Programmation des architectures hétérogènes à l'aide de tâches divisibles

Gwenolé Lucas

Programmation des architectures hétérogènes à l'aide de tâches divisibles

Rapport de stage

Inria - Équipe STORM

Enseirb-Matmeca - Filière Informatique - PRCD

Gwenolé Lucas

Encadrants :

Mathieu Faverge,
Nathalie Furmento,
Abdou Guermouche,
Pierre-André Wacrenier

Septembre 2019

Table des matières

1 Introduction

Depuis leur entrée sur le marché au début des années 2000, les processeurs multi-cœurs sont devenus indispensables à l'augmentation de la puissance de calcul de nos ordinateurs, rendant nécessaire un travail de parallélisation de nos algorithmes afin d'en tirer le maximum. À cela s'ajoute l'introduction des accélérateurs de calculs, des circuits spécialisés, par exemple des GPU, capables de réaliser certaines opérations bien plus efficacement qu'un processeur traditionnel. Ces architectures hétérogènes sont difficiles à programmer car elles complexifient les questions d'ordonnancement, de gestion de la mémoire, d'équilibrage de charge, *etc.*

StarPU [1] est un support d'exécution (ou *runtime*) dont l'objectif est d'adresser ses questions et de permettre l'écriture de programmes parallèles [1] performants à plus haut niveau. Pour cela il utilise un modèle à base de tâches, c'est à dire des séquences d'instructions définies par l'utilisateur formant des « unités de travail » qu'il utilise ensuite dans son programme, StarPU prenant en charge, entre autres, l'ordonnancement et les transferts mémoire entre les ressources de calcul lors de l'exécution.

L'objectif du stage est d'étudier le concept de tâches hiérarchiques dans StarPU. Leur usage permettrait en effet à StarPU d'adapter la granularité, c'est à dire la quantité de calcul générée par chaque tâche, pendant l'exécution afin de mieux répartir la charge entre les CPU, en général plus efficaces avec une granularité fine, et les GPU, demandant une grosse granularité pour exploiter leur plein potentiel. Cela se traduit par des problèmes d'ordonnancement et de partitionnement, un moyen de réduire la granularité d'une tâche étant de soumettre un ensemble de tâche travaillant sur des données plus petites mais réalisant la même opération.

Dans la section 2 je présente brièvement mon lieu de stage et dans la section 3, son contexte. StarPU étant au cœur du stage, j'y consacre la section 4. Dans la section 5 je présente le travail réalisé autour des tâches hiérarchiques pendant le stage et l'annexe A aborde le travail réalisé dans le cadre des deux mois passés au KIT (*Karlsruher Institut für Technologie*).

2 L'entreprise

Mon stage se déroule au sein de l'équipe STORM (STatic Optimizations, Runtime Methods) chez Inria Bordeaux - Sud-Ouest, un établissement public à caractère scientifique et technologique. L'équipe STORM est une équipe mixte (INRIA - Université de Bordeaux - CNRS - Bordeaux INP) travaillant principalement dans le domaine du Calcul Haute Performance sur des langages dédiés de haut-niveau, des supports d'exécution pour architectures hétérogènes et des outils d'analyse de performance. Elle comporte une vingtaine de membres.

3 Contexte du stage

L'évolution des architectures dédiées au calcul intensif révèle l'importance du rôle pris par les accélérateurs de calculs au cours des dernières années. Ainsi le système dominant le top 500 [2], *Summit*, tire l'immense majorité de ses 148.6 PFlops [2] de ses GPU NVIDIA Tesla V100. Ces accélérateurs sont couplés avec des CPU pour former les nœuds de calcul des plateformes de calcul intensif, dont le nombre peut varier de quelques dizaines à plusieurs milliers. Cette multitude de nœuds hétérogènes rend l'exploitation de la puissance de ces plateformes de plus en plus complexe. D'une part les performances obtenues par une application sur un système donné sont rarement portables, et le passage à un autre système nécessitera souvent de réécrire au moins partiellement le code pour retrouver des performances satisfaisantes. D'autre part il est nécessaire de distribuer au mieux les calculs à effectuer afin de minimiser l'*idle time* des processeurs, les temps morts où ils attendent le résultat d'autres calculs pour reprendre les leurs ; à cela s'ajoute une problématique d'équilibrage de charge, car plus un calcul est équitablement réparti entre les processeurs, plus son temps d'exécution pourra être réduit. Dans

1. Des programmes dont le travail peut être réparti sur différents processeurs pour réduire le temps d'exécution.
2. Flops : Floating-point operations per seconds

le cas des architectures hétérogènes, l'écart de puissance entre les CPU et les GPU exacerbe cette problématique car les premiers travaillent plus efficacement sur des petites quantités de calcul alors que les seconds, largement plus puissants, ont au contraire besoin de beaucoup de calculs sur pour être exploités à leur plein potentiel.

L'un des paradigmes du calcul intensif confronté à ces problèmes est celui de la programmation par tâche. Une application utilisant ce paradigme est mise sous la forme d'un DAG (*Directed Acyclic Graph*), un graphe dont les nœuds représentent des tâches et les arcs illustrent les dépendances devant être respectées pour assurer la correction du programme (par exemple, un arc $T_1 \rightarrow T_2$ indique que T_2 doit attendre la fin de T_1 pour être exécutée). L'apparition des problématiques ci-dessus a révélé le besoin d'affecter le graphe de tâche dynamiquement, c'est à dire de lui permettre d'évoluer au fur et à mesure de l'avancement du calcul, en fonction des ressources libres et de leurs performances. Cette approche a déjà été suivie par [3] et [4] qui s'intéressent respectivement aux tâches imbriquées d'OpenMP et aux DAG hiérarchiques dans le framework PaRSEC [5] (cf. sous-section 5.1). Les équipes de l'INRIA ont également implémenté cette idée de DAG hiérarchique grâce à la notion de tâches hiérarchiques, qui permettent de diviser une tâche en plusieurs tâches de plus petite granularité, ce qui permettrait d'affecter aux CPU et aux GPU des tâches comportant une quantité de calcul appropriée à leurs capacités respectives.

Le stage a donc pour objectif d'étudier ces nouvelles tâches hiérarchiques et leur utilisation. Une tâche hiérarchique va en effet avoir recours au partitionnement de la donnée qu'elle reçoit, afin de pouvoir soumettre des tâches d'une plus faible granularité, et à la récursion, les tâches filles étant susceptibles d'être elles-même hiérarchiques, créant ainsi plusieurs niveaux de granularité. Parvenir à établir une stratégie d'ordonnancement capable de gérer ces multiples niveaux de granularité et étudier les problèmes rencontrés lors de l'usage de tâches hiérarchiques sont les principaux buts du stage.

4 StarPU, un support d'exécution à base de tâches

Apparu en 2011, StarPU [1, 6] est un *runtime* visant à gérer l'exécution de programmes parallèles sur des architectures hétérogènes. Les programmes écrits pour StarPU suivent un paradigme de programmation par tâches (cf. section 4.1) et peuvent donc être mis sous la forme d'un graphe de tâches (cf. 3), que StarPU est chargé de répartir au mieux sur les différentes ressources de calcul, appelées *workers*, et d'ordonnancer leur exécution en respectant les différentes dépendances.

Bien qu'il soit possible de définir des dépendances explicitement entre deux tâches, elles sont principalement implicites et déduites des données qu'elles manipulent. Les données en question doivent être encapsulées dans un *handle* opaque pour pouvoir être utilisées par StarPU.

4.1 Tâches et codelets

Unités de base des applications utilisant StarPU, les tâches implémentées par le programmeur utilisent une abstraction appelée *codelets*. Un codelet rassemble, entre autres, la fonction contenant le code de la tâche, le nombre de handles en entrée et leur mode d'accès (`STARPU_[R|W|RW]`). Ce dernier argument affecte l'ordonnancement de la tâche associée. Par exemple si plusieurs tâches sont prêtes à être exécutées, et traitent une même donnée en lecture seule (`STARPU_R`) alors elles pourront être lancées en parallèle. *A contrario* si une tâche modifie une donnée D (`STARPU_[W|RW]`), les tâches suivantes manipulant D devront attendre sa terminaison.

D'autre part, il est possible de fournir au codelet plus d'une fonction, ce qui permet d'utiliser un même codelet pour plusieurs implémentations. Une tâche dont le codelet renvoie vers deux implémentation, une en CUDA à destination d'un GPU et une en C à destination d'un CPU par exemple, peut donc éventuellement être exécutée sur un CPU ou sur un GPU sans que cela ne soit précisé lors de sa soumission.

L'opération de soumission en question consiste en un appel à la fonction `starpu_task_insert()`, en précisant en argument le codelet et les *handles* vers les données utilisées par la tâche (cf. code 1).

```c
struct starpu_codelet T1_cl =
{
    .cpu_funcs = {task1_CPU_func},
    .cuda_funcs = {task1_CUDA_func},
    .nbuffers = 1,
    .modes = {STARPU_R}
};

struct starpu_codelet T2_cl =
{
    .cpu_funcs = {task2_CPU_func};
    .cuda_funcs = {task2_CUDA_func};
    .nbuffers = 2;
    .modes = {STARPU_RW, STARPU_W};
};

struct starpu_codelet T3_cl = {...};

struct starpu_codelet T4_cl = {...};

int main() {
    starpu_init(NULL);

    [...] /* Initialisation des données et enregistrement des handles */

    /* Soumission des tâches */
    starpu_task_insert(&T1_cl, STARPU_RW, handleA, STARPU_W, handleB, 0); //1
    starpu_task_insert(&T2_cl, STARPU_RW, handleB, 0);                    //2
    starpu_task_insert(&T2_cl, STARPU_RW, handleA, 0);                    //3
    starpu_task_insert(&T3_cl, STARPU_R, handleA, STARPU_W, handleB, 0);  //4
    starpu_task_insert(&T4_cl, STARPU_R, handleA, 0);                     //5

    /* Attente des tâches soumises et libération des handles */
    starpu_task_wait_for_all();
    starpu_data_unregister(handleA);
    starpu_data_unregister(handleB);

    starpu_shutdown();
    return 0;
}
```

Code 1 – Exemple de codelets et de soumission de tâches

Il est également possible de passer en argument des pointeurs vers des constantes, une fonction de *callback*, etc... Dans StarPU, la soumission des tâches doit être aussi asynchrone que possible, afin de donner au *runtime* un maximum de possibilités d'optimisation lors de leur exécution. Il est néanmoins possible d'insérer des barrières, par exemple avec `starpu_task_wait_for_all()` qui va attendre la terminaison de toutes les tâches soumises. Il existe également un mécanisme permettant de retarder la terminaison d'une tâche jusqu'à ce que certain points de notre choix aient été atteint au sein d'autres tâches.

La figure 1 présente le graphe de tâche obtenu en exécutant le code 1. L'ordre dans lequel les tâches sont soumises ayant une influence directe sur l'ordre dans lequel elles apparaissent dans le DAG, la tâche 1 est en tête du graphe et sera la première à être exécutée. Les tâches 2 et 3 travaillant sur des données différentes, elles peuvent être exécutées en parallèle après la terminaison de la tâche 1 qui utilisait leurs données respectives. Pour les même raisons d'accès aux données, la tâche 4 doit attendre les deux précédentes (2 et 3), mais son accès à la donnée A se faisant en lecture seule, elle peut être exécutée en parallèle de la tâche 5 qui utilise également A en lecture seule.

StarPU utilise les *handles* et les dépendances entre tâches afin de maintenir par défaut une cohérence

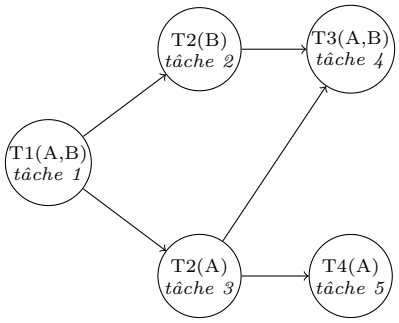

FIGURE 1 – Graphe de tâche correspondant au code 1

```
struct starpu_data_filter f =
{
    .filter_func = partition_func,
    .nchildren = 3
};

starpu_partition(handle, &f);
[...] /* Travail sur les sous-handles */
starpu_data_unpartition(handle, ...);
```

CODE 2 – Partitionnement/départitionnement de `handle`

séquentielle sur les données utilisées, cette contrainte pouvant se révéler trop forte dans certains cas, elle peut être relâchée sur certaines données.

4.2 Le partitionnement

Le partitionnement est une opération qui, à partir du handle initial H d'une donnée D, va permettre d'obtenir un certain nombre de « sous-handles » (H_i) dans lesquels sont répartis les informations de D, ces handles pouvant être utilisés par des tâches ou être partitionné à leur tour. Cette opération est réalisée par le biais de filtres donnant le nombre de sous-handles et la manière dont ils se partagent les informations de D (cf. code 2), des pointeurs vers les sous-handles étant ajoutés à H et accédés grâce à une fonction de l'API. Quand les sous-handles ne sont plus utiles, le partitionnement est « annulé » via une opération de départitionnement (cf. code 2).

Cependant, dans cette méthode, les appels à `starpu_partition` et `starpu_data_unpartition` sont synchrones, ce qui signifie qu'ils attendront la fin des tâches utilisant les données concernées, formant des barrières implicites qui peuvent nuire aux performances de l'application. Pour contrer cela, StarPU permet de partitionner/départitionner de manière asynchrone. Pour cela, un « plan » de partitionnement est soumis, initialisant les sous-handles de la partition sans la réaliser. Les partitionnement/départitionnement sont ensuite soumis avec le reste des tâches de l'application et seront pris en compte dans l'ordonnancement au *runtime*.

Ces deux méthodes sont également capables de créer plusieurs niveaux de partitionnement en redécoupant un handle issu d'un précédent partitionnement comme représenté dans la table 1 où la donnée possède deux niveaux de partitionnement L_1 et L_2. Dans cette situation plusieurs *handles* contenant différentes fractions de D coexistent. Afin de ne pas créer de conflit, il est interdit de soumettre des tâches avec un *handle* partitionné, par exemple si l'on souhaite soumettre une tâche travaillant sur D_2 il faudra d'abord s'assurer que D_{21} et D_{22} ne sont plus utilisables en départitionnant D_2.

```
struct starpu_data_filter f =
{
    .filter_func = partition_func,
    .nchildren = n
};

starpu_data_handle_t *subhandles = malloc(n*sizeof(starpu_data_handle_t));
starpu_data_partition_plan(handle, &f, subhandles);
[...] /* Travail sur handle */
starpu_data_partition_submit(handle, n, subhandles);
[...] /* Travail sur les handles de subhandles */
starpu_data_unpartition_submit(handle, n, subhandles, ...);
```

Code 3 – Partitionnement/départitionnement asynchrone de `handle`

L_0	D			
L_1	D_1		D_2	
L_2	D_{11}	D_{12}	D_{21}	D_{22}

Table 1 – Différents niveaux de partitionnement pour une donnée D

4.3 Les tâches hiérarchiques

Les tâches hiérarchiques, ou *bulles*, sont un ajout relativement récent à StarPU qui résulte du besoin des programmeurs de mieux contrôler la soumission des tâches. La soumission de ces tâches hiérarchiques produit un « graphe de contrôle » similaire au graphe de tâche mentionné plus haut. Chacune de ces tâches hiérarchiques soumettra lors de son exécution un sous-DAG, l'exécution du graphe de contrôle au complet soumettant l'ensemble des tâches constituant le graphe de tâche final de l'application.

La soumission d'une tâche hiérarchique garde une syntaxe similaire à celle de la soumission d'une tâche normale mais comporte quelques arguments supplémentaires. Le code 4 illustre cette soumission : `is_bubble` est une fonction renvoyant un booléen (avec `is_b_arg` en argument). Si ce booléen est `VRAI` alors la tâche soumise est hiérarchique et c'est la fonction `bubble_func` qui sera exécutée (avec `b_arg` en arguement), sinon cette fonction est ignorée et une tâche standard est soumise, dont les implémentations sont décrites dans `task_codelet`.

Les dépendances entre les tâches hiérarchiques sont construites et appliquées de la même manière que celles entre les tâches standards comme décrit dans la sous-section 4.1 : à partir de leur ordre de soumission et des données et leur mode d'accès. Une tâche hiérarchique ayant vocation à soumettre d'autres tâches, toutes les données utilisées par les tâches filles doivent être précisées à la soumission de la tâche hiérarchique mère, cela permettant d'assurer la correction du graphe de contrôle. Une dépendance entre deux tâches hiérarchiques correspond, dans le graphe de tâche final, à une dépendance entre les deux sous-graphes soumis par ces tâches.

Enfin l'utilisation des tâches hiérarchiques est soumise à plusieurs règles afin de garantir la correction

```
starpu_task_insert(&task_codelet,
                   STARPU_RW, handle,
                   STARPU_BUBBLE_FUNC, is_bubble,
                   STARPU_BUBBLE_FUNC_ARG, is_b_arg,
                   STARPU_BUBBLE_FUNC_GEN_DAG, bubble_func,
                   STARPU_BUBBLE_FUNC_GEN_DAG_ARG, b_arg, 0);
```

Code 4 – Exemple de soumission de tâche hiérarchique

de l'application :

1. Toute tâche hiérarchique doit soumettre des tâches (éventuellement elles-même hiérarchiques),
2. Il est possible de partitionner une donnée au sein d'une tâche hiérarchique, mais pas de créer plusieurs niveaux de partitionnement, par exemple dans la table 1, une tâche hiérarchique à le droit de partitionner de L_0 à L_1, mais pas d'enchaîner sur un partitionnement de L_1 à L_2,
3. Si une tâche reçoit une donnée partitionnée, elle ne peut soumettre de tâches que sur les enfants de cette donnée.

5 Utilisation des tâches hiérarchiques

5.1 État de l'art

Différentes études ont déjà été publiées suite à l'intérêt pour une génération plus dynamique de graphes de tâche, offrant des approches différant de celle des tâches hiérarchiques de StarPU.

5.1.1 Tâches imbriquées dans OpenMP

Perez et al. s'intéressent au modèle de tâche de OpenMP, et en particulier aux tâches imbriquées, qui permettent de créer des tâches utilisant elles-même des tâches. Dans [3], ils critiquent la rigidité du modèle qui oblige souvent à recourir à un `taskwait` (une directive qui oblige une tâche à attendre la fin de toutes ses sous-tâches) afin d'assurer la correction du programme au prix d'une potentielle sous-exploitation du parallélisme, car chaque tâche T ayant des sous-tâches devra attendre l'ensemble de leurs terminaisons, ce qui retient les tâches dépendant de T, même si elles n'ont aucune dépendance avec la dernière sous-tâche de T à terminer.

Afin de remédier à ce problème, ils proposent de relâcher, dans certains cas, la contrainte du `taskwait` et de la remplacer par une contrainte plus faible qui maintient la correction du code et la programmabilité en général tout en améliorant l'exploitation du parallélisme. Ils y parviennent en exploitant autant que possibles les dépendances exprimées entre les tâches de différents niveaux d'imbrication, et ajoutant de nouvelles directives. Là où les dépendances d'un groupe de sous-tâches étaient uniquement employée au sein de la tâche les englobant, elles peuvent maintenant être prises en compte en dehors de celle-ci, menant potentiellement à plus de parallélisme.

Les résultats qu'ils présentent montrent un meilleur passage à l'échelle lorsqu'on augmente le nombre de cœurs, et de meilleures performances en général, en particulier pour de petites tâches.

Contrairement aux tâches hiérarchiques de StarPU, cette approche ne s'applique cependant pas aux accélérateurs ce qui limite son intérêt dans le cas des architectures hétérogènes où l'on souhaiterait pouvoir disposer de tâches à gros grain pour les GPU et à grain fin pour les CPU.

5.1.2 DAG hiérarchiques avec PaRSEC

Cet objectif est celui visé par Wu et al. dans [4], en modifiant le support d'exécution du framework PaRSEC ils souhaitent dépasser la méthode employée jusqu'à présent pour fournir aux CPU et aux GPU des tâches d'une granularité appropriée. Celle-ci consistait simplement à faire un compromis entre les granularités idéales pour CPU et GPU, les premiers recevant des tâches au grain un peu trop gros et les seconds des tâches au grain un peu trop fin, cette solution permettant d'obtenir les meilleures performances, bien qu'aucune des ressources ne soit exploitées au maximum.

Ils proposent donc de partir du graphe de tâche à gros grain, puis lorsqu'une d'une tâche est récupérée dans une des files d'ordonnancement, de décider, en fonction de la ressource de calcul qui devra l'exécuter et de la granularité de la tâche, si la tâche doit être éclatée en un DAG de plus fine granularité ou non. Si la tâche est destinée à un GPU, ou si sa granularité est déjà assez fine pour un CPU, elle est exécutée, sinon un DAG à fine granularité est créé.

```c
struct starpu_codelet first_value_codelet = {
   .cpu_funcs = {print_first_value},
   .nbuffers = 1,
   .modes = {STARPU_RW},
};
struct starpu_codelet middle_value_codelet = {...}
struct starpu_codelet last_value_codelet = {...};

int is_bubble(struct starpu_task *t, void *arg) {return 1;}

void generate_dag(struct starpu_task *t, void *arg) {
    starpu_data_handle_t *v_handle = (starpu_data_handle_t *)arg;
    starpu_task_insert(&last_value_codelet, STARPU_RW, v_handle, 0);
    /* Seconde tâche hiérarchique */
    starpu_task_insert(&first_value_codelet,
                       STARPU_RW, v_handle[1],
                       STARPU_BUBBLE_FUNC, &is_bubble,
                       STARPU_BUBBLE_FUNC_GEN_DAG, &generate_dag_bis,
                       STARPU_BUBBLE_FUNC_GEN_DAG_ARG, value_handles, 0);
}

void generate_dag_bis(struct starpu_task *t, void *arg) {
    starpu_data_handle_t *v_handle = (starpu_data_handle_t *)arg;
    starpu_task_insert(&first_value_codelet, STARPU_RW, v_handle, 0);
    starpu_task_insert(&middle_value_codelet, STARPU_RW, v_handle, 0);
}

int main(int argc, char *argv[]) {
    int v[3]={1, 2, 3};
    starpu_data_handle_t v_handle;
    starpu_vector_data_register(&v_handle, v, ...);
    /* Premi‘ere tâche hiérarchique */
    starpu_task_insert(&last_value_codelet,
                       STARPU_RW, v_handle,
                       STARPU_BUBBLE_FUNC, &is_bubble,
                       STARPU_BUBBLE_FUNC_GEN_DAG, &generate_dag,
                       STARPU_BUBBLE_FUNC_GEN_DAG_ARG, v_handle, 0);
    starpu_data_unregister(v_handle);
    return 0;
}
```

Code 5 – Première utilisation de tâche hiérarchique

Ceci est rendu possible grâce à l'ajout d'une nouvelle opération générant les sous-DAG sans créer de conflit avec le graphe initial tout les laissant manipulables par le même ordonnanceur. Les *streams* CUDA sont également utilisés afin notamment de superposer calculs et communications dans le GPU.

Les tâches hiérarchiques de StarPU offrent une réponse similaire à ce problème, créant des sous-graphe de plus fine granularité mais permettent une plus grande généralité dans leurs applications.

5.2 Un cas simple sans récursion

Afin d'introduire l'utilisation des tâches hiérarchiques et l'outil permettant de visualiser les DAG créés, nous allons, dans un premier temps, partir du code 5. Toutes les tâches utilisent ici la même donnée : un tableau d'entier à trois éléments, chacun des trois codelets possédant une fonction permettant d'afficher l'un des éléments du tableau. La fonction `is_bubble` renvoyant systématiquement `VRAI`, toutes les tâches pouvant être hiérarchiques le seront, et les fonctions associées à leurs codelets seront ignorées.

L'une des fonctionnalités de StarPU est la génération, à l'aide de la bibliothèque FxT [7], de traces permettant de visualiser un ensemble d'informations concernant l'exécution d'un programme, telles que

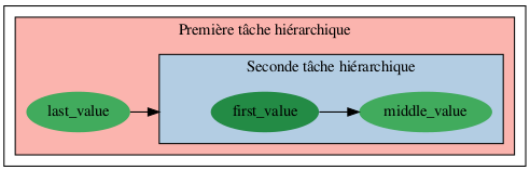

FIGURE 2 – Visualisation du DAG créé par le code 5

la répartition et durée d'exécution des tâches sur les ressources ou leurs dépendances dans le graphe de tâche. Ce dernier outil fut particulièrement utile au cours de mon stage puisqu'il permet de vérifier que le graphe généré par StarPU est bien celui que l'on essayait d'exprimer. La figure 2 illustre ainsi le graphe produit par le code 5 : les tâches hiérarchiques sont représentées par des cadres entourant le sous-graphe qu'elles ont généré, tandis que les tâches « ordinaires » prennent la forme d'ellipses.

La première tâche hiérarchique, soumise dans le `main()`, commence par soumettre une tâche régulière, puis une seconde tâche hiérarchique. Ces deux tâches ayant reçue la même donnée en lecture-écriture, une dépendance les lie et la tâche hiérarchique doit attendre la terminaison de la tâche régulière pour soumettre son DAG. Le DAG en question consiste simplement en deux autres tâches régulières, le graphe de tâche final étant donc une chaîne (last_value) → (first_value) → (middle_value) produisant la sortie `3, 1, 2`.

5.3 Tâches hiérarchiques et récursion

Afin de pouvoir remplir leur rôle correctement, les tâches hiérarchiques doivent pouvoir être utilisées récursivement. En effet une tâche T dont la granularité est jugée trop grosse doit être « éclatée » en plusieurs tâches (T_i) de plus faible granularité et formant un DAG dont l'exécution renvoie le même résultat que l'exécution de T. Si la granularité de certaines de ces sous-tâches est encore considérée trop grande, ces dernières doivent pouvoir se comporter en tant que tâches hiérarchiques et soumettre à leur tour un nouveau graphe de tâche à la granularité encore plus faible. Cette section est consacrée au travail réalisé durant le stage afin d'établir une méthode permettant d'utiliser le partitionnement et les tâches hiérarchiques récursivement.

5.3.1 Partitionner récursivement

Afin de pouvoir faire décroître la granularité des tâches d'une application, il est possible de diminuer la taille des données traitée par chaque tâche. Par exemple, pour une application ayant pour but de sommer deux vecteurs de taille N, une unique tâche pourra s'en charger en N opérations, mais il est possible d'en réduire la granularité en divisant ces vecteurs en deux et en soumettant deux tâches qui effectueront chacune $\frac{N}{2}$ sommes. L'idée de récursion est introduite dans le cas où ce nombre d'opération est toujours considéré trop élevé, afin de permettre à l'application de continuer à partitionner les vecteurs jusqu'à atteindre un nombre d'opération par tâche qui lui permet d'utiliser ses ressources de calcul de manière optimale.

L'exemple choisit ci-dessus est suffisamment simple pour pouvoir être réalisé sans tâches hiérarchiques. Il suffit en effet de créer deux types de tâches : une première T_{sum} qui somme les vecteurs qu'elle reçoit, et une seconde T_{rec} qui partitionne les vecteurs qu'elle reçoit, puis, en fonction de la taille des sous-vecteurs créés, décide de soumettre T_{sum} ou de soumettre de nouvelles T_{rec} avec les sous-vecteurs. Cela produirait un graphe de tâche ressemblant à la figure 3.

Le même type de graphe peut évidement être généré avec des tâches hiérarchiques en remplacant T_{sum} et T_{rec} par une tâche hiérarchique suivant le modèle du code 6. `sum_codelet` est le codelet anciennement associé à la tâche T_{sum}, tandis que la fonction `rec_func` partitionne les données et soumet de nouvelles tâches du même modèle sur les sous-données. La fonction `is_bubble` définit la condition d'arrêt en permettant de décider, en fonction de la taille de la donnée, si la tâche est

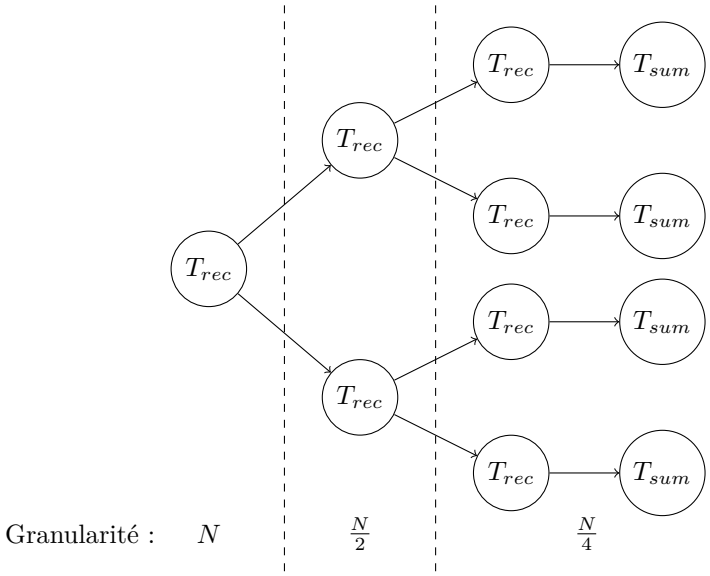

FIGURE 3 – Graphe de tâche d'une somme de vecteurs utilisant le partitionnement récursif

```
starpu_task_insert(&sum_codelet,
                   STARPU_R, v1_handle,
                   STARPU_RW, v2_handle,
                   STARPU_BUBBLE_FUNC, is_bubble,
                   STARPU_BUBBLE_FUNC_GEN_DAG, rec_func, 0);
```

CODE 6 – Tâche hiérarchique remplaçant T_{sum} et T_{rec}

hiérarchique ou non, c'est à dire si la fonction `rec_func` doit être exécutée (pour poursuivre la récursion) ou être ignorée au profit du codelet fourni (ce qui interrompt la récursion).

Écrire des codes réalisant ce genre d'opération n'a posé aucun problème, que ce soit avec ou sans tâches hiérarchiques. Néanmoins le partitionnement seul n'est pas suffisant car, comme précisé dans la sous-section 4.2, un *handle* partitionné ne peut pas être utilisé dans une tâche, il est donc crucial de pouvoir départitionner les données si l'on veut pouvoir continuer à les utiliser dans des tâches à grosse granularité.

5.3.2 Départitionner récursivement

L'objectif est donc maintenant de pouvoir écrire un code partitionnant récursivement une donnée puis la départitionnant correctement, de sorte à la rendre à nouveau utilisable telle quelle par des tâches de calculs. On choisit le cas d'une application partitionnant récursivement un vecteur et soumettant une tâche de calcul à chaque niveau de partitionnement avant de départitionner. Cette opération correspond à un graphe de tâche « en diamant » comme illustré dans la figure 4, les L_i représentant le niveau de partitionnement de la donnée dans chaque tâche. Contrairement à la sous-section précédente, à partir du moment où la donnée possède plus d'un niveau de partitionnement il n'est plus possible de se contenter des tâches ordinaires car elles ne peuvent pas exprimer ce type de graphe. En effet l'opération de départitionnement d'un handle H nécessite d'avoir accès à H et à ses sous-handles, ce qui oblige à utiliser la méthode décrite dans le code 7.

Soumettre une tâche de cette forme permet de générer correctement la première moitié du graphe de tâche, mais pas la seconde, produisant le graphe présenté dans la figure 5. Bien sûr un tel graphe ne peut pas être exécuté convenablement et abouti à une erreur de segmentation ou au déclenchement d'un

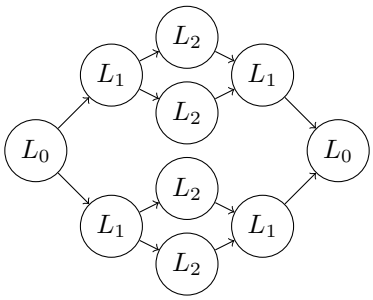

FIGURE 4 – Forme simplifiée du graphe de tâche attendu

```c
void recursive_partition_task() {
    starpu_data_handle_t handle, *subhandles;
    /* Partitionnement de handle */
    if (!stop_cond) {
        /* Soumission de recursive_partition_task sur les sous-handles */
    }
    /* Soumission de taches de calcul sur les sous-handles */
    /* Departitionnement de handle */
}
```

CODE 7 – Tâche principale d'un code tentant d'obtenir un graphe en diamant sans tâche hiérarchique

`assert`, car on soumet des tâches sur des handles partitionnés. La raison de cette erreur est la cohérence séquentielle : quand la tâche de niveau L_0 soumet le départitionnement de H, les tâches de niveau L_1 partitionnant les sous-handles de H n'ont pas encore été soumises et la tâche de départitionnement ne possède donc aucune dépendance vers les tâches L_1. Le départitionnement a donc lieu trop tôt.

L'usage des tâches hiérarchiques est donc obligatoire pour générer ce type de graphe. Cependant simplement transformer le code 7 en ajoutant l'utilisation de tâche hiérarchiques ne suffit pas, car là encore, les dépendances de la tâche de départitionnement ne seront pas correctes. Pour atteindre notre objectif nous avons dû effectuer quelques ajustements.

Tout d'abord, lorsque l'on utilise le partitionnement asynchrone de StarPU, le *runtime* va par défaut utiliser les plans de partitionnement indiqués par l'utilisateur pour effectuer automatiquement les partitionnement/départitionnement. Cela permet au programmeur ayant précisé un plan de soumettre des tâches sur le handle et ses sous-handles sans devoir partitionner ou départitionner lui même. Néanmoins dans notre situation, nous avons désactivé cette fonctionnalité afin de garder un contrôle

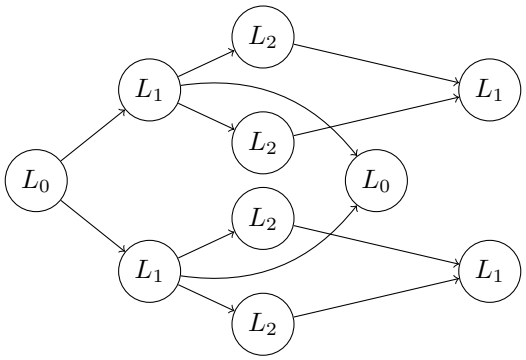

FIGURE 5 – Le graphe incorrect généré par la première tentative

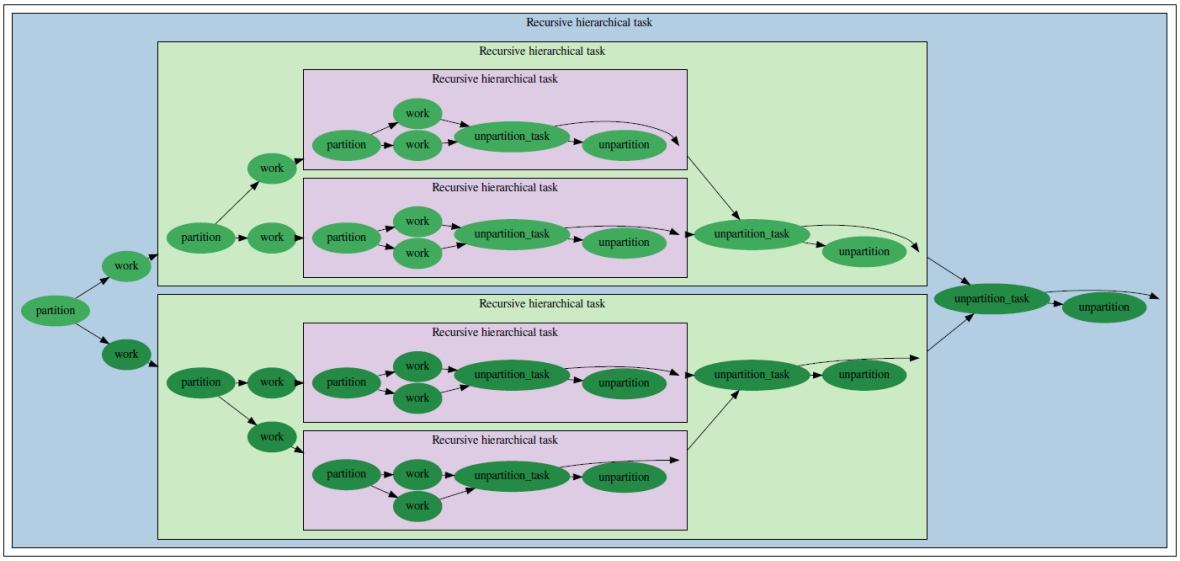

FIGURE 6 – Le graphe conforme à nos attentes généré avec des tâches hiérarchiques

maximal sur l'ordonnancement des opérations de partitionnement.

Deux fonctions, que nous appelerons `starpu_get_children` et `starpu_get_nb_children`, ont également été ajoutées à StarPU permettant respectivement d'accéder aux pointeurs des sous-handles et à leur nombre à partir des informations stockées dans le handle père. Cela permet de simplifier considérablement l'écriture de certaines tâches en évitant au programmeur de fournir l'ensemble des pointeurs des sous-handles, mais surtout de pouvoir récupérer ces informations dans des tâches où elles auraient été auparavant inaccessibles.

Enfin, il a fallu avoir recours à un mécanisme de barrière, spécifique aux tâches hiérarchiques, afin d'assurer le correct ordonnancement des tâches. Le mécanisme en question consiste en une sorte de compteur, propre à chaque tâche, qui empêche leur terminaison tant qu'il n'est pas nul. Initialisé à zéro par défaut, ce compteur peut être incrémenté et décrémenté par l'utilisateur pour lui donner plus de contrôle sur les tâches qu'il définit. Par exemple une tâche hiérarchique T_H générant un DAG formé d'une seule tâche T terminera après avoir soumis la tâche en question, ce qui lancera l'exécution des tâches qui dépendaient de T_H avant la fin de l'exécution de T. Mais en utilisant ce mécanisme de barrière, il est possible de mettre le compteur de T_H à 1 et d'ajouter à T un `callback` (une fonction qui sera exécutée à la fin de T) le décrémentant. De cette façon, T_H ne terminera pas après avoir soumis T et les tâches attendant la fin de T_H pour être exécutées devront également attendre la fin de T.

La figure 6 présente le graphe de tâche obtenu, et la figure 7 illustre le graphe de tâche « final », après l'exécution de toutes les tâches hiérarchiques. Les flèches partant de `unpartition_task` et ne pointant vers « aucune » tâche dans la figure 6 illustrent en réalité les barrières précédemment décrites. La génération de ce graphe peut être résumée par la tâche hiérarchique présentée dans le code 8. Comme précédemment, la condition d'arrêt est interne à la fonction `is_bubble`.

Cette méthode permet donc de partitionner un handle récursivement tout en soumettant des tâches de différentes granularités, puis de le départitionner correctement pour pouvoir continuer de l'utiliser. Différents codes ont été ajouté à ce premier exemple, afin de vérifier que cette méthode est aussi expressive que possible. Par exemple la figure 8 est le graphe de tâche d'une application effectuant le tri fusion d'un tableau d'entier. Le graphe reprend la forme du graphe précédant, mais les tâches de calculs `merge_task` sont désormais placées entre les départitionnement, et la correction de l'ordonnancement peut facilement être vérifiée en s'assurant que le tableau est bel et bien trié.

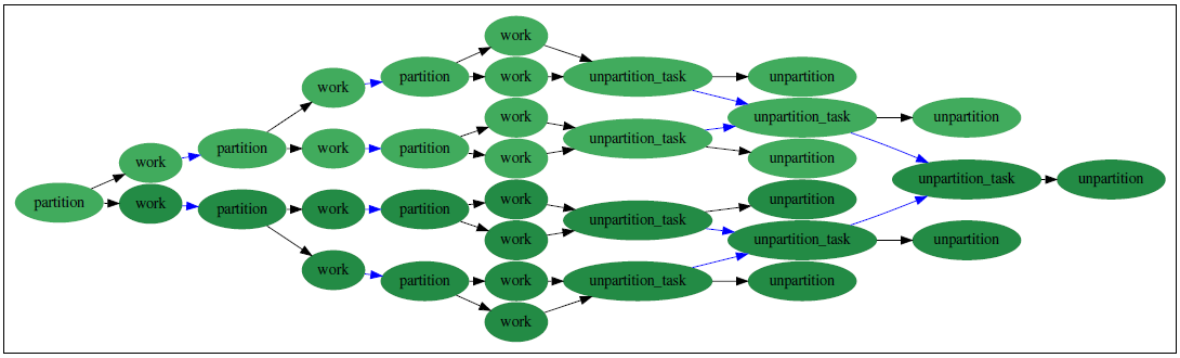

FIGURE 7 – Dépendances entre les tâches « ordinaires » de la figure 6

```c
void rec_hierarch_task(starpu_task *t, void *arg) {
    starpu_data_handle_t handle = *(starpu_data_handle *)arg;
    if (starpu_get_nb_children(handle) == 0) {
        starpu_data_handle_t *subhandles = malloc(...);
        /* Desactivation du partitionnement automatique sur subhandles */
        /* Partitionnement de handle */
    }
    for (sh in subhandles) {
        starpu_task_insert(&work_codelet, STARPU_RW, sh, 0); /* work */
        starpu_task_insert(&null_codelet, STARPU_RW, sh,
                           STARPU_BUBBLE_FUNC, is_bubble,
                           STARPU_BUBBLE_FUNC_GEN_DAG, rec_hierarch_task, 0);
    }
    /* Incrementer compteur */
    starpu_task_insert(&unpartition_codelet, ...,
                       STARPU_CALLBACK, /* Decrementer compteur */, 0);
}
```

CODE 8 – Tâche hiérarchique récursive produisant le graphe de la figure 6

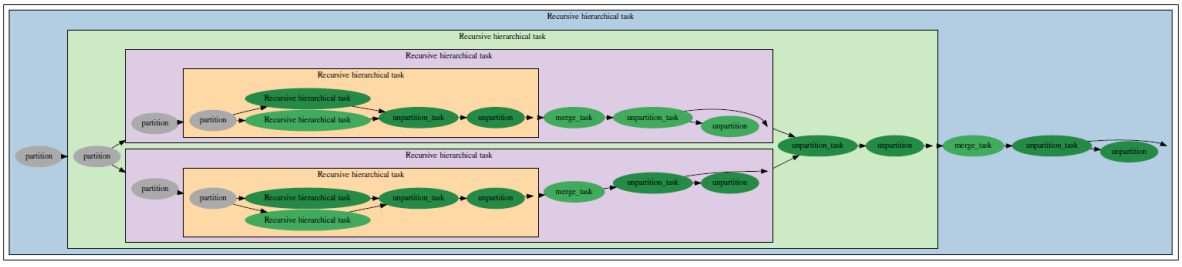

FIGURE 8 – Graphe de tâche d'un tri fusion avec tâches hiérarchiques

6 Conclusion

Au cours de ce stage, j'ai principalement travaillé sur l'utilisation récursive des tâches hiérarchiques de StarPU. Cela a nécessité, dans un premier temps, de me familiariser avec les concepts généraux de StarPU, comme l'implémentation de tâche en codelets, l'expression de leurs dépendances et le partitionnement des données, puis avec la notion de tâches hiérarchiques, qui permet d'ajouter du dynamisme à la soumission du graphe de tâche d'une application.

Le gros de mon travail a ensuite consisté a étudier le comportement des tâches hiérarchiques au sein de récursions. De nombreux problèmes sont apparus, en particulier concernant l'ordonnancement des départitionnements de données ayant été partitionnées récursivement. Des ajouts au code de StarPU et l'utilisation d'un mécanisme de barrière, contraignant une tâche à attendre l'exécution d'une partie spécifique du code qu'elle a soumis pour terminer, ont été nécessaire pour résoudre ces problèmes et obtenir un moyen fiable et systématique de réaliser ce genre d'opération.

Les stratégies d'ordonnancement présentées ici ne concernent que la correction des programmes et ne s'attaquent pas à la recherche des meilleures performances possibles. Afin de pouvoir réellement utiliser les tâches hiérarchiques pour générer des graphes de tâches ayant plusieurs granularités, il sera nécessaire d'établir des stratégies d'ordonnancement capables de décider quand créer des tâches de plus fine granularité en fonction, par exemple, des ressources de calculs disponibles à un instant t.

Bibliographie

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst et Pierre-André Wacrenier. « StarPU : a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures ». In : *Concurrency and Computations : Practice and Experience, Wiley* 23.2 (2011), p. 187–198 (↑ pp. 2, 3).

[2] *Top 500*. https://www.top500.org/ (↑ p. 2).

[3] Joseph M. Perez, Vicenç Beltran, Jesus Labarta et Eduard Ayguadé. « Improving the Integration of Task Nesting and Dependencies in OpenMP ». In : *2017 IEEE International Parallel and Distributed Processing Symposium* (mai 2017), p. 809–818 (↑ pp. 3, 7).

[4] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge et Jack Dongarra. « Hierarchical DAG Scheduling for Hybrid Distributed Systems ». In : *2015 IEEE International Parallel and Distributed Processing Symposium* (mai 2015) (↑ pp. 3, 7).

[5] *PaRSEC*. https://icl.utk.edu/parsec/index.html (↑ p. 3).

[6] *StarPU Handbook*. http://starpu.gforge.inria.fr/doc/html/ (↑ p. 3).

[7] *FxT : Fast User/Kernel Tracing*. https://savannah.nongnu.org/projects/fkt/ (↑ p. 8).

[8] *Informations sur Summit*. https://www.top500.org/system/179397 (↑ p. 16).

Annexe

A Benchmark de *Summit* avec StarPU et Chameleon

Pendant mon stage j'ai eu l'opportunité de passer deux mois au KIT (*Karlsruhe Institut of Technology*) en Allemagne. L'objectif de ces deux mois était de mesurer les performances pouvant être atteintes sur le supercalculateur *Summit* [8] en utilisant conjointement le runtime StarPU et la bibliothèque d'algèbre linéaire Chameleon.

A.1 L'architecture de *Summit*

Théoriquement capable d'atteindre les 200 PFlops en double-précision, *Summit* est actuellement la machine la plus puissante du monde. Ces performances sont le produit de plus de 4600 nœuds d'une puissance théorique individuelle d'environ 40 TFlops. Chacun de ces nœuds est composé de deux processeurs IBM POWER9 comportant chacun vingt-et-un cœurs et de six accélérateurs NVIDIA Volta V100, comme illustré par la figure 9. La majorité de la puissance d'un nœud repose sur les GPU, chacun étant capable d'effectuer approximativement 7 TFlops, là où les deux CPU n'atteignent « que » 1 TFlops.

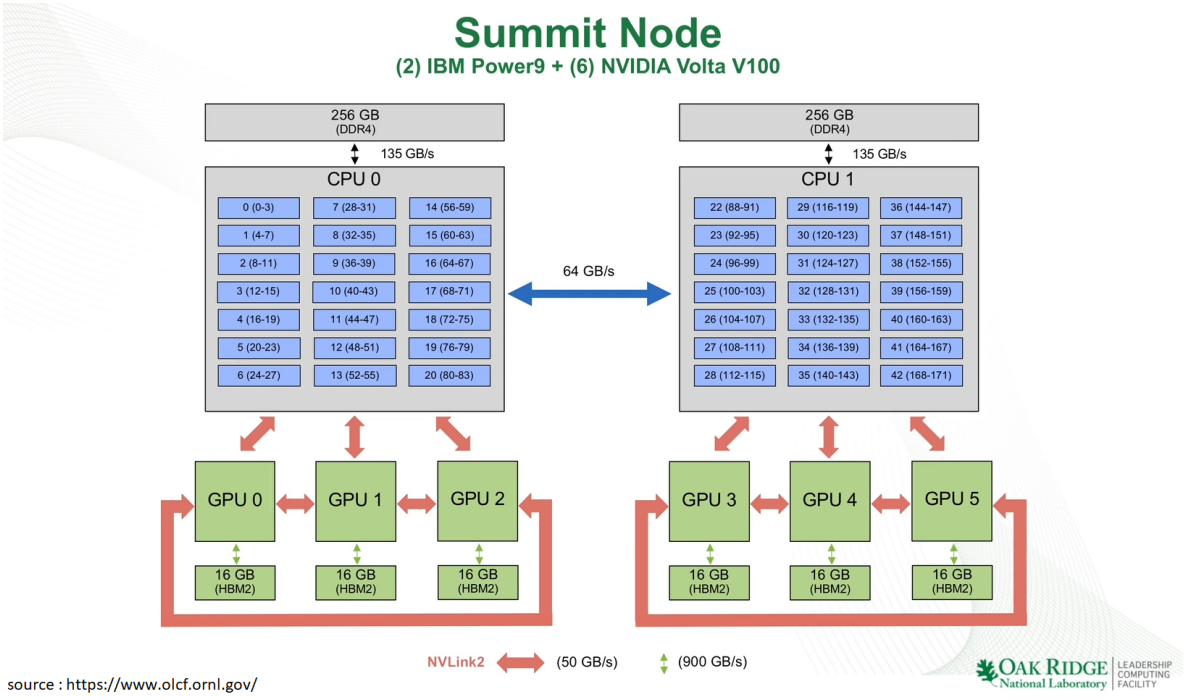

FIGURE 9 – Architecture d'un nœud de *Summit*

A.2 Utilisation de StarPU et Chameleon sur *Summit*

Utiliser StarPU et Chameleon sur *Summit* ne fut pas chose aisée. Les multiples versions des modules `Netlib-Lapack` et `gcc` présentes sur *Summit* n'étant pas compatibles entre elles, de nombreux problèmes d'édition de liens ont initialement gêné l'utilisation de Chameleon. Une fois ces problèmes réglés et l'installation des deux programmes complétée, il est rapidement apparu que les performances obtenues n'était pas du tout satisfaisantes. Les performances sur une factorisation de Cholesky par bloc ne dépassait pas les 30 GFlops quelque soit le nombre de CPU ou de GPU mis à contribution, soit moins de 0.5% de la puissance d'un seul GPU. Essayer d'autres stratégies d'ordonnancement de StarPU

ne produisant aucun résultat, j'ai utilisé les traces générées par StarPU pour comprendre la cause de ces faibles performances. La figure 10 est un diagrame de Gantt présentant la durée d'exécution et la répartition des différents noyaux utilisés dans la factorisation, chaque couleur étant associée à un type de noyau d'algèbre linéaire nécessaire dans la factorisation de Cholesky par bloc, avec en jaune les factorisations de Cholesky des blocs diagonaux (POTRF), en vert les résolutions de systèmes triangulaires (TRSM), en bleu les produits de matrices (GEMM), etc. Le rouge représente lui l'*idle time* des différentes ressources et doit donc idéalement être aussi peu représenté que possible. La figure 10 rend donc le problème clair : les noyaux lancés sur CPU sont infiniment plus lents que ceux lancés sur GPU, qui sont pratiquement invisibles dans la trace tant la différence de vitesse est grande. Les POTRF, qui ne possèdent pas d'implémentation CUDA pour GPU, devant obligatoirement être lancés sur un CPU, le meilleur ordonnancement possible est celui où les seuls noyaux exécutés sur CPU sont les POTRF, le TRSM qui s'était glissé sur CPU dans la figure 10 ayant eu un impact désastreux sur les performances. Malheureusement même dans ce cas de figure idéal, les performances obtenues ne dépassent pas les 300 GFlops, ce qui reste déplorable face aux capacités d'un nœud *Summit*.

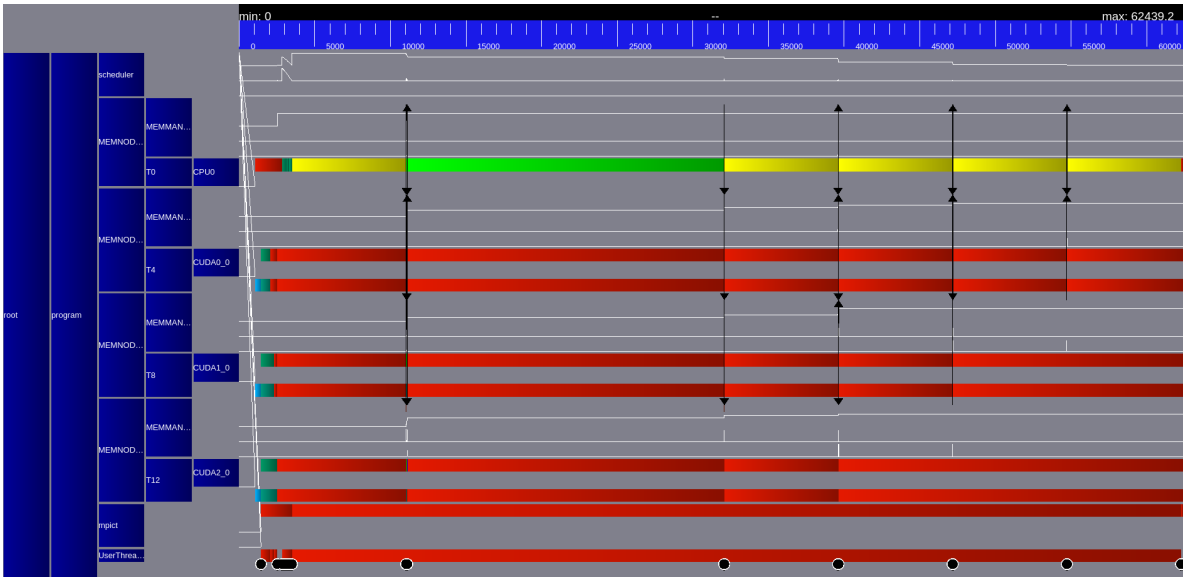

FIGURE 10 – Trace de l'exécution d'une factorisation de Cholesky avec l'installation initiale

Afin d'améliorer les performances, différentes approches ont été essayées. La première fut d'utiliser une ancienne version de Chameleon possédant des implémentations CUDA du POTRF afin de pouvoir se passer des CPU qui ne font que ralentir le programme. Cette tentative n'a cependant pas permis d'aboutir à un code fonctionnel et a donc été abandonnée.

Il fut également tenté de réduire drastiquement la taille de bloc utilisée dans la factorisation afin de fournir aux CPU des tâches ayant une granularité optimale pour leurs capacités. Afin de ne pas trop pénaliser les GPU, nous utilisons pour chacun d'entre eux plusieurs *CUDA Stream*, une solution permettant d'avoir plusieurs fils d'exécutions sur un même GPU, ce qui permettrait d'exploiter correctement les GPU malgré les petites tâches. Les résultats de cette méthode dépassèrent largement ceux obtenus précédemment, atteignant jusqu'à 1 TFlops, mais restèrent largement en dessous de ce qu'on pourrait espérer de *Summit*. Par ailleurs l'utilisation des *CUDA Stream* n'eût pas l'effet escompté, car les performances atteintes sans eux étaient équivalentes voire supérieures. La cause fut identifiée comme un problème de *binding*, les *CUDA Stream* d'un GPU étant tous liés à un unique cœur CPU au lieu d'être liés à des cœurs différents, les rendant sans intérêt.

Néanmoins afin de pouvoir régler ce problème, une solution plus intéressante a été atteinte. Il est en effet apparu que remplacer le module `Netlib-Lapack` par `ESSL`, une bibliothèque d'IBM implémentant entre autre BLAS et LAPACK, résultait en de très bonnes performances. Cela nécessite de légèrement

modifier le code de Chameleon pour utiliser les noms des fonctions d'`ESSL`. On peut voir dans la figure 11 que les noyaux sur CPU, bien que toujours disproportionnés par rapport à leurs homologues sur GPU, représentent maintenant une part plus raisonnable du temps d'exécution total. Je n'ai pas pu expliquer les différences de performances incroyables entre les implémentations de `Netlib-Lapack` et celles de `ESSL`, mais il est possible qu'elles soient due à un problème de `Netlib-Lapack` ou au fait que `ESSL` étant développé par IBM, ses performances soient optimales sur les POWER9. C'est donc cette méthode qui est utilisée pour produire les résultats présentés dans A.3.

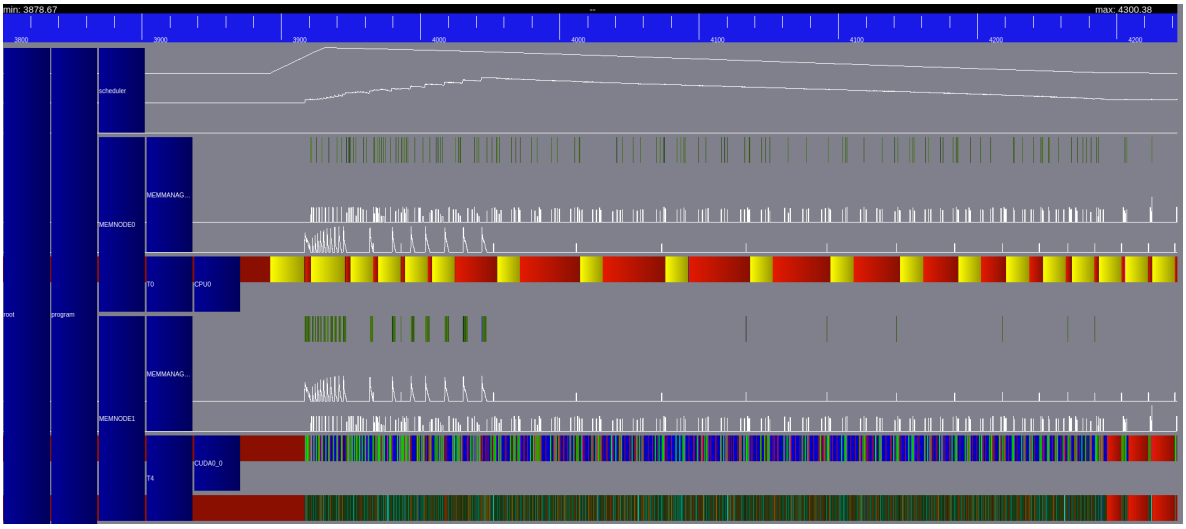

FIGURE 11 – Trace d'une factorisation de Cholesky avec ESSL

A.3 Résultats

Toutes les mesures de performances ont eu lieu sur un nœud de *Summit* et concernent une factorisation de Cholesky par bloc. La taille de bloc choisie jouant un grand rôle dans les performances de l'application, j'ai dans un premier temps cherché à déterminer la taille optimale pour obtenir les meilleures performances possibles. Dans le cas d'un seul GPU (cf. figure 12), cette taille de bloc optimale ne varie pas beaucoup en fonction de la taille de la matrice. En utilisant trois GPU (cf. figure 13), la taille de bloc optimale varie surtout pour des tailles de matrices « petites » par rapport à la puissance de calcul déployée par les trois GPU.

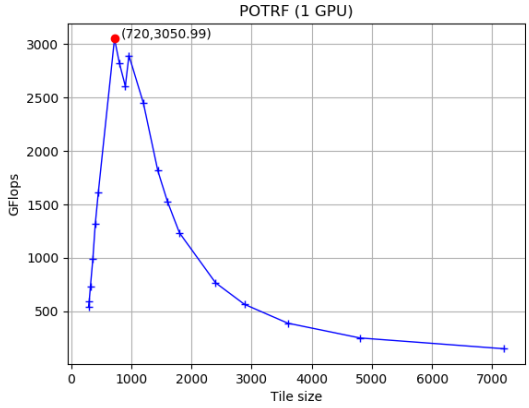

FIGURE 12 – Performances en fonction de la taille de bloc

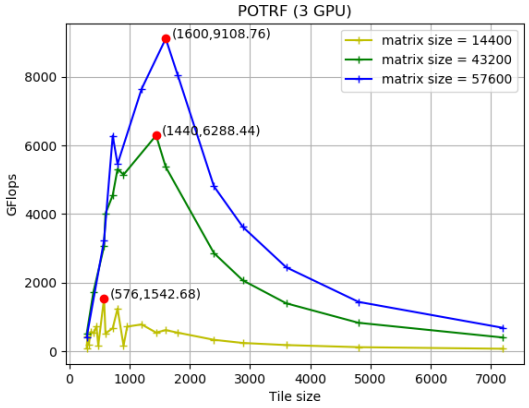

FIGURE 13 – Performances en fonction de la taille de bloc et différentes tailles de matrices

Utilisant ces tailles de blocs, on abouti aux résultats des figures 14 et 15 ci-dessous. Les lignes en pointillés indiquent les puissances théoriques des ressources utilisées. La figure 14 permet de constater qu'utiliser de multiples cœurs CPU est contre productif et réduit les performances dès que le nombre de GPU est supérieur à 1. La figure 15 illustre, elle, l'impact de la taille de bloc sur les performances, augmenter la taille de bloc de 33% après avoir atteint un palier permettant d'atteindre de meilleures performances.

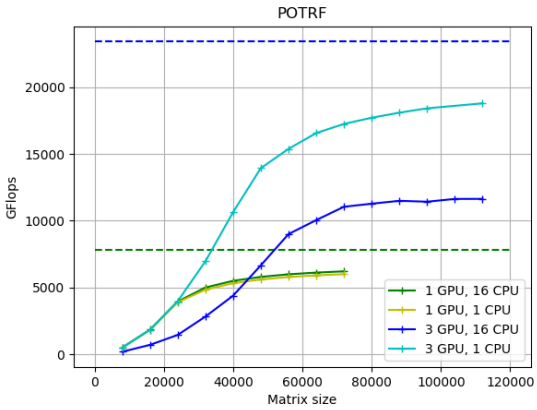

FIGURE 14 – Performances en fonction de la taille de matrices et limites théoriques des ressources utilisées

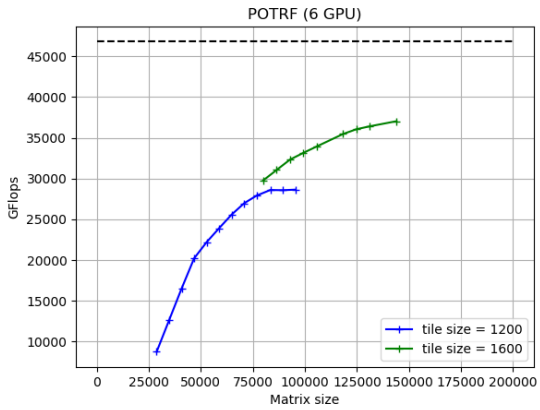

FIGURE 15 – Performances en fonction de la taille de matrices et limites théoriques des ressources utilisées sur un nœud entier

L'utilisation de StarPU et de Chameleon avec `ESSL` permet donc quasiment d'atteindre 80% de la puissance théorique d'un nœud entier de *Summit*, ce pourcentage augmentant quand le nombre de GPU diminue.