Slurm Workload Manager
Soumission de jobs sur un cluster de calcul

Workload manager

Il existe plusieurs gestionnaires de queue (Slurm, SGE, Condor,...).
Le principe de ces gestionnaires est de permettre à plusieurs utilisateur de se **partager les ressources d'un même système.**

Un utilisateur doit donc "soumettre" un job en précisant les ressources dont il a besoins. Ce job est alors mis en file d'attente jusqu'à ce que les ressources demandées soient disponibles.

Il est ensuite possible de définir des priorités : les jobs légers sont prioritaires, quota d'heures d'utilisation, quota d'utilisation mémoire, ...

Script Slurm

```
#!/bin/bash
#
#SBATCH --job-name=test_omp
#SBATCH --output=out_slurm.txt
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=12
#SBATCH --time=1:00:00
#SBATCH --mem-per-cpu=1G
#SBATCH --gres=gpu:1

export OMP_NUM_THREADS=12
./blas
```

Le script ci dessus permet de réserver 12 coeurs CPU et 1 GPU puis exécute le programme "blas". Ce script peut être soumis avec la commande **sbatch script.sh**.

Il est possible de surveiller la progression de ses jobs avec la commande **squeue**.

En cas d'erreur un job peut être annulé avec **scancel id**.

Enfin, il est aussi possible d'ouvrir un shell interactif avec une quantité de ressource voulue avec la commande **salloc** qui prend en parametre les mêmes arguments que le script de soumission.

Sauf cas exceptionnel : Ne jamais exécuter un programme directement sur un serveur de calcul géré par un gestionnaire de queue avec un "./"

GP-GPU computing
Introduction à CUDA

D. Cornu

M2-P2N Automne - 2019

Introduction

Les **GPU**, ou **Graphics Processing Unit**, sont des processeurs de calculs spécialisés dans la manipulation d'images et principalement destinés à l'affichage. Leur objectif étant de faire des manipulations "identiques" sur une grande quantité de pixels (ou de sommets en 3D), leur architecture repose fortement sur le principe **"SIMD"**.

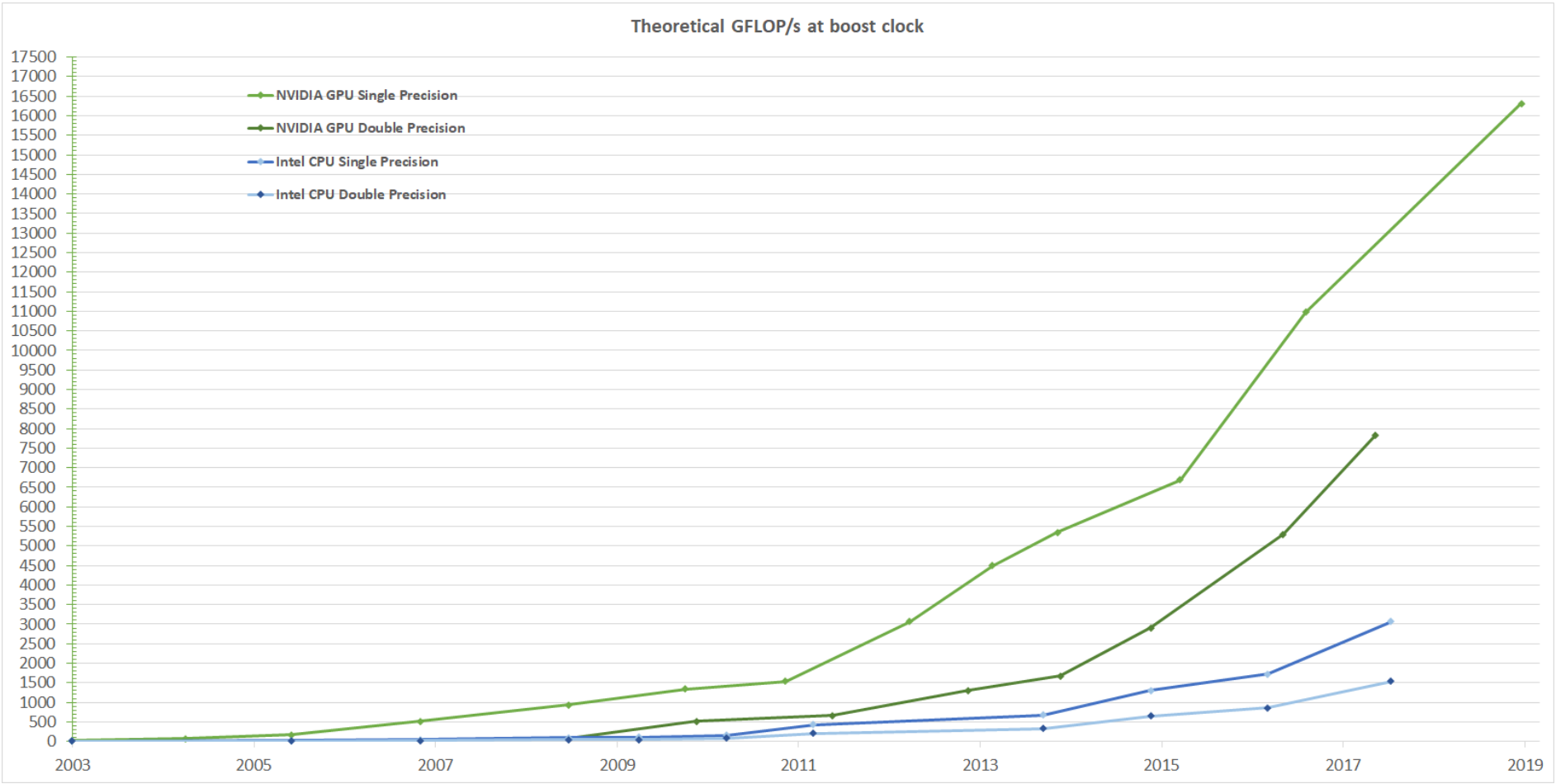
Propriétés physiques :

- Un nombre important de "petit" coeurs (jusqu'à 5000 sur GPU Pro)
- Quantité "importante" de mémoire par GPU (jusqu'à 32 Gos sur GPU Pro)
- Une mémoire très rapide et interfacée sur un "bus" très large (4096 bits en HBM2, bande passante pouvant atteindre le TB/s)

Il est possible de tirer avantage des ces capacités pour du calcul scientifique, on parle alors de **GPGPU, ou General-Purpose GPU.** Dans le cas de calculs hautement parallèles on peut alors espérer des performances significativement meilleurs que sur un CPU multi-coeurs.

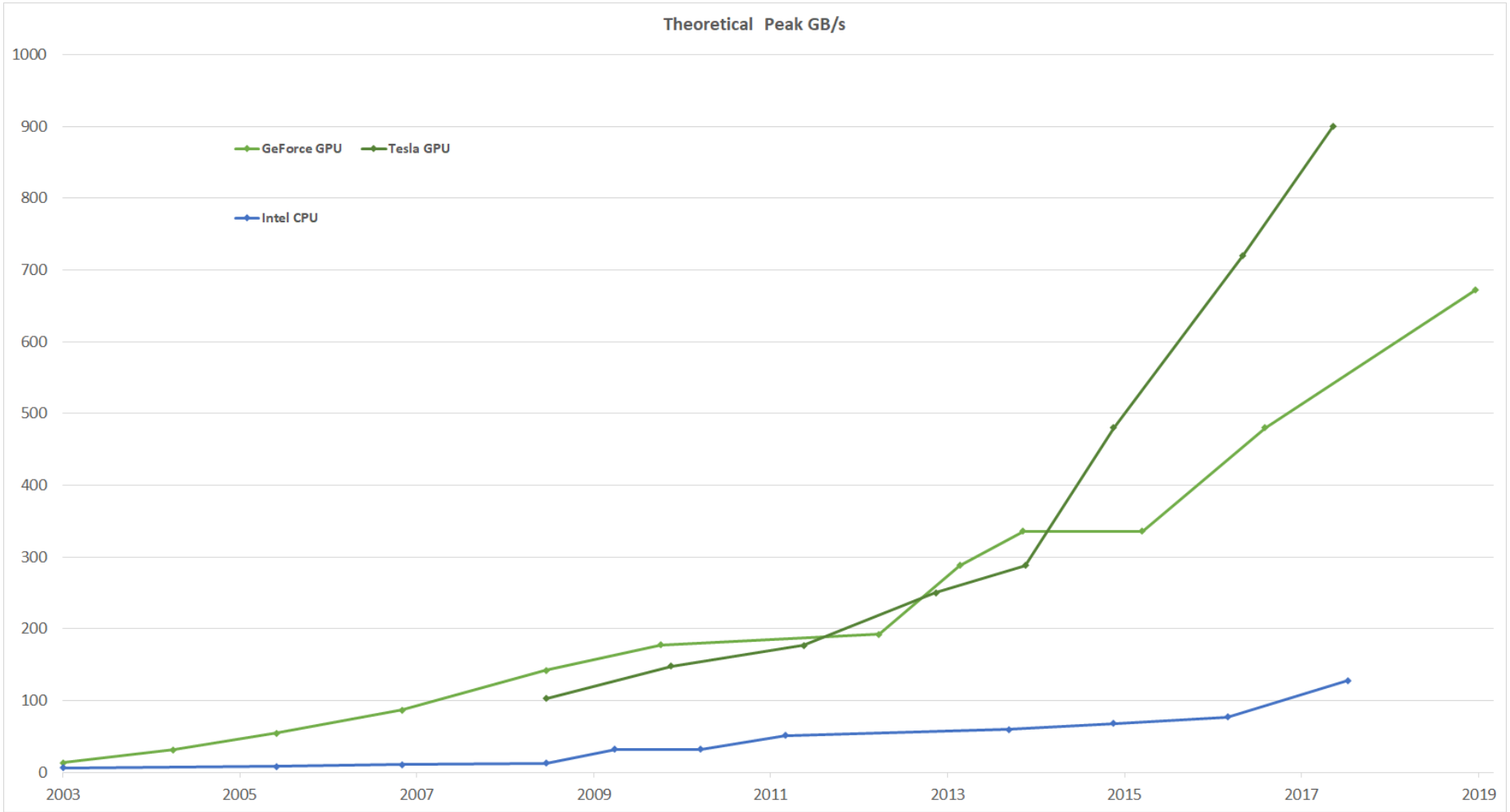
Trouver de la documentation :
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Comparaison des performances



Figures provenant de la documentation CUDA / Nvidia

Comparaison des performances



Figures provenant de la documentation CUDA / Nvidia

Programmer en GPGPU

Il existe plusieurs façon d'utiliser les capacités GPGPU, toutes ou presque sous la forme d'un **langage ou d'une extension de langage**.
Les deux plus répandues sont toutes deux dérivées du C standard :

CUDA

- Gratuit mais propriétaire (Nvidia)
- Compatible uniquement avec les GPU Nvidia
- Optimisation spécifique, à la pointe des technologies (ML, ...)

OpenCL

- Open source
- Compatible avec tous les GPU, **ET CPU** (au sens large)
- Moins optimisé par défaut, souvent en retard sur les technologies

Programmer en GPGPU

Il existe plusieurs façon d'utiliser les capacités GPGPU, toutes ou presque sous la forme d'un **langage ou d'une extension de langage**.
Les deux plus répandues sont toutes deux dérivées du C standard :

CUDA

- Gratuit mais propriétaire (Nvidia)
- Compatible uniquement avec les GPU Nvidia
- Optimisation spécifique, à la pointe des technologies (ML, ...)

OpenCL

- Open source
- Compatible avec tous les GPU, **ET CPU** (au sens large)
- Moins optimisé par défaut, souvent en retard sur les technologies

Les GPU sont aussi appelé des "accélérateurs graphique", il s'agit donc d'un composant additionnel dans une machine standard. Il faut donc toujours faire appel à un CPU hôte. GPU et CPU peuvent d'ailleurs travailler de concert sur un même problème, chacun gérant une partie du calcul plus adaptée à son architecture.

Programmer en GPGPU

Il existe plusieurs façon d'utiliser les capacités GPGPU, toutes ou presque sous la forme d'un **langage ou d'une extension de langage**.
Les deux plus répandues sont toutes deux dérivées du C standard :

CUDA

- Gratuit mais propriétaire (Nvidia)
- Compatible uniquement avec les GPU Nvidia
- Optimisation spécifique, à la pointe des technologies (ML, ...)

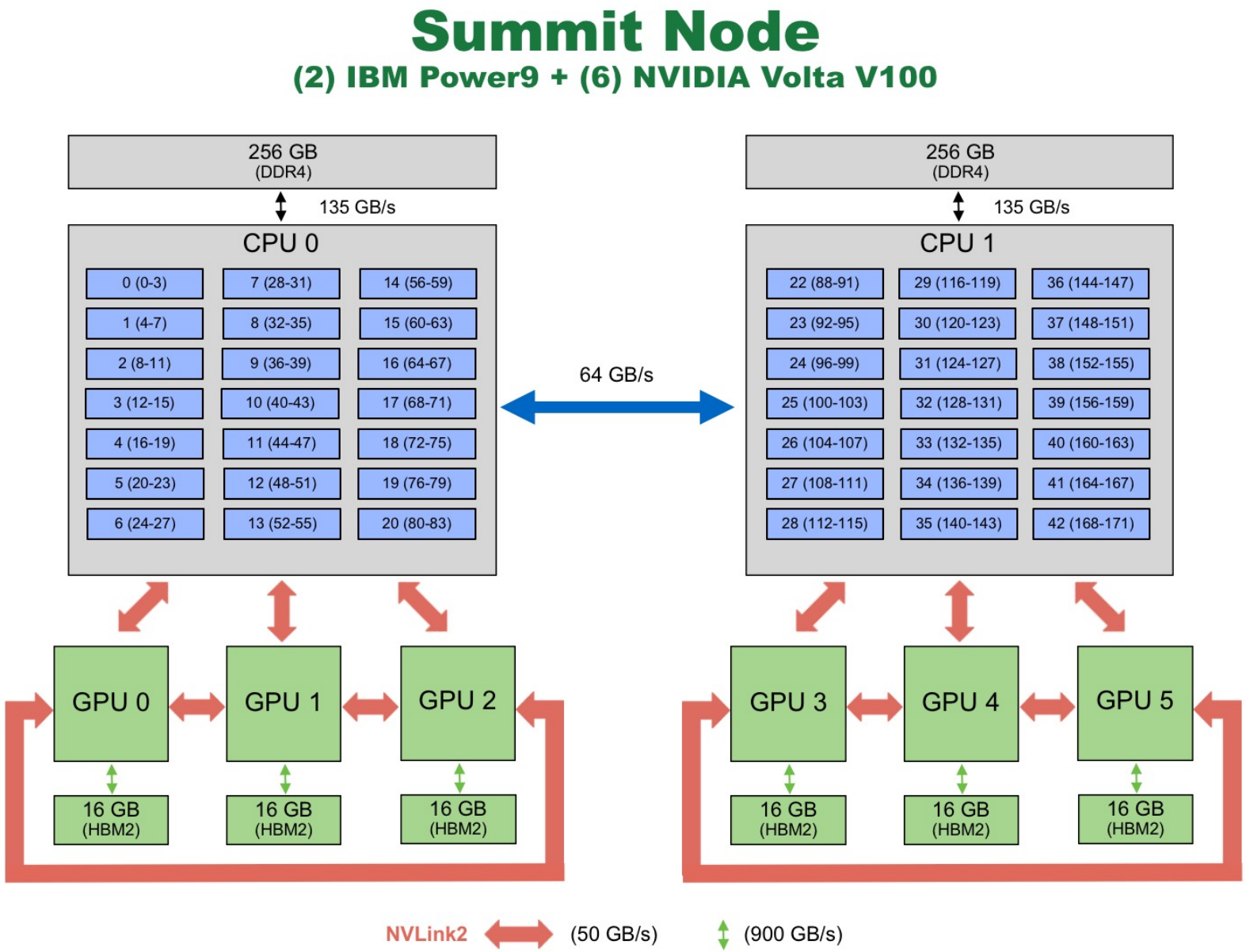
OpenCL

- Open source
- Compatible avec tous les GPU, **ET CPU** (au sens large)
- Moins optimisé par défaut, souvent en retard sur les technologies

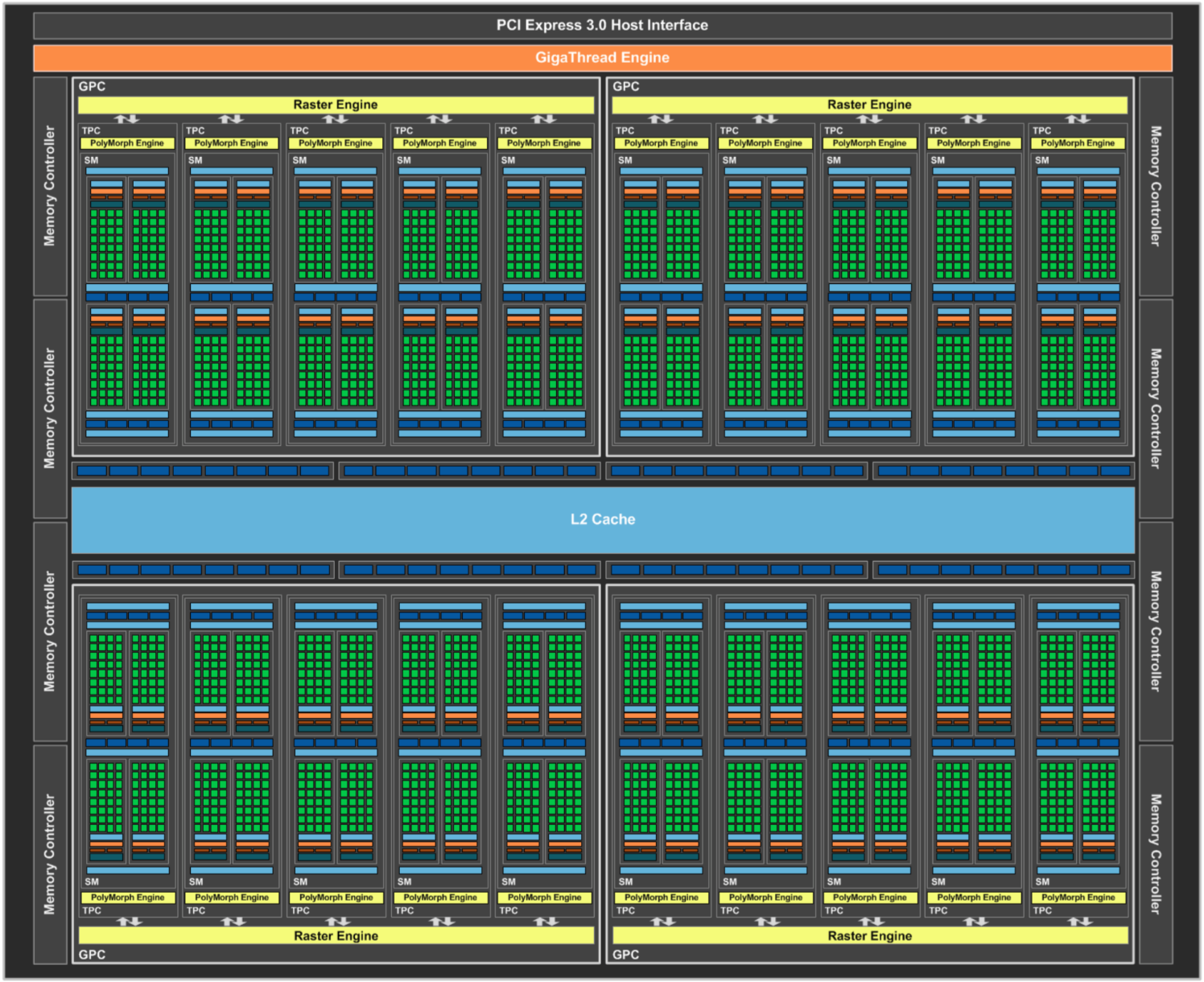
Les GPU sont aussi appelé des "accélérateurs graphique", il s'agit donc d'un composant additionnel dans une machine standard. Il faut donc toujours faire appel à un CPU hôte. GPU et CPU peuvent d'ailleurs travailler de concert sur un même problème, chacun gérant une partie du calcul plus adaptée à son architecture.

Plusieurs GPU peuvent être intégrés sur une même machine à mémoire partagée, ils peuvent donc être utilisés ensemble sur une même tache via des communications à travers "l'hôte". Plusieurs hôtes peuvent ensuite communiquer (ex : MPI), et chacun d'entre eux peut être équipé de plusieurs GPU.

Cluster de calcul modern : Multi-Nodes Multi-GPU



Architecture interne GPU Nvidia



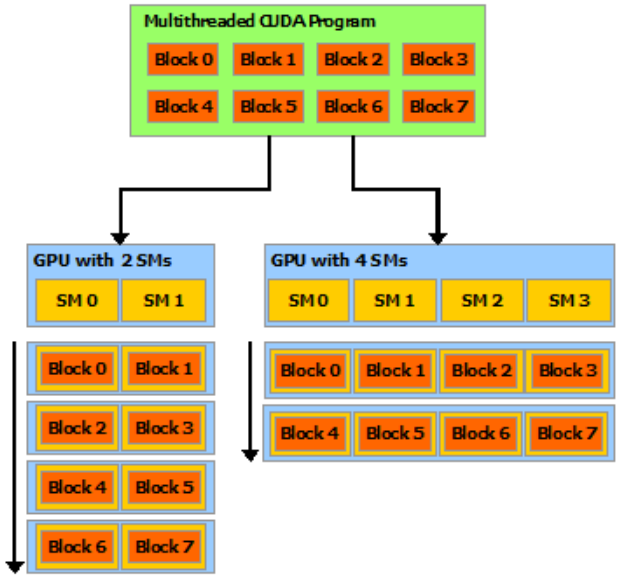
Exemple d'architecture d'un GPU Nvidia (GP 104 - Tesla P100)

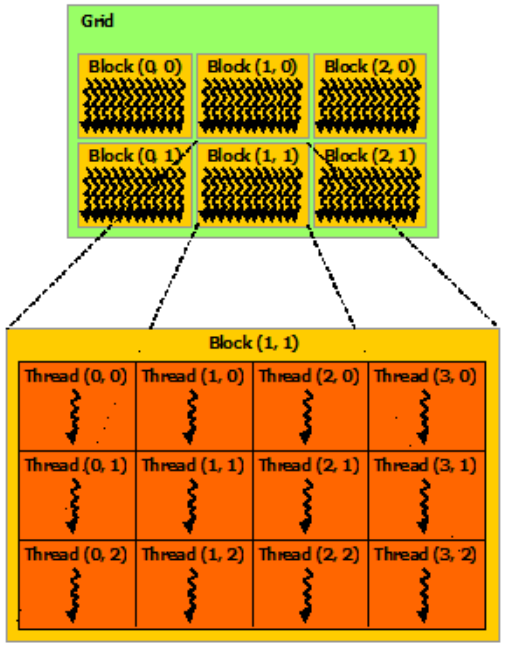
Le GPU est équipé d'une mémoire partagée par tous les SM qui partagent tous un même cache L2.

Chaque SM est équipé de son propre cache et peut lancer simultanément plusieurs "blocs d'instructions" SMPs contenant chacun 32 coeurs CUDA (128 coeurs / SM).

Architecture interne GPU Nvidia

NVIDIA utilise une architecture specifique reposant sur leur propre paradigme : Single instruction, multiple thread (SIMT).

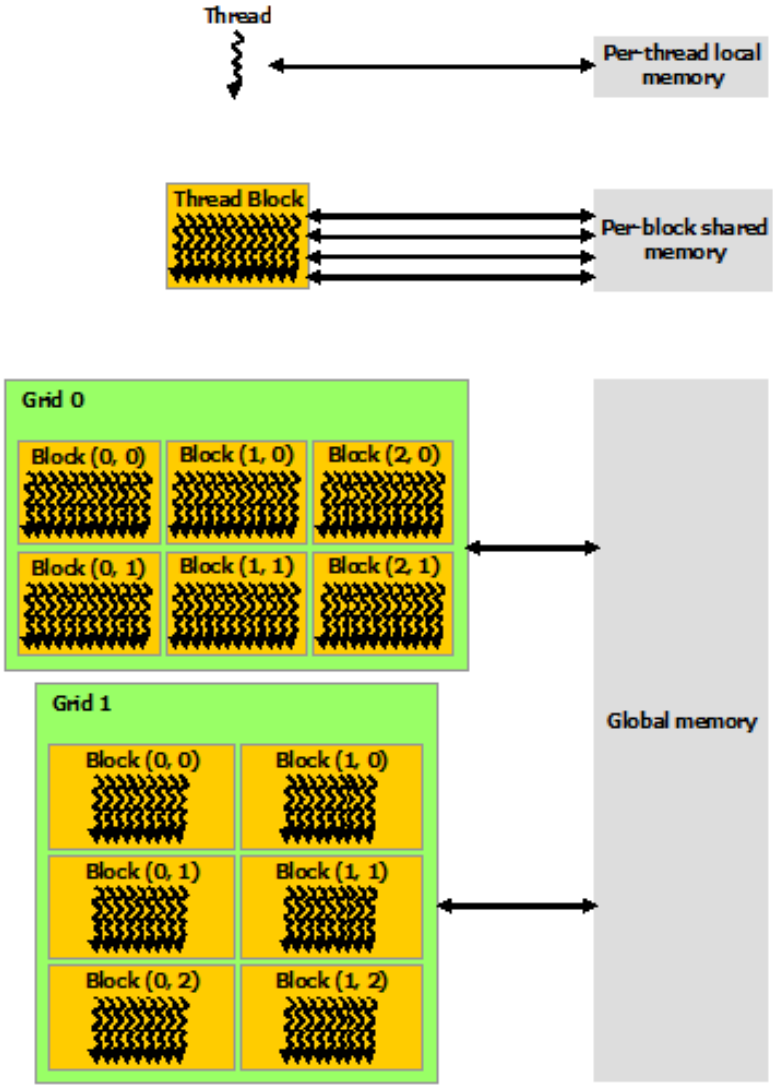




Au niveau materiel les SM (Streaming Multiprocessor) sont les unités physique de parallélisation. Ils peuvent ensuite executer un certains nombre de "blocs" de manière simultanée (ici 1) qui contient lui même des **"CUDA cores"**

Attention : Ne pas confondre les SM, blocs d'instructions (SMPs), et CUDA cores avec leur équivalent logiciel blocs et threads représentés dans la figure ci-dessus.

Organisation Mémoire



En plus des niveaux de caches standards (L2, ...), les GPU Nvidia ont plusieurs niveaux de mémoire spécifiques à leur architecture.

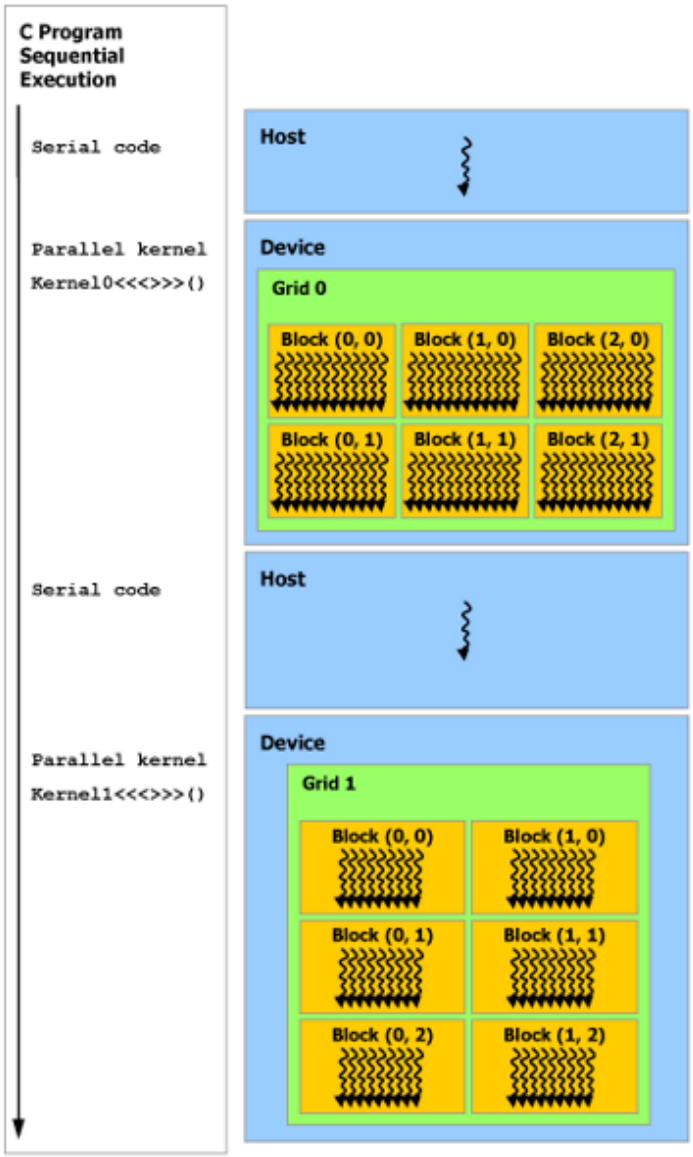
La mémoire globale : Mémoire la plus "lente", présente en grande quantité, qui est accessible depuis n'importe où depuis le GPU (similaire à la mémoire RAM).

La mémoire "partagée" : Mémoire plus rapide visible par tous les threads d'un même "bloc", présente en très faible quantité (96KB sur un GP104)

La mémoire locale : Équivalent à la mémoire partagée mais dont la visibilité est limitée à un seul thread.

Dans les versions modernes de CUDA il existe des déclarations mémoires qui sont partagées par l'host (CPU) et le(s) device(s) (GPUs). L'accès à ce type de mémoire induit des copies automatiques CPU $\leftrightarrow$ GPU.

Programmation "hétérogène"



La programmation CUDA est dite **hétérogène**. Les instruction CUDA sont exécutées parallèlement sur un *device*, qui se comporte donc comme un **co-processeur pour l'hôte.**

L'hôte exécute une série d'instructions puis fait appel à une instruction CUDA qui est envoyé vers le *device*. Une fois l'instruction terminée celui ci indique à l'hote qu'il peux continuer.

Dans la majorité des cas, l'hôte continue en fait son exécution jusqu'à la prochaine instruction CUDA → comportement non bloquant !

Ce fonctionnement s'explique par le fait que **seul certains types d'instructions (SIMD) peuvent être exécutées efficacement sur un GPU**. Une partie des taches reste donc à la charge du CPU.

Un code complet qui contient plusieurs appels à CUDA fonctionne donc sur un principe d'aller et retour entre l'hôte et le *device*.

Serveur de calcul du CompuPhys

Le master compuphys vient de faire l'acquisition d'un serveur de calcul avec les **caractéristiques suivantes :**

- Bi-CPU : Intel Xeon 4116 2.1-3.0 GHz, 12 coeurs / 24 threads - Soit un total de 24C / 48T
- 64 Go de ram DDR4 2666 MHz
- **1 GPU Nvidia Quadro RTX 5000 (Turing)** : 3072 cuda cores, 16 Go de ram (GDDR6-256bits), 11.15 TFLOPS FP32 (22.30 TFLOPS FP16), 384 tensor cores for ML

Ce serveur étant à vocation pédagogique, il est destiné à être utilisé **pour les cours de HPC et de ML**, mais peut également faire l'objet de demandes des étudiants du Master pour être utilisé dans le cadre de **projets**.

Par ailleurs ce serveur est également utilisé à des fins de développement d'outils de recherche en Machine Learning et calcul GPU.

Rappels de C

Les codes CUDA sont dérivés du C et les fichiers source ont généralement une extension ".cu". **Quelques éléments de langage C :**

```c
#include <stdlib.h>
//Contains general purpose funcitons
#include <stdio.h>
//Contains usual Input/Output functions

//Main function of a C program, the "main" name is mandatory
//the program will start with this function
int main()
{
  //an instruction end with a ";" character
  int i;//declaration of a integer type
  int i2 = 0;//declaration and affectation of an integer type

  float f;
  float f2 = 3.6f; //the f specify the simple precision of the real 3.6
  //f is not mandatory, an automatic conversion will be done from double to float

  double d;
  double d2 = 3.6; //double is the default presion of real numbers

  //only affectations
  i = 1;
  f = 4.5f; d = 4.5;
  //Note that multiple instruction can be given on one line

  //usual function to print something on the standard output
  printf("My integers: %d %d, my floats: %f %.2f, my doubles: %lf %g \n", i, i2, f, f2, d, d2);
  // the % are "specifiers" allowing to print numbers "d" is integer, "f" is float, "d" is double
  // "lf" is double, "g" is any real with automatic format, "[.precision]f" specify the printed precision
  // The function adapt the number of argument based on the number if specifers given.
  // "\n" specify a line break, which is not automated
  for(i = 0; i < 20; i++)
  {
    printf("i value : %d\n", i);
  }
  //Instruction block are marked with "{" and "}"
  //note that there is no ";" at the end of the "for" instruction line !

  //basic condition
  if(f2 > 1.0)
  {
    printf("Condition > 1.0\n");
  }
  else if(f2 >= 0.0)  //else if can be stacked any number of time
  {
    printf("Condition < 1.0 AND >= 0.0\n");
  }
  //all remaining cases
  else
  {
    printf("No matching case\n");
  }

  //Combined condition
  if(f > 2.4 && d < 12.6)
  {
    printf("Double condition checked successfully\n");
  }

  //Or statement
  if(f > 2.4 || d < 12.6)
  {
    printf("One of the combined condition is fulfilled\n");
  }


  exit(EXIT_SUCCESS);
  //Usual way of exiting properly the program, not mandatory
}
```

"Hello World"

Le compilateur CUDA est appelé avec la commande **"nvcc"**. Cette commande prend en compte la majorité des options du compilateur gcc standard. Vous pouvez donc compiler un code CUDA avec :

```
nvcc -O3 hello_world.cu -o hello_world
```

Exercice 1 :

Écrivez un programme hello_world.cu qui affiche quelque chose sur la console. Compilez le avec *nvcc* et exécutez le.

Kernel CUDA

Un **kernel** est une fonction spécifique à CUDA qui, lorsqu'elle est appelée, va s'exécuter N fois en parallèle sur N threads CUDA différents.

Un kernel se construit de la même façon qu'une fonction C standard mais est précédé de l'argument "__global__" qui indique qu'elle peut être appelée depuis l'hôte et s'exécutera sur le GPU.

Un kernel est ensuite appelé différemment d'une fonction. Il faut ajouter " $<<< M, N >>>$ " entre son nom et ses arguments. Avec M le nombre de blocs, et N le nombre de threads par bloc.

```
//...
__global__ void my_kernel(void)
{
  //do something
}

int main()
{
  //...
  my_kernel<<< 1 , 16 >>>();
  cudaDeviceSynchronize(); //used to force synchronization
  //...
}
```

Hello World kernel

```
//...
__global__ void my_kernel(void)
{
  //do something
}

int main()
{
  //...
  my_kernel<<< 1 , 16 >>>();
  cudaDeviceSynchronize(); //used to force synchronization
  //...
}
```

Exercice 2 :

Inspirez vous du kernel précédent pour écrire un kernel qui produit un affichage de type "hello world".
Exécutez votre programme.

Hello World kernel

Pour la majorité des opérations il sera nécessaire de connaître l'indice du thread en cours dans le kernel.
Cette valeur est contenue dans la variable "threadIdx.x".

```
//...
__global__ void my_kernel(void)
{
  int i = threadIdx.x;
  printf("Hello from thread : %d\n", i);
}

int main()
{
  //...
  my_kernel<<< 1 , 16 >>>();
  cudaDeviceSynchronize(); //used to force synchronization
  //...
}
```

Hello World kernel

Le nombre de thread par bloc est limité à 1024 (dépend de la version de CUDA). Pour toute opération de plus de 1024 élément il est nécessaire de décomposer en blocs. L'indice du bloc peut alors etre récupéré avec via "blockIdx.x".

```
//...
__global__ void my_kernel(void)
{
  int i = threadIdx.x;
  int b = blockIdx.x;

  printf("Hello from block : %d, threadId : %d\n", b, i);
}

int main()
{
  //...
  my_kernel<<< 4 , 16 >>>();
  cudaDeviceSynchronize(); //used to force synchronization
  //...
}
```

Hello World kernel

Afin que les blocs partagent réellement le travail il est souvent nécessaire de combiner l'indice du blocs et du thread en cours en utilisant la dimension des blocs "blockDim.x"

```
//...
__global__ void my_kernel(void)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int b = blockIdx.x;

  printf("Hello from block : %d, threadId : %d\n", b, i);
}

int main()
{
  //...
  my_kernel<<< 4 , 16 >>>();
  cudaDeviceSynchronize(); //used to force synchronization
  //...
}
```

Accès à la mémoire GPU

Comme pour toute fonction, les variables déclarées dans un kernel sont **locales au thread** qui l'exécute. Pour que le travail effectué soit persistent il faut utiliser la mémoire globale du GPU.

La mémoire GPU ne peut être modifiée que via des fonctions *device* (i.e des kernels). De plus, il n'est pas facile de gérer de la lecture de fichier depuis les kernels.

A l'inverse la mémoire GPU est alloué (sur le GPU) **depuis l'hôte**.

En pratique les données sont chargées sur l'hôte puis envoyées vers le GPU pour que celui ci puisse s'en servir et les modifier, puis rapatriées vers l'hôte.

Lecture → transfert vers GPU → Calcul GPU → transfert vers CPU → écriture ...

Accès à la mémoire GPU

```
//...
int *A, *d_A; //declare an int memory pointer

A = (int*) malloc(64 * sizeof(int));
//allocate a 64 int element array on host memory and store the location inside an int
     pointer

cudaMalloc(&d_A, 64*sizeof(int))
//allocate a 64 int element array on GPU memory and store the location inside a host
     int pointer

for(int i = 0; i < 64; i++)
{ //do something with the table A
  A[i] = i;
}

cudaMemcpy(d_A, A, 64*sizeof(int), cudaMemcpyHostToDevice);
//copy the host allocated table A into the GPU allocated table d_A

my_kernel <<< ... >>>(d_A);

cudaMemcpy(A, d_A, 64*sizeof(int), cudaMemcpyDeviceToHost);
//copy the GPU allocated table d_A into the the host allocated table A

free(A); //regular C free table
cudaFree(d_A); //cuda GPU memory free

//...
```

Opération sur mémoire GPU

Exercice 3 :

Ecrivez un code qui :

- Alloue les tableaux utiles aux points ci-dessous
- Initialise un tableau A à une valeur quelconque
- Transfert le contenu de ce tableau A dans un tableau d_A sur le GPU
- Effectue une transformation de ce tableau à l'aide d'un kernel simple qui prend d_A comme argument
- Récupère le tableau d_A sur l'hôte et affiche son contenu modifié

Note : les GPU sont équipés de coeurs physiques dédiés à un type de précision spécifique. i.e : lors d'un calcul en double précision les coeurs simple précision ne sont pas utilisés. Les GPU ayant une architecture tournée vers la simple précision, dans la majorité des cas les performances en double précision sont significativement plus faibles ($\div 32$) et ne doivent être utilisés que pour une partie très sensible des calculs. Certaines cartes entièrement dédiées au calcul scientifique (gamme Tesla pour Nvidia) réduisent cette perte de performance à un facteur 2 (minimum possible).

Opération sur mémoire GPU

```
#include <stdio.h>
#include <stdlib.h>


__global__ void add_kernel(int *tab, int
     size)
{
  //int i = threadIdx.x;
  int i = blockIdx.x * blockDim.x +
       threadIdx.x;

  if(i < size)
  {
    tab[i]*=2;
  }

}


int main()
{
  int i;
  int N = 64;
  int *table, *device_table;

  table = (int*) malloc(N*sizeof(int));
  for(i = 0; i < N; i++)
  {
    table[i] = i;
  }

  cudaMalloc(&device_table, N*sizeof(int));
  cudaMemcpy(device_table, table, N*sizeof(
       int), cudaMemcpyHostToDevice);

  add_kernel<<< 1, N>>>(device_table, N);

  cudaMemcpy(table, device_table, N*sizeof(
       int), cudaMemcpyDeviceToHost);

  for(i = 0; i < N; i++)
  {
    printf("%d\n",table[i]);
  }


  free(table);
  cudaFree(device_table);

  exit(EXIT_SUCCESS);
}
```

Confrontation d'un calcul simple CPU vs GPU

Pour comparer les performances entre CPU et GPU il faut s'assurer de ne **mesurer que le temps de calcul** (et non pas les temps de gestion mémoire).

Remarque : pour une évaluation réel des performances il sera bien sur pertinent à partir d'un moment de prendre en compte le cout des opérations mémoire !

Le comportement **Asynchrone** des opération CUDA rend la mesure du temps d'exécution délicate. Dans un premier temps il est possible d'utiliser "cudaDeviceSynchronize" après un kernel et avant la mesure du temps.

Vous trouverez dans les ressources "timer.c" du cours deux fonctions qui permettent de mesurer finement le temps entre deux balises en C.

```
//...
struct timeval timer;
//...
init_timing(&timer);

my_kernel<<< ... >>>(...);
cudaDeviceSynchronize();

printf("Kernel time : %f\n", ellapsed_time(timer));
//...
```

Confrontation d'un calcul simple CPU vs GPU

Exercice 4 :

Reprenez la structure de l'exercice 3 en ajoutant :

- Complexifiez votre kernel pour alourdir le calcul (division, comparaisons, ...)
- Mesurez le temps d'execution de votre kernel
- Résolvez le même calcul sur le CPU (sur tout un tableau également) et mesurez le temps

Pour rendre votre calcul indépendant à la taille du tableau :

- Définissez un nombre de threads par bloc (exemple 128,256, ... < 1024)
- Calculez le bon nombre de blocs en fonction de votre taille de tableau
- **Attention !** Votre taille de tableau n'étant pas forcément un multiple de votre nombre de threads par blocs vous passer la taille du tableau à votre kernel et ne pas effectuer d'opération pour les threads dépassant cette taille

Kernel avec blocs 3D

Nous avons utilisé jusqu'à présent des blocs 1D. Dans la majorité des cas cela est suffisant et plus efficace, il arrive souvent "d'aplatir" son problème pour le rendre uni unidimensionnel.

En réalité le langage CUDA défini un type **"dim3"** dédié à la création de blocs 3D non réguliers. Il est ensuite possible de décomposer un espace 3D en blocs de taille voulue et de se servir des indices 3D sans avoir à "aplatir" sont problème.

```
__global__ void my_kernel(...)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int k = blockIdx.z * blockDim.z + threadIdx.z;
  //...
}

int main()
{
  //...
  dim3 threadsPerBlock(16,8,4);
  dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x,
          (N + threadsPerBlock.y - 1) / threadsPerBlock.y,
          (N + threadsPerBlock.z - 1) / threadsPerBlock.z);

  my_kernel<<< numBlocks, threadsPerBlock >>>(...);
  //...
}
```

Kernel avec blocs 3D

Exercice 5 :

Reprenez l'exercice "loop workshare" fait dans le cours de HPC.

Ecrivez un programme CUDA qui effectue la même opération sur le GPU.

Mesurez le temps d'exécution du kernel et comparez le au temps d'exécution de l'exercice "loop workshare" sur CPU parallélisé.

Combien de coeurs CPU faut il pour rivaliser avec le GPU ?

Mesurez l'impact du choix de la taille des blocs, sachant que les indices à gauche sont les plus rapides dans la définition de "dim3".

Kernel avec blocs 3D

Voir correction sur code source

Profiling

La mesure du temps avec les timers introduits précédemment est vite limitée. Pour des mesures précise de performance sur le GPU Nvidia met à disposition des outils de *profling* avancés.

nvprof : outil a utiliser dans la console linux. Permets d'avoir des informations avancées sur le temps d'execution de chaque kernel, des chargement mémoire, des surcharges de l'API CUDA, ...
Outil non visuel mais utile sur des serveurs distants.

nvvp : (NVidia Visual Profiler) outil écrit en Java, qui permet d'effectuer un profiling complet via une interface graphique. Permets de générer une frise chronologique du code et de voir l'impact de chaque appel indépendamment. Permet d'effectuer des analyses poussées de chaque kernel (occupation, débit mémoire, utilisation calcul, ...)

Utilisez la commande nvprof sur votre programme précédent pour avoir une vue détaillée de ses performances.

```
nprof my_program
```

CUDA BLAS

Les GPU sont **particulièrement adaptés au calcul matriciel**. Pour autant, il est difficile d'écrire un calcul matriciel optimisé tirant parti de toute la complexité de l'architecture GPU.

C'est pour cela que **Nvidia propose une bibliothèque CuBLAS** qui fonctionne sur le même principe que OpenBLAS et qui contient les mêmes opérations.

```
//...
cublasHandle_t cu_handle;
if(cublasCreate(&cu_handle) != CUBLAS_STATUS_SUCCESS)
{
  printf("GPU handle create fail\n");
  exit(EXIT_FAILURE);
}
//...
//prototype of cublasSgemm function
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                           cublasOperation_t transa, cublasOperation_t transb,
                           int m, int n, int k,
                           const float           *alpha,
                           const float           *A, int lda,
                           const float           *B, int ldb,
                           const float           *beta,
                           float           *C, int ldc);
//...
```

CUDA BLAS

Exercice 6 :

En partant du code de calcul OpenBLAS du cours de HPC, construisez un programme qui effectue une opération matricielle identique sur le GPU et sur le CPU.

Comparez le résultat et le temps de calcul CPU et GPU.

Utilisez *nvprof* pour mesurer les performances détaillées de votre programme.

Voir correction sur code source

Interfacer CUDA

Il est possible d'écrire des **codes complets en CUDA** puisqu'il s'agit d'une surcouche du C standard. Pour autant le compilateur CUDA est en général plus lent et est en retard de plusieurs versions sur gcc. Il est donc commun de compiler le code CUDA à part puis de **l'interfacer avec un autre language** (C/C++, Python, ...)

Une interface C/CUDA est très facile à construire. Il suffit de définir les prototypes des fonctions du code CUDA encadrées de la mention "extern C ...", puis fournir au code C les prototypes de ces mêmes fonctions, de compiler les deux indépendemment, puis de faire l'édition des liens avec le compilateur C en ajoutant "-lcudart" pour linker l'API.

Une interface Python/CUDA se construit de la même manière que une interface Python/C. Il faut créer un fichier C faisant appel à la librairie "Python.h" qui permet de construire l'interface. Il faut ensuite compiler ce fichier et faire le lien avec les objets CUDA pré-compilés grace à la bibliothèque python "distutils.core.Extension" **(voir l'exemple sur le git)**