Accélération GPU sur ParaDiGMA

Branche dev_paroctree_GPU

Valable pour le commit du 03/09/2020 par Adrien Lafontan

Compilation

- Se mettre sur spiro-dmpe-visu ou spiro-commun-visu (plus de gens sur les GPU)
- Modules : cuda/10.1 cmake/3.12.3 gcc/8.3 openmpi/4.0.1-gcc-8.3 python/2.7.13
- Ajouter dans le script cmake launch : NVCC=nvcc -DPDM_ENABLE_CUDA=ON

Exécution

- Mpirun -np 1 pdm_t_knn_cube -gpu -t <nombre de pts cibles>
- CUDA profile : mpirun -np 1 nvprof pdm_t_knn_cube -gpu
- CUDA memcheck : mpirun -np 1 cuda-memcheck pdm_t_knn_cube -gpu
  - et toutes ses variantes : initcheck, racecheck, synccheck (cf doc cuda)
- CUDA gdb : mpirun -np 1 cuda-gdb pdm_t_knn_cube
  - puis run -gpu
  - on peut ajouter le memcheck : set cuda memcheck on
- débug sur plusieurs procs MPI : mpirun -np 2 xterm -e cuda-gdb pdm_t_knn_cube
  - puis faire run -gpu sur les deux terminaux
- IMPORTANT : si à l'exécution il apparaît des « unknown error », il faut compiler sur spiro-commun-visu, puis exécuter sur spiro-dmpe-visu (aucune idée de l'origine de l'erreur, mais cette solution marche)

Options de compilation

- Localisation : /paradigma/paradigma/paradigm/src/CmakeLists.txt
- trouver le if(PDM_ENABLE_CUDA)
- ajout d'une librairie (obligatoirement statique) pour permettre le link de toutes les fonctions

CUDA

- CUDA_SEPARABLE_COMPILATION ON : active la compilation séparée (cf rapport)
- target_compile_options : options de compilation pour nvcc
  - -maxrregcount=32 : limite le nombre de registres à utiliser, ne fonctionne pas sans cette option
  - --fmad=false : désactive une optimisation de calcul du mode release (fused multiply-add). En mode débug, on peut l'enlever
  - -G : infos de débug sur GPU (fonctions device)
  - -g : infos de débug sur CPU
  - -gencode=arch=compute_61 code=sm61 : génération de code compatible pour la compute capability 6.1 (cf hardware GPU). On peut ajouter d'autres lignes pour rendre le code compatible pour de plus nombreux hardwares (mais il faut au minimum 3.5)

PDM_t_knn_cube.c

- Ajout de l'option « -gpu », qui initialise la variable globale GPU_ACC à 1
- Ligne 289 : check pour voir si la librairie MPI est CUDA-aware
- Ligne 381 : si l'option gpu est utilisée, on appelle la fonction compute qui utilise le gpu. On transfère explicitement la variable statique « _closest_pts » car elle ne semble pas être partagée par les compilateurs gcc et nvcc

PDM_closest_points_old.cu

- Ancienne version de l'implémentation qui utilise la mémoire unifiée, ne fonctionne plus

PDM_closest_points.cu

- Ajout de headers .cuh
- PDM_closest_points compute_GPU : fonction qui va être appelée à la place de PDM_closest_points_compute
  - Rien ne change, à part la fonction PDM_para_octree_closest_point_GPU (voir plus loin)
  - On transfère manuellement les données de l'octree (comme précédemment), car le get_from_id n'arrive pas à récupérer la variable statique, car on a changé de compilateur (gcc->nvcc)

PDM_closest_points.cuh

- Header pour PDM_closest_points.cu

PDM_cuda_error.cuh

- Header qui contient une macro (gpuErrchk) qui permet de récupérer et d'afficher les erreurs des appels de l'API CUDA
- Cette macro sera systématiquement appelée pour chaque fonction de l'API CUDA (cudaMalloc, cudaMemcpy…)

PDM_cuda.cu

- Contient une fonction permettant d'assigner les valeurs du nombre de threads et de blocs en x, y et z (variables dim3)
  - implémentation nécessaire car faire : var = {x, y, z} passe avec gcc, mais ne fonctionne pas avec intel
- En commentaire, ancienne version du cudaRealloc. A ne pas utiliser car crée des erreurs à l'exécution (unknown error 999)

PDM_cuda.cuh

- Header de PDM_cuda.cu
- Reduce_kernel est une macro permettant de faire une réduction par blocs de threads avec n'importe quel type de vaiable.
  - WarpReduce fait partie de la fonction principale
  - La réduction est une somme sur tous les éléments d'une matrice
  - plus d'informations : https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
- cudaRealloc est une macro permettant la réallocation de données sur GPU
  - Fait une allocation, copie puis libère la mémoire de l'ancien pointeur
  - ATTENTION : ne fonctionne pas partout. Pour libérer le pointeur, il faut y avoir accès

sur GPU. Pour des structures de données complexes (doubles pointeurs, structures avec tableaux) l'allocation en CUDA utilise plusieurs pointeurs, et certains pointeurs peuvent être « inaccessibles » depuis GPU

PDM_morton.cu

- Contient des fonctions traduites en CUDA pour exécution sur GPU
- Fonctions _encode_coords : kernels pour l'exécution en parallèle. N'apporte pas d'accélération significative et n'est plus utilisé
- PDM_morton_encode_coords_CPU : implémentation de l'encodage de Morton en parallèle, en utilisant les kernels _encode_coords. N'apporte pas d'accélération significative et n'est plus utilisé
- PDM_morton_encode_coords_GPU : implémentation de PDM_morton_encode_coords pour l'exécution sur GPU. Est nécessaire pour le fonctionnement du code accéléré
- PDM_morton_binary_search_GPU : implémentation de PDM_morton_binary_search pour exécution sur GPU. Est nécessaire pour le fonctionnement du code accéléré
- PDM_morton_a_gt_b_GPU : implémentation de PDM_morton_a_gt_b pour exécution sur GPU. Est nécessaire pour le fonctionnement du code accéléré

PDM_morton.cuh

- Header de PDM_morton.cu

PDM_para_octree.cu

- Fichier où la majeure partie de l'accélération est codée
- _force_target_points, _closest_src_init, _binary_search_kernel, _get_row : kernels non-utilisés, n'apportent pas d'accélération significative
- _min_heap_reset, _get_octant_part_id, _min_heap_swap, _min_heap_heapify, _min_heap_pop, _octant_min_dist2, _octant_min_dist2_normalized, _min_heap_push, _maximal_intersecting_range, _min_heap_create, _min_heap_free
  - Fonctions adaptées pour l'exécution sur CPU et GPU, avec notamment mot clés __host__ et __device__, quasi-inchangées mis à part l'utilisation de fonctions spécifiques comme cudaRealloc
- _target_points_compute : kernel de calcul principal
  - Ligne 585 : i_tgt est l'indice du point cible, défini par l'index du thread qui lit le kernel.

points_shift est une variable dans le cas où il faudrait appeler le kernel plusieurs fois, c'est un décalage de l'indice des points cibles. Par défaut il vaut 0, et pour l'instant ce n'est pas utilisé, car le rappel de kernel n'est pas encore nécessaire pour – de 10 millions de points.

- Lignes 595-596 : variables pour la section critique (voir plus loin)
- Lignes 608-631 : Allocation des piles de feuilles, avec cudaMalloc. Précédemment, on utilisait des mallocs (voir commentaires), mais c'est peu efficace car ralentit l'exécution, et pose des problèmes de mémoire
- Lignes 634-654 : Allocation de tableaux nécessaires à la recherche. Même remarque que précédemment. On peut envisager de réduire leur taille, en utilisant des booléens pour is_visited par exemple
- Ligne 778 : un des seuls mallocs du kernel, assez petit pour que ça ne pose pas de problème
- Ligne 790 : Fonction provenant de PDM_morton.cu
- Ligne 1001 : utilisation d'une boucle for allant de 1 à 6 plutot qu'un enum, posait problème à NVCC qui compile en C++
- Lignes 1033-1095 : Section critique – seulement dans le cas de 2 procs MPI ou plus
  - Section de code qui subit de l'écriture concurrentielle. On est obligé de faire une exécution séquentielle ici
  - Le verrou (lock) permet de définir si un thread peut continuer son exécution : si lock=1 non, si lock=0 oui, et on assigne la valeur 1 à lock. Tout cela est fait par l'instruction *atomicExch(lock, 1) == 0* (voir fonctions atomiques dans rapport)
  - Les autres threads restent bloqués dans la boucle while tant qu'ils ne peuvent pas s'exécuter. Un thread qui a fini la section critique assigne à sa variable locale leaveLoop la valeur true, pour sortir de la boucle while.
  - Les opérations atomic assurent une écriture ou une incrémentation qui soient faite par un seul thread. En commentaire, il y a les instructions originelles.
  - Lignes 1057-1075 : section pour la réallocation si le tableau est trop petit. Le cudaRealloc ne fonctionne pas, on essaie d'éviter cette section en sur-allouant dès le départ. En commentaire sont des tentatives de solutions à ce problème.
- Lignes 1136-1144 : même remarque, le cudaRealloc ne fonctionne pas
- Lignes 1160-1171 : libération de la mémoire allouée précédemment

- _closest_points_local : fonction qui va appeler le kernel _target_points_compute
  - Lignes 1390 et 1392 : sur-allocation pour pallier le problème du cudaRealloc mentionné plus haut

- Lignes 1408-1422 : Répartit les calculs sur 2 GPU dans le cas de 2 processus MPI (ne fonctionne pas encore). Fait aussi un reset des GPU
- Lignes 1424-1470 : création de pointeurs qui vont être alloués sur le GPU (préfixe « d_ » device pour les différencier des pointeurs sur CPU, parfois on utilise aussi le préfixe « h_ » host pour des pointeurs sur CPU)
- Lignes 1472-1491 : Allocation sur GPU
  - Les arguments sont l'adresse du pointeur, puis la taille d'allocation en octets.
  - Dans le cas de tableaux à l'intérieur de structures (octree, octants), on va créer et allouer des pointeurs indépendants (ex : d_octree_points pour d_octree->points)
- Lignes 1494-1524 : Copie des données sur GPU
  - Les arguments sont le pointeur alloué, le pointeur CPU contenant les données à copier, la taille de copie en octets, et enfin le mot clé indiquant si on copie depuis le CPU vers le GPU ou inversement.
  - Dans le cas des tableaux dans une structure :
    - On copie d'abord la structure (les variables qui ne sont pas des tableaux seront copiées)
    - Puis on copie l'adresse du pointeur indépendant (mentionné plus haut) dans l'adresse du pointeur du tableau concerné (ex : &d_octree_points dans &(d_octree→points)). Cela permet d'indiquer au tableau de la structure « où se trouve son emplacement mémoire dédié sur le GPU »
    - Enfin, on copie effectivement les données dans cet emplacement mémoire (ex : octree→points dans d_octree_points)
- Lignes 1526-1562 : Allocation et copie de données dans le cas de 2 proc MPI ou plus
  - Ici on va devoir copier des double pointeurs (tableaux de tableaux)
  - On va allouer le premier tableau normalement (ex : d_tmp_send_tgt_lnum)
  - On utilise un « host pointer » pour pouvoir allouer les sous-tableaux (préfixe « h_ »). Ce pointeur est aussi un double pointeur, de même taille que le premier. On va allouer les adresses des sous-tableaux sur GPU (ex : &h_tmp_send_tgt_lnum[i])
  - Attention : les éléments de ce pointeur sont alloués sur GPU, mais le pointeur en lui-même reste sur CPU. C'est important pour effectuer les copies par la suite.
  - Pour indiquer au premier tableau « où sont les emplacements mémoire de ses sous-tableaux », on va copier le host pointer dans le device pointer (ex : h_tmp_send_tgt_lnum dans d_tmp_send_tgt_lnum)
- Lignes 1564-1572 : Permet de regarder la mémoire utilisée, et la mémoire libre restante après toutes les allocations et copies. Note : on peut voir la mémoire totale du GPU en

tapant « nvidia-smi » dans le terminal

- Lignes 1575-1581 : Augmentation à 1 GB de la limite d'allocation, pour les malloc qui sont dans le kernel. Par défaut la limite est très faible (environ 8 KB je crois).
- Lignes 1583-1588 : variables pour l'exécution du kernel
- Lignes 1590-1706 : lancement du kernel pour faire les calculs
  - Le code est fait pour permettre un relancement du kernel si on a pas assez de mémoire pour faire tous les calculs en 1 seule itération (normalement pas besoin pour moins de 10 millions de points cibles)
  - n_threads contient le nombre de threads par blocs (maximum 1024, donc 1024 en 1D ou 32*32 en 2D)
  - n_blocks contient le nombre de blocs de threads à utiliser. Pour être sur d'avoir suffisamment de threads pour le calcul, on fait `(n_pts + 1023)/1024` ce qui permet par ailleurs de s'assurer qu'on a au minimum 1 bloc si on a moins de 1024 points.
  - Pour appeler le kernel, on indique le nombre de threads et de blocs. Le nombre total de threads est la multiplication des 2. Ensuite, on indique en argument les pointeurs alloués sur GPU. Pour des variables contenant simplement une valeur, on peut directement les utiliser
  - Si le nombre de points dépasse la limite que l'on s'est fixée (points_limit, par exemple lorsque l'on a un dépassement mémoire), on envoie un certain nombre de points, puis on incrémente le compteur de calculs effectués (points_shift est le décalage de points, points_nb est le nombre de points restant à calculer)
  - cudaDeviceSynchronize est la même chose qu'un MPI_Barrier
- Lignes 1709-1759 : recopie des données résultats sur le CPU, mêmes principes que précédemment
- Lignes 1761-1803 : on libère la mémoire allouée sur GPU

- Lignes 1892-1976 : prototype pour une fonction de création d'octree sur GPU
- Lignes 1979-3236 : Fonction qui va appeler _closest_points_local
  - Lignes 2037-2191 : première implémentation GPU faite un peu à l'aveugle, fonctionne mais n'a pas d'intérêt car l'accélération de cette section est insignifiante
  - Sinon, tout est exactement pareil que dans la version séquentielle. L'accélération se passe dans le _closest_points local (ligne 2778)
- Lignes 3240-3325 : prototype pour la libération de mémoire allouée sur GPU pour l'octree

PDM_para_octree.cuh

- Header pour PDM_para_octree.cu

Notes

- Il y a quelques modifications orphelines qui trainent dans certains fichiers .c ou .h, principalement pour régler des problèmes de compatibilité dus au fait que NVCC est dérivé d'un compilateur C++ (déplacement des déclarations de structures par exemple)
- Demander à la DSI s'il faut installer une version de openmpi avec CUDA-aware MPI. Si mon stck n'est pas effacé, l'installation se trouve à cet endroit : /stck/alafonta/ avec les dossiers cuda-aware-mpi et openmpi-cuda
- On peut utiliser la « mémoire unifiée » pour laisser le GPU gérer les transferts de données (cf cudaMallocmanaged)
- On peut mettre en pause le kernel avec « __brkpt() ; », et l'équivalent d'un abort pour quitter l'exécution d'un kernel serait « __trap() ; »
- Il faut éviter de faire des indirections (ex : tab[idx[i]]) car cela ralentit
- On peut appeler de nouveaux kernels pour paralléliser encore davantage (par exemple les boucles for dans le kernel) – cf parallélisme dynamique