Throughput Oriented Analytical Models for Performance Estimation on Programmable Accelerators

Junjie Lai

ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Junjie LAI

préparée à l'unité de recherche INRIA – Bretagne Atlantique
Institut National de Recherche en Informatique et Automatique
Composante Universitaire (ISTIC)

Throughput-Oriented Analytical Models for Performance Estimation on Programmable Hardware Accelerators

Thèse sera soutenue à Rennes le 15 Février 2013

devant le jury composé de :

Denis Barthou
Professeur, Université de Bordeaux / *Rapporteur*

Bernard Goossens
Professeur, Université de Perpignan / *Rapporteur*

Gilbert Grosdidier
Directeur de Recherches CNRS, LAL, Orsay / *Examinateur*

Dominique Lavenier
Directeur de Recherches CNRS, IRISA, Rennes / *Examinateur*

Isabelle Puaut
Professeur Université de Rennes I / *Examinatrice*

Amirali Banisiadi
Professor University of Victoria, Canada / *Examinateur*

André Seznec
Directeur de Recherches INRIA, IRISA/INRIA Rennes / *Directeur de thèse*

What you do not wish for yourself,
do not do to others.
by Confucius

Remerciements

I want to thank all the jury members to give me this opportunity to defend my thesis.

I want to thank my colleagues and also friends of the current and previous team members, who make my 3-years' work in the ALF team a very pleasant experience.

I want to thank my wife and my parents for their constant support in my life.

Specially, I want to thank my supervisor for his guidance and kind help through my work.

Contents

Résumé en Français

L'ère du multi-cœur est arrivée. Les fournisseurs continuent d'ajouter des cœurs aux puces et avec davantage de cœurs, les consommateurs sont persuadés de transformer leurs ordinateurs en plateformes. Cependant, très peu d'applications sont optimisées pour les systèmes multi-cœurs. Il reste difficile de développer efficacement et de façon rentable des applications parallèles. Ces dernières années, de plus en plus de chercheurs dans le domaine de la HPS ont commencé à utiliser les GPU (Graphics Processing Unit, unité de traitement graphique) pour accélérer les applications parallèles. Une GPU est composée de nombreux cœurs plus petits et plus simples que les processeurs de CPU multi-cœurs des ordinateurs de bureau. Il n'est pas difficile d'adapter une application en série à une plateforme GPU. Bien que peu d'efforts soient nécessaires pour adapter de manière fonctionnelle les applications aux GPU, les programmeurs doivent encore passer beaucoup de temps à optimiser leurs applications pour de meilleures performances.

Afin de mieux comprendre le résultat des performances et de mieux optimiser les applications de GPU, la communauté GPGPU travaille sur plusieurs thématiques intéressantes. Des modèles de performance analytique sont créés pour aider les développeurs à comprendre le résultat de performance et localiser le goulot d'étranglement. Certains outils de réglage automatique sont conçus pour transformer le modèle d'accès aux données, l'agencement du code, ou explorer automatiquement l'espace de conception. Quelques simulateurs pour applications de GPU sont également lancés. La difficulté évidente pour l'analyse de performance des applications de GPGPU réside dans le fait que l'architecture sous-jacente de la GPU est très peu documentée. La plupart des approches développées jusqu'à présent n'étant pas assez bonnes pour une optimisation efficace des applications du monde réel, et l'architecture des GPU évoluant très rapidement, la communauté a encore besoin de perfectionner les modèles et de développer de nouvelles approches qui permettront aux développeurs de mieux optimiser les applications de GPU.

Dans ce travail de thèse, nous avons principalement travaillé sur deux aspects de l'analyse de performance des GPU. En premier lieu, nous avons étudié comment mieux estimer les performances des GPU à travers une approche analytique. Nous souhaitons élaborer une approche suffisamment simple pour être utilisée par les développeurs, et permettant de mieux visualiser les résultats de performance. En second lieu, nous tentons d'élaborer une approche permettant d'estimer la limite de performance supérieure d'une application dans certaines architectures de GPU, et d'orienter l'optimisation des performances.

Ce résumé est organisé de la manière suivante : la section 2 présente de simples modèles de flux de données d'application de la QCD sur réseau dans des architectures GPGPU GT200 et Cell B.E. La section 3 présente notre travail sur l'estimation de performance à l'aide d'une approche analytique, qui a fait partie du séminaire de travail Rapido 2012. La section 4 présente notre travail sur l'analyse de la limite de performance supérieure des applications de GPU ; il fera partie du CGO 2013. La section 5 conclut cette thèse et fournit des orientations pour le futur.

1 Modèle de flux de données de QCD sur réseau sur Cell B.E. et GPGPU

La chromodynamique quantique (QCD pour Quantum chromodynamics) est la théorie physique des interactions entre les éléments fondamentaux de la matière, et la QCD sur réseau est une approche numérique systématique pour l'étude de la théorie de la QCD. L'objectif de cette partie du travail de thèse est de fournir des modèles de performance analytique de l'algorithme de QCD sur réseau sur architecture multi-cœur. Deux architectures, les processeurs GPGPU GT200 et CELL B.E., sont étudiées et les abstractions matérielles sont proposées.

2.1 Comparaison de deux modèles analytiques

La Figure 1 offre une comparaison des deux modèles présentés. Les principales différences entre les deux plateformes de mise en œuvre de la QCD sur réseau sont les différences de hiérarchie de mémoire et de modèle d'interconnexion des différentes unités de processeurs, qui auront une influence sur le modèle d'accès à la mémoire. Le modèle d'accès à la mémoire est la clé des exigences en termes de flux de données, et donc, la clé de la performance.

2.2 Routine de QCD sur réseau Hopping_Matrix

Hopping_Matrix est la routine la plus longue de l'algorithme de QCD sur réseau : elle occupe environ 90 % du temps d'exécution total. Les structures de données d'entrée de la routine Hopping_Matrix incluent le champ de spineur, le champ de jauge,

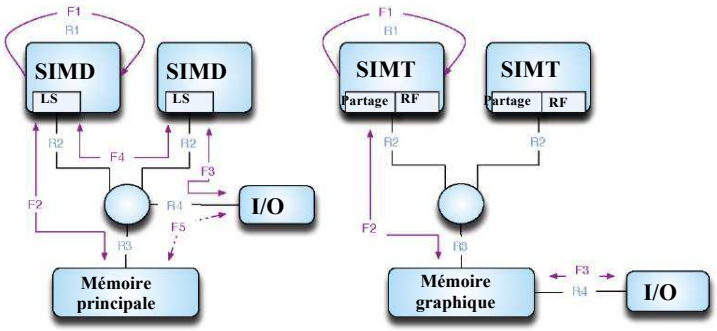

Figure 1 : *comparaison des modèles analytiques de Cell et GPU*

le résultat est le champ de spineur obtenu. Les données temporaires correspondent au champ du demi-spineur intermédiaire.

2.3 Analyse de performance

Notre méthodologie consiste à obtenir la performance potentielle sur la base de l'analyse du flux de données. Avec des modèles de processeurs et l'application, les modèles d'accès à la mémoire sont résumés, ce qui permet ensuite de générer les informations relatives au flux de données. Il est ensuite possible d'estimer les exigences des données en termes de bande passante sur la base des informations relatives au flux de données. En identifiant le composant du goulot d'étranglement, la performance potentielle de l'application est calculée par l'intermédiaire de la bande passante maximale du composant.

En utilisant les modèles analytiques présentés, on catégorise les modèles d'accès à la mémoire comme indiqué dans le Tableau 1.

Dans une mise en œuvre, tous les modèles peuvent ne pas être appliqués simultanément, en raison des contraintes de ressources du processeur. Pour différentes mises en œuvre, de nombreuses combinaisons de ces modèles sont donc applicables. Pour obtenir des performances optimales sur une architecture spécifique, il est possible de sélectionner la meilleure combinaison en fonction des caractéristiques de l'architecture.

Pour un processeur Cell B.E., la base locale peut contenir les données de champ d'un sous-réseau avec suffisamment de sites d'espace-temps. Les modèles P2 et P3 pourraient donc être appliqués. Le SPE pouvant émettre directement des opérations I/O, les données de champ du demi-spineur de limite peuvent être directement transférées sans être réécrites dans la mémoire principale. Le modèle P4 est donc réalisable. Différents SPE pouvant communiquer directement à travers l'EIB, le modèle P5 est également réalisable. La combinaison optimale pour le processeur Cell est (01111). Avec la combinaison de modèle (01111), la performance de pointe potentielle pour

P1	Reconstitution du champ de jauge dans le processeur
P 2	Partage total des données du champ de jauge entre les sites d'espace-temps voisins
P 3	Les données de champ du demi-spineur intermédiaire sont contenues dans la mémoire rapide locale, sans nécessiter de réécriture dans la mémoire principale
P 4	Les données de champ du demi-spineur de limite inter processeurs sont stockées dans la mémoire rapide locale, sans nécessiter de réécriture dans la mémoire principale
P 5	Les données de champ du demi-spineur de limite inter-cœurs sont stockées dans la mémoire rapide locale, sans nécessiter de réécriture dans la mémoire principale

Tableau 1 : *modèle d'accès à la mémoire*

DSlash est d'environ 35 GFlops (34 % de la performance de pointe théorique de Cell, avec 102,4 GFlops).

Pour la GPU GT200, il est impossible de stocker l'ensemble des données de champ de demi-spineur intermédiaire. La GPU n'étant pas capable d'émettre directement les opérations I/O, le modèle P4 est impossible. Il n'y a pas de communication directe entre cœurs dans le GPU. P5 est donc également irréalisable. Chaque GPU ayant une grande puissance de calcul, il est envisageable de reconstruire les données de champ de jauge à l'intérieur du processeur. La combinaison de modèle possible pourrait donc être (10000). Avec la combinaison de modèle (10000), si l'on tient uniquement compte d'un nœud de GPU simple, la performance potentielle est de 75,6 GFlops, soit environ 65 % de la performance de pointe théorique en double précision.

2 Estimation de performance des applications de GPU à travers l'utilisation d'une méthode analytique

L'objectif de la deuxième partie de ce travail de thèse est de fournir une approche analytique permettant de mieux comprendre les résultats de performance des GPU. Nous avons développé un modèle de temporisation pour la GPU NVIDIA GT200 et construit l'outil TEG (Timing Estimation tool for GPU) sur la base de ce modèle. TEG prend pour éléments de départ le code assembleur de noyau CUDA et le suivi des instructions. Le code binaire du noyau CUDA est désassemblé à l'aide de l'outil *cuobjdump* fourni par NVIDIA. Le suivi des instructions est obtenu grâce au simulateur Barra. Ensuite, TEG modélise l'exécution du noyau sur la GPU et collecte les informations de temporisation. Les cas évalués montrent que TEG peut obtenir une approximation de performance très proche. En comparaison avec le

approximation. En comparaison avec le nombre réel de cycles d'exécution, TEG présente généralement un taux d'erreur inférieur à 10 %.

3.1 Paramètres du modèle

Pour utiliser le modèle analytique dans TEG, il faut définir des paramètres du modèle. Cette section présente certains des principaux paramètres.

La latence d'*exécution* d'une instruction de chaîne désigne les cycles au cours desquels l'instruction est active dans l'unité fonctionnelle correspondante. Après la latence d'exécution, une instruction de chaîne émise est marquée comme terminée.

La *latence d'émission de la même chaîne* d'une instruction correspond aux cycles au cours desquels le moteur d'émission doit attendre avant d'émettre une autre instruction, après avoir émis une instruction de chaîne. Elle est calculée à l'aide du débit d'instruction.

La *latence d'émission de la même chaîne* correspond aux cycles au cours desquels le moteur d'émission doit attendre avant d'émettre une autre instruction issue de la même chaîne, après avoir émis une instruction de chaîne. Cette latence peut également être mesurée à l'aide de la fonction d'horloge() ; elle est généralement plus longue que la latence d'émission de plusieurs chaînes.

3.2 Évaluation

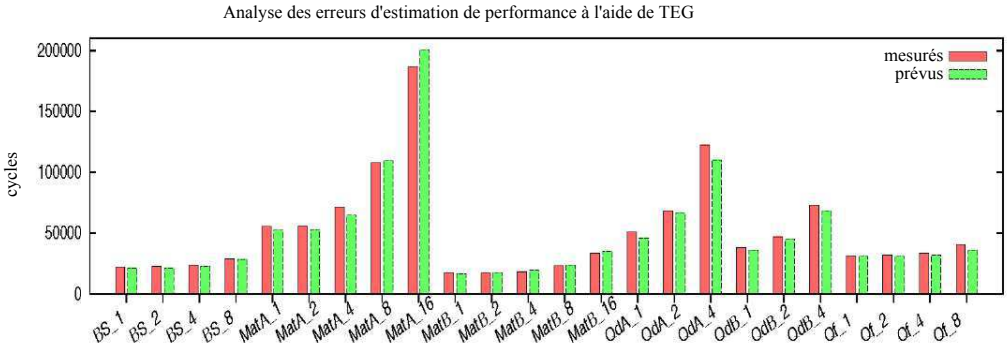

Figure 2 : analyse des erreurs de TEG

Nous évaluons TEG à l'aide de plusieurs repères selon différentes configurations et comparons les temps d'exécution mesurés et estimés du noyau. Le résultat est présenté dans la Figure 2. Le nom est défini ainsi : *NomDeNoyau-NombreDeChaîne*s. *BS, MatA, MatB, QdA, QdB, Qf* correspondent respectivement à Blackscholes, multiplication naïve de matrice, multiplication de matrice sans conflit de banque de mémoire partagée, noyau

QCD sur réseau en double précision avec accès mémoire non coalescé, noyau QCD sur réseau en double précision avec accès mémoire coalescé, et noyau QCD sur réseau en simple précision. *NombreDeChaînes* est le nombre de chaînes concomitantes attribuées à chaque SM. Ici, la même charge est attribuée à toutes les chaînes. Le résultat indique que TEG présente une bonne approximation et qu'il peut également déceler le comportement de mise à l'échelle des performances. Le taux moyen d'erreur absolue relative est de 5,09 % et le taux maximum d'erreur absolue relative est de 11,94 %.

3 Analyse de la limite de performance supérieure et optimisation de SGEMM sur les GPU Fermi et Kepler

Pour comprendre les résultats de performance des GPU, il existe de nombreux travaux traitant de la façon de prévoir/prédire la performance des applications CUDA à travers des méthodes analytiques. Toutefois, les modèles de performance des GPU reposent tous sur un certain niveau de mise en œuvre de l'application (code C++, code PTX, code assembleur...) et ne répondent pas à la question de la qualité de la version optimisée actuelle, et de l'utilité d'un éventuel effort d'optimisation supplémentaire. Différente des modèles de performance des GPU existants, notre approche ne prévoit pas la performance possible en fonction de certaines mises en œuvre, mais la limite de performance supérieure qu'une application ne peut dépasser.

4.1 Approche d'analyse générale pour la performance de pointe potentielle

L'approche d'analyse générale peut être la même pour toutes les applications, mais le processus d'analyse détaillée peut varier d'une application à l'autre.

En premier lieu, nous devons analyser les types d'instructions et le pourcentage d'une routine. En second lieu, nous devons trouver quels paramètres critiques ont un impact sur le pourcentage de mélange des instructions. Troisièmement, nous analysons de quelle manière le débit d'instruction varie en fonction de la modification de ces paramètres critiques. Quatrièmement, nous pouvons utiliser le débit d'instructions et la combinaison optimale des paramètres critiques pour estimer la limite de performance supérieure. Avec cette approche, nous pouvons non seulement obtenir une estimation de la limite de performance supérieure, connaître l'écart de performance restant et déterminer l'effort d'optimisation, mais aussi comprendre quels paramètres sont essentiels à la performance et comment répartir notre effort d'optimisation.

4.2 Analyse de la performance de pointe potentielle pour SGEMM

Pour SGEMM, tous les noyaux SGEMM correctement mis en œuvre utilisent la mémoire partagée de la GPU pour diminuer la pression exercée sur la mémoire globale. Les données sont d'abord chargées depuis la mémoire globale vers la mémoire partagée, puis les threads d'un même bloc peuvent partager les données chargées dans la mémoire partagée. Pour les GPU Fermi (GF110) et Kepler (GK104), des instructions arithmétiques telles que FFMA ne peuvent pas accepter d'opérandes en provenance de la mémoire partagée. Les instructions LDS étant nécessaires au chargement des données initial depuis la mémoire partagée vers les registres, la plupart des instructions exécutées en SGEMM sont des instructions FFMA et LDS.

4.2.1 Utilisation d'instructions de chargement plus étendues

Pour obtenir de meilleures performances, il est essentiel de réduire au minimum le pourcentage d'instructions auxiliaires. Par instructions auxiliaires, nous entendons les instructions non mathématiques, et notamment les instructions LDS. Le code assembleur pour CUDA sm_20 (GPU GF110 Fermi) et sm_30 (GPU GK104 Kepler) fournit des instructions LDS.64 et LDS.128 similaires aux instructions SIMD pour le chargement de données 64 et 68 bits à partir de la mémoire partagée. L'utilisation d'instructions de chargement plus étendues peut réduire le nombre total d'instructions LDS. Cependant, la performance globale n'est pas toujours améliorée par l'utilisation de telles instructions.

4.3 Facteurs de mise en blocs du registre et de la mémoire partagée

Une taille de mise en blocs du registre plus importante peut entraîner une plus forte réutilisation du registre pour un même thread, et un pourcentage plus élevé d'instructions FFMA. Toutefois, la taille de mise en blocs du registre est limitée par la ressource de registre sur le SM et la contrainte du jeu d'instructions. Avec un facteur de mise en blocs du registre B_R, $T_B * B^2_R$ est la taille de la sous-matrice C par bloc (chaque bloc a des threads TB) et $\sqrt{T_B * B^2_R} * L$ est la taille d'une sous-matrice pour A ou B (L est le pas). Pour un transfert des données et un calcul simultanés, des registres supplémentaires sont nécessaires afin d'acheminer les données de la mémoire globale vers la mémoire partagée, puisqu'aucun transfert direct n'est assuré entre les deux espaces de mémoire. Le pas L doit être choisi de manière à ce que chaque thread charge la même quantité de données (équation 1).

$$\sqrt{T_B} * B_R * L) \% T_B = 0$$

Si l'on considère que les données sont préalablement acheminées depuis la mémoire globale et que quelques registres stockent les adresses des matrices dans la mémoire globale et la mémoire partagée, (R_{adr}), la contrainte globale stricte pour le facteur de mise en blocs du registre peut être décrite à travers l'équation 2.

$$B^2{}_R + \frac{2*\sqrt{T_B}*R_R*L}{T_B} + B_R + 1 + R_{adr} \leq R_T \leq R_{Max} \quad (2)$$

La mémoire partagée étant attribuée en granularité par blocs, pour les blocs actifs Blk , $Blk * 2 * \sqrt{T_B}*B_R*L$ est nécessaire pour le stockage des données pré-acheminées de la mémoire globale (équation 3). Le facteur de mise en blocs de mémoire peut être défini ainsi : $B_{Sh} = \sqrt{T_B * B^2{}_R}$. Avec le facteur de mise en blocs de mémoire BSh, la performance limitée par la bande passante de la mémoire globale peut être estimée approximativement à l'aide de l'équation 4.

$$Blk * 2 * \sqrt{T_B}*B_R*L \leq Sh_{SM} \quad (3)$$

$$\frac{P_{MemBound}}{\#GlobalMem\_bandwidth} = \frac{2*B_{Sh2}}{2*B_{Sh}*4} \quad (4)$$

4.4 Performance de pointe potentielle pour SGEMM

Le facteur d'instruction F_I est le ratio d'instructions FFMA dans la boucle principale SGEMM (on ne tient compte ici que des instructions FFMA et LDS.X). Il dépend du choix de l'instruction LDS.X et du facteur de mise en blocs du registre B_R. Par exemple, si LDS.64 est utilisé avec un facteur de mise en blocs du registre de 6, $F_I = 0{,}5$.

Le facteur de débit FT est fonction du facteur de mise en blocs du registre (B_R), du nombre de threads actifs (T_{SM}), du débit des SPs (*#SP_TP*), des unités LD/ST *(#LDS.TP*) et des unités de répartition *(#Émission.TP*)) (équation 5).

$$Ft = f(B_R, \#Émission\_TP, \#SP\_TP, \#LDS\_TP, T_{SM}) \quad (5)$$

Avec le facteur de mise en blocs du registre B_R, le facteur d'instruction F_I et le facteur de débit F_T, la performance limitée par le débit de traitement des SM est estimée selon l'équation 6 et la performance globale selon l'équation 7.

$$P_{Limitée\ par\ SM} = \frac{B^2{}_R}{B^2{}_R + B_R*2*F_I} * F_T * P_{théorique} \quad (6)$$

$$P_{potentielle} = min(P_{Limitée\ par\ mémoire}, P_{Limitée\ par\ SM}) \quad (7)$$

L'analyse précédente nous permet d'estimer la limite de performance supérieure de SGEMM sur les GPU Fermi et Kepler. Par exemple, sur les GPU Fermi, en raison de la limite stricte de 63 registres *(R_{Max})* par thread, en tenant compte de l'acheminement préalable et de l'utilisation de la condition stricte de l'équation 2, le facteur maximal de mise en blocs n'est que de 6. Selon les équations 4, 6 et 7, la performance est limitée par le débit de traitement des SM, et la pointe potentielle est égale à environ 82,5 % ($\frac{6^2}{6^2+6*2*0.5} * \frac{30.8}{32}$) de la performance de pointe théorique pour SGEMM. La principale limite est due à la nature du jeu d'instructions Fermi et au débit d'émission limité des ordonnanceurs.

4 Conclusion

Ce travail nous a permis d'apporter deux contributions.

La première est le développement d'une méthode analytique pour prédire la performance des applications CUDA à l'aide du code assembleur de cuobjdump pour les GPU de génération GT200. Nous avons également développé un outil d'estimation temporelle (TEG) pour évaluer le temps d'exécution du noyau de GPU. TEG utilise le résultat d'un outil désassembleur NVIDIA, *cuobjdump*. *cuobjdump* peut traiter le fichier binaire de CUDA et générer des codes assembleurs. TEG n'exécute pas les codes, mais utilise uniquement des informations telles que le type d'instruction, les opérandes, etc. Avec le suivi des instructions et d'autres résultats nécessaires d'un simulateur fonctionnel, TEG peut fournir une estimation temporelle approximative des cycles. Cela permet aux programmeurs de mieux comprendre les goulots d'étranglement de la performance et le degré de pénalité qu'ils peuvent entraîner. Il suffit alors de supprimer les effets des goulots d'étranglement dans TEG, et d'estimer à nouveau la performance pour effectuer une comparaison.

La deuxième contribution principale apportée par cette thèse est une approche pour l'estimation de la limite supérieure de performance des applications de GPU basée sur l'analyse des algorithmes et une analyse comparative au niveau du code assembleur. Il existe de nombreux travaux sur la façon d'optimiser des applications de GPU spécifiques, et de nombreuses études relatives aux outils de réglage. Le problème est que nous ne savons pas avec certitude si le niveau de performance obtenue est proche de la meilleure performance potentielle qu'il est possible d'obtenir. Avec la limite de performance supérieure d'une application, nous connaissons l'espace d'optimisation restant et nous pouvons déterminer l'effort d'optimisation à fournir. L'analyse nous permet également de comprendre quels paramètres sont critiques pour la performance. En exemple, nous avons analysé la performance de pointe potentielle de SGEMM (Single-precision General Matrix Multiply) sur les GPU Fermi (GF110) et Kepler (GK104). Nous avons tenté de répondre à la question « quel est l'espace d'optimisation restant pour SGEMM, et pourquoi ? ». D'après notre analyse, la nature du jeu d'instruction Fermi (Kepler) et le débit d'émission limité des ordonnanceurs sont les principaux facteurs de limitation de SGEMM pour approcher la performance de pointe théorique. La limite supérieure de performance de pointe estimée de SGEMM représente environ 82.5 % de la performance de pointe théorique sur les GPU Fermi GTX580, et 57,6 % sur les GPU Kepler GTX680. Guidés par cette analyse et en utilisant le langage assembleur natif, en moyenne, nos mises en œuvre SGEMM ont obtenu des performances supérieures d'environ 5 % que CUBLAS dans CUDA 4.1 SDK pour les grandes matrices sur GTX580. La performance obtenue représente environ 90 % de la limite de performance supérieure de SGEMM sur GTX580.

Introduction

This thesis work is done in the context of the ANR PetaQCD project which amis at understanding how the recent programmable hardware accelerators such as the now abandoned Cell B.E. [41] and the high-end GPUs could be used to achieve the very high level of performance required by QCD (Quantum chromodynamics) simulations. QCD (Quantum chromodynamics) is the physical theory for strong interactions between fundamental constituents of matter and lattice QCD is a systematic numerical approach to study the QCD theory.

The era of multi-core has come. Vendors keep putting more and more computing cores on die and consumers are persuaded to upgrade their personal computers to platforms with more cores. However, the research and development in parallel software remain slower than the architecture evolution. For example, nowadays, it is common to have a 4-core or 6-core desktop CPU, but very few applications are optimized for the multi-core system. There are several reasons. First, developers normally start to learn serial programming and parallel programming is not the natural way that programmers think of problems. Second, there are a lot of serial legacy codes and many softwares are built on top of these legacy serial components. Third, parallel programming introduces more difficulties like task partition, synchronization, consistency than serial programming. Fourth, the programming models may be different for various parallel architectures. How to efficiently and effectively build parallel applications remains a difficult task.

In recent years, more and more HPC researchers begin to pay attention to the potential of GPUs (Graphics Processing Unit) to accelerate parallel applications since GPU can provide enormous computing power and memory bandwidth. GPU has become a good candidate architecture for both computation bound and memory bound HPC (High-Performance Computing) applications. GPU is composed of many smaller and simpler cores than desktop multi-core CPU processors. The GPU processor is more power efficient since it uses very simple control logic and utilizes a large pool of threads to saturate math instruction pipeline and hide the memory access latency. Today, many applications have already been ported to the GPU platform with programming interfaces like CUDA [2] or OpenCL [77] [99, 78, 38, 47, 89, 60, 102, 58, 66, 28, 97, 14, 79]. It is not difficult to port a serial application onto the GPU platform. Normally, we can have some speedup after simply parallelizing the original code and executing the application on GPU. Though little efforts are needed to functionally port applications on GPU, programmers still have to spend lot of time to optimize their applications to achieve *good* performance. Unlike the serial programming, programming GPU applications requires more knowledge of the underlying hardware features. There are many performance degradation factors on GPU. For example, proper data access pattern needs

to be designed to group the global memory requests from the same group of threads and avoid conflicts to access the shared memory. Normally, to develop real world applications, most programmers have to exhaustively explore a very large design space to find a good parameter combination and rely on their programming experience [80]. This process requires a lot of expert experience on performance optimization and the GPU architecture. The learning curve is very long. How to efficiently design a GPU application with very good performance is still a challenge.

To better understand the performance results and better optimize the GPU applications, the GPGPU community is working on several interesting topics. Some analytical performance models are developed to help developers to understand the performance result and locate the performance bottlenecks [61, 40, 85, 101, 26]. Some automatic tuning tools are designed to transform the data access pattern and the code layout to search the design space automatically [27, 100, 31, 62]. A few simulators for GPU applications are introduced too [83, 29, 11, 24]. The obvious difficulty for GPGPU application performance analysis is that the underlying architecture of GPU processors has very few documentations and sometimes, the vendors intentionally hide some architecture details [54]. Researchers have to develop performance models or automatic tuning tools without fully understanding the GPU hardware characteristics. Since most of the approaches developed so far are not mature enough to efficiently optimize real world applications and the GPU architecture is evolving very quickly, the community still needs to refine existing performance models and develop new approaches to help developers to better optimize GPU applications.

In this thesis work, we have mainly worked on two topics of GPU performance analysis. First, we studied how to better estimate the GPU performance with an analytical approach. Apparently it is not realistic to build detailed simulators to help developers to optimize performance and the existing statistics profilers cannot provide enough information. So we want to design an approach which is simple enough for developers to use and can provide more insight into the performance results. Second, although we can project the possible performance from certain implementations like many other performance estimation approaches, we still do not answer the question of how good the current optimized version is and whether further optimization effort is worthwhile or not. So we try to design an approach to estimate the performance upper bound of an application on certain GPU architectures and guide the performance optimization.

Contributions

There are two main contributions of this work.

As first contribution of this work, we have developed an analytical method to predict CUDA application's performance using assembly code from cuobjdump for GT200 generation GPU. Also we have developed a timing estimation tool (TEG) to estimate GPU kernel execution time. TEG takes the output of a NVIDIA disassembler tool *cuobjdump* [2]. *cuobjdump* can process the CUDA binary file and generate assembly codes. TEG does not execute the codes, but only uses the information such as instruction type, operands, etc. With the instruction trace and some other necessary output of a functional simulator, TEG can give the timing estimation

in cycle-approximate level. Thus it allows programmers to better understand the performance bottlenecks and how much penalty the bottlenecks can introduce. We just need to simply remove the bottlenecks' effects from TEG, and estimate the performance again to compare.

The second main contribution of this thesis is an approach to estimate GPU applications' performance upper bound based on application analysis and assembly code level benchmarking. There exist many works about how to optimizie specific GPU applications and also a lot of study on automatic tuning tools. But the problem is that there is no estimation of the distance between the obtained performance and the best potential performance we can achieve. With the performance upperbound of an application, we know how much optimization space is left and can decide the optimization effort. Also with the analysis we can understand which parameters are critical to the performance. As an example, we analyzed the potential peak performance of SGEMM (Single-precision General Matrix Multiply) on Fermi (GF110) and Kepler (GK104) GPUs. We tried to answer the question of how much optimization space is left for SGEMM and why. According to our analysis, the nature of Fermi (Kepler) instruction set and the limited issuing throughput of the schedulers are the main limitation factors for SGEMM to approach the theoretical peak performance. The estimated upper bound peak performance of SGEMM is around 82.5% of the theoretical peak performance on GTX580 Fermi GPU and 57.6% on GTX680 Kepler GPU. Guided by this analysis and using the native assembly language, on average, our SGEMM implementations achieve about 5% better performance than CUBLAS in CUDA 4.1 SDK for large matrices on GTX580. The achieved performance is around 90% of the estimated upper bound performance of SGEMM on GTX580. On GTX680, the best performance we have achieved is around 77.3% of the estimated performance upper bound.

Organization of the document

This thesis is organized as follows: Chapter 1 gives an introduction to GPU architectures and CUDA programming model, and the state of art on GPU performance modeling and analysis. Chapter 2 introduces simple data-flow models of lattice QCD application on GPGPU and the Cell B.E. architectures. Chapter 3 introduces our work on performance estimation using an analytical approach, which appeared in Rapido '12 workshop [53]. In Chapter 4, our work on GPU applications' performance upper bound analysis is presented, which is going to appear in CGO '13 [54].

Chapter 1

Performance Analysis of GPU applications

1.1 GPU Architecture and CUDA Programming Model

1.1.1 GPU Processor

Throughput-oriented GPU (Graphics Processing Unit) processor represents a major trend in the recent advance on architecture for parallel computing acceleration [57]. As the most obvious feature, GPU processor includes a very large number of fairly simple cores instead of few complex cores like conventional general purpose desktop multicore CPUs. For instance, the newly announced Kepler GPU K20X (November 2012) has 2688 SPs (Streaming Processor) [76]. Thus GPU processor can provide an enormous computing throughput with a relatively low clock. Again, the new K20X GPU has a theoretical single precision performance of 3.95 Tera FLOPS (FLoating point Operations Per Second) with a core clock of only 732MHz.

Unlike traditional graphics API, programming interfaces like CUDA [2, 68] and OpenCL [77] have been introduced to reduce the programming difficulty. These programming interfaces normally are a simple extention for C/C++. Thus to port on GPU, it is fairly easy comparing to platforms like Cell B.E. processor [41] or FGPA. Although the performance may not be very good, normally developers can construct a GPU-parallelized version based on the original serial code without a lot of programming effort. Thus, more and more developers are considering moving their serial application to GPU platform.

However, most of the time, it is easy to get some speedup porting the serial code on GPU, but a lot of efforts are needed to fully utilize the GPU hardware potential and achieve a very good performance. The code needs to be carefully designed to avoid the performance degradation factors on GPU, like shared memory bank conflict, uncoalesced global memory accessing and so on [2]. Also, there are many design variations like the computing task partition, the data layout, CUDA parameters, etc. These variations compose a large design space for the developers to explore. How to efficiently design a GPU application with a good performance remains a challenge.

The GPU programming model is similar to the single-program multiple-data (SPMD) programming model. Unlike single-instruction multiple-data (SIMD) model, the SPMD model

does not require all execution lanes execute the exact same instruction. In implementation, mechanism like thread masks are used to disable certain lanes which is not on the current execution path. Thus the GPU programming is more flexible than a SIMD machine.

1.1.2 Comparison of Recent Generations of NVIDIA GPUs

In our research, we used NVIDIA GPUs as target hardware platform. We have worked on three generations of NVIDIA GPUs, including GT200 GPU (GT200) [73], Fermi GPU (GF100) [74] and Kepler GPU (GK104) [75]. The most recent Nvidia GPU at the time of this thesis is K20X Kepler GPU (GK110), which is announced in November 2012.

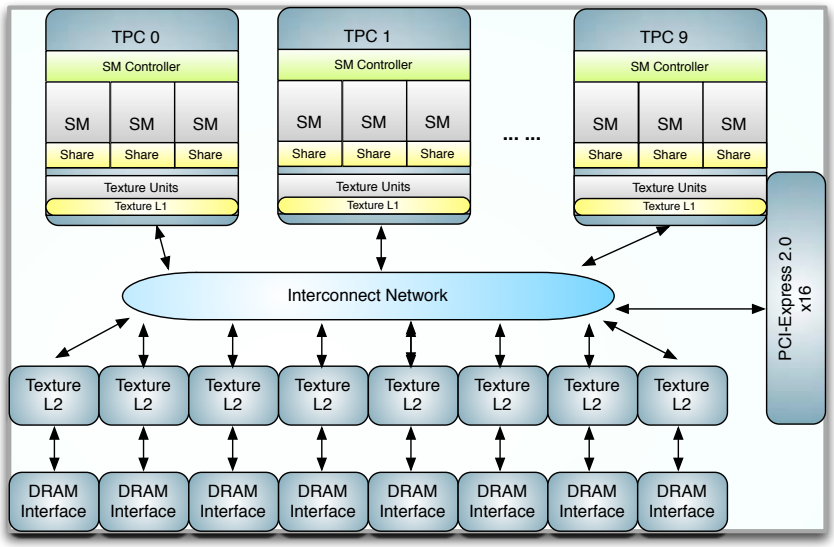

Figure 1.1: Block Diagram of GT200 GPU

GPU can be simply considered as a cluster of independent SMs (Streaming Multiprocessor). Figure 1.1 illustrates the block diagram of GT200. GT200 GPU is composed of 10 TPCs (Thread ProcessingCluster), each of which includes 3 SMs. SMs in one TPC share the same memory pipeline. Each SM further includes scheduler, 8 SPs (Streaming Processor), 1 DPU (Double Precision Unit) and 2 SFUs (Special Function Unit). SP executes single precision floating point, integer arithmetic and logic instructions. The SPs inside one SM, which is the basic computing component, are similar to a lane of SIMD engines and they share the memory resource of the SM like the registers and shared memory. DPU executes double precision floating point instructions. And SFU handles special mathematical functions, as well as single precision floating point multiplication instructions. If we consider SP as one core, then one GPU processor is comprised of 240 cores.

For Geforce GTX 280 model, with 1296MHz shader clock, the single precision peak performance can reach around 933GFlops. GT280 has 8 64-bit wide GDDR3 memory controllers. With 2214MHz memory clock on GTX 280, the memory bandwidth can reach around

140GB/s. Besides, within each TPC there is a 24KB L1 texture cache and 256KB L2 Texture cache is shared among TPCs.

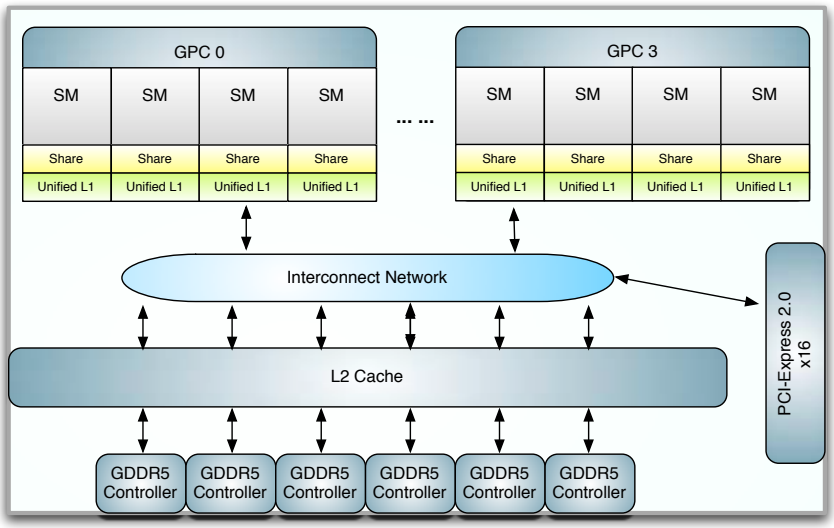

Figure 1.2: Block Diagram of Fermi GPU

Figure 1.2 is the block diagram of Fermi GPU. Fermi GPU has 4 GPCs (Graphics Processing Clusters) and in all 16 SMs. The number of SPs per SM increase to 32. The most significant difference is that Fermi GPU provides real L1 and L2 cache hierarchy. Local writes are written back to L1 when register resource is not sufficient. Global stores bypass L1 cache since multiple L1 caches are not coherent for global data. L2 cache is designed to reduce the penalty of some irregular global memory accesses.

As an example, Geforce GTX 580 has a shader clock of 1544 MHz and the theoretical single precision peak performance of 1581 GFlops. The memory controllers are upgraded to GDDR5 and a bandwidth of 192.4 GB/s.

The Kepler GPU's high level architecture is very close to Fermi GPU. The main difference is the scheduling functional units, which cannot be shown on the block diagram level.

A comparison of the three generations of NVIDIA GPUs is illustrated in Table 1.1. From GT200 to Kepler GPU, the number of SPs increases dramatically, from 240 (GTX280, 65nm) to 1536 (GTX680, 28nm) [75, 74]. Each SM in Fermi GPU consists of 32 SPs instead of 8 SPs on GT200 GPU. On Kepler GPU, each SM (SMX) includes 192 SPs. For GTX280, each SM has 16KB shared memory and 16K 32bit registers. In GTX580, shared memory per SM increases to 48KB and the 32bit register number is 32K. GTX680 has the same amount of shared memory with GTX580 and the register number increases to 64K. However, if we consider the memory resource (registers and shared memory) per SP, the on-die storage per SP actually decreases. The global memory bandwidth actually does not change a lot. Previous generations have two clock domains in the SM, the core clock for the scheduler and the shader clock for the SPs. The shader clock is roughly twice the speed of the core clock. On Kepler (GK104) GPU, shader clock no longer exists, the functional units with SMs run at the same

	GT200 (GTX280)	Fermi (GTX580)	Kepler (GTX680)
Core Clock (MHz)	602	772	1006
Shader Clock (MHz)	1296	1544	1006
Global Memory Bandwidth(GB/s)	141.7	192.4	192.26
Warp Scheduler per SM	1	2	4
Dispatch Unit per SM	1	2	8
Thread Instruction issuing throughput per shader cycle per SM	16	32	128?
SP per SM	8	32	192
SP Thread Instruction processing throughput per shader cycle per SM (FMAD/FFMA)	8	32	192?
LD/ST (Load/Store) Unit per SM	unknown	16	32
Shared Memory Instruction processing throughput per shader cycle per SM (LDS)	unknown	16	32
Shared Memory per SM	16KB	48KB	48KB
32bit Registers per SM	16K	32K	64K
Theoretical Peak Performance (GFLOPS)	933	1581	3090

Table 1.1: Architecture Evolution

core clock. However, to compare the different generations more easily, we still use the term shader clock on Kepler GPU and the shader clock is the same as the core clock. In the rest of this thesis, all throughput data is calculated with the shader clock.

1.1.3 CUDA Programming Model

The Compute Unified Device Architecture (CUDA) [2, 50] is widely accepted as a programming model for NVIDIA GPUs. It is a C-like programming interface with a few extensions to the standard C/C++. A typical CUDA program normally creates thousands of threads to hide memory access latency and math instruction pipeline latency since the threads are very light weight. One of the most important characteristics of GPU architecture is that the memory operation latency could be hidden by concurrently executing multiple memory requests or executing other instructions during the waiting period. The threads are grouped into 1D to 3D *blocks* or cooperative thread arrasy (CTAs) [57], and further into 1D or 2D *grids*. The *warp* is the basic execution and scheduling unit of a SM, and is composed of 32 threads within one block on current NVIDIA GPUs.

All threads have access to global memory space or device memory. Accessing global memory generally takes hundreds of cycles. The memory accesses by a warp of 32 threads could be combined into fewer memory transactions and referred to as coalesced global memory access. Threads within one block can share data in shared memory and synchronize with a barrier synchronization operation. The shared memory has very low latency comparing to global memory.

Efficiently utilizing shared memory can significantly reduce the global memory pressure and reduce average memory access latency. The shared memory is organized in banks. Bank conflict could happen if multiple threads in a warp access the same bank.

Each thread has its own local memory and register resource. Each block is assigned to one SM at execution time and one SM can execute multiple blocks concurrently. The shared memory and register resource consumed by one block has the same lifetime as the block. On the SM, since the memory resource like register file and shared memory is limited, only a limited set of threads can run concurrently (active threads).

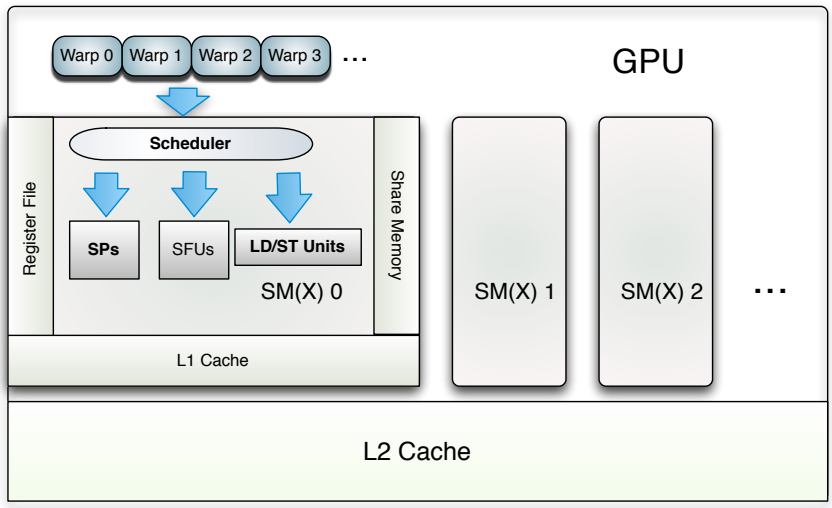

Figure 1.3: CUDA Execution Model on NVIDIA GPUs

Figure 1.3 presents a simplified CUDA execution model. The scheduler uses a score board to select one ready instruction from one of the active warps and then issue the instruction to the corresponding functional unit. There is no penalty to issue instruction other current warp. With this light weight context switching mechanism, some latency can be hidden. However, programmers still need to provide enough number of threads which can be executed concurrently to get good occupancy [2].

On one hand, the increased SPs per SM require more active threads to hide latency. On the other hand, the register and shared memory limit the number of active threads. For the same application, the active threads that one SP supports actually decreases because of the reduced memory resource per SP from Fermi GPU to Kepler GPU. More instruction level parallelism within one thread needs to be explored.

A CUDA program is composed of host code running on the host CPU, and device code running on the GPU processor. The compiler first split the source code into host code and device code. The device code is first compiled into the intermediate PTX (Parallel Thread eXecution) code [71], and then compiled into native GPU binary code by the assembler ptxas. The device binary and device binary code are combined into the final executable file. The compiling stages are illustrated in Figure 1.4. NVIDIA provides the disassembler *Cuobjdump*

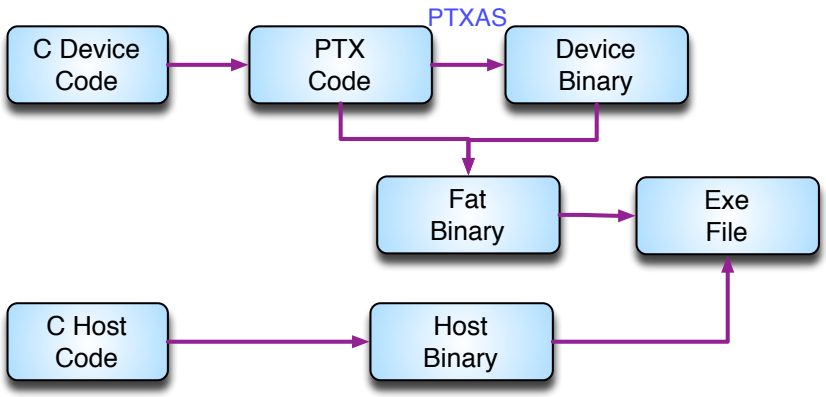

Figure 1.4: Compiling Stages of CUDA Programms

which can convert GPU binary code into human-readable assembly codes [2].

1.2 Performance Prediction of GPU Applications Using Simulation Approach

There are already several simulators for graphics architectures [83, 29, 11, 24]. The Qsilver [83] and ATTILLA [29] simulators are not designed for general purpose computing on GPUs and focus on the graphics features. The Barra simulator [24] is a functional simulator and does not provide timing information. The GPGPU-Sim [34, 11] is a cycle-accurate simulator for CUDA applications executing on NVIDIA GPUs and omits hardware not exposed to CUDA. The following part of this section briefly introduces the approach of GPGPU-Sim simulator.

1.2.1 Baseline Architecture

The GPGPU-Sim simulates a GPU running CUDA applications. Some hardware features of the baseline architecture are collected from NVIDIA pattern files. The simulated GPU consists of a cluster of shader cores, which is similar to SMs in NVIDIA GPUs. The shader cores are connected by an interconnection network with memory controllers.

Inside a shader core, a SIMD in-order pipeline is modeled. The SIMD width depends on the architecture that is to be modeled. The pipeline has six logical stages, including instruction fetch, decode, execute, memory1, memory2 and write back. Thread scheduling inside a shader core does not have overhead. Different warps are selected to execute in a round robin sequence. The warp encountering a long latency operation is taken out of the scheduling pool until the operation is served.

Memory requests to different memory space are also modeled. For off-chip access, or access to global memory, the request goes through an interconnection network which connects the shader cores and the memory controllers. The nodes of shader cores and memory controllers have a 2D mesh layout.

1.2.2 Simulation Flow

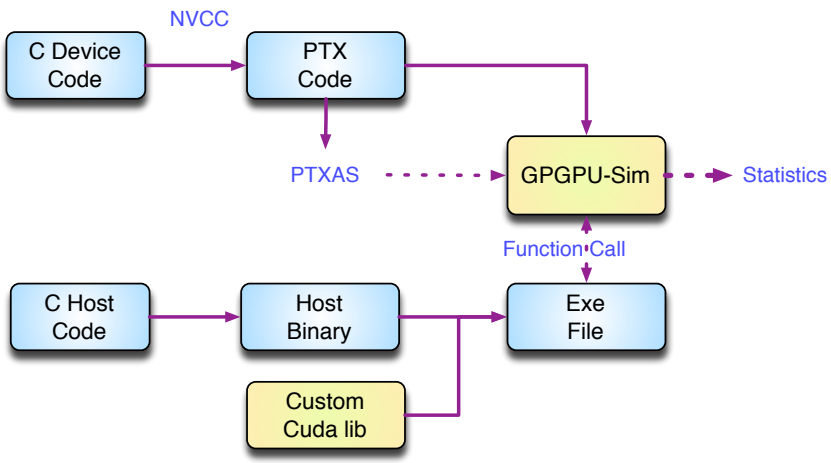

Figure 1.5: Simulation of CUDA Application with GPGPU-Sim

GPGPU-Sim simulates the PTX instruction set. The Figure 1.5 illustrates the simulation flow of GPGPU-Sim. Different from a normal CUDA application compiling and execution, the host binary is linked with custom CUDA library, which invokes the simulation for each device kernel call. The device code is first compiled into PTX code by nvcc. The PTX code serves as the simulation input. The assembler ptxas provides the register usage information to GPGPU-Sim since the register allocation happens when PTX code is compiled into device binary. Then GPGPU-Sim utilizes this information to limit the number of concurrent threads. PTX is a pseudo instruction set and the PTX code does not execute on the actual device. To improve the simulation accuracy and also reduce maintaining effort, a super set of PTX called PTXPlus is designed. PTXPlus has the similar syntax as PTX and can be converted from the assembly code, which can be get from the NVIDIA dis-assembler.

1.2.3 Accuracy

The intention of GPGPU-Sim is not to accurately model any particular commercial GPU but to provide a foundation for architecture researchers. The even though the baseline models can be configured according to one specific GPU model, the modeled architecture is only similar to the actual GPU architecture. In the latest manual of GPGPU-Sim [90], the authors provided a comparison between the simulated execution time with the calibrated GPGPU-Sim, and the actual execution time on the GT200 GPU and Fermi GPU. In terms of IPC (Instructions per Clock), for the Rodinia benchmark suite [19] and using the native hardware instruction set (PTXPlus), GPGPU-Sim obtains IPC correlation of 98.3% and 97.3% respectively. However, the average absolute errors are 35% and 62%.

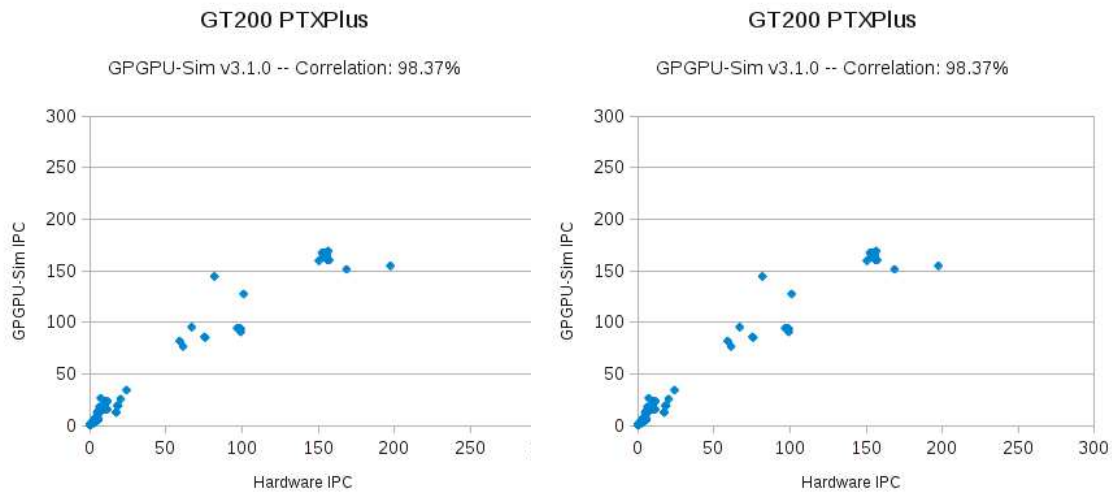

Figure 1.6: Correlation Versus GT200 & Fermi Architectures (Stolen from GPGPU-Sim Manual)

1.2.4 Limitations

The main limitation for simulation approach is that since vendors disclose very few hardware details, it is very difficult to build an accurate simulator for an existing GPU model. And it is very unlikely to build an accurate simulator for the new GPU generation. The baseline architecture model may differ very much from the real hardware characteristics. The accuracy of the simulator cannot be guaranteed without enough hardware details. It is not safe to draw the same conclusion on a real architecture with the result obtained on the simulation baseline architecture. Thus, it is better to use the simulator to explore different architecture configurations. For researchers and developers who study how to improve application performance on existing architectures, the simulator may not be a very good choice. Second, even with an accurate simulator, it is very unlikely for a common developer to use it to understand the performance results and make further optimizations because running a simulation would require a lot of time and long learning curve of the tool.

1.3 Performance Projection/Prediction of GPU Applications Using Analytical Performance Models

For superscalar processors, there is already a rich body of work that proposes analytical models for performance analysis [69, 64, 63, 48, 19, 45, 87, 3, 33]. However, since the general computing on GPU processors is still a fairly new research area, the models and approaches proposed to understand GPU performance results still need a lot of refinement. There exist some interesting works about how to project/predict CUDA applications' performance using analytical or simulation methods. Meng et al. proposed a GPU performance projection framework based

on code skeletons [61]. Bahsork et al. proposed an analytical performance-prediction moel based on work flow graph (WFG), which is similar to the control flow graph [9]. Hong et al. introduced the MWP-CWP model to predict CUDA application performance using PTX code [40]. Recently, Sim et al. extended the MWP-CWP model and utilize the assembly code of CUDA kernel to predict performance [85]. The quantitative GPU performance model proposed by Zhang et al. is also based on the native assembly code [101]. Kim et al. proposed a tool to analyze CUDA applications' memory access patterns [49]. Since very little information about the underlying GPU architecture is disclosed, it becomes very unlikely to build accurate simulators for each new GPU generation. Beside general performance models for GPUs, there also exist some works of model-driven performance optimization for specific kernels[20, 62, 30].

To optimize a GPU application, some general guidelines are provided. Normally developers needs to vary many parameter combinations to find the optimal solution. However, to thoroughly understand the GPU architecture and the performance result of CUDA applications remains difficult for developers. Tools like NVIDIA Visual Profiler [72] can provide stat data from the GPU hardware counter, such as the number of coalesced global memory access, the number of uncoalesced global memory access and the number of shared memory bank conflict. Normally programmers rely on this kind of tool to optimize their cuda applications. For example, if many global memory accesses are coalesced, the global memory access pattern might need to be carefully redesigned. However, the information that the profiler provides very few insights into the performance result.

Although simulation approach for certain architectures is available [35], it is not realistic for developers to use simulators to optimize applications since it is very time consuming. What developers need the most is a tool or an approach that does not require a long learning curve and still provides much insight into the performance result. The analytical approach fits this requirement. Generally, analytical GPU performance model does not need all the hardware details but only a set of parameters that could be obtained through benchmarking or public materials. Apparently, analytical approach cannot compete with the simulation approach for accuracy. Luckily, the performance prediction results of existing analytical performance models [61, 40, 85, 101, 26] show that we can have still very good approximation of GPU performance.

The rest of this section includes several recent analytical performance models for GPU and a brief summary.

1.3.1 MWP-CWP Model

In 2009, Hong et Kim [40] introduced the first analytical model for GPU processors to help to understand the GPU architecture or the MWP-CWP model. The key idea of their model is to estimate the number of parallel memory requests (memory warp parallelism or MWP. According to their reported result, the performance prediction result with their GPU performance model has a geometric mean of absolute error of 5.4% comparing to the micro-benchmarks and 13.3% comparing to some actual GPU applications.

The authors claimed that memory instructions' latency actually dominates the execution time of an application. In the paper, two main concepts are introduced to represent the degree of warp level parallelism. One is memory warp parallelism or MWP, which stands for the maximum number of warps that can access the memory in parallel during the period from the

cycle when one warp issues a memory request till the time when the memory requests from the same warp are serviced. This period is called one memory warp waiting period. The other is computation warp parallelism or CWP, which represents how much computation could be run in parallel by other warps while current warp is waiting for memory request to return the data.

When CWP is greater than MWP, it means that the computation latency is hidden by the memory waiting latency and the execution time is dominated by the memory transactions. The execution time can be calculated as 1.1. The $Comp\_p$ is the execution cycles of one computation period.

$$Exec\_cycles = Mem\_cycles * \frac{N}{MWP} + Comp\_p * MWP \tag{1.1}$$

Actually, if we compare the two parts of the $Exec\_cycles$, the $Comp\_p * MWP$ part is a small number comparing to the memory waiting period. The other part $Mem\_cycles * \frac{N}{MWP}$ can be simply interpreted as the sum of N warps' memory accessing latency parallelized by NWP channels. Thus we can simplify the conclusion as when CWP is greater than MPW or there is not enough memory accessing parallelism, the execution time is dominated by the global memory access latency and can be calculated as the memory accessing time of one warp multiplies the number of active warps and then divided by the degrees of memory access parallelism.

When MWP is greater than CWP, it means that the global memory access latency is hidden by the computation latency and the execution time is dominated by the computation periods. The total execution time can be calculated as 1.2.

$$Exec\_cycles = Mem\_p + Comp\_cycles * N \tag{1.2}$$

Similarly, if we compare the two parts of the $Exec\_cycles$ in this case, the Mem_p part is relatively a small value when each warp has many computation periods. The other part $Comp\_cycles * N$ can be interpreted as the sum of N warps' computation demand since the computation part of all N active warps cannot be parallelized. So we can draw a simpler conclusion as when MWP is greater than CPW or there is enough memory accessing parallelism, the execution time is dominated by the computation latency and can be calculated as the computation time of one warp times the number of active warps.

1.3.1.1 Limitations

The MWP-CWP model is the first analytical model introduced for GPU performance modeling and becomes the footstone of many later GPU performance models. It provides some interesting insight to understand the GPU performance result.

However, the model is too coarse grain since it simply separate an execution of an application into computation period plus the memory access period. The computation period is the instruction issue latency multiplies the number of instructions. The memory access period is a sum of all the memory access latency. Firstly, the model is too optimistic about the instruction level parallelism to calculate the computation period. Secondly, the model assumes memory transactions from one warp are serialized, which is not true. The performance model essentially

uses the ratio of computation time to the memory access latency to define whether the execution time is dominated by the memory access latency or the computation latency. The analysis takes an application as a whole entity. However, for many applications, the execution may have difference characteristics in different parts. For example, in some parts, the application may mainly load data from global memory and in other parts, it may mainly do the computation.

The model for uncoalesced global memory access is too rough. In the model, the uncoalesced global memory accesses are modeled as a series of continuous memory transactions. However, the changed pressure on memory bandwidth is not considered. Plus, the shared memory is not specially treated in the model. To effectively utilize shared memory is essential to achieve good performance on GPU. The model takes the shared memory access instruction as a common computation instruction. The behavior of shared memory access is very complicated. In one shared memory access instruction, if multiple threads within one warp access the same bank, it may introduce bank conflict. The bank conflict normally has a significant impact on ther performance. The new memory hierarchy like unified cache is not considered in the model either.

The model uses the PTX code of an application as the model input. Since the PTX code needs to be compiled into the native machine code to execute on the GPU hardware, it introduces some inaccuracy.

1.3.2 Extended MWP-CWP Model

Recently, Sim et al. [85] proposed a performance analysis framework for GPGPU applications based on the MWP-CWP model. This extended model includes several main improvements over the original MWP-CWP model. First, instruction level parallelism is not assumed to be always enough and the memory level parallelism is not considered to be always one. Second, it introduces the cache modeling and the modeling of the shared memory bank conflict. Third, the MWP-CWP model only utilizes information from PTX code. The extended model uses the compiled binary information.

The extended model requires a variety of information including the hardware counters from an actual execution. To get these information, a front end data collector was designed. The collector the CUDA visual profiler, an instruction analyzer based on Ocelot [36], a static assembly analysis tool. After the execution, the visual profiler provides stat information like number of coalesced global memory requests, the DRAM reads/writes, and cache hits/misses. The instruction analyzer mainly collect loop information to decide how many times each each loop is executed. The static analysis tool is used to obtain ILP (instruction level parallelism) and MLP (memory level parallelism) information from binary level. ILP or MLP obtained by the static analysis tool represents the intra-warp instruction or memory level parallelism.

The total execution time T_{exec} is a function of the computation cost T_{comp}, the memory access cost T_{mem} and the overlapped cost $T_{overlap}$, and defined as Equation 1.3. T_{comp} represents the time to execute computation instructions (including the memory instruction issuing time). T_{mem} is the amount of time of memory transactions. $T_{overlap}$ represents the amount of memory access cost that can be hidden by multithreading.

$$T_{exec} = T_{comp} + T_{mem} - T_{overlap} \tag{1.3}$$

T_{comp} includes a parallelizable part $W_{parallel}$ and a serializable part W_{serial}. The serializable part W_{serial} represents the overhead due to sources like synchronization, SFU resource contention, control flow divergence and shared memory bank conflicts. The parallelizable part $W_{parallel}$ accounts for the number of instructions executed and degree of parallelism.

T_{mem} is a function of the number of the memory requests, memory request latency and the degree of memory level parallelism.

$T_{overlap}$ represents the time that T_{comp} and T_{mem} can overlap. If all the memory access latency can be hidden, $T_{overlap}$ equals to T_{mem}. If none of the memory access can be overlapped with computation, $T_{overlap}$ is 0.

1.3.2.1 Limitations

The main improvements of the extended MWP-CWP model over the original MWP-CWP model include firstly, runtime information like the number of shared memory bank conflict and DRAM hits/misses is collected using Visual Profiler. Thus the shared memory bank conflict effect and the cache effect are introduced in the model. Secondly, the assembly code is served as the model input. Thus the instruction level parallelism can be correctly collected.

However, the model requires an actual execution of the program to collect the runtime information, which makes performance prediction less meaningful. The bandwidth effects of uncoalesced global memory accesses and shared memory bank conflict are still not included the model. The bad memory access is only considered to have a longer latency. The modeling of the shared memory access is still too simple since only bank conflict behavior is considered in the serial overhead W_{serial}. Even though the memory level parallelism and instruction level parallelism are calculated using the assembly code, since the two metrics is for the whole application, the model is still too coarse grain to catch the possibly varied behavior of different program sections.

1.3.3 A Quantitative Performance Analysis Model

In 2011, Zhang et Owens proposed a quantitative GPU performance model for GPU architectures [101]. The model is built on a microbenchmark-based approach. The author claims that with this model, programmers can identify the performance bottlenecks and their causes and also predict the potential benefits if the bottlenecks could be eliminated by optimization techniques.

The general idea of their proposition is to model the GPU processor as three major components: the instruction pipeline, the shared memory, and the global memory and to model the execution of an GPU application as instructions being served to different components based on the instruction type. With the assumption that non-bottleneck components is covered by the bottleneck component, the application bottleneck is identified by the component with the longest execution time.

As in Figure 1.7, the Barra simulator [24] is used to get the application runtime information, such as how many times each instruction is executed. Then this information is used to generate the number of dynamic instructions of each type, the number of shared memory transactions and the number of global memory transactions. Since Barra simulator does not provide bank

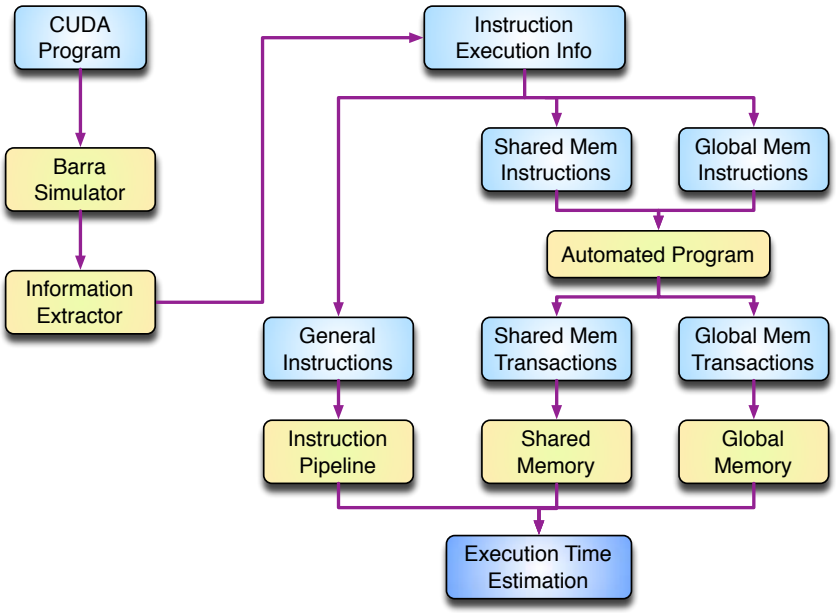

Figure 1.7: Performance Modeling Workflow Proposed by Zhang et Owens

conflict information, the authors wrote automated programs to get the effective shared memory transactions and global memory transactions. A suite of benchmarks are used to build the throughput model for 3 components. The execution time of each component is calculated by the load and the throughput of the component. By comparing the execution time, the performance bottleneck component is identified.

The instruction pipeline component is modeled to execute non-memory instructions. All instructions are classified by the number of functional units that can run the corresponding instruction. The peak throughput of one kind of instruction T_I is calculated as Equation 1.4.

$$T_I = \frac{\#numberFunctionalUnits * \#frequency * \#numberSM}{\#warpSize} \tag{1.4}$$

The theoretical peak throughput of the shared memory is calculated using the number of SPs and the processor frequency. For different number of active warps, a set of benchmarks are used to measure the throughput. To collect the bank conflicts information, an automated program was developed to get the effective number of shared memory transactions with the bank-conflict degree of each shared memory access.

Since all the SMs share the global memory and 3 SMs share a single memory pipeline on GT200 generation GPUs, the global memory is not modeled independently as the instruction pipeline and shared memory. The authors claimed that the global memory bandwidth is sensitive by three parameters, the number of blocks, threads per block and memory transactions per thread. A set of benchmarks varying these three parameters are developed to catch the global memory behavior.

1.3.3.1 Limitations

Zhang et Owen's quantitative GPU performance model is throughput-oriented. The GPU is essentially considered to be 3 major components and each component is modeled with only throughput information based on a set of benchmarks. The input of the model comes from the functional simulator Barra and is composed of 3 different kinds of loads on each component. The execution time of each component is calculated separately and the bottleneck component is the one with the longest execution time.

There are several limitation of this quantitative performance model. First, the model simply divides the GPU processor into 3 separate components. The execution time of each component is separately calculated. However, in the real execution of a GPU application, the math instructions and the memory instructions have complex dependence and the execution of different components have impact on each other. Second, in the model, there is no ILP (instruction level parallelism) and MLP (memory level parallelism) information. Apparently, the throughput depends on the ILP and MLP. If the application to be modeled has different ILP or MLP with the benchmarks, the performance prediction would not be accurate. Third, the component is modeled only with throughput information and no latency information is included in the model. And also, to model the global memory behavior, in the benchmarks, the 3 major parameters include the number of blocks, threads in a block and number of memory transactions per thread. We believe that the percentage of the memory instructions should also be considered in the benchmarks to get the proper bandwidth parameter.

1.3.4 GPU Performance Projection from CPU Code Skeletons

There exist several tools which can produce GPU code from an annotated legacy code or code template [95, 56, 13, 92, 51, 91]. The most recent work is proposed in 2011 by Meng et al. [61]. Meng et al. proposed a GPU performance projection framework, GROPHECY, based on annotated code skeletons. The authors claim that this framework can estimate the performance benefit of GPU applications without actual programming or hardware. The framework allows developers to estimate achievable performance from CPU code skeleton. The automatically transformed code layouts are used to depict structures of the corresponding GPU code and then to project performance for a given GPU architecture. The measured performance of manually tuned codes and the codes generated by GROPHECY have a difference of 17% in geometric mean.

A code skeleton is an abstraction of the CPU code structure and serves as the input for code transformation. After construction, the skeleton can be transformed into different code layouts to mimic GPU optimizations. The transformed code can be significantly different from the original CPU code. The code skeleton's expression include *data parallellism*, *task*, *data accesses*, *computation instructions*, *branch instructions*, *for loops*, *streaming loops* and *macros*.

The GPU performance projection framework includes three major steps to estimate the optimized performance. The first step is to abstract the CPU legacy code and form the annotated CPU code skeleton with the skeleton's expression. The user just needs to extract the parallelism, computation intensity and data accesses from the legacy code with the annotations. So

the user does not necessarily have the GPU knowledge. In the second step, the framework automatically explores the GPU design space by transforming the code skeleton, spatially and temporally. Each transformed code layout corresponds to a GPU implementation. In the last step, each transformed code layout is used to characterize one GPU implementation. The synthesized characteristics are served as inputs to an GPU performance model to estimate the corresponding implementation's performance. The performance model used in the last step is similar to the MWP-CWP model [40].

1.3.4.1 Limitations

Essentially, this proposition is using the MWP-CWP model to predict GPU performance. Unlike the proposition of Hong et al. [40], the code skeleton method does not generate the real GPU code, but only use the characteristics collected from the transformed code layout, which corresponds to an GPU implementation. This solution only adds inaccuracy to the performance prediction results. And like the original MWP-CWP model, instruction level parallelsim is not modeled.

The most obvious limitation of this proposition is that the user need to develop an annotated code skeleton for each legacy code separately. Although the user does not necessarily need the knowledge of GPU programming, the user has to be familiar with the annotation system and extract all the parallelism from the legacy codes. Otherwise the transformation results may not be the optimal. Although the annotation system reduces some programming efforts than programming interfaces like CUDA, it still may take a significant amount of time to well annotate the legacy code.

Like many other automatic code transformation tools, the GROPHECY tool cannot modify algorithms, or the data structures, which, in many cases, are essential to achieve good performance on parallel architectures. Apparently, GROCHECY cannot explore all the design space. It can only provides a good solution based on the annotated code version and the transform options the tool can provide.

1.3.5 Summary for Analytical Approaches

In this section, several important analytical performance models for GPUs are briefly introduced. Comparing with the simulation approach, the analytical approach is much easier to construct and use. Normally the analytical approach only utilizes a set of hardware parameters that are either provided by vendors or able to be collected from benchmarks. For simulation approach, to construct the tool is much harder since it requires a lot more hardware details which are difficult to acquire. Also, it does not require a lot of learning effort and still can provide much information. The analytical approach is easier for programmers to grasp and takes much fewer time to run.

Actually, the simulation approach and analytical approach is not that different. Apparently, the more hardware parameters are introduced in the models and the more underlying implementations are used, the more accurate the analytical models should be. A clear trend is that many recent analytical performance models are trying to utilize the machine code directly and before analytical models normally use algorithm level, C/C++ level or PTX level information.

Using machine codes directly can mimic the GPU execution more closely. Of course, using more underlying details would introduce more complex models, which would be closer to the simulation approach. There is not a clear boundary between analytical approach and simulation approach. Naturally, if we need more accuracy, we use more detailed model and more lower level application implementation.

However, the analytical methods alone cannot get functional information for instruction execution. For example, information like the instruction execution path, the thread masks, or shared memory bank conflict can only be fed in by users or other tools, like simulators or hardware counters. In many cases, programmers want to utilize analytical approaches to understand the penalty of some performance degradation factors. Some analytical tools can provide such information, but only when they are told where and how many these performance degradation events occur.

Generally speaking, from an end user's point of view, we would like the analytical performance models to have the following features. First, an analytical model should be able to be constructed by parameters obtainable. Second, a model should be able to predict the performance of certain implementation with enough accuracy. Third, a model should be able to break down the execution time so that the performance penalties could be quantified.

To satisfy these requirements, we believe that a good performance analysis/prediction tool should be a combination of a functional simulator + an analytical timing tool. To understand the underlying execution status of a GPU application, the input of the analytical model should be the machine code. The functional simulator could be from third party and provides the functional output of the implementation like the shared memory bank conflict events. With these information, we can obtain the exact execution trace, instruction level parallelism and the performance events. The analytical timing tool only consider the timing information and can be developed by parameters from benchmarks. In Chapter 3, we introduce our preliminary implementation of such timing tool.

1.4 Performance Optimization Space Exploration for CUDA Applications

The analytical models can provide some insights into the performance result and the ultimate goal is learn how to achieve better performance of course. Researchers and developers are interested in the outcome of different optimization combinations on GPUs. A very rich body of works study how to optimize specific kernels on GPUs [99, 78, 38, 47, 89, 60, 102, 58, 66, 28, 97, 14, 79]. Similar works are still fast growing. On one hand, this trend shows that many researchers are studying how to accelerate their applications using GPUs and GPU acceleration is effective. On the other hand, it also shows that GPU optimization is still very difficult and needs a lot of application-specific considerations, or at least the existing general auto-tuning methods are not effective enough. Some auto-tuning frameworks for GPU applications are also introduced [70, 46, 37, 25, 27, 100, 31, 62]. Most of the existing auto-tuning frameworks are application-specific, which means that such a framework defines a set of design variables or optimization options and automatically search the best design option in the design space constructed by the defined parameters. Apparently, the developers have to be familiar

with an application and the GPU architectures to build a good auto-tuning tool for the application. Some GPU compiler frameworks are also introduced to help automatically improve the performance [95, 12, 98].

The roofline model [94] is well known for estimating the optimization effects and the idea behind roofline model is actually the base of most auto-tuning frameworks. The recent work by Sim et al. [85], that we have briefly described in the last section, studied the effects of different optimization techniques on GPU using the similar approach as the roofline model. Ryoo et al. summarized some optimization categories and introduced how to better search the optimization space by calculating the efficiency and utilization metrices [80].

Since how to explore the design space is not a main focus of this thesis, we only summarize a few basic ideas proposed.

1.4.1 Program Optimization Space Pruning

Ryoo et al. summarized some optimization categories and introduced how to better search the optimization space by calculating the efficiency and utilization metrics [80]. The search space is pruned with a Pareto-optimal curve generated by the metrics. According to the authors, the exploration space can be reduced up to 98% of the who design space without missing the best configuration. To use the metrics, the global memory bandwidth cannot be the bottleneck for the performance.

The $efficiency$ metric of a kernel indicates the overall instructions need to be executed as in Equation 1.5. $Instr$ is the number of instructions need to be executed per thread and $Threads$ represents the total number of threads. In a nutshell, the fewer overall instructions need to be executed, the higher efficiency the optimization configuration achieves.

$$Efficiency = \frac{1}{Instr * Threads} \tag{1.5}$$

The $utilization$ metric represents the utilization of the compute resources on GPU considering the existence of blocking events and can be calculated as in Equation 1.6. $Regions$ is the instruction intervals determined by blocking instructions. So $\frac{Instr}{Regions}$ indicates the average number of instructions within a non-blocking code region in a warp. W_{TB} is the number of warps in a block and B_{SM} is the active blocks per SM. The $utilization$ metric actually stands for the work available to other warps when a warp is stalled by blocking events.

$$Utilization = \frac{Instr}{Regions}[\frac{W_{TB} - 1}{2} + (B_{SM} - 1) * W_{TB}] \tag{1.6}$$

For an implementation of a specific kernel with a given input size, we can calculate the efficiency and the utilization metrics. It is straight forward that the configuration with both high efficiency and utilization should achieve good performance. If we plot the metrics of optimization configurations and each axis stands for one metric, the configurations in the upper right corner of the graph should have good performance. The authors choose the configurations that have no superior in efficiency or utilization metric and construct the Pareto-optimal subset. For the benchmarks evaluated, the Pareto-optimal subset contains the best configurations. So we can search only the configurations in the Pareto-optimal subset instead of exhaustively searching the whole design space.

1.4.2 Roofline Model

Williams et al. proposed the roofline model to provide insights into the performance optimization choices [94]. The model essentially utilize the operational intensity to represent an application's characteristic. The operational intensity is a term meaning the operations per byte of DRAM traffic. The memory traffic considered in the operational intensity only refers to the traffic between the cache hierarchy and the main memory.

The roofline model visualize the relationship between the floating-point performance, operational intensity ($Operational\_Intensity$) and the memory performance in a 2D graph. The x-axis represents the operational intensity. The y-axis stands for the floating-point intensity. The peak floating-point peak performance (P_{peak}) and the peak memory bandwidth (B_{peak}) are from the hardware specifications or benchmarks. The achievable upper bound (P_{upper_bound}) of a kernel is calculated as Equation 1.7.

$$P_{upper\_bound} = min(P_{peak}, B_{peak} * Operational\_Intensity) \tag{1.7}$$

The for a given architecture, roofline model defines two upper-bound limits. One is bounded by the peak floating-point performance and the other is bounded by the peak memory bandwidth. In a graph, the two limits are two straight lines, and intersect at a ridge point with the peak arithmetic performance and peak memory bandwidth. The two limit lines actually form a roofline shape figure. If the ridge point is far to the right, it means that on the architecture, only kernels with very high computational intensity can achieve the maximum floating-point performance. If the ridge point is far to the left, it means that most kernels can potentially reach the peak floating-point performance. For a given kernel, from a point on the x-axis with the kernel's operational intensity, we can draw a vertical line and the intersect point of this line with the roofline is the upper-bound performance of the kernel.

To address the effects of different optimizations, the roofline model adds more 'ceilings' to the graph. Each ceiling corresponds to one optimization. Higher ceilings imply lower optimizations, which means that to break a ceiling, one needs to break all the ceilings below. The gap between ceilings apparently represents the potential optimization reward and suggest whether the optimization is worth the effort. The lower ceilings normally represent the optimizations likely to be realized by the compilers or relatively easy to implement for programmers.

1.4.3 Summary

For parallel programs, the design space is much larger than serial programs since there are more hardware and software variables. Although normally there are some optimization experience for each parallel architecture, which normally is the first thing developers need to get familiar with before actual optimization, the design options are still way too many to explore. The proposition by Ryoo et al. can prune the search space by calculating the efficiency and the utilization metrics. Given a set of optimization options and configurations, the proposition can help to narrow the search space into a smaller set. Then developers only need to test the configurations within the set.

However, essentially, the proposition considers operations over throughput. The efficiency is simply the number of operations (instructions) needed to finish the kernel. The utilization

is the number of operations (instructions) could be continually executed when one warp is blocked. For GPU architectures, different kinds of instructions have various throughput. Using only the number of operations as the metric is not precise. Also, the proposition misses the important instruction behaviors like instruction level parallelism, shared memory conflicts or uncoalesced global memory accesses. These behaviors are essential to performance optimizations on GPUs. Apparently, the two metrics are not enough to catch all the performance behaviors on GPUs.

The roofline model defines an performance upper bound for applications and also visualize the potential gain of different optimizations. By comparing the gaps between different ceilings, programmers have an estimation of whether the optimization is worthwhile and how much gap is between the current implementation with the peak performance. However, from section 1.4.2, we can see that the upper bound drawn by the roofline model is too optimistic. The upper-bound point reaches either the peak floating-point performance or the peak memory bandwidth, which is apparently too loose estimation. In the many-core era, local fast memory optimization is critical to achieve good performance. However, the roofline model does not account for features like caches, local stores or prefetching. The operational intensity may change with different optimization options to the memory accesses. Using the operational intensity to define an application or a kernel is not accurate. In the model, different optimizations are studied separately. However, in the real-world performance optimization, it is difficult to quantify the effect of a certain optimizations. Different optimizations normally have complex impact on each other. Finally, only bandwidth parameters are considered and no latency information is introduced in the model.

We believe that performance upper-bound analysis is very important in the multi-core era. Although it is easier than simulation approaches, it can still provide interesting insight into the critical system or application parameters. For example, Amdahl's law is probably the most well-known example of such upper-bound analysis [6]. The Amdahl's law itself is not difficult to understand that the performance gain of a parallel program is bounded by its serial part. But it can give much insight even today [39, 7, 82]. Both the proposition by Ryoo et al. and the roofline model provide some kind of upper bound estimation. The proposition by Ryoo et al. and the lower ceilings of the roofline model define local upper bounds. The global upper-bound estimation of the roofline model is too coarse grain. Normally, existing GPU performance tuning frameworks, either automatically or manually, reply on certain level of an application's implementation. The framework first define a few optimization options, apply a few combinations of these defined optimizations, and then check the performance directly or use some metrics to decide whether the configuration is good or not. The ideas are close to the roofline model. Existing analytical approaches do not answer the question of how good the current optimized version is comparing to an achievable peak performance. In the work of Chapter 4, we try to estimate a tight performance upper bound that an application cannot exceed on GPUs. Different from existing approaches which start from a base version and apply optimizations on top of the base version, we try to tackle the problem from up to bottom. We first assume an optimistic situation on GPUs (no shared memory bank conflict, global memory accesses are all coalesced, all the auxiliary operations like address calculations are neglected, etc.). Then we try to predict a performance upper bound when mapping an application on the GPU based on the constraints introduced by the architecture, the instruction set and the application itself,

or the constraints that we are not able to eliminate using optimization techniques. With a tight performance upper bound of an application, we have an evaluation on how much optimization space is left and can decide the optimization effort. Also, with the analysis, we can understand which parameters are critical to the performance and have more insights into the performance result. Hence, with these knowledge, it would be easier for the community to move to the new architecture.

Chapter 2

Data-flow Models of Lattice QCD on Cell B.E. and GPGPU

2.1 Introduction

The study presented in this chapter was done at the beginning of the thesis. The IBM Cell B.E. was then canceled by IBM without any successor. As the thesis was funded through the ANR PetaQCD project, a QCD code base was used for this study.

Lattice QCD simulation is one of the challenging problems for high performance computing community. Because of the extreme computing power needed for the simulation, many supercomputers [21] have been built and highly optimized software tools have been developed. While many of previous generations were relying on special purpose systems [18, 16], current trend is to use off-the-shelf processors due to increasing cost of chip development.

The goal of this part of the thesis work is to provide analytical performance models of Lattice QCD algorithm on multi-core architecture. The Hopping_Matrix computation kernel constitutes about 90% of the computation of the application. Therefore our modelization focuses on this kernel. The models are used to locate critical hardware and software hotspots. The ultimate goal is to understand the application's behavior on different architectures, to find a new modeling methodology to explore the potential performance of multi multi-core machine, and then to guide the performance optimization and hardware design. First, two multi-core architectures, GPGPU and CELL B.E. processor , are studied and the hardware abstractions are proposed; second, the analytical data-flow models for the computation and communication requirements of the Hopping_Matrix kernel are developed, and the potential performance of the critical kernel on the two architectures is estimated. The data-flow model proposed in this chapter is a preliminary one. In the second part of the thesis work, an approach mixing analytical model and simulation-like method is developed to estimate the performance more precisely.

The rest of this chapter includes four parts; Section 2.2 is the analytical models of two hardware platforms of current interest for the consortium, 2.3 is the analysis of the Lattice QCD Hopping_Matrix routine, Section 2.4 is the preliminary performance analysis based on these models, and the last section is the summary.

2.2 Analytical Data-flow Models for Cell B.E. and GPGPU

The architectures that we have studied include GPGPU and Cell Broad Band Engine processor, which have grasped the attention of the lattice QCD community in recent years. The future candidates include x86 multi-core processor and Blue Gene/P. Previous work of the community mainly focuses on optimization of Lattice QCD kernel, on either Cell B.E. processor [43, 42, 86, 65, 10], GPGPU [44, 32, 22, 23, 84, 4], Blue Gene/P [93] and other architectures. Very few works provide analytical insights to the problem [15, 17]. The goal of our model is not to provide accurate performance prediction, but to provide an analytical insight for system designers to choose underlying architecture and for software developers to find system bottleneck. The performance evaluation is used in an early system design stage when there may be several algorithms and several hardware architectures to choose. We try to provide uniform and concise models for lattice QCD on different multi-core architectures since it would be difficult to distinguish the key differences if the model for each architecture is too detailed. And we try to find the similarities of different platforms first and then locate the key differences, which may affect the lattice QCD performance.

Lattice QCD is apparently a data-centric or memory bound application, and operations on huge amount of data are very much alike. All these operations are mostly arithmetic operations and branch effects can be neglected. So we developed data-flow models for the application on different architectures. Our focus is to research the data flow between different functional units inside multi-core processor and estimate data bandwidth requirement on the interconnections. Here we consider the data as one kind of the data flow. The idea is very straightforward. By examining the throughput of different units, we can estimate the system performance and locate performance bottleneck.

The rest of this section illustrates our data-flow-oriented architecture abstraction for Cell B.E. processor and NVIDIA GPUs (GT200 and Fermi).

2.2.1 Cell Processor Analytical Model

As depicted in Figure 2.1, the Cell BE Processor is a heterogeneous processor with one PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). The PPE with 64-bit PowerPC Architecture core runs the operating system and controls the execution of all the SPEs. The eight SPEs are in-order single-instruction multiple-data (SIMD) processor elements optimized for compute-intensive work. Each SPE has 256KB local memory for instructions and data, and a register file of 128 128-bit registers. Each SPE has two instruction pipelines and can issue up to two instructions each cycle. The peak instruction throughput is 4 single-precision or 2 double precision fused multiply-add operations per SPE per cycle, or 204.8 GFlops single precision or 102.4 GFlops double precision peak performance of the whole processor at 3.2GHz(the computing power of the PPE is neglected).

All these processor elements, main memory interface and I/O interface are connected by the element interconnect bus (EIB). The EIB transfers data between processor elements, the main memory and the IO interface. At 3.2GHz it could offer a theoretical peak bandwidth up to 204.8 GB/s. Data transactions between the SPEs local memory and the main memory are via DMA operations. The DMA operation supports aligned transfer size of 1, 2, 4, 8, and 16

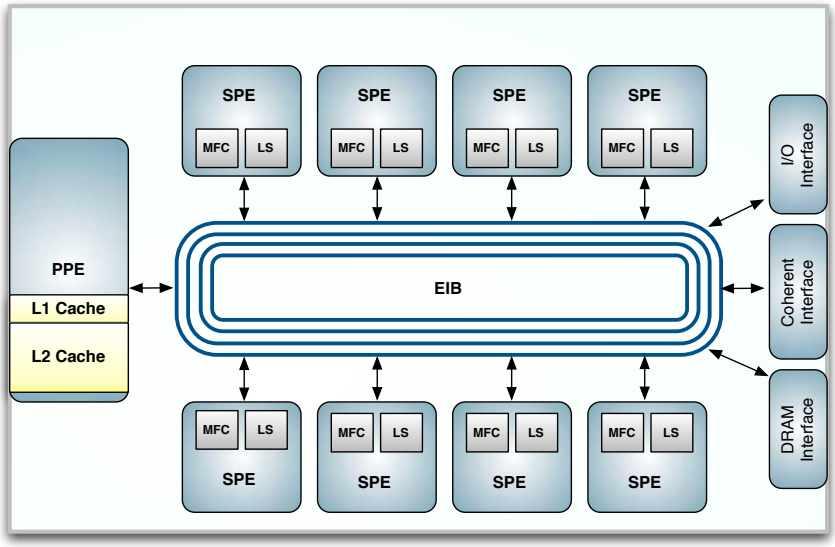

Figure 2.1: Cell B.E. Block Diagram

bytes and multiple of 16 bytes and can move up to 16KB at a time. With the double-buffer techniques, the DMA transfer latency could be covered by the application execution. The memory interface is XDR memory controller for Cell on QS20 Cell blade or DDR memory controller for Cell on QS22 Cell blade, which can provide up to 25.6 GB/s memory bandwidth.

The data flow is defined as the data movement between different functional units of the processor. This is how we decide the basic building blocks of the processor's analytical data-flow model. The abstraction of Cell B.E. processor with data flow is illustrated in Figure 2.2. R1 to R4 are data paths and F1 to F5 stand for data flows. Basically, the modeled Cell processor is separated into SPEs, main memory, the I/O interface and the EIB.

Data processing or the instruction execution inside the SPE corresponds to flow F1. There are different patterns of distributing workloads on SPEs inside Cell processor, either in serial pattern or in parallel pattern. In either way, different SPEs can communicate through local store. Data can be directly transferred from one SPE's local store to another SPE's local store, corresponding to flow F4.

Field data is loaded from main memory, through EIB to local store and final result needs to be written back to main memory since the data is too large to fit into local store. That corresponds to flow F2. Fourth, since the scale of the Lattice QCD problem is very large, we need thousands of processors to cooperate. And because of the nearest neighbor communication nature of the Lattice QCD algorithm, there is much traffic between Cell processors. Because SPE can issue I/O operations directly, the communication can go directly from local store to local store on another Cell processor. However, dedicated communication hardware is needed as interface [10]. Another option would be store the data back to main memory first and then send the data through I/O interface. However, the last option will increase the pressure on main

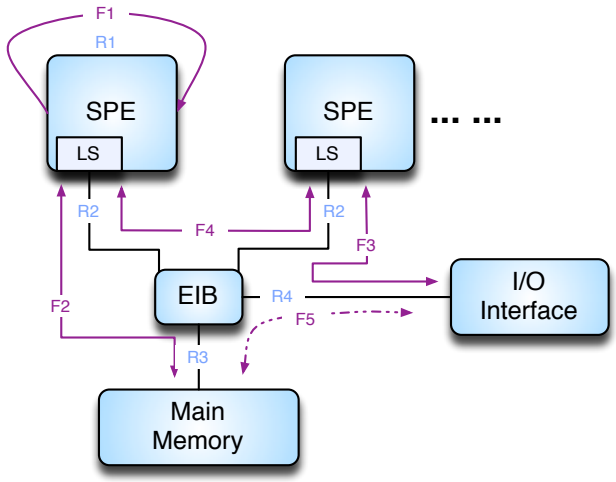

Figure 2.2: Analytical Model of Cell Processor with Data Flow

memory bandwidth, which is considered as one of the bottlenecks of the system. These two options correspond to flow F3 and flow F5.

A summary of the data-flow legends are listed in Table 2.1.

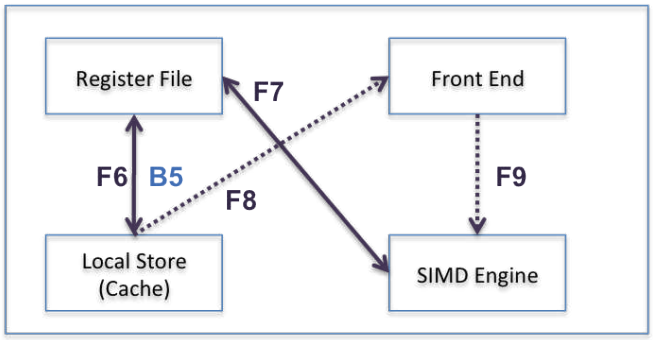

Figure 2.3: Analytical Model of SPE with Data Flow

As we can see from Figure 2.2, we partition the Cell processor according to the basic units responsible for data movement, that is the SPE, main memory, I/O interface and EIB.

If we look more closely on what is happening inside the SPE, we can further break the SPE down to smaller parts illustrated in Figure 2.3. The new parameters include the flow F6, which means the data load and store between register file and local store.

From the analysis above and the goal to derive the data flow information, we can see the main factors that can influence the data flow include how different units are connected, the behavior of the application, how much private and share resources present at each core, and of

Data Flow & Control Flow	
F1	Data processing of each SPE
F2	Data traffic between SPE's local store and main memory
F3	Data traffic between different Cell processors, direct communication between local stores
F4	Data traffic between local stores inside one Cell processor
F5	Data traffic between different Cell processors, communication through main memory
F6	Registers load from and store to local store
F7	Fetch data from and write back to register file
F8	Fetch instruction from local store
F9	Instruction control flow
Bandwidth	
B1	Data processing throughput
B2	Bandwidth between local store and EIB
B3	Main memory bandwidth
B4	Bandwidth to I/O device
B5	Bandwidth between register file and local store

Table 2.1: Legends of Cell Data-flow Model

course the bandwidth or throughput of interconnections or units.

2.2.2 GPU Analytical Model

GT200 GPU is composed of 10 TPCs (Thread Processing Cluster), each of which includes 3 SMs (Streaming Multiprocessor). Each SM further includes 8 SPs (Streaming Processor) and 2 SFUs (Special Function Unit).

The analytical model of GT200 GPU is illustrated in Figure 2.4. Since 3 SMs inside one TPC share the same front end and memory pipeline, we consider TPC as the basic processor core.

As in Figure 2.4, the basic building blocks include each TPC, the graphic memory, the main memory, and I/O interface. Each TPC is connected to graphic memory through GDDR3 memory controller, which is neglected in the model. Both graphic memory and I/O device are connected to the main memory. The resource we consider carefully here includes the share memory and the register file in the TPC.

Figure 2.5 is the detailed model of each TPC. Different from SIMD engine of Cell processor, GPU's computation scheme is so called SIMT (Single Instruction Multi Thread).

The analytical models for GT200 and Fermi are similar. First, data is processed inside each processor core (TPC or SM), corresponding to flow F1. Second, since the SIMT nature of GPU and resource like register file is specific to each thread, it is difficult to communicate inside one GPU. Threads within one block can exchange information through share memory and threads belonging to different blocks can only communicate through graphic memory. Because the basic block in our model is each processing core, thread architecture is invisible. So there is no

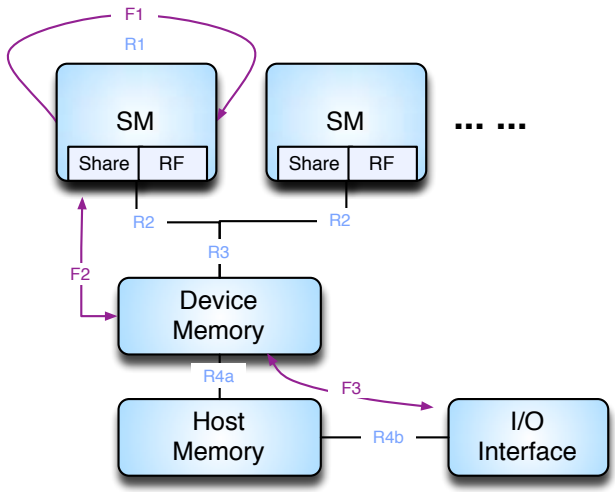

Figure 2.4: Analytical Model of GT200 GPU with Data Flow

explicit inside communication. Communication between threads inside one GPU is treated as accesses to share memory or graphic memory. Third, we assume that graphic memory is large enough to hold field data and intermediate results. On each iteration, field data is loaded from graphic memory and final result needs to be written back to main memory, corresponding to flow F2. So the data traffic to main memory is only related to inter-GPU communication since GPU cannot issue I/O operations directly. Data needs to be transferred to main memory first and then go to I/O interface. This corresponds to flow F3.

2.2.3 Comparison of Two Analytical Models

Figure 2.6 is the comparison of the two models presented. After some simplifications, we try to make the models easy to understand and compare.

Some key differences regarding the Cell and GPU analytical models and their characteristics are presented below. First, GPU has more memory controllers and can provide more memory bandwidth. As we know, Cell processor's memory interface can provide 25.6GB/s peak bandwidth while GPU usually can provide more than 100GB/s peak bandwidth. For some high-end GPUs, the bandwidth can reach more than 170 GB/s. Second, to single computing core, we need to carefully consider the amount of share memory and register file of GPU platform because of the SIMT programming model of GPU. Different threads of the same block can only communicate through the shared memory. On the other hand we need enough threads to get good occupancy. But more threads will lead to reduced per-thread resource, especially the register resource. Third, SIMD core can transfer data through fast memory (local store) while SIMT core has to communicate through the graphic memory, which increases the pressure on bandwidth to main memory.

To sum up the above research, we believe the key differences between the two platforms for

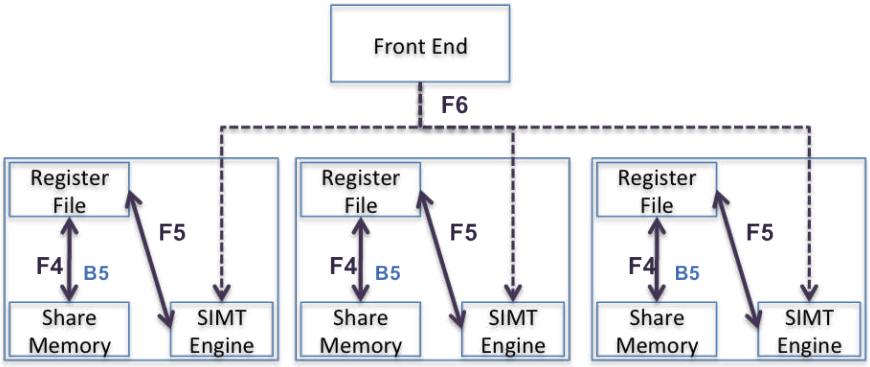

Figure 2.5: Analytical Model of TPC with Data Flow

Data Flow & Control Flow	
F1	Data processing of each processor core (TPC or SM)
F2	Data traffic between processor core and graphic memory
F3	Data traffic between graphic memory and main memory
F4	Registers load from and store to local store
F5	Fetch data from and write back to register file
F6	Instruction control flow
Bandwidth	
B1	Data processing throughput
B2	Bandwidth between each SM and graphic memory
B3	Bandwidth between graphic memory and main memory
B4	Bandwidth between main memory and I/O interface
B5	Bandwidth between register file and share memory

Table 2.2: Legends of GPU Data-flow Model

the Lattice QCD implementations are the differences of memory hierarchy and interconnection pattern of different processor units, which will influence the memory access pattern. The access pattern is the key to data flow requirement and ultimately the key to the performance.

2.3 Analysis of the Lattice-QCD Hopping_Matrix Routine

In this section, we try to derive the data flow requirement based on the architecture analytical model and the algorithm essence. First, we analyze the flow F1 and F2, and their requirement on B1 and B3. In the further research, more detailed analysis will be given. This section includes three parts, the first part is the Lattice QCD Hopping_Matrix analysis, since Hopping_Matrix is the most time-consuming routine in Lattice QCD algorithm and it consumes around 90% of the whole execution time.

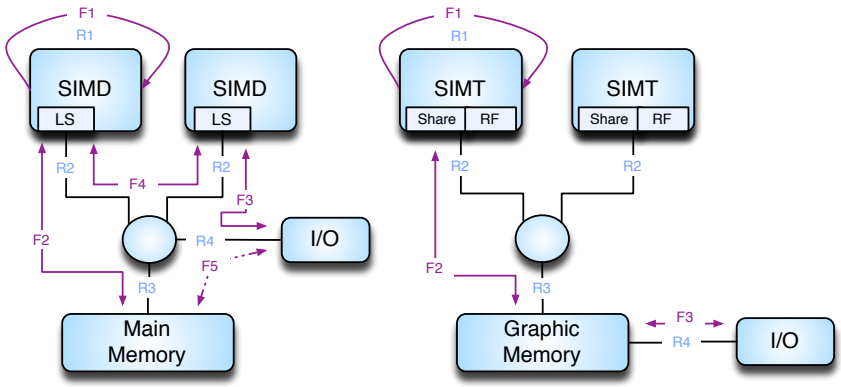

Figure 2.6: Comparison of Cell and GPU Analytical Models

The input data structures of Hopping_Matrix routine include the spinor field, the gauge field, the output is the result spinor field. The temporary data is the intermediate half spinor field. The main function of Hopping_Matrix is the Dirac operator, which is illustrated in the Equation 2.1.

$$D(x,y) = \sum_{\mu=1}^{4}[U_{\mu}^{+}(x+\mu,y)(1-\gamma_{\mu})\delta(x+\mu,y) + U_{\mu}^{\dagger}(x-\mu,y)(1+\gamma_{\mu})\delta(x-\mu,y)] \quad (2.1)$$

A 3x4 complex matrix represents the full spinor residing on each space-time site. The gauge field data residing on each link connecting neighbor sites is represented by 3x3 complex matrix. The half spinor is represented as 3x2 complex matrix, which is the temporary data generated on each of 8 space-time directions for one full spinor.

According to expression of Dirac operator, we can divide the operations into the following steps. First, the input full spinor is converted to intermediate half spinor. This step corresponds to the multiplication of $1-\gamma_{\mu}$ or $1-\gamma_{\mu}$. The conversion can be treated as the addition of complex matrix elements. This yields 12 real number additions. At this step, huge amount of intermediate data is generated. The main design choice of this step is whether the generated data should be stored in the fast memory or has to be stored back into the main memory which typically has much longer latency than local fast memory.

Second, the half spinor field matrix multiplies corresponding gauge field matrix on each of the 8 space-time directions. This corresponds to the multiplication of $U_{\mu}^{+}(x+\mu,y)$ or $U_{\mu}^{\dagger}(x-\mu,y)$. In other words, the operation equals to the multiplication of a 3x2 complex matrix and a 3x3 complex matrix. This operation needs 18 complex number multiplications and 12 complex number additions. That equals to 72 real number multiplications and 60 real number additions.

Third, 8 directions' temporary results are accumulated to the final spinor field. The operation is only simple matrix addition. This operation needs 24 real number additions. In all, there

are 1320 real number operations in the three steps, which corresponds to data flow F1 in the data-flow models.

The pseudo code of Hopping_Matrix routine is illustrated in Listing 2.1.

```
//Loop 1
/*********** 4 positive directions ***********/
Half_Spinor_Positvie <- Full_Spinor
Temp_Positvie = Half_Spinor_Positvie * U_Positive
/*********** 4 negative directions ***********/
Half_Spinor_Negative <- Full_Spinor

Synchronization();

//Loop 2
/********** 4 positive directions ************/
Full_Spinor += Temp_Positvie

/********** 4 negative directions ************/
Temp_Negative = Half_Spinor_Negative * U_Negative
Full_Spinor += Temp_Negative


While(ADDR_B < LOOP_END)
```

Listing 2.1: Hopping_Matrix Pseudo Code

To get further performance analysis result in the following sections, a few assumptions are made. First, there is enough parallelism in the Lattice QCD Hopping_Matrix routine (Matrix operations can be easily parallelized and dependence only exists between nearest neighbors). So arithmetic pipeline could be always full and there would be almost no penalty from branch mis-prediction. Second, when parameters of L and T (L and T represent the space-time dimension of the lattice) are very large, the main data structures (Field data) must be reloaded from main memory on each iteration, because the cache will always be not large enough to hold this data structure. Third, all the data can be perfectly pre-fetched into cache or in other words, the bandwidth between processor and main memory could be fully utilized. Computation and communication can be perfectly parallelized (Need careful programming). These assumptions mean that we neglect the cache influence and focus on the bandwidth analysis.

2.4 Performance Analysis

In this section, we build a simple data-flow model to analyze the potential performance of the Hopping_Matrix routine based on the hardware abstraction and the analysis of the routine. The detailed performance analysis is essentially based on different memory access patterns.

2.4.1 Memory Access Patterns Analysis

As described before, our methodology is to derive the potential performance based on the data flow analysis. With models of the processors and the application, the memory access patterns are summarized and then the data flow information can be generated. Then we can estimate the data bandwidth requirement based on the data flow information. By identifying the bottleneck component, the potential performance of the application is calculated using the component's peak bandwidth .

Using the analytical models presented, we categorize the memory access patterns as in Table 2.3.

$P1$	Reconstruct the gauge field in processor
$P2$	Fully share gauge field data between neighbor space-time sites
$P3$	Intermediate half spinor field data is hold in local fast memory, without the need to be written back to main memory
$P4$	Inter-processor boundary half spinor field data is stored in local fast memory, without the need to be written back to main memory
$P5$	Inter-core boundary half spinor field data is stored in local fast memory, without the need to be written back to main memory

Table 2.3: Memory Access Pattern

Pattern $P1$ is about whether we can reduce gauge field data access in main memory. The gauge field matrix is element of the SU(3) group. We can use fewer real numbers to parameterize the matrix. In practice, we may use only 12 real numbers or 8 real numbers instead of using 9 complex numbers (depend on the implementation). Using 8 real numbers, we can reduce the gauge field access by 10/18. However, with this method, extra computing power is needed. This pattern is applicable only when there is much computing power left in the processor.

Pattern $P2$ is also about whether we can reduce gauge field data access. The gauge field matrices on the same link used by two neighbor space-time sites have a simple relation. If we only store one copy in the fast memory and can process the two neighbors at the same life cycle of the gauge field data, the access of gauge field data could be reduced by half. Whether this pattern can be applied depends on the programming model (SIMD or SIMT) and shared memory size.

Pattern $P3$ is about the intermediate half spinor field data. Since for each space-time site, 8 copies of half spinor field data are generated. If the local fast memory is not large enough to hold them, they need to be written back to the main memory before further processing. Apparently, whether this pattern can be applied depends on the shared fast memory size and how to share data within one processor core.

Pattern $P4$ is about the data exchanged by different processor nodes. Inter-processor boundary half spinor field data is generated on the boundary of each processor's sub lattice and needs to be sent to logical adjacent processor nodes. Inter-processor boundary half spinor field data's size is not negligible. Whether it needs to be written first to main memory depends on the memory hierarchy (how processor cores are connected to the I/O ports) and communi-

cation hardware. The parameter α represents the ratio of inter-processor boundary half spinor field data to whole half spinor field data.

Pattern $P5$ is about the data exchanged by different processor cores. Normally different cores inside a processor node need to exchange boundary half field data. Whether this part of data can be stored in local fast memory and directly communicated also depends on the memory hierarchy. The parameter β is introduced to represent the fraction of inter-core boundary half spinor field data.

In an implementation, all the patterns may not be applied at the same time because of the processor resource constraints. So for different implementations, many combinations of these patterns could be applied. To get best performance on a specific architecture, we could select the best combination regarding the architecture features. In the following part, the requirements of those patterns are carefully studied.

The spinor field data needs to loaded and written back to main memory each at least once. The spinor field matrix is a 3x4 complex matrix occupying 192 bytes. 384 bytes data traffic per space-time site is added to the data flow $F2$. This corresponds to pressure on main memory bandwidth and memory controller bandwidth per processor core.

The gauge field matrix is a 3x3 complex matrix, which occupies 144 bytes. Per space-time site, 1152 bytes are needed for 8 directions. As input data, the gauge field needs to be loaded once on each iteration. If the processor core has much spare computing power, pattern $P1$ could be applied. Pattern $P2$ could be applied only if enough neighbor sites are processed at the same time in one processor core, there is enough local fast memory to store the gauge field and all gauge field data is visible to all threads.

The half spinor field is generated for neighbor sites. So the data may need to be sent to other thread, processor core or processor nodes. Each matrix needs 96 bytes. All 8 directions' data occupies 768 bytes per site. For example, if there are multiple threads per processor core and they can communicate only through share memory inside processor core, the size of the share memory becomes the dominant factor. If the share memory is not large enough, the half spinor field data needs to be written first to main memory. In this case, pattern $P3$ is not applicable. Considering pattern $P4$, suppose that the local fast memory is large enough to hold the inter-processor boundary half spinor field data, whether pattern $P4$ is applicable depends on the interconnection between processor nodes. If there exists a data path connecting fast memory of different nodes, the transfer does not need to go through the main memory. It is similar for pattern $P5$. Whether pattern $P5$ can be applied also depends on the local resource. It also depends on how different processor cores communicate. If different cores can directly communicate through fast memory and local fast memory is large enough, then pattern $P5$ is applicable.

According to different pattern configurations, data flows can be determined. Then bandwidth pressure on different interconnections could be calculated. For example, the main memory accesses, memory accesses from processor core to other parts, and I/O interface accesses. Three parameters are introduced here for further analysis as in Equations 2.2, 2.3, and 2.4.

$$R_{A/M} = \frac{\#Arithmetic\_Operations}{\#Main\_Memory\_Accesses} \tag{2.2}$$

(00000)	0.43	(00001)	$\frac{1320}{3072-1536\beta}$	(00010)	$\frac{1320}{3072-1536\alpha}$	(00011)	$\frac{1320}{3072-1536(\alpha+\beta)}$
(00100)	$\frac{1320}{1536+1536(\alpha+\beta)}$	(00101)	$\frac{1320}{1536+1536\beta}$	(00110)	$\frac{1320}{1536+1536\alpha}$	(00111)	0.86
(01000)	0.53	(01001)	$\frac{1320}{2496-1536\beta}$	(01010)	$\frac{1320}{2496-1536\alpha}$	(01011)	$\frac{1320}{2496-1536(\alpha+\beta)}$
(01100)	$\frac{1320}{960+1536(\alpha+\beta)}$	(01101)	$\frac{1320}{960+1536\beta}$	(01110)	$\frac{1320}{960+1536\alpha}$	(01111)	1.375
(10000)	0.54	(10001)	$\frac{1320}{2432-1536\beta}$	(10010)	$\frac{1320}{2432-1536\alpha}$	(10011)	$\frac{1320}{2432-1536(\alpha+\beta)}$
(10100)	$\frac{1320}{896+1536(\alpha+\beta)}$	(10101)	$\frac{1320}{896+1536\beta}$	(10110)	$\frac{1320}{896+1536\alpha}$	(10111)	1.47
(11000)	0.6	(11001)	$\frac{1320}{2176-1536\beta}$	(11010)	$\frac{1320}{2176-1536\alpha}$	(11011)	$\frac{1320}{2176-1536(\alpha+\beta)}$
(11100)	$\frac{1320}{640+1536(\alpha+\beta)}$	(11101)	$\frac{1320}{640+1536\beta}$	(11110)	$\frac{1320}{640+1536\alpha}$	(11111)	2.06

Table 2.4: Memory Access Pattern Combination ($P1$ $P2$ $P3$ $P4$ $P5$) & Relative Demands on Arithmetic Operations and Main Memory Access ($R_{A/M}$)

$$R_{A/C} = \frac{\#Arithmetic\_Operations}{\#Accesses\_from\_Processor\_Core\_to\_Other\_Parts} \tag{2.3}$$

$$R_{A/IO} = \frac{\#Arithmetic\_Operations}{\#I/O\_Data\_Accesses} \tag{2.4}$$

Table 2.4 includes all the combinations of above five patterns (in the order of $P1$ $P2$ $P3$ $P4$ $P5$) and the corresponding $R_{A/M}$. $R_{A/C}$ and $R_{A/IO}$ can be given in the similar way.

The calculation is like the following. Take the pattern combination (01111) for instance. On each iteration, the spinor field data needs to be loaded once at the beginning and written back once in the end. That yields memory traffic of 384 Bytes. The gauge field needs to be read only once and because of pattern $P2$ the traffic is cut in half. That equals to the data traffic of 576 Bytes. Because of pattern $P3$, $P4$ and $P5$, there is no need to write back any half-field data information the data traffic to main memory per site is 960 Bytes. So $R_{A/M}$=1.375.

Since α and β are related to problem size and data distribution configuration, we need to make an instantiation, and we use the same instantiation in the following analysis. $\alpha = 0.125$.

With the above analysis, we can further analyze the potential peak performance of Hopping_Matrix on the two architectures based on how to apply the combinations of these patterns.

2.4.1.1 Cell Performance Analysis

Each processor core (SPE) on Cell runs a single thread. The local resource, which is visible to the thread, includes the local store of 256 KB and the register file of 2 KB. According to the previous analysis, apparently, the local store can hold the field data of a sub lattice with enough space-time sites. So pattern $P2$ and $P3$ could be applied. Since SPE can issue I/O operations directly, then boundary half spinor field data can be directly transferred without being written back to main memory. So the pattern $P4$ is feasible. Because different SPE can directly communicate through EIB, pattern $P5$ is also feasible. The optimal combination of Cell processor is (01111).

With the pattern combination (01111), $R_{A/M} = 1.375$, $R_{A/C} = \frac{1320}{1536(\alpha+\beta)}$, and $R_{A/IO} = \frac{1320}{1536\alpha} = 6.875$. Clearly we have $P = B3 * R_{A/M} = 35.2$ GFlops. So the potential peak performance for DSlash is around 35GFlops (34% of theoretical peak performance of Cell, 102.4GFlops). Since it is not possible to fully utilize the gauge field data between neighbors, the potential peak will be lower.

2.4.2 GPU Performance Analysis

On each processor core (SM), there reside hundreds of threads. Each thread's registers are private and invisible to other threads. The local resource on each SM that could be explicitly controlled by programmer includes the registers and the shared memory. The shared memory is visible to all threads and can be used to communication and store the global data. Each SM has 32K registers and 48KB or 16KB shared memory. So the maximum local storage amount is 176KB. For GT200 GPU, to hide arithmetic latency, at least 192 threads per SM are needed and to hide memory latency, usually more than 256 threads are needed. We lack the same information for Fermi GPU by far. Using 256 threads, the resource per thread is about 700 Bytes. With this amount, to store all the intermediate half spinor field data is not possible. Because GPU cannot issue I/O operations directly, pattern $P4$ is not possible. There's no direct-communication between cores inside GPU. So $P5$ is also not feasible. Since there is a lot of computation power per GPU, we can consider reconstructing the gauge field data inside the processor. So the possible pattern combination could be (10000).

With the pattern combination (10000), we have $R_{A/M} = 0.54$, $R_{A/C} = 0.54$, and $R_{A/IO} = \frac{1320}{1536\alpha} = 6.875$. However, different GPU models have quite different configuration and computing power. Taking GeForce GTX280 for instance, it has around 140GB/s memory bandwidth. And suppose the PCIE bus can provide 8 GB/s bandwidth (20% overhead because of encoding). In double precision, we have $P = B3 * R_{A/IO} * 80\% = 44$ GFlops, 6.5% of the theoretical peak performance. If only consider single GPU node, the potential performance is $P = B2 * R_{A/M} = 75.6$ GFlops, about 65% of the theoretical double precision peak performance.

2.5 Summary

In this Chapter, we present simple data-flow models for lattice-QCD Hopping_Matrix routine on Cell B.E. and GPU processors. First we analyze the data paths of the two architectures and the computation and data accessing requirements of the Hopping_Matrix. Then we categorize 5 memory-access patterns of the routine. The data flow informations is generated based the memory-access patterns. Thus we can identify the bottleneck functional unit and derive the potential performance. The following is some observations while developing the models and analyzing the performance. From the analysis above, the main factors that can influence the data flow include how different functional units are connected, the behavior of the application itself, how much private and share resources present at each core, and of course the bandwidth or throughput of interconnections and functional units.

First, the Lattice QCD application is highly memory bound. The most effective optimization method is to reduce the pressure to the main-memory data path. And we should choose architecture with large bandwidth to main memory and on which it is easy to hide memory latency. Second, in many cases, there are many accesses to main memory that cannot be reduced because of the large problem scale and local resources constraints. Third, large and fast local storage is needed to store intermediate data and it seems that software-control cache is a better option since we can carefully program the data movement between main memory and fast local storage.

Chapter 3

Performance Estimation of GPU Applications Using an Analytical Method

This chapter presents a study that was presented at the RAPIDO 2012 workshop [53].

3.1 Introduction

The computation power of modern GPU has been increasing dramatically. Nowadays many applications have been ported to GPU architecture with interfaces like CUDA [2] or OpenCL [77]. These programming interfaces lower the entry barrier for GPU application development. However, since these programming interfaces are high level abstractions and few GPU hardware details are disclosed, programmers have little insights into GPU performance result. Generally, programmers have to develop their own experience for GPU application optimization. Although profiling tools such as CUDA Visual Profiler [2] are provided, much efforts are still needed to achieve a good performance and in many cases a large design space needs to be explored.

In last chapter of this thesis, we present simple data-flow models for lattice QCD applications on Cell and GPU architectures. The models can provide some rough insights into the performance bottleneck. However, the analysis is too coarse grain. The fine-grain application behavior cannot be analyzed with that approach. Since GPU platform is likely to be the candidate for the lattice QCD project, the rest of the thesis work focuses on the performance analysis of applications on GPGPUs. The intention of the second part of this thesis work is to provide an analytical approach, that helps to get more insights into GPU performance results.

We developed a timing model for NVIDIA GT200 GPU and constructed the tool TEG (Timing Estimation tool for GPU) based on the model. TEG takes the CUDA kernel assembly code and instruction trace as input. The CUDA kernel binary code is disassembled using tool *cuobjdump* provided by NVIDIA [2]. The instruction trace is obtained by Barra simulator [24]. Then TEG models the kernel execution on GPU and collects timing information. TEG does not execute the instructions directly but only utilizes the dependence and latency

information. With the timing model and the assembly code as input, TEG can estimate GPU cycle-approximate performance. The output of TEG includes the total execution cycles, load on function units, etc. Evaluation cases show that TEG can get very close performance approximation. Comparing with the real execution cycle number, normally TEG has a error rate less than 10%. Especially, TEG has good approximation for applications with very few active warps on SM. Thus we could better understand GPU's performance result and quantify bottlenecks' performance effects. Present profiling tools can only provide programmers with bottleneck statistics, like number of shared memory bank conflict, etc. TEG allows programmers to understand how much performance one bottleneck can impair and forsee the benefit of eliminating the bottleneck.

Several works using analytical methods to analyze GPU performance are presented [40, 9, 49, 101, 61, 85], which are briefly introduced in the first chapter. The main difference between our approach and these works includes that first, in our study, we use the binary code instead of PTX code as input because resource allocation happens at the compiling stage from PTX code to binary code and binary code is the native code running on GPU hardware. Second, we use instruction trace instead of instruction statistics as the tool input, and provide workload information on different function units of GPU. Different from cycle-accurate simulators, we chose a limited set of hardware model parameters, which could be obtained through benchmarks. We assume that we may still have very limited knowledge of the underlying hardware details for future GPGPU architectures.

This chapter is organized as follows: In Section 3.2 we present our timing model for GPU. Section 3.3 demonstrates TEG's workflow. In Section 3.4 we evaluate TEG with two cases. In Section 3.5 we use TEG to analyze GPU performance scaling behavior with a case study. Section 3.6 concludes this part of study and presents future direction.

3.2 Model Setup

In this section, we present an analytical performance model for GT200 GPU and the key parameters. Then we discuss some performance effects that TEG can demonstrate.

3.2.1 GPU Analytical Model

Our model for GT200 GPU is illustrated in Figure 3.1. In our model, each SM is taken as one processor core and the detail instruction pipeline stages and detail memory transaction behavior are not modeled. SM is fed with warp instructions. Inside one SM, there are issue engine, shared memory, register file and functional units like SP, DPU, SFU and LD/ST unit. 8 SPs are considered as one functional unit. Global memory load/store instructions are issued to LD/ST unit.

We define 32 instructions of threads in the same warp as warp instruction. An unmasked warp instruction launches 32 operations. Functional units has two properties, issue rate and bandwidth or throughput. Issue rate decides how many cycles one functional units can accept a new warp instruction, and bandwidth or throughput denotes how many warp instructions can be in flight. Every 2 cycles, the issue engine selects one ready warp instruction from active warps and issues the instruction to the ready functional units according to instruction type. A warp

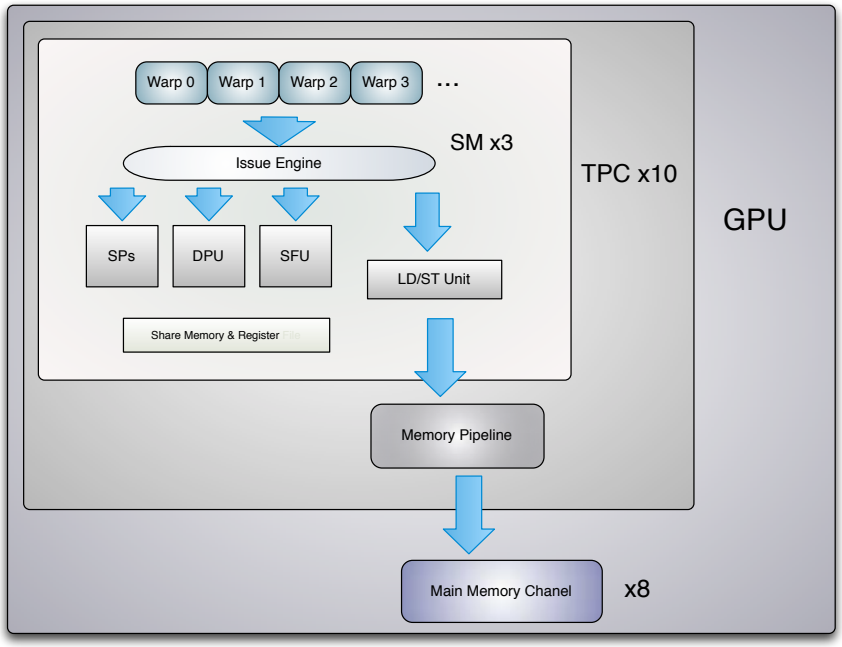

Figure 3.1: GPU Analytical Model

instruction can be be issued when all the source operands are ready. GPU uses a scoreboard mechanism to select the warp with a ready warp instruction. In our model, different scoreboard policies are implemented. For each warp, since instructions are issued in program order, if one instruction's source operands are not ready, all the successive instructions have to wait.

Every three SMs share the same memory pipeline in one TPC, and thus share 1/10 of peak global memory bandwidth. 8 channels connect the device memory chips with the GPU processor and each channel bandwidth cannot exceed 1/8 of peak global memory bandwidth. We do not model the on-die routing of memory requests, since the hardware details have not been disclosed.

Warp instruction has 3 kinds of latency properties (section 3.2.2.1). The latency information decides a warp instruction's life cycle. When there is performance degradation factor, such as the access pattern of one warp instruction leads to shared memory bank conflict, we just simply use the warp instruction's "degraded" latency information. Of course, we also uses the instruction's dependence, operator and operand type information in our model.

3.2.2 Model Parameters

To use the analytical model in TEG, we need to define some model parameters. In this section, some major parameters are introduced. Much work has been done to understand GPU architecture through benchmarking [96]. Some results and ideas are borrowed from this work.

3.2.2.1 Instruction Latency

Execution latency

Execution latency of a warp instruction is defined as the cycles that the instruction is active in the corresponding functional unit. After the execution latency, one issued warp instruction is marked as finished. The typical technique to measure instruction execution latency is to use the clock() function. The clock() function returns the value of a per-TPC counter. To measure instruction execution latency, we can just put dependent instructions between two clock() function calls. An extra 28 cycles is introduced because of the clock() function itself [96].

```
t0=clock();
r1=r1+r3;
r1=r1+r3;
...
r1=r1+r3;
t1=clock();
While(ADDR_B < LOOP_END)
```

Listing 3.1: CUDA Code Example

The typical technique to measure instruction latency is to use the clock() function. The clock() function returns the value of a per-TPC counter. To measure instruction execution latency, we can just put dependent instructions between two clock() function calls. For example, the CUDA code in Listing 3.1 is translated into PTX code in Listing 3.2.

```
mov.u32     %r6, %clock;
add.f32     %f4, %f4, %f3;
add.f32     %f4, %f4, %f3;
...
add.f32     %f4, %f4, %f3;
mov.u32     %r7, %clock;
```

Listing 3.2: PTX Code Example

```
S2R R3, SR1;
SHL R3, R3, 0x1;
FADD32 R4, R4, R2;
FADD32 R4, R4, R2;
...
FADD32 R4, R4, R2;
S2R R4, SR1;
SHL R4, R4, 0x1;
```

Listing 3.3: Assembly Code Example

The assembly code after compiling PTX code to binary code is in Listing 3.3. S2R instruction move the clock register to a general purpose register. A dependent shift operation after S2R suggests that the clock counter is incremented at half of the shader clock frequency. An extra 28 cycles is introduced because of the dependence between SHL and S2R (24 cycles), and the issue latency of SHL (4 cycles).

For 21 FADD32 instructions between the two clock measurements, the measured cycles are 514. So the execution latency of FADD32 is

$$(514 - 28 - 8)/20 \approx 24.$$

8 cycles are the issue latency of FADD32 in one warp (Please refer to 3.2.2.2 for more details).

Multiple-warp issue latency

Same-warp issue latency of one instruction is the cycles that the issue engine needs to wait to issue another instruction after issuing one warp instruction. It is calculated using instruction throughput. For example, the throughput for integer add instruction is 8 ops/clock. So the issue latency is $32/8 = 4$ cycles. In fact, the issue engine can issue a new instruction every 2 cycles. But if the next chosen warp instruction is also an integer add or other instructions that needs to be issued to SP, it looks like the issue engine has to wait 4 cycles to issue another instruction, since SP can only be issued with one new instruction every 4 cycles.

Same-warp issue latency

Same-warp issue latency is the cycles that the issue engine needs to wait to issue another instruction from the same warp after issuing one warp instruction. This latency can also be measured using the clock() function and is generally longer than multiple-warp issue latency. Thus it is not possible to achieve peak performance with only one active warp on SM even if most nearby instructions in one warp are independent. For example, float MAD instruction's multiple-warp issue latency is 4. If we execute only one warp, then the measured issue latency is 8. For a global memory load instruction GLD.U32, the same-warp issue latency is around 60 cycles while its multiple-warp issue latency is a much smaller value and we use 4 cycles in TEG.

Similar results are obtained for other arithmetic instructions and memory instructions, which suggests that a warp is occupied to issue one warp instruction while the issue engine can continue to issue instructions from other warps and the occupied period is normally longer than the waiting time of the issue engine to issue a new instruction from another warp. So we can redefine the *same-warp issue latency* as the cycles that one warp becomes inactive after issuing one warp instruction. During this period, the issue engine cannot issue instruction from this warp.

Some arithmetic instructions' execution latency and issue latency are listed in Table 3.1. For *Execution latency* and *Multiple-warp issue latency*, the data is borrowed from the work from Wong et al. [96]. Since float MUL operation can be issued into both SP and SFU. The instruction has higher throughput and shorter issue latency. In the table we only present the latency for 16 bit integer MUL and MAD, since 32-bit integer MUL and MAD operations are

Instruction Type	Execution Latency (cycles)	Issue Latency (multiple warps) (cycles)	Issue Latency (same warp) (cycles)
Integer ADD	24	4	8
Integer MUL (16bit)	24	4	8
Integer MAD (16bit)	24	4	8
Float ADD	24	4	8
Float MUL	24	2	8
Float MAD	24	4	8
Double ADD	48	32	32
Double MUL	48	32	32
Double FMA	48	32	32

Table 3.1: Arithmetic Instruction Latency

translated into the native 16-bit integer instructions and a few other instructions. In each SM, there is only one DPU which can processes double precision arithmetic instructions. Thus, the issue latency is much longer for double precision arithmetic instructions.

3.2.2.2 Performance Scaling on One SM

In the previous section, the issue latency is calculated assuming several warps are running concurrently. For example, float MAD instruction's issue latency for multiple warps is 4. But if we run only one warp, then the measured issue latency is 8. And for a global memory load instruction GLD.U32, the issue latency in the same warp is around 60 cycles while the issue latency for multiple warps is a much smaller value and we use 4 cycles in TEG. Similar results are obtained for other arithmetic instructions and memory instructions, which suggests that a warp is occupied to issue one instruction while the scheduler can continue to issue instructions from other warps and the occupied period is normally longer than the waiting time of the scheduler to issue a new instruction from another warp. Thus it is not possible to achieve peak performance with only one active warp on SM even if most nearby instructions in one warp are independent.

After one warp instruction is issued, the scheduler can switch to another warp to execute another instruction without much waiting. However, if the scheduler still issue instructions from the same warp, the longer issue latency is needed. This undocumented behavior may affect performance when there are very few active warps on SM.

3.2.2.3 Masked instruction

All 32 threads within a warp execute the same warp instruction at a time. When threads of a warp diverge due to a data-dependent branch, they may have different execution path. GPU executes each path in a serial manner. Thus, the warp instruction is masked by a condition dependent on thread index. For masked arithmetic instructions, we find that all behavior remains similar as the un masked behavior. That is to say, all the issue latency and execution latency

are the same as those of unmasked arithmetic instructions. For memory operations, since less data needs to be transfered, the latency is shorter and less memory bandwidth is occupied.

3.2.2.4 Memory Access

We consider the memory access separately from the other instructions because of 3 reasons. First, other functional units belong to one SM only, but each 3 SMs within one TPC share the same memory pipeline and all SMs share the same 8 global memory channels. Second, the scheduler needs to wait around 60 cycles after issuing one global memory instruction to issue another instruction in the same warp, but it can issue another instruction very quickly if it switches to another warp (Refer to Section 3.2.2.2). Third, memory access has much more complex behavior. For shared memory access, there might be bank conflicts (Section 3.2.3.3), and then all memory accesses of one half-warp are serialized. For global memory access, there might be coalesced and uncoalesced accesses (Section 3.2.3.4).

The typical shared memory latency is about 38 cycles and the global memory latency without TLB miss is about 436 to 443 cycles [96].

Let C_{mem} represent the maximum number of concurrent memory transactions per TPC and it is calculated as Equation 3.1 and 3.2.

$$\frac{N_{TPC} * N_{Warp} * ele\_size * C_{mem}}{mem\_latency * \frac{1}{Clk}} = B_{peak} \tag{3.1}$$

$$C_{mem} = \frac{B_{peak} * mem\_latency}{N_{TPC} * N_{Warp} * ele\_size * Clk} \tag{3.2}$$

N_{TPC}, N_{Warp}, ele_size, $mem\_latency$, Clk, and B_{peak} represent the number of TPCs, the number of threads per warp, the accessed data type size, the global memory latency, processor clock frequency, and the peak global memory bandwidth respectively. For double precision memory transactions, $C_{mem} \approx 18$. Thus the number of unfinished double precision memory transactions through the memory pipeline of a TPC cannot exceed 18.

3.2.3 Performance Effects

3.2.3.1 Branch Divergence

Masked instructions (Section 3.2.2.3) are warp instructions with a warp size mask. Each bit of the mask indicates whether the corresponding thread is active to execute the instruction. Threads of the same warp may have different execution path. Since SM has to finish each path in serial and then rejoin, extra execution time is introduced.

3.2.3.2 Instruction Dependence and Memory Access Latency

One of the motivations or advantages of GPU processor is that it can hide latency due to instruction dependence or memory access by forking large number of threads. However, when there are very few active warps, it is possible that at some point, all warps are occupied in issuing instructions. The scheduler is available but none of the active warps can be released. Thus

the latency cannot be perfectly hidden and may become an important factor to performance degradation.

3.2.3.3 Bank Conflicts in Shared Memory

The shared memory is divided in 16 memory modules, or banks, with the bank width of 4 bytes. The bank is interleaved so that successive 4 bytes words in shared memory space are in successive banks. Threads in a half-warp should access different banks to achieve maximum shared memory bandwidth. Otherwise the access is serialized [79], except all threads in a half-warp read the same shared memory address.

For example, the float ADD instruction
FADD32 R2, g[A1+0xb], R2;
has a operand *g[A1+0xb]* located in shared memory. The execution latency is around 74 cycles without bank conflict. If all threads within a half-warp access the same bank, the execution latency becomes about 266 cycles.

3.2.3.4 Uncoalesced Memory Access in Global Memory

The global memory of GPU has very high access latency comparing to shared memory latency. For global memory accesses of a half-warp, if certain conditions are satisfied, the memory transactions an be coalesced into one or two transactions. The required conditions depend on GPU hardware and CUDA compute capabilities. The general guideline is that threads of one half-warp should access adjacent memory elements. If the coalesced conditions cannot be met, more memory transactions are needed, introducing much performance loss. For example, if every thread loads 4 bytes from global memory, in the worst case, to serve each thread in the half-warp, 16 separate 32-byte transactions are issued. Thus 87.5% of the global memory bandwidth is wasted.

3.2.3.5 Chanel Skew in Global Memory

The global memory of GT200 GPU is divided into 8 partitions. The global memory thus can be accessed through 8 channels. The channel width is 256Btyes (32*8B) [79]. Similar as accessing to shared memory, concurrent accesses to global memory should be distributed evenly among all the partitions to achieve high global memory bandwidth. Load imbalance on the memory channels may significantly impair performance. If the application's memory access pattern has significant imbalance over different channels, much performance degradation will be introduced.

3.3 Workflow of TEG

Based on our timing model of GPU, we have developed the GPU timing estimation tool TEG. The workflow of TEG is illustrated in Figure 3.2. The CUDA source code is first compiled into binary code with NVIDIA compiler collection. The binary code includes the native kernel code that runs on GPU device. Second, the binary code is disassembled using tool *cuobjdump*

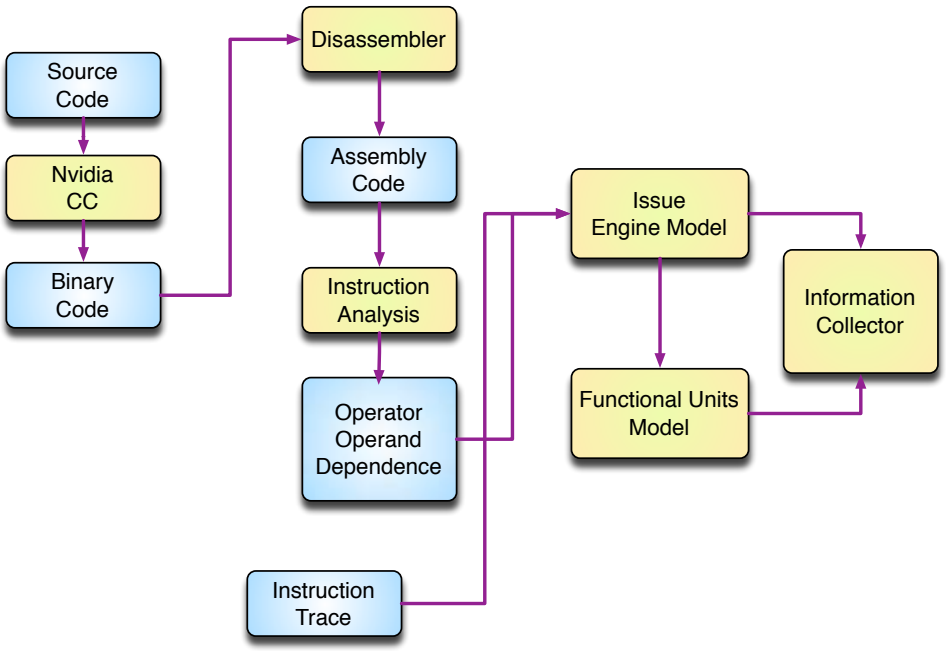

Figure 3.2: Workflow of TEG

provided by NVIDIA [2]. Third, TEG analyzes the generated assembly code and obtains information such as instruction type, and operands' type, etc.

We need the actual instruction traces in many cases. The instruction trace can be obtained with detailed GPU simulators, such as Barra[24] or GPGPU-Sim[11]. In our study, the instruction trace is provided by Barra simulator.

So after the third step, the assembly code information and instruction trace are served to issue engine model (see Figure 3.2). The issue engine model issues all the warp instructions to corresponding functional units model according to the instruction trace and our GPU timing model. At this stage, all runtime timing information can be collected by our information collector.

We can vary the configuration of TEG, such as the active warp number on SM to observe how performance scales from one warp to multiple concurrent warps. We can also compare the performance with or without one bottleneck by choosing whether or not to apply the bottleneck's effects in TEG. Thus we can quantify how much performance gain we may get by eliminating the bottleneck and programmers can decide whether it is worth the optimization efforts.

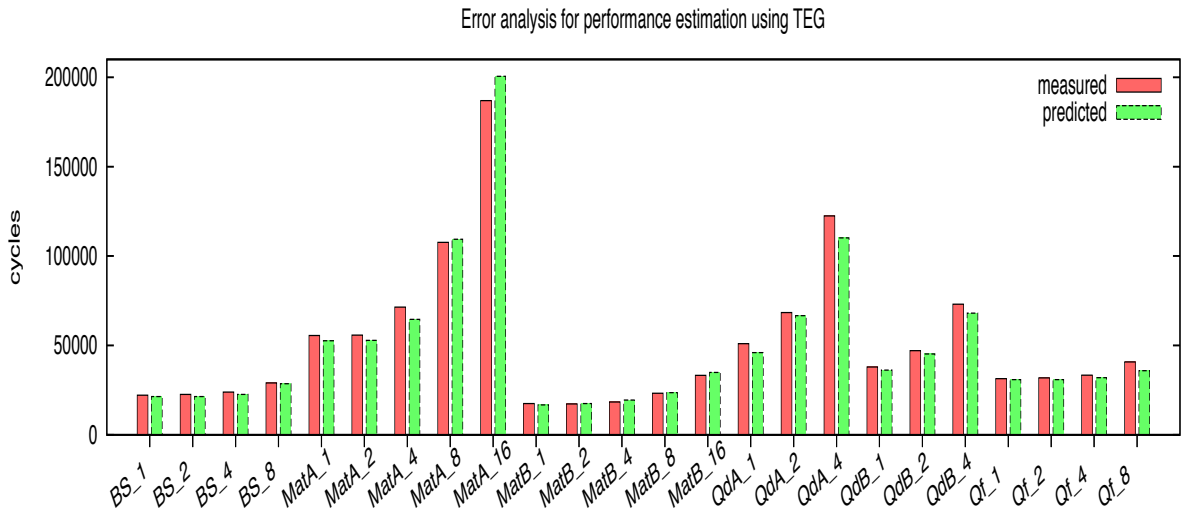

Figure 3.3: Erro analysis of TEG

3.4 Evaluation

We evaluate TEG with several benchmarks with different configurations and compare the measured and estimated kernel execution times. The result is shown in Figure 3.3. The name is defined as $KernelName\_WarpNumber$. BS, $MatA$, $MatB$, QdA, QdB, Qf stand for Blackscholes, naive matrix multiplication, matrix multiplication without shared memory bank conflict, double precision Lattice QCD kernel with uncoalesced memory access, double precision LatticeQCD kernel with coalesced memory access, single precision Lattice QCD kernel respectively. The $WarpNumber$ is the number of concurrent warps assigned to each SM. Here we assign the same amount of workload to each warp. The result shows that TEG has good approximation and it also can catch the performance scaling behavior. The average of absolute relative error is 5.09% and the maximum absolute relative error is 11.94%.

To study how much performance loss due to one performance degradation factor, with TEG, we just need to change the tool configuration. For example, one application has shared memory conflicts and we would like to know how much performance is impaired by this factor. Within TEG, we just set that all shared memory accesses are conflict-free. Thus we can estimate the performance without shared memory bank conflict and decide whether it is worth the optimization efforts. We do not have to implement each version of codes to compare.

3.4.1 Dense Matrix Multiplication

We choose one example of dense matrix multiplication in CUDA SDK and to demonstrate the function of TEG, we change $C = AB$ into $C = AB^T$.

WarpNum	1	2	4	8	16
Measured (cycles)	55605	55803	71465	107668	186958
Predicted (cycles)	52590	52878	64578	109364	200538
Error	-5.73%	-5.53%	-10.66%	1.55%	6.77%

Table 3.2: $C = AB^T$ with Bank Conflict

$$C(i,j) = \sum_k A(i,k) * B(j,k)$$

In the implementation, the three matrices A, B and C, are partitioned into 16x16 sub-matrix. The computation for a C sub-matrix is assigned to a CUDA block. A block is composed of 256 (16x16) threads and each thread computes one element in the C sub-matrix. In the CUDA kernel, at each step, a block of threads load the A and B sub-matrices first into shared memory. After a barrier synchronization of the block, each thread loads $A(i,k)$ and $B(j,k)$ from shared memory, and accumulates the multiplication result to $C(i,j)$.

However, since a half-warp of threads, load $B(j,k), B(j+1,k), \ldots, B(j+15,k)$, for a shared memory allocation like B[16][16], all the 16 elements will reside in the same bank and there would be bank conflicts in the shared memory.

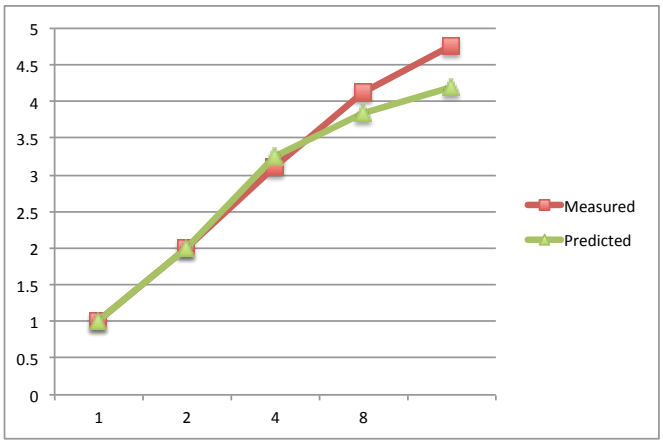

Figure 3.4: $C = AB^T$ with Bank Conflict

In the following experiment, we assign each warp with the same amount of workload and run 1 to 16 warps concurrently on one SM. We use clock() function to measure the execution time on device of one block, since the barrier synchronization is only applicable within one block. And for multiple blocks' total execution time, we use the measured host time to calculate the device execution time. For example, when there are 30 blocks, each SM can be assigned one block and when there are 60 blocks, each SM has two blocks to execute. Then we compare the host time for the two configurations and calculate the cycles for 2 blocks (16 warps) to finish on the SM.

WarpNum	1	2	4	8	16
Measured (cycles)	17511	17291	18330	23228	33227
Predicted (cycles)	16746	17528	19510	23630	34896
Error	-4.57%	1.35%	6.05%	1.70%	4.78%

Table 3.3: $C = AB^T$ Modified

The measured and predicted execution time of 1 to 16 concurrent warps on one SM is illustrated in Table 3.2. Then we normalize the execution time with the workload and show the speed up from 1 to 16 active warps on each SM in Figure 3.4.

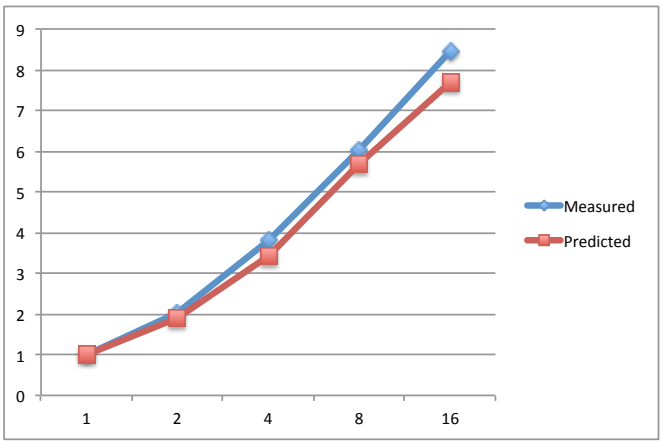

Figure 3.5: $C = AB^T$ Modified

For GPU performance optimization, programmers often come to the question that how much performance loss due to one performance degradation factor. With TEG, it is fairly easy to answer the question. We just need to change the configuration. In this case, in the tool, we just assume all shared memory accesses are conflict-free. Thus we can estimate the performance without shared memory bank conflict, which is illustrated in Figure 3.5 and Table 3.6.

We then modified the CUDA code to eliminate bank conflicts and compare the result with TEG's output. The comparison shows very good approximation.

3.4.2 Lattice QCD

We select one kernel in Hopping_Matrix routine [43] as our example. The input of the Hopping_Matrix kernel include the spinor field, the gauge field, the output is the result spinor field. The spinor field resides on the 4D space-time site and is represented by a 3x4 complex matrix data structure. The gauge field on the link connecting neighbor sites is implemented as a 3x3 complex matrix. The half spinor filed is represented by a 3x2 complex matrix, which is the temporary data generated on each of 8 space-time directions for one full spinor.

WarpNum	1	2	4
Measured (cycles)	51053	68383	122430
Predicted (cycles)	46034	66674	110162
Error	-10.90%	-2.56%	-11.14%

Table 3.4: Hopping_Matrix kernel with Uncoalesced Accesses

WarpNum	1	2	4
Measured (cycles)	37926	47038	73100
Predicted (cycles)	36202	45204	68104
Error	-4.76%	-4.06%	-7.34%

Table 3.5: Hopping_Matrix kernel with Coalesced Accesses

The functionality of the kernel is not important to our discussion. Instead, the memory layout is of interest. In the first version of our implementation, all the data is organized in array of structures. This is typical data layout for conventional processors to obtain good cache hit rate. However, GPU has much more concurrent threads. Normally different threads are assigned with different data structures. So the accesses of the threads in a warp have a long stride of the size of the data structure. Thus, accesses to global memory cannot be coalesced. The predicted and measured execution results are illustrated in Table 3.4 and Figure 3.6. Since each thread occupies much register resource, the active warp number is limited.

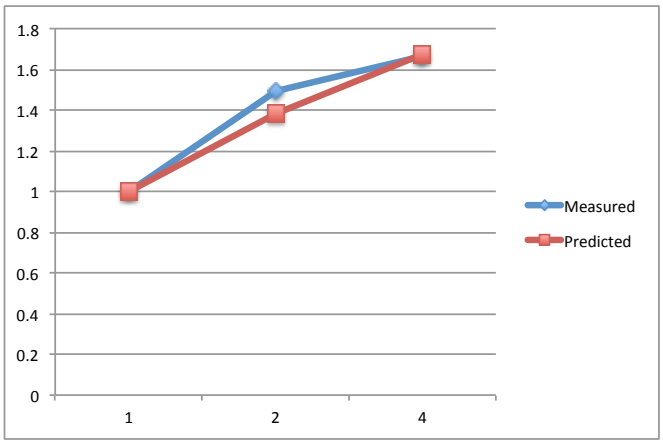

Figure 3.6: Hopping_Matrix kernel with Uncoalesced Accesses

If we reorganize the data layout into structure of arrays, the memory accesses of threads in a warp would be adjacent. Thus they can be coalesced. The result is shown in Table 3.5 and Figure 3.7. This case also shows that TEG can easily demonstrate the performance loss due to performance bottlenecks, such as uncoalesced memory accesses.

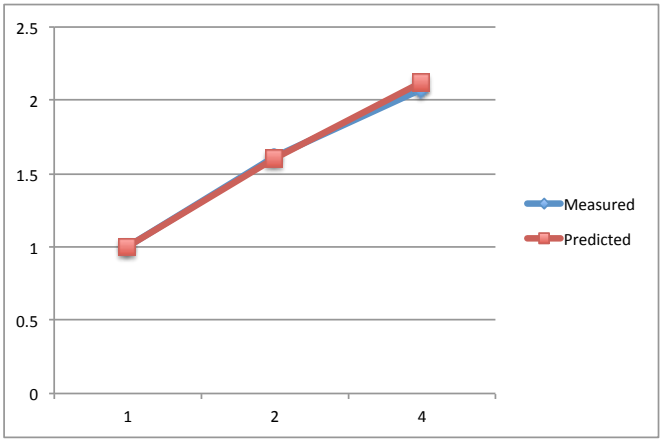

Figure 3.7: Hopping_Matrix kernel with Coalesced Accesses

WarpNum	1	2	4	8	16
Measured Cycles	17511	17291	18330	23228	33227
Predicted Cycles	16784	16868	18474	20852	34688
Error	-4.33%	-2.51%	0.78%	-11.39%	4.21%

Table 3.6: $C = AB^T$ Modified

3.5 Performance Scaling Analysis

In Section 3.4, we have shown that TEG has good approximation for GPU execution time estimation. And to study how much performance loss due to one performance degradation factor. Furthermore, we believe it is useful to understand the detail execution state of GPU's different function units, especially how performance scales when active warps increase on one SM.

We still use one example of dense matrix multiplication in CUDA SDK to demonstrate our study. Here we only use the version without shared memory bank conflict. The measured running cycles and estimated results are presented in Table 3.6.

From the result (Table 3.6), the performance scales almost perfectly from one warp to two concurrent warps since workload doubles and execution time almost remains the same, and scales very well until 8 warps. From 8 warps to 16 warps, the performance still gains from more concurrent threads, but not as well as before. We want to understand what factors devote to the execution time and how the performance scales like this.

Figure 3.8 presents how the PCs (program counter) change through the execution and the warp numbers are 1 and 8. The execution could be easily identified as 3 stages. At the first stage, each thread computes the addresses of its assigned data elements, according the thread and block index. The second stage is the main loop, 10 iterations in this case. A block of

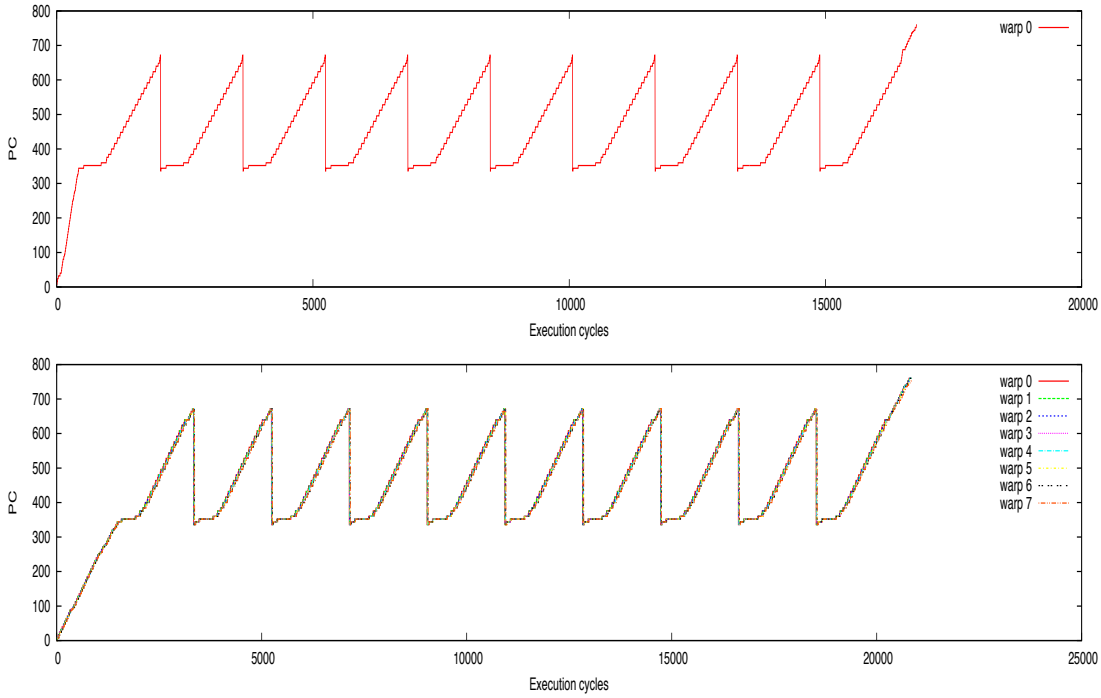

Figure 3.8: PC Trace

threads load the A and B sub-matrices into shared memory and accumulate the multiplication result. The 'flat' part is loading data from global memory to shared memory. The last stage is the address computation for the target data element $C(i, j)$ and the store back.

We have some interesting observation here. First, the time of index and address calculation is non eligible comparing to the matrix multiplication. Second, the fraction of address calculation increases when concurrent warps increase. Third, the curves of different warps are very close because in each iteration there is a barrier synchronization and also because we use a very simple score board policy to choose the next warp with ready instruction, $next\_wc = (current\_wc + 1)\%WARP\_NUM$. Fourth, the fraction of global memory accesses decreases when warps increase.

To understand these performance behavior, we further improve TEG to collect workload information of different function units. Workload here refers to the warp instructions executing concurrently in the function units. For example, SP can be issued in one warp instruction every 4 cycles. If all instructions issued to SP has an execution latency of 24 cycles, the maximum number of instructions executing in parallel is 6. Instructions with operands in shared memory have longer execution latency than instructions with all register operands. Thus the instructions executing in parallel could be more.

Figure 3.9 shows the workload for SP when there is only one warp. Apparently, the workload of SP is far from saturation. We suppose that for issuing each new warp instruction, SP is active for 4 cycles. The active percentage for SP is the fraction when SP is active during the ex-

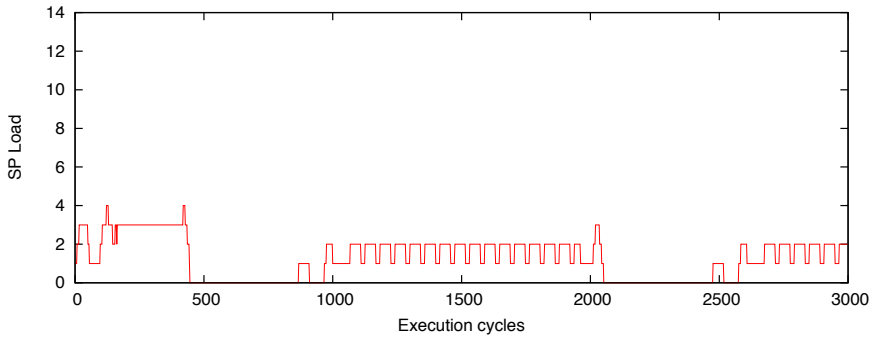

Figure 3.9: SP Load(1 warp)

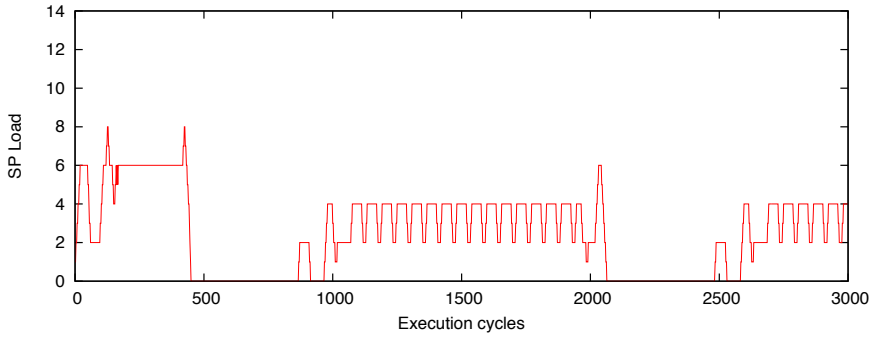

Figure 3.10: SP Load (2 warps)

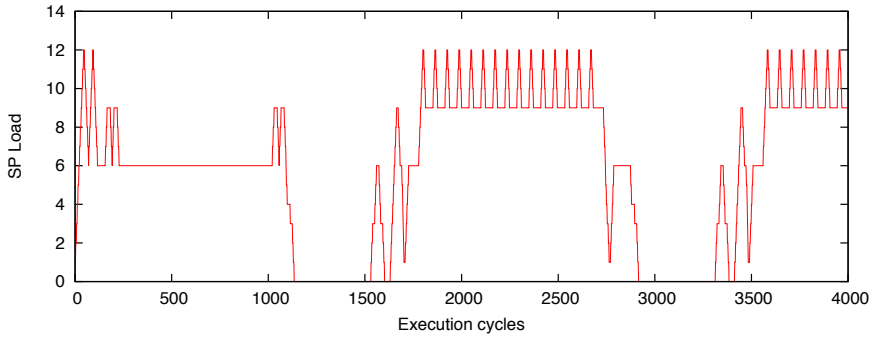

Figure 3.11: SP Load (6 warps)

ecution. And then active percentage of SP is 11% in this case. When there are two concurrent warps, the workload is illustrated in figure 3.10 and the active percentage is 22%. Similarly, for concurrent warp number of 4, 8, and 16, the active percentage of SP is 40%, 71.7%, and 86.2% respectively.

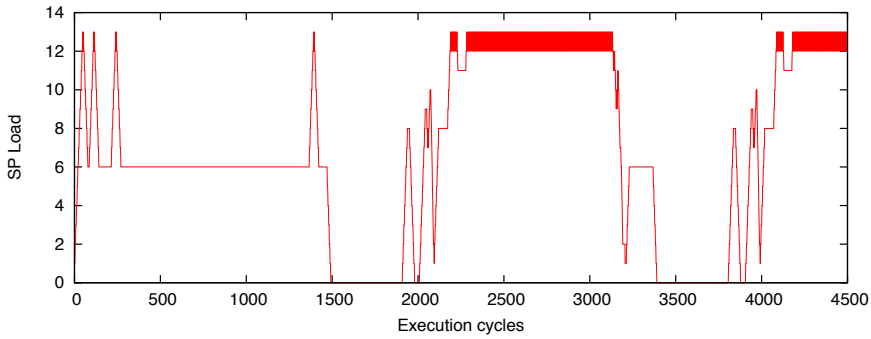

Figure 3.12: SP Load (8 warps)

As in Figure 3.10, for index calculation, the workload is already around 6. Since index calculation is mainly integer operations with register operands, it almost reaches the best performance when warp number is 2. Increasing warps cannot introduce much gain for index calculation.

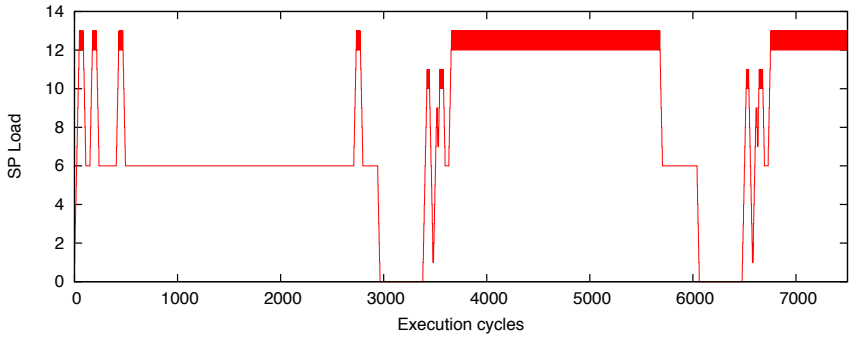

Figure 3.13: SP Load (16 warps)

Figure 3.11, 3.12 and Figure 3.13 present the workload for SP when there are 6, 8 and 16 concurrent warps. Comparing the results, we can find that for matrix multiplication, the peak workload is around 13. When the warp number is 6, the workload is about 9 12, already close to the saturation of SP.

The analysis of the workload on LD/ST unit is simpler. Figure 3.14 and Figure 3.15 illustrate the workload on LD/ST unit when there are 1 and 16 warps. As we can see, the workload with 16 warps is almost 16 times of the workload with 1 warp. In this application, the data is loaded first into shared memory, so the pressure on memory bandwidth is not heavy.

With the previous analysis, we can explain the performance scaling result. When the active warp number is 1, all function units are not saturated and the SP starts to be saturated at the index calculation part with 2 active warps. Thus, the scaling from 1 to 2 warps is almost perfect. From 2 to 4 active warps the performance scales very well but not as perfect as from 1 to 2

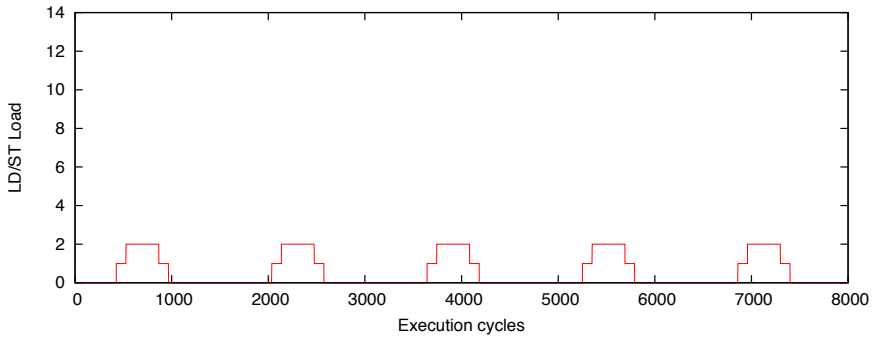

Figure 3.14: LD/ST Unit Load (1 Warp)

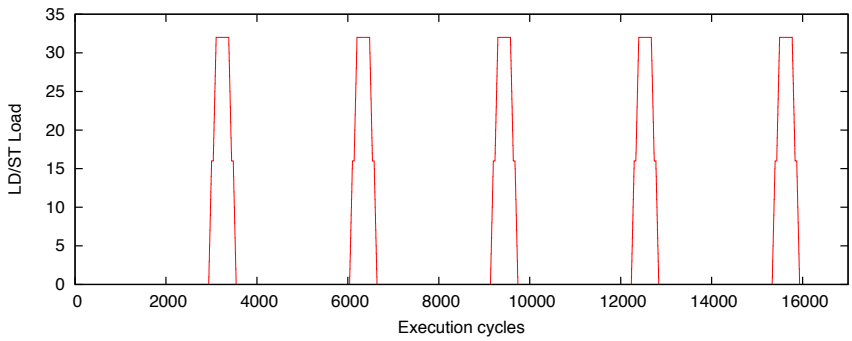

Figure 3.15: LD/ST Unit Load (16 Warps)

warps. The SP starts to be saturated at the matrix multiplication accumulation segment when there are 6 active warps. Also because this code segment devotes to a large portion of total execution time, the scaling from 4 to 8 warps is even worse. Increasing warps from 8 to 16 benefits very little in the matrix multiplication accumulation code segment. However since the memory is not full, there is still some performance gain.

3.6 Summary

In this Chapter, we use our GPU timing estimation tool TEG to analyze detailed performance scaling behavior on GPU. With the timing model and the assembly code as input, in coarse grain, TEG can estimate applications' cycle-approximate performance on GPU and has an acceptable error rate. Especially, TEG has good approximation for applications with very few active warps on SM. In fine grain, we can use TEG to break down the GPU execution time and gain more insight into GPU's performance behavior. Current profiling tools can only provide statistics information for one kernel while with TEG, it is easy to get how much performance one bottleneck can impair and foresee the benefit of removing this bottleneck.

The main limitation is that TEG cannot handle the situation when there is a high memory

traffic and a lot of memory contention occurs. We lack the knowledge of detailed on-die memory controller organization and the analysis is far too complicated for our analysis method. Adding cache impact, e.g. for Fermi architecture, could be considered as an extension of TEG.

Chapter 4

Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs

This Chapter presents a study that is going to be presented at CGO 2013 [54].

4.1 Introduction

There are many studies about optimizing specific kernels on GPU processors [99, 78, 38, 47, 89, 60, 102, 58, 66, 28, 97, 14, 79]. However, since the architecture is changing with each generation, we may need to repeat the optimization work again very soon. Unfortunately, no practical performance upper bound evaluation is available to the developers. In practice, developers apply several optimization techniques based on the analysis to the algorithm or serial code, and their expert experience. Then developers may modify the optimizations with feedback provided by tools like NVIDIA Visual Profiler [72]. However, they can not be sure how far the obtained performance is from the best achievable performance. In this chapter, we present an approach to project performance upper bound using algorithm analysis and assembly code level benchmarking.

As described in Chapter 1, there exist many works about how to project/predict CUDA applications' performance using analytical or simulation methods to understand GPU performance results [61, 40, 85, 101, 26]. However existing GPU performance models all rely on certain level of an application's implementation (C++ code, PTX code, assembly code...) and do not answer the question of how good the current optimized version is and whether further optimization effort is worthwhile or not. Different from existing GPU performance models, our approach does not project the possible performance from certain implementations, but the performance upper bound that an application cannot exceed.

Researchers are also interested in the outcome of different optimization combinations on GPUs. The roofline model [94] is well known for estimating the optimization effects. Many automatic or manual optimization frameworks have the similar ideas as the roofline model. However, the chosen optimizations normally rely on the initial code version and different opti-

mizations are likely to have complex impacts on each other. Our approach tackles the problem from the opposite angle as the roofline method. We first assume an optimistic situation on GPUs (no shared memory bank conflict, global memory accesses are all coalesced, all the auxiliary operations like address calculations are neglected, etc.). Then we try to predict a performance upper bound when mapping an application on the GPU based on the constraints introduced by the architecture, the instruction set and the application itself, or the constraints that we are not able to eliminate using optimization techniques. With a tight performance upper bound of an application, we have an evaluation on how much optimization space is left and can decide the optimization effort. Also, with the analysis, we can understand which parameters are critical to the performance and have more insights into the performance result. Hence, with these knowledge, it would be easier for the community to move to the new architecture.

As an example, we analyze the potential peak performance of SGEMM (Single-precision General Matrix Multiply) on Fermi (GF110) and Kepler (GK104) GPUs. GEMM [1] operation is essential for Level 3 BLAS (Basic Linear Algebra Subprograms) [1] routines and generally represents the practical best performance of a computer system. If we compare the performance of SGEMM from CUBLAS with the theoretical peak performance, on Fermi, it achieves around 70% and on Kepler, only around 42% of the theoretical peak performance. The initial intention of this research is to understand this huge performance gap with the theoretical peak performance.

There are already some articles about optimizing GEMM kernels on Fermi GPU [67] [88], and an auto-tuning framework has also been presented [52]. In this research, the focus is to answer the question of how much optimization space is left for SGEMM and why. We also show that the analysis can help optimization efforts since it uncovers critical parameters. Only single precision SGEMM is evaluated, since we could only access the GTX580 Fermi and the GTX680 Kepler Geforce cards, which have much poorer double precision performance than Tesla products. It is not really worth the effort to study the DGEMM performance on Geforce GPU.

Depending on whether to apply transpose operation on input matrix A or B, there are 4 variations for GEMM kernel. Guided by this analysis and using the native assembly language, our four SGEMM kernel variations achieved about 11% (NN), 4.5% (TN), 3% (NT) and 9% (TT) better performance than CUBLAS in CUDA 4.1 SDK for large matrices on GTX580 Fermi Card (N stands for "normal", T stands for "transpose"). The achieved performance is around 90% of the estimated upper-bound performance of GTX580. On GTX680 Kepler GPU, the best performance we achieved (NT) is around 1375GFLOPS, around 77.3% of the estimated performance upper bound.

In November 2012, NVIDIA has announced the new Tesla K20X Kepler GPU (GK110) and the documented SGEMM efficiency is around 73% of the theoretical peak performance [76]. The K20X Kepler GPU (GK110) architecture is different from the GTX680 (GK104) and uses a different instruction set (each thread can utilize maximum 255 registers on the new architecture while the limit is 63 on GTX680 GPU). With a Tesla GPU card, it should not be difficult to extend the analysis to SGEMM and DGEMM on the Tesla GPU using our approach.

[1] GEMM performs the matrix-matrix operation $C := alpha * op(A) * op(B) + beta * C$. alpha and beta are scalars, and A, B and C are matrices. op(X) is $op(X) = X$ or $op(X) = X^T$.

This chapter is organized as follows: Section 4.2 introduces our assembly level benchmarking approach. Section 4.3 presents our analysis for performance upper bound of SGEMM on Fermi and Kepler GPUs. In Section 4.4 assembly code level optimization methods and performance result of SGEMM are presented. Section 4.5 is the summary of this chapter.

4.2 CUDA Programming with Native Assembly Code

A typical CUDA [2] program normally creates thousands of threads to hide memory access latency or math pipeline latency. The *warp* is the basic execution and scheduling unit of a SM, and is composed of 32 threads. We define a **warp instruction** as the same instruction shared by all threads in the same warp, and a **thread instruction** as the instruction executed by one thread. So a *warp instruction* launches 32 operations or consists of 32 *thread instructions*. On the SM, only a limited set of threads can run concurrently (active threads). On one hand, the increased SPs require more active threads to hide latency. On the other hand, the register and the shared memory resource limits the number of active threads. For the same application, the active threads that one SP supports actually decreases because of the reduced memory resource per SP from Fermi GPU to Kepler GPU. More instruction level parallelism within one thread needs to be explored (Section 4.3.3).

For Fermi (and Kepler GK104) instruction set, there is a hard limit of maximum 63 registers per thread (for GT200 generation the limit is 127 registers per thread) since in the instruction encoding, only 6 bits are left for one register. To reduce the effects of register spilling, Fermi GPU introduces L1 cache. Local writes are written back to L1. Global stores bypass L1 cache since multiple L1 caches are not coherent for global data. L2 cache is also introduced in Fermi and reduces the penalty of some irregular global memory accesses.

For performing this study, we had to develop some software components and reverse engineer many characteristics of the hardware. We used GPU assembly code directly with an assembly tool Asfermi[8]. Asfermi was first developed to work on Fermi GPU. We patched Asfermi to support Kepler GPU (GK104) and managed to use native assembly language directly in the CUDA runtime source code. On Kepler GPU, new scheduling information is embedded in the CUDA binary file. We studied the scheduling information and found some patterns (Section 4.2.2). However, NVIDIA does not disclose the encoding of the control information and our decryption is still not enough. According to our benchmarks, we found that the instruction throughput is related to register indices on Kepler GPU (Section 4.2.3). We studied the register bank conflict problem in some math instructions and proposed a solution for SGEMM.

4.2.1 Using Native Assembly Code in CUDA Runtime API Source Code

Programming in assembly code on NVIDIA GPUs is not publicly supported by the company. However, our analysis is requiring such programming. With an assembly tool for Fermi GPU called Asfermi [8] and a little hacking into the CUDA programming compiling stages, we manage to use hand-tuned GPU assembly code in CUDA projects using CUDA runtime APIs .

There are several advantages of using assembly code or native machine code directly instead of using high level languages like C++. First, we can carefully control the register alloca-

tion since the register resource per thread is very limited and sometimes the compiler may spill many registers for programs utilizing much register resource per thread like SGEMM. Second, the instruction order can be carefully designed to better prefetch data from global memory and mix different instruction types to get better throughput. Third, SIMD-like instructions (LDS.64 or LDS.128) could be used intentionally to reduce the instruction number. Also, we can control the exact behavior of the machine code. For example, the compiler might choose to use wider load instructions (LDS.64 or LDS.128) based on the data alignment in shared memory. However, using wide load instructions does not always benefit the performance (Section 4.3.1).

A CUDA program is composed of host code running on the host CPU, and device code running on the GPU processor. The device code written in C/C++ is first compiled into PTX code, and then compiled into native GPU binary code. The binary file (cubin file) is an ELF format file which contains the native machine code. Asfermi can translate the assembly code into binary and generate CUDA binary file (.cubin file). The assembly code which Asfermi uses is similar to the output of NVIDIA's disassembler $cuobjdump$. Also, according to public materials, CUDA binary file cannot be directly used in a project built with CUDA runtime API. The CUDA binary (.cubin) file can only be used with the CUDA driver API. However, in our SGEMM implementation, we found that loading the .cubin file using the driver API may degrade the performance. Besides, many projects are programmed with CUDA runtime API. This restricts the usage of the code written in assembly language.

We manage to integrate our CUDA binary file into a CUDA runtime project. In a CUDA runtime API project, we keep all the intermediate files generated by nvcc (NVIDIA Compiler Collection). Then we replace the CUDA binary file with the one that is generated by Asfermi and rebuild the project. The PTX file should be removed in the compiling process. Otherwise, the GPU may utilize the PTX information embedded in the fat binary file other than the CUDA binary file that Asfermi generates.

1. Add "-v -keep" to nvcc options, so that all the intermediate files are saved and we can have all the compiling steps.

2. Write one .cu files, eg. $kernel.cu$, which has one dummy kernel function with the same device function name as your CUDA binary code.

3. Add $kernel.cu$ into the project.

4. Build the project and collect all the compiling command line.

5. Replace the compiled $kernel.cubin$ with the one generated by Asfermi. Regenerate the $kernel.fatbin.c$ file.

6. Regenerate the $kernel.cu.cpp$ and then $kernel.cu.o$ according to the original command line information.

7. Rebuild the whole project.

4.2.2 Kepler GPU Binary File Format

Asfermi was first developed to work for Fermi GPU. We patched Asfermi to support CUDA sm_30 (GK104 Kepler GPU). However, although the CUDA program can still run correctly on Kepler GPU, the performance is very poor. The reason is that new control information is embedded into the CUDA binary file to help processor scheduling. According to the GTX680 white paper [75], the scheduling functions on Kepler GPU have been redesigned to save energy and space. Because the math pipeline latencies are deterministic, the compiler does more work during compiling stages and put the scheduling information along with the actual instructions in the CUDA binary file.

According to our study of the output of NVIDIA disassembler $cuobjdump$, this information (we call it control notation) is placed before each group of 7 instructions. It is similar to the explicit-dependence lookahead used in Tera computer system [5]. Because Kepler GPU uses 64bit wide instruction, the control notation appears at addresses of 0x0, 0x40, etc. The control notation has the format of 0xXXXXXXX7 0x2XXXXXXX. 0x7 and 0x2 are identifiers and the rest of the notation is separated into 7 fields and associated with each following instruction. Unfortunately, NVIDIA does not disclose the encoding of the control notation. So far, we do not know how to generate the control notation exactly as the nvcc compiler. In our implementation of SGEMM on Kepler GPU, as a compromise, we use the same control notation for same kind of instructions and try to find the best combination of those notations for major instruction types. However, our decryption of the notations is still not enough.

4.2.3 Math Instruction Throughput on Kepler GPU

Understanding and modeling the behavior of math instructions on Kepler GPU is a major difficulty. We use two approaches to test the throughput of math instructions. First, a kernel is written in C++ code and compiled into binary with control notations embedded by nvcc. Second, a kernel is written in assembly code directly and the controlling notations are embedded with our parsing tool. Each thread executes the same 8192 math instructions. Each block has 1024 threads without synchronization and 40960 blocks are spawned to keep the GPU busy.

Instruction FFMA performs single precision fused multiply-add operation ($FFMA\ RA,\ RB,\ RC,\ RD$ performs the operation $RA := RB * RC + RD$). With the first approach, the instruction throughput of $FFMA\ R9,\ R8,\ R9,\ R5$ is measured as 129.2 operations per shader cycle [2]. With the second approach and the control notation of 0x25, the throughput is 132.0 operations per shader cycle (The actual shader clock cannot be obtained during execution. All throughput data is calculated by boost clock of 1058MHz[75]).

Some math instructions' throughput is illustrated in Table 4.1 measured with the second approach. In these cases, the scheduling function units on one SM can only issue about maximum 132 thread instructions per shader cycle, which is much lower than the SP's processing throughput (192 thread instructions per shader cycle). If some of the three source registers are the same (like FFMA RA, RB, RB, RA), with some carefully designed code structures, the FFMA throughput can approach around 178 thread instructions per shader cycle. However,

[2]The actual implementation is not 8192 *FFMA R9, R8, R9, R5* instructions per thread but 4 independent FFMA instructions like *FFMA R9, R8, R9, R5* unrolled by 2048 times.

FADD R0, R1, R0	128.7	FADD R0, R1, R2	132.0
		FADD R0, R1, R3	66.2
FMUL R0, R1, R0	129.0	FMUL R0, R1, R2	132.0
		FMUL R0, R1, R3	66.2
FFMA R0, R1, R4, R0	129.0	FFMA R0, R1, R4, R5	132.0
		FFMA R0, R1, R3, R5	66.2
		FFMA R0, R1, R3, R9	44.2
IADD R0, R1, R0	128.7	IADD R0, R1, R2	132.4
		IADD R0, R1, R3	66.2
IMUL R0, R1, R0	33.2	IMUL R0, R1, R2	33.2
		IMUL R0, R1, R3	33.2
IMAD R0, R1, R4, R0	33.2	IMAD R0, R1, R4, R5	33.1
		IMAD R0, R1, R3, R5	33.2
		IMAD R0, R1, R3, R9	26.5

Table 4.1: Examples of Math Instruction Throughput on Kepler GPU with Various Operand Register Indices

considering 'useful' FFMA's throughput, that is (FFMA RA, RB, RC, RA), the maximum single precision performance for many applications like SGEMM on GTX680 GPU (GK104) cannot exceed around 68.75% (132/192) of the claimed performance (3090GFlops) by NVIDIA.

Our benchmark result also shows that the instruction throughput is related to register indices. According to some other experiments, we speculate that the registers reside on four banks. Take the instruction $FFMA\ RA,\ RB,\ RC,\ RD$ for instance, if there are two different source registers on the same bank, the throughput drops by 50%, and if all three source registers RB, RC, RD are different registers on the same bank, the throughput is around 33.3% of the best case. We name the four banks as even 0 ($R_{index}\%8 < 4$ && $R_{index}\%2 == 0$), even 1 ($R_{index}\%8 \geq 4$ && $R_{index}\%2 == 0$), odd 0 ($R_{index}\%8 < 4$ && $R_{index}\%2 == 1$), and odd 1($R_{index}\%8 \geq 4$ && $R_{index}\%2 == 1$). Since we implement SGEMM with assembly code directly, the register indices have to be carefully chosen to make sure there is no bank conflict. The detailed optimization is illustrated in section 4.4.4.

4.3 Analysis of Potential Peak Performance of SGEMM

The general analysis approach can be similar for all applications while the detailed analysis process may differ from application to application. Our method is applicable for applications which use a few major instruction types and a simple execution path. Many high-performance computing kernels have this characteristic, especially linear algebra routines. Our analysis requires characteristics of the architecture such as register file size, maximum number of registers per thread, shared memory size, instruction throughput for different instruction mix, etc. Those characteristics need to be collected on the real hardware and are independent of the effective application.

	Parameters	Definition	Value
1	Blk	Active block number per SM	Implementation
2	T_B	Active thread number per block	Implementation
3	T_{SM}	Active thread number per SM	Implementation
4	R_T	Registers allocated for each thread	Implementation
5	R_{SM}	Register resource per SM	Hardware
6	Sh_{SM}	Shared memory per SM	Hardware
7	B_{Sh}	Blocking factor at shared memory level	Implementation
8	B_R	Blocking factor at register level	Implementation
9	#GlobalMem_bandwidth	Theoretical Global memory bandwidth	Hardware
10	R_{Max}	Maximum register number per thread	Hardware
11	R_{index}	Registers allocated for indices and addresses	Implementation
12	L	Stride of loading A, B sub matrices into shared memory	Implementation
13	$P_{theoretical}$	Theoretical peak performance	Hardware
14	F_T	Throughput effect of mixing FFMA and LDS.X instructions	Hardware
15	F_I	Instruction factor by using LDS.X instructions	Implementation
16	#SP_TP	SP Thread Instruction processing throughput	Hardware
17	#LDS_TP	LD/ST Unit Shared Memory Thread Instruction throughput	Hardware
18	#Issue_TP	LD/Dispatch Unit Thread Instruction issue throughput	Hardware

Table 4.2: Architecture and Algorithm Parameters

First, we should analyze the instruction types and percentage of a routine. Second, we should find the critical parameters which affect the different instructions' mixing percentage. Third, we analyze how the instruction throughput changes when we vary these critical parameters. Fourth, we can use the instruction throughput with critical parameters' optimal combination to estimate the performance upper bound. With this approach, not only we can have the performance upper bound estimation, know how much performance gap is left and decide the optimization effort, but we can also understand what parameters are essential to the performance and how to distribute our optimization effort.

For SGEMM performance upper bound analysis , the parameters we define are listed in Table 4.2.

For SGEMM, all well-implemented SGEMM kernels actually utilize shared memory on the GPU to reduce the global memory pressure as illustrated in Figure 4.1. First, data is loaded from global memory to shared memory and then threads within one block can share the loaded data in the shared memory. One possible implementation is illustrated in Listing 4.1.

For Fermi (GF110) and Kepler (GK104) GPUs, arithmetic instructions like FFMA cannot take operands from the shared memory. Since LDS instructions are needed to load data first from shared memory into registers, most of the instructions executed in SGEMM are FFMA and LDS instructions. For instance, in our SGEMM implementation with 1024x1024 matrix size, 80.5% of instructions executed are FFMA instructions and 13.4% are LDS.64 instructions. So essentially, in our analysis, we define a few key parameters and study the instruction throughput mixing FFMA and LDS.X instructions while varying these parameters.

The rest of this section is our analysis of SGEMM's performance upper bound. We show that the analysis can give good insights about how to optimize a specific kernel (SGEMM) and help us to understand the performance result.

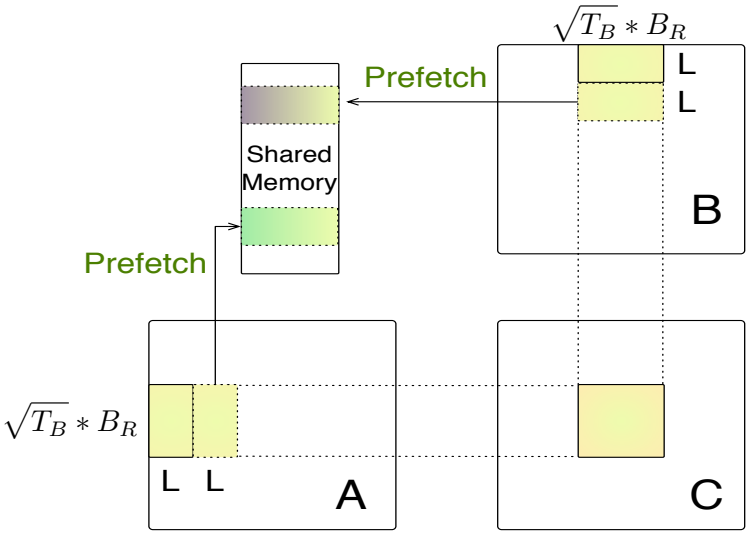

Figure 4.1: SGEMM Implementation

```
DO {
        //Calculate the addresses to load data from global memory
        Address_Calculation();
        //Prefetch A & B from global memory
        Prefetch_A_GlobalMem();
        Prefetch_B_GlobalMem();
        //Loading prefectched A & B from shared memory
        LDS_A_SharedMem();
        LDS_B_SharedMem();
        //Multiply A & B elements and accumulate on C
        FFMA_C_A_B();
        //Synchronize before storing the prefecthed data
        __syncthreads();
        //Store the prefetched data into shared memory
        STS_Prefetched_A();
        STS_Prefetched_B();
        //Synchronize
        __syncthreads();
}
//Loop Ending Condition
While(ADDR_B < LOOP_END)
```

Listing 4.1: GEMM Main Loop Pseudocode

4.3.1 Using Wider Load Instructions

To achieve better performance, it is essential to minimize auxiliary instructions' percentage. By auxiliary instructions, we mean non-math instructions, especially LDS instruction. The assembly code for CUDA sm_20 (GF110 Fermi GPU) and sm_30 (GK104 Kepler GPU) provides SIMD-like LDS.64 and LDS.128 instructions to load 64bit and 128bit data from the shared memory. Using wider load instructions can reduce the total number of LDS instructions. To use these two instructions, the start address in shared memory should be 64bit and 128bit aligned. Also, the indices of registers need to be 2 and 4 aligned respectively. With the native assembly language, it is possible for us to carefully design the data layout and register allocation to satisfy these requirements.

According to our benchmarks, on Fermi GPU, the peak throughput for LDS instruction is 16 32bit-operations per shader clock per SM. Using LDS.64 instructions does not increase the data throughput and the LDS.128 instruction normally leads to 2-way shared memory bank conflict on Fermi GPU. LDS.128 has the throughput of only 2 thread instructions per shader cycle on one SM. In other words, the LD/ST units need 16 shader cycles to process one LDS.128 warp instruction. On Kepler GPU, the throughput for LDS operation is measured as 33.1 64bit operations per shader clock per SM. Using the 32bit LDS operation actually decreases the data throughput in half comparing with using LDS.64 instructions and properly used LDS.128 instruction does not introduce penalty.

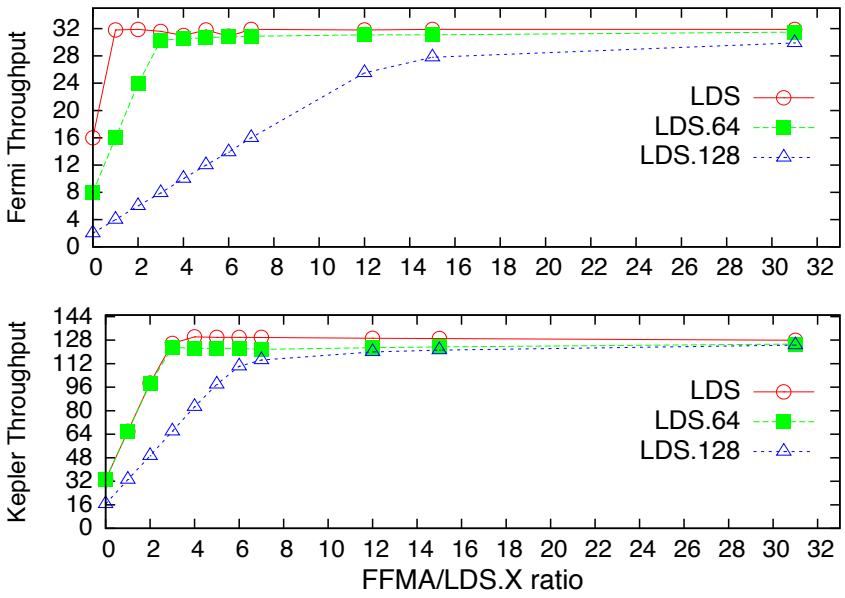

Figure 4.2: Thread Instruction Throughput Mixing FFMA and LDS.X

Figure 4.2 illustrates the instruction throughput of mixing FFMA and LDS.X instructions. While gradually increasing the ratio of FFMA instructions to LDS instructions, the overall instruction throughput approaches the FFMA's peak processing throughput. The instruction ratio of FFMA to LDS.X depends on the algorithm parameters such as register blocking size. Ap-

parently, the overall performance does not always benefit from using wider load instructions. However, the compiler might choose to use the wider load instructions based on the data alignment in the shared memory. With the native assembly language, it is possible for us to carefully design the data layout and use the best instruction type.

4.3.2 Register Blocking

As in Table 1.1, the scheduler of GT200 GPU can issue one warp instruction per shader cycle and since there are 8 SPs per SM, SPs need 4 shader cycles to process one warp instruction. Apparently, as the issuing throughput is higher than the SP's processing throughput, math instructions executed in SPs cannot fully utilize the scheduler's issuing throughput. So the scheduler has some 'free cycles' to issue other type of instructions. NVIDIA introduces the concept of dual-issue which means that the scheduler can use the 'free cycles' to issue other instructions to corresponding functional units, like SFUs (Special Functional Unit). The theoretical peak performance for math instructions is calculated as the sum of SPs' and SFUs' performance.

On Fermi GPUs, SM are redesigned with 2 warp schedulers and 32 SPs. Each warp scheduler, equipped with one dispatch unit, issues instructions to 16 SPs. With an issue rate of one warp instruction per shader cycle, the schedulers' ability could be fully utilized by 32 SPs. The theoretical peak performance for math instructions comes from the SPs' performance. The percentage of other instructions becomes an issue when there are many auxiliary instructions: there are fewer cycles left for schedulers to issue *useful* instructions like FFMA.

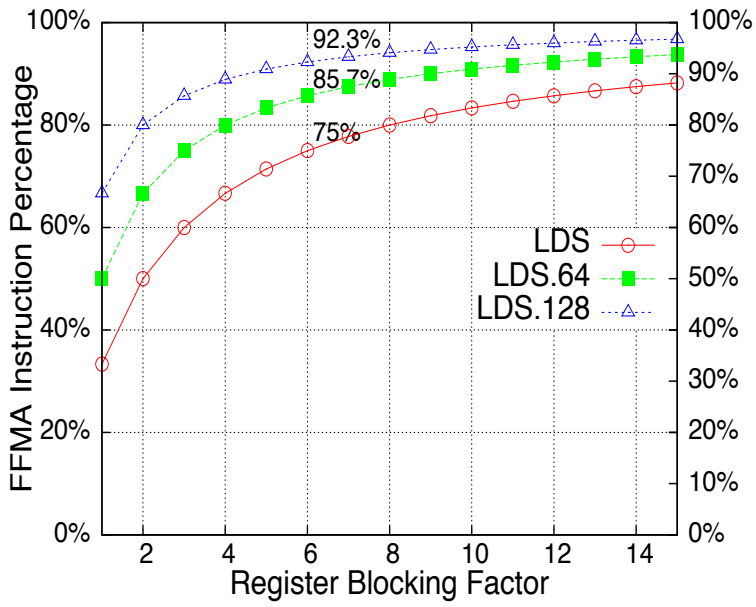

Figure 4.3: FFMA Instruction Percentage in SGEMM Main-loop with Different Register Blocking Factors

For Fermi and Kepler GPUs, according to the output of the disassembler cuobjdump, data from shared memory cannot be used as operands of arithmetic instructions like FFMA. The instruction LDS is needed to first load data from shared memory to register before the math instructions operate on the data. In the worst case, without any register reuse, 2 LDS instructions are needed to fetch data for 1 FFMA instruction in the SGEMM main loop. In that case, only 1/3 of the instructions are floating point operations. Blocking is a well-known technique to better utilize memory hierarchy for scientific programs [55, 59]. To increase the percentage of math instructions, register blocking is needed. We illustrate the percentage of FFMA instructions varying register blocking factors in Figure 4.3.

If 6-register blocking is used (which is the case of our SGEMM implementation on Fermi GPU), the FFMA/LDS.X ratios are 3:1, 6:1, and 12:1 if shared memory accesses are implemented with LDS, LDS.64 and LDS.128 respectively. The percentage of FFMA instructions is 75%, 85.7% and 92.3%. On Fermi GPU, the overall instruction throughputs for one SM in these cases are 31.3, 30.4 and 24.5 thread instructions per shader clock. Because using LDS.128 instruction may lead to extra penalties, even if all the accesses to shared memory are implemented with LDS.128, in the best case we can only achieve around 71% ($\frac{24.5}{32} * 92.3\%$) of SMs' single precision floating point performance. Also, in many cases, a lot of padding in shared memory has to be used to get proper data alignment. Apparently, it is not worth the programming effort to mix FFMA with LDS.128 for SGEMM on the Fermi GPU.

The new Kepler GPU (GTX680) has 192 SPs on the redesigned SM or SMX. Each SMX has 4 warp schedulers, each of which has 2 dispatch units. Shader cycle and core cycle are the same. Similar as on Fermi GPU, data has to be loaded first into registers and then can be fed into FFMA instructions. We also face the problem of increasing the percentage of FFMA instructions in the program. Register blocking is necessary.

4.3.3 Active Threads on SM

Normally, the more active threads one SM executes, the higher performance the GPU can achieve. Since register and shared memory resource is limited per SM, only a limited set of warps can be executed concurrently (T_{SM}).

$$R_T \leq R_{Max} \tag{4.1}$$

$$T_{SM} * R_T \leq R_{SM} \tag{4.2}$$

The registers that each thread can utilize (R_T) is less than or equal to 63 on Fermi and Kepler GPUs (R_{Max}). Furthermore, the register budget of the active warps cannot exceed the SM's register amount (R_{SM}) (Equation 4.2).

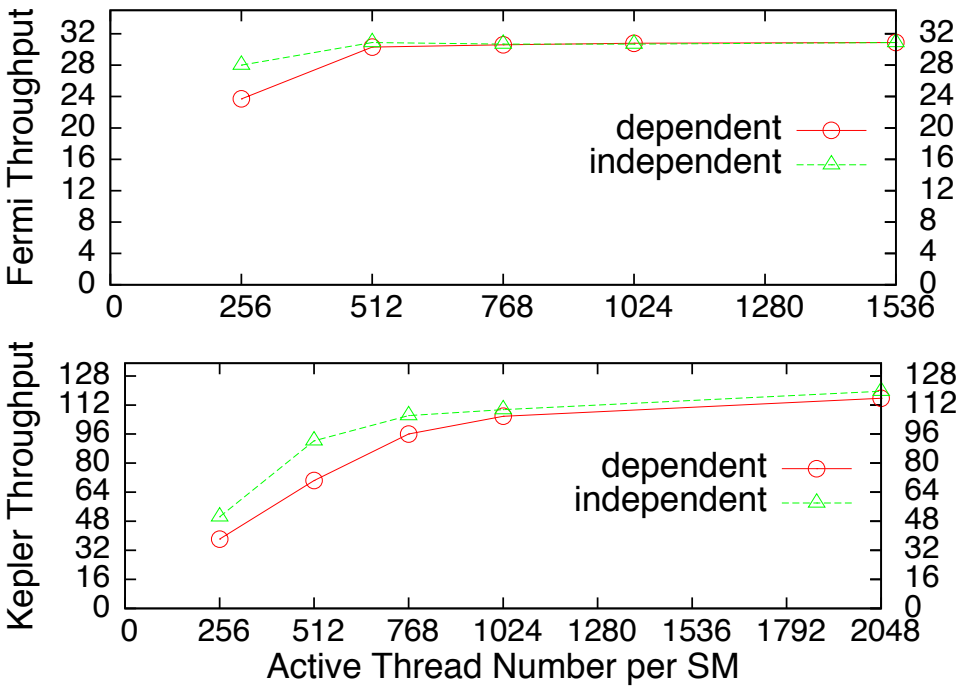

Figure 4.4: Instruction Throughput Mixing FFMA and LDS.64 with Ratio of 6:1

Figure 4.4 illustrates the instruction throughput mixing FFMA and LDS.64 instructions with ratio 6:1 under different number of active threads on one SM. We tested two cases. In the first case ('independent' in Figure 4.4), 6 FFMA and 1 LDS.64 instructions are all independent. In the second case ('dependent' in Figure 4.4), 6 FFMA instructions are dependent on one LDS.64 instruction. The second case is closer to the actual implementation of SGEMM. On Fermi GPU, with 512 active threads, the instruction throughput of the second case is already close to the best situation. On Kepler GPU, however, with fewer than 1024 active threads, the Kepler GPU is very sensitive to the dependences between instructions.

In our analysis, the L1 and L2 cache do not devote to the peak performance. L1 cache is not coherent for different SMs and just reduces the latency of accessing some local data. For L2 cache, since the executing sequence of different C sub matrices cannot be controlled by software and if we consider that after some cycles, the blocks executing on different SMs are computing C sub matrices from random positions, there will be little chance for different SMs getting a hit in L2 cache.

4.3.4 Register and Shared Memory Blocking Factors

Larger register blocking size can introduce more register reuse within one thread and higher percentage of FFMA instructions. However, the register blocking size is limited by the register resource on the SM and the instruction set constraint. With a register blocking factor B_R, if we only consider the registers needed for blocking, we can describe the resource constraint as Equation 4.3.

$$B_R^2 + B_R + 1 < R_T \leq R_{Max} \tag{4.3}$$

This loose condition for register blocking factor B_R can be used to roughly estimate B_R. B_R^2 is the register set needed to hold C sub-matrix per thread, B_R is one column/row of A or B sub-matrix. For instance, with maximum 63 registers per thread, $B_R \leq 7$.

As depicted in Figure 4.1, $T_B * B_R^2$ is the size of the C sub-matrix per block (each block has T_B threads) and $\sqrt{T_B * B_R^2} * L$ is the size of a sub-matrix for A or B (L is the stride). To overlap the data transfer and the computation, extra registers are needed to fetch data from global memory to shared memory since no direct data transfer is provided between the two memory space. The stride L needs to be chosen such that each thread loads the same amount of data (Equation 4.4).

$$(\sqrt{T_B} * B_R * L)\%T_B = 0 \tag{4.4}$$

Considering data prefetching from global memory and a few registers to store the addresses of matrices in global memory and shared memory (R_{addr}), the overall strict constraint for register blocking factor can be described as Equation 4.5.

$$B_R^2 + \frac{2 * \sqrt{T_B} * B_R * L}{T_B} + B_R + 1 + R_{addr} \leq R_T \leq R_{Max} \tag{4.5}$$

Since shared memory is allocated in block granularity, for Blk active blocks, $Blk * 2 * \sqrt{T_B} * B_R * L$ is needed to store prefetched global memory data (Equation 4.6). The shared memory blocking factor can be defined in Equation 4.7. With the shared memory blocking factor B_{Sh}, the performance bounded by global memory bandwidth can be roughly estimated using Equation 4.8.

$$Blk * 2 * \sqrt{T_B} * B_R * L \leq Sh_{SM} \tag{4.6}$$

$$B_{Sh} = \sqrt{T_B * B_R^2} \tag{4.7}$$

$$\frac{P_{MemBound}}{\#GlobalMem\_bandwidth} = \frac{2 * {B_{Sh}}^2}{2 * B_{Sh} * 4} \tag{4.8}$$

4.3.5 Potential Peak Performance of SGEMM

The instruction factor F_I is the ratio of FFMA instructions in the SGEMM main loop (We only consider FFMA and LDS.X instructions here). It depends on the choice of LDS.X instruction and register blocking factor B_R (Figure 4.3). For instance, if LDS.64 is used with register blocking factor 6, $F_I = 0.5$.

The throughput factor F_T is a function of register blocking factor (B_R), number of active threads (T_{SM}), throughput of SPs ($\#SP\_TP$), LD/ST units ($\#LDS\_TP$) and dispatch units ($\#Issue\_TP$)) (Equation 4.9). The function f for Fermi and Kepler

GPUs is illustrated in Figure 4.2 and in Figure 4.4 (only shows LDS.64) and obtained through benchmarks varying these parameters.

$$F_T = f(B_R, \#Issue\_TP, \#SP\_TP, \#LDS\_TP, T_{SM}) \tag{4.9}$$

With the register blocking factor B_R, the instruction factor F_I and the throughput factor F_T, the performance bounded by SMs' processing throughput is estimated as Equation 4.10 and the overall performance is as Equation 4.11.

$$P_{SMBound} = \frac{B_R^2}{B_R^2 + B_R * 2 * F_I} * F_T * P_{theoretical} \tag{4.10}$$

$$P_{potential} = min(P_{MemBound}, P_{SMBound}) \tag{4.11}$$

With the previous analysis, we can estimate the performance upper bound of SGEMM on Fermi and Kepler GPUs. On the Fermi GPU for instance, because of the hard limit of 63 registers (R_{Max}) per thread, considering prefetching and using the strict condition of Equation 4.5, the maximum blocking factor is only 6. The detailed register allocation is illustrated in Section 4.4.2. With the register blocking factor of 6, the register resource per SM can support up to 512 threads. Using Equation 4.4, we choose 256 threads per block.

To easily program the data prefetching, according to Equation 4.4, L could be 8, 16, 24, ... Considering the condition in Equation 4.5, we choose L as 16. With a 6-register blocking factor, mixing LDS or LDS.64 with FFMA instructions, the throughput can achieve close to 32 thread instructions per shader clock per SM. Using a LDS.64 instruction can increase the FFMA instruction percentage to 85.7% from 75% (using LDS). Though LDS.128 instruction can provide higher percentage of FFMA instructions, the instruction processing throughput is too low.

According to Equations 4.8, 4.10 and 4.11, the performance is bounded by SMs' processing throughput, and the potential peak is about 82.5% ($\frac{6^2}{6^2+6*2*0.5} * \frac{30.8}{32}$) of the theoretical peak performance for SGEMM. The main limitation comes from the nature of the Fermi instruction set and the limited issue throughput of schedulers.

It is similar to estimate the performance upper bound of SGEMM on Kepler GPU as Fermi GPU. The Kepler GPU (GK104) instruction set is very close to that of Fermi GPU. It means that the limit of 63 registers per thread still exists. Thus, 6-register blocking is also applicable. And the register resource can support 1024 active threads per SM (64K 32bit registers per SM). We can choose either 256 or 1024 threads per block. Similarly, if we use LDS.64 instructions, the FFMA instruction percentage is 85.7%. If we use LDS.128 instructions (need padding or data layout transform), the FFMA instruction percentage is 92.3%.

Similarly, according to Equations 4.8, 4.10 and 4.11, the performance is bounded by SM's processing throughput, and the potential peak is about 54.6% ($\frac{6^2}{6^2+6*2*0.5} * \frac{122.4}{192}$) of the theoretical peak performance for SGEMM using LDS.64 instructions. Using LDS.128 instructions, the potential peak is about 57.6% ($\frac{6^2}{6^2+6*2*0.25} * \frac{119.9}{192}$) of the theoretical peak. The main limitation factors are still the nature of instruction set and the limited issue throughput of schedulers.

4.4 Assembly Code Level Optimization

The estimated performance upper bound is a limit that an actual implementation cannot exceed. It can be a little optimistic since we only consider the major performance degradation factors. Besides the considered parameters, there might be other aspects which can limit the performance. The 'real' upper bound or the best possible performance is between the estimated upper bound and the achieved performance.

Depending on whether to apply transpose operation on input matrix A or B, there are 4 variations for GEMM kernel. Figure 4.5 illustrates the performance of four SGEMM variations from CUBLAS and our implementation (ASM) with 2400x2400 and 4800x4800 matrices. On GTX580 GPU, we achieve around 74.2% of the theoretical peak performance, i.e., about 90% of the estimated performance upper bound, which we think is good enough. In our analysis, we only study the two main instruction types. There are other auxiliary instructions which do not devote to the GFLOPS. And also, we do not consider the effect of barriers which will harm the performance too. We show that the 'real' upper bound is within this 10% and future optimization is unlikely to achieve a lot of speedup. On Kepler GPU, although we cannot provide the optimal controlling information as discussed in section 4.2.2, we achieve around 77.3% of the estimated upper bound. Similar to Fermi GPU, there are some factors we do not consider in our analysis. The larger gap between our achieved performance might be due to our very limited knowledge of the undisclosed scheduling information of Kepler GPU, which is critical to performance or to some hidden characteristics that we are not able to discover due to limited documentation. Figure 4.6 illustrates the performance comparison on Fermi GPU between our implementation (assembly), CUBLAS from CUDA 4.1 and MAGMA library [67]. Figure 4.7 is the performance comparison on Kepler GPU between our implementation (assembly), CUBLAS from CUDA 4.2 and MAGMA library.

The rest of the section briefly describes our optimizations on assembly code level of SGEMM.

4.4.1 Optimization of Memory Accesses

Assembly code level optimization of memory accesses is similar to high level language optimization. Global memory requests from the threads within a warp could be grouped (coalesced) into one or more memory transactions depending on the compute capability of the device and the memory accessing pattern. To access global memory efficiently, generally it is better to let threads in a warp access continuous data elements in global memory to get coalescing. Considering the majority of instructions in the SGEMM main loop are FFMA and LDS, and it is critical to reduce the number of LDS instructions (using LDS.64 or LDS.128), sub-matrices in shared memory should be grouped such that each thread accesses continuous B_R data elements. Also, proper padding needs to be applied to reduce shared memory access conflicts and satisfy the alignment restriction of the LDS instruction.

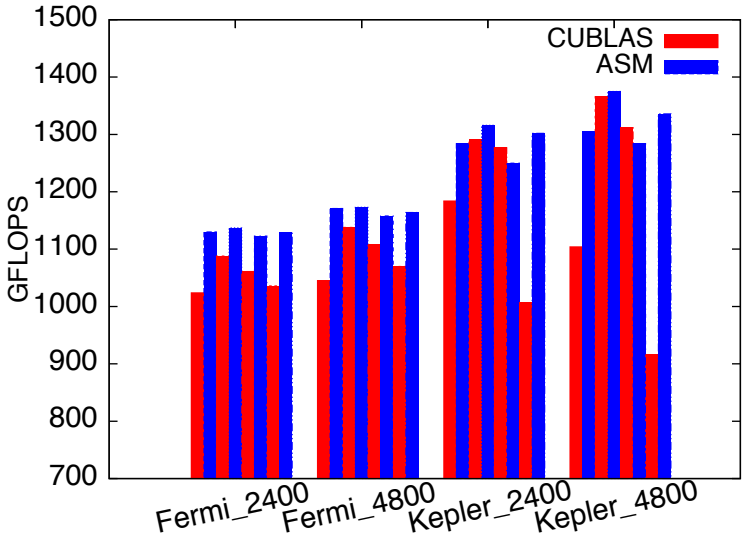

Figure 4.5: SGEMM Performance of CUBLAS and Our Implementation on Fermi and Kepler GPUs

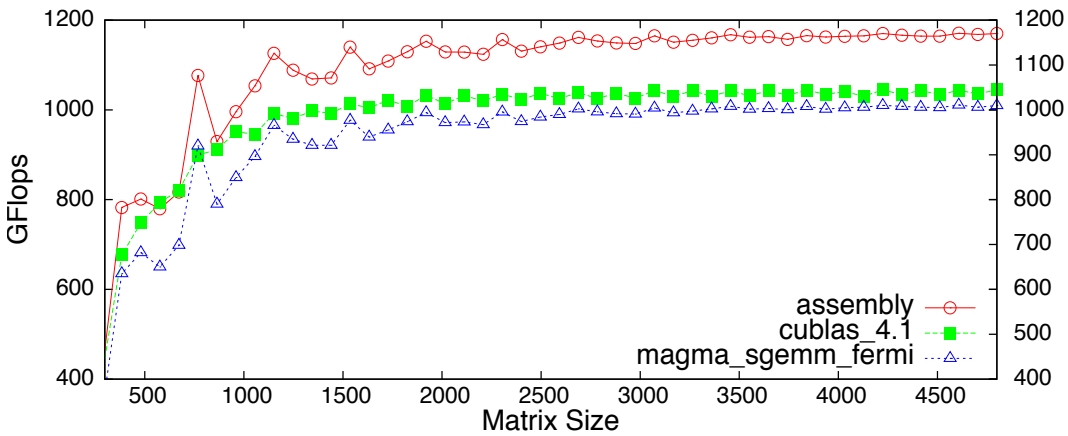

Figure 4.6: SGEMM_NN Performance on GTX580

4.4.2 Register Spilling Elimination

The register resource is 32K 32-bit registers per SM for the Fermi GPU and each thread can use a maximum of 63 registers. The register R1 is normally occupied as stack pointer. According to our analysis, the number of per-thread registers with prefetching is at least $B_R^2 + \frac{2*\sqrt{T_B}*B_R*L}{T_B} + B_R + 1 + R_{index}$. With the register blocking factor of 6 for Fermi GPU, the register allocation of our implementation is as the following. Note that we use 32bit addressing to save address registers.

1. B_R^2, 36 registers to save intermediate result for C matrix.
2. $\frac{2*\sqrt{T_B}*B_R*L}{T_B}$, 12 registers to prefecth A and B from global memory.

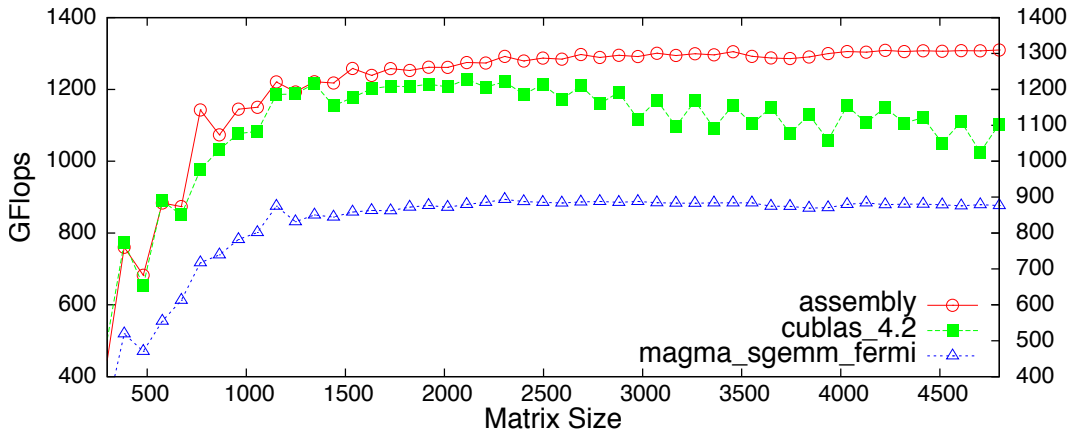

Figure 4.7: SGEMM_NN Performance on GTX680

3. $B_R + 2$, 6 registers to load A from shared memory and 2 registers to load B from shared memory during the main loop. Using 2 registers for B is because LDS.64 instruction is used.
4. 2 registers. Track of A, B in global memory during the prefetching.
5. 1 register to store the loop end condition.
6. 2 registers. Track of A, B in shared memory during the prefetching.
7. 2 registers. Track of A, B in shared memory in the main loop.

In all, 63 registers are used. Since we do not need thread stack, R1 is used to store the loop end condition in our code. Therefore, we are able to fully eliminate the register spilling.

4.4.3 Instruction Reordering

Generally, we try to interleave different instruction types to get better balance between functional units within one SM and better instruction throughput. We apply the following simple reordering optimizations:

1. In the main loop, between the 2 barriers are all shared memory accesses. By moving address calculation from start of the loop to mix with the shared memory accesses, we can achieve better performance.

2. Interleaving prefetching from global memory with FFMA and LDS instructions can benefit performance.

4.4.4 Register Allocation for Kepler GPU

As we describe in Section 4.2.3, to get the best throughput, the 3 source registers of FFMA instructions should reside on 3 different banks if they are different. In our current implementation, 6-register blocking is used. 6 registers are used to load A from

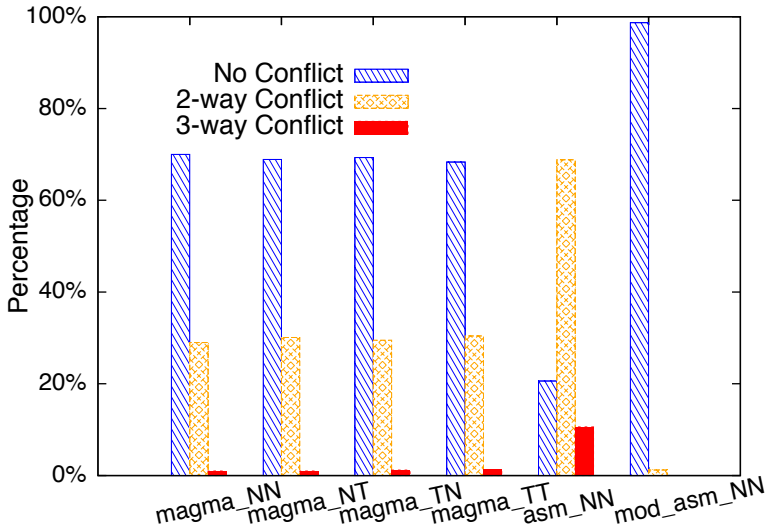

Figure 4.8: Register Conflict of FFMA Instruction

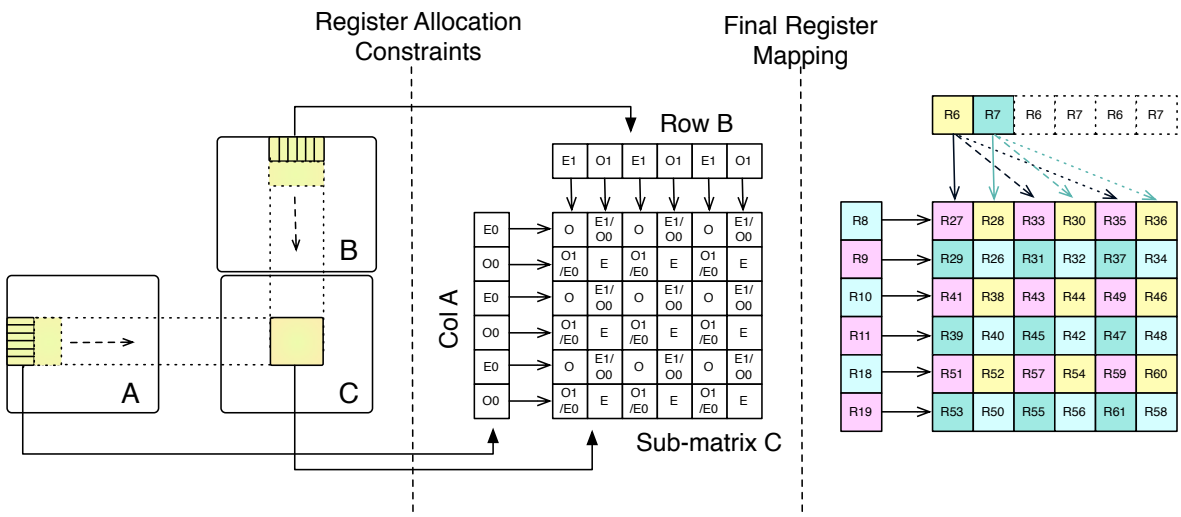

Figure 4.9: Register Allocation

the shared memory and 2 registers to load B from the shared memory in the main loop. 36 different registers (R26∼R61) hold the C sub-matrix. In this implementation, register spilling is eliminated.

As in Figure 4.8, around 30% of the FFMA instructions in the MAGMA [67] SGEMM binary for the Kepler GPU (nvcc generated) have the 2-way register bank conflict and 1% of the FFMA instructions have the 3-way register bank conflict. In our first version of SGEMM_NN on GTX680, which achieves around 1100GFLOPS, 68.8% of the FFMA instructions have the 2-way register bank conflict, and 10.6% of the 3-way

conflict. After applying the optimization, the modified version, which achieve around 1300GFLOPS, has only 1.2% of the 2-way FFMA register bank conflict and the 3-way conflict is fully removed.

Our optimization is depicted in Figure 4.9. In the SGEMM main loop, at each stage, one column from matrix A and one row from matrix B are processed. To use the register blocking and the LDS.64 instructions, at least 6 and 2 different registers are needed for column A and row B. Of course, there are many possible implementations, here we describe one possibility. We select registers from E0 and O0 for column A. Row B uses registers from E1 and O1. Then we use the first table in Figure 4.9 as the constraints of register allocation. In the final mapping stage, we make sure that 36 registers of C sub-matrix have 9 registers on each bank and had our register allocation as the second table, which does not have any register bank conflict to compute the 36 elements from the C sub-matrix.

4.4.5 Opportunity for Automatic Tools

Our study emphasizes that for Fermi and Kepler GPUs, it is essential to study the impact of algorithm parameters on instruction throughput to get insight into the performance result. The main optimization opportunity comes from the allocation of registers. For example, the four SGEMM variations of MAGMA library compiled with nvcc spill at least 10 registers (40 Bytes) on the Kepler GPU. When the active thread number is 512, at least 20KB L1 cache is needed to make sure that the spilled data stays in the L1 cache. However, since normally the unified 64KB shared memory/L1 cache is configured as 48KB shared memory and 16KB L1 cache, some data will be spilled out of L1 cache. As the active threads increase, more data is spilled out of L1 cache and the performance will be harmed. We already show that with careful design, register spilling could be eliminated. We also show that around 30% of FFMA instructions in the nvcc-generated SGEMM binary from MAGMA library have register bank conflict. We propos a simple solution in Section 4.4.4. It is possible for optimizers to detect the loop structure and remove the conflicts with proper register allocation.

An automatic tuning tool normally needs to explore a large design , and evaluate the performance of many configurations [52, 61, 80, 81]. It may take a significant amount of time. Normally, the automatic tuning tool is application-dependent and each includes several efficient optimizations for the specific application. To build the tool relies on the developers' understanding of the application and optimization experience. With the proposed analysis approach, we can better understand which parameters are critical to the performance. The estimated upper bound actually corresponds to a set of parameters and optimization options. This knowledge can help an automatic tool to explore the design space in a relatively small region. And of course by comparing the performance of an automatic tool's output code and the estimated performance upper bound, we can judge whether the optimized version is good enough.

In our analysis, to study the instruction throughput mixing FFMA and LDS.X instructions, we manually write some benchmarks varying several key parameters such as instruction type choice (LDS.X), the mixing ratio, the blocking factor, the instruc-

tions' dependence, active threads and study these parameters' impact on the instruction throughput. For many applications with few major instruction types, a similar approach can be used to estimate the performance upper bound. The difference would be the chosen instruction types and their mixing pattern (mixing ratio, dependence, etc.). Systematic and automatic development of a set of microbenchmarks to help to estimate the performance upper bound of other applications is possible. A family of assembly level microbenchmarks could be defined and evaluated in order to provide a small database of performance references that could be used by the auto-tuning tool, and also the developer to transform the code for performance. Generally, the assembly level microbenchmarks can also help to understand the difference between different GPU architectures. For example, the benchmarks illustrated in Figure 4.4 show the increasing need for active threads on Kepler GPU. Assembly level benchmarking requires an assembly tool chain which is missing from the official support. We manage to make it work on Fermi GPU. But on Kepler, there are some issues like the hidden scheduling information, which we cannot fully decrypt.

4.5 Summary

In this work, we have proposed an approach to analyze GPU applications' performance upper bound. Different from existing works on GPU performance models, our approach relies on application analysis and assembly level benchmarking. Essentially, in our analysis, we have studied the instruction throughput mixing FFMA and LDS.X instructions. We manually wrote some benchmarks varying several key parameters such as instruction type choice (LDS.X), the mixing ratio, the blocking factor, the instructions' dependence, active threads and studied these parameters' impact on the instruction throughput. For many applications with few major instruction types, we can use the similar approach. The difference would be the chosen instruction types and their mixing pattern (mixing ratio, dependence, etc.). Systematic and automatic development of a set of microbenchmarks to help to estimate the performance upper bound of other applications is possible. For an automatic tool, it is much easier to set up and evaluate a set of microbenchmarks using assembly code with a few instruction types than to automatically and safely transform the application's high level code. Generally, the assembly level microbenchmarks can also help to understand the difference of different GPU architectures. For example, the benchmarks illustrated in Figure show the increasing need of active threads on Kepler GPU.

As an example, we analyze the potential peak performance of SGEMM on Fermi and Kepler GPUs. We show that the nature of the Fermi (Kepler) instruction set and the limited issue throughput of schedulers are the main limitation factors for SGEMM to approach the theoretical peak performance. The general guideline is to reduce the auxiliary instructions and increase the FFMA instruction's percentage. Proper register allocation, shared memory data layout and memory access pattern need to be carefully designed to minimize the impact of memory accesses on performance. We also show that our analysis can help to decide some critical algorithm parameters and show how

much optimization space exists. Guided by the analysis, we further optimize the four SGEMM kernel variations and achieve better performance on Fermi GPU (around 5% on average for large matrices) than highly optimized routine provided by NVIDIA.

Conclusion

In recent years, general computing on GPU processors has become an interesting research topic. Like many other dedicated parallel architectures, current compilers fail to generate efficient parallelized machine code directly from legacy serial code. These architectures normally have different programming models and dedicated device APIs to launch tasks. Developers have to familiarize themselves with these programming models and device APIs through a fairly long learning curve. Although many automatic tuning tools have been developed to generate optimized codes for specific architectures and tasks, the existing approaches are still not efficient and general enough. Even expert developers need to spend much time on optimization to achieve good performance.

In the serial programming era, for architecture researchers, the general focus is how to build a more powerful processor. For developers, the underlying architecture is transparent. Developers only need to focus on the algorithm-level optimization. The bridge between the high level serial code and the hardware is well maintained by compilers. In the many-core or parallel-programming era, architects need to consider how to assign on-die resource for different cores and power becomes an important design factor. Developers need to learn more architectural characteristics to make full use of the hardware potential. For developers and performance-tuning researchers, the boundary between software design and hardware is becoming vague.

The ultimate solutions for the problems we face today might include, first, intelligent parallel compilers, which can generate very efficient parallelized code based on the architecture details, make the underlying hardware transparent to developers and things go back to the way in the serial programming era; second, intelligent processors, which can efficiently execute serial code in a parallel pattern, make the compilers' and developers' work much easier; third, without very intelligent compilers and processors, programmers and performance-tuning researchers develop a systematic and analytical way of performance optimization on new architectures. The second possibility is just a wild guess and the first solution seems to be more likely to happen in the not so long future. In our opinion, the third approach is the most possible solution.

For each of the new parallel architectures, normally three questions are raised.
Q1: Why does an implementation achieve a certain performance?
Q2: How we can improve the performance?
Q3: What is supposed to be the upper-bound performance which an application cannot exceed on a certain architecture?

Basically, in our work, we follows this train of thought.

To answer the first question, researchers generally rely on analytical or simulation methods. Simulators are powerful tools to evaluate new hardware design options and more useful for architecture researchers. The analysis approach is much easier to develop and requires less knowledge of the real hardware details which is difficult to get for commercial processors today. Apparently, the more hardware details we introduce in the analytical models, the more accurate the models should be. For example, in Chapter 2, we introduce simple data-flow models of lattice QCD application on Cell B.E. and GPU processors. Essentially, we utilize the computation and computation ratio and only have a evaluation of the rough performance estimation. To get more details of an implementation, we have developed an analytical method to predict CUDA application's performance using assembly code for GT200 generation GPU. We use a timing estimation tool (TEG) to estimate GPU kernel execution time. TEG can give the timing estimation in cycle-approximate level. Thus it allows programmers to better understand the performance results.

To answer the second question, developers normally rely on their expert experience and event statistics collected from hardware counters. Also, there are many literatures about optimization experiences on certain architectures. For a specific applications, besides these approaches, we have utilized TEG to estimate how much penalty different performance penalties can introduce. Using TEG, the performance penalties are associated with instructions' execution latency and throughput. So we can simply estimate the penalties' effects from TEG by changing the instruction latency and throughput information.

There are fewer studies on the third question. The conventional way of thinking of performance optimization problem is from bottom to top, which means that researchers study how much performance gain we can get by applying certain optimization combinations. As we have argued before, different optimization options normally have strong interactions. It is difficult to separate the effects of different options. With this approach, we can only get a performance evaluation of a set of predefined optimizations. Apparently, we do not have the confidence to find all the best optimizations for each application. So, instead of looking at this problem from bottom to top, we try to start from an optimistic situation which the achievable performance cannot exceed. We have developed an approach to estimate GPU applications' performance upper bound based on application analysis and assembly code level benchmarking. With the performance upperbound of an application, we know how much optimization space is left and can decide the optimization effort. Also with the analysis we can understand which parameters are critical to the performance.

There is no doubt that in the near future, the hardware accelerators would likely to have many more cores on one processor die. The processor's structure might be one super-scalar monster core, which is very complicated and designed for serial computing, plus many small and simple cores. The processor could also be composed of a sea of simple cores, like the GPU processor today. Either way, we can speculate that the parallel part of an application is processed by the sea of smaller cores. So it would be difficult to use simulation tools to study the performance result. The analytical approach should be the choice for programmers and performance-tuning researchers to

answer the three basic questions for future architectures before very smart compilers appear. We believe that for each new architectures, a set of systematic tools or models should be developed to understand the achieved performance, the main performance penalties of an implementation and the performance upper bound of an application on the architecture.

This work is supported by French National Research Agency (ANR) through COSINUS program (project PETAQCD N° ANR-08-COSI-010).

Bibliography

[1] Netlib. `http://www.netlib.org/blas/`.

[2] Nvidia. NVIDIA CUDA C Programming Guide 4.2.

[3] Albonesi, D. H., and Koren, I. An analytical model of high performance superscalar-based multiprocessors. In *In Proceedings of Conference on Parallel Architectures and Compilation Technology (PACT* (1995), pp. 194–203.

[4] Alexandru, A., Pelissier, C., Gamari, B., and Lee, F. X. Multi-mass solvers for lattice qcd on gpus. *J. Comput. Phys. 231*, 4 (Feb. 2012), 1866–1878.

[5] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. The tera computer system. In *Proceedings of the 4th international conference on Supercomputing* (New York, NY, USA, 1990), ICS '90, ACM.

[6] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[7] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A view of the parallel computing landscape. *Commun. ACM 52*, 10 (Oct. 2009), 56–67.

[8] Asfermi. `http://code.google.com/p/asfermi/`.

[9] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-m. W. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 105–114.

[10] Baier, H., Boettiger, H., Drochner, M., Eicker, N., Fischer, U., Fodor, Z., Frommer, A., Gomez, C., Goldrian, G., Heybrock, S., Hierl, D., Hüsken, M., Huth, T., Krill, B., Lauritsen, J., Lippert, T., Maurer, T., Mendl, B., Meyer, N., Nobile, A., Ouda, I., Pivanti, M., Pleiter, D., Ries, M.,

Schäfer, A., Schick, H., Schifano, F., Simma, H., Solbrig, S., Streuer, T., Sulanke, K., Tripiccione, R., Vogt, J., Wettig, T., and Winter, F. QPACE – a QCD parallel computer based on Cell processors. *ArXiv e-prints* (Nov. 2009).

[11] Bakhoda, A., Yuan, G., Fung, W., Wong, H., and Aamodt, T. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (april 2009), pp. 163 –174.

[12] Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 225–234.

[13] Baskaran, M. M., Ramanujam, J., and Sadayappan, P. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction* (Berlin, Heidelberg, 2010), CC'10/ETAPS'10, Springer-Verlag, pp. 244–263.

[14] Bell, N., and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 18:1–18:11.

[15] Belletti, F., Bilardi, G., Drochner, M., Eicker, N., Fodor, Z., Hierl, D., Kaldass, H., Lippert, T., Maurer, T., Meyer, N., Nobile, A., Pleiter, D., Schaefer, A., Schifano, F., Simma, H., Solbrig, S., Streuer, T., Tripiccione, R., and Wettig, T. QCD on the Cell Broadband Engine. *ArXiv e-prints* (Oct. 2007).

[16] Belletti, F., Schifano, S. F., Tripiccione, R., Bodin, F., Boucaud, P., Micheli, J., P?ne, O., Cabibbo, N., de Luca, S., Lonardo, A., Rossetti, D., Vicini, P., Lukyanov, M., Morin, L., Paschedag, N., Simma, H., Morenas, V., Pleiter, D., and Rapuano, F. Computing for lqcd: apenext. *Computing in Science and Engineering 8* (2006), 18–29.

[17] Bilardi, G., Pietracaprina, A., Pucci, G., Schifano, F., and Tripiccione, R. The potential of on-chip multiprocessing for qcd machines. In *High Performance Computing HiPC 2005*, D. Bader, M. Parashar, V. Sridhar, and V. Prasanna, Eds., vol. 3769 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 386–397.

[18] Boyle, P. A., et al. Hardware and software status of QCDOC. *Nucl. Phys. Proc. Suppl. 129* (2004), 838–843.

[19] Chen, X. E., and Aamodt, T. M. A first-order fine-grained multithreaded throughput model. In *HPCA* (2009), IEEE Computer Society, pp. 329–340.

[20] Choi, J. W., Singh, A., and Vuduc, R. W. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 115–126.

[21] Christ, N. H. Computers for lattice qcd. *Nucl. Phys. B, Proc. Suppl. 83*, hep-lat/9912009 (2000), 111–115.

[22] Clark, M. QCD on GPUs: cost effective supercomputing. In *Symposium on Lattice Field Theory* (2009).

[23] Clark, M. A., Babich, R., Barros, K., Brower, R. C., and Rebbi, C. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications 181* (Sept. 2010), 1517–1528.

[24] Collange, S., Daumas, M., Defour, D., and Parello, D. Barra: A parallel functional simulator for gpgpu. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on* (aug. 2010), pp. 351 –360.

[25] Cui, X., Chen, Y., Zhang, C., and Mei, H. Auto-tuning dense matrix multiplication for gpgpu with cache. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on* (dec. 2010), pp. 237 –242.

[26] Cui, Z., Liang, Y., Rupnow, K., and Chen, D. An accurate gpu performance model for effective control flow divergence optimization. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International* (may 2012), pp. 83 –94.

[27] Davidson, A., and Owens, J. Toward techniques for auto-tuning gpu algorithms. In *Applied Parallel and Scientific Computing*, vol. 7134 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 110–119.

[28] de Veronese, L., and Krohling, R. Differential evolution algorithm on the gpu with c-cuda. In *Evolutionary Computation (CEC), 2010 IEEE Congress on* (july 2010), pp. 1 –7.

[29] del Barrio, V., Gonzalez, C., Roca, J., Fernandez, A., and E, E. Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on* (march 2006), pp. 231 – 241.

[30] Di, P., and Xue, J. Model-driven tile size selection for doacross loops on gpus. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II* (Berlin, Heidelberg, 2011), Euro-Par'11, Springer-Verlag, pp. 401–412.

[31] Dotsenko, Y., Baghsorkhi, S. S., Lloyd, B., and Govindaraju, N. K. Auto-tuning of fast fourier transform on graphics processors. In *PPOPP* (2011), pp. 257–266.

[32] Egri, G., Fodor, Z., Hoelbling, C., Katz, S., Nogradi, D., and Szabo, K. Lattice QCD as a video game. *Computer Physics Communications 177* (Oct. 2007), 631–639.

[33] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst. 27*, 2 (May 2009), 3:1–3:37.

[34] Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2007), MICRO 40, IEEE Computer Society, pp. 407–420.

[35] Gpgpu-sim. `http://www.gpgpu-sim.org/`.

[36] A modular dynamic compilation framework for heterogeneous system. `http://code.google.com/p/gpuocelot/`.

[37] Guo, P., and Wang, L. Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus. In *Computational and Information Sciences (ICCIS), 2010 International Conference on* (dec. 2010), pp. 1154 –1157.

[38] Haller, I., and Nedevschi, S. Gpu optimization of the sgm stereo algorithm. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on* (aug. 2010), pp. 197 –202.

[39] Hill, M., and Marty, M. Amdahl's law in the multicore era. *Computer 41*, 7 (july 2008), 33 –38.

[40] Hong, S., and Kim, H. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 152–163.

[41] IBM. Cell broadband engine. `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`.

[42] Ibrahim, K. Z., and Bodin, F. Implementing wilson-dirac operator on the cell broadband engine. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), ACM, pp. 4–14.

[43] Ibrahim, K. Z., and Bodin, F. Efficient simdization and data management of the lattice qcd computation on the cell broadband engine. *Sci. Program. 17*, 1-2 (2009), 153–172.

[44] Ibrahim, K. Z., Bodin, F., and Pène, O. Fine-grained parallelization of lattice qcd kernel routine on gpus. *J. Parallel Distrib. Comput. 68*, 10 (2008), 1350–1359.

[45] Joseph, P. J., Vaswani, K., and Thazhuthaveetil, M. J. A predictive performance model for superscalar processors. In *MICRO* (2006), IEEE Computer Society, pp. 161–170.

[46] Kamil, S., Chan, C., Oliker, L., Shalf, J., and Williams, S. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (april 2010), pp. 1 –12.

[47] Karas, P., Svoboda, D., and Zemcik, P. Gpu optimization of convolution for large 3-d real images. In *Advanced Concepts for Intelligent Vision Systems*, J. Blanc-Talon, W. Philips, D. Popescu, P. Scheunders, and P. Zemcik, Eds., vol. 7517 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 59–71.

[48] Karkhanis, T. S., and Smith, J. E. A first-order superscalar processor model. In *Proceedings of the 31st annual international symposium on Computer architecture* (Washington, DC, USA, 2004), ISCA '04, IEEE Computer Society, pp. 338–.

[49] Kim, Y., and Shrivastava, A. Cumapz: a tool to analyze memory access patterns in cuda. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE* (2011), IEEE, pp. 128–133.

[50] Kirk, D. Nvidia cuda software and gpu parallel computing architecture. In *Proceedings of the 6th international symposium on Memory management* (New York, NY, USA, 2007), ISMM '07, ACM, pp. 103–104.

[51] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B. C., Ivanov, P., and Fasih, A. Pycuda: Gpu run-time code generation for high-performance computing. *CoRR abs/0911.3456* (2009).

[52] Kurzak, J., Tomov, S., and Dongarra, J. Autotuning gemm kernels for the fermi gpu. *Parallel and Distributed Systems, IEEE Transactions on PP*, 99 (2012), 1.

[53] Lai, J., and Seznec, A. Break down gpu execution time with an analytical method. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools* (New York, NY, USA, 2012), RAPIDO '12, ACM, pp. 33–39.

[54] Lai, J., and Seznec, A. Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization* (2013), CGO '13, IEEE.

[55] Lam, M. D., Rothberg, E. E., and Wolf, M. E. The cache performance and optimizations of blocked algorithms. *SIGPLAN Not. 26*, 4 (Apr. 1991), 63–74.

[56] Lee, S., and Eigenmann, R. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[57] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (Mar. 2008), 39–55.

[58] Liu, Y., and Hu, J. Gpu-based parallelization for fast circuit optimization. *ACM Trans. Des. Autom. Electron. Syst. 16*, 3 (June 2011), 24:1–24:14.

[59] McKellar, A. C., and Coffman, Jr., E. G. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM 12*, 3 (Mar. 1969), 153–165.

[60] Men, C., Gu, X., Choi, D., Majumdar, A., Zheng, Z., Mueller, K., and Jiang, S. B. Gpu-based ultrafast imrt plan optimization. *Physics in Medicine and Biology 54*, 21 (2009), 6565.

[61] Meng, J., Morozov, V. A., Kumaran, K., Vishwanath, V., and Uram, T. D. Grophecy: Gpu performance projection from cpu code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 14:1–14:11.

[62] Meng, J., and Skadron, K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 256–265.

[63] Michaud, P., and Seznec, A. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2001), HPCA '01, IEEE Computer Society, pp. 27–.

[64] Michaud, P., Seznec, A., and Jourdan, S. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *IN PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES* (1999), pp. 2–10.

[65] Motoki, S., and Atsushi, N. Development of qcd-code on a cell machine. *PoS LATTICE 2007* (2007), 040.

[66] Mussi, L., Nashed, Y. S., and Cagnoni, S. Gpu-based asynchronous particle swarm optimization. In *Proceedings of the 13th annual conference on Genetic*

and evolutionary computation (New York, NY, USA, 2011), GECCO '11, ACM, pp. 1555–1562.

[67] Nath, R., Tomov, S., and Dongarra, J. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl. 24*, 4 (Nov. 2010), 511–515.

[68] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. *Queue 6*, 2 (Mar. 2008), 40–53.

[69] Noonburg, D. B., and Shen, J. P. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th annual international symposium on Microarchitecture* (New York, NY, USA, 1994), MICRO 27, ACM, pp. 52–62.

[70] Nukada, A., and Matsuoka, S. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 30:1–30:10.

[71] NVIDIA. `http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf`. PARALLEL THREAD EXECUTION ISA VERSION 3.1.

[72] NVIDIA. Visual profiler. `https://developer.nvidia.com/nvidia-visual-profiler`.

[73] NVIDIA. Geforce gtx 200 gpu architectural overview. `http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf`, 2008.

[74] NVIDIA. Fermi whitepaper. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[75] NVIDIA. GTX680 Whitepaper. `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`, 2012.

[76] NVIDIA. NVIDIA Tesla K20/K20X GPU Accelerators Application Performance Technical Brief. `http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf`, Nov. 2012.

[77] Opencl. http://www.khronos.org/opencl/.

[78] Raimondo, F., Kamienkowski, J. E., Sigman, M., and Slezak, D. F. Cudaica: Gpu optimization of infomax-ica eeg analysis. *Intell. Neuroscience 2012* (Jan. 2012), 2:1–2:8.

[79] Ruetsch, G., and Micikevicius, P. Optimizing matrix transpose in cuda.

[80] Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S.-Z., Stratton, J. A., and Hwu, W.-m. W. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2008), CGO '08, ACM, pp. 195–204.

[81] Schaa, D., and Kaeli, D. Exploring the multiple-gpu design space. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (may 2009), pp. 1 –12.

[82] Seznec, A. `http://www.irisa.fr/alf/downloads/DAL/DAL.htm`. Defying Amdahls Law - DAL.

[83] Sheaffer, J. W., Luebke, D., and Skadron, K. A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), HWWS '04, ACM, pp. 85–94.

[84] Shi, G., Kindratenko, V., and Gottlieb, S. Cell processor implementation of a MILC lattice QCD application. *ArXiv e-prints* (Oct. 2009).

[85] Sim, J., Dasgupta, A., Kim, H., and Vuduc, R. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 11–22.

[86] Spray, J., Hill, J., and Trew, A. Performance of a Lattice Quantum Chromodynamics kernel on the Cell processor. *Computer Physics Communications 179* (Nov. 2008), 642–646.

[87] Taha, T. M., and Wills, S. An instruction throughput model of superscalar processors. *IEEE Trans. Comput. 57*, 3 (Mar. 2008), 389–403.

[88] Tan, G., Li, L., Triechle, S., Phillips, E., Bao, Y., and Sun, N. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 35:1–35:11.

[89] Tang, B., and Miao, L. Real-time rendering for 3d game terrain with gpu optimization. In *Computer Modeling and Simulation, 2010. ICCMS '10. Second International Conference on* (jan. 2010), vol. 1, pp. 198 –201.

[90] Tor M. Aamodt, W. W. F. Gpgpu-sim 3.x manual. http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual.

[91] Ueng, S.-Z., Lathara, M., Baghsorkhi, S. S., and Hwu, W.-M. W. Languages and compilers for parallel computing. Springer-Verlag, Berlin, Heidelberg, 2008, ch. CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15.

[92] Unat, D., Cai, X., and Baden, S. B. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 214–224.

[93] Vranas, P., Bhanot, G., Blumrich, M., Chen, D., Gara, A., Heidelberger, P., Salapura, V., and Sexton, J. C. The bluegene/l supercomputer and quantum chromodynamics. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), ACM, p. 50.

[94] Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76.

[95] Wolfe, M. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (New York, NY, USA, 2010), GPGPU '10, ACM, pp. 43–50.

[96] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. Demystifying gpu microarchitecture through microbenchmarking. In *ISPASS'10* (2010), pp. 235–246.

[97] Yan, D., Cao, H., Dong, X., Zhang, B., and Zhang, X. Optimizing algorithm of sparse linear systems on gpu. In *Chinagrid Conference (ChinaGrid), 2011 Sixth Annual* (aug. 2011), pp. 174 –179.

[98] Yang, Y., Xiang, P., Kong, J., and Zhou, H. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 86–97.

[99] Zhang, Y., Cohen, J., and Owens, J. D. Fast tridiagonal solvers on the gpu. *SIGPLAN Not. 45*, 5 (Jan. 2010), 127–136.

[100] Zhang, Y., and Mueller, F. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 155–164.

[101] Zhang, Y., and Owens, J. D. A quantitative performance analysis model for gpu architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)* (Feb. 2011).

[102] Zhou, Y., and Tan, Y. Gpu-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on* (may 2009), pp. 1493 –1500.

List of Figures

Abstract

This thesis work is funded by the ANR PetaQCD project. We have mainly worked on two topics of GPU performance analysis. We have designed an approach which is simple enough for developers to use and can provide more insight into the performance results. And we have designed an approach to estimate the performance upper bound of an application on GPUs and guide the performance optimization.

First part of the thesis work was presented at Rapido '12 workshop. We have developed an analytical method and a timing estimation tool (TEG) to predict CUDA application's performance for GT200 generation GPU. TEG passes GPU kernels' assembly code and collects information including instruction type, operands, etc. Then TEG can predict GPU applications' performance in cycle-approximate level with the instruction trace and other information collected from Barra simulator. TEG also allows to quantify some performance bottlenecks' penalties.

The second main part of this thesis is going to be presented at CGO '13 conference. We developed an approach to estimate GPU applications' performance upper bound based on application analysis and assembly code level benchmarking. With the performance upperbound of an application, we know how much optimization space is left and can decide the optimization effort. Also with the analysis we can understand which parameters are critical to the performance. As an example, we analyzed the potential peak performance of SGEMM (Single-precision General Matrix Multiply) on Fermi (GF110) and Kepler (GK104) GPUs. Guided by this analysis and using the native assembly language, on average, our SGEMM implementations achieve about 5% better performance than CUBLAS in CUDA 4.1 SDK for large matrices on GTX580. The achieved performance is around 90% of the estimated upper bound performance of SGEMM on GTX580.