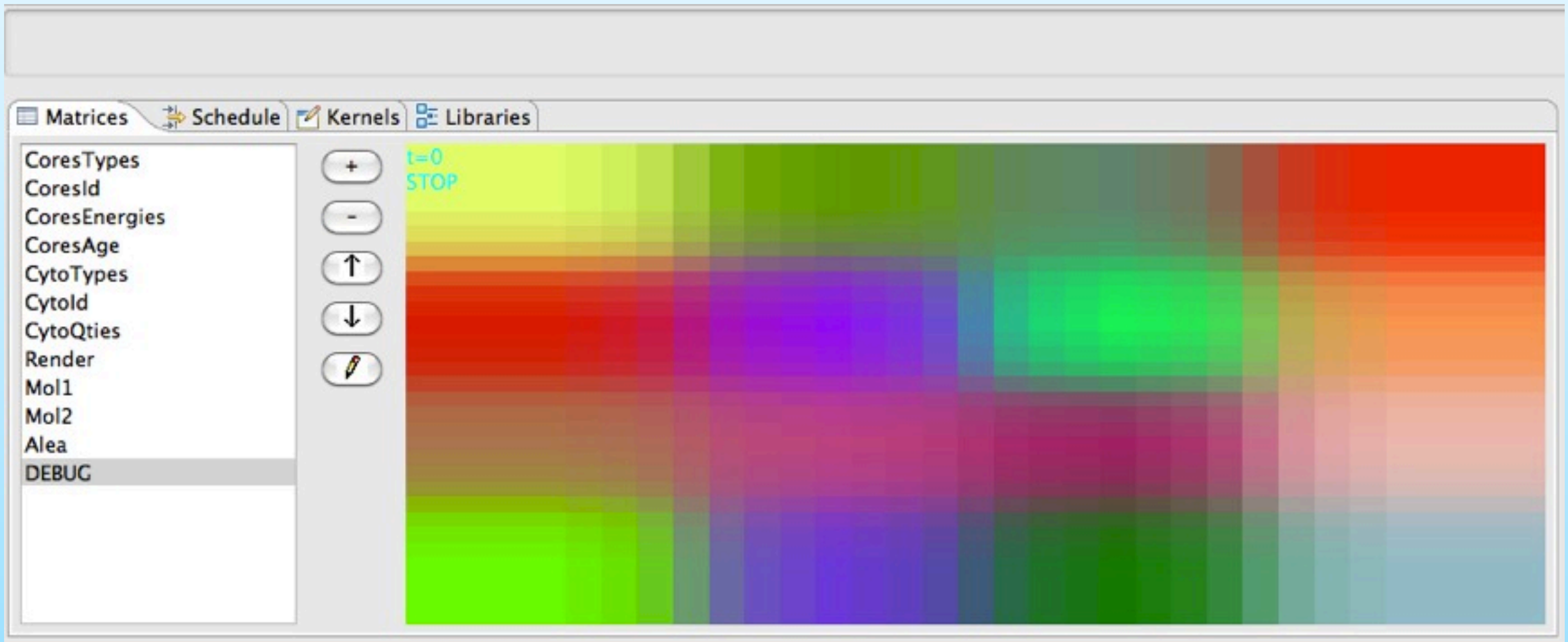
```
Compilation started at 00:22:30
Obtaining platform...
Set plateform to Apple
No OpenCL device attributed to the tasks
Set default device to GeForce 9600M GT
Device used:GeForce 9600M GT
Number of matrices: 11
Creating kernel: KerInit... OK.
Creating kernel: KerCytoProd... OK.
Creating kernel: KerCytoMvt... OK.
Creating kernel: KerCytoDegrad... OK.
Creating kernel: KerCoresMvt... OK.
Creating kernel: KerCoresGC... OK.
Creating kernel: KerCoresMitose... OK.
Creating kernel: KerCoresMol... OK.
Creating kernel: KerMolDiffusion... OK.
Creating kernel: KerMolDegrad... OK.
Creating kernel: KerRender... OK.
Compilation finished at 00:22:32
Compilation successful.
```

Visualisation de la compilation :
- Erreurs éventuelles
- Warning
- Récapitulatif de la configuration

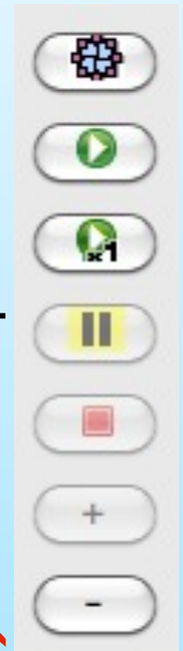
Matrix Studio

- Interface : pas à pas et débuggage

1) Créer une matrice DEBUG



3) Exécuter 1 pas seulement



2) Ecrire le code OpenCL pour placer les valeurs souhaitées dans la matrice DEBUG

```
void PlacerInfoDEBUG(uint p, global int* MatDEBUG, uint MatDEBUG_SX) {
    if(p == 0) {
        Mat[0] = 10;
        Mat[1] = 16;
    }
}
```

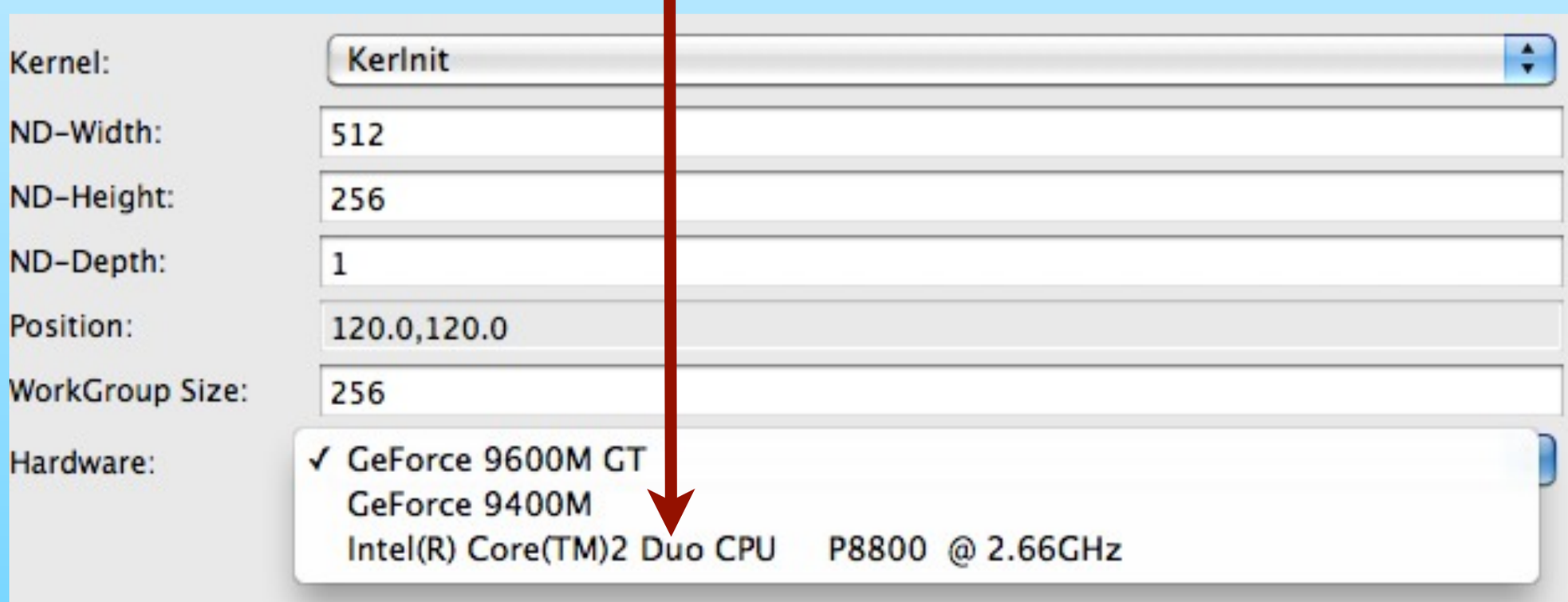
4) Lire les données dans la matrice DEBUG à l'aide de la souris

Matrix Studio

- Interface : pas à pas et débuggage (suite)

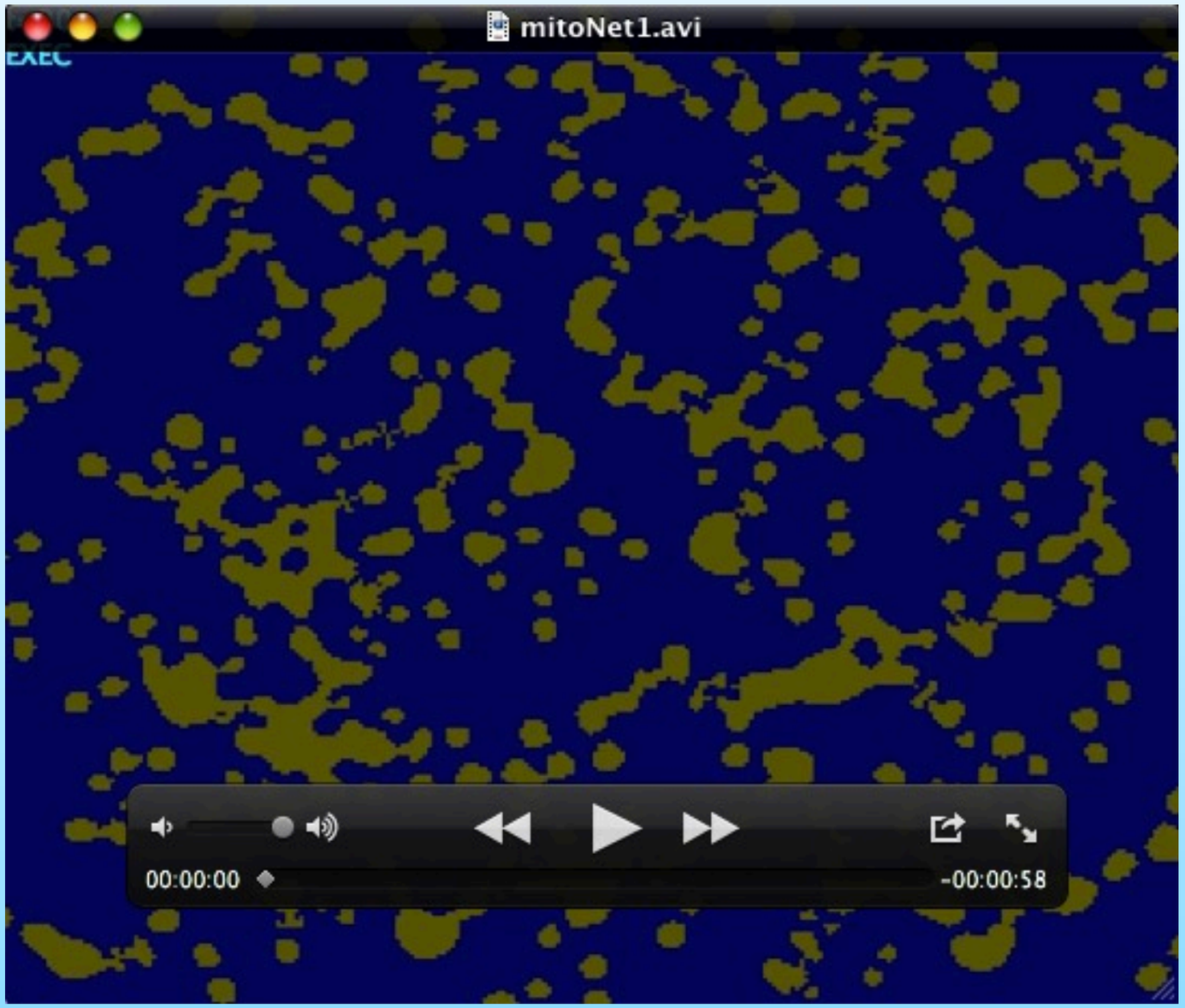
  Ou bien, choisir le CPU comme matériel pour l'exécution et utiliser le pas à pas.

  <u>Attention</u> : le choix du matériel (CPU, GPU...) est possible MAIS le pas à pas dans le code n'est pas encore développé (pour cela en mode CPU, gdb semble possible).



Matrix Studio

- Interface : export

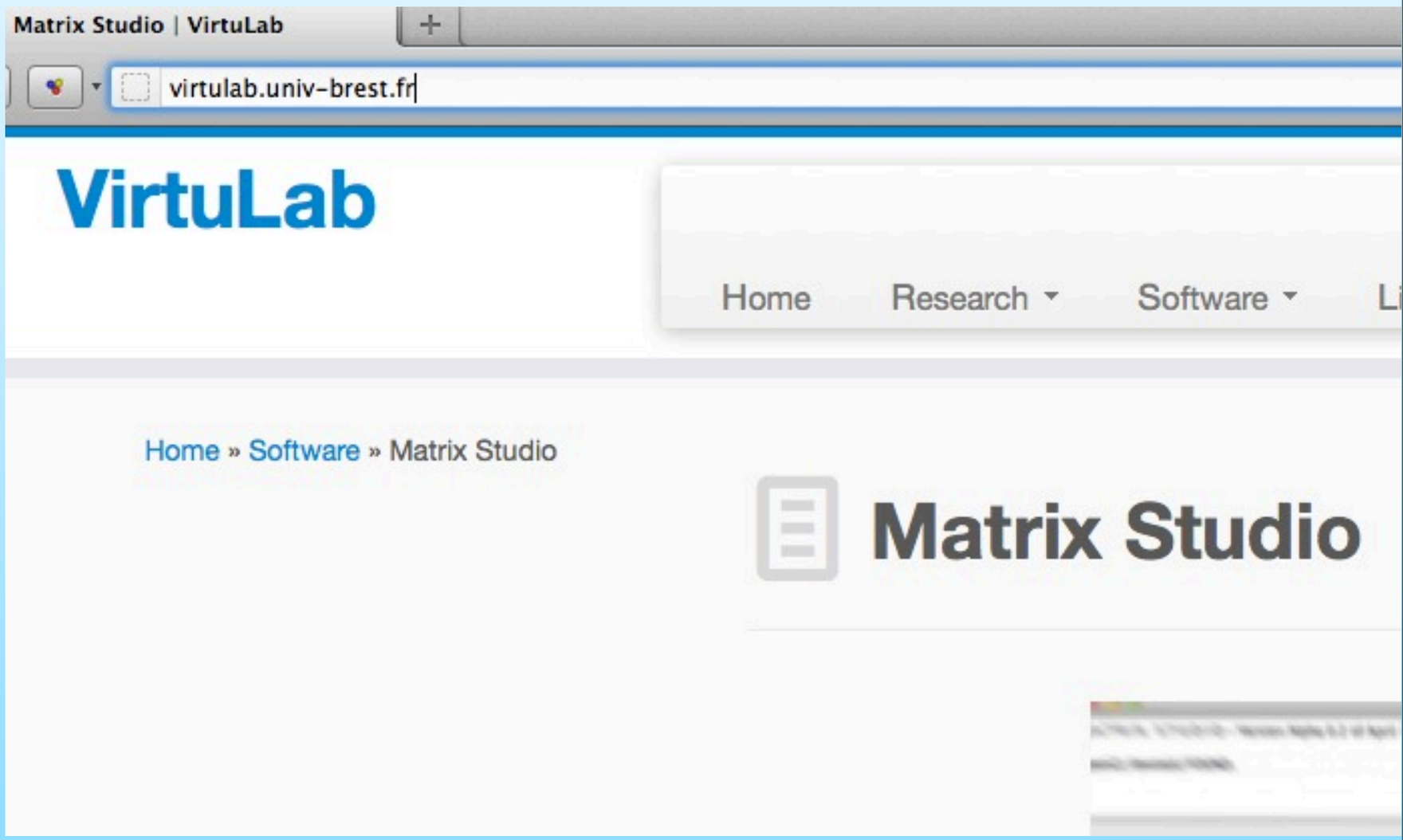


Enregistrement de la matrice courante au format avi (mjpeg).

Matrix Studio

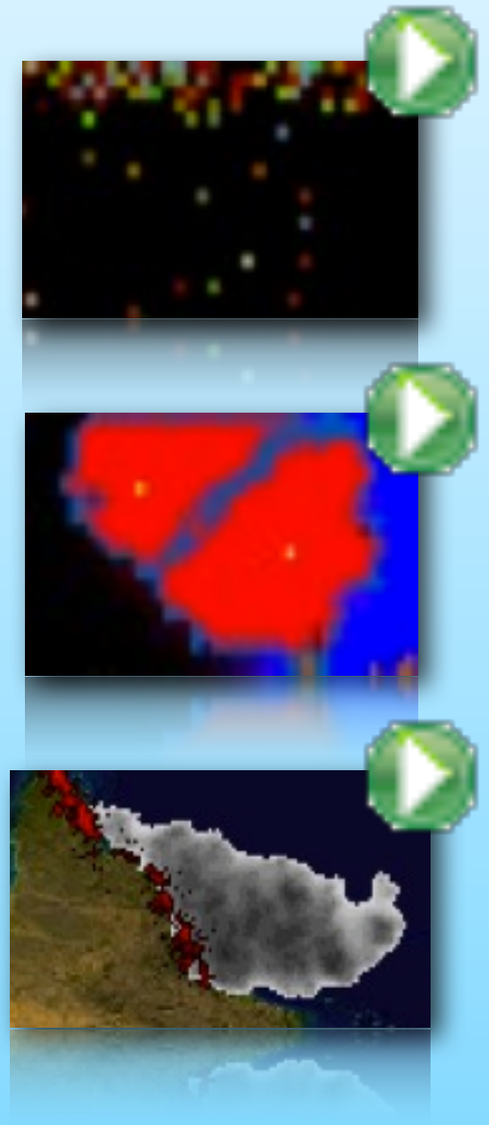
- Site web http://virtulab.univ-brest.fr

  - Infos
  - Tutoriaux
  - Binaires



Matrix Studio

- Exemples
  - GEcoBioDyn
  - MC-Cell
  - Australie en feu



Matrix Studio

- Conclusion +
  - Simplification réelle
  - Pratique à l'usage

- Conclusion -
  - Debuggage difficile
  - Plantage radical

Conclusion & perspectives

- Approche intéressante car
  - utilise le hardware avec efficacité
  - souple (SIMD, MISD, SISD, et MIMD)
- Mais complexe à mettre en oeuvre
  - Coté hôte (utiliser OpenCL avec C++, Java...)
  - Programmation parallèle difficile (car inhabituelle)
  - Débuggage de qualité impossible pour le moment