Etude de l'impact du power capping sur les performances des GPU

Albert d'Aviau de Piolant

Mémoire de stage de Master 2

Albert d'Aviau de Piolant

22 Juin 2023

Contents

1 Introduction

J'ai réalisé un stage de six mois au centre Inria de l'Université de Bordeaux dans l'équipe STORM. Inria est un institut de recherche qui s'intéresse aux thématiques de recherche liées au numérique. Le centre est en lien avec des instances universitaires comme celle de l'Université de Bordeaux, et avec des entreprises. Inria est divisé en plusieurs équipes-projets, qui accueillent des chercheurs, des ingénieurs, des doctorants ainsi que des stagiaires. L'équipe STORM (STatique Optimisation and Runtime Methods) dans laquelle j'ai travaillé, propose des méthodes et des outils pour la programmation sur architectures parallèles dans le domaine du calcul haute performance.

1.1 Calcul Haute Performance

Le calcul haute performance est une branche de l'informatique qui vise l'optimisation de programmes. Ces derniers sont parfois exécutés sur super calculateurs, et pour bénéficier au mieux de la puissance de la machine, on peut faire en sorte que le programme prenne en compte certains éléments, comme le parallélisme, les défauts de cache, les communications entre nœuds de calcul, etc... Ces techniques d'optimisation sont grandement utilisées dans la simulation scientifique, par exemple pour la météorologie, l'astrophysique, ou bien la biologie.

1.2 Problématique énergétique

Les super calculateurs sont des machines qui consomment énormément de puissance. Frontier, le superordinateur le plus puissant du monde selon le TOP500 [1], consomme au maximum 22,7 MW. C'est l'équivalent à la consommation d'un ville américaine d'environ 12000 foyers. De ce fait, la consommation d'énergie est un problème et la minimiser est un enjeu majeur que cela soit d'un point de vu écologique ou économique. Il existe plusieurs solutions pour réduire la consommation. On peut faire en sorte d'éteindre des nœuds de calcul [2] ou bien de faire du DVFS (Dynamic Voltage Frequency Scaling) [3]. Dans ce dernier cas, on vas modifier la fréquence de fonctionnement du matériel (d'un processeur par exemple), pour avoir un impact sur la puissance, l'énergie, et les performances. On peut gagner en puissance mais perdre en énergie, gagner en énergie mais perdre en performance, tout dépend de ce que l'on veut faire. Une autre option pour réduire la consommation, c'est d'utiliser le power capping.

1.3 Power capping

Le power capping (ou limitation de puissance en français) est une méthode utilisée pour limiter la puissance utilisable au niveau du composant (processeur, accélérateur de calcul, carte graphique, etc...) afin de baisser la consommation maximale d'une machine. C'est une limite de puissance que le composant ne pourra pas dépasser. Il existe plusieurs études qui utilisent le power capping pour réduire la consommation [4–6], mais celles-ci ne s'intéressent qu'à la consommation du CPU, ou bien qu'à une application statique. Nous proposons donc une nouvelle étude avec un power capping dynamique sur architectures hétérogènes. On dit qu'une machine possède une architecture hétérogène lorsque que cette dernière contient plusieurs types de processeurs en son sein. En général, cela veut dire qu'il y a des processeur (CPU) et des processeurs graphique (GPU), et que l'on exécute des programmes qui bénéficient des deux types de processeurs.

L'objectif du stage est donc de proposer une gestion dynamique du power capping sur architectures hétérogènes. C'est à dire d'optimiser la distribution de la puissance à disposition entre le CPU et le GPU, sous la contrainte d'une limite de puissance globale. Le but ici est de profiter d'un budget plus élevé (par exemple sur le GPU) pour améliorer les performances sans impact sur la consommation globale puisqu'on reste dans la limite imposée.

La suite de ce rapport se présente de la manière suivante : je vais décrire le contexte et les motivations derrière ce travail dans la première section (section 2) en présentant la situation initiale et les objectifs prévus pour le projet. Ensuite, dans la section 3, je décrirai l'algorithme proposé pour notre étude ainsi que la manière avec laquelle je l'ai implémenté. Dans la section qui suit (section 4) je décrirai les différents outils, applications et le matériel que j'ai utilisés durant mon stage. La section suivante (section 5) décrira les paramètres et la mise en place des expériences pour évaluer le travail apporté ainsi que les résultats préliminaires obtenus et je finirai par une conclusion et une description des travaux actuels et futurs (section 7).

2 Contexte et motivations

En baissant la limite de puissance, le power capping va avoir des conséquences différentes selon le profile de l'application. En effet si cette dernière est "CPU-intensive" (intense en calcul sur le CPU, et donc qui utilise surtout les unités de calcul du processeur), alors on pourra observer une baisse des performances, car les CPU font partie des composants qui demandent le plus de puissance dans une machine pour fonctionner. Une baisse de perfor-

mance liée à une limitation de la consommation de puissance peut aussi, en fonction des cas, générer une baisse ou un augmentation de la consommation en énergie.

Cette limite de puissance peut être imposée par le fournisseur du supercalculateur. Dans le cas où on travaille sur un seul nœud et un seul GPU, le power capping va nous faire perdre des performances par rapport à un usage sans limite de consommation imposée comme le montre la figure 1. Notre étude apporte une solution pour limiter la perte de performance sous la contrainte d'une limite de consommation des composants. Au lieu d'avoir une distribution statique du budget de puissance, nous proposons une distribution dynamique selon les besoins de l'application en cours d'exécution.

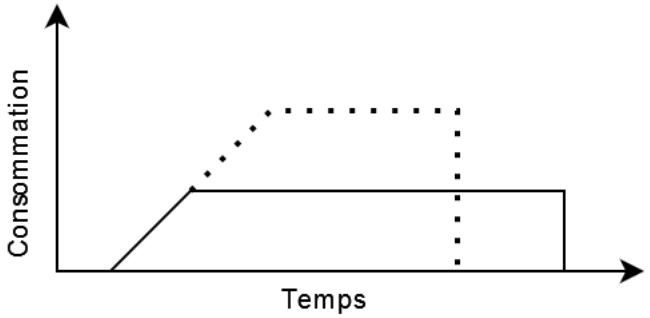
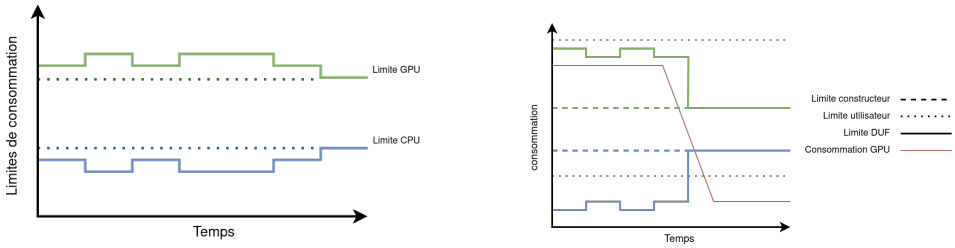


Figure 1: Consommation limitée par la puissance maximale du matériel en pointillés, et consommation limitée par un power capping imposé en ligne continue

Nous avons donc deux limites. La limite du constructeur qui n'est physiquement pas dépassable, et la limite imposée. L'idée, c'est de profiter des moments où le CPU fait moins de calcul, pour baisser sa limite de consommation, comme le montre la figure 2a. Cela va entraîner un surplus de budget, et donc nous pouvons augmenter la limite de consommation du GPU, et ainsi dépasser sa limite imposée, tout en respectant la limite imposée globale et sans impacter la consommation au niveau du nœud. Le GPU va pouvoir bénéficier de plus de puissance, et donc gagner en performance [7]. Par exemple, le CPU a une limite de consommation de 100W, et le GPU de 200W. Si un administrateur nous impose un power capping de 50%, le CPU va devoir tourner en dessous de 50W, et le GPU en dessous de 100W. Avec cette solution, si on détecte que le CPU consomme moins, on peut baisser sa limite de consommation à 40W, et donc augmenter celle du



(a) Répartition du budget de limite de consommation entre le CPU (en bleu) et le GPU (en vert)

(b) Détection de l'activité sur le GPU

Figure 2: Distribution dynamique du power capping sur CPU et GPU en considérant l'activité sur le GPU

GPU à 110W.

Cette solution nous permet de distribuer le budget de consommation en fonction des mesures prises sur le CPU, mais nous pouvons aller plus loin en regardant ce qu'il se passe sur le GPU. C'est pour cela que nous avons introduit des mesures sur ce dernier. De cette manière, si le GPU fait moins de calcul, on peu baisser sa limite de consommation pour basculer le budget sur le CPU, pour que ce dernier puisse bénéficier d'une puissance plus importante et donc gagner en performance (voir figure 2b).

2.1 DUF

Afin de mettre en œuvre cette idée, je me suis basé sur un outil existant, appelé DUF, qui gère différents paramètres pour réduire la puissance consommée. DUF (Dynamic Uncore Frequency scaling) [8] est un outil de gestion dynamique de la fréquence uncore du processeur, qui ne fonctionne que sur un seul nœud. Sur un processeur on distingue la fréquence "core" (du cœur) et la fréquence "uncore" (qui n'est pas du cœur). Cette dernière désigne la fréquence qui gère le dernier niveau de cache ainsi que le contrôleur mémoire, là où la fréquence "core" gère les unités de calcul et le cache L1 et L2. DUF opère uniquement sur le CPU, l'outil ne s'occupe pas des autres composants. DUF à pour but de limiter la consommation d'une machine ainsi que de limiter la perte de performance liée à cette baisse de consommation, en fonction d'un "slowdown" précisé par l'utilisateur.

DUF possède une extension appelée DUFP (Dynamic Uncore Frequency scaling and Power capping) [9], qui permet d'ajouter une gestion dynamique

du power capping au sein de DUF pour pousser la réduction de consommation encore plus loin. L'utilisateur peut donc choisir d'utiliser la fonctionnalité en ajoutant l'option du power capping lors de l'exécution. Dans la suite du rapport, j'écrirai DUF pour parler de DUF et de son extension DUFP.

2.1.1 Fonctionnement

L'outil fonctionne de la manière suivante: DUF est lancé en parallèle d'une ou plusieurs autres applications pour moduler la fréquence uncore ainsi que le power capping. L'utilisateur précise sur quel processeur il veut prendre des mesures. Il peut également définir un "slowdown" toléré, c'est à dire un pourcentage de perte de performance lié à la baisse de consommation qu'il considère comme acceptable. Il peut aussi choisir d'utiliser l'outil seulement avec sa capacité de mesure, sans modifier quoi que ce soit sur le matériel. Une dernière option importante est le choix d'utiliser la fonctionnalité du power capping dynamique ou non.

Toutes les x millisecondes (paramètre défini par l'utilisateur), DUF consulte les différents compteurs (FLOPS, mémoire et puissance) et récupère les données mesurées pour les enregistrer en mémoire. L'outil choisit quelle décision doit être prise pour la fréquence uncore et pour le power capping s'il est activé. Pour un processeur, DUF va faire varier la fréquence uncore jusqu'à ce que les performances atteignent le niveau de slowdown toléré par l'utilisateur. Il en est de même pour le power capping quand l'option est utilisée.

3 Principe de fonctionnement

Dans cette section, je vais présenter l'algorithme que nous proposons pour intégrer une gestion dynamique et hétérogène du power capping à DUF, ainsi que la manière dont je l'ai implémentée.

3.1 Algorithme

L'algorithme 3.1 représente la solution choisie pour apporter une gestion dynamique du power capping entre CPU et GPU au logiciel DUF. C'est une extension de ce dernier, l'algorithme est donc adapté à l'architecture de l'outil. Dans notre solution, le power capping dynamique possède deux limites maximales : la limite constructeur (REAL_CPU_MAX_PCAP) qui n'est physiquement pas dépassable, et la limite précisée par l'utilisateur

(CPU_MAX_PCAP), qui représente le power capping global que l'ont veut appliquer à l'exécution de notre application.

À l'initialisation, on récupère la consommation de base du GPU sur lequel aucune application ne s'exécute (ligne 3.1). Lorsque la mesure de l'activité sur le GPU est faite (ligne 3.1), il y a deux possibilités : soit elle est trop faible (donc égale à la consommation de base du GPU), soit elle a augmenté (il y a de l'activité sur le GPU). Dans le premier cas, on augmente le power capping du CPU jusqu'à sa limite physique, en dépassant la limite imposée par l'utilisateur, et on baisse le power capping du GPU en conséquence, c'est-à-dire du même nombre de milliwatts (lignes 3.1 - 3.1). Dans le second cas, soit DUF a décidé de baisser le power capping du CPU, donc on augmente le power capping du GPU (ligne 3.1) de la même intensité, soit DUF a décidé d'augmenter le power capping du CPU, alors on baisse le celui du GPU (ligne 3.1). Cela nous permet de toujours respecter le budget qui nous est attribué.

Algorithm 1 Power capping dynamique sur architectures hétérogènes

```
1:  gpu_base_power ← measure_gpu_power()
2:  loop                                              ▷ Chaque period
3:      gpu_power ← measure_gpu_power()
4:      if gpu_power = gpu_base_power then then
5:          power_diff ← (REAL_CPU_MAX_PCAP - CPU_PCAP)
6:          GPU_PCAP ← GPU_PCAP - power_diff
7:          CPU_PCAP ← REAL_CPU_MAX_PCAP
8:      else if CPU_powercap_has_decreased() then
9:          increase(GPU_PCAP)
10:     else if CPU_powercap_has_increase() then
11:         decrease(GPU_PCAP)
12:     end if
13: end loop
```

3.2 Mise en œuvre

Au départ, dans DUF, les GPU n'étaient pas pris en compte. Pour intégrer la gestion hétérogène, j'ai d'abord introduit le fait de pouvoir augmenter ou diminuer le power capping du GPU via deux fonctions `increase_gpu_powercap()` et `decrease_gpu_powercap()` en vérifiant bien que le changement ne dépasse pas les limites constructeur (minimale et maximale). J'ai pu ajouter des appels à ces deux fonctions à tous les endroits dans le code de DUF où leurs équivalentes pour le CPU étaient appelées. Comme cela, à chaque fois qu'on

baisse le power capping du CPU, on augmente celui du GPU, et inversement, toujours en vérifiant au préalable qu'on ne dépasse aucune limite de consommation constructeur, que ce soir du côté CPU ou GPU. De cette manière, les deux limites CPU et GPU se compensent naturellement.

Ensuite, j'ai ajouté une option au programme pour que l'utilisateur puisse choisir le pourcentage de consommation de puissance à ne pas dépasser. C'est une manière de simuler un power capping imposé par un administrateur. DUF fonctionnait déjà connaissant la limite de puissance constructeur, que j'ai remplacé dans le code par la limite imposée par l'utilisateur. De ce fait, DUF fonctionne normalement avec la limite de l'utilisateur qu'il prend pour la limite constructeur. La véritable limite constructeur a aussi été ajoutée pour la fonctionnalité de la détection de l'activité sur GPU.

Pour la détection de l'activité, j'ai utilisé des compteurs sur le GPU pour récupérer la consommation en quasi temps réel. DUF a un module de régulation pour décider quelle action réaliser après avoir pris des mesures. J'ai ajouté à ce module la détection de l'activité sur GPU comme étant la première chose à regarder. Si on détecte une hausse de la consommation du GPU par rapport à celle de base (mesurée au lancement de DUF) de plus de 20%, on considère que le GPU exécute une application, et donc on exécute DUF normalement (avec la gestion dynamique et hétérogène). En revanche, si on détecte que la consommation du GPU est la même que celle de base, on applique la dernière partie que j'ai implémentée. Le choix du 20% vient d'un résultat empirique que j'ai obtenu en mesurant la différence de consommation entre le GPU qui exécute une application, et le GPU qui n'exécute aucune application. Au départ je voulais utiliser les FLOPS pour détecter l'activité, mais il s'avère que je n'ai pas réussi à faire fonctionner les compteurs de performance de NVML. Ce serait quelque chose à investiguer pour des travaux futurs.

Le dernier ajout que j'ai implémenté dans DUF, c'est le fait d'augmenter le power capping du CPU jusqu'à la limite constructeur, et donc de dépasser la limite imposée par l'utilisateur, tout en baissant du même montant le power capping du GPU, toujours en respectant les limites minimale et maximale que ce soit du côté CPU ou GPU.

4 Configuration expérimentale

Cette section présente les application et les outils que j'ai utilisés lors de mon stage.

4.1 Applications

Les applications que je décris vont permettre de simuler le comportement d'une application qui s'exécute sur architecture hétérogène. Nous avons donc deux applications : NAS pour le CPU, et MAGMA pour le GPU.

- Numerical Aerodynamic Simulation (NAS) Parellel Benchmarks [10] est un ensemble de programmes d'évaluation pour architectures parallèles fourni par la NASA. Il y en a de plusieurs sorte pour pouvoir tester une machine sur plusieurs aspects, que ce soit au niveau de la mémoire, de la fréquence, etc... Ils sont dérivés de programmes de mécanique des fluides numérique. J'utilise ces programmes pour évaluer les performances de DUF avec les différentes modifications que j'ai apportées durant mon stage. Le choix s'est porté sur ce groupe de benchmarks car les concepteurs de DUF ont utilisé le même ensemble pour évaluer ses performances [8] [9].

  Plusieurs noyaux d'exécution sont proposés. Nous devons nous intéresser aux noyaux qui nous permettent de tester les différents aspects de notre version de DUF. Nous avons donc choisi un gradient conjugué (CG), qui est intense d'un point de vu mémoire, et un noyau dit "embarrassingly parallel" c'est-à-dire très intense d'un point de vue CPU car il peut s'exécuter sur tous les cœurs sans créer de conflits.

  Les benchmarks proposés possèdent plusieurs classes par noyaux. Les classes permettent d'organiser les benchmarks par taille de problème. Elles sont rangées de A à C pour les problèmes de taille standards et de D à F pour les problèmes plus large. Il y a aussi les tailles S et W pour les problèmes plus légers. Nous allons utiliser les benchmarks de classe C, car ce sont ceux possédant les problèmes les plus larges de la taille standard proposé par les NAS Parallel Benchmark. La classe B ne nous permettait pas d'avoir des temps d'exécution assez long pour avoir assez de données exploitables.

- Matrix Algebra for GPU and Multicore Architectures [11] est une bibliothèques C d'algèbre linéaire pour architecture hétérogène. Elle contient des noyaux exécutables sur processeur et sur processeur graphique. La bibliothèque contient également une série de mini programmes appelés "testings_x" avec x comme nom du noyaux (par exemple dgemm pour une multiplication de matrice avec des flottant de précision double).

J'ai utilisé un de ces dernier en tant que benchmark pour le processeur graphique, plus particulièrement la multiplication de matrice (gemm). Cependant, j'ai dû modifier ce programme (appelé testing_dgemm) pour le faire tourner autant de temps que je voulais. Étant limité à 16 Go de mémoire sur le GPU, je ne pouvais pas mettre une taille de matrice assez large pour que le temps d'exécution soit aussi long que celui des applications sur CPU. Les modifications apportées sont assez simples. J'ai simplement fait en sorte que le noyau s'exécute en boucle un nombre défini de fois sans qu'il sorte du GPU, pour que l'exécution impact le moins possible le processeur.

4.2 Outils de programmation

Dans cette partie je décris les différentes bibliothèques et API que j'ai utilisées lors de l'implémentation de notre solution, ainsi que les programmes et le matériel que j'ai utilisés pour réaliser les expériences.

4.2.1 PAPI

La Performance Application Programming Interface [12] est une API basée sur le C qui propose des outils de mesure bas niveau orientés sur la performance. Elle permet entre autre d'obtenir la liste des compteurs disponibles sur le matériel que l'on a à disposition (processeur, processeur graphique, etc...), et d'intégrer des outils de mesure au sein d'un programme pour avoir une mesure en quasi temps réel des compteurs présents sur les composants d'une machine.

PAPI fonctionne grâce à des événements qui représentent des paramètres de mesure. On va donc pouvoir construire des collections d'événements qui vont servir lors d'un appel à la fonction PAPI_read qui récupère les mesures correspondant aux événements de notre collection. Les mesures doivent être prises dans une zone de mesure délimitée par les fonctions PAPI_start et PAPI_stop. Dans notre cas, nous avons une collection pour les événements CPU, une autre pour les événements "uncore", et une dernière pour les événements GPU.

4.2.2 NVML

La NVIDIA Managment Library [13] est une API basée sur le C qui propose, comme PAPI, des outils de mesures bas niveau mais cette fois-ci orientés spécifiquement pour les processeurs graphiques du constructeur NVIDIA. L'utilisation de la bibliothèque fournie par NVML permet de sortir du

système d'événement de PAPI (qui utilise NVML pour les mesures sur processeur graphique NVIDIA), ce qui peut permettre de simplifier le code.

La bibliothèque C NVML fournie une grande quantité de fonctions qui vont permettre de récupérer des données lors du fonctionnement du processeur graphique comme la consommation, l'horloge graphique ainsi que d'autres éléments. Elle permet aussi de mettre à jour certains paramètres comme la limite de consommation maximale, ce qui est dans notre cas, très utile.

4.2.3 Powercap

Powercap [14] est un bibliothèque C utilisée par DUF qui permet de contrôler la limite de puissance sur les processeurs Intel via l'interface RAPL. La bibliothèque propose deux programmes powercap-info et powercap-set qui permettent respectivement d'obtenir des informations sur les limites de consommation possible, et de configurer une limite de consommation, le tout en précisant une zone et une contrainte sur le processeur. Dans le cadre du stage, j'ai utilisé powercap-info et powercap-set pour les expériences n'utilisant pas la fonctionnalité de gestion dynamique de la limite de consommation de DUF, à des fins de comparaison.

4.2.4 NVIDIA System Management Interface

NVIDIA System Management Interface [13] est un programme faisant parti de l'API NVML. Il permet, comme les programmes de powercap, d'obtenir des informations et de configurer les limites de consommation de puissance, mais cette fois ci sur les processeurs graphique NVIDIA. J'ai utilisé NVIDIA SMI dans le même contexte que Powercap, c'est à dire dans des expériences n'utilisant pas la fonctionnalité de gestion dynamique de la limite de consommation du DUF.

4.2.5 Platefrome d'exécution

Grid'5000 [15] est une plateforme de recherche regroupant des sites de cluster de calcul en France. La plateforme offre un choix conséquent d'architectures différentes en fonction des besoins avec ses 16144 cœurs sur 758 nœuds au sein de 43 clusters répartis sur 9 sites. L'avantage qu'apporte Grid'5000 c'est d'avoir accès aux droits administrateurs dans un environnement complètement contrôlé par l'utilisateur.

Chifflot est un cluster du site de Lille sur Grid'5000. Il possède 8 nœuds, avec chacun 2 Intel Xeon Gold 6126 12 cœurs (donc 48 cœurs par nœud).

Chaque nœud possède deux accélérateurs de calcul. Sur les six premiers nœuds, ce sont des NVIDIA Tesla P100 16 Go et sur les deux derniers (Chifflot-7 et Chifflot-8) ce sont des NVIDIA Tesla V100 32 Go. Durant les expériences, pour éviter les variations de consommation liée aux différents nœuds, j'ai mené toutes les expériences sur un seul nœud, le sixième du cluster Chifflot, Chifflot-6.

Nous avons choisi ce cluster car il contient des accélérateurs de calcul puissants ainsi que des processeurs de la même architecture (Skylake) et de la même génération que ceux du cluster utilisés par les concepteurs de DUF lors des tests réalisés [8]. L'avantage d'avoir une architecture similaire est de pouvoir utiliser le fichier de configuration déjà présent pour les anciens tests comme base pour faire un nouveau fichier pour l'architecture de Chifflot.

5 Expériences

Cette section à pour but de décrire les expériences menées et de montrer les résultats préliminaires que nous avons obtenus. Toutes les expériences sont effectuées sur deux processeurs, un GPU, au sein d'un seul nœud.

Les expériences menées servent à constater l'impact des modifications apportées à DUF. L'objectif de l'extension que j'ai implémentée est d'avoir de meilleures performances lorsqu'on utilise DUF sur architecture hétérogène par rapport à une exécution sans DUF, tout en respectant la limite de budget qui nous est accordée.

5.1 Configuration des expériences

Nous avons mis en place trois versions : une version qui n'utilise pas DUF, juste les benchmarks et de quoi changer le power capping (comme sur la figure 5.1.1), une autre version (que l'on retrouve sur la figure 5.1.2) avec DUF seulement en tant qu'outil de mesure, une version avec DUF qui utilise le power capping dynamique mais sans la détection d'activité sur GPU (un exemple est visible sur la figure 5.1.3). Nous ne testons pas la fonctionnalité de la détection de l'activité sur le GPU car pour l'instant, dans notre configuration, le GPU est utilisé pendant toute la durée des expériences. Pour tester cette fonctionnalité, il nous faudrait une application qui utilise le CPU et le GPU de manière irrégulière.

DUF fait des mesures toutes les 200ms, c'est l'intervalle qui permet d'avoir le meilleur compromis entre le temps d'exécution, et des mesures précises. En effet, un intervalle plus petit introduirait un temps d'exécution

plus long, un intervalle plus long ferait qu'on raterait des évènements (comme décrit dans la publication sur DUF [8])

Chaque version est lancée sous différents power capping : 90% et 50% de la puissance maximale. Cela nous permet de voir l'évolution du comportement de l'application en fonction de la limite imposée. Nous avons pris 90% pour avoir une référence proche des 100% mais sans les atteindre (on travaille toujours sous la contrainte d'un power capping imposé), et 50% car c'était le power capping le plus bas qu'on pouvait atteindre sans avoir des temps d'exécution trop longs. Chaque configuration est exécutée 5 fois pour obtenir une moyenne.

5.1.1 Version par défaut

Cette version est utile car elle nous permet d'avoir une base pour nos expériences, en effet elle mesure le temps d'exécution brut des benchmarks CPU et GPU qui tournent en même temps sans qu'il y ait d'autres applications qui parasitent les temps d'exécution. Puisque DUF n'est pas exécuté, il a fallu modifier le power capping via d'autres outils externes à DUF. Au lancement de l'itération, Powercap 4.2.3 est utilisé pour modifier le power capping initial du CPU, et NVIDIA-SMI 4.2.4 pour modifier le power capping initial du GPU.

5.1.2 Versions de mesure

Pour cette version, DUF est exécuté en même temps que les benchmarks, et il n'utilise que sa capacité de mesure. Cela nous permet de récupérer la puissance consommée par les benchmarks lors de leurs exécutions.

5.1.3 Version sans détection

Cette version utilise DUF ainsi que la gestion dynamique sur CPU et GPU implémentée durant le stage, mais sans la détection de l'activité sur GPU. Le temps d'exécution récupéré dans cette version va être comparé à celui obtenu dans la version par défaut 5.1.1 et la consommation mesurée sur cette version va être comparée à celle obtenue dans la version de mesure 5.1.2. Ces comparaisons vont nous permettre d'obtenir des ratios de puissance, de temps d'exécution et d'énergie pour les résultats.

5.2 Configuration des applications

À chaque itération de l'expérience, une fois que DUF est lancé, on exécute un noyau de NAS sur les 48 cœurs des 2 processeurs de Chifflot-6, et le noyau gemm de MAGMA sur GPU pour qu'ils s'exécutent en même temps. Lorsque les deux benchmark ont terminé, on copie les données et on recommence avec d'autres paramètres (nouvelle itération, power capping différent, autre noyau NAS). La multiplication de matrice de MAGMA est lancée avec les paramètres suivants : une taille de 20000 et 24 itération. Cela permet d'avoir un temps globalement équivalent à celui de l'application CPU. J'ai choisi 20000 car cela permettait d'utiliser une bonne partie de la mémoire sans pour autant la saturer. 24 itérations me permettait d'atteindre à peut-prêt le même temps d'exécution que les applications sur CPU.

5.3 Résultats préliminaires

La figure 3 nous montre le ratio de ce que nous avons gagné (ou perdu) en consommation de puissance, en temps d'exécution et en consommation d'énergie. La figure représente les données d'une expérience avec le noyau CG de NAS Parallel Benchmarks sur le CPU représenté par les valeurs non hachurées, et avec le noyau de multiplication de matrice de MAGMA sur le GPU représenté par les valeurs hachurées. Nous avons aussi les deux paramètres de power capping à 50% en jaune et à 90% en vert. "Gain" représente ce que l'on a gagné, c'est à dire que nous avons économisé en termes de consommation (puissance ou énergie), ou que l'on a un temps d'exécution plus court. Si c'est en dessous de 0 c'est que c'est de la perte, donc une consommation plus élevée ou un temps d'exécution plus long.

Les écarts en orange permettent de situer le maximum et le minimum en considérant les 5 itérations. On remarque que les écarts sur les mesures prisent sur le GPU sont plus importantes. On peut en déduire que les expériences sur GPU sont plutôt instables. Cela est sûrement du au fait que le noyau de multiplication de matrice est exécuté plusieurs fois d'affilé pour avoir un temps d'exécution équivalent à celui côté CPU 4.1, cela à pour conséquence d'additionner les variations et donc de créer des écarts plus importants.

En regardant la partie liée à la puissance, on observe que la marge d'erreur est un peu trop grande pour avoir une conclusion précise. Si on additionne les consommation CPU et GPU on constate que l'on consomme globalement un peu tout petit peu moins (moins de 1%), que ce soit à 50 ou à 90%.

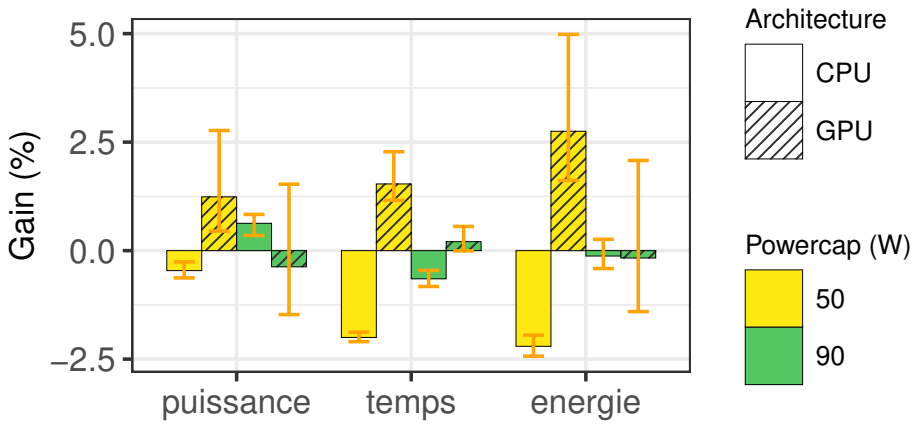


Figure 3: Résultats sur CG

Si on regarde le temps d'exécution, on a moins d'écart, mais les changements de comportement par rapport à la version par défaut restent très faibles (moins de 2%).

Pour ce qui est de l'énergie, on retrouve de grands écarts, et on reste avec des changements très faibles. Si on additionne les moyennes des ratios de consommation CPU et GPU on voit qu'on tombe très proche du 0%.

Ce changement très faible est sûrement dû au profil de l'application. En effet, CG consomme presque l'entièreté du budget qu'on lui accorde. À 90%, le noyau consomme entre 106 et 108 Watt, la consommation maximale autorisée sous la contrainte du power capping est à 112 Watt. CG ne permet donc pas à l'algorithme de baisser le power capping dynamique. Le noyau EP consomme un peu moins de puissance, mais pour l'instant les résultats que nous avons ne sont pas exploitables car nous ne comprenons pas certains comportements et sommes toujours en train d'étudier l'exécution en détail pour les expliquer.. Les expériences sont toujours en cours.

6 État de l'art

Il y a très peu de publication qui traitent le sujet du power capping dynamique. Nous avons [4–6] qui proposent des études à ce sujet, mais seulement sur CPU.

La plupart des publications qui traitent le sujet du power capping et de l'hétérogénéité le font de manière statique. [16–18] par exemple établissent

des modèles à partir de l'architecture ou des applications qu'ils font tourner pour décider la meilleure limite de puissance à appliquer en amont.

D'autres publications proposent un power capping statique, mais ces dernières le font sur un seul type de composant en particulier. On perd donc ici le côté hétérogène. Par exemple [19] propose une méthode de power capping statique mais seulement pour GPU.

Les autres publications ne proposent pas de méthode pour faire varier le power capping mais établissent des techniques d'ordonnancement [20] ou bien font de l'optimisation [21, 22] en considérant un power capping. À notre connaissance, nous sommes donc les seuls à avoir proposé une étude qui traite de power capping dynamique sur architecture hétérogène, ce qui permet de s'adapter au moment où l'application tourne, sans se soucier de comportements imprévisibles qui seraient difficiles à prévoir avec des heuristiques.

7 Conclusion

Lors de mon stage, j'ai pu apprendre l'architecture de l'outil DUF pour y implémenter une extension avec un algorithme de gestion dynamique du power capping sur architecture hétérogène. Pour l'ajouter, j'ai appris à me servir de plusieurs bibliothèques dont PAPI [12], NVML [13], ainsi que de plusieurs outils comme Powercap [14] et NVIDIA-SMI [13]. J'ai également appris à me servir d'une plateforme de cluster pour le calcul haute performance [15]. Aussi, j'ai appris, et je continue à apprendre, à mettre en place des expériences pour tester un programme.

Une grande partie de mon travail a été d'essayer d'avoir des résultats exploitables en faisant des expériences. J'aimerais continuer à en faire pour constater les réelles différences et est-ce que cette solution apporte des bénéfices en termes de performance à DUF.

Étant recruté en thèse à l'issue de mon stage, je continuerai ce travail pour faire avancer les expériences, obtenir des résultats ainsi que tester cette nouvelle version de DUF sur une véritable application et non des benchmarks. D'autres ajouts à cette extension de DUF pourrait être intéressants comme le support multi-GPU ou bien pousser plus loin la détection de l'activité sur le GPU pour être plus fin dans la prise de décision en ce qui concerne le power capping.

8 Remerciements

Les expériences présentées dans ce rapport ont été réalisées sur la plateforme Grid'5000, soutenue par groupement d'intérêt scientifique supporté par l'Inria, et comprenant le CNRS, RENATER et plusieurs universités ainsi que d'autres organisations (voir `https://www.grid5000.fr`).

Je souhaite remercier Amina Guermouche de m'avoir encadré durant le stage et de m'avoir guidé au long de ce projet que je continuerai en thèse.

Bibliographie

[1] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[2] Issam Raïs, Anne-Cécile Orgerie, Martin Quinson, and Laurent Lefèvre. Quantifying the impact of shutdown techniques for energy-efficient data centers. *Concurrency and Computation: Practice and Experience*, 30(17):e4471, 2018. e4471 cpe.4471.

[3] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 460–469, New York, NY, USA, 2009. Association for Computing Machinery.

[4] Srinivasan Ramesh, Swann Perarnau, Sridutt Bhalachandra, Allen D. Malony, and Pete Beckman. Understanding the impact of dynamic power capping on application progress. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 793–804, 2019.

[5] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. Copper: Soft real-time application performance using hardware power capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 31–41, 2019.

[6] Eric Rutten, Sophie Cerf, Raphaël Bleuse, Valentin Reis, and Swann Perarnau. Sustaining performance while reducing energy consumption: A control theory approach, 2021.

[7] Sridutt Bhalachandra, Brian Austin, and Nicholas J. Wright. Understanding power variation and its implications on performance optimization on the cori supercomputer. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 51–62, 2021.

[8] Étienne André, Rémi Dulong, Amina Guermouche, and François Trahay. Duf: Dynamic uncore frequency scaling to reduce power consumption. *Concurrency and Computation: Practice and Experience*, 34(3):e6580, 2022.

[9] Amina Guermouche. Combining uncore frequency and dynamic power capping to improve power savings. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1028–1037, 2022.

[10] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, 1991.

[11] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[12] Anthony Danalis and Heike Jagode. *Performance Application Programming Interface*. Sun, Baruah and Kaeli, 2022-12 2022.

[13] NVIDIA Corporation. Nvml api reference.

[14] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. CoPPer: Soft real-time application performance using hardware power capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 31–41, 2019.

[15] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[16] Reza Azimi, Chao Jing, and Sherief Reda. Powercoord: A coordinated power capping controller for multi-cpu/gpu servers. In *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, pages 1–9, 2018.

[17] Kazuki Tsuzuku and Toshio Endo. Power capping of cpu-gpu heterogeneous systems using power and performance models. In *2015 International Conference on Smart Cities and Green ICT Systems (SMARTGREENS)*, pages 1–8, 2015.

[18] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa, and Hiroshi Nakamura. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 349–356, 2013.

[19] Dong-Ki Kang, Yun-Gi Ha, Limei Peng, and Chan-Hyun Youn. Cooperative distributed gpu power capping for deep learning clusters. *IEEE Transactions on Industrial Electronics*, 69(7):7244–7254, 2022.

[20] Qi Zhu, Bo Wu, Xipeng Shen, Li Shen, and Zhiying Wang. Co-run scheduling with power cap on integrated cpu-gpu systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 967–977, 2017.

[21] Adam Krzywaniak and Paweł Czarnul. Performance/energy aware optimization of parallel applications on gpus under power capping. In Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 123–133, Cham, 2020. Springer International Publishing.

[22] Kramer Straube, Jason Lowe-Power, Christopher Nitta, Matthew Farrens, and Venkatesh Akella. Improving provisioned power efficiency in hpc systems with gpu-capp. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 112–122, 2018.