AMSI03 Projet 3
Fusion entre Cuda et MPI

Qingqing HU

1 Architecture de la version MPI + Cuda

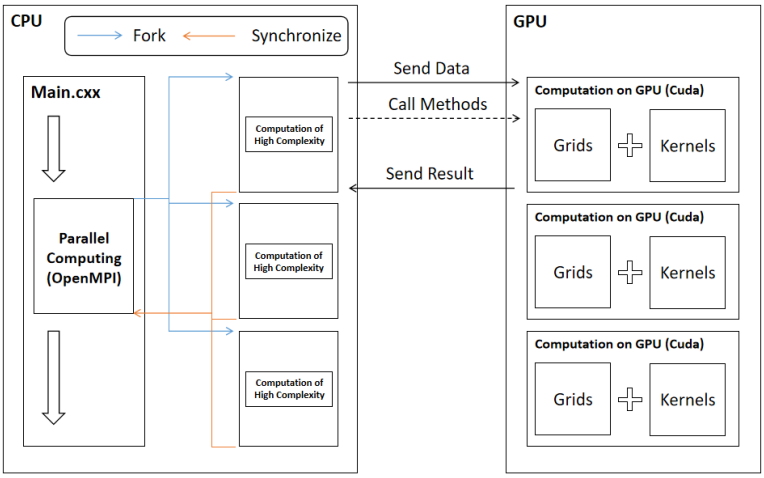

FIGURE 1 – MPI + Cuda

Le but du TP est d'étudier la fusion entre les deux versions Cuda et MPI, pour obtenir une version hybride destinée à être utilisée sur une machine parallèle dont chaque nœud contient une carte graphique. Le GPU est particulièrement efficace lors du traitement des opérations matricielles. Pour 10 itérations, l'exécution de la version MPI dure environs $\frac{55}{n}$ avec n le nombre des noyaux utilisés en parallele. Au contraire, l'exécution de la version Cuda dure seulement 5.7 secondes, qui est beaucoup plus rapide que la version sans GPU.

En général, on a la fonction main.cxx qui s'execute dans le CPU principal. Pour accélérer, on crée la région parallèle avec OpenMPI. Une partie du programme est copiée et parallelement executée sur des noyaux différents ou même sur des autres CPUs. Selon la fonction du programme principal, les noyaux communiquent entre eux, et enfin synchronisent leurs résultats et les renvoient au programme principal. D'après l'énoncé du TP, chaque nœud contient une carte graphique. Donc, on prédéfinit des fonctions de GPU et on exécute les calculs les plus complexes sur les GPUs. Figure 1 est le diagramme schématique d'une architecture simplifiée de la version MPI + Cuda.

2 Modification à partir du code MPI

Fichier Modifié	Fonction Modifiée	donnée ajouté sur CPU	donnée ajouté sur GPU
scheme.cxx scheme.hxx	Scheme : :iteration_domaine()	std : :vector<double> h_partialSums	double * *diffPartial* double * *diff*
values.cxx value.hxx	Values : :init() Values : :boundaries()	double * *h_ u*	double * *d_ u*

Table 1 – Fichiers et fonctions modifiés

Fichier ajouté	Explication
Dossier Cuda	Allocation et récupération de mémoire liées au GPU Fonctions associées au transfert des donneés Noyaux du calcul à GPU
variation.hxx dim.hxx iteration.hxx	Nouveaux header fichiers

Table 2 – Fichiers ajoutés au code MPI

Table 1 et Table 2 résument les modifications du code MPI pour faire les calculs internes dans chaque sous-domaine MPI par une carte graphique.

Le calcul interne le plus complexe est l'évolution de chaque itération. On considère donc à l'implémenter sur GPU. D'abord, on copie les fonctions *.cu* prédéfinies par la version Cuda dans le dossier et adapte les fichier *.hxx* aux nouvelles fonctions. Dans ce cas-là, l'évolution de chaque itération est implémentée à GPU par la fonction *iterKernel* dans le fichier *iteration.cu*. Le calcul de la variation entre les itérations est également géré de la même manière.

Pour éviter plusieurs fois de copie de *m_ u* entre CPU et GPU, on gère les données de façon plus intelligente. En terme des structures de données, *h_ u* et *d_ u* remplacent *m_ u* dans la classe *Value*. Originallement, *m_ u* est pointé vers une liste des valeurs de *u(t)* à CPU. Dans la nouvelle version, *h_ u* est pointé vers une adresse dans la mémoire de CPU. Au contraire, *d_ u* est pointé vers une adresse dans la mémoire de GPU. *h_ u* et *d_ u* représentent les mêmes données, mais sauvegardé respectivement à CPU et GPU. Comme *d_ u* est initialisé sur GPU en utilisant la fonction modifiée dans la classe *Values*, *h_ u* et *d_ u* ne sont synchronisées qu'au moment où on a besoin d'accéder aux données *h_ u*. À noter que la synchronisation peut être aussi automatique. La fréquence de la synchronisation est contrôlée par un attribut *freq*. Il est par défaut -1, qui signifie que les programmes ne se synchronisent pas pendant les itérations.

Dans le même but, on ajoute les attributs *diff* et *diffPartial* dans la classe *Scheme* pour transférer le calcul de variation à GPU. *diff* représente la liste de la variation de chaque élément de *u(t)* et *diffPartial* est la variation par block. Après le calcul de la variation à GPU, on synchronise *diffPartial* et *h_ partialSums.data()*. À noter que le dernier est sauvegardé à CPU. Enfin on somme chaque composant de *h_ partialSums* à CPU pour avoir la variation totale de chaque processus.

Voici le tableau (3) pour classifier l'exécution sur CPU ou GPU.

CPU Principal	**sous-domaines MPI**	**GPU**
Lancer main.cxx	\	\
Créer les sous-domaines	\	\
\	Initialiser *Prm, C, u_0* et adapter les conditions initiales	Allouer des mémoires et initialiser pour *d_u*, *diff* et *diffPartial*
\	Synchroniser les sous-domaines	\
\	Itération : Afficher *u_t* (Optionnel)	Copier *d_u* de GPU à CPU (*h_u*)
\	Itération : Appliquer la formule des itérations	Mettre à jour *d_u* en utilisant *iterKernel*
\	Itération : Calculer la variation	Mettre à jour *diffPartial* et faire la copie de *diffPartial* de GPU à CPU
\	Répeter les iterations et afficher les détails du temps utilisé	\
\	Libérer sous-domaines	\
Fin du main.cxx	\	\

TABLE 3 – Où les calculs et les copies de données sont faits

Fichier modifié	Explication
build.py run.py	Ajout de la compilation des fichiers Cuda
CMakeLists.txt	Choix d'un bon compilateur

TABLE 4 – Fichiers modifiés pour la compilation

À part le calcul, il faut aussi modifier les fichiers suivants pour la compilation : build.py, run.py et CMakeLists.txt. Une difficulté rencontrée ici est que le compilateur de Cuda (nvcc) ne s'adapte pas à OpenMPI.