DE LA RECHERCHE À L'INDUSTRIE

Introduction à la Programmation GPU avec OpenMP

Adrien Roussel | PGPU – ENSIIE

Outline

1 Introduction aux accélérateurs de calculs

2 Architecture des cartes graphiques NVIDIA

3 Introduction à la Programmation GPU

4 Programmation GPU avec OpenMP
- Modèle d'exécution des threads
- Gestion et Transferts de données
- Répartition des charges

5 Programmation GPU Avancée
- Accès Mémoire
- Recouvrement des Transferts mémoire

Outline

1 Introduction aux accélérateurs de calculs

2 Architecture des cartes graphiques NVIDIA

3 Introduction à la Programmation GPU

4 Programmation GPU avec OpenMP
- Modèle d'exécution des threads
- Gestion et Transferts de données
- Répartition des charges

5 Programmation GPU Avancée
- Accès Mémoire
- Recouvrement des Transferts mémoire

Évolutions matérielles

- Architectures actuelles
  - Augmentation du nombre de cœurs par nœud
    - Clovis (2010) : 12 cœurs / nœud (2 $\times$ 6 cœurs Intel Westmere-EP)
    - Romeo (2013) : 16 cœurs / nœud (2 $\times$ 8 cœurs Intel Sandy Bridge)
    - Romeo (2018) : 28 cœurs / nœud (2 $\times$ 14 cœurs Intel Skylake)
    - Intel Xeon Phi Knights Landing : 72 cœurs / processeur
  - Apparition des accélérateurs de calculs
    - GPU : plusieurs milliers de cœurs simplifiés
      Exemples :
      - K20 (Romeo 2013) : 2496 cœurs CUDA
      - P100 (Romeo 2018) : 3584 cœurs CUDA
      - V100 : 5120 cœurs CUDA
      - A100 : 8192 cœurs CUDA

Évolutions matérielles

- Architectures actuelles
  - Augmentation du nombre de cœurs par nœud
    - Clovis (2010) : 12 cœurs / nœud (2 × 6 cœurs Intel Westmere-EP)
    - Romeo (2013) : 16 cœurs / nœud (2 × 8 cœurs Intel Sandy Bridge)
    - Romeo (2018) : 28 cœurs / nœud (2 × 14 cœurs Intel Skylake)
    - Intel Xeon Phi Knights Landing : 72 cœurs / processeur
  - Apparition des accélérateurs de calculs
    - GPU : plusieurs milliers de cœurs simplifiés
      Exemples :
      - K20 (Romeo 2013) : 2496 cœurs CUDA
      - P100 (Romeo 2018) : 3584 cœurs CUDA
      - V100 : 5120 cœurs CUDA
      - A100 : 8192 cœurs CUDA

- Difficultés à surmonter pour le passage à l'échelle des codes
  - Besoin d'exposer toujours plus de parallélisme
    - Granularité de calculs plus fine
  - Efficacité parallèle accrue
    - Réduire les points de synchronisation
    - Toujours plus d'asynchronisme

Définition

Ressource distincte (située sur la même puce que le processeur principal ou non) définie pour un ensemble différents d'objectifs que le processeur central.

En résumé

- Processeur dédié à une classe spéciale d'applications
- Permet d'obtenir de meilleures performances dans son domaine d'application

- Les accélérateurs sont reliés au(x) processeur(s) principal(aux) grâce à une **interconnexion**.

- Comportement similaire à un périphérique
  - Le processeur principal contrôle l'accès.
  - Il exécute également la partie principale du code.

- Plusieurs types d'interconnexions :
  - Accélérateurs déportés.
  - Accélérateurs embarqués (sur la puce principale).

Différents types de connexion

- Via le bus système
  - GPGPU NVIDIA, MIC Intel
  - Avantages :
    - Faible coût
    - Intégration aisée
  - Inconvénients :
    - Performances modestes dûes à la généricité du bus
    - Bande passante et latence sous-optimales

- Via le bus du processeur
  - IBM Cell, Architecture Fusion d'AMD/ATI, NVIDIA Tegra
  - Avantages :
    - Accés rapide
    - Possibilité de partager des ressources avec le CPU
  - Inconvénients :
    - Intégration complexe
    - Dépendance avec le processeur

Outline

1 Introduction aux accélérateurs de calculs

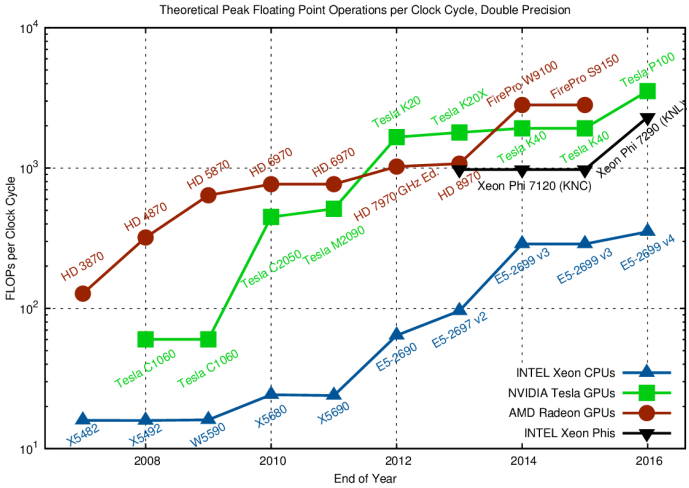
2 Architecture des cartes graphiques NVIDIA

3 Introduction à la Programmation GPU

4 Programmation GPU avec OpenMP
- Modèle d'exécution des threads
- Gestion et Transferts de données
- Répartition des charges

5 Programmation GPU Avancée
- Accès Mémoire
- Recouvrement des Transferts mémoire

Évolution des architectures NVIDIA – Performances flottantes



Source : https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Architecture NVIDIA GeForce

- Brique de base
  - Cœur de calcul très simple
  - Ensemble de cœurs de calculs regroupés dans un Streaming Multiprocessor (SMX)
  - (Qui sont eux même regroupés dans un Graphics Processing Clusters (GPC) – depuis Maxwell)

- Pourquoi *Stream* ?
  - Fait pour fonctionner sur un flux de données continu
  - Langage synchrone sur chaque multiprocesseur

- Design pour le parallélisme à grain fin

Architecture NVIDIA Kepler

- Introduite en 2012
  - Remplaçante de Fermi (2009)

- Sera remplacée par Maxwell en 2014 (Grand public)
  - Mais surtout par Pascal (gamme Tesla pour HPC) en 2016

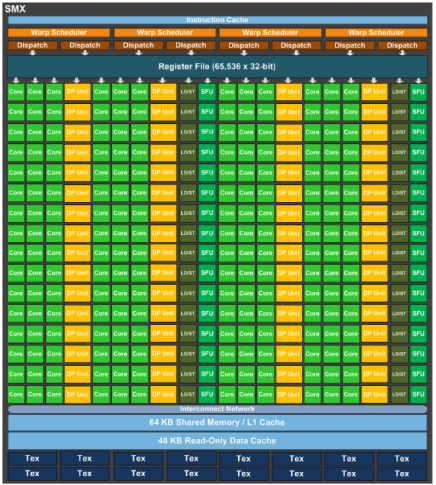
- Principales caractéristiques :
  - Augmentation du nombre de cœurs
  - Support des *compute capabilities 3.5*
  - Gestion optimisée du lancement de multiples noyaux de calcul
  - Amélioration des performances relatives en double précision

NVIDIA Kepler

- 7.1B Transistors
- 15 SMX
  - 192 cœurs CUDA / SMX
- $>$ 1TFlop FP64
- 1.5Mo L2 Cache
- 6 $\times$ contrôleurs mémoire 64-bits
- PCI-E Gen. 3

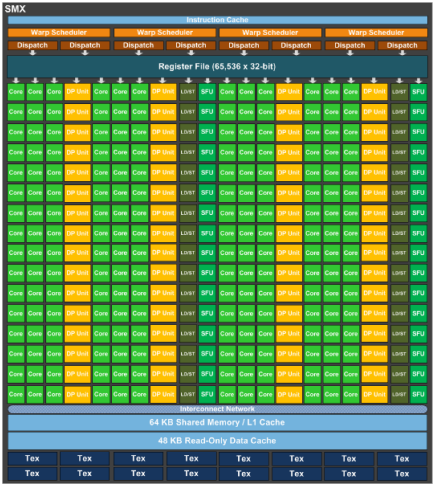
Architecture SMX Kepler en détails 1/2

- 192 cœurs compatible CUDA
  - Unité ALU
  - Unité Flottante Simple Précision
- 64 Unités Flottantes Double Précision
- 32 Unités Fonctionnelles Spéciales (SFU)
- 32 Unités Load/Store



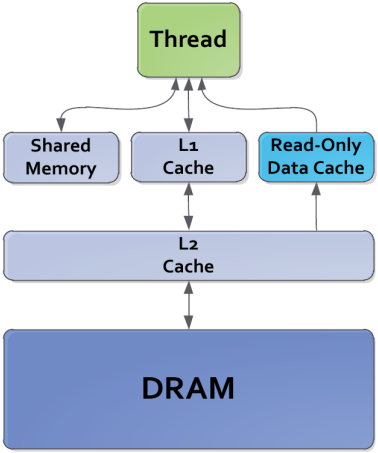
Architecture SMX Kepler en détails 2/2

- Exécution de type SIMT
  - Single Instruction Mutiple Threads
- Similaire au SIMD
- Exécution synchrone des cœurs de calculs
- Chaque SMX gère et exécute les threads par paquets de 32
  - Notion de warp
  - 4 Ordonnanceurs de Warps par SMX



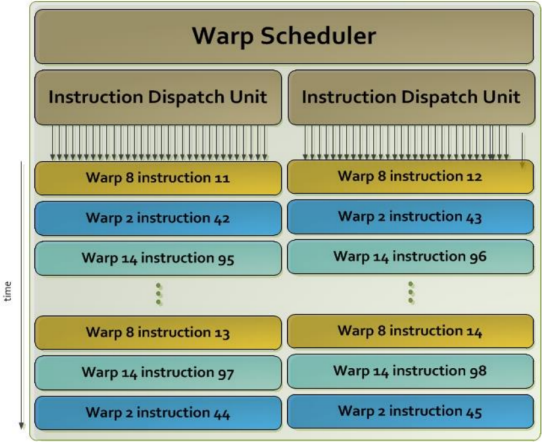
Kepler – Hiérarchie Mémoire

- Un thread est ordonnancé sur un cœur de calcul d'un SMX
- Accès mémoire via instruction Load ou Store
  - Accès mémoire partagée ⇒ pas de cache
  - Accès à la mémoire globale du device ⇒ passage par les 2 niveaux de cache
- Deux possibilités
- Physiquement, la même mémoire est utilisée pour le cache L1 et la shared memory
  - 64KB / SMX
  - Configurable au runtime
- Apparition d'un Read-Only Cache (48KB)



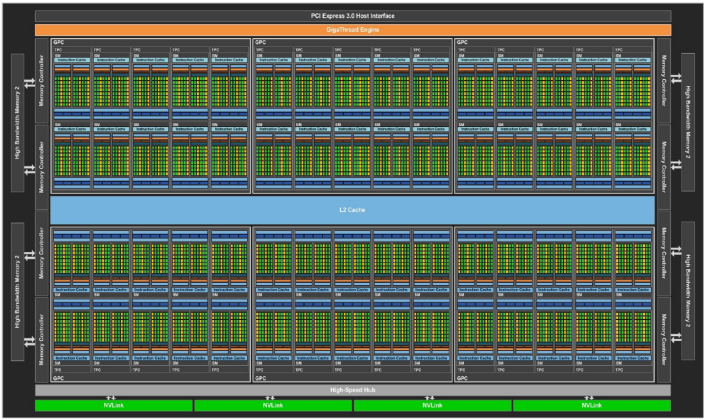
Kepler – Schéma d'exécution – Warp Scheduler

- 4 Warp Scheduler / SMX
  - 1 Warp = 32 threads
- 8 Dispatch Unit / SMX
  - 2 Dispatch Unit / Warp Scheduler
- Quad Warp Scheduler – Mode de fonctionnement
  - Sélection de 4 Warps (1 / Warp Scheduler)
  - 2 instructions indépendantes par Warp peuvent être envoyées à chaque cycle



Pascal : le (vrai) successeur de Kepler

- 6 × Graphic Processing Clusters (GPC)
- 30 × Texture Processing Clusters (TPC)
  - 5 TPC / GPC
- 60 × Pascal Streaming Multiprocessors (SM)
  - 2 SMs / TPC
  - 10 SMs / GPC
- 8 × Contrôleurs mémoire 512-bits
- Avancée majeure :
  - Support de la Mémoire Unifiée
    - Espace d'adressage virtuel unifié pour accéder aux données CPU et GPU avec transferts implicites



Pascal : Architecture d'un SM

- 2 × Processing Block
  - 32 cœurs Simple Précision
  - 16 Unités Double Précision
  - 8 Unités Load/Store
  - 1 Warp Scheduler
  - 2 Dispatch Unit

Ampere : Architecture actuelle

- 8 × Graphic Processing Clusters (GPC)
- 64 × Texture Processing Clusters (TPC)
  - 8 TPCs / GPC
- 128 × Ampere Streaming Multiprocessors (SM)
  - 2 SMs / TPC
  - 16 SMs / GPC
- 12 × Contrôleurs mémoire 512-bits
- 6 × High Bandwidth Memory (HBM2) Stacks

Ampere : Architecture d'un SM

- 4 × Processing Blocks
  - 16 Unités INT32
  - 16 Unités Simple Précision
  - 8 Unités Double Précision
  - 8 Unités Load/Store
  - 1 Warp Scheduler
  - 1 Dispatch Unit
- 32 warps actifs maximum

Hopper : Architecture future (2023)

- 8 × Graphic Processing Clusters (GPC)
- 72 × Texture Processing Clusters (TPC)
  - 9 TPCs / GPC
- 144 (132?)× Ampere Streaming Multiprocessors (SM)
  - 2 SMs / TPC
  - 18 SMs / GPC
- 12 × Contrôleurs mémoire 512-bits
- 6 × High Bandwidth Memory (HBM2) Stacks
- 60MB L2 Cache

Hopper : Architecture d'un SM

- 4 × Processing Blocks
  - 16 Unités INT32
  - 32 Unités Simple Précision
  - 16 Unités Double Précision
  - 8 Unités Load/Store
  - 1 Warp Scheduler
  - 1 Dispatch Unit
- 64 warps actifs maximum

- Langages Synchrones et Data Flow
- Langages Dédiés
  - Close To Metal (CTM)
  - CAL/IL
  - HIP
  - CUDA
- Langage Portable
  - OpenCL
- Directives de Parallélisation
  - HMPP (Caps Entreprise)
  - OpenACC
  - OpenMP (target)

Outline

1 Introduction aux accélérateurs de calculs

2 Architecture des cartes graphiques NVIDIA

3 Introduction à la Programmation GPU

4 Programmation GPU avec OpenMP
- Modèle d'exécution des threads
- Gestion et Transferts de données
- Répartition des charges

5 Programmation GPU Avancée
- Accès Mémoire
- Recouvrement des Transferts mémoire

Vue d'ensemble

- Deux parties :
  - Hôte : CPU
  - Device : GPU
- Interface entre les deux :
  - Transferts de données
  - Lancement de noyaux de calculs (kernel)
- Chaîne de compilation dédiée
  - Noyaux de calculs écrits dans un langage différent de la partie hôte
- Lien avec un runtime (en espace utilisateur) et un driver (en espace noyau)

Schéma de Programmation

1. Initialisation
   - Allocation mémoire
   - Lecture des entrées
   - Initialisation de données
   - ...
2. Allocation Mémoire sur le GPU
   - Gestion manuelle (ne partage pas le même espace d'adressage)
3. Transferts de données CPU $\rightarrow$ GPU
4. Exécution d'un noyau de calcul sur GPU
   - Possibilité de lancer plusieurs noyaux
5. Rapatriement des données GPU $\rightarrow$ CPU

Comment Compiler (avec un GPU NVIDIA) ?

- Structure du code
  - Partie hôte
  - Partie device
  - ⇒ Peuvent être mélangées dans le même fichier
- Programmation Hôte
  - C/C++
- Programmation Device
  - CUDA (sur-ensemble du C norme 99)
- Chaîne de compilation : NVCC
  - En tant que programmeur : utilisation du compilateur `nvcc`
  - Documentation très bien détaillée, n'hésitez pas à y jeter un oeil

Exemple : Addition de deux vecteurs

- Opération de base
  - `c[i] = a[i] + b[i]`
- Comment programmer ce noyau de calcul en séquentiel ?
  - Simple boucle
  - Application de l'opération de base sur chaque élèment

```
void vectAdd(
  double * a, double * b,
  double * c, int N)
{
  int i ;
  for (i=0;i<N;i++)
  {
    c[i] = a[i] + b[i]
  }
}
```

Portage de notre exemple sur GPU

```
void vectAdd(
  double * a, double * b,
  double * c, int N)
{
  int i ;

  for (i=0;i<N;i++)
  {
    c[i] = a[i] + b[i]
  }



}
```

Portage de notre exemple sur GPU

```
void vectAdd(
  double * a, double * b,
  double * c, int N)
{
  int i ;

  for (i=0;i<N;i++)
  {
    c[i] = a[i] + b[i]
  }

}
```

Transferts des tableaux `a, b et c`

Exécution parallèle sur GPU

Rapatriement de c sur CPU

Programmation sur l'hôte

- Gestion de l'hôte (CPU Central)
  - Besoin de gérer la mémoire du GPU
  - Besoin de transférer les données
  - Donne l'ordre d'exécution du kernel

- API CUDA de l'hôte
  - CUDA haut/bas niveau
  - `cudaMalloc` : allouer des données sur le device
  - `cudaMemcpy` : transférer des données vers le device ou vers l'hôte

Exemple avec CUDA

```
__global__
void AddVecKernel(double* A, double* B, double* C, int N)
{
  int col = threadIdx.x + blockDim.x * blockIdx.x;

  if((col < N))
  {
    C[col] = A[col] + B[col];
  }
}

int main(int argc, char** argv){
  int N;

  double *A;
  double *B;
  double *C;

  N = (argc < 2)?1000:atoi(argv[1]);

  // Memory allocation
  A = (double*) malloc(sizeof(double) * N);
  B = (double*) malloc(sizeof(double) * N);
  C = (double*) malloc(sizeof(double) * N);

  // Value initialization
  init(A, B, C, N);

  double *d_A, *d_B, *d_C;
  cudaMalloc(&d_A, sizeof(double) * N);
  cudaMalloc(&d_B, sizeof(double) * N);
  cudaMalloc(&d_C, sizeof(double) * N);

  cudaMemcpy(d_A, A, sizeof(double) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, sizeof(double) * N, cudaMemcpyHostToDevice);
  cudaMemcpy(d_C, C, sizeof(double) * N, cudaMemcpyHostToDevice);

  int nbBlocks = N / BLOCK_WIDTH;
  if(N % BLOCK_WIDTH) nbBlocks++;
  dim3 gridSize(nbBlocks);
  dim3 blockSize(BLOCK_WIDTH);

  AddVecKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);

  cudaMemcpy(C, d_C, sizeof(double) * N, cudaMemcpyDeviceToHost);

  free(A);
  free(B);
  free(C);

  cudaFree(A);
  cudaFree(B);
  cudaFree(C);

  return 0;
}
```

Overview CUDA

- Uniquement disponible pour GPUs NVIDIA (spécialisation de code)
- API bas niveau
  - Gestion explicite de la mémoire (allocation, transferts)
  - Configuration d'une grille de threads
  - Fonctionnalités spécifiques (Unified Shared Memory)
  - ...

  ⇒ Permet un contrôle fin de ce qui se passe sur GPU !

- Désavantages :
  - Programmation intrusive et difficile
  - Une version du code par accélérateurs et par génération d'accélérateurs
  - Difficulté de maintenabilité

- Problématique :
  - Comment suivre l'évolution des accélérateurs de types GPUs de façon pérenne et portable ?

Outline

1 Introduction aux accélérateurs de calculs

2 Architecture des cartes graphiques NVIDIA

3 Introduction à la Programmation GPU

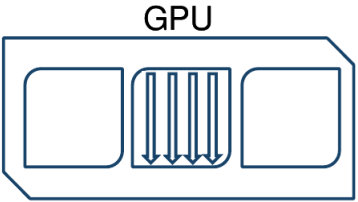
4 Programmation GPU avec OpenMP
- Modèle d'exécution des threads
- Gestion et Transferts de données
- Répartition des charges

5 Programmation GPU Avancée
- Accès Mémoire
- Recouvrement des Transferts mémoire

- OpenMP 4.0 – Juillet 2013
  - Introduction de la directive `target`
    - Déport de calculs sur un device
- OpenMP 4.0 – Juillet 2013
  - Asynchronisme CPU/GPU
    - Recouvrement communications/calculs
- OpenMP 5.0 – Novembre 2018
  - Support du Multi-Accélérateurs
  - Introduction de la directive `allocate`
    - Allocation de données dans mémoires hétérogènes
- OpenMP 6.0 – 2023 ?
  - Amélioration du support `device`
    - Meilleure coopération CPU/GPU
    - Amélioration du Multi-GPUs
- OpenMP : le modèle de programmation intra-noeud ?

OpenMP : Modèle d'exécution

- Analogie avec CUDA
  - Une grille,
  - De blocs,
  - De threads.
- Création d'un ensemble de threads organisés
  - Plusieurs directives à disposition
    - Une ligue,
    - D'équipes,
    - De Threads



Définition [OpenMP 5.1]

The **target** construct maps variables to a device data environment and executes the construct on that device.

Comportement

La région entourée par la directive target est

- déportée sur l'accélérateur (données + calculs),
- exécutée par un seul thread.

```
void func_GPU()
{
  #pragma omp target
  { // on GPU, 1 thread

    ...
  }
}
```

Principales clauses

- `device` (int)
  - Numéro du device sur lequel la région est déportée
- `map`
  - Spécifie les données qui doivent être transférées avant et après la région target
- `nowait`
  - La région target peut être exécutée de manière asynchrone

Informations à disposition !

- omp_get_num_procs
  - Retourne le nombre de processeurs disponibles sur le device courant
- omp_get_num_devices
  - Retourne le nombre de device "non hôtes" disponibles pour déporter des calculs
  - A utiliser depuis l'hôte ! (Comportement indéfini sur le device)
- omp_get_device_num
  - Retourne l'identifiant du device depuis lequel cette fonction est appelée
- omp_get_default_device
  - Retourne l'identifiant du device principal sur lequel les calculs sont déportés par défaut
- omp_set_default_device
  - Modifie le device principal sur lequel déporté du calcul par défaut (si la clause num_device n'est pas spécifiée)
- omp_is_initial_device
  - Retourne vrai si la fonction est appelée depuis le device initial (attention, le CPU est considéré comme un device)
- omp_get_initial_device
  - Retourne l'identifiant du device initial

Directive teams [OpenMP 5.1]

The teams construct creates a league of initial teams and the initial thread in each team executes the region.

Comportement effectif

- Création d'une ligue d'équipe
- Dans chaque équipe :
  - Un thread par équipe (thread initial),
  - Exécution de la région par chaque équipe de manière concurrente.
  - 1 équipe / SM

```
void func_GPU()
{
  #pragma omp target
  { // on GPU, 1 thread

    #pragma omp teams
    { // Ligue of teams, 1 thread / team

      ...
    }
  }
}
```

Principales clauses

- num_teams (int)
  - Nombre d'équipes dans la ligue
- thread_limit (int)
  - Définit le nombre de threads maximal par équipe
  - Nombre effectif inférieur ou égal

Obtenir des informations sur les équipes

- `omp_get_num_teams`
  - Renvoie le nombre d'équipes total
- `omp_get_team_num`
  - Renvoie l'identifiant de l'équipe courant
- `omp_get_max_teams`
  - Renvoie le nombre maximal d'équipe pouvant être créées
- `omp_get_teams_thread_limit`
  - Retourne le nombre maximal de threads OpenMP pouvant être créés dans une région `teams`

Prendre le contrôle sur la gestion des équipes

- `omp_set_num_teams`
  - Définit le nombre d'équipes créés lors de la rencontre d'une construction `teams`
  - Disponible uniquement depuis la spécification 5.1
- `omp_set_teams_thread_limit`
  - Définit le nombre maximal de threads OpenMP pouvant être créés dans une région `teams`

Directive `parallel` [OpenMP 5.1]

The parallel construct creates a team of OpenMP threads that execute the region.

Effets

Au sein d'une équipe :

- Crée un ensemble de threads au sein de celle-ci,
- S'exécutant sur un même SM, en parallèle.

Clauses principales

- num_threads (int)
- shared
- private / firstprivate / lastprivate

```
void func_GPU()
{
  #pragma omp target
  { // on GPU, 1 thread

    #pragma omp teams
    { // Ligue of teams, 1 thread / team

      #pragma omp parralel
      { // Team of threads
        ...
      }
    }
  }
}
```

Informations et Contrôles sur la région parallèle

- `omp_get_num_threads`
  - Renvoie le nombre de threads de la région parallèle courante
- `omp_set_num_threads`
  - Modifie le nombre de threads concourrants pour la prochaine région parallèle rencontrée
- `omp_get_thread_num`
  - Renvoie l'identifiant du thread courant dans une région parallèle
- ... Les mêmes routines que sur CPU !

Directive `simd`

The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently by using SIMD instructions).

Effets

1. Déroule une boucle,
2. La transforme via des instructions SIMD (e.g., AVX),
3. Exécution des instructions vectorisées.

- ... Quid du GPU ?
  - Pour l'instant :
    - Aucun effet
  - A terme :
    - Permettre l'exécution de threads sous forme de sous-warp

Transferts mémoire

- 2 environnements d'exécutions disjoints
  - Hôte,
  - Accélérateur.

- 2 espaces d'adressage virtuels différents
  - ⇒ Besoin de transférer les données !
    - Depuis le CPU vers l'Accélérateur
    - Depuis l'Accélérateur vers le CPU
    - Depuis l'Accélérateur vers un autre Accélérateur (Multi-Accélérateurs)

- Différents types de transferts
  - Explicites,
    - Clause `map`
  - Implicites.
    - Nécessite un support matériel de mémoire unifiée
    - Non abordé pendant ce cours

Définition [OpenMP 5.1]

La clause `map` détermine comment une variable d'origine (sur le device initial) dans un environnement mémoire est placée sur un device avec son environnement mémoire.
Elle s'applique principalement sur la directive `target`.

Différents types de mapping

- `alloc`
  - Alloue des données sur le device
- `to`
  - Alloue des données puis transfère les données sur le device
- `from`
  - Alloue des données puis transfère les données depuis le device
- `tofrom`
  - Alloue des données puis transfère depuis et vers le device

- Deux directives permettent de contrôler explicitement l'environnement de données d'un device grâce à la clause `map`

Directives `target enter data` / `target exit data`

La directive `target enter data` (resp. `target exit data`) spécifie que les variables sont mappées (resp. démappées) à un environnement de données sur le device.
Lorsque la construction est rencontrée, les éléments de la liste sont mis en correspondance avec l'environnement de données du device conformément à la sémantique de la clause `map`.

```
#pragma omp target enter data map(to: A[0:N],B[0:N],C[0:N])

  #pragma omp target
    {do lots of stuff with A, B and C}

    {do something on the host}

  #pragma omp target
    {do lots of stuff with A, B, and C}

#pragma omp target exit data map(from: C[0:N])
```

- `target enter data`
  - Les données peuvent être allouées et/ou copiées sur le device
- `target exit data`
  - Les données peuvent être étruites et/ou recopiées sur le host depuis device

Mise à jour des données via la directive `target update`

- Les données peuvent être mises à jour entre les régions `target`
  - ⇒ Directive `target update`

```
#pragma omp target enter data map(to: A[0:N],B[0:N],C[0:N])

  #pragma omp target
    {do lots of stuff with A, B and C}

    #pragma omp target update from(A[0:N])
    // Copy A from the device to A on the host

    {do something on the host}

    #pragma omp target update to(A[0:N])
    // Copy A from the host to A on the device

  #pragma omp target
    {do lots of stuff with A, B, and C}

#pragma omp target exit data map(from: C[0:N])
```

Répartition du calculs sur l'Accélérateur

- Les threads sur l'Accélérateur sont actifs,
- Les données sont transférées...

⇒ Partageons le travail sur l'ensemble des threads !

- Analogie avec CUDA
  - Chaque thread lance un noyau de calcul (i.e. un kernel)
    - Programmation par défaut multi-threadée,
    - Raisonnement par threads, non par boucle ou par itérations de boucles.
  - Directives pour partager le travail
    - `distribute`
    - `for`

Définition [OpenMP 5.1]

La construction spécifie que les itérations d'une ou plusieurs boucles seront exécutées par l'équipe initial dans le context de leur tâche implicite. Les itérations sont distribuées à travers les threads initiaux de toutes les équipes initiales qui exécute la région `teams` à laquelle la région `distribute` appartient.

En d'autres mots...

- Distribue les itérations d'une loupe sur chaque thread master des équipes exécutant la région `distribute`
- Pas de barrière implicite en fin de région

Principales clauses

- private, firstprivate, lastprivate
- collapse (int)
  - Spécifie combien de boucles sont associées à la directive `distribute`

```
#pragma omp target teams map(to:n)
{
  /* executed by the master thread of each team */
  int block_size = n/omp_get_num_teams();

  /* split iteration domain across teams master thread */
  #pragma omp distribute
  for (int i = 0; i < n; i += block_size)
  ...
}
```

Définition [OpenMP 5.1]

La construction spécifie que les itérations d'une ou plusieurs boucles associées seront exécutées en parallèle par les threads de l'équipe courante. Les itérations seront distrbuées parmi les threads qui existent déjà dan l'équipe, qui exécutent la région parallèle dont la construction `for` appartient.

Comportement sur GPU

Cette construction permet de distribuer les itérations d'une boucle à l'ensemble des threads composant une équipe, donc d'un SM.

Clauses principales

- Les mêmes que pour l'utilisation du CPU
  - Worksharing (private, firstprivate,. . .)
  - collapse
  - reduction
  - scheduling

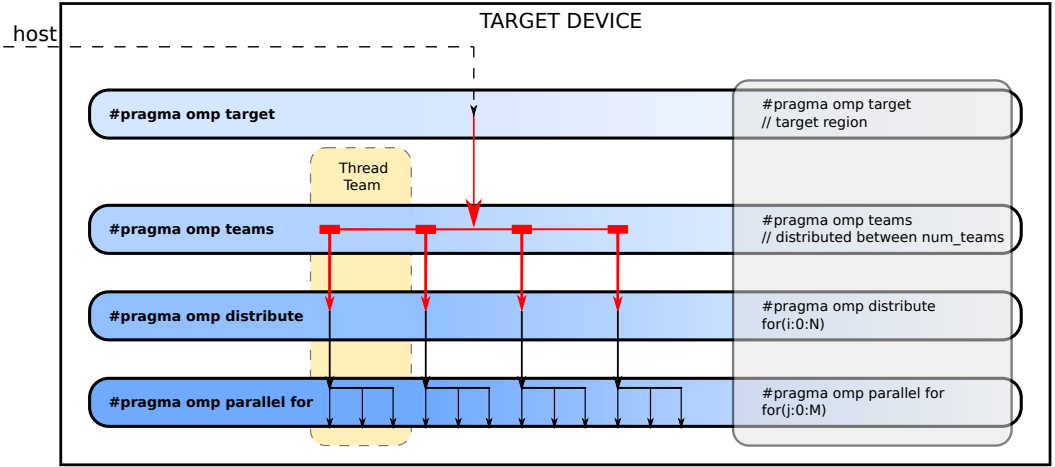
```
#pragma omp target teams map(to:n)
{
  /* executed by the master thread of each team */
  int block_size = n/omp_get_num_teams();

  /* split iteration domain across teams master thread */
  #pragma omp distribute
  for (int i = 0; i < n; i += block_size)
  {

    #pragma omp parallel for /* share work within a team */
    for (int j = i; j < i + block_size; j++) {...}
  }
}
```

Constructions imbriquées

- parallel for
  - Création d'un ensemble de threads dans l'équipe courante et partage un ensemble d'itérations d'une ou plusieurs boucles.
- teams distribute
  - Création d'une ligue d'équipe, et partage l'ensemble des itérations d'une ou plusieurs boucles sur les différentes équipes.
- teams distribute parallel for
  - Création d'une ligue d'équipe, et partage l'ensemble des itérations d'une ou plusieurs boucles sur les différentes équipes.
  - Puis, création d'un ensemble de threads dans l'équipe courante et partage un ensemble d'itérations d'une ou plusieurs boucles.


host
TARGET DEVICE
#pragma omp target
#pragma omp target
// target region
Thread
Team
#pragma omp teams
#pragma omp teams
// distributed between num_teams
#pragma omp distribute
#pragma omp distribute
for(i:0:N)
#pragma omp parallel for
#pragma omp parallel for
for(j:0:M)

Exemple : Addition de vecteurs

```
void addVec(float * y, float * x, int n) {
  /* the following code region will be offloaded and
  a league of threads teams will be created */
  #pragma omp target teams map(to:n,x[:n]) map(tofrom y[:n])
  {
    /* executed by the master thread of each team */
    int block_size = n/omp_get_num_teams();

    /* split iteration domain across teams master thread */
    #pragma omp distribute
    for (int i = 0; i < n; i += block_size)

    #pragma omp parallel for /* share work within a team */
    for (int j = i; j < i + block_size; j++)
    {
      y[j] = x[j] + y[j];
    }
  }
}
```

```
int main(int argc, char** argv)
{
  double *X = NULL, *Y = NULL;
  int N = 1000;
  if (argc > 1) N = atoi(argv[1]);

  X = (double*) malloc(sizeof(double) * N);
  Y = (double*) malloc(sizeof(double) * N);

  for(int i = 0; i < N; ++i)
  {
    X[i] = i;
    Y[i] = X[i] + i;
  }

  addVec(Y, X, N);

  return 0;
}
```

Outline

1 Introduction aux accélérateurs de calculs

2 Architecture des cartes graphiques NVIDIA

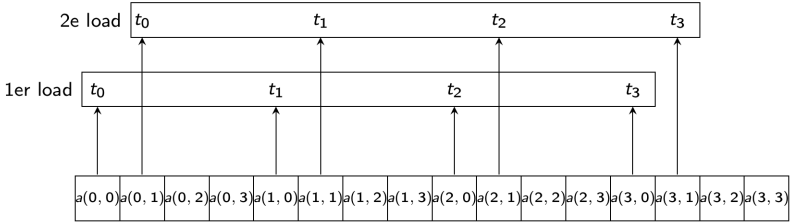
3 Introduction à la Programmation GPU

4 Programmation GPU avec OpenMP
- Modèle d'exécution des threads
- Gestion et Transferts de données
- Répartition des charges

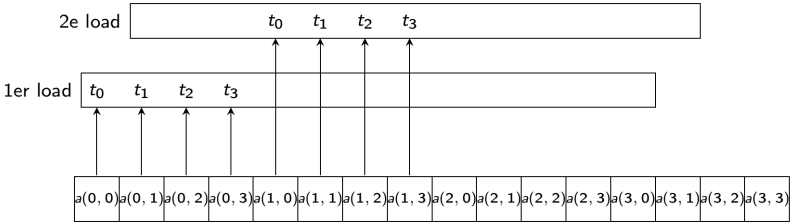
5 Programmation GPU Avancée
- Accès Mémoire
- Recouvrement des Transferts mémoire

- Plusieurs facteurs importants de performance sont à prendre en compte
  - Accès mémoire
    - Réduire leur fréquence d'apparition,
    - Uniquement ce qui est nécessaire,
    - Minimiser les transferts de faible BW
  - Schéma d'exécution
    - Optimisation de l'ordre des instructions
    - Recouvrement communications par du calcul
  - Structures de données
    - Les plus régulières possibles
    - Liées au schéma d'accès mémoire

- Lorsque tous les threads font un load, le hardware détecte si les threads accèdent à un espace mémoire contigu
- Dans ce cas, le hardware groupe (coalesces) les accès en un seul accès à différentes location de la DRAM
- Les accès concurrents des threads d'un warp se regroupent en un nombre de transactions égal au nombre de transactions de 128bits (= taille de ligne de cache) nécessaires pour desservir tous les threads du warp.



- Lorsque tous les threads font un load, le hardware détecte si les threads accèdent à un espace mémoire contigu
- Dans ce cas, le hardware groupe (coalesces) les accès en un seul accès à différentes location de la DRAM
- Les accès concurrents des threads d'un warp se regroupent en un nombre de transactions égal au nombre de transactions de 128bits (= taille de ligne de cache) nécessaires pour desservir tous les threads du warp.
  - + Regrouper le plus possible les données touchées par un même warp au même instant !
  - + Via l'utilisation de la clause `omp parallel for` <u>`schedule(static,1)`</u>

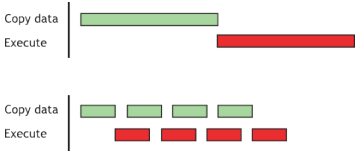


Définition

L'asynchronisme réfère au phénomène pour deux unités de calculs distinctes (e.g., deux coeurs de calculs ou un CPU et un GPU) ne pas être obligé de s'attendre mutuellement pour pouvoir exécuter du travail en parallèle.

Analogie avec CUDA

- Utilisation de **Streams**
  - Un Stream est une séquence d'opérations qui sont exécutées dans l'ordre défini sur le device.
  - Les opérations sur un même stream sont ordonnées (FIFO) et synchrones
  - Les opérations dans des streams différents peuvent s'exécuter en parallèle
  - Les opérations entre les différents streams peuvent s'entrelacer
- Des propriétés qui peuvent être utilisées pour cacher les transferts mémoires entre l'hôte et le device

Quid de OpenMP ?

- Utilisation de la clause **nowait**
  - directive `target`