HAL
open science

Design Space Exploration for data-dominated image applications with non-affine array references

Rosilde Corvino

To cite this version:

Rosilde Corvino. Design Space Exploration for data-dominated image applications with non-affine array references. Computer Science [cs]. Université Joseph-Fourier - Grenoble I, 2009. English. NNT : . tel-00456577

HAL Id: tel-00456577

https://theses.hal.science/tel-00456577v1

Submitted on 24 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée par

Rosilde CORVINO

pour obtenir le titre de
DOCTEUR de l'UNIVERSITÉ GRENOBLE I - JOSEPH FOURIER

École Doctorale Électronique, Électrotechnique, Automatique & Traitement du Signal
Spécialité Micro et Nano Électronique

Design Space Exploration for data-dominated image applications with non-affine array references

Thèse dirigée par M. Jeanny HERAULT
et co-encadrée par M. Stéphane MANCINI et M. Roberto GUIZZETTI

Date de soutenance : 14 Octobre 2009

Composition du jury:

Fan YANG	**Rapporteur**
Jacques JAY	**Rapporteur**
Virginie FRESSE	**Examinateur**
Jeanny HERAULT	**Directeur de thèse**
Stephane MANCINI	**Co-encadrant**
Roberto GUIZZETTI	**Co-encadrant**

Contents

French translation **11**

Introduction **23**

I Context and Previous Works **25**

1 Context and Previous Works **27**
Introduction 27
1.1 Image processing applications and architectures 28
1.1.1 Image processing applications domain 29
1.1.1.1 Definitions 29
1.1.1.2 Parallel and Sequential Algorithms 30
1.1.1.3 Application with affine or non-affine array references . . . 30
1.1.1.4 Algorithm representations in a C-like code 31
1.1.1.5 The uniform and non-uniform memory access 32
1.1.2 Parallel architecture for the image processing 32
1.1.2.1 Pipeline 33
1.2 The High Level Synthesis in the design method history 35
1.2.1 The Y-chart and the synthesis levels 36
1.2.2 The abstraction level of a HLS input model 37
1.2.3 The HLS loop-based tools 38
1.2.4 The input C-code of the used HLS tool 40
1.2.5 The parallelism levels inferred by the HLS commercial tool: inter-operations, intra-loop and inter-loops parallelism 42
1.2.5.1 The inter-operations parallelism 43
1.2.5.2 The inter-iterations parallelism 43
1.2.5.3 The inter-loops parallelism 43
1.3 The Data Transfer and Storage Management in Systems Design 44
1.3.1 The memory hierarchy 44
1.3.1.1 Methods used to construct an optimized memory hierarchy 45
1.3.2 The target code optimizations 46
1.3.2.1 The dependence analysis 47
1.3.2.2 The loop transformations 48

1.3.2.3 The non-affine array references . . . 49
1.4 Comprehensive methodologies . . . 50
Conclusion . . . 51

II Research works **61**

2 MEXP: a Design Space Exploration tool **63**
2.1 Motivation . . . 64
2.2 Introduction to the Design Space Exploration tool . . . 65
2.2.1 Advantages of the methodology . . . 66
2.2.2 Overview of the target architecture . . . 67
2.2.3 The MEXP flow . . . 67
2.2.4 MEXP input and output . . . 70
2.2.4.1 The MEXP Input . . . 70
2.2.4.1.1 The user-defined input C-functions . . . 70
2.2.4.1.2 The input parameters . . . 71
2.2.4.2 The MEXP output . . . 72
2.3 Conclusion . . . 72

3 The Target architecture **73**
3.1 Introduction . . . 73
3.2 The meta-architecture overview . . . 75
3.2.1 The MM table used to pre-fetch input from the external memory . 76
3.2.2 The IDX table used to read data from the internal memory . . . 78
3.2.3 Example of a TPU time-line . . . 78
3.2.4 Possible conflicts on hardware resources . . . 79
3.3 The HLS model of the TPU . . . 80
3.3.1 Re-calls of the HLS principles . . . 80
3.3.2 The TPU C-model . . . 81
3.3.3 The user-specified parallelism level inferred by MEXP: parallelism between output tiles computations . . . 82
3.4 Automatic generation . . . 83
3.5 Conclusion . . . 84

4 Super-tiling **89**
4.1 Discussion about the problem of tiling non-affine loop nest . . . 89
4.2 Presentation of the Super-tiling flow and of the Exploration of a set of possible Super-tiling . . . 92
4.2.1 The target application . . . 93
4.2.2 Profiling . . . 94
4.2.3 Non-uniform loop nest and I/O dependence list . . . 94
4.2.3.1 The user-defined function . . . 95
4.2.4 Tiling . . . 96

4.2.4.1 The Tiling algorithm . . . 97
4.2.4.2 The tiles labeling . . . 97
4.2.5 Tiling Projection . . . 99
4.2.5.1 Example of the super-tiling application . . . 99
4.2.5.2 The projection algorithm . . . 100
4.3 Obtained results . . . 100
4.4 Conclusion . . . 102

5 Scheduling 105
5.1 Introduction to the scheduling . . . 105
5.2 Mathematical formulation of the TSP and the BTSP . . . 108
5.3 The possible costs and problems to optimize the TPU . . . 109
5.4 The genetic algorithms, the TSP and the BTSP . . . 112
5.4.1 The Algorithms . . . 113
5.4.1.1 The creation of the initial population of genotypes . . . 114
5.4.1.2 The evaluation of the population through the phenotype and the selection of the parents . . . 114
5.4.1.3 The crossover . . . 115
5.4.1.4 The mutation . . . 116
5.4.1.5 The criterion to stop the evolution of the population . . . 116
5.5 Results supporting the chosen algorithm . . . 116
5.5.1 Comparison between the GATSP, the Lin-Kernighan solver and the Concorde . . . 117
5.5.2 Comparison between the GABTSP and the GATSP . . . 119
5.6 Conclusion . . . 120

6 Computation and Memory Mapping 121
6.1 Introduction . . . 121
6.2 The Computation Mapping . . . 124
6.3 The needed amount of internal buffers . . . 125
6.4 The Memory Mapping . . . 126
6.4.1 The lifetime of the input tiles . . . 129
6.5 How to reduce the area overhead due to the usage of MM . . . 130
6.6 Conclusion . . . 130

7 Design Space Exploration: System storage requirement and performance estimation 135
7.1 The Design Space Exploration . . . 135
7.2 The selection criteria . . . 137
7.2.1 The amount of internal memory used . . . 137
7.2.2 The Temporal Performance of the TPU . . . 139
7.2.2.1 The time to initialize a TPU task . . . 143
7.2.2.2 The times neglected during the computation of TP . . . 143
7.3 Conclusion . . . 144

III Applications **145**

8 Tools used for the results analysis **147**
8.1 Metrics for the results analysis 148
8.2 Graphical tools for the results analysis 151
8.3 Conclusion 152

9 The Log Sampling **155**
9.1 The TPU synthesizable C-model for the LOG sampling 157
9.2 The MEXP analysis on the LOG sampling 157
9.2.1 SQCIF input image 158
9.2.2 VGA input image 163
9.3 HDTV input image 167
9.4 Conclusion 171

10 The Pyramidal Log Sampling **173**
10.1 The TPU synthesizable C-model of the Pyramidal LOG sampling 175
10.2 The MEXP analysis on the Pyramidal LOG sampling 175
10.2.1 SQCIF input image 176
10.2.2 VGA input image 182
10.2.3 HDTV input image 186
10.3 Conclusion 190

11 The Polar Transform **191**
11.1 The TPU synthesizable C-model of the Polar transform 191
11.2 The MEXP analysis on the Polar transform 192
11.2.1 128×128 input image 193
11.2.2 300×300 input image 198
11.2.3 600×600 input image 202
11.3 Conclusion 206

Conclusion **206**

Appendix **213**

A The bilinear interpolation **213**

B The hardware implementation and the Look-Up Table **215**
B.1 Recalls on the Look-Up tables 215
B.2 The LUT associated to the LOG sampling 216

C The space-variant low-pass **223**

D A mipmapping application: the Pyramidal LOG sampling 225
D.1 Re-calls on the MIP mapping 225
D.2 The Pyramidal LOG sampling 226
D.2.1 The function to access and construct the pyramid levels 227
D.2.2 The pyramidal LOG sampling steps 228

List of Figures 233

List of Tables 236

French translation

Introduction

Ce mémoire présente une méthode d'exploration d'espace de conception destiné à trouver architecture mémoire optimisée pour les systèmes de traitement d'image. Dans ce cadre, deux problèmes majeurs doivent être traités : la complexité des calculs effectués et l'accès à de grandes quantités de données. Ces deux problèmes sont reliés entre eux et à la qualité des transformations d'image appliquées. Le but principal de la conception de systèmes de traitements d'images et, en général, des systèmes multimédias, a toujours été de trouvé un compromis entre la surface occupée et la performance temporelle du système cible. Actuellement les circuits multimédias sont intégrés dans des systèmes portatifs, comme les téléphones mobiles ou les ordinateurs portables. Ainsi leur conception doit intégrer une nouvelle préoccupation : la consommation en puissance [1]. Cette préoccupation s'inscrit, avant tout, dans le problème de trouver une micro-architecture optimisée pour améliorer la gestion du transfert et du stockage des données. En plus, une telle micro-architecture peut éviter de "heurter", dans les prochaines années, "le mur mémoire" [2], qui représente le fossé technologique entre le processeur et la bande passante de la mémoire. Pour être résolu, le problème du mur de la mémoire requiert des solutions architecturales et technologiques. Dans ce travail, nous nous sommes intéressés aux solutions architecturales possibles. Ces quinze dernières années, se sont développées des techniques d'exploration des niveaux systèmes, qui comprennent certaines optimisations orientées mémoire, effectuées tôt dans le flot de conception. Ces techniques permettent d'explorer des réalisations possibles d'un algorithme en prenant en compte, d'une part, le comportement de l'algorithme vis-à-vis de la gestion des données, et d'autre part, les problèmes technologiques et architecturaux qui peuvent empêcher les spécifications du système en terme de surface, de puissance et de consommation en puissance. Des travaux précédents, portant sur les optimisations orientées mémoire, se focalisent sur des applications avec des références à tableaux affines, pour lesquels des solutions très brillantes ont été proposées pour améliorer la gestion des accès aux données et de leur transfert. Néanmoins, comme dit dans [3], "de nombreux et importants problèmes, comme les algorithmes à matrices creuses, le calcul de maillages non-structurés, certaines méthodes de tri et des transformations d'image spatio-variantes, contiennent des références à tableaux non-affines, qui requièrent des accès en mémoire non-affines". Certaines optimisations du stockage et du transfert des données ont été étudiées au cours de ce doctorat, adaptées à un domaine défini d'algorithmes de traitement d'image et incluses dans un outil d'exploration de l'espace de conception, appelé MEXP (pour l'anglais Memory EXPloration). Le but de MEXP est d'étendre les optimisations orientées mémoire étudiées aux algorithmes de traitement d'image avec des références à des tableaux non-affines, dans le but de traiter l'implémentation d'applications spatio-variantes utilisées dans le modèle de la rétine numérique développée au GIPSA-lab [4].

Le mémoire est divisé en trois parties. La première partie présente le contexte des travaux de ce doctorat et s'organise en trois points : Un premier point (1.1) décrivant les paradigmes d'algorithmes de traitement d'image et présentant les architectures matérielles utilisées pour les réaliser. Un deuxième point (1.2) décrivant deux méthodes pour op-

timiser un sous-système mémoire : les transformations de code et les améliorations matérielles. Un troisième point (1.3) qui présente la synthèse de niveau système et son automatisation. En particulier, est présenté l'outil de synthèse de haut niveau utilisé au cours de ce doctorat. La deuxième partie présente l'outil d'exploration de l'espace de conception développé. Il contient sept chapitres : Le chapitre 2, qui introduit l'outil développé et son flot. Le chapitre 3, qui décrit le matériel cible. Les chapitres 4, 5, 6 et 7 qui décrivent chaque étape du flot de l'outil. La troisième partie présente trois analyses approfondies, réalisées sur trois applications cibles, et montre l'efficacité de l'outil développé.

Contexte

Ces travaux de thèse qui portent sur le développement d'un outil d'exploration des architectures mémoire possible pour des systèmes de traitement d'image. L'analyse de cet outil a été appliquée á des blocs fondamentaux de la rétine numérique développée au GIPSA-lab.
Le contexte de ce travail est celui de la synthèse de haut niveau, appliquée aux systèmes de traitement d'image. Plus en particulier, nous nous sommes attaqués au problème de l'accès et du transfert des données pour les algorithmes de traitement d'image manipulant un grand nombre de données. C'est pourquoi, dans la partie de contexte, nous allons vous présenter certaines caractèristiques des algorithmes de traitement d'image qui sont sensible de nous intéresser dans notre cas. Nous allons en suite présenter la synthèse de haut niveau et le problème de l'accès et du transfert des données dans le cas des algorithmes de traitement d'image.
Un algorithme de traitement d'image est un algorithme itératif, qui, dans un langage de haut niveau (type C), peut être décrit par une suite de boucles imbriquées qui itèrent la même opération sur un flot de données : les pixels. Une architecture correspondant à ce type de fonctionnement est le réseau systolique. Un réseau systolique est constitué d'une maille de processeur communicant par des liaisons point-à-point ou par des mémoires internes. Chaque processeur prend en entrée un flot de données, les traite et les passe à ses voisins pour la suite du traitement. La transformation d'un modèle algorithmique en un modèle structurale est applelé Synthèse de Haut Niveau (SHN). La SHN, en effet, permet de passer d'un modèle séquentiel algorithmique non timé vers un modèle structural parallèle et timé, appelé aussi modèle RTL pour Register Transfer Level. Les avantages de la SHN sont multiples nous rappelons l'augmentation de la productivité du designer, la réduction du nombres d'erreurs dans le phases qui permettent d'obtenir un modèle RTL et le fait que les temps de simulation d'un modèle de haut niveau sont plus couts par rapport aux temps de simulation d'un modèle structurale, ce qui permet d'effectuer une exploration de l'espace des architectures plus poussée.
Dans notre cas, nous nous sommes intéressés aux outils de SHN qui prennent en entrée des programmes en C décrits par des "boucles for" imbriquées et produisent en sortie des architectures du type réseaux systolique. Ces outils sont, donc, adaptés au cas des algorithmes de traitement d'image. Dans ces outils, la SHN s'effectue par deux étapes

principales : une synthèse comportementale et une synthèse structurale. La synthèse comportementale permet de passer d'un modèle séquentiel vers un modèle parallèle intermédiaire où à chaque boucle imbriquée corresponde un processeur. Pendant cette étape : l'outil effectue de l'analyse et de l'optimisation du code d'entrée, il trouve des ordonnancements possibles des itérations qui optimisent le parallélisme et respectent les contraints utilisateurs et, enfin, génère le contrôleur du séquencement des itérations sur le processeur associé à une boucle. Pendant la synthèse structurale, l'outil génère le chemin des données associé au coeur de chaque boucle imbriquée. Pour chaque chemin des données, on explore différentes solutions d'allocations de ressources et placement des opérations pour optimiser les performances du système généré et respecter les contraints utilisateurs.
Dans ce type d'outil la gestion de l'accès et du transfert des données doit être fournie par l'utilisateur. Le problème de l'accès et du transfert des données concerne principalement les systèmes gourmands en calcul et manipulant un grand nombre de données. Pour trouver une architecture optimisée l'utilisateur doit faire face à un problème double : d'un coté le stockage des données manipulées en mémoire interne et d'un autre coté le transfert des données depuis la mémoire externe. Le stockage en interne cause une augmentation de la surface occupée par le circuit et le transfert de la mémoire externe cause une dégradation des performances temporelles du circuit à cause de la latence pour l'accès à la mémoire externe. Comme montré par la figure sur le transparent, cette latence a plusieurs contributions : par exemple la requête d'une donné de la part d'une unité fonctionnelle peut être mise en attente pendant que le contrôleur mémoire traite des requêtes précédentes.
Considerons un cas concret du problème de l'accès et du transfert des données en considérant l'exemple de la rétine numérique. La rétine est constituée d'un bloc de prétraitement (l'échantillonnage logarithmique), d'un bloc de post- traitement (la projection polaire) et d'un coeur de la rétine, qui est à son tour la combinaison de plusieurs blocs fondamentaux : comme le filtrage spatio-temporel, le filtrage pass-haut temporel et l'adaptation à la luminance ambiante. Tous ces blocs ont des dépendances spatiales et temporelles, qui imposent d'utiliser des images intermédiaires, plus en particulier pour calculer une image de sortie en partant d'une image d'entrée, il faut stocker 17 image intermédiaire, il faut effectuer 300 opérations par pixel et, en plus, nous souhaitons avoir en sortie un taux d'au moins 15 images produites par seconde.
Sous ces conditions, nous avons estimé la quantité de mémoire interne nécessaire dans le cas hypothétique où toutes les images intermédiaires soient stockées en interne. Nous avons aussi calculé la complexité correspondante de l'algorithme et cela pour trois taille d'image d'entrée. Nous voyons que pour des images d'entrée de grande taille, la mémoire interne requise est d'environ 300mm2, ce qui est inacceptable si nous considérons que toute une application industrielle réelle tient sur 20mm2. D'un autre coté, il n'est pas envisageable de transférer les images intermédiaires directement de la mémoire externe, car cela dégraderait les performances temporelles du système ce qui n'est pas souhaitable vu la complexité déjà élevée de l'algorithme. Il faut donc trouver un compromis entre le stockage des données en interne et le transfert des données depuis la mémoire externe. Ce

compromis se trouve en proposant des solutions sur un plan architectural et des solutions sur un plan algorithmique.
Une solution sur un plan architectural consiste à créer une hiérarchie mémoire. C.à d. que les données manipulées par l'algorithme sont stockées en mémoire externe et une partie des données utiles dans l'immédiat sont recopiées dans une mémoire interne en créant ce qui s'appelle la localité des données. Une fois que les données contenues en mémoire interne ne sont plus utiles nous pouvons les substituer avec d'autres données en réutilisant les mêmes ressources mémoire. Finalement, si la quantité de mémoire allouée permet d'éviter les conflits en accès à la mémoire, il est possible de pre-fetcher (ou bien copier à priori) les données nécessaires pour effectuer la ta?che suivante pendant que nous effectuons la ta?che courante. Pour trouver une hiérarchie mémoire optimisée il est nécessaire de procéder à une exploration des solutions possibles en estimant et allouant les quantités de mémoires internes nécessaires et en assignat à chaque mémoire interne allouée le stockage d'un blocs de données. Sur un plan algorithmique, la solution au problème de l'accès et du transfert des données consiste à trouver un partitionnement des données accédées par l'algorithme et des opérations effectuées. Les solutions présentes dans la littérature consistent à partitionner l'espace des opérations effectuées par l'algorithme et trouver les footptints correspondant à chaque répartition. Où le footprint est la trace des données accédées par un groupe d'opérations. Il existe deux méthodes pour trouver les footprints : par analyse statique ou par analyse dynamique. Par analyse dynamique, on construit une trace des accès faits à la mémoire en exécutant l'algorithme. Dans ce cas, les footprints peuvent avoir des tailles variables et il n'y a pas de conditions sur les lois d'accès aux données. Par analyse statique, on détermine les dépendances des données par des méthodes de programmation linéaires. Dans ce cas, les lois d'accès aux données doivent être affines et les footprints ont une taille constante (car il sont affine entre eux).
Considérons le partitionnement dans le cas des footprints de taille variable. Pour faire cela, considérons l'exemple du pre-traitement de la rétine : l'échantillonnage logarithmique. Cet algorithme consiste à reconstruire une image de sortie en allant échantillonner l'image d'entrée selon une loi pseudo-logarithmique, donc les accès aux données ne sont pas affines. Si nous appliquons une répartition régulière à l'ensemble des opérations nécessaires pour calculer la sortie, nous aurons une répartition non régulière de l'ensemble des données en entrée. Une réalisation matérielle d'un algorithme ainsi partitionné doit inclure un contrôle qui peut être coûteux sur le transfert des blocs de données d'entrée.
Considérons le partitionnement dans le cas des footprints de taille constate. Ceci peut s'appliquer dans le cadre du modèle polyédrique. Le modèle polyédrique donne une représentation matricielle d'une boucle for imbriquée. (Par exemple, les indexes de boucle sont représentés en tant que vecteur et les dépendances entre données (calculé par analyse statique) sont représentées par une matrice). Appliquer une transformation de boucle dans le cadre du modèle polyédrique consiste à effectuer des produits matriciels entre les matrices représentant la boucle et la matrice de la transformation. La transformation permettant d'obtenir un partitionnement des itérations de la boucle (et donc des opérations effectuées) est le ´ loop tiling [a]. A? cette répartition des itérations correspond une répartition des données en blocs (ou tuiles) qui sont affines entre eux.

En conclusion, il existe deux méthode de répartitions des données et des opérations d'un algorithme. Une méthode qui permet de traiter des accès aux données non-affines et qui peut demander un contrôle complexe sur le transfert des données et une méthode qui ne permet de traiter que des lois d'accès affines mais qui requière un contrôle simple sur le transfert des données. Un de nos objectifs a été de définir une méthode qui permet de traiter des accès aux données non-affines et qui minimise le contrôle sur le transfert des données.
Plus en général l'objectif de ce doctorat a été de développer un outil d'exploration des architectures mémoires possibles pour des systèmes de traitement d'image. Cet outil devait être capable de gérer des accès aux données non-affines et de générer en sortie en programme en C qui peut être sysnhtèsé par un outil commercial de SHN. Plus en particulier, l'outil développé prend en entrée un programme en C, analyse les accès aux données, optimise ces accès et génère en sortie un programme en C optimisé pour la SHN. La génération est faite en customisant une architecture générique.

Chapitre 2

Dans ce chapitre nous avons présenté les principes de base de MEXP. MEXP permet d'explorer un ensemble de couples entrée/sortie candidats de tiling pour un algorithme manipulant un grand nombre de données et ayant des références aux tableaux non affines. L'exploration est réalisée en fonction de de deux critère d'optimisation : la quantité de mémoire interne utilisée et la performance temporelle du système généré.
En utilisant un modèle générique du code à généré et certaines fonctions spécifiées par l'utilisateur, MEXP est capable de générer un code-C optimisé, qui peut être synthétisé au moyen d'un outil commercial de synthèse de haut niveau.
Les optimisations appliquées au code sont déduites de la prise en compte le comportement temporel de l'algorithme et les caractéristiques estimées du matériel cible.

Chapitre 3

Dans ce chapitre nous avons présenté l'architecture cible générique customisée par MEXP pour générer le code-C de sortie. L'architecture est appelée unité de traitement des tuiles (Tile Processus Unit ? TPU) et réalise deux macro-t,ches parallèles : le pré-chargement et le calcul. Le calcul est réalisé au travers d'un pipeline d'opérations et le pré-chargement consiste à copier les tuiles d'entrée à partir d'une mémoire externe dans des tampons internes.
Pour une solution donnée, la correspondance entre les tampons des tuiles d'entrées et les tampons internes disponibles est pré-calculée par MEXP et utilisée par le TPU customisé pour réaliser le pré-chargement.
Le TPU a plusieurs niveaux de parallélisme qui peuvent être distingués en : un parallélisme assuré par l'outil commercial de synthèse de haut niveau sous les contraintes pré-calculées de MEXP et un parallélisme inter-tuiles de sortie déduits par MEXP.
Le parallélisme inter-tuile requiert l'instanciation de pipelines parallèles supplémentaires

réalisant le calcul. Le pré-chargement est commun à tous les différents pipelines, en vue de respecter la bande passante, et ceci peut limiter les possibilités de parallélisations supplémentaires.

Chapitre 4

Ce chapitre présentait le partitionnement superposé (SP pour l'anglais, Super-Tiling) qui est une méthode permettant d'appliquer le partitionnement sur des boucles imbriquées pas nécessairement affines. L'idée principale du SP est de partitionner séparément les ensembles de données d'entrée et de sortie en tuiles, avec un partitionnement régulier. Ensuite le partitionnement de sortie est projeté sur le partitionnement d'entrée selon la loi de référence aux tableaux. L'intersection de la projection avec le partitionnement d'entrée donne les dépendances entre les tuiles d'entrée et de sortie. Le SP a trois étapes : un profilage dynamique des références aux tableaux de l'algorithme, un partitionnement qui s'applique séparément aux espaces de données d'entrée et de sortie et une projection qui relie les tuiles d'entrée et de sortie entre elles. Le partitionnement des données de sortie correspond au partitionnement de l'espace de calcul de sortie. Ainsi, le partitionnement permet de paralléliser les calculs des différentes tuiles de sortie. Il rend également possible la parallélisation entre pré-chargement des tuiles d'entrée et le calcul des tuiles de sortie. Les résultats montrent que la non-affinité des références aux tableaux influence le rapport largeur-hauteur et ainsi, le montant de la mémoire interne du matériel réalisé.

Chapitre 5

Dans ce chapitre nous avons présenté l'ordonnancement (Scheduling). L'ordonnancement vise à réorganiser les calculs des tuiles de sortie dans le but d'optimiser le comportement du TPU. Il est possible de le faire vis-à-vis de trois coûts : la quantité de mémoire interne utilisée, le nombre d'accès à la mémoire externe et le nombre de tuiles d'entrée qui changent entre le calcul de deux tuiles de sortie successives. La minimisation de chacun de ces trois coûts produit une amélioration soit sur la puissance consommée par le système, soit sur la quantité de mémoire interne utilisée et le temps de pré-chargement. Selon le critère d'optimisation choisi, nous devrions résoudre soit le problème du voyageur de commerce (PVC) classique, soit le PVC qui évite les goulots d'étranglement. Dans notre travail, nous avons utilisé un algorithme génétique pour résoudre ces deux problèmes. Nous avons comparé le PVC résolu en utilisant les algorithmes génétiques par rapport à un résoluteur du type Lin-Kernighan et un autre résoluteur de référence : le concorde. Les expériences montrent que notre algorithme génétique trouve des solutions que celles d'une Lin-Kernigan dans 90 % des cas. L'écart entre les solution trouvés par le résoluteur Lin-Kerighan et celle de l'algorithme génétique sont en moyenne de 35 %. La comparaison de l'algorithme génétique par rapport au Concorde montre que le Concorde trouve de meilleures solutions dans les 86 % des cas. L'écart moyen entre les solutions de l'algorithme génétique et le Concorde est de 16 %. D'un autre côté, l'algorithme

génétique est jusqu'à 76 fois plus rapides que le concorde. C'est pourquoi, par la suite nous utilisons l'algorithme génétique développé.

Chapitre 6

Dans ce chapitre nous avons décrit le mappage des calculs et des données. Ces mappages sont calculés pour chacune des solutions analysées par MEXP. Le mappage des calculs divise les tuiles de sorti en Np groupes et alloue le calcul d'un groupe à un seul pipeline des Np pipelines instanciés. Les tuiles d'un groupe sont calculées séquentiellement par le même pipeline, alors que chacun des groupes sont calculés en parallèle par différents pipelines. Le pré-chargement des données pour tous les pipelines est effectué séquentiellement, et les différents pipelines sont synchronisés à la fin du calcul en parallèle des Np pixels de sorties. Le mappage des données vise à placer les tuiles d'entrée dans les mémoires tampon internes disponibles. La première étape du mappage de données est de compter le nombre de mémoires tampon internes nécessaires puis, pour chaque tuile de sortie, de calculer la position des tuiles d'entrée dans les tampons internes. Le mappage des données assure que les tuiles d'entrée partagées entre deux tuiles de sortie calculées successivement par le même pipeline sont copiées une seule fois depuis la mémoire externe. Il assure aussi que les blocs de pré-chargement et de chargement du TPU peuvent accéder sans conflit aux mémoires tampon internes.

Chapitre 7

Ce chapitre présentait la méthode utilisée pour réaliser l'exploration de l'espace des architectures mémoire possibles (EEAMP). Il décrit la structure des données utilisées pour classifier les solutions et la méthode employée pour réduire l'espace des solutions possibles par le biais de contraintes utilisateur. L'EEAMP est réalisée à travers un ´†filtre de validation†[a]qui réduit l'espace des solutions possibles et un ´†filtre de qualité†[a] qui qualifie les solutions de dominantes ou d'équivalentes. L'EEAMP est effectuée par rapport à deux critères de sélection : les performances temporelles du système et la quantité de mémoire interne utilisée. Ces critères sont estimés et leurs formules sont données.

Chapitre 8

Dans ce chapitre, on a présenté 5 critères et deux types de graphique. Ces outils seront utilisés dans les chapitres suivants. pour décrire les expériences réalisées. Les critères présentaient sont :

- Trois critères pour décrire les améliorations des optimisations de MEXP sur les performances temporelles : l'accélération due aux optimisations de MEXP (MEXP SU - pour MEXP Speed Up), l'accélération due au parallélisme (Parallel SU) et l'efficacité de parallélisme.

- Deux critères pour décrire le surcoût surfacique qui peut être dû soit aux optimisations de MEXP (MEXP AO - pour MEXP Area Overhead) ou au parallélisme (Parallel AO).

Les outils graphiques sont :

- les nuages de points de l'espace d'exploration de MEXP, qui classe les les solutions, dans l'espace exploré, en donnant leur performance temporelle en fonction de leur quantité de mémoire interne utilisée. Nous donnerons cette représentation pour trois fois valeurs de latence d'accès à la mémoire externe (30, 60 et 100 cycles) et trois valeurs de niveau de parallélisme (Np = 1, 2, 4);

- Une représentation de la performance temporelle (TP pour Temporal Performance) d'une solution en fonction de la latence d'accès à la mémoire externe. Cette représentation graphique fait apparai?tre trois régimes de l'évolution de la TP en fonction de la latence, selon la valeur prise par le rapport du temps de préchargement sur le temps de calcul. Quand ce rapport est inférieur à 1, la TP est indépendante de la latence et reste donc constante. Quand le rapport vaut environ 1, la performance temporelle augmente avec la latence de façon logarithmique. Quand le rapport est supérieur à 1, la TP augmente linéairement avec la latence d'accès à la mémoire externe.

Chapitre 9

Dans ce chapitre, nous avons présenté les résultats de trois explorations sur l'échantillonnage logarithmique. Nous avons analysé trois tailles d'image d'entrée (SQCIF, VGA et HDTV). Pour chaque taille d'image d'entrée, nous avons exploré des centaines de solutions. ¿ partir des résultats obtenus, on peut déduire que les optimisations de MEXP dépendent des tailles de tuiles d'entrée et de sortie et des tailles des espaces de données en entrée et en sortie. En particulier, elles sont plus efficaces quand on considère une grande image d'entrée (VGA et HDTV) plutôt qu'une plus petite image (SQCIF). L'accélération de la performance temporelle due aux optimisations de MEXP peut aller jusqu'à un facteur de 93.3 pour une augmentation de surface d'un facteur de 27.21 (résultats observés avec le format d'image d'entrée HDTV avec un niveau de parallélisme Np = 4). L'accélération maximale observée, sans considérer de parallélisme atteint quant à elle, un facteur de 23.7 pour une augmentation de surface d'un facteur de 9.11. (observation faite sur une image d'entrée HDTV). Le parallélisme n'est pas efficace dans le cas d'une image d'entrée SQCIF et est très efficace dans le cas de l'HDTV (efficacité supérieure à 0.9).

Chapitre 10

Dans ce chapitre, nous avons présenté les explorations réalisées sur l'échantillonnage logarithmique pyramidal. Nous avons exploré des centaines de solutions pour trois tailles

d'images d'entrée (SQCIF, VGA et HDTV). ¿ partir des résultats obtenus nous pouvons voir que les optimisations de MEXP sont plus efficaces pour l'échantillonnage logarithmique pyramidal que pour l'échantillonnage logarithmique simple. En effet, l'échantillonnage logarithmique pyramidal a un plus grand nombre d'accès à la mémoire, cependant l'accroissement de surface est plus faible pour l'échantillonnage logarithmique pyramidal du fait que les tailles des tuiles d'entrée et de sortie analysées sont plus adaptées à cette application. L'accélération des performances temporelles dues aux optimisations de MEXP peuvent atteindre un facteur de 134.12 pour une augmentation de surface correspondante de 21.9 (résuktats observés pour l'image d'entrée de HDTV avec un niveau de parallélisme Np = 4). L'accélération maximale observée, sans considérer de parallélisme (Np = 1), atteint un facteur de 34, pour un accroissement de surface de 6.9 (pour une image d'entrée HDTV). Le parallélisme n'est pas efficace dans le cas où une image d'entrée SQCIF et est très efficace pour les images d'entrée VGA et HDTV (E>0.9).

Chapitre 11

Dans ce chapitre, nous avons présenté les explorations réalisées pour la transformation polaire. Nous avons exploré des centaines de solutions pour trois tailles d'images d'entrée (128x128, 300x300 et 600x600). Des résultats, on peut voir que l'accélération des performances temporelles due aux optimisations MEXP peut monter jusqu'à un facteur de 40.9 pour une une augmentation de surface correspondante d'un facteur de 30.5 (résultats observés pour une image d'entrée de 300x300 avec un niveau de parallélisme de Np = 4). L'accélération maximale observée, sans considérer de parallélisme, atteint un facteur de 22 pour une augmentation de surface d'un facteur de 9.4 (pour une image d'entrée de 300x300). Le parallélisme n'est pas efficace dans le cas d'une image d'entrée de 128x128 et est très efficace dans les cas d'images d'entrée de 300x300 et de 600x600 (E>0.9). L'écart moyenne entre les estimations MEXP et les performances mesurées après synthèse sur l'outil de SHN est de l'ordre de 10%.

Conclusion

Le but de cette thèse de doctorat était d'étudier une méthodologie qui améliore le transfert et la gestion des données pour des applications à références de tableaux non-affines. Les applications cibles sont des algorithmes de traitement d'image qui ne sont pas récursifs et qui ont des dépendances statiques. Ces applications sont bien décrites par du code-C basées sur des boucles et peuvent subir une synthèse de haut niveau (SHN) qui déduit un modèle RTL d'un code-C en entrée. Le code d'entrée de la SHN peut être optimisé, par des transformations de boucles, en fonction de la gestion et du stockage des données. En fait, ces transformations augmentent la localité des données et permettent, au travers du partitionnement de données, d'accomplir le parallélisme de calcul et le préchargement des données. En particulier, le partitionnement des calculs et des données est atteint à travers une transformation appelé tiling ("tuilage"). Gr,ce aux transformations de boucle, certains outils de SDN existant sont capables de générer du matériel de haute

qualité. Dans la première partie de cette dissertation, nous avons présenté le contexte du problème et les travaux précédents qui y sont reliés. En particulier, nous avons souligné que les méthodes existantes, optimisant la gestion et le transfert de données, ne sont pas adaptées aux références de tableaux non affines. Dans le but de fournir une solution au problème de l'analyse et de l'optimisation des applications ayant des références de tableaux non-affines, nous avons développé un outil appelé MEXP (pour Memory EXPloration). MEXP est un outil d'exploration de l'espace de conception, destiné à trouver un tuilage d'entrée/sortie (E/S) adapté pour un traitement d'image ayant des références à des tableaux non-affines. Le but est de partitionner les données et les calculs de l'application en vue d'équilibrer le pré-chargement des données d'entrée et les calculs des sorties. Le matériel correspondant est généré à partir d'un modèle customisable appelé Unité de Traitement des Tuiles (TPU, pour l'anglais Tile Process Unit). Dans les chapitres 2 et 3, nous avons décrit le flot de l'outil et le matériel cible customisable. MEXP a l'avantage d'adapter le choix d'un couple de tuiles d'E/S à la non-affinité des référence de tableaux de l'application. Un couple approprié de tuiles d'E/S peut masquer complètement le temps de pré-chargement et assurer l'invariance des performances temporelles du TPU vis-à-vis de la latence de la mémoire externe. Dans le chapitre 4, nous avons décrit la méthode utilisée pour construire un ensemble de couples possibles de tuilages d'E/S . L'analyse prend en compte deux fonctions spécifiées par l'utilisateur, qui décrivent l'application cible, et plusieurs paramètres qui adaptent l'exploration de l'espace de conception opérée par MEXP. MEXP est aussi capable de ré-ordonnancer temporellement les calculs des tuiles de sortie en vue de réduire le transfert de données à partir de la mémoire externe. Cela permet une réduction de la consommation en puissance et une amélioration des performances temporelles du TPU. Le chpitre 5 décrit la méthode utilisée pour ré-ordonnancer les calculs des tuiles de sortie. Le TPU accède aux données gr,ce à une table donnant le mappage entre les tuiles d'entrée et les mémoires tampons internes. La tables est générée par MEXP pour chaque solution choisie. MEXP peut explorer et optimiser des centaines de solutions. Il classifie ces solutions en fonction de deux critères : la performance temporelle estimée et la mémoire interne utilisée, dans le matériel correspondant. Les chapitres 6 et 7 donnent les méthodes pour calculer le mappage mémoire et exécuter l'exploration de l'espace de conception. MEXP évalue aussi la possibilité de paralléliser les calculs de plusieurs tuiles de sortie en instanciant un matériel parallèle au sein du TPU. Le mappage de calcul correspondant est décrit dans le chapitre ??. Dans les chapitres 9, 10 et 11, nous décrivons les expériences réalisées pour trois applications cible : l'échantillonnage logarithmique, l'échantillonnage logarithmique pyramidal et la transformation polaire. Les expériences montrent que les optimisations de MEXP dépendent de la taille des tuiles d'E/S et la taille de l'espace des données d'E/S. Les résultats sont évalués en fonction de l'accélération que les optimisations de MEXP permettent sur les performances temporelles et sur le surcoût surfacique. La plus forte accélération observée atteint un facteur 134.12 pour un surcoût surfacique de 21.9 (résultats observés pour l'échantillonnage logarithmique pyramidal calculant 4 tuiles de sortie en parallèle et et pour une latence de mémoire externe de 100 cycles). Les expériences réalisées montrent également que, gr,ce à un choix judicieux de couple de tuiles

d'E/S, un équilibre idéal entre le pré-chargement et le calcul peut être atteint.

Une première limitation de MEXP consiste en ce que le générateur du code-C synthétisable est seulement exploitable pour l'outil de SHN étudié, alors que l'analyse opérée par MEXP peut être étendue à d'autres outils HLS ou pour mapper des traitements d'image sur d'autres types d'architectures et des processeurs programmables. Dans ce cas, le patron générique du code-C synthétisable devrait être adapté à l'architecture cible. Une autre limitation de la version actuelle de MEXP réside dans le fait que l'outil n'est appliqué qu'à un traitement d'image d'une seule étape. L'analyse n'est pas applicable à une transformation multi-étape, c'est-à-dire une transformation réalisée au travers d'une chaîne de plusieurs étapes inter-dépendantes. L'adaptation requerrait plusieurs changements dans l'architecture cible et du flot. Un solution possible serait d'instancier un TPU par étape dans l'application. Un contrôleur synchroniserait le démarrage et la communication des TPU. L'architecture TPU pourrait rester la même, mis à part que l'ordonnancement des tuiles de sortie du premier TPU serait relié à l'ordonnancement des tuiles de sortie du second TPU.

Introduction

This dissertation presents a Design Space Exploration (DSE) method aimed to find an optimized memory architecture for image processing systems. In this framework, two major problems need to be handled: the complexity of the performed computations and the access to a large amount of data. These two problems are related to each other and to the quality of the applied image transformations. The main goal of the design of image processing systems and, in general, of multimedia systems, has always been to find a trade-off between the **occupied area** and the **temporal performance** of the target system.

Nowadays the multimedia circuits are integrated in portable systems. Thus their design has to integrate a new concern: the **power consumption** [1].

The three previously enumerated concerns (area occupancy, temporal performance and power consumption) are related to the problem of finding an optimized micro-architecture to improve the data storage and transfer management (DSTM).

The DSTM is a part of the "memory wall" problem [2], which represents the technological gap between the processor speed and the memory bandwidth. To be solved, the memory wall problem needs technological and architectural solutions. In this work, we are interested in the possible architectural solutions.

The last fifteen years have seen the rising of the early system level exploration techniques which include some **memory aware optimizations** in the early phases of the design flow. These techniques allow to explore possible realizations of an algorithm by taking into account, on one hand, the algorithm behavior with respect to the data management and, on the other hand, the technological and architectural problems that may prevent to achieve the system specifications on area, performance and power consumption.

Previous works on memory aware optimizations focus on applications with affine array references[a] for which very brilliant solutions have been proposed to improve the data access and the storage management. Nonetheless, as said in [3]: "many important problems, such as sparse matrix algorithms, unstructured mesh calculations, some sorting methods and space-variant image transformations, contain non-affine array references, which require non-uniform memory accesses".

Some of the data transfer and storage optimizations have been studied during this Ph.D., adapted to a defined domain of image processing algorithms and included in a

[a] an affine formula is a linear formula with a translation, i.e. $y = a.x + b$

Design Space Exploration (DSE) tool called MEXP, from Memory EXPloration.

The goal of MEXP is to extend the studied memory aware optimizations to the image processing algorithms with non-affine array references, in order to handle the implementation of space variant applications used in the GIPSA-lab retina model [4].

This dissertation is divide into three parts:

The first part contains a single chapter and presents the context of the PhD works. This chapter contains three sub-sections:

- Sub-section 1.1, which describes the image processing algorithm paradigms and presents the hardware architectures used to realize them.
- Sub-section 1.2, which presents the System Level Synthesis and its automation. In particular, it presents the High Level Synthesis tool used during this PhD work.
- Sub-section 1.3, which describes two methods for optimizing a memory sub-system: the code transformations and the hardware improvements.

The second part presents the developed Design Space exploration tool. It contains seven chapters:

- Chapter 2, which gives an introduction to the developed tool and its flow.
- Chapter 3, which describes the target hardware.
- Chapter 4, chapter 5, chapter 6 and chapter 7, which describe the different steps of the tool flow.

The third part presents three extensive analyses, which are performed on three target applications and show the efficacy of the developed tool.

Part I

Context and Previous Works

Chapter 1

Context and Previous Works

Introduction

The context of this work is the High Level Synthesis (HLS) applied to the case of the image processing systems. In particular, we propose a possible solution to the problem of data storage and transfer management (DSTM). Thus in the first part of this dissertation we will present

- few characteristics of the image processing systems, which are interesting in our case;
- the HLS applied to the image processing systems;
- the existing solutions with respect to the problem of data storage and transfer management.

An image processing algorithm is an iterative algorithm which can be described in a high level language (as the "C") as a series of nested loops iterating the same operation on a flow of data: the pixels.

An architecture able to implement this behavior is the **systolic array**, which contains a mesh of processors communicating to each other through a direct link or through an internal memory.

The passage from an algorithmic model to an architectural model of a system is ensured by the HLS, which allows to extract a parallel, structural and timed model from a sequential, algorithmic and non-timed model [5].

The HLS allows to improve the designers productivity and dramatically reduces the errors during the phases leading to the structural model. Furthermore, the rapidity of the simulation of the high level model, permits to enlarge the design space and thus enforce the exploration results [6].

In our work, we are particularly interested to the HLS tools taking a loop-based C-program as input and producing a systolic array as output. These tools are adapted to the image processing case. In these kinds of tools, the HLS is applied in two steps: a behavioral synthesis and a structural synthesis. During the behavioral synthesis, the tool

applies some input code analysis and optimizations in order to enhance the data and task parallelism. Then, it finds a first parallel model in which a processor corresponds to each loop nest. Finally, it generates a sequence controller to execute the nested loop iterations on each instantiated processor. During the structural synthesis, the tool generates the data path corresponding to each loop core.

The main limitation in this kind of tools, with respect to the data transfer and storage management, is that the corresponding micro-architecture has to be designed by the user.

The problem of the data transfer and storage management is double: the user has to evaluate the pros and cons of storing the data into internal memories and of transferring data from an external memory. The storage into internal memories increases the area occupied by the target circuit and the transfer from external memory degrades the temporal performance of the system.

In the literature (cf. [7] and all the papers cited in it), we can find two kinds of solutions to this problem: one concerning the architectural structure of generated system and the other achieved by transforming the target algorithm. The architectural solution consists in creating a memory hierarchy and the algorithmic solution consists in partitioning the data access and the operations performed by the algorithm.

In this chapter we will present:

- The target image processing and architectures
- The High Level Synthesis
- The data transfer and storage management in system design.

1.1 Image processing applications and architectures

Image processing is a wide domain of applications which imply the manipulation of a large amount of data and the realization of a high number of complex computations. The basic data structure of an image processing application is a 2-dimensional table storing the spatial distribution of the luminance values (gray levels or color). The volume of manipulated data and performed computations, depends on the image size and is exacerbated when the process involves several images of a sequence or is a real-time application.

In this work, we will classify the image processing algorithms according to two characteristics:

- the existence of data dependences that may impede or allow to execute parts of the algorithm in parallel and
- the kind of the array references that the algorithms contain.

This classification helps to point out how the memory-aware optimizations apply to the image processing algorithms and how some of these optimizations are limited to a specific family of algorithms.

The hardware architectures used in image processing can be either general purpose or application specific architectures. The general purpose architectures are flexible, may realize a high number of complex operations and are usually used to validate an algorithm. The application specific architectures are used to obtain embeddable accelerators with a low surface, a low power consumption and a high computational performance.

In this work, we are interested to the application specific architectures.

An image processing can implement several level of parallelism: a *fine-grain parallelism*, which includes the operations pipelining[a] and the operations parallelism and a *coarse-grain parallelism*, which can be a data or task parallelism.

The data parallelism is inferred when all the data (in the manipulated set) undergo to identical (and inter-independent) computations and, thus they can be processed in parallel. The task parallelism consists in applying different tasks in parallel on different data sets.

In this work we are interested to the both kinds of coarse-grain parallelism: the one concerning data and the one concerning tasks.

The next paragraph presents:

- a general model of an image processing and a possible corresponding classification
- the parallel architecture paradigms used in image processing, by focusing only on the coarse grain parallelism.

1.1.1 Image processing applications domain

In this paragraph we will give the used nomenclature and the classification of the image processing applications according to the array references that they contain.

1.1.1.1 Definitions

We define the following terms:

- D^n is a n-dimensional discrete domain
- I is a set (called image) of elements (called pixels), which is defined on D^n
- $I_t(p)$ is a pixel of coordinate p in the image I and at the instant t. We will use $I_t(p)$ for the output pixels and $I_{t-1}(q)$ for the input pixels
- $W(p)$ is a window associated to the calculation of an output pixel $I_t(p)$ and containing m input pixels

Definition An image processing algorithm is a transformation defined on a n-dimensional discrete domain D^n, n is usually $n = 2$ or 3 but can be higher in some applications (e.g.

[a]This pipeline can be either explicitly realized or by using a Very Long Instruction Word (VLIW) architecture which parallelizes the operations thanks to a fixed schedule determined at the compilation of the program

applications using a pyramidal structure to compute a texture). Given the input and output images, I_{t-1} and I_t defined on D^n, each output pixel $I_t(p)$ is obtained by applying a function f to a window $W(p)$ of m input pixels.

$$\forall p,\ I_t(p) = f(W(p))$$

The window $W(p)$ contains pixels whose coordinates depend on p. We can classify the image processing algorithm with respect to two features:

- the kind of dependences between the input and output images and
- the formula giving the coordinates of the pixels needed to compute $I_t(p)$.

1.1.1.2 Parallel and Sequential Algorithms

Given the kind of dependences between the input and output images, the algorithms can be classified as:

- Parallel algorithms
- Sequential or recursive algorithms.

In **a parallel algorithm**, the calculation of an output pixel does not depend on the previously calculated output pixels, so that $W(p)$ contains only pixels of the input image I_{t-1}.

In **a sequential algorithm**, the calculation of an output pixel depends on the previously calculated output pixels. In this case

$$W(p) = \{I_{t-1}(q_1), \ldots, I_{t-1}(q_{m_1})\} \bigcup \{I_t(p_1), \ldots, I_t(p_{m_2})\}$$

with $m_1 + m_2 = m$. This means that the windows of pixels $W(p)$ contains m_1 pixels of the input image and m_2 pixels of the output image which have been previously calculated with respect to $I_t(p)$.

1.1.1.3 Application with affine or non-affine array references

The image processing algorithms can be classified with respect to the formula giving the coordinates of the pixels needed to compute the output pixel $I_t(p)$.

Let $g(p)$ and $h(p)$ be the coordinates of two needed input pixels. If $g(p)$ and $h(p)$ are affine they can be written as

$$g(p) = Ap + b$$

$$h(p) = Ap + c$$

and the two corresponding array references $I_{t-1}(g(p))$ and $I_{t-1}(h(p))$ are said affine themselves.

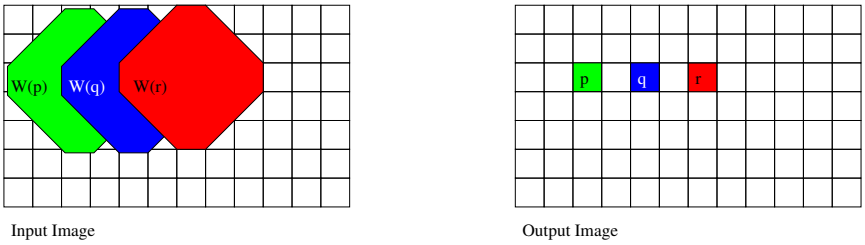


Figure 1.1: For uniform memory accesses the window of input pixels associated to the calculation of an output pixel $I_t(p)$ has invariant shape and size with respect to p

Definition It is possible to compute a **distance vector** between the two array references $I_{t-1}(g(p))$ and $I_{t-1}(h(p))$ as the difference between $g(p)$ and $h(p)$

$$d_{gh} = b - c$$

Remark If all the array references of an image processing algorithm are affine the window $W(p)$ associated to the computation of each output pixel preserves its shape and size for all the duration of the algorithm execution.

If the array references are not affine, $W(p)$ size and shape depend on the position p of the output pixel $I_t(p)$ to be calculated.

Example Let consider an image transformation f defined as follows:

$$I_1(2i, 2j) = f\left(I_0(2i, 2j-2), I_0(2i-2, 2j), I_0(2i, 2j), I_0(2i, 2j+2), I_0(2i+2, 2j)\right)$$

Figure 1.1 shows that, due to the affinity of array references, the windows of input pixels associated to the calculation of an output pixel $I_t(p)$ are equal by translation for different value of p.

1.1.1.4 Algorithm representations in a C-like code

An image processing application is intrinsically iterative and can be represented as a loop nest transforming a set of pixels into another according to a unique law. The loop nest depth depends on the multidimensional data to process.

Definition Let consider the iterative application **A** represented through a perfectly nested loop **L** (and given in table 1.1).

S and **T** are assignment statements.

A loop nest which assignment statements have affine array references is said to be affine.

In such a loop nest, the spaces of the input and output coordinates and the iterations space D^n are affine, which means that they are equal by translation. As a consequence, in an affine loop nest, it is actually possible to re-use the same memory space to store the input and output data if the data dependences are respected.

```
L: for 1 ≤ i ≤ N
    L: for 1 ≤ j ≤ M
      S: I1(i,j) = f(I0(g1(i,j)),...,I0(gm(j)))
      T: I0(i,j) = I1(i,j)
    end
  end
```

Table 1.1: Table presenting a generic loop nest

In an affine loop nest the loop bounds are usually constant or manifest, i.e. they are unambiguously determined at the compile-time. This enables several compiler optimizations, such as loop transformations and parallelization (cf. paragraph 1.3.2.2 and all the papers cited in it).

1.1.1.5 The uniform and non-uniform memory access

Definition Two memory accesses are said uniform when they require the same number of processor cycles to be performed.

Definition An image processing application and the loop nest representing it are said to be uniform, if they contain only uniform memory accesses.

In the frame of polyhedral (or hyper-plane) method (see paragraph 1.3.2 for more details), it is possible to apply some loop transformations to an affine loop, in order to reorganize both the array references and the data layout into the memory and, thus, to allow uniform memory accesses.

Very few studies (cf. paragraph 1.3.2.3) attempt to solve the problems of loop transformations and memory optimizations for a code with non-affine array references.

In this work we are interested in **parallel image processing algorithms with either affine or non affine array references**. In particular, we want to adapt some of memory-aware transformations, used to optimize affine loop nests, to the case of the non-affine ones.

The iterative behavior of the image processing algorithm is well realized on pipelined or systolic architectures. In the next paragraphs we will describe these kinds of architectures.

1.1.2 Parallel architecture for the image processing

By using the Flynn's taxonomy [8], we can describe the most two important parallel architecture paradigms of image processing, which exhibit, respectively, data and task parallelism.

The data parallelism can be achieved with a Single Instruction Multiple Data (SIMD) architecture and the tasks parallelism can be achieved by using a Multiple Instructions Multiple Data (MIMD) architecture.

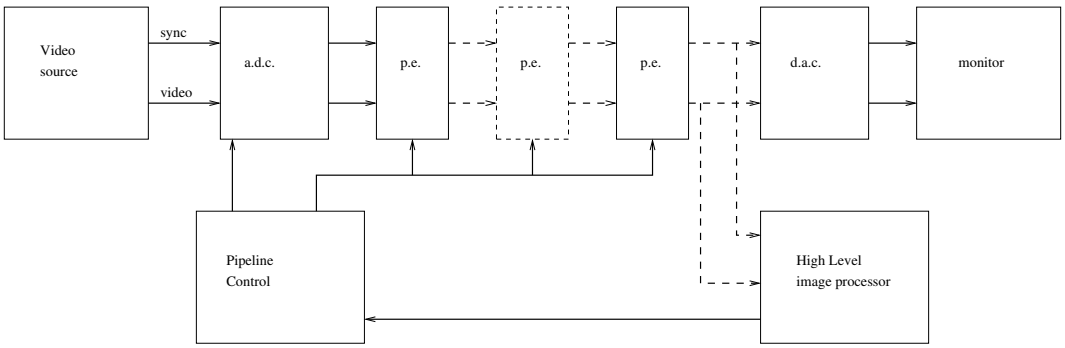


Figure 1.2: A pipelined image processor, (figure from [10])

Another parallelism technique widely spread and very adapted to the image processing is the use of pipelined processing units. These units allow iterating a set of complex instructions on a flow of data. By pipelining instructions either data or task parallelism (or both of them) can be achieved.

The rest of this paragraph shortly describes the pipelined architectures.

1.1.2.1 Pipeline

A pipeline structure consists of a chain of tasks, each one realized by a processor (or a functional unit). The parallelism is obtained by sequentially streaming data through the functional unit pipeline, i.e. by using the output of a unit as input of another unit [9].

An example of a pipeline image processor (taken from [10]) is given in figure 1.2. As stated in [10]: "the raster scan analogue image from the video source is digitized by the analogue to digital converter (a.d.c.) and transmitted with a synchronization signal (sync) to the first processing element (PE) of the pipeline". Each PE processes only few lines of the image frame. An example of a pipelined processing element is given in figure 1.3. Here the processing element performs a filter on a 3×3 window of pixels. The input image is rastered by lines and the first output pixel is produced after the pipelined has been filled (i.e. it has received the first 2 lines plus three pixels).

Several pipelined architectures have been realized to perform image processing: some with a unidirectional [11] and other with a multi-directional data flow [12]. In this last kind of pipelines, a combination of different directions allows to implement a pipeline of variable length.

A particular category of pipelined architectures are the **Systolic Arrays**, which are multi-dimensional pipelines[b] [13].

Figure 1.4 describes the functioning of a systolic array computing a matrix multiplication.

The first difficulty concerning the systolic array is to choose the basic structure of the architecture; this is strictly related to the problem of 'how to feed the processing elements

[b] Usually 2-dimensional

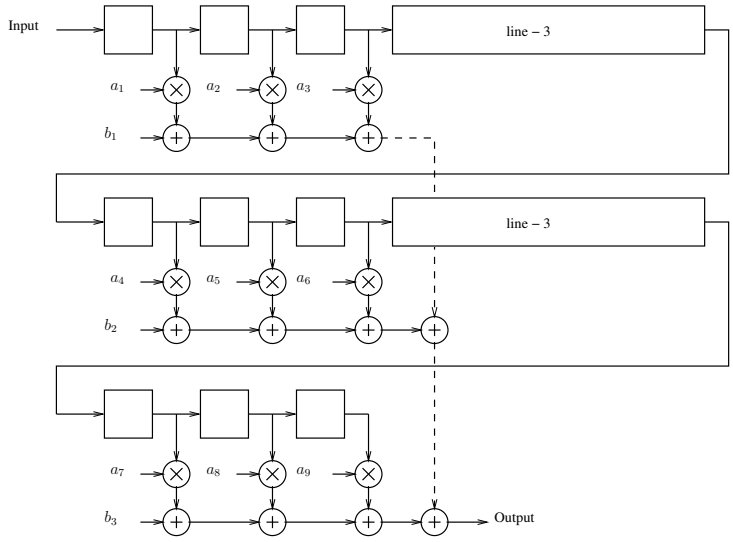


Figure 1.3: Example of a processing element in a pipeline, realizing a pipeline itself.

with data'. In figure 1.4 we have a squared systolic array whose elements are connected with the top, bottom, left and right neighbor. In [14], a different way of adapting the architecture structure to the target algorithm, is presented.

Another important problem, that has kept the most of interest for the automatic design, is: "how to map onto a finite number of physical PEs a larger number of the pipeline instructions". In the example of the figure 1.4, the chosen solution is the most obvious: there are as many processing elements as couples of values to be processed.

The problem of mapping applications onto a fixed-size systolic array is solved with a projection method, which finds the optimal placement of the instructions in time (scheduling) and space (mapping).

There are two possible ways of mapping applications on a fixed-size systolic arrays:

- Divide the instructions to be performed in blocks whose size corresponds to the number of available processing elements [15]. Then process in parallel the instructions of a block on the different available processing elements. This method is called the Locally Parallel and Globally Sequential (LPGS) mapping.

- Divide the instructions to be performed in blocks. Associate a virtual processing element to each block. Further divide the blocks in independent groups (whose number is equal to the number of available physical processing elements). Finally, process the different groups in parallel on different physical processing elements and the blocks of a same group, sequentially on the same physical processing element [16]. This method is called the Locally Sequential and Globally Parallel (LSGP) mapping.

The number of processing elements directly affects the temporal performance and the area cost of the architecture.

The different PEs in a pipeline or in a systolic array can realize either the same task, and in this case the corresponding architecture is a SIMD structure, or different tasks, and in this case there is an inter-task parallelism which includes the resulting architecture between the MIMD structures.

In this section we have presented the image processing classification with respect to the array references that they contain. An image processing application is (usually) an iterative algorithm that can be described as a loop nest. If all the array references that it contains are affine, than the image processing algorithms and the corresponding loop nest are said to be affine themselves. An affine loop can be transformed in order to reorganize its array references and the layout into the memory of the accessed data. This can lead to uniform memory accesses, i.e. memory accesses which require the same time to be performed.

In this section we also presented the parallel architecture paradigms used in image processing. The most important are the pipelined architectures. The systolic array are one of the most studied image processing architectures.

To conclude, an image processing system can be described at different abstraction levels. In our work, we have considered the algorithmic and the structural levels and the process that allows passing from the first to the second one: the High Level Synthesis.

1.2 The High Level Synthesis in the design method history

The High Level Synthesis (HLS) allows extracting a parallel, timed and structural model from a sequential, untimed algorithmic model. The HLS represents a forward step in the process and history of the design of the integrated systems.

In order to handle the designs complexity, the abstraction level of the input model has rose more and more for the last decades. As stated in [17], three historical phases can be distinguished:

- **The capture-and-simulate phase (from 1960s to 1980s)**, during which it was mandatory to realize the hardware in order to capture and validate its behavior through a simulation.

- **The describe-and-synthesize phase (from late 1980s to late 1990s)**. During which two abstraction models were introduced in order to validate the artifact behavior before its realization. The two models were the functional model and the gate-level structure. In the early 90s a new abstraction model was introduced between the previous ones: the Register Transfer Level (RTL) model, which gives the combinatorial logic realizing the target function and, at each clock-cycle, the status of the internal registers.

- **The Specify, Explore-and-Refine phase**. Since the early 2000s, a new trend has been taking place: to specify models at different abstraction levels and to

refine them by successive synthesis step. A new model above the RTL has been introduced: the System Level (SL) model, which gives a description of the artifact with respect to the communication system used: the busses, the links point-to-point, etc. In this framework, each model at different abstraction levels (the SL, the RTL and the gate level) is a refinement of the model at the previous level and the specifications for the current specify-explore-and-refine step.

Until nowadays, however, and according to [18], "the Electronic Design Automation (EDA) community has not succeeded in establishing a new layer of abstraction universally agreed upon (...)" as the RTL. Alway according to [18]: "The International Technology Road-map for Semiconductors (ITRS) in 2004 placed SL Design "a level above RTL including both HW and SW design". SL Design is defined to "consist of a behavioral (before HW/SW partitioning) and architectural level (after)" and is claimed to increase productivity by 200K gates/designer-year".

1.2.1 The Y-chart and the synthesis levels

In order to give an idea of the possible models and processes implied in the current hardware design flow, we describe the Y-chart, implemented by D. Gajski and presented in [17].

According to the Y-chart (see figure 1.5) we can state that any design can have three models, each one focusing on a design property:

- the behavioral model, pointing out the functionality of the system;
- the structural model, giving the structure of the macro-blocks that realize the target functionality and
- the physical model, which adds the dimensionality to the structure, i.e. information on the area occupancy, the temporal performance and the power consumption of the target structure.

In figure 1.5 these models are represented through three axes forming a "Y". This figure is adapted from [17, 19].

As pointed out by figure 1.5(a), for each model, there are several levels of abstraction according to the components granularity of the model. For example, at the System Level, the basic components are processors, co-processors, busses and memories; at the processor level, the basic components are the functional units (such as ALUs) and some storage elements (such as the register files); at the logic level, the basic components are the logic gates and the flip-flop and, finally, at the circuit level, the basic components are the transistors.

The abstraction levels on the Y-chart are represented by concentric circles which intersect the "Y" axes. As shown by figure 1.5(b), each motion along or across the "Y" axes represents a step of the specify-explore-refine design flow. For example: a radial movement along an axe represents a refinement or an abstraction of the considered model; a circular movement along an abstraction level is called *synthesis*, when it leads from a

behavioral to a structural model; *analysis*, in the inverse case; *generation*, when it brings from a structural to a physical model and *extraction*, in the inverse case.

At each abstraction level, it is possible to optimize the considered model by "exploring" different implementations.

As shown by figure 1.5(c) and by starting from the more abstract model, there are three possible syntheses:

- **The System Level Synthesis (SLS)**, which consists in passing from an system behavioral model to a system structural model, with programmable microprocessors, customized hardware, IPs, memories, buses and interfaces and is concluded with the synthesis of communications (see figure 1.6(a)). Several tools are available to run efficient Design Space Explorations at this level for SLS (Artemis[20], Metropolis[21]).

- **The High Level Synthesis (HLS)**, which consists in passing from a behavioral description of a processor to a structural RTL logically synthesizable model as shown in figure 1.6(b) and presented in [5].

  The synthesizable structural model contains a control unit (which is a Finite State Machine (FSM)) and a Data Path Unit, which executes the application under the FSM control.

- **The Logic Synthesis**, which allows to pass from a RTL model, through a gate level model, to a technology-specific network

On a same abstraction level and thanks to the synthesis it is possible to have models which are intermediate between the behavioral and the structural and have a different time granularity. These models are said to be approximate-timed, because they integrate more information than an un-timed (algorithmic model) and less than a cycle-accurate (implementation) model.

In our work, we are interested to the High Level Synthesis.

1.2.2 The abstraction level of a HLS input model

An efficient exploration front-end of the HLS is able to find a good trade-off between the parallelism level, the temporal performance, the power consumption and the area occupancy of the system. This exploration is related to the abstraction level of the input model to be optimized and also to the semantic of the used description language.

An un-timed behavioral code (usually written in C or C++) has a simulation time from 10 up to 100 times faster than a HDL[c] synthesizable model. This enables the possibility to rapidly change the algorithm or the chosen structure of the target hardware [22]. On the other side, even with HDL language it is possible to describe an algorithm in a behavioral (and thus in a non-synthesizable) manner. For this reason, in our opinion the HLS input problem does not concern the language used to describe it but its abstraction level. Nevertheless, the standard C needs some expedients to enable an optimized HLS.

[c] Hardware Description Language

In [23] it is stated that: “C/C++ is a poor choice for specifying hardware” and that “C-like thinking is actually detrimental to hardware design". The problem of C/C++ concerns first of all the impossibility to express concurrency and the hardware data-types[d]. Another limit of the C/C++ language is the memory model, which is a very flexible access-random memory, i.e. all the memory locations are equally cost to access. While in a hardware model the memory hierarchy introduces delays that may prevent to achieve the required temporal performance.

Two different approaches exist to overcome the limitations of a C-like input code for HLS:

- to add parallel constructs to the language, this is the case of the SystemC (which is actually a HDL-like language) or other software-like language that provides construct to dispatch collection of instructions, as for example Handle-C [24], Spec-C[25] or Bach C [6].
- to let the compiler the burden of inferring parallelism.

In both these cases, it seems a widely spread opinion that the results are all the better since the C-code to be synthesized into parallel handware is written from the architect point of view and contains knowledge about the target architecture.

However, the higher the abstraction level of the input model (i.e. un-timed and sequential) the greater the possibility we have to:

- give a model which is not architecture-specific
- make simulations from 10 to 100 times faster [6]
- reduce the workload to introduce changes into the hardware structure or the algorithmic functionality

These features enable the Exploration of a larger Design Space and thus significantly rise the possibility to find an optimal hardware solution [22]. Furthermore an automatic flow which infers a RTL from a behavioral model consequentially speeds up and secures the error-prone task of manually transforming a behavioral model into a hardware structure.

In the HLS framework, particular cases are the loop-based C- tools [22, 26] which mix the two approaches, by giving the possibility to express parallelism and data-width through extensions of the standard C and pragmas, but also automatically infer parallelism through loop transformations.

1.2.3 The HLS loop-based tools

In most of HLS tool which take as input an un-timed behavioral model, we can distinguish two phases of the HLS process [27, 28, 29, 30]:

[d] in particular, it is impossible to express a one bit-bool type

- A behavioral synthesis, which optimizes, transforms and partitions the input code and finally produces an architectural structure usually containing the control and data path unit;

- A structural synthesis, which results in a generation of a detailed data-path and controllers net-list related to the RTL library. At this step, the back-annotation techniques allow a fast estimation of the performance and area occupancy of the final hardware by taking into account a target technology.

Let consider the HLS tool used to support the work of the presented PhD thesis and described in [26, 27, 31, 32]. It is a loop-based C-tool, which takes as input a C-code containing a series of loop-nests.

The corresponding synthesis is based on a target architecture template (figure 1.7 and paper [26]) which contains a Very Long Instruction Word (VLIW) processor and a Systolic Array. The used HLS tool focuses on the synthesis of the Systolic array.

The systolic processor array contains interfaces to the VLIW processor and to the external main memory, internal memories and interconnected processing elements (PEs). The PE architecture is shown in figure 1.8 and contains a loop-specific instruction sequencer and a data path consisting of functional units (FU), distributed registers and sparse interconnect customized to the loop nest [26].

The HLS flow of the studied tool has two phases [27]: a **behavioral synthesis** which allows passing from the iteration space of an un-timed loop-based C-code to a timed space of the platform presented in figure 1.7, and a **structural synthesis** which leads from the parallel program to the hardware.

The behavioral synthesis goes through the following steps:

- The dependence analysis, which identifies the inter-iterations dependences.

- The code optimization based on loop transformations (Tiling in particular). The tiling divides the iterations into blocks.

- The placement of a n-deep iteration space in time and space.

- The recurrence-based control generation, which allows the loop-specific sequencer of a processing element to compute the coordinates of the iterations, the addresses to be used in the main memory, the coordinates in the cluster of iterations tile processed and other informations concerning the data communications. A part of this information is directly read as an input in order to simplify the iteration control.

The placement of the n-deep iteration space in time and space is performed through a loop transformation:

$$T = \begin{pmatrix} S \\ \dots \\ \Pi \end{pmatrix}$$

where S is a scheduling function, i.e. the placement of iterations in time, and Π is a mapping function, i.e. the placement of iterations onto the systolic arrays.

This technique has been previously presented in [15]. In the case of the used HLS tool it is performed by assigning each iteration in the space to an (n-1)-dimensional array of virtual processing elements, each one corresponding to a tile of iterations. The virtual PEs are then divided into clusters and the execution of cluster is assigned to a specific physical processing element. This technique is presented in [27, 16] and is directly inferred from the loop tiling.

Furthermore, in the used HLS tool, the iteration scheduling on each physical processing element is inferred according to a tight scheduling formula [33].

The structural synthesis goes through

- The Functional Unit allocation
- The Operations mapping and scheduling
- The Data path and iteration control generation
- System synthesis, which consists in the synthesis of several processing elements communicating through point-to-point connection or internal memories (see figure 1.7).

The Functional Unit allocation allocates the least-cost set of Functional Units by taking them from a library. Each functional unit can handle several kinds of operations. The allocation is performed by solving a mixed integer linear programming problem which minimizes the cost due to the allocation under the following constraints:

- The number of assigned operations is inferior or equal to the number of allocated FUs per initiation interval (II)[e]
- The number of assigned FUs which can handle a given type of operations is superior (or equal) to the number of operations of this type to be performed.

The Operations mapping and scheduling are performed by a VLIW compiler , Elcor [32], which compiles the operations on a programmable abstract VLIW. Once the optimized architecture is chosen only the required data-path and interconnect are generated. The compiler performs a modulo-scheduling by respecting the user-defined II and minimizing the area cost with an efficient mapping.

1.2.4 The input C-code of the used HLS tool

Loop-nest based C codes are particularly adapted to describe multimedia applications.

The input code is synthesized as a Pipelined Processing Array (or systolic architecture) on which it is possible to process more sets of input data (e.g. different frames in

[e] i.e. the initiation interval is the user-defined number of cycles between the start of the execution of two successive iterations on the same hardware. Suppose that there are two operations of a type “o", they can be handled by a unique FU of a type “f", if II=2.

an image flow). Each loop-nest of the code is synthesized as a Processing Element, which contains a control unit and data-path[f]. The control unit iteratively executes operations on the data-path which corresponds to the loop core.

As specified in paragraph 1.2.2, even if the usage of C language maintains high the abstraction of the input model and, thus, allows an efficient design space exploration, there are some limitations due to C semantics and philosophy.

- A problem of the C-language is the impossibility to have a single bit signals or signals which contains a number of bit different from 8, 16, 32 and 64 (which corresponds to the C data type char, short, int, long). The used HLS tool provides the possibility of specifying the signal bit-width through a pragma and, recently, with a C++ class (ac_int<... nbr. of bits ...>).

- Another important problem of the C language with respect to the evaluation of the inferred parallel model is its flexible access-random memory. Indeed, in a C-code all the memory locations have the same cost to be accessed.

  In order to overcome this problem, the HLS tools provide an expedient consisting in specifying a HLS compiler parameter (-DMEMORY_LATENCY=...) and a function as follows:

```
unsigned char delay(unsigned char dummy){
        #pragma bitsize delay
        #pragma bitsize dummy
        return dummy;
}
```

  This function is generated as a clock cycle delay and can be iteratively called in order to simulate a memory latency of more cycles.

- Finally, a C-code is un-timed and sequential, which is, on one side, desirable because of the possibility to extend the design space explored, but, on the other side, severely limits the possibility to introduce concurrency between process.

  The first way to introduce parallelism is to pipeline the iterations of a loop nest on the same hardware, in order to overlap parts of iterations with each other.

  Another way is to apply tiling, in order to parallelize the process of independent regions of the manipulated data set, which means to use more PEs for a single loop-nest.

  On the other side, independent loop-nests are processed in parallel on different PEs but, the used HLS tool also offers the possibility to parallelize loop-nests which depends on each other, by using streams.

  The **streams** are elements of communication between two loop nests, the one producing and the other consuming data. They have different implementations:

[f] i.e. a set of FUs

the one for the sequential and the other for the parallel model. In the sequential model, they are used to store all the data produced by the first loop nest until the second loop nest uses them. In the parallel model, they include a (handshake) communication protocol and a few temporary registers. The data produced by the first loop nest are either stored in the temporary registers until the second loop nest is ready to start or directly passed to the second loop-nest. For this reason, not all the produced data have to be stored and the two loop-nests, producing and using data, can be executed in parallel.

If inter-dependent loop nests communicate through memories they could be parallelized by using a **double buffering mechanism** [34], which consists in duplicating the memory buffer in order that the loop producing data fills the first buffer while the loop consuming data reads the second buffer and, vice-versa, the loop producing data fills the second buffer while the loop consuming data reads the first buffer.

In our work we encountered the problem of inter-dependent loop communicating through memories, but the size of the required double buffer would have been too important. Thus we have solved the problem as shown by figure 1.9.

We have applied tiling to each loop nest of the initial code (figure 1.9(a)) which accesses a memory of N elements, in this way we have obtained the code of figure 1.9(b). By merging the more external loops (on tiles) we obtain the code in figure 1.9(c) in which we can use a memory containing only $\frac{N}{T}$ elements. By using a double buffering mechanism on the smaller obtained memory, the two loop nests can be parallelized.

A problem arises from these transformations: the obtained code (figure 1.9(c)) does not contain perfectly nested loops, thus it is not synthesizable, according to the rules of the used HLS tool. In order to synthesize this code and as it was feasible, in our case, we have displaced the more external loop out of the synthesized systolic array, in order to be performed by the VLIW controller.

In other cases this solution could not be possible and it would be helpful if the vendor of the HLS tool proposes a memory, which (as for the streams) has two versions, one for the sequential model and one for the parallel model (including a synchronized access protocol).

1.2.5 The parallelism levels inferred by the HLS commercial tool: inter-operations, intra-loop and inter-loops parallelism

Thanks to the previously presented optimizations and design flow it is possible to have three levels of parallelism: the inter-operations, inter-iterations and inter-loops parallelism, which are inferred by the HLS tool under some user's constraints [31].

To detail the possible levels of parallelism let consider the loop-based code in figure 1.10. The notation $op_i(op_j, \blacksquare)$ means that the execution of the operation op_i depends on the operation op_j and uses the hardware resource $\blacksquare$. For example, $r_s(RQ : w_s, s)$ (in

the figure) means that the reading access to the stream s depends on the writing access to s made in the loop RQ. The hardware resource used is the stream s.

1.2.5.1 The inter-operations parallelism

The operations of a loop core can be scheduled sequentially or in parallel depending on the hardware resources and the inter-operation dependences. Figure 1.11 shows two possible scheduling for the operations in the RQ loop core of the code in figure 1.10 and their possible hardware realizations. The amount of resources used depends on the number of operations executed in parallel.

The inter-operations parallelism is automatically inferred by the HLS which found the solutions meeting the user's constraints on the temporal performance and using the minimum amount of hardware resources.

The hardware instantiated represents the processing element associated to the loop core.

1.2.5.2 The inter-iterations parallelism

The iterations of a same loop nest can be pipelined on the same processing element. To avoid conflicts on streams and memories accesses, a minimum interval of time (II-Initiation Interval) has to pass between the beginning of two successive iterations. The user can force the inter-iterations (and the inter-operations) parallelism by fixing the II as the minimum possible.

Figure 1.12 shows an example of pipelined iterations for the loop nest RQ. The hardware instantiated at this step counts a data path and a controller used to pipeline the iterations; it represents the processing element associated to the loop nest.

1.2.5.3 The inter-loops parallelism

Different loop-nests run in parallel but their communications (for example a memory or a stream access) create inter-loop dependencies which have to be respected. Two communicating loop nests can exchange data through a stream and, at the same time, be synchronized by the handshake communication protocol of the stream in order to execute in parallel.

Figure 1.13 shows an example of loop nests parallelism between the loops RQ and FH.

The biggest problem with the used loop-based High Level Synthesis tool is that the micro-architecture ensuring the data transfer and storage has to be defined by the user and included in the input C-code. The choice of an optimize memory architecture is a difficult problem that has to take into account many possible solutions. Thus in the next section we will describe this problem, the existing kinds of solutions, their advantages and their limitations. The ultimate aim is to propose a comprehensive methodology able to apply the described memory-aware optimizations to the parallel image processing

algorithms with affine or non-affine array references. This methodology has to be used as a front-end of the used HLS tool and thus, it has to refine an abstract C-model towards a model including a set of memory-oriented optimizations.

1.3 The Data Transfer and Storage Management in Systems Design

One of the big challenges in the design of current digital systems is the Data Transfer and Storage Management (DTSM) [35, 1]. This concern is inscribed in the framework of the «memory wall» problem, which represents the gap between the memory bandwidth and the processor speed [36]. This gap increases despite the miniaturization trend of technology and needs to be solved by proposing new architectural and algorithmic solutions.

The DTSM is a critical point especially for the implementation of multimedia applications, such as image processing ones. Indeed, the growing concern about the image quality requires to handle more and more important image sizes (cf. High Definition TV).

The DTSM affects the area, the power consumption and the temporal performance of the hardware and, in the last decades, it has retained the attention of parallel compiler, parallel architecture and computer aided design research communities.

As stated in [7], two fundamental kinds of optimizations are needed in order to improve the DTSM:

- the architectural optimizations
- the algorithmic optimizations.

The architectural optimizations are aimed to create a memory hierarchy in order to facilitate the data access and to reduce the area of the circuit reserved to the data storage. The algorithmic optimizations are aimed to enhance some algorithm characteristics as spatial and temporal data locality, data parallelism, etc. With respect to the DTSM problem the algorithmic optimizations consist in re-ordering the data accesses and partitioning the data and the instructions involved in the target algorithm.

In the next paragraphs we will describe what is a memory hierarchy and which are the related optimizations. Then we will describe the code optimizations aimed to improve the DTSM.

1.3.1 The memory hierarchy

The different existing kinds of memories can be classified with respect to the access time, the access cost and the volume of data that they can store. Usually the fastest and most expansive memories are the smallest ones. A memory hierarchy consists in optimizing the use of the differnt memories by taking into account their phisical caracteristics. In particular, applications using a large amount of data can store them in a larger and

slower off-chip memory (usually a DRAM [37]). Then, to reduce the data access time, they can use smaller on-chip memories (usually SRAM) to copy part of the manipulated data next to the the computation units.

The access to the off-chip stored data can be delayed due to many factors which contribute to increase the "external memory latency" (cf. figure 1.14 and [38]). For example the data request of a computation unit can be queued due to other requests with a high priority.

Among the possible on-chip memories we can distinguish two kinds of memories:

- the cache memory
- the scratch-pad memory

The cache memory [39] creates a copy of the external data in a new memory space. A control mechanism establishes if the required datum is contained in the cache (cache hit) or is not cached yet (cache miss). When a cache miss happens, the missing datum (and may be its neighborhood) is fetched from the external memory.

The scratch-pad [40, 41] is a memory in which data are mapped after the scheduling of the instructions (and thus of the memory accesses). For this kind of memory it is not necessary to check for the data availability.

The difference between the scratch-pad and the cache memory is that the scratch-pad always guarantees a data access in a single clock-cycle, while the cache is subject to the cache misses.

Several mechanisms can help masking the external memory latency by using local memories. More specifically, the data pre-fetching [42], which consists in copying the data needed for the next task during the computations of the current task. The pre-fetching requires to instantiate an amount of memory that avoids the conflicts on memory access, i.e. an amount of memory in which it is possible to store the data used by the current task and to copy the data needed by the next task. An example of pre-fetching instantiating the double of internal memory amount is the double buffering [34].

1.3.1.1 Methods used to construct an optimized memory hierarchy

To implement an optimized memory hierarchy the designer has to develop an exploration method including:

- the **estimation** of the needed memory amount [43, 44, 45, 46]
- the **allocation** of chosen number and types (single-port, dual-port, etc.) of used memories [47, 48, 49]
- the **assignment** of the sets of manipulated data

The **estimation** of the needed internal memory amount depends on the data access pattern (and thus on the instructions scheduling). It is possible to run a coarse estimation before the instruction scheduling and a finer estimation after. In this last case the

estimation can take into account the lifetime of the variables or array cases accessed during the algorithm execution [45, 50].

Most of the techniques coupling the estimation before and after the instruction scheduling consist in computing a set of possible scheduling and choosing the one using the least internal memory amount [51].

The memory **assignment** consists in mapping data into the allocated internal memories. It has to solve three problems:

- the conflict caused by mapping different data into the same memory location [52, 53, 54]
- the reduction of the number of used memory ports [7, 55]
- the tradeoff between the area occupancy and the power consumption [56, 57]

The conflict on the data access can be solved in two ways: by mapping dummy elements into the conflicting locations [58]; by tiling data before the memory mapping. In this last case the tiling has to respect the internal memory size and mask the external memory latency [53, 59, 60, 61].

The number of memory ports depends on the mapping of sets of data accessed in parallel into the same internal memory. This problem can be solved through conflict graph [7] or stream buffers [55].

The area occupancy and the power consumption depend on the number and size of the used internal memories. In particular large internal memories storing contemporary more data arrays occupy more circuit area, while a larger number of internal memories causes a higher power consumption. Methods exist to find a trade-off between the two previous solutions [56, 57].

Other methods to reduce the power consumption are to reduce the memory bus activity by an optimized data encoding [62] or by reorganizing the data accesses [56].

1.3.2 The target code optimizations

The target code optimizations have been largely studied in the last 90s by the parallel compiler community. They are essentially aimed to enhance the temporal and spatial locality of data and to allow the data and task parallelism.

During the last decade, these techniques have been included into the research frameworks of Computer Aided Design and High Level Synthesis communities. Indeed their effect on the DTSM is critical, especially for applications permitting a customized memory subsystem. Most of previous works focus on loop transformations, which are particularly adapted to optimize an image processing algorithm, usually represented by a series of nested loops.

The first step of a code optimization flow is the data dependence analysis. Thus in this section we will first describe the possible methods to infer the algorithm dependences and then we will describe the loop transformations used to improve the DTSM.

1.3.2.1 The dependence analysis

The dependence analysis is aimed to find the data and instructions dependences of an algorithm. In our work we are only interested in the data dependences. This kind of dependences can occur between two data accesses to the same memory space. The data dependences can be caused by the data address only (static dependences) or can also be due to the data-value. In our work we only focus on static dependences. The data addresses, in their turn, can be affine or non affine (cf. paragraph 1.1.1.3).

According to the type of dependences and data addresses there exist different methods to infer the data dependences. We will consider the two following main methods:

- the profiling of the code during its execution
- the data dependences analysis through a formal method

The first method consists in constructing a trace of the data accesses by executing the target algorithm [3, 63, 64]. This method associates a footprint, i.e. a set of accessed data, to each instruction or set of instructions and it does not impose a condition on the address affinity.

The formal dependence analysis consists in deducing the lexicographical order of the data accesses [65, 66] by solving linear programming problems [67]. This method describes the dependences by distance or direction vectors and impose that the data references are affine. For example in [65] and [66], the data dependences in a loop nest are represented as vectors, which coordinates are affine to the iteration indexes (e.g. in a n-dimensional space a reference array is $\overline{v} = (v_1, v_2, \ldots, v_n)$). For each pair of dependent array references (e.g. $\overline{v}$ and $\overline{w}$, with $\overline{v}$ dependent on $\overline{w}$ reference), the dependences can be described through:

- a **distance vector** which calculates the differences between the two vectors representing the array references; e.g. $\overline{d} = \overline{v} - \overline{w} = (v_1 - w_1, v_2 - w_2, \ldots, v_n - w_n)$ or
- a **direction vector** which indicates if the coordinates of the distance vector are less than, equal to or greater than 0; this vector is easier to compute but gives less information than the previous one; e.g. $\overline{d} = (d_1, \ldots, d_n)$ with $d_i = \begin{cases} = & \text{if } v_i - w_i = 0 \\ < & \text{if } v_i < w_i \\ > & \text{if } v_i > w_i \end{cases}$

Figure 1.15 gives an example of a perfectly nested affine loop, for which there are three true dependences due to the couples of array references $(A[i_0][i_1], A[i_0+1][i_1])$, $(A[i_0][i_1], A[i_0][i_1+1])$ and $(A[i_0][i_1], A[i_0-1][i_1-1])$.

The corresponding distance vectors are $(1, 0)$, $(0, 1)$ and $(-1, -1)$, while the directions vectors are $(>, =)$, $(=, >)$ and $(<, <)$.

After that the dependence analysis has been applied, it is possible to transform the input code by enhancing the data locality, the parallelism and other features related to the data storage and transfer. As the considered input code is a loop-based one, the code transformations are essentially loop transformations.

1.3.2.2 The loop transformations

A framework using the formal method analysis and proposing a set of code transformations is the polyhedral model [68, 69]. In this model a perfectly nested loop code is represented by a set of matrices. Methods exist to make a code perfectly nested [70]. A code transformation consist in multiplying all the matrices representing the target code by the matrix representing the transformation.

The transformations are divided in unimodular transformations, which are represented by an unimodular[g] matrix [71, 70, 72, 73] and the non-unimodular transformations [74], such as: the loop fusion [75], the loop pipelining [76, 77, 78], the loop unrolling [79] etc...

Unimodular transformation The transformations having a matrix representation T, defined and invertible on $\mathbb{N}^{n \times n}$, and, thus, with $det(T) = \pm 1$, are said to be unimodular transformations.

These transformations are applied to all the statements of a perfectly nested loop, at the same time. They transform the iterations distribution, the loop bounds and the array references, in order to enhance the data locality and, thus, allow uniform memory accesses. The unimodular transformations are: loop interchange (or permutation), loop reversal and loop skewing [72, 73, 30]. They are shortly presented in figure 1.16 and are legal if they respect the data dependence, i.e. if they preserve the temporal sequence of all the dependences.

Tiling The loop transformation allowing to apply the data and instruction partitioning is the **tiling**.

The problem of finding an optimal tiling was first introduced by [80] and is a double problem. On one hand it is necessary to chose an optimal tile shape [81] and on the other hand it is necessary to find an optimal tile size [82, 83, 84] (some authors have proposed to resolve both problems at once [85]).

The optimality of shape or size choice is defined with respect to the purpose of the tiling use. Basically we can distinguish two purposes: tiling for data locality enhancement [58, 86] and tiling for parallelism [87, 81, 88].

Tiling for data locality, i.e. for memory hierarchy, depends on: which architectural features are modeled (e.g. associativity, prefetching, etc...); which resolution methods are used (e.g. heuristic search based for example on genetic algorithms [89, 90], closed form solution, exhaustive search, etc.); which cost metric is used and if it is a direct cost (e.g. execution time with load balancing [59]) or an implicit one (e.g. cache misses [91, 90]).

Tiling for parallelism involves the search of the optimal tile size and shape with respect to a metric cost, namely the total execution time and the communications cost [92, 93].

The tiling validity According to [92], a valid tile is required to be:

[g] i.e. with a determinant $= \pm 1$

- **Bounded**, which means that the matrix representing it is non-singular.
- **Identical by translation**, which means that the matrix representing it is an integer matrix.
- **Respectful of the data dependences**, which means that the transformed dependences has to remain lexicografically positive.

The tiling advantages There are many advantages in the use of tiling. We re-call:

- The re-use of internal memory. Tiles of data used in different stages of the image processing can be stored in the same internal memory.
- The parallelism between independent tiles. As shown in figure 1.17, if the tile shape follows the inter-tile dependences, we can distinguish groups of tiles which do not depend on each other. They can be processed in parallel.

1.3.2.3 The non-affine array references

The previously presented transformations succeed in optimizing a large class of algorithms, however they require that the array references are affine, otherwise neither the polyhedral model nor the transformations leading to the uniform memory accesses are applicable.

Most methods assume that a non-affine reference causes a data dependence.

In [94], the non-affine array references are evaluated with respect to the related affine reference. For example, in the code of figure 1.18 there could be a dependence on the array a; but the first subscripts of the two references to the array (i^2 and 3 respectively) do not interfere since there is no integer for which $i^2 = 3$; the second subscripts of the two references ($2i$ and $2i + 1$ respectively) do not interfere because one refers to the odd locations and the other to the even ones. Thus, the two array references do not interfer with each other.

[3, 63] propose a method, that performs a run-time analysis, in order to find all the non-affine references and re-sort the corresponding instruction in order to create “bucket tiles" of data which are stored so that the access memory result uniform.

[64] proposes a method which applies tiling to a loop that may contain non-affine array reference. The associates to the different instruction tiles some footprints, i.e. blocks of needed input data, which can overlap with each other and have a variable size.

The major problem in this kind of solution is the cost of the mechanism used to transfer data from the external memory and to access internal memories. This mechanism has to take into account the size of the transferred (or accessed) blocks of input data. Yet the sizes of these blocks depend on the address of the output data to be computed. This is why the used mechanism can be very expensive in term of occupied area, power consumption and computation speed.

1.4 Comprehensive methodologies

A general approach in implementing a comprehensive memory-aware methodology should go trough the following steps:

- A phase during which the loop transformations are applied in order to create uniform memory accesses
- a phase for tiling, in order to enhance parallelism and allow data pre-fetching
- a phase of hardware-oriented estimations and/or optimizations

Comprehensive methodologies are the Data Transfer and Storage Exploration (DTSE) project [1, 35], EXPRESSION [95], PHIDEO [96], MMalpha [97] and [64].

Conclusion

In this chapter we presented the context of this PhD thesis, which is the High Level Synthesis applied to the image processing transformations. More in particular, we have focused on the problem of the Data Storage and Transfer Management for application manipulating a large number of data.

In the first part of this chapter, we have classified the image processing algorithms according to the array references that can be contained in code and that can be affine or non-affine. The considered codes are the loop-based ones which represent well the iterative behavior of image processing algorithms. The hardware architectures realizing such a kind of behavior are the pipelined structures, as for example the systolic arrays.

Some HLS tools allow extracting a systolic timed structure from an algorithmic loop based description by running a design exploration to find the best instructions scheduling and mapping among a set of analyzed solutions. Although this kind of tools obtain competitive results, they lack a design exploration to find an optimized micro-architecture for the data transfer and storage management. Thus, they require the user to specify the communication and memory architecture.

In the last decades many works have been published on the DSTM problem, due to its interest for different research communities. In particular, the first works on code transformations were run by parallel architecture and compilers communities and were focused on loop transformations to enhance data locality for parallelism. Successive works have shown the importance of the loop transformations on the data transfer and storage management [98] and have invested other domains of research, as for example the HLS. An affine loop based code can be transformed in order to reorganize its array references and the layout into the memory of the accessed data. This can lead to uniform memory accesses, i.e. memory accesses which require the same time to be performed. Another important loop transformation is the tiling which allows partitioning the iterations and the data of a loop nest. The resulting data and instructions partitioning can be exploited to explore an optimized micro-architecture for data storage and transfer. In particular, the tiling allows implementing a memory hierarchy by storing all the manipulated data in an external memory and copying only part of immediately used data next to the computation units. Some mechanisms exist to mask the external memory latency, such as the data pre-fetching.

However, the previous works on loop transformations were mainly limited to the case of loop having affine array references. The few works proposing a partitioning for loop having non affine array references are based on run-time dependence analysis and associate footprints of variable size to the different sets of instructions. These methods impose to instantiate a control on the data transfer and access which has to take into account the size of the blocks of data. The data block size, in its turn, depends on the specific set of instructions to be executed. Thus, the used control mechanism can be very expensive.

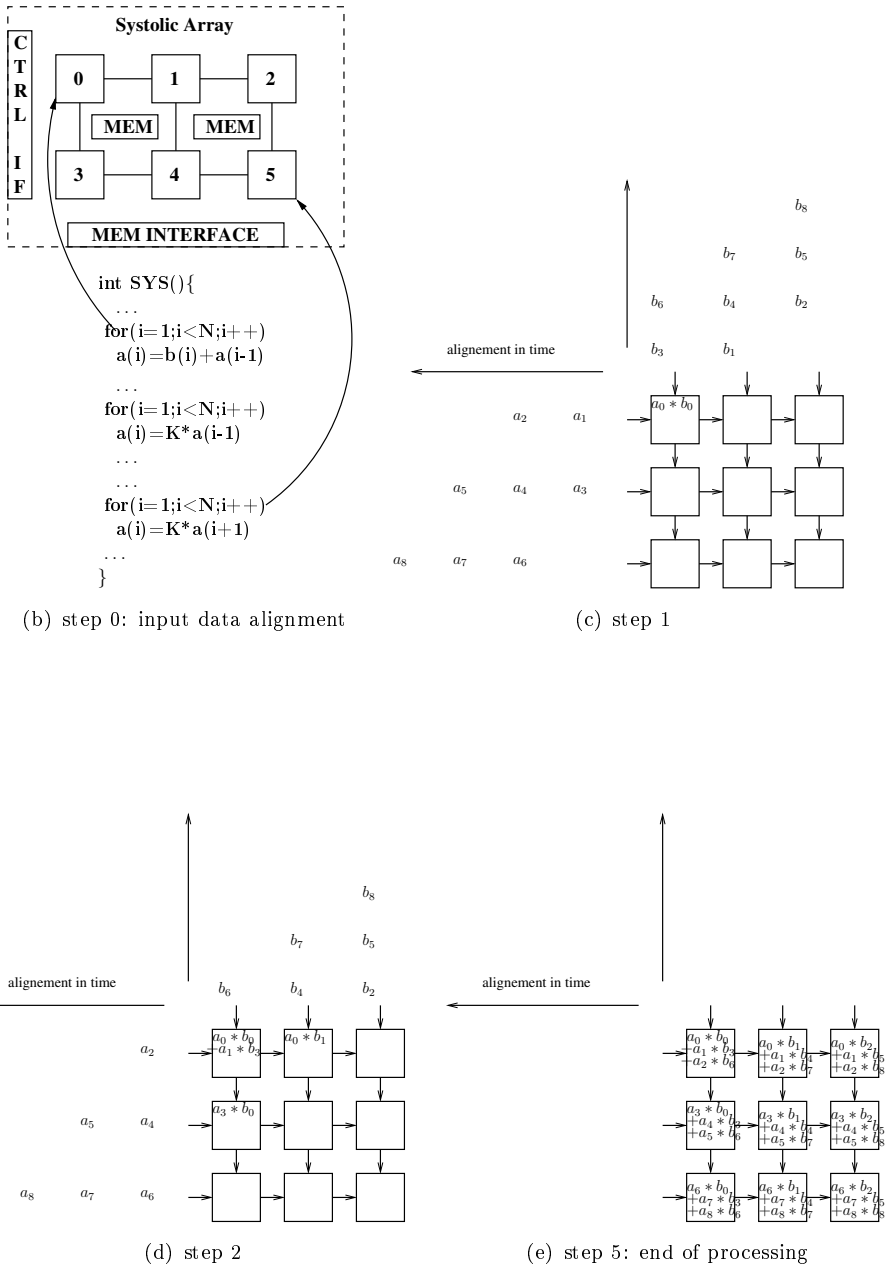
The aim of this PhD thesis has been to propose a method able to run a Design Space Exploration oriented to the optimization of the data transfer and storage management.

The corresponding developed tool has been used as a front-end of HLS in order to help the user to find an optimized memory micro-architecture. Our method is able to handle image processing applications with non-affine array references. It is able to apply a tiling which, on one hand, is based on a run-time dependence analysis and, on the other hand, uses disjoint and equal-by-translation tiles to partition the data and instruction sets. The non-affinity of the array references is taken into account by projecting the instruction tiling on the data tiling. This method leads to a memory micro-architecture that is, at the same time, adapted to the non-affinity of the array references of the application and has a cheap control on the data transfer because of the invariability of the size of transferred data blocks.

In the next part of this dissertation we will present the developed Design Space exploration tool.

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix} \quad \begin{pmatrix} b_0 & b_1 & b_2 \\ b_3 & b_4 & b_5 \\ b_6 & b_7 & b_8 \end{pmatrix}$$

(a) matrix multiplication



(b) step 0: input data alignment

(c) step 1

(d) step 2

(e) step 5: end of processing

Figure 1.4: Example of a systolic array computing a matrix multiplication

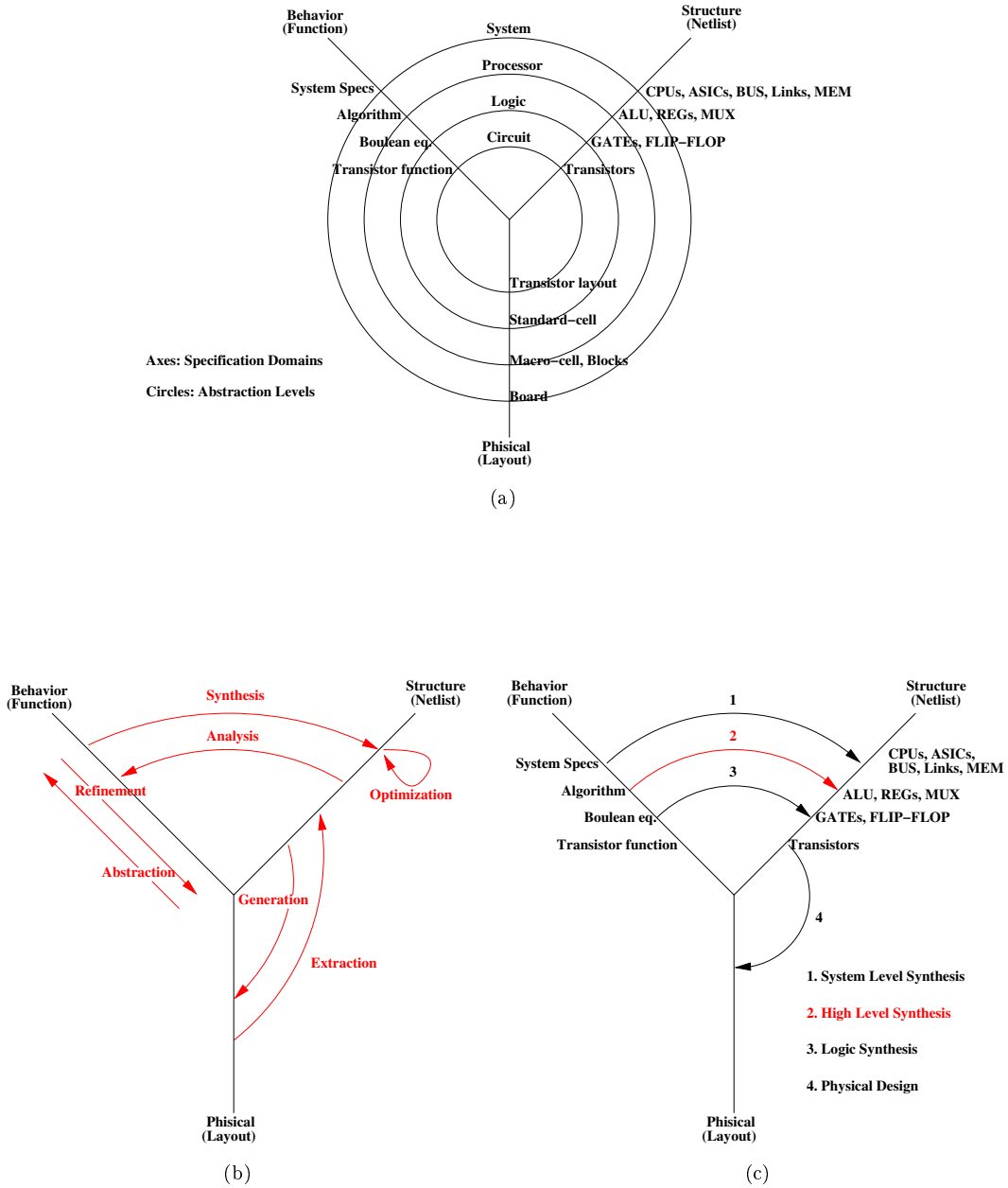


Figure 1.5: Y-chart, figure adapted from [17, 19]

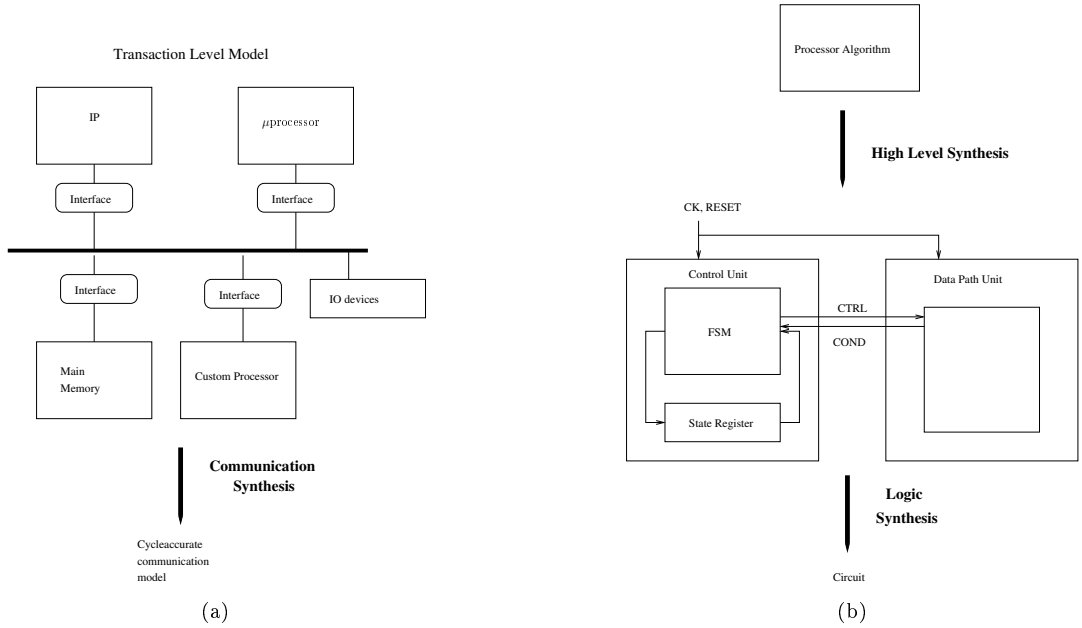


Figure 1.6: System Level design flow

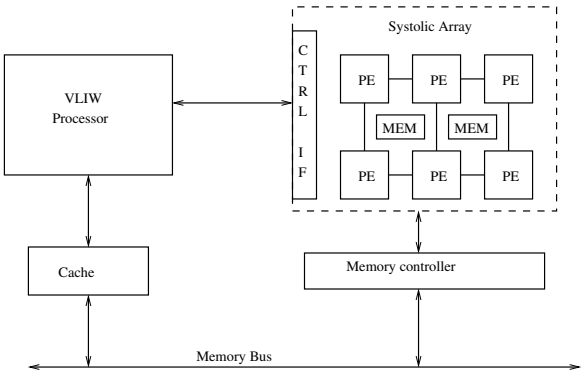


Figure 1.7: HLS tool architecture template, figure from [26]

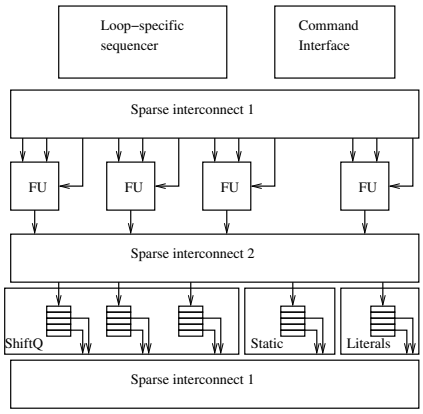


Figure 1.8: Processing Element architecture template, figure from [26]

```
P: For(k=0:N){
        ...
        ...
        IM[k]= ...
}
C: For(k=0:N){
        ...
        ...
        ...=IM[k]
}
```

(a) Input code

```
For(i=0:N/T){
        P: For(k=0:T){
                ...
                ...
                IM[i*T+k]= ...
        }
}
For(i=0:N/T){
        C: For(k=0:T){
                ...
                ...
                ...=IM[i*T+k]
        }
}
```

(b) Tiled code

```
For(i=0:N/T){
        P:For(k=0:T){
                ...
                ...
                IM[b1][k]= ...
        }
        C: For(k=0:T){
                ...
                ...
                ...=IM[b2][k]
        }
}
```

(c) Final code with a reduced internal memory

Figure 1.9: Code transformation to parallelize inter-dependent loop-nests communicating through memory. The transformation reduces the size of the internal memory used. “P" stands for “loop Producing data" and “C" stands for “loop Consuming data".

```
RQ  :   for (i=0:N) {
                c=a+b;                      // op1(,+)
                write_stream_s(c);          // w_s(op1,s)
                c1=2*c;                     // op2(op1,*)
                write_stream_s(c1);         // w_s(op2,s)
                a=c+a;                      // op3(op1,+)
                b=c+b;                      // op4(op1,+)
         }

FH  :   for (i=0:N) {
                 s1=read_stream_s();      // r_s(REQ:w_s,s)
                 s2=read_stream_s();      // r_s(REQ:w_s,s)
                 ad=s1+s2;                // op1(r_s,+)
                 internal_memory[ad]=i;   // w_a(op1,a)
         }
```

Figure 1.10: Loop-based pseudo code; in the comments "s" stands for stream and "a" for memory access.

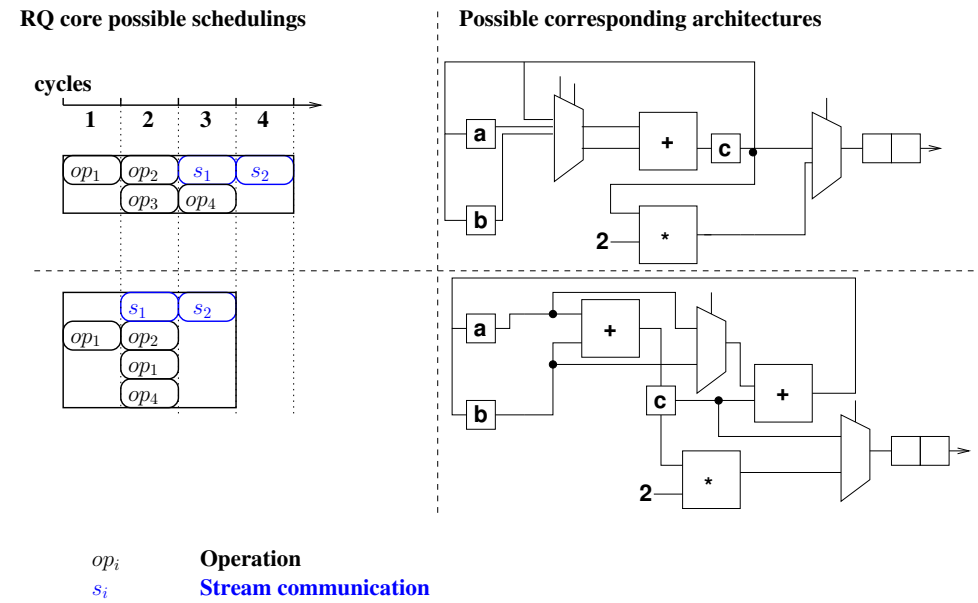


Figure 1.11: inter-operation parallelism

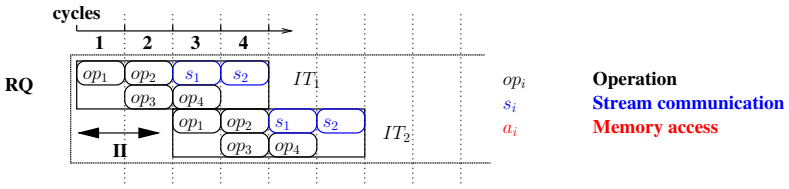


Figure 1.12: inter-operation and intra-loop parallelism

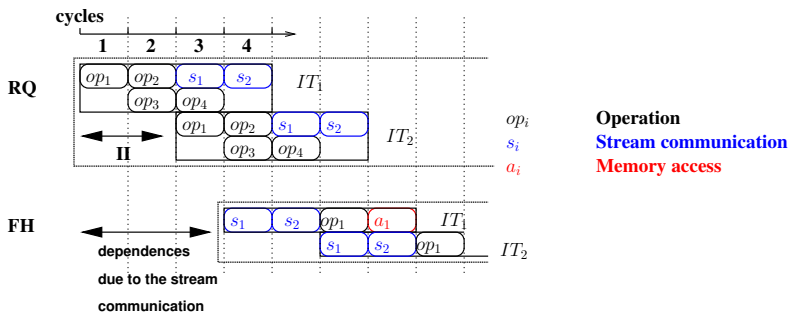
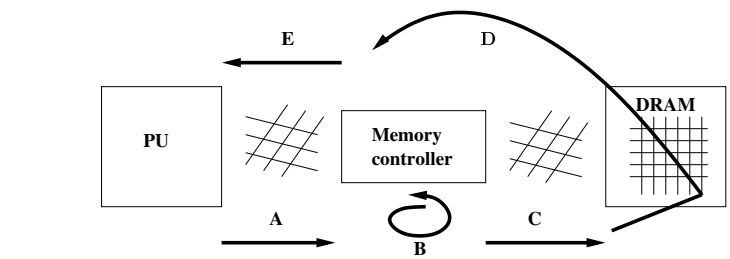


Figure 1.13: inter-operation, intra-loop and inter loop parallelism



A : possible delay due to a queued request and to the physical request transmission

B : convertion of a request into a commande sequence (can be queued)

C : Commands sent to a DRAM

D : delay due to the access mode (conventional, FPM, burst)

E: Transaction sent back to the Processing Unit (PU)

"DRAM latency"= A+B+C+D+E

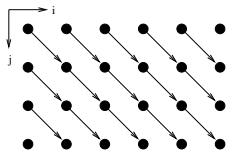
Figure 1.14: The DRAM latency, figure from [38]

```
for(i=0;i<N;i++){
  for(j=0;j<M;j++){
    A[i0][i1]=A[i0 +1][i1]+ A[i0][i1+1]+ A[i0-1][i1-1];
  }
}
```

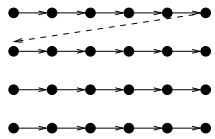
Figure 1.15: perfectly nested uniform loop

```
for(i=1;i<N;i++){
  for(j=1;j<M;j++){
    A[i][j]=A[i-1][j-1];
  }
}
```

(a) input code



(b) Data dependences



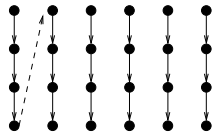
(c) Original data access pattern

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix}$$

(d) Loop interchange or permutation

```
for(j=1;j<M;j++){
  for(i=1;i<N;i++){
    A[i][j]=A[i-1][j-1];
  }
}
```

(e) output code



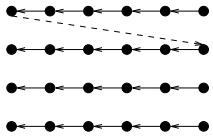
(f) Transformed data access pattern

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ -j \end{pmatrix}$$

(g) Loop reversal

```
for(i=1;i<N;i++){
  for(j=-M;j<-1;j++){
    A[i][j]=A[i-1][1-j];
  }
}
```

(h) output code



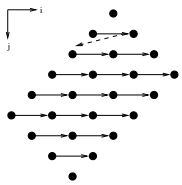
(i) Transformed data access pattern

$$\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ j-i \end{pmatrix}$$

(j) Loop skewing

```
for(i'=1;i'<N;i'++){
  for(j'=1+i';j'<M + i';j'++){
    A[i'][j' - i']=A[i'-1][j'-i'-1];
  }
}
```

(k) output code



(l) Transformed data dependence pattern

Figure 1.16: Unimodular Loop Transformations

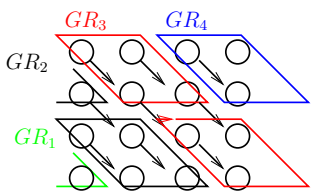


Figure 1.17: Example of parallelism between independent tiles

```
for i=10 : 20
  a[i^2][2i]=a[3][2i+1]+3;
```

Figure 1.18: example of a code with affine and non-affine array references

Part II

Research works

Chapter 2

MEXP: a Design Space Exploration tool

THIS chapter introduces MEXP, a Design Space Exploration tool aimed to find an optimized hardware realization for an algorithm with non-affine array references. MEXP uses the estimations of the hardware cost and performance as selection criteria. It chooses the pareto solutions in a set of possible candidates and, for each one of them, generates an optimized and synthesizable[a] *C-code by customizing a generic template. The customized C-models are afterwards synthesized by a commercial High Level Synthesis tool, which infers a RTL model from a C-code. The optimizations of the C-code improve the generated RTL model and are obtained through a loop transformation called tiling. The tiling divides the iteration and data spaces of a loop nest into disjoint and equal blocks. It has been used, in previous research, in order to optimize the memory accesses of the applications having affine array references. In this work it has been adapted to the applications with non-affine array references.*

Chapter contents

This chapter goes through the following contents:

- The motivations of this work
- An introduction to the developed Design Space Exploration tool
- A description of the tool flow
- A description of the target architecture template

[a] i.e. written according to the rules of the High-Level Synthesis commercial tool used to obtain the RTL.

2.1 Motivation

In the last 20 years, we have assisted to a massive trend towards the design automation of embedded systems, due to their increasing complexity.

Current automatic design flows (see chapter 1.2) refine an abstract model of a target application, through a series of steps. In particular the step inferring a Register Transfer Level (RTL) description from a behavioral model is called High Level Synthesis (HLS).

Each step of the automatic design includes a Design Space Exploration (DSE), which perform a fast estimation of the behavior of a large amount of solutions. The DSE chooses the solutions having a trade-off between the system performance and cost, without realizing them. Thus, on one hand, it eliminates the superfluous costs of useless or sub-optimal realizations and, on the other hand, it considerably rises the possibility to find the optimal solutions in a large set of possible candidates.

In our work we are interested into the memory aware optimizations, which can improve the area cost, the temporal performance and the power consumption of the whole system. In particular we are interested in pre-fetching and caching mechanisms which copy data close to the computation unit and, thus, can reduce or completely mask the time needed to access data. These mechanisms are made possible tanks to a data partitioning based on the temporal and spatial locality of the used data.

In the loop-based HLS tools, which take as input a loop-based C-code, the memory-aware optimizations can be obtained through some loop transformations, such as the loop tiling (see chapter 1.3).

The tiling partitions the iterations space of a loop nest and, as a consequence, the data accessed in the loop core, into disjoint and equal blocks. Thus, it can allow the data pre-fetching and the parallelization between the iterations of the loop. The tiling is used in many existing DSE methodologies, but as it is automatically inferred from the data dependence analysis, it requires that the data references of the loop core are affine (see section 1.3.2.2 of chapter 1.3).

Our contribution to the optimization of the data storage and transfer management problem has been to develop a DSE tool able to handle the exploration of possible tiling for algorithms with **non-affine data references**. The developed DSE tool, called MEXP (from Memory EXPloration), can be used as a front-end of the HLS in order to optimize its input C-code and thus its generated RTL model.

Figure 2.1 gives the whole design flow, which counts two steps: an early DSE run through MEXP and a second DSE run through a commercial HLS tool.

The MEXP DSE chooses the code transformations that are adapted to the non-affinity of the memory accesses and generates the synthesizable optimized C-models by customizing a generic template. The HLS DSE explores the space of possible corresponding RTL models for different values of clock rate $freq$ and of temporal constraints (detailed in section 3.3.1 of chapter 3).

The generated RTL benefits of both (the MEXP and the HLS) DSE and is optimized with respect to the area, the temporal performance and the power consumption.

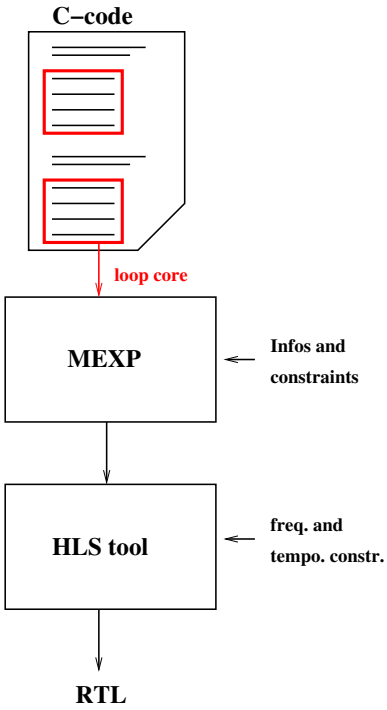


Figure 2.1: The whole flow

2.2 Introduction to the Design Space Exploration tool

MEXP is a Design Space Exploration tool that analyses a set of possible hardware realizations of a target application having non-affine array references.
MEXP performs a **dynamic profiling** of the application array references. It constructs two sets of possible partitioning for the input and output data spaces (called data tiling). The output data partitioning corresponds to a partitioning of the output computations (called loop tiling). MEXP combines the input and the output tilings and constructs the **space of possible solutions**, i.e. possible couples of I/O tilings. It **estimates** the behavior of the hardware realizations corresponding to the different couples of tilings. It **chooses** the candidates reducing the amount of **internal memory** and improving the **performance** of the system. Finally, It **customizes** an optimized and synthesizable C-template with the parameters corresponding to the chosen couple of input and output tiling.
The customized code can be used as an input of the HLS. And the measured performance of the RTL generated after the HLS can be compared to the MEXP estimation in order to endorse the MEXP analysis.

The MEXP analysis takes into account on one hand the application array references and on the other hand the estimation of the hardware performance. For this reason, the

chosen couples of input and output tilings are, at the same time, the most adapted to the non-affinity of the array references and have the best (estimated) hardware behavior with respect to the solutions of the analyzed set.

The hardware behavior evaluation takes into account the the target architectural model and is run with respect to two optimization criteria: the used internal memory and the temporal performance, i.e. the number of cycles needed by the hardware to execute the target application.

The target architectural model may compute more output tiles in parallel and for each output tile it performs two parallel macro-tasks: the pre-fetching of the next input and the computations of the current output.

To estimate the needed internal memory per solution, MEXP reckons both the internal memory needed to compute the current output tile and the internal memory needed to pre-fetch the next input data. This memory amount can be reduced by re-scheduling the output tile computations in order to compute in a line the output tiles sharing the most of input tiles.

To estimate the temporal performance of the resulting hardware, MEXP takes into account all the parallelism levels of the target architectural model: the parallelism between the tiles of computations and the parallelism between the pre-fetching and the computing.

A good choice of input and output tile sizes allows to mask the time needed to pre-fetch data with the time needed to perform the computations.

2.2.1 Advantages of the methodology

There are two advantages in using MEXP: on one hand, it adapts the input model of the HLS to the application behavior and, on the other hand, it provides an exploration at a higher level with respect to the HLS.

This allows to include, in the final designs, some essential memory aware optimizations, such as:

1. The tiling as a loop transformation. It is used to improve the parallelism level and implement the data pre-fetching.

2. The re-scheduling and mapping of the computations. They are used to improve the temporal performance and reduce the internal memory of the system.

3. Some hardware effective improvements (such as load balancing, data prefetching, etc.). They are used to reduce and tolerate the memory latency.

4. The usage of distributed memories. They are used to support a real parallelism between the output tile computations.

The tiling has been adapted to the algorithms with non-affine array references as follows: the input and output data spaces are separately partitioned with tiles that, due to the non-affinity of array references, can be different. They are related to each other trough a projection finding out which input tiles are needed to compute a given output tile.

One of the two input and output data tiling (usually the output data tiling for the image processing applications) corresponds to a loop tiling which divides also the computations into tiles. This allows to re-schedule the tiles of computations in order to minimize the amount of data to be prefetched.

The re-scheduling of the computation tiles can balance the prefetching and the computing which are run in parallel. As a consequence, it can reduce the execution time of the whole algorithm. It also reduces the number of external memory accesses and thus the power consumption of the generated hardware.

Some minor optimizations are included in the generic C-template customized by MEXP. As, for example, the usage of configurable shifts instead of dividers and a configurable fixed point arithmetic, which optimize the RTL generated after the HLS.

The next part of this section describes:

- The target architecture and
- The MEXP analysis flow

2.2.2 Overview of the target architecture

The MEXP analysis is based on a target architecture model that allows several level of parallelism.

The generic template of the target hardware is presented in figure 2.2 and is called Tile Processing Unit (TPU).

A TPU performs two parallel macro-tasks: the pre-fetching of the input data and a pipeline of three sub-tasks (REQ-FETCH-CALC) which is aimed to compute the output.

The pre-fetching, which consists in copying the input tiles from an external memory into some local buffers, is performed through some tables giving the mapping of the input tiles into the internal buffers (cf. MM in figure 2.2).

The pipeline (REQ-FETCH-CALC) is performed as follows: the module REQ reads which is the current output tile to be computed, from an internal register (OT in figure 2.2). For each datum in this output tile, it computes, through a user-defined Look Up Table (LUT), the non-affine coordinates of the needed input data. Finally, it requests the input data from the module FETCH.

FETCH analyzes the requests of REQ and infers from them the addresses of the input data in the internal buffers. Then it passes the needed input data to the module CALC, which computes the corresponding output.

According to the user specifications, it is possible to instantiate more parallel pipelines (REQ-FETCH-CALC) in order to compute more output tiles in parallel.

2.2.3 The MEXP flow

According to the work presented in [99], the MEXP DSE is performed through:

- a **validity filter**, which eliminates all the solutions that do not meet the user's specifications

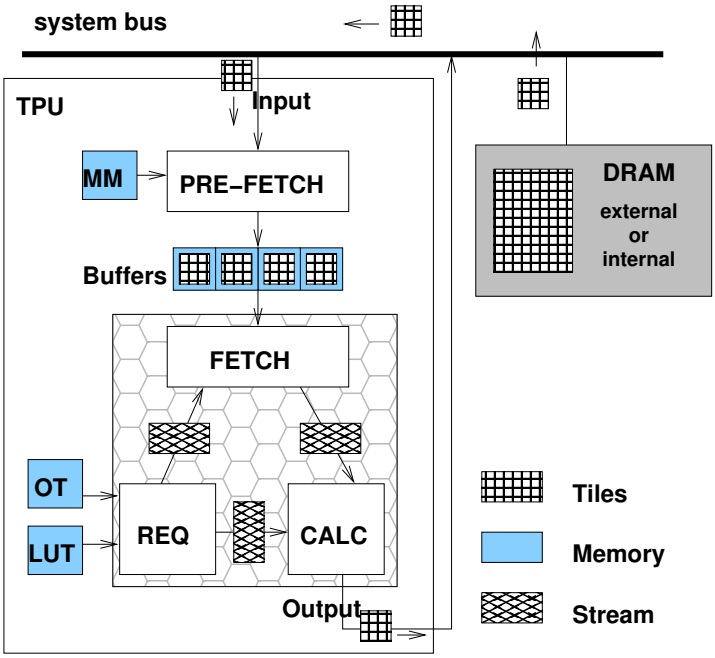


Figure 2.2: The TPU generic template

- and a **quality filter**, which classifies the solutions with respect to the two optimization criteria of internal memory used and temporal performance of the system.

The MEXP flow is detailed in figure 2.3 and is performed as follows: MEXP runs a **dynamic profiling** of the array references through the execution of an input C-function (REQ.c) specific to the application. This profiling only takes into account the dependences between the input and output data, but it does not consider the value of data nor the order of the memory accesses.

After the profiling and on the basis of user-defined volume of tiles, the tool selects the initial sets of input and output tile layouts with a rectangular shape; this step is called **tiling**.

For each possible couple of I/O tiling, MEXP performs the following steps:

1. **the Super-Tiling**. It projects the output tiling onto the input tiling and infers which input tiles are needed to calculate a given output tile. In this step, a first validity filter eliminates the solutions requiring more than a maximum of internal memory fixed by a corresponding user's constraint.

2. **the Scheduling**. It orders the computation of the output tiles by solving an associated Traveling Salesman Problem. A second validity filter eliminates all the solutions whose performance do not satisfy a user-defined constraint on the system performance.

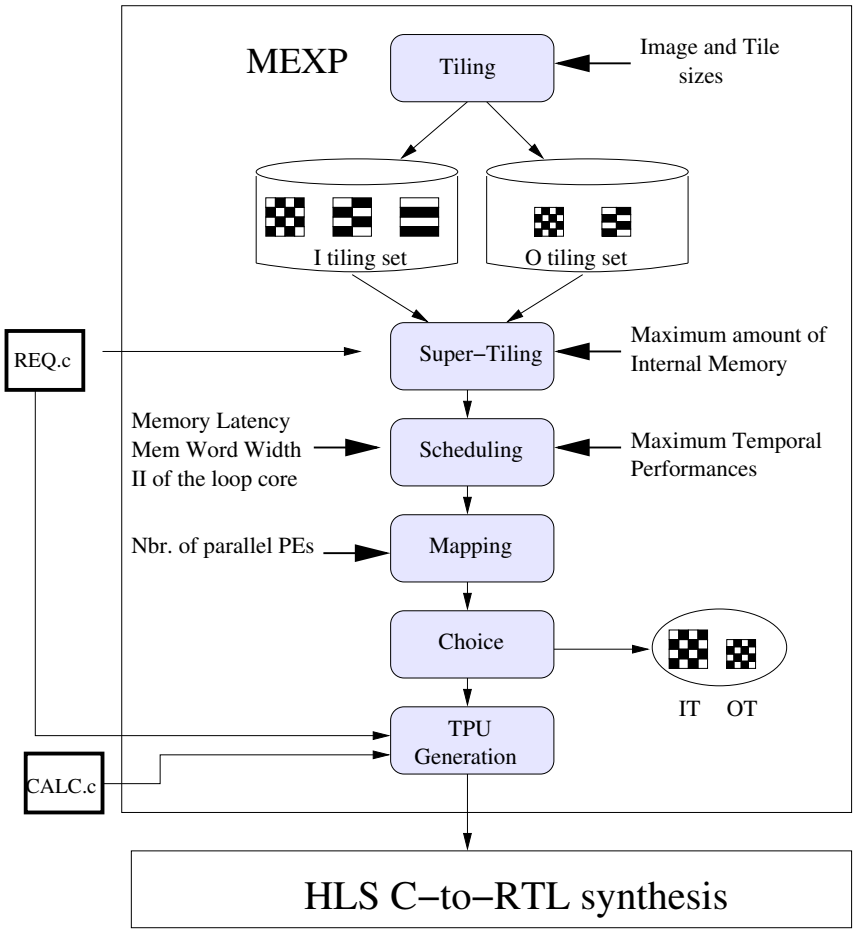


Figure 2.3: MEXP flow

3. **The Computation Mapping**. It allocates the calculation of different groups of output tiles to the available parallel pipelines REQ-FETCH-CALC. The number of pipelines is specified by a user constraint.

4. **The Memory Mapping**. It allocates the needed amount of internal buffers and computes the mapping between the needed input tiles and the available internal buffers. The Memory Mapping permits to directly access the internal buffer containing the needed input tile and ensures that the TPU executes without conflicts in memory accesses.

After that these four steps have been applied to all the solutions of the space, **the Quality filter** selects the Pareto solutions. For each solution MEXP evaluates the temporal performance and memory requirements of the corresponding TPU and chooses the solutions having the best characteristics. Finally, on the basis of a generic C-template, **the TPU generator** generates a synthesizable C-description of a TPU, which will be

the input of the HLS.

Each one of these steps will be detailed later in a dedicated chapter: the Supertiling is detailed in chapter 4, the Scheduling is detailed in chapter 5, the memory and computation mapping are described in chapter 6, the Quality filter is described in chapter 7 and the TPU generation is described in chapter 3.

2.2.4 MEXP input and output

The TPU template is adapted to realize all the algorithms which:

- can have non-affine array references,
- have static dependences[b] and
- are non-recursive.

The information needed to customize the TPU generic template contains two user-specified features:

- a user-defined function that describes the application behavior with respect to the input data requests (REQ) and to the output data computations (CALC);
- a user-defined Look Up Table (LUT), which gives the non-affine formula to compute the coordinates of the needed input data;

and three MEXP inferred features:

- the tables giving the mapping between the input tiles and the internal buffers;
- an optimized order to compute the output tiles and
- all the parameters characterizing the hardware (e.g. the internal memory size etc.).

2.2.4.1 The MEXP Input

The inputs of MEXP are the parameters used to tailor the DSE and the user-defined input C-functions, which will be analyzed and included in the output C-code.

2.2.4.1.1 The user-defined input C-functions The behavior of a particular TPU is defined by two input C-codes: one giving the algorithm behavior (with respect to the memory accesses and the output computation) and another defining a LUT which computes the non-affine addresses of the input data.

The first code defining the algorithm behavior contains (at least) two functions:

- REQ, which computes the input data coordinates by using the user-defined LUT and

[b] i.e. input coordinates do not depend on input data volume

- CALC, which computes the output.

These functions are intended to be synthesized with the commercial HLS tool, thus they have to be written according to the rules imposed by this tool (see chapter 1.2 for more details).

The second code can either directly define a LUT or define all the LUT parameters (e.g. the LUT size, the number of LUTs, etc..) and a function giving the non-affine formula of the array references to be realized with the LUT. In the second case the code is executed by MEXP to generate the static array LUT used in REQ.

2.2.4.1.2 The input parameters Besides the input C-functions, MEXP takes as input a set of parameters that tailor the DSE. These parameters are:

- A set of possible volumes for input and output tiles, i.e. the number of data that the user wants to be in a tile. From these volumes V different input and output tile layouts ($\frac{V}{2} \times 2$, $\frac{V}{4} \times 4$, ...) are inferred and then they are combined to each other in order to form the space of possible couples of I/O tiling.

- The maximum of Internal Memory (IM) that the circuit may have.

- The maximum number of clock cycles accepted to perform the algorithm for a given clock rate, used to run the HLS. The aim is to keep the clock rate as low as possible in order to reduce the power consumption. The MEXP analysis does not take into account the clock rate itself.

  If the two constraints (on IM and number of clock cycles) are too tight, no solutions are found.

- The external memory characteristics in terms of latency and word width

- The number of input data needed to compute an output datum, which will be also considered as the minimum number of clock cycles needed to perform an output datum computation.

- The desired parallelism level, i.e. the number of output tiles that the user wants to be calculated in parallel.

Thanks to this information, the MEXP exploration can use a cycle estimation of the system temporal performance which takes into account the delay due to the memory latency, to the memory bandwidth and to the processing elements performance (i.e. the number of input per output data and the parallelism level).

To estimate the area due to the memory sub-system, MEXP can take into account the memory word bandwidth, which influences the data layout in the memory and the memory area overhead due to the specified level of parallelism.

To conclude we can say that the early memory aware optimizations applied by MEXP to the C code are inferred by taking into account the run-time behavior of the algorithm and the estimated characteristics of the target hardware.

2.2.4.2 The MEXP output

To ensure that the TPU executes according to generic hardware template described in the paragraph 2.2.2, MEXP produces as output a set of synthesizable optimized C-models of TPU and gives for each one of them:

- The corresponding layouts of the I/O tilings
- A scheduling vector, which gives the order of the output tiles computations
- The Memory Mapping matrix, which gives the configuration of internal buffers during the execution of the whole algorithm and for all the parallel pipelines specified by the user.

2.3 Conclusion

In this chapter we have presented the basic principles of MEXP. MEXP allows to explore a set of candidate input/output couples of tilings for a data-dominated algorithm having non-affine array references. The exploration is performed with respect to two optimization criteria: the amount of internal memory used and the temporal performance of the generated system.

By using a generic template of the code to be generated and some user-specified functions, MEXP is able to generate an optimized C-code, which can be synthesized through a commercial HLS tool.

The optimizations applied to the code are inferred by taking into account the run-time behavior of the algorithm and the estimated characteristics of the target hardware.

Chapter 3

The Target architecture

THIS chapter introduces the target architecture used by MEXP. This architecture includes some hardware effective memory-aware optimizations, such as: the data prefetching and the balancing between the prefetching and the output computations. These memory-aware optimizations bring to an improvement of the performance of the whole system. The user can specify a certain level of parallelism which will be implemented with parallel pipelines computing more output tiles in parallel. The C-code corresponding to the target architecture is inferred from a generic template.

Chapter contents

This chapter goes through the following contents:

- An introduction
- A detailed description of the target architecture
- A presentation of the different levels of possible parallelism of the TPU
- The automatic generation of the output C-code

3.1 Introduction

The MEXP analysis ends with the generation of a synthesizable C code. This code is then transformed into a RTL model by a HLS tool. The target architecture of the RTL model is built with respect to a generic hardware template called Tile Processing Unit (TPU).

A TPU processes the output tiles according to the user-specified functionality and reads the input tiles from the external memory as calculated by the MEXP analysis. The functionality is specified through two user-defined C functions: a processing behavioral function, which gives the algorithm core, and a function giving the input/output data dependences.

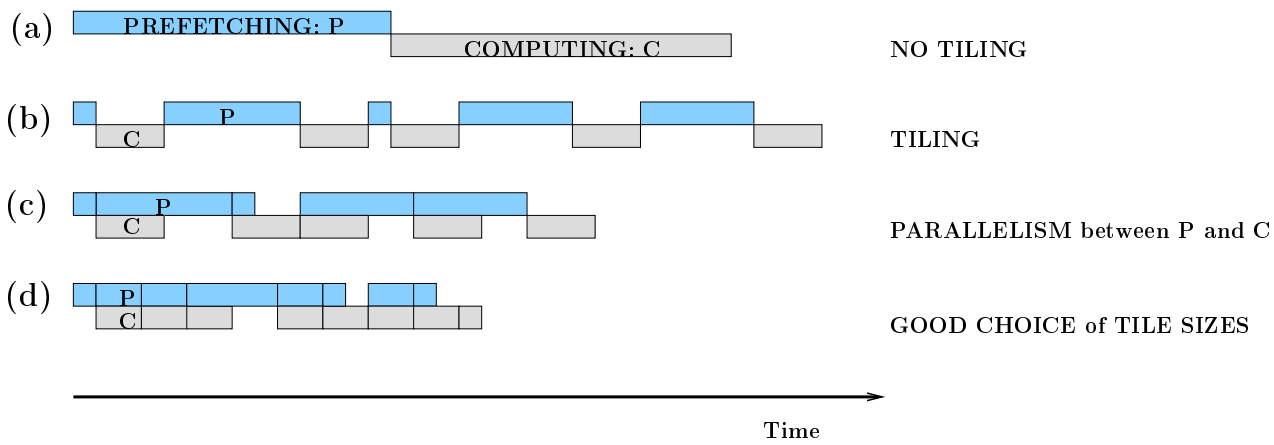


Figure 3.1: Effect of tiling, of pre-fetching and computing parallelism and of tile sizes on the temporal performance of the system.

The TPU model allows to perform in parallel the pre-fetching and the computations. The objective of a MEXP analysis is to find a couple of I/O tiling which balances the time to pre-fetch and the time to compute. This will improve the Temporal Performance of the target hardware.

Figure 3.1 explains the effects (on the TPU temporal performance) of the tiling and the parallelism between the pre-fetching and the computing:

- the tiling (figure 3.1(b)) partitions both the time to pre-fetch the input and the time to perform the computations. This allows two kinds of parallelism: the parallelism between output tile computations and the parallelism between the pre-fetching and the computing. It may occur that due to the non-affinity of the application array references, some input tiles are copied more than once into the internal memory in order to be used for the computation of two different (and non-successive) output tiles. This increases the total time needed to pre-fetch data and can be avoid with a good choice of I/O tiling and the re-scheduling of the output tiles computations.

- The TPU parallelism allows to execute the pre-fetching and the computations concurrently (figure 3.1(c)). This reduces the temporal performance of the final system (even if the pre-fetching time and thus the number of accesses to the external memory have been increased).

- The choice of an appropriate couple of input output tile sizes (figure 3.1(d)) can reduce the pre-fetching time. As a consequence it reduces the power consumption and improves the temporal performance of the system.

3.2 The meta-architecture overview

The TPU generic template is presented in figure 3.2. A TPU performs two parallel macro-tasks: it prefetches data from the external memory into the internal buffers and computes the output according to the user-specified functionality.

The pre-fetching is performed by the module PRE-FETCH. And the output computations are performed by the blocks REQ, FETCH and CALC which communicate through a handshake protocol and process all the output pixels in a pipeline.

The manipulated data are copied from an external memory into some internal buffers and each internal buffer contains an input tile.

As detailed in paragraph 3.3.3, the pipeline REQ-FETCH-CALC can be replicated N_p times in order to compute N_p parallel output tiles. N_p is a user-specified parameter.

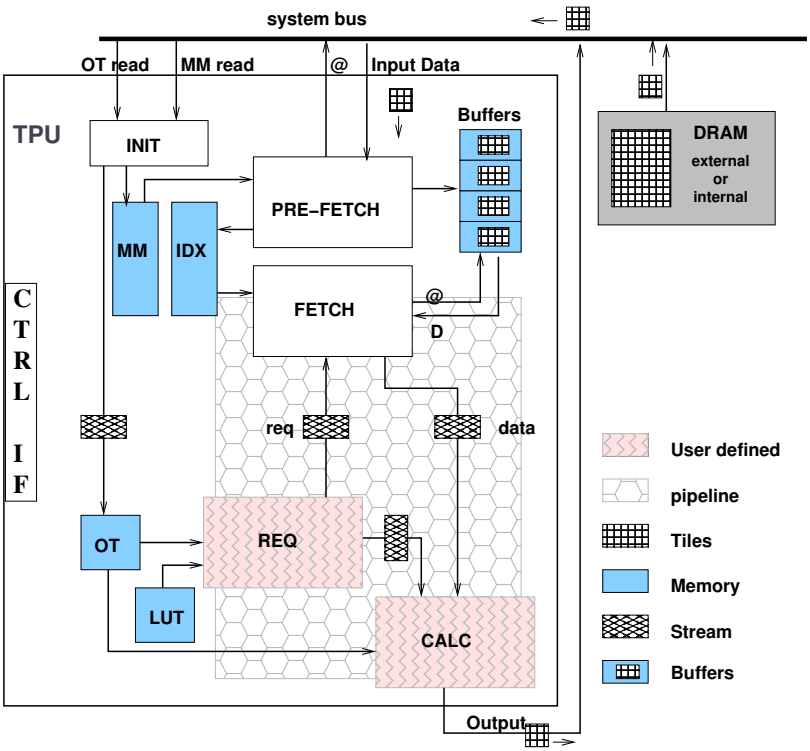


Figure 3.2: Generic TPU template

The TPU model is customized for a target application through the two user-specified functions (REQ and CALC) and through a MEXP generated table. This table gives the mapping between the input tiles and the internal buffers. It ensures that the buffers accesses are conflict free.

The access to the internal buffers is performed through the usage of two tables:

1. one used to pre-fetch the input tiles into the internal buffers (**MM** in the figure 3.2)

2. and the other used to directly read data from the internal buffers (**IDX** in the figure 3.2)

MM is generated by MEXP and IDX is computed on line by the TPU. The following paragraphs describe the two tables and their usage.

3.2.1 The MM table used to pre-fetch input from the external memory

MM is organized as presented in figure 3.3: each column index c corresponds to an output tile $OT(c)$ and each line index r corresponds to an internal buffer. An element $mm(c, r)$ of the table **MM** gives the number of one of the input tiles needed to compute $OT(c)$. This input tile has to be copied in the internal buffer r. A column of the **MM** table gives the set of all the input tiles needed to calculate a specific output tile. A line of **MM** gives the input tiles to be pre-fetched into a specific internal buffer during the execution of the whole algorithm.

MM is organized so that the input tiles common to more successive output tiles remain in the same memory buffer without being re-copied from the external memory. This reduces the power consumption and improves the temporal performance of the system.
On the other hand, **MM** ensures that the internal buffers written by PRE-FETCH are different from the internal buffers read by FETCH.

$$\underbrace{\begin{pmatrix} 1 & \dots & \dots & N \\ \vdots & & & \vdots \\ & \text{IT indices} & & \\ \vdots & & & \vdots \\ 0 & \dots & \dots & M \end{pmatrix}}_{\text{OT indices}} \left.\vphantom{\begin{pmatrix} 1 \\ \vdots \\ \\ \vdots \\ 0 \end{pmatrix}}\right\} \text{IB indices}$$

Figure 3.3: Table MM giving the memory mapping during the computation of all the output tiles

Figure 3.4 gives an example of a prefetching mechanism which uses the **MM** table of figure 3.4(b). The figure 3.4(c) presents the evolution of the internal memory all over the duration of the whole algorithm.

According to the pseudo-code in figure 3.4(a), if an input tile has to be copied into an internal buffer, i.e. if $MM[c][r] \neq 0$, its coordinates in the input data space are computed and then, the tile is copied from the external memory.

The time needed to copy each datum into the internal buffers depends only on the external memory speed and, if the external memory allows it, the datum can be copied in a single clock cycle.

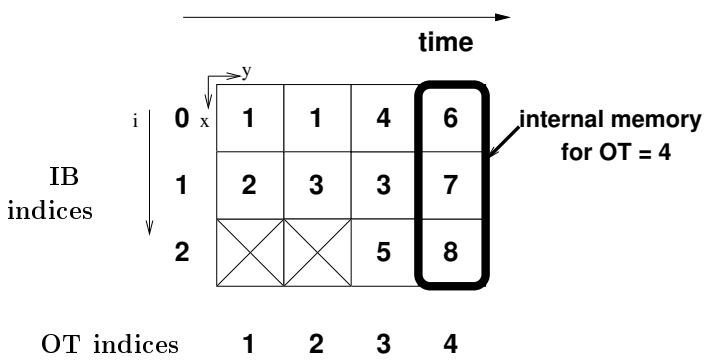
```
For a given output tile OT(c) to be computed
 For all the IB indices (r)
   Check if(MM[c][r]! = 0){
     Compute the IT coordinates = f(MM[c][r]);
     For all the data in the input tile
       IB[r][x][y] = read_from_ext_mem(x + IT coordinates );
     }
```

(a) pseudo-code of the pre-fetching mechanism

$$MM = \begin{pmatrix} 1 & 0 & 4 & 6 \\ 2 & 3 & 0 & 7 \\ 0 & 0 & 5 & 8 \end{pmatrix} \begin{matrix} 0 \\ 1 \\ 2 \end{matrix} \Bigg\} \text{IB indices}$$

$$\underbrace{1 \quad 2 \quad 3 \quad 4}_{\text{OT indices}}$$

(b) MM table



(c) Internal Memory configuration during the computation of all the output tiles

Figure 3.4: Example of a prefetching mechanism based on the MM table

To limit the area overhead due to the **MM** table we exploit the fact that a TPU call calculates a single (or a few) output tile(s) a time, thus **MM** is loaded a column a time.

3.2.2 The IDX table used to read data from the internal memory

The **MM** table associates an internal buffer to the storage of an input tile. Thus, given a buffer index, the index of the input tile stored in it, is read from **MM** in a cycle. The opposite is not true. In fact, given an input tile index, it could take M cycles to search in **MM** in which internal buffer the input tile is stored, where M is the total amount of internal buffers. Furthermore, as the input coordinates are non-affine, we have to search the corresponding internal buffer for all the requested input data. In fact it is not known, a priori, which input tile the data belong to. This could extremely degrade the temporal performance of the TPU.

In order to avoid this degradation, another table, called **IDX**, has been introduced.

$$\left.\begin{pmatrix} 1 \\ \vdots \\ \text{IM number} \\ \vdots \\ M \end{pmatrix}\right\} \text{IT number}$$

(a) generic IDX table

$$\left.\begin{matrix} \begin{pmatrix} ? \\ ? \\ ? \\ ? \\ ? \\ 0 \\ 1 \\ 2 \end{pmatrix} & \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} \end{matrix}\right\} \text{IT number}$$

(b) numerical example of IDX table (?=don't care)

Figure 3.5: Hash table IDX which gives the correspondence between the input tiles and the internal buffers

IDX associates the input tile indices to the internal buffer indices and is presented in the figure 3.5. For example, the **IDX** of figure 3.5(b) says that the input tile (IT) 6 is in the internal buffer (IB) 0, that the IT 7 is in the IB 1, etc ...

3.2.3 Example of a TPU time-line

To explain the usage of the two tables **MM** and **IDX**, we give an example of a typical TPU time-line as presented in figure 3.6.

The module REQ requests the input data needed to calculate an output tile $OT(c)$. Thanks to the prefetching and to the MEXP analysis and mapping, these data are already in the internal buffers.

The module FETCH examines the request of REQ, calculates which input tile contains the needed input data, accesses **IDX** to obtain the internal buffer where the input tile is stored and reads the needed input data from this buffer. Finally, it returns the read data to the module CALC, which computes the output.

Meanwhile the module INIT initializes the table **MM**, with the indices of the input tiles needed to compute the next output tile $OT(c+1)$. After INIT has finished, the

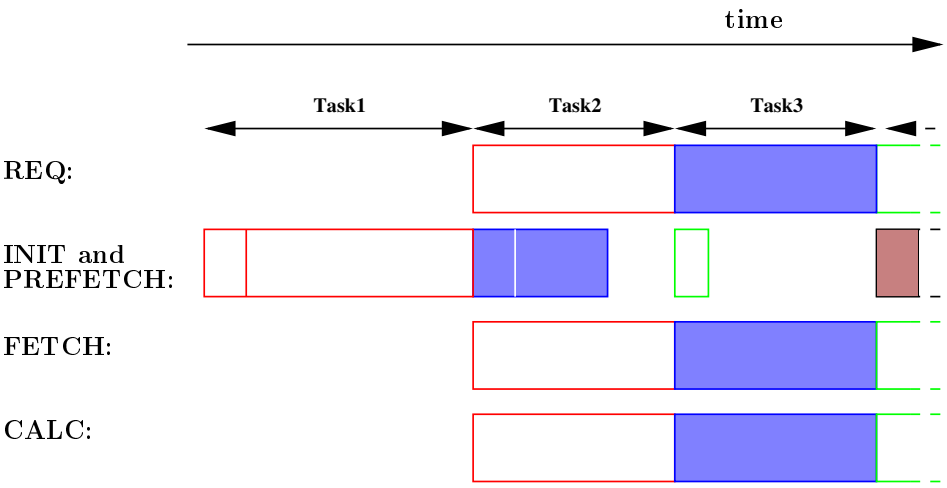


Figure 3.6: A typical TPU time-line

module PREFETCH reads **MM** to know which input tile to prefetch; copies the input tiles from the external and updates the content of the table **IDX**.

The tables **MM** and **IDX**, the internal buffers and external memory are supposed to be single port RAMs, i.e. they can only support a single write and a single read per cycle. For this reason INIT and PRE-FETCH are sequential. On the other hand, thanks to the tiling and mapping, INIT and PRE-FETCH are parallel to the pipeline REQ, FETCH and CALC.

Some hardware optimizations are possible to reduce the area overhead due to the **MM** table. They will be explained in chapter 6.

3.2.4 Possible conflicts on hardware resources

In the TPU architecture there are some possible conflicts on the hardware resources that could cause a malfunctioning of the system; we enumerate them and specify the corresponding found solutions.

- The modules FETCH and PREFETCH can have a conflict in accessing the internal buffers, for example if PREFETCH modifies a buffer read by FETCH. This conflict is solved by the MEXP analysis and mapping which ensures that internal buffers accesses are conflict free.

- Some conflicts can be caused by a simultaneous access to the system bus. To ease this problem we will consider that a single TPU a time uses the system bus and that it has a Direct External Memory Access. In a configuration with more TPUs connected on the system bus, an arbiter will manage the bus access according to a set of pre-calculated priorities.

  On the other hand, the “input data read" and the “output data write" of a single TPU can cause a conflict on the system bus access, thus two separate slots of time

are reserved to the reading and writing. Furthermore the following rules have to be respected: 1) the reading of an input tile is an atomic operation; 2) to avoid a deadlock, the calculated output data are temporary stored in a local memory and copied into the external memory before starting the calculation of a new output tile.

3.3 The HLS model of the TPU

3.3.1 Re-calls of the HLS principles

The HLS, which is detailed in chapter 1.2, takes as input a loop-based C-code and synthesizes it as a Pipeline Processing Array. That is a processor transforming more sets of input data (e.g. different input images) in a pipeline. Each loop nest of the code is synthesized as a Processing Element (PE) which contains a control path and a data path (DP). The control path iteratively executes operations on the DP which corresponds to the loop-core of the input C-code.

Two PAs (or loop nests), one producing and the other consuming the data, communicate through streams. This allows the PEs to execute in parallel (see chapter 1.2 for more details).

The HLS offer four possible levels of parallelism to realize the hardware:

1. a parallelism between the operations of the same loop core (called the inter-operations parallelism),
2. a parallelism between the iterations of the same loop (called inter-iterations or intra-loop parallelism)
3. a parallelism between different loop nests of the code (called inter-loops parallelism) and
4. a parallelism between different tasks of the PPA (called inter-tasks parallelism)

The inter-loop parallelism is ensured by the usage of the streams.

While the other kinds of parallelism are automatically inferred by the HLS tool under two user constraints: the Initiation Interval (II) and the Minimum Inter Task Interval (MITI).

The II is specific to a loop-nest and represents the number of clock cycles passing between the start of two following iterations of the loop-nest when they are processed on the target architecture. The II forces the HLS tool to instantiate more or less hardware resources in the processing element and thus allows the inter-operations and the inter-iterations parallelism.

The MITI represents the number of clock cycles passing between the start of two successive PPA tasks.

A task of the PPA is the execution of the whole algorithm realized by the PPA on a set of input (for example an input image). The MITI allows the pipeline of more tasks on a PPA (for example to process a set of different frames in a video).

By variating the values of the II and the MITI it is possible to run a DSE to find a trade-off between the amount of resources instantiated and the temporal performances of the system.

3.3.2 The TPU C-model

According to the HLS roles, MEXP is able to generate a TPU C-model written as a series of five perfectly nested loops[a]. The TPU code is given in figure 3.7.

This code contains 5 loop-nests corresponding to the TPU blocks (INIT, REQ, PRE-FETCH, FETCH and CALC).

The bold elements of the code can be parameters to be customized by a MEXP analysis (OT, IT, MM, ...), or user specified functions that characterizes the application (REQ and CALC), or streams.

In order to realize the pipeline REQ-FETCH-CALC, the TPU uses the streams (ADD_REQ and FROM_FETCH in the code).

The initialization and the prefetching are completely independent from the pipeline, thus they can be executed in parallel with it.

Besides the C-code describing the TPU, MEXP is able to generate the HLS constraints (MITI and II), which according to the HLS-vendor rules are the tightest possible.

The HLS inter-tasks parallelism is not possible for the TPU, in fact two successive tasks of the TPU are dependent on each other because of the pre-fetching. For these reasons the MITI is set to 1.

However MEXP offers the possibility to have a parallelism between the output tiles computations, i.e. an inter-tiles parallelism.

The user specifies the number N_p of output tiles to be computed in parallel and MEXP analyses the dependences, pre-calculates the scheduling and mapping of the computations and generates the corresponding parallelizable C-code, with N_p parallel pipelines REQ-FETCH-CALC.

The IIs of different loop-nests of the TPU are automatically deduced as follows:

- REQ and FETCH have an II=`N_DATA`, where `N_DATA` is the number of input data needed to compute an output datum and thus the number of accesses to the streams transmitting the corresponding information (data address and value).
- CALC has to read `N_DATA` and to write sequentially into the external memory the N_p output data of the N_p parallel REQ-FETCH-CALC pipelines, thus it has a II=max(`N_DATA`, N_p).
- PRE-FETCH has an II=1, because it reads 1 datum per cycles from the external memory.

This configuration allows the maximal level of inter-operations and inter-iterations parallelism.

The user-defined functions REQ and CALC should use some MEXP defined macros

[a] i.e. all the operations are performed in the loop core

and global variables in order to allow the TPU customization and the communication between the user-defined and the MEXP generated code.

Following are examples of these macros and variables:

- OT_order[] is the global table containing the order to the output tiles to be computed. It has a size equal to N_p.
- OT_i and IT_i are the width of the ith side of the output and input tiles respectively, the labels i are assumed to be as in figure 3.8.
- N_OT_i= $log_2(OT\_i)$ and N_IT_i= $log_2(IT\_i)$ these values are given in order to substitute divisions with shifts.
- out_i and in_i are the number of output and input tiles on the side i, with N_out_i= $log_2(out\_i)$ and N_in_i= $log_2(in\_i)$.
- O_WIDTH_i and I_WIDTH_i are the width of size i for the output and input image respectively
- LUT, LUT_WIDTH and LUT_STEP defines the LUT itself (which has been previously filled thanks to a user-defined function), the LUT width and the LUT step, with LUT_STEP_BITWIDTH=$log_2(LUT\_STEP)$
- PREC is the precision of the used fixed point arithmetic (in our experiments PREC=8 bits).

3.3.3 The user-specified parallelism level inferred by MEXP: parallelism between output tiles computations

MEXP analysis can be run for a user-specified inter-tiles parallelism N_p, which corresponds to the number of output tiles to be calculated in parallel and, thus, requires an additional amount of hardware resources.

In the TPU model presented in figure 3.2 of the paragraph 3.2, the output tiles are calculated sequentially on the same hardware. The corresponding C-code has essentially five loop nests, one for each module: INIT, REQ, CALC, PREFETCH and FETCH. And each loop nest is synthesized as a Processing Element with a single data path.

In a parallel TPU, the data paths of the different loop nests are replicated N_p times.

In this case all the data paths DP_i belonging to the different processing elements REQ, CALC and FETCH communicate to each other through a reserved set of streams, they access to a dedicated set of internal buffers and form an independent pipeline.

Figure 3.12 gives the code of a TPU able to compute two output tiles in parallel, it shows that the data paths DP_0 of the modules REQ, FETCH and CALC communicate to each other through the streams ADD_REQ_0 and $FROM\_FETCH\_0$, access to the buffers IM_0 and form the pipeline p_0.

Only the DPs in the pipeline containing the instructions REQ, FETCH and CALC are duplicated. While the prefetching of input tiles is made sequentially in order to respect the bus bandwidth.

The advantage to duplicate only the DPs is that the hardware dedicated to the control of the iterations pipeline is shared thus not all the TPU is duplicated. Furthermore the internal memory amount is not duplicated but adapted to the output tiles that the pipeline has to compute.

Figure 3.9 gives an example of several possible implementations to compute 8 output tiles. This figure shows that the more parallel pipelines are used the more the time to execute the whole algorithm is reduced.

Nonetheless, the scalability of the architecture is limited by the system bus bandwidth and the external memory latency. In fact, as the external memory is a single port memory, the prefetching of input data has to be performed sequentially for all the pipelines in the TPU. Thus the more the TPU contains parallel pipelines the more the time to prefetch data becomes preponderant with respect to the time to compute the parallel output tiles. On the other hand, the time needed to prefetch the input tiles depends also on the external memory latency whose increase can reduce the speedup due to the parallelism. Figure 3.10 gives an example of a time-line for a TPU containing two parallel pipelines. In this example, the pre-fetching of the input tiles needed to compute the output tiles OT_5 and OT_6 is preponderant with respect the time needed to compute in parallel the output tiles OT_3 and OT_4 (task 3).

For the algorithms tested the efficacy of the parallelism (i.e. the speedup of the execution time with respect to the the memory overhead due to the parallelism) is maximum for $N_p < 4$ and then decreases.

3.4 Automatic generation

This paragraph describes the synthesizable C-model of a TPU generated by MEXP.

Figure 3.11 shows the files of the model. The project counts:

- a *main.c*, which contains the TPU calls and will be turn into a benchmark of the RTL by the HLS tool. Given d the total number of output tiles and N_p the number of output tiles to be calculated in parallel, the *main.c* calls the TPU function $\frac{d}{N_p}$ times and during each call the TPU computes N_p output tiles.

- a *TPU_TOP.c*, which contains the TPU description as a set of loop nests. The *TPU_TOP.c* calls the functions TTC, REQ and CALC.

- a *TTC.c*, which contains the description of the INIT and PRE-FETCH macro-blocks and the definitions of other functions which describe the external memory accesses performed through the system bus. TTC stands for Tiles Transfer Control.

- a *hardware_config.h* file which contains the hardware configuration, the MEXP generated tables and the definition of the I/O and internal streams.

- a *HLS_ config.tcl* file which contains all the parameters to configure the High-level Synthesis.
- a user-defined file containing the REQ and CALC description. REQ and CALC are called N_p times in the *TPU_ TOP.c* and their calls correspond to as much of parallel pipelines instantiations needed to achieve the user-specified inter-tiles parallelism.
- a file containing the user-defined LUT which implement the non-affine law of the array references.

By using generic templates of the C-codes, MEXP automatically generates the main.c, the *TPU_ TOP.c*, the *TTC.c*, the *hardware_ config.h* and the *HLS_ config.tcl*; while the user has to specify the LUT and the functions REQ and CALC, according to the rules imposed by the HLS tool.

3.5 Conclusion

In this chapter we presented the generic target architecture customized by MEXP to generate its output C-code. The architecture is called Tile Processing Unit (TPU) and performs two parallel macro-tasks: the pre-fetching and the computing. The computing is performed through a pipeline of operations and the prefetching consists in copying the input tiles from the external memory into the internal buffers.

For a given solution, the mapping between the input tile buffers and the available internal buffers is pre-calculated by MEXP and used by the customized TPU to perform the prefetching.

The TPU has several level of parallelism, that we can distinguish in: the parallelism ensured by the commercial HLS tool under MEXP pre-calculated constraints and the inter-output tiles parallelism inferred by MEXP.

The inter-tile parallelism requires the instantiation of more parallel pipelines performing the computing. The pre-fetching is common to all the different pipelines, in order to respect the bus bandwidth, and this can limit the possibility of further parallelizations.

```
void TPU_TOP(int l){

INIT: INIT(l);

REQ: for(i=0;i<OT_0; i++){
   for(j=0;j<OT_1; j++){
      if(l>0){
         int DX[l];
         int ADD_REQ[N_DATA];
         REQ(i , j, 0, DX, ADD_REQ);
         HLS_stream_output_DX(DX[0]);
#pragma unroll z
         for(z=0;z<N_DATA;z++)
            HLS_stream_output_ADD_REQ(ADD_REQ[z]);
      }
   }
}


PRE-FETCH:   for(k=0;k<nbr_of_buffers;k++){
      for(i=0;i<IT_0;i++){
         for(j=0;j<IT_1;j++){
            (IT_x0,IT_x1)=func(MM[k]);   #IT origin
            IM[k][i][j]=read_from_ext_mem(i+IT_x0, j+IT_x1);
         }
      }
   }

FETCH:for(i=0;i<OT_0;i++){
   for(j=0;j<0T_1;j++){
      if(l>0){
         int x_in[N_DATA]:
         for(ii <N_DATA)
            x_in[ii]=stream_input_ADD_REQ();
         data_out=FETCH(d,x_in,IM,MM_idx);
         stream_output_FROM_FETCH();
      }
   }
}
CALC: for(i=0;i<OT_0; i++){
   for(j=0;j<OT_1; j++){
      if(l>0){
         int dx;
         int DATA[N_DATA];
         int out_DATA[N_DATA], out_ADD[N_DATA];
         dx = HLS_stream_input_DX();
#pragma unroll z
         for(z=0;z<N_DATA;z++)
            DATA[z]=HLS_stream_input_FROM_FETCH();
         CALC(i , j, 0, dx, DATA, out_DATA, out_ADD);
      }
   }
}
}
```

Figure 3.7: C-code of a TPU, this code can be customize for a given application and a particular solution chosen by MEXP.

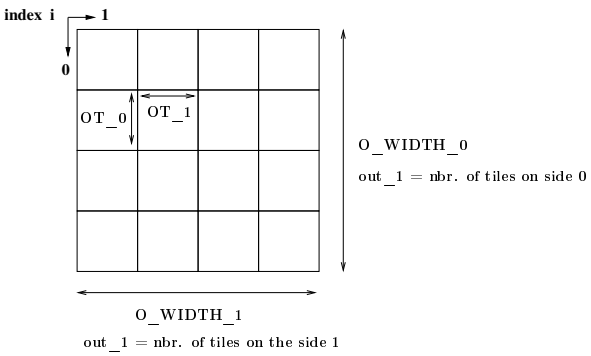


Figure 3.8: Example of some MEXP defined macros for an output image

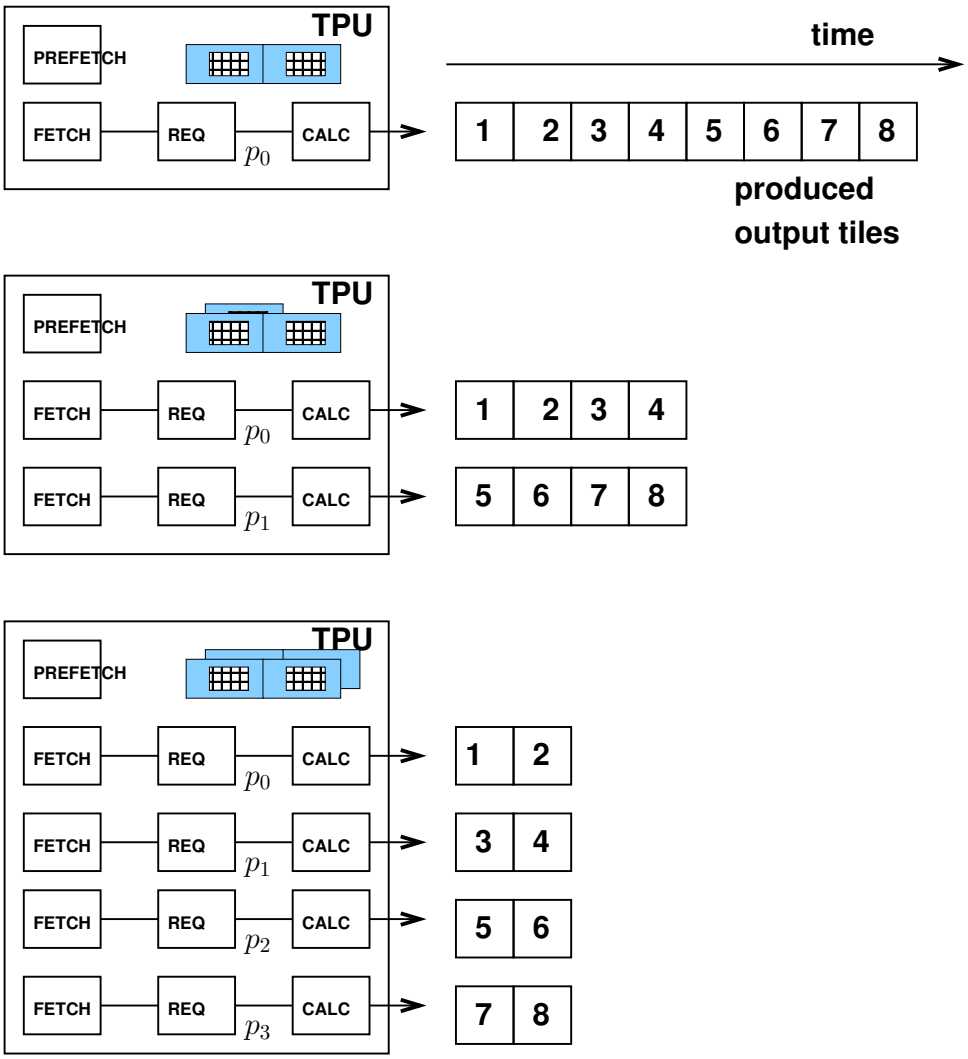


Figure 3.9: Example of a several possible implementations to compute 8 output tiles: a TPU with a single pipeline, a TPU with 2 parallel pipelines and a TPU with 4 parallel pipelines

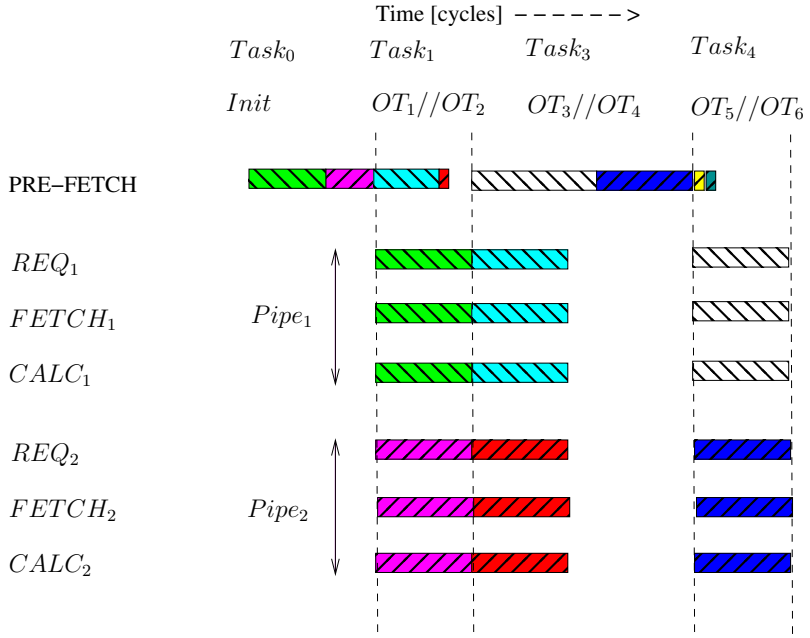


Figure 3.10: Example of a time-line for a TPU containing two parallel pipelines

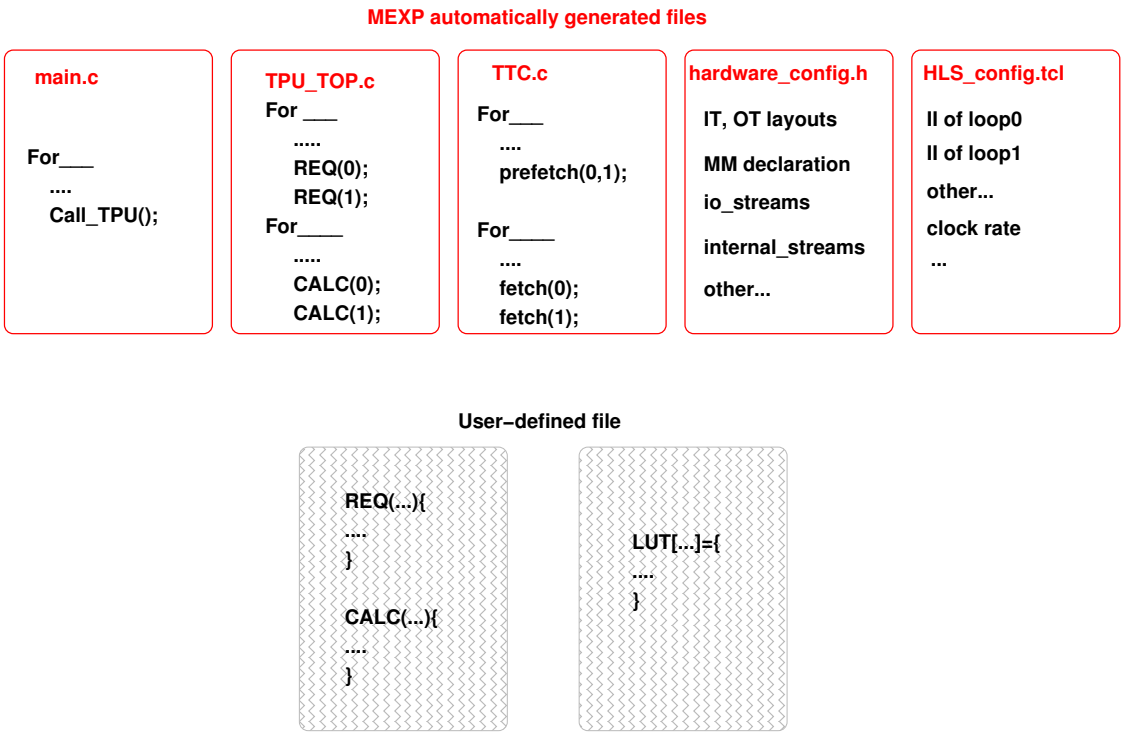


Figure 3.11: Set of files, contained in a MEXP generated project describing a TPU

```
void TPU_TOP(int l){

INIT: INIT(l);

REQ: for(i=0;i<OT_0; i++){
   for(j=0;j<OT_1; j++){
      if(l>0){
         int DX_0[1];
         int ADD_REQ_0[N_DATA];
         REQ(i , j, 0, DX_0, ADD_REQ_0);
         HLS_stream_output_DX_0(DX_0[0]);                                  DP0
#pragma unroll z
         for(z=0;z<N_DATA;z++)
            HLS_stream_output_ADD_REQ_0(ADD_REQ_0[z]);

         int DX_1[1];
         int ADD_REQ_1[N_DATA];
         REQ(i , j, 1, DX_1, ADD_REQ_1);
         HLS_stream_output_DX_1(DX_1[0]);                                  DP1
#pragma unroll z
         for(z=0;z<N_DATA;z++)
            HLS_stream_output_ADD_REQ_1(ADD_REQ_1[z]);
      }
   }
}

PRE-FETCH:   for(k=0;k<nbr_of_buffers;k++){
      for(i=0;i<IT_0;i++){
         for(j=0;j<IT_1;j++){
            if(k<nbr_of_buffers_0){
               (IT_x0,IT_x1)=func(MM[k]);   #IT origin
               IM_0[k][i][j]=read_from_ext_mem(i+IT_x0, j+IT_x1);
            }
            if(k>nbr_of_buffers_0 && k<nbr_of_buffers_1 ){
               (IT_x0,IT_x1)=func(MM[k]);   #IT origin
               IM_1[k][i][j]=read_from_ext_mem(i+IT_x0, j+IT_x1);
            }
         }
      }
   }

FETCH:for(i=0;i<OT_0;i++){
   for(j=0;j<0T_1;j++){
      if(l>0){
         int x_in_0[N_DATA]:
         for(ii <N_DATA)
            x_in_0[ii]=stream_input_ADD_REQ_0();                           DP0
         data_out=FETCH(0,x_in_0,IM_0,MM_idx_0);
         stream_output_FROM_FETCH_0();

         int x_in_1[N_DATA]:
         for(ii <N_DATA)
            x_in_1[ii]=stream_input_ADD_REQ_1();                           DP1
         data_out=FETCH(1,x_in_1,IM_1,MM_idx_1);
         stream_output_FROM_FETCH_1();
      }
   }
}
CALC: for(i=0;i<OT_0; i++){
   for(j=0;j<OT_1; j++){
      if(l>0){
         int dx;
         int DATA_0[N_DATA];
         int out_DATA_0[N_DATA], out_ADD_0[N_DATA];
         dx = HLS_stream_input_DX_0();
#pragma unroll z                                                           DP0
         for(z=0;z<N_DATA;z++)
            DATA_0[z]=HLS_stream_input_FROM_FETCH_0();
         CALC(i , j, 0, dx, DATA_0, out_DATA_0, out_ADD_0);

         int DATA_1[N_DATA];
         int out_DATA_1[N_DATA], out_ADD_1[N_DATA];
         dx = HLS_stream_input_DX_1();
#pragma unroll z                                                           DP1
         for(z=0;z<N_DATA;z++)
            DATA_1[z]=HLS_stream_input_FROM_FETCH_1();
         CALC(i , j, 1, dx, DATA_1, out_DATA_1, out_ADD_1);
      }
   }
}
```

Figure 3.12: C-code of a TPU, this code can be customize for a given application and a particular solution chosen by MEXP.

Chapter 4

Super-tiling

THIS chapter presents the first step of the MEXP methodology: the Super-Tiling. Given an iterative application represented through a loop-nest, the Super-Tiling allows to tile the iteration space even if the loop nest is non-affine. The Super-Tiling consists in separately partitioning the I/O spaces of the assignment statements in the loop core. The I/O partitioning, called tiling themselves, are related to each other through a projection whose law coincides with the data access law. The I/O dependences are determined through a dynamic profiling.

Chapter contents

This chapter uses the concepts of dependence vector, affine loop nest and loop tiling (defined in section 1.3.2 of chapter 1.3) and it goes though the following contents:

- Discussion about the problem of tiling non-affine loop nest
- Definition of the target applications
- Definition of the I/O dependence list
- Presentation of the Super-Tiling flow and of the Exploration of a set of possible Super-Tilings
- Presentation of the algorithms used to run the Super-Tiling
- Results supporting the Super-Tiling method

4.1 Discussion about the problem of tiling non-affine loop nest

MEXP computes and explores a set of possible optimized C-models of a given iterative data-dominated algorithm. The set is generated by transforming an input C-code through an appropriate loop tiling.

In the literature (see chapter 1.3), the tiling is a well defined loop transformation aimed to partition the iterations space of a loop nest and, thus, it is largely used in parallel compiler or in High-Level Synthesis tools to completely or partially parallelize groups of iterations. It exploits data locality and can be combined to other loop transformations in order to re-organize the array references.

Works in literature mainly focus on algorithms with affine array references while our aim is to define a tiling for algorithms with non affine array references.

We give 3 reasons that may prevent the tiling of a non-affine loop nest:

1. **The non-affinity of array references causes a non-regular partition of the input address space**

   In a affine-loop nest all the array references are affine to each other and to the loop indices. The first consequence is that the I/O address spaces and the iteration space are equal by translation, which directly derives from the definition of affinity. In this case, a loop tiling causes a regular partition of the I/O address spaces and of the iteration space.

   When the array references are not affine, the tiling of iterations space produces an irregular data partitioning.

   As an example, let consider the logarithmic sampling. The corresponding code in figure 4.1 shows that the output space address (with coordinates i and j) coincides with the iterations space but, due to the non-affinity of the array references (i.e. coordinates ii and jj in the code) it is different from the input address space.

```
LOG_SAMPLING(k){
...
r0=M/k;
r_LIM=M/2;
...
 for(i=0;i<N;i++){
    for(j=0;j<M;j++){
...
            r1=sqrt(i^2+j^2);
            r=r0/(r_LIM - r1);
            ii = r * i;
            jj = r * j;
            A1[i][j]= A0[ii][jj];
        }
    }
...
}
```

Figure 4.1: Pseudo code of a Logarithmic Sampling

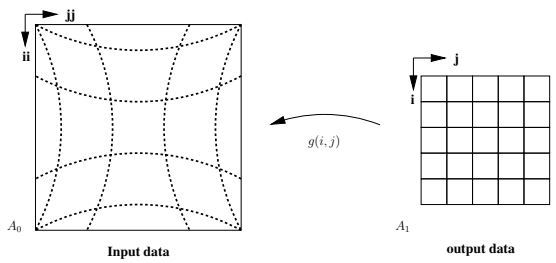


Figure 4.2: Tiling applied to a non-affine indexed application

   Applying the loop tiling on this example will cause a regular partitioning of output data and a non regular partitioning of input data (figure 4.2).

   This is an issue in the synthesis of ASIC, because the communication between the hardware modules depends on the number and kind of data to transfer and the easiest and cheapest way to realize it is to transfer blocks of a constant number of data.

2. **The tiling legality is verified with respect to the data dependences analysis, which is only possible with affine array references**

The tiling is legal only if it respects data dependences.

The HLS tools or the parallel compiler extract data dependences by applying linear programming solving methods (see chapter 1.3), thus, it is mandatory that the array references of the algorithm are affine.

The first immediate consequence is that for code with non-affine array references neither the tiling legality nor the data dependences can be verified through a static analysis of the code.

3. **The tiling size and shape are determined by solving linear programming problem with respect to the constraints imposed by the data dependences.**

   This method is obviously not adapted to the non-affine array references.

   To define an optimal tiling it is necessary to solve two problems: choosing an optimal tile shape and finding an optimal tile size. In an affine-loop nest these problems can be related to the dependence analysis because the dependences between data or iterations can be expressed by a "distance vector" (see chapter **??** for more details) containing two kinds of information:

   - the dependences between two different iterations and their temporal distance, in term of number of iterations;
   - the I/O data dependence and their spatial distance.

   An optimal loop tiling can be deduced from this information and produces at the same time an optimal partition of the iteration space and of the corresponding I/O data spaces. Furthermore it produces Input and Output data partitions of the same size and shape.

   In an affine loop nest the tile edges are computed as parallel to the dependence vectors in order to minimize the communications between the processing elements that compute the tiles.

   In a non-affine loop nest the vector, giving the dependences between I/O data, changes of sizes and directions over the iteration space thus they can not be chosen as a reference for the tile edges.

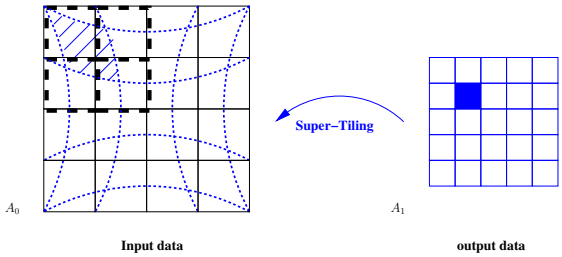


Figure 4.3: Super-Tiling

Our work overcomes these problems by replacing the "distance vector" with an I/O dependence list computed by performing a dynamic profiling of the application array references. A different data partitioning is separately applied to the input and output data sets and, thanks to the information contained in the I/O dependence list, it is possible to project the output tiling onto the input space and to find out which input tiles are needed to compute a given output tile (Figure 4.3). This technique is called Super-Tiling because it superposes the output to the input tiling through a projection. The partitioning of output data causes a tiling over the loop nest calculating the output. *We will call tiling either the loop tiling or the data partitioning.*
The input data tiling allows the HW model to transfer data per blocks of constant size (the tiles) meanwhile the variable number of input tiles to transfer per each output tile, adapts the transfer to the non-affinity of the array references.

4.2 Presentation of the Super-tiling flow and of the Exploration of a set of possible Super-tiling

As shown in figure 4.4, the Super-Tiling is applied in three steps.

First a dynamic profiling of the program establishes the I/O dependence list, i.e. which input data are needed to calculate a given output datum. Then two sets of possible input and output tiling are computed. Finally, for each couple of I/O tilings, a tiling projection calculates the dependence between the input and output tiles and give them through a Σ matrix.

The Tiling is computed by taking as input the size of the space to be tiled and a set of scalars. Each scalar represents the number of data that the user wants to be in a tile and is called **tile volume**. The tiling calculates a set of possible tile layouts corresponding to the given tile volumes.

The projection constructs the space of all possible combinations of I/O tilings and each possible combination is also referred to as a possible couple of I/O tilings or as a possible Super-Tiling.

In this section, we, first, give the generic form that a target application should have, then we give the algorithms used to run the three steps of the Super-Tiling: dynamic profiling, tiling and projection. For each step, the following information will be given:

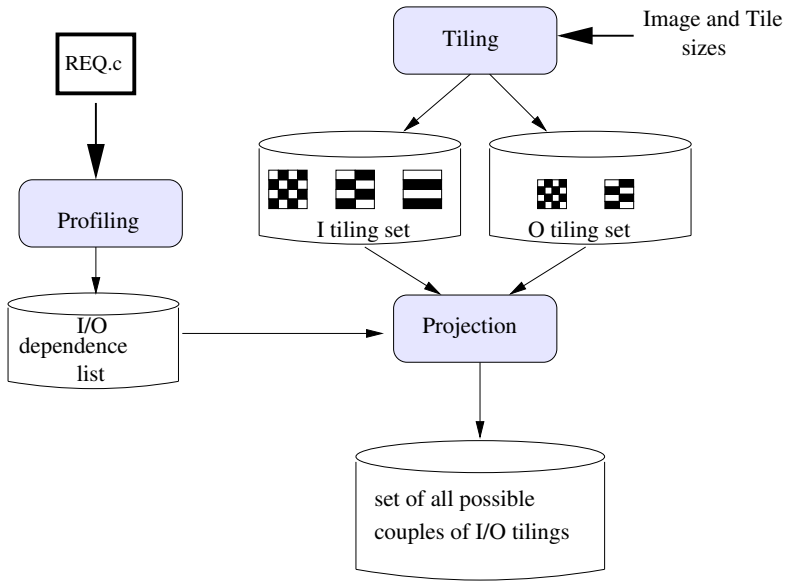


Figure 4.4: The Super-tiling Flow

1)the mathematical framework and formula 2) the pseudo-code used to implement the algorithm and 3) an example.

4.2.1 The target application

A typical application that can be tiled with the Super-Tiling is an iterative data-dominated algorithm which have static dependences (i.e. the dependences do not depend on data value) and is non-recursive (i.e. the output data depend only on the input data).

The iterative data-dominated algorithm is described by a loop nest as presented in figure 4.5.

$$\begin{aligned} &\mathbf{L}\text{: } \mathbf{for}\ 1 \leq p \leq \overline{N} \\ &\qquad \mathbf{S}\text{: } I_t(p) = f(W_{t-1}(p)) \\ &\quad \mathbf{end} \end{aligned}$$

Figure 4.5: A generic iterative data-dominated algorithm

A vector p parses a n-dimensional space $\overline{N} = (N_1, \ldots, N_n)$, where p can be both an index of the iteration space or the coordinates of an output datum. As a consequence $\overline{N}$ is either the iteration space or the space of the output coordinates.

The statement **S** in the code, computes an output datum $I_t(p)$ as a function of a window (or set) of input data $W_{t-1}(p)$, computed as follows:

$$W(p) = \bigcup_{i=1}^{m} \{I_{t-1}(g_i(p))\}$$

with $W_{t-1}(p) \subseteq I_{t-1}$ (see section 1.1.1 of chapter 1.1 for more details).

The window of needed input data contains all the input data $I_{t-1}(g_i(p))$ which coordinates are computed as a non-affine function $g_i(p)$ of the corresponding output data coordinates p. The main differences between this definition of a loop-nest and the one that is usually employed in previous related works is that:

1. no restrictions are given on the mathematical form of the addresses
2. the self-dependences are not allowed (i.e. algorithms are not recursive).

4.2.2 Profiling

The profiling, which code is given in figure 4.6, is aimed to compute the I/O dependences list of the loop nest; it takes as input the user-defined function `REQ`, which gives the non-affine formula of the array references, and recursively executes it in order to compute the coordinates of each datum in the output space.

```
— data structure—
dim: space (and tile) dimension
d: index parsing the space (and tile) dimensions
vo[]: output data to compute
vi: pointer to an element of type stk, i.e. a list of needed input data
ws[]: output space bounds
curr_datum: pointer to an element of type stack, i.e. a list of I/O dependences
————————————

extern stack *curr_datum; void inputmap(int dim, int d, int vo[], int ws[]){
    if(d>0){
        for(p=0;p<ws[d];p++){
            vo[d]=p;
            inputmap(dim, d-1, vo, ws);
        }
    }else{
        for(l=0;l<ws[d]; l++){
            vo[d]=l;
            vi=REQ(vo, &q);
            curr_datum=add_satck(curr_datum,vo,vi, q, dim);
        }
        d=dim;
    }
}
```

Figure 4.6: Code of the profiling algorithm

The profiling can be run for any algorithm which associates an output datum to one or more input data; and the I/O data sets can be n-dimensional sets with $n \in \mathbb{N}^*$.

The following paragraphs describe the method used to construct the I/O dependences list and the template that a user-defined `REQ` function should have.

4.2.3 Non-uniform loop nest and I/O dependence list

Given a target application as described in paragraph 4.2.1, we express the I/O data dependences of the application by listing them. We first construct for each output datum the sub-list of input needed to compute it:

$$D_{I/O}(p) = \{g_i(p), I_{t-1}(g_i(p)) \in W_{t-1}(p), \forall i \in [1, m]\} \tag{4.1}$$

Then we enlarge the list to all the output pixels, as follows:

$$D_{I/O} = \{D_{I/O}(p), \forall p \in \overline{N}\} \tag{4.2}$$

As an example, consider the loop $\mathbf{L_2}$:

$\mathbf{L_2}$: for $1 \leq p_1 \leq n_1$

for $1 \leq p_2 \leq n_2$

$\mathbf{S_2}$: $I_t(p_1, p_2) = f(I_{t-1}(g_1(p_1, p_2), g_2(p_1, p_2)), I_{t-1}(g_3(p_1, p_2), g_4(p_1, p_2)))$

The I/O dependence list is calculated as follows:

output datum index	input needed data indexes	
0,0 $\hookrightarrow$	$g_1(0,0), g_2(0,0)$	$g_3(0,0), g_4(0,0)$
0,1 $\hookrightarrow$	$g_1(0,1), g_2(0,1)$	$g_3(0,1), g_4(0,1)$
...		

4.2.3.1 The user-defined function

The user-defined function REQ has to be written according to the following rules:

- The function declaration has to be the following
  `stk *REQ(int vo[], int*q)`
  i.e. the function name has to be `REQ`. `REQ` takes as entries an integer vector v_o and a pointer to an integer q. v_o, represents the coordinates of an output datum to be computed and q gives the number of input data needed to compute the given output datum. `REQ` returns a pointer to a list of the q elements giving the coordinates v_i of the needed input data.

- The list of coordinates v_i is constructed through the function
  `stk *add_stk(stk* curr, int vi[], int dim)`
  which is declared in a header file called `mexp.h` and included, by the user, in the C-file defining `REQ`.

```
#include "mexp.h"
stk *REQ(int vo[], int *q){
    ...
    *q=...
    first= new_stk();
    curr=first;
    ...
    — computations of input addresses—-
    vi[]=...
    ...
    curr=add_stk(curr, vi, dim);
    ....
    return first;
}
```

Figure 4.7: template of a user-defined REQ function

Figure 4.7 gives an example of a `REQ` function template. This function corresponds to the loop core of the REQ module in the synthesizable C-code of the TPU; except that, in the synthesizable C-code, the MEXP streams are replaced by the HLS stream and the function declaration is simpler (`void REQ( int vo[], ...)`).

4.2.4 Tiling

The tiling is performed under the following hypothesis:

- Tiles are n-dimensional as the space that they divide.
- Only the orthogonal tilings are considered. This hypothesis is a limitations intended to ease the Supertiling analysis.
- In order to further ease the problem and to reduce the control cost in the generated hardware, the edges of the tiles and the space to be tiled are power of 2.

Let consider:

- a n-dimensional space $S^n = \prod_{i=0}^{n-1} S_i$ with $S_i = 2^{\alpha_i}$ and $\alpha_i \in \mathbb{N}^*$
- a set of M possible tile volumes, specified by the user, $V_T = \bigcup_{z=1}^{M} V^z$ with $V^z = 2^{\beta^z}$ and $\beta^z \in \mathbb{N}^*$

For each user-specified tile volume V^z, the tiling finds all the possible layouts which contain V^z data and exactly divide the space S^n into tiles equal by translation.

Furthermore, for each user-specified tile volume V^z, there are $C(n, \beta^z)$ possible layouts, with $C(n, \beta^z) = \sum_{i=0}^{\beta} C(n-1, \beta^z - i)$ and $C(2, \beta^z) = \beta^z + 1$.

For each couple of tile volume and tile layout, a tiling can be represented by its canonical tile T_k^z.

The set of all computed tilings is

$$T = \left\{ \bigcup_{k=1}^{C(n,\beta^z)} T_k^z, \forall z \in [1, M] \right\} \tag{4.3}$$

For each tiling T_k^z the following properties are true:

- $T_k^z = \prod_{i=0}^{n-1} 2^{\beta_i^z}$ with $\sum_{i=0}^{n-1} \beta_i^z = \beta^z$.
- $\beta_i^z \in [0, \beta^z]$.
- $\beta_i^z \leq \alpha_i, \forall i$

Example For $S = 8 \times 8$ and $V_T = \{4, 8\}$ the following tile layouts are possible:

- $V^z = 4 \rightarrow \beta^z = 2 \rightarrow C(2, 2) = 3$ layouts: $(2^2, 2^0); (2^1; 2^1); (2^0; 2^2)$.
- $V^z = 8 \rightarrow \beta^z = 3 \rightarrow C(2, 3) = 4$ layouts: $(2^3, 2^0); (2^2; 2^1); (2^1; 2^2); (2^0, 2^3)$.

Thus the set of possible tilings is

$$T = \{(4, 1); (2, 2); (1, 4); (8, 1); (4, 2); (2, 4); (1, 8)\}$$

4.2.4.1 The Tiling algorithm

The tiling algorithm is presented in figure 4.8. It is a recursive code that can execute for any I/O space dimension (n) and for any I/O space volume ($bound[]$). It finds all the tiles that exactly divides the I/O coordinate space and which volume is equal to the user-defined volume V.

```
— Leged –
UD: User Defined
C : Calculated
— data structure—
n: space (and tile) dimension — UD
V: tile volume (i.e. nbr of data in a tile) — UD
bound[]: space bounds — UD
T: possible tilings set — C
maxT: maximum size of a given tile edge — C
tile[]: tile edges — C
————————————
TOP(){
    set T=; maxT=V;
    ALGO(n-1);
}
————————————-
ALGO(j){
   for( i=1:maxT){
      if (j+≥1){
         if (bound[j]%i==0){
            tile[j]=i;
            ALGO(j-1);
         }
      }else{
         r=+∏_k tile[k];
         if (0<r<V){
            if(bound[j]%(V/r) == 0 && bound[j]>= DpW){
            tile[j]=V/r;
            T=T ∪ {tile[]}
         }
      }
      i=maxT;
   }
}
```

Figure 4.8: Code of the tiling algorithm

The tile bound has to exactly divide the bound of the space to which the tiling is applied and has to be adapted to the data layout into the used memories. In our designs we have chosen memories with 64 bits per memory word and, as we deal with image processing on B&W images, the pixels are coded on 256 levels of gray. Thus a datum (i.e. a pixel) is coded on 8 bits and a memory word contains 8 data.

In order to exploit this information and avoid exploration of non-realizable solutions, the tile bounds labeled $x(0)$ has to be superior than the number of data per memory Word (DpW), i.e. 8.

4.2.4.2 The tiles labeling

The tiling algorithm previously presented is separately applied to the input and output spaces, S^I and S^O respectively.

Let consider a given couple of input and output tilings $(IT_{k_{in}}^{z_{in}}, OT_{k_{out}}^{z_{out}})$, where IT stands for input tile and the couple (k_{in}, z_{in}) represents the tile layout and tile volume respectively. The same considerations are true for the output tile (OT).

For the given couple of input and output tilings, the input space S^I contains s tiles

and the output space S^O contains d tiles, so that

$$S^I = \bigcup_{j=1}^{s} IT_{k_{in}}^{z_{in}}(j) \text{ and } S^O = \bigcup_{i=1}^{d} OT_{k_{out}}^{z_{out}}(i)$$

with $s = \frac{S^I}{2^{\beta^{z_{in}}}}$ and $d = \frac{S^O}{2^{\beta^{z_{out}}}}$.

The tile indices, i and j are used to identify a specific translation of the canonic tile in a tiling. Figure 4.9 gives the layout of the indices for data spaces with dimensions of 2, 3 and 4.

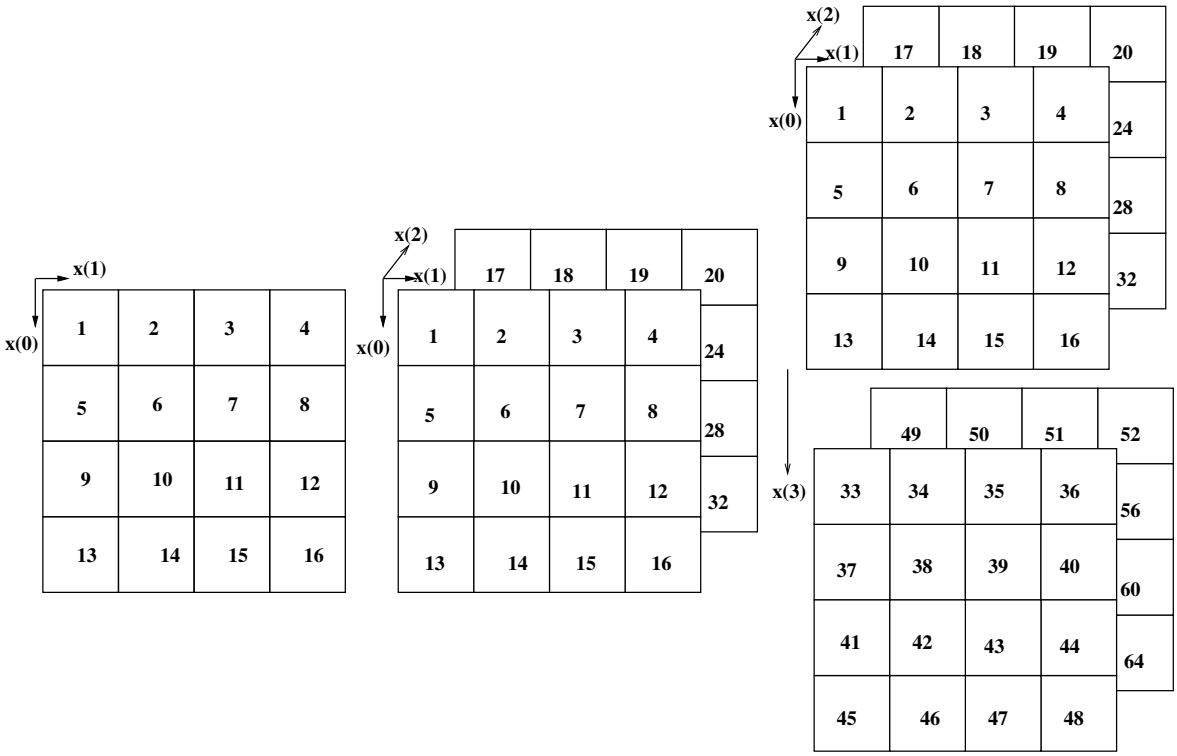


Figure 4.9: Layout of tile labels for a 2-dimensional, a 3-dimensional and a 4-dimensional space

We do not use the index "0" because it is reserved for special case that we will detail in chapter 6.

Given the layout of tile indexes as shown in figure 4.9 and given the coordinates of a specific datum $\overline{x} = (x(0), x(1), \ldots, x(n))$ the tile index is computed through the following formula:

$$i = \left\lfloor \frac{x(0)}{t_k(0)} \right\rfloor + \sum_{\delta=1}^{n-1} \left\{ \left\lfloor \frac{x(\delta)}{t_k(\delta)} \right\rfloor \prod_{i=0}^{\delta-1} \frac{S_i}{t_k(i)} + 1 \right\} \tag{4.4}$$

This formula is used twice: in the projection algorithm which associates the needed input

tile to the computation of a given output tile and in the hardware model of the TPU in order to fetch the requests of the input data.

Thanks to the fact that the edges of the tiles and data spaces are power of two, the formula 4.4 can be re-written as follows:

$$i = (x(0) >> \beta_0) + \sum_{\delta=1}^{n-1} \left\{ (x(\delta) << (\sum_{z=0}^{\delta-1} (\alpha_z - \beta_z) - \beta_\delta) \right\} \tag{4.5}$$

This simplifies the hardware operators dedicated to the computation of the data addresses.

4.2.5 Tiling Projection

For each couple of I/O tilings $(IT_{k_{in}}^{z_{in}}, OT_{k_{out}}^{z_{out}})$, the projection algorithm computes a Super-Tiling matrix $\Sigma^{d \times s}$ which gives the dependencies between the input and output tiles. An element $\sigma_{i,j}$ of the Super-Tiling matrix is 1 if the projection of the output tile $g(OT_{k_{out}}^{z_{out}}(i))$ intersects the input tile $IT_{k_{in}}^{z_{in}}(j)$:

$$\sigma_{i,j} = \begin{cases} 1 & \text{if } g(OT_{k_{out}}^{z_{out}}(i)) \cap IT_{k_{in}}^{z_{in}}(j) \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

The couple $(IT_{k_{in}}^{z_{in}}, OT_{k_{out}}^{z_{out}})$ is, then, characterized by a first estimation of internal memory needed to store the input tiles:

$$IM_{k_{in},k_{out}}^{z_{in},z_{out}} = V^{\beta^{z_{in}}} * \max_i (\sum_j \sigma_{i,j}),\ i \in [1, d] \tag{4.7}$$

where $V^{\beta^{z_{in}}}$ is one of the user specified volumes of the input tile.

The couple of I/O tilings is retained if it respects the user's constraint of maximum internal memory:

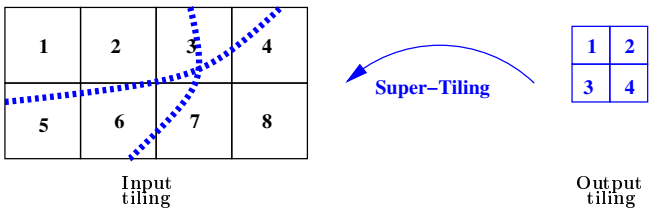
$$IM_{k_{in},k_{out}}^{z_{in},z_{out}} < IM_{max} \tag{4.8}$$

4.2.5.1 Example of the super-tiling application

Figure 4.10 gives an example of the Super-Tiling for the logarithmic sampling which code has been given in figure 4.1. The Super-Tiling matrix is given in figure 4.10(b).

For a given couple of I/O tilings, the line indices of the Σ matrix represent the output tiles and the column indices of the Σ matrix represent the input tiles. The tile indices are shown in figure 4.10(a).

Figure 4.10(b) gives the number of needed input tiles per output tile (on the right of the Σ matrix). It also gives the minimum number of internal buffers needed to implement the solution with the given couple of I/O tilings.



(a) Super-Tiling layout

$$
\begin{array}{c}
 \\
OT_{k_{out}}{}^{nbr.}
\end{array}
\left\{
\begin{array}{c}
\left(\begin{array}{cccccccccccccccc}
1&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&1&1&0&0&0&0&0&0&0&0\\
1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&1\\
0&0&0&0&0&0&0&0&1&1&0&0&1&1&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&1
\end{array}\right.
\\
\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{IT_{k_{in}}\ \text{nbr.}}
\end{array}
\begin{array}{c}
\sum_j \sigma_{i,j} \\
4\\3\\3\\4\\3\\1\\1\\3\\3\\1\\1\\3\\4\\3\\3\\4\\ \max_i(\sum_j \sigma_{i,j}) = 4
\end{array}
$$

(b) $\Sigma^{16\times 16}$ matrix

Figure 4.10: Super-Tiling example for a log sampling

4.2.5.2 The projection algorithm

Figure 4.11 gives the projection algorithm which:

- parses all the elements of an output tile;
- reads from the I/O dependence list which input references are needed;
- computes, through the formula 4.4, the indices of the corresponding I/O tiles;
- sets to 1 an element $\sigma_{i,j}$ of the Σ matrix, when the output tile $OT_{k_{out}}^{z_{out}}(i)$ depends on the input tile $IT_{k_{in}}^{z_{in}}(j)$.

In order to improve the projection algorithm the Σ matrix is compacted as follows: if each line contains more then 32 elements it is broken in $\frac{s}{32}$ elements which are stored into words of 32 bits. This reduces the amount of memory used to store the Σ matrix and the complexity of the following computations performed on it.

In the rest of this dissertation we will refer to a couple of I/O tiling as (IT,OT) without specifying the chosen tile volume and layout (k,z).

4.3 Obtained results

This paragraph presents the Super-Tiling results for the following algorithms:

```
— data structure ——
DMAXi, DMAXo: dimension of the I/O data spaces
IT[]: input tile edges
OT[]: output tile edges
IW[], OW[]: I/O data spaces edges
curr_datum, first _datum: pointers to an element of type stack,
        i.e. the list of I/O dependences; the structure contains:
        .vo[]: vector giving the output datum coordinates
        .vi: pointer to a list of vectors giving the
        needed input data coordinates
        .d: the number of needed input data to compute .vo[]
————————————
int SuperTiling(){
....init.......

curr_datum=first_datum;
while(curr_datum!=NULL){
    if((*curr_datum).q!=0){
        OT_index=find_index(OT,OW,(*curr_datum).vo,DMAXo);
        curr_input=(*curr_datum).vi;
        while(curr_input!=NULL){
            IT_index=find_index(IT,IW,(*curr_input).vi,DMAXi);
            Sigma_matrix[OT_index][IT_index]=1;
            curr_input=(*curr_input).next;
        }
    }
}
return Compute_IM(Sigma_matrix);
}
```

Figure 4.11: Projection algorithm

- A Mean filter on a window 2×2 pixels (M22) applied on an input image of 128×128 pixels
- Rotation (ROT) of 90° of an input image of 128×128 pixels
- A Logarithmic sampling (LOG) of an input image of 256×256 pixels (the output image contains 128×128 pixels)

The analysis has been run for input and output tiles containing 32, 16 and 8 data each. The list of the exploration results is given in table 4.1. As we consider only orthogonal tiles with the bound $x(0)$ superior than 8, we have 36 possible couples of I/O tilings per algorithm.

The analyzed solutions are divided in 3 groups according to the size of the input tile. The groups are: from solution 1 to 18 for an input tile volume of 32 data, from solution 19 to 30 for an input tile volume of 16 data and from solution 31 to 36 for an input tile volume of 8 data.

The figures 4.13(a) , 4.13(b) and 4.13(c) give the amount of IM used to realize any of the 36 analyzed solutions. The IM amounts correspond to the number of IT per OT given in table 4.1 multiplied by the input tile volume. In the figures 4.13(a) , 4.13(b) and 4.13(c), the amounts of IM are re-ordered from the lower to the higher value. From the figures, we can see that the used IM amounts depend on the sizes of the input and output tiles and on the kind of array references performed in the analyzed algorithm.

Because of the affinity of the array references, the best solutions for the **mean filter** (figure 4.13(a)) are those using the same tile layouts for both the input and output spaces. The solutions require a bigger amount of internal memory for a bigger input tile size.

Despite the affinity of the array references, the best solutions for the **rotation algorithm** (figure 4.13(b) and code 4.12) do not have the same I/O tile layouts. In fact the

```
for(i=0:128){
   for(j=0:128){
      A[i][j]=B[j][-i+128];
   }
}
```

Figure 4.12: Code of the ROT algorithm

Super-Tiling takes into account the inversion of the subscripts i and j from the input to the output space. The best solution for ROT algorithm uses an input tile layout having more lines than columns. Furthermore the 3 groups of solutions have all a solution with the minimum of internal memory amount, which means that for this algorithm the tiles layout is preponderant with respect to the tile size.

Let consider the histograms in figures 4.13(a) , 4.13(b) and 4.13(c). For the applications with affine array references, the histograms variate by steps. For the Log sampling (figure 4.13(c)), the envelope of the histogram is smoother, due to the non-affinity of array references.

The Super-Tiling step is not sufficient to choose an optimized solution. Let consider, for example, the solutions 18 and 36 for the Log sampling. The solution 36 uses a minor internal memory amount, as shown in figure 4.13(c), but it accesses a greater amount of input tiles per output tile (see the number of internal buffers in table 4.1) and thus, it could have a worse temporal performance. To formally choose among these solutions other explorations are necessary and they will be described in the next chapters.

4.4 Conclusion

This chapter presented the Super-Tiling which is a method to apply the tiling to the loop-nests not necessarily affine.

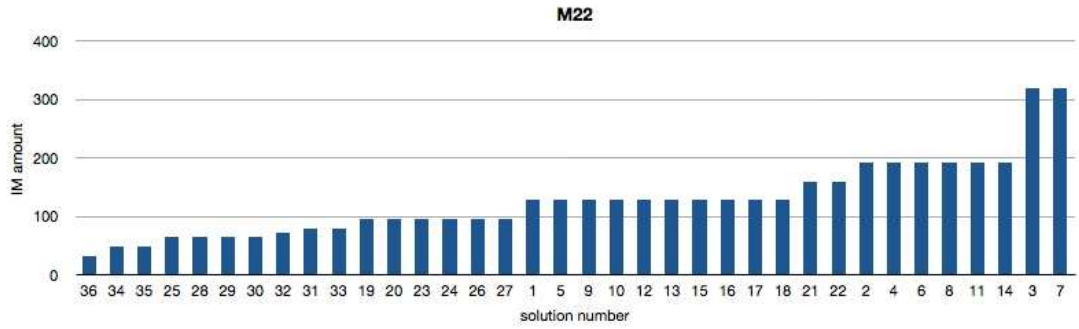
The main idea of the Super-Tiling is to separately partition the input and output data sets, with a regular tiling. Then the output tiling is projected on the input tiling according to the array reference law.

The intersection of the projection with the input tiling gives the dependences between the input and the output tiles.

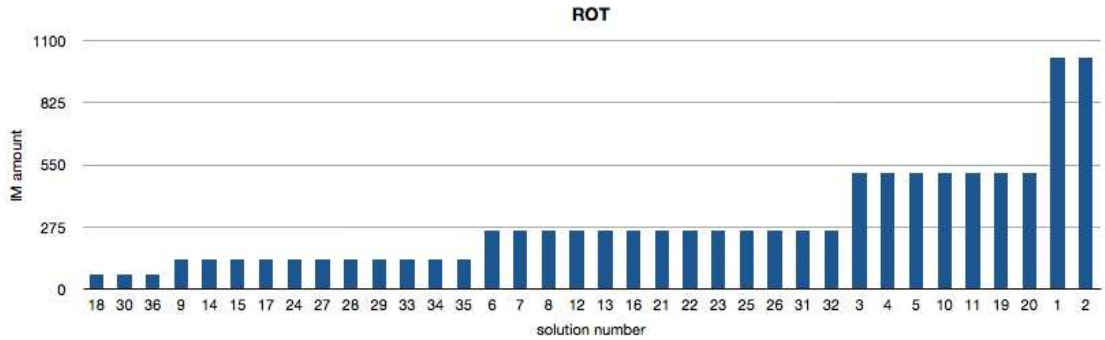
The Super-tiling has three steps: a dynamic profiling of the array references of the algorithm, a tiling which separately apply to the input and output data spaces and a projection which relates the I/O tilings to each other.

The output data tiling corresponds to a partitioning of the output computation space. Thus, the tiling enables the possibility of parallelize the different output tile computations with each other. It also enables the possibility to parallelize the prefetching of the input tiles and the computing of the output tiles.

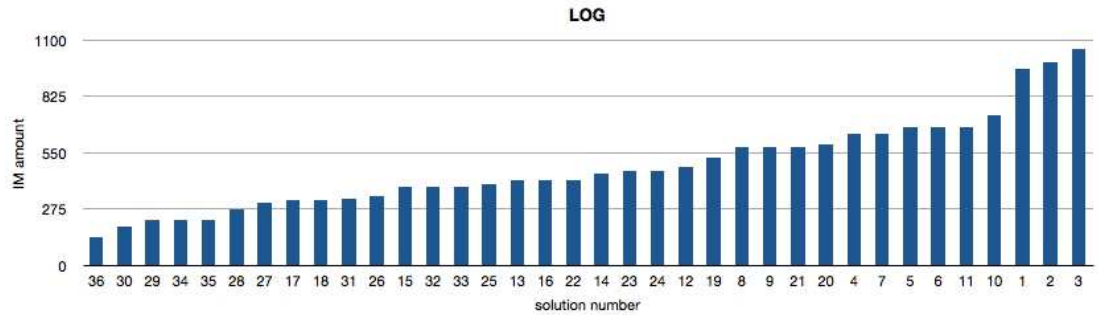
The results show that the non-affinity of the array references influences the tile layout and, thus, the amount of internal memory used to realize the hardware.



(a) MEAN 22, affine array references



(b) ROT, affine array references with inversion of coordinates i and j



(c) LOG, non-affine array references

Figure 4.13: Histograms corresponding to the nbr. of IT per OT given in table 4.1. These values are multiplied by the tile volume.

sol	IT	OT	nbr. of IT per OT		
			LOG	ROT	M22
1	(32,1)	(32,1)	30	32	4
2		(16,2)	31	32	6
3		(8,4)	33	16	10
4		(16,1)	20	16	6
5		(8,2)	21	16	4
6		(8,1)	21	8	6
7	(16,2)	(32,1)	20	8	10
8		(16,2)	18	8	6
9		(8,4)	18	4	4
10		(16,1)	23	16	4
11		(8,2)	21	16	6
12		(8,1)	15	8	4
13	(8,4)	(32,1)	13	8	4
14		(16,2)	14	4	6
15		(8,4)	12	4	4
16		(16,1)	13	8	4
17		(8,2)	10	4	4
18		(8,1)	10	2	4
sol	IT	OT	nbr. of IT per OT		
			LOG	ROT	M22
19	(16,1)	(32,1)	33	32	6
20		(16,2)	37	32	6
21		(8,4)	36	16	10
22		(16,1)	26	16	10
23		(8,2)	29	16	6
24		(8,1)	29	8	6
25	(8,2)	(32,1)	25	16	4
26		(16,2)	21	16	6
27		(8,4)	19	8	6
28		(16,1)	17	8	4
29		(8,2)	14	8	4
30		(8,1)	12	4	4
sol	IT	OT	nbr. of IT per OT		
			LOG	ROT	M22
31	(8,1)	(32,1)	41	32	10
32		(16,2)	48	32	9
33		(8,4)	48	16	10
34		(16,1)	28	16	6
35		(8,2)	28	16	6
36		(8,1)	17	8	4

Table 4.1: Table of the Super-Tiling explorations for ROT,LOG and M22 algorithms

Chapter 5

Scheduling

THIS chapter presents the second step of the MEXP methodology: the Scheduling. The scheduling is aimed to choose an optimized order of the output tile computations. The scheduling of the tiles computations can be optimized by reducing the amount of used internal memory or the amount of input tiles copied from the external memory. The reduction of the the input tiles copied from the external memory has two effects: it reduces the system power consumption and improves the temporal performance of the system, by reducing the time to pre-fetch the input. The scheduling also eliminates the useless computations and chooses the first output tile to be computed in order to further reduce the used internal memory. The re-organization of the output tiles computations is performed by solving the traveling salesman problem through a genetic algorithm, i.e. an evolutive algorithm.

Chapter contents

This chapter goes though the following contents:

- Motivation of the scheduling with a presentation of the different steps of the scheduling.
- Presentation of the used genetic algorithms.
- Results supporting the chosen algorithms.

5.1 Introduction to the scheduling

After the Super-Tiling has been applied, MEXP has to deal with the scheduling and mapping of the output tiles computations.

The output of the Super-Tiling is a matrix Σ, which gives, for each possible couple of input and output tilings, the dependences between the input and output tiles.

The scheduling has three steps:

- The elimination of useless computations.

- The re-ordering of the output tiles computations.
- The choice of the first output tile to be computed.

Due to the non-affinity of input data references it may occur that a needed input data is out of the actually existing input data space and thus corresponds to an invalid data request.

Generally an image processing algorithm gives a default value as response to an invalid input data request. Let consider, for example, the figure 5.1 in which we show the output of a logarithmic sampling. The black border of the output image is due to invalid input data requests.

Figure 5.1: Example of an LOG sampling output with black borders

If a whole output tile $OT(i)$ contains only invalid data requests, which means that there are only 0_s on the corresponding σ_i line of the super-tiling matrix Σ, **MEXP suppresses the useless computations** that should produce the output tile $OT(i)$ and thus the σ_i line of the Σ matrix.

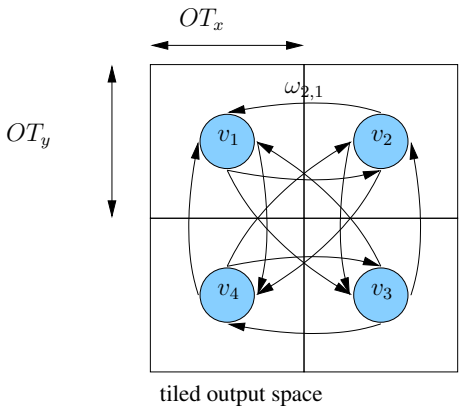


Figure 5.2: Example of a graph corresponding to a 2-dimensional space with 4 output tiles

Under the hypothesis that the application analyzed does not have self dependences[a],

[a] i.e. no dependences between output tiles

the output tiles computations can be re-ordered without any restriction.

The scheduling is aimed to compute an **optimized timetable of the output tiles computations** in order to minimize at once the number of external memory accesses and the amount of internal memory.

The scheduling problem can be formulated as a graph theory problem: given a possible the Super-tiling, i.e. a possible couple of input and output tilings and their dependences, it is possible to let correspond a weighted graph to the output tilings (see figure 5.2).

Each output tile, corresponds to a graph vertex and the passage from the computation of an output tile to another is represented through a weighted arc connecting the two corresponding vertices. The weight of the arc is the cost of this passage in term of either internal memory or number of external memory accesses.

The problem of scheduling the output tiles computations corresponds to visiting all the vertices of the graph exactly once by minimizing the chosen cost.

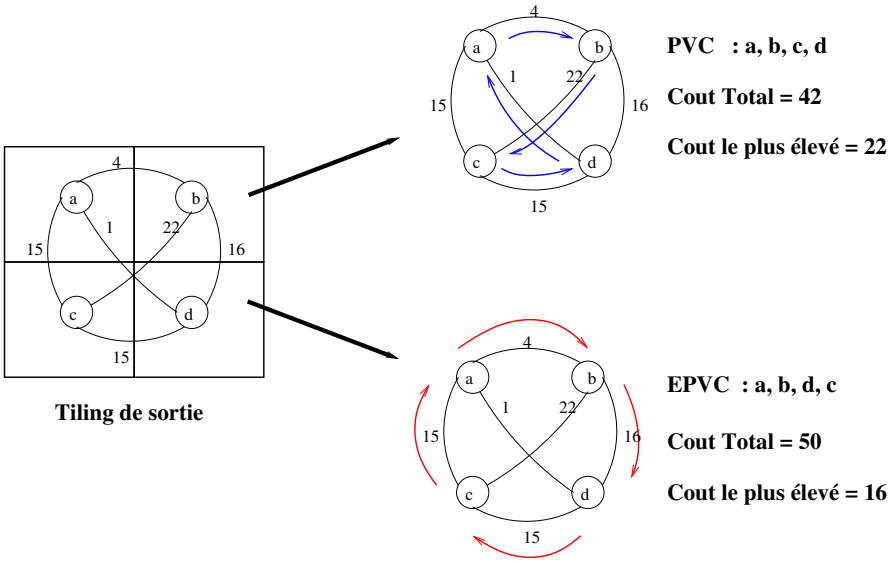


Figure 5.3: Example of a graph for which the two problems of BTSP and TSP have different solutions. The blue round-trip is a solution for a TSP with a total cost of the round trip of 11 and a maximum edge cost encountered of 6; the red round-trip is a solution for the BTSP with a total cost of 12 and a maximum edge cost encountered of 4

This problem can be formulated as one of two NP-hard problems: the classical Traveling Salesman Problem (TSP) or the Bottleneck Traveling Salesman Problem (BTSP), depending on which optimization criterion is chosen to compute the best round-trip.

Many readers will probably be familiar with the Traveling Salesman Problem (TSP) which was first proposed by Sir William Rowan Hamilton in the 1800s and since then has been well studied especially in the field of combinatorial optimizations.

Given a number of cities and the costs to travel from a city to another, the TSP wants to find the less expansive round-trip which visits all the cities exactly once.

The Bottleneck TSP is related to the TSP but instead of the total cost of the trip it

wants to minimize the maximum of all the costs that the traveler has to afford during his trip.

As an example let consider the graph in figure 5.3, the BTSP and the TSP have two different solutions each one optimizing a different criterion.

5.2 Mathematical formulation of the TSP and the BTSP

These definitions are adapted from [100]

Complete graph. A complete (weighted and oriented) graph of order d is a graph with d vertices, where each vertex is connected (through a weighted and oriented) arc to every other.

The cost matrix. Given a complete (weighted and oriented) graph of order d, the weights of the graph arcs are given by a squared $d \times d$ matrix

$$V_C = \begin{pmatrix} 0 & \omega_{1,2} & \dots & \omega_{1,d} \\ \omega_{2,1} & 0 & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{d,1} & \dots & \dots & 0 \end{pmatrix}$$

an element of the matrix $\omega_{r,c}$ corresponds to the cost to pass from the node v_r to the node v_c.

The Hamiltonian cycle. Given a d complete oriented graph $G(V, E)$, where V are the vertices and E the oriented edges connecting them, a Hamiltonian cycle is a cycle $\pi = (v_1, v_2, \dots, v_d, v_1)$ that visits once all the vertices of the graph.

The Traveling Salesman Problem. Given a d complete oriented graph $G(V, E)$, a cost matrix V_C giving the cost to pass from each vertex to every other and a set of possible Hamiltonian cycles Π of G, the TSP exact solution is the cycle $\pi = (v_1, v_2, \dots, v_d, v_1) \in \Pi$ such that the **total cycle cost** (TCC) is minimized:

$$TCC = \min \left\{ \omega_{d,1} + \sum_{i=1}^{n-1} \omega_{i,i+1} \right\} \tag{5.1}$$

The Bottleneck Traveling Salesman Problem. Given a d complete oriented graph $G(V, E)$, a cost matrix V_C and a set of possible Hamiltonian cycles Π of G, the BTSP exact solution is the cycle $\pi = (v_1, v_2, \dots, v_d, v_1) \in \Pi$ such that the **largest edge cost** (LEC) is minimized:

$$LEC = \min \left\{ \max \left\{ \omega_{d,1}, \bigcup_{i=1}^{n-1} \{\omega_{i,i+1}\} \right\} \right\} \tag{5.2}$$

The cost $\omega_{r,c}$ can be computed according to different formula which depend on the system characteristic that has to be optimized.

5.3 The possible costs and problems to optimize the TPU

As presented in section 4.2.5 of chapter 4, the output of the Super-Tiling is a Σ matrix giving the dependences between the considered input and the output tilings. An element $\sigma_{i,j}$ of the Σ matrix is 1 if the input tile $IT(j)$ is used for the computation of the output tile $OT(i)$.

Given two output tiles $OT(c)$ and $OT(r)$, the corresponding lines in the Σ matrix are binary words that can be combined through the following operators:

- A boolean "or", noted $+$. It has the following truth table: $\begin{pmatrix} \sigma_{c,j} & \sigma_{r,j} & \sigma_{c,j}+\sigma_{r,j} \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
- A boolean "and", noted $\bullet$. It has the following truth table: $\begin{pmatrix} \sigma_{c,j} & \sigma_{r,j} & \sigma_{c,j}\bullet\sigma_{r,j} \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
- A boolean "XOR", noted $\oplus$. It has the following truth table: $\begin{pmatrix} \sigma_{c,j} & \sigma_{r,j} & \sigma_{c,j}\bullet\sigma_{r,j} \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$
- A boolean "not", noted $\overline{\sigma_{r,j}}$.

As presented in chapter 3, when the TPU computes the tiles $OT(r)$ and $OT(c)$ one after the other, it performs two parallel macro-tasks: the computing of the output tile $OT(r)$ and the pre-fetching of the input tiles needed to compute the next output tile $OT(c)$.

The re-order of the output tile computations can be aimed to minimize three objective functions (also called costs):

- The amount of input tiles loaded from the external memory.
- The amount of used internal memory.
- The amount of input tiles changing between the computation of two successive output tiles.

The amount of input tiles loaded from the external memory into the internal buffers is:

$$\omega_1(r,c) = \sum_{j=0}^{s-1} \{\sigma_{c,j} \bullet \overline{\sigma_{r,j}}\} \tag{5.3}$$

Example: given $\sigma_r = (0\ 1\ 1\ 0)$ and $\sigma_c = (1\ 1\ 1\ 0)$, then $\omega_1(r,c) = 1$ and $\omega_1(c,r) = 0$.

As shown by figure 5.4, the value of $\omega_1(r,c)$ is related to the used amount of internal memory, the temporal performance and the power consumption of the system.

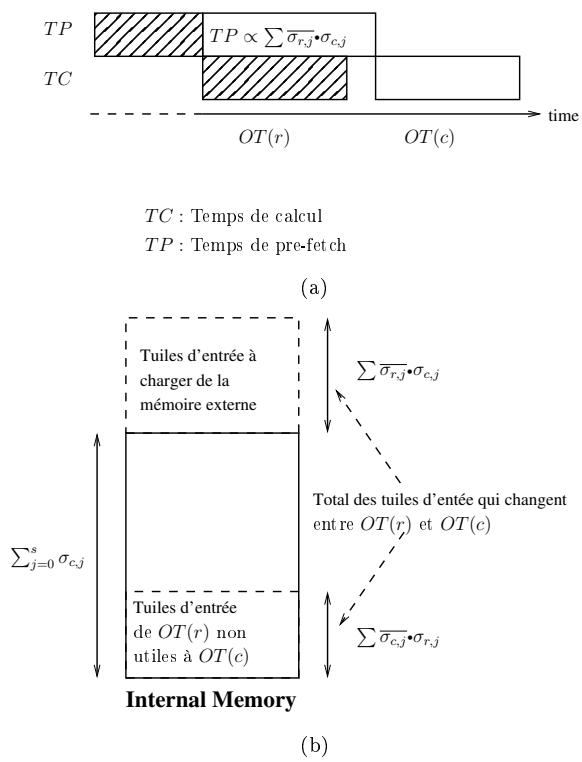


Figure 5.4

In particular, if we solve the BTSP with the cost $\omega_1(r, c)$ we minimize the time needed to perform the pre-fetching, for each computed output tile. At the same time, we minimize the part of the internal memory aimed to store the pre-fetched data.

If we solve the TSP with the cost $\omega_1(r, c)$ we reduce the total amount of input tiles pre-fetched and, thus, we reduce the power consumption of the system.

For a couple of two output tiles $OT(r)$ and $OT(c)$, **the amount of the used internal memory** is:

$$\omega_2(r, c) = \sum_{j=0}^{s-1} \{\sigma_{r,j} + \sigma_{c,j}\} \tag{5.4}$$

Example: given $\sigma_r = (0\ 1\ 1\ 0)$ and $\sigma_c = (1\ 1\ 1\ 0)$, then $\omega_2(r, c) = \omega_2(c, r) = 3$.

A scheduling, optimized with respect to this cost, is fund by solving a corresponding BTSP. It is possible to express ω_2 as follows:

$$\omega_2(r, c) = \omega_1(r, c) + \sum_{j=0}^{s-1} \sigma_{r,j}$$

This means that by solving the BTSP with respect to ω_2, we also reduce the the time needed to perform the pre-fetching (cf. figure 5.4).

The last considered cost corresponds to **the amount of the input tiles changing between the computation of the two successive output tiles** $OT(r)$ and $OT(c)$. This cost is

$$\omega_3(r,c) = \sum_{j=0}^{s-1} \{\sigma_{r,j} \oplus \sigma_{c,j}\} \tag{5.5}$$

Example: given $\sigma_r = (0\ 1\ 1\ 0)$ and $\sigma_c = (1\ 1\ 1\ 0)$, then $\omega_3(r,c) = \omega_3(c,r) = 1$.

It is possible to express ω_3 as follows:

$$\omega_3(r,c) = \sum_{j=0}^{s-1} \{\sigma_{c,j} \bullet \overline{\sigma_{r,j}} + \sigma_{r,j} \bullet \overline{\sigma_{c,j}}\}$$

Figure 5.4 shows that ω_3 is related to both the length of the time to pre-fetch the input data and to the amount of used internal memory. In particular by solving the BTSP with respect to the cost ω_3 we reduce both the time to pre-fetch data, for each output tile computation, and the amount of internal memory needed to store the input tiles changing between the computations of two successive output tiles. If we solve the TSP with respect to the cost ω_3 we reduce the power consumption to perform the whole target algorithm.

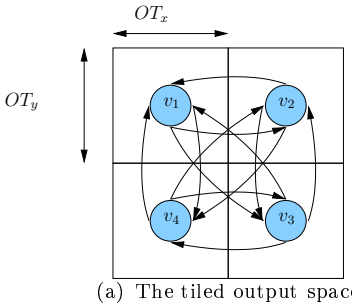
Definition A possible couple of input and output tilings corresponds to **a BTSP or a TSP instance**, i.e. a case of the problem to be solved. For each MEXP exploration, the user specifies several possible volumes for the input tile and several possible volumes for the output tile. For each possible input and output volumes there are several possible layouts. Each combination of all the possible I/O layouts represents a BTSP or a TSP instance. Thus a MEXP exploration could have to solve hundreds of BTSP or TSP instances.

The cost specified in the equation (5.3) produces an asymmetric instantiation of the problem, in fact $\omega_1(r,c) \neq \omega_1(c,r)$ i.e. the cost to pass from the computation of $OT(r)$ to the computation of $OT(c)$ is not the same as the cost to pass from the computation of $OT(c)$ to the computation of $OT(r)$.

The asymmetric BTPS or TSP has a higher complexity than the symmetric BTPS or TSP. In fact, an asymmetric BTSP or TSP may require twice the number of operations needed to solve a symmetric BTSP or TSP, it depends on the algorithm used to solve the problem. In order to reduce the complexity of the algorithm, we prefer using the costs specified in the equations (5.4) and (5.5), which have a symmetric instantiation.

Figure 5.5 gives examples of different V_c matrices according to the different considered costs. Figures 5.5(c), 5.5(d) and 5.5(e) give the V_c matrices for the costs given in equation (5.3), (5.4) and (5.5) respectively. Figure 5.5(c) gives an asymmetric matrix, while 5.5(d)

and 5.5(e) give symmetric matrices, which can be replaced by triangular matrices. This reduces the number of operations needed to prepare the input data for the BTSP or TSP solver. The complexity of the problem depends on the number (d) of the nodes to be



(a) The tiled output space associated to a graph

$$\Sigma = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(b) The Σ matrix giving the mapping between the input and output tiling

$$V_C = \begin{pmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

(c) The cost matrix for the optimization criterion of equation 5.3

$$V_C = \begin{pmatrix} 0 & 2 & 3 & 4 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

(d) The cost matrix for the optimization criterion of equation 5.4

$$V_C = \begin{pmatrix} 0 & 1 & 3 & 4 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

(e) The cost matrix for the optimization criterion of equation 5.5

Figure 5.5: Example of a symmetric (B)TSP instantiation for an output tiling applied to a 2-dimensional output space and producing 4 output tiles

visited during the round-trip. For large instances (i.e. d>20), it is impossible to solve the problem with an exhaustive method.

5.4 The genetic algorithms, the TSP and the BTSP

In our work, we have used a genetic algorithm to solve both the TSP and the BTSP. In particular, we have solved the TSP by minimizing the TTC with respect to the cost of equation (5.5) and the BTSP by minimizing the LEC with respect to the cost of equation (5.4).

The genetic algorithms (GAs) are part of the evolutionary algorithms and were introduced by John Holland between the 70s and the 80s. They are based on Darwin's evolution theory of natural selection and "survival of the fittest".

The GAs make evolve a **population** of individuals which are represented by genotypes (GTYPE) called also chromosomes. A **genotype** encodes the way an individual possesses a specific characteristic; the characteristic is evaluated by solving a **fitness function** and the value of the fitness function for a particular individual is called **phenotype** (PTYPE).

In an optimization problem with a single objective function (i.e. a single optimization criterion), the fitness function is the optimization criterion, a genotype is a given solution to the problem and the phenotype is the value of the cost to be optimized for the given solution.

For example in the BTSP a genotype is a particular Hamiltonian cycle, the fitness function is the LEC and the phenotype is the value, for the given solution, of the one of the costs defined in the equations (5.3), (5.4) or (5.5).

The evolution, which selects genotypes having more and more improved phenotypes, is an iterative process going through the following steps:

1. The creation of an initial population
2. The evaluation of the phenotypes for all the individuals in the population
3. The selection of the best candidate parents to make evolve the population
4. The reproduction of the parents (also called crossover)
5. The mutation of some individuals of the population
6. The update of the population and iteration from point 2 until the population reaches the desired level of evolution

Several problems exist which are related to the parameterization of a genetic algorithm; among them there are the following:

- The choice of the code to encode the genotypes (the first used by Holland was the binary code, but the most spread to solve a classical TSP has been the cardinal encoding, i.e. a natural number per each city in a tour)
- The choice of the operators implementing the selection, the crossover and the mutation
- The size of the population and the criteria that stop the genetic algorithm when the desired evolution is achieved

All these parameterizations offer a large possibility of a GA implementation; the choices that we made in our work are based on previous works. For example the work presented in [101] classifies the different operators for different instances of a classical TSP; on the basis of this study we have selected the operators to implement the crossover and the mutation; they will be presented in detail in the following paragraph.

The work presented in [102] shows that a hybrid GA, i.e. a GA including heuristics, converges faster and finds better solutions, thus as suggested in this paper we have constructed the initial population by using the Nearest Neighbor algorithm by choosing as initial point some of the cities in the round-trip.

The next paragraph gives the detail of the (B)TSP implementation through the hybrid GA.

5.4.1 The Algorithms

The following algorithms are run on a given d graph $G(V, E)$, with $d \in \mathbb{N}^*$. The vertices of the graph correspond to the output tiles to be computed.

5.4.1.1 The creation of the initial population of genotypes

The initial population of the GA is a sub-set of all possible Hamiltonian cycles of the complete graph representing the output tiling. It is chosen by running the Nearest Neighbor (NN) algorithm for the TSP defined in paragraph 5.2 and all the Hamiltonian cycles are found by choosing each vertex as a starting point of the cycle.

The complexity of the whole GA depends on the number of individuals in the population. The work [102] shows that the optimal population size depends on the number of vertices d and has to be $d \leq N_{pop} \leq 2 * d$. In our study, as we can have very high value of d (ex. 8192) and in order to keep a low complexity of the algorithm, we limit the population size to the $\max(d, 256)$.

The NN is performed as follows:

1. For all individuals in the population to be constructed, select a vertex as starting point of the cycle and mark it as visited

2. Find the next vertex having the minimum access cost from the current vertex and mark it as visited

3. While there are vertices which have not been visited iterate 2 and 3

4. Once the tour is found copy it back as a population member

5.4.1.2 The evaluation of the population through the phenotype and the selection of the parents

Once the initial population has been set we compute the phenotype per each individual in the population. The phenotype is the cost of the Largest Edge Cost (LEC) for the BTSP and the Total Cycle Cost (TCC) for the TSP.

Then, in order to prepare the future evolution of the population, two Hamiltonian cycles are selected as parents. We have tried three selection criteria:

- **The elitism**; where the best two Hamiltonian cycles are selected as future possible parents. This means that the selected parents have the minimum LEC for the BTSP and the minimum TTC for the TSP.

- **The roulette wheel**; where a probability is associated to each individual with respect to its phenotype. This means that the better the phenotype of an individual is, the greater the chance it will be selected as a parent. But, on the contrary of the elitism, it is not guaranteed that the fittest members go to the next generation.

- **The Tournament**; where 4 individuals are chosen randomly and the best two ones among the 4 are selected as parents.

The two selected parents generate two new individuals which will replace the two worst individuals and make the population evolve. The elitism and the roulette wheel will

always select more or less the same individuals and this increases the probability to find a sub-optimal solution, i.e. a local minimum for the optimization criterion.

In order to avoid the local minima we have chosen the tournament algorithm and we have introduced some diversity in the population through the generation of a random child with a certain probability. This probability is another possible parameter of the genetic algorithm; in our case we have chosen a probability of 2 random generated children each 10 generations, because experiments show that a greater probability prevent the convergence of the genetic algorithm and a lower probability does not introduce enough diversity.

Another mechanism which may prevent the convergence towards a sub-optimal solution is the mutation, which will be presented later.

5.4.1.3 The crossover

The reproduction of the two selected parents is run through the crossover operator called order crossover (OX1) in [101]. This operator has been shown to have a good behavior with respect to the accuracy of the solution and the convergence of the search.

The OX1 constructs the two children by selecting a sub-path from one parent and the order of the rest of vertices from the other. Next example is taken from [101]

Example Let consider the two parents

$$(1, 2, 3, 4, 5, 6, 7, 8)$$

and

$$(2, 4, 6, 8, 7, 5, 3, 1)$$

Suppose that two cuts are chosen: the first cut between the second and the third elements and the second cut between the fifth and sixth elements of the parents:

$$(1, 2|3, 4, 5|6, 7, 8)$$

and

$$(2, 4|6, 8, 7|5, 3, 1)$$

The vertices between the two cuts form a sub-path s_p. The first child takes the sub-path s_p from the first parent and the second child takes the sub-path s_p from the second parent.

$$(*, *|3, 4, 5|*, *, *)$$

and

$$(*, *|6, 8, 7|*, *, *)$$

The rest of the children paths are constructed by copying the missing vertices according to the order given by the parent from which they have NOT inherited the sub-path s_p.

$$(8, 7|3, 4, 5|1, 2, 6)$$

and

$$(4, 5|6, 8, 7|1, 2, 3)$$

5.4.1.4 The mutation

We perform a displacement mutation (DM), which has been shown to be efficient in [101].

The DM is presented in figure 5.6. It takes from the child to be mutated a sub-path of a random length and place it in another random-chosen position.

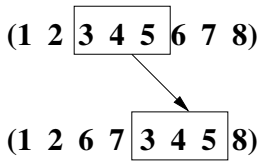


Figure 5.6: Example of a Displacement Mutation

5.4.1.5 The criterion to stop the evolution of the population

Several termination criteria exist to stop the genetic algorithm: a fixed number of iterations, a threshold value for the fitness function or a convergence criterion. In our work we have chosen a convergence criterion associated with a maximum number of iterations. We also use a condition to ensure that the algorithm run for a minimum number of iterations. The chosen criteria are:

- $100 \leq N_{iterations} \leq 1000$
- convergence criterion: the mean fitness of the population is less or equal to the fitness of the best genotype

5.5 Results supporting the chosen algorithm

In this paragraph we characterize three TSP solvers:

- The TSP and the BTSP solvers, which have been developed through the genetic algorithm and thus will be noted GATSP and GABTSP.
- A TSP solver, using the Lin-Kernighan method, which is a part of the Concorde.
- The Concorde, which is a reference TSP solver.

We have analyzed the results for three different algorithms applied on input images of different sizes, in particular:

- An algorithm (ALGO1) applied on an input image 128 x 128 and producing an output image 64x64
- An algorithm (ALGO2) applied on an input image 512 x 512 and producing an output image 256x128

- An algorithm (ALGO3) applied on an input image 1024 x 1024 and producing an output image 256x256

For each algorithm, we have performed some of the explorations presented in table 5.1.

	output tiles volume	output tiles layout	nbr. of analyzed instances
EXP1	128	128x1, 64x2, 32x4, 16x8, 8x16	25
EXP2	64	64x1, 32x2, 16x4, 8x8	20
EXP3	32	32x1, 16x2, 8x4	15
EXP4	16	16x1, 8x2	10
EXP5	8	8x1	5

Table 5.1: Explorations run for an input tile volume of 128 with an input tile layout among 128x1, 64x2, 32x4, 16x8, 8x16

In particular, we have analyzed the results of EXP1, EXP2, EXP3, EXP4 and EXP5 for ALGO1; EXP1, EXP2, EXP3 for ALGO2 and EXP1 for ALGO3 for a total of 155 instances of (B)TSP analyzed.

ALGO	EXP	# inst	# graph vertices		
			min.	av.	max.
ALGO1	EXP1	20	-	32	-
	EXP2	20	63	63.75	64
	EXP3	15	124	123.3	126
	EXP4	10	244	245.5	246
	EXP5	5	-	481	-
ALGO2	EXP1	25	-	256	-
	EXP2	20	-	512	-
	EXP3	15	-	1024	-
ALGO3	EXP1	25	480	491.6	510

Table 5.2: Table giving the complexity of the analyzed instances in terms of number of vertices to be visited. “min., av. and max." are respectively the minimum the average and the maximum of vertices in the analyzed number of instances (# inst)

Table 5.2 gives the complexity of the analyzed instances in terms of vertices to be visited.

5.5.1 Comparison between the GATSP, the Lin-Kernighan solver and the Concorde

We will compare the GATSP to a TSP solver based on the Lin-Kernighan method and to a reference TSP solver: the Concorde.

These three solvers (GATSP, Lin-Kernighan and Concorde) are all heuristics, thus it is not sure that they will find an optimal solution. But, the Concorde [103], which has

been developed by William Cook, is able to obtain optimal solutions for many instances of the TSP, even for very large ones[b], thus it will be used as reference. The Concorde uses the cutting plane method and solves symmetric TSP by using a linear programming solver: the Qsopt.

The Lin-Kernighan solver uses a set of complex moves (called kicks) to approach the optimal solution of a TSP and we have used the one that is furnished with the Concorde project.

The table 5.3 compares the results of the Lin-Kernighan solver with respect to the GATSP. The Lin-Kernighan has been run with a number of kicks of $d \times 100$. This table shows that, with this configuration, the Lin-Kernighan solver is faster than the GATSP solver. It also gives the rank[c] of the two solvers with respect the TCC, and shows that the GATSP finds solutions which are better than the Lin-Kernighan solution in 99% of cases. The mean error of the solutions found with the Lin-Kernighan solver with respect to the solutions found with the GATSP solver can be up to 101%.

ALGO	EXP	# inst	TCC			Exec. Time (sec.)		LK error WRT GATSP
			GATSP	LK	comm	GATSP	LK	
ALGO1	EXP1	20	19	1	0	<1	<1	44
	EXP2	20	20	0	0	1	1	69
	EXP3	15	15	0	0	2	2	92
	EXP4	10	10	0	0	5	4	86
	EXP5	5	5	0	0	8	5	101
ALGO1	EXP1	25	25	0	0	28	20	41
	EXP2	20	20	0	0	58	36	49
	EXP3	15	15	0	0	128	58	67
ALGO3	EXP1	25	25	0	0	9	9	64

Table 5.3: Table comparing the results of a GATSP and a Lin-Kernighan solver with a number of kicks of $d \times 100$, the experiments have been run for different target algorithms (ALGO) and different explorations (EXP)

Table 5.4 gives the comparison between the Concorde and the Lin-Kernighan with 10^6 kicks. Results show that, for this configuration, the Lin-Kernighan is slower than the GATSP. The GATSP finds still better solutions in 90% of cases. And the error of the solutions found with the Lin-Kernighan solver with respect the solutions of th GATSP is up to 35%.

Table 5.5 compares the quality of the results of the GATSP with respect to the Concorde. As attended the Concorde has better or equivalent solutions in 86% of cases.

This table shows also that the TSPGA is up to 76 times faster than the Concorde and, according to us, this is an advantage to be used in a DSE tool.

[b] up to 15.112 vertices in a graph, which is the largest instance of the TSP solved up to now.

[c] nbr. of instances with respect to all the analyzed instances for which the considered solver has a better solution.

ALGO	EXP	# inst	TCC			Exec. Time (sec.)		LK error WRT GATSP
			GATSP	LK	comm	GATSP	LK	
ALGO1	EXP1	20	17	3	0	<1	29	17
	EXP2	20	15	5	0	1	71	31
	EXP3	15	12	2	1	2	106	35
	EXP4	10	10	0	0	5	113	24
	EXP5	5	5	0	0	8	78	29
ALGO1	EXP1	25	25	0	0	28	376	13
	EXP2	20	20	0	0	58	388	16
	EXP3	15	15	0	0	128	339	24
ALGO3	EXP1	25	21	4	0	9	175	25

Table 5.4: Table comparing the results of a GATSP and a Lin-Kernighan solver with a number of kicks of 10^6, the experiments have been run for different target algorithms (ALGO) and different explorations (EXP)

The table 5.5 also gives the mean error of the TSPGA solutions with respect to the Concorde solutions. We can see that the mean error is up to 16 % for the TTC.

On the basis of these experiments we have decided to let the user chose among the usage of the Concorde and the TSPGA in order to have more precision for explorations on small solution spaces and to be faster to explore large solution spaces.

ALGO	EXP	# inst	TCC			Exec. Time (sec.)		GATSP error WRT CONC
			GATSP	CONC	comm	GATSP	CONC	
ALGO1	EXP1	20	5	10	5	<1	2	4
	EXP2	20	1	13	6	1	3	5
	EXP3	15	0	14	1	2	6	6
	EXP4	10	1	9	0	5	32	10
	EXP5	5	0	5	0	8	611	16
ALGO1	EXP1	25	0	25	0	28	296	4
	EXP2	20	0	20	0	58	818	6
	EXP3	15	0	15	0	128	4177	10
ALGO3	EXP1	25	1	23	1	9	34	4

Table 5.5: Table comparing the results of a GATSP and The Concorde solver, the experiments have been run for different target algorithms (ALGO) and different explorations (EXP)

5.5.2 Comparison between the GABTSP and the GATSP

Figure 5.7 gives the **amount of used internal memory per each output tile**, during all the round-trip. We can see that the BTSP allows to use less internal memory than the TSP. The TSP has one or few maxima of the internal memory which are higher than

the maximum of the BTSP solution, but except for these maxima it generally copies less input tiles from the external memory.

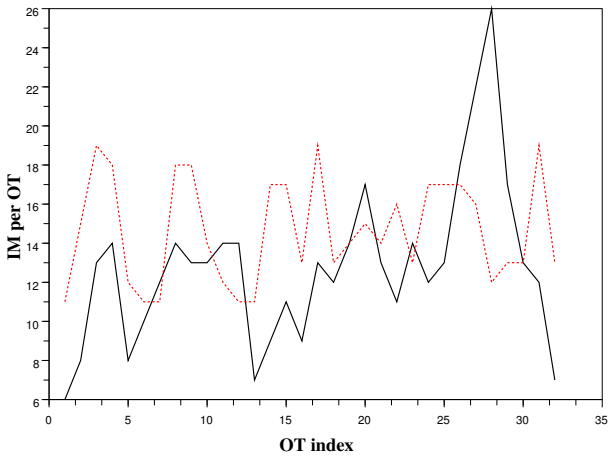


Figure 5.7: Comparison between a TSP and a BTSP solution for an instance with 128 vertices in a graph, i.e. 128 OT in the output image, ALGO1, a solution of EXP1

It is possible to optimize the TSP solution with respect to the amount of used internal memory. For example, the output tile that needs the maxima of internal memory can be divided into independent sub-tiles which will be computed sequentially and will require less internal memory. This optimization has not been tested in this work.

5.6 Conclusion

In this chapter we have presented the Scheduling. The scheduling is aimed to re-organize the output tiles computations in order to optimize the TPU behavior. It is possible to optimize the TPU behavior with respect to three costs: the internal memory used, the number of the external memory accesses and the amount of input tile changing between the computation of two successive output tiles. The minimization of each one of these three costs produces an improvement either on the power consumption of the system or on the amount of used internal memory and on the pre-fetching time. According to the criterion to be optimized we should solve a classical TSP or a BTSP.

In our work, We have used a genetic algorithm to solve the two problems. We have compared the TSP developed by using a genetic algorithm (GATSP) with respect to a a Lin-Kernighan solver and a reference TSP solver: the Concorde. Experiments show that the GATSP finds better solutions than the Lin-Kernighan in 90% of cases. The error of the Lin-Kernighan solutions with respect to the GATSP solutions is up to 35%.

The comparison of the GATSP with respect to the Concorde, shows that the Concorde finds better solutions than the GATSP in 86% of cases. The error of the GATSP solutions with respect to the Concorde solutions is up to 16%. On the other side the GATSP solver is up to 76 times faster than the Concorde. In our experiments to evaluate the results of MEXP analysis, we have used the GATSP.

Chapter 6

Computation and Memory Mapping

This chapter presents the Computation and Memory Mapping. The Computation Mapping allocates to one of the parallel pipelines the computations of a group of output tiles, while the Memory Mapping computes the amount of internal buffers needed by each independent pipeline and maps the needed input tiles into the available internal buffers. The Memory Mapping ensures that the memory accesses are conflict free and that the input tiles shared between two or more successive output tiles are copied only once into the internal memory.

Chapter contents

This chapter goes though the following contents:

- Introduction to the problem of the Computation and Memory Mapping
- Description of the Computation Mapping algorithm
- Computation of the needed amount of Internal Buffers
- Description of the Memory Mapping algorithm
- Description of the methods to reduce the area overhead due to the mapping tables

6.1 Introduction

The mapping is divided into two parts:

- a computations mapping, which allocates the computations of N_p groups of output tiles to the N_p parallel pipelines of the TPU. the parallelism level N_p is specified by the user. And the functioning of a TPU with parallel pipelines is presented in section 3.3.3 of chapter 3.
- a memory mapping, which instantiates for each one of the N_p pipelines the internal memory needed to store the input tiles. Then, it maps the needed input tiles into

the available internal buffer. This mapping ensures that the access to the internal buffers is conflict-free and that, in the same pipeline, the input tiles shared between two successive output tiles are copied only once from the external memory.

The Memory Mapping is aimed to compute the MM matrix described in chapter 3. This matrix gives the mapping between the internal buffers and the needed input tiles, for each output tile to be computed. MM is inferred from the Σ matrix, which gives the dependences between the input and output tiles. MM is computed after the scheduling, thus the lines of the Σ matrix have already been sorted.

An example of memory mapping MM is given in figure 6.1. MM is organized as follows:

- Each column index c corresponds to an output tile to be computed and
- each line index r corresponds to an available internal buffer where is stored a needed input tile;
- each element $MM(r, c)$ of the matrix gives the index of a needed input tile, used to compute the output tile $OT(c)$ and stored in the buffer r.

First of all, to compute the MM matrix, it is necessary to reckon the number M of the needed internal buffers.

M has to be determined according to the user-specified level of parallelism N_p and to the architectural model targeted. This model allows the parallelism between the computing and the pre-fetching. Thus the internal buffers have to contain a part from which to read the current input and a part in which to copy the input used in a successive time.

Each internal buffer contains a single input tile. Two internal buffers can be accessed simultaneously, one to be written from the PREFETCH module and the other to be read from the FETCH module (see chapter 3 for more details). The memory mapping ensures that the data loaded by the pre-fetching do not erase the data read by the fetching.

Let consider the MM matrix in figure 6.1. Let suppose that the corresponding TPU has to compute two successive output tiles $OT(c)$ and $OT(c+1)$. To compute the tile $OT(c)$ the TPU needs the input tiles (1 and 2) and to compute the tile $OT(c+1)$ it needs the input tile (2 and 3). The MM matrix will have two columns corresponding to two TPU tasks, and thus to two output tile computations. During the TPU task c, the TPU will pre-fetch the input data needed to compute the tile $OT(c)$. During the TPU task $c+1$, the TPU will compute the output tile $OT(c)$ and pre-fetch the input tile for $OT(c+1)$.

During the pre-fetching, it is necessary to ensure that:

- the shared input tiles (2 in the example) remain in the internal buffers and that
- the tiles needed to compute $OT(c)$ are not erased by the new tiles pre-fetched.

For a TPU with more parallel pipelines, the MM computation is modified as shown in figure 6.2: each column of a matrix corresponds to a TPU task and thus to the

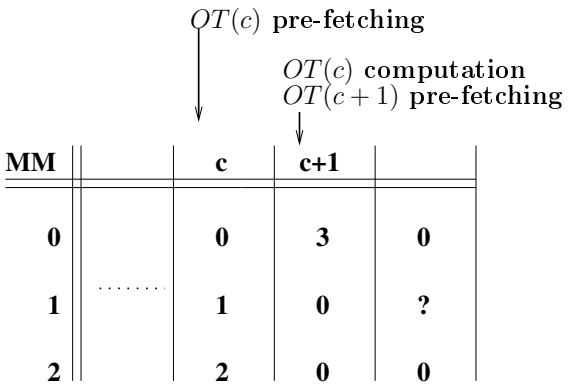


Figure 6.1: Example of a part of MM matrix for a TPU with a single pipeline ($N_p = 1$). Each column of the MM matrix corresponds to a TPU task and thus, to the computation of an output tile $OT(c)$. If an element of the matrix is 0, then the corresponding internal buffer is not used.

computation of N_p parallel output tiles. For each column of MM there are different zones corresponding to the Memory Mapping of each pipelines. For example in the figure 6.2, the internal buffers 0, 1 and 2 are reserved to the first pipeline while the internal buffers 3 and 4 are reserved to the second pipeline. The input tiles needed by both pipelines are duplicated in the different sets of the internal buffers (ex. the input tiles 1 and 3).

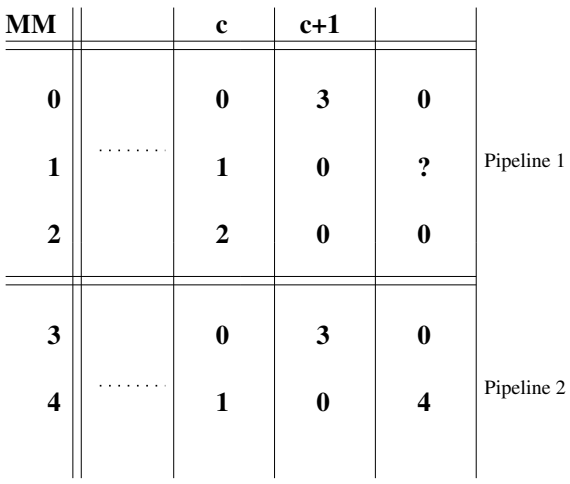


Figure 6.2: Example of a part of MM matrix for a TPU with a two pipelines ($N_p = 2$). Each column of the MM matrix corresponds to a TPU call and thus, to the computation of two parallel output tiles. The IB amount is not equal between the two parallel pipelines.

In the rest of this chapter we will describe the Computation Mapping, the formula giving the amount of the used internal buffers and the Memory Mapping.

6.2 The Computation Mapping

The target model for the TPU with parallel pipelines is detailed in section 3.3.3 of chapter 3. The pre-fetching is common for all the parallel pipelines and feeds the internal buffers, which are distributed to the different pipelines. An input tile shared between more pipelines is replicated into the internal buffers of each pipeline using it.

The problem of the Computation Mapping is to distribute the computations of the output tiles to the N_p parallel pipelines.

It is possible to exploit the optimization due to the previously performed scheduling by respecting as long as possible the computation order imposed by the vector OT_{order}.

The easiest way to perform this distribution is to assign the computation of $\lceil \frac{d}{N_p} \rceil$ output tile to the first $N_p - 1$ pipelines and the remaining tiles to the last pipeline.

According to this method, it is possible that the last pipeline remain inactive for a certain time. We can show that (for the performed analysis) the inactivity time (T_{inact}) of the last pipeline is negligible with respect to the Temporal Performance (TP) needed to perform the whole algorithm. ($\frac{T_{inact}}{TP}\% \approx \frac{N_p(N_p-1)}{d}$ is inferior to 10% for $d = 120$ and $N_p \leq 4$).

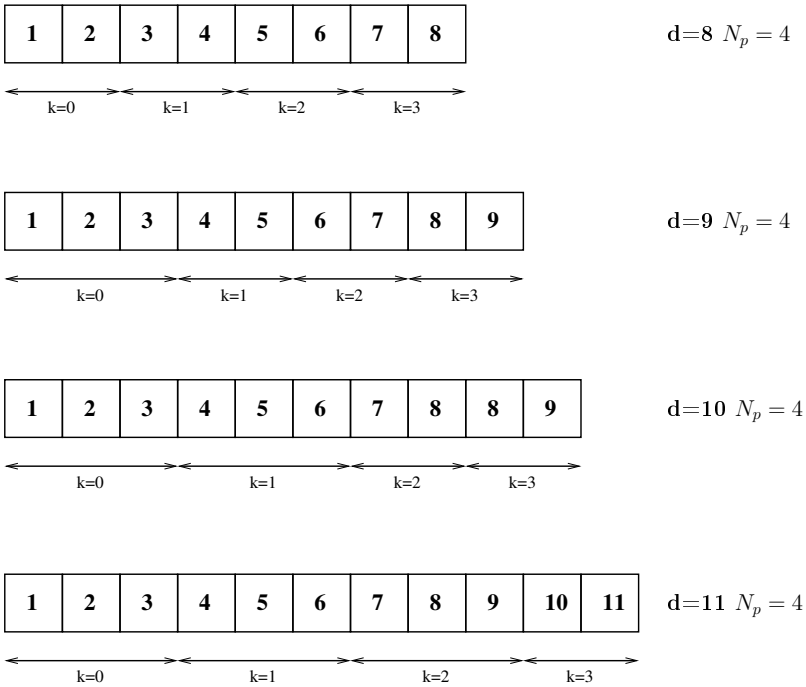


Figure 6.3: Example of Computation Mapping for $N_p = 4$ and different values of d.

However, another way to redistribute the output tile computations is to assign the computation of $\lceil \frac{d}{N_p} \rceil$ output tiles to the first $d\%N_p$ pipelines and $\lfloor \frac{d}{N_p} \rfloor$ output tiles to remaining pipelines.

This Computation Mapping is presented in figure 6.3 and ensures that all the pipelines

are inactive for 1 output tile computation at most.

6.3 The needed amount of internal buffers

Let consider a TPU with a single pipeline ($N_p = 1$), the first step of the Memory Mapping is to compute the amount of internal buffers needed to execute the whole image processing on the TPU.

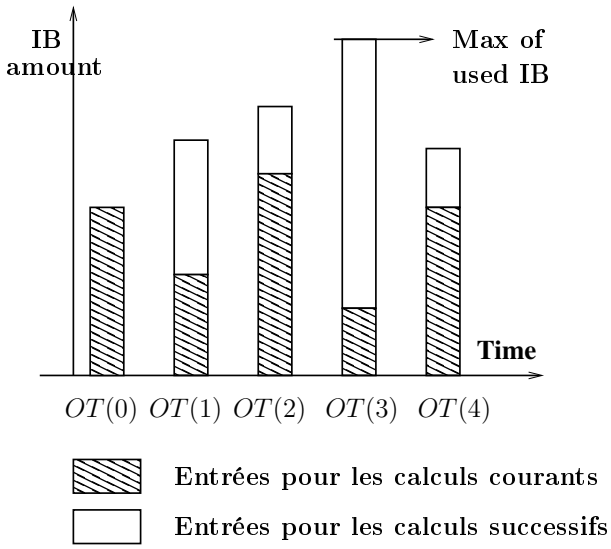


Figure 6.4

The amount of needed internal buffers, has to count two parts: one containing the Current Input Tiles (CIT) and one in which to copy the Successive Input Tiles (SIT). As shown in figure 6.4, the non-affinity of the array references in the target application can make the CIT and the SIT parts increase or decrease all the application execution long.

For this reason the formula computing the needed amount of internal buffers (IB) is:

$$IB = max_{i=1}^{d}(SIT(i) + CIT(i)) \tag{6.1}$$

Where $SIT(i) + CIT(i) = \omega_2(i, i+1)$ and $\omega_2(i, i+1)$ is defined by the equation (5.4) of chapter 5.

In a TPU with more parallel pipelines ($N_p > 1$), each pipeline has a set of internal buffers in which storing input data.

The amount $IB[k]$ of internal buffer associated to a pipeline k is computed by modifying the formula 6.1 as follows:

$$IB[k] = max_{i=LB}^{UB}(SIT(i) + CIT(i)) \tag{6.2}$$

where LB and UB are respectively the lower bound and the upper bound of the set of output tiles to be computed by the pipeline k. Their value depends on the Computation Mapping.

For example, for the first presented method of Computation Mapping

$$LB = k\left\lceil \frac{d}{N_p} \right\rceil \text{ and } UB = (k+1)\left\lceil \frac{d}{N_p} \right\rceil$$

for the $N_p - 1$ first considered pipelines and

$$LB = (N_p - 1)\left\lceil \frac{d}{N_p} \right\rceil \text{ and } UB = d$$

for the last considered pipeline.

For the Computation Mapping presented in figure 6.3 ,

$$LB = k\left\lfloor \frac{d}{N_p} \right\rfloor + d_{k_2} \text{ and } UB = (k+1)\left\lfloor \frac{d}{N_p} \right\rfloor + d_{k_2} + d_{k_1}$$

with $k \in [0, N_p[$, $d_{k_1} = \begin{cases} 1 & \text{if } k < d\%N_p \\ 0 & \text{otherwise} \end{cases}$ and $d_{k_2} = \begin{cases} k & \text{if } k < d\%N_p \\ 0 & \text{otherwise} \end{cases}$.

. The formula 6.2 adapts the amount of internal buffers to the specific set of output tiles that each pipeline has to compute.

6.4 The Memory Mapping

Suppose that the TPU is configured to execute the image processing according to a solution associated with a couple of input/output tiles (IT, OT) and, thus, associated to:

- a partitioning of the input space in s input tiles $IT(j)$ with $j \in [1, s]$
- a partitioning of the output space in d useful output tiles $OT(i)$ with $i \in [1, d]$
- a Super-Tiling matrix $\Sigma^{d \times s}$, which gives the dependences between the input and output tiles and whose elements are noted $\sigma_{i,j}$;
- a scheduling vector OT_{order} of d elements, which gives the order according to which the TPU has to compute the output tiles. The lines of the Σ matrix are re-ordered according to this scheduling.

Given the amount IB of internal buffers needed, the MM matrix is inferred by the Σ-matrix as presented by the code in figure 6.5.

In this code the available buffers are stored in a stack. When we assign a buffer to the storage of a input tile, we extract it from the stack. When a IT is no longer used we re-introduce the corresponding free buffer in the stack.

For each parallel pipeline $(0 \leq k < N_p)$ we consider the group of output tiles that the pipeline has to compute $(LB \leq i < UB)$. For each output tile to be computed, we evaluate which input tiles have to be copied into the internal buffers.

We consider two cases:

```
— data structure —
typedef struct IB{ // Element of the stack storing the available internal buffers
  struct IB next;
  int b;
}IB;

typedef struct PP{ // Structure storing the information about a pipeline mapping
  struct PP next;
  int Mapp[SMAX]; // Mapping between the IT and the IB
  int Mapp_to_preserve[SMAX]; // Flag preserving a buffer mapping
  IB top_of_the_stack; // Pointer to the stack of available buffers
}PP;
————————————————-
....
PP pipe[N_proc];
....
// each pipe[k].IB stack contains all the available buffers for the corresponding pipeline ....
  for(k=0;k<N_pipe;k++){
    ....
    for(i=0;i<UB;i++){
      new_i=LB+i;
      A: for(j=0;j<solution.s;j++){
        // The input tile IT(j) has not to be copied
        if(solution.Sigma[new_i][j]==0 && pipe[k].Mapp[j]!=-1&& pipe[k].Mapp_to_preserve[j]==-1){
          pipe[k].top_of_the_stack=Push( pipe[k].Mapp[j], pipe[k].top_of_the_stack);
          pipe[k].Mapp[j]=-1;
        }
      }
      B: for(j=0;j<solution.s;j++){
        // The input tile IT(j) has to be copied
        if(solution.Sigma[new_i][j]==1 && pipe[k].Mapp[j]==-1){
          pipe[k].Mapp[j]=Pop(& pipe[k].top_of_the_stack);
          solution.MM[ pipe[k].Mapp[j]][i]=j;
        }
      }
      C: for(j=0;j<solution.s;j++){
        if(solution.Sigma[new_i][j]==1)
          pipe[k].Mapp_to_preserve[j]=1;
        else
          pipe[k].Mapp_to_preserve[j]=-1;
      }
    }
    free_IB( pipe[k].top_of_the_stack);
  }
```

Figure 6.5: C-code to compute the MM matrix

- We first free the internal buffers from the tiles that have no longer to be used

(loop nest **A** in the code of figure 6.5). In order to do that, we evaluate for all the input tiles, if a given input tile $IT(j)$ has not to remain into the internal buffers ($solution.Sigma[new_i][j] == 0$). Then, we evaluate if it has been previously allocated into an internal buffer (we use the value $pipe[k].Mapp[j]$ which gives the buffer to which the input tile $IT(j)$ has been allocated to). If it has been previously allocated and it has not to be preserved ($pipe[k].Mapp_to_preserve[j] == -1$), then we cancel the corresponding buffer allocation and push the free buffer into the stack of available buffers.

The input tile to be preserved at the task i are those allocated during the task $i-1$ because they have to be used for the current computation.

- In a second moment we map the new needed input tiles (loop nest **B** in the code of figure 6.5). In order to do that, we evaluate if the input tile $IT(j)$ has to be copied from the external memory ($solution.Sigma[new_i][j] == 1$) and it is not already mapped into an internal buffer ($pipe[k].Mapp[j] == -1$). Then we extract the first available buffer from the stack of the considered pipe (i.e. POP on $pipe[k].top\_of\_the\_stack$) and we map the input tile in it.

Example Figures 6.7 and 6.8 give an example of the MM matrix computation.

Figure 6.7(a) gives the configuration of the I/O tilings with the dependences between the input and output tiles expressed by the corresponding Sigma matrix.

Figure 6.7(b) gives the Sigma matrix whose lines have already been re-ordered by the scheduling algorithm. Figure 6.7(c) gives the MM matrix and figure 6.7(d) gives the evolution in time of the internal buffer contents. The amount of internal buffers used is $IB = 3$.

Figure 6.8 gives the steps of the Memory Mapping to compute the matrix MM for a TPU with a single pipeline ($N_p = 1$).

The presented algorithm considers only the last mapped input tiles and does not check the current contents of the internal buffers. Thus it may occur that some input tiles already in the internal buffers may be mapped again in a different buffer. For the example of figure 6.1, during the memory mapping of the input for the last output tile ($OT_{index} = 1$), the input tile 1, which is already in the buffer 2, is mapped again in the buffer 0. This causes an increase of the external memory access and the power consumption that can be avoided.

In order to improve the Memory Mapping algorithm, we have to modify the loop (B) of the code in figure 6.5 as follows:

- We add a vector $Current\_IB\_Content$ giving the current content of the Internal Buffers.

- For each input tile to be mapped, we check if the vector $Current\_IB\_Content$ contains it.

- If the input tile is in the vector $Current\_IB\_Content$, then it is not mapped again and the buffer which currently contains it is extracted from the stack of available buffers. The function that extracts the internal buffers scans the content of the stack until the wanted buffer is not found.

These modifications are presented in figure 6.6.

```
// Current_IB_Content gives the current content of the Internal Buffers
// B=∑_k IB[k] where IB[k] is defined by the equation (6.2 )
int Current_IB_Content[B];
.......
      B: for(j=0;j<solution.s;j++){
        // The input tile IT(j) has to be copied
        if(solution.Sigma[new_i][j]==1 && pipe[k].Mapp[j]==-1){
          Copy_IT=-1;
          for(u=LB;u<UB;u++){
            if(Current_IB_Content[u]==j)
              Copy_IT=u;
          }
          if(Copy_IT==-1){
            pipe[k].Mapp[j]=Pop(& pipe[k].top_of_the_stack);
            solution.MM[ pipe[k].Mapp[j]][i]=j;
            Current_IB_Content[u]==pipe[k].Mapp[j];
          }else{
            extract(Copy_IT,pipe[k].top_of_the_stack);
          }
        }
      }
```

Figure 6.6: Modifications to the C-code to compute the MM matrix are in bold.

6.4.1 The lifetime of the input tiles

Some input tiles of a solution can be used to compute several output tiles during the target algorithm execution. If the output tiles are not successive, the corresponding input tiles can be:

- Erased from the internal memory in order to re-use their corresponding internal buffers. This reduces the total amount of used internal buffers but requires that the input tiles are copied again from the external memory.
- Kept into the internal buffers for all the duration of their lifetime. This reduces the number of accesses to the external memory, but requires the instantiation of more internal memory.

In our analysis we find a trade-off between these two hardware solutions by performing an output tile re-scheduling. This re-scheduling minimizes the amount of input tiles changing between the computation of two successive output tiles (cost ω_3 defined in equation (5.5) of chapter 5). Thus it reduces the lifetime of the major part of input tiles.

6.5 How to reduce the area overhead due to the usage of MM

The TPU needs to read the contents of the MM matrix in order to know the Memory Mapping. But, the MM matrix has a size of $d \times IB$ for $N_p = 1$ and a size of $\left\lceil \frac{d}{N_p} \right\rceil \times \sum_{k=0}^{N_p-1} IB(k)$ for $N_p > 1$, where d is the number of output tiles to be computed and IB is the amount of internal buffer used. In both these cases the area overhead due to the usage of the MM matrix can be reduced, in order to keep low the size of the internal memory used to configure the TPU.

The first improvement can be obtained by copying a MM column a time from the external memory.

The second improvement is the following: let consider the ratio R_{IB} between the maximum amount of internal buffers changing with respect to the total amount of internal buffer used:

$$R_{IB}[k] = \frac{\max_i \omega_1(i, i+1)}{IB[k]}$$

where ω_1 is defined in equation (5.3) of chapter 5, $IB[k]$ defined in equation 6.2 and with $0 \leq R_{IB} \leq 1$.

We can reduce the area overhead due to the usage of the MM matrix by splitting it into 2 matrices: one giving the indices of the input tiles to be copied from the external memory and one giving the indices of the internal buffers where to copy the needed input tiles.

An example of memory overhead reduction is given by table 6.1. This method is useful only if $R_{IB} < \frac{1}{2}$.

$$\begin{array}{c} \text{MM matrix} \\ \begin{pmatrix} 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 3 & 0 \end{pmatrix} \end{array} \Longrightarrow \begin{array}{l} \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \text{ IT indices} \\ \\ \begin{pmatrix} 1 & 0 & 3 & 2 \end{pmatrix} \text{ IB indices} \end{array}$$

Table 6.1: Example of MM divided into two tables

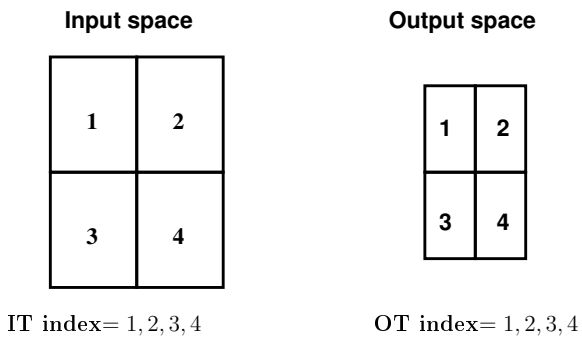
6.6 Conclusion

This chapter described the Computation and Memory mappings. These mappings are computed for each one of the MEXP analyzed solutions.

The Computation Mapping divides the output tiles in N_p groups and allocates the computation of a group to one of the N_p available pipelines. The tiles in a group are computed sequentially by the same pipeline, while the groups are computed in parallel by different pipelines. As described in section 3.3.3 of chapter 3, the prefetching of the

data for all the pipelines is performed sequentially and all the pipelines are synchronized at the end of the TPU task computing the N_p parallel output tiles.

The Memory Mapping algorithm is aimed to map the input tiles into the available internal buffers. The first step of the Memory Mapping is to reckon the amount of needed internal buffers. Then, for each output tile, the Memory Mapping computes the position of the input tiles into the internal buffers. The Memory Mapping ensures that the input tiles shared between two successive output tiles of a same pipeline are copied only once from the external memory. It also ensures that the prefetching and the fetching in the TPU can access the internal buffers without a conflict. Finally by scanning the current internal buffer content the mapping can further avoid useless accesses to the external memory.



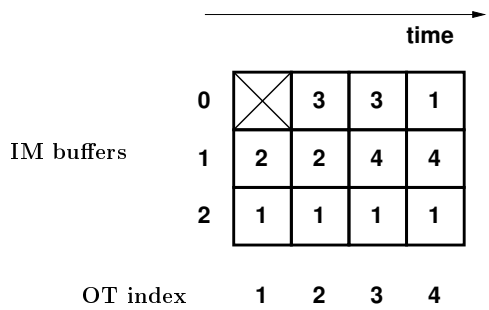
(a) I/O tilings

$$\Sigma = \underbrace{\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}}_{\text{IT index}} \begin{matrix} 2 \\ 3 \\ 4 \\ 1 \end{matrix} \Bigg\} \text{OT index according to the } OT_{order}$$

(b) Super-Tiling matrix corresponding to the I/O tilings

$$MM = \underbrace{\begin{pmatrix} 0 & 3 & 0 & \boxed{1} \\ 2 & 0 & 4 & 0 \\ \boxed{1} & 0 & 0 & 0 \end{pmatrix}}_{\text{OT index}} \begin{matrix} 0 \\ 1 \\ 2 \end{matrix} \Bigg\} \text{IM locations}$$

(c) Memory Mapping matrix. The input tile 1 is mapped twice. Section 6.4 gives the needed MM modifications to avoid this problem.



(d) Internal Memory configuration during the computation of all the output tiles

Figure 6.7: Example which compute an MM matrix from a Σ matrix.

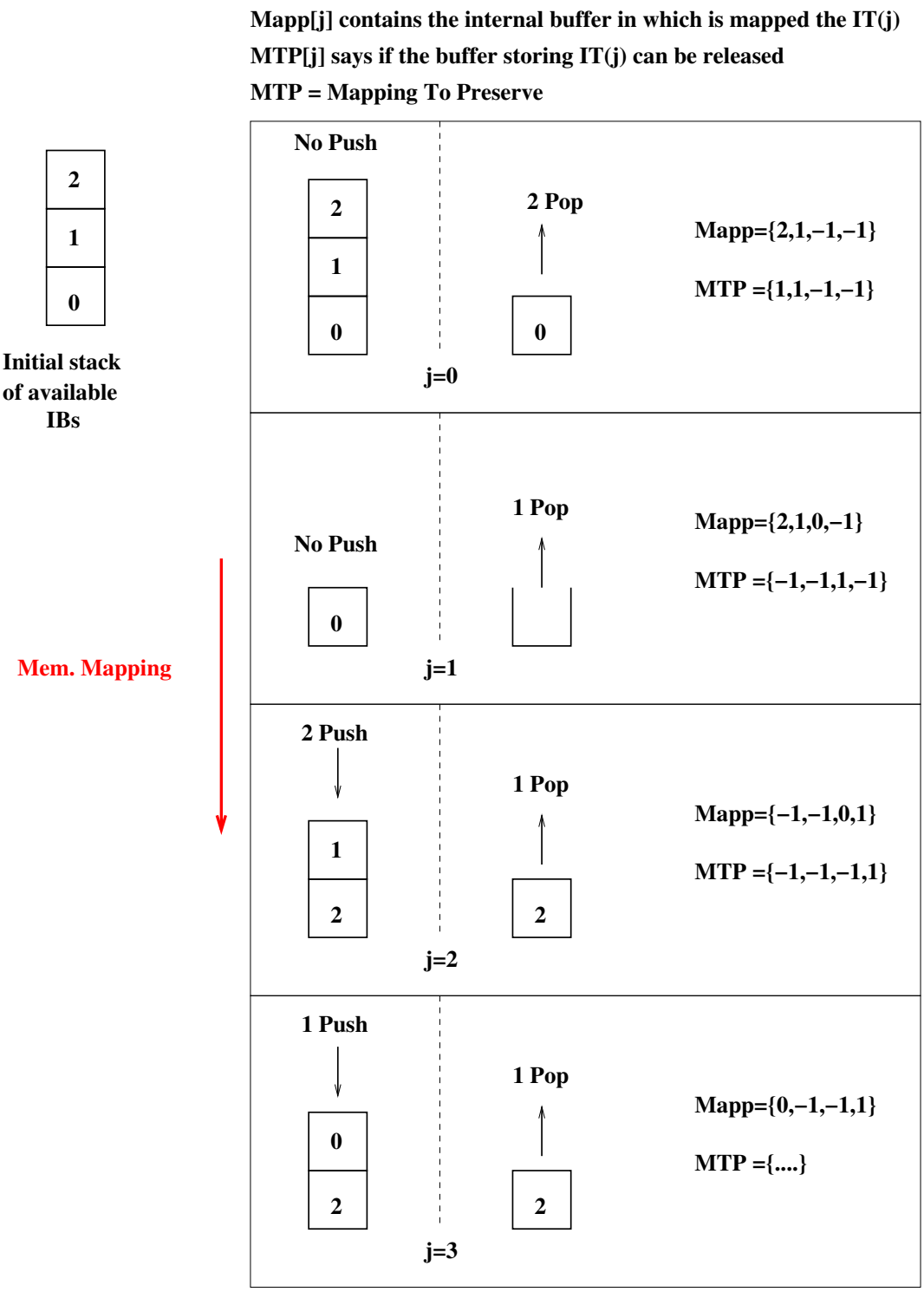


Figure 6.8: Example of the Memory Mapping for the I/O tiling presented in figure 6.7

Chapter 7

Design Space Exploration: System storage requirement and performance estimation

This chapter presents the Design Space Exploration method. This method is based on the estimation of two metrics used to evaluate a given solution. The solutions are classified with respect to these two metrics and the pareto solutions of the set are chosen at the end of the DSE. We apply a validity filter on the solution space, in order to reduce it. We also reduce the number of comparisons between the possible solutions by using a binary tree to classify them.

Chapter contents

This chapter goes through the following contents:

- Introduction to the Design Space Exploration
- Description of the algorithm used to perform the DSE
- Description of the selection criteria: the temporal performance of the TPU and the amount of internal memory used

7.1 The Design Space Exploration

According to the work [99], the MEXP Design Space Exploration (DSE) is performed through a validity filter and a quality filter. The **validity filter** eliminates all the solutions which do not match the user constraints. The **quality filter** finds the pareto solutions in the analyzed set. The quality filter uses two metrics to classify the solutions and relates them to each other trough two criteria of dominance and equivalence. The used metrics and criteria are defined as follows:

1. Let s_i, with $i \in [1, N_s]$, be a possible couples of I/O tiling which correspond to a possible hardware solution. We define **two metrics** $f_{IM}(s_i)$ and $f_{NC}(s_i)$, representing respectively the amount of Internal Memory used to implement the solution s_i and the number of cycles to execute the algorithm with an hardware solution s_i.

2. Given two solutions s_i and s_j with $i, j \in [1, N_s]$, we define the following **dominance criterion**: the solution s_i dominates the solution s_j if it is more efficient than s_j with respect to the two metrics $f_{IM}(s_i)$ and $f_{NC}(s_i)$, i.e. if

$$(f_{IM}(s_i) < f_{IM}(s_j)) \wedge (f_{NC}(s_i) < f_{NC}(s_j))$$

3. Given two solutions s_i and s_j with $i, j \in [1, N_s]$, we define the following **equivalence criterion**: the solutions s_i and s_j are equivalent if s_i is more efficient than s_j with respect to one of the two metrics and s_j is more efficient than s_i with respect to the other metric, i.e. if

$$(f_{IM}(s_i) < f_{IM}(s_j) \wedge f_{NC}(s_i) > f_{NC}(s_j)) \vee (f_{IM}(s_i) > f_{IM}(s_j) \wedge f_{NC}(s_i) < f_{NC}(s_j))$$

4. A **pareto solution** dominates or is at least equivalent to all the other solutions of a set. All the pareto solutions are equivalent to each other.

Figure 7.1 shows three pareto solutions s_1, s_2 and s_3 which are equivalent to each other and dominant with respect to the solution s_4. This figure also shows how the design space is tailored by two user's constraints (IM_{MAX} and NC_{MAX}).

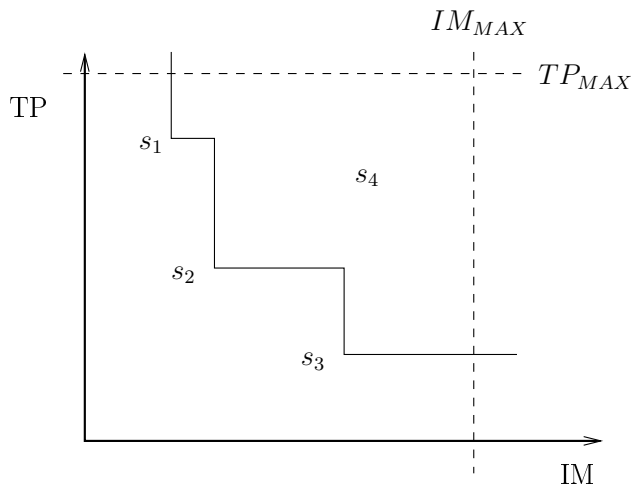
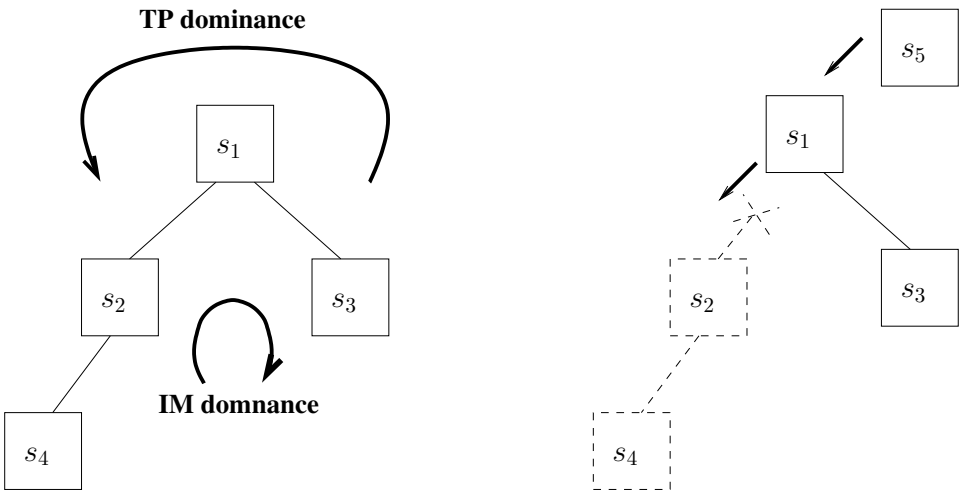


Figure 7.1: A Design Space is tailored by the user's constraints of IM_{MAX} and NC_{MAX}. The solutions s_1, s_2 and s_3 are pareto, they are equivalent to each other, while the solution s_4 is dominated by all the pareto solutions.

The solutions of the Design Space are sorted by using a binary tree, as shown in figure 7.2. A solution which dominates another becomes its leaf: the right if the dominance criterion is the internal memory amount and left if the dominance criterion is the number

of cycles. In figure 7.2 the dominance are expressed as follows: $f_{IM}(s_3) < f_{IM}(s_1) < f_{IM}(s_2)$ and $f_{NC}(s_2) < f_{NC}(s_1) < f_{IM}(s_3)$. If a new solution has to be inserted in the tree it will be compared only with the left or the right subpart of the tree and it could be inserted as a new equivalent solution or substitute another solution or substitute a whole part of the solution tree (as shown in figure 7.2(b)).



(a) The Design Space solution tree. (b) A new solution inserted

Figure 7.2: Example of a DSE solution tree with the dominance criteria and the insertion of a new solution.

7.2 The selection criteria

The two selection criteria to run the DSE are:

- The amount of internal memory used and
- The number of cycles that a TPU needs to execute the target algorithm. This criterion is also called the temporal performance of the TPU.

In the rest of this paragraph we present the formula used to estimate the value of these two criteria for a given algorithm and a given solution.

7.2.1 The amount of internal memory used

In a hardware design the area occupancy is due to both the sequential and the logic operators. In our estimation we do not count the area occupancy due to the logic operators and we estimate the area due to the internal memories without taking into account neither the local registers that the HLS could instantiate nor the area needed to store the user-defined LUTs.

A TPU uses some mapping tables to perform the prefetching and the fetching and uses internal buffers to store the input tile. Thus, the amount of internal memory used by the TPU model counts four known contributions:

- the internal buffers IB,
- the pre-fetching matrix MM which can be replaced by 2 smaller matrices as presented in paragraph 6.5
- the table ensuring a direct read access to the internal buffers (i.e. IDX) and
- the TPU configuration register(s) (i.e. OT), which give(s) the number(s) of the current output tile(s) to be computed.

The amount (in term of bits) of Internal Memory (IM) needed to store these contributions is computed as follows:

The Internal Memory to store IB (IM_{IB})

Let the TPU have N_p pipelines indexed by $k \in [0, N_p - 1]$, with $IB[k]$ the number of internal buffers used per each pipeline, V_{IT} the volume of an input tile[a] and m the memory word bit-width, then the amount (in number of bit) of internal buffers used is:

$$IM_{IB} = m * V_{IT} * \sum_{k=0}^{N_p-1} IB[k]$$

The Internal Memory to store MM (IM_{MM})

As presented in paragraph 6.5 of chapter 6, the MM size depends on the parallelism level, on the amount of output tiles to be computed and on the amount of IB used per each pipeline.

The amount of IM needed to store the MM matrix can be reduced by copying into the TPU a column of MM a time and by splitting MM in 2 smaller matrices. The splitting is performed with respect to the ratio $R_{IB}[k]$ defined in paragraph 6.5.

After these modifications the amount of internal memory used to store the MM matrix is:

$$IM_{MM} = 2 * \lceil lg_2(s) \rceil * (\sum_{k=0}^{N_p-1} R_{IB}[k] * IB[k])$$

where the factor 2 depends on the splitting of the MM matrix into two smaller matrices; $\lceil lg_2(s) \rceil$ is the number of bits needed to encode an input tile and $\sum_{k=0}^{N_p-1} R_{IB}[k] * IB[k]$ gives the number of lines of one of the two matrices in which MM has been split.

[a] V_{IT} also corresponds to the size of an internal buffer

The Internal Memory to store IDX (IM_{IDX}) table

The amount in number of bits of IM used to realize the table **IDX** is:

$$IM_{IDX} = s * \left\lceil lg_2 \left(\sum_{k=0}^{N_p-1} IB[k] \right) \right\rceil$$

where s is the number of input tiles and $\left\lceil lg_2 \left(\sum_{k=0}^{N_p-1} IB[k] \right) \right\rceil$ is the number of bits needed to encode an internal buffer index.

The Internal Memory to store OT register (IM_{OT})

The number of bits needed to encode the output tiles (and thus needed by the register OT) is

$$IM_{OT} = \lceil lg_2(d) \rceil * N_p$$

.

Finally, the amount of internal memory used by the TPU model is (at least[b])

$$IM = IM_{IB} + IM_{MM} + IM_{IDX} + IM_{OT}$$

A couple of input/output tilings (IT, OT) is retained as a possible solution if it respects the user's constraint of maximum internal memory possible:

$$IM \leq IM_{MAX} \tag{7.1}$$

7.2.2 The Temporal Performance of the TPU

The aim of MEXP is to partition both the input data space and the computation space. The data partitioning allows to store only parts of the input data space into the internal buffers. The computation partitioning allows to improve the parallelism level of the target hardware.

The Temporal Performance (TP) of the target hardware depends on:

- the chosen couple of input and output tilings,
- the external memory access,
- the target level of parallelism.

[b] we do not take into account neither the local registers that the HLS tool may instantiate to perform the computations nor the user-defined LUT memory requirements.

In our hardware model the **external memory** is a single port RAM **accessed according to a burst mode**. This means that the needed data can be read one after the other by starting from the required address to the end of the burst (see section 1.3.1 in chapter 1.1 for more details). The burst length depends on the particular architecture of the memory.

In our model, we suppose that a burst contains a whole input tile and that a memory word contains m elementary data (i.e. pixels).

Due to the burst mode, there is a latency of L cycles between the reading of two successive input tiles.

As described in section 3.3.3 of chapter 3, during a TPU task, two sub-tasks are executed in parallel: the prefetching and the computing.

Due to the fact that the external memory is a single port RAM, in a TPU with N_p parallel pipelines there is a single prefetching sub-task which sequentially copies the needed input tiles for all the parallel pipelines. On the other hand, the TPU contains N_p independent sets of internal buffers, that can be accessed in parallel. Thus the computing sub-tasks can compute N_p output tiles in a real parallelism. At the and of each TPU task all the pipeline and the prefetching are synchronized. Figure 3.9 in chapter 3 gives the typical time-line of a TPU with 2 parallel pipelines.

We define the following four values:

- TP: the temporal performance of the TPU.
- TP_i^p: the time to pre-fetch the input tiles.
- TP_i^c: the time to compute the output of a TPU task. This can be a single output tile when $N_p = 1$ or N_p parallel output tiles when $N_p > 1$.
- TP_i: the time to execute a TPU task, i.e. to execute in parallel the prefetching and the computation.

The previously defined values are computed (in number of cycles) as follows: The **Temporal Performance of a TPU** is the sum of times (TP_i) to execute $\left\lceil \frac{d}{N_p} \right\rceil$ TPU tasks:

$$TP = \sum_{i=1}^{\left\lceil \frac{d}{N_p} \right\rceil} TP_i \text{ cycles} \tag{7.2}$$

The **time to execute a TPU task** (TP_i) (i.e. to execute in parallel the prefetching and the computing) is the maximum between the time to prefetch and the time to compute the output of a TPU task:

$$TP_i = \max\{TP_i^p, TP_i^c\} \text{ cycles} \tag{7.3}$$

For each output tile, the TPU pre-fetches ω_1 input tiles from an external memory (with ω_1 defined in 5.3 of chapter 5). The external memory contains m elementary data (i.e. pixel) per memory word and has a latency of L cycles.

The **time to prefetch** is defined as

$$TP_i^p = \left(L + \frac{V^{z_{in}}}{m}\right) * \omega_1(i, i+1) \tag{7.4}$$

where $V^{z_{in}}$ is the volume of an input tile.

To define the **time to compute** the output of a TPU task, we have to consider the following observations:

- A TPU task consists in computing 1 or N_p output tiles in parallel, according to the user-specified level of parallelism. The TPU contains 4 main loop nests (REQ, Prefetch, FETCH and CALC). Thanks to the MEXP mapping and the usage of streams the 4 considered loop nests (REQ, Prefetch, FETCH and CALC) can be executed in parallel. Furthermore, thanks to the usage of N_p independent sets of internal buffers and logic operators (each one affected to a pipeline), the N_p pipelines can access the input and compute the output in a true parallelism.

- Except for the loop nest performing the prefetching, the other loop nests have the same number of iterations, and this number is equal to the number of data contained in an output tile. The loop cores of the three loops (REQ, FETCH and CALC) perform the memory accesses and the computations concerning a single output datum.

- As explained in paragraph 1.2.5.2 of chapter 1.2, the iterations of a loop nest can be pipelined on the same hardware. In order to avoid a conflict on the memory accesses, the starting times of successive iterations have to be delayed with respect to each other, as shown in figure 7.3.

  From figure 7.3, we can deduce that:

  1. The delay between the beginning of two successive iterations (also called Initiation Interval (II)) is the maximum between the time to read the input (e) and the time to write the output (w).
  2. The time to execute the whole loop nest is $TP = II*$(Nbr. of loop iterations)$+ T$, with T depending on the time to perform the operations of the loop core. When the number of iteration increases, T becomes negligible.

Let suppose that, in the target TPU, the computation of an output datum needs (at most) e input data. Each pipeline has to access its own set of internal buffers at most e times. After computing the output, the pipeline has to copy the result in a temporary output buffer. The temporary output buffer can be written only by a pipeline a time, thus N_p pipelines will take N_p cycles to store their outputs. This time-line is described in figure 7.4.

We can conclude that, in a TPU, the II of the loop nests (REQ, FETCH and CALC) is :

$$II = \max(e, N_p)$$

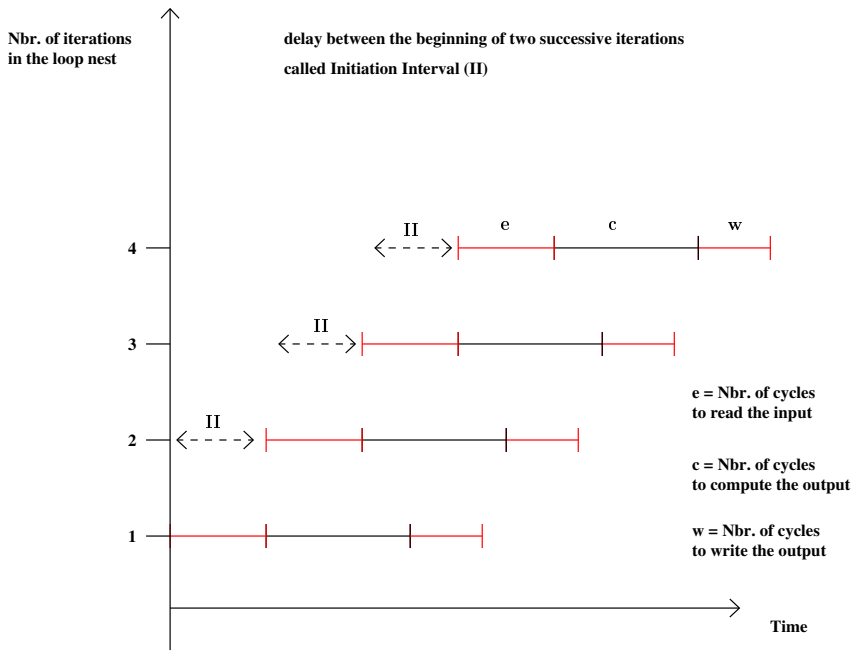


Figure 7.3: Example of the parallelization between the iterations of a loop nest. The beginnings of the iterations are delayed one with respect to the other.

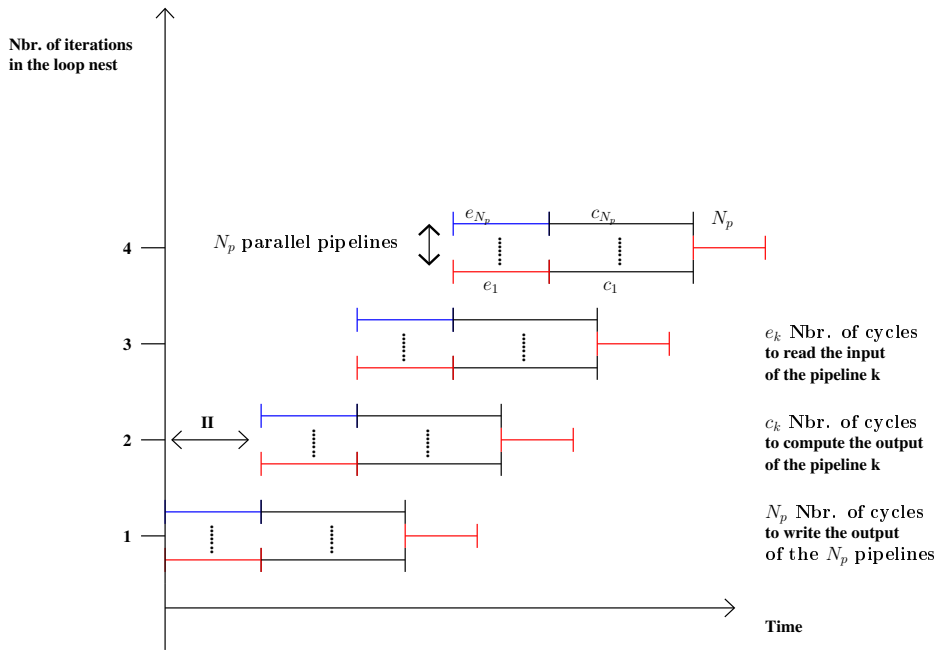


Figure 7.4: Example of the parallelization between the iterations of a loop nest with N_p parallel pipelines.

As the number of iterations of the considered loop nests is equal to the output tile volume $V^{z_{out}}$, **the time to compute** the output of a TPU task is:

$$TP_i^c = V^{z_{out}} * II$$

Once computed the temporal performance of the chosen solution, the corresponding couple of input/output tilings (IT, OT) is retained if it respects the user's constraint of maximum temporal performance:

$$TP \leq TP_{MAX} \tag{7.5}$$

7.2.2.1 The time to initialize a TPU task

The initialization of a TPU task, corresponds to load from the external memory the corresponding column of the Memory Mapping (MM) matrix.

As the MM matrix is read from the external memory and the external memory is a single port memory, the initialization has to be performed sequentially with respect to the input tiles prefetching.

In order to parallelize the initialization and the computation of the output of a TPU task, we use a double buffering mechanism on the Internal Memory storing the MM column. The double buffering mechanism is described in paragraph 1.2.4 of chapter 1.2.

To take into account the initialization of a TPU task we have to modify the formulae giving the TPU Internal Memory amount and Temporal Performance, as follows: The used amount of Internal Memory becomes

$$IM = IM_{IB} + \mathbf{2} * IM_{MM} + IM_{IDX} + IM_{OT}$$

where the factor **2** is due to the double buffering.

The time to execute a TPU task becomes

$$TP_i = \max\left\{TP_i^p + TP_i^{INIT}, TP_i^c\right\} \text{ cycles}$$

where TP_i^{INIT} is the time to initialize a TPU task and it is:

$$TP^{INIT} = L + 2 * \sum_{k=0}^{N_p-1} \{R_{IB}[k] * IB[k]\}$$

where the factor 2 depends on the splitting of the MM matrix into two smaller matrices.

7.2.2.2 The times neglected during the computation of TP

During the computation of the temporal performance of a TPU we neglect two contributions:

- The time due to the dependences between the loop nests REQ, FETCH and CALC.
- The time to copy the produced output tiles back into the external memory.

The blocks of the pipeline REQ, FETCH and CALC are not execute in a true parallelism. In fact the block CALC depends on FETCH, which depends on REQ. But the delay to the dependences is negligible with respect to the computation of a whole output tile.

We do not consider the time to copy the output tiles back into the external memory for two reasons:

1. We suppose that there is no latency to write into the external memory. Thus, the time to copy all the produced output tiles back to the external memory is the same for all the analyzed solutions.

2. In a hardware model containing a cascade of TPUs, the output tiles of a TPU are directly copied into the internal buffer of the successive TPU, without using a temporary output buffer nor the external memory.

7.3 Conclusion

This section presented the Design Space Exploration method. It described the data structure used to classify the solutions and the method to reduce the space of possible solutions by tailoring it with the user constraints.

The DSE is performed, through a validity filter which reduces the solutions space, and a quality filter, which classify solutions as dominant or equivalent.

The DSE is run with respect to two selection criteria: the system temporal performance and the amount of internal memory used. These criteria are estimated and their formulae are given.

Part III

Applications

Chapter 8

Tools used for the results analysis

This part describes the applications analyzed with MEXP and synthesized thanks to a commercial HLS tool. These applications are taken from the digital retina model developed at the GIPSA-lab [4].

The GIPSA-lab retina model is based on biological observations and reproduces a human retina functioning. A Human retina counts three bodies of neuronal cells (photo-receptors, bipolar and ganglion cells) separated by two synapses. The synapses are chemical junctions through which the neuronal cells communicate.

Among the neuronal cells we have only considered the photo-receptors. They are distributed on the retina according to a pseudo logarithmic law in order to ensure a vision roughly spherical, blurred on the borders and neat on the center.

The photo-receptors distribution and communication is emulated by two transformations: a spatial-variant low pass and a LOG sampling. The spatial-variant low pass has a filtering coefficient variable on the pixel eccentricity and emulates the blurred vision on the borders scene. The LOG sampling emulates the geometrical distribution of the photo-receptors on the retina.

After that the photo-receptors have caught the scene, they transfer it to the visual cortex through the optic nerve. The visual cortex is the brain part in charge of elaborating the visual information. The projection of the scene on the visual cortex is modeled by a polar projection [104].

The applications chosen from the retina model are:

- The LOG sampling
- The polar transform
- The pyramidal LOG sampling

The LOG sampling consists in sampling an input image according to a pseudo logarithmic law.

The polar transform is a projection of the input image which changes the coordinates of the input pixels from the Cartesian to the polar ones.

The pyramidal LOG sampling is a mip-map based method, which reproduces the behavior of a spatial-variant low-pass followed by a log sampling.

These applications have been analyzed with MEXP and synthesized thanks to a commercial HLS tool.

All the HLS experiments has been run for a clock rate of 100 Mhz and for a 45nm technology.

For each application we give the MEXP exploration results, the HLS results and we evaluate the solutions with respect to the temporal performance speed up and the area overhead due to the optimizations and to the parallelism.

8.1 Metrics for the results analysis

For each one of the chosen applications we have performed a MEXP exploration for three different sizes of input image:

- SQCIF, VGA and HDTV for the LOG sampling and the Pyramidal LOG sampling
- input images containing $128x128$, $300x300$ and $600x3600$ pixels for the polar transform

Each exploration has been run for three different values of external memory latency of 30, 60 and 100 cycles and for a mapping on 1, 2 and 4 parallel pipelines. Thus each exploration evaluates hundreds of solutions.

The time to execute a MEXP exploration varies between 1 sec for a SQCIF and 10 sec for a HDTV, but the explorations for different memory latency and different number of parallel pipelines only execute a short sub-part of the MEXP flow.

For each image size we will give

- The results of the MEXP exploration.
- The area and temporal performance measured after the HLS and their comparisons with the MEXP estimations.
- The temporal performance of the generated RTL models corresponding to the MEXP optimized solutions. They will be compared with each other and with a solution that does not use an internal memory.

The temporal performance of the solutions will be evaluated with respect to the following metrics:

- **The Speed Up** of the temporal performance due to the MEXP optimizations. It will be called “MEXP SU" and will be measured as:

$$MEXP\ SU(s.i, N_p) = \frac{TP(NO\ IM)}{TP(s.i, N_p)}$$

where $TP(NO\ IM)$ is the TP of the implementation without internal memory and $TP(s.i, N_p)$ is the measured TP of the solution $s.i$ and having a parallelism level $N_p = 1,\ 2\ or\ 4$.

$MEXP\ SU > 1$ implies that the MEXP optimizations are efficient.

- **The Speed Up on the temporal performance due to the parallelism**. It will be called "Parall. SU" and will be measured as:

$$Parall.\ SU(s.i, N_p) = \frac{TP(s.i, N_p = 1)}{TP(s.i, N_p = 2,\ 4)}$$

where $TP(s.i, N_p = 2,\ 4)$ is the temporal performance of the solution $s.i$, which can have a parallelism level $N_p = 2\ or\ 4$, and $TP(s.i, N_p = 1)$ is the temporal performance of the solution $s.i$ having a single pipeline.

$Parall\ SU > 1$ implies that the parallelism is efficient.

We can also measure the parallelism Efficacy as:

$$E = \frac{Parall.\ SU}{N_p}$$

with $0 \leq E \leq 1$. If $E \geq 0.5$ then the parallelism is efficient. E gives the percentage of hardware resources exploited during the algorithm execution and for E=1 all the hardware resources are exploited.

The area cost of the analyzed solutions will be characterized by the following metrics:

- **The Area Overhead due to the MEXP optimizations**. It will be called "MEXP AO" and will be computed as:

$$MEXP\ AO(s.i, N_p) = \frac{Total\ cost(s.i, N_p)}{Total\ cost(NO\ IM)}$$

where $Total\ cost(s.i, N_p)$ is the number of gates necessary to implement the RTL corresponding to the solution $s.i$ with a parallelism N_p and $Total\ cost(NO\ IM)$ is the number of gates necessary to realize a solution without internal memory.

- **The Area overhead due to the parallelism** is evaluated as follows:

$$Parall.\ AO = \frac{Total\ cost(s.i, N_p = 2\ or\ 4)}{Total\ cost(s.i, N_p = 1)}$$

The $Parall.\ AO$ should be inferior than N_p.

For each application and for each size of input image explored, we give two tables:

1. A table summarizing the information on the Temporal Performance of the solutions.
2. A table summarizing the information on the Area occupancy of the solutions.

An example of table summarizing the information on the temporal performance is given in figure 8.1. This table gives the Temporal Performance (TP) of two of the MEXP chosen solutions. For each solution, it gives two values of TP: the MEXP estimated TP and the TP measured after a logic simulation of the RTL generated by the HLS. The error on the estimation is given as a percentage with respect to the measured value.

This table compares the MEXP solutions with a solution without internal memory (and thus without prefetching mechanism). It also compares the MEXP solutions with each other and the implementations of the same solution with a different level of parallelism. In order to do that, the table gives: the speed up due to the MEXP optimization (MEXP SU), the speed up due to the parallelism(Parall. SU) and the parallelism efficacy (E).

		Measured TP NO IM (cycles)
Latency	30	...
	60	...
	100	...

		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.i					
Latency for $N_p = 1$	30	...	...	...	...	...	...
	60	...	...	...	...	...	...
	100	...	...	...	...	...	...
Latency for $N_p = 2$	30	...	...	...	...	...	...
	60	...	...	...	...	...	...
	100	...	...	...	...	...	...
Latency for $N_p = 4$	30	...	...	...	...	...	...
	60	...	...	...	...	...	...
	100	...	...	...	...	...	...
		mean value		...			
		max value			...		

Table 8.1: Example of a table summarizing the information on the TP.

An example of table summarizing the information on the area occupancy is given in figure 8.2. This table compares the area occupancy of a given MEXP solution $s.i$ with the area occupancy of an implementation without internal memory (NO IM). All the area costs are measured after the RTL generation and take into account the following contributions:

- The internal memories due to the MEXP optimizations (i.e. the mapping tables, the internal buffers, etc..)
- The memory requirements of the user-defined LUT and
- The local registers instantiated by the HLS tool.

The table gives the Area overhead due to the MEXP optimizations (MEXP AO) and the area overhead due to the parallelism (Parall AO).

	NO IM		
	cost - mm^2 total (combi, seq) ...)		
N_p	cost - mm^2	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
	s.i		
1	...	...	
2	...	...	...
4	...	...	...

Table 8.2: Example of table giving the information on the area occupancy of a solution

8.2 Graphical tools for the results analysis

We have considered two graphical representations of the exploration results:

- A scatter representation of the solution space which classifies the solutions explored by MEXP with respect to their Temporal Performance and their area occupancy. A point of the scatter graph corresponds to a MEXP solution (i.e. to an input and output tiling).

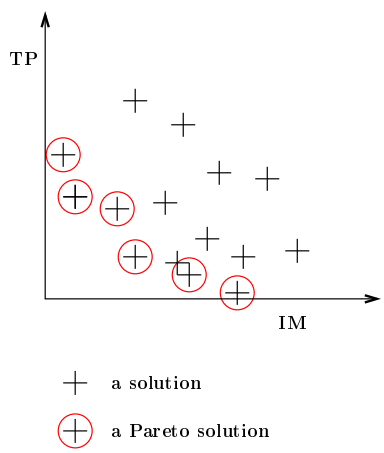


Figure 8.1: Example of a scatter representation of the solution space.

- A representation of the Temporal Performance of a solution with respect to the external memory latency.

  If we consider the ratio $\frac{TP^p}{TP^c}$ between the average time to pre-fetch TP^p and the average time to compute TP^c, we can individuate three zone in the TP representation:

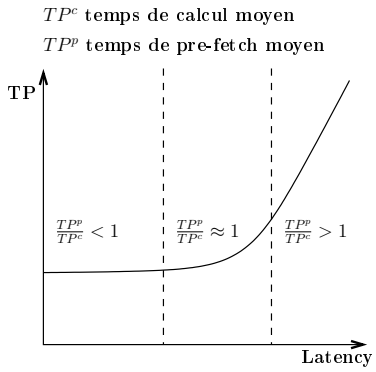


Figure 8.2: Example of a representation of TP with respect to the external memory variations.

1. A zone where the time to compute is preponderant with respect to the time to pre-fetch. During this phase the temporal performance of the final system does not depend upon the variations of the external memory latency.
2. A zone where the variations of the time to pre-fetch start to influence the Temporal Performance of the whole system.
3. A zone where the time to pre-fetch is preponderant with respect to the time to compute, thus the Temporal Performance varies linearly with the latency.

The variations of the time to prefetch can be due to either the variations of the external memory latency or the increase of the parallelism level N_p.

For both the scatter representation of the solution space and the representation of the Temporal Performance, the graphs are given for different values of external memory latency and different levels of parallelism N_p.

8.3 Conclusion

In this chapter, we have presented five metrics and two type of graphics. These tools will be used, in the next chapters, to describe the performed experiments.

The metrics presented are:

- Three metrics to describe the improvements of the MEXP optimizations on the temporal performance: the Speed Up due to the MEXP optimizations (MEXP SU), the Speed Up due to the parallelism (Parall SU) and the parallelism efficacy (E).
- Two metrics to describe the area overhead which can be due either to the MEXP optimizations (MEXP AO) or to the parallelism (Parall.AO).

The graphical tools are:

- A scatter representation of the MEXP exploration space, which classifies the solutions, in the explored space, by giving their temporal performance with respect to their used amount of internal memory. We will give this representation for 3 values of external memory latency (30, 60 and 100 cycles) and 3 values of parallelism level ($N_p = \{1,\ 2,\ 4\}$).
- A representation of the Temporal Performance(TP) of a solution with respect to the external memory latency. In this graph, there are three different zones where the value of the TP change in a different manner, depending on the ratio of the time to pre-fetch with respect to the time to compute. When this ratio is inferior than 1, the TP remain constant. When the ratio is around 1 the TP increases in a logarithmic way. When the ratio is superior than 1, the TP increase linearly with the increasing external memory latency.

Chapter 9

The Log Sampling

The pseudo-log sampling (LOG sampling) is a space-variant sampling of an image, which samples the pixels according to a pseudo-logarithmic law.

It can be used to correct the geometrical lens distortions, which occur when "the size of each pixel in the image plane is magnified in a different way" [105, 106, 107]. These distortions are symmetric with respect to the center of the image and can be corrected digitally and in real-time.

In the retina model developed at GIPSA lab [4], the log sampling is used to emulate the distribution of the photo-receptors (cones and rods) on the human retina. As the photo-receptors are numerous around the foveal zone, which is the center of the retina, the output image of a log sampling is magnified in the center. Figure 9.1 gives an example of the input and output of the log sampling, in this example there is also a reduction of the output image size.

(a) input (b) output

Figure 9.1: Example of an input and output images for a log sampling

Let consider the distances ρ_{in} and ρ_{out}, which are presented in figure in figure 9.2 and are the radial distances of an input and an output pixel respectively.

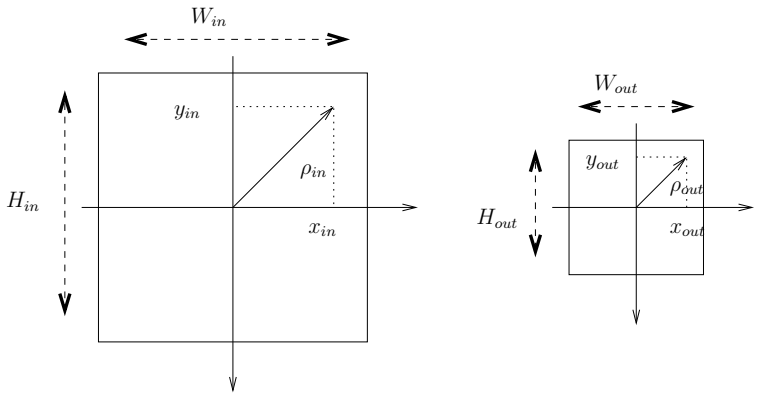


Figure 9.2: The input and output radial distances ρ_{in} and ρ_{out}.

These two distances are related by the pseudo-logarithmic law:

$$\rho_{out} = \rho_{lim} * \frac{\rho_{in}}{\rho_{in} + \rho_0} \tag{9.1}$$

with $\rho_{in} = \sqrt{(x_{in} - \frac{W_{in}}{2})^2 + (y_{in} - \frac{H_{in}}{2})^2}$ and $\rho_{out} = \sqrt{(x_{out} - \frac{W_{out}}{2})^2 + (y_{out} - \frac{H_{out}}{2})^2}$; ρ_{lim} is a constant fixing the reduction factor of the image sizes and ρ_0 a parameter which determines the compression intensity. ρ_{lim} and ρ_0 are related by the formula:

$$\rho_{lim} = \frac{1}{k} * \left(max(\frac{W_{in}}{2}, \frac{H_{in}}{2}) + \rho_0 \right)$$

and their effect on the output image are shown in figure 9.3.

(a) $\rho_0 = \frac{W_{in}}{2}$ and $k = 2$

(b) $\rho_0 = \frac{W_{in}}{2}$ and $k = 4$

(c) $\rho_0 = \frac{W_{in}}{4}$ and $k = 2$

(d) $\rho_0 = \frac{W_{in}}{4}$ and $k = 4$

Figure 9.3: Example of output image for different values of ρ_0 and k

In our experiments we have used $\rho_0 = \frac{W_{in}}{2}$ and $k = 2$.

From 9.1 it is possible to deduce the law giving the coordinates of the input pixels to be sampled from the coordinates of the output pixel to be computed. This law is:

$$\begin{pmatrix} x_{in} - \frac{W_{in}}{2} \\ y_{in} - \frac{H_{in}}{2} \end{pmatrix} = \frac{\rho_0}{\rho_{lim} - \rho_{out}} * \begin{pmatrix} x_{out} - \frac{W_{out}}{2} \\ y_{out} - \frac{H_{out}}{2} \end{pmatrix} \tag{9.2}$$

In our implementation we use a bilinear interpolation to avoid the aliasing caused by the non-affinity of the law computing the input pixels coordinates (details are given by annex A).

Furthermore, the non-linear function computing the input pixel coordinates is realized by using a Look Up Table (details are given in annex B).

9.1 The TPU synthesizable C-model for the LOG sampling

Once the user has performed all the studies concerning the LUT, he has all the information needed to construct the input for a MEXP analysis.

In order to customize the TPU generic model, the user has to define the functionality of REQ and CALC and generate, through MEXP, the hardware configuration.

In the TPU model for the LOG sampling, the module REQ computes (by using the user-defined LUT) the coordinates of the 4 input pixels needed to perform a bilinear interpolation and requests them to the FETCH module. The CALC module receives the input pixels and computes the output by performing the bilinear interpolation.

Figure 9.4 gives the C-code of the user-defined functions REQ and CALC for the LOG sampling.

The REQ function computes the coordinates (OTx and OTy in the code) of the currently computed output tile. It infers from them the absolute coordinates (a0 and a1 in the code) of the output pixel to be produced. From these coordinates it computes the LUT entry and, then, from the LUT, the needed input pixel coordinates. Finally it transfers the address of the needed input pixel to the FETCH module. The transfer is performed by streams. The CALC function receives the data from FETCH and computes the output through a bilinear interpolation.

9.2 The MEXP analysis on the LOG sampling

The MEXP explorations for the LOG sampling have been run with respect to three input image sizes: SQCIF, VGA and HDTV. For each one of these sizes we have considered several I/O tile sizes.

Table 9.1 gives an overview of the experiments run.

I_I and I_O are the input an output images spaces; S^I and S^O are the I/O spaces which contains I_I and I_O and whose dimensions are power of two, V_{IT} and V_{OT} are the sets of possible I/O tile sizes to be explored; N_s is the number of analyzed couples of I/O tilings. For example, for a SQCIF input image an analyzed possible couple of I/O tiles is $(IT, OT) = (4 \times 4, 4 \times 2)$.

```
#define MASK(L)(1<<L)-1
#define N_PIXEL 4
void REQ( int i, int j , int np , int *DX, int *ADD_REQ){
.... declarations ....
OTy = ((OT_order[np]-1)&MASK(N_out_0))<<N_OT_0;
OTx = ((OT_order[np]-1)>>N_out_0)<<N_OT_1;

a0 = i + OTx;
a1 = j + OTy;

index = (a0 - O_WIDTH_0)*(a0 - O_WIDTH_0)
   + (a1 - O_WIDTH_1)*(a1 - O_WIDTH_1);
index_int = index >>LUT_STEP_BITWIDTH;

if( index_int > LUT_WIDTH - 2){
#pragma unroll z
  for(z=0;z<N_PIXEL;z++)
    ADD_REQ[z]=0xFFFFFFFF;
}else{
  p1 =LUT[index_int];
  p2 =LUT[index_int +1];
  dy = p2-p1;
  dx = (index)&MASK(PREC);
  p=((dx*dy)>>(LUT_STEP_BITWIDTH+PREC))+p1;

  x = p*(a0 - (O_WIDTH_0>>1))
   + (I_WIDTH_0>>1)<<PREC;
  y = p*(a1 - (O_WIDTH_1>>1))
   + (I_WIDTH_1>>1)<<PREC;

  dx = x&MASK(PREC); x = x >> PREC;
  dy = y&MASK(PREC); y = y >> PREC;

  if(x < I_WIDTH_0 && y < I_WIDTH_1){
    DX[0] = (dx<<16)|dy;
    ADD_REQ[0] = (x<<16)|y;
    ADD_REQ[1] = ((x+1)<<16)|y;
    ADD_REQ[2] = (x<<16)|(y+1);
    ADD_REQ[3] = ((x+1)<<16)|(y+1);
  }else{
#pragma unroll z
  for(z=0;z<N_PIXEL;z++)
    ADD_REQ[z]=0xFFFFFFFF;
  }
}
}
```

```
void CALC(int i, int j, int np, int *DX,
      int *DATA, int *out_ADD, int *out_DATA){
.... declarations ....
OTy = ((OT_order[np]-1)&MASK(N_out_0))<<N_OT_0;
OTx = ((OT_order[np]-1)>>N_out_0)<<N_OT_1;
dy = DX [0] &MASK(16);
dx = (DX [0]>> 16) &MASK(16);
*out_DATA = Interpol(dx, dy, DATA[0],
      DATA[1], DATA[2], DATA[3]);
*out_ADD = ((i+oTx)<<16)|(j+oTy);
}
```

```
int Interpol(int dx, int dy, int a, int b, int c, int d){
val = (1-dx)*(1-dy)*a+dx*(1-dy)*b+(1-dx)*dy*c+dx*dy*d;
}
```

Figure 9.4: REQ and CALC code. All the macros and global variables, shared between the user-defined and MEXP generated code are in bold.

	I_I	I_O	S^I	S^O	V_{IT}	V_{OT}	N_s
SQCIF	128 × 96	64 × 48	128 × 128	64 × 64	32, 16, 8	32, 16, 8	36
VGA	640 × 480	320 × 240	1024 × 512	512 × 256	512, 256, 128	512, 256, 128	324
HDTV	1920 × 1080	960 × 540	2048 × 2048	1024 × 1024	2048,1024	2048,1024	272

Table 9.1: Experiments run for different input image sizes

For each possible couple of input and output tiling, we consider an implementation with a TPU with 1, 2, or 4 parallel pipelines. Thus the solution space contains $N_s \times 3$ solutions. Furthermore, the temporal performance of each on of the explored solutions has been evaluated with respect to a memory latency of 30, 60 and 100 cycles. Thus each input image size $N_s \times 9$ explorations have been run.

The tools used to summarize the analysis are given in chapter 8.

9.2.1 SQCIF input image

In this paragraph we will describe the results of the MEXP exploration for a SQCIF input image with I/O tile sizes of 32, 16, 8. The design space contains 108 possible solutions

analyzed for 3 values of external memory latency.

Figure 9.5 gives the scatter representation of the MEXP exploration results for a SQCIF input image. From the sub-figures we can see that the increase of external memory latency degrades the temporal performance of the TPU. The increase of the parallelism level improves the temporal performance and enlarges the area occupancy of the solutions. The magnification of each sub-figure shows that the chosen solutions (which are enclosed in a circle) are pareto. The following table shows some of the MEXP analyzed solutions for an exploration with a single pipeline.

The pareto solutions are not necessarily trivial and bring to a real improvement with respect to the non-pareto solutions.

Table 9.3 summarizes the information on the Temporal Performance (TP) for the two MEXP chosen solutions (s.13 and s.6). Form this table we can conclude that the MEXP SU variates between 3.2 and 6.7 and the "Parall SU" variates between 0.97 and 1.29. This means that, for a SQCIF input image, in the worst case and due to the bandwidth limitations, the parallelism could even degrade the temporal performance of the system. The mean error of the MEXP estimations is less than 10%.

Table 9.4 summarizes the information on the area occupancy of the two MEXP solutions (s.13 and s.6). As we could have expected and due to the usage of the internal buffers, the area cost of a MEXP solution is increased of a factor 2.5 with respect to a solution without internal memory. However, the area of the sequential part of the circuit is increased more than the area of the logic part, but its amount remains non-preponderant with respect to the total area of the circuit.

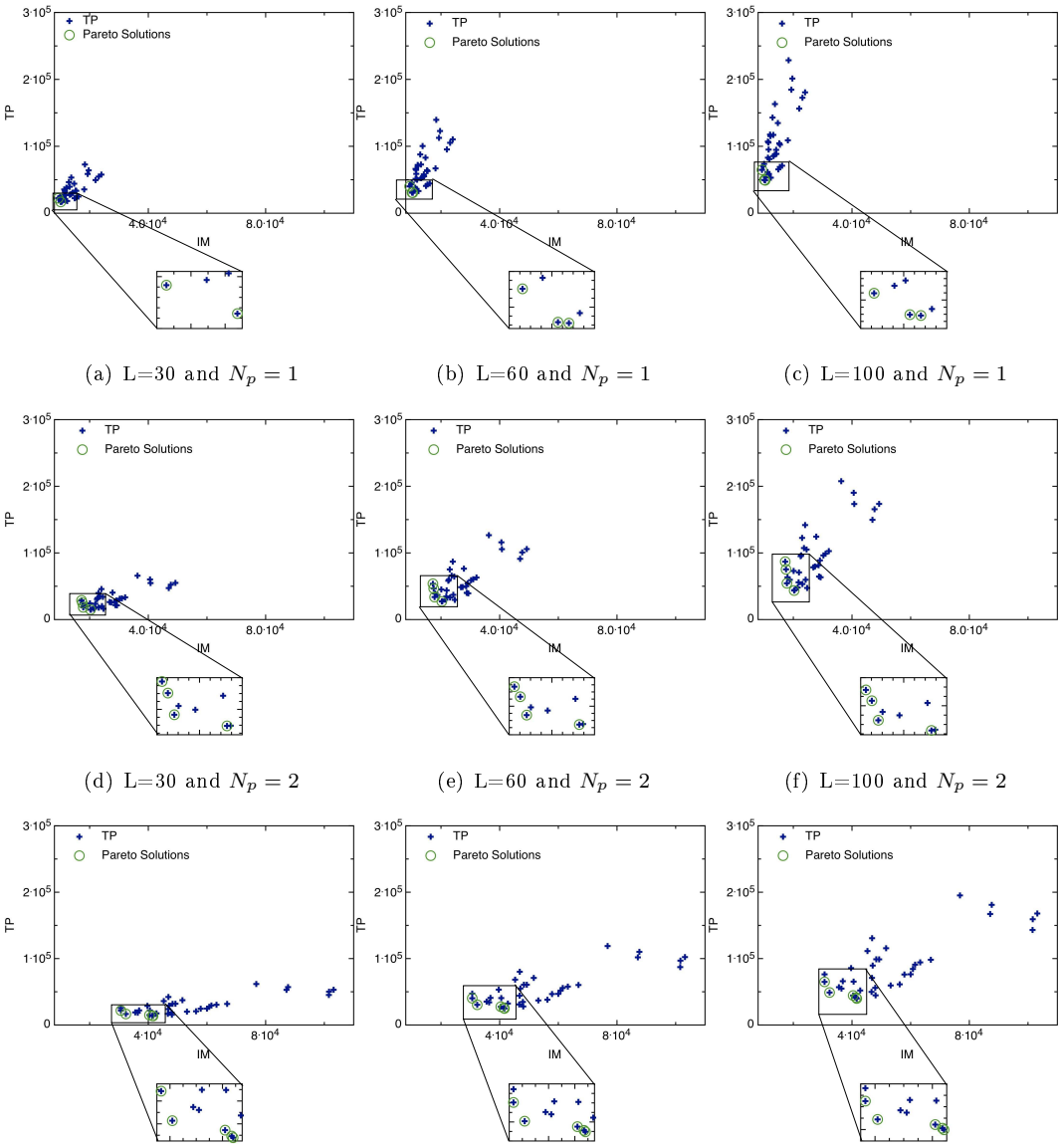
On the other hand, the area overhead due to the parallelism is optimized, in fact a parallelism with $N_p = 2$ increases the area of a factor around 1.6 and a parallelism with $N_p = 4$ increases the area of a factor around 2.6.

Figure 11.5 gives the graph of the Temporal Performance of the MEXP chosen solutions (s.13 and s.6) and of a solution without internal memory.

We can observe that, for both the solutions (s.13 and s.6), the ratio of the time to pre-fetch with respect to the time to compute is superior than 1. Thus any variation of the time to prefetch influences the temporal performance of the system.

In the graphs 9.6(a), 9.6(b) and 9.6(c) the temporal performance of the MEXP solutions s.13 and s.6 are compared with each other. We can see that solution s.6 has a better temporal performance than solution s.13, as we could have expected due to the fact that s.6 uses a larger internal memory than s.13.

Graph 9.6(d) compares the MEXP solutions with a realization not using an internal memory and confirms that the Speed Up due to the MEXP optimization variates between 3.2 and 6.7.



(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

Figure 9.5: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF.

sol.	(IT,OT)	Area	Est. TP (for Latency)		
			30	60	100
s.3	(16×2, 32×1)	10144	16878	30780	49700
s.13	(8×4, 16×1)	9312	22354	40128	64728
s.6	(8×4, 32×1)	10400	16896	30516	49116
s.35	(8×1, 8×1)	18390	72796	139576	228616

Table 9.2: Solutions s.3, s.13 and s.6 are pareto solutions, while s.35 is a non-pareto solution.

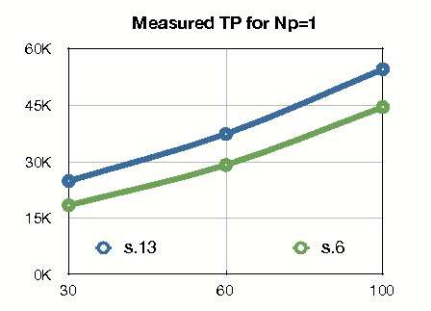
		Measured TP NO IM (cycles)
Latency	30	79883
	60	172043
	100	294923

		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.13					
	30	22354	24952	-10.41	3.2		
Latency for	60	40128	37420	7.24	4.6		
$N_p = 1$	100	64728	54620	18.51	5.4		
	30	18303	20030	-8.62	4	1.25	0.62
Latency	60	33767	33155	1.85	5.2	1.13	0.56
for $N_p = 2$	100	54447	50675	7.44	5.8	1.1	0.54
	30	16554	19298	-14.22	4.2	1.29	0.32
Latency	60	30401	32798	-7.31	5.2	1.14	0.28
for $N_p = 4$	100	48881	50798	-3.78	5.8	1.1	0.27
		s.6					
	30	16896	18480	-8.57	4.3		
Latency for	60	30516	29185	4.56	5.9		
$N_p = 1$	100	49166	44585	10.27	6.6		
	30	14794	16690	-11.36	4.7	1.11	0.55
Latency	60	27124	28480	-4.76	6	1.02	0.51
for $N_p = 2$	100	43564	44200	-1.44	6.7	1	0.5
	30	13574	16690	-19.86	4.7	1.09	0.27
Latency	60	24848	29358	-15.36	5.9	0.99	0.25
for $N_p = 4$	100	39829	45918	-13.26	6.4	0.97	0.24
		mean value		9.4			
		max value			6.7		

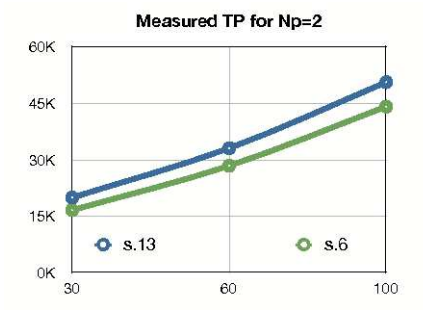
Table 9.3: Estimated and measured Temporal Performance (TP) for a SQCIF.

	NO IM		
	cost - mm^2 total (combi, seq) 0,039 (0,03 - 0,009)		
N_p	cost - mm^2	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.13	
1	0.096 (0.06 - 0.036)	2.49 (1.99 - 4.1)	
2	0.153 (0.093 - 0.06)	3.95 (3.14 - 6.56)	1.58 (1.58 - 1.58)
4	0.24 (0.16 - 0.08)	6.2 (5.29 - 9.1)	2.49 (2.65 - 2.23)
		s.6	
1	0.097 (0.06 - 0.037)	2.5 (1.99 - 4.19)	
2	0.163 (0.09 - 0.073)	4.2 (3.2 - 7.46)	1.67 (1.6 - 1.78)
4	0.270 (0.16 - 0.11)	6.9 (5.5 - 11.8)	2.7 (2.76 - 2.81)

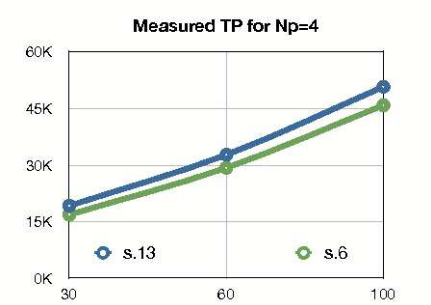
Table 9.4: Measured cost of the TPU after the RTL generation



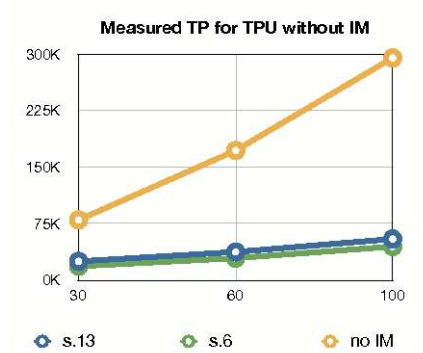
(a) $N_p = 1$



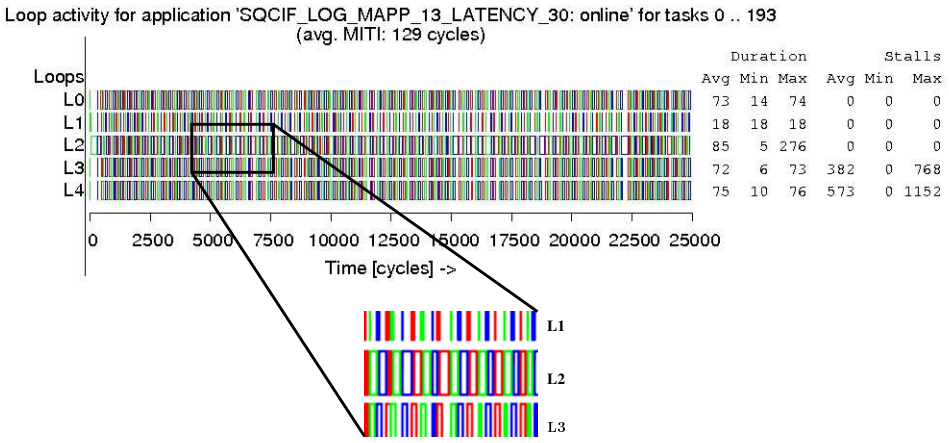
(b) $N_p = 2$



(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model

Figure 9.6: Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF.

9.2.2 VGA input image

In this paragraph we will describe the results of the MEXP exploration for a VGA(640×480) input image with I/O tile sizes of 512, 256, 128. The design space contains 972 possible solutions, analyzed for 3 values of external memory latency.

Figure 9.7 gives the scatter representation of the MEXP explorations. Form this figure we can infer that:

- the temporal performance of the set of analyzed solutions depends less on the variations of the external memory latency with respect to the SQCIF and
- the improvements due to the parallelism are more efficient than those obtained for a SQCIF

In fact, the temporal performance remains around $3 * 10^5$ cycles for $N_p = 1$ and is bounded between $1.5 * 10^5$ and $2 * 10^5$ for $N_p = 2$. It varies more for $N_p = 4$.

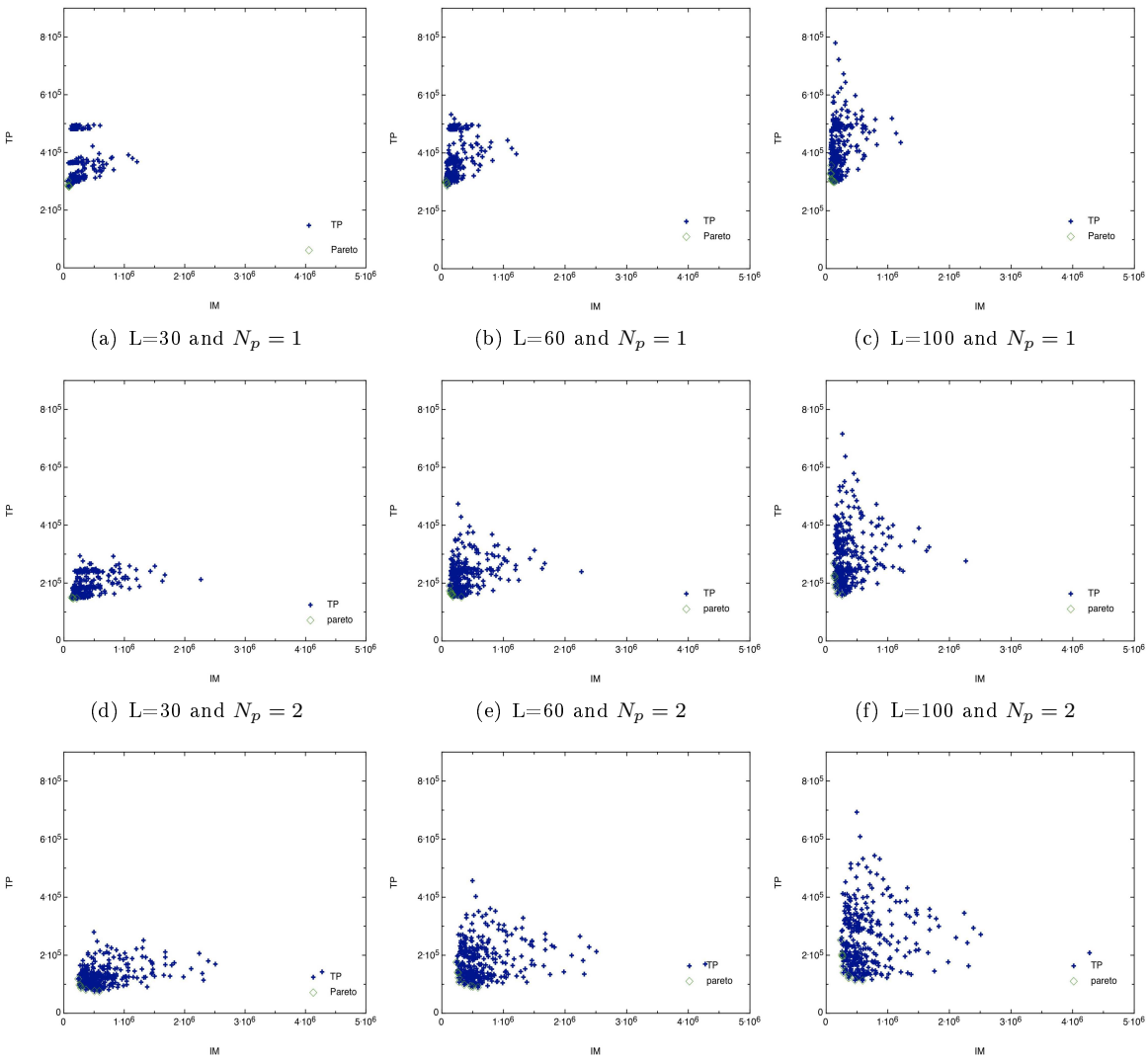
Table 9.5 gives the information on the temporal performance for the solutions (s.220 and s.223). Where s.220 is pareto with respect to the used amount of internal memory and s.223 is pareto with respect to the temporal performance. From table 9.5, we can see that the MEXP SU varies between 6.43 and 36.13. The Parallelism efficacy is superior than 0.9 for $N_p = 2$ and $N_p = 4$. But it decreases with the increasing of the external memory latency.

The mean error of the MEXP estimations is less than 6%, thus the MEXP exploration can be considered quite accurate.

Table 9.6 gives the information on the area occupancy for the solutions (s.220 and s.223). From this table we can infer that the Area Overhead with respect to a solution without internal memory is around 4 (for an implementation without parallelism) and around 9 for a parallelism level $N_p = 4$. The AO due to the parallelism with respect to the solution with a single pipeline is around 1.6 for $N_p = 2$ and around 2.9 for $N_p = 4$.

Figure 9.8 gives the graphical representation of Temporal Performance with respect to the latency variations. From this figure we can infer that:

- for the solution s.220, the ratio of the time to pre-fetch with respect to the time to compute is inferior than 1 for $N_p = 1$, around 1 for $N_p = 2$ and higher than 1 for $N_p = 4$;
- for the solution s.223, which uses a smaller amount of internal memory, this ratio is around 1 for $N_p = 1$ and $N_p = 2$ and higher than 1 for $N_p = 4$.



(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

Figure 9.7: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA.

		Measured TP NO IM (cycles)
Latency	30	1996841
	60	439981
	100	7372841

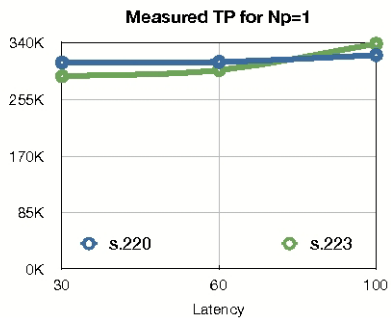
		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.220					
	30	302179	310650	-2.72	6.43		
Latency for	60	303511	311431	-2.54	13.8		
$N_p = 1$	100	328143	321577	2.04	22.93		
	30	152844	155845	-1.93	12.81	1.99	0.99
Latency	60	173606	171781	1.06	25.04	1.81	0.91
for $N_p = 2$	100	226072	211906	6.68	34.79	1.51	0.75
	30	97768	79936	22,31	24.98	3.88	0.97
Latency	60	141650	146162	-3.09	29.42	2.13	0.53
for $N_p = 4$	100	203322	204050	-0.35	36.13	1.58	0.39
		s.223					
	30	282727	289832	-2.45	6.89		
Latency for	60	297571	299118	-0.52	14.38		
$N_p = 1$	100	356155	338718	5.14	21.77		
	30	153306	145535	5.34	13.72	1.99	0.99
Latency	60	193656	189279	2.31	22.72	1.58	0.79
for $N_p = 2$	100	263376	247781	6.29	29.75	1.36	0.68
	30	111862	77332	44.65	25.82	3.74	0.93
Latency	60	162638	163165	-0.32	26.36	1.83	0.45
for $N_p = 4$	100	233182	228993	1.83	32.19	1.47	0.37
		mean value		6.2			
		max value			36.13		

Table 9.5: Estimated and measured Temporal Performance (TP) for a VGA input image.

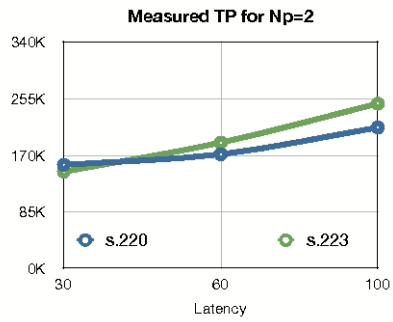
NO IM
cost - mm^2
total (combi, seq)
0.041 (0.031, 0.01)

N_p	cost - gates	AO WRT NO IM (nX)	AO due to the Parall. (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.220	
1	0.14 (0.06, 0.08)	3.56 (2.06, 8.23)	
2	0.25 (0.1, 0.15)	6.11 (3.19, 15.15)	1.71 (1.55, 1.84)
4	0.45 (0.17, 0.28)	10.99 (5.52, 27.9)	3 (2.68, 3.39)
		s.223	
1	0.17 (0.06, 0.11)	4.2 (2.14, 10.62)	
2	0.27 (0.1, 0.17)	6.7(3.25, 17.52)	1.59 (1.52, 1.64)
4	0.48 (0.17, 0.31)	11.6 (5.47, 30.6)	2.75 (2.55, 2.88)

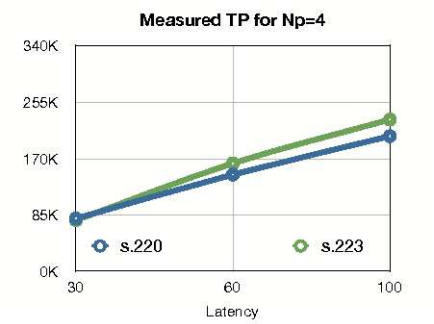
Table 9.6: Measured cost of the TPU after the RTL generation.



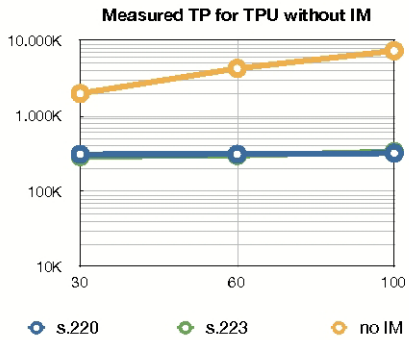
(a) $N_p = 1$



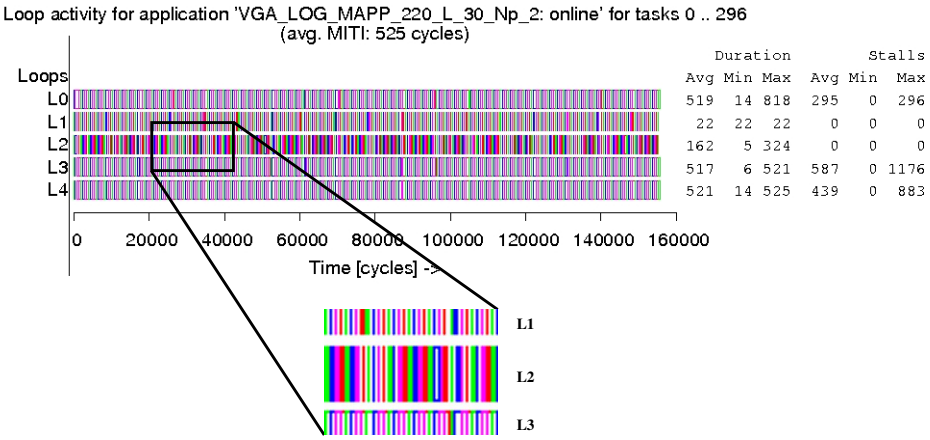
(b) $N_p = 2$



(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model

Figure 9.8: Measured temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA.

9.3 HDTV input image

This paragraph gives the results of the MEXP exploration for a HDTV(1920X1080) input image, with I/O tiles sizes of 2048 and 1024. The explored space contains 272 solutions evaluated with respect to three values of external memory latency (30, 60 and 100 cycles).

Figure 9.9 gives the scatter representation of the MEXP explorations. From this figure we can see that the Temporal Performance of the solutions does not vary neither with the increasing of the external memory latency nor with the increasing of the parallelism level, except for the case $N_p = 4$ and $L = 100$.

On the other side, the parallelism is efficient; in fact, the Temporal Performance is around $2 * 10^5$ for $N_p = 1$; $1 * 10^5$ for $N_p = 2$ and $5 * 10^4$ for $N_p = 4$.

We will consider the solutions s.252 and s.165 from the analyzed space, where s.252 is a pareto solution and s.165 is non-pareto. Their configuration is shown by table 9.7.

Table 9.8 gives the temporal performance of the RTL realizations of the considered solutions (s.252 and s.165). From this table we can infer that :

- The MEXP SU variates between 6.4 and 93.3 for the pareto solution.
- The parallelism efficacy is always superior than 0.9 except for the non-pareto solution.
- The mean error on the MEXP estimations is 9.4%.

Table 9.9 gives the area overhead due to the MEXP optimizations and the area overhead due to the parallelism. From this table we can observe that:

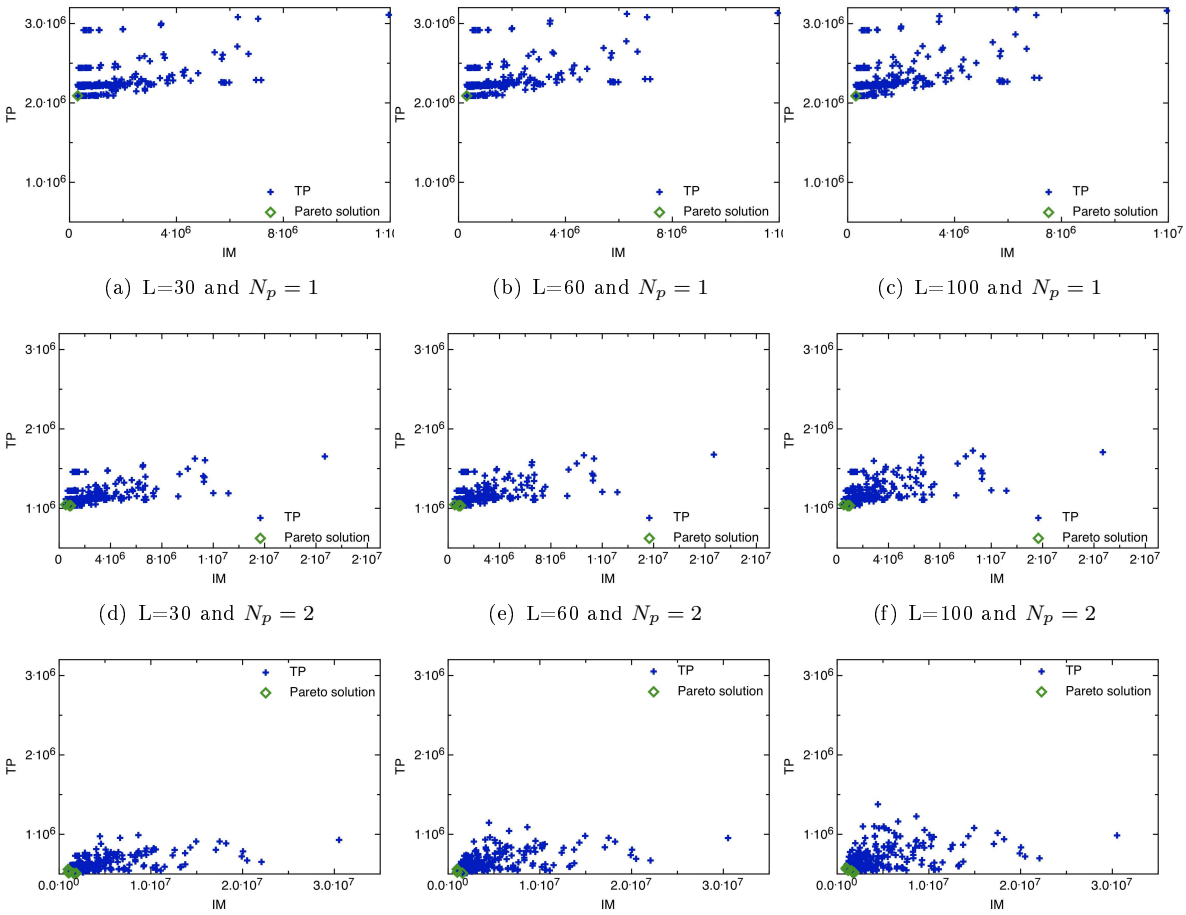
- the AO due to the MEXP optimization is important (form 9.11 to 27.21 for the pareto solution s.252 and from 28.6 to 87.5 for the non-pareto solution s.165). This may be due to :
  - the analyzed sizes of the input and output tiles
  - the total amount of input tiles, in fact this amount affects the size of the IDX memory (which is used to directly access the internal buffers - see paragraph 3.2.2 of chapter 3 for more details)

However the pareto solution s.252 has a maximal speed up of 93.3 (for $N_p = 4$) with a corresponding area overhead of 27.2X.

While the non-pareto solution s.165 has a lower speed up (65 for $N_p = 4$) for an area overhead 4 times bigger (i.e. 87). This result shows the consequence, on the realized RTL, of the choice of an input and output couple of tilings which are not adapted to the target application.

Figure 9.10 gives the temporal performance variations with respect to the increasing latency. For the pareto solution, the temporal performance does not depend on the

latency variations and the parallelism is efficient. In fact, the ratio of the time to prefetch with respect to the time to compute is of 0.14.

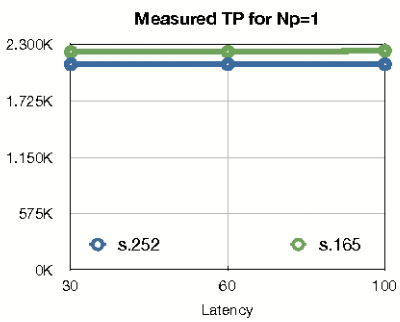


(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

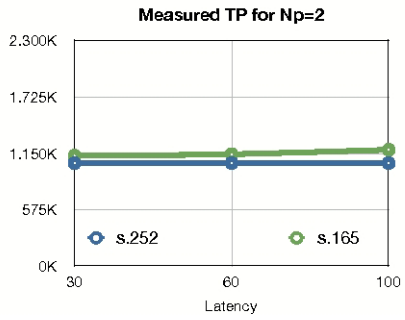
(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

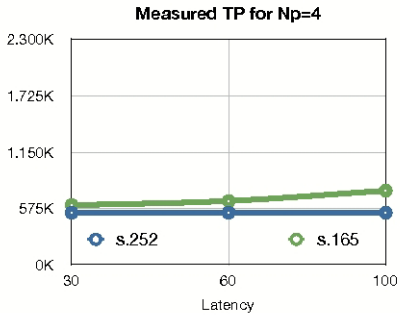
Figure 9.9: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV.



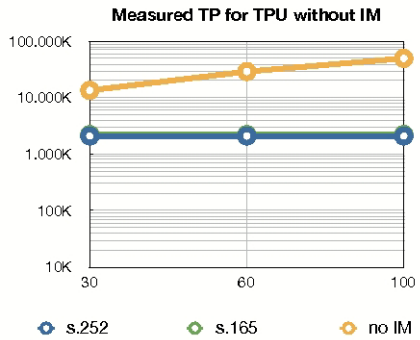
(a) $N_p = 1$



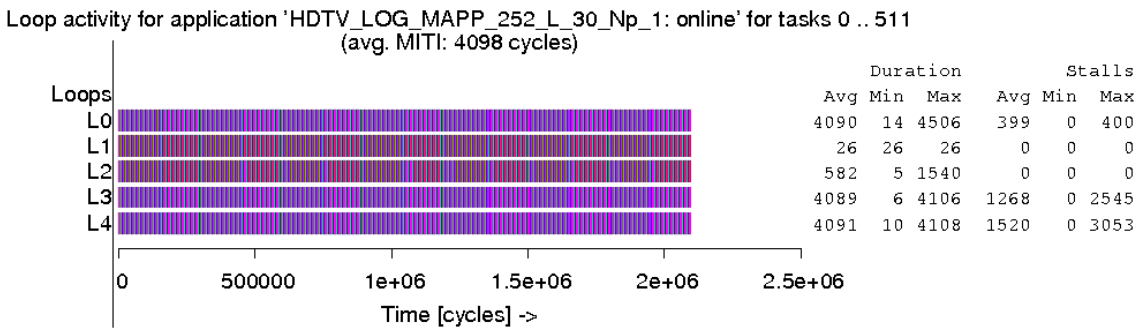
(b) $N_p = 2$



(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model

Figure 9.10: Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV.

sol.	(IT,OT)	cost (bits)	Est. TP (for Latency) 30	60	100
s.252	(32×32, 64×16)	304420	2089476	2089596	2089756
s.165	(256×4, 32×64)	1138540	2213814	2215708	2225484

Table 9.7

		Measured TP NO IM (cycles)
Latency	30	13478419
	60	29030419
	100	49766419

		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.252					
	30	289476	2098097	-0.41	6.42		
Latency for	60	2089596	2098097	-04	13.8		
$N_p = 1$	100	2089756	2098097	-0.39	23.7		
	30	1045001	1051866	-0.6	12.81	1.99	0.99
Latency	60	1045121	1051866	-0.6	27.6	1.99	0.99
for $N_p = 2$	100	1045281	1051866	-0.6	47.3	1.99	0.99
	30	517400	533390	-2.09	25.26	3.93	0.98
Latency	60	522232	533390	-2.09	54.42	3.93	0.98
for $N_p = 4$	100	562360	533390	5.4	93.3	3.93	0.98
		s.165					
	30	2213814	2223553	-0.44	6		
Latency for	60	2215708	2225473	-0.44	13		
$N_p = 1$	100	2225484	22236112	-0.47	22.25		
	30	1117536	1127892	-0.91	11.9	1.97	0.98
Latency	60	1132420	1142290	-0.86	25.4	1.94	0.97
for $N_p = 2$	100	1165070	1188116	-1.94	4188	1.88	0.94
	30	605390	610149	-0.77	22	3.64	0.91
Latency	60	648054	652750	-0.72	44.5	3.4	0.85
for $N_p = 4$	100	713210	758350	-5.9	65.62	2.94	0.74
		mean value		9.4			
		max value			93.3		

Table 9.8: Estimated and measured Temporal Performance (TP) for a HDTV input image.

NO IM
cost - mm^2
total (combi, seq)
0,041 (0,03 - 0,01)

N_p	cost - mm^2	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.252	
1	0.37 (0.07 - 0.3)	9.11 (2.2 - 30.7)	
2	0.6 (0.1 - 0.5)	14.8 (3.34 - 50.37)	1.62 (1.54 - 1.64)
4	1.12 (0.17 - 0.95)	27.21 (5.59 - 94.3)	2.98 (2.58 - 3)
		s.165	
1	1.18 (0.09 - 1.09)	28.6 (2.75 - 109.5)	
2	2.02 (0.13 - 1.9)	48.9 (4.32 - 187.6)	1.7 (1.6 - 1.7)
4	3.6 (0.2 - 3.4)	87.5 (6.85 - 337.9)	3.05 (2.5 - 3)

Table 9.9: Measured cost of the TPU after the RTL generation.

9.4 Conclusion

In this chapter we have presented the results of three explorations on the LOG sampling. We have analyzed three input image sizes (SQCIF, VGA and HDTV). For each input image size we have explored hundreds of solutions. From the results we can infer that the MEXP optimizations depend on the I/O tile sizes and on the I/O data space sizes. In particular, they are more efficient when we consider an large input image (VGA and HDTV) than a smaller image (SQCIF).

The speed up of the temporal performance due to the MEXP optimizations can be up to 93.3 for a corresponding area overhead of 27.21 (results observed for the HDTV input image with a parallelism level $N_p = 4$). The highest speed up observed, without considering the parallelism is up to 23.7 for a corresponding area overhead of 9.11 (for a HDTV input image).

The parallelism is not efficient in the case of a SQCIF input image and is very efficient for the HDTV (E>0.9).

Chapter 10

The Pyramidal Log Sampling

The LOG sampling is preceded by a space-variant low-pass for two reasons:

- In order to respect the Nyquist-Shannon sampling theorem and avoid the aliasing, it is necessary to filter the high frequencies of the input image. As the LOG sampling is space-variant, also the preceding low-pass has to be space-variant and in particular its filtering factor has to increase with the increasing eccentricity of the pixel position.

- The space-variant low-pass reproduces the behavior of the human retina, which provides a vision blurred on the borders and neat on the center.

Figure 10.1 gives an example of the input and output of the chain composed by a space-variant low-pass followed by a LOG sampling. We can see that the output image is

(a) input (b) output

Figure 10.1: Example of an input and output images for a space-variant low-pass followed by a LOG sampling

blurred on the borders and neat on the center.

The space-variant low-pass have several levels of dependences that require the storage of the whole processed image and may prevent the application of the MEXP optimizations (see annex C for more details).

In order to eliminate these dependences, we have approximated the chain composed by the low-pass and the log sampling with a MIP mapping LOG sampling, where MIP stands for "Multum in Parvo"[a].

The MIP-mapping method consists in computing a pyramid of several levels indexed by C. Each level C contains an image which is filtered and down-sampled with respect to the image of the level $C-1$.

In the implemented version, that will be called pyramidal LOG-sampling, the MIP-map contains 3 levels. The function used to construct and access the MIP-map levels is $log_k(f(x_{out}, y_{out}))$, where $f(x_{out}, y_{out}) = \frac{\rho_0}{\rho_{lim}-\rho_{out}}$ is the LOG sampling transformation. k is a parameter of the LOG sampling which gives the reduction factor of the output image size and can have the following values $k = 2,\ 4,\ 8\ldots$ (for more details on why we have chosen the function $log_k()$ and in general on the Pyramidal LOG sampling see annex D).

An example of the pyramid is given in figure 10.2.

Figure 10.2: Example of a pyramid with 3 levels and constructed with a low-pass having a window size variable with the number of level

The non-affinity of array references can cause an aliasing when sampling the input pixels. In order to avoid it, we apply a tri-linear interpolation, which is composed of a bilinear interpolation of the pixels sampled in a level and a linear interpolation of the pixels sampled on two successive levels.

Furthermore in the hardware implementation of the Pyramidal LOG sampling we have used a Look Up Tables to realize the LOG sampling transformation $f(x_{out}, y_{out})$ and a Look Up Table to realize the pyramid access law $log_k()$.

[a] many things in a small space

10.1 The TPU synthesizable C-model of the Pyramidal LOG sampling

Figure 10.3 gives the code of the REQ and CALC functions for the Pyramidal LOG sampling.

The first difference with the LOG sampling code is that the Pyramidal sampling uses two LUTs : LUT_0 realizing the LOG sampling transformation and LUT_1 realizing the function $log_k()$ used to access the pyramid levels.

By using the LUT_0 output, REQ computes on one hand the coordinates of the input pixel to be sampled (x and y in the code) and on the other hand the entry of the LUT_1. The LUT accesses are always performed together with a linear interpolation in order to improve the precision of the LUT, for more detail see annex B.

The pyramid level is noted C, in the code. C is not an integer, thus, in order to avoid the aliasing, two levels of the pyramid are accessed: those indexed by the superior and inferior integers of C.

From each level, 4 input pixels coordinates have to be requested and these coordinates are computed from the value of x and y.

In total REQ sends 8 requests to FETCH, which responds by sending the 8 corresponding data to CALC.

CALC receives the 8 data and performs two bilinear interpolations, one for each level. The results of the bilinear interpolations are store in variables $v1$ and $v2$, which are, then, linearly interpolated. Finally, CALC sends out the computed output datum and the coordinates in the output image, where it has to be stored.

The loops realizing the modules REQ, CALC and FETCH (which is not given in figure 10.3) lasts a number of cycles which is 8 times the volume of the output tile.

Furthermore, because of the stream accesses, FETCH is delayed of 8 cycles with respect to REQ and CALC is delayed of 8 cycles with respect to CALC.

10.2 The MEXP analysis on the Pyramidal LOG sampling

As for the LOG sampling, we have run the MEXP analysis for three input image sizes: SQCIF, VGA and HDTV. To each one of these image sizes corresponds a 3 levels pyramid, which is the real input image I_I of the application. The output image I_O is the same as for the LOG sampling.

The table 10.1 gives, for each analyzed input image size (SQCIF, VGA and HDTV), the size of the corresponding pyramid (I_I) and the size of the produced output image (I_O). It also gives the size of the analyzed spaces (S^I and S^O) which are power of two and contain respectively the the input and output used images. Finally, it gives the volumes of the input and output tiles V_{IT} and V_{OT}, each one producing several input or output tiling layouts, and the number N_s of analyzed couples of I/O tilings.

Each one of the N_s couples of input and output tiles corresponds to a hardware model whose temporal performance and area occupancy will be evaluated with respect to three values of external memory latency (30, 60 and 100 cycles) and three levels of

	I_I	I_O	S^I	S^O	V_{IT}	V_{OT}	N_s
SQCIF	224 × 168	64 × 48	256 × 256	64 × 64	128, 64	128,64	72
VGA	1120 × 840	320 × 240	2048 × 1024	512 × 256	1024, 512	1024, 512	210
HDTV	3360 × 1890	960 × 540	4096 × 2048	1024 × 1024	2048	1024	72

Table 10.1: Experiments run for different input image sizes

possible inter-tiles parallelism (i.e. N_p pipelines computing N_p output tiles in parallel, with $N_p = \{1\ 2\ 4\}$). Thus the explored space contains $N_s \times 9$ possible solutions.

The temporal performance and the area occupancy of the solutions are evaluated with respect to the metrics defined in paragraph 8.1 of chapter 8.

10.2.1 SQCIF input image

In this paragraph we will describe the results of the MEXP exploration for a SQCIF input image with I/O tile sizes of 128 and 64. The design space contains 216 possible solutions, analyzed for 3 values of external memory latency.

Figure 10.4 gives the scatter representation of the MEXP explorations.

We can see that the Temporal Performance of the solutions increases with the increasing of the external memory latency and the parallelism level.

Table 10.2 gives two pareto solutions (s.33 and s.65) of the space and a non-pareto solution (s.71). The pareto solutions have a better temporal performance and use less internal memory than the solution s.71, which has a trivial layout.

sol.	(IT,OT)	Area	Est. TP (for Latency)		
			30	60	100
s.33	(16×8, 32×2)	20956	24751	24969	26022
s.65	(16×4, 32×2)	17920	24842	25184	27760
...					
s.71	(8×8, 8×8)	21036	24768	25768	31784

Table 10.2: Examples of solutions in the analyzed space

For $N_p = 1$, the solution s.33 is pareto with respect to the temporal performance and the solution s.65 is pareto with respect to the internal memory used.

From **table 10.3**, we can infer that:

- The MEXP SU varies between 7.8 and 48.9.
- The mean error of the MEXP estimations is 15.26%.
- The parallelism efficacy is around 0.8 for $N_p = 2$ and of 0.5 for $N_p = 4$.. The parallelism efficacy is reduced by the increase of the external memory latency and this especially for the solution s.65 which uses a smaller amount of internal memory.

From **table 10.4** we can infer that:

- The maximum area overhead due to the MEXP optimizations is of 5.8X; the area overhead depends especially on the usage of internal buffers which is more important than for the LOG sampling.
- The parallelism has a maximum area overhead of 1.56 for $N_p = 2$ and of 3 for $N_p = 4$

Figure 11.5 gives the Temporal Performance of the solutions along the external memory latency variations. From figures 10.5(a), 10.5(b) and 10.5(c), we can infer that, for both the solutions, the ratio of the time to pre-fetch with respect to the time to compute is inferior than 1 for $N_p = 1$, around 1 for $N_P = 2$ and higher than 1 for $N_p = 4$.

This is also shown by figures 10.5(e) and 10.5(f) which give the time-line of the two solutions for $N_p = 1$ and $Latency = 30$. The average length of the loop performing the pre-fetching (L_2) is shorter than the length of the computations loops L_0, L_3 and L_4.

But the average length of the prefetching of the solution s.65 is more important than that of the solution s.33, thus the temporal performance of solution s.65 is more affected by the increase of the parallelism level, than that of solution s.33.

		Measured TP NO IM (cycles)
Latency	30	202817
	60	479287
	100	847927

		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.33					
Latency for $N_p = 1$	30	24751	25763	-3.9	7.87		
	60	24969	26085	-4.2	18.4		
	100	26022	26952	-3.4	31.5		
Latency for $N_p = 2$	30	12467	13897	-10.3	14.6	1.85	0.93
	60	13103	14814	-11.5	32.3	1.76	0.88
	100	15412	17323	-11	48.9	1.55	0.77
Latency for $N_p = 4$	30	6499	9625	-32.5	21.07	2.67	0.66
	60	8041	12748	-36.9	37.6	2.4	0.51
	100	11049	17879	-38.2	47.42	1.5	0.37
		s.65					
Latency for $N_p = 1$	30	24842	25837	-3.85	7.8		
	60	25182	26293	-4.26	18.22		
	100	27760	27774	-0.05	30.53		
Latency for $N_p = 2$	30	12561	14041	-10.5	14.44	1.84	0.92
	60	14327	15978	-10.3	29.99	1.64	0.82
	100	21092	22618	-6.74	37.5	1.22	0.61
Latency for $N_p = 4$	30	7326	10504	-30.25	19.3	2.46	0.61
	60	12204	17164	-28.9	27.9	1.53	0.38
	100	18884	26164	-27.82	32.4	1.067	0.26
		mean value		15.26			
		max value			48.9		

Table 10.3: Estimated and measured Temporal Performance (TP) for a SQCIF input image.

NO IM
cost - mm^2
total (combi, seq)
0,039 (0,03 - 0,009)

N_p	cost - gates	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.33	
1	0.14 (0.08 - 0.06)	1.9 (1.42 - 4)	
2	0.218 (0.124 - 0.094)	3 (2.1 - 6.8)	1.56 (1.47 - 1.7)
4	0.41 (0.21 - 0.2)	5.8 (3.7 - 14.4)	3 (2.63 - 3.7)
		s.65	
1	0.137 (0.084 - 0.053)	1.88 (1.43 - 3.8)	
2	0.21(0.12 - 0.09)	2.92 (2.1 - 6.4)	1.55 (1.46 - 1.69)
4	0.42 (0.23 - 0.195)	5.78 (3.7 - 14.3)	3 (2.66 - 3.7)

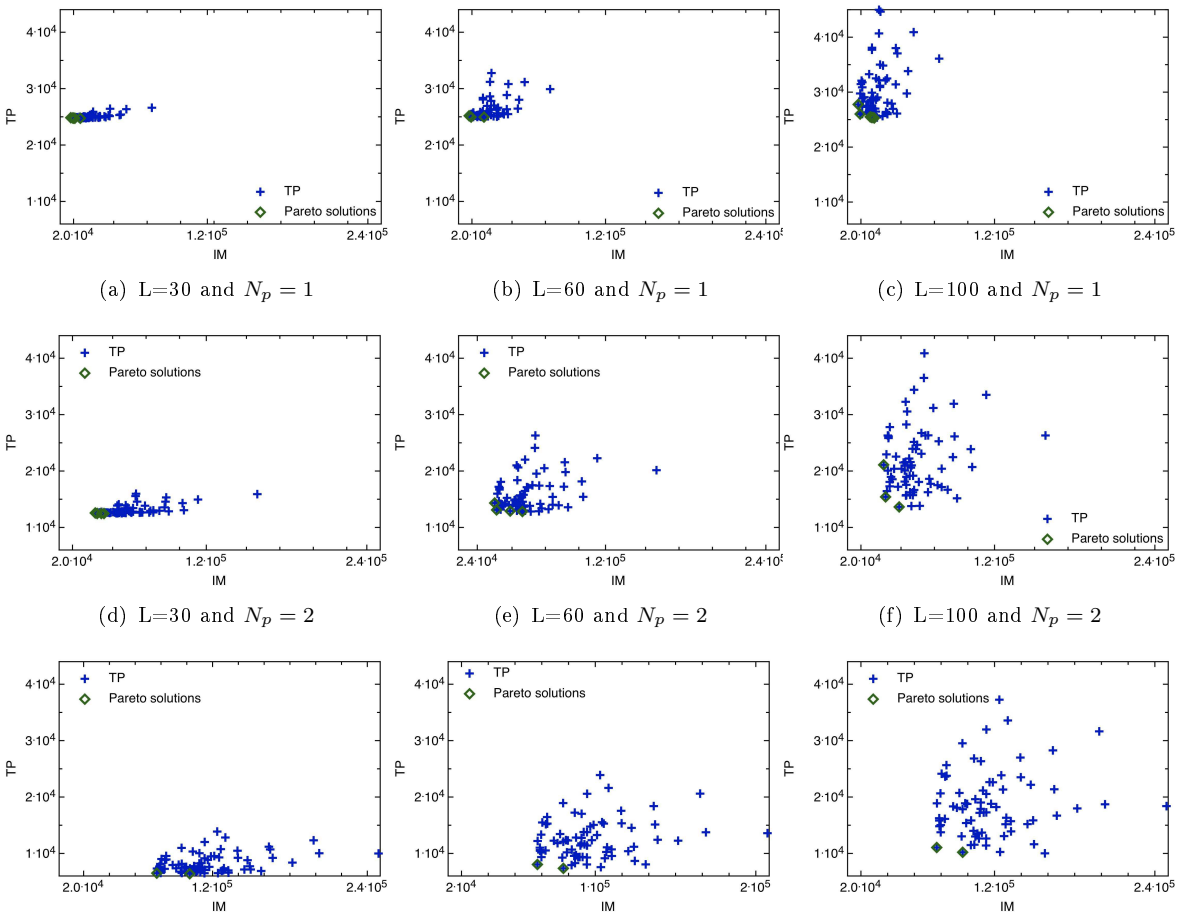
Table 10.4: Measured cost of the TPU after the RTL generation.

```
#define MASK(L)(1<<L)-1
#define N_PIXEL 4
void REQ( int i, int j , int l , int *DX, int *ADD_REQ){
.... declarations ....
OTy = ((OT_order[l]-1)&MASK(N_out_0))<<N_OT_0;
OTx = ((OT_order[l]-1)>>N_out_0)<<N_OT_1;

a0 = i + OTx;
a1 = j + OTy;

index = (a0 - O_WIDTH_0)*(a0 - O_WIDTH_0)
   + (a1 - O_WIDTH_1)*(a1 - O_WIDTH_1);
index_int = index >>LUT_STEP_BITWIDTH_0;

if( index_int > LUT_WIDTH_0 - 2){
#pregma unroll z
  for(z=0;z<N_PIXEL;z++)
    ADD_REQ[z]=0xFFFFFFFF;
}else{
  p1 =LUT_0[index_int];
  p2 =LUT_0[index_int +1];
  dy = p2-p1;
  dx = (index)&MASK(PREC);
  p=((dx*dy)>>(LUT_STEP_BITWIDTH_1+PREC))+p1;

  x = p*(a0 - (O_WIDTH_0>>1))
   + (I_WIDTH_0>>1)<<PREC;
  y = p*(a1 - (O_WIDTH_1>>1))
   + (I_WIDTH_1>>1)<<PREC;

  if(x < I_WIDTH_0 && y < I_WIDTH_1){
    index_int = p >>LUT_STEP_BITWIDTH_1;
    p1 =LUT_1[index_int];
    p2 =LUT_1[index_int +1];
    dy = p2-p1;
    dx = (index)&MASK(PREC);

    C=((dx*dy)>>(LUT_STEP_BITWIDTH_1+PREC))+p1;
    C_1=C_>>PREC;
    C_2=C_1+1;

    x1 = (x >> C_1) >> PREC;
    y1 = (y >> C_1) >> PREC;
    if(C_1==1){          x1+=I_WIDTH_0;
      y1+=I_WIDTH_1;
    }

    dx = (x>>C_1)&MASK(PREC); x = x >> PREC;
    dy = (y>>C_1)&MASK(PREC); y = y >> PREC;

    DX[0] = (dx<<16)|dy;
    ADD_REQ[0] = (x1<<16)|y1;
    ADD_REQ[1] = ((x1+1)<<16)|y1;
    ADD_REQ[2] = (x1<<16)|(y1+1);
    ADD_REQ[3] = ((x1+1)<<16)|(y1+1);

    x1 = (x >> C_2) >> PREC;
    y1 = (y >> C_2) >> PREC;
    if(C_2==1){
      x1+=I_WIDTH_0;
      y1+=I_WIDTH_1;
    }else{
      x1+=I_WIDTH_0+I_WIDTH_0>>1;
      y1+=I_WIDTH_1+I_WIDTH_1>>1;
    }

    dx = (x>>C_2)&MASK(PREC); x = x >> PREC;
    dy = (y>>C_2)&MASK(PREC); y = y >> PREC;

    DX[1] = (dx<<16)|dy;
    ADD_REQ[4] = (x1<<16)|y1;
    ADD_REQ[5] = ((x1+1)<<16)|y1;
    ADD_REQ[6] = (x1<<16)|(y1+1);
    ADD_REQ[7] = ((x1+1)<<16)|(y1+1);
    DX[2]=C&MASK(PREC);
  }else{
#pregma unroll z
  for(z=0;z<N_PIXEL;z++)
    ADD_REQ[z]=0xFFFFFFFF;
  }
}
}
```

```
void CALC(int i, int j, int l, int *DX,
      int *DATA, int *out_ADD, int *out_DATA){
.... declarations ....
OTy = ((OT_order[l]-1)&MASK(N_out_0))<<N_OT_0;
OTx = ((OT_order[l]-1)>>N_out_0)<<N_OT_1;

dx[0][1] = DX [0] &MASK(16);
dx[1][1] = DX [1] &MASK(16);
dx[0][0] = (DX [0]>> 16) &MASK(16);
dx[1][0] = (DX [1]>> 16) &MASK(16);

v1 = Interpol(dx[0][0], dx[0][1], DATA[0],
      DATA[1], DATA[2], DATA[3]);
v2 = Interpol(dx[1][0], d[1][1], DATA[4],
      DATA[5], DATA[6], DATA[7]);

*out_DATA =(1 -DX[2])v1 +DX[2]*v2;
*out_ADD = ((i+oTx)<<16)|(j+oTy); }
```

```
int Interpol(int dx, int dy, int a, int b, int c, int d){
val = (1-dx)*(1-dy)*a+dx*(1-dy)*b+(1-dx)*dy*c+dx*dy*d;
}
```

Figure 10.3: Pyramidal TPU code. All the macros and global variables, shared between the user-defined and MEXP generated code are in bold.

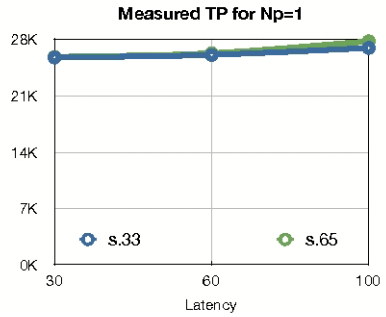


(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

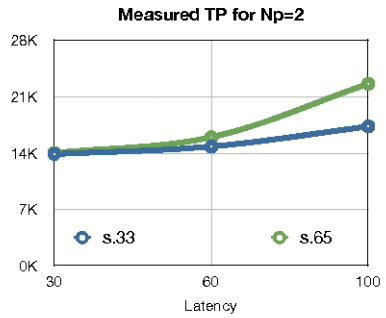
(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

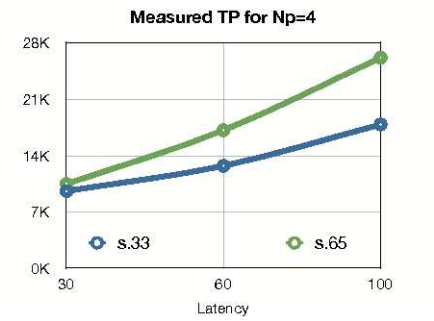
Figure 10.4: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF.



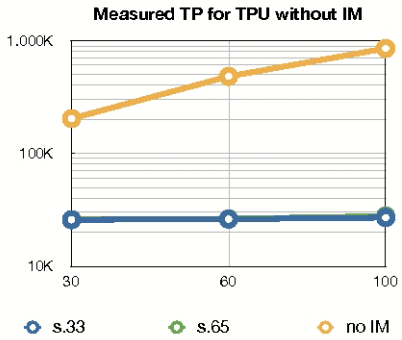
(a) $N_p = 1$



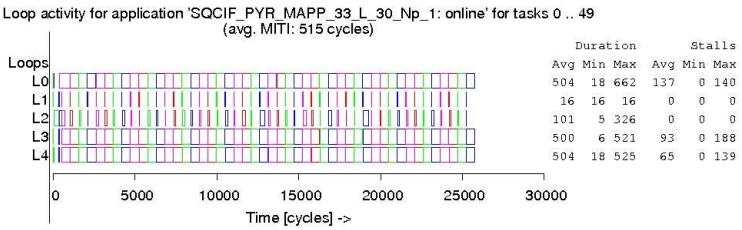
(b) $N_p = 2$



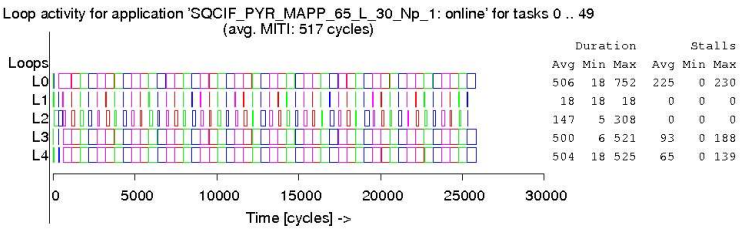
(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model



(f) Simulation of the TPU verilog model

Figure 10.5: Measured temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF.

10.2.2 VGA input image

This paragraph describes the results of the MEXP exploration for a VGA input image with I/O tile sizes of 1024 an 512. The design space contains 630 possible solutions analyzed for 3 values of external memory latency.

Figure 10.6 gives the scatter representation of the MEXP explorations. From this figure we can infer that the temporal performance of the solutions does not vary with the increasing latency and is improved by the parallelism. In fact the TP of the solutions is around $8 * 10^5$ for $N_p = 1$, around $4 * 10^5$ for $N_p = 2$ and around $2 * 10^5$ for $N_p = 4$.

For a variable latency $L = 30, 60, 100$ and a parallelism level $N_p = 1$, we can distinguish three separate clouds of solutions whose temporal performance is around $9 * 10^5$, $7 * 10^5$ and $6 * 10^5$. The temporal performance depends on the I/O tilings layout, in fact the solutions in the cloud with the worse TP have an output tile layout which is the less squared possible. For several solutions the area occupancy can became very high because of the parallelism.

Table 10.5 compares the MEXP estimations on the area occupancy and the temporal performance for two solutions (the pareto s.200 and the non-pareto s.150) with $N_p = 1$.

sol.	(IT,OT)	Area	Est. TP (for Latency)		
			30	60	100
s.200	(16×32, 32×16)	161310	598337	598457	598617
s.150	(16×32, 64×16)	213900	615099	615339	615659

Table 10.5: Examples of explored solutions

The Temporal Performance of the two solutions does not vary with respect to the latency variations, but s .200 has a better TP than s.150 and uses less internal memory.

From **table 10.6**, we can infer that:

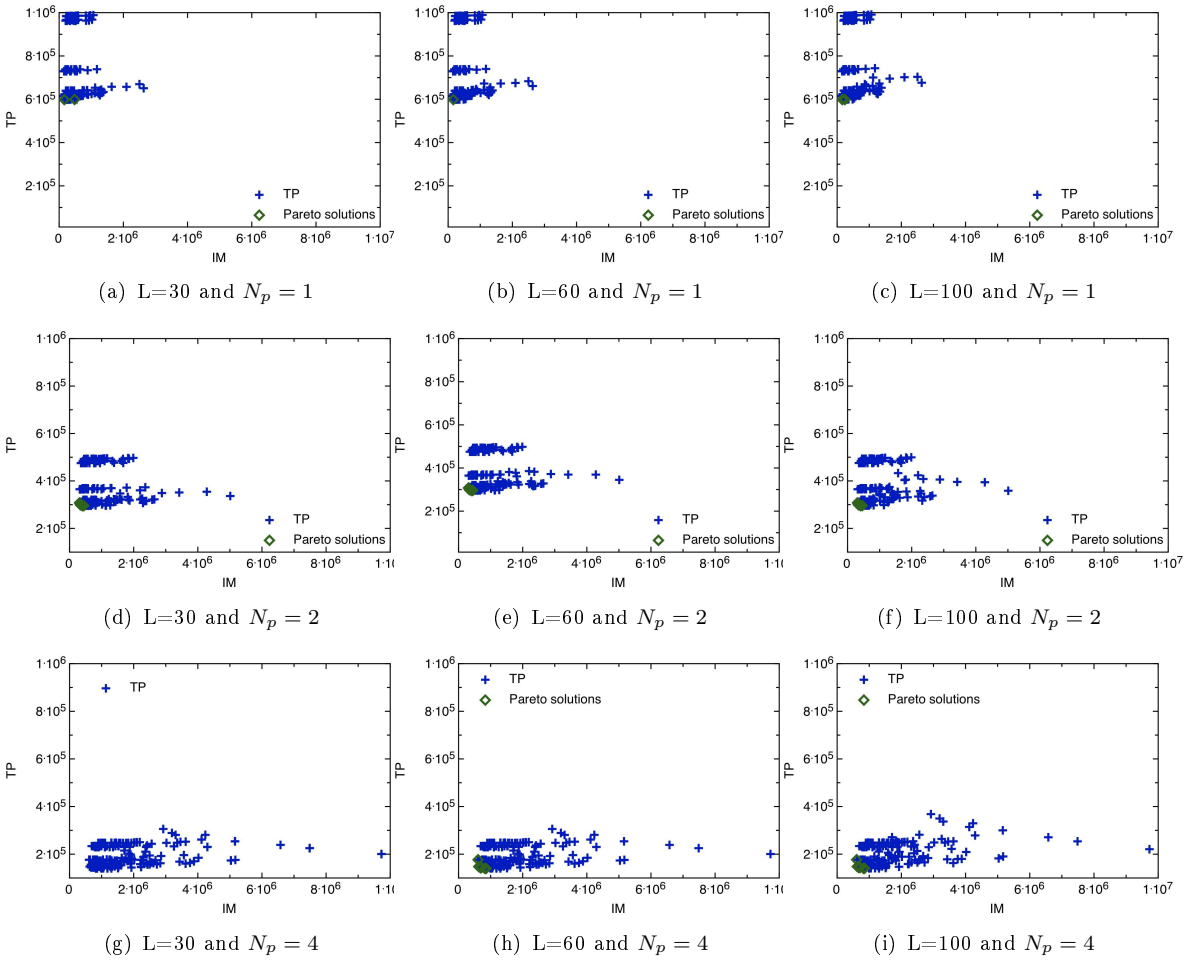
- The MEXP SU varies between 8.22 and 127.81.
- The mean error of the MEXP estimations is 5.69%.
- The parallelism efficacy is always superior than 0.93 (except for the non pareto solution and $N_p = 4$).

From **table 10.7** we can infer that:

- The maximum area overhead due to the MEXP optimizations is of 14.47 for the non-pareto solution and of 12.3 for the pareto solution.
- The parallelism has a maximum area overhead (with respect to the solution with a single pipeline) of 1.84 for $N_p = 2$ and of 3.4 for $N_p = 4$.

The results obtained after the HLS confirm that the pareto solution is faster and uses less internal memory than the non pareto solution.

Figures 10.7(a), 10.7(b) and 10.7(c) confirm that the temporal performance of the solutions are invariant with the increasing latency. Figure 10.7(d) confirms that the speed up due to the MEXP optimization can be up to 127.81. Figures 10.7(e) and 10.7(f) confirm that, for both the pareto and the non-pareto solutions, the ratio of the time to pre-fetch with respect to the time to compute is inferior than 1. But the pareto solution is faster than the non-pareto solution. In fact the pareto solution identifies 4 output tiles, on the corners of the output image, whose corresponding input data requests are invalid and thus can be eliminated.



(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

Figure 10.6: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA.

		Measured TP NO IM (cycles)
Latency	30	5068865
	60	11980855
	100	21196855

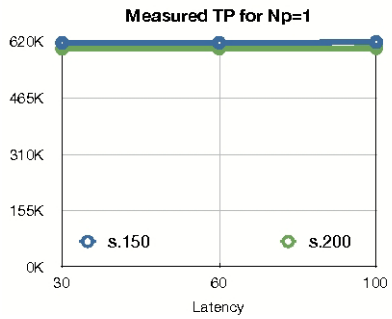
		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.150					
	30	615099	616673	-0.25	8.22		
Latency for	60	615339	616673	-0.21	194		
$N_p = 1$	100	615659	616673	-0.45	34.27		
	30	295622	315552	-6.31	16.06	1.95	0.97
Latency	60	295862	315552	-6.23	37.97	1.95	0.97
for $N_p = 2$	100	296182	315552	-6.13	67.17	1.95	0.97
	30	140245	163689	-14.32	30.92	3.76	0.94
Latency	60	140245	163689	-14.32	73.19	3.76	0.94
for $N_p = 4$	100	140565	174289	-19.34	121.61	3.54	0.88
		s.200					
	30	598337	600882	-0.42	8.43		
Latency for	60	598457	600882	-0.4	19.94		
$N_p = 1$	100	598617	600882	1-0.6	35.2		
	30	299333	301345	-0.66	15.77	1.87	0.93
Latency	60	299353	301345	-0.63	37.28	1.87	0.93
for $N_p = 2$	100	299613	301359	-0.57	65.96	1.87	0.93
	30	143829	158435	-9.21	31.99	3.79	0.94
Latency	60	143829	158435	-9.21	75.62	3.79	0.94
for $N_p = 4$	100	144130	165835	-13	127.81	3.63	0.9
		mean value		5.9			
		max value			127.81		

Table 10.6: Estimated and measured Temporal Performance (TP) for a VGA input image.

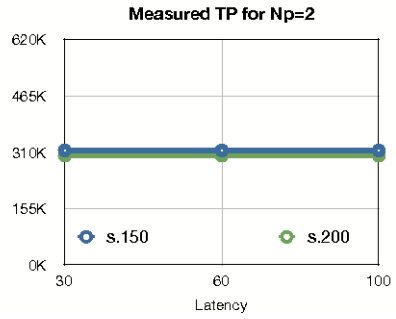
NO IM
cost - mm^2
total (combi, seq)
0,07 (0,06 - 0,01)

N_p	cost - gates	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.150	
1	0.3(0.08 - 0.23)	4.24 (1.44 - 16.4)	
2	0.58 (0.13 - 0.45)	7.8 (2.19 - 32.15)	1.84 (1.52 - 1.96)
4	1.08(0.24 - 0.84)	14.47 (5.29 - 60)	3.4 (2.7 - 3.66)
		s.200	
1	0.27 (0.08 - 0.19)	3.6 (1.42 - 13.2)	
2	0.48 (0.13 - 0.35)	6.4 (2 - 25.6)	1.77 (1.44 - 1.93)
4	0.92 (0.23 - 0.69)	12.3 (3.8 - 49.5)	3.4 (2.67 - 3.74)

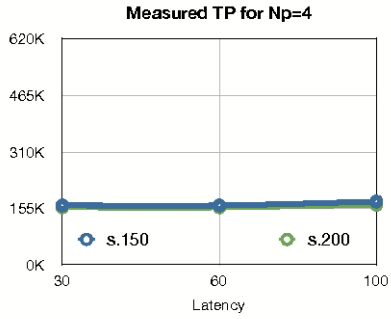
Table 10.7: Measured cost of the TPU after the RTL generation.



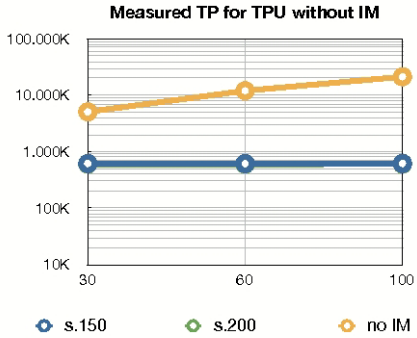
(a) $N_p = 1$



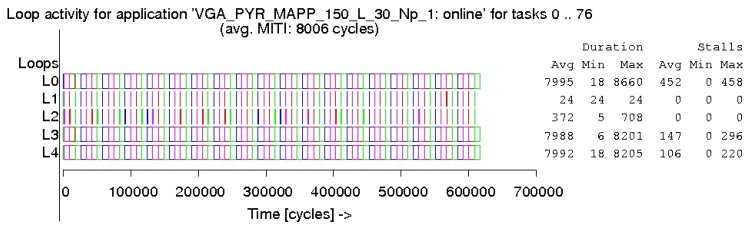
(b) $N_p = 2$



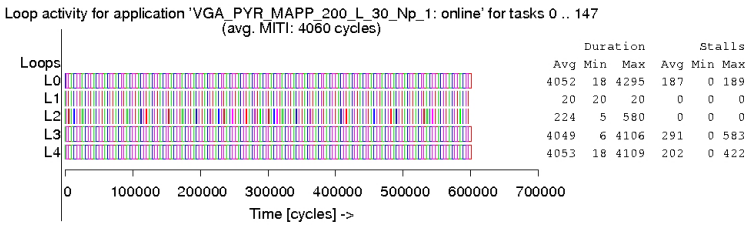
(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model



(f) Simulation of the TPU verilog model

Figure 10.7: Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA.

10.2.3 HDTV input image

This paragraph describes the results of the MEXP exploration for a HDTV input image with an input tile size of 2048 and an output tile size of 1024. The design space contains 216 possible solutions, analyzed for 3 values of external memory latency.

Figure 10.8 gives the scatter representation of the MEXP explorations.

As for the explorations on a VGA input image, the temporal performance of the solution does not depends on the latency variations and the parallelism is efficient.

Table 10.8 compares the MEXP estimations on the area occupancy and the temporal performance for three solutions with $N_p = 1$: two solutions are pareto (s.44 and s.55) and the other is non-pareto (s.45) .

Concerning the pareto solutions, one of them is pareto with respect to the temporal performance (s.44) and the other is pareto with respect to the area occupancy (s.55).

sol.	(IT,OT)	Area	Est. TP (for Latency)		
			30	60	100
s.44	(64×32, 64×16)	406716	4178816	4178936	4179096
s.55	(32×64, 8×128)	342610	4883041	4883131	4883251
s.45	(64×32, 32×32)	505098	4178819	4178936	4179096

Table 10.8: Examples of solutions from the analyzed space

The non pareto solution has a worse temporal performance and uses more internal memory than the two other solutions.

From **table 10.9**, we can infer that:

- The speed up due to the MEXP optimizations variates between 6.97 and 134.12.
- The mean error of the MEXP estimations is 1.2%.
- The parallelism efficacy is always superior than 0.98 and is not reduced by the increase of the external memory latency.

From **table 10.10** we can infer that:

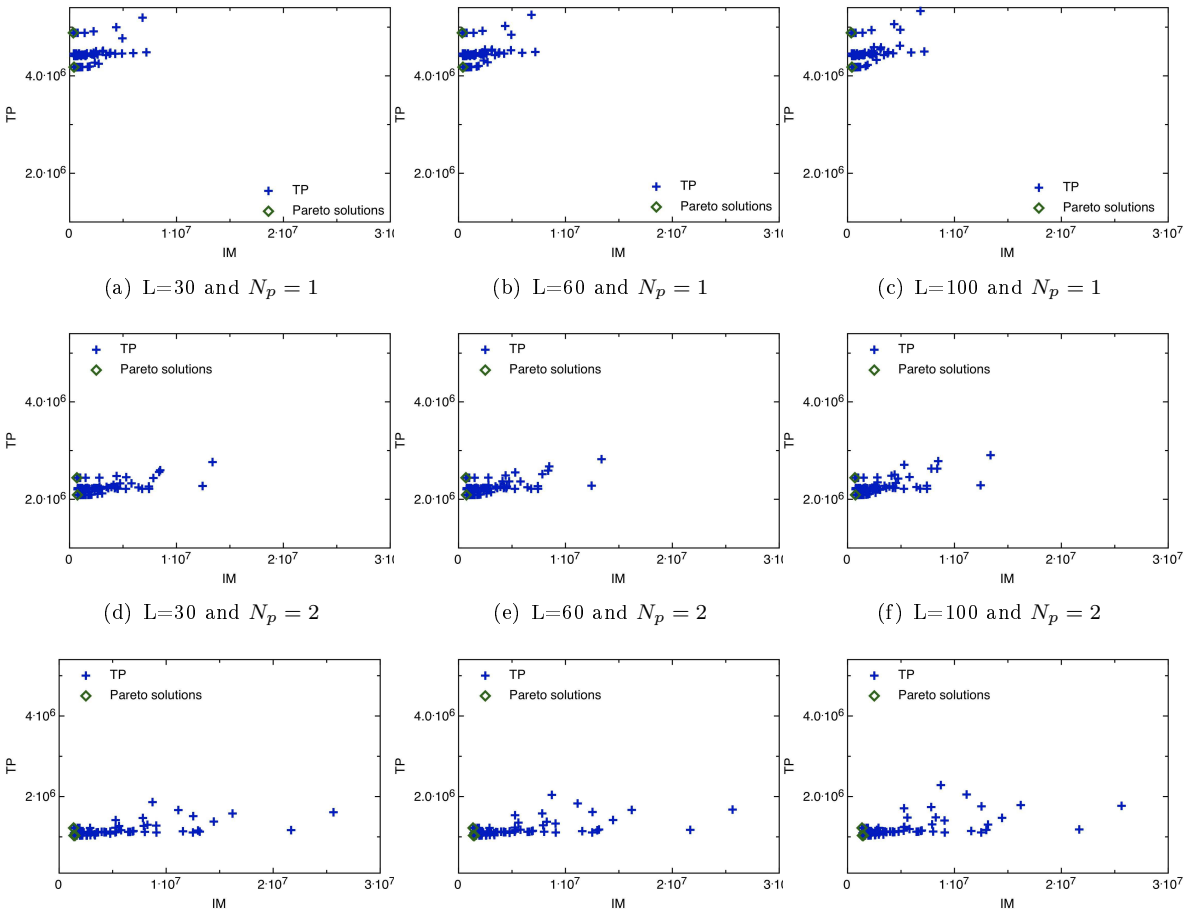
- The maximum area overhead due to the MEXP optimizations is of 21.9.
- The parallelism has a maximum area overhead of 1.71 for $N_p = 2$ and of 3.1 for $N_p = 4$

Figures 10.9(e) and 10.9(f) confirm that, for both the pareto solutions, the average length of the pre-fetching is shorter than the average length of the other loops.

The differences between the temporal performance of the two solutions is due to their output tile layouts. In fact, the minimum temporal performance[b] to execute the

[b]This temporal performance corresponds to the case when the whole input image already is in an internal memory and each input pixel is accessible in a clock cycle.

algorithm is of 960*540*8=4147200 cycles (where 960*540 is the output image size and 8 is the number of input needed per output tiles). For the solution s.55, the output tile layout is so that the amount of computations includes some useless computations which are not eliminated.



(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

Figure 10.8: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV.

		Measured TP NO IM (cycles)
Latency	30	3421439
	60	8087435
	100	143078435

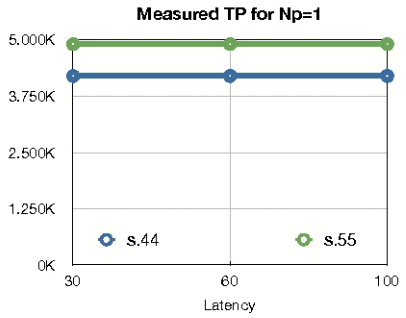
		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.44					
	30	4178816	4196116	-0.41	8.1		
Latency for	60	4178936	4196116	-0.44	19.3		
$N_p = 1$	100	4179096	4196116	-0.4	34		
	30	2089860	2105650	-0.7	16.2	1.99	0.99
Latency	60	2089980	2105650	-0.7	38.4	1.99	0.99
for $N_p = 2$	100	2090140	2105650	-0.75	67.9	1.99	0.99
	30	1033102	1066750	-3.15	32	3.93	0.98
Latency	60	1033222	1066750	-3.14	75.8	3.93	0.98
for $N_p = 4$	100	1033382	1066750	-3.12	134.12	3.93	0.98
		s.55					
	30	4883041	4902692	-0.4	6.97		
Latency for	60	4883131	4902692	-0.39	16.5		
$N_p = 1$	100	4883251	4902692	-0.39	29.18		
	30	2441829	2459153	-0.7	13.9	1.99	0.99
Latency	60	2441919	2459153	-0.7	32.88	1.99	0.99
for $N_p = 2$	100	2442039	2459153	-0.7	58.18	1.99	0.99
	30	1221242	1245064	-1.91	4.2	27.5	0.98
Latency	60	1221332	1245064	-1.9	69.9	3.93	0.98
for $N_p = 4$	100	1221452	1245064	-1.89	114.9	3.93	0.98
		mean value		1.2			
		max value			134.12		

Table 10.9: Estimated and measured Temporal Performance (TP) for a HDTV input image

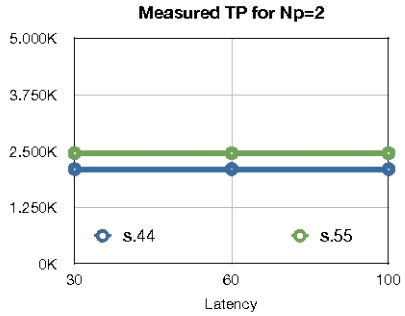
NO IM
cost - mm^2
total (combi, seq)
0,07 (0,05 - 0,02)

N_p	cost - mm^2	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.44	
1	0.5 (0.09 - 0.41)	6.9 (1.6 - 24.9)	
2	0.86 (0.13 - 0.73)	11.9 (2.23 - 44.3)	1.71 (1.44 - 1.77)
4	1.57 (0.22 - 1.35)	21.9 (4- 81.5)	3.1 (2.59 - 3.26)
		s.55	
1	0.43 (0.085 - 0.34)	6 (1.54 - 21)	
2	0.8 (0.126 - 0.68)	11.2 (2.28 - 40.9)	1.7 (1.48 - 1.78)
4	1.53 (0.22 - 1.31)	21.27 (4 - 78.8)	3.1 (2.6 - 2.81)

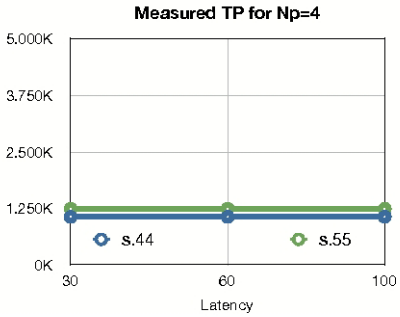
Table 10.10: Measured cost of the TPU after the RTL generation.



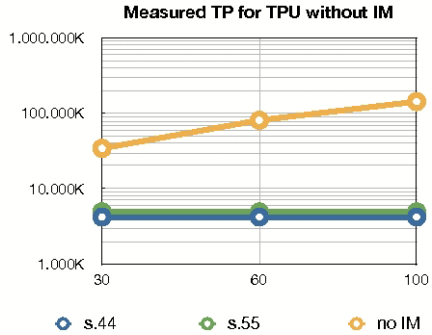
(a) $N_p = 1$



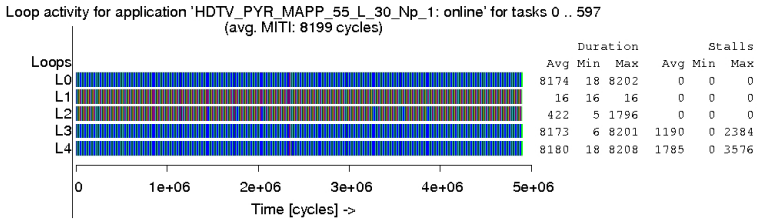
(b) $N_p = 2$



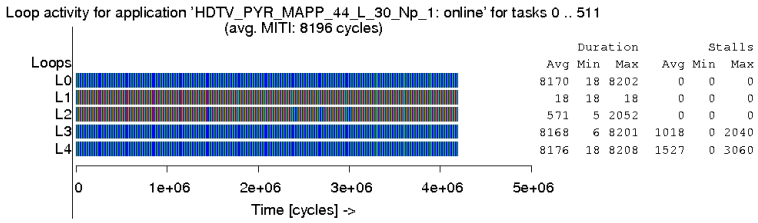
(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model



(f) Simulation of the TPU verilog model

Figure 10.9: Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV.

10.3 Conclusion

In this chapter we have presented the explorations performed for the pyramidal LOG sampling. We have explored hundreds of solutions for three input image sizes (SQCIF, VGA and HDTV). From the results we can see that the MEXP optimizations are more efficient for a Pyramidal LOG sampling than for a simple LOG sampling. In fact, the pyramidal LOG sampling has a higher number of memory access, however the area overhead is lower for the pyramidal LOG sampling because the analyzed I/O tile sizes are more adapted to the application.

The speed up of the temporal performance due to the MEXP optimizations can be up to 134.12 for a corresponding area overhead of 21.9 (results observed for the HDTV input image with a parallelism level $N_p = 4$). The highest speed up observed, without considering the parallelism is up to 34 for a corresponding area overhead of 6.9 (for a HDTV input image).

The parallelism is not efficient in the case of a SQCIF input image and is very efficient for both the VGA and the HDTV input image (E>0.9).

Chapter 11

The Polar Transform

The polar transform is a projection of the input image which changes the coordinates of the input pixels from the Cartesian to the polar ones. The corresponding formulae are:

$$x = \rho * cos(\theta)$$
$$y = \rho * sin(\theta) \tag{11.1}$$

Where x, y, ρ, θ are as shown in figure 11.1.

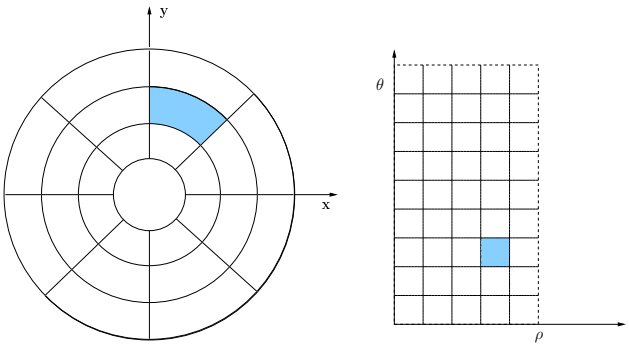


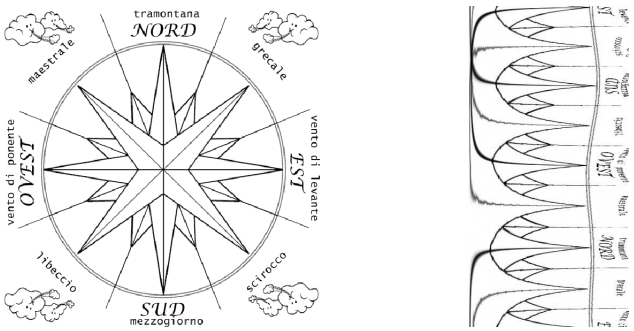
Figure 11.1: Example of a polar transform

This transformation emulates the projection of a scene on the visual cortex. And thanks to it, it is possible to easily code a rotations of the objects in the visual scene as a translation.

Figure 11.2 gives an example of a polar transform input and output image. In our hardware implementation the formula 11.1 are implemented through two LUTs whose sizes are fixed at the value Θ_{max}. In these experiments, we have chosen $\Theta_{max} = 128$.

11.1 The TPU synthesizable C-model of the Polar transform

Figure 11.3 gives the code of the REQ and CALC functions for the Polar transform.



(a) input (b) output

Figure 11.2: Example of an input and output images for a Polar transform

In the code, LUT_0 realizes the $sin(\theta)$ and LUT_1 realizes $cos(\theta)$.

From the output tile coordinates (OT_x and OT_y in the code), REQ computes the absolute coordinates of the desired pixel in the output image. By accessing the LUT_0 and LUT_1, REQ computes the input pixel coordinates (x and y in the code).

As these coordinates are not integer, REQ requests the coordinates of the input pixels whose coordinates are the superior and inferior integer of x and y.

In total REQ sends 4 requests to FETCH, which responds by sending the 4 corresponding data to CALC.

CALC receives the 4 data, performs a bilinear interpolations and sends out the computed output datum and the coordinates in the output image, where it has to be stored.

11.2 The MEXP analysis on the Polar transform

We have run the MEXP analysis for three input squared images: 128×128, 300×300 and 600×600. As $\Theta_{max} = 128$, the output image sizes I_O, corresponding to the analyzed inputs, are $\frac{width}{2} \times 128$.

The following table gives, for each analyzed input image size, the size of the produced output image (I_O) , the size of the analyzed spaces (S^I and S^O) which are power of two and contain respectively the the input and output used images, the volumes of the input and output tiles V_{IT} and V_{OT}, which will produce several tiling layouts, and the number N_s of analyzed couples of I/O tilings.

I_I	I_O	S^I	S^O	V_{IT}	V_{OT}	N_s
128×128	64×128	128×128	64×128	128, 64,32	128,64,32	132
300×300	150×128	512×512	256×128	512,256	512,256	156
600×600	300×128	1024×1024	512×128	1024,512	1024,512	210

Table 11.1: Experiments run for different input image sizes

11.2.1 128×128 input image

In this paragraph we will describe the results of the MEXP exploration for a 128×128 input image with I/O tile sizes of 128, 64 and 32. The design space contains 396 possible solutions, analyzed for 3 values of external memory latency.

Figure 11.4 gives the scatter representation of the MEXP explorations. The temporal performance of the solutions varies with the increasing latency. The parallelism for $N_p = 4$ is not efficient and for $N_p = 2$ is efficient only for the pareto solutions. Some pareto solutions are not selected because they hit the user constraint on the maximum of internal memory.

Table 11.2 gives two pareto solutions (s.38 and s.96) of the space and a non-pareto solution (s.86); the pareto solution s.38 has a better temporal performance and the solution s.96 uses less internal memory than s.86. The non-pareto solution s.86 has a trivial layout, while the pareto solutions have a non-trivial layout.

sol.	(IT,OT)	Area	Est. TP (for Latency)		
			30	60	100
s.38	(8×16, 16×4)	14016	32848	34783	41500
s.96	(8×8, 32×1)	9818	33713	40073	56393
...					
s.86	(8×8, 8×8)	14480	32980	38460	54540

Table 11.2: Examples of solutions n the analyzed space

From **table 11.3**, we can infer that:

- The speed up due to the MEXP optimizations variates between 5.9 and 31.7.
- The mean error of the MEXP estimations is 8.6%.
- The parallelism efficacy is around 0.7 for $N_p = 2$ and inferior than 0.5 for $N_p = 4$. Thus for $N_p = 4$ the parallelism is not efficient.

From **table 11.4** we can infer that:

- The maximum area overhead due to the MEXP optimizations is of 7.68.
- The parallelism has a maximum area overhead of 1.6 for $N_p = 2$ and of 2.9 for $N_p = 4$.

Figures 11.5(a), 11.5(b) and 11.5(c) show that, for $N_p = \{1\ 2\}$, the ratio of the time to prefetch with respect to the time to compute is around 1. For $N_p = 4$, this ratio is higher than 1. In fact, figures 11.5(e) and 11.5(f) show that, for both the solutions and $N_p = 1$, the average time to compute is preponderant with respect to the average time to pre-fetch. But, for solution s.96, the maximum time to pre-fetch is higher than the maximum time to compute. **Figure 11.5(d)** confirms that the speed up due to the MEXP optimizations is up to 31.7.

		Measured TP NO IM (cycles)
Latency	30	213028
	60	458788
	100	766468

		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.38					
	30	32848	34324	-4.3	6.2		
Latency for	60	34783	34582	0.58	13.26		
$N_p = 1$	100	41500	37082	11.9	21.2		
	30	16704	17655	-5.38	12.06	1.94	0.97
Latency	60	20288	19402	-4.56	23.64	1.78	0.89
for $N_p = 2$	100	28594	24659	15.9	31.89	1.5	0.75
	30	10468	11688	-10.3	18.26	2.94	0.73
Latency	60	20288	17097	-5.3	26.8	2	0.5
for $N_p = 4$	100	28594	24811	-1.7	31.7	1.5	0.37
		s.96					
	30	33713	36097	-6.6	5.9		
Latency for	60	40073	39214	2.1	11.7		
$N_p = 1$	100	56393	45894	22.8	17.13		
	30	18174	36097	-8.5	10.7	1.81	0.9
Latency	60	26957	39214	4.1	17.7	1.51	0.75
for $N_p = 2$	100	41557	45894	16.11	21.9	1.28	0.64
	30	13055	15872	-17.74	13.42	2.27	0.56
Latency	60	22312	25522	-12.6	17.98	1.53	0.38
for $N_p = 4$	100	34872	36202	-3.7	21.72	1.26	0.31
		mean value		8.58			
		max value			31.7		

Table 11.3: Estimated and measured Temporal Performance (TP) for an input image containing $128x128$ pixels

NO IM
cost - mm^2
total (combi, seq)
0,028 (0,02 - 0,008)

N_p	cost - mm^2	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.38	
1	0.075 (0.04 - 0.035)	2.68 (2.11 - 4.2)	
2	0.12 (0.07 - 0.05)	4.3 (2.1 - 7.2)	1.6 (1.52 - 1.7)
4	0.22 (0.12 - 0.1)	7.77 (3.7 - 13.4)	2.9 (2.68 - 3.2)
		s.96	
1	0.07 (0.04 - 0.03)	2.48 (2.03 - 3.68)	
2	0.11(0.06 - 0.05)	4.2 (3.2 - 6.8)	1.68 (1.58 - 1.84)
4	0.2 (0.11 - 0.084)	7.1 (5.6 - 10.98)	2.86 (2.78 - 2.97)

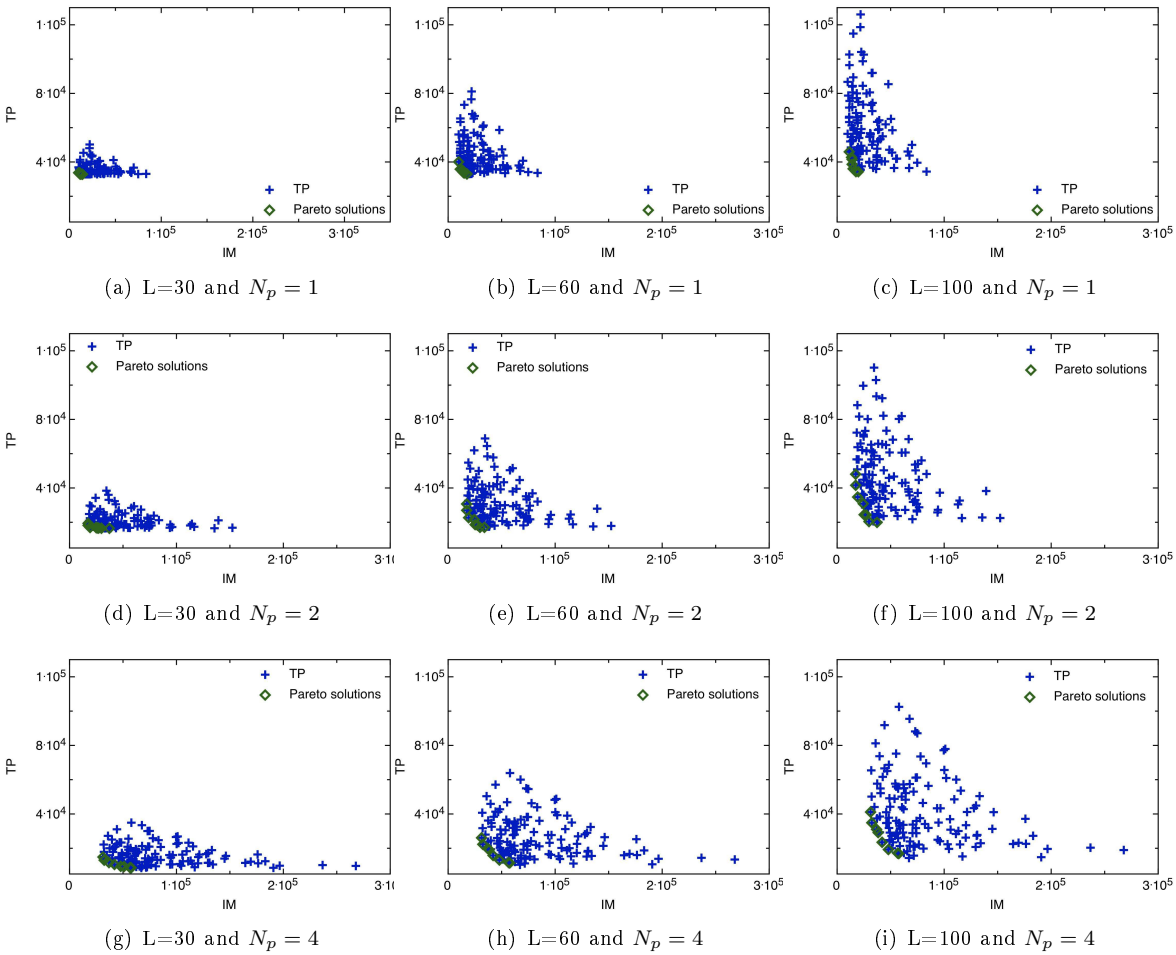
Table 11.4: Measured cost of the TPU after the RTL generation.

```
#define MASK(L)(1<<L)-1
#define N_PIXEL 4
void REQ( int i, int j , int np , int *DX, int *ADD_REQ){
.... declarations ....
OTy = ((OT_order[np]-1)&MASK(N_out_0))<<N_OT_0;
OTx = ((OT_order[np]-1)>>N_out_0)<<N_OT_1;

a0 = i + OTx;
a1 = j + OTy;

x=a1*LUT_0[a1];
y=a1*LUT_1[a1];
dx=x&MASK(PREC);
dy=y&MASK(PREC);
x1=x>>PREC +I_WIDTH_0>>1;
y1=y>>PREC +I_WIDTH_0>>1;

  if(x1 < I_WIDTH_0 && y1 < I_WIDTH_1){

    DX[0] = (dx<<16)|dy;
    ADD_REQ[0] = (x1<<16)|y1;
    ADD_REQ[1] = ((x1+1)<<16)|y1;
    ADD_REQ[2] = (x1<<16)|(y1+1);
    ADD_REQ[3] = ((x1+1)<<16)|(y1+1);

  }else{
#pregma unroll z
  for(z=0;z<N_PIXEL;z++)
    ADD_REQ[z]=0xFFFFFFFF;
  }
}
}
```

```
void CALC(int i, int j, int np, int *DX,
      int *DATA, int *out_ADD, int *out_DATA){
.... declarations ....
OTy = ((OT_order[np]-1)&MASK(N_out_0))<<N_OT_0;
OTx = ((OT_order[np]-1)>>N_out_0)<<N_OT_1;
dy = DX [0] &MASK(16);
dx = (DX [0]>> 16) &MASK(16);
*out_DATA = Interpol(dx, dy, DATA[0],
      DATA[1], DATA[2], DATA[3]);
*out_ADD = ((i+oTx)<<16)|(j+oTy); }
```

```
int Interpol(int dx, int dy, int a, int b, int c, int d){
val = (1-dx)*(1-dy)*a+dx*(1-dy)*b+(1-dx)*dy*c+dx*dy*d;
}
```

Figure 11.3: Polar transform TPU code. All the macros and global variables, shared between the user-defined and MEXP generated code are in bold.

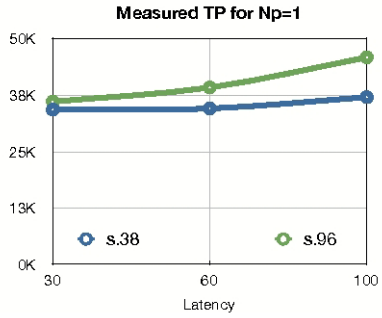


(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

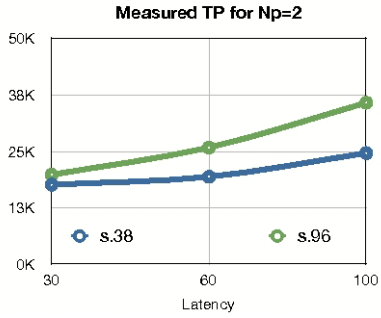
(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

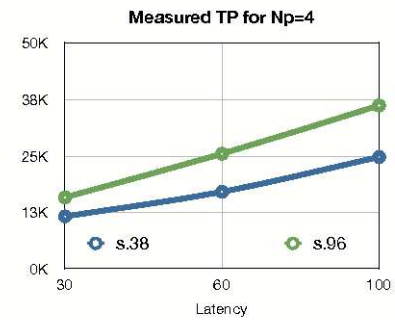
Figure 11.4: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a 128×128.



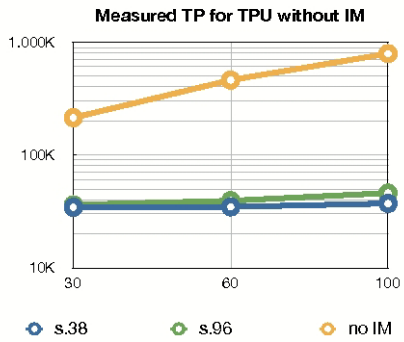
(a) $N_p = 1$



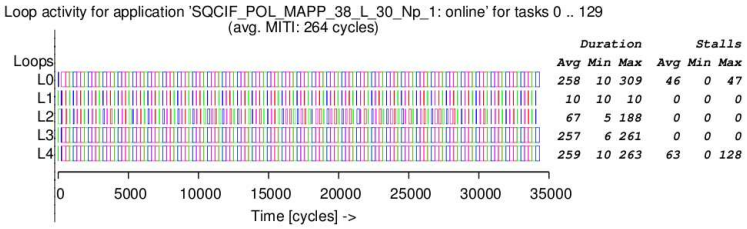
(b) $N_p = 2$



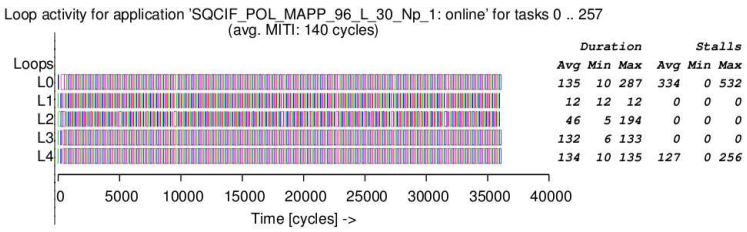
(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model



(f) Simulation of the TPU verilog model

Figure 11.5: Measured temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a 128×128.

11.2.2 300×300 input image

This paragraph describes the results of the MEXP exploration for a input image of 300×300 pixels, with I/O tile sizes of 512 and 256. The design space contains 468 possible solutions, analyzed for 3 values of external memory latency.

Figure 11.6 gives the scatter representation of the MEXP explorations.
From this figure we can infer that:

- When $N_p = 1$, the temporal performance of the solutions does not vary with the increasing latency.
- The temporal performance is improved by the parallelism (especially for the pareto solutions).

Table 11.5 compares with each other the MEXP estimations on the area occupancy and the temporal performance for two pareto solutions (s.68 and the s.71) with $N_p = 1$.

sol.	(IT,OT)	Area	Est. TP (for Latency)		
			30	60	100
s.68	(32×16, 64×4)	77688	98528	98618	98936
s.71	(32×16, 8×32)	168932	78682	79312	80472

Table 11.5: Examples of explored solutions

The solution s.68 is pareto with respect to the area occupancy and the solution s.71 is pareto with respect to the temporal performance.

From **table 11.6**, we can infer that:

- The speed up due to the MEXP optimizations variates between 5 and 40.9.
- The mean error of the MEXP estimations is 6.14%.
- The parallelism efficacy is around 0.9 except for $N_p = 4$.

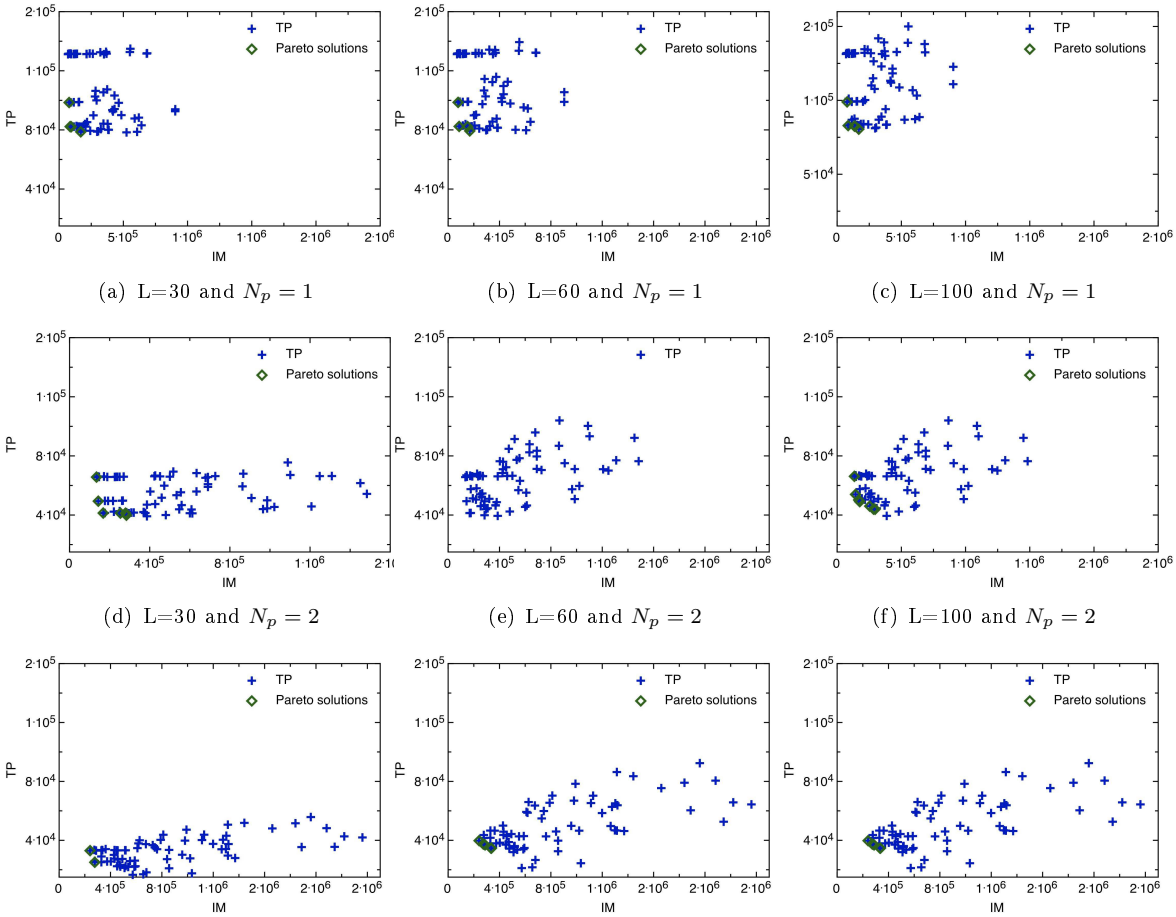
From **table 11.7** we can infer that:

- The maximum area overhead due to the MEXP optimizations is of 30.48 for the solution s.71 and of 21.9 for the solution s.68.
- The parallelism has a maximum area overhead of 1.9 for $N_p = 2$ and of 3.4 for $N_p = 4$.

Figures 11.7(a), 11.7(b) and 11.7(c) confirm that the solution s.71 has better temporal performance than the solution s.68. They also show that the temporal performance of solution s.68 does not vary with the increasing latency and when $N_p = \{1\ 2\}$. For

$N_p = 4$ and the external memory latency of 100 cycles, the ratio of the time to pre-fetch with respect to the time to compute is higher than 1.

Figure11.7(d) shows that the speed up due to the MEXP optimization is up to 40.9. Figures 11.7(e) and 11.7(f) confirm that, for both the solutions, $N_p = 1$ and a latency of 30 cycles, the average time to pre-fetch (loop L_2) is shorter than the average time to compute.



(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

Figure 11.6: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image contains 300×300 pixels.

		Measured TP NO IM (cycles)
Latency	30	499236
	60	1075236
	100	1843236

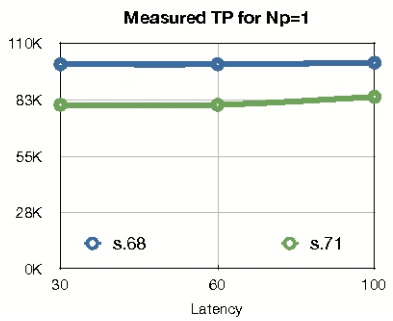
		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.68					
Latency for $N_p = 1$	30	98528	99831	-1.3	5		
	60	98618	99831	-1.2	10.8		
	100	98936	100631	-1.7	18.3		
Latency for $N_p = 2$	30	49379	50783	-2.8	9.8	1.97	0.98
	60	49469	50783	-2.6	21.2	1.97	0.98
	100	53947	54713	-1.4	33.7	1.84	0.92
Latency for $N_p = 4$	30	25980	27884	-6.8	17.9	3.58	0.9
	60	25980	27884	-6.8	38.6	3.58	0.9
	100	41532	46565	-10.8	39.6	2.16	0.5
		s.71					
Latency for $N_p = 1$	30	78682	80085	-1.8	6.2		
	60	79312	80085	-1	13.4		
	100	80472	83966	-4.2	22		
Latency for $N_p = 2$	30	39866	41934	-4.9	11.9	1.9	0.95
	60	39866	41934	-4.9	25.6	1.9	0.95
	100	44000	49328	-10.8	37.4	1.7	0.85
Latency for $N_p = 4$	30	22158	25217	-12.1	19.8	3.2	0.8
	60	22158	25217	-12.1	42.6	3.2	0.8
	100	34344	45028	-23.7	40.9	1.9	0.5
		mean value		6.16			
		max value			40.9		

Table 11.6: Estimated and measured Temporal Performance (TP) for an input image containing $300x300$ pixels.

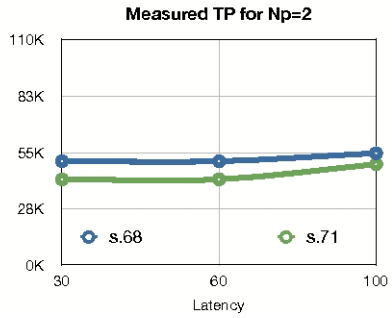
NO IM
cost - mm^2
total (combi, seq)
0,03 (0,02 - 0,01)

N_p	cost - mm^2	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.68	
1	0.22 (0.04 - 0.78)	7.76(2.2 - 24.2)	
2	0.36 (0.07 - 0.29)	12.6 (3.3 - 40.3)	1.63 (1.55 - 1.66)
4	0.63 (0.13 - 0.5)	21.9 (6.2 - 68.7)	2.83 (2.8 - 2.84)
		s.71	
1	0.27(0.05 - 0.22)	9.4 (2.5 - 29.9)	
2	0.49 (0.07 - 0.42)	17 (3.6 - 57)	1.81 (1.43 - 1.9)
4	0.87(0.13 - 0.74)	30.5 (5.8 - 103.3)	3.2 (2.35 - 3.45)

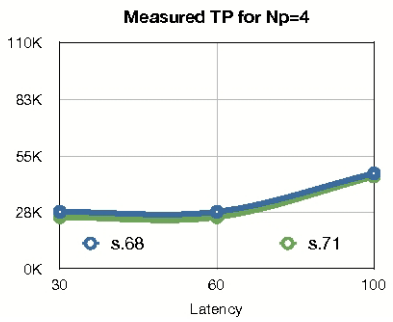
Table 11.7: Measured cost of the TPU after the RTL generation.



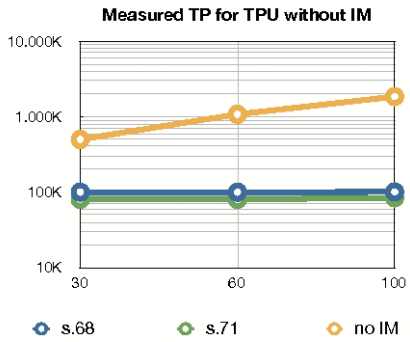
(a) $N_p = 1$



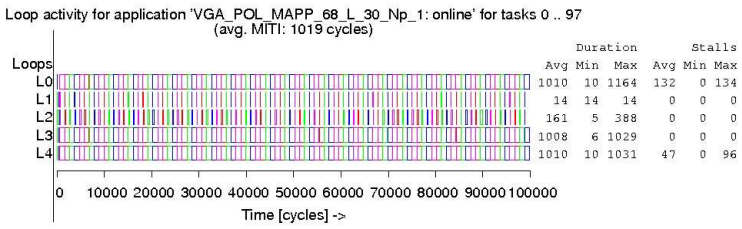
(b) $N_p = 2$



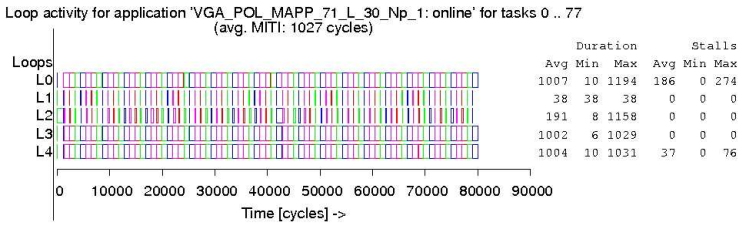
(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model



(f) Simulation of the TPU verilog model

Figure 11.7: Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA.

11.2.3 600×600 input image

This paragraph describes the results of the MEXP exploration for an input image containing 600×600 pixels and with two input and output tile sizes of 1024 and 512. The design space contains 630 possible solutions, analyzed for 3 values of external memory latency.

Figure 11.8 gives the scatter representation of the MEXP explorations.

From this figure we can see that the Temporal Performance of the solutions does not depends on the latency variations and that the parallelism is efficient.

Table 11.8 compares the MEXP estimations on the area occupancy and the temporal performance for two pareto solutions: the s.197, which is pareto with respect to the area occupancy, and the s.200, which is pareto with respect to the temporal performance. In this table, $N_p = 1$.

sol.	(IT,OT)	Area	Est. TP (for Latency)		
			30	60	100
s.197	(16×32, 256×2)	150816	260556	262944	262944
s.200	(16×32, 32×16)	183320	162262	173082	173082

Table 11.8: Examples of solutions from the analyzed space

From **table 11.9**, we can infer that:

- The speed up due to the MEXP optimizations variates between 3.77 and 30.61.
- The mean error of the MEXP estimations is 13.78%.
- The parallelism efficacy is around 0.99, except for $N_p = 4$.

From **table 11.10** we can infer that:

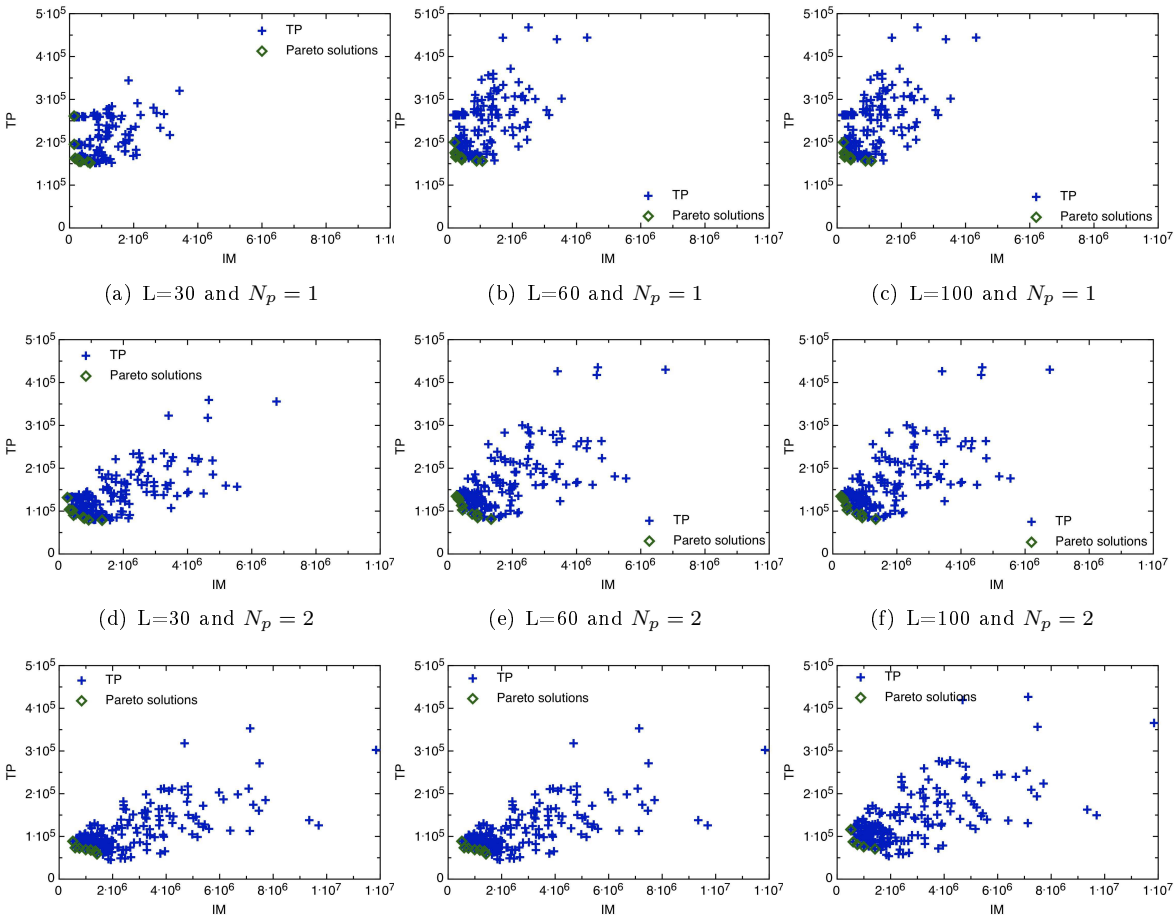
- The maximum area overhead due to the MEXP optimizations is of 34.18.
- The parallelism has a maximum area overhead of 1.9 for $N_p = 2$ and of 3.7 for $N_p = 4$

From **figures 11.9(a), 11.9(b) and 11.9(c)** we can see that:

- For solution s.197, the ratio of the time to-prefetch with respect to the time to compute is inferior than 1 for $N_p = 1$ and $N_p = 2$ and is higher 1 for $N_p = 4$ and a latency of 100 cycles.
- For solution s.200, the ratio of the time to-prefetch with respect to the time to compute is inferior around 1 for $N_p = 1$ and it is higher 1 for $N_p = \{2\ 4\}$ and a latency of 100 cycles.

Figures 11.9(e) and 11.9(f) confirm that the average length of the pre-fetching is shorter than the average length of the other loops, for both solution, with $N_p = 1$ and a latency of 30 cycles. However the prefetching length is more important for solution s.200, which explains the difference with respect to the other solution.

Figure 11.9(d) confirms that the temporal performance is increased of a factor 34.18 with respect to a solution without any internal memory.



(a) L=30 and $N_p = 1$ (b) L=60 and $N_p = 1$ (c) L=100 and $N_p = 1$

(d) L=30 and $N_p = 2$ (e) L=60 and $N_p = 2$ (f) L=100 and $N_p = 2$

(g) L=30 and $N_p = 4$ (h) L=60 and $N_p = 4$ (i) L=100 and $N_p = 4$

Figure 11.8: DSE for different external memory latency (L) and for different number of parallel pipelines (N_p).

		Measured TP NO IM (cycles)
Latency	30	9984
	60	2150415
	100	3686415

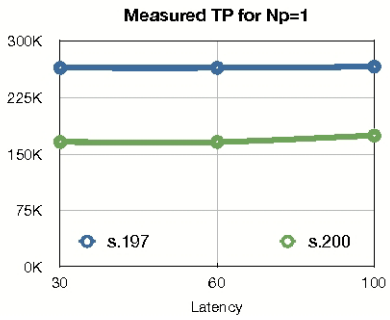
		Est. TP (cycles)	Meas. TP (cycles)	error (%)	MEXP SU	Parall. SU	Parall. E
		s.197					
	30	260556	264531	-1.5	3.8		
Latency for	60	260556	264531	-1.5	8.1		
$N_p = 1$	100	262944	266131	-1.2	13.9		
	30	131656	133661	-1.5	7.5	1.98	0.99
Latency	60	131656	133661	-1.5	16.1	1.98	0.99
for $N_p = 2$	100	134624	138538	-2.8	26.6	1.92	0.96
	30	88571	70244	26.1	142	3.77	0.94
Latency	60	88571	70244	26.1	30.6	3.77	0.94
for $N_p = 4$	100	116131	125659	-7.6	29.3	2.12	0.53
		s.200					
	30	162262	166097	-2.31	6		
Latency for	60	162262	166097	-2.3	12.95		
$N_p = 1$	100	173082	174771	-0.9	21.09		
	30	100911	85663	17.8	1.6	1.94	0.97
Latency	60	100911	85663	17.8	25.1	1.94	0.97
for $N_p = 2$	100	126238	130112	-2.98	28.3	1.34	0.67
	30	91396	56631	61.39	17.63	2.93	0.73
Latency	60	91396	56631	61.39	37.97	2.93	0.73
for $N_p = 4$	100	120276	135675	-11.35	27.17	1.29	0.32
		mean value		13.78			
		max value			30.6		

Table 11.9: Estimated and measured Temporal Performance (TP) for an input image containing $600x600$ pixels.

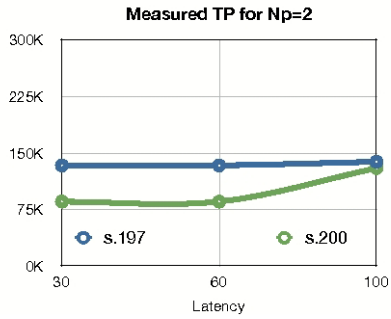
NO IM
cost - mm^2
total (combi, seq)
0,03 (0,02 - 0,01)

N_p	cost - mm^2	MEXP AO (nX)	Parall. AO (nX)
	total (combi, seq)	total (combi, seq)	total (combi, seq)
		s.197	
1	0.24 (0.05 - 0.19)	7.9 (2.22 - 24.8)	
2	0.35 (0.07 - 0.27)	11.4 (3.27 - 35.4)	1.43(1.47 - 1.42)
4	0.64 (0.132 - 0.52)	21 (5.27- 66.3)	2.65 (2.59 - 2.67)
		s.200	
1	0.29 (0.05 - 0.24)	9.6 (2.31 - 31.16)	
2	0.54 (0.08 - 0.46)	17.7 (3.55 - 59.6)	1.8 (1.53 - 1.91)
4	1.05 (0.144 - 0.9)	34.18 (6.29 - 116.43)	3.55 (2.7 - 3.7)

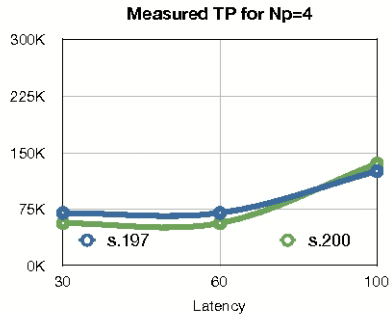
Table 11.10: Measured cost of the TPU after the RTL generation.



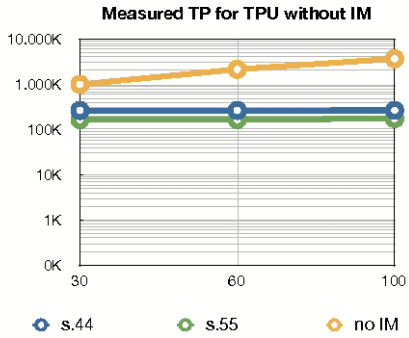
(a) $N_p = 1$



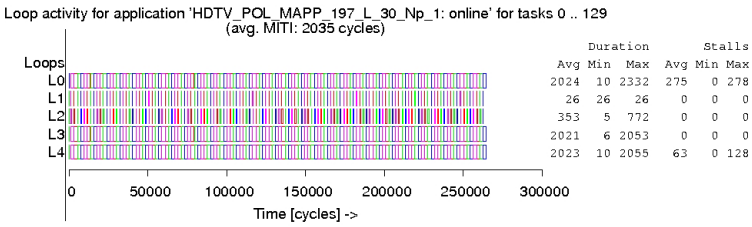
(b) $N_p = 2$



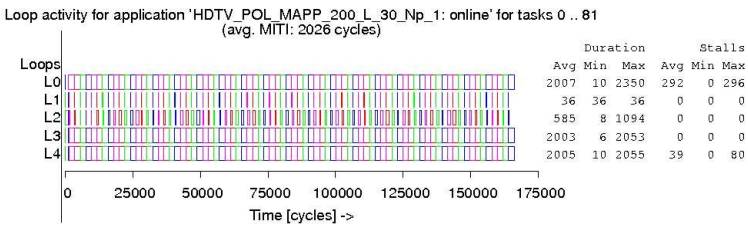
(c) $N_p = 4$



(d) No Internal Memory vs MEXP solutions



(e) Simulation of the TPU verilog model



(f) Simulation of the TPU verilog model

Figure 11.9: Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p)

11.3 Conclusion

In this chapter we have presented the explorations performed for the polar tansform. We have explored hundreds of solutions for three input image sizes ($128x128$, $300x300$ and $600x600$). From the results we can see that the speed up of the temporal performance due to the MEXP optimizations can be up to 40.9 for a corresponding area overhead of 30.5 (results observed for the $300x300$ input image with a parallelism level $N_p = 4$). The highest speed up observed, without considering the parallelism is up to 22 for a corresponding area overhead of 9.4 (for a $300x300$ input image).

The parallelism is not efficient in the case of a $128x128$ input image and is very efficient for both the $300x300$ and the $600x600$ input image (E>0.9).

The mean error on the MEXP estimations is around 10%.

Conclusion

The aim of this Ph.D. thesis is to study a methodology that improves the data transfer and management for application having non-affine array references.

The target applications are iterative image processing algorithms which are non-recursive and have static dependences. These applications are well described by a loop based C-code and they can undergo a High Level Synthesis (HLS) which infer a RTL model from an input C-code.

The input code of the HLS can be optimized, by the loop transformations, with respect to its data storage and management. In fact, these transformations enhance the data locality and allow, through the data partitioning, to achieve the computation parallelism and the data prefetching. In particular the data and computation partitioning is achieved through a transformation called tiling. Thanks to the loop transformations, some existing HLS tools are able to generate a high quality hardware.

In the first part of this dissertation, we have presented the context of the problem and the previous related works. In particular, we have underlined that the existing methods to optimize the data transfer and management are not adapted to the application with non affine array references.

In order to provide a solution to analyze and optimize the application with non-affine array references we have developed a tool called MEXP (Memory EXPloration).

MEXP is a Design Space Exploration tool aimed to find an adapted I/O tiling for image processing with non-affine array references. The aim is to to partition the data and the computations of the application in order to balance the input data pre-fetching and the output computations. The corresponding hardware is generated from a customizable model called Tile Processing Unit (TPU). In the chapters 2 and 3 we have described the tool flow and the customizable target hardware.

MEXP has the advantage of adapting the choice of a couple of I/O tiling to the non-affinity of the application array references. An appropriate couple of I/O tiling can completely mask the time to prefetch and ensure the invariance of the TPU temporal performance with respect to the external memory latency. In chapter 4 we have described the method used to construct a set of possible couples of I/O tiling. The analysis takes into account two user-specified functions, which describes the target application, and several parameters, which tailor the MEXP Design Space Exploration.

MEXP is also able to re-schedule the output tiles computations in order to reduce the the data transfer from the external memory. This permits a reduction of the power con-

sumption and an improvement of the TPU temporal performance. Chapter 5 describes the method used to re-schedule the output tiles computations.

The TPU accesses the data thanks to a table giving the mapping between the input tiles and the internal buffers. This table is generated by MEXP for each chosen solution.

MEXP is able to explore and optimize hundreds of solutions. It classifies these solutions with respect to two metrics: the estimated temporal performance and the used internal memory of the corresponding hardware.

Chapters 6 and 7 give the methods to compute the memory mapping and to perform the Design Space Exploration.

MEXP also evaluates the possibility to parallelize the computations of several output tiles by instantiating parallel hardware into the TPU. The corresponding Computation Mapping is described in chapter 6.

In chapters 9, 10 and 11 we describe the experiments performed for three target applications: the LOG sampling, the pyramidal LOG sampling and the polar transform. The experiments show that the MEXP optimizations depend on the I/O tile sizes and on the I/O data space sizes. The results are evaluated with respect to the Speed Up that the MEXP optimizations permits on the Temporal Performance and the area overhead required to implement the optimizations. The highest Speed Up observed is up to 134.12 for a corresponding area overhead of 21.9 (results observed for a pyramidal LOG sampling computing 4 output tiles in parallel and for an external memory latency of 100 cycles). The performed experiments also show that, thanks to a good choice of couple of I/O tiling, the balance between the prefetching and the computing can be achieved.

A first limitation of MEXP is that the generator of synthesizable C-code is only exploitable for the studied HLS tool, while the MEXP analysis can be extended to other HLS tools or to map image processing on other kind of architectures and programmable processors. In this case, the generic template of the synthesizable C-code should be adapted to the target architecture.

Another limitation of the current version of MEXP, is that it only applies to a single step image processing, i.e. to a transformation computing an output from an input image by applying a single transformation. The analysis is not applicable to a multi-step dependent application, i.e a transformation applied through a chain of several inter-dependent steps. The adaptation will require several transformations in the MEXP target architecture and flow. A possible solution could be to instantiate a TPU per each step in the application. A controller would synchronize the TPUs start and communication. The TPU architecture could remain the same except that the output tiles scheduling of the first TPU would be related to the output tile scheduling of the second TPU.

Appendix

Appendix A

The bilinear interpolation

When the law computing the input from the output pixels coordinates is non-affine, the result of the computation may be not integer. But the pixels coordinates are integers, thus it is mandatory to round up or down the computation results. As a consequence, this introduces an aliasing.

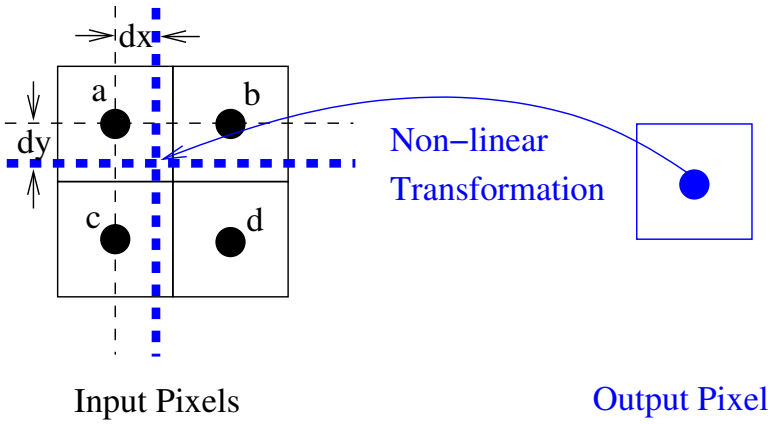


Figure A.1: Example of a bilinear interpolation.

In order to limit the aliasing it is possible to perform a bilinear interpolation, whose mechanism is shown in figure A.1. Given an output pixel, to which the non-linear transformation associates non-integer input coordinates (the spotted bold lines in the figure), the bilinear interpolation computes the output value by a linear weighted combination of the 4 input pixels surrounding the non-integer coordinates (a, b, c and d in the figure).

The weighted linear combination of the input pixels values depends on the distances (dx and dy in the figure) between the point with non-integer coordinates and the point with the inferior integer coordinates. It is performed as follows:

$$val = (1 - dx) * (1 - dy) * a + dx * (1 - dy) * b + (1 - dx) * dy * c + dx * dy * d$$

Figure A.2 compares two figures: figure A.2(a) is obtained by sampling the input pixels having the inferior integer coordinates and figure A.2(b) is obtained by using a bilinear interpolation. The improvements due to the bilinear interpolation are visible.

(a) without bilinear interpolation (b) with a bilinear interpolation

Figure A.2: Magnification of a detail of an output image showing the anti-aliasing effect of the bilinear interpolation

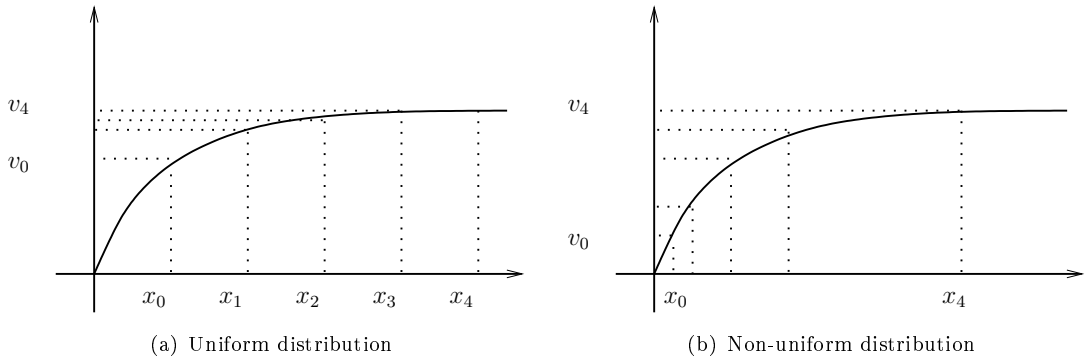
Appendix B

The hardware implementation and the Look-Up Table

B.1 Recalls on the Look-Up tables

One of the fastest and easier way to realize non-linear function is to use a Look-up table (LUT), since it is often faster to retrieve a datum from a memory then to undergo “expansive" computations . A LUT is a memory which takes as input a set of samples of a function domain and gives as output the corresponding function values.

The precision of the LUT depends on the number of the samples taken into the function domain and thus on the LUT size; but it also depends on the uniformity or not of the samples distribution. Figure B.1 shows how two LUTs with the same number of entries can have a different precision due to the fact that the samples distribution follows the non-linearity of the function to be realized. When the samples distribution is not uniform it is mandatory to provide the LUT with an address decoder.



(a) Uniform distribution (b) Non-uniform distribution

Figure B.1: Example of two different LUTs realizations with a different distribution of the samples into the function domain. The LUT entries are data in the intervals $[x_i, x_{i+1}[$ and the LUT output are exactly the data v_i, with $i \in [0, 4]$.

Real computations are often made on values which are different from the chosen

samples x_i, thus it is necessary to round the manipulated values to one of the LUT possible entries. Given x an input value so that $x_i \leq x \leq x_{i+1}$, x can be rounded down to x_i or up to x_{i+1}.

An efficient way to improve the LUT precision and to maintain low its size is to approximate the function to be realized by a piecewise linear function, i.e. given two successive samples of the function domain x_i and x_{i+1}, the function is approximated by the line passing through v_1 and v_2. In this case, given x an input value so that $x_i \leq x \leq x_{i+1}$, the LUT output will be a weighted linear interpolation of the values v_1 and v_2.

In the hardware realization of the log sampling, a LUT has been used to compute the input pixels coordinates. In order to simplify the mechanism used to access the LUT we have chosen a uniform distribution of the function domain samples. On the other hand to improve the LUT precision we have approximated the pseudo-logarithm with a piecewise linear function.

B.2 The LUT associated to the LOG sampling

The function to be realized is:

$$f(x_{out}, y_{out}) = \frac{\rho_0}{\rho_{lim} - \rho_{out}} \tag{B.1}$$

with $\rho_0 = max(\frac{W_{out}}{2}, \frac{H_{out}}{2})$ and $k = 2$ (i.e. $\rho_{lim} = \rho_0$)[a].

The formula B.1 can be expressed as:

$$f(x_{out}, y_{out}) = \frac{\rho_0}{\rho_{lim} - \sqrt{(x_{out} - \frac{W_{out}}{2})^2 + (y_{out} - \frac{H_{out}}{2})^2}} \tag{B.2}$$

where $e(x_{out}, y_{out}) = (x_{out} - \frac{W_{out}}{2})^2 + (y_{out} - \frac{H_{out}}{2})^2$ is taken as the entry of the LUT and if we suppose that $\frac{W_{out}}{2} > \frac{H_{out}}{2}$, then the entry of the LUT is a radial symmetric function which varies in the following interval: $0 \leq e(x_{out}, y_{out}) < 2 * (\frac{W_{out}}{2})^2$. Figure B.2 gives two examples of the entry function $e(x_{out}, y_{out})$, one for $W_{out} = H_{out} = 128$ and one for $W_{out} = 256$ and $H_{out} = 128$.

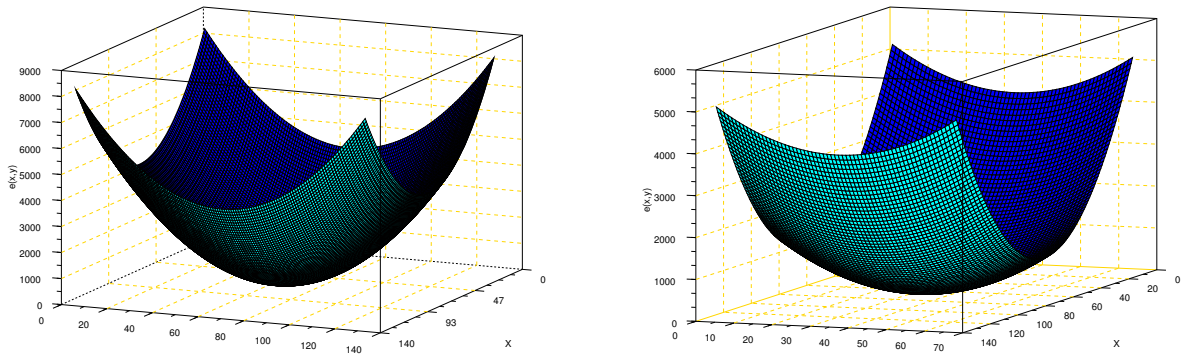
The upper bound of the entry function $e(x, y)$ is $(\frac{W_{out}}{2})^2 + (\frac{H_{out}}{2})^2$, but it is overestimated as $2 * (\frac{w}{2})^2$, where $w = max(W_{out}, H_{out})$. Thus the upper bound of the entry function depends on the input image size.

For a Given possible size for the LUT noted L_{size}, which also represents a possible number of function domain samples or a possible number of LUT entries, the uniform LUT step is defined as follows:

$$L_{step} = \frac{2 * (\frac{w}{2})^2}{L_{size}}$$

The problem is how to determine a good L_{size} that gives an efficient image quality and requires an acceptable area overhead due to the LUT utilization.

[a] This configuration reproduces the behavior of the photo-receptors in a human retina.



(a) $W_{out} = H_{out} = 128$

(b) $W_{out} = 256$ and $H_{out} = 128$

Figure B.2: The entry function of the LUT.

First of all we can imply that, as the upper bound of the entry function depends on the input image size, the LUT size should depend on it either.

The size of the LUT in hardware depends also on its depth which, in our case, is of 16 bits per LUT word. In fact, in our hardware implementations, we use a fixed point arithmetic and the numerical values are represented on 16 bits with a precision (i.e. the number of bits following the points) of 8bits.

In order to determine a good L_{size}, we have evaluated some possible values of L_{size} ($\frac{W}{4}$, $\frac{W}{2}$, W and $2W$)[b] with respect to two aspects: the precision of the approximated function and the quality of the produced output image.

The precision of the approximated function

The function to be approximated is described by the formula B.2 and by the figure B.3(a). Figure B.3(b) gives the approximated function, which, in this case, is obtained by using a LUT of 32 entries and in order to process a SQCIF input image.

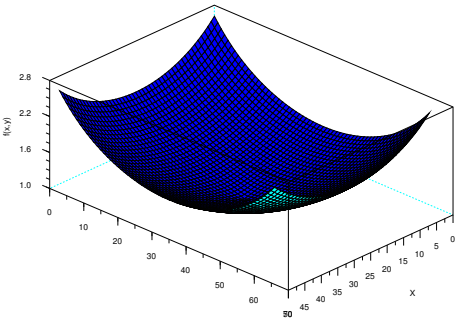
To evaluate the precision of the approximated function we have analyzed the following values:

- The Root Mean Squared Deviation (RMSD), which is a good measure of an approximation accuracy and is defined as:

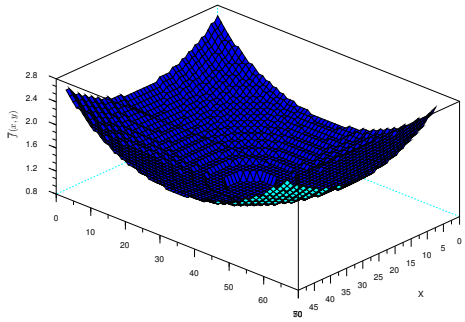
$$RMSD = \sqrt{\frac{1}{W_{out} * H_{out}} * \sum_{i=0}^{W_{out}*H_{out}} (f(x,y) - \overline{f}(x,y))^2}$$

where $f(x,y)$ and $\overline{f}(x,y)$ are respectively the function to be approximated and its

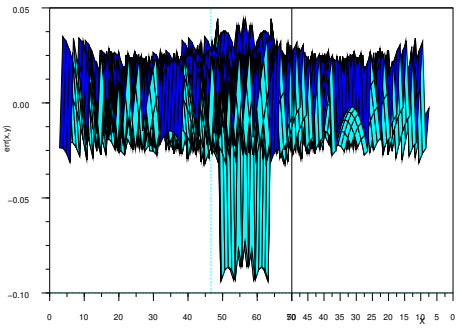
[b] $W = max(W_{out}, H_{out})$



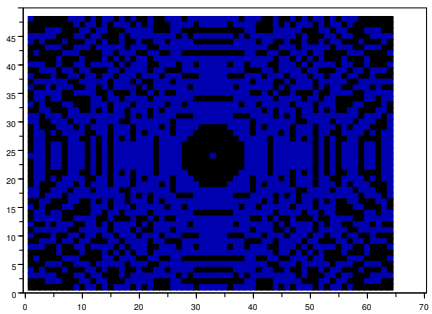
(a) Reference function



(b) Approximated function



(c) Residuals between the reference function and its approximation



(d) Distribution of the error with respect to the output pixels coordinates

Figure B.3: Example of a $f(x, y)$ approximation for a SQCIF input image (i.e. 128×96 pixels). The figure gives the error distribution on the pixels. An error on a pixel position can be at most of one along both directions x and y.

approximation, while $f(x, y) - \overline{f}(x, y)$ are called residuals. The more the RMSD is negligible, the more the approximation is accurate.

- The percentage of output pixels for which the approximated function gives an error on the estimation of the corresponding input pixels coordinates; this percentage will be noted P_ϵ.
- The distribution of these errors in the space of output pixels coordinates.
- The maximum error on the input coordinates estimation, which will be noted max_ϵ.

For example, figure B.3(c) gives the residuals between the function reference and its approximation, the residuals are more significant at the center, where the function to be approximated variates the most and, as a consequence, the LUT approximation is

the less accurate. Figure B.3(d) gives the distribution of the error with respect to the output pixel coordinates: a blue position corresponds to a correct estimation and a black position corresponds to a wrong estimation. In this case, the maximum error on the input pixels coordinates is of ± 1, which means that if there is an error, instead of sampling the correct pixel we will sample one of its 8 nearest neighbors.

	SQCIF					VGA					HDTV				
	L_{size}					L_{size}					L_{size}				
	$\frac{W}{4}$	$\frac{W}{2}$	W	$2W$	$4W$	$\frac{W}{4}$	$\frac{W}{2}$	W	$2W$	$4W$	$\frac{W}{4}$	$\frac{W}{2}$	W	$2W$	$4W$
RMSD	0,041	0,021	0,01	0,005	0,002	0,008	0,004	0,002	0,001	0,0006	0,003	0,001	0,0008	0,0004	0,0002
P_ϵ	65,45	40,32	23,23	11,41	5,54	58,69	35,37	19,99	10,87	4,96	55,74	33,11	17,49	8,93	4,49
max_ϵ	3	1	1	1	1	3	2	1	1	1	3	2	1	1	1

Table B.1: Values of the different indicators for different image sizes: SQCIF (128 × 96), VGA (640 × 320) and HDTV (1920 × 1080)

Table B.1 gives the values of the indicators RMSD, P_ϵ and max_ϵ for different values of input image SQCIF (128 × 96), VGA (640 × 320) and HDTV (1920 × 1080). These indicators have been evaluated for different sizes of LUT ($\frac{W}{4}$, $\frac{W}{2}$, W, $2W$ and $4W$)[c].

Table B.1 shows that to have an percentage of incorrect pixels inferior than 10%, a LUT of $4W_{in}$ should be used. For a HDTV input image, the LUT would have a hardware size of 0,12 mm^2[d].

In order to reduce this size, it is possible to tolerate some errors on the output image provided that its quality remains of an acceptable.

The quality of the output image

To define the quality of an image there are different indicators, we have used the Peak Signal to Noise Ratio (PSNR) and the Structural SIMiliarity (SSIM). These two indicators are used to estimate the distortions introduced by a compression algorithm.

The PSNR formula is the following:

$$PSNR = 10 * log_{10}\left(\frac{1}{W_{out} * H_{out}}\sqrt{(I(x,y) - \overline{I}(x,y))^2}\right)$$

where $I(x,y)$ and $\overline{I}(x,y)$ are the reference image and the approximated image respectively. A good quality approximation has a PSNR superior than 30dB.

While the PSNR takes into account the existing differences pixel by pixel and under the hypothesis that the human eye is more sensible to the structural changes of an image, the SSIM measures the structural similarities between the reference and approximated images.

The SSIM formula [108] is the following:

$$SSIM = \frac{(2\mu_x\mu_y + C_1)(2cov_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

[c] $W = max(W_{out}, H_{out})$ and $W_{out} = \frac{W_{in}}{k}$ with $k = 2$.

[d] This estimation has been performed for a 45 nm technology and is given by the formula $16 \times L_{size} \times 0,91$, where 16 is the number of bit per LUT word and 0,91 a factor due to the considered technology.

where $x = I(x, y)$ and $y = \overline{I}(x, y)$ are the reference image and the approximated image respectively. μ_x, μ_y, σ_x^2, σ_y^2 and σ_{xy} are the mean of x, the mean of y, the variance of x, the variance of y and the covariance of x and y.

$C_1 = (k_1 L)^2$ and $C_2 = (k_2 L)^2$ are two constants used to ensure the stability of the SSIM formula; where L is the dynamic range of the pixels values (i.e. $L = 255$ for 8 bit/pixel gray scale images); $k_1 = 0.01$ and $k_2 = 0.03$ as suggested in [108].

The SSIM variates between -1 and 1. It is 1 fro two identical image $x = I(x, y)$ and $y = \overline{I}(x, y)$.

In order to evaluate the local structural differences between the reference and the approximated image the SSIM is computed on a sliding window of 8×8 pixels and a mean, which parses the image from the top left to the bottom right corner. The quality of the whole image is evaluated as the mean of the SSIM values (MSSIM) for all possible positions of the sliding window.

	lena		saurus		cars		man	
	PSNR	MSSIM	PSNR	MSSIM	PSNR	MSSIM	PSNR	MSSIM
				SQCIF				
$\frac{W}{4}$	23.22	0.999918	26.93	0.999403	28.36	0.999957	20.96	0.981926
$\frac{W}{2}$	27.24	0.999971	32.72	0.999874	33.48	0.999984	24.73	0.995750
W	32.09	0.999996	39.26	0.999995	38.47	0.999994	28.37	0.998626
$2W$	37.52	1	45.87	0.999995	47.73	1	34.8	0.999938
$4W$	50	1	53.15	0.999999	56.16	1	49.8	0.999974
				VGA				
$\frac{W}{4}$	29.93	0.999996	33.79	0.999962	36.38	0.999996	28.13	0.951211
$\frac{W}{2}$	33.99	0.999999	39.1	0.999991	39.75	0.999998	30.63	0.957258
W	38.19	1	45.23	0.999998	43.1	1	33.99	0.959493
$2W$	40.85	1	51.13	146.19	1	35.9	0.960229	
$4W$	42.42	1	56.49	1	47.67	1	37.63	0.960313
				HDTV				
$\frac{W}{4}$	37.26	0.999999	42.84	0.999994	42.62	0.999999	33.51	0.976242
$\frac{W}{2}$	40.51	1	48.72	0.999999	45.2	1	35.58	0.9941170
W	43.94	1	54.28	1	48.1	1	38.7	0.997749
$2W$	46.15	1	58.95	1	50.4	1	40.79	0.999106
$4W$	49.13	1	62.65	1	53.42	1	43.44	0.999218

Table B.2: PSNR and MSSIM for several images of different sizes. $W = W_{out}$.

Table B.2 gives the PSNR and the SSIM for several images (lena, saurus, cars, man) with different sizes[e].

From this table we can infer that a LUT size of $\frac{W_{out}}{2}$ gives a good output image quality for each input image size tested.

Image B.4 gives an example of an output image for a VGA input image and for a LUT size of $\frac{W_{out}}{4}$ or $\frac{W_{out}}{2}$. It shows that the image distortions due to the insufficiency of LUT size are not visible for a LUT size of $\frac{W_{out}}{2}$.

[e]It was important to test for different images because the PSNR and the SSIM depend on the image content.

(a) $L_{size} = \frac{W_{out}}{4}$

(b) $L_{size} = \frac{W_{out}}{2}$

(c) reference image

Figure B.4: Output of an approximated LOG sampling applied to a VGA image and using different LUT size

Appendix C

The space-variant low-pass

The space-variant low pass used in the retina model of the GIPSA-lab is the following.

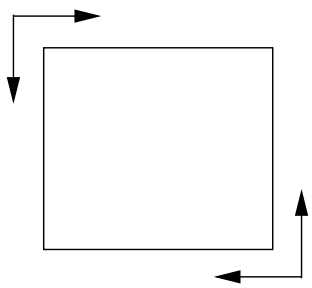


Figure C.1: Example of a spatial-variant low pass, the input image is filtered along the two directions (x,y) and in the two ways for each direction

As shown by figure C.1, the space-vatriant low pass can be divided into four filters F_1, F_2, F_3 and F_4, so that the input image is filtered along the two directions of the image plane (x,y) and in the two ways for each direction (from bottom upperwards, from up bottomwards, from left rightwards and from right leftwards).

The formula of the filters F_1, F_2, F_3 and F_4 are given in the group of equations C.1. These filters are recursive and iterative functions which are applied one after the other on an input image (noted I_0) in order to obtain an output image (noted I_4).

$$\begin{aligned}
F_1: \quad I_1(x,y) &= (1-\alpha(x,y))I_0(x,y) + \alpha(x,y)I_1(x,y-1) \\
F_2: \quad I_2(x,y) &= (1-\alpha(x,y))I_1(x,y) + \alpha(x,y)I_2(x,y+1) \\
F_3: \quad I_3(x,y) &= (1-\alpha(x,y))I_2(x,y) + \alpha(x,y)I_3(x-1,y) \\
F_4: \quad I_4(x,y) &= (1-\alpha(x,y))I_3(x,y) + \alpha(x,y)I_4(x+1,y)
\end{aligned} \tag{C.1}$$

The filtering factor $\alpha(x,y)$ variates with the excentricity of the pixel position and has

the following formula

$$\alpha(x,y) = \sqrt{\frac{|x - \frac{Height}{2}| + |y - \frac{Width}{2}|}{\frac{Height+Width}{2}}}$$

In this application there are three kinds of dependences:

- the dependence between output pixels on a line for filters F_1 and F_2, in fact these filters re-use the previously computed output pixel (on the left for F_1 and on the right for F_2) in order to compute the current output pixel.
- the dependence between output pixels on a column for filters F_3 and F_4, in fact these filters re-use the previously computed output pixel (on the top for F_1 and on the right for F_2) in order to compute the current output pixel.
- the dependences between the different filters F_1, F_2, F_3 and F_4. The order of the filters is not strict, in fact the filters can be executed in any desired order with a slight difference on the borders of the image that is acceptable for the target application. But it is mandatory to execute all the filters to obtain the desired output and, no matter which order we choose, it is necessary to store, at least once, the whole processed image in order to respect these dependences. We give some examples of the possible orders, each one of them requiring different hardware resources:
  - F_1 is applied on the first line then, after it has processed the whole first line, F_2 is applied to the first line while F_1 processes the second line. The two filters are pipelined to process the lines until the bottom line of the image is reached. After processed and store the whole image with the filter F_1 and F_2, it is possible to apply F_3 on the last column of the image then, after it has processed the whole last column, F_4 can be applied to the last column while F_3 processes the penultimate column. The two filters are pipeline to process the columns of the image until the rightest column is reached. The memory resources needed to implement this version of the low-pass are : a local memory to store the line of pixels produced by F_1 before the start of F_2, a local memory to store the column produced by F_2 before the start of F_4 and an internal memory of the size of input image to store the output of F_2 before the start of F_3.
  - F_1 and F_3 are pipelined to procces a single pixel from the top left to the bottom right of the input image, then F_2 and F_4 are pipelined to proccess a single pixel from the bottom right to the top left of the input image. The memory resources needed to implement this version of the low-pass are: two local registers to store the output of F_1 and F_2 before the start of F_3 and F_4 rispectively; an internal memory of the size of the input image to store the results of F_3 before the start of F_2.

Appendix D

A mipmapping application: the Pyramidal LOG sampling

D.1 Re-calls on the MIP mapping

The MIP mapping is a method used in 3D computer graphics to compute the texture objects. It was introduced in 1983 by Lance Williams [109].

Let consider an image representing 3D object, if the image is re-sized or the observer point of view changes, the image has to be rendered by changing the texture of the 3D objects. This will take into accounts the changes in the perspective and will increase the realism and the information content of the image.

A MIP-map is a set of filtered and down-sampled versions of the original texture. These versions are instanced belove and to the right of the originals in a series of smaller and smaller images. Each image has half the linear dimension (and a quarter the number of samples) of its parent.

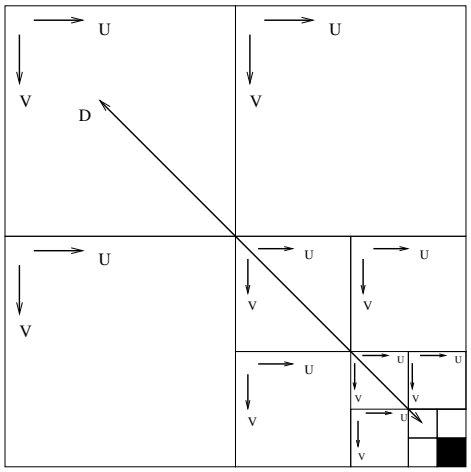


Figure D.1: Example of a MIP-map with 4 levels

A MIP-map has a pyramidal structure, as shown in figure D.1[a], and can be indexed by three coordinates (U,V and D in the figure). U and V are spatial coordinates of the map and D indexes the levels of the pyramid. We will use the texture contained in top left level to map zone with more precision and the bottom right level to map zones with less precision or more distant.

The MIP-map can be used with a trilinear interpolation in order to avoid the aliasing. The trilinear interpolation is composed by a bilinear interpolation between the pixels of a level and a linear interpolation between different levels.

D.2 The Pyramidal LOG sampling

The MIP mapping can be applied to approximate the chaine composed by the space-variant low-pass followed by the LOG sampling.

In this case we will construct a pyramid and then samples the input pixels from a level of the pyramid which is a function of the output pixel excentricity: if the output pixel to be computed is in the center of the image, the needed input pixels will be sampled from the top right levels, if the output pixel is on the borders of the image the needed input pixels will be sampled from the bottom right levels.

One of the major problem of this approach is the memory requirement to store the MIP-map. Our aim is to store the MIP-map into the internal memory and use the tiling to pre-fetches only some parts of the it when they are needed. However the MIP-map size remain a problem for the external memory size.

Two observations can bring to a reduction of the memory requirement.

A first observation is that the LOG sampling has a radial symmetry, thus we need to use only the levels on the dyagonal of the MIP-map.

A second observation is that the kind of low-pass applied to obtain the different levels of the MIP-map influences the correcteness of the results.

As an example let consider the pyramids shown in figure D.2; these pyramids are obtained by applying two kinds of filters:

- A mean fliter on a window of 5×5 pixels
- A mean filter whose window size is a function of the level of the pyramid.

We can observe that the mean filter with a variable window produces a pyramid having on the bottom right levels the same strength of blurring as on the borders of the reference image.

To conclude we can obtain the desired strength of blurring either by increasing the number of levels of the pyramid or by adapting the low-pass to the pyramid levels. The last solution reduces the memory requirement to store the pyramid but increases the number of operations performed to construct it.

[a] In the paper of Lance Williams [109] the pyramid is constructed in a symmetric way with respect the ours.

(a) Pyramid obtained with a mean filter 5×5

(b) Pyramid obtained with a mean filter having applied on a window of variable size

Figure D.2: Examples of pyramids constructed with different kinds of low-pass.

D.2.1 The function to access and construct the pyramid levels

The formulae of the LOG sampling are given in equation 9.2 of chapter 9; they compute the needed input pixels coordinates by multiplicating the output pixel coordinates by a fuction $f(x_{out}, y_{out})$, which depends on the output pixel excentricity. This function is:

$$f(x_{out}, y_{out}) = \frac{\rho_0}{\rho_{lim} - \rho_{out}} \tag{D.1}$$

Where $\rho_0 = \frac{W}{2}$ and $\rho_{lim} = \frac{W}{k}$, with $W = max(W_{out}, H_{out})$ and k a parameter of the LOG sampling which gives the reduction factor of the output image size and can have the following values $k = 2,\ 4,\ 8 \ldots$.

As the function $f(x_{out}, y_{out})$ is used (in the LOG sampling) to compute the needed input pixels coordinates, the access law to the pyramid levels has to depend on it. The

access law searched has to be bounded between 0 and $L-1$, where L is the number of levels in the pyramid.

Remark The function $f(x_{out}, y_{out})$ is maximal when the output pixel excentricity is maximal. The output pixel excentricity is $\rho_{out} = \sqrt{(\frac{W_{out}}{2})^2 + (\frac{H_{out}}{2})^2} < \frac{W}{2}\sqrt{2} - \epsilon$, with $W = max(W_{out}, H_{out})$ and ϵ a negligeable term, which garantees that $\rho_{out} < \rho_{lim}$. We can say that the values of the function $f(x_{out}, y_{out})$ are contained in the following interval

$$\frac{k}{2} \leq f(x_{out}, y_{out}) < \frac{k}{2 - k\sqrt{2} + \overline{\epsilon}}$$

with $\overline{\epsilon} = \frac{K\epsilon}{W}$ a negligeable term.

Among all the possible access law, we have chosen the $log_k()$, which is bounded between 0 and 1.41 as shown in table D.1. In this configuration the constructed pyramid

k	$log_k\left(\frac{k}{2}\right)$	$log_k\left(\frac{k}{2-k\sqrt{2}+\overline{\epsilon}}\right)$
2	0	1.41
4	0.5	1.2
8	0.6	1.13
16	0.75	1.10

Table D.1: Bounds of the function $log_k(f(x_{out}, y_{out}))$ with respect to k

has three levels indexed by $l \in \{0, 1, 2\}$. The function $log_k()$ is used twice:

- It is used to compute the variable size of the mean filter window used to construct the pyramid levels. The window size is $n \times n$ with $n = m(log_k(l)+1)$. The strength of the blurred can be renforced by fixing the value of m. In our experiments we have observed that $m = 11$ gives good results.
- It is used to access the pyramid levels in order to sample the needed input pixels.

D.2.2 The pyramidal LOG sampling steps

The Pyramidal log sampling goes through the following steps:

- From the output pixels coordinates, we compute $f(x_{out}, y_{out})$ and the needed input pixel coordinates (x_{in}, y_{in}). Due to the non-affinity othe memory accesses, the found input coordinates are not integer, thus we sample the 4 input pixel surrounding the non-integer coordinates in order to perform a bilinear interpolation.
- Then we compute the pyramid level $l = log_k(f(x_{out}, y_{out}))$, as this value is not an integer we take the two levels $l_1 = \lfloor l \rfloor$ and $l_2 = l_1 + 1$, in order to perform a linear interpolation between the results of the two previously computed bilinear interpolations.

List of Figures

1.1 For uniform memory accesses the window of input pixels associated to the calculation of an output pixel $I_t(p)$ has invariant shape and size with respect to p . . . 31
1.2 A pipelined image processor, (figure from [10]) . . . 33
1.3 Example of a processing element in a pipeline, realizing a pipeline itself. . 34
1.4 Example of a systolic array computing a matrix multiplication . . . 53
1.5 Y-chart, figure adapted from [17, 19] . . . 54
1.6 System Level design flow . . . 55
1.7 HLS tool architecture template, figure from [26] . . . 55
1.8 Processing Element architecture template, figure from [26] . . . 56
1.9 Code transformation to parallelize inter-dependent loop-nests communicating through memory. The transformation reduces the size of the internal memory used. “P" stands for “loop Producing data" and “C" stands for “loop Consuming data". . . . 56
1.10 Loop-based pseudo code; in the comments “s" stands for stream and “a" for memory access. . . . 57
1.11 inter-operation parallelism . . . 57
1.12 inter-operation and intra-loop parallelism . . . 57
1.13 inter-operation, intra-loop and inter loop parallelism . . . 58
1.14 The DRAM latency, figure from [38] . . . 58
1.15 perfectly nested uniform loop . . . 58
1.16 Unimodular Loop Transformations . . . 59
1.17 Example of parallelism between independent tiles . . . 59
1.18 example of a code with affine and non-affine array references . . . 59

2.1 The whole flow . . . 65
2.2 The TPU generic template . . . 68
2.3 MEXP flow . . . 69

3.1 Effect of tiling, of pre-fetching and computing parallelism and of tile sizes on the temporal performance of the system. . . . 74
3.2 Generic TPU template . . . 75

3.3 Table MM giving the memory mapping during the computation of all the output tiles . . . 76
3.4 Example of a prefetching mechanism based on the MM table . . . 77
3.5 Hash table IDX which gives the correspondence between the input tiles and the internal buffers . . . 78
3.6 A typical TPU time-line . . . 79
3.7 C-code of a TPU, this code can be customize for a given application and a particular solution chosen by MEXP. . . . 85
3.8 Example of some MEXP defined macros for an output image . . . 85
3.9 Example of a several possible implementations to compute 8 output tiles: a TPU with a single pipeline, a TPU with 2 parallel pipelines and a TPU with 4 parallel pipelines . . . 86
3.10 Example of a time-line for a TPU containing two parallel pipelines . . . 87
3.11 Set of files, contained in a MEXP generated project describing a TPU . . 87
3.12 C-code of a TPU, this code can be customize for a given application and a particular solution chosen by MEXP. . . . 88

4.1 Pseudo code of a Logarithmic Sampling . . . 90
4.2 Tiling applied to a non-affine indexed application . . . 90
4.3 Super-Tiling . . . 92
4.4 The Super-tiling Flow . . . 93
4.5 A generic iterative data-dominated algorithm . . . 93
4.6 Code of the profiling algorithm . . . 94
4.7 template of a user-defined REQ function . . . 95
4.8 Code of the tiling algorithm . . . 97
4.9 Layout of tile labels for a 2-dimensional, a 3-dimensional and a 4-dimensional space . . . 98
4.10 Super-Tiling example for a log sampling . . . 100
4.11 Projection algorithm . . . 101
4.12 Code of the ROT algorithm . . . 102
4.13 Histograms corresponding to the nbr. of IT per OT given in table 4.1. These values are multiplied by the tile volume. . . . 103

5.1 Example of an LOG sampling output with black borders . . . 106
5.2 Example of a graph corresponding to a 2-dimensional space with 4 output tiles . . . 106
5.3 Example of a graph for which the two problems of BTSP and TSP have different solutions. The blue round-trip is a solution for a TSP with a total cost of the round trip of 11 and a maximum edge cost encountered of 6; the red round-trip is a solution for the BTSP with a total cost of 12 and a maximum edge cost encountered of 4 . . . 107
5.5 Example of a symmetric (B)TSP instantiation for an output tiling applied to a 2-dimensional output space and producing 4 output tiles . . . 112
5.6 Example of a Displacement Mutation . . . 116

5.7 Comparison between a TSP and a BTSP solution for an instance with 128 vertices in a graph, i.e. 128 OT in the output image, ALGO1, a solution of EXP1 120

6.1 Example of a part of MM matrix for a TPU with a single pipeline ($N_p = 1$). Each column of the MM matrix corresponds to a TPU task and thus, to the computation of an output tile $OT(c)$. If an element of the matrix is 0, then the corresponding internal buffer is not used. 123
6.2 Example of a part of MM matrix for a TPU with a two pipelines ($N_p = 2$). Each column of the MM matrix corresponds to a TPU call and thus, to the computation of two parallel output tiles. The IB amount is not equal between the two parallel pipelines. 123
6.3 Example of Computation Mapping for $N_p = 4$ and different values of d. . 124
6.5 C-code to compute the MM matrix 127
6.6 Modifications to the C-code to compute the MM matrix are in bold. . . . 129
6.7 Example which compute an MM matrix from a Σ matrix. 132
6.8 Example of the Memory Mapping for the I/O tiling presented in figure 6.7 133

7.1 A Design Space is tailored by the user's constraints of IM_{MAX} and NC_{MAX}. The solutions s_1, s_2 and s_3 are pareto, they are equivalent to each other, while the solution s_4 is dominated by all the pareto solutions. 136
7.2 Example of a DSE solution tree with the dominance criteria and the insertion of a new solution. 137
7.3 Example of the parallelization between the iterations of a loop nest. The beginnings of the iterations are delayed one with respect to the other. . . 142
7.4 Example of the parallelization between the iterations of a loop nest with N_p parallel pipelines. 142

8.1 Example of a scatter representation of the solution space. 151
8.2 Example of a representation of TP with respect to the external memory variations. 152

9.1 Example of an input and output images for a log sampling 155
9.2 The input and output radial distances ρ_{in} and ρ_{out}. 156
9.3 Example of output image for different values of ρ_0 and k 156
9.4 REQ and CALC code. All the macros and global variables, shared between the user-defined and MEXP generated code are in bold. 158
9.5 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF. 160
9.6 Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF. 162
9.7 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA. 164

9.8 Measured temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA. 166
9.9 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV. 168
9.10 Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV. 169

10.1 Example of an input and output images for a space-variant low-pass followed by a LOG sampling 173
10.2 Example of a pyramid with 3 levels and constructed with a low-pass having a window size variable with the number of level 174
10.3 Pyramidal TPU code. All the macros and global variables, shared between the user-defined and MEXP generated code are in bold. 179
10.4 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF. 180
10.5 Measured temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a SQCIF. 181
10.6 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA. 183
10.7 Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA. 185
10.8 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV. 187
10.9 Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a HDTV. 189

11.1 Example of a polar transform 191
11.2 Example of an input and output images for a Polar transform 192
11.3 Polar transform TPU code. All the macros and global variables, shared between the user-defined and MEXP generated code are in bold. 195
11.4 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a 128×128. 196
11.5 Measured temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a 128×128. 197
11.6 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image contains 300×300 pixels. . . . 199

11.7 Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p); the input image is a VGA. 201
11.8 DSE for different external memory latency (L) and for different number of parallel pipelines (N_p). 203
11.9 Estimated temporal performance for different external memory latency (L) and for different number of parallel pipelines (N_p) 205

A.1 Example of a bilinear interpolation. 213
A.2 Magnification of a detail of an output image showing the anti-aliasing effect of the bilinear interpolation 214

B.1 Example of two different LUTs realizations with a different distribution of the samples into the function domain. The LUT entries are data in the intervals $[x_i, x_{i+1}[$ and the LUT output are exactly the data v_i, with $i \in [0, 4]$. 215
B.2 The entry function of the LUT. 217
B.3 Example of a $f(x, y)$ approximation for a SQCIF input image (i.e. 128×96 pixels). The figure gives the error distribution on the pixels. An error on a pixel position can be at most of one along both directions x and y. . 218
B.4 Output of an approximated LOG sampling applied to a VGA image and using different LUT size 221

C.1 Example of a spatial-variant low pass, the input image is filtered along the two directions (x,y) and in the two ways for each direction 223

D.1 Example of a MIP-map with 4 levels 225
D.2 Examples of pyramids constructed with different kinds of low-pass. 227

List of Tables

1.1 Table presenting a generic loop nest 32

4.1 Table of the Super-Tiling explorations for ROT,LOG and M22 algorithms 104

5.1 Explorations run for an input tile volume of 128 with an input tile layout among 128x1, 64x2, 32x4, 16x8, 8x16 117
5.2 Table giving the complexity of the analyzed instances in terms of number of vertices to be visited. “min., av. and max." are respectively the minimum the average and the maximum of vertices in the analyzed number of instances (# inst) 117
5.3 Table comparing the results of a GATSP and a Lin-Kernighan solver with a number of kicks of $d \times 100$, the experiments have been run for different target algorithms (ALGO) and different explorations (EXP) 118
5.4 Table comparing the results of a GATSP and a Lin-Kernighan solver with a number of kicks of 10^6, the experiments have been run for different target algorithms (ALGO) and different explorations (EXP) 119
5.5 Table comparing the results of a GATSP and The Concorde solver, the experiments have been run for different target algorithms (ALGO) and different explorations (EXP) 119

6.1 Example of MM divided into two tables 130

8.1 Example of a table summarizing the information on the TP. 150
8.2 Example of table giving the information on the area occupancy of a solution 151

9.1 Experiments run for different input image sizes 158
9.2 Solutions s.3, s.13 and s.6 are pareto solutions, while s.35 is a non-pareto solution. 160
9.3 Estimated and measured Temporal Performance (TP) for a SQCIF. . . . 161
9.4 Measured cost of the TPU after the RTL generation 161
9.5 Estimated and measured Temporal Performance (TP) for a VGA input image. 165
9.6 Measured cost of the TPU after the RTL generation. 165

9.8 Estimated and measured Temporal Performance (TP) for a HDTV input image. 170
9.9 Measured cost of the TPU after the RTL generation. 170

10.1 Experiments run for different input image sizes 176
10.2 Examples of solutions in the analyzed space 176
10.3 Estimated and measured Temporal Performance (TP) for a SQCIF input image. 177
10.4 Measured cost of the TPU after the RTL generation. 178
10.5 Examples of explored solutions 182
10.6 Estimated and measured Temporal Performance (TP) for a VGA input image. 184
10.7 Measured cost of the TPU after the RTL generation. 184
10.8 Examples of solutions from the analyzed space 186
10.9 Estimated and measured Temporal Performance (TP) for a HDTV input image 188
10.10 Measured cost of the TPU after the RTL generation. 188

11.1 Experiments run for different input image sizes 192
11.2 Examples of solutions n the analyzed space 193
11.3 Estimated and measured Temporal Performance (TP) for an input image containing $128x128$ pixels 194
11.4 Measured cost of the TPU after the RTL generation. 194
11.5 Examples of explored solutions 198
11.6 Estimated and measured Temporal Performance (TP) for an input image containing $300x300$ pixels. 200
11.7 Measured cost of the TPU after the RTL generation. 200
11.8 Examples of solutions from the analyzed space 202
11.9 Estimated and measured Temporal Performance (TP) for an input image containing $600x600$ pixels. 204
11.10 Measured cost of the TPU after the RTL generation. 204

B.1 Values of the different indicators for different image sizes: SQCIF (128×96), VGA (640×320) and HDTV (1920×1080) 219
B.2 PSNR and MSSIM for several images of different sizes. $W = W_{out}$. 220

D.1 Bounds of the function $log_k(f(x_{out}, y_{out}))$ with respect to k 228

Bibliography

[1] F Catthoor et al. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, 2002.

[2] WA Wulf and SA McKee. Hitting the memory wall. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[3] N Mitchell, L Carter, and J Ferrante. Localizing non-affine array references. *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 192–202, 1999.

[4] Jeanny Hérault and Barthélémy Durette. Modeling visual perception for image processing. *IWANN*, 4507:662–675, 2007.

[5] D Gajski and L Ramachandran. Introduction to high-level synthesis. *IEEE Design & test of computers*, 11:44–54, 1994.

[6] T Kambe et al. A C-based synthesis system, Bach, and its application (invited talk). *Proceedings of ASP-DAC'01*, pages 151–155, 2001.

[7] PR Panda, F Catthoor, ND Dutt, K Danckaert, E Brockmeyer, C Kulkarni, A Vandercappelle, and PG Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.

[8] M Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, pages 1901–1909, 1966.

[9] R Duncan. A survey of parallel computer architectures. *Computer*, 23:5–16, 1990.

[10] N Storey and RC Staunton. An adaptive pipeline processor for real-time image processing. *Proc. SPIE, Paper*, 1197:238–246, 1989.

[11] R Lougheed and D McCubbrey. The cytocomputer: A practical pipelined image processor. *Proceedings of the 7th annual symposium on Computer Architectures*, pages 271–277, 1980.

[12] EW Kent, MO Shneier, and R Lumia. Pipe. *Journal of Parallel and Distributed Computing*, 2:50–78, 1985.

[13] H Kungt and C Leisersont. *Systolic arrays (for VLSI)*, pages 256–282. Society for industrial and applid mathemathics, 1979.

[14] Abbas Bigdeli et al. A new pipelined systolic array-based architecture for matrix inversion in FPGAs with Kalman filter case study. *EURASIP J. Appl. Signal Process.*, 2006:1–13, 2006.

[15] D Moldovan and J Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35:1–12, 1986.

[16] A Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION, the VLSI journal*, 12:293–304, 1991.

[17] DD Gajski. System-level design methodology, lecture notes, university of california. http://camars.kaist.ac.kr/ aeng/cs710/esd07/System Level Design Methodology.pdf, 2003.

[18] A Sangiovanni-Vincentelli. Quo vadis, sld? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[19] C Jego. Conception de systèmes numériques le codesign, lecture enst bretagne. http://public.enst-bretagne.fr/ marzel/codesign/cours_CoDesign.ppt, 2006.

[20] A Pimentel et al. Exploring embedded-systems architectures with artemis. *Computer*, 34:57–63, 2001.

[21] A Sangiovanni-Vincentelli. Compositional modeling in metropolis. *Second International Workshop on Embedded Software (EMSOFT)*, 2491:93–107, 2002.

[22] Mentor Graphics. Catapult C synthesis-based design flow: Speeding implementation and increasing flexibility. http://www.techonline.com/electronics-directory/techpaper/193102520.

[23] S Edwards. The challenges of hardware synthesis from c-like languages. *DATE'05*, pages 66–67, 2005.

[24] D Rao and M Venkatesan. An efficient reconfigurable architecture and implementation of edge detection algorithm using handle-c. *ITCC'04*, 2:846, 2004.

[25] J Zhu et al. Syntax and semantics of SpecC language. *SASIMI'97*, pages 75–82, 1997.

[26] Vinod Kathail, Shail Aditya, and B Ramakrishna. PICO: Automatically designing custom computers. *Computer*, 35:39–47, 2002.

[27] R Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *The Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[28] Stefaan Note et al. Cathedral-III architecture-driven high-level synthesis for high throughput DSP applications. *Proceedings of DAC'91*, pages 597–602, 1991.

[29] Sumit Gupta, Manev Luthra, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Hardware and interface synthesis of FPGA blocks using. *International Conference on Parallel and Distributed Computing Systems*, 2003.

[30] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Loop shifting and compaction for the high-level synthesis. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, 1:10114, 2004.

[31] Nitin Chawla et al. Multimedia application specific engine design using high level synthesis. *Designcon'08*, 2008.

[32] Shail Aditya, B Ramakrishna, and Rau Vinod Kathail. Automatic architectural synthesis of vliw and epic processors. *Proceedings of the 12th international symposium on System Synthesis*, page 107, 1999.

[33] R Schreiber, B Rau, A Darte, and F Vivien. A constructive solution to the juggling problem in processor array synthesis. *14th International Parallel and Distributed Processing Symposium*, pages 815–821, 2000.

[34] C Zinner and W Kubinger. ROS-DMA: a DMA double buffering method for embedded image processing with resource optimized slicing. *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 361–372, 2006.

[35] Imec. Data transfer and storage exploration. http://www.imec.be/wwwinter/mediacenter/en/SR2005/html/142230.html.

[36] SA McKee. Reflections on the memory wall. *Proceedings of the 1st conference on Computing frontiers*, page 162, 2004.

[37] M Gries. A survey of synchronous ram architectures. Technical Report 71, Computer Engineering and Networks Laboratory, 1999.

[38] B Jacob. Lecture 2 university of meriland, DRAM circuit and architecture basics. www.ece.umd.edu/ blj/dram/, 2003.

[39] Wikipedia. Cache. http://en.wikipedia.org/wiki/Cache.

[40] R Banakar, S Steinke, BS Lee, M Balakrishnan, and P Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, 2002.

[41] S Udayakumaran and R Barua. An integrated scratch-pad allocator for affine and non-affine code. *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 925–930, 2006.

[42] AJ Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[43] IM Verbauwhede, CJ Scheers, and JM Rabaey. Memory estimation for high level synthesis. *Proceedings of the 31st annual conference on Design automation*, pages 143–148, 1994.

[44] S Wuytack, JP Diguet, FVM Catthoor, and HJ De Man. Formalized methodology for data reuse: exploration for low-powerhierarchical memory mappings. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4):529–537, 1998.

[45] T Van Achteren, G Deconinck, F Catthoor, and R Lauwereins. Data reuse exploration techniques for loop-dominated applications. *Proceedings of the conference on Design, automation and test in Europe*, page 428, 2002.

[46] F Balasa et al. Computation of storage requirements for multi-dimensional signal processing applications. *IEEE Transactions on VLSI Systems*, 15:447–460, 2007.

[47] J Seo, T Kim, and PR Panda. Memory allocation and mapping in high-level synthesis–an integrated approach. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11:928–938, 2003.

[48] A Darte, R SChreiber, and G Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[49] Darko Kirovski, Chunho Lee, Miodrag Potkonjak, and William H Mangione-smith. Application-driven synthesis of memory-intensive systems-on-chip. *IEEE Transactions on Computer-Aided Design*, CAD-18:1316–1326, 2001.

[50] E De Greef, F Catthoor, and H De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 13:1811–1837, 1997.

[51] W Thies, F Vivien, J Sheldon, and S Amarasinghe. A unified framework for schedule and storage optimization. *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 232–242, 2001.

[52] HL Muller, PWA Stallard, and DHD Warren. The application of skewed-associative memories to cache only memory architectures. *Proceedings of the international conference on Parallel Processing*, pages 1150–1154, 1995.

[53] GE Suh, S Devadas, and L Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 117–128, 2002.

[54] G Memik, M Kandemir, M Haldar, and A Choudhary. A selective hardware/compiler approach for improving cache locality. Technical Report CPDC-TR-9909-016, Northwestern University, 1999.

[55] M Weinhardt and W Luk. *Memory access optimization and RAM inference for pipeline vectorization*, pages 61–70. New York : Springer Verlag, 1999.

[56] F Catthoor et al. *Custom memory management methodology: Exploration of memory organisation for embedded multimedia.* Kluwer Academic Publishers, 1998.

[57] E Brockmeyer, A Vandecappelle, and F Catthoor. Systematic cycle budget versus system power trade-off: a new perspective on system exploration of. *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 137–142, 2000.

[58] PR Panda, H Nakamura, ND Dutt, A Nicolau, S Inc, and M View. Augmenting loop tiling with data alignment for improved cacheperformance. *IEEE transactions on computers*, 48(2):142–149, 1999.

[59] G Goumas, A Sotiropoulos, and N Koziris. Minimizing completion time for loop tiling with computation and communication overlapping. *Proceedings of 15th International Parallel and Distributed Processing Symposium*, page 39, 2001.

[60] F Rastello and Y Robert. Loop partitioning versus tiling for cache-based multiprocessors. *Proceedings of International Conference on parallel and Distributed computing Systems*, pages 477–483, 1998.

[61] Sungdo Moon and Rafael H Saavedra. Hyperblocking: A data reorganization method to eliminate cache conflicts in tiled loop nests. Technical report, Computer Science Department; University of Southern Carolina, 1998.

[62] M Stan and W Burleson. Bus-invert coding for low-power i/o. *IEEE Transactions on Very Large Scale Integration (VLSI)*, 3:49–57, 1995.

[63] L Carter and J Ferrante. CROPS: coordinated restructuring of programs and storage. *ACM SIGSOFT Software Engineering Notes*, 25(1):38–39, 2000.

[64] C Huang, S Ravi, A Raghunathan, and N Jha. Generation of heterogeneous distributed architectures for memory-intensive applications through high level synthesis. *IEEE Transactions on Very Large Scale Integration*, 15:1191–1204, 2007.

[65] T Jacobson and G Stubbendieck. Dependency analysis of for-loop structures for automatic parallelization of C code. *Mathematics and Computer Science Department South Dakota School of Mines and Technology*, pages 1–13, 2002.

[66] Yi qing Yang, Corinne Ancourt, and Centre De Recherche En Informatique. Minimal data dependence abstractions for loop transformations (extended version). *International Journal on Parallel Processing*, 23:359–388, 1995.

[67] W Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, 1991.

[68] C Bastoul, A Cohen, S Girbal, S Sharma, and O Temam. Putting polyhedral loop transformations to work. *Lecture Notes in Computer Science*, 2958:209–225, 2004.

[69] H Le Verge, V Van Dongen, and DK Wilde. Loop nest synthesis using the polyhedral library. Technical report, INRIA, Unité de recherche de Rennes, Rennes, FRANCE, 1998.

[70] J Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, 1997.

[71] M O'Boyle and G Hedayat. Load balancing of parallel affine loops by unimodular transformations. Technical report, Departement of Computer Science, University of Manchester, 1992.

[72] RH Saavedra, W Mao, D Park, J Chame, and S Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 39–45, 1996.

[73] J Torres, E Ayguade, J Labarta, and M Valero. Loop parallelization: revisiting framework of unimodular transformations. *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, 1996. PDP'96.*, pages 420–427, 1996.

[74] J Ramanujam. Non-unimodular transformations of nested loops. *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 214–223, 1992.

[75] SK Singhai and KS McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[76] JMP Cardoso. Self-loop pipelining and reconfigurable dataflow arrays. *Computer Systems: Architectures, Modeling, and Simulation*, 3133:234–243, 2004.

[77] J Cortadella, RM Badia, and F Sanchez. A mathematical formulation of the loop pipelining problem. Technical report, Universitat Politecnica de Catalunya, 1996.

[78] J Jeon and K Choi. Loop pipelining in hardware-software partitioning. *Proceedings of the ASP-DAC'98*, pages 361–366, 1998.

[79] PMW Knijnenburg, T Kisuki, and MFP O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.

[80] F Irigoin and R Triolet. Supernode partitioning. *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329, 1988.

[81] K Hogstedt, L Carter, and J Ferrante. On the parallel execution time of tiled loops. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):307–321, 2003.

[82] Gabriel Rivera and Chau wen Tseng. Tiling optimizations for 3D scientific computations. *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 32, 2000.

[83] C Hsu and U Kremer. A quantitative analysis of tile size selection algorithms. *The Journal of Supercomputing*, 27(3):279–294, 2004.

[84] PMW Knijnenburg, T Kisuki, and MFP O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.

[85] J Ramanujam and P Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[86] Chung-Hsing Hsu. A stable and efficient loop tiling algorithm. *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2000.

[87] P Boulet and J Dongarra. Tiling for heterogeneous computing platforms. Technical Report UT-CS-97-373, University of Tennessee, 1997.

[88] A Darte, GA Silber, and F Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[89] S Parsa and S Lotfi. A new genetic algorithm for loop tiling. *The Journal of Supercomputing*, 37(3):249–269, 2006.

[90] Jaume Abella, Antonio Gonzalez, Josep Llosa, Xavier Vera, and Malardalens Hogskola. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. *ICPP'02*, page 568, 2002.

[91] Zhiyuan Li and Yonghong Song. Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Languages and Systems*, 24(17):1639–41, Sep 2000.

[92] P Boulet, A Darte, T Risset, and Y Robert. (pen)-ultimate tiling? *Integration-The VLSI Journal*, 17(1):33–52, 1993.

[93] F Rastello and Y Robert. Automatic partitioning of parallel loops with parallelepiped-shapedtiles. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):460–470, 2002.

[94] DE Maydan, JL Hennessy, and MS Lam. Effectiveness of data dependence analysis. *International Journal of Parallel Programming*, 23:63–81, 1995.

[95] A Halambi, P Grun, V Ganesh, A Khare, N Dutt, and A Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. *Proceedings of the conference on Design, automation and test in Europe*, page 485, 1999.

[96] P Lippens, J Van Meerbergen, and A Van der Werf. Phideo: a silicon compiler for high speed algorithms. *Euro DAC'91*, pages 436–441, 1991.

[97] D Cachera et al. Hardware design methodology with the alpha language. *FDL'01*, 2001.

[98] A Darte et al. High level code transformations, compsys. http://jeudi.inrialpes.fr/2003/Raweb/compsysuid28.html, 2003.

[99] S Abraham, B Rau, and R SChreiber. Fast design space exploration through validity and quality filtering of subsystem designs. Technical report, Hewlett-Packard, 2000.

[100] J LaRusic. A heuristic for solving the bottleneck traveling salesman problem. Master's thesis, University of New Brunswick, Canada, 2005.

[101] P Larranaga, CMH Kuijpers, RH Murga, I Inza, and S Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[102] Bernd Freisleben and Peter Merz. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. *Proceedings of IEEE International conference on Evolutionary Computation*, pages 616–621, 1996.

[103] William Cook. Concorde. http://www.tsp.gatech.edu/concorde.html.

[104] M. Schira and al. Two-dimensional mapping of the central and parafoveal visual field to humain visual cortex. *J. Neurophysiology*, 97:4284–4295, 2007.

[105] W Yu, SIT Center, and SK Daejeon. An embedded camera lens distortion correction method for mobile computing applications. *IEEE Transactions on Consumer Electronics*, 49(4):894–901, 2003.

[106] AGJ Nijmeijer, MA Boer, CH Slump, MM Samson, MJ Bentum, GJ Laanstra, H Snijders, J Smit, and OE Herrmann. Correction of lens-distortion for real-time image processingsystems. *VLSI Signal Processing, VI, 1993.,[Workshop on]*, pages 316–324, 1993.

[107] L Qiang and N Allinson. Spatial optical distortion correction in an fpga. *IEEE Workshop on Signal Processing Systems Design and Implementation*, pages 268–273, 2006.

[108] AC Brooks and TN Pappas. Using structural similarity quality metrics to evaluate image compression techniques. *IEEE International Conference on Acoustics, Speech and Signal Processing, 2007. ICASSP 2007*, 1, 2007.

[109] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983.

Résumé de thèse :
Dans le cadre de la synthèse de haut niveau (SHN), qui permet d'extraire un modèle structural à partir d'un modèle algorithmique, nous proposons des solutions pour optimiser l'accès et le transfert de données du matériel cible.
Une méthodologie d'exploration de l'espace des architectures mémoire possibles a été mise au point.
Cette méthodologie trouve un compromis entre la quantité de mémoire interne utilisée et les performances temporelles du matériel généré.
Deux niveau d'optimisation existe :

1. Une optimisation architecturale, qui consiste à créer une hiérarchie mémoire,
2. Une optimisation algorithmique, qui consiste à partitionner la totalité des données manipulées pour stocker en interne seulement celles qui sont utiles dans l'immédiat. Pour chaque répartition possible, nous résolvons le problème de l'ordonnancement des calculs et de mapping des données. À la fin, nous choisissons la ou les solutions pareto.

Nous proposons un outil, front-end de la SHN, qui est capable d'appliquer l'optimisation algorithmique du point 2 à un algorithme de traitement d'image spécifié par l'utilisateur. L'outil produit en sortie un modèle algorithmique optimisé pour la SHN, en customisant une architecture générique.

Abstract:
In this dissertation we present a method able to run a Design Space Exploration oriented to the optimization of the data transfer and storage management.
The corresponding developed tool has been used as a front-end of HLS in order to help the user to find an optimized memory micro- architecture.
Our method is able to handle image processing applications with non- affine array references. It is able to apply a paving which, on one hand, is based on a run-time dependence analysis and, on the other hand, uses disjoint and equal-by- translation blocks to partition the data and instruction sets. The non- affinity of the array references is taken into account by projecting the instruction paving on the data paving.
This method leads to a memory micro-architecture that is, at the same time, adapted to the non- affinity of the array references of the application and has a cheap control on the data transfer because of the invariability of the size of transferred data blocks.