Synthèse article ADA

Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow

Olivier Pelgrin

28 octobre 2017

1 Introduction

Ces dernières décennies, le développement de jeux vidéos graphiquement complexes en 3-dimensions a conduit à l'élaboration de processeurs spécialisés dans le calcul d'images en 3D.
En effet pour répondre à la complexité croissante de graphismes des jeux vidéos, le développement d'accélérateurs spécialisés a été nécessaire pour offrir le niveau de performance requis.
Ces processeur spécialisés sont appelés GPUs (Graphics Processing Units).
Au fil des années, et pour répondre aux nouveaux besoins des jeux vidéos, les GPUs sont devenus de plus en plus flexibles, devenant programmables, remplaçant une large partie de leur fonctions fixes par des programmes *shaders* écrits par les développeurs d'applications.
Depuis plus d'une dizaine d'année, les GPUs sont devenus *turing-complet* ouvrant la voie à de nouvelles utilisations, autres que le jeux vidéo. Les GPUs sont depuis devenus très utilisée pour le calcul scientifique grâce à l'apparition d'APIs de calcul généralistes (comme CUDA et OpenCL par exemple), permettant la programmation des GPUs en dehors du contexte des jeux vidéos.

2 Contexte et problématique

2.1 Architecture des GPUs

A l'origine, les tâches pour lesquels les GPUs ont été créés sont hautement parallélisable par nature (opérations par pixel/vertex), ce qui fait que les GPUs sont des architectures très parallèles.
Pour réduire la complexité des flux de contrôle, la plupart des architectures GPUs utilisent des pipelines single-instruction multiple-data (SIMD). Cela permet un haut niveau de parallélisme pour un coût relativement faible.

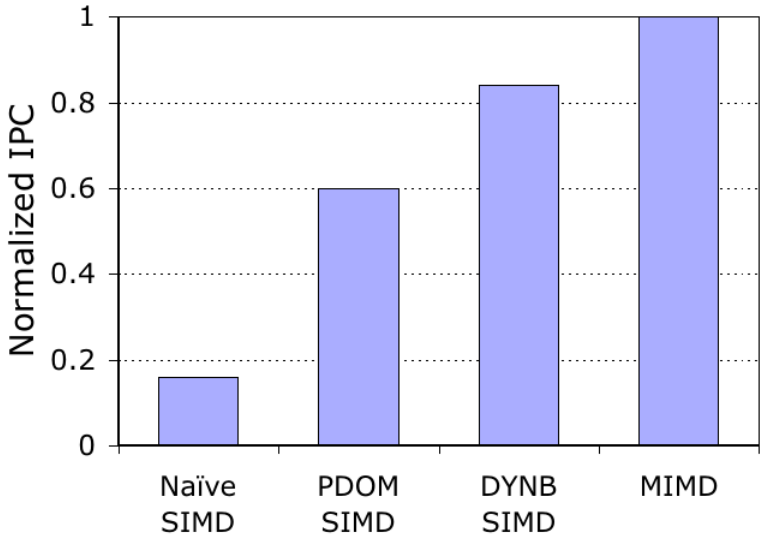
Le plus souvent, les GPUs groupent ensemble des threads exécutant le même programme dans des *warps*. Tous les threads d'un warp s'exécutent de manière synchronisée sur une ligne SIMD du GPU. Tous les threads exécutent exactement la même instruction en même temps, mais sur des données différentes.

2.2 Problématique des flux de contrôle

Pendant longtemps, cette approche a eu de très bon résultats sur les *shaders* des jeux vidéos. En effet, historiquement, ces shaders sont pauvres en opérations de contrôle, ce qui permet aux warps de s'exécuter du début à la fin de manière synchronisée.
Lors d'opérations de contrôle, il est possible que les threads d'un warp ne peuvent s'exécuter de manière synchronisée, par exemple si des threads prennent une branche différente lors d'un branchement. Dans ce cas, il faut exécuter chaque partie du branchement séparément, ce qui peut amener à une sous utilisation de la ligne SIMD, avec seulement un faible nombre de threads s'exécutant simultanément. Sur la figure 1, on peut observer la perte de performance due aux branchements à l'exécution de threads SPMD sur des architectures SIMD.

La divergence de branchements au sein d'un warp était à l'époque de la publication de l'article (2007) encore assez peu problématique dans le cadre de shaders de jeu vidéo (ce qui est moins vrai maintenant en 2017). Cependant les applications de calcul généralistes sur GPUs souffraient déjà de ce problème.
Les auteurs de l'article proposent plusieurs mécanismes permettant de réduire significativement l'impact des branchements divergeant sur les architectures SIMD des GPUs.

FIGURE 1 – Perte de performance lié aux branchements sur des architectures SIMD



3 Solutions proposées

La façon naïve de gérer la divergence de branchements au sein d'un warp consiste à sérialiser l'exécution des threads. Cette méthode est facile à implémenter et permet de gérer la divergence. Cependant, il est possible que les threads continuent de diverger lors de branchements divergeant successifs. Il ainsi possible de se retrouver dans une situation où tous les threads d'un warp sont isolés des autres, et s'exécutent les uns après les autres la ligne SIMD. De manière évidente, une telle solution est inacceptable sur le plan des performances, ce montre le besoin de solutions plus efficaces pour gérer les branchements.

Dans leur article, les auteurs font plusieurs propositions, la première est un mécanisme de reconvergence des threads dès que possible après un (des) branchements divergeant.

3.1 Reconvergence

Cette première solution consiste à faire reconverger dès que possible des threads ayant divergés pour les re-synchroniser lorsqu'ils se retrouvent à nouveau sur le même flux de contrôle.

4 Résultats

graphs