RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Exploitation et partage de GPU dans des grappes de calcul hybrides

Matthieu Ospici[1,2,3], Luigi Genovese[4], Thierry Deutsch[2]

[1] BULL SAS Grenoble, 1 rue de Provence, 38130 Echirolles, France
[2] CEA, Laboratoire de Simulation Atomistique (L Sim), 17 Av. des Martyrs, 38054 Grenoble, France
[3] LIG, Laboratoire d'Informatique de Grenoble, équipe INRIA Mescal, ZIRST 51, avenue Jean Kuntzmann, 38330 Montbonnot, France
[4] ESRF, European Synchrotron Radiation Facility, 6 rue Horowitz, BP 220, 38043 Grenoble, France

---

Résumé
Dans cet article, nous étudions la programmation et l'exploitation des architectures de calcul hybrides. Nous nous intéressons plus précisément aux grappes de calcul hybrides constituées d'un ensemble potentiellement grand de cœurs généralistes et de quelques accélérateurs (GPU). Nous analysons les différentes approches à mettre en œuvre pour que les cœurs puissent utiliser efficacement les accélérateurs (GPU). Cette analyse a débouché sur la définition et l'implémention d'une bibliothèque de partage de GPU entre différents cœurs : la bibliothèque `S_GPU`. Nous comparons l'utilisation et les performances de cette bibliothèque avec d'autres approches pour partager les accélérateurs entre les cœurs. Cette évaluation a été effectuée sur le code de simulation atomistique BigDFT.

Mots-clés : Multi-GPU, Multi-cœurs, Partage de ressources, Grappe hybride, Simulations atomistiques.

---

1. Introduction

Le concept de processeur multicœur a été appliqué avec succès aux processeurs des cartes graphiques (GPUs). Ces dernières possèdent des centaines de cœurs et permettent d'atteindre une puissance théorique de plusieurs centaines de Gflops. Il est donc particulièrement tentant d'utiliser ces GPUs, non plus pour des jeux vidéo, mais pour du calcul généraliste (GPGPU, General-Purpose computation on Graphics Processing Units) ; les ordinateurs contenant des GPUs et des cœurs généralistes sont aujourd'hui communément appelés « architecture hybride ». Dans cet article, nous nous intéressons à des simulations pour les nanosciences et les nanotechnologies.
La programmation et l'exploitation des architectures hybrides à base de GPUs se heurtent à plusieurs difficultés. Nous allons ici nous intéresser à l'exécution de programme sur des grappes de calcul hybrides constituées de nœuds SMP avec des cartes graphiques (GPUs). Généralement, sur un nœud hybride, le nombre de cartes graphiques sera inférieur à celui des cœurs CPU. Par conséquent, à l'intérieur d'un nœud SMP, des processus ou des threads placés sur différents processeurs peuvent demander simultanément un calcul à des GPUs. Les accès concurrents de plusieurs processus aux GPUs sont au cœur de cet article.
Notre contribution est la bibliothèque logicielle `S_GPU` qui permet de partager un ou plusieurs GPUs NVIDIA entre plusieurs processus. Nous avons utilisé la bibliothèque `S_GPU` dans le code de simulation atomistique BigDFT [2] (code fortran + MPI). BigDFT a déjà été adapté [1] à l'utilisation des GPUs grâce au langage CUDA[3]. Sur un nœud SMP, plusieurs processus MPI sont lancés par BigDFT. Ils peuvent au choix utiliser une des cartes graphiques NVIDIA du nœud pour faire certains calculs ou utiliser un cœur CPU. BigDFT est un code qui possède de très bonnes propriétés de scalabilité et qui est très régulier ; toutes les tâches MPI déroulent exactement les mêmes calculs sur des données différentes. Nous utiliserons notre bibliothèque dans BigDFT et nous étudierons les performances du partage de GPU avec `S_GPU` dans une grappe hybride.

La suite de l'article est organisée de la façon suivante : la première partie sera consacrée à une description des architectures hybrides à base de GPUs et aux applications existantes qui les exploitent. Nous présenterons ensuite les choix de conception et d'implémentation de la bibliothèque logicielle `S_GPU`. Nous terminerons cet article par une étude de performance en comparant `S_GPU` à une approche plus statique du partage.

2. Exploitation des grappes de calcul hybrides

La tendance actuelle dans les grands centres de calcul est de proposer des grappes de calcul hybrides basées sur des architectures de processeur généraliste et d'accélérateur. L'exemple le plus connu à ce jour est la machine Roadrunner, classée en tête du classement du TOP500. Dans cet article, nous étudierons les architectures hybrides à base de GPUs NVIDIA.

2.1. Grappes de calcul hybrides

Une grappe de calcul hybride à base de GPUs est une grappe de calcul où certains nœuds sont hybrides (ils ont un ou plusieurs GPUs), et où les autres nœuds n'ont aucun GPU. Typiquement, sur un nœud de grappe hybride, on trouve quatre ou huit cœurs CPU, un ou deux Intel Xeon quadricœur par exemple, associés à un ou deux GPUs. Lors de nos expérimentations sur des grappes hybrides, nous avons uniquement utilisé les nœuds possédant des GPUs.
Tous les GPUs utilisés sont de la génération GT200 et permettent de faire de manière simultanée des calculs et des transferts mémoire, nous pouvons ainsi recouvrir des transferts mémoire par des calculs.
Étudions maintenant quelques exemples significatifs d'applications scientifiques exploitant ce type d'architecture et les limites de leurs approches.

2.2. Quelques exemples d'applications hybrides

La littérature fait état d'utilisations réussies de différents codes de calcul sur des grappes hybrides à base de GPUs. Par exemple, pour la mise en œuvre du benchmark LINPACK [7], NVIDIA utilise une méthode qui permet d'utiliser tous les CPUs et tous les GPUs des nœuds d'une grappe hybride. Pour ce faire, la matrice utilisée par LINPACK est découpée en plusieurs parties. Dans le cas où il y aurait un GPU et plusieurs CPUs, la matrice sera découpée en deux parties de tailles inégales. Une des parties s'exécutera sur le GPU en faisant des appels à la bibliothèque NVIDIA CUBLAS [3], laquelle met en œuvre des calculs d'algèbre linéaire sur GPU. Le second bloc de la matrice sera quant à lui calculé sur les CPUs grâce à la MKL, une bibliothèque de calcul scientifique multithreadée. Il suffit donc de spécifier à la MKL le nombre de cœurs CPU disponibles pour pouvoir tous les utiliser. Lorsque le GPU et les CPUs sont venus à bout de leurs calculs, les deux morceaux de la matrice sont combinés afin d'obtenir le résultat final. Tout le problème de cette approche est la découpe de la matrice, il faut placer de manière statique une certaine quantité de données sur les GPUs et sur les CPUs car leurs performances en calculs sont différentes.
Une autre approche consiste à créer sur un nœud autant de tâches MPI que de cœurs CPU, chaque tâche MPI utilisant un GPU. Nous pouvons citer NAMD [5], code de dynamique moléculaire, comme code utilisant cette approche et FEAST [6], qui permet de faire des calculs des éléments finis. Les expériences ont été menées sur des architectures à base de deux processeurs AMD bi-cœurs et de quatre cartes graphiques pour NAMD ; à base d'un processeur et d'une carte graphique pour FEAST. Les auteurs de FEAST et NAMD n'ont pas étudié le cas où il y a moins de GPUs que de cœurs généralistes (CPUs).
Dans une machine contenant un nombre différent de GPUs et de CPUs, le problème soulevé est le placement des données ; la question est donc de déterminer la quantité de données que chaque cœur CPU et chaque GPU doit traiter. La répartition (ou placement) globale des données entre les GPUs et les cœurs CPU dans le LINPACK de NVIDIA est efficace pour une application effectuant une seule opération matricielle. Par contre elle est peu intéressante pour une application comportant plusieurs tâches MPI désirant accéder aux GPUs de manière concurrente comme NAMD.
Pour exploiter efficacement une architecture SMP hybride, nous avons choisi d'évaluer une stratégie où les GPUs sont partagés par un groupe de cœurs CPU dans un nœud SMP grâce à la bibliothèque `S_GPU` ; tous les cœurs CPU qui partagent les GPUs possèdent une même quantité de données. Nous présentons dans la partie suivante la bibliothèque `S_GPU`. GPU.

3. S_GPU, une bibliothèque pour partager les GPUs

S_GPU a pour but de partager les accès à une ou plusieurs cartes graphiques par un ensemble de processus sur un nœud multiprocesseur. S_GPU permet également d'optimiser les temps de transfert mémoire en les recouvrant avec des calculs.
L'objet de cette section est de décrire cette bibliothèque logicielle de manière détaillée.

3.1. Présentation de la bibliothèque

S_GPU est une couche logicielle qui se place entre des fonctions de pilotage de la carte et la carte graphique elle-même (fig 1). Chaque processus exécute ses opérations sur la carte en utilisant les fonctions fournies par S_GPU. S'il y a plusieurs cartes GPU disponibles, S_GPU va automatiquement attribuer à chaque processus un GPU, ce donc n'est plus au programmeur de faire cela.
La bibliothèque S_GPU est construite autour de la notion de flot. Un flot de S_GPU peut être vu comme une liste de dépendances, si l'on empile les trois opérations A, B, C dans un flot, cela indique que l'opération B a besoin que l'opération A soit finie pour être exécutée, et de même pour C qui a besoin de B.

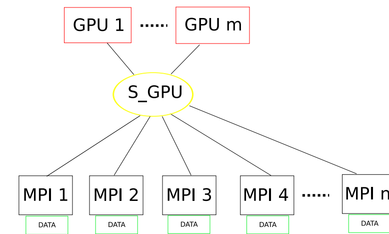

FIG. 1 – S_GPU se place entre l'application et les cartes graphiques (GPUs)

Lors d'un appel à une fonction de la bibliothèque, comme un transfert mémoire, la commande est empilée dans un flot et n'est pas exécutée immédiatement sur la carte. Il est possible d'empiler un nombre quelconque d'opérations dans un flot et d'avoir un nombre quelconque de flots. Si l'on veut par exemple effectuer cent fois la chaîne d'opérations : transfert CPU → GPU , calcul, transfert GPU → CPU, on peut créer cent flots qui contiennent chacun les trois commandes transfert CPU → GPU , calcul, transfert GPU → CPU.
Ensuite, pour exécuter tous ces flots, il suffit d'appeler une fonction de la bibliothèque : launch_all_streams().
Grâce à cette notion de flot, il est possible d'imaginer certaines optimisations, mais dans la version actuelle de S_GPU les flots sont exécutés de manière séquentielle sans aucune optimisation.

3.1.1. Principales fonctions et exemple de code

S_GPU se place au même niveau d'abstraction que CUDA (ou OPENCL[4]) et est écrite principalement en C++ avec quelques appels à des fonctions CUDA pour gérer les cartes graphiques. Beaucoup d'applications scientifiques — dont BigDFT — utilisent le langage Fortran. Nous avons donc offert au programmeur le moyen d'utiliser S_GPU nativement en fortran. Les exemples de code présentés dans cette partie sont tous en fortran et proviennent de la version de BigDFT qui utilise S_GPU.

Interface de programmation

`init_gpu_sharing(int NB_PROC_Par_nœud, int NB_GPU)` Initialise les processus d'un même nœud et les associe aux cartes graphiques. Il faut y préciser le nombre total de processus et le nombre de GPUs à partager.

`create_stream(streamPtr)` Crée un nouveau flot et renvoie un pointeur sur celui-ci.

`launch_all_streams()` Exécute toutes les instructions placées dans les flots créés.

`gpu_allocate(int size, GPUptr, errCode)` Alloue de la mémoire sur GPU et renvoie un pointeur sur cette zone mémoire.

`cpu_pinned_allocation(int size, CPUptr, errCode)` Alloue de la mémoire punaisée sur la mémoire centrale du CPU. Nous verrons plus loin que cette fonction est indispensable pour les recouvrements entre les calculs et les transferts sur GPU.

`gpu_send_st(int nsize, CPU_pointer_pi, GPU_pointer, errCode, streamPtr)` Ajoute dans le flot passé en paramètre une instruction permettant de faire un transfert mémoire du CPU au GPU. Il existe une fonction symétrique pour le transfert des données du GPU vers le CPU.

`calcul(param...)` Exécute un calcul sur le GPU. Dans cette version de S_GPU, il faut créer, pour chaque calcul, une fonction permettant de s'interfacer avec la bibliothèque.

Exemple de code

```
1        call create_stream(stream_ptr)
2
3        do iorb=1,num_orbs
4            call mem_copy_st(... ,
5                            stream_ptr)
6            call gpu_send_st(... ,
7                            stream_ptr)
8        end do
9        call calcul(... ,
10                   stream_ptr)
11       call gpu_recv_st(... ,
12                           stream_ptr)
13       call mem_copy_st(... ,
14                           stream_ptr)
15
16       call launch_all_streams()
```

Cet exemple, issu de BigDFT, certes simplifié mais conservant le même squelette que le code réel, crée un flot, effectue plusieurs transferts mémoire du CPU au GPU, lance un calcul et rapatrie les données du GPU vers la mémoire CPU. Il faut noter qu'il y a deux opérations en plus dans ce code : les deux `mem_copy_st` (ligne 4 et 13). Cette fonction fait une copie d'un tableau de données vers une zone mémoire CPU allouée par `cpu_pinned_allocation`; les transferts GPU → CPU (l.6) ou CPU → GPU (l.11) se font à partir de cette zone mémoire. Ceci est fait afin de pouvoir recouvrir les transferts et les calculs — i.e. faire des transferts mémoire en même temps que des calculs — car CUDA contraint à avoir des pages mémoire « punaisées », c'est-à-dire des pages mémoire bloquées que le système de pagination ne peut pas réutiliser.
La l.16 permet l'exécution de toutes les commandes empilées dans le flot créé.
Dans la section suivante, nous présenterons de manière détaillée le fonctionnement de la bibliothèque. `S_GPU`

3.1.2. Mise en œuvre du partage des GPU

Plusieurs solutions permettent de partager des GPU. La première, qui est aussi la plus simple, consiste a laisser le GPU ordonnancer lui même les opérations sur la cartes. Cela est tout a fait possible, chaque processus MPI demandant alors un calcul ou un transfert à n'importe quel moment.

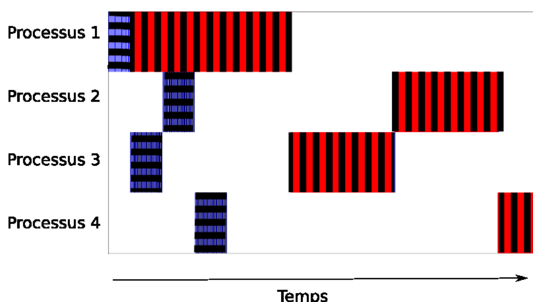

FIG. 2 – Trace d'exécution : lignes horizontales : transferts, lignes verticales calculs. Nous observons que les transferts sont recouverts par les calculs

A l'inverse, pour `S_GPU`, nous avons choisi de synchroniser l'accès à la carte. Avec cette approche, les processus peuvent accéder à la carte uniquement si elle libre. Pour mettre en œuvre cela, nous avons décider d'utiliser des sémaphores UNIX. Étant donné qu'il est possible d'avoir des transferts et des calculs simultanément sur les cartes NVIDIA récentes, nous utilisons deux sémaphores par carte ; le premier pour gérer les accès concurrents en mémoire, le second pour gérer les demandes de calcul concurrentes. Grâce à cela, si un processus a le sémaphore « calcul », un autre processus peut faire un transfert simultanément si le sémaphore « transfert » est libre. Par contre, il est bien entendu impossible qu'un processus fasse un calcul si le sémaphore « calcul »est pris par un autre processus.
Lors de l'appel de `init_gpu_sharing(int NB_PROC, int NB_GPU)`, la bibliothèque `S_GPU` va assigner à chaque processus une carte graphique. Par exemple, s'il y a huit processus et deux GPUs, quatre processus auront la première carte, les quatre autres la seconde. Chaque groupe de quatre processus possédera deux sémaphores.
Afin de visualiser ce qu'il se passe, nous avons développé un petit outil de trace qui permet d'afficher, sous forme graphique, les opérations des processus à un instant t. Nous avons exécuté le code présenté précédemment avec quatre processus MPI qui partagent un GPU. Une trace de l'exécution interprétée par notre outil est affichée figure 2.
Les lignes horizontales correspondent aux transferts, les lignes verticales aux calculs. Le travail des sémaphores est clairement visible : lorsque un processus a le sémaphore transfert, les autres ne peuvent pas faire de transferts mais ils peuvent lancer un calcul. Ainsi, on ne paye que le premier transfert, les

autres sont totalement recouverts par du calcul.

4. BigDFT, le code de simulation scientifique utilisé

BigDFT est un logiciel libre (GNU – GPL) qui permet de faire des simulations *ab-initio*. Dans ce type de simulations, on part des propriétés de base des atomes et on va en déduire le comportement de plus grosses structures. Nous allons détaillé dans cette section les calculs effectués par BigDFT et comment les GPUs sont utilisés.

4.1. Exploitation par BigDFT des grandes grappes de calcul

BigDFT est un code de simulation scientifique réalisé en fortran et qui utilise MPI pour communiquer dans les grandes grappes de calcul. L'efficacité de la parallélisation est très bonne, de l'ordre de 95%.

BigDFT utilise autant de tâche MPI que de cœurs CPU présents dans la grappe hybride utilisée. Ainsi, sur un nœud de calcul de la grappe, un ensemble de tâches MPI est créé (avec donc autant de tâches MPI que de cœurs CPU comptés dans le nœud). Ensuite, chacune des tâches MPI utilise le GPU pour exécuter une partie des calculs. Si le partage de GPUs n'est pas activé, il est possible de spécifier quelle(s) tâche(s) MPI peuvent utiliser les GPUs et quelles tâches utilisent uniquement le CPU.

BigDFT fonctionne de manière itérative, la liste suivante montre les opérations effectuées lors d'une itération.

1. Construction de la densité (**)
2. Solveur de poisson (**)
3. Hamiltonien local (**)
4. Hamiltonien non-local
5. Préconditionneur (**)
6. Mise à jour des fonctions d'ondes
7. Orthogonalisation

Entre les étapes 4 et 5, 5 et 6 et 6 et 7, ont lieu des communications MPI.

Les étapes marquées de (**) utilisent le GPU, ce sont principalement des calculs de convolution qui sont détaillés dans [1].

Il faut bien noter que, lors d'une itération, seule une partie des calculs se fait sur les GPUs.

4.1.1. La distribution des orbitales dans les tâches MPI

Dans BigDFT, toutes les tâches MPI font exactement les mêmes opérations. Ce sont les données que l'on va partager. Le type de données élémentaire est appelé *orbitale*. Lorsque l'on fait une simulation sur un système physique, ce système possède un certain nombre d'orbitales n_{orbs}. Supposons que lors d'une exécution, BigDFT crée n_{MPI} tâches MPI. Sans manipulation supplémentaire, le nombre d'orbitales traité par chaque processus est : n_{orbs}/n_{MPI}. On appelle cette répartition, « répartition homogène », ou « dynamique »

On peut également avoir une répartition non-homogène, dans ce cas, il est possible de spécifier précisément le nombre d'orbitales qu'un processus MPI particulier va traiter. On appelle cette répartition « répartition non-homogène », ou « statique ».

Les différentes répartitions d'orbitales présentées ici vont nous permettre d'évaluer les performances de `S_GPU` en comparant notre bibliothèque à d'autre approche de répartition de donnée. Ce sera l'objet de la prochaine section.

5. Évaluation des performances

Les expérimentations avec BigDFT et `S_GPU` ont été principalement effectuées sur la grappe de calcul hybride IBLIS du CINES. Nous allons décrire dans la section suivante les caractéristiques des plates-formes utilisées pour les expériences. Nous présenterons ensuite des résultats d'expériences à l'intérieur d'un noœud hybride de calcul et des expériences sur une grappe de calcul hybride.

5.1. Contexte expérimental

La société Bull a mis à notre disposition une plate-forme de développement et d'évaluation à base de processeurs Intel Xeon X5472 quadri-cœur, 3.00GHz et d'un espace mémoire de 16 Gigas Octets. Cette plate-forme dispose de deux accélérateurs GPU de type NVIDIA TESLA C1070 (4GB de mémoire). Nous

avons été contraints d'utiliser la bêta version 2.2 de CUDA pour permettre le partage d'un GPU par les 8 coeurs de 2 processeurs (la version stable 2.1 de CUDA ne le permettant pas !).
Pour les expériences au niveau grappe hybride, nous avons utilisé la grappe IBLIS du CINES. Les nœuds de cette grappe sont basés sur la même architecture que le nœud mis à disposition par Bull, mais avec seulement 8 GB de RAM par nœud. C'est la version 2.1 de NVIDIA CUDA qui est installée sur la grappe IBLIS du CINES.
Les compilateurs utilisés pour les expériences sont ceux de la société Intel (version 10.1 avec les options de compilation `-O2 -xT`). Toutes les expériences seront menées avec le code BigDFT et la configuration suivante : 30 atomes et 144 orbitales.

5.2. Expériences sur un nœud de calcul hybride

Deux types d'expériences vont être menés. Un premier type d'expériences où la répartition des orbitales (données) sera régulière (homogène) sur les processus (tâches MPI). Chaque tâche MPI s'exécutera sur un cœur et toutes les tâches MPI auront le même volume de calcul à effectuer. Toutes les tâches MPI du programme accéderont donc au GPU. Avec ce type d'expérience, nous évaluerons l'intérêt d'utiliser la bibliothèque `S_GPU`.
Un second type d'expérience sera effectué où la répartition des orbitales (données et calculs) sera irrégulière sur les processus (tâches MPI). Certaines tâches MPI auront le privilège d'utiliser un GPU et disposeront donc d'une capacité de calcul plus importante. Pour ces tâches privilégiées, le nombre d'orbitales qui leur sera assigné sera donc plus important. Ce type d'expérience nous permettra d'évaluer le coût du partage du GPU entre les différentes tâches.

5.2.1. Répartition régulière des données

Dans cette sous section, nous évaluons l'utilisation d'un GPU par plusieurs processus (tâches MPI). Nous avons fait varier le nombre de tâches MPI (cœurs qui s'exécutent dans le nœud (les colonnes du tableau 1).
Sur la première ligne du tableau 1, on trouve le temps d'exécution de BigDFT sans utilisation de GPU. Sur la seconde ligne, la répartition des données est régulière ; une tâche utilise de manière exclusive le GPU, les autres tâches calculent sur un cœur. Pour la troisième ligne„ la répartition est toujours régulière, mais les tâches accèdent toutes directement au GPU. C'est donc le GPU qui assure l'ordonnancement des calculs. Pour la dernière ligne du tableau, c'est la bibliothèque `S_GPU` qui ordonne les calculs sur le GPU.

Cœur(s) utilisés (nb tâche MPI)	1	2	3	4	6	8
Sans GPU (s)	268	142	95,1	74,3	56,5	46,7
1 GPU, rep. régulière (s)	55,8	137	91,1	73,3	54,0	45,8
1 GPU, sans synchronisation (s)	56,1	61,1	55,8	53,8	52,4	51,7
1 GPU partagé avec `S_GPU` (s)	**56,7**	**46,4**	**41,2**	**38,7**	**38,2**	**37,1**

TAB. 1 – Temps d'exécution de BigDFT en fonction du nombre du cœur CPU utilisé avec 0 GPU, 1 GPU partagé avec `S_GPU` et sans synchronisation

La dernière ligne du tableau met en évidence de meilleures performances par rapport aux deux lignes précédentes. Le gain est assez significatif par rapport à la troisième ligne où le partage était assuré par le GPU. Pour la seconde ligne, les résultats s'expliquent que même si la tâche liée au GPU est intrinsèquement plus rapide, elle doit se synchroniser avec les autres tâches et cette synchronisation lui coûte chère. Ce tableau montre aussi globalement les différents gains mesurés avec une application utilisant des GPUs.
Dans la suite de cette section, nous allons briser cette régularité en plaçant plus de données (orbitales) sur certaines tâches afin de savoir si cette approche peut être meilleure que l'utilisation de `S_GPU`.

5.2.2. Répartition irrégulière des données

Nous allons maintenant nous intéresser à une approche où la répartition des données est irrégulière. Pour cela, une tâche MPI de BigDFT est liée au GPU, les autres tâches MPI utilisent uniquement les

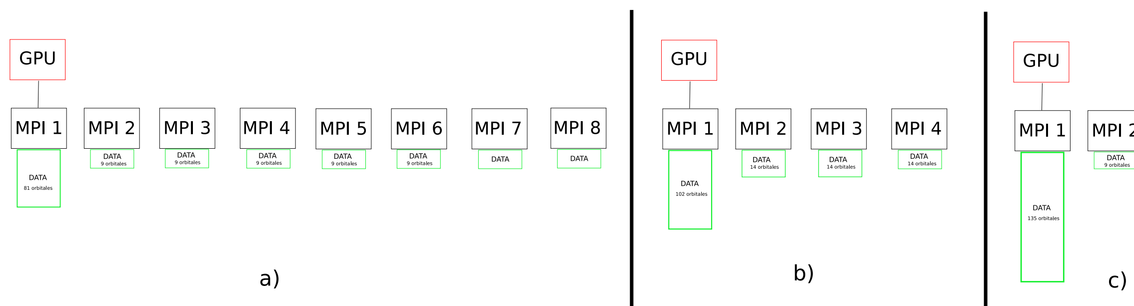

FIG. 3 – Répartition irrégulière des données : les figures a), b) et c) représentent trois configurations différentes.

cœurs pour faire leurs calculs. La tâche assignée au GPU aura plus de données et de calcul à effectuer. Les expérimentations ont été menées sur 3 configurations présentées dans le tableau 1.
Pour les expérimentations suivantes, nous avons procédé de manière empirique pour déterminer les répartitions statiques optimales des orbitales qui sont données dans le tableau 1.

Configuration à 2 tâches MPI

Dans cette configuration (fig 3 c)), la répartition optimale est de 135 orbitales pour la tâche connectée au GPU et 9 pour l'autre tâche. Le temps d'exécution est de $52,4$ s (contre $46,4$ s pour la version partagée, cf tab 1). Le partage des GPUs avec `S_GPU` est meilleur, il n'existe aucune répartition des orbitales qui permette à la répartition irrégulière d'être plus performante. Notons que pour le cas avec quatre processus, fig 3 b) `S_GPU` est aussi toujours meilleur.

Configuration à 8 tâches MPI

Dans cette configuration (figure 3 a)), sept tâches MPI effectuent leurs calculs exclusivement sur CPU, une seule tâche utilise le GPU. Nous avons expérimenté avec plusieurs répartitions des données, celle qui nous a donné le meilleur résultat est celle avec 81 orbitales sur la tâche liée au GPU et 9 orbitales sur les sept autres tâches MPI. Nous obtenons un temps de $34,9$ s, contre $37,1$ s pour la version partagée (cf tab 1). Ainsi, pour huit processus MPI utilisés, en répartissant les données de manière non homogène, nous obtenons un temps d'exécution total de BigDFT légèrement plus rapide que dans le cas où les processus sont partagés avec `S_GPU`.

Analyse

Pour une simulation qui traite n_{orbs} orbitales, le temps d'exécution peut globalement se calculer avec la somme du temps CPU T_{CPU} et du temps GPU T_{GPU}.
En utilisant `S_GPU`, les accès à la carte sont synchronisés. Ainsi, en répartissant les données de manière homogène sur plusieurs processus, T_{GPU} reste constant. C'est T_{CPU} qui diminue avec l'accroissement du nombre de processus et de cœurs. Par conséquent, avec un grand nombre de cœurs et tâches MPI, le temps d'exécution de BigDFT sera fortement dépendant et limité par T_{GPU}.
Comme le jeu de données et la simulation sont petits, cette limitation peut rapidement être atteinte comme on commence à l'observer sur la dernière ligne du tableau tab 1 : à partir de quatre processus, le temps d'exécution reste quasiment constant. Pour évaluer la scalabilité des performances et du partage [8], nous allons prochainement faire évoluer la taille du jeu de données et de la simulation en fonction du nombre de cœurs.
Ceci dit, nous pouvons tout de même conclure que sur un nœud hybride, avec BigDFT, `S_GPU` offre de bonnes performances, supérieures même dans la majorité des cas à une approche irrégulière. Ainsi, grâce à `S_GPU`, pour ce type d'application, il n'est pas nécessaire d'effectuer une répartition irrégulière pour avoir de bonnes performances. Nous allons maintenant effectuer des expériences sur la grappe hybride IBLIS du CINES.

5.3. Expériences sur une grappe de calcul hybride

Des expérimentations avec S_GPU vont maintenant être menées sur plusieurs nœud d'une grappe hybride. Pour cela, aucune modification du code MPI n'est nécessaire. Sur les nœuds de la grappe sur lesquels S_GPU est utilisé, le partage de GPU est effectué de manière transparente pour l'utilisateur.

Nous avons exécuté BigDFT sur la grappe de calcul hybride IBLIS du CINES sur 3, 4, 5, 6, 7 et 8 nœuds hybrides. Le système que nous avons simulé possède 576 orbitales. Sur huit nœuds, BigDFT utilise $8 * 8 = 64$ cœurs CPU et $8 * 2 = 16$ GPUs. Le tableau 2 présente les temps d'exécution et le facteur d'accélération entre une exécution sans GPU et avec S_GPU.

Comme sur les nœuds de la grappe, nous utilisons deux cartes graphiques pour 8 coeurs CPU. La bibliothèque S_GPU va automatiquement attribuer à quatre processus la première carte et aux quatres autres la seconde carte.

Nombre de nœud de la grappe IBLIS	4	5	6	7	8
Sans GPU	1220	987	804	704	608
Avec 2 GPUs par nœud partagés avec S_GPU (s)	456	281	234	204	174
Accélération	**2,68**	**3,51**	**3,44**	**3,45**	**3,49**

TAB. 2 – Temps d'exécution en fonction du nombre de nœuds d'IBLIS utilisés. S_GPU partage deux GPUs pour huit processus MPI donc chaque GPU est associé à quatre processus sur chaque nœuds d'IBLIS

A l'aide de S_GPU, le facteur d'accélération obtenu grâce aux GPUs reste très stable avec la grappe de calcul IBLIS du CINES. Des expérimentations du même type vont être très prochainement menées sur la nouvelle grappe de calcul en cours d'installation au GENCI-CEA.

6. Conclusion et perspectives

Dans cet article, nous avons présenté la bibliothèque S_GPU qui permet de partager de manière efficace des GPUs entre plusieurs cœurs généralistes. Nous avons utilisé avec succès S_GPU sur un nœud et sur les nœuds d'une grappe hybride. Nous avons comparé S_GPU avec d'autres approches d'exploitation de GPUs. Pour le cas de l'application BigDFT, la bibliothèque S_GPU a permis d'obtenir de très bonnes performances. Il reste bien évidemment à démontrer l'efficacité de S_GPU avec d'autres types d'applications et simulations dont les comportements sont moins réguliers que BigDFT. Des expérimentations vont démarrer sur la grappe de calcul du GENCI-CEA. Sur cette grappe, les GPUs sont les mêmes que ceux de IBLIS, la différence se situe au niveau des processeurs généralistes Intel Nehalem.

Bibliographie

1. L. Genovese, M. Ospici, T. Deutsch, J.F. Méhaut, A. Neelov, S. Goedecker, Density Functional Theory calculation on many-cores hybrid CPU-GPU architectures, 2009, http://www.citebase.org/abstract?id=oai:arXiv.org:0904.1543
2. L. Genovese et al., J. Chem. Phys 129, 014109 (2008), http://inac.cea.fr/sp2m/LSim/BigDFT
3. NVIDIA CUDA Programming Guide, version 2.1, and CUBLAS http://www.nvidia.com/object/cuda_home.html
4. OPENCL http://www.khronos.org/opencl/
5. J.C. Phillips, J.E. Stone, K. Schulten, Adapting a message-driven parallel application to GPU-accelerated clusters, SC '08 : Proceedings of the 2008 ACM/IEEE conference on Supercomputing,
6. D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S.H.M. Buijssen, M. Grajewski , S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Comput., vol 33, 2007, pages 685–699
7. M. Fatica, Accelerating linpack with CUDA on heterogenous clusters, GPGPU-2 : Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pages 46-51
8. I. Foster, Designing and Building Parallel Programs, Addison Wesley, 1995