Methods and tools for rapid and efficient parallel implementation of computer vision algorithms on embedded multiprocessors

Vítor Schwambach

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel du 7 août 2006

Présentée par

Vítor Schwambach

Thèse dirigée par **Stéphane Mancini**
et codirigée par **Sébastien Cleyet-Merle et Alain Issard**

préparée au sein du
Laboratoire TIMA et STMicroelectronics
et de l'École Doctorale
Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII)

Methods and Tools for Rapid and Efficient Parallel Implementation of Computer Vision Algorithms on Embedded Multiprocessors

Thèse soutenue publiquement le **30/03/2016**
devant le jury composé de:

Dominique Borrione
Université de Grenoble, France, Président
François Berry
Université Blaise Pascal, Clermont-Ferrand, France, Rapporteur
Steven Derrien
Université de Rennes 1, France, Rapporteur
Dietmar Fey
Friedrich-Alexander Universität Erlangen-Nürnberg, Germany, Examinateur
Stéphane Mancini
Université de Grenoble, France, Directeur de thèse
Sébastien Cleyet-Merle
STMicroelectronics, France, Invité

Abstract

Embedded computer vision applications demand high system computational power and constitute one of the key drivers for application-specific multi- and many-core systems. A number of early system design choices can impact the system's parallel performance – among which the parallel granularity, the number of processors and the balance between computation and communication. Their impact in the final system performance is difficult to assess in early design stages and there is a lack for tools that support designers in this task. The contributions of this thesis consist in two methods and associated tools that facilitate the selection of embedded multiprocessor's architectural parameters and computer vision application parallelization strategies. The first consists of a *Design Space Exploration* (DSE) methodology that relies on Parana, a fast and accurate parallel performance estimation tool. Parana enables the evaluation of what-if parallelization scenarios and can determine their maximum achievable performance limits. The second contribution consists of a method for optimal $2D$ image tile sizing using constraint programming within the Tilana tool. The proposed method integrates non-linear DMA data transfer times and parallel scheduling overheads for increased accuracy.

Résumé

Les applications de vision par ordinateur embarquées demandent une forte capacité de calcul et poussent le développement des systèmes multi- et many-cores spécifiques à l'application. Les choix au départ de la conception du système peuvent impacter sa performance parallèle finale – parmi lesquelles la granularité de la parallélisation, le nombre de processeurs et l'équilibre entre calculs et l'acheminement des données. L'impact de ces choix est difficile à estimer dans les phases initiales de conception et il y a peu d'outils et méthodes pour aider les concepteurs dans cette tâche. Les contributions de cette thèse consistent en deux méthodes et les outils associés qui visent à faciliter la sélection des paramètres architecturaux d'un multiprocesseur embarqué et les stratégies de parallélisation des applications de vision embarquée. La première est une méthode d'exploration de l'espace de conception qui repose sur Parana, un outil fournissant une estimation rapide et précise de la performance parallèle. Parana permet l'évaluation de différents scénarios de parallélisation et peut déterminer la limite maximale de performance atteignable. La seconde contribution est une méthode pour l'optimisation du dimensionnement des tuiles d'images $2D$ utilisant la programmation par contraintes dans l'outil Tilana. La méthode proposée intègre pour plus de précision des facteurs non-linéaires comme les temps des transferts DMA et les surcoûts de l'ordonnancement parallèle.

Résumé Étendu

Abstract

Le développement d'une plateforme multiprocesseur pour une application spécifique, nécessite de répondre à deux questions essentielles : *(i)* comment tailler la plateforme multiprocesseur pour atteindre les pré-requis de l'application avec la surface minimum et la plus faible consommation ; et *(ii)* comment paralléliser l'application cible de façon à maximiser l'utilisation de la plateforme. Dans ce travail, nous présentons une méthodologie originale d'estimation de la performance parallèle et d'optimisation des transferts de données, à partir des traces d'une exécution de l'application séquentielle et d'un modèle d'optimisation par contraintes des transferts de données. L'estimation à la fois de la performance parallèle et du temps des transferts de données doit être rapide car elle est au coeur du processus d'exploration de l'espace de conception de la plateforme multiprocesseur. Cette méthodologie est implémentée dans les outils Parana, un simulateur abstrait très rapide, basé sur des traces, et Tilana, un outil pour optimisation des transferts de données selon une modélisation analytique dans le cadre d'une optimisation par contraintes. Les outils ciblent des applications OpenMP sur la plateforme STxP70 Application-Specific Multiprocessor (ASMP) de STMicroelectronics. Les résultats pour un benchmark du NAS Parallel Benchmark et une application de vision embarquée démontrent une marge d'erreur de l'estimation de performance inférieur à 10% en comparaison aux simulateur ISS *cycle-approximate* de référence et à un prototype FGPA, avec un effort de modélisation réduit. Tilana permet de trouver des paramètres de transferts de données qui optimisent le temps de transferts de tuiles d'images pour une application donnée.

Contents

R.1 Introduction

Grâce à leur flexibilité et leur puissance de calcul potentielle, les architectures multicoeurs massivement parallèles devraient être les supports de choix pour de nombreuses applications embarquées. Cependant, concevoir de tels systèmes efficaces nécessite de lever un certain nombres d'obstacles [29]. La difficulté de leur programmation efficace, qui maximise la puissance de calcul effective pour bien utiliser les ressources matérielles, réside non seulement dans la gestion des communications et synchronisations [83], mais aussi dans la prise en compte des ressources matérielles de mémorisation et de calcul de chaque noeud [183]. Dans le domaine de la vision embarquée sur plateforme multiprocesseur, les grandes quantités de données en jeu, la diversité des traitements et les dépendances des calculs aux données poussent ces systèmes à leurs limites.

Dans un système hétérogène multicoeurs, il est envisagé d'intégrer des clusters multicoeurs adaptés aux besoins spécifiques des utilisateurs. Chaqu'un de ces accélérateur multicoeurs est taillé sur mesure pour la gamme d'applications ciblées. Définir les paramètres d'architecture tels que le nombre de coeurs, la topologie des communications et les ressources de mémorisation, nécessite de prédire les performances de l'accélérateur multicoeur au plus tôt lors du processus de conception, et ceci pour les applications typiques envisagées.

La difficulté à estimer les performances d'une architecture multicoeurs exécutant un programme complexe, dont le déroulement dépend des données, provient de phénomènes d'attentes difficiles à prédire car ils sont d'une part liés aux communications et synchronisations des différentes tâches et d'autre part sont aussi dus au comportement de la plateforme matérielle. Ces surcoûts doivent être pris en compte lors de l'estimation de performance de l'application car, sinon, ils peuvent conduire à une sous-évaluation des performances réelles d'un circuit. Par exemple, un simulateur qui ne permet pas de mesurer assez finement certains phénomènes temporels peut conduire à de graves erreurs d'appréciation des performances [183]. Un outil d'estimation de performance doit aussi être suffisamment précis afin de préserver l'ordre des solutions de l'espace de conception. C'est-à-dire que si deux paramétrages de l'architecture et de la parallélisation conduisent à deux estimations de performance, alors l'ordre des performances réelles sur le système doit être le même que celui des estimations.

Pour les applications à parallélisme de données qui nécessitent de transférer d'importantes quantités de données, le logiciel doit être conçu afin de compenser les surcoûts en choisissant des stratégies de parallélisation adaptées. Parmi les options offertes, les politiques d'ordonnancement, les mécanismes de transfert de données ou encore la granularité du parallélisme, liée à la granularité des communications, ont un impact qu'il est nécessaire d'évaluer au plus tôt du processus de conception. Il est nécessaire de pouvoir les évaluer rapidement, avant même que le circuit ne soit disponible.

Les techniques d'estimation de performance disponible offrent différents compromis entre la précision et la vitesse. A l'opposé de la technique la plus lente, la simulation précise au niveau transfert de registre (RTL), les simulations basées sur des *instruction-set simulators* (ISS) ou bien sur de la *dynamic binary translation*(DBT)(§R.7.2), sont les plus rapides mais en contrepartie d'une perte de précision qui peut être dommageable. En tant que solution intermédiaire, le prototypage permet un compromis vitesse/qualité intéressant mais n'est pas toujours possible et nécessite malgré tout une conception RTL poussée. Toutes ces stratégies nécessitent une plateforme de simulation du parallélisme ainsi qu'une version parallèle de l'application qui

l'exploite. Selon le modèle de calcul, il peut aussi être nécessaire de disposer d'un système d'exploitation et d'un environnement d'exécution, qui doivent être simulés aussi. Dans le domaine des serveurs de calcul, il existe également des outils d'estimation de performance qui permettent au concepteur d'obtenir des mesures impossibles à obtenir sur le processeur lui-même. Ces outils, comme Kismet [105], Parallel Prophet [119], et Intel Advisor XE [97], ne sont pas adaptés à l'exploration de l'espace de conception d'accélérateurs multicoeurs.

Dans cette thèse, je propose une méthodologie d'exploration précoce de l'espace de conception basée sur des outils rapides d'estimation de performance à partir d'un code séquentiel (§ R.2). L'objectif est de permettre au concepteur :

- D'estimer le potentiel d'accélération de l'application
- De comparer différentes stratégies de parallélisation
- D'identifier l'origine des surcoûts et des points bloquants
- De paramétrer la plateforme multiprocesseur

La méthodologie est dite « précoce » dans le sens où elle concerne l'estimation du potentiel de parallélisation et non pas la parallélisation proprement dite. Aussi, le premier point est déterminant car il permet au concepteur de rapidement savoir s'il est intéressant de se consacrer à la conception d'une application parallèle plus poussée ou s'il y a des point bloquants à résoudre avant.

Cette méthodologie, implantée dans les outils **Parana** et **Tilana**, se décompose en trois grandes phases :

- Caractériser numériquement une plateforme réelle (ou simulée) et son modèle de programmation en mesurant les surcoûts et différentes caractéristiques ;
- Acquérir des mesures sur une application séquentielle, à l'aide des trace issues d'un simulateur (ISS ou *cycle-true*) ;
- Estimer les performances de l'application par simulation abstraite, à partir des caractéristiques et mesures précédentes ;
- Optimiser les transferts de données selon un modèle d'optimisation par contraintes.

L'étape de simulation abstraite est très rapide car le code de l'application n'est pas exécuté à proprement parler, comme détaille la section R.3. La simulation abstraite est en cela assez proche d'un calcul mathématique des performances, mais avec une vitesse de simulation supérieure celle d'un simulateur de processeur qui doit interpréter ou exécuter chaque instruction de l'application. À la seconde étape, la mesure de performance du code séquentielle à partir d'un simulateur au cycle près (*cycle-true*) permet d'obtenir des mesures de temps bien plus précises que celles d'un simulateur DBT, et ce quel que soit l'implémentation de l'*instruction-set architecture*(ISA) – pipeline, *instruction-level parallelism*(ILP) – ou même en présence d'instructions spécifiques.

Du point de vue du concepteur qui utilise ces outils, l'exploration d'architecture consiste donc soit à caractériser la plateforme en intégrant des paramètres d'architecture au modèle numérique (par exemple le nombre de processeurs), soit à faire une caractérisation pour chaque plateforme étudiée, puis à passer à l'estimation de performance par simulation abstraite, ce qui est plus rapide que de simuler l'application sur chaque plateforme. Cette méthodologie devrait permettre d'obtenir la précision d'un simulateur au cycle-près avec une vitesse d'un ordre de grandeur supérieure aux simulateurs d'instruction.

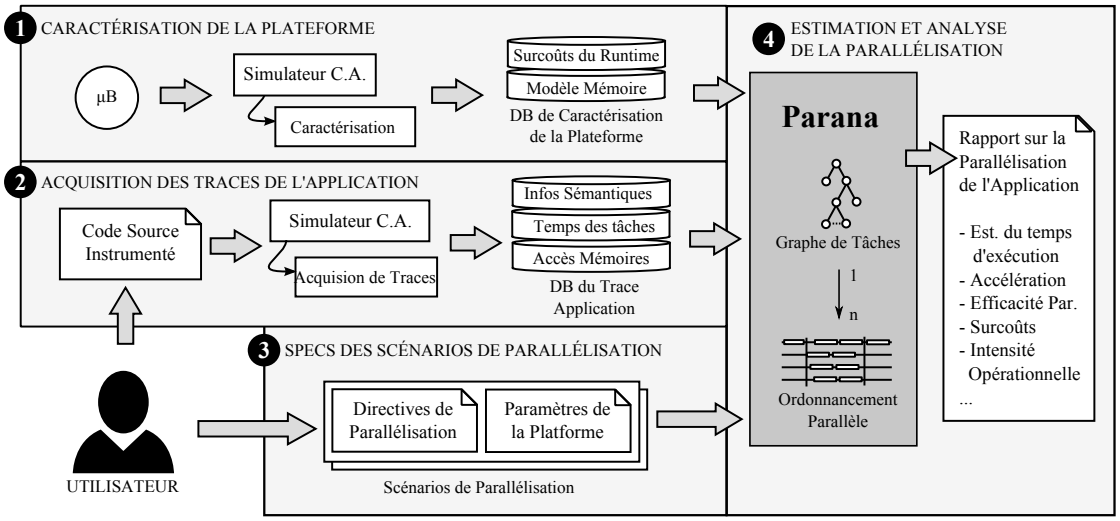


Figure R.1: *Aperçu des quatre pas du flot proposé pour l'estimation de la performance parallèle d'une application et l'exploration de l'espace de conception avec Parana.*

Cette méthodologie et les outils Parana et Tilana ont été validés sur la plateforme Application Specific MultiProcessor (ASMP) de STMicroelectronics. La plateforme est composée de grappes de processeurs STxP70 et le modèle de programmation choisi est OpenMP. La méthodologie et l'outil pourraient être généralisés pour d'autres plateformes. L'outil Parana a été validé en comparant les performances estimées aussi bien aux mesures issues d'un simulateur qu'à un prototype FPGA de la plateforme. Comme détaillé en section R.6, les résultats montrent que Parana donne une très grande précision de l'estimation, avec une erreur de 10% par rapport à la plateforme réelle, et ceci avec une vitesse de simulation supérieure à un simulateur de type ISS d'un ordre de grandeur.

R.2 Flot de conception proposé

La méthode de simulation rapide proposée repose sur un simulateur abstrait calculant les performances de l'application sur la plateforme parallèle à partir de traces de l'application séquentielle et de mesures de caractérisation des surcoûts induits par la plateforme et son environnement de parallélisation. Ainsi, il devient très rapide d'explorer les performances de configurations logicielles et matérielles à partir de traces et caractéristiques préparées par pré-traitement. Dans ces travaux, la cible consiste à une plateforme multicoeur intégrée STxP70 ASMP (§R.5.1) gérée par l'environnement d'exécution OpenMP. La figure R.1 décrit les quatre étapes de cette méthode, détaillées par la suite :

Etape 1 : Caractérisation de la plateforme. La première étape consiste à évaluer numériquement les différentes caractéristiques de la plateforme multicoeur et de son environnement qui ont le plus d'impact sur les surcoût de performance. La base de données des caractéristiques contient des informations statistiques mesurées sur des applications de référence instrumentées. Les informations concernent aussi bien les différents temps de calcul en surplus induits par l'environnement OpenMP et les différentes directives afférentes que des caractéristiques spécifiques à la plateforme matérielle telles que les débits et latences d'accès aux différents niveaux de mémoire et de cache. Les mesures sont réalisées à partir d'une version étendue du micro-benchmark EPCC OpenMP [43]. Les informations temporelles peuvent être extraites à

Listing R.1: *Exemple d'un code source instrumenté pour parallélisation avec l'outil Parana (c.f. directives dans Listing R.2).*

```
1  void vAdd(int *r, int *a, int *b, int n) {
2    int i;
3    //PA_START_TASK(<var>, <label>)
4    PA_START_TASK(_parallel0_, "parallel0");
5    for(i=0; i < n; i++) {
6      PA_START_TASK(_for0_, "for0");
7      r[i] = a[i] + b[i];
8      PA_END_TASK(_for0_);
9    }
10   //PA_END_TASK(<var>)
11   PA_END_TASK(_parallel0_);
12 }
```

Listing R.2: *Exemple de fichier de propriétés avec des directives de parallélisation pour la fonction vAdd (c.f. code source dans Listing R.1).*

```
1 vAdd = #define NUM_THREADS 8
2 vAdd = #define CHUNK_SIZE 1
3 vAdd.parallel0 = #pragma omp parallel num_threads(NUM_THREADS)
4 vAdd.for0 = #pragma omp for schedule(static, CHUNK_SIZE)
```

partir d'un simulateur (de préférence au cycle-près) ou même d'un émulateur matériel. Les mesures sont traitées par notre outil de caractérisation qui en déduit une base de données spécifique à chaque version de l'architecture ciblée. Cette base de données servira pour générer rapidement des estimations de performance.

Etape 2 : Caractérisation de l'application. À partir du profilage d'un code séquentiel, cette seconde étape extrait les informations spécifiques à l'application. L'objectif est de déduire de l'exécution séquentielle un graphe de tâches à paralléliser par la suite. L'extraction du parallélisme est faite de façon soit implicite soit explicitée par l'utilisateur. Dans la version implicite, l'hypothèse est que, dans l'environnement OpenMP, les tâches sont liées aux appels de fonctions dans les boucles et à la présence de commande OpenMP de parallélisation (pragmas). L'utilisateur peut également indiquer des zones parallèles explicites en insérant des commandes dans le code source à l'aide de macros spécifiques. Le profilage mesure également des statistiques des accès mémoires dans chacune des tâches identifiées afin d'estimer les temps d'accès et collisions aux mémoires partagées. L'utilisateur peut également ajouter divers informations sémantiques pour affiner son analyse. Listing R.1 montre un exemple d'instrumentation d'une fonction de somme vectorielle *vAdd*.

Etape 3 : Spécification des scénarios de parallélisation. L'utilisateur peut définir un ensemble de scénarios de parallélisation qui correspondent chacun à une configuration de l'espace de conception. Un scénario de parallélisation comporte un groupe de paramètres spécifiques à la plateforme matérielle (nombre de processeurs, mémoires, etc.) et un second spécifique à l'application. Ce dernier groupe est en fait le jeu de paramètres qui décrit la stratégie de parallélisation choisie (ordonnancement statique/dynamique, nombre de tâches, etc.) et correspond aux directives OpenMP associées à chaque zone de parallélisme identifiée à l'étape 2. Listing R.2 montre le fichier de propriétés pour un possible scénario de parallélisation de la fonction *vAdd* dans Listing R.1.

Etape 4 : Estimation de performance de la parallélisation. Cette dernière étape consiste à estimer les temps de calcul de l'application parallélisée selon les directives indiquées, et ceci à partir de la base de données de caractéristiques et des mesures obtenues par le profilage. Cette

étape produit une analyse détaillée de l'origine des surcoûts induits par l'environnement de parallélisation, l'équilibrage de charge, les zones séquentielles, les accès mémoires et autres phénomènes difficiles à estimer analytiquement. Cette étape est extrêmement rapide car le code de l'application n'est pas exécuté et la plateforme matérielle n'est pas simulée dans le détail, ceci ayant été fait une fois pour toutes à l'étape 1.

La boucle de conception peut se focaliser sur les étapes 3 et 4, et, en quelques secondes, l'utilisateur peut analyser des dizaines de configuration et éliminer les plus aberrantes. Les performances des configurations retenues peuvent alors être validées par une simulation classique. L'avantage de la méthode est qu'elle permet d'éviter d'implanter chaque configuration matérielle et chaque version parallèle du logiciel en partant d'un code séquentiel et de passer au crible rapidement de nombreuses stratégies de parallélisation.

R.2.1 Caractérisation de la plateforme

R.2.2 Instrumentation de l'application

Notre infra-structure d'acquisition de traces d'instruction est capable de générer une trace de tâches automatiquement aux appels de fonctions. Accessoirement, nous fournissons une librairie d'instrumentation qui permet à l'utilisateur de agrémenter ces traces avec des tâches labellisées. Ces dernières sont définies de façon explicite par moyen d'un ensemble de macros d'instrumentation qui enserrent la section de code appartenant à chaque tâche. Les tâches explicites créent un niveau hiérarchique supplémentaire dans l'arbre de tâches, ce qui permet de regrouper toutes les sous-tâches à l'intérieur de sa section de code. Listing R.1 montre un exemple d'une fonction de somme vectorielle *vAdd* instrumenté avec deux tâches labellisées : *parallel0* et *for0*.

R.2.3 Spécification des scénarios de parallélisation

Lors de la définition d'un scénario de parallélisation, des directives OpenMP peuvent être attachées aux tâches tracés – qu'elles relèvent des appels de fonctions ou de l'instrumentation explicite –, leur ajoutant une sémantique de parallélisme, ordonnancement et/ou synchronisation. Nous utilisons un fichier de *propriétés* pour spécifier les directives de parallélisation et les relier aux tâches. Un fichier de *propriétés* consiste à une série de combinaisons clés-valeurs, où les clés sont les noms des tâches (noms des fonctions ou labels utilisateurs), et les valeurs sont les directives OpenMP. Un deuxième fichier de *propriétés* définit les valeurs des paramètres de la plateforme multiprocesseur.

Listing R.1 montre les directives d'une possible parallélisation pour la fonction *vAdd* (c.f. § R.2.2) qui ouvre une région parallèle avec 8 threads contenant une boucle for avec ordonnancement statique et un bloc de taille 1. Des multiples fichiers de *propriétés* peuvent être crées pour décrire les différents scénarios de parallélisation à explorer.

R.3 Parana

Parana utilise les données de la caractérisation de la plateforme multiprocesseur, les traces de l'application et le fichier de spécification des scénarios de parallélisation, pour estimer et analyser la performance parallèle de l'application. Initialement, il construit un graphe de tâches à partir des traces de l'application. Puis, pour chaque scénario de parallélisation, un nouveau graphe est obtenu en attachant aux tâches les directives de parallélisation. Finalement, un

scénario de parallélisation produit un calcul de l'ordonnancement parallèle correspondant. Cet ordonnancement prend en compte les surcoûts liés à l'environnement OpenMP et au modèle mémoire de la plateforme. Finalement, un rapport de parallélisation détaillé est généré.

R.3.1 Construction du graphe de tâches

La représentation interne des traces de tâche consiste en un graphe de tâches hiérarchique – un graphe orienté acyclique (DAG) où les sommets sont des tâches et les arcs représentent des relations père-fils. Du fait du profiling à partir du code séquentiel, il y a un recouvrement temporel entre les tâches pères et les tâches fils, ce qui nécessite un pré-traitement sur le graphe pour procéder à l'ordonnancement car nous n'avons besoin que des tâches feuilles. Cette étape d'insertion de tâches feuilles aux intervalles où il n'y a pas de recouvrement père-fils et de suppression des parties recouvrantes est dénommée *segmentation* et produit un graphe de tâche *hiérarchique*.

R.3.2 Ordonnancement des tâches

Un ordonnancement est à son tour représenté comme un DAG où les sommets sont des tâches dites *ordonnançables* et les arcs représentent des relations de précédence entre de telles tâches. Les tâches *ordonnançables* sont annotées d'informations supplémentaires liées à l'ordonnancement, telles que la date initiale dans l'ordonnancement et l'identifiant du processeur cible.

Initialement, Parana construit un ordonnancement sur un seul coeur qui sera la référence pour le calcul de l'accélération et de l'efficacité parallèle. La génération de l'ordonnancement se fait en parcourant l'arbre de tâches *hiérarchique* en profondeur et, à chaque tâche feuille trouvée, une tâche *ordonnançable* est ajoutée dans le DAG de l'ordonnancement.

Ensuite, l'ordonnancement parallèle est effectué pour chaque scénario de parallélisation défini par l'utilisateur. Pour réaliser l'ordonnancement parallèle, le DAG *hiérarchique* est parcouru et lorsqu'une tâche possédant une directive OpenMP est trouvée, l'ordonnanceur émule l'ordonnancement qui serait effectué par l'environnement d'exécution OpenMP. C'est à ce moment que le simulateur introduit des tâches supplémentaires pour modéliser les surcoûts liés au lancement, à la synchronisation et au rassemblement des sous-tâches, entre autres, dont les temps ont été estimés selon la caractérisation de la plateforme.

R.3.3 Analyse de la parallélisation

Un rapport détaillé est généré à la suite de chaque ordonnancement, incluant notamment : le temps total d'exécution estimé de l'application ; l'accélération vis-à-vis de la référence séquentielle ; l'efficacité de la parallélisation. Ce rapport note également les statistiques pour les portions séquentielles et parallèles de l'ordonnancement. Toutes les mesures sont décomposées en temps de calcul, temps d'inactivité et surcoûts. Ces informations sont utilisées pour créer des *cycle-stacks*, un type de graphe qui permet de visualiser la décomposition des facteurs qui contribuent au temps d'exécution dans un ordonnancement donné.

En outre, Parana est capable de générer des rapports pour chaque région parallèle de l'application. Dans ce cas, il fournit aussi la localisation de la région parallèle dans l'arbre d'appels et, pour chaque tâche, le nombre de cycles moyen par exécution. Cela permet au concepteur de retrouver sur un seul rapport les statistiques nécessaires pour analyser les gains et les goulot d'étranglement potentiels.

R.4 Tilana

Les applications de traitement d'image embarquées utilisent souvent des buffers d'entrée et sortie placés dans une mémoire externe. Ces buffers de données sont utilisés soit comme un moyen d'échanger des données entre les différents sous-systèmes, ou simplement car la quantité de données est trop importante pour tenir dans la mémoire locale réduite d'un système embarqué. Etant donné que le temps d'accès à la mémoire externe est beaucoup supérieur à celui de la mémoire interne, les données doivent d'abord être transférées à la mémoire locale avant d'être traitées par les unités de calcul. Sur les systèmes avec mémoire gérée explicitement, sans cache de données, il est la responsabilité du programmeur de gérer les transferts de données entre les espaces mémoires interne et externe. Une solution courante consiste à utiliser le tuilage, qui consiste à subdiviser les données en plus petites zones, les tuiles, qui sont assez petites pour rentrer dans la mémoire interne et qui peuvent être traitées de façon indépendante.

Pour mieux utiliser les ressources disponibles, le tuilage est implémenté comme un pipeline logiciel de trois étapes qui consistent à : *(i)* lire les données d'entrée depuis la mémoire externe et les écrire dans la mémoire interne ; *(ii)* processer les données localement ; et *(iii)* écrire les données de sortie sur la mémoire externe. La Figure 6.1 illustre ce pipeline de calcul de trois étapes. Les avantages du tuilage incluent la réduction de l'empreinte de données dans la mémoire locale et la réduction de la latence de l'application grâce au chevauchement entre transferts de données et calcul.

La question est comment sélectionner les dimensions optimales de tuilage ? La taille et la forme des tuiles ont d'importante conséquences sur le temps et la quantité de données transferts. Ces divers facteurs et leur impact sont difficiles à estimer tôt dans le flot de conception, empêchant les programmeurs de trouver une solution optimale.

Pour relâcher la contrainte sur les programmeurs, certains compilateurs intègrent le support au tuilage automatique, via des techniques récentes de compilation polyédrique. Des exemples de tels compilateurs sont PLuTO [40] et PolyMage [153], les deux par Bondhugula et al., bien comme le compilateur commercial R-Stream [184, 14] par Reservoir Labs. Ces compilateurs performent des transformations affines de boucle qui visent à augmenter la localité des données et la bande passante mémoire sur les architectures avec cache de données. Darte et al. ont prouvé en [58] que les techniques de compilation polyédriques peuvent produire ordonnancements asymptotiquement optimales pour des cas avec dépendances uniformes. Leur optimalité n'est donc pas garantie pour ordonnancements à faible nombre d'itérations (ou tuiles). Même si un nombre élevé d'itérations est une contrainte plausible pour un système de haute performance, cela n'est pas le cas pour des systèmes embarqués de temps-réel. Par ailleurs, les techniques de compilation polyédriques ne peuvent pas intégrer des non-linéarités dans le model, qui comme démontré à la Section 4.7.5, sont nécessaires pour obtenir une bonne précision. D'autres méthodes se basent sur le tuilage pour obtenir un ordonnancement à gros grain sur les systèmes parallèles [111, 152]. Aucune de ces méthodes n'est capable de modéliser des transferts de données explicites avec des caractéristiques non-linéaires.

Dans un travail récent, Saïdi et al. [177] proposent un modèle analytique pour l'optimisation d'algorithmes de traitement de tableaux sur des systèmes avec de la mémoire gérée explicitement et l'expriment en termes d'un modèle pour optimisation par contraintes. Néanmoins, leur travaux repose sur un modèle de régression linéaire des temps de transfert de données du DMA qui, comme démontré en section 4.7.5, peuvent amener à des résultats trop imprécis sur certaines architectures. Par ailleurs, cette méthode assume que les dimensions de l'image sont des multiples entiers de la dimension des tuiles, une contrainte que je juge trop contraignante et que si surmontée pourrait amener à des solutions plus optimales.

Le chapitre 6 présente donc une méthode pour sélectionner les paramètres de tuilage qui minimisent le temps d'exécution d'un noyau d'application parallèle sur un multiprocesseur

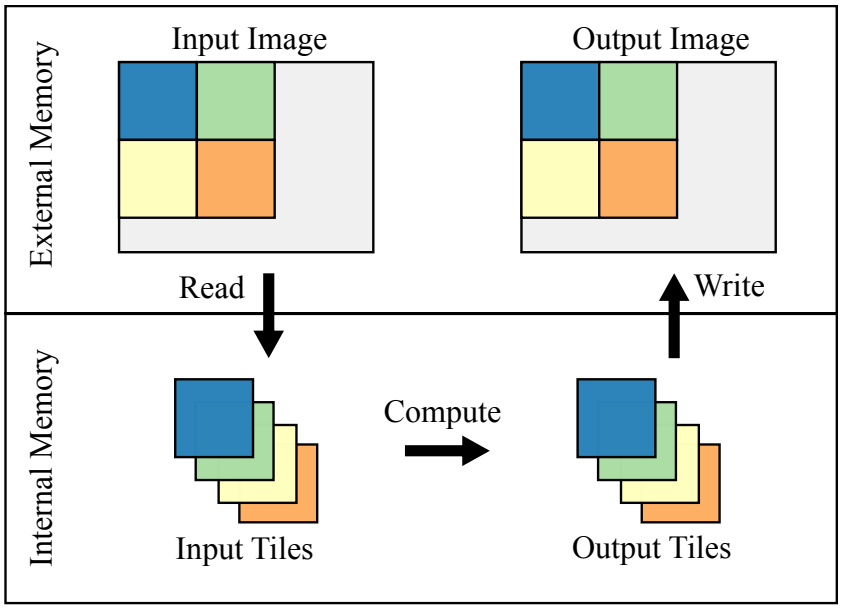


Figure R.2: *Aperçu du concept de pipeline de tuilage d'image proposé par Ievgen [91]. Cela consiste à décomposer l'image en une série de tuiles d'image qui sont : (i) lues depuis la mémoire externe, (ii) traitées localement, et les résultats (iii) réécrits sur la mémoire externe pour recomposer l'image de sortie.*

embarqué. Premièrement, un nouveau modèle analytique est présenté. Ce modèle est alors converti en un problème d'optimisation par contraintes implémenté dans l'outil Tilana utilisant le framework d'optimisation par contraintes Choco3 [200].

Les *contributions* principales du chapitre 6 sont :

1. proposer un nouveau modèle analytique pour estimer le temps d'exécution d'un noyau d'application parallèle qui implémente le tuilage, pour toutes les dimensions valides de tuiles ;
2. intégrer un modèle non-linéaire de la performance du DMA, décrit en section 4.7.5, pour un meilleure précision sur les estimations du temps de transferts du DMA ;
3. dériver et intégrer un modèle des surcoûts d'ordonnancement parallèle et ses dépendances sur une plateforme multiprocesseur ;
4. définir et implémenter un modèle d'optimisation par contraintes basé sur des contraintes non-linéaires qui permettent déterminer les dimensions optimales qui minimisent le temps d'exécution du noyau d'application parallèle.

R.5 Configuration des expériences

R.5.1 Architecture de la plateforme

Notre plateforme cible est le STxP70 Application-Specific Multiprocessor (ASMP) de STMicroelectronics. Cette plateforme est très similaire à un cluster STHORM [149]. Le STxP70 ASMP possède une architecture SMP configurable comptant jusqu'à 16 coeurs STxP70 – des CPUs RISC 32-bit dual-issue. La Figure 5.8 illustre son gabarit architectural. L'ASMP possède une mémoire L1 partagée, organisée en bancs, et accessible en un cycle via l'« interconnect », ce qui permet l'accès simultané à plusieurs bancs mémoire. En cas de conflit d'accès sur un banc mémoire, les processeurs sont bloqués et l'accès au banc en question est géré de façon équitable par un arbitre

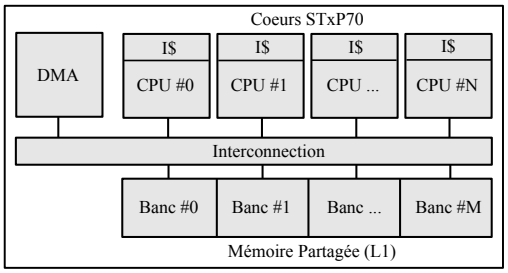


Figure R.3: *Gabarit de l'architecture du STxP70 ASMP.*

round-robin. En outre, chaque processeur compte 16KB de cache programme, mais pas de cache de données. Pour les expériences, une configuration avec 512KB de mémoire partagée, organisée en 32 bancs mémoire est utilisée.

R.5.2 Outils d'exécution

Simulateur Gepop. Gepop est un simulateur cycle-approximate pour la plateforme STxP70 ASMP et est notre simulateur de référence pour la caractérisation de la plateforme dans le flot proposé. Il est basée sur des simulateurs STxP70 monocoeurs ISS et intègre des modèles des autres composants de la plateforme, tels que le DMA, les mémoires, et l'interconnect. Sa précision est évalué à moins de 10% de marge d'erreur par rapport à un circuit physique.

Prototype FPGA. Un prototype du STxP70 ASMP sur le FPGA Xilinx VC707 est utilisé pour comparaison. En raison des contraintes d'occupation, le nombre de coeurs et la mémoire L1 sont limités à 8 coeurs et 512KB, respectivement. Le temps d'exécution sur FPGA est utilisé pour valider les temps d'exécution obtenus avec Gepop et Parana.

Parana. L'outil proposé, un simulateur basé sur des traces. Il estime la performance parallèle de l'application à partir des traces d'une exécution séquentielle de l'application et d'un ensemble de directives OpenMP pour de différents scénarios de parallélisation. Les traces Gepop sont utilisées à la fois pour la caractérisation de la plateforme et comme base pour les estimations de la performance parallèle.

R.5.3 Applications

Integer Sort (IS). J'ai sélectionné le benchmark IS parmi ceux de la suite NAS Parallel Benchmarks (NPB) [108], car une implémentation OpenMP en C est disponible. Deux adaptations furent nécessaires : *(i)* du fait des limitations des ressources mémoire, une classe de données plus petite que celle fournie avec le benchmark a été créée, et a été nommée Tiny (T) ; *(ii)* le runtime OpenMP pour ASMP ne supporte pas la directive *threadprivate*, utilisée dans la fonction qui génère les valeurs aléatoires à trier, et donc cette dernière a été remplacée par la fonction *rand_r*.

Détecteur de points caractéristiques FAST. J'ai porté l'algorithme de détection de points caractéristiques FAST 9-16 sur une cible STxP70 *Application-Specific Multiprocessor* (ASMP). Cet algorithme est couramment utilisé dans le domaine de la vision par ordinateur pour sélectionner les points caractéristiques dans des nombreuses applications de détection et de suivi. La référence est l'algorithme FAST fourni dans OpenCV version 2.4.6, lequel a été transcrit en C pour le STxP70 ASMP et parallélisé avec OpenMP. J'ai évalué deux scénarios de parallélisation distincts, le premier avec un ordonnancement de boucles statique et le deuxième avec un ordonnancement de boucles dynamique.

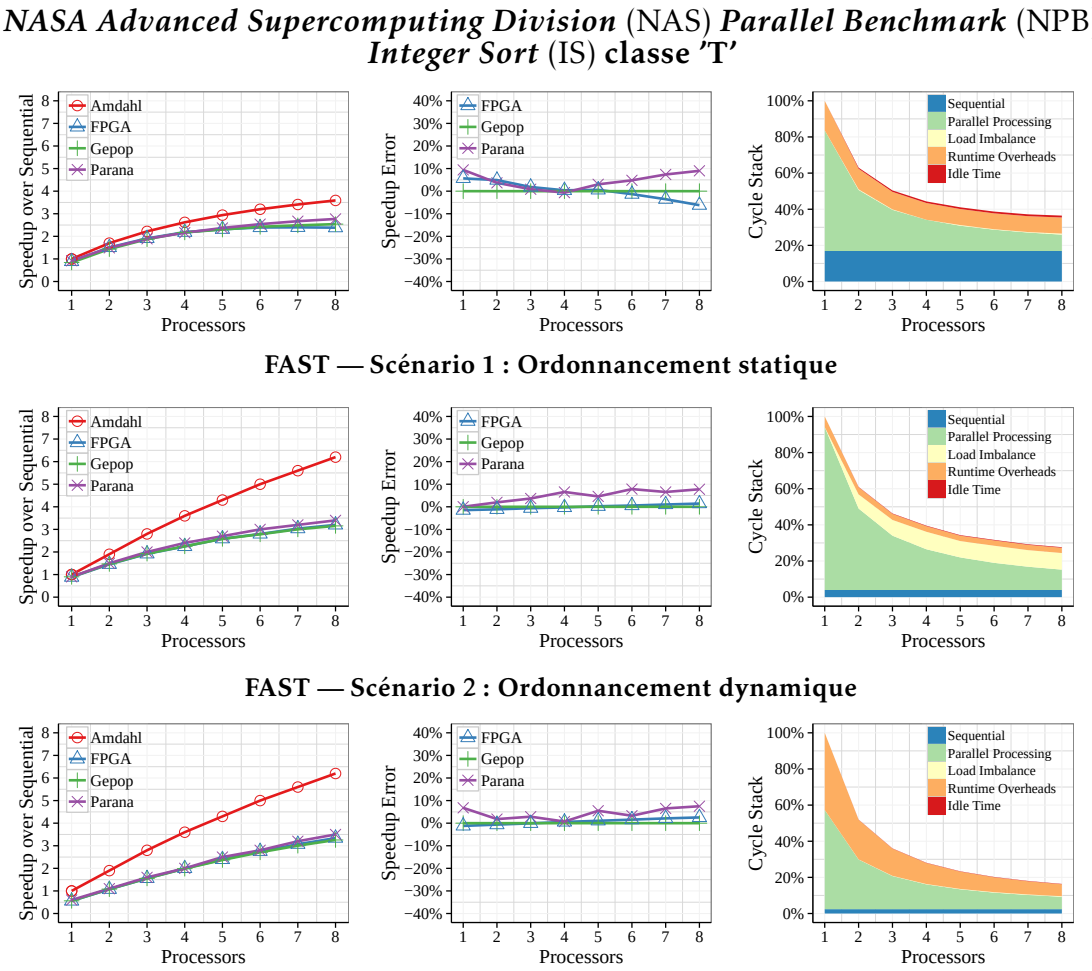


Figure R.4: *Comparaison du facteur d'accélération obtenu avec Parana, le simulateur cycle-approximate Gepop (référence pour l'acquisition de traces pour Parana) et un prototype FPGA, pour le benchmark NPB Integer Sort(IS) et le détecteur de points caractéristiques FAST. Gauche : facteurs d'accélération pour 1 à 8 coeurs. Centre : erreur relative au facteur d'accélération obtenu avec Gepop. Droite : Cycle stacks qui montrent la répartition par catégorie des cycles de l'application.*

R.6 Résultats

R.6.1 Estimations du facteur d'accélération

Les résultats sur les facteurs d'accélération pour les applications sélectionnées, ainsi que les erreurs relatives à la plateforme de simulation de référence (Gepop), sont présentés sur la Figure R.4. L'erreur moyenne de l'estimation du facteur d'accélération de l'outil Parana est de 3.8% avec une erreur maximum de 9.5% pour les scénarios testés.

Avec le noyau de référence « tri d'entier » NPB-IS l'erreur de l'estimation augmente avec le nombre de threads. C'est principalement dû à une opération de réduction qui dépend du nombre de threads. Cette opération de réduction n'est pas exécutée dans la version séquentielle de l'application et, par conséquent, n'est pas comptabilisée dans les traces exploitées par l'outil Parana.

Pour l'algorithme de détection de coins FAST, la représentation sous forme de graphique

Outil	Type	Entrées		Design + Config.		DSE			
		Application	Instr.	Temps	Effort	Temps d'exécution	Productivité	Erreur	Précision
FPGA	Prototype Matériel	Parallèle	Non	Jours	Important	60.0s	Basse	<5%	Haute
—	Sim. C-Accurate	Parallèle	Non	Heures	Important	—	Basse	<5%	Haute
Gepop	Sim. C-Approximate	Parallèle	Non	Heures	Moyenne	24.3s	Moyenne	<10%	Moyenne
Kismet	Chemin Critique	Séquentiel	Non	Seconds	Bas	—	Basse	>10%	Basse
Parallel Prophet	Modèle Analytique	Séquentiel	Oui	Minutes	Moyen	—	Basse	<10%	Moyenne
Parana	Modèle Analytique	Séquentiel	Oui	Minutes	Bas	1.6s	Haute	<10%	Moyenne

Table R.1: *Comparaison des méthodes de simulation et outils semblables pour l'estimation de la performance parallèle. Effort de mise en oeuvre : temps nécessaire pour préparer le code et la plateforme pour chaque point de l'espace de conception (faible : aucune modification au code original ; moyen : légères modifications du code ; important : modifications significatives du code source). Productivité : liée au temps pour configurer et évaluer de multiples points de conception (basse : long ; haute : rapide). Précision : précision relative à la performance du système physique (basse : >10% d'erreur ; moyenne : <10% d'erreur ; haute : <5% d'erreur).*

cumulé des cycles extrait par l'outil Parana met en évidence que l'équilibrage de charge est la principale cause de baisse d'efficacité pour un ordonnancement statique alors que pour un ordonnancement dynamique les pénalités ont plutôt pour origine les surcoûts de l'environnement d'exécution d'OpenMP. Cette analyse permet aussi de conclure que les irrégularités sur le facteur d'accélération avec un ordonnancement statique proviennent du déséquilibre de charge entre les différents threads, dont le nombre varie en fonction du nombre de coeurs. Enfin l'estimation de l'outil Parana est encore plus précise sur cette application avec une erreur moyenne de 3% dans les 2 scénarios (ordonnancement statique ou dynamique des tâches).

En conclusion, la précision de Parana est suffisante pour pourvoir observer les effets des changements de politique de parallélisation et préserve l'ordre des performances.

Les résultats pour les expériences sur l'outil Tilana sont présentés et discutés à la Section 6.6.

R.6.2 Temps d'exécution

Le tableau R.1 liste les moyennes de temps de simulation sur la machine hôte pour chaque configuration évaluée lors de nos expérimentations avec les plateformes Gepop, le prototype FPGA et l'outil Parana. Le temps de simulation avec Parana est en moyenne 15.9 fois plus rapide que le temps de simulation de la plateforme avec le simulateur Gepop, et jusqu'à 20 fois plus rapide dans certains cas. La plateforme FPGA est encore plus lente à cause du temps de chargement de l'application et des interactions avec le PC hôte pour les appels système. Dans ce cas la simulation avec Parana est 36.7 fois plus rapide que la simulation FPGA.

R.7 Travaux connexes

R.7.1 Modélisation de la performance parallèle

La loi d'Amdhal [25] a été l'un des premiers essais de modélisation de la performance des applications parallèles. De nombreux modèles dérivés ont été proposés, y compris certains modèles récents introduisant des multiprocesseurs hétérogènes [85]. D'autres outils, comme Kismet [105], utilisent une analyse hiérarchique du chemin critique pour estimer le facteur d'accélération potentiel d'une application séquentielle. Ces approches ne demandent que peu d'effort de développement et fournissent une estimation haute et souvent très optimiste.

L'outil Intel Advisor XE [97] estime les performances sur une architecture x86 qui repose sur une instrumentation utilisateur des régions parallèles et des boucles. Il manque à cette approche la modélisation des bandes passantes ainsi que les pénalités de cache. L'outil Parallel Prophet [119] repose aussi sur une instrumentation pour construire un graphe de tâches de l'application et estimer son facteur d'accélération avec différentes politiques d'ordonnancement. D'autres méthodes se concentrent sur le partitionnement et l'assignement de parties de l'application sur un système multiprocesseur [20], mais ne couvrent pas le parallélisme de données. Aucun des outils cités ne s'intéresse à la prédiction de performances pour du parallélisme de données sur des plateformes multiprocesseur spécifiques embarquées. De plus notre outil supporte la définition de différents scénarios de parallélisme avec des stratégies différentes et plusieurs configurations multiprocesseur, ce qui autorise une exploration rapide de l'espace de conception.

R.7.2 Techniques de Simulations Multiprocesseurs

Le tableau R.1 compare l'outil Parana avec d'autres outils de simulations, prototypes et outils de prédiction de performance. Les prototypes physiques et les simulateurs au cycle-près (cycle-accurate) sont très précis. Cependant les prototypes prennent du temps à mettre en oeuvre et ont des ressources limitées, alors que les simulations cycle-accurate sont lentes, ce qui limite l'exploration de l'espace de conception. La traduction binaire dynamique (DBT) ou les simulateurs de jeu d'instruction, tel que la plateforme de simulation Gepop, ont une précision de la mesure de cycle plus faible au profit d'une vitesse de simulation plus rapide. Des travaux récents sur des simulateurs au niveau code source (qui annotent le code source avec des informations temporelles et le simulent ensuite avec SystemC [188]) et des simulateurs basés sur l'échantillonnage [47], présentent une précision plus grande et une vitesse de simulation plus grande. Cependant, toutes ces approches nécessitent de travailler sur les versions parallèles de l'application pour chaque point de l'espace de conception. En tant que simulateur analytique, Parana associe une vitesse de simulation rapide, avec une bonne précision, à la possibilité de modéliser différents scénarios d'ordonnancement OpenMP à partir d'une application séquentielle, pour explorer l'espace de conception.

R.7.3 Principaux avantages

La méthodologie proposée fourni une estimation précise très tôt dans le flot de conception. Avant même d'avoir une version parallèle fonctionnelle de l'application. Ceci permet une exploration rapide de l'espace de conception à la fois pour les scénarios de parallélisation et pour la configuration du système multiprocesseur.

La méthode est extrapolable à d'autres familles de processeur et permet aussi d'évaluer l'impact de l'ajoût d'instructions spéciales. En effet, l'estimation de l'ajout d'une extension du jeu d'instruction du processeur ne nécessite que de produire une nouvelle trace d'exécution de l'application séquentielle recompilée avec l'utilisation de ces instructions.

Finalement, la production de documents d'analyse sous forme de « cycle-stacks » aide significativement l'architecte système à identifier les points bloquants et le potentiel de parallélisation de l'application.

R.7.4 Limitations connues

La limitation principale de notre méthode est qu'elle ne modélise pas correctement les applications qui contournent les constructions OpenMP standard pour implanter d'autres types de distributions parallèles (statiques ou dynamiques). Les contentions mémoire et les transferts

DMA ne sont pas encore modélisés et seront adressés dans un futur proche. Les applications de nos expérimentations n'utilisent pas le DMA et ne semblent pas souffrir de dégradations dues aux contentions mémoires. Les appels aux APIs du DMA peuvent aisément être interceptés avec notre méthodologie et ordonnancés sur une ressource dédiée dans le modèle d'ordonnancement de l'outil Parana pour modéliser le DMA. Les contentions mémoires peuvent être estimées en utilisant des histogrammes du nombre d'accès par banc mémoire pour en dériver un modèle statistique des surcoûts associés aux contentions.

R.8 Conclusion

Ce travail présente une méthodologie et l'outil Parana pour accélérer l'exploration conjointe de l'espace de conception des stratégies de parallélisation et de la configuration d'une plateforme multiprocesseur. L'estimation se fait à partir d'une trace d'exécution de l'application séquentielle et de directives de parallélisation. Les résultats de la méthodologie proposée ont été comparés aux mesures réalisées sur trois scénarios d'applications OpenMP, chacun ayant été mesuré sur une plateforme de simulation de jeu d'instruction (Gepop) et sur un prototype FPGA de la plateforme STxP70 ASMP. Il a été démontré que la précision de Parana dans l'estimation du facteur d'accélération parallèle est supérieure à 90% par rapport au simulateur de référence. Une telle précision montre ainsi son intérêt dans l'identification des facteurs limitant la performance parallèle via l'analyse détaillée des temps d'exécution. Cette précision est atteinte avec un temps de simulation 15 fois plus rapide que le simulateur de référence. Ainsi, cette simulation très rapide et précise peut être intégrée dans une boucle de conception pour parcourir l'espace de conception.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Abstract

This introductory chapter establishes the context and highlights the motivations of the work presented in this thesis. The chapter provides a brief introduction to the embedded computer vision domain. It discusses the main challenges for efficient parallel implementation of computer vision algorithms in embedded multiprocessor architectures. A brief overview of existing design aid tools is provided next. Finally, the problems this thesis addresses are presented, followed by a description of the contributions and the outline of the dissertation.

Contents

1.1 Embedded Computer Vision

The growing utilization of embedded cameras in portable devices led to the miniaturization and a significant reduction of production costs. These two factors have in turn opened the possibility to use these cameras in many other application domains. Furthermore, the computing power of embedded devices keeps increasing thanks to the shift towards more energy efficient multi- and many-core architectures and to continued advances in related technologies such as compilers and parallel programming models. As such, more sophisticated vision processing algorithms can be embedded in these systems. The combination of the sensing capability provided by the camera, along with the intelligence of the vision algorithms, led to what are known as smart camera systems. Smart cameras not only image the environment, but are able to extract information from the captured scenes and possibly make automated decisions based on this information.

Computer vision is being increasingly used in a number of different application domains. It is seen as a differentiation factor in a new wave of smart appliances and has the potential to increase traffic safety. Some of the application domains where embedded vision is gaining traction are:

Smart Objects and the Internet of Things (IoT). Vision-based gesture recognition has been incorporated to video game consoles since 2010 [221], with the launch of the first Microsoft Kinect, and to high-end television sets from Samsung in 2012 [110]. New smart home surveillance cameras, such as the Nest Cam [156] can detect motion in the video and automatically send remote alerts to the homeowner's smartphone. Netatmo's Welcome camera [157, 158] further integrates face recognition technology for monitoring presence, useful to know when children arrive from school or be warned of the presence of a stranger in the house.

Intelligent Video Surveillance Systems (IVSS) *and Smart Traffic Control*. As the number of connected surveillance cameras grows and cameras become *"smarter"*, new distributed architectures are emerging with local video processing in the camera heads [173]. Cameras in IVSS [205] can then detect events directly and notify the central in case of *"abnormal"* activities. Smart traffic control [33, 22] systems rely on embedded vision processing to provide information for adaptive traffic light control in *Intelligent Transportation Systems* (ITS), as well as to collect statistics about parking lot occupancy and traffic flow.

Advanced Driver Assistance Systems (ADAS). Several automotive constructors are embedding ADAS features in their vehicles for increased safety. Toyota introduced passenger facing cameras for driver attention monitoring in some of its models in 2006 [24]. Since 2010, Volvo S60 and V60 models integrate camera and radar based collision avoidance features, including pedestrian collision warning from Mobileye [144]. In October 2015, Tesla Motors has introduced steering autopilot via a software update to its Model S line [196].

Embedded vision technology is gaining traction and becoming the rule in all of the aforementioned applications. J. Bier president of the *Embedded Vision Alliance* (EVA) and of BDTI, a consulting firm on embedded signal processing, enumerates in [39] the reasons why embedded vision is proliferating:

1. It has the potential to create significant value in several application domains.
2. Consumers will increasingly expect it in new products.
3. It's now made possible by sufficiently powerful, low-cost and energy-efficient processors.

Computer vision algorithms can indeed be very computationally intensive. But not all vision processing is alike. First, the computational requirements often vary with the precision required by the target application and its complexity. Second, different stages of a vision processing pipeline present distinguishable characteristics. They can in fact be grouped into layers which

share common features. Classifications proposed by Dipert et al. in [63] and by Chitnis et al. in [48] identify the following basic levels:

1. Low-Level Processing or Pixel Processing.
2. Intermediate Level Processing or Object Processing.
3. High Level Processing or Object Recognition.
4. Control Logic or Decision Making and Application Control.

Vision algorithms in each of these levels present different computational and data transfer requirements. It is important thus to consider the characteristics of each level when designing an application-specific embedded vision system, as different levels might require different processor architecture trade-offs. As pointed by Chitnis et al. in [48], lower level vision functions typically present more available parallelism and are more suited to architectures with a high number of small processing cores, while higher level functions tend to show more control flow dependencies and perform better on fewer more powerful processors.

Vision algorithms are typically initially developed on PCs with general-purpose CPUs, as they are easier to program and have broader tool support. When targeting an embedded solution the algorithms then need to be ported to a suitable embedded architecture. Application-specific multiprocessors have the advantage of providing a machine architecture that can be easily tuned and extended to match the application's requirements, and on which applications can be directly ported.

1.2 Application-Specific Multiprocessor Systems Design

For many years, the computational performance improvements expected from House's [109, 155] derivation of Moore's Law [147] have been achieved through a combination of device, architecture and compiler advances. In the race to scale single-core processor performance, architects have crashed into the so-called *power wall* [29], which hindered Dennard scaling [61] in recent years. Profiting from transistor count increases, the shift to more energy efficient multi- and many-core designs aims to continue the proportional scaling of performance within a fixed power envelope [134]. This trend is also observed on embedded processors for battery powered mobile devices – where low power consumption is key – and ITRS predicts that the core count for multiprocessor systems will continue to increase in the near future by 1.4x per year [102].

According to the ITRS [102], applications are the driving force that push the limits of existing platforms and lead to the development of new devices. Wolf et al. state in [215] that embedded computer vision is an emerging domain that will continue to drive the development of new multiprocessor architectures. Embedded vision algorithms require vast amounts of computational power and possess a high degree of available parallelism that can be exploited to achieve high performance in multi- and many-core architectures.

When designing an application-specific multiprocessor system and developing the parallel application the designers needs to make a series of design choices. They must both size the multiprocessor to ensure enough resources are available to the application, while also selecting the most appropriate parallelization strategies to ensure the application makes efficient use of the available resources. As the architecture is customizable, a physical prototype often comes too late in the flow, when architects have already committed to a particular configuration. As the design grows more and more complex, it is thus important to provide the designers with tools that enable them to analyze a target application early in the design flow and quickly evaluate design configurations and parallelization strategies to select the ones which will render the best results.

A critical point in order to achieve high parallel efficiency, and thus, high system utilization,

is the load balancing. Some computer vision applications present very data-dependent behavior which can negatively impact their performance on parallel devices. Stefanizzi et al. [189] evaluated the performance of a face detection application written in OpenCL on a *General-Purpose computing on Graphics Processing Units* (GPGPU) system. Their experiments showed that the parallel performance on the GPU was highly impacted by the data-dependent behavior of the application which caused load imbalance.

Furthermore, although recent many-core parallel architectures such as GPU architectures have a high number of processing elements, their compute units are typically *Single Instruction Multiple Thread* (SIMT). The processing elements in such architectures can only execute one same instruction per cycle. As threads in data-dependent algorithms take different control flow paths, branch divergence penalties arise. Many-core architectures such as Kalray's *Multi-Purpose Processor Array* (MPPA) [59] and STMicroelectronics's STHORM [149] are composed of *Multiple Instruction Multiple Data* (MIMD) processor clusters that better cope with data-dependent algorithms [141].

As discussed by Czechowski et al. in [56], another important point is finding the right balance between communication and computation. As the core count increases and the memory bandwidth usage becomes higher, algorithms which are originally compute-bound on single-core architectures can eventually become memory-bound when ported to multi- and many-core architectures. Embedded vision algorithms are susceptible to this effect as well, as they process video streams in real-time and require significant memory bandwidth.

1.3 Parallel Design Aid Tools

Given the number of possibilities, parallelizing and tuning a parallel application to a particular multiprocessor architecture can be very time-consuming. Recent work on design aid tools that help in the parallelization process – described in Section 3.4 – focus primarily on desktop-class machines with a predetermined system configuration.

In order to account for dynamic data-dependent application behavior, the tools need runtime application profiling data. The ompP [75] tool, a parallel profiling tool and library, can generate runtime profiling reports of parallel code execution. It is certainly useful in that it helps in determining the time spent in each parallel section of the application, but it is intrusive and does not monitor important characteristics such as the memory accesses and data transfers. Furthermore, profiling tools and simulators require a working parallel version of the application for each execution run, resulting in an increased design and validation effort by developers.

Existing parallel performance prediction tools such as Intel's Parallel Advisor [97] focus on desktop-class machines with known system configurations. Parkour [106] and Kismet [105] are parallel performance prediction tools based on critical path analysis but, while they require less user intervention and can provide estimates of the maximum attainable speedup, they are often overoptimistic and lack the accuracy of mechanistic models. Parallel Prophet [119] is performance estimation tool based on a mechanistic model that includes cache effects, but do not include explicit support for embedded multiprocessors or for exploration of application parallelization scenarios.

1.4 Objectives

The objective of this thesis is to develop new tools and methods to aid in the parallelization of embedded computer vision applications on application-specific multiprocessor systems. More particularly, the work in this thesis targets STMicroelectronics multi- and many-core platforms

STHORM, and its successor STxP70 *Application-Specific Multiprocessor* (ASMP). As these are customizable application-specific platforms, the methods and tools proposed need to support the exploration and selection of both application parallelization options and multiprocessor platform's parameters.

Given an initially sequential application, the user should be able to analyze its hot-spots and test different configurations. The goal is to quickly and accurately evaluate different trade-offs in terms of an application's:

1. parallel granularity (image, tile, line or pixel);
2. data-parallel scheduling/workload distribution (static or dynamic);
3. working data placement; and
4. data transfer strategy:
    - individual/collaborative;
    - tiling.

1.5 Contributions

The selected approach consists in characterizing *(i)* the multiprocessor platform and *(ii)* target applications, and then combining these two information to help the developers choose the proper system configurations quickly and before committing to a particular implementation.

Characterization relies on the Trace Filter tool for automatic application task trace collection from simulator instruction traces. A profiler tool generates application call tree profiles from the collected task traces. An instrumentation library allows the control of the trace acquisition infra-structure and the explicit definition of tasks in the source code. Platform characterization is based on the *Edinburgh Parallel Computing Center* (EPCC) OpenMP microbenchmark suite, which was extended with new memory and DMA benchmarks. Collected microbenchmark task traces are processed by a characterization module embedded in the Parana tool, which can automatically produce a platform characterization database. Based on the platform and application characterization information, two design aid methods and associated tools were developed.

The first is a method for fast parallel performance estimation and *Design Space Exploration* (DSE) from sequential application traces for the STxP70 ASMP using the Parana tool. Parana is a trace-driven simulator that uses a mechanistic model of the STxP70 ASMP's OpenMP runtime to estimate the parallel performance of an application. It relies on platform characterization data and application traces to predict the performance of user defined parallelization scenarios. It produces a detailed report indicating not only the estimated runtime for a range of multiprocessor configurations, but a break-down of the execution time in cycle stacks.

The second method targets the optimization of $2D$ image tiling parameters in the STxP70 ASMP using the Tilana tool. First an analytical model is created to estimate the execution time of an image processing kernel on the STxP70 ASMP. The analytical model is used to define a constraint optimization problem. A constraint solver is then applied to find the tiling parameters that minimize the execution time for a given application kernel. The proposed method integrates non-linear constraints to model DMA data transfers and the OpenMP parallel runtime parameters for increased accuracy.

The following list summarizes the contributions of this thesis:

1. Extension of the EPCC OpenMP microbenchmarks with DMA and memory microbenchmarks for platform characterization and development of a trace collection framework and a characterization tool to automatically extract platform and parallel runtime characterization parameters from the microbenchmark traces.

2. A method for fast and accurate parallel performance prediction and early DSE of embedded multiprocessor architectural parameters and application parallelization strategies from sequential code;
3. A method for optimal tiling parameter selection that relies on non-linear constraints to minimize the execution time of an image processing application kernel.

1.6 Thesis Organization

The remainder of the thesis is organized as follows:

Chapter 2 presents the problems addressed in this thesis. A critical case study consisting of porting and parallelizing a face detection application on the STHORM many-core platform is presented. This case study allows to identify the main problems and difficulties in the design flow from a practical standpoint.

Chapter 3 reviews the background and the state of art in the domains of embedded computer vision applications, embedded multiprocessor systems and parallel design aid tools.

Chapter 4 presents the application trace collection and processing infra-structure and the details the characterization of the STxP70 ASMP multiprocessor platform, its OpenMP parallel runtime, its memory access latencies and its DMA data transfer times.

Chapter 5 presents the method for fast parallel performance estimation and DSE from sequential code and the associated Parana tool.

Chapter 6 presents the method for computing optimal tile dimensions via constraint programming and the associated Tilana tool.

Chapter 7 concludes this thesis and presents possible future research directions.

Chapter 2

Problem Definition

Abstract

The goal of this thesis is to design methods and tools that facilitate the selection of application-specific multiprocessor's architectural parameters and the development of efficient parallel computer vision applications on such systems. Embedded computer vision applications demand high system computational power and need flexibility to address different market segments. They constitute one of the key drivers of embedded application-specific multi- and many-core architectures, which provide the necessary computational power and flexibility. It is known, however, that a number of early system design choices can impact the final performance – among which the parallel granularity, the number of processors and the balance between computation and communication. The impact of these choices in the final system performance is not easy to assess early in the design cycle and there is a lack for tools that support designers in this task. This chapter introduces the central issues addressed in this thesis. A critical case study highlights the problems found when porting and optimizing an embedded vision application onto the STHORM many-core platform. These issues are recast as design challenges that support the need for fast and precise application-level performance estimation tools.

Contents

2.1 Introduction

This chapter introduces and defines the central problems addressed in this thesis. The goal of this thesis is that of providing practical tools and methods to help designers to parallelize their applications on an embedded application-specific multiprocessor system and quickly explore different implementation trade-offs. The central problems of the thesis are introduced by means of a practical critical case study, whose objective is to highlight the relevance of identified issues and discuss how addressing these points could improve existing development flows.

The chapter starts by presenting the critical case study in which a face detection application is ported and parallelized on the STHORM [149] many core accelerator using the C language and the OpenCL programming model. A methodology for application performance optimization is applied to determine the best trade-offs in the parallel design process. The development tools used are the STHORM *Software Development Kit* (SDK) and the included STHORM cycle-approximate Gepop simulation platform. Simulation results are compared against those of a physical prototype. Based on the experiments, the main limiting factors for application-level performance optimization are identified and discussed. Such factors are broken down into three categories: those inherent to the application behavior, those due to the OpenCL parallel programming model, and those arising from the STHORM environment. This critical case study is used to identify the central problems with the existing tools and introduce the investigations and contributions of this thesis.

The remainder of this chapter is structured as follows. Section 2.2 presents the STMicroelectronics' multiprocessor architectures targeted in this thesis and provides an overview of the parallel runtimes they support, as well as of the available development tools. In the sequence, Section 2.3 details the face detection case study. It starts with a brief description of the face detection application in Subsection 2.3.1, and then outlines the methodology followed in Subsection 2.3.2. Subsection 2.3.3 describes the hot-spot analysis and the selection of the parallelization candidate code sections. The parallel implementation is discussed in Subsection 2.3.4, with the ensuing results presented in Subsection 2.3.6. Section 2.4 discusses the main issues observed, while Section 2.5 highlights the design challenges identified in the process. Finally, Section 2.6 summarizes and concludes this chapter.

2.2 STMicroelectronics's Multiprocessor Architectures

2.2.1 STxP70 Processor

STMicroelectronics' STxP70 processor is an extensible processor for embedded and real-time applications. This processor is the base processing element in both STHORM and STxP70 *Application-Specific Multiprocessor* (ASMP). It has an in-order 32-bit *Reduced Instruction Set Computer* (RISC) architecture with seven pipeline stages. It can be configured as either a single-issue or dual-issue processor, for a theoretical maximum *Instructions per Cycle* (IPC) of 2. As the processor has a single data memory port, load/store instructions are always executed on the first lane.

The STxP70's *Instruction Set Architecture* (ISA) has a variable length encoding for compactness and allows manipulation of 32-bit, 16-bit or 8-bit data. Its ISA can be further enriched via so-called instruction-set *extensions*. The latter consist of tightly coupled hardware accelerators that group a set of custom instructions and an independent register file. Extensions are modular and can be easily shared among projects. Moreover, they can be used to seamlessly add new capabilities to the STxP70 processor, such as adding hardware floating point support via the FPx extension, or 256-bit *Single Instruction Multiple Data* (SIMD) vector operations via the VECx extension.

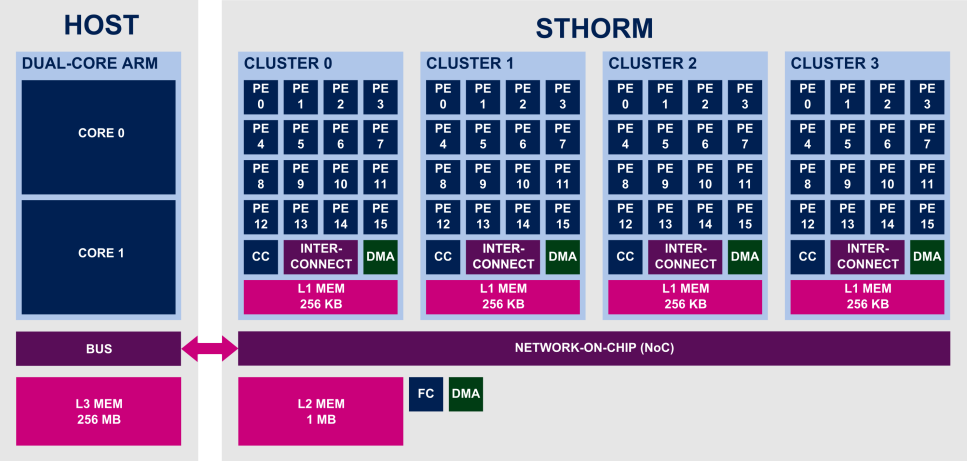


Figure 2.1: *STHORM many-core architecture block diagram in a programmable accelerator configuration paired to a dual-core ARM host processor.*

2.2.2 STHORM Many-Core Processor

STHORM [149] is a many-core processor designed by STMicroelectronics to handle compute intensive embedded applications. It is derived from Platform2012 [36], a joint effort between STMicroelectronics and CEA (Atomic Energy and Alternative Energies Commission), and can be used either as a stand-alone processor or as an accelerator coupled to a host processor. STMicroelectronics discontinued the STHORM platform in 2013, with the STxP70 ASMP platform becoming its follow-up standard embedded multiprocessing solution.

2.2.2.1 STHORM Architecture

STHORM has a scalable architecture, organized as clusters of processing elements(PEs), configurable from 1 to 4 clusters with up to 16 processing elements each [149]. Figure 2.1 shows a high-level block diagram of the STHORM architecture. The cluster's processing elements are dual-issue STxP70 processors with a floating point extension. The clusters are interconnected via a *Network On Chip* (NoC) and have integrated *Digital Voltage and Frequency Scaling* (DVFS) capabilities that can be controlled on a per-cluster granularity. STHORM counts with one additional STxP70 processor that acts as a chip-level fabric controller (FC) that controls the communication infra-structure. Additionally, each cluster also counts with one extra STxP70 processor that acts as a dedicated cluster controller (CC) responsible for controlling cluster-level features such as DVFS.

Internally, each cluster counts with 256KB of shared memory accessible by all processors in the cluster. In order to reduce the probability of conflict, the memory is organized in 32 banks with address interleaving. The logarithmic interconnect, with a mesh-of-trees (MoT) [171] topology, provides concurrent single-cycle memory access. In case of conflict – when two or more processors access a same bank simultaneously –, a single request is serviced per cycle, while the remaining requests keep pending. Although processors block until their request is serviced, a round-robin mechanism ensures fair access to the contended resources.

The STHORM architecture has three memory levels: L1, L2 and L3. All memory zones are directly accessible by each processor via a single global addressing space. The platform has no data cache, only instruction caches. It consists thus of an explicitly managed memory system, where data transfers must be explicitly handled by the programmer. A cluster's local

shared data memory is refereed to as the L1 memory. The next memory level, the L2 memory, consists of an intermediate capacity integrated data memory shared by all clusters and accessed via the NoC. Finally, the L3 memory is an external DDR memory with a larger capacity, but much higher access latency. Processors can access each memory level via direct memory load and store operations, albeit with consequent latency penalties, or by explicit use of the DMAs. Each cluster has a dedicated DMA, shared by all of its processing elements, which can be used for transferring data between the L1 and L2/L3 memories. An additional centralized DMA is used for transfers between L2 and L3 memories.

2.2.2.2 STHORM Programming Models

The standard parallel programming model supported by the STHORM architecture is the OpenCL 1.1 [114] programming model. The OpenCL programming model is described in details in Section 3.4.4.2. A dedicated STHORM SDK based on the Eclipse *Integrated Development Environment* (IDE) included an OpenCL compiler, the STHORM Gepop simulator, a cycle-approximate simulator of the STHORM platform, as well as analysis tools such as a OpenCL kernel-level profiler and a timeline visualization plugin. Tentative support for OpenMP at cluster level has been investigated, but was never included in an official STHORM SDK and was only made officially available in its successor platform, the STxP70 ASMP (see Section 2.2.3.2).

Although not officially supported in the STHORM SDK, some of the programming models adopted in the Platform 2012 project, from which STHORM is derived, were still used with STHORM. Although these programming models were not officially supported, a list of said programming models is provided for completeness. These programming models, as listed in [163], are:

1. Standards-based programming models, namely the OpenCL 1.1 [114] programming model.
2. Native programming models, such as:
   (a) Parallel Programming Patterns (PPP), a model consisting of a set of parallel programming patterns implemented in a component framework.
   (b) Predicated Execution Synchronous Dataflow (PEDF) model, a streaming-oriented programming model targeted at applications that made use of in-cluster hardware accelerator engines.
   (c) Dynamic Task Dispatching (DTD), a model for fine-grain task scheduling on a single cluster.
3. Native programming layer (NPL), consisting of a low-level *Application Programming Interface* (API) for directly accessing platform resources, but which lacked portability across different variants of the platform.

2.2.3 STxP70 ASMP

The STxP70 ASMP is an embedded multiprocessor with an explicitly managed, shared memory architecture, and a centralized DMA. By explicitly managed memory, we mean a system with no data cache and that relies on explicit DMA calls to move data across the memory hierarchy. The STxP70 ASMP is analogue to a single cluster of the STHORM many-core platform described in Section 2.2.2.

2.2.3.1 STxP70 ASMP Architecture

The STxP70 ASMP is a configurable SMP architecture with up to 16 STxP70 cores in a dual-issue configuration. Figure 2.2 depicts the architectural template of the STxP70 ASMP. It counts with a one cycle access shared L1 data memory, organized in several banks with

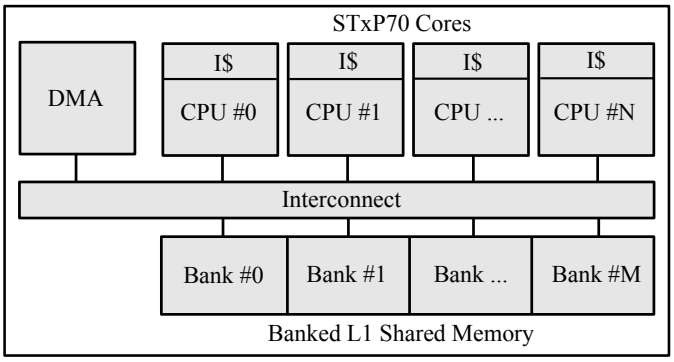


Figure 2.2: *Architectural template of the STxP70 ASMP.*

interleaved addressing, all of which can be accessed simultaneously. In case of bank access conflict, processors block until access is granted by a round-robin arbiter that ensures fair access for all processors. Additionally, each processor has 16KB of program cache (I$), but no data cache. The physical prototype for the STxP70 ASMP platform is based on an *Field Programmable Gate Array* (FPGA) implementation whose configuration is limited to 8 STxP70 cores and 512 KB of shared data memory, organized into 32 memory banks. Although the effective clock frequency in the FPGA implementation is of 40 MHz, timing figures used in experiments are computed from cycle counts and reported considering a clock frequency of 500 MHz for all implementation platforms. Support for user-defined instruction set extensions is envisaged, although it is not yet supported by the development tools.

2.2.3.2 STxP70 ASMP Programming Model

The standard parallel programming model supported by STMicroelectronics's STxP70 ASMP is the OpenMP 2.5 [159] programming model. In the OpenMP programming model, the application code is annotated with a series of *pragmas* that are interpreted by a compiler to produce parallel code. An OpenMP runtime is linked with the application and is responsible for thread creation, dispatching, parallel scheduling, synchronization and thread termination, among others. Upon compiling code with OpenMP *pragmas*, the compiler automatically applies the necessary transformations and adds calls to the OpenMP runtime. The user can also explicitly call OpenMP runtime functions in application code. If parallel code generation is disabled in the compiler options, the OpenMP *pragmas* in the user code are ignored producing a sequential executable. This is especially useful for debugging purposes, for evaluating the actual parallelization benefits over the sequential version of the application, and for ensuring portability between sequential and parallel systems. A more detailed description of the OpenMP programming model is provided in Section 3.4.4.1.

2.3 Critical Case Study: Parallelization of a Face Detection Application on STHORM

This critical case study's interest is to evaluate the process of porting and parallelizing a representative embedded vision application on STMicroelectronics's STHORM architecture. Flyvbjerg states in [71] that "a critical case can be defined as having strategic importance in relation to the general problem" and supports the use of case studies as an important tool for in-depth analysis of a research problem. This case study is thus used as a tool to first map the issues arising from the parallelization process and from STHORM's own development environment

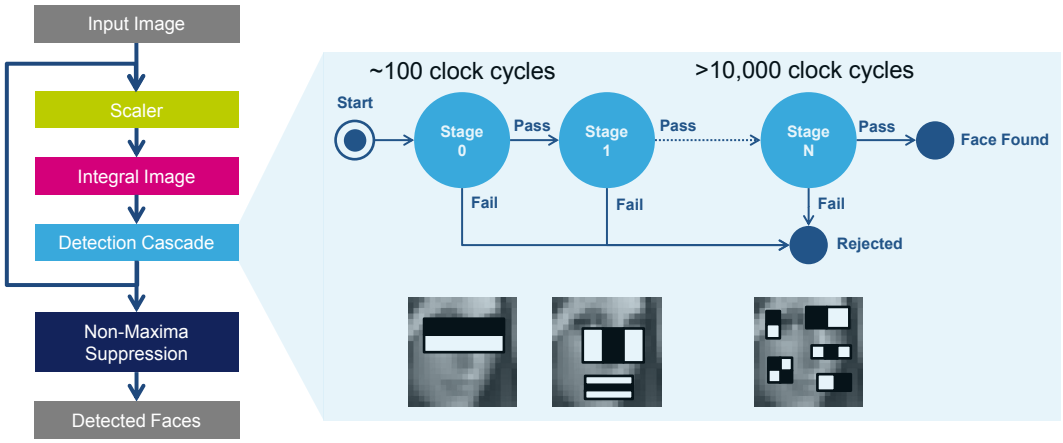


Figure 2.3: *Face detection algorithm's main steps. The detection cascade's inner structure is composed of a series of stages, each with an increasing number of features.*

and then cast them in the form of design challenges that represent research opportunities for the work in this thesis.

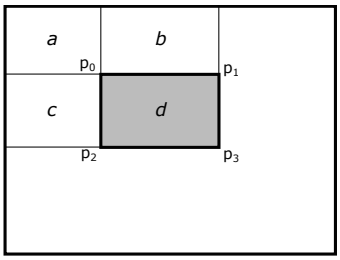
The critical case study starts from an existing sequential face detection application and defines the main steps necessary to derive a parallel implementation. Experiments are done in order to determine the application's final performance on the STHORM architecture. Comparisons are done between a set of implementation choices to try to select the most efficient one. The results indicate that existing tools do not allow the acquisition of precise application-level performance metrics prior to a functional physical prototype implementation. Moreover, the mismatch observed between simulation and physical prototype results further limit the usefulness of the simulation as a means of evaluating different parallelization alternatives early in the design cycle.

2.3.1 Application Description

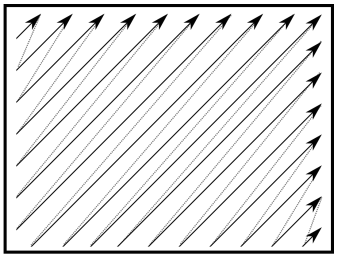
This section describes the sequential face detection algorithm used as a starting point for this study. The algorithm used is based on the approach originally proposed by Viola and Jones in [210]. In this approach, the detector is structured as a cascaded classifier based on Haar-like features, and trained offline using AdaBoost [74, 179]. Figure 2.3 depicts the main steps in the face detection application which will be the detailed next.

The detector is applied to an integral image, an image representation where the value of each pixel corresponds to the sum of all pixels above it and to its left on the original image. Figure 2.4(a) shows an integral image containing a region of interest and its four corner points. The integral image representation allows to compute the sum of the pixel values in the region of interest d by means of simple arithmetic operations on four points of the integral image, as follows: $d = p_3 - p_2 - p_1 + p_0$. Such an integral image can be constructed either using a raster scan or a wavefront scan, shown in Figure 2.4(b).

The classifier is evaluated at regular intervals using a scanning window technique over an image pyramid to achieve scale invariance. Figure 2.5 illustrates this multiscale detection process, where a detector is scanned over a set of decreasing resolution versions of a same input image. More precisely, unlike the original Viola and Jones approach in [210] which scales the detector, in this particular implementation multiscale detection is achieved by scaling the image while keeping detector size constant. This approach is more suited to memory constrained embedded systems, as it presents a lower local memory footprint. For more details, refer to [182].



(a) *Example of a feature computation from integral image*



(b) *Wavefront scan for creating an integral image*

Figure 2.4: *Integral image and a region of interest with four corner points in Figure 2.4(a). Representation of the wavefront scan order for building an integral image in Figure 2.4(b).*

Figure 2.5: *Multiscale processing via a series of lower resolution images forming an image pyramid. For faces to be detected, their dimensions need to be similar to the detector size. In order to detect faces at any distance from the camera, the 20x20 pixel detection window is scanned over all possible image pyramid locations.*

The complete face detection algorithm pipeline consists of the following four main steps:

1. *Scaling*: produces a lower-resolution image pyramid level from the original input image.
2. *Integral Image*: computes the integral image of the down-scaled pyramid image.
3. *Detection Cascade*: runs the face detection cascade on the integral image and reports positive detection locations.
4. *Detection Merging*: consists in fusing multiple overlapping detections by averaging the coordinate values of their corner points.

The first three steps are executed for every image pyramid level, while the final step is executed once at the end of each frame. Figure 2.5 depicts these four main steps and details the detection cascade structure.

The classifier cascade is organised as a series of stages with the goal of rejecting non-face locations as early as possible. Stages have a set of features from which a response is computed and tested against bounds defined at training time. If the stage response falls within bounds, the classifier proceeds to evaluate the next stage, otherwise, the detection is aborted and the current window is rejected.

The first stages execute more often and have very few features. Each new stage is more selective, checking for finer details with a growing number of features to evaluate. Finally, if all stages in the classifier cascade are successful, the window is accepted and a positive detection result is reported at that location. At the final detection merging step, neighboring detections are fused based on a distance metric and assigned a score. Solutions with a low score are discarded.

Notice that while a cascaded execution results in a reduced overall execution time, its behavior is highly data-dependent. The amplitude of the difference in execution time of detection cascades for different windows presents a real challenge for parallelization of the code. This is especially true on vector-thread architectures such as GPUs, as discussed by Stefanizzi et al. in [189].

2.3.2 Methodology

It is known from Amdahl's law [25] that the speed-up of a parallel application is limited by the execution time of its sequential portion. As such, the rationale used is to parallelize the portions or steps of the application which contribute the most to the execution time first, as these have the highest potential for application-level acceleration.

The initial step is to profile the application to identify its hotspots. The profiled functions are then ranked in descending order of their cumulative execution time. The top ranked functions are the initial candidates for parallelization. These functions are refactored into OpenCL kernels with clear inputs and outputs.

The following step is to define the parallelization strategy. Some key parallel implementation decisions must be made at this point, as they will impact the parallel performance. Important points to consider at this stage are the overlap of computation and communication and the load balance. Some of the key levers for application and system designers are:

- the parallel granularity, e.g.: image frame, line, window, or pixel;
- the workload distribution strategy: static or dynamic workload distribution;
- the working data placement: global, local or private spaces;
- the data transfer strategy: individualistic or collaborative.

Besides the aforementioned points, the STHORM OpenCL simulator and runtime can also be parameterized with different configurations in terms of the number of physical clusters and processing elements of the target platform. Furthermore, when launching an OpenCL kernel, its local and global work-group dimensions also need to be tuned.

This performance optimization flow is iterative, with simulations done to estimate the impact of different design choices in the execution time. Its goal is to determine the best architectural parameters and algorithm parallelization strategy in order to meet the functional and non-functional requirements of the application. The number of design points can be consequent. Therefore, the simulator needs to be fast enough to allow iterative design space exploration, and precise enough to allow comparison of different alternatives and to ensure that the final implementation will meet the application requirements.

2.3.3 Hot-Spot Analysis

The goal of the application hot-spot analysis is to determine the code sections which contribute the most to the execution time. The first difficulty found for performing this analysis on STHORM, in the host-accelerator configuration, is that the sequential application executes on the ARM host processor, but the STHORM SDK does not include any profiling or tracing support for the host. This host-accelerator configuration, depicted in Figure 2.1, is however the only configuration supported by the STHORM SDK.

Alternatively, the reference sequential application was profiled on a cycle-approximate simulator of the STxP70 processor, in the same configuration as that of a STHORM's *Processing Element* (PE). The analysis presented in this section is based on the worst-case QVGA image in STMicroelectronics' internal face detection test database, an image with 24 faces, designated herein simply as the test image.

Table 2.1: *Profiling results for the sequential face detection application on the worst-case image (24 faces) in ST's face detection test database. The results are obtained on a cycle-approximate STxP70 500 MHz.*

Application Phase	Cycles	Time (ms)	% of Total
Detection Cascade	61,879,829	123.8	56.8%
Integral Image	27,159,728	54.3	24.9%
Scaler	14,627,913	29.3	13.4%
Other	5,196,975	10.4	4.8%
Total	108,864,445	217.7	100.0%

Table 2.1 reports the profiling results for the test image, grouped by algorithmic phase and ranked according to their cumulative execution cycles. Note that, while these figures cannot be directly compared to the global application-level performance on STHORM, they represent reference times for the parallel kernels. The three hotspots identified are: the classifier cascade, the integral image generation, and the scaler. These three phases together account for $\sim 95\%$ of the execution time, and were thus selected as candidates for parallelization in this critical case study.

2.3.4 Parallel Implementation on STHORM

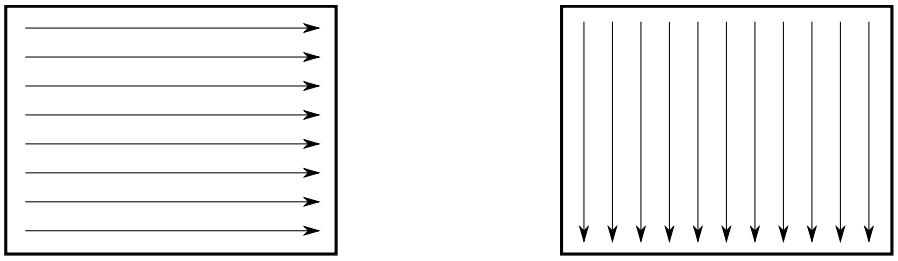
2.3.4.1 Partitioning into OpenCL Kernels

Once the parallelization candidates were determined from the hot-spot analysis, they were refactored into OpenCL kernels. The goal of this methodology step is to refine the functionalities covered by each kernel, and possibly either fusion them or break them apart. In this case study, each kernel execution processes a single image pyramid iteration, with multiscale detection requiring successive executions of the kernels. Figures 2.5 and 2.3 depict the application flow.

Since OpenCL allows no local or private data-persistence between kernel executions, an important factor to consider is the memory bandwidth required to stream data in and out of the cluster's shared memory for each kernel. In order to exploit data locality, the integral image generation was fused into the classifier kernel, allowing to keep the integral image in the cluster's local memory at all times. This results in a factor four external memory bandwidth reduction, as the integral image data is 32-bit, while the input image data is only 8-bit. This is thus a trade-off between having to recalculate integral image data for overlapping zones, versus transferring more data. The important point to note is that this represents a trade-off that would need to be evaluated by the user in a more in-depth evaluation and the tools should provide the needed support to evaluate this trade-off. Regarding the remaining functionalities, no sensible advantage was identified in re-partitioning scaler or detection merging functionalities and thus they were kept as independent kernels.

2.3.4.2 Scaler Kernel

The scaler kernel implements a bilinear scaler and, as such, produces an output pixel by interpolating four input pixels. An outer loop processes each line of the output image, whereas an inner loop processes each pixel of a line. As the computation is well balanced and not data-dependent, a simple static workload allocation scheme is used in which a set of lines is allocated



(a) *First pass: horizontal accumulation*

(b) *Second pass: vertical accumulation*

Figure 2.6: *Representation of the scan order in a two-pass approach for building an integral image.*

to work-items[1]. In this scheme, a work-item is assigned a number of consecutive lines to process based on its global ID and the total number of available global work-items. This scheme also advantages scalability, since it seamlessly partitions the load across processing elements in all clusters.

On each outer loop iteration, a processing element will:

1. fetch the input image lines from global memory,
2. compute the output line in the local memory, and
3. write-back the resulting output image line to global memory.

In order to hide the data transfer latency, double buffering is implemented for both input and output image data transfers. This scheme essentially results in a software pipeline with three stages – fetch, process and write-back.

2.3.4.3 Classifier Kernel

The classifier kernel encompasses the integral image generation and classifier cascade execution. Experiments were done with two different data transfer strategies, *collaborative* and *individualistic*, which are detailed in the next section.

Integral image generation. Similarly to the scaler phase, the integral image generation is not data-dependent and thus well-balanced. Many methods to compute the integral image exist, but a two-pass approach can reduce the number of operations [120] and is amenable to parallel implementation [219]. Figure 2.6 depicts the selected two-pass approach where the first pass consists of an horizontal scan that accumulates the elements in each line, while the second pass consists in a vertical scan that accumulates the elements in each column. On the parallel version, a static allocation of lines and columns to processing elements is used for the first and second pass, respectively, with barriers after each pass to ensure correct synchronization.

Classifier cascade. In this phase the classifier is applied to integral image windows. Figure 2.3 depicts the internal structure of the classifier cascade. Since this computation is very data-dependent and presents a highly variable execution time for different windows, a dynamic workload allocation scheme is used where work-items increment an atomic counter to determine the next window to process. In case of a positive detection result, the work-item adds an entry with the coordinates of the window to a list residing in the global memory. A global atomic

[1] Work-item: OpenCL's terminology for a charge of work submitted to one processing element as part of a work-group executing on a compute unit. It represents one element in a collection of parallel executions of a kernel invoked on a device by a command, and is distinguished from other executions within the collection by global and local IDs.

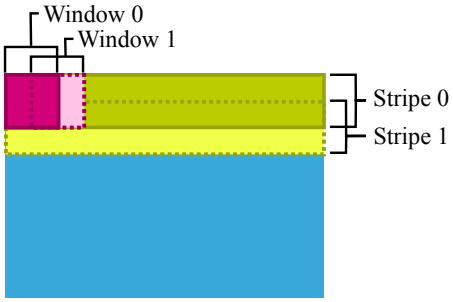


Figure 2.7: *Representation of the stripe and window concepts. A stripe is an full horizontal band of the image. A window is a smaller region of interest to which the detector is applied. The latter is typically square and it might be contained inside a stripe.*

counter determines the next available position in the list. Scalability is achieved by partitioning the input image in horizontal image stripes allocated to different work-groups.

2.3.4.4 Data Management

The classifier cascade constant data placement is also critical. The STHORM OpenCL runtime places cascade data in STHORM's L2 memory by default. Since it has higher latency and the cascades are frequently accessed, cascade data is explicitly copied to the local memory at the start of a work-group execution. This is true for both the collaborative and individualistic data transfer approaches for the Classifier Kernel.

Figure 2.7 depicts the representation of the image stripe and window concepts. A stripe is a full horizontal band in the image, while a window delimits a zone to which a detector is applied. The latter is typically square and might be located inside a stripe.

Collaborative approach. In the collaborative approach, work-groups load entire horizontal image stripes via a work-group copy call[2], with double buffering on input to hide the latency of loading subsequent stripes. Figure 2.7 depicts the stripe and window concepts. Allocation of image stripes to work-groups is static. The integral of the entire image stripe is then computed in parallel by local work-items. A barrier call synchronizes work-items prior to starting the classifier phase, with dynamic allocation of windows in an image stripe to work-items. A second barrier call ensures all work-items finished executing the classifier cascade on the current stripe prior to moving to the next stripe.

Individualistic approach. In the individualistic approach, each work-item fetches and processes an image window autonomously. Work-items obtain the index of the next window to process dynamically and load the window into local memory via a work-item copy call. Once the transfer is complete, the work-item generates the integral image for the window on a private buffer. Then the classifier evaluates the window and reports any successful detection. Barriers are not needed in this case, since work-items are completely autonomous. With the dependency among work-items removed, load balancing can be improved at the expense of increased data transfer and computation.

2.3.5 Experimental Setup

The experiments in this case study are designed to compare the performance of the two parallel implementation strategies detailed previously, both on a simulator and on a physical

[2]OpenCL's API call for transferring data at work-group level.

Table 2.2: *Cumulative execution time for the OpenCL kernels of the face detection application on the STHORM simulator and prototype, in a configuration comprising 4 clusters of 16 processing elements clocked at 500 MHz. % are relative to the total time.*

	Simulator				Prototype			
Data Transfer Strategy	Collaborative		Individualistic		Collaborative		Individualistic	
Kernel Processing Time	6.9 ms	21.5%	7.8 ms	34.7%	44.9 ms	26.3%	12.4 ms	13.9%
Kernel Prolog & Epilog	13.6 ms	42.2%	14.1 ms	62.4%	47.9 ms	28.0%	32.4 ms	36.1%
Time Spent in Runtime	11.7 ms	36.3%	0.7 ms	2.9%	78.0 ms	45.6%	44.8 ms	50.0%
– Asynchronous Copies	0.3 ms	0.9%	0.3 ms	1.2%	0.0 ms	0.0%	1.6 ms	1.8%
– Waiting for Events	0.4 ms	1.1%	0.3 ms	1.2%	30.2 ms	17.7%	43.2 ms	48.2%
– Waiting on Barriers	11.1 ms	34.3%	0.1 ms	0.5%	48.3 ms	28.3%	0.0 ms	0.0%
Total Time in Kernels	32.2 ms	100.0%	22.7 ms	100.0%	170.9 ms	100.0%	89.7 ms	100.0%

prototype. The goal is to identify the main problems found when porting and, parallelizing an application onto the STHORM platform from practical experiments.

Experiments take the form of simulation runs with the Gepop simulator in the STHORM SDK version 2013.2. Gepop is an STMicroelectronics' proprietary modular cycle-approximate simulator engine. The simulator models an heterogeneous platform with an ARM processor as OpenCL host and the STHORM many-core accelerator as an OpenCL device. The Posix-XP70 configuration of the simulator is used, which is functional for the host, and cycle-approximate for the device. No cycle-approximate simulator for the host is available in the SDK.

A STHORM prototype board is used for comparison. It counts with an ARM host and a STHORM device fabricated in STMicroelectronics's 28 nm process. The L3 memory is connected via a bridge with a bandwidth of 400 MB/s, while the L2 and L1 memories are integrated into STHORM. In both cases, STHORM is setup in a configuration with 4 clusters of 16 processing elements running at 500 MHz.

2.3.6 Results

2.3.6.1 Performance Measurements

The performance measurements reported in this section list the cumulative execution time for the OpenCL kernels of the face detection application on the STHORM simulator and prototype. Table 2.2 provides the list of the performance measurements for the worst-case QVGA image (24 faces) in STMicroelectronics's test database, for both the collaborative and individualistic approaches. The results in the leftmost pair of columns are obtained from simulator runs, while the results in the rightmost pair of columns are obtained from runs on the prototype board. The *kernel processing time* reflects effective processor computation time and local memory accesses, *kernel prologue and epilogue* accounts for overheads in launching and terminating kernels, while the *time spent in runtime* encompasses the asynchronous data transfer time, as well as the time spent waiting for events and on synchronization barriers.

These results show that the collaborative version is negatively impacted by the synchronization barriers, which take roughly a third of the *total time in kernels*. The individualistic version provides better overall performance at both simulator and board, mainly due to the reduced synchronization overhead. The highest source of inefficiency according to the simulator results is the kernel prologue and epilogue.

The simulator results indicate that the collaborative approach has smaller *kernel processing time* (6.9 ms) than the individualistic approach (7.8 ms). However, the prototype results show an

inversion, with the collaborative approach presenting a higher *processing time* (44.9 ms) than the individualistic approach (12.4 ms). Furthermore, while the time spent *waiting for events* is close to 1% on the simulator, it can amount to nearly half of the *total kernel time* on the prototype board. Thus, although the STHORM simulator is said to be cycle-approximate, a large mismatch between the simulator and the prototype board results was observed. The inversion in the *kernel processing time* for both configurations on the prototype versus the simulator, while not resulting in an inversion of the *total time in kernels*, could be problematic as it might possibly lead to the selection of a worse configuration if a decision is made based solely on simulation results.

2.3.6.2 Detailed Analysis

A detailed analysis of the results listed in Table 2.2 show that on the simulator the highest contributor to the total time is the time spent in the *kernel prologue and epilogue*. Figure 2.8 shows a partial trace visualization of the collaborative approach execution, from which it can be seen that the *kernel prologue and epilogue* accounts not only for the time to launch and terminate kernels, but to any interstices between work-group executions where the cluster is idle. These typically arise due to inter-work-group load imbalance, as the data-dependent behavior causes some work-groups to finish earlier than others, or due to the interaction with the host processor.

The *kernel processing time* results on the prototype board are higher than on the simulator. The reason is that the STHORM simulator does not accurately model memory access times, which, except for DMA transfers, are accounted for in the *kernel processing time*. The simulator does not model memory conflicts. This, together with the higher latency and limited bandwidth to the global memory on the prototype board, leads to a high mismatch between simulator and board. Moreover, as synchronization barriers require all processors to reach the barrier call to proceed, the increased processing time will cause processors on the critical path to take longer to reach the barriers, and thus lead to increased time *waiting on barriers*.

The time spent *waiting for events* is the figure with the highest mismatch between the simulator and the prototype. When launching a DMA transfer via a non-blocking asynchronous copy, an event handle is returned by the runtime. Processors can perform other operations asynchronously and then do a wait call on the event handle, which returns only when the transfer is complete. Thus, the time spent *waiting for events* in the experiments actually corresponds to the time waiting for non-blocking DMA transfers to complete. The high mismatch indicates that the simulator does not precisely model the DMA transfer times found on the prototype. No parameters are available in the STHORM SDK to compensate for this mismatch.

The total time lost due to load imbalance cannot be precisely estimated from the figures provided, since they do not discriminate among different contributing factors. Nonetheless, a large portion of the *kernel prologue and epilogue* is relative to inter-work-group imbalance, as shown in Figure 2.8, and the time lost due to inter- and intra-work-group load imbalance could amount to up to 70% of the *total kernel time* for the collaborative approach on the simulator.

The individualistic approach virtually eliminates intra-work-group imbalance, but still presents high inter-work-group imbalance. Even though the time spent waiting for data transfers using the individualistic approach on the board is higher, it still yields better performance than the collaborative approach, as the latter incurs higher memory conflict penalties and presents worse load balance.

2.3.6.3 Discussion

The face detection application presents a data-dependent behavior, which leads to load imbalance in parallel platforms. Different parallelization strategies could provide better load balance, but each strategy would need to be implemented and simulated, which is time-consuming.

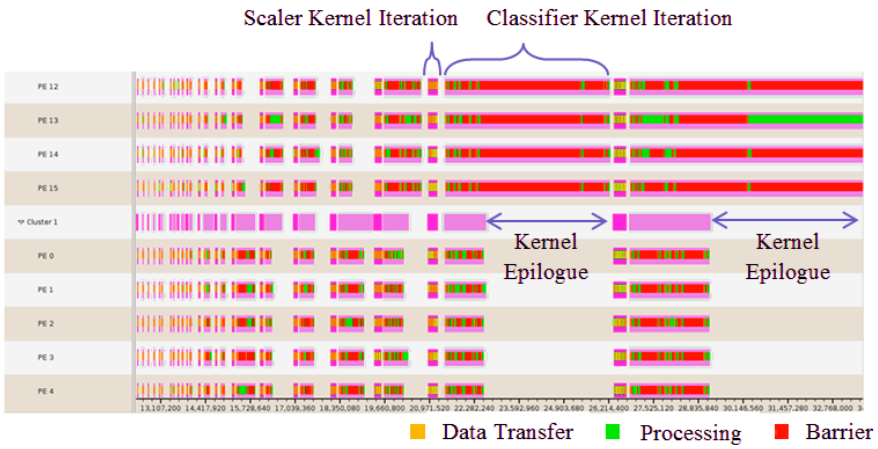


Figure 2.8: *Portion of a trace for the face detection collaborative approach on STHORM. It shows the kernel execution traces of multiple image pyramid iterations for a single image frame, from smaller pyramid levels on the left to larger pyramid levels on the right.*

A larger number of parallelization strategies could however be evaluated even before having a parallel implementation using tools for early parallel performance estimation.

The OpenCL programming model has limited data placement options, which hinders optimal data placement. Custom vendor extensions are needed to enable fine tuning of the data placement. Control over the scheduling is also limited, leading to processing gaps due to inter-work-group load imbalance. A more dynamic kernel enqueueing mechanism as proposed in OpenCL 2.0 [115] could provide better load balancing, but was not yet supported by the STHORM SDK.

As the STHORM SDK provides no cycle-approximate simulator for the host, it is not possible to estimate the global application-level performance. On the device side, although using a cycle-approximate simulator of the STHORM processing elements, a large mismatch has been observed between the simulator and the prototype board results.

The analysis points that the largest mismatches arise from the memory models. The inclusion – or calibration – of the memory latency and bandwidth parameters in the SDK, the modeling of local memory conflicts on the simulator and the usage of a cycle-approximate host simulator could reduce the mismatch and allow to obtain application-level results. Addressing these issues should enable application-level profiling and optimisation with enough precision, so that algorithm-architecture co-design trade-offs can be evaluated early in the design flow.

2.4 Observed Issues

The face detection case study and the associated experiments revealed a number of issues that arise when parallelizing an application onto the STHORM many-core platform. In this section, the main problems identified are listed and a brief discussion ensues.

No application-level performance measurements and profiling. The STHORM SDK did not count with a cycle-approximate simulator of the host processor, nor any other means to acquire timing or profiling data for it via simulation. Only by means of the prototype board was it possible to acquire precise timing information for the host. The physical prototype, however, has a fixed architecture and limitations such as the use of an external memory bridge with limited bandwidth. Additionally, profiling on the board is implemented via sampling, which limits the

timing resolution for observed effects and might be susceptible to aliasing effects, as discussed by McCanne et al. in [140].

Large mismatches between Gepop simulation results and prototype board results. The results of the face detection case study experiments showed a large mismatch between the simulation and prototype results. Although it is difficult to determine precisely the sources of this mismatch, several contributor factors were identified, such as no timed- or cycle-approximate simulator for the host, and an imprecise memory model.

Design space exploration comes late in the flow and is time-consuming. Given the large simulation time mismatches observed and the lack of application-level performance analysis tools in the STHORM SDK, effective design space exploration of the parallelization strategies and architectural parameters can only be done late in the flow. A fully-functional parallel version of the application is necessary for every design point to evaluate, which must then be simulated individually. Exploring the trade-offs of different parallelization strategies, or determining the optimal tiling granularity for instance, is thus very time-consuming.

Impossible to discern the processor idle time due to load imbalance. The list of results generated by the profiling tool for the STHORM OpenCL kernels does not allow to distinguish important factors for parallel application design, such as the load imbalance.

These are the main issues identified via the face detection case study. Addressing these will result in consequent time savings that will lead to increased developer productivity. The latter will translate in cost savings and more performing systems thanks to broader design space exploration.

2.5 Design Challenges

The design challenges ahead consist in providing solutions for the issues listed in Section 2.4. In an ideal development environment the designer should be able to:

Obtain fast and accurate application-level performance measurements. As has been shown in this case study, the STHORM tools were not able to provide application-level performance measurements. The OpenCL kernel-level measurements reported by the simulator were inaccurate, which might lead a developer to make design decisions that would render suboptimal results in the final physical implementation. It is thus necessary that the results reported by any of the tools the user relies to assess application performance are accurate.

Access to detailed profiling tools and measurements for analyzing application hot-spots. Having access to profiling tools is necessary not only to identify application hot-spots, but also to discover what causes them and how they can be eliminated or at least mitigated. This is a starting point in any performance optimization process.

Be able to discern and identify the factors that might limit an application's speedup. In the STHORM SDK results, it was not possible to precisely determine the load imbalance time in data-parallel loops. This is however a critical parameter in parallel application design. It is thus important to define meaningful categories that allow the precise identification of the factors that limit an application's speedup and parallel efficiency.

Quickly perform parallel application and multiprocessor design space exploration as early as possible in the design flow. While useful to functionally validate an application and to acquire performance metrics, simulators still require a fully functional parallel version of the application code. Furthermore, each design point needs to be simulated independently, which can be time-consuming. Tools for early parallel performance prediction and design space exploration can be helpful in quickly analyzing and selecting promising parallelization strategies and determining the multiprocessor configurations early in the design flow.

Analyze the trade-offs between the data transfers and computation. When partitioning the application's data-parallel loop iterations among threads, the developer should be able to evaluate and determine the partitioning scheme that leads to better platform utilization and optimal execution time. Determining these factors via simulation is not very efficient as it requires simulating multiple parameters and then selecting the best option. Modeling the execution time analytically and including tool support for automatically determining the best trade-offs in this partitioning would lead to faster and better results.

2.6 Conclusion

This chapter presents the central problems addressed in this thesis. We are particularly interested in addressing the lack of methods and tools for early, fast and accurate application-level parallel performance estimation and optimization when developing or porting applications on STMicroelectronics' multiprocessors. This claim is supported by a preliminary case study, which consisted in porting and parallelizing an application onto STMicroelectronics' STHORM many-core platform.

The STHORM platform was discontinued in 2013 and gave way to a derived platform, the STxP70 ASMP. Although the present chapter focuses on problems originally identified on the STHORM platform, the design challenges identified are also valid for other embedded multi-processor platforms, including STHORM's successor, the STxP70 ASMP platform. Subsequent chapters will present the methods and tools conceived to aid designers in the parallelization process, by providing early estimates of the parallel performance that are fast and accurate, and that allow to select optimal parameters for image tiling.

Software developers often focus on writing portable code and try to abstract the low-level details of the target platform and parallel runtime. However, accounting for the overheads inherent to the target hardware architecture and low-level software runtimes is necessary for the accurate evaluation of the performance and efficiency of a system – and specially a parallel one.

In order to help the designer make early parallelization decisions, we propose and evaluate two methods and associated tools. The first consists in a parallel performance estimation and analysis tool, Parana, presented in Chapter 5. The second consists in a tool for determining the best tiling configuration for a given application kernel, Tilana, detailed in Chapter 6. In both cases, the application and multiprocessor platform timing parameters must be acquired in a characterization step. This characterization step is described in Chapter 4.

Chapter 3

Background and Related Work

Abstract

The work developed in this thesis covers a number of fields, from embedded multiprocessor platform architectures for computer vision applications to parallel software programming and associated design aid tools. This chapter reviews the main concepts and prominent recent work on such fields, which are relevant to the work presented in this thesis.

Contents

3.1 Introduction

This thesis' main goal, as discussed in Chapter 2, is to propose new methods and tools that aid the development of efficient vision algorithms on embedded multiprocessors. As such the work in this thesis is multidisciplinary, covering a number of fields from computer vision, embedded multiprocessor architecture and programming, as well as parallel design aid tools. This chapter presents the main background concepts in these domains and recent related work.

The first part in section 3.2 aims at providing an overview of important concepts in embedded computer vision and existing software libraries. The second part in section 3.3 describes the background in embedded multiprocessor architectures, as well as the current state of the art multiprocessor platforms and dedicated vision processors. The third part in section 3.4 focuses on embedded software parallelization concepts, existing programming models and benchmarks. Finally, the fourth and final part in section 3.5 covers the methods and tools for parallel application performance analysis and optimization. These four sections together provide a global picture of the current state of the art in efficient implementation of computer vision applications on embedded multiprocessors.

3.2 Embedded Computer Vision

Computer vision applications have long been constrained to run on powerful machines such as mainframes, servers and workstations. Over the last decade, we have observed a dissemination of miniaturized CMOS cameras, as well as a dramatic increase in the computational power of embedded systems. These two factors combined with more efficient algorithms have made it possible for computer vision applications to cross the barrier into embedded systems. Embedded computer vision is thus the fusion of two technologies: computer vision and embedded systems. An alternative definition is given by the *Embedded Vision Alliance* (EVA) [18] which considers embedded vision as "the practical use of computer vision in machines that understand their environment through visual means" [19]. This section presents a review of some key embedded computer vision concepts important when working in the embedded domain.

3.2.1 Concepts and Characteristics

3.2.1.1 Vision Processing Levels

A vision processing system can be represented as a data processing pipeline composed of a series of algorithms. Such algorithms are grouped in functional stages or levels. Algorithms in each level share some characteristics that make it more suitable to implementation on particular machine architectures. Dipert et al. identify three main levels in [63], namely:

1. image acquisition and optimization;
2. converting pixels into objects;
3. analysis of, and reasoning about objects.

The classification provided by Chitnis et al. in [48] splits the third level in two, with object recognition in one category and decision making or control as a separate category. They further establish a rough mapping between the most prominent characteristics of each vision processing level and the types of machine architectures which are most suitable. Figure 3.1 depicts these four main vision processing functional levels, as presented by Chitnis et al. in [48] and described in details in the sequence.

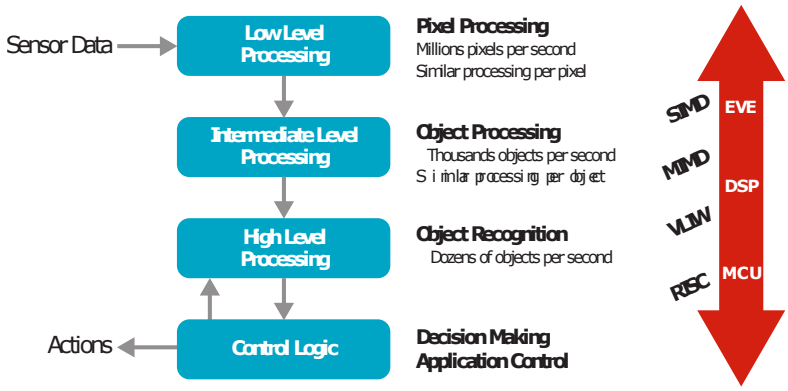


Figure 3.1: *The four main vision processing functional stages or levels as defined by Chitnis et al. in [48].*

Low-Level or Pixel Processing corresponds to the functional level that improves the quality of or otherwise transforms the input image into an image that is suitable for further object processing levels. Examples of algorithms performed at this level are noise filtering and contrast enhancement to improve quality, cropping or scaling to adapt the image dimensions, background suppression to discard unwanted information. Other types of filters, such as gradient filters for producing edge images, might also be necessary to transform the image for subsequent levels. Algorithms in this class are point or local algorithms, simpler and massively parallel, meaning that they can be very easily parallelized over a large number of processors.

Intermediate Level or Object Processing is the functional level that is responsible for passing from a purely image based representation to an object representation. Its goal is to pinpoint potential regions of interest for further high level processing. Thus, it consists of segmentation algorithms such as blob or bounding box detection, as well as feature extraction algorithms. Such algorithms still need to treat a vast amount of data, but are more complex than those of the pixel processing stage, as they present more control flow branches. They are therefore less suited to *Single Instruction Multiple Data* (SIMD) machines, being more adapted to *Multiple Instruction Multiple Data* (MIMD) architectures which suffer less from control flow changes.

High Level Processing or Object Recognition consists in classifying an object as a member of a class – i.e. a human face in face detection –, or as a particular instance of an object – i.e. a specific person in face recognition. These algorithms have a control flow which is still more complex than those of the intermediate level, and while the cost for processing an object might be higher, the number of objects to process should be smaller. Thus compared to algorithms in the previous stage, these algorithms should be more adapted to platforms with higher single-core performance, but possibly with a lower core count.

Application Control or Decision Making Level forms the last vision processing level. As its name implies, this level relies on information acquired at previous steps to make high level decisions and perform additional actions. The algorithms at this level are essentially control algorithms that are more suited to coarse grained task level parallelism or sequential execution on more standard, general purpose architectures.

These four levels or categories summarize the main stages of a computer vision pipeline. Each level presents common characteristics that make it more suited to particular machine architectures, as discussed. While homogeneous systems are simpler to implement and validate, heterogeneous architectures might provide a range of processor architectures adapted to each level of the vision processing pipeline for improved efficiency. Furthermore, in distributed vision systems the execution of such levels so that higher level functions are done in a central processor, while lower level functions are offloaded to accelerators close to the camera. Thus, the system architect needs to determine the partition that best fits the requirements of the target application.

3.2.1.2 Data Dependency

Algorithms can be further classified into two groups, static or dynamic, according to the impact of input data on its control flow and execution time.

Static or Non-Data-Dependent algorithms are those whose control flow and execution time are not dependent on the contents of the images being processed. The variations in the execution time for each image location is thus typically much smaller than for data-dependent algorithms. Common examples of non-data-dependent algorithms are spatial filters – Gaussian, binomial and sobel filters among others –, as well as SVM classifiers.

Dynamic or Data-Dependent algorithms present a variable execution time from one image, or image location, to the next. This variability is a result of changes in control flow that are dependent on the contents of the image being processed. The parallelization of data-dependent algorithms is to be handle with care, as unlike non-data-dependent algorithms, they are likely to suffer degradation due to branch divergence and load imbalance. Higher level computer vision functions with a complex control flow are typically data-dependent. Common examples of data-dependent algorithms are morphological skeletons in image processing [136], cascaded classifiers such as the Viola and Jones' face detector [210], decision tree classifiers [123] and corner detectors like FAST [175].

Such a data dependency classification depends on the particular implementation of an algorithm. Two implementations of a same algorithm might react differently to input image contents, and thus belong to different categories. For example, an originally static algorithm might have a fast implementation which skips over unpromising solutions or unnecessary computations, thus becoming data-dependent. Conversely, an originally data-dependent algorithm might be implemented in a context which prefers real-time predictability over performance. An hypothetical hardware implementation of such algorithm with a fixed latency and throughput would thus be classified as static or non-data-dependent.

3.2.1.3 Data Locality

Computer vision and image processing algorithms can be further categorized according to their data locality, which in this context refers to the footprint of input data necessary to compute a single output pixel. This subsection defines three data locality categories – point, local and global – and discusses their characteristics.

Point algorithms are those where the computation of any given output pixel depends solely on the value of a single pixel in the input image. So, its main characteristic is that there is no dependency on other regions of the input image. Examples of such algorithms are color space transformations, color filters, or brightness offsets. They typically belong to the lower pixel processing level and are often massively parallel algorithms, as the amount of parallelism is high.

Local algorithms rely on an input image region or window around a given point to compute its output value. The size of the region will depend on the algorithm and its parameters. Examples of such types of algorithms are spatial kernel filters, convolutions and most image scaling transformations. They can belong to any of the vision functional processing levels. It is usual for implementations of such algorithms on embedded multiprocessors to keep the input local window in the shared memory.

Global algorithms have a dependency on the entirety of the image being processed. Examples of such algorithms are global contrast enhancement, color balance, histograms. They are typically harder to implement in a embedded platform, since an entire image frame might not fit in an embedded multiprocessor's shared memory. Implementing global algorithms for large images

in an embedded system requires either multiple passes, which will have a negative impact on performance and memory bandwidth, or to find a way to decompose the algorithm so that it can be implemented as a pipeline containing a series of local or point algorithmic steps.

3.2.1.4 Scale and rotation invariance

Computer vision applications often require the system to be capable of detecting and recognizing objects at different orientations and at a full range of distances from the camera. This section discusses two of the most common methods to make the algorithms scale and/or rotation invariant, and the impact of these techniques on the system design.

Rotation invariance, supposing the algorithm is not inherently rotation invariant, can be achieved by two means. A first technique consists in finding the principal orientation of the region of interest. Then, either the algorithm can be adapted to take into consideration this principal orientation, or the image can be rotated so that the principal orientation aligns with a predetermined axis, executing the algorithm, and then eventually rotating the results back to the original orientation. A second technique consists in generating multiple views of the input image at predetermined rotation angles and executing the algorithm separately on each view. Unless the algorithm can handle the rotation information internally, in both cases the image needs to be rotated prior to execution. Although it results in a robustness to changes in rotation, these techniques imply a high computational cost as the system needs to perform some or all of the following steps:

1. Determine the principal orientation of the input image or window;
2. Produce one or more rotated views of the image;
3. Execute the algorithm on each rotated view of the image.

Scale invariance can be achieved by executing a same algorithm over multiple image scales and then assembling the results accordingly. This particular technique is called pyramidal image processing, as the images at successive scales form a pyramid-like shape when stacked. Two pyramidal image processing approaches exist. The first consists in scaling the input image at each pyramid level and running an unmodified detector on said level. The second consists in keeping only a single full resolution image and scaling the detector itself instead, thus emulating a virtual pyramid. While the second technique is often implemented in more powerful systems, it requires multiple accesses to the entire image, which might not fit an embedded system's local memory.

For embedded systems the first approach of scaling the input image can be more efficient in terms of memory and bandwidth utilization. Moreover, the required image downscaling can be efficiently implemented with a dedicated hardware scaler. From a parallelization point of view, however, processing multiple images at different resolutions might prove challenging, as larger iterations have more parallelism that can be exploited by platforms with a higher core-count, while smaller iterations can suffer significant degradation due to increased parallelization overheads and load imbalance, as well as underutilize the available platform resources.

3.2.1.5 Sliding Window

Several of the computer vision algorithms in the experiments performed in this thesis rely on this sliding window technique. These algorithms are local algorithms as per the definition given in Section 3.2.1.3. The scanning window technique consists in applying a local algorithm sequentially over several image locations using a predefined scan order.

Such sliding window technique has been used for implementing image convolution with a kernel for many decades. White proposed a programmable digital architecture for a $3x3$ kernel

convolution with an image in 1981 [212]. This technique has since then made the leap into object detection and localization algorithms, such as in notable works from Viola et al. for face detection [211] and pedestrian detection [210] and Schneiderman et al. for detection of faces and cars [181]. In these works an object detector is successively applied at windows scanned over the entire image with the goal of localizing an object in the image. This technique remains useful with some recent works by Sudowe et al. [190] and Lampert et al. [125] focusing on optimizing the search strategy so as to restrain the search locations and therefore reduce the number of times the detector is evaluated.

When parallelizing an algorithm with a scanning window technique, the local windows typically represent a natural parallelization granularity. In this case, the parallelization workload for each processor will be a set of windows over an image that will be processed concurrently. While the scanning window technique can be applied over the entire image in one pass, it is often the case that the image does not fit into the system's local shared memory. In this case, the image must be partitioned into smaller sub-images and the scanning window technique is applied on each sub-image.

3.2.1.6 Tiling and Data Partitioning

The partitioning of application data and the transfer of such data across an embedded system's memory hierarchy is critical for the final application performance. Although the computational capacity of current embedded systems has grown significantly, such systems still have very limited internal memory capacity when compared to general purpose systems. Furthermore, some embedded platforms have no data cache and require explicit DMA calls to transfer data across the memory hierarchy. Due to the memory and communication constraints, embedded applications need to be designed to make efficient use of the available memory space and hide the communication time to the most. In this context, data parallel loops can be tiled to adapt the parallelization granularity according to characteristics of the target platform. This section discusses recent work in data partitioning, tiling and scheduling of associated data transfers.

According to the definition provided by F. Irigoin in [99] "tiling is a program transformation used to improve the spatial and/or temporal memory locality of a loop nest by changing its iteration order, and/or to reduce its synchronization or communication overhead by controlling the granularity of its parallel execution." Effectively, tiling consists in subdividing a given multidimensional loop iteration space into smaller sets of iterations that are executed in a grouped manner.

F. Irigoin et al. first developed the concept of *supernode partitioning* in [100]. It consists in partitioning loop iterations into *supernodes*, sets of loop iterations that are scheduled atomically on a given processor. M. Wolfe et al. [216, 217] describe more general *iteration space tiling* techniques, also capable of handling imperfectly nested loops.

Feautrier et al. proposed space-time mapping methods for the automatic parallelization of loop nests based on the polytope model in [69]. The proposed method was able to perform scheduling and allocation of iterations to processors via linear algebra and linear programming. Darte et al. proved in [58] that it is possible to build such a schedule with an asymptotically optimal execution time for cases with uniform dependencies. To handle irregular dependencies, Griebl, Feautrier et al. proposed the *index set splitting* method in [79] which partitions the problem space into parts, so as to build individual schedules for each part, much like a piecewise placement function. However, none of these methods accounted for communication times.

Recent work on automatic tiling support in compilers aims optimize application performance while at the same time easing the burden on the programmer. Bondhugula et al. [40] propose the use of polyhedral compilation techniques based on the polytope model to automati-

cally perform loop tiling, which were implemented in the PLuTO framework [41]. In a recent work, Bondhugula et al. design a framework named PolyMage [153] that relies on polyhedral compilation of a domain specific language for optimizing entire image processing pipelines. Another such polyhedral compiler is the R-Stream compiler [184, 14], which targets stream computing applications and uses the polyhedral model for a number of tasks, including [184, 206]: identifying and extracting parallel tasks; loop transformation and locality optimization; SIMD and DMA code generation; data layout transformation and communication optimization.

Polyhedral compilers perform affine loop transformations that increase data locality and memory throughput. However, as shown by Darte et al. in [58], they are only asymptotically optimal, and therefore suitable only for modeling systems with high iteration/tile counts. While this is a plausible hypothesis for *High-Performance Computing* (HPC) applications, this is less so for real-time embedded vision systems that often work with lower resolution images. Furthermore, polyhedral compilation cannot integrate non-linear performance models. In [152], Mullapudi et al. discuss the limitations of current polyhedral compilers, such as PLuTo. The authors conclude that the current validity conditions for tiling are too conservative and miss desirable tiling opportunities.

Andonov. et al [27] take a different approach and formulate the problem of finding optimal tiling dimensions as a constraint optimization problem. A set of analytical expressions is developed to model the execution time of a system for a given set of input parameters. A constraint solver is then capable to determine the best solution for the problem, one that minimizes the execution time. Unlike linear programming solvers typically used in polyhedral compilers however, a constraint optimization solver can handle non-linear expressions or constraints. In [177], Saïdi et al. apply this approach for constructing the tiling execution time model for array processing algorithms in systems with explicitly managed memory and define the analogous constraint optimization problem. They particularly target the IBM Cell architecture [90, 89] and obtain timing parameters by characterizing the target platform. In their work, however, they only model a platform with distributed DMAs. Furthermore, their method assumes tile dimensions are integer divisors of the image dimensions, a constraint we believe is too limiting and that more optimal solutions might be found if this restriction was lifted. Lifting this restriction, however, requires careful modeling of the remaining tiles at the last column and on the last row of the tiling space.

The use of tiling in heterogeneous GPGPU and embedded accelerator systems have also been explored in recent work. In [23], Alias, Darte et al. present a method for optimizing data accesses for offloaded kernels in an *Field Programmable Gate Array* (FPGA) accelerator platform. They automatically define the sets of tile data to be read in a remote accelerator system, as well as a source-to-source code generator that produces C-code that *High-Level Synthesis* (HLS) tools can synthesize to FPGA. Grosser et al. propose in [80] an automatic parallelization tool for *General-Purpose computing on Graphics Processing Units* (GPGPU) systems that generates CUDA code implementing index set split tiling. Larabi et al. [126] and Mancini et al. [135] have proposed 2D cache memory systems for efficient data access management in image processing applications. Although data partitioning and tiling have been studied for a few decades, active research is still being conducted in this field for reducing the burden on the programmer and automatically optimizing data partitioning and communication in newer system architectures.

3.2.2 Computer Vision Libraries

Computer vision libraries facilitate the integration of computer vision functionality to a software program. This section briefly reviews the main existing computer vision libraries.

OpenCV [42] is an open-source computer vision library written in C/C++. It is among the largest and most renowned computer vision libraries with a community of over 47,000 users [42].

It supports a range of systems such as servers, desktops and more recently embedded computing platforms such as Apple's iOS and Google's Android. Semiconductor firms such as NVidia and Intel provide implementations of OpenCV algorithms optimized for their machine architectures, with the use of advanced features such as SIMD vectorization via built-in assembly instructions. The C *Application Programming Interface* (API) has been deprecated and is now only a wrapper to the C++ core library. This limits its usage to embedded architectures which count with an efficient C++ compiler.

Khrono's OpenVX [113, 117] is an open and standard computer vision acceleration API that targets low-power and embedded applications. Its goal is to provide a standard framework for computer vision application development across different machine architectures. Optimized implementation of OpenVX's functions are provided by hardware vendors. OpenVX can be used directly as a standalone library, or indirectly to accelerate other libraries such as OpenCV. It relies on a graph-based specification and execution model, which allows vendor implementations to analyze and optimize computations and data transfers on each platform. Furthermore, an user tiling extension to the standard [116] is provided which allows users to specify tiling properties for the computation nodes. Rainey et al. [172] have demonstrated the system-level optimization of OpenVX graphs, while Tagliani et al. [193, 194] focus on optimization of memory bandwidth with tiling.

FastCV is a mobile-optimized closed source computer vision library developed by Qualcomm. The library is provided alongside the FastCV Computer Vision SDK [167]. The company claims the library supports any ARM-based architecture, but that it is optimized for their Snapdragon line of application processors. Some of the functionalities provided by this library are: gesture recognition; face detection, tracking and recognition; text recognition and tracking; and augmented reality.

VLFeat [207] is an open-source computer vision library written in C. Its main focus is on image understanding and the extraction of local features. As it is written in C, this library is more suited to embedded system implementation.

Halcon [154] is a proprietary vision library developed by MVTec Software and targeted primarily at industrial, surveillance and security applications. The company provides an interactive graphical *Integrated Development Environment* (IDE), named HDevelop, in which the user can describe the application graphically and the IDE generates the C, C++, C# or VB.NET code. This library supports several desktop and embedded platforms, as well as industrial cameras.

Matrox Imaging Library (MIL) [139] is a proprietary vision library developed by Matrox Imaging. It is targeted at industrial machine vision and medical imaging software.

Cognex Vision Library (CVL) [50] is an industrial machine vision library developed by Cognex, and written in C++. It includes algorithms for image alignment, measurement, inspection and identification. This library supports only Microsoft's Windows XP and Windows 7 operating systems and Intel processors, and is therefore not suitable for embedded systems.

Vision-Something-Library (VXL) [15] is not a computer vision library in itself, but a collection of C++ libraries where the X letter in the acronym can be substituted to refer to one of VXL's libraries. It is a research library developed by a group of international researchers and hosted in Sourceforge. Some of the included libraries are VNL for numerical applications, VIL for imaging applications, and VGL for geometrical calculations, among others.

Although this list is non-exhaustive, it is possible to identify a trend for modern computer vision libraries to be written in C++, and except for Qualcomm's FastCV and portions of the OpenCV library, they are not particularly adapted for embedded multiprocessor systems. More particularly, the multiprocessor architectures targeted in this thesis are the STHORM and the STxP70 *Application-Specific Multiprocessor* (ASMP), which are STxP70-based multiprocessors. As the STxP70's C++ compiler produces code that runs on average 20% slower than the STxP70

C compiler, the latter is preferred. Therefore neither FastCV, which officially supports only ARM-based platforms, and OpenCV, which is C++ based, can be supported by the STHORM and STxP70 ASMP platforms. There is thus a lack of a C-based computer vision library for efficient implementation of embedded computer vision applications in these two platforms.

3.3 Multiprocessor Architectures for Vision

Multiprocessor architectures are increasingly adopted in embedded systems. Using a higher number of small cores that run at a lower frequency is more energy-efficient, in terms of CPU operations per watt, than using a single processor core running at a higher frequency. Vision processing, specially the lower levels, often present high intrinsic parallelism that can be exploited in multiprocessor architectures. This section reviews existing multiprocessor architectures for vision. It first provides the definitions of architectural characteristics that can be used to relate and classify multiprocessor architectures. It then discusses the specificities of embedded multiprocessors targeted for the computer vision domain. Existing embedded vision multiprocessor architectures are then presented and analyzed.

3.3.1 Definitions

According to the definition of Culler et al. [55], a generic *multiprocessor* is a collection of computers – CPU and memory – communicating over an interconnect network. More particularly, the term *embedded multiprocessor*, as defined by Wolf in [214] and as used in the context of this thesis, refers to a system with multiple *Processing Element* (PE) in a single chip, otherwise known as *Chip Multiprocessor* (CMP) or *Multiprocessor System-On-Chip* (MPSoC).

3.3.1.1 Taxonomy

Flynn's taxonomy [70] categorizes multiprocessors according to their capacity to handle multiple instruction and data streams. This yields the following categories:

Single Instruction Single Data (SISD) represents sequential processors that can execute operations on a single data.

Multiple Instruction Single Data (MISD) represents hypothetical processors that would be able to execute different instructions on a same data point. Although possible, this is largely considered impractical.

Single Instruction Multiple Data (SIMD) represents processors that can execute an operation on multiple data concurrently, such as in vector or array processors, and implies that the data operations occur in lockstep.

Multiple Instruction Multiple Data (MIMD) represents multiple processors that can execute different operations or programs on multiple data concurrently, such as in a modern PC's multi-core processor.

Note that some architectures might present hybrid characteristics. Modern MIMD multiprocessor cores often possess vector instruction set extensions with SIMD characteristics, such as ARM's Neon or Intel's AVX instruction sets. The aforementioned taxonomy has since been extended to better represent some modern architectures. New classification includes:

Single Program Multiple Data (SPMD) is a particular case of MIMD where a single program performs operations on multiple data concurrently, but in which the operations are not done in lockstep as in SIMD architectures.

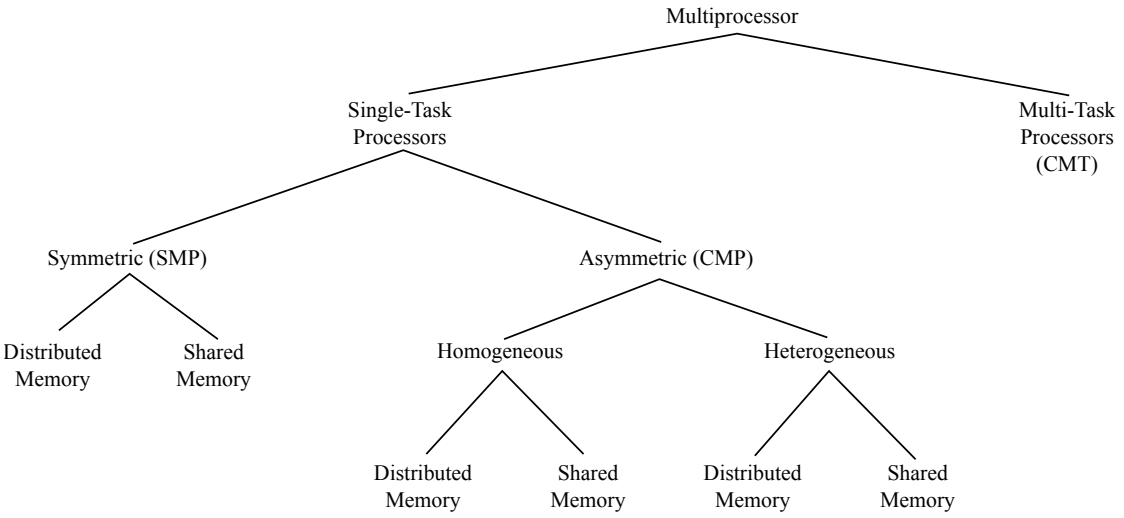


Figure 3.2: *The taxonomy of multiprocessors as proposed by Ventroux and David in [208]*

Multiple Program Multiple Data (MPMD) is the particular case of MIMD where instructions on different cores belong to different programs.

Single Instruction Multiple Thread (SIMT) represents SIMD architectures where the parallel operations are not explicit vector or array operations, but result from the concurrent execution of multiple standard "sequential" threads concurrently in lockstep. This paradigm has been introduced in recent GPU architectures from NVidia [132].

Beyond these classification criteria, multiprocessor architectures can also be grouped according to the homogeneity or symmetry of processing elements, or according to their memory hierarchy topology. In [208], Ventroux and David classify the multiprocessor architectures as depicted in Figure 3.2. Their classification relies on the following criteria:

Homogeneous or Symmetric Multiprocessors are those composed of PE that are multiple instances of a same processing core.

Heterogeneous or Asymmetric Multiprocessors are composed of PEs with different processor architectures.

Shared Memory (SHMEM) *Multiprocessors* possess a memory zone directly accessible by all of its PEs. Note that even though seen as a single memory address space, it may be implemented as separate physical memory banks. This leads to a further subdivision:

- *Uniform Memory Access* (UMA), where all processors observe the same access time accross the entire memory address space.
- *Non-Uniform Memory Access* (NUMA), where access time might vary according to the memory and network topology.

Distributed Memory Multiprocessors contain physically separate local memories for each PEs. Access to data in these memories by other PEs is done via communication primitives.

3.3.2 Vision Processors

The rise of embedded vision has driven the development of a number of application-specific multiprocessor architectures. This section reviews some recent embedded multiprocessor architectures dedicated to vision processing. Processor architectures are grouped according to the type of specialization used to accelerate vision processing.

3.3.2.1 Processors with Specialized Instruction Sets

Instruction set extensions allow system designers to customize a processor by adding new instructions. Such processors are known as *Application-Specific Instruction Set Processors* (ASIPs). Designers are then given the option of designing and implementing new instructions to accelerate a particular application – or ensemble of applications in a particular domain. Cadence/Tensilica's IVP [195] is an example of a *Digital Signal Processor* (DSP) architecture with an instruction set specialized for vision processing. It has a 4-way *Very Long Instruction Word* (VLIW) with a 32-way vector SIMD architecture and has several image processing specific operations to accelerate operations on 8-, 16- and 32-bit pixel data types and video operation patterns. It supports further customization with user-defined instructions.

3.3.2.2 Processors with Coupled Hardware Accelerators

A common option to accelerate vision processing is by designing an *Accelerated Processing Unit* (APU) that couples a multiprocessor to dedicated hardware accelerators for vision. Analog Devices' Pipeline Vision Processor (PVP) [26] is a stream processing engine attached to a DMA channel that processes the data as it flows through the channel. Eutecus's Multi-Core Video Analytics Engine (MVE™) [66] relies on biologically inspired, massively parallel hardware acceleration units. Apical Spirit [204] and Assertive Vision Engines [17] implement programmable hardware engines for real-time detection, classification and tracking of people and objects. Synopsys' DesignWare EV5x [192] architecture couples a quad-core Synopsys ARC CPU to up to eight hardware object detection engines.

3.3.2.3 Processors with VLIW/SIMD Accelerators

The usage of VLIW/SIMD accelerators is also a common way to accelerate low-level pixel processing layers in APUs. Such architectures can perform multiple operations on data in a single cycle. Although there is only a single instruction stream, the long instruction word encodes the operations to be performed by all units. The effort of scheduling the operations on the several execution units is thus performed offline by the compiler. Some examples of such architectures that use VLIW/SIMD accelerators are: CEVA's XM4 [45, 46], CogniVue's CV220x Image Cognition Processor [51], MobileEye's EyeQ4 [49], Toshiba's MPEs [203] and Videantis v-MP4000HDX [209].

3.3.2.4 VLIW/SIMD Vector Processors

VLIW/SIMD units can further exist as standalone vector processors or IPs that can be integrated in an MPSoC. CogniVue's APEX [142, 34], Movidius' Streaming Hybrid Architecture Vector Engine (SHAVE) [146] and Texas Instruments' AccelerationPac *Embedded Vision Engines* (EVEs) [131] are examples of vector processing IPs. The Freescale's SCP2200 Image Cognition Processor [73] is a vector processor IC that includes one CogniVue's APEX core and a number of peripherals.

3.3.2.5 Vision MPSoCs

A number of complex MPSoCs dedicated to vision processing have been developed in the last years. *Advanced Driver Assistance Systems* (ADAS) applications seem to be the driving force behind these architectures. Such architectures can have multiple subsystems that cover several vision processing layers. Texas Instruments TDA3x [197, 198] architecture, for instance,

counts with two ARM Cortex A-15 processors for control level processing; four ARM Cortex M4 processors for high level processing; two C66x DSPs for signal processing; and a Vision AccelerationPac with four EVEs for pixel level processing. Examples of other complex MPSoC architectures for vision are Analog Devices' BF609 [26], Freescale's S32V230 Platform [72, 202], Inuitive's NU3000 [98], MobileEye's EyeQ4 [49], Movidius' Myriad 2 Vision Processor [150, 146], and Toshiba's TMPV760 series [203]. For a a list of the aforementioned vision processors and a description of their main architectural features refer to Appendix D.

3.3.2.6 Discussion

The rise of embedded computer vision applications has pushed companies to develop a number of vision processing architectures. Algorithms in different vision processing layers present characteristics that make them suitable to particular machine architectures. Low level pixel processing is best suited to hardware accelerators or wide VLIW/SIMD units. Intermediate and high level processing is typically implemented in MIMD multiprocessors with small in-order processor cores. Finally, higher level control functions can be harder to parallelize and need greater flexibility and are typically implemented in larger out-of-order processors with fewer cores.

The architectural trade-offs will ultimately depend on the characteristics of the target application and its requirements. On application-specific multi- and many-core processors, such as STMicroelectronics's STHORM and STxP70 ASMP, the system designers can customize a number of architectural parameters. They should be able to analyze the application and quickly evaluate different implementation trade-offs.

3.4 Embedded Software Parallelization

Efficient embedded software parallelization requires careful analysis and design of the application. As the focus of the work is on analyzing and estimating the parallel performance of an application so as to select efficient parallel implementations, this section reviews some important concepts in parallel programming and performance optimization. It further reviews existing parallel programming models and the microbenchmarks that can be used to characterize a multiprocessor and its parallel programming models.

Section 3.4.1 discusses the classes of parallelism at varying abstraction levels. Section 3.4.2 describes the theoretical limits of the parallel speedup of data parallel applications, as they are the main focus of this thesis. Section 3.4.3 lists the types of performance bottlenecks in a parallel application. Finally, Section 3.4.4 presents the OpenMP and OpenCL parallel programming models used in this thesis and reviews other existing parallel programming models.

3.4.1 Parallelism Classes

Parallelism can be expressed at different abstraction levels. In [214], Wolf lists the three main classes of parallelism:

Instruction Level Parallelism (ILP) consists in executing a number of instructions in parallel on distinct processor lanes. A dual-issue VLIW processor such as the STxP70, for example, can execute two instructions in parallel and has a maximum theoretical *Instructions per Cycle* (IPC) of 2.

Data Level Parallelism (DLP) consists in dividing a given data set across a number of *Processing Elements* (PEs) that process each data subset in parallel. All of the PEs execute the same operation on the data.

Task Level Parallelism (TLP) consists in executing different tasks or operations in parallel on a same input data set. Each PE thus executes distinct tasks or operations on the data.

Pipeline Level Parallelism (PLP) is an additional form of parallelism which consists in executing dependent tasks or operations in parallel. Tasks are linked so that the output of a task is the input of the following task. Data is thus read by the first pipeline task and transferred from one task to the next until the final stage produces the output data.

Data and task parallelism are arguably the most common parallelism types exploited by software programmers. Parallel programming languages such as OpenMP and OpenCL, as argued by Thies in [201], focus on task and data parallelism, but do not provide specific mechanisms for software pipeline parallelism which needs to be explicitly coded by the programmer. Instruction parallelism is very dependent on the machine architecture and on the compiler to maximize the IPC count.

3.4.2 Maximum Parallel Speedup

Amdahl's Law [25] models the performance gains that can be achieved by parallelizing an application on a given number of processors. The execution time model he proposed assumes the application contains: a serial portion s, whose execution time is constant regardless of the number of parallel PEs; and a parallel portion p, whose execution time is inversely proportional to the number of PEs. This yields the following expression for the parallel execution time $t(n)$ on n PEs for an application with a sequential portion of s:

$$p{=}1-s \tag{3.1}$$

$$t(n){=}t(1)\cdot\left(s+\frac{p}{n}\right) \tag{3.2}$$

The speedup of the parallel execution time over the sequential execution time can be thus computed as:

$$\mathcal{S}(n)=\frac{t(1)}{t(n)}=\frac{1}{s+\frac{p}{n}} \tag{3.3}$$

As $t(n)$ represents an ideal parallel execution time, $\mathcal{S}(n)$ represents the maximum parallel speedup for the application for a given number of processors. Furthermore, as the number of PEs in the system grows, the speedup will be eventually limited by the sequential portion of the application:

$$\lim_{n\to+\infty}\mathcal{S}(n)=\frac{1}{s} \tag{3.4}$$

The parallel speedup from Amdahl's law is applicable to homogeneous architectures and is valid only if the amount of computation is constant. Several extensions of the Amdahl's law have been proposed, as discussed by Al-babtain et al. in [21]. A notable extension is that of Gustafson in [81], which became known as Gustafson's law. He argues that as the number of PEs in a parallel system increases, the system is capable to treat proportionally larger data sets. Assuming the problem size scales with the number of PEs, the speedup would continually increase and not be submitted to the bounds from Eq. (3.4). Its applicability is however limited, since not all applications can benefit from increasingly large data sets. Another notable extension is that of Marty and Hill which have described in [85] an extension to Amdahl's law for modeling asymmetric and dynamic multiprocessors.

3.4.3 Performance Bottlenecks

Performance bottlenecks represent the factors that limit an application's performance. The application's performance is thus said to be *bound* by the bottleneck. Determining the bottleneck of the system allows a designer to modify the application or add resources to increase performance. Performance optimization flows constantly seek to determine the bottleneck of the system and eliminate it if possible. Performance bounds can be categorized under four categories, namely:

CPU Bound means the speed at which an application advances is limited by the speed of the CPU. A task that executes many operations on a small set of data is likely to be CPU bound.

I/O Bound means that an application's speed is limited by the latency and/or bandwidth of the I/O subsystem. A task that counts the number of lines in a given file is likely to be I/O bound.

Memory bound means that an application's speed is limited by the memory's latency and bandwidth, as well as its capacity. A task that performs few operations on a large set of data is likely to be memory bound.

Cache bound means the an application's execution time is limited by the cache memory subsystem. Tasks that iteratively process a data set larger than can fit in the cache is likely to be cache bound.

A designer must be able to determine the bottleneck of a particular parallel application on a multiprocessor system. This information serves as a guide to make architectural or parallelization design trade-offs so as to improve performance.

3.4.4 Parallel Programming Models

3.4.4.1 OpenMP

OpenMP [12] is a parallel programming model targeted at homogeneous shared memory systems. It specifies a series of compiler directives, library functions and environment variables which can be used to express parallelism in Fortran and C/C++ programs. OpenMP handles parallelism by dispatching the execution of parallel code sections on multiple parallel threads, usually one thread per processor core. It is based on a fork-join model of computation. The sequential code sections are executed on a master thread. Upon entering a parallel section the OpenMP runtime forks execution on worker threads. When leaving a parallel section a synchronization barrier ensures all threads are done and execution continues on the master thread.

Parallelism is explicit, meaning the developer inserts compiler directives and library calls directly to the application code to control the parallelization. An OpenMP compiler interprets the OpenMP directives in the application code and applies the necessary transformations and runtime calls.

This thesis focuses on OpenMP 2.5 [159], which is the version the STxP70 ASMP supports. This version supports both task and data parallelism. A parallel region is created using the `omp parallel` directive. Task parallelism is supported via the definition of independent sections in a parallel region with the `omp sections` directive. Data parallelism is supported via the `omp for` work sharing directive, which schedules `for` loop iterations on separate worker threads. OpenMP 3.0 [161] adds support for dynamic task creation, while OpenMP 4.0 [160] adds support for heterogeneous systems by scheduling threads for execution on GPUs or other programmable accelerators.

Listing 3.1: *Example of an OpenMP application from [160] that implements a parallel sum.*

```
#include <stdio.h>
  int main (void) {
  int a, i;

  //Declare a parallel region
  #pragma omp parallel shared(a) private(i)
  {
    //Initialize accumulator variable 'a' on the master thread only
    #pragma omp master
    a = 0;

    //Synchronizes all threads
    #pragma omp barrier

    //Declare a data parallel for loop
    #pragma omp for schedule(static, 4) num_threads(8) reduction(+:a)
    for (i = 0; i < 256; i++) {
      a += i;
    }

    //Any single thread prints the result (implicit barrier)
    #pragma omp single
    printf("Sum is %d\n", a);
  }
}
```

OpenMP directives apply to the immediately following code section, either a line of code or a code block delimited by curly brackets. The `omp parallel` directive defines a parallel region that encompasses the next code section. The number of threads can be specified by appending the `num_threads(`*n*`)` option to the directive. Otherwise, the default value will be used, which can be modified by calling the `omp_set_num_threads(`*n*`)` or by setting the environment variable `OMP_NUM_THREADS`. This value represents only an upper bound as the effective number of threads will depend on how many threads are available upon entering the parallel region. All threads that participate in a parallel region execute the entire code section, unless a work sharing directive is used inside the parallel region.

Task Parallelism in OpenMP relies on the definition of independent parallel sections inside a parallel region. This is done by attaching the `omp sections` directive to an outer code block, and marking each independent inner code block with a `omp section` directive. The compiler creates individual microtasks for each such section that are scheduled on different thread at runtime.

Data Parallelism is specified by attaching a `omp for` directive to for loops inside a parallel region. The compiler extracts the loop body into a microtask and replaces it with a call to said microtask. Microtasks will be scheduled on all worker threads, which will execute a subset of the loop iterations, called *chunks*. The number of iterations in a chunk will depend on the particular scheduling chosen, or can be forced by the user. The schedule is determined by appending the `schedule(type [, chunk size])` option to the directive, where `type` is the scheduling type and `chunk size` is an optional integer value. The five OpenMP loop schedules are:

- `Static`: Allocates chunks statically to each worker thread in a cyclic way. The default chunk size is the number of loop iterations divided by the number of threads in the parallel region rounded up to the nearest integer.

- `Dynamic`: Allocates chunks dynamically to each worker thread as they become available. The runtime needs some form of synchronization or atomic operations to determine the next chunk to process. The actual implementation is platform dependent. The default chunk size is one.
- `Guided`: The guided schedule behaves like a static schedule where the chunk size is variable and depends on the amount of remaining iterations to process. It strikes a compromise by starting with larger chunks that show less scheduling overheads, and gradually decreasing the chunk size to improve load balance.
- `Auto`: The compiler or runtime is free to determine the best schedule to use – `static`, `dynamic` or `guided`.
- `Runtime`: The schedule is determined according to the value of the `OMP_SCHEDULE` environment variable at runtime.

Data Access. By default, OpenMP considers that variables declared outside the scope of a parallel region are shared among all threads. Variables declared in the parallel region are private to the thread. Access modifiers can be appended to a parallel region definition to specify if particular variables are shared or private, and to define if their values should be updated upon entering or leaving a parallel region. OpenMP also defines reduction operations that can be applied on private thread variables upon termination of a parallel construct.

Synchronization. The main OpenMP synchronization directives are:

- `omp barrier`: Threads wait until all other threads reach that particular point in the program before continuing.
- `omp master`: The code section to which it is applied is executed only by the master thread.
- `omp single`: The code section to which it is applied is executed by the a single thread, the first to reach it.
- `omp critical`: Any code sections to which it is applied are protected and only one thread can execute any of the critical sections at a time. A name can be given to the critical section, in which case only one thread can execute any of the homonym critical sections. Parallel execution of critical sections with different names is allowed.
- `omp atomic`: An update to a variable is guaranteed to be executed atomically.

Note that the implementation of these directives depends on a particular OpenMP implementation and the support offered by the underlying machine architecture. Listing 3.1 shows an example application from [160] that illustrates the use of several OpenMP directives to implement a parallel sum.

3.4.4.2 OpenCL

OpenCL [112] is a parallel programming model that targets heterogeneous systems. It was originally developed for *General-Purpose computing on Graphics Processing Units* (GPGPU) applications, but it is increasingly being used as a standard model for programming embedded multi- and many-core processors as well. OpenCL applications are launched on a host processor and can offload computation kernels on compute devices. The OpenCL version supported on STMicroelectronics's STHORM platform is the version 1.1 [114] of the standard which is described next.

Compute devices can be GPUs, multi- or many-core processors. OpenCL assumes a compute device has one or more compute units, each of which is composed of a number of *Processing Elements* (PEs). Clusters in many-core or GPU architectures will thus be seen as independent compute units.

An OpenCL kernel is analogous to a C function, which takes in a number of parameters, processes them and returns a result. Each kernel is written in a separate file using a subset of

the C language. A kernel's workload must be partitioned among work-groups and work-items. Unlike OpenMP however, OpenCL does not provide any directives that automate the workload sharing among work-groups and work-items. The OpenCL runtime schedules the execution of the work-groups and work-items on the compute units and their processing elements and invokes the kernel, but it is the programmer's responsibility to explicitly distribute the workload. Two types of IDs are provided by the OpenCL runtime, the local ID which refers to the ID of the work-item in its work-group, and the global ID is a unique work-item ID across all work-groups. The programmer must then explicitly call runtime functions to determine the kernel's global and/or local ID and use it to select different subsets of the data to process. A common strategy to perform fine-grained parallelization of a `for` loop in a kernel, for example, is to use the work-item's global ID as the initial loop index offset and the total number of work-items as the loop stride. Several work-group level synchronization mechanisms are available, such as barriers, locks and atomic operations, but global synchronization is discouraged.

Four distinct data address spaces exist that, according to the underlying machine architecture, can be mapped onto different memory hierarchy levels:

- `global`: a read/write zone accessible by all work-items in a compute device.
- `constant`: a read-only zone accessible by all work-items in a compute device.
- `local`: a read/write zone accessible by work-items in a same work-group.
- `private`: a read/write zone accessible private to a single work-item.

The STHORM OpenCL runtime allocates global data buffers on the L3 host memory, while constant data are placed on the L2 memory. Local and private data are placed in the L1, the cluster's shared memory.

Data transfers among the different memory zones can be programmed via specific OpenCL data copy functions. The OpenCL API specifies asynchronous work-group level copy functions to transfer data between such buffers, where a single DMA transfer is launched for the entire work-group. STHORM's OpenCL implementation extends the API with a work-item copy function, allowing individual work-items to launch DMA transfers autonomously.

3.4.4.3 Other Programming Models

APEX Core Framework (ACF) *[202]* is a development environment for parallel vision applications on CogniVue's APEX cores. It uses a graph-based formalism to represent applications as a processing graph. The development tools compile the graph into a native APU program and are responsible for parallelizing the application on the APEX cores.

Array Oriented Language (Array-OL) [60] is a graphical formalism for specifying multi-dimensional signal processing applications. It uses a two-level approach:

1. A global level specifies the structure of the application via a graph whose nodes represent computations that exchange multidimensional array data.
2. A local level that details the computation performed by each node on the data.

Intel Cilk Plus [92, 174] specifies a number of C/C++ language extensions for specifying task and data parallelism on multiprocessors. Similarly to OpenMP, it is the compiler that modifies the generated code to introduce the necessary runtime calls for parallel execution. However, whereas OpenMP uses compiler pragmas, Cilk relies on new language keywords. Three main keywords control the parallelization [174]:

1. `cilk_spawn`: Causes the following statement, such as a function call, to be executed on a new thread for task parallelism.
2. `cilk_for`: Loop iterations are executed on parallel threads for data parallelism.
3. `cilk_join`: Waits for parallel thread completion.

Halide [170, 168] is a functional language that allows specifying parallel image processing pipelines. It focuses on the separation of the application's functionality and parallel scheduling [169]. The authors claim that decoupling the functionality from the parallel specification allows a more concise description of the application, as well as the customization of the schedules to better suit different machine architectures. From both specifications, the Halide compiler applies a series of transformations, such as loop fusion and vectorization, and generates the application executable.

Heterogeneous System Architecture (HSA) *[10]* establishes a series of standards that define a common architecture [87] and parallel programming runtime [86, 88] for heterogeneous systems. HSA uses a single coherent virtual memory addressing space among the components of the platform. This enables seamless passage of data pointers across devices without the need for explicit data transfers or mapping. HSA uses an architecture-independent virtual *Instruction Set Architecture* (ISA), the *HSA Intermediate Layer* (HSAIL), that is compiled *Just in Time* (JiT) to the target machine architecture. It is designed to serve as a common framework that supports the execution of parallel applications written in other languages, such as OpenMP and OpenCL.

Message Passing Interface (MPI) *[11]* is a communication protocol for parallel programs on distributed memory systems. It provides message passing communication primitives that allow program instances running machines to exchange data and collaborate. It is often used in conjunction with OpenMP in a hybrid model where MPI is used for coarse level parallelism in a distributed system and OpenMP is used for thread level parallelism.

Pthreads [13] is a language-independent parallel thread execution model. The Pthreads standard [13] defines an API for thread management and synchronization, directly supported by POSIX-conformant operating systems or implemented as a software library. It also serves as an underlying threading mechanism in other parallel runtime implementations.

Intel's Thread Building Blocks (TBB) *[94]* is a parallel programming C++ library [53] based on C++ templates. As a library, it does not require any particular compiler support. TBB provides functions [96] – such as `parallel_for`, `parallel_while` and `parallel_reduce` among others – that allow the definition of DLP, TLP and PLP parallelism in C++ applications. A number of design patterns [95] covering common parallelization scenarios are supported. TBB's scheduler implements work stealing for better load balancing.

3.5 Parallel Performance Analysis and Estimation

Embedded system designers have numerous possibilities in terms of architectural and parallelization design choices. Analyzing and evaluating the impact of these choices early in the design flow is a challenge. This section reviews recent methods and tools have been proposed to aid designers in this task.

3.5.1 Parallelism Profiling

Profiling allows to gain insight into the runtime performance and the bottlenecks of an application. Performance optimization flows typically rely on application profiling to detect hotspots that provide the most opportunity for performance improvements and eventual bottlenecks that must be eliminated. Besides representing a mean for developers to gain insight into the performance of an application, profiling can be an intermediate step for parallelization analysis methods and tools described in the following sections.

Profiling tools need runtime information about the application that can be directly captured by the execution platform such as a simulator, or that can be obtained via some form of instrumentation. Saeed et al. [176] identify three major components of a profiler:

1. data collection,
2. data analysis, and
3. presentation.

3.5.1.1 Profiling Analysis

Parallelism profilers can collect data and provide a number of metrics to the developer. This section focuses on three types of profilers that are targeted at parallel application performance analysis, as well as loop and data dependency profilers.

Parallel Performance Profiling tools perform dynamic or post-mortem analysis of an application run to acquire performance metrics. The *OpenMP Profiler* (ompP) [75] is an OpenMP profiling tool that provides a detailed analysis of a parallel program performance. It can determine the effective parallel computation time, synchronization and idle times, as well as the time lost due to load imbalance and insufficient parallelism. It relies on an automatic instrumentation of the OpenMP source code via the Opari [145] source-to-source translation tool. Cilkview [82] and Cilkprof [180] are performance profiling tools for Cilk Plus parallel applications. Intel VTune Amplifier [93] is a performance profiler and analysis tool for Intel *Central Processing Units* (CPUs). The Score-P [122] tool is a performance measurement runtime that can be used as a common data acquisition infra-structure for a number of parallel performance analysis tools, including Vampir [121], Scalasca [76], TAU [185] and Periscope [77]. As with the ompP, the Score-P runtime is linked with the application itself and produces the traces dynamically at runtime. These tools however rely on PAPI [151] hardware counters that are not supported on the STHORM and the STxP70 ASMP architectures targeted in this thesis.

Loop Profiling tools focuses on loop analysis as a means to identify an application's hot-spots and potential parallelization candidates. The LoopProf [148] is a loop profiler proposed by Moseley et al. that relies on *Dynamic Binary Instrumentation* (DBI) via Intel's Pin [133] tool for collection of runtime execution data.

Data Dependency Profiling consists in analyzing an application's instruction and/or memory traces to extract data dependencies. A number of frameworks for discovery of potential parallelism such as Kremlin [107], Prospector [118] and Cilkprof [180], rely on data dependency analysis to determine the critical path of an application and predict its scalability or potential parallel speedup (see Section 3.5.2). Kremlin and Prospector rely on source-to-source transformations for source code instrumentation, while Cilkprof instruments the code in an *Low Level Virtual Machine* (LLVM) compilation pass. Another usage for data dependency analysis is to pinpoint the sources of potential parallelization bottlenecks, as is the case in the Intel Advisor XE [97] tool.

None of the aforementioned tools for parallel profiling support the STHORM or the STxP70 ASMP architectures targeted in this thesis. Furthermore, while profiling support exists for the single-core STxP70, no call-tree or parallel profiling support existed for the STxP70 ASMP as of the writing of this thesis. Therefore there was a need for adding performance profiling functionality to the Parana tool, which is based on acquired Gepop simulation traces as discussed in Section 4.6.1.

3.5.1.2 Instrumentation and Data Acquisition

As seen, profiling tools need to acquire and process execution trace data. To acquire trace data, most tools instrument the application so that it automatically collects the necessary data upon execution. Instrumentation can be performed at various levels:

1. Source level instrumentation.
2. Compiler inserted instrumentation.
3. *Dynamic Binary Instrumentation* (DBI).

All of these mechanisms result in integrating the data acquisition and trace file generation in the application binary. As discussed in Section 4.5.2 this direct trace file generation method can slow the execution time of *Instruction Set Simulators* (ISSs) in more than one order of magnitude for fine-grained instrumentation. This is due to the fact that the simulator must interpret the trace collection infra-structure and associated file I/O calls included in the simulated application code. If the simulator supports trace generation, a more efficient solution is to have the simulator write the traces directly on the simulation host, as discussed in Section 4.5.3.

3.5.2 Parallelism Discovery and Bounds

Several approaches have been proposed for the discovery of potential or total parallelism in sequential applications based on some form of data dependency analysis. Analysis can be static, such as performed by parallelizing compilers at compilation time, or dynamic, acquired by execution time data dependency profiling.

Critical Path Analysis (CPA) relies on dynamic data dependency analysis to determine the critical path of an application. From the critical path, CPA tools are able to estimate the application's potential parallelism using a *work-span* [52] analysis. Early works on CPA include those of Kumar [124] and Austin [32]. In a *work-span* analysis, *work* represents the duration of the sequential execution $t(1)$ of a program, while the *span* is the duration of the critical path. The critical path represents a lower bound on the execution time of the application with infinite resources $t(\infty)$. Therefore, the maximum potential speedup of an application would be bound to:

$$\lim_{n \to +\infty} \mathcal{S}(n) \leq \frac{\text{work}}{\text{span}} = \frac{t(1)}{t(\infty)} \tag{3.5}$$

However, the maximum speedup results using CPA are excessively optimistic [104]. More recently, Jeon, Garcia et al. have proposed in [107] an *Hierarchical Critical Path Analysis* (HCPA) method for increased accuracy. Their Kremlin [107] parallelism profiling tool generates a list of the parallelization candidate regions and their maximum speedup. Jeon extended the work with the Parkour [106] tool, which integrates resource constraints to predict the maximum speedup at application level. Another extension was proposed by Jeon et al. in the Kismet [105] tool that introduces the notion of *expressible parallelism*, which consists in identifying different forms of parallelism and taking into account only the parallelism that can be expressed in the target platform. Cilkprof [180] is a parallelism profiler tool for Cilk Plus applications proposed by Schardl et al. that applies *work-span* analysis using critical path to determine maximum speedup bounds.

Dependency Analysis is used in other notable tools for discovering the parallelization potential of an application. Larus et al.'s pp tool [127] estimates the parallelism in application loop nests. Zhang et al.'s Alchemist [220] and Kim et al.'s Prospector [118] are tools that rely on data dependency analysis for parallelism discovery and for suggesting parallelization candidates.

3.5.3 Parallel Speedup Prediction

Parallel speedup estimates can be used to evaluate and compare parallelization opportunities. Speedup prediction tools allow possible performance bottlenecks and the expected parallelization gains to be assessed early in the development flow. Some existing tools focus on analyzing the scalability of already parallelized applications on systems with higher core count to uncover so-called *scalability bugs*[1] [44]. Simulation can also be used to predict the performance of the final parallel system, but it requires a fully-functional parallel version of the application. The focus of the parallel performance prediction work in this thesis is to predict and analyze the parallel performance of serial applications under different parallelization scenarios. Related work on parallel performance modeling and prediction are reviewed next.

Emulation and Simulation. Physical prototypes and cycle-accurate simulators can provide very accurate performance estimates. However, prototypes take time to setup and have constrained resources, while cycle-accurate simulations are time-consuming. Fast simulation engines typically trade some timing accuracy for faster simulation times. *Dynamic Binary Translation* (DBT) simulators, such as QEMU [35], dynamically translate the instructions from the target architecture binary into instructions of the simulation host for fast execution. ISS simulators, such as STMicroelectronics's STxP70 *cycle-approximate* simulator, interpret the target binary instructions and model a number of characteristics of the target processor such as processor pipelines and memory hierarchies to provide accurate timing estimates. More recent source-level simulators annotate the code with low-level timing information that can be used by fast simulators to provide more accurate timing estimation. Stattelmann et al. [188] proposed a fast source-level simulation technique for estimating the timing of complex MPSoC applications on a SystemC *Transaction Level Modeling* (TLM) simulator. Sampling simulators [47] mix two simulation models, a functional untimed model and a timed model. The timed model is slower and only used at certain timing sampling intervals to statistically derive the execution time of the application. All of these approaches, however, require complete, fully-functional parallel versions of the application for each design point to explore. This translates in a larger effort from the designer to modify and validate the application for each design point to explore and thus results in a longer exploration time.

Analytical Performance Models estimate the application execution time algebraically. Early works on analytical performance modeling of parallel applications include the work of Amdahl and Gustafson, as previously discussed in Section 3.4.2. Extensions to their work for modeling more recent multiprocessor and heterogeneous architectures have been proposed by Hill and Marty [85] and Sun et al. [191]. The *work-span* analysis and *Critical Path Analysis* (CPA) can also be used to estimate an application's parallel performance, as discussed in Section 3.5.2. HCPA tools such as Parkour [106], Kismet [105] and Cilkprof [180] are able to predict the parallel performance of a sequential application. While fast, these tools tend to produce overly optimistic results.

Empirical Performance Models estimate application performance with a *black-box* approach. They rely on a set of acquired performance data to build a performance model. Joseph et al. use a regression model for processor performance estimation in [162]. A power consumption estimate using a regression model is proposed by Lee et al. in [130]. Lee et al. also use inference and machine learning to derive parallel performance models in [128] and propose in [129] a composable performance regression technique for fast performance prediction towards design space exploration of multiprocessor architectures. Such models, however, require significant amounts of input data in order to derive the performance model. Furthermore, such models can only account for characteristics which have been exercised in the input data set.

Mechanistic Performance Models are *white-box* models that rely on knowledge on the inner workings of the multiprocessor platform and its parallel runtime. This knowledge is thus

[1] A *scalability bug* is a part of the program whose scaling behavior is unintentionally poor [44]

explicitly integrated into the model allowing it to account for effects such as scheduling policies and other platform-dependent behaviors that can be difficult to model with a simple analytical or regression model. The Intel Advisor XE [97] tool defines a methodology for parallelism discovery and evaluation that relies on first profiling the code to discover the application's *hot-spots* and then perform a so-called *suitability* analysis. This suitability analysis relies on dynamic runtime data obtained from user-inserted instrumentation of parallel regions and loops to estimate the speedup of the application on a number of x86 architectures. Parallel Prophet [119] is also a mechanistic parallel performance prediction tool for sequential applications that is part of the Prospector [118] framework. It relies on user instrumentation to collect application traces, build a task graph of the application and estimate its parallel speedup. It models memory contention statistically by introducing a *burden factor* that penalizes the speedup according to the density of memory accesses. These tools are focused on desktop and server class machine architectures with different characteristics from STMicroelectronics's STHORM and STxP70 ASMP embedded platforms.

Multiprocessor Platform Characterization is necessary to measure a series of platform-dependent parameters such as memory access and DMA data transfer times and the OpenMP runtime overheads. A common way to acquire such data is via microbenchmarks. Unlike benchmarks that focus on measuring and comparing parallel application performance across different architectures [38, 37, 186, 218, 30, 164, 108], microbenchmarks typically exercise specific functionalities and can be used to measure the overheads of individual parallel constructs. Both Kismet and Parallel Prophet rely on the *Edinburgh Parallel Computing Center* (EPCC) OpenMP microbenchmarks [43] and extensions [62, 65]. The EPCC OpenMP microbenchmarks have C implementations supported on STMicroelectronics's STxP70 ASMP with minor modifications and were thus used as the basis for the characterization work in Chapter 4. While the EPCC microbenchmark suite exercises a number of OpenMP scheduling and synchronization directives, it is generic and lacks microbenchmarks for measuring memory access and DMA data transfer timings. The EPCC microbenchmark suite has thus been extended in this thesis with two new microbenchmarks, *membench* and *dmabench*.

3.5.4 Limitations of Current Approaches

This section reviewed a number of existing tools that support a developer in the discovery and analysis of an application's parallelization potential on a desktop or server architecture. Existing tools typically make use of a combination of application source code or binary instrumentation and hardware counters that is intrusive and not supported in embedded multiprocessor architectures, particularly the STMicroelectronics's multiprocessors targeted in this thesis. When capturing traces on a *cycle-approximate* simulator the instrumentation intrusiveness translates in a significant raise in simulation time for fine-grained tracing of functions and loops. The overhead induced by the instrumentation might further interfere with the scheduling in the microbenchmarks when characterizing the OpenMP runtime. Another point is that these tools do not offer enough support for quickly defining and comparing the parallel performance of different application parallelization scenarios and platform parameters. There is thus a lack of tools that support efficient instrumentation, profiling and parallel performance analysis of embedded applications early in the design flow of an application-specific multiprocessor.

3.6 Conclusion

This chapter reviewed the background and the state of the art in embedded vision processing and dedicated processor architectures for vision. Additionally, it reviewed relevant related work in embedded software parallelization and early performance prediction. The analysis of the

work presented in this chapter shows a lack in methods and tools for early parallelization analysis and parallel performance prediction adapted for embedded multiprocessors. Another important aspect of the implementation of vision algorithms on resource constrained embedded architectures is tiling. Current tiling parameter selection methods lack the ability to accurately model a number of non-linear DMA data transfer characteristics and parallelization overheads necessary for increased accuracy.

This thesis proposes new methods and tools that facilitate the development of an embedded application-specific multiprocessor system. The methods proposed in Chapters 4 and 5 allow the fast and accurate evaluation and analysis of different trade-offs in terms of the application parallelization strategies and of the architectural parameters. Furthermore, tiling is an important aspect of the implementation of vision algorithms on resource constrained embedded architectures. Chapter 6 proposes an analytical model and an optimization framework for analyzing and selecting the best tiling parameters for a particular application kernel.

Chapter 4

Application and Multiprocessor Platform Characterization

Abstract

The parallel application performance analysis and tiling analysis methods proposed in this thesis need prior characterization data of *(i)* the target multiprocessor platform and its parallel runtime, and *(ii)* of the user application. This chapter presents the characterization flow designed to automatically acquire such characterization data. The first part of the chapter describes the application source code instrumentation library, two task trace acquisition methods, followed by a brief discussion on their implementation trade-offs. It then presents the Trace Filter and Parana tools used to build a call tree profile of an application. The second part details the microbenchmarks used to characterize the STxP70 ASMP platform and its OpenMP parallel runtime. The developed characterization library and tools are able to automatically acquire all the necessary platform characterization data, with low intrusiveness and low simulation time overheads. The results obtained serve as the base for the parallel performance estimation and tiling optimization work in Chapters 5 and 6.

Contents

4.1 Introduction

When developing an application-specific multiprocessor system, the system designers must *(i)* size the multiprocessor platform and *(ii)* optimize the application on such platform. The designers needs to configure the number of cores and additional platform parameters to ensure the application has enough computing resources available to fulfill its requirements. Concurrently, they must optimize the application so as to ensure the available resources are used efficiently. Addressing these two aspects requires in-depth knowledge of the characteristics of both platform and applications.

This thesis proposes two tools – Parana and Tilana– to aid designers to perform early application parallelization performance analysis and tiling analysis, presented in Chapters 5 and 6, respectively. However, these tools require characterization data input from both the platform and the application to perform their analyses.

A common method for acquiring platform metrics is via microbenchmarks. Differently from benchmarks, which gauge the platform performance over complex application scenarios, microbenchmarks are composed of small code sections that exercise single features at a time. Several microbenchmarks exist for standard parallel programming languages and multiprocessor systems. When characterising a new system which is not supported, however, the microbenchmarks must be ported to the new target architecture. Furthermore, microbenchmarks for parallel runtimes model only the features covered by this parallel runtime specifications or APIs. In order to characterise specific platform features, such as its memory hierarchy or DMA data transfer times, microbenchmarks need to be extended with custom tests.

One of the goals of the work described in this chapter is to characterize the STxP70 *Application-Specific Multiprocessor* (ASMP) platform and its OpenMP runtime. Although no particular microbenchmarks existed for this platform, it supports OpenMP applications written in C. Therefore, a C-based open-source microbenchmark suite was selected, which is easily extensible: the *Edinburgh Parallel Computing Center* (EPCC) OpenMP microbenchmark suite. This suite was ported to the STxP70 ASMP platform and extended with two new microbenchmarks to characterize its memory hierarchy latencies and DMA transfer times.

While the reports produced by the EPCC OpenMP microbenchmarks provide useful insights to the designer and allows to compare the performance of different OpenMP implementations, they do not provide the level of detail necessary to build an accurate performance model of the platform and its parallel runtime. To overcome this, low-level task traces of the microbenchmarks were collected and used to generate a complete call tree of the application. Two methods were developed to acquire these traces, *(i)* one which relies on an instrumentation library to generate the task traces directly upon execution, and *(ii)* the other which parses STxP70 instruction traces to produce said task traces. For this second method, a Trace Filter tool was developed to parse instruction traces and produce said task traces. While the first method allows trace acquisition on the *Field Programmable Gate Array* (FPGA), where no instruction traces are available, the second method is faster, less intrusive and allows to monitor memory accesses and function calls.

Parana reads the collected task traces and uses them to build an application task graph and call tree profile. Moreover, to automate the extraction of platform characterization parameters, a characterization mode was added to Parana in which it monitors OpenMP runtime calls to automatically extract finer-grained parallel runtime parameter values.

Characterization of an user application and of the platform share the same characterization framework. Both of them use the aforementioned process to collect task traces of either an user application or microbenchmarks. The platform characterization process then performs an additional step in which the microbenchmark task traces are analyzed to automatically acquire the platform characterization data. An additional instrumentation library allows the user to

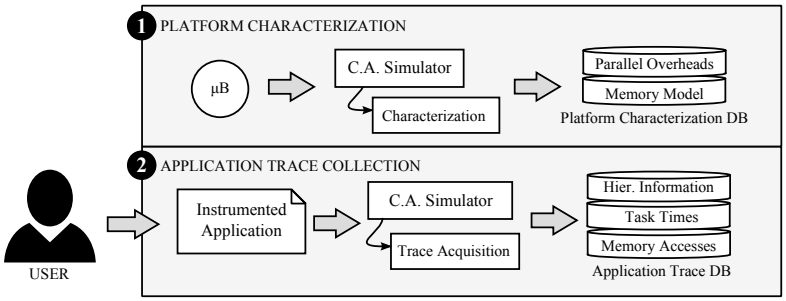


Figure 4.1: *Overview of the characterization flow conceived to generate the platform and application characterization databases. In the platform characterization phase, a set of microbenchmarks is executed on a cycle-approximate simulator, their task traces acquired and processed by a characterization tool that automatically populates the platform characterization database. In the application characterization phase, a similar process is used to simulate an user-instrumented application and collect its task traces.*

control task trace acquisition and define explicit tasks at the desired granularity level directly via pragmas in the C source code.

This chapter thus presents the flow used to characterize *(i)* the STxP70 ASMP platform and its associated OpenMP parallel runtime, and *(ii)* an user application. The outputs of this characterization flow are the characterization databases that will serve as an input for Parana and Tilana in Chapters 5 and 6.

The remaining of the chapter is organized as follows. Section 4.2 provides an overview of the characterization flow and the associated tools. Section 4.4 then presents the code instrumentation *Application Programming Interface* (API), while Section 4.5 details the task trace formats, the trace preprocessing stages, and discusses their implementation trade-offs. Section 4.6 shows how such task traces are used to derive the application call tree and profiling information. Section 4.7 presents how the aforementioned source code instrumentation API and the trace acquisition infra-structure are used to collect OpenMP microbenchmark traces in order to characterize the multiprocessor platform. Finally, Section 4.8 summarizes the main points presented and concludes this chapter.

4.2 Overview of the Characterization Flow

The proposed characterization flow automatically generates the platform and application characterization databases. Figure 4.1 depicts this flow, and its two phases. The platform characterization phase characterizes the multiprocessor platform and its parallel runtime by means of dedicated microbenchmarks used to acquire task traces, which are post-processed in order to generate a platform characterization database. The latter is independent of the user application and only needs to be regenerated if either the platform or the parallel runtime are updated. The application characterization phase characterizes an user-instrumented application and populates the application trace database. In either phase, an executable binary is first simulated with a reference cycle-approximate simulator to collect execution traces, which are then post-processed differently by each phase.

Characterization is performed by running an executable binary compiled for the target platform on a simulator of said platform in order to produce a task trace of the application execution. A task trace can be generated either directly upon execution by the instrumentation library or by post-processing the simulator instruction traces with the Trace Filter tool. This task trace can then be used to reconstitute a †of the applications' tasks, from which its call-structure and profiling information can be extracted as described in Section 4.3.

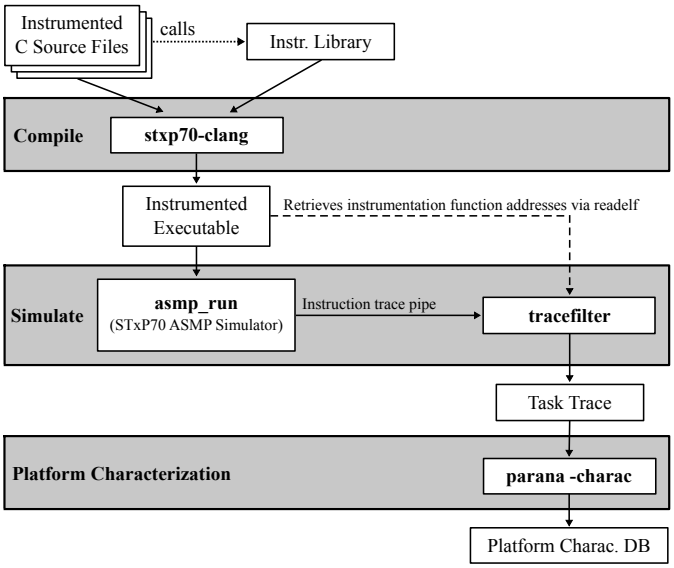


Figure 4.2: *Overview of the characterization tools*

The Trace Filter tool automatically inserts tasks for each function call when parsing the instruction traces, while the provided instrumentation library allows the user to define additional tasks via a set of instrumentation macros. User defined tasks are used to instrument particular code sections of interest to the user, such as loop iterations that constitute potential parallelization targets in subsequent steps. Another use for the instrumentation is to delimit and attribute a label to a particular code section, so that it appears as a separate entry in the call tree. This mechanism can be used to have better control over the characterization and even acquiring dynamic profiling information. It is useful, for instance, to discern the execution time of a unique code section executed with different sets of parameters in the profile results, simply by assigning unique labels for each parameter set. Furthermore, these task traces are compact and represent only a fraction of the size of the original instruction traces, while still retaining the necessary information for performance analysis. The result of using a compact trace format is an improved scalability of the method.

Figure 4.2 details the three phases involved in the characterization of an application and the associated tools. First the instrumented application code is compiled and linked with the instrumentation library to produce an STxP70 ASMP binary. Said binary is simulated with the Gepop cycle-approximate simulator. An application task trace is then generated either directly by the instrumentation library itself, or by the Trace Filter tool as depicted in Figure 4.2, which parses the simulator instruction traces and generates the task trace. This task trace constitutes the application characterization database.

In the case of the platform characterization, the process to generate task traces is the same as for the application characterization, but an additional step is necessary to extract the platform parameters from the task traces. The goal in this work is to characterize the STxP70 ASMP platform (§2.2.3) and its associated OpenMP [159] parallel runtime. To characterize this platform, the characterization flow is executed on a set of microbenchmarks designed to exercise a number of OpenMP constructs and platform functionalities. The characterization module of the Parana tool will build a call tree of the microbenchmarks, based on the acquired task traces, in which OpenMP runtime calls are readily visible. The overheads of the runtime calls are measured by the characterization framework directly from the task traces, which then collects first order statistics over all executions of said runtime calls in a microbenchmark and saves the results into a platform characterization database file.

Listing 4.1: *Reference source code of a vector addition function* `vAdd`*. It receives two arrays* `a` *and* `b` *and the number of elements* `n` *as input and writes the result to the output array* `r`*.*

```
1 void vAdd(int *r, int *a, int *b, int n) {
2     int i;
3     for(i=0; i < n; i++) {
4         r[i] = a[i] + b[i];
5     }
6 }
```

Listing 4.2: *Example of a profiling report for a reference* `vAdd` *function with no instrumentation (see Listing 4.1).*

```
1 Profiling of sequential schedule
2 --------------------------------
3                             #Calls    Cml.   Avg. Cml.    Self   Avg. Self
4 + -1 root
5   + 0 main                       1    5561        5371      46          46
6     + 1 vAdd                     1    5371        5371    5371        5371
```

4.3 Function Call Tree Profile

The characterization of an executable binary requires analyzing its runtime information and call tree structure. A call tree profiling report represents the hierarchical function calls as a tree for easier visualization. Each entry in a call tree profile typically contains a number of synthetic information regarding that particular function call. As no profiling support existed for the target platform, the STxP70 ASMP, particular profiling support was added to the Parana tool as described in Section 4.6.1. This section describes the call tree profile produced by Parana.

To illustrate the call tree report, Listing 4.1 shows an example of a simple vector addition function *vAdd*. Listings 4.2 then shows the profile call tree that would be generated by Parana for the `vAdd` function. All timing figures in the profile call trees are reported in terms of processor cycles. The call tree lists, for each task:

1. its id, function name and label;
2. the number of times it was called (#Calls);
3. its cumulative time (Cml.), which includes its self time and its children time recursively;
4. its self time (Self); and
5. the average cumulative time (Avg. Cml.) and average self time (Avg. Self) per call.

Listing B.1 in Appendix B provides an example of a real profile call tree on STxP70 ASMP for the *Fast Features for Accelerated Segment Test* (FAST) corner detector application. This application is presented in Chapter 5.

4.4 Source Code Instrumentation

An instrumentation library is provided to allow the user to *(i)* enable or disable trace acquisition and to *(ii)* define explicit labeled tasks at the desired granularity. This section describes the functionality provided by the instrumentation library via a set of macros through which the user can control the trace acquisition and define new tasks. Two implementations of the trace library have been developed, one in which the task traces are directly written by the instrumentation library as the macros are executed, the other by writing to flags in static

Instrumentation Macro	Description
PARANA_INIT()	Initializes trace files
PARANA_ENABLE()	Enables trace collection
PARANA_DISABLE()	Disables trace collection
PARANA_RELEASE()	Closes trace files
PARANA_TASK_INIT(*var*, *label*)	Initializes a labeled task
PARANA_TASK_START(*var*)	Start a labeled task
PARANA_TASK_END(*var*)	End a labeled task

Table 4.1: *List of available instrumentation macros.*

Listing 4.3: *Example of the* `vAdd` *function source code with instrumentation at the for loop iteration level.*

```
1  #include "parana.h"
2
3  void vAdd(int *r, int *a, int *b, int n) {
4      PARANA_TASK_INIT(_parallel0_, "parallel0");
5      PARANA_TASK_INIT(_for0_, "for0");
6      int i;
7      PARANA_TASK_START(_parallel0_);
8      for(i=0; i < n; i++) {
9          PARANA_TASK_START(_for0_);
10         r[i] = a[i] + b[i];
11         PARANA_TASK_END(_for0_);
12     }
13     PARANA_TASK_END(_parallel0_);
14 }
```

Listing 4.4: *Example of a profiling report for a reference* `vAdd` *function with instrumentation at the for loop iteration level (see Listing 4.3).*

```
1  Profiling of sequential schedule
2  --------------------------------
3                              #Calls    Cml.   Avg. Cml.    Self   Avg. Self
4  + -1 root
5   + 0 main                        1    5561        5561     190         190
6    + 1 vAdd                       1    5371        5371      46          46
7     + 2 vAdd.parallel0            1    5325        5325      25          25
8      + 3 vAdd.for0              100    5300          53    5300          53
```

memory positions that are decoded when post-processing the instruction traces. These two implementations will be covered in sections 4.5.2 and 4.5.3, respectively.

Table 4.1 lists the available macros in the instrumentation library. The PARANA_ENABLE and PARANA_DISABLE macros control trace collection and are used to bypass trace acquisition for code sections that are irrelevant for the performance analysis, such as debug code. Macros with the PARANA_TASK prefix are used to define labeled tasks and mark their start and end in order to delimit particular code sections.

To illustrate the utilization of the instrumentation library in the user code, Listing 4.3 shows how the `vAdd` function from Listing 4.1 can be instrumented with two labeled tasks: `parallel0` and `for0`. These tasks will appear in the application call tree profile and will be available to serve as parallelization targets for the Parana tool, presented in Chapter 5.

Task Field	Description
id	Identifier
t_{start}	Start time
t_{end}	End time
$func$	Function name
$label$	Label
$type$	Type
id_{parent}	Identifier of the parent task
$l1\_rd$	Number of L1 read accesses
$l1\_wr$	Number of L1 write accesses
$l2\_rd$	Number of L2 read accesses
$l2\_wr$	Number of L2 write accesses
$l3\_rd$	Number of L3 read accesses
$l3\_wr$	Number of L3 write accesses

Table 4.2: *List of traced task descriptor fields.*

Labeled tasks are regarded as regular function calls in that they define hierarchical sub-regions of the call tree in the acquired trace database and constitute typical parallelization targets. Listing 4.4 shows the profile call tree that would be generated for the execution of the instrumented `vAdd` function. Notice how the instrumentation directives allow to gather much finer profiling details regarding the loop iteration timings than the original profile from Listing 4.2.

4.5 Task Trace Acquisition

The task trace plays a central role in the characterization process, which lays the base for the subsequent work on parallel performance analysis. This section describes the task trace's format and how it is generated. Two implementations are presented, the first consisting in direct generation of the task trace file by the instrumentation library upon application execution, and the second relying on post-processing of simulator instruction traces to generate the task traces. These two techniques are described and their benefits and limitations are discussed. While the file based trace generation is more portable and flexible, the instruction trace based method is less intrusive and provides more detailed information. The latter can be used to rebuild the entire application call tree, an important aspect for characterization of both the application and the multiprocessor platform.

4.5.1 Task Trace Format

The task trace database is stored as a CSV file containing a set of task trace entries. A task trace entry is a snapshot of the fields listed in Table 4.2. It captures a task's name, timestamp, per-zone memory access counts, as well as hierarchical context data that will allow a detailed call tree graph of the application to be built. One task trace entry is generated at each user defined task start and end points, specified by the PARANA_TASK_START and PARANA_TASK_END macros, respectively. Additionally, task trace entries are generated by the Trace Filter tool when parsing the instruction traces for each function call and return from function instructions.

4.5.2 File Based Task Trace Generation

This section presents the first of two task trace generation approaches, the file based approach. In this approach, it is the instrumentation library itself that generates the task traces during the simulation. Upon initializing the instrumentation library, a task trace file is created on the simulation host. Then, upon execution of the PARANA_TASK_START and PARANA_-TASK_END macros, the target library writes a trace entry to the trace file at run time. The timing figures are obtained using the target platform's cycle counters, which are halted during the execution of the instrumentation library. An initialization loop determines the overhead in processor clock cycles for halting/resuming the cycle counter, and each time the library halts the cycle counter, its value is compensated, reducing the intrusiveness of this method. Finally, when the application releases the instrumentation library, it flushes and closes the trace file.

The main advantage of this method is that it supports the writing of the task trace files by means of the explicit instrumentation entries on physical or emulation platforms that do not support instruction trace file generation, such as CPUs, FPGAs or ASICs. On the STxP70 ASMP's FPGA prototype, calls to file I/O functions are intercepted by the platform's debugger which relays the accesses to the host PC file system.

The disadvantages of this method, however, are numerous and limit its practical utilization. First, as this method relies on library code executing on one of the simulated platform cores to write the task trace files directly, and halts the cycle counter in the process, if other cores in the system are active, it is hard to account for their timing. Therefore, synchronization of the cores before calling an instrumentation macro would be critical in this case. This makes it difficult to capture task traces for parallel code, specially with dynamic scheduling, since the iterations would continue to be executed by other threads while the master thread executes the library code. If the section to be measured is small, the relative instrumentation overhead could be so high that all of the loop iterations might be executed dynamically on other threads while the master thread executes the instrumentation library code. In the case where only a single thread is executed, such as for the application characterization, this might be a lesser issue, but specially for the characterization of the platform, which is done with a parallel execution, this constitutes a major impairment.

Furthermore, since the call tree cannot be saved with this method, apart from instrumenting the OpenMP library itself, it is impossible to expose the OpenMP runtime calls in the profile results. However, tracing the OpenMP runtime calls is necessary for the characterization of the multiprocessor platform and parallel runtime. Finally, the library function calls that write to the trace files can consume thousands of cycles for each trace entry, and can be very time-consuming, typically slowing simulation time by one order of magnitude.

This method has been used in the early stages of the project, with great care for synchronization, to generate high granularity profiling information for the microbenchmarks running on the FPGA, but cannot provide enough information for a full characterization of the OpenMP runtime.

4.5.3 Instruction Based Task Trace Generation

This section presents an alternative method for generating the task traces based on the online parsing of the processor instruction traces. This method consists in activating the generation of instruction traces for the master processor of the STxP70 ASMP simulator and extracting profiling information from the instruction logs and specifically inserted instrumentation. The instruction traces are either written to a regular file on the host for later processing by the Trace Filter tool, or can be redirected to a host Linux FIFO – also called a named pipe – to be processed concurrently with the simulation. The usage of a FIFO provides the added benefit that the possibly very large instruction trace file is not written to the disk, but only buffered in the

Instrumentation Event	Signaling Flag Value
TASK_START	0
TASK_END	1

Table 4.3: *List of the supported values for signaling between the instrumentation library and the Trace Filter tool. Signaling is performed by writing such values to the instrumentation flag's address in memory.*

simulation host memory and treated as it is generated. This has allowed to process instruction traces in excess of 30 GB for some of the experiments in the thesis with a FIFO of only 4 KB and produce a more manageable task trace with only tens of MB. Furthermore, as the instruction trace processing in the Trace Filter tool is typically faster than the simulation itself, and provided that there are available computing resources in the host machine to allow parallel execution of both the simulator and the Trace Filter tool, the execution time of the simulation is only marginally affected.

The Trace Filter tool works by parsing the instruction trace file, in order to produce a task trace file. The instrumentation library adds signaling information to the instruction traces, which is detected and interpreted by the Trace Filter. The signaling consists in performing write operations to flags which are statically assigned in a global static buffer in the shared memory. Each address in this buffer corresponds to a flag. Table 4.3 lists the supported signaling values that can be written to a flag.

When a PARANA_TASK_INIT macro is called, it first reserves an address in the static memory buffer for the flag and then writes its label name and address to a mapping file. This can be seen as a lazy initialization process and is done only once at the initialization of a flag producing a singleton instance of the flag. The PARANA_TASK_START macro, when executed, writes the TASK_START event value of the event types enumeration to the address of the flag. Analogously, the PARANA_TASK_END macro writes the TASK_END enumerated value to the flag address.

Upon starting, the Trace Filter tool gets the address of the global static event buffer from the application binary via the *readelf* utility. It then monitors the instruction trace for any store instructions to addresses inside the event buffer. When such a store operation is detected, the Trace Filter captures the stored value and interprets it as either a task start or end, adding the corresponding entry to the task trace file. The name of the trace entry is retrieved from the file written at the initialization, or, if not yet available, a generic name is generated and substituted by the correct name later in the process. The instrumentation instructions themselves are discarded and do not contribute to the cycles or memory access statistics collected.

The advantages of this method over direct writing of the task trace file by the instrumentation library are numerous. It is faster, much less intrusive and does not require any particular synchronization. It allows capturing the complete application call tree, including the OpenMP runtime calls, as well as tracking memory accesses, keeping memory access counters for each zone.

However, as the instruction traces are not available in the FPGA platform, this method cannot be used to acquire traces when running the application on FPGA. This limitation could be overcome by adding support for debug cells in the STxP70 ASMP, like a Nexus [138, 16] debug cell similar to the ARM CoreSight [143, 28] debug and trace solution. A Nexus cell is able to produce compact traces at the basic block level, at each control flow change. Extracting the PC addresses and timestamps in the Nexus traces, and matching it to a disassembled code of the binary application, would allow to generate the task traces. Although the Nexus debug cell is already supported as an optional feature of the STxP70 single-core processor, it is not currently supported on the STxP70 ASMP due to the high circuit area overheads that would ensue. The support of this trace acquisition mechanism on an FPGA platform is thus left as a future work.

4.6 Application Characterization and Profiling

The application characterization consists in defining a representative test case for the application, running this test case on the STxP70 ASMP simulator and collecting the task traces. Note that at this stage the application is still in its original sequential state. The trace acquisition methods described in the previous section are used to generate a task trace file for the simulator run of the desired test case. Once a task trace of the application is collected, it can be used to generate an application profile.

4.6.1 Profiling

The Parana tool parses the task traces to generate the application profile. It first builds an *Hierarchical Task Graph* (HTG) of the application in which edges represent parent-child relationships between tasks. It then computes the profiling statistics from the HTG graph and generates a profiling call tree. The Parana tool is discussed in further details in Chapter 5.

When given an input application task trace file, Parana reads the task trace entries, matching start and end entries and creating a Task object to store trace information. Task objects are added as nodes to an *Hierarchical Task Graph* (HTG). Parent-child relationships between tasks, determined via the hierarchical information contained in the task trace, are marked as edges on the hierarchical task graph.

In order to produce a profiling report of the application, Parana creates a new Profile Task Graph for storing ProfileTasks – a specialization of a base Task object that stores statistical summary information for a set of Tasks. The HTG is traversed in a depth-first manner, and each Task found is added to a corresponding ProfileTask in a Profile Task Graph. The ProfileTask object will keep track of information about the added tasks such as to be able to generate the summary statistics for all tasks which have been added to it. In the end of the process, the profile task graph will be populated with ProfileTasks, one for each set of original tasks with a same name, and at the same hierarchical level in the call tree. Finally, a reporting module scans the finalized Profile Task Graph to print a formatted call tree containing the profiling information into a report file. Listings 4.2 and 4.4 show two examples of call trees, with and without explicit instrumentation. For a description of the call tree format refer to Section 4.3.

4.6.2 *Video Analytics Library* (VAL)

The characterization process largely relies on instrumentation of an application's source code. The creation of a common framework that included the necessary libraries was thus envisaged so as to facilitate the development of embedded video analytics and computer vision applications.

A first requirement for such a library was that it should be entirely developed in C, for optimal performance on the STxP70 ASMP platform. Moreover, the usage of proprietary libraries from other vendors was disallowed due to licensing issues. Existing STMicroelectronics vision libraries focused on providing high-level hardware models and were not optimized for parallel systems such as the STxP70 ASMP. As none of the vision libraries reviewed in Section 3.2.2 were adapted to the above requirements, it was necessary to develop a new vision library. This new *Video Analytics Library* (VAL) is based on existing STMicroelectronics libraries, but optimized for the STxP70 ASMP.

VAL defines a common framework to all of the target applications developed and used in the thesis' experiments. It is a modular library in which each module provides a set of functions. Modules can then be combined to produce more complex applications. Automated build scripts

allow the modules to be built for the native host machine, or for either the STxP70 single-core processor or the STxP70 ASMP multiprocessor with or without OpenMP compilation enabled.

The library is built around a base vaCore module which is used by all other modules. This module provides base types such as images, lists, and other data structures, as well as optimized I/O and other auxiliary functions. Scripts automate the creation of new modules, making it easier to add new functionality. The isolation provided by modules allows developers to work concurrently on different modules.

For a seamless use of the instrumentation library described in this chapter, support for it has been included in the core module of the VAL library, and can be enabled or disabled in the compilation phase via a simple command-line switch. The applications contained in this framework and used for the experiments will be detailed as needed in subsequent chapters.

4.7 Multiprocessor Platform Characterization

The multiprocessor platform characterization relies on the same flow described in previous sections for user application characterization, but applying it to specially crafted microbenchmarks. The task traces of these microbenchmarks are then fed to the Parana tool characterization module, that automatically extracts platform characterization information from the task trace of the microbenchmarks. The Parana tool is discussed in further details in Chapter 5.

This section first describes the platform characterization parameters identified so as to accurately model the platform. Then, it presents the microbenchmarks developed to exercise the several aspects of the STxP70 ASMP platform and its associated OpenMP runtime. Finally, it provides and discusses the results of the platform characterization step.

4.7.1 Platform Characterization Parameters

Based on the analysis of the call tree profiling of the OpenMP runtime for several parallelization constructs, a set of parameters was selected to build a mechanistic analytical model that accounts for the main multiprocessor platform and OpenMP runtime costs. Table 4.4 lists the platform and OpenMP runtime parameters to be automatically extracted in the platform characterization step. Note that a set of OpenMP thread management (OP) and for loop scheduling (OF) parameters are gathered separately for OpenMP parallel regions and OpenMP parallel for loops under different scheduling policies – static and dynamic. The final characterization of the DMA data transfer costs relies on an interpolation model from the characterization data points. The description of how the parameters are extracted from the microbenchmarks task traces by the Parana characterization module, and their meaning is detailed in the sequence.

4.7.2 Microbenchmarks

A set of four OpenMP microbenchmarks has been used to exercise a number of platform functions and allow automatic extraction of the parameters listed in Table 4.2. These microbenchmarks are derived from the EPCC OpenMP microbenchmark framework [43], which has been ported onto the STxP70 ASMP platform and adapted to rely on the platform's cycle counters, so as to report measurements in terms of processor clock cycles. This EPCC OpenMP microbenchmark suite was selected because it is a free and open source microbenchmark, entirely written in ANSI C, the preferred language for programming the STxP70 ASMP. The framework provided

Benchmark	Parameter	Description
schedbench	OP_{par_open}	Time to open a parallel region
	OP_{par_close}	Time to close a parallel region
	OP_{thr_create}	Time to create a worker thread
	$OP_{thr\_create\_gap}$	Time gap between creation of two threads
	OP_{thr_launch}	Time to launch worker thread
	$OP_{thr\_launch\_gap}$	Time gap between launch of two threads
	$OP_{thr\_launch\_master}$	Time to launch master thread
	$OP_{thr\_prologue}$	Overhead at thread prologue
	$OP_{thr\_epilogue}$	Overhead at thread epilogue
	OP_{thr_free}	Time to release worker thread
	$OP_{thr\_free\_master}$	Time to release master thread
	$OF_{for\_prologue}$	Overhead at an OpenMP for prologue
	$OF_{for\_epilogue}$	Overhead at an OpenMP for epilogue
	OF_{for_gap}	Time gap between two OpenMP for iterations
	$OF_{for\_next\_chunk}$	Time to schedule next chunk in OpenMP for
syncbench	$OS_{critical\_ovh}$	Overhead for an OpenMP critical section
	$OS_{barrier\_ovh}$	Overhead for an OpenMP barrier
membench	OM_{l1_rd}	L1 memory read latency
	OM_{l1_wr}	L1 memory write latency
	OM_{l2_rd}	L2 memory read latency
	OM_{l2_wr}	L2 memory write latency
	OM_{l3_rd}	L3 memory read latency
	OM_{l3_wr}	L3 memory write latency

Table 4.4: *List of STxP70 ASMP platform and OpenMP runtime characterization parameters, categorized as: OpenMP thread management (OP), OpenMP for loop scheduling (OF), OpenMP Synchronization (OS) and Memory (OM).*

has been found to be very easy to use, to port to the STxP70 ASMP and to extend with new benchmarks. The four microbenchmarks used in the proposed characterization process are:

1. ***schedbench***. This microbenchmark is part of the original EPCC suite. It exercises a number of OpenMP scheduling directives such as parallel section creation, for loop scheduling under several scheduling policies and with increasing chunk sizes.

2. ***syncbench***. This microbenchmark is also part of the original EPCC suite and contains a number of tests for measuring the impact of several of the synchronization primitives. This microbenchmark allows to measure the overheads of the critical section and the barrier synchronization primitives.

3. ***membench***. This microbenchmark consists in a new extension of the EPCC OpenMP microbenchmark suite designed to characterize memory access latencies across different hierarchical memory levels of the STxP70 ASMP.

4. ***dmabench***. The EPCC OpenMP microbenchmark suite was further extended with this microbenchmark to characterize DMA data transfer times across different hierarchical memory levels of the STxP70 ASMP.

The characterization of the OpenMP parallel runtime using the *schedbench* and *syncbench* microbenchmarks is described in subsection 4.7.3. The characterization of the latencies of read and write accesses for each memory level of the STxP70 ASMP using the *membench* microbenchmark is described in subsection 4.7.4. Finally, the characterization of DMA data transfer costs using the *dmabench* is described in subsection 4.7.5.

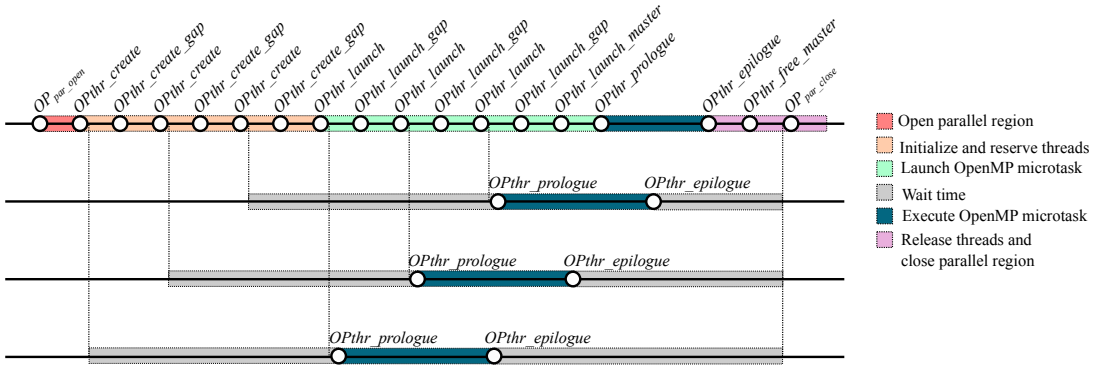


Figure 4.3: *Timeline graph showing the main phases and steps in an OpenMP parallel region execution. The dots show the point in time where the homonymous runtime characterization parameters listed in Table 4.4 appear in a parallel region execution.*

4.7.3 Parallel Runtime Characterization

The parallel runtime characterization's goal consists in identifying the key characteristics of the platform and its parallel runtime so as to build a parallel performance model and characterize its main parameters. This is accomplished by inspecting the call trees and task traces of the OpenMP microbenchmarks described in Section 4.7.2 for a number of simulations using a range of processors to determine how the platform's parallel runtime implements parallel regions and thread management, parallel loop scheduling and synchronization directives. The following subsections present the findings and results for each of the aforementioned runtime functionalities.

4.7.3.1 Characterization of OpenMP parallel regions

The parallel runtime model was built from directly inspecting task traces and call tree profiles of the *schedbench* microbenchmark. Through these elements, it was possible to describe a mechanistic model of the OpenMP runtime parallel region creation and for loop scheduling whose parameters are listed in Table 4.4. Figure 4.3 shows a timeline graph showing the steps involved in creating an OpenMP parallel region, their dependencies and the related model parameters. The first line represents the master thread, while the remaining ones represent three slave threads. The main zones are indicated via colored boxes. A detailed description of the parameters is given next.

The OP_{par_open} and OP_{par_close} parameters represent fixed costs for opening and closing a parallel region. Just after opening a new parallel region, a first phase is observed in which the OpenMP runtime first initializes and reserves each thread, sequentially. The OP_{thr_create} parameter captures this cost, while the $OP_{thr\_create\_gap}$ models the cost of the operations done in the interval between two thread creations. From inspecting profiles of parallel regions spanning different number of threads, the number of OP_{thr_create} instances has been found to be exactly $p-1$, with p representing the number of processors in the parallel region. The minus one term is due to the fact the master thread is already active and doesn't need to be re-initialized.

Once all threads have been initialized and reserved, the master thread launches the parallel code section execution on each of the slave threads. It does so by calling an OpenMP microtask on each slave thread, sequentially. An OpenMP microtask is a wrapper function introduced at compilation time around the code section encompassed by a parallelization directive. The time to launch an OpenMP microtask on a slave thread is given by OP_{thr_launch} and the interval between subsequent launches is given by $OP_{thr\_launch\_gap}$. The latter are both executed $p-1$

times, with p being the number of processors in the parallel region and present a constant time regardless of the target processor. As soon as the OpenMP microtask is called on a slave thread, the target processor starts executing the parallel section code comprised in said microtask. There are overheads however both when entering and exiting a microtask. These two overheads are modeled by the $OP_{thr\_prologue}$ and $OP_{thr\_epilogue}$ parameters, respectively.

After the master thread has launched a microtask on all slave threads, it will also execute said microtask. The overhead for launching the microtask on the master thread is given by $OP_{thr\_launch\_master}$. After launching the OpenMP microtask, the master processor behaves exactly as the slave or worker thread does and thus also perceives the thread prologue and epilogue costs.

At the end of the execution of a parallel section, a barrier ensures all threads are done before sequentially releasing the worker threads. Such a barrier cost is comprised in the $OP_{thr\_free\_master}$, the parameter that models the time for releasing the master thread. All other slave threads are then released sequentially with a cost of OP_{thr_free}. Finally, the master thread releases all allocated resources and exits the parallel region with a cost of OP_{par_close}.

4.7.3.2 Characterization of OpenMP parallel for loop scheduling

Considering a `parallel for` loop, the operation is similar to the creation of a parallel region. The `parallel` directive will create a new OpenMP parallel region, inside which the OpenMP microtasks will be launched. The parallel region creation is modeled as described in Section 4.7.3.1. Then, inside each microtask the loop iterations are distributed according to the scheduling determined in the OpenMP `parallel for` directive. In a static scheduling context, each OpenMP microtask determines which loop iterations to execute, while in a dynamic scheduling context they coordinate with other threads in a same work-sharing region to determine the next chunk of loop iterations to execute.

The overheads when entering and exiting a parallel `for` loop on a worker thread are captured by the $OP_{for\_prologue}$ and $OP_{for\_epilogue}$ parameters, respectively. In a parallel for loop, the loop iterations are first grouped in chunks which are then scheduled on the worker threads. A chunk represents a subset of the loop iterations that are executed successively on a same thread. Each thread thus executes only a portion of the loop iterations. The scheduling overhead between two consecutive loop iterations on a same chunk is modeled by the gap factor OP_{for_gap}. To schedule the execution of one chunk, both the scheduling time and order will vary greatly according to the OpenMP scheduling parameters. The time for scheduling a chunk is given by the $OP_{for\_next\_chunk}$ parameter.

A set of all of the aforementioned parameters relative to thread management and scheduling is captured for each type of schedule – simple parallel region creation, or for loop scheduling with static or dynamic scheduling policies. In a *static* schedule, scheduling a chunk is much faster, since the scheduling is known statically and therefore threads don't need to perform any implicit synchronization. Inversely, in a *dynamic* schedule, the scheduling of a chunk takes longer as each thread needs a synchronized access to runtime data structures in order to fetch the next available chunk.

4.7.3.3 Characterization of OpenMP synchronization directives

The overheads of the OpenMP thread synchronization directives – barrier and critical section – are captured via the *syncbench* microbenchmark and modeled by the parameters $OS_{barrier\_ovh}$ and $OS_{critical\_ovh}$. Note that these represent only the runtime overheads, supposing the calls to the barrier or critical section are not blocked by other threads. The blocking behavior can be modeled by dependencies among tasks in Parana's task graph, plus the measured overhead.

Parameter	Avg. Value (cycles)		
	Parallel Region	Static For Loop	Dynamic For Loop
OP_{par_open}	1091	1111	1108
OP_{par_close}	364	366	371
OP_{thr_create}	116	116	116
$OP_{thr\_create\_gap}$	28	28	28
OP_{thr_launch}	51	51	51
$OP_{thr\_launch\_gap}$	15	15	15
$OP_{thr\_launch\_master}$	32	33	33
$OP_{thr\_prologue}$	56	87	86
$OP_{thr\_epilogue}$	297	297	297
$OP_{thr\_free\_master}$	11	11	11
$OF_{for\_prologue}$	–	367	474
$OF_{for\_epilogue}$	–	689	787
OF_{for_gap}	–	59	59
$OF_{for\_next\_chunk}$	–	371	464
$OS_{barrier\_ovh}$	267	267	267
$OS_{critical\_ovh}$	276	276	276

Table 4.5: *Results for the characterization of the platform's OpenMP thread management (OP) and synchronization (OS) parameters.*

4.7.3.4 Automating the Parallel Runtime Characterization Process

Finally, after the initial investigation phase, specific characterization functionality has been incorporated into Parana to automate the characterization process. It can parse the task traces of the microbenchmark executions to rebuild their call tree profile. Parana is thus able to detect the inner function calls of the OpenMP runtime in the call tree, for any OpenMP-enabled application. From the knowledge acquired during initial investigations, the OpenMP runtime function calls on the microbenchmarks' traces can be mapped to the OpenMP directives they implement in the source code in a mechanistic fashion.

Parana collects timing information directly from the microbenchmarks task traces via the start and end timestamps of the associated OpenMP runtime functions. Each microbenchmark executes an outer-loop which is repeated three times and on each outer loop run, an inner loop which is also repeated several times. Microbenchmark tests with different sets of parameters, such as when characterizing the timings of parallel regions for a range of chunk sizes, are explicitly differentiated via explicit instrumentation directives which assigns different labels to each parameter set. Parana is thus able to parse the generated microbenchmark task traces and automatically collect the timing of the parallel runtime parameters over these multiple executions. Then, at the end of the process, it calculates the summary timing statistics for each parameter and stores the parameter statistics to the platform characterization database.

4.7.3.5 Parallel Runtime Characterization Results

Table 4.5 summarizes the results of the parallel runtime characterization parameters. Such parameters have been identified by analyzing the task traces of the OpenMP *schedbench* and *syncbench* microbenchmarks over simulations with a range of one to eight processors. The characterization has been automated using the Parana tool. The results of Table 4.5 were thus automatically extracted by the Parana characterization module from task traces of the *schedbench* and *syncbench* microbenchmarks.

Parameter	Avg. Value (cycles)	
	Gepop simulator	FPGA prototype
OM_{l1_rd}	3.0	2.8
OM_{l1_wr}	1.0	0.3
OM_{l2_rd}	14.0	16.4
OM_{l2_wr}	14.0	3.3
OM_{l3_rd}	14.0	–
OM_{l3_wr}	14.0	–

Table 4.6: *Results for the characterization of the platform's memory latency (OM) parameters.*

4.7.4 Memory Characterization

A memory characterization microbenchmark has been created based on the EPCC suite framework. The goal of this microbenchmark is to exercise a number of memory reads and write accesses in the different memory zones (L1, L2 and L3) to extract the average latencies for each operation, on each memory zone.

4.7.4.1 Memory Read Latency Characterization

The read latencies have been obtained by executing several runs of a pointer chase loop with hundreds of iterations. To measure the read latency two loops are implemented, a reference loop and a measurement loop. The reference loop reads a pointer increment value from a local variable which is stored in a processor register, sums this value to a pointer variable and accesses the ensuing pointer location. The measurement loop performs the same operation, but reads the pointer increment value from memory, then proceeds as the reference loop. The only difference is thus from where the pointer increment is read, from a register or from a memory location. The memory read access and the usage of the read value are self-contained in the same loop iteration, in consecutive instructions. The pointer increment value location has been declared volatile and loop unrolling was disabled to ensure memory accesses or the pointer computation is not optimized away or simplified. The difference in the average duration of a reference loop iteration and of a measurement loop iteration represents the extra latency for a memory read operation compared to directly accessing a register value. This operation is repeated for each memory zone in the STxP70 ASMP memory hierarchy.

4.7.4.2 Memory Write Latency Characterization

Measuring the write latency requires a modification of the pointer chase loop used for characterizing the read latency. To characterize the write latency, the value must first be written to memory, then it must be read again in the sequence, so as to create an explicit data-dependency between the write and read operations. This will effectively delay the read operation by the amount of cycles needed to write the value to memory. In this case, the reference loop for the write operation thus consists of one memory read operation that reads the pointer increment value, sums this value to a pointer variable and accesses the ensuing pointer location. The measurement loop first writes the pointer increment value to memory, and then proceeds as the reference loop. The only difference is thus that the measurement loop updates the increment value prior to reading it. The difference between the average duration of the reference loop iterations and of the measurement loop iterations gives the write latency for the target memory zone. This operation is repeated for each memory zone in the STxP70 ASMP memory hierarchy.

4.7.4.3 Memory Characterization Results

Table 4.6 lists the results of the memory characterization microbenchmark for the STxP70 ASMP as measured on the Gepop simulator and on an FPGA prototype. The fact that a write instruction is directly followed by a read instruction in the L1, does not have a big impact, probably because the latency is small enough not to halt the CPU pipeline. When accessing more remote memories, such as the L2, both the read and write latencies show similar amount of cycles on the Gepop simulator, but on the FPGA prototype the write operation continues to show a reduced latency compared to the read operation. The FPGA results for the L3 memory could not be gathered, as declaration of L3 memory resulted in an error while loading the binary file on the FPGA, error which remains unsolved. As for the L3 memory latency on the Gepop simulator, the tests showed the latencies are set to be identical to those of the L2 memory. These latency measurement results could then be used when evaluating the impact of the memory latency on the parallel performance, such as changing the latency profile of a memory access on the parallel performance estimation method to perform what-if analysis.

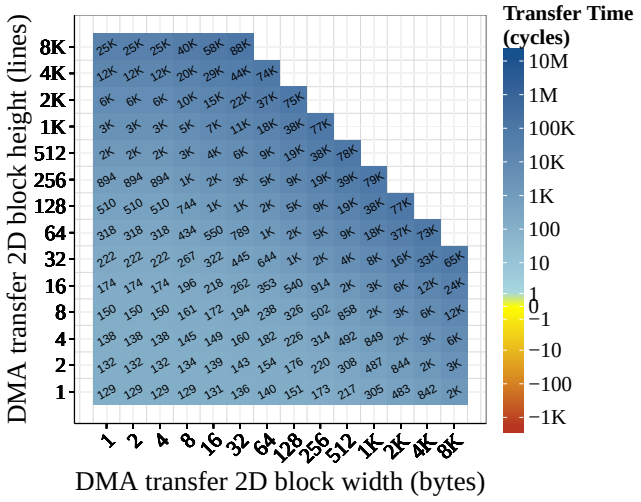
4.7.5 DMA Characterization

Modeling of the DMA data transfer times is necessary to build application performance models that integrate the memory transfers. This is particularly important in the context of this thesis for determining the best tiling dimensions so as to be able to overlap computation and data transfers in a multiprocessor with the Tilana tool, as discussed in Chapter 6.

The DMA performance model presented in this section consist of an empirical models derived from the DMA characterization data. This is in contrast with the models of the parallel runtime overheads described in previous sections which were based on mechanistic models. A method previously proposed by Saïdi et al. in [177] was used to build a first DMA performance model for the STxP70 ASMP, based on linear regression as described in Section 4.7.5.2. However, this model resulted in large prediction errors. A mixed interpolation and linear regression model is then presented in Section 4.7.5.3 which can more accurately predict the data transfer times for the STxP70 ASMP.

4.7.5.1 DMA Data Transfer Time Measurements

In order to build the performance model of the STxP70 ASMP's DMA, the first step was to characterize its data transfer times. As the DMA supports both 1D and 2D data transfers, it was important to characterize both types of transfers. The 2D data transfers are particularly useful in the context of this thesis for modeling the transfer times of images and/or image tiles in the Tilana tool. To accomplish this, a new *dmabench* microbenchmark was added to the EPCC suite. The *dmabench* microbenchmark measures the time taken to launch and wait for the completion of a DMA transfer for all valid combinations of source and destination memory zones (L1 to L2/L3, or L2/L3 to L1). It tests a comprehensive set of 1D and 2D tile sizes, for all values of width and height that are power of 2, from a minimum of 1x1 byte, up to the maximum transfer size supported by the STxP70 ASMP DMAs (256 KB). The collected information is used to build a characterization database that contains statistical information of the measured DMA transfer times over multiple runs. Figure 4.4 shows a plot of the mean timing values for varying tile sizes for L2 to L1 memory transfers.



***(a)** Characterization Values*

***Figure 4.4:** DMA transfer times obtained from the characterization with the STxP70 ASMP cycle-approximate simulator for a set of tile width and height values.*

4.7.5.2 Linear Regression Model

The goal of the DMA characterization is to build a model that can accurately predict the DMA transfer time on the STxP70 ASMP. This subsection presents an empirical model based on linear regression from the DMA characterization data of Figure 4.4. Although the linear regression model is very compact, it cannot however accurately model the timings of the DMA data transfers on the STxP70 ASMP.

The linear regression model developed in this section is derived from a DMA performance model proposed in previous work by Saïdi et al. in [177] to model an IBM Cell's DMA data transfer times. Their model assumes two distinct phases exist. The first is a *command initialization* phase, which corresponds to an initial latency that is independent of the amount of data to transfer. The second is the *data transfer* phase, whose time is proportional to the amount of data transferred. The authors thus model the DMA transfer time (t_{DMA}) of a 2D image tile ($\mathcal{T}$) as a function of its width ($w_\mathcal{T}$) and height ($h_\mathcal{T}$) by the following equation:

$$t_{DMA}(w_\mathcal{T}, h_\mathcal{T}, ds_\mathcal{I}) = I_0 + I_1 \cdot h_\mathcal{T} + \alpha(ds_\mathcal{I} \cdot w_\mathcal{T} \cdot h_\mathcal{T}) \tag{4.1}$$

where the I_0 term is the DMA *command initialization* cost, I_1 is the initialization cost for each line, α is a fixed transfer cost per byte, $ds_\mathcal{I}$ is the data size of an image pixel in bytes and $w_\mathcal{T}$ and $h_\mathcal{T}$ represent a tile $\mathcal{T}$'s width and height in pixels.

Linear regression cannot be directly applied to the model in this form as it presents interactions between variables $ds_\mathcal{I}$, $w_\mathcal{T}$ and $h_\mathcal{T}$. The objective is to recast Eq. (4.1) as a multivariate linear predictor model in the form:

$$\begin{aligned} \mathbf{Y} &= \beta \mathbf{X} \\ &= \beta_0 + \beta_1 \mathbf{X}_1 + \ldots + \beta_k \mathbf{X}_k \end{aligned} \tag{4.2}$$

An additional interaction variable is thus created according to the procedure described by Sosa-Escudero in [187]. The variable introduced is the area of the tile T expressed in bytes $\mathcal{A}^b_\mathcal{T}$, which is given by:

$$\mathcal{A}^b_\mathcal{T} = ds_\mathcal{I} \cdot w_\mathcal{T} \cdot h_\mathcal{T} \tag{4.3}$$

The linear prediction model thus assumes the following form:

$$t_{DMA}(h_T, \mathcal{A}_T^b) = \beta_0 + \beta_1 \cdot h_T + \beta_2 \cdot \mathcal{A}_T^b \tag{4.4}$$

which is equivalent to Eq. (4.1) for $\beta = (I_0, I_1, \alpha)$.

A linear regression is then performed on the characterization data to estimate the values of the β coefficients. Figure 4.5(a) shows a plot of the DMA transfer time estimated by the model for varying tile sizes. This plot shows a clear issue, which is the fact that negative values are generated for wide tiles with up to two lines.

Figure 4.5(b) shows a plot of the errors in the estimated values for the DMA transfer time, relative to the observed values. It is possible to notice that the errors are very high for small tile sizes – up to 260% of error –, but gradually decrease for large tile sizes.

We thus conclude that the linear model is not capable of capturing inherent non-linear characteristics of the DMA transfer times. These characteristics might arise due to architectural characteristics or even due to software API characteristics. Regardless of the source of any non-linear characteristics, the fact is that the model needs to account for them in order to produce accurate estimations.

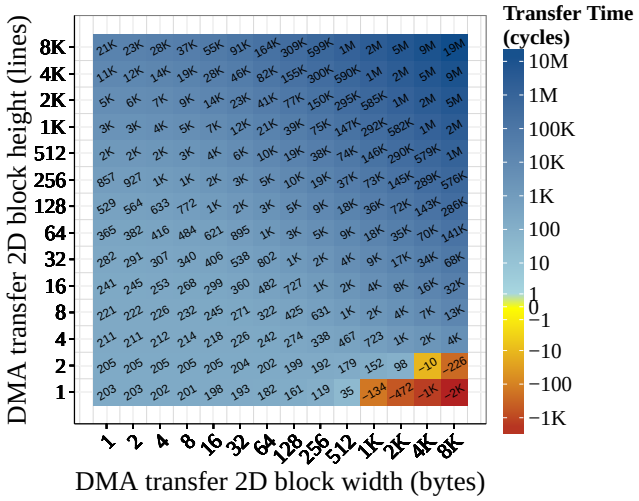
4.7.5.3 Mixed Interpolation/Regression Model

This section presents an alternative model that produces more accurate results. To minimize the error rates, especially for smaller tile sizes, we propose to estimate DMA transfer times by directly interpolating from characterization values. A simple bilinear interpolation [78] algorithm was used, a 2D interpolation algorithm traditionally used for image processing applications. As such, the transfer times of unknown points can be computed from the characterization values of its four neighbouring points very efficiently.

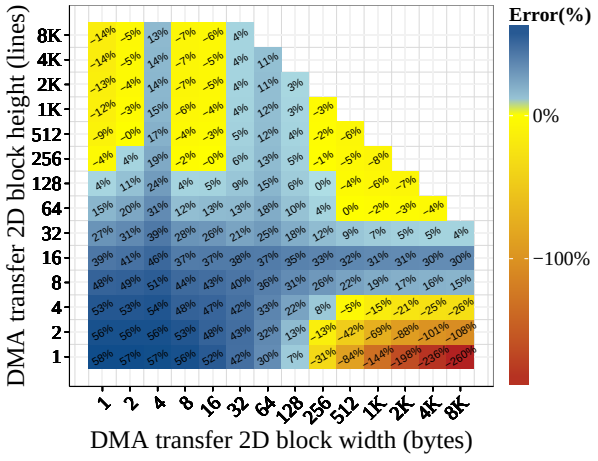
The DMA however imposes a limit on the maximum transfer data (256 KB). This implies that the maximum area of a tile cannot exceed a certain threshold, a boundary beyond which no characterization values exist. A simple bilinear interpolation algorithm is therefore incapable of predicting the transfer times of points that lie close to or on this boundary, since one of the four neighbouring points might lie beyond the boundary, and therefore have no characterized value.

To solve this issue, and allow the estimation of the transfer time for any valid point that lies close to or at such boundary, missing characterization values for points that lie beyond the boundary are extrapolated using the the predictions of the linear model from Eq. (4.4). As the error for the linear regression model around the maximum DMA transfer size boundary is low, this enables the interpolation model to still accurately predict the values of points even on the boundary.

Figure 4.6(b) shows the error results for the mixed linear regression/interpolation model. At the characterization points the interpolation error is zero by definition. As the curve is monotonic, it would be possible to determine error bounds for any point inside a zone delimited by any four characterization points, but such error bounds would be very pessimistic. An important point to notice though is that the interpolation allows a trade-off between the density of the characterization grid and the desired precision. Therefore, the density of the characterization grid can be increased until the desired precision is reached. A denser characterization grid will certainly lengthen the characterization time, but the characterization is done offline and only once for each platform, a longer characterization time is not critical for the user. The observable impact for the user on the evaluation time of the model on a denser characterization grid is marginal.

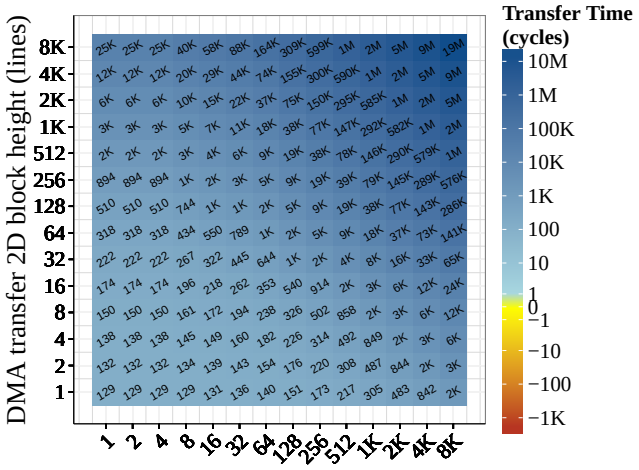


(a) Linear Regression Predictions

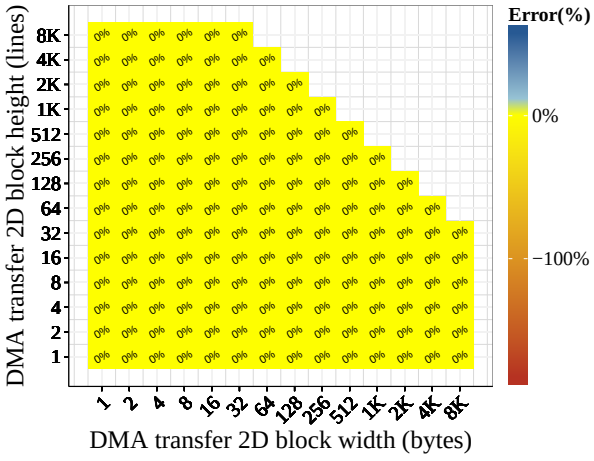


(b) Linear Regression Errors

Figure 4.5: *DMA transfer times obtained from the characterization with the linear regression model for a set of tile width and height values.*



***(a)** Mixed Interpolation/Regression Predictions*



***(b)** Mixed Interpolation/Regression Errors*

***Figure 4.6:** DMA transfer times obtained from the mixed interpolation/regression model for a set of tile width and height values.*

4.8 Conclusion

This chapter presents the process of characterizing the STxP70 ASMP multiprocessor platform, its associated OpenMP parallel runtime, as well as the process to characterize an user application. The outputs of these characterization steps are the platform and application characterization databases that will be used for the parallel application performance prediction work in Chapter 5 and for the image tiling optimization work in Chapter 6. This process can also be used by the user to profile and analyze the user application via the generated profile call trees.

First an instrumentation library is presented that allows the explicit instrumentation of the source code to control the task trace acquisition and its granularity. Examples are shown of how the instrumentation library can be used to instrument a user application and to obtain finer profiling results.

Next, this chapter discusses how the task traces of an application are generated and describes two methods to achieve this goal. The first method consists in generating the instrumentation directly in the instrumentation library, which allows the generation of the task traces even in the FPGA prototype, but which do not allow the capture of function-level information, besides adding a high overhead to the simulation. A second method which relies on a light-weight instrumentation via signaling consisting in adding write instructions to static flags which are intercepted by the Trace Filter tool has thus been preferred. This method, although not allowing the capture of timing information in the FPGA platform, allows to generate a task trace that captures the application call tree comprising explicitly instrumented code sections.

Finally, the chapter describes the multiprocessor platform characterization. The platform characterization relies on the same application task trace acquisition infra-structure to acquire traces of four EPCC OpenMP microbenchmarks. These microbenchmarks are used to characterize the OpenMP parallel runtime and the STxP70 ASMP memory hierarchy and DMA transfers. The collected task traces of the microbenchmarks are post-processed by the Parana characterization module to automatically generate the platform characterization database to be used in Chapters 5 and 6.

Chapter 5

Parallel Performance Prediction

Abstract

When designing an application-specific multiprocessor, two key questions arise: *(i)* how to size the multiprocessor platform to meet application requirements with lowest area and power consumption; and *(ii)* how to parallelize the target application in order maximize the utilization of the platform. This chapter presents a methodology for early joint parallel application and multiprocessor *Design Space Exploration* (DSE) from sequential application traces and parallelization scenarios. The DSE methodology relies on Parana, a fast and accurate performance estimation tool. Parana enables the evaluation of what-if parallelization scenarios and determines their maximum achievable performance limits. Results for a *NASA Advanced Supercomputing Division* (NAS) *Parallel Benchmark* (NPB) and two computer vision applications show that Parana provides estimations with a margin of error of less than 10% from the reference simulator, with lower modeling effort and one order of magnitude faster execution time.

Contents

5.1 Introduction

In recent years, embedded systems have followed the shift to multiprocessor architectures for their performance and energy efficiency. Nonetheless, the development of efficient and scalable parallel applications still represents a challenge [29]. An application's parallel efficiency and scalability can be severely impacted by load imbalance, as well as synchronization and communication overheads [83]. If not accurately modeled, such overheads can lead to significant mismatches between simulation results and the observed performance on physical devices, as discussed in Section 2.3. For data-parallel applications, it is well known that these overheads largely result from key software design choices, such as the parallel granularity, the scheduling policies and the data transfer strategies.

In order to compare among different implementation strategies, developers often rely on time-consuming cycle-accurate simulations, or prototypes. Fast *Instruction Set Simulator* (ISS) or *Dynamic Binary Translation* (DBT) simulators trade some accuracy loss for faster simulation speeds. Nonetheless, they all require a working parallel version of the application for each design point, which can limit the exploration space due to the effort required to produce these versions in the first place. On the other hand, existing parallel performance prediction tools – such as Kismet [105], Parallel Prophet [119], and Intel Advisor XE [97] – focus solely on desktop-class applications and do not support embedded application-specific multiprocessor design space exploration.

This chapter describes a new methodology for early *Design Space Exploration* (DSE) of application parallelization strategies and multiprocessor configurations, that allows developers to:

1. Estimate the potential speed-up of their applications.
2. Compare different parallelization strategies and multiprocessor configurations.
3. Identify performance bottlenecks and their origins.

The *contributions* of this chapter may be summarized as follows:

1. It proposes a methodology for joint parallel application and multiprocessor design space exploration from sequential code that does not require a fully-parallel version of the application.
2. It shows that said methodology can predict the parallel performance one order of magnitude faster than an ISS simulator, with a margin of error in the order of 10% for the benchmarked applications.

5.2 Overview of the Proposed Flow

The proposed methodology consists in designing an abstract trace-driven simulator to accurately estimate the performance of an embedded application under different parallelization scenarios. The target platform, in the context of this work, is the STxP70 *Application-Specific Multiprocessor* (ASMP) platform and the associated OpenMP parallel framework [159]. Figure 5.1 depicts the four steps of the parallelization prediction and analysis flow using the Parana tool, which are detailed in the sequel.

Step 1: Platform Characterization. The first step is to characterize the target multiprocessor platform and its parallel programming framework in order to build a characterization database, as discussed in Section 4.7. This database contains statistical information of the measured overheads for the OpenMP directives, as well as inherent characteristics of the multiprocessor platform, such as the memory latency parameters for each memory hierarchy level. For this,

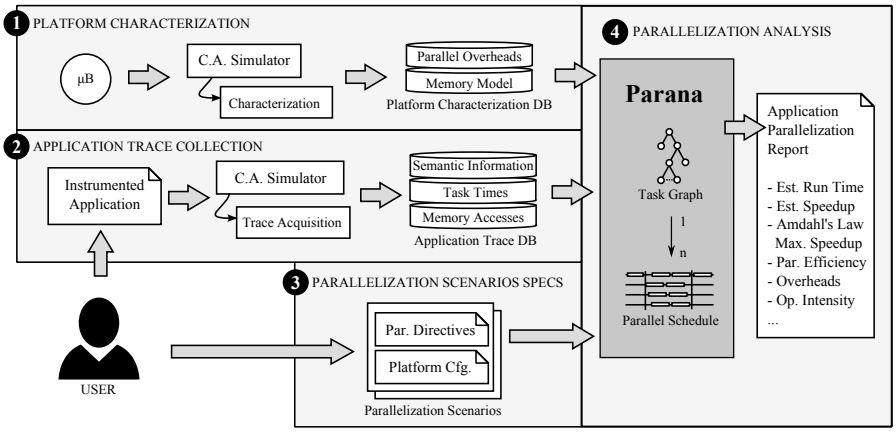


Figure 5.1: *Overview of the four steps of the proposed flow for parallel application performance prediction and design space exploration with Parana.*

an enriched version of the *Edinburgh Parallel Computing Center* (EPCC) OpenMP microbenchmarks [43] is used. The traces generated by a reference simulator or prototype are fed to a characterization tool that generates the characterization database. This database only needs to be generated once for a given multiprocessor platform and can be reused in subsequent steps.

Step 2: Application Trace Collection. This step aims to collect execution traces from the sequential application, as discussed in Section 4.5. Tasks are created from function calls in the application, which are annotated with their timestamps and hierarchical information. Summary memory access statistics are also gathered for each task. This task trace can be enriched by user inserted instrumentation macros that:

1. control the trace acquisition, and
2. define explicit tasks at lower granularity levels.

Explicit tasks allow to instrument and acquire timing information from loops and critical regions that will be later used in the parallelization analysis.

Step 3: Parallelization Scenario Specifications. This step consists in defining a number of parallelization scenarios, comprising the parallelization directives and the multiprocessor configurations the user wishes to evaluate. Two *properties* files are used – one to specify the OpenMP directives, the other to specify the multiprocessor parameters. Note that the semantics of OpenMP directives is adopted to reduce the learning curve of the tool and to enable a direct comparison with a real implementation, but the directives added at this step represent only an indication of the desired parallelization strategy to be applied in order to obtain parallel performance estimates. A full parallel implementation would require a more detailed description, greater care in defining the variable access modes and in its functional validation.

Step 4: Parallelization Analysis. In this last step, Parana uses the platform characterization and application trace databases, as well as the parallelization scenario specifications, to predict the parallel performance of the application. Parana first builds a directed acyclic graph (DAG) of the application's tasks. Then, for each parallelization scenario, it schedules this DAG to produce a corresponding parallel schedule. At the end of this process, statistics are gathered from each parallel schedule to generate a detailed parallelization report. The user can then either try new parallelization scenarios or refine the application and repeat the process to explore new design points.

Listing 5.1: *Example of instrumented code for parallelization with the Parana tool (see directives in Listing 5.2).*

```
#include "parana.h"

void vAdd(int *r, int *a, int *b, int n) {
    PARANA_TASK_INIT(_parallel0_, "parallel0");
    PARANA_TASK_INIT(_for0_, "for0");
    int i;
    PARANA_TASK_START(_parallel0_);
    for(i=0; i < n; i++) {
        PARANA_TASK_START(_for0_);
        r[i] = a[i] + b[i];
        PARANA_TASK_END(_for0_);
    }
    PARANA_TASK_END(_parallel0_);
}
```

Listing 5.2: *Example of a properties file with parallelization directives for the vAdd function (see code in Listing 5.1).*

```
vAdd = #define NUM_THREADS 8
vAdd = #define CHUNK_SIZE 1
vAdd.parallel0 = #pragma omp parallel num_threads(NUM_THREADS)
vAdd.for0 = #pragma omp for schedule(static, CHUNK_SIZE)
```

5.3 Parallel Scenario Specifications

When evaluating a given parallelization scenario, OpenMP directives can be attached to the traced tasks – either function calls or explicitly instrumented regions –, adding parallelism, scheduling and/or synchronization semantics. A *properties* file is used to specify the parallelization directives, consisting in a series of key-value pairs where keys are task names (function names or user-defined labels), and values are the intended parallelization directives. The second *properties* file is used to define ranges of values for the multiprocessor platform parameters, such as the number of processors and memory latency parameters that can override the values obtained in the characterization.

As an example, Listing 5.1 shows a vector addition function *vAdd*, the same as shown in Listing 4.3 from Chapter 4, instrumented with two labeled tasks: *parallel0* and *for0*. Listing 5.2 shows one possible set of parallelization directives for the *vAdd* function that creates a parallel region with 8 threads and a for loop with static scheduling and a chunk size of 1. Multiple such *properties* files can be provided to specify different parallelization scenarios and multiprocessor configurations and then used by the Parana tool to predict the performance in each scenario.

5.4 Parana

Parana is a trace-driven simulator that uses a mechanistic model to emulate the OpenMP runtime scheduling of an application's tasks. A number of schedules are produced, one for each parallelization scenario, and used to generate a detailed parallelization analysis report.

5.4.1 Performance Modeling

The underlying performance model of the Parana trace-driven simulator is a mechanistic[1] model of the multiprocessor platform and of the OpenMP runtime. This model defines how the OpenMP directives will impact the application task scheduling and, consequently, the execution time. Its goal is to be able to quickly estimate the performance of an application for different parallelization scenarios.

The reason for selecting such a model is because it can accurately model the dynamic data-dependent task times related to a particular input data set as well as the parallel runtime overheads with enough accuracy for evaluating parallelization trade-offs with less effort and faster than with an ISS simulator.

The performance model was constructed by analyzing the task traces and the call tree profiles of the EPCC OpenMP *schedbench* and *syncbench* microbenchmarks used to characterize the OpenMP parallel runtime. Specific characterization functionality has been added to Parana so that it can automatically extract the values of the characterization parameters. Section 4.7.3 describes the characterization of the platform's parallel runtime that leads to the performance model defined herein.

As a mechanistic performance model, this model estimates the parallel performance by replaying the collected application task traces and emulating the behavior of the OpenMP parallel runtime on the STxP70 ASMP. Task traces are loaded once and then used to produce multiple schedules for different parallelization scenarios. Tasks in the generated schedules are assigned particular categories for reporting purposes so as to enable the user to identify particular bottlenecks of the application. Therefore, separate categories are defined for the sequential and parallel computations, for the OpenMP runtime overheads, as well as for load imbalance and idle time. The duration of tasks that represent load imbalance and idle time result from inter-task dependencies and are derived from the parallel schedule.

This performance modeling section is structured as follows. First, Section 5.4.1.1 defines the elements that represent a task, a task graph and a schedule. Then, Section 5.4.1.2 lists the steps in Parana's performance estimation flow. Section 5.4.1.3 describes how the memory latency information can be incorporated to adjust task times analytically. Finally, Section 5.4.1.4 details the algorithms for constructing the parallel schedule.

5.4.1.1 Definitions

Task T. A Task T stores information from the application characterization, such as its start and end time in the reference sequential execution, and the memory access summaries. Tasks are created from the application traces at each application function call and user-instrumented code section. A distinction is done between hierarchical Tasks and leaf Tasks.

Hierarchical Task Graph (HTG) H. Parana builds an HTG to represent the application structure, and is an input of Parana's task scheduler. The HTG is a directed acyclic graph (DAG), where vertexes are Tasks, and edges represent parent-child relations between Tasks. An HTG is thus represented as $H = G(V, E)$, where G is a DAG with a vertex set V and a directed edge set E. A directed edge from Task T_a to Task T_b is represented as $(T_a \rightarrow T_b)$.

Schedulable Task ST. A schedulable task ST is a wrapper that adds scheduling information to a leaf Task T, such as its start time $t_{start}(T)$ and duration $w(T)$ in a given Schedule, a mapping

[1] Sarokin provides a concise definition of what is a mechanistic model in [178], where he states that: "A mechanistic model assumes that a complex system can be understood by examining the workings of its individual parts and the manner in which they are coupled." . It is therefore a model that relies on the knowledge of the inner workings of a system, as opposed to models derived solely from empirical observations. For more information on mechanistic modeling refer to the works of Craver [54] and Tham [199].

to a Processor, and a type for reporting purposes. Task types can be one of: sequential processing (SEQ_PROC), parallel processing (PAR_PROC), runtime overhead (RT_OVH), load imbalance (LD_IMB), or idle time ($IDLE$). Schedulable Tasks that do not have a defined duration and whose duration is determined by inter-task dependencies in the Schedule are called Filler Schedulable Tasks. The latter are mainly used to derive idle and load imbalance time.

Schedule S. A Schedule is the output of Parana's task scheduler, and consists in a DAG, where vertexes are Schedulable Tasks and edges represent end-to-start precedence relations between them.

5.4.1.2 Performance Estimation Steps

Algorithm 5.1 lists the steps in Parana's performance estimation flow. In order to exemplify the execution of the estimation flow, consider the code of the *Fast Features for Accelerated Segment Test* (FAST) Corner Detection applications in Listing 5.3 and the properties file in Listing 5.4 which defines a static inner-loop parallelization scenario.

Upon launch, Parana loads the platform and application databases and builds an hierarchical task graph H. Figures 5.2 and 5.3 depict an execution timeline for the FAST application and the corresponding hierarchical task graph H. In H, hierarchical Tasks encompass their children, and as such, present a timing overlap with them. In order to eliminate these overlaps, the *segmentation* process adds, to each parent Task, new leaf Tasks which cover the regions where the parent Task does not overlap with its children, so as to allow scheduling. Figures 5.4 and 5.5 show the segmented tasks added in a timeline view and in the segmented hierarchical task graph H_{seg}.

After H has been segmented to produce H_{seg}, a parallelization scenario is loaded, and the tasks in H_{seg} are annotated with the specified parallelization directives to produce the annotated hierarchical task graph H_{ann}. Figure 5.6 illustrates the annotated hierarchical task graph H_{ann} for the FAST application, whose annotated tasks are denoted by the # symbol. The new annotated HTG H_{ann} is the input of Parana's task scheduler, which produces a number of Schedules S for each set of platform configuration parameters. The task scheduler takes into account the OpenMP overheads and the memory model of the platform characterization database to produce precise estimations. Figure 5.7 depicts sequential and parallel schedules generated for the FAST application. Finally, the parallel schedules are used to produce a detailed parallelization report, and the process is repeated for each parallelization scenario.

5.4.1.3 Memory Latency Modeling

As the sequential execution traces are acquired from a cycle-approximate simulator of the same target platform, a leaf task's instruction sequence and timing are assumed to remain roughly unchanged inside parallel sections. The execution time w of a schedulable task ST introduces memory latency overheads, allowing the user to estimate the impact of a different set of memory latency parameters, and is given by:

$$w(ST) = w(T) + \sum_{\substack{\ell \in \{l1,l2,l3\} \\ \theta \in \{rd,wr\}}} T_{\ell\_\theta} \times OM_{\ell\_\theta} \tag{5.1}$$

where T is a leaf Task, ST is a Schedulable Task, $w(T)$ is the execution time of T, T_{ℓ_θ} is the number of accesses of type $\theta \in \{rd, wr\}$ in memory level $\ell \in \{l1, l2, l3\}$ in Task T, and OM_{ℓ_θ} is the memory latency parameter for accesses of type θ in memory level ℓ.

Algorithm 5.1 Parana's Performance Estimation Steps

1: Open the platform characterization database;
2: Open the application trace database;
3: Build an hierarchical task graph H of the application;
4: Segment H into H_{seg} by introducing hierarchically non-overlapping leaf Tasks for scheduling;
5: Load a parallelization scenario;
6: Annotate the tasks in H_{seg} with the parallelization directives to produce H_{ann}
7: Perform a baseline sequential scheduling of H_{ann} for one set of platform parameters;
8: Write results to the report;
9: Perform a parallel scheduling for a given number of cores with the same platform parameters;
10: Write results to the report;
11: Repeat steps 9-10 for each target number of cores;
12: Repeat steps 7-11 for each set of platform parameters;
13: Repeat steps 5-12 for each parallelization scenario;
14: Exit;

Listing 5.3: *Example of instrumented code for parallelization with the Parana tool (see directives in Listing 5.2).*

```
#include "parana.h"
...

void fastCornerDetect(...) {
   PARANA_TASK_INIT(y_e, "for_y");
   PARANA_TASK_INIT(x_e, "for_x");
   ...
   //Outer loop over lines
   for (y = 3; y < height - 2; y++) {
       PARANA_TASK_START(y_e);
       ...
       //Inner loop over pixels
       for (x = 3; x < width - 3; x++) {
           PARANA_TASK_START(x_e);
           //Compute FAST corner at point (x,y)
           ...
           PARANA_TASK_END(x_e);
       }
       ...
       PARANA_TASK_END(y_e);
   }
   ...
}
```

Listing 5.4: *Example of a properties file with parallelization directives for the vAdd function (see code in Listing 5.1).*

```
fastCornerDetect.for_x = \
        #pragma omp parallel for schedule(static) num_threads(8)
```

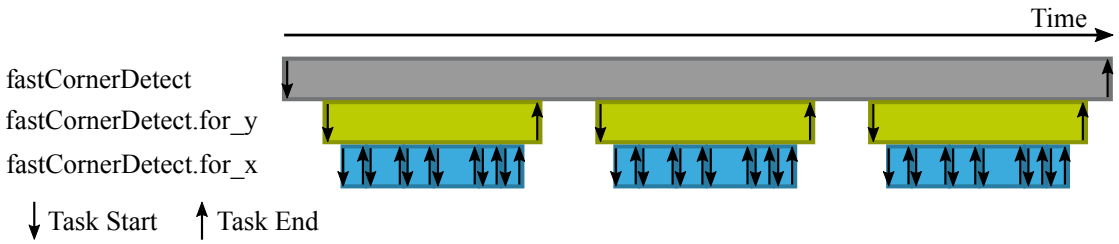


Figure 5.2: *Representation of a FAST Corner Detection execution timeline and the Parana task trace events generated.*

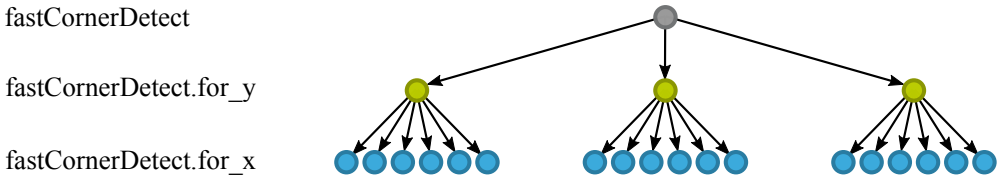


Figure 5.3: *Representation of a Parana's Hierarchical Task Graph H for the FAST Corner Detection application.*

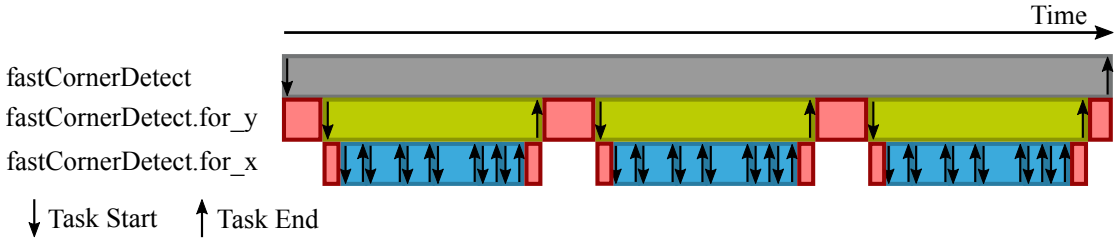


Figure 5.4: *Representation of a segmented FAST Corner Detection execution timeline and the Parana task trace events.*

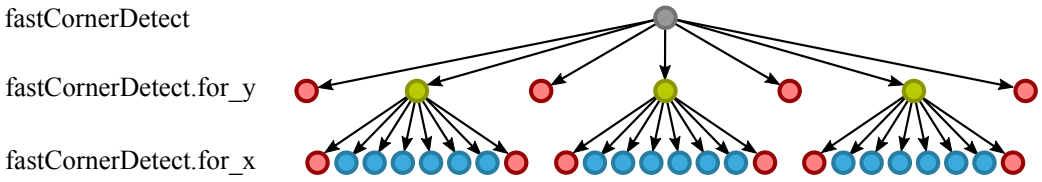


Figure 5.5: *Representation of a Parana's Segmented Hierarchical Task Graph H_{seg} for the FAST Corner Detection application.*

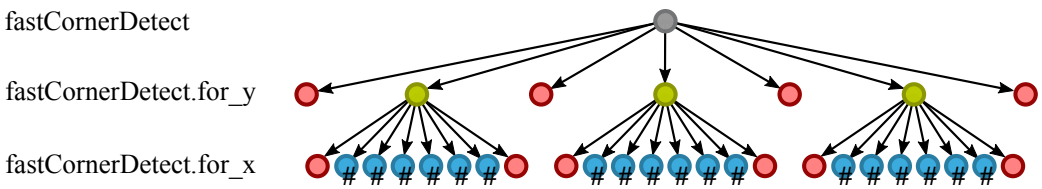


Figure 5.6: *Representation of a Parana's Annotated Hierarchical Task Graph H_{ann} for the FAST Corner Detection application.*

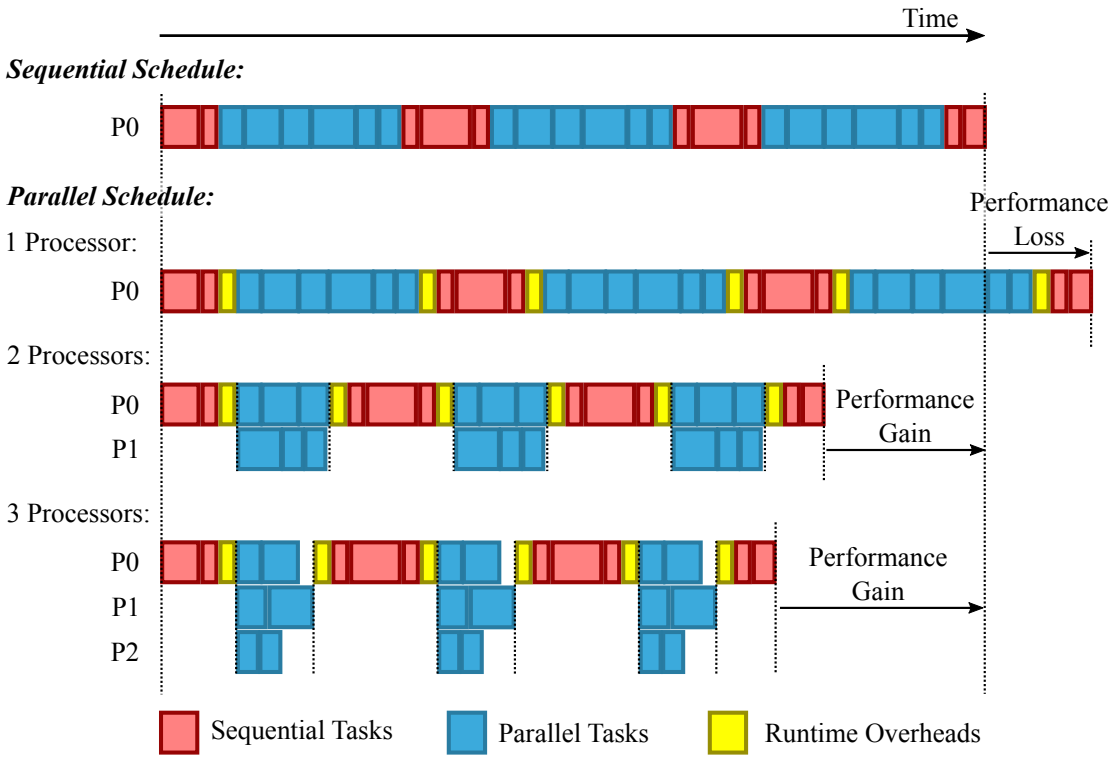


Figure 5.7: *Simplified representation of sequential and parallel Parana schedules for the FAST Corner Detection application.*

5.4.1.4 Task Scheduling

Parana's task scheduler is a key element in the parallel performance estimation flow. Parana initially builds a sequential schedule that will serve as a reference for computing speedup and parallel efficiency figures. Then, it repeats the process to build one parallel schedule for each parallelization scenario. When constructing a parallel schedule, Parana's task scheduler emulates the scheduling performed by the OpenMP runtime in a mechanistic way.

The task scheduler builds a schedule by traversing the annotated hierarchical task graph H_{ann} in a depth-first order and, for each leaf Task found, wrapping it in a Schedulable Task and adding it to the schedule. As Schedulable Task are added to the schedule, their start and end times are updated according to the constraints defined in Eqs. (5.2)–(5.4).

$$w(ST) = t_{end}(ST) - t_{start}(ST) = t_{exe}(ST) \geq 0 \tag{5.2}$$

$$t_{start}(ST) \geq max(t_{end}(ST_{in})), \forall ST_{in} \in V \,|\, (ST_{in} \rightarrow ST) \in E \tag{5.3}$$

$$t_{end}(ST) \leq min(t_{start}(ST_{out})), \forall ST_{out} \in V \,|\, (ST \rightarrow ST_{out}) \in E \tag{5.4}$$

Upon encountering tasks with attached OpenMP directives while traversing the hierarchical task graph, the scheduler will enforce the OpenMP scheduling policies and will add tasks to model the OpenMP runtime overheads. Figure 5.7 exemplifies the sequential and the parallel schedules for the FAST application with a static inner-loop parallelization from Listings 5.3 and 5.4. Algorithms 5.2–5.3 exemplify the creation of a schedule, the scheduling of a sequential task and its children, as well as the scheduling of a new parallel region. The schedule of parallel for loops is similar, using the same principles to throttle the launch of threads and loop iterations, and adding OpenMP for loop (OF) scheduling overheads listed in Table 4.5 from Section 4.7.3.

Algorithm 5.2 Parana's Task Graph Scheduling

Input: A Task Graph HTG and number of processors n
Output: A schedule S

```
1: function Schedule(HTG, n)
2:     S ← NewEmptySchedule(HTG, n)
3:     P ← GetProcessors(S)
4:     P_master ← GetMasterProcessor(P)
5:     T_root ← GetRootTask(HTG)
6:     ScheduleSequentialTask(T_root, P_master, S)
7:     return S
8: end function
```

Input: A Task T, a processor P and a schedule S

```
1: function ScheduleSequentialTask(T, P, S)
2:     if HasDirective(T, parallel) then
3:         ScheduleParallelRegion(T, P, S)
4:     else
5:         if IsLeaf(T) then
6:             AddTailNode(NewSchedTask(T), P, S)
7:         else
8:             for all T_child ∈ ChildrenOf(T, S) do
9:                 ScheduleSequentialTask(T_child, P, S)
10:            end for
11:        end if
12:    end if
13: end function
```

Algorithm 5.3 Parana's Parallel Region Scheduling

```
1: function ScheduleParallelRegion(T, P, S)

   // 1. Create sub-schedule and open parallel region
2:     n_thr ← GetOMPDirectiveValue(T, num_threads)
3:     n_thr ← Min(n_thr, NumProcessors(S))
4:     S_sub ← NewSubSchedule(n_thr, P, S)
5:     AddTailNode(NewSchedTask(OP_par_open), P, S_sub)

   // 2. Create worker threads
6:     for all P_i ∈ Processors(S_sub) | P_i ≠ P do
7:         ST_thr_create ← NewSchedTask(OP_thread_create)
8:         AddTailNode(ST_thr_create, P_i, S_sub)
9:         ST_thr_create_gap ← NewSchedTask(OP_thr_create_gap)
10:        AddTailNode(ST_thr_create_gap, P_i, S_sub)
11:        ST_idle ← NewFillerSchedTask(IDLE)
12:        AddTailNode(ST_idle, P_i, S_sub)
13:        ST_fork_wait ← NewFillerSchedTask(RT_OVH)
14:        AddTailNode(ST_fork_wait, P_i, S_sub)
15:        AddEdge((ST_thr_create → ST_fork_wait), S_sub)
16:    end for
```

(To be continued on the next page)

Algorithm 5.2 (Continued) Parana's Parallel Region Scheduling

// 3. Launch worker threads
for all $P_i \in$ Processors$(S_{sub}) \mid P_i \neq P$ **do**
  $ST_{thr\_launch} \leftarrow$ NewSchedTask(OP_{thread_launch})
  AddTailNode$(ST_{thr_launch}, P, S_{sub})$
  $ST_{thr\_launch\_g} \leftarrow$ NewSchedTask$(OP_{thr_launch_gap})$
  AddTailNode$(ST_{thr_launch_g}, P, S_{sub})$
  $ST_{fork\_done} \leftarrow$ NewFillerSchedTask(RT_OVH)
  AddTailNode$(ST_{fork_done}, P_i, S_{sub})$
  AddEdge$(\left(ST_{thr_launch} \rightarrow ST_{fork_done}\right))$
end for

// 4. Launch master thread
$ST_{thr\_launch\_m} \leftarrow$ NewSchedTask$(OP_{thr_launch_master})$
AddTailNode$(ST_{thr_launch_m}, P)$

// 5. Schedule parallel task on each thread
for all $P_i \in$ Processors(S_{sub}) **do**
  ScheduleParallelTask(T, P_i, S_{sub})
  $ST_{ld\_imb} \leftarrow$ NewFillerSchedTask(LD_IMB)
  AddTailNode(ST_{ld_imb}, P_i)
end for

// 6. Join parallel threads
$ST_{join} \leftarrow$ NewSchedTask(RT_OVH)
AddTailNode(ST_{join}, P)
for all $P_i \in$ Processors$(S_{sub}) \mid P_i \neq P$ **do**
  $ST_{ld\_imb} \leftarrow$ LastTask(P_i, S_{sub})
  AddEdge$(\left(ST_{ld_imb} \rightarrow ST_{join}\right))$
end for

// 7. Release master thread
$ST_{thr\_free\_master} \leftarrow$ NewSchedTask$(OP_{thr_free_master})$
AddTailNode$(ST_{thr_free_master}, P, S_{sub})$

// 8. Release worker threads
for all $P_i \in$ Processors$(S_{sub}) \mid P_i \neq P$ **do**
  $ST_{thr\_release\_i} \leftarrow$ NewSchedTask(OP_{thr_free})
  AddTailNode$(ST_{thr_free_i}, P, S_{sub})$
end for

// 9. Close parallel region
$ST_{par\_close} \leftarrow$ NewSchedTask(OP_{par_close})
AddTailNode$(ST_{par_close}, P, S_{sub})$

end function

5.4.2 Parallelization Analysis Report

Parana's parallelization analysis reports provide detailed information for any given parallelization scenario estimation. The information in such reports is used to build cycle stacks, a graph that clearly shows the execution time breakdown and helps the user identify performance bottlenecks. Parana's task schedules are the primary source of information for the parallelization analysis reports, which list for each scenario:

1. the total execution time for the schedule;
2. the speedup and parallel efficiency with respect to the baseline sequential schedule;
3. Amdahl's law's [25] maximum theoretical speedup; and
4. separate execution time statistics for the sequential and the parallel portions of the schedule, broken down into processing time, idle time, and runtime overheads.

Additionally, Parana can produce similar reports per parallel or work sharing region. This gives the user the ability to view, in a single report, the detailed statistics at application-level, as well as for each OpenMP parallel region and more readily assess parallelization gains and bottlenecks. An example of a Parana's output report for the FAST application – for the parallelization scenario consisting of an outer-loop parallelization with dynamic scheduling – is given in Appendix A, Listing A.1.

5.5 Experimental Setup

5.5.1 Platform Architecture

The target platform is the STxP70 ASMP from STMicroelectronics, which is analogue to a single cluster of the STHORM many-core platform [149]. The STxP70 ASMP is a configurable *Symmetric Multiprocessing* (SMP) architecture with up to 16 STxP70 cores – 32-bit dual-issue *Reduced Instruction Set Computer* (RISC) CPUs. Its architectural template is depicted in Figure 5.8. The experiments are done on an STxP70 ASMP configuration with 512 KB of shared data memory, organized into 32 memory banks. For more details on the STxP70 ASMP platform and its parallel runtime refer to Section 2.2.3.

5.5.2 Execution Vehicles

Gepop Simulator. Gepop is a cycle-approximate simulator for the STxP70 ASMP platform and constitutes the reference platform for the platform characterization in Parana. It is built over STxP70 ISS simulators and integrates hardware device models for other platform components such as DMAs, memories and interconnects. Its margin of error is evaluated to be in the order of 10% from that obtained from a physical device.

Field Programmable Gate Array (FPGA) *Prototype.* A prototype of the STxP70 ASMP on the Xilinx VC707 FPGA is used for comparison. Due to device occupancy constraints, the core count and the L1 shared data memory size in the FPGA prototype are limited to 8 STxP70 cores and 512 KB, respectively. FPGA prototype runs are used to validate the timing figures provided by Gepop and the Parana estimates.

Parana. The proposed tool is a trace-diven simulator that can estimate the parallel performance of an application from its sequential execution traces and a set of OpenMP directives for different parallelization scenarios. It uses Gepop produced traces for characterization, as well as for the reference sequential traces for its parallelization estimates.

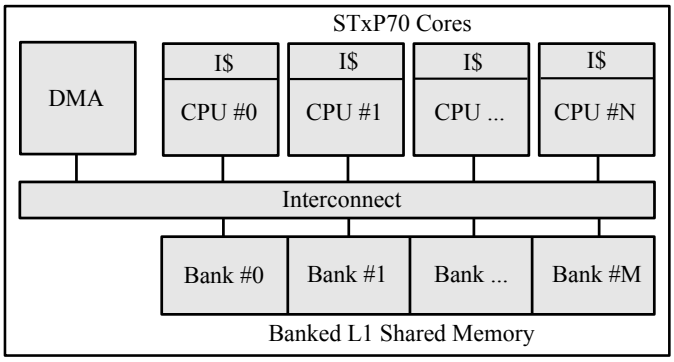


Figure 5.8: *Architectural template of the STxP70 ASMP.*

5.5.3 Applications

Integer Sort (IS). The *NASA Advanced Supercomputing Division* (NAS) *Parallel Benchmark* (NPB) suite's [108] IS benchmark was selected as a parallel OpenMP version implemented in C is directly provided. Two adaptations were made to the original version. First, due to the limited shared L1 data memory size, a smaller data class has been implemented to fit the memory constraints, which was named Tiny class, or T class. Second, as the STxP70 ASMP's OpenMP runtime does not yet support the *threadprivate* OpenMP directive used in the random function that generates the values to be sorted, they were replaced with the *rand_r* function.

Fast Features for Accelerated Segment Test (FAST). A 9-16 FAST corner detector commonly used in computer vision to select key points in a number of tracking and detection applications has been ported onto the STxP70 ASMP. The reference implementation is that provided in OpenCV [42] version 2.4.6, which has been rewritten in C for the STxP70 ASMP and parallelized with OpenMP. Four parallelization scenarios are tested:

1. Inner-loop parallelization with static scheduling.
2. Inner-loop parallelization with dynamic scheduling.
3. Outer-loop parallelization with static scheduling.
4. Outer-loop parallelization with dynamic scheduling.

Edge Detection (ED). The edge detection application contains three main kernels: the first is a 3×3 low-pass filter; the second is a 3×3 separable Sobel filter used to compute edge orientation and magnitude, and then mark edge pixels and assign them a weight; and the third is a non-maxima suppression kernel that keeps only the strongest edge pixels in a 3×3 neighborhood. The kernels execute in sequence and are parallelized independently. Four parallelization scenarios are tested, which are the same as for the FAST application.

5.6 Results

5.6.1 Speedup Estimates

Figures 5.9–5.11 provide the application parallel speedup results for the selected applications, as well as the relative error of Parana's estimations relative to the reference Gepop simulator. The application parallel speedup for n processors is computed as $S(n) = \frac{t_{seq}}{t_{par}(n)}$, or the ratio between the duration of the sequential baseline schedule and the duration of the parallel schedule with n processors in parallel. From these results, we conclude that Parana's parallel speedup estimates have an average mean percentage error of 5.2%, with a maximum absolute error of 12.4%.

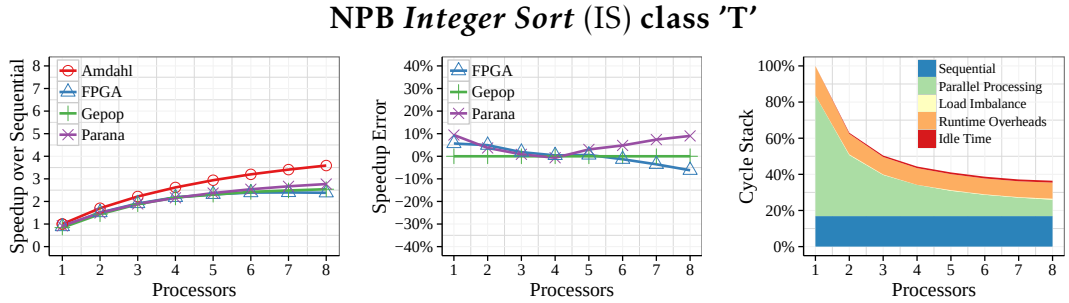


Figure 5.9: *Comparison of the NPB IS benchmark speedup estimates from Parana against measurements from the Gepop ISS simulator and an FPGA prototype. Left: speedup relative to the sequential version. Center: Speedup error relative to Gepop. Right: Cycle stacks showing the impact of different overhead sources.*

The NPB Integer Sort benchmark presents a high portion of sequential code, which can be seen in the cycle stack in Figure 5.9. The high proportion of sequential code limits the maximum theoretical parallel speedup attainable to $\sim 3.5x$ for 8 cores, as can be seen in the Amdahl's Law speedup curve. The low portion of parallel code incurs from the small data set used. The overall speedup for this data set is thus very poor. From the speedup and error curves it is possible to see that Parana's speedup predictions follow the Gepop and FPGA measurements. Higher mismatches occur with increasing number of threads, which are mostly due to a reduction operation that is dependent on the number of threads. This operation is not executed in the sequential version of the application, and thus not captured in the application traces used by Parana. The contribution of the runtime overheads are significant for such a reduced dataset, especially for higher processor counts.

The FAST Corner Detection application in Figure 5.10, has a very high parallel portion for an outer-loop parallelization, which enables a good parallel efficiency. For the inner-loop parallelization, the maximum attainable speedup factor as per the Amdahl's Law is just above $6x$, but both the static and the dynamic scheduling policies saturate under $3.5x$ speedup due to the increased overheads for a finer-grained scheduling. It is possible to see that the sources of the overheads differ for both scheduling policies. While the static configuration presents a lower runtime overhead, it also presents a poor load balancing. The dynamic configuration virtually eliminates the load imbalance, but incurs higher runtime overheads.

The Edge Detection application in Figure 5.11, also presents a very high parallel portion for an outer-loop parallelization, enabling nearly ideal speedups. Due to the high granularity and the absence of load imbalance, the scheduling policies are not critical and present almost identical results. For the inner-loop scheduling, the parallel portion drops, but is still higher than for the FAST application. This results in a high parallel efficiency even using a finer granularity. Differently from the FAST application, the static scheduling in the inner-loop parallelization shows no significant load imbalance, with the highest overhead remaining the runtime itself. Since scheduling costs for the dynamic scheduling policy are significantly higher, the speedup achieved in scenario 2 saturates at $\sim 4x$ speedup for 8 cores.

Overall, Parana performance estimates show a good precision when compared to the Gepop and FPGA measurements. The Amdahl's Law figures computed by Parana provide clear guidance to the programmer on the maximum speedup that can be expected. Moreover, effects that negatively impact the parallel performance can be easily identified with the provided cycle stacks.

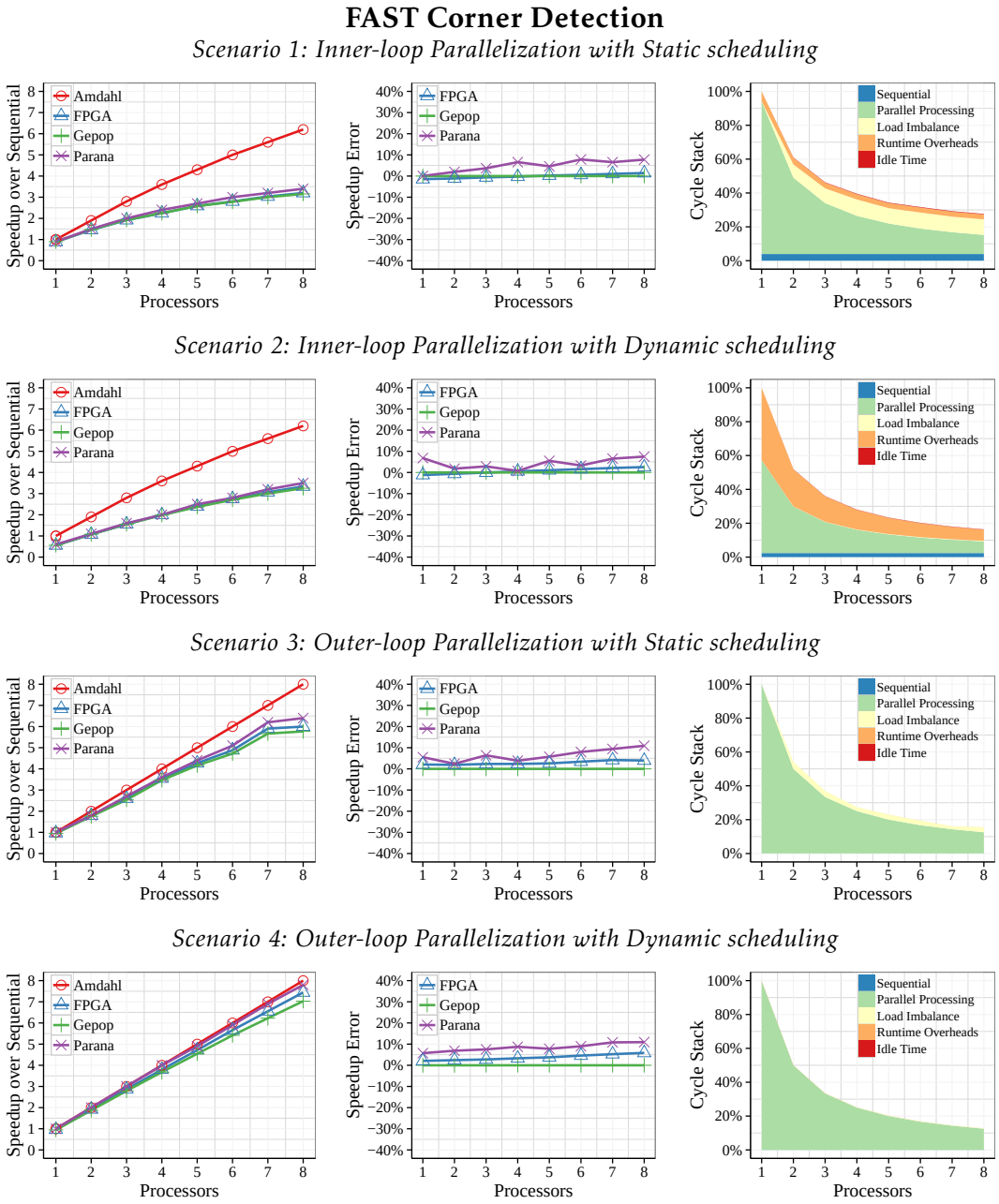


Figure 5.10: *Comparison of the FAST Corner Detection speedup estimates from Parana against measurements from the Gepop ISS simulator and an FPGA prototype. Left: speedup relative to the sequential version. Center: Speedup error relative to Gepop. Right: Cycle stacks showing the impact of different overhead sources.*

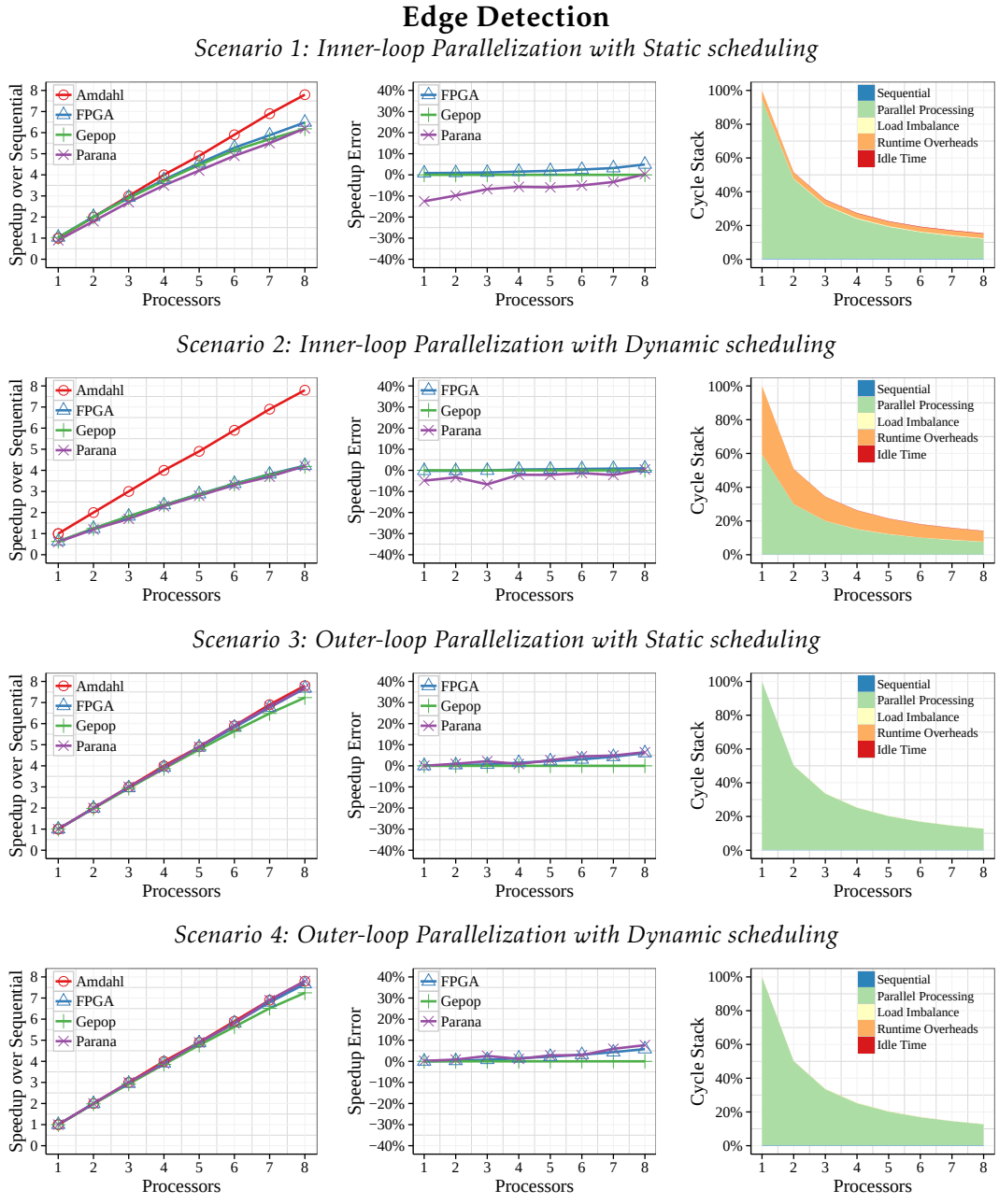


Figure 5.11: *Comparison of the Edge Detection speedup estimates from Parana against measurements from the Gepop ISS simulator and an FPGA prototype. Left: speedup relative to the sequential version. Center: Speedup error relative to Gepop. Right: Cycle stacks showing the impact of different overhead sources.*

Tool	Type	Inputs		Design + Setup		DSE			
		Application	Instr.	Time	Effort	Exe. Time	Productivity	Error	Accuracy
FPGA	Hardware Prototype	Parallel	No	Days	High	4:16s	Low	< 5%	High
—	Cycle-Accurate Sim.	Parallel	No	Hours	High	—	Low	< 5%	High
Gepop	Cycle-Approximate Sim.	Parallel	No	Hours	Medium	4:33s	Medium	< 10%	Medium
Kismet	Critical Path Analysis	Sequential	No	Seconds	Low	—	Low	> 10%	Low
Parallel Prophet	Mechanistic Analytical Model	Sequential	Yes	Minutes	Medium	—	Low	< 10%	Medium
Parana	Mechanistic Analytical Model	Sequential	Yes	Minutes	Low	0:12s	High	< 10%	Medium

Table 5.1: *Comparison of simulation methods and related tools for parallel performance prediction. Setup effort: time needed to prepare the software and platform for each design point (low: no modifications to original code; medium: some modification required; high: fully-functional version required). Productivity: overall efficiency to setup and evaluate multiple design points (low: time-consuming; high: fast). Accuracy: reported results accuracy compared to the final system performance (low: >10% error; medium: <10% error; high: <5% error).*

5.6.2 Execution Time

Table 5.1 lists the average execution time for all design points evaluated in the experiments with the Gepop simulator, the FPGA prototype and Parana. Parana's execution time was in average 22x faster than the Gepop simulator. The emulation with the FPGA prototype is not much faster than Gepop due to the time it takes to load the application code and to interactions with the host PC in system calls, resulting in a 20.5x faster execution time with Parana on average.

5.6.3 Key Benefits

The proposed methodology provides accurate parallel performance estimates early in the design flow, even before having a working parallel version of the application. It allows for fast design space exploration of both application parallelization scenarios and multiprocessor configurations. The support of instruction set extensions demands only reacquiring the traces with a rebuilt application on the extended platform. The cycle stacks aid system architects to identify the most important parallel performance bottlenecks in the application.

5.6.4 Known Limitations

A key limitation of the methodology described herein is that it does not properly model applications that circumvent standard OpenMP constructs to implement other sort of explicit or dynamic parallel workload distribution. Memory contention and DMA transfers are not currently modeled, but will be addressed in the near future. The applications in the experiments do not make use of the DMA and do not seem to suffer from degradation due to memory contention. Calls to the DMA *Application Programming Interface* (API) can be easily intercepted with the proposed methodology and scheduled onto a dedicated resource in Parana's schedule to model a DMA.

As for memory contention modeling, although a detailed memory contention analysis would require tracing individual memory accesses timestamps, we believe the current model can be easily extended to add a burden factor that estimates the contention overhead. This factor would be determined by a statistical model based on the memory load in a given time interval, similar to the cache modeling approach in [119].

5.7 Conclusion

This section presents a methodology and a tool – Parana– for early joint design space exploration of application parallelization strategies and multiprocessor platform's parameters from sequential application traces. The results of the proposed methodology were compared against metrics obtained from the execution of three OpenMP applications on the Gepop cycle-approximate simulator and on an FPGA prototype of the STxP70 ASMP, under multiple parallelization scenarios. We have demonstrated the accuracy of the solution in estimating the parallel speedup, with a margin of error in the order of 10% from the reference ISS simulator, as well as its interest in identifying the sources of scalability issues via the cycle stacks. Parana is thus accurate enough to help the designer to select only the most promising parallelization strategies very early in the design process, an order of magnitude faster and with less effort than with a traditional ISS simulator.

Our future explorations will aim at modeling DMA transfers and shared memory conflicts to increase Parana's domain of addressable applications and prediction accuracy. We also wish to investigate the usage of Parana's task graphs to estimate the effect of higher-level loop transformations, such as loop fusion or loop tiling.

Chapter 6

Image Tile Sizing Based on Non-Linear Constraints

Abstract

Tiling is a key aspect of the design of embedded image processing applications, due to local memory constraints. To maximize system performance, the designer must select a suitable tile size that balances data transfers and computation. This chapter presents a method for optimal $2D$ image tile sizing using constraint programming. Unlike previous algebraic methods, like those from Darte et al. [57] and Feautrier [67, 68], and recent models based on constraint optimization, such as proposed by Saïdi et al. [177], the model proposed herein is a constraint optimization model that integrates non-linear DMA data transfer times and parallel scheduling overheads for an increased accuracy. The experiments with a binomial filter demonstrates that the proposed method allows to compute the optimal tiling dimensions that minimize the execution time for different image sizes and internal memory constraints. This technique provides invaluable information for both application developers and system architects that can quickly explore design trade-offs.

Contents

6.1 Introduction

Image processing applications that run on embedded multiprocessor platforms often use input/output data buffers that are located in an external memory. Such data buffers are used either as a way to exchange data between different subsystems, or simply because the amount of data is too large to fit a system's limited data memory. As an external memory's access time is much higher than that of the local memory, the data needs to be transferred to the local memory to be processed locally. The output data are then written back to the external memory. In systems with explicitly managed memory, with no data caching mechanism, it is up to the programmer to manage all data transfers between external and internal memory spaces. A common solution to this problem is to apply the tiling technique, which consists in subdividing the data into smaller subregions, also called tiles, that are small enough to fit the local memory and that can be processed independently.

In order to make a better use of the available resources, tile processing is usually implemented as a three-stage pipeline whose stages are: *(i)* read input data from the external memory and store it in a temporary local memory buffer; *(ii)* process the data locally; and *(iii)* write the output data back to the external memory. Figure 6.1 illustrates such a three-stage tile processing pipeline. Tiling benefits include reducing the footprint of local memory buffers and reducing the application's latency by allowing the overlap of data transfers and computation.

The question is how to select optimal tiling dimensions? Larger tiles present better data sharing, but result in larger latencies for the read and write stages. Conversely, smaller tiles incur higher overheads and their transfer time might be dominated by the DMA latency. Moreover, the tile aspect ratio is also important. While a square tile might improve data sharing, wider tiles tend to present lower transfer times. This is due to the asymmetry in DMA data transfer times, which incur additional overheads when accessing a different line. All of these factors must be jointly considered when selecting the tile size, making it difficult for the programmer to find the optimal solution.

To ease the burden on programmers, some compilers have integrated automatic tiling support, via recent polyhedral compilation techniques. Examples of such compilers are PLuTO [40] and PolyMage [153], both from Bondhugula et al., as well as the commercial R-Stream compiler [184, 14] from Reservoir Labs. Such compilers perform affine loop transformations that increase data locality and memory throughput in cached architectures. Darte et al. proved in [58] that polyhedral compilation techniques can produce asymptotically optimal schedules for cases with uniform dependencies. Their optimality is therefore not guaranteed for system schedules with low iteration (or tile) counts. While a high number of iterations is a plausible hypothesis for High Performance Computing (HPC) applications, this is less so for real-time embedded vision systems that often work with fairly low resolution images. Furthermore, polyhedral compilation cannot integrate non-linear performance models, which as the experiments from Section 4.7.5 show, are necessary to obtain a higher accuracy. Other methods use tiling to obtain a coarser scheduling granularity in parallel systems [111, 152]. None of these methods, however, is able to model explicit data transfers and scheduling dependencies with non-linear characteristics.

In a recent work by Saïdi et al. [177], the authors propose an analytical model for optimizing the tiling of array processing algorithms in systems with explicitly managed memory and define the analogous constrained optimization problem. However, their work relies on a linear regression model of the DMA data transfer times that, as shown in section 4.7.5, can yield poor results in some architectures. Furthermore, this method assumes tile dimensions are integer divisors of the image dimensions, a constraint I believe is too limiting and that more optimal solutions might be found if this restriction was lifted. Removing this restriction, however, requires careful modeling of the remaining tiles at the last column and on the last row.

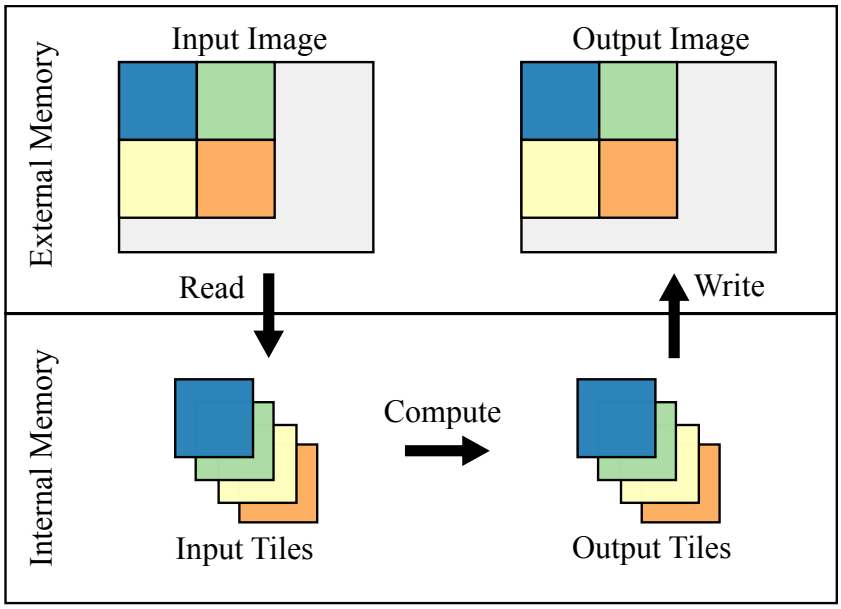


Figure 6.1: *Overview of the tiling pipeline concept as presented by Ievgen in [91]. It consists in subdividing the image in a series of tiles, each of which can is (i) read from the external memory, (ii) processed locally, and the results (iii) written back to the external memory to recompose an output image.*

This chapter describes a method for selecting the tiling parameters that minimizes the execution time of a tiled parallel application kernel on an embedded multiprocessor. First, a new accurate analytic model of the execution time is presented. The model presented here differs from the work of Saïdi et al. [177] in that it targets an embedded multiprocessor platform with a centralized DMA. It further integrates more accurate non-linear models of the DMA transfer times and can handle any valid tile sizes. This model is then translated into a constraint optimization problem and implemented in the Tilana tool using the Choco3 [200] constraint optimization framework.

The *contributions* in this chapter are:

1. provide a new analytical model to estimate the execution time of an application kernel that implements tiling, for any valid tile dimensions;
2. integrate a non-linear DMA performance model, described in section 4.7.5, for higher accuracy in DMA timing estimations;
3. derive and integrate a model of the parallel overheads and of the tile scheduling dependencies in a multiprocessor platform;
4. define and implement a constrained optimization model based on non-linear constraints that allows to determine the optimal tile dimensions that minimize the execution time of the parallel application kernel.

The remainder of this chapter is organized as follows. Section 6.2 states the problem and section 6.3 provides the preliminary definitions. Section 6.4 details the step-by-step construction of the execution time model for an application kernel that implements tiling. This initial model is gradually extended to integrate tile scheduling constraints and overheads. Section 6.5 presents how the analytical model is translated into a constrained optimization problem and implemented in Tilana using the Choco3 [200] constraint optimization framework. Section 6.6 presents the experiments done and the results obtained. A discussion ensues on how the model can be used to analyze the impact of tiling dimensions on the performance in several cases. Finally, section 6.7 summarizes and concludes this chapter.

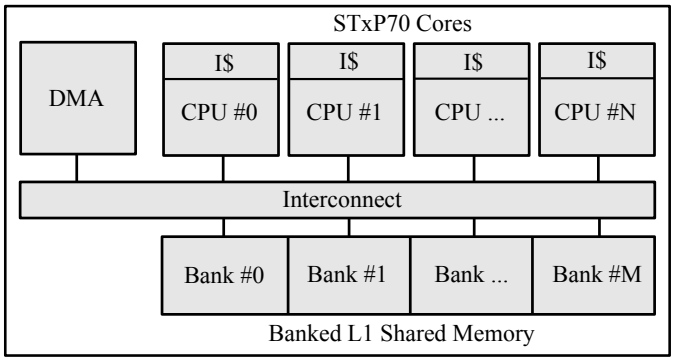


Figure 6.2: *Architectural template of the STxP70 ASMP.*

6.2 Problem Definition

The problem addressed in this chapter is that of finding optimal tile dimensions that minimize the execution time of an application kernel on an embedded multiprocessor platform with explicitly managed memory. In this context, an application kernel is an user-defined code section or function that takes a given set of input data, processes it, and returns a set of output data based on the input data. It is assumed that a kernel is a non data-dependent point or local algorithm that processes each point of the image iteratively. Furthermore, it is assumed that kernel iterations are independent, that is, the computation of a given kernel iteration does not depend on the results of any other iteration of the same kernel, and can therefore be executed in parallel.

6.2.1 Target Multiprocessor Architecture and Programming Model

The work developed in this chapter focuses on embedded multiprocessor platforms with explicitly managed memory. Figure 6.2 depicts the target architectural template. It is an homogeneous multiprocessor architecture with a number of independent *Multiple Instruction Multiple Data* (MIMD) processing elements. There exist at least two memory zones: an internal shared memory with lower access latency but a limited capacity; and an external memory with higher capacity but also increased access latency. Processing elements have access to both internal and external memory zones. The internal memory is a multi-banked memory which allows simultaneous access to independent banks and access conflicts are assumed to be negligible. A centralized DMA handles data transfers between the internal and external memories. It has independent read and write channels that handles data transfer requests sequentially, in arrival order. The term *read* designates a transfer from the external memory to the internal memory. Conversely, the term *write* designates a transfer from the internal memory to the external memory.

The target parallel programming model is OpenMP [159]. Application kernels are assumed to be parallelized via OpenMP data-parallel constructs, such as OpenMP parallel for loops. The OpenMP parameters are acquired by the characterization process described in section 4.7.3.

More particularly, the platform modeled in this work is STMicroelectronics's STxP70 ASMP, detailed in section 2.2.3. Nonetheless, this work is not limited to this particular platform, and can model any embedded multiprocessor platform matching the aforementioned architectural criteria. The platform's memory access times and DMA data transfer timing model are obtained via the characterization process described in sections 4.7.4 and 4.7.5, respectively.

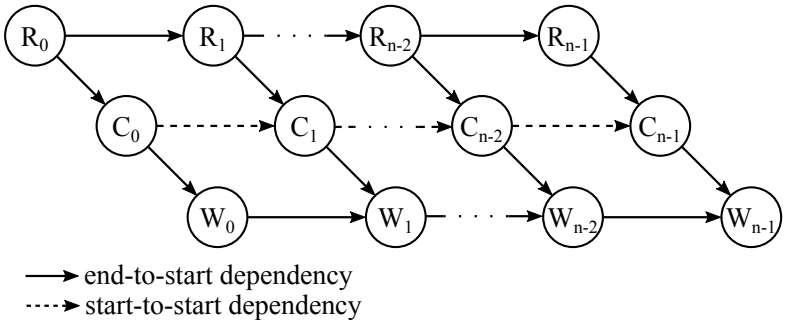


Figure 6.3: *Dependency graph between software pipeline stages for reading a tile into the local memory (R), computing the tile (C), and writing back the results (W).*

6.2.2 Application Kernel Tiling as a Software Pipeline

The analytical model assumes an application kernel's input and output data are located in the external memory. Processing these data directly in the external memory is not a viable option as the access latency can be orders of magnitude higher than that of the local memory. It is thus necessary to first move the input data to the local memory, process it locally, and then copy the results back to the output buffers in the external memory. This results in three software stages: *Read* (*R*), *Compute* (*C*) and *Write* (*W*). Performing these three stages sequentially on the entirety of the data, however, would not be efficient, as it would lead to an underutilization of the processing elements and of the DMA read and write channels.

A more efficient solution is to use tiling so as to subdivide the data into smaller chunks and to implement a software pipeline where the three aforementioned stages can be executed in parallel. This solution achieves a higher utilization of the available resources and reduces the execution time of the application kernel. Figure 6.3 shows the dependency relations between the three software pipeline stages when using the tiling software pipeline solution to implement the application kernel. Note that in a multiprocessor platform, compute stages can execute in parallel, thus only their starting order is defined. As the target platform possesses a single centralized dual-channel DMA, only one read and one write operation can execute at any given time, hence the end-to-start relations between subsequent read and write nodes.

6.2.3 Finding the Optimal Tiling Parameters

The tile dimensions can have a deep impact in the final application kernel execution time and need to be carefully selected. Depending on the kernel and target platform characteristics, the execution time can be dominated by either one of the three pipeline stages. Such tiling parameters must be adjusted for each combination of application kernel and multiprocessor platform. The optimization of tile dimensions is addressed by defining a constrained optimization problem with the goal of determining the width and height of the main tile type $\mathcal{T}^0$ that minimizes the execution time of the application kernel on the target platform.

Figure 6.4 shows three scheduling cases for four tiles of type $\mathcal{T}^i$, assuming a single computing resource. In each case the execution time is dominated by a particular pipeline stage. The execution time of the read and write stages are computed using the DMA performance model described in section 4.7.5.3. The read stage time $R(\mathcal{T}^i)$ is the time to perform a $2D$ DMA data transfer of the dimensions of tile $\mathcal{T}^i$ from the STxP70 ASMP's L3 or L2 memory to the L1 memory. Conversely, the write stage time $W(\mathcal{T}^i)$ is the time to perform a $2D$ DMA data transfer of the dimensions of tile $\mathcal{T}^i$ from the STxP70 ASMP's L1 memory to the L2 or L3 memory. The execution time of the compute stage $C(\mathcal{T}^i)$ is a function of the $\mathcal{T}^i$ tile dimensions and of the time

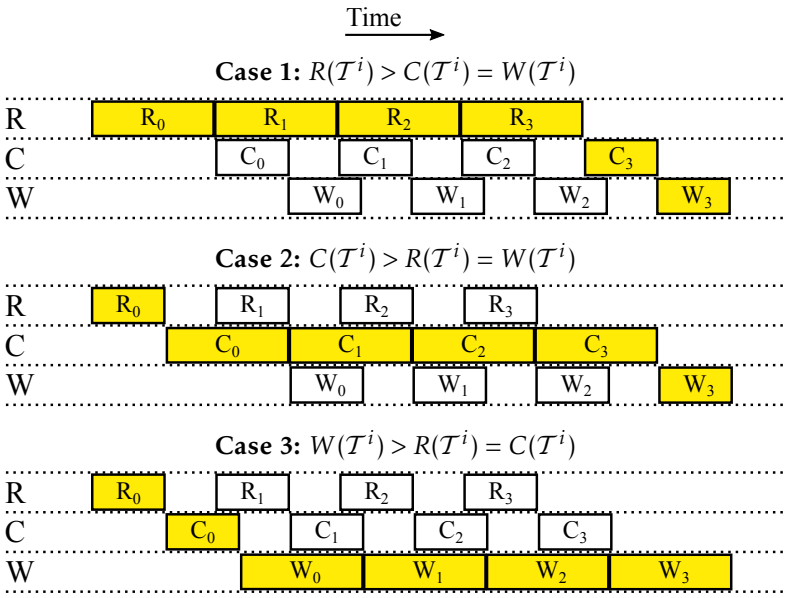


Figure 6.4: *Example of a scheduling for four tiles of type $\mathcal{T}^i$, assuming a single compute resource. Three different cases are shown, where the time for R, C and W, respectively, dominate the other two. Notice how the highlighted critical path changes in each case. In cases 1 and 3, the system is memory-bound, while in case 2, the system is compute-bound.*

to process the application kernel on a single point of the image, which is an input parameter of the model and can be obtained by profiling the application. In the following sections, the details of the application's execution time analytical model are presented.

6.3 Preliminaries

6.3.1 Input Parameters

The problem of finding optimal tile dimensions is addressed in this thesis by defining a constrained optimization problem with the goal of minimizing the execution time $t_{exe}(\mathbf{K}, \mathbf{I})$ of the application kernel **K** on an input image **I**. A kernel **K** represents the data footprint needed to compute a single output pixel.

Table 6.1 lists the complete set of input parameters for the tiling optimization, which consist in: the input image width ($w_{\mathcal{I}}$), height ($h_{\mathcal{I}}$) and data type size ($ds_{\mathcal{I}}$); kernel width ($w_{\mathcal{K}}$), height ($h_{\mathcal{K}}$) and kernel execution time per iteration ($t_{\mathcal{K}}$); and the tiling memory size (m_s).

Parameter	Description
$w_{\mathcal{I}}$, $h_{\mathcal{I}}$, $ds_{\mathcal{I}}$	Image width, height and data type size
$w_{\mathcal{K}}$, $h_{\mathcal{K}}$, $t_{\mathcal{K}}$	Kernel width, height and iteration execution time
m_s	Memory size for tiling buffers

Table 6.1: *List of input parameters for tiling optimization.*

6.3.2 Iteration and data footprint spaces

A distinction is made between the loop iteration space variables and data footprint space variables. The former are used to count the number of loop iterations in a tile, where each loop iteration processes a single point in the image. The latter are used to compute the dimensions of the input data window necessary to process the tile. As an image processing kernel needs surrounding data to process a single image point, the tile data footprint might exceed its dimensions in loop iteration space. The convention established herein is that the $2D$ loop iteration space for a tile is represented as an ordered pair (x, y), while data footprint is denoted by the ordered pair (w, h).

$$\text{is}(\mathcal{K})=(x_\mathcal{K}, y_\mathcal{K}) =(1,1) \tag{6.1}$$

$$\text{df}(\mathcal{K})=(w_\mathcal{K}, h_\mathcal{K}) \tag{6.2}$$

$$\text{is}(\mathcal{I})=(x_\mathcal{I}, y_\mathcal{I}) \tag{6.3}$$

$$\text{df}(\mathcal{I})=(w_\mathcal{I}, h_\mathcal{I}) \tag{6.4}$$

$$\tag{6.5}$$

where is(obj) is a function that retrieves the size of a given object obj in iteration space, and df is a function that retrieves the size of the parameter object in data footprint space. A kernel's iteration space is a single point, while its data footprint corresponds to the dimension of the input data necessary to compute said point. Variables $w_\mathcal{K}$, $h_\mathcal{K}$, $w_\mathcal{I}$ and $h_\mathcal{I}$ are input parameters, and the $x_\mathcal{I}$ and $y_\mathcal{I}$ values are computed as:

$$x_\mathcal{I}=w_\mathcal{I} - w_\mathcal{K} + 1 \tag{6.6}$$

$$y_\mathcal{I}=h_\mathcal{I} - h_\mathcal{K} + 1 \tag{6.7}$$

6.3.3 Tile types

The full iteration space of the image (is($\mathbf{I}$)) is divided into tiles of four types. Figure 6.5 depicts the four tile types: regular tiles ($\mathcal{T}^0$), tiles in the last column ($\mathcal{T}^1$), tiles in the last row ($\mathcal{T}^2$), and the last bottom right tile ($\mathcal{T}^3$). The relation between the tile dimensions and the number of tiles of each type in the image is given by:

$$x_\mathcal{I}=a \cdot x_0 + b \cdot x_1 \tag{6.8}$$

$$y_\mathcal{I}=c \cdot y_0 + d \cdot y_1 \tag{6.9}$$

$$\text{is}(\mathcal{I})=(x_\mathcal{I}, y_\mathcal{I}) \tag{6.10}$$

$$\text{is}(\mathcal{T}^0)=(x_0, y_0) \tag{6.11}$$

$$\text{is}(\mathcal{T}^1)=(x_1, y_0) \tag{6.12}$$

$$\text{is}(\mathcal{T}^2)=(x_0, y_1) \tag{6.13}$$

$$\text{is}(\mathcal{T}^3)=(x_1, y_1) \tag{6.14}$$

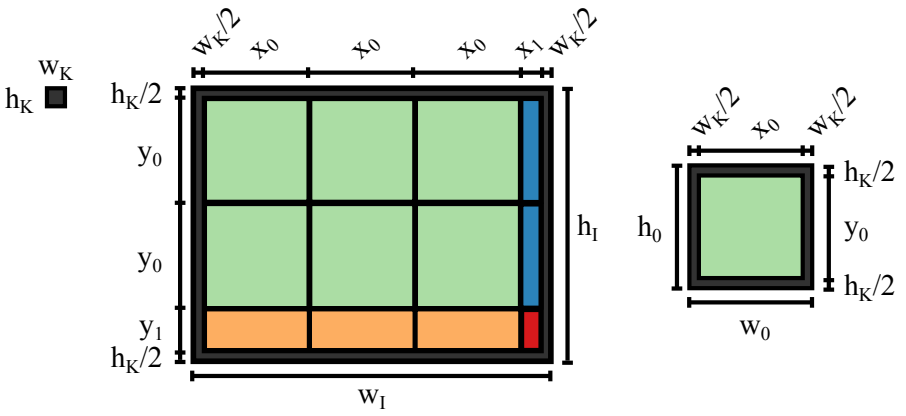


Figure 6.5: *Example of a 2D image tiling instantiation.*

An image **I** of type $\mathcal{I}$, has a set of tiles $\mathbb{T}$. The subset of all tiles of type $\mathcal{T}^i$ in **I** is denoted as $\mathbb{T}^i$. A function $\eta\ (\mathbb{T}^i)$ is defined to retrieve the number of tiles $\mathcal{T}^i$ in a set of tiles $\mathbb{T}^i$, with η_x further representing the number of tiles in the horizontal direction, and η_y the number of tiles in the vertical direction:

$$\eta(\mathbb{T}^0)=a\cdot c \quad (6.15) \qquad \eta_x(\mathbb{T}^0)=a \quad (6.19) \qquad \eta_y(\mathbb{T}^0)=c \quad (6.23)$$
$$\eta(\mathbb{T}^1)=b\cdot c \quad (6.16) \qquad \eta_x(\mathbb{T}^1)=b \quad (6.20) \qquad \eta_y(\mathbb{T}^1)=c \quad (6.24)$$
$$\eta(\mathbb{T}^2)=a\cdot d \quad (6.17) \qquad \eta_x(\mathbb{T}^2)=a \quad (6.21) \qquad \eta_y(\mathbb{T}^2)=d \quad (6.25)$$
$$\eta(\mathbb{T}^3)=b\cdot d \quad (6.18) \qquad \eta_x(\mathbb{T}^3)=b \quad (6.22) \qquad \eta_y(\mathbb{T}^3)=d \quad (6.26)$$

The data footprint for each tile type is related to the tile iteration space dimensions and the kernel data footprint by:

$$w_i=\mathcal{H}(x_i-1)\cdot(x_i+w_{\mathcal{K}}-1), \quad \forall i\in\{\mathcal{I},0,1\} \quad (6.27)$$
$$h_i=\mathcal{H}(y_i-1)\cdot(y_i+h_{\mathcal{K}}-1), \quad \forall i\in\{\mathcal{I},0,1\} \quad (6.28)$$

$$\mathrm{df}(\mathcal{I})=(w_{\mathcal{I}},h_{\mathcal{I}}) \quad (6.29)$$

$$\mathrm{df}(\mathcal{T}^0)=(w_0,h_0) \quad (6.30)$$
$$\mathrm{df}(\mathcal{T}^1)=(w_1,h_0) \quad (6.31)$$
$$\mathrm{df}(\mathcal{T}^2)=(w_0,h_1) \quad (6.32)$$
$$\mathrm{df}(\mathcal{T}^3)=(w_1,h_1) \quad (6.33)$$

where $\mathcal{H}(i)$ is the Heaviside step function, defined as:

$$\mathcal{H}(i)=\begin{cases}0 & \text{if } i<0,\\ 1 & \text{if } i\geq 0.\end{cases} \quad (6.34)$$

Area values for each tile type in loop iteration space of data footprint space – denoted by $\mathcal{A}(\text{is}(\mathcal{T}))$ and $\mathcal{A}(\text{df}(\mathcal{T}))$ – are computed by multiplying the respective width and height:

$$\mathbf{v}=(v_0, v_1) \tag{6.35}$$

$$\pi_i(\mathbf{v})=v_i \tag{6.36}$$

$$\mathcal{A}(\mathbf{v})=\prod_i \pi_i(\mathbf{v}) \tag{6.37}$$

$$\mathcal{A}(\text{is}(\mathcal{I}))=\prod_i \pi_i(\text{is}(\mathcal{I})) = x_{\mathcal{I}} \cdot y_{\mathcal{I}} \tag{6.38}$$

$$\mathcal{A}(\text{df}(\mathcal{I}))=\prod_i \pi_i(\text{df}(\mathcal{I})) = w_{\mathcal{I}} \cdot h_{\mathcal{I}} \tag{6.39}$$

Appendix C provides a summary list of the tiling parameters and mathematical definitions, each followed by their associated descriptions.

6.4 Execution Performance Model

This section details the step-by-step construction of the execution performance model for an application kernel that implements tiling. This initial model is gradually extended to integrate tile scheduling constraints and overheads. Initially, subsection 6.4.1 defines the time of the compute pipeline stage execution for a single tile, which is referred to in subsection 6.4.2 for defining the isolated computation time of a single tile. Then, subsection 6.4.3 presents the model for computing the execution time for a set of tiles of a same type in an unbounded schedule, that is, a schedule considering an unlimited number of processing elements. Subsequently, subsection 6.4.4 derives the model for computing the execution time for a set of tiles of a same type in a bounded schedule, one in which the number of processors is fixed. The combination of the bounded execution time model for all tile types in an image is given in subsection 6.4.5, considering no parallel runtime overheads. The final subsection 6.4.6 introduces the OpenMP parallel overheads, producing the complete execution performance model for an application kernel that implements tiling. This model will serve as a basis for deriving the constraint optimization problem in section 6.5.

6.4.1 Single Tile Computation Time

The analytical model assumes the computation time for one kernel execution to be constant and given by the input kernel execution time parameter (t_K). The total time for completing the execution of the compute stage is thus the sum of the kernel computation time at each iteration point in a tile of class $\mathcal{T}^i$:

$$C(\mathcal{T}^i)=t_K \cdot \mathcal{A}(\text{is}(\mathcal{T}^i)) \tag{6.40}$$

6.4.2 Single Tile Execution Time

The execution time of a single tile $t^s_{exe}(\mathcal{T}^i)$ for a tile of class $\mathcal{T}^i$, assuming no inter-tile dependencies, is a sum of the duration of the three pipeline stages R, C and W for such a tile, and is given by:

$$t^s_{exe}(\mathcal{T}^i) = R(\mathcal{T}^i) + C(\mathcal{T}^i) + W(\mathcal{T}^i) \tag{6.41}$$

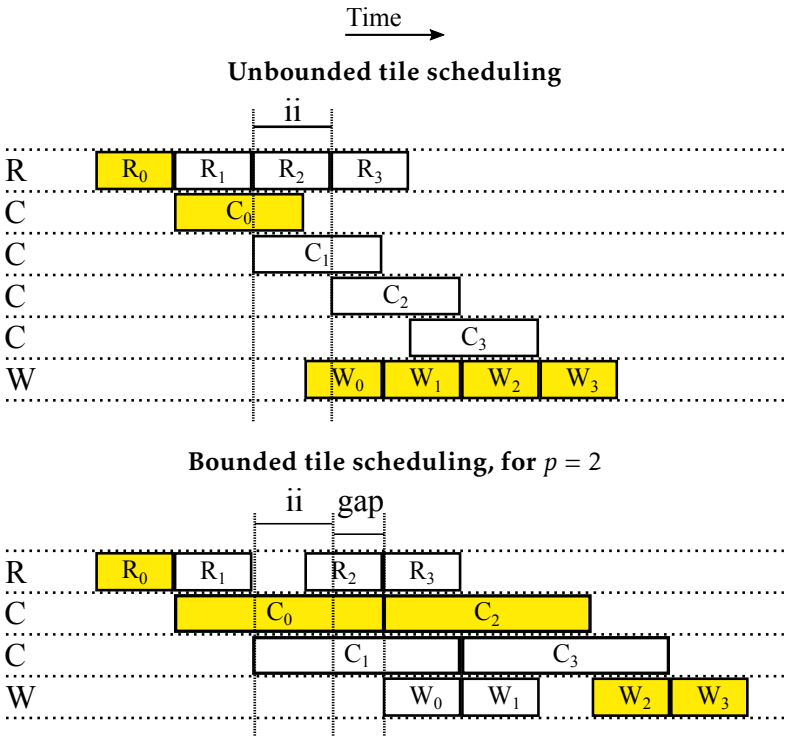


Figure 6.6: *Example of unbounded and bounded tile scheduling. From the unbounded tile schedule, which assumes one computing resource per tile, it is possible to determine the initiation interval* (ii) *– the time interval between two iterations in the steady-state.*

6.4.3 Unbounded Tile Scheduling

To compute the execution time for the schedule of all tiles $\mathbb{T}^i$ of a given class $\mathcal{T}^i$, let's assume initially an unbounded number of processors. As the DMA is centralized, it is a shared resource that can only be used to read and write one tile in each channel at a time. Figure 6.6 shows an example of one such unbounded schedule. The time interval between the start of two consecutive tiles of a same class is modeled by the initiation interval ii. This factor is simply the maximum between the read stage R and write stage W times:

$$\text{ii}(\mathcal{T}^i) = \max(R(\mathcal{T}^i), W(\mathcal{T}^i)) \tag{6.42}$$

The unbounded execution time $\text{t}^u_{exe}(\mathbb{T}^i)$ for the set of tiles $\mathbb{T}^i$ of class $\mathcal{T}^i$ can then be written as:

$$\text{t}^u_{exe}(\mathbb{T}^i) = \mathcal{H}(\eta(\mathbb{T}^i) - 1) \cdot \text{t}^s_{exe}(\mathcal{T}^i) + \mathcal{R}(\eta(\mathbb{T}^i) - 1) \cdot \text{ii}(\mathcal{T}^i) \tag{6.43}$$

where $\mathcal{R}(i)$ is the ramp function:

$$\mathcal{R}(i) = \max(i, 0) \tag{6.44}$$

6.4.4 Bounding the Number of Processors

When building a schedule with a bounded number of processors p, if all p processors are busy, the next tile can only be schedule once one of the processors is done. Figure 6.6 provides an example of a bounded schedule with two computing resources. It is possible to see from the schedule that C_2 could not start by the next initiation interval since all computing resources were busy, and had to be delayed until C_0 was completed. The resulting time shift induced by these dependencies can be thought of as gaps in the schedule, and are modeled by a gap factor. As C_3 is already one *initiation interval* after C_2, its start time coincides with the end time of C_1, resulting in no additional *gaps*.

In such a bounded schedule, two cases might arise. If the duration of the compute stage $C(\mathcal{T}^i)$ for a given tile class $\mathcal{T}^i$ is inferior or equal to its initiation interval $\text{ii}(\mathcal{T}^i)$, the computation is memory-bound and increasing the number of processing elements should bring no benefits. In this case, the initiation interval alone will throttle the execution of subsequent iterations and the gap factor will be zero. If, however, the compute stage's duration $C(\mathcal{T}^i)$ is superior to the initiation interval $\text{ii}(\mathcal{T}^i)$, the schedule of loop iterations might present dependencies with previous iterations. Such dependencies are modeled by the gap factor $\text{gap}(\mathcal{T}, p)$, a function of a given tile class $\mathcal{T}^i$ and the number of processors p, defined as follows:

$$\text{gap}(\mathcal{T}^i, p) = \mathcal{R}(C(\mathcal{T}^i) - \text{ii}(\mathcal{T}^i) \cdot \mathcal{R}(p-1)) \tag{6.45}$$

The number of times a gap will occur on the schedule, during the execution of a set of tiles $\mathbb{T}^i$ of class $\mathcal{T}^i$ on p processors is given by $\eta_{\text{gap}}(\mathbb{T}^i, p)$ as follows:

$$\eta_{\text{gap}}(\mathbb{T}^i, p) = \left\lfloor \frac{\mathcal{R}(\eta(\mathbb{T}^i) - 1)}{p} \right\rfloor \tag{6.46}$$

Reintroducing these two factors in the expression of the unbounded tile execution time $\text{t}^u_{exe}(\mathbb{T}^i)$ from Eq. (6.43) leads to the following expression for the bounded tile execution time $\text{t}^b_{exe}(\mathbb{T}^i, p)$:

$$\text{t}^b_{exe}(\mathbb{T}^i, p) = \mathcal{H}(\eta(\mathbb{T}^i) - 1) \cdot \text{t}^s_{exe}(\mathcal{T}^i) + \mathcal{R}(\eta(\mathbb{T}^i) - 1) \cdot \text{ii}(\mathcal{T}^i) + \eta_{\text{gap}}(\mathbb{T}^i, p) \cdot \text{gap}(\mathcal{T}^i) \tag{6.47}$$

6.4.5 Combining All Tile Types

Computing the execution time for the entire image $\text{t}^{all}_{exe}(\mathcal{I})$ thus requires the combination of the execution times of all of the image tiles. This model assumes a grouped execution of all image tiles of a same type $\mathcal{T}^i$, therefore belonging to subset $\mathbb{T}^i$. Therefore, the first tile of a class $\mathcal{T}^i$ in $\mathbb{T}^i$ will only be scheduled after the last tile of the previous class $\mathcal{T}^{i-1}$ in $\mathbb{T}^{i-1}$. Thus, $\text{t}^{all}_{exe}(\mathcal{I})$ is computed by summing the bounded execution time $\text{t}^b_{exe}(\mathbb{T}^i)$ from Eq. (6.43) for all tile types.

The implementation of such a scheduling order is possible via the definition of a mapping function from a linear iteration index, to the location of the corresponding tile in the image, as per the scheduling order established herein. Such a mapping function would be needed in any case, unless the parallel $2D$ loop is collapsed via OpenMP directives, in which case the compiler generates the necessary mapping functions. Defining this particular iteration order, however, has the added benefit of grouping tiles with similar execution times together for improved load balance. Furthermore, it enables the construction of a simpler accurate analytical model, than would otherwise be possible if a simple raster scan order was used.

When assembling the combined scheduling of tiles of different types, two further aspects need to be modeled. The first is the starting processor in which the first tile of a type is allocated,

which might impact the number of gaps observed for a given tile type. The second is the overlapping of the last tile of a type with the first tile of the following type in the schedule.

To compose the execution times of all tile types, first the end time in the schedule $\mathrm{t}^{all}_{end}(\mathbb{T}^i, p)$ is defined for a set of tiles $\mathbb{T}^i$. This end time is defined in relation to the start time $\mathrm{t}^{all}_{start}(\mathbb{T}^i, p)$ and to the execution time $\mathrm{t}^{all}_{exe}(\mathbb{T}^i, p)$ as follows:

$$\mathrm{t}^{all}_{end}(\mathbb{T}^i, p) = \mathrm{t}^{all}_{start}(\mathbb{T}^i, p) + \mathrm{t}^{all}_{exe}(\mathbb{T}^i, p) \tag{6.48}$$

Addressing the first point requires taking into account the processor in which the first tile of a given tile type will execute. A gap will only occur when trying to schedule a new tile when all of the processors are already busy. Changing the starting processor might thus impact the number of gaps in the schedule of tiles in a given tile set $\mathbb{T}^i$. A gap offset factor gap_{off} is thus added to the expression of the number of gaps – $\eta_{\mathrm{gap}}(\mathbb{T}^i, p)$ from Eq. (6.46) – for tiles in a given tile set $\mathbb{T}^i$ with i larger than zero[1], what yields:

$$\mathrm{gap}_{off}(\mathbb{T}^i, p) = \begin{cases} \left(\sum_{j=0\ldots i} \eta(\mathbb{T}^j) \right) \bmod p & \text{if } i > 0, \\ 0 & \text{if } i \leq 0. \end{cases} \tag{6.49}$$

$$\eta^{all}_{\mathrm{gap}}(\mathbb{T}^i, p) = \left\lfloor \frac{\mathcal{R}(\eta(\mathbb{T}^i) - 1) + \mathrm{gap}_{off}(\mathbb{T}^{i-1}, p)}{p} \right\rfloor \tag{6.50}$$

The expression for the bounded tile execution time $\mathrm{t}^{b}_{exe}(\mathbb{T}^i, p)$ in Eq. (6.47) can then be rewritten to consider the modified expression for the number of gaps in Eq. (6.50) as follows:

$$\mathrm{t}^{all}_{exe}(\mathbb{T}^i, p) = \mathcal{H}(\eta(\mathbb{T}^i) - 1) \cdot \mathrm{t}^{s}_{exe}(\mathcal{T}^i) + \mathcal{R}(\eta(\mathbb{T}^i) - 1) \cdot \mathrm{ii}(\mathcal{T}^i) + \eta^{all}_{\mathrm{gap}}(\mathbb{T}^i, p) \cdot \mathrm{gap}(\mathcal{T}^i) \tag{6.51}$$

The second point relative to the overlapping of the last tile of a type with the first tile of the following type in the schedule can be more easily addressed by separating the problem in two. First, the earliest possible start time $\mathrm{t}^{all}_{earliest}(\mathbb{T}^i, p)$ for tiles in $\mathbb{T}^i$ is computed. For this, it is assumed the first tile's write stage starts as soon as the last tile of the previous type in $\mathbb{T}^{i-1}$ has completed the execution of its write stage:

$$\mathrm{t}^{all}_{earliest}(\mathbb{T}^0, p) = \mathrm{t}^{all}_{start}(\mathbb{T}^0, p) \tag{6.52}$$

$$\mathrm{t}^{all}_{earliest}(\mathbb{T}^i, p) = \mathrm{t}^{all}_{end}(\mathbb{T}^{i-1}, p) - \mathcal{H}(\eta(\mathbb{T}^i) - 1) \cdot [R(\mathcal{T}^i) + C(\mathcal{T}^i)] \qquad , \forall i \neq 0 \tag{6.53}$$

Second, the time $\mathrm{t}^{all}_{next}(\mathbb{T}^i, p)$ at which one next tile of class $\mathcal{T}^i$ could be scheduled is computed, supposing $\mathbb{T}^i$ contained one more tile:

$$\mathrm{t}^{all}_{next}(\mathbb{T}^i, p) = \mathcal{H}(\eta(\mathbb{T}^i) - 1) \cdot [\mathrm{t}_{end}(\mathbb{T}^i, p) - C(\mathcal{T}^i) - W(\mathcal{T}^i) + \mathcal{Z}(\mathrm{gap}_{off}(\mathbb{T}^i, p)) \cdot \mathrm{gap}(\mathbb{T}^i, p)] \tag{6.54}$$

where $\mathcal{Z}$ is an indicator function of the value zero defined as:

$$\mathcal{Z}(i) = \begin{cases} 1 & \text{if } i = 0, \\ 0 & \text{if } i \neq 0. \end{cases} \tag{6.55}$$

[1] The gap offset factor gap_{off} is always zero for the first set of tiles $\mathbb{T}^0$.

Finally, the starting time $t^{all}_{start}(\mathbb{T}^i, p)$ for the first tile in $\mathbb{T}^i$ is the latest time among the earliest time a tile of the same type could start $t^{all}_{earliest}(\mathbb{T}^i, p)$ and, $t^{all}_{next}(\mathcal{T}^j, p)$, the time the next tile in $\mathbb{T}^j$ would start, for all previous tile sets $\mathbb{T}^j$ where $j \in \{0 \dots i-1\}$:

$$t^{all}_{start}(\mathbb{T}^i, p) = \max(t^{all}_{earliest}(\mathbb{T}^i, p), \max_{j=0\dots i-1}(t^{all}_{next}(\mathbb{T}^j, p)) \tag{6.56}$$

Assuming no parallelization overheads, the timings for the entire image are related to the individual tile type times by:

$$t^{all}_{start}(\mathcal{I}, p) = t^{all}_{start}(\mathbb{T}^0, p) \tag{6.57}$$

$$t^{all}_{end}(\mathcal{I}, p) = t^{all}_{end}(\mathbb{T}^3, p) \tag{6.58}$$

$$\begin{aligned} t^{all}_{exe}(\mathcal{I}, p) &= t^{all}_{end}(\mathcal{I}, p) - t^{all}_{start}(\mathcal{I}, p) \\ &= t^{all}_{end}(\mathbb{T}^3, p) - t^{all}_{start}(\mathbb{T}^0, p) \end{aligned} \tag{6.59}$$

6.4.6 Introducing Parallel Scheduling Overheads

The OpenMP parallel runtime introduces a number of scheduling overheads which need to be accounted for in the model. Such overheads are characterized using the methodology described in section 4.7.3 of this thesis. It is assumed the application kernel loops are parallelized with an OpenMP *parallel for* directive, with a static schedule, over all available processors. The scheduling overheads needed to account for the OpenMP parallel for loop are collapsed into the following three factors: OP_{open}, the overhead to open an OpenMP parallel region; OP_{sched}, the overhead at each OpenMP tile loop iteration; and OP_{close}, the overhead to close the parallel region. Therefore, this section redefines some of the expressions from previous sections to include these overheads.

Parallel regions open and close overheads occur upon entering and exiting the application kernel's processing loop. To model these overheads, two new variables are defined, namely: $t^{par}_{start}(\mathcal{I})$ and $t^{par}_{end}(\mathcal{I})$. These two variables are used to add such overheads around the start and end times of the image on the original bounded schedule, $t^{b}_{start}(\mathcal{I})$ and $t^{b}_{end}(\mathcal{I})$ from Eq. (6.57) and (6.58), as follows:

$$t^{par}_{start}(\mathcal{I}) = t^{par}_{start}(\mathbb{T}^0) - OP_{open} \tag{6.60}$$

$$t^{par}_{end}(\mathcal{I}) = t^{b}_{end}(\mathbb{T}^3) + OP_{close} \tag{6.61}$$

The next step is to model the overhead of a each OpenMP tile loop iteration. To achieve this, first the value of OP_{sched} is added to the execution time of the first tile of any given tile class $t^{s}_{exe}(\mathcal{T}^i)$ from Eq. (6.41). Then, the OP_{sched} value is added to the initiation interval for subsequent tiles $ii(\mathcal{T}^i)$ from Eq. (6.42) as well. The new expressions for the $t^{s,par}_{exe}$ and ii^{par} then become:

$$ii^{par}(\mathcal{T}^i) = ii(\mathcal{T}^i) + OP_{sched} \tag{6.62}$$

$$t^{s,par}_{exe}(\mathcal{T}^i) = t^{s}_{exe}(\mathcal{T}^i) + OP_{sched} \tag{6.63}$$

Finally, the execution time of the application kernel on an image, including OpenMP overheads, can be expressed as:

$$\text{gap}^{par}(\mathcal{T}^i,p)=\mathcal{R}(C(\mathcal{T}^i)-\text{ii}^{par}(\mathcal{T}^i)\cdot\mathcal{R}(p-1)) \tag{6.64}$$

$$\text{t}^{par}_{earliest}(\mathbb{T}^0,p)=\text{t}^{par}_{start}(\mathbb{T}^0,p) \tag{6.65}$$

$$\text{t}^{par}_{earliest}(\mathbb{T}^i,p)=\text{t}^{par}_{end}(\mathbb{T}^{i-1},p)-\mathcal{H}(\eta(\mathbb{T}^i)-1)\cdot[R(\mathcal{T}^i)+C(\mathcal{T}^i)] \quad ,\forall i\neq 0 \tag{6.66}$$

$$\begin{aligned}\text{t}^{par}_{next}(\mathbb{T}^i,p)=&\mathcal{H}(\eta(\mathbb{T}^i)-1)\\&\cdot[\text{t}^{par}_{end}(\mathbb{T}^i,p)-C(\mathcal{T}^i)-W(\mathcal{T}^i)+\mathcal{Z}(\text{gap}_{off}(\mathbb{T}^i,p))\cdot\text{gap}^{par}(\mathbb{T}^i,p)]\end{aligned} \tag{6.67}$$

$$\text{t}^{par}_{start}(\mathbb{T}^i,p)=\max(\text{t}^{par}_{earliest}(\mathbb{T}^i,p),\max_{j=0\ldots i-1}(\text{t}^{par}_{next}(\mathbb{T}^j,p)) \tag{6.68}$$

$$\text{t}^{par}_{exe}(\mathbb{T}^i,p)=\mathcal{H}(\eta(\mathbb{T}^i)-1)\cdot\text{t}^{s,par}_{exe}(\mathcal{T}^i)+\mathcal{R}(\eta(\mathbb{T}^i)-1)\cdot\text{ii}^{par}(\mathcal{T}^i)+\eta_{\text{gap}}(\mathbb{T}^i,p)\cdot\text{gap}^{par}(\mathcal{T}^i) \tag{6.69}$$

$$\text{t}^{par}_{end}(\mathbb{T}^i,p)=\text{t}^{par}_{start}(\mathbb{T}^i,p)+\text{t}^{par}_{exe}(\mathbb{T}^i,p) \tag{6.70}$$

$$\begin{aligned}\text{t}^{par}_{exe}(\mathcal{I},p)=&\text{t}^{par}_{end}(\mathcal{I},p)-\text{t}^{par}_{start}(\mathcal{I},p)\\=&\text{t}^{par}_{end}(\mathbb{T}^3,p)-\text{t}^{par}_{start}(\mathbb{T}^0,p)+OP_{open}+OP_{close}\end{aligned} \tag{6.71}$$

This model presented in this section is able to determine the execution time of the application kernel tiling for a given image and number of processors. The performance model also indirectly models the impact of changing tile dimensions via the definition of the tile size parameters previously established in section 6.3. The tile sizes will impact the durations of the three pipeline stages, as well as the number of tiles of each class, which are directly used in the model defined in this section. This model is the base for the constraint optimization problem presented in section 6.5.

6.5 Constraint Optimization

6.5.1 Objectives

The execution model described in Section 6.4 allows to compute the execution time given a set of tiling, application and platform parameters. However, in order to determine the values of the input parameters that minimize the execution time, an optimization problem must be defined. This section presents the transformation of the analytical execution performance model into a constrained optimization problem. Figure 6.7 depicts the constraint optimization flow and the relationships between its components and the execution model. This constrained optimization problem is then processed by a constraint optimization solver, which is able to determine the optimal solution(s) that satisfies a given minimization condition.

The analytical model for the application kernel's execution time has been implemented in the Choco3 [200] *Constraint Programming* (CP) framework. A CP solver performs a search over the domain of possible values for the variables that satisfy all of the given constraints. Several advanced techniques are used in the solver engine to optimize the search strategy, and thus enable it to handle large search spaces [166]. This is the main advantage of relying on a constraint solver to find a solution, rather than exploring the entire search space and, for each

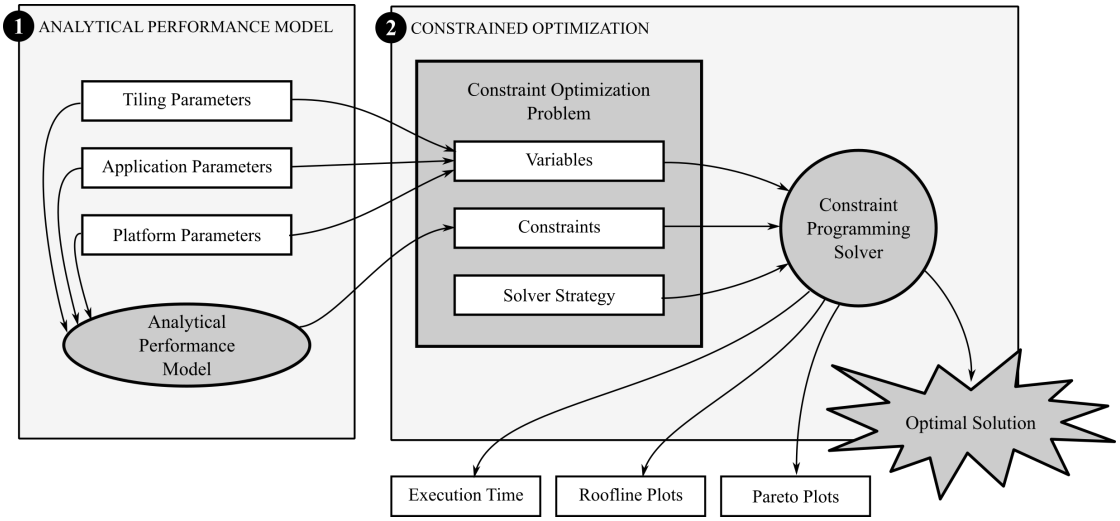


Figure 6.7: *Constrained Optimization Flow*

point, computing the output variables using the analytical model. Furthermore, and contrarily to an *Linear Programming* (LP) solver, a CP solver is capable of handling non-linear constraints, which enables its usage for the model described in this work.

The CP solver is used to, given a min-max range of values for the input parameters in Table 6.1, find the tile size $w_{\mathcal{T}}$ and $h_{\mathcal{T}}$ that minimize the parallel execution time of the kernel computation over the entire image on p processors, $\mathrm{t}_{exe}^{par}(\mathcal{I}, p)$ from Eq. (6.71).

6.5.2 Constrained Optimization Problem Definition

The implementation of the analytical model consists in defining boolean or integer variables in the CP solver for every analytical model variable and determining their domain bounds [165]. Then, the arithmetic and logical expressions of the analytical model are implemented as constraints on the previously defined CP variables. When defining the CP variables, care must be taken when initializing the domain ranges for the bounded integer variables. In this implementation, the integer variable's domain ranges are determined programmatically, from the values of the input parameters, upon initialization of a new solver instance.

A particular issue encountered, was that of multiplying inversely proportional variables, such as $\mathcal{A}(\mathrm{df}(\mathcal{T}^0]))$ and $\eta(\mathcal{T}^0)$. As the tile size increases, the number of tiles decreases, so that the multiplication of the two is always less than the image size $\mathcal{A}(\mathrm{df}(\mathcal{I}))$. The regular integer *times* constraint in the CP solver updates the domain of the product naively by multiplying the lower bounds and upper bounds of the multiplicand variables. It is often the case, especially for higher values variables, that the resulting upper bound for the product variable overflows the integer range. To avoid this problem, the solution was to implement a *times* constraint that lazily evaluates and updates the domain of the variables, only restricting the product variable's domain when portions of it are unreachable. Moreover, internal calculations are done with 64-bit *long* variables to detect and report 32-bit integer overflows.

The DMA model presented in section 4.7.5 is implemented as a new user-defined constraint between input variables representing the $2D$ data transfer size and an output variable, representing the time for the data transfer. The input variables are then bound to the tile sizes, and the output variables to their respective read and write stage time variables.

6.5.3 Constraint Propagation, Solving and Optimization

After the solver has been instantiated with all variables and constraints, two main phases take place: constraint propagation and constraint solving/optimization. The constraint propagation phase, performs a first propagation of the variable domains over the constraints with the goal of reducing variable domains to only attainable values that respect the imposed constraints. Constraint optimization is performed by first solving to find a feasible solution and, at each subsequent solver run, imposing that the new solution be strictly better than the previous one.

Notice that additional constraints can be easily added to impose new restrictions or simply to reduce the search space. For instance, subsampling the domains of the input variables to speed-up resolution can be achieved by specifying two additional constraints where the modulo of both $\mathcal{T}^0$ width and height by a subsampling coefficient are zero. The system is thus very flexible.

6.6 Results

6.6.1 Experimental Setup

The method described in this chapter was applied to a binomial filter [137] kernel, a commonly used image processing kernel, so as to find optimal tile dimensions. The binomial filter kernel was profiled to obtain the kernel timing parameter ($t_{\mathcal{K}}$) on a cycle-approximate simulator of the target platform, the STxP70v4 ASMP multiprocessor [103] described in Section 2.2.3. The performance of the said kernel is analyzed for a varying number of processors, from 1 to 8 processors, with QVGA and VGA images, and both with or without considering OpenMP overheads.

The first set of results in Figure 6.8 are presented as roofline graphs [213], providing a straightforward way to visually compare the performance of several design points. In these results, the line defined by the maximum nominal computational performance of the system and the performance limits due to the maximum system memory bandwidth define the so-called *roofline*. The roofline model thus indicates when the system is memory-bound – performance limited by the memory bandwidth – or compute-bound, where performance is limited by the computing resources. The maximum computational performance is obtained by estimating the execution time for a given image using the same analytical expressions, but considering DMA read and write transfers are instantaneous, so that $R(\mathcal{T}^i) = W(\mathcal{T}^i) = 0$ for any tile type $\mathcal{T}^i$. Similarly, to obtain the performance limits due to the memory transfers, the execution time is computed considering the computation stage is instantaneous, that is, $C(\mathcal{T}^i) = 0$.

A second set of results in Figure 6.9 compares the execution time for varying kernel execution time ($t_{\mathcal{K}}$) values. This is particularly useful to show that what-if analysis can be easily performed to evaluate the trade-offs when increasing the number of processors, versus adding instruction-set extensions or dedicated accelerators that would reduce the kernel execution time ($t_{\mathcal{K}}$) value.

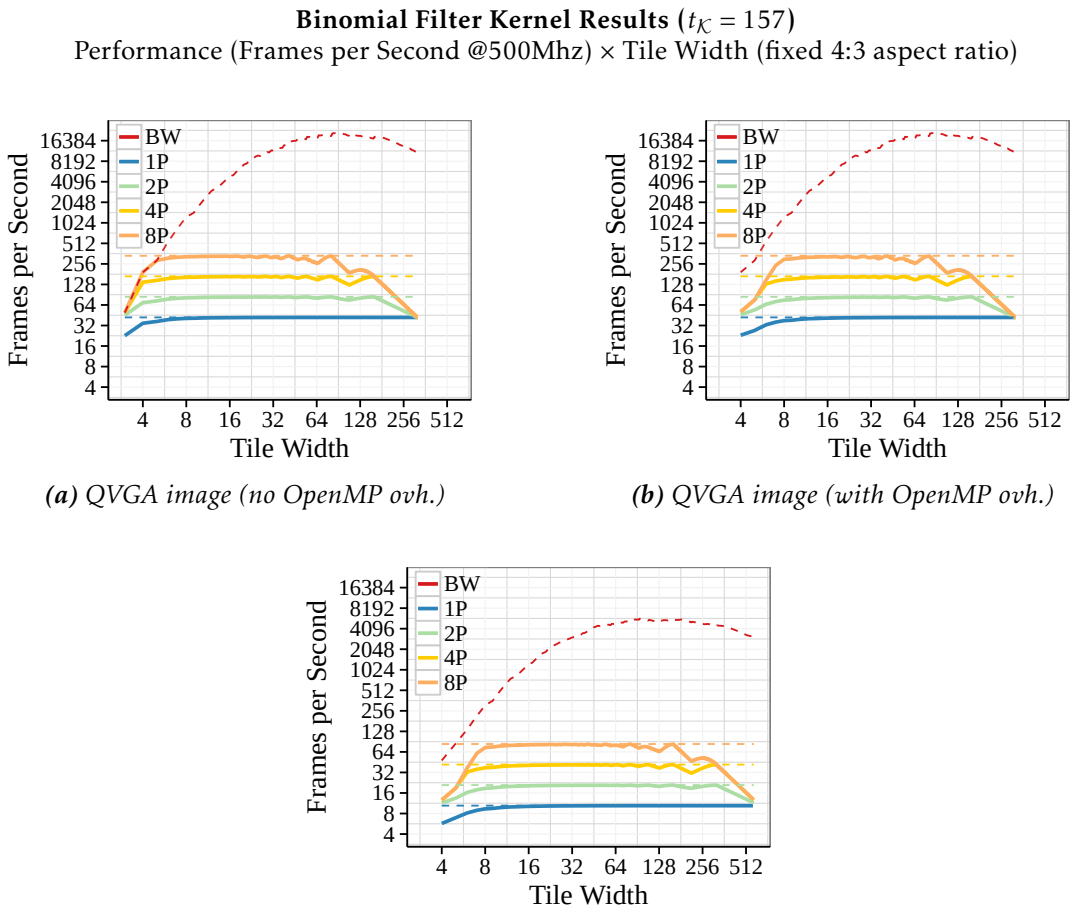
Finally, the last set of results in Figures 6.10–6.12 provide a number of Pareto graphs of the execution time versus the total data transfer amount or the tile width. From these results it is possible, for example, to select a trade-off between data transfer amount and execution time.

6.6.2 Performance Analysis

Figure 6.8 shows the roofline graphs of the Binomial Filter estimated performance in terms of frames per second versus the tile width, for (a-b)QVGA and (c)VGA images. A fixed tile aspect ratio of 4:3 is assumed, thus resulting in a cross-section of the $2D$ tile size space. From the Figure 6.8, it is possible to see that the maximum computational performance curves (shown as straight dashed lines) only cross the maximum bandwidth curve (top-most dashed curve) for small tile sizes and high processor counts. Only the 8-processor performance curve actually meets the maximum memory bandwidth, and is thus fully memory-bound. As tile size increases, the number of tiles to process decreases and load imbalance appears, generating some loss of performance that varies irregularly with the tile size, a consequence of the non-linear behavior of the system. Increasing the tile size further limits the available parallelism due to an insufficient number of tiles, up to the point where performance drops back to that of a single processor.

Overall, performance is maximized at intermediary tile sizes, with all cases from 1 to 8 processors showing good scalability and nearly reaching the nominal computational performance of the system at some ranges. However, the tile width range that maximizes the performance shifts towards :

- lower values when increasing the number of processors, or
- higher values when increasing the input image size.



(a) *QVGA image (no OpenMP ovh.)*

(b) *QVGA image (with OpenMP ovh.)*

(c) *VGA image (with OpenMP ovh.)*

Figure 6.8: *Execution time of the Binomial Filter kernel on QVGA and VGA images with and without OpenMP scheduling overheads.*

For cases (b) and (c), that take into account the OpenMP parallelization overheads, a larger impact in performance is observed for smaller tile sizes, a result of the finer parallelization granularity. Increasing the tile size, and thus coarsening the parallelization granularity, amortizes the parallelization overheads to the point where they eventually become negligible and the performance virtually matches that of case (a) with no OpenMP overheads.

6.6.3 Exploring Implementation Trade-offs

Figure 6.9 shows the execution time results for varying kernel execution times (t_K) from 1 to 8 processors. With this type of graph it is possible to perform what-if scenarios and evaluate the expected performance gains to be obtained by decreasing the kernel time via hardware accelerators and instruction-set extensions, versus increasing the number of processor cores. Comparing the performance of the 8-core configuration in (a), versus the 1-core configuration in (c), it is possible to see that the first presents better results for smaller tile sizes, while the second

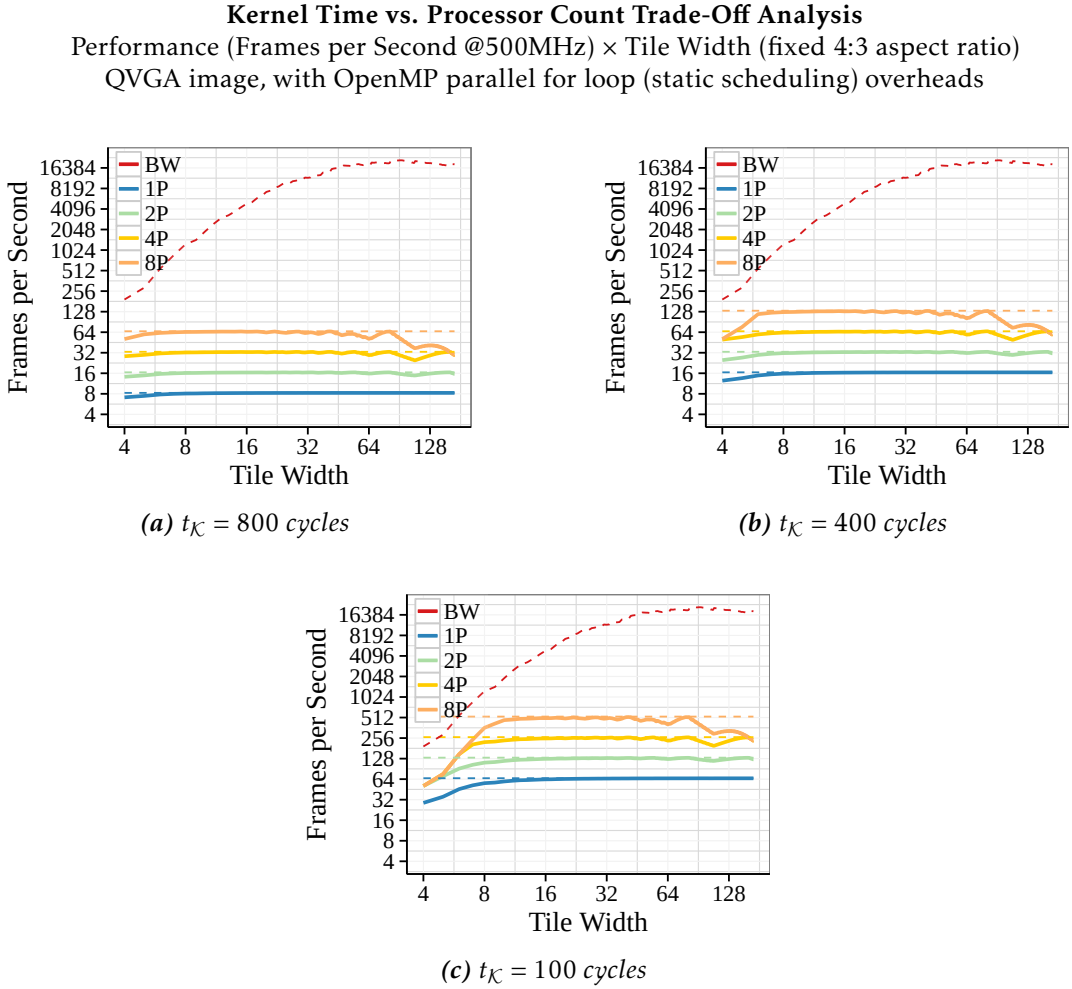
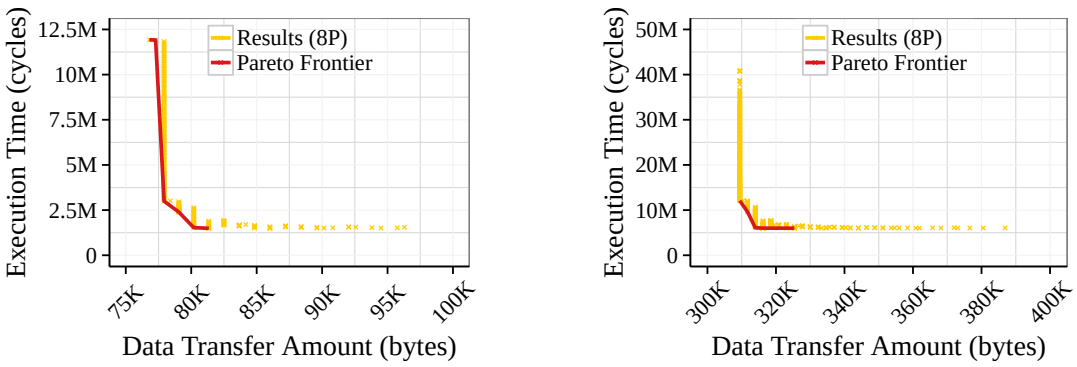


Figure 6.9: *Trade-off analysis of accelerating the execution time of a kernel (t_K) either by increasing the number of processors, or by using co-processors or dedicated accelerators to reduce the t_K value. Results shown are for processing a VGA image and take into account OpenMP static scheduling overheads.*

will present less load imbalance and be more efficient when processing larger tiles. The method described herein thus allows to select appropriate tile sizes so as to maximize performance in each case.

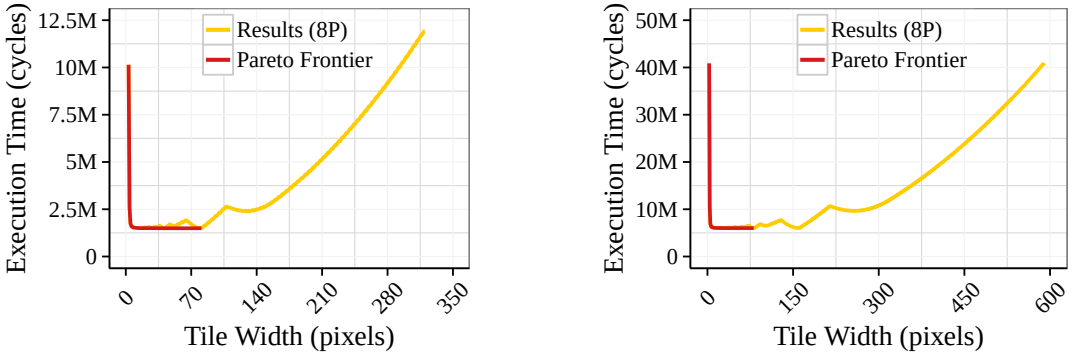
6.6.4 Execution Time versus Transferred Data or Tile Size

A trade-off exists between the amount of transferred data and the execution time. Figure 6.10 shows Pareto plots that allows the evaluation of such trade-offs for the Binomial Filter kernel. These plots show the trade-offs between total transferred data in bytes and the execution time for a single image processed by the Binomial Filter kernel in an STxP70 ASMP platform with 8 processors. Plots are shown for both QVGA images in Figure 6.10(a) and VGA images in Figure 6.10(b). Yellow crosses and lines represent the result data points, while the darker red line represents the Pareto frontier.



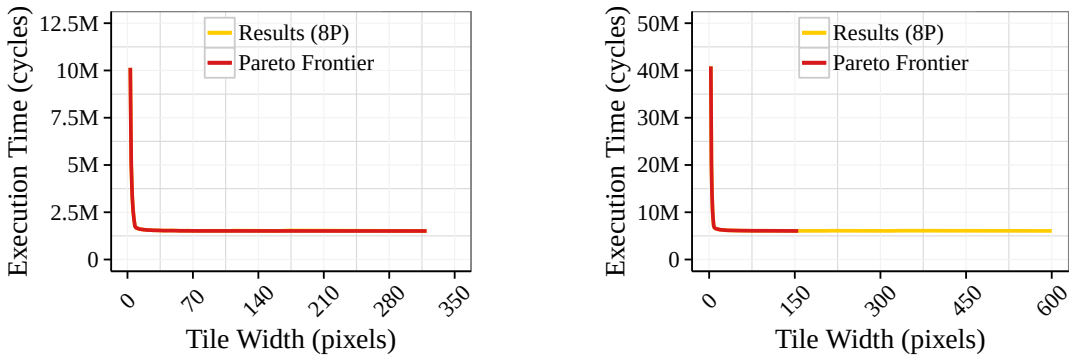
***(a)** QVGA image* ***(b)** VGA image*

***Figure 6.10:** Binomial Filter kernel's Pareto frontier for minimum Execution Time vs. Data Transfer Amount for a clock frequency of 500 Mhz.*



***(a)** QVGA image* ***(b)** VGA image*

***Figure 6.11:** Binomial Filter kernel's Pareto Frontier for minimum Execution Time vs. Tile Width for a fixed 4:3 aspect ratio and a clock frequency of 500 Mhz.*



(a) *QVGA image* ***(b)*** *VGA image*

Figure 6.12: *Binomial Filter kernel's Pareto frontier for minimum Execution Time vs. Tile Width for a free aspect ratio and a clock frequency of 500 MHz.*

It is further possible to analyze the trade-offs between execution time and the tile size. Figure 6.11 shows a Pareto plot of the execution time versus the tile width for a fixed aspect ratio of 4:3. Figure 6.12 depicts the same trade-offs obtained by releasing the aspect ratio constraint. Removing the aspect ratio constraint allows the solver to adjust the tile height so as to maintain low execution times over a larger range of tile widths.

Overall, the proposed system is very flexible and can aid the designer in exploring several scenarios. It allows designers to have a better understanding of the system behavior and to determine the optimal tile size parameters for maximizing the system performance under a given set of constraints. Unlike previous methods for determination of optimal data partitioning and tiling, such as those proposed by from Andonov et al. [27] and Saïdi et al. [177], ours is capable of accounting for the non-linear characteristics of the measured DMA data transfer times so as to better model the platform and the application kernel execution time under tiling.

6.7 Conclusion

This chapter described a method for computing optimal $2D$ image tile sizes, suitable for embedded image processing and computer vision applications. The proposed method expands on previous methods as it :

1. Provides an analytical model which can estimate the application kernel performance under tiling for any valid tiling dimensions, accounting for remaining last row/column tiles.
2. Integrates an accurate DMA performance model based on a mix of linear regression and interpolation from DMA characterization data.
3. Models the intrinsic non-linear behavior of the system in a constraint optimization problem.

It was shown how a designer might use such a system to gain insights into the behavior of the system, explore system architecture trade-offs, and finally select appropriate tile sizes that maximize the system performance. The benefits of using *Constraint Programming* (CP) over methods relying on polyhedral compilation is that CP can handle non-linear constraints and provide optimal solutions, whereas polyhedral compilers only handle linear constraints and are only guaranteed to provide asymptotically optimal solutions.

As a further work, the method described in this chapter can be integrated in a joint application and architecture design-space exploration flow. The goal being to automatically characterize the application and determine the tiling strategy that provides the best trade-off between application performance, data transfer amount and shared memory buffer sizes.

Chapter 7

Conclusions and Perspectives

Abstract

This section presents the conclusions of the thesis and the future work.

Contents

7.1 Conclusions

The objective of this thesis was to develop new methods and tools to aid in the parallelization of embedded computer vision applications on application-specific multiprocessor systems. More particularly, the work targeted STMicroelectronics multi- and many-core platforms. Initially, the STHORM many-core platform was evaluated in order to determine the main problems when porting an embedded application onto a parallel system. As the STHORM many-core platform was discontinued in 2013, the work was redirected towards STHORM's successor, the STxP70 *Application-Specific Multiprocessor* (ASMP) platform.

Several problems were identified in Chapter 2 from an initial critical case study that consisted in porting and parallelizing a face detection application on STHORM. These problems were restated in terms of design challenges – the capabilities and features that new methods and tools needed to provide to the user in order to address the identified issues. The design challenges identified and listed in Section 2.5 were:

Obtain fast and accurate application-level performance measurements. As has been discussed in Section 2.3, the STHORM tools were not able to provide application-level performance measurements for OpenCL applications, as they did not model the platform's ARM host. The introduction of the STxP70 ASMP platform and the change to the OpenMP parallel programming model dispensed the need for an external host and made the collection of application-level performance measurements possible. The method described in Chapter 4 for task trace collection and application profiling allowed to gather detailed performance measurements of the application and function level execution time via the generated function call trees profiles with low overheads. Furthermore, the parallel performance estimation method described in Chapter 5 allowed to obtain fast and accurate performance measurements for a number of different parallelization scenarios and platform configurations.

Access to detailed profiling tools and data for analyzing application hot-spots. Neither STHORM nor STxP70 ASMP had profiling tools capable of producing a function call tree profile for analysis of application hot-spots. Chapter 4 presented a trace collection framework that relied on the Trace Filter tool to process low-level instruction traces and produce high-level task traces of the application. These task traces were then used in the Parana tool to produce reports with function-level traces and function call tree profiling. This functionality is used both to allow users to analyze their applications, as to automatically collect platform and parallel runtime characterization information from microbenchmark traces.

Be able to discern and identify the factors that might limit an application's speedup. In the STHORM *Software Development Kit* (SDK) results, it was not possible to precisely determine the contribution due to some important factors such as the load imbalance time in data-parallel loops. A series of categories was defined for the Parana parallelization analysis reports in Section 5.4.2. The information contained in these reports allows the user to identify the sources of parallelization overheads and to determine their impact on the application's execution time.

Quickly perform parallel application and multiprocessor design space exploration as early as possible in the design flow. While useful to functionally validate an application and to acquire performance metrics, simulators still require a fully functional parallel version of the application code. Furthermore, each design point needs to be simulated independently, which can be time-consuming. In Chapter 5 a method for fast parallel performance estimation and *Design Space Exploration* (DSE) of application parallelization strategies and multiprocessor architectural parameters was presented. The method relied on the Parana tool, a trace-driven simulator that integrates a mechanistic model of the STxP70 ASMP's OpenMP parallel runtime to accurately estimate the parallel performance of an application. This method further allows to automatically determine the Amdahl's Law maximum theoretical speedup for that configuration and to compute the parallel efficiency of the configuration.

Analyze the trade-offs between the data transfers and computation. When partitioning the application's data-parallel loop iterations among threads, the developer needs to evaluate and determine the partitioning scheme that leads to better platform utilization and optimal execution time. A method for optimal $2D$ tiling parameter selection using constraint programming was thus presented in Chapter 6. The method uses non-linear constraints to more accurately model the DMA data transfer times and the OpenMP parallel runtime overheads. It allows the designer to minimize the execution time of an image processing application kernel that uses tiling, as well as to evaluate different trade-offs between the execution time and the amount of data transfers.

The contributions of this thesis can thus be summarized as follows:

1. Extension of the *Edinburgh Parallel Computing Center* (EPCC) OpenMP microbenchmarks with DMA and memory microbenchmarks for platform characterization and development of a trace collection framework and a characterization tool to automatically extract platform and parallel runtime characterization parameters from the microbenchmark traces.
2. A method for fast and accurate parallel performance prediction and early DSE of embedded multiprocessor architectural parameters and application parallelization strategies from sequential code;
3. A method for optimal tiling parameter selection that relies on non-linear constraints to minimize the execution time of an image processing application kernel.

7.2 Future Work

Include DMA data transfer models in Parana. The input and output data for a given computation are exchanged with external devices via the external memory, but due to the high access latency should be processed in the local shared memory. Transferring the data in and out of the local memory can often represent a bottleneck in the application. Although Tilana has support for the DMA data transfer models, this support still needs to be added for the parallel performance estimation flow in Parana. Two ways can be envisaged to achieve this:

1. In a simpler implementation, the user could simply annotate the tasks in the parallel scenario specifications with DMA input and output transfers. The timing of such transfers would then be estimated using the DMA performance model and taken into consideration by the Parana tool when performing the parallel task schedules.
2. In a more automated implementation, the data buffers to be handled would need to be instrumented in the source code, so that accesses to these buffers could be intercepted in the instruction traces by the Trace Filter tool. A bounding box geometry around the memory accesses could be added to the task traces, which would then be used by the Parana tool to estimate the DMA data movement costs when generating the schedule.

Integrate Tilana 2D tiling optimization with Parana. Parana can profile an application to extract a function call tree. With the help of instrumentation, it could automatically extract the necessary input parameters to the tiling analysis in Tilana. It could then, upon user request via specific directives in the parallel scenario specs, invoke the Tilana tool to determine the best tiling parameters and estimate the impact of said tiling inside the application-level parallel performance estimation flow.

Include support for data-dependent kernels in Tilana. The tiling optimization model currently used by Tilana assumes application kernels have constant execution times. Modeling data-dependent kernels would require either using probabilistic models for kernel execution times or, like Parana, the task trace of a particular application simulation run.

Detection of data-dependencies between loop iterations in Parana. Parana, like the OpenMP programming model, cannot handle data-dependencies between different iterations. Some

parallelization design aid tools such as Intel's Advisor can detect such data-dependencies and warn users so that they can take the appropriate measures.

Automatic parallel source code generation from the parallelization scenario specs. The OpenMP directives in the parallelization scenario specifications could be combined to the original sequential code of the application to automatically generate a parallel source code.

Automatic application code instrumentation. Adding automatic instrumentation to all possible parallelization candidate loops and sections might reduce the burden on the user and can lead to a further automation of the parallelization DSE process. Several methods can be envisaged to perform automatic instrumentation:

1. Using a code transformation tool like Coccinelle or OPARI to preprocess the source code and instrument it before calling the compiler.
2. Adding a CLANG compiler pass to instrument the code during the compilation process.
3. Doing binary instrumentation with a program like Pin. Either a new program specific to STMicroelectronics's processors would need to be built, or support for STxP70's binary code would have to be added to Pin.

In any case, a method for selecting the loops and code sections to which apply the instrumentation would need to be determined. Critical path analysis tools such as Kismet and Parkour instrument all loops in the source code, but this might lead to unnecessarily large task trace files as very small loops would rarely constitute good parallelization candidates.

Automatic Design Space Exploration (DSE). If automatic identification of potential parallelization targets and automatic instrumentation are available, a DSE methodology could be envisaged in which Parana could automatically generate and evaluate several parallelization scenarios. Genetic algorithms could be used to select suitable candidates and evolve towards a practical solution.

Include hardware IP models support. In a previous work by Ishikawa et al.[101], the authors include hardware IP models in a cross platform performance estimation tool. A parametrizable performance model of the hardware IP function is used to predict its execution time. A stub function is used in acquiring application characterization data. Upon predicting the application-level performance with the IP model, the stub function time is substituted with the timing estimated by the analytical performance model. This method effectively allows to estimate the gains that would be produced by replacing a software implementation of a particular function by a hardware IP that implements the same function. The integration of such functionality in Parana would allow the user to explore trade-offs in using IP models in different parallelization scenarios and multiprocessor configurations.

Include energy consumption estimates. Characterizing the STxP70 ASMP's energy consumption for different operating modes and for each instruction could allow to quickly estimate the energy consumed in a particular application parallelization scenario in Parana. Herglotz, Fey et al. [84] propose a similar mechanism for functional simulators. Ducroux et al. [64] describe a method for providing power estimations on the STHORM simulator, while more recently Atitallah et al. [31] proposed a power estimation methodology based on STHORM traces. Therefore, such energy consumption information of the instructions could be collected directly from the simulator or included by the Trace Filter tools upon processing the instruction traces. If tasks in the application task traces were annotated with such information, we believe the Parana tool could be extended to provide fairly accurate application energy consumption estimates in its parallelization reports.

Appendix A

Example of a Parana's Parallelization Analysis Report

Listing A.1: *Example of Parana's parallelization analysis report for the FAST application. This report refers to the parallelization scenario consisting of an outer-loop parallelization with dynamic scheduling. Refer to Chapter 5, Section 5.5 for more details on the experimental setup.*

```
1  --------------------------------------------------------------------------------
2                                      PARANA
3                                  Parallel Analyzer
4  --------------------------------------------------------------------------------
5
6  Module:              <module name>
7  SVN Revision:        <svn revision tag>
8  Build Configuration:<cmake build configuration>
9  Compiler:            <compiler name>
10 Executable:          <executable file>
11 Command line:        <command line arguments>
12 Simulator:           <simulator name>
13 Core Configuration: <core configuration register values>
14 Date:                <simulation date>
15
16 --------------------------------------------------------------------------------
17 Parallelization Analysis
18 --------------------------------------------------------------------------------
19
20 Sequential schedule
21 -------------------
22
23     Number of processors                                    1
24     Execution time          (cycles)              18,644,418
25     Sequential time         (cycles)          12,757 (  0.1%)
26     Parallel time           (cycles)      18,631,661 ( 99.9%)
27      - Processing           (cycles)      18,631,661 ( 99.9%)
28      - Load imbalance       (cycles)               0 (  0.0%)
29      - Limited parallelism (cycles)               0 (  0.0%)
30     Scheduling runtime      (s)                          5.3
31
32 Parallel schedule with 1 processor
33 ---------------------------------
34
35     Number of processors                                    1
36     Execution time          (cycles)              18,689,291
37     Speedup                                            1.0x
38     Amdahl's Law Max. Speedup                          1.0x
39     Parallel efficiency                               99.8%
40     Sequential time         (cycles)          12,757 (  0.1%)
41     Parallel time           (cycles)      18,676,534 ( 99.9%)
42      - Processing           (cycles)      18,631,661 ( 99.7%)
43      - Load imbalance       (cycles)               0 (  0.0%)
44      - Limited parallelism (cycles)               0 (  0.0%)
45      - Overhead             (cycles)          44,873 (  0.2%)
46      - Idle                 (cycles)               0 (  0.0%)
47     Scheduling runtime      (s)                          4.3
```

```
48
49  Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
50  -------------------------------------------------------
51      [*0] 1 __asmp_ind_crt0.
52      [+1] 13 main.
53      [+2] 14 vaFastCornerDetect.
54      [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
55      Number of processors                1
56      Number of calls                     1
57                               Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
58      Total time                    6,019,785      32.2%      6,019,785     32.2%
59      Execution time               18,676,534     100.0%     18,676,534    100.0%
60      Sequential time                       0       0.0%              0      0.0%
61      Parallel time                18,676,534     100.0%     18,676,534    100.0%
62       - Processing                 5,974,912      32.0%      5,974,912     32.0%
63       - Load imbalance                     0       0.0%              0      0.0%
64       - Limited parallelism                0       0.0%              0      0.0%
65       - Overhead                      44,873       0.2%         44,873      0.2%
66       - Idle                               0       0.0%              0      0.0%
67
68
69  Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
70  -------------------------------------------------------------------
71      [*0] 1 __asmp_ind_crt0.
72      [+1] 13 main.
73      [+2] 14 vaFastCornerDetect.
74      [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
75      [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
76      Number of processors                1
77      Number of calls                     1
78                               Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
79      Total time                    6,018,475      32.2%      6,018,475     32.2%
80      Execution time               18,675,224     100.0%     18,675,224    100.0%
81      Sequential time                       0       0.0%              0      0.0%
82      Parallel time                18,675,224     100.0%     18,675,224    100.0%
83       - Processing                 5,974,912      32.0%      5,974,912     32.0%
84       - Load imbalance                     0       0.0%              0      0.0%
85       - Limited parallelism                0       0.0%              0      0.0%
86       - Overhead                      43,563       0.2%         43,563      0.2%
87       - Idle                               0       0.0%              0      0.0%
88
89  Parallel schedule with 2 processors
90  -----------------------------------
91
92      Number of processors                                    2
93      Execution time         (cycles)                 9,364,866
94      Speedup                                             2.0x
95      Amdahl's Law Max. Speedup                           2.0x
96      Parallel efficiency                                99.5%
97      Sequential time        (cycles)          12,757 (  0.1%)
98      Parallel time          (cycles)       9,352,109 ( 99.9%)
99       - Processing          (cycles)       9,315,831 ( 99.5%)
100      - Load imbalance      (cycles)          12,884 (  0.1%)
101      - Limited parallelism (cycles)               0 (  0.0%)
102      - Overhead            (cycles)          23,233 (  0.2%)
103      - Idle                (cycles)             162 (  0.0%)
104     Scheduling runtime     (s)                            4.1
105
106 Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
107 -------------------------------------------------------
108     [*0] 1 __asmp_ind_crt0.
109     [+1] 13 main.
110     [+2] 14 vaFastCornerDetect.
111     [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
112     Number of processors                2
113     Number of calls                     1
114                              Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
```

```
    Total time                   3,023,735      32.3%      3,023,735     32.3%
    Execution time               9,352,109     100.0%      9,352,109    100.0%
    Sequential time                      0       0.0%              0      0.0%
    Parallel time                9,352,109     100.0%      9,352,109    100.0%
     - Processing                2,987,456      31.9%      2,987,456     31.9%
     - Load imbalance               12,884       0.1%         12,884      0.1%
     - Limited parallelism               0       0.0%              0      0.0%
     - Overhead                     23,233       0.2%         23,233      0.2%
     - Idle                            162       0.0%            162      0.0%

Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
-------------------------------------------------------------------
    [*0] 1 __asmp_ind_crt0.
    [+1] 13 main.
    [+2] 14 vaFastCornerDetect.
    [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
    [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
    Number of processors                2
    Number of calls                     1
                             Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
    Total time                   3,022,387      32.3%      3,022,387     32.3%
    Execution time               9,350,761     100.0%      9,350,761    100.0%
    Sequential time                      0       0.0%              0      0.0%
    Parallel time                9,350,761     100.0%      9,350,761    100.0%
     - Processing                2,987,456      31.9%      2,987,456     31.9%
     - Load imbalance               12,870       0.1%         12,870      0.1%
     - Limited parallelism               0       0.0%              0      0.0%
     - Overhead                     22,061       0.2%         22,061      0.2%
     - Idle                              0       0.0%              0      0.0%


Parallel schedule with 3 processors
-----------------------------------

    Number of processors                                        3
    Execution time         (cycles)                     6,259,883
    Speedup                                                  3.0x
    Amdahl's Law Max. Speedup                                3.0x
    Parallel efficiency                                     99.3%
    Sequential time        (cycles)           12,757 (  0.2%)
    Parallel time          (cycles)        6,247,126 ( 99.8%)
     - Processing          (cycles)        6,210,554 ( 99.2%)
     - Load imbalance      (cycles)           20,285 (  0.3%)
     - Limited parallelism (cycles)                0 (  0.0%)
     - Overhead            (cycles)           16,071 (  0.3%)
     - Idle                (cycles)              216 (  0.0%)
    Scheduling runtime     (s)                             4.1

Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
-------------------------------------------------------
    [*0] 1 __asmp_ind_crt0.
    [+1] 13 main.
    [+2] 14 vaFastCornerDetect.
    [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
    Number of processors                3
    Number of calls                     1
                             Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
    Total time                   2,028,210      32.5%      2,028,210     32.5%
    Execution time               6,247,126     100.0%      6,247,126    100.0%
    Sequential time                      0       0.0%              0      0.0%
    Parallel time                6,247,126     100.0%      6,247,126    100.0%
     - Processing                1,991,637      31.9%      1,991,637     31.9%
     - Load imbalance               20,285       0.3%         20,285      0.3%
     - Limited parallelism               0       0.0%              0      0.0%
     - Overhead                     16,071       0.3%         16,071      0.3%
     - Idle                            216       0.0%            216      0.0%

```

```
182
183 Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
184 ---------------------------------------------------------------------
185     [*0] 1 __asmp_ind_crt0.
186     [+1] 13 main.
187     [+2] 14 vaFastCornerDetect.
188     [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
189     [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
190     Number of processors                3
191     Number of calls                     1
192                             Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
193     Total time                   2,026,798      32.5%      2,026,798     32.5%
194     Execution time               6,245,714     100.0%      6,245,714    100.0%
195     Sequential time                      0       0.0%              0      0.0%
196     Parallel time                6,245,714     100.0%      6,245,714    100.0%
197      - Processing                1,991,637      31.9%      1,991,637     31.9%
198      - Load imbalance               20,267       0.3%         20,267      0.3%
199      - Limited parallelism               0       0.0%              0      0.0%
200      - Overhead                     14,893       0.2%         14,893      0.2%
201      - Idle                              0       0.0%              0      0.0%
202
203
204 Parallel schedule with 4 processors
205 ----------------------------------
206
207     Number of processors                                    4
208     Execution time         (cycles)                 4,709,628
209     Speedup                                              4.0x
210     Amdahl's Law Max. Speedup                            4.0x
211     Parallel efficiency                                 99.0%
212     Sequential time        (cycles)          12,757 (  0.3%)
213     Parallel time          (cycles)       4,696,871 ( 99.7%)
214      - Processing          (cycles)       4,657,915 ( 98.9%)
215      - Load imbalance      (cycles)          26,191 (  0.6%)
216      - Limited parallelism (cycles)               0 (  0.0%)
217      - Overhead            (cycles)          12,522 (  0.3%)
218      - Idle                (cycles)             243 (  0.0%)
219     Scheduling runtime     (s)                    4.1
220
221 Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
222 -------------------------------------------------------
223     [*0] 1 __asmp_ind_crt0.
224     [+1] 13 main.
225     [+2] 14 vaFastCornerDetect.
226     [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
227     Number of processors                4
228     Number of calls                     1
229                             Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
230     Total time                   1,532,684      32.6%      1,532,684     32.6%
231     Execution time               4,696,871     100.0%      4,696,871    100.0%
232     Sequential time                      0       0.0%              0      0.0%
233     Parallel time                4,696,871     100.0%      4,696,871    100.0%
234      - Processing                1,493,728      31.8%      1,493,728     31.8%
235      - Load imbalance               26,191       0.6%         26,191      0.6%
236      - Limited parallelism               0       0.0%              0      0.0%
237      - Overhead                     12,522       0.3%         12,522      0.3%
238      - Idle                            243       0.0%            243      0.0%
239
240
241 Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
242 ---------------------------------------------------------------------
243     [*0] 1 __asmp_ind_crt0.
244     [+1] 13 main.
245     [+2] 14 vaFastCornerDetect.
246     [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
247     [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
248     Number of processors                4
```

```
249      Number of calls                     1
250                                Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
251      Total time                  1,531,208      32.6%      1,531,208     32.6%
252      Execution time              4,695,395     100.0%      4,695,395    100.0%
253      Sequential time                     0       0.0%              0      0.0%
254      Parallel time               4,695,395     100.0%      4,695,395    100.0%
255       - Processing               1,493,728      31.8%      1,493,728     31.8%
256       - Load imbalance              26,171       0.6%         26,171      0.6%
257       - Limited parallelism              0       0.0%              0      0.0%
258       - Overhead                    11,309       0.2%         11,309      0.2%
259       - Idle                             0       0.0%              0      0.0%
260
261  Parallel schedule with 5 processors
262  -----------------------------------
263
264      Number of processors                                      5
265      Execution time         (cycles)                 3,794,241
266      Speedup                                              4.9x
267      Amdahl's Law Max. Speedup                            5.0x
268      Parallel efficiency                                 98.3%
269      Sequential time        (cycles)            12,757 (  0.3%)
270      Parallel time          (cycles)         3,781,484 ( 99.7%)
271       - Processing          (cycles)         3,726,332 ( 98.2%)
272       - Load imbalance      (cycles)            44,474 (  1.2%)
273       - Limited parallelism (cycles)                 0 (  0.0%)
274       - Overhead            (cycles)            10,418 (  0.3%)
275       - Idle                (cycles)               259 (  0.0%)
276      Scheduling runtime     (s)                    4.4
277
278  Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
279  -------------------------------------------------------
280      [*0] 1 __asmp_ind_crt0.
281      [+1] 13 main.
282      [+2] 14 vaFastCornerDetect.
283      [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
284      Number of processors                5
285      Number of calls                     1
286                                Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
287      Total time                  1,250,134      33.1%      1,250,134     33.1%
288      Execution time              3,781,484     100.0%      3,781,484    100.0%
289      Sequential time                     0       0.0%              0      0.0%
290      Parallel time               3,781,484     100.0%      3,781,484    100.0%
291       - Processing               1,194,982      31.6%      1,194,982     31.6%
292       - Load imbalance              44,474       1.2%         44,474      1.2%
293       - Limited parallelism              0       0.0%              0      0.0%
294       - Overhead                    10,418       0.3%         10,418      0.3%
295       - Idle                           259       0.0%            259      0.0%
296
297
298  Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
299  --------------------------------------------------------------------
300      [*0] 1 __asmp_ind_crt0.
301      [+1] 13 main.
302      [+2] 14 vaFastCornerDetect.
303      [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
304      [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
305      Number of processors                5
306      Number of calls                     1
307                                Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
308      Total time                  1,248,594      33.0%      1,248,594     33.0%
309      Execution time              3,779,944     100.0%      3,779,944    100.0%
310      Sequential time                     0       0.0%              0      0.0%
311      Parallel time               3,779,944     100.0%      3,779,944    100.0%
312       - Processing               1,194,982      31.6%      1,194,982     31.6%
313       - Load imbalance              44,453       1.2%         44,453      1.2%
314       - Limited parallelism              0       0.0%              0      0.0%
315       - Overhead                     9,159       0.2%          9,159      0.2%
```

```
316       - Idle                                0       0.0%              0      0.0%
317
318
319   Parallel schedule with 6 processors
320   -----------------------------------
321
322       Number of processors                                        6
323       Execution time         (cycles)                 3,163,801
324       Speedup                                                 5.9x
325       Amdahl's Law Max. Speedup                               6.0x
326       Parallel efficiency                                    98.2%
327       Sequential time        (cycles)            12,757 (  0.4%)
328       Parallel time          (cycles)         3,151,044 ( 99.6%)
329        - Processing          (cycles)         3,105,277 ( 98.2%)
330        - Load imbalance      (cycles)            36,460 (  1.2%)
331        - Limited parallelism (cycles)                 0 (  0.0%)
332        - Overhead            (cycles)             9,037 (  0.3%)
333        - Idle                (cycles)               270 (  0.0%)
334       Scheduling runtime     (s)                      4.6
335
336   Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
337   -------------------------------------------------------
338       [*0] 1 __asmp_ind_crt0.
339       [+1] 13 main.
340       [+2] 14 vaFastCornerDetect.
341       [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
342       Number of processors                 6
343       Number of calls                      1
344                               Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
345       Total time                   1,041,586      33.1%      1,041,586     33.1%
346       Execution time               3,151,044     100.0%      3,151,044    100.0%
347       Sequential time                      0       0.0%              0      0.0%
348       Parallel time                3,151,044     100.0%      3,151,044    100.0%
349        - Processing                  995,819      31.6%        995,819     31.6%
350        - Load imbalance               36,460       1.2%         36,460      1.2%
351        - Limited parallelism               0       0.0%              0      0.0%
352        - Overhead                      9,037       0.3%          9,037      0.3%
353        - Idle                            270       0.0%            270      0.0%
354
355
356   Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
357   --------------------------------------------------------------------
358       [*0] 1 __asmp_ind_crt0.
359       [+1] 13 main.
360       [+2] 14 vaFastCornerDetect.
361       [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
362       [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
363       Number of processors                 6
364       Number of calls                      1
365                               Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
366       Total time                   1,039,982      33.0%      1,039,982     33.0%
367       Execution time               3,149,440     100.0%      3,149,440    100.0%
368       Sequential time                      0       0.0%              0      0.0%
369       Parallel time                3,149,440     100.0%      3,149,440    100.0%
370        - Processing                  995,819      31.6%        995,819     31.6%
371        - Load imbalance               36,438       1.2%         36,438      1.2%
372        - Limited parallelism               0       0.0%              0      0.0%
373        - Overhead                      7,726       0.2%          7,726      0.2%
374        - Idle                              0       0.0%              0      0.0%
375
376   Parallel schedule with 7 processors
377   -----------------------------------
378
379       Number of processors                                        7
380       Execution time         (cycles)                 2,717,958
381       Speedup                                                 6.9x
382       Amdahl's Law Max. Speedup                               7.0x
```

```
383      Parallel efficiency                                    98.0%
384      Sequential time        (cycles)          12,757 (  0.5%)
385      Parallel time          (cycles)       2,705,201 ( 99.5%)
386       - Processing          (cycles)       2,661,666 ( 97.9%)
387       - Load imbalance      (cycles)          35,189 (  1.3%)
388       - Limited parallelism (cycles)               0 (  0.0%)
389       - Overhead            (cycles)           8,069 (  0.3%)
390       - Idle                (cycles)             278 (  0.0%)
391      Scheduling runtime     (s)                    4.1
392
393  Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
394  -------------------------------------------------------
395      [*0] 1 __asmp_ind_crt0.
396      [+1] 13 main.
397      [+2] 14 vaFastCornerDetect.
398      [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
399      Number of processors                7
400      Number of calls                     1
401                              Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
402      Total time                     897,094      33.2%        897,094     33.2%
403      Execution time               2,705,201     100.0%      2,705,201    100.0%
404      Sequential time                      0       0.0%              0      0.0%
405      Parallel time                2,705,201     100.0%      2,705,201    100.0%
406       - Processing                  853,559      31.6%        853,559     31.6%
407       - Load imbalance               35,189       1.3%         35,189      1.3%
408       - Limited parallelism               0       0.0%              0      0.0%
409       - Overhead                      8,069       0.3%          8,069      0.3%
410       - Idle                            278       0.0%            278      0.0%
411
412
413  Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
414  --------------------------------------------------------------------
415      [*0] 1 __asmp_ind_crt0.
416      [+1] 13 main.
417      [+2] 14 vaFastCornerDetect.
418      [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
419      [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
420      Number of processors                7
421      Number of calls                     1
422                              Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
423      Total time                     895,426      33.1%        895,426     33.1%
424      Execution time               2,703,533     100.0%      2,703,533    100.0%
425      Sequential time                      0       0.0%              0      0.0%
426      Parallel time                2,703,533     100.0%      2,703,533    100.0%
427       - Processing                  853,559      31.6%        853,559     31.6%
428       - Load imbalance               35,166       1.3%         35,166      1.3%
429       - Limited parallelism               0       0.0%              0      0.0%
430       - Overhead                      6,702       0.2%          6,702      0.2%
431       - Idle                              0       0.0%              0      0.0%
432
433  Parallel schedule with 8 processors
434  -----------------------------------
435
436      Number of processors                                       8
437      Execution time         (cycles)                2,391,652
438      Speedup                                              7.8x
439      Amdahl's Law Max. Speedup                            8.0x
440      Parallel efficiency                                 97.4%
441      Sequential time        (cycles)          12,757 (  0.5%)
442      Parallel time          (cycles)       2,378,895 ( 99.5%)
443       - Processing          (cycles)       2,328,958 ( 97.4%)
444       - Load imbalance      (cycles)          42,295 (  1.8%)
445       - Limited parallelism (cycles)               0 (  0.0%)
446       - Overhead            (cycles)           7,359 (  0.3%)
447       - Idle                (cycles)             284 (  0.0%)
448      Scheduling runtime     (s)                    4.1
449
```

```
Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
-------------------------------------------------------
    [*0] 1 __asmp_ind_crt0.
    [+1] 13 main.
    [+2] 14 vaFastCornerDetect.
    [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
    Number of processors                8
    Number of calls                     1
                            Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
    Total time                     796,801      33.5%        796,801     33.5%
    Execution time               2,378,895     100.0%      2,378,895    100.0%
    Sequential time                      0       0.0%              0      0.0%
    Parallel time                2,378,895     100.0%      2,378,895    100.0%
     - Processing                  746,864      31.4%        746,864     31.4%
     - Load imbalance               42,295       1.8%         42,295      1.8%
     - Limited parallelism               0       0.0%              0      0.0%
     - Overhead                      7,359       0.3%          7,359      0.3%
     - Idle                            284       0.0%            284      0.0%


Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
--------------------------------------------------------------------
    [*0] 1 __asmp_ind_crt0.
    [+1] 13 main.
    [+2] 14 vaFastCornerDetect.
    [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
    [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
    Number of processors                8
    Number of calls                     1
                            Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
    Total time                     795,069      33.4%        795,069     33.4%
    Execution time               2,377,163     100.0%      2,377,163    100.0%
    Sequential time                      0       0.0%              0      0.0%
    Parallel time                2,377,163     100.0%      2,377,163    100.0%
     - Processing                  746,864      31.4%        746,864     31.4%
     - Load imbalance               42,272       1.8%         42,272      1.8%
     - Limited parallelism               0       0.0%              0      0.0%
     - Overhead                      5,934       0.2%          5,934      0.2%
     - Idle                              0       0.0%              0      0.0%
```

Appendix B

Example of a Parana's Application Profile Report

Listing B.1: *Example of Parana's application profile report for the FAST application. This profile refers to the parallelization scenario consisting of an outer-loop parallelization with dynamic scheduling with a single processor. All timing figures are reported in processor cycles. Average (Avg.) is relative to the number of calls. Cumulative (Cml.) includes self and children times, while Self time excludes children times. Child tasks which are the result of segmentation – those with prologue, epilogue or interstice label suffixes – do not count twice towards the cumulative times. Refer to Chapter 5, Section 5.5 for more details on the experimental setup.*

```
1  --------------------------------------------------------------------------------
2                               PARANA
3                          Parallel Analyzer
4  --------------------------------------------------------------------------------
5
6  Module:             <module name>
7  SVN Revision:       <svn revision tag>
8  Build Configuration:<cmake build configuration>
9  Compiler:           <compiler name>
10 Executable:         <executable file>
11 Command line:       <command line arguments>
12 Simulator:          <simulator name>
13 Core Configuration: <core configuration register values>
14 Date:               <simulation date>
15
16 --------------------------------------------------------------------------------
17 Parallelization Analysis
18 --------------------------------------------------------------------------------
19
20 Parallel schedule with 1 processor
21 ---------------------------------
22
23     Number of processors                                    1
24     Execution time         (cycles)              18,689,291
25     Speedup                                            1.0x
26     Amdahl's Law Max. Speedup                          1.0x
27     Parallel efficiency                               99.8%
28     Sequential time        (cycles)         12,757 (  0.1%)
29     Parallel time          (cycles)      18,676,534 ( 99.9%)
30      - Processing          (cycles)      18,631,661 ( 99.7%)
31      - Load imbalance      (cycles)               0 (  0.0%)
32      - Limited parallelism (cycles)               0 (  0.0%)
33      - Overhead            (cycles)          44,873 (  0.2%)
34      - Idle                (cycles)               0 (  0.0%)
35     Scheduling runtime     (s)                      4.3
36
37 Parallel Region: 42 vaFastCornerDetect.for_y [PARALLEL]
38 ------------------------------------------------------
39     [*0] 1 __asmp_ind_crt0.
40     [+1] 13 main.
41     [+2] 14 vaFastCornerDetect.
42     [=3] 42 vaFastCornerDetect.for_y [PARALLEL]
```

```
43      Number of processors                 1
44      Number of calls                      1
45                              Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
46      Total time                   6,019,785      32.2%      6,019,785     32.2%
47      Execution time              18,676,534     100.0%     18,676,534    100.0%
48      Sequential time                      0       0.0%              0      0.0%
49      Parallel time               18,676,534     100.0%     18,676,534    100.0%
50       - Processing                5,974,912      32.0%      5,974,912     32.0%
51       - Load imbalance                    0       0.0%              0      0.0%
52       - Limited parallelism               0       0.0%              0      0.0%
53       - Overhead                     44,873       0.2%         44,873      0.2%
54       - Idle                              0       0.0%              0      0.0%
55
56  Parallel Region: 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
57  ---------------------------------------------------------------------
58      [*0] 1 __asmp_ind_crt0.
59      [+1] 13 main.
60      [+2] 14 vaFastCornerDetect.
61      [+3] 42 vaFastCornerDetect.for_y [PARALLEL]
62      [=4] 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]
63      Number of processors                 1
64      Number of calls                      1
65                              Total (cycles)  Total (%)  Avg. (cycles)  Avg. (%)
66      Total time                   6,018,475      32.2%      6,018,475     32.2%
67      Execution time              18,675,224     100.0%     18,675,224    100.0%
68      Sequential time                      0       0.0%              0      0.0%
69      Parallel time               18,675,224     100.0%     18,675,224    100.0%
70       - Processing                5,974,912      32.0%      5,974,912     32.0%
71       - Load imbalance                    0       0.0%              0      0.0%
72       - Limited parallelism               0       0.0%              0      0.0%
73       - Overhead                     43,563       0.2%         43,563      0.2%
74       - Idle                              0       0.0%              0      0.0%
75
76  Profiling of parallel schedule with 1 processor
77  -----------------------------------------------
78
79  Function Call Tree                                                          \
                #Calls     #Cml.  #Avg. Cml.     #Self  #Avg. Self
80  + -1 root.
81   + 1 __asmp_ind_crt0.                                                       \
                     1  18689291  18689291.0      3546      3546.0
82    + 2 __asmp_runtime_init.                                                 \
                     1      3208      3208.0       125       125.0
83     + 3 __asmp_rt_thread_init.                                              \
                     1      2906      2906.0      1200      1200.0
84      - 4 __asmp_rt_thread_init.__asmp_rt_thread_init_memset_prologue        \
                     1        95        95.0        95        95.0
85      - 5 memset.                                                            \
                    24      1706        71.1      1706        71.1
86      - 6 __asmp_rt_thread_init.memset_memset_interstice                     \
                    23      1048        45.6      1048        45.6
87      - 7 __asmp_rt_thread_init.memset___asmp_rt_thread_init_epilogue        \
                     1        57        57.0        57        57.0
88     - 8 __asmp_runtime_init.__asmp_runtime_init___asmp_rt_thread_init_prologue  \
                     1        48        48.0        48        48.0
89     - 9 __asmp_runtime_init.__asmp_rt_thread_init___asmp_rt_lock_table_init_in...\
                     1        16        16.0        16        16.0
90     - 10 __asmp_rt_lock_table_init.                                         \
                     2       177        88.5       177        88.5
91     - 11 __asmp_runtime_init.__asmp_rt_lock_table_init___asmp_rt_lock_table_in...\
                     1         4         4.0         4         4.0
92     - 12 __asmp_runtime_init.__asmp_rt_lock_table_init___asmp_runtime_init_epi...\
                     1        57        57.0        57        57.0
93    + 13 main.                                                               \
                     1  18682148  18682148.0        98        98.0
94     + 14 vaFastCornerDetect.                                                \
                     1  18682050  18682050.0      4725      4725.0
```

```
95      + 15 vaMAlloc.                                                                  \
                    1       507       507.0        20        20.0
96       + 16 malloc.                                                                   \
                    1       487       487.0       301       301.0
97        + 17 __malloc_lock.                                                           \
                    1        50        50.0        20        20.0
98         - 18 __malloc_lock.__malloc_lock___asmp_rt_lock_acquire_re_prologue         \
                    1        10        10.0        10        10.0
99         - 19 __asmp_rt_lock_acquire_re.                                             \
                    1        30        30.0        30        30.0
100        - 20 __malloc_lock.__asmp_rt_lock_acquire_re___malloc_lock_epilogue         \
                    1        10        10.0        10        10.0
101       + 21 malloc_extend_top.                                                      \
                    1       142       142.0       106       106.0
102        + 22 _sbrk_r.                                                               \
                    1        36        36.0        18        18.0
103         - 23 _sbrk_r._sbrk_r__sbrk_prologue                                        \
                    1         8         8.0         8         8.0
104         - 24 _sbrk.                                                                \
                    1        18        18.0        18        18.0
105         - 25 _sbrk_r._sbrk__sbrk_r_epilogue                                        \
                    1        10        10.0        10        10.0
106        - 26 malloc_extend_top.malloc_extend_top__sbrk_r_prologue                   \
                    1        45        45.0        45        45.0
107        - 27 malloc_extend_top._sbrk_r_malloc_extend_top_epilogue                   \
                    1        61        61.0        61        61.0
108       + 28 __malloc_unlock.                                                        \
                    1        52        52.0        20        20.0
109        - 29 __malloc_unlock.__malloc_unlock___asmp_rt_lock_release_re_prologue     \
                    1        10        10.0        10        10.0
110        - 30 __asmp_rt_lock_release_re.                                             \
                    1        32        32.0        32        32.0
111        - 31 __malloc_unlock.__asmp_rt_lock_release_re___malloc_unlock_epilogue     \
                    1        10        10.0        10        10.0
112       - 32 malloc.malloc___getreent_prologue                                       \
                    1         7         7.0         7         7.0
113       - 33 __getreent.                                                             \
                    1         9         9.0         9         9.0
114       - 34 malloc.__getreent___malloc_lock_interstice                              \
                    1        39        39.0        39        39.0
115       - 35 malloc.__malloc_lock___divw_interstice                                  \
                    1        37        37.0        37        37.0
116       - 36 __divw.                                                                 \
                    1        36        36.0        36        36.0
117       - 37 malloc.__divw_malloc_extend_top_interstice                              \
                    1        33        33.0        33        33.0
118       - 38 malloc.malloc_extend_top___malloc_unlock_interstice                     \
                    1        57        57.0        57        57.0
119       - 39 malloc.__malloc_unlock_malloc_epilogue                                  \
                    1        25        25.0        25        25.0
120      - 40 vaMAlloc.vaMAlloc_malloc_prologue                                        \
                    1         7         7.0         7         7.0
121      - 41 vaMAlloc.malloc_vaMAlloc_epilogue                                        \
                    1        13        13.0        13        13.0
122     + 42 vaFastCornerDetect.for_y [PARALLEL]                                       \
                    1  18676534  18676534.0      1310      1310.0
123      + 43 vaFastCornerDetect.for_y [FOR SCHEDULE(DYNAMIC)]                         \
                    1  18675224  18675224.0     45915     45915.0
124       + 44 vaFastCornerDetect.for_y [FOR_ITERATION SCHEDULE(DYNAMIC)]              \
                  235  18629309     79273.7    934006      3974.5
125        + 45 vaMAlign.                                                              \
                    1       168       168.0        20        20.0
126         - 46 vaMAlign.vaMAlign___modw_prologue                                     \
                    1         9         9.0         9         9.0
127         - 47 __modw.                                                               \
                    1       148       148.0       148       148.0
128         - 48 vaMAlign.__modw_vaMAlign_epilogue                                     \
```

```
                    1        11        11.0        11        11.0
129         + 49 vaFastCornerDetect.for_x                                    \
                   73476  17418377       237.1  14945798       203.4
130          - 50 vaFastCornerDetect.for_x_vaFastCornerScore_prologue        \
                    3129   2199313       702.9   2199313       702.9
131          - 51 vaFastCornerScore.                                         \
                    3129   2472579       790.2   2472579       790.2
132          - 52 vaFastCornerDetect.vaFastCornerScore_for_x_epilogue        \
                    3129     89736        28.7     89736        28.7
133         - 53 vaFastCornerDetect.for_y_vaMAlign_prologue                  \
                    1        12        12.0        12        12.0
134         - 54 vaFastCornerDetect.vaMAlign_memset_interstice               \
                    1        23        23.0        23        23.0
135         - 55 memset.                                                     \
                  236    276758      1172.7    276758      1172.7
136         - 56 vaFastCornerDetect.memset_memset_interstice                 \
                    1        56        56.0        56        56.0
137         - 57 vaFastCornerDetect.memset_for_x_interstice                  \
                  234      5880        25.1      5880        25.1
138         - 58 vaFastCornerDetect.for_x_for_x_interstice                   \
                73242    442510         6.0    442510         6.0
139         - 59 vaFastCornerDetect.for_x_for_y_epilogue                     \
                  234    475930      2033.9    475930      2033.9
140         - 60 vaFastCornerDetect.for_y_memset_prologue                    \
                  234      6318        27.0      6318        27.0
141         - 61 vaFastCornerDetect.memset_for_y_epilogue                    \
                    1      3277      3277.0      3277      3277.0
142        - 62 vaFastCornerDetect.for_y_for_prologue_P0                     \
                    1       260       260.0       260       260.0
143        - 63 vaFastCornerDetect.for_y_for_y_interstice                    \
                  234      2352        10.1      2352        10.1
144        - 64 vaFastCornerDetect.for_y_for_next_chunk                      \
                  234     42822       183.0     42822       183.0
145        - 65 vaFastCornerDetect.for_y_load_imbalance_P0                   \
                    1                   0.0                 0.0
146        - 66 vaFastCornerDetect.for_y_for_epilogue_P0                     \
                    1       481       481.0       481       481.0
147       - 67 vaFastCornerDetect.for_y_parallel_fork                       \
                    1                   0.0                 0.0
148       - 68 vaFastCornerDetect.for_y_init_parallel_region                \
                    1       652       652.0       652       652.0
149       - 69 vaFastCornerDetect.for_y_launch_master_worker_thread         \
                    1                   0.0                 0.0
150       - 70 vaFastCornerDetect.for_y_worker_thread_prologue_P0           \
                    1        26        26.0        26        26.0
151       - 71 vaFastCornerDetect.for_y_worker_thread_epilogue_P0           \
                    1       281       281.0       281       281.0
152       - 72 vaFastCornerDetect.for_y_release_master_worker_thread        \
                    1        27        27.0        27        27.0
153       - 73 vaFastCornerDetect.for_y_load_imbalance_P0                   \
                    1                   0.0                 0.0
154       - 74 vaFastCornerDetect.for_y_release_parallel_region             \
                    1       324       324.0       324       324.0
155       - 75 vaFastCornerDetect.for_y_parallel_join                       \
                    1                   0.0                 0.0
156      + 76 vaFree.                                                       \
                    1       252       252.0       161       161.0
157       + 77 __malloc_lock.                                               \
                    1        50        50.0        20        20.0
158        - 78 __malloc_lock.__malloc_lock___asmp_rt_lock_acquire_re_prologue \
                    1        10        10.0        10        10.0
159        - 79 __asmp_rt_lock_acquire_re.                                  \
                    1        30        30.0        30        30.0
160        - 80 __malloc_lock.__asmp_rt_lock_acquire_re___malloc_lock_epilogue \
                    1        10        10.0        10        10.0
161       - 81 vaFree.vaFree___getreent_prologue                            \
                    1        30        30.0        30        30.0
```

```
162       - 82 __getreent.                                                                  \
                        1         9         9.0         9         9.0
163       - 83 vaFree.__getreent___malloc_lock_interstice                                   \
                        1        26        26.0        26        26.0
164       - 84 vaFree.__malloc_lock___asmp_rt_lock_release_re_interstice                    \
                        1        95        95.0        95        95.0
165       - 85 __asmp_rt_lock_release_re.                                                   \
                        1        32        32.0        32        32.0
166       - 86 vaFree.__asmp_rt_lock_release_re_vaFree_epilogue                             \
                        1        10        10.0        10        10.0
167      - 87 vaFastCornerDetect.vaFastCornerDetect_vaGetImageType_prologue                 \
                        1        69        69.0        69        69.0
168      - 88 vaGetImageType.                                                              \
                        1        16        16.0        16        16.0
169      - 89 vaFastCornerDetect.vaGetImageType_vaGetImageDataType_interstice              \
                        1        15        15.0        15        15.0
170      - 90 vaGetImageDataType.                                                          \
                        1        16        16.0        16        16.0
171      - 91 vaFastCornerDetect.vaGetImageDataType_vaMAlloc_interstice                    \
                        1      4490      4490.0      4490      4490.0
172      - 92 vaFastCornerDetect.vaMAlloc_for_y_interstice                                 \
                        1       104       104.0       104       104.0
173      - 93 vaFastCornerDetect.for_y_vaFree_interstice                                   \
                        1         4         4.0         4         4.0
174      - 94 vaFastCornerDetect.vaFree_vaFastCornerDetect_epilogue                        \
                        1        43        43.0        43        43.0
175     - 95 main.main_vaFastCornerDetect_prologue                                         \
                        1        67        67.0        67        67.0
176     - 96 main.vaFastCornerDetect_main_epilogue                                         \
                        1        31        31.0        31        31.0
177    + 97 exit.                                                                          \
                        1      3545      3545.0        64        64.0
178     + 98 _cleanup_r.                                                                   \
                        1      3391      3391.0       300       300.0
179      + 99 __sfp_lock_acquire.                                                          \
                        1        50        50.0        20        20.0
180       - 100 __sfp_lock_acquire.__sfp_lock_acquire___asmp_rt_lock_acquire_re_pr...\
                        1        10        10.0        10        10.0
181       - 101 __asmp_rt_lock_acquire_re.                                                 \
                        1        30        30.0        30        30.0
182       - 102 __sfp_lock_acquire.__asmp_rt_lock_acquire_re___sfp_lock_acquire_ep...\
                        1        10        10.0        10        10.0
183      + 103 fclose.                                                                     \
                        4      2989       747.3       540       135.0
184       + 104 __sfp_lock_acquire.                                                        \
                        4       180        45.0        80        20.0
185        - 105 __sfp_lock_acquire.__sfp_lock_acquire___asmp_rt_lock_acquire_re_p...\
                        4        40        10.0        40        10.0
186        - 106 __asmp_rt_lock_acquire_re.                                               \
                        4       100        25.0        25         6.3
187        - 107 __sfp_lock_acquire.__asmp_rt_lock_acquire_re___sfp_lock_acquire_e...\
                        4        40        10.0        40        10.0
188       + 108 __libc_lock_acquire_recur.                                               \
                        4       200        50.0        81        20.3
189        - 109 __libc_lock_acquire_recur.__libc_lock_acquire_recur___asmp_rt_loc...\
                        4        41        10.3        41        10.3
190        - 110 __asmp_rt_lock_acquire_re.                                               \
                        4       119        29.8       119        29.8
191        - 111 __libc_lock_acquire_recur.__asmp_rt_lock_acquire_re___libc_lock_a...\
                        4        40        10.0        40        10.0
192       + 112 _fflush_r.                                                               \
                        4       727       181.8       342        85.5
193        + 113 __libc_lock_acquire_recur.                                              \
                        4       180        45.0        81        20.3
194         - 114 __libc_lock_acquire_recur.__libc_lock_acquire_recur___asmp_rt_lo...\
                        4        41        10.3        41        10.3
195         - 115 __asmp_rt_lock_acquire_re.                                             \
```

```
                    4        99        24.8        99        24.8
196          - 116 __libc_lock_acquire_recur.__asmp_rt_lock_acquire_re___libc_lock_...\
                    4        40        10.0        40        10.0
197          + 117 __libc_lock_release_recur.                                        \
                    4       205        51.3        81        20.3
198          - 118 __libc_lock_release_recur.__libc_lock_release_recur___asmp_rt_lo...\
                    4        41        10.3        41        10.3
199          - 119 __asmp_rt_lock_release_re.                                        \
                    4       124        31.0       124        31.0
200          - 120 __libc_lock_release_recur.__asmp_rt_lock_release_re___libc_lock_...\
                    4        40        10.0        40        10.0
201          - 121 _fflush_r._fflush_r___libc_lock_acquire_recur_prologue            \
                    4       123        30.8       123        30.8
202          - 122 _fflush_r.__libc_lock_acquire_recur___libc_lock_release_recur_int...\
                    4       163        40.8       163        40.8
203          - 123 _fflush_r.__libc_lock_release_recur__fflush_r_epilogue            \
                    4        56        14.0        53        13.3
204          + 124 __sclose.                                                         \
                    4       412       103.0       176        44.0
205          - 125 __sclose.__sclose___errno_prologue                                \
                    4        68        17.0        68        17.0
206          - 126 __errno.                                                          \
                   12       132        11.0       132        11.0
207          - 127 __sclose.__errno___errno_interstice                               \
                    4        20         5.0        20         5.0
208          - 128 __sclose.__errno__close_interstice                                \
                    4        16         4.0        16         4.0
209          - 129 _close.                                                           \
                    4       104        26.0       104        26.0
210          - 130 __sclose._close___errno_interstice                                \
                    4        24         6.0        24         6.0
211          - 131 __sclose.__errno___sclose_epilogue                                \
                    4        48        12.0        48        12.0
212          + 132 __libc_lock_release_recur.                                        \
                    4       209        52.3        81        20.3
213          - 133 __libc_lock_release_recur.__libc_lock_release_recur___asmp_rt_loc...\
                    4        41        10.3        41        10.3
214          - 134 __asmp_rt_lock_release_re.                                        \
                    4       128        32.0       128        32.0
215          - 135 __libc_lock_release_recur.__asmp_rt_lock_release_re___libc_lock_r...\
                    4        40        10.0        40        10.0
216          + 136 __libc_lock_close_recursi.                                        \
                    4       118        29.5        98        24.5
217          - 137 __libc_lock_close_recursi.__libc_lock_close_recursi___asmp_rt_loc...\
                    4        45        11.3        45        11.3
218          - 138 __asmp_rt_lock_destroy.                                           \
                    4        20         5.0        20         5.0
219          - 139 __libc_lock_close_recursi.__asmp_rt_lock_destroy___libc_lock_clos...\
                    4        53        13.3        53        13.3
220          + 140 __sfp_lock_release.                                               \
                    4       204        51.0        80        20.0
221          - 141 __sfp_lock_release.__sfp_lock_release___asmp_rt_lock_release_re_p...\
                    4        40        10.0        40        10.0
222          - 142 __asmp_rt_lock_release_re.                                        \
                    4       124        31.0       124        31.0
223          - 143 __sfp_lock_release.__asmp_rt_lock_release_re___sfp_lock_release_e...\
                    4        40        10.0        40        10.0
224          - 144 fclose.fclose___getreent_prologue                                 \
                    4        28         7.0        28         7.0
225          - 145 __getreent.                                                       \
                    4        36         9.0        36         9.0
226          - 146 fclose.__getreent___sfp_lock_acquire_interstice                   \
                    4        96        24.0        96        24.0
227          - 147 fclose.__sfp_lock_acquire___libc_lock_acquire_recur_interstice    \
                    4        71        17.8        71        17.8
228          - 148 fclose.__libc_lock_acquire_recur__fflush_r_interstice             \
                    4        51        12.8        51        12.8
```

```
229       - 149 fclose._fflush_r___sclose_interstice                                  \
                      4        60        15.0        60        15.0
230       - 150 fclose.__sclose___libc_lock_release_recur_interstice                  \
                      3        72        24.0        72        24.0
231       - 151 fclose.__libc_lock_release_recur___libc_lock_close_recursi_interstice\
                      4        12         3.0        12         3.0
232       - 152 fclose.__libc_lock_close_recursi___sfp_lock_release_interstice        \
                      4         8         2.0         8         2.0
233       - 153 fclose.__sfp_lock_release_fclose_epilogue                             \
                      4        56        14.0        56        14.0
234       + 154 _free_r.                                                              \
                      1       363       363.0       245       245.0
235        + 155 __malloc_lock.                                                       \
                      1        50        50.0        20        20.0
236         - 156 __malloc_lock.__malloc_lock___asmp_rt_lock_acquire_re_prologue      \
                      1        10        10.0        10        10.0
237         - 157 __asmp_rt_lock_acquire_re.                                        \
                      1        30        30.0        30        30.0
238         - 158 __malloc_lock.__asmp_rt_lock_acquire_re___malloc_lock_epilogue     \
                      1        10        10.0        10        10.0
239        - 159 _free_r._free_r___malloc_lock_prologue                             \
                      1        18        18.0        18        18.0
240        - 160 _free_r.__malloc_lock___divw_interstice                            \
                      1       156       156.0       156       156.0
241        - 161 __divw.                                                            \
                      1        36        36.0        36        36.0
242        - 162 _free_r.__divw___asmp_rt_lock_release_re_interstice                \
                      1        61        61.0        61        61.0
243        - 163 __asmp_rt_lock_release_re.                                         \
                      1        32        32.0        32        32.0
244        - 164 _free_r.__asmp_rt_lock_release_re__free_r_epilogue                 \
                      1        10        10.0        10        10.0
245       - 165 fclose.__sclose__free_r_interstice                                  \
                      1        34        34.0        34        34.0
246       - 166 fclose._free_r___libc_lock_release_recur_interstice                 \
                      1        52        52.0        52        52.0
247      + 167 __sfp_lock_release.                                                  \
                      1        52        52.0        20        20.0
248       - 168 __sfp_lock_release.__sfp_lock_release___asmp_rt_lock_release_re_pr...\
                      1        10        10.0        10        10.0
249       - 169 __asmp_rt_lock_release_re.                                          \
                      1        32        32.0        32        32.0
250       - 170 __sfp_lock_release.__asmp_rt_lock_release_re___sfp_lock_release_ep...\
                      1        10        10.0        10        10.0
251      - 171 _cleanup_r._cleanup_r___sfp_lock_acquire_prologue                    \
                      1        32        32.0        32        32.0
252      - 172 _cleanup_r.__sfp_lock_acquire_fclose_interstice                      \
                      1        64        64.0        64        64.0
253      - 173 _cleanup_r.fclose_fclose_interstice                                  \
                      3       104        34.7       104        34.7
254      - 174 _cleanup_r.fclose___sfp_lock_release_interstice                      \
                      1        81        81.0        81        81.0
255      - 175 _cleanup_r.__sfp_lock_release__cleanup_r_epilogue                    \
                      1        19        19.0        19        19.0
256     - 176 exit.exit___call_exitprocs_prologue                                   \
                      1        18        18.0        18        18.0
257     - 177 __call_exitprocs.                                                     \
                      1        90        90.0        90        90.0
258     - 178 exit.__call_exitprocs__cleanup_r_interstice                           \
                      1        11        11.0        11        11.0
259     - 179 exit._cleanup_r_exit_epilogue                                         \
                      1        35        35.0        35        35.0
260    - 180 __asmp_ind_crt0.__asmp_ind_crt0___syscall_prologue                     \
                      1       168       168.0       168       168.0
261    - 181 __syscall.                                                            \
                      1        43        43.0        43        43.0
262    - 182 __asmp_ind_crt0.__syscall___asmp_runtime_init_interstice               \
```

```
                      1        53        53.0        53        53.0
263     - 183 __asmp_ind_crt0.__asmp_runtime_init_main_interstice                  \
                      1        25        25.0        25        25.0
264     - 184 __asmp_ind_crt0.main___asmp_runtime_fini_interstice                  \
                      1        23        23.0        23        23.0
265     - 185 __asmp_runtime_fini.                                                 \
                      1        70        70.0        70        70.0
266     - 186 __asmp_ind_crt0.__asmp_runtime_fini_exit_interstice                  \
                      1         8         8.0         8         8.0
267     - 187 __asmp_ind_crt0.exit___asmp_ind_crt0_epilogue                        \
                      1         0         0.0         0         0.0
```

Appendix C

List of Tiling Parameters and Mathematical Definitions

Object Classes	Description
$\mathcal{I}$	An image class
$\mathcal{K}$	A kernel class
$\mathcal{T}$	A tile class
$\mathcal{T}^i$	The i^{th} tile class

Table C.1: *Tilana's object classes list and their description.*

Object Sets and Instances	Description
$\mathbf{I}$	An image instance
$\mathbb{T}$	A set of tiles $\mathbb{T}$
$\mathbb{T}^i$	The subset containing all tiles of class $\mathcal{T}^i$ in the $\mathbb{T}$ set
$\mathbf{T}$	A tile instance
$\mathbf{T}_i^j$	The j^{th} tile of type $\mathcal{T}^i$ in the $\mathbb{T}$ set

Table C.2: *Tilana's object sets and instances list and their description.*

Parameters and Variables	Description	Unit
t	A time value	Seconds
m_s	Memory size	Bytes

Table C.3: *Tilana's parameters and variables list and their description.*

Functions	Description	Unit
$\mathcal{A}(\mathrm{df}(\mathbf{T}))$	$\mathcal{A}$ Area of a **T**'s data footprint	Pixels^2
$\mathcal{A}(\mathrm{is}(\mathbf{T}))$	$\mathcal{A}$ Area of a **T**'s iteration space	Pixels^2
$\eta(\mathbb{T})$	Number of elements in a set $\mathbb{T}$	
$\mathcal{H}(i)$	Heaviside step function of i	
$\mathcal{R}(i)$	Ramp function of i	
$\max(i,j)$	Maximum between i and j	
$\min(i,j)$	Minimum between i and j	
$\mathrm{gap}(\mathcal{T},p)$	Gap time for tile class $\mathcal{T}$ on p processors	Seconds
$\mathrm{ii}(\mathcal{T},p)$	Initiation interval time for tile class $\mathcal{T}$ on p processors	Seconds
$\mathrm{is}(\mathcal{T})$	$2D$ iteration space of tile class $\mathcal{T}$	Iterations
$\mathrm{is}_x(\mathcal{T})$	Horizontal value of the $2D$ iteration space of a tile of class $\mathcal{T}$	Iterations
$\mathrm{is}_y(\mathcal{T})$	Vertical value of the iteration space of a tile of class $\mathcal{T}$	Iterations
$\mathrm{df}(\mathcal{T})$	$2D$ data footprint of a tile of class $\mathcal{T}$	Pixels
$\mathrm{df}_x(\mathcal{T})$	Horizontal data footprint of a tile of class **T**	Pixels
$\mathrm{df}_y(\mathcal{T})$	Vertical data footprint of a tile of class $\mathcal{T}$	Pixels
$\mathrm{sz}(\mathcal{T})$	$2D$ size of a tile of class $\mathcal{T}$	Pixels
$\mathrm{sz}_x(\mathcal{T})$	Width of a tile of class $\mathcal{T}$	Pixels
$\mathrm{sz}_y(\mathcal{T})$	Height of a tile of class $\mathcal{T}$	Pixels
$\mathrm{t}_{start}(\mathbf{T})$	Start timestamp of tile **T**	Seconds
$\mathrm{t}_{end}(\mathbf{T})$	End timestamp of tile **T**	Seconds
$\mathrm{t}_{exe}(\mathbf{T})$	Execution time (duration) of a given tile **T**	Seconds

Table C.4: *Tilana's function list and their description.*

Appendix D

List of State of the Art Vision Processors and MPSoCs

Analog Devices' Pipeline Vision Processor (PVP) [26] is a streaming engine that provides hardware implementations of a number of image and vision processing algorithms. It processes data directly from DMA streams. The hardware functions implemented are: 5x5 convolution, polar coordinate conversion, edge classification, arithmetic, threshold and integral operations, image scaling and data format conversion.

Analog Devices' BF609 [26] is an heterogeneous *Multiprocessor System-On-Chip* (MPSoC) for vision applications. It counts with two Freescale Blackfin processor cores and the Pipeline Vision Processor (PVP) streaming engine for vision processing. Programming is done in C/C++ and supports Analog Devices' VisualDSP++ operating system.

Apical's Spirit [204] and Assertive Vision Engines [17] are programmable hardware engines for real-time detection, classification and tracking of people and objects. They contain 16 dedicated classifier engines that provide information for higher level software layers running on an ARM Cortex-M4 CPU.

Cadence/Tensilica's IVP [195] is a *Very Long Instruction Word* (VLIW)/*Single Instruction Multiple Data* (SIMD) DSP core with specialized instructions for computer vision and pixel processing applications. Programming is done with a C/C++ compiler with automatic vectorization support.

CEVA's XM4 [45, 46] is a VLIW/SIMD vision processor IP developed by CEVA. It contains four scalar processing units, two load/store units and two vector processing units. As a SIMD processor, a single instruction word encodes the operations to be executed by all units. It can perform vector gather/scatter memory accesses with up to eight data points. Programming is done directly using a C compiler which supports vector intrinsics or auto-vectorization. It can be coupled to a host processor and used to accelerate calls to the CEVA-CV library, offloading them to the XM4. A service layer named SmartFrame automatically performs tiling and programs DMA data transfers between the host and the XM4.

CogniVue's APEX [142, 34] cores are massively parallel processors. The "Opus" APEX-1282 core counts with two Array Processor Units (APUs) each of which has an array control processor and 64 VLIW/SIMD compute units (CUs). APUs are interconnected with an AXI-like fabric to a shared data management engine with multi-channel and streaming DMAs.

CogniVue's CV220x Image Cognition Processor [51] is an heterogeneous multiprocessor platform with an ARM9 processor coupled to a CogniVue's APEX core for image processing acceleration.

Eutecus's Multi-Core Video Analytics Engine (MVE™) [66] is an heterogeneous accelerator for embedded video analytics. A 32-bit RISC processor executes the Instant Vision Embedded library and is coupled to C-MVA™and C-MVA-F™IP cores. The company states that C-MVA™and C-MVA-F™have a cascade of optimized processing blocks in massively biologically inspired parallel cellular array structures for low-level vision processing. The company provides the MVE™as

either an IP core for *Field Programmable Gate Array* (FPGA) implementation or integration on a clients' *Application-Specific Integrated Circuit* (ASIC).

Freescale's SCP2200 Image Cognition Processor [73] is a processor integrating CogniVue's APEX core and a series of peripherals in an ASIC package.

Freescale's S32V230 Platform [72, 202] is an heterogeneous MPSoC for *Advanced Driver Assistance Systems* (ADAS). It contains a mix of CPUs, GPUs, and image processors. It has a quad-core ARM Cortex-A53, an ARM Cortex-M4 microcontroller, an *Image Signal Processor* (ISP), a CogniVue's APEX core and Vivante's GC3000 GPU. It has a 4 MB scratchpad memory, as well as instruction and data caches, with *Error-Correcting Code* (ECC) features for safety. Programming relies on the OpenCL programming model.

Inuitive's NU3000 [98] is an heterogeneous multiprocessor with an ARM Cortex A5 host processor with Neon vector instructions support, two CEVA MM3101 vector DSPs and a proprietary 3D image multiprocessor.

MobileEye's EyeQ4 [49] is an heterogeneous platform IP with a diverse range of processor cores. It counts with four 32-bit InterAptiv MIPS CPUs from Imagination and ten specialized processors. These processors are six Vector Microcode Processors (VMPs), two Multithreaded Processing Cluster (MPC) cores, and two Programmable Macro Array (PMA) cores.

Movidius' Streaming Hybrid Architecture Vector Engine (SHAVE) [146] is a Movidius' 128-bit VLIW/SIMD vector processor with variable length instructions and two 64-bit memory ports.

Movidius' Myriad 2 Vision Processor [150, 146] is an heterogeneous multiprocessor architecture based on twelve Movidius' 128-bit vector VLIW "SHAVE" processors optimized for vision processing workloads, a number of configurable hardware accelerators for image and vision processing, two 32-bit *Reduced Instruction Set Computer* (RISC) processors, with a 2 MB shared memory.

Synopsys' DesignWare EV5x [192] is a vision multiprocessor IP. It counts with a quad-core Synopsys ARC CPU and up to eight hardware object detection engines. EV5x use a shared memory architecture and have a single DMA with an external AXI interconnection. Programming relies on OpenCV and OpenVX libraries.

Texas Instruments' AccelerationPac [131] with up to four *Embedded Vision Engines* (EVEs) for lower level vision processing. Each EVE unit has a 32-bit RISC core and a vector coprocessor interconnected to three 32 KB memory banks with 256-bit width data ports, plus a dedicated DMA.

Texas Instruments TDA3x [197, 198] is an MPSoC targeted at ADAS applications. It contains: two ARM Cortex A-15 processors for control level processing; four ARM Cortex M4 processors for high level processing; two C66x DSPs for signal processing; and a Vision AccelerationPac. Programming is based on a "Links and Chains" framework [48] for interprocessor communication via message passing.

Toshiba's MPEs [203] is an homogeneous multiprocessor composed of multiple Toshiba's MeP scalar processors, each with a dedicated floating-point VLIW/SIMD co-processor.

Toshiba's TMPV760 series [203] are image recognition processors. The TMPV7608XBG model counts with two 32-bit Toshiba MeP RISC control CPUs, an MPE multiprocessor and 14 dedicated hardware accelerators for common vision processing functions.

Videantis v-MP4000HDX [209] is a multiprocessor IP core architecture. It is an heterogeneous architecture which combines multiple VLIW/SIMD media processors (v-MPs) for vision processing and stream processors (v-SPs) for video bitstream encoding/decoding. The v-MP has three dual-issue 64-bit or single-issue 128-bit VLIW/SIMD units, a local memory and a dedicated multi-channel DMA engine. No data cache is present in the architecture.

Glossary

ACF	APEX Core Framework
ADAS	Advanced Driver Assistance Systems
API	Application Programming Interface
APU	Accelerated Processing Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
ASMP	Application-Specific Multiprocessor
CEA	Atomic Energy and Alternative Energies Commission
CMP	Chip Multiprocessor
CP	Constraint Programming
CPA	Critical Path Analysis
CPU	Central Processing Unit
DBI	Dynamic Binary Instrumentation
DBT	Dynamic Binary Translation
DLP	Data Level Parallelism
DSE	Design Space Exploration
DSP	Digital Signal Processor
DVFS	Digital Voltage and Frequency Scaling
ECC	Error-Correcting Code
ED	Edge Detection
EPCC	Edinburgh Parallel Computing Center
EVA	Embedded Vision Alliance
EVE	Embedded Vision Engine
FAST	Fast Features for Accelerated Segment Test
FIFO	"First In, First Out"
FPGA	Field Programmable Gate Array
GPGPU	General-Purpose computing on Graphics Processing Units
HCPA	Hierarchical Critical Path Analysis
HLS	High-Level Synthesis
HPC	High-Performance Computing
HSA	Heterogeneous System Architecture
HSAIL	HSA Intermediate Layer
IDE	Integrated Development Environment
LP	Linear Programming
ILP	Instruction Level Parallelism
IoT	Internet of Things
IPC	Instructions per Cycle
IS	Integer Sort
ISA	Instruction Set Architecture
ISP	Image Signal Processor
ISS	Instruction Set Simulator
IVSS	Intelligent Video Surveillance Systems
ITS	Intelligent Transportation Systems
JiT	Just in Time
LLVM	Low Level Virtual Machine
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
MPMD	Multiple Program Multiple Data
MPPA	Multi-Purpose Processor Array
MPSoC	Multiprocessor System-On-Chip
NoC	Network On Chip

NAS	NASA Advanced Supercomputing Division
NPB	*NASA Advanced Supercomputing Division* (NAS) Parallel Benchmark
NUMA	Non-Uniform Memory Access
ompP	OpenMP Profiler
PE	Processing Element
PLP	Pipeline Level Parallelism
RISC	Reduced Instruction Set Computer
SDK	Software Development Kit
SHMEM	Shared Memory
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SMP	Symmetric Multiprocessing
SPMD	Single Program Multiple Data
TBB	Thread Building Blocks
TLM	Transaction Level Modeling
TLP	Task Level Parallelism
UMA	Uniform Memory Access
VAL	Video Analytics Library
VLIW	Very Long Instruction Word

Publications and Other Scientific Activities

Refereed Conferences and Workshops

[1] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. Fast Parallel Application and Multiprocessor Design Space Exploration from Sequential Code. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Amsterdam, Netherlands. Forthcoming, 2015. ***[Accepted]***

[2] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. Image Tiling for Embedded Applications with Non-Linear Constraints. Submitted to the *Conference on Design & Architectures for Signal & Image Processing (DASIP 2015)*, Cracow, Poland. Forthcoming, 2015. ***[Accepted]***

[3] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. Estimation rapide et précise de l'accélération d'applications séquentielles sur des multiprocesseurs embarqués. In *Proceedings of the Conférence en Parallélisme, Architecture et Système (ComPAS'2015)*, Lille, France. ***[Published]***

[4] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. (2015). Estimating the Potential Speedup of Computer Vision Applications on Embedded Multiprocessors. In *Proceedings of the DATE Friday Workshop on Heterogeneous Architectures and Design Methods for Embedded Image Systems (HIS 2015)* on arXiv preprint arXiv:1502.07446. ***[Published]***

[5] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. (2014). Application-level Performance Optimization: A Computer Vision Case Study on STHORM. In *Proceedings of the International Conference on Computational Science (ICCS) Workshop on Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems (ALCHEMY)*. Procedia Computer Science, 29, 1113–1122. ***[Published]***

Patents

[6] Talayssat, J., Cleyet-Merle, S., **Schwambach, V.** Procédé et dispositif de génération d'une représentation multi-résolutions d'une image et application à la détection d'objet utilisant une fenêtre de détection. FR. Patent 1553461, filed April 17, 2015.

Poster Presentations

[7] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. (2015). Parana: Fast Parallel Application and Multiprocessor Design Space Exploration from Sequential Code. Poster presented at the *Design Automation Conference (DAC) Work-in-Progress Session*, San Francisco, CA, USA.

[8] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. (2015). Estimating the Potential Speedup of Computer Vision Applications on Embedded Multiprocessors. Poster presented at the *18ème Journées Nationales du Réseau Doctoral en Micro-nanoélectronique (JNRDM)*, Bordeaux, France.

[9] **Schwambach, V.**, Cleyet-Merle, S., Issard, A., & Mancini, S. (2014). Optimisation de performance au niveau applicatif : étude de cas de vision embarquée sur STHORM. Poster presented at the *Conférence en Parallélisme, Architecture et Système (ComPAS'2014)*, Neuchâtel, Switzerland.

References

[10] Heterogeneous System Architecture (HSA) Foundation. URL http://www.hsafoundation.com/.

[11] Message Passing Interface (MPI) Forum. URL http://www.mpi-forum.org/.

[12] The OpenMP® API specification for parallel programming. URL http://www.openmp.org.

[13] The Open Group Base Specifications Issue 7 IEEE Std 1003.1™, 2013 Edition. URL http://pubs.opengroup.org/onlinepubs/9699919799.

[14] Be a Hero, Use R-Stream. URL https://www.reservoir.com/product/r-stream/.

[15] VXL - C++ Libraries for Computer Vision Research and Implementation. URL http://vxl.sourceforge.net/.

[16] IEEE-ISTO 5001™-2003, the Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface Version 2, 2003.

[17] Apical Unveils the Assertive Engine Embedded Vision Processor Core, 2013. URL http://www.embedded-vision.com/industry-analysis/video-interviews-demos/2013/10/13/apical-unveils-assertive-engine-embedded-vision-.

[18] Embedded Vision Alliance, 2015. URL http://www.embedded-vision.com.

[19] What is Embedded Vision?, 2015. URL http://www.embedded-vision.com/what-is-embedded-vision.

[20] P. Agrawal, P. Raghavan, M. Hartman, N. Sharma, L. der Perre, and F. Catthoor. Early exploration for platform architecture instantiation with multi-mode application partitioning. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–8, New York, New York, USA, 2013. IEEE, ACM Press. ISBN 9781450320719. doi: 10.1145/2463209.2488896. URL http://dl.acm.org/citation.cfm?doid=2463209.2488896.

[21] B. M. Al-babtain, F. J. Al-kanderi, M. F. Al-fahad, and I. Ahmad. A Survey on Amdahl's Law Extension in Multicore Architectures. *International Journal of New Computer Architectures and their Applications (IJNCAA)*, 3(3):30–46, 2013.

[22] D. Alessandrelli, A. Azzarà, M. Petracca, C. Nastasi, and P. Pagano. ScanTraffic: smart camera network for traffic information collection. In *Wireless Sensor Networks*, pages 196–211. Springer, 2012.

[23] C. Alias, A. Darte, and A. Plesco. Optimizing remote accesses for offloaded kernels. *ACM SIGPLAN Notices*, 47(8):285, 2012. ISSN 03621340. doi: 10.1145/2370036.2145856.

[24] Allied Business Intelligence Inc. Volvo and Mercedes-Benz Driving Roll Out of ADAS as Standard Equipment in Cars, 2013. URL https://www.abiresearch.com/press/volvo-and-mercedes-benz-driving-roll-out-of-adas-a/.

[25] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 1–4, 1967. URL http://dl.acm.org/citation.cfm?id=1465560.

[26] Analog Devices. Blackfin Dual Core Embedded Processor ADSP-BF606/ADSP-BF607/ADSP-BF608/ADSP-BF609 Datasheet. 2014.

[27] R. Andonov, S. Rajopadhye, and N. Yanev. Optimal orthogonal tiling. In *Euro-Par'98 Parallel Processing*, pages 480–490. Springer, 1998.

[28] ARM. CoreSight Debug and Trace. URL `https://www.arm.com/products/system-ip/debug-trace/index.php`.

[29] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, and Others. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, oct 2009. ISSN 00010782. doi: 10.1145/1562764.1562783. URL `http://cacm.acm.org/magazines/2009/10/42368-a-view-of-the-parallel-computing-landscape/fulltexthttp://portal.acm.org/citation.cfm?doid=1562764.1562783`.

[30] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. *Shared Memory Parallel Programming, R. Eigenmann and M. Voss, Eds.*, 2104:1–19, 2001. doi: 10.1007/3-540-44587-0{_}1. URL `http://link.springer.com/chapter/10.1007/3-540-44587-0{_}1`.

[31] Y. B. Atitallah, J. Mottin, N. Hili, T. Ducroux, G. Godet-bar, Y. B. Atitallah, J. Mottin, N. Hili, T. Ducroux, and G. G.-b. A. A Power Consumption Estimation Approach for Embedded Software Design using Trace Analysis. In *Euromicro Conference series on Software Engineering and Advanced Applications (SEAA)*, 2015.

[32] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 342–351. ACM, 1992.

[33] Axxon. Smart traffic control. axxonsmarttraffic. URL `http://www.axxonsoft.com/integrated{_}security{_}solutions/lpr/atcs.php`.

[34] BDTI. CogniVue's "Opus" APEX Generation 3: Vision Processing With Implementation Flexibility, 2015.

[35] F. Bellard. QEMU , a Fast and Portable Dynamic Translator. *USENIX Annual Technical Conference. Proceedings of the 2005 Conference on*, pages 41–46, 2005. ISSN 0014-2999. URL `https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/freenix/bellard.html`.

[36] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012. ISBN 9783981080186. URL `http://dl.acm.org/citation.cfm?id=2492954`.

[37] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. *5th Annual Workshop on Modeling, Benchmarking and Simulation*, pages 1–9, 2009. doi: 10.1145/1454115.1454128. URL `http://www-mount.ece.umn.edu/{~}jjyi/MoBS/2009/program/02E-Bienia.pdf`.

[38] C. Bienia, S. Kumar, J. J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. *... of the 17th international conference ...*, pages 72–81, 2008. URL `http://dl.acm.org/citation.cfm?id=1454128`.

[39] J. Bier. How New Chips Are Enabling the Proliferation of Machines That See 2011 IMS Intelligent Video Conference. 1(510):1–18, 2011.

[40] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, pages 132–146. Springer, 2008.

[41] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. PLuTo: A practical and fully automatic polyhedral program optimization system. *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*, pages 1–15, 2008.

[42] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[43] J. M. Bull and D. O'Neill. A microbenchmark suite for OpenMP 2.0. *ACM SIGARCH Computer Architecture News*, 29(5):41–48, 2001. ISSN 01635964. doi: 10.1145/563647.563656.

[44] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, pages 1–12, 2013. ISSN 21674337. doi: 10.1145/2503210.2503277. URL `http://dl.acm.org/citation.cfm?doid=2503210.2503277`.

[45] CEVA. CEVA-XM4 - Intelligent Vision Processor. URL `http://www.ceva-dsp.com/CEVA-XM4`.

[46] CEVA. CEVA-XM4™ Intelligent Vision Processor (White Paper), 2015.

[47] C.-c. C.-F. Chen, Y.-c. Peng, C.-c. C.-F. Chen, W.-s. Wu, Q. Min, C. Ye, T. Che, P.-C. Yew, W. Zhang, and T.-F. Chen. DAPs: Dynamic Adjustment and Partial Sampling for Multi-threaded/Multicore Simulation. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014. ISBN 9781450327305.

[48] K. Chitnis, R. Staszewski, and G. Agarwal. TI Vision SDK, Optimized Vision Libraries for ADAS Systems (White Paper).

[49] P. Clarke. Mobileye's Next Vision Processor Targets Autonomous Driving, 2015. URL `http://electronics360.globalspec.com/article/5088/mobileye-s-next-vision-processor-targets-autonomous-driving`.

[50] Cognex. Cognex Vision Library (CVL). URL `http://www.cognex.com/products/machine-vision/cognex-vision-library/`.

[51] Cognivue. CV220x (G1-APEX SoC). URL `http://temp.cognivue.com/CV220x.php`.

[52] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Multithreaded Algorithms. In *Introduction to Algorithms*, pages 772–812. The MIT Press, third edit edition, 2009. doi: 10.1007/BF02837777.

[53] J. Cownie. Building High Performance Threaded Applications using Libraries. In *Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[54] C. F. Craver. When mechanistic models explain. *Synthese*, 153(3):355–376, 2006. ISSN 00397857. doi: 10.1007/s11229-006-9097-x.

[55] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*, volume 7. Gulf Professional Publishing, 1999. ISBN 1558603433. doi: 10.1109/MCC.1999.766975.

[56] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc. Balance principles for algorithm-architecture co-design. *USENIX Wkshp. Hot Topics in Parallelism (HotPar)*, pages 1–5, 2011. URL http://www.usenix.org/events/hotpar11/tech/final{_}files/Czechowski.pdf.

[57] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–496, 1997.

[58] A. Darte, L. Khachiyan, and Y. Robert. Linear Scheduling Is Nearly Optimal. *Parallel Processing Letters*, 1(02):73–81, 1991. ISSN 0129-6264. doi: 10.1142/S0129626491000021.

[59] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, T. Strudel, B. D. D. Dinechin, and P. G. D. Massas. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. *2013 International Conference on Computational Science. Procedia Computer Science.*, 18(0):1654–1663, jan 2013. ISSN 1877-0509. doi: 10.1016/j.procs.2013.05.333. URL http://linkinghub.elsevier.com/retrieve/pii/S1877050913004766.

[60] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. Array-OL: Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *15ème Colloque GRETSI sur le traitement du signal et des images*. GRETSI, Groupe d'Etudes du Traitement du Signal et des Images, 1995. ISBN 9780874216561. doi: 10.1007/s13398-014-0173-7.2. URL http://www.ncbi.nlm.nih.gov/pubmed/15003161http://cid.oxfordjournals.org/lookup/doi/10.1093/cid/cir991http://www.scielo.cl/pdf/udecada/v15n26/art06.pdfhttp://www.scopus.com/inward/record.url?eid=2-s2.0-84861150233{&}partnerID=tZOtx3y1.

[61] R. H. Dennard, F. H. Gaensslen, H.-N. YU, V. L. RIDEOUT, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974. URL http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=1050511.

[62] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos. A Microbenchmark Study of OpenMP Overheads under Nested Parallelism. In *International Workshop on OpenMP (IWOMP)*, pages 1–12, 2008. doi: 10.1007/978-3-540-79561-2{_}1. URL http://link.springer.com/chapter/10.1007/978-3-540-79561-2{_}1.

[63] B. Dipert and A. Shoham. Eye, robot: Embedded vision, the next big thing in digital signal processing. *IEEE Solid-State Circuits Magazine*, 4(2):26–29, 2012. ISSN 19430582. doi: 10.1109/MSSC.2012.2193077.

[64] T. Ducroux, G. Haugou, V. Risson, and P. Vivet. Fast and accurate power annotated simulation: Application to a many-core architecture. *2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation, PATMOS 2013*, pages 191–198, 2013. doi: 10.1109/PATMOS.2013.6662173.

[65] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona openMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openMP. *Proceedings of the International Conference on Parallel Processing*, pages 124–131, 2009. ISSN 01903918. doi: 10.1109/ICPP.2009.64.

[66] Eutecus. Multi-Core Video Analytics Engine (MVE™). URL https://eutecus.com/mve-multi-core-video-analytics-engine.

[67] P. Feautrier. Some efficient solutions to the affine scheduling problem, Part I: One-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[68] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992.

[69] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103. Springer, 1996.

[70] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[71] B. Flyvbjerg. Five misunderstandings about case-study Research. *Qualitative Inquiry*, 12(2):219–245, 2006. ISSN 1077-8004. doi: 10.1177/1077800405284363. URL http://qix.sagepub.com/cgi/doi/10.1177/1077800405284363.

[72] Freescale. S32V230 Family of Processors for Advanced Driver Assistance Systems, . URL http://www.freescale.com/products/arm-processors/s32-processors-and-microcontrollers/s32v230-family-of-processors-for-advanced-driver-assistance-systems:S32V230.

[73] Freescale. SCP2200: Image Cognition Processors, . URL http://www.freescale.com/products/more-processors/32-bit-mcu-and-mcp/image-cognition-processors/image-cognition-processors:SCP2200.

[74] Y. Freund and R. E. Schapire. A decision theoretic generalization of on-line learning and an application to boosting. *Computer Systems Science*, 57:119–139, 1997.

[75] K. Fürlinger and M. Gerndt. ompP: A profiling tool for OpenMP. In *OpenMP Shared Memory Parallel Programming*, pages 15–23. Springer, 2008.

[76] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010. ISSN 15320626. doi: 10.1002/cpe.1556.

[77] M. Gerndt, K. Fürlinger, and E. Kereku. Periscope: Advanced Techniques for Performance Analysis. In *PARCO*, volume 33, pages 15–26, 2005. ISBN 3000173528.

[78] R. C. Gonzalez and R. E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 013168728X.

[79] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000. ISSN 08857458. doi: 10.1023/A:1007516818651.

[80] T. Grosser, A. Cohen, P. H. J. Kelly, S. Verdoolaege, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 24–31. ACM, 2013. ISBN 9781450320177. doi: 10.1145/2458523.2458526. URL http://dl.acm.org/citation.cfm?id=2458523.2458526{&}coll=DL{&}dl=ACM{&}CFID=221581876{&}CFTOKEN=87697892.

[81] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[82] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 145–156. ACM, 2010. ISBN 9781450300797. doi: 10.1145/1810479.1810509. URL http://dl.acm.org/citation.cfm?id=1810479.1810509.

[83] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 38–49. IEEE, Ieee, nov 2011. ISBN 978-1-4577-2064-2. doi: 10.1109/IISWC.2011.6114195. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6114195.

[84] C. Herglotz, J. Seiler, A. Kaup, A. Hendricks, M. Reichenbach, and D. Fey. Estimation of Non-functional Properties for Embedded Hardware with Application to Image Processing. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 190–195, 2015. doi: 10.1109/IPDPSW.2015.58. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7284308.

[85] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7): 33–38, 2008. URL http://www.youtube.com/watch?v=KfgWmQpzD74.

[86] HSA Foundation. HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG), 2013.

[87] HSA Foundation. HSA Platform System Architecture Specification, 2014.

[88] HSA Foundation. HSA Runtime Programmer's Reference Manual, 2015.

[89] IBM. Cell Broadband Engine Architecture Version 1.02, .

[90] IBM. The Cell Project, . URL www.research.ibm.com/cell/.

[91] K. Ievgen. Tile-based image processing, 2014. URL http://computer-vision-talks.com/articles/tile-based-image-processing/.

[92] Intel. Intel Cilk Plus, . URL https://www.cilkplus.org/.

[93] Intel. Intel® VTune™ Amplifier 2016, . URL http://www.intel.com/software/products/vtune/.

[94] Intel. Intel Thread Building Blocks (Intel TBB) 4.4, . URL https://www.threadingbuildingblocks.org.

[95] Intel. Intel® Threading Building Blocks Design Patterns, .

[96] Intel. Intel® Threading Building Blocks Reference Manual, .

[97] Intel Corporation. Intel Advisor XE. \url{https://software.intel.com/en-us/intel-advisor-xe}, last accessed in November 2014. URL https://software.intel.com/en-us/intel-advisor-xe(LastaccessedonNovember17,2014).

[98] Inuitive. 3D Imaging & Computer Vision Multi-Core Processor. URL http://inuitive-tech.com.

[99] F. Irigoin. Tiling. In *Encyclopedia of Parallel Computing*, pages 2040–2049. Springer, 2011.

[100] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988.

[101] Y. Ishikawa, H. Uetani, K. Funaoka, N. Tojo, H. Matsuzaki, N. Matsumoto, and T. Tokuyoshi. Rapid Development of Embedded Image Recognition Software Using Cross Performance Estimation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). Designer Track: Embedded System Models, Design and Applications*, 2013.

[102] ITRS. International Technology Roadmap for Semiconductors. \url{http://www.itrs.net}, 2011.

[103] Y. Janin, V. Bertin, H. Chauvet, T. Deruyter, C. Eichwald, O.-A. Giraud, V. Lorquet, and T. Thery. Designing tightly-coupled extension units for the stxp70 processor. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1052–1053. EDA Consortium, 2013.

[104] D. Jeon. Parallel Speedup Estimates for Serial Programs. 2012.

[105] D. Jeon, S. Garcia, C. Louie, and M. Taylor. Kismet: parallel speedup estimates for serial programs. *ACM SIGPLAN Notices*, 2011. URL http://dl.acm.org/citation.cfm?id=2048108.

[106] D. Jeon, S. Garcia, C. Louie, and M. Taylor. Parkour: Parallel speedup estimates for serial programs. *... of the USENIX workshop on Hot ...*, 2011. URL https://www.usenix.org/legacy/event/hotpar11/tech/final{_}files/Jeon.pdf?origin=publication{_}detail.

[107] D. Jeon, S. Garcia, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *ACM SIGPLAN Notices*, volume 46, pages 458–469. ACM, 2011. ISBN 9781450301190. URL http://dl.acm.org/citation.cfm?id=1941595.

[108] H. Jin, M. Frumkin, J. Yan, and M. Field. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report October, Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[109] M. Kanellos. Moore's Law to roll on for another decade. *CNET*, 2003. URL http://www.cnet.com/news/moores-law-to-roll-on-for-another-decade/.

[110] D. Katzmeier. Samsung Smart Interaction: Hands-on with voice and gesture control, 2012. URL http://www.cnet.com/news/samsung-smart-interaction-hands-on-with-voice-and-gesture-control.

[111] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. *HotOS XIII*, 2011.

[112] Khronos Group. OpenCL: The open standard for parallel programming of heterogeneous systems, . URL https://www.khronos.org/opencl.

[113] Khronos Group. OpenVX: Portable, Power-efficient Vision Processing, . URL https://www.khronos.org/openvx/.

[114] Khronos OpenCL Working Group. The OpenCL Specification. *Version 1.1, Document Revision: 44*, 2010.

[115] Khronos OpenCL Working Group. The OpenCL Specification. *Version 2.0, Document Revision: 19*, 2013.

[116] Khronos Vision Working Group, S. Kyo, T. Lepley, E. Rainey, and F. Brill. The OpenVX ™ User Node Tiling Extension Version 1.0, 2013.

[117] Khronos Vision Working Group, S. Gautam, and E. Rainey. The OpenVX ™ Specification Version 1.0.1, 2014.

[118] M. Kim, H. Kim, and C.-K. Luk. Prospector: A dynamic data-dependence profiler to help parallel programming. *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010. URL http://static.usenix.org/event/hotpar10/final{_}posters/Kim.pdf.

[119] M. Kim, P. Kumar, H. Kim, and B. Brett. Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model. *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1318–1329, may 2012. doi: 10.1109/IPDPS.2012.128. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6267933.

[120] B. Kisacanin. Integral Image Optimizations for Embedded Vision Applications. *Image Analysis and Interpretation, 2008. SSIAI 2008. IEEE Southwest Symposium on*, (1):181–184, 2008.

[121] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[122] A. Knüpfer, C. Rössel, and D. an Mey. Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. *Tools for High Performance Computing 2011*, pages 1–12, 2012. doi: 10.1007/978-3-642-31476-6. URL http://link.springer.com/chapter/10.1007/978-3-642-31476-6{_}7.

[123] S. B. Kotsiantis. Decision trees: a recent overview. *Artificial Intelligence Review*, 39(4): 261–283, 2013.

[124] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *Computers, IEEE Transactions on*, 37(9):1088–1098, 1988. ISSN 00189340. doi: 10.1109/12.2259.

[125] C. H. Lampert, M. B. Blaschko, and T. Hofmann. Beyond sliding windows: Object localization by efficient subwindow search. *26th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2008. ISSN 1063-6919. doi: 10.1109/CVPR.2008.4587586.

[126] Z. Larabi, Y. Mathieu, and S. Mancini. Efficient data access management for FPGA-Based image processing SoCs. In *20th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'2009)*, pages 159–165, Washington, DC, USA, 2009. IEEE Computer Society.

[127] J. J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *Parallel and Distributed Systems, IEEE Transactions on*, 4(7):812–826, 1993. ISSN 10459219. doi: 10.1109/71.238302. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=238302.

[128] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. a. McKee. Methods of inference and learning for performance modeling of parallel applications. *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '07*, page 249, 2007. doi: 10.1145/1229428.1229479. URL http://portal.acm.org/citation.cfm?doid=1229428.1229479.

[129] B. C. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 270–281, 2008. ISSN 1072-4451. doi: 10.1109/MICRO.2008.4771797. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4771797.

[130] B. C. B. Lee and D. D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGARCH Computer Architecture News*, 34(5):185, 2006. ISSN 01635964. doi: 10.1145/1168917.1168881. URL http://portal.acm.org/citation.cfm?id=1168881.

[131] Z. Lin, J. Sankaran, and T. Flanagan. Empowering automotive vision with TI's Vision AccelerationPac (White Paper).

[132] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.31.

[133] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005. ISBN 1595930809. doi: 10.1145/1065010.1065034. URL http://dl.acm.org/citation.cfm?id=1065010.1065034.

[134] C. Mack. The Multiple Lives of Moore's Law. *IEEE Spectrum*, 52(4):31–31, 2015. ISSN 0018-9235. doi: 10.1109/MSPEC.2015.7065415. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7065415.

[135] S. Mancini, N. Gac, M. Desvignes, O. Bourrion, and O. Rosseto. Application d'un cache 2D prédictif à l'accélération de la rétroprojection TEP 2D. In *Gretsi*, volume 23, pages 391–404, 2006.

[136] P. Maragos and R. Schafer. Morphological skeleton representation and coding of binary images. *ICASSP '84. IEEE International Conference on Acoustics, Speech, and Signal Processing*, 9, 1984. ISSN 0096-3518. doi: 10.1109/ICASSP.1984.1172472.

[137] P. Marchand and L. Marmet. Binomial smoothing filter: A way to avoid some pitfalls of least-squares polynomial smoothing. *Review of scientific instruments*, 54(8):1034–1041, 1983.

[138] G. Martin. NEXUS 5001 Forum Debug Interface Standard. In *Embedded Systems Show*, 2000.

[139] Matrox Imaging. Matrox Imaging Library (MIL). URL http://www.matrox.com/imaging/fr/products/software/mil/.

[140] S. McCanne and C. Torek. A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling. In *Proceedings of the Winter 1993 USENIX Conference: January*, pages 25–29. Citeseer, 1993. URL papers://2cd573f8-44b1-4938-8484-cf0e6dcd735b/Paper/p525.

[141] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1137–1142. ACM, 2012. ISBN 9781450311991. URL http://dl.acm.org/citation.cfm?id=2228568.

[142] R. Merritt. Vision Core Tightens its Focus, 2015. URL http://www.eetimes.com/document.asp?doc{_}id=1326577.

[143] R. Mijat. Better Trace for Better Software (White Paper), 2010.

[144] Mobileye. Mobileye Pedestrian Collision Warning (PCW). URL http://www.mobileye.com/technology/applications/pedestrian-detection/pedestrian-collision-warning/.

[145] B. Mohr, A. D. Malony, S. S. Shende, and F. Wolf. Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, 2001.

[146] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe. Myriad 2: Eye of the Computational Vision Storm. In *HotChips 2016*, 2014.

[147] G. E. Moore. Cramming more components onto integrated circuits. 38(8), 1965.

[148] T. Moseley, D. Grunwald, D. a. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. LoopProf : Dynamic Techniques for Loop Detection and Profiling. *WBIA '06 - Workshop on Binary Instrumentation and Applications*, 2006.

[149] J. Mottin, M. Cartron, and G. Urlini. The STHORM Platform. In *Smart Multicore Embedded Systems*, pages 35–43. Springer, 2014.

[150] Movidius. Myriad 2 Vision Processor Product Brief, 2014.

[151] P. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. *Proc. Dept. of Defense HPCMP Users Group Conference*, 32:7–10, 1999. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.6801{&}amp;rep=rep1{&}amp;type=pdf`.

[152] R. T. Mullapudi and U. Bondhugula. Tiling for Dynamic Scheduling. In *IMPACT 2014. Fourth International Workshop on Polyhedral Compilation Techniques. In conjunction with HiPEAC*, pages 1–9. IMPACT, 2014.

[153] R. T. Mullapudi, V. Vasista, and U. Bondhugula. PolyMage : Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–443. ACM, 2015.

[154] MVTec Software GmbH. Halcon: The Power of Machine Vision. URL `http://www.halcon.com/`.

[155] R. Nambiar and M. Poess. Transaction performance vs. Moore's law: a trend analysis. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 110–120. Springer, 2011.

[156] Nest. Nest Cam. URL `https://nest.com/camera/meet-nest-cam`.

[157] Netatmo. Netatmo Welcome: Home Security Camera with Face Recognition. URL `https://www.netatmo.com/en-US/product/camera`.

[158] Netatmo. Netatmo unveils Welcome, the camera that recognizes each family member (Press Release), 2015.

[159] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 2.5, may 2005. URL `http://www.openmp.org/mp-documents/spec25.pdf`.

[160] OpenMP Architecture Review Board. OpenMP Application Program Interface Examples Version 4.0.0, 2013.

[161] OpenMP Architecture Review Board, O. Architecture, and R. Board. OpenMP Application Program Interface Version 3.0. (May), 2008.

[162] P. J. Joseph, K. Vaswani, and M. T. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, 2006. URL `http://csa.iisc.ernet.in/TR/2005/16/IISc-CSA-TR-2005-16.pdf`.

[163] P. Paulin. Programming challenges \& solutions for multi-processor SoCs: an industrial perspective. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 262–267. IEEE, 2011. ISBN 9781450306362.

[164] J. a. Poovey, T. M. Conte, M. Levy, and S. Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, 2009. ISSN 02721732. doi: 10.1109/MM.2009.74.

[165] C. Prud'homme and J. G. Fages. An introduction to Choco 3.0. 2013.

[166] C. Prud'homme, X. Lorca, and N. Jussien. Explanation guided large neighborhood search. *CP Doctoral Program*, pages 25–30, 2013.

[167] Qualcomm Technologies Inc. FastCV Computer Vision SDK. URL `https://developer.qualcomm.com/software/fastcv-sdk`.

[168] J. Ragan-Kelley. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing*. PhD thesis, 2014.

[169] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):1–12, jul 2012. ISSN 07300301. doi: 10.1145/2185520.2335383. URL `http://dl.acm.org/citation.cfm?doid=2185520.2335383`.

[170] J. Ragan-Kelley, A. Adams, S. Paris, F. Durand, C. Barnes, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 519–530, 2013. ISSN 03621340. doi: 10.1145/2491956.2462176. URL `http://doi.acm.org/10.1145/2491956.2462176`.

[171] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters. In *Design, Automation \& Test in Europe Conference \& Exhibition*, 2011. ISBN 9783981080179.

[172] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill. Addressing System-Level Optimization with OpenVX Graphs. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, pages 658–663. IEEE, 2014. ISBN 978-1-4799-4308-1. doi: 10.1109/CVPRW.2014.100. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6910050`.

[173] B. Rinner and W. Wolf. An introduction to distributed smart cameras, 2008. ISSN 00189219.

[174] A. D. Robison. Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST*, 18, 2012.

[175] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. *Computer Vision–ECCV 2006*, pages 1–14, 2006. URL `http://link.springer.com/chapter/10.1007/11744023{_}34`.

[176] M. A. Saeed, P. Maier, P. Trinder, L. Georgieva, and P. Maier. Critical Analysis of Parallel Functional Profilers. pages 1–65, 2013.

[177] S. Saidi, P. Tendulkar, T. Lepley, and O. Maler. Optimal 2D Data Partitioning for DMA Transfers on MPSoCs. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, number Program 1, pages 584–591. IEEE, 2012. URL `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=6386945`.

[178] D. Sarokin. What Is a Mechanistic Model? URL `http://smallbusiness.chron.com/mechanistic-model-12706.html`.

[179] R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee. Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods. *The Annals of Statistics*, 26(5):1651–1686, 1998. ISSN 00905364. doi: 10.1214/aos/1024691352.

[180] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson. The Cilkprof Scalability Profiler. *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures - SPAA '15*, pages 89–100, 2015. doi: 10.1145/2755573.2755603. URL http://dl.acm.org/citation.cfm?id=2755573.2755603.

[181] H. Schneiderman and T. Kanade. A statistical method for 3D object detection applied to faces and cars. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 1, pages 746–751. IEEE, 2000.

[182] V. Schwambach. Optimization of a Face Detection Algorithm for Real-Time Mobile-Phone Applications, 2009.

[183] V. Schwambach, S. Cleyet-Merle, A. Issard, and S. Mancini. Application-level Performance Optimization: A Computer Vision Case Study on STHORM. *Procedia Computer Science*, 29:1113–1122, 2014. ISSN 18770509. doi: 10.1016/j.procs.2014.05.100. URL http://www.sciencedirect.com/science/article/pii/S1877050914002774.

[184] E. Schweitz, R. Lethin, A. Leung, and B. Meister. R-stream: A parametric high level compiler. *Proceedings of HPEC*, sep 2006.

[185] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006. ISSN 1094-3420. doi: 10.1177/1094342006064482.

[186] J. P. Singh, W.-d. Weber, and A. Gupta. SPLASH : STANFORD PARALLEL APPLICATIONS FOR SHARED-MEMORY Jaswinder Pal Singh Wolf-Dietrich Weber Anoop Gupta Technical Report No . CSL-TR-91-469. (April), 1991.

[187] W. Sosa-Escudero. Extensions of the Linear Model. Technical report, Introduction to Applied Econometrics (Econ 471), University of Illinois at Urbana-Champaign, 2009.

[188] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *Proceedings of the 48th Design Automation Conference*, pages 486–491. ACM, 2011.

[189] B. Stefanizzi. The Programmer's Guide to a Universe of Possibility. In *AMD Fusion Developper Summit*, 2012.

[190] P. Sudowe and B. Leibe. Efficient use of geometric constraints for sliding-window object detection in video. *Proceedings of the 8th international conference on Computer vision systems*, pages 11–20, 2011. URL http://link.springer.com/chapter/10.1007/978-3-642-23968-7{_}2$\delimiter"026E30F$nhttp://dl.acm.org/citation.cfm?id=2045399.2045402.

[191] X.-H. Sun and Y. Chen. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010. ISSN 07437315. doi: 10.1016/j.jpdc.2009.05.002. URL http://linkinghub.elsevier.com/retrieve/pii/S0743731509000884.

[192] Synopsys. DesignWare EV Family of Vision Processors. URL https://www.synopsys.com/dw/ipdir.php?ds=ev52-ev54.

[193] G. Tagliavini, G. Haugou, and L. Benini. Optimizing memory bandwidth in OpenVX graph execution on embedded many-core accelerators. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–8. IEEE, 2014.

[194] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. A framework for optimizing OpenVX applications performance on embedded manycore accelerators. *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems - SCOPES '15*, (June):125–128, 2015. doi: 10.1145/2764967.2776858. URL http://dl.acm.org/citation.cfm?doid=2764967.2776858.

[195] Tensilica. Tensilica IVP Product Line of DSP Cores for Image/Video Processing Datasheet, 2014.

[196] Tesla Motors. Model S Software Version 7.0, 2015. URL http://www.teslamotors.com/presskit/autopilot.

[197] Texas Instruments. TDA3x SoC Processors for Advanced Driver Assist Systems (ADAS) Technical Brief. 2014.

[198] Texas Instruments. New TDA3x SoC for ADAS solutions in entry- to mid-level automobiles. 2015.

[199] M. Tham. Overview of Mechanistic Modeling Techniques. Technical report, 2000. URL http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:OVERVIEW+OF+MECHANISTIC+MODELLING+TECHNIQUES{#}9.

[200] The Choco Team. Choco3: A Free and Open-Source Java Library for Constraint Programming., 2015. URL http://choco-solver.org/?q=Choco3.

[201] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 356–368, 2007. ISSN 10724451. doi: 10.1109/MICRO.2007.38.

[202] Tirias Research. Freescale's S32V ADAS Processor Family (White Paper), 2015.

[203] Toshiba. Automotive Solutions System Catalog, 2015.

[204] M. Tusch. Using Vision to Create Smarter Consumer Devices with Improved Privacy. In *Embedded Vision Summit*, 2015.

[205] M. Valera and S. A. Velastin. Intelligent distributed surveillance systems: a review. In *Vision, Image and Signal Processing, IEE Proceedings-*, volume 152, pages 192–204. IET, 2005. doi: 10.1049/ip-vis.

[206] N. Vasilache, M. Baskaran, B. Meister, and R. Lethin. Memory Reuse Optimizations in the R-Stream Compiler. *Gpgpu*, 2013. doi: 10.1145/2458523.2458528.

[207] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. In *Proceedings of the international conference on Multimedia*, pages 1469–1472. ACM, 2010. ISBN 9781605589336.

[208] N. Ventroux and R. David. Les architectures parallèles sur puce. *Journal techniques et sciences informatiques*, 29(345-378):15, 2010.

[209] Videantis. v-MP4000HDX Processor IP. URL http://www.videantis.com/products/processor-ip.

[210] P. Viola and M. Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004. URL http://link.springer.com/article/10.1023/B:VISI.0000013087.49260.fb.

[211] P. Viola, M. J. Jones, and D. Snow. Detecting pedestrians using patterns of motion and appearance. *Proceedings Ninth IEEE International Conference on Computer Vision*, 63(Iccv): 734–741 vol.2, 2003. doi: 10.1109/ICCV.2003.1238422. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1238422.

[212] S. White. Architecture for a digital programmable image processing element. *ICASSP '81. IEEE International Conference on Acoustics, Speech, and Signal Processing*, 6:658–661, 1981. ISSN 07367791. doi: 10.1109/ICASSP.1981.1171243.

[213] S. Williams, A. Waterman, and D. Patterson. Roofline : An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[214] W. Wolf. *High-performance embedded computing: Architectures, applications, and methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2006. ISBN 978-0123694850. URL http://store.elsevier.com/High-Performance-Embedded-Computing/Wayne-Wolf/isbn-9780123694850/.

[215] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor System-on-Chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[216] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361. Society for Industrial and Applied Mathematics, 1987.

[217] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.

[218] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs. *ACM SIGARCH Computer Architecture News*, 23(2):24–36, 1995. ISSN 01635964. doi: 10.1145/225830.223990. URL http://dl.acm.org/citation.cfm?id=225830.223990.

[219] N. Zhang. Working towards efficient parallel computing of integral images on multicore processors. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages 30–34. IEEE, Ieee, apr 2010. ISBN 978-1-4244-6347-3. doi: 10.1109/ICCET.2010.5485338. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5485338.

[220] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 47–58. IEEE Computer Society, 2009. ISBN 978-0-7695-3576-0. doi: 10.1109/CGO.2009.15.

[221] Z. Zhang. Microsoft kinect sensor and its effect. *MultiMedia, IEEE*, 19(2):4–10, 2012.

Methods and Tools for Rapid and Efficient Parallel Implementation of Computer Vision Algorithms on Embedded Multiprocessors

Abstract Embedded computer vision applications demand high system computational power and constitute one of the key drivers for application-specific multi- and many-core systems. A number of early system design choices can impact the system's parallel performance – among which the parallel granularity, the number of processors and the balance between computation and communication. Their impact in the final system performance is difficult to assess in early design stages and there is a lack for tools that support designers in this task. The contributions of this thesis consist in two methods and associated tools that facilitate the selection of embedded multiprocessor's architectural parameters and computer vision application parallelization strategies. The first consists of a DSE methodology that relies on Parana, a fast and accurate parallel performance estimation tool. Parana enables the evaluation of what-if parallelization scenarios and can determine their maximum achievable performance limits. The second contribution consists of a method for optimal $2D$ image tile sizing using constraint programming within the Tilana tool. The proposed method integrates non-linear DMA data transfer times and parallel scheduling overheads for increased accuracy.

Keywords: *Performance Modeling, Computer Vision, Multiprocessing, Parallel Programming, Constraint Optimization.*

Méthodes et outils pour implémentation rapide et efficace d'algorithmes de vision par ordinateur sur des multiprocesseurs embarqués

Résumé Les applications de vision par ordinateur embarquées demandent une forte capacité de calcul et poussent le développement des systèmes multi- et many-cores spécifiques à l'application. Les choix au départ de la conception du système peuvent impacter sa performance parallèle finale – parmi lesquelles la granularité de la parallélisation, le nombre de processeurs et l'équilibre entre calculs et l'acheminement des données. L'impact de ces choix est difficile à estimer dans les phases initiales de conception et il y a peu d'outils et méthodes pour aider les concepteurs dans cette tâche. Les contributions de cette thèse consistent en deux méthodes et les outils associés qui visent à faciliter la sélection des paramètres architecturaux d'un multiprocesseur embarqué et les stratégies de parallélisation des applications de vision embarquée. La première est une méthode d'exploration de l'espace de conception qui repose sur Parana, un outil fournissant une estimation rapide et précise de la performance parallèle. Parana permet l'évaluation de différents scénarios de parallélisation et peut déterminer la limite maximale de performance atteignable. La seconde contribution est une méthode pour l'optimisation du dimensionnement des tuiles d'images $2D$ utilisant la programmation par contraintes dans l'outil Tilana. La méthode proposée intègre pour plus de précision des facteurs non-linéaires comme les temps des transferts DMA et les surcoûts de l'ordonnancement parallèle.

Mots-Clés : *Modèles de Performance, Vision par Ordinateur, Multiprocesseurs, Programmation Parallèle, Optimisation par Contraintes.*

Thèse préparée au laboratoire TIMA (Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs), 46 Avenue Félix Viallet, 38031, Grenoble Cedex, France

ISBN: 978-2-11-129211-6