Rapport Projet de fin d'étude

Traces d'exécutions pour programmes CUDA

Encadrant :
François Trahay

Étudiants :
Paul-Émile SUBLET
Hervé LOEFFEL

Contenu

Introduction

Les divers logiciels (jeux vidéo, simulation d'événements physiques, météorologiques) créés se veulent de plus en plus performants et donc comportent des calculs augmentant en complexité. Ainsi les programmes deviennent de plus en plus lourds et longs à s'exécuter et à fournir les résultats souhaités. Une des principales solutions à ce problème est l'utilisation de la parallélisation. Il est alors possible de paralléliser une tache entre les cœurs d'une même machine (avec OpenMP), mais aussi de la partager entre plusieurs machines différentes (avec MPI) afin de gagner en vitesse d'exécution.

Afin de gagner encore d'avantage de temps, il est également possible de partager certains calculs dans des threads sur le GPU (Graphics Processing Unit). En effet, il a pour avantage d'effectuer les calculs beaucoup plus rapidement que le CPU. Pour exploiter le GPU il est possible d'utiliser des technologies telles que OpenCL ou CUDA. C'est essentiellement à cette dernière que nous allons nous intéresser durant notre projet.

Cependant il n'est pas aisé pour un programmeur de pouvoir optimiser le temps d'exécution de ces programmes. Pour remédier à cela il existe des logiciels tels que EZTrace pouvant générer une trace d'exécution au programme analysé, voir les durées d'exécution de threads, les échanges entre le CPU et le GPU, ainsi que les divers événements (utilisations de fonctions…).

Néanmoins, malgré une forte utilisation de CUDA dans le développement de logiciels de pointe, EZTrace ne comportait pas de module permettant de faire apparaitre des événements liés à des fonctions CUDA. C'est précisément là où se situe l'objectif de notre projet de fin d'études : développer un nouveau module permettant d'intercepter les fonctions CUDA mais aussi de faire en sorte que les événements liés à ces fonctions soient interprétables par un logiciel de visualisation de traces.

Notre rapport se composera de plusieurs parties distinctes. La première partie portera sur une présentation générale d'EZTrace ainsi que des explications sur son fonctionnement. La seconde section sera sur notre travail concernant l'implémentation du module CUDA. Dans la dernière partie nous parlerons des tests de performance effectués sur notre programme.

Mots clés : CUDA, traces d'exécutions, performances.

I) Fonctionnement d'EZTrace

1) Présentation du logiciel

EZTrace est un outil dont le but est de générer des traces d'exécution de programmes afin de vérifier leur comportement ainsi que leurs performances. Il permet également d'avoir des statistiques sur les ressources utilisées dans les programmes analysés. Ces traces peuvent être modélisées par des logiciels permettant de modéliser ces traces, tel que VITE (Visual Trace Explorer) qui est Open Source.

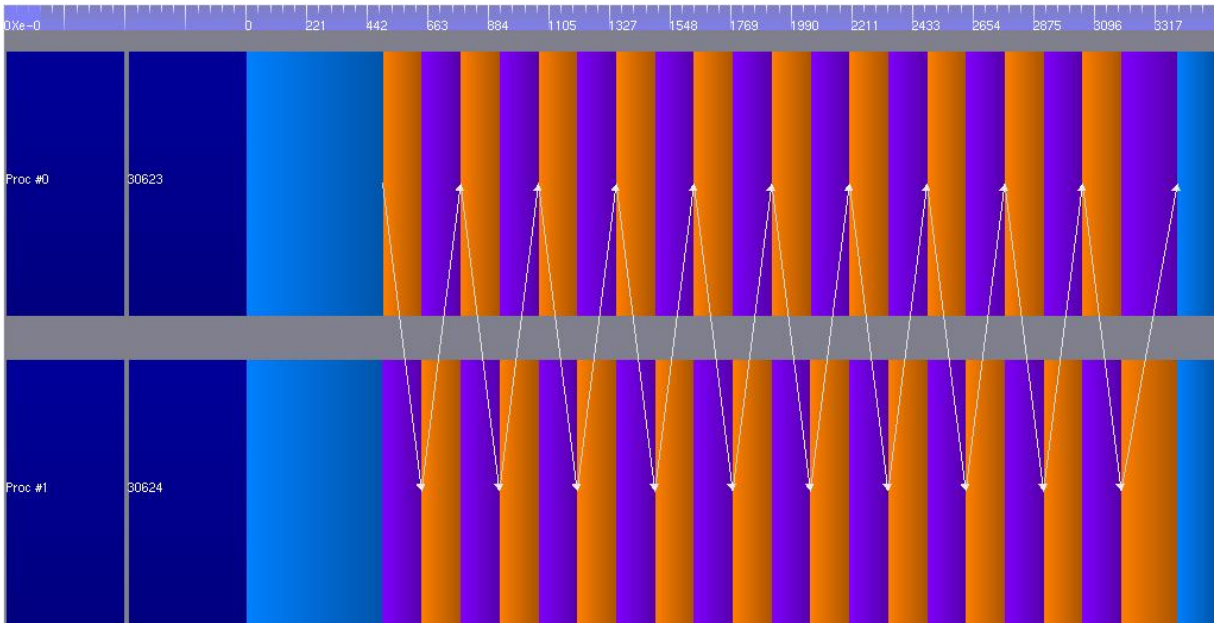
L'utilité d'un tel logiciel pour un développeur est de pouvoir analyser les durées d'exécutions de threads, de synchronisation entre threads, et de diverses fonctions. Ainsi lorsque le programme comporte des taches parallélisées, il permet de comprendre pourquoi l'accélération (temps d'exécution du programme non parallélisé par rapport au même programme parallélisé) peut-être faible et ainsi trouver des solutions pour optimiser son programme et le rendre plus rapide.

Le logiciel EZTrace fonctionne avec des programmes écris en langage C et Fortran et peut reconnaitre :

- L'usage des fonctions de la librairie PThread (création de threads, synchronisation, exclusions mutuelles).
- L'utilisation de fonctions de la librairie OpenMP (parallélisation de programmes entre les cœurs d'une même machine).
- L'utilisation de fonctions de la librairie MPI (parallélisation de programmes entre plusieurs machines différentes)
- Les fonctions d'entrée-sortie (lecture/écriture de fichiers,…)
- L'allocation dynamique de la mémoire

EZTrace peut également analyser des programmes comportant plusieurs de ces librairies (ex : MPI + OpenMP) sans que cela ne pose de problèmes.

Les traces générées par EZTrace sont par défaut au format Pajé, mais il peut également générer une trace au format OTF. La trace peut alors être interprétée par des logiciels comme VITE ou VampirTrace qui permettent à l'utilisateur de la visualiser sous forme graphique.



Exemple de trace visualisée sur VITE

2) Interception des fonctions

Le principe de base pour générer une trace est de redéfinir les fonctions des bibliothèques que l'on souhaite intercepter. Lorsqu'on lance un programme avec EZTrace, celui-ci va utiliser les fonctions qui seront redéfinies et enregistrer des événements dans une trace brute stockée d'abord dans la ram, puis dans un fichier temporaire. Ce fichier sera ensuite converti dans un format standard comme Pajé ou OTF grâce à un convertisseur de trace.

Dans cet exemple, une application CUDA appelle la fonction cudaMalloc. Lorsqu'on lance le programme avec EZTrace avec une commande du type :

```
$ eztrace ./monApplication
```

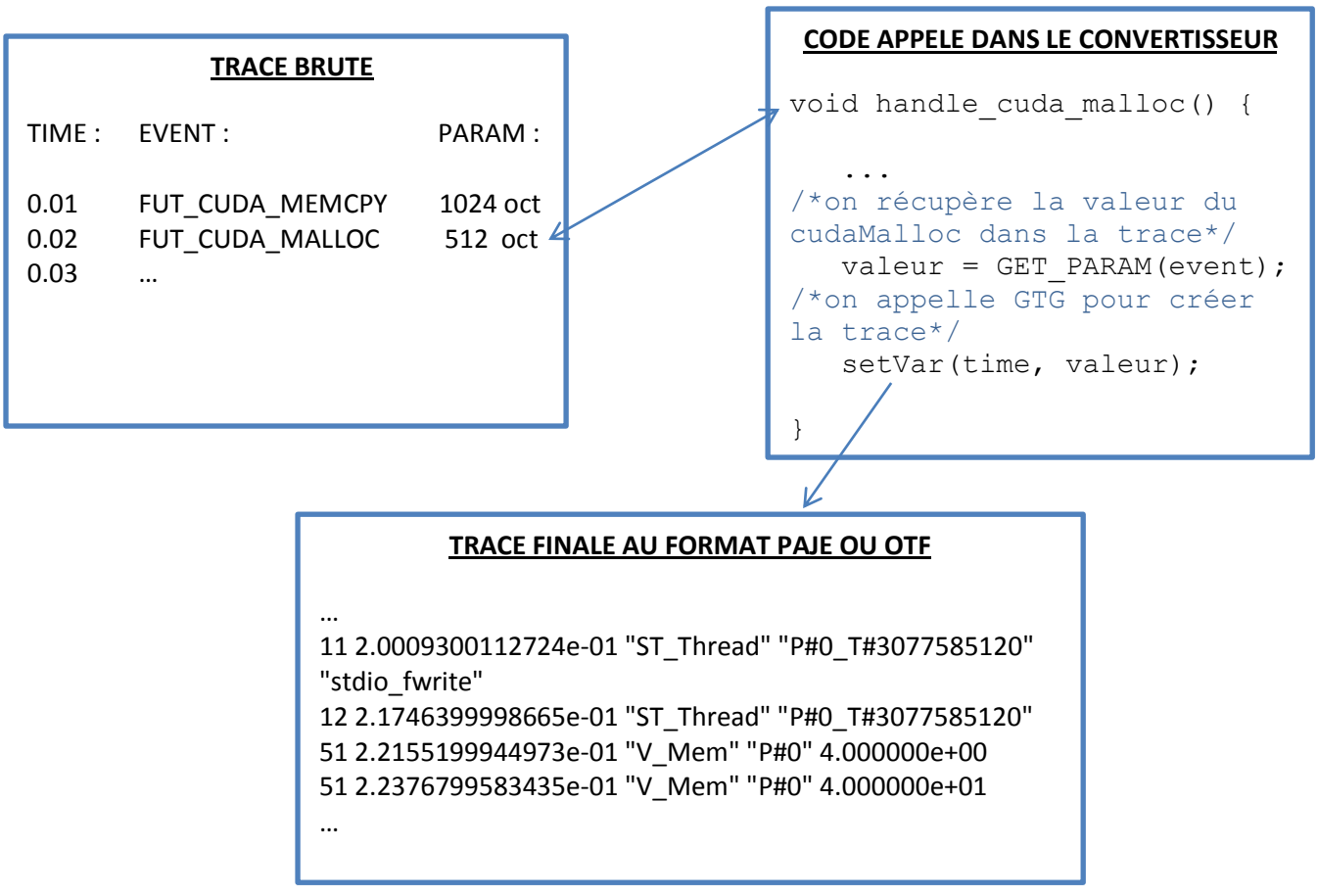
L'application se lance avec EZTrace. Plutôt que d'utiliser le vrai cudaMalloc de la bibliotheque libcuda, l'application va utiliser le cudaMalloc redéfini dans le fichier cuda.cu (sur le schéma : libEZTraceCuda) qui enregistre des événements avant et après l'appel au vrai cudaMalloc et qui seront stockés dans la trace brute et interprétés plus tard. Ainsi, on

peut savoir, grâce aux événements, exactement quand a été exécuté le cudaMalloc dans l'application.
En effectuant cette méthode sur un ensemble de fonctions redéfinie, on arrive à générer une trace représentative de l'exécution d'une application, moyennant un léger surcout en temps, lié à la sauvegarde des événements en temps réel. Il reste encore à transformer la trace brute, stockée dans un fichier binaire illisible, en une trace visualisable grâce a VITE.

3) Représentation des fonctions

La représentation des fonctions est donc la 2ème étape du processus de création de traces. Elle consiste à transcrire la trace brute en une trace standard. Pour cela, plusieurs étapes sont nécessaires. Il faut tout d'abord créer des fonctions qui vont correspondre et traiter chaque type d'événement sauvegardé dans la 1ère étape. Elles permettront de récupérer des paramètres passés dans les événements. Ces fonctions vont ensuite appeler des fonctions d'une bibliothèque nommée GTG qui se chargera de créer la trace proprement dite.

Dans notre exemple précédent avec cudaMalloc, on trouvera dans le programme de conversion d'EZTrace une fonction nommée handle_cuda_malloc() qui sera appelée à chaque fois qu'un événement FUT_CUDA_MALLOC sera lu dans la trace brute. Cette fonction récupèrera la taille du malloc demandé en analysant l'événement et appellera elle-même une fonction de GTG qui va permettre l'affichage de la valeur de la mémoire allouée sur le GPU en fonction du temps.

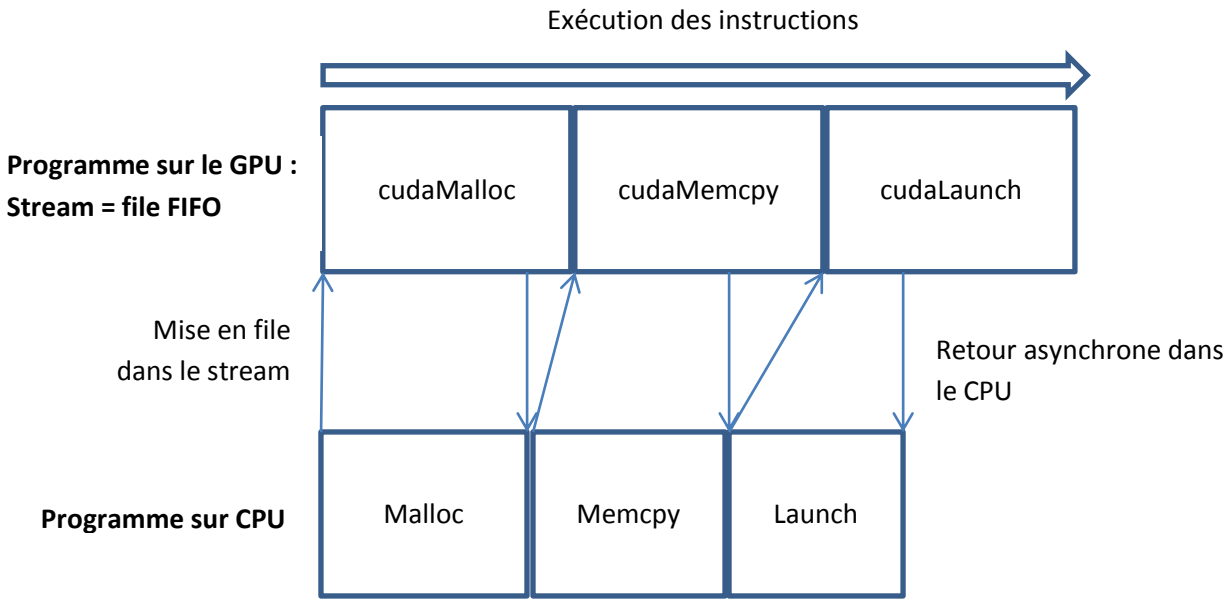


II) Implémentation du module CUDA

1) Fonctionnement de CUDA :

CUDA (Compute Unified Device Architecture) est une technologie de GPGPU (General-Purpose Computing on Graphics Processing Units), c'est-à-dire utilisant un processeur graphique (GPU) pour exécuter des calculs généraux habituellement exécutés par le processeur central (CPU). CUDA permet de programmer des GPU en C grâce à un ensemble de fonctions spécifiques.

L'exécution des commandes par le GPU se fait de manière FIFO. Un stream, correspondant à une file FIFO, reçoit un ensemble de commandes CUDA qui sont ensuite exécutées au fur et à mesure par le GPU. Ainsi, de nombreuses fonctions CUDA ne font que placer une instruction dans cette file et reviennent immédiatement dans le CPU, c'est pourquoi on les qualifie d'asynchrones :



Le fait que certaines fonctions soient asynchrones va nous poser des problèmes lors de la conception du module. Nous en parlerons dans les parties qui suivront.

2) Ajout d'un nouveau module

La structure d'EZTrace est modulaire, ainsi, il y a un module s'occupant des fonctions de la librairie MPI, un autre pour celles d'OpenMP, etc… L'ajout du nouveau module s'est fait par analogie par rapport au module memory (module s'occupant des fonctions malloc, free, etc…) déjà existant. Les fichiers sources sont alors cuda.cu qui sert à l'interception des fonctions, cuda_ev_codes.h qui contient le code des différents événements EZTrace et eztrace_convert_cuda.c qui sert à convertir la trace brute générée en un fichier au format Pajé ou OTF. En pratique, nous avons copié le module memory, interceptant les fonctions de gestion de mémoire dynamique (malloc, realloc…) car il était relativement simple par rapport aux autres. Nous avons enlevé le contenu sur l'interception des fonctions pour ne garder que la base de tout module EZTrace et avons modifié les noms des variables pour définir le module cuda. La première étape de la construction du module était faite mais il fallait maintenant pouvoir l'intégrer dans EZTrace.

La compilation d'EZTrace se fait grâce à l'utilisation de makefile générés automatiquement grâce à un script configure. Il a très vite fallu modifier le configure pour pouvoir compiler le module et l'intégrer à EZTrace. Sans cette étape, il est impossible de tester le module CUDA. Le principe est de tester la présence de CUDA sur la machine où est installé EZTrace. Si la machine possède une installation de CUDA dans le répertoire par défaut ou dans celui spécifié par l'utilisateur, l'ordinateur compilera le module CUDA.

Une fois cette étape passée, nous avons pu tester simplement la présence du module encore vide de CUDA en tapant la commande suivante dans un terminal :

```
$ eztrace_avail
     …
     Cuda     Module for cuda functions
```

Pour comprendre comment EZTrace intercepte les fonctions, nous avons pu étudier des modules exemples qui sont fournis avec EZTrace et permettent de mieux appréhender le concept de surcharge de fonctions. Maintenant que nous avions les clés en main pour pouvoir lancer le codage du module, nous devions faire des choix concernant les fonctions à intercepter et trouver lesquelles étaient les plus intéressantes.

3) Choix des fonctions interceptées

Nous avons cherché à intercepter les fonctions les plus pertinentes, c'est-à-dire celles qui sont régulièrement utilisées par un programme en CUDA et peuvent apporter une information utile au programmeur. Comme nous l'expliquions, ce sont les fonctions qui prennent le plus de temps qui sont susceptible d'être les plus intéressantes.

cudaMalloc : fonction permettant d'allouer de la mémoire sur le GPU. Cette mémoire permet de rapatrier les calculs faits dans les threads par exemple. Il est intéressant d'intercepter cette fonction pour savoir à tout moment dans le programme la quantité de mémoire allouée sur le device. Ce genre de fonction n'est pas spécialement lent à s'exécuter, c'est pourquoi nous avons choisi de ne pas intercepter un événement de début et de fin de cudaMalloc, mais seulement un seul événement.

cudaLaunch : fonction indispensable et présente a priori dans tout programme CUDA. Cette fonction n'est pas directement écrite dans le code produit par le programmeur. C'est lors de la compilation qu'elle est ajoutée par le compilateur NVIDIA. Elle permet de lancer un kernel de façon asynchrone, qui exécute des calculs dans le GPU. C'est une fonction intéressante car c'est le centre de tout programme CUDA et elle peut prendre beaucoup de temps à s'exécuter dans le cas où on lance un gros kernel.

cudaMemcpy : Cette fonction copie un nombre d'octets correspondant au paramètre count d'une zone de mémoire correspondant à *src à une zone de mémoire correspondant à *dst. Concrètement, elle permet de copier un segment de mémoire du CPU vers le GPU ou l'inverse. Elle sert donc en quelque sorte de vrai « retour » à l'exécution d'un kernel, puisque ces derniers ne renvoient pas de valeurs utiles pour l'utilisateur. Cette fonction peut prendre un petit peu de temps à s'exécuter, puisqu'elle dépend de la communication avec le bus PCI-E, qui n'est pas forcément des plus rapides en termes de débit.

cudaThreadSynchronize : Cette fonction permet de synchroniser tous les threads du GPU lorsqu'ils exécutent des kernels. Comme le lancement des kernels est asynchrone, il faut parfois synchroniser ces threads pour récupérer les données sur le GPU de façon sûre et ne pas prendre le risque que les kernels ne soient pas terminés. C'est donc une fonction extrêmement intéressante à capturer puisqu'on peut passer beaucoup de temps à synchroniser des threads.

cudaEventSynchronize : De la même manière que la fonction précédente, cette fonction permet de synchroniser un événement, c'est-à-dire attendre que celui-ci soit effectivement enregistré par le GPU.

```
cudaEvent event; //création d'un event CUDA
... // Lancement de kernels ou d'opérations sur le GPU
cudaEventRecord(event); //placement de l'event dans le stream du GPU
cudaEventSynchronize(event); //attente de l'enregistrement de l'event
```

C'est donc une fonction qui peut prendre beaucoup de temps à s'exécuter, tout dépend de ce qu'il y a en attente dans le stream du GPU.

__cudaRegisterFunction : Cette fonction est exécutée lors du lancement de n'importe quel programme CUDA. Elle permet de « recenser » les kernels définis dans le programme et de récupérer des données importantes telles que le nombre de threads utilisés par le kernel. Cette fonction est donc très utile pour récupérer le nom des kernels lancés avec cudaLaunch mais ce n'est pas une fonction que l'on va chercher à représenter dans la trace.

4) Développement de l'interception des fonctions

Nous avons commencé par intercepter des fonctions simples pour apprendre à créer des modules pour EZTrace. Ce travail consistait d'abord à choisir le nombre d'événements que nous souhaitions enregistrer. En effet, certaines fonctions prennent beaucoup de temps à s'exécuter. On doit alors mesurer le temps d'exécution de celles-ci. Dans le cas de fonctions comme cudaMalloc, qui prennent peu de temps à s'exécuter, il ne sert pas à grand-chose d'enregistrer le début et la fin de la fonction. On peut enregistrer un seul événement qui servira aussi à transmettre la taille du malloc effectué sur la carte graphique. Nous allons vous parler maintenant des fonctions interceptées qui présentent un intérêt dans ce rapport.

- **cudaMalloc** :

La première fonction à avoir été interceptée a été cudaMalloc. Le but était d'essayer de faire apparaitre dans la trace « brute » l'évènement correspondant au cudaMalloc (FUT_CUDA_MALLOC). Après quelques essais sur nos ordinateurs personnels non fructueux, nous avons testé sur les ordinateurs de la salle b313 et le code fonctionnait bien. Nous pensons que cela peut venir d'une mauvaise installation de CUDA, en effet, celle-ci n'est pas triviale… Ainsi nous avons choisi de sauvegarder l'événement cudaMalloc avec la taille de la mémoire allouée sur le GPU.

- **cudaMemcpy** :

La seconde fonction que nous avons interceptée est cudaMemcpy. Ici nous avons dû faire preuve de plus d'astuce car cette fonction peut être utilisée pour envoyer des données du GPU vers le CPU ou l'inverse. En prévision de l'affichage que nous voulions en faire, nous avons choisi de sauvegarder le nombre d'octets qui étaient transférés et le sens du transfert. Pour cela, nous analysons les paramètres qui sont passés à la fonction et récupérons le type de memcpy effectué et grâce à un switch, nous enregistrons les événements correspondant.

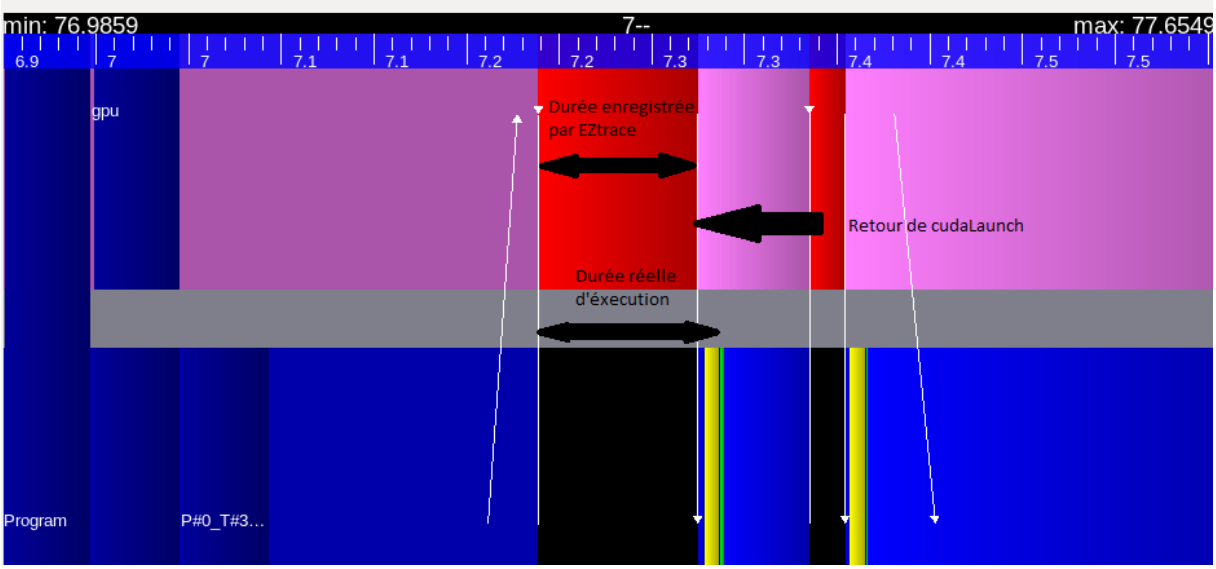
- **cudaLaunch** :

Nous avons vite décidé qu'il fallait intercepter les kernels CUDA, concept fondamental de la programmation GPU. Nous nous sommes vite aperçu qu'il y aurait un problème dû au fait que les kernels n'ont pas un nom défini dans les programmes CUDA. Il n'est donc pas possible d'intercepter les kernels de la même façon que les autres fonctions CUDA, c'est-à-dire en récupérant le pointeur de la fonction originale. Heureusement, nous avons découvert en partie comment se déroulait la compilation et l'exécution d'un programme CUDA. Lors de la compilation, les kernels (écrits avec une syntaxe du type :

`myKernel<<<dimGrid,dimBlock>>> (…);`) sont transformés dans un fichier temporaire en appels aux différentes fonctions pour préparer l'appel du kernel et en cudaLaunch, pour effectuer l'appel au kernel. C'est donc cette fonction que nous allons intercepter et afficher dans la trace.
Un des problèmes de cudaLaunch est que l'appel aux kernels est asynchrone, c'est-à-dire que lorsqu'on appelle un kernel, celui-ci est lancé sur les différentes unités de calculs du GPU et le retour dans le CPU est fait très rapidement après, même si le GPU n'a pas fini son calcul. Dans un premier temps nous n'avions pas considéré le fait que cette fonction effectuait un retour asynchrone et nous étions partis sur le schéma classique consistant à intercepter les évènements de la façon suivant :

```
cudaError_t cudaLaunch (const char *entry) // réécriture de la function cudaLaunch
{
	…
	EZTRACE_EVENT0(FUT_CUDA_LAUNCH_ENTRY);	// événement de lancement du kernel
	libcudaLaunch(kernelInfo); // lancement du kernel
	EZTRACE_EVENT0(FUT_CUDA_LAUNCH_EXIT);	// événement de terminaison du kernel
}
```

Le schéma ci-dessous montre les résultats que nous obtenions et le caractère asynchrone, la partie jaune étant la synchronisation des threads du kernel, donc la fin effective du déroulement du kernel :



Une solution évidente de ce problème aurait été de mettre une synchronisation des threads entre l'appel au vrai cudaLaunch et l'enregistrement de l'événement de terminaison :

```
libcudaLaunch(kernelInfo); // lancement du kernel
cudaThreadSynchronize(); // synchronisation des threads du GPU
EZTRACE_EVENT0(FUT_CUDA_LAUNCH_EXIT); // événement de terminaison du kernel
```

De cette façon, on est sûr que le kernel a bien fini de s'exécuter lorsqu'on enregistre l'événement de terminaison et que l'on revient dans le CPU. Le gros problème de cette méthode est que le caractère asynchrone ne doit pas forcément être perdu car il peut être utilisé pour augmenter les performances de l'application. En effet, un programmeur voudra peut-être effectuer des calculs sur le CPU pendant que le kernel s'exécutera sur le GPU, ou peut-être voudra-t-il lancer plusieurs kernels à la suite sans attendre la synchronisation. On remarque donc que ce n'est pas une solution acceptable.
Nous avons donc dû réfléchir à une autre solution basée cette fois sur les événements CUDA et sur les possibilités de mesurer des temps d'exécutions avec. Le principe est de placer au début de notre cudaLaunch un événement CUDA qui signale le lancement effectif du kernel et à la fin de placer un événement qui signale la terminaison du kernel par tous les threads du GPU. Il suffit de calculer le temps entre les deux événements pour avoir le temps d'exécution réel du kernel. Voilà le principe simplifié de notre méthode :

```
cudaEvent start, stop ; // création des événements
cudaEventRecord(start) ; // enregistrement de la date de l'événement de début
libcudaLaunch(kernelInfo); // lancement du kernel
cudaEventRecord(stop); // enregistrement de la date de l'événement de fin
cudaEventSynchronize (stop); // attente du passage effectif par l'événement stop
executionTime = cudaElapsedTime(start, stop); // calcul du temps passé entre les deux événements
```

Pour que cette méthode marche même si on appelle un kernel 2 alors que l'on n'a pas fini de calculer le temps d'exécution du kernel 1, on doit créer un tableau d'événements CUDA et faire varier sa taille en fonction de la demande. Nous avons mis en place un tableau alloué dynamiquement grâce à des realloc qui permettent de résoudre ce problème.
De plus, on constate que cette méthode permet de récupérer le temps passé à exécuter le kernel mais ne donne pas d'information sur la date de lancement effectif du kernel. Il manque donc de l'information pour que l'on puisse afficher correctement dans VITE la trace de notre programme. La solution trouvée a été de créer un événement de référence qui associe une date CPU avec une date GPU. Ainsi, il suffit de faire un simple calcul du temps passé entre cet événement de référence et l'événement start pour avoir la date d'exécution effective du kernel. Le temps d'exécution et la date de démarrage du kernel sont passés en paramètres d'un événement EZTrace, spécialement défini pour cela, lors de l'appel à cudaThreadSynchronize. Nous sommes partis du principe que chaque programme CUDA appelle au moins à un moment un cudaThreadSynchronize et c'est dans cette fonction que nous calculons les différents temps sur les derniers kernels appelés depuis le dernier cudaThreadSynchronize. La définition de cet événement de référence se fait lorsqu'on appelle pour la première fois cudaMalloc car c'est une des premières fonctions appelée dans un programme CUDA. Nous parlerons plus en détail du principe de cet événement dans la partie suivante.

- **__cudaRegisterFunction :**

Cette fonction est un peu spéciale car elle n'est pas directement écrite dans le code produit par l'utilisateur. C'est une fonction qui est appelée lors du lancement de n'importe quel programme CUDA. Elle permet à l'ordinateur de savoir toutes les informations nécessaires pour lancer les kernels correctement (nom du kernel, nombre de threads alloués pour le calcul sur le GPU…). Cette fonction est intéressante à intercepter pour récupérer toutes ces

informations utiles à la visualisation sur la trace, même si il faut faire attention lors de la manipulation des données qu'elle utilise puisque l'exécution correcte du programme dépend de celles-ci.

Nous avons choisi de stocker les informations des différents kernels contenus dans le programme dans une liste chainée. Cela permet par la suite de retrouver le nom du kernel exécuté avec cudaLaunch et de l'intégrer à la trace. A terme, l'utilisation de cette liste chainée pourrait permettre de véhiculer des informations telles que le nombre de blocs et la taille de la grille utilisée pour le cudaLaunch.

5) Développement de la conversion de la trace brute

La conversion de la trace brute, générée lors de l'exécution du programme avec EZTrace est la deuxième étape de la création d'une trace visualisable. Elle consiste à traiter tous les événements stockés de façon binaire dans la trace brute pour pouvoir générer une trace dans un format standard comme OTF ou Pajé. Pour cela, EZTrace utilise une bibliothèque nommée GTG pour Generic Trace Generator. Grâce à cette bibliothèque, nous n'avons pas eu besoin d'écrire directement la trace. Il nous a suffi d'appeler les fonctions de GTG pour écrire un fichier trace générique.

Notre travail a donc consisté à écrire un ensemble de fonctions qui correspondent aux événements enregistrés. Ce travail était fait plus ou moins en parallèle du travail d'interception de fonctions puisque l'un ne va pas sans l'autre. Il y a en effet besoin de transmettre, via les événements, des données intéressantes pour la trace finale. Ce travail n'était pas entièrement fait de codage en C mais aussi de choix au niveau de la représentation des événements. Nous avons cependant fait face à quelques difficultés dont nous parlerons plus loin.

- **cudaMalloc :**

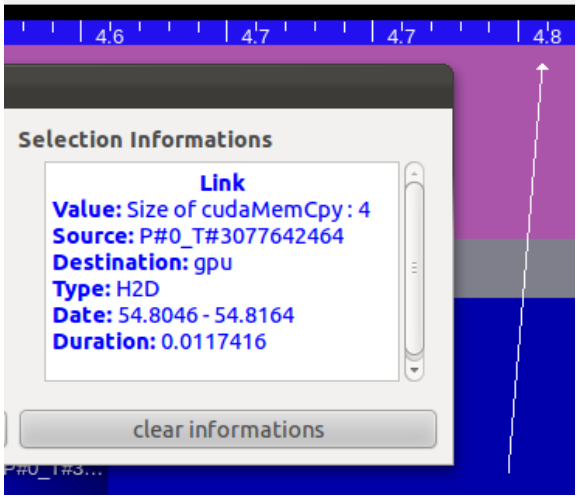
Nous avons commencé naturellement par faire notre première représentation d'événements avec cudaMalloc. Nous avons choisi d'utiliser la représentation de la taille de la mémoire utilisée en fonction du temps. Pour cela, nous récupérons la taille du malloc effectué dans le programme de l'utilisateur grâce à l'événement enregistré lors de l'exécution du programme :

```
Void handle_cuda_malloc() // fonction appelée pour gérer l'événement cudaMalloc
{
        …
        Int block_size = CUDA_GET_PARAM(CUR_EV, 1) ; // récupération de la valeur du malloc
        CUDA_CHANGE() setVar(CURRENT, …, block_size); //fonction GTG pour afficher une variable
        …
}
```

Nous obtenons au final une trace qui comporte une fonction représentant l'évolution de la mémoire :

- **cudaMemcpy :**

Nous avons choisi cette fois de représenter cet événement sous la forme d'une flèche qui part du GPU vers le CPU ou l'inverse suivant le type de cudaMemcpy appelé. Nous en avons profité pour indiquer le nombre d'octets que transférait le cudaMemcpy :

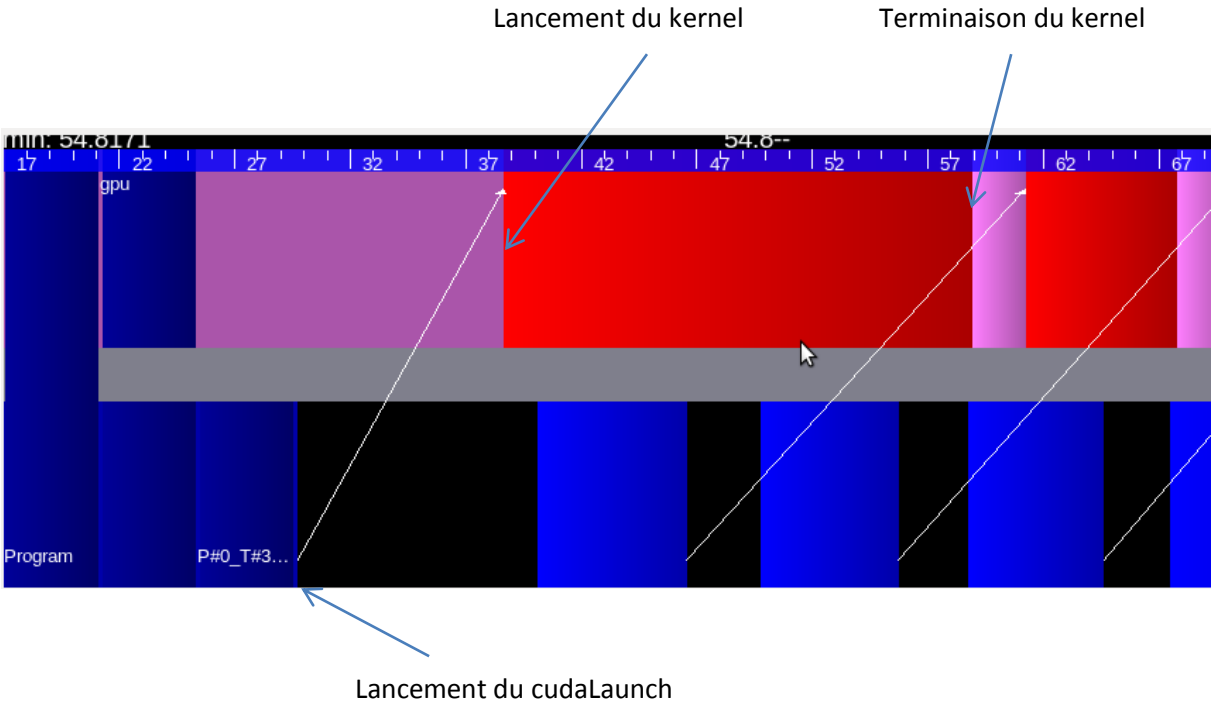


- **représentation des kernels :**

Cette étape a été la plus compliquée à mettre en place dans cette partie du développement du module. En effet, le passage du nom du kernel se fait via les événements de lancement de kernel et l'affichage se fait dans la fonction liée à l'événement spécial (qui contient le temps d'exécution d'un kernel ainsi que sa date de lancement) enregistré lors du cudaThreadSynchronize (FUT_CUDA_KERNEL_EXEC_TIME). Nous avons mis en place une file d'attente de type FIFO pour passer les différentes informations nécessaires à la représentation d'un kernel CUDA de la fonction traitant les événements d'entrée dans cudaLaunch à la fonction traitant les événements contenant les informations sur les temps d'exécutions des kernels.

Pour pouvoir positionner sur la trace finale le déroulement du kernel, il faut utiliser la référence de temps dont nous parlions tout à l'heure. Le principe est d'intercepter un événement correspondant à une référence entre le GPU et le CPU. Avec cette référence, on va pouvoir placer le lancement effectif du kernel puisqu'il suffit de le placer au temps correspondant à la somme de la position de la référence et de la position du départ du kernel par rapport à la référence.

Nous obtenons alors des traces de ce type avec, en rouge, les kernels :



III) Tests de performance

Nos différents essais ont pour but d'évaluer le ralentissement d'un programme lorsqu'il est analysé par EZTrace. On souhaite alors avoir un ratio temps d'exécution avec EZTrace par rapport au temps sans analyse le plus proche possible de 1 (sachant qu'un résultat au-dessus de 1 serait incohérent et peut montrer une erreur dans l'implémentation de notre module), afin de faire gagner du temps à l'utilisateur.

1) Lancement de kernels multiples

Ce test consiste à utiliser un programme simple comportant un nombre important de kernels à lancer. Ces derniers ne font pas de calculs complexes. Pour que le programme soit plus «réaliste », nous avons effectué une synchronisation tous les dix lancements de kernels. On passe alors en paramètre le nombre de kernels divisé par dix. La mesure de la durée d'exécution se fait en plaçant des timers au début et à la fin des programmes puis en faisant la différence entre les deux valeurs.

On obtient alors les résultats suivants :

sans EZTrace

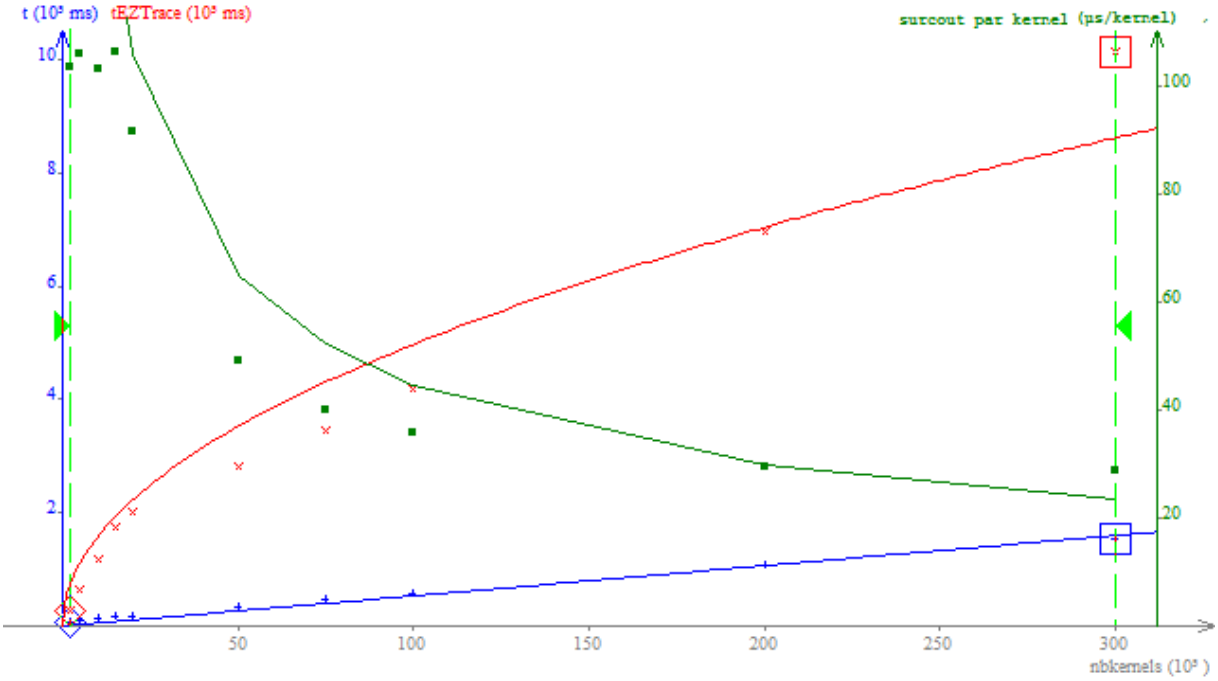
nombre de kernels	2000	5000	10000	15000	20000	50000	75000	100000	200000	300000
temps d'exécution (ms)	70,95	116,31	136,68	154,7	180,73	338,86	451,5	582,62	1060,08	1547,13

avec EZTrace

nombre de kernels	2000	5000	10000	15000	20000	50000	75000	100000	200000	300000
temps d'exécution (ms)	278	645,34	1166	1747,61	2011,63	2792,39	3458,13	4171,59	6955,55	10127
facteur de ralentissement	3,918252	5,548448	8,530875	11,29677	11,13058	8,240542	7,659203	7,160053	6,561344	6,545668
surcout total	207,05	529,03	1029,32	1592,91	1830,9	2453,53	3006,63	3588,97	5895,47	8579,87
surcout par kernel	0,103525	0,105806	0,102932	0,106194	0,091545	0,049071	0,040088	0,03589	0,029477	0,0286

Le facteur de ralentissement correspond au rapport entre le temps mesuré de l'exécution du programme analysé par EZTrace par rapport au temps d'exécution du programme seul. Le surcoût quant à lui correspond à la différence entre ces deux valeurs. On a également rapporté cette valeur par rapport au nombre de kernels lancés.

Ce qui nous donne les courbes suivantes :



- temps sans EZTrace (ms) - temps avec EZTrace - surcoût par kernel

On constate que le ralentissement et le surcoût par kernel, dû à EZTrace diminue lorsque l'on utilise un nombre de kernels très élevé, pour atteindre une valeur inférieure à 30 µs. Ceci peut s'expliquer par le fait que lorsque les évènements ont déjà été créés ils peuvent être recréés plus rapidement. Cependant ce surcoût reste très élevé. Ce coût s'explique par le fait que l'on utilise plusieurs fonctions pour l'interception du lancement d'un kernel. En effet, il y a la création d'évènements CUDA, la nécessité d'utiliser realloc pour allouer de la mémoire afin d'enregistrer ces évènements. Il y a également la recherche des noms de kernels lancés dans une liste. Chacune de ces actions a alors un coût en temps.

2) Utilisation multiple de cudaMemcpy

Pour ce test nous utilisons un programme simple se contentant de lancer un nombre multiple de fonctions cudaMemcpy. Comme précédemment le paramètre du programme est le nombre d'itérations à effectuer dans la boucle for contenant cudaMemcpy. Nous procédons à cette expérience avec et sans analyse par EZTrace et pour différents nombres d'itérations.

Nous obtenons alors les valeurs suivantes :

sans EZTrace

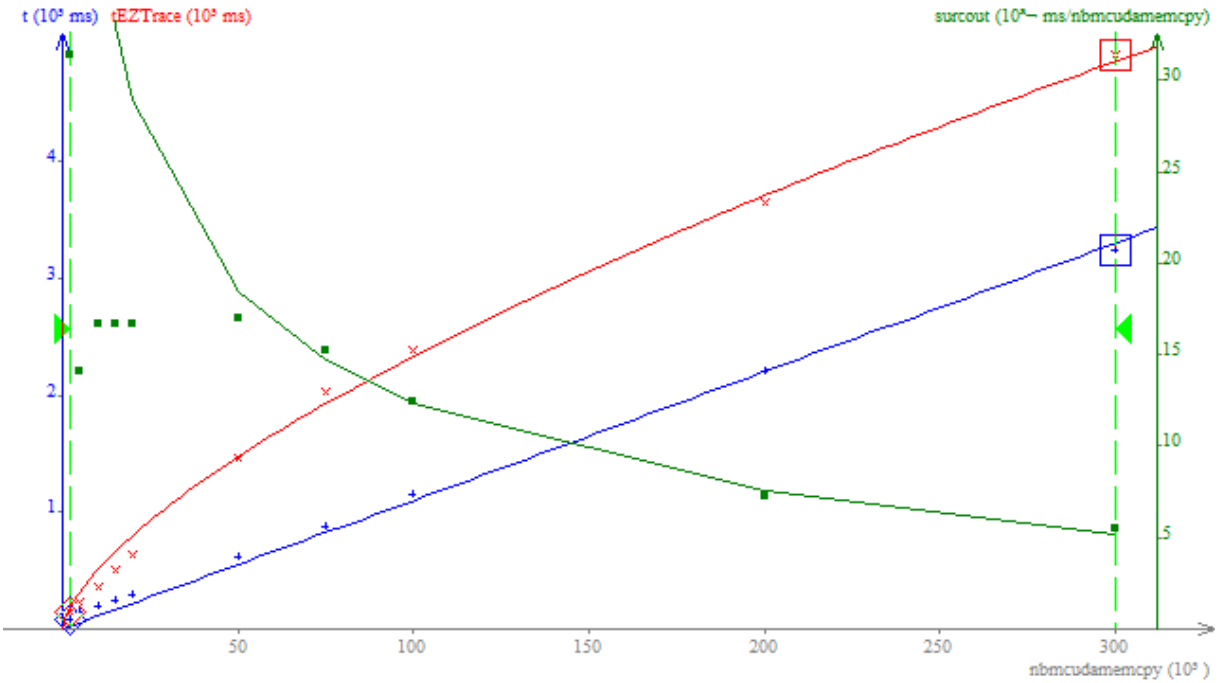
nombre de cudaMemcpy	2000	5000	10000	15000	20000	50000	75000	100000	200000	300000
temps d'exécution (ms)	82,43	149,18	187,09	243,81	295,84	611,8	874,88	1149,41	2197,9	3245,99

avec EZTrace

nombre de cudaMemcpy	2000	5000	10000	15000	20000	50000	75000	100000	200000	300000
temps d'exécution (ms)	145,15	219,51	353,56	494,19	629,38	1458,5	2018,3	2388,37	3648,4	4905,13

ralentissement	1,76	1,47	1,89	2,03	2,13	2,38	2,31	2,08	1,66	1,51
surcout total	62,72	70,33	166,47	250,38	333,54	846,7	1143,42	1238,96	1450,5	1659,14
surcout par cudaMemcpy	0,0314	0,0141	0,01665	0,01669	0,0167	0,01693	0,01525	0,01239	0,0073	0,00553

Nous obtenons le graphique suivant :



- temps sans EZTrace (ms) - temps avec EZTrace - surcoût par cudaMemcpy

Comme pour l'utilisation répétée de cudaLaunch nous obtenons un surcoût lié à l'interception de la fonction. Le surcoût par kernel décroit lorsque le nombre d'utilisation de cudaMemcpy augmente pour atteindre 5 µs, ce qui est bien moindre par rapport au test précédent. Cela peut s'expliquer par le fait que l'interception de la fonction cudaMemcpy est plus légère que l'interception de la fonction cudaLaunch. En effet la fonction réécrite de cudaMemcpy contient juste la création de l'évènement EZTrace selon que ce soit du CPU vers le GPU, ou du GPU vers le CPU.

3) Utilisation de l'algorithme de Mandelbrot

Dans cet essai nous utiliserons l'algorithme de Mandelbrot calculé par le GPU, que nous avons effectué dans le cadre du module « Systèmes hautes performances ». Ainsi des calculs seront effectués dans un kernel. Les calculs sont d'autant plus complexes que le paramètre du programme est élevé. Nous procédons alors de la même façon que pour le programme précédent.

On obtient les résultats suivant :

sans EZTrace

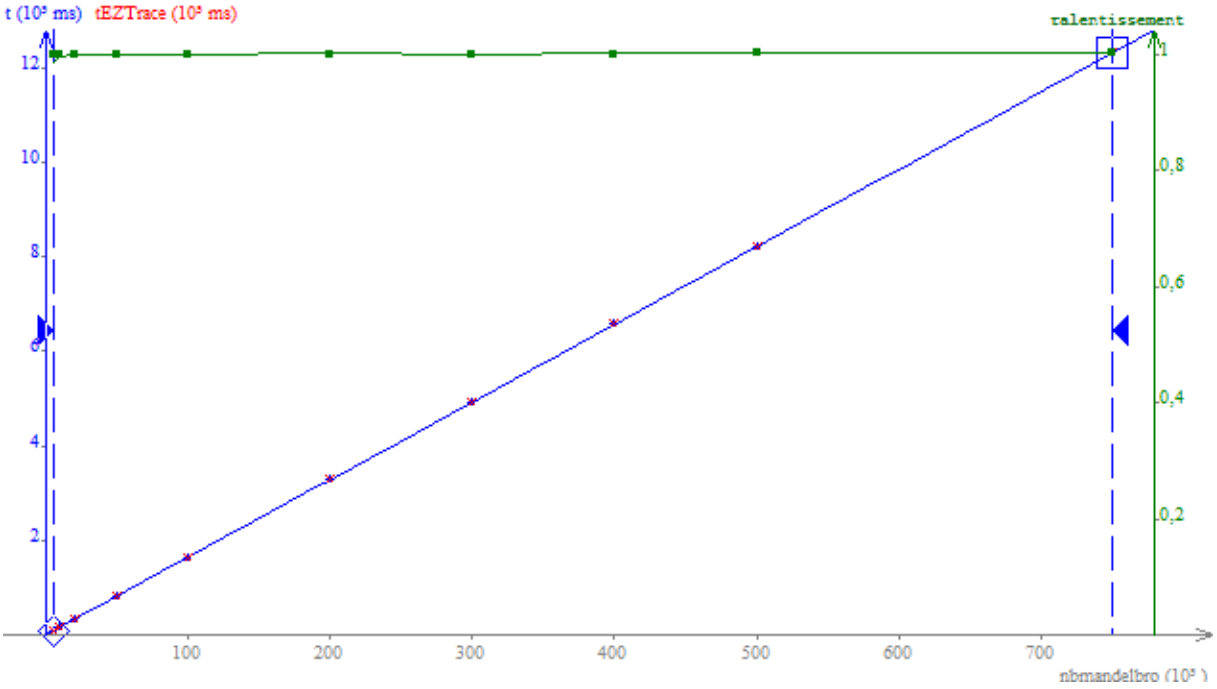
paramètre	5000	10000	20000	50000	100000	200000	300000	400000	500000	750000
temps d'exécution (ms)	83,34	165,54	329,87	822,17	1643,5	3288,5	4930,85	6571,26	8209,81	12314

avec EZTrace

paramètre	5000	10000	20000	50000	100000	200000	300000	400000	500000	750000
temps d'exécution (ms)	83,31	165,58	329,88	822,54	1644,5	3289,2	4930,88	6575,49	8227,76	12329

Facteur de ralentissement	0,99964	1,0003	1	1,0004	1,0006	1,0002	1,00001	1,00064	1,00219	1,0012

Et le graphique suivant :



On constate que les deux courbes (temps d'exécution avec et sans EZTrace) se superposent : le temps d'exécution ne varie quasiment pas, ce qui fait que le ralentissement dû à l'analyse reste à 1. Cela s'explique facilement par le fait que le nombre de fonctions interceptées reste très limité, et qu'EZTrace n'influe pas sur la durée des calculs dans le GPU. Ces calculs se faisant tous dans un même kernel, ce nombre de fonctions n'évolue pas avec le paramètre de Mandelbrot.

Conclusion

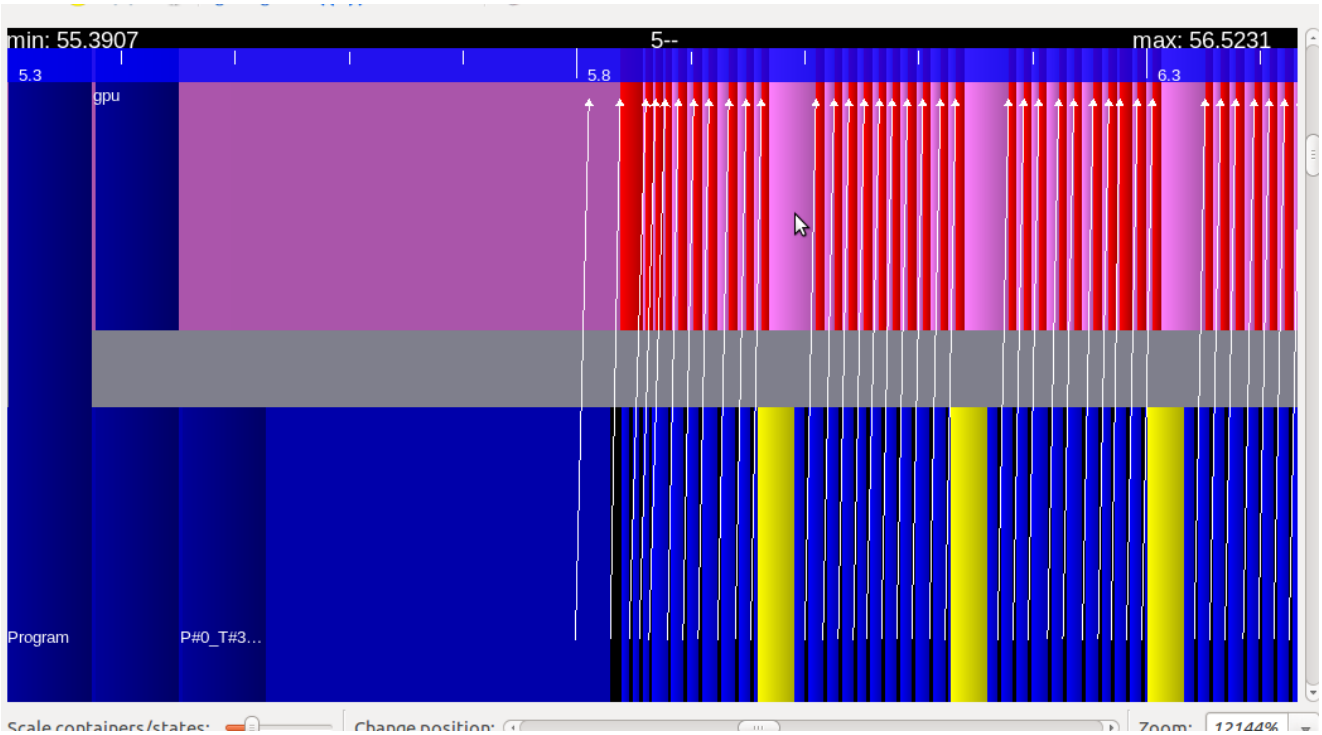
Ce projet aura été l'occasion de travailler sur un logiciel déjà existant. Cela n'a pas été sans difficultés car il a fallu s'immerger dans le fonctionnement du programme, comprendre sa structure, sachant qu'il a été commencé par des développeurs plus expérimentés que nous. Néanmoins on a pu implémenter notre nouveau module CUDA pour EZTrace en s'inspirant de ceux déjà existant.

Notre module permet alors d'intercepter les fonctions CUDA les plus pertinentes, c'est-à-dire, celles couramment utilisées, qui prennent du temps, qui peuvent apporter des informations utiles au développeur. Mais pour que cela soit utile, il a également été nécessaire de pouvoir représenter ces fonctions, de les faire apparaitre dans les fichiers contenant la trace, afin de pouvoir visualiser les évènements liés à ces fonctions sur un logiciel comme VITE.

Nous avons par la suite effectué les tests de performances d'EZTrace avec le module CUDA. De tels tests sont nécessaires pour vérifier que l'utilisation d'EZTrace pour l'analyse ne soit pas gênante pour l'utilisateur. Les différents essais nous ont montré que le ralentissement provient essentiellement des interceptions de fonctions et des calculs supplémentaires induits pour la préparation de la trace. En effet l'exemple du programme calculant l'algorithme de Mandelbrot, nous a montré qu'avec un nombre de fonctions CUDA restreint par rapport au temps de calcul dans le GPU, le surcoût devient faible.

Dans l'avenir, ce nouveau module sera implanté dans une prochaine version d'EZTrace, qui permettra aux entreprises utilisant CUDA d'avoir un outil leur permettant d'optimiser leurs résultats. Cependant il reste des points pouvant améliorer EZTrace, comme l'ajout de statistiques (temps moyen passé dans un kernel, etc…), l'ajout d'autre fonctions CUDA et le support des streams multiples, ou l'ajout d'un module OpenCL.

Annexe



Lancement de multiples kernels (partie rouge) avec synchronisation de threads tous les 10 kernels lancés (jaune). Les flèches représentent les appels du CPU vers le GPU ou l'inverse.