Cartes graphiques : calcul, cryptographie et sécurité

Antoine Joux

DGA et
Université de Versailles
PRISM
45 avenue des États-Unis, F-78035, Versailles CEDEX, France
antoine.joux(@)m4x.org

Résumé Afin de répondre aux besoins croissants des applications graphiques, les cartes graphiques connaissent actuellement une impressionnante augmentation de leur puissance de calcul. Une tendance récente appelée GPGPU (General-Purpose computation on Graphics Processing Units) consiste à détourner cette puissance afin de réaliser des calculs indépendamment de toute problématique d'affichage. Dans cet article, nous nous intéresserons aux applications possibles de cette technique aux algorithmes utilisés en cryptographie.

D'autre part, l'introduction de telles cartes graphiques dans un ordinateur modifie profondément la nature de son architecture interne et est susceptible d'y introduire de nouvelles vulnérabilités.

1 Introduction

Pendant longtemps, le calcul parallèle a été l'apanage des ordinateurs très haut de gamme. Cependant, ces dernières années, le parallélisme a fait son entrée dans les ordinateurs de bureau. Cette tendance prend plusieurs formes distinctes. Tout d'abord, ces ordinateurs ne contiennent plus un seul processeur (CPU) mais 2, 4 voire plus. De plus, chaque CPU dispose de capacité de parallélisme interne. Une autre voie est l'apparition au sein des ordinateurs d'unités de calcul indépendantes du processeur. Par exemple, on a pu voir des cartes à base de circuits reconfigurables (FPGA) à intégrer dans les machines de bureau, mais cela reste un marché de niche. Les cartes graphiques (GPU) sont une nouvelle incarnation de cette idée d'unités de calcul séparées du processeur. Elle présente un avantage économique indéniable, celui de s'appuyer sur un marché grand public en perpétuelle expansion, celui de l'image numérique et du jeu vidéo. De plus, sur le plan technique, ces cartes graphiques offrent une grande puissance de calcul pour un coût comparativement faible. La tentation est donc grande d'utiliser ces cartes pour toutes sortes de calculs. Cette possibilité est généralement connue sous le nom de GPGPU[1]. L'intérêt pour cette approche est

[1] General-Purpose computation on Graphics Processing Units

telle que certains constructeurs de cartes graphiques en font à la fois une priorité et un argument commercial. Un exemple particulièrement frappant est celui des cartes CUDA de Nvidia.

Dans cet article, nous nous intéresserons à l'application du GPGPU à la sécurité et à la cryptographie en nous appuyant sur l'architecture CUDA. Dans la section 2, nous présenterons quelques rappels sur l'architecture CUDA actuelle. Dans la section 3, nous illustrerons par quelques exemples les possibilités qu'offrent la puissance de calcul ainsi libérée. Finalement, dans la section 4, nous évoquerons les vulnérabilités que l'utilisation de ces cartes pourrait introduire.

2 Quelques notions sur CUDA

Le but de cette section étant simplement d'offrir un aperçu rapide de CUDA, le lecteur intéressé est invité pour des informations plus approfondies à se reporter à la page http://www.nvidia.com/cuda/. Cette page mentionne, en particulier, de très nombreuses applications de la technologie au calcul numérique. Grâce à CUDA, le développeur peut utiliser la puissance de calcul d'une carte graphique par l'intermédiaire d'une version adaptée du langage C. Ces adaptations permettent de refléter au niveau logiciel les spécificités de l'architecture interne des cartes graphiques utilisées et de gérer la communication entre le CPU et ces cartes graphiques. Elles conduisent aussi à certaines limitations du langage C. La plus notable de ces limitations est sans doute la suppression de la notion de récursivité. En effet, dans les codes de calcul exécutés via CUDA sur cartes graphiques, il n'est pas possible d'invoquer une fonction récursivement.

Autour de cette base de développement C, des extensions sont apparus qui permettent l'utilisation de CUDA avec des applications écrites dans d'autres langages. Par exemple, CUDA peut s'interfacer avec du code écrit en Fortran ou en Python.

2.1 Structure générale d'un programme

Toute application CUDA nécessite de coordonner le travail du CPU et du (ou des) GPU. Le code source d'une telle application consiste donc en deux parties, un code principal exécuté par le CPU et un code secondaire exécuté par le GPU sur demande du CPU. Au sein du code secondaire, il convient de distinguer deux fonctions très particulières, celles chargées de démarrer le code CUDA sur le GPU. La première de ces deux fonctions tourne sur le CPU, effectue les allocations mémoire sur le GPU, les transferts entre la mémoire du CPU et celle du GPU, puis invoque une exécution parallèle de la seconde fonction sur GPU. Après la fin de l'exécution parallèle, elle transfère les résultats vers le CPU et libère la mémoire du GPU.

La seconde de ces deux fonctions est invoquée de façon très particulière de manière parallèle. Ce parallélisme est à deux niveaux et il est décrit de manière géométrique. Le premier niveau de parallélisme permet de voir logiciellement le GPU comme une grille de calcul à une ou deux dimensions[2] formée de blocs de calcul. Tous les blocs sont exécutés par le GPU, parallèlement ou en séquence, en fonction des ressources de calcul disponibles. Cette exécution est asynchrone et, si nécessaire, un appel spécial permet de mettre les blocs en attente pour les synchroniser. Chaque bloc est lui même décomposé en une matrice à une, deux ou trois dimensions de processus légers (*threads*). Bien que cela n'apparaisse pas dans la description logicielle, l'exécution des processus légers d'un même bloc est regroupé par paquets appelés *warps* au sein[3] desquels des règles complexes d'exécution synchrone et d'accès à la mémoire entre en jeu. Nous n'entrerons pas ici dans les détails de ces mécanismes, mais leur prise en compte est essentielle pour réaliser des codes de calculs efficaces.

Après invocation de cette seconde fonction, l'environnement CUDA se retrouve en train d'exécuter, en parallèle, un grand nombre de *threads* répartis entre les blocs. Par l'intermédiaire d'un mécanisme offert par CUDA, chaque *thread* apprend les coordonnées du bloc qui le contient dans la grille, ainsi que ses propres coordonnées dans ce bloc. Ce sont ces coordonnées qui permettent à chaque *thread* de connaître la part de travail qui lui revient.

2.2 Organisation de la mémoire

L'un des avantages majeurs de l'architecture CUDA réside dans le fait que l'ensemble des processus légers exécutés sur le GPU disposent d'une mémoire de travail partagée. Ce modèle permet une gestion assez simple du parallélisme pour le programmeur. En revanche, au niveau du matériel, un tel choix est difficilement compatible avec des mécanismes de mémoire cache. Pour conséquent, cette mémoire partagée ne dispose pas, pour l'essentiel, de tels mécanismes, ce qui rend les accès à la mémoire relativement lents. En fait, il est possible de disposer de caches sur une partie de cette mémoire, à condition de la déclarer en lecture seule. Bien que pratique, cela n'est pas adapté à toutes les applications.

Pour compenser l'absence de caches, il existe un second niveau de mémoire partagée, commun à tous les processus légers d'un même *warp*. Cette mémoire est de petite taille, mais d'un accès très rapide. En fait, elle peut, en un certain sens, être vu comme un cache de premier niveau, géré manuellement par le programmeur. De

[2] Dans la version actuelle de CUDA, il est envisageable que dans le futur des grilles à trois dimensions deviennent possibles.

[3] Il existe aussi une division encore plus fine, par moitié de *warp*, mais, par souci de simplification, nous ignorerons ici cette distinction.

plus, au sein de chaque *warp*, les processus se partagent un très grand nombre de registres sur 32 bits. Notons que dans les cartes actuelles, la capacité de stockage de cette ensemble de registre est supérieure à celle de la mémoire partagée du *warp*.

3 Exemples d'applications

3.1 Mise en œuvre d'algorithmes de chiffrement

AES L'AES (*Advanced Encryption Standard*, voir [1]) est le standard actuel en ce qui concerne le chiffrement par bloc. Cet algorithme chiffre des blocs de données de 128 bits, à l'aide d'une clef de 128, 192 ou 256 bits. Compte-tenu de la flexibilité de CUDA, il est bien sûr possible d'implanter l'AES sur GPU afin d'en exécuter de nombreuses copies en parallèle. Pour un exemple de mise en œuvre, on pourra consulter [4].

Toutefois, lors du déploiement de l'AES sur GPU, deux problèmes d'importance émergent. Le premier est lié au choix du mode d'opératoire utilisé, le second au temps de transfert des données entre le CPU et le GPU. Concernant le premier problème, rappelons tout d'abord que l'utilisation d'un algorithme de chiffrement par bloc pour chiffrer des messages entiers s'assortit toujours de l'utilisation d'un mode opératoire. Le but de ce mode est à la fois de décomposer le message d'origine en blocs de la bonne taille et de garantir, à condition bien sûr que l'algorithme de chiffrement soit sûr, la sécurité du message dans son ensemble. En particulier, un mode d'opératoire sûr doit faire en sorte de cacher, s'il y en a, les répétitions du message de départ. Pour cette raison, l'idée naturelle consistant à simplement découper le message en blocs puis à chiffrer ces blocs indépendamment les uns des autres n'est pas suffisante. La difficulté qui apparaît alors est que l'un des modes opératoires les mieux connus et les plus utilisés, le CBC (*Cipher Block Chaining*) est par nature totalement séquentiel. Dans ce mode, on ne peut débuter le chiffrement d'un bloc qu'après avoir terminé le chiffrement du bloc précédent. Il reste possible de chiffrer simultanément plusieurs messages distincts, mais le mode CBC interdit l'utilisation de toute forme de parallélisme massif et ne se prête donc pas à la mise en œuvre sur GPU. Heureusement, il existe d'autres modes opératoires qui peuvent être utilisés à la place du CBC. Par exemple, le mode compteur ou mieux un mode parallélisable de chiffrement authentifié comme le mode IAPM proposé par Jutla dans [7].

Le temps de transfert des données entre le CPU et le GPU est un réel goulot d'étranglement. En effet, d'après [4], l'impact des transferts de données ralentit d'un facteur 2 environ la vitesse de chiffrement observée. À ce sujet, les auteurs observent que ce problème pourra être résolu dès lors que le CPU et le GPU partagerons le même espace mémoire.

Au final, en prenant en compte les transferts de données, l'accélération constatée dans [4] est d'un facteur 2 seulement par rapport à un CPU classique. D'autre part, avec l'apparition d'instructions spécialisées sur certains CPU permettant d'accélérer le chiffrement AES, il ressort que le chiffrement AES n'est pas un problème réellement adapté au traitement sur GPU.

RSA Le chiffrement RSA [9] est, sans doute, l'algorithme de chiffrement à clef publique le plus utilisé. Il se base sur l'utilisation de l'exponentiation modulo un grand nombre composé. De nombreuses méthodes ont été utilisées pour mettre en œuvre efficacement ce calcul sur des composants électroniques. Une contribution récente [5] montre que ces méthodes peuvent également permettre de réaliser le (dé)chiffrement RSA sur GPU.

Toutefois, il en ressort qu'une telle mise en œuvre n'est réellement efficace que pour traiter simultanément un grand nombre de calcul RSA avec la même clef. En termes d'usage, on ne peut donc envisager un gain que pour certaines applications spécifiques. Un exemple typique est celui d'un serveur web sécurisé devant déchiffrer un grand nombre de requêtes simultanément.

3.2 Utilisations pour la cryptanalyse

Recherche exhaustive ou semi-exhaustive Comme souvent lorsqu'il est question de calcul parallèle, les premières applications sont celles pour lesquelles le parallélisme est évident (souvent désignées par l'expression « *embarrasingly parallel* »). Dans le cas de la cryptanalyse, un grand nombre d'attaques de ce type existent et nous allons maintenant en décrire quelques unes.

Attaques par force brute Les attaques par force brute, dans lesquelles il suffit de tester massivement des clefs ou des mots de passe, se prêtent tout naturellement à une parallélisation massive. Un exemple frappant de cette approche est celui de la société Elcomsoft qui distribue des logiciels de récupération de mots de passe et propose, en particulier, d'utiliser des GPU pour accélérer cette récupération. Sur un GPU haut de gamme, l'accélération annoncée atteint un facteur de l'ordre de 30 par rapport à un CPU quadruple cœur. Pour plus de détails, on pourra se reporter à `http://www.elcomsoft.com/edpr.html`.

Pour rendre les attaques par forte brute plus efficaces, les compromis temps-mémoire sont fréquemment utilisé. Ce type d'attaque peut aussi être accéléré en utilisant une carte GPU. Ainsi, il est possible d'utiliser le compromis temps-mémoire de Hellman ou la méthode des tables arc-en-ciel (*rainbow tables*) présentée dans [8].

Factorisation par la méthode des courbes elliptiques La méthode des courbes elliptiques pour la factorisation [6] consiste pour factoriser un entier N à multiplier sur de nombreuses courbes elliptiques un point de base par un grand entier K. Lorsque l'on rencontre une courbe pour laquelle le nombre de points modulo l'un des facteurs p de N divise le grand entier K, on détecte alors le facteur p. Cette propriété arrive suffisamment souvent pour que cette méthode de factorisation soit efficace, surtout lorsque N contient des facteurs relativement petits.

Comme il s'agit de tester un grand nombre de courbes et comme de plus les tests nécessitent beaucoup de calculs et peu d'accès à la mémoire, cette méthode de factorisation peut utilement tirer parti du parallélisme des GPU. Un article récent [3] décrit une telle implantation. Cette implantation tire également profit de nouvelles formules dites d'Edwards permettant d'accélérer les calculs sur courbes elliptiques.

Autres utilisations Au-delà de ces utilisations dans lesquelles le parallélisme est évident, CUDA est encore peu utilisé en cryptanalyse. En effet, malgré le potentiel en termes de puissance de calcul, la technologie est récente et susceptible d'évolutions. De plus, le développement de codes complexes nécessite un investissement important. Parmi les utilisations possibles au-delà de la recherche exhaustive, les applications les plus évidentes sont celles qui s'appuient sur de l'algèbre linéaire. En effet, de telles applications ont déjà été développées avec succès dans le domaine du calcul scientifique. Toutefois, l'algèbre linéaire utilisée en cryptographie est le plus souvent réalisée sur des corps finis et non sur les réels ou les complexes. Les développements nécessaires sont donc spécifiques au domaine.

Parmi les algorithmes d'algèbre linéaire, deux grandes familles se distinguent, l'algèbre linéaire dense et l'algèbre linéaire creuse. L'algèbre linéaire dense est, par exemple, utilisée lors des calculs de bases de Gröbner ou de réduction de réseau, tous deux très utiles en cryptanalyse. L'algèbre linéaire creuse apparaît, en particulier, dans la dernière phase des algorithmes de calcul d'index, utilisés pour la factorisation et le calcul de logarithmes discrets.

A titre d'illustration, nous présentons en annexe un code élémentaire de multiplication de matrices booléennes utilisant CUDA. Ce code élémentaire est largement plus rapide qu'un code élémentaire sur CPU. Toutefois, la comparaison n'est pas forcément significative, en effet, sur CPU, ce type de multiplications peut être très largement amélioré[4] par l'utilisation de l'algorithme des quatre russes [2] et de celui de Strassen [10]. Notons que la mise en œuvre de ces deux méthodes sur GPU n'est pas évidente. En effet, l'algorithme des quatre russes est très gourmand en mémoire rapide et celui de Strassen fait appel à la récursivité.

[4] En particulier, implantation réalisée dans le logiciel SAGE est tout à fait remarquable.

4 Impact sur la sécurité

Au-delà des applications cryptographiques, les cartes CUDA ont également des conséquences sur la sécurité. Tout d'abord, la puissance de calcul disponible peut être utilisée au service de la sécurité. Ainsi, le GPU peut être utilisé pour accélérer la recherche de signatures pour des logiciels anti-virus ou de détection d'intrusion. Une telle application est, par exemple, décrite dans [11].

À l'opposé, l'utilisation de cartes graphiques complexes est susceptibles d'introduire des vulnérabilités. Un premier indice vient de la constatation que certaines erreurs de programmation sur GPU nécessitent un redémarrage complet de l'ordinateur hôte pour être corrigé. D'autre part, le cloisonnement mémoire entre processus au sein des cartes graphiques n'est pas aussi bien assuré qu'au sein des CPU. Toutefois, la structure du langage CUDA constitue un facteur de mitigation du risque, car les transferts mémoires entre le CPU et le GPU dans ce langage se font à l'initiative du CPU. En conséquence, la détection et l'exploitation de faille de sécurité dues à la présence d'un GPU nécessite vraisemblablement d'accéder aux fonctionnalités de bas-niveau de ces cartes et de leur bus hôte.

Sur le plan de la sécurité, le risque majeur envisageable dans les prochaines années est lié à la possible introduction d'un espace mémoire partagé entre le CPU et le GPU. En effet, comme nous l'avons déjà remarqué, cela permettrait d'accélérer notablement certaines applications et cette fonctionnalité pourrait donc apparaître. Dans ce cas de figure, il serait essentiel que le GPU dispose des mêmes mécanismes de contrôle des accès mémoire que le CPU ou, pour le moins, que les processus sensibles s'exécutent uniquement sur le CPU et dans un espace mémoire non accessible au GPU.

Références

1. Advanced encryption standard (aes). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.
2. V. L. Arlazarov, E. A. Dinic, M. A. Kronod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. *Soviet Math. Dokl.*, 11 :1209–1210, 1970.
3. Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 483–501. Springer, 2009.
4. Owen Harrison and John Waldron. Practical symmetric key cryptography on modern graphics hardware. In *17th USENIX Security Symposium*, LNCS, pages 195–209, 2008.
5. Owen Harrison and John Waldron. Public key cryptography on modern graphics hardware. Booklet of posters, Eurocrypt 2009, Cologne, April 2009.
6. H.W. Lenstra Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(2) :649–673, 1987.
7. Charanjit S. Jutla. Encryption modes with almost free message integrity. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 529–544. Springer, 2001.

8. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.
9. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2) :120–126, 1978.
10. Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13 :354–356, 1969.
11. G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S.Ioannidis. Gnort : High performance network intrusion detection using graphics processors. Available on http://www.nvidia.com/cuda/.

A Exemple de multiplication de matrices booléennes CUDA

c.f. http://actes.sstic.org/