

CentraleSupélec

Mineure CalHau2

CUDA basics

Stéphane Vialle

Stephane.Vialle@centralesupelec.fr
http://www.metz.supelec.fr/~vialle

CentraleSupélec

CUDA basics

- Principe d'exécution d'un pgm CUDA
- Variables et transferts de données CPU/GPU
- Définition de la grille de blocs
- Définition et exécution d'un 1^{er} kernel
- Compilation d'une application CUDA

Principes d'exécution d'un pgm CUDA

Lancement de calculs GPU depuis le CPU

Etapes d'exécution d'une application CUDA :

- Lancement sur le CPU d'un pgm d'apparence classique:
- Démarrage sur le CPU (initialisation de variables, calculs légers)
- Transfert des données depuis la mémoire du CPU vers la mémoire du GPU
- Lancement depuis le CPU de calculs sur le GPU:
 - exécution distante et massivement parallèle de « kernels » GPU.
- Transfert des résultats depuis la mémoire du GPU vers la mémoire du CPU

Les codes des kernels sont transférés aussi depuis le CPU

Rmq : les transferts CPU/GPU sont très couteux il faut minimiser le nombre et le volume des transferts CPU/GPU

Principes d'exécution d'un pgm CUDA

Exec. de grilles de blocs de threads

Le programme CPU demande l'exécution d'un ensemble de threads (des « gpu threads ») :

- threads identiques,
- threads organisés en blocs, chaque bloc s'exécutant sur un seul multiprocesseur,
- blocs organisés au sein d'une grille, qui répartit ses blocs sur tous les multiprocesseurs,
- grille et blocs 1D, 2D ou 3D
 - une grille 2D de blocs 2D
 - une grille 2D de blocs 1D
 - ...

Principes d'exécution d'un pgm CUDA

Exec. de grilles de blocs de threads

Le programme CPU demande l'exécution d'un ensemble de threads (des « gpu threads ») :

- le scheduler de blocs répartit les blocs sur les différents Stream Multiprocesseurs
- différents GPU arriveront sans problème à exécuter la même grille de blocs (chacun à son rythme)

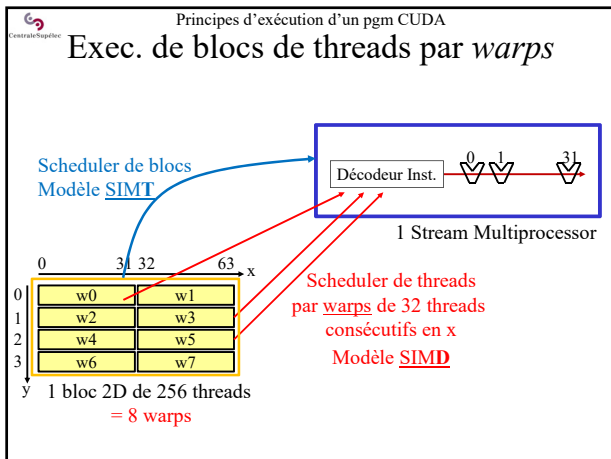
Principes d'exécution d'un pgm CUDA

Exec. de blocs de threads par warps

Scheduler de blocs
Modèle SIMT

1 Stream Multiprocessor

1 bloc 2D de 256 threads



Principes d'exécution d'un pgm CUDA
Contraintes sur la taille des blocs

Créer des blocs contenant un nombre entier de warps

- un décodeur d'instruction pilote 32 threads hardware (≈ 32 ALU)
- le scheduler de threads active des warps de 32 threads si possible consécutifs en x dans le bloc (il privilégie la dimension en x)

→ Créer des blocs d'au moins 32 threads
sinon une partie des ALU seront toujours inutilisées

→ Créer des blocs ayant un nombre de threads multiple de 32
sinon le dernier warp sera incomplet et des ALU seront inutilisées

→ Et ... créer des blocs ayant une dimension en x multiple de 32 sinon la « coalescence » sera moins bonne ou plus compliquée (voir plus loin)...

Mais parfois ça marche encore très bien avec une dimension en x multiple de 16...!

Principes d'exécution d'un pgm CUDA
Liens Kernel GPU – Grille de blocs

L'algorithme d'un kernel GPU et son implantation sont liés aux structures de données accédées sur le GPU et à la grille de blocs de threads prévue pour son déploiement

Ex : Un thread d'un bloc 2D ne doit pas faire les mêmes calculs qu'un thread d'un bloc 1D pour identifier la case de tableau qu'il doit traiter (voir plus loin)

global void myKernel(...)

Algorithme & code de calcul

Data structures

→ Il sera nécessaire de développer de manière cohérente un kernel GPU, son déploiement (sa grille de blocs) et les structures de donnée sur le GPU

CUDA basics

- Principe d'exécution d'un pgm CUDA
- Variables et transferts de données CPU/GPU
- Définition de la grille de blocs
- Définition et exécution d'un 1^{er} kernel
- Compilation d'une application CUDA

Variables et transferts de données CPU/GPU
« Qualifiers » de CUDA

Fonctionnement des « qualifiers » de CUDA :

	<u>__device__</u>	<u>__constant__</u>	<u>__shared__</u>
Variables	Mémoire globale GPU Durée de vie de l'application Accessible par les codes GPU et CPU	Mémoire constante GPU Durée de vie de l'application Ecrit par code CPU, lu par code GPU	Mémoire partagée d'un multiprocesseur Durée de vie du block de threads Accessible par le code GPU, sert à cacher la mémoire globale GPU
Fonctions	<u>__device__</u> Appel sur GPU Exec sur GPU	<u>__host__</u> (default) Appel sur CPU Exec sur CPU	<u>__global__</u> Appel sur CPU Exec sur GPU

→ Les « qualifiers » différencient les parties de code GPU et CPU.

Variables et transferts de données CPU/GPU
Transfert de données CPU-GPU

Variables allouées sur GPU à la compilation (« symboles ») :

```
float TabCPU[N]; // Array on CPU
__device__ float TabGPU[N]; // Array on GPU (symbol)
```

Transfert de données du CPU vers un « symbole » stocké sur GPU :

```
// Copy all TabCPU array into TabGPU array
cudaMemcpyToSymbol(TabGPU, &TabCPU[0],
    sizeof(float)*N, 0,
    cudaMemcpyHostToDevice);

// Copy 2nd half of TabCPU array into 2nd half TabGPU array
cudaMemcpyToSymbol(TabGPU, &TabCPU[N/2],
    sizeof(float)*N/2, sizeof(float)*N/2,
    cudaMemcpyHostToDevice);
```


Transfert de données d'un « symbole » stocké sur GPU vers le CPU :

```
// Copy all TabGPU array into TabCPU array
cudaMemcpyFromSymbol(&TabCPU[0], TabGPU,
    sizeof(float)*N, 0,
    cudaMemcpyDeviceToHost);
```

Variables et transferts de données CPU/GPU

Variabiles statiques ou dynamiques ?

Variabiles **statiques sur GPU**

- Entièrement définies à la compilation
- Connues et directement accessibles depuis le code GPU
→ inutile de passer leurs adresses en paramètres des kernels
- MAIS : tout le code GPU qui utilise ces variables doit être dans le même fichier que leurs déclarations!** 
(on arrive rapidement à un seul gros fichier qui contient tout le code GPU!)
- En fait, ces variables/symboles peuvent être partagées entre fichiers (déclarées extern dans des fichiers.h), mai seulement en **activant le mode de compilation séparé**

```

Compilation :
nvcc --relocatable-device-code=true ...
Ou bien : nvcc -rdc=true ...

Edition de liens :
nvcc --device-link ...
Ou bien : nvcc -dlink ...

Voir le document « NVIDIA CUDA Compiler Driver NVCC »

```

Variables et transferts de données CPU/GPU

Transfert de données CPU-GPU

Variabiles allouées sur GPU à l'exécution :

```

float *TabCPU; // Dynamic array on CPU
float *TabGPU; // Dynamic array on GPU
cudaError_t cudaStat; // Result of op on dynamic CUDA vars.

// Allocation of the dynamic arrays from the CPU
TabCPU = (float *) malloc(N*sizeof(float));
cudaStat = cudaMalloc((void **) &TabGPU, N*sizeof(float));

```

Copie de variables dynamiques (CPU → GPU) :

```

// Copy TabCPU dynamic array into TabGPU dynamic array
cudaStat = cudaMemcpy(TabGPU, TabCPU, sizeof(float)*N,
                      cudaMemcpyHostToDevice);

```

Copie de variables dynamiques (GPU → CPU) :

```

// Copy TabGPU dynamic array into TabCPU dynamic array
cudaStat = cudaMemcpy(TabCPU, TabGPU, sizeof(float)*N,
                      cudaMemcpyDeviceToHost);

```

Libération des allocations dynamiques

```


free(TabCPU);
cudaStat = cudaFree(TabGPU);

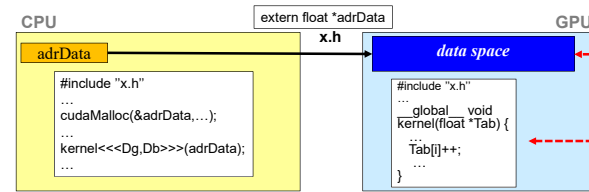
```

Variables et transferts de données CPU/GPU

Variabiles statiques ou dynamiques ?

Variabiles **dynamiques sur GPU**

- La plupart sont allouées et libérées par le CPU:
 - leurs pointeurs sont stockées sur le CPU (le CPU possède la cartographie de la mémoire GPU!)
 - leurs pointeurs doivent être passés en paramètres lors des appels aux kernels GPU
- Ces variables peuvent être partagées entre plusieurs fichiers (variables déclarées « extern » dans les fichiers .h) 



Variables et transferts de données CPU/GPU

CUDA basics

- Principe d'exécution d'un pgm CUDA
- Variabiles et transferts de données CPU/GPU
- Définition de la grille de blocs**
- Définition et exécution d'un 1^{er} kernel
- Compilation d'une application CUDA

Définition de la grille de blocs

Limites sur les tailles de grille et de blocs

Type CUDA de dimension de bloc ou de grille :

- dim3** est un type structuré de 3 int (_x, _y et _z)
- qui permet de définir des **descripteurs de grilles et de blocs (Dg et Db)**

Limites (fortes) sur la taille des blocs :

- Barres, matrices, ou cubes de threads.
- Db.x ≤ 1024, Db.y ≤ 1024, Db.z ≤ 64
- Nbr total de threads / bloc ≤ 1024

```

// GPU thread management
dim3 Dg, Db;
// Block of 128x4 threads
Db.x = 128;
Db.y = 4;
Db.z = 1;

```

Limites (faibles) sur la taille de la grille :

- Barres, matrices, ou cubes de blocs.
- Dg.x ≤ 2³¹ - 1
- Dg.y ≤ 65535, Dg.z ≤ 65535

```

// Grid of 512 blocks
Dg.x = 512;
Dg.y = 1;
Dg.z = 1;
// → 262144 threads!

```

Lancement d'un kernel à partir de sa grille de blocs :

```

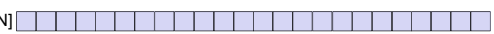
// Classical call from a CPU routine
Kernel<<< Dg, Db >>>(parameter);

```

Définition de la grille de blocs

Grille de blocs sans débordement

Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :

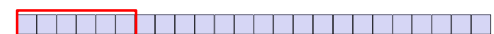
Tab[N] 

Pour chaque case « i » :
Tab[i] = Tab[i] * 2.0

Stratégie (simpliste) :

- 1 thread GPU traitera 1 case
- Blocs 1D de threads

Définition des blocs (1D) :



```

Db.x = BLOCK_SIZE_X; // = 32/64/128/256/512/1024
Db.y = BLOCK_SIZE_Y; // = 1
Db.z = BLOCK_SIZE_Z; // = 1

```

Définition de la grille de blocs

Grille de blocs sans débordement

Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :

Définition de la grille (1D) si $(N \% Db.x) = 0$:

On « pave » les données avec des blocs.

Exemple :

```
Dg.x = N/Db.x = N/BLOCK_SIZE_X
Dg.y = 1
Dg.z = 1
```

Et si $(N \% Db.x) \neq 0$?? ...

Définition de la grille de blocs

Grille de blocs avec débordement

Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :

Définition de la grille (1D) si $(N \% Db.x) \neq 0$:

On « pave » les données avec des blocs entiers, même si cela déborde !

Exemple :

```
if (N%BLOCK_SIZE_X == 0)
    Dg.x = N/BLOCK_SIZE_X;
else
    Dg.x = N/BLOCK_SIZE_X + 1;
Dg.y = 1;
Dg.z = 1;
```

Rmq : on va créer « trop de threads » : il faudra en tenir compte dans le code (voir plus loin)...

Définition de la grille de blocs

Grille de blocs avec débordement

Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :

Définition de la grille (1D) si $(N \% Db.x) \neq 0$:

On « pave » les données avec des blocs entiers, même si cela déborde !

Exemple :

```
Dg.x = N/BLOCK_SIZE_X + (N%BLOCK_SIZE_X ? 1 : 0);
Dg.y = 1;
Dg.z = 1;
```

Rmq : on va créer « trop de threads » : il faudra en tenir compte dans le code (voir plus loin)...

Définition de la grille de blocs

Grille de blocs avec débordement

Exemple (très simple) d'une « grille 1D de blocs 1D de threads » :

Définition de la grille (1D) si $(N \% Db.x) \neq 0$:

On « pave » les données avec des blocs entiers, même si cela déborde !

Exemple :

```
Dg.x = (N-1)/BLOCK_SIZE_X + 1;
Dg.y = 1;
Dg.z = 1;
```

Rmq : on va créer « trop de threads » : il faudra en tenir compte dans le code (voir plus loin)...

Définition de la grille de blocs

Créer beaucoup de petits threads

Masquage des temps d'accès mémoires des GPU :

- un GPU passe d'un warp de threads à un autre très rapidement
- un GPU masque la latence de ses accès mémoires par multi-threading

→ Ne pas hésiter à créer un grand nombre de petits threads GPU pour réaliser un calcul

Ex.: pour traiter une table de N éléments :

- des Threads traitant UN élément chacun
- une Grille de blocs de N threads au total

vs

- des Threads traitant n éléments chacun
- une Grille de blocs de N/n threads au total

Définition de la grille de blocs

Granularité de la grille et des blocs

Combien de threads/bloc et de blocs/grille ?

Le scheduler de threads d'un blocs souhaite avoir « des blocs assez gros » (« beaucoup de threads dans un bloc »)

→ Pour avoir des warps de threads à activer en réserve, afin de recouvrir des temps d'accès à la mémoire

Le scheduler de blocs souhaite avoir « plein de blocs pas trop gros »

→ Des blocs pas trop gros pour en charger plusieurs en « résident » dans chaque SM, afin de recouvrir des temps d'accès à la mémoire

→ Beaucoup de blocs pour pouvoir remplir tous les SM du GPU

Le GPU ne démarre un bloc de threads sur un SM que s'il a assez de registres et autres rsrc disponibles

Avoir des petits blocs permet donc d'en charger plus en « résidents » dans chaque multiprocesseur

Vaut-il mieux faire peu de gros blocs ou beaucoup de petits blocs ?

Définition de la grille de blocs

Granularité de la grille et des blocs

Combien de threads/bloc et de blocs/grille ?

Plusieurs stratégies possibles :

- Faire des blocs de taille moyenne (128/256 threads) pour permettre aux deux schedulers d'optimiser l'exécution
- Calculer la taille des blocs menant à l'occupation maximale des ressources du GPU...
- Expérimenter diverses tailles de blocs de 32/64/128/256/512/1024 !

La solution optimale dépend du code CUDA et du modèle de GPU

- Trouver la granularité optimale de grille de blocs peut demander de nombreuses expérimentations
- Optimisation un peu moins sensible avec les nouveaux GPU
→ Voir TP

Définition de la grille de blocs

Granularité de la grille et des blocs

Source : <https://en.wikipedia.org/wiki/CUDA>

Technical specifications	Compute capability (version)															
	1.0	1.1	1.2	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0 (7.2?)	7.5
Maximum number of resident grids per device concurrent kernel execution	1 to 4			16	4			32			16	128	32	16		128
Maximum dimensionality of grid of thread blocks	2										3					
Maximum x-dimension of a grid of thread blocks	65535										2 ³¹ - 1					
Maximum y- or z-dimension of a grid of thread blocks											3					
Maximum dimensionality of thread block																
Maximum x- or y-dimension of a block	512										1024					
Maximum z-dimension of a block											64					
Maximum number of threads per block	512										1024					
Warp size											32					
Maximum number of resident blocks per multiprocessor	8				16						32				16	
Maximum number of resident warps per multiprocessor	24	32	48								64				32	
Maximum number of resident threads per multiprocessor	768	1024	1536								2048				1024	
Number of 32-bit registers per multiprocessor	8 K	16 K	32 K	64 K	128 K						64 K					
Maximum number of 32-bit registers per thread block	N/A	32 K	64 K	32 K	64 K	64 K	32 K	64 K	32 K	64 K	32 K	64 K	32 K	64 K		
Maximum number of 32-bit registers per thread	124		63								255					

Certaines caractéristiques sont très stables, d'autres moins ...

CUDA basics

- Principe d'exécution d'un pgm CUDA
- Variables et transferts de données CPU/GPU
- Définition de la grille de blocs
- Définition et exécution d'un 1^{er} kernel
- Compilation d'une application CUDA

Définition et exécution d'un 1^{er} kernel

« Qualifiers » de CUDA

Fonctionnement des « qualifiers » de CUDA :

	<u>_device_</u>	<u>_constant_</u>	<u>_shared_</u>
Variables	Mémoire globale GPU Durée de vie de l'application Accessible par les codes GPU et CPU	Mémoire constante GPU Durée de vie de l'application Ecrit par code CPU, lu par code GPU	Mémoire partagée d'un multiprocesseur Durée de vie de la block de threads Accessible par le code GPU, sert à cacher la mémoire globale GPU
Fonctions	<u>_device_</u> Appel sur GPU Exec sur GPU	<u>_host_ (default)</u> Appel sur CPU Exec sur CPU	<u>_global_</u> Appel sur CPU Exec sur GPU

→ Les « qualifiers » différencient les parties de code GPU et CPU.

Définition et exécution d'un 1^{er} kernel

1^{er} Kernel (traitant 1 donnée par thread)

Kernel utilisant la mémoire globale et des registres
Une barre de threads par bloc, et une barre de blocs par grille (un choix).
Un thread traite une seule donnée.
Hyp : Nd = k * BLOCK_SIZE_X

```

Db = {BLOCK_SIZE_X, 1, 1}
Dg = {Nd / BLOCK_SIZE_X, 1, 1}

```

```

_global_ void k1(void)
{
  int idx; // Registers:
  float data; // 16 Kreg per
  float res; // multipro.

  // Compute data idx of the thread
  idx = threadIdx.x +
        blockIdx.x * BLOCK_SIZE_X;
  // Read data from the global mem
  data = InGPU[idx];
  // Compute result
  res = (data + 1.0f) * data ...;
  // Write result in the global mem
  OutGPU[idx] = res;
}

```

Définition et exécution d'un 1^{er} kernel

1^{er} Kernel (traitant 1 donnée par thread)

Calcul de l'indice de la donnée traitée par chaque thread

```

// Compute data idx of the thread
idx = blockIdx.x * BLOCK_SIZE_X
    + threadIdx.x;
// Read data from the global mem
data = InGPU[idx];

```

2 variables implicites et propres à chaque thread :
dim3 threadIdx
dim3 blockIdx

Grille de blocs de threads: Bloc 0, Bloc 1, Bloc 2

Tableau de données: 0 1 ... N-1

idx = blockIdx.x * BLOCK_SIZE_X + threadIdx.x
Indexage permettant des accès coalescents

CentralesSupélec

Définition et exécution d'un 1^{er} kernel

1^{er} Kernel (traitant 1 donnée par thread)

Calcul de l'indice de la donnée traitée par chaque thread

```
// Compute data idx of the thread
idx = (N-1) - (threadIdx.x +
             blockDim.x*BLOCK_SIZE_X);
// Read data from the global mem
data = InGPU[idx];
```

2 variables implicites et propres à chaque thread :
 dim3 threadIdx
 dim3 blockDim

Grille de blocs de threads

Tableau de données

$idx = (N-1) - (blockIdx.x * BLOCK_SIZE_X + threadIdx.x)$

Mais ce serait un indexage moins coalescent... !!

CentralesSupélec

Définition et exécution d'un 1^{er} kernel

1^{er} Kernel (traitant 1 donnée par thread)

Calcul de l'indice de la donnée traitée par chaque thread

```
// Compute data idx of the thread
idx = threadIdx.x +
      blockDim.x*BLOCK_SIZE_X;
if (idx % 2 == 1)
    idx = (N-1) - idx;
// Read data from the global mem
data = InGPU[idx];
```

2 variables implicites et propres à chaque thread :
 dim3 threadIdx
 dim3 blockDim

Grille de blocs de threads

Tableau de données

$idx = (N-1) - (blockIdx.x * BLOCK_SIZE_X + threadIdx.x)$

Mais ce serait un indexage NON coalescent... !!

CentralesSupélec

Définition et exécution d'un 1^{er} kernel

1^{er} Kernel (traitant 1 donnée par thread)

Kernel utilisant la mémoire globale et des registres
 Une barre de threads par bloc, et une barre de blocs par grille (un choix).
 Un thread traite une seule donnée.

Hyp : $N_d \# k \cdot BLOCK_SIZE_X$

```
global__ void k1(void)
{
    int idx; // Registers:
    float data; // 16Kreg per
    float res; // multipro.
    // Compute data idx of the thread
    idx = threadIdx.x +
          blockDim.x*BLOCK_SIZE_X;
    // If the elt indexed exists:
    if (idx < Nd) {
        // Read data from the global mem
        data = InGPU[idx];
        // Compute result
        res = (data + 1.0f)*data ...;
        // Write result in the global mem
        OutGPU[idx] = res;
    }
}
```

Db = {BLOCK_SIZE_X, 1, 1}
 Dg = {Nd/BLOCK_SIZE_X, 1, 1}
 Dg = {Nd/BLOCK_SIZE_X + 1, 1, 1}

Pavage classique

Protection classique :
 Les threads « en trop » ne font rien...

CentralesSupélec

Définition et exécution d'un 1^{er} kernel

Exécution de la grille de blocs

Exécution depuis le CPU d'une grille de threads sur le GPU :

```
// Usually only 2 arguments are specified:
Kernel<<< Dg, Db >>>(parameter, ..., ... ..);
```

→ Descripteur d'un bloc 3D de threads
 → Descripteur d'une grille 3D de blocs 3D de threads

```
// Complete syntax
Kernel<<< Dg, Db, Ns, S >>>(parameter, ..., ... ..);
```

→ Numéro du stream d'interaction CPU/GPU (le 0 par défaut)... utile pour du recouvrement transferts/calculs en utilisant plusieurs streams
 → Allocation dynamique de shared memory au lancement de chaque bloc sur un SM... (0 octets par défaut)

CentralesSupélec

CUDA basics

- Principe d'exécution d'un pgm CUDA
- Variables et transferts de données CPU/GPU
- Définition de la grille de blocs
- Définition et exécution d'un 1^{er} kernel
- **Compilation d'une application CUDA**

CentralesSupélec

Compilation d'une application CUDA

Compilation 100% CUDA

Compilation d'applications CUDA – entièrement développées en CUDA :

Définitions de variables et fonctions avec « qualificateurs » CUDA

Code C, ou C++ avec des appels à la lib CUDA

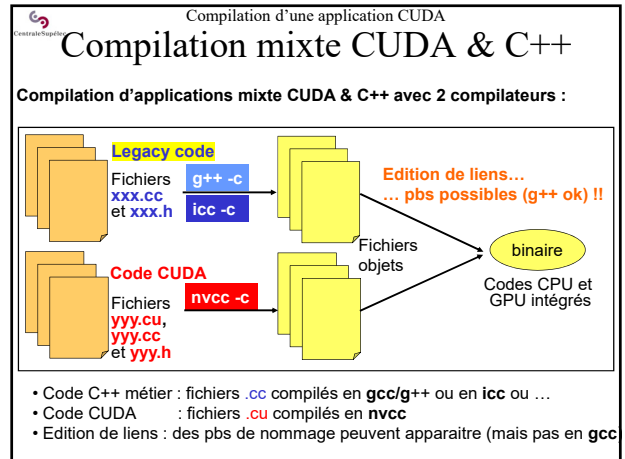
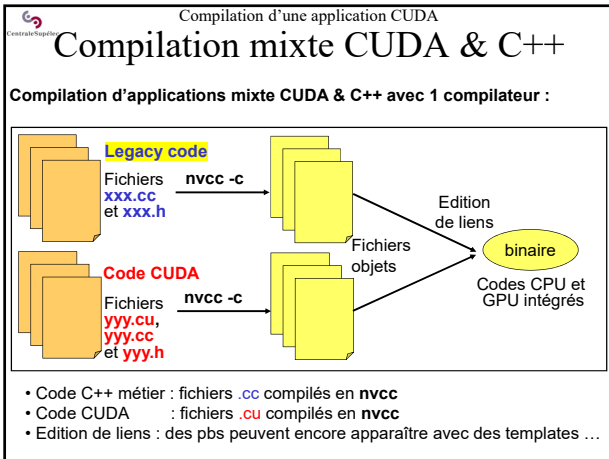
Code C, ou C++ « standard »

Fichiers xxx.cu et xxx.h, et xxx.cc incluant si besoin cuda.h et cuda_runtime.h

nvcc → binaire (Codes CPU et GPU intégrés)

Pour les codes C/C++ simples :

- Il est possible de simplement tout recompiler en nvcc dans des fichiers xxx.cu (et xxx.cc incluant cuda.h et cuda_runtime.h)
- Mais les optimisations sérielles peuvent en souffrir...



CentraleSupélec

CUDA basics

End