Notes de programmation CUDA

Michaël Mathieu

12 août 2023

Résumé

Ces modestes notes de programmation tentent de proposer une synthèse rapide de ce qu'il m'a fallu afin de pouvoir écrire mes premiers code CUDA.
Le guide de programmation NVIDIA reste bien évidemment la référence.

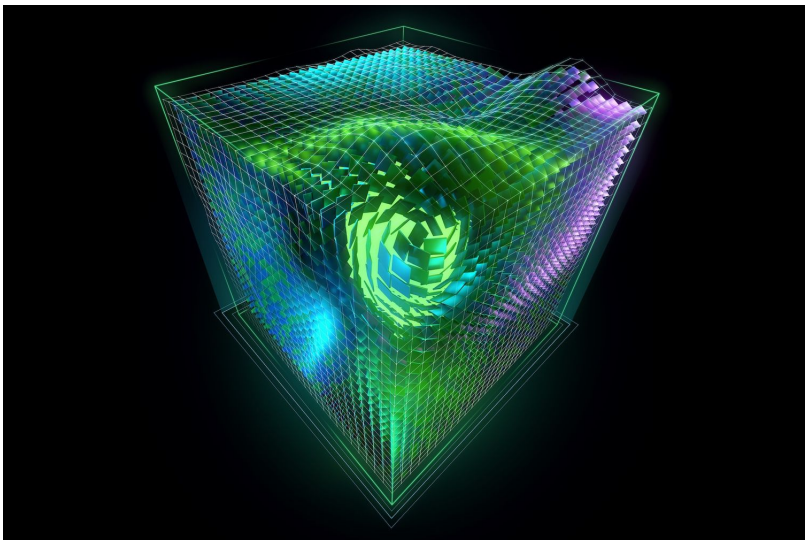


Table des matières

1 Guide de référence

Il existe plusieurs livres, mais force est de constater que certaines parties deviennent obsolètes car les versions de CUDA évoluent et des fonctionnalités sont supprimées, alors que d'autres sont ajoutées.
Par exemple, lorsque l'on fait du parallélisme dynamique, il n'est plus possible dans un kernel d'attendre la fin de l'exécution des kernels enfants avec `cudaDeviceSynchronize()` alors que c'était autorisé jusqu'à la version 11.x.
Sur les forums, NVIDIA confirme que cela n'est pas sans impact pour les codes déjà existants :

"Yes, this will require refactoring your code.
There is no zero-impact workaround."

Ainsi, la **seule référence** que le programmeur doit avoir est le guide de programmation NVIDIA [1] qui est très complet et bien écrit (avant qu'il ne soit proposé gratuitement sur leur site, il était en vente chez l'éditeur Wiley [2]).
Dans ces notes de programmation, nous ferons au maximum référence à ce guide afin que le lecteur puisse obtenir davantage de précisions ou vérifier une explication.
Nous avons utilisé la version 12.2.0 [3].

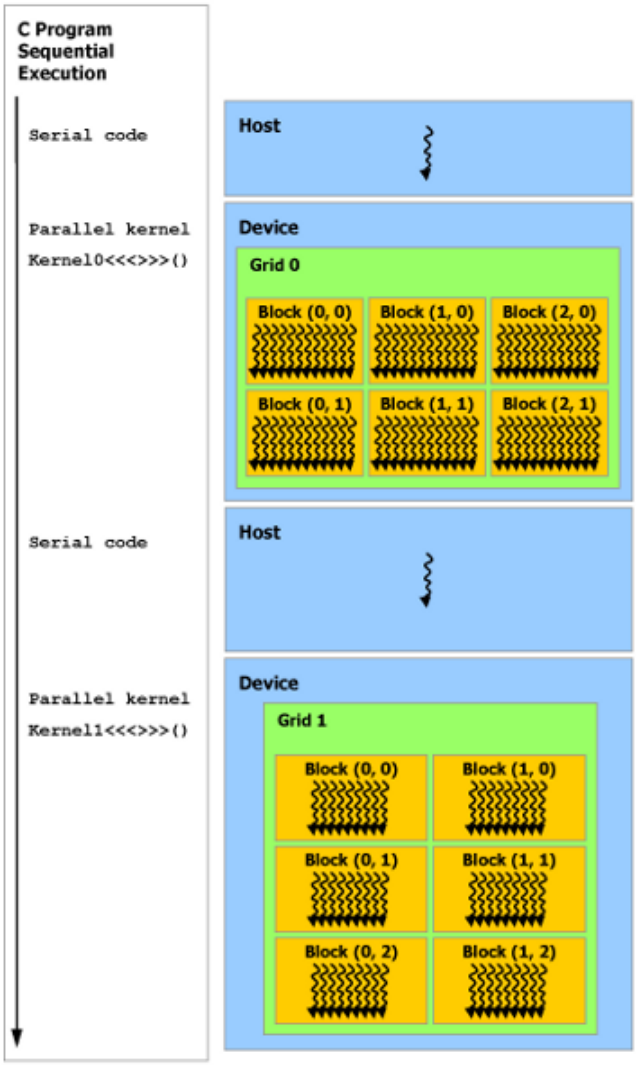
1. https://docs.nvidia.com/cuda/cuda-c-programming-guide/
2. https://www.wiley.com/en-fr/Professional+CUDA+C+Programming-p-9781118739327
3. https://developer.nvidia.com/cuda-toolkit-archive

2 Architecture

Lorsque l'on aborde la programmation CUDA, plusieurs termes sont utilisés et même pour les programmeurs expérimentés, la définition de ces termes n'est pas forcément limpide.

host	machine sur laquelle la carte graphique (GPU) est installée
host code	code qui est exécuté sur le CPU
device	carte graphique
kernel	code qui est exécuté sur le GPU
device code	synonyme de kernel
thread	processus qui va exécuter un kernel
block	ensemble de threads qui exécutent le même kernel
grid	ensemble de blocks dont les threads exécutent le même kernel
warp	collection de 32 threads qui exécutent la même ligne de code avec les données propres à chaque thread, voir le point 5.6 Warp divergence

Cf. guide 9.1.2



Cf. guide 2.4

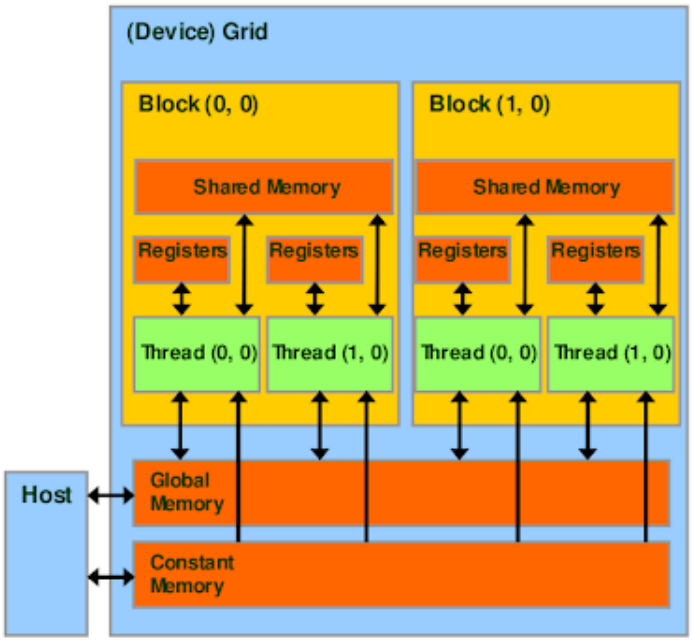
Le host code (serial code) s'exécute pendant qu'en parallèle s'éxécutent les kernels (Kernel0 et Kernel1). Pour attendre qu'un kernel se termine avant d'en relancer un autre, il faut dans le host code mettre l'instruction `cudaDeviceSynchronize()` avant l'appel du nouveau kernel. Voir le point 5.5 Synchronisation.

3 Mémoires

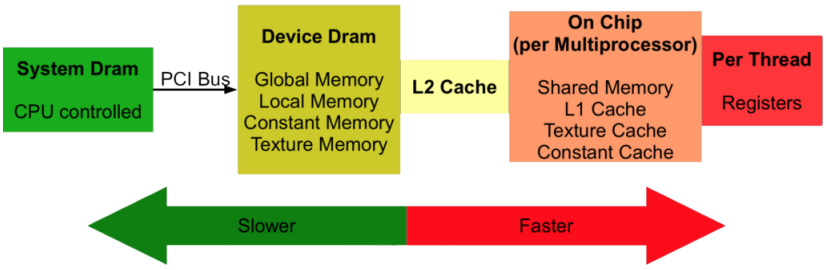
Les kernels n'utilisent pas la mémoire du host, mais la mémoire du device. Il existe plusieurs types de mémoires et c'est au programmeur de choisir laquelle sera la plus judicieuse.

Mémoire	Accès	Visibilité	Durée de vie	Vitesse	Note
global	RW	tous les threads + host	permanent	lent	beaucoup d'espace
constant	R	tous les threads + host	permanent	rapide	limité à 64KB
shared	RW	par block	block	rapide	limité, utilisé notamment pour la communication entre les threads
local	RW	par thread	thread	lent	limité à 512KB
register	RW	par thread	thread	rapide	limité, voir 5.2 Limites du GPU

R = Read (lecture), RW = Read + Write (lecture + écriture)



Cf. `https://doc.sling.si/en/workshops/programming-gpu-cuda/CUDA/cuda/`



Cf. `http://thebeardsage.com/cuda-memory-hierarchy/`

3.1 Global

L'espace de stockage le plus grand et qui persiste après la fin de l'exécution d'un kernel, mais est aussi la mémoire la plus lente. Quand on dit qu'une carte graphique a –par exemple– 2GB de mémoire, cette taille indique celle de la mémoire globale.

Pour utiliser la mémoire globale, au niveau du host code, il faut utiliser les fonctions `cudaMalloc`, `cudaMemcpy`/`cudaMemcpyToSymbol`/`cudaMemcpyFromSymbol`, `cudaMemset`, `cudaFree` cf. guide 6.2.2. Depuis un kernel, pour utiliser la mémoire globale, nous parlons de *"Dynamic Global Memory Allocation and Operations"* cf. guide 10.34.

3.2 Constant

Cette mémoire rapide, mais limitée à 64KB n'est accessible qu'en lecture pour les kernels. Seul le host code peut écrire avant l'appel à des kernels.

Pour utiliser la mémoire constante, au niveau du host code, il faut utiliser la fonction `cudaMemcpyToSymbol`.

3.3 Shared

Cette mémoire n'est pas accessible depuis le host code. Elle est partagée par tous les threads du même block.

3.4 Local

La mémoire locale et la mémoire globale ont les mêmes performances (lent).
Il existe cependant 3 différences :

1. indépendant des autres threads
2. durée de vie du thread
3. limité à 512KB

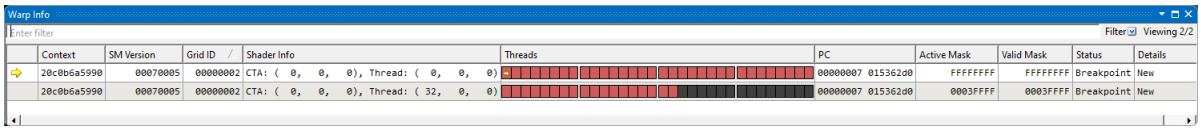
3.5 Register

La mémoire la plus rapide, mais très limitée.
⚠si nous dépassons les capacités de cette mémoire, les variables seront stockées dans la mémoire locale ce qui peut avoir de terribles conséquences sur la performance.

Les tableaux de **taille fixe** qui ne dépassent pas la taille disponible sont stockés dans les registres.

4 Environnement de développement – Visual Studio

Sous Windows, Visual Studio avec le plugin Nsight offre de belles possibilités pour le debugging, notamment la fenêtre *warp info* qui permet de visualiser les threads utilisés et de se déplacer dans chacun d'eux afin de voir l'état des variables.



Nsight Warp Info ; lors du debugging : Extensions → Nsight → Windows → Warp Info

4.1 Installation de Nsight

1. télécharger et installer le CUDA Toolkit [4]
2. Copier tous les fichiers
   `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.2\extras\visual_studio_integration\MSBuildExtensions`
   ⇒ `C:\Program Files\Microsoft Visual Studio\2022\Community\MSBuild\Microsoft\VC\v170\BuildCustomizations`

4.2 Créer un projet

1. créer un projet de type C++ console
2. renommer le fichier de base en `.cu` au lieu de `.cpp`
3. sur la solution → bouton droit → Build Dependencies → Build Customizations..
   ⇒ cocher *CUDA 12.2*
4. dans les *Properties* (`Alt+Enter`) du fichier `.cu` :
   — General → Item Type → CUDA C/C++ + cliquer sur *Apply*
   — CUDA C/C++ → Device → Verbose PTXAS Output → Yes

4.3 Exécuter un projet

1. démarrer Nisight Monitor (après chaque redémarrage de la machine uniquement)
2. Build → Build Solution (à chaque fois que votre code change)
3. Extensions → Nsight → Start CUDA Debugging (Next-Gen)

Il existe de multiples exemples de projets Visual Studio sur `https://github.com/NVIDIA/cuda-samples`.

4. `https://developer.nvidia.com/cuda-downloads`

5 Programmation

5.1 Hello World

Pour ce premier exemple, nous allons créer un kernel qui ne fait qu'afficher
`Hello World from thread#N`
où N correspond à l'identifiant du thread. Cet identifiant est calculé en utilisant :
— le n° du block
— le nombre de threads par block
— le n° du thread dans le block

Etant donné que CUDA a été conçu pour effectuer des calculs graphiques, nous pouvons avoir des threads réparti en 1 (x), 2 (x, y) ou 3 (x, y, z) dimensions. Dans cet exemple, nous travaillons uniquement avec une dimension. Pour travailler avec 2 ou 3 dimensions, au lieu de mettre un paramètre en `int` il faut utiliser le type `dim3` cf. guide 5.2.
Ces paramètres sont indiqués par l'utilisateur lors de l'appel du kernel (`<<<nbBlocks, nbThreadsPerBlock>>>`).

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

__global__ void gpuHelloWorld() {
    int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
    printf("Hello World from thread#%d\n", threadId);
}

int main()
{
    int nbBlocks = 2;
    int nbThreadsPerBlock = 4;
    gpuHelloWorld<<<nbBlocks, nbThreadsPerBlock>>>();

    cudaDeviceSynchronize();

    return 0;
}
```

5.2 Limites du GPU

Chaque GPU a des limites matérielles et il est intéressant d'écrire du code qui permet de s'adapter à ces dernières ou d'informer l'utilisateur si le GPU n'offre pas assez de ressources.

L'API CUDA permet de récupérer par GPU un object de type `cudaDeviceProp`[5] qui donne toutes les spécificités matérielles.
⚠certaines limites sont universelles et ne sont donc pas disponible via l'API, par exemple la taille maximale de la mémoire locale par thread qui a toujours été limitée à 512KB cf. guide 16.2.
Dans l'exemple ci–dessous, nous pouvons déterminer le nombre maximum de registers supportés par thread (si on considère le pire des cas où le block utiliserait le maximum de threads possible).

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

void printGpuCapabilities() {
  cudaDeviceProp prop;
  int nbGPU;
  cudaGetDeviceCount(&nbGPU);
  if (nbGPU == 0) {
    printf("host not CUDA capable");
  }
  for (int i = 0; i < nbGPU; i++) {
    cudaGetDeviceProperties(&prop, i);
    printf("Device#%d %s (capability %d.%d)\n", i, prop.name, prop.major, prop.minor);
    printf("Total global memory %d bytes\n", prop.totalGlobalMem);
    printf("Total constant memory %d bytes\n", prop.totalConstMem);
    printf("%d multiprocessors\n", prop.multiProcessorCount);
    printf("Max %d blocks per multiprocessor\n", prop.maxBlocksPerMultiProcessor);
    printf("Max %d threads per block\n", prop.maxThreadsPerBlock);
    printf("%d threads per warp\n", prop.warpSize);
    printf("Shared memory per block %d bytes\n", prop.sharedMemPerBlock);
    printf("%d registers of 32-bit per block\n", prop.regsPerBlock);
    printf("%d registers of 32-bit per thread\n", prop.regsPerBlock / prop.maxThreadsPerBlock);
  }
}

int main() {
  printGpuCapabilities();

  return 0;
}
```

5. https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html

5.3 Mémoires

Afin de clarifier le propos, nous proposons pour chaque situation un exemple complet avec le strict minimum.

5.3.1 Global

Pour qu'une donnée puisse être lu/écrite par le kernel, puis récupérée par le host code, nous devons utiliser des pointeurs et ensuite copier les données du host au device et inversément.

Dans les deux exemples qui suivent, vous noterez les changements suivants :
— `__device__ int gpuValues[NB_VALUES]` vs. `cudaMalloc`
— `cudaMemcpyToSymbol` vs. `cudaMemcpy(..., cudaMemcpyHostToDevice)`
— si statique, le tableau n'est pas envoyé en paramètre
— `cudaMemcpyFromSymbol` vs. `cudaMemcpy(..., cudaMemcpyDeviceToHost)`
— si statique, pas besoin de libérer de la mémoire les tableaux (`free` et `cudaFree`).

⚠vous noterez le garde–fou (`if (threadId < nbValues)`) au cas où le kernel devait être appelé avec plus de blocks/threads que prévu. C'est une bonne pratique.

Taille des données statique

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

#define NB_VALUES 10

__device__ int gpuValues[NB_VALUES];

__global__ void modifyGlobalMemory() {
    int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
    if (threadId < NB_VALUES) {
        gpuValues[threadId] += 5;
    }
}

int main()
{
    int values[NB_VALUES];
    for (int i = 0; i < NB_VALUES; i++) {
    values[i] = i;
        printf("before values[%d] => %d\n", i, values[i]);
    }

    cudaMemcpyToSymbol(gpuValues, values, NB_VALUES * sizeof(int));
    modifyGlobalMemory<<<1, 2>>>();

    cudaDeviceSynchronize();

    cudaMemcpyFromSymbol(values, gpuValues, NB_VALUES * sizeof(int));
    for (int i = 0; i < NB_VALUES; i++) {
        printf("after values[%d] => %d\n", i, values[i]);
    }

    return 0;
}
```

Taille des données dynamique

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

__global__ void modifyGlobalMemory(int* values, int nbValues) {
    int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
    if (threadId < nbValues) {
        values[threadId] += 5;
    }
}

int main()
{
    int nbValues = 10;
    int* values = (int*) malloc(nbValues * sizeof(int));
    for (int i = 0; i < nbValues; i++) {
        values[i] = i;
        printf("before values[%d] => %d\n", i, values[i]);
    }

    int* gpuValues;
    cudaMalloc(&gpuValues, nbValues * sizeof(int));
    cudaMemcpy(gpuValues, values, nbValues * sizeof(int), cudaMemcpyHostToDevice);
    modifyGlobalMemory<<<1, 2>>>(gpuValues, nbValues);

    cudaDeviceSynchronize();

    cudaMemcpy(values, gpuValues, nbValues * sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < nbValues; i++) {
        printf("after values[%d] => %d\n", i, values[i]);
    }

    cudaFree(gpuValues);
    free(values);

    return 0;
}
```

5.3.2 Constant

Fonctionne comme de la mémoire globale de taille statique. Comme qualificateur, au lieu de `__device__`, on utilisera `__constant__`. Bien entendu, la fonction `cudaMemcpyFromSymbol` n'est pas pertiente avec de la mémoire constante, car elle n'est peut être modifiée par les kernels.

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

#define NB_VALUES 10

__constant__ int CONST_VALUES[NB_VALUES];

__global__ void useConstantMemory() {
    int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
    if (threadId < NB_VALUES) {
        printf("CONST_VALUES[%d] => %d\n", threadId, CONST_VALUES[threadId]);
    }
}

int main()
{
    int values[NB_VALUES];
    for (int i = 0; i < NB_VALUES; i++) {
        values[i] = i;
     }

    cudaMemcpyToSymbol(CONST_VALUES, values, NB_VALUES * sizeof(int));
    useConstantMemory<<<1, 5>>>();

    cudaDeviceSynchronize();

    return 0;
}
```

5.3.3 Shared

La mémoire partagée est souvent utilisée pour la communication entre les threads. Dans ce cas, on souhaite réaliser un traitement lorsque tous les threads auront réalisés leur opération. Pour cela, nous utilisons `__syncthreads` qui garantit, qu'au moment où le code continuera, que tous les autres threads auront terminé d'exécuter le code qui précède.
Dans l'exemple ci–dessous, nous souhaitons afficher une seule fois le contenu du tableau. Nous avons arbitrairement décidé que cela sera effectué par le thread 0.

Taille des données statique Le kernel peut avoir plusieurs variables partagées.

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

#define NB_VALUES 10

__global__ void useStaticSharedMemory() {
    __shared__ int values[NB_VALUES];

    int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
    if (threadId < NB_VALUES) {
        values[threadId] = threadId;
    }

    __syncthreads();

    if (threadId == 0) {
        for (int i = 0; i < NB_VALUES; i++) {
            printf("values[%d] => %d\n", i, values[i]);
        }
    }
}

int main()
{
    useStaticSharedMemory<<<1, 2>>>();

    cudaDeviceSynchronize();

    return 0;
}
```

Taille des données dynamique La taille en bytes est spécifiée lors de l'appel du kernel, comme $3^{ème}$ paramètre (`<<<nbBlocks, nbThreadsPerBlock, sharedMemorySizeInBytes>>>`). ⚠dans ce cas, il ne peut y avoir qu'une seule variable partagée et elle doit être préfixée du mot–clé `extern`. Si plusieurs variables sont nécessaires, il faut les grouper dans cet unique tableau et le découper en tranches [6].

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

__global__ void useDynamicSharedMemory(int nbValues) {
    extern __shared__ int values[];

    int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
    if (threadId < nbValues) {
        values[threadId] = threadId;
    }

    __syncthreads();

    if (threadId == 0) {
        for (int i = 0; i < nbValues; i++) {
            printf("values[%d] => %d\n", i, values[i]);
        }
    }
}

int main()
{
    int nbValues = 10;
    useDynamicSharedMemory<<<1, 2, nbValues * sizeof(int)>>>(nbValues);

    cudaDeviceSynchronize();

    return 0;
}
```

6. Voir https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

5.3.4 Local/Register

Dans l'exemple ci–dessous, seul le thread 0 va afficher le contenu du tableau. Etant donné que chaque thread a un tableau différent, seul le tableau du thread 0 sera affiché (`[5, 0, 0, 0, 0, 0, 0, 0, 0, 0]`).

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

#define NB_VALUES 10

__global__ void useRegisterLocalMemory() {
    int values[NB_VALUES];

    int threadId = threadIdx.x + (blockIdx.x * blockDim.x);
    if (threadId < NB_VALUES) {
        values[threadId] = 5;
    }

    __syncthreads();

    if (threadId == 0) {
        for (int i = 0; i < NB_VALUES; i++) {
            printf("values[%d] => %d\n", i, values[i]);
        }
    }
}

int main()
{
    useRegisterLocalMemory<<<1, 2>>>();

    cudaDeviceSynchronize();

    return 0;
}
```

5.4 Dynamic Parallelism

Le parallélisme dynamique permet à partir d'un thread parent de créer des threads enfants cf. guide 12.

Pour que cela soit possible, vous devez modifier les *Properties* de votre fichier `.cu` :
CUDA C/C++ → Common → Generate Relocatable Device Code → Yes

5.5 Synchronisation

Il existe 2 manières de synchroniser du code CUDA :

1. `cudaDeviceSynchronize` permet dans le host code d'attendre que *tous* les threads sur la GPU soient terminés avant de continuer
   ⚠avant la version 12 de CUDA, il était possible d'utiliser aussi cette fonction pour qu'un thread parent puisse attendre la fin de l'exécution des threads enfants qu'il aurait créé. Désormais, il faut réaliser cette attente au niveau du host code cf. guide 12.2.1.3.
2. `__syncthreads` permet de synchroniser des threads du même block

5.6 Warp divergence

Dans un warp, chaque thread exécute la *même* instruction avec les données propres à son contexte. Cela est très efficace, mais les performances peuvent se dégrader si le traitement a beaucoup de contrôle de flux : `if`, `while`, etc.
Imaginons qu'en nous basant sur le `threadId`, nous devions effectuer un traitement différent que ce chiffre soit pair ou impair. Lorsque le code pour les chiffres paires s'exécutera, l'autre moitié des threads attendra ; idem quand le code pour les chiffres impaires s'exécutera.

Ainsi, certains algorithmes peuvent poser problèmes et il nous faudra être ingénieux afin d'optimiser ce genre de situation, par exemple en modifiant l'algorithme pour qu'un block s'occupe des chiffres paires et un autre des chiffres impaires.
C'est tout l'enjeux du développement CUDA : trouver des manières astucieuses de modéliser le problème de base afin qu'il soit efficacement traité en parallèle. Sans quoi, il sera plus lent que s'il était traité de manière séquentielle par le CPU.