CUDA, acronyme de Compute Unified Device Architecture, est une technologie GPGPU développée par NVIDIA. Elle permet de décupler les performances de calcul du système en exploitant la puissance des processeurs graphiques (GPU) et donc en effectuant des calculs en programmation parallèle à l’aide du langage C/C++.

CUDA nous permet de développer des algorithmes de hautes performances accélérés par des milliers de threads parallèles. Pour ce faire, aucun prérequis n’est nécessaire en GPU ou programmation parallèle, seulement en C++.

CUDA a plusieurs particularités, telles une bonne hiérarchisation des zones de mémoires permettant d’organiser simplement les transferts entre CPU-GPU, ainsi que le regroupement en grille de blocks eux-mêmes composés de threads, le tout en 1, 2 ou 3 dimensions pour favoriser tous types de calculs.

On retrouve plusieurs étapes dans le déroulement du programme:

Déclaration et Exécution

Dans la terminologie de CUDA, le CPU est désigné par *host* et le GPU par *device,* et une fonction quelconque s’exécutant sur le GPU est appelée *kernel*. Pour stipuler qu’un kernel doit s’exécuter sur le GPU on utilise le préfixe *\_\_global\_\_* dans la déclaration/ définition. Lors de l’appelle de la fonction kernel, il nous faut préciser entre triples chevrons l’environnement d’exécution avant de spécifier les paramètres : *nom\_Fonction<<<nb\_Blocks,nb\_Threads>>>(param, param)*. Les variables entre chevrons sont donc de types tridimensionnelles, représentant le nombres de blocks/threads en x, y ou z.

Par ailleurs, en CUDA de la même façon que lors de l’exécution de boucles classiques, il est indispensable de connaître l’indice afin connaître de l’itération courante. CUDA introduit donc ici de nouvelles variables semblables à l’indice d’une boucle : *threadIdx, blockIdx,* et *blockDim* (nombre de threads par blocks). Ces variables connaissent automatiquement leurs indices, soit leur x, y et z.

Allocation Mémoire et Transfert en Mémoire (copie)

Pour allouer de la mémoire sur GPU, en vue de la copie d’une variable pour le calcul provenant du CPU, puis la retourner au CPU pour obtenir un résultat, il existe deux méthodes.

La première est d’allouer la mémoire sur GPU avec la fonction *cudaMalloc* puis de copier les données sur le device ou l’host avec la fonction *cudaMemcpy* en spécifiant en paramètres le sens de la copie (HostToDevice, DeviceToHost etc) ainsi que la taille de l’objet à copier.

La deuxième méthode est d’utiliser la fonction *cudaMalloManaged*, la mémoire unifiée, qui permet d’éliminer le besoin d’un mouvement de donnée ainsi que celui de l’allocation mémoire sur GPU, et correspond à une sorte de mémoire partagée CPU/GPU. Il est également possible de déclarer une variable en tant que *\_\_device\_\_ \_\_managed\_\_* qui sera naturellement visible par le CPU/GPU comme une variable globale.

Ensuite, la synchronisation des threads est nécessaire avant de retourner le résultat en fin de calcul pour s’assurer que tous les calculs sont terminés et cohérents. Pour ceci on utilise la fonction *cudaDeviceSynchronize*.

Il est ensuite important en fin de programme de libérer la mémoire précédemment allouée sur GPU, avec la fonction *cudaFree*.