Introduction au calcul parallèle avec OpenCL

Julien Dehos

Séminaire du 05/01/2012

Sommaire

- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

Sommaire

- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

Introduction

- Problématique :
  - Calcul haute performance (HPC)
  - Tendance actuelle : calculs en parallèle
  - Différentes architectures matérielles
  - Différentes technologies logicielles
  - Pas de consensus, évolutions

Introduction

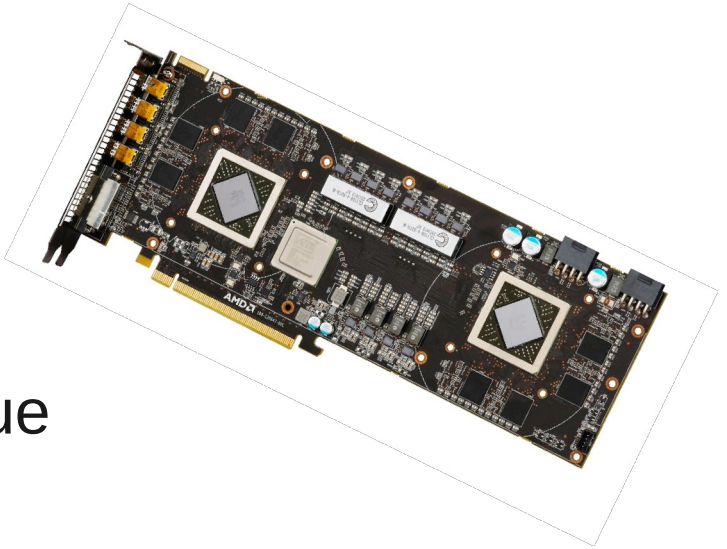
- Architectures matérielles :
  - CPU :
    - Plusieurs CPU sur la carte mère
    - Plusieurs cœurs par CPU
    - Plusieurs threads par cœur : hyperthreading
    - Plusieurs données par instructions (SIMD) : SSE, AVX

→ quelques unités de calcul

→ pour des calculs complexes

Introduction

- Architectures matérielles :
  - GPU :
    - Plusieurs cartes graphiques
    - Plusieurs GPU par carte graphique
    - Plusieurs cœurs par GPU
    - Cœurs vectoriels (ou pas)



→ nombreuses unités de calcul

→ pour des calculs simples

Introduction

- Architectures matérielles :
  - CPU + GPU :
    - Toutes sortes de combinaisons possibles
    - Mais souvent les CPU servent uniquement à contrôler les GPU

Introduction

- Architectures matérielles :
  - Comparaisons GPU/CPU :
    - Prendre en compte les temps de transfert GPU
    - Comparer avec une version CPU optimisée

"Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU", Lee et al., ISCA, 2010

Introduction

- Technologies logicielles :
  - CPU : pthread, OpenMP, SSE
  - Shaders GPU : HLSL, GLSL, Cg
  - GPGPU : Brooks, Sh, Cuda, ATI Stream
  - Génériques : MPI, HMPP, OpenCL

Sommaire

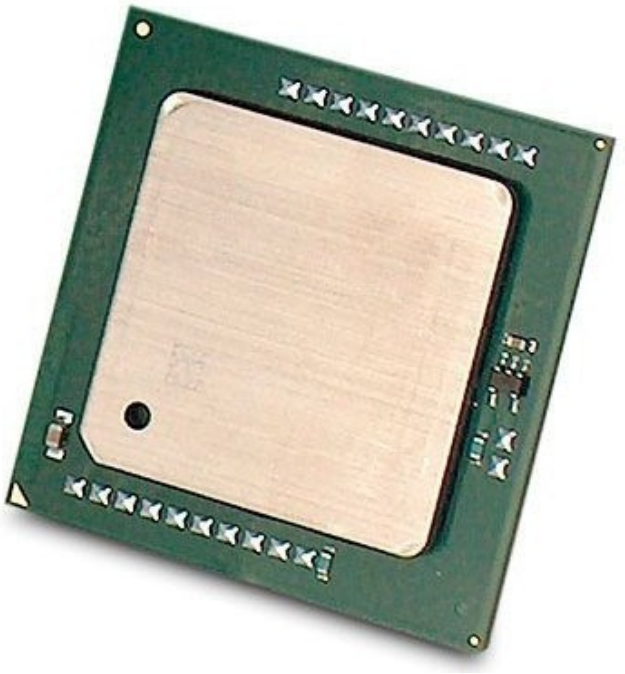
- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

Le calculateur du CGR/LISIC/LMPA

- Caractéristiques du calculateur :
  - 2 CPU Intel Xeon X5660
  - 144 Go de mémoire vive DDR3
  - 2 disques dur SATA II 500 Go 7200 trs/min
  - 8 cartes Tesla C2050

Le calculateur du CGR/LISIC/LMPA

- Caractéristiques du X5660 :
  - Architecture Gulftown
  - 6 cœurs à 2,8 GHz (soit 12 threads en HT)
  - 12 Mo de cache L3
  - SSE 4.2
  - 3 Bus DDR3-1333 → 32 Go/s



Le calculateur du CGR/LISIC/LMPA

- Caractéristiques du C2050 :
  - 1 GPU GF100 à 585 MHz
  - 448 thread processors
  - 3 Go de mémoire vive
  - Bus GDDR5 à 144 Go/s
  - MAD (SP) : 1,030 TFLOPs
  - FMA (DP) : 0,515 TFLOPs

Le calculateur du CGR/LISIC/LMPA

- Système :
  - Debian 6 "squeeze"
  - NFS (comptes réseaux)
  - 1 disque pour les données de calculs : /data
  - Cuda, pycuda, debugger, profiler, bibliothèques
  - OpenCL, pyopencl
  - NVIDIA_GPU_COMPUTING_SDK

→ c.f. la doc sur le réseau

Le calculateur du CGR/LISIC/LMPA

- Accès :
  - baru.univ-littoral.fr
  - 193.49.192.7
  - ssh depuis le réseau du CGR
  - Penser à utiliser /data

Le calculateur du CGR/LISIC/LMPA

- Avant d'exécuter sur le calculateur, il est sage de tester sur votre machine (si possible)
- Attention à la disponibilité des paquetages OpenCL

Le calculateur du CGR/LISIC/LMPA

- Installation automatisée (Archlinux + NVIDIA) :
  - Installer le driver NVIDIA
  - Installer le runtime OpenCL
  - Installer les entêtes OpenCL
  - Installer pyopencl

→ `Pacman -S nvidia opencl-nvidia opencl-headers python2-opencl`

  - NVIDIA_GPU_COMPUTING_SDK (attention cuda requiert gcc <= 4.4)

Le calculateur du CGR/LISIC/LMPA

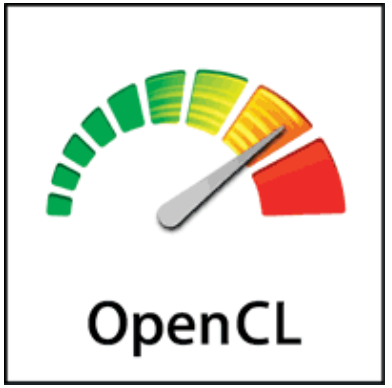
- Installation manuelle (Debian + NVIDIA) :
  - Installer les linux-headers (via apt-get)
  - Installer le driver NVIDIA
  - Installer Cuda
  - Installer le NVIDIA_GPU_COMPUTING_SDK
  - Récupérer les includes OpenCL (+ cl.hpp)
  - Installer python-setuptools (via apt-get)
  - Installer pyopencl

Sommaire

- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

Généralités sur OpenCL

- Historique :
  - Initié par Apple
  - En collaboration avec AMD, IBM, Intel, NVIDIA
  - Puis géré par le Kronos group
  - Version 1.0 (08/12/2008)
  - Version 1.1 (14/06/2010)
  - Version 1.2 (15/11/2011)

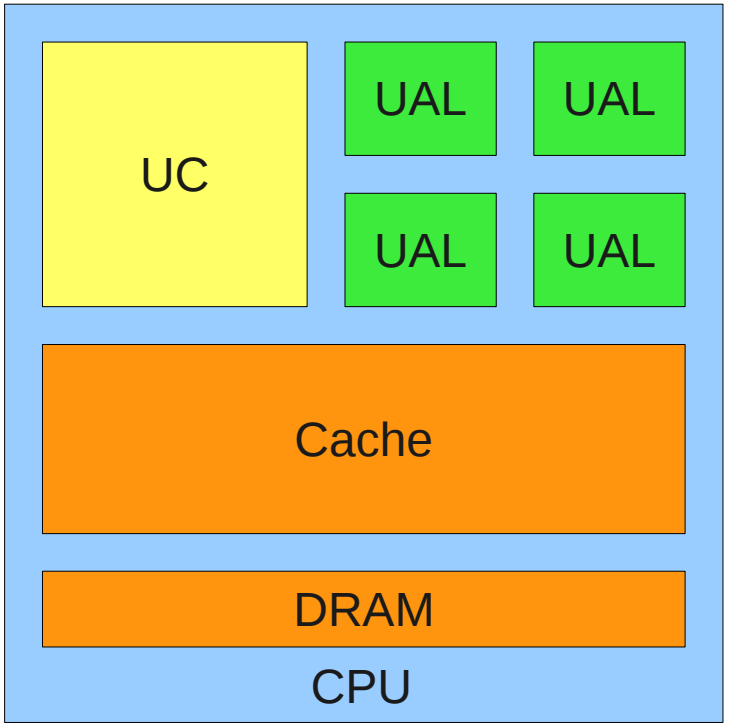


Généralités sur OpenCL

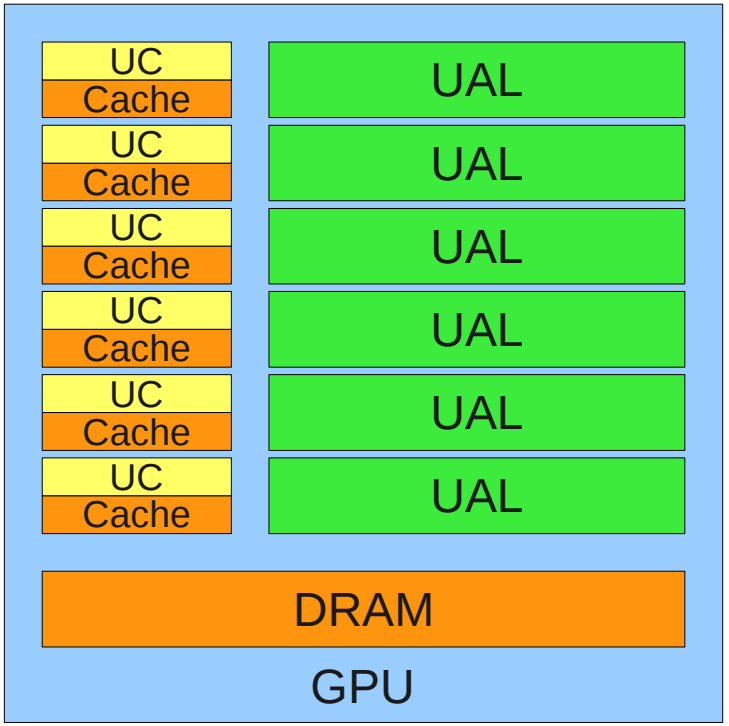
- Principe :
  - Standard ouvert
  - Framework pour écrire des programmes de calcul parallèle
  - Exécution sur des systèmes hétérogènes
  - Parallélisme de taches et de données

Généralités sur OpenCL

- Architectures :



Architecture généraliste



Parallélisme massif de données

Généralités sur OpenCL

- Mise en œuvre (en théorie) :
  - Installer les drivers matériels
  - Installer le runtime OpenCL
  - Installer le SDK OpenCL
- Problème (en pratique) :
  - Pour l'instant, pas de cohabitation des runtimes
  - SDK Intel → CPU Intel
  - SDK NV → GPU NV
  - SDK ATI → GPU ATI + CPU Intel/AMD

Généralités sur OpenCL

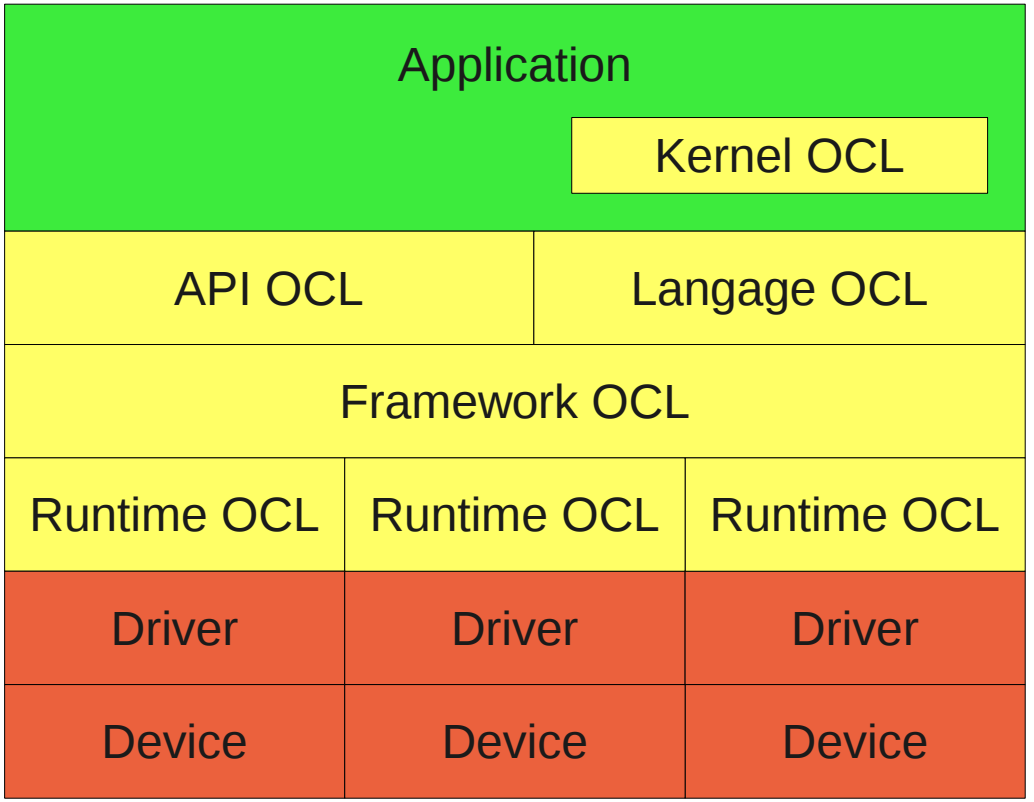
- Avantages de OpenCL :
  - Standard ouvert
  - Systèmes hétérogènes
  - Mécanisme de file d'attente évolué
- Avantages de Cuda :
  - Haut niveau (cuBLAS, cuFFT)
  - Mature (debugger, profiler...)

Sommaire

- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

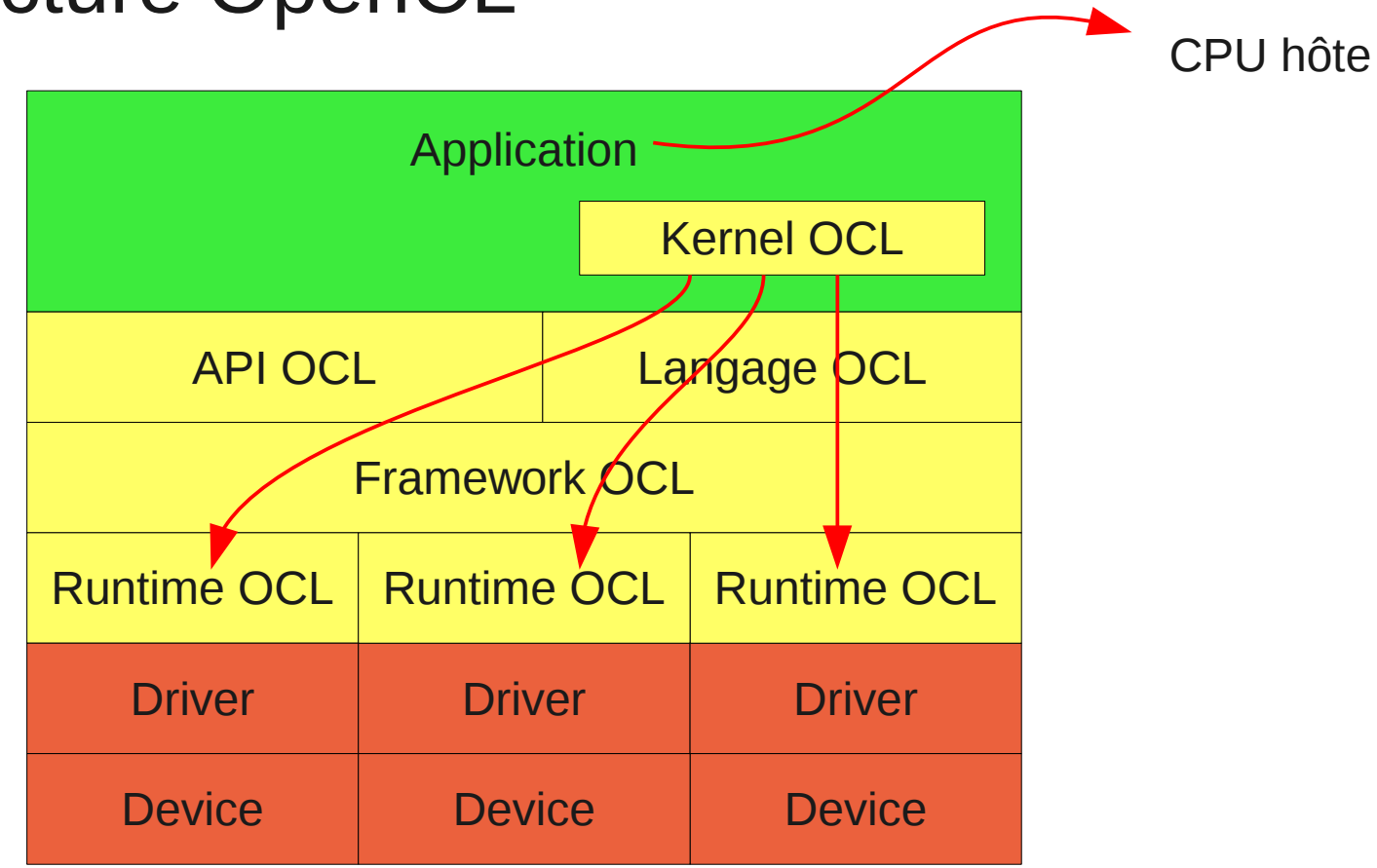
Modèles

- Architecture OpenCL



Modèles

- Architecture OpenCL

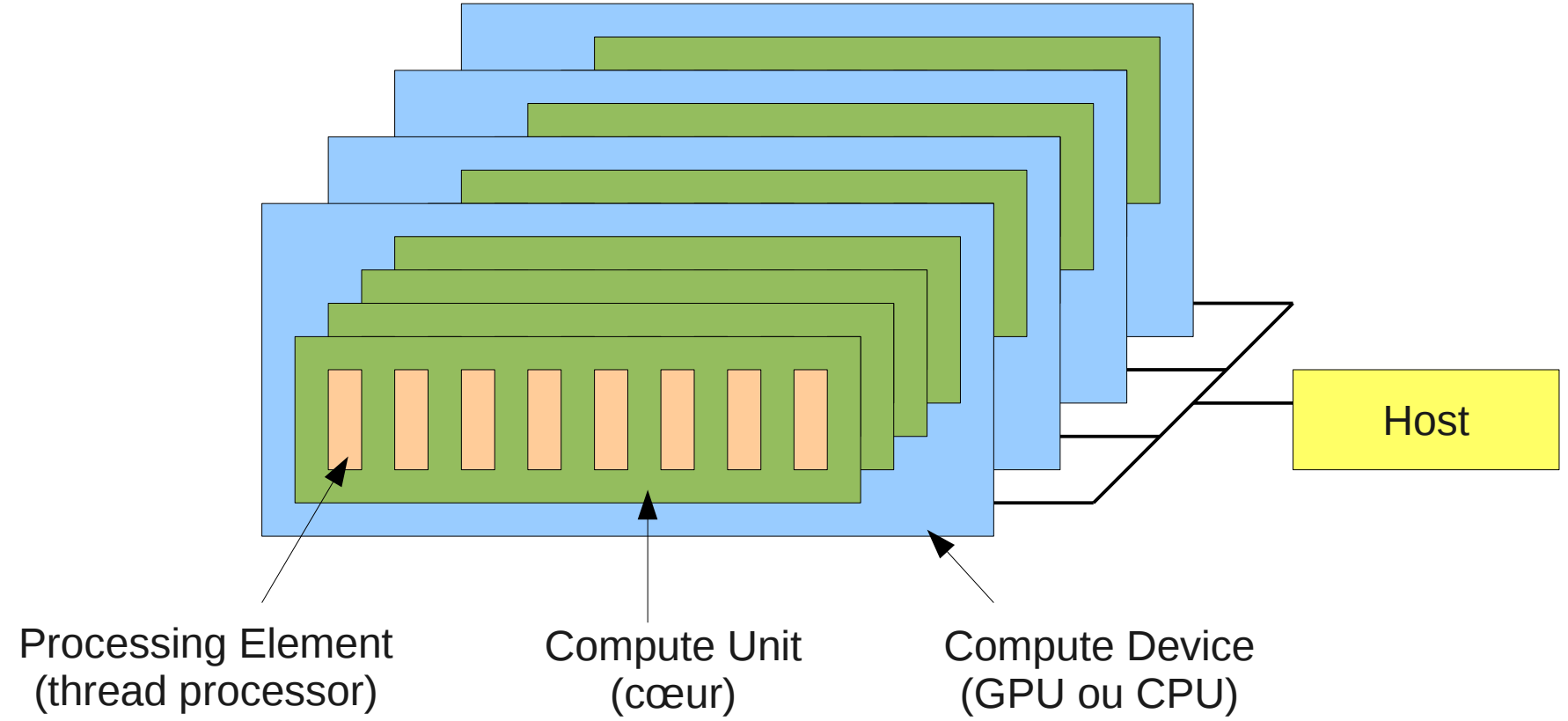


Modèles

- Architecture OpenCL
  - Baru (actuellement) :
    - Driver NVIDIA 285.05.09
    - Framework CUDA 4.1.1
    - OpenCL version 1.1
    - pyopencl 2011.2

Modèles

- Modèle de plate-forme



Modèles

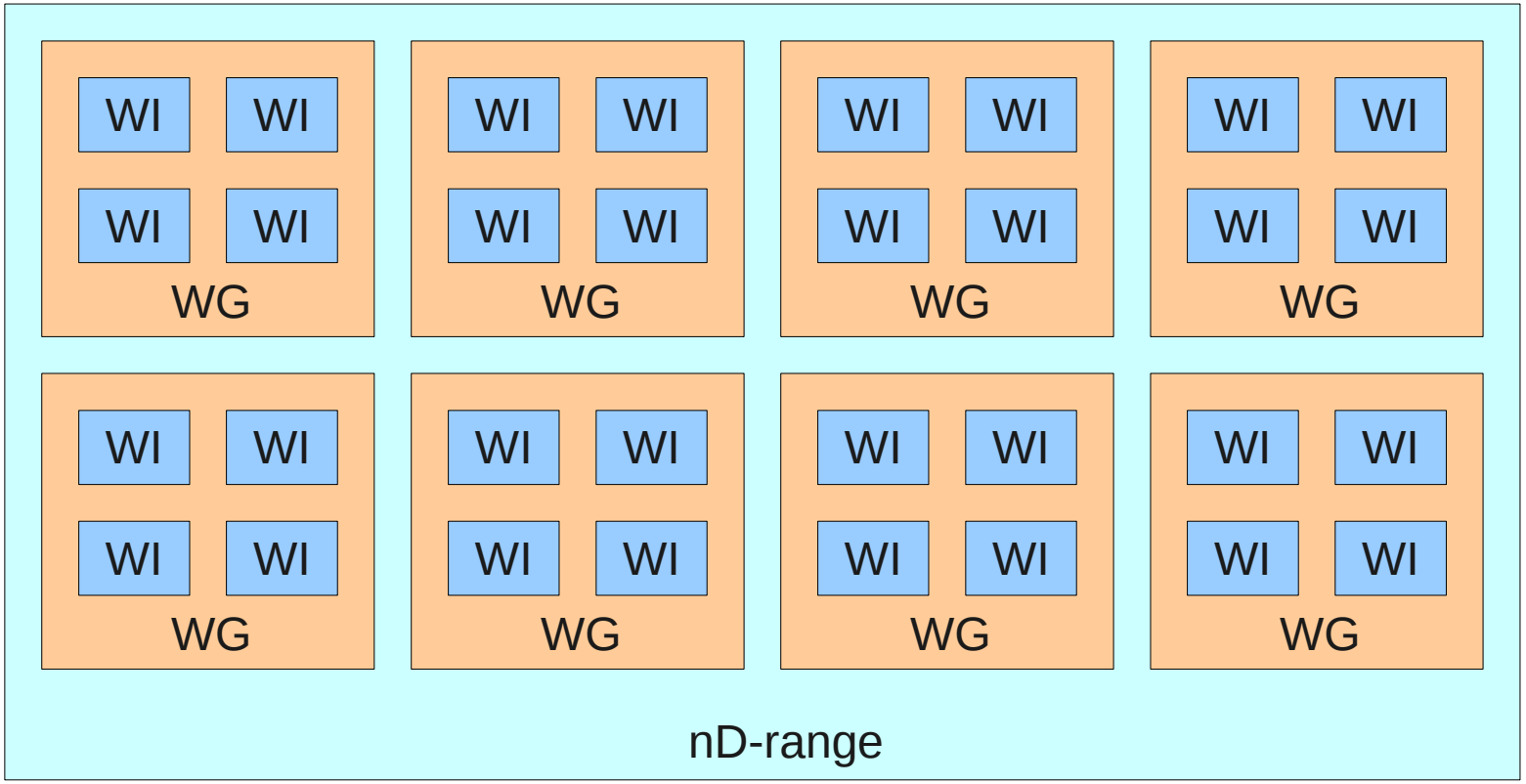
- Modèle de plate-forme
  - Baru :
    - 8 devices (GPU)
    - 14 compute units par device
    - 32 processing elements par compute unit
    - NB : 14x32 = 448 thread processors par device

Modèles

- Modèle d'exécution :
  - L'application (host) envoie les kernels sur les devices où ils sont instanciés en work-items
  - Chaque work-item a un identifiant qui lui permet de trouver ses données à utiliser
  - Notions de work-group, nD-range, warp

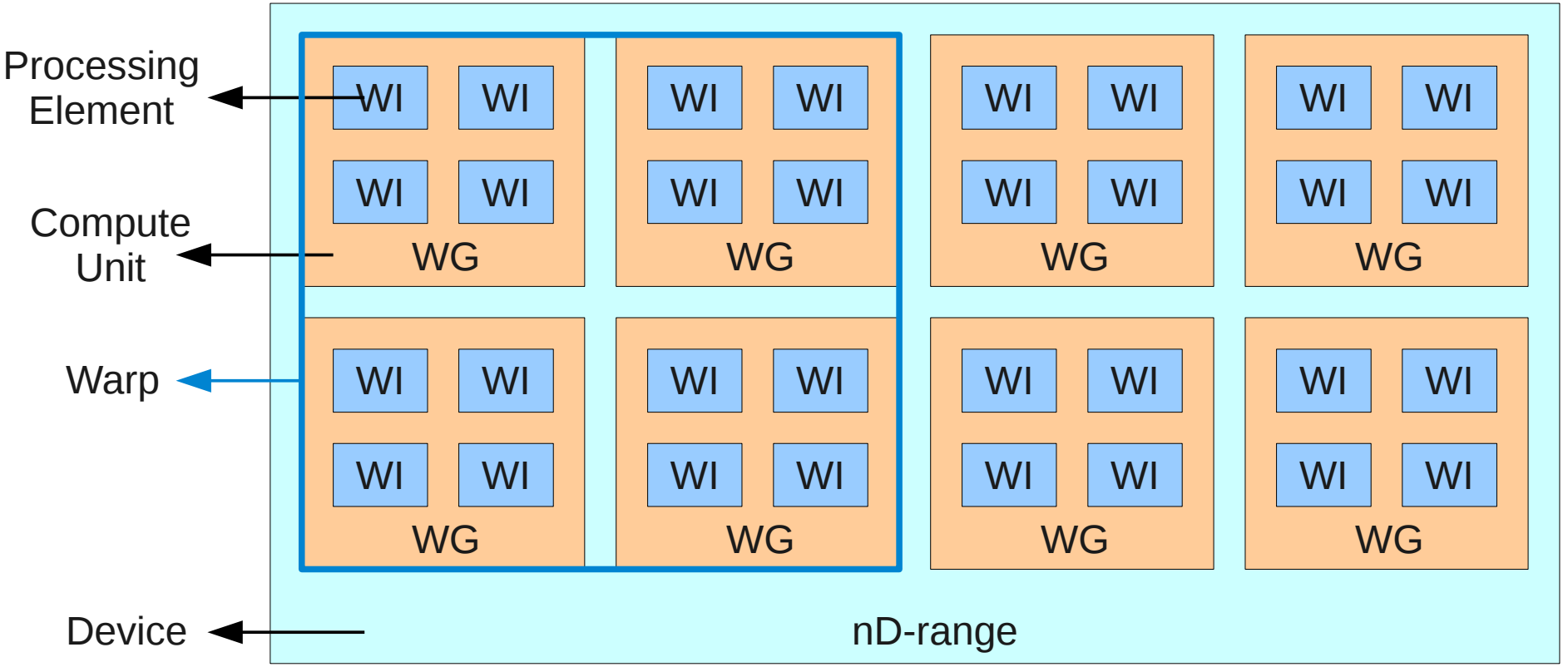
Modèles

- Modèle d'exécution



Modèles

- Modèle d'exécution

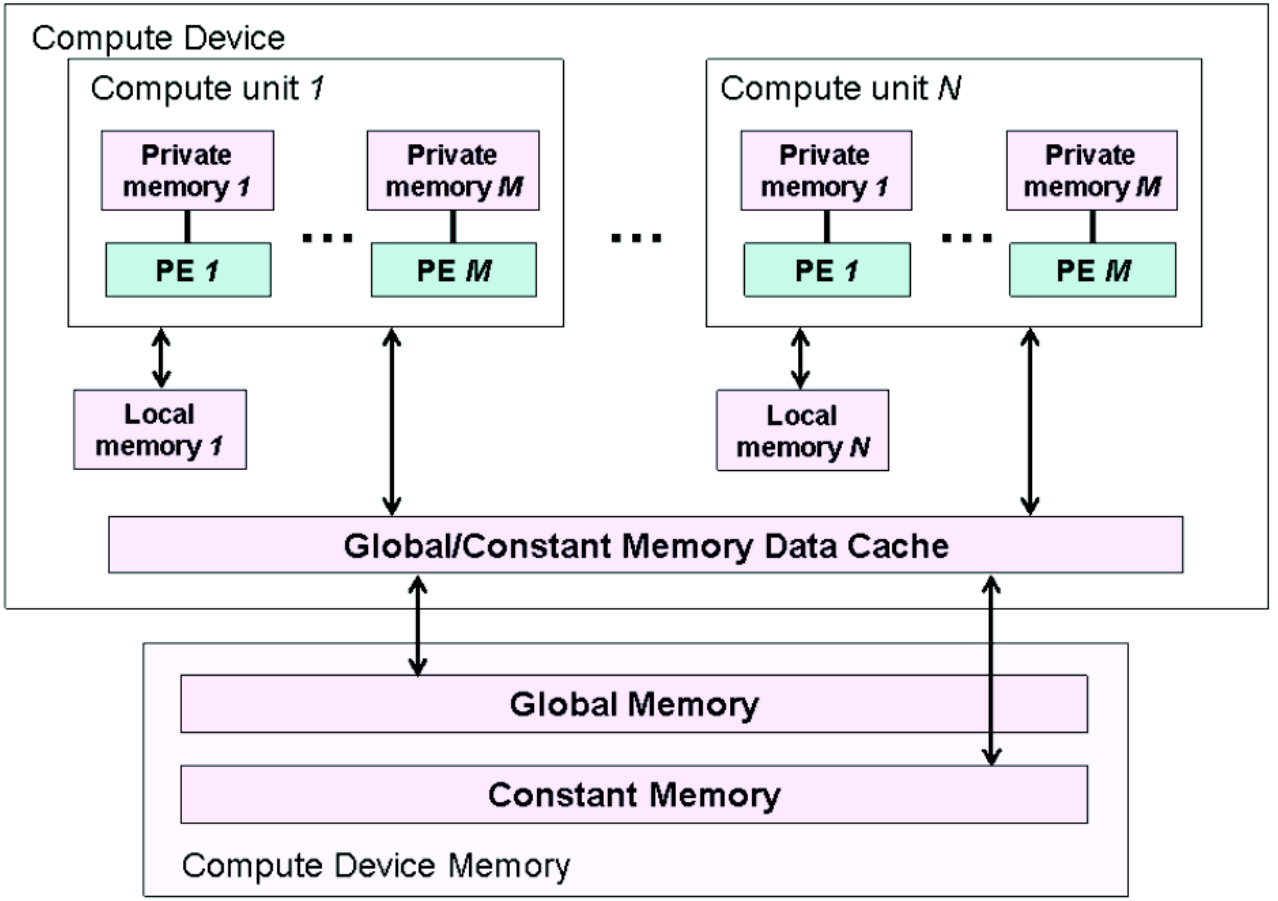


Modèles

- Modèle mémoire :
  - Global : accessible à tous les work-items
  - Constant : mémoire globale constante
  - Local : restreinte aux work-items d'un work-group
  - Private : restreinte au work-item

Modèles

- Modèle mémoire



Modèles

- Modèle mémoire

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static allocation Read-only access	Static allocation Read / Write access	Static allocation Read / Write access

Modèles

- Modèle de programmation :
  - Parallélisme de données :
    - Un même calcul (kernel) est effectué sur des données différentes (→ work-items)
    - Souvent : 1 work-item ↔ 1 données

    → GPU, CPU

Modèles

- Modèle de programmation :
  - Parallélisme de taches :
    - Le kernel est indépendant de l'espace de données
    - Revient à définir un kernel pour un seul work-item

    → CPU

Sommaire

- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

Programmation

- SDK OpenCL :
  - Host : API en C ; interfaces C++, java et python
  - Kernel : C99 + built-in functions

Programmation

- Références :
  - http://www.khronos.org/opencl/
  - NVIDIA_GPU_COMPUTING_SDK/OpenCL/doc/
  - http://mathema.tician.de/software/pyopencl/

Programmation

- Étapes :
  - Écrire le programme host
  - Écrire le kernel
  - Compiler, exécuter
  - NB : le kernel est compilé puis exécuté sur le device à l'exécution du programme host (compilation just-in-time).

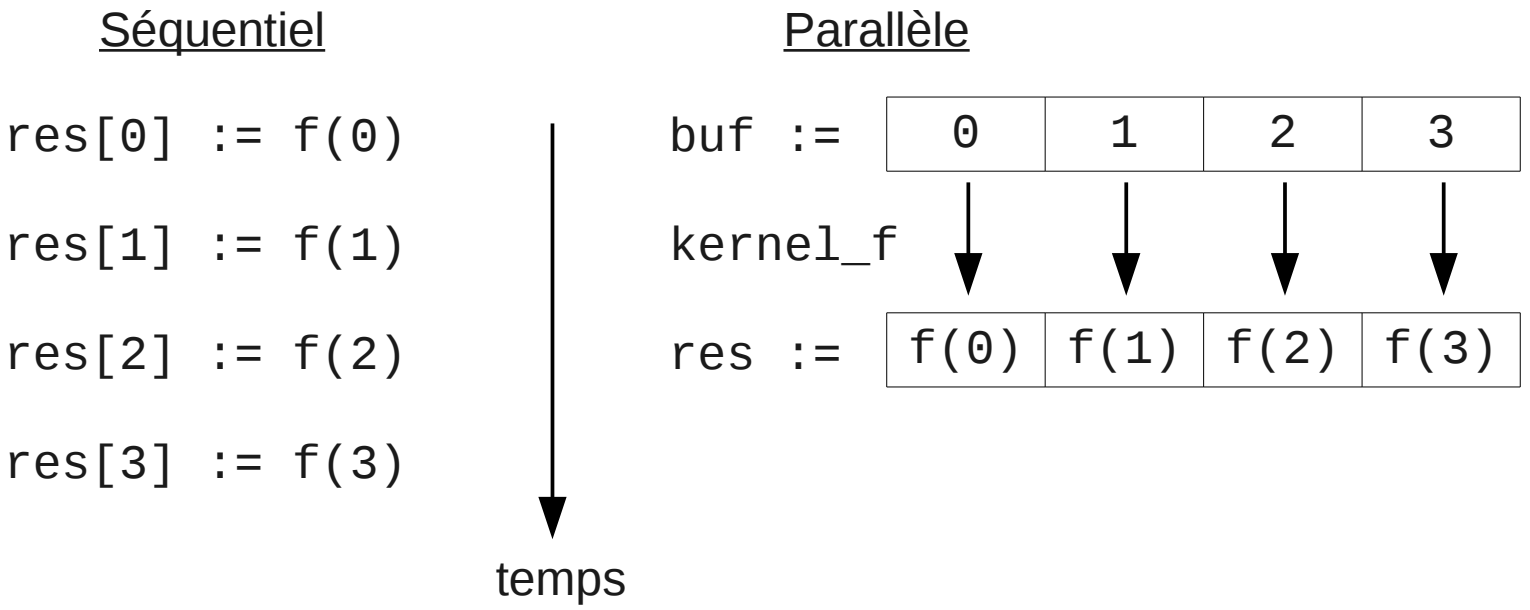
Programmation

- Programme type :
  - Importer OpenCL
  - Créer un contexte
  - Créer une file de commandes
  - Allouer et initialiser la mémoire du device
  - Charger et compiler le kernel
  - Insérer le kernel dans la file de commandes
  - Récupérer les données dans la mémoire du device
  - Libérer les ressources

Programmation

- Application type (parallélisme de données) :

```
Pour i de 0 a 3 :
   res[i] := f(i)
```



Programmation

- Exemple :
  - Ajouter 42 à tous les éléments d'un tableau de réels

Programmation

- Exemple en Python (1) :

```
#!/usr/bin/python
import numpy

# importer opencl
import pyopencl as cl

# creer un contexte
myContext = cl.create_some_context()

# creer une file de commandes
myQueue = cl.CommandQueue(myContext)

# allouer et initialiser la memoire du device
inputData = numpy.random.rand(50000).astype(numpy.float32)
outputData = numpy.empty_like(inputData)
myFlags = cl.mem_flags
inputBuffer = cl.Buffer(myContext,
            myFlags.READ_ONLY | myFlags.COPY_HOST_PTR, hostbuf=inputData)
outputBuffer = cl.Buffer(myContext, myFlags.WRITE_ONLY, inputData.nbytes)
```

Programmation

- Exemple en Python (2) :

```
# charger et compiler le kernel
myProgram = cl.Program(myContext, """
    __kernel void add42(__global const float *data, __global float *result)
    {
      int gid = get_global_id(0);
      result[gid] = data[gid] + 42.f;
    }
    """).build()

# ajouter le kernel dans la file de commandes
# recuperer les donnees dans la memoire du device
myProgram.add42(myQueue, inputData.shape, None, inputBuffer, outputBuffer)
cl.enqueue_copy(myQueue, outputData, outputBuffer)

# verifier le resultat du calcul
if abs(numpy.linalg.norm(outputData – (inputData + 42))) < 1e-6 :
    print "passed"
else:
    print "failed"
```

Programmation

- Exemple en C (1) :

```
// importer opencl
#include <CL/cl.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

// initialise les donnees a traiter
#define DATA_SIZE 50000
void fillData(float *data, unsigned int size) {
  int i;
  for (i=0; i<size; i++)
    data[i] = i;
}

// code du kernel
const char *kernelSource[] = {
  "__kernel void add42(__global float* data, __global float* result) {",
  " unsigned int i = get_global_id(0);",
  " result[i] = data[i] + 42.f;",
  "}"
};
```

Programmation

- Exemple en C (2) :

```
int main (int argc, char **argv) {
  // creer un contexte
  cl_platform_id platform;
  clGetPlatformIDs (1, &platform, NULL);
  cl_device_id device;
  clGetDeviceIDs (platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);
  cl_context context = clCreateContext (0, 1, &device, NULL, NULL, NULL);

  //creer une file de commandes
  cl_command_queue commandQueue = clCreateCommandQueue(context, device,0,0);

  // allouer et initialiser la memoire du device
  float inputDataHost[DATA_SIZE];
  fillData(inputDataHost, DATA_SIZE);
  cl_mem inputBufferDevice = clCreateBuffer (context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR, sizeof(float) * DATA_SIZE, inputDataHost, 0);
  cl_mem outputBufferDevice = clCreateBuffer (context, CL_MEM_WRITE_ONLY,
                                              sizeof (float) * DATA_SIZE, 0, 0);
```

Programmation

- Exemple en C (3) :

```
// charger et compiler le kernel
cl_program kernelProgram = clCreateProgramWithSource (context, 4,
                                                      kernelSource, 0, 0);
clBuildProgram (kernelProgram, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel (kernelProgram, "add42", NULL);
clSetKernelArg (kernel, 0, sizeof (cl_mem), (void *) &inputBufferDevice);
clSetKernelArg (kernel, 1, sizeof (cl_mem), (void *) &outputBufferDevice);

// ajouter le kernel dans la file de commandes
size_t WorkSize[1] = { DATA_SIZE };
clEnqueueNDRangeKernel (commandQueue, kernel, 1, 0, WorkSize, 0, 0, 0, 0);

// recuperer les donnees calculees dans la memoire du device
float outputDataHost[DATA_SIZE];
clEnqueueReadBuffer (commandQueue, outputBufferDevice, CL_TRUE, 0,
                DATA_SIZE * sizeof (float), outputDataHost, 0, NULL, NULL);
```

Programmation

- Exemple en C (4) :

```
  // liberer les ressources
  clReleaseKernel (kernel);
  clReleaseProgram (kernelProgram);
  clReleaseCommandQueue (commandQueue);
  clReleaseMemObject (inputBufferDevice);
  clReleaseMemObject (outputBufferDevice);
  clReleaseContext (context);

  // validation
  int i;
  for (i=0; i<DATA_SIZE; i++)
    if (fabs((inputDataHost[i] + 42.f) - outputDataHost[i]) > 1e-2)
      break;
  if (i == DATA_SIZE)
    printf ("passed\n");
  else
    printf ("failed\n");

  return 0;
}
```

Programmation

- Exemple en C++ (1) :

```
// importer opencl
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>
#include <iostream>
#include <cmath>

#define DATA_SIZE 50000

// code du kernel
const char kernelSource[] =
  "__kernel void add42(__global float* data, __global float* result) {"\
  " unsigned int i = get_global_id(0);"\
  " result[i] = data[i] + 42.f;"\
  "}";

int main() {
  // initialise les donnees a traiter
  float inputDataHost[DATA_SIZE];
  float outputDataHost[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++)
    inputDataHost[i] = i;
```

Programmation

- Exemple en C++ (2) :

```
try {
  // creer un contexte
  std::vector<cl::Platform> platforms;
  cl::Platform::get(&platforms);
  std::vector<cl::Device> devices;
  platforms[0].getDevices(CL_DEVICE_TYPE_ALL, &devices);
  cl::Context context(devices);

  // creer une file de commandes
  cl::CommandQueue queue(context, devices[0]);

  // allouer et initialiser la memoire du device
  cl::Buffer inputBufferDevice(context, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, sizeof(float) * DATA_SIZE, inputDataHost);
  cl::Buffer outputBufferDevice(context, CL_MEM_WRITE_ONLY,
                                sizeof(float) * DATA_SIZE);
```

Programmation

- Exemple en C++ (3) :

```
    // charger et compiler le kernel
    cl::Program::Sources source(1,
                          std::make_pair(kernelSource, strlen(kernelSource)));
    cl::Program program = cl::Program(context, source);
    program.build(devices);
    cl::Kernel kernel(program, "add42");
    kernel.setArg(0, inputBufferDevice);
    kernel.setArg(1, outputBufferDevice);

    // ajouter le kernel dans la file de commandes
    queue.enqueueNDRangeKernel(kernel, cl::NullRange,
                               cl::NDRange(DATA_SIZE), cl::NullRange);

    // recuperer les donnees calculees dans la memoire du device
    queue.enqueueReadBuffer(outputBufferDevice, CL_TRUE, 0,
                            DATA_SIZE * sizeof (float), outputDataHost);
  }
```

Programmation

- Exemple en C++ (4) :

```
  catch (cl::Error err) {
    std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")\n";
  }

  // validation
  for (int i=0; i<DATA_SIZE; i++)
    if (fabs((inputDataHost[i] + 42.f) - outputDataHost[i]) > 1e-2) {
      printf ("failed\n");
      return EXIT_FAILURE;
    }
  printf ("passed\n");
  return EXIT_SUCCESS;
}
```

Programmation

- Exemple en C++ (5) :
  - Avantages par rapport à l'interface C :
    - (Hiérarchie de classes)
    - Syntaxe un peu plus légère
    - Mécanisme d'exceptions
    - Libération automatique des ressources

Programmation

- Synchronisation :
  - Principe du calcul parallèle :
    - L'utilisateur envoie des requêtes (calculs, transferts mémoire)
    - Le système calcule en parallèle en maximisant l'utilisation des unités de calculs (plus de calculs en parallèle = temps de calcul total plus court)
    - Cohérence, synchronisation

Programmation

- Synchronisation :
  - Command queue :
    - File de commandes (calculs, transferts, barrières de synchronisation)
    - Plusieurs files possibles
    - File in-order ou out-of-order (pour le lancement des commandes)
  - Event :
    - Événement associé à une commande

Programmation

- Synchronisation :
  - Mécanismes de synchronisation (par granularité decroissante) :
    - clFinish (le host attend la fin de la file)
    - clWaitForEvent (le host attend la fin d'une commande)
    - clEnqueueBarrier (le device attend la fin des commandes antérieures)
    - clEnqueueWaitForEvents (le device attend la fin d'une commande)

Programmation

- Objets mémoire :
  - Types d'objet :
    - Buffer 1D
    - Image 2D ou 3D
  - Types de données :
    - Scalaire
    - Vectoriel
    - Structure définie par l'utilisateur

Programmation

- Transferts mémoire :
  - Coûteux
  - Lors de l'allocation ou via la file de commandes
  - Bloquants ou non-bloquants

Sommaire

- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

Optimisation

- Références (SDK NVIDIA) :
  - NVIDIA OpenCL Best Practices
  - NVIDIA OpenCL Programming Guide for the CUDA Architecture

Optimisation

- Généralités :
  - Méthode générale :
    - Valider l'implémentation
    - Mesurer/profiler l'exécution
    - Si nécessaire, optimiser les sections de code coûteuses
    - Boucler

Optimisation

- Généralités :
  - OpenCL :
    - Le code est portable (presque)
    - Mais pas les performances
    - Quelques principes généraux
    - Spécificités bas niveau
  - Remarques sur les GPU actuels :
    - ATI : unités vectorielles, accès mémoire par blocs
    - NV : unités scalaires, accès mémoire coalescents

Optimisation

- Quelques pistes :
  - Divergence :
    - i.e. part d'unités de calcul inoccupées
    - à minimiser :
      - Utiliser plusieurs files
      - Files out-of-order
      - Anticiper les accès/transferts mémoire
      - ...

Optimisation

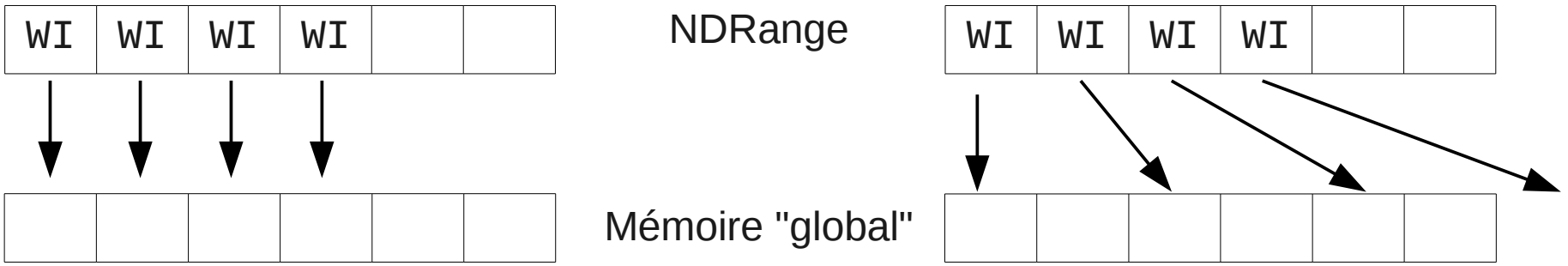
- Quelques pistes :
  - Tailles des work-groups :
    - Doivent être un multiple de la taille du warp
    - Influencent beaucoup l'efficacité des GPU
    - Dépendent du matériel

Optimisation

- Quelques pistes :
  - Mémoire "local" :
    - Accès global = environ 600 cycles
    - Accès local = environ 4 cycles
    - Mais local au work-group

Optimisation

- Quelques pistes :
  - Coalescing :
    - Accès contigu à la mémoire
    - Souvent plus rapide (moins de translations d'adresse)



Accès coalescent

Accès non-coalescent

Optimisation

- Quelques pistes :
  - Unrolling
  - Pipelining
  - Vectorisation
  - ...

Optimisation

- Exemple :
  - Multiplication de matrices

Sommaire

- Introduction
- Le calculateur du CGR/LISIC/LMPA
- Généralités sur OpenCL
- Modèles
- Programmation
- Optimisation
- Conclusion

Conclusion

- OpenCL :
  - Standard ouvert pour le calcul parallèle
  - Systèmes de calcul hétérogènes
  - Mécanismes intéressants (command queue)
  - Mais :
    - En maturation (API, runtime, outils)
    - Pas de portabilité des performances
    - Effort d'apprentissage (OpenCL, matériel)
    - Intégration propre dans un projet ?

Conclusion

- Remarques :
  - À long terme : parallélisme
  - Actuellement GPU intéressant
  - Choix Cuda/OpenCL :
    - Technologies très proches
    - Effort d'apprentissage
  - Évolution vers du CPU massif ?