UNIVERSITÉ DE MONTRÉAL

OUTILS ET MÉTHODES POUR LE TRAITEMENT PARALLÈLE DE CALCULS SUR DES TABLEAUX

NORMAND BÉLANGER

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSPHIÆ DOCTOR (Ph.D.)
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 1997

©Normand Bélanger 1997.

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-32988-7

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

OUTILS ET MÉTHODES POUR LE TRAITEMENT PARALLÈLE DE CALCULS SUR DES TABLEAUX

présenté par: BÉLANGER Normand

en vue de l'obtention du diplôme de: Philosophiae Doctor (Ph.D.)

a été dûment acceptée par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président

M. SAVARIA Yvon, Ph.D., directeur de recherche

M. DAGENAIS Michel, PhD., membre

M. KROPF Peter, Ph.D., membre externe

M. ABOULHAMID ElMostapha, Ph.D., membre

Remerciements

Je voudrais remercier mon directeur, Yvon Savaria, pour m'avoir fourni un environnement où mes idées ont, non seulement germées, mais aussi où elles ont pu arriver à maturité. Je voudrais également le remercier pour m'avoir appuyé au cours de toutes ces années.

Je voudrais aussi signifier ma gratitude à ma conjointe, Sylvie Fortin, pour m'avoir soutenu au cours de mes études doctorales et pour avoir fait preuve de patience dans l'attente du moment où la présente thèse serait terminée. Je voudrais aussi souligner l'aide qu'elle m'a apportée dans l'amélioration de cette thèse par une patiente et méticuleuse revue du texte.

Résumé

Le traitement parallèle est très important pour certaines applications car elles pourraient mettre à profit une augmentation de plusieurs ordres de grandeur de la performance des ordinateurs les plus puissants disponibles. Or, l'amélioration des technologies et de l'architecture des ordinateurs mono-processeurs ne permet pas ce niveau d'augmentation de performance. D'un autre côté, la parallélisation automatique d'applications pose de nombreux problèmes. Dans la présente thèse, trois de ces problèmes sont abordés soient:

- le calcul rapide d'adresses,
- la programmation à haut niveau d'ordinateurs SIMD et
- le partitionnement automatique de tableaux.

Le traitement structuré de tableaux permet une plus grande performance que le traitement non-structuré puisqu'il permet:

- le transfert des données avant qu'elles ne soient requises,
- l'utilisation d'instructions vectorielles et
- une meilleure utilisation d'une hiérarchie de mémoire.

Pour que le transfert de données entre la mémoire et le processeur ne ralentisse pas leur traitement par le processeur, le calcul des adresses doit être efficace et il doit être effectué par un organe de calcul autre que le processeur. Par contre, les transformations sur les tableaux qui modifient leur adressage sont des transformations qui sont très souvent linéaires. En conséquence, on propose un algorithme qui supporte ce type de calcul d'adresses. On montre que cet algorithme est efficace et qu'il peut être implanté en matériel avec une faible complexité.

Les architectures SIMD sont très appropriées pour le traitement structuré de tableaux puisque cette architecture matérielle reflète la structure des calculs. Cependant, à ce jour, aucun langage de programmation n'a été proposé qui permette de décrire un traitement structuré sur des

tableaux en utilisant des opérateurs sur des tableaux (i.e. en utilisant le niveau d'abstraction le plus judicieux) tout en visant la compilation vers les architectures SIMD. On propose un tel langage et on démontre comment on peut rendre efficace le code généré par un compilateur. En particulier, on montre comment l'utilisation de tampons circulaires et d'instructions vectorielles peut améliorer la performance lorsqu'on effectue des convolutions.

Le traitement structuré de tableaux implique essentiellement le traitement de sections de tableaux. Or, le fait que trouver la meilleure distribution des données entre les processeurs est NP-complet peut être contourné en limitant l'analyse des calculs à effectuer à l'analyse des opérations sur les sections de tableaux et en utilisant le modèle de parallélisme du langage HPF. De plus, comme ce modèle impose une structure régulière à la distribution des tableaux, son utilisation ne diminue pas la performance des applications qui effectuent un traitement régulier. Dans cette thèse, on propose une méthode et des algorithmes de parallélisation qui vont dans le sens décrit. On montre que ces algorithmes on une faible complexité temporelle et qu'ils permettent de générer des directives de parallélisation HPF qui peuvent améliorer la performance. Cependant, cette amélioration est mitigée par le manque de maturité du compilateur HPF utilisé.

Finalement, on montre comment formaliser et généraliser le modèle de partitionnement HPF à l'aide de MOA et du λ-calcul.

Abstract

Parallel processing is very important to many applications, because they can take advantage of an improvement of more than one order of magnitude over the performance of the most powerful computers currently available, and because technological and architectural improvements cannot provide such a sharp increase in performance. On the other hand, automatic distribution of applications is difficult in many respects. In this thesis, three of these difficulties are tackled, namely:

- fast address computations,
- high-level programming of SIMD computers, and
- automatic distribution of arrays.

Structured array processing can achieve higher performance than unstructured processing because it allows:

- prefetching data,
- the use of vector instructions, and
- a better use of a memory hierarchy.

In order to prevent data transfers between the memory and the processor from slowing down the processing of that data, address computations must be efficient and they must not be performed by the processor. On the other hand, transformations on arrays that modify the way arrays are accessed are often linear. Thus, an algorithm is proposed that support these types of transformations. It is shown that this algorithm is efficient and that it can be implemented in hardware at a very small cost.

SIMD architectures are very appropriate for structured array processing because this type of architecture is similar to the structure of the computations. On the other hand, to this day, no programming language has been proposed that allows the description of structured computations through support of array operators while aiming at compiling for SIMD architectures. Such a language is proposed in this thesis, and it is shown

how a compiler for that language can generate efficient code. In particular, it is shown how to improve performance through the use of circular buffers and vector instructions.

Structured array processing is essentially the processing of array sections. Also, the NP-completeness of the automatic distribution problem can be circumvented by limiting the analysis to array section interactions and by using the HPF model of parallelism. Furthermore, since this model imposes a regular structure to the distribution of arrays, its use does not lower performance of (parallelized) applications if they perform structured processing. In this thesis, a partitioning method and algorithms are proposed. It is shown that the algorithms have a low time complexity and that they allow generating HPF directives that can improve performance but that this improvement is mitigated by the lack of maturity of the HPF compiler used.

Finally, it is shown how to formalize and generalize the HPF model of parallelism by using MOA and the λ-calculus.

Table des Matières

Remerciements iv

Résumé v

Abstract vii

Liste des Tableaux xii

Liste des Figures xiii

Liste des Annexes xv

1 Introduction 1

1.1 Modèle de parallélisme de HPF 3

1.1.1 Directives de parallélisation HPF 3

1.2 Architecture du SIMD de Pulse 5

2 Revue de la littérature 7

2.1 Partitionnement de boucles imbriquées 7

2.2 Partitionnement de code de haut niveau 8

2.3 Géneration d'adresse pour les tableaux 9

2.4 Langages de haut niveau pour la programmation SIMD 10

3 Analyse des besoins 11

3.1 Méthodes itératives 12

3.2 Méthodes directes 13

3.3 Contexte matériel 13

3.4 Discussion 14

3.5 Mise en oeuvre d'un objet-tableau C++ 15

3.5.1 Opérateurs implantés 15

3.5.2 Applications implantées 16

3.6 Conclusions 16

4 Génération d'adresses **18**
4.1 Introduction 18
4.2 Patrons d'adresses 20
4.3 Algorithme 21
4.4 Implantation logicielle 25
4.4.1 Code et modèle temporel 25
4.4.2 Évaluation du temps d'exécution 27
4.5 Implantation matérielle 32
4.6 Transformations 35
4.6.1 Transformations du Fortran 90 36
Section 36
Transposition 37
Spread 37
Reshape 37
4.6.2 Autres transformations 38
Partition 38
"Warping" 39
Renversement 39
Damier 40
4.7 Paramètres 41
4.7.1 Distance 42
4.7.2 Forme 43
4.7.3 Adresse de départ 43
4.7.4 Pas 44
4.7.5 Généralisation 44
4.8 Conclusions 45

5 Un langage de haut niveau pour les ordinateurs SIMD **47**
5.1 Description du langage 48
5.1.1 Sous-ensemble du C supporté 48
5.1.2 Extensions ajoutées au C 49
5.2 Sémantique 50
5.2.1 Structures de support pour les tableaux 51
Opérateurs 51

L'énoncé "where" 52
Fonction intrinsèques d'entrée/sortie 52
5.2.2 Distribution 52
5.3 Tampons circulaires 53
5.3.1 Bande passante requise par les registres vectoriels 53
5.3.2 Stratégie d'allocation dans les tampons circulaires 55
5.3.3 Évaluation de la stratégie d'allocation 58
Gain de vitesse 58
Quantité de mémoire utilisée 59
5.3.4 Étude du cas 3D 60
5.4 Exemple de programme 61
5.5 Analyse des performances obtenues 64
5.6 Conclusions 66

6 Génération automatique de directives HPF 67
6.1 Cadre conceptuel et algorithmes 67
6.1.1 Fonction de coût 68
6.1.2 Extraction de l'information 69
6.1.3 Algorithmes 70
6.1.4 Complexité temporelle des algorithmes 73
6.2 Implantation 74
6.2.1 Bancs d'essais 74
6.3 Conclusions 76

7 Généralisation et formalisation du modèle de partitionnement ... 77
7.1 Classe de distribution 78
7.2 Algorithmes 79
7.2.1 Exemple 83
7.3 Conclusions 85

8 Conclusions 86

Bibliographie 89

Annexes 94

Liste des Tableaux

Tableau 4.1 Résumé des pertes de vitesse pour le processeur simple (S) et pour le processeur super-scalaire (SS) pour les deux premières expériences 31
Tableau 6.1 Liste des relations de l'exemple de l'équation 2 70
Tableau 6.2 Temps d'exécution pour l'application MacCormack 75
Tableau 6.3 Temps d'exécution pour l'application Semad 75

Liste des Figures

Figure 1.1 Exemples de directives HPF 4
Figure 1.2 Architecture du SIMD de **Pulse** 6
Figure 4.1 Algorithme de calcul d'adresse simplifié 22
Figure 4.2 Algorithme de calcul d'adresses 24
Figure 4.3 Boucles principales de l'algorithme en langage C 25
Figure 4.4 Temps moyen de traitement par élément pour un tableau de $n \times 8 \times m$ où $n \times m = 1024$ 29
Figure 4.5 Temps de traitement pour un nombre variable de dimensions et un nombre d'éléments constant 30
Figure 4.6 Temps de traitement moyen par élément pour une matrice carrée 30
Figure 4.7 Diagramme-bloc du générateur d'adresse 34
Figure 4.8 Section (2:9,2:9) d'un tableau de forme (10,10) 36
Figure 4.9 Tableau de forme (10,10) transposé 37
Figure 4.10 Un tableau de forme (10,10) dupliqué trois fois selon la dimension 2 38
Figure 4.11 Un tableau de (10,10) partitionné selon la dimension 2 en 2 parties 39
Figure 4.12 "Warping" de la section (3:8,3:8) d'un tableau de forme (10,10) selon la dimension 2 et en décalant de 2 selon la dimension 1 40
Figure 4.13 Reversement d'un tableau de forme (10,10) selon la dimension 2 40
Figure 4.14 Damier de 4 cases sur 8 fait à partir d'un tableau de forme (16,16) 42
Figure 4.15 Transformation arbitraire 45
Figure 5.1 Gain de vitesse entre les vectorisations partielle et totale .. 58
Figure 5.2 Quantité supplémentaire de mémoire requise 59
Figure 5.3 Exemple de code HPCP 62
Figure 5.4 Code C-Pulse généré 63
Figure 5.5 Premier programme de test HPCP 64
Figure 5.6 Deuxième programme de test HPCP 65

Figure D.1 Code C-PULSE généré pour le programme de la figure 5.5 . 107
Figure D.2 Code assembleur généré pour le programme de la figure 5.5 108
Figure D.3 Code assembleur généré pour le programme de la figure 5.5 (suite) 109
Figure D.4 Code C-PULSE généré pour le programme de la figure 5.6 . 110
Figure D.5 Code assembleur généré pour le programme de la figure 5.6 111
Figure D.6 Code assembleur généré pour le programme de la figure 5.6 (suite) 112
Figure D.7 Code assembleur généré pour le programme de la figure 5.6 (fin) 113
Figure D.8 Code assembleur généré pour le programme de la figure 5.3 114
Figure D.9 Code assembleur généré pour le programme de la figure 5.3 (suite) 115
Figure D.10 Code assembleur généré pour le programme de la figure 5.3 (fin) 116

Liste des Annexes

A Introduction à MOA 94

B Code pseudo-assembleur pour la génération d'adresse 98

C Grammaire de HPCP 100

D Code C-PULSE et assembleur des programmes de test 106

Chapitre 1

Introduction

Avant tout, il convient de signaler que presque tous les travaux présentés dans la présente thèse ont été effectués dans le cadre du projet **Pulse**. Ce projet consiste à concevoir un ordinateur SIMD ainsi que des logiciels (par exemple, un compilateur, un assembleur et des applications) permettant de programmer ledit ordinateur.

Certaines applications pourraient mettre à profit une puissance de calcul de quelques ordres de grandeur supérieure à celle des ordinateurs les plus puissants présentement disponibles (par exemple, la prévision météorologique, la modélisation de climat, la simulation de fluides ou de champs de particules tel qu'atomes et molécules ou corps planétaires). Étant donné que l'architecture des ordinateurs mono-processeurs est déjà fortement contrainte par la vitesse de propagation des signaux et qu'aucune nouvelle technologie (qui permettrait des fréquences d'horloge nettement plus élevées) n'est sur le point de prendre la relève, on se doit de se tourner vers les ordinateurs multi-processeurs si on veut maintenir le taux d'augmentation de performance que l'on connait depuis quelques décennies [22].

Cependant, pour exécuter plus rapidement les programmes, il faut diviser le travail de façon équilibrée entre les processeurs. Pour le cas général, trouver la répartition optimale est un problème NP-complet [37]. Par contre, sachant que la plupart des applications concernées traitent des tableaux, on peut limiter le champ des applications à supporter à ces dernières. Ce champ d'applications comprend la solution d'équation(s) différentielle(s) ordinaire(s) ou aux dérivées partielles, la solution de systèmes d'équations linéaires (ces deux types d'équations sont utilisés pour modéliser et simuler différents aspects de notre univers) et la minimisation de

fonctions à "plusieurs" variables (ce qui permet d'extraire de l'information d'un signal comme, par exemple, analyser le signal d'un radar météorologique pour détecter les tornades en formation et prévenir la population — ces problèmes sont dits de déconvolution).

Les langages de programmation Fortran 90, HPF (High Performance Fortran) et, plus récemment, Fortran 95 ont pour objectif, entre autres, de permettre l'expression d'algorithmes sous forme intrinsèquement parallèle et de faciliter la parallélisation des programmes. Cependant, la génération de code parallèle présente des problèmes qui ne sont pas encore résolus: comment doit-on distribuer le travail et les données entre les processeurs de façon à minimiser le temps nécessaire à l'exécution d'un programme?

De plus, le modèle de programmation Fortran 90/HPF semble très approprié à la mise en oeuvre d'applications de traitement de signal sur ordinateur SIMD (Single Instruction Multiple Data) mais peu de travaux ont été effectués dans ce sens (pour une description des architectures SIMD, voir [19]).

Finalement, les tableaux traités demandent généralement une quantité considérable de mémoire et accéder rapidement à ces données représente un grand défi parce que, premièrement, lorsqu'une mémoire a une grande capacité, elle est aussi lente et, deuxièmement, le calcul d'adresse des éléments des tableaux demande un effort de calcul substantiel.

Dans cette thèse, on présente des méthodes permettant de résoudre ces trois problèmes (partitionnement, programmation de SIMD à haut niveau et accès rapide aux éléments de tableaux). Le chapitre 2 présente une revue de la littérature sur ces trois sujets alors que le chapitre 3 décrit les besoins des applications qu'on vise à supporter. Le chapitre 4 traite de la génération d'adresses alors que le chapitre 5 aborde le sujet de la programmation à haut niveau d'ordinateurs SIMD et que le chapitre 6 décrit la méthode de parallélisation de programmes HPF proposée. Au chapitre 7, on généralise et formalise le modèle de parallélisme du HPF. Finalement, le chapitre 8 tire les conclusions de cette thèse. En annexe A, on trouve une introduction à MOA (qui est utilisé dans le chapitre 7) alors que l'annexe C contient la grammaire du langage HPCP (qui est décrit dans le chapitre 5). Il est à noter que le chapitre 5 vise spécifiquement les architectures SIMD alors que les autres chapitres font abstraction de l'architecture de l'ordinateur.

1.1 Modèle de parallélisme de HPF

Étant donné que cette thèse présente des travaux relatifs au langage de programmation HPF et que ce dernier est présentement relativement peu utilisé, cette section décrit brièvement le modèle de parallélisme supporté par ce langage.

Ce modèle de parallélisme consiste uniquement en un parallélisme sur les données. Pour paralléliser un programme HPF, un programmeur doit, dans un premier temps, indiquer au compilateur comment aligner les tableaux. Deux éléments de tableaux (différents) qui sont alignés l'un par rapport à l'autre seront traités par le même processeur (une fois que les tableaux seront distribués). Cette étape permet de forcer des éléments de tableaux qui interagissent à être situés dans la mémoire du même processeur (sur un ordinateur à mémoire distribuée), ce qui, en général, permet de réduire les communications. Dans le cas d'un ordinateur à mémoire partagée, cela permet d'améliorer l'efficacité des antémémoires qu'on y retrouve généralement.

La deuxième étape de description de la parallélisation consiste à indiquer au compilateur comment effectuer le partitionnement des tableaux. Le modèle de partitionnement consiste à décrire l'ordinateur sous forme d'un tableau de processeurs et de diviser certaines des dimensions des tableaux de données selon la longueur des dimensions du tableau de processeurs. Une dimension d'un tableau de données peut être divisée en blocs, de façon cyclique (avec des groupes d'un élément ou plus par processeur) ou elle peut ne pas être partitionnée du tout. Finalement, les dimensions du tableau de processeurs sont utilisées en ordre lexicographique. Le programmeur peut décider de ne pas spécifier la forme du tableau de processeurs, auquel cas, le compilateur est libre de générer le code de façon à profiter au mieux du système utilisé.

1.1.1 Directives de parallélisation HPF

La description du partitionnement selon le modèle décrit ci-haut se fait à l'aide de directives, c'est-à-dire des instructions qui n'effectuent aucun traitement sur les variables impliquées mais qui donnent des indications (suggestions) au compilateur au sujet du partitionnement (que le compilateur peut décider d'ignorer).

La description de ces directives, dans ce qui suit, est partielle et est fonction des besoins présents; pour une description complète voir [24]. Les exemples de directives

```
REAL EX1(100, 100, 100, 100)
REAL EX2(100, 100, 100)
REAL EX3(100, 100)

!HPF$ TEMPLATE :: TEMPO(200, 200, 200, 200)
!HPF$ DISTRIBUTE(BLOCK, CYCLIC, CYCLIC(5), *) :: TEMPO
!HPF$ ALIGN EX1(i0, i1, i2, :) WITH TEMPO(i1, i0 + 3, 2 * i2, :)

!HPF$ PROCESSORS PROCO(10)
!HPF$ DISTRIBUTE(BLOCK, *) ONTO PROCO :: EX3
!HPF$ ALIGN EX2(:, *, :) WITH EX3(:, :)
```

Figure 1.1: Exemples de directives HPF

qui accompagnent les explications qui suivent sont données à la figure 1.1.

Les directives sont vues par un compilateur Fortran 90 comme des commentaires; pour ce faire, elles débutent par le caractères "C" ou "!" selon le format utilisé (fixe ou libre — voir [2]). Pour indiquer à un compilateur HPF que ce sont des directives, ce premier caractère est immédiatement suivi par la chaîne de caractères "HPF$".

La directive permettant d'aligner deux tableaux ou un tableau et un gabarit ("template" — voir le prochain paragraphe) est "align". Cette directive permet, à l'aide de variables présentes dans les expressions des deux entités, de spécifier quelles dimensions sont alignées, avec quel déphasage et avec quel pas relatif. Par exemple, la directive de la figure 1.1, qui aligne la variable EX1 et le gabarit TEMPO, aligne les dimensions 1, 2, 3 et 4 de EX1 avec les dimensions 2, 1, 3 et 4 de TEMPO respectivement. De plus, le déphasage entre les éléments du gabarit selon sa première dimension et de la variable selon sa deuxième dimension est de 3 alors qu'il est de 0 pour les autres dimensions (i.e. les éléments (i, j, k, l) de EX1 sont sur le même processeur que les éléments $(j, i+3, k, l)$ de TEMPO et ce, pour j, k, et l allant de 1 à 100 inclusivement et i allant de 1 à 97 inclusivement). Un raisonnement similaire s'applique au fait que le pas selon la troisième dimension de TEMPO est deux fois plus grand que celui de EX1 selon cette même dimension. Finalement, pour indiquer qu'on ne doit pas tenir compte d'une dimension lors de l'alignement, le caractère "*" est utilisé (voir l'alignement de EX2 avec EX3).

Un gabarit est un tableau fictif (i.e. il ne cause aucune allocation de mémoire)

permettant d'aligner différents tableaux entre eux et de leur donner la même distribution. Ceci est nécessaire lorsque l'on veut distribuer des tableaux alignés entre-eux avec un déphasage non-nul ou avec des dimensions permutées parce qu'alors les frontières des tableaux ne sont pas toujours situées au même endroit dans l'espace d'indexation donc une directive de distribution ne serait pas suffisante pour spécifier complètement la distribution des tableaux. Un gabarit est décrit par la directive "template" (voir la figure 1.1).

La directive permettant de décrire la distribution d'un tableau ou d'un gabarit est "distribute". Pour chaque dimension de l'entité à distribuer, il faut utiliser soit "block", soit "*", soit "cyclic" (ce dernier avec ou sans un entier entre parenthèses). La première de ces possibilités permet d'indiquer qu'on veut que la dimension soit divisée en groupes de sous-tableaux les plus gros possibles à raison d'un groupe par processeur. La deuxième alternative permet de spécifier que la dimension ne doit pas être distribuée alors que la troisième permet de forcer le nombre d'éléments par processeur à une valeur égale au paramètre spécifié ou à 1 si ce dernier est absent. Pour la distribution cyclique, si le nombre de processeurs selon la dimension concernée multiplié par le nombre d'éléments par processeur est inférieur au nombre d'éléments du tableau selon cette dimension, on continue la distribution à partir du premier processeur et ce, jusqu'à ce qu'il ne reste plus d'éléments du tableau à distribuer. Ainsi, pour TEMP0, le processeur (1, 1, 1, 1) (i.e. le "premier" processeur) recevra la section $(1 : 200/p_1, 1 : 200 : p_2, 1 + 5 \times p_3 \times i : 5 + 5 \times p_3 \times i, :)$ où $0 \leq i < 200/(5 \times p_3)$. Les p_i indiquent la forme du réseau de processeur.

La directive "processors" permet de décrire la forme d'un réseau de processeurs (tel PROC0 dans l'exemple) et ce réseau peut être utilisé ensuite pour distribuer un tableau ou un gabarit (EX3 dans l'exemple). Le réseau est vue comme un tableau de processeurs.

1.2 Architecture du SIMD de Pulse

Étant donné que certaines explications contenues dans la présente thèse font référence à certains aspects de l'architecture du SIMD de **Pulse**, on présente ici un résumé de cette architecture. La figure 1.2 contient un diagramme-bloc représentant ladite architecture. On y remarque que les processeurs élémentaires (PE) communiquent

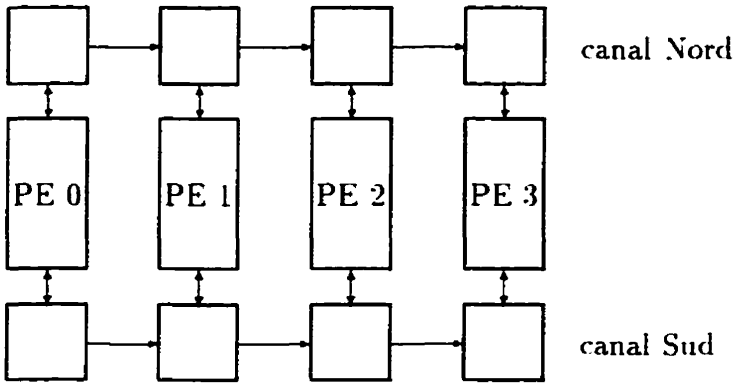


Figure 1.2: Architecture du SIMD de **Pulse**

via des canaux de communication nommés **Nord** et **Sud**. Ces canaux fonctionnent sous forme de registre à décalage c'est-à-dire que, lorsqu'une instruction de communication est effectuée, les données présentes dans le canal affecté par l'instruction vont d'un processeur à son voisin de droite. Les instructions de communication sont appelées **nsr** et **ssr** ("North shift right" et "South shift right").

Le répertoire d'instructions de **Pulse** comprend les instructions arithmétiques et logiques habituelles ainsi que les instructions de contrôle qu'on retrouve sur les processeurs SIMD. En plus, l'architecture **Pulse** comprend des instructions dédiées au traitement d'images qui utilisent plus de deux opérandes et/ou produisent plus d'un résultat (par exemple, "Compare-and-swap", "rank", "min" et "max" à trois sources). Également à signaler est le fait que les PEs peuvent effectuer des instructions vectoriels. Les informations contenues dans les instructions vectorielles sont les mêmes que dans les instructions scalaires auxquelles on ajoute le nombre de fois que l'instruction est répétées ainsi que l'incrément aux numéros de registre ou aux adresses utilisés dans l'instruction.

Chapitre 2

Revue de la littérature

2.1 Partitionnement de boucles imbriquées

Par le passé, la plupart des travaux qui visaient à paralléliser le traitement de tableaux s'attachaient aux algorithmes implantés sous forme de boucles imbriquées comme c'est le cas lorsqu'on utilise des langages de programmation comme le FORTRAN 77 ou le langage C (par exemple, voir [47]). Dans ce cas, on utilise souvent le concept de vecteur de dépendance [48]. Si l'on partitionne le tableau parallèlement aux vecteurs de dépendance (si c'est possible), alors il n'y a aucune communication causée par le partitionnement. Cependant, ces conditions ne sont presque jamais rencontrées en pratique, ce qui fait qu'on doit trouver des heuristiques qui permettent de faire un compromis entre les différents vecteurs de dépendance et assembler les itérations en groupes qui seront assignés aux différents processeurs [17, 31, 46].

Il y a trois problèmes associés à cette approche: le premier est qu'on peut difficilement qualifier la qualité de la solution parce qu'on utilise des heuristiques et que la solution optimale n'est pas connue. Le deuxième problème est qu'on n'utilise pas une approche systématique et, donc, que la solution ne s'intègre pas dans un cadre conceptuel clair et bien défini. Ceci peut avoir plusieurs résultats négatifs; par exemple, certains chercheurs reproduisent essentiellement les mêmes travaux que d'autres mais utilisent un vocabulaire différent: comme les concepts véhiculés dans [44] par rapport au concept de vecteur de dépendance [46]. Le troisième problème est que certains de ces travaux ne tiennent pas compte de la répartition équitable de la charge de travail entre les processeurs. Or, ceci est un problème important puisque,

si un seul processeur est occupé pendant un temps (même petit), le gain de vitesse peut se dégrader considérablement. Ce phénomène est mis en lumière par la loi d'Amdahl [23, p. 575]. Par exemple, si un seul processeur, dans un ordinateur qui en contient 100, a une tâche qui est de 10% plus longue que celle des autres processeurs, alors le gain de vitesse est d'environ $\frac{1}{0.001+\frac{0.999}{100}} \simeq 91$. Donc, dans cet exemple, près de 10% du gain de vitesse potentiel est perdue à cause d'une petite différence entre les charges de travail. Il apparaît donc plus important d'avoir une bonne répartition du travail que d'avoir un algorithme de partitionnement optimal au niveau des communications. Il est à noter, cependant, que le modèle HPF ne permet pas de séparer les tableaux de façon à bien répartir le travail car, dans le cas général, les frontières des tableaux nécessitent un moins grand effort de calcul que le "centre".

2.2 Partitionnement de code de haut niveau

La deuxième approche de partitionnement consiste à utiliser un langage de programmation qui contient des opérateurs sur les tableaux comme, par exemple, l'APL [26], le langage J [27], NIAL [28] ou Fortran 90 [2]. Bien que certains chercheurs aient exploré cette avenue par le passé (par exemple, [14]), c'est la venue du Fortran 90 et du HPF qui a suscité un intérêt marqué parmi les chercheurs vis-à-vis cette approche.

Les chercheurs s'entendent pour dire que le partitionnement automatique demande beaucoup d'effort de calcul ce qui fait que plusieurs utilisent des heuristiques pour diminuer cet effort. Les approches préconisées peuvent être classées en deux catégories: celles qui sont basées sur l'évaluation de partitionnements-candidats et celles qui "calculent" la meilleure solution possible (en éliminant les contraintes les moins coûteuses — voir chapitre 6). Dans [29], une méthode du premier groupe est proposée. Malgré que cette méthode ne considère pas l'alignement intra-dimensionnel, les heuristiques utilisées ont une complexité temporelle trop élevée pour être implantées dans un compilateur. Chatterjee [13, 11, 12] propose des algorithmes pour calculer l'alignement et la distribution des tableaux. Cependant, bien que la méthode utilisée permette une redistribution dynamique, les alignements sont choisis avant les distributions donc, même si une dimension n'est, à la fin, pas distribuée, elle influencera, malgré tout, l'alignement des tableaux puisque toutes les dimensions sont utilisées pour calculer le coût des différents alignements considérés.

Bau *et al.* [7] proposent une approche algébrique dont la complexité est faible; cependant, ils n'abordent pas le problème de la sélection des contraintes à satisfaire (ou non) et ils ne résolvent que le problème de l'alignement sans traiter la distribution des tableaux. Finalement, Knobe [32, 33] utilise un graphe pour décrire les contraintes. Les contraintes qui seront satisfaites sont sélectionnées en construisant un arbre recouvrant qui est ensuite augmenté des autres contraintes qui ne causent pas de conflits. Cette méthode est très performante mais elle est limitée aux systèmes SIMD et elle utilise (comme [29]) un modèle plus général que celui du HPF. Il est à noter que le fait que cette méthode soit limitée aux systèmes SIMD est important puisque la synchronisation rigide des processeurs rend l'efficacité du partitionnement plus sensible à la répartition de la tâche car un déséquilibre dans les sous-ensembles de chaque tâches ralentit le traitement alors que, dans le cas d'un ordinateur MIMD, le ralentissement n'est fonction que du déséquilibre entre l'ensemble des tâches des processeurs. Or, comme il a été souligné précédemment, le modèle HPF se prête mal à l'équilibrage fin des tâches, donc, viser à supporter le modèle HPF et les architectures SIMD sont des objectifs relativement conflictuels et c'est pourquoi il a été décidé, dans cette thèse, de mettre l'emphase sur le modèle HPF uniquement.

Donc, aucune méthode, à la fois:

- ne supporte le modèle de partitionnement du HPF,
- a une faible complexité temporelle,
- permet tant l'alignement que la distribution des tableaux et
- vise les calculs structurés (voir chapitre 3).

On vise, dans cette thèse, à proposer une solution qui rencontre ces besoins.

2.3 Géneration d'adresse pour les tableaux

La génération d'adresses pour le traitement de tableaux n'a pas attiré l'attention de beaucoup de chercheurs. En fait, aucune référence sur ce sujet n'a été trouvée. Par contre, il existe un circuit intégré, le TMC2301 [45], qui effectue le calcul de coordonées de matrices. Étant donné qu'on ne dispose d'aucune information sur le fonctionnement interne de ce circuit, on ne peut en discuter.

2.4 Langages de haut niveau pour la programmation SIMD

Plusieurs langages de haut niveau ont été proposés pour la programmation d'ordinateurs SIMD et MIMD. Fortran 90 [2], HPF (High Performance Fortran) [24] et APL [26] sont des exemples des langages à usage général qui, de par leur généralité, ne sont pas appropriés au traitement structuré de tableau parce que cette généralité nuit à la génération de code compact et efficace.

Un des objectifs de la présente thèse est de permettre la distribution automatique des données et du traitement entre les processeurs. Donc, des structures de contrôle définissant des blocs séquentiels et d'autres parallèles (comme dans Occam [38], Uc [6] et BLAZE [34]) doivent être évitées. Également, il est préférable d'inclure dans le langage les expressions sur des tableaux pour permettre un niveau d'abstraction plus élevé ainsi que pour faciliter la distribution automatique des tableaux donc, les boucles imbriquées (comme dans Apply [21] et AL [50]) sont à éviter. Évidemment, le "forall' (comme dans BLAZE) et le concept d'ensemble d'index (comme dans Uc) sont intéressants mais ne sont pas suffisants dans le présent contexte. Donc, aucun langage ne possède toutes les caractéristiques requises pour décrire le traitement structuré de tableaux à un niveau d'abstraction élevé et qui permettrait la génération de code parallèle efficace (en mémoire et en temps). Un nouveau langage est proposé dans la présente thèse dans le but de combler ce vide.

Chapitre 3

Analyse des besoins

Dans ce chapitre, on tente de donner un aperçu des besoins typiques des applications qui traitent des tableaux et qui requièrent une grande puissance de calcul. Les besoins des applications diffèrent beaucoup, évidemment. Par contre, on peut diviser les algorithmes de traitement de tableaux en deux grandes catégories: ceux qui effectuent un traitement régulier et ceux dont la structure est irrégulière.

Les méthodes numériques qui discrétisent des équations mathématiques et en font des systèmes d'équations algébriques linéaires sont très répandues et comprennent (entre autres): les méthodes aux différences finies, la méthode des éléments finis, les méthodes de volumes finis ainsi que les méthodes multi-grille. Ces méthodes ont en commun une importante caractéristique: les systèmes d'équations générés sont creux (au sens où chaque équation dépend d'une petite fraction seulement des variables). Ceci est dû au fait que les relations entre les variables sont de nature très localisée (seuls des éléments voisins dans des tableaux de données interagissent). Un bon exemple de cette situation est montré dans [25] aux pages 417 et suivantes. Il s'agit de la solution d'une équation aux dérivées partielles à l'aide d'une méthode de différences finies. La matrice qui décrit ce système est creuse à cause de la façon dont la solution est exprimée; plus précisément, les inconnues sont assemblées en un vecteur plutôt qu'en une matrice (qui représente la forme du problème).

Il existe d'autres situations pour lesquelles la représentation matricielle produit une matrice creuse; par exemple, un problème dont la structure est un réseau (comme un réseau de distribution d'énergie) dont la connectivité est faible.

Pour ces deux types de situations (structures régulières ou non mais représentées sous forme matricielle), les chercheurs s'entendent pour dire que les méthodes itératives sont plus efficaces (du point de vue de l'effort de calcul nécessaire pour obtenir une solution à la précision désirée) que les méthodes directes [36]. Donc, sachant que la représentation matricielle n'est utile que pour les méthodes directes, il est plus important de supporter les méthodes itératives que les méthodes directes (i.e. le calcul sur des tableaux plutôt que sur des matrices). Cette assertion est aussi supportée par le fait que beaucoup de méthodes ont été proposées pour solutionner des systèmes dont la structure est particulière (par exemple, les systèmes tridiagonaux et pentadiagonaux). Ces systèmes ont la même structure que ceux créés par des méthodes itératives, mais ils sont parfois solutionnés en n'ayant pas recours à une méthode itérative (exemple: méthode des éléments finis sur une grille régulière).

Donc, supporter le traitement de tableaux fait à la manière des méthodes itératives est important car ces dernières sont plus faciles à supporter (car le traitement est régulier — voir section 3.1), elles sont de plus en plus répandues et certaines méthodes directes ont une structure de calcul similaire.

3.1 Méthodes itératives

Les méthodes itératives (ainsi que d'autres algorithmes qui ont une structure de calculs similaire) sont très répandues (e.g. méthodes aux différences finies, méthodes multi-grilles, méthodes de volumes finis, algorithmes de traitement de signal). L'équation $A \times T + B = 0$ est la représentation sous forme matricielle d'un système à résoudre (où A est la matrice, T est le vecteur des inconnues et B est un vecteur de constantes). Solutionner ce système de façon itérative consiste à assigner une valeur initiale quelconque à T et de calculer la valeur du membre de gauche de l'équation. Le résultat ne sera pas zéro comme dans l'équation mais un certain ΔT qui sera ajouté à T pour obtenir une deuxième valeur à T qui soit plus près de la véritable solution. Autrement dit, $T_{n+1} = T_n + \Delta T$ pour chaque itération et où ΔT est calculé à l'aide du membre de gauche de l'équation. Il existe différentes variantes à cette méthode de base pour accélérer la convergence et la précision, mais elles ne seront pas décrites ici puisque cela dépasse le cadre des présents travaux. Il est important de noter que, comme ces systèmes d'équations représentent des calculs structurés sur

des éléments de tableau qui sont voisins. $A \times T$ est, en fait, une convolution.

3.2 Méthodes directes

Les méthodes directes les plus connues sont la décomposition LU et l'élimination gaussienne. Ces méthodes ont en commun un problème qui est celui dit de remplissage ("fill-in"). Ce problème vient du fait que ces méthodes transforment la matrice qui décrit la solution en une autre qui a un taux de remplissage plus élevé. Donc, solutionner le système une fois la matrice transformée impose plus de calculs que requis par la structure du problème.

Ces méthodes sont souvent utilisées lorsque l'on veut décrire de façon simple un problème non-structuré, par exemple, des graphes ou réseaux. Ces graphes et réseaux peuvent décrire beaucoup de types de systèmes différents comme: des systèmes continus mais discrétisés de manière non-structurée, des systèmes discrets comme des réseaux logiques ou de distribution de puissance ou des systèmes composés de "particules" comme des molécules et/ou atomes ou des corps planétaires.

Dans ce genre de situation, l'approche la plus judicieuse est un algorithme où les données ne sont pas structurées (par exemple, en graphe) et où la méthode de solution est itérative.

3.3 Contexte matériel

Il a été démontré que, parmi la classe de réseau "k-ary n-cube", le choix optimal consiste à utiliser un réseau de deux (rarement trois) dimensions lorsqu'on désire minimiser la latence des communications, maximiser la bande passante et/ou minimiser l'impact des points chauds sur la performance globale [16]. Étant donné que la plupart des ordinateurs fabriqués à ce jour utilisent ce genre de réseau et que cette structure correspond très bien au matérial disponible (en terme de forme), il est raisonnable de penser que de ne supporter, au niveau logiciel, que des réseaux en forme de tableaux est un choix judicieux, puisqu'il inclut ces "k-ary n-cube".

D'autre part, les problèmes qui demandent un grand effort de calcul traitent des données sous forme de tableaux d'au moins deux dimensions (généralement trois). Donc, la dimensionalité des tableaux de données est normalement au moins aussi

grande que celle du réseau de processeurs. Ceci implique qu'il est raisonnable de ne pas permettre à une dimension de tableau de données d'être partitionnée plus d'une fois. Donc, à cet égard, le modèle HPF ne limite la performance que dans des cas particuliers.

3.4 Discussion

Étant donné que la convolution est l'opération la plus utile pour les méthodes itératives (avec les opérations arithmétiques, évidemment), il est important de s'y attarder. La caractéristique principale de cette opération est l'extrême régularité de la structure des calculs et du patron d'accès à la mémoire. Ce point est intéressant puisqu'il permet d'effectuer beaucoup d'optimisations à la compilation (par exemple, utiliser les unités fonctionnelles et leur pipeline au maximum, transférer de façon optimale les données entre la mémoire et les antémémoires). Également, et c'est là un des points qui nous intéressent, trouver le partitionnement idéal est beaucoup plus facile que pour des structures de calculs moins régulières. En effet, la forme du noyau de convolution donne directement la forme des sous-tableaux (ou sections) qui devront être transférés d'un processeur à un autre si un tableau est partitionné.

Il est à noter que la convolution est l'opération principale de plusieurs algorithmes de traitement de signal. En effet, cette opération constitue la structure de base des filtres (par exemple, FIR, IIR, filtres polyphasés) et que ces filtres implantent les fonctions de base du traitement de signal (par exemple, lissage, détection d'arêtes, réduction du bruit). Ceci rend la classe des applications qui effectuent un traitement régulier sur des tableaux encore plus importante.

La seule autre opération qui soit souvent utilisée et qui puisse causer des communications, lorsqu'un programme est parallélisé, est la réduction. Celle-ci est surtout utilisée dans des problèmes de déconvolution. Cette opération est également relativement simple à gérer (lorsqu'il s'agit de trouver le partitionnement optimal) puisque, si une seule dimension est réduite, une quantité considérable de données devrait être transférée si on partitionnait cette dimension. De plus, si un tableau est réduit au complet, on ne peut pas avoir une quantité "raisonnable" de communications, si on utilise un des langages impératifs les plus utilisés, quelles que soient les dimensions partitionnées. En effet, les langages les plus courants doivent définir

dans quel ordre sont effectuées les opérations de calcul puisqu'un des besoins les plus importants des usagers des applications est que les résultats doivent être les mêmes quel que soit l'ordinateur qui a exécuté l'application. Il s'en suit qu'on ne peut pas, normalement, effectuer la réduction des partitions pour, ensuite, faire la réduction de ces résultats. Donc, la quantité de communication est strictement fonction de la forme du réseau de processeurs et des dimensions partitionnées et réduites.

Finalement, il est à noter que de supporter efficacement le traitement de matrices creuses n'est pas désirable puisque la répartition des tâches entre les processeurs est très difficile (et qu'elle doit être modifiée pendant l'exécution) et qu'il est plus difficile d'utiliser efficacement les ressources d'un ordinateur (par exemple, les antémémoires et les unités fonctionnelles) à cause de l'absence de régularité dans le traitement.

3.5 Mise en oeuvre d'un objet-tableau C++

Dans le but de déterminer quels opérateurs sur les tableaux sont les plus utiles, un object-tableau a été implanté dans le langage C++. Les opérateurs implantés *a priori* sont les opérateurs arithmétiques, les opérateurs les plus usuels de MOA [41] (voir l'annexe A) ainsi qu'un opérateur de convolution.

Également, deux applications ont été traduites en C++ en utilisant cet objet.

3.5.1 Opérateurs implantés

Les opérateurs implantés sont:

- les opérateurs arithmétiques habituels ($+$, $-$, $*$, $/$) entre deux tableaux et entre un tableau et un scalaire,
- les assignations C++ suivantes $=$, $+=$, $-=$, $*=$, $/=$,
- les comparaisons $>$, $<$, $==$, $!=$, $<=$, $>=$,
- des opérateurs de lecture et d'écriture de tableaux: $<<$, $>>$,
- des opérateurs logiques: **set_gt**, **set_lt**, **set_eq**, **set_ne**, **set_le** et **set_ge**,
- des fonctions mathématiques usuelles: **max**, **pow**, **sqrt** et **abs**,

- des opérateurs MOA: **reshape**, **delta**, **rho**, **tau**, **iota**, **red_add**, **red_mult**, **take**; ainsi que deux opérateurs effectuant le travail combiné de deux opérateurs **omega_mult**, **omega_add**,
- un opérateur d'indexation [ ] et
- deux opérateurs de convolution.

3.5.2 Applications implantées

Les applications implantées consistent en un problème aux différences finies et un problème de déconvolution. Le premier consiste à simuler un écoulement de fluide à l'aide du schème de MacCormack [20], alors que le deuxième consiste à extraire un estimé du champ de vent à partir de données de précipitation provenant d'un radar Doppler [35]. Dans ce deuxième cas, le schéma semi-lagrangien a été remplacé par un schéma de différences finies parce qu'il régularise la structure des calculs et demande un effort de calcul moindre.

3.6 Conclusions

Les opérateurs qui se sont révélés utiles sont:

- $+, -, *, /, =, *=, +=, -=$,
- I/O (<<, >>),
- convolution,
- réduction additive,
- **omega_mult** (i.e. le "spread" du Fortran 90 ou $_{x}\Omega$ en MOA),
- **set_lt**,
- **take**, **rho**, **delta**,
- **sqrt**, **pow**.

L'utilité de la plupart de ces opérateurs n'est pas étonnante. Cependant, il est à noter que certains opérateurs ne se sont avérés utiles que parce que le cadre de cette implantation est plus contraignant que les langages de programmation impératifs usuels; en particulier, **set_lt**, **omega_mult** et **take** ne seraient pas utiles en Fortran 90, par exemple.

Aussi, la convolution peut être exprimée assez facilement à l'aide de sections de tableaux en Fortran 90 (bien que l'expression peut devenir très longue dans certains cas); de plus, un opérateur de convolution souffre d'un certain manque de souplesse. En effet, le traitement des frontières des tableaux requiert un traitement spécifique qui peut difficilement être exprimé à l'aide d'un opérateur. Il semble donc que la solution idéale pour l'expression de la convolution reste à trouver.

En conclusion, l'objectif de ce chapitre est d'identifier les opérateurs les plus importants pour les applications typiques. On constate que ces opérateurs sont les opérateurs de calculs (arithmetiques, **sqrt**, **pow** etc.), la convolution, la réduction et le "spread" du Fortran 90.

Chapitre 4

Génération d'adresses

4.1 Introduction

Une des difficultés majeures rencontrées lorsqu'on vise à maximiser la performance d'un processeur consiste à transférer les données entre la mémoire principale et le processeur de façon à ne pas ralentir ce dernier dans l'exécution de ses tâches. Ce ralentissement peut survenir dans deux situations: 1. les données dont le processeur a besoin ne sont pas encore disponibles et 2. les données dont le processeur n'a plus besoin engorgent la mémoire locale (comme, par exemple, dans une anté-mémoire ou dans des registres) et empêchent le chargement de données dont le processeur a besoin. De plus, les technologies courantes ne permettent pas de fabriquer des mémoires principales qui soient aussi rapides que les processeurs, tout en ayant une capacité de stockage suffisante pour contenir les programmes et données associées des applications usuelles. Ceci a pour conséquence l'utilisation d'une hiérarchie de mémoires dans la plupart des ordinateurs de haute-performance (par exemple, anté-mémoires, mémoires statiques sur la puce, registres vectoriels) pour diminuer l'impact de cette différence de vitesse.

Un défi majeur dans la conception d'un ordinateur consiste à compenser la latence élevée de la mémoire principale qui résulte des caractéristiques de la technologie utilisée. Par contre, la bande passante de cette mémoire peut être augmentée de façon relativement simple en améliorant la structure du sous-système-mémoire (par exemple, en utilisant des bus plus larges, des mémoires entrelacées ou un mode de transfert en rafale). En d'autres mots, bien que la bande passante puisse être

améliorée assez facilement, diminuer la latence nécessite une technologie plus rapide et dispendieuse, sauf si on peut effectuer des chargements et déchargements anticipés (ce qui est la seule autre alternative). Évidemment, ces deux solutions peuvent être utilisées simultanément.

Le traitement de tableaux permet d'effectuer ces transferts anticipés puisque les patrons d'adresses qu'ils entraînent sont réguliers donc prévisibles. Dans ce chapitre, on discute d'un algorithme permettant de transférer efficacement un tableau entre un processeur et sa mémoire. Deux implantations de cet algorithme (une logicielle et une matérielle) sont décrites et on démontre leur efficacité ainsi que la complexité de l'implantation matérielle.

L'algorithme proposé est intimement lié à l'adressage symétrique de tableaux tel que proposé par Becker [8]. L'adressage symétrique consiste à décrire l'adresse d'un élément d'un tableau comme une somme pondérée des indices de l'élément (à laquelle on ajoute une adresse de base). Certains des gains apportés par l'adressage symétrique sont:

- éviter de copier de grandes quantités de données lorsqu'un tableau est transformé,
- permettre de fusionner plusieurs transformations en une seule par la composition de celles-ci (c'est-à-dire que ces transformations peuvent être combinées en une seule au moment de la compilation et que le temps d'exécution peut être réduit au temps d'une seule transformation),
- permettre de traiter des tableaux non-contigus en mémoire.

On peut éviter de copier les données parce qu'une transformation qui ne modifie que l'adressage d'un tableau (par exemple, la transposition, ou extraire une section) est effectuée en modifiant simplement les facteurs de pondération.

L'algorithme proposé utilise une transformation linéaire d'un ensemble de vecteurs d'indices en une séquence d'adresses et calcule cette séquence de façon très efficace. On prouve cette efficacité en montrant que le temps nécessaire pour calculer les adresses pour un cas complexe (c'est-à-dire pour un tableau ayant de nombreuses dimensions) est presque aussi rapide que pour le cas simple (c'est-à-dire un vecteur). Un gain majeur apporté par cet algorithme est qu'il ne nécessite aucune

multiplication au moment de l'exécution contrairement à l'algorithme classique permettant de calculer une adresse à partir d'un vecteur d'indices [3]. Évidemment, un sous-système-mémoire de haute performance est nécessaire pour utiliser au mieux les capacités d'un tel algorithme mais ce sujet dépasse le cadre de cette thèse.

La section 4.2 décrit les types de transformations qui doivent être supportés pour que l'algorithme soit suffisamment flexible et performant pour être considéré utile et général pour le traitement de tableaux. La section 4.3 décrit l'algorithme proposé tandis que la section 4.4 contient une description de l'implantation logicielle et du niveau de performance qu'elle permet. La section 4.5 fait de même pour l'implantation matérielle. Dans la section 4.6, différentes transformations sont décrites alors que la section 4.7 définit les équations nécessaires au calcul des pas utilisés par le générateur d'adresses (c'est-à-dire l'implantation matérielle). Finalement, la section 4.8 tire des conclusions sur ce sujet.

Dans ce chapitre, on utilise une notation basée sur la syntaxe du Fortran 90 [2]. Les dimensions sont numérotées de 1 pour la dimension de poids fort au nombre de dimensions du tableau pour la dimension de poids faible.

4.2 Patrons d'adresses

Lorsque des tableaux sont transformés dans une application, ils le sont par un opérateur ou, dans le cas du Fortran 90, ils peuvent aussi l'être par une fonction intrinsèque. Le Fortran 90 est utilisé en guise de référence parce que les différents dialectes de Fortran sont utilisés pour programmer la plupart des applications scientifiques qui nécessitent beaucoup de temps de calcul et que ces applications traitent généralement des tableaux. Les fonctions intrinsèques du Fortran 90 qui transforment un tableau sont: CSHIFT, EOSHIFT, TRANSPOSE, MATMUL, SPREAD et RESHAPE.

Il existe une autre façon de transformer un tableau: utiliser seulement une partie du tableau soit une section (selon le vocabulaire du Fortran 90). En Fortran 90, on décrit une section par une borne inférieure, une borne supérieure et un pas selon chacune des dimensions.

Toutes ces opérations (sauf CSHIFT) impliquent une transformation linéaire d'un ensemble de vecteurs d'indices vers un ensemble d'adresses ce qui fait que, pour

accéder aux tableaux **sans les copier en mémoire** (donc, en les accédant sur place) on n'a besoin que de:

1. l'adresse du premier élément (après la transformation).
2. la forme du tableau transformé et
3. la distance en mémoire entre deux éléments contigus selon chacune des dimensions du tableau.

Autrement dit, bien que les différentes opérations impliquent différents patrons d'accès en mémoire (par exemple, TRANSPOSE peut introduire un pas négatif), les données énumérées ci-dessus sont les seules requises pour effectuer les calculs d'adresses. Il est à noter que CSHIFT nécessite l'accès à deux sous-tableaux qui, chacun, impliquent une transformation linéaire. Ces deux transformations, si elles étaient combinées ne constitueraient pas une transformation linéaire.

4.3 Algorithme

L'agorithme de la figure 4.1 implante les calculs désirés et il est proche de l'algorithme désiré. La différence est qu'il ne supporte pas un nombre variable de dimensions.

Cet algorithme montre que:

- le vecteur "cur" est utilisé pour mémoriser les indices de l'élément courant du tableau,
- le tableau entier est traversé (c'est-à-dire du vecteur d'indices $\vec{0}$ au vecteur d'indices $\vec{\text{shape}} - 1$).

L'algorithme proposé est énoncé à la figure 4.2. Le "while" extérieur (lignes 7 à 28) est exécuté jusqu'à ce que le traitement du tableau soit complété. Le premier "while" intérieur (lignes 10 à 14 transfère une rangée du tableau et le deuxième (lignes 20 à 27) gère les indices de l'élément courant (sauf le dernier qui est géré par le premier "while" intérieur). La condition de fin de traitement est $j < 0$, puisque cela veut dire qu'un sous-tableau de dimensionalité égale à celle du tableau a été traité (donc, le tableau lui-même). Le vecteur "shape" contient la forme du tableau

```
Adresses(start, shape[ndim], incr[ndim])
    integer : i, res
    integer : cur[ndim]

    res = start
    cur[0] = 0
    Do while cur[0] < shape[0]
        cur[1] = 0
        Do while cur[1] < shape[1]
            cur[2] = 0
            Do while cur[2] < shape[2]
                ⋮
                    cur[ndim − 2] = 0
                    Do while cur[ndim − 2] < shape[ndim − 2]
                        cur[ndim − 1] = 0
                        Do while cur[ndim − 1] < shape[ndim − 1]
                            Move memory[res]
                            res = res + incr[ndim − 1]
                            cur[ndim − 1] = cur[ndim − 1] + 1
                        end do
                        res = res + incr[ndim − 2]
                        cur[ndim − 2] = cur[ndim − 2] + 1
                    end do
                ⋮
            res = res + incr[1]
            cur[1] = cur[1] + 1
        end do
        res = res + incr[0]
        cur[0] = cur[0] + 1
    end do
```

Figure 4.1: Algorithme de calcul d'adresse simplifié

transformé tandis que "cur" est le vecteur d'indices de l'élément de tableau qui est en cours de traitement. "Incr" contient les incréments d'adresse dont on a besoin pour aller du dernier élément d'un sous-tableau au premier élément du sous-tableau suivant (dont la dimensionalité correspond à la position de l'incrément dans son vecteur). Il est à noter que ces incréments sont exprimés en terme de la granularité de la mémoire plutôt que selon le nombre d'éléments de tableaux. Finalement, "res" est le résultat des calculs d'adresses (donc, c'est l'adresse de l'élément courant): sa valeur initiale est celle de l'adresse du premier élément à transférer.

Cet algorithme est équivalent à l'algorithme de la figure 4.1 sauf qu'il permet de gérer un nombre variable de dimensions.

Le deuxième algorithme gère les boucles imbriquées grâce à une boucle qui gère les indices de boucles et les incréments d'adresses. Il a aussi les caractéristiques suivantes:

- il peut effectuer n'importe quelle transformation linéaire entre un ensemble de vecteurs d'indices et une séquence d'adresses,
- il n'utilise que des additions, des comparaisons et des boucles (et aucune multiplication) et
- il peut être facilement divisé en plusieurs portions qui peuvent être exécutées en parallèle (comme démontré ci-après).

La capacité de cet algorithme d'implanter toutes les transformations linéaires vient du fait que, lorsqu'on se déplace selon une dimension, un pas est ajouté à l'adresse courante. Cette addition et les autres effectuées pour les dimensions de poids plus faible effectuent le travail du facteur de pondération de cette dimension dans l'équation de la transformation linéaire.

Le fait que l'algorithme n'utilise aucune multiplication est un facteur important en ce qui a trait à la vitesse d'exécution parce que la multiplication requiert typiquement plus d'un étage de pipeline contrairement aux opérations logiques, de contrôle et arithmétiques simples. Par exemple, le R10000 a besoin de 6 étages de pipeline pour effectuer une multiplication de nombres entiers de 32 bits [40].

L'absence de multiplication dans l'algorithme implique également qu'une implantation matérielle doit avoir une complexité faible puisqu'un multiplieur rapide est un

```
1   Adresses(start, ndim, shape[ndim], incr[ndim])
2       integer : i, j, res
3       integer : cur[ndim]

4       res = start
5       cur = 0
6       j = ndim − 1
7       Do while j >= 0
8           j = ndim − 1
9           i = 0
10          Do while i < shape[j]
11              accès à mémoire[res]
12              res = res + incr[j]
13              i = i + 1
14              end do
15          If j >= 0
16              j = j −1
17              res = res + incr[j]
18          end if
19          cur[j] = cur[j] +1
20          Do while j >= 0 and cur[j] = shape[j]
21              cur[j] = 0
22              j = j −1
23              If j >= 0
24                  res = res + incr[j]
25                  cur[j] = cur[j] +1
26                  end if
27              end do
28          end do
```

Figure 4.2: Algorithme de calcul d'adresses

module complexe, tant par le nombre de transistors que par son architecture (il s'agit d'une structure semi-régulière).

4.4 Implantation logicielle

4.4.1 Code et modèle temporel

Dans le but de caractériser l'efficacité de l'algorithme, il a été traduit dans le langage C. La partie du programme qui effectue les calculs d'adresses proprement dits est montré à la figure 4.3. Dans ce code, la variable "reg" signifie un registre alors que "res" est l'adresse où est effectué l'accès en mémoire.

```
while(j >= 0)
   {for(i = 0; i < shape[ndim-1]; i++)
      {reg = *res;
      res += incr[ndim-1];
      }

   j = ndim - 2;
   res += incr[j];
   while(j >= 0 && ++(cur[j]) == shape[j])
      {cur[j] = 0;
      j--;
      if(j >= 0)
         res += incr[j];
      }
   }
```

Figure 4.3: Boucles principales de l'algorithme en langage C

Il est à noter que ce fragment de programme nécessite un tableau à deux dimensions ou plus, même si le cas général traité à la section 4.3 supporte tous les cas y compris les tableaux à une seule dimension (qui ne requiert pas un algorithme aussi complexe que celui proposé ici).

L'intérêt principal de ce code est que son temps d'exécution peut être modélisé facilement. Ce modèle ne requiert que quatre paramètres:

t_v: temps nécessaire à l'exécution du "while" extérieur moins le temps d'exécution du "while" intérieur et du "for".

t_w: temps d'exécution d'une itération du "while" intérieur incluant l'exécution de la condition du "if" mais pas celui de l'instruction du corps de ce "if".

t_{if}: temps d'exécution de l'instruction du corps du "if" et

t_l: temps d'exécution d'une itération du "for".

Le modèle du temps d'exécution complet est le suivant:

$$T_t = t_o + t_f + t_r + t_i$$

où

T_t: temps total,

t_o: temps du "while" extérieur,

t_f: temps du "for",

t_r: temps du "while" intérieur et

t_i: temps du "if".

$$t_o = n_i \times t_v$$

$$n_i = n_e / l_r$$

$$t_f = n_e \times t_l$$

$$t_r = n_r \times t_w$$

$$t_i = n_r \times t_{\mathrm{if}}$$

$$n_r = n_e \sum_{i=0}^{\text{n. dim.} -2} \frac{1}{\prod_{j=i}^{\text{n. dim.} -1} s[j]}$$

où

n_i: est le nombre d'itérations du "while" extérieur,

n_e: est le nombre d'éléments du tableau transformé et

l_r: est le nombre d'éléments dans une rangée du tableau.

s: est la forme du tableau.

Dans l'équation de n_r, le produit ($\prod$) calcule le nombre d'éléments d'un sous-tableau de i dimensions, donc n_e divisé par ce nombre d'éléments donne le nombre de sous-tableaux de i dimensions contenus dans le tableau. En conséquence, additionner le nombre de sous-tableaux pour toutes les dimensions sauf celle de poids faible donne le nombre de fois qu'une itération du "while" intérieur a été effectuée (i.e. n_r).

4.4.2 Évaluation du temps d'exécution

Dans le but d'obtenir un estimé des valeurs des paramètres du modèle temporel de l'algorithme, une version du code a été écrite dans un pseudo-langage d'assemblée. Un pseudo-langage a été choisi plutôt qu'un langage réel dans le but d'obtenir un estimé de performance préliminaire et pour déterminer si un modèle réel et des simulations seraient nécessaires (dû, entre autres, à la latence variable des anté-mémoires). Le programme en question est donné à l'annexe B. Ce programme suppose qu'il y a suffisamment de registres pour contenir les variables suivantes:

1. les scalaires i, j et res,
2. la constante ndim,
3. les vecteurs shape, cur et incr et
4. un scalaire temporaire.

Le nombre de registres nécessaires s'élève à 26 si le tableau transféré a 7 dimensions (ce qui est le maximum permis par le Fortran 90). Étant donné que la plupart des processeurs de haute-performance contiennent 32 registres entiers, il est réaliste de supposer que ces variables y sont maintenues.

Pour compléter le calcul de paramètres du modèle temporel, il ne manque qu'un estimé du nombre de cycles nécessaires à l'exécution des différentes instructions. Les règles suivantes ont été appliquées:

- une instruction registre-registre a une latence d'un cycle d'horloge,

- une instruction mémoire-registre a une latence de deux cycles d'horloge.
- une instruction de saut a une latence d'un cycle.

Ces valeurs ont été choisies parce que l'étape d'exécution du pipeline d'un processeur requiert généralement un cycle d'horloge pour être complétée pour une instruction registre-registre (sauf pour la multiplication, la division et les opérations en virgule flottante qui n'apparaissent pas dans le code) et qu'une opération mémoire-registre est plus lente, en général, mais est quand même rapide à cause de la présence d'anté-mémoires sur la plupart des processeurs. Il est à noter que la seconde hypothèse nécessite que les adresses successives soient souvent contigues en mémoire pour obtenir un taux de succès d'accès aux anté-mémoires suffisamment élevé. Étant donné que la destination des instructions de saut est prédite avec un taux de succès élevé par la plupart des processeurs de haute-performance, on a supposé que leur latence est d'un seul cycle d'horloge (c'est-à-dire que la pénalité de mauvaise prédiction est, en moyenne, négligeable). De toute évidence, on suppose que le processeur a une structure en pipeline. Également, on suppose, dans un premier temps, que le processeur n'est **pas** super-scalaire; cette hypothèse sera modifiée plus tard.

À partir de ces hypothèses et du code de l'annexe B, les valeurs des paramètres suivantes ont été calculées:

t_v 4 cycles

t_w 7 cycles

t_{if} 1 cycles

t_l 5 cycles

À l'analyse de ces valeurs, il est évident que l'essentiel du temps d'exécution sera utilisé par la boucle "for", puisqu'elle nécessite 5 cycles pour traiter chaque élément de tableau, alors que le temps de traitement d'une rangée est du même ordre de grandeur.

Cette constatation est corroborée par le calcul des temps d'accès (grâce au modèle décrit à la section 4.4.1) pour un tableau de forme n par 8 par m où $n \times m = 1024$ et n prend les valeurs: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 et 1024. La longueur de la deuxième dimension a été fixée à 8 dans le but de garder le temps de gestion des dimensions supérieures à une valeur relativement constante. La figure 4.4 montre que, pour des longueurs de rangée raisonables, la perte de temps pour la gestion des dimensions supérieures est faible (en comparaison du cas où les rangées ont une

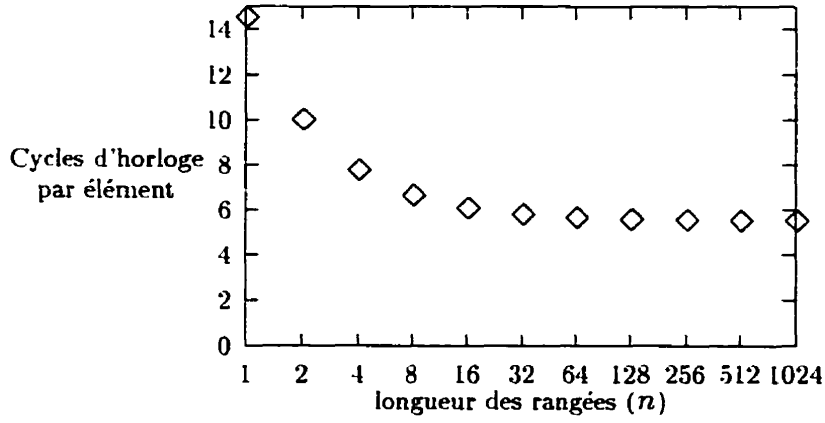


Figure 4.4: Temps moyen de traitement par élément pour un tableau de $n \times 8 \times m$ où $n \times m = 1024$

longueur de 1024, la perte est de $\sim$ 10.5% pour des rangées de longueur 16 et de $\sim$ 5% lorsque cette longueur est de 32).

Dans le but de confirmer cette conclusion, des temps ont été calculés pour un cas où le nombre de dimensions est varié. Un tableau de 4096 par 4096 par n a été changé en un tableau de 5 dimensions de longueur 64 dans chaque dimension, puis en un tableau de 9 dimensions de longueur 8 sauf la dimension de poids faible qui est restée constante. De plus, n a été fixé à 8, 16 et 32. Pour toutes ces situations, la perte de vitesse totale est restée faible (au maximum, $\sim$ 32% pour des rangées de 8, $\sim$ 21% pour des rangées de 16 et $\sim$ 16% pour des rangées de 32 éléments) tel que démontré par la figure 4.5. Cette perte est définie comme étant le temps passé à effectuer d'autres opérations que celles de la boucle "for" (en l'occurence, le "for" demande 5 cycles d'horloge par élément). Ces valeurs de perte sont considérées faibles parce que la longueur des rangées est anormalement faible et que le nombre de dimensions est anormalement élevé (donc, il s'agit d'une situation pessimiste) et que, malgré tout, la perte reste acceptable.

Finalement, une troisième expérience a été effectuée avec une matrice carrée de grandeur réaliste. La figure 4.6 montre que le temps supplémentaire par rapport au traitement d'un vecteur contenant le même nombre d'éléments est négligeable.

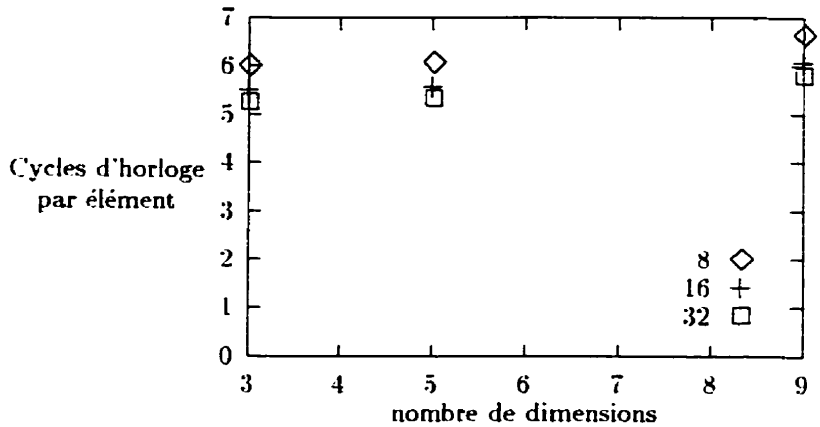


Figure 4.5: Temps de traitement pour un nombre variable de dimensions et un nombre d'éléments constant

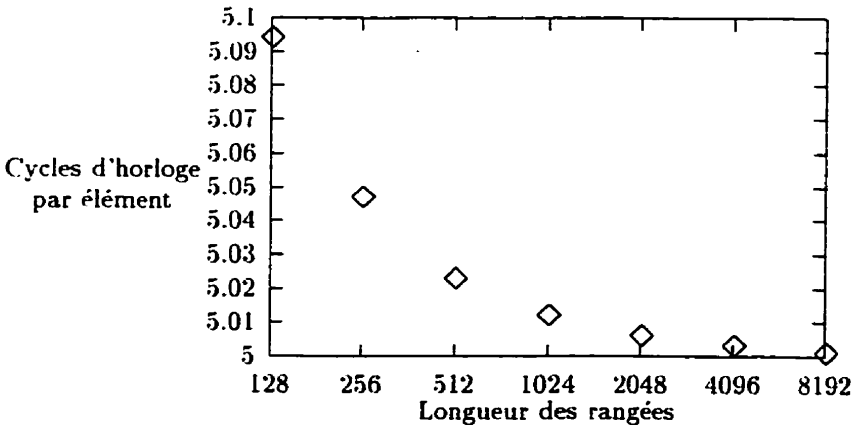


Figure 4.6: Temps de traitement moyen par élément pour une matrice carrée

Tableau 4.1: Résumé des pertes de vitesse pour le processeur simple (S) et pour le processeur super-scalaire (SS) pour les deux premières expériences

Expérience	longueur des rangées	S	SS
E1	16	10.5%	18.5%
E1	32	5%	9%
E2	8	32%	55%
E2	16	21%	33%
E2	32	16%	23%

L'étape suivante consiste à caractériser le comportement d'un processeur super-scalaire. En supposant un processeur pouvant exécuter une opération en virgule flottante, deux instructions en nombre entier et une opération registre-mémoire en même temps (ce qui constitue un processeur minimalement super-scalaire), les paramètres deviennent:

t_v 2 cycles

t_w 5 cycles

t_{if} 1 cycles

t_l 2 cycles

Dans ce nouveau contexte, les deux premières expériences (c'est-à-dire une forme de $n \times 8 \times m$ et de 4096×4096 à $64 \times 64 \times 64 \times 64 \times 64$ et à un tableau 9-D de longueur 8 pour toutes les dimensions) entraînent des pertes plus grandes, tel que démontré par le tableau 4.1, quoique suffisamment faible dans le présent contexte. Donc, le temps de transfert d'un élément est toujours le paramètre dominant (il prend maintenant deux cycles d'horloge). La perte pour la troisième expérience reste négligeable (elle est du même ordre de grandeur).

Trois conclusions se dégagent de ces résultats:

1. pour un processeur super-scalaire, le goulot d'étranglement est le temps d'accès à la mémoire,

2. cet algorithme demande un effort de calcul important ($\sim$ 2 instructions par cycle pour un processeur super-scalaire qui peut effectuer une opération mémoire-registre en parallèle avec une addition entière suivi d'un saut) et

NOTE TO USERS

Page(s) not included in the original manuscript are unavailable from the author or university. The manuscript was microfilmed as received.

pgs 32-33

UMI

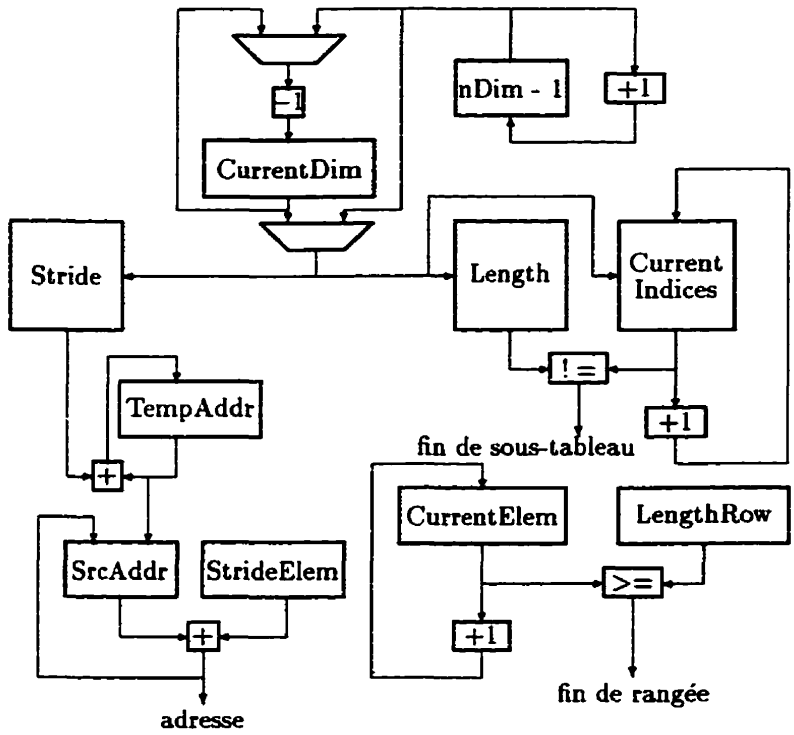


Figure 4.7: Diagramme-bloc du générateur d'adresse

dimension courante (c'est-à-dire une partie du "while" intérieur). Il est à noter que, à cause du parallélisme inhérent du générateur d'adresses, la valeur de l'élément de **Stride** pour la deuxième dimension de poids faible ne doit pas tenir compte du pas de la dimension de poids faible, puisque les incrémentations de cette dernière sont mémorisées dans **SrcAddr** plutôt que **TempAddr**. Donc, les valeurs contenues dans **Stride** sont la différence entre les adresses du premier tableau de $n-1$ dimensions du nouveau sous-tableau de n dimensions et de l'adresses du sous-tableau qui suivrait le dernier sous-tableau de $n-1$ dimensions du sous-tableau de n dimensions précédent sauf pour la deuxième dimension de poids faible pour qui le pas est la différence entre deux éléments qui sont voisins selon cette dimension.

Le parallélisme de cette implantation permet à ce module de générer une adresse par cycle d'horloge à la condition que la longueur des rangées du tableau soit supérieure au nombre de dimensions du même tableau. En pratique, cette condition devrait généralement être respectée puisqu'il est rare d'avoir des rangées plus

courtes que 8 éléments (qui est le nombre maximum de dimensions de cette implantation).

Par opposition à ce niveau de performance, une unité DMA classique devrait être reprogrammée pour chaque rangée de tableau plutôt qu'une fois pour tout le tableau. Ceci réduirait significativement le niveau de performance à cause des communications nécessaires entre le processeur et l'unité DMA et parce que cela imposerait un effort de calcul substantiel au processeur. Par contre, si les rangées sont longues et si le tableau a un faible nombre de dimensions, alors l'effort de calcul requis peut être assez faible mais ceci demande que le processeur ait beaucoup de mémoire locale (e.g. 16Koctets pour chaque tableau si les tableaux ont des rangées de 1024 éléments double-précision) ou que le tableau soit traité séquentiellement (i.e. chaque élément de tableau-résultat est calculé à partir de quelques éléments d'une même rangée). Cependant, les situations usuelles n'ont pas ces caractéristiques.

4.6 Transformations

Dans cette section, différentes transformations sur des tableaux couramment utilisées sont décrites. Étant donné que ces transformations créent une transformation linéaire entre un ensemble de vecteurs d'indices et une séquence d'adresses, elles peuvent être composées, c'est-à-dire qu'après avoir appliqué une transformation à un tableau, une autre transformation peut lui être appliquée et on peut calculer les paramètres décrivant la transformation composée (voir [8]).

Il est à noter que, dû au parallélisme des opérations du générateur d'adresses (tel qu'expliqué précédemment), les distances pour un tableau non-transformé sont: 1 pour la dimension de poids faible, la longueur des rangées pour la deuxième dimension de poids faible et de 0 pour les autres dimensions. En effet, comme les éléments et sous-tableaux sont placés séquentiellement en mémoire, la distance entre deux rangées est identique à leur longueur et l'adresse du sous-tableau qui suivrait le dernier sous-tableau (selon une dimension donnée) est la même que celle du premier sous-tableau lorsqu'on passe au sous-tableau de dimensionalité supérieure suivant.

La section 4.6.1 décrit des transformations supportées par le Fortran 90 alors que la section 4.6.2 décrit des transformations autres.

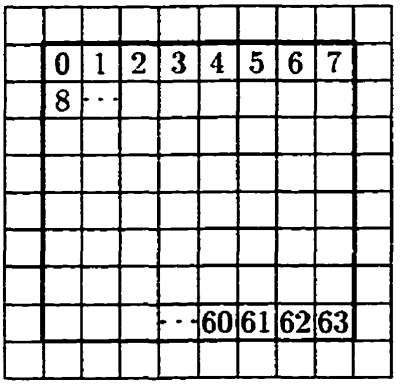


Figure 4.8: Section (2:9,2:9) d'un tableau de forme (10,10)

4.6.1 Transformations du Fortran 90

De toutes les transformations sur des tableaux supportées par le Fortran 90, la seule qui ne soit pas supportée par le générateur d'adresses est la fonction intrinsèque **cshift** parce qu'il s'agit d'une transformation qui est linéaire par morceaux seulement. L'implantation des autres transformations est donnée ci-après. Il est à noter que la fonction intrinsèque **eoshift** n'est pas décrite mais qu'elle est sémantiquement identique à une section et un appel à la fonction intrinsèque **spread** et que ces dernières sont décrites.

Section

Une section en Fortran 90 est une portion de tableau en forme de parallélépipède parallèle aux axes du tableau. Elle est décrite par une borne inférieure, une borne supérieure et un pas, et ce, pour chaque dimension du tableau.

La figure 4.8 contient un exemple de section. Les nombres présents dans les emplacements des éléments du tableau sont leur position dans la séquence d'adresses. Les distances pour ce tableau transformé sont 10 pour la dimension de poids fort et 1 pour la dimension de poids faible parce que, comme il a été expliqué précédemment, le pas de la dimension de poids faible est la distance en mémoire entre deux éléments consécutifs d'une rangée alors que le pas de la deuxième dimension de poids faible est la distance entre deux éléments situés au même endroit dans deux rangées successives.

0	10								
1	⋮								
2									
3									
4									
5									⋮
6									96
7									97
8									98
9									99

Figure 4.9: Tableau de forme (10,10) transposé

Transposition

La transformation qui sera décrite est la transposition généralisée c'est-à-dire une permutation des axes d'un tableau (la fonction intrinsèque Fortran 90 **transpose** est une transposition particulière). Cette transformation est décrite par un vecteur de permutation c'est-à-dire un vecteur contenant les valeurs de 1 au nombre de dimensions du tableau dans l'ordre désiré.

La figure 4.9 contient un exemple de tableau transposé. Les distances pour cet exemple sont 1 et 10.

Spread

Il s'agit de la fonction intrinsèque Fortran 90 **spread**. Cette transformation peut aussi être utilisée implicitement dans la fonction **matmul**. Elle crée une nouvelle dimension en répétant une dimension du tableau. L'information qui décrit cette transformation est la dimension qui est dupliquée et le nombre de fois où elle l'est.

La figure 4.10 montre un exemple de cette transformation. La forme du tableau transformé est (10,3,10) et les distances sont 10, 0, et 1.

Reshape

Étant donné que la mémoire d'un ordinateur est adressée comme un vecteur et que la fonction intrinsèque Fortran 90 **reshape** transforme un vecteur en un tableau ayant un nombre de dimensions arbitraire, cette fonction est l'équivalent de la création d'un

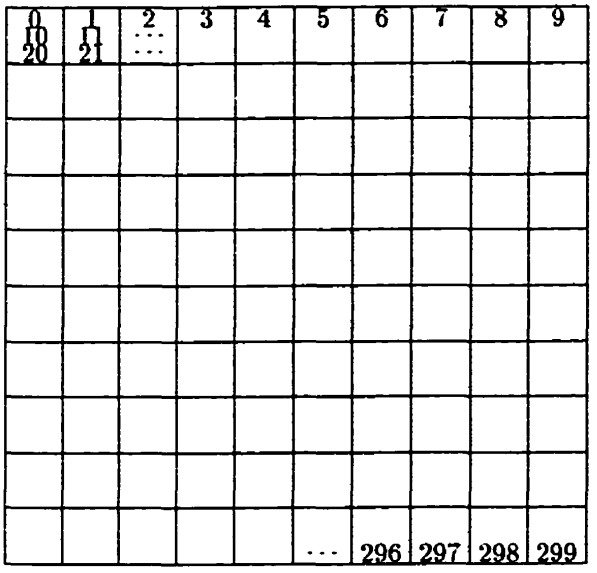


Figure 4.10: Un tableau de forme (10,10) dupliqué trois fois selon la dimension 2

tableau à partir d'une zone de mémoire. Donc, il s'agit d'une fonction très générale et qui est équivalente à ce qui est décrit à la section 4.7.5 donc on ne discutera pas de cette fonction séparément.

4.6.2 Autres transformations

Partition

Cette transformation est utilisée lorsqu'on veut diviser une dimension en plusieurs parties d'égale longueur et qu'on veut créer une nouvelle dimension pour pouvoir accéder les différents parties en séquence. Ceci peut être utile lorsqu'on veut effectuer un traitement par blocs (pour des exemples de calculs par blocs, voir [18, Section 5.4]). L'information nécessaire pour décrire la transformation est la dimension à partitionner et en combien de parties elle sera découpée.

Il est également possible de diviser le tableau en créant des régions de recouvrement. Ceci est utile lorsque le traitement effectué fait en sorte que des éléments du tableau interagissent avec des éléments qui sont situés dans différentes parties. L'information additionnelle nécessaire est la longueur du recouvrement.

La figure 4.11 donne un exemple de partition. Dans cet exemple, la forme du

0	1	2	3	4	50	51	···		
5	6	7	8	9					
10	···								
		···	48	49			···	98	99

Figure 4.11: Un tableau de (10,10) partitionné selon la dimension 2 en 2 parties

tableau transformé est (5,10,2) et les distances sont -95, 10 et 1.

"Warping"

Le type de "warping" supporté consiste en un partitionnement selon une dimension du tableau et un décalage proportionnel à la position des parties selon une autre dimension. L'information nécessaire est la dimension partionnée, la direction du décalage (le numéro de la dimension) et la magnitude du décalage exprimée selon le nombre de positions de décalage à la fin du tableau.

La figure 4.12 donne un exemple de cette transformation. Il est à noter qu'il est nécessaire d'extraire une section du tableau avant d'effectuer le "warping" parce que le générateur d'adresses ne peut pas effectuer de bouclage ("wrap-around") donc la séquence d'adresses contiendrait des valeurs illégales. Ce tableau transformé a une forme de (2,3,6) et les distances sont -26, 12 et 1.

Renversement

Renverser un tableau signifie l'accéder en traversant un de ses axes en partant de la fin. En Fortran 90, on décrit cette transformation à l'aide d'une section dont le pas est négatif mais, comme cette transformation est implantée sous forme d'un opérateur dans certains langages (par exemple, l'APL), elle est décrite séparément. La seule information nécessaire pour décrire cette transformation est la dimension renversée.

		0	1						
		6	7	2	3				
				8	9	4	5		
						10	···		
		···	31						
				32	33				
						34	35		

Figure 4.12: "Warping" de la section (3:8,3:8) d'un tableau de forme (10,10) selon la dimension 2 et en décalant de 2 selon la dimension 1

9	8	7	6	5	4	3	2	1	0
							···	11	10
99	98	97	96	···					

Figure 4.13: Reversement d'un tableau de forme (10,10) selon la dimension 2

La figure 4.13 donne un exemple de cette transformation. Les distances pour ce cas sont 10 et -1.

Damier

Dans le but de montrer la flexibilité du générateur d'adresses, on montre qu'il peut générer la séquence d'adresses nécessaire au parcours des cases d'une même couleur sur un damier. Une des nombreuses manières par laquelle on peut décrire cette transformation consiste à indiquer en combien de zones doit être diviser le tableau selon chaque dimension (c'est-à-dire qu'on ne se limite pas à des damiers "classiques"

de 8 cases sur 8 mais qu'on peut spécifier un nombre de cases arbitraire — à condition qu'il soit un diviseur de la longueur du tableau selon la dimension pertinente).

La figure 4.14 donne un exemple de damier. La forme du tableau transformé est (2,4,4,2,2) et les distances sont -4, 50, -60, 16 et 1.

Le pas pour la dimension 5 est de 1 parce que c'est la distance entre, par exemple, les éléments étiquetés 0 et 1. Le pas de la dimension 4 est la distance entre deux éléments successifs d'une colonne (par exemple, 0 et 2) ce qui est la même chose que la longueur d'une rangée (c'est-à-dire 16).

Les pas de la dimension 3 est la distance entre la rangée qui, par exemple, suit la rangée qui débute par l'élément nommé 6 et celle qui débute avec l'élément 8, c'est-à-dire $-4 \times 16 + 4$.

Le pas de la dimension 2 est la distance entre le premier élément du sous-tableau 2D qui suivrait celui qui commence par l'élément 24 et celui qui débute par l'élément 32. Mais, comme les rangées se suivent en mémoire, le "17e" élément de la première rangée est l'élément 2, donc la distance est $3 \times 16 + 2$ c'est-à-dire 50.

Finalement, le pas de la dimension 1 est la distance entre le troisième sous-tableau 3D, s'il y en avait un (qui débute, par hasard, par l'élément 72), et l'élément 64 (soit -4).

4.7 Paramètres

Avant de calculer les pas nécessaires au travail du générateur d'adresses, il faut calculer la distance (en mémoire) entre des éléments adjacents du tableau transformé selon chacune de ses dimensions, ainsi que sa forme et son adresse de départ. Ensuite, on peut calculer les pas à partir de ces données. Les pas sont différents des distances parce que ces premiers tiennent compte du fait qu'une partie de la distance a été parcourue lors de l'accès aux sous-tableaux de plus faible dimensionalité (voir page 34). Les pas et distances sont assemblés en deux vecteurs où l'élément i est l'élément qui concerne la dimension i.

Il est à noter que, lorsqu'une nouvelle dimension est créée, les anciennes dimensions sont décalées d'une position si elles ont un poids plus faible que la nouvelle dimension. Également à noter, la dimension 1 est celle de poids fort dans les équations et l'élément 1 des vecteurs de distance (d), de forme (S_h) et de pas (S_t) correspondent

0	1			8	9			16	17			24	25		
2	3			10	11			18	19			26	27		
4	5			12	13			20	21			28	29		
6	7			14	15			22	23			30	31		
		32	33			40	41			48	49			56	57
		34	35			42	43			50	51			58	59
		36	37			44	45			52	53			60	61
		38	39			46	47			54	55			62	63
64	65			72	73			80	81			88	89		
66	67			74	75			82	83			90	91		
68	69			76	77			84	85			92	93		
70	71			78	79			86	87			94	95		
		96	97			104	105			112	113			120	121
		98	99			106	107			114	115			122	123
		100	101			108	109			116	117			124	125
		102	103			110	111			118	119			126	127

Figure 4.14: Damier de 4 cases sur 8 fait à partir d'un tableau de forme (16,16)

à la dimension de poids fort. Le caractère "$'$", dans les équations, indique le nouveau contenu d'un vecteur (par opposition à celui d'avant la transformation).

4.7.1 Distance

Les nouvelles distances sont identiques aux anciennes sauf:

transposition: $d'[i] = d[T_v[i]]\ \forall 1 < i \leq N_d$ où T_v est le vecteur de permutation et N_d est le nombre de dimensions du tableau,

partition: $d'[1] = d[D_p] * S_h[D_p]/p$ où p est le nombre de partitions et D_p est la dimension partitionnée,

partition avec recouvrement: $d'[1] = d[D_p] * (S_h[D_p] - O_v)/p$ où O_v est la valeur du recouvrement,

"warping": $d'[D_w] = (d[D_w] * S_h[D_w])/(|N_p| + 1) + N_p/|N_p| * d[D_d]$ où D_w est la dimension partitionnée, D_d est la dimension selon laquelle est effectué le décalage et N_p est l'amplitude du décalage à l'extrémité du tableau,

spread: $d'[D_s + 1] = 0$ où D_s est la dimension dupliquée,

damier: $d'[5] = d[2]$, $d'[4] = d[1]$, $d'[3] = d[2] * 2 * S_h[2]/N_z[2]$, $d'[2] = d[1] * S_h[1]/N_z[1] + d[2] * S_h[2]/N_z[2]$, $d'[1] = d[1] * 2 * S_h[1]/N_z[1]$ où N_z est le nombre de zones selon chaque dimension,

renversement: $d'[D_r] = -d[D_r]$ où D_r est la dimension renversée.

4.7.2 Forme

La forme du tableau reste inchangée sauf pour:

section: $S'_h[i] = u[i] - l[i] + 1 \ \forall 1 < i \leq N_d$ où l et u sont les bornes inférieures et supérieures respectivement de la section,

transpose: $S'_h[i] = S_h[T_v[i]] \ \forall 1 < i \leq N_d$,

partition: $S'_h[1] = p$, $S'_h[D_p] = S_h[D_p]/p$,

partition avec recouvrement: $S'_h[1] = p$, $S'_h[D_p] = (S_h[D_p] - O_v)/p + O_v$

"warping": $S'_h[D_w] = |N_p| + 1$, $S_h[D_w + 1] = S_h[D_w]/(N_p + 1)$,

spread: $S'_h[D_s] = N_r$, où N_r est le nombre de répétitions de la dimension,

damier: $S'_h[1] = N_z[1]/2$, $S'_h[2] = 2$, $S'_h[3] = N_z[2]/2$, $S'_h[4] = S_h[1]/N_z[1]$, $S'_h[5] = S_h[2]/N_z[2]$.

4.7.3 Adresse de départ

L'adresse de départ n'est pas modifiée sauf pour:

section: $S'_a = \sum_{i=1}^{N_d} l[i] * d[i]$,

renversement: $S'_a = S_a + d[D_r] * (S_h[D_r] - 1)$.

4.7.4 Pas

Les pas sont calculés à l'aide de l'équation 1 sauf pour $S_t[1] = d[1]$ et $S_t[2] = d[2]$. Il est à noter que, pour calculer $S_t[i]$, on a besoin de la valeur des $S_t[j]$ tels que $i < j < N_d$ donc les pas doivent être calculés en commençant par celui de la dimension de poids faible et en allant vers la dimension de poids fort. Également, il est évident que le calcul du produit ($\prod$)de chacun des pas peut utiliser celui du pas de poids faible précédent en guise de résultat partiel.

$$S_t[i] = d[i] - \sum_{j=i+1}^{N_d-1} S_t[j] * \prod_{k=i+1}^{j} S_h[k] \qquad \forall 3 < i \leq N_d \tag{1}$$

4.7.5 Généralisation

La manière la plus générale d'exprimer l'adressage d'un tableau est de spécifier:

- l'adresse de départ,
- le nombre de dimensions du tableau,
- la forme du tableau (i.e. un vecteur) et
- un vecteur de distances.

Utiliser de tels paramètres permet des transformation arbitraires (au sens où elles n'ont pas de signification particulière). La figure 4.15 montre une de ces transformations. Il est à noter que, pour simplifier la représentation, le tableau est montré comme une matrice de forme (10,8) qui aurait été modifiée mais cette forme n'est pas pertinente du point de vue de l'algorithme.

Cet exemple crée un tableau de trois dimensions de forme 4 par 4 par 4 en utilisant un vecteur de distances égal à -9 pour la dimension de poids fort, 10 pour la dimension intermédiaire et 1 pour la dimension de poids faible. Il est à remarquer que les quatres sous-tableaux à deux dimensions qui composent ce tableau sont identifiés par les nombres 0, 1, ..., 15 pour le premier plan, 16, 17, ..., 31 pour le deuxième plan, 32, 33, ..., 47 pour le troisième plan et 48, 49, ..., 63 pour le dernier plan. Les trois flèches de la figure 4.15 représentent les directions des trois dimensions du tableau transformé (i.e. là où mènent les éléments du vecteur de distances).

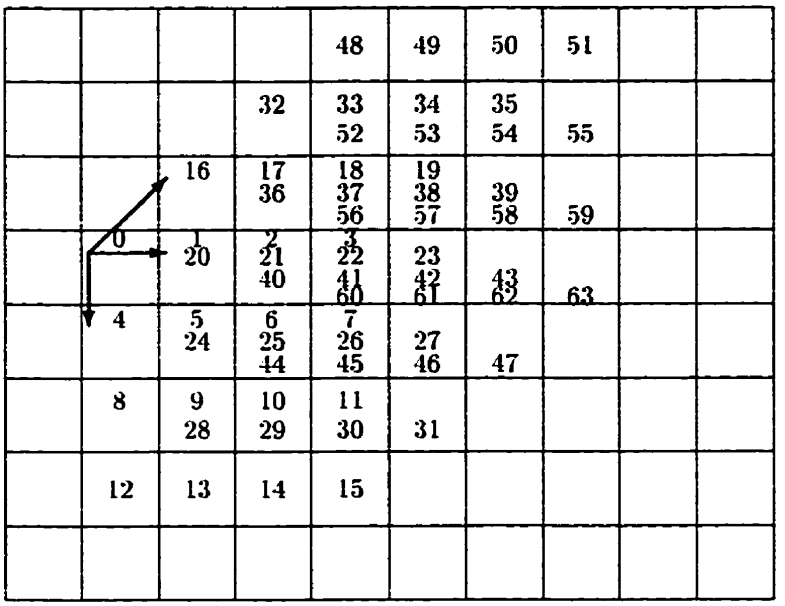


Figure 4.15: Transformation arbitraire

4.8 Conclusions

Dans ce chapitre, on a décrit un algorithme qui permet de transférer efficacement un tableau transformé entre la mémoire d'un ordinateur et son processeur. On a montré qu'il permet de transférer un élément de tableau par cycle de mémoire s'il est implanté en logiciel sur un processeur super-scalaire et qu'il permet un transfert par cycle d'horloge s'il est implanté en matériel. Cet algorithme supporte toutes les transformations linéaires entre un ensemble de vecteurs d'indices et une séquence d'adresses en mémoire ce qui le rend très flexible. Également, il peut être facilement augmenté pour supporter les transformations polynomiales de degré plus élevé puisque les paramètres quadratiques impliquent une modification des pas (paramètres linéaires) et de même pour les paramètres d'ordre supérieur. Donc, en ajoutant des fichiers de registres pour contenir les nouveaux paramètres et en utilisant un algorithme similaire à celui utilisé par le générateur d'adresses qui a été décrit, on pourrait supporter les transformations polynomiales avec une vitesse de calcul essentiellement aussi grande mais qui nécessiterait substantiellement plus de ressources matérielles.

Le générateur d'adresses proposé est très rapide (normalement, une adresse par cycle d'horloge) et flexible et calculer les paramètres qui lui sont nécessaires est simple à cause du cadre conceptuel (i.e. transformations linéaires).

Chapitre 5

Un langage de haut niveau pour les ordinateurs SIMD

À la section 2.4, on a constaté qu'aucun des langages de programmation existants ne rencontre tous les objectifs suivants:

1. utiliser les tableaux (et les opérations sur ceux-ci) en guise d'abstraction de haut niveau ainsi que de paradigme de parallélisme (i.e. parallélisme sur les données),
2. permettre la compilation et la parallélisation automatique d'applications en un code exécutable de haute performance,
3. cibler les architectures SIMD et
4. évaluer quelles limitations peuvent être imposées sur la grammaire d'un langage pour faciliter la parallélisation sans contraindre indûment la programmation.

Pour combler ce vide, le langage HPCP ("High Performance C for **Pulse**") est proposé. Ce langage contient les éléments du langage C [30] qui sont appropriés pour la description (parallèle) d'applications traitant de façon structurée des tableaux, et ce, sur un processeur SIMD et il est étendu là où le C est déficient.

Dans le but de produire des programmes exécutables performants et compacts, le compilateur HPCP qui a été créé dans le cadre du présent travail utilise des tampons circulaires (pour stocker localement des éléments de tableaux) et des instructions vectorielles (pour obtenir du code rapide et compact).

La section 5.1 contient une description du langage, alors que la section 5.2 décrit la sémantique de certaines de ses composantes. La section 5.3 explique pourquoi une mémoire est, dans le contexte présent, plus appropriée que les registres vectoriels pour contenir des vecteurs; la méthode de gestion des tampons circulaires qui est utilisée par le compilateur est également décrite. La section 5.4 donne un exemple de programme source et de programme généré, alors que la section 5.5 contient une étude du niveau de performance (en temps et en espace-mémoire) du code généré. Finalement, la section 5.6 énonce les conclusions de ce chapitre.

5.1 Description du langage

Cette section est divisée en deux parties: la première décrit la portion du langage C qui a été retenue pour HPCP, alors que la deuxième décrit les extensions qui lui ont été ajoutées. La grammaire complète du langage est donnée à l'annexe C.

5.1.1 Sous-ensemble du C supporté

Les seuls types de données supportés sont "int" et "long", parce que les applications DSP traitent habituellement des nombres entiers et parce que l'architecture ciblée (**Pulse**) ne supporte que ces types de données. Les types de données définis par l'usager ne sont pas supportés; ce sont "struct", "union", "enum", "fields" ainsi que la directive **typedef**. Le qualificatif "register" n'est pas supporté non plus, parce qu'il n'est pertinent qu'à un niveau d'abstraction plus bas (i.e. pour des langages comme AL [29]). Les constantes symboliques sont supportées via le qualificatif "const".

Les structures de contrôle **séquentielles** (i.e. les boucles "while", "do" et "for") ne sont pas permises à cause de l'objectif visant à décrire les algorithmes de façon **parallèle** (d'un autre côté, un énoncé "loop" a été ajouté — voir la section 5.1.2). Également, l'énoncé "switch" n'est pas supporté parce qu'il est trop général pour les besoins spécialisés d'un ordinateur SIMD. De plus, les énoncés "goto", "break" et "continue" ne sont pas supportés. Donc, le seul énoncé de contrôle qui soit supporté est le "if".

Tous les opérateurs (arithmétiques, logiques et relationnels) sont supportés sauf "++", "−−" (parce qu'ils ont un effet secondaire), l'opérateur conditionnel (parce qu'il est sémantiquement identique au "if" bien qu'ils aient un contexte syntaxique

différent), l'opérateur "," (parce qu'il est surtout utilisé dans des énoncés qui ne sont pas supportés) et la division et l'opérateur "modulo" (parce qu'ils ne sont pas supportés par **Pulse**). Les assignations ne sont **pas** permises dans les expressions conditionnelles.

Les pointeurs ne sont pas supportés parce qu'ils ne sont pas utiles dans le contexte de **Pulse** et parce qu'ils rendent la parallélisation plus difficile à cause de l'équivalence qu'ils peuvent entraîner. Les chaînes de caractères ne sont pas supportées parce qu'elles ne seraient pas utiles (encore une fois, dans le contexte de **Pulse**). Par contre, les tableaux multi-dimensionnels sont supportés évidemment et ils ont une structure où la dimension de gauche est celle de poids fort (comme en C).

5.1.2 Extensions ajoutées au C

Les ajouts au langage C sont regroupés en quatres catégories:

1. support pour les instructions des processeurs élémentaires (PE) de **Pulse** qui ne sont pas supportées par le C,
2. des structures permettant une description plus compacte de traitement de tableaux (pour permettre une description à haut niveau des algorithmes),
3. un nouvel énoncé de contrôle et
4. deux directives ("pragmas").

Les instructions des PE qui ne sont pas supportées par le C sont implantées sous forme de fonctions intrinsèques (ces instructions incluent, par exemple, "compare-and-swap", "clip" et "median"). Les structures de support pour les tableaux sont tirées du Fortran 90, parce que c'est le langage qui, parmi ceux qui ont les caractéristiques désirées, est le plus proche du langage C. Les structures en question sont de trois types: les sections de tableaux, les opérateurs sur les tableaux et une structure de contrôle parallèle. Une section de tableau est un morceau de tableau décrit par une borne inférieure et une borne supérieure pour chaque dimension du tableau. Par exemple, si le tableau **A** a une forme de [10][10][10] alors **A**[1:5][2:6][3:9] est un sous-tableau de forme [5][5][7] qui débute à l'élément **A**[1][2][3], i.e. le deuxième élément de la première dimension, le troisième de la seconde et le quatrième de la dernière

dimension. Il est à noter que, contrairement au Fortran 90, le pas n'est pas supporté parce qu'il fallait limiter le langage le plus possible à cause du manque de ressources et que, sachant que les algorithmes DSP ne l'utiliserait pas souvent, on ne prévoit que peu d'intérêt à avoir cette caractéristiques. Les opérateurs sur les tableaux sont les mêmes que sur les scalaires et leur sémantique est expliquée à la section 5.2.

La structure de contrôle parallèle supportée est le "where"; il s'agit d'un énoncé de contrôle (plus spécifiquement, de sélection) parallèle de haut niveau d'abstraction (voir la section 5.2 pour une description de sa sémantique). Un énoncé "forall" (comme en HPF) serait probablement utile, mais il n'est pas supporté par les algorithmes de parallélisation du chapitre 6 (parce que ce chapitre traite de la parallélisation de traitement **structuré** de tableaux alors que le "forall" permet le traitement non-structuré) donc, il n'a pas été ajouté.

Les deux directives supportées sont: "distribute" et "configuration". La directive "distribute" permet au programmeur d'indiquer comment les tableaux doivent être distribués entre les PE, alors que "configuration" décrit la configuration matérielle nécessaire à l'exécution du programme.

La forme de la directive "distribute" est: le mot-clé "#pragma" suivi du mot-clé "distribute" et, pour chaque tableau utilisé dans l'énoncé, son nom suivi, pour chacune de ses dimensions, de la description de la distribution (qui peut être "block", "cyclic(n), ou "*") entre crochets (le "(n)" associé à la distribution cyclique est optionnel). Les descriptions de distribution des différents tableaux à l'intérieur d'une même directive sont séparés par une virgule.

Finalement, le nouvel énoncé de contrôle est le "loop". Il est sémantiquement identique à l'énoncé C "while(1)"; c'est-à-dire qu'il s'agit d'une boucle sans fin qui ne peut être interrompue que par une interruption du SIMD. Ceci est utile lorsqu'on traite des flots de données.

5.2 Sémantique

Cette section décrit la sémantique de certains éléments de HPCP qui peuvent être plus obscurs à quelqu'un qui est familier avec le C.

5.2.1 Structures de support pour les tableaux

Opérateurs

Tous les opérateurs ont la même sémantique qu'en langage C, sauf lorsqu'ils sont utilisés avec des tableaux ou des sections auquel cas, elle est étendue de la façon suivante: l'opérateur scalaire est appliqué à chaque paire d'éléments de tableaux (un de chaque tableau/section) et chacun de ces éléments provient de la même position dans chaque tableau/section. Donc, les tableaux/sections doivent avoir la même forme. Si une condition utilise un ou des opérateurs logiques sur des tableaux, alors la sémantique de la condition devient: la condition est vraie si le tableau de valeurs booléennes généré ne contient que des valeurs vraies, sinon, elle est fausse. Il pourrait être utile d'obtenir une valeur vraie lorsqu'au moins un élément du tableau généré; ce cas peut être traité en utilisant, dans l'expression de la condition, les opérateurs de comparaison complémentaires à ceux désirés et en inversant le résultat de ces comparaisons.

Les opérateurs d'assignation voient leur sémantique étendue de la même façon, sauf que les assignations (scalaires) sont effectuées, conceptuellement, en même temps. Par exemple,

```
A[1:10][1:10] = A[0:9][0:9] + B + C
```

a la même sémantique que le programme C suivant:

```
for(i = 1; i < 10; i++)
   for(j = 1; j < 10; j++)
      temp[i][j] = A[i-1][j-1] + B[i-1][j-1] + C[i-1][j-1];

for(i = 1; i < 10; i++)
   for(j = 1; j < 10; j++)
      A[i][j] = temp[i][j];
```

Ce qui signifie que les expressions sur des tableaux n'introduisent aucune dépendance entre les éléments des tableaux de la partie droite de l'assignation et ceux du tableau de la partie gauche.

L'énoncé "where"

Les énoncés contrôlés par un "where" doivent avoir la même forme que la condition. Un élément d'une expression à la droite d'une assignation (ou un élément d'une condition) de la partie "where" de l'énoncé "where" est calculé seulement si l'élément correspondant de la condition du "where" est vrai; ceci est également le cas pour l'assignation à un élément de l'expression du côté gauche d'une assignation. Les calculs et assignations situés dans la partie "else where" sont effectués lorsque l'élément correspondant de la condition est faux.

Fonction intrinsèques d'entrée/sortie

Les fonctions intrinsèques **read**, **write**, **input** et **output** sont utilisées pour lire ou écrire une variable (scalaire ou tableau) en mémoire externe et pour recevoir ou envoyer un tableau au monde extérieur respectivement.

5.2.2 Distribution

Le modèle de distribution est adapté de celui du HPF [24] de la façon suivante: tous les tableaux d'une expression doivent avoir la même distribution. Dans le cas d'une distribution par blocs, les éléments de tableaux qui sont utilisés par plus d'un PE sont répliqués. La distribution cyclique est utilisée pour réduire la pression sur les mémoires internes. Cela signifie qu'il ne s'agit pas d'une distribution entre les PE mais plutôt d'une distribution dans le temps, c'est-à-dire qu'une portion seulement de la dimension du tableau sera traitée à la fois. Le résultat est une forme de traitement par blocs (pour des exemples de traitement par blocs, voir [18]). Il a été décidé d'adapter le modèle de distribution du HPF parce que ce modèle est orienté vers le traitement structuré de tableaux et que la plupart des algorithmes DSP sont de type structuré.

Étant donné que la mémoire disponible sur le même circuit intégré que le SIMD est normalement très limitée et que les tableaux sont gérés sous forme de tampons circulaires (tel qu'expliqué à la section 5.3), une distribution par blocs implique que les tableaux ne sont pas accédés en ordre lexicographique mais, plutôt, que chaque PE reçoit un élément à la fois et que tous les PE en reçoivent un en même temps (donc, un générateur d'adresses comme celui décrit au chapitre 4 est nécessaire). Un

effet important de cette stratégie est que les tableaux doivent être stockés localement par opposition à être reçus directement du monde extérieur. Cette contrainte, qui découle simplement d'un manque de temps pour l'implantation du prototype de compulateur, pourra être levée dans l'avenir.

Il est à noter que, dans le présent modèle de partitionnement, tous les tableaux ont le même alignement (i.e. l'alignement est de 0 entre eux) donc, il n'est pas décrit dans le programme-source (contrairement au modèle HPF).

5.3 Tampons circulaires

Étant donné que les instructions vectorielles permettent une plus grande densité de code (i.e. moins d'espace-mémoire nécessaire pour exprimer le même algorithme), elles sont supportées par l'architecture **Pulse**. Cependant, les processeurs vectoriels utilisent en général des registres vectoriels qui ne seraient pas efficaces dans le contexte du projet **Pulse**. Dans la présente section, il est démontré que les registres vectoriels sont inefficaces et qu'une mémoire locale gérée correctement est plus appropriée pour les applications basées sur des convolutions. La technique de gestion de la mémoire proposée est basée sur le concept de tampon circulaire; on montre comment l'adapter pour permettre l'utilisation efficace d'instructions vectorielles.

5.3.1 Bande passante requise par les registres vectoriels

L'utilisation de registres vectoriels nécessite parfois plus de bande passante à la mémoire principale parce que la structure des processeurs qui les utilisent fait en sorte que:

1. on ne peut les accéder qu'à partir de leur premier élément,
2. on doit les recharger en entier à chaque fois qu'au moins un
3. nouvel élément de donnée est requis et
4. lorsqu'on calcule une convolution, il arrive souvent que deux registres doivent contenir les même éléments mis à part quelques-uns aux extrémités des registres.

Ce plus grand besoin de bande passante est illustré par les résultats du programme d'évaluation de performance STREAM [39, 49] qui montrent que, par exemple, les ordinateurs vectoriels CRAY et NEC SX ont une valeur d'équilibre ("balance" — le rapport entre le nombre maximum d'opérations que le processeur peut effectuer en une seconde et la bande passante utilisable de la mémoire exprimée en nombre de mots par secondes) d'environ 1 alors que la plupart des microprocesseurs ont une valeur d'équilibre d'environ 10 (mis à part la famille d'ordinateurs IBM RS6000 qui ont une valeur d'environ 3). Évidemment, le fait que différents marchés sont visés par ces ordinateurs est une raison significative pour expliquer cette différence mais une valeur aussi faible que 1 ne serait pas utile si les données étaient réutilisées selon les besoins (e.g. éviter de recharger un registre vectoriel complet lorsqu'on n'a besoin que d'un seul nombre) car la bande passante disponible ne serait pas utilisée à pleine capacité. Ce plus grand besoin de bande passante est également illustré par le fait que l'architecture Torrent [5, 4] (qui est aussi une architecture vectorielle basée sur des registres) vise les calculs matriciels [51] où deux matrices différentes interagissent (i.e. lorsque les vecteurs sont réutilisés, ils le sont en entier) donc, l'utilisation de registres vectoriels dans ce contexte est efficace.

Dans le but de quantifier la bande passante gaspillée (pour le calcul de convolutions 2D) lorsqu'on utilise des registres vectoriels, p représente la longeur des rangées du tableau à traiter, n est le nombre de colonnes dans le noyau de convolution, m est le nombre de rangées de ce noyau et l_v est le nombre d'éléments que peut contenir un registre vectoriel. Pour obtenir le maximum de performance d'un processeur vectoriel, il est généralement conseillé d'avoir des rangées de tableaux dont la longueur est un multiple de l_v. Dans le cas de la convolution, il faut que $p - (n - 1)$ soit un multiple de l_v. Puisqu'il s'agit là du meilleur cas, du point de vue de la performance, cette hypothèse sera utilisée dans le reste de cette section.

Dans les paragraphes qui suivent, la bande passante requise pour calculer une rangée du tableau résultant d'une convolution est calculée et comparée au nombre minimum de transferts nécessaires. On fait l'hypothèse que le contenu des registres ne peut être réutilisée pour calculer plus d'une rangée du tableau-résultat (ce qui est réaliste puisque les registres doivent être rechargés pour calculer chaque élément du résultat). Pour calculer un vecteur du tableau-résultat, $r \times m \times n$ vecteurs doivent être chargés, où r est le rapport entre le nombre de coefficients non-nuls sur le nombre

de coefficients total du noyau de convolution (nm). Aussi, $\frac{p-(n-1)}{l_v}$ vecteurs doivent être calculés pour obtenir une rangée du résultat. Donc,

$$rmn(\frac{p-(n-1)}{l_v})l_v$$

chargements (d'éléments de tableaux) sont effectués alors que seulement pm éléments de tableaux sont nécessaires. Cela signifie que la surcharge relative est

$$\frac{rmn(\frac{p-n+1}{l_v})l_v - pm}{pm}$$

Après simplification, cette équation devient $\frac{rn}{p}(p-n+1)-1$ mais, lorsque $p \gg n$ (ce qui normalement le cas), cette équation tends vers $rn-1$. Lorsque r est raisonablement élevé (par exemple, $r > 0.5$), la surcharge est plus élevée que le nombre d'éléments de tableau requis dans un rapport de plusieurs fois. De plus, ceci s'aggrave très rapidement à mesure que n s'accroît (ce qui est le cas pour les applications DSP lorsque la puissance de calcul croît puisque cela permet de réaliser, par exemple, des filtres de taille plus grande) donc, cette surcharge doit être évitée à tout prix.

5.3.2 Stratégie d'allocation dans les tampons circulaires

Dans la section qui précède, on discute d'une façon de vectoriser les calculs sur des tableaux qui consiste à diviser les tableaux en vecteurs et d'effectuer les calculs sur ces derniers. Dans ce contexte, allouer séquentiellement des éléments de tampon circulaire aux éléments d'un tableau est efficace. Cependant, cette méthode nécessite l'utilisation de plus de résultats temporaires si on veut éviter de recharger les éléments des tableaux. Par contre, si la convolution utilise un noyau suffisamment grand et dense, utiliser une instruction vectorielle pour calculer chacun des éléments du tableau-résultat permet de minimiser la quantité de mémoire locale requise tout en étant aussi efficace du point de vue de la vitesse de calcul. Cette méthode de vectorisation nécessite, cependant, une nouvelle stratégie d'allocation des éléments de tampon circulaire aux éléments de tableau.

Le but de la stratégie d'allocation est de stocker les éléments d'un tableau 2D dans un tampon circulaire, de façon à ce que les éléments nécessaires au calcul d'un élément du résultat d'une convolution soient dans des positions situées à égale distance entre

eux dans le tampon circulaire et ce, dans le but de permettre de calculer chaque élément du résultat à l'aide d'une **seule** instruction vectorielle. On se limite à des tableaux 2D parce que la quantité de mémoire locale à un processeur ne permet pas, en général, de conserver suffisamment d'éléments pour éviter de les recharger dans le cas d'un tableau à plus de deux dimensions. Cependant, la section 5.3.4 esquisse une solution pour le cas où le tableau a trois dimensions.

Une manière directe d'effectuer l'allocation (et qui ne fonctionne **pas**) consiste à utiliser un tampon de longueur $p-1-m$, où p est la longueur d'une rangée de tableau et m est le nombre de colonnes du noyau de convolution et d'allouer les éléments de tampon séquentiellement. On obtient alors l'allocation suivante si un noyau de 3 par 3 est utilisé (où les valeurs présentent dans le tableau représentent la position de l'élément correspondant du tableau dans le tampon):

$$\begin{bmatrix} 0 & 1 & 2 & 3 & \cdots & p-4 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 & \cdots & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & \cdots & 5 & 6 & 7 & 8 \\ & & & & \vdots & & & & \end{bmatrix}$$

Le problème qui se pose avec cette allocation est que, par exemple, pour calculer l'élément [1][1] (selon la notation du langage C) du résultat nécessite, entre autres, les éléments [0][0], [0][1] et [0][2]. Ces éléments seraient mis aux positions 0, 1 et 2 du tampon respectivement mais, lorsque vient le temps de calculer le dit élément, ces positions du tampon ont déjà été modifiées par l'écriture des trois derniers éléments de la première rangée du tableau. Donc, un tampon pouvant contenir plus d'éléments de tableaux est nécessaire.

La longueur minimale du tampon nécessaire pour éviter d'effacer prématurément des éléments du tableau est $p(n-1)+m$ où n est le nombre de rangées dans le noyau de convolution. Avec un tampon de cette taille et un noyau de 3 par 3, l'allocation devient:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & \cdots & p-4 & p-3 & p-2 & p-1 \\ p & p+1 & p+2 & p+3 & \cdots & 2p-4 & 2p-3 & 2p-2 & 2p-1 \\ 2p & 2p+1 & 2p+2 & 0 & \cdots & 3p-4 & 3p-3 & 3p-2 & 3p-1 \\ 3p & 3p+1 & 3p+2 & 0 & \cdots & 4p-4 & 4p-3 & 4p-2 & 4p-1 \\ & & & & \vdots & & & & \end{bmatrix}$$

On constate que chacune des portions de rangée de tableau utilisées forme un vecteur dans le tampon (si on tient compte du bouclage — "wraparound"). Ceci signifie que n instructions vectorielles et $n-1$ instructions scalaires sont nécessaires pour calculer un élément du résultat. Ceci est sous-optimal puisqu'il faut redémarrer le pipeline pour chaque instruction vectorielle donc, calculer un élément du résultat avec une seule instruction vectorielle augmenterait la performance (et la densité de code). En conséquence, une nouvelle stratégie d'allocation est nécessaire. La stratégie qui vient d'être décrite sera appelée "allocation séquentielle" dans le reste de cette section.

L'idée de base de la nouvelle stratégie d'allocation consiste à allouer les éléments consécutifs d'une **colonne** du tableau (par opposition à ceux d'une rangée dans la stratégie séquentielle) à des positions consécutives du tampon et d'allouer les éléments d'une rangée à des positions dont la distance est égale au nombre de rangées du noyau (i.e. n). La longueur du tampon doit alors être de $np-1$ pour obtenir le bon bouclage à la fin du tampon. Le résultat de cette allocation pour un noyau de 3 par 3 est:

$$\begin{bmatrix}
0 & 3 & 6 & 9 & \cdots & n(p-3) & n(p-2) & n(p-1) \\
1 & 4 & 7 & 10 & \cdots & n(p-3)+1 & n(p-2)+1 & n(p-1)+1 \\
2 & 5 & 8 & 11 & \cdots & n(p-3)+2 & n(p-2)+2 & n(p-1)+2 \\
3 & 6 & 9 & 12 & \cdots & n(p-3)+3 & n(p-2)+3 & n(p-1)+3 \\
 & & & & \vdots & & & \\
np-6 & np-3 & 1 & 4 & \cdots & & & \\
np-5 & np-2 & 2 & 5 & \cdots & & & \\
np-4 & 0 & 3 & 6 & \cdots & & & \\
 & & & & \vdots & & &
\end{bmatrix}$$

Cette stratégie satisfait donc les deux objectifs: le tampon est géré correctement (i.e. il suffisamment long pour éviter l'effacement prématuré des données contrairement à la première méthode décrite) et les éléments nécessaires au calcul d'un élément du résultat sont dans des positions successives du tampon.

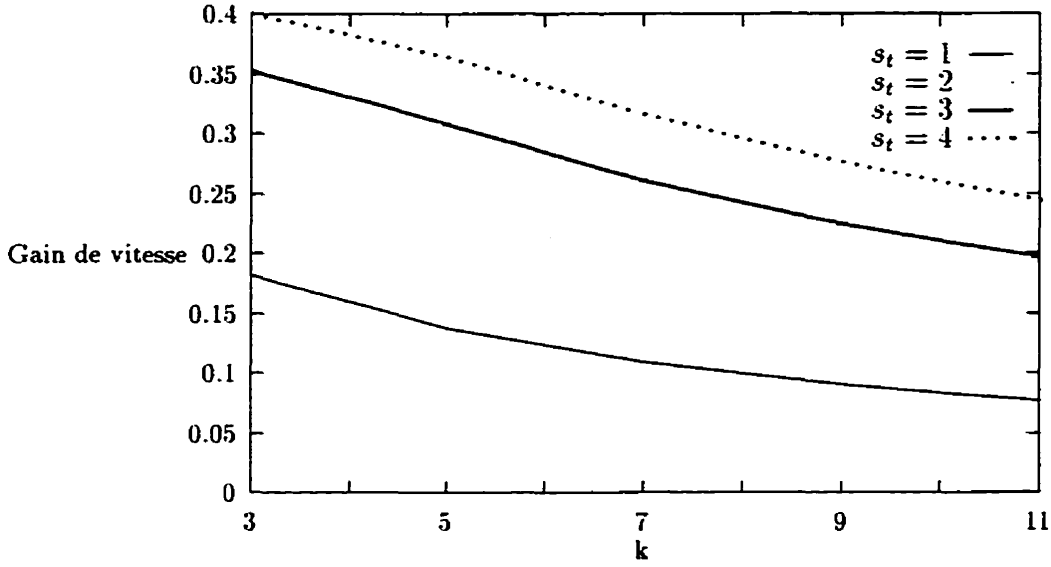


Figure 5.1: Gain de vitesse entre les vectorisations partielle et totale

5.3.3 Évaluation de la stratégie d'allocation

Gain de vitesse

Pour évaluer le gain de vitesse de la nouvelle stratégie d'allocation, le nombre de cycles nécessaire pour calculer un élément de résultat à l'aide d'un processeur vectoriel pour un noyau carré de $k \times k$ est calculé pour l'allocation séquentielle et pour la nouvelle stratégie. On suppose que le débit établi est égal à un. L'équation du temps pour l'allocation séquentielle est $t_s = k \times (s_t + (k-1)) + k - 1$ et, pour la nouvelle stratégie, elle est $t_n = s_t + k \times k - 1$ où s_t est le temps de démarrage.

La différence relative entre ces deux vitesses de calcul est montrée à la figure 5.1. Le temps de démarrage pour les instructions vectorielles, s_t, prend les valeurs 1 à 4. Ces valeurs sont très petites (i.e. elles supposent un processeur très performant) mais elles sont réalistes pour un processeur visant à supporter les applications DSP sur des nombres entiers. Augmenter ces valeurs ne ferait qu'améliorer le gain de vitesse de la nouvelle stratégie donc, ceci est le pire scénario pour la nouvelle stratégie et, malgré tout, le gain de vitesse varie entre 7% et 40%.

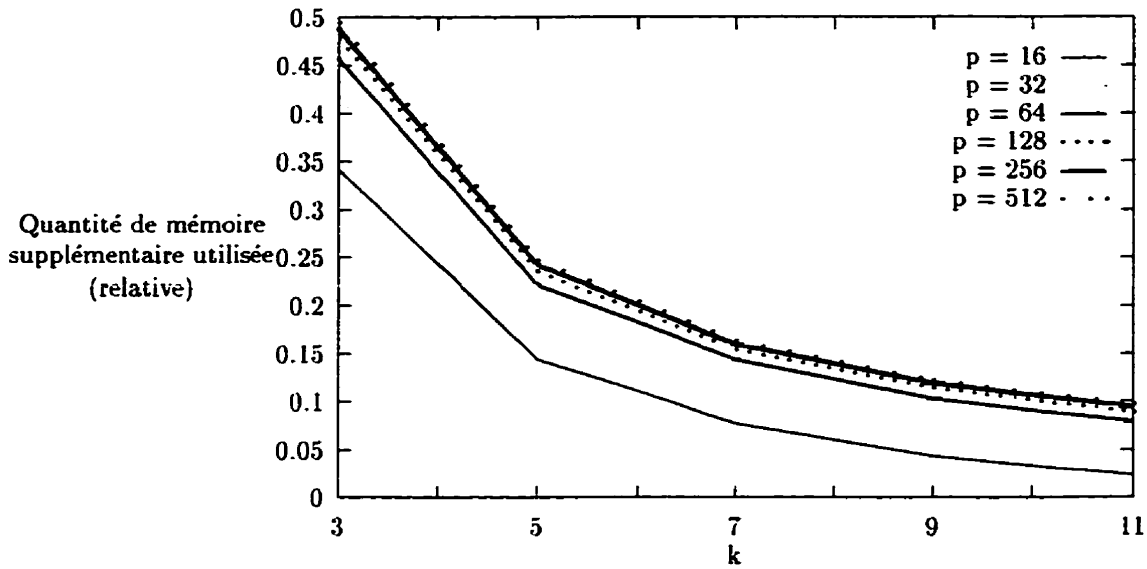


Figure 5.2: Quantité supplémentaire de mémoire requise

Quantité de mémoire utilisée

Cette nouvelle stratégie permet une grande efficacité d'utilisation de la puissance de calcul d'un processeur vectoriel (si le noyau de convolution est grand par rapport au temps de démarrage). Elle permet aussi une grande densité de code par l'utilisation efficace d'instructions vectorielles. Cependant, son désavantage est l'utilisation d'une plus grande quantité de mémoire que le minimum absolument nécessaire. Ce minimum est celui requis par l'allocation séquentielle et il est de $p(n-1)+m$ (voir la section 5.3.2). Donc, la quantité supplémentaire de mémoire requise est de $np-1-(p(n-1)+m)$, en valeur absolue, et de $\frac{np-1-(p(n-1)+m)}{p(n-1)+m}$ en valeur relative. Simplifier ces expressions donne $p-m-1$ et $\frac{p-m-1}{p(n-1)+m}$ respectivement.

La figure 5.2 montre la quantité de mémoire supplémentaire relative nécessaire pour des valeurs raisonables de p et k (où $k=n=m$, i.e. on ne montre le résultat que pour des noyaux carrés car les conclusions tiennent pour les noyaux non-carrés).

Cette figure montre que la quantité de mémoire supplémentaire requise peut être assez grande. La quantité de mémoire excédentaire est d'autant plus grande (en valeur relative) que le noyau est petit. Aussi, il est évident à l'étude des équations qu'effectuer les calculs par blocs est nécessaire lorsque le tableau a de longues rangées

(par exemple, 1024 éléments) puisque la quantité de mémoire requise devient très grande par rapport à la quantité de mémoire locale habituellement disponible à un processeur.

5.3.4 Étude du cas 3D

Utiliser cette stratégie pour les tableaux tridimensionnels peut être utile mais elle doit être modifiée pour être correcte. La raison en est que le premier élément d'un plan doit être situé à la position dans le tampon qui suit le premier élément du plan précédent donc, le premier élément d'une rangée du tableau doit être à une distance de l (le nombre de plans du noyau de convolution) du premier élément de la rangée précédente pour que les éléments nécessaires au calcul d'un élément du résultat soient à des positions successives du tampon. Selon le même raisonnement, des éléments successifs d'une rangée du tableau doivent être à une distance de ln (où n est le nombre de rangées du noyau). Ceci implique que le tampon devrait avoir une longueur de $pln - l$ (ou $l(pn - 1)$). Dans ce contexte, le bouclage à la fin du tampon ne peut revenir à la position 1 (tel que nécessaire) parce que le pas d'allocation (ln) et la longueur du tampon ($l(pn-1)$) sont tous deux des multiples de l. Ceci implique que la stratégie doit être modifiée pour qu'elle puisse fonctionner tel que désiré. La situation désirée, lorsque le noyau a une forme de 3 par 3 par 3, est la suivante:

$$\begin{bmatrix} \begin{bmatrix} 0 & 9 & 18 & 27 & \cdots & (p-2)ln & (p-1)ln \\ 3 & 12 & 21 & 30 & \cdots & (p-2)ln+l & (p-1)ln+l \\ 6 & 15 & 24 & 33 & \cdots & (p-2)ln+2l & (p-1)ln+2l \\ & & & & \vdots & & \end{bmatrix} \\ \begin{bmatrix} 1 & 10 & 19 & 28 & \cdots & (p-2)ln+1 & (p-1)ln+1 \\ 4 & 13 & 22 & 31 & \cdots & (p-2)ln+l+1 & (p-1)ln+l+1 \\ 7 & 16 & 25 & 34 & \cdots & (p-2)ln+2l+1 & (p-1)ln+2l+1 \\ & & & & \vdots & & \end{bmatrix} \\ \vdots \end{bmatrix}$$

Une façon simple d'adapter la stratégie est de remarquer que la position du premier élément de chaque plan du tableau est indépendante de la position courante si on décide de les mettre dans des positions du tampon qui soient successives. Par conséquent, une solution consiste à utiliser un deuxième compteur qui est incrémenté

de 1 à la fin de chaque plan et qui est utilisé en guise d'adresse de départ pour chaque nouveau plan.

5.4 Exemple de programme

La figure 5.3 montre un exemple de programme HPCP. Ce programme consiste en une paire de convolutions classiques. On peut y voir des appels aux fonctions **read** et **write** ainsi que l'énoncé "loop" qui a été expliqué ci-haut. Aussi, on peut remarquer comment les convolutions sont décrites: elles consistent en des sections de même forme mais situées à différents endroits à l'intérieur du tableau.

Le compilateur génère du code C-PULSE [1]. La figure 5.4 montre le code généré à partir du code-source de la figure 5.3 (certaines modifications y ont été apportées pour que l'exemple ne dépasse pas une page). On peut constater que ce code généré contient des tampons circulaires (implantés par les fonctions intrinsèques dont le nom contient "bufA" ou "bufB") plutôt que des tableaux lorsque plus d'un élément du tableau sont nécessaires en même temps, sinon, une variable scalaire (par exemple, la variable **h**) est utilisée. En particulier, dans cet exemple, le tableau **d** s'est vu attribué le tampon A et le tableau **g** utilise le tampon B. Les convolutions sont générées en calculant les valeurs des constantes utilisées (en effectuant leur distribution — au sens mathématique — lorsque nécessaire) pour permettre de décrire la convolution sous forme de somme de produits. Les valeurs ainsi calculées sont stockées dans un vecteur constant (par exemple, **_hpcp_coeff0**) alors que les calculs sont implantés sous forme d'une fonction intrinsèque (**_convolIterbufAw** et **_convolIterbufBw**). La distribution des constantes n'est **pas** effectuée pour un opérateur situé à la fin des calculs d'une expression (par exemple, >> 4 dans l'exemple) parce qu'il peut servir à faire une mise à l'échelle des résultats donc la distribution des constantes pourrait diminuer la précision des calculs si elle était effectuée.

Il est à noter comment les tampons circulaires sont initialisés (**_initbuf**) et utilisés (**_writebuf**). Également à remarquer, la fonction **_convolIterbuf** qui extrait une portion d'un tampon circulaire et effectue un produit scalaire avec un vecteur en une seule instruction vectorielle. Finalement, il est à remarquer que le tableau est lu et certains de ses éléments sont transférés entre les PE en utilisant le même canal de communication (i.e. **_North**).

```
main ()
   {int d[10][10], g[10][10], h[10][10];

#pragma distribute d[*][block], g[*][block], h[*][block];

   read(d);

   g[1:8][1:8] =  (d[0:7][0:7] + d[0:7][2:9] + d[2:9][0:7] +
                  d[2:9][2:9] + (d[2:9][1:8] + d[0:7][1:8] +
                  d[1:8][2:9] + d[1:8][0:7]) * 2 +
                  d[1:8][1:8] * 4) >> 4;

   h[1:9][1:9] = max(abs(g[0:8][1:9] - g[1:9][1:9]),
                     abs(g[1:9][0:8] - g[1:9][1:9]));

   write(h);
   }
```

Figure 5.3: Exemple de code HPCP

```
void main ()
   {
   int h, _hpcp_idx0, _hpcp_idx1, _hpcp_nshift, _hpcp_temp[3];
   const int _hpcp_coeff0[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
   const int _hpcp_coeff1[2] = {-1, 1}, _hpcp_coeff2[3] = {-1, 0, 1};

   _initbufAr(3, 0);
   _initbufAw(3, 0);
   _initbufBr(2, 0);
   _initbufBw(2, 0);

   for(_hpcp_idx0 = 0; _hpcp_idx0 < 10; _hpcp_idx0++){
      for(_hpcp_idx1 = 0; _hpcp_idx1 < 4; _hpcp_idx1++){
         if(_hpcp_idx0 >= 0 && _hpcp_idx0 <= 9)
            if(_hpcp_idx1 >= 0 && _hpcp_idx1 <= 9){
               for(_hpcp_nshift = 0; _hpcp_nshift < 4; _hpcp_nshift++)
                  _NorthShift();
               _writebufAw(_North);
               if(_hpcp_idx1 <= 2)
                  _hpcp_temp[_hpcp_idx1] = _North;
               if(_hpcp_idx1 >= 1){
                  _North = _hpcp_temp[_hpcp_idx1 - 2];
                  _NorthShift();
               }
         if(_hpcp_idx0 >= 1 && _hpcp_idx0 <= 8)
            if(_hpcp_idx1 >= 1 && _hpcp_idx1 <= 8)
               _writebufBw(_convolIterbufAw(_hpcp_coeff0, 9) >> 4);
         if(_hpcp_idx0 >= 1 && _hpcp_idx0 <= 9)
            if(_hpcp_idx1 >= 1 && _hpcp_idx1 <= 9)
               h = _max(_abs(_convolIterbufBw(_hpcp_coeff1, 2)),
                        _abs(_convolIterbufBw(_hpcp_coeff2, 3)), -32768);
         if(_hpcp_idx0 >= 0 && _hpcp_idx0 <= 9)
            if(_hpcp_idx1 >= 0 && _hpcp_idx1 <= 9){
               _South = h;
               for(_hpcp_nshift = 0; _hpcp_nshift < 4; _hpcp_nshift++)
                  _SouthShift();
               }
         }
      }
   }
```

Figure 5.4: Code C-Pulse généré

5.5 Analyse des performances obtenues

Étant donné que le compilateur créé ne comprend pas les optimisations habituelles, le code généré n'est pas très performant. En particulier, à l'analyse du code de la figure 5.4, on constate que les boucles imbriquées traversent tout l'espace d'indexation et qu'un énoncé "if" est utilisé pour les énoncés qui correspondent à chaque énoncé du source HPCP. Il serait beaucoup plus efficace d'extraire les itérations qui ne font que le traitement des frontières et de limiter l'espace d'indexation parcouru. Également, certains de ces énoncés "if" pourraient être éliminés par fusion ou par élimination (lorsque leur condition est toujours vraie i.e. lorsque le traitement à faire doit l'être sur tout le nouvel espace d'indexation — après optimisation). Finalement, le code assembleur généré pourrait aussi être amélioré. La principale amélioration consiste à utiliser les instructions "push", "pop" et "dbr" pour implanter les boucles imbriquées plutôt que des "Sub", "Ifc", "BNPA" et "BU".

Puisque la performance du code généré est surtout limité par l'absence d'optimisations classiques et que ces dernières dépassent le cadre des présents travaux, on ne comparera pas la performance du code généré avec celle de code écrit directement en assembleur. On s'attardera plutôt sur le temps ajouté pour effectuer un traitement par rapport à n'effectuer que le transfert d'un tableau de l'entrée vers la sortie (i.e. d'un canal vers un autre) en assignant un tableau lu à un autre tableau et en effectuant l'écriture de ce deuxième tableau vers l'extérieur. Le programme utilisé pour effectuer ce transfert est donné à la figure 5.5.

Le code C-PULSE et le code assembleur générés sont donnés à l'annexe D. Le

```
main(){
   int d[8][8], h[8][8];

#pragma distribute d[*][block], h[*][block];

   read(d);
   h = d;
   write(h);
   }
```

Figure 5.5: Premier programme de test HPCP

```
main(){
   int d[10][10], h[10][10];

#pragma distribute d[*][block], h[*][block];

   read(d);

   h[1:8][1:8] =  (d[0:7][0:7] + d[0:7][2:9] + d[2:9][0:7] +
                  d[2:9][2:9] + (d[2:9][1:8] + d[0:7][1:8] +
                  d[1:8][2:9] + d[1:8][0:7]) * 2 +
                  d[1:8][1:8] * 4) >> 4;

   write(h);
   }
```

Figure 5.6: Deuxième programme de test HPCP

deuxième programme de test utilisé est celui de la figure 5.6 (les codes assembleur et C-PULSE sont également donnés en annexe).

En comparant les deux programmes assembleurs générés, on constate que:

- il y a quelques instructions supplémentaires pour l'initialisation des tampons circulaires,
- une instruction "Ld" (**Load**) a été remplacé par un appel à "_writebufAw" (qui sera remplacé par une seule instruction dans un avenir prochain) et
- une instruction "Ld" (pour effectuer l'assignation "h = d;") est remplacée par un appel à "_convolIterbufAw" (qui deviendra une seule instruction vectorielle sous peu) et par une instruction "Srl".

Donc, la seule perte en vitesse de transfert est due au temps de calcul et ce dernier est minimal puisqu'il ne consiste qu'en une instruction vectorielle et une instruction de mise à l'échelle. Ceci implique que le code généré pour une convolution aurait un niveau de performance maximale si les compilateurs utilisés contenaient toutes les optimisations qu'on retrouve habituellement dans un compilateur.

Pour corroborer cette conclusion, un programme un peu plus élaboré (celui de la figure 5.3) a été compilé (le fichiers assembleur est également donné dans l'annexe D).

On constate que d'avoir deux assignations contenant des convolutions ne diminue en rien l'afficacité du code généré donc le compilateur HPCP créé supporte efficacement les convolutions.

5.6 Conclusions

Un nouveau langage de programmation (HPCP) qui rencontre des objectifs qu'aucun autre langage ne supporte a été décrit. On a également démontré qu'une mémoire locale gérée sous forme de tampon circulaire est plus appropriée que des registres vectoriels dans les cas où on effectue des convolutions. De plus, l'efficacité des tampons circulaires (en vitesse de calcul et espace-mémoire) a été quantifiée.

Finalement, on a démontré que la performance du code C-PULSE généré par le compilateur HPCP conçu dans le cadre du présent travail n'est essentiellement limitée que par l'absence d'optimisations classiques qui dépassent le cadre des présents travaux.

Chapitre 6

Génération automatique de directives HPF

La parallélisation automatique de programmes est une tâche difficile. Les travaux de plusieurs chercheurs ont permis la mise au point d'algorithmes permettant d'y arriver dans certains contextes et selon certains objectifs. Il a été montré à la section 2.2 qu'aucune des méthodes proposées ne permet de rencontrer simultanément les objectifs suivants:

- supporter le modèle d'alignement et de distribution du HPF,
- être faite d'algorithmes dont la complexité temporelle est faible et
- calculer tant l'alignement que la distribution des tableaux.

Dans ce chapitre, une méthode qui rencontre ces objectifs est décrite. La section 6.1 décrit le cadre conceptuel utilisé, ainsi que les algorithmes permettant la parallélisation automatique, alors que la section 6.2 décrit le traducteur qui a été implanté pour valider les algorithmes proposés, ainsi que le résultats des tests effectués pour évaluer la qualité de la parallélisation. Finalement, la section 6.3 énonce les conclusions de ce chapitre.

6.1 Cadre conceptuel et algorithmes

Dans cette section, on décrit la fonction de coûts (de communications) utilisée dans les algorithmes d'alignement et de distribution des tableaux. On énonce également

comment l'information nécessaire à la prise de décision est extraite du programme à paralléliser. Finalement, les algorithmes utilisés sont décrits et expliqués.

6.1.1 Fonction de coût

Le résumé du modèle HPF de parallélisation de la section 1.1 montre que ce modèle supporte surtout les calculs basés sur les sections de tableaux parce que:

1. les partitions possibles sont des sections,
2. le réseau de processeurs est décrit comme un tableau et
3. le modèle ne permet de réduire les communications que de deux façons, soient: utiliser des partitions "épaisses"[1] et permuter les dimensions lors de l'alignement.

Par contre, l'utilisation d'une distribution cyclique avec des partitions minces peut réduire l'impact d'un mauvais équilibre des charges de calcul lorsque le traitement n'est pas totalement structuré. Ceci est possible parce que les portions de tableaux qui requierent un plus grand effort de calcul sont réparties entre les processeurs (par exemple, la décomposition LU). De toute évidence, il s'agit d'un compromis entre la répartition de la charge de calcul et la quantité de communications requise.

Ceci implique qu'un outil de génération automatique d'alignements et de distributions (qui vise les applications effectuant un traitement structuré) pourrait supporter les sections définies au moment de l'exécution de l'application ou il pourrait être limité aux sections définies au moment de la compilation (i.e. les sections définies à l'aide de constantes). La première situation nécessiterait soit une compilation spéculative, soit la redistribution ou soit une recompilation basée sur le profilage de l'exécution des applications. Par contre, les sections définies à l'exécution ont un comportement plus dynamique (par définition) donc, les utiliser avec un modèle de parallélisation aussi restrictif que celui du HPF est, dans une certaine mesure, tenter l'impossible parce que le modèle n'a pas le niveau d'expressivité nécessaire pour supporter ces sections. Donc, il a été décidé de limiter l'analyse des applications aux

[1] Les sections de tableau qui représentent les partitions n'ont une faible longueur (e.g. 1, 2 ou 3) pour aucune de leurs dimensions.

sections définies à la compilation seulement. Néanmoins, ceci devrait être suffisant pour supporter les applications visées comme, par exemple, les applications DSP et celles basées sur une grille structurée et une méthode de calcul itérative (par exemple, une méthode aux différences finies avec ou sans une méthode multi-grille).

De cette discussion, il se dégage que ce dont on a besoin pour trouver le meilleur alignement et la meilleure distribution pour chaque tableau est les sections qui interagissent. Ces relations forment un graphe dont les sommets sont les tableaux et les arcs sont les alignements nécessaires pour éviter les communications. Le graphe d'une application contient habituellement des alignements qui sont conflictuels donc, un arbre recouvrant doit être extrait du graphe dans le but d'éliminer ces conflits. Cela signifie qu'on doit choisir quels alignements seront satisfaits ce qui, en retour, implique qu'une fonction de coût doit être mise au point pour effectuer cette sélection.

Le modèle de coût est le suivant: une relation qui n'est pas satisfaite impose des communications pour transférer un nombre d'éléments de tableau égal à la somme, pour chaque dimension, du produit de la surface d'une coupe (de partitionnement) par la différence entre la valeur d'alignement de la relation et l'alignement effectif. Parce que le résultat de ce calcul est linéaire par morceaux en fonction de la différence entre les alignements des relations et l'alignement effectif, la fonction de coût devient la somme, pour toutes les dimensions, du produit de la surface de coupe par l'alignement requis par la relation. Il est à noter que cette fonction évalue le coût des communications des processeurs qui ont le coût le plus élevé; à l'opposé, les processeurs situés aux extrémités du réseau ont moins de communications car ils ont moins de voisins. Ceci n'entraîne pas d'imprécision de la fonction de coût puisque les processeurs qui effectuent moins de communications devront attendre les autres.

6.1.2 Extraction de l'information

L'unique information utilisée, pour chaque relation, est les tableaux qui interagissent et la borne inférieure, pour chaque dimension, des sections de ces tableaux. Seule la borne inférieure est utilisée parce que:

1. gérer les cas où le pas n'est pas 1 rendrait l'analyse beaucoup plus complexe alors que cette situation ne se produit pas souvent en pratique et

Tableau 6.1: Liste des relations de l'exemple de l'équation 2

tableau		alignement	
a1	a1	-1	-3
a1	a2	-1	0
a1	a2	0	3
a1	a3	-2	-1
a1	a3	-1	2
a2	a3	-1	-1

2. le fait que les sections ont la même forme et qu'on ne considère pas le pas implique que la borne supérieure n'est pas utile (i.e. la différence entre les bornes inférieures est la même que la différence entre les bornes supérieures).

Il existe une exception notable où le pas est utile; il s'agit de l'ensemble des méthodes multi-grille mais ces dernières utilisent différents pas pour des sections d'un même tableau donc, il s'agit de relations inutiles pour l'alignement.

En guise d'exemple, l'énoncé suivant:

$$a1(1:5,1:5) = a2(2:6,1:5) + a3(3:7,2:6) + a1(2:6,4:8) \qquad (2)$$

contient les relations présentées au tableau 6.1.

Les opérations supportées par les algorithmes qui sont décrits dans le reste de la présente section sont les sections, les opérateurs ainsi que les fonctions intrinsèques CSHIFT, EOSHIFT, TRANSPOSE, ALL, ANY, COUNT, PRODUCT, SUM, MAXVAL, MINVAL, SIZE et SPREAD. Ces opérations effectuent (explicitement ou implicitement) soit l'extraction d'une section, soit la déduction d'un tableau (sauf SPREAD qui fait l'opération inverse d'une réduction).

6.1.3 Algorithmes

Cette section contient, dans l'ordre, la description des étapes à franchir pour effectuer l'alignement et la distribution ainsi que les algorithmes qui implantent ces étapes.

La première étape consiste à recueillir l'information sur les relations. Dans le but de mieux représenter les coûts de communications, si une relation entre les deux même tableaux et avec le même alignement apparaît plus d'une fois dans le même

énoncé, elle est considérée comme étant une seule relation parce qu'un compilateur optimisant regroupe les communications dues à un énoncé (autant que possible); ce qui signifie que les communications ne se produisent qu'une fois par énoncé (dans le pire cas). Donc, considérer qu'une telle relation est présente à plus d'une reprise serait trop pessimiste.

La deuxième étape consiste à trier les relations selon l'identificateur du premier tableau, puis celui du second et, finalement, de la norme euclidienne de l'alignement désiré. Ce tri permet, ensuite, de regrouper les relations entre les mêmes tableaux qui ont le même alignement désiré (le nombre d'apparitions de la relation est conservé).

L'étape suivante consiste à créer les gabarits. L'algorithme suivant est utilisé:

1. trouver la dimensionalité la plus élevée parmi les tableaux qui n'ont pas encore de gabarit,
2. utiliser un tableau parmi ceux-là en guise de référence,
3. trouver tous les tableaux qui sont liés à cette référence,
4. créer un gabarit ayant la dimensionalité requise et lui lier tous ces tableaux,
5. répéter les étapes 1 à 4 jusqu'à ce que tous les tableaux aient un gabarit.

Ensuite, on doit choisir les dimensions à partitionner. Si la forme du réseau de processeurs est inconnue, toutes les dimensions (des gabarits) sont partitionnées, sinon, l'algorithme suivant est utilisé pour effectuer la sélection des dimensions:

```
pour toutes les dimensions du réseau
  pour tous les gabarits
    pour toutes les relations
      si le gabarit de la relation courante est le gabarit courant
        pour toutes les dimensions du gabarit
          si l'alignement de la relation courante selon la dimension courante est
              grand
            le coût de la dimension courante est fixé à l'infini
          sinon, si le coût de la dimension courante n'est pas infini
            calculer la surface de coupe des tableaux de la relation
```

additionner au coût de la dimension courante le produit de
l'alignement par la surface de coupe et par le nombre
d'apparitions de la relation courante
trouver la dimension au plus faible coût (en cas d'égalité, choisir celle qui a la
meilleure répartition de l'effort de calcul)
assigner à la permutation de cette dimension (de gabarit) le numéro de
dimension courante du réseau
remettre à zéro les coûts des dimensions

Lorsqu'on dit que l'alignement est grand, cela signifie qu'une opération de transposition ou de réduction est utilisée donc que les éléments des tableaux interagissent de façon plus complexes que celle supportée par le modèle (i.e. des sections qui interagissent).

L'étape suivante consiste à extraire l'arbre recouvrant; il s'agit, dans un premier temps, de trier les relations selon:

1. le produit de la norme euclidienne pour les dimensions partitionnées par le nombre d'apparitions de la relation et

2. selon le nombre d'apparitions de la relation uniquement.

Le deuxième critère a été choisi parce que, pour une quantité de communications donnée, utiliser un moins grand nombre de blocs de données (de plus grande dimension) diminue, habituellement, la charge imposée au réseau. Deuxièmement, les relations qui ont les coûts les plus élevés sont choisies (dans le but **d'éviter** ces coûts) jusqu'à ce que l'arbre recouvrant soit complet.

Finalement, les tableaux doivent être alignés; ceci est effectué par l'algorithme suivant:

pour tous les gabarits
trouver un tableau qui utilise le gabarit courant et utiliser ce tableau en guise
de référence
mettre à zéro tous les éléments de l'alignement de cette référence
tant qu'on n'a pas terminé

indiquer que, par défaut, on a terminé
pour toutes les relations qui lient deux tableaux différents
si le gabarit de la relation courante est le gabarit courant et si
un seul des tableaux de la relation courante a été aligné
aligner l'autre tableau en utilisant l'alignement de la relation courante
accumuler les valeurs minimales et maximales des alignements pour
chaque dimension
indiquer qu'on n'a pas terminé
soustraire la valeur minimale des alignements de chacun des alignements (des tableaux) dans le but de ramener à zéro celui qui a la valeur la plus petite
créer la forme du gabarit courant (qui est la forme de la référence plus les maximums des décalages moins leurs minimums)

6.1.4 Complexité temporelle des algorithmes

La complexité des différentes étapes décrites à la section précédente sont:

trouver les relations: O(nombre d'opérateurs par expression fois nombre d'expressions)

trier les relations: O(nombre de relations fois son logarithme)

créer les gabarits: O(nombre de tableaux)

choisir les dimensions à partitionner: O(nombre de dimensions du réseau fois le nombre de gabarits fois le nombre de relations)

trier les relations (à nouveau): O(nombre de relations conservées fois son logarithme) (le nombre de relations conservées est O(nombre de tableaux))

créer les alignements: O(nombre de relations conservées)

Étant donné que le nombre de relations est beaucoup plus grand que le nombre de gabarits, que le nombre de dimensions du réseau et que le nombre d'occurences des opérateurs, le temps de tri des relations domine (i.e. la complexité temporelle est O(nombre de relations multiplié par son logarithme)).

6.2 Implantation

L'implantation a été faite sous forme d'un traducteur source-source qui ajoute des directives de parallélisation HPF à un programme Fortran 90. Le but étant de prouver le concept, le traducteur n'est pas un compilateur complet.

La grammaire implantée est celle de [2, pp. 665-689] mais elle a été modifiée dans le but d'éliminer certaines ambiguités et pour rendre l'analyse syntaxique plus facile. Donc, le traducteur ne supporte pas le Fortran 90 complet.

6.2.1 Bancs d'essais

Deux applications ont été utilisées en guise de bancs d'essai: la première est une simulation de fluides qui utilise le schème de différences finies de MacCormack [20] alors que la seconde est une application de déconvolution de signal qui calcule un estimé d'un champ de vent à partir de données de précipitations provenant d'un radar Doppler [35] (cette application sera dénomée **Semad** ci-après). Dans cette deuxième application, le schème semi-lagrangien a été remplacé par un schème aux différences finies dans le but de rendre le traitement plus régulier et pour diminuer l'effort de calcul requis.

Les deux applications consistent en 291 et 376 lignes de code respectivement (en une seule fonction car le compilateur **xlhpf** qui a été utilisé semble produire du code erroné lorsqu'il y a des appels a des fonctions définies par l'usager). Aussi, le temps d'exécution du traducteur (pour ces applications) est négligeable (i.e. quelques secondes) sur un SparcStation 2. Ceci confirme la faible complexité temporelle des algorithmes.

Les tableaux 6.2 et 6.3 donne les temps d'exécution des applications sur un ordinateur IBM SP/2 qui contient quatre processeurs. Les applications ont été compilées avec **xlhpf** et ont été exécutées sous l'environnement **POE** (mais ont été soumise par l'intermédiaire de **LoadLeveler**). Chaque donnée représente le temps moyen de 9 exécutions au minimum. La colonne "temps sans alignement" est le temps d'exécution lorsque l'alignement est fixé à 0 pour tous les tableaux alors que la colonne "charge" indique combien d'autres applications étaient exécutées en même temps que celle sous étude (une valeur de 0.75 signifie que 3 applications **séquentielles** étaient exécutées sur 3 des 4 processeurs).

Tableau 6.2: Temps d'exécution pour l'application MacCormack

nombre de processeurs	temps avec alignement	temps sans alignement	amélioration (%)	charge
Réseau de forme non-spécifiée				
1	310	310	0	0
2	203	204	0.49	0
4	300	312	3.8	1
Réseau de forme spécifiée				
2	294	203	-45	0
4	673	667	-0.9	1
2×2	818	325	-152	1

Tableau 6.3: Temps d'exécution pour l'application Semad

nombre de processeurs	temps avec alignement	temps sans alignement	amélioration (%)	charge
Réseau de forme non-spécifiée				
1	110	105	-4.8	0
2	75	60	-25	0
4	99	64	-55	0.75
Réseau de forme spécifiée				
2	38	47	19	0
4	68	95	28	1
2×2	80	65	-18	1

À l'analyse de ces tableaux, on constate que:

- il y a peu de cohérence dans les résultats,
- utiliser le réseau du SP/2 comme avec une forme de 2×2 entraîne une perte de performance,
- la qualité du code généré par **xlhpf** semble variable et cette variabilité semble dominer le changement de performance dû à la qualité de la parallélisation,
- **Semad** a une structure des calculs plus complexe et le gain de performance associé à l'utilisation de l'outil de parallélisation semble plus grand donc il semble que l'outil soit profitable lorsque les applications sont complexes,
- la structure des calculs de l'application **MacCormack** (qui est très régulière) fait en sorte que plusieurs alignements entraînent les même coûts de communications ce qui fait que la différence de performance est souvent très faible entre les cas avec alignement et ceux sans alignement.

6.3 Conclusions

Des algorithmes d'alignement et de distribution ont été décrits et on a démontré qu'ils ont une faible complexité temporelle.

Des applications ont été compilées à l'aide d'un outil qui implante ces algorithmes et leur exécution semble montrer que les compilateurs HPF ne sont pas suffisamment matures pour permettre de prédire le niveau de performance selon la configuration du système et les directives de parallélisation. Il semble donc qu'un outil de génération de directives de parallélisation doive tenir compte du compilateur pour pouvoir générer des directives judicieuses.

Chapitre 7

Généralisation et formalisation du modèle de partitionnement

Le modèle de partitionnement utilisé jusqu'à présent est celui du HPF. Au chapitre 6, des algorithmes qui permettent de calculer ce partitionnement de façon automatique ont été décrits. Une version plus contrainte de ce modèle a également été utilisée dans le chapitre 5. Cependant, le modèle HPF est très contraignant, en particulier, en ce qui concerne le fait qu'une dimension ne puisse être partitionnée qu'une seule fois. Bien que, dans le cas général, ceci ne cause pas de problème, il est bon de rendre le modèle plus flexible pour mieux supporter les cas qui seraient pathologiques avec le modèle HPF. C'est-à-dire que, même si ces cas sont plutôt rares, une perte potentielle de performance qui serait dramatique mérite qu'on améliore le support pour ces applications.

Dans le présent chapitre, on montre comment généraliser le modèle de distribution (i.e. le modèle d'alignement n'est pas modifié). La description du nouveau modèle de distribution est faite en utilisant MOA (qui est décrit à l'annexe A) et le λ-calcul [15] dans le but de formaliser le modèle.

Pour pouvoir effectuer une distribution selon la méthode qui sera décrite dans ce chapitre, la seule information nécessaire concernant le réseau de communication est sa forme, c'est-à-dire qu'on se limite à des réseaux pouvant être décrit sous forme de tableau et l'information spécifique dont on a besoin est la forme de ce tableau (i.e. on utilise le même modèle de réseau que celui supporté par HPF). Cette restriction permet malgré tout l'utilisation des réseaux les plus courants soient les réseaux à

mailles et les "k-ary n-cubes". Dans le premier cas, la forme du réseau s'obtient directement par inspection alors que dans le deuxième cas, la forme du réseau est un vecteur de n éléments valant tous k.

La section 7.1 décrit quel type de distribution on veut supporter alors que la section 7.2 décrit les algorithmes nécessaires à l'implantation de la distribution. Finalement, la section 7.3 tire des conclusions sur ce chapitre.

Il est à noter que ce chapitre est une généralisation de ce qui a été décrit dans [10].

7.1 Classe de distribution

L'objectif premier étant de définir un environnement de travail pour décrire et implanter des algorithmes de distribution, on vise à solutionner ces problèmes pour un sous-ensemble des types de distribution possibles. Puisqu'on vise à supporter les applications qui effectuent un traitement structuré (régulier) sur des tableaux, on se penche sur une classe de distribution qui se décrit facilement en termes de transformations sur des tableaux.

La classe de distribution visée est celle qui consiste à diviser un tableau perpendiculairement à un de ses axes. Étant donné qu'un ordinateur parallèle a souvent plus d'une dimension, ce processus de subdivision sera effectué pour chaque dimension du réseau. Plus précisement, pour chaque dimension du réseau, le tableau de données sera partitionné en un nombre de parties égal à la longueur du réseau dans cette dimension. Ceci est une modification au modèle HPF puisque ça permet de distribuer une dimension du tableau de données plus d'une fois. La distribution sera exprimé par un vecteur nommé $\vec{v}_p$ et utilisera les informations suivantes:

$\vec{s}_p$: forme du réseau,

$\vec{s}_a$: forme du tableau de données (qui sera noté ξ_a).

$\vec{v}_p$ définit la distribution à effectuer de la façon suivante: $\vec{v}_p[i]$ indique quelle dimension de ξ_a est partitionnée par la dimension i du réseau. Cette dimension de ξ_a est donc partitionnée en $\vec{s}_p[i]$ sous-tableaux.

On voit donc que $\vec{v}_p$ doit respecter les conditions suivantes:

$$\tau\vec{v}_p \equiv \tau\vec{s}_p$$

$$0 \leq \vec{v}_p[i] < \delta\xi_a \equiv \tau\vec{s}_a$$

Cette classe de distribution est intéressante parce qu'elle supporte les applications qui effectuent un traitement structuré sur des tableaux puisqu'elle favorise les communications locales tout en étant plus générale que celle du HPF.

7.2 Algorithmes

Étant donné $\vec{v}_p$. **Partition** (qui est exprimée à l'aide du λ-calcul) calcule la forme des partitions à partir de la forme du tableau de données et de la forme du réseau.

$$\begin{aligned}\textbf{Partition} : \lambda\vec{s}_p.\vec{v}_p.\vec{s}_t \quad \text{if} \quad & [\vec{v}_p \equiv \Theta, \vec{s}_t, \\ & \textbf{Partition}(1 \bigtriangledown \vec{s}_p, 1 \bigtriangledown \vec{v}_p, \vec{v}_p[0] \bigtriangleup \vec{s}_t +\!\!+ \tfrac{\vec{s}_t[\vec{v}_p[0]]}{\vec{s}_p[0]} +\!\!+ \\ & (\vec{v}_p[0]+1) \bigtriangledown \vec{s}_t)]\end{aligned}$$

Cette expression s'appelle elle-même récursivement et, à chaque fois, elle divise l'élément pertinent de la forme temporaire $\vec{s}_t$ par le bon élément de la forme du réseau donc, si $\vec{s}_t$ vaut $\vec{s}_a$ au début, il contient la forme des partitions $\vec{s}_{part}$ à la fin. Dans le but d'exprimer la distribution en fonction de transformations sur un tableau, elle sera exprimée comme une opération qui transforme un tableau de forme $\vec{s}_a$ en un tableau dont la forme est la concaténation de $\vec{s}_p$ et de $\vec{s}_{part}$. Donc, si $\vec{s}_{new}$ est la forme du tableau après la distribution:

$$\begin{aligned}\vec{s}_{new} &\equiv \vec{s}_p +\!\!+ \textbf{Partition}(\vec{s}_p, \vec{v}_p, \vec{s}_a) \\ &\equiv \vec{s}_p +\!\!+ \vec{s}_{part}\end{aligned}$$

Maintenant, on doit trouver comment transformer ξ_a en ξ_{new} (le tableau de forme $\vec{s}_{new}$). On ne peut pas simplement faire un "reshape" ($\hat{\rho}$) de ξ_a parce que cet opérateur préserve l'ordre lexicographique. Si, par exemple, $\rho\xi_a \equiv < 4\ 6 >$, $\rho\xi_p \equiv < 2\ 3 >$ et

$$\xi_a \equiv \begin{bmatrix} 86 & 24 & 53 & 45 & 74 & 90 \\ 6 & 56 & 43 & 15 & 84 & 82 \\ 83 & 51 & 76 & 47 & 25 & 32 \\ 35 & 68 & 79 & 42 & 21 & 91 \end{bmatrix}$$

Alors, avec le "reshape", on obtiendrait (en notant que $\vec{s}_{part}$ est $< 2\ 2 >$):

$$\xi_{new} \equiv \begin{bmatrix} \begin{bmatrix} 86 & 24 \\ 53 & 45 \end{bmatrix} \begin{bmatrix} 74 & 90 \\ 6 & 56 \end{bmatrix} \begin{bmatrix} 43 & 15 \\ 84 & 82 \end{bmatrix} \\ \\ \begin{bmatrix} 83 & 51 \\ 76 & 47 \end{bmatrix} \begin{bmatrix} 25 & 32 \\ 35 & 68 \end{bmatrix} \begin{bmatrix} 79 & 42 \\ 21 & 91 \end{bmatrix} \end{bmatrix}$$

Mais on doit avoir:

$$\xi_{new} \equiv \begin{bmatrix} \begin{bmatrix} 86 & 24 \\ 6 & 56 \end{bmatrix} \begin{bmatrix} 53 & 45 \\ 43 & 15 \end{bmatrix} \begin{bmatrix} 74 & 90 \\ 84 & 82 \end{bmatrix} \\ \\ \begin{bmatrix} 83 & 51 \\ 35 & 68 \end{bmatrix} \begin{bmatrix} 76 & 47 \\ 79 & 42 \end{bmatrix} \begin{bmatrix} 25 & 32 \\ 21 & 91 \end{bmatrix} \end{bmatrix}$$

si on veut implanter l'algorithme décrit ci-haut.

Étant donné qu'une dimension du tableau de données est divisée pour "créer" chaque dimension du réseau, on doit entrelacer les dimensions des partitions avec celles du tableau de processeurs pour obtenir la forme de tableau dans laquelle les partitions sont intactes. Pour le démontrer, on a besoin du théorème 1 où ξ_e est un tableau de données non-vide, j est la dimension de ξ_e qui est partitionnée, $\vec{d}$ indique comment la dimension j de ξ_e est partitionnée (par exemple, si elle est partitionnée en 3 et que les partitions résultantes sont partitionnées en 5 et que les partitions résultantes sont partitionnées en 4 alors $\vec{d} \equiv < 3\ 5\ 4\ \frac{(\rho\xi_e)[j]}{(3\times5\times4)} >$) et $\vec{v}_s$ est la forme après que la dimension j ait été partitionnée.

Le théorème montre que, si on effectue un "reshape" d'un tableau en remplaçant une de ses dimensions par un certain nombre de dimensions pour lequel le nombre total de sous-tableaux reste le même, alors ces sous-tableaux (indexés par $\vec{\imath}$ dans le théorème) restent les mêmes parce que le "reshape" préserve l'ordre lexicographique. Ce qui change est l'ordre dans lequel ces sous-tableaux sont combinés pour former le tableau complet. Il est à noter que ce théorème montre que chaque dimension est indépendante des autres sous cette transformation et, donc, qu'on peut appliquer la distribution à plusieurs dimensions à la fois.

Théorème 1 *Si* $\exists j,\ 0 \leq j < \delta\xi_e$ *tel que* $\pi\vec{d} \equiv (\rho\xi_e)[j]$ *et que* $0 \leq^* \vec{\imath} <^* (j+1) \Delta \rho\xi_e$, $0 \leq^* \vec{k} <^* \vec{d}$, $0 \leq^* \vec{l} <^* \vec{v}_s$

Si on pose

$$\vec{v}_s \equiv (j \triangle \rho\xi_e) +\!\!+ \vec{d} +\!\!+ ((j+1) \triangledown \rho\xi_e)$$

$$\vec{k} \equiv \gamma'(({}^{-}1 \triangle \vec{\imath})[0] \; ; \; \vec{d})$$

Alors

$$\vec{\imath}\psi\xi_e \equiv (({}^{-}1 \triangledown \vec{\imath}) +\!\!+ \vec{k})\psi(\vec{v}_s \, \hat{\rho}\, \xi_e)$$

<u>Preuve:</u>

$$\begin{aligned}
\rho(({}^{-}1 \triangledown \vec{\imath}) +\!\!+ \vec{k})\psi(\vec{v}_s \, \hat{\rho}\, \xi_e) &\equiv ((\tau\vec{\imath}) - 1 + (\tau\vec{k})) \triangledown \rho(\vec{v}_s \, \hat{\rho}\, \xi_e) \quad \text{Définition de } \psi \text{ et } \hat{\rho} \\
&\equiv ((j+1) - 1 + \tau\vec{d}) \triangledown \vec{v}_s \quad \text{Définition de } \rho \\
&\quad \text{et Psi Correspondence Theorem[42]} \\
&\equiv (j + \tau\vec{d}) \triangledown ((j \triangle \rho\xi_e) +\!\!+ \vec{d} +\!\!+ ((j+1) \triangledown \rho\xi_e)) \\
&\quad \text{Définition de } \vec{v}_s \\
&\equiv (j+1) \triangledown \rho\xi_e \quad \text{Définition de } \triangle \triangledown \\
&\equiv \tau\vec{\imath} \triangledown \rho\xi_e \quad \text{Substitution} \\
&\equiv \rho(\vec{\imath}\psi\xi_e) \quad \text{Définition de } \psi
\end{aligned}$$

$$\begin{aligned}
\vec{l}\psi(({}^{-}1 \triangledown \vec{\imath}) +\!\!+ \vec{k})\psi(\vec{v}_s \, \hat{\rho}\, \xi_e) &\equiv (({}^{-}1 \triangledown \vec{\imath}) +\!\!+ \vec{k} +\!\!+ \vec{l})\psi(\vec{v}_s \, \hat{\rho}\, \xi_e) \quad \text{Définition de } \psi \\
&\equiv (({}^{-}1 \triangledown \vec{\imath}) +\!\!+ \vec{k} +\!\!+ \vec{l})\psi(\vec{v}_s \, \hat{\rho}\, \mathbf{rav}\, \xi_e) \\
&\equiv (\mathbf{rav}\, \xi_e)[\gamma(({}^{-}1 \triangledown \vec{\imath}) +\!\!+ \vec{k} +\!\!+ \vec{l} \; ; \; \vec{v}_s) \bmod \tau\xi_e]
\end{aligned}$$

et

$$\vec{l}\psi\vec{\imath}\psi\xi_e \equiv (\mathbf{rav}\, \xi_e)[\gamma(\vec{\imath} +\!\!+ \vec{l} \; ; \; \rho\xi_e) \bmod \tau\xi_e]$$

Donc, on doit montrer que

$$\gamma({}^{-}1 \triangledown \vec{\imath} +\!\!+ \vec{k} +\!\!+ \vec{l} \; ; \; \vec{v}_s) \equiv \gamma(\vec{\imath} +\!\!+ \vec{l}; \rho\xi_e)$$

$$\gamma({}^{-}1 \triangledown \vec{\imath} +\!\!+ \vec{k} +\!\!+ \vec{l} \; ; \; \vec{v}_s) \equiv \gamma(({}^{-}1 \triangledown \vec{\imath}) +\!\!+ \vec{k} +\!\!+ \vec{l}; j \triangle \rho\xi_e +\!\!+ \vec{d} +\!\!+ (j+1) \triangledown \rho\xi_e)$$

$$\equiv \gamma((^{-}1 \nabla \vec{\imath}) +\!\!+ (^{-}1 \triangle \vec{\imath}) +\!\!+ \vec{l}\,;\ (j \triangle \rho\xi_e) +\!\!+ (\rho\xi_e)[j] +\!\!+ ((j+1) \nabla (\rho\xi_e)))$$

Parce que $\gamma(\vec{k}; \vec{d}) \equiv (^{-}1 \triangle \vec{\imath})[0]$

et $\pi\vec{d} \equiv (\rho\xi)[j]$

$$\equiv \gamma(\vec{\imath} +\!\!+ \vec{l}\,;\ ((j+1) \triangle \rho\xi_e) +\!\!+ ((j+1) \nabla (\rho\xi_e)))$$

Définition de $+\!\!+$

$$\equiv \gamma(\vec{\imath} +\!\!+ \vec{l}\,;\ \rho\xi_e) \quad \text{Définition de } +\!\!+$$

CQFD

Ce théorème montre que, pour distribuer un tableau en le divisant selon une de ses dimensions, on doit faire un "reshape" en utilisant, en guise de nouvelle forme, le vecteur obtenu par la concaténation de la longueur du tableau de processeurs selon chaque dimension qui partitionne la première dimension du tableau de données et de la dimension de cette première dimension (après la distribution) et en répétant pour chaque dimension du tableau de données.

La lambda-expression suivante donne la forme désirée:

$$\begin{aligned}
\textbf{Init} : \lambda\vec{s}_p.\vec{v}_p.\vec{s}_{part}.\vec{g}_v.\vec{s}_{init} \quad \text{if} \quad & [\tau\vec{g}_v \equiv 1, \vec{s}_{init} +\!\!+ \vec{s}_p[\vec{g}_v[0]] +\!\!+ \vec{s}_{part}[0] \\
& \text{if } [\vec{v}_p[\vec{g}_v[0]] \equiv \vec{v}_p[\vec{g}_v[1]], \textbf{Init}(\vec{s}_p, \vec{v}_p, \vec{s}_{part}, 1 \nabla \vec{g}_v, \\
& \qquad\qquad \vec{s}_{init} +\!\!+ \vec{s}_p[\vec{g}_v[0]]) \\
& \textbf{Init}(\vec{s}_p, \vec{v}_p, 1 \nabla \vec{s}_{part}, 1 \nabla \vec{g}_v, \vec{s}_{init} +\!\!+ \vec{s}_p[\vec{g}_v[0]] +\!\!+ \vec{s}_{part}[0])]]
\end{aligned}$$

Donc, le tableau après cette opération est donnée par:

$$\begin{aligned}
\xi_{init} &\equiv \textbf{Init}(\vec{s}_p, \vec{v}_p, \vec{s}_{part}, \text{gu } \vec{v}_p, \Theta)\, \hat{\rho}\, \xi_a \\
&\equiv \vec{s}_{init}\, \hat{\rho}\, \xi_a
\end{aligned}$$

Il ne reste plus alors qu'à transposer ξ_{init} pour obtenir ξ_{new} parce que, dans ξ_{init}, les dimensions du tableau de processeurs sont entrelacées avec celles des partitions alors qu'on les veut séparées. Le résultat de **Gentv**, donnée ci-dessous, est le vecteur qui donne la permutation nécessaire.

$$\begin{aligned}
\textbf{Gentv} : \lambda\vec{v}_p.\vec{g}_v.\vec{t}_v.\text{end} \quad \text{if} \quad & [\tau\vec{g}_v \equiv 1, \vec{t}_v +\!\!+ \vec{g}_v[0] +\!\!+ \text{end} \\
& \text{if } [\vec{v}_p[\vec{g}_v[0]] \equiv \vec{v}_p[\vec{g}_v[1]], \textbf{Gentv}(\vec{v}_p, 1 \nabla \vec{g}_v, \vec{t}_v +\!\!+ \vec{g}_v[0], \text{end}) \\
& \textbf{Gentv}(\vec{v}_p, 1 \nabla \vec{g}_v, \vec{t}_v +\!\!+ \vec{g}_v[0] +\!\!+ \text{end}, \text{end} + 1)]]
\end{aligned}$$

Donc, le tableau partitionné est donné par:

$$\begin{aligned}\xi_{new} &\equiv \mathbf{Gentv}(\vec{v}_p, \text{gu } \vec{v}_p, \Theta, \tau\vec{v}_p) \textcircled{/} (\mathbf{Init}(\vec{s}_p, \vec{v}_p, \mathbf{Partition}(\vec{s}_p, \vec{v}_p, \vec{s}_a), \text{gu } \vec{v}_p, \Theta)\, \hat{\rho}\, \xi_a) \\ &\equiv \vec{t}_v \textcircled{/} (\vec{s}_{init}\, \hat{\rho}\, \xi_a)\end{aligned}$$

Avec ces expressions, tout ce qui manque pour effectuer une distribution particulière est la valeur de $\vec{v}_p$ pour cette distribution. Ce dernier peut être calculé grâce à l'algorithme de sélection de dimensions de la section 6.1.3 (page 70) dans lequel la sélection de la dimension serait modifiée pour permettre la répétition d'une ou plusieurs dimensions.

7.2.1 Exemple

Dans cette section, on montre comment utiliser **Partition**, **Init** et **Gentv** pour calculer $\vec{s}_{part}$, $\vec{s}_{init}$ et $\vec{t}_v$ à partir de $\vec{s}_p$, $\vec{s}_a$ et $\vec{v}_p$.

Supposons que:

$$\begin{aligned}\vec{s}_p &\equiv < 5\ 3\ 4\ 3\ 3\ 4\ 5\ 3 > \\ \vec{s}_a &\equiv < 900\ 576 > \\ \vec{v}_p &\equiv < 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1 >\end{aligned}$$

Alors, chaque appel récursif de **Partition** donne:

$\vec{s}_p$	$\vec{s}_t$	$\vec{v}_p$
< 5 3 4 3 3 4 5 3 >	< 900 576 >	< 0 0 1 0 1 1 0 1 >
< 3 4 3 3 4 5 3 >	< 180 576 >	< 0 1 0 1 1 0 1 >
< 4 3 3 4 5 3 >	< 60 576 >	< 1 0 1 1 0 1 >
< 3 3 4 5 3 >	< 60 144 >	< 0 1 1 0 1 >
< 3 4 5 3 >	< 20 144 >	< 1 1 0 1 >
< 4 5 3 >	< 20 48 >	< 1 0 1 >
< 5 3 >	< 20 12 >	< 0 1 >
< 3 >	< 4 12 >	< 1 >
Θ	< 4 4 >	Θ

Donc, $\vec{s}_{part} \equiv$< 4 4 >.

Étant donné que gu $\vec{v}_p \equiv < 0\ 1\ 3\ 6\ 2\ 4\ 5\ 7 >$, chaque appel récursif de **Init** donne:

$\vec{s}_{part}$	$\vec{g}_v$	$\vec{s}_{init}$
$< 4\ 4 >$	$< 0\ 1\ 3\ 6\ 2\ 4\ 5\ 7 >$	Θ
$< 4\ 4 >$	$< 1\ 3\ 6\ 2\ 4\ 5\ 7 >$	$< 5 >$
$< 4\ 4 >$	$< 3\ 6\ 2\ 4\ 5\ 7 >$	$< 5\ 3 >$
$< 4\ 4 >$	$< 6\ 2\ 4\ 5\ 7 >$	$< 5\ 3\ 3 >$
$< 4 >$	$< 2\ 4\ 5\ 7 >$	$< 5\ 3\ 3\ 5\ 4 >$
$< 4 >$	$< 4\ 5\ 7 >$	$< 5\ 3\ 3\ 5\ 4\ 4 >$
$< 4 >$	$< 5\ 7 >$	$< 5\ 3\ 3\ 5\ 4\ 4\ 3 >$
$< 4 >$	$< 7 >$	$< 5\ 3\ 3\ 5\ 4\ 4\ 3\ 4 >$

Donc, $\vec{s}_{init} \equiv < 5\ 3\ 3\ 5\ 4\ 4\ 3\ 4\ 3\ 4 >$.
Chaque appel récursif de **Gentv** donne:

$\vec{g}_v$	$\vec{t}_v$	end
$< 0\ 1\ 3\ 6\ 2\ 4\ 5\ 7 >$	Θ	8
$< 1\ 3\ 6\ 2\ 4\ 5\ 7 >$	$< 0 >$	8
$< 3\ 6\ 2\ 4\ 5\ 7 >$	$< 0\ 1 >$	8
$< 6\ 2\ 4\ 5\ 7 >$	$< 0\ 1\ 3 >$	8
$< 2\ 4\ 5\ 7 >$	$< 0\ 1\ 3\ 6\ 8 >$	9
$< 4\ 5\ 7 >$	$< 0\ 1\ 3\ 6\ 8\ 2 >$	9
$< 5\ 7 >$	$< 0\ 1\ 3\ 6\ 8\ 2\ 4 >$	9
$< 7 >$	$< 0\ 1\ 3\ 6\ 8\ 2\ 4\ 5 >$	9

Donc, $\vec{t}_v \equiv < 0\ 1\ 3\ 6\ 8\ 2\ 4\ 5\ 7\ 9 >$.
Pour montrer que ces résultats sont corrects, on dérive $\vec{s}_{new}$.

$$
\begin{aligned}
\vec{s}_{new} &\equiv \rho(\vec{t}_v \bigcirc\!\!\!\!\backslash\ \xi_{init}) \\
&\equiv (\rho\xi_{init})[\text{gu } \vec{t}_v] \\
&\equiv \vec{s}_{init}[\text{gu } \vec{t}_v] \\
&\equiv \vec{s}_{init}[< 0\ 1\ 5\ 2\ 6\ 7\ 3\ 8\ 4\ 9 >] \\
&\equiv < 5\ 3\ 4\ 3\ 3\ 4\ 5\ 3\ 4\ 4 > \\
&\equiv \vec{s}_p \# \vec{s}_{part}
\end{aligned}
$$

7.3 Conclusions

On a montré comment MOA et le λ-calcul permettent de décrire la distribution d'un tableau de manière rigoureuse et conceptuellement simple.

La seule donnée à calculer pour utiliser cette méthode de distribution est $\vec{v}_p$.

Chapitre 8

Conclusions

Tel que discuté dans cette thèse, le traitement structuré (régulier) de tableaux englobe un grand nombre d'applications et ces applications forment une classe importante (au sens de l'utilité) en pratique.

Dans le but d'accélérer le traitement de ces applications, on a décrit un algorithme permettant d'effectuer rapidement et à peu de frais le calcul d'adresses des éléments de tableaux transformés. On a également décrit un générateur d'adresses qui implante une version parallèle de cet algorithme. Ce générateur d'adresses supporte toutes les transformations linéaires entre un vecteur d'indice et une adresse. De plus, on a montré que l'implantation matérielle a une très faible complexité et qu'elle permet de calculer une adresse par cycle d'horloge normalement (ce qui est très rapide).

On a démontré que les registres vectoriels sont une forme de mémoire locale qui n'est pas appropriée pour les convolutions, puisqu'ils forcent un gaspillage de la bande passante entre la mémoire et le processeur. Une méthode de gestion de mémoire locale sous forme de tampons circulaires a été décrite. Elle consiste à charger les éléments successifs de tableaux à des positions dans le tampon circulaire qui sont à une distance égale au nombre de rangées dans le noyau de convolution et à utiliser un tampon dont la taille force un "wrap-around", de façon à ce que les éléments d'une colonne du tableau soient à des positions successives dans le tampon. On a démontré que cette méthode permet d'extraire un maximum de performance d'instructions vectorielles. Le seul compromis de cette méthode est qu'elle nécessite légèrement plus de mémoire que le minimum nécessaire pour éviter de charger des

éléments de tableaux plus d'une fois.

On a également proposé un langage de programmation qui permet de décrire les applications qui traitent des tableaux de façon structurée à un haut niveau d'abstraction, tout en permettant tant la génération de code performant (pour des algorithmes basés sur des convolutions), ainsi que la parallélisation des applications pour des architectures SIMD. Pour permettre la génération de code performant, on s'appuie sur l'utilisation efficace de tampons circulaires et d'instructions vectorielles qui permettent d'effectuer une multiplication-accumulation par cycle d'horloge sur une architecture qui supporte ce type d'instruction.

Il a également été montré qu'il est possible de paralléliser les applications grâce à des algorithmes dont la complexité temporelle est faible. Pour ce faire, on utilise la forme et la position des sections de tableaux qui sont utilisées dans le programme à paralléliser et on trouve les sections qui doivent être alignées ainsi que les dimensions qui doivent être partitionnées de façon à minimiser les communications. Cependant, comme le langage HPF a été utilisé pour en faire la démonstration et que le compilateur utilisé manque de maturité, il n'a pas été possible de quantifier la qualité de la parallélisation obtenue. Une avenue intéressante pour y parvenir consiste à utiliser les dits algorithmes dans le compilateur HPCP créé dans le cadre de la présente thèse. Évidemment, comme l'environnement HPCP est plus contraint que celui du HPF, les conclusions de tels travaux ne pourraient pas être aussi générales.

Finalement, on a décrit comment formaliser et généraliser le modèle de partitionnement du HPF de façon à permettre de partitionner une dimension de tableau de données plus d'une fois.

Les pistes de recherche les plus intéressantes qui découlent des travaux de cette thèse sont:

- intégrer les algorithmes de parallélisation au compilateur HPCP et évaluer leur qualité tel que décrit ci-haut et

- étendre l'ensemble des fonctions intrinsèques du HPCP pour supporter des réductions et le "spread" du Fortran 90 (ceci permettrait de supporter un plus large ensemble d'applications incluant la solution de systèmes d'équations linéaires denses et les applications de déconvolution),

- supporter l'énoncé "forall" dans les algorithmes de parallélisation HPF,

- rendre le générateur d'adresses plus général en faisant en sorte qu'il supporte les transformations quadratiques dont les paramètres peuvent être rationnels (ceci permettrait de supporter plusieurs transformations utilisées en traitement d'images).

Bibliographie

[1] ACHIM, M., BONELLO, C., VAN DONGEN, V.. C-Pulse — a language for parallel DSP systems. In *Proceedings of the* 11th *Annual International Symposium on High Performance Computing Systems (HPCS'97)*, 1997.

[2] ADAMS, J., BRAINERD, W., MARTIN, J., SMITH, B., WAGENER, J., *Fortran 90 Handbook*. McGraw-Hill, New-York, 1992.

[3] AHO, A., SETHI, R., ULLMAN, J. D., *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] ASANOVIĆ, K., BECK, J., T0 engineering data. Technical Report TR-96-057, International Computer Science Intitute, Berkeley, California, USA, December 1996.

[5] ASANOVIĆ, K., JONSHON, D., Torrent architecture manual. Technical Report TR-96-056, International Computer Science Intitute, Berkeley, California, USA, December 1996.

[6] BAGRODIA, R., CHANDY, K. M., KWAN, E., Uc: a language for the connection machine. In IEEE Computer Society Press, editor, *Proceedings of Supercomputing '90*, pages 525–534, November 1990.

[7] BAU, D., KODUKULA, I., KOTLYAR, V., PINGALI, K., STODGHILL, P., Solving alignment using elementary linear algebra. *Lecture Notes in Computer Science*, 892 (Proceedings 7th International Workshop on Languages and Compilers for Parallel Computing):46–60, August 1994.

[8] BECKER, G., MURRAY, N., Symmetric indexing of arrays. In *Proceedings of MASPLAS '96 The Mid-Atlantic Student Workshop on Programming languages and Systems*, pages 4.1–4.10. SUNY at New-Paltz, April 1996.
Available at URL: `http://www.mcs.newpaltz.edu/masplas96`.

[9] BÉLANGER, N., ANTAKI, B., SAVARIA, Y., An algorithm for fast array transfers. In *Proceedings of the* 11th *Annual International Symposium on High Performance Computing Systems (HPCS'97)*, 1997.

[10] BÉLANGER, N., MULLIN, L., SAVARIA, Y., Formal methods for the partitioning, scheduling and routing of arrays on a hierarchical bus multiprocessing architecture. Technical Report 841, Université de Montréal, June 1992. Proceedings of ATABLE'92.

[11] CHATTERJEE, S., GILBERT, J., SCHREIBER, R., The alignment and distribution graph. *Lecture Notes in Computer Science*, 768 (Proceedings 6th International Workshop on Languages and Compilers for Parallel Computing):234–252, August 1993.

[12] CHATTERJEE, S., GILBERT, J., SCHREIBER, R., SHEFFLER, T., Array distribution in data-parallel programs. *Lecture Notes in Computer Science*, 892 (Proceedings 7th International Workshop on Languages and Compilers for Parallel Computing):76–91, August 1994.

[13] CHATTERJEE, S., GILBERT, J., SCHREIBER, R., TENG, S., Automatic array alignment in data-parallel programs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28. ACM, January 1993.

[14] CHING, W., Automatic parallelization of APL-style programs. *APL Qoute Quad*, 20(4):76–80, July 1990.

[15] CHURCH, A., *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[16] DALLY, W., Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[17] D'HOLLANDER, E., Partitioning and labeling of index sets in do loops with constant dependance vectors. In *Proceedings of the 1989 International Conference on Parallel Processing*, 1989.

[18] DONGARRA, J., DUFF, I., SORENSEN, D., VAN DER VORST, H., *Solving linear systems on vector and Shared Memory Computers*. SIAM, 1991.

[19] FLYNN, M., Very high-speed computing systems. *Proc. IEEE*, 54(12):1901–1909, December 1966.

[20] GARCIA, R., KAHAWITA, R., Numerical solution of the St. Venant equations with the MacCormack finite-difference scheme. *International Journal for Numerical Methods in Fuids*, 6:259–274, 1986.

[21] HAMEY, L., WEBB, J., WU, I., An architecture independant programming language for low level vision. *Computer vision, Graphics, and Image Processing*, 48:246–264, 1989.

[22] HENNESSY, J., JOUPPI, N., Computer technology and architecture: an evolving interaction. *Computer*, 24(9):18–29, September 1991.

[23] HENNESSY, J., PATTERSON, D., *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann publishers, Palo Alto, 1990.

[24] HIGH PERFORMANCE FORTRAN FORUM, High performance fortran language specification, version 1.0. Technical Report CRCP-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993.

[25] HOFFMAN, J., *Numerical Methods for Engineers and Scientists.* McGraw-Hill, 1992.

[26] IVERSON, K., *A Programming Language.* Wiley, New-York, 1962.

[27] IVERSON, K., *ISI Dictionary of J.* Iverson Software Inc., 1990.

[28] JENKINS, M., *The Q'Nial Reference Manual.* Nial systems Ltd., Ottawa, Canada, 1985.

[29] KENNEDY, K., KREMER, K., Automatic data layout for high performance fortran. Technical Report CRPC-TR94498-S, Rice University, December 1994.

[30] KERNIGHAN, B., RITCHIE, D., *The C programming language.* Prentice Hall, Englewood Cliffs, N.J., 1988.

[31] KING, C., NI, L., Grouping in nested loops for parallel execution on multicomputers. In *Proceedings of the 1989 International Conference on Parallel Processing*, 1989.

[32] KNOBE, K., LUKAS, J., STEELE, G., Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[33] KNOBE, K., NATARAJAN, V., Automatic data allocation to minimize communication on SIMD machines. *The Journal of Supercomputing*, 7(4):387–416, December 1993.

[34] KOELBEL, C., MEHTROTRA, P., ROSENDALE, J., Semi-automatic domain decomposition in BLAZE. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 521–524, August 1988.

[35] LAROCHE,S., ZAWADZKI, I., A variationnal analysis method for retrieval of three-dimensional wind field from single-doppler radar data. *Journal of the Atmospheric Sciences*, 51(18):2664–2682, September 1994.

[36] LEISS, E., *Parallel and Vector Computing: a practical Introduction.* McGraw-Hill, 1995. ISBN 0-07-037692-1.

[37] MACE, M., *Memory Storage Patterns in Parallel Processing.* Kluwer Academic, Boston, MA, 1987.

[38] MAY, D., Occam. *ACM Sigplan Notices*, 18(4):69–79, April 1983.

[39] MCCALPIN, J., Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, December 1995.

[40] MIPS., *R10000 Microprocessor User's Manual.* Available at URL: `http://www.sgi.com/MIPS/products/r10k/UMan_V2.0/HTML/t5.Ver.2.0.book_4.html` Section 1.9.

[41] MULLIN, L., *A Mathematics of Arrays.* Ph.D. dissertation, Syracuse University, December 1988.

[42] MULLIN, L., The psi correspondence theorem: Array mapping using the Psi calculus, 1992. Private communication.

[43] PERRY, D., *VHDL.* Computer Engineering. McGraw-Hill, 1994.

[44] RAMANUJAM, J., SADAYAPPAN, P., Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[45] RAYTHEON ELECTRONICS (SEMICONDUCTOR DIVISION), *TMC2301 Image Resampling Sequencer Product specification.*

[46] SHANG, W., FORTES, J., Independent partitionning of algorithms with uniform depedencies. *IEEE Transactions on Computers*, 41(2):190–206, February 1992.

[47] SHEU, J., CHANG, C., Synthesizing nested loops algorithms using nonlinear transformation method. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):304–317, July 1991.

[48] SHEU, J., TAI, T., Partitioning and mapping nested loops on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):430–439, October 1991.

[49] STREAM results available at URL: `http://www.cs.virginia.edu/stream/`.

[50] TSENG, P., *A Systolic Array Parallelizing Compiler.* Kluwer Academic Publishers, 1990.

[51] WAWRZYNEK, J., ASANOVIĆ, K., KINGSBURY, B., DAVIDSON, J, BECK, J., MORGAN, N., Spert-II: A vector microprocessor system. *IEEE Computer*, pages 79–86, March 1996.

Annexe A

Introduction à MOA

MOA [41] est un formalisme mathématique permettant de manipuler les tableaux monolithiques c'est-à-dire des structures de données orthogonales dont les éléments sont des scalaires. De l'utilisation de tableaux monolithiques et des concepts véhiculés par la définition d'un tableau, il découle un certain nombre d'opérateurs: δ donne le nombre de dimension d'un tableau; par exemple, si le tableau ξ_e est un tableau de 4 par 5 par 6, alors $\delta\xi_e \equiv 3$. L'opérateur ρ donne le vecteur qui décrit la forme d'un tableau; par exemple, $\rho\xi_e \equiv < 4\ 5\ 6 >$. L'opérateur τ calcule le nombre total d'éléments contenu dans un tableau; soit $\tau\xi_e \equiv 4 \times 5 \times 6 \equiv 120$. L'opérateur π donne le produit des éléments du tableau donc $\pi\rho\xi \equiv \tau\xi$ est toujours vrai et indique que le nombre total d'éléments de ξ est égal au produit des éléments de sa forme. L'opérateur ψ est l'opérateur d'indexation de MOA. Le résultat de cet indexation est le sous-tableau obtenu en accédant le tableau à l'aide du deuxième argument. Par exemple, toujours en utilisant le même tableau, $\forall n$ tel que $0 \leq n < 4$. $\rho(< n > \psi\xi_e) \equiv < 5\ 6 >$. Aussi $\forall n, m$ tels que $0 \leq n < 4$ et $0 \leq m < 5$. $\rho(< n\ m > \psi\xi_e) \equiv < 6 >$. L'opérateur **rav** transforme un tableau en un vecteur ayant les mêmes éléments placés dans le même ordre lexicographique (avec la dimension 0 qui a le poids le plus grand). Par exemple, si

$$\xi_f \equiv \begin{bmatrix} 2 & 4 & 6 \\ 12 & 14 & 16 \end{bmatrix}$$

alors

$$\mathbf{rav}\,\xi_f \equiv < 2\ 4\ 6\ 12\ 14\ 16 >$$

Les opérateurs γ et γ' permettent respectivement d'adresser un tableau transformé par **rav** et d'adresser un élément dans un tableau sachant sa position dans le tableau un fois transformé avec **rav**. Plus précisément,

$$(\,\mathbf{rav}\,A)[\gamma(\vec{v};\rho A)] \equiv \vec{v}\psi A$$

$$\gamma'(n;\rho A)\psi A \equiv (\,\mathbf{rav}\,A)[n]$$

où les crochets([ ]) dénotent les indices de l'élément qu'on accède dans un tableau. Finalement, notons qu'un tableau vide est appelé Θ.

L'opérateur $\hat{\rho}$ est utilisé pour changer la forme d'un tableau; par exemple:

$$< 3\ 2 > \hat{\rho}\xi_f \equiv \begin{bmatrix} 2 & 4 \\ 6 & 12 \\ 14 & 16 \end{bmatrix}$$

L'opérateur ι produit un tableau dont les sous-tableaux contiennent leurs propres coordonées dans le tableau. On notera que l'argument de ι ne peut être qu'un scalaire ou un vecteur. Par exemple, $\iota 10 \equiv < 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 >$ ou encore

$$\iota < 2\ 2 > \equiv \left[\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\right]$$

L'opérateur $+\!\!+$ effectue la concaténation de deux tableaux qui consiste à abouter les tableaux selon la dimension 0 donc les tableaux doivent avoir la même longueur pour les dimensions autres que 0 pour que le résultat soit toujours un tableau. Par exemple, si $\vec{a} \equiv < 2\ 4\ 6\ 8 >$ et $\vec{b} \equiv < 1\ 3\ 5 >$ alors $\vec{a} +\!\!+ \vec{b} \equiv < 2\ 4\ 6\ 8\ 1\ 3\ 5 >$.

L'opérateur $\triangle$ divise un tableau en deux selon la dimension 0 et ne conserve qu'un des deux tableaux résultants. La longueur du tableau résultant est donné en argument: si ce scalaire est positif alors le résultat est pris à partir de la coordonée 0 s'il est négatif alors le résultat est pris à la fin du tableau et s'il est zéro, le résultat est vide (Θ). Par exemple, $\rho(2 \triangle \xi_e) \equiv < 2\ 5\ 6 >$. L'opérateur ∇ a sensiblement le même effet sauf que l'argument scalaire indique quelle partie du tableau doit être enlevée; par exemple, $\rho(1 \nabla \xi_e) \equiv < 3\ 5\ 6 >$.

L'opérateur ϕ inverse l'ordre des éléments d'un tableau selon la dimension 0. Par exemple:

$$\phi\xi_f \equiv \begin{bmatrix} 12 & 14 & 16 \\ 2 & 4 & 6 \end{bmatrix}$$

L'opérateur $\circledcirc$ effectue une permutation des dimensions d'un tableau. Cette permutation est effectuer selon le contenu d'un vecteur $\vec{a}$ de la façon suivante: la $i^{\text{ième}}$ dimension du tableau devient la dimension $< i > \psi\vec{a}$. Par exmple, avec $\vec{a} \equiv < 0\ 1 >$, $< 0\ 1 > \circledcirc \xi_f \equiv \xi_f$ et avec $\vec{a} \equiv < 1\ 0 >$:

$$< 1\ 0 > \circledcirc \xi_f \equiv \begin{bmatrix} 2 & 12 \\ 4 & 14 \\ 6 & 16 \end{bmatrix}$$

L'opérateur θ effectue une rotation des éléments d'un tableau. Si l'opérande qui spécifie la rotation est un scalaire σ alors:

$$< i > \psi(\sigma\theta\xi) \equiv < (i + \sigma) \bmod (\rho\xi)[0] > \psi\xi$$

Si l'opérande est un tableau, on effectue une rotation sur chaque vecteur selon la dimension 0 du tableau et l'amplitude de la rotation sur chacun des vecteurs est donnée par un des éléments de la deuxième opérande.

L'opérateur **gu** crée le vecteur qui contient les index dont on a besoin pour accéder le vecteur (donné en opérande) en ordre croissant. Autrement dit, $\vec{v}$[gu $\vec{v}$] est en ordre croissant.

MOA défini également des opérateurs à haut niveau. Le premier est Ω; cet opérateur permet d'appliquer un autre opérateur à des sous-tableaux. Lorsqu'on l'utilise avec un opérateur binaire, il nécessite les opérandes suivantes:

$$\xi_l\ {}_g\Omega_{\vec{d}}\ \xi_r$$

où g est l'opérateur binaire, $\vec{d} \equiv < \sigma_l\ \ \sigma_r >$ et $\sigma_l \geq 0,\ \ \sigma_r \geq 0$. Dans cette situation, Ω divise ξ_l en sous-tableaux de σ_l dimensions (en laissant intactes les dimensions de plus faible poids), il divise également ξ_r en sous-tableaux de σ_r dimensions. Finalement, après avoir appliqué l'opérateur g sur les paires de sous-tableaux (un de ξ_l et un de ξ_r), il effectue la concaténation des tableaux résultants.

Lorsqu'on l'utilise avec un opérateur unaire, il nécessite les opérandes suivantes:

$${}_f\Omega_{\vec{d}}\ \xi$$

où f est l'opérateur unaire, $\vec{d} \equiv < \sigma >$ et $\sigma \geq 0$. Dans cette situation, Ω divise ξ en sous-tableaux de σ dimensions, il applique ensuite f sur ces tableaux et, finalement, il effectue la concaténation des tableaux résultants.

L'opérateur $\otimes$ est le produit externe généralisé. Les opérandes dont il a besoin sont deux tableaux et un opérateur scalaire. Le résultat de cet opérateur est un tableau qui contient le résultat de l'opérateur scalaire appliqué à toutes les paires d'éléments possibles (un de chaque tableau). Donc, la forme du tableau résultant est la concatenation des formes des opérandes. Par exemple, si $\vec{v} \equiv < 1\ 3\ 4 >$ et $\vec{u} \equiv < 2\ 5\ 6\ 7 >$ alors $\rho(\vec{v} \otimes_+ \vec{u}) \equiv < 3\ 4 >$ et

$$\vec{v} \otimes_+ \vec{u} \equiv \begin{bmatrix} 3 & 6 & 7 & 8 \\ 5 & 8 & 9 & 10 \\ 6 & 9 & 10 & 11 \end{bmatrix}$$

Annexe B

Code pseudo-assembleur pour la génération d'adresse

```
for_init:  ADD   R_i, R_shape_j_1, #0          ; while overhead
for:       LOAD  R_eg, R_es                    ; loop time
           ADD   R_es, R_es, R_incr_j_1        ; loop time
           SUB   R_i, R_i, #1                  ; loop time
           JNZ   R_i, for                      ; loop time

           SUB   R_j, R_ndim, #2               ; while overhead
           ADD   R_es, R_es, R_incr_j_2        ; while overhead

           JGE   R_j, o_while                  ; while time
           ADD   R_cur_j, R_cur_j, #1          ; while time
           SUB   R_temp, R_cur_j, R_shape_j    ; while time
           JNE   R_temp, o_while               ; while time

           SUB   R_cur_j, R_cur_j, R_cur_j     ; while time
           SUB   R_j, R_j, #1                  ; while time
           JLT   R_j, i_while                  ; while time
           ADD   R_es, R_es, R_incr_j          ; if time
i_while:   JGE   R_j, o_while                  ; while time
           ADD   R_cur_j_1, R_cur_j_1, #1      ; while time
```

```
            SUB   R_temp, R_cur_j_1, R_shape_j_1  ; while time
            JNE   R_temp, o_while                 ; while time

            SUB   R_cur_j_1, R_cur_j_1, R_cur_j_1 ; while time
            SUB   R_j, R_j, #1                    ; while time
            JLT   R_j, i_while_1                  ; while time
            ADD   R_es, R_es, R_incr_j_1          ; if time
            ⋮
i_while_n_1:
o_while:    JGE   R_j, for_init                   ; while overhead
```

Annexe C

Grammaire de HPCP

```
primary_expr:        identifier
                     | constant
                     | '(' conditional_expr ')'

postfix_expr:        primary_expr
                     | identifier duos
                     | identifier '(' argument_expr_list ')'

duos :               duo
                     | duos duo

duo:                 '[' duo_elem ':' duo_elem ']'
                     | '[' duo_elem ']'

duo_elem:            constant_expr
argument_expr_list:  conditional_expr
                     | argument_expr_list ',' conditional_expr

unary_expr:          postfix_expr
                     | unary_operator cast_expr
```

```
unary_operator:        '+'
                       | '-'
                       | '~'
                       | '!'

cast_expr:             unary_expr

multiplicative_expr:   cast_expr
                       | multiplicative_expr '*' cast_expr

additive_expr:         multiplicative_expr
                       | additive_expr '+' multiplicative_expr
                       | additive_expr '-' multiplicative_expr

shift_expr:            additive_expr
                       | shift_expr '<<' additive_expr
                       | shift_expr '>>' additive_expr
relational_expr:       shift_expr
                       | relational_expr '<' shift_expr
                       | relational_expr '>' shift_expr
                       | relational_expr '<=' shift_expr
                       | relational_expr '>=' shift_expr

equality_expr:         relational_expr
                       | equality_expr '==' relational_expr
                       | equality_expr '!=' relational_expr

and_expr:              equality_expr
                       | and_expr '&' equality_expr

exclusive_or_expr:     and_expr
                       | exclusive_or_expr '^' and_expr
```

```
inclusive_or_expr:      exclusive_or_expr
                        | inclusive_or_expr '|' exclusive_or_expr

logical_and_expr:       inclusive_or_expr
                        | logical_and_expr '&&' inclusive_or_expr

logical_or_expr:        logical_and_expr
                        | logical_or_expr '||' logical_and_expr

conditional_expr:       logical_or_expr
assignment_expr:        identifier assignment_operator conditional_expr
                        | identifier duos assignment_operator conditional_expr
                        | identifier '(' argument_expr_list ')'

assignment_operator:    '='
                        | '*='
                        | '+='
                        | '-='
                        | '<<='
                        | '>>='
                        | '&='
                        | '^ ='
                        | '|='

constant_expr:          conditional_expr

declaration:            declaration_specifiers init_declarator_list ';'

declaration_specifiers: type_specifier
                        | declaration_specifiers type_specifier

init_declarator_list:   init_declarator
                        | init_declarator_list ',' init_declarator
```

```
init_declarator:        declarator
                        | identifier '=' initializer

initializer:            constant_expr


type_specifier:         'long'
                        | 'int'
                        | 'const'

declarator:             identifier
                        | declarator '[' constant_expr ']'


statement:              compound_statement
                        | expression_statement
                        | selection_statement
                        | pragma_statement
                        | where_statement
                        | loop_statement


compound_statement:     '{' '}'
                        | '{' statement_list '}'
                        | '{' declarations statement_list '}'
                        | '{' declarations dist_list statement_list '}'

declarations:           declaration_list

dist_list:              distribute_pragma
                        | dist_list distribute_pragma
```

```
distribute_pragma:      '#pragma' 'distribute' pragma_list ';'

pragma_list:            pragma_item
                        | pragma_list ',' pragma_item

pragma_item:            identifier dist_descs

dist_descs:             dist_desc
                        | dist_descs dist_desc

dist_desc:              '[' qualif ']'

qualif:                 '*'
                        | 'block'
                        | 'cyclic'
                        | 'cyclic' '(' constant_expr ')'


declaration_list:       declaration
                        | declaration_list declaration
statement_list:         statement
                        | statement_list statement

expression_statement:   assignment_expr ';'

selection_statement:    'if' '(' conditional_expr ')' statement
                        | 'if' '(' conditional_expr ')' statement 'else' statement

where_statement:        'where' '(' conditional_expr ')' statement

loop_statement:         'loop' statement

pragma_statement:       '#pragma' 'configuration' constant_expr
```

```
file:                 function_definition

function_definition:  identifier '(' ')' function_body

function_body:        compound_statement

identifier:           (alpha | '_')(alpha | digit | '_')*

alpha:                [a-zA-Z]

digit:                [0-9]

constant:             0 [xX] xdigit+
                      | 0 digit+
                      | digit+
                      | xdigit+

xdigit:               [0-9a-fA-F]
```

Annexe D

Code C-PULSE et assembleur des programmes de test

```
void main (){
   int d, h;
   int _hpcp_idx0, int _hpcp_idx1, _hpcp_nshift, _hpcp_temp[1];

   for(_hpcp_idx0 = 0; _hpcp_idx0 < 8; _hpcp_idx0++){
      for(_hpcp_idx1 = 0; _hpcp_idx1 < 2; _hpcp_idx1++){
         if(_hpcp_idx0 >= 0 && _hpcp_idx0 <= 7)
            if(_hpcp_idx1 >= 0 && _hpcp_idx1 <= 7){
               for(_hpcp_nshift = 0; _hpcp_nshift < 4; _hpcp_nshift++)
                  _NorthShift();
               d = _North;
               if(_hpcp_idx1 <= 0)
                  _hpcp_temp[_hpcp_idx1] = _North;
               if(_hpcp_idx1 >= 1){
                  _North = _hpcp_temp[_hpcp_idx1 - 2];
                  _NorthShift();
               }
         if(_hpcp_idx0 >= 0 && _hpcp_idx0 <= 7)
            if(_hpcp_idx1 >= 0 && _hpcp_idx1 <= 7)
               h = d;
         if(_hpcp_idx0 >= 0 && _hpcp_idx0 <= 7)
            if(_hpcp_idx1 >= 0 && _hpcp_idx1 <= 7){
               _South = h;
               for(_hpcp_nshift = 0; _hpcp_nshift < 4; _hpcp_nshift++)
                  _SouthShift();
               }
         }
      }
   }
```

Figure D.1: Code C-PULSE généré pour le programme de la figure 5.5

```
    Ld #0 rb1
  Label L0
    Sub rb1 #8 rb3
    Ifc rb3 r0
      BNPA L2
    Ld #0 ra1
  Label L3
    Sub ra1 #2 ra4
    Ifc ra4 r0
      BNPA L5
    Ld #0 rb4
    Sub rb1 #7 ra5
    Sub rb1 #0 rb5
    Ifc rb5 #65536 r0
    Ld #0 ra6
    Sub ra1 #7 rb6
    Sub ra1 #0 ra7
    Ifc ra7 #65536 r0
    Ld #0 ra2
  Label L6
    Sub ra2 #4 rb7
    Ifc rb7 r0
      BNPA L8
    NSR
Restore
  Label L7
    Inc ra2 r0
      BU L6
  Label L8
    Ld nport ra3
    Ifc ra1 #0 r0
ResetSP _hpcp_temp
    Ld ra8 _hpcp_temp
    Ld nport ra8
Restore
    Ifc ra1 #1 r0
    Sub ra1 #2 rb8
ResetSP _hpcp_temp
    Ld ra9 _hpcp_temp
    St ra9 nport
    NSR
Restore
    Ld #0 rb9
    Sub rb1 #7 ra10
```

Figure D.2: Code assembleur généré pour le programme de la figure 5.5

```
    Sub rb1 #0 rb10
    Ifc rb10 #65536 r0
     Ld #0 ra11
    Sub ra1 #7 rb11
    Sub ra1 #0 ra12
    Ifc ra12 #65536 r0
     Ld ra3 rb2
Restore
Restore
     Ld #0 rb12
    Sub rb1 #7 ra13
    Sub rb1 #0 rb13
    Ifc rb13 #65536 r0
     Ld #0 ra14
    Sub ra1 #7 rb14
    Sub ra1 #0 ra15
    Ifc ra15 #65536 r0
     St rb2 sport
     Ld #0 ra2
  Label L9
    Sub ra2 #4 rb15
    Ifc rb15 r0
      BNPA L11
    SSR
Restore
  Label L10
    Inc ra2 r0
      BU L9
  Label L11
Restore
Restore
Restore
Restore
Restore
  Label L4
    Inc ra1 r0
      BU L3
  Label L5
Restore
  Label L1
    Inc rb1 r0
      BU L0
  Label L2
    Ret
```

Figure D.3: Code assembleur généré pour le programme de la figure 5.5 (suite)

```
void main (){
   int h, _hpcp_idx0, _hpcp_idx1, _hpcp_nshift, _hpcp_temp[3];
   const int _hpcp_coeff0[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};

   _initbufAr(3, 0);
   _initbufAw(3, 0);

   for(_hpcp_idx0 = 0; _hpcp_idx0 < 10; _hpcp_idx0++){
      for(_hpcp_idx1 = 0; _hpcp_idx1 < 4; _hpcp_idx1++){
         if(_hpcp_idx0 >= 0 && _hpcp_idx0 <= 9)
            if(_hpcp_idx1 >= 0 && _hpcp_idx1 <= 9){
               for(_hpcp_nshift = 0; _hpcp_nshift < 4; _hpcp_nshift++)
                  _NorthShift();
               _writebufAw(_North);
               if(_hpcp_idx1 <= 2)
                  _hpcp_temp[_hpcp_idx1] = _North;
               if(_hpcp_idx1 >= 1){
                  _North = _hpcp_temp[_hpcp_idx1 - 2];
                  _NorthShift();
               }
         if(_hpcp_idx0 >= 1 && _hpcp_idx0 <= 8)
            if(_hpcp_idx1 >= 1 && _hpcp_idx1 <= 8)
               h = _convolIterbufAw(_hpcp_coeff0, 9) >> 4;
         if(_hpcp_idx0 >= 0 && _hpcp_idx0 <= 9)
            if(_hpcp_idx1 >= 0 && _hpcp_idx1 <= 9){
               _South = h;
               for(_hpcp_nshift = 0; _hpcp_nshift < 4; _hpcp_nshift++)
                  _SouthShift();
               }
         }
      }
   }
```

Figure D.4: Code C-PULSE généré pour le programme de la figure 5.6

```
   Call _initbufAr
   Call _initbufAw
     Ld #0 rb1
  Label L0
    Sub rb1 #10 ra3
    Ifc ra3 r0
      BNPA L2
     Ld #0 ra1
  Label L3
    Sub ra1 #4 rb3
    Ifc rb3 r0
      BNPA L5
     Ld #0 ra4
    Sub rb1 #9 rb4
    Sub rb1 #0 ra5
    Ifc ra5 #65536 r0
     Ld #0 rb5
    Sub ra1 #9 ra6
    Sub ra1 #0 rb6
    Ifc rb6 #65536 r0
     Ld #0 ra2
  Label L6
    Sub ra2 #4 ra7
    Ifc ra7 r0
      BNPA L8
    NSR
Restore
```

Figure D.5: Code assembleur généré pour le programme de la figure 5.6

```
  Label L7
    Inc ra2 r0
      BU L6
  Label L8
   Call _writebufAw
    Ifc ra1 #2 r0
    Sra r8 nport r8
ResetSP _hpcp_temp
     Ld rb7 _hpcp_temp
     Ld r8 rb7
Restore
    Ifc ra1 #1 r0
    Sub ra1 #2 ra9
ResetSP _hpcp_temp
     Ld rb9 _hpcp_temp
     St rb9 nport
    NSR
Restore
     Ld #0 ra10
    Sub rb1 #8 rb10
    Sub rb1 #1 ra11
    Ifc ra11 #65536 r0
     Ld #0 rb11
    Sub ra1 #8 ra12
    Sub ra1 #1 rb12
    Ifc rb12 #65536 r0
   Call _convolIterbufAw
    Srl r0 #4 ra13
     Ld ra13 rb2
Restore
Restore
```

Figure D.6: Code assembleur généré pour le programme de la figure 5.6 (suite)

```
     Ld #0 rb13
    Sub rb1 #9 ra14
    Sub rb1 #0 rb14
    Ifc rb14 #65536 r0
     Ld #0 ra15
    Sub ra1 #9 rb15
    Sub ra1 #0 ra16
    Ifc ra16 #65536 r0
     St rb2 sport
     Ld #0 ra2
  Label L9
    Sub ra2 #4 rb16
    Ifc rb16 r0
      BNPA L11
    SSR
Restore
  Label L10
    Inc ra2 r0
      BU L9
  Label L11
Restore
Restore
Restore
Restore
Restore
  Label L4
    Inc ra1 r0
      BU L3
  Label L5
Restore
  Label L1
    Inc rb1 r0
      BU L0
  Label L2
    Ret
```

Figure D.7: Code assembleur généré pour le programme de la figure 5.6 (fin)

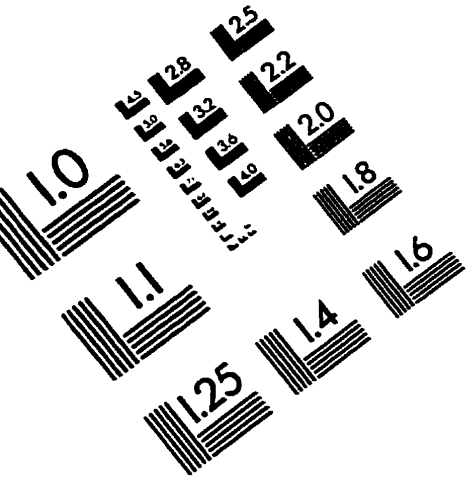
```
   Call _initbufAr
   Call _initbufAw
   Call _initbufBr
   Call _initbufBw
     Ld #0 rb1
  Label L0
    Sub rb1 #10 ra3
    Ifc ra3 r0
      BNPA L2
     Ld #0 ra1
  Label L3
    Sub ra1 #4 rb3
    Ifc rb3 r0
      BNPA L5
     Ld #0 ra4
    Sub rb1 #9 rb4
    Sub rb1 #0 ra5
    Ifc ra5 #65536 r0
     Ld #0 rb5
    Sub ra1 #9 ra6
    Sub ra1 #0 rb6
    Ifc rb6 #65536 r0
     Ld #0 ra2
  Label L6
    Sub ra2 #4 ra7
    Ifc ra7 r0
      BNPA L8
    NSR
Restore
  Label L7
    Inc ra2 r0
      BU L6
  Label L8
   Call _writebufAw
    Ifc ra1 #2 r0
    Sra r8 nport r8
ResetSP _hpcp_temp
     Ld rb7 _hpcp_temp
     Ld r8 rb7
Restore
    Ifc ra1 #1 r0
    Sub ra1 #2 ra9
```

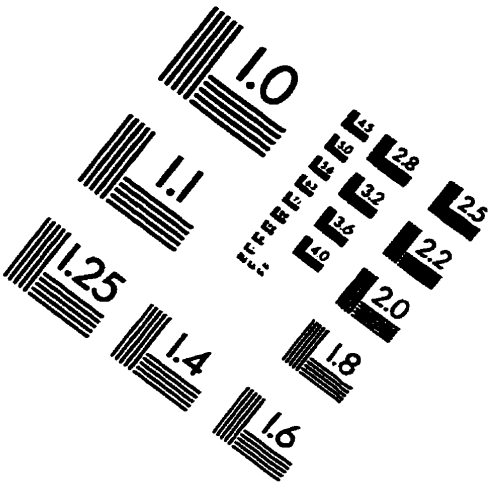
Figure D.8: Code assembleur généré pour le programme de la figure 5.3

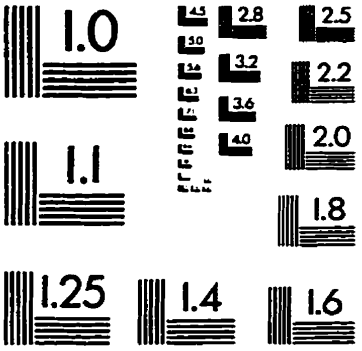
```
ResetSP _hpcp_temp
     Ld rb9 _hpcp_temp
     St rb9 nport
    NSR
Restore
     Ld #0 ra10
    Sub rb1 #8 rb10
    Sub rb1 #1 ra11
    Ifc ra11 #65536 r0
     Ld #0 rb11
    Sub ra1 #8 ra12
    Sub ra1 #1 rb12
    Ifc rb12 #65536 r0
   Call _convolIterbufAw
    Srl r0 #4 ra13
   Call _writebufBw
Restore
Restore
     Ld #0 rb13
    Sub rb1 #9 ra14
    Sub rb1 #1 rb14
    Ifc rb14 #65536 r0
     Ld #0 ra15
    Sub ra1 #9 rb15
    Sub ra1 #1 ra16
    Ifc ra16 #65536 r0
   Call _convolIterbufBw
    Abs r0 r0
   Call _convolIterbufBw
    Abs r0 r0
    Max r0 r0 #-32768
     Ld #-32768 rb2
Restore
Restore
```

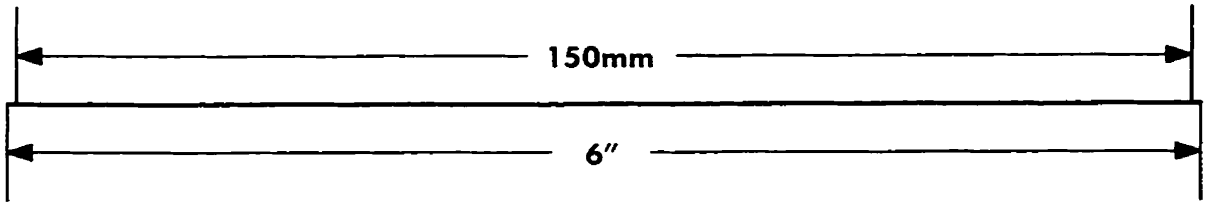
Figure D.9: Code assembleur généré pour le programme de la figure 5.3 (suite)

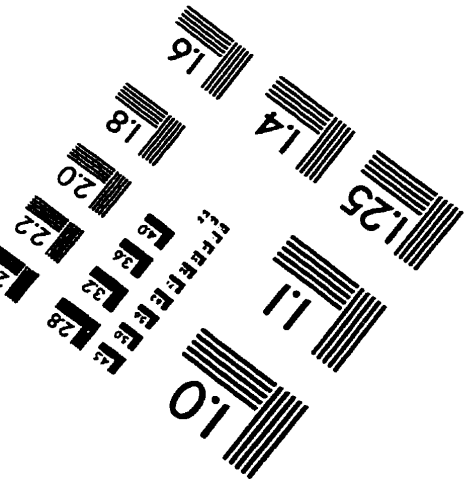
IMAGE EVALUATION TEST TARGET (QA-3)

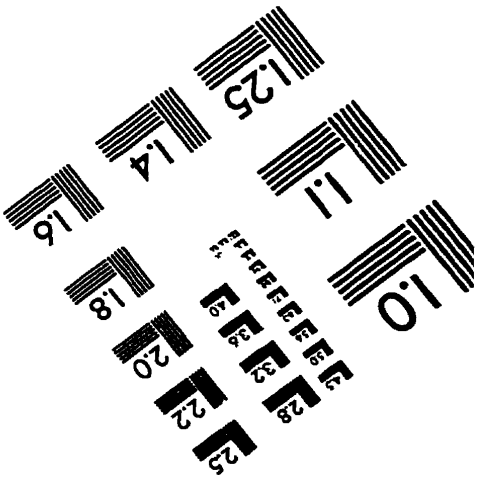












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved