Implémentations parallèles de MD6, une fonction de hachage cryptographique moderne

Arnaud Bienner, Benoit Dequidt
arnaud.bienner@gmail.com, benoit.dequidt@gmail.com

18 juin 2009

Table des matières

1 Introduction et contexte **2**

2 MD6 **2**
- 2.1 Présentation 2
- 2.2 Principe 2
- 2.3 Mode d'opération 3
- 2.4 Fonction de compression 3
- 2.5 Implémentation de référence 4

3 Implémentation parallèle sur CPU avec l'Intel TBB **4**
- 3.1 Présentation de l'Intel TBB 4
- 3.2 Implémentation 4

4 Implémentations sur GPU **5**
- 4.1 Cuda 5
  - 4.1.1 Présentation 5
  - 4.1.2 Particularités de programmation 5
- 4.2 Chaque bloc en parallèle 5
- 4.3 Parallélisation de la fonction de compression 6
- 4.4 Augmentation du nombre de threads par bloc 6
- 4.5 Utilisation d'un tableau circulaire . . . 6
- 4.6 Tabulation des valeurs pour l'accès au tableau circulaire 6

5 Évaluation des performances **7**
- 5.1 Machines de test 7
- 5.2 Tests 7

6 Conclusion **8**

Résumé

Cet article étudie les différentes possibilités d'implémentations parallèles de la fonction de hachage cryptographique MD6.
MD6 fait partie des fonctions de hachage candidates pour le concours SHA-3, destiné à trouver une remplaçante à l'actuel SHA-2, de plus en plus fragile.
Nous avons cherché à voir à quel point une fonction de hachage comme MD6 pouvait tirer parti efficacement du parallélisme présent dans les architectures actuelles (machines multi-cœurs/processeurs, GPGPU).

1 Introduction et contexte

Les fonctions de hachages cryptographiques : La sécurité des données est vitale en informatique. C'est un secteur qui se développe grâce aux signatures électroniques et surtout à internet avec les flux d'informations échangés via ce média. Les fonctions de hachages cryptographiques peuvent être utilisées pour vérifier l'intégrité de données. Il faut en effet être capable de vérifier que le document n'a pas été altéré par une personne malveillante.
De plus, elles permettent des mécanismes d'authentification par mot de passe sans stockage de ce dernier[1].

SHA-3 : Pendant longtemps, MD5 fut utilisé. Face à la découverte de failles et à cause de l'augmentation de la puissance de calcul des ordinateurs (rendant certaines attaques autrefois inexploitables réalisables), SHA-1 fut mis en place.
La famille SHA-2, une variante de SHA-1 (comprenant SHA-224, SHA-256, SHA-384 et SHA-512) fut aussi développée.

Malheureusement, des attaques ont été développées qui permettent de casser SHA-1 beaucoup plus rapidement que par une recherche exhaustive. Même si ces attaques sont actuellement à la limite de ce qui est réalisable, cela suffit à rendre SHA-1 non cryptographiquement sûr. SHA-2 étant basé sur le même algorithme que SHA-1, il pourrait être touché par ses attaques.

En conséquence, une compétition a été organisé par la NIST pour trouver une nouvelle fonction de hachage qui sera appelée **SHA-3**.

[1]En effet, dans un tel système, il suffit juste de stocker l'empreinte, et non pas le mot de passe initial

L'un des candidats les plus intéressant de cette compétition est l'algorithme **MD6**, que nous étudions dans cet article.

Le parallélisme : Pendant des années, il y a eu une course à la fréquence au niveau des processeurs avec des gains importants. Cependant aujourd'hui nous atteignons des limites physiques qui limite cette évolution.

Ce sont donc les architectures parallèles qui se développent aujourd'hui : multiprocesseurs et multi-cœurs qui permettent de gagner en puissance de calcul sans augmenter la fréquence.
Booster par le business des jeux vidéo, les cartes graphiques (et surtout leurs processeurs graphiques) ce sont également énormément développées. Étant par nature destinées à des applications hautement parallèle comme le rendu graphique, elles possèdent un grande nombre d'unité de calcul.
Elle sont aujourd'hui exploitables pour exécuter des algorithmes parallèles qui ne sont pas forcément en rapport avec le traitement d'image, grâce à des outils comme CUDA par exemple (voir 4).

2 MD6

2.1 Présentation

MD6 fait partie des vingt fonctions de hachages favorites pour la compétition de la NIST.

MD6 a été développée par Ronald L. Rivest, cryptologue américain qui inventa MD5 et participa au développement de RSA, avec Shamir et Adleman.

Nous nous intéresseront dans cette partie au fonctionnement de MD6, pour comprendre où se trouve les sources de parallélisme.

Nous ne nous attarderons pas sur la sécurité de MD6, qui n'est pas le but de cet article, et qui a été démontré dans [1] et [2].

2.2 Principe

MD6 prend en entrée un message de taille au plus 2^{64} bits, pour lequel il calcule une empreinte de d bits, où $0 < d \leqslant 512$ bits.

d vaut 256 par défaut mais peut être spécifié.

De même, un certain nombre de paramètres, possédant des valeurs par défaut, peuvent être modifiés :

Clé K : nulle par défaut (et donc de taille 0). Peut être utilisée pour du salage.

Hauteur de l'arbre L . En effet, comme décrit plus loin dans 2.4, MD6 utilise un arbre de Merkle quaternaire. Sa hauteur peut donc être spécifié. Si L vaut 0, la compression se fait selon un schéma séquentiel (de type Merkle-Damgård) permettant d'obtenir le même résultat. Si L est inférieur à 64, la compression peut être hybride, car commençant pour un parcours d'arbre pour finir pour un parcours séquentiel.

Nombre de tour r : par défaut, $r = 40 + \lfloor d/4 \rfloor$, donc avec la valeur par défaut de d (256), $r = 104$.

Les autres parmètres : les constantes utilisées par la fonction de compression (Q), ou les indices des valeurs hachés à réutiliser pour le hachage du tour de boucle suivant($t1, t2, t3, t4, t5$) peuvent aussi être modifié, sous certaines conditions. Par exemple, les valeurs des ti ne peuvent excéder 89.

2.3 Mode d'opération

Le mode d'opération par défaut repose sur un arbre de Merkle quaternaire.

C'est ce mode qui nous intéresse ici, car il est une source naturelle de parallélisme, de par sa nature d'arbre : en effet, chaque nœud de l'arbre peut être calculé en parallèle.

L'unité de mesure est le **mot**, qui vaut 8 octets, ou 64 bits.

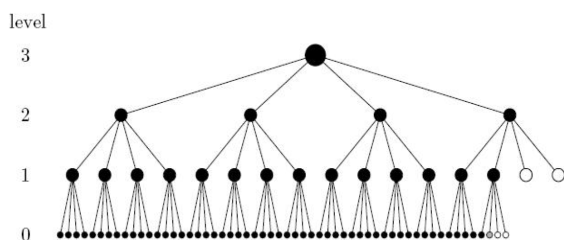

FIG. 1 – L'arbre de Merkle de MD6

Les feuilles de l'arbre proviennent du fichier à hacher. Elles sont éventuellement complétées par des 0 pour obtenir des feuilles de taille 16 mots, soit 128 octets ou 1024 bits. De plus, chaque nœud ayant 4 fils, des feuilles/nœuds supplémentaires composés de 0 sont ajoutés pour obtenir le bon nombre de feuilles/nœuds.

Chaque bloc, composé de quatre nœuds, est compressé (avec la fonction de compression détaillée dans la partie 2.4) pour obtenir en sortie un nœud de taille 16 mots.

Ainsi, on remonte dans l'arbre jusqu'à n'avoir plus qu'un seule nœud : la racine.

L'empreinte de la fonction de hachage est calculée en tronquant les d derniers bits de la racine de l'arbre (256 par défaut).

2.4 Fonction de compression

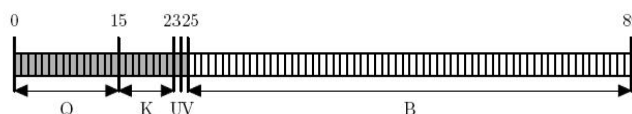

FIG. 2 – Données en entrée de la fonction de compression

La fonction de compression prend en entrée un vecteur de 89 mots, représenté à la figure 2 et composé des éléments suivants :

Q un vecteur de 15 mots, égal à la partie fractionnaire de $\sqrt{6}$

K les 8 mots de la clé (ou 0 s'il n'y a pas de clé)

U 1 mot, indiquant la position du bloc, dont le contenu est détaillé à la figure 3

V 1 mot, dont le contenu est détaillé à la figure 4

B 1 bloc de 64 mots de données, correspondant à 4 nœuds de 16 mots.

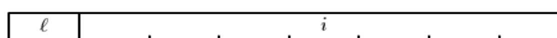

FIG. 3 – Mot unique d'id U, entrée auxiliaire de la fonction de compression, composé d'un octet représentant le niveau dans l'arbre, et de 7 octets représentant la position dans le niveau.

La fonction de compression, effectue ensuite r tours (par défaut 104), composé chacun de **16 boucles indépendantes pouvant donc être parallélisées**. Chacune des ses 16 boucles effectue une quinzaine d'opérations logiques ou de décalage de bits[2].

[2]Les opérations logiques sont préférées aux opérations arithmétiques et aux opérations dépendant de branchement afin d'empêcher les attaques par canaux auxiliaires

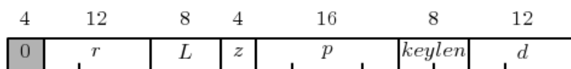

FIG. 4 – Mot de contrôle V, entrée auxiliaire de la fonction de compression. r est le nombre de tour, L la hauteur maximale de l'arbre, z vaut 1 lors de la dernière compression (celle qui permet d'obtenir la racine) 0 sinon, p est le nombre de 0 ajouté (*padding*), $keylen$ la taille de la clé et d la taille de l'empreinte à générer

Chaque tour calcule 89 mots, à partir des 89 mots calculés précédemment (les 89 premières valeurs sont le vecteur de 89 mots de la figure 2 passé en entrée de la fonction de compression).

Les 64 derniers mots calculés sont la sortie de la fonction de compression.

2.5 Implémentation de référence

L'implémentation de référence, disponible sur le site de la compétition (`http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html`) est basée sur le principe des listes de Radix, ce qui garantit un coût mémoire en $\log_4 n$.
Chaque niveau est rempli avec 4 blocs. Dès qu'un niveau est plein, on le vide en compressant les 4 blocs qu'il contient pour obtenir un bloc du niveau supérieur. Le niveau 0 est rempli avec les données lues dans le fichier à hacher.

Cette version n'est directement parallélisable. Nous avons donc développé notre propre implémentation de MD6, qui compresse les données niveau par niveau. Nous avons un coût mémoire en $O(n)$ car au début, tout le fichier est stocké en mémoire. Nous avons besoin à chaque instant de 2 tableaux : un premier stockant les blocs du niveau courant (à compresser) et un second, 4 fois plus petit, contenant les résultats de la compression des blocs du niveau actuel. Une fois que le niveau supérieur est remplit, ce niveau supérieur devient le niveau courant et on alloue un nouveau tableau vide 4 fois plus petit qui devient le niveau suivant/supérieur à remplir.

Nous avons réimplémenté seulement le mode d'opération, pour pouvoir facilement le paralléliser. La code passant le plus grande partie de son temps dans la fonction de compression (que nous n'avons pas modifiée), les performances obtenues sont les mêmes pour notre version et pour celle fournie par R.L. Rivest.

Sur notre machine de test (dont les caractéristiques sont données en 5.1) le débit obtenu est de **27,5 Mo/s** en moyenne (en hachant des fichiers de taille différente).

3 Implémentation parallèle sur CPU avec l'Intel TBB

3.1 Présentation de l'Intel TBB

Threading Building Blocks (TBB) est une bibliothèque C++ développée par Intel afin d'écrire des programmes qui tirent profit des architectures multiprocesseurs/multicœurs[3]. C'est une bibliothèque contenant un ensemble de structures de données et d'algorithmes qui permettent aux développeurs d'écrire leurs codes sans se soucier de la gestion et des threads comme dans certain *package* comme POSIX Threads, Windows threads ou Boost Threads dans lesquels il faut gérer la création, la synchronisation et la terminaison de chaque thread. La bibliothèque permet de considérer les opérations comme des tâches qui vont être automatiquement et dynamiquement réparties sur l'ensemble des ressources disponibles, tout en utilisant de manière efficace le cache.

3.2 Implémentation

Nous avons choisi d'effectuer le parcours en largeur de l'arbre en parallèle. Ainsi pour chaque niveau, on calcule en parallèle l'ensemble des blocs de mots, la TBB gérant la répartition de ceux-ci sur les différents processeurs/cœurs.
Nous avons mesuré sur un fichier de 850 Mo, l'influence du nombre de processeurs/cœurs sur le débit de hachage. Voici les résultats sur la machine Idkoiff, les résultats sont sensiblement les mêmes sur la machine ensibm (voir la partie 5.1 pour les descriptions des machines).

Sur un bicœur, l'algorithme est deux fois plus rapide que sur un seul cœur.

Cependant, on constate qu'au delà de huit cœurs, l'augmentation des performances n'est plus linéaire par rapport au nombre de processeurs/cœurs.

Ceci semble dû à la saturation du bus mémoire de la machine.

L'algorithme est tout de même huit fois plus rapide en utilisant 16 processeurs/cœurs.

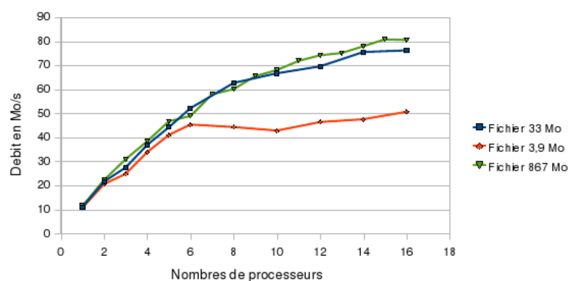

FIG. 5 – Débit de la version TBB en fonction du nombre de processeurs/cœurs sur trois fichiers de tailles différentes

Remarque : Pour utiliser la TBB, nous avons utilisé g++ 4.3.3, alors que pour toutes les autres implémentations nous utilisons gcc 4.3.3. Les versions compilées avec g++ sont presque 3 fois plus lentes pour des raisons qui nous échappent alors que le code et les options de compilations semblent identiques.

4 Implémentations sur GPU

Remarque : les débits présentés dans les parties suivantes ont été calculés à partir de la moyenne du débit de hachage de plusieurs gros fichiers de taille différente, et en prenant en compte toutes les copies mémoire, de façon à avoir un résultat proche de la réalité et de la durée d'exécution réellement ressentie par l'utilisateur.

4.1 Cuda

4.1.1 Présentation

Cuda (*Compute Unified Device Architecture*) est une technologie de GPGPU (*General-Purpose Computing on Graphics Processing Units*). Il permet d'écrire des programmes qui utilisent le processeur graphique (GPU) pour exécuter des calculs habituellement exécutés par le processeur central (CPU).

Cuda est en fait une extension du langage C et est donc facilement intégrable dans d'autre programme C ou C++.

Il existe aussi des extensions Cuda pour les langages Python, Fortran et Java.

Toutes les cartes graphiques NVIDIA récentes (GeForce série 8, les Teslas et certaines Quadro et les futures cartes) sont compatibles CUDA.

4.1.2 Particularités de programmation

Pour programmer de manière efficace sous CUDA, il est nécessaire de comprendre le fonctionnement des cartes graphiques NVIDIA.

Une très bonne documentation détaillée [4] est fournie à ce sujet par NVIDIA.

Les cartes graphiques sont composées de plusieurs multiprocesseurs, partageant une même mémoire partagée et pouvant fonctionner en parallèle. Ces multiprocesseurs sont composés de plusieurs dizaines de cœurs pouvant eux aussi fonctionner en parallèle.

De plus, la carte possède une mémoire globale à tout les processeurs, mais très lente d'accès, et une mémoire constante, lente mais qui peut être mise en cache.

Au niveau de l'API, le programme principal est le noyau, ou *kernel*. Ce noyau est appelée depuis une fonction exécutée par le processeur. Lors de l'appel on spécifie sur combien de blocs le noyau doit être exécuté, et combien de threads doit contenir chaque bloc.

Les blocs s'exécutent indépendamment les uns des autres, sur des multiprocesseurs différents.

Les threads d'un même bloc s'exécutent donc sur un même multiprocesseur, et peuvent partager des données (stockées dans la mémoire partagée). De plus, ils peuvent se synchroniser à l'aide de barrière de synchronisation.

Chaque thread possède une structure de données lui permettant de savoir son numéro et le numéro du bloc dans lequel il s'exécute, afin de permettre des traitements différents selon les threads et les blocs.

Le kernel peut appeler des fonctions auxiliaires, qui seront remplacées (*inline*) dans le code final.
Les appels récursifs ne sont pas permis, ainsi que les pointeurs de fonctions.

Les opérations complexes (allocation, copie mémoire, etc.) sont à effectuer par le processeur.

En conclusion, on constate que la façon de programmer sous CUDA est particulière, prévue pour le calcul parallèle, et que les performances dépendent du problème traité et de la façon dont la solution est implémentée.

4.2 Chaque bloc en parallèle

Dans la première version CUDA que nous avons implémenté, nous compressons les blocs en parallèle, de la même manière que dans l'implémentation de la TBB (voir 3.2). Chaque bloc de threads (au sens de

CUDA) est composé d'un thread qui compresse un bloc de mots (au sens de MD6).

Le débit moyen est alors de **2,7 Mo/s**, soit 10 plus lent que le version CPU...

Ceci s'explique par le fait que nous exploitons peu le parallélisme disponible.

4.3 Parallélisation de la fonction de compression

La compression d'un bloc de mots est composée d'étapes indépendantes, en particulier les 16 boucles exécutée lors d'un tour de compression, qui est la partie du code la plus souvent exécuté.

Nous avons parallélisé cette boucle, en espérant augmenter les performances de manière significative.

Afin d'avoir une version la plus parallèle possible, nous avons aussi parallélisé toute les étapes de l'algorithme qui pouvaient l'être, en particulier les copies mémoires dans les tableaux à la fin de chaque tour ou lors de la préparation du vecteur d'entrée de la fonction de compression.

Nous avons alors un débit de **28 Mo/s** (10 fois plus que la version précédente).

Cependant, cette version n'est pas plus rapide que la version CPU. Et de plus, pour les petits fichiers, le temps des copie mémoires vers la carte graphique n'est pas amorti.

Ceci semble dû au fait que l'on fait peu de calcul par rapport au nombre d'accès en mémoire effectués.

Afin de vérifier notre intuition, nous avons modifié MD6 de façon à faire 160 boucles au lieu des 16 initiales.

Ainsi, le temps d'exécution GPU de la version modifié est 10 fois plus rapide que la version CPU modifiée équivalente.

Pour augmenter les performances, nous avons donc besoin de limiter les accès mémoire et d'augmenter le nombre de calcul.

4.4 Augmentation du nombre de threads par bloc

Ne pouvant évidemment pas augmenter le nombre de calculs, nous avons essayé d'augmenter le nombre de threads par bloc. Chaque bloc de threads compresse alors plusieurs blocs de nœuds en parallèle.

Le problème est que la compression se fait en parcourant un tableau temporaire et en remplissant les valeurs au fur et à mesure, en fonction des 89 valeurs précédentes du tableau. Ainsi, on n'a jamais besoin de plus de 89 valeurs, mais on a un tableau de taille $89 \times N$, où N est le nombre de bloc de mots compressés dans un bloc de threads.

Avec plusieurs threads compressant plusieurs bloc de nœuds, il est nécessaire d'avoir plusieurs tableaux de cette taille. Mais on ne peut pas stocker plus d'un tableau de cette taille dans la mémoire partagée, et il n'est pas envisageable (pour des raisons de performances) de stocker ces tableaux dans la mémoire globale.

De plus, un tableau de taille trop grande réduit les performances, car chaque case est accédée une seule fois, et les systèmes de cache ne peuvent être utilisé.

Nous avons donc décider de modifier notre implémentation pour utiliser un tableau circulaire (plus petit). Ainsi, il devient possible d'augmenter de nombre de threads par bloc.

4.5 Utilisation d'un tableau circulaire

Le tableau est circulaire est un tableau de taille 89. Pour pouvoir le rendre circulaire, nous avons modifié notre implémentation afin que chaque calcul d'indice du tableau soit fait modulo 89.

Nous avions tout d'abord pensé utilisé un tableau circulaire de taille 128, afin de pouvoir effectuer des opérations logiques (avec masque de bits) pour calculer les indices, au lieu des coûteuses opérations arithmétiques que sont les modulo.

Malheureusement, des tableaux de cette taille sont trop gros pour pouvoir être stockés en cache, et les performances n'était pas bonnes.

Nous avons donc décidé des garder des tableaux circulaires de taille 89.

Nous utilisons 20×16 threads par bloc au lieu de 16 précédemment. C'est la limite au delà de laquelle on ne peut plus avoir de nouveaux tableaux circulaires en mémoire partagée.

Le débit obtenu est alors de **80 Mo/s**.

Néanmoins, le temps de calcul pourrait être réduit en tabulant les valeurs des indices modulo 89, pour éviter des calculs arithmétiques coûteux.

4.6 Tabulation des valeurs pour l'accès au tableau circulaire

On tabule toutes les valeurs modulo 89 nécessaires, c'est à dire celles qui pourront être utilisées (de 0 à $r \times$

89).

Le tableau ainsi créé est copié dans la mémoire constante de la carte graphique.

Ainsi, le coût de calcul d'un indice est remplacé par un accès au tableau. Sachant que les données de la mémoire constante peuvent être mise en cache et qu'un grand nombre de threads accèdent à ces données, le gain obtenu est important.

On obtient un débit de **105 Mo/s** sur cette version.

5 Évaluation des performances

5.1 Machines de test

Nous avons utilisé trois machines pour effectuer nos tests :

- Une machine personnelle avec un Intel Core2Duo E6420 @3.1 GHz et une 8800Gts, Linux 2.6.28 gcc 4.3.3
- Idkoiff, une machine de calcul de l'INRIA avec huit Dual core AMD Opteron @2.2Ghz, deux gtx280, Linux 2.6.26 gcc 4.3.3
- Ensibm un serveur de l'Ensimag composé de 16 IBM Power5+ @ 1.5Ghz Linux 2.6.9, g++ 3.4

Sur les deux premières machines nous avons testé toutes nos implémentations : CPU normal, CPU avec TBB et les différentes implémentations avec CUDA. Sur Ensibm, nous avons seulement tester nos versions pour CPU.

5.2 Tests

Pour chaque implémentation, nous avons mesuré les temps d'exécution avec ou sans allocation et copie mémoire, car le coût de ces opérations dépend énormément de la machine. En effet sur notre machine personnelle le débit de copie est de 2200 Mo/s contre 700 Mo/s pour Idkoiff.

Nous avons mesuré le débit de hachage pour le CPU et pour le GPU sur un ensemble de fichiers dont la taille varie entre 2 octets et la saturation en mémoire de la carte : 60Mo pour la 8800Gts contre plus de 350Mo pour la gtx280.

Sur notre machine personnelle, on peut également prendre en compte les coûts mémoires qui sont relativement faibles.

Les résultats obtenus sont présenté à la figure 6.

Les versions GPU sont bien plus rapide que la version CPU, et on peut noter que le coups des allocations et copies mémoires sont amorties avec la taille du fichier. Il y a environ un facteur 5 entre le débit maximal de hachage GPU et celui CPU : 150Mo/s pour le GPU contre 30 Mo/s pour le CPU. Les coûts mémoires supplémentaires dûs aux copies entre la mémoire centrale et la mémoire de la carte sont amortis avec la taille du fichier. Cependant nous n'avons pas pu tester avec de plus gros fichiers car nous étions limité par la capacité mémoire de la 8800Gts. Pour des fichiers de taille plus importante il faudrait couper le fichier en un multiple de quatre fichiers plus petits et appliquer notre algorithme sur ces fichiers plus petits.

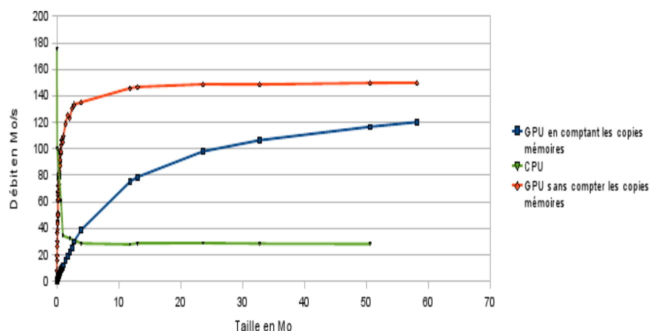

FIG. 6 – Comparaison des performances entre les versions CPU (vert), GPU avec copies mémoire (bleue) et sans (rouge)

Nous avons également effectué les mêmes tests sur Idkoiff qui possède des CPU plus puissants et des cartes graphiques plus puissantes (dans notre cas nous en utilisons une seule).

Cependant sur cette machine les coûts mémoires sont très élevés à causes d'un faible débit (surement dû à l'architecture de la machine).

Les résultats obtenus sont présenté à la figure 7

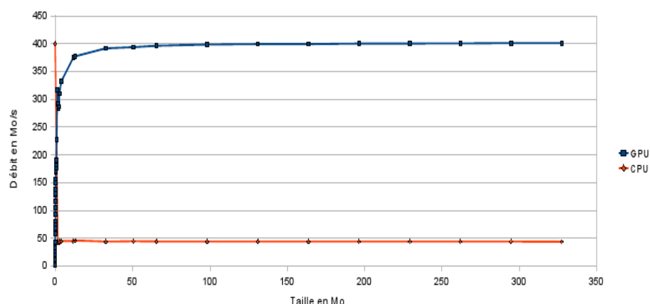

FIG. 7 – Comparaison versions GPU (bleue) et CPU non parallèle (rouge) sur Idkoiff

Nous atteignons donc un débit de hachage de 400 Mo/s sur des gros fichiers avec une gtx280 contre 42 Mo/s avec les opterons.

Le débit obtenu sur la gtx280 est presque 10 fois plus grand que celui des opterons, plus de 13 fois plus grand que celui de notre core2Duo et 2.5 fois plus grand que celui obtenu avec notre 8800Gts.

6 Conclusion

Une implémentation parallèle de MD6, compte tenu de la nature de l'algorithme, est possible sur GPU et sur plusieurs CPU.

Sur CPU, en utilisant la bibliothèque TBB, les performances augmentent de manière linéaire, tant que le nombre d'accès mémoire ne sature pas le bus de la machine.

De plus, peu de modifications sont nécessaires afin d'avoir une version utilisant la TBB.

Le développement pour CUDA s'avère plus délicat, en particulier car c'est une technologie récente et qu'elle nécessite de comprendre l'architecture des cartes graphiques NVIDIA, différente de celles habituellement rencontrée dans un CPU.

CUDA ayant un certain nombre de restrictions, les implémentations doivent être adaptées afin de les respecter.

Un certain nombre de critères doivent être pris en compte afin d'obtenir des performances intéressantes. En particulier, à cause des coûts importants des copies mémoires vers la carte graphique, il n'est pas intéressant d'utiliser une version CUDA pour des petits volumes de données.

Il est donc intéressant d'avoir plusieurs versions, et de choisir laquelle exécuter en fonction de la quantité de données à traiter (et donc du niveau de parallélisme disponible). Un algorithme adaptatif permettrait de tirer parti au mieux de la puissance de calcul disponible dans une machine.

Malgré ces inconvénients, une implémentation parallèle sur GPU reste plus efficace qu'une implémentation sur CPU (dès que la quantité de données à traiter est importante) malgré un temps de développement plus long.

Il est intéressant de constater que les GPU sont moins cher qu'un ensemble de processeurs, et permettent d'obtenir des performances supérieurs dès que le nombre de calcul à faire est important (dès que les fichiers à hacher sont assez gros, dans notre cas).

De plus, un GPU consomme moins d'énergie que 16 processeurs.

En ce qui concerne MD6, on constate que même si une implémentation CUDA ne tire pas autant parti du parallélisme que certaines applications hautement parallélisable effectuant beaucoup de calculs, le gain obtenu est très intéressant.

Ceci nous permet de conclure que MD6 tire parti de la puissance disponible dans les architectures modernes et futures, ce qui en fait une fonction de hachage moderne, et que la puissance de calcul des GPU, si on sait l'exploiter, permet d'avoir des versions très performantes d'algorithme parallélisables.

Références

[1] R. L. Rivest, "The md6 hash function, a proposal to nist for sha-3," Octobre 2008. http://groups.csail.mit.edu/cis/md6/submitted-2008-10-27/Supporting_Documentation/md6_report.pdf

[2] C. Y. Crutchfield, "Security proofs for the md6 hash function mode of operation," Thèse de Master, Massachusetts Institute of Technology, Juin 2008. http://groups.csail.mit.edu/cis/md6/docs/2008-06-crutchfield_ms_thesis.pdf

[3] *Intel Threading Building Blocks Reference Manual*, Intel Corporation, 2009. http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference%20(Open%20Source).pdf

[4] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, NVIDIA Corporation, Novembre 2007. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf