Titre: Title:	Traçage et profilage de systèmes hétérogènes
Auteur: Author:	Arnaud Fiorini
Date:	2020
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Fiorini, A. (2020). Traçage et profilage de systèmes hétérogènes [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/5397/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Traçage et profilage de systèmes hétérogènes

ARNAUD FIORINI
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Traçage et profilage de systèmes hétérogènes

présenté par **Arnaud FIORINI**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Gilles PESANT, président
Michel DAGENAIS, membre et directeur de recherche
Guillaume-Alexandre BILODEAU, membre

REMERCIEMENTS

Je tiens à exprimer toute ma reconnaissance pour mon directeur de recherche, Michel Dagenais, pour m'avoir donné l'opportunité de participer à ce projet de recherche au sein du laboratoire DORSAL. Son expérience m'a beaucoup guidé dans les différents problèmes rencontrés au cours de ma maitrise.

Je tiens aussi à remercier le soutien financier, technique et matériel des entreprises : Ericsson, Advance Micro Devices (AMD), EfficiOS ainsi que du Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG).

Je remercie spécifiquement Geneviève Bastien, Matthew Khouzam, Guillaume Champagne et Evgeny Shcherbakov pour leurs conseils et leur aide tout au long de mon projet.

Par ailleurs, j'aimerais exprimer ma gratitude à tous mes collègues du laboratoire DORSAL qui m'ont souvent apporté des conseils ou de l'aide.

Enfin, j'aimerais remercier ma famille ainsi que mes amis pour leur soutien constant.

RÉSUMÉ

Les systèmes hétérogènes sont de plus en plus présents dans tous les ordinateurs. En effet, de nombreuses tâches nécessitent l'utilisation de coprocesseurs spécialisés. Ces coprocesseurs ont permis des gains de performance très importants qui ont mené à des découvertes scientifiques, notamment l'apprentissage profond qui n'est réapparu qu'avec l'arrivée de la programmation multiusage des processeurs graphiques.

Ces coprocesseurs sont de plus en plus complexes. La collaboration et la cohabitation dans un même système de ces puces mènent à des comportements qui ne peuvent pas être prédits avec l'utilisation d'analyse statique. De plus, l'utilisation de systèmes parallèles qui possèdent des milliers de fils d'exécution, et de modèles de programmation spécialisés, rend la compréhension de tels systèmes très difficile. Ces problèmes de compréhension rendent non seulement la programmation plus lente, plus couteuse, mais empêchent aussi le diagnostic de problèmes de performance.

C'est pourquoi il est nécessaire d'avoir recours à des outils d'analyse dynamique qui permettent d'analyser l'état d'un système au cours du temps. Ces outils permettent d'étudier en détail le comportement d'un système après l'exécution de celui-ci. Idéalement, ces outils ont un impact minimal pour s'assurer de la précision des résultats obtenus. Il est aussi nécessaire que les résultats obtenus permettent d'avoir suffisamment d'information pour décrire complètement le système, sinon l'analyse est fortement biaisée.

Il est cependant de plus en plus difficile d'obtenir ces outils, car dans les systèmes hétérogènes de nombreux processeurs fonctionnent ensemble, tout en ayant des architectures, des gestionnaires de périphérique et des modèles de programmation différents. Cette diversité doit être prise en compte dans l'élaboration d'un outil de traçage. L'avènement de l'architecture HSA permet le fonctionnement dans un cadre commun, basé sur un modèle d'exécution générique, de ces différents coprocesseurs.

Cette architecture permet donc d'obtenir des outils suffisamment génériques pour s'adapter à la diversité des systèmes hétérogènes. Dans ce mémoire, nous présentons une approche en deux temps. En premier lieu, l'utilisation de ROC-profiler et de ROC-tracer qui permet de recueillir des traces qui peuvent être corrélées avec des moyens de traçage plus traditionnels pour obtenir une vue d'ensemble d'un système. Ensuite nous avons développé une analyse des données de traçage et de profilage. Cette analyse s'effectue en deux étapes, la lecture séquentielle des évènements qui permet de générer un historique des états et ensuite, l'affichage de ces états pour générer plusieurs visualisations. La génération de traces s'effectue

avec les outils ROC-profiler et ROC-tracer qui font partie de ROCm, une implémentation de la spécification HSA. Ensuite une analyse de cette trace et des compteurs de performance a été développée grâce à TraceCompass.

Cette analyse permet de visualiser les appels à l'interface de programmation HIP et HSA effectués par l'utilisateur, les transferts mémoires et les compteurs de performance. Ces informations sont affichées au sein d'une même vue. Cela permet à l'utilisateur de visualiser toutes les données pertinentes, synchronisées au niveau du temps. L'outil utilisé permet de corréler facilement ces données avec les données d'exécution du système d'exploitation, du CPU et d'autres systèmes communs à tous les systèmes hétérogènes. Cette approche est facilement adaptable à n'importe quel système respectant la spécification HSA, ce qui est le cas de nombreux coprocesseurs qui supportent cette spécification.

ABSTRACT

Heterogeneous systems are becoming increasingly relevant and important with the emergence of powerful specialized coprocessors. Because of the nature of certain problems, like graphics display, deep learning and physics simulation, these devices have become a necessity. The power derived from their highly parallel or very specialized architecture is essential to meet the demands of these problems. Because these use cases are common on everyday devices like cellphones and computers, highly parallel coprocessors are added to these devices and collaborate with standard CPUs.

The cooperation between these different coprocessors makes the system very difficult to analyze and understand. The highly parallel workload and specialized programming models make programming applications very difficult. Troubleshooting performance issues is even more complex. Since these systems communicate through many layers, the abstractions hide many performance defects.

Therefore, it is becoming essential to use diagnostic toolkits that record performance details. These performance details can be traces or performance counters. They are very useful in order to analyze the state of a system, with respect to time, and its behaviour. However, these tools should minimize the impact on the execution of the studied applications to be effective in finding performance defects. Also, to cover most use cases, it is necessary to have enough data to adapt and produce different kinds of analysis.

It is very difficult to obtain tools that are compatible with most coprocessors and that can be correlated with more traditional tracers and profilers. There are many different programming models and architectures, and most tools are closed source and do not provide access to the relevant information that would allow correlating with other sources. The HSA foundation enables many software improvements that would be generic enough to support different programming models.

In this thesis, we propose a novel approach that uses ROC-profiler, ROC-tracer and TraceCompass to achieve offline analysis of heterogeneous program executions. The tracing and profiling data collection tools that record runtime information, ROC-tracer and ROC-profiler, are developed mainly by AMD and are open source. Therefore they can be easily modified to correlate their data with other tools. We use these tools to generate a trace and performance counters. Then, we developed an analysis with TraceCompass to process this trace and that produces two new views.

These two views show HIP and HSA API calls, memory transfers and compute kernels, all synchronized with respect to time. This enables the user to have an overview of the program execution on heterogeneous systems. This approach allows the user to simultaneously profile and trace the program, is compatible with different hardware, and would easily adapt to other modern processors that comply with the HSA requirements.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

GPU	Graphic Processing Unit
UCT	Unité Centrale de Traitement
FPGA	Field Programmable Gate-Array
AMD	Advanced Micro Devices
CUDA	Compute Unified Device Architecture
ROCm	Radeon Open Compute platforM
HIP	Heterogeneous-Computing Interface for Portability
GDB	GNU Debugger
DSP	Digital Signal Processor
OTF	Open Tracing Format
CTF	Common Trace Format
VTF	Vampir Trace Format
API	Application Protocol Interface
TAU	Tuning and Analysis Utilities
CUPTI	CUDA Profiling Tools Interface
PAPI	Performance API
PTX	Parallel Thread Execution
LTTng	Linux Trace Toolkit next generation
HSA	Heterogeneous System Architecture
LRU	Least Recently Used
PLRU	Pseudo LRU
MRU	Most Recently Used
GCN	Graphics Core Next
TLP	Thread Level Parallelism
SIMD	Single Instruction Multiple Data
HPC	High Performance Computing
PCIe	Peripheral Component Interconnect express
DMA	Direct Memory Access

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Les systèmes hétérogènes sont de plus en plus présents pour répondre aux besoins des utilisateurs. En effet, de nombreuses tâches nécessitent des processeurs spécialisés pour être exécutées avec la rapidité voulue. Ces tâches ayant des demandes particulières en termes de performance, il est important d'avoir les outils nécessaires pour atteindre ces objectifs.

Les analyses statiques sont limitées et ne peuvent pas aider beaucoup pour l'analyse de performance d'applications complexes. Il est donc nécessaire d'avoir recours à des techniques d'analyse dynamique comme le traçage et le profilage. Ces techniques sont difficiles à appliquer dans des systèmes de calcul à haute performance, car le surcout est proportionnel au nombre d'évènements. Afin de ne pas modifier le comportement des applications étudiées, on cherche à minimiser le surcout des outils qui collectent les données de performance. Ceci impose des contraintes importantes pour développer des outils d'analyse de performance efficaces. De plus, les systèmes hétérogènes présentent de nombreuses architectures différentes qui collaborent au sein d'un même programme. Cette collaboration nécessite l'utilisation d'une multitude d'outils pour pouvoir enregistrer les données propres à chaque architecture. Ces outils doivent pouvoir être modifiés pour corréler les données entre elles, et les outils doivent avoir un surcout faible.

Développer de tels outils est difficile, et une grande partie des mécanismes internes pourrait être réutilisée sur d'autres architectures. Avec l'avènement d'architectures hétérogènes qui possèdent de nombreux coprocesseurs, la demande pour des outils génériques qui fonctionnent avec plusieurs types d'architecture est forte. De plus, des outils génériques permettront de corréler les données de différentes sources. C'est pourquoi nous nous intéressons à ce problème dans le présent travail de recherche.

1.1 Définitions et concepts de base

Dans cette section, nous allons définir quelques concepts qui sont importants pour la bonne compréhension de la suite du mémoire.

1.1.1 Système parallèle

Un système parallèle est un système qui exécute plusieurs calculs simultanément sur plusieurs processeurs. Ces calculs sont divisés en instructions et ces instructions sont exécutées simultanément. Il est à différencier du système réparti qui comporte plusieurs ordinateurs qui

communiquent entre en eux à travers une interface réseau. Dans un système réparti, plusieurs ordinateurs se synchronisent sur un temps commun et utilisent une mémoire distincte.

Le processeur central (CPU) est aujourd'hui capable d'effectuer des calculs parallèles. Cependant, on aura souvent recours à des processeurs spécialisés dans le calcul parallèle comme les processeurs graphiques (GPU), les processeurs de signaux numériques (DSP) ou les circuits logiques programmables (FPGA). Ces processeurs spécialisés sont beaucoup plus efficaces pour les tâches nécessitant de nombreux calculs qui sont souvent sur des données ayant plusieurs dimensions, ce qui nécessite un nombre élevé de fils d'exécution.

1.1.2 Système hétérogène

Un système hétérogène est un système composé d'un processeur central et de plusieurs coprocesseurs qui généralement partagent une même mémoire. Ces coprocesseurs sont souvent spécialisés dans certaines tâches et permettent au processeur central de répartir les tâches pour une exécution plus efficace. Aujourd'hui, la mémoire partagée est virtuelle, car souvent les coprocesseurs possèdent leur propre mémoire et doivent maintenir une synchronisation avec la mémoire principale.

Nous avons vu précédemment que les GPU, DSP et FPGA étaient plus appropriés pour traiter des tâches nécessitant du calcul parallèle. C'est pourquoi ces coprocesseurs sont souvent utilisés en supplément du CPU pour des raisons de performance.

1.1.3 Traçage logiciel

Le traçage logiciel est l'enregistrement d'une suite d'évènements d'un programme ou d'un système. Ces évènements sont une représentation d'un état ou d'un changement d'état du système et permettent de traquer l'état du système par rapport à une échelle du temps. Ils comportent une étiquette de temps et des informations par rapport à l'état du programme. Une trace est donc une séquence de ces évènements et permet, en analysant cette séquence, de retracer l'évolution du programme dans le temps et de mieux comprendre le comportement de celui-ci.

Le traçage peut avoir lieu à différents niveaux : soit dans le contexte du système d'exploitation ou alors dans le contexte utilisateur. Si des points de trace sont déjà présents dans le système d'exploitation, on dit que le système est instrumenté. Ces points de trace sont des instructions spécifiques permettant d'enregistrer les informations liées à un évènement.

Le traçage peut être statique, c'est-à-dire utiliser les points de trace déjà présents dans le programme, ou dynamique, où l'on insère à l'exécution des points de trace.

1.1.4 Analyse de trace

L'analyse de trace est la lecture d'une séquence d'évènements qui permet, grâce à une machine à états ou à un algorithme particulier, de produire soit une visualisation de l'état du programme, soit des statistiques sur l'exécution. Le plus souvent, la production d'un graphique montrant l'exécution de différentes tâches, ou de l'état d'un fil d'exécution par rapport au temps, est effectuée.

1.2 Éléments de la problématique

Les systèmes hétérogènes nécessitent donc des outils spécialisés qui sont suffisamment génériques pour pouvoir s'adapter à différentes architectures. Des outils existent aujourd'hui qui permettent l'analyse de processeurs spécialisés. Cependant, ces outils sont presque tous uniquement compatibles avec un modèle de programmation et une architecture bien précise. De plus, il est souvent impossible d'adapter ces outils pour les rendre plus génériques ou les utiliser pour corréler leurs informations avec d'autres outils, car ils sont à source fermée. L'arrivée de spécifications qui définissent une architecture générique permet de développer une approche adaptable et qui permet de répondre à ce problème.

L'analyse d'un système hétérogène est très complexe, car de nombreux systèmes communiquent entre eux et chaque système a sa propre architecture. Il devient ainsi difficile d'avoir une vue d'ensemble du fonctionnement d'un programme. Il est donc nécessaire d'avoir une trousse d'outils qui permet d'obtenir toutes les visualisations nécessaires pour comprendre et détecter les problèmes de performance.

Ces difficultés sont causées par le manque d'un modèle générique de fonctionnement d'un système hétérogène et le manque d'interfaces communes de communication avec ces coprocesseurs. De plus, les outils existants ne permettent pas d'avoir assez d'information pour générer une vue d'ensemble du système. Il est aussi nécessaire d'étudier les avancées actuelles dans le domaine des coprocesseurs pour pouvoir mieux les analyser et instrumenter les librairies nécessaires.

La spécification HSA est un standard qui permet à un système hétérogène d'avoir une interface commune pour accéder aux différents coprocesseurs grâce à une API générique. Cette spécification a été conçue pour répondre aux besoins d'un système hétérogène. Ceci nous permet de développer une application pour ce standard en nous assurant d'être génériques pour pouvoir adapter l'approche à l'avenir. Nous choisirons ROCm comme implémentation du standard HSA. En effet, c'est la seule implémentation suffisamment complète qui permette de supporter les modèles de programmation parallèles les plus utilisés tels que OpenMP,

OpenCL, CUDA et HIP. De plus, des outils de traçage et de profilage associés à ROCm permettent de commencer à expérimenter plus rapidement. Enfin, ROCm est à source ouverte, ce qui permet d'instrumenter le code de façon efficace.

1.3 Objectifs de recherche

L'objectif principal de cette recherche est de produire une approche permettant d'obtenir une vue d'ensemble du fonctionnement d'un système hétérogène comprenant les évènements des coprocesseurs.

Pour atteindre cet objectif, il est nécessaire en premier lieu de comprendre les nouvelles technologies qui sont présentes dans les coprocesseurs, et ensuite d'évaluer la spécification HSA. Par la suite, il faut implémenter à l'aide de ROCm une approche pouvant profiler et tracer : le comportement du programme dans le contexte utilisateur, les communications avec le coprocesseur, et le programme s'exécutant sur le coprocesseur. ROCm est une collection de librairies qui implémente le standard HSA. Cette collection étant à source ouverte, nous pourrons l'utiliser facilement pour développer une approche qui sera adaptable à d'autres environnements.

Un système hétérogène fait intervenir de nombreux coprocesseurs ayant des architectures différentes. Nous avons vu précédemment que pour comprendre un tel système, il est nécessaire d'obtenir les données de tous les composants pour avoir une vue d'ensemble. Donc l'un des objectifs est d'avoir un outil capable de s'adapter à chaque coprocesseur. Il est ainsi nécessaire de fonctionner avec plusieurs architectures.

Comme nous développons une approche qui doit s'exécuter sur des programmes où la performance est primordiale, il sera important de garder un surcout relativement faible. Enfin, il sera nécessaire de développer plusieurs vues qui permettent d'offrir une vue d'ensemble de l'exécution du programme sur un outil de visualisation. Notre approche utilisera Trace Compass pour la visualisation, car cet outil est à source ouverte, est facilement adaptable, et possède tous les outils de corrélation nécessaires pour compléter notre visualisation.

Questions de recherche

Nous pouvons donc poser les questions de recherche suivantes :

1. Quelles sont les données requises pour l'analyse d'un système hétérogène ?
2. À partir de quels outils pouvons-nous produire ces données ?
3. Quelles visualisations permettent de rendre les données plus accessibles et compréhen-

sibles ?

1.4 Plan du mémoire

Suite à cette introduction, nous présenterons, dans le chapitre 2, l'état de l'art des processeurs graphiques, processeurs de signaux et autres coprocesseurs. Nous étudierons en détail les avancées dans le domaine du calcul sur système hétérogène et ensuite nous verrons les différentes techniques et outils de traçage, profilage et débogage.

Le chapitre 3 est consacré à la présentation d'un article scientifique qui détaille l'approche utilisée, son évaluation et les résultats de cette recherche.

Le chapitre 4 est une discussion supplémentaire, suite à cet article, qui complétera l'article avec quelques détails additionnels.

Nous conclurons dans le chapitre 5 en faisant une synthèse des travaux, en présentant ses limites et en proposant des améliorations futures.

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE

Dans cette revue de littérature, nous allons d'abord aborder les problématiques liées aux calculs sur des systèmes hétérogènes. En effet, cette partie est primordiale pour bien comprendre quelles sont les causes des défauts de performance sur les systèmes hétérogènes et quels sont les défis qu'apportent ces systèmes. Cette première partie aura donc pour but d'analyser ces défis pour pouvoir produire ensuite des outils et des visualisations pertinentes qui permettront une analyse ciblée. Cette analyse nous permettra ensuite dans une deuxième partie d'étudier comment et avec quels outils, ces problèmes sont résolus. Ces outils nous permettront d'établir une base, pour adapter notre approche en fonction de ce qui existe déjà. De plus, ces outils peuvent être utilisés dans le futur pour recueillir certaines informations, ou corriger d'éventuelles lacunes de notre approche.

2.1 Calcul sur systèmes hétérogènes

Cette première partie se porte sur les trois domaines de recherche qui cherchent à optimiser et réduire le temps d'exécution pour des applications de calculs à haute performance sur des systèmes hétérogènes : le partage de la mémoire et la communication entre les différents composants du système, l'ordonnancement des calculs et comment est réparti la charge sur les accélérateurs et le CPU, enfin comment sont résolus les problèmes au niveau de la mémoire tampon qui est un enjeu majeur pour la performance. La majeure partie des recherches sur les systèmes hétérogènes se portent sur les GPUs. En effet, ce type de coprocesseur est le plus utilisé aujourd'hui dans les systèmes hétérogènes. Conséquemment, une grande partie des sujets exposés dans cette partie ne sont pas pertinents pour d'autres types de coprocesseurs.

De plus, même si les téléphones possèdent de nombreux coprocesseurs et sont une des cibles de cette recherche, les technologies utilisées ne sont pas encore disponibles sur cette plateforme.

2.1.1 Mémoire partagée / Communication interprocesseurs

Aujourd'hui, le principal ralentissement des programmes est dû aux lectures et écritures en mémoire. En effet, la différence de vitesse entre les processeurs et les mémoires a considérablement augmenté [2, 3]. C'est pourquoi la communication interprocesseurs est très importante dans un système hétérogène, surtout dans le cas d'un très grand nombre de calculs parallèles.

Accès direct en mémoire

Dans un système hétérogène, des échanges d'information sont nécessaires entre le CPU et d'autres types de puces comme un processeur graphique (GPU), un processeur de traitement numérique des signaux (DSP) ou une matrice FPGA. Ces échanges utilisent un mécanisme d'accès direct en mémoire [4–7]. Celui-ci fonctionne à l'aide d'un circuit supplémentaire, le contrôleur DMA, qui effectue les traductions d'adresse, interrompt le CPU quand le transfert est terminé, et gère le trafic du bus vers le CPU.

Ce type de transfert de mémoire n'a pas été conçu pour communiquer avec des accélérateurs hautement parallèles. De nombreux défauts de ce système sont donc présents :

Ces transferts de mémoire doivent être faits explicitement à travers l'API fournie (CUDA, OpenCL). En général, ces échanges de données nécessaires introduisent de grandes inefficacités [6] : des accès non uniformes qui sont nécessaires dans de nombreuses tâches : algorithme de graphe, inférence par apprentissage automatique [4,8]. De plus, ces accélérateurs ont besoin de grandes quantités de mémoire, et cela pose différents problèmes soulignés par [5, 6]. Les communications intensives dans un programme ont tendance à générer beaucoup de bulles ("pipeline stalls"). Ces bulles peuvent être masquées en changeant l'exécution vers un autre groupe de fils d'exécution [4, 9]. Cependant, dans des programmes très intensifs au niveau de la mémoire, les fils d'exécution se concurrencent et cela peut avoir un impact important sur la performance. Même avec un débit entre 10 et 40 GB/s et en donnant des capacités en mémoire infinies, l'accélération due à ces changements n'est que de 10% [5]. Ces synchronisations imposent une lourde charge pour les programmeurs et les compilateurs [2]. Des propositions ont été faites pour tenter d'améliorer le mécanisme d'accès en mémoire [5, 7]. Cependant les récentes recherches comme les récents GPUs utilisent plutôt une technologie de mémoire virtuelle unifiée [4, 6, 10–19].

Mémoire virtuelle unifiée

La mémoire virtuelle unifiée permet de simplifier grandement la programmation du point de vue de l'utilisateur. Elle permet d'avoir une mémoire virtuelle unifiée entre le CPU et les autres processeurs. Cette technologie a été introduite avec l'arrivée de "CUDA Unified Memory" avec CUDA 6.X, et OpenCL 2.0 "Shared Virtual Memory". Ce mécanisme introduit une facilité pour les utilisateurs qui ne manipulent plus qu'un seul espace de mémoire virtuelle. Cependant les premiers résultats de performance montrent un surcout par rapport à un programme qui définit les appels manuellement [10]. Les deux principaux problèmes de cette mémoire unifiée sont la surinscription de pages mémoire et la latence due aux défauts de

pages [10, 11, 14, 18–20]. En effet, dans les cas où la mémoire utilisée rentre complètement dans la mémoire GPU et que le problème de surinscription de pages n'est pas présent, alors la mémoire virtuelle partagée permet de gagner en performance [14].

La surinscription de pages est présente lorsque plus de mémoire est allouée au CPU qu'au GPU. Cela cause un grand nombre de défauts de pages, à cause d'une part de la politique de remplacement de pages, et d'autre part une stratégie de prélecture trop agressive [11, 13, 19]. Pour éviter ce problème, il est possible d'utiliser une politique de remplacement de pages plus optimale, pour limiter le nombre de défauts de page en analysant la localité des accès et faisant varier la taille des pages utilisées [11, 21].

La latence due aux défauts de pages est un autre problème majeur, car les GPUs sont des processeurs qui ont une grande bande passante, mais aussi une grande latence. Donc chaque défaut de page a un gros impact sur la rapidité d'exécution d'un programme. Une technique qui tient compte de ce problème-là et du problème précédent est de choisir l'endroit où est allouée la mémoire en fonction de la sensibilité du programme. Si le programme est sensible à la latence alors on alloue directement sur le GPU pour éviter un défaut de pages sinon on alloue sur le CPU et on attend que le GPU fasse un défaut de pages [19].

Une autre technique a été développée par Wu et al. [8] pour résoudre les problèmes de performance des accès en mémoire non continue.

Communication interprocesseurs

La communication interprocesseurs est très utile pour envoyer les informations de calculs à des circuits spécialisés. Cependant, ceux-ci peuvent utiliser des technologies très spécialisées pour s'assurer de la rapidité des communications. Ces technologies nécessitent souvent de passer vers des moyens logiciels spécialisés et des librairies propriétaires. En effet, pour les puces de l'entreprise NVidia, les pilotes logiciels sont propriétaires ; pour AMD, la plupart des librairies constituant l'ensemble ROCm sont à code source ouvert, ce qui permet de mieux comprendre et de modifier ces pilotes logiciels. De plus en plus, les puces peuvent communiquer entre elles sans avoir à passer par la mémoire de l'UTC. Des technologies comme NVidia Collective Communication Library, PCIe v3, NVLink v1 et v2, permettent d'échanger des données nécessaires afin d'accélérer grandement les calculs.

D'après Young et al. [22], un système possédant quatre GPUs est ralenti de 50% en comparaison à un système idéal. Malheureusement, dû à la multiplication des périphériques dans un seul système, cette intercommunication doit pouvoir s'étendre sans mener à des goulots d'étranglement. Young et al. propose donc d'augmenter la taille du cache pour chaque GPU

afin de limiter les communications entre le CPU et les GPUs. Dans le nouveau système vendu par NVidia, DGX-1, une recherche a été faite pour mesurer l'impact d'une stratégie de communication en anneaux [23]. Cette stratégie [24] s'applique sur un réseau composé de 8 GPUs et utilise seulement la technologie NVLink pour communiquer, celle-ci offrant une latence moindre, et une meilleure bande passante. En utilisant cette technique, ils réussissent à doubler la performance de GPUs utilisant uniquement PCIe v3.

Une évaluation des diverses technologies de communication par A. Li et al. [25] montrent que l'intercommunication entre plusieurs GPUs est essentielle pour obtenir une meilleure performance sur des machines hétérogènes comme le DGX-1 ou DGX-2.

Il est aussi possible d'utiliser GPUDirect pour établir une communication entre FPGA et GPU [26]. La multiplication des cas d'utilisation qui nécessitent l'interopération entre GPUs, FPGAs et DSPs va nécessiter un standard pour établir une interface de communication où chaque processeur peut échanger des informations. Ce bus de communication commun est déjà présent dans le standard HSA [27]. De plus, de nombreuses applications aujourd'hui nécessitent l'utilisation de GPUs dans l'infonuagique. Ces cas d'utilisation ont aussi besoin de limiter les copies intermédiaires et d'accéder directement à la mémoire du GPU, par exemple en utilisant la technologie GPUDirect.

CUDA

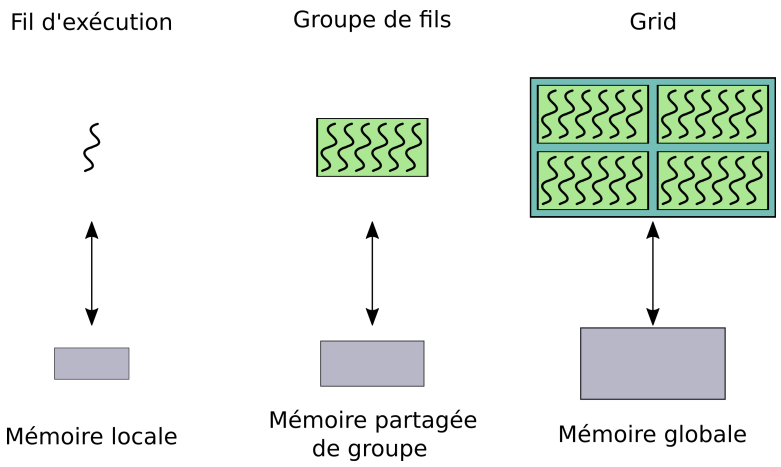


Figure 2.1 Hiérarchie de mémoire pour un GPU

CUDA est une des premières technologies à avoir existé pour effectuer du HPC sur GPU (GPGPU). Celle-ci permet de programmer directement les processeurs du GPU à travers des fonctions appelées kernels. Cette technologie utilise un mécanisme de transferts de mémoire entre UTC et GPU qui doit être explicite pour l'utilisateur. Depuis quelques années de

nouvelles technologies ont permis la création d'une mémoire unifiée qui permet une synchronisation beaucoup plus simple pour l'utilisateur. Cette mémoire unifiée utilise un mécanisme de défauts de pages pour synchroniser le CPU et les autres processeurs possédant la région de mémoire qui a été modifiée.

Pour exécuter des programmes sur un GPU, CUDA se repose sur la compilation de petites fonctions de code appelées "kernel". Il est aussi possible d'écrire du code directement avec l'utilisation de *Parallel Thread Execution (PTX)*, qui est l'architecture de jeux d'instructions de CUDA. CUDA utilise un modèle hiérarchique de mémoire qui ressemble à la hiérarchie dans la multiplication des fils d'exécution, comme on peut le voir dans la figure 2.1. En effet la mémoire partagée avec le groupe de fils d'exécution et au niveau des grids permet une plus rapide disponibilité et renforce le principe de localité [28].

OpenCL

OpenCL est un standard de calcul parallèle sur divers types de matériel existant. L'un des principaux avantages est sa compatibilité avec plusieurs plateformes. Après le développement d'un kernel sous OpenCL, l'exécution peut être sur un CPU, un GPU, un FPGA ou même un DSP [29,30]. OpenCL n'est qu'une spécification abstraite, il permet donc une plus grande portabilité. Cependant, une évaluation de la portabilité d'OpenCL montre que des bogues subsistent dans l'implémentation de chaque vendeur. De plus, des comportements différents sont adoptés par les constructeurs quand la documentation n'est pas claire.

Cette spécification, avant la version 2.0, obligeait l'utilisateur à explicitement déclarer les transferts de mémoire. Avec la dernière version de OpenCL, une plus fine granularité est possible et il est possible d'utiliser une mémoire unifiée virtuelle pour faciliter la programmation avec cette spécification.

HSA

HSA est un standard développé par la fondation HSA. Des entreprise notables ont contribué à créer cette fondation : ARM, AMD, Qualcomm et Samsung. Cependant, seulement AMD produit des coprocesseurs compatibles actuellement. Ce standard décrit comment les accélérateurs doivent communiquer avec le CPU et entre eux. Ce standard a pour but d'augmenter la programmabilité de ces accélérateurs et d'améliorer la performance des systèmes hétérogènes. Ce standard définit une mémoire virtuelle unifiée qui peut soit être de granularité grosse, ou de granularité fine, en fonction du déroulement optimal du programme. Pour envoyer des commandes aux accélérateurs, l'envoi de paquets en mode usager, AQL, est utilisé

pour permettre une certaine flexibilité, et aussi pour ne pas avoir à changer de contexte. Ceci permet un gain de performance qui a été mis en évidence par Sun et al. [31]. En comparant ROCm, l'implémentation de HSA par AMD, à l'ancien pilote graphique, ils ont montré que cette approche permettait de gagner de la performance grâce aux files d'attente utilisateur et une plus fine granularité. HSA permet aussi le *Device Side Enqueuing* qui permet à un accélérateur de lancer un kernel sans interagir avec le CPU, à la fin de l'exécution d'un autre kernel. Le support pour échanger des signaux entre le CPU et le GPU a été ajouté, permettant une communication rapide et évitant le *polling* qui est couteux en performance [32]. De plus, il permet un contrôle de l'exécution plus important, ce qui permet de nombreuses optimisations quant à l'ordonnancement [33].

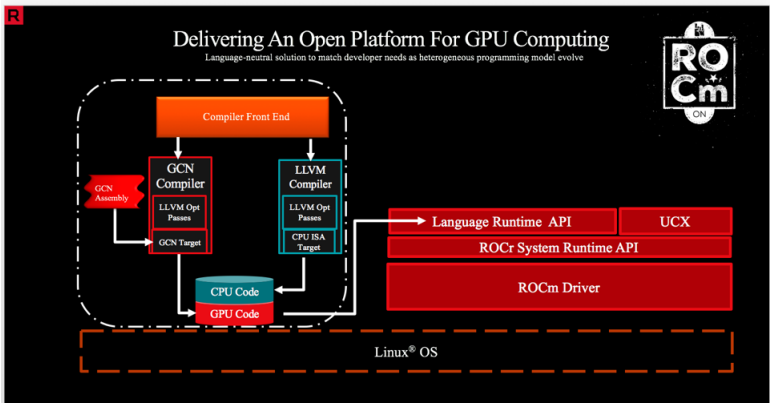


Figure 2.2 Relation des librairies dans l'environnement ROCm ©2020, AMD. Reproduit avec permission.

Ce standard est implémenté par AMD qui a développé ROCm, un ensemble de librairies dans le but d'être modulaire, décrit dans la figure 2.2. Cet ensemble permet d'utiliser des accélérateurs du vendeur AMD qui utilisent HSA. De plus cet ensemble est pour la majorité à source ouverte, ce qui permet une modification et une instrumentation des pilotes graphique plus aisée.

2.1.2 Ordonnancement des calculs et répartition des charges

Sur les systèmes hétérogènes, il est très important de tenir compte des capacités des différents coprocesseurs. En effet, ceux-ci ayant des architectures spécialisées, un programme peut avoir un temps d'exécution très différent selon le coprocesseur sur lequel il est exécuté. De plus, les communications entre le CPU et les différents coprocesseurs sont souvent la cause des principaux défauts de performance. Donc il est nécessaire d'ordonnancer correctement les charges de travail pour éviter de surcharger les transferts mémoires tout au long de l'exécution d'un

programme. Nous allons étudier les différents modèles de performance et les caractéristiques des charges de travail d'une part. D'autre part, nous allons voir comment il est possible de faire collaborer plusieurs coprocesseurs pour atteindre de meilleures performances.

Modèles de performance et Ordonnancement

Pour comprendre comment un programme est inefficace pour une architecture donnée, il faut comprendre comment celle-ci fonctionne ou, du moins, avoir une abstraction qui est réaliste. Une architecture multi SIMD, telle que celle d'un GPU, nécessite une bonne compréhension de l'architecture pour pouvoir écrire un programme efficace. Cependant, pour des standards tels qu'OpenCL qui se veulent multi plateforme, l'utilisation d'une architecture type permet d'abstraire certains détails techniques. Cette abstraction est utile pour obtenir une performance raisonnable sur différentes architectures semblables, mais de vendeurs différents. De plus, l'utilisation d'un modèle de performance est utile dans les cas d'utilisation où l'on doit prédire la performance d'un kernel pour répartir une charge de travail. Nous avons vu précédemment qu'une charge de travail trop importante pouvait donner lieu à des pertes de performance à cause des synchronisations de mémoire entre GPU et CPU.

Pour évaluer automatiquement la performance d'un programme sur un GPU, une approche analytique est moins performante qu'un modèle statistique d'apprentissage machine [34]. En effet, le modèle statistique est plus robuste aux changements d'architecture. De plus, il est difficile d'estimer certains paramètres qui sont propres aux vendeurs comme la politique d'ordonnancement des *warps* dans un *compute unit*. L'utilisation de compteurs de performance et de données du compilateur permet donc de prédire, avec une certaine erreur, la performance de fonctions kernel qui s'exécutent sur un accélérateur. D'autres modèles [35] prédisent les performances du CPU pour pouvoir améliorer la performance des systèmes hétérogènes.

L'ordonnancement permet d'améliorer grandement la performance d'un système hétérogène. En effet le temps d'exécution d'un grand nombre d'applications dépend grandement de l'ordre dans lequel ces tâches sont exécutées. De plus l'utilisation de nouvelles technologies, comme la mémoire virtuelle unifiée, influe sur la performance obtenue et doit être prise en compte au niveau d'un modèle de prédiction [36].

La plupart des modèles de prédiction proposés sont linéaires [34, 36–38] et comprennent pour la plupart majoritairement des compteurs de performance. Cependant, d'autres données statiques proviennent du compilateur, ce qui permet de prendre en compte d'autres paramètres. Ces modèles sont ensuite utilisés pour l'ordonnancement et permettent des gains entre 20 et 30% en temps d'exécution [34, 37].

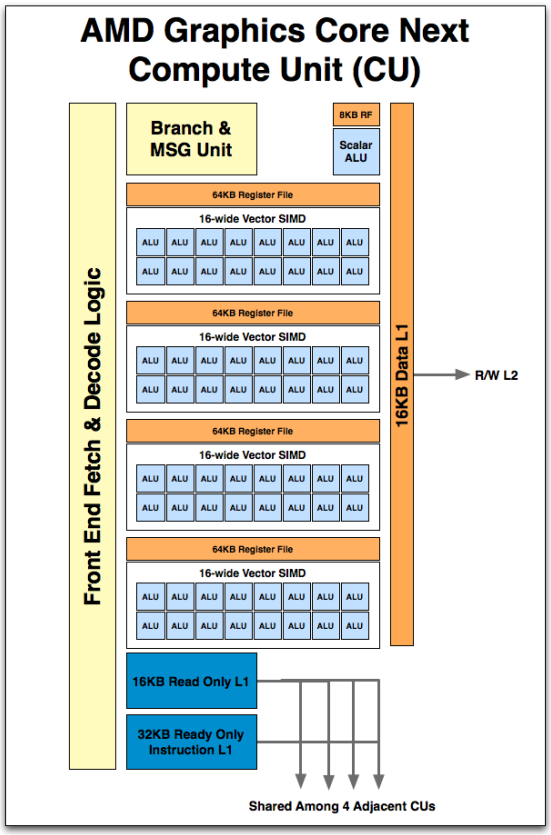


Figure 2.3 Architecture d'une unité de calcul GCN [1] sur un GPU ©2012, AMD. Reproduit avec permission.

L'architecture d'un GPU est composée de multiples *Compute Units* (selon la terminologie d'OpenCL) qui permettent d'exécuter plusieurs kernels simultanément. Chaque *Compute Unit*, a une architecture semblable à celle illustrée dans la figure 2.3. Elle est composée de plusieurs processeurs SIMD qui exécutent chacun des groupes de fils d'exécution.

Caractéristiques des charges de travail

Pour correctement évaluer les modèles de performance présentés, la plupart des recherches se basent sur des bancs d'essai qui présentent diverses charges de calculs suffisamment différentes les unes des autres pour représenter l'ensemble des programmes qui peuvent être exécutés sur un accélérateur. Une partie de ces bancs d'essai permettent d'évaluer certaines capacités matérielles de façon pratique comme la vitesse du bus, la bande passante, le nombre de calculs à virgule flottante, etc. [39, 40]. Sinon, ils utilisent des applications spécifiques de simulation physique, mathématique ou d'autres problèmes nécessitant une forte puissance de calculs [39, 41–43]. Ces bancs d'essai sont fortement utilisés dans les recherches pour évaluer

une politique d'ordonnancement, la prédiction d'un modèle, stratégie de préchargement, etc.

Ces bancs d'essai prennent en compte différentes caractéristiques de charges de travail. Sur un GPU, il est possible d'avoir un programme qui n'utilise pas toutes les ressources disponibles d'un *compute unit*. On mesure donc le taux d'occupation, qui est le pourcentage de *SIMD* utilisé par *compute unit* durant l'exécution d'un programme. De plus, à cause des branches conditionnelles, certaines instructions sont exécutées alors que la condition n'est pas valide, car tous les fils d'exécution sont synchronisés. Il y a donc une divergence. La performance d'un programme ayant une grande divergence dépend des autres kernels qui s'exécutent simultanément, à cause de l'utilisation intensive de la mémoire tampon. D'autre part, la mémoire dans un système hétérogène est très importante et a un impact non négligeable sur la performance. C'est pourquoi les bancs d'essai doivent avoir une bonne variation dans l'utilisation de la mémoire, pour s'assurer qu'une meilleure performance atteinte n'est pas due à un biais qui privilégie le résultat du banc d'essai.

Une évaluation de ces bancs d'essai [44], en les comparant entre eux, montre que certains ne couvrent qu'une faible variance dans l'espace des programmes existants. En revanche certains, comme [41], sont relativement efficaces, mais restent biaisés dans leur évaluation car la majorité des programmes ont une grande occupation. L'idéal est donc de combiner plusieurs bancs d'essai pour éviter de biaiser les résultats et d'avantager une caractéristique en particulier.

La prise en compte de ces diverses caractéristiques est essentielle pour obtenir un banc d'essai qui évalue objectivement la performance d'une technologie, d'une optimisation, ou d'une stratégie.

Mesurer la charge de travail

Pour mesurer l'occupation, la divergence et l'utilisation de la mémoire, les logiciels de profilage ont recours aux compteurs de performance. Nous avons déjà vu précédemment que ceux-ci étaient très efficaces pour prédire le temps d'exécution d'un programme. En effet, ces compteurs sont obtenus de différentes manières. L'utilisation d'un simulateur de GPU permet de définir et d'obtenir n'importe quelle sorte de compteurs. L'utilisation de compteurs matériels est souvent limitée, on ne peut pas les obtenir tous simultanément. Souvent, seulement une partie de ces compteurs sont utilisables en même temps donc, nous sommes obligés d'activer seulement une partie des compteurs et d'exécuter plusieurs fois l'application pour obtenir l'ensemble des compteurs voulus.

Une des mesures les plus utilisées pour évaluer la performance de la mémoire est le taux de

fautes de caches. En effet, dans de nombreuses études [45–47], ce compteur de performance permet d'évaluer la capacité de l'accélérateur et des pilotes à utiliser efficacement la bande passante. De plus, ce compteur est souvent disponible à différents niveaux de caches. Cette mesure est donc utilisée pour comparer différents algorithmes de remplacement des lignes de cache ou pour diriger l'optimisation d'un programme qui est limité par la bande passante.

Pour mesurer l'occupation d'un GPU, on a souvent recours au parallélisme des fils d'exécution (TLP). Cette mesure indique si le programme exécuté occupe de façon optimale toutes les ressources disponibles de l'accélérateur. On utilise le nombre moyen de warps par cycle comme métrique pour mesurer ce paramètre [48, 49]. Le TLP est notamment utilisé pour réguler la charge de travail dans un système hétérogène, car un trop grand TLP peut surcharger la mémoire et ralentir drastiquement l'exécution.

Le nombre d'instructions exécutées par cycle (IPC) est aussi utilisé pour identifier la cause d'une amélioration d'un programme. Cependant, une amélioration de l'IPC ne mène pas toujours à un plus petit temps d'exécution. Des bulles qui bloquent des instructions mènent à une réexécution de ces instructions, ce qui explique un plus grand IPC associé à un plus grand temps d'exécution. D'après Jablin et al. [50], dans le cas où un plus grand IPC est associé à une exécution plus rapide, alors l'IPC peut remplacer la TLP.

Pour prédire le temps d'exécution d'un kernel, on peut utiliser le nombre d'instructions : vectorielles, scalaires, de branches, etc. En effet, ces informations fournies par le compilateur sont très corrélées avec le temps d'exécution [51]. Cependant, les programmes s'exécutant sur le même système hétérogène ont une grande influence sur la performance des kernels. Il est donc essentiel de tenir compte de ces programmes colocalisés [51].

Répartition des charges

Nous avons vu comment les programmes sont évalués grâce à des caractéristiques de performance pour prédire leur temps d'exécution sur divers coprocesseurs. En effet, il est important de répartir les charges, dans un système hétérogène, pour s'assurer que le coprocesseur soit adapté aux tâches qu'on lui envoie. De plus, il est souvent nécessaire de fusionner ou de diviser des tâches pour mieux les répartir. Nous allons voir ici les différentes techniques qui permettent aux coprocesseurs de mieux collaborer.

Une approche classique pour faire collaborer plusieurs processeurs est de diviser les données en autant de processeurs, pour que chacun exécute une partie. Cette première approche n'est pas tout le temps applicable, mais reste une technique très performante et très utilisée dans les systèmes *MapReduce*. Cette technique a été évaluée dans le contexte d'un système

hétérogène composé d'un CPU et d'un FPGA [52]. L'étude a utilisé le banc d'essai Chai qui est composé d'algorithmes de traitement d'images. La meilleure performance a été atteinte en calculant une partie des images sur le CPU et l'autre partie sur le FPGA, et en utilisant un ordonnancement dynamique (contrôlé par un fil durant l'exécution).

Une autre stratégie est de séparer les tâches entre les différents coprocesseurs. Selon Huang et al. [52], les gains de cette stratégie sont similaires au gain du partitionnement de données. De plus, la séparation de tâches a un plus grand cout pour les transferts de mémoire. Cependant, les processeurs ont des périodes où ils sont à l'arrêt avec cette stratégie, donc une amélioration de performance est possible, et surement atteignable en évaluant le meilleur coprocesseur pour la tâche. Toutefois, ces approches ne permettent pas toujours de gagner en performance, comme l'ont souligné Zhang et al. [53]. Ils ont développé un modèle permettant de prédire si le programme peut bénéficier d'une stratégie de collaboration ou non. Il est donc important de toujours évaluer ces stratégies sur des programmes variés pour être certains des gains de performance qui ne sont pas toujours présents.

2.1.3 Gestion de la mémoire tampon

Nugteren et al. [54] montre que "comme les GPUs cachent la latence d'accès à la mémoire interne du GPU à travers l'utilisation de nombreux fils d'exécution, ils nécessitent une grande bande passante. L'utilité principale de la mémoire tampon n'est donc pas de réduire la latence d'accès à la mémoire interne du GPU, mais de réduire les transferts de mémoire entre le CPU et le GPU". Dublish et al. [55] montre en effet que la bande passante entre le CPU et le GPU limite la performance. Cependant, l'un des facteurs les plus limitants est la bande passante de la mémoire tampon L2. La latence n'est donc jamais un facteur limitant, car le nombre de fils d'exécutions permet de pallier les défauts de cache, sans impacter la performance, tant que l'application ne nécessite pas de grandes quantités de mémoire. Au cas où l'application nécessite beaucoup de mémoire, alors c'est la bande passante qui limite le temps d'exécution. Une utilisation efficace de la mémoire tampon devient alors essentielle pour améliorer le temps d'exécution.

Le principe de localité

La localité des données est le paramètre exploité par la mémoire tampon pour améliorer la performance en réduisant la latence d'accès. Il existe deux types de localités, la localité temporelle et la localité spatiale. La localité temporelle est l'utilisation des mêmes ressources dans un temps proche. La localité spatiale est l'utilisation de ressources proches ou similaires à d'autres ressources accédées récemment. Il est donc primordial d'optimiser la localité des

données dans un programme. De nombreuses techniques d'exécution ont été développées dans ce but et le principe de localité dirige les analyses de performance sur les CPUs, qui ont ensuite été adaptées pour les accélérateurs [54, 56]. Nous verrons ces techniques dans la partie concernant le profilage et le traçage.

Nous verrons comment l'ordonnancement, à plusieurs niveaux, est primordial. Ensuite, ralentir volontairement l'exécution peut limiter certaines interférences améliorant le temps d'exécution. Ne pas utiliser la mémoire tampon dans certains cas peut être bénéfique. Enfin, en raison de la spécificité des accélérateurs, les algorithmes de remplacement de ligne de cache utilisés traditionnellement sur les CPU peuvent être modifiés pour être plus optimaux sur les accélérateurs.

Ordonnancement des kernels

L'exécution de code sur un accélérateur se fait par fonction kernel. Ces kernels sont ordonnancés grâce à une file d'attente dans la plupart des cas. Cette file d'attente étant remplie par l'hôte. Il est souvent possible sur des accélérateurs d'exécuter simultanément plusieurs kernels. L'exécution peut aussi être répartie de façon à minimiser le temps d'exécution. On peut diviser chaque kernel en plusieurs sous-kernel. Un test [57] avec Rodinia [41] montre qu'en fonction du kernel, la performance varie grandement entre une exécution sérielle, une exécution simultanée, un découpage variable et un découpage statique. L'article conclut que le découpage pour améliorer la performance est nécessaire. L'ordonnancement et la division de kernels pour améliorer la performance est donc primordiale.

Pour ordonnancer correctement les kernels, l'état de l'art est de posséder une étape de profilage pour évaluer la performance et l'impact qu'ont chaque kernel [57, 58]. Il est aussi possible de mesurer le nombre de certains types d'instruction pour estimer certains paramètres [59]. Suite à cette étape, grâce à l'utilisation d'un modèle de prédiction, on peut déterminer quelles combinaisons de kernels sont optimales. Après l'obtention des différentes combinaisons à exécuter simultanément, il faut déterminer comment diviser la charge de travail de manière à occuper toutes les unités de calcul de l'accélérateur [57]. Ces divisions tiennent compte des données de profilage, combinées aux données spécifiques à l'architecture utilisée.

Ordonnancement des *warps*

Dans l'architecture du constructeur Nvidia un *warp*, ou *wavefront* dans l'architecture du constructeur AMD, est un ensemble de fils d'exécution sur un processeur graphique. Cet ensemble représente un certain nombre de calculs et donc de cycles d'exécution. Par exemple

un *wavefront* s'exécute en quatre cycles sur l'architecture GCN d'AMD.

Ces blocs de calculs sont ordonnancés au niveau des multiprocesseurs de flux. Ces *warps* sont ordonnancés en deux étapes : l'attente du chargement des instructions et des données, et la priorisation. À l'étape de la priorisation, plusieurs ordonnanceurs (dépendant de l'architecture) priorisent différents *warps* en utilisant un algorithme comme le round-robin [60]. Ensuite un seul *warp* est sélectionné et exécuté en quatre cycles, que ce soit sur l'architecture AMD ou Nvidia.

Les accès mémoire pour préparer l'exécution des *warps* peuvent mener à une saturation des MSHR, ce qui empêche les autres warps d'accéder à la mémoire tampon [60,61]. La méthode OAWS [60] effectue une prédiction du nombre de MSHRs utilisés par un warp et ordonne les *warps* de façon à maximiser l'utilisation de ces registres, sans les saturer, tout en estimant le nombre optimal de *warps* simultanés. La méthode Mascar [61] sélectionne un *warp* et lui donne la priorité pour ses requêtes mémoire de manière à garantir l'exécution complète de ce *warp*. Celui-ci pourra donc être exécuté sans ralentir grandement l'exécution, car tous les *warps* en cours d'exécution saturent les registres et donc attendent chacun leur requête mémoire. Le ralentissement de l'exécution est donc parfois bénéfique au temps d'exécution, du fait des interférences entre chaque *warp* au niveau de la mémoire tampon.

Une autre approche est de tirer profit de l'hétérogénéité des *warps* [62]. Le système Me-DiC classifie les *warps* en différents types. Ensuite, l'ordonnancement des requêtes mémoire pour préparer les *warps* à l'exécution, et l'utilisation du cache, dépendent du type. D'autres approches ne sont pas détaillées ici, mais ont fait l'objet d'un article : PRO [63], DAWS [64].

Contournement de la mémoire tampon

Certains des *warps* n'utilisent pas efficacement la mémoire tampon. Pour ceux-ci, Ausavarungnirun et al. [62] proposent de contourner l'utilisation de la mémoire tampon. La mémoire tampon n'améliorant pas beaucoup l'exécution de ces *warps*, cela permettra de réduire les conflits au niveau du cache.

Dans d'autres cas, il est nécessaire d'ignorer la mémoire tampon et d'envoyer directement le résultat d'une seule requête mémoire. En effet, si l'on décide d'utiliser le cache, celui-ci devra libérer d'autres variables qui, à leur tour, vont entrainer un défaut de cache quand elles seront utilisées à nouveau. Ce cas arrive régulièrement dans les applications nécessitant beaucoup d'appels mémoire. Un des moyens pour éviter de surcharger la mémoire tampon, appelé prédiction de la distance de protection (PDP), est de protéger chaque ligne du cache avec un compteur. Ce compteur est décrémenté à chaque accès. Lorsque le compteur arrive

à zéro, alors la ligne peut être remplacée. Cette technique existe depuis longtemps, mais est toujours utilisée pour améliorer l'utilisation du cache dans des programmes et soulager l'emballement de la mémoire tampon [65–68].

Indexation adaptative de la mémoire tampon

Une autre façon d'optimiser l'utilisation du cache est d'adapter la manière d'indexer différemment celui-ci. Cette approche étend de précédents travaux qui ont été effectués sur les CPUs [69]. Nous avions vu précédemment que le conflit est une des trois causes de défauts de cache. En modifiant l'indexation, il est possible de mieux répartir l'utilisation de la mémoire tampon sur les processeurs graphiques [66]. Cette technique a récemment été utilisée dans un contexte de calcul générique, mais Kim et al. [70] avait obtenu une réduction dans le nombre de défauts de cache pour une application utilisant des processeurs graphiques.

En effet cet article combine l'utilisation d'une indexation adaptative avec le contournement de la mémoire tampon et la limitation de *warps* pour fournir la meilleure performance possible. Une autre approche est d'utiliser un cache complètement associatif, ce qui permet de réduire dramatiquement les conflits [71]. Cette approche reste cependant étudiée grâce à un simulateur et n'est pas encore implémentée.

Algorithme de remplacement de lignes de cache

Nous avons vu précédemment qu'une indexation avait un impact significatif sur la performance et le nombre de défauts de cache. L'utilisation d'un algorithme de remplacement de lignes de cache spécifique aux accélérateurs et à leur architecture est aussi une très bonne façon d'optimiser la performance.

Les CPUs Intel aujourd'hui utilisent un algorithme Pseudo-LRU (PLRU) pour les caches L1 et L2 [72], mais ce type d'algorithme n'est pas forcément applicable directement aux accélérateurs. En effet, dans certains cas, MRU est plus performant, et donc en fonction du type de warp, on privilégiera un algorithme plutôt qu'un autre [62]. Il a déjà été montré qu'en utilisant uniquement LRU, la performance était loin d'être optimale, et qu'en utilisant une stratégie qui maximise certains paramètres comme le nombre d'instructions par cycle, ou alors qui minimise le nombre de défauts par cycle, alors on obtenait des résultats bien plus proches de l'optimal [73].

2.2 Traçage, profilage et débogage de systèmes hétérogènes

On a vu précédemment des techniques pour améliorer différents aspects de la performance des systèmes hétérogènes. Le profilage de systèmes hétérogènes consiste à évaluer certains paramètres pour pouvoir mesurer la performance. On peut résumer ces problèmes en quatre catégories [74] : l'efficacité d'exécution du kernel sur un accélérateur, les interactions entre l'hôte et l'accélérateur, les interférences entre plusieurs exécutions sur un accélérateur et les communications entre accélérateurs. Il existe trois approches pour prendre des mesures : le traçage, le profilage et le débogage. Les trois approches nécessitent une analyse pour pouvoir être compréhensibles pour l'utilisateur. Nous allons donc étudier chacune de ces approches de façon détaillée dans cette partie.

2.2.1 Traçage et analyse

Le traçage est souvent très couteux et donc il est nécessaire de limiter le traçage horizontalement (quelle partie du programme est instrumentée) et verticalement (dans quelle fonction instrumente-t-on ?) [75].

LTTng

LTTng (*Linux Trace Toolkit next-generation*), est sorti initialement en 2005. Le projet a été créé par Mathieu Desnoyers pour permettre le traçage sur le système d'exploitation Linux. C'est une trousse d'outils permettant le traçage à travers l'utilisation de point de traçage, de *kprobes* et d'informations de profilage. En effet, les points de traçage sont des éléments de code permettant l'exécution de code à certains endroits précis. Le code s'exécutant est dénommé *kprobe* pour *kernel probe*. Il s'agit en effet de permettre à l'utilisateur de sonder quand est-ce que le kernel exécute certaines fonctionnalités et de pouvoir donc mieux comprendre son fonctionnement, étudier sa performance et comment il interagit avec les programmes utilisateurs.

LTTng se sert donc de ces points de traçage pour instrumenter le kernel Linux avec ses propres fonctions pour enregistrer les différents évènements. En effet, l'enregistrement des informations de trace peut être complexe dû à la fréquence avec laquelle ceux-ci sont produits. Des millions d'évènements ont lieu chaque seconde et il faut les enregistrer sans impacter fortement la performance, dans un format qui reste utilisable. D'autres librairies existent pour réaliser la même tâche : SystemTap [76], DTrace [77], ftrace [78]. Cependant, chaque librairie possède sa propre spécificité et ses propres fonctionnalités.

La particularité de LTTng est qu'il n'utilise aucun verrou pour supporter plusieurs fils d'exé-

cution simultanés. L'utilisation de verrous dans un système parallèle est souvent cause de nombreux problèmes de performance dus aux différents changements de contexte et au maintien de la cohérence de la mémoire tampon. Les algorithmes qui permettent l'exécution parallèle sans verrous sont donc très performants. Cela permet à LTTng d'être un outil de traçage qui possède un des surcouts les plus faibles [79] pour le traçage du noyau Linux.

LTTng possède aussi la capacité de tracer en mode utilisateur pour enregistrer des traces de programmes à l'aide de différents mécanismes. L'instrumentation est soit effectuée par l'utilisateur du programme, soit injectée automatiquement par le compilateur (typiquement aux débuts et fins de fonctions), soit déjà incluse dans une librairie utilisée. Les informations de trace sont envoyées à un *daemon*, qui les enregistre dans un fichier de trace.

Le format utilisé par LTTng est le format CTF (*Common Trace Format*). C'est un format de stockage binaire qui permet le stockage efficace d'une trace. En effet le format étant séquentiel et possédant plusieurs canaux, plusieurs fils d'exécution enregistrent simultanément les évènements. De plus, un fichier de méta données enregistre les informations nécessaires pour la lecture et l'analyse de la trace. Ce format est aussi utilisé pour de nombreuses analyses sur l'outil de visualisation Trace Compass.

CLUST

Couturier et Dagenais [80] ont présenté un outil appelé LTTng CLUST qui permet de tracer l'exécution d'un système hétérogène fonctionnant avec OpenCL. En effet, la librairie OpenCL a été instrumentée à l'aide de LTTng pour fournir cette fonctionnalité. De plus, comme LTTng a la capacité de tracer le noyau Linux, cet outil permet d'observer à la fois l'exécution dans le contexte du noyau et dans le contexte du processus exécutant l'application.

On peut donc observer les commandes asynchrones envoyées à travers l'interface d'OpenCL synchronisées avec le temps du CPU. Cet outil utilise Trace Compass pour visualiser le fonctionnement de l'application dans sa globalité. Le surcout de cette approche est significatif, mais augmente peu avec l'augmentation de la charge de travail. Une des limitations de cet outil est l'absence de données de profilage. En effet, la seule donnée existante est la durée d'exécution du kernel. L'outil ne permet pas donc facilement d'améliorer le temps d'exécution des kernels sur les accélérateurs, ce qui est un enjeu important du calcul de haute performance.

LTTng-HSA

Margheritta et Dagenais [81] ont présenté un outil appelé LTTng-HSA. Cet outil permet de tracer l'exécution d'une application sur un système hétérogène sur la plateforme ROCm.

En effet, il se sert des capacités d'interfaçage de ROCm pour insérer des points de trace à l'aide de LTTng. De plus, il se sert de la librairie GPA pour collecter les compteurs de performance liés à l'exécution. Dû aux limites de registres sur le processeur graphique utilisé, il est possible de prendre plusieurs mesures pour pallier ce problème. De plus, les évènements étant asynchrones, il est nécessaire de trier la trace obtenue pour les réordonner.

Plusieurs vues ont ensuite été développées sur Trace Compass pour visualiser les différentes files d'attente créées, et leur état durant l'exécution du programme. Cet outil permet donc l'obtention de données de profilage, en plus des données de traçage, mais n'a pas de capacités d'analyse de ces données. De plus, les données de traçage ne sont pas synchronisées avec le traçage du noyau, ce qui aurait permis une analyse plus complète du système. Dû à l'évolution de ROCm, cet outil n'est plus à jour, car l'interface utilisée a évolué. Des tests avec des bancs d'essai connus comme Rodinia [41] ou Hetero-mark [42] auraient permis d'évaluer plus en détail les capacités de cet outil.

Dyninst

Dyninst [82] est une librairie développée en C++ qui permet de manipuler un programme après sa compilation. En effet, cela permet à un utilisateur de cette librairie d'instrumenter dynamiquement son programme durant son exécution. L'utilisateur a à sa disposition une interface de protocole d'application. Grâce à cette librairie, il est donc possible d'instrumenter et de tracer, profiler ou déboguer une application à l'exécution. Cet outil n'est pas un outil de traçage, mais sert à créer des outils de traçage ou de profilage [83, 84].

Pour insérer dynamiquement du code au niveau d'un programme en exécution, Dyninst se sert des mêmes mécanismes qu'un débogueur pour localiser et accéder au processus. Ensuite, pour générer le code, il transforme le code à insérer en langage machine et il utilise un trampoline pour exécuter ce code. Un trampoline est une technique permettant d'insérer une longue section de code à l'intérieur sans interagir avec l'état du programme. En effet, une instruction de saut est utilisée pour exécuter un "trampoline" qui va ensuite à son tour sauter vers un "mini-trampoline" qui va enregistrer l'état du programme et le restaurer après l'exécution du code à insérer. Ensuite ce "mini-trampoline" va aller exécuter le code à insérer. De cette façon, le code du programme est très peu modifié et, si l'on choisit le point d'insertion correctement, la logique n'est pas impactée.

Instrumentation du kernel

Le traçage d'une application au niveau de l'hôte permet d'obtenir une vue d'ensemble du fonctionnement de l'application. Il permet de diagnostiquer les problèmes de performance et d'étudier son comportement. Cependant, dans un système hétérogène, il est nécessaire, dans certains cas d'instrumenter le code au niveau de l'accélérateur.

Il existe deux approches : instrumenter à l'aide d'un simulateur et instrumenter le langage machine de l'accélérateur. Le premier permet une plus grande liberté et souplesse pour instrumenter, car le simulateur permet d'accéder à tous les composants du processeur. Cependant, la précision n'est pas toujours bonne et dépend du simulateur et de l'architecture utilisée. En outre, l'exécution peut être très longue et rendre l'expérimentation difficile. Le second peut perturber les mesures à cause des effets de bord de l'instrumentation. De plus, le surcout dû à l'instrumentation dépend beaucoup du kernel [85].

Lynx

Le système Lynx, développé par Farooqui et al. [85], permet d'instrumenter dynamiquement un kernel en insérant des instructions au niveau du jeu d'instructions PTX. Ce système se base sur un outil précédent, Ocelot [86], qui permet de transformer un kernel en y ajoutant des instructions PTX. Ces instructions fonctionnent donc uniquement avec les accélérateurs graphiques Nvidia, car AMD utilise un jeu d'instruction différent. Cependant, il a été adapté plus récemment pour être compatible avec les accélérateurs d'AMD [87]. Ocelot modifie le code du programme après sa compilation pour l'instrumenter, ce qui permet d'avoir des capacités similaires à Dyninst pour un kernel qui s'exécute sur un accélérateur.

Lynx permet d'ajouter des instructions en langage C à l'intérieur du kernel compilé en PTX. De plus, les appels à *cudaMalloc*, *cudaMemcpy*, *cudaFree* ont été instrumentés pour permettre à l'accélérateur de copier les données de traçage vers l'hôte pour enregistrer une trace. Lynx offre la possibilité de tracer par fil d'exécution, par groupe de fils, *warps* et SM. Cela permet de configurer facilement la façon dont un programme est tracé ou profilé, et de réduire le surcout si celui est trop grand. Il supporte aussi l'instrumentation définie par l'utilisateur, alors que les outils donnés par Nvidia ne permettent que le choix d'un certain nombre de compteurs de performance à activer. Des outils ont été développés avec Lynx pour permettre d'optimiser des applications durant l'exécution, en utilisant les informations de profilage [88].

Trace Compass

Trace Compass est un outil de visualisation de trace à source ouverte développé en Java et basé sur l'infrastructure Eclipse. Il permet d'effectuer un certain nombre d'analyses à partir des informations de trace. Ces évènements pour les traces noyaux comportent des informations spécifiques à propos du système. La plus importante est l'estampille de temps. Celle-ci permet de dériver l'état de chaque processus et chaque fil d'exécution grâce à leur numéro d'identification. D'autre part, des informations spécifiques à chaque point de trace permettent de définir un état pour le processus. Il est aussi possible de définir des dépendances entre chaque processus, d'une part en sachant sur quel processeur le processus est en train d'être exécuté. D'autre part, en analysant les causes de la mise en attente des processus, il est possible de construire le chemin critique de n'importe quel processus en exécution pendant l'enregistrement de la trace [89].

Trace Compass fonctionne grâce à une machine à états qui parcourt la trace en ordre chronologique et qui attribue à chaque fil d'exécution un état. Au fur et à mesure que les évènements sont traités, chaque fil d'exécution va changer d'état en fonction des évènements traités. Le résultat est présenté sous forme d'une frise avec pour chaque ligne un fil d'exécution ou un processus.

Trace Compass permet l'analyse de n'importe quelle trace, tant que l'on spécifie les objets dont on souhaite traquer l'état et comment l'état est modifié par les évènements. De nombreuses analyses existent aujourd'hui pour différentes applications : analyser la pile d'appels en fonction du temps, l'ouverture et la fermeture des fichiers par chaque processus en fonction du temps, le *flamegraph* d'une application, la consommation de mémoire en fonction du temps, etc.

Trace Compass étant construit avec Eclipse, il hérite de son interface utilisateur. Les interfaces utilisateur évoluant maintenant très rapidement, un nouveau projet a été créé par la fondation Eclipse : Theia. Ce projet a pour but d'avoir un éditeur de code avec un découplage entre l'interface et le coeur de l'application. TraceCompass a été modifié pour supporter ce découplage et il est possible d'exécuter un serveur TraceCompass et d'analyser des traces sur un serveur distant. Un module implémentant une interface pour Trace Compass sur Theia est donc en développement pour avoir une interface plus moderne, et plus facilement modifiable.

L'analyse de distance de réutilisation

L'analyse de distance de réutilisation (RDA) est un modèle populaire [47, 54, 56, 90, 91] qui permet de mesurer l'efficacité d'un programme à tirer profit de la mémoire tampon. Initiale-

ment pour une architecture de CPU, ce modèle a été adapté pour une architecture parallèle. Dans ce cas-ci, l'ordre d'exécution est difficile à connaitre, car de nombreux fils d'exécution s'exécutent de façon concurrente et l'algorithme d'ordonnancement des warps n'est pas toujours connu. De plus, l'algorithme de remplacement des lignes de cache n'est pas toujours le même pour les architectures parallèles.

Les défauts de cache ont trois causes : le conflit, la capacité et l'obligation [92]. Le conflit survient lorsque la place est suffisante en mémoire tampon, mais différentes régions de la mémoire utilisées sont alignées. Ces régions de mémoire seront alors en conflit à cause de l'utilisation des mêmes lignes de cache (*n-way set associative cache*). Un défaut de cache peut survenir à cause d'une capacité limitée et un défaut de cache initial est obligatoire pour charger une valeur dans la mémoire cache.

Nous avons vu précédemment que l'utilisation du cache est primordiale pour limiter les transferts entre l'accélérateur et le CPU. En effet, c'est la principale source de ralentissement et ces transferts limitent la performance sur les systèmes hétérogènes [46,47,90,93]. Dans l'analyse de distance de réutilisation, seulement les défauts de cache obligatoires et ceux dus à la limite de capacité sont pris en compte, car la méthode RDA est basée sur un cache complètement associatif. La distance de réutilisation correspond au nombre d'adresses différentes accédées entre deux accès à la même case mémoire. Cette distance est utile pour profiler un programme et obtenir une mesure de l'efficacité de l'utilisation de la mémoire tampon. Des approches différentes ont été prises pour prendre en compte divers paramètres : l'associativité du cache [47], les registres permettant de traquer les diverses requêtes en mémoire après un défaut de cache [47, 54, 94], et l'exécution simultanée de kernels [90]. Aucune approche cependant ne combine tous ces paramètres. De plus, certains de ces paramètres sont très dépendants de l'architecture, et donc d'un vendeur à un autre, d'une génération à une autre, ils sont susceptibles de changer.

Analyse de l'empreinte mémoire

L'analyse de l'empreinte mémoire permet de quantifier la mémoire utilisée par un programme. D'après Kiani et al. [91], cette métrique permet de mesurer la quantité de données réutilisées, ce qui permet de connaitre l'efficacité de la mémoire tampon. L'empreinte mémoire est calculée à partir d'une trace de chaque accès. On calcule l'usage moyen à partir d'une fenêtre dynamique qui permet d'obtenir un ensemble d'accès. À partir de cet ensemble d'accès on obtient alors l'usage moyen en un temps donné. Un défaut majeur de cette analyse est que l'algorithme naïf pour la produire est de complexité $O(n^3)$ [95].

Cette analyse permet d'obtenir une analyse plus compréhensive qu'en utilisant l'analyse de

distance de réutilisation. De plus, elle est plus facile à accélérer en exploitant le parallélisme d'un accélérateur. C'est donc une approche qui peut être utilisée pour estimer la performance du cache dans le contexte d'un accélérateur GPU, FPGA ou DSP.

2.2.2 Profilage sur processeurs distants

Le profilage se différencie du traçage par l'utilisation de mesures de performance agrégées. En effet, le traçage recueille un ensemble d'évènements lors de l'exécution d'un programme. Le profilage ne garde que l'agrégation de ces évènements. Le profilage est donc plus utilisé dans des applications de calculs de haute performance, car l'impact sur la performance est moindre. Dans cette partie nous allons voir quels types d'information nous pouvons récupérer grâce au profilage, et diverses façons d'analyser ces données.

Compteurs de performance

Les compteurs de performance sont des registres matériels qui évaluent le nombre d'un type d'évènements qui est lié à un certain type d'évènements au niveau du matériel. Il peut s'agir de défauts de cache, d'instructions exécutées, du nombre de pages chargées, d'accès à la mémoire, etc. Le nombre de compteurs disponibles est généralement limité et il est donc possible de configurer le type d'évènements à compter. Le nombre d'évènements est donc exact. En revanche, la façon de récupérer la valeur du registre influe significativement sur le résultat, notamment dû aux interférences liées à la synchronisation entre les accélérateurs et l'hôte et aux différents programmes qui s'exécutent simultanément.

Il diffère du profilage logiciel, car il ne nécessite aucune modification du code source. En revanche, le nombre limité de registres oblige parfois l'utilisateur à effectuer ses mesures sur plusieurs répétitions de l'exécution.

Profilage basé sur des évènements

Similaire aux compteurs de performance, le profilage basé sur des évènements permet de compter un certain type d'évènements logiciels (un appel de fonction, une allocation mémoire, un changement de contexte) grâce à une instrumentation du code. Les mécanismes pour instrumenter le code source sont similaires aux techniques d'instrumentation pour le traçage. Le profilage est souvent plus léger et moins couteux que le traçage car, à chaque évènement, le compteur est seulement incrémenté, alors qu'avec un évènement de trace, le temps, le numéro d'identification du processus, du fil d'exécution et d'autres informations sont collectés. La collecte de ces informations nécessite beaucoup de ressources et de nombreux changements

de contexte.

Pour minimiser l'utilisation de ces ressources, Mahlke et al. [96] instrumente dynamiquement l'exécutable. L'outil DynamoRIO [97] a été utilisé pour réaliser cette opération. L'avantage d'utiliser cette technique est qu'elle permet d'avoir accès à l'ensemble du code et les librairies qui y sont reliées. La spécificité de leur approche est qu'ils n'instrumentent pas l'ensemble du code, mais seulement certaines régions. Le programme est surveillé grâce aux compteurs de performance et, selon les critères définis par l'utilisateur, la région du code qui est en train d'être exécutée est dynamiquement instrumentée. L'exécution est redirigée vers une copie du binaire et l'instrumentation est faite en insérant les instructions nécessaires, directement dans une cette copie à l'exécution.

Profilage statistique

Le profilage statistique évalue l'état moyen d'un système en l'arrêtant de nombreuses fois et en évaluant l'état à chaque arrêt. L'échantillonnage de l'état du système permet d'évaluer en moyenne le temps passé dans chaque état. En d'autres termes, il est possible d'évaluer en moyenne le temps passé dans un appel de fonction si l'on arrête l'exécution assez souvent.

Le principal problème de cette approche est que si l'on échantillonne trop souvent, l'exécution du programme va être fortement impactée et les mesures, faussées. Si l'on ne mesure pas assez souvent, alors les mesures ne seront pas assez représentatives statistiquement et peuvent ne pas être précises. Il existe donc un équilibre qu'il faut obtenir pour obtenir de bonnes mesures. Cet équilibre est malheureusement fortement dépendant de l'application qu'il faut profiler [84].

L'outil développé par Stoker et al. [84] utilise la librairie Dyninst pour instrumenter dynamiquement une application. Cet outil utilise une approche mathématique pour déterminer quelle fréquence maximale d'échantillonnage permet d'obtenir un surcout minimal. En effet, les perturbations dues au profilage ou au traçage impactent la précision du résultat. Dans certains cas, ils peuvent aussi rendre l'exécution trop lente et donc rendre les expérimentations impraticables.

De nombreuses approches existent et combinent compteurs de performance, profilage d'évènements, profilage statistique et traçage pour minimiser l'impact sur la performance, tout en ayant le plus de détails possible pour aider l'utilisateur à mieux comprendre le fonctionnement du programme et à mieux l'optimiser.

Modèle roofline

Cette technique de modélisation permet de visualiser où le programme testé se place par rapport aux limites théoriques d'une microarchitecture. En effet, l'exécution d'un programme est limitée par le nombre de calculs par seconde et par les transferts de mémoire. Le *roofline model* possède deux plans : le plan de la mémoire et le plan de calcul qui se sectionnent en une ligne. Le programme testé est représenté par un point. On peut déterminer simplement s'il est limité par la mémoire ou le calcul en regardant s'il est à droite ou à gauche respectivement du point pivot. Dans la Figure 2.4, le programme P_2 est limité par la bande passante, alors que les programmes P_1 et P_3 sont limités par la performance du processeur. Deux modèles existent : *Original Roofline Model (ORM)* [98] *Cache-aware Roofline Model (CARM)* [99–102]. En effet ce deuxième type prend en compte les différentes vitesses de transfert pour les différentes hiérarchies de mémoire tampon. Cela permet d'analyser la capacité de notre programme d'utiliser efficacement la mémoire tampon.

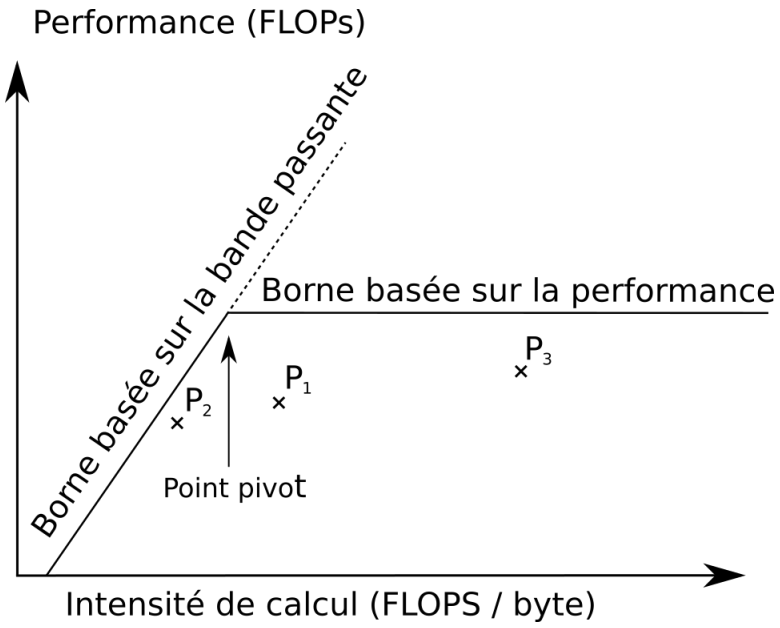


Figure 2.4 Modèle roofline

Ces modèles sont efficaces pour savoir si l'on utilise au mieux notre matériel. Cependant, ils n'indiquent pas la cause du manque d'efficacité. Si un programme est inefficace à cause de la mémoire, la cause peut être multiple : un mauvais accès qui ne prend pas en compte la localité, un algorithme peu efficace, une mauvaise gestion du nombre de fils d'exécution, ou une contention au niveau du CPU. De plus, il n'indique pas au constructeur du circuit intégré pourquoi la puce est inefficace sur ce programme.

Top-Down Microarchitecture Analysis Method (TMAM)

Cette méthode est décrite dans le manuel d'optimisation d'Intel [103, 104]. Cette méthode utilise les compteurs de performance pour caractériser les interactions entre le programme et la microarchitecture. Elle permet d'analyser les problèmes de performance des architectures à exécutions dans le désordre. Cette méthode permet de détecter des problèmes de performance sans avoir à connaitre les détails de la microarchitecture et permet de concentrer les efforts sur les problèmes importants.

Cette méthode se base sur une hiérarchie des problèmes potentiels d'une microarchitecture. Au premier niveau, il existe quatre catégories principales : *front-end bound, back-end bound bad speculation, retiring*. En passant à travers les niveaux, on répond à un arbre décisionnel qui nous permet de détecter une borne de la performance.

CodeXL

CodeXL [105] est un outil développé par **AMD**. C'est un outil qui permet de réaliser à la fois du traçage et du profilage. Il a l'avantage d'être à source ouverte, ce qui permet d'analyser son fonctionnement. Il utilise une librairie qui est injectée à l'exécution dans l'interface du protocole d'application utilisée par le programme qu'on souhaite profiler. Ceci a l'avantage d'être un mécanisme simple. En revanche, cette approche reste lente, même si l'impact est faible, car les appels à l'interface ne sont pas trop fréquents.

Cet outil utilise un format texte pour enregistrer les données de trace, ce qui n'est pas performant en plus d'occuper beaucoup d'espace. De plus, cet outil se concentre sur les données en provenance de l'environnement d'application d'accès au GPU et ne tient pas en compte les autres couches comme l'interaction générale avec le système d'exploitation. Il était très utilisé dans les anciennes versions de l'écosystème ROCm. L'évolution rapide de cet environnement a rendu le logiciel désuet, car il n'est pas mis à jour assez régulièrement. Il a depuis été marqué obsolète par AMD. Par le passé, il était possible de déboguer un programme pour processeur graphique en utilisant deux puces graphiques, l'une pour l'affichage, l'autre pour l'exécution du programme.

Nsight

Nsight [106] est l'équivalent de CodeXL, mais développé par **Nvidia**. Il s'agit d'un outil graphique permettant de visualiser l'exécution d'un programme qui exploite les accélérateurs graphiques du même vendeur. Cet outil a été créé en fusionnant NVProf [107] à Nsight. Il possède donc un environnement de développement pour les applications utilisant les capacités

de calcul générique, ainsi que pour les fonctions de calcul graphique. Il n'est pas à code ouvert. Conséquemment, il est difficile de connaitre son fonctionnement. Il est utilisé dans de nombreuses études [91] pour comparer différentes analyses aux données obtenues par cet outil.

Il permet d'obtenir des frises chronologiques d'exécution des différents kernels, les transferts de mémoire ainsi que les appels du programme à l'interface de programmation. Il permet ainsi de voir, grâce aux frises chronologiques, la synchronisation de différentes files d'attente, les barrières de synchronisations et l'état des compteurs de performance à un moment précis de l'exécution.

Il est aussi possible d'inspecter des rapports d'exécution après un plantage. On peut inspecter l'état du processeur graphique au moment du plantage, par exemple si un défaut de page a déclenché le plantage, l'information à propos de ce défaut de page et les *warps* qui étaient actifs à ce moment. Toutes ces informations permettent d'aider l'utilisateur à inspecter l'état du programme pour pouvoir déboguer son application.

PAPI

PAPI [108] est un projet qui vise à établir une interface de protocole d'application pour accéder les compteurs de performance de processeurs. Ces compteurs de performance sont utilisés pour diverses analyses que nous avons vues précédemment : modèle roofline, la méthode d'analyse descendante de microarchitecture, mesurer les défauts de cache et estimer la durée de certaines opérations. Cette interface est donc utilisée par chaque vendeur pour fournir à l'utilisateur, et à d'autres outils comme Tau, les compteurs de performance. Chez Nvidia, CUPTI [109] (*CUDA Profiling Tools Interface*) se charge d'effectuer cette opération ; chez AMD, un composant de ROCm utilise la librairie *ROCProfiler* pour fournir les compteurs de performance.

Un utilitaire appelé *papi_native_avail* se charge de lister les compteurs qui sont disponibles sur les différentes plateformes compatibles. Ensuite le composant propre au constructeur exécute le programme et envoie les compteurs de performance de l'exécution à une autre application compatible avec PAPI [74,108,110] comme TAU [111,112], ou HPCToolkit [113].

TAU

TAU [111, 112] (*Tuning and Analysis Utilities*) est une collection d'outils d'instrumentalisation, de mesure et d'analyse, qui permettent d'étudier la performance d'une application de calcul de haute performance [112]. Ces outils fonctionnent avec divers standards comme

PAPI pour visualiser les données générées.

À l'origine, les outils étaient orientés vers des systèmes de calcul haute performance homogènes fonctionnant grâce à des technologies de calcul sur CPU : MPI, OpenMP [111, 112]. Ses outils sont organisés en trois couches. L'instrumentalisation, la première, est interfacée avec la couche de mesure en utilisant le *TAU measurement API*. Deux types de mesure sont possibles : le traçage ou le profilage. Ensuite, les outils ParaProf et PerfDMF sont utilisés pour visualiser les données de profilage, et Vampir, Jumpshot et Paraver sont utilisés pour la visualisation de trace [112].

Plus récemment, des outils ont été développés pour supporter les processeurs graphiques [114], mais ces outils sont adaptables aussi pour d'autres types d'accélérateurs. Plusieurs méthodes existent pour effectuer une mesure de compteurs sur un processeur graphique. La première est une méthode de mesure synchrone des compteurs de performance : le CPU mesure les compteurs avant et après l'exécution d'un kernel. Ce type de mesure est cependant peu flexible et peu précise pour plusieurs raisons : l'hôte peut être bloqué et donc le CPU peut obtenir les compteurs avec du retard ; plusieurs kernels peuvent être en exécution en même temps et donc les interférences faussent le résultat. Une meilleure méthode est d'utiliser la méthode d'évènements de file d'attente. Le CPU prend la mesure, mais les ordres sont envoyés dans une file d'attente. Les mesures sont donc prises immédiatement avant et après l'exécution. Le principal défaut est qu'il repose sur le fait qu'un mécanisme d'exécution en file d'attente existe, ce qui est le cas pour ROCm, CUDA et OpenCL. Cependant, des restrictions s'appliquent à quels types de paquets il est possible d'envoyer. La troisième méthode consiste à utiliser un mécanisme de rappel. Après l'exécution d'un kernel, le GPU déclenche la procédure de rappel, ce qui permet des mesures très proches de la réalité et donc une estimation précise.

Vampir

Vampir [115] est une suite d'outils permettant d'instrumenter, d'enregistrer des compteurs de performance, de profiler certains évènements définis par l'utilisateur, de produire des traces et d'analyser les profils et les traces obtenus.

L'instrumentation peut se faire à trois niveaux : l'instrumentation automatique du compilateur, l'utilisation d'un outil insérant des points de trace de source à source [116], l'utilisation d'une librairie déjà instrumentée, ou l'instrumentation manuelle du code [117].

Cet outil utilise les formats OTF, OTF2 (*Open Trace Format*), EPILOG (*ELF*) et VTF (*Vampir Trace Format*) pour enregistrer les traces. Ensuite, la visualisation est réalisée à l'aide de VampirServer. La visualisation permet d'observer l'état des différents processus et

fils d'exécution d'un programme parallèle. L'état est représenté par la couleur de la frise, les processus sont répartis sur l'axe vertical, tandis que l'axe horizontal représente le temps. Les compteurs peuvent être visualisés en fonction du temps avec l'état des différents processus.

Une autre partie de la visualisation permet d'analyser des statistiques à propos des processus : les messages reçus et envoyés entre les processus, le temps total pris par chaque fonction ou encore le temps moyen de chaque fonction.

2.2.3 Débogage sur processeurs distants

Le débogage présente des mécanismes intrusifs dans les programmes hétérogènes. Ces mécanismes sont souvent similaires aux mécanismes de traçage. C'est pourquoi nous allons étudier les technologies existantes de débogage sur les systèmes hétérogènes. Dans cette section, nous allons d'abord voir le mécanisme de débogage sur un CPU et ensuite nous allons voir les différentes techniques de débogage sur des systèmes hétérogènes.

Débogage sur CPU

Le débogage est l'élimination des bogues dans un programme. Plusieurs outils existent pour atteindre ce but, mais l'outil le plus connu reste le débogueur. Celui-ci permet d'arrêter le programme cible à l'aide de points d'arrêts, d'exécuter les instructions une par une, d'inspecter l'état du programme et d'insérer des points de surveillance.

Pour pouvoir effectuer ces opérations, un débogueur fait appel au noyau pour pouvoir récupérer le processus en exécution. En effet, cela permet de modifier la mémoire du processus, accéder à l'exécutable et accéder aux symboles si ceux-ci sont inclus.

Pour implémenter un point d'arrêt, le débogueur remplace l'instruction ciblée par une instruction qui déclenche une faute (*trap*). Cette faute permet au débogueur de reprendre l'exécution du programme en main. Ensuite l'utilisateur peut effectuer diverses opérations pour modifier ou explorer l'état du programme. Lorsque le débogueur relance le programme, il réinitialise le compteur du programme d'une instruction, remplace le point d'arrêt par la bonne instruction, exécute uniquement cette instruction et remet en place le point d'arrêt. Ensuite le débogueur peut relancer le programme.

Détection automatique d'anomalies

Le débogage couvre aussi l'utilisation d'outils pour découvrir des bogues et pour analyser l'exécution d'un programme pour localiser l'origine du problème.

En effet, l'analyse de données de trace ou de profilage peut être utilisée pour détecter des anomalies. Un nombre d'évènements plus grand que la normale ou des baisses soudaines de débit peuvent signaler une anomalie de performance, l'analyse de latence peut révéler des comportements anormaux. La création d'outils de visualisation, tels que Trace Compass qui permettent d'analyser les données de traçage, ou des outils de surveillance comme Grafana et Kibana, est idéale pour aider un utilisateur à détecter des problèmes de performance. Cependant, de plus en plus d'outils utilisent l'apprentissage machine pour effectuer ces analyses automatiquement [118–121].

D'autres méthodes permettent de détecter automatiquement l'endroit où a eu lieu la faute comme Duan et al. [122], qui utilise la trace de la pile et l'analyse syntaxique du code pour pouvoir retrouver et évaluer automatiquement la cause de l'exception de pointeur nul dans le langage Java. D'autre part, des outils comme *Electric Fence*, Memsherlock [123], et BovInspector [124] permettent de faciliter le débogage de la mémoire.

Débogage distant avec GDB

Il est parfois nécessaire de déboguer un programme qui se trouve sur un système où les ressources sont limitées. Le débogueur ne peut donc pas être exécuté sur ce système. Dans ce cas, GDB propose deux solutions. L'une en utilisant un *stub* qui communique avec GDB grâce à un protocole, appelé *GDB serial remote protocol*. L'autre en utilisant *GDBserver* qui est un sous-ensemble de GDB qui fonctionne uniquement sur le système cible.

Dans le cas où l'on souhaite déboguer un programme sur un circuit spécialisé comme un accélérateur, alors chaque fonctionnalité doit être développée de façon à être adaptée au fonctionnement de l'accélérateur. Nvidia a créé un CUDA-GDB [125] à partir de GDB pour effectuer ces opérations sur ses processeurs graphiques, tandis que AMD a développé d'abord un débogueur intégré à CodeXL, mais récemment a développé un débogueur pour l'infrastructure ROCm, ROC-GDB.

CUDA-GDB

CUDA-GDB [125] est un débogueur développé à partir du projet GDB. Il utilise les capacités de celui-ci pour bloquer et explorer l'état d'un GPU en fonctionnement. Il permet de placer des points d'arrêt dans un programme kernel et de lire l'état de la mémoire. Il fonctionne en interrogeant l'interface de programmation de Nvidia, comme tous les outils de débogage utilisant les puces graphiques de cette marque. Il existe quelques outils de débogage conçus avec différentes interfaces par ARM, RogueWave ... Mais ceux-ci utilisent tous la même

interface pour communiquer avec le GPU Nvidia [126]. Récemment, la mémoire partagée entre le CPU et le GPU permet la préemption de fonctions de kernel. Cette fonctionnalité permet donc de déboguer un GPU sans bloquer entièrement son fonctionnement et ainsi permet le débogage en utilisant un seul GPU.

ROC-GDB

Chez AMD, l'initiative HSA a donné lieu à la création de ROCm, un ensemble de librairies destinées au domaine de la haute performance de calcul. En effet, le langage utilisé (HIP) comporte les mêmes commandes que CUDA. Cette ressemblance est voulue, car, elle permet à un script de traduire un programme CUDA en programme HIP. Cet ensemble de libraires utilise un système de files d'attente en mémoire partagée entre le CPU et le GPU. Celle-ci permet donc un débogage avec préemption similaire à celui de Nvidia.

Le débogueur fonctionne avec une interface de protocole d'application qui permet à d'autres débogueurs de fonctionner avec les accélérateurs AMD. L'interface et l'adaptation de GDB sont à sources ouvertes. Plusieurs concepts ont été rajoutés à GDB pour rendre compatible l'application. Il est possible de gérer des agents, un agent désigne un processeur - un CPU, GPU, FPGA, DSP -. Un programme hétérogène peut créer des files d'attente pour un agent, et envoyer des paquets dans la file d'attente. Un paquet de *dispatch* permet de lancer l'exécution d'un kernel sur un agent. Les fils d'exécution sur un agent peuvent être regroupés en un "groupe de travail". Chaque groupe peut partager de la mémoire. Ces constructions peuvent varier d'un agent à un autre selon son architecture. ROC-GDB manipule les agents, files d'attente et les paquets comme les fils d'exécution sur un CPU.

Toutes les pistes d'un fil d'exécution sont synchronisées. Lorsqu'un point d'arrêt conditionnel est atteint, la condition est évaluée pour chaque piste. Toutes les pistes s'arrêtent si l'une seule d'entre elles s'arrête. Comme un accélérateur peut exécuter un très grand nombre de fils en même temps, il est possible de limiter les points d'arrêt à un ou plusieurs fils d'exécution.

Détection automatique sur un système hétérogène

Ces outils sont similaires dans leur manière de fonctionner aux outils vus précédemment pour un système classique. Cependant, dans la conception de systèmes hétérogènes pour du calcul haute performance, il est nécessaire de développer des outils spécialisés pour la détection, la correction et la vérification des problèmes liés à l'exécution parallèle [127].

Nous allons d'abord voir les différents outils permettant d'analyser les problèmes liés à la mémoire. La mémoire étant distribuée et spécifique à chacun des fils d'exécution, il est sou-

vent nécessaire d'obtenir l'information avec une granularité fine pour permettre la détection d'anomalies [128]. Deux de ces outils utilisent des informations de profilage pour pouvoir donner un diagnostic sur les problèmes de performance liés à l'usage de la mémoire [128,129]. L'un [128] donne à l'utilisateur des possibles causes des problèmes détectés, tandis que ScaAnalyzer [129] est plus spécialisé pour démontrer des problèmes de scalabilité. Le troisième outil [130] permet de détecter automatiquement les dépassements de tampon. La technique utilise des canaris pour pouvoir détecter des dépassements, une technique similaire à celle utilisée sur les CPUs.

D'autres outils sont plus spécialisés pour détecter un problème majeur des algorithmes parallèles, les situations de concurrence : Barracuda [131] et CURD [132]. Les deux outils instrumentent le programme pour pouvoir effectuer leurs analyses. Cependant, Barracuda cherche à détecter des concurrences d'écriture au niveau de la mémoire, alors que CURD est plus générique et détecte d'autres types de problèmes.

2.3 Conclusion de la revue de littérature

Dans cette revue de littérature, nous avons étudié en détail les optimisations de performance pour mieux comprendre les défis qu'apportent les systèmes hétérogènes. En effet, ceux-ci mettent en relation différents coprocesseurs, qui possèdent pour la plupart une très grande bande passante. La communication est donc un problème majeur. De plus, l'ordonnancement et la répartition des charges sont eux aussi très importants pour s'assurer d'une collaboration optimale entre les coprocesseurs et le CPU. Ensuite, nous avons exploré les différents outils qui existent déjà. Parmi ces outils, LTTng-HSA est l'outil qui répond le mieux aux problèmes exposés précédemment.

Cependant, LTTng-HSA présente plusieurs limitations. Les différentes cibles de traçage ne permettent pas de tracer tous les évènements en une seule exécution du programme. Cela complexifie et ralentit significativement le processus. D'autre part, les vues proposées ne permettent pas d'avoir une vue d'ensemble du fonctionnement du système. Il est donc possible d'exploiter l'idée de base de LTTng-HSA pour s'intégrer à ROCm. Cependant, il est nécessaire de trouver un mécanisme différent qui est plus flexible et permet d'obtenir plus d'information, sans être obligé d'exécuter le programme à plusieurs reprises. Par ailleurs, cela permettra de développer une analyse plus poussée.

CHAPITRE 3 MÉTHODOLOGIE

Ce chapitre permet de mieux comprendre l'article qui suit. En effet, dans ce chapitre, nous allons d'abord présenter le contexte de cette recherche et les outils utilisés. Ensuite le travail réalisé sera détaillé et commenté et l'environnement de test sera décrit.

3.1 Environnement de travail

Cette partie explique l'environnement de travail utilisé pour développer l'approche de traçage, profilage et analyse de performances.

Nous avons choisi d'utiliser ROC-profiler et ROC-tracer pour enregistrer l'ensemble des données de traçage et de profilage. Ce choix a été fait car ROC-profiler et ROC-tracer sont les seules librairies qui supportent les dernières versions de ROCm. Nous n'avons pas décidé de continuer le travail effectué sur LTTng-HSA [81], car la capacité de traçage était limitée à certaines fonctions par exécution. Cela nécessitait plusieurs exécutions du même programme pour accumuler l'ensemble des données nécessaires, et une étape de fusion des données. Enfin, nous n'avons pas décidé de recréer un outil, car l'approche prise par ROC-tracer et ROC-profiler est très proche de celle de LTTng-HSA. L'instrumentation est similaire à l'une des premières versions de LTTng-HSA.

Enfin, nous avons choisi d'utiliser Trace Compass, car celui-ci supporte déjà de nombreuses sources, ce qui permettra d'obtenir plus facilement une vue d'ensemble en tirant parti du travail déjà réalisé. De plus, cet outil supporte le format des traces de ROC-profiler et ROC-tracer. Trace Compass est aussi à source ouverte et peut facilement être étendu pour analyser spécifiquement les traces HSA.

3.1.1 ROCm

ROCm est une implémentation du standard HSA qui permet donc d'accéder à l'API HSA. Cette API permet d'exécuter des programmes kernel, effectuer des transferts de mémoire et obtenir des métriques de performance des coprocesseurs connectés. Cette collection de librairies est accessible à source ouverte en ligne et est distribuée par AMD. La libraire ROCR s'assure de maintenir l'API conforme à l'API de HSA en agissant comme intermédiaire entre le GPU et l'utilisateur, ou le pilote graphique du système d'exploitation et l'utilisateur. La communication avec le système d'exploitation se fait en effectuant une libraire qui permet une translation des requêtes (*thunk*).

3.1.2 ROC-profiler

ROC-profiler est une librairie faisant partie de ROCm, développée par AMD. Nous utilisons cette librairie pour obtenir les compteurs de performance. Cette librairie limite la performance du programme profilé, car elle oblige les fonctions kernel à s'exécuter séquentiellement. Cette librairie produit un tableau des exécutions des fonctions kernel ainsi que les compteurs de performance. Le nombre de registres étant limité sur le GPU, il existe plusieurs catégories de registres qui représentent différents systèmes. Chaque catégorie est limitée à quelques compteurs, selon les capacités du GPU utilisé.

3.1.3 ROC-tracer

ROC-tracer est une librairie faisant partie de ROCm, aussi développée par AMD. Cette librairie permet de collecter les évènements de trace dans un tampon circulaire. De plus, un outil de traçage permettant d'instrumenter les différentes API est accessible dans cette libraire. Nous utilisons cette librairie pour produire les fichiers de trace. Ces fichiers sont ensuite formatés en une seule trace au format *TraceEvent*. Ce fichier possède notamment des informations qui permettent de corréler les évènements de l'API et les exécutions de fonctions kernel.

Il est important de mentionner que l'utilisateur n'utilise pas souvent ROC-tracer directement. Par défaut, l'outil ROC-profiler se charge de la configuration et de l'exécution de ROC-tracer et de l'outil de traçage. Dans la suite, on utilisera le nom ROC-profiler même lorsqu'il s'agit de générer des traces, puisque l'utilisateur interagit avec ce composant.

3.1.4 Trace Compass

Trace Compass est un outil de visualisation qui permet d'effectuer l'analyse de traces. Ce logiciel est à source ouverte et une documentation est disponible, donc il est facilement modifiable. En effet, il est possible de créer ses propres analyses en réutilisant des fonctions déjà programmées. Dans ce travail, nous avons donc développé une analyse qui permet de passer d'un fichier de trace à un graphe d'états. De plus, de nombreuses sources sont déjà disponibles sur ce logiciel. Notamment, il est possible d'analyser les traces du système d'exploitation et de les corréler avec d'autres traces. Tous ces éléments montrent qu'il s'agit d'un logiciel idéal pour développer notre approche et qu'elle sera extensible dans le futur.

3.2 Travail réalisé

3.2.1 Analyse des informations de trace

Initialement, un programme a été développé pour transformer le format des traces de ROC-profiler. Ensuite, une analyse préliminaire a été développée pour supporter l'affichage des exécutions de fonctions kernel. Cette analyse préliminaire était un prototype permettant de s'assurer que cette approche fonctionnait correctement. Cependant, de nombreux évènements n'étaient pas disponibles, et les visualisations générées étaient très similaires à celles obtenues avec LTTng-HSA.

De plus, cette approche a vite été abandonnée, car elle nécessitait de mettre à jour trop régulièrement la traduction des fichiers de trace et l'analyse. De plus, il est devenu possible de lire directement le fichier de trace généré qui utilisait le format *TraceEvent*. Ce format est supporté par Trace Compass. Cependant, des limitations dans l'implémentation de Trace Compass et dans le format lui-même (précision en microseconde et non en nanoseconde) ont dû être rectifiées.

Le module développé lit actuellement les informations du fichier de trace, remplacent les temps des évènements par leur temps en nanoseconde pour pouvoir supporte des évènements ayant une durée inférieure à une microseconde. Ensuite, les évènements de fonctions kernel, de l'API et des transferts mémoires sont corrélés pour identifier les dépendances entre les évènements. Cette corrélation est faite avec une marge d'erreur à cause du manque de précision des informations. Dû à une évolution de Trace Compass et de ROC-profiler, celle-ci n'a pas pu être maintenue, mais elle peut être rétablie.

3.2.2 Analyse des informations de profilage

Les informations de profilage n'étaient disponibles que dans le tableau créé par ROC-profiler initialement. Ensuite, ces données ont été intégrées au fichier de trace, directement dans les métadonnées de l'évènement. Ces compteurs de performance sont enregistrés entre le début d'une exécution d'une fonction kernel et sa fin. Ces compteurs sont récupérés et insérés dans la représentation intermédiaire de Trace Compass. Cela permet donc de récupérer ces valeurs et de garder l'association à l'évènement de la fonction kernel.

Suite à cette étape, la valeur est affichée comme elle est enregistrée. Le compteur change de valeur à la fin de l'exécution d'une fonction kernel. Cet affichage a l'avantage de montrer les variations entre deux fonctions kernel. Comme ces kernels pourront peut-être s'exécuter simultanément, afficher les deux valeurs permet de détecter, grâce à d'autres analyses, des

interférences entre deux fonctions kernel.

3.2.3 Récupération de métadonnées

Des métadonnées doivent pouvoir être récupérées pour certaines analyses. Ces métadonnées comprennent certaines informations, comme la fréquence de la mémoire, des unités de calcul et les versions du matériel utilisées. Ces métadonnées sont donc enregistrées dans un évènement de métadonnée et enregistrées pour l'analyse. Ils ne sont pas dépendants par rapport au temps et donc n'ont pas besoin d'être enregistrés dans la représentation intermédiaire de Trace Compass.

Ces données seront utilisées pour générer les modèles d'analyse comme l'analyse *roofline* qui a besoin des capacités de transferts mémoire et de calculs pour calculer les bornes. De plus le modèle TMAM nécessite d'avoir la relation entre les compteurs de performance et les métriques du modèle pour estimer l'endroit du système qui est bloquant en termes de performance.

3.2.4 Compatibilité avec Theia Compass

Theia Compass est une extension de l'éditeur de code Theia qui se connecte au serveur Trace Compass. Ce serveur utilise les mêmes composants que Trace Compass. Ainsi, les analyses développées sur Trace Compass sont visibles sur Theia Compass. Certaines modifications ont été réalisées pour rendre compatible les modules développés avec le *Trace Server Protocol*. Ceci permet d'avoir une approche intégrée avec des outils de développement moderne.

Les compteurs de performance peuvent être affichés directement dans Theia, car les capacités de visualisation ne sont pas les mêmes. Cependant, de nombreuses fonctionnalités ne sont pas présentes, mais de futurs travaux pourront très bien développer cette avenue.

3.3 Environnement de test

L'environnement utilisé pour réaliser ce travail était un ordinateur avec les caractéristiques suivantes :

- Processeur graphique : AMD Radeon Vega 56
- Processeur central : Intel Core i7-4770
- Compilateur : HIPCC 3.5.20231-93097e0
- Clang : 11.0.0
- Système d'exploitation : Ubuntu 18.04.4

La version de ROCm utilisé était la dernière disponible au moment des tests et correspond à la version du compilateur. Celle-ci se conforme à la version 1.2 de HSA. Le banc d'essai Rodinia [41] a été utilisé pour effectuer les expériences. Sur les vingt programmes de ce banc d'essai, seulement dix-huit ont été utilisés : nw, gaussian, b+tree, hybridsort, backprop, streamcluster,kmeans, nn, heartwall, dwt2d, lud, pathfinder, srad, bfs, lavaMD, cfd, particlesort, hotspot. Deux n'ont pas été retenus à cause de divers problèmes : leukocyte (erreur de segmentation) et myocyte (erreur de compilation).

Avec l'environnement décrit dans le présent chapitre, nous avons développé un module d'analyse intégré à Trace Compass. Ce module permet de créer des visualisations à partir de données de trace et de profilage. Celles-ci permettent à l'utilisateur de comprendre et de vérifier le comportement du programme durant l'exécution. De plus, cela permet à des analyses antérieures d'être développées en utilisant les données provenant de ce module. Avec ce module et l'environnement de test décrit précédemment, une évaluation de la performance de cet outil ainsi que l'étude d'un cas d'utilisation ont été réalisées. Ces extensions ainsi que les résultats obtenus sont décrits dans l'article intitulé "*Visualisation of Profiling and Tracing in CPU-GPU Programs*" présenté dans le prochain chapitre.

CHAPITRE 4 ARTICLE 1 : VISUALISATION OF PROFILING AND TRACING IN CPU-GPU PROGRAMS

Authors

Arnaud Fiorini <**arnaud.fiorini@polymtl.ca**>

Michel R. Dagenais <**michel.dagenais@polymtl.ca**>

Department of Computer and Software Engineering, École Polytechnique de Montréal

Submitted to : Concurrency and Computation : Practice and Experience

Keywords : tracing, profiling, tracecompass, rocm, heterogeneous, cpu-gpu, gpu, visualisation

Abstract

As the complexity of the toolchain increases for heterogeneous CPU-GPU systems, the needs for comprehensive tracing and debugging tools also grows. Heterogeneous platforms bring new possibilities but also new performance issues that are hard to detect. Some techniques that were used on CPU programs are now adapted to GPUs. However, there are some concepts specific to GPUs, like SIMD processing, and the effects of the close interactions between the CPUs and the GPUs, with shared virtual memory and user-level queues. Multiple sources of data need to be extracted and correlated to obtain a more global view of the performance.

In this paper, we introduce a novel approach for measuring and visualizing performance defects inside CPU-GPU programs by combining kernel events, compute kernel events, user API calls and memory transfers. We also created two new views that combine this information, to help provide a global view. This framework uses the open source user queue system described in the HSA standard. It can easily be adapted to any user queue system for heterogeneous computing devices. We compare this framework with current existing tools and test it against the Rodinia benchmark. We look at how the program characteristics and its execution behaviour affect the tracing and profiling overhead. Then, we use Trace Compass to analyze and visualize the resulting trace.

4.1 Introduction

Most High Performance Computing clusters now rely heavily on heterogeneous systems with Central Processing Units (CPUs) and Graphical Processing Units (GPU) coprocessors. Tracing and profiling CPU-GPU programs thus has become essential for High Performance Computing. The development of computing environments that can exploit the computing power GPUs and of specialized applications that exploit this computing power has greatly increased the power of computers for number crunching [133]. Developing for GPU devices is challenging and the toolset is limited [134]. This article presents a novel approach to trace and profile CPU-GPU programs that rely on user-level queues. Furthermore, existing tools offer many different analyses to investigate performance issues in conventional CPU programs. The goal for heterogeneous systems is to correlate CPU and coprocessor informations in order to have a more global picture of what is happening. This was typically not possible with most GPU tools, since they were proprietary tools, provided by the hardware vendors, in an effort to hide the hardware inner details. The advent of the GPU Open initiative, and the standards from the Heterogeneous Systems Architecture (HSA) foundation, open up a number of interesting possibilities in this area.

Statistical profiling works reasonably well to detect many performance problems in CPU bound programs. However, they are insufficient when significant latencies are caused by interactions with the operating system, coprocessors or even other nodes [135], as is the case in heterogeneous systems. This is why tracing and profiling are often used to solve those issues.

In recent years, heterogeneous systems have become more tightly integrated, with new technologies like unified memory models and user mode queuing. The emergence of shared virtual memory between CPUs and GPUs enables several optimizations, reducing the number of operations that require a system call to the GPU device driver [10, 11, 14, 18–20]. Furthermore, the HSA foundation has defined a model for queues that enables a more precise control over scheduling the GPU workloads. These advances have led to numerous issues that can not be detected or studied with current tracers and profilers. Most tracers and profilers focus on the CPU, the operating system, or on one coprocessor and do not provide a global view of the system. This is an issue when all these components should work tightly together, and performance defects arise in some scenarios because of interaction problems. A performance diagnostic toolkit that supports heterogeneous hardware and can integrate and correlate tracing and profiling information from all levels is needed in this context.

GPU simulators are sometimes used to analyze new hardware and software approaches to

memory and scheduling issues [136–140]. The use of GPU simulators is limited since they do not simulate complete systems and thus cannot really account for the interactions with the rest of the system like the memory hierarchy, system busses and the CPUs. The challenge is to obtain a precise and global view of heterogeneous systems execution, while not significantly impacting their performance. Indeed, a large trace and profile collection overhead can hide the performance issues we are trying to find and solve.

Trace Compass is an advanced trace analysis and viewing tool. It performs offline analysis and already supports many different tracing sources that are used to provide a global system view. In LTTng-HSA [81] and in [141] heterogeneous systems were studied using different trace sources integrated into Trace Compass. While LTTng-HSA was particularly interesting, integrating information from the HSA user-level API, it still missed information about other runtime APIs such as HIP and KFD. Furthermore, the asynchronous memory transfers were not shown as well and there was less metadata recorded in each events. This limited the analyzes that could possibly be done at a later stage.

The Heterogeneous System Architecture (HSA) model, proposed by the HSA foundation, was designed to support multiple programming models and coprocessor types. One of the key requirements for an HSA device is the support of user mode queuing to send commands. HSA queues are a low-level queue implementation directly available to the user context. It allows user applications to send low-latency Architected Queuing Language (AQL) packets directly to coprocessors and to precisely control parallel workloads [142]. This fine granularity enables improvements in scheduling approaches and better performance [33, 143, 144].

Currently, GPU developers rely on vendor supplied tools to trace and profile their programs, like Nvidia Nsight [106] or AMD CodeXL [105], or libraries like Nvidia CUPTI [109] used by tools like Vampir [115]. These tools are useful in the context of GPU focused development. However, it is difficult to correlate the information that they provide and integrate it with other sources. This is a significant problem when computer systems are increasingly heterogeneous and use different coprocessors that need to be well coordinated to handle a diversified set of tasks. It has been shown that correlating trace events with other coprocessors is beneficial [141]. Tools that only get information from a specialized coprocessor and are closed source cannot easily be used to associate traces of different sources.

The Radeon Open Compute platform (ROCm) is an AMD open source initiative that implements HSA. This platform is a modular toolchain to compile and run software using the HSA standard, but is also able to run programs on Nvidia hardware. ROCm is open source, it was designed to be integrated into larger tools that can provide a global view of the system performance. It thus allows to integrate and correlate different tracing and profiling sources

more easily. For this reason, we decided to use ROCm for this research along with Trace Compass. Trace Compass is an open source trace visualiser. It can correlate between multiple sources by using synchronization [145] and event matching algorithms [89, 146]. It can easily be extended to support specialized views and analysis.

In this study, we propose a novel approach to trace, profile and analyze CPU-GPU programs exploiting data from the Linux kernel, GPU device drivers, user-level applications and the GPU runtime support using the ROCm ROC-profiler [147]. The proposed framework was evaluated by executing the Rodinia benchmark [41] to see the performance overhead associated with the proposed approach. The analysis and visualisation were developed using Trace Compass (Link to the modified version). The main contribution is to provide an efficient and extensible (open source) framework for tracing, profiling and analyzing the behaviour of CPU-GPU programs, measuring the overhead of tracing and profiling a GPU application, and its impact in relation with the program characteristics.

4.2 Related Work

In this section, we look at the main tools that are used to profile and trace High Performance Computing (HPC) applications. First, we look at TAU and Vampir, which are toolkits that were developed for HPC applications. They were later enhanced to access GPU performance counters. Then, we investigate CodeXL and Nsight, tools developed specifically for GPU development, by GPU vendors AMD and Nvidia respectively. Thereafter, we examine LTTng-HSA and compare it with our approach.

4.2.1 TAU

TAU is a toolkit which contains tracing, profiling and visualisation tools. It was originally built for HPC applications running the Message Passing Interface (MPI) on multiple nodes, each possibly containing multiple cores and running OpenMP [111, 112]. This toolkit is composed of three layers : instrumentation, measurement and visualisation. The instrumentation and measurement layers are decoupled with the *TAU measurement API*. This architecture allows the user to configure how the tracing is done, and how much impact it will have on the application execution time.

There are different tracing and profiling mechanisms. For tracing, TAU is capable of handling manual, preprocessor based, library preload based (using an already instrumented library) and binary instrumentation. It is possible to adjust the proportion of functions calls traced (e.g. not trace trivial functions) to reduce the overhead to a minimum [111]. TAU interfaces

with different performance API, like PAPI or CUPTI, to access GPU performance counters, and visualise the results. However, it does not have access to operating system events and only offers access to a limited number of GPU API events.

4.2.2 Vampir

Vampir is a toolchain, similar to TAU, that enables instrumentating, profiling and analyzing program behaviour. The instrumentation can be achieved using four different mechanisms : compiler instrumentation, instructions inserted using source-to-source techniques [116], manual instrumentation or using an already instrumented library. To record traces and profiles, it can use Score-p [148], Scalasca, TAU, VampirTrace or other libraries that produces traces using the OTF or OTF2 format.

These tools are very mature for CPU centered parallel development. It is possible to get the performance counters of devices that comply with PAPI. However, they lack the support for heterogeneous performance evaluation, and in particular the interactions between the CPU and coprocessors, including support for operating system events. Furthermore, correlating between multiple trace sources, and the support for user-level queues, are not present. It is therefore necessary to extend CPU focused tools to GPUs and other coprocessors. In order to better understand the needs of tools for GPU applications, we will review the two major offerings : CodeXL developed by AMD and Nsight developed by Nvidia.

4.2.3 CodeXL

CodeXL [105] is a tool developed by AMD to profile and trace GPU programs that use the AMD hardware and device drivers. As part of the GPU open initiative, it was decided to put the emphasis on existing open source toolchains and provide open source library modules to feed these tools, rather than a separate tool like CodeXL. CodeXL was then open sourced but is deprecated as further development effort goes into the different open source modules that comprise ROCm such as the rocprofiler, the roctracer, or extensions to GDB (ROCgdb) to debug programs running on GPUs. CodeXL was one of the main tools which could help GPU programmers to troubleshoot their programs. It had an integrated debugger which has also been deprecated, being replaced by ROCgdb. CodeXL had the capability to profile the CPU in a program, giving this tool an advantage over GPU-focused tools like Nsight. It offered similar features for the GPU, tracing memory transfers, API interactions and compute kernel launches. It could measure performance counters for GPU applications and calculate derived metrics as Nsight does.

Even though it is now deprecated, after being open sourced, it is interesting to study this tool because it contains some features to analyze CPU behaviour and performance counters. However, it misses crucial functionalities to support operating system events and many programming models and hardware. It supports the profiling and tracing of OpenCL programs, but correlating CPU and GPU information is not possible. The new open source tools that are part of ROCm are now built to integrate well with other open source tools such as TAU, Trace Compass or GDB. We will look next at Nvidia Nsight which is a complete profiler and tracer for GPUs.

4.2.4 Nvidia Nsight

Nvidia Nsight [106] is a tool built by Nvidia to give users of their hardware the ability to profile and trace their programs. It is similar in function to CodeXL [105]. NVProf [107] was merged with Nsight so that it offers a development environment for heterogeneous compute applications. This tool is not open source and it is difficult to know how it operates. It has been used as a reference in studies to compare performance analyses [91].

It has the capacity to provide chronological analyses for the execution of programs. These chronological views show the different relevant operations that occur during the execution of a heterogeneous compute application. It shows memory transfers, compute kernels, runtime API calls and driver API calls. There is also the capability to view the streams in which are running the different compute kernels. It enables the user to have an overall system view showing the general behaviour of the program. Then, for profiling, Nsight can record specific performance counters for compute kernels. These counters are used to calculate derived performance metrics that can be used by the developers to troubleshoot performance issues. A roofline model [98] can also be generated to analyze the specific bounds of a program (i.e. to know if a program is compute or memory bound).

The main issue is that the only way to do further analysis with this tool is to use the CUPTI library [109]. The current state of the API does not allow to do offline correlation and synchronization with other event sources such as operating system events or user-level runtime library events. This is crucial to obtain a global view of the program behaviour and performance. Nvidia Nsight is an advanced tool to analyse GPU programs. However, the main issue is that it only allows GPU related analysis. Much insight is to be gained by combining CPU and system analysis with GPU related analyses.

Since our goal is to combine tracing and profiling information from all levels, to obtain a global view of the performance and behaviour of complex heterogeneous applications, Nsight and CUPTI were not suitable for our needs. Instead, the modular and open source ROCm

toolchain provides us will the needed interfaces and facilities, and they could even be modified if needed. ROCm supports multiple platforms and, being based on HSA standards, it should work with other HSA-compliant systems in the future.

4.2.5 LTTng-HSA

LTTng-HSA [81] is a set of tools that allow tracing and profiling programs that uses the ROC-runtime. This runtime uses the HSA standard in order to run GPU computations. The HSA standard was developed to provide a common API between different vendors of heterogeneous computation devices. LTTng-HSA uses the ROCm infrastructure that implements the HSA standard in order to insert tracepoints inside the API. LTTng [149] is used to insert different types of tracepoints. It used LD_PRELOAD to insert tracepoints at the beginning and end of each API function. LTTng user-space tracepoints were used. However, some events of the ROC-runtime are synchronous and others are asynchronous, and many target areas cannot be traced simultaneously.

The solution proposed was to trace different targets and chose the relevant target for the application. This is not ideal for multiple reasons. We cannot collect all the information in one execution. This necessitates multiple executions of the program (assuming that the behaviour does not change between executions) and subsequent merge of the traces. This requirement has a significant impact on the time it takes to get a complete trace and also complicates the tracing process. Then, the trace is analysed using Trace Compass. They proposed two new analyses, the first called *call stack view* serves to see which calls happened on each queue with respect to time. Each queue is represented by a lane, and each segment represents an API call. The second analysis, called *the queue profiling view*, shows the state of the queue, and of the compute kernels executed in the program. These views are quite limited in their use to understand the behaviour of a CPU-GPU program. We propose in this article an improvement over LTTng-HSA, built upon the ROC-profiler [147] library. Because LTTng-HSA is not collecting HIP, KFD and asynchronous transfer events, the views generated are very limited. With the additionnal events provided by ROC-profiler it opens up a number of possibilities for new and more detailed views. It would be possible to add support to LTTng-HSA to get those events as additionnal targets. However, as each target is traced separately and merged, this does not scale well.

LTTng-HSA [81] was developed to instrument the ROC-runtime. This implementation allowed the creation of multiple views in Trace Compass and had a low execution overhead. While visualising GPU command queues and API calls with a low overhead was possible, the capabilities of this tool were otherwise limited. LTTng-HSA was using LD_PRELOAD

to inject an instrumented library. Our approach uses a custom ring buffer tracing library built into ROC-profiler instead of using LTTng. The ROC-profiler library allows tracing the Heterogeneous-Compute Interface for Portability (HIP) and Kernel Fusion Driver (KFD) API, where LTTng-HSA did not. HIP is a language interface like CUDA that allows the user to program compute kernels and memory transfers to any HSA agent. The KFD is the HSA kernel driver used by ROCm.

4.3 Methodology

We chose in this article to use the ROCm framework because it is built on a generic model (HSA) that ensures that the techniques used are adaptable to other coprocessors and heterogeneous systems. Another contributing factor is the support of new technologies directly integrated within HSA. The unified virtual memory and user mode queuing are important characteristics of recent heterogeneous systems that intend to use collaborative computing with coprocessors. For example, newer GPUs from AMD, ARM and Intel all support virtual shared memory, which the HSA model can support. A benchmark of the ROC platform has shown that it is an improvement over other AMD drivers [31], especially concerning the HIP programming language. It is important to note that the profiling and tracing analysis approach presented here uses ROCm as a prototyping vehicle but is general, and could be used with other heterogeneous systems, beyond ROCm.

The tracing and profiling analyses proposed in this paper use multiple tools and libraries. To instrument and recover execution information, the ROC-profiler library, working with the ROC-runtime library, is used. Then Trace Compass is used to analyze the tracing and profiling information, and to generate visualisations. The tracing mechanism is outlined in Figure 4.1 and the tracing framework is explained in details in three parts : Tracing, Profiling and Visualisation. Then experiments were conducted in order to demonstrate the new capabilities of the proposed framework and to measure the tracing and profiling overhead.

4.3.1 Tracing

The ROC-profiler [147] library is loaded by the ROC-runtime using an environment variable. This environment variable is used by the ROC-runtime to dynamically load every compatible tool. Then, an API table is sent to the tools when it is loaded so that the tool can have access to the inner workings of the runtime. It can then intercept API calls and register its own hooks. In our case, the ROC-profiler library and the tracer tool will intercept API calls by instrumenting each call with a tracepoint before and after. This mechanism is shown in Figure

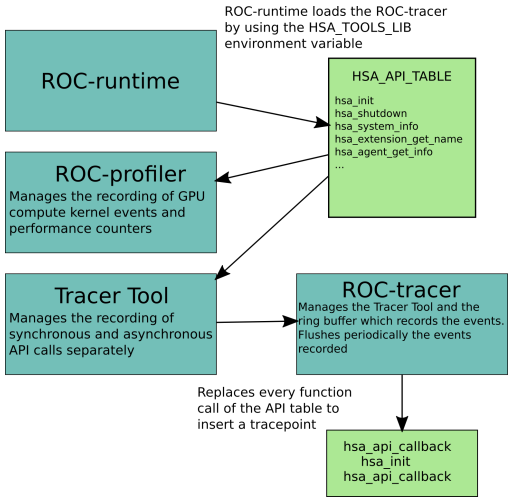


Figure 4.1 ROC-profiler tracepoint insertion mechanism

4.1. The API tracepoints are registered using the ROC-tracer library by replacing functions pointers. The ROC-tracer [150] library also manages the ring buffer that records the different events, and periodically flushes the events on disk. The API table is instrumented such that each different API type is identified and relevant information is recorded for HSA, HIP or KFD. The tracer tool can be configured to trace HIP and HSA together, or HSA, HIP and KFD separately.

The GPU events being collected asynchronously, they are instrumented differently. A mechanism similar to LTTng-HSA [81] has been implemented to sort the events recovered. This way, the events in the trace are reordered to insure that they appear in the trace file by increasing time of occurrence. The compute kernels are launched by enqueuing AQL packets onto the runtime user queue owned by one kernel agent. This packet is a *kernel dispatch packet* enqueued inside a buffer. Then the packet processor monitors the queue and detects when there is a packet ready to be consumed through a *doorbell signal*. Then the execution of the packet is asynchronous and can start at any time [27]. By replacing the API functions pointers, the tracer tool tracks these packets using a mechanism that is flexible and efficient. It records multiple events, and associated information, from these tracepoints. These events are described in Table 4.1. The events that have been recorded are then stored in a temporary database. This enables further processing of the trace produced, ordering the events properly, to simplify offline analysis.

Some events (e.g. the compute kernel events) are produced asynchronously (the event entries are created after the event has passed). These events need to be processed to be merged in the trace, such that order and dependency relationships are preserved. For example, performance

counters are recovered and added to the related compute kernel events. Dependency relations are also added such that links can be made between events, kernel compute events, kernel launch commands, memory copy commands and memory transfers.

Events	Description
HIP events	Events derived from HIP calls from the user program
HSA events	Events from API calls done through the ROC runtime
KFD events	Events from program communication to the kernel driver thunk
rocTX API	API for registering runtime API callbacks that can be associated to API calls using a correlation ID
Compute kernel events	Recorded by the ROC-profiler library, hardware parameters are also registered
Performance counters	Performance counters evaluated and recorded per compute kernel events
ROC-profiler backend API	Access to the information recorded, profiling activation, and intercept to customize the behaviour of ROC-profiler

Tableau 4.1 Events and data accessible through ROC-profiler and ROC-tracer

4.3.2 Performance Counters

Performance Counters are recorded using the ROC-profiler [147] library. It is injected into the ROC-runtime using the same environment variable. This tool is configurable in order to adapt to a specific set of performance counters that are available on the specified agent. This file describes the performance counters to be recorded during the execution. To record these counters, GPU registers are used. Because there is a limited number of registers, not all performance counters can be enabled simultaneously. Furthermore, each register needs to be reset, before and after each compute kernel is run, to obtain an accurate reading. Because the performance counters are recorded for the whole device, running multiple compute kernels concurrently will have a huge impact on the resulting counter values. For this reason, the compute kernels are executed sequentially. This is interesting to characterise in details the behaviour of a compute kernel, but reduces the potential for parallelism between compute kernels and can incur a very significant overhead.

Nonetheless, it is important to note that the performance counters of a compute kernel are tied to its performance characteristics and are a very good predictor of its future performance [36, 37]. Therefore, it is not required to run the whole program to get a good picture of the behaviour of one compute kernel. In other words, it is possible to get the performance counters needed for performance analysis using a simple example of the target compute kernel. This avoids the significant impact that this profiling will have on a large program.

The ROC-profiler library can also record derived counters that are described by the user as a combination of the raw counters. This allows the users to define their own set of counters using the basic counters available on the agent used. Using higher-level user-defined counters is very useful to experiment and troubleshoot performance issues. For example, it is possible to calculate the fetch size of a compute kernel using multiple low-level counters. This eases

the analysis and simplifies experimenting and diagnosing.

4.3.3 Visualisation

The next step after recording the trace is to analyze and visualize the events using different kinds of views. This last step is performed using Trace Compass. A ROCm plugin, integrated into Trace Compass, was developed. This plugin analyzes the trace generated by ROC-profiler. The trace is generated with a python script that accumulates the information and does some post processing, merging together multiple sources.

The analysis runs using a state machine which records the state of different processes. It handles events in chronological order and records the state changes. The events in the trace must thus be sorted before running the analysis, because the state cannot be changed in the past during the analysis.

Two new views are produced after the analysis, the performance counter view in Figure 4.3 and the ROCm system view in Figure 4.2. The performance counter view shows the state of each counter throughout the program execution. It allows the user to compare different compute kernel executions. Also by using derived counters, it can be used to analyze various performance metrics. In Figure 4.3, we can see the counters being updated after each compute kernel run. The graph shows the counters relative to the value being shown, so that it is easier to compare performance counters between different compute kernels.

The ROCm system view is showing data tranfers between the host (CPU) and the devices (in this case GPUs), API calls by the analyzed program, and compute kernel executions by the GPU. In Figure 4.2, the left column represents a tree where each lane is classified into different categories. An event can be a GPU event, a system event (HIP API or HSA API) or a memory transfer. For each GPU and for each queue, a lane is created for every compute kernel. For system tracing, each lane is under one thread under each types of API. When simultaneous events arises in one lane, another lane is created under it to display the two events at the same time. The two views are also time-synchronized which helps user interpretation.

In the example shown in Figure 4.2, a compute kernel is run, and we can see the user thread waiting for the result, and then a memory copy is launched after the result is received. This memory copy command is then launched asynchronously from the device to the host.

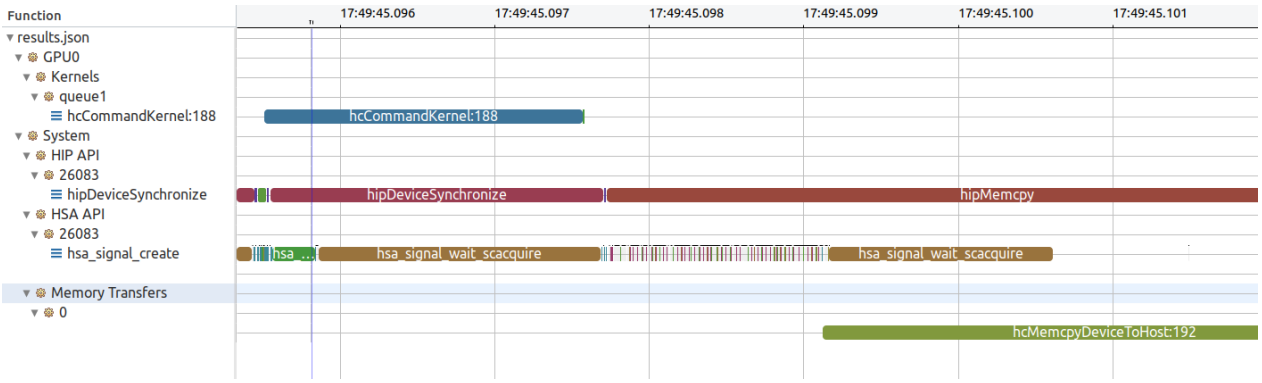


Figure 4.2 ROCm Plugin System View in Trace Compass

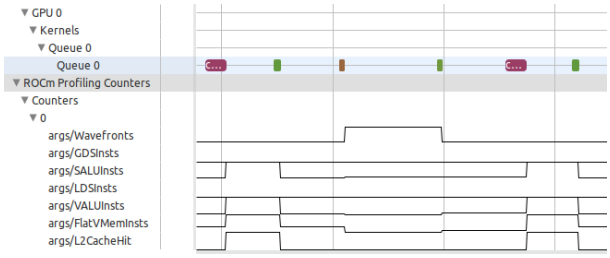


Figure 4.3 ROCm Performance counter view View in Trace Compass

4.4 Results

4.4.1 Overhead Analysis

In this paper, we evaluate our tracing and profiling framework using a benchmark. Recent benchmarks, specifically designed for heterogeneous systems, exist but are not yet available for the HIP language. Therefore, we used the Rodinia benchmark [41]. It correctly represents a variety of workload [44] and is a proven workload to test computing and profiling techniques.

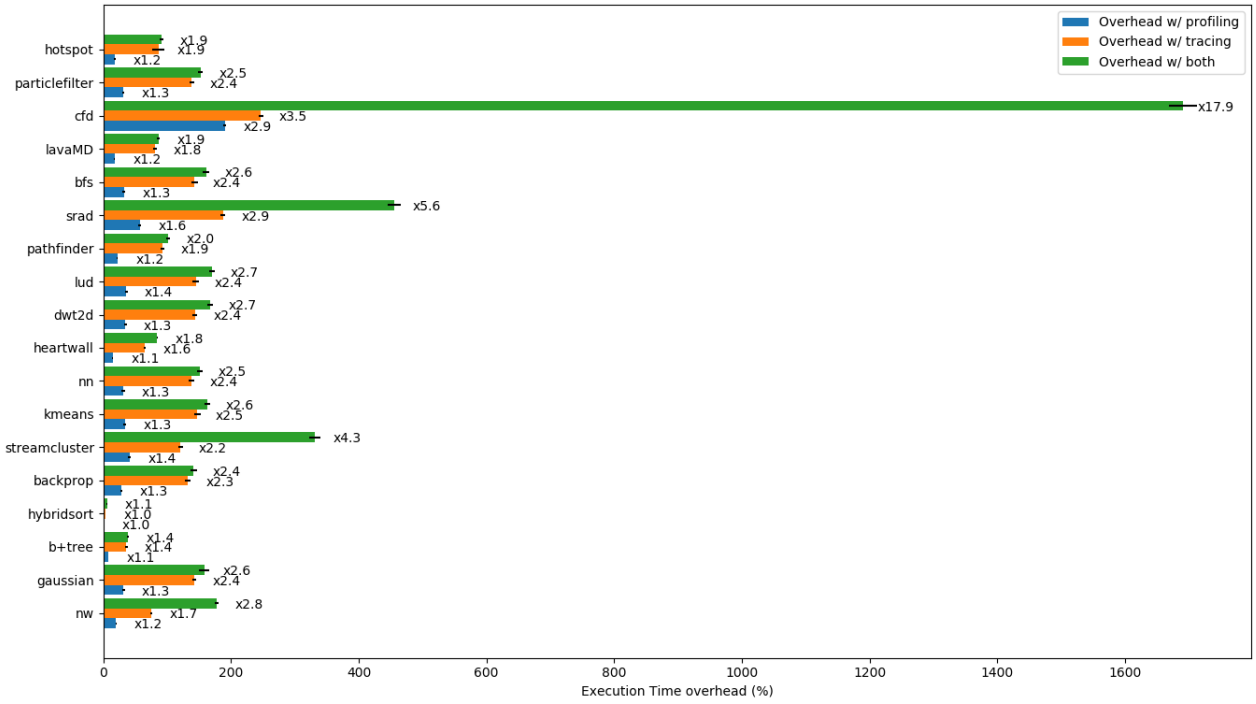


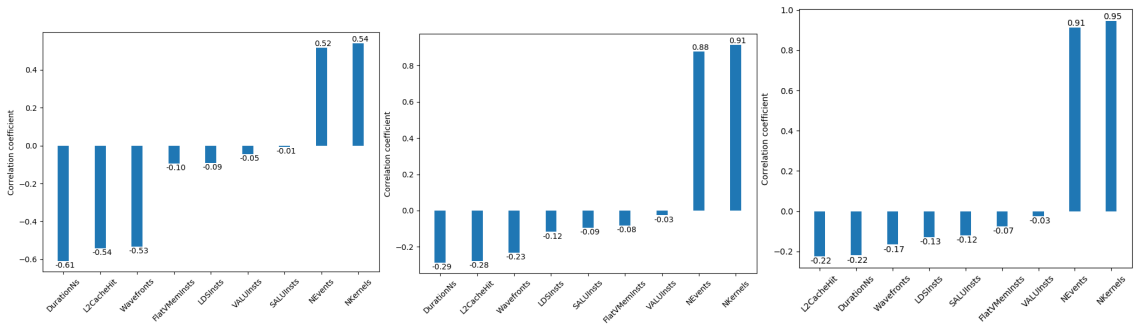
Figure 4.4 Overhead results from the Rodinia benchmark (%)

The following results were obtained using an Intel Core i7-4770 (Haswell) with 16 GB of main memory and a Radeon Vega 56 (GFX900). All benchmarks were compiled using HIPCC version 3.5.20231-93097e0 with clang version 11.0.0. They were executed on Ubuntu Linux release 18.04.4. Not all programs from the Rodinia benchmark were run due to different errors. Eighteen programs were run : nw, gaussian, b+tree, hybridsort, backprop, streamcluster, kmeans, nn, heartwall, dwt2d, lud, pathfinder, srad, bfs, lavaMD, cfd, particlesort, hotspot. Two programs were not run : leukocyte (segmentation fault) and myocyte (compiler error).

We defined 4 different scenarios : i) the control where the program is run without any interaction, ii) the profiling case where only the profiler is enabled, iii) the tracing case where the tracer is enabled, and iv) the tracing and profiling case where both the tracer and the

profiler are enabled. These scenarios were executed 20 times consecutively for each program of the Rodinia benchmark, totaling 1440 executions. The results are shown in Figure 4.4. The timings values were checked for outliers using the standard deviation (σ). All values exceeding $\overline{x} + (3 \times \sigma)$ were replaced with the 85th percentile. All values below $\overline{x} - (3 \times \sigma)$ were replaced with the 15th percentile.

In Figure 4.4, the overhead is calculated against the average execution time of the control case. The profiling case has the minimal overhead and the profiling and tracing case has the maximal overhead, with 17 times the control execution time for the cfd program. There are multiple reasons that can explain this overhead. The number of events can be signicant, which impacts the input and output of the tracing and profiling. If there are a lot of concurrent compute kernels, then the tracing and profiling have a big impact, since the profiling forces the compute kernels to run sequentially and resets the performance counters for each compute kernel. The tracing has to record every compute kernel execution and request specific information between each run. The impact of both of these running at the same time can add up and explains this overhead.



(a) Tracing (b) Profiling (c) Tracing and profiling

Figure 4.5 Correlation between overhead and performance metrics

Next, we look at the correlation between the overhead, previously obtained, and the performance metrics recorded on each compute kernel. The Rodinia experiments were each run once, separately from the experiments measuring the runtime using the tracer and profiler. During this, the following parameters were recorded : SALUInsts (average scalar instructions), VALUInsts (average vector instructions), FlatVMemInsts (average flat instructions), LDSInsts (average LDS read or LDS write), L2CacheHit (percentage of instructions that hit the L2 cache), Wavefronts (total number of wavefronts), DurationNs (the duration of the compute kernel in nanoseconds), NKernels (the number of compute kernels executed), NEvents (the number of events recorded in the trace). These values were recorded for each Rodinia experiment. Since these values were available for each compute kernel instance, they

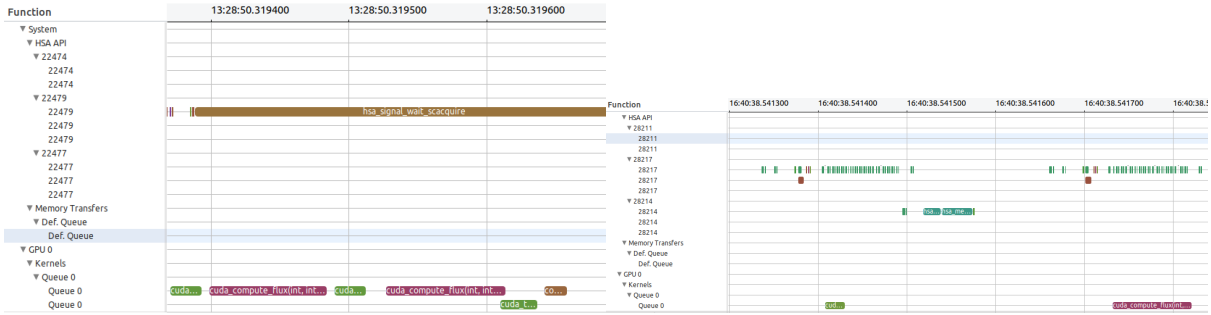
were averaged for each experiment, except for the NKernels and Nevents. The decision of averaging these values was taken because it keeps these metrics more independent. Indeed, for example, the summed duration is strongly correlated with the number of compute kernels, whereas the average duration shows a clearer picture of what is happening during the execution.

In Figure 4.5a, the most positively correlated feature is the number of compute kernels, with a correlation coefficient of 0.55. This is easily explained as there are several compute kernels events for each compute kernel, and it thus directly impacts the number of events that need to be traced. The fact that the overhead is less correlated with the number of events than with the number of compute kernel events, can be explained by the fact that compute kernel events take a longer time to be traced because they carry a largent event payload. The average duration of compute kernels is inversely correlated with a coefficient of -0.64. This shows that if the compute kernels run faster then, the programs spend relatively more time generating compute kernel events, and waiting for the CPU, and therefore are more impacted by the tracer.

In Figure 4.5b, there is a much stronger association between the number of compute kernels and the overhead generated. As with the previous result, we can conclude that the impact on the program scales with the number of compute kernels running. This is explained by the fact that the performance counters have to be reset between each compute kernel execution and the compute kernels have to run sequentially.

In Figure 4.5c, the combination of tracing and profiling has a much bigger impact on the runtime. It is less about the duration of the compute kernels, and is more correlated with the number of compute kernels. The effect of tracing and profiling at the same time increases the overhead substantially.

Next, we analyzed the traces generated during these experiments to determine why certain workloads have a significant overhead, whereas others have practically none. By opening up the trace in Trace Compass, it was possible to measure statistics on the runtime of the different compute kernel types. Overall, the time duration of the compute kernel was very stable. It is easy to see in Figure 4.6b that the program spends a lot of time freeing memory and requesting data from the runtime. The GPU is not active most of the time. Profiling has a significant cost by forcing all the compute kernels to be run sequentially. When observing the trace where profiling is not run in Figure 4.6a, it is clear that the overhead is significantly lower. The GPU is running most of the time, and often multiple compute kernels are running simultaneously.



(a) Tracing only (b) Tracing and Profiling

Figure 4.6 Trace Compass comparison between tracing only and tracing with profiling enabled

4.4.2 Use case analysis

We tested our tools on one use case to prove they actually highlight performance defects in heterogeneous systems. To this end, we selected the Needleman Wunsch (nw) program of the rodinia benchmark. This program is used to align protein or nucleotides sequences. We ran the tracing and profiling tools with the P1 configuration in Table 4.2. Then we analyzed the results.

The runtimes recorded in Table 4.2 are the average of 20 executions of the program. Looking at the trace of P1 in Trace Compass, we can see that 200ms is spent transfering memory to and from the GPU and 160ms for the actual algorithm. This suggests that we can gain execution time by improving the setup of the program. For the scope of this study, we will focus on the parameters of the NW program to improve its runtime. We looked at the derived metric "VALUUtilization" that shows in percentage how many vector threads are active inside a wavefront. This value is quite low for P1, at 38%. It can be low for two reasons : there is a lot of thread divergence or the work-group size is not adapted.

Because the work-group size is not a multiple of 64 which is the size of a wavefront on the GPU of our machine (Vega 56). To improve the runtime, we changed the work-group size to 64 so that the work-group size is adapted to our GPU. Then we ran the nw program and the runtime is 8.62% less with the new parameters (experiment P2). However, when looking at the L2 cache hit rate metric, the nw program has a lower hit rate than before with the new parameters. This indicates that there is further room for improvement. When testing with larger sequence sizes, the runtime improvement is much more obvious, dividing the runtime almost by two.

This use case shows that derived metrics are very important when looking at optimising the GPU compute time. However, depending on the parameters, the performance defects can be

located elsewhere in the program.

Experiment	Work-group size	Size of sequence	Penalty value	Runtime (s)	Runtime SD	VALUUtilization	L2CacheHit
P1	32	8192	10	0.8682	0.00570	38.00	18.04
P2	64	8192	10	0.7934	0.00756	87.50	7.58
P3	16	16384	10	4.006	0.01038		
P4	64	16384	10	2.385	0.00512		

Tableau 4.2 Experimental results for Needleman Wunsch

4.5 Conclusion

In this paper, we proposed a comprehensive profiling and tracing framework for CPU-GPU programs. The proposed framework relies on ROC-profiler and Trace Compass. This tool offers some of the same functionality as Nsight but brings several new capabilities and opens up more avenues for extensions. All the libraries and software modules used are easily extendable through open source development and the code is readily available.

More detailed information is made available through this framework as compared to previous systems such as LTTng-HSA. However, for specific programs, we have seen that the impact on the program runtime can be significant, because of the constraints associated with data collection. As part of future work, it will be interesting to further reduce the data collection overhead, notably using a different more compact trace format for rocprofiler and roctracer. In addition, it will be interesting to add more data sources, combining and synchronizing the operating system level, runtime level, user level and GPU level traces from several nodes in distributed parallel applications. This is possible using the network tracepoints and exploiting the synchronization algorithms already implemented in Trace Compass [151]. The critical path analysis, also available within Trace Compass [89], could be enhanced to handle distributed heterogeneous CPU-GPU applications. This will help users to analyze and get the global picture for the performance and behaviour of distributed programs.

4.6 Acknowledgements

We would like to thank Advanced Micro Devices (AMD) for providing the hardware and technical advice that made this research possible as well as EfficiOS, Ericsson, The Natural Sciences and Engineering Research Council of Canada (NSERC) for funding this project.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Retour sur les résultats obtenus

5.1.1 Évaluation du surcout de la solution proposée

Le banc d'essai utilisé est Rodinia [41]. Ce banc d'essai est éprouvé et a été conçu pour les systèmes hétérogènes. Il a été mis à jour plusieurs fois et de nombreux programmes ont été ajoutés. Comme le traçage s'effectue sur les commandes effectuées par l'utilisateur, on a pu observer que le surcout est proportionnel au nombre de fonctions kernel envoyées. De plus, ce banc d'essai est très représentatif des différentes charges de travail qu'on peut avoir sur un GPU, d'après Ryoo et al. [44].

D'autres bancs d'essai plus récents existent. Certains, comme Parboil [152], Mars [153], GPGPU-sim [137] n'ont pas été choisi, car ils n'ont pas été jugé suffisamment variés, et donc pas représentatifs des différentes charges de travail par Ryoo et al. [44]. Ensuite d'autres bancs d'essai comme Nupar [43] sont spécifiquement créés pour évaluer les GPUs et donc ne répondent pas vraiment à nos besoins. Enfin SHOC [39] et Valar [154] n'ont pas été évalués pour vérifier leur capacité à se différencier de Rodinia.

Il aurait été intéressant de comparer les résultats obtenus avec Rodinia à d'autres bancs d'essai. De plus, Rodinia a évolué, mais certaines recherches préfèrent utiliser une approche mixte en mélangeant différents bancs d'essai [31]. Cette approche n'a pas été prise et aurait pu apporter plus de variabilité dans les programmes du banc d'essai.

5.2 Adaptabilité de la solution

ROCm permet l'exécution de programmes écrits avec OpenCL, OpenMP et HIP. Cette compatibilité est très importante pour adapter l'approche de traçage et de profilage. Cependant, les programmes utilisant CUDA nécessitent des modifications, car ROCm ne supporte qu'un sous-ensemble des commandes CUDA. ROCm évolue très rapidement et des outils existent pour transformer des programmes du langage CUDA au langage HIP automatiquement. Ces outils ont encore quelques lacunes, cependant le langage HIP assure une compatibilité avec CUDA et le matériel Nvidia.

Cette approche est aussi compatible avec tous les coprocesseurs supportant le fonctionnement avec HSA. Comme la chaine d'outils utilise l'interface HSA pour récolter les informations de profilage, n'importe quel matériel qui utilise cette interface est compatible. De plus, le mécanisme de traçage ne change pas, car il ne fait que remplacer les fonctions HSA. Si la plateforme étudiée utilise une interface de programmation similaire, il est donc possible d'adapter l'approche, en modifiant les fonctions tracées par cette nouvelle interface. Le profilage nécessite que la plateforme ait des compteurs de performance, mais dès que ceux-ci sont accessibles à travers l'interface de programmation, le profilage est possible. De plus, les visualisations produites lors de ce travail peuvent être alimentées avec d'autres sources de données dès que le nouveau système utilise des files d'attente utilisateurs.

5.3 Comparaison avec d'autres approches

Dans cette partie, nous ferons une comparaison entre notre approche et différents outils. Nous verrons pourquoi il est difficile de comparer les outils existants, et quels sont les avantages et désavantages de ces outils par rapport à notre approche.

5.3.1 CodeXL et LTTng-HSA

CodeXL et LTTng-HSA utilisent la même plateforme que notre outil, ROCm. Cependant, ils utilisaient une version antérieure. Cette version antérieure ne répondait pas à la même version du standard HSA. Donc CodeXL et LTTng-HSA ne fonctionnent pas avec la version actuelle de ROCm. Il était ainsi impossible de tester ces deux outils. Quelques tests de surcout avaient été effectués pour LTTng-HSA, mais ceux-ci n'ont été faits que sur un programme qui n'est pas représentatif d'un réel banc d'essai. Cependant, d'après ces essais, LTTng-HSA a un surcout plus faible comparé à notre approche.

Nous pouvons donc dire que LTTng-HSA a un surcout plus faible que notre approche, mais possède de nombreuses limitations quant à la quantité d'informations que nous pouvons obtenir. Ce surcout plus faible peut être exploité pour recueillir des informations préliminaires avant d'appliquer notre méthode. En revanche, il faut noter qu'il est difficile de garder LTTng-HSA à jour avec ROCm, car cette plateforme évolue très vite et il est nécessaire de mettre à jour régulièrement les outils qui interagissent directement avec l'exécution de ROCm. Ce n'est pas le cas avec notre approche, car ROC-profiler et ROC-tracer garantissent de garder un format similaire entre les différentes versions. Cela permet d'avoir une approche qui reste compatible dans le futur.

5.3.2 Nsight et systèmes basés sur CUPTI

Nsight est un système assez particulier, car il permet à d'autres outils d'accéder à la librairie CUPTI qui donne accès à toutes les données de profilage. Donc la plupart des outils qui offrent la capacité de profiler le matériel Nvidia utilisent cette interface.

Ces interfaces n'utilisent pas un système de files d'attente utilisateur donc il est difficile de le comparer avec notre approche utilisant ROCm. De plus, il n'existe pas d'analyse de surcout pour cette méthode. En conséquence, il est difficile de comparer Nsight et notre approche.

Nsight possède de nombreuses fonctionnalités et est l'un des outils les plus complets pour les programmes s'exécutant sur des GPUs. En effet, il est possible d'obtenir les compteurs de performance ainsi que les traces des programmes s'exécutant avec CUDA. Cependant, il ne supporte pas les files d'attente utilisateurs qui permettent un contrôle de l'exécution des fonctions kernel avec une granularité très fine. D'autre part, Nsight permet une analyse de la mémoire et un affichage des états de chaque fil durant l'exécution. Notre approche s'effectue uniquement après l'exécution du programme.

5.3.3 TAU

TAU permet de profiler les processeurs graphiques récents avec PAPI. Il utilisait donc CUPTI pour faire l'interface entre PAPI et Nsight. Récemment, ce système a été adapté pour supporter ROCm. Cependant, ce support utilise ROC-profiler. Donc notre approche utilise les mêmes mécanismes. Le surcout devrait être équivalent pour les deux approches.

TAU permet d'instrumenter les fonctions de l'interface de programmation pour un programme qui s'exécute sur GPU. Il permet aussi de visualiser les valeurs des compteurs de performance après l'exécution de ce programme. Les fonctionnalités sont donc similaires à notre approche. Cependant, il n'est pas possible pour cet outil de différencier les différentes interfaces de programmation disponibles. Donc il est difficile de produire des visualisations qui sont faciles à interpréter par la suite.

CHAPITRE 6 CONCLUSION

Nous présenterons d'abord une synthèse des travaux de cette recherche. Ensuite, nous examinerons les limites de l'approche et critiquerons certains aspects de la solution. Enfin, nous proposerons des améliorations possibles et des travaux supplémentaires à effectuer.

6.1 Synthèse des travaux

Durant ce projet de recherche, nous avons étudié les travaux et les outils déjà présents pour atteindre nos objectifs. Le travail fait par LTTng-HSA présentait plusieurs limitations, et les vues proposées étaient insuffisantes. Nous avons donc choisi d'utiliser ROC-tracer et ROC-profiler qui proposaient les mêmes fonctionnalités, mais sans les limitations de LTTng-HSA. En effet, il est possible de tracer et profiler et d'obtenir toutes les données en un seul passage. Enfin, nous avons décidé d'utiliser TraceCompass pour analyser les données, car c'est un logiciel à source ouverte qui permet facilement d'être modifié pour supporter des sources supplémentaires.

Le traçage s'effectue grâce à un remplacement des pointeurs des fonctions de l'interface à tracer. Ce mécanisme étant performant et flexible, il peut être modifié au besoin si nous avons besoin d'adapter l'outil. Le profilage des fonctions kernels se fait grâce à l'interface HSA. Cela permet d'avoir un mécanisme flexible, relativement générique, avec l'agent utilisé au lieu de passer par un gestionnaire de périphériques dans le noyau Linux. Comme HSA est un mécanisme où la majorité des appels sont dans le contexte utilisateur, cela permet aussi de gagner en performance.

L'analyse est faite grâce à une machine à états qui lit les évènements dans l'ordre. Les traces n'étant pas toujours ordonnées, l'analyse n'est pas très performante pour des fichiers de trace qui contiennent plusieurs millions d'évènements. Cette analyse produit deux vues : la première contient les évènements HSA, HIP, transferts de mémoire et exécutions de fonctions kernels. La seconde montre l'évolution des compteurs de performance qui sont mis à jour après chaque exécution de fonction kernel.

Nous avons donc proposé des améliorations significatives à l'approche LTTng-HSA, notamment de permettre en une seule passe le traçage de tous les évènements d'un programme et le profilage. De plus, les vues permettent d'avoir une meilleure vue d'ensemble du système et de son fonctionnement. Cependant, cette approche possède quelques lacunes au niveau de la performance. De plus, les informations maintenant disponibles permettent de réaliser de

nombreuses analyses additionnelles dans le futur.

6.2 Limitations de la solution proposée

Les compteurs de performance ne sont pas suffisamment exploités. Comme nous l'avons vu lors de la revue de littérature, ces compteurs sont très souvent utilisés pour modéliser l'exécution d'une fonction kernel sur un GPU. Ces compteurs représentent donc un excellent moyen pour déterminer des défauts de performance pour les fonctions kernels.

La vue des compteurs de performance est limitée dans son usage, car il est possible uniquement de comparer la valeur des compteurs entre deux exécutions. Cette vue ne permet pas de voir la valeur numérique du compteur. De plus cette vue propose une visualisation synchronisée avec le temps, ce qui n'est pas forcément très pertinent, car la valeur de ces compteurs varie peu si la fonction kernel exécutée est la même. Il serait donc plus pertinent d'afficher une vue indépendante du temps.

Nous avons vu que le traçage et le profilage impactent significativement le temps d'exécution. Cet impact est proportionnel au nombre de fonctions kernels exécutées. Cet impact est d'autant plus important si l'on effectue le traçage et le profilage en même temps. Comme le profilage oblige une exécution séquentielle et le traçage a un surcout à cause de l'enregistrement des évènements, la combinaison des deux peut ralentir l'exécution. Cependant, les programmes comme *hybridsort* ayant des fonctions kernels relativement longues limitent beaucoup cet impact.

L'analyse repose sur un format texte pour passer à travers les évènements. De plus ce format n'impose pas que les évènements soient ordonnés. Ces deux défauts rendent l'analyse très peu performante.

6.3 Améliorations futures

Notre approche peut inclure des analyses supplémentaires qui utilisent les compteurs de performance. En effet, nous avons vu dans la revue de littérature le modèle *roofline* qui permet de mesurer si un programme est borné par la bande passante ou par le nombre d'instructions par seconde. Il est possible aussi d'implémenter le modèle TMAM (Top-down MicroArchitecture Model). Ces modèles sont donc des améliorations importantes à réaliser, car elles ne sont pas très répandues pour les GPUs.

Pour l'instant il est possible de tracer le noyau d'exploitation et le programme HSA en même temps. Cependant, les traces résultantes ne peuvent pas encore être synchronisées. Il sera

donc intéressant de pouvoir synchroniser ces traces dans le futur pour pouvoir proposer une vue plus générale que celle présente en ce moment.

Le calcul réparti est aussi très utilisé dans le domaine de la haute performance. Il serait possible de se servir des évènements du système d'exploitation pour synchroniser la visualisation des évènements de coprocesseurs de différents ordinateurs. Ceci permettrait de supporter les calculs utilisant des technologies comme MPI, ou n'importe quelle technologie qui permet le calcul réparti sur plusieurs noeuds.

La solution étant peu performante, il serait nécessaire d'optimiser l'approche pour avoir un surcout beaucoup plus faible. L'utilisation de l'API de ROC-profiler permettrait d'activer le profilage uniquement pour une seule exécution de fonction kernel. De cette façon, il est possible d'obtenir les compteurs de performance sans trop dégrader le temps d'exécution. Comme les compteurs de performance ne varient pas beaucoup d'une exécution à l'autre, il suffirait de prendre quelques échantillons pour avoir les données suffisantes pour les analyses ultérieures.

RÉFÉRENCES

[1] Advanced Micro Devices Inc., "AMD Graphics Cores Next (GCN) Architecture," 2012.

[2] D. Shen, X. Liu et F. X. Lin, "Characterizing emerging heterogeneous memory," dans *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2016. New York, NY, USA : Association for Computing Machinery, 2016, p. 13–23. [En ligne]. Disponible : https://doi.org/10.1145/2926697.2926702

[3] S. Mittal et J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, n°. 4, juill. 2015. [En ligne]. Disponible : https://doi.org/10.1145/2788396

[4] L.-W. Chang, J. Gómez-Luna, I. El Hajj, S. Huang, D. Chen et W.-m. Hwu, "Collaborative computing for heterogeneous integrated systems," dans *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA : Association for Computing Machinery, 2017, p. 385–388. [En ligne]. Disponible : https://doi.org/10.1145/3030207.3030244

[5] D. A. Jamshidi, M. Samadi et S. Mahlke, "D2ma : Accelerating coarse-grained data transfer for gpus," dans *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014, p. 431–442.

[6] K. Koukos, A. Ros, E. Hagersten et S. Kaxiras, "Building heterogeneous unified virtual memories (uvms) without the overhead," *ACM Trans. Archit. Code Optim.*, vol. 13, n°. 1, mars 2016. [En ligne]. Disponible : https://doi.org/10.1145/2889488

[7] M. Bauer, H. Cook et B. Khailany, "Cudadma : Optimizing gpu memory bandwidth via warp specialization," dans *SC '11 : Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, p. 1–11.

[8] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey et J. Dongarra, "Gpu-aware non-contiguous data movement in open mpi," dans *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA : Association for Computing Machinery, 2016, p. 231–242. [En ligne]. Disponible : https://doi.org/10.1145/2907294.2907317

[9] R. Ausavarungnirun, K. K. Chang, L. Subramanian, G. H. Loh et O. Mutlu, "Staged memory scheduling : Achieving high performance and scalability in heterogeneous systems," dans *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, p. 416–427.

[10] W. Li, G. Jin, X. Cui et S. See, “An evaluation of unified memory technology on nvidia gpus,” dans *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, p. 1092–1098.

[11] D. Ganguly, Z. Zhang, J. Yang et R. Melhem, “Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory,” dans *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 224–235. [En ligne]. Disponible : https://doi.org/10.1145/3307650.3322224

[12] J. Heuschkel, R. Vogel, M. Blöcher et M. Mühlhäuser, “Blow up the cpu chains ! opencl-assisted network protocols,” dans *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, Oct 2018, p. 657–665.

[13] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo et J. Yang, “A framework for memory oversubscription management in graphics processing units,” dans *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 49–63. [En ligne]. Disponible : https://doi.org/10.1145/3297858.3304044

[14] A. Mishra, L. Li, M. Kong, H. Finkel et B. Chapman, “Benchmarking and evaluating unified memory for openmp gpu offloading,” dans *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC’17. New York, NY, USA : Association for Computing Machinery, 2017. [En ligne]. Disponible : https://doi.org/10.1145/3148173.3148184

[15] W. Li, G. Jin, X. Cui et S. See, “An evaluation of unified memory technology on nvidia gpus,” dans *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, ser. CCGRID ’15. IEEE Press, 2015, p. 1092–1098. [En ligne]. Disponible : https://doi.org/10.1109/CCGrid.2015.105

[16] M. Dashti et A. Fedorova, “Analyzing memory management methods on integrated cpu-gpu systems,” dans *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2017. New York, NY, USA : Association for Computing Machinery, 2017, p. 59–69. [En ligne]. Disponible : https://doi.org/10.1145/3092255.3092256

[17] K. Koukos, A. Ros, E. Hagersten et S. Kaxiras, “Building heterogeneous unified virtual memories (uvms) without the overhead,” *ACM Trans. Archit. Code Optim.*, vol. 13, n°. 1, mars 2016. [En ligne]. Disponible : https://doi.org/10.1145/2889488

[18] S. Chien, I. Peng et S. Markidis, "Performance evaluation of advanced features in cuda unified memory," dans *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, Nov 2019, p. 50–57.

[19] K. Y. Kim et W. Baek, "Blpp : Improving the performance of gpgpus with heterogeneous memory through bandwidth- and latency-aware page placement," dans *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, p. 358–365.

[20] Q. Yu, B. Childers, L. Huang, C. Qian et Z. Wang, "A quantitative evaluation of unified memory in gpus," *The Journal of Supercomputing*, 11 2019.

[21] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi et D. Kaeli, "Umh : A hardware-based unified memory hierarchy for systems with multiple discrete gpus," *ACM Trans. Archit. Code Optim.*, vol. 13, n°. 4, déc. 2016. [En ligne]. Disponible : https://doi.org/10.1145/2996190

[22] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans et O. Villa, "Combining hw/sw mechanisms to improve numa performance of multi-gpu systems," 10 2018, p. 339–351.

[23] K. Ranganath, A. Abdolrashidi, S. L. Song et D. Wong, "Speeding up collective communications through inter-gpu re-routing," *IEEE Computer Architecture Letters*, vol. 18, n°. 2, p. 128–131, July 2019.

[24] M. Harris, "Nvidia dgx-1 : The fastest deep learning system," Jun 2018. [En ligne]. Disponible : https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system/

[25] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent et K. J. Barker, "Evaluating modern gpu interconnect : Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, n°. 1, p. 94–110, Jan 2020.

[26] Y. Thoma, A. Dassatti, D. Molla et E. Petraglio, "Fpga-gpu communicating through pcie," *Microprocessors and Microsystems*, vol. 39, n°. 7, p. 565 – 575, 2015. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0141933115000150

[27] AMD, ARM, Codeplay, General Processor, Imagination, Mediatek, MultiCore Ware, National Taiwan University, Northeastern University, Qualcomm, Rice University, Samsung Electronics, Sandia National Laboratories, SUSE LLC et Via Alliance Technologies, *HSA Runtime Programmer's Reference Manual Version 1.2*. HSA Foundation, 2018.

[28] "Cuda c programming guide." [En ligne]. Disponible : https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[29] T. Sorensen et A. F. Donaldson, "The hitchhiker's guide to cross-platform opencl application development," dans *Proceedings of the 4th International Workshop on OpenCL*,

ser. IWOCL '16. New York, NY, USA : Association for Computing Machinery, 2016. [En ligne]. Disponible : https://doi.org/10.1145/2909437.2909440

[30] J. Li, C. Kuan, T. Wu et J. K. Lee, "Enabling an opencl compiler for embedded multicore dsp systems," dans *2012 41st International Conference on Parallel Processing Workshops*, Sep. 2012, p. 545–552.

[31] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan et D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," dans *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, p. 209–218.

[32] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari et D. Kaeli, "A comprehensive performance analysis of hsa and opencl 2.0," dans *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, p. 183–193.

[33] S. Puthoor, A. M. Aji, S. Che, M. Daga, W. Wu, B. M. Beckmann et G. Rodgers, "Implementing directed acyclic graphs with the heterogeneous system architecture," dans *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16. New York, NY, USA : Association for Computing Machinery, 2016, p. 53–62. [En ligne]. Disponible : https://doi.org/10.1145/2884045.2884052

[34] T. T. Dao, J. Kim, S. Seo, B. Egger et J. Lee, "A performance model for gpus with caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, n°. 7, p. 1800–1813, July 2015.

[35] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal et A. Cristal, "A machine learning approach for performance prediction and scheduling on heterogeneous cpus," dans *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2017, p. 121–128.

[36] F. Khorshahiyan, S. . Shekofteh et H. Noori, "Predicting execution time of cuda kernels with unified memory capability," dans *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, Oct 2019, p. 437–443.

[37] N. Boran, D. Yadav et R. Iyer, *Performance Modelling and Dynamic Scheduling on Heterogeneous-ISA Multi-core Architectures*. Springer Singapore, 08 2019, p. 702–715.

[38] K. Dev et S. Reda, "Scheduling challenges and opportunities in integrated cpu+gpu processors," dans *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia*, ser. ESTIMedia'16. New York, NY, USA : Association for Computing Machinery, 2016, p. 78–83. [En ligne]. Disponible : https://doi.org/10.1145/2993452.2994307

[39] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju et J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," dans *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA : Association for Computing Machinery, 2010, p. 63–74. [En ligne]. Disponible : https://doi.org/10.1145/1735688.1735702

[40] X. Yan, X. Shi et Q. Sun, "An opencl micro-benchmark suite for gpus and cpus," dans *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dec 2012, p. 53–58.

[41] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee et K. Skadron, "Rodinia : A benchmark suite for heterogeneous computing," dans *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, p. 44–54.

[42] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas et D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," dans *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, p. 1–10.

[43] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry et D. Kaeli, "Nupar : A benchmark suite for modern gpu architectures," dans *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA : Association for Computing Machinery, 2015, p. 253–264. [En ligne]. Disponible : https://doi.org/10.1145/2668930.2688046

[44] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song et L. K. John, "Gpgpu benchmark suites : How well do they sample the performance spectrum ?" dans *2015 44th International Conference on Parallel Processing*, Sep. 2015, p. 320–329.

[45] V. Garcıa, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade et A. J. Pena, "Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications," dans *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, p. 1–10.

[46] L. Feng, S. Sinha, W. Zhang et Y. Liang, "Camas : Static and dynamic hybrid cache management for cpu-fpga platforms," dans *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, p. 165–172.

[47] M. Kiani et A. Rajabzadeh, "Efficient cache performance modeling in gpus using reuse distance analysis," *ACM Trans. Archit. Code Optim.*, vol. 15, n°. 4, déc. 2018. [En ligne]. Disponible : https://doi.org/10.1145/3291051

[48] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir et C. R. Das, "Controlled kernel launch for dynamic parallelism in gpus," dans *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, p. 649–660.

[49] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu et C. R. Das, "Managing gpu concurrency in heterogeneous architectures," dans *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, p. 114–126.

[50] J. A. Jablin, T. B. Jablin, O. Mutlu et M. Herlihy, "Warp-aware trace scheduling for gpus," dans *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014, p. 163–174.

[51] S. Lee et C. Wu, "Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference," dans *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017, p. 43–53.

[52] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen et W.-m. Hwu, "Analysis and modeling of collaborative execution strategies for heterogeneous cpu-fpga architectures," dans *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA : Association for Computing Machinery, 2019, p. 79–90. [En ligne]. Disponible : https://doi.org/10.1145/3297663.3310305

[53] F. Zhang, J. Zhai, B. He, S. Zhang et W. Chen, "Understanding co-running behaviors on integrated cpu/gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, nº. 3, p. 905–918, 2017.

[54] C. Nugteren, G. van den Braak, H. Corporaal et H. Bal, "A detailed gpu cache model based on reuse distance theory," dans *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, p. 37–48.

[55] S. Dublish, V. Nagarajan et N. Topham, "Evaluating and mitigating bandwidth bottlenecks across the memory hierarchy in gpus," dans *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, p. 239–248.

[56] T. Tang, X. Yang et Y. Lin, "Cache miss analysis for gpu programs based on stack distance profile," dans *2011 31st International Conference on Distributed Computing Systems*, June 2011, p. 623–634.

[57] S. . Shekofteh, H. Noori, M. Naghibzadeh, H. Fröning et H. S. Yazdi, “ccuda : Effective co-scheduling of concurrent kernels on gpus,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, n°. 4, p. 766–778, 2020.

[58] T. Li, V. K. Narayana et T. El-Ghazawi, “Symbiotic scheduling of concurrent gpu kernels for performance and energy optimizations,” dans *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF ’14. New York, NY, USA : Association for Computing Machinery, 2014. [En ligne]. Disponible : https://doi.org/10.1145/2597917.2597925

[59] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu et C. R. Das, “Scheduling techniques for gpu architectures with processing-in-memory capabilities,” dans *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT ’16. New York, NY, USA : Association for Computing Machinery, 2016, p. 31–44. [En ligne]. Disponible : https://doi.org/10.1145/2967938.2967940

[60] B. Wang, Y. Zhu et W. Yu, “Oaws : Memory occlusion aware warp scheduling,” dans *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, p. 45–55.

[61] A. Sethia, D. A. Jamshidi et S. Mahlke, “Mascar : Speeding up gpu warps by reducing memory pitstops,” dans *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, p. 174–185.

[62] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir et O. Mutlu, “Exploiting inter-warp heterogeneity to improve gpgpu performance,” dans *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, p. 25–38.

[63] J. Anantpur et R. Govindarajan, “Pro : Progress aware gpu warp scheduling algorithm,” dans *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, p. 979–988.

[64] T. G. Rogers, M. O’Connor et T. M. Aamodt, “Divergence-aware warp scheduling,” dans *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, p. 99–110.

[65] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang et W.-M. W. Hwu, “Adaptive cache bypass and insertion for many-core accelerators,” dans *Proceedings of International Workshop on Manycore Embedded Systems*, ser. MES ’14. New York, NY, USA : Association for Computing Machinery, 2014, p. 1–8. [En ligne]. Disponible : https://doi.org/10.1145/2613908.2613909

[66] K. Y. Kim, J. Park et W. Baek, “Iacm : Integrated adaptive cache management for high-performance and energy-efficient gpgpu computing,” dans *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, p. 380–383.

[67] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari et H. Zhou, “Locality-driven dynamic gpu cache bypassing,” dans *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS ’15. New York, NY, USA : Association for Computing Machinery, 2015, p. 67–77. [En ligne]. Disponible : https://doi.org/10.1145/2751205.2751237

[68] X. Zhu, R. Wernsman et J. Zambreno, “Improving first level cache efficiency for gpus using dynamic line protection,” dans *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA : Association for Computing Machinery, 2018. [En ligne]. Disponible : https://doi.org/10.1145/3225058.3225104

[69] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio et J. M. Garcıa, “Adaptive selection of cache indexing bits for removing conflict misses,” *IEEE Transactions on Computers*, vol. 64, nº. 6, p. 1534–1547, 2015.

[70] Chun-Ho Kim, Yeon-Ho Im et Lee-Sup Kim, “Miss-rate reduction in texture cache by adaptive cache indexing,” *Electronics Letters*, vol. 40, nº. 10, p. 597–598, 2004.

[71] B. Wang, Z. Liu, X. Wang et W. Yu, “Eliminating intra-warp conflict misses in gpu,” dans *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, p. 689–694.

[72] A. Abel et J. Reineke, “Reverse engineering of cache replacement policies in intel microprocessors and their evaluation,” dans *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, p. 141–142.

[73] L. R. Hsu, S. K. Reinhardt, R. Iyer et S. Makineni, “Communist, utilitarian, and capitalist cache policies on cmps : Caches as a shared resource,” dans *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’06. New York, NY, USA : Association for Computing Machinery, 2006, p. 13–22. [En ligne]. Disponible : https://doi.org/10.1145/1152154.1152161

[74] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole et C. Lamb, “Parallel performance measurement of heterogeneous parallel systems with gpus,” dans *2011 International Conference on Parallel Processing*, Sep. 2011, p. 176–185.

[75] T. Ball et J. R. Larus, “Optimally profiling and tracing programs,” *ACM Trans. Program. Lang. Syst.*, vol. 16, nº. 4, p. 1319–1360, juill. 1994. [En ligne]. Disponible : https://doi.org/10.1145/183432.183527

[76] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston et B. Chen, "Architecture of systemtap : a linux trace/probe tool," 2005.

[77] B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*, ser. Oracle Solaris Series. Pearson Education, 2011. [En ligne]. Disponible : https://books.google.ca/books?id=jseJ56fUjJgC

[78] T. Bird, "Measuring function duration with ftrace," dans *Proceedings of the Linux Symposium*. Citeseer, 2009, p. 47–54.

[79] M. Gebai et M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead," *ACM Comput. Surv.*, vol. 51, n°. 2, mars 2018. [En ligne]. Disponible : https://doi.org/10.1145/3158644

[80] D. Couturier et M. Dagenais, "Lttng clust : A system-wide unified cpu and gpu tracing tool for opencl applications," *Advances in Software Engineering*, vol. 2015, p. 1–14, 08 2015.

[81] P. Margheritta et M. Dagenais, "Lttng-hsa : Bringing lttng tracing to hsa-based gpu runtimes," *Concurrency and Computation : Practice and Experience*, 04 2019.

[82] B. Buck et J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, n°. 4, p. 317–329, nov. 2000. [En ligne]. Disponible : https://doi.org/10.1177/109434200001400404

[83] B. Welton et B. P. Miller, "Diogenes : Looking for an honest cpu/gpu performance measurement tool," dans *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA : Association for Computing Machinery, 2019. [En ligne]. Disponible : https://doi.org/10.1145/3295500.3356213

[84] G. Stoker et J. K. Hollingsworth, "Towards a methodology for deliberate sample-based statistical performance analysis," dans *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, p. 1258–1265.

[85] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan et S. Yalamanchili, "Lynx : A dynamic instrumentation system for data-parallel applications on gpgpu architectures," dans *2012 IEEE International Symposium on Performance Analysis of Systems Software*, 2012, p. 58–67.

[86] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili et K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," dans *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA : Association for Computing Machinery, 2011. [En ligne]. Disponible : https://doi.org/10.1145/1964179.1964192

[87] N. Farooqui, K. Schwan et S. Yalamanchili, “Efficient instrumentation of gpgpu applications using information flow analysis and symbolic execution,” dans *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA : Association for Computing Machinery, 2014, p. 19–27. [En ligne]. Disponible : https://doi.org/10.1145/2588768.2576782

[88] N. Farooqui, I. Roy, Y. Chen, V. Talwar et K. Schwan, “Accelerating graph applications on integrated gpu platforms via instrumentation-driven optimizations,” dans *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF ’16. New York, NY, USA : Association for Computing Machinery, 2016, p. 19–28. [En ligne]. Disponible : https://doi.org/10.1145/2903150.2903152

[89] F. Giraldeau et M. Dagenais, “Wait analysis of distributed systems using kernel tracing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, nº. 8, p. 2450–2461, 2016.

[90] M. Kiani et A. Rajabzadeh, “Skerd : Reuse distance analysis for simultaneous multiple gpu kernel executions,” dans *2017 19th International Symposium on Computer Architecture and Digital Systems (CADS)*, Dec 2017, p. 1–6.

[91] M. Kiani et A. Rajabzadeh, “Analyzing data locality in gpu kernels using memory footprint analysis,” *Simulation Modelling Practice and Theory*, vol. 91, p. 102 – 122, 2019. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S1569190X18301849

[92] M. D. Hill et A. J. Smith, “Evaluating associativity in cpu caches,” *IEEE Transactions on Computers*, vol. 38, nº. 12, p. 1612–1630, Dec 1989.

[93] B. Jang, D. Schaa, P. Mistry et D. Kaeli, “Exploiting memory access patterns to improve memory performance in data-parallel architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, nº. 1, p. 105–118, Jan 2011.

[94] Y. Gu et L. Chen, “Dynamically linked mshrs for adaptive miss handling in gpus,” dans *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 510–521. [En ligne]. Disponible : https://doi.org/10.1145/3330345.3330390

[95] X. Xiang, B. Bao, T. Bai, C. Ding et T. Chilimbi, “All-window profiling and composable models of cache sharing,” *SIGPLAN Not.*, vol. 46, nº. 8, p. 91–102, févr. 2011. [En ligne]. Disponible : https://doi.org/10.1145/2038037.1941567

[96] S. Mahlke, T. Moseley, R. Hank, D. Bruening et H. K. Cho, “Instant profiling : Instrumentation sampling for profiling datacenter applications,” dans *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*,

ser. CGO '13. USA : IEEE Computer Society, 2013, p. 1–10. [En ligne]. Disponible : https://doi.org/10.1109/CGO.2013.6494982

[97] Google et VMware, "Dynamorio," 2020. [En ligne]. Disponible : https://dynamorio.org/

[98] S. Williams, A. Waterman et D. Patterson, "Roofline : an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, nº. 4, p. 65–76, 2009.

[99] A. Ilic, F. Pratas et L. Sousa, "Cache-aware roofline model : Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, nº. 1, p. 21–24, 2014.

[100] A. Ilic, F. Pratas et L. Sousa, "Beyond the roofline : Cache-aware power and energy-efficiency modeling for multi-cores," *IEEE Transactions on Computers*, vol. 8353, p. 1–1, 01 2017.

[101] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry et Z. A. Matveev, "Performance analysis with cache-aware roofline model in intel advisor," dans *2017 International Conference on High Performance Computing Simulation (HPCS)*, July 2017, p. 898–907.

[102] N. Denoyelle, B. Goglin, E. Jeannot, A. Ilic et L. A. Sousa, "Modeling non-uniform memory access on large compute nodes with the cache-aware roofline model," *IEEE Transactions on Parallel and Distributed Systems*, p. 1–1, 2018.

[103] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, Intel Corporation, August 2007.

[104] A. Yasin, "A top-down method for performance analysis and counters architecture," dans *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2014, p. 35–44.

[105] AMD, *CodeXL User Guide*, Advanced Micro Devices, Inc. [En ligne]. Disponible : https://github.com/GPUOpen-Tools/CodeXL/releases/download/v2.6/CodeXL_Quick_Start_Guide.pdf

[106] Nvidia, *Nsight Developer tools documentation*, Nvidia. [En ligne]. Disponible : https://docs.nvidia.com/nsight-graphics/UserGuide/index.html

[107] Nvidia, *Cuda Toolkit Documentation*, Nvidia. [En ligne]. Disponible : https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[108] S. Browne, J. Dongarra, N. Garner, G. Ho et P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, nº. 3, p. 189–204, août 2000. [En ligne]. Disponible : https://doi.org/10.1177/109434200001400303

[109] Nvidia, *API Reference Guide for CUPTI*, Nvidia. [En ligne]. Disponible : https://docs.nvidia.com/cupti/Cupti/index.html

[110] D. Terpstra, H. Jagode, H. You et J. Dongarra, *Collecting performance data with PAPI-C.* Springer Berlin Heidelberg, 05 2010, p. 157–173.

[111] S. Shende, A. Malony, A. Morris et P. Beckman, "Performance and memory evaluation using tau," 01 2006.

[112] S. S. Shende et A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, n°. 2, p. 287–311, 2006.

[113] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey et N. R. Tallent, "Hpctoolkit : Tools for performance analysis of optimized parallel programs http ://hpctoolkit.org," *Concurrency and Computation : Practice and Experience*, vol. 22, n°. 6, p. 685–701, avr. 2010.

[114] A. Malony, S. Shende, W. Spear, C. W. Lee et S. Biersdorff, "Advances in the tau performance system," dans *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel et M. M. Resch, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 119–130.

[115] T. U. Dresden, "Vampir," 2020. [En ligne]. Disponible : https://vampir.eu/

[116] B. Mohr, A. D. Malony, S. Shende et F. Wolf, "Design and prototype of a performance tool interface for openmp," *J. Supercomput.*, vol. 23, n°. 1, p. 105–128, août 2002. [En ligne]. Disponible : https://doi.org/10.1023/A:1015741304337

[117] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller et W. E. Nagel, "The vampir performance analysis tool-set," dans *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer et A. Schulz, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 139–155.

[118] I. Kohyarnejadfard, M. Shakeri et D. Aloise, "System performance anomaly detection using tracing data analysis," dans *Proceedings of the 2019 5th International Conference on Computer and Technology Applications*, ser. ICCTA 2019. New York, NY, USA : Association for Computing Machinery, 2019, p. 169–173. [En ligne]. Disponible : https://doi.org/10.1145/3323933.3324085

[119] H. Abbasi, N. Ezzati-Jivan, M. Bellaiche, C. Talhi et M. Dagenais, "Machine learning-based edos attack detection technique using execution trace analysis," *Journal of Hardware and Systems Security*, 01 2019.

[120] P. Mulinka et P. Casas, "Stream-based machine learning for network security and anomaly detection," dans *Proceedings of the 2018 Workshop on Big Data Analytics and*

Machine Learning for Data Communication Networks, ser. Big-DAMA '18. New York, NY, USA : Association for Computing Machinery, 2018, p. 1–7. [En ligne]. Disponible : https://doi.org/10.1145/3229607.3229612

[121] O. Ibidunmoye, F. Hernández-Rodriguez et E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Comput. Surv.*, vol. 48, n°. 1, juill. 2015. [En ligne]. Disponible : https://doi.org/10.1145/2791120

[122] J. Duan, S. Jiang, Q. Yu, K. Lu, X. Zhang et Y. Yao, "An automatic localization tool for null pointer exceptions," *IEEE Access*, vol. 7, p. 153 453–153 465, 2019.

[123] E. C. Sezer, P. Ning, C. Kil et J. Xu, "Memsherlock : An automated debugger for unknown memory corruption vulnerabilities," dans *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA : Association for Computing Machinery, 2007, p. 562–572. [En ligne]. Disponible : https://doi.org/10.1145/1315245.1315314

[124] F. Gao, L. Wang et X. Li, "Bovinspector : Automatic inspection and repair of buffer overflow vulnerabilities," dans *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA : Association for Computing Machinery, 2016, p. 786–791. [En ligne]. Disponible : https://doi.org/10.1145/2970276.2970282

[125] Nvidia, *User Manual for CUDA-GDB*, Nvidia. [En ligne]. Disponible : https://docs.nvidia.com/cuda/cuda-gdb/index.html

[126] Nvidia, *API Reference Guide for the CUDA debugger*, Nvidia. [En ligne]. Disponible : https://docs.nvidia.com/cuda/debugger-api/index.html

[127] G. Gopalakrishnan et J. Sawaya, "Achieving formal parallel program debugging by incentivizing cs/hpc collaborative tool development," dans *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure : Research, Experience, Applications and Models*, ser. SCREAM '15. New York, NY, USA : Association for Computing Machinery, 2015, p. 11–18. [En ligne]. Disponible : https://doi.org/10.1145/2753524.2753531

[128] A. Horga, S. Chattopadhyay, P. Eles et Z. Peng, "Systematic detection of memory related performance bottlenecks in gpgpu programs," *Journal of Systems Architecture*, vol. 71, 08 2016.

[129] X. Liu et B. Wu, "Scaanalyzer : a tool to identify memory scalability bottlenecks in parallel programs," dans *SC '15 : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, p. 1–12.

[130] C. Erb, M. Collins et J. L. Greathouse, “Dynamic buffer overflow detection for gpgpus,” dans *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO ’17. IEEE Press, 2017, p. 61–73.

[131] A. Eizenberg, Y. Peng, T. Pigli, W. Mansky et J. Devietti, “Barracuda : Binary-level analysis of runtime races in cuda programs,” *SIGPLAN Not.*, vol. 52, n°. 6, p. 126–140, juin 2017. [En ligne]. Disponible : https://doi.org/10.1145/3140587.3062342

[132] Y. Peng, V. Grover et J. Devietti, “Curd : A dynamic cuda race detector,” *SIGPLAN Not.*, vol. 53, n°. 4, p. 390–403, juin 2018. [En ligne]. Disponible : https://doi.org/10.1145/3296979.3192368

[133] Zhe Fan, Feng Qiu, A. Kaufman et S. Yoakum-Stover, “Gpu cluster for high performance computing,” dans *SC ’04 : Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, 2004, p. 47–47.

[134] A. K. Ziabari, R. Ubal, D. Schaa et D. Kaeli, “Visualization of opencl application execution on cpu-gpu systems,” dans *Proceedings of the Workshop on Computer Architecture Education*, ser. WCAE ’15. New York, NY, USA : Association for Computing Machinery, 2015. [En ligne]. Disponible : https://doi.org/10.1145/2795122.2795125

[135] F. Schmitt, R. Dietrich et G. Juckeland, “Scalable critical-path analysis and optimization guidance for hybrid mpi-cuda applications,” *The International Journal of High Performance Computing Applications*, vol. 31, n°. 6, p. 485–498, 2017. [En ligne]. Disponible : https://doi.org/10.1177/1094342016661865

[136] J. Power, J. Hestness, M. S. Orr, M. D. Hill et D. A. Wood, “1gem5-gpu : A heterogeneous cpu-gpu simulator.”

[137] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong et T. M. Aamodt, “Analyzing cuda workloads using a detailed gpu simulator, in ieee international,” 2009.

[138] R. Ubal, B. Jang, P. Mistry, D. Schaa et D. Kaeli, “Multi2sim : a simulation framework for cpu-gpu computing,” dans *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, p. 335–344.

[139] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi et D. Kaeli, “Mgpusim : Enabling multi-gpu performance modeling and optimization,” dans *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 197–209. [En ligne]. Disponible : https://doi.org/10.1145/3307650.3322230

[140] G. Malhotra, S. Goel et S. Sarangi, “Gputejas : A parallel simulator for gpu architectures,” *2014 21st International Conference on High Performance Computing, HiPC 2014*, 06 2015.

[141] T. Bertauld et M. Dagenais, “Low-level trace correlation on heterogeneous embedded systems,” *EURASIP Journal on Embedded Systems*, vol. 2017, 12 2017.

[142] B. Gaster, L. Howes et D. Hower, “Chapter 6 - hsa queuing model,” dans *Heterogeneous System Architecture*, W. mei W. Hwu, édit. Boston : Morgan Kaufmann, 2016, p. 77 – 96. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/B9780128003862000055

[143] T. Biswas, P. Kuila et A. Ray, “Multi-level queue for task scheduling in heterogeneous distributed computing system,” dans *International Conference on Advanced Computing and Communication Systems (ICACCS)*, 01 2017, p. 1–6.

[144] S. Puthoor, X. Tang, J. Gross et B. M. Beckmann, “Oversubscribed command queues in gpus,” dans *Proceedings of the 11th Workshop on General Purpose GPUs*, ser. GPGPU-11. New York, NY, USA : Association for Computing Machinery, 2018, p. 50–60. [En ligne]. Disponible : https://doi.org/10.1145/3180270.3180271

[145] B. Poirier, R. Roy et M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events,” *SIGOPS Oper. Syst. Rev.*, vol. 44, n°. 3, p. 75–87, août 2010. [En ligne]. Disponible : https://doi.org/10.1145/1842733.1842747

[146] H. Daoud et M. Dagenais, “Multilevel analysis of the java virtual machine based on kernel and userspace traces,” *Journal of Systems and Software*, vol. 167, p. 110589, 2020. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0164121220300698

[147] Advanced Micro Devices et E. Shcherbakov, “Roc-profiler,” https://github.com/ROCm-Developer-Tools/rocprofiler, 2020.

[148] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg et F. Wolf, “Score-p : A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir,” dans *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel et M. M. Resch, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 79–91.

[149] M. Desnoyers et M. R. Dagenais, “LTTng, Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer,” dans *Linux Foundation Collaboration*

Summit 2009 (LFCS 2009), avr. 2009. [En ligne]. Disponible : /static/publications/desnoyers-lfcs2009-paper.pdf

[150] Advanced Micro Devices et E. Shcherbakov, "Roc-tracer," https://github.com/ROCm-Developer-Tools/roctracer, 2020.

[151] T. Compass, "Trace compass," *En ligne : https ://projects. eclipse. org*, 2015.

[152] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu et W. mei W. Hwu, "Parboil : A revised benchmark suite for scientific and commercial throughput computing," dans *Technical Report IMPACT-12-01.* University of Illinois, 2012.

[153] B. He, W. Fang, Q. Luo, N. K. Govindaraju et T. Wang, "Mars : A mapreduce framework on graphics processors," dans *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA : Association for Computing Machinery, 2008, p. 260–269. [En ligne]. Disponible : https://doi.org/10.1145/1454115.1454152

[154] P. Mistry, Y. Ukidave, D. Schaa et D. Kaeli, "Valar : A benchmark suite to study the dynamic behavior of heterogeneous systems," dans *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA : Association for Computing Machinery, 2013, p. 54–65. [En ligne]. Disponible : https://doi.org/10.1145/2458523.2458529

ANNEXE A RÉSULTATS DU BANC D'ESSAI RODINIA

Experience	Control (s)	SD	Tracing (s)	SD	Profiling (s)	SD	Tracing and profiling (s)	SD
nw	0.8634	0.003262	1.5086	0.011638	1.0353	0.006544	2.3956	0.020424
gaussian	0.3286	0.003557	0.7966	0.010594	0.4328	0.006207	0.8481	0.022446
b+tree	1.3488	0.007351	1.836	0.025387	1.4545	0.005482	1.8742	0.018777
hybridsort	14.4465	0.038518	14.985	0.014726	14.6211	0.009303	15.2632	0.028888
backprop	0.3539	0.00333	0.8237	0.013319	0.4549	0.003315	0.8551	0.013921
streamcluster	4.6653	0.048892	10.2981	0.128327	6.5732	0.030737	20.11	0.282729
kmeans	0.3233	0.0039	0.7986	0.011404	0.4317	0.005292	0.852	0.013225
nn	0.3357	0.00389	0.8003	0.01276	0.4404	0.004985	0.8438	0.013551
heartwall	0.7691	0.003122	1.2672	0.008485	0.8824	0.004748	1.4169	0.007582
dwt2d	0.3259	0.001905	0.7928	0.011557	0.4392	0.004946	0.8725	0.013426
lud	0.3302	0.002827	0.8079	0.015429	0.4489	0.006575	0.893	0.013128
pathfinder	0.5015	0.002202	0.9648	0.016099	0.6104	0.006711	1.0089	0.015741
srad	0.378	0.004405	1.0856	0.010062	0.5951	0.007968	2.1017	0.029571
bfs	0.3306	0.003969	0.8027	0.012421	0.4363	0.005313	0.8619	0.011832
lavaMD	0.5741	0.002142	1.0404	0.017261	0.6743	0.005147	1.071	0.011651
cfd	1.8553	0.009138	6.4362	0.057383	5.3886	0.036907	33.2273	0.324247
particlefilter	0.3464	0.00289	0.8263	0.011749	0.4531	0.004867	0.875	0.010566
hotspot	0.5569	0.004146	1.0376	0.052621	0.6592	0.008362	1.0635	0.013507

Tableau A.1 Temps d'exécution et écarts types des programmes du banc d'essai Rodinia

Experience	Wavefronts	L2CacheHit	VALUInsts	SALUInsts	FlatVMemInsts	LDSInsts	DurationNs	NKernels	NEvents
nw	64.25	38.82	495.98	122.96	34.50	144.00	37089.44	255	22611
gaussian	4.50	11.00	23.00	10.00	4.50	0.00	7407.75	4	8612
b+tree	32000.00	84.00	66.50	59.50	11.50	0.00	176822.50	2	8702
hybridsort	3489326.07	49.57	355808.70	321138.90	17107.10	0.00	115731426.16	120	15753
backprop	16384.00	59.50	281.00	21.50	6.50	7.50	161118.50	2	8528
streamcluster	1106.76	32.28	628.31	313.03	77.06	0.00	161034.68	5549	444822
kmeans	4.00	2.56	668.22	331.33	79.33	0.00	66916.33	9	9183
nn	672.00	2.00	60.00	4.00	1.00	0.00	9927.00	1	8364
heartwall	204.00	70.25	1005405.20	231325.80	111191.35	205308.45	21233310.05	20	11192
dwt2d	196.67	31.93	169.74	46.41	13.70	20.67	12401.26	27	9984
lud	110.78	44.07	545.00	105.70	37.20	299.78	31231.65	46	10888
pathfinder	1852.00	14.00	230.60	334.60	21.60	118.40	43231.40	5	8622
srad	2884.46	16.51	171.43	37.08	15.54	11.35	38204.57	502	39819
bfs	64.00	44.75	61.56	13.31	16.38	0.00	14981.88	16	9557
lavaMD	2000.00	96.00	145599.00	11259.00	221.00	6697.00	57368053.00	1	8418
cfd	2277.01	30.50	404.66	46.12	23.87	0.00	50250.42	16004	914649
particlefilter	16.00	43.44	2173.33	12380.89	5.00	0.00	296984.56	9	9850
hotspot	7396.00	66.00	149.00	83.00	2.00	17.00	90671.00	1	8398

Tableau A.2 Compteurs de performance moyens par kernel du banc d'essai Rodinia

ANNEXE B INSTRUCTIONS POUR EXÉCUTER LE BANC D'ESSAI RODINIA

Les outils utilisés pour exécuter le banc d'essai sont disponibles à l'adresse suivante : https://github.com/arfio/ROCm-test-rodinia. Cette adresse contient les instructions nécessaires pour reproduire les résultats présentés dans ce mémoire. Les fichiers de ce banc d'essai sont tirés du dépôt de code https ://github.com/ROCm-Developer-Tools/HIP-Examples. Certains fichiers ont été ajoutés pour réaliser les expériences et sont décrits ci-après.

Liste des fichiers en suppléments au banc d'essai

- analyze_correlation.py – Produit les graphiques de corrélation entre les compteurs et le surcout pour les trois expériences (traçage, profilage, traçage et profilage).
- display_runtime_overhead.py – Produit le graphique montrant le surcout pour chaque programme du banc d'essai et chacune des trois expériences.
- run_benchmark_profiling.sh – Compile et exécute toutes les expériences.
- collect_stats.py – Collecte les temps d'exécution, calcule les surcouts, les écarts types et rassemble le tout dans un fichier au format csv.
- collect_counters.py – Collecte les compteurs de performance d'une expérience et est exécutée une fois par programme.