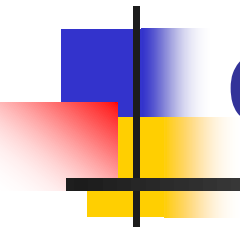
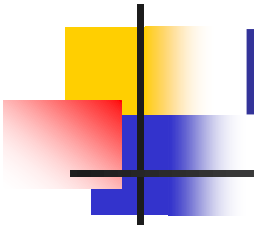


Architecture et programmation d'accélérateurs matériels

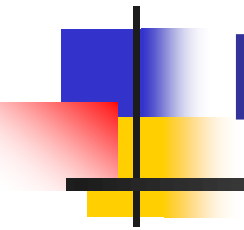
Cours 2 : Programmation CUDA avancée et OpenCL

Julien Jaeger
Patrick Carribault
Julien.jaeger@cea.fr

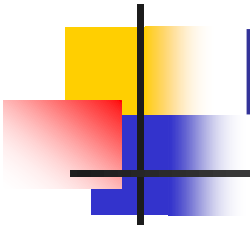


Plan du cours

- Programmation avancée CUDA
- Outils et bibliothèques annexes
- Programmation OpenCL

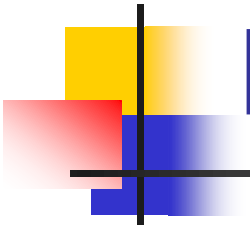


Programmation avancée CUDA



Plan - Programmation

- Langage de programmation
  - Mots clés
  - Fonctions disponibles
  - Exemples
- Optimisations de *kernel*
- Multi-GPUs



Programmation CUDA

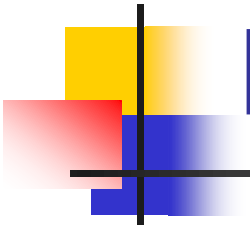
- Noyau de calcul
  - Basé sur la norme C99
  - Quelques restrictions
  - Quelques ajouts
- Extensions
  - Mots clés
  - Variables définies par défaut
  - Fonctions
- Voir l'appendice B de la documentation *CUDA C Programming Guide*

Noyau de base

- Syntaxe de base pour un noyau
  - Fonction qui ne renvoie rien (retour de type *void*)
  - Attribut définissant la fonction comme s'exécutant sur le *device*
    - `__global__`
  - Arguments en entrée

- Exemple

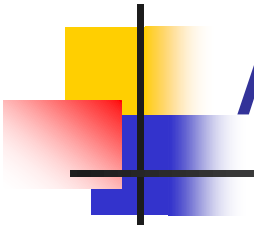
```
__global__ void vecAddKernel( double *a,
        double *b, double *c, int N ) {
   int i ;
   i = blockIdx.x * blockDim.x + threadIdx.x ;
   if ( i<N ) {
        c[i] = a[i]+b[i];
   }
}
```



Extensions – Mots clés

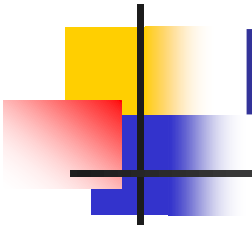
- Définition de nouveaux mots clés
- Catégories
  - Attributs de fonctions
  - Attributs de variables
  - Types
- Ensemble de variables définies par défaut
  - Utilisant les nouveaux types de données

Attributs de fonction

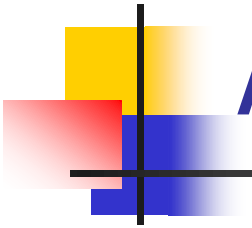


- Mot clé à ajouter dans la déclaration et la définition de la fonction
  - Ajout entre le type de retour et le nom de la fonction
- Fonction s'exécutant sur le device et appelable depuis l'hôte
  - `__global__`
- Fonction dédiée sur l'hôte ou le *device* (combinable)
  - `__host__`
  - `__device__`
- Par défaut, équivalent à `__host__`

Restrictions des fonctions



- Fonction déclarée `__global__`
  - Type de retour `void`
  - Appel avec un contexte d'exécution (nombre de blocs, nombre de threads par bloc, …)
  - Appel asynchrone
  - Arguments passée par la mémoire *shared* (256B) ou mémoire constante (4KB)
    - Attention : en fonction des *CUDA capabilities* de la carte !
  - Impossible de capturer son pointeur
- Fonction s'exécutant sur le *device*
  - Pas de variable statique
  - Pas de nombre d'arguments variable
  - Récursion restreinte (uniquement pour les fonctions déclarées `__device__`)



Attributs de variables

- Nouveaux attributs de variables permettant de définir l'emplacement mémoire des variables
- Variable résidente sur le *device*
  - `__device__`
  - Par défault dans la mémoire globale, accessible par tous les threads, pendant toute la durée de l'application
- Variable résidente dans la mémoire constante
  - `__constant__`
  - Durée de vie de l'application
  - Ne peut pas être défini sur le *device*
- Variable dans la mémoire *shared*
  - `__shared__`
  - Partagée entre tous les threads d'un même bloc
  - Une copie par bloc
  - Durée de vie du bloc
- Par défaut une variable déclarée sur le *device* est stockée dans un registre

Variable *volatile*

- Synchronisation des données communes accédées de façon concurrente
- Exemple d'accès concurrents

```
// myArray is an array of non-zero integers
// located in global or shared memory
__global__ void MyKernel(int* result) {
    int tid = threadIdx.x;
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```

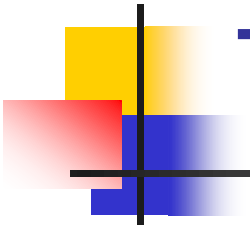
- Que vaut `result[tid]` ?
  - `myArray[tid]` est dans un registre, donc `ref1==ref2`
- Par contre, si déclaré *volatile*, alors ok (ou alors mettre une barrière mémoire - *memory fence*)
  - Mais cela ne garantie pas l'ordre d'exécution

Restrictions des variables

- Gestion dynamique des variables *shared*

```
extern __shared__ char array[];
__device__ void func() {
    short* array0 = (short*)array;
    float* array1 = (float*)&array[128];
    int* array2 = (int*)&array[64];
}
```

- Besoin de gérer à la main l'allocation des données si on décide d'utiliser la mémoire *shared* de façon dynamique
  - Respect des règles d'alignements



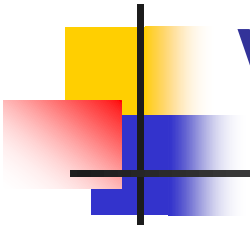
Type de données

- Nouveau types
  - Vecteur
  - Entiers multi-dimensions
- Vecteurs
  - Type de base + nombre de données
  - Exemple : int2, float4
  - Besoin de respecter les règles d'alignements
  - Fonctions associées pour construire un tel type
    - Exemple : `int2 make_int2(int x, int y);`

Type de données (suite)

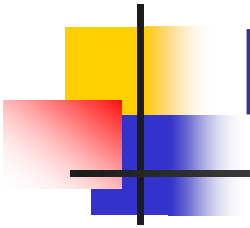
- Entiers 3 dimensions
  - `dim3`
  - Equivalent au type de vecteur `uint3`
  - Accès aux composantes par les champs `x`, `y` et `z`
  - Par défault, initialisé à 1
- Exemple

```
dim3 a ;
a.x = 4 ;
```



Variables définies

- Dimensions de la grille
  - `dim3 gridDim`
  - Contient le nombre de blocs dans les 3 dimensions de la grille
- Indice d'un bloc dans la grille
  - `dim3 blockIdx`
  - Contient les coordonnées du bloc courant dans la grille
- Dimensions des blocs
  - `dim3 blockDim`
  - Contient le nombre de threads dans les 3 dimensions des blocs
- Indice du thread dans le bloc
  - `uint3 threadIdx`
  - Contient l'indice du thread courant dans le bloc courant (3 dimensions)
- Taille d'un warp
  - `int warpSize`

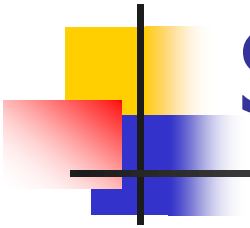


Fonctions disponibles

- Barrière mémoire
- Synchronisations
- Mathématiques
- Opérations atomiques
- *Timing*
- I/O

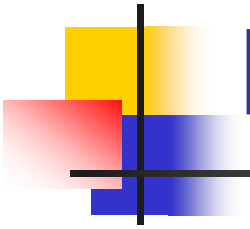
Barrière mémoire

- Points de synchronisations au niveau des transactions mémoire
  - L'ordre des écritures peut ne pas être respecté du point de vue d'un autre thread
- But : permettre d'avoir une garantie que les accès mémoire lancés sont effectivement visibles par un ensemble de threads
- Différents niveaux de barrière
  - Bloc : `void __threadfence_block();`
  - Device : `void __threadfence();`
  - Device + hôte : `void __threadfence_system();`



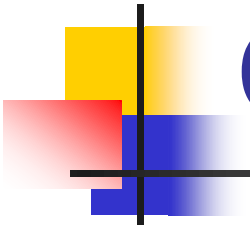
Synchronisation

- Synchronisation entre tous les threads d'un même bloc
  - `void __syncthreads();`
  - Permet également une synchronisation des données
  - Attention au flot de contrôle !
- Pour les cartes compatibles 2.0
  - `int __syncthreads_count(int predicate);`
  - `int __syncthreads_and(int predicate);`
  - `int __syncthreads_or(int predicate);`



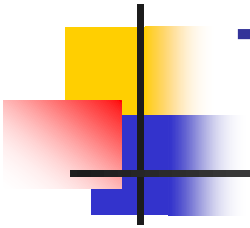
Mathématiques

- Ensemble de fonctions mathématiques optimisées pour GPU
  - Ex: sin(float), cos(double), …
- Option de compilation pour utiliser les fonctions optimisées
  - `-use_fast_math`
  - Seulement pour les calculs simple précisions



Opérations atomiques

- Instructions assurants une atomicité
  - Ex : `int atomicAdd(int* address, int val);`
- Fonctions disponibles
  - Addition : `atomicAdd`
  - Soustraction : `atomicSub`
  - Echange : `atomicExch`
  - Min/Max : `atomicMin`
  - Incrément/Décrément : `atomicInc`
  - CAS, …
- Restrictions
  - Fonctionne sur les entiers
  - Certaines fonctionnent sur les flottants 32bits



Timing et I/O

- Timestamp par multiprocesseur
  - `clock_t clock();`
  - Nombre de cycles
  - Possibilité de prendre la mesure par thread
- Sortie formattée
  - `int printf(const char *format[, arg, ...]);`
  - Cartes supportant les capacités 2.0
  - Fonction par thread
  - Format final fait sur l'hôte

Exemple

- Multiplication de matrices
- Utililisation d'une structure C
  - Possibilité d'avoir accès sur CPU et GPU
- Utilisation des variables définies avec 2 dimensions
- Tout le monde calcule un élément de la matrice résultante

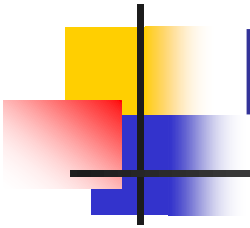
```
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

- Que se passe-t-il si la matrice est plus grande que le nombre max de threads ?

```
__global__ void
    MatMulKernel(Matrix A, Matrix
    B, Matrix C)
{

float Cvalue = 0;
int row = blockIdx.y * blockDim.y
    + threadIdx.y;
int col = blockIdx.x * blockDim.x
    + threadIdx.x;

for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row *
    A.width + e]
          * B.elements[e * B.width
    + col];
C.elements[row * C.width + col] =
    Cvalue;
}
```

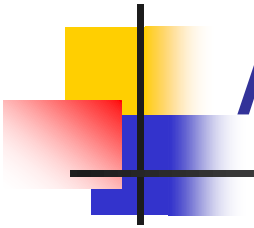
Exemple

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix
   C)
{

float Cvalue = 0;
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;

for (int r = row ; r < C.height ; r += blockDim.y )
   for (int c = col ; c < C.width ; c += blockDim.x )
       for (int e = 0; e < A.width; ++e)
              Cvalue += A.elements[row * A.width + e]
                     * B.elements[e * B.width + col];
              C.elements[row * C.width + col] = Cvalue;
}
```



Plan - Programmation

- Langage de programmation
- Optimisations de *kernel*
  - Allocation de registres
  - Flot de contrôle
  - Accès mémoire
  - Synchronisations
- Multi-GPUs



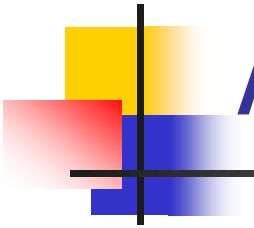
Allocation de registres

- Chaque noyau de calcul a besoin d'utiliser plusieurs registres
- En fonction des instructions présentes dans le noyau
  - Transformations/optimisations du compilateur
  - Allocation de registres
- Mais
  - Le nombre de registres est limité
  - Les registres sont partagés entre les threads s'exécutant sur un même *Streaming Multiprocessor*
- Relation avec le nombre de threads ?

Allocation de registres

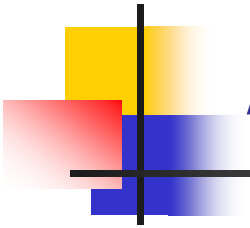
- Option pour définir une borne au compilateur
  - `-maxrregcount=N`
- Attribut pour donner une indication sur le nombre maximum de threads et de blocs

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
  minBlocksPerMultiprocessor)
MyKernel(...)
{
    ...
}
```



Accès mémoire

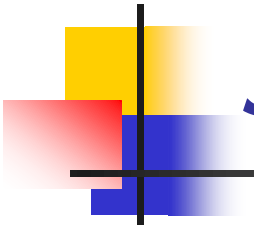
- A cause de cette exécution synchrone
  - Un thread exécutant un chargement mémoire ralenti les autres
  - Plusieurs accès mémoire simultanées peuvent être sérialisés
  - Plus l'accès est long, plus il nous faudra de threads pour recouvrir cet accès
- Optimisations possibles
  - *Load coalescing*
  - Eviter les conflits de bancs



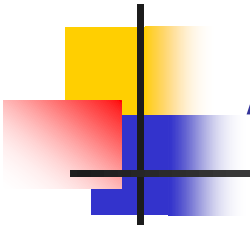
Load coalescing

- Accès à la mémoire globale
- Accès concurrent émis par les threads d'un même warp
  - Tous les threads d'un même warp exécute la même instruction au même instant
- Les requêtes sont sérialisés par paquets de 128 octets (taille de la ligne de cache)
  - Optimisation si ces accès sont contigus !
- Exemples…

Load coalescing et mémoire *shared*



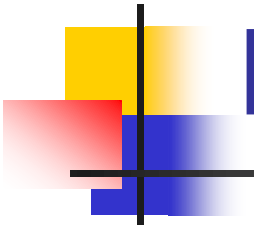
- Utilisation de la mémoire shared
  - Nécessite de déclarer des *buffers* résidant dans la mémoire shared
  - Transferts des données en début de noyau
  - Mise à jour de la mémoire globale à la fin du noyau
- Optimisation : profiter de ce premier transferts (global → shared) pour faire des accès contigus aux données
  - Même si toutes les données ne sont pas nécessaires !
- Attention aux conflits de bancs



Best Practices

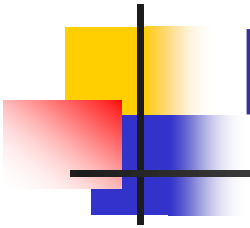
- Priorité haute
  - Penser parallèle
  - Minimiser les transferts hote/device
  - Accès coalescer à la mémoire globale
  - Utilisation de la mémoire shared
  - Eviter de multiplier les chemins d'exécution dans le code

- Priorité moyenne/basse
  - Eviter les conflits de banc de la mémoire shared
  - Avoir un grand nombre de threads par blocs (multiple de 32)
  - Utilisation des fonctions mathématiques optimisées



Plan - Programmation

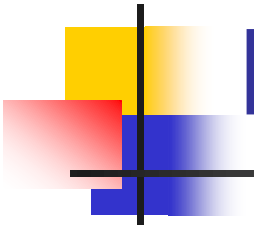
- Langage de programmation
- Optimisations de *kernel*
- Multi-GPUs



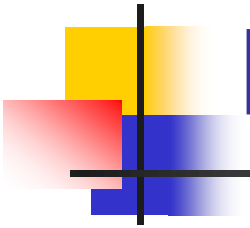
Programmation Multi-GPUs

- Supercalculateur hybride
  - Plusieurs cartes graphiques par noeud
  - Exemple : serveur Tesla
- Comment programmer pour plusieurs cartes compatible CUDA ?
  - Selection du *device* : `cudaSetDevice(int)`
  - Utilisation dans un contexte MPI et/ou OpenMP

Programmation Multi-GPUs

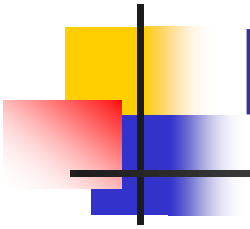


- Communication entre les cartes du même noeud
  - Besoin de passer par l'hôte pour transférer d'une carte à l'autre
  - Pas de possibilité de communication directe entre les cartes
- Communication entre cartes sur des noeuds différents
  - Idem
  - L'hôte fait ensuite la communication sur le réseau (par exemple, via MPI)



Plan – Outils et bibliothèques

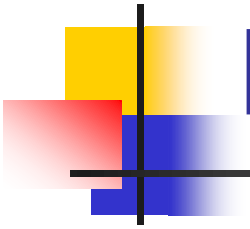
- Bibliothèques optimisées
  - BLAS
  - FFT
- Ordonnancement hybride
- Génération de code
- Portabilité



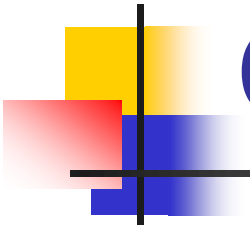
Plan – Outils et bibliothèques

- Bibliothèques optimisées
- Ordonnancement hybride
  - StarPU
- Génération de code
- Portabilité

Plan – Outils et bibliothèques

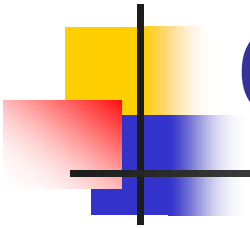


- Bibliothèques optimisées
- Ordonnancement hybride
- Génération de code
  - OpenACC
  - OpenMP 4+
- Portabilité



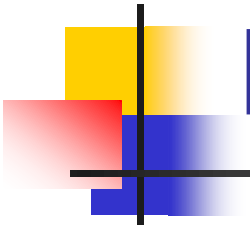
OpenACC

- Sous-groupe du compité OpenMP en charge du support des accélérateurs
  - *Fork* afin de créer une nouvelle norme à base de directives pour la gestion des accélérateurs
  - Membres de ce groupe : Cray, PGI, CAPS
- Principe :
  - Directive à la OpenMP pour la gestion des accélérateurs
  - Description des données en entrée et sortie pour les transferts automatiques
  - Gestion de l'asynchronisme
  - Gestion avancée des transferts
- Version courante
  - 2.5 (version initiale 1.0 depuis SC2011)
  - Future intégration dans la norme OpenMP ?



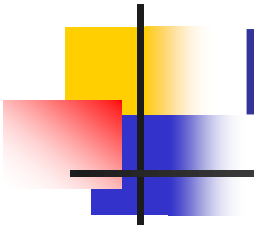
OpenMP 4+

- Nouvelle version du standard OpenMP avec directive pour la programmation d'accélérateur
  - Très voisins des directives OpenACC
  - La plupart des membres du sous-comité « accelerator » appartiennent aussi au comité OpenACC.
- Principe :
  - Directive pour la gestion des accélérateurs
  - Description des données en entrée et sortie pour les transferts automatiques
  - Gestion de l'asynchronisme
  - Gestion avancée des transferts
  - Découplée des autres directives OpenMP
- Version courante
  - 4.5 (version « 1.5 » des directies pour accélérateurs)



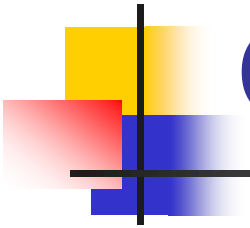
Plan – Outils et bibliothèques

- Bibliothèques optimisées
- Ordonnancement hybride
- Génération de code
- Portabilité
  - GPU Ocelot
  - Vers un langage portable



Portabilité

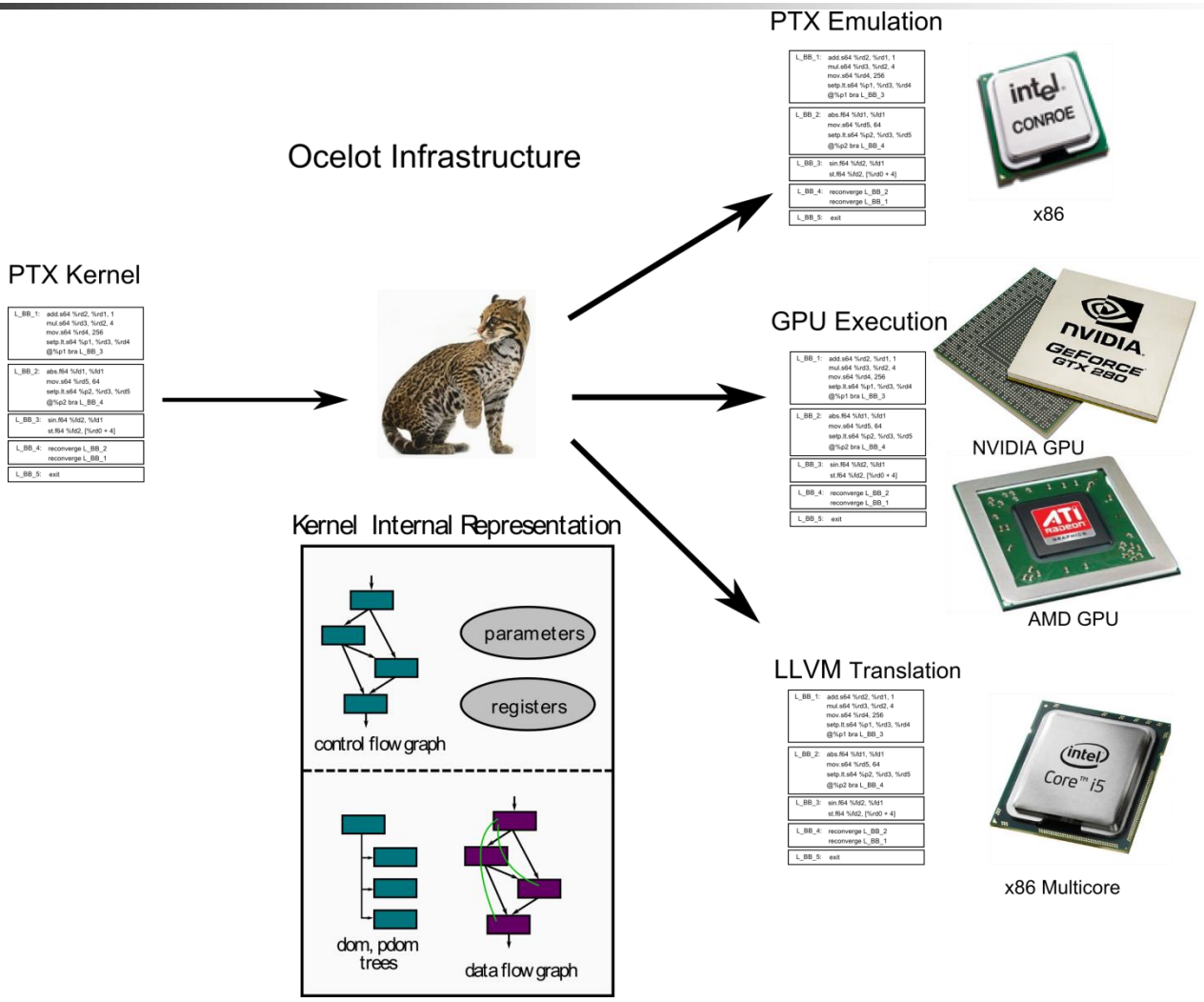
- Même s'il existe des générateurs de code plus ou moins automatique, la portabilité reste un problème
- Approches afin de porter *automatiquement* un code vers d'autres langage
  - GPU ocelot
- Développement d'un langage portable
  - OpenCL

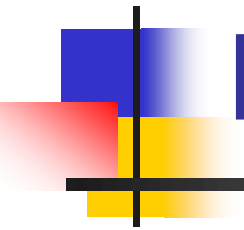


GPU Ocelot

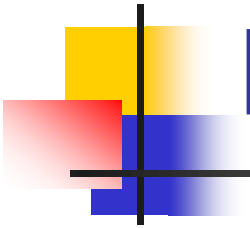
- Exécution de code assembleur CUDA sur plusieurs architectures
- Entrée : code PTX
- Sortie : code pour CPU ou autre accélérateurs
- Utilisation
  - Portabilité
  - Debugging

GPU Ocelot


Ocelot Infrastructure
PTX Kernel
Kernel Internal Representation
parameters
registers
control flow graph
dom, pdom trees
data flow graph
PTX Emulation
x86
GPU Execution
NVIDIA GPU
AMD GPU
LLVM Translation
x86 Multicore

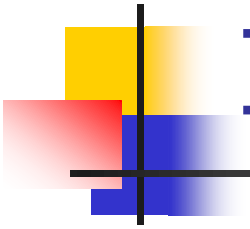


Programmation OpenCL



Plan – Programmation OpenCL

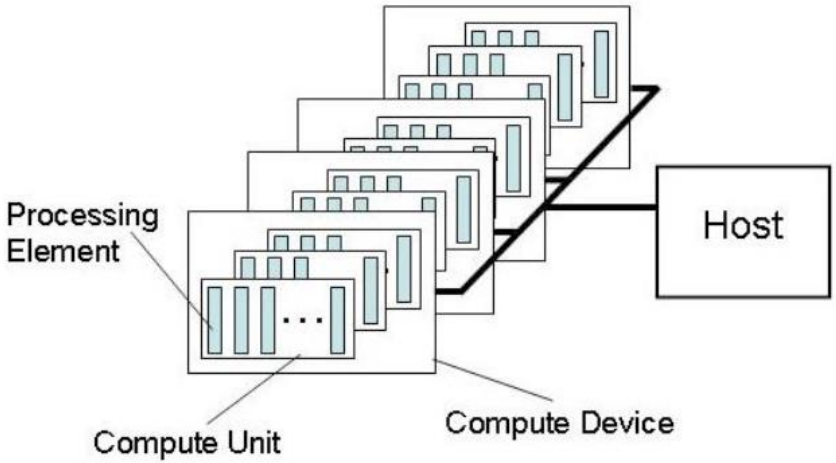
- Introduction
  - Vision globale
  - Modèle d'exécution
- Abstraction
- Comparaison avec CUDA



Introduction OpenCL

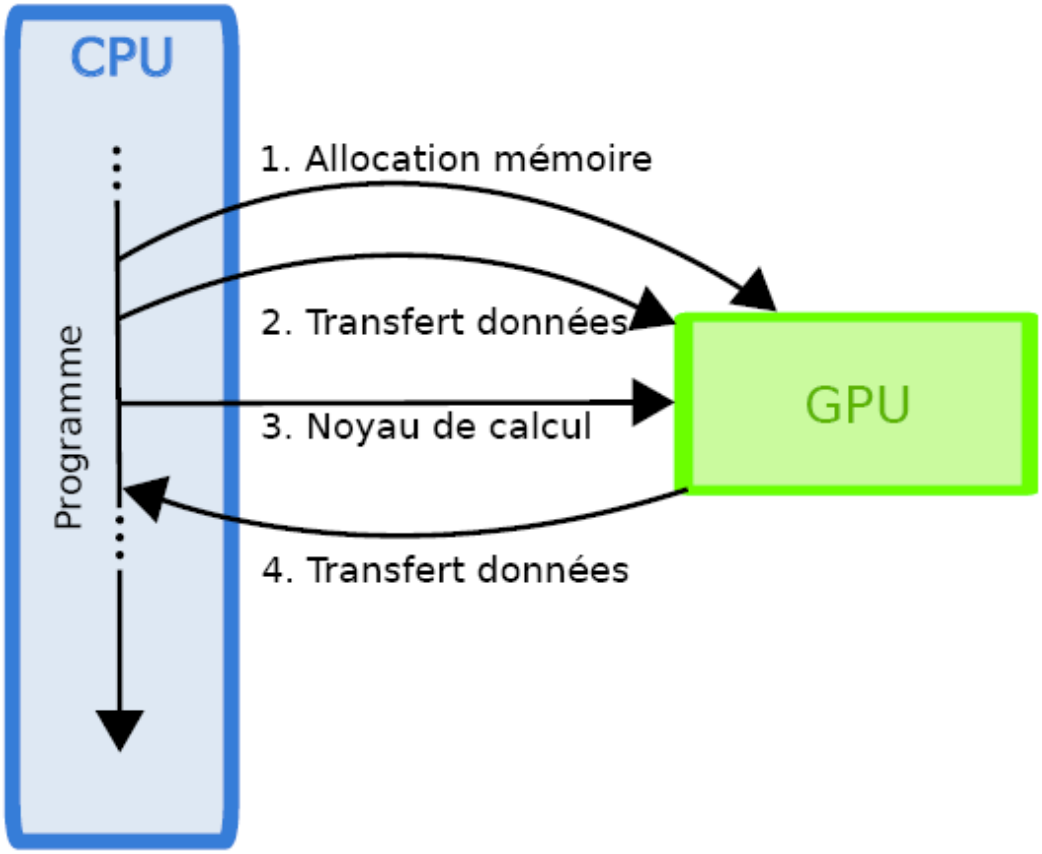
- Open Compute Language
  - http://www.khronos.org/opencl
  - Version courante 1.2
- Modèle de programmation portable
  - Utilisation des processeurs généralistes (CPUs)
  - Utilisation des cartes graphiques (GPUs)
  - Utilisation des accélérateurs (Cell)

Vision globale



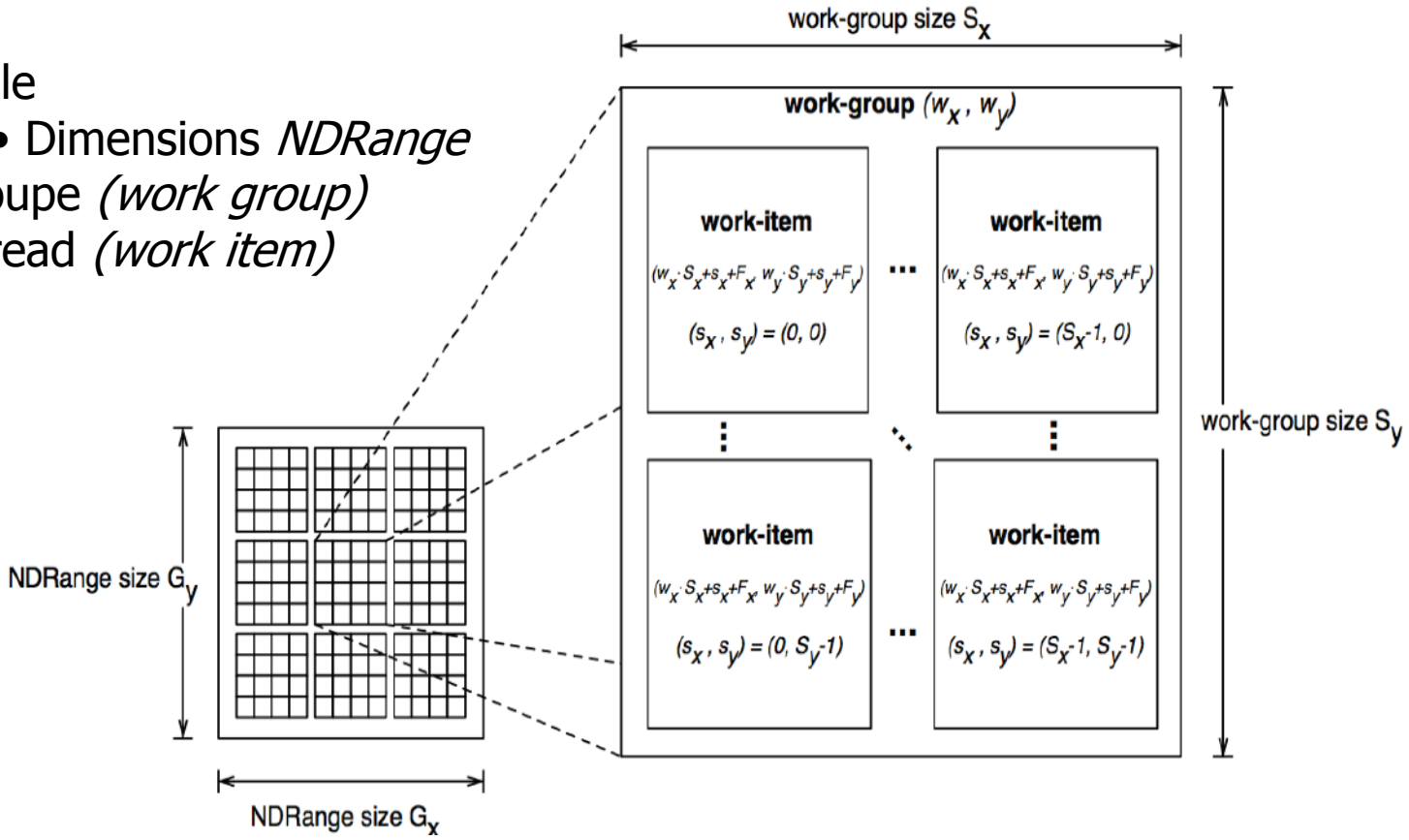
- Un hôte contrôle les accélérateurs
- Vision hiérarchique :
  - *Compute device*
  - *Compute unit*
  - *Processing element*

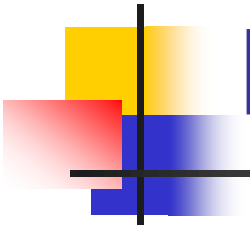
Principe d'exécution



Modèle d'exécution

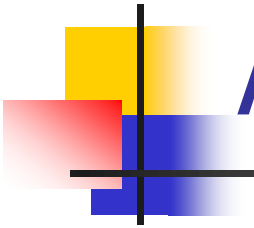
- Grille
  - Dimensions *NDRange*
- Groupe *(work group)*
- Thread *(work item)*





Plan – Programmation OpenCL

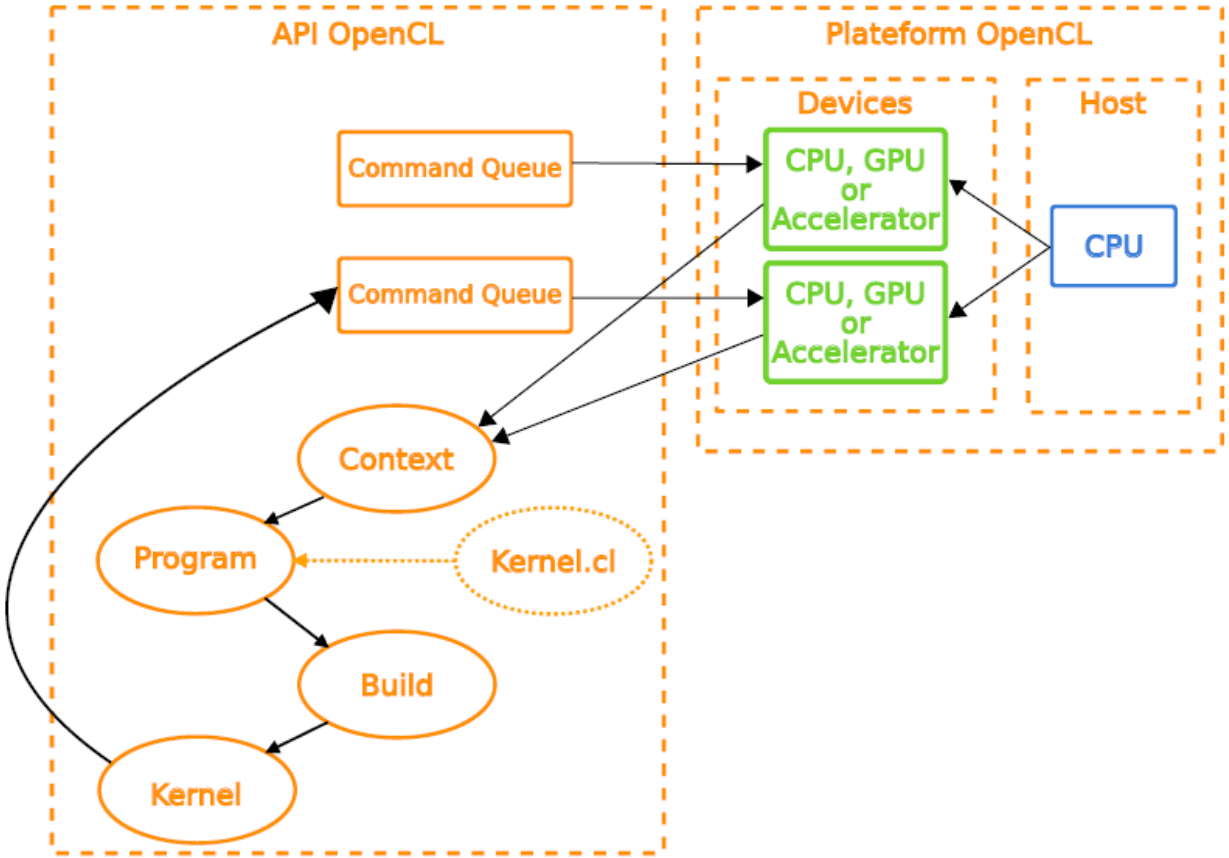
- Introduction
- Abstraction
  - Vision logique
  - File d'exécution
- Comparaison avec CUDA

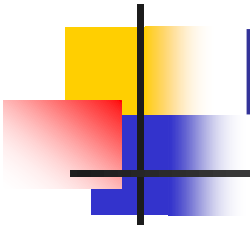


Abstraction

- Vision de plusieurs devices de types différents
  - Les processeurs multicoeurs sont vus comme un device
- Le programme est compilé à la volée
  - Création d'un context, programme, arguments, …
- Actions (exécution d'un noyau, transferts)
  - Command queue
  - Notion de dépendances (exécution dans le désordre)
  - Notion d'événements

Abstraction d'OpenCL





Plan – Programmation OpenCL

- Introduction
- Abstraction
- Comparaison avec CUDA
  - Hôte
  - Device

Hôte – CUDA

```
1  /*****************/
2  /*Programme hote
3  /*****************/
4  main(){
5  //1- Allocation memoire
6  cudaMalloc(&C, sizeof(float)*niter);
7  cudaMalloc(&A, sizeof(float)*niter);
8  cudaMalloc(&B, sizeof(float)*niter);
9
10 //2- Transfert CPU -> GPU
11 cudaMemcpy(A,hA,sizeof(float)*niter,
       cudaMemcpyHostToDevice);
12 cudaMemcpy(B,hB,sizeof(float)*niter,
       cudaMemcpyHostToDevice);
13
14 //3- Execution du noyau de calcul
15 addVectorKernel<<<64,128>>>(C,niter,A,B)
       ;
16
17 //4- Transfert GPU -> CPU
18 cudaMemcpy(hC,C,sizeof(reals)*niter,
       cudaMemcpyDeviceToHost);
19 }
```

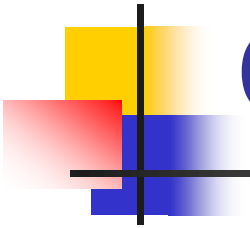
Hôte – OpenCL

```
1  /*****************/
2  /*Programme hote
3  /*****************/
4  main(){
5  //Initialisation simplifiee plus de 200
       lignes de codes.
6  //Fonctions utilisees :
7  // - clGetPlatformIDs
8  // - clGetDeviceIDs
9  // - clCreateContext
10 // - clCreateCommandQueue
11 // - clCreateProgramWithSource
12 // - clBuildProgram
13 // - clCreateKernel
14
15 //1- Allocations memoires
16 A = clCreateBuffer(context,
       CL_MEM_READ_ONLY,size,NULL,&errCl);
17 B = clCreateBuffer(context,
       CL_MEM_READ_ONLY,size,NULL,&errCl);
18 C = clCreateBuffer(context,
       CL_MEM_WRITE_ONLY,size,NULL,&errCl)
        ;
19
20 //2- Transfert CPU -> GPU
21 clEnqueueWriteBuffer(cmdQueue, A,CL_TRUE
       ,0,size,hA,0,NULL,&event[0]));
22 clEnqueueWriteBuffer(cmdQueue, B,CL_TRUE
       ,0,size,hB,0,NULL,&event[1]));
23
24 //3- Initialisation des arguments et
       appel du noyau de calcul
25 clSetKernelArg(kernel,0,size,&C);
26 clSetKernelArg(kernel,1,size,&niter);
27 clSetKernelArg(kernel,2,size,&A);
28 clSetKernelArg(kernel,3,size,&B);
29
30 //3- Execution du noyau de calcul
31 clEnqueueNDRangeKernel(cmdQueue,kernel
       ,1,NULL,8192,128,2,event,&event[2]);
32
33 //4- Transfert GPU -> CPU
34 clEnqueueReadBuffer(cmdQueue, C,CL_TRUE
       ,0,size,hC,1,&event[2],&event[3]) ;
35 }
```

Noyau – CUDA vs. OpenCL

```
1 /******************/
2 /*Noyau de calcul
3 /******************/
4 __global__ void addVectorKernel(float *C
      ,int niter,float *A,float * B)
5 {
6  //Indice globale du thread
7  int i = blockIdx.x * blockDim.x +
       threadIdx.x;
8          C[i] = A[i]+B[i];
9 }
```

```
1 /******************/
2 /*Noyau de calcul
3 /******************/
4 __kernel void addVectorKernel(__global
      float *C,int niter,float *A,
      __global float * B)
5 {
6  //Indice globale du thread
7  int i = get_global_id(0);
8
9          C[i] = A[i]+B[i];
10 }
```



Conclusion – OpenCL

- Modèle de programmation unifié pour accélérateur parallèle
- Avantages :
  - Portabilité
  - Exécution dynamique (gestion des dépendances)
- Inconvénients
  - Verbeux
  - Portabilité en option pour chaque implémentation
  - Bas niveau