Aoues GUESMI
Aymen FADHEL

ASR9 - Projet de Fin d'Études

Trace d'exécution pour programmes CUDA

Encadré par: Mr. François TRAHAY

Octobre 2012 - Janvier 2013

Remerciements

C'est avec un grand plaisir que nous réservons ces quelques lignes en signe de gratitude et de reconnaissance envers tous ceux qui ont contribué à la réalisation de ce travail. Nous tenons à remercier et à exprimer notre gratitude à notre encadrant Mr. François TRAHAY qui nous a prodigués de l'aide tout le long de ce projet.
Enfin, nous exprimons de plus nos remerciements aux professeurs de la VAP d'Architecte de Services informatique en Réseaux pour l'intérêt qu'ils ont porté à notre travail.

Résumé

Dans un monde d'informations où le calcul haute performance devient de plus en plus omniprésent et essentiel pour le bon fonctionnement de plusieurs services, le besoin d'un outil de traçage de programmes HPC devient encore plus important.
Relativement nouveau, « EZTrace »essaye de combler ce vide et offre des outils à la fois puissants et simples qui permettent d'intercepter une multitude de librairies comme MPI, pthread, stdio et OpenMP ... etc. Néanmoins, malgré la forte utilisation de CUDA dans les technologies de pointe, et de plus en plus dans les applications à caracètre HPC, « EZTrace »ne présente pas encore un module de traçage des fonctions CUDA. Ce projet vise donc à étudier le fonctionnement d'EZTrace afin d'implémenter et d'évaluer un module pour le traçage des fonctions de la bibliothèque CUDA.

Mots-clés : EZTrace, CUDA, module, performance

Table des matières

Table des figures

Introduction

La puissance des ordinateurs ne cesse d'augmenter jour après jour, et la loi de Moore datant de 1965 reste encore et toujours valide (en adaptant subtilement son énoncé). Le fait que nos ordinateurs personnels possèdent la puissance des supercalculateurs datant de quelques années est assez époustouflant. Cet avancement matériel ne cesse de pousser les programmes informatiques vers de nouveaux horizons et de nouvelles normes. En effet, certaines applications qui gèrent un assez grand nombre de données et de contraintes comme la météorologie et les simulations nucléaires atteignent rapidement les limites des machines existantes. C'est dans ce genre d'applications que s'institue l'informatique du calcul intensif ou HPC (pour High Performance Computing). Cette branche de l'informatique s'intéresse aux outils d'accélération de programmes comme la programmation multi-threads et multi-cœur (OpenMP), la programmation multi-processus et multi-machines (MPI) ainsi que la programmation sur carte graphique (CUDA/OpenCL). Par contre, l'utilisation de ce genre d'outils apporte son lot de problèmes et notamment la difficulté de l'analyse de l'exécution des applications utilisant un ou plusieurs de ces outils. « EZ-Trace »est un logiciel libre permettant de générer la trace d'exécution des programmes utilisant des accélérateurs de performances. Plusieurs bibliothèques sont supportées, dont MPI, pthread et OpenMP, mais le manque de support pour les programmes utilisant le GPU constitue un manque qu'on ne peut pas s'empêcher de constater.

Ce projet de fin d'études, intitulé **Trace d'exécution pour programmes CUDA**, s'inscrit dans le cadre du module **CSC5005** de la voie d'approfondissement **ASR**. C'est un projet qui s'est étalé tout au long du semestre 9 de notre cursus sous l'encadrement de Monsieur **François TRAHAY**, enseignant-chercheur à **Télécom SudParis**.

Dans ce projet nous avons essayé de créer et d'évaluer un module CUDA pour « EZ-Trace »capable de tracer les programmes s'exécutant sur la carte graphique en utilisant l'API CUDA de Nvidia.

Chapitre 1

Présentation d'EZTrace

EZTrace (« EZ »comme « Easy ») est un outil libre développé grâce à une collaboration entre Télécom SudParis, Inria Bordeaux et les laboratoires de l'université de Tennessee. Ce projet date depuis 2010, et nous comptons actuellement 8 version stables. La première étant EZTrace0.1, et la dernière étant EZTrace0.8 sortie le 30-10-2012. Ce projet comporte 17 membres.
Le but de ce projet étant la génération de traces d'exécution des programmes de calcul parallèle, ou intensif, ceci afin de vérifier leurs comportements ainsi que leurs performances. « EZTrace »permet entre autres d'obtenir des statistiques sur les différentes ressources utilisées dans les programmes analysés. Les traces obtenues peuvent être toutefois visualisées sur des outils de visualisation tels que VampirTrace, ou ViTE (Visual Trace Explorer, également un autre outil développé par Télécom SudParis et Inria Bordeaux). « EZTrace »fonctionne avec des programmes écrits en langage C ou en Fortran et il peut tracer :

- Les fonctions de la librairie **stdio.h** (fonctions d'entrée/sortie).
- Les fonctions de la librairie **OpenMP** (parallélisation des programmes sur les cœurs d'une même machine).
- Les fonctions de la librairie **MPI** (parallélisation des programmes sur plusieurs machines).
- Les fonctions de la librairie **pthread.h** (gestion de multi-threading, exclusion mutuelle, architecture centralisée ...).
- Les fonctions mémoire (allocation/désallocation dynamique)

Ce sont actuellement les modules disponibles dans la dernière version stable d'EZTrace, avec un dernier module qui est **papi** (module responsable de la gestion des compteurs de performance).

1.1 État de l'art

Nous avons eu l'occasion de découvrir un autre outil de génération de traces, durant le module CSC5001, qui est VampirTrace. Cependant, ces deux outils présentent des grandes différences parmi lesquelles on cite :

- EZTrace s'utilise pendant l'exécution, alors que VampirTrace s'utilise pendant la phase de compilation du programme. Le fait d'utiliser un outil pendant l'exécution paraît plus facile et plus confortable à l'utilisateur, au lieu de l'utiliser avec la commande de compilation (besoin de changer le Makefile à chaque fois, besoin d'avoir le code source ...)
- EZTrace permet de choisir les librairies à intercepter, via un chargement de module(s), chose que VampirTrace ne permet pas.
- EZTrace permet aussi de sélectionner la partie du code source à intercepter, alors que VampirTrace trace tout le code par défaut.
- EZTrace supporte deux formats de traces, à savoir *.paje* et *.otf*, alors que VampirTrace ne supporte que la dernière.

1.2 Traçage avec EZTrace

Le traçage avec EZTrace est assez simple : EZTrace redéfinit les fonctions à intercepter. Dans les nouvelles fonctions déclarées, « EZTrace »lance 2 évènements : La premier pour signaler l'entrée à la fonction, le second pour la sortie, et entre les deux, on appelle la fonction dans sa librairie. À la fin, l'interception de la fonction dans sa librairie se fait grâce à la macro $INTERCEPT$. Il est à noter que $INTERCEPT$ intervient au moment de l'établissement des liens logiques avec les librairies. Supposons par exemple que nous voulons intercepter une fonction $f(params) : void$:

```
void f(params) {
/* enregistre un évènement d'entrée à la fonction f */
EZTRACE_EVENT(MACRO_FONCTION_ENTREE, liste_des_arguments);
/* appel à la fonction libF() qui appellera elle-même la fonction f()
dans la librairie où elle est définie */
libF(params);
/* enregistre un évènement de sortie de la fonction f */
EZTRACE_EVENT(MACRO_FONCTION_SORTIE, liste_des_arguments);
}
```

```
/* interception de libF avec la macro INTERCEPT */
INTERCEPT(libF);
```

Ceci s'explique davantage par le schéma de la **FIGURE 1.1**.

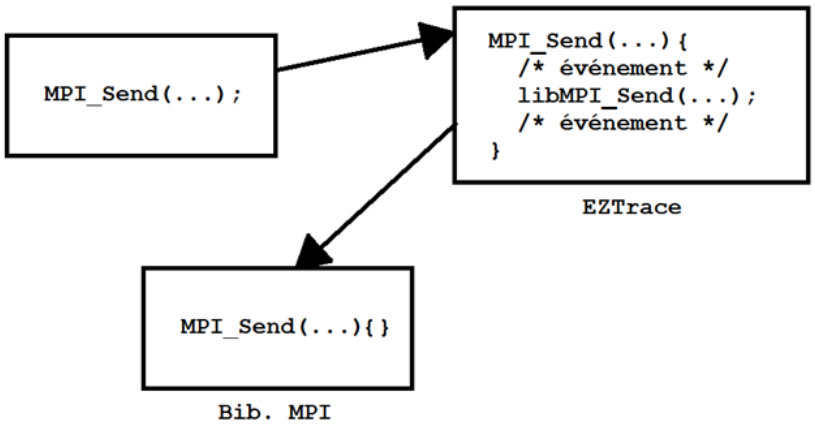

FIGURE 1.1 – Interception de MPI_Send() avec EZTrace

Les macros $MACRO\_FONCTION\_ENTREE$ et $MACRO\_FONCTION\_SORTIE$ sont enregistrées comme des codes permettant de faire la correspondance entre les évènements enregistrés, et la trace brute générée par la suite.

De plus, Notons que ces évènements là sont capturés par des « handlers »qui se chargent d'enregistrer la trace. Chaque type d'évènement possède son propre handler (handler d'entrée, handler de sortie ...).

Ces handlers vont céder le contrôle à la bibliothèque GTG - l'interface de génération de traces - qui va se charger de faire des *popState*() et des *pushState*() suivant le type d'évènement : S'il s'agit d'un évènement d'entrée, l'état d'entrée à la fonction en question est ajouté dans la trace, et s'il s'agit d'un évènement de sortie, on ajoute l'était de sortie de la fonction dans la trace. Entre autres, GTG permet aussi la gestion des conteneurs, c'est-à-dire, il permet de créer une hiérarchie de ressources sur lesquels les programmes s'exécutent, par exemple, un conteneur pour chaque processus, qu'il va contenir à son tour des sous-conteneurs pour les différents threads CPU ou GPU lancés dans ce processus.

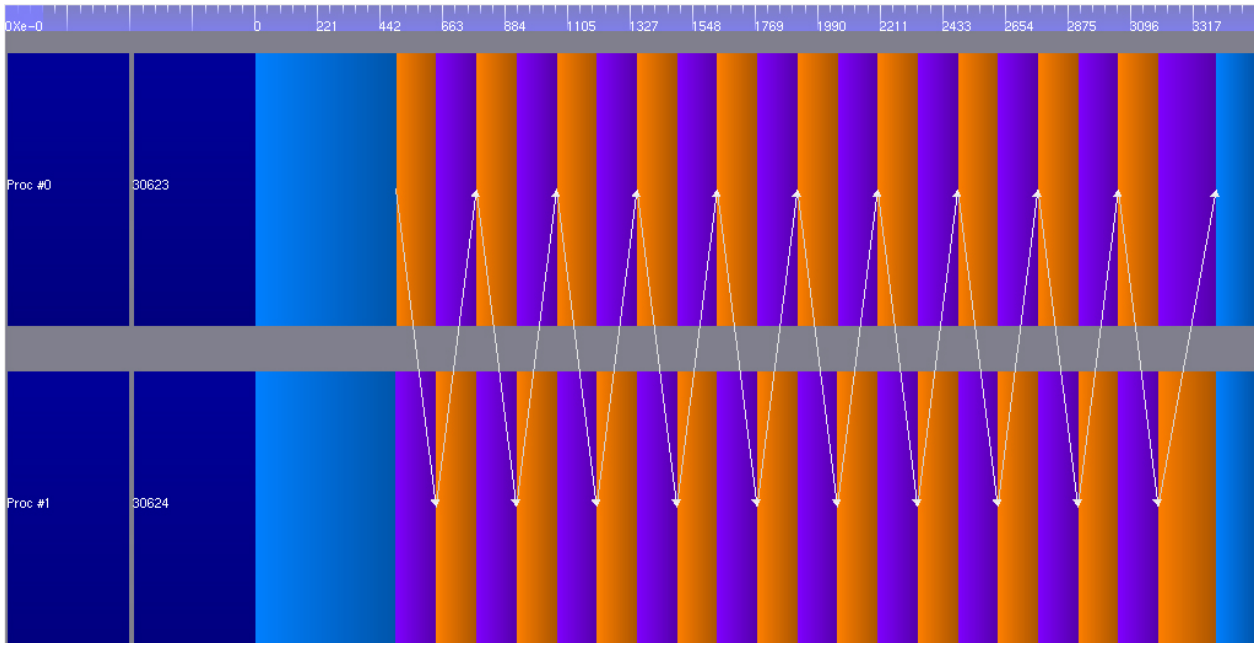

FIGURE 1.2 – Exemple d'une trace générée par EZTrace, et visualisée sur ViTE

Chapitre 2

Déroulement du projet

Dans ce chapitre, nous allons présenter la démarche que nous avons suivie pour la réalisation de ce projet. Ceci comporte la conception globale du module et le choix de la méthode d'interception, la mise en œuvre des différents composants du module ainsi que les actions prises face aux différents défis et obstacles rencontrés tout au long du projet.

2.1 Ajout d'un module

2.1.1 Structure d'EZTrace

Comme expliqué dans le chapitre précédent, « EZTrace »se base sur une architecture modulaire. En effet, chaque librairie possède son propre module et tous les modules sont indépendants. Nous trouvons par exemple un module MPI (servant à tracer les fonctions de la bibliothèque *mpi.h* tels que $MPI\_Send()$, $MPI\_Receive()$ ...), un module OpenMP, un module stdio (afin de tracer les fonctions d'entrée/sortie standards du C) et pleins d'autres. La **FIGURE 2.2** présente la structure d'EZTrace, on y trouve :

- **Le dossier doc :** Qui contient la documentation d'EZTrace.
- **Le dossier example :** Dans lequel se trouvent des programmes simples permettant de tester rapidement les différentes fonctionnalités d'EZTrace.
- **Le dossier extlib :** Ce dossier contient les différentes librairies tierces nécessaires au bon fonctionnement d'EZTrace, notamment GTG et FXT qui permettent la génération et la conversion de la trace.
- **Le dossier src :** Qui contient tout le code source d'EZTrace dont ceux des modules standards. La **FIGURE 2.1** détaillera la hiérarchie de ce dossier.
- **Le dossier test :** Qui contiendra à son tour des codes de test propres à chaque module d'EZTrace.

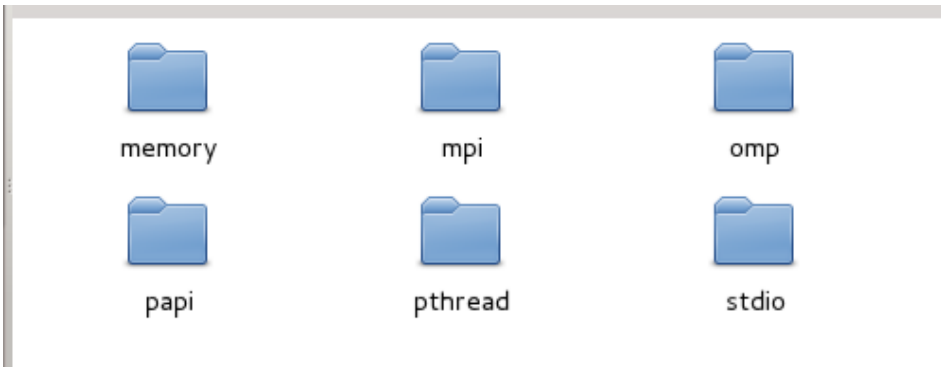

FIGURE 2.1 – Modules installés dans EZTrace

- **bootstrap.sh :** Un exécutable qui permet de compiler l'outil EZTrace et le préparer à l'installation des modules.
- **configure.ac :** Un script de configuration qui permet de configurer EZTrace à l'aide de l'outil « autoconf ».
- **Makefile.am :** Un Makefile pour la compilation des modules et leur intégration dans EZTrace, en utilisant l'outil « automake ».

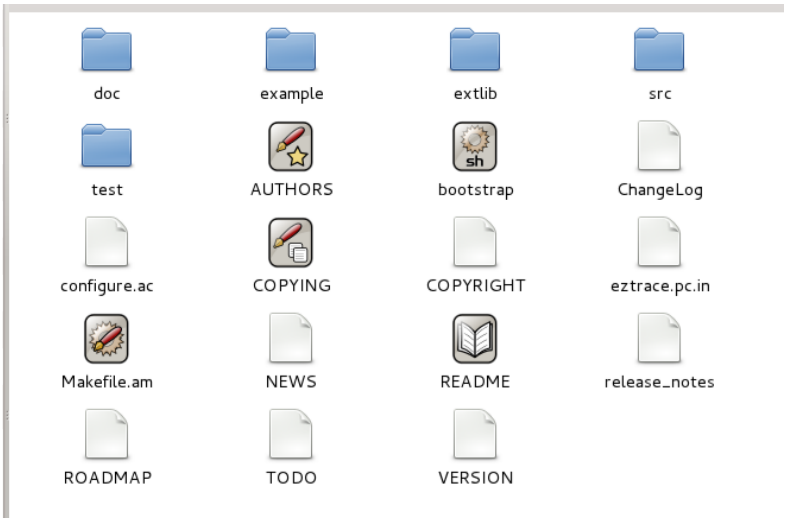

FIGURE 2.2 – Structure d'EZTrace

2.1.2 Structure d'un module

La première étape vers la création du nouveau module consistait à ajouter un dossier **cuda** sous *src/modules* et de le peupler des fichiers suivants : (voir **FIGURE 2.3**)

- *cuda.cu* : Traçage des différentes fonctions CUDA.

- *eztrace_convert_cuda.c* : Conversion de la trace brute vers une trace qui sera visualisable à l'aide d'un outil de visualisation approprié.
- *cuda_ev_codes.h* : Définition de codes numériques pour la gestion des différents évènements avec FXT.
- *Makefile.am* : Un Makefile portable permettant de générer automatiquement un Makefile approprié à l'environnement de compilation.

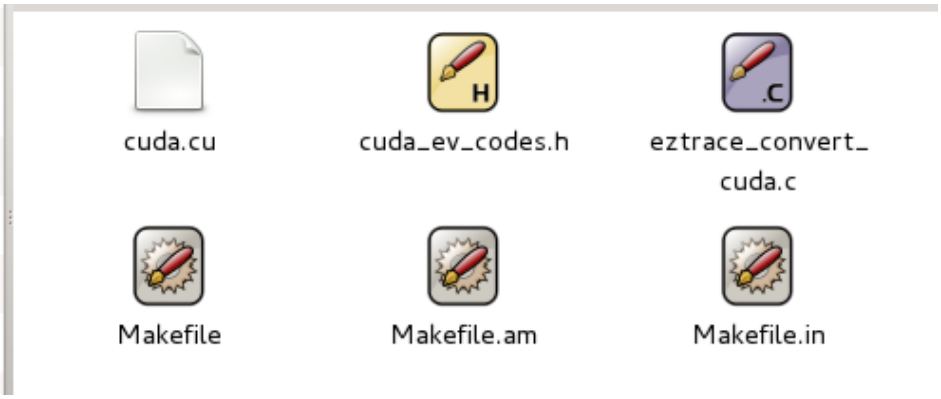

FIGURE 2.3 – Structure du module CUDA d'EZTrace

2.1.3 Outils

La première étape de la construction du module étant achevée, passons maintenant à l'intégration du module avec le reste d'EZTrace. Comme nous venons de le mentionner, la compilation d'EZTrace se fait grâce à l'utilisation de deux outils indispensables :

1. **Automake :** Outil libre, écrit en Perl, et développé par le projet GNU, permettant, à partir d'un fichier *Makefile.am*, la génération de fichiers *Makefile.in* permettant une compilation portable du programme en automatisant la génération du Makefile suivant la configuration matérielle et logicielle de la machine hôte. Automake vise à permettre au programmeur d'écrire un Makefile dans un langage de haut niveau, plutôt que d'avoir à écrire tout le Makefile manuellement. Dans les cas simples, il suffit de donner :
   - Une ligne qui déclare le nom du programme de construction.
   - La liste des fichiers sources.
   - La liste des options à passer au compilateur, et éventuellement d'autres options à passer à l'éditeur de liens.

À partir de ces informations, Automake génère un fichier Makefile qui permet à l'utilisateur de :

- Compiler le programme proprement (c'est à dire, supprimer les fichiers résultant de la compilation).
- Installer le programme dans les répertoires standards.
- Désinstaller le programme à partir duquel il a été installé.
- Créer une archive de la distribution source.
- Vérifier que cette archive est auto-suffisante, et en particulier que le programme peut être compilé dans un répertoire autre que celui où les sources sont déployées. [1]

2. **Autoconf :** Pour que Automake puisse fonctionner, il doit être utilisé avec Autoconf, un autre outil libre de production de scripts de configuration pour la compilation, d'installation, et de paquetage des logiciels sur des systèmes informatiques. Autoconf permet entre autres de générer des scripts shell de configuration d'un code source afin de le faire fonctionner sous différents environnements UNIX, et ceci à partir d'un script de configuration *configure.ac*. Ce script configure un logiciel à s'installer sur un système cible particulier. Après avoir exécuté une série de tests sur le système cible, le script génère des fichiers d'en-tête et un Makefile, personnalisant ainsi le logiciel pour le système cible. [2]
   Le format du fichier *configure.ac* est le suivant :

   ****

   AC_PREREQ (version) : Cette macro peut être utilisée pour s'assurer qu'une version assez récente du programme Autoconf est disponible pour traiter le fichier *configure.ac*.

   AC_INIT(package, version, bug-report-address) : Cette macro est requise dans chaque fichier *configure.ac*. Elle indique le nom et la version du logiciel pour lequel on souhaite la générer un script de configuration.

   Des informations sur le paquetage
   Des contrôles pour les programmes
   Des contrôles pour les bibliothèques
   Des contrôles pour les fichiers d'en-tête
   Des contrôles pour les types
   Des contrôles pour les structures
   Des contrôles pour les caractéristiques du compilateur

Des contrôles pour les fonctions des bibliothèques
Des contrôles pour les services système

AC_CONFIG_FILES ([file ...])

AC_OUTPUT
****

La **FIGURE 2.4** montre le processus complet du fonctionnement de Automake et de Autoconf.

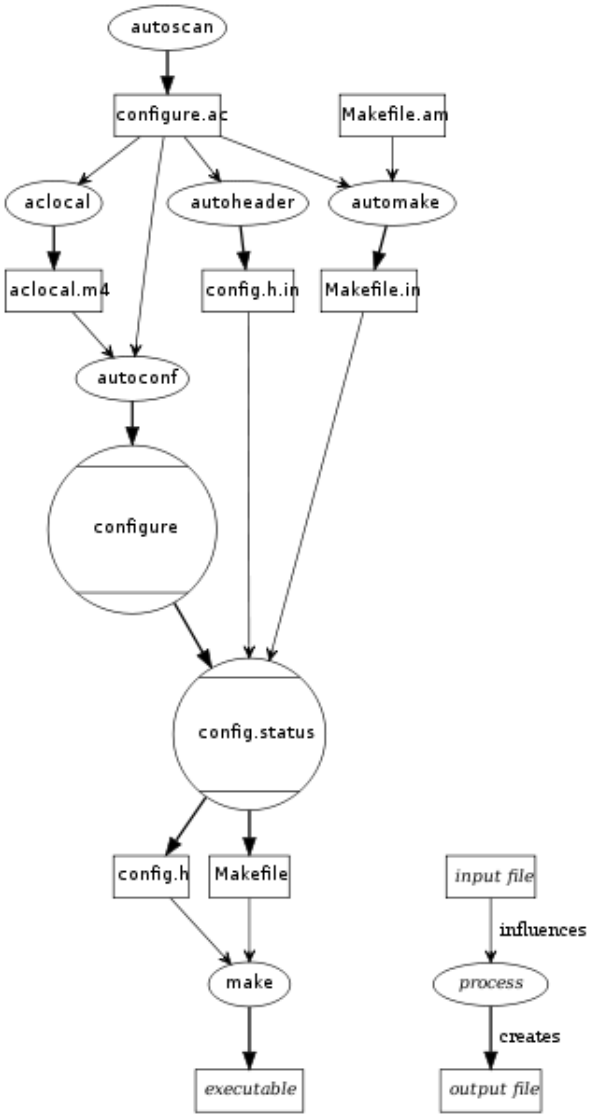

FIGURE 2.4 – Processus de fonctionnement d'autoconf et d'automake

2.2 Installation d'un module dans EZTrace

L'étape suivante consiste à configurer ce nouveau module, à l'aide de auto-tools, afin de l'intégrer à « EZTrace ». Pour commencer, nous devons exécuter la commande :

$ ./bootstrap

Cette commande se charge d'initier « EZTrace »pour l'installation de nouveaux modules. Par la suite, il faudra compiler le module précédemment créé avec l'outil « EZTrace », ceci se fait par le biais des commandes suivantes :

$ ./configure−−prefix=<EMPLACEMENT_POUR_INSTALLER_EZTRACE>

On pourra éventuellement passer plusieurs options à cette commande, pour spécifier les différentes librairies externes à inclure pour la configuration, par exemple :

−−with-fxt=$FXT_ROOT
−−with-gtg=$GTG_ROOT
−−with-mpi=$MPI_ROOT

Une fois « EZTrace »est configuré, la compilation des différents modules est requise, il suffit d'exécuter :

$ make && make install

Pour vérifier que les opérations ci-dessous ont été exécutées avec succès, il suffit d'exécuter :

$ eztrace_avail

Affichage en cas de succès : Module CUDA ajouté

Nous avons maintenant à notre disposition un module CUDA prêt à fonctionner avec « EZTrace ».

2.3 Interception des fonctions CUDA

Maintenant que nous avons les clés en main pour pouvoir commencer le codage du module, nous devons passer tout d'abord par le choix concernant les fonctions à intercepter, et sélectionner par la suite lesquelles seront les plus intéressantes à intercepter.

2.3.1 Choix des fonction CUDA à intercepter

Choisir les fonctions CUDA à intercepter revient à déterminer lesquelles sont les plus susceptibles à être présentes dans un programme CUDA. Une étude de quelques programmes

CUDA ainsi que de la documentation officielle des différentes fonctions nous a permis de conclure que les fonctions ci-dessous sont essentielles dans tout programme CUDA :

- *cudaMalloc* : Cette fonction permet d'allouer de la mémoire sur le GPU. Cette mémoire permettra par la suite le stockage d'informations des calculs effectués sur le GPU. Il sera intéressant d'intercepter cette fonction afin de connaître à tout moment la quantité de mémoire transférée vers le GPU.
- *cudaMemcpy* : Cette fonction permet de copier des octets de la mémoire CPU vers la mémoire GPU, et vice versa.
- *cudaLaunchKernel* : Cette fonction est indispensable pour tout programme CUDA, en fait elle permet de lancer un noyau GPU dans lequel les calculs seront effectués. Cette fonction pourra être appelée éventuellement plusieurs fois quand nous avons besoin à lancer plusieurs noyaux GPU.
- *cudaThreadSynchronize* : Le but de cette fonction consiste à synchroniser tous les threads du GPU. Vu que le lancement des noyaux GPU est asynchrone, il faut par la suite synchroniser les threads afin d'obtenir des données cohérentes depuis le GPU. Il est à noter que cette fonction prend généralement trop de temps à s'exécuter, surtout dans le cas où nous avons un travail lent sur le GPU, ou bien plusieurs noyaux qui sont lancés.

2.3.2 Choix de l'API d'exécution de CUDA

La question qui se posait plus tard est de choisir le niveau des API CUDA le plus adéquat pour l'interception. Comme le montre la **FIGURE 2.5**, il faudra choisir entre l'**API Runtime** et l'**API Driver**.

En fait, pour chaque appel à une fonction de l'API Runtime, sa correspondante dans l'API Driver est appelée. Alors il s'avère inutile de tracer les fonctions de l'API Runtime (sauf dans certains cas spécifiques où l'équivalent driver est inexistant) puisqu'elles sont de plus haut niveau d'abstraction que le niveau Driver. Par exemple, l'appel à *cudaMalloc* engendre un appel implicite à *cuMemAlloc*, cet appel est expliqué davantage dans la **FIGURE 2.5** et dans le code suivant :

```
1 cudaError_t cudaMalloc (void **devPtr, size_t size) {
2         /* des instructions d'initialisation */
3         ...;
4         /* appel a cuMemAlloc qui alloue size octets sur le
5         dispositif GPU */
```

```
6           cuMemAlloc((CUdeviceptr) devPtr, size);
7           /* eventuellement d'autres instructions */
8           ...;
9   }
```

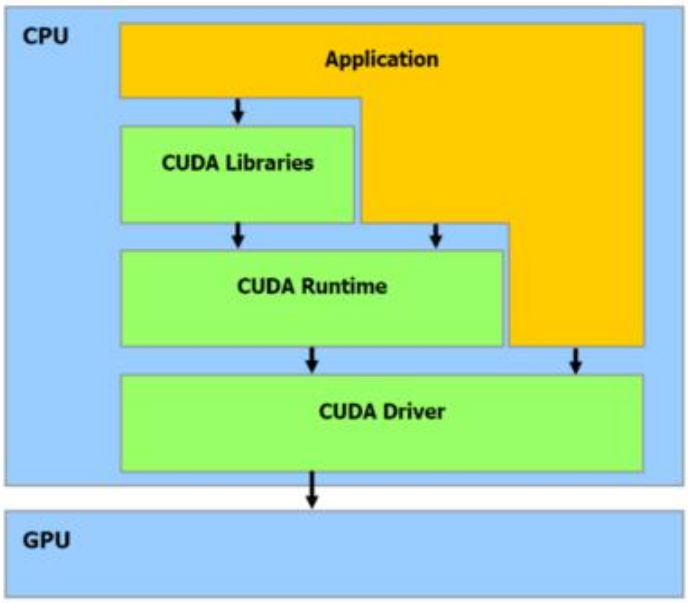

FIGURE 2.5 – Hiérarchie des APIs de CUDA

2.3.3 Interception

Dans un premier temps, nous avons essayé de suivre la démarche expliquée dans le chapitre précédent, à savoir tracer chaque fonction manuellement en enregistrant des évènements de début et de fin d'exécution de chaque fonction. Pour illustrer cette démarche, prenons l'exemple de la fonction $cuMemAlloc(CUdeviceptr, size\_t) : CUresult$:

```
CUresult cuMemAlloc(CUdeviceptr *dptr, size_t bytesize) {
/* enregistrement d'un évènement d'entrée à la fonction cuMemAlloc */
EZTRACE_EVENT(MEMALLOC_ENTREE, &dptr, bytesize);
/* appel à la fonction libCuMemAlloc qui appellera elle-même la fonction
   cuMemAlloc dans la librairie cuda.h */
libCuMemAlloc(&dptr, bytesize);
/* enregistrement d'un évènement de sortie de la fonction cuMemAlloc */
EZTRACE_EVENT(MEMALLOC_SORTIE, &dptr, bytesize);
```

Les macros $MEMALLOC\_ENTREE$ et $MEMALLOC\_SORTIE$ sont enregistrées dans *cuda_ev_codes.h*. Elles permettent de faire correspondre les évènements enregistrés

à la trace brute générée.

Plus tard, cette méthodologie (le fait d'utiliser des EZTRACE_EVENT) s'est avérée assez difficile, à cause de quelques problèmes de configuration. En effet, l'incorporation de *nvcc* (compilateur CUDA) avec les auto-tools et le processus de compilation global d'EZTrace n'était pas une tâche simple à achever, en particulier ceci est dû au fait que *nvcc* est un compilateur C++ alors que le reste d'EZTrace était écrit en C.

Étant donné que la stratégie précédente n'a pas abouti à un résultat, et que le temps commençait à se couler de nos mains, nous avons opté à une architecture différente, se basant sur la bibliothèque **CUPTI**.

2.3.4 Utilisation de CUPTI

CUPTI (**CU**DA **P**rofiling **T**ools **I**nterface) est une technologie de profilage fournie par NVIDIA (2010), et qui offre des outils d'analyse de performance et des informations détaillées sur la façon avec laquelle les applications utilisent le GPU dans un système. CUPTI fournit deux mécanismes simples et puissants qui permettent l'analyse de performance tels que le générateur de profils visuels NVIDIA (NVIDIA Visual Profiler), tout ceci afin de comprendre le fonctionnement interne d'une application sur le GPU. [3]

L'un des composants les plus utilisés de CUPTI est l'API de rappel, ou dite « callback API ». Elle permet d'injecter des points d'analyse dans l'entrée et la sortie de chaque fonction de l'API Runtime ou Driver de CUDA.

Un autre composant à garder en tête est l'API responsable de la mesure des métriques et de l'analyse de performance du code CUDA. Ce composant permet d'enregistrer des compteurs tels que le nombre d'instructions, les opérations de la mémoire cache ... etc.

La solution que nous disposons maintenant est d'intercepter les fonctions CUDA grâce à l'API callback de CUPTI. Pour cela, il faudra fournir à CUPTI une fonction callback qui sera appelée à chaque évènement, dans l'entrée et la sortie de chaque fonction CUDA. Ces appels nous ont permis de filtrer les évènements afin d'intercepter uniquement ceux qui nous intéressaient. La fonction callback se chargeait de tout le rôle d'interception en invoquant des évènements « EZTrace »au moment voulu. L'exemple ci-dessous montre l'interception de la fonction $cuMemAlloc$, grâce à l'appel de la fonction de callback de CUPTI :

```
void CUPTIAPI my_callback(void *userdata, CUpti_CallbackDomain domain,
CUpti_CallbackId cbid, const void *cbdata) {
const CUpti_CallbackData *cbInfo = (CUpti_CallbackData *)cbdata;
```

```
int entry = 0;

    if (cbInfo->callbackSite == CUPTI_API_ENTER) {
     /* the callback was called before the event */
       entry = 1;
    } else if (cbInfo->callbackSite == CUPTI_API_EXIT) {
       /* the callback was called after the event */
       entry = 0;
    }

    if (cbid == CUPTI_DRIVER_TRACE_CBID_cuMemAlloc_v2) {
cuMemAlloc_v2_params *funcParams = (cuMemAlloc_v2_params*)(cbInfo->functionParams);
   if(entry) {
    EZTRACE_EVENT1(FUT_CUDA_CUMEMALLOC_START, funcParams->bytesize);
     printf("Entrée à cuMemAlloc");
   } else {
     EZTRACE_EVENT1(FUT_CUDA_CUMEMALLOC_STOP, funcParams->bytesize);
     printf("Sortie de cuMemAlloc");
   }
}
}
```

[4]

Nous pouvons voir ici que la fonction de callback se charge de la capture des différents points d'entrée et de sortie des fonctions CUDA. Pour cela, un entier - qui joue le rôle d'un booléen ici - indiquera le déclenchement d'un évènement d'entrée ou de sortie. Ensuite, l'identification de la fonction à intercepter se fait via son *cbid*, qui est un identifiant CUPTI permettant d'identifier chacune des fonctions CUDA Driver et Runtime d'une manière unique. S'il s'agit d'un point d'entrée, un évènement d'entrée est envoyé à EZTrace, sinon, c'est le tour de l'évènement de sortie.

2.4 Interception du noyau sur le GPU

En adoptant la méthodologie précédente, nous ne pouvons pas tracer le travail effectué sur le noyau GPU. En effet, l'appel au noyau GPU se fait par le biais de la fonction $cuLaunchKernel$. En interceptant cette fonction, il est impossible de savoir la quantité de travail effective effectué sur le GPU, nous aurons dans ce cas un traçage suite à l'appel de $cuLaunchKernel$, qui se charge que du lancement du noyau. Ainsi, la trace générée va

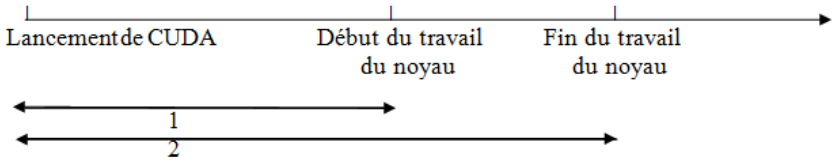

FIGURE 2.6 – Méthodologie du traçage du travail du GPU

inclure le lancement du noyau dans le CPU et pas sur le CPU.

Pour contourner cette problématique, il est indispensable de créer manuellement des évènements propres au fonctionnement du noyau dans le GPU (début et fin). Ceci grâce aux fonctions *cuEventCreate*(), *cuEventRecord*() et *cuEventSynchronize*(). Entre autres, il faudra créer un conteneur pour contenir les évènements s'exécutant sur le GPU, ainsi chaque processus CPU aura son propre thread GPU associé dans la trace.

NB : le temps du travail du noyau GPU = (2) - (1)

Afin de tracer le travail du noyau GPU, nous devons tout d'abord définir trois évènements : Un évènement de début et un évènement de fin, et le dernier sera un évènement de référence qui va servir pour le calcul du temps de travail GPU, cet évènement se contente d'enregistrer la date de début du fonctionnement de CUDA dans notre programme.

```
cuEvent cuda_start, start, stop;
```

Il fallait par la suite enregistrer ces évènements là au bon moment. Pour l'évènement de démarrage du noyau, nous devons l'enregistrer naturellement au moment du lancement de *cudaLaunchKernel*, et pour son arrêt, il devra se faire au moment de la fin de *cudaLaunchKernel*. Ces évènements seront liés par la suite à l'enregistrement du travail du noyau sur le GPU.

```
/* création de l'évènement de début du travail du noyau */
cuEventCreate(start);
/* création de l'évènement de fin du travail du noyau */
cuEventCreate(stop);

/* enregistrement de la date de l'évènement de début */
cuEventRecord(start);
```

```
/* enregistrement de la date de l'évènement de fin */
cuEventRecord(stop);
```

Après, il faut synchroniser ces nouveaux évènements, c'est-à-dire, attendre la fin des enregistrements du travail du noyau pour qu'on ait des valeurs cohérentes. Ceci s'effectue pendant l'interception de *cudaThreadSynchronize*, la méthode responsable à la synchronisation des noyaux lancés sur le GPU.

```
/* attente du passage effectif par l'évènement start */
cuEventSynchronize(start);

/* attente du passage effectif par l'évènement stop
cuEventSynchronize(stop);
```

La dernière étape consiste à extraire les temps de début et de fin du travail du noyau GPU, à partir des évènements précédemment enregistrés, ceci par le biais de :

```
/* calcul du temps passé entre les deux évènements */
temps1 = cuElapsedTime(&start, cuda_start, start);
temps2 = cuElapsedTime(&stop, cuda_start, stop);
```

Le plus pratique est de faire ceci au moment de la synchronisation, puisque tous les évènements sont bien enregistrés, et le noyau aura bien fini de travailler.

2.5 Support de plusieurs noyaux GPU

Une étape clé de ce projet est d'autoriser le traçage de plusieurs noyaux GPU lancés. En fait, dans les applications de pointe utilisant CUDA, il est très fréquent d'avoir plusieurs noyaux GPU qui se lancent, afin d'exploiter au meilleur la performance de calcul élevée des cartes graphiques. Dans ce cas là, il faut s'assurer d'intercepter tous les noyaux sur les différents threads GPU d'une manière cohérente. Que ce soient des noyaux lancés en séquentiel ou bien en parallèle. Pour se faire, nous avons commencé à créer une structure dynamique d'évènements *start* et *stop*, chacun s'occupait d'un noyau à la fois. Ainsi la capture des évènements se faisait au moment du lancement du noyau sur le CPU, et la synchronisation se faisait au moment de la synchronisation de CUDA. Et par la suite, le calcul des durées se fait de la même manière que dans la section précédente.

Chapitre 3

Évaluation du module

Les performances du module sont aussi importantes que les traces générées, et encore plus en considérant le champ d'application. Une particularité de la majorité des programmes du monde du HPC est le fait qu'ils sont assez longs à s'exécuter - certains durent des semaines voire même plus - ce qui fait qu'un programme de traçage qui augmente le temps d'exécution par un facteur de quatre sera sans doute inutilisable. Nous avons donc décidé d'effectuer non pas un mais deux types de tests de performance. Dans ces tests nous avons mesuré le surcout brut du module ainsi que le surcout du traçage d'un programme qui calcule la fractale de mandelbrot.

3.1 Environnement de test

Toutes les mesures ont été effectuées sur des machines équipées d'un processeur quadri-cœur Intel Xeon, d'une carte graphique NVIDIA Quadro 2000 et de 4Go de mémoire vive. Du côté software, le tout fonctionnait sous la version 17 de la distribution Linux Fedora, avec la version 4.2 de CUDA et la version 1.4.1 de MPICH2.

3.2 Surcout brut

Cette première phase d'évaluation consiste à mesurer le surcout ajouté par le traçage des fonctions élémentaires de CUDA. Nous pouvons distinguer deux types de fonctions, à savoir celles qui sont tracées uniquement au niveau du CPU ($cuMemAlloc$) et celles qui sont tracées à la fois sur le CPU et sur le GPU ($cudaLuanchKernel$, $cuMemCopy$).

3.2.1 Traçage sur le CPU

Pour le premier test, nous avons calculé le temps d'exécution d'un programme faisant uniquement un appel à $cuMemAlloc$. Afin d'obtenir des résultats plus précis, nous avons

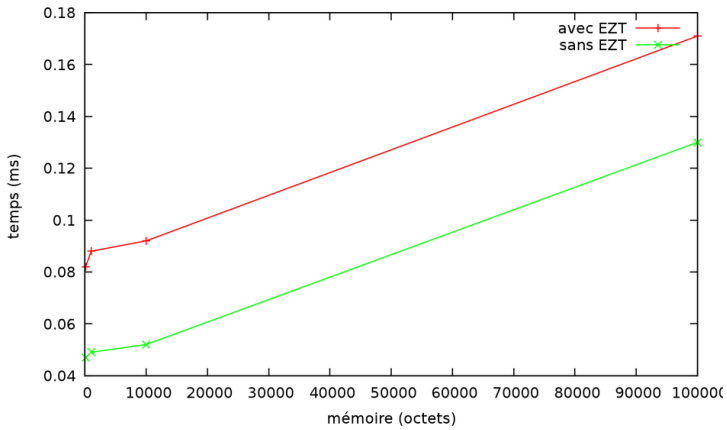

FIGURE 3.1 – Surcout brut de *cuMemAlloc*

en fait calculé la moyenne sur 10^5 appels. Les résultats de l'exécution de ce programme de test avec et sans l'utilisation d'EZTrace sont montrés par la **FIGURE 3.1**. Nous observons ici un surcout constant de 40µs indépendamment de la taille des données allouées. Cette indépendance était prévisible puisque le comportement du module est indépendant de la taille des données à allouer. En effet, le module se contente d'enregistrer deux évènements « EZTrace », le premier au lancement de l'appel à *cudaMemAlloc* et le second lors de sa terminaison (sur le processeur).

3.2.2 Traçage sur le CPU et le GPU

Le traçage au niveau du GPU nécessite un travail supplémentaire du côté du module. En plus des évènements EZTrace, le module crée et enregistre des évènements CUDA qui seront synchronisés au prochain appel de *cudaThreadSynchronize* (ou *cudaDeviceSynchronize*). Ceci n'influence pas gravement les performances du module et on observe un surcout similaire à celui de *cudaMemcopy* (**FIGURE 3.2**).

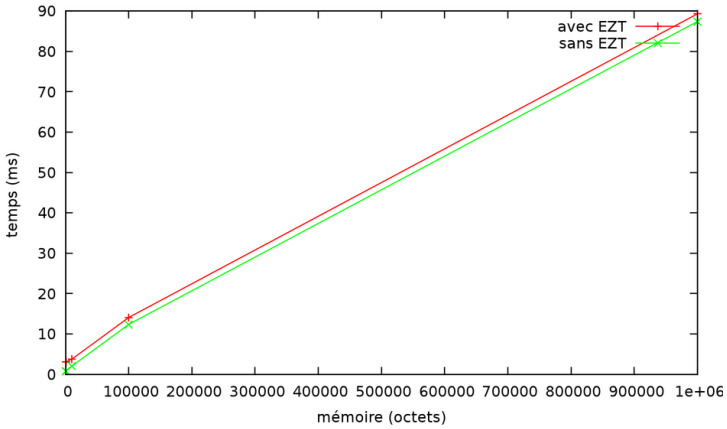

FIGURE 3.2 – Surcout brut de cuMemCopy

3.3 Surcout global

Nous avons utilisé un programme de calcul de la suite de mandelbrot afin d'évaluer les performances du module avec un vrai programme CUDA. Comme le montre la **FIGURE 3.3**, le surcout est presque fixe (~1.9 secondes) et devient donc de plus en plus négligeable avec l'augmentation du temps d'exécution du programme initial.

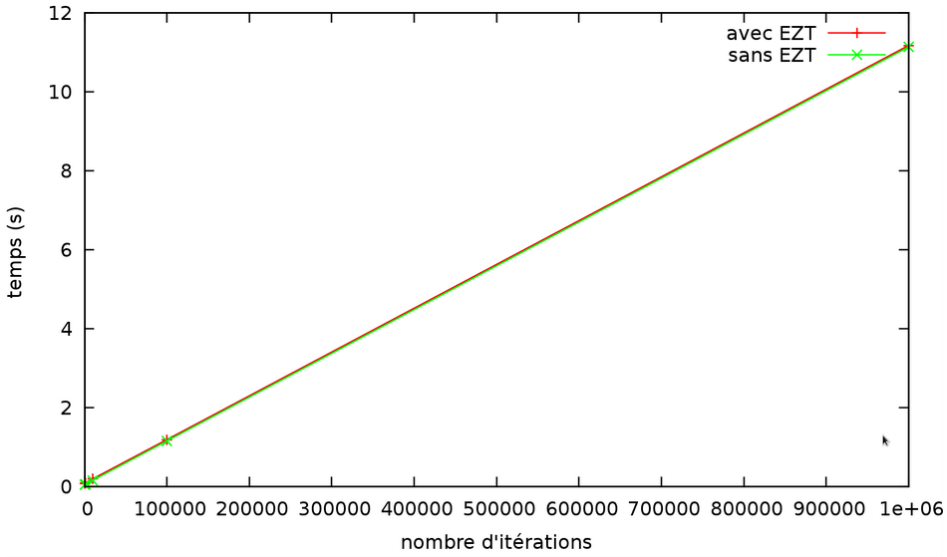

FIGURE 3.3 – Surcout global

3.4 Synthèse

Les quelques tests que nous avons pu effectuer ont montré que les performances du module sont assez satisfaisantes. En particulier, le fait que le surcout global dépend uniquement du nombre d'appels aux fonctions CUDA et du nombre de noyaux lancés prouve que le module n'est même pas affecté par les effets d'échelle (scalability) qui est un facteur majeur dans le domaine du « big data ».

Chapitre 4

Conclusion

La course aux performances des programmes informatiques a conduit à une émergence des accélérateurs de programmes tels que MPI et CUDA. Ces accélérateurs permettent une nette amélioration des performances, surtout avec des programmes hybrides et des architectures matérielles bien adaptées (grilles et grappes). Les outils d'analyse d'exécution comme « EZTrace »deviennent une nécessité pour pouvoir étudier et optimiser l'exécution de ce genre de programmes. Nous avons donc essayé d'amener « EZTrace »un pas en avant en implémentant un module permettant de tracer l'exécution des fonctions CUDA. Nous avons utilisé la bibliothèque CUPTI pour contourner les problèmes rencontrés avec le compilateur *nvcc* et *auto-tools*. Nous avons pu intercepter les fonctions les plus utilisées telles que $cuMemcopy$ et $cuLaunchKernel$. Il est à noter que l'ajout du reste des fonctions CUDA demande uniquement du temps puisque le traitement de toutes les fonctions est presque identique. Nous avons ensuite généré les traces en utilisant GTG et FXT. Nous avons enchainé ceci avec le support de plusieurs noyaux et GPUs pour conclure àla fin avec des tests de performance du module. Dans son état actuel, le module fonctionne correctement tout en ayant des bonnes performances et sera donc intégré à une prochaine version stable d'EZTrace. Certaines tâches comme le calcul de statistiques avec les métriques CUPTI n'ont pas pu être achevées à cause de la contrainte du temps, et elles feront le sujet de futures améliorations par les développeurs d'EZTrace. Il est susceptible que le module évoluera ultérieurement pour devenir plus portable en OpenCL aussi.

Bibliographie

[1] Introduction à Automake.

[2] Introduction à Autoconf.

[3] Introduction à CUPTI.

[4] Documentation de CUPTI.