Framework pour une exécution efficace sur systèmes GPU et CPU+GPU

Jean-François Dollinger

Université de Strasbourg

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire des sciences de l'ingénieur, de l'informatique et de l'imagerie (ICube)

THÈSE présentée par :

Jean-François DOLLINGER

Soutenue le : **01 juillet 2015**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**
discipline/spécialité : **Informatique**

A framework for efficient execution on GPU and CPU+GPU systems

Thèse dirigée par :	
M. CLAUSS Philippe	Professeur, Unistra, Strasbourg
Rapporteurs :	
M. COHEN Albert	Directeur de Recherche INRIA, ENS, Paris
M. BODIN François	Professeur, IRISA, Rennes
Examinateurs :	
M. HACK Sebastian	Professeur, Universität des Saarlandes, Saarbrücken, Allemagne
M. LOECHNER Vincent	Maître de conférences, Unistra, Strasbourg

Contents

List of Figures

Chapitre 1

Résumé en Français

1.1 Introduction

Le rapide déploiement d'applications graphiques fin des années 80 et le surcoût occasionné en terme de consommation de ressources a poussé à l'utilisation de coprocesseurs spécialisés appelés Graphics Processing Unit (GPU). A leur origine les GPUs présentent une architecture peu flexible n'autorisant qu'un certain spectre d'applications ; seul un nombre restreint d'opérations précâblées étaient exposées aux programmeurs. Avec l'arrivée des langages CUDA en 2007 et OpenCL en 2009, le modèle de programmation s'est affranchi des contraintes de programmabilité archaïques inhérentes aux GPUs d'ancienne génération. Aujourd'hui, les GPUs se révèlent être une alternative viable pour des traitements autres que graphiques, en particulier pour combler l'insatiable demande en performance de la communauté scientifique.

Malgré les efforts de simplification engagés par les fournisseurs de puces graphiques, le portage d'un code pour une exécution efficace sur GPU n'est pas chose aisée. Un programmeur doit non seulement se familiariser avec le modèle de programmation, mais également avoir une connaissance fine de l'architecture et transformer le code pour utiliser le GPU à son plein potentiel.

Les nids de boucles concentrent la majorité du temps d'exécution d'un programme. Le modèle polyédrique est une représentation mathématique d'un nid de boucles affine paramétré, soit un nid de boucles dont les accès mémoire et les bornes de boucles sont exprimés via des combinaisons affines des paramètres et des itérateurs des boucles englobantes. Ce cadre théorique autorise l'application de transformations de code automatiques (e.g. inversion de boucles, tuilage, skewing, etc.) tout en préservant la sémantique initiale. Mes travaux de thèse ainsi que les outils utilisés s'appuient principalement sur le modèle polyédrique.

Des outils de parallélisation automatiques polyédriques tels que PLUTO [25], C-to-CUDA [15], PPCG [145], R-Stream [131], Par4All [7, 9], sont capables de générer du code optimisé pour une architecture cible. Ces outils manipulent des SCoP (Static Control Parts) renfermant une succession de nids de boucles affines. Le processus typique de compilation, du fait de sa nature statique, réduit le champ des opportunités des compilateurs. A défaut de générer du code performant dans toutes les situations, les performances restent généralement moyennes car issues d'heuristiques : il est néces-

saire de considérer les paramètres dynamiques et l'ensemble des ressources de calcul d'un système.

Les performances d'une version de code peuvent varier en fonction du contexte d'exécution ; elles sont fonction des paramètres de nid de boucles. Pour répondre à cette problématique, le "multiversioning" consiste à produire plusieurs versions de code sémantiquement équivalentes au programme original. La meilleure des versions doit être sélectionnée à l'aide d'une prédiction du temps d'exécution sur GPU.

Pour tirer les meilleures performances des systèmes hétérogènes, c'est à dire de systèmes hébergeant des processeurs aux architectures différentes, il est nécessaire de déterminer les affinités entre les codes et les Unités de Calcul (processeur central, GPU, accélérateurs, etc.), appelées UC dans la suite du texte. Nous présentons une méthode dynamique capable de sélectionner l'architecture cible la plus adaptée, tout en tenant compte de l'impact de l'environnement d'exécution. Celle-ci consiste à exécuter simultanément un même code sur plusieurs processeurs et sélectionner le plus rapide. Pour cela nous proposons une technique d'interruption anticipée de codes exécutés sur GPU et CPU.

Une autre méthode que nous avons implémentée consiste à exploiter toutes les ressources de calcul ; elle requiert l'ajustement de la charge de travail attribuée à chaque UC. En effet, pour être performante, la distribution d'un calcul requiert de partager équitablement les temps d'exécution. Nous présentons un ordonnanceur guidé par des prédictions de temps d'exécution et capable, si c'est avantageux, de sélectionner l'architecture adéquate ou d'utiliser conjointement plusieurs UCs pour effectuer un calcul. La taille de la charge de travail est raffinée successivement jusqu'à obtenir des temps d'exécution équivalents sur toutes les UCs engagées.

1.2 Prédiction et sélection adaptative de version de code

1.2.1 Introduction

Rendre les performances d'un programme adaptative au contexte d'exécution est un travail difficile et fastidieux, surtout pour des architectures complexes tels que les CPU généralistes ou les cartes accélératrices. La prédiction des performances d'un code est souvent difficile statiquement. En effet, l'efficacité d'un code varie selon l'architecture, la charge et les jeux de données à traiter. Pour ces raisons, les compilateurs peuvent difficilement prendre des décisions statiquement. Les techniques de compilation iteratives facilitent la recherche de la meilleure des versions pour un contexte donné, mais négligent généralement la variation des paramètres d'entrée et/ou du matériel. Les compilateurs devraient fournir un moyen au programme de s'adapter par le biais de la compilation dynamique (JIT) ou du multiversioning. La compilation dynamique a l'inconvénient d'introduire un surcoût, potentiellement non négligeable. Si l'objectif de l'utilisateur est uniquement la performance, le multiversioning est mieux adapté. Les versions de code, toutes sémantiquement équivalentes, mais de caractéristiques de performance différentes, sont choisies en fonction du contexte d'exécution. Ainsi, nous

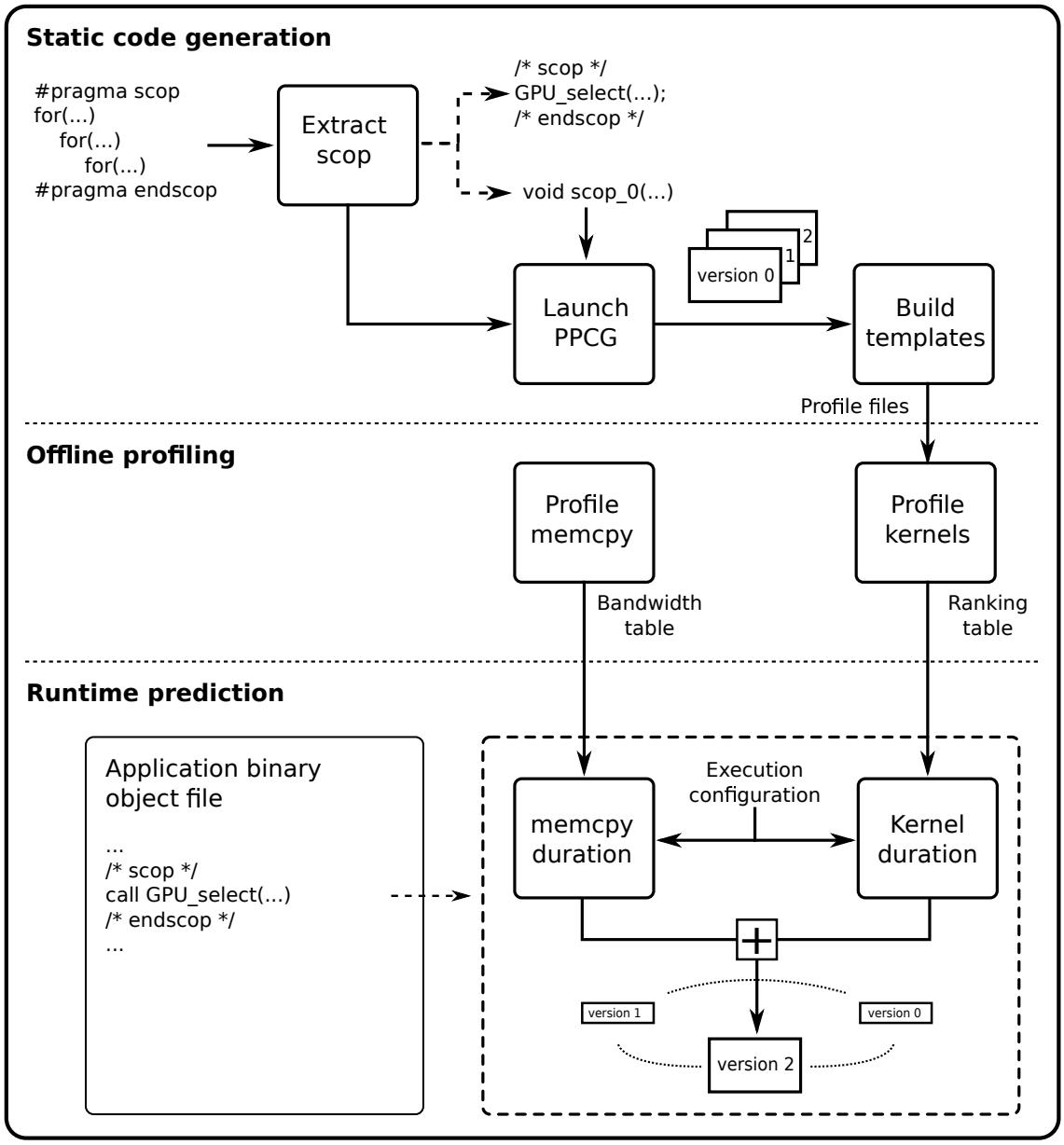


FIGURE 1.1 – Vue d'ensemble de l'infrastructure du framework.

avons développé un *framework* capable de prédire des temps d'exécution et de sélectionner la meilleure version d'un code sur une architecture hétérogène CPU+GPU.

1.2.2 Vue d'ensemble du framework

Le *framework*, représenté en Fig. 1.1, repose sur trois étapes : génération de code, profilage et prédiction. Les nids de boucles annotés dans le code source original sont extraits dans des fonctions. Plusieurs versions de ces nids de boucles sont générées à l'aide de PPCG, un compilateur polyédrique source-à-source capable de produire du code CUDA optimisé à partir de code C. Elles diffèrent par leur ordonnancement, taille de tuile et de bloc CUDA, utilisation de mémoire partagée, etc. PPCG a été

modifié afin de produire certaines informations issues de la compilation, notamment les polynômes d'Ehrhart représentant la taille du domaine d'itération et de la grille CUDA. Les fichiers générés sont transmis à un ensemble de programmes afin de construire le code de profilage et de prédiction.

Le profilage effectué avant toute exécution de l'application considérée, à l'installation par exemple, est constituée de deux phases distinctes pour traiter le cas des GPUs. Une première étape consiste à caractériser la bande passante disponible entre la mémoire centrale et la mémoire du périphérique et produit une *table de bande passante*. La seconde étape consiste à simuler l'exécution des différentes versions de code sur l'architecture cible. Le code de profilage dédié se charge de construire une *table de classement* par version de code. Ces informations sont utilisées à l'exécution afin de calculer une prédiction du temps d'exécution du code considéré avec un faible surcoût.

Dans la taxonomie CUDA, une fonction C exécutée par un ensemble de threads CUDA est communément appelée un *kernel*. Le *code hôte* réalise les appels *kernels* avec leur configuration d'exécution et s'occupe des copies mémoire. L'analyse du *code hôte* et des *kernels* générés par PPCG repose sur *pycparser*, un parseur de code C99 écrit en python, étendu pour supporter CUDA et un sous-ensemble de directives *pragmas*. Ces directives sont utilisées pour fournir des informations complémentaires au compilateur, comme les sections de code à optimiser ou à exécuter en parallèle. Les interactions avec le parseur sont contrôlées par un *wrapper* exposant les nœuds de l'Arbre de Syntaxe Abstraite (AST), une représentation en arbre d'un code, au travers d'une interface haut-niveau.

1.2.3 Equité

Pour concevoir notre profileur nous avons contrôlé empiriquement la propriété de commutativité des dimensions de la grille CUDA pour les codes. D'après les résultats obtenus, le nombre de blocs par dimension $sz_{x,y}$, n'avait pas une grande influence sur les performances $perf$:

$$perf(sz_x * sz_y) \sim perf(sz_{x\prime=y} * sz_{y\prime=x})$$

Nous distinguons les *paramètres séquentiels*, apparaissant dans les boucles séquentielles à l'intérieur et à l'extérieur des kernels des *paramètres parallèles* apparaissant dans les dimensions parallèles. Les paramètres parallèles déterminent la taille de la grille CUDA. Dans le profileur, pour maintenir l'homogénéité, les paramètres parallèles sont calculés pour correspondre au nombre de blocs requis, par dimension. La valeur des paramètres séquentiels est augmentée séparément, jusqu'à ce que leur impact devienne stable. Leur influence est modélisée par le biais d'une fonction linéaire de régression. Nous avons également remarqué que des tuiles incomplètes n'avait que peu d'incidence sur les temps d'exécution.

1.2.4 Prédiction

Pour éviter une dépendance vis-à-vis des particularités bas-niveau du matériel qui risquent de devenir obsolètes avec l'évolution du matériel, le framework développé manipule des temps d'exécution. Les expérimentations ont montré que les performances

des codes étaient principalement déterminées par le nombre de threads. Lors d'une phase de profilage, avant la première exécution de l'application, les versions sont évaluées sur la machine cible. Les principaux facteurs de performance affectant les temps d'exécution sont pris en compte par le système.

Les *facteurs de performance statiques*, sont constants lors de l'exécution et sont naturellement pris en compte pas le profilage. Ils comprennent le temps d'exécution de toutes les instructions : opérations arithmétiques et en virgule flottante, instructions de branchement (incluant la prédiction), etc. Les *facteurs de performance dynamiques externes* sont difficiles à contrôler car ils sont issus de l'environnement d'exécution. Leur fréquence et leur durée impactent la qualité des mesures effectuées et par conséquent influencent la précision des prédictions. Les *facteurs de performance dynamiques internes* émanent des interactions entre l'application cible et le matériel. Notre système doit caractériser ces facteurs de performances car ils fluctuent en fonction du problème en entrée : d'une exécution à une autre, le temps d'exécution peut différer.

Dans ce travail nous nous concentrons principalement sur des grandes tailles de problèmes qui représentent le cas d'utilisation typique des GPUs. Pour s'affranchir des effets de cache difficilement modélisables, du surcoût de l'appel kernel, et pour mettre en lumière l'influence de la coalescence et des conflits de bancs, la taille du domaine d'itération est manipulée artificiellement.

Des essais préliminaires ont mis en exergue deux importants facteurs de performance dynamiques internes, pour réaliser des prédictions précises. Le premier concerne les contentions d'accès mémoire et les conflits de banc. Comme les paramètres parallèles sont calculés pour correspondre au mieux au nombre de blocs requis, l'empreinte mémoire qu'ils induisent est encapsulée dans les mesures. Les paramètres séquentiels requièrent un traitement spécifique comme leur valeur peut varier pour une taille de grille donnée. Leur influence n'est pas nécessairement constante. Les expérimentations ont montré que leur incidence suivait une tendance linéaire pour les temps d'exécution par itération collectés. Leur influence est modélisée via une fonction linéaire de régression. Pour prendre cela en considération, la table de classement est remplie avec les fonctions linéaires des paramètres séquentiels.

L'ordonnanceur de blocs et de warps est capable de cacher les latences mémoire en couvrant les blocs et warps en attente d'un accès mémoire avec d'autres calculs. La répartion de la charge au niveau bloc est donc le second paramètre de performance dynamique à considérer. Pour ce faire, les mesures sont effectuées pour différentes tailles de grille. Les portions linéaires des mesures sont détectées afin de réduire l'espace de recherche pour limiter la durée du profilage. Pour chacune des mesures une entrée est placée dans la table de classement.

Le profileur néglige cependant le surcoût de l'appel du kernel CUDA et les effets de cache en manipulant artificiellement le domaine d'itération. Pour éviter d'effectuer des mesures redondantes, le code de profilage bénéficie de nombreuses optimisations : détection des phases affines, commutativité de la grille CUDA, etc. En sortie, il produit une *table de classement* contenant des temps d'exécution par itération pour chaque version.

Un programme de mesure se charge de calculer la bande passante disponible entre l'hôte et le périphérique. Les caractéristiques de celle-ci sont fortement liées au matériel

sous-jacent. Les spécifications de la carte mère et la capacité du bus de données a une forte influence sur le débit disponible. En sus, le débit d'un périphérique PCIe est asymétrique et non linéaire. En effet, le débit s'accroît et tend vers la bande passante asymptotique lorsque l'on augmente la taille des données à transférer. Des transferts de petites taille ont une efficacité bien moindre. Un micro-benchmark dédié se charge de collecter les mesures de bandes passantes pour différentes tailles de message et construit une table de bande passante par sens de transfert. Le débit est modélisé par une *table de bandes passantes* représentant des fonctions affines par morceaux.

Lors de la phase de prédiction, des nids de boucles simplifiés prédisent les temps d'exécution des versions à l'aide des informations de profilage. A cet instant, le contexte d'exécution est connu. Le code de prédiction est construit en remplaçant l'appel kernel et les copies mémoire par le calcul associé. En premier lieu, le code de prédiction récupère les fonctions des paramètres séquentiels pour l'intervalle de blocs comprenant la taille de la grille CUDA considérée. Ces fonctions sont instanciées avec la valeur des paramètres séquentiels. Les résultats sont interpolés linéairement pour calculer le temps d'exécution. Puis, on y ajoute le temps prédit nécessaire pour la transmission des données, basé sur la taille des messages et la direction des transferts. La meilleure des versions GPU est sélectionnée et exécutée.

1.2.5 Expérimentations

La première plateforme de test est composée de deux Asus GTX 590 connectée à une carte mère Asus P8P67-Pro et sont couplées à un core i7 2700k. La seconde plateforme de test contient une Nvidia GTX 680 connectée à une carte mère Asus P8P67-Deluxe et couplée à un core i7 2600. Les cartes Asus GTX 590 reposent sur une architecture Fermi, tandis que la GTX 680 repose sur une architecture Kepler. Les expérimentations sont effectuées sur les codes issus de PolyBench.

Les Figures 1.2 et 1.3 montrent la précision de notre système. La barre de gauche représente le temps d'exécution prédit tandis que la barre de droite indique le temps d'exécution effectif. Chaque barre est divisée en deux parties représentant le temps de communication et le temps d'exécution d'un kernel. Quelques imprécisions sont obtenues pour des petites tailles de problème, mais celles-ci sont équivalentes pour chaque version et n'influencent donc pas le mécanisme de sélection de la version de code la plus performante.

Sur GTX 590 l'erreur moyenne pour un jeu de données standard est de 2.3% et tombe à 1.13% en excluant syrk3. Sur GTX 680 l'erreur moyenne est de 5.8% pour le jeu de données standard. Ces résultats assurent un bon niveau de fiabilité quant à la version de code sélectionnée.

Selon l'architecture la meilleure version de code peut varier. Ainsi pour la première plateforme de test les codes : bicg1 *vs* bicg3, gemver3 *vs* gemver2, mvt1 *vs* mvt2, gesumv1 *vs* gesumv2, sont respectivement les plus rapides sur GTX 590 *vs* GTX 680. Au final, notre méthode est capable de sélectionner la version de code la plus rapide en fonction de l'architecture cible.

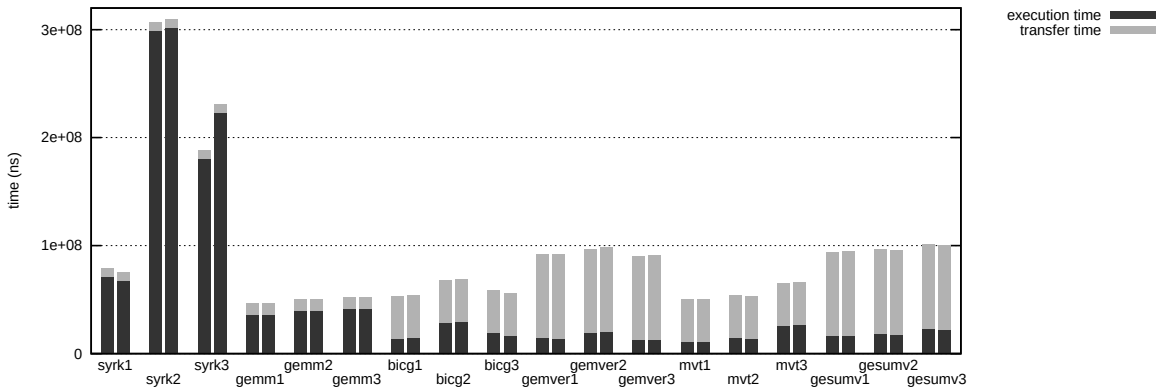


FIGURE 1.2 – Comparaison entre temps d'exécution prédit (barre gauche) et effectif (barre droite) sur PolyBench avec le jeu de données standard sur GTX 590.

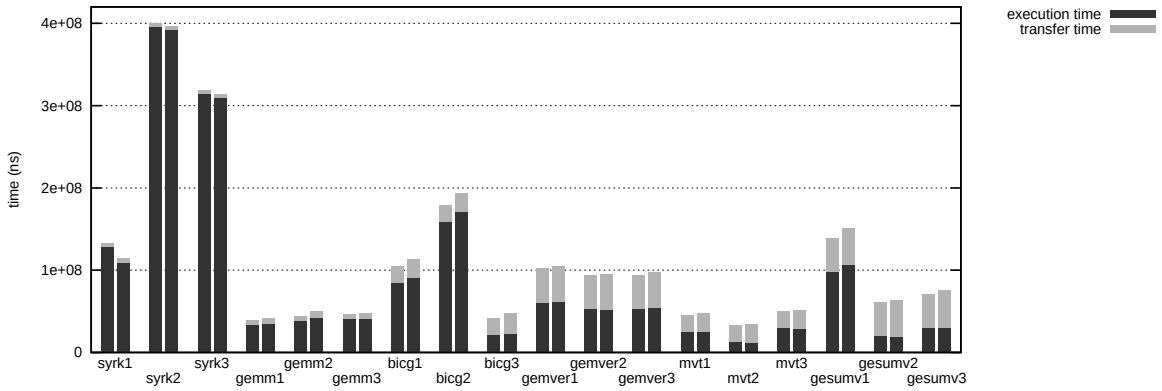


FIGURE 1.3 – Comparaison entre temps d'exécution prédit (barre gauche) et effectif (barre droite) sur PolyBench avec le jeu de données standard sur GTX 680.

1.3 Calcul hétérogène : CPU *vs* GPU

1.3.1 Introduction

Avec la profusion d'architectures alternatives, choisir le (co)processeur approprié est crucial pour assurer une exécution efficace d'un code. Choisir une architecture statiquement peut conduire à rater des opportunités de performances, sur une grille de serveurs acceptant plusieurs *jobs* utilisateurs par exemple. Dans ces circonstances, il est préférable d'exécuter un code sur un autre processeur pour conserver un bon niveau de performances. Pour satisfaire à ces contraintes, nous avons développé une méthode dynamique de sélection d'architecture. La technique proposée est capable de s'adapter aux contraintes dynamiques et selectionner la meilleure architecture pour le problème en entrée. Néanmoins, même si elle s'adapte bien à la dynamicité cela se fait au détriment de la consommation électrique. De ce fait, pour choisir l'architecture la plus à même d'exécuter un code et palier à l'influence de l'environnement d'exécution, un code GPU est mis en concurrence avec un code CPU. Le vainqueur signale sa terminaison et met prématurément fin à l'exécution des autres compétiteurs. Or, il n'existe pas de

moyen propre et efficace de terminer l'exécution de codes parallèles, ni avec OpenMP ni avec CUDA. Pour ce faire, les codes parallèles sont instrumentés afin de piloter la terminaison anticipée de l'exécution.

1.3.2 Code CPU

Sur CPU les boucles *for* parallèles (e.g. *omp parallel for*) sont extraites et sont placées dans des sections parallèles (e.g. *omp parallel*). L'implémentation OpenMP utilisée est basée sur les threads POSIX, autorisant la terminaison d'un thread par l'envoi de signaux et l'utilisation des fonctions de restauration de contexte (i.e. *long jump*). Ainsi, nous évitons l'utilisation d'un mécanisme d'attente active ayant plus d'impact sur les performances. En outre, cela a l'avantage de ne pas affecter les nids de boucles parallélisés sur le niveau de boucle le plus externe. Le code est généré de sorte que les itérations de la boucle parallèle soient distribuées selon l'ordonnancement statique introduit par OpenMP. Le gestionnaire de signaux effectue un saut à la fin de la section parallèle pour terminer l'exécution rapidement. Ainsi, les threads ne sont pas annulés, pour éviter les comportements non-définis, mais leur flot d'exécution est redirigé vers la fin de la section parallèle. Pour ce faire, les meta-données du threads (*e.g.* thread id, contexte) sont sauvegardées à l'entrée d'une section parallèle, pour une utilisation ultérieure. En outre, un *flag* est fixé afin d'indiquer une terminaison artificielle de l'exécution de la section parallèle. A nouveau, en dehors d'une section parallèle le flot d'exécution est redirigé vers la sortie de la fonction implémentant le code CPU.

Cependant, les threads sortis de la section parallèle, ne doivent pas y retourner auquel cas un saut pourrait corrompre la pile et avoir un comportement indéfini. De plus, selon le standard OpenMP un saut ne peut violer le critère d'entrée et de sortie des threads. Le code séquentiel est protégé par un verrou pour empêcher ce type de comportement. Dès qu'un thread de calcul entre ou sort d'une section parallèle, il désactive ou active le verrou qui lui est associé.

1.3.3 Code GPU

Sur GPU, une garde est insérée dans la deuxième dimension du nid de boucles parallèle. Un flag permet de commander l'interruption de l'exécution d'un kernel à l'aide de l'instruction *trap*. Ainsi, à chaque itération de la boucle instrumentée tous les threads actifs contrôlent un flag localisé dans la mémoire globale du périphérique. En effet, contrôler la valeur d'une variable allouée en mémoire centrale peut grever les performances, surtout si un nombre conséquent de threads tente d'y accéder simultanément. Par ailleurs, le flag scruté est déclaré volatile, afin d'empêcher le compilateur d'optimiser la variable dans un registre. Après la terminaison de l'exécution d'un kernel, un flag similaire est contrôlé afin de rediriger l'exécution vers la fin du code CUDA hôte.

Depuis CUDA 4.0, le contexte d'exécution associé à un périphérique est partagé entre les différents threads. Par conséquent, il est possible d'envoyer des commandes vers un même périphérique depuis plusieurs threads dans des *streams* différents, sans que les opérations ne soit traitées séquentiellement. A cette fin, le code généré par PPCG est transformé afin d'utiliser les appels fonction CUDA appropriés, afin que les

mouvements de données et les kernels soient placés dans la file d'un stream CUDA. Pour interrompre l'exécution d'un kernel, le CPU modifie la valeur du flag sur le GPU via une copie mémoire placée dans un stream différent. Un flag est également contrôlé après chaque mouvement de données, car ces derniers sont ininterruptibles.

1.3.4 Expérimentations

La Figure 1.4 montre des temps d'exécution normalisés par rapport à la version de code la plus rapide sur la plateforme de test comprenant deux GTX 590 et un core i7 2700k. On observe que pour le jeu de données standard covariance, *2mm*, *3mm*, *doitgen*, *gemm*, *syr2k*, *syrk*, *reg-detect*, *fdtd-2d*, *jacobi-1d-imper* and *jacobi-2d-imper* sont plus efficaces sur GPU. A l'opposé *atax*, *bicg*, *gemver*, *gesummv*, *mvt*, *trmm*, *lu*, *floyd-warshall* and *fdtd-apml* sont plus rapides sur CPU. Pour le jeu de données standard l'accélération maximale est de 10.46x pour *2mm*, tandis qu'elle est minimale pour *fdtd-2d* avec 1.32x. Lorsque le CPU gagne, l'accélération maximale est de 2.96x pour *floyd-warshall*, tandis qu'elle est minimale pour *bicg* avec 1.02x. En moyenne l'accélération est de 1.57x, et de 3.88x lorsque respectivement, le CPU ou le GPU gagne.

Le choix de l'architecture est réalisé dynamiquement grâce à ce système de sélection de code basé sur la compétition entre les ressources. Néanmoins, ceci se fait au détriment d'une consommation électrique plus importante. En revanche les facteurs de performances dynamiques externes sont pris en compte et permettent d'adapter le choix de l'architecture cible. En tout et pour tout, cette technique est capable de sélectionner la meilleure des versions de code avec un surcoût contenu.

1.4 Calcul hétérogène : CPU + GPU

1.4.1 Introduction

Une troisième partie de la thèse consiste en l'exploitation conjointe des CPUs et GPUs pour l'exécution d'un code. Une utilisation efficace des ressources de calcul hétérogènes est un problème difficile, particulièrement lorsque les unités de calcul (UC) reposent sur des environnements de compilation et d'exécution différents en plus des différences architecturales.

Sur des CPUs multicoeurs, une exécution efficace repose sur la parallélisation (avec OpenMP par exemple), l'exploitation de la localité de cache (tuilage de boucle par exemple), des optimisations bas niveau ; tandis que l'exploitation des GPUs requiert des optimisations des transferts mémoire entre hôte et périphérique, distributions des calculs sur une grille de blocs SIMD avec des limitations sur leur taille, une exploitation explicite de la hierarchie mémoire, la coalescence des accès mémoire, etc.

Dans ce travail, notre objectif est de résoudre ces problèmes automatiquement, en l'occurence par la génération de codes efficaces exécutés sur plusieurs UCs dans un contexte dynamique. La plus grosse difficulté de ce problème est de calculer une bonne répartition de charge entre les UCs hétérogènes. Pour ce faire, nous nous reposons sur des prédictions de temps d'exécution sur chaque PU, basé sur un générateur de code statique, un profilage offline et une prédiction et un ordonnancement à l'exécution.

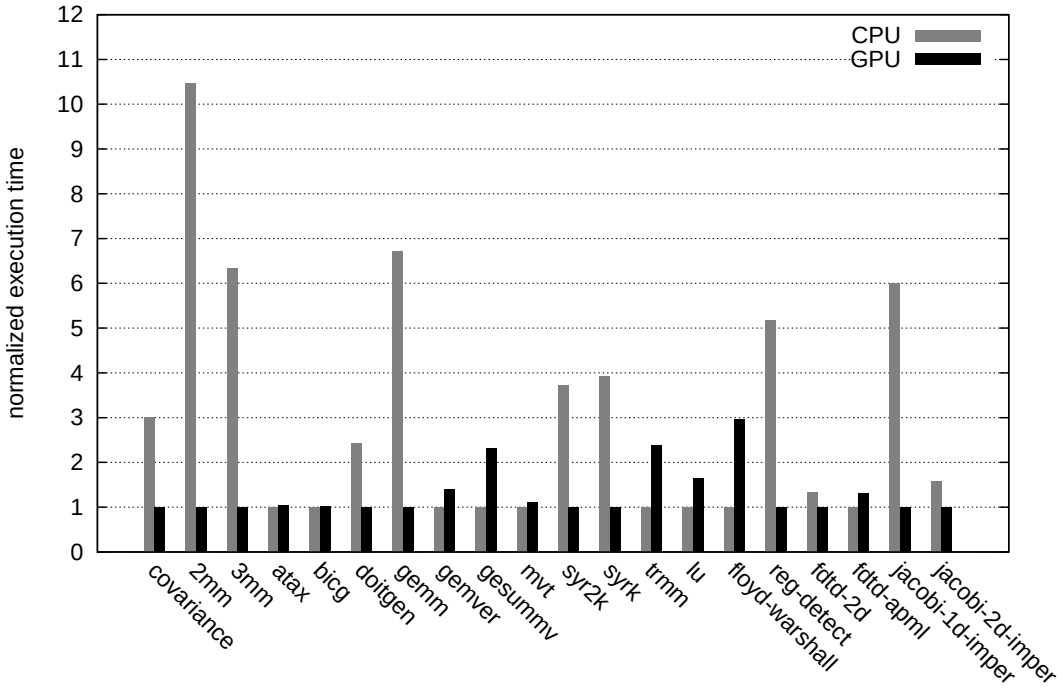


FIGURE 1.4 – Temps d'exécution normalisé pour le jeu de données standard sur PolyBench

Notre plateforme de développement cible des processeurs à mémoire partagée et un ou plusieurs GPUs CUDA.

1.4.2 Génération de code

Lors d'une étape préliminaire, le code est parallélisé avec le *backend* OpenMP C de PPCG. Des bornes de boucles artificielles sont injectées dans les boucles parallèles pour contrôler le domaine d'itération. Cela permet de diviser le domaine d'itération initial des boucles parallèles en morceaux, appelés *chunks*. A l'exécution, chaque chunk est associé à une UC et dimensionné pour assurer une bonne répartition de la charge. Pour ce faire, PLUTO et PPCG génèrent des versions de code spécialisées des chunks parallèles, optimisées pour le CPU et le GPU. Comme les chunks peuvent être exécutés dans n'importe quel ordre la sémantique du code est préservée et les calculs peuvent être distribués sur les UCs disponibles sans risque. Les scripts développés calculent également la boîte englobante des accès tableau pour générer le moins possible de communications entre CPU et GPU.

Pour des codes parallèles appelés itérativement, la quantité de données à transférer est ajustée pour éviter les copies redondantes. Plus précisément, les scripts déterminent quels éléments de tableau ont été modifiés en dehors du code parallèle et nécessitent d'être transférés vers le GPU. Pour les transferts restants, seules les données manquantes, après re-partitionnement, sont envoyées vers le GPU en contrôlant les paramètres de partitionement. Enfin, les résultats calculés sur GPU sont systématiquement

transférés dans la mémoire du CPU.

1.4.3 Répartition de charge

La distribution des calculs est réalisée grâce à l'analyse de dépendences et l'ordonnancement polyédriques : des compilateurs comme PLUTO ou PPCG prennent un SCoP en entrée et génèrent des nids de boucles parallèles et séquentielles. Les boucles parallèles les plus externes sont découpées en chunks, dont la taille est contrôlable et sont exécutés indépendemment sur plusieurs UCs.

Afin de préserver les performances, il est nécessaire de considérer la répartition de charge, notamment en prédisant les temps d'exécution. En effet, nous considérons que le temps d'exécution minimum d'un calcul distribué est obtenu lorsque les temps d'exécution sont parfaitement équitables entre les UCs. Le temps d'exécution d'un chunk sur une UC donnée est prédit à l'exécution, à chaque exécution du code cible. Le calcul est basé sur : (1) la taille de la boucle (*i.e.* le nombre d'itérations et les données accédées), évalués avec des outils polyédriques ; (2) le temps d'exécution par itération et par donnée accédée moyen, basés sur les tables générées automatiquement au profilage et dépendant du contexte d'exécution (taille de la grille CUDA, répartition de charge entre les coeurs CPUs). Au final, la répartition de charge entre les différentes UCs est obtenue en ajustant la taille des chunks pour distribuer les temps d'exécution équitablement.

Notre système de prédiction des temps d'exécution sur GPU est complété par le travail de Benoit Pradelle *et al.* [120] pour prédire les temps d'exécution sur CPU multi-cœurs. Les codes CPU sont générés avec PLUTO, un compilateur polyédrique source-à-source générant un code optimisé pour la localité, parallélisé avec OpenMP et tuilé. Le *runtime* consiste en deux composants : un ordonnanceur et un répartiteur, et fait usage des résultats du profilage pour prédire des temps d'exécution. La boucle parallèle la plus externe est décomposée en chunks, chacun associé à une UC spécifique. La technique d'ordonnancement est décrite par l'algorithme 1.1.

Le temps d'exécution par itération de chaque chunk fluctue non-linéairement en fonction de la taille du chunk comme nous l'avons décrit dans la Section 1.2. Par conséquent il n'existe pas de méthode directe de calcul de la taille des chunks, mais ce problème d'optimisation requiert d'être raffiné itérativement.

La taille des chunks est ajustée de manière à ce que les temps d'exécution soient distribués équitablement entre les UCs. Pour arriver au résultat escompté trois étapes sont nécessaires : l'*initialisation*, le *rafinage* et le *partitionement*. La phase d'initialisation distribue les itérations de la boucle chunkée équitablement entre les PUs. L'idée est de tendre au maximum vers une répartition équivalente des temps d'exécution, c'est à dire un temps d'exécution approximativement équivalent pour tous les chunks $T_0 \approx T_1 \approx ... \approx T_{n-1}$. Le rafinage débute par le calcul d'une prédiction du temps d'exécution de chaque chunk T_i et leur somme T_{all}. Ainsi, la proportion du temps d'exécution de chaque chunk $R_i = T_i/T_{all}$ doit tendre vers $o = 1/n$ pour obtenir une bonne répartition de charge. Notez que la répartition de charge optimale prédite est obtenue pour $R_i = o$ pour tous les i. La taille de chaque chunk est ajustée en la multipliant par o/R_i pour se rapprocher de la répartition de charge optimale. Cependant ces ajustements

Algorithm 1.1 Scheduler algorithm

```
#step 1 : initialize to equal distribution
chnk_size ← (ub − lb)/num_pu
for i ← 0 to num_PU − 1 do
    PUs[i].lb ← i ∗ chnk_size
    PUs[i].ub ← PUs[i].lb + chnk_size
end for

#step 2 : refine
for s ← 0 to MAX_STEPS do
    time ← 0.
    for i ← 0 to num_PU − 1 do
        PUs[i].size = PUs[i].ub − PUs[i].lb
        if PUs[i].size ≠ 0 then
            PUs[i].time_val = PUs[i].time(PUs[i].lb, PUs[i].ub)
            time ← time + PUs[i].time_val
        end if
    end for
    for i ← 0 to num_PU − 1 do
        if PUs[i].time_val ≠ 0 then
            adjst = time/(num_PU ∗ PUs[i].time_val)
            PUs[i].size = PUs[i].size ∗ adjst
        end if
    end for

    #normalize the chunk bounds
    (PUs, max_card) ← normalize(PUs)
    for i ← 0 to num_PU − 1 do
        if PU.cardPU/max_card < 0.1 then
            PUs ← eliminate(PUs, i)
        end if
    end for
    reg ← register_schedule(PUs)
end for

#step 3 : select schedule
PUs ← select_schedule(reg)
```

sont calculés indépendamment pour chaque chunk et cela conduit à des situations où la somme de la taille des chunks n'est plus égale au nombre total d'itérations. Ainsi, la phase de partitionnement normalise la taille des chunks de telle sorte que toutes les itérations de la boucle chunkée soit exécutées, ni plus ni moins. Les itérations éliminées par les arrondis d'entiers sont assignées à une UC arbitraire (le CPU par défaut dans l'implémentation courante). Les UCs inefficaces sont retirées en se basant sur le nombre d'itérations des chunks et sont réutilisables pour d'autres calculs.

Après chaque redimensionnement, les temps d'exécution sont recalculés jusqu'à obtenir stabilisation, dans la limite de 15 itérations. Puis, le répartiteur lance l'exécution des chunks sur leurs UCs respectives. Afin de ne transmettre que les données utiles, la boite englobante des accès mémoire est calculée. Cette méthode peut être associée à un algorithme qui sélectionne les UCs respectant au mieux les contraintes énergétiques et de performance imposées.

1.4.4 Multiversioning

Notre framework est capable de générer plusieurs versions de codes CPU et GPU et de sélectionner la combinaison de versions la plus performante à l'exécution. L'ordonnanceur est appelé pour chacune des combinaisons et retourne le temps d'exécution prédit. Le système sélectionne la combinaison ordonnancée qui minimise le temps d'exécution. Comme le nombre de combinaisons croît exponentiellement nous avons limité le nombre de versions à 3 par UC (9 combinaisons pour une plateforme à base de CPU + GPU).

1.4.5 Consommation d'énergie

Le thème de la consommation énergétique est un des sujets brûlants en recherche sur les systèmes informatiques en raison des problèmes de quantité d'énergie requise, de budget et écologiques. L'efficacité énergétique, est généralement fortement corrélée aux performances d'un programme. Plus un code est rapide, plus faible est la consommation en énergie, qu'importe l'architecture sous-jacente.

Dans un contexte hétérogène, les ordonnanceurs peuvent favoriser des architectures efficaces, quand la consommation en énergie est un problème. Dans ce cas, les UCs exposant un ratio calcul/consommation énergétique acceptable font partie du calcul.

Pour cette raison nous introduisons une nouvelle unité : les watts par itération. Dans cette relation le temps est laissé de coté, comme le rôle de l'ordonnanceur est de dimensionner les chunks de telle façon que leurs temps d'exécution soient similaires. Plus le nombre d'itérations allouées au chunk est grand plus l'UC est considérée efficace. Cette technique est suffisante pour éliminer les UCs énergétiquement inefficaces.

1.4.6 Experimentations

La Figure 1.5 montre l'accélération obtenue en utilisant différentes combinaisons d'UCs comparé aux temps d'exécution sur CPU seul ou sur GPU seul. Notre système atteint une accélération maximale de 20x sur *gemm* et une accélération de 7x en moyenne en comparant le meilleur et le moins bon temps d'exécution. Ces résultats montrent

que *gemm*, *2mm*, *3mm*, *syrk*, *syr2k* (les cinq sur la gauche de la Fig. 1.5) sont mieux adaptés à une exécution sur GPU tandis que *doitgen*, *gesummv*, *mvt*, *gemver* sont plus performants sur CPU. Notez que doitgen est plus performant sur le CPU en raison d'un temps de calcul moins élevé sur CPU que sur GPU, et non pas à cause du temps de transfert des données. Une exécution combinée sur CPU+GPU profite de façon remarquable à trois codes (*syr2k*, *doitgen* et *gemver*). Grâce à ce système d'ordonnancement nous sommes capables d'exécuter conjointement un code sur des systèmes comprenant un CPU et plusieurs GPU. Les accélérations dénotent de la précision des prédiction et de la qualité du résultat obtenu par notre ordonnanceur.

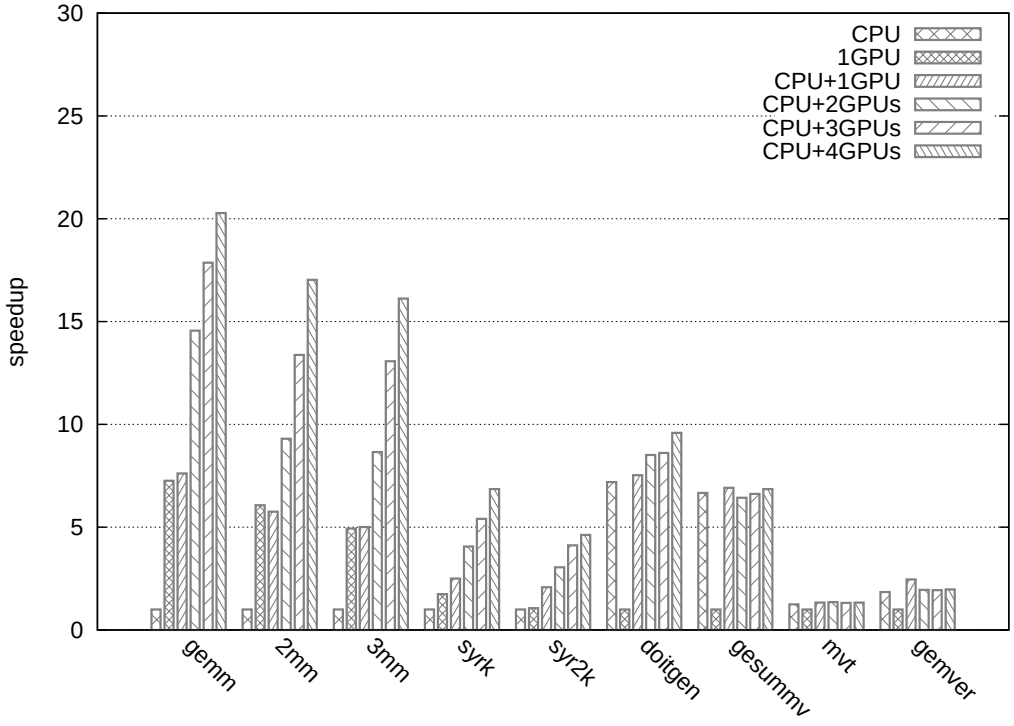


FIGURE 1.5 – Speedup to execution time on CPU or GPU alone.

1.5 Parallélisation spéculative

1.5.1 Introduction

Durant ma thèse j'ai participé au projet VMAD (Virtual Machine for Advanced and Dynamic analysis of programs), un système de parallélisation spéculative développé au sein de l'équipe INRIA CAMUS. Le *framework* est constitué d'un composant statique implémenté sous forme de passes LLVM et d'une machine virtuelle. Plusieurs versions d'un nid de boucles, qui diffèrent par les instrumentations appliquées, sont générées statiquement. Celles-ci communiquent avec la machine virtuelle via des *callbacks*. Le passage d'une version à une autre, décidé par la machine virtuelle, est assuré de façon transparente pour l'application. La machine virtuelle se présente sous la forme d'une

bibliothèque partagée greffée à l'application lors de l'exécution (préchargement). Un ensemble de plug-ins chargés à la demande implémentent une définition des *callbacks*.

1.5.2 Problématique

La plupart des compilateurs traditionnels restent conservatifs en cas d'incapacité à garantir la validité sémantique d'une transformation de programme statiquement. La parallélisation spéculative est une approche optimiste faisant l'hypothèse qu'un code est au moins partiellement parallèle. Cette technique requiert l'emploi de code de vérification pour détecter une éventuelle violation des dépendances à l'exécution. Les boucles *while*, les nids de boucles partiellement parallèles ou faisant intervenir des pointeurs, constituent la classe de codes candidats à la parallélisation automatique spéculative.

1.5.3 Machine virtuelle

A l'exécution, un échantillon du domaine d'itération est considéré, afin de construire les fonctions affines des accès mémoire à l'aide d'itérateurs virtuels. J'y ai adjoint un algorithme de calcul des bornes de boucles. Une analyse de dépendance dynamique extrait le parallélisme des nids de boucles considérés. Selon la dimension de boucle parallèle, l'exécution est redirigée vers le *motif* parallèle adéquat. Le motif contient le nid de boucles original parallélisé et transformé en boucles *for*. L'exécution des nids de boucles cibles est découpée en intervalles d'itérations contigus de la boucle externe, appelés *chunks*. Un module de décision implémente un automate chargé de sélectionner le type de code à exécuter dans un *chunk*. Les *chunks* d'instrumentation se chargent notamment de récupérer les adresses mémoire accédées et les valeurs des itérateurs virtuels. Après cette étape et selon le résultat de l'analyseur de dépendance dynamique, un *chunk* parallèle (code parallèle) ou original (code séquentiel original) peut être lancé. Si un *chunk* parallèle s'est avéré erroné, son exécution est rejouée par un *chunk* original, suivie par une phase d'instrumentation.

1.5.4 Contributions

Des instructions de vérification se chargent de tester la validité des *chunks* lors de l'exécution. J'ai intégré un système de commit/rollback équivalent aux mécanismes employés pour les bases de données. Avant l'exécution d'un *chunk* parallèle, les données sont sauvegardées pour les plages d'accès mémoire en écriture prédites. Pour optimiser les copies en mémoire, une stratégie de fusion des plages d'accès mémoire, dépendant de leur distance en octets, est préalablement appliquée. En cas de rollback, les données antérieurement sauvegardées sont propagées et écrasent toutes les éventuelles modifications du dernier *chunk*. La génération de code est réalisée statiquement, le code est générique et complété à l'exécution par une étape d'instantiation que j'ai développée. Celle-ci requiert les fonctions affines issues de la phase d'instrumentation.

Chapter 2

Introduction

2.1 Context

Diversity and complexity of programs have continually urged the microprocessor industry to enhance computing speed. Before the early 2000s, software and hardware communities lived a smooth story. During 30 years, performance of the processors grew exponentially, by a factor of 30x in average, every decade [60]. In that world, software could directly benefit from hardware technological improvements, barely requiring any code refactoring. Most of the performance gains stemmed from processor higher clock rates and transistor count [53], materialized in the following relation:

$$Performance = Frequency * IPC^{1}$$

While frequency scaling fastens the raw execution process, electronic miniaturization allows the processors to implement a series of optimization dedicated circuitry, such as, wider and hierarchized caches, out of order unit, memory prefetcher, branch predictor, etc. In addition, the emerged ILP (Instruction Level Parallelism) and DLP (Data Level Parallelism) paradigms, in conjunction with redundant or larger functional units, paved the way to superscalar and vector-processing architectures.

Nowadays, the semiconductor industry faces numerous technological locks. The *power wall*, arising from energy constraints, in particular, power consumption and heat production, has drastically limited frequency scaling potential. Also electronic components shrinking has emphasized tunnelling effects, leading to more energy draining, as a result to current leakage. In addition to physical barriers, the *ILP wall* (limitations inherent to exploiting instruction parallelism) and the *memory wall* (the gap between processor and memory speed), considerably limit hardware performance improvements.

In his 1970s projection, Moore highlighted that transistor density would roughly double every two years. In modern processors, this conjecture materializes, inter alia, by embedding multiple cores in a single chip. Dennard observed that power was a function of the surface of the circuits rather than density: "Moore's law gives us more transistors... Dennard scaling makes them useful". This watershed in the design of processors, marked the end of the *Free Lunch* [137] in the software field. To fully exploit

[1] IPC: Instruction Per Cycle

this new godsend, programmers have not only to rethink their applications, but also their way of writing, ultimately, efficient programs. However, according to observation on CPU specifications, the number of cores will likely remain stable and intra-core performance may even drop in the next few years [96]. Indeed, the performance trends inverted to the point where *"software is getting slower more rapidly than hardware becomes faster"*, as stated in *Wirth's law* [153]. An emerging concept, relying on a good usage of accelerating cards, such as GPUs, FPGAs, Xeon Phi and other DSPs sets a new standard to achieve performance: heterogeneous computing was born, not to mention the plight of programmers.

2.2 Problematic

It is commonly accepted, especially in scientific computing, that most computation time of the programs is spent in loop nests or recursive calls. Regularity and syntactic construction of *for*-loops ease their optimisation. The polytope model provides loop nest analysis and manipulation facilities. Affine loop nests are abstracted into geometrical objects, namely bounded polyhedra. Loop bounds and array subscripts are parametric affine functions of the enclosing loop iterators. Under these conditions, application of code transformations becomes an exact ILP (Integer Linear Programming) mathematical problem.

To push the compilers in generating efficient code, it becomes common, if not mandatory, to parallelize and apply high-level transformations. An approach, called *approximate computing*, consists in granting some of the hardware or software reliability, to improve performance and energy consumption [110]. Such a technique is out of the scope of this thesis, as optimizations should be neutral to the program result. Parallelizing a code is a tedious and error-prone task, requiring, when appropriate, anti-dependence removal (*e.g.* variable- renaming, privatization, scalar expansion) and the application of transformations (*e.g.* loop- shifting, splitting, skewing), to expose parallelism. Although the codes are parallel, improving performance often requires to take advantage of data locality (*e.g.* loop- tiling, interchange, fusion) and maximize throughput (*e.g.* loop- unrolling, strip-mining). In particular, hardware limitations of GPUs, including the number of registers, size of shared memory, number of blocks, size of the blocks, and separated address space for discrete GPUs set the limits of code optimization.

To preserve the original semantics, it is compulsory to respect the initial code dependences. Finding the optimal combination of transformations is tricky due to the large number of optimizations and dynamic context sensitivity. Dynamicity arises for two main reasons: the execution environment variations (*e.g.* hardware characteristics and availability, compiler optimizations) and input data size variation (*e.g.* from a call to a function to another). On the other hand, compilers have to take static decisions to generate the best possible performing code on average. But, as a result of the dynamic context, they miss many optimization opportunities.

To relieve programmers, these problems suggest the use of assisted programming or automatic optimizing tools. High-level optimization and automatic parallelization are still a hot-topic in current compilers. Initiatives such as Graphite [114] and Polly [58]

have brought polyhedral optimizers in the mainstream GCC [52] and LLVM [89] compilers. High-level languages are natural candidates to polyhedral compilers as the control flow is unaltered. Ad-hoc source-to-source tools, such as PLUTO [25], C-to-CUDA [15], PPCG [145], PoCC [13], PIPS [8] transcript the codes into a target-optimized counterpart. Internal performance models are generally parametrized by heuristics. Tweaking is therefore delegated to the programmer or external tuning frameworks.

Since the scope of possible static optimizations to gain performance is limited, new techniques are required to get the most of modern processors. Context sensitive optimization is a difficult topic, hardware evolution and dynamicity call genericity. Feedback-oriented techniques refine the optimizer passes post-mortem, based on program execution traces and profiling. For this purpose, the compiler may identify and instrument hotspots [6, 20, 58, 146], that is, code regions requiring further optimizations. Other methods use static information, such as processor specifications or programmer hints, to parametrize their performance models. The concept of multiversioning relies on multiple potentially efficient versions of codes, generated statically. At runtime, control flow is routed to the supposedly fastest code version. Execution behaviour can be characterized before or during application execution. A discriminant metric (*e.g.* time) enables the code selector to choose the fastest version, based on pre-gathered profiling data or partial code executions at runtime.

Programmability and scalability of accelerator cards offer new speed-up perspectives. Following on from OpenMP [36], directive languages, such as HMPP [41], OpenACC [152], OmpSs [113] now target heterogeneous (co)processors. Code regions of interest are annotated and may be tagged with execution configuration and transformation hints. Efficient exploitation of heterogeneous computing resources is a difficult problem, especially when the processing units (PUs) run different compilation and runtime environments, in addition to different hardware. On multicore CPUs, efficient computing relies on parallelization (using OpenMP for example), cache locality exploitation (loop tiling for example), low-level optimizations (vectorization, instruction reordering, etc.). While exploitation of GPUs requires optimization of memory transfers between host and device, distribution of the computations on a grid of SIMD blocks with limitations on their sizes, explicit memory hierarchy exploitation, global memory accesses coalescing, etc. To benefit from heterogeneous environments it is compulsory to accurately distribute the load. Task-based systems dispatch the computations according to hardware affinity and availability. Tasks, typically structured by a graph, are picked from a pool with respect to dependencies. History-based methods construct an affinity database between tasks and processors for a given input problem. The most suited architecture is selected before re-executing the code. Fine-grained techniques slice loop iteration domains and balance the load. Computation may be run jointly on multiple heterogeneous processors. To enhance effectiveness, the schedulers are parametrized by energy constraints.

The majority of traditional compilers stay conservative when it comes down to parallelization. In fact, compilers take conservative decisions when they do not have enough information to guarantee the correctness of the codes statically. Speculative parallelism is an optimistic approach, making the assumption that a code at least exposes partial parallelism. A Thread Level Speculation (TLS) framework, aims to

execute the target code with multiple threads generally backed-up by a commit/rollback system, either supported by software or hardware. Most of the TLS rely on profiling, execution sample of the target code is analysed at runtime. Based on the collected information, the code may be parallelized. The more advanced tools are capable to perform code transformations and dynamic optimizations at runtime. For instance, tile size may be decided through multiple empirical tests at runtime. Also, such a system may decide whenever it is best to run a code on CPU, GPU or both.

The backbone contributions of this thesis are three techniques to tackle GPU and CPU performance:

- A code selection mechanism built on multiversioning, based on predicted execution times. Different versions are statically generated by PPCG [145], a source-to-source polyhedral compiler, able to generate CUDA code from static control loops written in C. The code versions differ by their block sizes, tiling and parallel schedule. The profiling code carries out the required measurements on the target machine: throughput between host and device memory, and execution time of the kernels with various parameters. At runtime, we rely on those results to calculate predicted execution times on GPU, and select the best version.

- A method to jointly use CPU and GPU in order to execute a balanced parallel code, automatically generated using polyhedral tools. To evenly distribute the load, the system is guided by predictions of loop nest execution times. Static and dynamic performance factors are modelled by two automatic and portable frameworks targeting CPUs and CUDA GPUs. There are multiple versions of the loop nests, so that our scheduler balances the load of multiple combinations of code versions and selects the fastest before execution.

- A CPU vs GPU code selection mechanism capable to adapt to dynamicity, by running the codes simultaneously.

Additionally, we provide insights of the backup and dynamic code generation mechanisms of VMAD. We elaborate on potential issues to port the current speculative mechanism to GPUs. For this purpose we provide a route, to enable speculative computation on GPUs.

2.3 Outline

To synthesize, the thesis is structured in the following way. In Chapter 3, we present the polyhedral model and describe the context of this thesis. In Chapter 4 we provide a description of our mechanism to predict execution times on GPU and the subsequent code selection mechanism. In Chapter 5 we thoroughly describe a dynamic CPU vs GPU and an hybrid CPU + GPU mechanism to tackle heterogeneous systems. We briefly go through an hybrid CPU vs GPU technique and focus on a strategy to handle energy constraints applied to the CPU + GPU technique. Finally, in Chapter 6 we present VMAD, a software speculative parallelism system designed by Alexandra Jimborean a former Ph.D. student of the INRIA CAMUS group.

Chapter 3

Context and related work

Recent history has revived interest for coprocessors and accelerating devices. Arithmetic coprocessors were initially dedicated to quickly dispatch floating point operations. In fact, in the early days of modern CPUs, floating point operations were emulated. As circuitry evolved, arithmetic coprocessors rather fell into disuse. In the recent years, inability of CPUs to scale in performance revived the interest in specialized coprocessors. Indeed, the number of general purpose high performance computing architectures increased drastically: AMD and Nvidia GPUs, Intel MIC, Kalray MPPA, FPGAs, etc. This microcosm of heterogeneous architectures raises the question of efficiency *vs* development time and integration in a computing environment [151].

Nevertheless, introduction of GPUs in the ecosystem of processors has tremendously impacted the typical usage of CPUs. Arora et al. [10] show that porting GPU-friendly code sections leaves a code of different characteristics for the CPU, typically harder to characterize with the state of the art prefetchers, branch predictions and exposing less ILP. Similarly to natural species, the authors of this paper suggest specialization to ensure efficiency. Thus, to address the new challenges CPUs must cope with the new situation. If this trend turns true, heterogeneity will reach new levels.

Graphics processors broke through thanks to inherently parallel multimedia applications. GPUs come for cheap and are widely available: in mobile devices (smartphones, tablets, embedded systems), as parts of a System On a Chip (SoC) (Snapdragon, Tegra, Exynos) or Accelerated Processing Units (APU) (AMD FX, Intel core i7), game consoles, laptop and desktop computers and now, supercomputers. Today, they also turned out to be a viable alternative to achieve performance with parallel programs. Henceforth, it is not unusual to list GPUs powered supercomputers in the *top500* and *green500* rankings [4, 3]. The toolchain to tackle GPUs has reached a certain level of maturity, either in programming models, debugging tools, number of exposed features or bug correction.

This performance potential does not come for free as it requires software adaptation. Nevertheless, porting a code to the GPU does not always translate to better performance as it may as well lead to slowdown. Modern compilers must work on several fronts: discover the best optimization combination for an architecture, find good parameters for parametric optimization, and potentially find the best architecture for a code in collaboration with runtime systems. In fact, there was a time, a compiler was

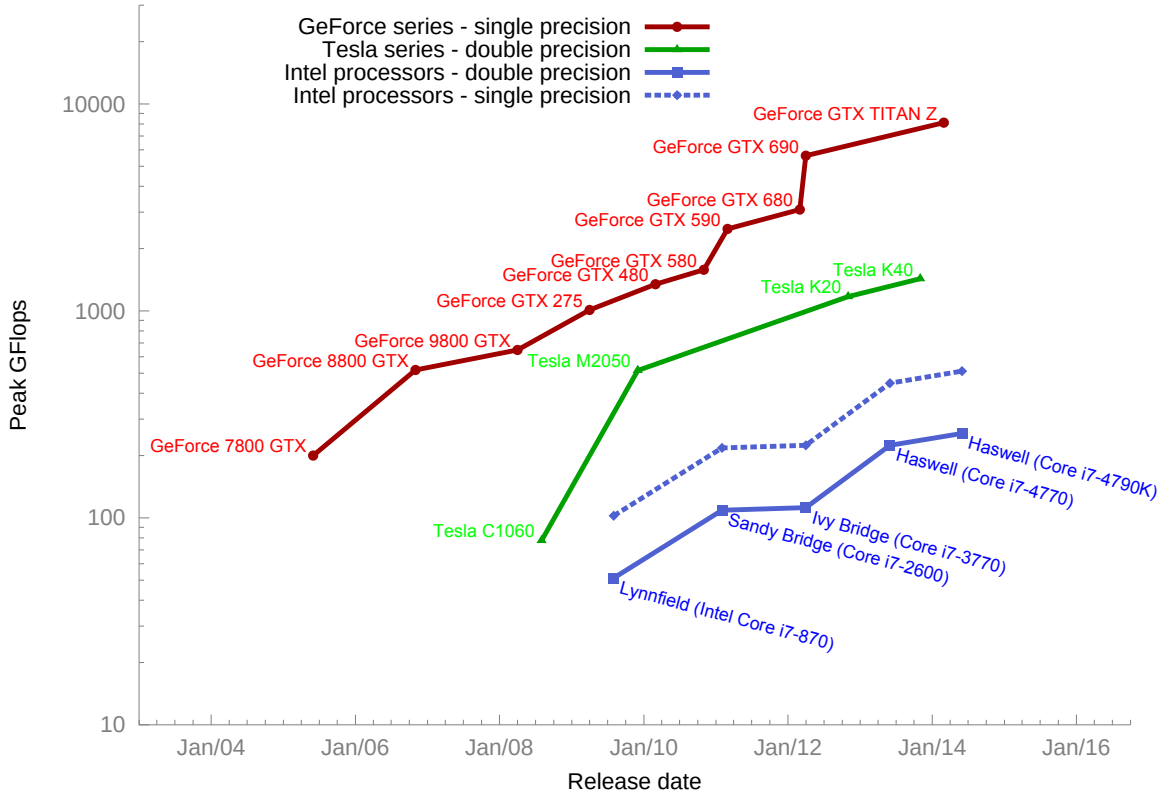


Figure 3.1 – Recomputed theoretical peak performance trends between Intel CPUs and Nvidia GPUs.

solely asked to find some good, if not the best, optimizations for a given architecture. Already this proved to be a difficult matter, mainly due to the lack of information, language ambiguity and semantic restrictions. Ideally, modern compilers should provide architecture optimized codes and provide support to an execution on an heterogeneous system. However, adding flexibility to a compiler inevitably complexifies decision making. For GPUs, involved performance models are by nature more complicated as they must take communication time and specifics of throughput oriented architectures into account. To relieve compilers, we propose a runtime system that takes the appropriate decisions in order to execute a code following a criterion, for instance performance or energy, during execution.

In this chapter we present the context and the various tools which relate to this work.

3.1 Genesis of GPU computing

The rapid adoption of computers in businesses and homes in the late 80s, encouraged development of intuitive and user-friendly graphical Operating Systems (OS). Several OSes, such as Mac OS, Windows, AmigaOS and Unix-alikes, proposed their own Graphical User Interface (GUI). Already, the idea to kick the display computa-

tions from the CPU caught on. The first graphics accelerators were primarily output peripherals, endowed with rather limited functionalities such as rasterization of polygons preprocessed on the CPU. The advent of 3D First Person Shooters (FPS) games, such as Duke Nukem 3D and Doom, challenged hardware improvement. Flexibility advances allowed to offload more and more of the graphics pipeline stages, *i.e.* the set of operations resulting in graphics objects display, to accelerators. Initially, the graphics pipeline was consisting in a series of configurable fixed functions stream. The introduction of *shaders*, small functions intended to be executed on a GPU, in DirectX 8 and OpenGL Extensions, allows programmers to write and run their own programs on GPUs. The vertex and pixel shaders respectively process geometrical objects vertices and image pixels while having their own limitations. Soon, hackers came up with an idea: hijack this tremendous computing power for general purpose calculations [139]. The principle: lure GPUs by passing general computations as graphics treatments. The operations are described inside a pixel shader, such that processed elements are considered as pixels. This was reserved to hot-headed computer scientists as it requires knowledge in shader languages (GLSL, Cg), OpenGL or DirectX and expertise to overcome the numerous restrictions, such as floating point precision, lack of integers. For an example of OpenGL implementation, we direct the reader to sort algorithms [108] implementations. The release of Brook [28], a stream computing language ancestor of CUDA and OpenCL and built on top of OpenGL and DirectX, was a significant step towards GPU computing. For further details, the reader may consult an implementation of a parallel prefix sum [62] in Brook. The graphics processors successive gains in generality are denoted by the concept of General Purpose Graphics Processing Units (GPGPU). The natural continuation was the release of the CUDA framework and OpenCL later-on, to easily harness GPUs as general purpose coprocessors. Yet, GPUs evolved to complex pieces of hardware, rather complicated to use at full potential. In fact, graphics processors have the difficult task to accommodate graphics treatments and generic computations. A performance trajectory between CPUs and GPUs is depicted in Fig. 3.1. The plotted values represent the theoretical peak GFlops based on the constructors hardware specifications. This computing throughput potential has many applications, particularly in High Performance Computing (HPC). However, the price to pay w.r.t. development time to achieve a good level of performance is high. Under these conditions, automatization becomes a requirement, as a mean to optimize the code, or to distribute the computations over the available computing resources.

Automatic code optimization is by no means a recent research topic. In that field the polyhedral model broke through due to its mathematical frame and applicability (although restricted), in comparison to traditional techniques. State of the art tools, such as PIPS [63], PLUTO [25], Cetus [80], Rose [122] have already been tackling shared memory processor code optimizations. Concerning GPUs, effective code generation was enabled by the release of PPCG [145]. The polytope model and tools will be presented in Section 3.7 and following.

GPGPUs have a particular execution model, in that they explicitly expose a mix of typical *Flynn's* MIMD and SIMD models to the programmer. Execution model is definitely throughput-oriented: available parallelism serves to hide memory latency. Accordingly, the user is generally encouraged to run several hundreds of threads. This

contrasts with the traditional multi-core CPU programming habit, since adding more threads would overwhelm the scheduler. For further details, an overview of the architecture is given in Section 3.2.1.

Numerous sources have reported huge speedups achieved on GPUs [56, 102]. Although, competitors have complained to the numbers reliability [81]. The scope of this thesis is not to make an architectural comparison, but to benefit from either CPU, GPU or both CPU + GPU. The GPUs are arguably an architecture to be looked at in future hardware design and software should take that into account.

The accessibility of GPUs as a computation medium, gave momentum to a tremendous effort to port algorithms. In a mainstream context: video encoding/decoding, image transformations, parts of operating systems [136] to more fancy applications, such as bitcoin mining, have been targeting GPUs. This was not always for the best, as shown in [111] for porting a lossless compression algorithm to the GPU. However, good ending stories are legion in scientific context, even with irregular workload, for instance in finding the collision-free path to displace objects [78], object collision detection and bounding volume [71], Viola and Jones face-detection [59], speech recognition algorithms [64] and solving boolean-satisfiability (SAT) problems [94, 37], or even more traditional linear algebra [127, 149] and more generally, scientific calculations [51] (Molecular Dynamics, Fast Fourier Transform, Dense Linear Algebra). Moreover, as a demonstration of the longevity of GPGPUs, OpenGL ES 3.1 introduced a *compute shader*, allowing general purpose calculations, closely tied to graphics computing (particularly real time object physics).

3.2 GPU architecture

GPUs philosophically differ from CPUs in their execution models. GPUs, depicted in Fig. 3.2, are throughput-oriented processors, *i.e.* they put latency optimization aside for a better use of the high level of parallelization. Current hardware typically embed a high number of cores (2496 cores for a Tesla K20). To fit on a single die, these cores are simplified and the design of the processor is rationalized. Firstly, resources are shared among multiple cores: 1 double precision unit per 8 CUDA cores, on Fermi GeForce GPUs. Secondly, there is one instruction dispatch unit shared by multiple cores, which consequently execute on a SM in a relaxed SIMD fashion, referred to as SIMT. In SIMT, the execution of threads may diverge; *i.e.* non-participating threads are deactivated on conditional branching. Finally, the highly parallel oriented architecture has hardware support to quickly spawn and schedule threads. On GPUs the lack of raw Instructions Per Cycle (IPC) is compensated by maintaining a high amount of concurrent computations inside a single SM, similarly to the functioning of an instruction pipeline. Thus, the execution of a whole group of threads (warps or blocks) does not monopolize the resources continuously. Concretely, memory latency is hidden by covering accesses with computations of another thread. To backup this execution model and to serve a huge number of threads concurrently, memory accesses benefit from a high bandwidth and specific optimizations. While the GPUs are becoming mature piece of hardware (with the addition of unified virtual addressing, unified memory, overlapped execution, PTX extensions, etc. on Nvidia GPUs), each microarchitecture

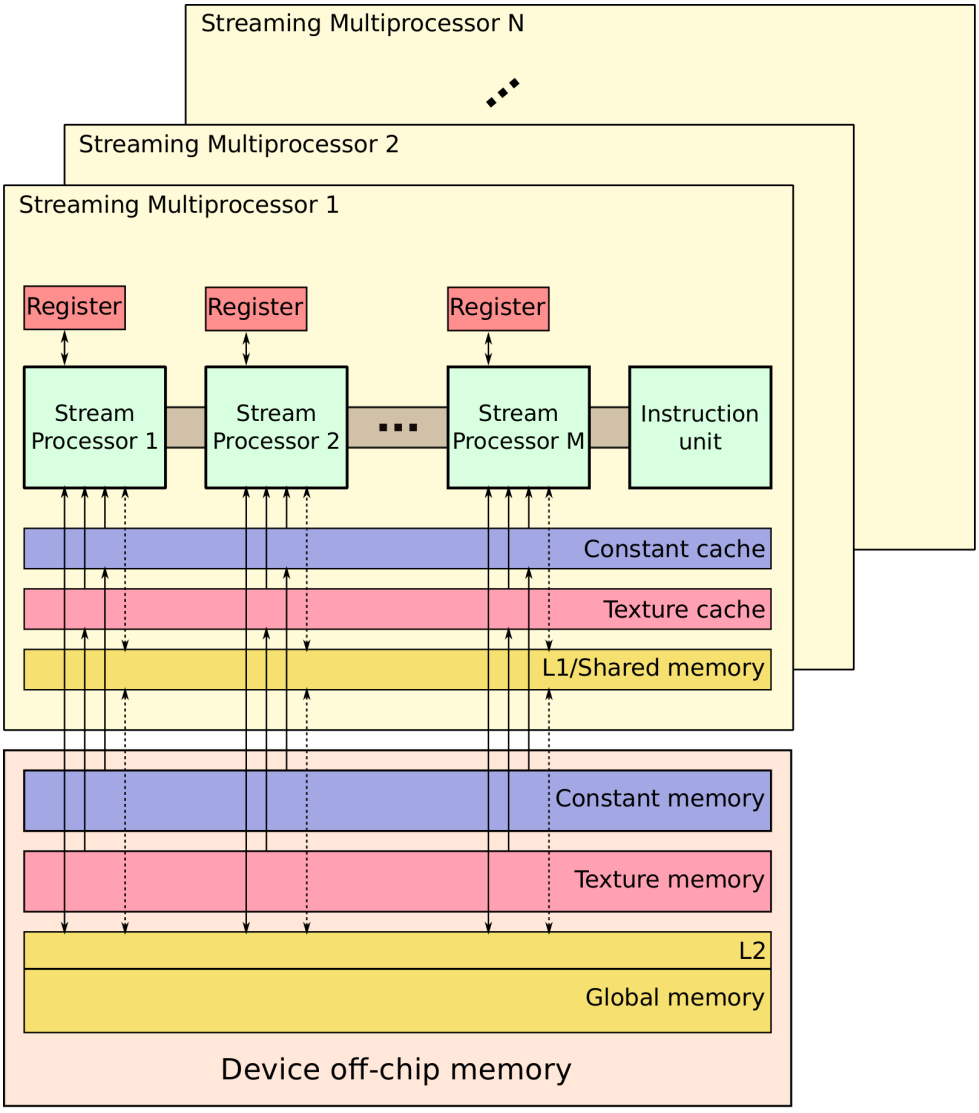


Figure 3.2 – Nvidia discrete graphics card architecture.

generation adds up new architectural tweakings. In addition, a single microarchitecture (Tesla, Fermi, Kepler, Maxwell) may offer a variety of products which differ by their specifications (Geforce, Tesla, Quadro). The notion of *Compute Capability* (CC) designates the hardware version through a *major.minor* revision number. The compute capability denotes the features exposed by the hardware, and is used to refine the code at compilation.

In this section we provide an in-depth presentation of the main architectural details of GPUs. We mainly focus on Nvidia GPUs as it is the architecture targeted in this thesis (see Section 3.2.6). A thorough understanding of the Nvidia CUDA hardware and software model is required to broach the design of a performance model, as presented in Chapter 4.

3.2.1 CUDA

Compute Unified Device Architecture [103] (CUDA) is a framework released in 2007, to target Nvidia GPU, as coprocessors for general purpose computations. CUDA designates, a processor architecture, software model and API. *Tesla*, *Fermi*, *Kepler*, *Maxwell* device generations are chronological instances of this architecture. A device generation usually adds several features and hardware modifications, such as processor and memory organization (e.g. memory banks, SMX), performance and execution conditions relaxation (e.g. coalescing, recursivity), etc. C-for-CUDA is a C language extension to program the GPU. The language adds several constructions such as kernel execution configuration, constructor initializations (e.g. for vector types), templates (e.g. for texture memory), and numerous built-ins and function qualifiers. The API adds functions to prepare and terminate execution of a CUDA kernel. The toolkit provides the *runtime* and *driver* APIs, which mainly differ by their granularity and expressiveness, to interact with the GPU.

3.2.2 Processor space

A GPU is a class of many-core, specialized processor constituted by a grid of hierarchical compute units. A Compute Unified Device Architecture (CUDA) device, is a device architecture exposing the GPU as a general purpose calculation coprocessor. In the CUDA paradigm, a GPU is composed of *Streaming Multiprocessors* (SM), themselves containing *Streaming Processors* (SP). A SM, depicted in Fig. 3.3, follows a relaxed multithreaded Single Instruction, Multiple Data (SIMD) execution model, referred to as Single Instruction Multiple Threads (SIMT), in the CUDA taxonomy. In SIMT, a SIMD lane is exposed explicitly as a CUDA core or Streaming Processor (SP).

A CUDA thread, the smallest unit of execution, is mapped on a fully pipelined SP, comparable to an Arithmetic and Logic Unit (ALU). These scalar threads process individual elements and may exhibit a divergent execution flow. Thus, a thread possesses his own context made of a program counter, state registers, etc. required for its execution and allowing fast switching. Functional units, including Special Function Units (SFU), double precision floating points, etc. are shared among multiple SPs. Comparatively, original CPU extensions such as SSE and AVX, encapsulate multiple

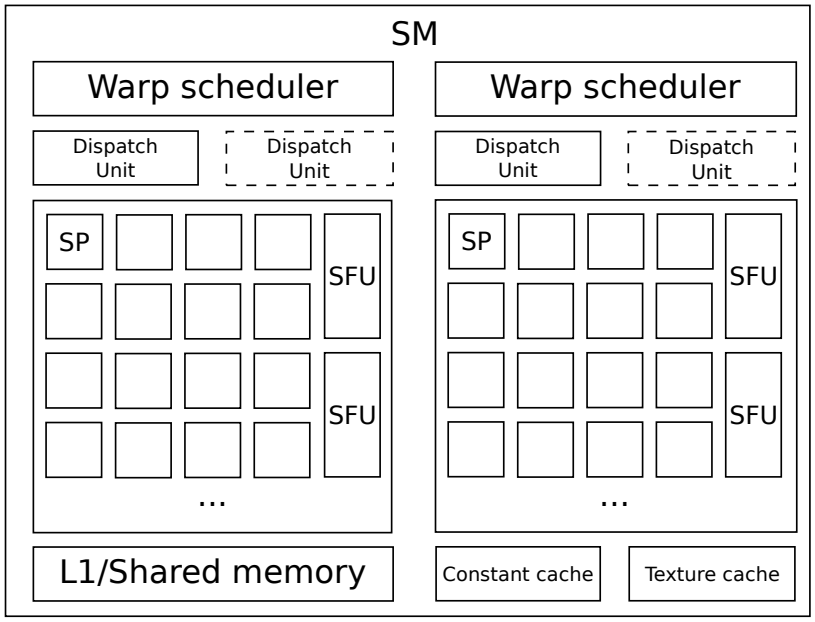


Figure 3.3 – Insight of a Streaming Multiprocessor (SM) organization.

data words into a single wide register to operate on.

Instructions are issued to groups of CUDA threads, namely *warps*, executed in lock-step. Multiple warps are run concurrently and independently on an SM. Inside a warp, divergent execution flow may lower computational intensity or, in the worst case, require conditional branches to be evaluated sequentially. Non-participating threads are disabled and standby until the whole branch is executed.

Inter- and intra- SM communications are performed via off-chip device memory and on-chip programmable scratchpad memory. To hide memory latency ($200 \sim 800$ cycles to off-chip memory), hardware schedulers implement multi-level Thread Level Parallelism (TLP). Memory pending warps or blocks are replaced by their ready-to-go counterparts, to maintain a high level of computing resources usage. All in all, the best performance is obtained by using a combination of TLP and ILP (Instruction Level Parallelism). The programmer has no direct control on the scheduler, decisions are taken internally by the hardware using heuristics. Fermi (CC 2.1) and Kepler architectures implement multiple dispatch units per warp scheduler, and thus provide superscalar capabilities. Two independent instructions can be simultaneously issued to a single warp. In Fermi, the SM schedules a warp and broadcasts an instruction to 16 cores. The cores *hotclock* frequency is twice as fast compared to a SM clock, thus a complete warp is processed in one SM cycle.

Block size should be a multiple of 32 in order to serve full warps on each SM. Note that, once a block is scheduled, it is executed until the kernel function return. Occupancy expresses the resource usage according to the hardware constraints based on register and shared memory usage and block size. It denotes the ratio of the number of active warps to the maximum number of warps. Intuitively, more occupancy may be required for memory bound applications, while less occupancy is needed if there is

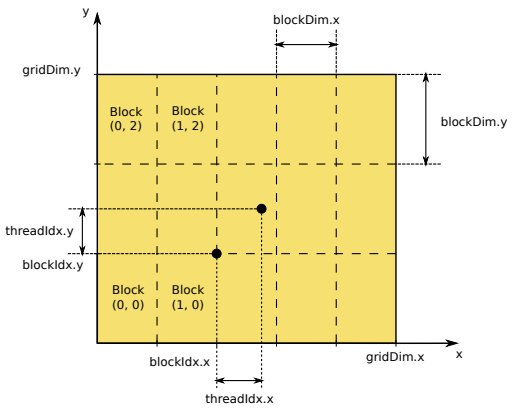


Figure 3.4 – CUDA execution model.

Instruction Level Parallelism (ILP).

3.2.3 Software model

CUDA extends C with several constructions and provides a full API to tackle Nvidia GPUs. The CUDA software model distinguishes two entities: *host* and *device*. The *host* controls the execution of the device functions, called *kernels*. A *kernel* is a function, marked with the ___global___ qualifier and intended to be executed on the GPU. Device functions, tagged with the ___device___ qualifier, are called from within a kernel and live in the calling thread. By default, a kernel is called by the host and runs asynchronously. As the memory address space of discrete GPUs (as opposed to integrated Graphics) is separated, the host carries the data movements between central and the graphics card memory. In accordance with the processor space, the execution model is decomposed into blocks of threads, constituting a grid, and respectively mapped on SMs and SPs. Figure 3.4 gives an overview of indexing threads on a 2D CUDA grid. The *execution configuration* denotes the number, size and dimension of the blocks involved in the kernel execution. Also, the size of shared memory to be allocated dynamically and the order of execution of the kernels, implied by a stream id, can be passed to the CUDA runtime. Inside a kernel, the thread position in the grid is retrieved through special registers. The scope of shared memory and synchronization mechanisms is limited to the block level. Threads are globally synchronized at the end of a kernel execution.

High-level code is translated into *Parallel Thread eXtension PTX*, an assembly language, exposing the entire public GPU programming interface. PTX code may be inlined into higher level code, inside a device function. The programmer decides on the type of device code to be embedded in the final binary: *binary images* and/or *PTX jitting*. The code generated by the *nvcc* compiler is aggregated into a *fat-binary*. A *fat-binary* embeds the binary images and/or PTX of the device code. A binary image is dedicated to run on a specific microarchitecture designated by its compute capability. The CUDA runtime is in charge of selecting the binary image compatible with the targeted GPU. In case the required binary is missing it may compile compatible PTX

code, if available. In that case, PTX is JIT-compiled by the driver and the output binary image cached for further executions on a GPU. Note that caching policy can be controlled by a user flag. As a side note, mainstream compilers such as LLVM, now implement PTX back-ends to tackle Nvidia GPUs.

A kernel termination is controlled by the CUDA runtime through different polling policies, possibly controlled by the user. As of now, this includes *host thread active polling*, *host thread yielding* and *host thread blocking*, each of these having different impacts on performance (on CPU or GPU side).

3.2.4 Memory space

3.2.4.1 Device memory

To bring data closer to the GPU, discrete graphics cards embed off-chip DRAM memory, called *global memory*. Graphics cards support PCIe *central memory-global memory* (CPU-GPU) and *global memory-global memory* (GPU-GPU) copies. A typical use case consists in copying data to the GPU before execution, for initialization purpose, and fetching back the results after execution of the kernel. In GPU computing, transfers via PCIe are generally considered the main performance bottleneck. Actually, it intrinsically depends on the kernel temporal memory usage. Computations exposing a bad $\frac{execution\ time}{transfer\ time}$ ratio, are most likely to perform better on the CPU. Static determination of this ratio, requires transfer and program behaviour characterization. In addition, CPU specification and availability should also be known.

To reduce the pressure and the latency on global memory, one may think about using *on-chip* memory. L1 and L2 cache levels accelerate memory fetches targeting to global memory. Global memory transactions are always cached through an L2 memory of maximum 768KB in current hardware. L1 caching may be disabled at compile time.

Global memory is persistent across successive kernel executions. For efficiency and productivity purposes, the memory engine of graphics cards supports Direct Memory Access (DMA). To ensure communications coherency, the range of accessed memory is pinned into physical memory. Host-initiated memory copies can benefit from this, as the data path is shorter. The so-called *zero-copy* technique, allows a running kernel to issue central memory requests. While this benefits productivity and functionality, an unconsidered use may severely impact the performance. In fact, constriction on the PCIe bus may quickly lead to 1200+ cycles per memory access. In CUDA 6.0, unified memory is an attempt to hide the complexity of the explicit transfers and let the runtime move the data around through the use of only one pointer. Modern memory copy engines support bi-directional copies and kernel execution overlapping. Execution-independent memory copies can be issued asynchronously, in different streams.

To efficiently use buses and reduce the number of independent requests, contiguous memory accesses are grouped and issued in a single transaction. Throughput efficiency is an increasing function of the number of on-flight transactions [95]. Fermi and Kepler devices relax coalescing conditions, so that:

- Accesses can be disordered,
- Multiple threads may access the same address.

Memory requests cached in L1 and L2 are packed into 128 bytes memory transactions, whereas L2-cached requests are issued in 32 bytes memory transactions.

- 8-bytes accesses by a warp (32 threads) require two 128 byte transactions, one per half-warp
- 16-bytes accesses by a warp require four 128 byte transactions, one per quarter-warp
- Misalignment may lead to multiple and fractional 128 bytes memory transactions

For alignment purpose arrays may be padded so that width is a multiple of a warp size.

The usual capacity of the global memory exceeds several Gigabytes, distributed among 6 to 8 partitions (similar to banks) in packets of 256 bytes on Tesla architecture. In fact, on older hardware, this memory organization has highlighted *partition camping* effects on performance. This issue arises when memory requests stack up on the same partition. In that case, memory requests targeting the same partition are serialized. Indeed, memory is contiguously mapped to the partitions, so that this affects tiled line-wise accesses. This behaviour is a macroscopic scale of the similar shared memory bank conflicts, described in Section 3.2.4.3.

Padding the allocation, by the width of the partition size for instance, or rearranging the coordinate system, can alleviate its effects [126]. Fermi architectures addressed this problem by introducing a hash-scheme so that consecutive data are distributed among different partitions. Depending on the block scheduling, memory access pattern and hash algorithm, these effects might still be encountered.

3.2.4.2 Registers

Registers assigned to a thread are picked from a register file duplicated on each SM. Register files respectively contain 32K and 64K, 32 bits registers, on Fermi and Kepler. A thread is limited to a maximum of 63 registers on Fermi and 255 registers on Kepler. The register pressure is directly proportional to the number of threads in a block. If so, this might reduce opportunities to hide memory latency with TLP. Also, register exhaustion[1] results in spill code: before reuse, the register content is written back to memory. Register spills are cached in the L1 and L2 caches. The compiler decides whether small constant arrays should be expanded into registers.

3.2.4.3 L1/Shared memory

The SMs embed an on-chip, scratchpad memory of 48KB on Fermi and Kepler. This memory is partitioned into a programmable *shared memory* and an automatic L1 cache. The dedicated L1- and shared memory size can be configured by the programmer.

L1/Shared memory is a low latency memory, optimized for concurrent accesses. To handle parallel accesses, shared memory is composed of 32 banks. The memory banks deliver 32 bits in 2 cycles on Fermi and 64 bits in 1 cycle on Kepler. Banks are allocated

[1]Note that 64 bits types consume 2 registers

in a round-robin fashion, so that address $0000h$ maps to bank 0 and address $007Ch$ to bank 31, on Fermi. There is a bank-conflict when multiple threads of a warp access different addresses associated to the same bank. In that case, the requests must be processed serially for each thread. The bank conflict is said *N-Way*, *N* describing the number of conflicts occurring. Note that one conflicting 64 bits access, always results in a 2-way bank conflict on Fermi. If several threads access the same memory address there is no conflict, the data is broadcasted.

Under optimal conditions, shared memory has a latency of about 50 cycles, which makes it more responsive than the underlying global memory. Efficient use of L1/-Shared memory generally leads to substantial performance improvements. Apart from hardware limitations, there is no size restriction on the use of shared memory in a block.

3.2.4.4 Constant memory

Constant memory resides in device memory and is partitioned into a compiler- and user-accessible area of 64KB each. It is backed by a 8KB non-coherent cache embedded by each SM. Constant memory is initialized by the host and is read-only from within a kernel. Constant kernels parameters, ___constant___, typically reside in the constant memory. This memory is designed to efficiently broadcast data accessed simultaneously by multiple threads. A cache hit is serviced in one cycle, whereas a cache miss is serviced at global memory speed. Separated accesses are divided into multiple requests and are serviced sequentially. Using constant cache reduces pressure on global memory by freeing up memory bandwidth.

3.2.4.5 Texture memory

Texture memory is read-only, resides in device memory, has a dedicated SM cache of 12KB [104] for Fermi and 48KB [105] for Kepler and flows through L2 cache. Constant memory focuses on temporal locality, whereas texture memory is more adapted to spatial locality fetches. Just as the constant cache, it can be used to relieve pressure on global memory. Memory address calculations can be issued to the texture unit in order to free resources for the actual computation. This memory is most suitable for unaligned memory accesses, as in other cases, L1 bandwidth is superior. Texture memory cache is referred to as *Read-Only* cache in the Kepler GPU generation.

3.2.5 OpenCL

OpenCL is a Khronos group initiative to tackle heterogeneous systems and emerged in 2009. It adds several extensions to the C language with vector types and keywords and provides an API. A single OpenCL code is aimed to run on either a CPU, GPU or other accelerator, transparently to the user. OpenCL draws its strength from its adaptability and portable design. In fact, hardware is abstracted by delaying the compilation to runtime. Through API calls, the target device is selected and the code compiled with the suitable driver. For this purpose it provides back-ends for x86 processors, AMD IL and PTX for Nvidia GPUs, and VHDL. The OpenCL programming model

is inspired from the GPU architectures. Indeed, it exposes hardware particularities, such as on-chip memory, and follows the GPU execution model. Still, it is possible to inline assembly code, in exchange for some genericity. For performance purpose, it is generally advisable to write one version of a code per target processor, to achieve performance.

3.2.6 CUDA vs OpenCL

CUDA and OpenCL mainly differ in their philosophy rather than programming model. CUDA is almost entirely dedicated to Nvidia GPUs, with the exception of the recent support for ARM CPUs. OpenCL is profoundly designed to target multiple processor architectures. Both programming frameworks almost expose the same level of performance, for similar optimizations [70, 44]. In a biased manner, one could say that CUDA may be slightly more efficient as it supports the latest GPU constructions to tackle performance. In the community, there seem to be more interest in CUDA than in OpenCL. This may be (empirically) emphasized by a *google fight* over the last few years. The main argument for us to use CUDA lies in the disposal of PPCG, a robust polyhedral compiler generating CUDA code, which is one of the basic block of this work. The major drawback of CUDA is its limitation to Nvidia GPUs. However, portability towards other architectures was not a huge concern in this thesis, as achieving performance required us to generate target-optimized codes. This portability issue can be addressed by frameworks such as Ocelot [40, 39]. CUDA is overall a mature framework and has great support, despite Nvidia proprietary policy. All in all, we are confident that most methods developed in this thesis are transposable to OpenCL.

3.3 Directive languages

Following the success of OpenMP for CPUs, introduction of directive-based languages, such as OpenACC [107] or HMPP [41] have leveraged the programmer task to exploit accelerators. Through simplified one-liner expressions, the programmer is able to parallelize or port a code to another computational architecture. Thus, language directives, hide the programming complexity induced by programming models such as CUDA or OpenCL. In practice, the programmer is generally brought to give hints on memory movements to be operated. More particularly, the directive languages often provide performance constructs in the hands of programmers, to generate more efficient code.

Although they provide means to easily target a single device, they generally lack in straightforward distribution of the computations onto all the available PUs. The work of Komoda et al. [77] is a first step towards using all resources of a machine with OpenACC. Still, it needs refinements and requires the implementation of scheduling to be efficient on heterogeneous systems. OmpSs [43] brings extensions to the OpenMP standard, in order to run computations on accelerators, while handling load balance. We aim to automatically generate code and distribute computations on CPU+GPU architectures using precise performance predictions.

Listing 3.1– Example of a straightforward *matmul* OpenACC implementation

```
void matmul(int ni, int nj, int nk, double *A, double *B,
   double *C) {
    #pragma acc kernels copyin(A[0:ni*nk], B[0:nk*nj]), copy(
       C[0:ni*nj])
    {
        #pragma acc loop gang vector(32)
        for(i = 0; i < ni; i++)
            #pragma acc loop gang vector(16)
            for(k = 0; k < nk; k++)
                for(j = 0; j < nj; j++)
                    C[i * nj + j] = beta * C[i * nj + j] +
                       alpha * A[i * nk + k] * B[k * nj + j];
    }
}
```

3.4 Approaches to performance modelling

Three compilation techniques coexist and are employed in several production compilers. Traditional *static compilation methods* [89, 52, 25, 82, 15, 145, 61, 133, 93, 35, 141, 143, 57, 8] are often unable to fully exploit hardware. This results from a conservative approach to code optimization: performance is average no matter the execution context. In fact, lack of runtime knowledge paired with genericity may prevent optimization refinements. Parametrized code transformations such as tiling, blocking, unrolling, fusion, require further knowledge of execution context. In this topic, the most prominent research involves machine learning techniques. They typically check for similarities between programs in order to predict performance. The degree of exactitude is unfortunately dependent on the training dataset. Other static compilers employ less sophisticated methods, based on heuristics, to identify optimization opportunities (for instance, smart-fuse in PLUTO [25], which determines the level of loop-fusion to find a balance between locality and parallelism).

Hybrid methods [88, 38, 50, 116, 49, 85, 120, 12, 92, 73, 79, 91] reconcile static compilation and dynamicity. Time consuming operations are relegated to static or offline stages, to perform quick decisions at runtime. The program is prepared in advance, offline collected data or execution history parametrizes a decision algorithm designed to improve performance; execution context is (at least) partially considered. A typical technique, iterative compilation, consists in feeding back by information gathered during dry runs of the target program on the target machine. The code is improved and a new binary generated by the compiler, until satisfactory results are obtained. The main grievance against iterative compilation is the expensive search space traversal. Alternatively, code multiversioning restricts the search space to a few code versions, which may be profiled offline and selected at runtime.

Dynamic methods [14, 66, 124, 31, 87, 123, 113, 22, 27, 30, 138, 129] postpone and

delegate decision making to a runtime system. Such a system is capable to detect unexpected system performance fluctuations at runtime. In fact, performance analysis occurs on partial or complete executions. The original code is prepared or instrumented by the compiler to facilitate decision making at runtime. One of the most prominent method, is Just In Time (JIT) compilation. The idea is to bring the traditional static compilation optimization passes to the runtime, when execution context is known. The to-be-jitted code is generally in a pre-compiled state, for instance Java byte code, LLVM Intermediate Representation (IR) or Parallel Thread eXecution (PTX). Some compilation passes, possibly selected by users, are applied with a view to optimize execution. However, the compilation overhead should be compensated by the benefits to be profitable.

3.5 Optimization techniques

Optimization techniques encompass the methods used in order to improve the performance of a code. In scientific programs loop nests generally concentrate the execution time and are favoured by the compilers. In contrast to recursive constructions, loop nests, made of **for**-loops in particular, are easier to analyse for compilers. Moreover, affine loop nests can be represented in a polyhedral format in order to perform legal transformations. Optimizations typically stem from program transformations which reshape the structure of a code so that it maximizes the wished property; *i.e.* performance in our case. High-level loop optimizations, such as loop- skewing, vectorization, fusion/fission, peeling, shifting, unrolling, tiling, blocking, interchange, etc. may be performed by compilers in order to improve two crucial characteristics: parallelism and data locality. This is coupled to lower-level optimizations such as loop- invariants code motion, unswitching and more traditional passes such as constant folding, constant propagation, common sub-expression elimination, etc. The number of possible transformations and their interactions makes it difficult to precisely determine the benefits. Also, on GPU, on-chip memory usage, memory access patterns and block size have a tremendous impact on performance. The listed transformations typically represent the so-called static optimizations. In the literature we identified two additional classes of optimization techniques: dynamic and hybrid optimizations. Hybrid optimizations take dynamic parameters into account in order to refine the optimizations during a compilation process or to forward information to a runtime system. Dynamic compilation techniques discover optimization opportunities at runtime. In this section, the provided references are not restricted to techniques targeting GPUs.

3.5.1 Hand-tuning vs. Automatic optimization

Hand-tuning a program is a vast and difficult matter and is generally reserved to experts. To guide programmers, a variety of tools may analyse programs and produce reports. The most notable CPU oriented tools, are Valgrind and Oprofile. Valgrind [97] runs a program and simulates the execution of every instruction. Instructions are instrumented based on the type of analysis to be performed. Oprofile [83] captures low level performance counter events with which it annotates the source code. Neverthe-

less, the GPU profiling environment is rather limited. The Nvidia profiler is part of the CUDA toolkit, and provides help to programmers in order to optimize their code. This profiler analyses executions of considered kernels in a program. For this purpose it accesses the GPU performance counters and checks memory transfers. The tool presents the results in the form of reports, highlighting the performance issues and proposes advices to address them. The provided hints encompass occupancy, program divergence issues, computation units and memory bandwidth saturation, computation/communication ratio, etc. Then, improving the program falls under the responsibility of the programmers. For performance-critical applications, unreliability, interactions and fluctuating environments are often the crucial parameters to address. In fact, it is difficult to forecast the performance of a program after the application of optimizations. Multiple different transformations result in interactions that may lower the output performance. System specifications and load, have an impact on performance as well.

A computer is cut out for quickly solving problems. Why not use this principle to solve these problems even quicker ? Automatic optimization tools relieve the programmer from performing code transformations. However, automatic tools cannot solve all the imaginable problems. They try to achieve their best with the information they are provided with. Coarse grain program analyses fall short as the program input parameters change. In general performance models are trained by running either the target program or similar codes.

Several papers [51, 54, 149, 150] have studied the performance behavior of codes ported to GPU. Yet even with the intervention of experts, it is generally a requirement to run the codes with different execution configurations. In fact, this does not perfectly fit the fact that varying problem sizes impact code performance. In the scope of this thesis we leave algorithmic issues on the borderside and focus on means to achieve performance, by confronting multiple reshaped versions of codes for instance.

3.5.2 Static methods

Achieving performance on GPUs is a problem of finding the right optimization balance [148]. Compilers, especially in the field of polyhedral compilation, are capable to generate an optimized version of a code by performing transformations. Among applied transformations, parametric optimizations generally require the programmer to perform empirical measurements to obtain further improvements. Due to their centrality to this work we provide a description of the state of the art polyhedral compilers in Section 3.9 and following.

Interactions inside of the optimization space make it hard to produce optimal code statically. Interesting results have been presented for programmer-oriented feedback systems. An analytical model known as MWP-CWP [61] (Memory Warp Parallelism-Computation Warp Parallelism) tries to consider code and architectural information in order to provide hints on where to put development efforts [133]. Altough it was demonstrated as being quite precise, this system is architecture dependent and thus must be adapted as new generations of GPUs get available. Our objective is to try to achieve portability over different GPU architectures.

The GROPHECY [93] framework is able to forecast a CPU code performance when

run on a GPU, through its hardware specification. More specifically, from a CPU source code, the programmer writes a skeleton program. The system then successively applies common transformations such as gridification, interchange, unrolling and padding. The system iterates over the transformation parameters space, to find a good performing set of parameters. Then, the MWP-CWP performance model is fed with the generated code performance parameters and characteristics to determine an execution time.

Cui et al. [35] also solicit the programmers to provide hints on code portions that present similarities in their performance characteristics. Compared to these approaches, we try to automatize the runtime selection.

3.5.3 Hybrid methods

The authors of G-Adapt [88] demonstrate that program input may have an important influence on the kernel performance. While having similarities with our approach, it mostly considers pre-optimized codes and relies on optimization hints provided by the programmer.

Peng Di et al. [38] propose a technique to automatically select a good-performing tile size on polyhedral codes. They primarily focus on doaccross loops for which they extract inter-tile and intra-tile wavefront parallelism. The generated codes expose a specific pattern, which is exploited by their prediction model. This includes memory movements to/from shared memory and synchronization points (syncthreads, syncblocks). For their part, computation performances are evaluated during simulated runs, with full tile executions. The tile size selection process is performed by successively comparing the execution times of different tile sizes configurations.

Collective mind [50] aims to resemble experiences from the scientific community in order to rationalize code tuning. The user selects uncharacterised or barely explored code optimizations and dumps the results along with partial execution context to a publicly available repository. The history of measurements is modelled through mathematical tools, such as Multivariate Adaptive Regression Splines (MARS) and Support Vector Machines (SVM) to highlight relations with the program characteristics. Compiler options can thus be chosen to either maximize performance, or minimize executable size.

Iterative compilation is generally considered as a heavyweight approach to code optimization, due to huge compilation times. In fact, even though a problem has been optimized, for a certain parameter size, it may be judicious to generate multiple versions. This makes this compilation technique a viable option for embedded systems and homogeneous systems, which have stable hardware specification.

Pouchet et al. [116] elaborate on an iterative compilation technique in the frame of the polyhedral model. For this purpose, they design an algorithm that limits the number of potential loop schedules in the search space. Relevant candidates are executed with reference parameters in order to find the best version.

Fursin et al. [49] present an alternative iterative compilation technique based on multiversioning. The proposed system assumes programs performance behaviour is decomposable into periodically stable phases. This predictable behaviour enables equitable comparison of the phases of a program performance. Thus optimizations, either

fixed at compilation or at runtime, can be evaluated for several consecutive executions. The results are bound to an execution context and stored in a database. Program phases are detected because they expose comparable performance, which enables easy comparison between several optimization strategies. Iterative compilation is performed through calls of the initial program to code section of interest. Bodin et al. [24] present an iterative compilation algorithm for embedded systems. The algorithm considers unrolling, tiling, and padding and their interactions to minimize execution time. The algorithm seeks for local execution time minimum through the manipulation of performance parameters, to eventually converge to a potential global minimum after a fixed number of steps.

Li et al. [85] elaborate on a technique that generates multiple variants of code and tries to find the right variant through empirical measurements. Typical input parameters are used to train the code selector. This study was limited to *gemm*. Ryoo et al. [128] generate multiple code variants and try to prune the search space with Pareto sets to find good code configurations.

Lomüller et al. [91] present a framework for code tuning. The considered code is rewritten in a specific language and put in a function called "compilette" in order to be tuned at runtime. An offline profiling phase evaluates the code for different input configurations to generate variants. To limit the code size and yet ensure performance the system specializes the code at runtime with deGoal [34].

B. Pradelle et al.'s framework [120] chooses and executes one of the best versions of a parallel loop nest on a multicore CPU. A python code generator prepares the code for profiling and prediction. An offline profiling phase is in charge of evaluating the code, for instance at install time. It produces a ranking table parametrized by the version and the number of threads. This study demonstrates that load balance is the most impacting dynamic performance factor on CPUs. Therefore, measurements are performed by incrementing the number of threads up to the number of available cores. To avoid cache effects, the size of the iteration domain is increased exponentially until two consecutive execution times per iteration show stability.

At runtime, as the considered loop nest is reached, the execution flow is transferred to the prediction code. Simplified versions of the loop nest count the number of iterations performed by each thread. Then, it computes the number of overlapping iterations per quantity of threads. To obtain an approximated execution time the result is multiplied by the corresponding value in the ranking table. The computation can be synthesized as:

$$time = \sum_{i=1}^{C} (it_i - it_{i+1}) * rk_i$$

where *time* represents the approximated execution time of the loop nest, C the number of cores, it_i the number of iterations per thread quantity i, and rk the ranking table storing the average execution time per iteration for i active threads. Note that $it_{C+1} = 0$. Finally, the version of the code that takes the least time is executed. While these algorithms ensure efficient load-balance on multicore CPUs, our objective is to design new profiling and selection methods to adapt to the GPU performance characteristics, while keeping the same framework infrastructure. The description of the proposed framework and a thorough description of Pradelle et al.'s method are available

in Chapter 4.

3.5.4 Dynamic methods

One famous dynamic optimization technique is Just In Time (JIT) compilation. The jitter regenerates a binary from the target code section, typically a function, while considering execution context for optimization. This technique counterbalances the initially conservative approach to code optimization of static compilers. The major drawback originates from the implied runtime overhead, provoked by the compilation passes. In fact, the benefit of the optimizations should overcome the overhead to be profitable.

EvolveTile [138] aims to dynamically select the best performing tile size on CPU with parametric loop tiling. The tile selection algorithm works by doubling or halving the tile size until a satisfactory result is found.

Bagnère et al. [14] demonstrate a multiversioning based technique, capable to dynamically select the best code version on CPU. For this purpose it makes use of switchable schedules, *i.e.* schedules that share meeting points. The system evaluates each version by switching from one to the other and successively evaluates them. The version exposing the lowest execution time is selected for the remaining execution.

Thread Level Speculation (TLS) frameworks are employed to extract parallelism of codes difficult to analyse statically and perform transformations dynamically. Speculative parallelism techniques are presented in Section 3.15.

3.5.5 Conclusion

Achieving performance on GPUs is a problem of finding the right optimization balance [148]. Interactions inside of the optimization space make it hard to produce optimal code statically. Hardware constraints make it particularly difficult to design an efficient implementation.

Typically, compilers focus on what is actually statically achievable, rather than scatter on multiple fronts. Tweakings are performed by side tools, which train a performance model to highlight potential performance hints or opportunities. We will follow this paradigm through a multiversioning technique. Full static systems, although using target specifications, may miss fine-grain performance fluctuations due to the lack of dynamic information. For instance, in comparison to static compilers, we try to choose the best block size automatically, at runtime. Moreover, finding a suitable block size is insufficient: there is no guarantee that the resulting code will have the best performance. Indeed, choosing a too large block size might drastically reduce block level parallelism, and prevent hiding memory access latencies. Too small block sizes might lead to underutilization of streaming multi-processors.

Hybrid techniques take best of the two worlds but require training in order to provide good results. Iterative compilation, for instance, requires to explore the space of effective code optimizations, which is time consuming. To solve that, we rely on a few number of versions, and pick the fastest.

Finally, dynamic methods for code optimizations may add a high level of overhead

to the execution compared to the outcome of the optimizations performed on the code. Furthermore, they may be mislead by performance fluctuations arising from execution environment.

3.6 Heterogeneous computing

Most modern systems are equipped with directly usable, multiple general purpose heterogeneous processors. Their specialized architecture, availability and programmability, make them viable alternatives to offload computations originally designed for the CPU. Typical target codes include compute-intensive subparts of bigger computations. Whereas operating system execution, input/output operations, peripheral management, etc., are handled by the CPU. A computer system is said heterogeneous when processors which it is made of, differ by their Instruction Set Architecture. A CPU, coupled with FPGAs and/or GPUs, for instance, is such a system. Heterogeneous computing aims to use these resources in order to improve execution and/or energetic performance. Efficiently addressing heterogeneity often requires different optimization strategies to fit the architecture. Indeed, a programmer is usually constrained to provide, at least, one code version per target processor. Caches with disparate properties and programmability, memory access and latency hiding mechanisms, functional units availability and clock-rate, are all to be considered. More fundamentally, internal representation of data words, numbers normalization and precision, for example, makes simultaneous use of the resources difficult in some applications. Also, targeting heterogeneity often requires to cope with the constraints of the devices, as for example memory and execution model restrictions. For board and system level heterogeneity, the inter-connection of the processors is a critical performance bottleneck.

Although, parallelizing and optimizing a code for a given architecture was already difficult, usage of multiple heterogeneous (co)processors considerably hardens the task. The objective is to schedule the programs on these resources, so that execution time is minimized. Two school of thoughts stand, and orient the design of the schedulers. A coarse grain approach consists in dispatching a program subparts according to processor affinities. Code regions are marked, either statically or at runtime and are scheduled accordingly. The fine-grain approach relies on the execution of the same piece of code on multiple processors, conjointly. The workload is chunked and distributed to the available Processing Units (PUs).

Another crucial execution performance factor, beyond data locality, is load balance. Ensuring load balance, is the art of distributing an application calculation, so that all the contributing processors are busy and terminate simultaneously. The execution time of a code must be quantified so that a system is able to distribute the workload. Interactions between the processors, for instance device polling, may substantially impact one or more resources performance. Numerous works have studied this problem on computing grids [5, 75, 132]. In the scope of this thesis we will focus on intra-machine work distribution *i.e.* PUs inter-connected via a motherboard inside a single machine.

Our target system configuration, consists in one or more multi-core CPUs with one or more GPUs. Even-though there is a strong correlation between performance and energy consumption, under-utilization might lead to significantly more power drowning.

To prevent that, a scheduler must detect and eliminate weakly contributing PUs based on their assigned workload or more elaborate power models. Some codes are evicted as they do not benefit from dispatching and would exhibit the scheduler overhead.

Scheduling techniques can be classified into three categories: static, dynamic and hybrid workload partitioning. Static methods range from simple hardware characterization to more sophisticated machine learning techniques. Fixed scheduling strategies are tied to a system configuration and dataset size. Program characterization allows compilers to take decisions based on the executions of other codes. However, inherent inaccuracies and lack of dynamic context consideration may translate into imbalance. Dynamic methods compute scheduling information and decision at runtime. Thus, they are naturally able to take dynamic context into account and distribute the workload accordingly. A pre-scheduling profiling is used to determine a workload repartition. Also, work-group size might punctually or systematically underutilize the resources. Inefficiencies are detected at runtime, through preliminary potentially hindered executions. As such, it might miss compilation opportunities as everything is deported to the runtime. Hybrid strategies rely on a static profiling which they base their decisions on at runtime. These techniques try to take the best of both worlds: performance and adaptivity. An offline profiling stage allows to quickly take decisions, while as the execution context is known, the system adapts to input dataset.

We present a thorough list of existing methods and implementations, followed by our strategy to tackle heterogeneity.

3.6.1 Static partitioning

When the system specifications, the workload size and the resource availability are known in advance, the problem may be statically partitioned. Tsoi et al. [141] elaborate on an n-Body simulation development process, targeting the *Axel* heterogeneous cluster. The nodes of the system comprise a CPU associated to a FPGA and a GPU, which specifications are known. A generic high level hardware model called Hardware Abstraction Model (HAM) encapsulates the specifications of the processors. The considered code arithmetic throughput and communication time are computed for their target processor and used to determine a minimum execution time. This information is used in conjunction with empirical runs to statically decide of a fixed workload repartition.

A simple work-sharing model, to help programmers to distribute the workload of stencils on CPU+GPU platforms is described by Venkatasubramanian et al. [143]. The lack of performance model suggests that the authors estimated the load distribution through manual experimentations.

The scheme of Grewe et al. [57], includes an AST traversal pass to build a model of the program, referencing the code features, such as arithmetic operations, memory accesses, data movements, etc. Arbitrarily selected algorithms are executed on the target machine to determine a computation partitioning. These codes are run on the CPU and the GPU, by varying the calculation distribution associated to PU from 0% to 100%. Programs which share strong affinities are grouped according to a supervised classification algorithm. The computation balance is chosen at runtime by taking as

reference quasi-similar codes.

PIPS [8] is capable to analyse the accessed memory regions and the computational complexity to give a hint on whether it is efficient to offload or not. To do so, it performs an asymptotic comparison of the two polynomials denoting an estimation of memory footprint and computational complexity. Offloading the loops is decided when computational intensity significantly exceeds memory footprint.

3.6.2 Hybrid partitioning

The StarPU runtime system [12] schedules tasks onto the available computing resources. The programmer has to write the tasks as codelets, provide their dependencies, and decide which scheduling policy to use. To maintain load balance, the *heft-tm* strategy [11] relies on a history-based time prediction to assign tasks to PUs, so that execution time is minimized. In order to characterize performance, multiple actual target code executions are required, for all execution contexts. Conversely, our framework profiles the code before the first execution of the application, thus immediately enabling maximum performance for any parameters values at runtime.

Other strategies build a history of the quality of the versions. Kicherer et al. [73] propose to run the code either on GPU, CPU or any other accelerator based on previous executions on the same hardware. While being an interesting approach, it does not consider different versions of codes. Thus it might miss some opportunities to achieve even better performance.

The Qilin framework [92] dynamically balances the workload to the CPU and GPU. The system predicts execution times, based on actual code executions. To train the performance model, the workload is evenly dispatched to the PUs. To quickly model dataset influence, the PU workload are sliced into subparts. The collected execution times are stored in a profiling database and linearly interpolated to compute predictions. To balance the load, the CPU execution time prediction is weighted by the GPU runtime scheduling requirements. Current implementation evidently only supports 2 target processors. The system determines whether work dispatching is benefiting or not and suitably maps the work to CPU, GPU or CPU+GPU.

In the SKMD system [79] the loop partitioning is computed by generating multiple combinations of compute device workload until they minimize execution time. To predict the performance, a table stores performance values expressed in work groups per millisecond. As mentioned in the paper, the target codes are hand-written. Moreover, host-device bandwidth is considered constant, which may imply prediction inaccuracies.

3.6.3 Dynamic partitioning

StarSs [113] extends the OpenMP directives with constructions to specifically handle and offload code portions. The scheduling strategy relies on a training phase during which tasks are run on the available processors, and it builds affinities between processors and tasks. Then, the tasks are associated to the processors so that the computational load and the overall execution time is minimized. This coarse-grained approach is best effort as it does not guarantee load balance.

In the HDSS scheduling scheme [22] loops are decomposed into chunks. The first chunk performs a training phase to model the performance for each compute device with a logarithmic fit function. This function is then used to determine a weight, controlling the chunk size associated to each PU. The performance model is coarse grain, in order to characterize different architectures.

Boyer et al. [27] focus on hardware availability and environment dynamicity but not on problem size influence. To train the scheduler, work groups aggregated into chunks, which size is exponentially increased at each step, are run on the target processor. Each chunk execution triggers data movements to the target processors. As an arbitrary threshold is reached the measurements are considered as relevant, training stops. Due to its dynamic nature, the system is inclined to adapt to certain external performance-impacting events, for instance, punctual clock rate scaling, processor load, etc. Based on the last execution times, the scheduler linearly dispatches the rest of the computations to the PUs. The authors only evaluated their framework for problem sizes occupying the whole memory.

An approach to heterogeneous computing is to rely on work queues in a producer/-consumer scheme. Chen et al. [30] have implemented a Molecular Dynamics algorithm on GPU. The execution control and computation feeding is operated by the CPU. GPU threads are spawned and are persistent across the computation. The CPU enqueues tasks, while the GPU actively polls for work. The end of the execution is notified by a termination message. The execution configuration is fixed and determined empirically. Tasks are finely chunked so that load balance is ensured.

3.6.4 Conclusion

The profusion of tools to address heterogeneity attests to the interest of distributing calculations. To our knowledge, the main gap concerns the lack of fully automated technique: from code generation to the execution on a heterogeneous platform. To address this challenge we demonstrate several techniques in order to run codes, typically in the form of consecutive loop nests, on CPU + GPU systems.

In the literature, current hybrid methods usually require multiple actual target code executions, for all execution contexts, in order to characterize performance. Conversely, we aim to profile the code before the first execution of the application, thus immediately enabling maximum performance for any parameters values at runtime. However, *offline* profiling is more difficult, as the profiling space needs to be taken care of.

On the other hand dynamic methods perform decisions during an actual application execution. As such, they may weaken performance, due to the series of calls to the runtime system in order to take decisions.

Overall, in comparison to the task-based StarSs [113] and StarPU [12] systems, which may stall on dependencies, we propose a runtime that makes immediate and continuous use of all the hardware resources; and it is fully automatic, once the programmer has marked the region of (sequential) code of interest with a pragma. On the other hand our framework handles only SCoP codes, which can be handled by optimizing polyhedral compilers, while StarSs and StarPU can handle any parallel code that the programmer writes.

3.7 Polytope Model

The polytope model, also called polyhedral model, is a fine-grained mathematical abstraction to analyse and manipulate programs. The polytope model primarily targets affine loop nests: statements enclosed in loops such that all loop bounds and array references are affine functions of the enclosing loop iterators and loop-invariant parameters. Usual program representations, such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), are not sufficient to model dependencies. Alias analysis and Iteration Space Graph, do not precisely capture the dependencies or rely on exhaustive impractical descriptions. In the polytope model, the executions of a statement are denoted by a set of points, contained in a polytope, defined through a conjunction of inequalities. Solutions to an optimization problem are found through linear programming techniques.

In this work we focus on statically analyzable *for*-loop nests, namely Static Control Parts (SCoPs). A code fits the polytope model if memory accesses and loop bounds are affine functions of enclosing loop iterators, *parameters* and integer constants. A *parameter* is a symbolic loop nest invariant; the set of parameters often bounds the problem size. A polytope is a geometrical object delimited by flat facets. A convex polytope is the set of solutions of a finite system of linear inequalities. The *iteration domain* of a statement is formed by integer points, representing *instances* of the statement execution. A *dependence polyhedron* captures the dependences between these points. The geometrical properties of polytopes guarantee that the points are preserved across affine unimodular transformations. A polytope may be defined in two different ways, depending on the considered problem. The vertex representation defines a polytope as the convex hull of the vertices. While the half-space representation, defines a polytope as the intersection of half-spaces. In general we denote a constraint matrix in the form $\mathcal{D} = (A|b)$. An n-dimensional iteration domain is represented by an n-polytope, a polytope of similar dimensionality. The code built from a polyhedral representation, scans integer points with respect to the dependences. A *scattering function* encapsulates the order in which the statements are executed, typically implied by a transformation.

This section provides a basic overview of the polyhedral model. The reader is invited to consult [19, 26, 117] for further explanations and technicalities involving the model. First of all, the basic mathematical objects used to characterize polytopes are defined. To illustrate this in practice we provide an example of code section which fits the polyhedral model. Then, we show the usefulness of scattering functions to perform polyhedral transformations on the code. To perform transformations it is crucial to represent access functions, as they define the dependences ruling a program execution order.

Definition 1 (Affine function)**.** Function $f : \mathbb{K}^m \to \mathbb{K}^n$ is affine iff a matrix $A \in \mathbb{K}^{m \times n}$ and a vector $\vec{b} \in \mathbb{K}^n$ exist, such that:

$$f(\vec{x}) = A\vec{x} + \vec{b}$$

Definition 2 (Affine hyperplane)**.** An affine hyperplane is an affine $(n-1)$-dimensional subspace in a n-dimensional space. For $\vec{c} \in \mathbb{K}^n$ with $\vec{c} \neq \vec{0}$ and a scalar $b \in \mathbb{K}$ an affine

hyperplane is the set of all vectors $\vec{x} \in \mathbb{K}^n$, such that:

$$\vec{c} \cdot \vec{x} = b$$

It generalizes the notion of planes: for instance, a point, a line, a plane are hyperplanes in 1-, 2- and 3- dimensional spaces.

Definition 3 (Affine half-space)**.** A hyperplane divides the space into two half-spaces H_1 and H_2, so that:

$$H_1 = \{\vec{x} \in \mathbb{K}^n | \vec{c} \cdot \vec{x} \leq b\}$$

and

$$H_2 = \{\vec{x} \in \mathbb{K}^n | \vec{c} \cdot \vec{x} \geq b\}$$

$\vec{c} \in \mathbb{K}^n$ with $\vec{c} \neq \vec{0}$ and $b \in \mathbb{K}$.

Definition 4 (Convex polytope)**.** A convex polytope is the intersection of a finite number of half-spaces. We denote $A \in \mathbb{K}^{m \times n}$ a constraints matrix, $b \in \mathbb{K}^n$ a constraints vector and P a convex polytope so that $P \subset \mathbb{K}^n$:

$$P = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + \vec{b} \geq 0\}$$

Definition 5 (Parametric polytope)**.** A parametric polytope denoted $P(\vec{p})$ is parametrized by a vector of symbols denoted $\vec{p}$. We define $A \in \mathbb{K}^{m \times n}$ a constraints matrix, $B \in \mathbb{K}^{m \times p}$ a coefficient matrix, a vector $\vec{b} \in \mathbb{K}^m$ and P a convex polytope so that $P \subset \mathbb{K}^n$:

$$P = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + B\vec{p} + \vec{b} \geq 0\}$$

Definition 6 (Polyhedron image)**.** The image of a polyhedron $\mathcal{P}^n \in \mathbb{K}^n$ by an affine function f, is another polyhedron $\mathcal{P}^m \in \mathbb{K}^m$. Notice that this is true when $\mathbb{K}^n$ is a field, but not if it is a ring like $\mathbb{Z}^n$.

Definition 7 (Iteration vector)**.** A statement instance coordinates is defined by a vector $\vec{s} \in \mathbb{K}^n$ with n the dimensionality of the loops enclosing the statement.

Note that functions are now replaced by relations in modern implementations (including PPCG).

3.7.1 SCoP

Statically analysable code sections are ideal candidates for the application of transformations due to their deterministic behaviour. A typical polyhedral side-effects free compilation unit is called a *Static Control Part* (SCoP). Originally, a SCoP is the longest section of code which gathers consecutive affine loop nests and conditional structures fitting in the polyhedral model. Now it is relaxed to any part of code that fits the model. An affine loop nest denotes a subclass of general, possibly imperfect, nested loops, whose loop bounds and conditionals are expressed in terms of affine functions of the enclosing iterators, parameters and numeric constants. A static control nest may embed several simple, multi-level statements. More specifically a statement

Listing 3.2– Example of a valid SCoP: *gramschmidt* kernel

```
#pragma scop
for(k = 0; k < nj; k++) {
    S0: nrm = 0;
    for(i = 0; i < ni; i++)
        S1: nrm += A[i][k] * A[i][k];
    S2: R[k][k] = sqrt(nrm);
    for(i = 0; i < ni; i++)
        S3: Q[i][k] = A[i][k] / R[k][k];
    for(j = k + 1; j < nj; j++) {
        S4: R[k][j] = 0;
        for(i = 0; i < ni; i++)
            S5: R[k][j] += Q[i][k] * A[i][j];
        for(i = 0; i < ni; i++)
            S6: A[i][j] = A[i][j] - Q[i][k] * R[k][j];
    }
}
#pragma endscop
```

is a source code instruction, assigning a value, potentially the result of arithmetic operations, into a memory location. Breaking the control flow with instructions such as *break*, *goto*, *return* is illegal inside a SCoP. A SCoP may either be flagged manually or detected automatically [20, 58, 146]. Optimization opportunities are tied to the length of a SCoP, *i.e.* the number of affine loop nests a SCoP aggregates.

A typical SCoP polyhedral representation requires three components: a context, a constraints matrix, and a scattering function, the latter two being bound to each statement. The context encodes the conditions on the parameters, to prune the result space. A constraints matrix defines the inequalities bounding an iteration domain. Each statement possesses its own constraints matrix. Inversely, a constraints matrix is tied to a statement, *i.e.* a constraints matrix is meaningless if it is not associated to a statement. The scattering functions provide an order on the statements in a loop nest.

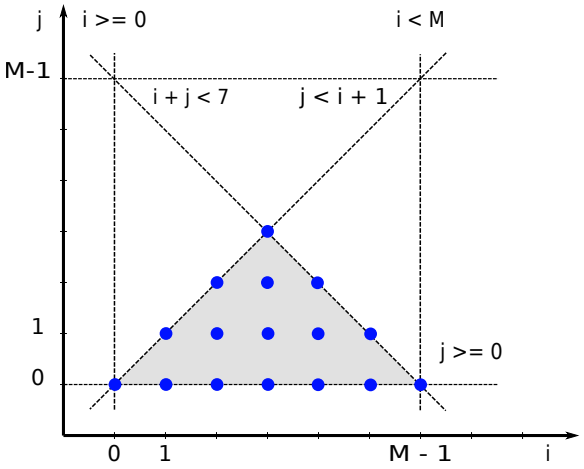
All in all, these information are gathered in order to characterize the code, to perform semantically legal transformations, detect parallelism, count the number of iterations, etc. An example of valid SCoP is provided in Listing 3.2.

Definition 8 (Iteration domain)**.** An iteration domain is the set of integer points corresponding to the actual executions of a statement. The iteration domain of a SCoP statement S can be modelled by an n-polytope, $\mathcal{D}^S(\vec{p}) \subset \mathbb{K}^n$, such that:

$$\mathcal{D}^S(\vec{p}) = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + B\vec{p} + \vec{b} \geq 0\}$$

```
for(i = 0; i < M; i++)
    for(j = 0; j < i + 1; j
        ++)
        if(i + j < 7) {
            S0(i, j);
        }
    }
}
```

(a) *for*-loop affine nest



(b) polyhedron

Figure 3.5 – Illustration of a loop nest and its associated polyhedron

$$\begin{cases} i \geq 0 \\ j \geq 0 \\ -i + M - 1 \geq 0 \\ -j + i \geq 0 \\ -j - i + 6 \geq 0 \end{cases} \qquad \mathcal{D}^{S_0}(M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{K}^n \middle| \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 \\ -1 & -1 & 0 & 6 \end{pmatrix} \begin{pmatrix} i \\ j \\ M \\ 1 \end{pmatrix} \geq 0 \right\}$$

Figure 3.6 – Constraint inequalities and associated constraint matrix

3.7.2 Polyhedral model illustration

Fig. 3.5 depicts a toy affine loop nest and its associated polyhedron. The statement named S_0 is enclosed by a two dimensional loop nest. A conditional restricts its execution when $i + j < 7$. The half-spaces intersection forms a triangular iteration domain shown in Fig. 3.5b. Every integer point (dot in the figure) corresponds to an instance of the statement. M is a parameter of the loop nest. The j-loop is bound by the outer-loop iterator, i. To express an iteration domain, the bounds are syntactically extracted to form a system of inequalities. The resulting constraint matrix encodes a canonical form of the affine inequalities of a loop nest. The polyhedron associated to statement S_0 (see Fig. 3.6) has the same dimensionality as the loop nest, 2 in this example.

3.7.3 Access functions

A statement expression groups a set of arithmetic operations on operands (memory fields or scalars) and stores the result into a memory field. Access functions of arrays are essential to determine loop-carried dependences. At the source level, a single statement may aggregate multiple arithmetic operations, whose operands are arrays or scalars. In a SCoP a memory access can be characterized by an affine function. A general

Listing 3.3– Example to illustrate memory access functions

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        x[i] = x[i] + beta * A[j][i] * y[(n - 1) - j];
```

linearized access function, for contiguously row-major allocated arrays, can be written as:

$$BA + (\vec{s}[n] \times \vec{d}[n] \times ... \times \vec{d}[1] + ... + \vec{s}[1] \times \vec{d}[1] + \vec{s}[0]) \times sizeof(*BA)$$

With $\vec{s}$, a vector containing the subscripts of the BA array, and $\vec{d}$ containing the size of the array. To be polyhedral model compliant, each entry of $\vec{s}$ must be an affine expression. This can be denoted in a matrix format. We characterize an affine access function, so that:

$$f_{\{R,W\}}(\vec{x}) = F\vec{x} + \vec{f}$$

where $F \in \mathbb{K}^{d \times v}$ is a coefficient matrix of the array subscripts, $v = dim(\vec{x})$, $\vec{x}$ is an iteration vector and $\vec{f}$ is a constant vector. R, W, indicate either a Read or a Write from/to the indiced memory field. In a compacted form $\vec{x'}$ denotes the concatenation $\vec{x'} = \begin{pmatrix} \vec{x} \\ \vec{p} \\ \vec{1} \end{pmatrix}$, of the iteration vector $\vec{x}$, the parameter vector $\vec{p}$ and a constant vector $\vec{1}$, so that:

$$f_{\{R,W\}}(\vec{x'}) = F'\vec{x'}$$

and their assignation to a memory field. This gives the following results for Listing 3.3:

$$f_{W^x}(\vec{x}) = f_{R^x}(\vec{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \Leftrightarrow x[i]$$

$$f_{R^A}(\vec{x}) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \Leftrightarrow A[j][i]$$

$$f_{R^y}(\vec{x}) = \begin{pmatrix} 0 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \Leftrightarrow y[(n-1) - j]$$

$$f_{R^{beta}}(\vec{x}) = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \Leftrightarrow (\&beta)[0]$$

3.7.4 Scattering matrix

To express the position of a statement within a loop nest, an iteration vector is not sufficient. In fact, consecutive statements, nested in the same loop cannot be totally ordered. Also assigning a single integral position to a statement reveals impractical. Feautrier [47] proposes to encode a statement position in a loop nest through a multi-dimensional timestamp. A timestamp is a logical date, similar to the conventional time system, e.g. hour:minute:seconds. To address the shortcoming of using only iteration vectors, a simple idea is to add a constant to each dimension. A statement position is then characterized by a 2-uplet per dimension: the dimension iterator and a constant. A schedule, or scattering function, gives the order of execution of the statements inside of a loop nest. The lexicographic order, denoted with $\ll$, defines the sequential (total) execution order of the statements, so that:

$$(a_1, \ldots, a_n) \ll (b_1, \ldots, b_n) \Leftrightarrow \exists i : 1 \leq i \leq n,\ \forall m : 1 \leq m < i,\ a_m = b_m \wedge a_i < b_i$$

This encoding expresses a relative order between the statements. A total execution order is deduced from the order, components by components, between the timestamps, starting from the most significant component. In other words, a statement precedes another, if one of its timestamp component is less than that of the follower and preceding m dimensions are equal. If two statements possess equal timestamps, they can be executed in any order, *i.e.* they can be executed in parallel. Also, it describes the location of a statement within a loop nest. To understand the concept of timestamps, we provide the exact schedule of Listing 3.2:

$$\theta^{S_0}(\vec{x}) = (0, k, 0, 0, 0, 0, 0)$$

$$\theta^{S_1}(\vec{x}) = (0, k, 1, 0, 0, i, 0)$$

$$\theta^{S_2}(\vec{x}) = (0, k, 2, 0, 0, 0, 0)$$

$$\theta^{S_3}(\vec{x}) = (0, k, 3, 0, 0, i, 0)$$

$$\theta^{S_4}(\vec{x}) = (0, k, 4, j, 0, 0, 0)$$

$$\theta^{S_5}(\vec{x}) = (0, k, 4, j, 1, i, 0)$$

$$\theta^{S_6}(\vec{x}) = (0, k, 4, j, 2, i, 0)$$

Definition 9 (Affine schedule function)**.** An affine schedule function maps a date in the initial iteration domain to another in the transformed domain, so that:

$$\forall \vec{x} \in \mathcal{D}^S, \theta^S(\vec{x}) = \Theta^S \vec{x} + \vec{t}$$

Cohen et al. [32] propose a canonical form of the scheduling matrix. Such a matrix is associated to each statement in the SCoP. The textual position information is

Listing 3.4– Example schedule: *syrk* from *PolyBench 3.2*

```
#pragma scop
for (i = 0; i < ni; i++)
    for (j = 0; j < ni; j++)
        S0: C[i][j] *= beta;
for (i = 0; i < ni; i++)
    for (j = 0; j < ni; j++)
        for (k = 0; k < nj; k++)
            S1: C[i][j] += alpha * A[i][k] * A[j][k];
#pragma endscop
```

interleaved with the rest of the definition.

$$\Theta^S = \left[\begin{array}{ccc|ccc|c} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ A_{1,1}^S & \cdots & A_{1,d}^S & \Gamma_{1,1}^S & \cdots & \Gamma_{1,p}^S & \Gamma_{1,p+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ A_{2,1}^S & \cdots & A_{2,d}^S & \Gamma_{2,1}^S & \cdots & \Gamma_{2,p}^S & \Gamma_{2,p+1}^S \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{d,1}^S & \cdots & A_{d,d}^S & \Gamma_{d,1}^S & \cdots & \Gamma_{d,p}^S & \Gamma_{d,p+1}^S \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_d^S \end{array}\right]$$

By convention, this matrix is defined by three sub-matrices, such that:

- $A^S \in \mathbb{K}^{d^S \times d^S}$ the coefficient sub-matrix, with d^S the depth of statement S, is applied to the iteration vectors.
- $\Gamma^S \in \mathbb{K}^{d^S \times (d^p+1)}$ the coefficient sub-matrix, with d^p the number of global parameters, is applied to the parameters.
- $\beta^S \in \mathbb{K}^{d^S+1}$ the constants sub-matrix, encodes the textual order.

Modifications in the three sub-matrices, A^S, Γ^S and β^S imply different transformations. One may obtain one or more transformations by applying a combination of changes in the sub-matrices. Loop interchange, skewing or reversal are defined in A^S. A modification in Γ^S allows to define a loop shifting. Finally, modifying β^S, is equivalent to loop fusion or fission. A^S and Γ^S are *dynamic* components as they affect instances of the iteration domain, whereas β^S is called the *static* component, as it describes the statement textual order, fixed at code generation.

To understand the utility of scheduling, we propose a concrete example. Listing 3.4 shows an implementation of the *symmetric rank-k (syrk)* kernel; in Listing 3.5 initialization loop was fused with the main computation loop. The transformed schedule can be described with the three matrices, A^S, Γ^S and β^S, so that:

$$A^{S_0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \Gamma^{S_0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \beta^{S_0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Listing 3.5– Simple re-scheduled example: *syrk* from *PolyBench 3.2*

```
#pragma scop
for (i = 0; i < ni; i++)
    for (j = 0; j < ni; j++)
        S0: C[i][j] *= beta;
        for (k = 0; k < nj; k++)
            S1: C[i][j] += alpha * A[i][k] * A[j][k];
#pragma endscop
```

Listing 3.6– Toy example benefiting from loop-shifting

```
#pragma scop
for(i = 0; i < M; ++i) {
        for(j = 0; j < N-1; ++j) {
                A[i][j] = 0;
        }
        for(j = 0; j < N-1; ++j) {
                A[i][j+1] += ...;
        }
}
#pragma endscop
```

$$A^{S_1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Gamma^{S_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

The imperfect loop nest in Listing 3.6 is a candidate for loop-shifting. The idea is to fuse the j loops to benefit from data locality on array A. To do so, a prologue and epilogue must be generated (loop peeling); initialization of array A to 0 is performed before summation. At each iteration of the fused j loop, the same memory field of A is accessed. The code in Listing 3.7 reflects the changes, for the sub-matrices A^S, Γ^S and β^S, equal to:

$$A^{S_0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Gamma^{S_0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \beta^{S_0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A^{S_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Gamma^{S_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Listing 3.7– Rescheduled toy example to allow fusion of the j loops

```
if (N >= 3) {
        for (t1=0;t1<=M-1;t1++) {
                A[t1][0]=0;
                for (t2=1;t2<=N-2;t2++) {
                        A[t1][t2]=0;
                        A[t1][t2]+=...;
                }
                A[t1][N-2+1]+=...;
        }
}
```

3.7.5 Dependence

Definition 10 (Data dependence)**.** There is a data dependence whenever two statements $S_i(\vec{x_i})$ and $S_j(\vec{x_j})$ access the same memory location, and at least one access is a write, so that an execution order is imposed.

Dependence analysis is the process that aims to detect data dependences between statements. The Bernstein conditions enumerate the requirements for two statements to be parallel:

(i) $R(S_i) \cap W(S_j) = \emptyset$

(ii) $W(S_i) \cap R(S_j) = \emptyset$

(iii) $W(S_i) \cap W(S_j) = \emptyset$

With $\mathcal{W}_S$ the set of write-accesses in statement S and $\mathcal{R}_S$ the set of read accesses in statement S, and $\theta^{S_i}(\vec{x_i}) \ll \theta^{S_j}(\vec{x_j})$. A dependence, so that, a memory location is write-after-read (i) is called an *anti-dependence*. A read-after-write (ii) to the same memory location is called a *true-dependence* or *flow-dependence*. Finally, a write-after-write (iii) is called an *output-dependence*. Output and anti-dependence may be solved through variable renaming techniques, scalar expansion [109] or array expansion [45]. Flow-dependence fix the semantics of the program and thus, must be respected. Read-after-read pseudo-dependence, may also be considered to improve memory locality for instance, as reordering does not affect the program semantics. A *legal* schedule combines transformations so that the sequential execution order of dependences is respected. A *loop carried dependence* arises when two different iterations of the loop access the same memory location. A *data dependence graph* defines the statement dependences, so that vertices represent the *source* and the *target* statements and edges the inter- and intra-statement dependences. An edge e can be translated into a *dependence polyhedron* [46] $\mathcal{P}_e$, containing the exact instance dependencies. In other words, a dependence polyhedron expresses the dependence relation between two statements. Note that P_e is a subset of the cartesian product of the iteration domains. Let us

denote S_i, the *source* and S_j the *target* of the dependence. A valid formulation of a dependence polyhedron must respect the lexicographic order, $\theta^{S_i}(\vec{x_i}) \ll \theta^{S_j}(\vec{x_j})$. Also, the instances generating the dependency must exist, $\vec{x_i} \in \mathcal{D}^{S_i}$ and $\vec{x_j} \in \mathcal{D}^{S_j}$. Eventually, the memory accesses must touch the same memory location, $f^{S_i}(\vec{x_i}) = f^{S_j}(\vec{x_j})$. The code presented in Listing 3.4 contains a flow-dependence, between $\mathcal{W}_{S_0}(C[i][j])$ and $\mathcal{R}_{S_1}(C[i][j])$. After computing the *access domain*, that is to say the image of the accesses, the following dependence polyhedron describes this dependence:

$$\mathcal{P}_e = \left\{ ((i_1, j_1), (i_2, j_2, k_2)) \middle| \begin{array}{c} 0 \leq i_1, i_2, j_1, j_2 \leq ni - 1 \\ 0 \leq k_2 \leq nj - 1 \\ i_1 = i_2 \\ j_1 = j_2 \\ i_2 \leq i_1 \\ j_2 \leq j_1 \end{array} \right\}$$

3.7.6 Transformations

Definition 11 (Transformation)**.** A transformation maps a statement instance logical date, to a date in the transformed space, with respect to the original program semantics. Transformations are typically applied to enhance data locality, reduce control code and expose or improve the degree of parallelism. The new schedule $\Theta^S \vec{x}$ might express a new execution order on statement S.

Lemma 1 (Affine form of the Farkas lemma)**.** *Let $\mathcal{D}$ be a non-empty polyhedron, defined by the inequalities $A\vec{x} + \vec{b} \geq 0$. Then, any affine function $f(\vec{x})$, is non negative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}(A\vec{x} + \vec{b}), \text{ with } \lambda_0 \geq 0 \text{ and } \lambda \geq 0$$

where λ_0 and $\vec{\lambda}$ are called Farkas multipliers.

During an optimization process, iterations might be rescheduled to expose better performance characteristics. The polyhedral model provides a frame to achieve *high-level transformations*. Such transformations are applied to higher level code representations, possibly reconstructed from binary [121]. A set of frequent and worthwhile high-level transformations encompass tiling, interchange, skewing, shifting, fission, fusion. Transformations may be generated and their validity tested thereafter. However, a typical way to find transformations, is to restrict the search space to legal schedules. Intuitively, the dependence vector orientation should be non-negative, meaning that the execution order remained equivalent, *i.e.* only the dependence distance may have changed. More formally, the difference of the schedules $\Theta^{S_j}\vec{x_j}$ and $\Theta^{S_i}\vec{x_i}$ must be positive, such that, $\forall \vec{x_i}, \vec{x_j} \in \mathcal{P}_e, \Theta^{S_j}\vec{x_j} - \Theta^{S_i}\vec{x_i} \geq 0$ [48]. In this formulation the entries of the iteration vectors $\vec{x_j}$ and $\vec{x_i}$ have unknown coefficients defined by the generic scheduling matrices Θ^{S_j} and Θ^{S_i}. The problem is reformulated to involve the dependence polyhedron and is linearized with the affine form of the Farkas lemma. According to the Farkas lemma the inequality may be described by an equality involving the Farkas multipliers, such that:

$$\forall \vec{x_i}, \vec{x_j} \in \mathcal{P}_e, \Theta^{S_j}\vec{x_j} - \Theta^{S_i}\vec{x_i} = \lambda_0 + \vec{\lambda}(A(\vec{x_i}|\vec{x_j}) + \vec{b}), \text{ with } \lambda_0 \geq 0 \text{ and } \vec{\lambda} \geq \vec{0}$$

Eventually, the Fourier-Motzkin algorithm is applied to eliminate the Farkas multipliers. The set of solutions describes the space of the legal transformations. To illustrate the usefulness of the Farkas Lemma, we take the previously defined dependence polyhedron $\mathcal{P}_e$. For convenience, the target statement iterators and redundant or unnecessary constraints are eliminated from the calculation. The application of the Farkas Lemma gives the following equation:

$$\begin{aligned} f(\vec{x}) &= \lambda_0 + \vec{\lambda} \begin{pmatrix} i_1 \\ j_1 \\ -i_1 + ni - 1 \\ -j_1 + ni - 1 \end{pmatrix} \\ &= \lambda_0 + \lambda_1 * (i_1) + \lambda_2 * (j_1) + \lambda_3 * (-i_1 + ni - 1) + \lambda_4 * (-j_1 + ni - 1) \end{aligned}$$

where $i_1 = i_2, j_1 = j_2$ and $\lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0$. A transformation shape is expressed by the function, $f(\vec{x})$, which defines two coefficients c_1 and c_2 on i_1 and j_1, and a constant c_3. From there, we deduce the equalities involving the Farkas multipliers and the coefficients:

$$f(\vec{x}) = c_1 \times i_1 + c_2 \times j_1 + c_3 \text{ such that, } \begin{cases} \lambda_1 - \lambda_3 = c_1 \\ \lambda_2 - \lambda_4 = c_2 \\ \lambda_3(ni-1) + \lambda_4(ni-1) = c_3 \\ \lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0 \end{cases}$$

The resulting system can then be used to bound the coefficients, through Fourier Motzkin elimination. Yet, any transformation picked from the resulting legal transformation space must be legal w.r.t. the other statement dependence.

A statement instance dependence must not cross iterations, for a loop dimension l to be parallel. Consequently, a polyhedron $\forall \vec{x_i}, \vec{x_j} \in \mathcal{P}_e, \Theta_i^S[l..*] < \Theta_j^S[l..*]$ must be empty, meaning that the loop does not carry a dependence. Intuitively, this means that there must not be two successive iterations accessing the same data in the same loop level, one access being a write. A dependence polyhedron can be augmented with this condition to check for parallelism.

3.7.7 Integer points counting

Counting the number of points in a polyhedron is an important problem. The number of integer points may describe the memory access counts, the communication volume, or the number of iterations of a statement. Counting the number of integer points in a polytope has a tremendous impact in code performance. *Barvinok* [147] is a polyhedral integer points counting library based on *isl* [144]. The core algorithm is based on the proposal of Barvinok to enumerate the number of points in a polytope. The result is a parametric piecewise quasi-polynomial whose coefficients are periodic numbers. *Polylib* [90], is another library capable to generate Ehrhart polynomials from a polyhedral description. In the scope of this work, these polynomials are mainly used to count the number of integer points in a polytope. Load balance and execution time per iteration calculation, are the two main applications of the polynomials in this work.

3.8 CLooG

A typical polyhedral compilation workflow, consists in taking a polyhedral code, translate it to a polyhedral representation, apply transformations and generate the new code. The Chunky Loop Generator (CLooG) [18] generates the loop nests *scanning* a union of polyhedra described in the polyhedral format. This tool is used by many polyhedral compilers, from PLUTO to mainstream projects such as Graphite or Polly. An equivalent alternative named Pet is found in PPCG for instance. CLooG provides optimizations, to avoid control redundancy. Also, the user can provide the degree of fusion of a loop nest. CLooG takes as input a polyhedral representation, defined by a context, constaint matrices and scattering functions.

3.9 PLUTO

PLUTO [25] is an automatic polyhedral source-to-source parallelizer and optimizer. The compiler takes a subset of C annotated with **#pragma** scop...**#pragma** endscop namely Static Control Part (SCoP). Clan [17], a C parser, extracts the marked code sections and generates a polyhedral representation. The loop dependencies are analysed with Candl [16], which produces a dependency polyhedron. The framework is designed to extract synchronization-free data-parallelism, or pipelined parallelism from loop nests. Transformations are expressed as hyperplanes and guarantee that the initial dependencies are preserved. The PLUTO core scheduler performs a broad range of transformations, such as skewing, fusion, reversal, shifting as well as permutation. A common technique to improve data locality consists in partitioning the iteration domain into tiles. From the legality conditions, PLUTO determines the shape of the tiles, so that, while exposing coarse parallelism, inter-tile communications and reuse distance are minimized. For this purpose, it expresses an upper bound of the required inter-tile communications. Basically, the cost function consists in diminishing the number of traversed hyperplanes by changing the direction of the dependence vectors. ClooG, translates the polyhedral representation into C or Fortran code, carrying the transformations. A post-processing phase, annotates the source code with adequate **#pragma** omp directives. The output code may also be prepared for unrolling and vectorization.

C-to-CUDA [15] is a CUDA code generation extension of PLUTO. The core algorithm is similar to that in PLUTO, with some GPU specific additions. The system looks for parallel loops in the tiled dimensions. This allows to map the tiles onto the CUDA blocks of threads, *i.e.* each parallel loop instance is mapped to a thread. Also, the cost function is extended to take coalescence of the accesses into consideration. To avoid expensive global memory accesses, a shared memory buffer is allocated based on the size of accessed data. The shared memory usage consists in a fetch, compute, store scheme with adequate block synchronization. The framework provides a mean to globally synchronize the threads as it may migrate sequential loops onto the GPU. The actual implementation does not consider data reuse and thus caches all the global accesses into shared memory, except for the following cases. Read only arrays which access function is independent w.r.t the parallel loops, are stored into constant mem-

ory. Keep in mind that independent accesses to the constant cache are serialized into several memory transactions. Conversely, array accesses scanned by the parallel loop iterators are stored in registers, as they live the whole thread life. Here, the notion of tile and block is the same, the choice of size is delegated to the programmer. All in all, the framework leaves empirical tweaking to the programmer.

3.10 Par4All

Par4All [7, 9] is a python source-to-source compiler, mainly oriented towards heterogeneous architectures: CUDA, OpenCL and OpenMP code generation is supported. To extract parallelism and control data-movements, it is based on a inter-procedural analysis framework called PIPS [63]. This tool is capable to analyse **for** and **while** loops with complex control flow and determine an over-approximation of the memory access hull. Parallel loops are detected statically using an array regions analysis without performing any particular code transformation. The parallelization process is followed by a loop fusion pass. For accelerators, Par4All extracts the parallel loops into a function and prepares the code so that it is CUDA/OpenCL-compliant, *i.e.* parallel loops are replaced by guards and their iterators are mapped on the thread grid (gridification). Par4All uses its own abstraction for the common GPU operations, such as memcpy, kernel launch and function attributes. The current implementation does not take advantage of shared memory in CUDA/OpenCL kernels. To avoid redundant data movements the compiler copies data to GPU as early as possible, and delays copy-back as late as possible.

3.11 R-Stream

R-Stream [131] is a polyhedral compiler developed by Reservoir Labs. To optimize code towards the target architecture, the compilation process is driven by architecture information stored inside XML files [82]. This makes it unsuitable for building a single generic executable and launching it on several cloudy heterogeneous target machines.

3.12 PPCG

PPCG for Polyhedral Parallel Code Generation [145], is a source-to-source polyhedral compiler supporting CUDA, OpenMP to a lesser extent and OpenCL recently. For GPUs, it generates host and device code from static control loops written in C. PPCG uses Pet, a Polyhedral Extraction Tool [146], to extract a polyhedral representation from the original source code. It builds the constraints on the domain, fetches the loops schedule and computes memory access dependencies. Pet relies on Clang as a frontend for C code. During a preprocessing phase Clang performs macro expansion on the original source code. The PPCG scheduler is largely based on the PLUTO algorithm. To comply with the CUDA model, multidimensional parallelism is extracted from the tilable bands. The tiles are then mapped to blocks and scanned by the threads. Tile and block sizes are independent, thus allowing more flexibility in finding an efficient

combination. For instance, mapping a bigger tile size w.r.t a block, translates into unrolling the thread statements. User-specified scheduling policies such as maximal or minimal fusion can be forwarded to the compiler. For minimizing global memory access contention, data reusability is detected and appropriate shared memory code is generated. Also, originally non-coalesced accesses can be fetched into shared memory in a coalesced way, for later use. Data depending only on parallel iterators are mapped to registers. If these conditions are not met, accesses are directed to global memory. The shared memory usage scheme is similar to that in PLUTO, and makes use of intra-block synchronization points. For now, PPCG only supports shared memory and register usage. By default, PPCG sizes the grid according to the problem parameters and uses a generic block size, to compute the memory requirements. All in all, PPCG allows users to provide hints on the tile, block sizes and scheduling policy. Outer sequential loops are run outside of the kernel, *i.e.* they contain a kernel call statement if a parallel loop was detected. PPCG is the state of the art polyhedral compiler targeting heterogeneous platforms. Therefore, it will be extensively used in the scope of this work.

3.13 Skeleton compilers

Skeleton compilers rely on classes which match to particular code patterns. Basically, to one code pattern is associated a semantic rule, which allows a restricted set of transformations. Thus, once a code is determined to belong to a class, a transformation, valid for that class may be applied. This is illustrated in the work of Nugteren et al. [101], who propose *Bones* a skeleton source-to-source compiler. To apply valid code transformations, the class of a code is provided by the programmer through directives. The generated code is prepared to run on GPU. The decision to run a code on GPU is made through a derivative of the *roofline model* called *boat hull model* [100].

3.14 Libraries

Thrust [21] is a library porting some of the C++ Standard Template Library operations on the GPU. Common functions include data reordering (e.g. radix sort), reductions and scan operations, generators (e.g. sequence of numbers) and transformations (e.g. vector addition, multiplication), etc. Thrust is built upon CUDA, and provides abstractions to common functions, for convenience. For example, adequate data movements must be called by the user. The library does not natively support multi-GPU parallelization. Yet, to enable multi-GPU support, the user manually transforms an application through the usage of CUDA Streams or OpenMP. CUB [1] and CUDPP [2] libraries provide similar functionalities as Thrust.

Recently, there has been a lot of interest in providing multi-GPU accelerated mathematical libraries. The available functions are often building blocks for scientific computations. *CuBLAS-XT* [106] is an Nvidia multi-GPU capable BLAS library, however, it is not clear wether load balance is maintained on heterogeneous GPUs configuration. MAGMA [142] is a similar linear algebra library that manages heterogeneous

CPU+GPU platforms by relying on dynamic scheduling provided by StarPU [12].

3.15 Speculative parallelization

The early speculative parallelization techniques rely on the inspector/executor methods. The idea is that the inspector first computes all the accesses performed by the threads. Then, it checks whether the code can be parallelized by looking at the dependence exposed by such accesses.

Rauchwerger et al. [124] use syntactical pattern matching on the statements in order to find reductions. This is followed by a dependence analysis to check that the variable is not used elsewhere in the nest. For deciding whether or not it would be good to parallelize, the authors suggest profiling or static analysis of the considered loop nest. Ratio of sequential code/parallelized code must be very low to perform speculative execution, so that it's worth it.

Cintra et al. [31] propose to check the dependencies entirely through software. The threads act on memory cells which is logged by the system. The detection of a conflict is performed by looking at the recording data-structures. Execution is performed by a window of chunks which are assigned to threads, in order to handle load balance. Liu et al. [87] pre-select tasks based on a profiling stage: evaluates prefetch ability and task potential not to be re-executed. They use a Thread Level Speculation (TLS)-enabled simulator in order to validate their proposal. In fact they assume that hardware supports TLS. Processor speedup is obtained by running tasks concurrently. The system of Raman et al. [123] targets loops, tries to execute them speculatively. Requires hardware to handle memory conflicts. This tool is capable to handle load balance.

Parangon [129] is a speculative framework that uses either CPU or GPU to perform a computation. In case of a speculative execution, the code is simultaneously executed in parallel on the GPU, and in sequential on the CPU. Memory accesses are tagged during the actual parallel computation for subsequent dependency verification.

VMAD (Virtual Machine for Advanced Dynamic analysis) [66], is a TLS framework combining a static component and a virtual machine. Multiple versions of a loop nest, differing by the applied instrumentations are statically generated. These versions communicate with the virtual machine via callbacks. Switching from one version to the other is transparent to the original application. During execution, a sample of the iteration domain is profiled in order to build affine prediction functions of the loop bounds and memory accesses. After that, a dynamic dependence analysis stage computes the legality of code transformations and extracts the available parallelism. Based on the dependence analysis the code is run in parallel after a dynamic code generation. In case of a misprediction of the speculative functions, a backup mechanism replaces the computed data with the original ones. A detailed description of the framework may be found in Chapter 6.

Chapter 4

Code versioning and profiling for GPU

4.1 Introduction

It is a difficult and tedious task to ensure program performance, especially when writing a program to be run on complex hardware such as multicore CPUs and accelerator cards. It is even more difficult in varying contexts – for example when the available hardware changes, when a binary is compiled once to be distributed on various architectures; or when the program execution itself depends on parameters that vary from one run to another or in different phases of its execution. In all these cases, programs must be able to adapt to the current context to ensure performance.

For these reasons, compilers cannot take static optimization decisions. Iterative compilation helps to choose the best performing version in a particular context, but if the input parameters or the available hardware varies it is of no help. The compiler should provide a way to adapt to the running program, by using for example dynamic compilation or multiversioning. Dynamic compilation and in particular just-in-time (JIT) compilation have the drawback of inducing a possibly large startup delay, especially when doing aggressive optimizations. Dynamic compilation is especially suited for object languages, that need runtime specialization and optimization. But if the user's concern is only performance, he would probably prefer the second alternative. In multiversioning, the compiler generates many different versions of the same piece of code or function, and the best performing one should be chosen at runtime.

This problem has been tackled by B. Pradelle et al. [120] in the scope of CPUs. At compile time their framework generates many different versions of a loop nest, analysed and transformed using the polyhedral model, with different parallel schedules and tile sizes. These versions are profiled in different execution contexts at install time. At runtime, the execution time of the different versions are predicted using the profiling results in the current execution context, and the best version is then run on the available CPU cores. Our goal is to extend this framework to heterogeneous architectures including GPUs. While the main dynamic performance factor identified on CPU was load-balance, we had to take into account other factors such as the block size, the kernel execution time variations and the memory communications, making

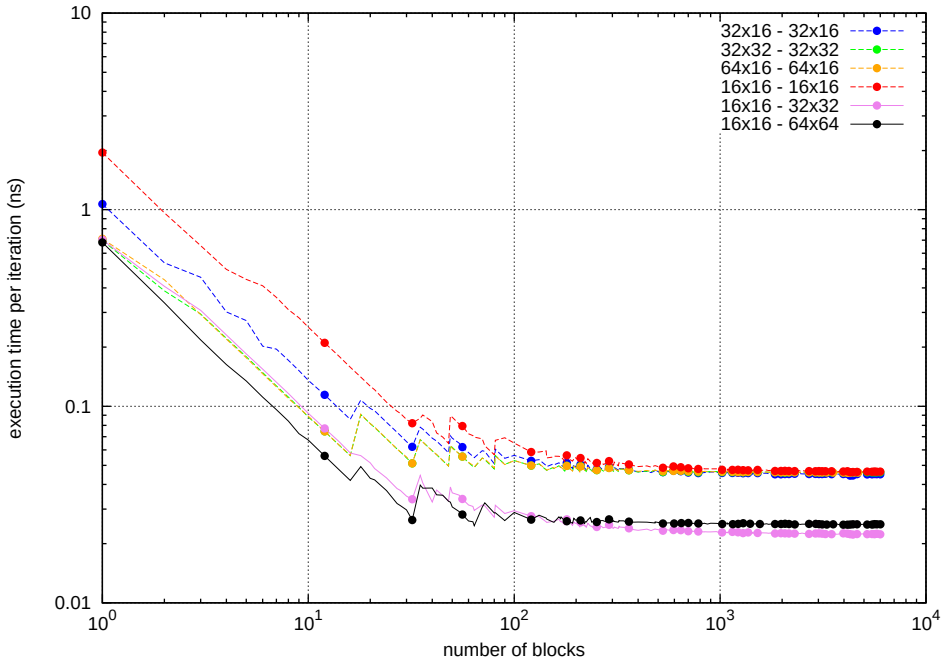


Figure 4.1 – Comparison of actual execution time per iteration for several block - tile sizes for *gemm* ($matrix\ size = 1024 * 1024$).

the profiling and the prediction algorithms completely different from Pradelle et al.'s work.

In the context of GPU computing, the problem of statically generating efficient code is crucial: decisions to be taken by the compiler are more difficult than for multicore CPUs. There are many hardware constraints of GPUs that need to be taken into account to generate the best performing code, such as choosing the right block size, avoid conflicting memory accesses in the generated kernels, minimize the memory transfers between main memory and device memory, etc. Different versions can provide the best performance depending on the context. Figure 4.1 depicts the performance variations of *gemm* with different block-tile configurations. Solid lines are associated to code versions which exploit superscalar capabilities of GPUs. One may observe that versions of equal block and tile size, for instance $64 * 16$ and $32 * 32$, expose similar execution times per iteration.

According to the input data size, the same piece of code should be run on the multicore CPU in some cases and on the GPU in others, or even taking advantage of both CPU and GPU when possible [22]. This chapter presents one of the main contributions of this work: a reliable and precise execution time prediction of parallel loop nests on GPUs, using profiling information. In chapter 5 we will make use of this prediction method to distribute workload and select versions on CPU+GPU configurations.

The prediction framework that we implemented relies on three stages: an automatic *static* code generation, an *offline* profiling and a *runtime* prediction.

4.1.0.1 Code generation

We used PPCG, the Polyhedral Parallel Code Generator [145] to generate different versions of CUDA code, starting from static control loop-nests written in C. The kernels themselves differ in the tiling that is applied to the loop nests, in the parallel schedules – that may change the loop depths of the kernel calls –, and in the fusion or fission applied to all statements of the loop nests. To do so, we generate the different versions by hand, by passing adequate arguments to PPCG. Block size should be a multiple of the warp size (32 threads in Fermi and Kepler) to only occupy full warps. The kernel calls are also parametrized by the block size of the code to be run on GPU. We designed python scripts to automatically generate the profiling and the prediction codes.

4.1.0.2 Profiling

At install time, the different versions are run with different parameters to measure their execution times on the target architecture. To predict an accurate execution time, all stages of the execution need to be considered. First, the memory copies are profiled and a table of transfer time (in both ways) per byte is filled in. Then, the different kernels are run with different values of the program parameters and using different grid sizes. We deduce from these runs the average execution time per iteration, depending on the grid size.

4.1.0.3 Prediction

At runtime, each time before running the code of the targeted loop nest, a fast *prediction nest* is run for each version, that computes a predicted execution time using the profiling results. Then, the fastest version is selected and run on the GPU and/or on the CPU as presented in Chapter 5.

This chapter is organized as follows. In Section 4.2 we detail B. Pradelle et al.'s framework. Then, the heart of our framework is presented in Section 4.3 and 4.4 for the offline profiling and in Section 4.5 for the runtime selection. Our experiments on the polyhedral benchmark suite are given in Section 4.6, and the perspectives in Section 4.7.

4.2 Related Work

One of the objectives of our work is to pick the fastest GPU kernel among multiple versions, differing by their performance characteristics. To achieve this goal, our framework is able to predict execution and transfer times.

The fundamentals of our method are based on B. Pradelle et al.'s framework [120] which chooses and executes one of the best versions of a parallel loop nest on a multicore CPU. A python code generator prepares the code for profiling and prediction. An offline profiling phase is in charge of evaluating the code, for instance at install time. It produces a ranking table parametrized by the version and the number of threads. This study demonstrates that load balance is the most impacting performance factor on CPUs. Therefore, measurements are performed by incrementing the number of

Num. threads	Execution time per iteration (ns)
1	1.265918
2	0.655208
3	0.439858
4	0.335243
5	0.383580
6	0.328795
7	0.343022
8	0.302826

(a) CPU *gemm* ranking table

Thread	Num. iterations (id)	Time (ns)
1	2048000	0
2	2048000	0
3	2048000	0
4	2048000	0
5	2048000	0
6	2048000	0
7	2048000	$384000 * 7 * 0.343022$
8	1664000	$1664000 * 8 * 0.302826$
	0	Total: $time = 4953262,848$

(b) CPU *gemm* prediction

Figure 4.2 – Example of a CPU ranking table and prediction for *gemm*.

threads up to the number of available cores. To avoid cache effects, iteration domain size is increased exponentially until two consecutive execution times per iteration show stability.

At runtime, as the considered loop nest is reached, the execution flow is transferred to the prediction code. Simplified versions of the loop nest count the number of iterations performed by each thread. Then, it computes the number of overlapping iterations per quantity of threads. To obtain an approximated execution time the result is multiplied by the corresponding value in the ranking table. The computation can be synthesized as:

$$time = \sum_{i=1}^{C} (it_i - it_{i+1}) * rk_i$$

where $time$ represents the approximated execution time of the loop nest, C the number of cores, it_i the number of iterations per thread quantity i, and rk the ranking table storing the average execution time per iteration for i active threads. Note that $it_{C+1} = 0$. Figure 4.2 presents an example of CPU execution time prediction.

Finally, the version of the code that takes the least time is executed. While these algorithms ensure efficient load-balance on multicore CPUs, we had to design new profiling and selection methods to adapt to the GPU performance characteristics.

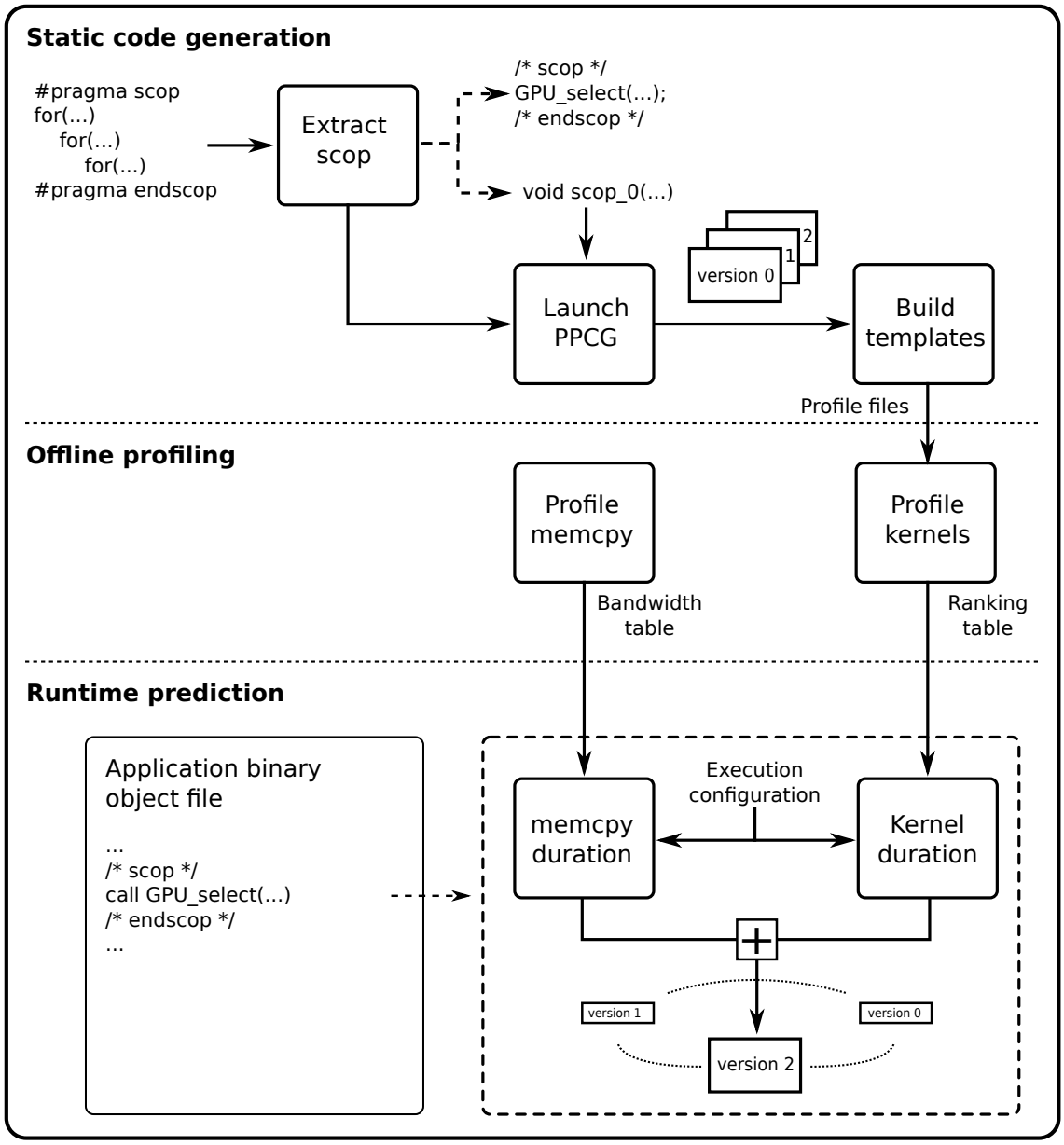


Figure 4.3 – Global framework workflow overview.

4.3 Framework overview

Our framework relies on three consecutive stages which we will develop thoroughly. It is summarized in Fig. 4.3.

As a first step, multiple and semantically equivalent versions of parallel loop nests are generated *statically*. For this purpose, we rely on PPCG, a source-to-source polyhedral compiler [145]. It produces CUDA host and device code from static control loops written in C. To comply with the CUDA model, multidimensional parallelism is extracted from the tilable bands. The tiles are then mapped to blocks and scanned by the threads. Tile and block sizes are independent, thus allowing more flexibility in finding an efficient combination. For minimizing global memory access contention,

data reusability is detected and appropriate shared memory code is generated. While applying complex transformations to the input code, PPCG allows users to provide hints on the tile and block sizes and scheduling policy. These parameters are stored in a file and passed to PPCG for generating the code versions.

We slightly modified PPCG so that it outputs context and iteration domain information of the considered kernels. For each of them, it furnishes the loop nest *parameters* involved in the sequential and parallel dimensions. We used the Barvinok counting library [147] for computing the number of times each statement is executed, expressed as parametric piecewise quasi-polynomials. Those polynomials will be instantiated in the profiling and prediction codes to respectively compute an average execution time per iteration and propose an approximation of the kernels duration. It is also convenient to keep trace of the CUDA grid size computed by PPCG. This will be useful to find back appropriate parameters size in the profiling and seek for a precise execution time per iteration in the prediction stage.

It is essential to prepare the code in order to evaluate the CUDA kernels and predict their duration. PPCG uses Pet, the polyhedral extraction tool [146], to extract a polyhedral representation from the original source code. Pet builds the constraints on the domain, fetches the loops schedule and computes memory access dependencies. It relies on Clang as a frontend for C code. During a preprocessing phase Clang performs macro expansion on the original source code. While allowing PPCG to produce optimized CUDA kernels, this behaviour prevents us from doing measurements on codes involving macros as parameters. Therefore, right before code compilation by PPCG, a python script replaces those macros by plain variables. Thus, it is possible to preserve their symbolic name for their evaluation.

With the help of the *pycparser* module, python scripts extract the static control part in the original source into a function. Then, for each version to generate, PPCG is called and fed with the function created in the previous step. The versions are given as a set of furnished PPCG configuration files. They describe the tile size, the block size and the scheduling algorithm to employ. The scripts collect the context and domain information dumped by our modified version of PPCG in order to build the profiling codes.

The second step of our framework consists in an offline profiling, that runs preliminarily on the target machine. This step will be described more precisely in Section 4.4. First, a microbenchmark models the available bandwidth between host and device. Since bandwidth depends on the message size, we build up a table representing a piecewise affine function of the message size. Second, the kernels are evaluated by simulating their execution for different execution contexts. We determined multiple performance factors that are decisive in order to make accurate predictions. Our system neglects the kernel call overhead and cache effects by increasing the domain size. However, the profiler needs to take the number of blocks into account to consider hardware memory latencies covering. Only the parallel dimensions are fixed for a given grid size, leaving freedom to the size of the other independent dimensions. The related memory incidences must also be characterized by the profiler. To diminish the number of measurements the profiler detects linear portions and automatically adjusts its behaviour.

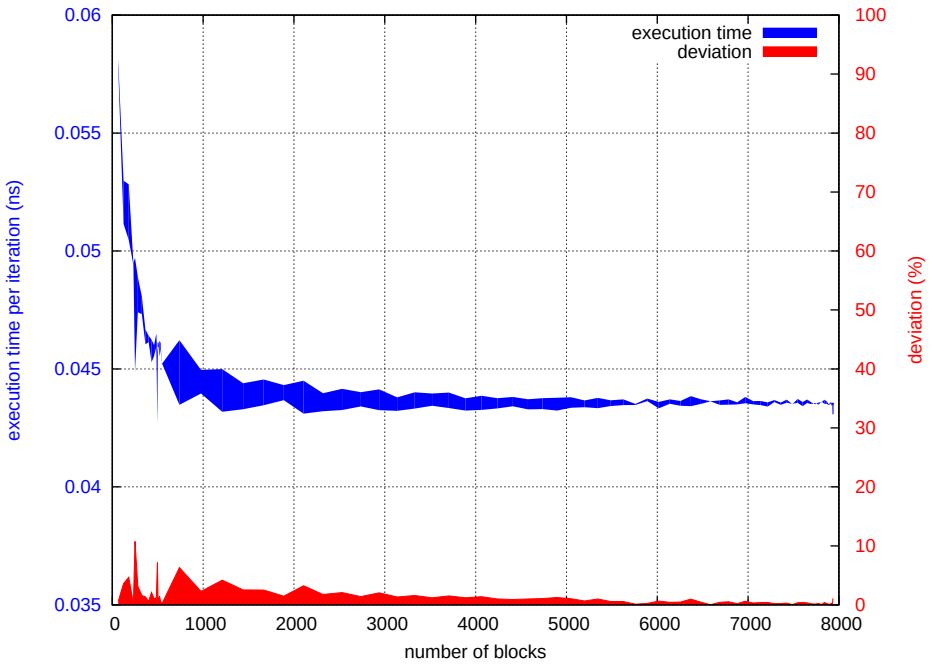


Figure 4.4 – Comparison (top) and deviation (bottom) of actual execution time per iteration to empirically demonstrate commutativity of the block dimensions for *gemm, Block/Tile size:* $32 \times 16/32 \times 16$*, NK = 2000.*

The last stage of our framework, at runtime, will be described more precisely in Section 4.5. In this step, we compute an approximation of the global execution times for each kernel. For this purpose, the prediction code uses the ranking and bandwidth tables forwarded by the profiler. In order to compute the execution times, the measurements are interpolated linearly. Once all the versions have been evaluated the one predicted as the fastest is run. At this point, we do not know whether it is better to run the code on the CPU or on the GPU.

4.4 Profiling

An offline profiling stage fetches the memory bandwidth and simulates the execution of the kernels. This step preceeds any start of the original application, at install time, for instance. Through experiments we identified multiple factors as having an influence on the execution.

4.4.1 Equity

To compare the different versions of the code it is essential to maintain equity. Instead of relying on low level hardware specificities, that would become obsolete as the hardware evolves, our framework manipulates execution times. Experiments highlighted that the performance of the codes was mostly driven by the number of threads.

To design our profiler we empirically checked the grid dimension commutativity

properties for codes. The results demonstrated that the number of blocks per dimension $sz_{x,y}$, had no major impact on code performances $perf$:

$$perf(sz_x * sz_y) \sim perf(sz_{x\prime=y} * sz_{y\prime=x})$$

Figure 4.4 depicts execution time per iteration and deviation when commuting the grid dimensions, for *gemm*. However, it highlighted that the performance of the codes was mostly driven by the number of threads. Thus, we decided to delegate block size to the version of code.

During profiling, the whole kernel duration is considered, from its call up to its completion. The recovered time is then divided by the sum of each statement domain size. The result of this computation is an average execution time per statement.

We distinguish *sequential parameters* involved in the sequential loops inside and outside of the kernels from *parallel parameters* appearing in the parallel dimensions. The parallel parameters determine the grid size. In the profiler, for maintaining homogeneity, the parallel parameters are computed to fit the required number of blocks per dimension. Sequential parameters are increased separately, until their impact becomes stable. Through experiments we observed that they follow linear patterns and thus we approximated them by linear regression. Furthermore, we observed that incomplete tiles seem not to have much incidence on the execution time.

4.4.2 Static performance factors

Static performance factors have a constant impact over the execution of the kernel. They are by nature taken into account by our framework as we measure execution times. They encompass the time taken by all instructions: arithmetic and floating point operations, branching instructions (including prediction), etc. Note that latency to execute an instruction, arising from dependencies on the input operands, may be variable. Also we ignore small variations due to constriction of the arithmetic units for instance. They always represent the same quantity of time in the execution time per iteration.

4.4.3 Dynamic performance factors

The execution time of a kernel may vary from an execution to another depending on *dynamic performance factors*. *External* factors depict the variation in the environment, and *internal* factors the kernel parameters in the application itself:

4.4.3.1 External dynamic performance factors

External dynamic performance factors may appear randomly during the execution of the kernels, making it difficult to predict them. Their frequency and duration alter the quality of the measurements. As an example, a running X server may cause efficiency fluctuations on a GPU, due to its accesses to the device. We noticed that host-to-device transfers have a performance penalty when an X server is running. Surprisingly, device-to-host communications actually seemed to benefit from a running X server on

one of our testing platforms. Concurrent kernel execution, GPU polling strategy, CPU charge or bus charge may also disturb the profiling, leading to less accuracy. Thus, to adapt to environment changes, the profiling data may be recomputed at a given frequency or when many wrong predictions occur. This is the only way that we take coarse grain external dynamic factors into account.

4.4.3.2 Internal dynamic performance factors

Internal dynamic performance factors need to be addressed specifically, unlike static performance factors. GPUs expose a complex memory hierarchy with advanced properties, such as access coalescing and SRAM memory banking. Memory access coalescing consists in issuing many simultaneous memory transactions per warp, to compensate the memory access latency penalty. Coalescing needs memory access patterns to respect certain contiguity conditions, depending on the device generation. Although multiple threads are able to access SRAM simultaneously, it is subject to bank conflicts. When multiple threads access the same bank concurrently, accesses are serialized and data is sent to all the threads that provoked the conflict, leading to as many broadcasts as there are conflicts. Many other dynamic parameters can affect code performances, we will focus on the most impacting ones.

Fermi architecture introduced an L2 cache level for global accesses and an L1 cache level coexisting with shared memory for local accesses. In this work we mostly consider larger problem sizes which represent the conventional use case of GPUs. In fact, problem sizes that fit in the cache levels have more chances to run faster. Moreover, for relatively small data sizes, cache algorithms usually introduce artifacts in the measurements, which may affect the prediction results. Finally, kernel calls overhead is quite significative for very short kernels. Thus, to avoid cache effects arising out of small problem sizes, kernel calls overhead and to alleviate coalescing and bank conflicts, the sequential parameters (tuning the kernel size itself) are increased until measurements become stable.

Preliminary experiments revealed that the predictor requires two dynamic performance factors to approximate execution times accurately. The first one deals with memory accesses contention and bank conflicts. Since the parallel parameters are computed to best fit the required number of blocks, the memory footprint they induce becomes encapsulated in the measurement. Sequential parameters need a specific treatment since their value varies for one given grid size. This mainly stems from memory access contention and bank conflicts. Also, their influence is not necessarily constant. The tests have shown variations following a linear scheme for the collected execution times per iteration. As their impact is moderate, their influence is modelled via a linear regression function. To take this into consideration, the ranking table is filled with linear functions of the sequential parameters.

The hardware block and warp scheduler is able to hide memory latencies by covering the blocks or warps waiting on memory accesses with other computations. Block level load balance is therefore the second dynamic performance factor to consider. For this, the measurements are performed for different grid sizes. We detect the linear portions of the measurements to improve the profiling speed. For each of these measurements we put an entry in the ranking table, containing a linear function of the sequential

parameters.

Figures 4.5 to 4.8 show that the ranking table values interpolations (plain line) overlap the measurements (dashed line) and thus capture the performance variations. In Fig. 4.5 and 4.6, the horizontal axis represents the number of blocks and the vertical axis the execution time per iteration in nanoseconds; the scale is logarithmic. In our experiments, the CUDA kernel profiling duration ranges from 15 minutes to 5 hours per code version, depending on the kernel durations and on the number of required kernel runs.

Figures 4.5a, 4.5d and 4.6d demonstrate the incidence of sequential parameters as the predicted and effective execution times per iteration do not perfectly overlap. Also, we noticed that increasing or decreasing a sequential parameter value by 1 may change the performance by more than 10%, especially around the powers of 2. Figure 4.7 highlights the fluctuations of the execution time per iteration for different values of the sequential parameters. To accurately map the global tendency of the curve, the linear regression computation has to get rid of noisy measurements.

4.4.3.3 Memory transfers

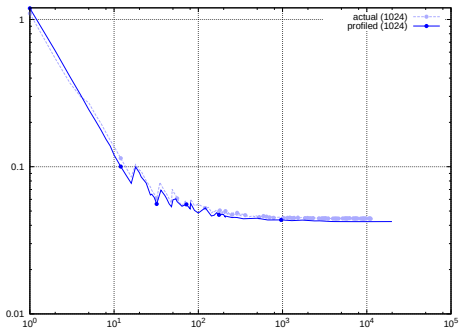
Memory transfer bandwidth is closely dependent on the underlying hardware. Motherboard specifications and data bus capabilities may have a strong influence on the available throughput. In the case of multi-GPU systems, PCI express lanes could be shared accross the graphics cards, thus drastically increasing the transfer time.

Figure 4.8 depicts the bandwidth as a function of the data size, for an Nvidia GTX 680 and an Asus GTX 590. Both devices were put into second generation PCI express slots. The observed throughput differences between the cards mainly reside in the lane width negotiation. Indeed, the GTX 590 is assigned only half of the bus link size (x8) due to the presence of another graphics card.

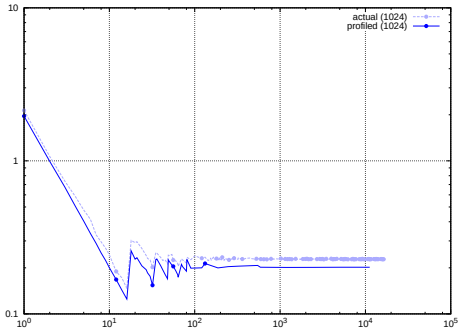
Through investigation, we observed that the bandwidth is not symmetric: host-to-device and device-to-host throughput can noticeably differ as depicted by Fig. 4.8. On the presented plot, the maximum gap between the two transfer directions is of 20 percent. Therefore, we will compute a bandwidth table per transfer direction. Besides, bandwidth is not an affine function of the transferred data size. Indeed, throughput grows and tends to the asymptotic bandwidth as we increase the size of the data. Small transfers achieve distinctly lower performance, as expected.

These parameters need to be taken into consideration, especially for codes that are bound by communications between CPU and GPU, such as *transpose*, *gemver*, *bicg* or *covariance* in the PolyBench suite [118]. The evaluation consists in launching a series of measurements on the target machine. Presently, as PPCG (version 0.01) does not take advantage of DMA, we only focus on non-pinned memory. Memory bandwidth is independent of the running codes, and thus it is not necessary to measure it for each code version.

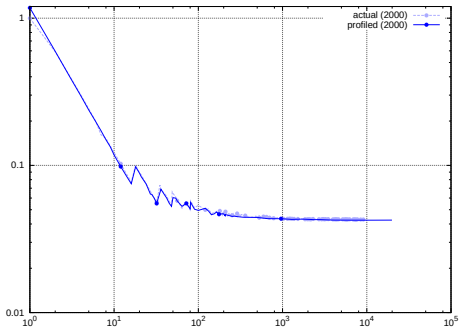
To ensure prediction accuracy it is important to consider the bandwidth variations. A good compromise consists in modelling it as a piecewise affine function. A dedicated microbenchmark collects the bandwidth measurements for various message sizes to transfer. The algorithm starts by evaluating the bandwidth for a message size of 1 byte. Further measurements are performed by increasing the message sizes with a fixed



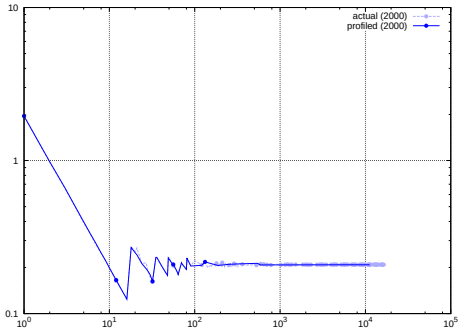
(a) *gemm* (seq.: 1024)



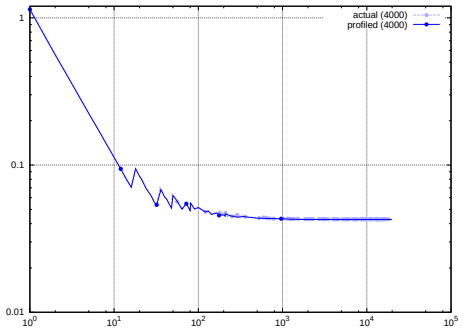
(d) *syrk* (seq.: 1024)



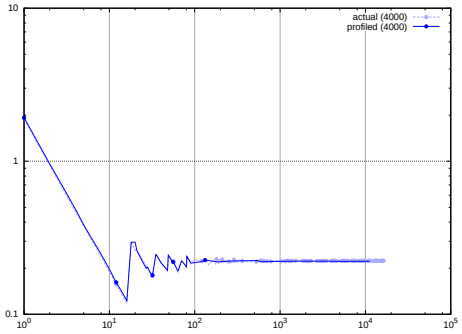
(b) *gemm* (seq.: 2000)



(e) *syrk* (seq.: 2000)



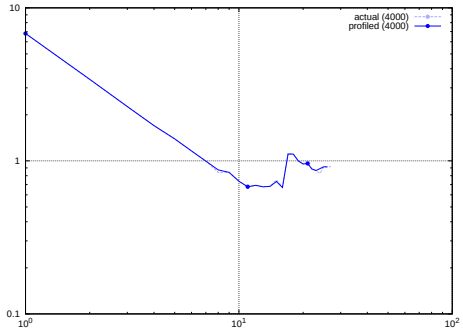
(c) *gemm* (seq.: 4000)



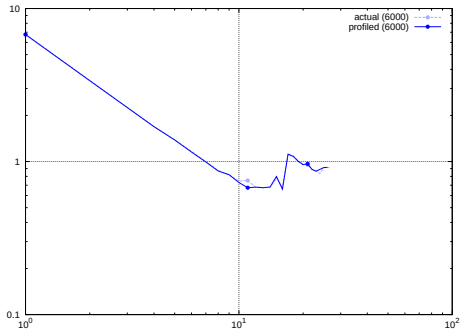
(f) *syrk* (seq.: 4000)

Figure 4.5 – Comparison between profiled and measured execution times per iteration (1).

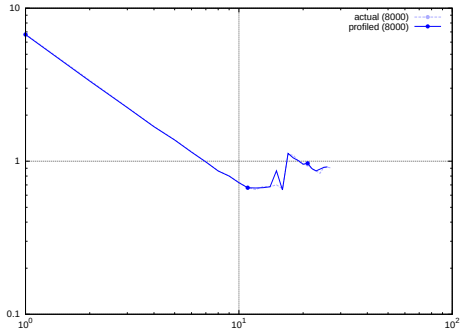
stride. As the algorithm progresses, it computes linear regression functions with the preceding points. If consecutive points follow an affine pattern, the stride is increased and measurements continue. A measurement is determined as belonging to the function if the gap between the expected bandwidth, computed by solving the regression function and the real measurement does not exceed ten percent. In the case where this condition is not fulfilled, the stride is divided and we start back at the previous measurement in order to fetch the possible intermediate variations. The use of least square approximation allows a reduction in the incidence of noise in the measurements and enables the system to take proper decisions. This algorithm continues until it reaches the memory size of the GPU.



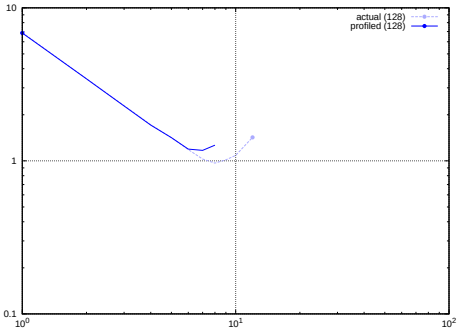
(a) *mvt* (seq.: 4000)



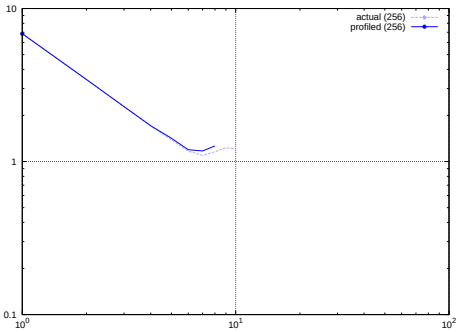
(b) *mvt* (seq.: 6000)



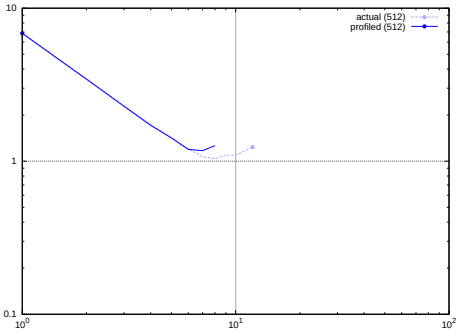
(c) *mvt* (seq.: 8000)



(d) *doitgen* (seq.: 128)



(e) *doitgen* (seq.: 256)



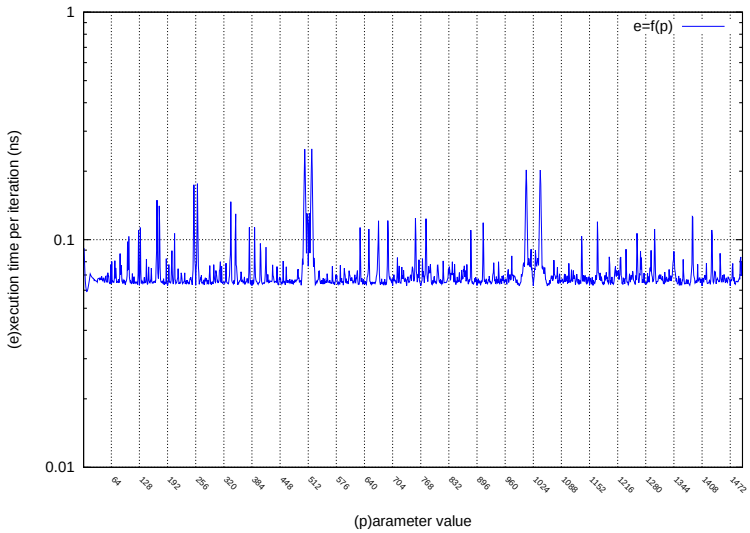
(f) *doitgen* (seq.: 512)

Figure 4.6 – Comparison between profiled and effective execution times per iteration (2).

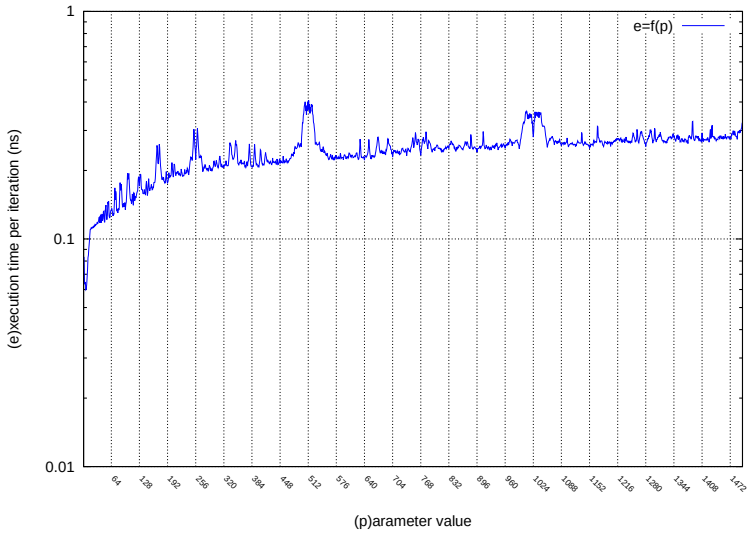
We show the accuracy of our method by showing the *real* and *profiled* measures in Fig. 4.8. As can be observed, the profiled piecewise affine curve closely follows the real transfer curve on two test platforms, and thus guarantees the quality of the subsequent predictions.

4.4.3.4 Kernel execution

Every kernel is evaluated by simulating its execution and modelling its behaviour, as shown in Fig. 4.9. To fetch the hardware capabilities the profiling is run directly on the target machine. Unlike some other methods which rely on hardware counters, our system expresses results as execution times, which ensures portability. A ranking table



(a) Evolution of execution time per iteration on sequential parameter variation for *syrk*, *Block/Tile size*: $32 \times 16/32 \times 32$



(b) Evolution of execution time per iteration on sequential parameter variation for *syrk*, *Block/Tile size*: $32 \times 16/32 \times 16$

Figure 4.7 – Sequential parameters influence for different tile configurations for *syrk*.

contains functions which correspond to execution times per iteration.

Algorithm 4.1 details the functioning of the profiling. The kernels are evaluated outside any sequential enclosing loop. We consider each kernel total execution time, from the kernel call to its termination. Before execution, arrays are filled with arbitrary values. The device is reset after each evaluation to avoid any incidence on the next run. The execution time per iteration is computed by dividing the kernel execution time by the sum of the number of iterations of each statement [147]. The result is expressed as an average statement execution time per iteration, which we refer to as *execution time per iteration*.

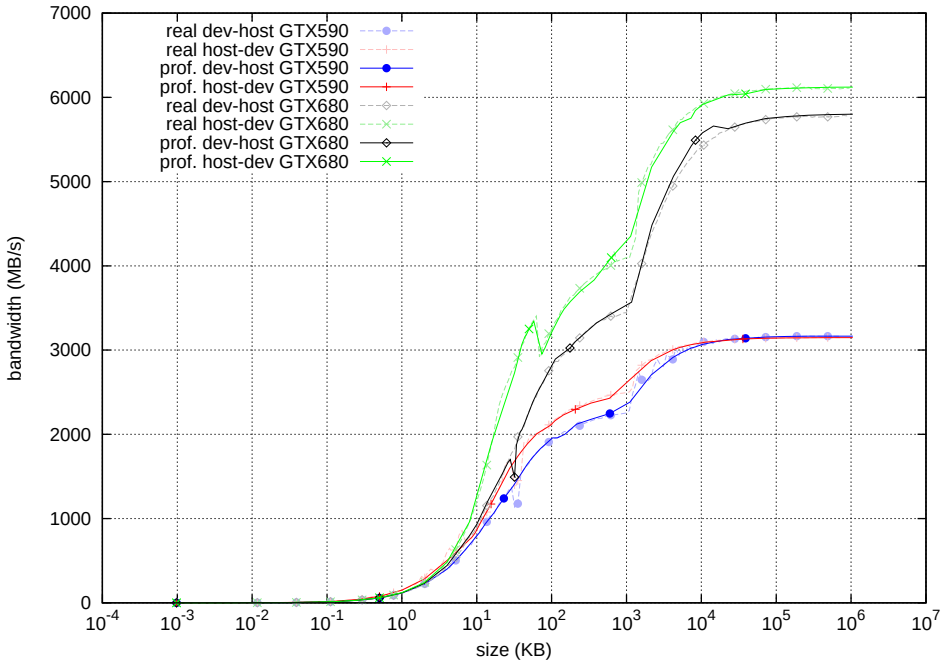


Figure 4.8 – Bandwidth evaluation on Asus P8P67-Pro motherboard with Asus GTX 590 GPU and Asus P8P67-Deluxe with Nvidia GTX 680 GPU.

The profiling code performs different measurements by controlling the grid size. Parallel parameters are calculated thanks to piecewise affine functions counting the iterations of the parallel dimensions. The combination of parameter values that first fit the grid size in each dimension is selected.

Due to the potentially high number of combinations, it is necessary to restrict the profiling domain to representative data. We use the grid commutativity property in order to avoid repeating measurements for the same number of blocks.

In addition, the profiler detects linear patterns in successive measurements. To that end, it computes the linear regression of some previous measurements. In this case, the stride is increased by a function of the error.

The behaviour observed in empirical measurements suggests that the execution time per iteration follows a piecewise affine function, see Fig. 4.1. More specifically the curves generally follow a sawtooth shape for the lower grid sizes. They have certain characteristics allowing to bound the profiling space. In fact, the profiler detects linear patterns in successive measurements. For a small number of CUDA blocks, the curve presents strong sawtooth variations. As the hardware is able to more efficiently cover memory latencies with computable blocks, this behaviour tends to fade. Eventually, execution time per iteration stabilizes for higher number of blocks. The profiler stops when it reaches the maximum number of blocks supported by the hardware.

Let $\mathcal{F}$ be a linear regression function computed with the least squares method, on the N last measured points ($N = 5$ in the current implementation). A point p is said to belong to the regression function, if the distance of p to the regression function, denoted $\mathcal{D}(p.y, \mathcal{F}(p.x))$, is lower than an arbitrary threshold (10% in the current implementation). Note that the deviation is accumulated to consider weakly

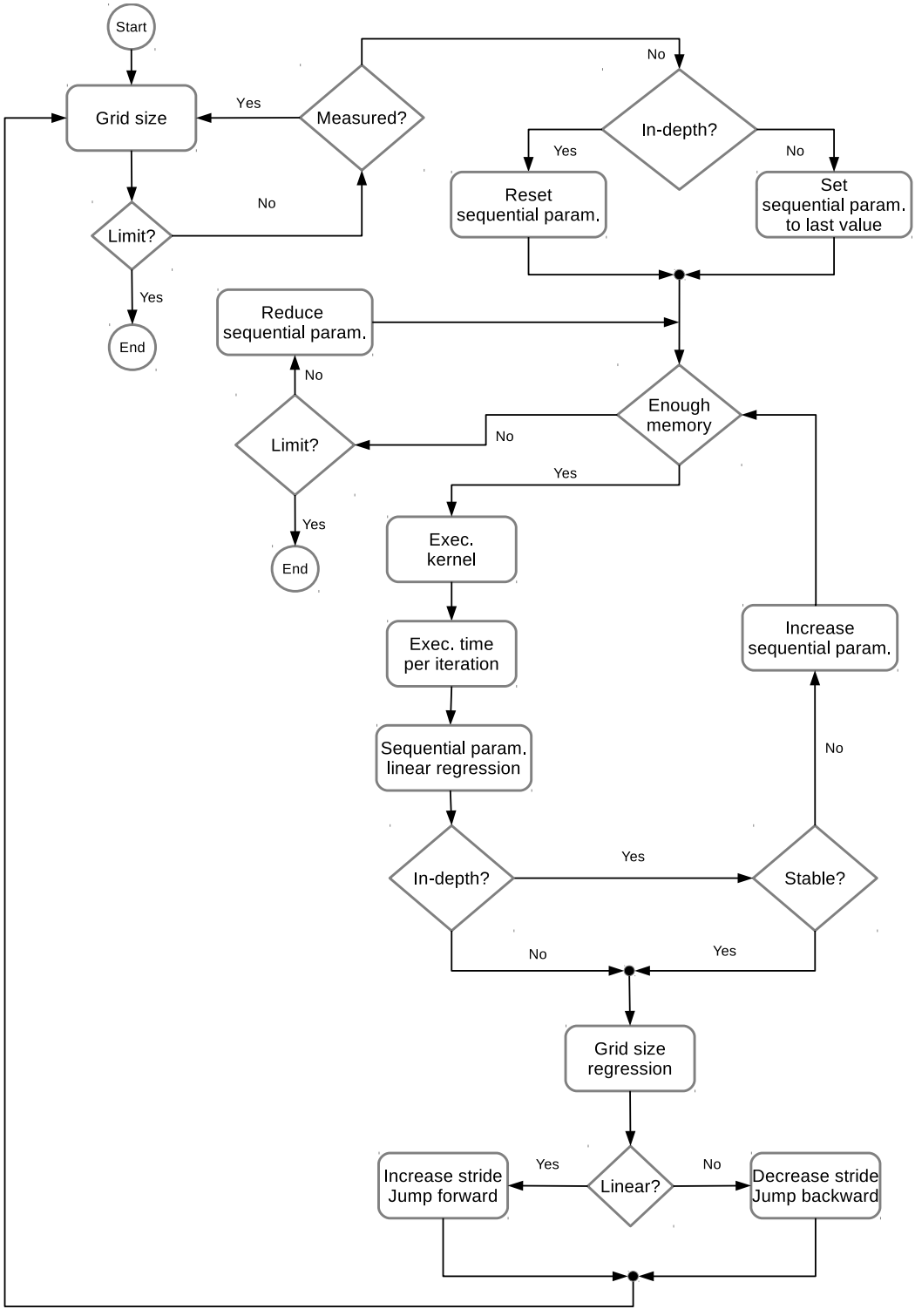


Figure 4.9 – Profiling flow chart.

Algorithm 4.1 Profiling algorithm

```
for gdim ← 1 to P do
    for gdim_x ← gdim * S to (gdim − 1) * S do
        for gdim_y ← gdim * S + 1 to (gdim − 1) * S do
            if is_dim_nok(wdim, gdim_x, gdim_y) then
                seek_dim(wdim, &gdim)
                continue
            end if
            set_parallel_param_size()
            while is_measurement_nok do
                init_array(..., memsz_0)
                init_array(..., memsz_1)
                cudaMemcpy(..., memsz_0, HostToDevice)
                cudaMemcpy(..., memsz_1, HostToDevice)
                start ← time()
                kernel0 <<< dim3(gdim_x, gdim_y), dim3(dim0_x, dim0_y) >>> (...)
                cudaDeviceSynchronize()
                end ← time()
                p.x ← gdim_x * gdim_y
                p.y ← (start − end)/card_kernel0(...)
                CHECK_SEQUENTIAL_PARAMS()
                cudaDeviceReset()
            end while
            NEXT_GRID_SIZE()
        end for
    end for
end for
```

fluctuating curve vertices. The system then proceeds to a sixth measurement, two options are available. The measurement belongs to $\mathcal{F}$, next measurement is performed for a larger grid. Otherwise, it is compulsory to recheck the preceding intervals in case fluctuations would have been missed.

The interval between measurements is computed by increasing a stride, expressed as a number of blocks, by a certain percentage. Either the point belongs to the regression function, and in this case the stride between two measurements is increased by $\mathcal{T}(\mathcal{E}(p.y, \mathcal{F}(p.x)))$ or the profiler needs to take actions. $\mathcal{T}(x)$ is an inverse function of the error, defined on $1. \leq \mathcal{T}(x) \leq 2.$, so that the algorithm progresses slowly when the error is high, and quickly when the error is low. Also, the pseudo-random behaviour of the function limits the influence of side effects which would arise on periodic patterns (*i.e.* number of blocks being a power of two) with a fixed pattern.

In the other case, the algorithm may have missed inflections, not only between the current point C, and $C-1$, but also potentially between the M last points. Intuitively, such cases can be handled with a sliding window. The idea is to jump back to the $C-J$ point with $J = stride/(2 * N), 1 \leq J \leq N$ and to divide the stride by $2 * N$. The integer division by $2 * N$ allows to adjust the window size according to the stride; the larger the stride the larger the window. Note that, the stride does not strictly follow square function due to systematic measurements error.

The most accurate profiling results were obtained for $N = 5$. Note that this moderately impacts the duration of the profiling. Also, on recovery, the stride is increased

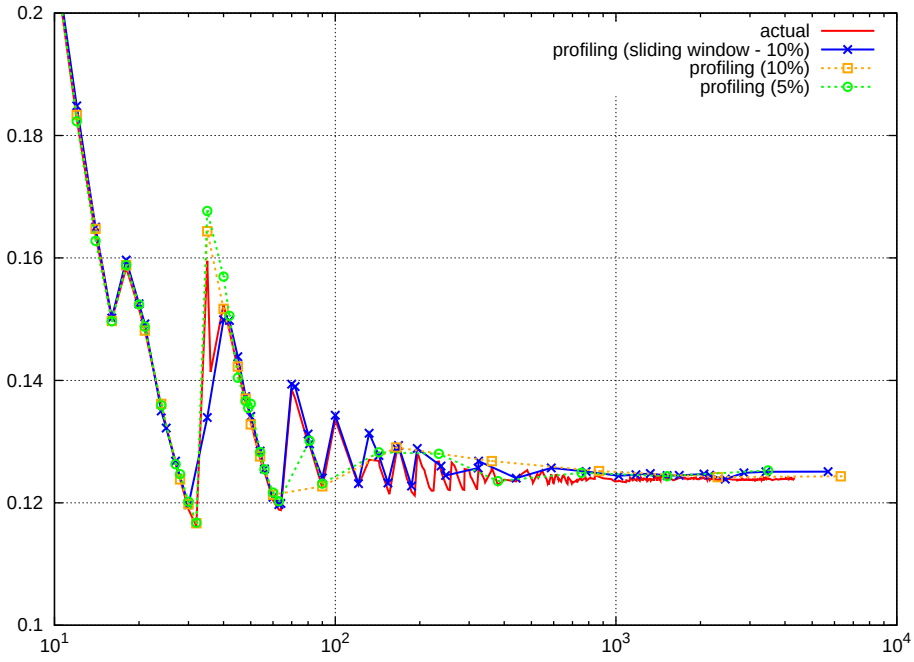


Figure 4.10 – Demonstration of the capability of our profiler to map the actual execution time per iteration measurements (solid) for *gemm, Block/Tile size:* $16 \times 16/128 \times 128$, $NK = 2000$. The profiler makes use of a parametric sliding window to perform measurements in a wider spectrum (solid, cross). This configuration is compared to a single point based decision, with a maximum allowed deviation of 10% (dashed, square) and 5% (dashed, circle).

by a percentage of the number of blocks in the interval in which the measurement falls. This reduces the profiling duration, especially when there is a failure for a larger number of blocks. This behaviour is synthesized in Algorithm 4.2.

Figure 4.10 demonstrates the capability of our profiler to accurately map the linear interpolation to the actual execution times on three experiments. Also, it provides a comparison to three decision methods (easily changeable via parameters). The first profiling (sliding window - 10%) is based on a parametric sliding window and takes its decisions on the $N = 5$ previous points. If a deviation is detected, the next measurement is performed right after the $(C - J)$th measurement. In the 30 block range, the profiler sliding window strategy missed the edge, since it converged to a different set of sequential parameters to perform the measurements. Small inflections are left aside starting at the 200 blocks range, however, this does not harm much the accuracy of predictions. Also, this could be captured by restraining the maximal deviation. The second and third profiling which respectively accept a maximal deviation of 10% and 5%, only rely on the previous measurement to take decisions. Fluctuations are clearly missed in the block interval $[60, 400]$. These experiments show that it is insufficient to only tune the maximal deviation, as variations may be missed. The sliding window technique provides a mean to detect most of the variations and thus provide better accuracy at runtime. By the way, getting exactly the same results from one simulation to the other is difficult due to the indeterministic nature of the profiler. Some

Algorithm 4.2 Grid size calculation, for next measurement

```
procedure NEXT_GRID_SIZE( )
    if p.y ∈ F then
        stride ← stride * T(E(p.y, F(p.x)))
        wdim ← p.x + stride
    else
        R ← fast_recovery(C, C − 1)
        J ← stride/(2 * N)                          ▷ 1 ≤ J ≤ N
        stride ← max(J, R)                          ▷ stride ≥ 1
        wdim ← seek_point_x(C − J)
    end if
end procedure
```

inaccuracies also arise from sequential parameters.

Sequential loops influence is characterized by making the sequential parameters vary. They are incremented by a fixed stride until stabilization. To prune measurement artefacts, a linear regression is calculated and an average error is computed. If the error is below a given threshold, the linear regression is validated. If it is not, the measurements continue, and the stride can be increased if the number of measurements becomes too large. In case no stabilization was found, half of the points are removed and measurements continue. The functioning is described in Algorithm 4.3.

Measures are considered stable when the geometric average of deviation to the linear best fit tends to 1. The profiler generates linear regression functions with combinations of points obtained for a fixed CUDA grid size. For each combination, a certain number of points are eliminated (20% in the current implementation). The linear regression function exposing the lowest standard deviation is selected and put in the ranking table. The presumably bad points are blacklisted to avoid them in the next measurements.

Furthermore, we exploit the parallelism of these computed linear regressions for consecutive grid sizes. Thus, it is possible to limit in-depth analysis to some grid sizes, in order to hold in profiling duration.

4.4.4 CUDA grid size evaluation

In CPU code parallelization, it is a common practice to only parallelize the outermost parallel loop. On CUDA architectures, the thread execution grid is multi-dimensional; consecutive enclosed parallel loops may be mapped to the CUDA grid. The CUDA grid size is expressed by the product of the number of blocks in each dimension, which forms a rectangular box. To most closely fit the potentially encountered grid sizes, the profiler must generate and perform measurements for a wide variety of grid sizes. The grid size, advocated by the profiler for the next measurement, is 1-dimensional (see Algorithm 4.2). In fact, predicting the total number of blocks for the next measurement, rather than each dimension size separately, simplifies decision making. Otherwise, one would have to increase the n dimensions and end up with a grid size that grows exponentially. Moreover, in the ranking table, we associate the total number of blocks to the measurement. This has two implications: the profiler must size the grid dimensions

Algorithm 4.3 Sequential parameters measurement noise removal

```
procedure EliminateNoise(f, p, rm, n)
    ev = 0.2 * n
    combinations(f, p, rm, n, ev)
end procedure
procedure CHECK_SEQUENTIAL_PARAMS( )
    EliminateNoise(f, p, rm, n)
    if is_stbl(f, p, n) = False then

        #Check n the number of measurements in array p - Remove and drop noisy points
        if (n mod V) = 0 then
            remove_bad_pt(rm, p)
            drop_pt(p, rm)
        end if

        #Check total number of measurements tn for current grid size (W = 40)
        #Remove half of the points
        if (tn mod W) = 0 then
            rm = half(p, n)
        end if
    else
        is_measurement_nok ← False
    end if
    SEQ_PARAM = NEXT_SEQ_PARAM()
end procedure
```

so that the total number of blocks tends to the profiler request, and to a lesser extent, size the dimensions to avoid large disparities. Two variants were studied:

4.4.4.1 Divisors lookup

Let us take an integer $n > 1$, which does not belong to the set of prime numbers. The problem is to find the multiples of n. We use the property in that, dividing n by a multiple gives another multiple of n. In the generated set take the pair of multiples, which maximizes a and b for the product $n = a * b$. This product has the property of giving an exact solution to the number of blocks to reach. In a systematic integer divisors enumeration algorithm, the first multiple $a \leq \sqrt{n}$ with $a \in \mathbb{N}^*$ starting from $\sqrt{n}$ is picked. The expectations are that the gap in blocks, between the x-dimension and the y-dimension is reasonably low. To handle disparity between the grid dimensions and the prime numbers case, one can find a more suitable grid size, close to the one requested by the profiler. For problems with dimensionality exceeding 2, that same approach may be employed recursively on the operands, b and/or a.

4.4.4.2 Iterative lookup

The current implementation uses an iterative approach to find suitable grid dimension sizes. For this purpose we generate the grid sizes with several enclosed loops, in a similar way to a tiling. An external loop gives the current search tile, enclosed loops scan the tiles. Search stops at first suitable grid size, *i.e.* a grid size smaller than- or

equal to the one requested by the profiler. As the profiler grid size request may be lower or higher, the current search tile index is adjusted. Adjusting the search tile size has an incidence on the precision of the lookup ; the gap between two consecutive grid sizes as we go increase the tile size. A window size of 1 comes down to generating a squared grid size. By the way, the total number of blocks supported by the hardware is assumed to be equal to 65535. To summarize, the following combinations of grid dimensions are generated to suit the profiler grid size request for a 2D kernels with $gdim = 1$ and a window size of $S = 10$:

$$for\ gdim = 1, \forall (dim_x, dim_y) \in \{10, 9, ..., 2, 1\} \times \{11, 10, ..., 2, 1\},$$

$$dim_x * dim_y = \{110, 90, ..., 3, 2, 1\}$$

4.5 Runtime prediction

Our code generator automatically replaces the original static control part of the program with a function call to the evaluation of the versions: $GPU\_select()$.

The function calls the prediction code and then executes the fastest version on GPU. The predictor computes a global approximate execution time, equal to the sum of all the predicted execution and transfer times per code version. At this time the execution context is entirely known. Thus, it is possible to instanciate the Ehrhart polynomials representing the domain size and the required grid size. The execution time per iteration, estimated for the current number of blocks, is fetched and multiplied by the number of iterations. The transfer time is computed from the size of the data to copy. As we reach a faster version, a pointer to the kernel wrapper is updated.

4.5.1 Prediction

Simplified loop nests serve to compute the global execution time of a code version. They are built by replacing the original kernel and memory transfer calls with their associated calculation, as shown in the example Fig. 4.11. The transfer time is computed by $ttime()$ which makes a linear interpolation of the measurements provided by the profiler. Function $ttime()$ takes as input the bandwidth table bt, the size of the message $memsz$, and, as host-to-device and device-to-host bandwidth are asymetric a third parameter indicates the direction of the transfer.

Function $etime()$ calculates one kernel execution time, using its associated ranking table rk and domain size $domsz$. First, it fetches the functions of the sequential parameters by seeking for an interval of number of blocks to which $gridsz$ belongs. The functions are solved for the average sequential parameters value, $\overline{seqsz}$. The results are then interpolated to compute the intermediate execution time per iteration. This latter value is multiplied by the domain size to deduce an execution time. The sum of the transfer time and the execution time gives an approximated total execution time.

To illustrate the functioning of the predictor we provide an example with the ranking tables in Table 4.1-4.2. Each ranking table corresponds to a specific version of *gemm*.

function ORIGINAL( )
$cudaMemcpy(..., memsz_0, HostToDevice)$
$cudaMemcpy(..., memsz_1, HostToDevice)$
for ... **do**
$kernel1 <<< (blk0, blk1), ... >>> ()$
end for
$kernel2 <<< (blk2, blk3), ... >>> ()$
$cudaMemcpy(..., memsz_2, DeviceToHost)$
end function

(a) Original code

function PREDICTION(rk, bt)
$e \leftarrow 0$
$t \leftarrow 0$
$t \leftarrow t + ttime(bt, memsz_0, hosttodevice)$
$t \leftarrow t + ttime(bt, memsz_1, hosttodevice)$
for ... **do**
$e \leftarrow e + etime(rk_0, blk0 * blk1,$
$domsz_0, avg(seqsz_0))$
end for
$e \leftarrow e + etime(rk_1, blk2 * blk3, domsz_1,$
$avg(seqsz_1))$
$t \leftarrow t + ttime(bt, memsz_2, devicetohost)$
return $(e + t)$
end function

(b) Prediction code

Figure 4.11 – A code version (a) and its associated prediction code (b).

Table 4.1 – Ranking table version 1 (v1)

grid sz	reg. fct
1248	**-0.000000*seqsz+0.037797**
1681	**-0.000000*seqsz+0.037609**
2303	**-0.000000*seqsz+0.037665**
3068	**-0.000000*seqsz+0.037559**

Table 4.2 – Ranking table version 2 (v2)

grid sz	reg. fct
567	**-0.000000*seqsz+0.038413**
841	**-0.000000*seqsz+0.038183**
1147	**-0.000000*seqsz+0.038076**
1681	**-0.000000*seqsz+0.038060**

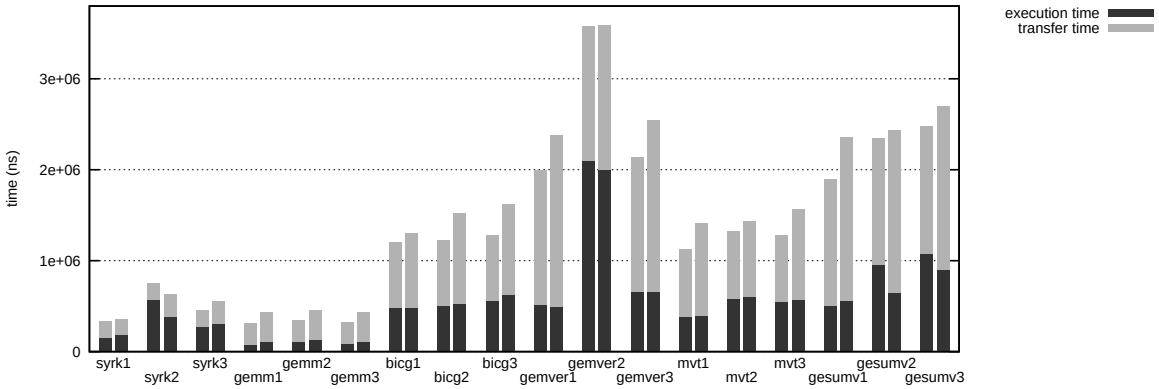


Figure 4.12 – Comparison between prediction (left bar) and real (right bar) execution times on PolyBench with small dataset on GTX590.

These versions differ by their block sizes, $blk_{v1} = (16, 32)$ for version 1, $blk_{v2} = (32, 32)$ for version 2. The loop nest parameters are set to 1024 in the original source file, thus $\overline{seqsz} = 1024$. The grid size of each version is equal to, $gridsz_{v1} = \frac{1024^2}{16*32} = 2048$ and $gridsz_{v2} = \frac{1024^2}{32^2} = 1024$.

To begin with, the predictor seeks an interval in the ranking table, such that $gridsz_v \in [gridsz_{v,lb}; gridsz_{v,ub}]$. The execution times per iteration are interpolated by solving the lower and upper bound regression functions, $it_{v1} = rk_{v1,lb1} + (rk_{v1,ub1} - rk_{v1,lb1}) * ((gridsz_{v1} - gridsz_{v1,lb})/(gridsz_{v1,ub} - gridsz_{v1,lb})) = 0,037642\ ns$ and $it_{v2} = 0.038119\ ns$. Then, a prediction of the execution time is computed by using the domain size, $e_{v1} = it_{v1} * domsz_{v1} = 0.037642 * (1024^2 + 1024^3) \simeq 40,457,260\ ns$ and $e_{v2} = it_{v2} * domsz_{v2} \simeq 40,969,935\ ns$.

Bandwidth table interpolation enables us to compute the transfer time, $t = 2 * 3,326,162 + 3,324,061 = 9,976,385\ ns$; two transfers from host to device of $3,326,162\ ns$ and one from device to host of $3,324,061\ ns$. The transfer time is similar for both kernels, thus the total execution times can be computed, $tot_{v1} = e_{v1} + 9,976,385 = 40,457,260 + 9,976,385 = 50,433,645\ ns$ and $tot_{v2} = 50,946,320\ ns$. Consequently, version 1 is selected since it has been determined as being the fastest version.

4.6 Experiments

4.6.1 Testbed

The first test platform is composed of two Asus GTX 590 plugged into an Asus P8P67-Pro motherboard. Each GTX 590 card is composed of two GPUs sharing 3 GB of GDDR5 and relying on the Fermi architecture. Note that for the benchmarks only a single GPU was used. Each graphics processor on the GTX 590 embeds a total of 512 Streaming Processors[1] ($16\ SM * 32\ SP$). The motherboard provides a PCIe 2.0 x16 bus for connecting the peripherals. The two graphics card individually support PCIe x16 and share half of the bus width (x8) in our configuration. The host processor is

[1]SM: Streaming Multiprocessors, SP: Streaming Processors

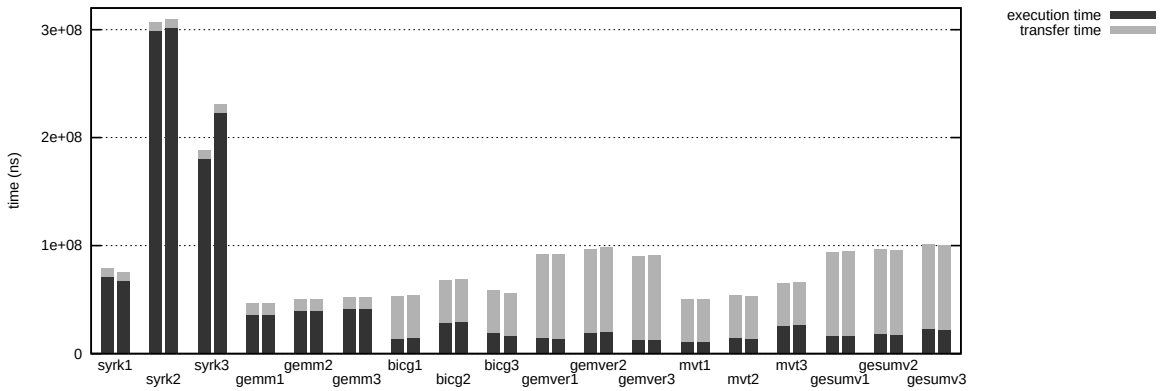


Figure 4.13 – Comparison between prediction (left bar) and real (right bar) execution times on PolyBench with standard dataset on GTX590.

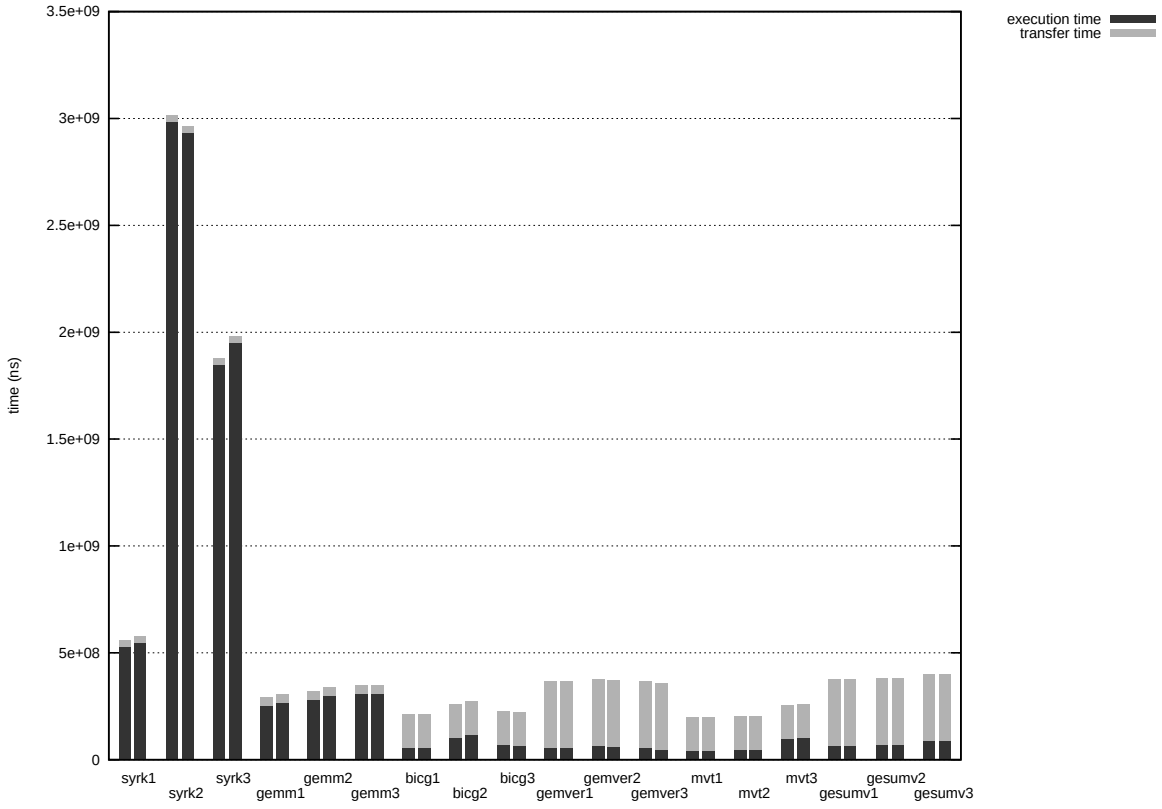


Figure 4.14 – Comparison between prediction (left bar) and real (right bar) execution times on PolyBench with large dataset on GTX590.

an Intel core i7-2700 with 4 hyperthreaded cores. For this platform, the codes were compiled with the CUDA 5.0 compilation tools.

The second test platform contains one Nvidia GTX 680 plugged into an Asus P8P67-Deluxe. The GTX 680 GPU is based on the Kepler architecture and embeds a total of 1536 Streaming processors ($8\ SM * 192\ SP$) connected with 2 GB of GDDR5. This series of GPU support dynamic overclocking and adapt their clock frequency from 1006

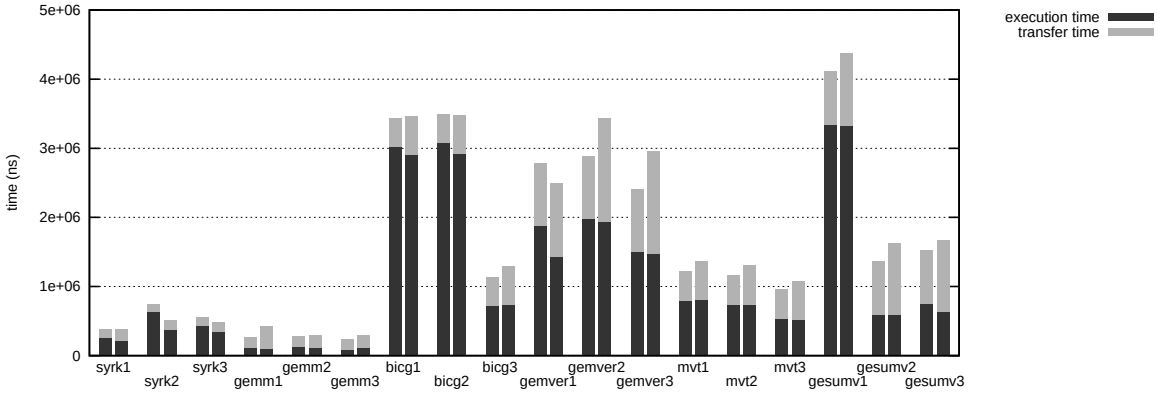


Figure 4.15 – Comparison between prediction (left bar) and real (right bar) execution times on PolyBench with small dataset on GTX680.

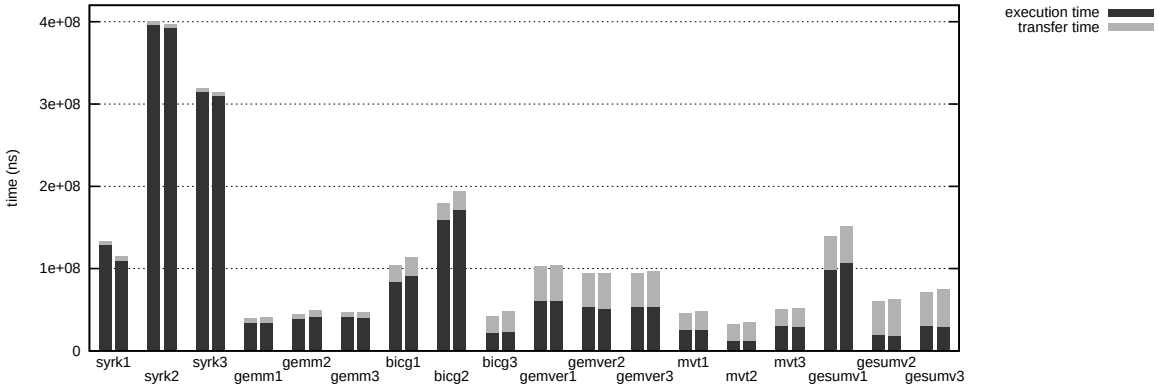


Figure 4.16 – Comparison between prediction (left bar) and real (right bar) execution times on PolyBench with standard dataset on GTX680.

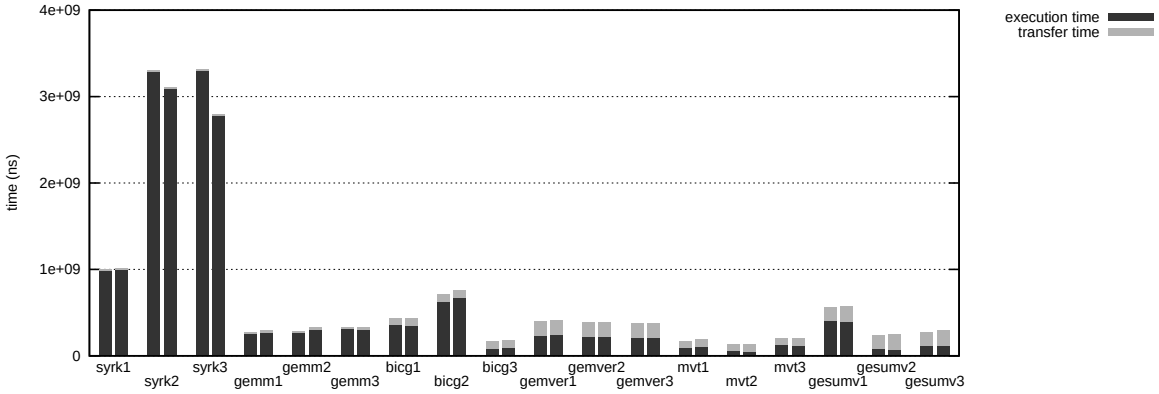


Figure 4.17 – Comparison between prediction (left bar) and real (right bar) execution times on PolyBench with large dataset on GTX680.

to 1058 Mhz depending on the load and TDP. The motherboard provides a PCIe 2.0 x16 bus for connecting the peripherals. The host processor is an Intel core i7-2600 with

4 hyperthreaded cores. For this platform, the codes were compiled with the CUDA 4.0 compilation tools.

For our tests, we used the -O3 optimization flags and compiled the code directly for the target architecture. The codes used a partitioning of 48 KB of shared memory and 16 KB of L1 cache.

4.6.2 Prediction accuracy

The benchmarks are taken from the polyhedral benchmark suite [118]. We tested our predictor on three versions of each of the six codes that we run: *syrk*, *gemm*, *bicg*, *gemmver*, *mvt* and *gesummv*.

Figures 4.12 to 4.17 illustrate the accuracy of our prediction. Left hand side of each couple of bars is the predicted execution time, and the right hand side is the measured execution time. Each bar is divided into two components: a communication time and a kernel execution time.

We notice that our predictor often underestimates the communication time for small datasets (Fig. 4.12 and 4.15), probably due to some overhead of the communication function that was not taken into account by the profiling. However, and even for those very small datasets and short execution times (around $1ms$ total execution time), the estimation is pretty accurate. Notice also that those inaccuracies are similar for all three versions of each code, so the order from the fastest to the slowest version is preserved, which ensures that the best one will be elected by the runtime selection.

For the standard datasets (Fig. 4.13 and 4.16), only one example fails to be predicted accurately: *syrk*, version number 3 on GTX 590. This is due to the size of the dataset being equal to a power of 2 in this example: version 3 has a very different execution time than the neighboring values of dataset sizes that were profiled, and as a consequence, the predicted execution time is inaccurate. This may happen sometimes, but only in very specific cases where the dataset size conflicts with some specific configuration of the GPU. The execution times of the two other versions of *syrk* are predicted accurately.

Predictions are very accurate on GTX 590: within a margin of error of 30% for the small datasets, and less than 4% percent for the standard datasets when excluding the atypical *syrk3* example. The average error on the standard dataset is 2.3%, and it drops to 1.13% when excluding *syrk3*. We made similar experiments on the large dataset and got an average error of less than 2%.

On GTX 680 (Fig. 4.15 to 4.17) the average error for standard and large dataset is of 5.8%. In comparison with the GTX 590 the slight loss of accuracy of the predictions might be due to the GPU dynamic overclocking capability. Regarding the small dataset, the average error is of 15% which is satisfactory considering that these measures are more vulnerable to execution variations. These experiments demonstrate that our method keeps relevant in case of hardware dissimilarities thanks to overall very accurate predictions.

We measured that the runtime prediction overhead does not exceed $15\mu s$ per version, for the considered codes. This ensures that our prediction code has a negligible impact on the code performance for those codes.

Our experiments show that different versions may achieve the best performance

for different dataset sizes: gemver1 is fastest for the small dataset (Fig. 4.12) while gemver3 is the fastest for the standard dataset (Fig. 4.13) on GTX590. As we compare Fig. 4.13 and Fig. 4.16, it is noticeable that some code versions perform differently on the two architectures: bicg1 *vs* bicg3, gemver3 *vs* gemver2, mvt1 *vs* mvt2, gesumv1 *vs* gesumv2, are respectively the fastest on GTX 590 *vs* GTX 680. *syrk1* and *gemm1* are the fastest on both architectures.

4.7 Perspectives and conclusion

We presented a method that is capable to select the most efficient version of a code on the GPU. The selection is based on a very accurate prediction of the kernel transfer and execution times on GPU, based on an offline preliminary profiling. While there are many other methods to tune programs for GPUs, none have demonstrated such an accuracy and portability to our knowledge.

Despite the fact that it is currently written in CUDA we are confident it could easily be transposed on AMD devices, for instance by using the Ocelot libraries. Our future plans include improvements in reducing the profiling duration. The profiling could be done on some predefined configurations and distributed to the end users, who would only need to select the closest one to their architecture. However, this would probably lead to some inaccuracies.

The complexity induced by accurately profiling GPUs required several adaptations and changes to Pradelle et al. method. The prediction mechanism is the cornerstone of the thesis. It brings capabilities to predict execution times, enabling a broad range of opportunities in heterogeneous resource exploitation. The primary objective of the work was to produce accurate predictions. Not only, is it a requirement to reliably select the best version of a code in a multiversioning context, but also to enable heterogeneous computations. The reduction in profiling time, induced by the complexity of the performance mechanism of GPUs could be brought further. For instance, through, partial execution of the iteration domain, when the target code permits: for regular loop nests, in the absence of phases. These elements could be automatically detected by the compiler, in order to produce an adapted profiling code. Also, periodicity detection mechanisms may help to bound profiling. The time prediction method was designed assuming that transfers and kernel calls are synchronous. In extension to the current work, one could consider parallel execution streams (*e.g.* data movements hiding, simultaneous kernel execution). Moreover, current solution requires to generate the versions manually by selecting the size of a block, tile or loop schedule. A compiler may automatically generate a set of versions which performance are guaranteed to be good. Also, the improvement priority should be given to a finer consideration of the statements in a loop nest. The presented methods, characterize an entire loop nest and produce average execution times per iteration. Thus, this principle works well for perfect loop nests and workload balanced problem sizes.

Chapter 5

Heterogeneous computing

The wide availability and efficiency of modern coprocessors sets heterogeneous computing to a common practice. In utopia, every code would benefit from such a system configuration. But lets have feet on the ground ! To achieve maximal performance it is generally required to rely on automation. From a compiler perspective, it is pretty hard to determine what applications one architecture is better at executing at. Current state of the art automatic polyhedral parallelizers (PLUTO, PCCG, etc.) generate code optimized towards one architecture. To get the most of heterogeneous systems, it is mandatory to determine affinities between code and PUs. The characteristics of a code and the workload size are the dominant factors which determine the target architecture to be employed.

Computational intensity, hardware resource constraints (register pressure, amount of memory, amount of functional units, etc.), propensity of code to fit the target architecture (memory access patterns, SIMD, etc.), hardware availability, energy constraints and so on, have to be taken care of. We focus on two specific techniques to address the utilization of heterogeneous resources: CPU vs GPU and CPU + GPU. We demonstrate a dynamic way of running the codes on the best architecture, taking the impact of the execution environment into account. This method consists in running CPU and GPU specific versions of a code on the same dataset and select the fastest version. For this purpose, we propose an early termination technique to stop the execution of codes on CPU and GPU. We formalize an hybrid method relying on the execution time prediction methods on CPU and GPU described in chapter 4. During an offline profiling, several versions of a code are characterized with their target-specific profiling strategy. At runtime, the code selection mechanism transparently selects the code version minimizing the predicted execution times. In a general manner, exploitation of all the computation resources requires to adjust the workload assigned to each PU. In fact, to be efficient, distribution of a calculus requires to equitably share execution times. We demonstrate a transparent scheduler guided by execution times predictions and able, if advantageous, to select the adequate architecture or jointly use multiple processors to perform a computation. The assigned workload is successively refined, until predicted execution times are equivalent on all the participating processors.

Finally, we were also interested in energy savings, by demonstrating that our scheduler can easily be extended with power metrics. The exclusion mechanism is

parametrized by the predicted energy consumption, computed with execution time predictions. To achieve this, we performed coarse-grain power measurements right out of the power socket for one specific code version serving as reference. The measured instant power consumption is then used to estimate the energy consumed over a certain amount of time.

5.1 CPU vs GPU execution: a dynamic approach

5.1.1 Introduction

With the profusion of alternative architectures, choosing the appropriate (co)processor is critical to ensure the execution efficiency of a code. Statically choosing an architecture may lead to performance fluctuations, on a grid server handling several user jobs for instance. In this section, we demonstrate a dynamic architecture selection technique. This method simultaneously executes a code on multiple PUs and cancels the slowest ones. There is no neat way to efficiently terminate running parallel code, nor with OpenMP neither with CUDA.

Performance may be recorded with the actual parameter values, to build a performance history and set task processor affinities. In general, the first run is sacrificed to train a performance model. Also, this method can be used as a basic block to more elaborated systems. For critical code portions, part of the workload may be executed on CPU and GPU simultaneously to discover the fastest architecture. In that case, the bad performing processor execution should be cancelled so that it terminates. The actual best architecture should be chosen for the rest of the computation. The underperforming PU may as well be used for other computations. A dynamic Thread Level Speculation system (TLS), could for example be such a system. Nevertheless, such a system provides information on which architecture to use. Finally, this selection method provides a technical solution to interrupt CUDA kernels and OpenMP parallel code sections. Neither the CUDA API nor the Nvidia proprietary drivers nor the *nouveau* drivers [99] easily expose any interface to quickly interrupt a kernel. On the other hand, forking a process and exiting it is inapplicable as there is a 5 seconds delay, which is unacceptable for fast kernels.

The execution performance of a program may be affected by external dynamic performance factors. In some circumstances, execution might be preferable on another available PU. But not only performance fluctuations are handled by this technique. In general, it is difficult to statically determine the best architecture, as the input dataset may have an influence on performance. As such, this technique is capable of adapting to dynamic constraints and selecting the best architecture for the input dataset. However, this technique adapts best to dynamicity at the price of higher power consumption. A more power efficient technique is presented in section 5.3 with a scheduling technique capable to select and distribute computations.

```
procedure ELECTION(f, e, n)
    r ← atomicAdd(e)
    if r = −1 then
        while n < T − SM ∗ B do
            s ← poll(f)
            if s = 0 then
                trap()
            end if
        end while
    end if
    computation()
    n ← atomicAdd(n)
end procedure
```

Figure 5.1 – GPU kernel termination algorithm with thread election.

5.1.2 CPU vs GPU attempt

Primarily thought as an extension to the version selection framework (Chapter 4), we propose to select the best architecture between CPU and GPU with a "fastest wins" algorithm. To achieve this, the codes must be stopped as soon as one architecture finishes its work. Two control threads simultaneously launch the execution on the CPU and on the GPU. The first control thread that detects the termination of its version is declared the winner and signals the other one to stop. If the fastest code was run on a GPU, it is then allowed to copy the results back to main memory. This is a first attempt to a fully dynamic way to implement a CPU vs GPU mechanism.

5.1.2.1 CPU code

We use PLUTO [25] to generate OpenMP code for the CPU parallel loop nest. But OpenMP does not allow early exit from parallel loops. To circumvent this problem we inserted guards at the outermost sequential loop level that prevent the inner nested loops being executed. The condition in the guard polls a flag in main memory to know whether or not the execution should continue. Through this method it is possible to stop the execution with low overhead.

5.1.2.2 GPU code

Since neither OpenCL nor CUDA API provide a neat way to stop running kernels, we implemented our own method, similar to the one we used on CPU. Modern Nvidia GPUs are capable of directly accessing main memory (DMA). Therefore, it is possible to communicate with GPUs during kernel execution. A flag notifies the end of the execution to the GPU. As one can expect, letting all the threads poll a flag in main memory would cause an important overhead due to the high access latency. We solve this problem thanks to the hardware mecanism to hide access latency. Only one thread takes charge of the flag polling, while the others do the real computations. Once

Table 5.1 – CPU vs GPU execution times

version		syrk	gemm	bicg	gemver	mvt	gesummv
standard	CPU	159.8 ms	125.5 ms	26.1 ms	77.2 ms	26.4 ms	30.3 ms
	GPU	100.0 ms	128.9 ms	40.1 ms	77.7 ms	40.1 ms	80.0 ms
	Winner	GPU	GPU	CPU	CPU	CPU	CPU
large	CPU	910.4 ms	701.4 ms	88.7 ms	305.3 ms	69.7 ms	87.8 ms
	GPU	699.5 ms	712.9 ms	156.8 ms	305.8 ms	164.3 ms	312.5 ms
	Winner	GPU	GPU	CPU	CPU	CPU	CPU

a block is selected by the scheduler it is executed until the end. As mentioned ealier, an exception to that occurs when access latency must be hidden. However, there is no guarantee on the order in which the blocks are executed. An election algorithm, described in Figure 5.1 takes care of choosing the right thread. To achieve this, we increment a global variable set to -1 beforehand, with the *atomicAdd* intrinsic. The first thread that executes the atomic operation sees -1 as the previous value of the variable and is chosen to poll main memory. The other threads continue to the computation and increment another global variable n as they finish. To maintain some load balance the polling stops whenever n reaches $T - SM * B$, T being the total number of threads, SM the number of streaming multiprocessors, B the number of threads per block.

5.1.2.3 Experimentations

Once the best GPU version has been selected, our system runs it simultaneously with a CPU version. Table 5.1 depicts actual execution times for the different GPU and CPU versions. It contains total execution times representing the duration of the codes up to their termination. As mentioned earlier, termination may be activated prematurely: as soon as a code finishes the other concurrent code is asked to stop. Let us illustrate this with an example. For *syrk*, the kernel and transfers complete after 100.0 ms on the GPU. Thus, a minimum of $159.8 - 100.0 = 59.8$ ms were necessary to terminate the CPU code version. For *gemm* the provided times seem to be more advantageous to the CPU, although the GPU won. In this particular case, the quick termination of the CPU code highlighted the data transfer time from the GPU. Indeed, in order to avoid data races the CPU acknowledges the stop request so that results can be fetched back from the GPU. Finally, the GPU wins on two codes, *syrk* and *gemm* for which it achieves good performances.

The delay until the codes actually stop seems high in some cases, but it is not inevitably disabling. In case the CPU wins, it can directly run another code and reuse GPU as it becomes available again. If GPU is the winner, other threads can be launched on the available cores. In this case, the performance hit may promote the GPU.

Still, the overhead shows the limit of artificial techniques to stop the kernels. One of the main problems arise when the hardware is unable hide latencies with ready blocks. We solve this issue and elaborate on a more robust technique in section 5.1.3.

Algorithm 5.1 CPU code shape after instrumentation

```
function CPU Parallel section( )
    tid ← omp_get_thread_num()
    threadTid[tid] ← pthread_self()
    time2go2bed ← setjmp(threadJmp[tid])
    omp_lock_unset(&threadLocks[tid])
    if time2go2bed = 1 then
        goto bed
    end if
    compute_schedule()
    ...
    computation()
    ...
    omp_lock_set(&threadLocks[tid])
    label bed
end function
```

5.1.3 Early termination of CPU and GPU codes

The problem of the strategy exposed in 5.1.2 resides in monopolizing a warp for polling the central memory. The following strategy, avoids the use of zero-copy memory to achieve the same goal.

To avoid race conditions, each PU works on its own copy of the original arrays. On GPU victory, the computed data are transferred to the host, and the original arrays content is overwritten. Otherwise, arrays touched by CPU code are kept as is. The computation is launched on separate OpenMP threads, nested parallelism activated. To automate the code generation procedure, CPU and GPU codes are respectively generated with PLUTO and PPCG. All the code transformations are automatically generated from the source code.

5.1.3.1 CPU code

Cancellation points were introduced in OpenMP 4.0[1]. The programmer can set static cancellation points which terminate a parallel region. In order to avoid active polling, we conjugate the use of signal handlers and long jumps. This has the advantage of highlighting a low overhead on outermost parallel loop nests, as only very few operations are needed.

In that case, mostly non-outermost parallel codes are affected by the termination handling technique exposed. The python scripts transform *parallel for* constructs into a *parallel section* equivalent. Restriction to put a for-loop following the OpenMP pragma is consequently relieved. Nevertheless, the workload is distributed with respect to the original static OpenMP schedule.

On most implementations (including linux GOMP) the OpenMP runtime relies on POSIX threads, which allows to make use of certain traditional *pthread* operations, in

[1]http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf

Algorithm 5.2 GPU host code shape after instrumentation

```
function HOST CODE( )
    cudaMemcpyAsync(..., cudaHostToDevice, stream1)
    if time2go2bed = 1 then
        goto bed
    end if
    ...
    kernel <<< (blk0, blk1), ..., stream1 >>> ()
    cudaStreamSynchronize(stream1)
    if time2go2bed = 1 then
        goto bed
    end if
    ...
    label bed
end function
```

particular signal handling. At the beginning of a parallel code section, the threads are registered with their system-wide thread ID and save their execution context (program counter, stack frame, stack pointer, etc.) in prevision to a long jump, see Algorithm 5.1. When the GPU completes a computation first, it executes a function initiating the thread cancellation. Every CPU computing thread is notified with a POSIX signal which redirects its current execution to a signal handler. The signal handler then performs a long jump back to the end of the parallel section[2]. Thus, the threads are not cancelled, to avoid undefined behaviours, but their execution flow is redirected to the end of the parallel section. Also, a flag is set to indicate the artificial termination of the parallel section. Again, out of a parallel section, the execution flow is redirected to the end of the CPU code function.

However, threads that returned from the computation function must not be forced back in the parallel section. In that case, a long jump would corrupt the stack and result in undefined behaviour. Also, following the standard, OpenMP requires *longjmp* not to violate the entry/exit criteria. Non-parallel sections are thus protected by a lock to forbid this behaviour. As a computing thread enters or leaves the parallel section, it unsets or sets its associated lock. On termination request, if free, the lock is set so that the computing thread cannot leave the function. To avoid deadlocks, the termination function tests the lock beforehand, so that it is only set if it is free.

5.1.3.2 GPU code

To control the execution of CUDA kernels and favour quick reaction, an if condition is injected into the second loop level. Thus, every iteration of the instrumented loop all the active threads poll a flag located in the device off-chip memory. As demonstrated in Fig. 5.2 zero-copy memory latency is quickly increasing as the number of threads accessing memory increases. One-dimensional loop nests are discarded, as they generally evaluate quickly. By default, the flag is set to continue the computation.

[2]http://www.gnu.org/software/libc/manual/html_node/Longjmp-in-Handler.html

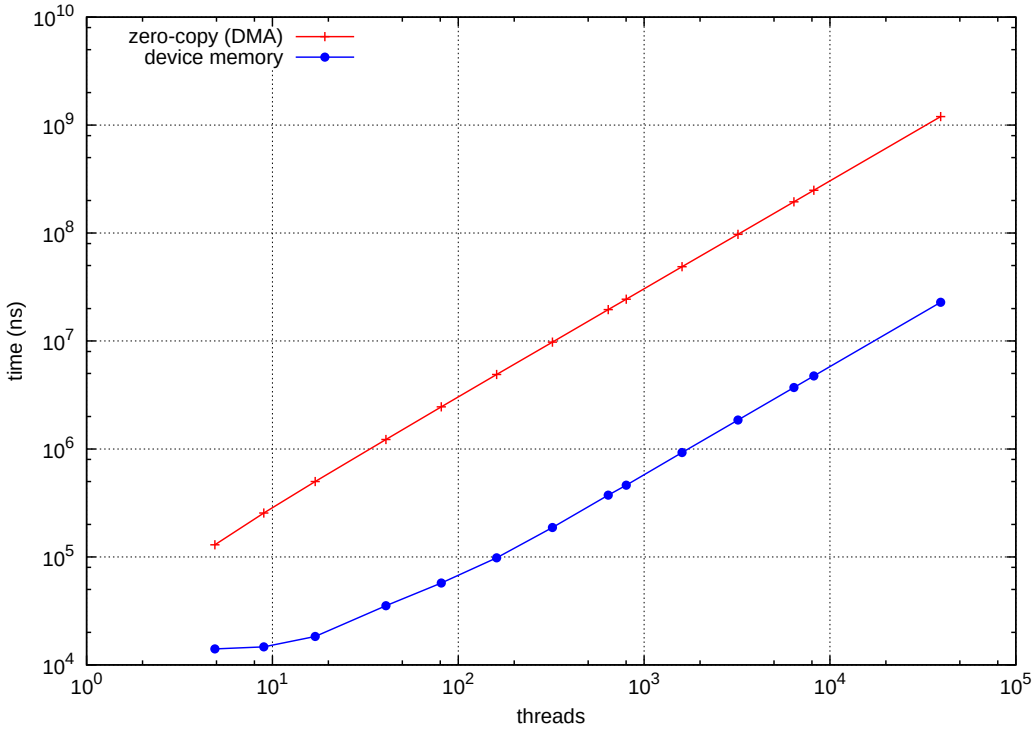


Figure 5.2 – Comparison of execution time between zero-copy (accessing central memory) and memcpy (accessing device off-chip memory) techniques when varying the number of threads.

The polled flag is set volatile, so that it is not optimized into a register. The first threads to detect the value flip of the flag, execute the *trap* PTX instruction, which terminates the whole kernel execution with *unspecified error code*. On kernel termination, a similar flag is checked to redirect the execution to the end of the CUDA host code. Note that, in that case, GPU computed results are not transferred back to host.

Host-device communications are hardly interruptible. To handle this case *memcpy* are consistently followed by a check instruction.

The CUDA runtime implicitly creates a context for each device of a system, called *primary context*. This context enables interactions between CUDA and the device drivers and contains information such as the memory mapping. Noticeably, since CUDA 4.0, a primary context is shared between the host threads for the *default compute mode*. Therefore, running multiple contexts to harness a single device would force the operations to be serialized. Thus, commands are issued to different computing streams in order to be overlapped. For this purpose, the code generated by PPCG is refactored to use appropriate function calls, so that *memcpy* and *kernel* are enqueued in a stream. Eventually, the capacity of a device to overlap computations and communications relies on specific hardware resources. With these requirements, the CPU thread can perform a *memcpy* asynchronously w.r.t. the running CUDA kernels. Thus, on completion, the main thread, dedicated to CPU computation performs a *memcpy* to the GPU computing device to change the value of the flag residing in off-chip memory,

to provoke GPU execution termination.

This method is best effort as there is no guarantee the memcpy will overlap the kernel execution, due to potential resource constrictions.

5.1.4 Experimentations

We evaluate our technique on the PolyBench benchmark (3.2). Optimized and target specific codes are automatically generated by PLUTO (0.9.0) and PPCG (0.1). On PPCG all the code samples where compiled with *min-fusion*, default block and tile size set. On PLUTO, we used one level of tiling, default tile size and use heuristic for fusion (*i.e. smart fusion*). Take note that PPCG takes advantage on macro expansion to make further simplifications. Codes for which compilation failed were set aside. The following results are presented by removing the context initialization and liberation as this inevitably adds up a 500ms overhead on CUDA codes.

Figures 5.3 and 5.4 show execution times normalized to the fastest code. We observe that for the standard dataset *covariance*, *2mm*, *3mm*, *doitgen*, *gemm*, *syr2k*, *syrk*, *reg-detect*, *fdtd-2d*, *jacobi-1d-imper* and *jacobi-2d-imper* run better on GPU. On the opposite, *atax*, *bicg*, *gemver*, *gesummv*, *mvt*, *trmm*, *lu*, *floyd-warshall* and *fdtd-apml* run better on CPU. Interestingly, the CPU was slightly faster than the GPU on *fdtd-2d* with large dataset, while it was the opposite on standard dataset. This is due to the volume of communication, since CPU computational rate outperforms the bandwidth of memory movements. When the GPU wins, the maximum speedup is 10.46x for *2mm*, while the minimum is 1.32x for *fdtd-2d*. When the CPU wins, the maximum speedup is $2,96$x for *floyd-warshall*, while the minimum is 1.02x for *bicg*. In average the speedup is of 1.57x, and 3.88x when respectively, the CPU or the GPU wins. For the large dataset, the speedup to slowest PU surprisingly increases to 1.70x when CPU wins and drops to $3,34$x when GPU wins.

Figures 5.5 and 5.6 depict the overhead implied by the additional instructions and concurrent execution. Overall, the overhead is rather low, barely reaching 10%. To detect a cancellation request, there is no active polling on the CPU side, which induces that the overhead should be really low for the outermost parallel loop nest. In fact the termination request is done through signal handling. Also, checking a flag in GPU global memory is not much impacting performance. Noticeable is the low overhead of *lu* for which 2nd loop dimension is parallel and *trmm*, *floyd-warshall* for which 3rd loop dimension is parallel, when run on the CPU. That demonstrates that our technique to interrupt OpenMP parallel sections has a very small impact on the code performance, even when enclosed by sequential loops. The overhead observable for *atax*, *bicg*, *gemver*, *gesummv*, *mvt* mainly stems from communications between CPU and GPU. On the opposite, GPU codes themselves are not much affected from CPU simultaneous execution, as shown for small codes, such as *jacobi-1d-imper* and *jacobi-2d-imper*. When GPU wins the race, the overhead is of about 4.75% and 2.85% for standard and large dataset respectively. On GPU the incidence of loading a volatile variable inside the parallel loops is fairly low. When CPU wins the race, the overhead is of about 35.36% and 32.49% for standard and large dataset respectively. Note that *fdtd-apml* gets removed from the large dataset results, as there is not enough memory

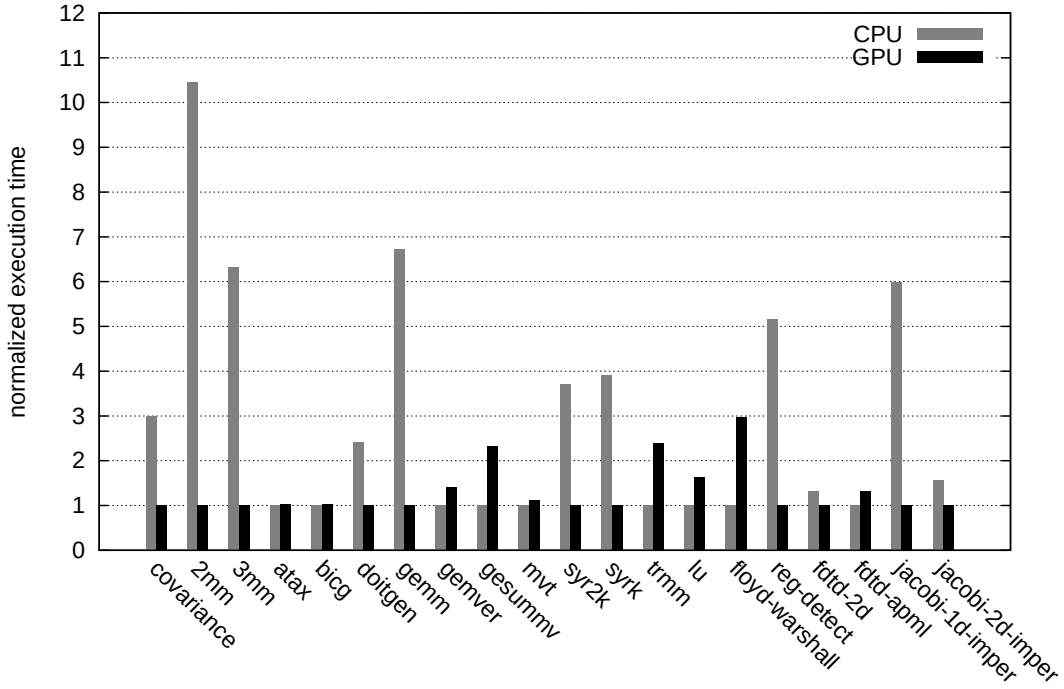


Figure 5.3 – Normalized execution time for the PolyBench standard dataset.

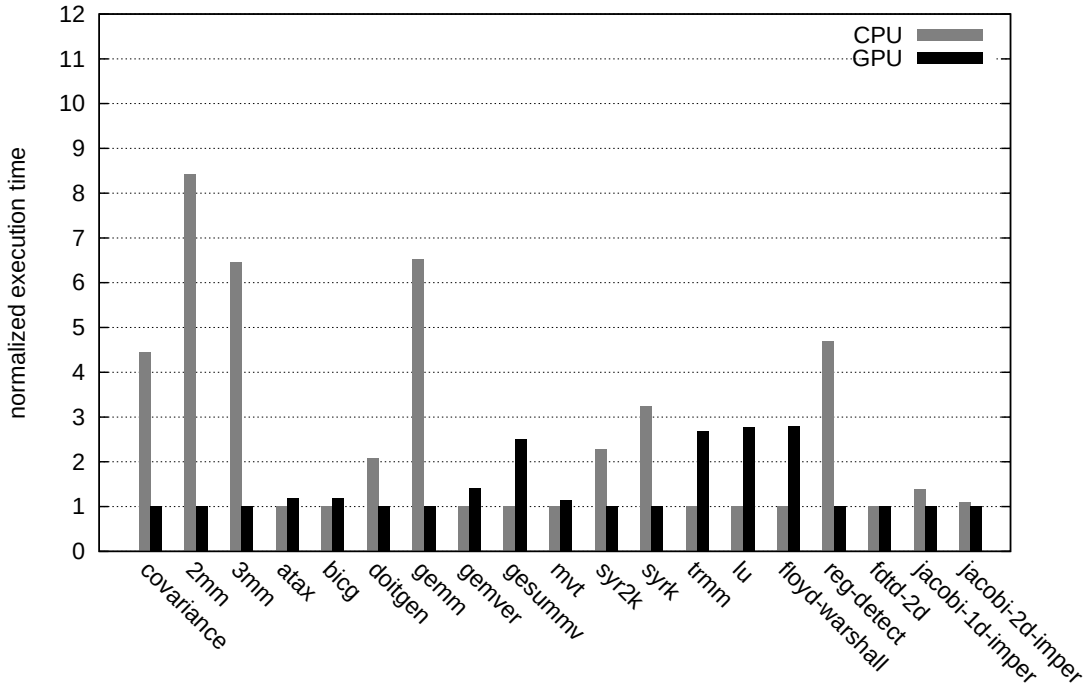


Figure 5.4 – Normalized execution time for the PolyBench large dataset.

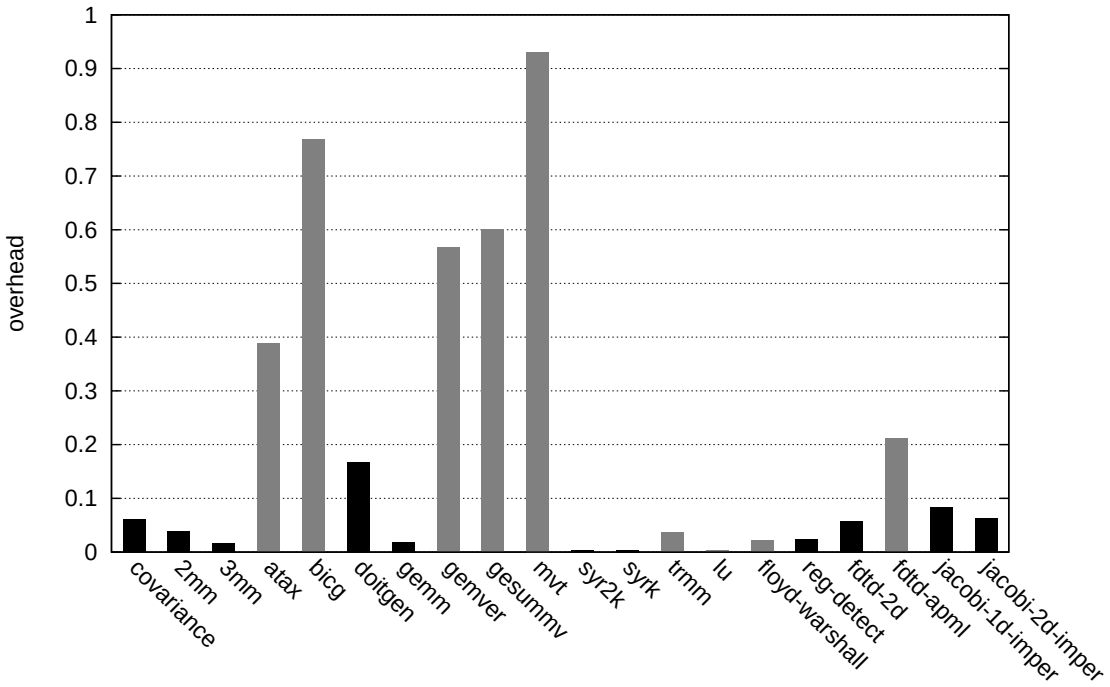


Figure 5.5 – Overhead to the single-PU execution time for the PolyBench standard dataset.

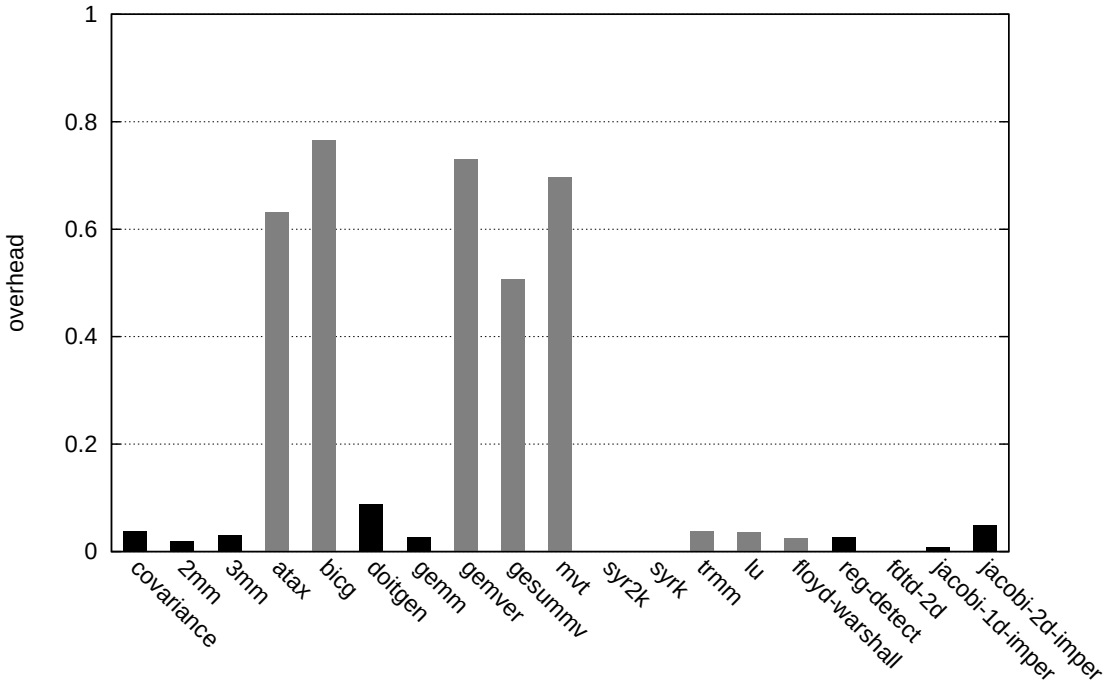


Figure 5.6 – Overhead to the single-PU execution time for the PolyBench large dataset.

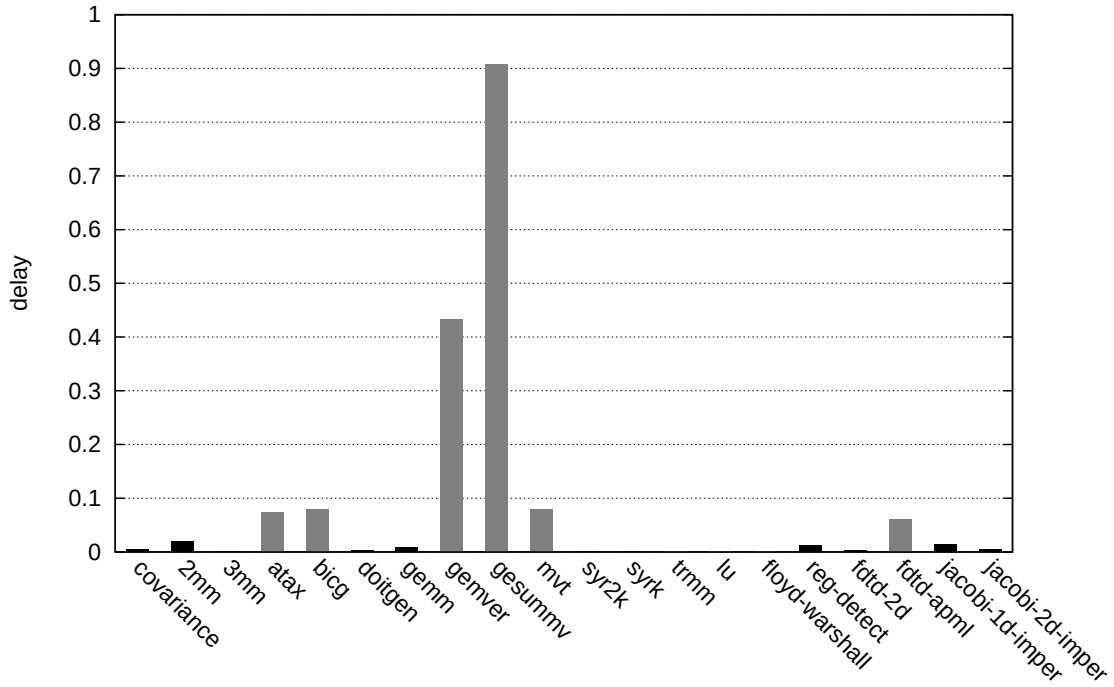


Figure 5.7 – Delay to shut the computation down on the PU that lost the race, for standard dataset.

to fit the data on GPU. The geometric average overhead for standard dataset is of 17.55% and decreases to 5.20% when removing *atax*, *bicg*, *gemver*, *gesummv*, *mvt* for which a first run should disable automatic selection due to overhead. These number respectively drop to 15.96% and 1.95% for the large dataset. For these codes, kernel execution is extremely fast. The bottleneck comes from the communication between CPU and GPU. Although host memory is pinned, there are interactions with CPU running computations, as the transfers get through the cache. Also, CUDA default device scheduling mode was left enabled, as performance of iteratively called kernels was impacted.

The average delay to shut the computation down, shown in Fig. 5.7 and 5.8 is of 6.99% for standard dataset and 3.57% for large dataset, which is acceptable and allows the processor to be reused quickly afterwards. The stop delay, drastically drops from 90.62% to 3.34%, for *gesummv* when comparing the standard to the large dataset figures. This most probably originates from uninterruptible memory transfers, as in the large dataset case, the termination request is probably caught in between transfers or during kernel execution.

5.1.5 Perspective and limitations

We showed a method to quickly terminate OpenMP and CUDA kernels run simultaneously. This method is appropriate on punctually stressed systems, for which one wants to still ensure good performance, by choosing the right processor. Potentially,

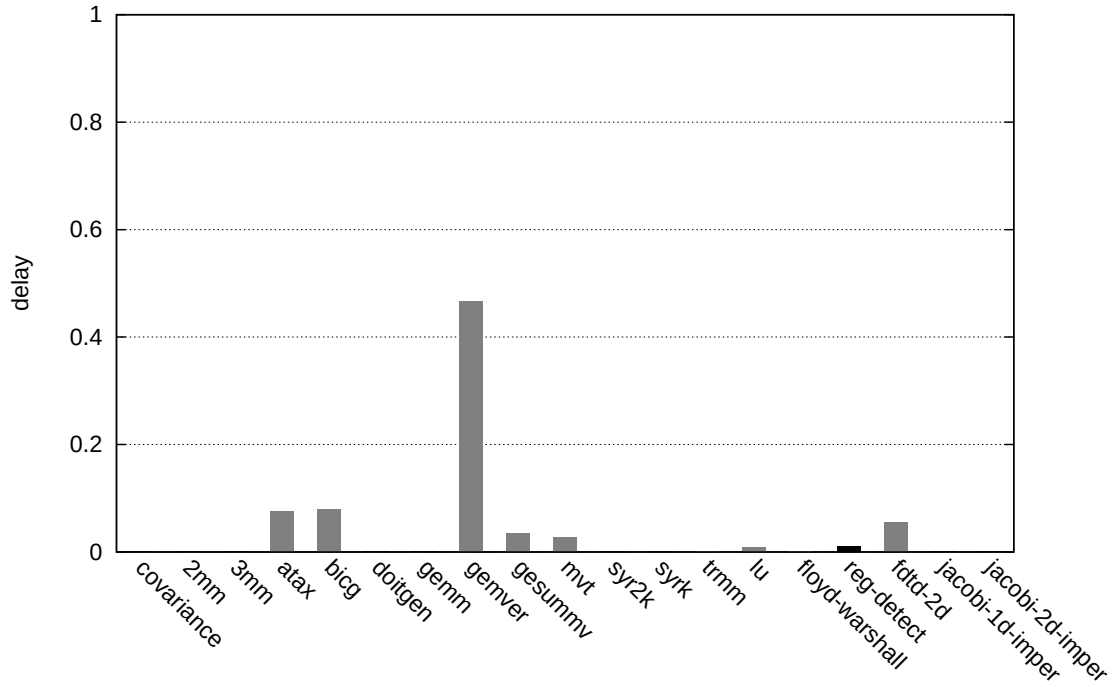


Figure 5.8 – Delay to shut the computation down on the PU that lost the race, for large dataset.

this technique may add up to the stress of the system especially to the memory, when the CPU wins. Also, this technique can be used punctually to train a performance model or be used in a dynamic system as a complement to evaluate CPU and GPU performance. We showed that the overhead was low, especially as there are some instructions that have been added. Also the processors are quickly reusable after a termination request, to perform further computations. Yet, if energy consumption is an issue, this method is not suited, as processors are run simultaneously to perform the exact same computation.

Figure 5.9 depicts an estimation of maximum potential energy consumption. This energy consumption is an estimation which originates from the product of maximal device consumption (measured for a reference code at wall socket) by execution time. This plot gives a broad idea of power consumption, and more accurate results may be achieved by including host code and memory movements energy consumption. Nevertheless, one can see that for 10 codes the slowest PU consumes more than the combined CPU and GPU power consumption in *CPU vs GPU* strategy. Patently, the larger the performance disparity, the better the power consumption in the *CPU vs GPU* strategy.

5.2 CPU vs GPU execution: a hybrid approach

Through our prediction scheme (Chapter 4) it is possible to accurately predict the execution time of a kernel. Also, Pradelle et al. [120] implemented a prediction method,

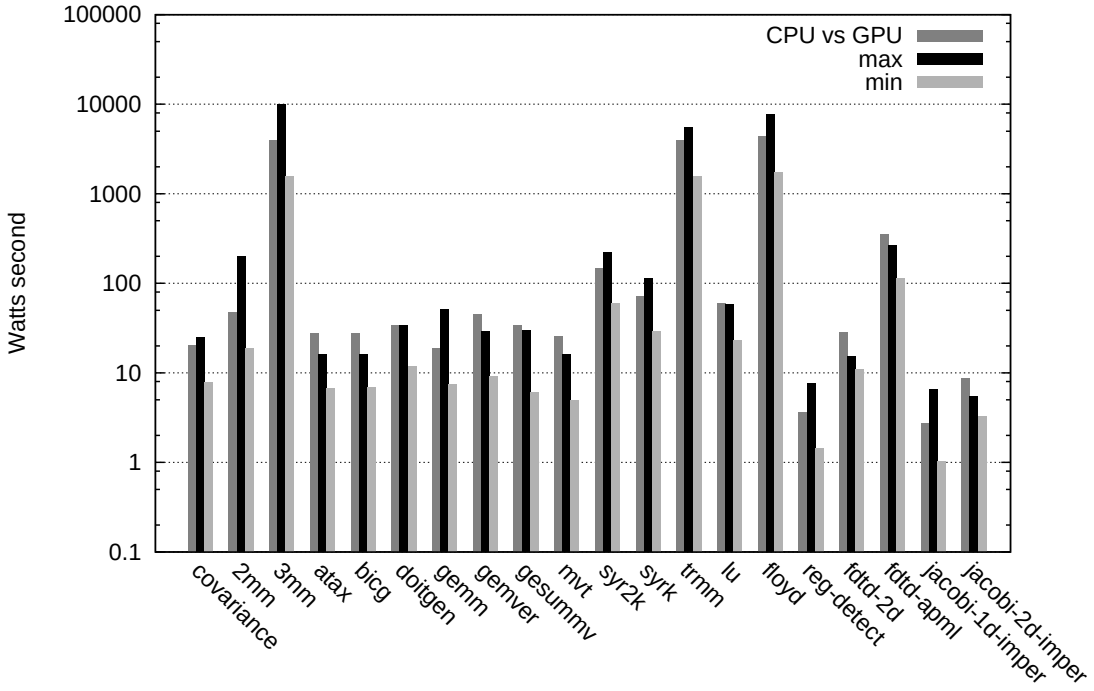


Figure 5.9 – Energy consumption resulting from simultaneously running CPU and GPU codes, for standard dataset. Energy consumption is expressed in Watts second and originates from the product of maximal device consumption (measured for a reference code at wall socket) by execution time.

which gives pretty accurate results on tiled polyhedral codes. In section 5.1 we presented a fully-dynamic method to select the best target architecture. The major drawbacks of such a method is rather clear: energy consumption is high and GPU memory transfers impact the CPU code performance. This leads towards other methods, which predict performance beforehand. To handle multiple architectures, the CPU and GPU prediction methods presented in Chapter 4 may be employed. During code generation, the compiler considers CPU- and GPU-targeted codes as different versions of code. At runtime, the selection mechanism, transparently selects the code version based on predicted execution times; the targeted processor is implied by the underlying generated code.

Each prediction nest is enclosed in a function and evaluated at runtime. A function pointer, which wraps the computation, is assigned to a variable, referencing the best code version to execute. A code sample is shown in Listing 5.1. Such a technique is advantageous: although it is very lightweight at runtime, we may still obtain drastic performance improvements. In fact, this strategy allows to quickly dispatch the computations to the adequate architecture while keeping the multiversioning capability. We expect that despite slight mispredictions the right architecture is chosen. To cope with moderate dynamic behaviour, the ranking tables used for prediction could be adjusted at runtime.

Although this architecture selection method is straightforwardly implemented, nu-

Listing 5.1– Code handling architecture selection and multiversioning

```
{
        mint = MAX_DBL;
        t = predict_cpu_v0(...);
        if(t < mint) {
                ptr = cpu_v0;
                mint = t;
        }
        //...
        predict_gpu_v0(...);
        if(t < mint) {
                ptr = gpu_v0;
                mint = t;
        }
        exec(ptr);
}
```

merous works have tackled this problem and current state of the art has become the distribution of computation over multiple heterogeneous. To that extent, we present a more general method in section 5.3

5.3 CPU + GPU joint execution

5.3.1 Introduction

Efficient exploitation of heterogeneous computing resources is a difficult problem, especially when the processing units (PUs) run different compilation and runtime environments, in addition to different hardware. On multicore CPUs, efficient computing relies on parallelization (using OpenMP for example), cache locality exploitation (loop tiling for example), low-level optimizations (vectorization, instruction reordering, etc.); while exploitation of GPUs requires optimization of memory transfers between host and device, distribution of the computations on a grid of SIMD blocks with limitations on their sizes, explicit memory hierarchy exploitation, global memory accesses coalescing, etc.

Scientific codes are sensitive to their dynamic context. Dynamicity arises for two main reasons: the execution environment variations (*e.g.* hardware characteristics and availability, compiler optimizations) and input data size variation (*e.g.* from a call to a function to another). On the other hand, compilers have to take static decisions to generate the best possible performing code. But, as a result of the dynamic context, they miss many optimization opportunities.

In this work, we aim to address these issues automatically, namely to generate efficient code, that will run on multiple PUs and fully exploit the hardware, in a dynamic context. The most difficult problem is to achieve load balance between heterogeneous PUs. We rely on execution time predictions on each PU, based on a static code gener-

ator, an offline profiling and a runtime prediction and scheduling. Our current development platform targets shared memory cores (one- or multi-socket multicore CPUs) and one or multiple CUDA GPUs.

For achieving execution time prediction and distribution of computations, we target static control parts (SCoPs) of programs, namely polyhedral codes [25]. Computation distribution is made possible by polyhedral dependence analysis and scheduling: polyhedral compilers like PLUTO [25] or PPCG [145] take a SCoP as input and generate nests of parallel and sequential loops. The outermost parallel loops are chunked into controllable size partitions and executed independently on different PUs.

The execution time of a chunk on a given PU is predicted at runtime, at each run of the target code. This computation is based on: (1) the loop size (*i.e.* number of iterations and accessed data), evaluated using polyhedral tools; (2) the average execution time per iteration and per accessed data, based on tables that are generated automatically during profiling and depending on the context of the execution (number of blocks on GPUs, load balance between cores on CPU). Finally, load balance between different PUs is obtained by adjusting the size of the chunks such that their predicted execution times are as close as possible.

The main contribution of this work is an automatic framework for data-parallel workload partitioning and balancing on CPU+GPU systems. The runtime implements a low overhead dynamic scheduler, driven by pre-collected profiling data and by the program parameters. Completely ineffective PUs are automatically eliminated according to the prediction of their performance. Our system integrates a multiversioning mechanism capable to select the best performing combination of code versions, differing by their performance characteristics. Moreover, our implementation combines automatic polyhedral tools to tackle heterogeneous architectures (CPUs and GPUs) transparently.

A typical use-case of this framework is to compile a library, like a BLAS library. Compile and profile time is not an issue, but performance on the machine hardware and adaptivity to different parameters are crucial: the user of this library wants to exploit efficiently all available resources of his machine, in all the calls he will make to the library, possibly using different parameters (like matrix sizes). For compiling such a library in our framework, one would first mark all computationally intensive SCoPs in the source, then call the script that generates the profiling and executable codes and distribute them. The user of the library would run the profiling code on his computer at installation time, and then the code of the library would automatically adapt to the hardware environment (multicore CPUs, number of GPUs, relative performance) and to the dynamic parameters of each call to the library at runtime.

5.3.1.1 Work comparison

We provide a comparison of our method to the one of Boyer et al. [27] in the form of an unrolled example, to show that their method can suffer from prediction errors. We quickly recall the algorithm and the paper original parameters. For training, the system executes an arbitrary number of runs, denoted by R, on each target architecture. The initial chunk represents 7% of the size of the total work and serves as a warm-up. After each execution, the system checks whether the other PUs reached R steps or not. If

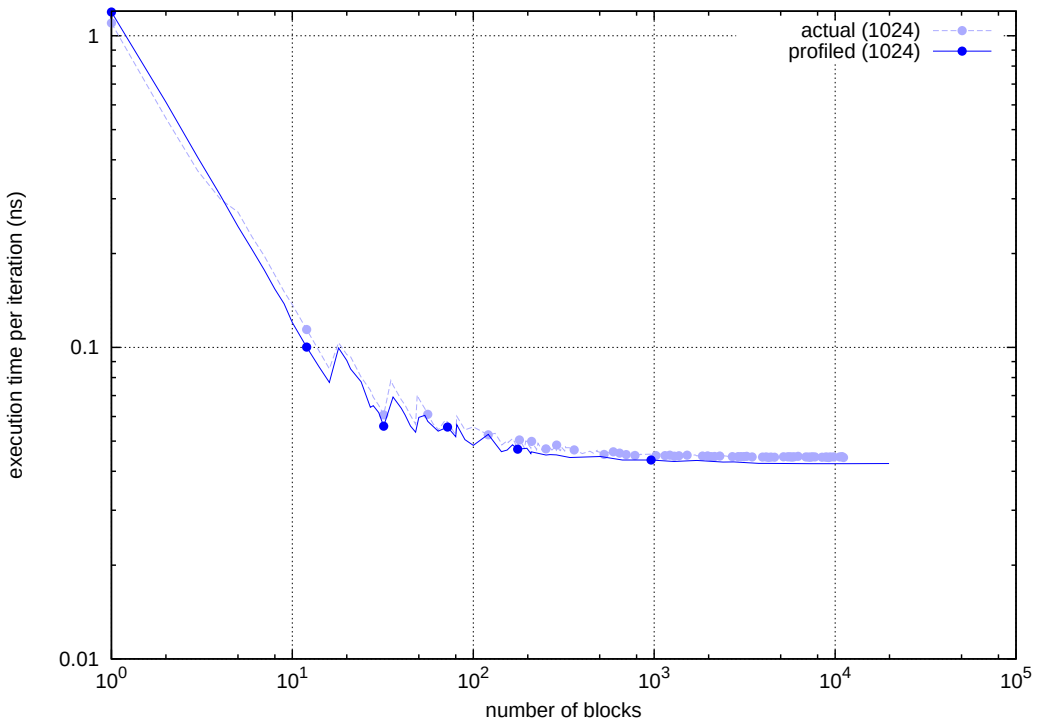


Figure 5.10 – Evaluation of the execution time per iteration for *gemm* for various block sizes and sequential parameter fixed to 1024

it did, the system balances and dispatches the remaining work, based on the last run execution time per work group. Otherwise, the task size is increased by 50% and the process is repeated.

We consider a work group size to be equivalent to the size of a single block of 32×16. We reasonably assume that the algorithm stops at the second step for the studied GPU. Execution times per work group are replaced by equivalent execution time per iteration. For ease of reading, we introduce a fictive PU with id. 1 in the calculations. Also, to keep it simple, only two PUs are considered and transfer times are ignored, as we focus on computation time prediction errors.

In this comparison we use the results obtained in Fig. 5.10 for *gemm*. Parameters NI, NJ and NK are set to $512 \times 512 \times 1024$, with NK the innermost sequential parameter. Let $B_{all} = \lceil 512/32 \rceil \times \lceil 512/16 \rceil = 512$, be the number of blocks required for the computation. The PU 0 reference execution time fetched for $B_{ref0} = \lceil 0.07 \times 1.5 \times B_{all} \rceil = 54$ blocks, equals $E_{ref0} = 0.063616$ ns per iteration. Yet there are $B_{rem} = B_{all} - 2 \times (B_{ref0} + \lceil 0.07 * B_{all} \rceil) - B_{PU1} = 251$ blocks, with $B_{PU1} = B_{ref0} * 1.5 = 81$ the number of blocks computed by PU 1. Let us assume that PU 1 is slightly faster, at $E_{ref1} = 0.061345ns$ per iteration. We compute that PU 0 and PU 1 finish their two steps at $3,227,082$ns and $3,022,797$ns. From PU 0 point of view, the third step of PU 1 is assumed to take $0.061345 * 81 * 512 * 1024 - (3,227,082 - 3,022,797) = 2,400,872$. From this, we find that the last chunk distribution is of 50.92% and 49.08% of the remaining iterations B_{rem}, for PU 0 and PU 1. Actually, PU 0 approximately takes $E_{rem0} = 0.052ns$ per iteration for $B_{rem}/2$, that is to say a misprediction of around 20%.

Assuming that PU 1 execution time is accurately predicted, PU 0 and PU 1 should share 56% and 44% of the remaining chunck. In fact, PU 1 third step and remaining work predictions can be wrong, thus accentuating the load imbalance in Boyer et al.'s method. For this case, the GPU prediction error of our system stays within 5%.

5.3.2 Code generation

In this section we present how we automatically generate OpenMP code for CPU and CUDA code for GPU. A set of python scripts orchestrates the code generation process for its execution on a heterogeneous configuration. To provide source code analysis and modification capabilities, the scripts implement a wrapper on pycparser [23], a C parser written in python. We extended it to handle C for CUDA and a pragma to mark the regions of code of interest. To build the target code, the generator makes extensive use of template files. For specializing the parallel loop nests we rely on PLUTO [25] and PPCG [145], two source-to-source polyhedral compilers. Both compilers generate optimized parallel code from static control loops written in C. PLUTO is focused towards parallelizing sequential codes with OpenMP and is devoted to the application of a broad range of transformations such as loop tiling, loop unrolling, loop fusion or fission, and linear transformations (reversal, skewing, interchange). PPCG produces CUDA host and device codes from sequential loops, using the same transformations. To comply with the CUDA model, multidimensional parallelism is extracted from the tilable bands. The tiles are then mapped to blocks and scanned by the threads. To improve data locality, it generates code to exploit shared memory on CUDA GPUs.

During a first stage, the code is parallelized using the *OpenMP C* backend of PPCG. Artificial parametric loop bounds are injected in the parallel loops to control the iteration domains. This enables the iteration domains of the parallel loops to be cut into chunks. At runtime, each chunk will be assigned a PU and sized to ensure load balance. To this end, PLUTO and PPCG generate specialized versions of the parallel chunks, optimized towards CPU and GPU. As the chunks can be executed in any order, the code semantics is preserved and they can be safely distributed on the available PUs. Our scripts also compute the geometrical bounding box of the accessed arrays to generate minimal data communications between CPU and GPUs. This operation is performed through calls to the isl library [144] via its python bindings islpy [76] and a loop nest polyhedral representation extracted with Pet [146].

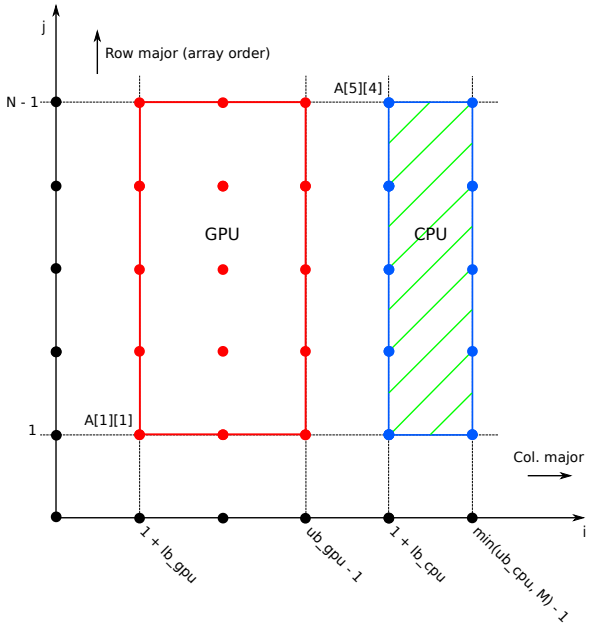
Figure 5.11 shows an example of a chunked parallel loop nest and its associated array access polyhedron. Points with integer coordinates $\{i, j : i >= 1, i < M, j >= 1, j < N, (i, j \in \mathbb{Z})\}$ represent the elements written to array A. Loop chunking infers on the array regions accessed by the PUs. In case of multidimensional arrays, attention must be paid to multiple PUs writing column-wise to chunked array regions, as illustrated in Fig. 5.11b. Indeed, to preserve consistency, it is illegal to call *cudaMemcpy* as presented in Listing 5.3, since it would overwrite the CPU computations. Introduction of loops performing an exact copy is required in this case, as shown in Listing 5.4, but may lead to performance issues.

For iteratively called parallel codes, the amount of data to transfer is adjusted so that we avoid redundant copies. Specifically, the scripts determine which array

Listing (5.2)Chuncked loop nest

```
forall(i = 1 + lb; i < min(ub, M); ++i) {
        for(j = 1; j < N; ++j) {
                A[j][i] = ...;
        }
}
```

(a) Example of a chunked parallel code writing to array A.



(b) A parametric write accesses bounding box.

Figure 5.11 – Illustration of the mechanisms required for communication.

elements have been modified outside of the parallel code and need to be transferred to the GPU. For the remaining transfers, only missing data, due to repartitioning, are sent to the GPU by checking the partitioning parameters. All in all, GPU results are systematically copied back to the CPU memory.

One single version of a code may not perform well under all circumstances. Multiversioning is an adaptive technique consisting in generating semantically equivalent versions of a code, differing by their performance characteristics. The versions are built by PPCG and PLUTO, launched with version-specific arguments, such as the tile size, tiling level, block size and schedule. The scripts generate all the version combinations by successively assigning the parallel and prediction code function pointers to internal

Listing 5.3– Standard memcpy generated for row-major

```
cudaMemcpy(&A[1][0], &dev_A[1][0],
                (N-2)*(M-1)*sizeof(*A), D2H);
```

Listing 5.4– Handling special memcpy case depending on chuncked dimension

```
for(c0 = 1; c0 < N; ++c0) {
    cudaMemcpy(&A[c0][lb_gpu + 1],
                &dev_A[c0][lb_gpu + 1],
                ((ub_gpu-1)-(lb_gpu+1)+1)*(sizeof(*A)), D2H);
}
```

structures. For each combination it places a call to the scheduler and the selection function. The combinations are scheduled at runtime, supposedly the best is executed on CPU+GPU. The snippet in Listing 5.5 shows the code generated for *gemm*.

Listing 5.5– Code handling multiversioning

```
{
    tdt0->xsize = dt0->xsize = M - 1;
    min_etime = DBL_MAX;
    reset(tdt0->pus, GPU, gpu_etime_0_0, gpu_call_0_0, domsz_0);
    reset(tdt0->pus, CPU, cpu_etime_0_0, cpu_call_0_0, domsz_0);
    etime = cuschedule(dt0_tmp);
    mv_select(etime, &min_etime, dt0, tdt0);
    reset(tdt0->pus, GPU, gpu_etime_0_0, gpu_call_0_0, domsz_0);
    reset(tdt0->pus, CPU, cpu_etime_0_1, cpu_call_0_1, domsz_0);
    etime = cuschedule(dt0_tmp);
    mv_select(etime, &min_etime, dt0, tdt0);
    //...
}
```

The *reset* function reassigns execution time prediction and kernel call functions for each combination. *mv_select* is in charge of setting *dt*0 if a better combination is found. Each combination partitioning is computed by a call to the scheduler *cuschedule*.

5.3.3 CPU+GPU Runtime

5.3.3.1 Scheduler

The scheduler implements a work sharing mechanism similar to the OpenMP static scheduling strategy. The quantity of work of each PU, controlled by the chunk size, is determined before the execution of the loop nest.

The scheduler relies on the execution time predictions to distribute iterations to the PUs. Let $T_i = t_i \times Q(P_i) + C(P_i)$ be the chunk i predicted duration, where $0 \leq i < n$, n being the number of PUs. Function $Q(P_i)$, generated with the barvinok counting library [147], computes the number of iterations of the union of the statements iteration domains: it is a symbolic piecewise quasipolynomial instantiated with the parameters values P_i at runtime. t_i is the execution time per iteration. Function $C(P_i)$ is in charge of estimating the data transfer time on GPUs; for CPUs it returns 0.

For a given parallel loop nest, the load balance problem can be expressed as an equality between the chunks durations: $T_0 \approx T_1 \approx ... \approx T_{n-1}$. The upper parallel

loop bound of each chunk i is the lower bound of chunk $i+1$. The execution time per iteration t_i of each chunk fluctuates non-linearly depending on the chunk size as described in Chapter 4. As a consequence, there is no direct method to compute the chunk sizes, but this optimization problem requires iterative refinements.

Through problem reformulation, achieving load balance comes down to make T_i tend to T_{all}/n, where T_{all} is the sum of the PUs execution times: $T_{all} = \sum_{i=0}^{n-1} T_i$. We implemented a low overhead iterative algorithm in three steps: *initialization*, *refinement* and *partitioning*. The refinement and partitioning stages are repeated until convergence is reached, or a maximum of 15 steps is attained. In the initialization phase, the iterations of the chunked loops are equally distributed between the PUs. No preliminary assumptions can be made concerning the chunks execution times.

The refining stage starts by computing each chunk execution time T_i and their sum T_{all}. Each chunk execution time proportion $R_i = T_i/T_{all}$ must tend to $o = 1/n$ to achieve load balance. Note that an optimal predicted load balance is obtained for $R_i = o$ for all i. Each chunk size is then ajusted by multiplying it by o/R_i, to get closer to optimal load balance. However, these adjustments are computed independently for each chunk, and this leads to situations where the sum of the chunk sizes is not equal to the total number of iterations. Thus, the partitioning phase normalizes the chunk sizes so that all iterations of the chunked loop are processed. Iterations eliminated by integer rounding are assigned to an arbitrary PU (the CPU by default in our current implementation).

To get rid of very inefficient PUs faster, a chunk smaller than x times the biggest chunk is eliminated: $x = 10\%$ by default in our implementation. It can be increased if energy consumption is an issue: in that case one will want to eliminate inefficient PUs faster.

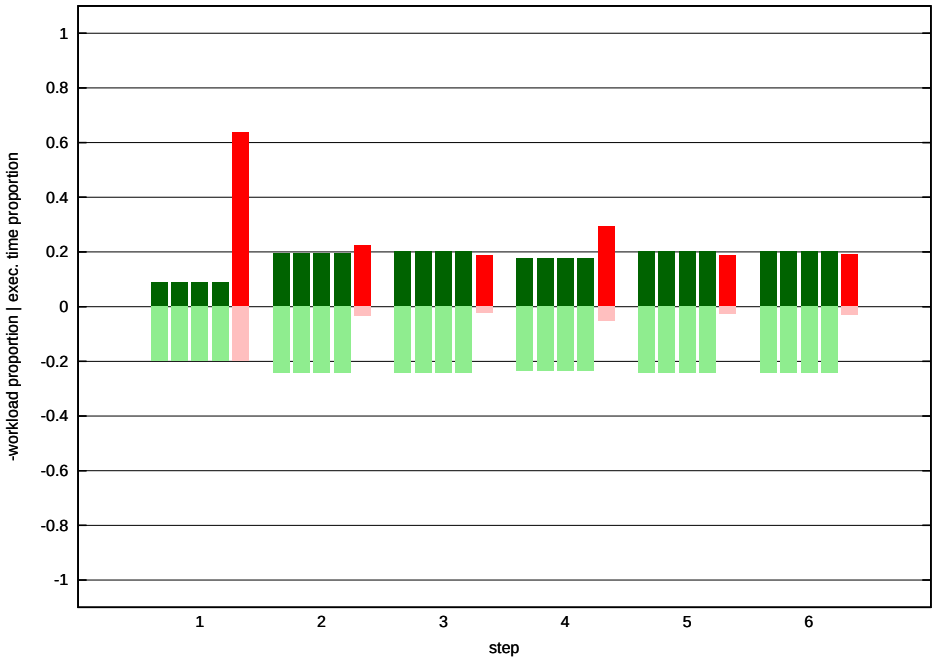
The full algorithm is presented in Alg. 5.3, and Fig. 5.12 shows two typical examples of the scheduler steps. Two PUs are considered: 1 GPU (on the left of each couple of bars) + 1 CPU (on the right). The blue bars represent the size of the iteration domains of each chunk ($Q(P_i)$), and the red bar the corresponding execution times (T_i). In Fig. 5.12a, the GPU is assigned much more iterations for approximately the same execution time than the CPU, in 6 steps of the scheduler. In Fig. 5.12b, the GPU is so inefficient (more than 10 times slower than the CPU) that it is eliminated at the first step.

5.3.3.2 Dispatcher

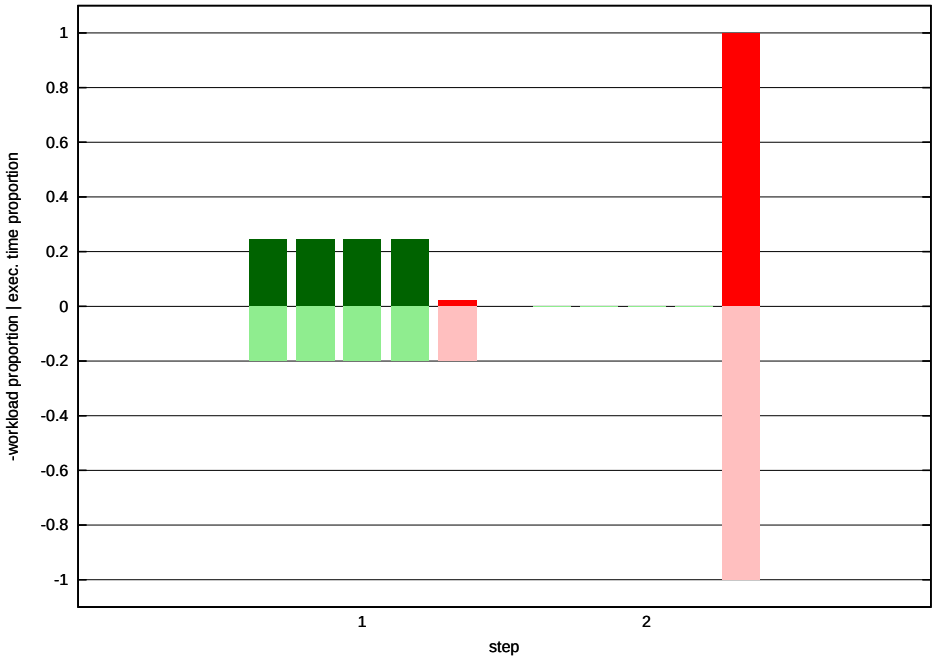
The dispatcher is in charge of launching the codes on the different PUs. Each PU is assigned a thread using OpenMP parallel sections. Before launching the computation, a device initialization function is called. On CPUs it sets the number of threads required for the computation and activates nested parallelism. For GPUs, it selects the device and modifies the CUDA device scheduling policy. Indeed, we observed that the scheduling policy has an impact on the CPU threads performance. By default it will busy-wait if enough processing resources are available and yield the threads otherwise. To get rid of any overhead, we chose another strategy which blocks the polling threads until the device finishes its work (*i.e.* blocking synchronization). Due to device initialization purposes, the first called CUDA function (*e.g.* cudaMalloc(...))

Algorithm 5.3 Scheduler algorithm

```
#step 1: initialize to equal distribution
chnk_size ← (ub − lb)/num_pu
for i ← 0 to num_PU − 1 do
    PUs[i].lb ← i ∗ chnk_size
    PUs[i].ub ← PUs[i].lb + chnk_size
end for

#step 2: refine
for s ← 0 to MAX_STEPS do
    time ← 0.
    for i ← 0 to num_PU − 1 do
        PUs[i].size = PUs[i].ub − PUs[i].lb
        if PUs[i].size ≠ 0 then
            PUs[i].time_val = PUs[i].time(PUs[i].lb, PUs[i].ub)
            time ← time + PUs[i].time_val
        end if
    end for
    for i ← 0 to num_PU − 1 do
        if PUs[i].time_val ≠ 0 then
            adjst = time/(num_PU ∗ PUs[i].time_val)
            PUs[i].size = PUs[i].size ∗ adjst
        end if
    end for

    #normalize the chunk bounds
    (PUs, max_card) ← normalize(PUs)
    for i ← 0 to num_PU − 1 do
        if PU.cardPU/max_card < 0.1 then
            PUs ← eliminate(PUs, i)
        end if
    end for
    reg ← register_schedule(PUs)
end for

#step 3: select schedule
PUs ← select_schedule(reg)
```



(a) gemm



(b) gesummv

Figure 5.12 – Scheduler algorithm steps for *gemm* and *gesummv*. GPU is eliminated after first step in *gesummv*.

consumes more time. To avoid noise in the measurements, we introduced a fake runtime call. At the end of the computation, the threads are synchronized using a barrier. All data movements are carried by the thread handling each PU.

5.3.4 Evaluation

5.3.4.1 Benchmarks

The test platform is composed of two Asus GTX 590 plugged into an Asus P8P67-Pro motherboard. Each GTX 590 card is composed of two Fermi GPUs sharing 3 GB of GDDR5. Each graphics processor on the GTX 590 embeds a total of 512 Streaming Processors[3] (16 $SM \times 32$ SP). The motherboard provides a PCIe 2.0 x16 bus for connecting the peripherals. The two graphics cards individually support PCIe x16 and share half of the bus width (x8) in our configuration. The host processor is an Intel core i7-2700 (Sandy Bridge) with 4 hyperthreaded cores for which we enabled dynamic overclocking.

The benchmark programs that we run are taken from the Polyhedral Benchmark suite [118]. We used the *extra-large* dataset size by default, reducing it on some of the tested programs so that they fit the GPU memory. We did not consider some programs because they are very inefficient on GPU: the CPU version is much faster in any case, and there is no point in trying to exploit a GPU version of them. We did however include some benchmarks that fall in this category in our experiments (*gesummv*, *mvt* and *gemver*). All loop nests of depth 1 are ignored by our framework.

We compiled the benchmarks using gcc version 4.4.6 with *-O3 -march=native* optimization flags. On GPUs, the codes were compiled with the CUDA 5.5 compilation tools. The GPU on-chip memory partitioning was set to 48 KB of shared memory and 16 KB of L1 cache. In our first experiment presented in Fig. 5, only one code version was generated. To generate CUDA code, the minimum loop nest fuse flag was provided to PPCG and automatic cache management code generation was enabled. The CUDA block and tile sizes have been set to the default provided by PPCG. Communications between host and device are handled with synchronous non-pinned memory copies. Similarly, we run PLUTO with the default parameters, and disabled tiling for *mvt*, *gemver* and *gesummv* as these tiled CPU codes are strongly affected by performance fluctuations. For the multiversioning experiments presented in Fig. 5.15 and following, we generated CUDA codes with different couple *(block size, tile size)* respectively for the (c1, c2, c3), (c4, c5, c6), (c7, c8, c9) combinations. CPU versions were generated with one level of tiling, of size 32, 64 and 128, respectively for the (c1, c4, c7), (c2, c5, c8), (c3, c6, c9) combinations. Table 5.2 and 5.3 describe all the versions used in this experimentation. We averaged all measurements on five runs.

Figure 5.13 depicts the speedup obtained by using different combinations of PUs compared to the execution time on CPU alone or on GPU alone. Our system achieves a maximum speedup of 20x for *gemm* and a speedup of 7x on average comparing the best and worst execution times. These results show that *gemm*, *2mm*, *3mm*, *syrk*, *syr2k* (the five on the left of Fig. 5.13) better suit the GPU while *doitgen*, *gesummv*, *mvt*, *gemver* better suit the CPU. Note that *doitgen* better suits the CPU because of a lower computation time on CPU than on GPU, and not because of the data transfer times. It is interesting to notice that combined CPU+GPU execution provide noticeable benefits for three benchmarks (*syr2k*, *doitgen* and *gemver*). When the GPU version is faster, the average speedup of our system on CPU plus 4 GPUs against 1

[3]SM: Streaming Multiprocessors, SP: Streaming Processors

	Versions *(block size, tile size)*		
code	(c1, c2, c3)	(c4, c5, c6)	(c7, c8, c9)
gemm 2mm 3mm syrk	$(32 \times 16, 32 \times 16)$	$(32 \times 16, 64 \times 64)$	$(32 \times 32, 32 \times 32)$
syr2k	$(32 \times 16, 32 \times 16)$	$(32 \times 16, 64 \times 64)$	$(16 \times 16, 16 \times 16)$
doitgen	$(32 \times 16, 32 \times 16)$	$(16 \times 16, 16 \times 16)$	$(32 \times 32, 32 \times 32)$
gesummv mvt	$(512, 512)$	$(256, 256)$	$(128, 128)$
gemver	$(64, 64)$	$(32, 64)$	$(128, 128)$

Table 5.2 – GPU code versions description

	Versions tile size		
code	(c1, c4, c7)	(c2, c5, c8)	(c3, c6, c9)
gemm 2mm 3mm syrk syr2k doitgen	32	64	128
gesummv mvt gemver	untiled	untiled	untiled

Table 5.3 – CPU code versions description

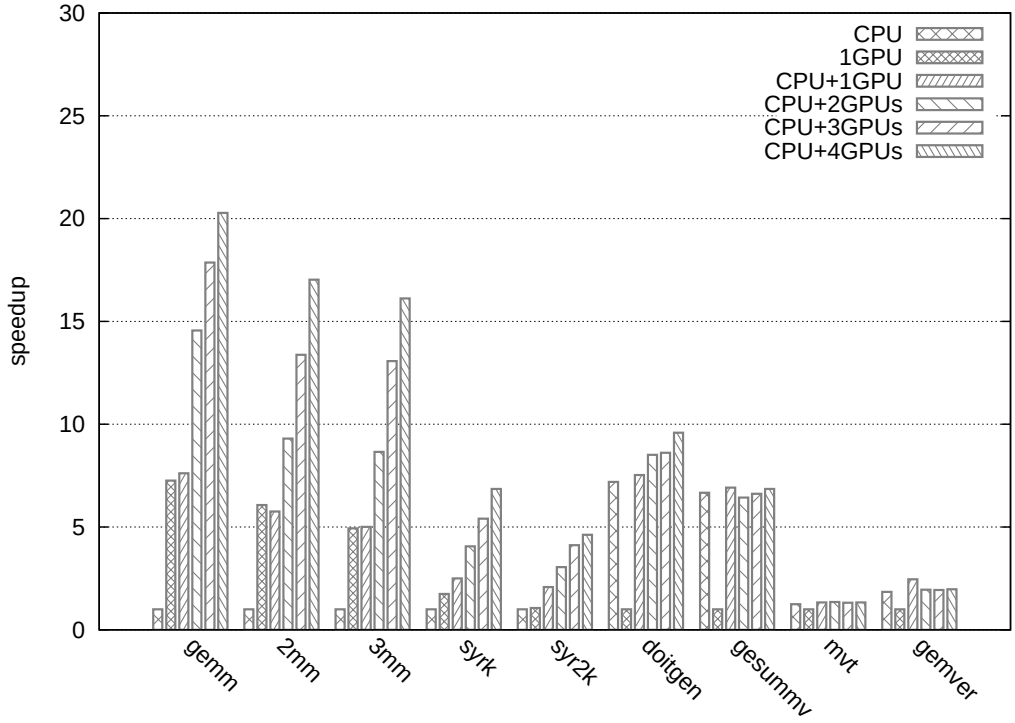


Figure 5.13 – Speedup to execution time on CPU or GPU alone.

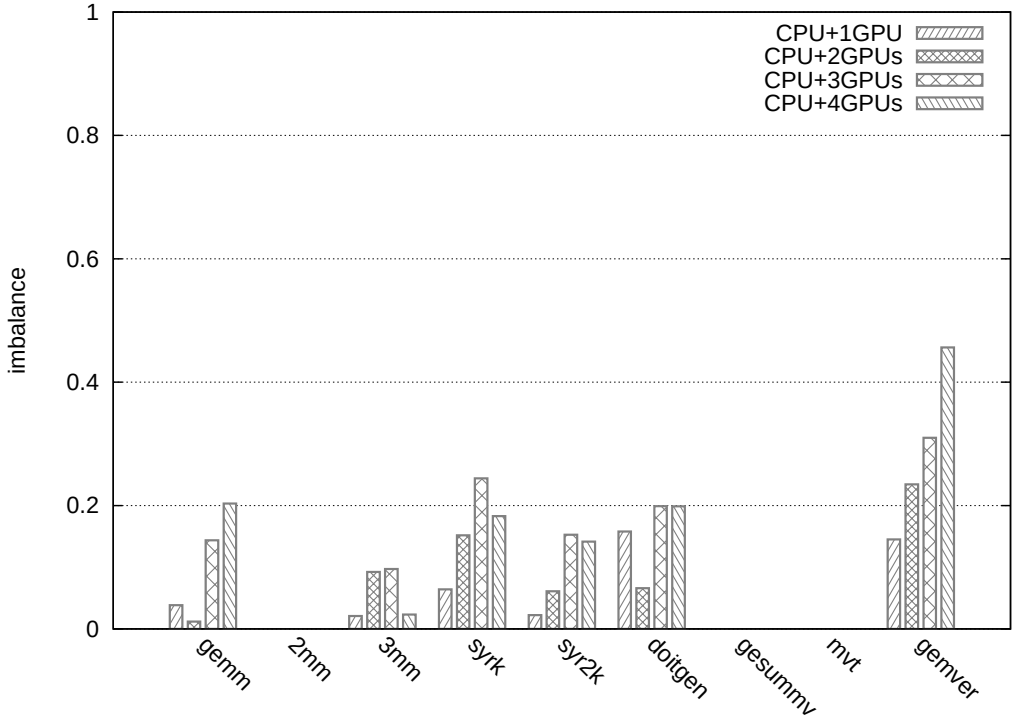


Figure 5.14 – Execution time imbalance ratio for several combinations of PUs.

GPU, is 3.4x and is greater than 4x for *syr2k*; for the programs where the CPU version is faster, the average speedup on CPU plus 4 GPUs against 1 CPU is negligible, as in these case GPUs were generally unused. Also notice that, apart from *2mm* (for which CPU was unused), the 1 CPU + 1 GPU version is always faster to the CPU alone and GPU alone versions. Figure 5.14 shows the imbalance ratio between the total longest and shortest execution times of the different PUs. In *2mm* the CPU was eliminated, and in *gesummv* and *mvt* the GPU was eliminated[4]. Figure 5.14 shows an average of 12% load imbalance. The imbalance is mostly due to prediction inaccuracies rather than bad scheduling decisions. In fact, the sum of the absolute prediction errors drives imbalance. In particular, this affects fast codes with shallow loop nests, such as *gemver* for which the imbalance peaks at 45% for CPU+4GPUs. These codes are strongly affected by multiple running GPUs.

Gesummv, *mvt* and *gemver* are also noticeable due to the elimination of the GPU for certain loop nests computation. For *gesummv* and *mvt* the whole computation is run by the CPU. This happens when the ratio between the communication and the computation times is too high. Also, reducing the problem size may eliminate non-necessary PUs. Remark that there are performance interactions between CPU and GPU, especially on the host code side as memory transfers get through the CPU caches. As an example, the spinning CUDA runtime scheduling policy, set by default, impacts CPU performance by 30% on *gemm* with 4 GPUs running. The opposite effect of using blocking scheduling is that small codes repeatedly executed tend to run slower.

Our system overhead (including all prediction and scheduling calls) is low: it caps at $2ms$ for *doitgen*, that is to say 0.02% of the execution time. It tends to average below $1ms$ for most of the codes, which is a reasonable figure for codes that are suited to run on GPUs (executing for more than a second).

5.3.5 Multiversioning

Our framework is able to generate multiple versions of the CPU and GPU codes and to select the best performing combination at runtime. The scheduler is called for each combination and returns its predicted execution time. The runtime selects the scheduled combination of versions which minimizes the execution time. As the number of combinations grows exponentially, we limited our experiments to 3 versions per PU (9 combinations). Note that for *syr2k*, the maximum scheduler overhead was of $450 * 9 = 3600\mu s$, that is to say less than 0.01% of the execution time. The following description will focus on *gemm* and *syr2k*. Figures 5.15, 5.17, 5.21, 5.23, 5.25, 5.27, 5.29, 5.31 and 5.33 show speedups to the slowest combination of versions. The concurring imbalance is depicted in Fig. 5.16, 5.18, 5.22, 5.24, 5.26, 5.28, 5.30, 5.32 and 5.34. Usefulness of multiversioning on GPU is emphasized by the performance variations of the *GPU only* executions in the (c1), (c4), (c7) combinations in Fig. 5.15 and 5.17. The (all) bars refer to the final combination selected by our runtime system. At best, it was able to achieve a 1.53x speedup for *gemm* and a 3.46x speedup for *syr2k* against the slowest combination. For *gemm*, it is noticeable that best performance were obtained when CPU was evicted. On the opposite, combinations benefit from the use of CPU

[4]Figure 5.13 shows small variations of the CPU+nGPU versions due to measurements inaccuracies.

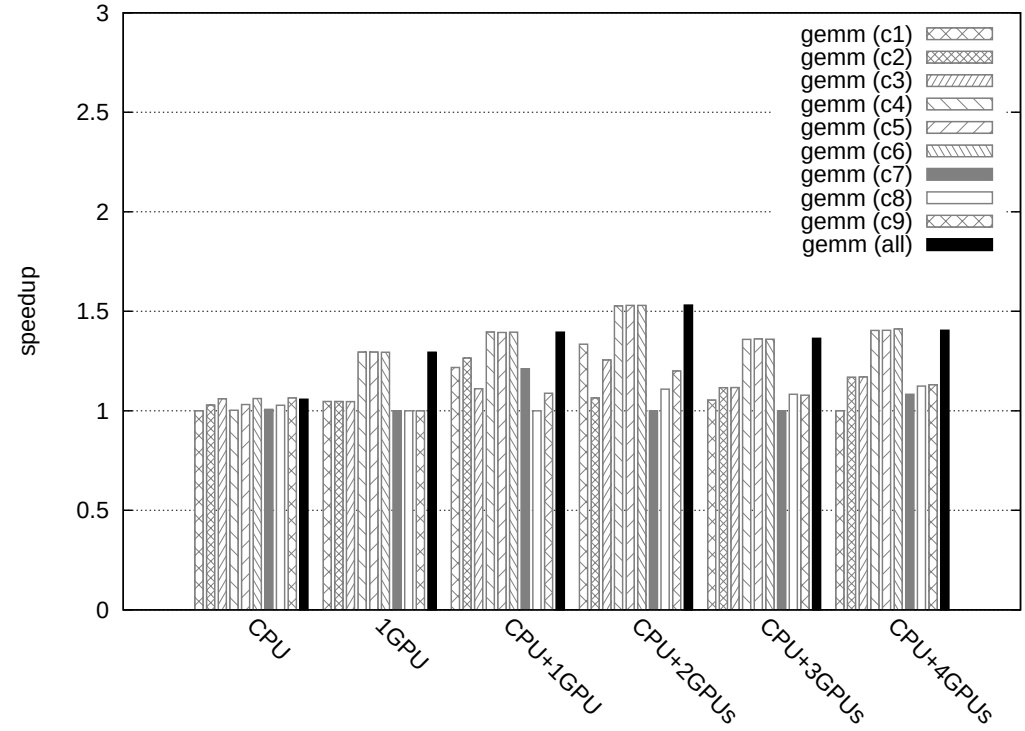


Figure 5.15 – Speedup to execution time of slowest code version combination for *gemm*.

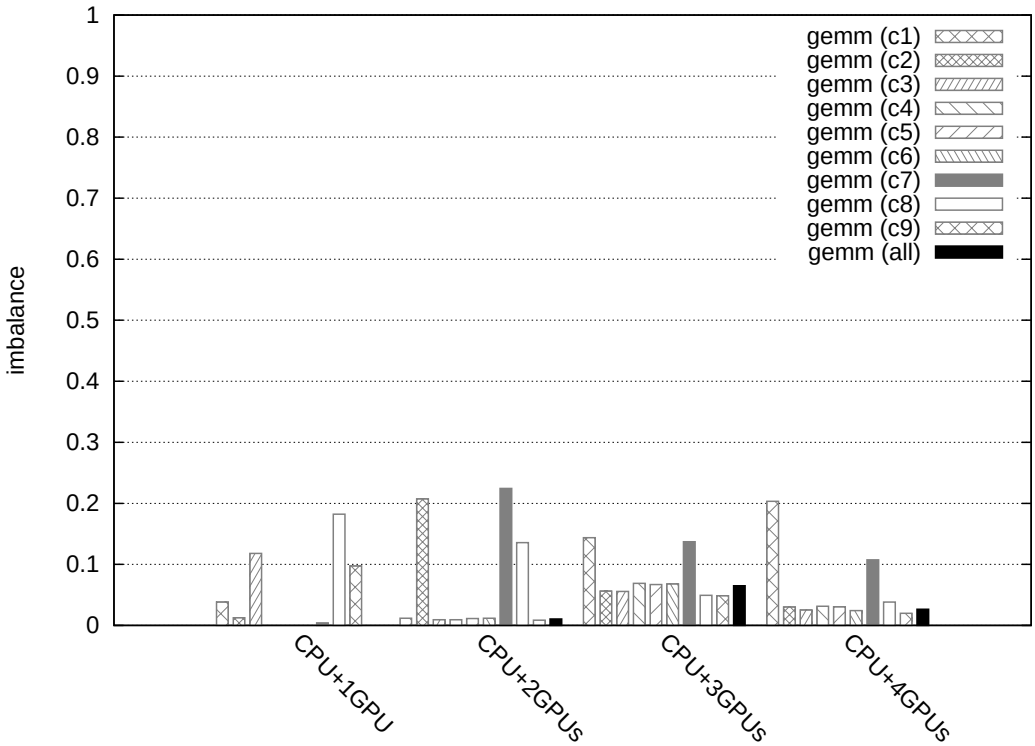


Figure 5.16 – Execution time imbalance ratio for several combination of code versions for *gemm*.

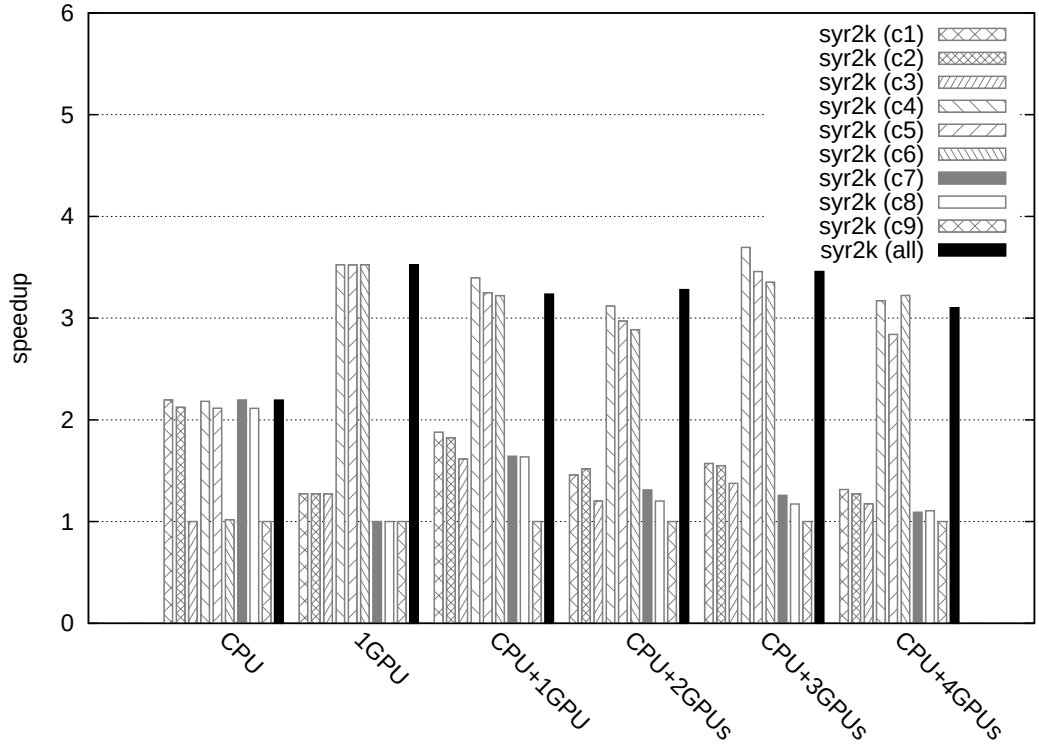


Figure 5.17 – Speedup to execution time of slowest code version combination for *syr2k*.

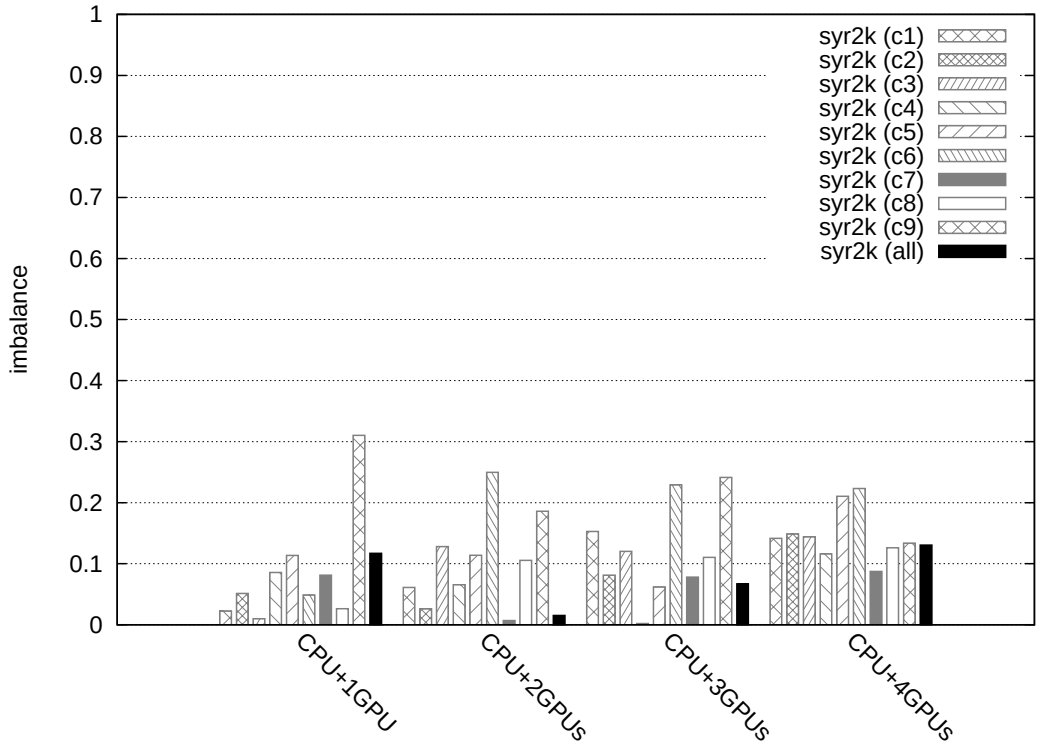


Figure 5.18 – Execution time imbalance ratio for several combination of code versions for *syr2k*.

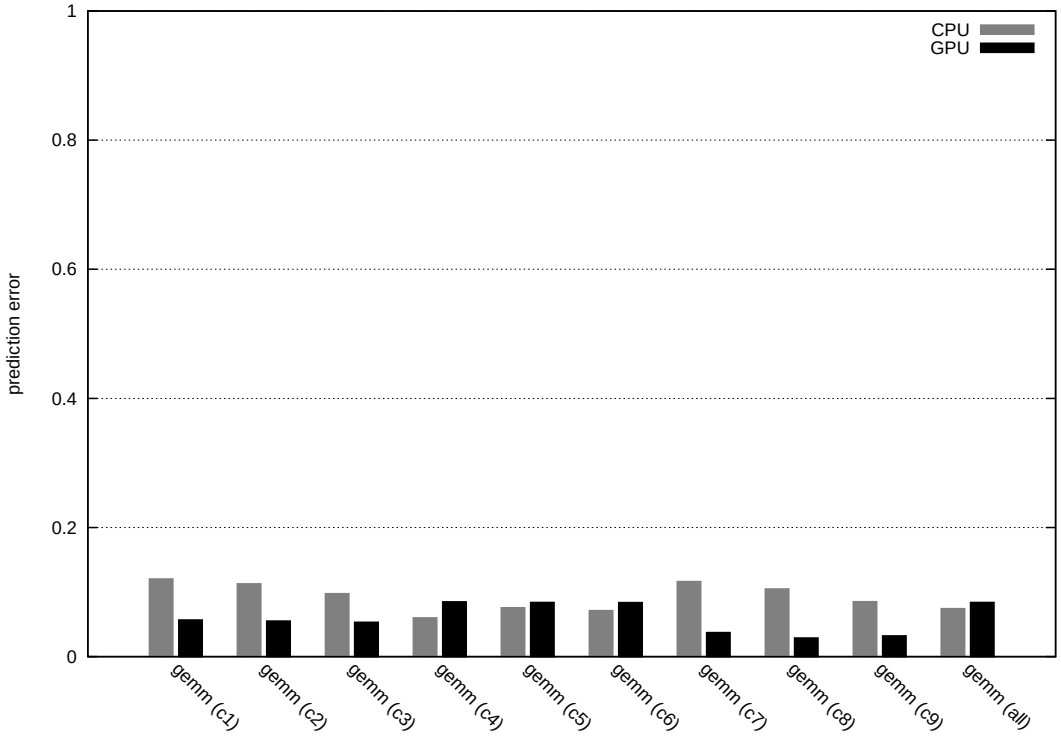


Figure 5.19 – Average prediction error ratio of CPU and GPU for *gemm*.

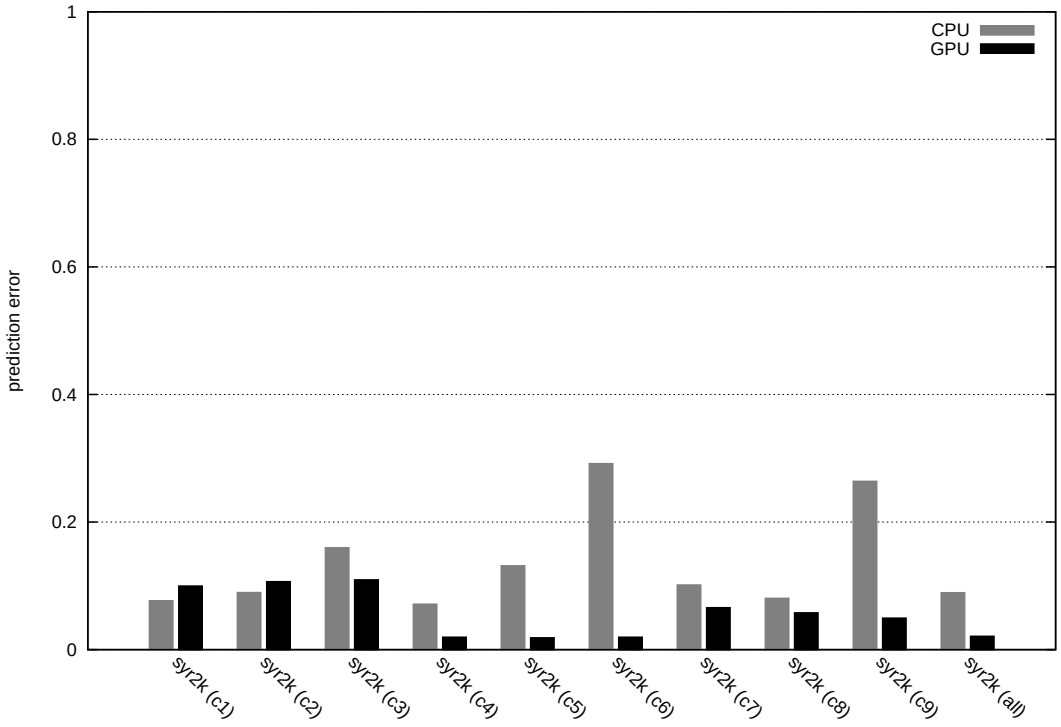


Figure 5.20 – Average prediction error ratio of CPU and GPU for *syr2k*.

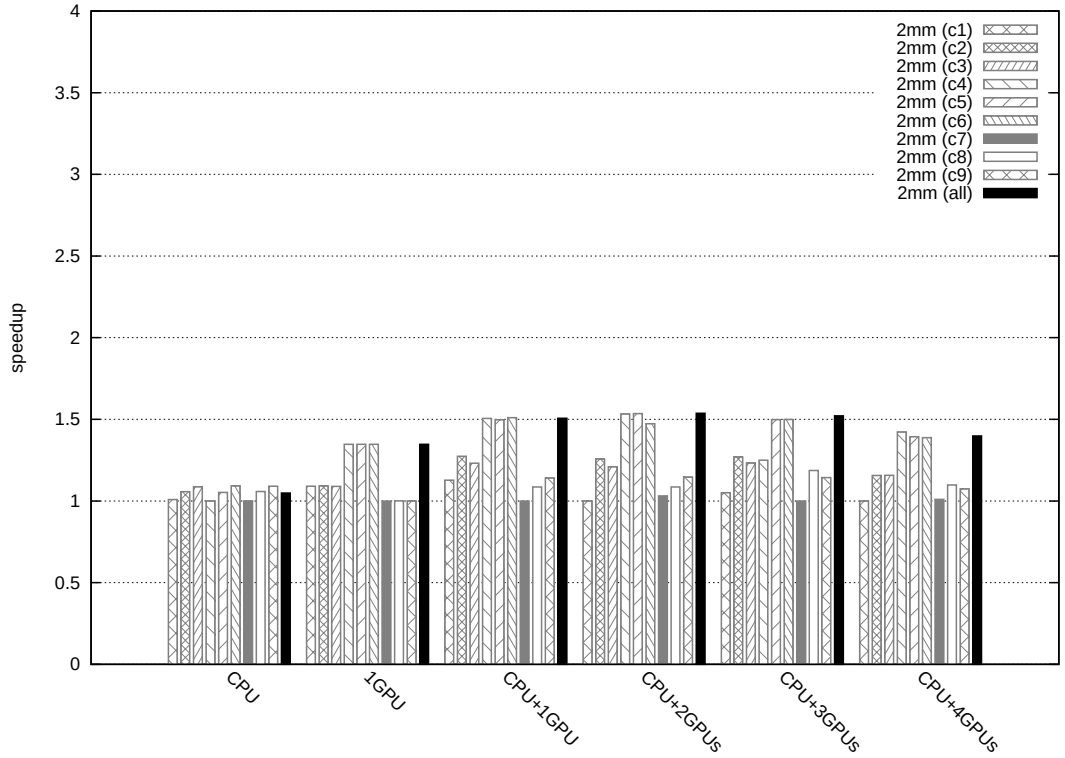


Figure 5.21 – Speedup to execution time of slowest code version combination for *2mm*.

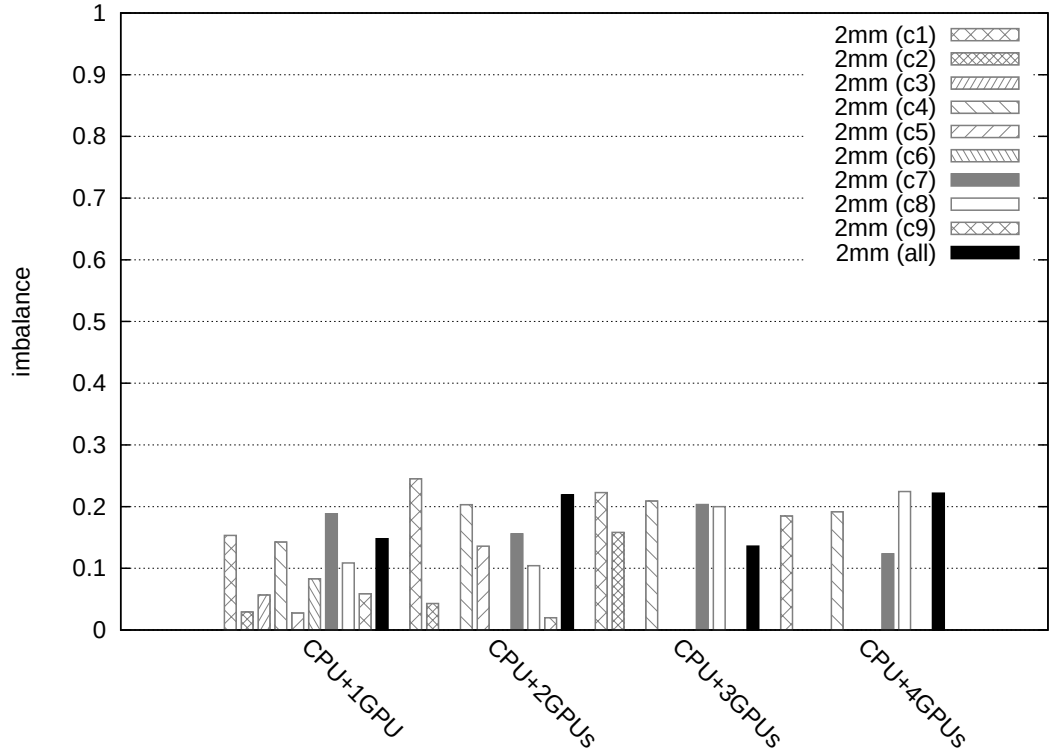


Figure 5.22 – Average prediction error ratio of CPU and GPU for *2mm*.

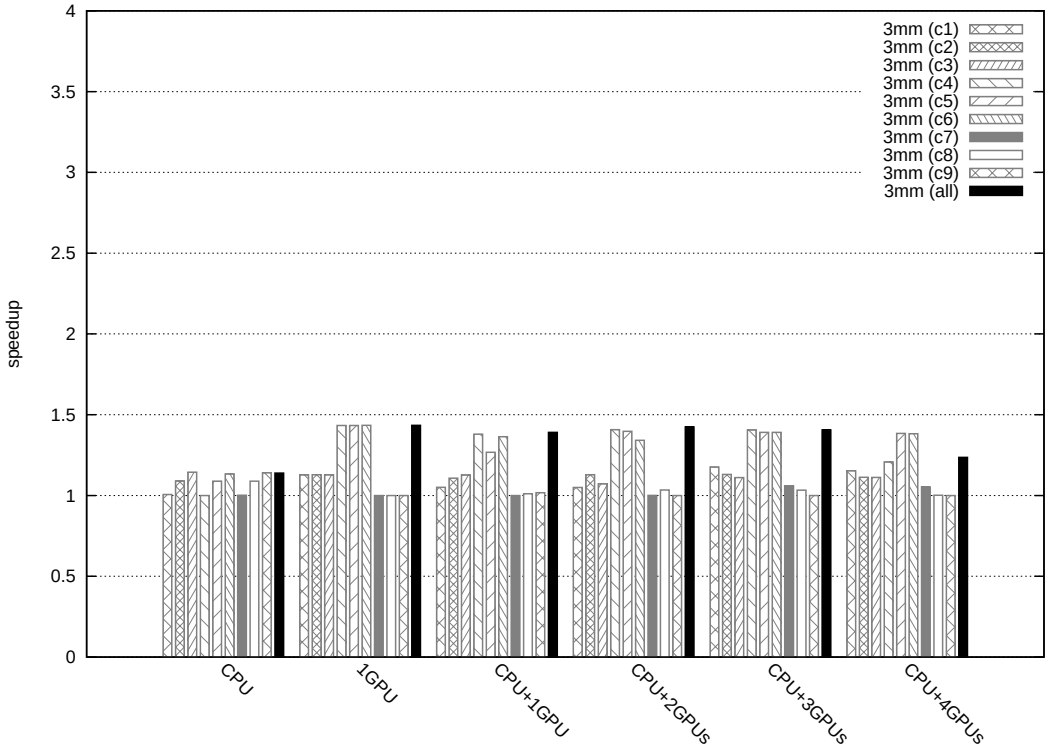


Figure 5.23 – Speedup to execution time of slowest code version combination for *3mm*.

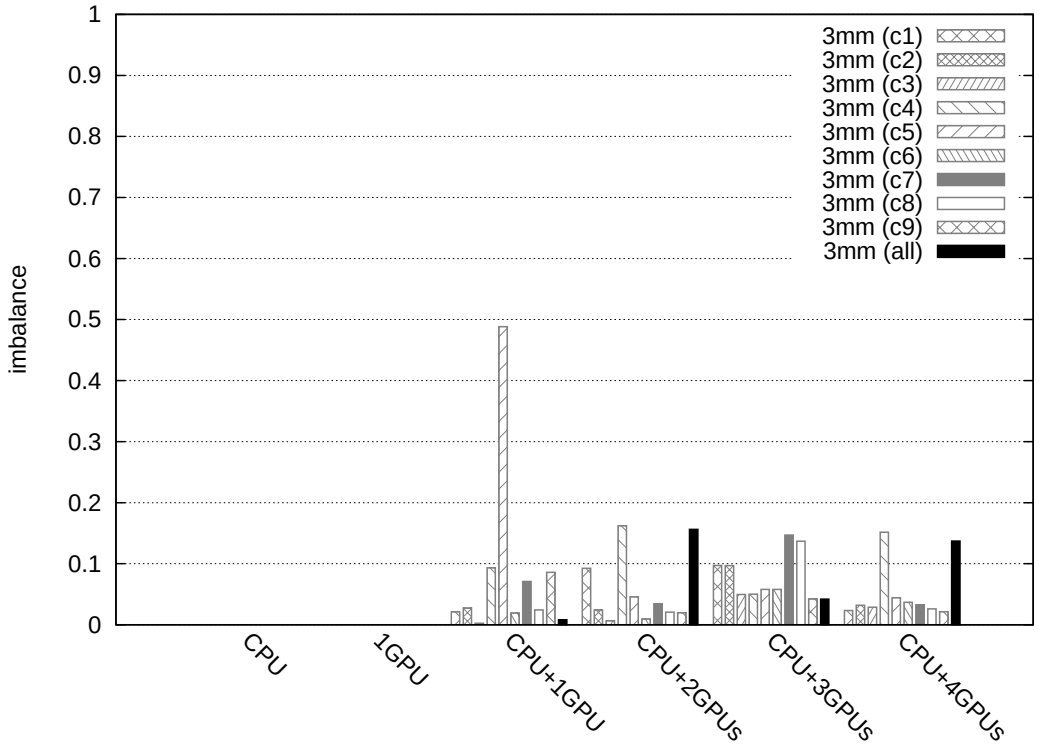


Figure 5.24 – Execution time imbalance ratio for several combination of code versions for *3mm*.

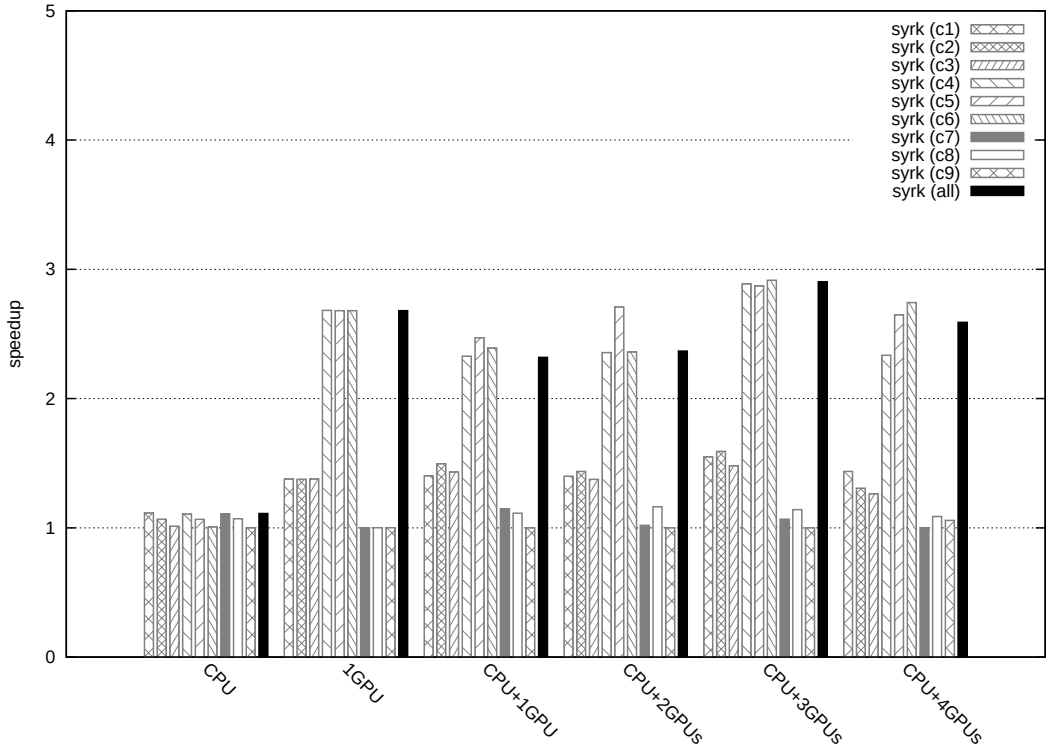


Figure 5.25 – Speedup to execution time of slowest code version combination for *syrk*.

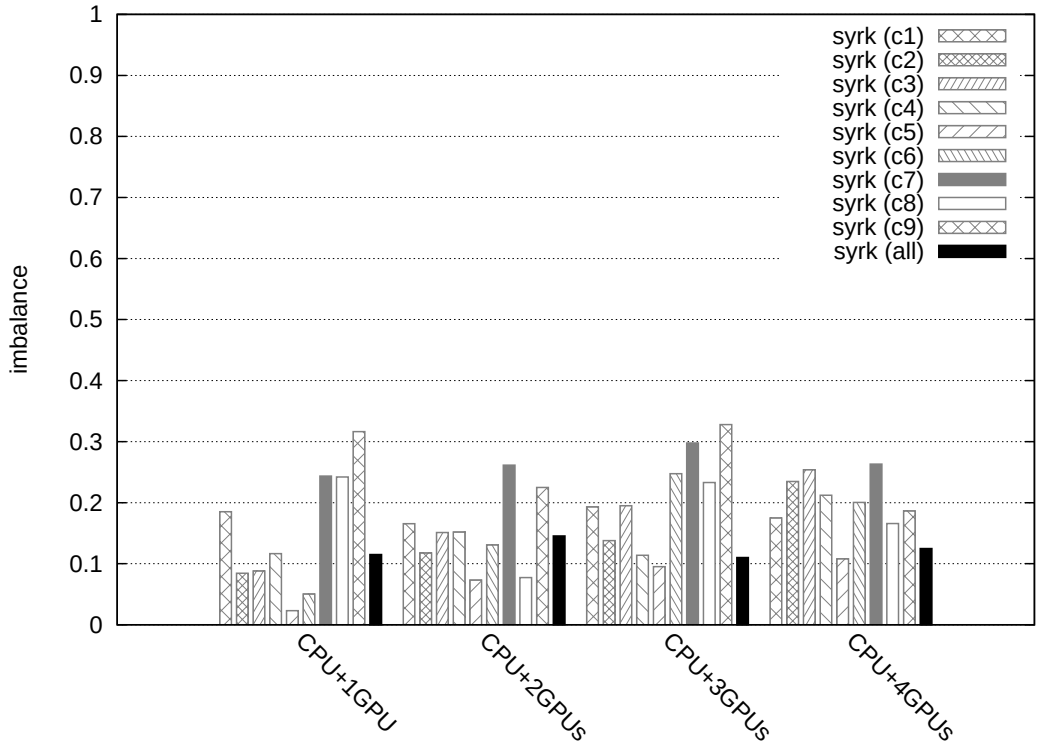


Figure 5.26 – Execution time imbalance ratio for several combination of code versions for *syrk*.

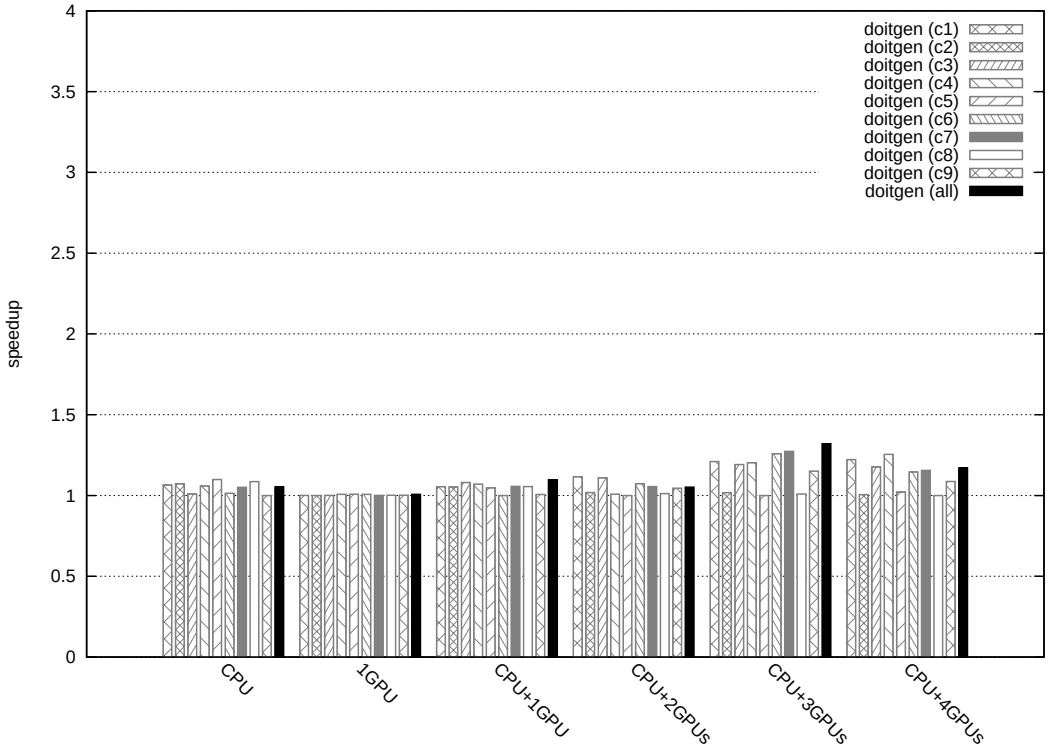


Figure 5.27 – Speedup to execution time of slowest code version combination for *doitgen*.

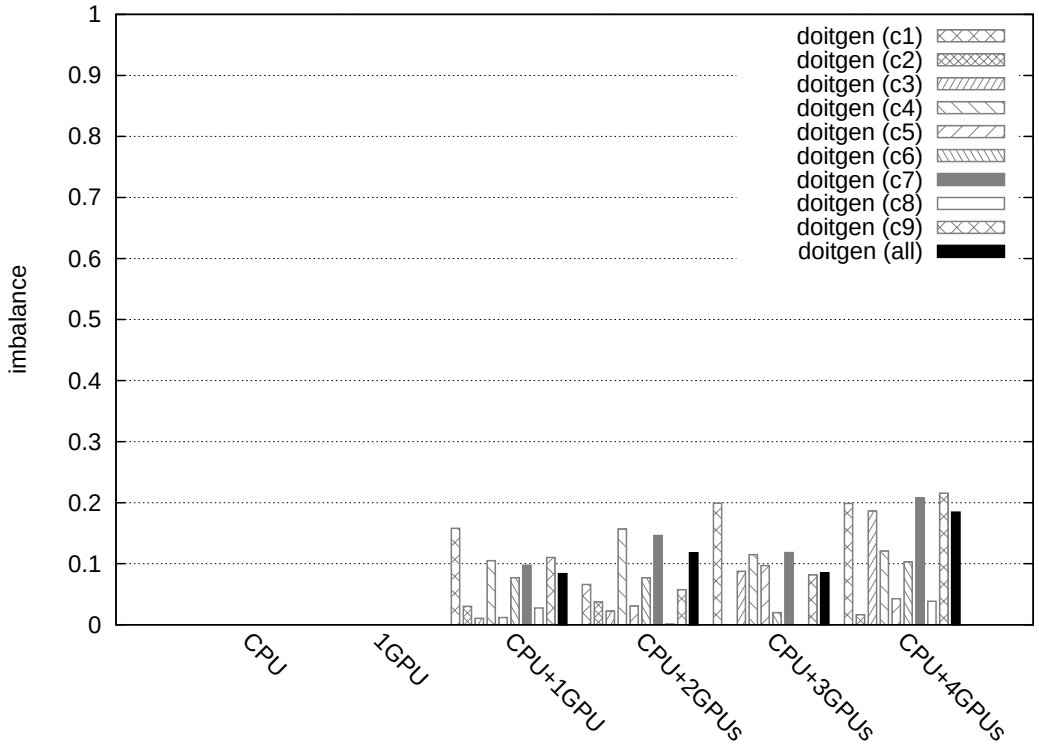


Figure 5.28 – Execution time imbalance ratio for several combination of code versions for *doitgen*.

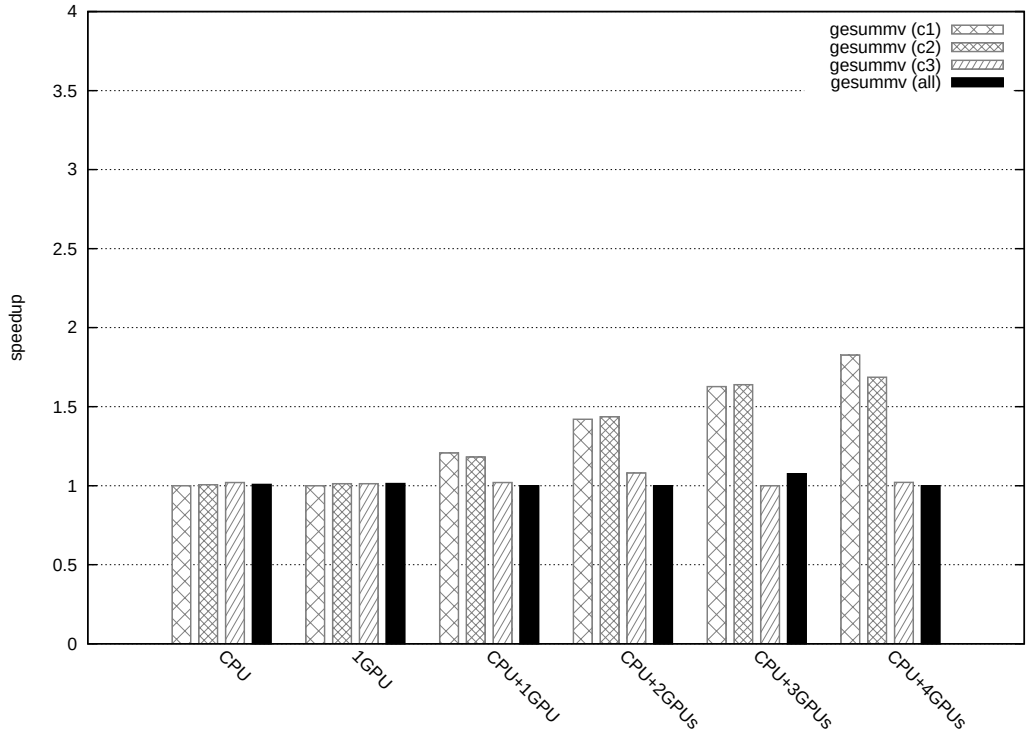


Figure 5.29 – Speedup to execution time of slowest code version combination for *gesummv*.

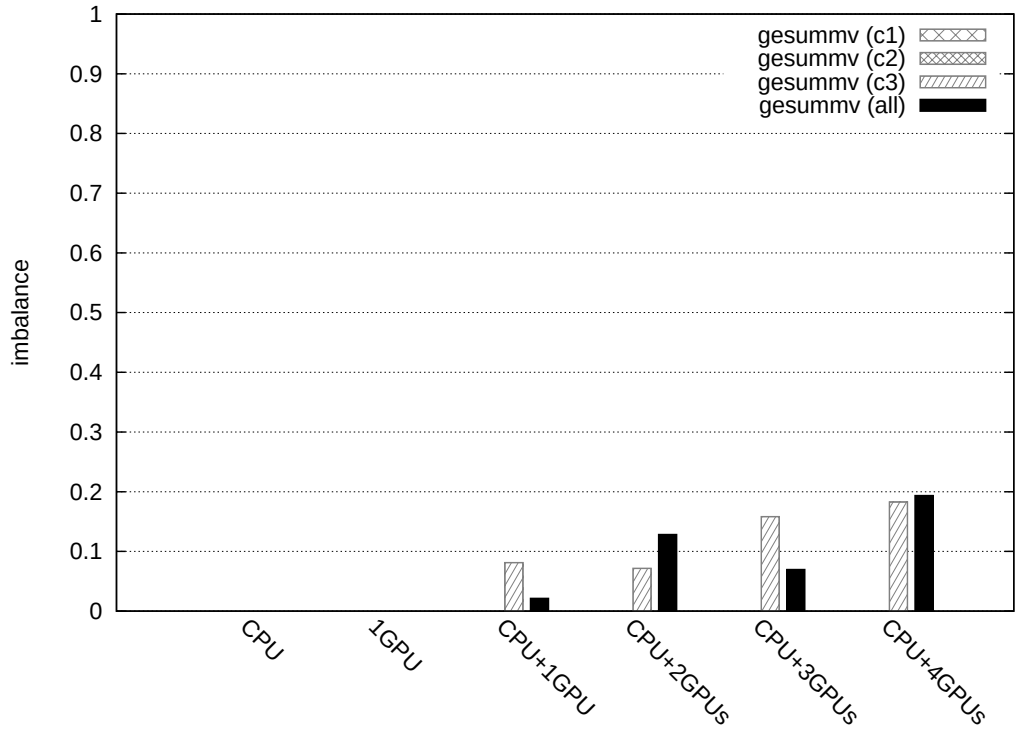


Figure 5.30 – Execution time imbalance ratio for several combination of code versions for *gesummv*.

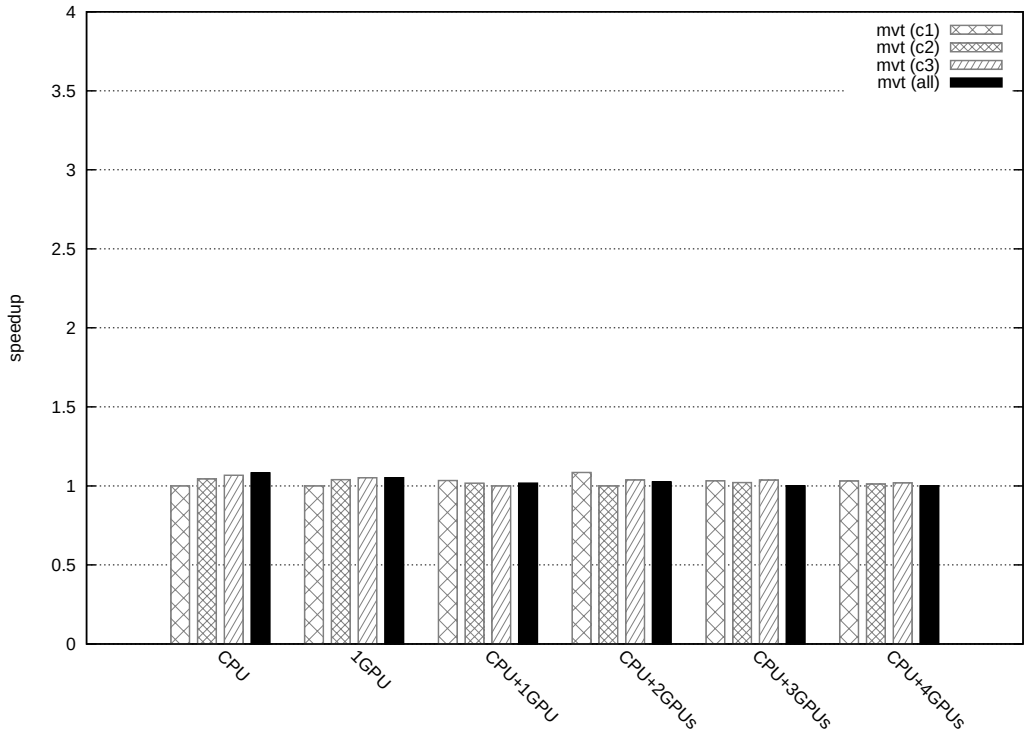


Figure 5.31 – Speedup to execution time of slowest code version combination for *mvt*.

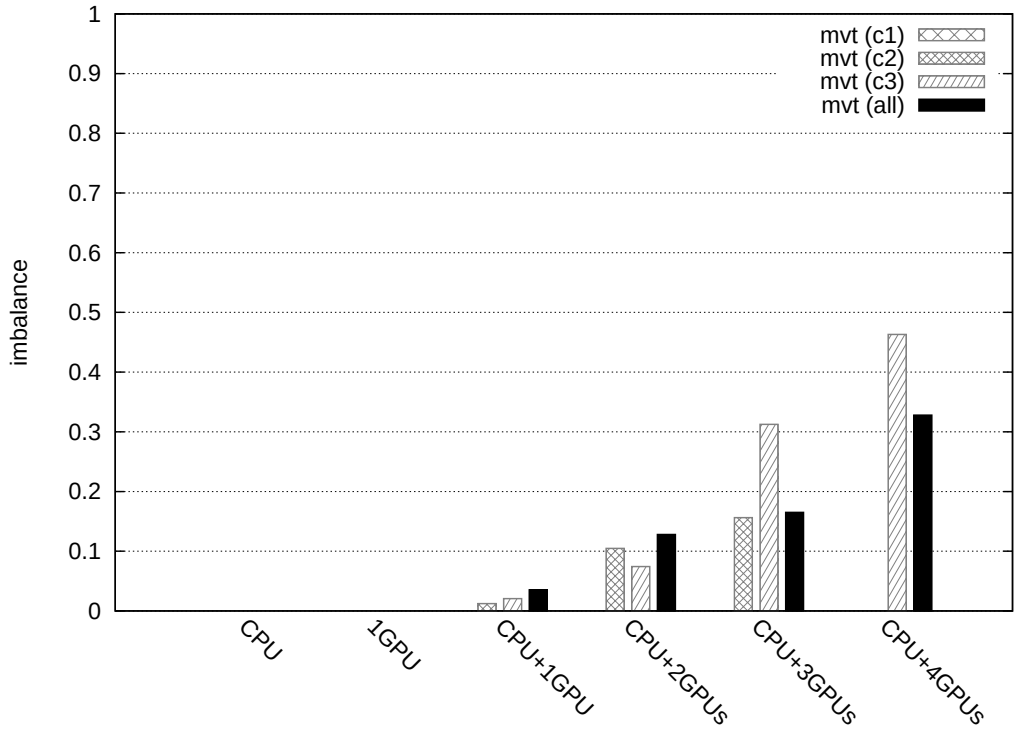


Figure 5.32 – Execution time imbalance ratio for several combination of code versions for *mvt*.

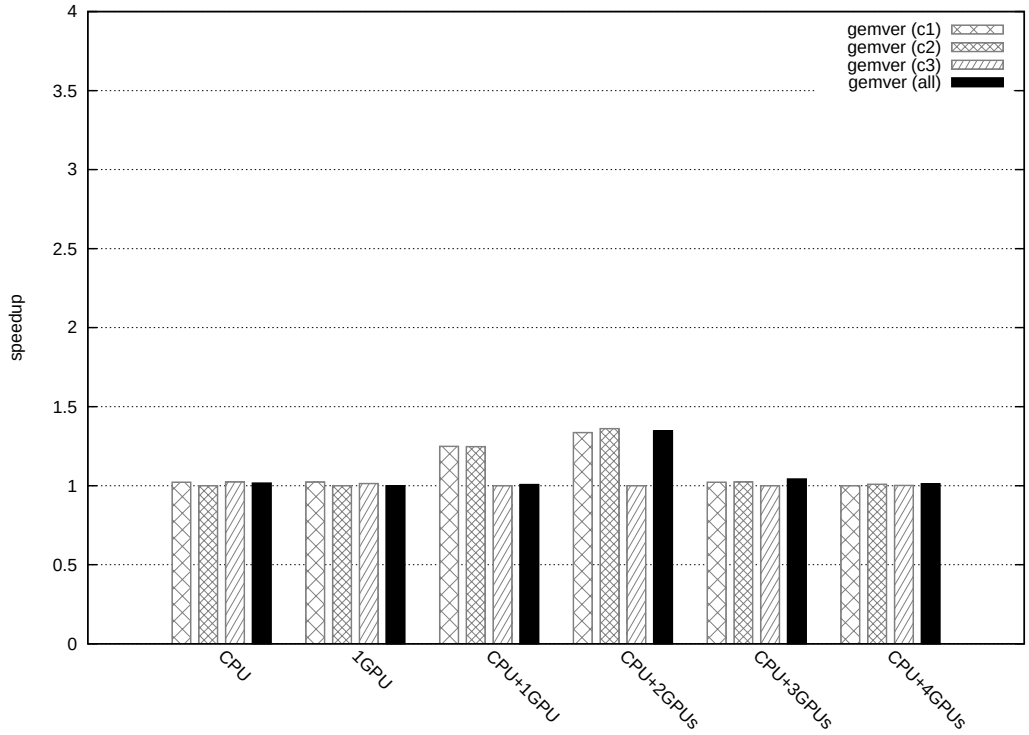


Figure 5.33 – Speedup to execution time of slowest code version combination for *gemver*.

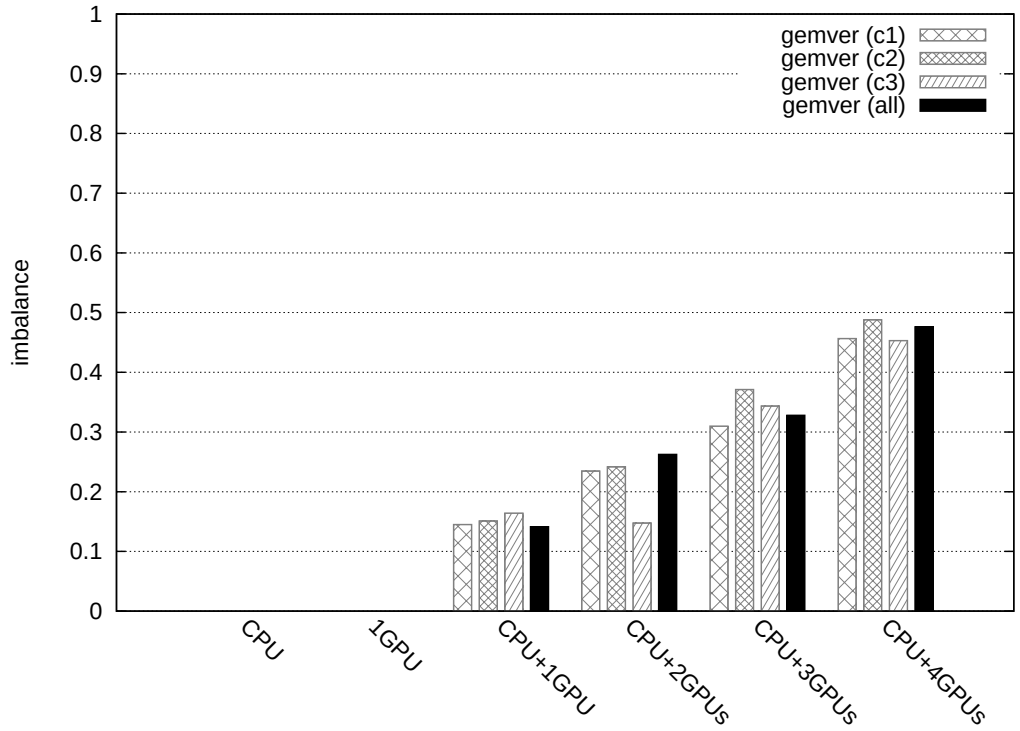


Figure 5.34 – Execution time imbalance ratio for several combination of code versions for *gemver*.

and GPUs for *syr2k*.

Imbalance shown in Fig. 5.16 and 5.18 mainly results from fluctuations in CPU time predictions. Despite the good accuracy of predictions, as confirmed by Fig. 5.19 and 5.20, slight changes in the partition size can significantly alter the predicted execution time and mislead the scheduler. This behavior is highlighted in *gemm (c7)*, for which the imbalance reaches 22%. However, imbalance is acceptable as it averages out at 5% and 8% for all the combinations of *gemm* and *syr2k*. Accuracy of our execution time prediction methods for *gemm* and *syr2k* is shown in Fig. 5.19 and 5.20. The plotted prediction errors are derived from the average error for all the PUs combination. Those results validate our methods for accurately predicting execution times.

Overall, the experiments show that our multiversioning system was systematically selecting the best version, thus improving performance. The design and low overhead of our runtime system allows the comparison of multiple schedules, combining different code versions, during execution.

5.3.6 Power-guided scheduling

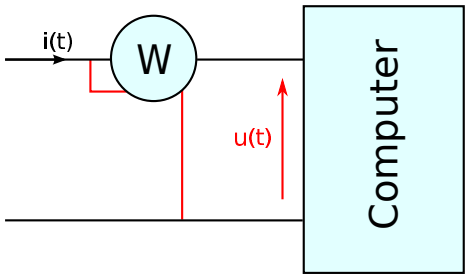


Figure 5.35 – Scheme of the measurement platform circuit with Wattmeter.

Energy consumption is the current hot-topic in computer system research, on account of power envelope, budget and ecological concerns. Energetic efficiency, is in general strongly correlated to the performance of a program. The faster the code, the lower the energy consumption, whatever architecture lies underneath. The GPUs generally expose a high absolute energy consumption, generally in several hundreds of watts: $365W$[5] for the GTX 590 (dual GPU), $195W$[6] for the GTX 680. Despite a potentially high peak power consumption, GPUs compensate by exposing a high flop per watt rate. In a heterogeneous context, schedulers may favour energy efficient architectures, when power consumption is a concern. In that case, PUs that expose an acceptable computing/energy consumption ratio are part of the computation. Others are kept idle, and can be fed with more appropriate work.

Why is the power consumption parameter important ? Because it allows the scheduler to take architecture selection decisions based on consumption. In particular, the decision point, which indicates whether to sideline a processor or not, shifts as we change the nature of the selection. For that purpose we introduce a new unit: watts per iteration. In that relation, time is left aside, as the role of the scheduler is to size

[5]http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-590/specifications
[6]http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications

chunks so that they have similar duration. The higher the number of iterations the more energy efficient the PU. This technique is sufficient to tap out inefficient PUs from the computation. The number of iterations is computed by *Ehrhart polynomials* which divide the appropriate device maximum energy consumption.

In this method the elimination of a PU may not always lead to a lower total energy consumption. To handle this, the scheduler could be called back recursively with the new PU configuration. The chosen configuration is the one for which predicted energy consumption is the lowest. This case is marginal, but it may happen when the PUs have strong power consumption disparities, therefore we use the simpler approach.

Energy-related experimentations targeting GPUs are rather difficult. In fact, to provide documentation Peres [112] describes the power management mechanism, which was reverse-engineered in the frame of the nouveau driver. In our test platform appropriate hardware counters are not exposed to the programmers. Measurements were performed with a *Fluke 41B* digital wattmeter. Measurements are performed in an upstream wattmeter electrical installation described in Fig. 5.35. The current is measured via an ammeter clamp, set up between socket and power supply. As usual, voltage is measured in parallel to the power supply. For this purpose we modified a traditional multi-socket in order to perform the measurements, right out of the wall socket. The measured phase shift is $cos(\phi) = 0.9$ on average and remained stable at ± 0.02. Provided numbers state active power, traditionally referred to as $P = U \times I \times cos(\phi)$.

In order to take energy into consideration, the proposed scheduler only requires slight modifications. The original processor exclusion mechanism bases its decisions on the relative number of iterations performed by a PU, in comparison to the maximal number of iterations. The PUs that execute less than 10% of the biggest chunk, are excluded from the computation. We apply the same principle by comparing the chunks power per iteration. This brings the low energy consuming PUs in front, with a good level of performance.

A more accurate energy consumption model would be unnecessary and impractical for the coarse-grain decisions taken by our scheduler. Also, building such a model requires thorough manual experiments in order to provide accurate energy consumption predictions. As performance is our main concern, we decided to take a single reference code version, and sticked to it during the energy experimentations. Extensive measurements were operated for *gemm*, *syrk* and *doitgen*. The scheduler is parametrized by the maximal power consumed during the experiments as reference. Also, communications and processor usage are considered equal in terms of power consumption. This is a first glimpse towards handling the energy case in a CPU+GPU scheduler. Further refinements to this model will be provided in future work.

An increase in the number of blocks, provokes an increase in power consumption, as demonstrated by Fig. 5.36 and Fig. 5.37. For comparison, we provide the power consumed by the host *core i7 2700k* processor, referred to as *CPU norm.*, for 8 active threads. A PU active power is given by the relation, $P_{PU} = P_a - B_a$ where B_a, the basic active power of the system is subtracted from the measured active power P_a. During the experiments, only the tested device was stressed. We considered execution environment variations as insignificant.

Through experiments we noticed that *gemm* was consistently draining more power

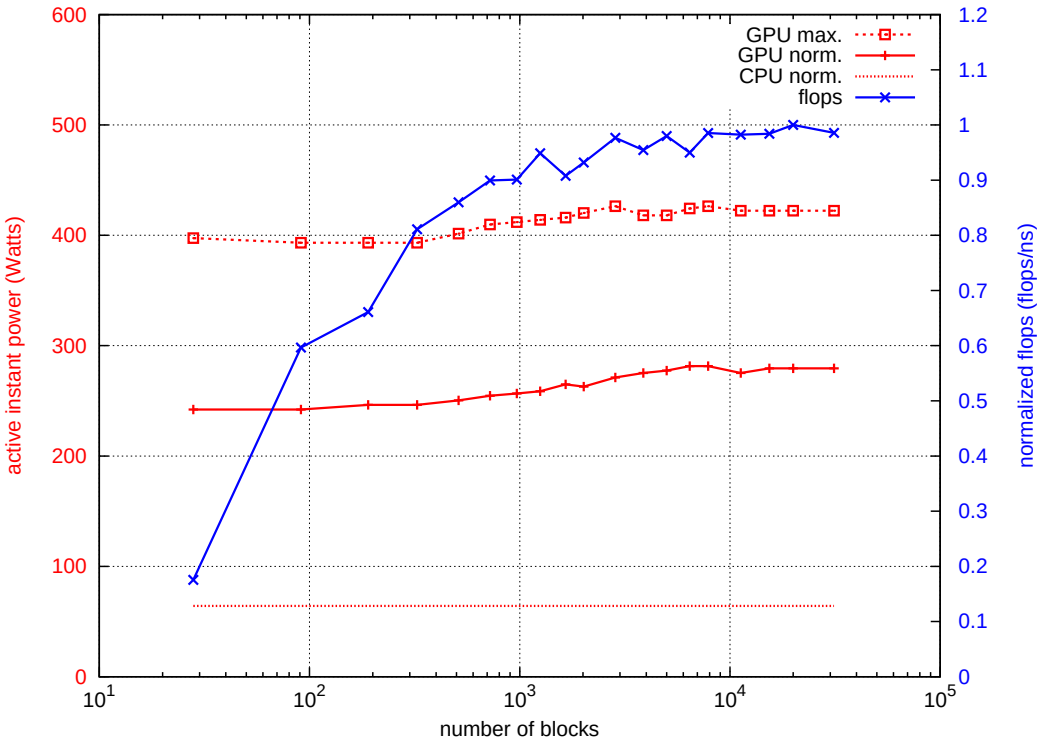


Figure 5.36 – Instant power consumption (y) and normalized flops (y2) for *gemm*.

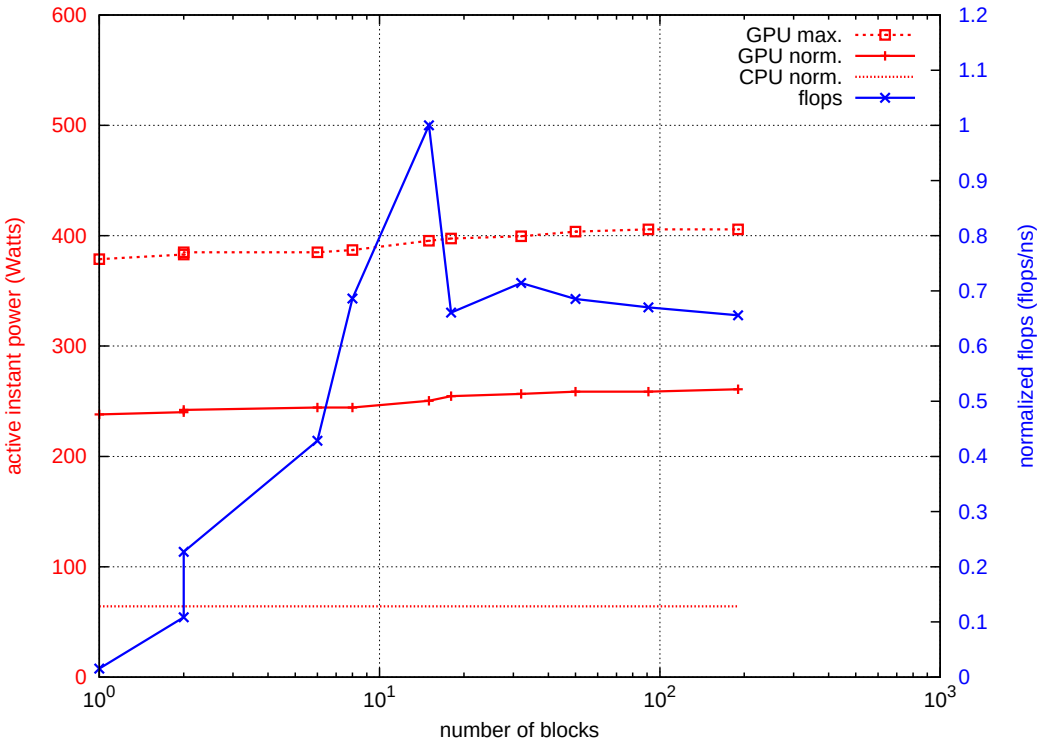


Figure 5.37 – Instant power consumption (y) and normalized flops (y2) for *doitgen*.

than other codes. This observation may be related to the intensive computational resource usage of this particular code. On our dual-GPU card platform (GTX 590), it appeared that the driver was setting the card at maximum performance when submitting CUDA commands, see *GPU max.* in Fig. 5.36 and 5.37. This punctually leads to a huge energy consumption overhead when only one over two GPUs is targeted. This overconsumption lasts between 10 to 15s in general, and is denoted with dashed lines. As this behaviour seems strongly coupled to our test platform, it is not taken into account by the scheduler. The power requirements of the sole computation are denoted by the solid curves. We noticed that there is no direct relationship between the flops and the energy consumption, preventing the reduction of manual measurements.

For the considered codes, the CPU energy consumption tops when the number of threads reaches the number of physical cores. The energy consumption stabilizes at $\sim 64W$, when the virtual cores are used (Hyper-Threading).

The consideration of energy constraints by the scheduler, produces the speedups presented in Fig. 5.38, 5.39 and 5.40, for respectively *gemm*, *syrk* and *doitgen*. In the same order, energy consumption is depicted in Fig. 5.41, 5.42 and 5.43. Since running *gemm* on the CPU requires significantly more power it is eliminated. In fact, there is approximately a factor of 4 between the CPU and GPU computed watts per iteration. However, as is noticing in the plot, CPU was already eliminated with the iteration count based strategy due to its weaker performance for that particular code. In that particular case, with only GPUs running, imbalance averages to 2%. Conversely, the GPUs are eliminated from the computation for *doitgen*, as observable in Fig. 5.43. By the way, impact on the speedup is reasonable, especially compared to the power consumption improvement. For *syrk*, the scheduler chooses to pursue the computation on the CPU and GPU except for versions 4, 5 and 6. This is confirmed by the speedups showed in Fig. 5.25. The scheduler overhead is similar to the results obtained in section 5.3.4.1.

5.3.7 Perspectives and conclusion

We presented an original method for achieving load balance between CPUs and GPUs in a dynamic context. It is based on an accurate prediction of the CPU and GPU execution times of codes, using the results of a profiling of those codes. We implemented it using Python scripts, calling several polyhedral compilation tools, and we tested it on the polyhedral benchmark suite, showing that it is effective on a platform composed of one CPU and 4 GPUs. The scheduler works independently from the prediction method and allows extensions to other prediction mechanisms in the future.

Also, we propose an extension to our performance-oriented scheduler to take energy into consideration. We showed that with slight modifications the scheduler was adapted to improve energy consumption by deactivating the inefficient PUs. The experiments have shown that it is able to select the right PU based on energy restrictions. This is a first step towards handling consumption in the scheduler and may be extended in the future, with hardware exposing more information and fine-grain energy models.

Our future plans include extending this work to handle other types of hardware, for example Xeon Phi processors and larger systems including 10's of GPUs and 100's of cores.

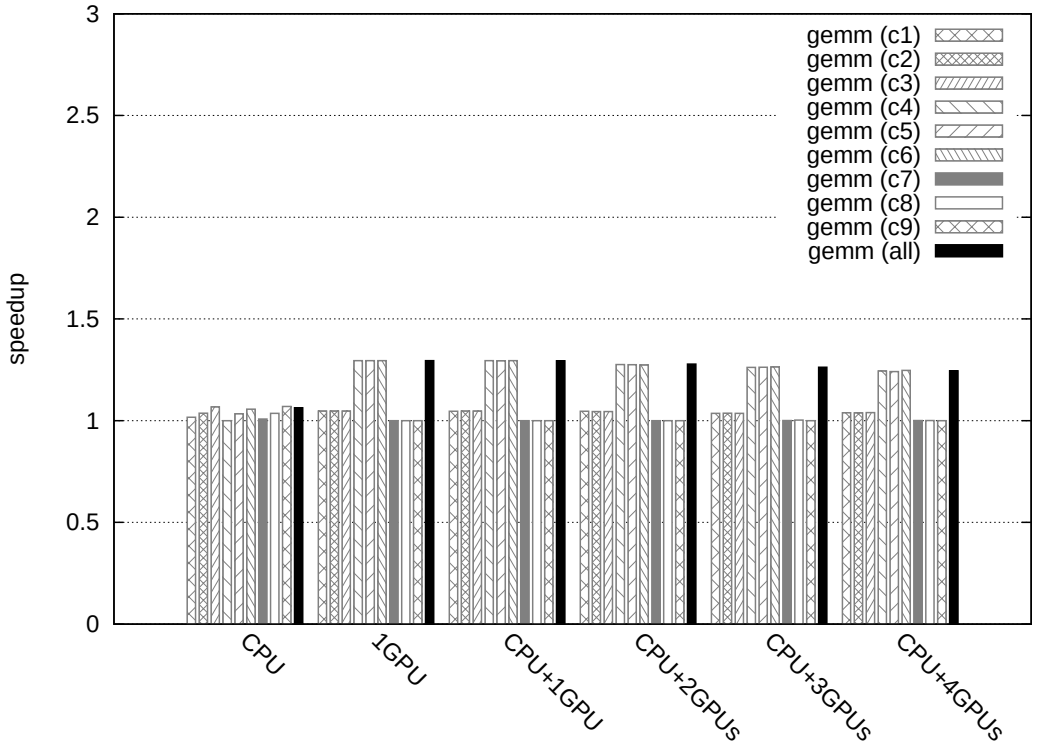


Figure 5.38 – Speedup to execution time of slowest code version combination for *gemm* with energy-enabled scheduler.

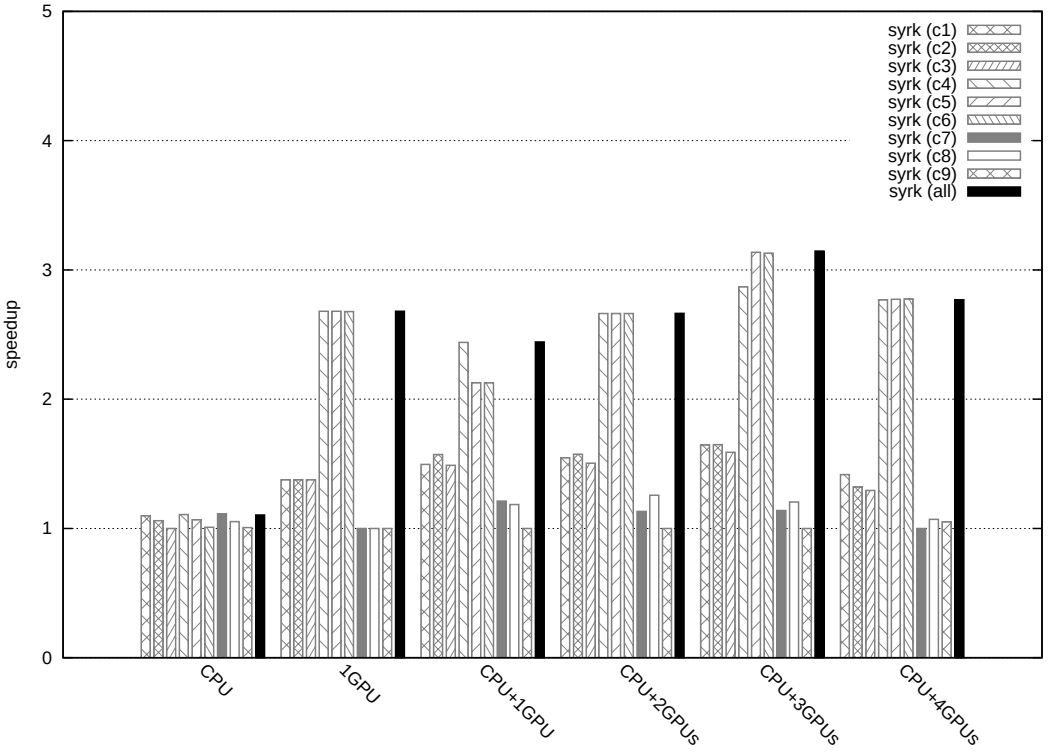


Figure 5.39 – Speedup to execution time of slowest code version combination for *syrk* with energy-enabled scheduler.

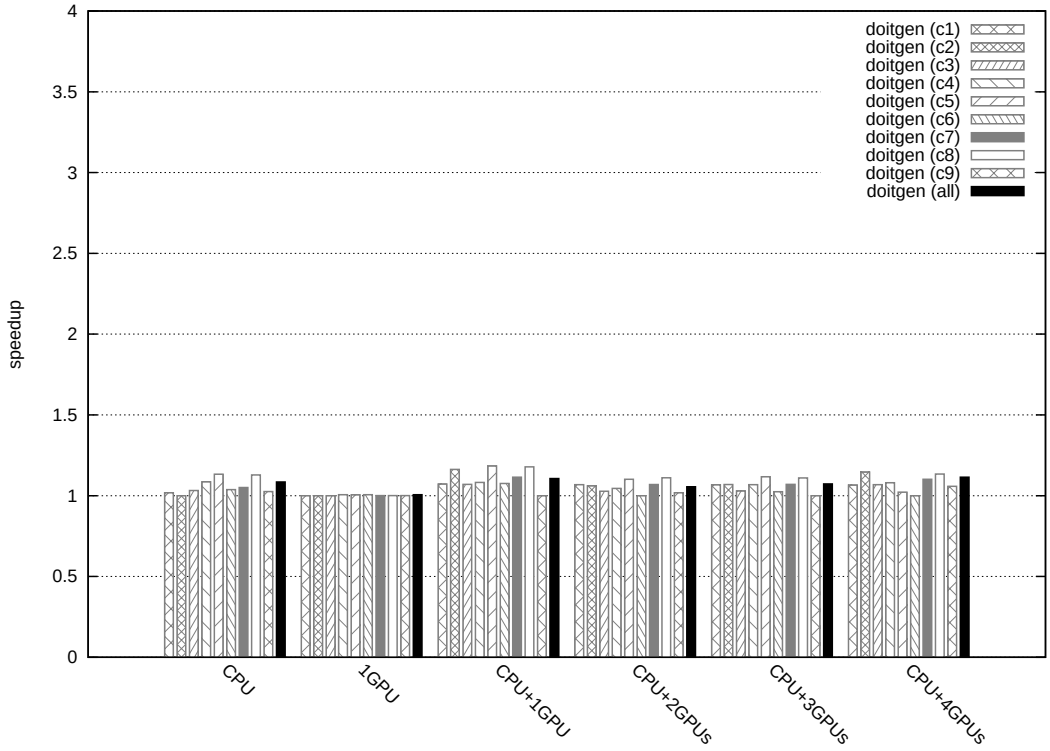


Figure 5.40 – Speedup to execution time of slowest code version combination for *doitgen* with energy-enabled scheduler.

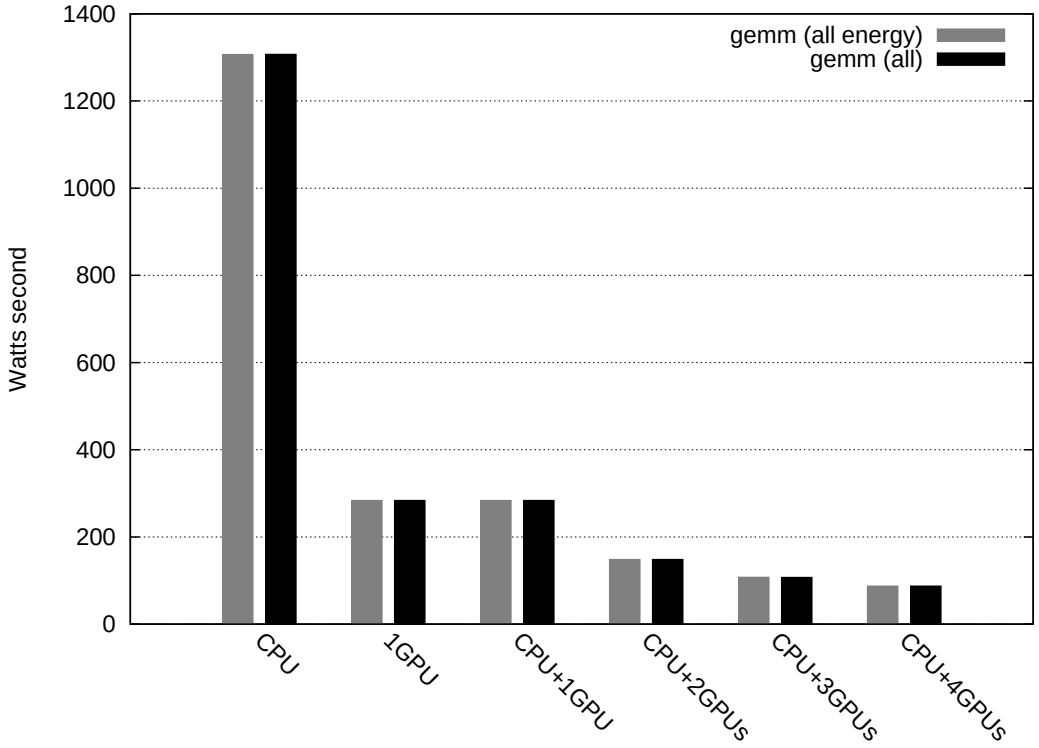


Figure 5.41 – Comparison of energy consumption for *gemm* between energy enabled scheduler (left bar) and iteration count based elimination (right bar).

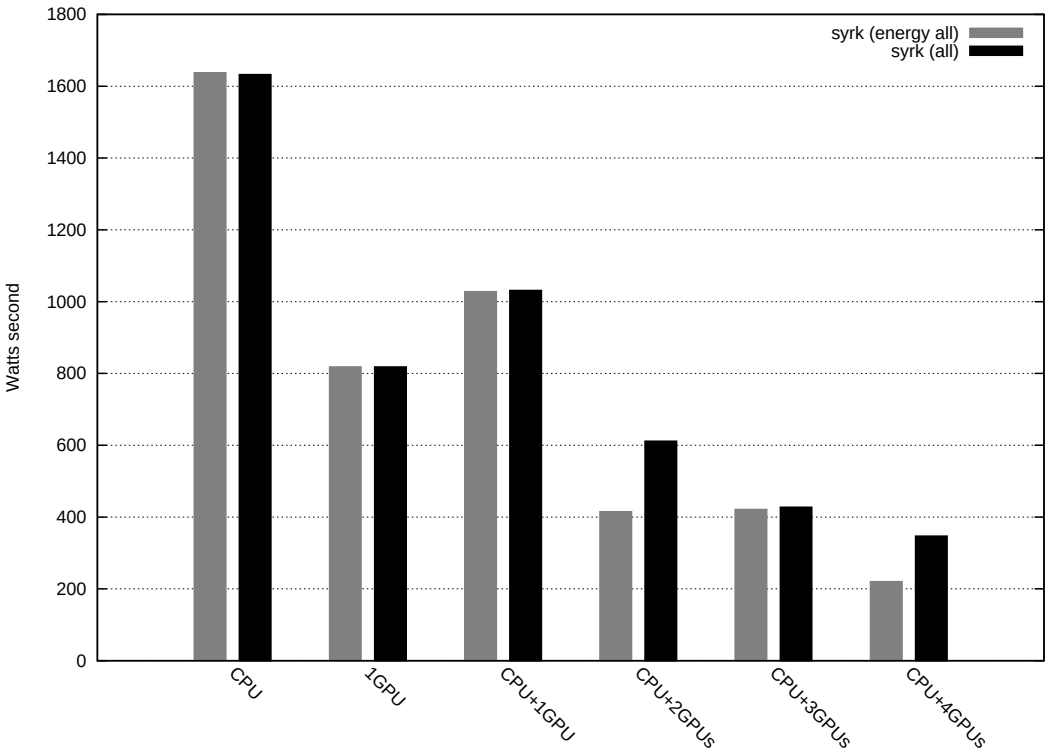


Figure 5.42 – Comparison of energy consumption for *syrk* between energy enabled scheduler (left bar) and iteration count based elimination (right bar).

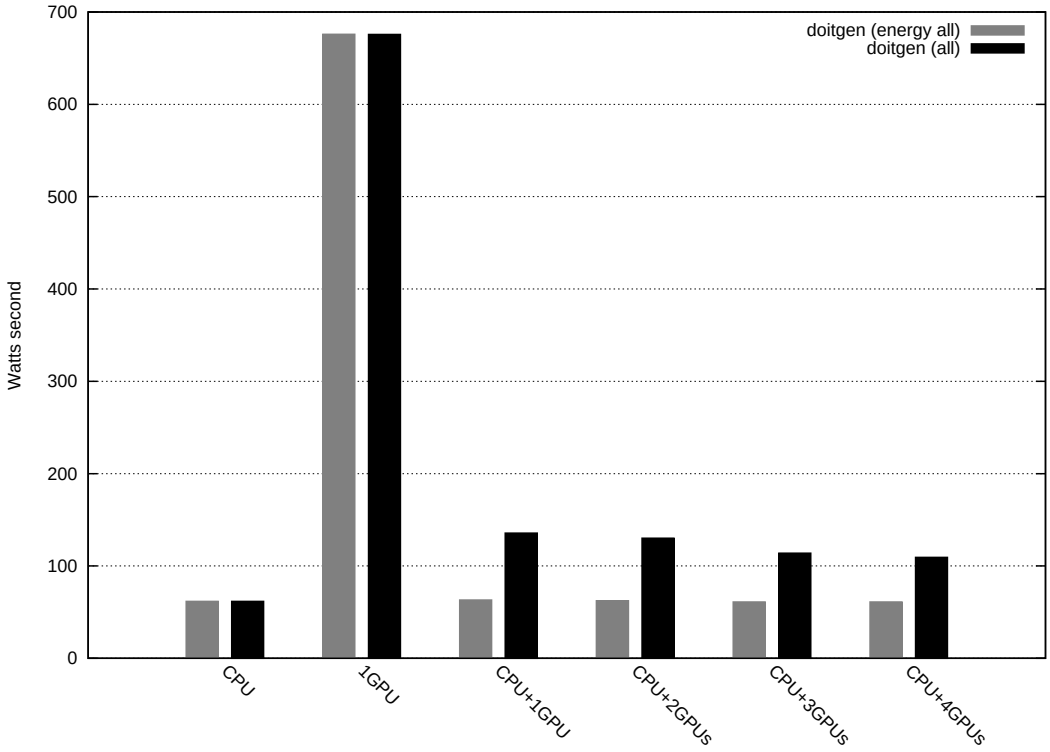


Figure 5.43 – Comparison of energy consumption for *doitgen* between energy enabled scheduler (left bar) and iteration count based elimination (right bar).

Chapter 6

Thread Level Speculation

6.1 CPU speculative execution

This work is the result of a collaboration with Alexandra Jimborean, who defended her Ph.D. in 2012 in our team. My main contributions reside in the virtual machine: the memory backup system in Section 6.4 and 6.6 and a part of dynamic code generation in Section 6.3. A proposal towards the extension of the framework for GPUs is presented in Section 6.7. This Chapter is mainly taken from our publication [66].

6.1.1 Introduction

Automatically parallelizing sequential code became increasingly important with the advent of multicore processors. Particularly, the polyhedral model [25], originally designed for compile-time loop optimizations and parallelization, is known to show immense benefits for kernels written as loops with affine control of their iteration counts and array accesses. However, frequently, even scientific codes embed loop nests with bounds that cannot be statically predicted, having complex control flows or containing pointers leading to issues such as memory aliasing. As static analysis is too fragile, one relies on run-time speculations to achieve a stable performance [74]. Additionally, to generate highly performing code, it is crucial to perform optimizations prior to parallelization. It is, nevertheless, a challenging task to parallelize code at runtime, due to the time overhead generated by the required analysis and transformation phases.

Runtime parallelization techniques are usually based on thread-level speculation (TLS) [124, 86, 123] frameworks, which optimistically allow a parallel execution of code regions before all dependences are known. Hardware or software mechanisms track register and memory accesses to determine if any dependence violation occur. In such cases, the register and memory state is rolled back to a previous correct state and sequential re-execution is initiated. Traditional TLS systems perform a simple, straightforward parallelization of loop nests by simply slicing the outermost loop into speculative parallel threads [124, 86, 69]. As soon as a dependence is carried by the outermost loop, this approach leads to numerous rollbacks and performance drops. Moreover, even if infrequent dependences occur, nothing ensures that the resulting instruction schedule improves performance. Indeed, poor data locality and a high

amount of shared data between threads can yield a parallel execution slower than the original sequential one. To gain efficiency, TLS systems must handle more complex code optimizations that can be profitably selected at runtime, depending on the current execution context.

At a higher level of parallel programming, a growing interest has been paid to the use of algorithmic skeletons [33, 84], since many parallel algorithms can be characterized and classified by their adherence to one or more generic patterns of computation and interaction. In [33], Cole highlights that the use of skeletons offers scope for static and dynamic optimization by explicitly documenting information on algorithmic structure which would often be impossible to extract from equivalent unstructured programs. On the other hand, Li *et al.* propose in [84] a lower level implementation of data-parallel skeletons in the form of a library, as a good compromise between simplicity and expressiveness.

According to these features, we extend the concept of algorithmic skeletons in the following ways. Our skeletons:

- are generated automatically at compile-time and embedded in the final executable file;
- are specialized to the loop nests that are initially marked in the source code using a dedicated pragma;
- implement a given combination of polyhedral loop transformations as loop interchange, skewing or tiling [25];
- have a fixed algorithmic structure with instructions of three categories: (1) original program instructions, (2) polyhedral transformation instantiation and (3) speculation management, where categories (2) and (3) contain parameterized code, *i.e.*, with unknown values of some variables;
- are rendered executable at runtime by assigning values to the parameters of instruction categories (2) and (3).

Such skeletons, embedded in the executable file of the target program, significantly improve dynamic and speculative parallelization opportunities by allowing very fast code generation and advanced automatic parallelizing transformations of loop nests. Unlike previous works [98, 72, 134] which report on simple patterns (templates) with "holes", replacing branches, that are filled dynamically, we design skeletons which dynamically instantiate a polyhedral loop transformation and embed instructions to manage speculative parallelization. Starting from the skeletons, different, optimized code versions can be generated, by assigning values to some parameters.

To support the generation and the use of these skeletons, we propose a lightweight static-dynamic system called VMAD – for *Virtual Machine for Advanced Dynamic analysis and transformation* – which is a subclass of TLS systems, devoted to loop nests that exhibit linear behavior phases at runtime. We define linear behavior phases as being characterized by outermost loop slices where:

- All accessed memory addresses can be represented as affine functions of the loop indices;

- All loop bounds, except the outermost, can be represented as affine functions of the enclosing loop indices;
- The values assigned to some particular variables, called *basic scalars*, can also be represented as affine functions of the loop indices. These scalars are detected at compile-time as being variables defined by ϕ-nodes in the LLVM Static Single Assignment (SSA) intermediate representation. They have the interesting property of being at the origin of the computations of all other scalars used in the loop bodies.

Our contributions can be summarized as:

- Perform dynamic and advanced loop transformations and generate efficiently the resulting parallel code by using compiler-generated algorithmic skeletons at runtime;
- Exhibit parallelism in codes that cannot be parallelized in the original form;
- Adapt dynamically to the current behaviour of the code and apply a suitable skeletal code transformation;
- Exploit partial parallelism, *i.e.*, parallelism that can only be exploited on some slices of the outermost loop;
- Apply the polyhedral model on *for*, *while* and *do-while* loops that exhibit linear behaviour phases;
- Do not require any hardware support dedicated to speculative parallelization.

Preliminary ideas of our speculative system are presented in our previous work [67], while the proposal presented in [68] details the instrumentation and analysis phase, required for building an abstract model of the loop nest. The current chapter focuses on using the abstract representation of the loops in order to automatically perform speculative optimizations and parallelization at runtime. All aspects and details of the whole framework can be found in [65].

6.2 Overview of our system

This proposal extends our work on designing a TLS framework able to apply polyhedral transformations at runtime [67], such as tiling, skewing, loop interchange, etc., by speculating on the linearity of the loop bounds, of the memory accesses and of the values taken by specific variables, the basic scalars. Speculations are guided by online profiling phases. The instrumentation and analysis processes are thoroughly described in our previous work [68]. The system is entirely automatic and attempts the parallelization of the loop nests without any intervention of the user. Moreover, the programmer can, but is not required to, specify the loop nests of interest. All code manipulations are performed in the intermediate representation (LLVM IR), hence our framework is both programming language and target agnostic.

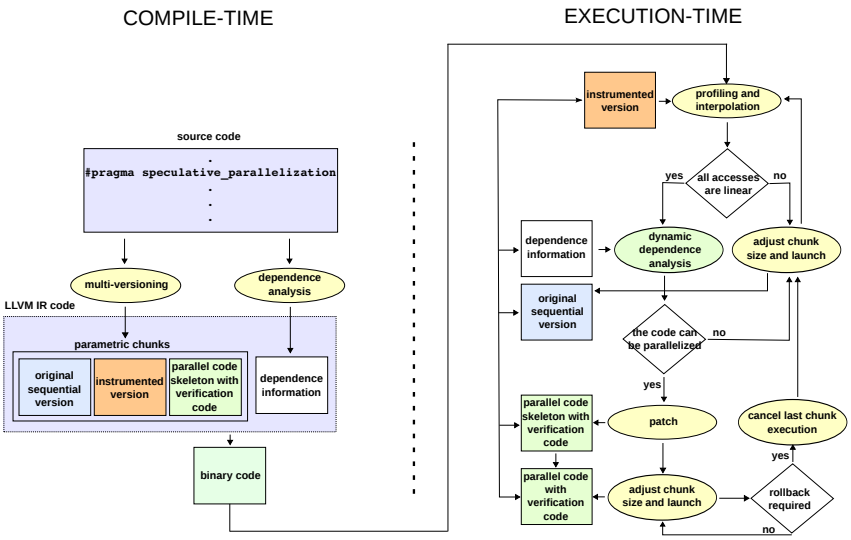


Figure 6.1 – Static-dynamic collaborative framework

Using the chunking mechanism presented in Fig. 6.2, we slice the iteration space of the outermost loop in successive chunks. The bounds of the chunks are determined dynamically to adapt to different execution phases of a loop. Execution starts with a profiling phase, whose results are used to validate a suitable polyhedral transformation. Next, a new transformation is proposed for each phase, and a customized parallel version is generated, by patching the skeleton. If several parallelizing transformations are validated, VMAD includes a module dedicated to select the best performing code version. This is achieved by launching successively small chunks embedding one of the possible parallel versions, and selecting the best performing one, according to the resulting average of execution time per iteration. Phase detection is detailed in the end of this section and relies on the code dedicated to monitor the speculations.

During the speculative execution, the predictions are verified, initiating a rollback upon a misspeculation and resuming execution with a sequential chunk. If validation succeeds, a new parallel chunk is launched. The implementation of VMAD consists of two parts: a *static* part, implemented in the LLVM compiler [89], designed to prepare the loops for instrumentation and parallelization, and generate customized parallel skeletons, and a *dynamic* part, in the form of an x86-64 runtime system whose role is to build interpolating functions, perform dynamic dependence analysis and transformation selection, instantiate the parallel skeleton code and guide the execution, as illustrated in Fig. 6.1. Since the compiler is target agnostic, the framework is independent of the target architecture (as long as LLVM provides a back-end), nevertheless, the runtime system has to be ported on the new architecture.

Static component Our modified LLVM compiler generates customized versions of each loop nest of interest: original, instrumented and several algorithmic skeletons each supporting a specific class of polyhedral transformations, together with a mechanism for switching between the versions (a decision block preceding the code versions).

To complete the loop's execution and adapt to the current phase, we automatically

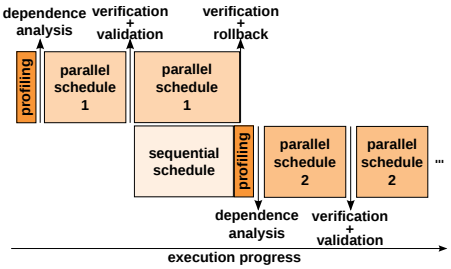


Figure 6.2 – The chunking mechanism

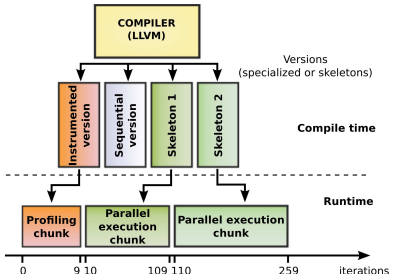


Figure 6.3 – Alternate execution of different versions during one loop nest's run

link up at runtime the different versions of the original code. Each version is launched in a chunk to execute a subpart of the loop and another one continues, as in relay races. The support for chunking the outermost loop and linking distinct versions is illustrated in Fig. 6.3. The instrumented, original and two skeletons are built at compile time. At runtime, one or another version is automatically selected to be executed for a number of iterations.

We build skeletons from which several parallel code versions can be generated at runtime, by patching predefined code areas. The skeletons are embedded in the compiler-generated binary file. The advantages are that the code size is significantly reduced and that many different parallel code versions can be build dynamically, guided by the results of the dynamic analysis. Also, patching a binary skeleton is considerably faster than fully dynamic code generation using JIT (Just-In-Time) compilation. However, the limitation of this approach is that it can only support a subset of the possible polyhedral transformations, namely those preserving a given loop structure and avoiding to reorder the statements inside the body of the loops. More details on the design of the skeletons are given in the next section.

A set of polyhedral loop transformations, defined as matrices, is generated statically and is encoded in the data section of the binary file. Their computation follows a static dependence analysis, which ensures that the dependences which can be statically identified will not invalidate the schedules, thus preventing useless scheduling alternatives. Generally, even if no static dependence can be identified to guide the selection of transformations, it is convenient to consider "classic" transformation matrices that are unimodular and resulting in combinations of loop exchange and skewing.

Dynamic component The runtime system collaborates tightly with the static component. During the instrumentation phase, it retrieves the accessed memory locations,

the values assigned to the basic scalars, and computes interpolating linear functions of the enclosing loop indices. Instrumentation is performed on samples to limit the time overhead and to launch parallel code as soon as possible. Thus, the computed linear functions speculatively characterize the behaviour of the loop. Instrumentation is followed by a dependence analysis which evaluates whether any of the proposed polyhedral transformations can be efficiently applied. If successful, the runtime system assigns values to the coefficients of the linear functions in the corresponding code skeleton and launches it.

The dynamic dependence analysis is an incremental process that computes the distance vectors used in validating polyhedral transformations, based on the actual memory addresses accessed during the run of an instrumented chunk, and on the functions interpolating them. In order to ensure that the computed distance vectors entirely characterize the current code behaviour, the interpolating functions are used to check if any memory instruction couple, where at least one is a write, and for which no distance vectors have been computed, may carry a dependence. This is achieved by a fast value range analysis of the touched memory addresses and a GCD test. Additional information is available in [65].

The speculative execution is monitored by the runtime system. As soon as a misspeculation is identified, it is followed by a rollback which restores the memory to a correct state. For this purpose, the runtime system creates proactively a copy of the memory area that is going to be modified by the next parallel chunk, since it can be predicted using the interpolating linear functions. In case a rollback is performed, the memory is overwritten with the content of the copy, and the rollbacked iterations are re-executed using the original sequential schedule; next instrumentation is re-initiated. Otherwise, a new parallel chunk is launched.

Our system dynamically adapts to each phase of the loop by launching the corresponding code version for a subset (chunk) of iterations. Thus, the size of each chunk is computed based on the currently observed behaviour of the loop: a stable behaviour leads to an increase of the chunk size until a fixed threshold, whereas a change in the behaviour resets the size to a default starting value.

Next, we focus on the dynamic, lightweight code generation of parallel optimized code using skeletons.

6.3 Binary skeletons

The skeletons are prepared statically (corresponding to Skeleton 1 and Skeleton 2 versions in Fig. 6.3) and instantiated at runtime to generate distinct code versions (parallel schedule 1 and parallel schedule 2 in Fig. 6.2). For the purposes of clarity, we start with a pedagogical example in which the code can be statically analyzed, and we detail in what follows the implications on non-statically analyzable code and our solutions to handle it.

First, consider the simple two-loop nest, with indices i, j in the first column of Table 6.1. Performing the affine transformation (i, j) $\rightarrow$ (x, y) = (i+j, i) on the original loops, one obtains a new version. We can then rewrite the code to loops on x and y instead of i and j, obtaining the skewed routine from Table 6.1, second column. As one

Table 6.1 – Simple loop transformations

```
do i = 1,6
 do j = 1,5
  A(i,j) =
  A(i-1,j+1)+1
```

```
do x = 2,11
 do y = max(x-5,1),
        min(6,x-1)
  i = y
  j = x-y
  A(i,j) =
  A(i-1,j+1)+1
```

```
do x = low_x, upp_x
 low_y = max(a*x+b,
     cst)
 upp_y = min(c*x+d,
     cst)
 do y = low_y,upp_y
  i = e*x+f*y+g
  j = h*x+k*y+l
  A(i,j) =
  A(i-1,j+1)+1
```

can notice, the loop structure remained the same (except the loop bounds and the initialization code), despite the affine transformation that has been applied. By rewriting the loop bounds and the memory accesses as generic affine functions of the enclosing loop indices, we can build a skeleton from which an infinite number of parallelizing transformations can be applied, provided that the loop structure and the order of the statements remain unchanged, as shown in the third column of Table 6.1. At runtime, the coefficients of the linear functions computing the loop bounds and the original iterators are assigned values according to the affine transformation to be applied. Each set of coefficients is equivalent to a new polyhedral transformation. A skeleton could, for example, support loop skewing combined with loop interchange, in which the *the first loop level* is parallel, while another skeleton can combine the same two types of transformations with *the second loop level as parallel*. By assigning different values to the coefficients in the same skeleton, different skewed and/or interchanged loop versions can be obtained. Similarly, one can design new skeletons to support other classes of polyhedral transformations.

To be able to handle all types of loops in the same manner, being them for-, while- or do-while loops, we introduce the notion of *virtual iterators*. They are canonical iterators inserted in the loops, starting from 0 and incremented with a step of 1. They allow us to handle loops that originally did not have any iterators in the code and to apply polyhedral transformations. As an example, consider the loop nest in Table 6.2, column 1 (original version) and its equivalent form in column 2 with virtual iterators (sequential version in Fig. 6.3). The virtual iterators are part of our chunking mechanism, allowing the runtime system to control the number of executed iterations in each chunk, independently of the executed version's nature: instrumented, sequential or parallel.

To preserve the correct semantics of the original code and to perform the required speculation management tasks, the skeletons are completed with *guarding*, *initialization* and *verification code*, as shown in column 3 and detailed below. Additionally, skeletons include explicit calls to the GOMP/OpenMP library [55] to spawn parallel threads. All these code manipulations are performed at the intermediate representation level.

Table 6.2 – Simplified skeletons

```
while (p!=NULL){
 q = q0;
 while(q!=NULL){
  p->val += q->
      val;
  q = q->next;}
 p=p->next;}
```

```
for i = low_chunk,
          upp_chunk
 for j = 0, u*i+
     v
  if (p!=NULL)
   ...
   if (q!=NULL)
    ...
```

```
do x = lb_x , ub_x {
 do y = lb_y , ub_y {
  /* initialization code */
  p = a*x + b*y + c
  if (!(p!=NULL))
   rollback();
  else {
   q = q0;
   if (guarding code){
    /* initialization code */
    q = d*x + e*y + f
    if (!(q!=NULL)) {
     if (j ≠ αx + βy + γ)
      rollback();
     else continue;}
    else {
     p->val += q->val;
     q = q->next; }
    if (guarding code)
     p=p->next;
} } } }
```

6.3.1 Guarding code

Since any target loop nest is first transformed as a for-loop nest, the computation of the new loop bounds of the parallelized loop nest has to be done automatically at runtime, for any loop nest depth. This is classically done using the Fourier-Motzkin elimination algorithm [130]. For this purpose, we use an implementation of the algorithm available in the software library FM [115]. Note that the conditions of the original while loops are preserved in the code by copying the original loop bodies in the skeleton, thus ensuring that we do not execute mispredicted iterations. Similarly, we check that all iterations have been executed, by verifying that the exiting iteration, with respect to the sequential order, executes when predicted. We call this *guarding code* and it is aimed to verify our speculation on the loop bounds, as detailed in Sect. 6.3.3. The not perfectly nested instructions are embedded in conditionals which ensure that they are executed only at the right iterations. The *guarding code* is inserted in the innermost loop, thus allowing various affine transformations combined with loop interchange.

6.3.2 Initialization code

We use the linear functions obtained from the profiling phase, to initialize the basic scalars at runtime. In Table 6.1, the basic scalars are the original iterators i and j, preserved in the skeleton code to ease the computations, while in Table 6.2, the basic scalars are p, q and j: p and q correspond to the *phi*-nodes in the original code and

their values are employed in the computation of accessed memory locations, whereas j contributes to the computation of the exit conditions of the subloop.

Our value prediction mechanism is similar to the ones presented in [140, 119]. The initialization code is equivalent to *privatization*, since all values that depend on other iterations are re-declared locally in each thread and initialized using the predicting linear functions. Thus, the new shape of the loop nest complies with the polyhedral model and the loops can be further transformed as in the case of statically analyzable code, by applying an affine, unimodular transformation T. For example, for a loop nest of depth 2:

$$T \cdot \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \Leftrightarrow \begin{pmatrix} i \\ j \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

we obtain a new loop version in x and y. The bounds of the loops and the coefficients of the linear functions are assigned values dynamically. The coefficients of the linear functions are computed by applying the transformation matrix T on the predicting linear functions obtained from profiling and linear interpolation.

6.3.3 Verification code for speculative parallelization

Since all code transformations, including parallelization, rely on speculations, one must periodically check the predictions in order to validate/invalidate the execution. Not only we maintain the correct memory state, but we also transform the control flow of the loop, using new loop iterators and bounds.

The model we propose is based on the linear description of the memory accessing behaviour. Hence, our speculations consist of the linear functions that predict the memory addresses being accessed. Validating a transformation is equivalent to verifying the correctness of the interpolating linear functions. Under these circumstances, it suffices to compare the actual addresses being accessed, by the original instructions, to our predictions, given by the linear functions. Recall that the code inside the body of the loops in the skeletons is a copy of the original code. *Thus, the memory accesses are performed by a copy of the original memory instructions, whose target addresses are computed directly or indirectly from the basic scalars, which are initialized at each iteration.*

We divide the type of verification in three categories, depending on the instances being verified:

1. Basic scalars. When the execution of the iteration completes, we verify that the value computed by the code of the loop body and the value we predict coincide. For this verification, *we compare the actual value with the one expected for the next iteration according to the sequential order.* Validation of basic scalars ensures that all values computed in the loop reach the predicted values. Hence, the result of the dependence analysis is preserved as long as the interpolating linear functions used for initializations and verifications remain valid.

2. Memory accesses. Note that some iterations might execute before being validated by the preceding iteration according to the sequential order. Hence, one is re-

quired to verify all memory accesses performed in the current iteration to ensure that each targeted location has been correctly predicted. This has twofold consequences. First, it ensures that no invalid access is performed. And second, it guarantees that the memory state can be safely restored, as no modification outside the predicted memory is done. Although memory accesses are verified prior to being performed, indirect array addressing can still be handled by our system, since it is modelled as two separate memory accesses, each of them verified independently.

3. Loop bounds. The iteration counts of the subloops have a direct role in the polyhedral transformation being applied. The verification code relies heavily on the *guarding* code, presented previously. The transformed loop bounds control the number of iterations to be executed by each loop of the nest and together with the *guarding code*, it must be verified that: (i) each loop executes *all* its iterations, (ii) but no loop executes more iterations than it should. Due to the out-of-order execution of the iterations, the code must allow the execution of the last iteration of a loop (according to the sequential order) without exiting, as it might be followed by other iterations according to the parallel schedule. As an example, consider the transformed loop in the code snippet in Table 6.2. The bounds of the outermost loop cannot be predicted, therefore a rollback is triggered when the original outermost condition becomes false. In contrast, the subloops' bounds are interpolated, and, thanks to this prediction, the execution order of their iterations can be changed, while precisely controlling the loop exiting iterations.

6.4 Memory backup

As the execution is speculative, the framework integrates a system of commit/rollback, equivalent to the mechanisms employed in database. Before any execution of a parallel chunk, data are backed-up for the range of written predicted memory addresses. On rollback, data saved previously are propagated and overwrite any modification performed during the last chunk execution. After a valid chunk execution, previous backed-up data are dropped.

To compute the range of memory supposedly touched by the next chunk, one must determine the bounds. We respectively compute a minimum and maximum lower and upper bound for each loop level. In fact, the values are successively propagated in the bounds of the enclosed loop to compute the ranges of the iterators. Finally, we solve the affine access functions with the min/max values obtained for each iterator. In fact, the iterator stride is assumed to be monotonically increasing.

As memcpy is a highly optimized function, results may vary depending on memory address alignments, data to be copied, etc. As a consequence, the threshold for performing either just one call to copy a large array, or several calls to copy smaller chunks of memory, should be adjusted depending on the data layout for optimal results. We carried out tests on arrays of 100MB of the form:

|——|xx|——|xx|——|xx|——|xx|...|

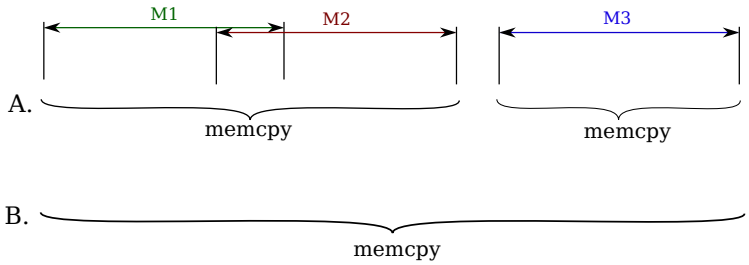


Figure 6.4 – memcpy merging strategy.

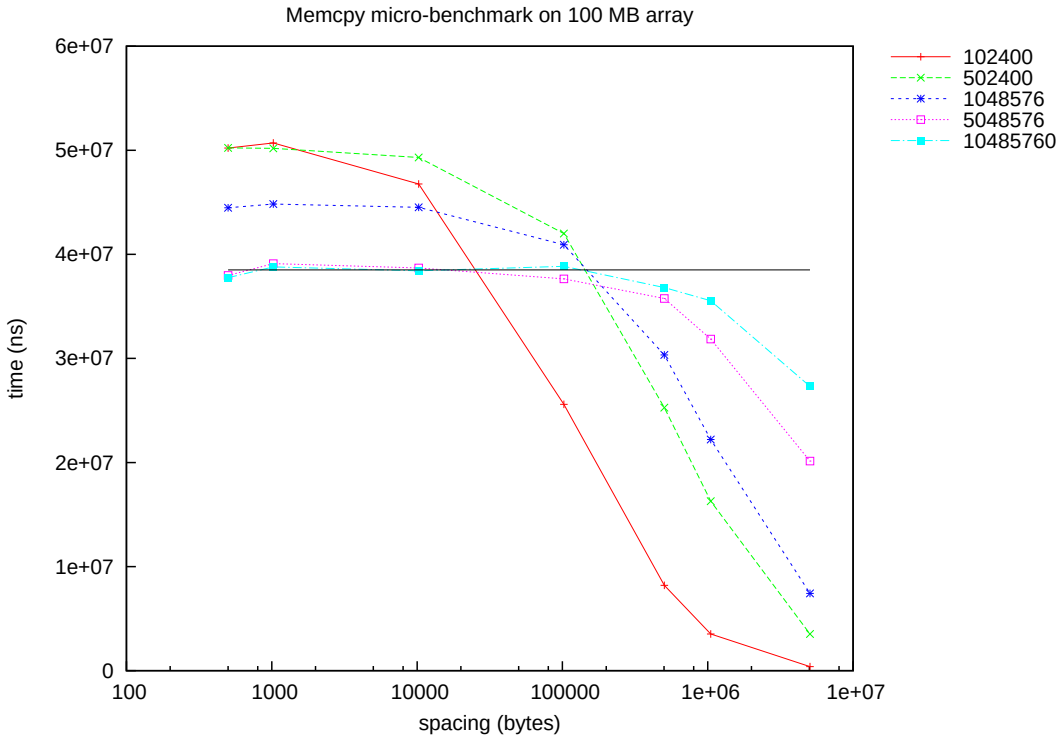


Figure 6.5 – memcpy behaviour on multiple spacing configurations, for different chunk sizes.

where a memory range is represented by: |——|, and the spacing between the ranges by |xx|. We are interested in determining the point at which it becomes interesting to fuse the memory copies. Figure 6.5 depicts the time to transfer chunks of fixed sizes (denoted by the different curves), when varying the spacing between two consecutive transfers. Performing numerous small memcpy significantly degrades the performance. The larger the chunk size, the closer is the performance to a full copy (cyan, violet) and inversely. The best performance are achieved for small chunks, with a wide spacing (for chunk sizes of 1KB, $\sim$ 5KB and 1MB). Through these empirical experiments we observed that it is worth performing a large memcpy as soon as the unnecessarily saved area is lower or equal to the total memory range to be copied. This strategy is illustrated in Fig. 6.4. For this purpose, the address ranges are sorted in ascending order, based on the lowest accessed address. The memory range is then traversed and interval between two accessed regions is calculated. Regions are fused based on the

Algorithm 6.1 VMAD decision module, backup management

```
function BACKUP( )
    l ← camus_min_max_loop_bounds()
    m ← camus_min_max_mem_access(l)
    m ← camus_sort_min_max(m)
    m ← camus_fuse_mem_access(m)
end function
```

aforementioned strategy. An overview of the stages executed by the decision module is presented in Algorithm 6.1. We vary the size of chunks and measure the time taken to transfer them using several memcpy calls. The black continuous line is the reference and it represents the time required to copy the entire array (chunks and spacing) in one memcpy.

6.5 Experimental results

In this section we present the experiments we conducted to evaluate our approach of applying the polyhedral model at runtime, in the view of speculatively parallelizing the loop nests. Our benchmarks were run on two architectures. The first platform embeds two AMD Opteron 6172, of 12 cores each, at 2.1 Ghz, running Linux 3.2.0-27-generic x86_64, while the second platform is an Intel Xeon X5650 at 2.67GHz, with 12 cores hyper-threaded, running Linux 3.2.0-24-generic x86_64. We have selected a set of benchmarks from different sources: the polyhedral benchmark suite [118], the Rosetta codes [125], the Rodinia benchmark suite [29] and the DSPstone benchmarks [42]. Notice that although some of these codes could have been handled statically, they are used to show the effectiveness of our system. Hence, we have modified them to use dynamically allocated arrays or pointers, which would prevent static analysis. Our measurements are given in Table 6.3. We compare the speed-up of our system for the target loop nests relatively to manual parallelization using OpenMP, when such a parallelization is possible, on both architectures, with 12 and 24 threads. Note that the speed-up obtained by OpenMP code is the highest that can be reached with straightforward parallelization (identity transformation matrix), since it does not require any dynamic analysis, dynamic code generation or support for speculative execution. In contrast, it places the burden on the programmer to analyze and parallelize the code, and to ensure its correctness. On the other hand, VMAD is entirely automatic and does not rely on any hardware support for speculative executions. Thus, it is readily applicable on any target architecture, merely by porting the runtime system, at the cost of an inherent overhead for offering a purely software support for speculative parallelization.

One of the interesting outcomes is that the behaviour of the same code on the two different architectures is very different, in terms of scalability with the numbers of threads. We observed that this behaviour is not specific to VMAD, as we obtained similar results when parallelizing the codes manually, with OpenMP. Each processor is better adapted for a particular type of applications, and the codes benefit differently

Table 6.3 – Code speculatively parallelized with VMAD, compared to OpenMP

Program	# threads	AMD Opteron 6172		Intel Xeon X5650		Polyhedral Transf.
		Speed-up VMAD	Speed-up OpenMP	Speed-up VMAD	Speed-up OpenMP	
adi	12	1.78	13.49	1.80	5.21	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
	24	1.82	13.34	4.09	4.75	identity 1^{st} loop par.
backprop	12	12.53	11.24	1.23	1.83	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
	24	15.62	17.86	1.86	2.05	interchange 1^{st} loop par.
cholesky	12	1.93	N/A	1.67	N/A	$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
	24	1.81	N/A	1.47	N/A	polyh. tr. 2^{nd} loop par.
floyd	12	0.77	N/A	0.73	N/A	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
	24	1.43	N/A	0.73	N/A	identity 2^{nd} loop par.
fir2dim	12	2.74	N/A	3.29	N/A	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
	24	2.61	N/A	2.93	N/A	identity 1^{st} loop par.
covariance	12	4.30	6.19	4.03	5.86	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
	24	7.45	12.07	4.55	8.92	identity 1^{st} loop par.
correlation	12	4.29	6.26	3.88	5.73	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
	24	7.47	12.18	4.64	8.55	identity 1^{st} loop par.
qr_decomp	12	2.87	12.02	2.11	11.03	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
	24	4.69	20.02	2.48	12.22	identity 1^{st} loop par.
grayscale	12	1.81	8.73	1.13	3.59	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
	24	2.03	6.61	1.25	2.28	identity 1^{st} loop par.

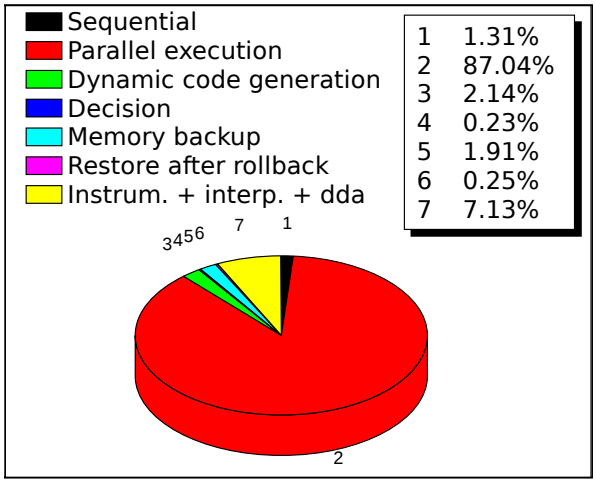


Figure 6.6 – Runtime overhead of covariance

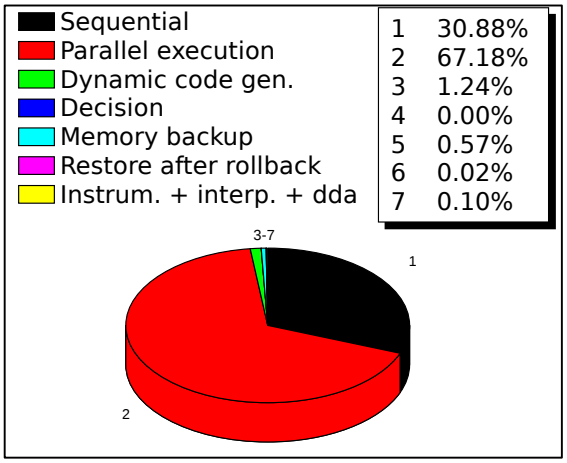


Figure 6.7 – Runtime overhead of backpropagation

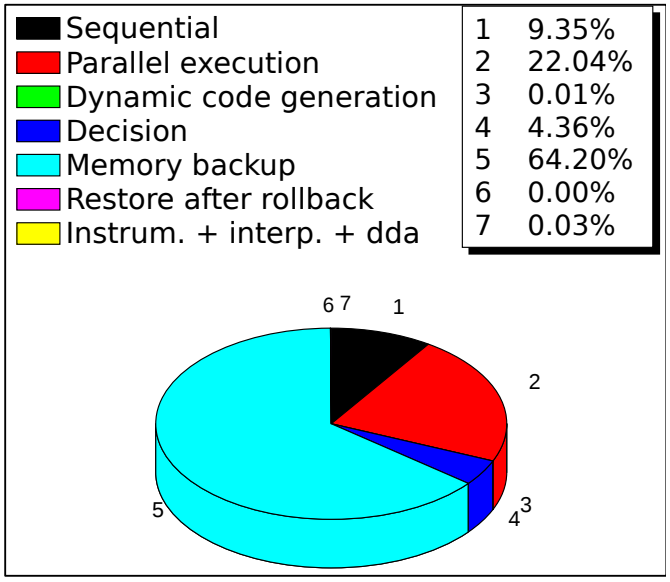


Figure 6.8 – Runtime overhead of adi

from the hardware support such as the hardware prefetcher, the branch predictor, etc.

Additional experiments with a varying number of threads showed that even without any transformation other than straightforward parallelization of the outermost loop, our system outperforms OpenMP, thanks to the execution in chunks of the loops, which is similar to strip-mining, having a positive effect on data locality (in Table 6.3 we show only the results with 12 and 24 threads). With `adi` on the Intel processor, the execution with 16 threads using our system is slightly better than the code parallelized with OpenMP. However, the benefits of chunking and parallelization are hidden by the overhead when running `adi` on the AMD processor. We analyzed the overhead and concluded that it stems from copying the data prior to parallelization using `memcpy`. More details on the overhead of our system are given in the end of this section.

An interesting example is `backprop`, which can be parallelized in its original form using OpenMP. This code is handled similarly by any traditional TLS system, by parallelizing the outermost loop. On the other hand, our system discovers that a loop interchange is possible, which brings significant performance gains, by improving data locality. This benchmark underlines the fact that our system can significantly improve embarrassingly parallel codes, unlike traditional TLS systems, even outperforming manual parallelization. Additionally, it shows that the runtime overhead of the system is hidden by the gains provided by applying the polyhedral transformation.

Another example highlighting this contribution to the state of the art is `cholesky`, which is not parallel in its original form. Therefore, previous TLS systems cannot handle this code, nor can it be manually parallelized with OpenMP, since every loop carries dependences. In contrast, our system analyses the runtime behaviour of the code and finds a suitable polyhedral transformation which allows the loop from the second level in the transformed code to be executed in parallel, as shown in the last column of Table 6.3. Performance can be further improved by generating a skeleton handling tiling transformations, which is one of the first targets of our future work.

The example `floyd` illustrates the capability of our system to adapt dynamically to the behaviour of the code and to exploit partial parallelism. This benchmark embeds a conditional that does not allow parallelization because it does not have a predictable behaviour during the first iterations. Nevertheless, our system executes a sequential chunk and monitors again the loop. The second profiling phase identifies that one branch of the conditional is now always executed, and parallelization becomes possible. Moreover, the result of the dependence analysis indicates the second loop level as being parallel, which is not handled by traditional TLS systems. Although the inherent overhead of the system stemming from the support for speculative execution hides the benefits of parallelization, the benchmark is a suitable candidate to exhibit VMAD's capabilities of performing partial parallelization. We detail on the sources of overhead in the end of the section, emphasizing the main bottlenecks and suggestions on how the penalties can be reduced.

The benchmark `fir2dim` contains a loop nest of depth 3 performing memory accesses via pointers. Arrays are represented as dynamically allocated pointers and their parsing is performed by using pointer arithmetic. OpenMP fails to parallelize such codes, due to the impossibility of predicting the starting value of the pointers for each thread. On the contrary, our system is successful in parallelizing these examples, thanks

to its instrumentation phase, which builds interpolating linear functions.

Other examples, such as `covariance` and `correlation` show that codes parallelized with our system have a good overall performance. Nevertheless, most of the figures indicate that the speed-up could be considerably improved by reducing the overhead, thus we plan to revise some of our implementation and design choices.

Various strategies could be employed in order to validate or guide the optimizations applied dynamically, such as guarding the loop nests with tests and proposing one code version or another, depending on the some key values unknown statically. However, we aimed to design a more general approach for codes on which such conditionals would not suffice. Codes such as `fir2dim`, `qr_decomp` and `grayscale` use dynamic allocation and pointers that cannot be handled properly using pointer analysis, since memory access functions depend on the input size unknown at compile-time, thus making them non-linear, but linear at runtime. The same issue about non-linearity arises with `cholesky`. Hence, accurate static dependence analysis is impossible. A conditional code would even correspond to something close to our system itself. For `backprop`, loop interchange is not always beneficial, depending on the size parameters translating to loop bounds. Flexibility is required. Our system is adapted to component programming and the use of library code which are becoming prevalent, and not in favor of pointer analysis requiring the whole program. Also, spurious and infrequently occurring dependences are treated conservatively with static analysis, to produce sound results across all inputs. It could not handle codes like `floyd`. All these facts argue for our dynamic approach for maximizing parallelism in the multicore era. We ran the well-known polyhedral static parallelizer PLUTO [25] on the benchmark codes when possible (for-loops, linear array accesses): on `cholesky` –that had to be rewritten with linear accesses–, we outperform PLUTO thanks to our chunking system; PLUTO is not exchanging loops in `backprop`; similar execution times are obtained with the other codes, except `adi`, where our system is outperformed.

The systems's runtime overhead As expected, the overhead's impact strongly depends on the characteristics of the code, since it is relative to the time of the total execution. Thus, for loop nests in which the outermost loop has a large number of iterations, the profiling phases, consisting in instrumentation, interpolation of memory accesses and dynamic dependence analysis, have almost a negligible overhead. In practice, we noticed that in many situations, this overhead did not pose significant problems. In Fig. 6.6, 6.7 and 6.8 we depict the time taken by each phase of executing codes with our system, relative to the total execution. The `Sequential` phase refers to the execution of the last chunk, which is always run sequentially, to ensure that all iterations of the loop were executed. `Dynamic code generation` is the time taken to specialize the skeletons, using runtime information. `Decision` is the time taken by the runtime system to select the code version run by the next chunk and to set its size. `Memory backup` is the safe copy that is performed before launching a speculative chunk. `Restore after rollback` is the time required to restore the correct state of the memory, upon a misprediction. And finally, the time taken to instrument the code, interpolate the results and run the dynamic dependence analysis is depicted as `Instrumentation + interpolation + dda`. In Fig. 6.6, 6.7 and 6.8, the pie chart

illustrates the total execution time divided in the time taken by each action, when running on the AMD Opteron 6172. One can notice that dynamic code generation is negligible, which argues in favor of using binary skeletons. Similarly, the overhead incurred by instrumentation and dependence analysis is minimal, thanks to our sampling mechanism. Nevertheless, our strategy to back-up memory is costly, and requires further refinements. A first improvement would be to include back-up each location independently prior to be speculatively accessed. This strategy would parallelize the memory back-up process and thus provide better performance.

6.6 Memory backup extensions

We may possibly backup too many data for certain access function, for instance, column-major access function. The problem is illustrated in some toy accesses, in Samples 6.1, 6.2, 6.3. In fact, the backup to be performed may not always be applied on dense memory ranges. To handle that issue, we use a recursive approach which computes all the access ranges for the next chunk to be executed. The affine access function is instantiated with the min/max values obtained in the decision module. The computed list of ranges is then processed by our range fusion heuristic in order to perform more efficient memcpy. The resulting fused ranges are copied by one of our implementation of memcpy. A decision mechanism based on empirical experiments, chooses the right function to execute depending on the memory area to backup: the basic memcpy, a vectorized version, or a parallel OpenMP version. However, experiments have shown that most of the time the standard memcpy was outperforming an SSE version of memcpy on our test platform.

$$\begin{matrix} [0][0] & [0][1] & [0][2] & [0][3] \\ [1][0] & [1][1] & [1][2] & [1][3] \\ [2][0] & [2][1] & [2][2] & [2][3] \end{matrix} \tag{6.1}$$

$$[0][0] \quad [0][1] \quad [0][2] \quad [0][3] \quad \ldots \quad [2][0] \quad [2][1] \quad [2][2] \quad [2][3] \tag{6.2}$$

$$[0][0] \quad [0][1] \quad [0][2] \quad [0][3] \quad \ldots \quad [2][0] \quad [2][1] \quad [2][2] \quad [2][3] \tag{6.3}$$

6.7 GPU extension proposal

VMAD already tackles multi-core CPUs by speculatively parallelizing sequential codes. In this section we propose some directions to enable efficient speculative parallelization on GPUs. In fact, targeting GPUs generally requires drastic transformations for a code to be efficient.

6.7.1 Verification code

To validate the semantics of the parallel chunks while ensuring performance, one has to decide of the adequate method. Two interesting techniques may be employed in

order to handle code verification on GPU. Note that since parallel chunks are run on the GPU in a different address space, asynchronous pointer modifications are hardly noticeable.

The first possibility is to generate a GPU-specific skeleton that would have the same characteristics as the CPU skeletons. In that case, the verification code is embedded into the kernel. At each iteration, before any memory access, the address must be verified to be conform to predictions. Particular attention should be paid to the memory address space as pointers do not lie in the same memory. In case the arrays are allocated on the GPU, one should translate the original CPU base addresses, in the verification code. Scalars are not directly impacted by this policy as they can be treated as conventional values on the GPU. Any unpredicted memory access could be followed by a *trap* instruction or equivalent, terminating the kernel execution (see Section 5.1). Observe that notifying the GPU threads could also be achieved by injecting a guard polling on a volatile variable residing in global memory. The high register requirement would ineluctably limit the number of active threads, due to potential exhaustion of the register file. To recall the reader, GPUs rely on parallelism in order to hide memory latencies. Also spill code, may drastically impact the performance of the skeletons. A high register consumption therefore puts a limit on the number of blocks simultaneously executed by a multi-processor and exerts pressure on the memory subsystem.

As an alternative, the inspector/executor scheme could become handy, for several reasons we describe in the following. The inspector thread is run on the CPU beforehand, possibly asynchronously to the execution on a GPU of a controlled chunk. A valid verification flags the chunk so that it is marked as ready to execute. As only memory accesses and scalars are checked, we can reasonably assume verification is performed quickly. The moderate impact of such a mechanism would still allow to concurrently run computations on CPU. First of all, this would drastically simplify GPU code generation. In particular, it would not require an address translation mechanism to compute the address in the different memory spaces. Many additional verification instructions, required for code checking, could be dropped, to release pressure on the registers. Running the verification beforehand also allows to determine the number of iterations.

6.7.2 Memory address space

In the presented speculative environment, handling the address space and data movement is crucial in order to run codes on GPU. Since the discrete GPU memory resides in an address space different than that of the CPU, multiple solutions are at disposal for data movements. The memcpy-based backups have already been proved useful in Jimborean et al. [65] work, Pradelle et al. [121] work and Section 6.4. In the case of a GPU-only execution, memory should directly be sent to GPU as no backup is necessary if the central memory stays untouched. In fact, restoring the memory simply consists in avoiding to fetch the computation from the GPU. In case of a CPU + GPU computation, the backup copy should be kept in the CPU central memory, as it ensures faster recovery in case of a rollback.

With zero-copy memory, GPUs can directly access the host central memory. While

this technique simplifies code generation, it may significantly impact performance. This technique possibly works for computation-bound kernels which expose a high reuse potential. Also, through experiments, we noticed that latency of Direct Memory Access (DMA) central memory from GPU drastically increases as the number of thread requests increase (see Section 5.1). More reliably, performing a memcpy on the area of the presumably touched arrays seems to be more adapted to our performance requirements.

To synthesize, the address ranges computed by the decision module must be copied to the GPU before the execution, for performance purpose. The coefficients of the linear functions, patched during execution, can also be copied using the same mechanics. Note that since CPU and GPU memory reside in different address spaces it is a requirement to translate the address in the case of the first verification procedure presented in 6.7.1.

6.7.3 Finding adequate block size

To handle the problem of finding adequate block size, one can use parametric tiling and map the tiles to the CUDA blocks. In this manner, one iteration of the parallel loop is attached to one CUDA thread. The chunk size should be adjusted in order to be a multiple of the grid size, while the block size should be a multiple of the warp size, *i.e.* 32 threads in current hardware. For small chunks, the blocks should at least contain several warps, to hide latency; in that case the minimum block size should be of 64 threads. Finding the optimal block size is difficult in the general case due to side effects on GPU latency hiding mechanisms and to a lesser extent, hardware constraints. Actually, to perform reliable choices the hardware should be loaded with a sufficient number of blocks. This is not always possible, depending on the problem size. Size of the blocks could be readjusted several times, until a good size is found. To accelerate the process, a performance model could predict well performing block sizes based on a few experiments, until convergence.

Note that it is possible to generate multiple tiled skeletons statically, with fixed tile size, and select the best one at runtime, after a small *offline* profiling as described in Section 4.

6.7.4 CPU + GPU execution

To dynamically handle load balance in a CPU + GPU context, an inspiring method is described in [22] and fits the execution model of VMAD. A first chunk, which size is defined arbitrarily is run on CPU and GPU. The execution time of the chunks may be modelled by a logarithmic regression, for instance. In fact, to make the chunk-sizing steps profitable, one can fit the execution times with a logarithm function to train the performance model. The training phase is considered valid, if deviation is inferior to a defined threshold, and should be limited by a certain number of attempts. With the help of the logarithmic regressions one can find a chunk size which is predicted to perform well. Handling load balance is tricky as the total number of iterations of the parallel loop is unknown. To ensure load balancing, chunks should be assigned in a task-based fashion to PUs; the faster PUs get more tasks so that the execution time is

equally distributed between slower and faster PUs. To achieve maximum throughput while ensuring load balance, the number of chunks assigned to one or the other PUs may for instance be based on the worst PU.

Note that if the number of iterations is known in advance, the CPU + GPU technique presented in Section 5.3 may be applicable. Again, the size of the chunks can be adjusted based on the worst PU.

6.8 Conclusions and perspectives

In this Chapter, we showed a new use of algorithmic skeletons, as an efficient support for speculative and dynamic parallelization, and proved that they can be automatically generated and then specialized to target codes and cover a large class of advanced runtime code transformations at a low cost.

Thanks to this automatic skeletal parallelization, VMAD provides important contributions and advancements to the state of the art and is successful in optimizing and parallelizing scientific kernels, that are not accessible to traditional TLS systems or to static analysis. The system can handle codes in any form and is not hindered by the type of memory allocation, being capable of handling pointers, static or indirect array accesses, multi-dimensional or linearized arrays, or any types of linked data structures, as soon as a linear memory behaviour has been detected in some execution phases. Moreover, unlike OpenMP, we can handle multiple exit loops and pointer-chasing loops. We conclude by reminding the main contributions, underlined by the benchmarks.

1. VMAD is able to automatically parallelize codes which do not exhibit parallelism in their original form. Thus, they cannot be handled efficiently by existing TLS systems (due to numerous rollbacks) and cannot be parallelized manually, unless they are transformed.

2. VMAD can discover optimization opportunities in codes that can already be parallelized in the original form. By applying such optimizing transformations prior to parallelization, the performance of the generated code can be significantly boosted.

3. The overhead of our system can be masked by the performance improvements provided both by parallelization and by the optimizing polyhedral transformations.

In the near future, we plan to extend the use of skeletons in order to provide more freedom to the runtime system on the kind of optimizing and parallelizing transformations. It will consist in building elementary skeletons at compile-time that will be assembled at run-time, following an enclosing algorithmic skeleton associated to a specific class of transformations. Transformations that change significantly the structure of the original sequential code will then also be handled efficiently by the system.

Moreover, since the framework follows a modular approach, it can be easily extended to target new types of optimizations (e.g. vectorization) following the results of analysis phases.

To overcome the main flaws of VMAD, the original speculation system has been redesigned. Transformations were dynamically selected within a set of transformations at disposal. The system was choosing the first legal transformation in that set. The order of the available transformation matrices had an influence on the performance.

Also, in its preliminary design VMAD was not properly handling imperfectly nested loops, this is now solved. Latest extensions by Sukumaran Rajam et al. [135], allow to handle non-linear codes, by introducing the notion of relaxed polytope. Linear regression functions are used to get a peek of the memory access behaviour and allow transformations to be performed by PLUTO. Future improvements include speculative execution on GPU and other accelerators (Xeon Phi, MPPA, etc.).

Chapter 7

Conclusion

7.1 Contributions

Our main contribution is a multiversioning mechanism that targets GPUs. The selection between multiple versions is based on an accurate prediction of the execution times of programs on GPUs. This prediction method takes the raw kernel execution time into account. Discrete GPUs need to be addressed specifically, as they require data to be transferred; *i.e.* they operate on a different memory space than central memory. The bandwidth of the communications is modelled through a micro-benchmark. Message size and transfer direction are the important parameters.

We successfully used this method in order to perform code selection. This mechanism allows to select the best code version of a loop nest and runs it on GPU. To achieve that, prediction accuracy is crucial in order to perform the right decision. The profiler performs several measurements, based on empirically determined performance characteristics. The profiling stage performs executions of the code on the target machine, in order to build *ranking tables*. They are used as a stub to predict execution times at runtime, when the execution context is known. Overall, the results are good, the system is accurate in most cases.

- In contrary to iterative compilation techniques which require to explore significant parts of the optimization space, our technique is capable to pick the best performing version at runtime, from a batch of limited number of versions, with very low overhead. Although, the versions are generated by hand for now (*i.e.* by varying compiler options), compilers could eliminate bad-performing versions at code generation, or generate different versions known to perform well in various environments.

- The profiler provides a generic way to predict execution times. This is an interesting point since analytic methods require modifications in order to be reliable for the next generation devices. In fact, it does not rely on any hardware counter and does not make any assumption tightly coupled to hardware. We make the hypothesis that the same technique would work similarly for hardware branded by other manufacturers. Also, we believe that this technique could be adapted to characterize MPPA processors for instance.

- The added-value of this approach is adaptation to some dynamic internal performance factors. In fact, the code versions expose different optimizations which behave differently depending on the input dataset. The advantage of hybrid methods is to bring the most of the work to the offline stage, to relieve the runtime, while keeping runtime-dependent considerations.

- The regularity of polyhedral codes facilitates execution time prediction. In fact, execution path is not driven by data values. Also, they are candidates to analytically compute the number of iterations, through *Ehrhart* polynomials. This allows to quickly compute an execution time prediction at runtime, lowering the overhead.

- The framework is entirely automatic and runs transparently to the user. Profiling and target application codes are generated by an automatic compiler. The technique could be easily plugged to another compiler, as long as it generates the required information. Also, an attempt could be made to handle more general codes, which behave regularly, but do not suit polyhedral analysis. Finally, linear algebra libraries could benefit from such a system, since it could embed the selection mechanism, to handle newer hardware architectures.

We developed a CPU vs GPU technique which runs the code simultaneously on both a multicore CPU and a GPU. This method is well suited to handle dynamic performance fluctuations.

- We designed a dynamic method, capable to select the best architecture, by running the code simultaneously on multiple PUs. Such a full-dynamic system is able to quickly adapt to uncontrollable dynamic performance factors. This contrasts to other CPU vs GPU systems, which generate an history of the execution times, and rely on that to dispatch computations. All in all, we noticed that running both the codes is in general less power-hungry than running the code on the slowest PU.

- We have provided a method to terminate running kernels on a CUDA GPU, as the CUDA API does not expose that feature. Also, we provide a way to terminate an OpenMP execution, although it is not portable.

- After running the code on the PUs, the fastest PU and problem size should be paired and recorded. This doublet could be used later, so that the right PU is automatically selected in future executions. However, this would lead the method to loose part of its dynamic advantage. To partially handle that case, execution times should be updated and compared after each execution.

- This dynamic method could be used as a basic block to more sophisticated systems. In general, history-based load balancing methods run the code once on each architecture, to train their model. Our technique would guarantee the best execution times, even for the first execution. In other systems, such as the one presented in Chapter 6, codes could be run on both architectures simultaneously.

Around the prediction method we designed an algorithm to distribute work on the available multicore CPUs and GPUs. The outermost parallel loop is split into chunks. The chunks are associated to the available PUs. The algorithm manipulates the size of the chunks in order to balance execution time. To handle dynamic parameters, the algorithm recomputes predicted execution times after each step.

- We developed a scheduler that is capable to handle more than two PUs, in contrary to many tools presented in literature which are limited to 1 CPU + 1 GPU. This is made possible, thanks to the converging nature of the algorithm, which relies on affine functions to determine the chunk size, at each step, for all the PUs.

- The scheduler is not only capable of balancing the load to multiple PUs, it also has a mechanism to eliminate predictably non-efficient PUs. The fastness of the scheduling algorithm allows multiple calls to be performed consecutively, in order to handle multiversioning. The low overhead at runtime allows to confront combination of versions, ones oriented towards CPU, the others towards GPU, and select the fastest.

- This technique is a proof-of-concept: we demonstrate that our GPU prediction method and Pradelle et al. CPU prediction method [120] that were proposed, are sufficient to achieve acceptable load balance in a CPU + GPU context. More than to select the best code version, a good accuracy is mandatory to compare execution times of each architectures.

- We demonstrate that the scheduler can easily be parametrized by energy constraints.

- With the scheduler algorithm we provide a way to execute polyhedral codes cooperatively on CPU + GPU. This is done entirely transparently, from code version generation, to version combination selection. To our knowledge no such a method was proposed yet.

7.2 Perspectives

Future works of this thesis include refinement of the code execution time prediction technique. The profiling may handle the loop bounds independently in order to provide a per-statement accuracy, rather than an average of the execution time of the statements. We are confident that this technique is suited to other architectures, such as MPPA. The efforts should especially be oriented towards portability across GPU brands, especially with the addition of OpenCL code generation in PPCG. Eventually, a work should address the lack of multiversioning features in current state of the art compilers.

We proposed a completely dynamic method to handle the CPU vs GPU case. The two main concerns of the presented technique are energy and system overload. Tools such as HDSS and OmpSs tend to consider partial runtime executions. This may

be sufficient to characterize the code to handle the heterogeneous case and account external performance factors. We also showed an hybrid CPU vs GPU technique, based on CPU and GPU execution time predictions. However current state of the art tools focus on the joint use of CPU and GPU. We do not completely exclude this method as it may be effective in the case of a task based system.

Finally, we demonstrate an hybrid CPU + GPU technique which makes use of both CPU and GPU to perform a calculation. To reduce load imbalance, accuracy in the CPU prediction technique should be improved. Investigations in energy consumption strategies may help to reduce power consumption through better decision making. Also, the algorithm should be evaluated on systems with several more PUs to GPU powered supercomputers in order to demonstrate its scaling capability.

All in all robustness of the implemented tools should be improved in order to provide a safe and user friendly way to benefit from CPU + GPU systems.

Bibliography

[1] CUB. `http://nvlabs.github.io/cub/`.

[2] CUDA data parallel primitives library. `http://cudpp.github.io/cudpp/2.0/`.

[3] The green500 list. `http://www.green500.org`, June 2014.

[4] The top500 list. `http://www.top500.org`, June 2014.

[5] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. *SIGARCH Comput. Archit. News*, 40(1):61–74, March 2012.

[6] C. Akel, Y. Kashnikov, P. de Oliveira Castro, and W. Jalby. Is source-code isolation viable for performance characterization? In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 977–984, Oct 2013.

[7] Mehdi Amini. *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 2012.

[8] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPS is not (just) polyhedral software adding GPU code generation in PIPS. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO 2011*, Chamonix, France, April 2011. 6 pages.

[9] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[10] M. Arora, S. Nath, S. Mazumdar, S.B. Baden, and D.M. Tullsen. Redefining the role of the CPU in the era of CPU-GPU integration. *Micro, IEEE*, 32(6):4–16, Nov 2012.

[11] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Delft, Netherlands, August 2009.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *EuroPar 2009*, LNCS, Delft, Netherlands, 2009.

[13] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna, Austria, July 2010.

[14] Lénaïc Bagnères and Cédric Bastoul. Switchable scheduling for runtime adaptation of optimization. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 222–233. Springer International Publishing, 2014.

[15] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. 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*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] C. Bastoul. Chunky ANalyzer for Dependences in Loops. `http://icps.u-strasbg.fr/~bastoul/development/candl/`, 2008. Related to the candl tool.

[17] C. Bastoul. Extracting polyhedral representation from high level languages. Technical report, LRI, Paris-Sud University, 2008. Related to the Clan tool.

[18] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Cédric Bastoul. Improving data locality in static control programs. *Thèse de doctorat, Université Paris 6*, 2004.

[20] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, Texas, october 2003.

[21] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems*, 7, 2011.

[22] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.

[23] E. Bendersky. pycparser. `https://github.com/eliben/pycparser`, 2010.

[24] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[25] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*, pages 101–113. ACM, 2008. http://pluto-compiler.sourceforge.net.

[26] Uday Kumar Reddy Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, The Ohio State University, 2008.

[27] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 21:1–21:10, New York, NY, USA, 2013. ACM.

[28] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

[29] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE, 2009.

[30] Long Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[31] Marcelo Cintra and Diego R Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *ACM SIGPLAN Notices*, volume 38, pages 13–24. ACM, 2003.

[32] Albert Cohen, Sylvain Girbal, and Olivier Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par 2004 Parallel Processing*, pages 292–303. Springer, 2004.

[33] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[34] Damien Couroussé, Victor Lomüller, and Henri-Pierre Charles. Introduction to Dynamic Code Generation: An Experiment with Matrix Multiplication for the STHORM Platform. In Massimo Torquati, Koen Bertels, Sven Karlsson, and François Pacull, editors, *Smart Multicore Embedded Systems*, pages 103–122. Springer New York, 2014.

[35] Huimin Cui, Lei Wang, Jingling Xue, Yang Yang, and Xiaobing Feng. Automatic library generation for BLAS3 on GPUs. In *Parallel Distributed Processing Symposium (IPDPS), 2011, IEEE*, pages 255–265.

[36] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[37] Hervé Deleau, Christophe Jaillet, and Michaël Krajecki. GPU4SAT: solving the SAT problem on GPU. In *PARA 2008 9th International Workshop on State–of–the–Art in Scientific and Parallel Computing, Trondheim, Norway*, 2008.

[38] Peng Di and Jingling Xue. Model-driven tile size selection for doacross loops on gpus. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 401–412. Springer Berlin Heidelberg, 2011.

[39] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating GPU binaries to tiered SIMD architectures with ocelot. 2009.

[40] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 353–364. ACM, 2010.

[41] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[42] DSPstone benchmarks. `http://www.ice.rwth-aachen.de/research/tools-projects/entry/detail/dspstone/`.

[43] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[44] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[45] P. Feautrier. Array expansion. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, pages 429–441, New York, NY, USA, 1988. ACM.

[46] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[47] Paul Feautrier. Some efficient solutions to the affine scheduling problem part ii multidimensional time, 1992.

[48] Paul Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *Proceedings of the 7th international conference on Supercomputing*, pages 175–184. ACM, 1993.

[49] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005*, pages 29–46. Springer Verlag, 2005.

[50] Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, D. Malony, Allen, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. Collective mind: Towards practical and collaborative auto-tuning. *Scientific Programming*, 22(4):309–329, July 2014.

[51] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE micro*, (4):13–27, 2008.

[52] GNU compiler collection. `https://gcc.gnu.org`.

[53] P. Gepner and M.F. Kowalik. Multi-core processors: New way to achieve high system performance. In *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, pages 9–13, Sept 2006.

[54] Sayan Ghosh, Terrence Liao, Henri Calandra, and Barbara M Chapman. Experiences with OpenMP, PGI, HMPP and OpenACC directives on ISO/TTI kernels. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 691–700. IEEE, 2012.

[55] GOMP — An OpenMP implementation for GCC - GNU Project. `http://gcc.gnu.org/projects/gomp`.

[56] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 2. IEEE Press, 2008.

[57] Dominik Grewe and Michael FP O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Compiler Construction*, pages 286–305. Springer, 2011.

[58] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012.

[59] Daniel Hefenbrock, Jason Oberg, Nhat Thanh, Ryan Kastner, and Scott B Baden. Accelerating Viola-Jones face detection to FPGA-level using GPUs. In *FCCM*, pages 11–18, 2010.

[60] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[61] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[62] D. Horn. Chapter 36. Stream reduction operations for GPGPU applications. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*. 2005.

[63] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 5th international conference on Supercomputing*, pages 244–251. ACM, 1991.

[64] Ian Lane J. Kim, J. Chong. HYDRA: a hybrid CPU/GPU speech recognition engine for real-time LVCSR. In *GPU Technology Conference*, 2013.

[65] Alexandra Jimborean. *Adapting the polytope model for dynamic and speculative parallelization*. PhD thesis, Strasbourg, 2012.

[66] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[67] Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo, and Vincent Loechner. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *PPoPP '12*, 2012.

[68] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. VMAD: An advanced dynamic program analysis and instrumentation framework. In Michael O'Boyle, editor, *Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 220–239. Springer Berlin Heidelberg, 2012.

[69] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP '07*. ACM, 2007.

[70] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *arXiv preprint arXiv:1005.2581*, 2010.

[71] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99. ACM, 2013.

[72] Minhaj Ahmad Khan, H.-P. Charles, and D. Barthou. Improving performance of optimized kernels through fast instantiations of templates. *Concurr. Comput. : Pract. Exper.*, 21(1), January 2009.

[73] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. Seamlessly portable applications: Managing the diversity of modern heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 8(4):42:1–42:20, January 2012.

[74] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. Automatic speculative DOALL for clusters. In *CGO '12*. ACM, 2012.

[75] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[76] A. Kloeckner. islpy. `https://pypi.python.org/pypi/islpy`, 2011.

[77] Toshiya Komoda, Shinobu Miwa, Hiroshi Nakamura, and Naoya Maruyama. Integrating multi-GPU execution in an OpenACC compiler. In *42nd International Conference on Parallel Processing - ICPP*, Lyon, France, 2013. IEEE.

[78] Christian Lauterbach, Qi Mo, and Dinesh Manocha. gProximity: Hierarchical GPU-based operations for collision and distance queries. In *Computer Graphics Forum*, volume 29, pages 419–428. Wiley Online Library, 2010.

[79] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[80] Sang-Ik Lee, TroyA. Johnson, and Rudolf Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 539–553. Springer Berlin Heidelberg, 2004.

[81] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[82] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61. ACM, 2010.

[83] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.

[84] Chong Li, Frédéric Gava, and Gaétan Hains. Implementation of data-parallel skeletons: A case study using a coarse-grained hierarchical model. In *ISPDC*, pages 26–33, 2012.

[85] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning GEMM for GPUs. In *Computational Science–ICCS 2009*, pages 884–892. Springer, 2009.

[86] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a tls compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167. ACM, 2006.

[87] Wei Liu, James Tuck, Luis Ceze, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A profiler-enhanced TLS compiler that leverages program structure.

[88] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[89] LLVM compiler infrastructure. `http://llvm.org`.

[90] Vincent Loechner. PolyLib: A library for manipulating parameterized polyhedra. `http://icps.u-strasbg.fr/polylib/`, 1999.

[91] Victor Lomüller and Henri-Pierre Charles. Speculative runtime parallelization of loop nests: Towards greater scope and efficiency. *17th Workshop on Compilers for Parallel Computing*, July 2013.

[92] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[93] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. GROPHECY: GPU performance projection from CPU code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 14:1–14:11, New York, NY, USA, 2011. ACM.

[94] Quirin Meyer, Fabian Schonfeld, Marc Stamminger, and Rolf Wanka. 3-SAT on CUDA: Towards a massively parallel SAT solver. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 306–313. IEEE, 2010.

[95] Paulius Micikevicius. GPU performance analysis and optimization. In *GPU Technology Conference*, 2012.

[96] Chuck Moore. Data processing in Exascale-class computing systems. In *The Salishan Conference on High Speed Computing*, April 2011.

[97] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[98] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: implementation and experimental study. In *Int. Conf. on Computer Languages*. IEEE Computer Society Press, 1998.

[99] Nouveau driver. `http://nouveau.freedesktop.org/wiki/`.

[100] Cedric Nugteren and Henk Corporaal. The boat hull model: adapting the roofline model to enable performance prediction for parallel computing. In *ACM Sigplan Notices*, volume 47, pages 291–292. ACM, 2012.

[101] Cedric Nugteren and Henk Corporaal. Introducing Bones: a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 1–10. ACM, 2012.

[102] Nvidia. CUDA: performance of applications. `http://www.nvidia.com/object/gpu-applications.html`.

[103] Nvidia. CUDA: Compute Unified Device Architecture. `http://www.nvidia.com/object/cuda_home_new.html`, 2007.

[104] Nvidia. Nvidia Fermi GF100 whitepaper. `http://www.nvidia.com/object/IO_86775.html`, 2010.

[105] Nvidia. Nvidia Kepler GK110 whitepaper. `http://www.nvidia.com/content/PDF/kepler/Nvidia-Kepler-GK110-Architecture-Whitepaper.pdf`, 2010.

[106] Nvidia Corporation. cuBLAS-XT. `https://developer.nvidia.com/cublasxt`, 2014.

[107] OpenACC corporation. OpenACC. `http://www.openacc-standard.org/`, 2012.

[108] R. Westermann P. Kipfer. Chapter 46. Improved GPU sorting. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*. 2005.

[109] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[110] Krishna Palem and Avinash Lingamneni. What to do about the end of Moore's law, probably! In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 924–929, New York, NY, USA, 2012. ACM.

[111] R.A. Patel, Yao Zhang, J. Mak, A. Davidson, and J.D. Owens. Parallel lossless data compression on the GPU. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9, May 2012.

[112] Martin Peres. Reverse engineering power management on Nvidia GPUs-anatomy of an autonomic-ready system. In *ECRTS, Operating Systems Platforms for Embedded Real-Time applications 2013*, 2013.

[113] J. Planas, R.M. Badia, E. Ayguade, and J. Labarta. Self-adaptive OmpSs tasks in heterogeneous environments. In *IEEE 27th Int. Symposium on Parallel Distributed Processing (IPDPS)*, pages 138–149, 2013.

[114] Sébastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. 2006.

[115] Louis-Noël Pouchet. FM: the Fourier-Motzkin library. `http://www.cse.ohio-state.edu/~pouchet/software/fm`.

[116] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Notices*, volume 43, pages 90–100. ACM, 2008.

[117] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, P Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 549–562. ACM, 2011.

[118] Louis-Noël Pouchet. PolyBench 3.1. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`, 2011.

[119] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP '03*. ACM, 2003.

[120] Benoit Pradelle, Philippe Clauss, and Vincent Loechner. Adaptive runtime selection of parallel schedules in the polytope model. In *19th High Performance Computing Symposium - HPC 2011*, Boston, USA, April 2011. ACM/SIGSIM.

[121] Benoît Pradelle, Alain Ketterlin, and Philippe Clauss. Polyhedral parallelization of binary code. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):39, 2012.

[122] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[123] Easwaran Raman, Ram Rangan, David I August, et al. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code Generation and Optimization*, pages 175–184. ACM, 2008.

[124] Lawrence Rauchwerger and David A Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 10(2):160–180, 1999.

[125] Rosetta Codes. `http://rosettacode.org/wiki/Rosetta_Code`.

[126] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in CUDA. `http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf`, 2010.

[127] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[128] Shane Ryoo, Christopher I Rodrigues, Sam S Stone, Sara S Baghsorkhi, Sain-Zee Ueng, John A Stratton, and Wen-mei W Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204. ACM, 2008.

[129] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: collaborative speculative loop execution on GPU and CPU. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 64–73. ACM, 2012.

[130] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., NY, USA, 1986.

[131] Eric Schweitz, Richard Lethin, Allen Leung, and Benoit Meister. R-stream: A parametric high level compiler. *HPEC*, 2006.

[132] K. Shirahata, H. Sato, and S. Matsuoka. Hybrid map task scheduling for GPU-based heterogeneous clusters. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 733–740, Nov 2010.

[133] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 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*, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[134] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *J. Funct. Program.*, 13(3), May 2003.

[135] Aravind Sukumaran-Rajam, Luis Esteban Campostrini, Juan Manuel Martinez Caamano, and Philippe Clauss. Speculative runtime parallelization of loop nests: Towards greater scope and efficiency. *HIPS + LSPP*, May 2015.

[136] Weibin Sun and Robert Ricci. Augmenting operating systems with the GPU. *arXiv preprint arXiv:1305.3345*, 2013.

[137] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*.

[138] Sanket Tavarageri, L Pouchet, J Ramanujam, Atanas Rountev, and P Sadayappan. Dynamic selection of tile sizes. In *High Performance Computing (HiPC), 2011 18th International Conference on High Performance Computing*, pages 1–10. IEEE, 2011.

[139] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[140] Chen Tian, Min Feng, and Rajiv Gupta. Speculative parallelization using state separation and multiple value prediction. In *Int. Symp. on Memory Management*, ISMM '10. ACM, 2010.

[141] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 115–124, New York, NY, USA, 2010. ACM.

[142] University of Sydney. MAGMA. `http://magma.maths.usyd.edu.au/`, 1993.

[143] Sundaresan Venkatasubramanian, Richard W Vuduc, et al. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. ACM.

[144] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[145] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[146] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, January 2012.

[147] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

[148] Vasily Volkov. Better performance at lower occupancy. *GPU Technology Conference 2010 (GTC 2010)*.

[149] Vasily Volkov and James Demmel. LU, QR and cholesky factorizations using vector capabilities of GPUs. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May*, pages 2008–49, 2008.

[150] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[151] Sandra Wienke, Dieter an Mey, and MatthiasS. Müller. Accelerators for technical computing: Is it worth the pain? a TCO perspective, 2013.

[152] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[153] Niklaus Wirth. A plea for lean software. *Computer*, 28(2):64–68, February 1995.

Jean-François DOLLINGER

A framework for efficient execution on GPU and CPU+GPU systems

Résumé

Les verrous technologiques rencontrés par les fabricants de semi-conducteurs au début des années deux-mille ont abrogé la flambée des performances des unités de calculs séquentielles. La tendance actuelle est à la multiplication du nombre de cœurs de processeur par socket et à l'utilisation progressive des cartes GPU pour des calculs hautement parallèles. La complexité des architectures récentes rend difficile l'estimation statique des performances d'un programme. Nous décrivons une méthode fiable et précise de prédiction du temps d'exécution de nids de boucles parallèles sur GPU basée sur trois étapes : la génération de code, le profilage offline et la prédiction online. En outre, nous présentons deux techniques pour exploiter l'ensemble des ressources disponibles d'un système pour la performance. La première consiste en l'utilisation conjointe des CPUs et GPUs pour l'exécution d'un code. Afin de préserver les performances il est nécessaire de considérer la répartition de charge, notamment en prédisant les temps d'exécution. Le runtime utilise les résultats du profilage et un ordonnanceur calcule des temps d'exécution et ajuste la charge distribuée aux processeurs. La seconde technique présentée met le CPU et le GPU en compétition : des instances du code cible sont exécutées simultanément sur CPU et GPU. Le vainqueur de la compétition notifie sa complétion à l'autre instance, impliquant son arrêt.

Summary

Technological limitations faced by the semi-conductor manufacturers in the early 2000's restricted the increase in performance of the sequential computation units. Nowadays, the trend is to increase the number of processor cores per socket and to progressively use the GPU cards for highly parallel computations. Complexity of the recent architectures makes it difficult to statically predict the performance of a program. We describe a reliable and accurate parallel loop nests execution time prediction method on GPUs based on three stages: static code generation, offline profiling, and online prediction. In addition, we present two techniques to fully exploit the computing resources at disposal on a system. The first technique consists in jointly using CPU and GPU for executing a code. In order to achieve higher performance, it is mandatory to consider load balance, in particular by predicting execution time. The runtime uses the profiling results and the scheduler computes the execution times and adjusts the load distributed to the processors. The second technique, puts CPU and GPU in a competition: instances of the considered code are simultaneously executed on CPU and GPU. The winner of the competition notifies its completion to the other instance, implying the termination of the latter.