Accélérateurs de calculs

Évolution des architectures haute-performance

Sylvain HENRY

sylvain.henry@labri.fr

Version 1.2 - 20 novembre 2012

Sommaire

1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

Lignes directrices

1 Introduction
- Historique
- Les accélérateurs
- Programmation des accélérateurs

2 Quelques exemples d'accélérateurs
- IBM CELL BroadBand Engine
- Cartes graphiques (GPU)
- Architectures hybrides GPU-CPU

3 Modèles de programmation
- Bas niveau : graphe de commandes
- Abstractions

Loi de Moore

Le nombre de transistors des microprocesseurs double tous les 2 ans

Microprocessor Transistor Counts 1971-2011 & Moore's Law

Transistor count

2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300

curve shows transistor count doubling every two years

16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
Quad-Core Itanium Tukwila
8-Core Xeon Nehalem-EX
AMD K10
POWER6
Itanium 2 with 9MB cache
Six-Core Opteron 2400
Core i7 (Quad)
AMD K10
Core 2 Duo
Cell
Itanium 2
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802

1971 1980 1990 2000 2011

Date of introduction

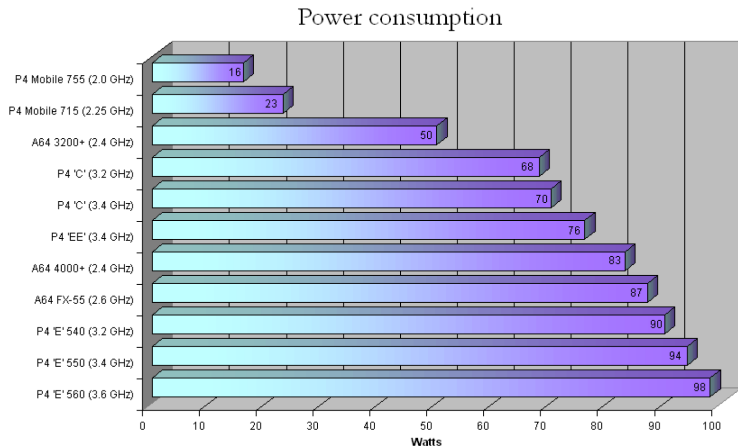
Loi de Moore

Que faire de l'espace libéré (i.e. avec les nouveaux transistors) ?

1. Multiplier le nombre d'unités
   1. Architectures super-scalaires
   2. Unités vectorielles (SIMD)
2. Multiplier le nombre de cœurs
   1. Architectures multi-cœurs
   2. Architectures multi-mémoires (NUMA)
3. Complexifier l'architecture
   1. Out-of-order
   2. Hiérarchie de caches améliorée (dimensions, nombre de niveaux...)
   3. Prédiction de branchement et exécution spéculative
   4. Renommage de registres

Limitations

Consommation énergétique et dissipation thermique



Limitations

Plus le nombre de cœurs augmente...

- Concurrence pour l'accès à la mémoire
- Coût des mécanismes de cohérence de cache
- Coût des mécanismes pour simuler une seule mémoire globale
  - Héritage des architectures de type Von Neumann
  - Plusieurs mémoires physiques
  - Les applications ne « voient » qu'une mémoire

Problème

Ne passe pas à l'échelle (plusieurs centaines, milliers de cœurs ?)

Limitations

Plus le nombre de cœurs augmente...

- Concurrence pour l'accès à la mémoire
- Coût des mécanismes de cohérence de cache
- Coût des mécanismes pour simuler une seule mémoire globale
  - Héritage des architectures de type Von Neumann
  - Plusieurs mémoires physiques
  - Les applications ne « voient » qu'une mémoire

Problème

Ne passe pas à l'échelle (plusieurs centaines, milliers de cœurs ?)

Solutions ?

- Architectures sans cohérence de cache
  - nccNUMA : non cache-coherent NUMA (vs ccNUMA)
  - Ne met en cache que la mémoire locale
  - Accès mémoires distantes plus chers
- Architectures spécialisées
  - Digital Signal Processor (DSP), systolic arrays, etc.
- Accélérateurs de calculs
  - Les différentes mémoires physiques sont gérées explicitement par les applications
  - Cœurs moins rapides mais beaucoup (beaucoup) plus nombreux

Dans tous les cas : changer assez radicalement d'architecture

Solutions ?

- Architectures sans cohérence de cache
  - nccNUMA : non cache-coherent NUMA (vs ccNUMA)
  - Ne met en cache que la mémoire locale
  - Accès mémoires distantes plus chers
- Architectures spécialisées
  - Digital Signal Processor (DSP), systolic arrays, etc.
- Accélérateurs de calculs
  - Les différentes mémoires physiques sont gérées explicitement par les applications
  - Cœurs moins rapides mais beaucoup (beaucoup) plus nombreux

Dans tous les cas : changer assez radicalement d'architecture

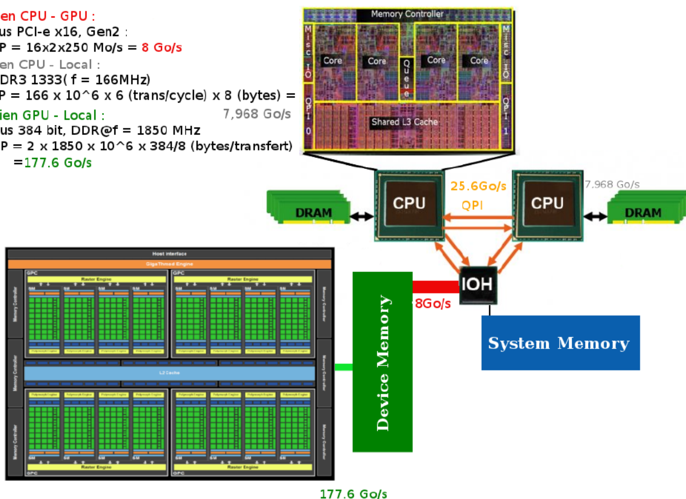
Lignes directrices

1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

Les accélérateurs

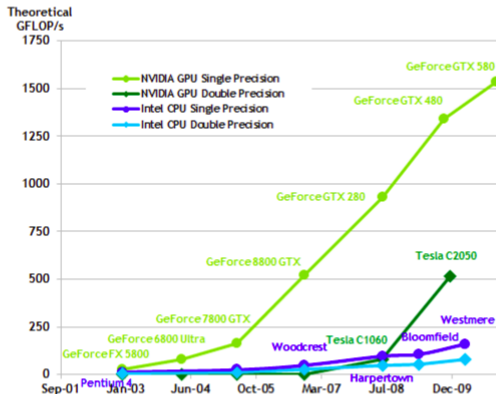
- Architectures hétérogènes
  - 1 ou plusieurs cœurs « classiques »
  - Plusieurs cœurs spécialisés (vectoriels en général)
  - Plusieurs mémoires
- Lien entre la mémoire de l'hôte et celles des accélérateurs
  - PCI Express
  - Lien spécifique
- Exemples
  - IBM CELL BroadBand Engine
  - Cartes graphiques
  - Architectures hybrides GPU/CPU (Intel MIC, AMD Fusion...)

Exemple d'architecture avec un accélérateur



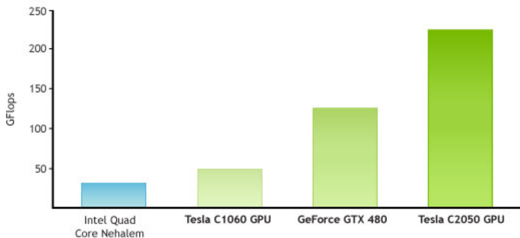
Performances

Performances théoriques (source NVIDIA)



Performances

Sur un exemple de code N-Body (source NVIDIA)



Performances N-body sur les CPU x86 avec plusieurs types de GPU

Top500

Classement de juin 2012

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom / 2011 IBM	1572864	16324.75	20132.66	7890.0
2	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	705024	10510.00	11280.38	12659.9
3	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	786432	8162.38	10066.33	3945.0
4	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR / 2012 IBM	147456	2897.00	3185.05	3422.7
5	National Supercomputing Center in Tianjin China	**Tianhe-1A** - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 / 2010 NUDT	186368	2566.00	4701.00	4040.0
6	DOE/SC/Oak Ridge National Laboratory United States	**Jaguar** - Cray XK6, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA 2090 / 2009 Cray Inc.	298592	1941.00	2627.61	5142.0
7	CINECA Italy	**Fermi** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	163840	1725.49	2097.15	821.9
8	Forschungszentrum Juelich (FZJ) Germany	**JuQUEEN** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom / 2012 IBM	131072	1380.39	1677.72	657.5
9	CEA/TGCC-GENCI France	**Curie thin nodes** - Bullx B510, Xeon E5-2680 8C 2.700GHz, Infiniband QDR / 2012 Bull	77184	1359.00	1667.17	2251.0
10	National Supercomputing Centre in Shenzhen (NSCS) China	**Nebulae** - Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 / 2010 Dawning	120640	1271.00	2984.30	2580.0

Lignes directrices

1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

Gestion mémoire

- Manuelle et explicite
  - transferts asynchrones (DMA)
  - pas de pagination
  - pas de mécanisme de swap
- Scratchpads
  - gérés explicitement
  - parfois partagés
  - pas de mécanisme de cohérence avec la mémoire globale
- Capacité mémoire faible
  - parfois quelques kilo-octets par cœur
- Contraintes similaires aux systèmes embarqués

Gestion mémoire

- Manuelle et explicite
  - transferts asynchrones (DMA)
  - pas de pagination
  - pas de mécanisme de swap
- Scratchpads
  - gérés explicitement
  - parfois partagés
  - pas de mécanisme de cohérence avec la mémoire globale
- Capacité mémoire faible
  - parfois quelques kilo-octets par cœur
- Contraintes similaires aux systèmes embarqués

Gestion mémoire

- Manuelle et explicite
  - transferts asynchrones (DMA)
  - pas de pagination
  - pas de mécanisme de swap
- Scratchpads
  - gérés explicitement
  - parfois partagés
  - pas de mécanisme de cohérence avec la mémoire globale
- Capacité mémoire faible
  - parfois quelques kilo-octets par cœur
- Contraintes similaires aux systèmes embarqués

Codes

- Différents modèles/langages de programmation
  - SIMD, SPMD...
- Différents compilateurs et plusieurs binaires par application
- Chargement explicite des binaires sur l'accélérateur
- Retour d'anciennes techniques
  - « Code overlay » lorsque la mémoire est insuffisante pour contenir le code complet du programme

Codes

- Différents modèles/langages de programmation
  - SIMD, SPMD...
- Différents compilateurs et plusieurs binaires par application
- Chargement explicite des binaires sur l'accélérateur
- Retour d'anciennes techniques
  - « Code overlay » lorsque la mémoire est insuffisante pour contenir le code complet du programme

Codes

- Différents modèles/langages de programmation
  - SIMD, SPMD...
- Différents compilateurs et plusieurs binaires par application
- Chargement explicite des binaires sur l'accélérateur
- Retour d'anciennes techniques
  - « Code overlay » lorsque la mémoire est insuffisante pour contenir le code complet du programme

Codes

- Différents modèles/langages de programmation
  - SIMD, SPMD...
- Différents compilateurs et plusieurs binaires par application
- Chargement explicite des binaires sur l'accélérateur
- Retour d'anciennes techniques
  - « Code overlay » lorsque la mémoire est insuffisante pour contenir le code complet du programme

Aspect distribué

- Similaire à l'utilisation d'un réseau de machines
  - Recouvrir les communications par le calcul
  - Limiter/optimiser les transferts
  - Placer les tâches à proximité des données qu'elles utilisent
- Équilibrage de charge
  - Quel type de tâche donner à chaque type de cœur ?
- Différences entre architectures à prendre en compte
  - Boutisme (*endianness*), alignements...

Aspect distribué

- Similaire à l'utilisation d'un réseau de machines
  - Recouvrir les communications par le calcul
  - Limiter/optimiser les transferts
  - Placer les tâches à proximité des données qu'elles utilisent
- Équilibrage de charge
  - Quel type de tâche donner à chaque type de cœur ?
- Différences entre architectures à prendre en compte
  - Boutisme (*endianness*), alignements...

Aspect distribué

- Similaire à l'utilisation d'un réseau de machines
  - Recouvrir les communications par le calcul
  - Limiter/optimiser les transferts
  - Placer les tâches à proximité des données qu'elles utilisent
- Équilibrage de charge
  - Quel type de tâche donner à chaque type de cœur ?
- Différences entre architectures à prendre en compte
  - Boutisme (*endianness*), alignements...

Lignes directrices

1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

IBM CELL BroadBand Engine

- Alliance entre Sony, Toshiba et IBM
- 4 années de développement, début en 2001
- 1 cœur de type Power4 simplifié (in-order) + 8 cœurs vectoriels
- Très bon rapport performance/consommation

- Supercalculateur RoadRunner
  - 6,120 Opteron (2 cœurs) + 12,240 PowerXCell 8i (9 cœurs)
  - En 2008, 1er du Top500 et 4e au Green500

- PlayStation 3 (2005)
  - Succés commercial
  - Accès « grand public » à l'architecture

IBM CELL BroadBand Engine

- Alliance entre Sony, Toshiba et IBM
- 4 années de développement, début en 2001
- 1 cœur de type Power4 simplifié (in-order) + 8 cœurs vectoriels
- Très bon rapport performance/consommation

- Supercalculateur RoadRunner
  - 6,120 Opteron (2 cœurs) + 12,240 PowerXCell 8i (9 cœurs)
  - En 2008, 1er du Top500 et 4e au Green500

- PlayStation 3 (2005)
  - Succés commercial
  - Accès « grand public » à l'architecture

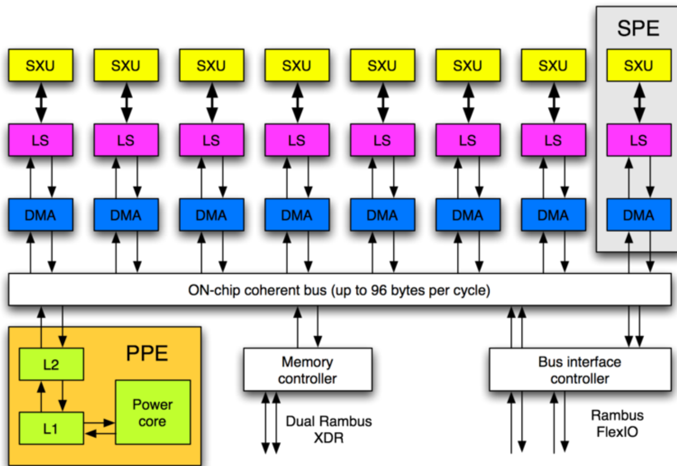
IBM CELL BroadBand Engine

- Alliance entre Sony, Toshiba et IBM
- 4 années de développement, début en 2001
- 1 cœur de type Power4 simplifié (in-order) + 8 cœurs vectoriels
- Très bon rapport performance/consommation

- Supercalculateur RoadRunner
  - 6,120 Opteron (2 cœurs) + 12,240 PowerXCell 8i (9 cœurs)
  - En 2008, 1er du Top500 et 4e au Green500

- PlayStation 3 (2005)
  - Succés commercial
  - Accès « grand public » à l'architecture

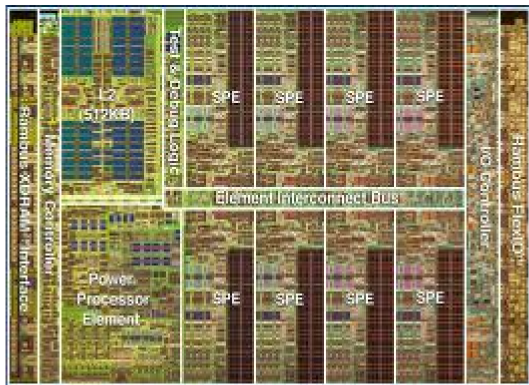
Schéma de l'architecture



Caractéristiques

- PPE (Power Processor Element)
  - Dual-cœur, in-order
  - L1 64ko L2 512ko
- SPE (x8, Synergistic Processing Element)
  - Uni-cœur, in-order
  - 128 registres de 128bits (SIMD)
  - Local store 256ko (LS, scratchpad, code + données)
  - 1 contrôleur DMA
- EIB (Element Interconnect Bus)
- Mémoire

Aspect physique



Super-calculateur RoadRunner

Cluster PS3

Cluster de test de l'US Air Force (336 PS3)

Programmation

- Context d'exécution
  - Exécution d'un programme sur un SPE
  - Une fois actif, le reste jusqu'à la fin du programme
  - Pas plus de 1 context actif par SPE
- Étapes sur l'hôte
  - Création de context : spe_create_context
  - Chargement d'un programme : spe_load_program
  - Exécution du programme : spe_context_run (bloquante)

Programmation

Exemple de code hôte

```
#include <libspe2.h>

extern spe_program_handle_t mon_kernel;

char buffer[128];

int main() {
    int entry = SPE_DEFAULT_ENTRY;

    int nb_spe = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
    spe_context_ptr_t context = spe_context_create(0, NULL);

    spe_program_load(context, &mon_kernel);

    spe_context_run(context, &entry, 0, buffer, (void*)128, NULL);

    spe_context_destroy(context);

    return 0;
}
```

Programmation

Exemple de code SPE

```c
#include <spu_mfcio.h>

int main(unsigned int speid, unsigned long long argp, unsigned long long envp) {

  char buffer[128] __attribute__((aligned(128)));

  int tag = 1, tag_mask = 1<<tag;

  mfc_get(buffer,(unsigned int) argp, envp, tag, 0, 0);

  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_all();

  return 0;
}
```

Lignes directrices

1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

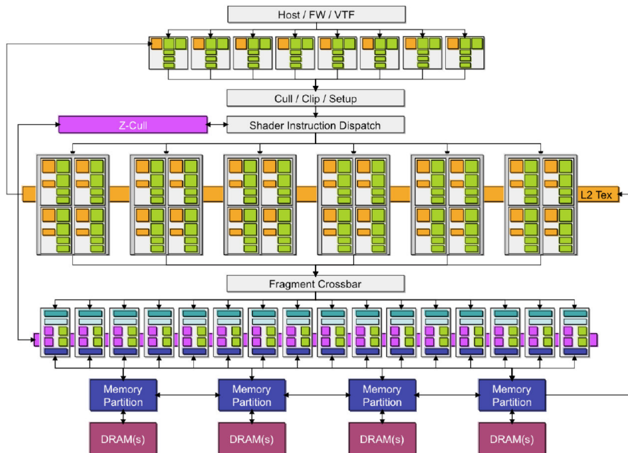
Cartes graphiques programmables (GPGPU)

General-Purpose computing on Graphics Processing Units (GPGPU)

- Quasiment toutes les machines sont équipées
- Programmables par le biais des shaders (pixel shaders, vertex shaders...)
  - Architectures massivement parallèles
    - calcul indépendant pour chaque sommet, chaque pixel, etc.
- Utilisation détournée pour faire des calculs non graphiques
  - Render-to-texture (RTT) plutôt qu'affichage à l'écran
- Gains de performances notables (x20...)

Architecture NVIDIA

GeForce 7800



Architecture NVIDIA

GeForce 7800

- Pipeline graphique programmable (3 niveaux)
  - pixel shaders
  - vertex shaders
  - geometry shaders
- Utilisation détournée pour le calcul non graphique
  - Difficile car les API (e.g. OpenGL, Cg) ne sont pas prévues pour ça
  - Difficulté pour équilibrer la charge des 3 niveaux
- Performance et faible consommation
  - Threads indépendants
  - Plusieurs bancs mémoire

Programmation

Exemple de code Cg (source Wikipedia)

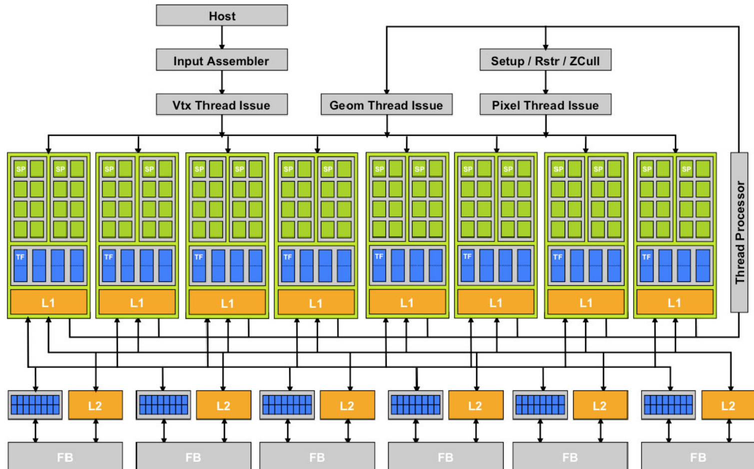
```
struct VertIn {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

struct VertOut {
    float4 pos   : POSITION;
    float4 color : COLOR0;
};

// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos     = mul(modelViewProj, IN.pos);
    // calculate output coords
    OUT.color   = IN.color;
    // copy input color to output
    OUT.color.z = 1.0f;
    // blue component of color = 1.0f
    return OUT;
}
```

Architecture NVIDIA

GeForce 8800



Architecture NVIDIA

GeForce 8800

- Architecture unifiée « Unified Shader Architecture »
- 8 cœurs contenant chacun 16 PE (Processing Element)
  - Les PE peuvent effectuer les trois types de shaders précédents
- Introduction de CUDA : Compute Unified Device Architecture
  - Langage proche du C pour le code exécuté sur le GPU
  - Extensions C/C++ pour programmer facilement les exécutions depuis le code hôte
- Architecture Tesla
  - « GPU » sans sortie graphique
  - Dédiée au calcul

Architecture NVIDIA

GeForce 8800

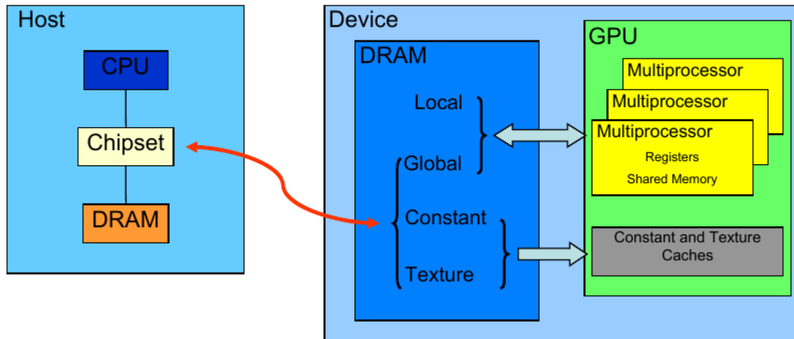
- Architecture unifiée « Unified Shader Architecture »
- 8 cœurs contenant chacun 16 PE (Processing Element)
  - Les PE peuvent effectuer les trois types de shaders précédents
- Introduction de CUDA : Compute Unified Device Architecture
  - Langage proche du C pour le code exécuté sur le GPU
  - Extensions C/C++ pour programmer facilement les exécutions depuis le code hôte
- Architecture Tesla
  - « GPU » sans sortie graphique
  - Dédiée au calcul

Architecture NVIDIA

GeForce 8800

- Architecture unifiée « Unified Shader Architecture »
- 8 cœurs contenant chacun 16 PE (Processing Element)
  - Les PE peuvent effectuer les trois types de shaders précédents
- Introduction de CUDA : Compute Unified Device Architecture
  - Langage proche du C pour le code exécuté sur le GPU
  - Extensions C/C++ pour programmer facilement les exécutions depuis le code hôte
- Architecture Tesla
  - « GPU » sans sortie graphique
  - Dédiée au calcul

Architecture NVIDIA
Modèle mémoire CUDA



Architecture NVIDIA

Modèle d'exécution CUDA

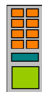
Software

Hardware

Thread

Thread Processor

Thread Block



Multiprocessor

Grid

Processor array

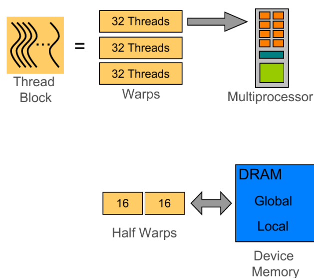
Architecture NVIDIA

Les différentes mémoires et leurs propriétés

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Architecture NVIDIA

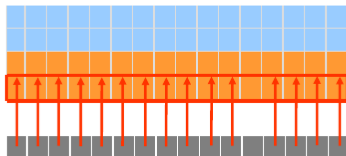
Modèle d'exécution CUDA : les warps



- Blocs de threads décomposés en « Warps » de 32 threads
  - Les threads d'un warp sont exécutés physiquement en parallèle (SIMD)
- Accès à la mémoire globale en une transaction par demi-wrap (16 threads) en cas de **coalescence**

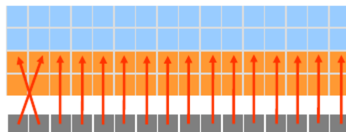
Architecture NVIDIA

Accès mémoire globale coalescent : Compute Capability 1.0 et 1.1

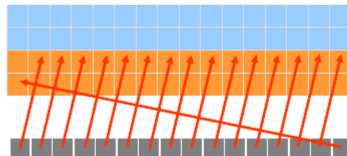


Coalesces – 1 transaction

Out of sequence – 16 transactions

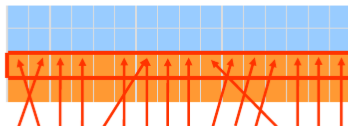


Misaligned – 16 transactions



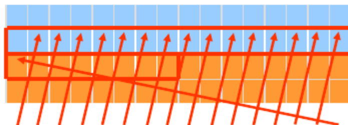
Architecture NVIDIA

Accès mémoire globale coalescent : Compute Capability 1.2

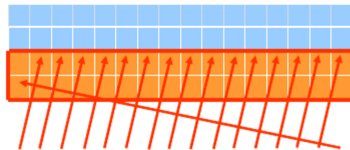


1 transaction - 64B segment

2 transactions - 64B and 32B segments

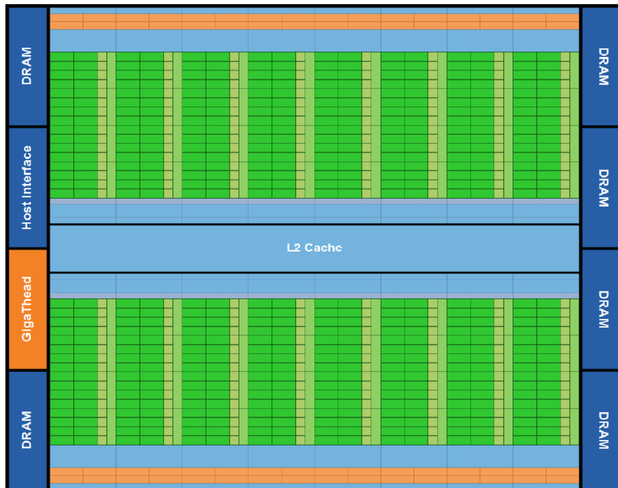


1 transaction - 128B segment

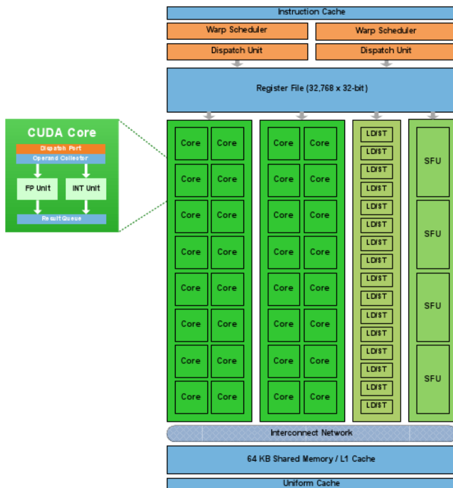


Architecture NVIDIA

Fermi

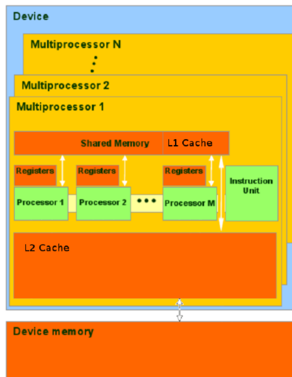


Architecture NVIDIA
Fermi



Architecture NVIDIA

Fermi : hiérarchie mémoire (cache L1 configurable + cache L2)



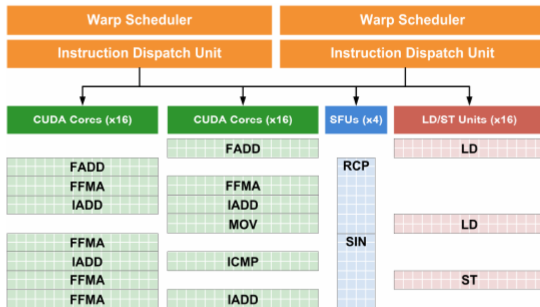
Architecture NVIDIA

Fermi

- Plusieurs SM (Streaming Multiprocessor)
  - contenant chacun 32 cœurs
  - 16 unités LOAD/STORE : accès aux différents bancs mémoire
  - le nombre de SM dépend... du prix de la carte

Architecture NVIDIA

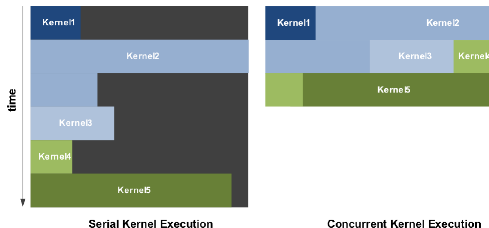
Fermi



- 32 instructions par cycle sur 2 des 4 blocs d'exécution
- les instructions proviennent de 2 warps
    - 1 warp est un ensemble de 32 threads

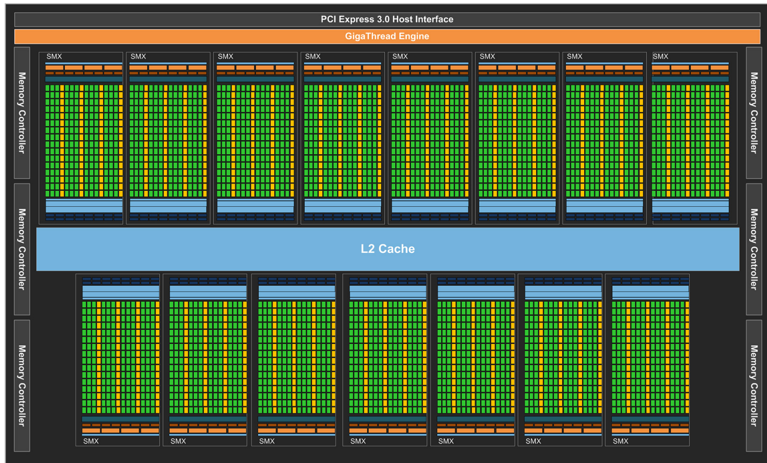
Architecture NVIDIA

Fermi : exécution concurrente de kernels



Architecture NVIDIA

Kepler



Architecture NVIDIA
Kepler

- 192 cœurs par SMX (streaming multiprocessor extreme)
- « Dynamic Parallelism »
  - Le tâches exécutées sur le GPU peuvent soumettre d'autres tâches
  - Moins de contrôle effectué par le CPU
  - Diminue le nombre de transferts sur le lien PCI Express
- Plus de coeurs, moins rapides
  - Meilleur rapport performance/consommation (x3 annoncé)

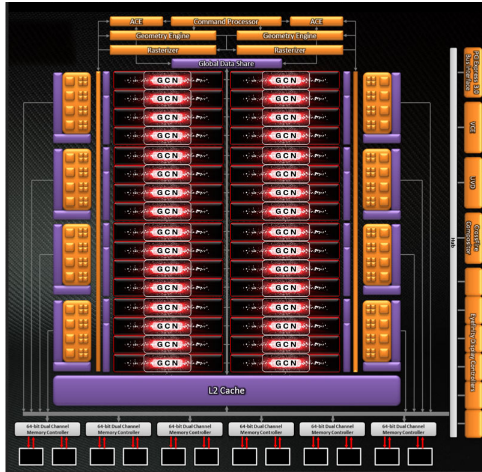
Architecture AMD/ATI

Anciennes architectures

- VLIW (Very Long Instruction Word)
  - *Bundles* d'instructions indépendantes formés par le compilateur

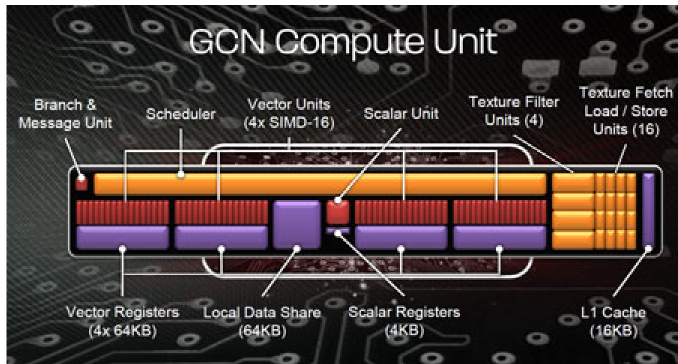
Architecture AMD/ATI

Radeon HD 7970 « Tahiti »



Architecture AMD/ATI

Graphics Core Next (GCN) architecture



Lignes directrices

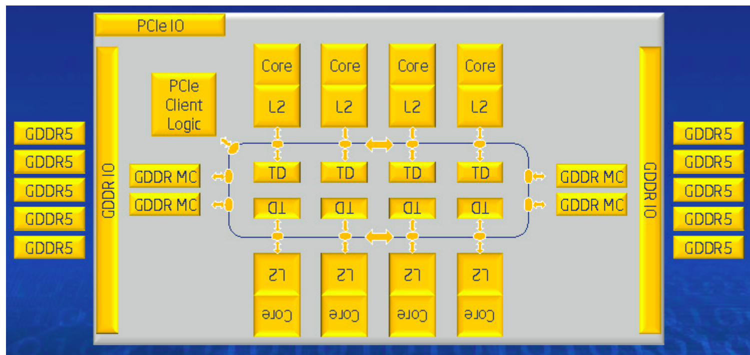
1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

Architectures hybrides GPU-CPU

- Intel ne veut par perdre son avantage face aux GPU
  - Architectures hybrides : ISA x86 avec architecture type GPU
  - Prototypes : Larrabee, SCC, MIC
  - Achat de concurrents (e.g. RapidMind)
  - Argument : plus simple à programmer (mêmes outils que sur CPU)
- Réaction de AMD
  - Fusion / Heterogeneous Systems Architectures (HSA)
  - Achat d'ATI en 2006

Intel Xeon Phi / MIC

Intel Many Integrated Core architecture (MIC)



- Plus d'informations après Super Computing 2012

Lignes directrices

1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

Bas niveau : graphe de commandes

Ce que l'application doit gérer

- Gestion mémoire
  - Allocation d'espaces dans les mémoires
  - Transferts de données entre les mémoires
    - Programmation des contrôleurs DMA
  - Libération des espaces alloués
- Gestion des codes
  - Compilation pour les différents types de cœurs
  - Transfert/chargement des binaires
  - Placement et ordonnancement

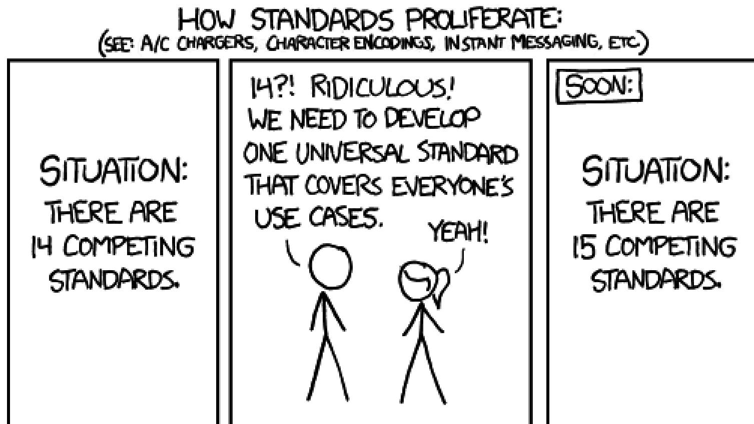
Frameworks

Frameworks utilisant ce modèle

- Bibliothèque SPE : programmation bas niveau du CELL BE
- CUDA : spécifique aux cartes graphiques NVidia
    - CUDA + Ocelot : cible aussi les CPUs
- CAL : spécifique aux cartes graphiques ATI/AMD
- Bibliothèque NUMA : programmation sur architectures NUMA
- OpenCL : nouveau standard censé unifier tous les autres

Standards

OpenCL et CUDA... (source : http ://xkcd.com/927/)

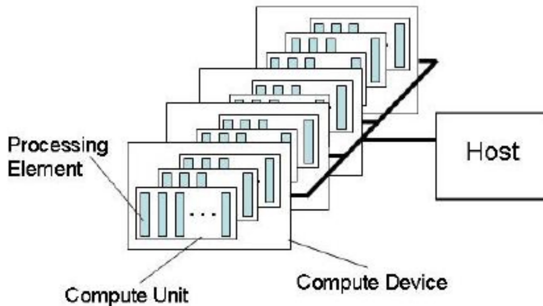


OpenCL

Open Computing Language

- Spécification du Khronos Group pour le calcul sur architectures hétérogènes
  - AMD/ATI, Apple, Intel, NVIDIA, SGI...
  - Première version en 2008
  - Dernière version à ce jour : version 1.2 le 15 novembre 2011
- Fortement inspiré par CUDA de NVIDIA (propriétaire)
- Disponible pour plusieurs accélérateurs et CPU
- Spécifie
  - Une API pour l'hôte, i.e. l'application qui contrôle les accélérateurs
  - Un langage avec lequel programmer les kernels exécutés sur les accélérateurs

OpenCL
Modèle global



- Un hôte connecté à un ou plusieurs « compute devices »

OpenCL

Modèle d'exécution

- Un programme exécuté sur l'hôte
  - Gère l'exécution des kernels
- Les *kernels* (noyaux de calcul) exécutés sur les accélérateurs
  - Un espace d'indices est défini lors de leur exécution
    - 1, 2 ou 3 dimensions
  - Pour chaque indice, une instance du kernel est exécutée
    - L'instance est appelée **work-item**
    - Identifiée par l'indice appelé **global ID**
    - Tous les work-items exécutent le même code mais peuvent diverger

OpenCL

Exemple : addition de matrices (kernel)

```
40 __kernel void matrix_add(float * a, float * b, float * c) {
41     int x = get_global_id(0);
42     int y = get_global_id(1);
43     int n = get_global_size(0);
44 
45     int off = y*n + x;
46 
47     c[off] = a[off] + b[off];
48 }
```

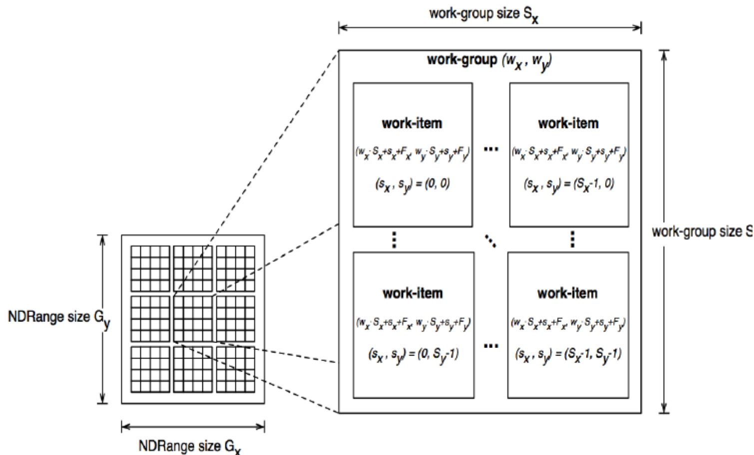
OpenCL

Modèle d'exécution

- L'espace d'indices est sub-divisé en **work-groups**
  - 1, 2 ou 3 dimensions (la même que l'espace d'indices)
  - Identifiés par un **work-group ID**
  - Les work-items se voient attribuer un **local ID** au sein de leur work-group
- Un work-item peut être identifié de façon unique par :
  - Son global ID
  - L'ID de son work-group et son propre local ID

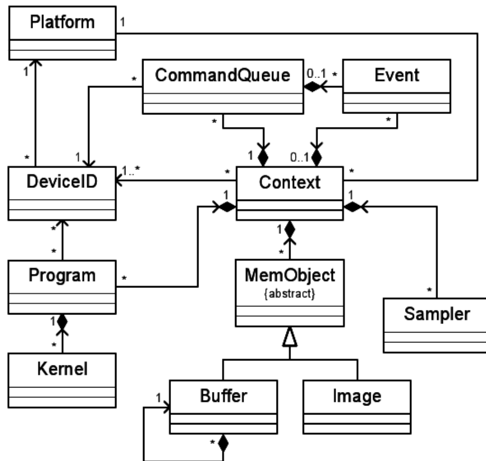
OpenCL

Modèle d'exécution : index space, work-groups, work-items...



OpenCL

API hôte : diagramme de classes (source : spécification 1.2)



OpenCL

Contextes et files de commandes

- Les contextes pour l'exécution des codes sur accélérateurs incluent
  - **Devices** Les accélérateurs utilisables par l'hôte
  - **Kernels** Les noyaux de calculs exécutables par les accélérateurs
  - **Program** Le code source et/ou les binaires des kernels
  - **Memory objects** Les espaces mémoires manipulés par l'hôte et les accélérateurs. Les kernels ne peuvent accéder qu'à ces objets
- Les files de commandes (**command queues**) coordonnent les exécutions de commandes asynchrones
  - Exécution de kernels
  - Transferts de données
  - Synchronisations

OpenCL

Files de commandes et évènements

- Les files de commandes peuvent être configurées
  - **In order** Les commandes soumises seront exécutées successivement lorsque la précédente termine
  - **Out of order** Les commandes sont exécutés successivement mais n'attendent pas que la précédente termine
- À chaque commande soumise est associé un évènement (**event object**)
  - Gestion fine des dépendances entre commandes et évènements

OpenCL
Types de kernels

- OpenCL supporte deux types de kernels
  - Kernels OpenCL écrits avec l'OpenCL C Language
  - Kernels natifs dépendant de l'implémentation
- Certains accélérateurs ne supportent que l'un des deux types

OpenCL
Modèle mémoire

- Global memory
  - Accessible en lecture/écriture par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Constant memory
  - Accessible en lecture seule par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Local memory
  - Mémoire locale à un work-group
  - Partagée en lecture/écriture par les work-items du work-group
- Private memory
  - Mémoire privée d'un work-item

OpenCL
Modèle mémoire

- Global memory
  - Accessible en lecture/écriture par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Constant memory
  - Accessible en lecture seule par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Local memory
  - Mémoire locale à un work-group
  - Partagée en lecture/écriture par les work-items du work-group
- Private memory
  - Mémoire privée d'un work-item

OpenCL

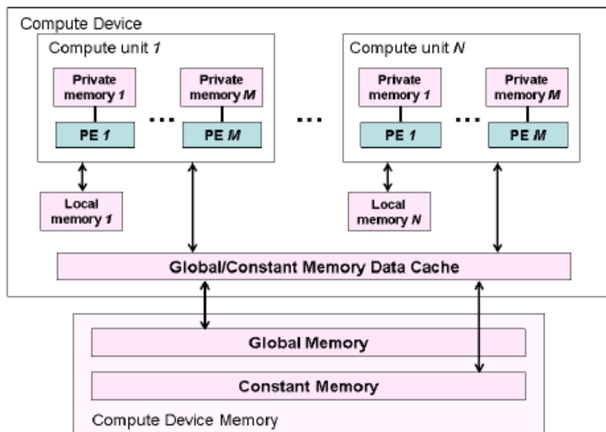
Modèle mémoire

- Global memory
  - Accessible en lecture/écriture par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Constant memory
  - Accessible en lecture seule par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Local memory
  - Mémoire locale à un work-group
  - Partagée en lecture/écriture par les work-items du work-group
- Private memory
  - Mémoire privée d'un work-item

OpenCL
Modèle mémoire

- Global memory
  - Accessible en lecture/écriture par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Constant memory
  - Accessible en lecture seule par tous les work-items
  - Accessible par l'hôte en lecture/écriture
- Local memory
  - Mémoire locale à un work-group
  - Partagée en lecture/écriture par les work-items du work-group
- Private memory
  - Mémoire privée d'un work-item

OpenCL
Modèle mémoire



OpenCL

Modèle mémoire : allocation et accès

	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

OpenCL
Cohérence mémoire

- OpenCL utilise un modèle mémoire à cohérence relâchée
- Au sein d'un work-item
  - load/store consistency
- Local memory
  - cohérence entre les work-items d'un work-group après une barrière
- Global memory
  - cohérence entre les work-items d'un work-group après une barrière
  - pas de cohérence entre les work-items de différents work-groups

OpenCL
Cohérence mémoire

- OpenCL utilise un modèle mémoire à cohérence relâchée
- Au sein d'un work-item
  - load/store consistency
- Local memory
  - cohérence entre les work-items d'un work-group après une barrière
- Global memory
  - cohérence entre les work-items d'un work-group après une barrière
  - pas de cohérence entre les work-items de différents work-groups

OpenCL

Cohérence mémoire

- OpenCL utilise un modèle mémoire à cohérence relâchée
- Au sein d'un work-item
  - load/store consistency
- Local memory
  - cohérence entre les work-items d'un work-group après une barrière
- Global memory
  - cohérence entre les work-items d'un work-group après une barrière
  - pas de cohérence entre les work-items de différents work-groups

OpenCL
Cohérence mémoire

- OpenCL utilise un modèle mémoire à cohérence relâchée
- Au sein d'un work-item
  - load/store consistency
- Local memory
  - cohérence entre les work-items d'un work-group après une barrière
- Global memory
  - cohérence entre les work-items d'un work-group après une barrière
  - pas de cohérence entre les work-items de différents work-groups

OpenCL

Synchronisations

- Synchronisations entre work-items d'un work-group
  - Barrières
  - Tous les work-items doivent passer par chaque barrière
  - Pas de mécanisme de synchronisation entre les work-groups
- Synchronisations entre les commandes
  - Files de commandes en mode « in-order »
  - Barrières dans les files de commandes
    - Exécute les commandes après la barrière quand celles avant la barrière ont terminé
  - Évènements
    - À chaque commande soumise est associé un évènement
    - Les commandes peuvent dépendre d'évènements

OpenCL
Synchronisations

- Synchronisations entre work-items d'un work-group
  - Barrières
  - Tous les work-items doivent passer par chaque barrière
  - Pas de mécanisme de synchronisation entre les work-groups
- Synchronisations entre les commandes
  - Files de commandes en mode « in-order »
  - Barrières dans les files de commandes
    - Exécute les commandes après la barrière quand celles avant la barrière ont terminé
  - Évènements
    - À chaque commande soumise est associé un évènement
    - Les commandes peuvent dépendre d'évènements

OpenCL
Synchronisations

- Synchronisations entre work-items d'un work-group
  - Barrières
  - Tous les work-items doivent passer par chaque barrière
  - Pas de mécanisme de synchronisation entre les work-groups
- Synchronisations entre les commandes
  - Files de commandes en mode « in-order »
  - Barrières dans les files de commandes
    - Exécute les commandes après la barrière quand celles avant la barrière ont terminé
  - Évènements
    - À chaque commande soumise est associé un évènement
    - Les commandes peuvent dépendre d'évènements

OpenCL
Synchronisations

- Synchronisations entre work-items d'un work-group
  - Barrières
  - Tous les work-items doivent passer par chaque barrière
  - Pas de mécanisme de synchronisation entre les work-groups
- Synchronisations entre les commandes
  - Files de commandes en mode « in-order »
  - Barrières dans les files de commandes
    - Exécute les commandes après la barrière quand celles avant la barrière ont terminé
  - Évènements
    - À chaque commande soumise est associé un évènement
    - Les commandes peuvent dépendre d'évènements

OpenCL

Exemple : somme des éléments d'un vecteur (kernel)

```
2  __kernel void vector_sum(float * v, float * res) {
3      int g_x = get_global_id(0);
4      int l_x = get_local_id(0);
5      int b_x = get_group_id(0);
6      int n = get_global_size(0);
7
8      __local v2[256];
9
10     v2[l_x] = v[g_x];
11
12     barrier(CLK_LOCAL_MEM_FENCE);
13
14     for (i=128; i!=0; i/=2) {
15        if (l_x < i) {
16           v2[l_x] += v2[l_x + i];
17        }
18
19        barrier(CLK_LOCAL_MEM_FENCE);
20     }
21
22     if (l_x == 0) {
23        res[b_x] = v2[l_x];
24     }
25 }
```

OpenCL

Exemple FAUX : tous les threads ne rencontrent pas la barrière

```
29 __kernel void vector_sum_wrong(float * v, float * res) {
30     int g_x = get_global_id(0);
31     int l_x = get_local_id(0);
32     int b_x = get_group_id(0);
33     int n = get_global_size(0);
34
35     __local v2[256];
36
37     v2[l_x] = v[g_x];
38
39     barrier(CLK_LOCAL_MEM_FENCE);
40
41     for (i=128; i!=0; i/=2) {
42        if (l_x < i) {
43           v2[l_x] += v2[l_x + i];
44
45           barrier(CLK_LOCAL_MEM_FENCE);
46        }
47
48     }
49
50     if (l_x == 0) {
51        res[b_x] = v2[l_x];
52     }
53 }
```

OpenCL
Objets mémoire

- OpenCL supporte deux types d'objets mémoire
  - Buffer et Image (regroupés sous le type **cl_mem**)
- Buffer
  - 1 dimension
  - Collection d'éléments de n'importe quel type (int, float, structures...)
  - Accessible directement (pointeur)
- Image
  - 2 ou 3 dimensions
  - Format prédéfini choisi parmi une liste
  - Utilisée comme texture ou comme *frame buffer*
  - Accessible à travers des fonctions prédéfinies
    - Le format de stockage n'est pas forcément le même que le format utilisé par le kernel

OpenCL
Objets mémoire

- OpenCL supporte deux types d'objets mémoire
  - Buffer et Image (regroupés sous le type **cl_mem**)
- Buffer
  - 1 dimension
  - Collection d'éléments de n'importe quel type (int, float, structures...)
  - Accessible directement (pointeur)
- Image
  - 2 ou 3 dimensions
  - Format prédéfini choisi parmi une liste
  - Utilisée comme texture ou comme *frame buffer*
  - Accessible à travers des fonctions prédéfinies
    - Le format de stockage n'est pas forcément le même que le format utilisé par le kernel

OpenCL
Objets mémoire

- OpenCL supporte deux types d'objets mémoire
  - Buffer et Image (regroupés sous le type **cl_mem**)
- Buffer
  - 1 dimension
  - Collection d'éléments de n'importe quel type (int, float, structures...)
  - Accessible directement (pointeur)
- Image
  - 2 ou 3 dimensions
  - Format prédéfini choisi parmi une liste
  - Utilisée comme texture ou comme *frame buffer*
  - Accessible à travers des fonctions prédéfinies
    - Le format de stockage n'est pas forcément le même que le format utilisé par le kernel

OpenCL
Le framework OpenCL

- OpenCL Platform Layer
  - Découverte des accélérateurs OpenCL et de leurs caractéristiques par l'hôte et création de contextes
- OpenCL Runtime
  - Manipulation des contextes et autres entités
- OpenCL Compiler
  - Compilation des kernels à exécuter sur les accélérateurs

OpenCL
Le framework OpenCL

- OpenCL Platform Layer
  - Découverte des accélérateurs OpenCL et de leurs caractéristiques par l'hôte et création de contextes
- OpenCL Runtime
  - Manipulation des contextes et autres entités
- OpenCL Compiler
  - Compilation des kernels à exécuter sur les accélérateurs

OpenCL
Le framework OpenCL

- OpenCL Platform Layer
  - Découverte des accélérateurs OpenCL et de leurs caractéristiques par l'hôte et création de contextes
- OpenCL Runtime
  - Manipulation des contextes et autres entités
- OpenCL Compiler
  - Compilation des kernels à exécuter sur les accélérateurs

OpenCL
API hôte : les plateformes

```
cl_int  clGetPlatformIDs (cl_uint num_entries,
                          cl_platform_id *platforms,
                          cl_uint *num_platforms)

cl_int  clGetPlatformInfo (cl_platform_id platform,
                           cl_platform_info param_name,
                           size_t param_value_size,
                           void *param_value,
                           size_t *param_value_size_ret)
```

cl_platform_info = CL_PLATFORM_PROFILE, CL_PLATFORM_VERSION, CL_PLATFORM_NAME, CL_PLATFORM_VENDOR, CL_PLATFORM_EXTENSIONS

OpenCL
API hôte : les accélérateurs

```
cl_int     clGetDeviceIDs4 (cl_platform_id platform,
                            cl_device_type device_type,
                            cl_uint num_entries,
                            cl_device_id *devices,
                            cl_uint *num_devices)

cl_int     clGetDeviceInfo (cl_device_id device,
                            cl_device_info param_name,
                            size_t param_value_size,
                            void *param_value,
                            size_t *param_value_size_ret)
```

OpenCL

API hôte : les accélérateurs

cl_device_type	Description
CL_DEVICE_TYPE_CPU	An OpenCL device that is the host processor. The host processor runs the OpenCL implementations and is a single or multi-core CPU.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU. By this we mean that the device can also be used to accelerate a 3D API such as OpenGL or DirectX.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (for example the IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCIe.
CL_DEVICE_TYPE_CUSTOM	Dedicated accelerators that do not support programs written in OpenCL C.
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system. The default device cannot be a CL_DEVICE_TYPE_CUSTOM device.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system except CL_DEVICE_TYPE_CUSTOM devices..

OpenCL

API hôte : informations sur les accélérateurs

cl_device_info	Return Type	Description
CL_DEVICE_TYPE	cl_device_type	The OpenCL device type. Currently supported values are: CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_ACCELERATOR, CL_DEVICE_TYPE_DEFAULT, a combination of the above types or CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_VENDOR_ID	cl_uint	A unique device vendor identifier. An example of a unique device identifier could be the PCIe ID.
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint	The number of parallel compute units on the OpenCL device. A work-group executes on a single compute unit. The minimum value is 1.
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	cl_uint	Maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. (Refer to **clEnqueueNDRangeKernel**). The minimum value is 3 for devices that are not of type CL_DEVICE_TYPE_CUSTOM.

OpenCL

API hôte : informations sur les accélérateurs

CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t []	Maximum number of work-items that can be specified in each dimension of the work-group to **clEnqueueNDRangeKernel**. Returns *n* size_t entries, where *n* is the value returned by the query for CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS. The minimum value is (1, 1, 1) for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t	Maximum number of work-items in a work-group executing a kernel on a single compute unit, using the data parallel execution model. (Refer to **clEnqueueNDRangeKernel**). The minimum value is 1.
CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR **CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT** **CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT**	cl_uint	Preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector.

OpenCL
API hôte : informations sur les accélérateurs

CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG **CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT** **CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE** **CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF**		If double precision is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF must return 0.
CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR **CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT** **CL_DEVICE_NATIVE_VECTOR_WIDTH_INT** **CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG** **CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT** **CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE** **CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF**	cl_uint	Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. If double precision is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE must return 0. If the **cl_khr_fp16** extension is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF must return 0.

OpenCL

API hôte : informations sur les accélérateurs

CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint	Maximum configured clock frequency of the device in MHz.
CL_DEVICE_ADDRESS_BITS	cl_uint	The default compute device address space size specified as an unsigned integer value in bits. Currently supported values are 32 or 64 bits.
CL_DEVICE_MAX_MEM_ALLOC_SIZE	cl_ulong	Max size of memory object allocation in bytes. The minimum value is max (1/4th of **CL_DEVICE_GLOBAL_MEM_SIZE** , 128*1024*1024) for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_IMAGE_SUPPORT	cl_bool	Is CL_TRUE if images are supported by the OpenCL device and CL_FALSE otherwise.
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint	Max number of simultaneous image objects that can be read by a kernel. The minimum value is 128 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE.
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint	Max number of simultaneous image objects that can be written to by a kernel. The minimum value is 8 if

OpenCL

API hôte : informations sur les accélérateurs

CL_DEVICE_MEM_BASE_ADDR_ALIGN	cl_uint	The minimum value is the size (in bits) of the largest OpenCL built-in data type supported by the device (`long16` in FULL profile, `long16` or `int16` in EMBEDDED profile) for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_cache_type	Type of global memory cache supported. Valid values are: CL_NONE, CL_READ_ONLY_CACHE and CL_READ_WRITE_CACHE.
CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint	Size of global memory cache line in bytes.
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong	Size of global memory cache in bytes.
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong	Size of global device memory in bytes.
CL_DEVICE_MAX_CONSTANT_	cl_ulong	Max size in bytes of a constant buffer

OpenCL

API hôte : informations sur les accélérateurs

BUFFER_SIZE		allocation. The minimum value is 64 KB for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint	Max number of arguments declared with the __constant qualifier in a kernel. The minimum value is 8 for devices that are not of type CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_local_mem_type	Type of local memory supported. This can be set to CL_LOCAL implying dedicated local memory storage such as SRAM, or CL_GLOBAL. For custom devices, CL_NONE can also be returned indicating no local memory support.
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	Size of local memory arena in bytes. The minimum value is 32 KB for devices that are not of type CL_DEVICE_TYPE_CUSTOM.

OpenCL
API hôte : informations sur les accélérateurs

CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool	Is CL_TRUE if the device implements error correction for all accesses to compute device memory (global and constant). Is CL_FALSE if the device does not implement such error correction.
CL_DEVICE_HOST_UNIFIED_MEMORY	cl_bool	Is CL_TRUE if the device and the host have a unified memory subsystem and is CL_FALSE otherwise.
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t	Describes the resolution of device timer. This is measured in nanoseconds. Refer to *section 5.12* for details.
CL_DEVICE_ENDIAN_LITTLE	cl_bool	Is CL_TRUE if the OpenCL device is a little endian device and CL_FALSE otherwise.

OpenCL
API hôte : informations sur les accélérateurs

CL_DEVICE_PLATFORM	cl_platform_id	The platform associated with this device.
CL_DEVICE_NAME	char[]	Device name string.
CL_DEVICE_VENDOR	char[]	Vendor name string.

OpenCL

API hôte : les contextes

```
cl_context   clCreateContext (const cl_context_properties *properties,
                              cl_uint num_devices,
                              const cl_device_id *devices,
                              void (CL_CALLBACK *pfn_notify)(const char *errinfo,
                                                 const void *private_info, size_t cb,
                                                 void *user_data),
                              void *user_data,
                              cl_int *errcode_ret)

cl_int       clReleaseContext (cl_context context)
```

OpenCL
API hôte : informations sur les contextes

```
cl_int    clGetContextInfo (cl_context context,
                            cl_context_info param_name,
                            size_t param_value_size,
                            void *param_value,
                            size_t *param_value_size_ret)
```

cl_context_info	Return Type	Information returned in param_value
CL_CONTEXT_REFERENCE_COUNT[8]	cl_uint	Return the *context* reference count.
CL_CONTEXT_NUM_DEVICES	cl_uint	Return the number of devices in *context*.
CL_CONTEXT_DEVICES	cl_device_id[]	Return the list of devices in *context*.
CL_CONTEXT_PROPERTIES	cl_context_properties[]	Return the *properties* argument specified in **clCreateContext** or **clCreateContextFromType**.

OpenCL

API hôte : les files de commandes

```
cl_command_queue   clCreateCommandQueue (cl_context context,
                                         cl_device_id device,
                                         cl_command_queue_properties properties,
                                         cl_int *errcode_ret)
```

Command-Queue Properties	Description
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Determines whether the commands queued in the command-queue are executed in-order or out-of-order. If set, the commands in the command-queue are executed out-of-order. Otherwise, commands are executed in-order. For a detailed description about CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, refer to *section 5.11*.
CL_QUEUE_PROFILING_ENABLE	Enable or disable profiling of commands in the command-queue. If set, the profiling of commands is enabled. Otherwise profiling of commands is disabled.

OpenCL
API hôte : les buffers

```
cl_mem    clCreateBuffer (cl_context context,
                          cl_mem_flags flags,
                          size_t size,
                          void *host_ptr,
                          cl_int *errcode_ret)

cl_int    clReleaseMemObject (cl_mem memobj)

cl_int    clSetMemObjectDestructorCallback (cl_mem memobj,
                          void (CL_CALLBACK *pfn_notify)(cl_mem memobj,
                                                         void *user_data),
                          void *user_data)
```

OpenCL
API hôte : les buffers

cl_mem_flags	Description
CL_MEM_READ_WRITE	This flag specifies that the memory object will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	This flag specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with CL_MEM_WRITE_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE and CL_MEM_WRITE_ONLY are mutually exclusive.
CL_MEM_READ_ONLY	This flag specifies that the memory object is a read-only memory object when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY and CL_MEM_READ_ONLY are mutually exclusive.

OpenCL

API hôte : les buffers

CL_MEM_USE_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by *host_ptr* as the storage bits for the memory object. OpenCL implementations are allowed to cache the buffer contents pointed to by *host_ptr* in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same *host_ptr* or overlapping host regions is considered to be undefined. Also refer to *section C.3* for a description of the alignment rules for *host_ptr* for memory objects (buffer and images) created using CL_MEM_USE_HOST_PTR.
CL_MEM_ALLOC_HOST_PTR	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.

OpenCL
API hôte : les buffers

CL_MEM_COPY_HOST_PTR	This flag is valid only if *host_ptr* is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by *host_ptr*. CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive. CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.
CL_MEM_HOST_WRITE_ONLY	This flag specifies that the host will only write to the memory object (using OpenCL APIs that enqueue a write or a map for write). This can be used to optimize write access from the host (e.g. enable write-combined allocations for memory objects for devices that communicate with the host over a system bus such as PCIe).

OpenCL

API hôte : les buffers

CL_MEM_HOST_READ_ONLY	This flag specifies that the host will only read the memory object (using OpenCL APIs that enqueue a read or a map for read). CL_MEM_HOST_WRITE_ONLY and CL_MEM_HOST_READ_ONLY are mutually exclusive.
CL_MEM_HOST_NO_ACCESS	This flag specifies that the host will not read or write the memory object. CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_READ_ONLY and CL_MEM_HOST_NO_ACCESS are mutually exclusive.

OpenCL

API hôte : les transferts de données hôte <-> device

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t size,
                            void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)

cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_write,
                             size_t offset,
                             size_t size,
                             const void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

OpenCL

API hôte : les transferts de données buffer -> buffer

```
cl_int  clEnqueueCopyBuffer (cl_command_queue command_queue,
                             cl_mem src_buffer,
                             cl_mem dst_buffer,
                             size_t src_offset,
                             size_t dst_offset,
                             size_t size,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

OpenCL

API hôte : les transferts de données hôte -> buffer (avec padding)

```
cl_int clEnqueueWriteBufferRect (cl_command_queue command_queue,
                                 cl_mem buffer,
                                 cl_bool blocking_write,
                                 const size_t *buffer_origin,
                                 const size_t *host_origin,
                                 const size_t *region,
                                 size_t buffer_row_pitch,
                                 size_t buffer_slice_pitch,
                                 size_t host_row_pitch,
                                 size_t host_slice_pitch,
                                 const void *ptr,
                                 cl_uint num_events_in_wait_list,
                                 const cl_event *event_wait_list,
                                 cl_event *event)
```

OpenCL

API hôte : les transferts de données buffer -> hôte (avec padding)

```
cl_int clEnqueueReadBufferRect (cl_command_queue command_queue,
                                cl_mem buffer,
                                cl_bool blocking_read,
                                const size_t *buffer_origin,
                                const size_t *host_origin,
                                const size_t *region,
                                size_t buffer_row_pitch,
                                size_t buffer_slice_pitch,
                                size_t host_row_pitch,
                                size_t host_slice_pitch,
                                void *ptr,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

OpenCL

API hôte : les transferts de données buffer -> buffer (avec padding)

```
cl_int clEnqueueCopyBufferRect (cl_command_queue command_queue,
                                cl_mem src_buffer,
                                cl_mem dst_buffer,
                                const size_t *src_origin,
                                const size_t *dst_origin,
                                const size_t *region,
                                size_t src_row_pitch,
                                size_t src_slice_pitch,
                                size_t dst_row_pitch,
                                size_t dst_slice_pitch,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

OpenCL

API hôte : *mapping* en mémoire hôte

```
void *  clEnqueueMapBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_map,
                            cl_map_flags map_flags,
                            size_t offset,
                            size_t size,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event,
                            cl_int *errcode_ret)

cl_int  clEnqueueUnmapMemObject (cl_command_queue command_queue,
                                 cl_mem memobj,
                                 void *mapped_ptr,
                                 cl_uint num_events_in_wait_list,
                                 const cl_event *event_wait_list,
                                 cl_event *event)
```

OpenCL

API hôte : *mapping* en mémoire hôte

cl_map_flags	Description
CL_MAP_READ	This flag specifies that the region being mapped in the memory object is being mapped for reading. The pointer returned by **clEnqueueMap{Buffer | Image}** is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMap{Buffer | Image}** command has completed
CL_MAP_WRITE	This flag specifies that the region being mapped in the memory object is being mapped for writing.

OpenCL

API hôte : *mapping* en mémoire hôte

	The pointer returned by **clEnqueueMap{Buffer | Image}** is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMap{Buffer | Image}** command has completed
CL_MAP_WRITE_INVALIDATE_REGION	This flag specifies that the region being mapped in the memory object is being mapped for writing. The contents of the region being mapped are to be discarded. This is typically the case when the region being mapped is overwritten by the host. This flag allows the implementation to no longer guarantee that the pointer returned by **clEnqueueMap{Buffer | Image}** contains the latest bits in the region being mapped which can be a significant performance enhancement. CL_MAP_READ or CL_MAP_WRITE and CL_MAP_WRITE_INVALIDATE_REGION are mutually exclusive.

OpenCL

API hôte : migration d'objets mémoire

```
cl_int clEnqueueMigrateMemObjects (cl_command_queue command_queue,
                                   cl_uint num_mem_objects,
                                   const cl_mem *mem_objects,
                                   cl_mem_migration_flags flags,
                                   cl_uint num_events_in_wait_list,
                                   const cl_event *event_wait_list,
                                   cl_event *event)
```

cl_mem_migration flags	Description
CL_MIGRATE_MEM_OBJECT_HOST	This flag indicates that the specified set of memory objects are to be migrated to the host, regardless of the target command-queue.
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED	This flag indicates that the contents of the set of memory objects are undefined after migration. The specified set of memory objects are migrated to the device associated with *command_queue* without incurring the overhead of migrating their contents.

- Les objets mémoire sont associés à un contexte, pas à un device
  - Besoin de forcer l'allocation sur un device particulier

OpenCL

API hôte : création de programmes

cl_program **clCreateProgramWithSource** (cl_context *context*,
cl_uint *count*,
const char ***strings*,
const size_t **lengths*,
cl_int **errcode_ret*)

cl_program **clCreateProgramWithBinary** (cl_context *context*,
cl_uint *num_devices*,
const cl_device_id **device_list*,
const size_t **lengths*,
const unsigned char ***binaries*,
cl_int **binary_status*,
cl_int **errcode_ret*)

cl_int **clReleaseProgram** (cl_program *program*)

OpenCL

API hôte : compilation de programmes + édition de liens

```
cl_int    clBuildProgram (cl_program program,
                          cl_uint num_devices,
                          const cl_device_id *device_list,
                          const char *options,
                          void (CL_CALLBACK *pfn_notify)(cl_program program,
                                                         void *user_data),
                          void *user_data)
```

OpenCL

API hôte : compilation de programmes

```
cl_int    clCompileProgram (cl_program program,
                             cl_uint num_devices,
                             const cl_device_id *device_list,
                             const char *options,
                             cl_uint num_input_headers,
                             const cl_program *input_headers,
                             const char **header_include_names,
                             void (CL_CALLBACK *pfn_notify)(cl_program program,
                                                            void *user_data),
                             void *user_data)
```

OpenCL

API hôte : édition de liens

```
cl_program   clLinkProgram (cl_context context,
                            cl_uint num_devices,
                            const cl_device_id *device_list,
                            const char *options,
                            cl_uint num_input_programs,
                            const cl_program *input_programs,
                            void (CL_CALLBACK *pfn_notify)(cl_program program,
                                                           void *user_data),
                            void *user_data,
                            cl_int *errcode_ret)
```

OpenCL
API hôte : informations sur la compilation

```
cl_int    clGetProgramBuildInfo (cl_program program,
                                 cl_device_id device,
                                 cl_program_build_info param_name,
                                 size_t param_value_size,
                                 void *param_value,
                                 size_t *param_value_size_ret)
```

OpenCL

API hôte : informations sur la compilation

cl_program_buid_info	Return Type	Info. returned in *param_value*
CL_PROGRAM_BUILD_STATUS	cl_build_status	Returns the build, compile or link status, whichever was performed last on *program* for *device*. This can be one of the following: CL_BUILD_NONE. The build status returned if no **clBuildProgram**, **clCompileProgram** or **clLinkProgram** has been performed on the specified program object for *device*. CL_BUILD_ERROR. The build status returned if **clBuildProgram**, **clCompileProgram** or **clLinkProgram** whichever was performed last on the specified program object for *device* generated an error. CL_BUILD_SUCCESS. The build status returned if **clBuildProgram**, **clCompileProgram** or **clLinkProgram** whichever was performed last on the specified program object for *device* was successful. CL_BUILD_IN_PROGRESS. The build status returned if **clBuildProgram**, **clCompileProgram** or **clLinkProgram** whichever was performed last on the specified program object for *device* has not finished.

OpenCL

API hôte : informations sur la compilation

CL_PROGRAM_BUILD_ LOG	char[]	Return the build, compile or link log for **clBuildProgram**, **clCompileProgram** or **clLinkProgram** whichever was performed last on *program* for *device*. If build status of *program* for *device* is CL_BUILD_NONE, an empty string is returned.

OpenCL

API hôte : création de kernels

```
cl_kernel   clCreateKernel (cl_program program,
                            const char *kernel_name,
                            cl_int *errcode_ret)

cl_int      clCreateKernelsInProgram (cl_program program,
                                      cl_uint num_kernels,
                                      cl_kernel *kernels,
                                      cl_uint *num_kernels_ret)

cl_int      clReleaseKernel (cl_kernel kernel)
```

- Un kernel est une fonction qualifiée par `__kernel` dans le code du programme

OpenCL
API hôte : paramètres des kernels

```
cl_int clSetKernelArg (cl_kernel kernel,
                       cl_uint arg_index,
                       size_t arg_size,
                       const void *arg_value)
```

- `arg_value` peut pointer vers un `cl_mem`
- Pour les paramètres en mémoire locale
  - `arg_value` est à NULL
  - `arg_size` indique la quantité de mémoire à allouer

OpenCL

API hôte : exécution de kernels

```
cl_int  clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                cl_kernel kernel,
                                cl_uint work_dim,
                                const size_t *global_work_offset,
                                const size_t *global_work_size,
                                const size_t *local_work_size,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

OpenCL

API hôte : exécution de kernels natifs

```
cl_int  clEnqueueNativeKernel (cl_command_queue command_queue,
                               void (CL_CALLBACK *user_func)(void *)
                               void *args,
                               size_t cb_args,
                               cl_uint num_mem_objects,
                               const cl_mem *mem_list,
                               const void **args_mem_loc,
                               cl_uint num_events_in_wait_list,
                               const cl_event *event_wait_list,
                               cl_event *event)
```

OpenCL
API hôte : évènements

- Les évènements sont associés aux commandes soumises
- Les commandes peuvent dépendre d'évènements
  - Elles sont exécutées quand les commandes associées aux évènements sont terminées (état CL_COMPLETE).

cl_event	**clCreateUserEvent** (cl_context *context*, cl_int **errcode_ret*)
cl_int	**clSetUserEventStatus** (cl_event *event*, cl_int *execution_status*)
cl_int	**clWaitForEvents** (cl_uint *num_events*, const cl_event **event_list*)
cl_int	**clReleaseEvent** (cl_event *event*)

OpenCL

API hôte : évènements

```
cl_int    clGetEventInfo (cl_event event,
                          cl_event_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret)

cl_int    clSetEventCallback (cl_event event,
                          cl_int command_exec_callback_type,
                          void (CL_CALLBACK *pfn_event_notify)(cl_event event,
                                          cl_int event_command_exec_status,
                                          void *user_data),
                          void *user_data)
```

OpenCL

API hôte : marqueurs, barrières...

```
cl_int    clEnqueueMarkerWithWaitList (cl_command_queue command_queue,
                                       cl_uint num_events_in_wait_list,
                                       const cl_event *event_wait_list,
                                       cl_event *event)

cl_int    clEnqueueBarrierWithWaitList (cl_command_queue command_queue,
                                        cl_uint num_events_in_wait_list,
                                        const cl_event *event_wait_list,
                                        cl_event *event)

cl_int    clFinish (cl_command_queue command_queue)
```

OpenCL

API hôte : profiling

```
cl_int    clGetEventProfilingInfo (cl_event event,
                                   cl_profiling_info param_name,
                                   size_t param_value_size,
                                   void *param_value,
                                   size_t *param_value_size_ret)
```

OpenCL

API hôte : profiling

cl_profiling_info	Return Type	Info. returned in *param_value*
CL_PROFILING_COMMAND_QUEUED	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* that has been enqueued is submitted by the host to the device associated with the command-queue.
CL_PROFILING_COMMAND_START	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* starts execution on the device.
CL_PROFILING_COMMAND_END	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* has finished execution on the device.

OpenCL

OpenCL Language : généralités

- Sous-ensemble du C, étendu
- Qualificateurs d'espace d'adressage
  - `__global`, `__local`, `__constant`, `__private`
  - `__kernel` : qualificateur de kernel

```
__kernel void my_func(…)
{
        __local float    a;    // A single float allocated
                               // in local address space
```

OpenCL

OpenCL Language : fonctions prédéfinies (global ID)

uint **get_work_dim** ()	Returns the number of dimensions in use. This is the value given to the *work_dim* argument specified in **clEnqueueNDRangeKernel**. For **clEnqueueTask**, this returns 1.
size_t **get_global_size** (uint *dimindx*)	Returns the number of global work-items specified for dimension identified by *dimindx*. This value is given by the *global_work_size* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_global_size()** returns 1. For **clEnqueueTask**, this always returns 1.
size_t **get_global_id** (uint *dimindx*)	Returns the unique global work-item ID value for dimension identified by *dimindx*. The global work-item ID specifies the work-item ID based on the number of global work-items specified to execute the kernel. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_global_id()** returns 0. For **clEnqueueTask**, this returns 0.

OpenCL

OpenCL Language : fonctions prédéfinies (local ID)

size_t **get_local_size** (uint *dimindx*)	Returns the number of local work-items specified in dimension identified by *dimindx*. This value is given by the *local_work_size* argument to **clEnqueueNDRangeKernel** if *local_work_size* is not NULL; otherwise the OpenCL implementation chooses an appropriate *local_work_size* value which is returned by this function. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_local_size()** returns 1. For **clEnqueueTask**, this always returns 1.
size_t **get_local_id** (uint *dimindx*)	Returns the unique local work-item ID i.e. a work-item within a specific work-group for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_local_id()** returns 0. For **clEnqueueTask**, this returns 0.

OpenCL

OpenCL Language : fonctions prédéfinies (work-group ID)

size_t **get_num_groups** (uint *dimindx*)	Returns the number of work-groups that will execute a kernel for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values of *dimindx*, **get_num_groups ()** returns 1. For **clEnqueueTask**, this always returns 1.
size_t **get_group_id** (uint *dimindx*)	**get_group_id** returns the work-group ID which is a number from 0 .. **get_num_groups**(*dimindx*) – 1. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values, **get_group_id()** returns 0. For **clEnqueueTask**, this returns 0.
size_t **get_global_offset** (uint *dimindx*)	**get_global_offset** returns the offset values specified in *global_work_offset* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() – 1. For other values, **get_global_offset()** returns 0. For **clEnqueueTask**, this returns 0.

OpenCL

OpenCL Language : fonctions prédéfinies (barrière)

void **barrier** (cl_mem_fence_flags *flags*)	All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the **barrier**. This function must be encountered by all work-items in a work-group executing the kernel. If **barrier** is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the **barrier**. If **barrier** is inside a loop, all work-items must execute the **barrier** for each iteration of the loop before any are allowed to continue execution beyond the **barrier**. The **barrier** function also queues a memory fence (reads and writes) to ensure correct ordering of memory operations to local or global memory.

OpenCL

OpenCL Language : fonctions prédéfinies (barrière)

	The *flags* argument specifies the memory address space and can be set to a combination of the following literal values. CLK_LOCAL_MEM_FENCE - The **barrier** function will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory. CLK_GLOBAL_MEM_FENCE – The **barrier** function will queue a memory fence to ensure correct ordering of memory operations to global memory. This can be useful when work-items, for example, write to buffer or image objects and then want to read the updated data.

OpenCL

Exemple : addition de matrices (kernel)

```
40 __kernel void matrix_add(float * a, float * b, float * c) {
41     int x = get_global_id(0);
42     int y = get_global_id(1);
43     int n = get_global_size(0);
44
45     int off = y*n + x;
46
47     c[off] = a[off] + b[off];
48 }
```

OpenCL

Exemple : addition de matrices (hôte)

```
3   #define N 256*n
4   size_t size = N * N * sizeof(float);
5
6   float matA[N*N] = ...
7   float matB[N*N] = ...
8   float matC[N*N];
9   cl_event ev_writeA, ev_writeB, ev_ker, ev_readC;
10
11  cl_context ctx = clCreateContext(...);
12  cl_command_queue cq = clCreateCommandQueue(...);
13
14  cl_program prg = clCreateProgramWithSource(ctx, ...);
15  clBuildProgram(prg, ...);
16
17  cl_kernel ker = clCreateKernel(prg, "matrix_add", NULL);
18
19  cl_mem bufA = clCreateBuffer(ctx, CL_MEM_READ_ONLY, size, NULL, NULL);
20  cl_mem bufB = clCreateBuffer(ctx, CL_MEM_READ_ONLY, size, NULL, NULL);
21  cl_mem bufC = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY, size, NULL, NULL);
22
23  clEnqueueWriteBuffer(cq, bufA, 0, 0, size, matA, 0, NULL, &ev_writeA);
24  clEnqueueWriteBuffer(cq, bufB, 0, 0, size, matB, 0, NULL, &ev_writeB);
25
26  clSetKernelArg(ker, 0, size, bufA);
27  clSetKernelArg(ker, 0, size, bufB);
28  clSetKernelArg(ker, 0, size, bufC);
29
30  size_t globalDim[] = {N, N};
31  size_t localDim[] = {256, 1};
32  cl_event deps[] = {ev_writeA, ev_writeB};
33  clEnqueueNDRangeKernel(cq, ker, 2, NULL, globalDim, localDim, 2, deps, &ev_ker);
34
35  clEnqueueReadBuffer(cq, bufC, 0, 0, size, matC, 1, &ev_ker, &ev_readC);
36
37  clWaitForEvents(1, &ev_readC);
38
```

Lignes directrices

1. Introduction
   - Historique
   - Les accélérateurs
   - Programmation des accélérateurs
2. Quelques exemples d'accélérateurs
   - IBM CELL BroadBand Engine
   - Cartes graphiques (GPU)
   - Architectures hybrides GPU-CPU
3. Modèles de programmation
   - Bas niveau : graphe de commandes
   - Abstractions

Rappel : graphe de commandes

Ce que l'application doit gérer

- Gestion mémoire
  - Allocation d'espaces dans les mémoires
  - Transferts de données entre les mémoires
    - Programmation des contrôleurs DMA
  - Libération des espaces alloués
- Gestion des codes
  - Compilation pour les différents types de cœurs
  - Transfert/chargement des binaires
  - Placement et ordonnancement

Rappel : graphe de commandes
Inconvénients

- L'application doit gérer des choses qui incombaient à l'OS

	Multi-cœur / NUMA	Accélérateurs (graphe de commandes)
Espace(s) d'adressage	Unique	Un par accélérateur + hôte
Code(s) pour une fonction	Unique	Un par type d'architecture
Placement	Processus/Threads : automatique	Kernels : manuel

- Comment abstraire au-dessus des frameworks bas niveau ?

Rappel : graphe de commandes

Inconvénients

- L'application doit gérer des choses qui incombaient à l'OS

	Multi-cœur / NUMA	Accélérateurs (graphe de commandes)
Espace(s) d'adressage	Unique	Un par accélérateur + hôte
Code(s) pour une fonction	Unique	Un par type d'architecture
Placement	Processus/Threads : automatique	Kernels : manuel

- Comment abstraire au-dessus des frameworks bas niveau ?

Implémentations

1. Bibliothèques / supports exécutifs pour des langages existants
2. Extensions de langages existants (e.g. pragma)
3. Nouveaux langages

- Exemples : OmpSS/StarSS, OpenACC, HMPP, StarPU, SOCL, Microsoft Accelerator, REPA, LibSH/Rapidmind/Intel Ct/Intel ArBB, ZPL/Chapel, x10, Fortress, PetaBricks, SaC, ViperVM...

Implémentations

1. Bibliothèques / supports exécutifs pour des langages existants
2. Extensions de langages existants (e.g. pragma)
3. Nouveaux langages

- Exemples : OmpSS/StarSS, OpenACC, HMPP, StarPU, SOCL, Microsoft Accelerator, REPA, LibSH/Rapidmind/Intel Ct/Intel ArBB, ZPL/Chapel, x10, Fortress, PetaBricks, SaC, ViperVM...

Implémentations

1. Bibliothèques / supports exécutifs pour des langages existants
2. Extensions de langages existants (e.g. pragma)
3. Nouveaux langages

- Exemples : OmpSS/StarSS, OpenACC, HMPP, StarPU, SOCL, Microsoft Accelerator, REPA, LibSH/Rapidmind/Intel Ct/Intel ArBB, ZPL/Chapel, x10, Fortress, PetaBricks, SaC, ViperVM...

Implémentations

1. Bibliothèques / supports exécutifs pour des langages existants
2. Extensions de langages existants (e.g. pragma)
3. Nouveaux langages

- Exemples : OmpSS/StarSS, OpenACC, HMPP, StarPU, SOCL, Microsoft Accelerator, REPA, LibSH/Rapidmind/Intel Ct/Intel ArBB, ZPL/Chapel, x10, Fortress, PetaBricks, SaC, ViperVM...

Mémoire virtuelle

Principe

- Les kernels manipulent la mémoire par l'intermédiaire des buffers
  - Pas d'accès en dehors des buffers (e.g. par manipulation de pointeurs)
- Les buffers sont passés explicitement à chaque kernel en paramètres
  - Pas de variable globale

Idée

Disposer d'une mémoire virtuelle contenant tous les buffers. Le support exécutif doit alors gérer les transferts de données entre les mémoires physiques.

Mémoire virtuelle
Principe

- Les kernels manipulent la mémoire par l'intermédiaire des buffers
  - Pas d'accès en dehors des buffers (e.g. par manipulation de pointeurs)
- Les buffers sont passés explicitement à chaque kernel en paramètres
  - Pas de variable globale

Idée

Disposer d'une mémoire virtuelle contenant tous les buffers. Le support exécutif doit alors gérer les transferts de données entre les mémoires physiques.

Mémoire virtuelle
Principe

- Les kernels manipulent la mémoire par l'intermédiaire des buffers
  - Pas d'accès en dehors des buffers (e.g. par manipulation de pointeurs)
- Les buffers sont passés explicitement à chaque kernel en paramètres
  - Pas de variable globale

Idée

Disposer d'une mémoire virtuelle contenant tous les buffers. Le support exécutif doit alors gérer les transferts de données entre les mémoires physiques.

Mémoire virtuelle

Ce que le support exécutif doit prendre en charge

- Transfert des données en mémoire avant l'exécution d'un kernel
- Recouvrement transferts/exécution de code
- Pré-chargement (*prefetching*)
- Gestion des évictions lorsqu'il faut libérer de l'espace mémoire
- Conversions pour l'endianness
  - Nécessite d'avoir des buffers typés
- Choix des alignements
  - Dépend de l'architecture et des codes qui vont utiliser le buffer

Mémoire virtuelle

Exemple StarPU : enregistrement d'une matrice

— Function: void **starpu_matrix_data_register** (*starpu_data_handle_t *handle, uint32_t home_node, uintptr_t ptr, uint32_t ld, uint32_t nx, uint32_t ny, size_t elemsize*)

Register the *nxxny* 2D matrix of *elemsize*-byte elements pointed by *ptr* and initialize *handle* to represent it. *ld* specifies the number of elements between rows. a value greater than *nx* adds padding, which can be useful for alignment purposes.

```
float *matrix;
starpu_data_handle_t matrix_handle;
matrix = (float*)malloc(width * height * sizeof(float));
starpu_matrix_data_register(&matrix_handle, 0, (uintptr_t)matrix,
                            width, width, height, sizeof(float));
```

Kernels multi-architecture

Constat & Idée

- Pour un calcul, potentiellement un kernel différent par architecture
- On choisit le kernel en fonction de l'architecture sur laquelle on veut exécuter le calcul
- Les différents kernels ont la même interface
  - Mêmes paramètres
  - Même comportement mathématique

Idée

Regrouper ces différents kernels au sein d'un même « meta-kernel »

Kernels multi-architecture

Constat & Idée

- Pour un calcul, potentiellement un kernel différent par architecture
- On choisit le kernel en fonction de l'architecture sur laquelle on veut exécuter le calcul
- Les différents kernels ont la même interface
  - Mêmes paramètres
  - Même comportement mathématique

Idée

Regrouper ces différents kernels au sein d'un même « meta-kernel »

Kernels multi-architecture

Implémentations possibles

1. Un code par architecture :
   1. MetaKernel : : HashMap Architecture Kernel
2. Un code « générique » pouvant cibler différents architectures
   1. MetaKernel : : Source
3. Les deux à la fois
   1. si pas de code spécifique à l'architecture, on prend le code générique
   2. MetaKernel : : (Source, HashMap Architecture Kernel)
4. Généralisation avec plusieurs codes génériques
   1. Plusieurs algorithmes possibles
   2. Le support exécutif doit choisir
   3. MetaKernel : : ([Source], HashMap Architecture Kernel)

Kernels multi-architecture
Implémentations possibles

1. Un code par architecture :
   1. MetaKernel : : HashMap Architecture Kernel
2. Un code « générique » pouvant cibler différents architectures
   1. MetaKernel : : Source
3. Les deux à la fois
   1. si pas de code spécifique à l'architecture, on prend le code générique
   2. MetaKernel : : (Source, HashMap Architecture Kernel)
4. Généralisation avec plusieurs codes génériques
   1. Plusieurs algorithmes possibles
   2. Le support exécutif doit choisir
   3. MetaKernel : : ([Source], HashMap Architecture Kernel)

Kernels multi-architecture

Implémentations possibles

1. Un code par architecture :
    1. MetaKernel : : HashMap Architecture Kernel
2. Un code « générique » pouvant cibler différents architectures
    1. MetaKernel : : Source
3. Les deux à la fois
    1. si pas de code spécifique à l'architecture, on prend le code générique
    2. MetaKernel : : (Source, HashMap Architecture Kernel)
4. Généralisation avec plusieurs codes génériques
    1. Plusieurs algorithmes possibles
    2. Le support exécutif doit choisir
    3. MetaKernel : : ([Source], HashMap Architecture Kernel)

Kernels multi-architecture

Implémentations possibles

1. Un code par architecture :
   1. MetaKernel : : HashMap Architecture Kernel
2. Un code « générique » pouvant cibler différents architectures
   1. MetaKernel : : Source
3. Les deux à la fois
   1. si pas de code spécifique à l'architecture, on prend le code générique
   2. MetaKernel : : (Source, HashMap Architecture Kernel)
4. Généralisation avec plusieurs codes génériques
   1. Plusieurs algorithmes possibles
   2. Le support exécutif doit choisir
   3. MetaKernel : : ([Source], HashMap Architecture Kernel)

Kernels multi-architecture

Ce que le support exécutif doit prendre en charge

- Choix du kernel lorsqu'un calcul doit être effectué sur une architecture
  - i.e. lorsqu'un meta-kernel est ordonnancé sur une architecture
- Si plusieurs kernels disponibles pour une même architecture : stratégie(s) de sélection
  - Benchmarking
  - Fonctions de prédiction de performance en fonction des paramètres d'entrée

Kernels multi-architecture

Exemple : code générique avec OpenACC

```
1  void BlackScholes( float *call_result, float *put_result,
   float *stock_price, float *option_strike, float
   *option_years, float Riskfree, float Volatility, int
   nb_opt )
2
3  #pragma acc kernels copyin(option_strike[0:nb_opt],
   stock_price[0:nb_opt], option_years[0:nb_opt])
   copyout(call_result[0:nb_opt], put_result[0:nb_opt])
4  {
5    int opt = 0;
6    float sqrtT, expRT, K;
7    float d1, d2, CNDD1, CNDD2;
8
9    #pragma acc loop independent
10   for(opt = 0; opt < nb_opt; opt++) {
11     sqrtT = sqrtf(option_years[opt]);
12     d1 = (logf(stock_price[opt] / option_strike[opt]) +
   (Riskfree + 0.5f * Volatility * Volatility) *
   option_years[opt]) / (Volatility * sqrtT);
13     d2 = d1 - Volatility * sqrtT;
14     K = 1.0f / (1.0f + 0.2316419f * fabsf(d1));
15     CNDD1 = RSQRT2PI * expf(- 0.5f * d1 * d1) * (K * (A1 +
   K * (A2 + K * (A3 + K * (A4 + K * A5)))));
16     K = 1.0f / (1.0f + 0.2316419f * fabsf(d2));
17     CNDD2 = RSQRT2PI * expf(- 0.5f * d2 * d2) * (K * (A1 +
   K * (A2 + K * (A3 + K * (A4 + K * A5)))));
18
19     //Compute Call and Put simultaneously
20     expRT = expf(- Riskfree * option_years[opt]);
21     call_result[opt] = stock_price[opt] * CNDD1 -
   option_strike[opt] * expRT * CNDD2;
22     put_result[opt]  = option_strike[opt] * expRT * (1.0f
   - CNDD2) - stock_price[opt] * (1.0f - CNDD1);
23     }
24   }//end of Kernels region
25 }
```

Kernels multi-architecture

Exemple : « codelet » SGEMM avec StarPU

```
void sgemm_cpu_func(void *descr[], void *cl_arg) {
     int transA, transB, M, N, K, LDA, LDB, LDC;
     float alpha, beta, *A, *B, *C;

     A = STARPU_MATRIX_GET_PTR(descr[0]);
     B = STARPU_MATRIX_GET_PTR(descr[1]);
     C = STARPU_MATRIX_GET_PTR(descr[2]);

     starpu_unpack_cl_args(cl_arg, &transA, &transB, &M,
                           &N, &K, &alpha, &LDA, &LDB, &beta, &LDC);

     sgemm(CblasColMajor, transA, transB, M, N, K,
           alpha, A, LDA, B, LDB, beta, C, LDC);
}

void sgemm_cuda_func(void *descr[], void *cl_arg) {
     int transA, transB, M, N, K, LDA, LDB, LDC;
     float alpha, beta, *A, *B, *C;

     A = STARPU_MATRIX_GET_PTR(descr[0]);
     B = STARPU_MATRIX_GET_PTR(descr[1]);
     C = STARPU_MATRIX_GET_PTR(descr[2]);

     starpu_unpack_cl_args(cl_arg, &transA, &transB, &M,
                           &N, &K, &alpha, &LDA, &LDB, &beta, &LDC);

     cublasSgemm(magma_const[transA][0], magma_const[transB][0],
                 M, N, K, alpha, A, LDA, B, LDB, beta, C, LDC);
     cudaThreadSynchronize();
}

struct starpu_perfmodel_t cl_sgemm_model = {
     .type   = STARPU_HISTORY_BASED,
     .symbol = "sgemm"
};

starpu_codelet sgemm_codelet = {
     .where     = STARPU_CPU|STARPU_CUDA, // who may execute?
     .cpu_func  = sgemm_cpu_func, // CPU implementation
     .cuda_func = sgemm_cuda_func, // CUDA implementation
     .nbuffers  = 3, // number of handles accessed by the task
     .model     = &cl_sgemm_model // performance model (optional)
};
```

Kernels multi-architecture

Exemple : « transform » SGEMM avec PetaBricks

```
1  transform MatrixMultiply
2  from A[c,h], B[w,c]
3  to AB[w,h]
4  {
5    // Base case, compute a single element
6    to(AB.cell(x,y) out)
7    from(A.row(y) a, B.column(x) b) {
8      out = dot(a,b);
9    }
10
11   // Recursively decompose in c
12   to(AB ab)
13   from(A.region(0,   0, c/2, h   ) a1,
14        A.region(c/2, 0, c,   h   ) a2,
15        B.region(0,   0, w,   c/2) b1,
16        B.region(0, c/2, w,   c   ) b2) {
17     ab = MatrixAdd(MatrixMultiply(a1, b1),
18                    MatrixMultiply(a2, b2));
19   }
21   // Recursively decompose in w
22   to(AB.region(0,   0, w/2, h   ) ab1,
23      AB.region(w/2, 0, w,   h   ) ab2)
24   from( A a,
25         B.region(0,   0, w/2, c   ) b1,
26         B.region(w/2, 0, w,   c   ) b2) {
27     ab1 = MatrixMultiply(a, b1);
28     ab2 = MatrixMultiply(a, b2);
29   }
30
31   // Recursively decompose in h
32   to(AB.region(0, 0,   w, h/2) ab1,
33      AB.region(0, h/2, w, h   ) ab2)
34   from(A.region(0,   0, c, h/2) a1,
35        A.region(0, h/2, c, h   ) a2,
36        B b) {
37     ab1=MatrixMultiply(a1, b);
38     ab2=MatrixMultiply(a2, b);
39   }
40 }
```

Ordonnanceurs de kernels
Constat & Idée

- Les applications doivent déterminer où placer les kernels
- Portabilité des performances très difficile à obtenir
  - Architectures très variables (unités de calcul, interconnexions...)
- À réfaire pour chaque application

Idée

Factoriser le code d'ordonnancement des kernels dans une bibliothèque / un support exécutif.

Ordonnanceurs de kernels
Constat & Idée

- Les applications doivent déterminer où placer les kernels
- Portabilité des performances très difficile à obtenir
  - Architectures très variables (unités de calcul, interconnexions...)
- À réfaire pour chaque application

Idée

Factoriser le code d'ordonnancement des kernels dans une bibliothèque / un support exécutif.

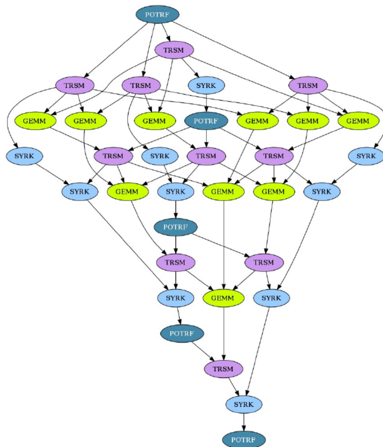
Ordonnanceurs de kernels

Ce que le support exécutif doit prendre en charge

- Support d'un graphe de tâches en entrée
  - i.e. graphe de meta-kernels paramétrés
- Ordonnancement des tâches sur les unités disponibles
  - Parmi les unités qui peuvent les exécuter
- Forte intéraction entre l'ordonnancement des tâches et la mémoire virtuelle
  - Pour limiter les transferts
  - Pour anticiper les transferts
- Stratégies d'ordonnancements
  - Estimation temps de calcul pour chaque tâche pour chaque unité
  - Temps de transfert des données

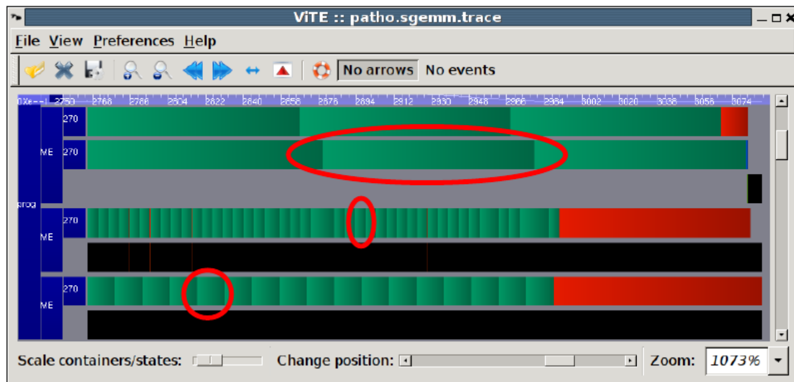
Ordonnanceurs de kernels

Exemple : graphe de tâches (Cholesky)



Ordonnanceurs de kernels

Exemple ordonnanceur : mauvais placement de tâches !



Ordonnanceurs de kernels

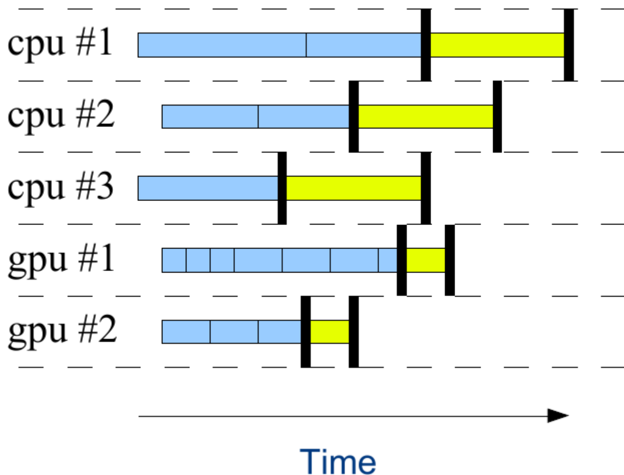
Exemple StarPU : historique des performances

```
(~)(shenry@hannibal:pts/2)
% starpu_perfmodel_display -l
file: <SGEMM.hannibal>
file: <FLOATMATRIX_TRANSPOSE.hannibal>
file: <FLOATMATRIX_SCALE.hannibal>
file: <FLOATMATRIX_SUB.hannibal>
file: <FLOATMATRIX_STRSM.hannibal>
file: <FLOATMATRIX_SPOTRF.hannibal>
file: <FLOATMATRIX_DUPLICATE.hannibal>
file: <FLOATMATRIX_SUBMATRIX.hannibal>
file: <FLOATMATRIX_ADD.hannibal>
file: <FLOATMATRIX_SET.hannibal>
file: <FLOATMATRIX_STRMM.hannibal>
```

```
(~)(shenry@hannibal:pts/2)
% starpu_perfmodel_display -s FLOATMATRIX_SPOTRF
performance model for cpu_impl_0
# hash      size      mean           stddev         n
3e921964    65536     4.650055e+04   5.912348e+04   10
914f3bef    1048576   1.571605e+04   2.612296e+03   58
aa6d4ef7    4194304   7.764298e+04   1.892561e+03   3
performance model for cuda_0_impl_0
# hash      size      mean           stddev         n
914f3bef    1048576   8.984344e+04   4.617692e+04   10
3e921964    65536     1.649468e+04   3.420279e+03   12
37274d3d    1024      8.849191e+03   8.969578e+03   3
aa6d4ef7    4194304   2.208919e+05   4.982061e+04   9
performance model for cuda_1_impl_0
# hash      size      mean           stddev         n
3e921964    65536     1.152254e+04   1.729861e+03   40
914f3bef    1048576   1.313580e+05   5.392499e+04   10
aa6d4ef7    4194304   2.221325e+05   1.807337e+05   3
performance model for cuda_2_impl_0
# hash      size      mean           stddev         n
aa6d4ef7    4194304   9.265122e+04   0.000000e+00   1
3e921964    65536     8.141800e+04   1.507312e+05   10
914f3bef    1048576   8.459304e+04   7.537374e+04   10
```

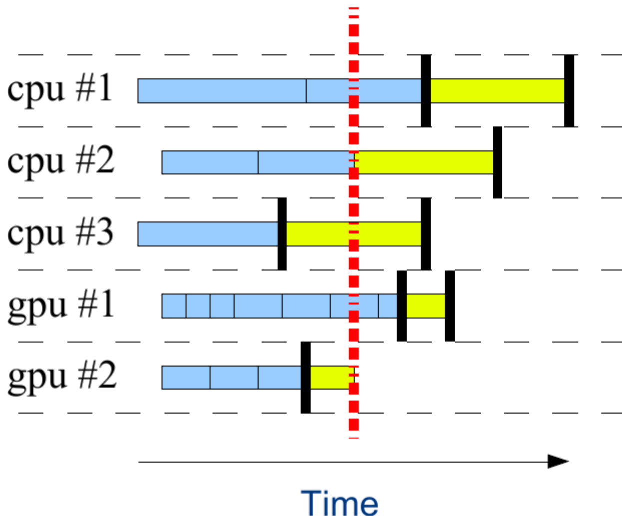
Ordonnanceurs de kernels

Exemple StarPU : ordonnanceur HEFT (Heterogeneous Earliest Finish Time) (1/3)



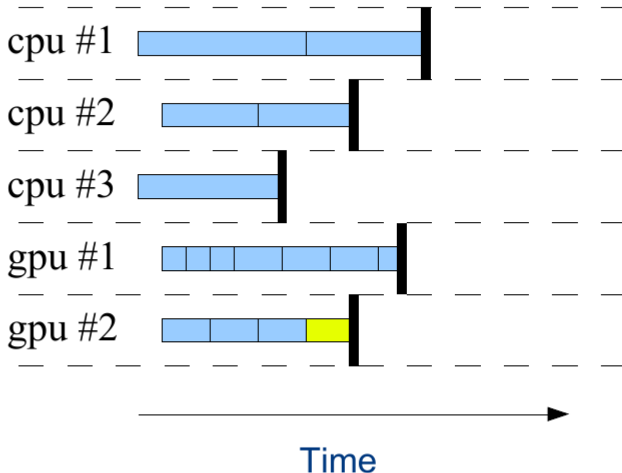
Ordonnanceurs de kernels

Exemple StarPU : ordonnanceur HEFT (Heterogeneous Earliest Finish Time) (2/3)



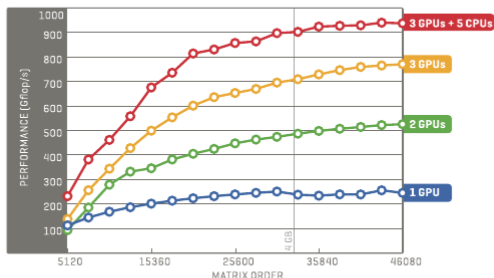
Ordonnanceurs de kernels

Exemple StarPU : ordonnanceur HEFT (Heterogeneous Earliest Finish Time) (3/3)



Ordonnanceurs de kernels

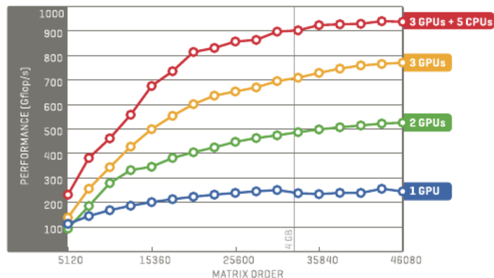
Exemple factorisation de Cholesky : prise en compte de l'hétérogénéité (StarPU)



- 3 GPUs : 780 Gflops ; 3 GPUs + 5 CPUs : 900 Gflops (+120)
- GPU : sgemm 333 Gflop/s, spotrf 56 Gflop/s
- 5 CPUs : sgemm < 100 Gflop/s, spotrf ? Gflop/s
- 80% des spotrf sur les CPUs

Ordonnanceurs de kernels

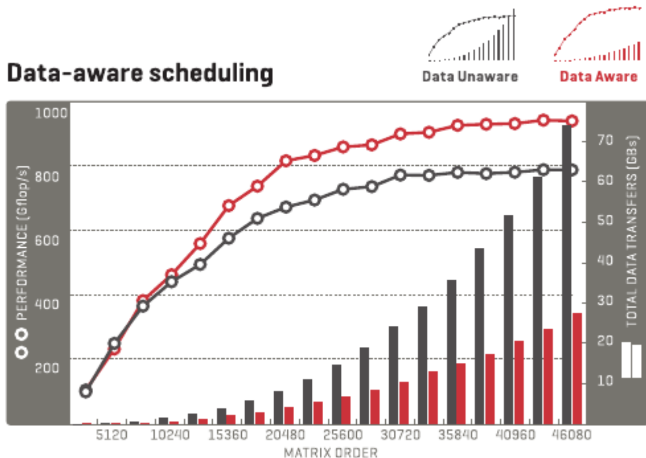
Exemple factorisation de Cholesky : prise en compte de l'hétérogénéité (StarPU)



- 3 GPUs : 780 Gflops ; 3 GPUs + 5 CPUs : 900 Gflops (+120)
- GPU : sgemm 333 Gflop/s, spotrf 56 Gflop/s
- 5 CPUs : sgemm < 100 Gflop/s, spotrf ? Gflop/s
- 80% des spotrf sur les CPUs

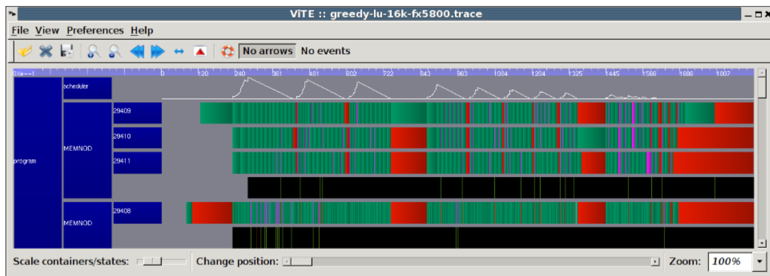
Ordonnanceurs de kernels

Exemple factorisation de Cholesky : prise en compte des données (StarPU)



Ordonnanceurs de kernels

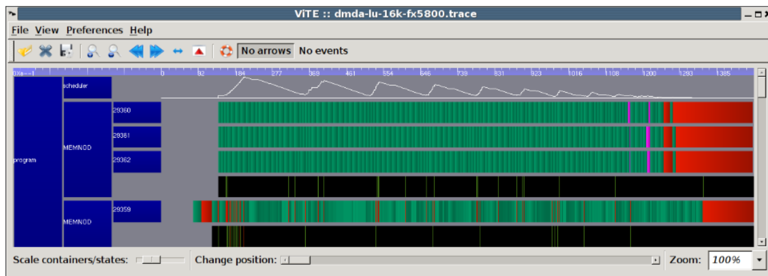
Exemple factorisation LU : ordonnanceur « Greedy » (StarPU)



- Nombre de tâches à exécuter parfois à 0 (courbe blanche)
- Performance : 175 Gflops

Ordonnanceurs de kernels

Exemple factorisation LU : ordonnanceur « DMDA » (StarPU)



- Nombre de tâches à exécuter toujours supérieur à 0
- Performance : 239 Gflops

Graphe de dépendances implicite
Constat & Idée

- Les dépendances entre tâches sont souvent dues aux données
- Dépendances :
  - lecture après écriture
  - écriture après lecture ou écriture
- On peut connaitre les types d'accès (RO, RW) de chaque kernel à chacun de leurs paramètres

Idée

Inférer automatiquement les dépendances entre les tâches à partir du type d'accès aux données qu'elles font

Graphe de dépendances implicite

Constat & Idée

- Les dépendances entre tâches sont souvent dues aux données
- Dépendances :
  - lecture après écriture
  - écriture après lecture ou écriture
- On peut connaitre les types d'accès (RO, RW) de chaque kernel à chacun de leurs paramètres

Idée

Inférer automatiquement les dépendances entre les tâches à partir du type d'accès aux données qu'elles font

Graphe de dépendances implicite

Ce que le support exécutif doit prendre en charge

- Pour chaque donnée D, il doit se souvenir :
  - de la dernière tâche soumise en écriture : $Task_w(D)$
  - des tâches soumises en lecture depuis $Task_W(D)$: $Tasks_R(D)$
- Lorsqu'une nouvelle tâche T est soumise, pour chacun de ses paramètres P, si l'accès est :
  - en lecture :
    - dépendance sur $Task_W(P)$ et ajout à la liste $Tasks_R(P)$
  - en écriture :
    - dépendance sur $Task_W(P)$ et sur toutes les tâches de $Tasks_R(P)$
    - mises à jour : $Task_W(P) = T$ et $Tasks_R(P) = Nil$

Graphe de dépendances implicite

Conséquences

- L'ordre de soumission des tâches est important
  - Similaire à une exécution out-of-order d'un programme impératif

Graphe de dépendances implicite

Exemple : Cholesky avec StarPU (1/2)

```
1  void hybrid_cholesky(starpu_data_handle **Ahandles,
2                       int M, int N, int Mt, int Nt, int Mb)
3  {
4   int lower = Lower;     int upper = Upper; int right = Right;
5   int notrans = NoTrans; int conjtrans = ConjTrans;
6   int nonunit = NonUnit; float one = 1.0f; float mone = -1.0f;
7
8   int k, m, n, temp;
9   for (k = 0; k < Nt; k++)
10  {
11   temp = k == Mt-1 ? M-k*Mb : Mb ;
12   starpu_Insert_Task(spotrf_codelet,
13     VALUE, &lower, sizeof(int), VALUE, &temp, sizeof(int),
14     INOUT, Ahandles[k][k],      VALUE, &Mb, sizeof(int), 0);
15
16   for (m = k+1; m < Nt; m++)
17   {
18    temp = m == Mt-1 ? M-m*Mb : Mb ;
19    starpu_Insert_Task(strsm_codelet,
20      VALUE, &right, sizeof(int),     VALUE, &lower, sizeof(int),
21      VALUE, &conjtrans, sizeof(int), VALUE, &nonunit, sizeof(int),
22      VALUE, &temp, sizeof(int),      VALUE, &Mb, sizeof(int),
23      VALUE, &one, sizeof(float),     INPUT, Ahandles[k][k],
24      VALUE, &Mb, sizeof(int),        INOUT, Ahandles[m][k],
25      VALUE, &Mb, sizeof(int),        0);
26   }
```

Graphe de dépendances implicite

Exemple : Cholesky avec StarPU (2/2)

```
28     for (m = k+1; m < Nt; m++)
29     {
30      temp = m == Mt-1 ? M-m*Mb : Mb;
31      for (n = k+1; n < m; n++)
32      {
33       starpu_Insert_Task(sgemm_codelet,
34         VALUE, &notrans, sizeof(notrans),
35         VALUE, &conjtrans, sizeof(conjtrans),
36         VALUE, &temp, sizeof(int),    VALUE, &Mb, sizeof(int),
37         VALUE, &Mb, sizeof(int),      VALUE, &mone, sizeof(float),
38         INPUT, Ahandles[m][k],        VALUE, &Mb, sizeof(int),
39         INPUT, Ahandles[n][k],        VALUE, &Mb, sizeof(int),
40         VALUE, &one, sizeof(one),     INOUT, Ahandles[m][n],
41         VALUE, &Mb, sizeof(int),      0);
42      }
43
44      starpu_Insert_Task(ssyrk_codelet,
45        VALUE, &lower, sizeof(int),   VALUE, &notrans, sizeof(int),
46        VALUE, &temp, sizeof(int),    VALUE, &Mb, sizeof(int),
47        VALUE, &mone, sizeof(float), INPUT, Ahandles[m][k],
48        VALUE, &Mb, sizeof(int),      VALUE, &one, sizeof(float),
49        INOUT, Ahandles[m][m],        VALUE, &Mb, sizeof(int), 0);
50     }
51    }
52
53    starpu_task_wait_for_all();
54   }
```

Libération paresseuse des données

Constat & Idée

- Pour libérer une donnée, il faut attendre que toutes les tâches l'utilisant aient terminé
    - Attente explicite de la terminaison des tâches
- Risque d'oubli d'une tâche
    - Libération d'une donnée alors qu'elle est utilisée
- Risque d'oubli de faire la libération
    - Le support exécutif gère des données inutilisées et peut occasionner des transferts superflus

Idée

Lorsque toutes les tâches utilisant une donnée ont été soumises, on indique au support exécutif qu'elle ne sera pas utilisée par les prochaines tâches qui seront soumises.

Libération paresseuse des données

Constat & Idée

- Pour libérer une donnée, il faut attendre que toutes les tâches l'utilisant aient terminé
  - Attente explicite de la terminaison des tâches
- Risque d'oubli d'une tâche
  - Libération d'une donnée alors qu'elle est utilisée
- Risque d'oubli de faire la libération
  - Le support exécutif gère des données inutilisées et peut occasionner des transferts superflus

Idée

Lorsque toutes les tâches utilisant une donnée ont été soumises, on indique au support exécutif qu'elle ne sera pas utilisée par les prochaines tâches qui seront soumises.

Libération paresseuse des données

Ce que le support exécutif doit prendre en charge

- Pour chaque donnée, maintien du nombre de tâches non exécutées ou en cours d'exécution qui utilisent la donnée
    - Valeur initiale à 1 tant que l'hôte n'a pas soumis toutes les tâches
    - Lorsque l'une de ces tâches termine, le nombre est décrémenté
    - Lorsque le nombre arrive à 0, le support exécutif libère la donnée

Libération paresseuse des données

Exemple StarPU

```
1
2 starpu_vector_data_register(&handle, -1, 0, n, sizeof(float));
3
4 starpu_insert_task(&produce_data, STARPU_W, handle, 0);
5 starpu_insert_task(&compute_data, STARPU_RW, handle, 0);
6 starpu_insert_task(&summarize_data, STARPU_R, handle, STARPU_W, result_handle, 0);
7
8 starpu_data_unregister_submit(handle);
```

Libération paresseuse des données

Support des ramasses-miettes

- Chaque donnée est identifiée dans le code hôte par un objet
- Lorsqu'il n'y a plus aucune référence vers cet objet dans le code, l'objet peut être collecté
- Lors de la collection d'objets de ce type, on indique au support exécutif que la donnée peut être libérée de façon paresseuse
    - Il n'y a plus de référence à la donnée dans le code hôte donc il n'y aura pas de nouvelle tâche soumise utilisant la donnée

Libération paresseuse des données

Exemple HaskellPU : support des ramasse-miettes

```
type Handle = ForeignPtr ()
type UnsafeHandle = Ptr ()

foreign import ccall unsafe "starpu_matrix_data_register" matrixRegister ::
   Ptr UnsafeHandle -> Int -> Ptr () -> Word -> Word -> Word -> CSize -> IO ()

foreign import ccall unsafe "&starpu_data_unregister_submit" p_dataUnregisterLazy ::
   FunPtr(UnsafeHandle -> IO ())

floatMatrixRegister :: Ptr () -> Int -> Word -> Word -> Word -> IO Handle
floatMatrixRegister ptr node width height ld = alloca $ \handle -> do
  matrixRegister handle node ptr nld nx ny 4
  hdl <- peek handle
  newForeignPtr p_dataUnregisterLazy hdl
  where
    nld = fromIntegral ld
    nx = fromIntegral height
    ny = fromIntegral width
```

Allocation paresseuse
Constat & Idée

- Les allocations dans la mémoire virtuelle se font de façon synchrone
  - i.e. tous les buffers sont alloués avant la soumission du graphe de tâches
- Il faut prévoir à l'avance les dimensions des données

Idée

Permettre l'allocation asynchrone des buffers

Allocation paresseuse
Constat & Idée

- Les allocations dans la mémoire virtuelle se font de façon synchrone
  - i.e. tous les buffers sont alloués avant la soumission du graphe de tâches
- Il faut prévoir à l'avance les dimensions des données

Idée

Permettre l'allocation asynchrone des buffers

Allocation paresseuse
Méthodes

1. Allouer uniquement un descripteur de la donnée. L'espace mémoire sera effectivement alloué lorsqu'un kernel voudra accéder à la donnée.
   1. Ne résoud pas le problème d'allocation de buffers dont les dimensions dépendent d'un calcul précédent
2. Inclure le code d'allocation des buffers en output dans le meta-kernel
   1. Les allocations se font en fonction des paramètres du meta-kernel
   2. Les buffers alloués sont renvoyés au moment de la soumission du kernel

Graphe de tâches fonctionnel

Constat & Idée

- Supposons qu'on interdise la modification du contenu d'un buffer déjà initialisé
  - On peut modifier le contenu d'une copie d'un buffer cependant
- Un programme fonctionnel est un graphe de *super-combinators*
  - i.e. graphe de fonctions qui ne dépendent que de leurs paramètres
  - similaire à un graphe de tâches

Idée

Exprimer le graphe de tâches en utilisant un langage fonctionnel

Graphe de tâches fonctionnel
Constat & Idée

- Supposons qu'on interdise la modification du contenu d'un buffer déjà initialisé
  - On peut modifier le contenu d'une copie d'un buffer cependant
- Un programme fonctionnel est un graphe de *super-combinators*
  - i.e. graphe de fonctions qui ne dépendent que de leurs paramètres
  - similaire à un graphe de tâches

Idée

Exprimer le graphe de tâches en utilisant un langage fonctionnel

Graphe de tâches fonctionnel

Ce que le support exécutif doit prendre en charge

- Interprétation du programme fonctionnel
  - Réduction du graphe en parallèle
- Optimisations pour limiter les copies de données superflues
  - Si un kernel est le dernier à modifier une donnée, pas besoin de la dupliquer

Graphe de tâches fonctionnel

Exemple ViperVM : multiplication de matrices par blocs

```
2 -- Multiplication de matrices par blocs de dimension n x n
3 matrixMulBlocks n a b = unsplit $ crossWith dot' (rows a') (columns b')
4   where
5     a' = split n n a
6     b' = split n n b
7     dot' x y = reduce matrixAdd $ zipWith matrixMul x y
```

Opérateurs « Data-Parallel »

Constat & Idée

- Kernels difficiles à écrire notamment à cause de la gestion des indices des tableaux et des synchronisations.
- On sait comment programmer efficacement certains patterns de codes (opérations « data-parallel »)
  - homomorphism (i.e. *map*)
  - réductions (e.g. *fold*, *scan*, *reduce*)

Idée

Écrire les kernels sous forme de compositions d'opérateurs data-parallel

Opérateurs « Data-Parallel »

Constat & Idée

- Kernels difficiles à écrire notamment à cause de la gestion des indices des tableaux et des synchronisations.
- On sait comment programmer efficacement certains patterns de codes (opérations « data-parallel »)
  - homomorphism (i.e. *map*)
  - réductions (e.g. *fold*, *scan*, *reduce*)

Idée

Écrire les kernels sous forme de compositions d'opérateurs data-parallel

Opérateurs « Data-Parallel »

Exemple Microsoft Accelerator (1/3) : opérations par élément

Operation	Definition
Add	$R_{i,j} = A_{i,j} + B_{i,j}$
Subtract	$R_{i,j} = A_{i,j} - B_{i,j}$
Multiply	$R_{i,j} = A_{i,j} \times B_{i,j}$
Divide	$R_{i,j} = A_{i,j} \div B_{i,j}$
Max	$R_{i,j} = \max\left(A_{i,j}, B_{i,j}\right)$
Min	$R_{i,j} = \min\left(A_{i,j}, B_{i,j}\right)$
Select	$R_{i,j} = \begin{cases} B_{i,j} & \text{if } A_{i,j} > 0 \\ C_{i,j} & \text{otherwise} \end{cases}$
Cos	$R_{i,j} = \cos A_{i,j}$
Sqrt	$R_{i,j} = \sqrt{A_{i,j}}$
And	$R_{i,j} = A_{i,j} \wedge B_{i,j}$
Or	$R_{i,j} = A_{i,j} \vee B_{i,j}$
CompareEqual	$R_{i,j} = \begin{cases} \text{true} & \text{if } A_{i,j} = B_{i,j} \\ \text{false} & \text{otherwise} \end{cases}$
CompareGreater	$R_{i,j} = A_{i,j} > B_{i,j}$
CompareGreaterEqual	$R_{i,j} = A_{i,j} \geq B_{i,j}$
CompareLess	$R_{i,j} = A_{i,j} < B_{i,j}$
CompareLessEqual	$R_{i,j} = A_{i,j} \leq B_{i,j}$
Cond	$R_{i,j} = \begin{cases} B_{i,j} & \text{if } A_{i,j} = \text{true} \\ C_{i,j} & \text{otherwise} \end{cases}$

Opérateurs « Data-Parallel »

Exemple Microsoft Accelerator (2/3) : opérations de réduction

Operation	Definition
Sum(1)	$R_i = \sum_j A_{i,j}$
Product(1)	$R_i = \prod_j A_{i,j}$
MaxVal(1)	$R_i = \max_j A_{i,j}$
MinVal(1)	$R_i = \min_j A_{i,j}$
All(1)	$R_i = \bigwedge_j A_{i,j}$
Any(1)	$R_i = \bigvee_j A_{i,j}$

Opérateurs « Data-Parallel »

Exemple Microsoft Accelerator (3/3) : opérations de transformation

Operation	Definition
Section $(b_i, c_i, s_i,\ b_j, c_j, s_j)$	$R_{i,j} = A_{b_i+s_i\times i,\ b_j+s_j\times j}$
Shift(s_i, s_j)	$R_{i,j} = A_{i-s_i,\ j-s_j}$
Rotate(s_i, s_j)	$R_{i,j} = A_{(i-s_i) \bmod \text{size}_i,\ (j-s_j) \bmod \text{size}_j}$
Replicate$(\text{size}_i, \text{size}_j)$	$R_{i,j} = A_{i \bmod \text{size}_i,\ j \bmod \text{size}_j}$
Expand$(b_i, a_i,\ b_j, a_j)$	$R_{i,j} = A_{(i-b_i) \bmod \text{size}_i,\ (j-b_j) \bmod \text{size}_j}$
Pad$(b_i, a_i,\ b_j, a_j, c)$	$R_{i,j} = \begin{cases} A_{i-b_i,\ j-b_j} & \text{if in bounds} \\ c & \text{otherwise} \end{cases}$
Transpose(1, 0)	$R_{i,j} = A_{j,i}$

Opérateurs « Data-Parallel »

Exemple ZPL : régions

```
7  region R = [1..n, 1..n];            -- the computation indices
8         BigR = [0..n+1, 0..n+1];     -- the declaration indices
9
10 var A: [BigR] double;               -- the main data values
11     New: [R] double;                -- the new iteration's values
12     delta: double;                  -- change between iterations
13
14 direction north = [-1, 0];          -- the four cardinal directions
15           south = [ 1, 0];
16           east  = [ 0, 1];
17           west  = [ 0,-1];

33        New := (A@north + A@south +  -- five-point stencil on A
34                A@east  + A@west)/4.0;
```

Recherches en cours

Partitionnement automatique des données

- Adaptation de la granularité à l'architecture
  - Dimensions des données : GPU > CPU > SPU (CELL)

```
matrixAdd :: Num a => Matrix a -> Matrix a -> Matrix a
matrixAdd a b = zipWith2D (+) a b

-- Partitioning
= \' h1 v1 h2 v2 -> zipWith2D (+) a' b'
  where
    a' = concat2D h1 v1 $ split2D h1 v1 a
    b' = concat2D h2 v2 $ split2D h2 v2 b

-- Unification (h1=h2, v1=v2) + ZipWith2D/Concat2D rule
= \' h v -> concat2D h v $ zipWith2D (+) a' b'
  where
    a' = split2D h v a
    b' = split2D h v b
```

Recherches en cours

Partitionnement automatique des données

- Adaptation de la granularité à l'architecture
  - Dimensions des données : GPU > CPU > SPU (CELL)

```
matrixAdd :: Num a => Matrix a -> Matrix a -> Matrix a
matrixAdd a b = zipWith2D (+) a b

-- Partitioning
= \' h1 v1 h2 v2 -> zipWith2D (+) a' b'
  where
    a' = concat2D h1 v1 $ split2D h1 v1 a
    b' = concat2D h2 v2 $ split2D h2 v2 b

-- Unification (h1=h2, v1=v2) + ZipWith2D/Concat2D rule
= \' h v -> concat2D h v $ zipWith2D (+) a' b'
  where
    a' = split2D h v a
    b' = split2D h v b
```

Recherches en cours

Expressions d'algorithmes avec des langages de haut niveau

- À partir de code Latex (D. Barthou)
- À partir de langages fonctionnels (S. Henry, M. Chakravarty)

Recherches en cours

Consommation énergétique

- Intégration de la consommation énergétique dans les modèles de performance
- Support de la mise en veille / extinction dynamique des cœurs

Recherches en cours

Intéractions réseau + accélérateurs

- Support matériel DMA entre carte réseau et carte graphique
  - Supprime des contentions sur la mémoire hôte
- Modèles de programmation
  - Support exécutif pour accélérateurs + MPI ?
  - Modèle type MapReduce ?
  - Modèle de type graphe de tâches ?