Description du travail réalisé: Parallélisation avec Cuda AMS I-03

Jincheng KE

16 février 2024

1 Environnement de Développement

Le développement de ce projet a été réalisé sur une plateforme équipée d'un processeur AMD R9 3900 doté de 12 cœurs et 24 threads, et d'une carte graphique NVIDIA RTX 2070. L'ensemble fonctionnait sous un système d'exploitation Windows 11. Les tests de performance et de fonctionnalité ont été effectués principalement sous le sous-système Windows pour Linux (WSL2) avec Ubuntu, en raison de difficultés rencontrées lors de l'exécution directe sous Windows.

2 Objectifs du Projet

Le but principal de ce projet était d'étendre une solution existante de résolution de l'équation de Poisson, initialement conçue pour un traitement séquentiel sur CPU, afin d'exploiter le potentiel des calculs parallèles réalisés avec CUDA sur GPU.

3 Version à Grain Fin

Dans cette version, nous avons d'abord examiné le fichier `dim.cu`, qui permet à la GPU de connaître le nombre de points de discrétisation de l'espace d_n, le minimum de l'intervalle spatial h_xmin, l'incrément spatial h_dx, ainsi que d_lambda. Ces préparatifs facilitent grandement l'implémentation des kernels par la suite.

Dans le fichier `memmove_cuda.cu`, nous avons inclus des fonctions pour la gestion de la mémoire sur l'ordinateur et la carte graphique, telles que l'allocation d'espace dans la mémoire et la mémoire vidéo, ainsi que le transfert et la copie de données. La classe `values` joue un rôle crucial en fournissant les méthodes `dataCPU` et `dataGPU` pour accéder aux données dans la mémoire et la mémoire vidéo, respectivement. De plus, la méthode `synchronized` permet de synchroniser les données entre la mémoire et la mémoire vidéo.

Pour faciliter davantage l'implémentation, le fichier `user.cu` définit pour la GPU les fonctions représentant les conditions initiales (`cond_ini`), les conditions aux limites (`cond_lim`) et les forces extérieures (`force`) de l'équation aux dérivées partielles.

L'implémentation principale réside dans le fichier `iteration.cu`, où nous avons d'abord adopté l'approche la plus conventionnelle. Pour chaque indice tridimensionnel $J = (i, j, k)$, où $imin \leq i \leq imax$, $jmin \leq j \leq jmax$, et $kmin \leq k \leq kmax$, un thread GPU est associé à un indice tridimensionnel $J = (i, j, k)$ pour effectuer le travail correspondant, puis mettre à jour les données. Ceci est réalisé par la fonction `__global__ void iteration_kernel`.

Pour l'affectation de `gridSize` et `blockSize`, étant donné que nos indices sont tridimensionnels, nous avons défini ces tailles en utilisant le type de données `dim3`. Cela nous permet d'établir la correspondance suivante entre les indices et les threads GPU :

$$
\begin{aligned}
i &= blockIdx.x \times blockDim.x + threadIdx.x + imin, \\
j &= blockIdx.y \times blockDim.y + threadIdx.y + jmin, \\
k &= blockIdx.z \times blockDim.z + threadIdx.z + kmin.
\end{aligned}
$$

Cette correspondance assure que chaque thread GPU travaille sur un indice unique dans l'intervalle spécifié. Pour exclure les threads GPU qui dépassent cet intervalle, une condition `if` est utilisée. Pour les

threads dans l'intervalle, leur travail est similaire à celui effectué dans la version monofil, à l'exception de la sommation des variations absolues des états actuel et précédent, qui pourrait conduire à des conflits en parallèle. Contrairement à l'approche de réduction utilisée avec OpenMP pour le calcul parallèle sur CPU, dans la version CUDA, nous évitons d'abord la sommation et effectuons d'abord tous les calculs, puis calculons et sommons les différences d'état ultérieurement. Cette stratégie, implémentée dans `variation.cu`, optimise l'efficacité de la GPU en minimisant les communications coûteuses.

Lors de l'utilisation pratique de ce code, il est crucial de bien ajuster les valeurs de `gridSize` et `blockSize` pour garantir une correspondance entre les threads GPU et les indices, tout en évitant de dépasser excessivement le nombre total d'indices. Un `gridSizeblockSize` supérieur au nombre d'indices est nécessaire pour couvrir tous les indices, mais un excès trop important réduirait l'efficacité de la GPU. En effet, les threads excédentaires, au-delà du nombre d'indices, ne participeraient pas au calcul, entraînant une baisse de performance. De plus, le nombre d'indices dépasse généralement le nombre de cœurs physiques du GPU (par exemple, pour une RTX 2070, qui possède 2304 cœurs physiques), ce qui peut mener à une concurrence inutile pour les ressources du GPU entre les threads actifs et inactifs, diminuant ainsi l'efficacité du GPU. Il est donc essentiel, tout en assurant que le `gridSizeblockSize` couvre tous les indices, de minimiser cette valeur autant que possible. La fonction `get_gridSize` a été conçue pour calculer automatiquement la taille de grille optimale étant donné un `blockSize` et le nombre d'indices, afin d'optimiser l'utilisation du GPU tout en réduisant au minimum le nombre de threads inactifs.

4 Résultats Expérimentaux pour la Version à Grain Fin

Nous avons réalisé des expériences en fixant `blockSize` à la valeur $(8, 8, 8)$ et avons testé les cas où $n = 401$ et $n = 501$. Pour chaque configuration, nous avons observé que les résultats des calculs CUDA étaient cohérents avec ceux obtenus en utilisant une seule thread CPU (en comparant la variance à chaque étape), ce qui confirme la précision de notre code CUDA.

Dans le cas où $n = 401$, le temps de calcul sur une seule thread CPU était de 24.82 secondes, tandis que le temps de calcul avec la version CUDA était de 1.256 secondes, atteignant ainsi un *speedup* de 19.76, ce qui montre une amélioration significative de la performance. Pour le cas où $n = 501$, le temps de calcul sur une seule thread CPU était de 52.18 secondes, et le temps de calcul avec la version CUDA était de 2.505 secondes, résultant en un *speedup* de 20.83, ce qui constitue également une amélioration de performance notable.

5 Version à Grain Quasi-Grossière

Malgré une nette amélioration des performances par rapport au CPU, nous avons remarqué que dans cette version conventionnelle, un grand nombre de threads (déterminé par `gridSize*blockSize`) était nécessaire, et chaque thread effectuait des calculs relativement simples. Cela pourrait présenter un goulot d'étranglement similaire à celui de la granularité fine dans le calcul parallèle sur CPU, suggérant qu'une légère augmentation de la granularité pourrait améliorer les performances. Ainsi, nous avons développé une seconde version du code CUDA, désignée comme la version à granularité quasi-grossière, pour la distinguer de la première version à granularité fine. Dans cette nouvelle version, chaque thread GPU exécute une triple boucle sur `i, j, k`, avec la longueur de chaque boucle définie par `sx, sy` et `sz`, tout en veillant à ne pas dépasser les valeurs maximales de `i, j, k`. Nous désignons le triplet (`sx, sy, sz`) comme `xyz_size`. Pour un `blockSize` et un `xyz_size` donnés, la fonction améliorée `get_gridSize` permet de déterminer le `gridSize` correspondant.

Cependant, la valeur de `xyz_size` ne reflète pas directement la quantité de ressources GPU allouées (`gridSize*blockSize`). Pour une répartition plus intuitive et efficace des ressources GPU, notre fonction `get_xyz_size_opt` agit comme une sorte de "fonction inverse", déterminant le `xyz_size` optimal étant donné un `blockSize`, le nombre d'indices et une taille cible pour `gridSize`. Cette fonction n'est pas une véritable fonction inverse car tous les `gridSize` cibles ne possèdent pas de solution entière. Lorsqu'une solution entière est possible, la fonction fournit le `xyz_size` optimal ; sinon, elle retourne le `xyz_size` optimal pour le plus petit `gridSize` supérieur à la cible ayant une solution entière. Nous pouvons résumer cette relation par l'inégalité suivante :

$$\text{get\_gridSize}(\text{blockSize}, \text{ijk\_size}, \text{get\_xyz\_size\_opt}(\text{blockSize}, \text{ijk\_size}, \text{gridSize})) \geq \text{gridSize},$$

où `ijk_size` représente la taille de l'index. Lorsqu'une solution entière pour `gridSize` est disponible, nous pouvons atteindre l'égalité.

La répartition des ressources GPU peut ainsi être optimisée. Nous avons implémenté l'algorithme à granularité quasi-grossière dans
`__global__ void iteration_kernel_coarse`.
Selon notre correspondance établie, nous obtenons :

$$i = sx \times (blockIdx.x \times blockDim.x + threadIdx.x) + imin;$$
$$j = sy \times (blockIdx.y \times blockDim.y + threadIdx.y) + jmin;$$
$$k = sz \times (blockIdx.z \times blockDim.z + threadIdx.z) + kmin;$$

Ces valeurs servent de point de départ pour les indices de la boucle for (`ii, jj, kk`), avec des valeurs maximales (exclusives) définies comme suit :

$$ii\_max = \min(i + sx, imax + 1);$$
$$jj\_max = \min(j + sy, jmax + 1);$$
$$kk\_max = \min(k + sz, kmax + 1);$$

L'ordre des boucles for n'affecte pas la précision des résultats mais influence la performance des calculs. Dans notre cas, `ii`, étant l'indice le plus densément emballé dans la mémoire vidéo, se trouve dans la boucle la plus interne pour augmenter les chances de toucher le cache. `kk`, à l'inverse, étant l'indice le plus dispersé, se trouve dans la boucle la plus externe. Ainsi, nos boucles for suivent l'ordre `kk`, puis `jj`, et enfin `ii`.

Pour la répartition des ressources GPU, nous envisageons d'allouer le plus grand nombre de parallélismes au niveau le plus externe, `kk`, puis à `jj`, et enfin à `ii`. Il est donc prévisible que `kk` se voie attribuer davantage de ressources (`gridSize*blockSize`), tandis que `ii` en recevra moins. La question demeure : comment choisir spécifiquement chaque dimension de `blockSize` et `gridSize` ? Nous remarquons que pour chaque paire (`gridSize, blockSize`), leur efficacité théorique maximale est limitée par le ratio `eff = (ijk_size / (blockSize * xyz_size)) / ceil(ijk_size / (blockSize * xyz_size))`, où le numérateur représente la performance théorique nécessaire et le dénominateur, les ressources réellement allouées. Nous pouvons donc expérimenter avec des paires (`gridSize, blockSize`) ayant un `eff` élevé. À cette fin, nous avons conçu la fonction `test_block_size`, qui affiche toutes les paires (`gridSize, blockSize`) existantes et leur efficacité théorique `eff` correspondante.

En intégrant l'expérience acquise lors de la programmation parallèle mixte OpenMP et MPI, nous assimilons le nombre de `gridSize` au nombre de processus MPI et le `blockSize` au nombre de threads OpenMP. Selon notre expérience, attribuer un plus grand nombre de `gridSize` à `kk` peut accélérer les calculs, tandis qu'un `blockSize` adéquat pour `ii` peut équilibrer efficacement la charge et optimiser l'utilisation du GPU.

6 Résultats Expérimentaux pour la Version à Grain Quasi-Grossière

Après avoir sélectionné les combinaisons avec un facteur d'efficacité théorique élevé et basé sur nos hypothèses, nous avons procédé à des expériences. Pour $n = 401$, nous avons observé de meilleures performances lorsque `blockSize` était de $(57, 7, 1)$ et `gridSize` de $(1, 19, 399)$. Dans ce cas, le temps de calcul pour la version quasi-granulaire était de 1.124 secondes, offrant un *speedup* de 22.08 par rapport au CPU, ce qui représente une amélioration par rapport au *speedup* de 19.76 de la version à granularité fine. Pour $n = 501$, la meilleure performance a été obtenue avec un `blockSize` de $(126, 1, 1)$ et un `gridSize` de $(1, 499, 499)$, où le temps de calcul était de 2.288 secondes. Cela a donné un *speedup* de 22.81 par rapport au CPU, dépassant également le *speedup* de 20.83 de la version à granularité fine.

7 Conclusion

En résumé, nos expériences démontrent que l'utilisation de GPU peut considérablement améliorer les performances de calcul. Cependant, il est important de minimiser la communication fréquente avec le GPU et de réduire autant que possible le nombre de threads GPU inactifs pour maximiser l'efficacité du GPU. Nous pouvons également envisager d'utiliser une version à Grain relativement plus grossière pour améliorer davantage les performances.