Impact des schedulers sur la prédictibilité dans les GPU

David Defour

ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Impact des schedulers sur la prédictibilité dans les GPU

David Defour

Université de Perpignan Via Domitia,
Laboratoire DALI-LIRMM
52 avenue Paul Alduy
66860 Perpignan Cerdex - France
david.defour@univ-perp.fr

Résumé
Les GPU sont des architectures massivement multicoeurs gérant plusieurs dizaines de milliers de threads concurrents. Cette concurrence, maintenue par le biais de différents schedulers, est nécessaire pour conserver de bonne performance mais nuie à la prédictibilité. Dans ce travail nous proposons une mesure de la prédictibilité ainsi que les tests CUDA permettant de s'approcher de cette mesure. Enfin nous donnons des mesures de prédictibilité lié aux schedulers de warp et de block pour différentes architectures Nvidia allant du G80 au K10.

Mots-clés : scheduling, GPU, prédictibilité, WCET.

---

1. Introduction

L'exploitation efficace des architectures multicoeur modernes passe par une structuration hiérachique du calcul mais aussi par la concurrence d'exécution. Cette concurrence peut faire partie intégrante des spécifications du problème comme dans le cas de système réactif ou basé sur des événements (ex : serveur web). Elle peut être une des fonctionnalités nécessaires (ex : système distribué), ou tout simplement permettre un fonctionnement correct en bornant le temps d'exécution dans des programmes interactifs. Dans tous les cas, elle impacte le déterminisme et la reproductibilité des logiciels rendant leur développement plus complexe. Dans ce modèle, il est en effet difficile de s'assurer que le comportement d'une tache complexe se comporte correctement pour toutes les combinaisons possibles d'interaction entre ces taches. Ces interactions utilisent des hypothèses sur l'ordre d'exécution ou l'utilisation de mécanisme de synchronisation comme les verrous, les fonctions atomiques ou les barrières.
Le déterminisme est une des solutions répondant aux nombreux problèmes liés à la concurrence. On a ainsi pu observer l'arrivé de nombreux travaux liés à la reproductibilité, à la prédictibilité et au déterminisme. La *reproductibilité*, notion la plus forte, consiste à avoir un comportement identique pour au minimum un jeu de donnée et un programme identique, et ce potentiellement indépendamment du compilateur et du processeur. La reproductibilité totale est très difficile à atteindre. On parle alors plutôt de *déterminisme* qui consiste à observer un comportement identique pour un jeux de donnée et un programme identique. Le *déterminisme* diffère de la *reproductibilité* notamment vis à vis des mesures de temps. Ce déterminisme est souvent obtenu à l'aide de techniques matérielles couteuses. Enfin, la *prédictibilité* consiste sim-

plement à être capable de connaître à l'avance le résultat ou comportement d'un processeur pour une suite identique d'état et d'action.
Si la prédictibilité est une notion à l'origine de nombreux mécanismes en architecture des ordinateurs (prédicteur de branchement, d'adresse, de valeur, ...), elle est souvent manipulée sous sa forme binaire, à savoir est-on capable de prédire ou pas. Mais il peut s'avérer pertinent de considérer cette prédictibilité sous une version probabiliste, notamment pour ce qui est de la prédictibilité relative au comportement d'une architecture. Ces informations sont essentielles pour caractériser le comportement des processeurs pour la construction de simulateurs [13], estimer le WCET [23], définir l'espace des exécutions possibles ou à l'inverse être utile pour mesurer la pertinence de générateurs de nombres aléatoires basés sur le non-déterminisme dans l'accès à une ressource partagée [11].
Dans cet article nous proposons d'analyser l'impact des schedulers sur la prédictibilité des processeurs graphiques que nous présentons en section 2. Pour celà, nous proposons une mesure probabiliste de la prédictibilité en section 3 et présentons en section 4 les résultats obtenus sur des GPU Nvidia de différentes générations.

2. Présentation des processeurs graphiques

Dans cet article nous considérons des GPU Nvidia de différentes générations dont les caractéristiques sont décrites en Table 1. Ces coprocesseurs sont utilisés en support du CPU pour la réalisation de taches exhibant du parallélisme de données. Ces tâches sont divisées en thread opérant en mode SIMT sur des données et pris en charge par du matériel spécifique.

TABLE 1 – Description de GPU Nvidia de différentes générations.

Nom Commercial	Architecture	CUDA Capability	#MP/ GPC	#MP	CUDA Core / MP	Warp Scheduler / MP	GPU Clock (Mhz)	Memory Clock (Mhz)
C870	G80	1.0	2	16	8	1	1350	800
9800 GX	G92	1.1	2	16	8	1	1500	1000
GTX 480	GF100	2.0	4	15	32	2	1401	1848
GTX 560	GF114	2.1	4	7	48	2	1620	2004
GTX 680	K10	3.0	3	8	192	4	1059	3004

En terminologie CUDA, les GPU sont constitués de coeurs CUDA aussi appelé Streaming Processors organisés de façon hiérarchique. Ces coeurs CUDA sont regroupés en multiprocesseur (MP). Le nombre de coeur CUDA par multiprocesseur varie entre 8 et 64 en fonction de ce que l'on appel CUDA capability. De façon similaire les multiprocesseurs sont regroupés afin de former les *Grahics Processing Cluster* (GPC). Un niveau de regroupement additionnel des threads et transparent pour le développeur, est introduit au niveau matériel : les warp.

2.1. Sources d'indéterminisme

Le fait même que les GPU soient des coprocesseurs les isole de certaines sources d'indéterminisme comme les interruptions ou les changements de contexte que l'on retrouve dans le cas

des processeurs généralistes. Cependant, les futures générations pourraient être plus sensibles à ces sources, si l'architecture de celle-ci vennaient à se rapprocher des CPU.
Les GPU ont plusieurs domaines d'horloge, chacun fonctionnant à une fréquence optimale. Les circuits de synchronisation entre ces domaines peuvent introduire de l'indéterminisme lié aux délais induits par les changements de phases lorsque des messages sont échangés [28]. Le poids de cette source devrait augmenter dans les générations futures dans la mesure où les techniques de gestion dynamique de fréquence et de voltage (DVFS) devraient se généraliser à la granularité du multiprocesseur.
Une autre source d'indéterminisme est liée au temps d'accès à la mémoire hors puce car celui-ci est fonction de l'endroit physique (partition mémoire) où la donnée est placée [31]. Sur les GPU modernes, la probabilité que l'allocation et le placement mémoire soient identiques entre deux appels consécutifs est très faible. De plus, les variations dans les cellules DRAM invite à l'utilisation de techniques de rafraichissement dynamique introduisant des temps d'accès aux cellules DRAM variables [24].
Avec la miniaturisation des transistors l'apparition de fautes matérielles devient de plus en plus courantes. Aussi, ces fautes sont dans certains cas corrigées par des mécanismes introduisant des délais variables dans les latences d'accès mémoires [30].
Enfin l'utilisation de schedulers non initialisés introduit de l'indéterminisme en modifiant l'ordre d'exécution et potentiellement les assignations entre logiciel et matériel. Ce qui regroupe les schedulers de wavefront dans chaque unité de calcul, le distributeur de workgroup, et les arbitres de bus dans les réseaux d'interconnexion. Même si ces éléments sont initialisés lors de leur mise sous tensions, ils ne le sont pas entre chaque lancement de kernel. Aussi l'état de ces unités est donc dépendant des lancements de kernels précédents, et pouvent donc être considérés comme non prédictibles.

2.1.1. Les schedulers

Les caractéristiques des schedulers dépendent de la génération de GPU Nvidia considérée, correspondant à leur version de *compute capability* [26]. Cette classification va de 1.x à 3.x, pour les dernières architectures disponibles en 2013.
Les architectures correspondant à une version compute capabilitie 1.x, disposent d'un seul scheduler de warp capable de lancer une instruction entière ou flottante sur 4 cycles d'horloge.
Les architectures correspondant à une version compute capabilitie 2.x, disposent de deux schedulers de warp. Pour les architectures compute capability 2.0, chacun des deux schedulers peut lancer une instruction d'un warp, alors que les architectures compute capability 2.1, chaque scheduler peut lancer jusqu'à 2 instructions indépendantes. Dans ce cas, le premier scheduler se charge des warps avec un ID impair et le deuxième des warps avec un ID pair, sauf lorsqu'il s'agit d'une instruction flottante double précision. Un scheduler de warp, ne lance une instruction que sur la moitié des coeurs CUDA. Si une instruction non atomique exécutée par un warp écrit au même endroit en mémoire global, alors un seul thread du warp effectue l'écriture et le thread effectuant l'écriture est indéterminé.
Les architectures correspondant à une version compute capabilitie 3.x, disposent de quatre schedulers de warp par multiprocesseur. Lorsqu'un multiprocesseur a un warp à exécuter, il est premièrement distribué entre l'un des 4 schedulers de warp. Ensuite, à chaque lancement d'instruction, chaque scheduler lance deux instructions indépendantes pour le warp qui lui est assigné.

3. Mesure de la prédictibilité

La prédictibilité, dans un modèle d'exécution parallèle, correspond à la possibilité de prédire le comportement où les résultats d'une exécution. Or ces éléments ne sont pas quantitatifs mais plutôt qualitatifs. En effet, il est difficile de donner une mesure de distance d'un résultat ou comportement par rapport à une prédiction ; celle-ci est soit juste, soit fausse. Cependant, si l'on s'autorise de réaliser des observations sur plusieurs exécutions, il est alors possible de collecter des informations sur l'espace des résultats possibles associés à des probabilités d'occurrences.
La prédictibilité peut être associée à des observations très diverses. Ce peut être une observation sur le temps d'exécution, sur des résultats finaux où bien des états intermédiaires. Si l'observation du temps d'exécution est la plus simple à réaliser, elle est aussi la moins précise car elle se contente de caractériser le comportement global et ne permet donc pas de se focaliser sur un élément précis. Il en va de même pour l'observation du résultat final. Il est donc pertinent d'analyser les états intermédiaires.
Nous avons vue en section 2.1, qu'il existe plusieurs sources possibles d'indéterminisme susceptiblent d'altérer le résultat ou comportement observable pour l'exécution d'un programme sur un processeur graphique. Dans ce travail, nous nous concentrons sur le scheduling de block et de warp.
Une première solution pour déterminer l'ordre d'exécution des warps et blocks est de lire le registre d'horloge pour chaque warp exécuté. L'ordre d'exécution relatif est obtenu en comparant les mesures entre elles. Dans la suite de ce travail nous appellerons cette méthode *horloge*. Elle donne une information relativement précise sur le scheduling de warp à l'intérieur d'un MP. Cette mesure peut être sujet à critique pour l'analyse du scheduling de block puisque le registre d'horloge est propre à chaque MP. En effet, les GPU étant de gros processeurs, la compléxite de l'arbre d'horloge force chaque MP à disposer de sa propre horloge. Nous noterons cependant que même si les registres d'horloge sont indépendant, ils sont incrémentés avec la même fréquence. Aussi afin de compléter cette mesure, nous proposons d'utiliser une donné partagée en mémoire globale et de garantir l'unicité de l'accès à l'aide d'instructions atomiques. Nous ferons référence à cette solution dans la suite par solution *atomic*. L'inconvénient de cette mesure est qu'elle combine aussi les artéfacts liés aux schedulers de bus et l'indéterminisme liés aux accès mémoires.

3.1. Programmes de test

Nous avons réalisé une implémentation CUDA des tests précédemment proposés. Ces tests permettent de générer des vecteurs d'exécution qu'il est ensuite possible de comparer entre eux. Ces vecteurs se composent d'autant de valeurs que de threads/warps/blocks exécutés. Les valeurs définissent un ordre d'exécution correspondant soit à la date à laquelle les threads ont eu accès à l'horloge pour la méthode *horloge* (Listings 1), soit l'ordre dans lequel ils ont eu accès à la ressource partagée via l'instruction atomic pour la solution *atomic* (Listings 2).
A chaque exécution nous obtenons ainsi entre 1 et N vecteurs d'exécution. Le code est exécuté C fois. Nous avons ainsi N.C vecteurs d'exécution. Nous utilisons le mode statistique qui correspond à la fréquence d'apparition du vecteur le plus présent. Par exemple, si sur dix vecteurs d'exécution nous obtenons l'ensemble des observations suivant $\{x, y, z, w, x, x, y, x, z, t\}$ alors la prédictibilité est de 40% car le vecteur x apparait quatre fois sur les 10 observations. Cette mesure permet de définir le comportement le plus probable, et par exemple de déterminer le bien fondé de générateur pseudo-aléatoire basé sur ce type d'indéterminisme [11].
Une autre mesure est celle de la cardinalité de l'ensemble des observations. Chaque vecteur est constitué de l'ordre dans lequel l'élément mesuré a eu accès à l'élément partagé (horloge

ou compteur global). Ainsi, si chaque vecteur est composé de p éléments, nous avons p! arrangements possibles. Il suffit alors de compter le nombre de vecteurs différents sur les N.C vecteurs d'exécutions et de le comparer aux p! arrangements possibles. Cette mesure nous permet d'analyser le nombre de scheduling possibles et ainsi déterminer la viabilité de solutions de type record-and-replay, ou tests de couverture pour la détection de bug.

```
1 __global__ void TestOrderHologe(int* dMem) {
2         const unsigned int gid =  blockDim.x * blockIdx.x + threadIdx.x;
3
4         dMem[gid] = clock();
5 }
```

Listing 1 – Test de l'ordre d'exécution en utilisant l'horloge

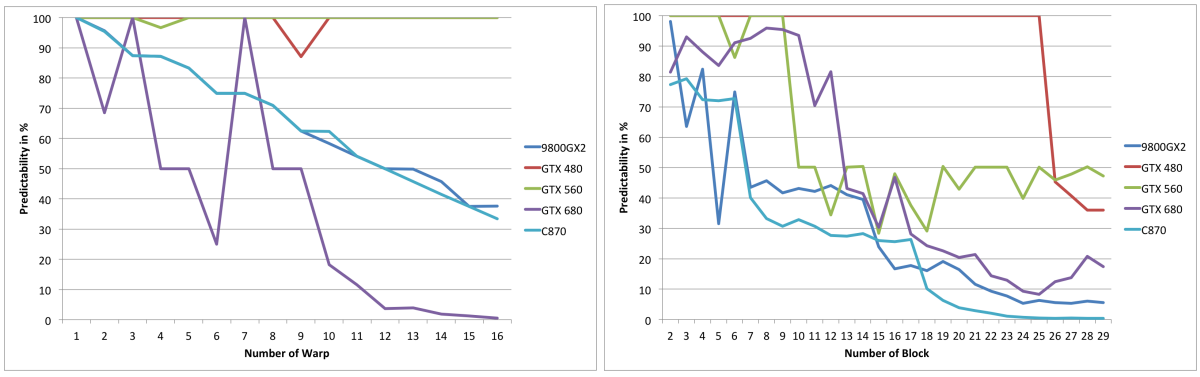
```
1 __global__ void TestOrderAtomic(int* dMem, int *v) {
2         const unsigned int gid =  blockDim.x * blockIdx.x + threadIdx.x;
3         int order;
4         order = atomicAdd(v, 1);
5         dMem[gid] = order;
6 }
```

Listing 2 – Test de l'ordre d'exécution en utilisant une valeur globale

4. Résultats

Nous avons testé des cartes graphiques de différentes générations décrites en Section 2 afin d'évaluer la prédictibilité des schedulers de block et de warp à l'aide des deux tests présentés en section 3.1. Nous analysons dans la Section 4.1 les données issues du mode statistique et dans la Section 4.2 celles issues de la mesure de la cardinalité de l'ensemble.

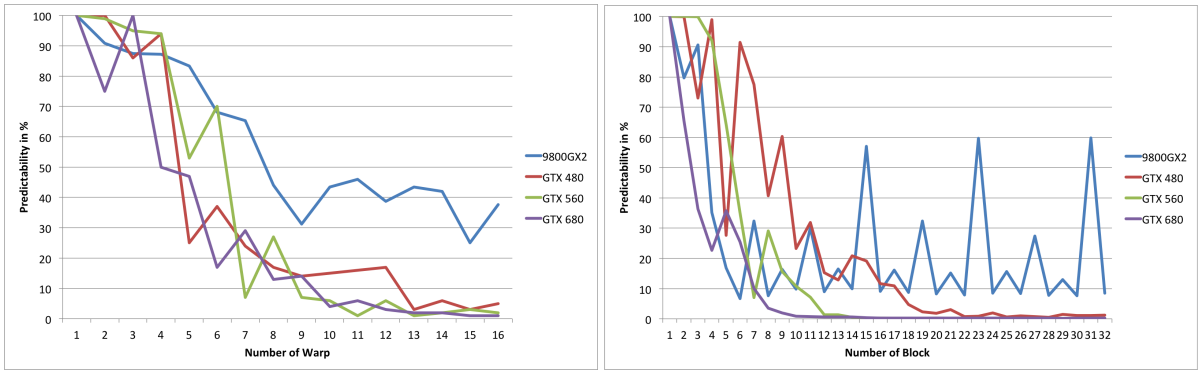
4.1. Mode statistique



(a) warp (b) block

FIGURE 1 – Mesure du mode statistique basé horloge

Les figures 1 et 2 présentent la prédictibilité en utilisant le mode statistique pour respectivement les méthodes *horloge* et *atomic*. Pour les mesures présentées dans les figures 1b et 2b, les blocks étaient composés d'un seul warp afin de se concentrer sur le scheduling de block. Nous avons



(a) warp (b) block

FIGURE 2 – Mesure du mode statistique basé atomic

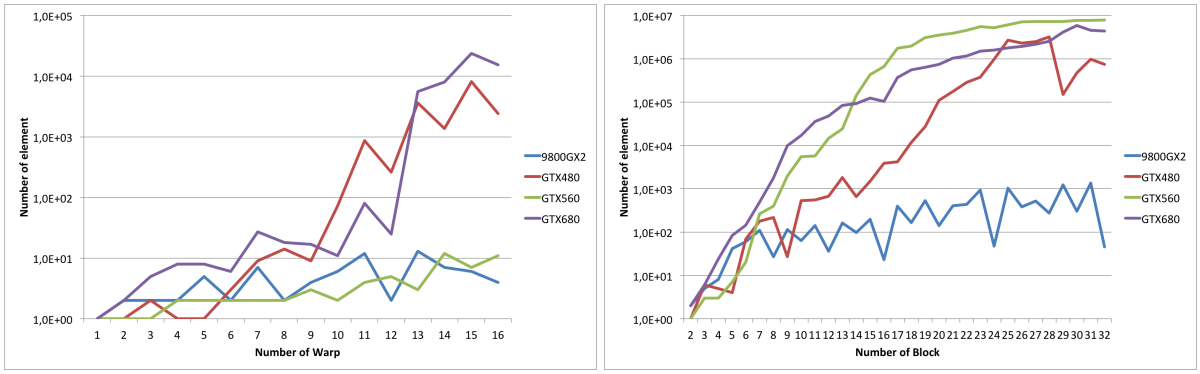
fait varier le nombre de block de 1 à 32 sur chaque architecture. Pour les mesures présentées dans les figures 1a et 2a, le nombre de block lancé correspondait au nombre de multiprocesseur disponible sur l'architecture testée afin que chaque multiprocesseur schedule uniquement les warps d'un seul block et pour un nombre de thread variant entre 1 et le nombre maximum de thread par block admissible. Les tests ont été lancés 1000 fois sans reboot de la machine, ni réinitialisation.

Nous constatons sur le graphique 1a que le scheduling de warp est très prédictible pour la GTX480 et la GTX560, puisque nous obtenons la même séquence sur l'ensemble des 1000 tests. Les schedulers de warp de la C870 et des 9800GTX ont un comportement identique avec une prédictibilité qui décroit de manière linéaire par rapport au nombre de warp lancé. Ceci s'explique par une configuration des MP identique entre les deux architectures (Tableau 1). La GTX680 a un scheduling de warp comparativement plus chaotique qui s'explique par sa gestion dynamique de la fréquence et du voltage. Les courbes de ce graphique nous montre que le scheduling de warp est fortement dépendant de la version de compute capability et que les architectures les plus prédictibles sont les architectures avec une compute capability 2.x.

Ce comportement est sensiblement différent lorsque les warps essaient d'accéder à une ressource partagée en mémoire globale (Figure 2a). En effet, nous constatons sur ce graphique que les comportements des GTX480, GTX560 et GTX680 sont identiques et moins prédictibles dans l'accès atomique à une ressource partagée que pour l'accès au compteur d'horloge. Par exemple, la prédictibilité est inférieure à 10% à partir de 13 warps lancés dans ces cas. A l'inverse, la 9800 GX2 continue d'offrir un comportement prédictible à plus de 40% et ce jusqu'au maximum de 16 warps lancés par multiprocesseur.

La figure 1b nous renseigne sur l'ordre dans lequel les block ont eu accès au registre d'horloge. Même si ce registre n'est pas le même entre les block, il est mis à jour avec la même périodicité puisque les blocks ont la même fréquence de fonctionnement. Nous y observons que le comportement du scheduling de block relativement au test d'accès à l'horloge est différent pour les 2 architectures compute capability 1.x. Cette différence avait été précédemment observée dans [14] en utilisant des mesures de consommation. Les GTX480 et GTX560 restent les deux architectures pour lesquelles la prédictibilité est la plus élevée, égale à 100% jusqu'à 25 blocks lancées pour la GTX480 et proche de 50% pour la GTX560.

Le graphique 2b présente les tests *atomic* de scheduling de block. Nous constatons que pour l'ensemble des architectures testées, la prédictibilité devient inférieure à 20% à partir de 12



(a) warp (b) block

FIGURE 3 – Mesure de la cardinalité de l'ensemble des vecteurs observés

blocks lancés. Encore une fois, le scheduling de block de la 9800 GX2 exhibe une prédictibilité en dent de scie caractéristique, fonction du nombre de block lancé. Ainsi sur cette architecture, un nombre de block multiple de huit plus le nombre de multiprocesseur moins un est la configuration qui exhibe la prédictibilité la plus élevée (proche de 60%).

4.2. Cardinalité

Afin d'évaluer le comportement des schedulings nous avons collecter 10^7 vecteurs d'exécution que ce soit pour le scheduling de block ou celui des warps.
La Figure 3b présente sur une échelle logarithmique la cardinalité de l'ensemble des vecteurs observés pour un nombre de block compris entre 2 et 32. Les vecteurs correspondent à une mesure basée sur le test atomic avec un seul warp par block de façon à se concentrer sur le scheduling de block. Nous pouvons observer que le nombre de scheduling observé pour la 9800GX2 est inférieure de 3 ordres de grandeur par rapport aux autres cartes plus récentes. Nous pouvons encore une fois constater le même comportement en dent de scie présent sur la Figure 2b. On remarque que vient ensuite la GTX480 dont le nombre de configuration observée est un ordre de grandeur de moins que la GTX560 et la GTX680. Il est à noter que l'ensemble des combinaisons possibles pour un nombre de block b est de b! ; soit 2.10^{35} pour $b = 32$, 2.10^{18} pour $b = 20$ et 2.10^{6} pour $b = 10$. Aussi, même si la cardinalité de l'ensemble des vecteurs observés est importante, nous constatons que le nombre de configuration reste bien inférieure à l'ensemble des combinaisons possibles.
La Figure 3a qui présente sur une échelle logarithmique la cardinalité de l'ensemble des vecteurs observés pour un nombre de warp compris entre 1 et 16. Les vecteurs correspondent à une mesure basée sur le test horloge. Nous pouvons encore une fois remarquer que le nombre de combinaison observée est très faible concernant le scheduling de warp et est conforme avec les mesures réalisées pour le mode statistique (Figure 1a).

5. Travaux précédent

De nombreux travaux liés au déterminisme ont accompagné le développement des architectures multicoeurs. Le problème fondamental permettant de garantir le déterminisme de programmes parallèles est de permettre au système d'appréhender et de vérifier les interactions en mémoire partagée entre les différentes instances du programme parallèle. Les solutions proposées impliquent au moins l'un des 4 éléments nécessaires à l'exécution d'un programme

parallèle : le langage, le compilateur, le logiciel ou le matériel.
Certains travaux [7] se sont focalisés sur les langages de programmation qui permettent à un programme d'être déterministe par construction. Par exemple, le langage SHIM [17] permet aux programmes développés en passage de message d'avoir un comportement déterministe. Cette même propriété est rendu disponible pour la classe des langages fonctionnels comme avec NESL [6] ou Data Parallel Haskell [19].
Les approches reposant sur le compilateur pour garantir le déterminisme, transforment un programme séquentiel (avec ou sans annotation) dans une version parallèle ou le déterminisme est préservé [10, 2]. Ces solutions sont à rapprocher des solutions logicielles qui testent, à l'exécution de programme parallèle, la violation du déterminisme [1, 29].
La solution basée matérielle offre une exécution déterministe, pour un surcoût minime en terme de temps d'exécution, au prix d'une augmentation de la complexité matérielle. Ces solutions peuvent réduire les possibilités de scheduling afin que le scheduling observable reste identique [3, 4, 25]. Il est communément admis que ce type de solution simplifie le test, le débogage, la réplication et le record-replay de programmes multithreadés.
Certains pense que le déterminisme peut se révéler couteux, et qu'il n'est ni suffisant, ni nécessaire pour obtenir une exécution fiable. Il n'est pas suffisant car sa définition est très restrictive ("donnée d'entrée + programme identique = comportement identique") et qu'il ne permet pas de considérer l'impact de modifications mineures. Par exemple, un système parfaitement déterministe peut assigner chaque donnée à un scheduler arbitrairement, de sorte qu'une modification d'un bit dans les données d'entrées ou l'ajout d'une ligne de débogage pourrait conduire à des schedulings très différents, réduisant artificiellement la fiabilité du programme confronté à des perturbations. Des solutions basées sur un déterministe [15] faible ont alors été proposées.
Le problème de l'indétermine au sein d'exécution sur GPU a lui aussi été abordé sous l'angle logiciel et matériel. Premièrement, les GPU intègrent nativement un certain nombre de mécanismes de synchronisation (inter ou intra-SM). Ces mécanismes permettent de limiter l'indéterminisme dans l'accès à une ressource partagée, mais ceux-ci peuvent se révéler très couteux [12, 32]. Diverses extensions ont été proposées afin d'améliorer le déterminisme. Par exemple, Timegraph [21] considère la problématique du scheduling de tache sur GPU dans un environnement multitâche. KiloTM [18] utilise les mémoires transactionnelles, tandis que Boyer et Skadron [9] proposent un émulateur de GPU pour détecter les "race condition" et les conflits de banc dans les programmes CUDA. PUG [22] et GPUVerify [5] se focalisent aussi sur la détection de "race condition" par une analyse statique. GPUDet [8] propose d'utiliser les spécificités des GPU comme le z-buffer et l'exécution SIMT des GPU pour supprimer l'indéterminisme. Enfin, une méthode pour l'évaluation du déterminisme basée sur le calcul d'une signature par le biais d'une fonction de hachage est proposée dans [20].
Alors que ces travaux se sont focalisés sur l'amélioration du déterminisme des GPU, certains ont tenté de mieux comprendre le comportement d'architecture GPU en utilisant la technique du microbenchmarking [27]. Mais à notre connaissance aucun d'entre eux ne se sont penché sur la problématique de la mesure de la prédictibilité liées aux schedulers des GPU.

6. Conclusion

De nombreux paramètres impactent la prédictibilité des processeurs. Dans cet article, nous nous sommes concentré sur les schedulers de block et de thread des GPU Nvidia et tenté de mesurer leur contribution à l'indéterminisme observable. Nous avons proposé de construire des vecteurs d'exécution exhibant les différences et non basé sur un simple hash. Ceci nous permettra dans de futurs travaux, d'analyser les corrélations entre les exécutions. A l'aide du

mode statistique et de la cardinalité de l'ensemble des scheduling observables, nous avons montrer les différences entre les architectures mais aussi entre le scheduling de block et de threads et caractérisé les configurations exhibant la pus grande prédictibilité.
Comme nous l'avons évoqué précédemment, il existe de nombreux facteurs impactant la prédictibilité et nous n'en avons exploré que deux. Le travail reste donc important sur ce point. Mais au delà de la simple mesure, certes utile, la suite logique est d'analyser les possibilités d'altérer cette prédictibilité. En effet, nous avons remarqué que certaines architectures exhibe un mode statistique important tout en ayant une cardinalité élevée comme c'est le cas pour le scheduler de warp de la GTX480. Aussi, pour ce type d'architecture il pourrait être utile d'augmenter artificiellement la probabilité d'apparition de vecteur de scheduling considérés comme peu probables afin d'accélérer les tests de couvertures, ou améliorer les générateurs de nombres aléatoires. Pour y parvenir, on peut envisager une altération logicielle du mécanisme de scheduling semblable à ce qui a été réalisé pour éviter le scheduling sur les unités défectueuses [16].
A l'inverse, si l'on souhaite limiter l'indéterminisme pour rejouer des jeux de test, faciliter le calcul du WCET ou la compréhension d'erreur de type Heisenbug, il faut trouver des moyens de limiter les comportements possibles pour permettre l'observation d'un comportement donné et prédictible. Il reste donc à montrer comment certains facteurs tels que la fréquence de fonctionnement, l'initialisation du coprocesseur ou la synchronisation impactent la prédictibilité et à en évaluer le coût.

Bibliographie

1. Allen (M. D.), Sridharan (S.) et Sohi (G. S.). – Serialization sets : a dynamic dependence-based parallel execution model. *In : Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (14th PPOPP'09)*. pp. 85–96. – Raleigh, NC, USA, février 2009.
2. Allen (R.) et Kennedy (K.). – *Optimizing Compilers for Modern Architectures : A Dependence-Based Approach*. – Morgan Kaufmann, 2002.
3. Aviram (A.), Weng (S.-C.), Hu (S.) et Ford (B.). – Efficient system-enforced deterministic parallelism. *Commun. ACM*, vol. 55, n5, 2012, pp. 111–119.
4. Berger (E. D.), Yang (T.), Liu (T.) et Novark (G.). – Grace : safe multithreaded programming for C/C++. *ACM SIGPLAN Notices*, vol. 44, n10, octobre 2009, pp. 81–96.
5. Betts (A.), Chong (N.), Donaldson (A.), Qadeer (S.) et Thomson (P.). – GPUVerify : a verifier for GPU kernels. *ACM SIGPLAN Notices*, vol. 47, n10, octobre 2012, pp. 113–132.
6. Blelloch (G. E.). – NESL. *In : Encyclopedia of Parallel Computing*, éd. par Padua (D. A.), pp. 1278–1283. – Springer, 2011.
7. Bocchino (R.), Adve (V.), Adve (S.) et Snir (M.). – Parallel programming must be deterministic by default. *In : Proc. HotPar '09 (1st USENIX Workship on Hot Topics in Parallelism), USB Data Stick*. – Berkeley, CA, mars 2009. UIUC.
8. Bond (M.). – GPUDet : a deterministic GPU architecture. *ACM SIGPLAN Notices*, vol. 48, n 4, avril 2013, pp. 1–12.
9. Boyer (M.), Skadron (K.) et Weimer (W.). – Automated Dynamic Analysis of CUDA Programs.
10. Bridges (M. J.), Vachharajani (N.), Zhang (Y.), Jablin (T. B.) et August (D. I.). – Revisiting the sequential programming model for the multicore era. *IEEE Micro*, vol. 28, n1, 2008, pp. 12–20.
11. Chan (J.), Sharma (B.), Lv (J.), Thomas (G.), Thulasiram (R.) et Thulasiraman (P.). – True

random number generator using gpus and histogram equalization techniques. *In : High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pp. 161–170.

12. chun Feng (W.) et Xiao (S.). – To gpu synchronize or not gpu synchronize ? *In : Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 3801–3804.
13. Collange (S.), Daumas (M.), Defour (D.) et Parello (D.). – Barra : A parallel functional simulator for gpgpu. *In : Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pp. 351–360.
14. Collange (S.), Defour (D.) et Tisserand (A.). – Power consumption of GPUs from a software perspective. *In : Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, éd. par Allen (G.), Nabrzyski (J.), Seidel (E.), van Albada (G. D.), Dongarra (J.) et Sloot (P. M. A.). pp. 914–923. – Springer.
15. Cui (H.), Simsa (J.), Lin (Y.-H.), Li (H.), Blum (B.), Xu (X.), Yang (J.), Gibson (G. A.) et Bryant (R. E.). – Parrot : a practical runtime for deterministic, stable, and reliable threads. *In : ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, éd. par Kaminsky (M.) et Dahlin (M.). pp. 388–405. – ACM.
16. Defour (D.) et Petit (E.). – Gpuburn : A system to test and mitigate gpu hardware failures. *In : ICSAMOS*, pp. 263–270.
17. Edwards (S. A.) et Tardieu (O.). – SHIM : a deterministic model for heterogeneous embedded systems. *IEEE Trans. VLSI Syst*, vol. 14, n8, 2006, pp. 854–867.
18. Fung (W.), Singh (I.), Brownsword (A.) et Aamodt (T.). – Kilo tm : Hardware transactional memory for gpu architectures. *Micro, IEEE*, vol. 32, n3, 2012, pp. 7–16.
19. Hill (J. M. D.). – *Data Parallel Haskell : Mixing old and new glue*. – Rapport technique n611, QMW CS, décembre 1992.
20. Hill (M. D.) et Xu (M.). – Racey : A stress test for deterministic execution. – 2009.
21. Kato (S.), Lakshmanan (K.), Rajkumar (R.) et Ishikawa (Y.). – Timegraph : Gpu scheduling for real-time multi-tasking environments. *In : Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. pp. 2–2. – Berkeley, CA, USA, 2011.
22. Li (G.) et Gopalakrishnan (G.). – Scalable SMT-based verification of GPU kernel functions. *In : Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, éd. par Roman (G.-C.) et Sullivan (K. J.). pp. 187–196. – ACM.
23. Lisper (B.). – Towards parallel programming models for predictability. *In : 12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, éd. par Vardanega (T.). pp. 48–58. – Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
24. Liu (J.), Jaiyen (B.), Veras (R.) et Mutlu (O.). – RAIDR : Retention-aware intelligent DRAM refresh. *In : ISCA*. pp. 1–12. – IEEE.
25. Liu (T.), Curtsinger (C.) et Berger (E. D.). – Dthreads : Efficient deterministic multithreading. *In : Proceedings of the 23rd ACM Symposium on Operating Systems Principles (23rd SOSP'11)*. pp. 327–336. – Cascais, Portugal, octobre 2011.
26. NVIDIA Corporation. – *NVIDIA CUDA C Programming Guide*, June 2011.
27. Papadopoulou (M.), Sadooghi-Alvandi (M.) et Wong (H.). – *Micro-benchmarking the GT200 GPU*. – Rapport technique, Computer Group, ECE, University of Toronto, 2009.
28. Sarangi (S. R.), Greskamp (B.) et Torrellas (J.). – CADRE : Cycle-accurate deterministic replay for hardware debugging. *In : DSN*. pp. 301–312. – IEEE Computer Society.
29. Welc (A.), Jagannathan (S.) et Hosking (A.). – Safe futures for Java. *ACM SIGPLAN Notices*, vol. 40, n10, octobre 2005, pp. 439–453.
30. Wittenbrink (C. M.), Kilgariff (E.) et Prabhu (A.). – Fermi GF100 GPU architecture. *IEEE*

Micro, vol. 31, n2, mars/avril 2011, pp. 50–59.

31. Wong (H.), Papadopoulou (M.-M.), Sadooghi-Alvandi (M.) et Moshovos (A.). – Demystifying gpu microarchitecture through microbenchmarking. *In : ISPASS*.
32. Xiao (S.) et chun Feng (W.). – Inter-block GPU communication via fast barrier synchronization. *In : IPDPS*. pp. 1–12. – IEEE.