Titre: Title:	Traçage logiciel d'applications utilisant un processeur graphique
Auteur: Author:	Paul Margheritta
Date:	2017
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Margheritta, P. (2017). Traçage logiciel d'applications utilisant un processeur graphique [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2838/

UNIVERSITÉ DE MONTRÉAL

TRAÇAGE LOGICIEL D'APPLICATIONS UTILISANT UN PROCESSEUR GRAPHIQUE

PAUL MARGHERITTA
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

TRAÇAGE LOGICIEL D'APPLICATIONS UTILISANT UN PROCESSEUR GRAPHIQUE

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

Mme NICOLESCU Gabriela, Doctorat, présidente
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. BILODEAU Guillaume-Alexandre, Ph. D., membre

REMERCIEMENTS

Je tiens tout d'abord à remercier le professeur Michel Dagenais qui a supervisé ce projet de recherche. Son soutien constant, la qualité de son suivi et son intérêt pour les enjeux techniques et scientifiques ont été précieux tout au long de cette maîtrise.

Je souhaite adresser des remerciements tout particuliers à l'ensemble de mes camarades et amis du laboratoire DORSAL. Leur présence a contribué à apporter une bonne humeur rafraîchissante dans les profondeurs du pavillon Lassonde.

Je voudrais également remercier toutes les personnes qui, au cours de cette maîtrise, ont pu me guider, m'apporter des conseils ou formuler des commentaires concernant mon projet. Plus particulièrement, je remercie les associés de recherche du laboratoire DORSAL, ainsi que les personnes rencontrées sur le site d'AMD à Boston.

Enfin, je tiens à souligner la participation financière d'Ericsson, d'EfficiOS, du Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), de Prompt, ainsi que la contribution d'AMD dans le matériel informatique utilisé.

RÉSUMÉ

Dans le domaine informatique, les architectures matérielles tendent vers un niveau de parallélisme toujours plus élevé, de manière à pouvoir traiter de plus en plus d'informations simultanément. L'émergence des processeurs multicœurs a fait naître de nombreux problèmes liés à la concurrence des algorithmes, ce qui a contribué au développement d'outils performants permettant d'analyser finement l'activité des systèmes parallèles.

Cependant, les outils de diagnostic de problèmes et d'analyse de performance pour les systèmes parallèles restent encore peu adaptés à l'étude des processeurs graphiques (GPU). Cela est dû aux grandes disparités dans les architectures existantes et au très haut niveau de parallélisme qu'il est nécessaire de gérer lorsque l'on traite avec des processeurs graphiques. Il est donc toujours relativement complexe d'analyser l'exécution d'un programme sur processeur graphique, alors même que ces accélérateurs matériels connaissent un gain très important de popularité en raison de leur structure se prêtant bien au calcul de haute performance et à l'apprentissage automatique, entre autres.

Dans le cadre de ce projet de recherche, nous cherchons à exploiter des méthodes de traçage qui ont prouvé leur efficacité pour l'analyse de programmes s'exécutant sur processeur central (CPU) multicœurs, afin d'analyser l'activité d'un ou plusieurs processeurs graphiques au cours de l'exécution d'un programme. L'objectif est, grâce au traçage, de fournir des outils appropriés permettant de détecter de possibles problèmes de performance.

Grâce à LTTng, un traceur performant pour le système d'exploitation GNU/Linux, nous avons créé un utilitaire permettant de générer des traces unifiées rendant compte de l'activité du processeur graphique pour des programmes basés sur l'interface de programmation HSA. Cette interface, commune à plusieurs acteurs majeurs du marché, se donne comme objectif d'accélérer et faciliter la communication entre processeurs de calcul dans un contexte hétérogène, ce qui en fait une plateforme intéressante pour analyser l'activité d'un processeur graphique.

Notre solution, nommée LTTng-HSA, se base sur un ensemble de bibliothèques logicielles pouvant être préchargées à l'exécution d'un programme, de manière à insérer une instrumentation destinée à mieux comprendre le fonctionnement du processeur graphique. Les traces générées au format CTF proposent des informations sur les durées d'exécution des noyaux de calcul, sur les appels de fonctions de l'interface de HSA et donnent accès à diverses métriques venues du processeur graphique. Ce mémoire présente également des solutions de visualisation de ces traces, ainsi que les techniques de fusion et de tri aboutissant à l'obtention d'une

trace unifiée représentant l'activité du processeur graphique.

LTTng-HSA permet d'obtenir des informations utiles sur la performance d'un programme parallèle basé sur HSA, sans avoir à intervenir pour modifier le programme et avec un faible surcoût temporel. C'est donc un outil intéressant pour trouver l'origine de problèmes de performance dans le cadre d'un programme accéléré par matériel.

ABSTRACT

In recent years, computer architectures have become increasingly parallel. More than ever, processors can now carry out multiple operations simultaneously. The rise of multi-core processors has brought a whole new set of concurrency-related problems, but has also helped us design specific analysis tools for parallel systems.

However, many of these tools are focused on a multi-core CPU architecture and are not really adapted to GPU-oriented performance analysis, mostly because of the very high level of parallelism and multiple architectures that need to be taken into account when dealing with GPUs. Therefore, while GPU computing is becoming ubiquitous in fields such as high-performance computing and machine learning, we still lack analysis tools for these hardware accelerators.

This research project is focused on using software tracing methods for GPU analysis purposes. Tracing methods have already proved successful in multi-core CPU contexts. Our main research objective is to provide powerful trace-based tools to help find performance issues for GPU-accelerated programs.

We used LTTng, a highly efficient tracing framework for GNU/Linux systems, to create a tool that can generate unified GPU-oriented traces for HSA-based programs. HSA is a cross-vendor standard that aims to streamline communications between compute devices in a heterogeneous context. Therefore, HSA is a good choice as a platform for GPU performance analysis.

LTTng-HSA, our solution, is a set of libraries that are meant to be preloaded when executing a GPU-accelerated program. Each of these libraries automatically inserts interesting trace-points that allow us to generate CTF traces providing information about HSA API function calls, GPU kernel timestamps or GPU hardware metrics. LTTng-HSA also includes helpful views for these traces. The process that leads to a unified GPU-oriented trace, which involves trace merging and sorting, is explained in this thesis.

LTTng-HSA can easily be used to pinpoint performance issues in HSA-based parallel programs. Moreover, our solution has low overhead and does not require any modification to the program being traced. Therefore, LTTng-HSA is a helpful tool for the analysis of GPU-accelerated software.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AMD	*Advanced Micro Devices*
API	*Application programming interface*
APU	*Accelerated Processing Unit*
AQL	*Architected Queuing Language*
BOINC	*Berkeley Open Infrastructure for Network Computing*
BPF	*Berkeley Packet Filter*
CLOC	*Computing Language Offline Compiler*
C++ AMP	*C++ Accelerated Massive Parallelism*
CPU	*Central processing unit*
CTF	*Common Trace Format*
CUDA	*Compute Unified Device Architecture*
DSP	*Digital signal processor*
eBPF	*Extended Berkeley Packet Filter*
FFmpeg	*Fast Forward Moving Picture Experts Group*
FPGA	*Field programmable gate array*
GCN	*Graphics Core Next*
GDB	*GNU Debugger*
GIMP	*GNU Image Manipulation Program*
GNU	*GNU's Not Unix*
GPA	*GPU Performance API*
GPA	*Graphics Performance Analyzers*
GPU	*Graphics processing unit*
GPGPU	*General-purpose graphics processing unit*
HCC	*Heterogeneous Compute Compiler*
HIP	*Heterogeneous-Compute Interface for Portability*
HSA	*Heterogeneous System Architecture*
kprobe	*Kernel probe*
IGP	*Integrated graphics processor*
LTT	*Linux Trace Toolkit*
LTTng	*Linux Trace Toolkit next generation*
MATLAB	*Matrix Laboratory*
MPI	*Message Passing Interface*
OpenCL	*Open Computing Language*

OpenMP	*Open Multi-Processing*
OTF	*Open Trace Format*
POSIX	*Portable Operating System Interface*
pthread	*POSIX thread*
RCP	*Radeon Compute Profiler*
ROCk	*Radeon Open Compute kernel*
ROCr	*Radeon Open Compute runtime*
SETI	*Search for extraterrestrial intelligence*
SIMD	*Single instruction, multiple data*
TAU	*Tuning and Analysis Utility*
TBB	*Threading Building Blocks*
TraceFS	*Trace File System*
TPU	*Tensor processing unit*
UCT	Unité centrale de traitement
UST	*User space tracer*
USDT	*User Statically Defined Tracing*
VPU	*Visual processing unit*
XML	*Extensible Markup Language*

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Dans de nombreux domaines des affaires et de la recherche scientifique, les ordinateurs sont de plus en plus amenés à devoir traiter de très grands volumes de données dans des temps toujours plus réduits. Ces tendances impliquent la nécessité pour les constructeurs de proposer des architectures de processeurs permettant de paralléliser au maximum les calculs afin de gagner en capacité de traitement. Les marges d'amélioration de la parallélisation sur processeur central (CPU) restant relativement réduites, les processeurs graphiques (GPU) se sont imposés comme les accélérateurs de référence pour ce type de problème grâce à leurs architectures très parallèles. Les processeurs graphiques, initialement conçus pour accélérer la création d'images, sont donc devenus incontournables pour la réalisation de calculs extrêmement variés.

Le développement du calcul sur ces architectures complexes fait cependant apparaître de nouveaux problèmes, avec une performance parfois trop faible ou des résultats qui peuvent être non conformes aux attentes. Alors que de nombreux outils existent pour diagnostiquer de telles erreurs de performance dans l'utilisation du processeur central, il est plus difficile d'analyser l'utilisation d'un ou plusieurs processeurs graphiques par une application. En effet, les processeurs graphiques ont tendance à être considérés comme de simples périphériques plutôt que comme des unités de calcul au même titre qu'un processeur central. De plus, l'évolution rapide et la diversité des architectures proposées pour les processeurs graphiques ne facilitent pas l'émergence d'outils universels d'analyse de performance sur de tels accélérateurs.

Il reste donc des marges d'amélioration importantes dans l'analyse de l'utilisation des processeurs graphiques par des applications.

1.1 Définitions et concepts de base

Dans cette section, nous définissons les concepts de base qui sont pertinents pour la bonne compréhension de ce mémoire.

1.1.1 Processeur central et processeur graphique

Le **processeur central** est l'unité principale de calcul d'un ordinateur. On le désigne également sous le nom de ***central processing unit*** **(CPU)** ou d'**unité centrale de traitement (UCT)**. Un processeur central actuel comporte généralement plusieurs **cœurs**, typiquement quatre ou huit, qui constituent chacun une unité de calcul indépendante permettant

de prendre en charge un **fil d'exécution** à la fois, c'est-à-dire d'exécuter une seule suite d'instructions en même temps.

Un processeur graphique, aussi connu sous les noms de ***graphics processing unit*** **(GPU)**, d'**accélérateur graphique** ou d'**accélérateur matériel**, est un type de processeur traditionnellement utilisé dans un ordinateur, en addition du processeur central, pour accélérer les calculs permettant la production d'images à l'écran. Il est habituellement situé sur une carte électronique désignée sous le nom de **carte graphique**. L'idée principale d'un processeur graphique est de proposer un très grand nombre d'unités de calcul par rapport à un processeur central (jusqu'à plusieurs milliers), permettant ainsi une parallélisation massive des calculs. Sous l'influence de la tendance nommée ***general-purpose graphics processing unit*** **(GPGPU)**, les processeurs graphiques sont couramment utilisés dans le but d'accélérer des calculs dont l'intérêt n'est pas graphique en premier lieu.

Le processeur central communique avec le processeur graphique à l'aide de **files d'attente** (ou **queues**) par lesquelles sont transmises les tâches à effectuer sur l'accélérateur. Une tâche de calcul est généralement décrite sous la forme d'une fonction que le processeur graphique doit exécuter en parallèle sur chacune des sous-parties de l'ensemble de données à traiter définies par l'utilisateur. Dans le cas le plus général, ce type de fonction est appelé **noyau de calcul** ou simplement **noyau**. Dans le cadre de l'utilisation graphique d'un accélérateur matériel, un noyau de calcul est communément désigné sous le nom de **nuanceur** ou ***shader***.

Lorsqu'un processeur central est utilisé en combinaison avec un ou plusieurs processeurs graphiques et, plus théoriquement, d'autres composants programmables de type *field programmable gate array* (FPGA) ou *digital signal processor* (DSP), on parle d'un **système hétérogène**.

1.1.2 Traçage logiciel

Le **traçage logiciel**, ou simplement **traçage**, est une technique d'analyse de systèmes informatiques consistant à enregistrer les événements qui se produisent sur un système, de telle manière que ces informations soient exploitables après coup, afin de détecter les problèmes qui se sont produits. Le traçage se distingue de la simple journalisation d'événements par son orientation plus bas niveau et son extension à l'ensemble du système.

Le traçage peut intervenir dans les deux espaces d'exécution traditionnellement définis : l'espace noyau et l'espace utilisateur. La combinaison du traçage dans ces deux espaces permet d'avoir une vision complète des événements qui se produisent sur le système. Dans chacun des deux espaces, il est attendu que le traçage ait un impact minimal en matière de temps

et de mémoire utilisée afin de ne pas perturber le déroulement habituel de l'exécution.

Afin de permettre le traçage, des **points de trace** sont définis dans le code source du noyau du système d'exploitation ou de l'application à tracer. L'insertion de points de trace à certains endroits bien définis du code source à des fins d'analyse est connue sous le nom d'**instrumentation**. Cette instrumentation peut être réalisée soit manuellement, auquel cas elle est dite **statique**, soit automatiquement au moment de l'exécution, auquel cas on parle d'**instrumentation dynamique**. Au cours de l'exécution, lorsqu'un point de trace est rencontré dans le code source, un **événement de trace** correspondant au point de trace présent est ajouté à la trace avec une estampille de temps. La trace garde donc une liste chronologique de tous les points de trace rencontrés au fur et à mesure de l'exécution.

1.2 Éléments de la problématique

À l'heure actuelle, nous disposons de quelques outils avancés pour la détection de problèmes sur processeur graphique. Cependant, plusieurs défis restent à relever afin de faciliter l'analyse de performance des applications utilisant ces systèmes. Premièrement, les outils proposés sont souvent centrés sur l'analyse de l'exécution du programme sur le processeur graphique seulement, alors que dans le cadre d'un système hétérogène, d'autres types d'appareils peuvent être impliqués ; il s'agit donc de pouvoir analyser l'activité de tous les types de processeurs qui contribuent au calcul. De plus, certains outils actuels se limitent à quelques environnements d'exécution et interfaces de programmation permettant de réaliser du calcul sur processeur graphique, sans proposer de solution à visée universelle. Enfin, parmi les environnements d'exécution et outils d'analyse de performance proposés, plusieurs sont à code source fermé, ce qui rend complexe la compréhension des mécanismes implémentés et l'amélioration des solutions existantes. Nous cherchons donc à concevoir des outils à code source ouvert permettant d'analyser l'exécution d'applications utilisant un ou plusieurs processeurs graphiques dans le cadre d'un système possiblement hétérogène, et devant être utilisables, dans la mesure du possible, avec différents environnements d'exécution.

HSA est un standard permettant la programmation de systèmes hétérogènes, incluant notamment des processeurs centraux et processeurs graphiques, de manière transparente : indépendamment de sa nature, chaque processeur est traité comme une unité de calcul utilisable de la même manière. HSA expose une interface de programmation bas niveau qui, accompagnée de son environnement d'exécution *Radeon Open Compute runtime* (ROCr), dont le code source a été ouvert dans le cadre du projet GPUOpen, permet en particulier d'exécuter des noyaux de calcul sur processeur graphique. Au-delà de ROCr, des implémentations des environnements d'exécution OpenCL et C++ AMP basées sur HSA existent dans le but

d'accélérer l'exécution des programmes correspondants sur les processeurs graphiques optimisés pour HSA. Nous retrouvons donc dans HSA les caractéristiques que nous recherchions précédemment : par sa nature adaptée aux systèmes hétérogènes, sa capacité à servir de base à des environnements d'exécution connus auparavant tels qu'OpenCL, et son orientation vers des outils à code source ouvert, HSA est une plateforme qui se prête bien à l'analyse d'applications exploitant des processeurs graphiques.

Le choix d'une plateforme pour notre analyse doit être complété par le choix de mécanismes permettant la récupération de données pertinentes. Les outils actuels proposent essentiellement des vues représentant diverses métriques utiles pour comprendre l'activité du processeur graphique. En arrière-plan, certains outils se basent explicitement sur du traçage logiciel et d'autres non. Dans tous les cas, la notion de traçage est peu mise en avant, les outils existants préférant se concentrer sur une approche très visuelle. Ce choix a l'avantage de présenter à l'utilisateur des données déjà traitées et arrangées de manière plaisante et facile à observer, dans un cadre de haut niveau. Cependant, l'utilisateur souhaitant aller plus loin dans l'analyse des données, que ce soit en adaptant les vues proposées, en créant de nouvelles vues, ou en voulant traiter des données brutes de traçage, n'y trouvera pas toujours son compte, et pourrait se trouver quelque peu limité par les outils actuels. Pour cette raison, nous souhaitons proposer une solution qui soit principalement centrée sur l'obtention d'une trace logicielle, quitte à proposer par ailleurs des outils visuels pouvant se greffer sur cette trace, tout en laissant à l'utilisateur le maximum de flexibilité.

Nous nous orientons donc vers une solution basée sur l'instrumentation d'environnements d'exécution compatibles avec HSA, dans le but de permettre la production d'une trace rendant compte de l'activité du processeur graphique. Puisque nous travaillons avec le système d'exploitation GNU/Linux, nous faisons le choix d'utiliser LTTng, un des outils de traçage les plus aboutis sur cette plateforme. LTTng permet en effet le traçage dans le noyau Linux (avec LTTng-modules) ainsi que l'espace utilisateur (grâce à LTTng-UST) avec un surcoût minimal, ce qui nous ouvre de nombreuses possibilités tout en garantissant une certaine efficacité. De plus, LTTng génère par défaut des traces au format CTF. CTF est un format binaire capable d'enregistrer des traces de manière compacte ; les traces peuvent ensuite être traduites vers d'autres formats grâce à des outils dédiés tels que Babeltrace. De plus, comme son nom l'indique, CTF a la vocation de servir de standard universel pour la description de traces logicielles. Le logiciel Trace Compass, qui permet l'analyse interactive de traces, y compris au format CTF, nous sera utile pour proposer des vues permettant d'avoir une interprétation graphique des résultats obtenus.

Au cours de nos expériences de traçage de l'activité d'un processeur graphique, plusieurs

enjeux intéressants sont apparus. Il s'agit d'abord de réfléchir à la gestion des événements asynchrones : en effet, alors que certains événements peuvent être tracés au moment où il se produisent, d'autres événements se produisent à des temps qui ne sont connus que plus tard durant l'exécution. De plus, nous constatons que tous les événements intéressants ne peuvent être tracés en une seule exécution d'une application. Nous devrons donc nous intéresser à la question de la remise en ordre d'événements et à celle de la fusion de traces provenant de plusieurs exécutions.

1.3 Objectifs de recherche

Notre objectif principal de recherche consiste à fournir des outils permettant de générer des traces logicielles rendant compte de l'activité des processeurs graphiques au cours de l'exécution d'une application.

De cet objectif découlent plusieurs objectifs spécifiques. La première étape de nos travaux de recherche se concentre sur la compréhension des mécanismes déjà présents dans les outils d'analyse existants. Nous devons ensuite adapter ces mécanismes à ROCr, l'environnement d'exécution compatible avec HSA dont le code source a été ouvert dans le cadre du projet GPUOpen, en insérant des points de trace pertinents grâce à LTTng-UST. Cette étape implique de réfléchir à la solution la plus appropriée pour gérer le traçage de différents événements au cours de plusieurs exécutions d'une même application. Nous devons également nous assurer que le surcoût imposé par cette architecture reste faible afin de ne pas perturber le fonctionnement habituel de l'application tracée. En supposant que nous sommes capables de générer les traces qui nous semblent pertinentes, notre objectif de recherche suivant consiste à fournir des mécanismes de traitement pour nos traces permettant de réordonner les événements et de regrouper en une trace unique toutes les informations collectées. En se basant sur cela, nous devrons également proposer des outils visuels afin de faciliter la compréhension des résultats.

Un objectif complémentaire de notre projet de recherche est la meilleure compréhension des interactions entre l'activité du processeur graphique et celle du noyau du système d'exploitation. Notre approche se basera sur l'utilisation de LTTng-modules pour tracer le noyau Linux au cours de l'exécution d'un programme exploitant un ou plusieurs processeurs graphiques.

1.3.1 Questions de recherche

Nous cherchons donc à répondre aux questions de recherche suivantes :

Q1. Comment intégrer des mécanismes de traçage performants dans un environnement

d'exécution préexistant ?

Q2. Comment construire une architecture de traçage et traiter les traces obtenues de manière à obtenir une trace temporellement cohérente ?

Q3. Quels types d'outils peut-on développer pour interpréter facilement ces traces ?

1.4 Plan du mémoire

Après cette introduction, le chapitre 2 de ce mémoire consiste en une revue critique de la littérature. L'objectif est de renseigner le lecteur sur l'état de l'art en ce qui concerne les processeurs graphiques, leur exploitation à des fins de calcul, ainsi que les différents mécanismes déployés dans le but d'analyser l'exécution de programmes, y compris ceux utilisant des processeurs graphiques.

Le chapitre 3 est consacré à la méthodologie utilisée pour obtenir nos résultats. Cette partie détaille la configuration utilisée ainsi que la manière dont les résultats présentés ont été obtenus.

Le chapitre 4 consiste en un article scientifique présentant en détail l'architecture de notre solution ainsi que les résultats observés. Une revue de littérature des concepts pertinents dans le cadre de l'article est également présentée.

Dans le chapitre 5, nous proposons un approfondissement et une analyse critique des résultats présentés au chapitre 4.

Ce mémoire se termine par une conclusion qui, après avoir résumé les travaux de recherche entrepris, discute des limitations observées pour notre solution et évoque des pistes intéressantes pour poursuivre cette recherche.

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE

Dans ce chapitre, nous nous intéressons aux travaux précédemment effectués dans le domaine, qui peuvent éclairer le lecteur dans la lecture de ce mémoire.

2.1 Calcul parallèle, systèmes parallèles

Cette première section aborde l'idée générale de calcul parallèle et fait le point sur les systèmes existants à l'heure actuelle pour mener à bien ce type de calcul.

2.1.1 Principes théoriques

L'approche souvent vue comme la plus naturelle pour la conception d'un algorithme est sérielle : l'algorithme est vu comme une suite d'instructions s'exécutant l'une à la suite de l'autre sur un processeur unique et produisant certaines données de sortie à partir d'un ensemble de données d'entrée. Le calcul parallèle consiste en l'idée de mener un calcul en faisant travailler plusieurs processeurs ou éléments de calcul de manière simultanée (Almasi, 1985) : le traitement n'est alors plus sériel ; il a été parallélisé.

L'idée générale conduisant au calcul parallèle se fonde sur l'intuition selon laquelle l'exécution en parallèle d'un algorithme serait plus rapide que la version sérielle. Il est en effet clair que si l'on parallélise de manière idéale une portion d'un algorithme, cette section s'exécutera en un temps divisé par un facteur correspondant au nombre d'éléments de calcul mis en œuvre. La loi d'Amdahl (Amdahl, 1967) formalise cette intuition de la manière suivante (Hill and Marty, 2008) :

$$\frac{t_s}{t_p} = \frac{1}{1 - p + \frac{p}{n}} \tag{2.1}$$

où :

- t_p est le temps d'exécution de l'algorithme parallélisé ;
- t_s est le temps d'exécution de l'algorithme sériel ;
- p est la taille de la section parallèle de l'algorithme proportionnellement à la taille totale de l'algorithme ($0 \leq p \leq 1$) ;
- n est le nombre d'éléments de calcul mis en œuvre dans l'exécution de la section parallèle de l'algorithme ($n \geq 1$).

Le rapport t_s/t_p, toujours supérieur ou égal à 1, est l'accélération théorique tirée de la parallélisation d'une fraction de l'algorithme.

Si la loi d'Amdahl traduit les bénéfices du calcul parallèle en matière d'économie de temps de calcul, il faut en revanche également tenir compte d'un certain nombre d'autres facteurs. En effet, selon la méthode utilisée pour mener à bien la parallélisation d'une ou plusieurs sections de l'algorithme, des temps supplémentaires pourraient s'ajouter en raison de la nécessité de mettre en place les mécanismes de parallélisation et de rassembler les résultats obtenus de plusieurs éléments de calcul. Des contraintes financières peuvent également apparaître concernant l'acquisition de matériel approprié et la transformation du code source vers l'utilisation d'algorithmes parallèles (Hill and Marty, 2008).

2.1.2 Types de parallélisme

La parallélisation d'un algorithme peut s'envisager de deux manières différentes. On définit le parallélisme de tâches et le parallélisme de données (Hennessy and Patterson, 2011).

Le parallélisme de tâches consiste à organiser son algorithme de manière à le découper en différentes fonctions (« tâches ») qui s'exécutent en parallèle sur différents processeurs. Ces fonctions peuvent prendre en entrée des données identiques ou différentes. Un exemple typique illustrant le parallélisme de tâches est le cas d'un programme qui démarrerait différents fils d'exécution, chacun gérant une tâche différente : l'écriture d'un fichier, l'attente d'une entrée de l'utilisateur, la mise à jour d'une interface graphique, etc. La chaîne de traitement, communément connue sous le nom de *pipeline*, est une architecture classique permettant de gérer efficacement le parallélisme de tâches, dans laquelle les processeurs travaillent simultanément en prenant chacun en charge une étape de la chaîne totale. Cette architecture est par exemple souvent rencontrée dans le cadre d'applications liées au traitement d'images (Subhlok and Vondran, 2000), un domaine dans lequel le parallélisme de tâches peut venir en complément du parallélisme de données qui y apparaît naturellement.

Le parallélisme de données consiste à découper l'ensemble de données d'entrée en un certain nombre de sous-ensembles qui sont traités simultanément par une même fonction s'exécutant sur plusieurs processeurs. Ce type de parallélisme se retrouve au niveau matériel dans les processeurs de type *single instruction, multiple data* (SIMD) : un processeur suivant ce modèle peut simultanément exécuter une même instruction sur plusieurs entrées. En combinant plusieurs processeurs SIMD, il est possible d'atteindre un haut niveau de parallélisme résultant en un gain de performance important (Ahlander et al., 1996).

Ces deux conceptions du parallélisme répondent à des besoins différents et ne sont pas mutuellement exclusives. Le parallélisme de tâches est applicable dès que le problème général est séparable en différentes tâches pouvant s'exécuter simultanément. Le parallélisme de données apparaît naturellement lorsqu'un traitement identique doit être appliqué à des données

structurées de telle sorte qu'elles puissent aisément être découpées et traitées simultanément. Cela est typiquement le cas des données vectorielles ou matricielles, omniprésentes dans le domaine scientifique et dans les applications graphiques (traitement d'images, animation, etc.).

Un type particulier de calcul parallèle est le calcul réparti (aussi connu sous le nom de calcul distribué). On parle de système réparti lorsque les éléments de calcul impliqués constituent plusieurs ordinateurs distincts, chacun étant un nœud, et reliés par un réseau informatique (Kshemkalyani and Singhal, 2011). L'ensemble des ordinateurs mis en œuvre est une grappe de calcul. Un système réparti couple généralement des mécanismes locaux de calcul parallèle à un échange de messages entre nœuds afin de distribuer le calcul.

2.1.3 Problèmes et défis engendrés par le calcul parallèle

Indépendamment des technologies et mécanismes choisis pour paralléliser un algorithme, plusieurs problèmes sont soulevés par l'idée même d'exécuter plusieurs tâches simultanément.

Accès concurrents et verrouillage de la section critique

Un des problèmes majeurs pouvant émerger dans le cadre du calcul parallèle concerne les accès concurrents. De manière générale, toute écriture d'une ressource de manière concurrente par plusieurs processus est susceptible de causer un résultat inattendu. En effet, les opérations d'écriture n'étant généralement pas atomiques, les instructions correspondant à plusieurs écritures simultanées pourraient s'intercaler et ainsi produire un résultat non conforme à l'intuition (Netzer and Miller, 1992).

Différentes stratégies existent pour pallier le problème des accès concurrents. Il s'agit la plupart du temps de verrouiller la section critique du programme, c'est-à-dire la partie potentiellement exécutée simultanément par plusieurs processus, de manière à ce qu'au plus un unique processus puisse l'exécuter à la fois. Cette méthode permet de garantir la validité du résultat, mais va à l'encontre du principe même d'exécution parallèle, empêchant le parallélisme sur une partie du programme, entraînant nécessairement une perte de performance. On veillera donc à réduire au minimum la longueur des sections critiques (Suleman et al., 2009).

Une solution populaire pour le verrouillage est le mutex (pour *mutual exclusion*), un type de sémaphore assurant l'exclusivité d'accès à la section critique pour un seul processus au plus. Un exemple typique de mutex est donné par la norme *Portable Operating System Interface* (POSIX) : à l'entrée de la section critique, le fil d'exécution courant essaye de saisir le verrou ; si cela réussit, la section critique s'exécute jusqu'à la fin et le verrou est libéré ; dans

le cas contraire, le fil d'exécution est mis en attente jusqu'à l'obtention du verrou (Lumetta and Culler, 1998). Une autre solution simple réside dans l'utilisation d'un verrou tournant (couramment connu sous le nom de *spinlock*) : au lieu d'être bloqué pour l'obtention d'un verrou, le processus courant est placé en situation d'attente active, c'est-à-dire qu'il entre une boucle sans autre fonction que la vérification de la disponibilité du verrou. Cette solution, qui présente l'inconvénient important de consommer inutilement du temps de processeur, comporte l'avantage de ne pas nécessiter de changement de contexte entre processus, ce qui peut en faire une solution plus enviable dans des contextes de bas niveau tels que le noyau du système d'exploitation (Mellor-Crummey and Scott, 1991).

Gestion des interblocages

L'interblocage est un type de problème couramment rencontré dans un contexte de calcul parallèle. Il s'agit d'une situation dans laquelle deux processus s'exécutant de manière concurrente s'attendent mutuellement, résultant en un blocage du programme. L'interblocage peut survenir dans divers contextes et à différents niveaux de parallélisme : dans la gestion des verrous, des barrières mémoire, ou encore des messages entre nœuds d'un système réparti. Coffman et al. (1971) a défini quatre conditions qui, lorsqu'elles sont réunies simultanément, caractérisent l'apparition d'un interblocage.

Plusieurs stratégies peuvent être mises en place dans le but d'éviter l'apparition d'interblocages. Par exemple, dans le cas où plusieurs processus cherchent à acquérir les mêmes verrous, définir un ordre d'obtention, et implémenter l'accès aux sections critiques de sorte que les verrous soient obtenus à chaque fois dans l'ordre choisi, permet de prévenir les interblocages dus aux verrous. De plus, l'apparition d'interblocages peut être repérée en traçant un graphe présentant les demandes d'accès aux ressources par les différents processus (Chen et al., 1996).

L'interblocage actif est une situation similaire à l'interblocage présenté précédemment (dit passif), à la différence qu'au lieu de s'attendre mutuellement, les processus communiquent répétitivement entre eux sans faire progresser la logique du programme. Les processus ne sont pas bloqués mais le programme est coincé dans un état de manière similaire à l'interblocage passif (Tai, 1994).

Analyse et débogage de systèmes parallèles

Au cours du développement d'un projet, des bogues peuvent apparaître et il est souvent nécessaire d'avoir recours à des outils permettant une compréhension fine de la manière dont

un programme se déroule. Le débogage consiste à suivre pas à pas l'exécution d'un programme afin, notamment, de pouvoir analyser précisément l'état des variables en chaque point. Si le débogage est relativement aisé pour un programme utilisant un seul fil d'exécution, ce type de traitement peut en revanche se révéler plus complexe pour un programme parallélisé sur plusieurs fils d'exécution (McDowell and Helmbold, 1989). Chaque fil d'exécution peut en effet posséder ses propres ressources, ce qui signifie qu'une variable identifiée par un nom unique dans le code source peut à un instant donné avoir des valeurs différentes selon le fil d'exécution considéré. La conception et l'utilisation d'un débogueur dans le cadre d'un système parallèle sont donc sensiblement plus complexes que dans un contexte sériel. De manière générale, les outils permettant d'analyser le bon déroulement de l'exécution et la performance d'un programme doivent utiliser des stratégies spécifiques pour s'adapter à des situations d'exécution concurrente.

Il convient d'abord de mentionner que l'idée naïve de déboguer un programme en ajoutant des instructions affichant à l'écran la valeur d'une variable, à un point quelconque du code source, est peu appropriée dans le cadre d'un programme parallèle. En effet, afficher une chaîne de caractères équivaut à une écriture dans le flux de sortie standard. Dans un contexte parallèle, sans mécanisme spécifique de verrouillage, on se retrouve donc dans une situation d'écritures concurrentes comme mentionné plus tôt. Si de telles écritures sont faites dans une section parallèle du programme, il est probable que les sorties de chaque fil d'exécution soient mélangées, produisant un résultat peu compréhensible. Il est donc plus prudent de recourir à un débogueur spécifiquement conçu pour ce type de besoin.

Un des débogueurs les plus populaires pour le système d'exploitation GNU/Linux est *GNU Debugger* (GDB) (Stallman et al., 1988). GDB propose des commandes spécifiques pour les programmes s'exécutant sur plusieurs fils d'exécution, permettant notamment d'obtenir des informations sur les fils d'exécution existants et de passer d'un fil à l'autre. Il est également possible d'appliquer un ensemble de commandes à un ensemble de fils d'exécution, ce qui permet à l'utilisateur de travailler de manière naturelle et transparente avec les fils d'exécution, comme il pourrait le faire avec autant de programmes distincts.

Concernant le calcul réparti, des outils ont été spécialement conçus pour analyser des programmes s'exécutant sur une grappe de calcul. L'enjeu est ici de s'intéresser aux échanges de messages entre processus de manière à détecter des problèmes de délais ou encore des interblocages. Un outil d'analyse tel que Paraver (Chung et al., 2006) permet d'obtenir des données intéressantes sur plusieurs niveaux de parallélisme, typiquement un ou deux niveaux de parallélisme local et un niveau de parallélisme réparti basé sur un échange de messages. Des diagrammes proposent une visualisation graphique des échanges de messages entre nœuds.

2.1.4 Technologies de calcul parallèle sur processeur central

Dans cette sous-section, nous nous intéressons aux différentes technologies permettant de réaliser du calcul parallèle. La section suivante de la revue de littérature étant plus précisément consacrée au calcul sur processeur graphique, nous nous limitons ici principalement au cas du processeur central.

Fils d'exécution avec `pthread`

`pthread` (Nichols et al., 1996) est une bibliothèque logicielle implémentant les fils d'exécution définis par la norme POSIX. `pthread` est couramment utilisé pour gérer plusieurs fils d'exécution dans une application et possède des implémentations pour différents systèmes d'exploitation dont GNU/Linux et Microsoft Windows. Une des fonctionnalités centrales de la bibliothèque est d'exécuter une fonction donnée dans un nouveau fil d'exécution et d'attendre qu'un fil se termine. `pthread` fournit également une implémentation de divers mécanismes de synchronisation pertinents pour la gestion d'une application parallélisée sur plusieurs fils d'exécution, comme les mutex et les barrières mémoire.

`pthread` est généralement vu comme une solution simple pour une première étape dans la parallélisation d'un programme existant. Son fonctionnement se prête naturellement à un parallélisme de tâches, mais peut aisément être appliqué à un parallélisme de données. Cela impose néanmoins à l'utilisateur de fournir lui-même une solution de partitionnement de son ensemble de données d'entrée. `pthread` réalise donc une forme de parallélisme dans laquelle une certaine quantité de travail reste à la charge du développeur. Cela peut être indésirable, mais présente l'avantage de ne pas apporter trop d'abstraction dans le fonctionnement de l'algorithme.

Intel TBB

Intel *Threading Building Blocks* (TBB) (Willhalm and Popovici, 2008) est une bibliothèque logicielle implémentant des mécanismes de calcul parallèle sur plusieurs fils d'exécution du processeur central. TBB propose plusieurs modèles classiques d'algorithmes permettant de traiter des données en parallèle, tels que l'itération et la réduction sur un intervalle d'un vecteur de données. Le développeur associe un de ces modèles à ses propres fonctions de traitement, décrivant les opérations à effectuer au moment de l'itération et de la réduction, par exemple. TBB se charge ensuite lui-même de partitionner l'ensemble de données d'entrées et de répartir le travail sur les différents processeurs disponibles, de sorte qu'il est ainsi plus facile d'optimiser le découpage des données et de maximiser l'occupation de chaque

processeur. Plusieurs partitionneurs sont mis à disposition, présentant des logiques différentes ainsi que des fonctionnalités permettant d'influencer la granularité du partitionnement. Ces choix ont potentiellement une influence cruciale sur la performance du programme, mais TBB propose des choix par défaut qui tentent de trouver le meilleur compromis en fonction du problème courant.

TBB est donc une solution de parallélisation qui pourrait être classée dans la même catégorie que le calcul parallèle avec `pthread`, dans le sens où un effort important d'adaptation de l'algorithme est nécessaire. Dans ces deux cas, il est préférable de concevoir directement les algorithmes dans un contexte de traitement parallèle, car la parallélisation d'un algorithme sériel existant pourrait représenter des coûts supplémentaires. De plus, les choix faits par TBB pour le partitionnement des entrées peuvent parfois être inattendus pour un développeur peu expérimenté. En revanche, TBB apporte une intelligence supplémentaire par rapport à `pthread` dans la gestion du partitionnement et de la répartition du travail qui en fait une solution enviable en matière de performance lorsqu'elle est bien maîtrisée.

OpenMP

Open Multi-Processing (OpenMP) (Dagum and Menon, 1998) est une interface de programmation permettant de transformer un algorithme sériel dans le but d'une exécution parallèle, essentiellement pour une exécution à plusieurs fils d'exécution sur processeur central. Si ses mécanismes profonds de fonctionnement, basés sur un partitionnement automatique d'un ensemble de données d'entrée et une optimisation de la répartition du travail, sont comparables à ceux de TBB, OpenMP s'utilise en revanche de manière très différente, ce qui en fait une solution pouvant être adaptée dans des cas différents. En effet, OpenMP se base sur un ensemble de directives adressées au compilateur et insérées dans un code source sériel préalablement écrit afin de le paralléliser. Dans le cas de l'implémentation pour les langages C et C++, les instructions sont adressées au préprocesseur par le biais de la directive `#pragma`. Il est donc possible avec OpenMP de se baser sur des algorithmes existants et de les rendre parallèles moyennant peu d'efforts. Au-delà de cette fonctionnalité centrale, OpenMP propose un certain nombre d'instructions permettant de délimiter une section critique ou de définir des barrières mémoire afin de paramétrer plus finement la manière dont le programme doit être parallélisé.

OpenMP constitue donc un bon choix pour paralléliser rapidement des algorithmes existants sans avoir à repenser la logique sous-jacente. Cependant, en raison de l'idée centrale de l'interface de programmation, basée sur des directives de préprocesseur, le fonctionnement réel d'OpenMP peut paraître obscur, et dans certains cas le succès ou non de la paralléli-

sation peut apparaître comme relativement peu clair. Les imprévisibilités des algorithmes de partitionnement relevées pour TBB demeurent également valables pour OpenMP. Malgré ces défauts, OpenMP est souvent d'une utilisation relativement simple et permet une accélération appréciable, ce qui en fait une solution populaire.

OpenCL

Open Computing Language (OpenCL) (Stone et al., 2010) est une infrastructure logicielle dédiée à l'exécution d'applications dans des systèmes hétérogènes, c'est-à-dire des systèmes comportant possiblement différents types de processeurs, dont des processeurs centraux multi-cœurs, des processeurs graphiques ou encore des circuits de type FPGA. Plus spécifiquement, OpenCL est donc capable de fournir des capacités de calcul parallèle sur processeur central. Cependant, OpenCL est plus communément utilisé dans un contexte prioritairement dirigé vers l'exécution de programmes sur des processeurs graphiques. Nous abordons spécifiquement le cas de cette utilisation d'OpenCL dans la section dédiée aux processeurs graphiques.

2.2 Processeurs graphiques

Dans cette section, nous nous intéressons à l'utilisation des processeurs graphiques ainsi qu'à leur architecture et aux technologies disponibles pour les programmer à des fins de calcul parallèle.

2.2.1 Historique et principes généraux

Depuis les années 1970, avec l'avènement des jeux vidéo, la spécificité des calculs en lien avec les affichages graphiques s'est fait ressentir de plus en plus fortement. Ces calculs sont généralement fortement fondés sur des notions d'algèbre linéaire : les objets mathématiques principalement manipulés sont les vecteurs et les matrices, en deux ou trois dimensions.

Une caractéristique mathématique fondamentale de nombreux calculs vectoriels est la possibilité de diviser le calcul en autant de parties indépendantes que la dimension des vecteurs en question. Nous prenons ici l'exemple de l'addition et du produit scalaire de deux vecteurs. Considérons les vecteurs X et Y ainsi que leurs coordonnées $(x_1, \ldots, x_n)$ et $(y_1, \ldots, y_n)$ dans

un espace vectoriel de dimension $n \geq 1$:

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \qquad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{2.2}$$

La somme $X + Y$ des deux vecteurs ainsi que leur produit scalaire $X \cdot Y$ se définissent respectivement par :

$$X + Y = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix} \qquad X \cdot Y = \sum_{i=1}^{n} x_i \times y_i \tag{2.3}$$

Concernant le vecteur $X + Y$, chaque composante peut alors être calculée indépendamment pour constituer le vecteur somme obtenu. Dans le cas du produit scalaire $X \cdot Y$, chaque terme de la somme peut être calculé indépendamment comme un simple produit de deux coordonnées ; il suffit ensuite d'additionner chacun des produits pour obtenir le résultat attendu.

Dans un contexte informatique, chacun des vecteurs X, Y et $X + Y$ est typiquement représenté comme un tableau de n flottants ; le produit scalaire $X \cdot Y$ est quant à lui représenté par un flottant. Grâce aux remarques précédemment formulées, il est possible de calculer simultanément chaque composante du vecteur somme et de l'écrire dans la case correspondante du tableau. De même, concernant le produit scalaire, chaque terme de la somme peut être calculé simultanément et ajouté à un accumulateur qui contiendra finalement le résultat attendu ; c'est l'idée d'une réduction, un algorithme parallèle classique. De tels calculs parallèles, ici montrés sur de simples vecteurs, sont généralisables à des additions ou multiplications de matrices, opérations très courantes dans les calculs d'affichage graphique.

Si de tels calculs sont fréquents, avec des données pouvant être de grande dimension, un processeur central ne pourra pas offrir un grand niveau de parallélisation : en effet, les processeurs centraux multicœurs possèdent typiquement deux, quatre ou huit cœurs. Il devient donc utile de concevoir du matériel capable d'un niveau de parallélisation bien plus élevé. Des cartes électroniques spécialisées pour les calculs vectoriels en parallèle sont alors apparues, notamment dans les consoles de jeu vidéo, et ont gagné en puissance au fil du temps, jusqu'à gagner le grand public à la fin du vingtième siècle. Le processeur graphique fourni avec la carte est alors désigné sous le nom de GPU (McClanahan, 2010) ou *visual processing unit* (VPU).

Avec les progrès de performance importants constatés pour les processeurs graphiques, ceux-ci ont commencé à être largement utilisés dans des contextes non graphiques dès le début du vingt et unième siècle. De très nombreux domaines scientifiques et industriels comportent des calculs hautement parallélisables, typiquement en lien avec de l'algèbre linaire, pouvant donc être accélérés grâce à un processeur graphique. Cette prise de conscience a fait naître la tendance GPGPU, qui consiste en l'utilisation de processeurs graphiques dans des situations où un rendu graphique n'est pas nécessairement attendu (Owens et al., 2008).

2.2.2 Applications courantes

Les applications des processeurs graphiques sont nombreuses. L'usage de processeurs graphiques pour accélérer des calculs tend à devenir la norme.

Jeux vidéo

Pour le grand public, les processeurs graphiques sont couramment utilisés pour accélérer les rendus graphiques en trois dimensions, essentiellement dans un contexte vidéoludique. Une des opérations de base dans le cadre d'une application graphique est la génération de polygones à l'écran. Les processeurs graphiques sont couramment employés pour accélérer le dessin des polygones, des textures, ainsi que la prise en charge des translations et rotations qui sont par exemple associées au déplacement d'un personnage dans un jeu. Une application telle qu'un jeu vidéo peut donc faire à tout instant un usage très important d'une carte dédiée à l'accélération des calculs, ce qui a participé à la démocratisation des processeurs graphiques et à l'émergence de modèles de puissance et prix divers.

Traitement d'images et de vidéos

Les images et vidéos obéissant la plupart du temps à une structure matricielle, les processeurs graphiques sont reconnus comme particulièrement efficaces pour les traitements les concernant. Il est courant que des logiciels de traitement d'images et de vidéos proposent une accélération matérielle. C'est par exemple souvent le cas pour une compagnie telle qu'Adobe : les logiciels Photoshop et Lightroom (traitement d'images) ainsi qu'After Effects et Premiere Pro (traitement de vidéos) disposent de telles options. Ces logiciels sont couramment employés dans des cadres professionnels. Le logiciel *GNU Image Manipulation Program* (GIMP) peut également utiliser la puissance de calcul d'un processeur graphique.

Le transcodage vidéo est un autre domaine dans lequel l'utilisation de processeurs graphiques est essentielle. Le logiciel libre de transcodage FFmpeg (*Fast Forward Moving Picture Experts*

Group) s'appuie sur une accélération matérielle pour ses calculs. FFmpeg et des logiciels comparables sont également couramment utilisés dans des contextes professionnels, notamment dans le monde de la télévision.

Calcul scientifique

Les processeurs graphiques sont quasiment utilisés dans tous les domaines de la science pour accélérer des calculs. En effet, de nombreux problèmes numériques, après discrétisation, donnent lieu à des équations matricielles et vectorielles particulièrement appropriées pour une accélération sur processeur graphique. Un exemple d'application numérique classique est la méthode des éléments finis ou une de ses variations, qui aboutit à la résolution d'un problème d'algèbre linéaire. Les applications finales de cette idée incluent une résolution numérique des équations de Navier-Stokes en thermodynamique (Goddeke et al., 2009) ou encore la simulation d'ondes sismiques (Komatitsch et al., 2010). Dans un cadre général, des logiciels de calcul tels que MATLAB (*Matrix Laboratory*) proposent une accélération graphique.

À plus grande échelle, des réseaux de calcul réparti ont été mis en place afin de former des grappes de calcul géantes potentiellement formées de plusieurs milliers de processeurs graphiques en réseau dans le but de résoudre des problèmes de très grande taille. Le logiciel *Berkeley Open Infrastructure for Network Computing* (BOINC) de l'Université de Californie à Berkeley (Anderson, 2004) permet de mettre en relation un grand nombre de clients volontaires pour donner une partie de la puissance de calcul de leur ordinateur (essentiellement par le biais d'un processeur graphique) dans le cadre de projets de calcul réparti. Parmi ces projets, on trouve SETI@home, qui analyse des signaux radio dans le but de trouver une possible intelligence extraterrestre, ou encore Folding@home (Beberg et al., 2009), qui simule les repliements de protéines pour la recherche biomédicale. Ce type de projets fait ainsi directement le lien entre le calcul scientifique et l'intérêt public.

Apprentissage profond

L'apprentissage profond, un sous-domaine de l'apprentissage automatique spécialisé dans la modélisation matricielle de données diverses, a pris un essor récent avec la montée en puissance des processeurs graphiques. Ce domaine repose en effet sur de grandes quantités de calculs liés à l'algèbre linéaire. Parmi les applications classiques de l'apprentissage profond, on peut citer la reconnaissance du langage naturel et la vision par ordinateur. L'apprentissage profond ayant fait l'objet de financements considérables de la part d'acteurs privés importants tels que Google, Apple, Facebook, Amazon et Microsoft, mais aussi d'acteurs publics,

de nombreuses avancées ont pu être réalisées, qui reposent grandement sur la puissance de calcul des processeurs graphiques. Ceux-ci sont par exemple utilisés pour l'application d'algorithmes liés aux réseaux de neurones artificiels, plus particulièrement ceux spécialisés pour l'apprentissage profond comme les réseaux de neurones convolutionnels.

La bibliothèque TensorFlow de Google Brain (Abadi et al., 2016) ainsi que la bibliothèque Theano, développée à l'Université de Montréal (Bastien et al., 2012), sont toutes deux capables d'exploiter la puissance de calcul des processeurs graphiques afin de réaliser des calculs pour ces réseaux de neurones. Du côté du matériel, les gammes de processeurs graphiques Tesla de Nvidia et Radeon Instinct d'AMD sont spécifiquement conçues pour une orientation GPGPU, et plus précisément pour l'apprentissage profond. Cependant, bien au-delà de la simple utilisation des processeurs graphiques par des bibliothèques spécialisées, la tendance est à la conception de matériel spécialisé pour les calculs tensoriels mis en œuvre dans l'apprentissage profond. Google a ainsi conçu un « processeur tensoriel », en anglais *tensor processing unit* (TPU) à cette fin (Wu et al., 2016). Ce type de processeur pourrait à terme remplacer les processeurs graphiques, jugés trop génériques dans le cadre de l'apprentissage profond.

2.2.3 Architectures de processeurs graphiques

Il n'existe pas de modèle générique pour l'architecture d'un processeur graphique. Un processeur graphique est avant tout un processeur optimisé pour les calculs impliqués dans les affichages graphiques, mais la réalisation de ce concept peut différer grandement selon les constructeurs et les modèles ; ainsi, un processeur graphique orienté vers le jeu vidéo est construit différemment d'un processeur graphique orienté vers le GPGPU. De plus, les architectures évoluent rapidement dans le temps. Il existe cependant des points communs entre les différentes architectures proposées pour les processeurs graphiques.

Structure externe

Avant de s'intéresser à la structure interne des processeurs graphiques, mentionnons qu'ils se présentent sous deux formes : en tant que carte graphique dédiée ou en tant que processeur graphique intégré.

Dans le premier cas, le processeur graphique se situe dans une carte graphique indépendante reliée à la carte mère de l'ordinateur. Un processeur graphique dédié dispose de sa propre mémoire intégrée à la carte sur laquelle il se situe. AMD et Nvidia sont les principaux constructeurs de cartes graphiques dédiées. Dans le second cas, le processeur graphique est

localisé comme le processeur central sur la carte mère de l'ordinateur et partage sa mémoire avec lui. On désigne souvent cette configuration sous le terme anglophone *integrated graphics processor* (IGP). Un processeur graphique intégré peut soit être physiquement séparé du processeur central, soit être fusionné avec le processeur central, comme dans les modèles *Accelerated Processing Unit* (APU) d'AMD ou Intel HD Graphics.

Les processeurs graphiques dédiés sont généralement plus puissants, mais consomment plus d'énergie et sont plus chers que les processeurs graphiques intégrés. La configuration intégrée convient à des usages modérés d'applications matériellement accélérées, alors que la configuration dédiée est plus adaptée pour des usages intensifs, tels que les jeux vidéo en trois dimensions ou le calcul scientifique.

Structure interne

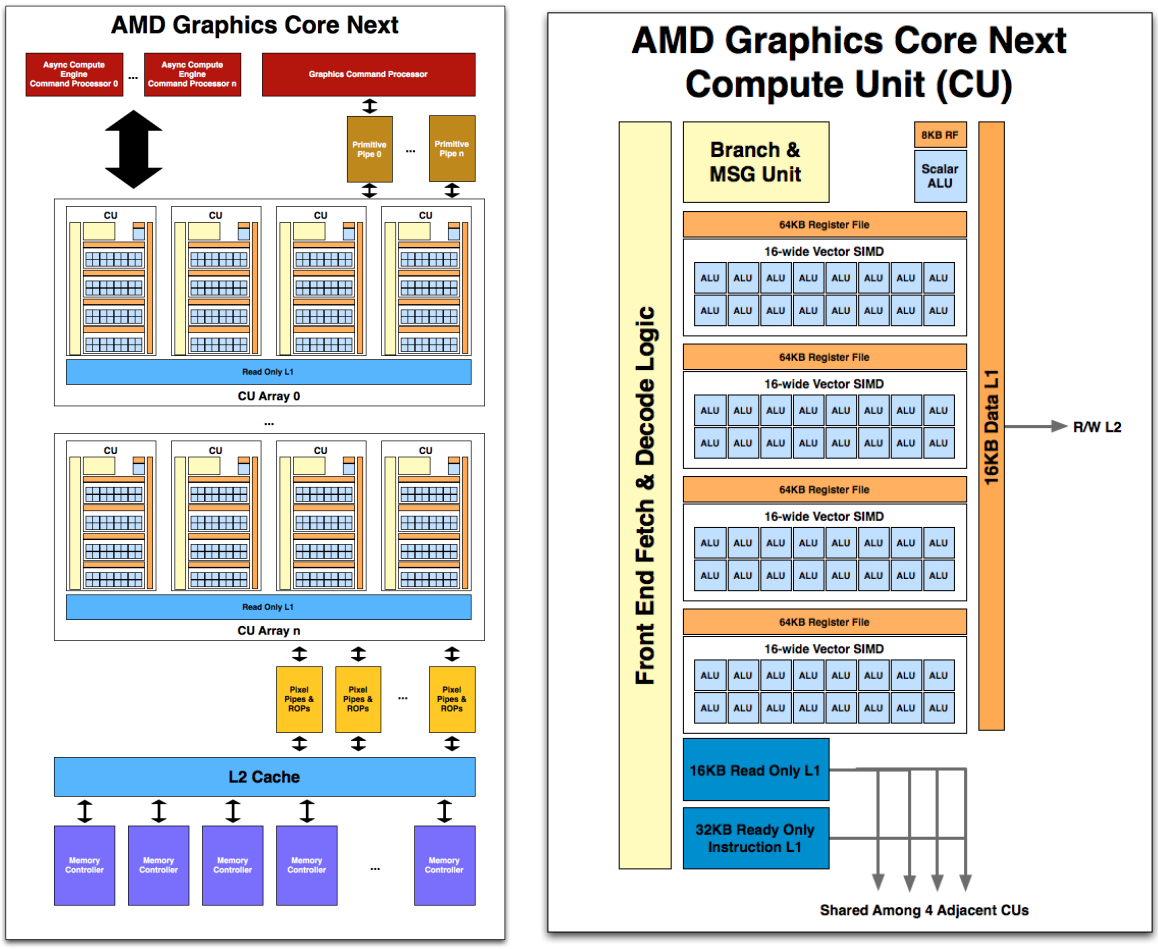

Figure 2.1 L'architecture GCN (Smith, 2011). © Purch, 2011. Reproduit avec permission.

Les processeurs graphiques s'inspirent des architectures vectorielles et du modèle SIMD pour permettre un haut niveau de parallélisation. Puisqu'un processeur SIMD est capable d'exécuter une seule instruction à la fois sur un ensemble de données, un processeur graphique doit regrouper plusieurs processeurs SIMD pour ne pas se limiter à une traiter une instruction à la fois. La manière dont les processeurs SIMD sont regroupés et l'organisation des mémoires cache comptent parmi les différences entre les architectures de processeurs graphiques.

À titre d'exemple, nous considérons ici l'architecture GCN d'AMD (Mantor, 2012), illustrée par la figure 2.1, qui peut être considérée comme une architecture typique pour un processeur graphique actuel. Le processeur graphique est, au plus haut niveau, constitué d'un ensemble d'unités de calcul qui partagent la mémoire cache de second niveau (L2). Chaque unité de calcul se compose de quatre processeurs SIMD ainsi qu'un processeur scalaire qui partagent une mémoire cache de premier niveau (L1). Dans le cadre de l'architecture GCN, un processeur SIMD peut traiter 16 données simultanément. L'architecture ne fixe pas le nombre d'unités de calculs. Si l'on note n le nombre d'unités de calcul, l'architecture GCN comporte $n \times 4 \times 16$ cœurs intégrés aux processeurs SIMD et ainsi capables de traiter des données en parallèle. Le modèle Radeon R9 Nano comporte par exemple 64 unités de calcul, ce qui correspond à 4096 cœurs au total. Selon la terminologie, un cœur de processeur graphique peut également être désigné par le nom de *shader* ou de processeur de flux.

2.2.4 Ordonnancement sur processeur graphique

Pour décrire le fonctionnement de l'opération d'ordonnancement et la correspondance faite entre l'architecture physique du processeur graphique et l'organisation logique du calcul, chaque modèle de programmation possède sa terminologie et des différences peuvent exister dans la manière de mener le calcul. Nous prenons ici l'exemple du modèle de programmation HSA qui sera abordé à la section suivante. Une illustration des concepts liés à l'ordonnancement dans le cadre de l'architecture HSA est montrée en figure 2.2.

Une fonction s'exécutant sur le processeur graphique est habituellement désignée sous le nom de noyau de calcul. Après avoir transmis les données d'entrée à la mémoire du processeur graphique, le noyau de calcul doit être réparti sur le processeur graphique pour être appliqué aux données. Cette opération s'accompagne de la définition d'une grille d'exécution pour les cœurs de calcul, en une, deux ou trois dimensions, selon la logique du calcul. On précise typiquement la taille de la grille dans chaque dimension, ou taille globale, et la taille des groupes dans chaque dimension, ou taille locale. Dans chaque dimension, il est classique que la taille globale soit un multiple de la taille locale afin d'éviter les groupes incomplets. Ces tailles doivent être choisies en fonction des caractéristiques du processeur graphique et de la

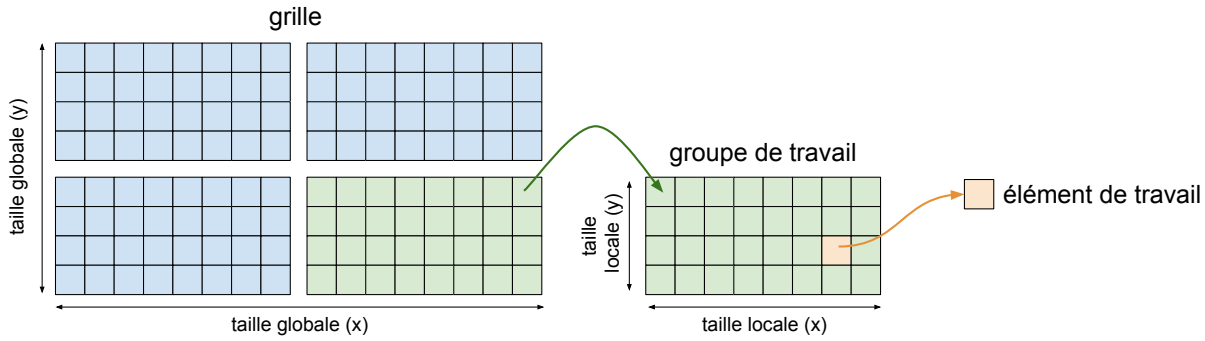

Figure 2.2 Illustration d'une grille d'exécution en deux dimensions.

nature du problème considéré. Ces concepts sont illustrés par la figure 2.2 pour le cas à deux dimensions.

Dans le cadre d'une grille définie pour l'exécution d'un noyau de calcul, on dit qu'un cœur représente un élément de travail (*work item*) ou fil d'exécution (*thread*). Les éléments de travail sont regroupés en groupes de travail (*work groups*) ou groupes de fils d'exécution (*thread groups*) ou blocs de fils d'exécution (*thread blocks*) ou tuiles (*tiles*). Au moment de l'exécution, chaque groupe de travail est assigné à une unité de calcul. À l'intérieur d'une unité de calcul, les éléments de travail exécutés simultanément constituent un front d'onde (*wavefront*) ou chaîne (*warp*). Les fronts d'onde s'exécutent successivement sur l'unité de calcul jusqu'à ce que tous les éléments de travail du groupe aient été traités.

2.2.5 Technologies de calcul parallèle sur processeur graphique

Dans cette sous-section, nous passons en revue les différentes solutions de calcul parallèle sur processeur graphique.

Technologies issues d'interfaces de calcul graphique

L'interface de programmation Direct3D de Microsoft et l'interface de programmation ouverte OpenGL sont les deux principales technologies permettant d'utiliser un processeur graphique pour des calculs parallèles à vocation graphique. Nous n'entrerons pas dans les détails de ces interfaces, car nous nous concentrons ici sur les solutions présentant une orientation vers le GPGPU, c'est-à-dire consacrées au calcul parallèle non graphique sur processeur graphique.

Cependant, afin de couvrir le maximum de besoins, des interfaces de programmation spécia-

lisées pour le GPGPU ont été dérivées de Direct3D et OpenGL. Il s'agit de DirectCompute pour Direct3D (Ni, 2009) et du concept de *shader* de calcul (*compute shader*) d'OpenGL (Bailey, 2016). Ces solutions s'utilisent plus volontiers en tant que complément de Direct3D et OpenGL que comme solutions indépendantes. Dans notre cas, nous nous concentrerons sur les interfaces de programmation spécialement dirigées vers l'exécution de calculs non graphiques sur le processeur graphique. Néanmoins, l'existence de ces deux interfaces de programmation a le mérite de montrer la complémentarité entre le domaine graphique et le domaine du GPGPU.

OpenMP

Nous avons précédemment abordé l'utilisation d'OpenMP pour l'exécution en parallèle de calculs sur un processeur central multicœurs. OpenMP présente un certain nombre d'avantages tels que sa relative simplicité d'utilisation puisque la logique algorithmique n'a en général pas à être grandement modifiée dans ce cadre.

Des efforts ont également été déployés dans le but d'utiliser OpenMP pour exécuter des calculs en parallèle sur un processeur graphique de marque Nvidia (Beyer and Larkin, 2016). Une configuration spécifique, avec le compilateur Clang, est requise, et un environnement *Compute Unified Device Architecture* (CUDA) doit être installé. Dans ces conditions, la directive `target` d'OpenMP permet de déplacer une région du code vers le processeur graphique pour profiter de son accélération. En plus de cela, des optimisations de parallélisme et d'ordonnancement doivent néanmoins être faites avec OpenMP afin d'observer une accélération effective.

L'exécution de calculs en parallèle sur processeur graphique avec OpenMP est donc encore relativement complexe et les possibilités sont assez limitées, mais cette piste pourrait s'avérer prometteuse à l'avenir.

C++ AMP

C++ Accelerated Massive Parallelism (C++ AMP) est une bibliothèque proposée par Microsoft permettant d'écrire facilement du code pouvant s'exécuter sur un processeur graphique (Wynters, 2016). Le fonctionnement de C++ AMP rappelle celui d'Intel TBB dans le sens où la bibliothèque propose un ensemble de modèles d'algorithmes permettant notamment d'itérer sur les valeurs d'un vecteur en effectuant une opération définie par une fonction anonyme. En cela, C++ AMP se concentre sur l'aspect algorithmique de la parallélisation plutôt que les détails techniques de la communication avec le processeur graphique.

C++ AMP ne fait pas partie des solutions les plus populaires pour la programmation sur processeur graphique. Cependant, dans le cadre du projet GPUOpen, AMD a mis à disposition *Heterogeneous Compute Compiler* (HCC) (Sander et al., 2015), un compilateur à source ouverte permettant de convertir du code C++ AMP en code directement exécutable par les processeurs graphiques basés sur l'architecture GCN d'AMD, ce qui apporte un intérêt nouveau à C++ AMP.

CUDA

CUDA (Kirk et al., 2007) est la technologie de référence pour le calcul orienté GPGPU sur processeur graphique Nvidia. En tant que plateforme de calcul officielle pour Nvidia, CUDA est exclusivement supporté sur le matériel de ce constructeur, mais y est la solution la plus optimisée et la plus fiable. CUDA est une technologie à code source fermé, mais tend à être plus utilisé que son concurrent ouvert OpenCL sur le matériel Nvidia en raison d'un meilleur support et de sa relative facilité d'utilisation.

Afin de pallier l'aspect exclusif de CUDA à Nvidia, le constructeur AMD a proposé dans le cadre du projet GPUOpen l'outil ouvert *Heterogeneous-Compute Interface for Portability* (HIP) permettant de transformer du code source CUDA non portable en code source C++ portable pouvant être exécuté à la fois sur du matériel Nvidia et sur du matériel AMD par le biais du compilateur HCC décrit précédemment. Ce type d'outil permet d'ouvrir CUDA à du matériel de différents constructeurs.

OpenCL

OpenCL (Stone et al., 2010) est une interface de programmation à code source ouvert destinée à permettre l'exécution de calculs parallèles dans un contexte hétérogène. Outre CUDA sur les plateformes Nvidia, OpenCL est le choix le plus populaire pour le développement d'applications utilisant un processeur graphique. OpenCL fonctionne de manière comparable à CUDA. Il permet de choisir un appareil (processeur central, processeur graphique, etc.) sur lequel vont être exécutés les calculs, de créer une file qui va servir à la communication entre l'application et l'appareil et d'y enfiler des noyaux de calcul qui vont être répartis sur l'appareil cible, typiquement un processeur graphique. Après avoir appliqué la fonction voulue sur les données d'entrée, les résultats peuvent être lus et interprétés.

Comme d'autres interfaces permettant d'accélérer des calculs grâce à un processeur graphique, OpenCL permet d'atteindre une accélération importante par rapport à la version sérielle s'il est utilisé de manière appropriée. En revanche, il n'est pas aisé de transformer une

application pour la rendre capable d'utiliser un accélérateur matériel ou plus généralement de la rendre capable de s'exécuter dans un contexte hétérogène. Le programme en question devra être profondément transformé. De plus, par rapport à CUDA, OpenCL possède le léger désavantage de ne reconnaître les noyaux de calcul que lorsqu'ils sont livrés sous la forme d'une chaîne de caractère, là où CUDA peut interpréter des fonctions écrites de manière native. Ce problème peut néanmoins être facilement contourné en dédiant un fichier spécialisé à l'écriture des noyaux de calcul OpenCL.

OpenCL constitue en réalité un langage de programmation issu du langage C et utilisé pour écrire les noyaux de calcul destinés à s'exécuter en contexte hétérogène, souvent sur des processeurs graphiques. Du point de vue de la syntaxe, le langage OpenCL est en apparence identique au C mais propose en plus la gestion native de types vectoriels, permettant ainsi des opérations directes sur les types `int`n, `float`n et `double`n, correspondant à des vecteurs de n nombres.

HSA

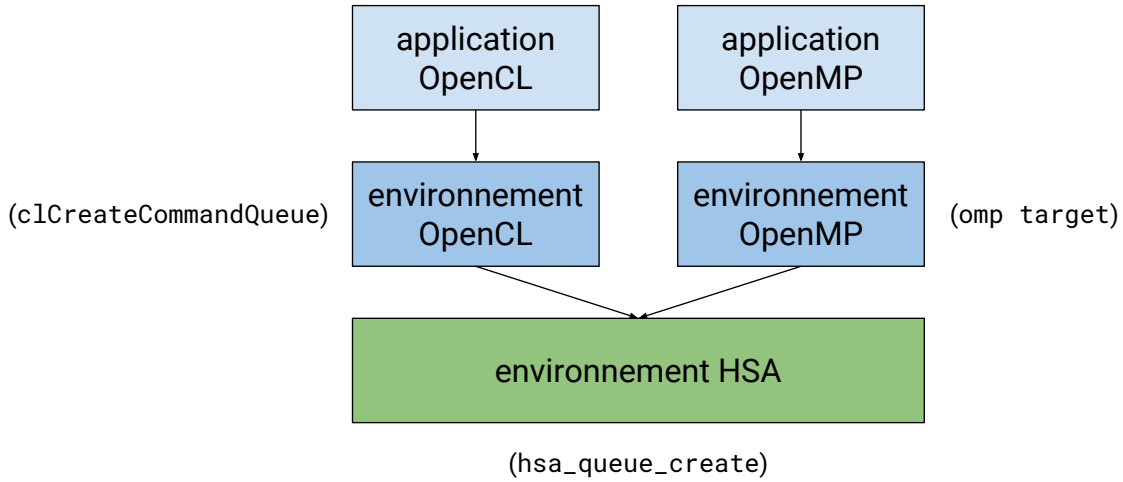

Figure 2.3 L'architecture logicielle de HSA vue comme base pour des environnements OpenCL et OpenMP.

HSA est un ensemble de spécifications développé par un consortium comprenant notamment les constructeurs AMD et ARM. Il est destiné à faciliter les communications entre appareils dans le contexte d'un système hétérogène. HSA permet d'établir des communications entre différents types de processeurs sans avoir à transférer les données d'entrée et de sortie d'une mémoire à une autre. Ce standard vise donc à faciliter et accélérer les communications entre processeurs centraux, processeurs graphiques et autres types d'appareils de calcul (Mukherjee

et al., 2016). L'architecture proposée par le standard se base sur l'idée que les technologies actuelles permettant la parallélisation de programmes (telles que OpenCL, OpenMP, etc.) peuvent se servir de l'interface de programmation de HSA pour gérer la répartition du code source parallélisable (noyau de calcul en OpenCL, section de code en OpenMP, etc.) vers un appareil susceptible de l'exécuter, tel qu'un processeur graphique ou un processeur central multicœurs. En construisant ces différents environnements d'exécution avec HSA comme base commune, la performance devrait être améliorée. Ce principe est illustré par la figure 2.3. Une implémentation d'OpenCL basée sur HSA a été mise à disposition.

Le standard HSA introduit également le concept de files en mode utilisateur, qui permet à l'application de communiquer directement avec un appareil tel qu'un processeur graphique, dans un contexte utilisateur, sans passer par le système d'exploitation et les pilotes mis en place pour assurer la prise en charge de ce processeur. Les tâches à transmettre sont contenues dans des paquets circulant dans les files de communication, dont la structure est définie par le langage *Architected Queuing Language* (AQL).

HSA dispose donc de fonctionnalités particulièrement intéressantes qui justifient d'y consacrer des outils d'analyse particuliers.

2.3 Traçage logiciel

Nous présentons ici les différentes solutions existantes dans le domaine du traçage.

2.3.1 Traçage du processeur central

Le traçage du processeur central est une technique d'analyse appréciée des administrateurs système et des développeurs cherchant à détecter et résoudre des problèmes de performance bas niveau et parfois difficiles à diagnostiquer. Cela consiste à enregistrer, dans un fichier nommé « trace », la survenue de plusieurs événements précis, dans le but de permettre la détection de problèmes par l'analyse de la trace. Le traçage peut intervenir dans le noyau du système d'exploitation ou dans l'espace utilisateur.

Nous recensons ici les outils de traçage disponibles pour les systèmes GNU/Linux, qui se prêtent bien à des activités de traçage en raison de leur code source ouvert et facilement modifiable et redistribuable.

`strace`

`strace` est un traceur simple pour les systèmes GNU/Linux. Il peut être considéré comme un des premiers outils à essayer pour avoir une meilleure compréhension des interactions entre un programme et le système dans sa globalité. `strace` est capable, en prenant en entrée un exécutable à tracer, de lister chronologiquement les appels système impliqués avec leurs arguments et leur valeur de retour. Cet outil donne donc très simplement la possibilité d'analyser à bas niveau de possibles causes de ralentissement, ou, plus généralement, d'anomalies liées aux appels système mis en œuvre. `strace` peut également être employé pour simuler la réexécution d'un programme dans le but de prédire son comportement sous une configuration différente (Burton and Kelly, 1998). L'utilisation d'un tel outil ne nécessite pas de posséder le code source l'application tracée puisque `strace` est utilisable sans aucune recompilation ou reconfiguration.

Malgré cela, `strace` reste assez limité puisqu'il ne permet pas d'aller plus loin que lister les appels système et signaux observés sur le système. Cet outil ne nous permet pas de mettre en place notre propre instrumentation dans le but d'obtenir les informations qui nous intéressent ; les fonctionnalités attendues n'y sont donc pas. La performance est un autre problème central de `strace` : l'outil se base sur l'appel système `ptrace`, utilisé pour permettre à un processus d'en contrôler et analyser un autre, qui est connu pour causer d'importants ralentissements (Gregg, 2014). De plus, `strace` n'est pas adapté aux spécificités des programmes à plusieurs fils d'exécution : quel que soit le nombre de processus participant à l'exécution du programme, la sortie du traceur a lieu dans un unique fichier, nécessitant ainsi de mettre en place un mécanisme de verrouillage, ce qui a un impact sur la performance.

Ftrace

Ftrace (Rostedt, 2009) est un traceur de référence pour GNU/Linux. Il est intégré par défaut au noyau Linux, ce qui facilite son utilisation sous des systèmes très variés, que ce soit au niveau de leur architecture ou de leur configuration. Ce traceur se compose en pratique de plusieurs outils de traçage utilisables de manière distincte, qui permettent d'obtenir des informations de nature variée. Ftrace se base principalement sur des techniques d'instrumentation statique grâce aux nombreux points de trace insérés dans le noyau Linux, bien que de l'instrumentation dynamique par le biais de *kprobes* soit désormais possible. Contrairement à d'autres traceurs ou, plus généralement, d'autres outils d'analyse de performance, Ftrace possède la particularité d'être configuré et contrôlé par le biais d'un système de fichiers spécial nommé *Trace File System* (TraceFS) plutôt que par un ensemble de commandes.

Ftrace, bien qu'étant un outil de traçage reconnu pour les systèmes GNU/Linux, est par sa conception adapté au traçage dans le noyau du système d'exploitation plutôt que dans l'espace utilisateur. Or, dans le cadre de notre projet, nous souhaitons nous intéresser au traçage d'un environnement d'exécution pour processeur graphique, ce qui nécessite des capacités de traçage dans l'espace utilisateur. Bien que Ftrace présente des caractéristiques intéressantes pour un traceur, il ne sera donc pas l'outil le plus approprié pour nos travaux.

SystemTap

SystemTap (Eigler and Hat, 2006) est un traceur puissant pour GNU/Linux. Il permet d'attacher des scripts à des points de trace insérés statiquement ou dynamiquement, dans le noyau du système d'exploitation ou l'espace utilisateur. Les scripts en question sont écrits à l'aide d'un langage spécifique à SystemTap. Ce traceur, dans sa conception générale et son utilisation de scripts, est similaire à DTrace, un traceur pour les systèmes Unix. La possibilité d'attacher des scripts définis entièrement par l'utilisateur fait de SystemTap un outil très polyvalent avec un grand potentiel de personnalisation. Il est par exemple facile de construire rapidement des outils de profilage basiques comme un compteur d'appels de fonctions.

L'ajout de points de trace statiques dans des applications de l'espace utilisateur est introduit par SystemTap sous le nom de *User Statically Defined Tracing* (USDT). La terminologie et le fonctionnement général de ce type d'instrumentation sont directement repris de DTrace (Gregg and Mauro, 2011). Cette fonctionnalité est intéressante dans le cadre de nos travaux, car elle nous permettrait d'apporter notre propre instrumentation à un environnement d'exécution pour processeur graphique. Cependant, des problèmes de performance significatifs ont été relevés dans SystemTap lorsqu'un important volume de données est enregistré, ce qui pourrait être un handicap. De plus, notre objectif étant de construire une trace de l'activité du processeur graphique, nous n'avons pas nécessairement besoin d'attacher des opérations complexes à chaque rencontre d'un point de trace ; l'éventail de possibilités offert par les scripts de SystemTap pourrait donc se révéler inutile.

eBPF

Extended Berkeley Packet Filter (eBPF) est un outil intégré au noyau Linux pouvant être exploité à des fins de traçage. C'est une version étendue de *Berkeley Packet Filter* (BPF), une machine virtuelle intégrée au noyau Linux permettant de définir des programmes réalisant du filtrage de paquets réseau. eBPF apporte à BPF la possibilité de travailler au-delà du seul domaine de la réseautique, en attachant des programmes BPF à des points de trace statiques ou dynamiques, dans le noyau du système d'exploitation ou l'espace utilisateur. L'instrumen-

tation statique d'applications de l'espace utilisateur se base sur USDT de DTrace, comme SystemTap. eBPF améliore également la performance initiale grâce à des fonctionnalités de compilation à la volée. Des outils ont été développés pour convertir des traces obtenues avec eBPF au format CTF, un format binaire très compact, qui facilite la lecture et l'écriture performantes de traces.

eBPF est donc un outil prometteur dans le domaine du traçage, tant du côté du noyau du système d'exploitation que de l'espace utilisateur. De la même manière que SystemTap, il peut en revanche se révéler trop étendu pour l'usage que nous attendons. De plus, eBPF a fait l'objet de nombreuses attentions dernièrement puisque son potentiel est encore en phase d'exploration. Ce traceur pourrait donc être un outil exploitable à l'avenir, mais son utilisation actuelle présente quelques risques d'instabilité que nous préférons éviter.

LTTng

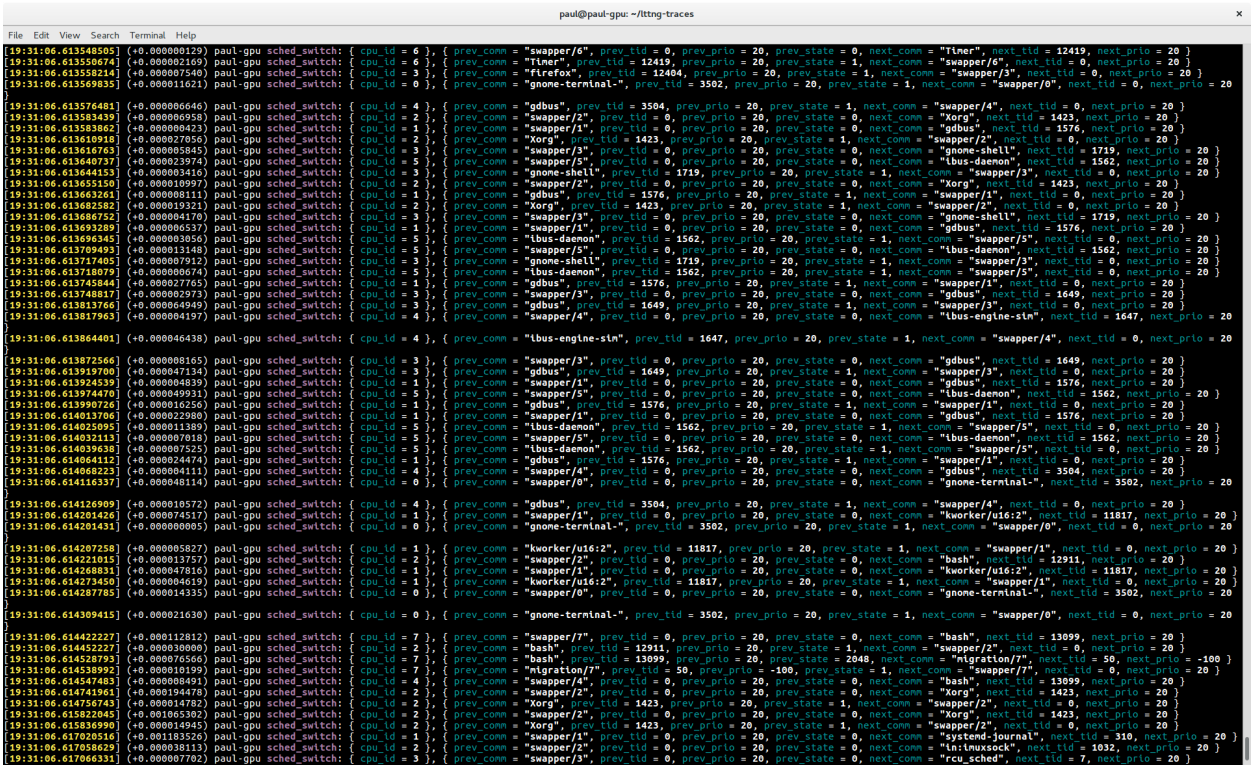

Figure 2.4 Une trace CTF générée avec LTTng et visualisée avec Babeltrace.

LTTng (Desnoyers and Dagenais, 2006) est un environnement de traçage à code source ouvert pour GNU/Linux qui est devenu un des outils de référence pour la détection de problèmes

en profondeur sur cette plateforme. Il succède à *Linux Trace Toolkit* (LTT). LTTng permet de tracer à la fois le noyau du système d'exploitation Linux et les programmes de l'espace utilisateur et a été conçu dans une optique de haute performance : le traceur doit avoir le plus faible impact possible sur les activités courantes du système d'exploitation.

Du côté du noyau du système d'exploitation, LTTng propose, dans le cadre du projet LTTng-modules, un certain nombre de modules qui permettent d'activer les points de trace insérés dans le noyau. Ainsi, l'utilisateur peut indiquer les types d'événements qu'il souhaite observer et obtenir la trace correspondante. Typiquement, LTTng permet d'obtenir des événements de trace correspondant à chaque appel système effectué, permettant ainsi de détecter un appel manquant ou encore des appels dupliqués. La figure 2.4 donne un aperçu d'une trace noyau générée par LTTng et montrant les différents changements de contexte enregistrés dans le système d'exploitation sur un court intervalle de temps. Le traceur permet également d'écrire ses propres modules afin d'obtenir des événements correspondant aux points de trace qui ne sont pas pris en compte par défaut dans LTTng. Au-delà des points de trace statiques, LTTng est capable de tracer les événements correspondant aux points de trace dynamiques connus sous le nom de *kprobes* (*kernel probes*). En ce qui concerne le traçage de l'espace utilisateur, LTTng-UST permet d'insérer ses propres points de trace statiques et de les activer afin d'obtenir les événements correspondant à ces points. Le traceur propose des outils permettant d'instrumenter des programmes écrits en langage C, C++, Java ou Python. Ce traçage de l'espace utilisateur peut très facilement être couplé au traçage dans le noyau du système d'exploitation pour une compréhension plus fine de ce qui entre en jeu. Dans tous les cas, LTTng génère des traces au format binaire CTF, qui, comme nous l'avons vu, est bien adapté à l'enregistrement et au traitement efficaces de traces.

LTTng parvient à atteindre une performance élevée par plusieurs partis-pris techniques intéressants. Premièrement, LTTng enregistre l'activité de chacun des cœurs du processeur central séparément, en constituant autant de « canaux ». Ceci permet d'éliminer le besoin de toute synchronisation entre processeurs, chacun d'entre eux écrivant dans un fichier, et donc d'économiser du temps. De plus, les événements sont enregistrés dans des tampons circulaires avant d'être écrits. Ces tampons sont utilisés de telle sorte que si trop d'événements sont produits par un processeur par rapport à la capacité d'écriture de la trace sur le disque, LTTng fera le choix de volontairement jeter le trop-plein d'événements plutôt que de ralentir le système pour écrire tous les événements à tout prix (Desnoyers and Dagenais, 2012).

LTTng est donc un traceur qui combine une gamme intéressante de fonctionnalités avec une haute performance. C'est donc un bon choix pour nos travaux.

2.3.2 Traçage et analyse de l'activité du processeur graphique

En dehors des outils consacrés au traçage du processeur central, des outils spécialisés ont été conçus afin d'analyser l'activité des processeurs graphiques. Les constructeurs de cartes graphiques sont des acteurs majeurs du développement d'outils de traçage pour processeur graphique, produisant le plus souvent des outils à code source fermé. Des solutions ouvertes, indépendantes d'un fabricant particulier, ont néanmoins vu le jour.

Nous présentons ici quelques-unes des solutions disponibles pour le traçage du processeur graphique et, plus généralement, de l'analyse de son activité, qu'elle découle d'une infrastructure de traçage explicite ou non. Nous pouvons donc inclure dans cette catégorie tout outil donnant accès à des informations précisant l'état des noyaux de calcul, à un compte-rendu des appels de fonctions impliqués, à des métriques permettant de mieux comprendre la performance du processeur graphique, ou à d'autres données pertinentes dans ce cadre.

Outils ouverts : TAU et VampirTrace

Tuning and Analysis Utilities (TAU) et VampirTrace sont deux outils spécialisés dans l'analyse du comportement de programmes à haut niveau de parallélisme. Ces utilitaires ont la particularité d'être à source ouverte.

TAU (Shende and Malony, 2006) est une suite d'outils permettant le traçage et le profilage d'applications parallèles. Cette suite d'outils est capable d'insérer par différentes techniques une instrumentation dans le programme cible afin de récupérer des informations intéressantes sur les appels de fonctions réalisés, les fils d'exécution mis en œuvre, etc. Un ensemble d'outils graphiques est fourni afin de permettre une analyse visuelle des données obtenues. Dans le cadre d'un programme accéléré par processeur graphique, basé sur OpenCL ou sur la plateforme CUDA du constructeur Nvidia, TAU déploie des mécanismes permettant d'instrumenter les fonctions de l'environnement d'exécution utilisé. TAU utilise son propre format de trace pour représenter les informations extraites de l'analyse, mais des outils de conversion sont proposés, notamment vers le format *Open Trace Format* (OTF) dans le but de visualiser les traces générées avec le logiciel propriétaire Vampir, ou encore vers le format utilisé par le logiciel Paraver.

VampirTrace (Müller et al., 2007) est une bibliothèque logicielle à code source ouvert à l'origine orientée vers le traçage de programmes hautement parallèles, typiquement accélérés sur plusieurs fils d'exécution grâce à l'interface OpenMP et fonctionnant de manière distribuée grâce à la norme *Message Passing Interface* (MPI). VampirTrace est également capable de générer des événements de trace en lien avec l'activité du processeur graphique. Le fonc-

tionnement général de VampirTrace est comparable à celui de TAU, avec différentes options d'instrumentation mises à disposition de l'utilisateur. Les traces sont par défaut générées au format OTF, lisible avec le visualiseur Vampir, d'où son nom.

Outils propriétaires des constructeurs Nvidia et Intel

Les constructeurs Nvidia et Intel, qui comptent parmi les principaux fabricants de cartes graphiques, mettent à disposition leurs propres plateformes logicielles permettant de comprendre l'activité du processeur graphique au cours de l'exécution d'un programme accéléré par matériel. Ces outils présentent l'avantage d'être directement supportés par les constructeurs, ce qui les rend plus susceptibles de rester compatibles à long terme avec le matériel utilisé, et de s'adapter aux évolutions d'architecture des processeurs graphiques. En revanche, les utilitaires directement proposés par les constructeurs de cartes graphiques sont généralement à source fermée, et sont exclusifs au matériel du constructeur en question.

L'utilitaire proposé par Nvidia est Nsight. Nsight possède des fonctionnalités de débogage pour la plateforme CUDA, des outils de traçage et profilage ainsi que différents moyens de visualisation des informations obtenues. Au-delà des fonctionnalités orientées vers les plateformes de calcul telles que CUDA, des outils sont proposés pour analyser les programmes utilisant l'accélération matérielle à des fins graphiques.

Dans le contexte du fabricant Intel, deux utilitaires majeurs sont proposés : VTune et *Graphics Performance Analyzers* (GPA). VTune (Reinders, 2005) est la plateforme de référence pour l'analyse de performance sur une grande diversité de plateformes Intel incluant les processeurs centraux et les processeurs graphiques. VTune peut en cela être vu comme l'équivalent pour Intel de CodeXL pour AMD et Nsight pour Nvidia. Dans le cadre de programmes accélérés par processeur graphique, VTune est principalement dirigé vers l'analyse de programmes construits avec OpenCL. Grâce à des mécanismes de traçage, il permet en particulier d'obtenir des métriques d'activité du processeur graphique et des diagrammes chronologiques. Intel GPA est une suite d'outils proposant des fonctionnalités similaires, spécialement pour les programmes construits grâce à l'interface OpenGL, consacrée aux applications graphiques.

CodeXL

CodeXL est la plateforme logicielle de référence du constructeur AMD pour l'analyse de programmes basés sur l'utilisation d'un processeur graphique. Un exemple de l'interface de CodeXL est illustré en figure 2.5. CodeXL permet d'analyser l'exécution de programmes exploitant le processeur graphique à des fins de calcul en se basant soit sur OpenCL, soit sur

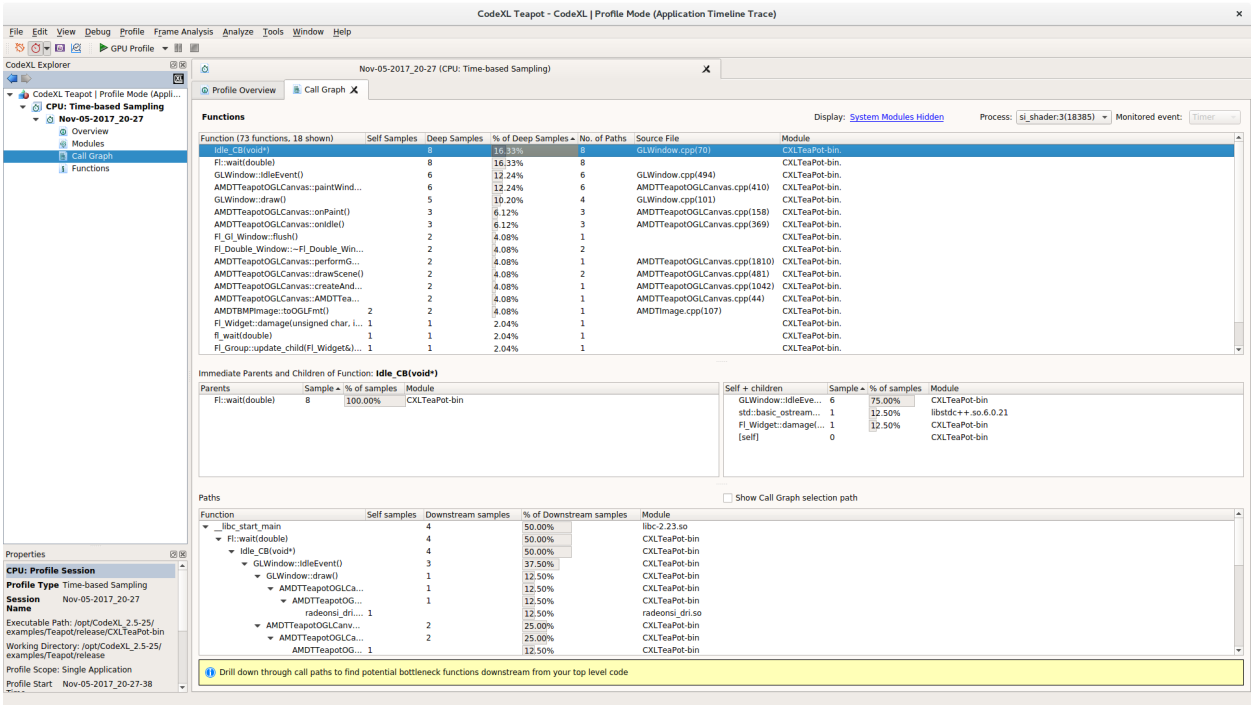

Figure 2.5 L'interface de CodeXL.

l'interface de programmation de HSA. Il est également possible d'obtenir des informations sur des programmes basés sur OpenGL, à orientation plus graphique. CodeXL se base sur des traces enregistrées au format texte permettant d'obtenir des informations sur les appels aux fonctions de l'interface de programmation et l'exécution des noyaux de calcul impliqués dans le déroulement du programme. Le format texte utilisé, nommé ATP, est spécifique aux outils d'AMD et est structuré de manière à être lisible par une personne humaine. À ce titre, le format vient sans documentation particulière et n'est pas conçu dans une optique d'optimisation des temps de lecture et écriture associés ou de l'espace utilisé.

CodeXL possède le double avantage d'être à code source ouvert comme Tau et VampirTrace, et d'être développé et soutenu directement par AMD, un des acteurs majeurs du marché, ce qui lui assure d'être bien adapté aux spécificités du matériel proposé par ce constructeur. L'aspect ouvert de CodeXL en fait une bonne source d'analyse des différents mécanismes qu'il est possible de mettre en place pour l'obtention d'informations utiles sur l'état du processeur graphique. CodeXL peut également être noté pour sa structure modulaire : plusieurs outils utilisés dans le cadre de CodeXL, comme l'utilitaire de profilage *Radeon Compute Profiler* (RCP) ou l'interface de programmation *GPU Performance API* (GPA), destinée à la collecte des compteurs de performance, ont été libérés et sont utilisables en tant que tels, nous offrant

ainsi de nombreuses possibilités.

CLUST

CLUST (de *CL* et *UST*) (Couturier and Dagenais, 2015) est un outil permettant d'obtenir des traces unifiées de l'activité du processeur central et du processeur graphique dans le cadre d'une application basée sur OpenCL. CLUST intercepte les appels aux fonctions d'OpenCL et y insère des points de trace pour le traceur LTTng-UST. CLUST implémente des mécanismes pour tenir compte à la fois des fonctions réalisant un traitement synchrone et celles travaillant de manière asynchrone.

Cet outil nous offre l'avantage d'obtenir des informations sur l'activité du processeur graphique via l'analyse des appels aux fonctions OpenCL, tout en poursuivant le traçage du système entier. CLUST s'intègre donc dans une perspective globale d'analyse, ne prenant pas simplement en compte l'activité due aux processeurs graphiques mis en œuvre, mais l'ensemble des causes possibles de problèmes de performance. À ce titre, la démarche de CLUST est pertinente dans l'analyse de systèmes hétérogènes où il est intéressant de raisonner au-delà d'un certain type de processeur. De plus, alors que les outils mentionnés précédemment sont souvent fortement couplés à des analyses graphiques prédéfinies, nous donnant peu l'occasion de travailler directement avec la trace elle-même, CLUST présente l'intérêt de générer une trace au format CTF, qui est bien approprié pour la représentation de ce type de traces. L'utilisateur peut ensuite lui-même décider de la conduite à adopter avec la trace obtenue : comme nous le verrons plus loin, elle peut être convertie dans divers formats, dont un format texte facilement lisible par une personne, ou bien analysée de manière interactive grâce à des outils tiers.

Si CLUST se limite à OpenCL, ses mécanismes internes peuvent en revanche être répliqués dans d'autres cadres. CLUST peut donc servir d'inspiration pour nos travaux.

2.4 Analyse et traitement de traces logicielles

Dans cette section, nous nous intéressons aux outils et techniques permettant de travailler avec des traces logicielles préalablement obtenues afin d'en tirer des informations pertinentes.

2.4.1 Conversion et modification de traces avec Babeltrace

Les traces obtenues par LTTng sont par défaut au format CTF, un format binaire optimisé pour l'enregistrement, la lecture et l'écriture d'événements de traces (Marangozova-Martin

and Pagano, 2013). Ce format n'est pas lisible par une personne souhaitant observer la trace pour avoir un simple aperçu des événements obtenus ou pour du travail plus poussé. Nous pourrions donc avoir besoin de convertir la trace binaire vers un format textuel facilement interprétable à l'œil. De même, plusieurs autres formats de trace existent ; passer d'un format à l'autre pourrait être utile pour assurer la compatibilité entre LTTng et d'autres utilitaires.

L'outil Babeltrace fournit l'implémentation officielle du format CTF et est l'outil de référence pour la lecture et l'écriture de traces dans ce format. Une utilisation typique de Babeltrace concerne la conversion du format CTF vers un format texte compréhensible pour une personne humaine : chaque événement est présenté sur une ligne avec l'estampille de temps correspondante, le contexte associé à l'événement, les différents arguments, l'identifiant du processeur, etc. Au-delà de ce cas d'utilisation classique, Babeltrace est capable de convertir des traces au format CTF vers d'autres formats de traces, et peut également convertir des traces dans un format différent vers CTF.

Ces fonctionnalités sont permises par l'interface de programmation de Babeltrace, écrite en langage C. Nous pouvons utiliser cette interface de programmation pour lire et écrire à notre guise des traces au format CTF. Une interface en langage Python est également proposée, ce qui permet une approche « script » très adaptée pour le traitement *a posteriori* de traces. Ce type de bibliothèque peut nous être utile dans plusieurs cadres. Premièrement, il est facile grâce à une telle bibliothèque de construire une fausse trace ressemblant exactement à ce que l'on souhaite, ce qui est un bon moyen de tester un outil quelconque d'analyse de trace. On peut aussi s'en servir pour construire une version filtrée ou réordonnée d'une trace : la trace en question peut être lue une première fois puis réécrite dans une nouvelle version en ne conservant que quelques éléments ou en changeant la position d'un événement. De la même manière, il est également facile de fusionner les événements de plusieurs traces.

Un outil tel que Babeltrace peut donc nous aider à connaître le contenu de nos traces et à faire des opérations dessus.

2.4.2 Visualisation de traces avec Trace Compass

Une trace se compose d'une suite d'événements chronologiques, chaque événement traduisant généralement un changement d'état ; il est intéressant de pouvoir suivre l'historique de ces changements d'état afin de détecter de possibles problèmes. Néanmoins, il est courant de devoir traiter avec des traces de grande taille. Dans ce type de cas, il n'est pas envisageable par exemple de travailler avec une liste textuelle des états atteints : il serait plus pertinent de fournir une représentation graphique rendant compte des différents états du programme ou du système au cours du temps. C'est ce que permet Trace Compass, un logiciel de visualisation

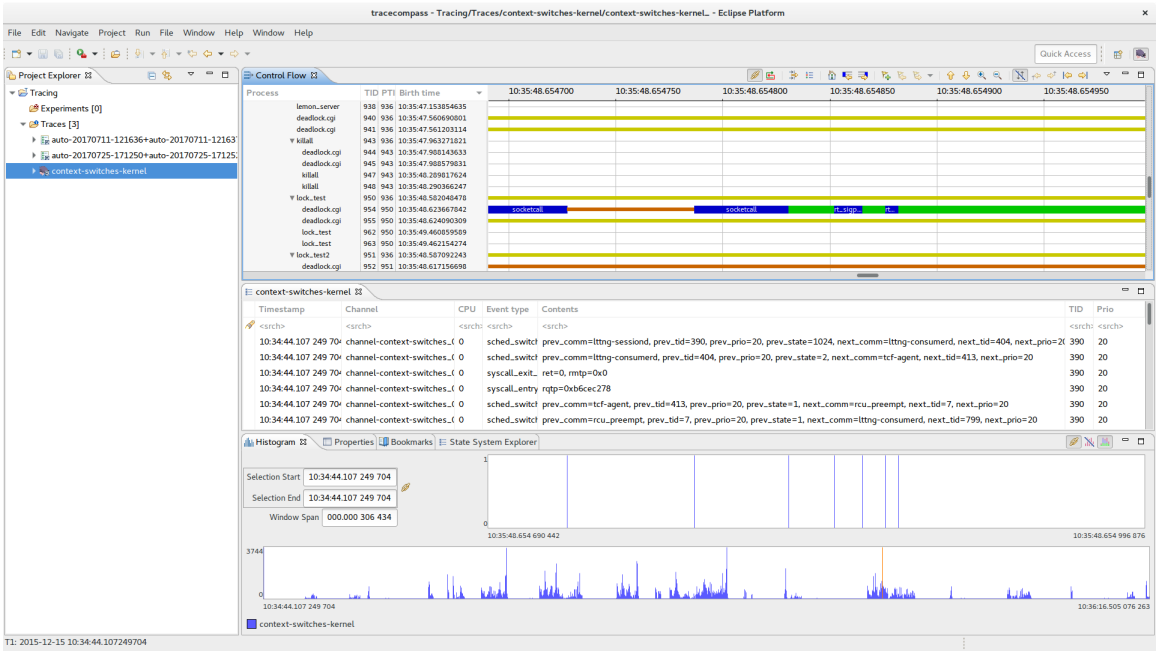

Figure 2.6 L'interface de Trace Compass.

interactive de traces enregistrées dans divers formats. Trace Compass est un logiciel à source ouverte écrit en Java et basé sur Eclipse. Son interface est montrée à la figure 2.6.

Trace Compass se base sur une structure de données optimisée pour la sauvegarde d'un grand nombre d'états au cours du temps (Montplaisir-Gonçalves et al., 2013). Cette structure, connue sous le nom d'arbre d'historique d'état (*state history tree*), se présente sous la forme d'un arbre dont la construction est optimisée et où chaque nœud représente un intervalle de temps auquel un état est associé. Une telle structure a comme contrainte de nous permettre d'accéder rapidement à l'état correspondant à une certaine estampille de temps, même si l'arbre comprend un très grand nombre de nœuds. Lorsque la trace est chargée dans Trace Compass, l'arbre est construit et chacun des états est calculé selon la trace et la logique de la vue courante. Dans la plupart des cas, un diagramme chronologique retrace alors les états inscrits dans l'arbre d'historique d'état à chaque instant. Les vues proposées par défaut dans Trace Compass s'appliquent le plus souvent à des traces du noyau Linux. Une des vues les plus importantes de Trace Compass, appelée vue du flux de contrôle (*control flow view*) indique, pour chaque processus rencontré dans la trace, l'état du processus à chaque instinct du traçage (actif, bloqué, en attente, en mode noyau, etc.).

Au-delà des vues proposées par défaut, il est possible de définir ses propres vues grâce à la fonctionnalité d'analyse XML (Kouame et al., 2015). Il s'agit principalement de décrire dans un fichier XML un automate fini qui donne la logique de la vue selon les événements rencontrés

dans la trace. Chaque transition de l'automate peut être reliée à une action modifiant une ou plusieurs valeurs dans l'arbre d'historique d'état. On construit donc ainsi un arbre qui donne l'évolution des états pertinents pendant toute la durée de la session de traçage, exactement de la même manière que pour les vues prédéfinies. Il est ainsi possible de créer sa propre vue. Les possibilités de représentation graphique des états restent néanmoins sommaires dans le cadre des analyses XML, puisqu'il faut s'en tenir à des lignes chronologiques très simples, alors que les vues par défaut montrent des représentations plus riches.

Un mécanisme permettant d'attacher des scripts Python aux traces ouvertes dans Trace Compass a également été mis en place pour réaliser des traitements plus fins sur les traces sans pour autant avoir à développer de nouvelles fonctionnalités en Java à l'intérieur de Trace Compass.

2.4.3 Synchronisation de traces

Lorsque l'on travaille avec plusieurs traces, il arrive que l'on doive avoir recours à des opérations de synchronisation entre celles-ci. Cela se produit typiquement lorsque les référentiels temporels dans lesquels les traces ont été relevées ne sont pas identiques. Il s'agit donc d'ajuster les estampilles de temps d'une ou plusieurs traces afin que les sources de temps soient alignées et que la chronologie soit cohérente.

Dans le cadre de ce recensement des techniques permettant de traiter les traces obtenues après analyse, nous nous intéressons à la synchronisation hors-ligne de traces. L'algorithme de synchronisation par enveloppe convexe est couramment utilisé pour aligner les sources temporelles de traces préalablement obtenues (Poirier et al., 2010). L'idée de cet algorithme est d'estimer les paramètres d'une fonction permettant de convertir les temps d'une source A en ceux d'une source B. L'algorithme se base sur une représentation graphique des temps d'envoi et de réception de messages entre les deux sources. Chaque message est représenté par un point dans un quart de plan où les abscisses mesurent les temps de la source A et les ordonnées les temps de la source B. À partir de là, l'algorithme identifie les droites de pente minimale et maximale pouvant représenter une fonction de conversion acceptable et sélectionne la droite bissectrice des deux droites précédentes comme représentation de la fonction de conversion optimale entre les deux sources. Au-delà de ce cas de synchronisation hors-ligne, des méthodes plus avancées de synchronisation en ligne de traces ont été développées (Jabbarifar and Dagenais, 2014) dans le cas où l'on souhaiterait faire intervenir la synchronisation de traces à mesure de l'enregistrement des événements.

Afin d'automatiser ce type d'opérations, Trace Compass propose de travailler avec des « expériences » (*experiments*) qui consistent essentiellement en un regroupement de plusieurs

traces. Au sein d'une même expérience, les événements des différentes traces apparaissent fusionnés et l'expérience est manipulable de manière transparente comme s'il s'agissait d'une trace réelle. Il est possible d'appliquer un décalage temporel à chaque trace de l'expérience à des fins de synchronisation manuelle dans les cas les plus simples. Dans les cas plus complexes, Trace Compass propose un algorithme permettant de synchroniser automatiquement les traces entre elles, à l'aide de l'information associée aux échanges de messages entre les nœuds tracés, en choisissant une des traces comme référence.

Conclusion de la revue de littérature

Dans cette revue de littérature, nous avons pu analyser les travaux précédemment effectués et les technologies déjà disponibles pour le calcul parallèle, l'utilisation des processeurs graphiques, le traçage logiciel et l'analyse des traces obtenues.

Nous avons eu l'occasion de distinguer certaines solutions comme étant plus ou moins appropriées dans le cadre de notre travail. L'analyse des technologies existantes pour la parallélisation de calculs sur processeur central et processeur graphique nous a permis de constater qu'une solution comme HSA s'insère réellement dans un contexte hétérogène en facilitant la communication entre appareils de différents types. Nous avons également relevé l'existence de plusieurs outils pertinents pour le traçage tant orienté vers l'analyse de l'activité du processeur central que vers celle du processeur graphique. Le couplage de LTTng avec des mécanismes déployés dans CodeXL et CLUST constitue une solution prometteuse dans le contexte de notre projet. Enfin, de nombreuses possibilités s'offrent à nous après la collecte de traces afin de fournir des mécanismes de traitement et d'analyse des données obtenues.

L'ensemble des outils recensés constitue un environnement riche pour nos travaux. Dans la suite de ce mémoire, nous exploitons les solutions les plus intéressantes afin d'explorer notre thématique de recherche.

CHAPITRE 3 MÉTHODOLOGIE

Dans ce chapitre, nous présentons la méthodologie utilisée pour mener à bien notre travail de recherche. Le détail de la méthodologie peut être utile dans le but de répliquer les résultats obtenus ou encore de les comparer à ceux obtenus sous une autre configuration. Nous détaillons ici à la fois la configuration matérielle utilisée et l'environnement logiciel dans lequel nous avons travaillé. En lien avec cette méthodologie, nous fournissons en annexe des indications sur la manière dont les outils proposés dans ce mémoire peuvent être utilisés.

3.1 Configuration matérielle

La station de travail utilisée est un ordinateur de bureau muni des composants suivants :

— processeur graphique dédié AMD Radeon R9 Nano ;
— processeur central Intel Core i7-4790 de fréquence 3,60 GHz ;
— mémoire vive DDR3, 32 Gio de fréquence 1600 MHz.

Le processeur graphique est ici l'élément fondamental pour notre projet. Celui que nous utilisons fait partie d'une génération récente compatible avec l'architecture HSA. D'autres processeurs graphiques plus récents de la gamme Radeon, dont les modèles Radeon Vega Frontier Edition ou Radeon Instinct pourraient également être utilisés pour donner accès à de nouvelles fonctionnalités telles qu'une gamme agrandie de compteurs de performance. De manière générale, les processeurs graphiques tendant à gagner en fonctionnalités promues par le modèle HSA, telles que le partage de mémoire virtuelle avec le processeur central, il est à prévoir que de plus en plus de modèles soient compatibles avec l'architecture HSA.

3.2 Environnement logiciel

Nous présentons dans cette section les différents logiciels installés sur notre poste de travail nous permettant d'obtenir les résultats présentés par la suite.

3.2.1 Système d'exploitation

Le système d'exploitation choisi est Ubuntu 16.04.3 LTS [1]. Ubuntu est une distribution GNU/Linux populaire, souvent supportée par les fabricants de cartes graphiques et autres types de matériel couramment utilisables sur une machine de bureau. Nous choisissons de

1. `https://www.ubuntu.com/`

plus une version proposant du support à long terme dans le but de garantir une certaine stabilité de notre environnement de travail.

3.2.2 Projet GPUOpen

Nous utilisons les logiciels de la plateforme ROCm en version 1.4 [2], issue du projet GPUOpen, pour faire fonctionner notre processeur graphique. ROCm inclut les composants suivants :

— *Radeon Open Compute kernel* (ROCk), le noyau Linux en version 4.6.0 accompagné des pilotes permettant la prise en charge du processeur graphique ;

— ROCr, l'environnement basé sur HSA permettant d'exécuter des noyaux de calcul sur le processeur graphique.

Les autres logiciels fournis dans le cadre du projet GPUOpen que nous utilisons sont les suivants :

— CodeXL [3], l'outil en interface graphique de référence d'AMD pour l'analyse de performance de programmes sur processeur central et processeur graphique, ainsi que ses sources ;

— GPUPerfAPI [4], la bibliothèque permettant de collecter les compteurs de performance sur le processeur graphique, ainsi que ses sources ;

— *Computing Language Offline Compiler* (CLOC) [5], le compilateur transformant des noyaux de calcul OpenCL en code exécutable dans un environnement HSA.

3.2.3 Outils liés au domaine du traçage

Nous utilisons le traceur LTTng [6] en version 2.10.0-pre ainsi que le convertisseur de traces Babeltrace [7] en version 2.0.0-pre1.

Le visualiseur interactif de traces Trace Compass [8] est installé en version 3.0.0.

Les scripts permettant de traiter les traces obtenues grâce à l'interface de programmation de Babeltrace sont écrits en Python 3 [9] et nécessitent donc l'interpréteur correspondant.

2. https://rocm.github.io/
3. https://gpuopen.com/compute-product/codexl/
4. https://github.com/GPUOpen-Tools/GPA
5. https://github.com/HSAFoundation/CLOC
6. https://lttng.org/
7. https://diamon.org/babeltrace/
8. http://tracecompass.org/
9. https://www.python.org/

Nous avons dans ce chapitre présenté la méthodologie que nous avons adoptée dans le cadre de notre projet pour aboutir à notre solution. Dans le chapitre suivant, nous nous intéressons en détail à cette solution. En particulier, nous y abordons l'implémentation de l'outil proposé et présentons les résultats tirés d'un cas d'utilisation.

CHAPITRE 4 ARTICLE 1 : LTTNG-HSA: BRINGING LTTNG TRACING TO HSA-BASED GPU RUNTIMES

Authors

Paul Margheritta <paul.margheritta@polymtl.ca>
Michel R. Dagenais <michel.dagenais@polymtl.ca>
Department of Computer and Software Engineering, École Polytechnique de Montréal

Submitted to: Wiley Concurrency and Computation: Practice and Experience

Keywords: performance analysis, software tracing, heterogeneous systems, parallel processing, GPU computing

Abstract

In this paper, we propose LTTng-HSA, a set of tools that allow for the collection of a single, unified software GPU trace in ROCr, an HSA-based API and runtime. HSA is a cross-vendor standard facilitating the programming of heterogeneous systems that include CPUs, GPUs and possibly other types of devices. Our open-source solution is generic and easily adaptable to diverse GPU runtimes or APIs. Using LTTng, a highly efficient Linux tracer, it collects different types of events over multiple executions of an application and aims to gather all the data into a single trace, offering an easy way to generate GPU-related traces. Our instrumentation is achieved simply by preloading libraries, without recompiling the target application, which makes it flexible and easy to use. The resulting traces, which include API call stack information, GPU hardware metrics, command queue and compute kernel profiling, are well adapted for post-processing and further analysis. Our solution also includes tracing data from the Linux kernel and proposes views for Trace Compass, an interactive trace visualizer.

4.1 Introduction

Tracing is a useful technique to better understand what is happening when a program runs. Software tracing tools such as LTTng allow us to record events, both on the Linux kernel side and on the user space side. The resulting trace conveniently helps to diagnose bugs,

including crashes, bottlenecks or unexpected delays.

However, this standard technique faces multiple challenges when dealing with complex systems. Among them, GPUs (Graphics Processing Units) have been gaining ground in recent years. GPUs were first designed as useful hardware accelerators for graphics, and are now extensively used for general purpose computing (GPGPU) (Owens et al., 2008). They feature a specialized, highly parallel architecture that can be far harder to fully exploit and monitor than conventional CPU-based architectures.

Beyond the case of a single GPU, multiple GPUs may be used simultaneously (Lee et al., 2013), possibly in combination with other processor architectures such as CPUs or FPGAs, which is known as a heterogeneous systems (Tsoi and Luk, 2010). Moreover, GPUs themselves are becoming increasingly complex, with dedicated architectures, CPU-like features, and a growing number of cores (McClanahan, 2010). As GPUs became more programmable, APIs have emerged to allow developers to use them for general purpose computing, including scientific computation (Götz et al., 2012). Popular APIs dedicated to general purpose applications include CUDA from Nvidia and the open standard OpenCL.

Specific tools may then be created for the performance analysis of GPU-based systems, and, in a broader perspective, of heterogeneous systems. The large GPU vendors have each developed and released their own performance analysis software : Graphics Performance Analyzer for Intel, Nsight for Nvidia, and CodeXL for AMD. However, these solutions remain vendor-specific, mostly proprietary efforts. Moreover, they prove unable to generate traces that encompass all the information that can be gathered about the GPU usage of an application, only focusing on a number a views. For analysis purposes, a unified trace would be easier to handle and would allow for a general understanding of how the GPU is used by our application.

The Heterogeneous System Architecture (HSA) is a cross-vendor standard allowing the programmer to work with several compute devices such as CPUs, GPUs and others simultaneously in a seamless fashion (Rogers, 2015). With HSA, programmers can make elements of a heterogeneous system work together for computation, without dealing with the very specific peculiarities of each device. HSA was proposed by a consortium comprising major actors such as AMD, ARM, and others. AMD then released a whole open-source toolchain as part of the GPUOpen project, including ROCr, an HSA-enhanced API and runtime that allows us to run GPU-compatible programs (Stoner, 2016).

We choose to base our work on ROCr for its open-source, cross-vendor, heterogeneous orientation. By modifying the ROCr project, we are able to provide extensive tracing and profiling of the runtime and API, allowing the user to gather information about how the program runs.

However, due to technical hardware limitations, it is not possible to gather in one pass all the relevant events in a trace. We propose to collect traces from several runs of the application and provide merging mechanisms to combine all the resulting events into a single trace. The whole process is an original addition to the current GPU-oriented performance analysis tools.

In this article, we present a set of tools allowing the gathering of runtime events representing GPU activity and data into a unified trace, encompassing all the information that can be obtained. The main contribution of this work is to provide efficient mechanisms to record all the relevant information for heterogeneous systems applications, in this case at the GPU and CPU levels. Furthermore, we propose several new analysis and views to display the exact state of each of the resources involved, CPU cores, GPU command queues and GPUs. This is invaluable to really understand and optimize the performance of heterogeneous applications.

The paper is divided into four main parts: first, we review the existing relevant related work; then, we describe the technical implementation of our solution and the views built for it, which is followed by an analysis of the overhead of our approach and the presentation of a use case for our solution; finally, we conclude and discuss possible future work.

4.2 Related work

Tracing is now a well-known technique in performance analysis. The main idea of tracing is recording runtime events during the execution of a program. These events are recorded via tracepoints that are inserted at relevant points in the code. Analyzing the resulting trace can then help us understand what caused the problems encountered at runtime. Traces may also be used to simulate the reexecution of a program (Burton and Kelly, 1998). Most of the recent work related to tracing has been done in a GNU/Linux environment, but tracing architectures have also been created for Unix systems and Microsoft Windows.

Tracing relies on an instrumentation of the program than can be either static or dynamic. A static instrumentation features tracepoints manually inserted in the code by the programmer. Dynamic instrumentation occurs when tracepoints are automatically inserted at runtime (Gregg and Mauro, 2011). Moreover, tracing exists at both the operating system kernel level and the user space level.

In this section, after discussing which GPU API to use, we present the most common tracing frameworks in use, and how they are used to provide CPU and GPU tracing and profiling.

4.2.1 Choice of a GPU API

Before discussing the tracing architecture, it is important to have a broad look at the APIs and runtime that can be used to program and control the GPU. We are looking for a compute-oriented (or GPGPU-oriented) API rather than a purely graphics-oriented one, since the use of GPU computation in many scientific research fields has become very common and is our focus.

Graphics-oriented APIs and compute shaders

OpenGL and Microsoft DirectX have originally provided graphics-oriented APIs, but in recent years they have adopted compute-oriented features. These are known as Compute Shaders for OpenGL and DirectCompute for Microsoft. Although these APIs may be interesting to study, they are only part of larger, mainly graphics-oriented frameworks. Here, we will choose to focus on GPGPU-specialized APIs. Moreover, DirectCompute is closed-source and only available for Microsoft Windows.

CUDA

CUDA (Garland et al., 2008) is the Nvidia reference GPU compute platform and API. Nvidia tends to primarily support CUDA over alternatives such as OpenCL, making CUDA a very popular choice on Nvidia platforms. However CUDA is closed-source, which makes it harder to study.

OpenCL

OpenCL is a reference cross-vendor, open-source, compute-oriented GPU API. Another very popular choice, it is compatible with Intel, Nvidia and AMD hardware. GPU tracing with OpenCL, combined with CPU tracing, has already been studied (Couturier and Dagenais, 2015).

HSA / ROCr

HSA (Heterogeneous System Architecture) is a recent cross-vendor standard that aims to allow programmers to work transparently in a heterogeneous environment, meaning that CPUs, GPUs and other devices can be controlled in the same way (Rogers, 2015). ROCr was released by AMD as an HSA-compliant GPU API and runtime. As part of the GPUOpen project, ROCr is open-source software (Stoner, 2016) (for the main part: functions from

the `hsa_ext_tools` module currently remain closed-source), which helps us understand its mechanisms. What is interesting in HSA and the ROCr runtime is a lower-level approach to GPU programming than OpenCL, giving us new challenges to solve. Moreover, other runtimes such as OpenCL can be built on top of HSA in order to improve the communication between compute devices. HSA-based runtimes were shown to achieve better performance than OpenCL runtimes (Mukherjee et al., 2016).

Our choice, therefore, is the HSA-based ROCr runtime and API.

4.2.2 CPU tracing and profiling

CPU tracing has already been extensively studied and has become a major tool for performance analysis and troubleshooting. GNU/Linux, as an open-source project, gives access to its source code and therefore provides a tracing-friendly environment. It is the reference platform for tracing and other performance analysis tools. Tracing is also available for Microsoft Windows with ETW and some Unix-based systems with DTrace.

Profiling refers to gathering certain metrics, which can be useful to point out the origin of any performance issue. Sampling hardware performance counters, and measuring the duration of specific function calls, are tasks that are typically seen as profiling.

In this subsection, we focus on the most prominent tools for CPU tracing and profiling on Linux platforms. Although they may serve the same final purpose, they nevertheless vary in terms of features and performance.

strace

`strace` is a simple tracing tool for Linux systems. It can record events associated with system calls for a program. However, this tool has serious performance issues: due to `strace` being based on the `ptrace` API, two context switches are made by `strace` for each traced system call, which can lead to a large overhead (Gregg, 2014).

Ftrace and perf

Ftrace and `perf` are two performance analysis tools that are directly built into the Linux kernel. This feature makes them easily available, adaptable and usable for different architectures, kernel versions and software distributions.

Ftrace (Rostedt, 2009) (Function Tracer) is a tracing platform designed for the Linux kernel. Ftrace includes multiple tracers that may be used to collect different types of information.

Although it was first designed to support only static tracepoints, Ftrace is now able to hook to dynamic kernel tracepoints (*kprobes*). Control and output is accessed using the TraceFS special file system, which offers a clean interface for Ftrace, although it may be unsettling to use in comparison with more intuitive tools that can be controlled with a simple command.

`perf` is a Linux kernel tool originally intended for gathering CPU performance counters, allowing the user to access hardware-oriented numeric data that can prove useful to diagnose performance issues. Besides these profiling features, `perf` has static and dynamic tracing capabilities, just like Ftrace. It also provides a front-end for Ftrace, meaning that `perf` can be used for Ftrace tracing in a more familiar, command-oriented fashion.

While Ftrace and `perf` can be useful to point out latencies and other performance problems in the Linux kernel, these tools are mostly geared towards kernel-side tracing and profiling, and do not provide the static user-space tracing capabilities we would need in a simple and secure way.

SystemTap

SystemTap is a tool that allows the user to attach scripts to statically or dynamically inserted tracepoints or probes. Much like Ftrace, its main focus was initially directed towards debugging the Linux kernel, but it has since evolved with user space probing features. SystemTap is similar to DTrace, a tool created for Unix systems that was intended to be used in the same debugging context. Both define specific scripting languages that may be used for various operations, including profiling, static and dynamic tracing at both kernel and user space level, although they were initially designed for dynamically inserted *kprobes*. Static user space tracing is defined as USDT (User-level Statically Defined Tracing) and looks like something that would fit our needs. However, SystemTap suffers from serious performance issues when dealing with multithreaded programs, due to a large synchronization overhead (Desnoyers and Dagenais, 2012).

Although SystemTap is a powerful and useful tool that could be appropriate in our case, it has performance issues for large applications.

LTTng

LTTng (*Linux Trace Toolkit Next Generation*) (Desnoyers and Dagenais, 2006) is a tracer for Linux systems that can be used for both kernel and user space tracing. It relies on hooking to static tracepoints and dynamically inserted kernel probes (*kprobes*). It was designed to provide traces with minimal overhead, especially for multithreaded programs. This high

performance is achieved by using separate buffers for each CPU core, which allows gathering events without requiring communication between the cores. The circular buffers used to collect the events are only written to disk when full, minimizing the writing operations and thus saving time and resources. If the disk throughput cannot handle the amount of data to be written, LTTng will choose to discard events rather than block the execution. Moreover, writing uses lockless synchronization, avoiding additional wait time Desnoyers and Dagenais (2012).

LTTng outputs traces in the CTF (Common Trace Format) standardized format. CTF is binary and was designed with a focus on performance. Tools have been created to process and analyze CTF traces. Babeltrace is the reference implementation of CTF and acts as a simple tool for CTF-to-text conversion, especially allowing us to print CTF traces in a human-readable way. Trace Compass, from the Eclipse Foundation, is a project that provides interactive visualization of traces, including a number of useful views for kernel-side traces. As an open-source project, Trace Compass can be extended with new views and analyses.

The efficiency of LTTng, its performance-oriented design, its ease of use on various Linux systems, and the various trace processing and analysis tools, make it the best tracing tool for our needs. We choose to use LTTng as our tracing architecture for our work.

4.2.3 GPU tracing and profiling

With the increasing use of hardware graphic acceleration for both purely graphic and scientific purposes, tools have been specifically designed to achieve a better understanding of the events happening on the GPU. Due to significant differences in hardware architectures between vendors and models, GPU-oriented performance analysis tools tend to be less generic than their strictly CPU-oriented counterparts.

By "GPU tracing," we refer to any tracing architecture that is explicitly aimed at understanding GPU-accelerated program executions. In this sense, GPU tracing may include CPU or GPU events. GPU profiling deals with the collection of GPU hardware performance counters and the gathering of timing information for GPU kernels.

Intel Graphics Performance Analyzer and VTune

Graphics Performance Analyzer (GPA) is a tool chain developed for Intel CPU and GPU architectures. It focuses on GPUs used for graphics purposes and is dedicated to the debugging and optimization of games. Under Linux systems, it supports applications using the OpenGL API. It is capable of recording traces for GPU-accelerated programs and providing

visual analyses using external trace viewers such as the one from Google Chrome. It also allows the user to visualize metrics for the profiled application, such as CPU and GPU usage and hardware counters.

VTune is Intel's reference performance analysis platform. It includes a GPU analysis module that, unlike GPA, focuses on general purpose applications, especially those built using the OpenCL API. For these programs, it offers features similar to those of GPA, including GPU usage analysis, timelines and hardware metrics.

GPA and VTune support is mostly limited to Intel hardware.

Nsight

Nsight is a development environment provided by Nvidia for GPU debugging and performance analysis. It can be integrated with Microsoft Visual Studio or Eclipse. It provides native CUDA debugging, OpenCL kernel tracing, call stacks and graphics-oriented performance tools for applications built with Microsoft Direct3D. Nsight requires a Nvidia GPU to work properly.

CodeXL and related AMD tools

CodeXL is AMD's counterpart to Nvidia Nsight. It exists as a standalone application or as a Microsoft Visual Studio extension providing GPU debugging, tracing and profiling. CodeXL supports compute applications built with OpenCL or HSA, as well as graphics applications built with OpenGL. It can generate a text-format trace, giving information about the API calls made and the GPU kernels executed during the course of a run.

CodeXL, in contrast with similar tools, is now open-source software, which means that we can easily look at its sources to understand how tracing and profiling mechanisms are implemented. CodeXL includes a number of libraries and tools that have been made available as standalone tools, making it easier for the user to launch analyses without using the traditional GUI, and giving them the opportunity to work on smaller parts of the whole project.

One interesting subproject of CodeXL is the GPU Performance API (GPA), a C++ library that allows us to collect GPU performance counters, giving us interesting hardware information. GPA supports compute applications such as those built with HSA or OpenCL, as well as graphics applications built with OpenGL or Microsoft DirectX.

4.3 Implementation

In this section, we describe how our solution is implemented and we look at how traces can be visualized.

4.3.1 General concept

Our solution is built around a number of shared libraries in the ELF format (Executable and Linkable Format) that can be preloaded by the user when executing an HSA-accelerated application. The preloading is usually done by setting the `LD_PRELOAD` environment variable to the path of the `.so` library that should be loaded. The preloaded library will intercept the relevant function calls and automatically add GPU-related static instrumentation in the form of LTTng user space tracepoints. When the application is executed with the preloading activated, the user gets a trace containing the relevant GPU-related events, without having to recompile the original application.

The current solution, relying on shared library preloading and function interception, was built as an improvement over an early solution that directly modified the ROCr runtime to add the relevant instrumentation using an HSA-specific interception mechanism. This mechanism relies on a table storing pointers to the API functions; therefore, intercepting an API call simply boils down to changing a pointer in this table. This early implementation lacked stability because of its dependency on runtime updates. Moreover, for our use case, this interception mechanism brought no significant advantage when compared to a `LD_PRELOAD`-based implementation, which gives us similar results without having to deal with the pointer table. The pointer table even has drawbacks as we will see in the section about the call stack *tracing target*.

While identifying the relevant events that could be instrumented, it appeared that, because of technical limitations from the HSA runtime, not every event could be instrumented simultaneously during the same execution of the application. Moreover, some events are synchronous (the trace event is generated at the same time as it happens), while some others are asynchronous (the trace event can only be generated somewhat later, for instance when callback information becomes available). These events may then need reordering in the trace file to keep the events sorted by time (Couturier and Dagenais, 2015). Therefore, our solution is to define what we call *tracing targets*. Each target corresponds to a set of GPU-related events that can be traced simultaneously. Whenever we want to trace our application, we choose one of the targets and get a trace containing the corresponding events.

In order to get a complete, unified trace, we propose mechanisms to merge and sort traces

obtained for the same application, traced for different targets.

4.3.2 Tracing targets

This subsection explores the different tracing targets defined:

1. the call stack target
2. the queue profiling target
3. the kernel timing target
4. the performance counters target.

Call stack target

The call stack target includes events needed to build a call stack from the HSA API functions calls on the CPUs. Two events are available: `function_entry`, indicating that an API function has been entered, and `function_exit`, indicating that a function has exited. The name of the function is provided in the payload of the event.

The shared library for the call stack target intercepts all the HSA API functions and replaces them with wrappers, augmenting the original functions with two tracepoints: the first one at the entry and the second one at the exit of the function (as shown in figure 4.1). As it may be painful to manually write all the replacement functions, the source of this library is automatically generated by a Python script. The script reads the header file from the ROCr runtime containing the prototypes of the API functions (`core/inc/hsa_internal.h`), and parses it to extract the names and argument lists for all the functions. Then, the library file with each newly instrumented function is generated by the script.

The call stack target is useful as a way to profile the API calls. Tracing an application with this target shows what API calls are executed in the runtime and how long they last, allowing the user to pinpoint abnormally long-lasting calls. This target highlights one of the drawbacks of using the runtime built-in interception mechanism. Indeed, as the pointer table mechanism is triggered by the `hsa_init` API function, this function cannot be intercepted for tracing purposes. Moreover, without further modifications, only the HSA API functions may be intercepted with the default mechanism, while ELF-based interception allows us to expand our call stack features with additional functions if necessary.

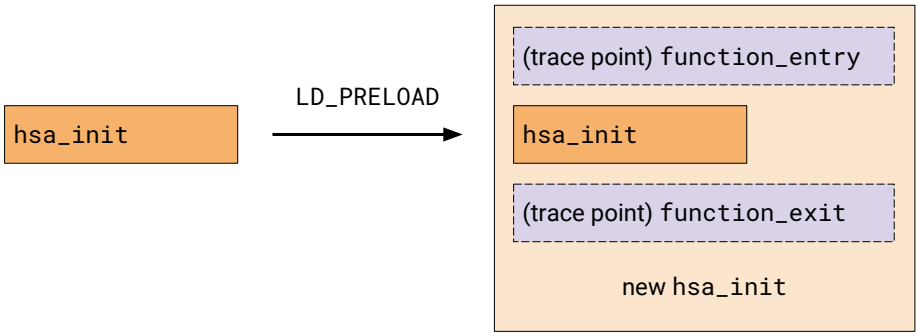

Figure 4.1 The `hsa_init` API function intercepted and instrumented within the call stack target. The new `hsa_init` function is a wrapper calling the original `hsa_init` function.

Queue profiling target

The queue profiling target is intended to provide more specific information about how the GPU user-level command queues are managed. In a general sense, *queues* are the mechanism for the CPU to send compute instructions to the GPU. HSA introduces user-mode queuing, which means that the communication between the CPU and the GPU does not involve the operating system or the GPU drivers. The queue profiling target features two kinds of events describing the state of each queue. Examples of these events are shown in the timeline of figure 4.2.

The creation and destruction of any queue (API functions `hsa_queue_create` and `hsa_queue_destroy`) are instrumented in a similar way as in the call stack target, with the payload of the event providing information about the GPU device involved, which is useful when multiple devices are used, and an identifier for the relevant queue. These trace events are simply called `queue_created` and `queue_destroyed`.

The target also provides events describing the submission of a packet into a queue, called `aql_packet_submitted`. There is no easy way to instrument this kind of event because, in HSA, the submission of a packet is made by filling the fields of a `struct`, not by calling a specific function that could be instrumented. However, the ROCr runtime provides a way to register a callback function that is called every time an AQL (Architected Queuing Language) packet is submitted into a queue. Registering a callback function that adds tracepoints thus provides instrumentation of the submission events.

This instrumentation is done with the `hsa_ext_tools_register_aql_trace_callback` function, but it will only work when setting the `HSA_SERVICE_GET_KERNEL_TIMES` environment variable to 0. This requires these events to be in a different target than, for example, the

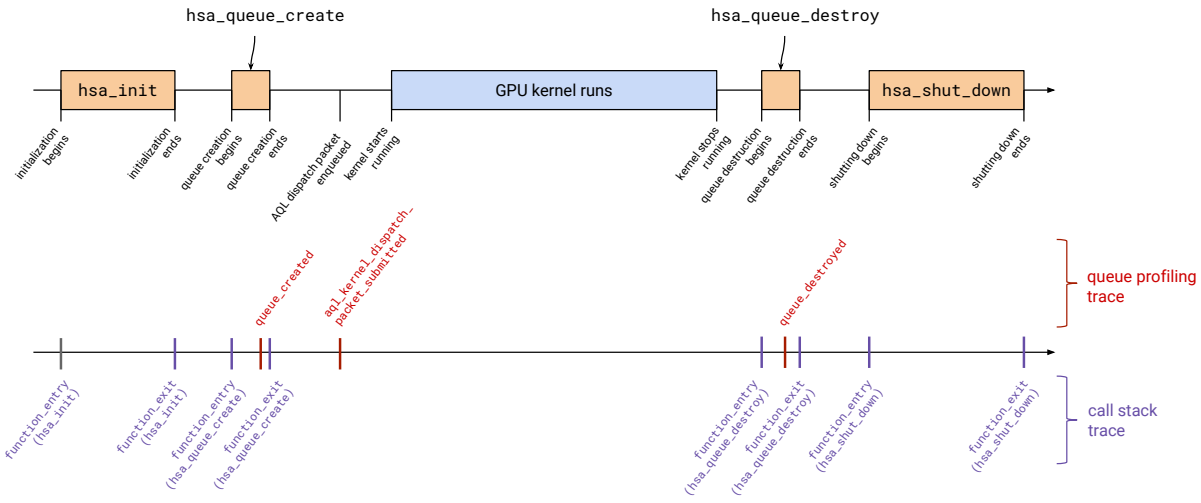

Figure 4.2 An execution timeline of a simple common HSA-based application with corresponding events from the call stack target and queue profiling target represented in a single trace.

kernel timing target, which requires this variable to be non-zero, as we will see.

Kernel timing target

The kernel timing target aims to generate trace events providing precise timing information for the compute kernels executed on the GPU. It creates trace events that mark the beginning and the end of the execution of a GPU kernel. Again, there is no easy way to instrument the beginning and end of a kernel execution. This is because of the low-level nature of HSA: while higher-level APIs such as OpenCL provide functions specifically dedicated to dispatching a kernel (see the `clEnqueueNDRangeKernel` function in OpenCL), HSA relies on more generic storing functions that may be used in unrelated contexts.

The ROCr runtime provides pre-dispatch and post-dispatch callback functions that can be specifically used for the purpose of wrapping around a GPU kernel dispatch. We may be tempted to see this as an interesting solution to get timing information for the GPU kernel executions. However, this solution is not suitable since the callbacks are only executed before and after the dispatch (the packet enqueuing operation) rather than being called just before and after the GPU kernel execution itself. This means that, in the best case, the resulting times will not perfectly match the real beginning and end times that were expected. Actually, in the worst case, the results may be very far from what we expect, because there is no guarantee that the kernel will be executed just after being dispatched.

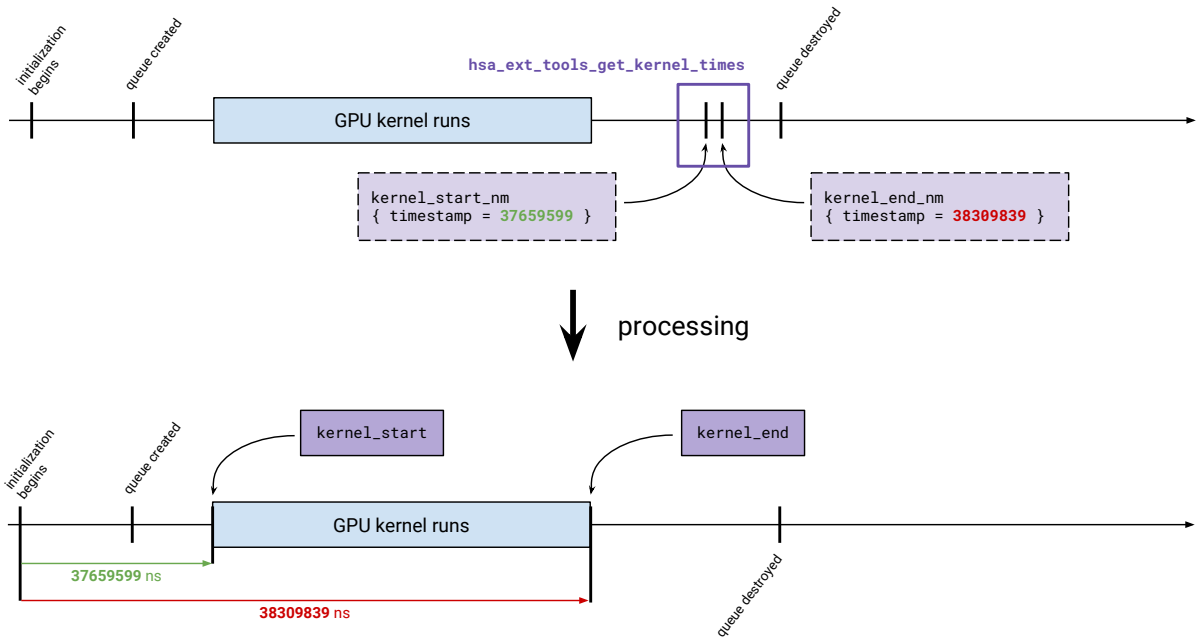

Figure 4.3 An example of event sorting for the kernel timing target. The `kernel_start_nm` and `kernel_end_nm` events are sorted to appear, as expected, at their time of occurrence in the trace.

Another solution is offered with the `hsa_ext_tools_get_kernel_times` function. This function seems a perfect fit for our needs, but it imposes some other constraints. It will only work if we choose to use a specific kind of queue known as a *profiled queue*. As mentioned before, it also requires the `HSA_SERVICE_GET_KERNEL_TIMES` environment variable to be defined to a value other than 0. These are two reasons why we chose to consider the GPU kernel timing features as a separate target. However, dealing with these constraints is not difficult. Intercepting the queue creation call, replacing it with a call to `hsa_ext_tools_queue_create_profiled`, and not setting any additional HSA-related environment variable is sufficient to deal with these constraints.

While all the kernel timing architecture is initialized when creating a new queue, we still need to read the timing results. As this information is only available after the GPU kernel execution, we are in the case where we deal with asynchronous trace events. We chose to collect the recorded timing information for all the kernels when destroying the command queue, instead of trying to access this information after each GPU kernel execution. While this choice requires the runtime to store timing information for every executed GPU kernel, collecting timestamps for every GPU kernel would cause additional overhead without yielding better results.

Our solution, then, relies on intercepting the `hsa_queue_destroy` call to add a step dedicated to gathering and processing GPU kernel timings. For each GPU kernel execution start, we will create a `kernel_start_nm` trace event, and for each GPU kernel execution end, a `kernel_end_nm` trace event will be generated. Each of these events provides the relevant timestamp in their payload. Here, the `nm` stands for *non-monotonic*, meaning that these events are collected in an asynchronous manner. They do not appear where they were expected in the trace, and will need to be sorted according to the provided timestamp.

When looking at the recorded timestamps, it appears that the `hsa_ext_tools_get_kernel_times` function uses the monotonic clock from the host. A monotonic clock timestamp is not sufficient to know where the event should be moved to in the trace, because we do not know the equivalent real-world time. However, we know that the very first trace event from all the traces generated from our targets will be the `function_entry` event associated with the `hsa_init` function. Knowing that, we can record the monotonic clock timestamp at the exit of `hsa_init` and compute the time delta with our beginning and end timestamps. After merging all the events into a single trace, we will simply have to move the asynchronous events to the beginning of the trace and apply an offset corresponding to the time delta previously computed. In our case, we provide a nanosecond-precision time delta, relative to the start of the unified trace, in the payload of the `kernel_start_nm` and `kernel_end_nm` events. While this simple synchronization strategy, shown in figure 4.3, is easy to implement and will work for basic cases, more specialized and complex trace synchronization algorithms could be used instead (Jabbarifar and Dagenais, 2014).

Performance counters target

The performance counters target intends to provide information about the GPU hardware metrics. It includes events showing the values of the counters for each GPU kernel execution. The GPU performance counters can be accessed with the GPA (GPU Performance API) library. With the help of GPA, we set up a profiling architecture that automatically samples and records the performance counters while a GPU kernel runs. Every time a GPU kernel finishes executing, one trace event for each sampled counter is recorded. There are four trace events defined for performance counters, each of them adapted to a specific counter type. For instance, `kernel_perf_counter_uint32_nm` is the trace event dedicated to 32-bit unsigned integer counters. Just like the GPU kernel timing events mentioned before, these events are recorded asynchronously and may have to be sorted in a trace.

To help us automate the sampling of performance counters, we chose to use the pre-dispatch and post-dispatch callback functions available in the ROCr runtime. Defining such call-

backs is also a technique used in CodeXL. To collect performance counters, we need to wrap the GPU kernel execution in a sampling context, a sampling session, and a sampling pass. Contexts, sessions and passes are opened in the pre-dispatch callback and closed in the post-dispatch callback. Callbacks are set up when creating a queue by intercepting the `hsa_queue_create` API call.

It should be noted that only one context may be opened at a time. For this reason, we chose to use mutex locking around the code section that uses a context, in order to force at most one context to be open at once. This solution allows parallel GPU kernels to be profiled in multithreaded cases. However, it may have a significant impact on how the GPU is used, because it serializes the execution of GPU kernels, possibly leading the GPU to be largely underused. Tracing with the performance counters target may therefore cause a large overhead for multithreaded applications.

Pre-dispatch and post-dispatch callback functions are only made available when the Soft CP mode is enabled, which is done by setting the `HSA_EMULATE_AQL` environment variable to 1. Moreover, the GPA library needs to be dynamically loaded. These specific settings and the potential impact of performance counters justify the definition of a specific tracing target for them.

4.3.3 Additional Linux kernel tracing

In addition to the different user space events mentioned before, some Linux kernel-side tracing may provide information about the interactions between the operating system and the GPU. The `amdgpu` Linux kernel driver features a number of tracepoints that may be activated. As these tracepoints are not part of the default LTTng setup, we wrote the corresponding stubs in the adaptation layer, enabling LTTng to access these tracepoints. These tracepoints are designed to provide memory-related information, especially the management of buffer objects (for instance, `amdgpu_bo_create` is hit when a buffer object is created by the driver).

While we can easily generate a kernel trace for these events, things get more interesting when combining the Linux kernel tracing with one of our user space tracing targets. LTTng, indeed, is capable of tracing both in the Linux kernel and in user space in the same session, and can generate the corresponding trace with both kinds of events (Fournier et al., 2009).

Combining the Linux kernel tracing with the call stack tracing target shows which API calls involve the Linux kernel and which do not. API functions that typically trigger a lot of memory management by the driver include `hsa_init`, `hsa_shut_down`, `hsa_queue_create` and `hsa_queue_destroy`. This kind of information may prove very useful to diagnose driver-

related problems.

4.3.4 Trace merging and event sorting

As mentioned before, some GPU-related events can only be recorded in an asynchronous manner. This means that they only appear in the resulting trace after the actual event has happened, and that they will have to be moved to the correct time location in the trace. This is what we call *event sorting*. We also discussed the merging of traces generated from different targets into a single trace. In practice, after obtaining our traces, we will first merge them into a single trace, and then sort the events that need further processing. Our implementation required that the events to sort appear at the end of the merged trace; therefore, we will first generate traces that do not need processing (such as those obtained with the call stack target or the queue profiling target), and then traces that need event sorting (such as those including kernel timing and performance counter events).

Since comparing variations between executions of a program is a challenging task that may require specific tools (Trumper et al., 2013), it should be noted that merging traces can only be performed safely if they were generated by similar executions (i.e. performed the same operations scheduled in the same order). If one of the executions of the application turns out to last much longer than the others, merging this trace with other shorter traces may not be relevant. As will be shown in the next section, the call stack, kernel timing and queue profiling targets allow us to generate traces with small overhead in comparison with the initial run, therefore limiting the risk of obtaining very dissimilar executions. However, to ensure that we get the most accurate merged trace, one may want to generate several traces for each target and try merging the traces from the most similar executions.

Babeltrace, the trace format converter that mainly acts as a trace pretty-printer, provides an API to read and write CTF traces. We successfully used its Python bindings to merge kernel timing events into a call stack trace, or a queue profiling trace, and move them to the correct time location. The technique is applicable to any synchronous or asynchronous event. The main idea is to copy an event from the source trace, compute its time delta with a reference point in the trace (in most cases, the exit of `hsa_init`, marked as a `runtime_initialized` trace event), and compute the resulting timestamp in the target trace. For asynchronous events, the corresponding time delta is provided in the payload.

Another option worth considering is using Trace Compass, the interactive trace analysis tool, to merge traces and sort events. The purpose is slightly different because Trace Compass provides no way to output the final, unified trace: events are simply merged "on the fly," for analysis purposes, without reconstructing a valid CTF trace. In Trace Compass, the

concept of merged trace is represented by *experiments*. Multiple traces can be stored into an experiment to make their events appear combined, offering analysis capacities on this newer trace. A static time offset can also be applied to each of the original traces, but it is not possible to tune the offset for each event, as would be necessary for our asynchronous events. This means that this technique cannot be used to integrate kernel timing events or performance counters events to a larger trace. However, merging synchronous events such as those from the call stack target and the queue profiling target is possible. It can be achieved by having a same trace event in both traces and synchronizing on it to find the right time offset to apply.

4.3.5 Trace Compass views

Trace Compass is an interactive trace visualization tool. It provides useful default views for all kinds of purposes, especially for extensive performance analysis of the operating system. As open source software, it can be improved with new views that can be written in Java, but it also offers a way to define views at a higher level, known as *XML analyses*. XML analyses provide some markup allowing the user to define the finite state machines (Kouame et al., 2015) that will be used to populate the *state history tree*, an optimized data structure designed to store interval data (Montplaisir-Gonçalves et al., 2013).

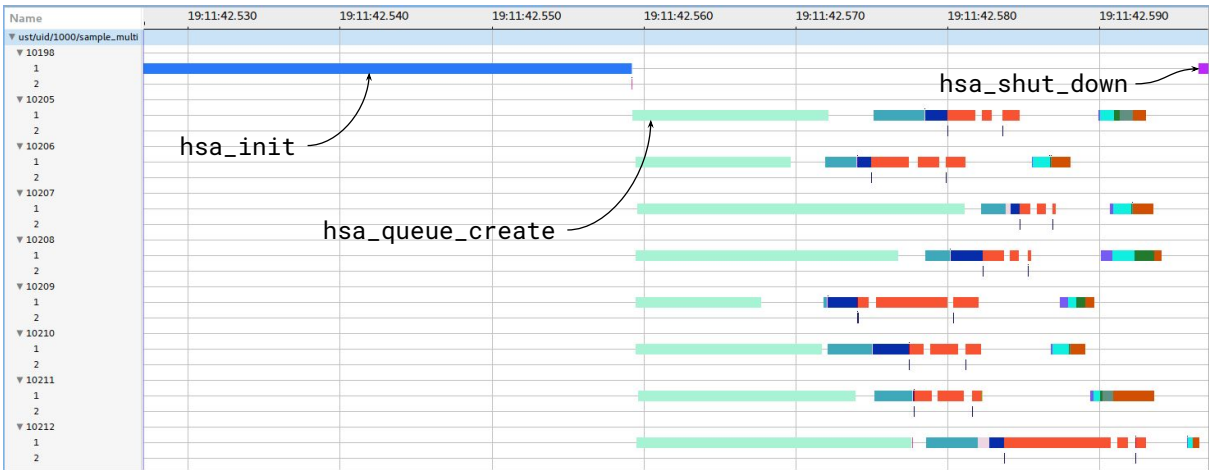

Figure 4.4 The call stack view of a simple application running eight kernels concurrently. The main thread and the eight children threads are shown, with two levels of nested calls in each case. For each segment, the corresponding function call and duration are shown when hovering on it.

We created two XML analyses:

— The *call stack view* (figure 4.4) is based on events from the call stack target. For each

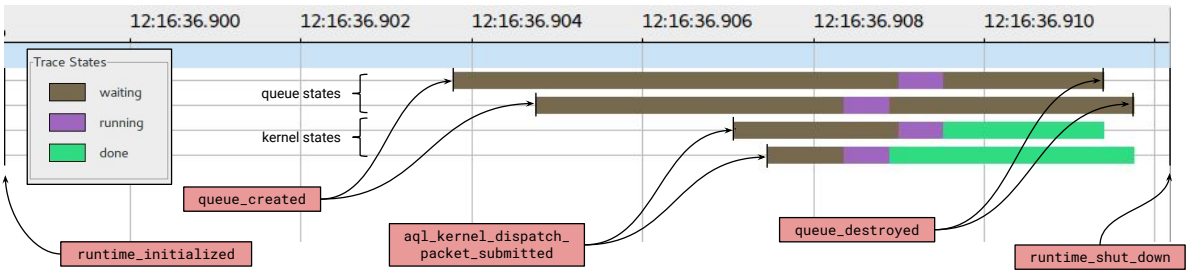

Figure 4.5 The queue profiling view of a simple application running two GPU kernels dispatched from two separate queues. The view shows the state of each queue (first two timelines) and each kernel (last two timelines). For each segment, the corresponding state and duration are shown when hovering on it. All the trace events linked with the view, including those indicated here, are interactively visible when using the view in Trace Compass.

API call, an interval is created with a length depending on the duration of the call. Nested calls are shown by using several height levels.

— The *queue profiling view* (figure 4.5) can be applied to merged and sorted traces, combining events from the queue profiling target and the kernel timing target. It represents the state of a queue during the course of the execution: non-existing, waiting (when no GPU kernel dispatched from this queue is being executed), or running (when a GPU kernel dispatched from this queue is currently running). The state of GPU kernels is also represented as waiting, running, or done. The state of a GPU device with several queues could be easily inferred from the state of each of its queues.

4.4 Experimental results

In this section, we focus on the performance of our solution and how it can be used to detect unexpected behaviour in a GPU-based application.

4.4.1 Overhead

We tested our solution on two sample programs to assess its performance. Our goal is to evaluate the overhead for each of our tracing targets. To estimate these values for each sample program, we compare over 100 runs the execution time for the program without instrumentation, and with inserted instrumentation, for each of the tracing targets.

Both sample programs are written in C++.

— `sample_test` is largely inspired by the sample program that is already provided in the runtime. It creates a command queue, then dispatches a compute kernel in it. The

GPU kernel copies a value from an input vector to an output vector. It is executed on a vector containing 2^{20} items. When the work is done, the program exits.

— `sample_multi` does the same processing as `sample_test`, but 8 times in parallel. 8 command queues are created and packets are dispatched in parallel using the `pthread` library.

Table 4.1 Experimental results for the sample programs

target	*none*	CS	QP	KT	PC
`sample_test`					
mean time (ms)	58.56	59.79	60.78	61.36	91.62
st. dev. (ms)	1.32	0.64	0.56	0.80	0.76
overhead	0.00%	2.10%	3.79%	4.78%	56.45%
`sample_multi`					
mean time (ms)	75.29	76.09	79.12	79.43	138.79
st. dev. (ms)	3.80	2.35	2.92	3.21	3.51
overhead	0.00%	1.06%	5.09%	5.50%	84.34%

(CS = call stack target, QP = queue profiling target,
KT = compute kernel timing target, PC = performance counters target)

The results are shown in table 4.1. For both programs, as expected, the data shows that a small overhead, between 1% and 5.5% of the original time, is added when using the call stack, compute kernel timing and queue profiling targets. However, the overhead added when using the performance counters target is significantly higher (around 55%), which is not surprising since sampling hardware metrics implies the use of a whole dedicated library. With the multithreaded program, the overhead is even higher, which was expected due to the mutex locking used in the performance counters target. Nevertheless, additional timing information, collected after disabling the callbacks setting up the gathering of performance counters, shows that the overhead is mainly added by the initialization of the required profiling architecture, not by the collection of performance counters itself. This makes the overhead more acceptable. It should also be noted that, even for a simple multithreaded application instrumented with the performance counters target, no explosion of the execution time is observed: at worst, our program runs in less than twice the initial duration (around 85% longer).

It is also interesting to compare the difference in overhead according to the tracing target.

The call stack target, which is a very classic and straightforward kind of instrumentation, adds the smallest overhead. The compute kernel timing and queue profiling targets, however, suffer a larger overhead because they use specific, more advanced features such as profiled command queues and callback functions.

We conclude that our experimental results match what was expected and show satisfying performance for our sample programs.

4.4.2 Matrix multiplication use case

We tried our proposed tools on a practical use case to verify that it actually allows us to detect significant performance issues when using GPUs for computing. In order to simulate a realistic scenario, we worked with OpenCL kernels instead of pure HSA kernels, and compiled them to HSA-compatible kernels using the CL Offline Compiler (CLOC) tool from the HSA Foundation (Sander, 2016).

Our tests revolve around the problem of matrix multiplication, which is a classic computation that appears in most practical computation-oriented uses of GPUs. It is therefore a well-documented problem that is likely to take place in a real-life context. We work with square matrices of 2^{20} elements stored as one-dimensional arrays with a row-major structure.

Cache hit ratio

Let's suppose that our first matrix multiplication GPU kernel, named `matrix_multiply`, takes longer than expected to finish. Using our tools, we generate a unified trace and get the following information:

— Thanks to the kernel timing target, we find that the computation lasts around 57.9 ms.
— Thanks to the performance counters target, we find that the value of the `CacheHit` counter is around 50.5%, which leaves a lot of room for improvement.

A low cache hit rate is often a sign of an algorithm that fails to take advantage of spatial locality. In our case, this especially happens when the matrices are accessed in a way that prevents existing cache lines from being used. This is a common problem since any mistake in the indexing of matrices can lead to non-contiguous elements being accessed consecutively, creating cache faults. Looking back at our implementation, we can fix our indexing problem and write the `matrix_multiply_fixed` GPU kernel. Using our tools again, we get the following data for this algorithm:

— The computation lasts around 12.3 ms, 4.7 times faster than before.
— The value of the `CacheHit` counter is around 74.1%, which is much better than before.

Thanks to the timing information and cache hit ratio data from our tools, we managed to fix our problem. `matrix_multiply_fixed` is the typical naive implementation of the matrix multiplication.

Vector and scalar ALU instructions

We now seek to further improve our algorithm. Some other striking figures appear when looking at the performance counters gathered in the trace:

— The value of the `VALUInsts` counter (the average number of vector ALU instructions executed per work item) is around 15,000.

— The value of the `SALUInsts` counter (the average number of scalar ALU instructions executed per work item) is around 1000.

These results show that a large number of vector and scalar instructions take place for each work item. Although we cannot use this information to pinpoint a specific problem, this indicates that further optimizations are possible, which is consistent given that the current kernel is very straightforward.

For such use cases, a common optimization is to use tiling mechanisms in order to take advantage of GPU local caching. At the same time, as a side effect of improving the algorithm, many ALU instructions could be avoided, especially when computing matrix indices. After writing and tracing a tiled version of the matrix multiplication algorithm, named `matrix_multiply_tiled`, we get the following results:

— The computation lasts around 5.10 ms, 2.4 times faster than the previous version.

— The value of the `VALUInsts` counter is around 2000.

— The value of the `SALUInsts` counter is around 70.

The number of ALU instructions could therefore be a useful indicator of the performance of algorithms. Here, local caching is used to make our GPU kernel more optimized. We may think that the cache hit ratio data would be interesting to look at in such a case. However, it could not be used in our case because the corresponding counter only takes L2 cache hits into account, while local caching takes advantage of the L1 cache. Therefore, the cache hit ratio is not as relevant in this case. Recent devices, however, can give access to both L1 and L2 cache hit ratios.

4.5 Conclusion & future work

In this paper, we explored how a low-level GPU API and runtime could be instrumented to generate useful traces, and how such traces could be combined to provide a unified, ready-

to-use, output trace. We showed that it is possible to build an architecture to automatically instrument the ROCr runtime by simply preloading libraries, without recompiling the target application. Beyond that, we proposed Linux kernel tracing for the `amdgpu` drivers, which offers a new perspective on GPU tracing, as well as Trace Compass views to help understand the resulting traces. We showed that such GPU tracing could be done without excessive overhead. The implemented mechanisms use open-source solutions, making them easy to adapt and improve, and are usable with any HSA-based API. For non-HSA APIs and runtimes, similar mechanisms may be set up with few changes, especially for call stack and queue profiling purposes.

Future work may include further analysis of the Linux kernel-side tracing, with specific processing, views and analyses. Even though some general principles may be re-used, our solution for instrumentation could be made more generic, which would make it directly usable for graphics-oriented APIs such as OpenGL or other runtimes that are not expected to be built on top of HSA in the near future. Additionally, the event sorting and trace merging could be further investigated in order to have more streamlined methods. Finally, several unified traces of the same application may be collected, and, using similar methods, could be averaged in one final trace, giving a typical execution profile of the application.

4.6 Acknowledgements

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

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous revenons sur les résultats présentés dans l'article précédent ainsi que sur les contributions additionnelles réalisées au cours du projet de recherche.

5.1 Estimation du surcoût

L'estimation du surcoût effectuée dans le cadre de l'article exposé précédemment montre un surcoût globalement faible pour notre solution. En effet, nous avons considéré que le temps d'exécution ajouté lors de l'utilisation de nos outils était suffisamment faible pour garantir que le comportement du programme tracé ne sera pas modifié excessivement.

Cette observation n'est pas anodine, car elle s'insère dans une question plus générale : celle de l'impact des outils d'analyse sur le phénomène à observer. Il est crucial que les outils d'analyse utilisés aient une influence faible en matière de temps d'exécution et de mémoire utilisée, sans quoi l'état du système changerait, possiblement au point de modifier le problème initialement observé. Si l'impact de l'outil d'analyse est trop important, le problème pourrait être amplifié ou pourrait même disparaître (on parle dans ce cas d'un *heisenbug*).

Dans notre cas, un des facteurs qui permettent d'aboutir à un surcoût temporel raisonnable est l'utilisation d'outils et mécanismes à faible surcoût à la base. En effet, une des caractéristiques principales du traceur LTTng est son très faible impact sur le système tracé. De plus, les bibliothèques correspondant aux cibles de traçage `call_stack`, `kernel_timing` and `queue_profiling` se contentent d'ajouter aux fonctions existantes des opérations représentant un faible surcoût, comme la définition d'un point de trace ou d'une fonction de rappel, ou encore l'écriture d'une valeur dans une table d'association. Il en est en revanche autrement pour la cible de traçage `perf_counters` qui doit sérialiser l'exécution des noyaux de calcul pour obtenir les résultats attendus : dans ce cas, un ralentissement non négligeable pourra être observé, mais ce surcoût est inévitable dans ce contexte.

Au-delà de l'intérêt de maintenir un surcoût général faible, l'étude de l'impact temporel de chaque cible de traçage peut jouer un rôle déterminant dans la synchronisation de traces. En effet, si nous savons qu'une cible de traçage particulière occasionne un certain délai, ce délai peut être pris en compte au moment de fusionner les événements de cette cible avec d'autres événements dans le but de construire une trace cohérente.

5.2 Étude de cas

Notre étude de cas, centrée autour du problème de multiplication de matrices, cherche à montrer quel est l'intérêt concret des outils d'analyse de performance dans le développement de programmes accélérés matériellement. La multiplication de matrices se retrouvant de manière presque systématique dans les contextes scientifiques et graphiques, il est utile de constater que nos outils peuvent améliorer la compréhension de problèmes de performance dans ce contexte.

Les analyses faites sur les compteurs de performance liés aux fautes de cache et aux instructions vectorielles et scalaires ne sont que des exemples d'exploitation de données récupérées par des mécanismes de traçage. D'autres compteurs de performance peuvent se révéler pertinents pour détecter des comportements anormaux, comme la quantité de données lues depuis la mémoire vidéo. Les interprétations possibles ne se bornent néanmoins pas à l'étude des compteurs de performance : en cas de problème dans la gestion des files d'attente du processeur graphique, une simple représentation graphique des durées d'exécution des différents noyaux de calcul et des états des files mises en œuvre pourrait par exemple suffire à comprendre l'origine d'un problème.

L'analyse des compteurs de performance a également permis de mettre en lumière les problèmes causés par les différences d'architecture entre processeurs graphiques. En effet, selon les modèles de processeurs graphiques, des compteurs de performance différents peuvent être proposés. Ainsi, l'existence de compteurs séparés pour les fautes de cache L1 et L2 n'est disponible que sur les modèles les plus récents, limitant ainsi les possibilités de diagnostic. La grande variabilité des architectures de processeurs graphiques apparaît donc clairement comme un défaut au moment d'analyser concrètement l'utilisation de l'accélérateur par un programme.

5.3 Contributions additionnelles

Dans le cadre d'un environnement d'exécution basé sur HSA, qui privilégie les interactions directes entre appareils destinés au calcul, le noyau du système d'exploitation n'est par exemple pas impliqué dans la communication entre le processeur central et un processeur graphique. Il est donc pertinent d'étudier dans quel type de situation les pilotes du processeur graphique sont actifs et dans quelle mesure ils interviennent dans le déroulement des calculs. C'est la raison pour laquelle nous avons également généré des traces combinant des événements de l'espace utilisateur avec des événements du noyau du système d'exploitation. Cette étude n'a pas été approfondie dans le cadre de ce projet de recherche, mais elle illustre l'idée que

l'analyse de l'activité du processeur graphique exploite tous les domaines du traçage, ce qui en fait un sujet d'étude intéressant.

CHAPITRE 6 CONCLUSION

Pour conclure ce mémoire, nous faisons une synthèse du travail effectué dans le cadre de ce projet de recherche et analysons les limitations rencontrées. Nous formulons ensuite des propositions d'amélioration qui peuvent être retenues comme pistes de recherche futures.

6.1 Synthèse des travaux

Au cours de ce projet de recherche, nous avons premièrement cherché à découvrir et prendre en main les mécanismes existants pour l'analyse de l'activité du processeur graphique, et, plus généralement, de systèmes hautement parallèles. Nous avons fait le choix du traceur LTTng et avons décidé de l'utiliser dans le cadre d'environnements d'exécution hétérogènes basés sur HSA. Il est apparu qu'il était possible d'insérer des points de trace dans les fonctions de l'environnement d'exécution grâce au préchargement d'une bibliothèque logicielle, ce qui a répondu à notre première question de recherche.

Étant donné qu'il est apparu impossible de collecter en une unique exécution des informations de diverses natures, nous avons fait le choix de travailler sur plusieurs exécutions d'un même programme. Nous avons alors développé plusieurs bibliothèques proposant différentes instrumentations, dans l'idée de récolter différentes traces d'un même programme qui pourraient ensuite être combinées. Cela nous a amenés à nous pencher sur les questions liées au traitement de traces après leur génération, ce qui comprend les problèmes de fusion, tri et synchronisation de traces. Ces efforts s'inscrivent dans l'idée de constituer une trace unifiée temporellement cohérente malgré la présence d'événements synchrones et asynchrones. Cette partie de notre projet a permis de répondre à notre deuxième question de recherche.

Après avoir travaillé sur des outils permettant de générer des traces unifiées rendant compte de l'activité du processeur graphique au cours de l'exécution d'un programme, nous avons cherché à rendre la détection de problèmes plus facile à l'aide d'outils graphiques. Nous avons exploité les fonctionnalités du visualiseur Trace Compass afin de bâtir nos propres analyses, ce qui permet d'offrir des vues utiles pour l'analyse de performance et la détection de problèmes. Cela nous a permis de répondre à notre troisième question de recherche.

L'ensemble de ce projet de recherche aboutit donc à une solution logicielle permettant de générer des traces utiles pour l'analyse de l'activité du processeur graphique, moyennant un surcoût faible. Les outils développés exploitent le traceur LTTng et se basent sur l'architecture HSA. Des mécanismes de traitement et d'analyse visuelle des traces obtenues ont été proposés.

Nous en concluons donc que nos objectifs de recherche ont été atteints.

6.2 Limitations de la solution proposée

Au cours du développement de notre solution, plusieurs limitations sont apparues.

Premièrement, nous nous sommes heurtés au manque de solutions universelles pour la gestion des traces après leur obtention. Nous avons développé nos propres outils pour répondre au besoin de fusion et tri d'événements de traces, mais ceux-ci sont relativement spécialisés pour nos besoins et s'adaptent difficilement à des besoins plus génériques.

Ensuite, les méthodes de synchronisation employées pour obtenir une trace unifiée manquent de précision. Nous aurons bien le résultat attendu si les différentes traces combinées proviennent d'exécutions très similaires, mais si les exécutions divergent un peu trop, les événements pourraient ne pas se retrouver dans l'ordre espéré. Il est en général nécessaire de tracer plusieurs fois le même programme avec chacune des cibles afin de pouvoir choisir les exécutions les plus similaires pour la fusion.

Enfin, les vues que nous proposons pour interpréter les traces obtenues restent à l'heure actuelle relativement sommaires et ne permettent pas de tirer profit au maximum des informations collectées.

6.3 Améliorations futures

Plusieurs améliorations sont encore possibles dans le cadre de ce projet de recherche. Nous formulons ici quelques pistes pouvant être suivies à l'avenir.

Une première étape d'amélioration consisterait à améliorer le mécanisme de synchronisation développé ici pour gérer la fusion des événements de deux traces. Nous pourrions implémenter un des algorithmes spécifiquement dédiés à la synchronisation de traces afin d'obtenir un résultat plus fiable.

De plus, nous pourrions mettre à profit les avancées récentes dans le domaine de la conception de vues personnalisées dans Trace Compass. Des vues permettant de représenter directement les valeurs des compteurs de performance pourraient être développées en adaptant la manière dont les compteurs sont actuellement représentés dans nos traces.

Enfin, l'utilisation de traçage du côté du noyau du système d'exploitation dans le but d'analyser l'activité du pilote permettant le support du processeur graphique pourrait être approfondie, dans le but de permettre la détection de possibles problèmes intervenant directement au niveau du système d'exploitation.

RÉFÉRENCES

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow : Large-scale machine learning on heterogeneous distributed systems", *arXiv preprint arXiv :1603.04467*, 2016.

A. Ahlander, M. Taveniku, et B. Svensson, "A multiple SIMD approach to radar signal processing", dans *TENCON'96. Proceedings., 1996 IEEE TENCON. Digital Signal Processing Applications*, vol. 2. IEEE, 1996, pp. 852–857. DOI : 10.1109/tencon.1996.608457

G. S. Almasi, "Overview of parallel processing", *Parallel Computing*, vol. 2, no. 3, pp. 191–203, 1985. DOI : 10.1016/0167-8191(85)90001-8

G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", dans *Large Scale Computing Capabilities AFIPS Conference Proceedings*, 1967. DOI : 10.1145/1465482.1465560

D. P. Anderson, "Boinc : A system for public-resource computing and storage", dans *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on.* IEEE, 2004, pp. 4–10. DOI : 10.1109/grid.2004.14

M. Bailey, "OpenGL compute shaders", 2016.

F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, et Y. Bengio, "Theano : New features and speed improvements", *arXiv preprint arXiv :1211.5590*, 2012.

A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, et V. S. Pande, "Folding@ home : Lessons from eight years of volunteer distributed computing", dans *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 1–8. DOI : 10.1109/ipdps.2009.5160922

J. Beyer et J. Larkin, "Targeting GPUs with OpenMP 4.5 device directives", 2016.

A. N. Burton et P. H. Kelly, "Workload characterization using lightweight system call tracing and reexecution", dans *Performance, Computing and Communications, 1998. IPCCC'98., IEEE International.* IEEE, 1998, pp. 260–266. DOI : 10.1109/pccc.1998.659975

S. Chen, Y. Deng, P. Attie, et W. Sun, "Optimal deadlock detection in distributed systems

based on locally constructed wait-for graphs", dans *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on.* IEEE, 1996, pp. 613–619. DOI : 10.1109/icdcs.1996.508012

I.-H. Chung, R. E. Walkup, H.-F. Wen, et H. Yu, "MPI performance analysis tools on Blue Gene/L", dans *SC 2006 Conference, Proceedings of the ACM/IEEE.* IEEE, 2006, pp. 16–16. DOI : 10.1109/sc.2006.43

E. G. Coffman, M. Elphick, et A. Shoshani, "System deadlocks", *ACM Computing Surveys (CSUR)*, vol. 3, no. 2, pp. 67–78, 1971. DOI : 10.1145/356586.356588

D. Couturier et M. R. Dagenais, "LTTng CLUST : A system-wide unified CPU and GPU tracing tool for OpenCL applications", *Advances in Software Engineering*, vol. 2015, p. 2, 2015. DOI : 10.1155/2015/940628

L. Dagum et R. Menon, "OpenMP : An industry standard api for shared-memory programming", *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. DOI : 10.1109/99.660313

M. Desnoyers et M. R. Dagenais, "The LTTng tracer : A low impact performance and behavior monitor for GNU/Linux", dans *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, Linux Symposium, 2006, pp. 209–224.

——, "Lockless multi-core high-throughput buffering scheme for kernel tracing", *ACM SIGOPS Operating Systems Review*, vol. 46, no. 3, pp. 65–81, 2012. DOI : 10.1145/2421648.2421659

F. C. Eigler et R. Hat, "Problem solving with SystemTap", dans *Proceedings of the Ottawa Linux Symposium*, vol. 2006, 2006.

P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, "Combined tracing of the kernel and applications with LTTng", dans *Proceedings of the 2009 Linux Symposium*, 2009, pp. 87–93.

M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, et V. Volkov, "Parallel computing experiences with CUDA", *IEEE micro*, vol. 28, no. 4, 2008. DOI : 10.1109/mm.2008.57

D. Goddeke, S. H. Buijssen, H. Wobker, et S. Turek, "GPU acceleration of an unmodified parallel finite element Navier-Stokes solver", dans *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on.* IEEE, 2009, pp. 12–21. DOI : 10.1109/hpcsim.2009.5191718

B. Gregg, “strace wow much syscall”, Mai 2014. En ligne : http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html

B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, et R. C. Walker, “Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. Generalized born”, *Journal of Chemical Theory and Computation*, vol. 8, no. 5, pp. 1542–1555, 2012. DOI : 10.1021/ct200909j

J. L. Hennessy et D. A. Patterson, *Computer Architecture : A Quantitative Approach*. Elsevier, 2011.

M. D. Hill et M. R. Marty, “Amdahl’s law in the multicore era”, *Computer*, vol. 41, no. 7, 2008. DOI : 10.1109/hpca.2008.4658638

M. Jabbarifar et M. Dagenais, “LIANA : Live incremental time synchronization of traces for distributed systems analysis”, *Journal of Network and Computer Applications*, vol. 45, pp. 203–214, 2014. DOI : 10.1016/j.jnca.2014.07.021

D. Kirk *et al.*, “NVIDIA CUDA software and GPU parallel computing architecture”, dans *ISMM*, vol. 7, 2007, pp. 103–104. DOI : 10.1145/1296907.1296909

D. Komatitsch, G. Erlebacher, D. Göddeke, et D. Michéa, “High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster”, *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010. DOI : 10.1016/j.jcp.2010.06.024

K. Kouame, N. Ezzati-Jivan, et M. R. Dagenais, “A flexible data-driven approach for execution trace filtering”, dans *Big Data (BigData Congress), 2015 IEEE International Congress on*. IEEE, 2015, pp. 698–703. DOI : 10.1109/bigdatacongress.2015.112

A. D. Kshemkalyani et M. Singhal, *Distributed Computing : Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

J. Lee, M. Samadi, Y. Park, et S. Mahlke, “Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems”, dans *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE, 2013, pp. 245–255. DOI : doi:10.1109/pact.2013.6618821

S. S. Lumetta et D. E. Culler, “Managing concurrent access for shared memory active messages”, dans *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*. IEEE, 1998, pp. 272–278. DOI : 10.1109/ipps.1998.669925

M. Mantor, “AMD Radeon™ HD 7970 with Graphics Core Next (GCN) architecture”, dans *Hot Chips 24 Symposium (HCS), 2012 IEEE*. IEEE, 2012, pp. 1–35. DOI : 10.1109/hotchips.2012.7476485

V. Marangozova-Martin et G. Pagano, “Gestion de traces d’exécution pour le systèmes embarqués : contenu et stockage”, 2013.

C. McClanahan, “History and evolution of GPU architecture”, *A Survey Paper*, p. 9, 2010.

C. E. McDowell et D. P. Helmbold, “Debugging concurrent programs”, *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 593–622, 1989. DOI : 10.1145/76894.76897

J. M. Mellor-Crummey et M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors”, *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991. DOI : 10.1145/103727.103729

A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, et M. R. Dagenais, “State history tree : An incremental disk-based data structure for very large interval data”, dans *Social Computing (SocialCom), 2013 International Conference on*. IEEE, 2013, pp. 716–724. DOI : 10.1109/socialcom.2013.107

S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, et D. Kaeli, “A comprehensive performance analysis of HSA and OpenCL 2.0”, dans *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 183–193. DOI : 10.1109/ispass.2016.7482093

M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, et W. E. Nagel, “Developing scalable applications with Vampir, VampirServer and VampirTrace”, dans *PARCO*, vol. 15, 2007, pp. 637–644.

R. H. Netzer et B. P. Miller, “What are race conditions ? : Some issues and formalizations”, *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992. DOI : 10.1145/130616.130623

T. Ni, “Direct Compute : Bring GPU computing to the mainstream”, dans *GPU Technology Conference*, 2009, p. 23.

B. Nichols, D. Buttlar, et J. Farrell, *Pthreads Programming : A POSIX Standard for Better Multiprocessing*. O'Reilly Media, Inc., 1996.

J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, et J. C. Phillips, "GPU computing", *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. DOI : 10.1109/jproc.2008.917757

B. Poirier, R. Roy, et M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events", *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, 2010. DOI : 10.1145/1842733.1842747

J. Reinders, "VTune performance analyzer essentials", *Intel Press*, 2005.

P. Rogers, "HSA overview", dans *Heterogeneous System Architecture*, 1er éd. Morgan Kaufmann, Nov. 2015, p. 11.

S. Rostedt, "Finding origins of latencies using Ftrace", dans *11th Real-Time Linux Workshop*, 2009, pp. 28–30.

B. Sander, "ROCm with harmony : Combining OpenCL, HCC, and HSA in a single program", Juin 2016. En ligne : http://gpuopen.com/rocm-with-harmony-combining-opencl-hcc-hsa-in-a-single-program/

B. Sander, G. Stoner, S.-c. Chan, W. Chung, et R. Maffeo, "HCC : A C++ compiler for heterogeneous computing", *HSA Foundation, Tech. Rep.*, 2015.

S. S. Shende et A. D. Malony, "The TAU parallel performance system", *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006. DOI : 10.1177/1094342006064482

R. Smith, "AMD's Graphics Core Next preview : AMD's new GPU, architected for compute", Déc. 2011. En ligne : https://www.anandtech.com/show/4455/amds-graphics-core-next-preview-amd-architects-for-compute

R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with GDB", *Free Software Foundation*, vol. 675, 1988.

J. E. Stone, D. Gohara, et G. Shi, "Opencl : A parallel programming standard for heterogeneous computing systems", *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010. DOI : 10.1109/mcse.2010.69

G. Stoner, “ROCm : Platform for a new era of heterogeneous in HPC and ultrascale computing”, Jan. 2016. En ligne : http://gpuopen.com/radeon-open-compute-new-era-heterogeneous-in-hpc-ultrascale-computing-the-boltzmann-initiative-delivering-new-opportunities-in-gpu-computing-research/

J. Subhlok et G. Vondran, “Optimal use of mixed task and data parallelism for pipelined computations”, *Journal of Parallel and Distributed Computing*, vol. 60, no. 3, pp. 297–319, 2000. DOI : 10.1006/jpdc.1999.1596

M. A. Suleman, O. Mutlu, M. K. Qureshi, et Y. N. Patt, “Accelerating critical section execution with asymmetric multi-core architectures”, dans *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1. ACM, 2009, pp. 253–264. DOI : 10.1145/1508284.1508274

K.-C. T. K.-C. Tai, “Definitions and detection of deadlock, livelock, and starvation in concurrent programs”, dans *Parallel Processing, 1994. ICPP 1994 Volume 2. International Conference on*, vol. 2. IEEE, 1994, pp. 69–72. DOI : 10.1109/icpp.1994.84

J. Trumper, J. Dollner, et A. Telea, “Multiscale visual comparison of execution traces”, dans *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 53–62. DOI : 10.1109/icpc.2013.6613833

K. H. Tsoi et W. Luk, “Axel : A heterogeneous cluster with FPGAs and GPUs”, dans *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2010, pp. 115–124. DOI : 10.1145/1723112.1723134

T. Willhalm et N. Popovici, “Putting Intel® Threading Building Blocks to work”, dans *Proceedings of the 1st International Workshop on Multicore Software Engineering*. ACM, 2008, pp. 3–4. DOI : 10.1145/1370082.1370085

Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, “Google’s neural machine translation system : Bridging the gap between human and machine translation”, *arXiv preprint arXiv :1609.08144*, 2016.

E. Wynters, “Fast and easy parallel processing on GPUs using C++ AMP”, *Journal of Computing Sciences in Colleges*, vol. 31, no. 6, pp. 27–33, 2016.

ANNEXE A FONCTIONNEMENT DE LA SOLUTION PROPOSÉE

La solution proposée est disponible à l'adresse suivante : `https://github.com/pmargheritta/LTTng-HSA`. Nous précisons dans cette annexe son utilisation.

Traçage

La solution proposée, une fois compilée à l'aide du *makefile* fourni, se présente sous la forme de quatre bibliothèques logicielles partagées (extension `.so`) dans le sous-dossier `lib/`. Chacune, en étant préchargée lors de l'exécution d'un programme, permet d'insérer dans ce dernier une instrumentation donnant accès au traçage d'un ensemble d'événements formant ce que nous appelons une « cible de traçage ».

Le préchargement d'une telle bibliothèque exploite le mécanisme `LD_PRELOAD` de la sorte :

```
LD_PRELOAD=<chemin/vers/LTTng-HSA>/lib/hsa_call_stack.so ./mon-programme
```

Dans ce cas, les points de trace correspondant à la cible de traçage `call_stack` sont insérés dans l'exécutable `mon-programme`. En traçant l'exécution à l'aide de LTTng, on obtient une trace donnant des informations sur les appels de fonctions de l'interface de programmation de HSA. En plus du préchargement, certaines cibles de traçage requièrent des valeurs spécifiques de variables d'environnement :

- la cible de traçage `queue_profiling` requiert `HSA_SERVICE_GET_KERNEL_TIMES=0` ;
- la cible de traçage `perf_counters` requiert `HSA_EMULATE_AQL=1`.

Pour simplifier la tâche du traçage, nous utilisons un script de ce type :

```
lttng create
lttng enable-event -u hsa_runtime:*
lttng start
LD_PRELOAD=<chemin/vers/LTTng-HSA>/lib/hsa_call_stack.so ./mon-programme
lttng stop
lttng view
lttng destroy
```

En complexifiant légèrement ce script par l'utilisation de paramètres, il est facile de générer des traces pour différentes cibles et différents programmes.

Traitement des traces

Trois scripts sont fournis avec notre solution afin de manipuler les traces obtenues. Ils se trouvent dans le sous-dossier `scripts/`.

— Le script `merge.py` prend comme arguments deux traces d'entrée et une trace de sortie. Les événements des deux traces d'entrée sont fusionnés dans la trace de sortie.
— Le script `sort.py` prend comme arguments une trace d'entrée et une trace de sortie. Il repère les événements asynchrones à trier, calcule leur nouvelle estampille de temps et les replace dans l'ordre chronologique avec les événements synchrones dans la trace de sortie.
— Le script `combine.sh` utilise les deux scripts précédents pour réaliser la fusion et le tri de deux traces d'entrée dans une trace de sortie.

Grâce à ces différents scripts, nous pouvons donc aisément soit combiner directement deux traces, soit nous intéresser aux résultats des algorithmes de fusion et de tri utilisés individuellement.

Visualisation des traces

La visualisation de traces combinées s'effectue avec le logiciel Trace Compass, en utilisant les analyses XML proposées dans le sous-dossier `analyses/` de notre solution. Une trace combinée grâce aux scripts présentés précédemment doit premièrement être importée dans le logiciel. Il suffit ensuite d'ajouter l'analyse XML souhaitée à celles utilisées dans Trace Compass, puis d'ouvrir la vue correspondante pour obtenir la visualisation attendue, comme montré en figure A.1.

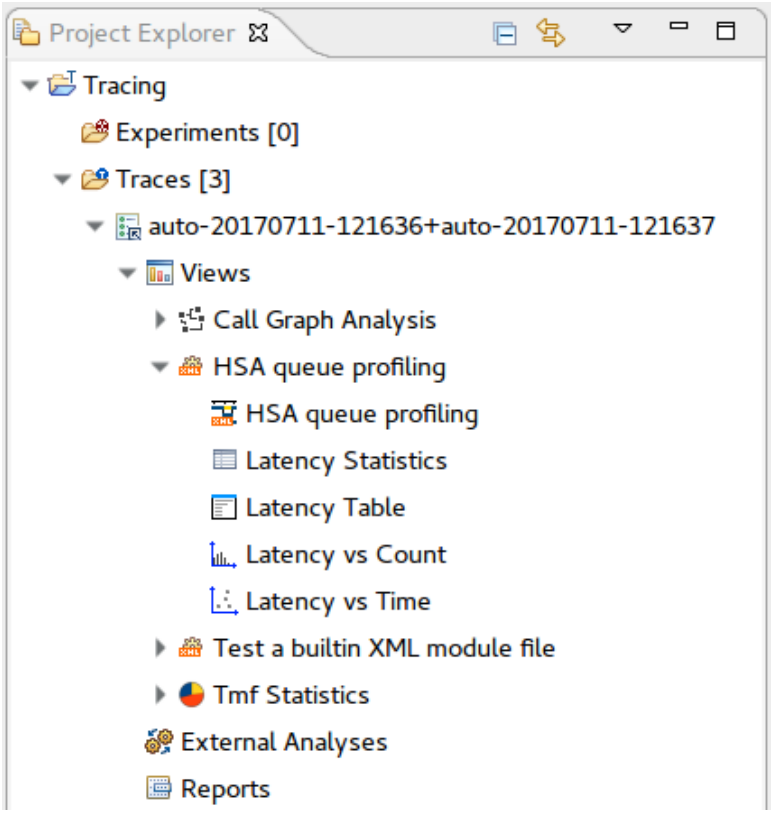

Figure A.1 L'explorateur de projets de Trace Compass montrant la vue associée à l'analyse XML *HSA queue profiling* pour une trace.