UNIVERSITÉ DU QUÉBEC

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN SCIENCE APPLIQUÉES

PAR

MICHEL LANGLAIS, ING.

INTÉGRATION D'INSTRUCTIONS DATA-PARALLÈLES DANS LE LANGAGE PSC ET COMPILATION POUR PROCESSEUR SIMD (INTEL SSE)

AVRIL 2013

RÉSUMÉ

Il existe des instructions data-parallèles dans les processeurs modernes. Ces instructions permettent d'effectuer la même opération sur plusieurs données différentes en parallèle. Présentement, il est difficile de programmer des logiciels qui utilisent ces instructions data-parallèles avec les solutions existantes. Nous avons donc exploré l'utilisation d'un langage destiné à la programmation des circuits parallèles comme les FPGA (Field Programmable Gate Array) pour fabriqué un logiciel qui permet d'utiliser ces instructions data-parallèles de manière simple et efficace. Un langage de haut niveau pour la programmation des FPGA, le langage psC- Parallel and Synchronous C- a été choisi. Sa syntaxe proche du C, son paradigme entièrement parallèle et la disponibilité du code source ont justifié ce choix.

Il y a plusieurs années, les gens pensaient qu'aujourd'hui l'optimisation ne serait plus aussi importante qu'elle l'était pour eux. Ils disaient que la quantité de mémoire et la puissance de calculs des processeurs ferait en sorte que le gain en temps ne vaudrait pas l'effort de programmation nécessaire pour programmer du code optimisé. Maintenant, nous savons que ce n'est pas le cas. Les processeurs ont certes eu un gain de performance important, mais les tâches qu'ils accomplissent nécessitent de plus en plus de puissance de calculs et de mémoire. Aujourd'hui, une bonne partie de la puissance de calculs s'obtient par l'utilisation des instructions data-parallèles disponibles dans les processeurs modernes. Pour inclure ces instructions data-parallèles dans un logiciel, il n'y a pas beaucoup d'alternatives disponibles.

Ce travail a consisté à réaliser un compilateur complet pour machine SIMD. Une nouvelle syntaxe permettant de supporter les instructions data-parallèles a été définie et intégrée à celle du langage psC. L'algorithme de génération de code assembleur pour les instructions data-parallèles de type SSE d'Intel a été implémenté et testé. Finalement, trois applications ont été programmées et les performances de rapidité d'exécution comparées à diverses méthodes classiques de programmation.

Les résultats montrent que les performances obtenu par le langage psC est toujours situé entre celui obtenu par un expert codant en langage assembleur et celui obtenu par les compilateurs C et C++. Ceci correspond à ce qui était désiré.

En conclusion, ce travail de recherche a démontré qu'il était possible d'utiliser un langage HL-HDL (High Level Hardware Description Language) pour générer du code qui bénéficie des instructions data-parallèles. Le gain en performance de l'implémentation psC est présenté pour tous les cas étudié, et se rapproche de l'implémentation assembleur qui est le maximum atteignable.

REMERCIEMENTS

Je tiens à remercier tout particulièrement ma conjointe qui a été présente pour moi tout au long de cette grande aventure. Elle n'a jamais cessé de m'encourager et de croire en moi.

Je remercie également mes enfants qui ont été compréhensifs quand je devais me concentrer pour travailler au lieu d'être avec eux.

Enfin, je remercie mon directeur de maîtrise pour ses bons conseils et sa patience.

TABLE DES MATIÈRES

TABLE DES FIGURES

TABLE DES CODES

TABLE DES TABLEAUX

1

INTRODUCTION

Il y a plusieurs années, les gens du domaine de l'informatique pensaient qu'aujourd'hui l'optimisation ne serait plus aussi importante qu'elle l'était pour eux. Ils disaient que la quantité de mémoire et la puissance de calculs des processeurs ferait en sorte que le gain en temps ne vaudrait pas l'effort de programmation nécessaire pour programmer du code optimisé. Maintenant, nous savons que ce n'est pas le cas. Les processeurs ont certes eu un gain de performance important, mais les tâches qu'ils accomplissent nécessitent de plus en plus de puissance de calculs et de mémoire. Nous ne citerons que quelques exemples, qui n'étaient tout simplement pas imaginable voilà quelques années à peine: le traitement d'images à haute résolution en temps réel, le traitement de la voix et la simulation physique d'écoulement des fluides.

1.1 Parallélisme

Pendant plusieurs années, le gain de performance provenait principalement de la cadence des processeurs. Maintenant le gain de performance provient principalement du parallélisme. La définition du parallélisme selon le dictionnaire Larousse est : « Technique d'accroissement des performances d'un système informatique fondée sur l'utilisation

simultanée de plusieurs processeurs ». Selon ce même dictionnaire, la définition de processeur est : « Organe destiné, dans un ordinateur, à interpréter et exécuter des instructions ». Dans cet ouvrage, nous désignerons un processeur comme une unité de traitement (UT). Le mot processeur quant à lui, désignera ce qui est généralement reconnu par tous comme étant la composante électrique complète avec toutes les unités de traitement qui le compose.

1.1.1 Parallélisme de bit

Le parallélisme de bit consiste à traiter plusieurs bits simultanément dans une opération. Les premiers processeurs constitué de relais étaient de seulement 1 bit. C'est-à-dire qu'ils ne traitaient qu'un seul bit à la fois. Avec l'avenue des transistors, les unités de traitement pouvant traiter plusieurs bits en simultanés sont apparus, doublant le nombre de leur prédécesseurs à chaque évolution. L'apparition des ordinateurs personnels s'est faite avec des unités de traitement à 8 bits pour la compagnie IBM et les ordinateurs compatibles. La compagnie Apple à utiliser des unités de traitement de 32 bits. Présentement, les unités de traitement de 32 bit utilisés dans les ordinateurs personnels laissent graduellement la place à ceux à 64 bits.

Lorsqu'un calcul nécessite plus de bits que ce que l'unité de traitement en offre, il est nécessaire d'effectuer des calculs partiels et d'assembler les résultats partiels pour obtenir le résultat final désiré. Inutile de mentionner que ceci implique un coût en temps de calcul qui serait non nécessaire si l'unité de traitement possédait un nombre de bits suffisant pour effectuer le calcul en une seule opération.

Les programmeurs ne peuvent rien faire pour améliorer ce parallélisme, il est disponible où il ne l'est pas. La Figure 1-1 démontre la logique utilisée en fonction du nombre de bits nécessaires pour effectuer un calcul versus le nombre disponible dans le registre.

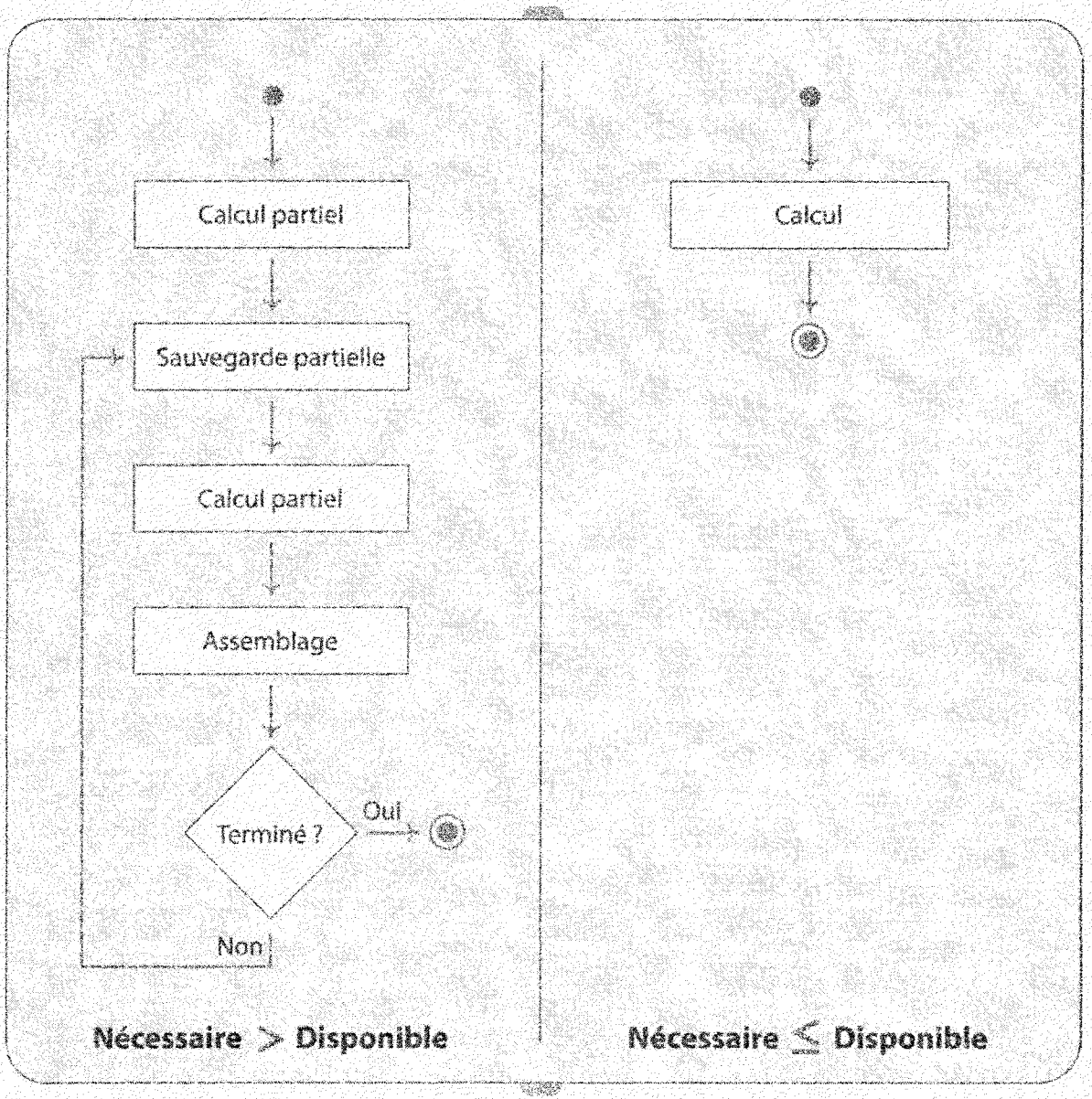

Figure 1-1: Calcul de bit.

1.1.2 Le parallélisme d'instructions

Une instruction est une commande donnée à un processeur. Une instruction peut résulter en une ou plusieurs opérations. Lors de l'exécution d'une opération, il y a plusieurs étapes impliquées dans le processeur. Le nombre d'étapes et l'ordre de celles-ci dépend de l'architecture matérielle du processeur. Cependant, il est possible d'exécuter plusieurs de ces étapes en parallèle, c'est-à-dire en simultané, dans un pipeline. La taille du pipeline a atteint 31 étapes dans les processeurs Pentium D. La Figure 1-2 montre un exemple de pipeline à 5 étapes.

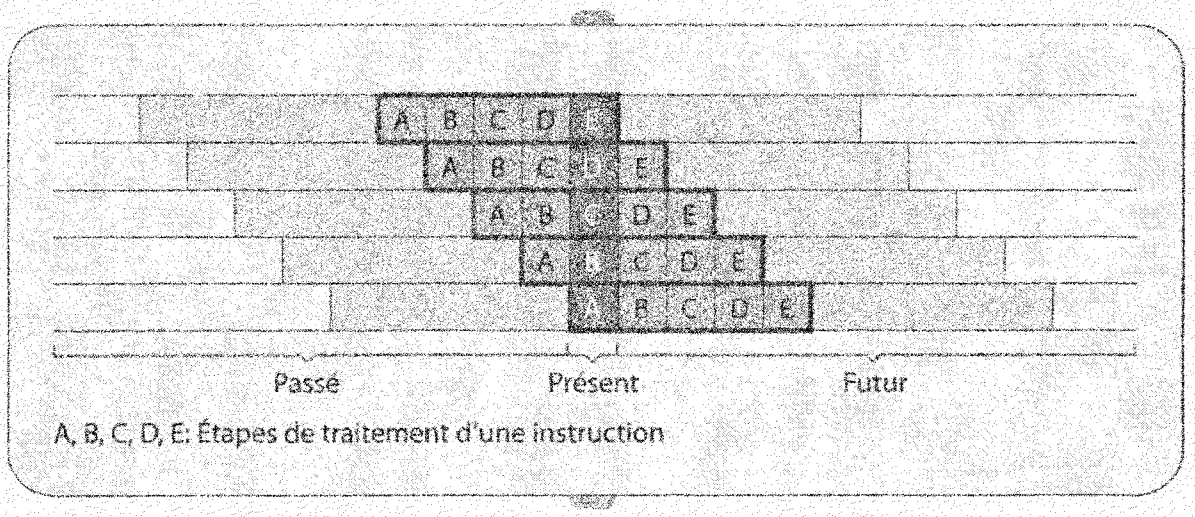

Figure 1-2: Pipeline d'instructions à cinq étapes.

Les programmeurs ont peu de contrôle sur ce parallélisme. Tout ce qu'ils peuvent faire est d'ordonner les instructions pour aider le processeur à effectuer son travail de parallélisme.

1.1.3 Le parallélisme de données

Dans plusieurs algorithmes, particulièrement les calculs vectoriels (basé sur des vecteurs), une instruction est appliquée à plusieurs données différentes. Historiquement, il était nécessaire d'appliquer la même instruction à chaque donnée. Lorsque la même instruction peut être appliquée pour effectuer le même travail sur plusieurs données, on effectue du parallélisme de données, ou data-parallélisme. Ceci réduit le nombre d'instructions par un facteur égal au nombre de données traités simultanément lorsque l'instruction data-parallèle est utilisée. Par exemple, une instruction data-parallèle qui traite quatre données en simultané réduit d'un facteur quatre le nombre d'instructions nécessaire, comparativement à une architecture qui nécessite une instruction pour chaque donnée.

Les programmeurs sont fortement impliqués dans ce parallélisme. Nous traiterons comment plus en détail dans le reste de cet ouvrage. La Figure 1-3 montre la logique d'un calcul parallèle de j opérations data-parallèles qui traitent i données à chaque instruction.

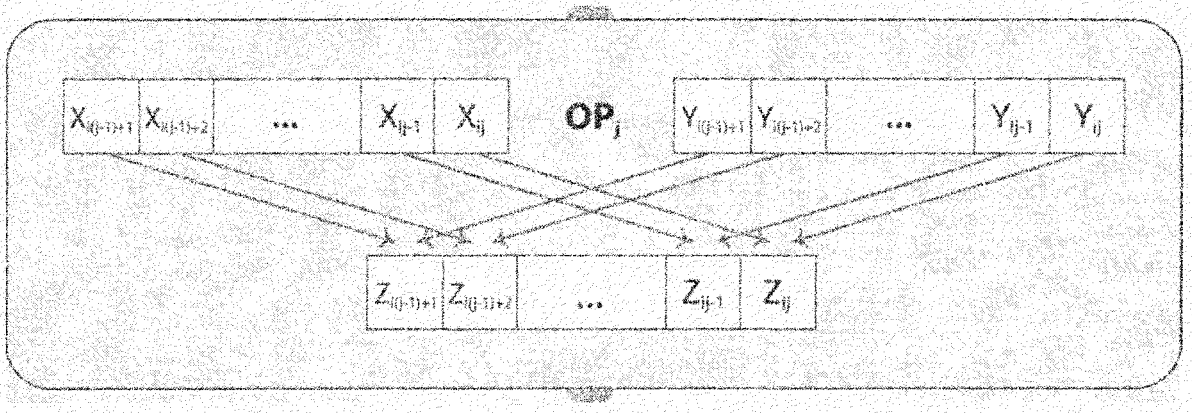

Figure 1-3: Calcul parallèle.

1.1.4 Le parallélisme de tâches

Lorsque l'on parle de parallélisme, c'est naturellement celui auquel les gens pensent. Il consiste en l'accomplissement de plusieurs tâches distinctes simultanément. Par exemple, effectuer la compression d'un fichier, jouer de la musique et effectuer un téléchargement. La Figure 1-4 le démontre graphiquement.

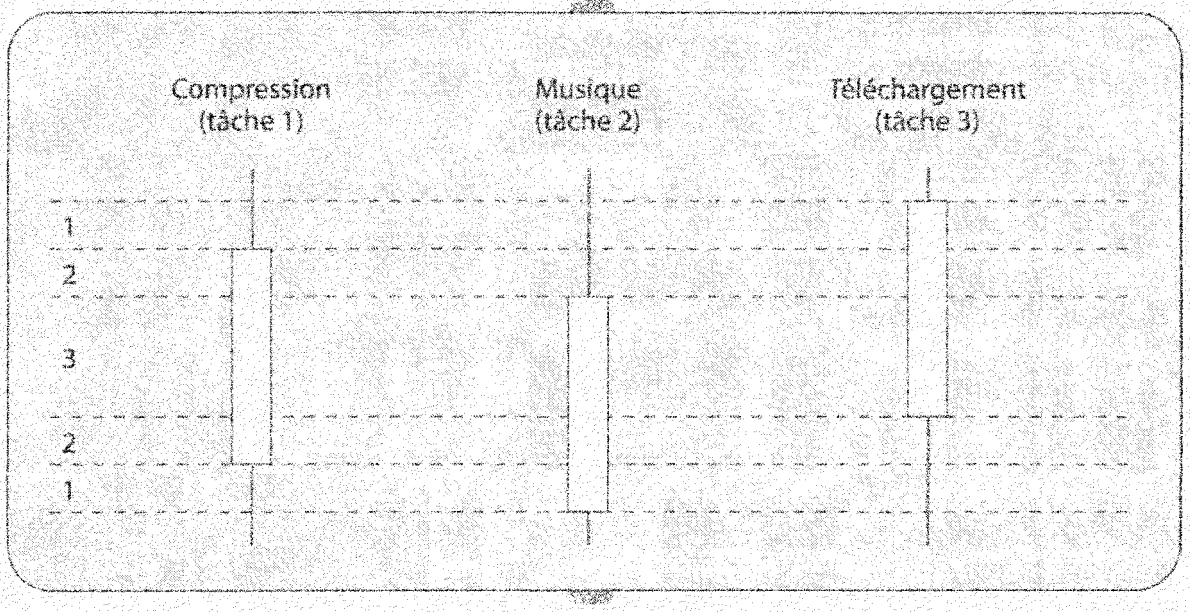

Figure 1-4: Tâches parallèles.

Les programmeurs utilisent peu ce type de parallélisme. Lorsqu'ils le font, il est nécessaire de gérer le parallélisme et toutes les synchronisations nécessaires de manière explicite. Lorsqu'implémenté sur un seul processeur, ce type de parallélisme ne produit aucun gain de performance.

1.2 Architectures parallèles

Il existe plusieurs architectures matérielles qui permettent d'effectuer des opérations parallèles. Michael J. Flynn a inventé une classification des architectures des ordinateurs (Flynn, 1972). Le Tableau 1-1 indique les types existant dans cette classification et elles sont décrites plus en détails ici-bas.

Mnémonique	Terme Complet
IUDU	Instruction Unique Donnée Unique
IMDU	Instructions Multiples Donnée Unique
IUDM	Instruction Unique Données Multiples
IMDM	Instruction Multiples Données Multiples

Tableau 1-1 : Classification des architectures des ordinateurs selon Flynn.

1.2.1 Instruction unique donnée unique

Une architecture IUDU consiste en un seul flux d'instruction qui opère sur un seul flux de données. À chaque pas, une seule instruction opère sur une seule donnée. Ce modèle est la base de l'informatique. À l'exception de quelques instructions, les processeurs à un seul cœur (unité de traitement) adhèrent à ce modèle inventé par John Von Neumann à la fin des années quarante (Wikipedia, 2010). Un algorithme qui s'exécute de manière IUDU est dit séquentiel ou sériel. Il peut contenir du parallélisme de bits et du parallélisme d'instructions. La Figure 1-5 en fait l'illustration.

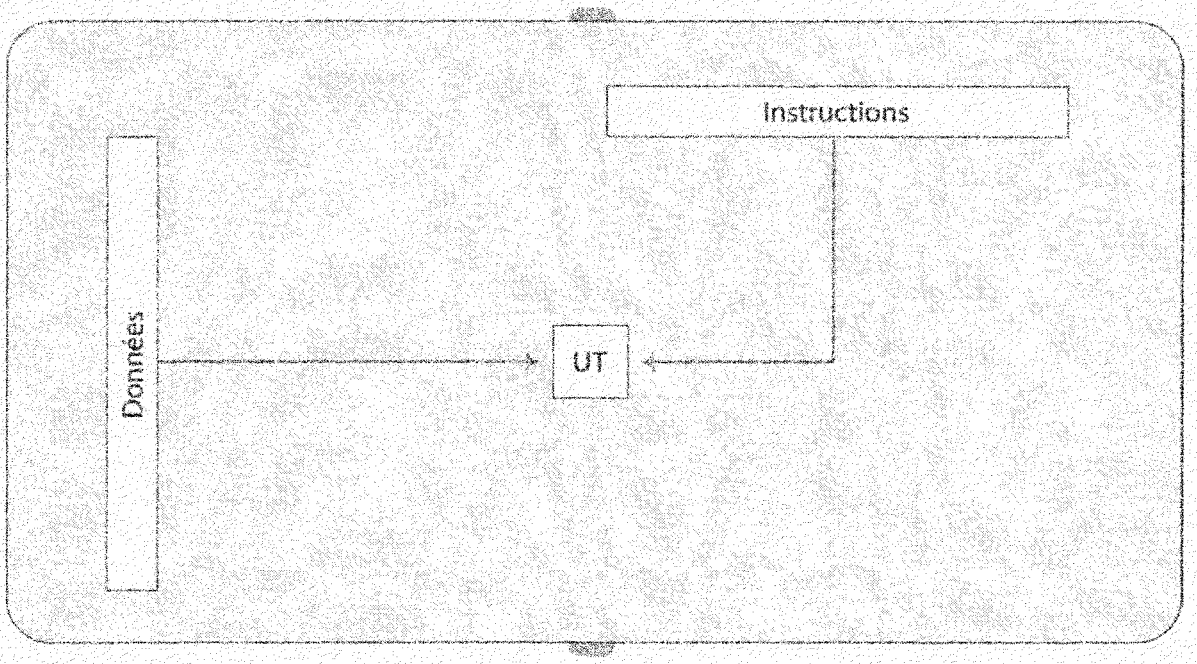

Figure 1-5: Instruction Unique Donnée Unique.

1.2.2 Instructions multiples donnée unique

Une architecture IMDU consiste en plusieurs flux d'instructions qui opèrent tous sur le même flux de données. À chaque pas, plusieurs instructions opèrent sur la même donnée. Par exemple, on peut vouloir additionner le nombre B au nombre A et également soustraire de nombre B du nombre A. Chacune de ces tâches sont distinctes et peuvent être effectuées en simultanés. La Figure 1-6 en fait l'illustration. Ce modèle est peu utilisé car peu de problèmes exigent ce type d'architecture parallèle.

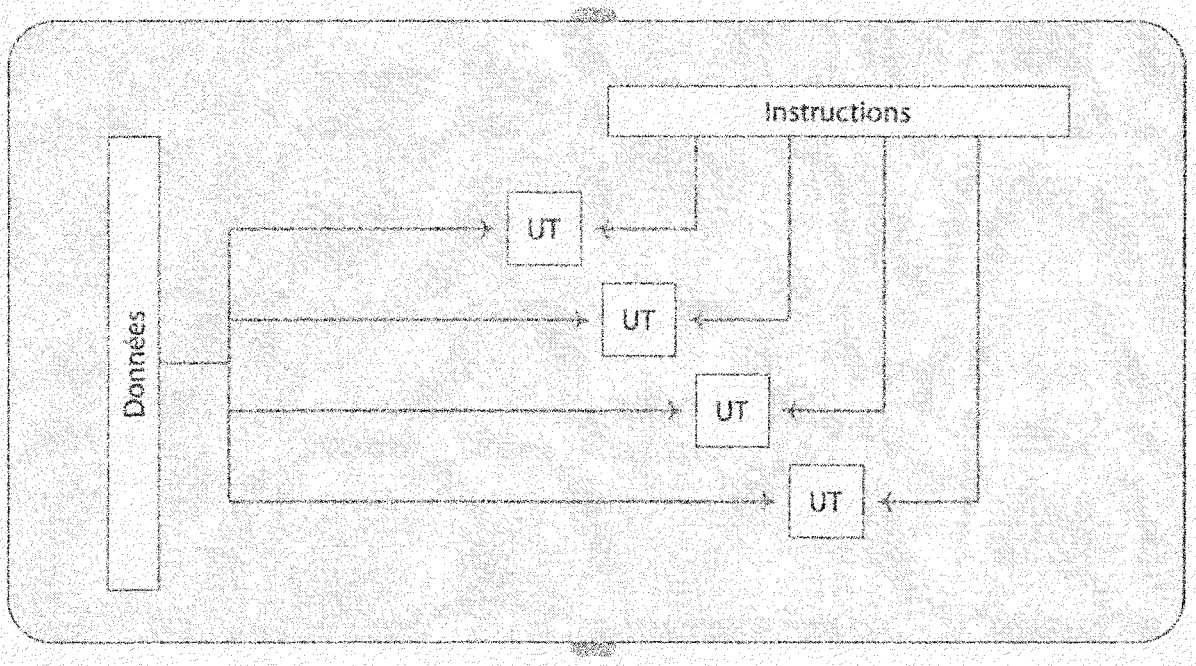

Figure 1-6: Instructions Multiples Donnée Unique.

1.2.3 Instruction unique données multiples

Une architecture IUDM consiste en un seul flux d'instruction qui opère sur plusieurs flux de données. À chaque pas, une seule instruction opère sur plusieurs données différentes. Par exemple, on peut vouloir additionner le nombre B au nombre A et le nombre D au nombre C. Chacune de ces tâches sont distinctes et peuvent être effectuées en simultanés. La Figure 1-7 en fait l'illustration. Ce modèle est utilisé dans tous les algorithmes qui traitent des vecteurs de données comme le traitement d'images, l'analyse et le traitement des signaux numériques, les comparaisons, etc. On nomme ceci le data-parallélisme.

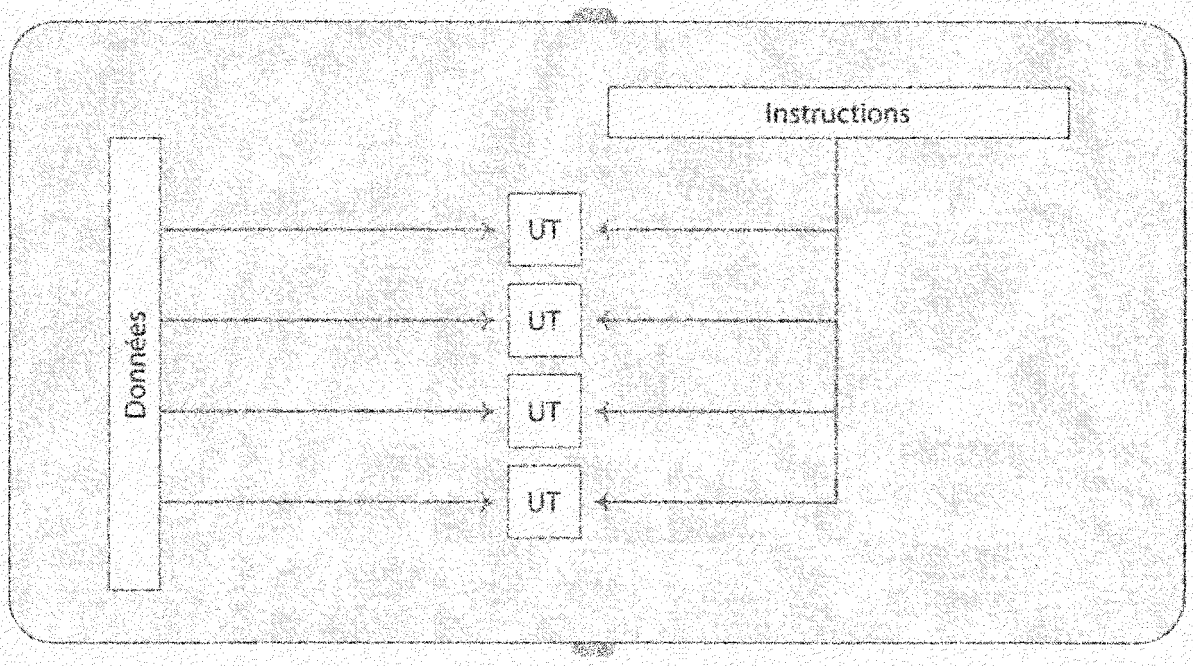

Figure 1-7: Instruction Unique Données Multiples.

1.2.4 Instructions multiples données multiples (IMDM)

Une architecture IMDM consiste en plusieurs flux d'instructions qui opèrent sur plusieurs flux de donnés. À chaque pas, plusieurs instructions opèrent sur plusieurs données différentes. Par exemple, on peut vouloir décoder un fichier encrypté et jouer de la musique. Ce modèle est utilisé pour effectuer plusieurs tâches non liées entre elles en simultané. La Figure 1-8 en fait l'illustration. Souvent cette architecture est réalisée avec des ordinateurs multiprocesseurs, qu'ils soient ou non sur un même support matériel. Chaque unité de traitement est en soit un IUDU.

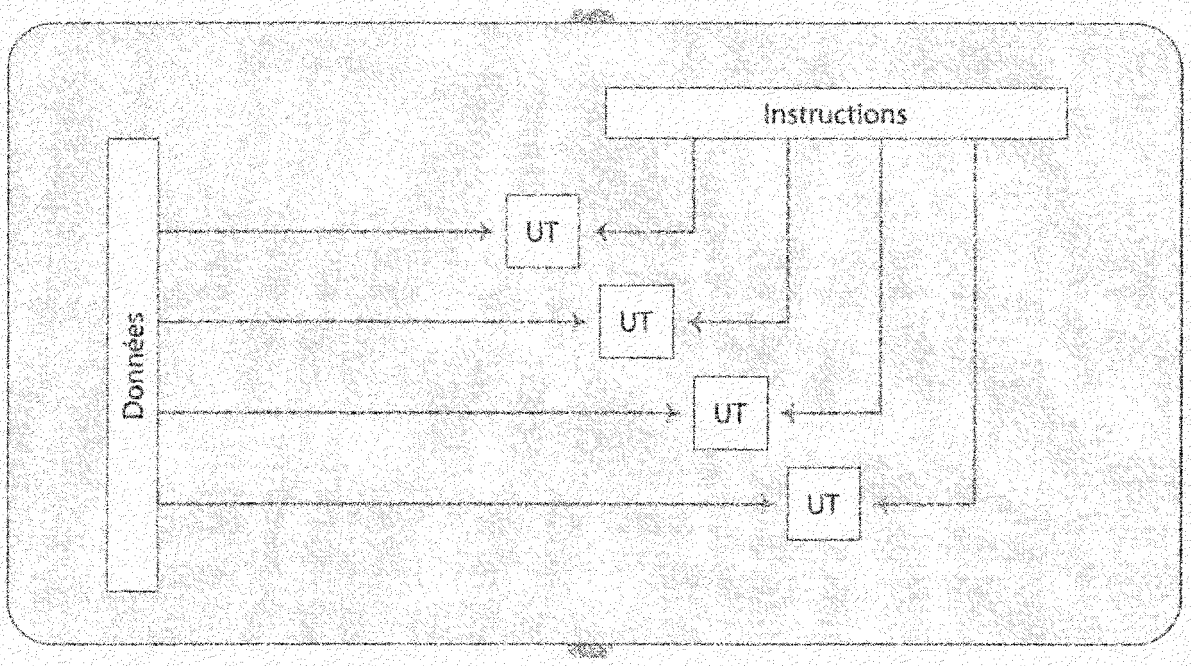

Figure 1-8: Instructions Multiples Données Multiples.

1.3 Architecture des ordinateurs personnels

Selon la classification vue précédemment, les architectures des ordinateurs personnels et des superordinateurs sont classé soit en tant que IMDM ou IUDM. Les processeurs disponibles dans les ordinateurs personnels, fabriqué principalement par les compagnies Intel et AMD, possèdent plusieurs cœurs et sont donc de types IMDM. Chacun de ces cœurs est de type IUDU. Autrement dit, les cœurs sont séquentiels. Cependant, ils possèdent des jeux instructions data-parallèles qui permettent certaines opérations IUDM.

1.3.1 Jeu d'instruction MMX

Le jeu d'instruction MMX est le premier jeu d'instruction data-parallèle disponible pour les ordinateurs personnels dans les processeurs Intel en 1996. Les instructions utilisent les mantisses des huit registres de calcul en points flottant (voir la Figure 1-9) pour

effectuer des opérations d'arithmétique entière condensé. Il est possible de travailler sur une valeur de 64 bits, deux valeurs de 32 bits, quatre valeurs de 16 bits ou huit valeurs de 8 bits.

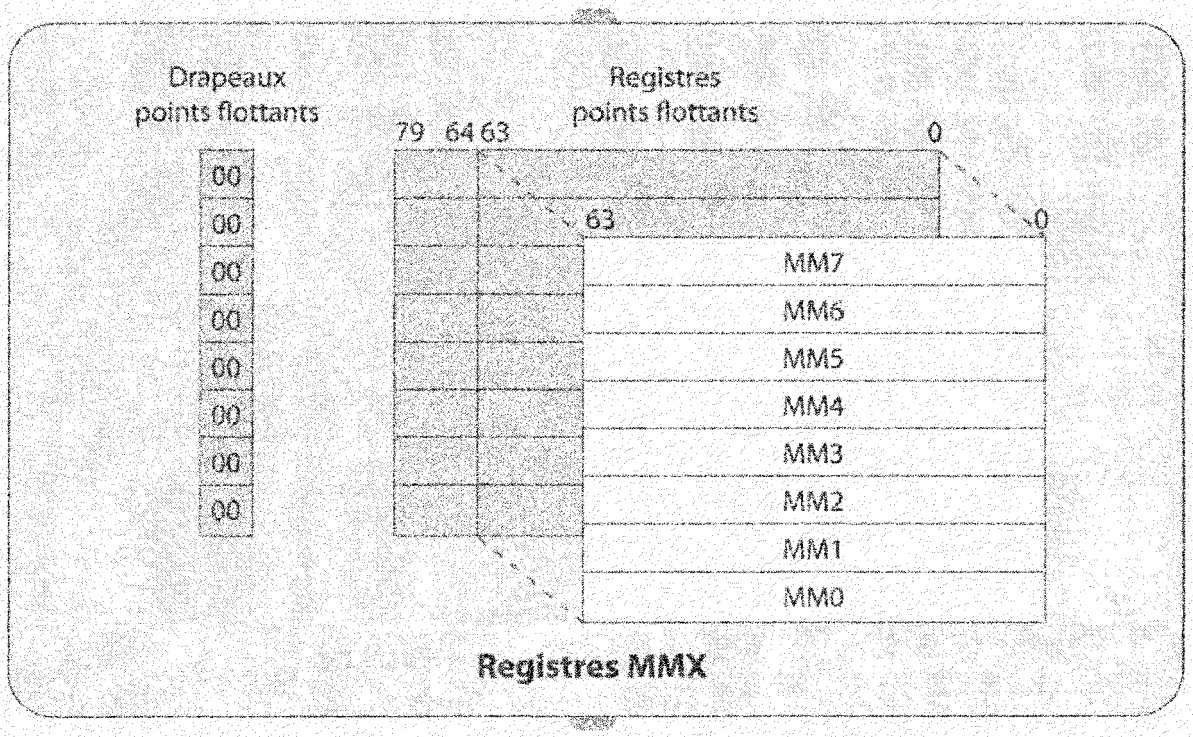

Figure 1-9: Registres MMX.

Ce jeu d'instruction comprend 57 instructions divisé en sept catégories : arithmétique de base, comparaison, conversion, logique, décalages de bits, transfert de données et gestion.

Le but de ce jeu d'instructions est d'augmenter les performances des applications multimédia et scientifiques. Toutes les applications utilisant beaucoup de calculs sur des entiers peuvent bénéficier de ce jeu d'instruction.

1.3.2 Jeu d'instruction 3DNow!

Suite à l'introduction du jeu d'instruction MMX par la compagnie Intel, la compagnie AMD a fabriqué le jeu d'instruction 3DNow! Disponible en 1998. Ce jeu d'instruction ajoute 21 instructions, dont certaines permettent d'effectuer des calculs en points flottant condensés (deux variables de 32 bits). Tout comme le jeu d'instruction MMX, ce jeu d'instruction partage les registres avec ceux du calcul en points flottant.

Tout comme le jeu d'instruction MMX et les jeux d'instructions qui lui succèderont, le but est d'améliorer les performances des applications effectuant beaucoup de calculs, peu importe qu'ils soient en valeur entières ou en points flottants.

1.3.3 Jeux d'instructions SSE et successeurs

En réplique à l'addition des opérations à points flottant condensé, la compagnie a développé le jeu d'instruction SSE, disponible depuis 1999. Il consiste en 70 instructions, traitant principalement de calculs en points flottant condensés. Contrairement aux jeux d'instructions précédant, huit nouveaux registres de 128 bits ont été ajoutés (voir la Figure 1-10).

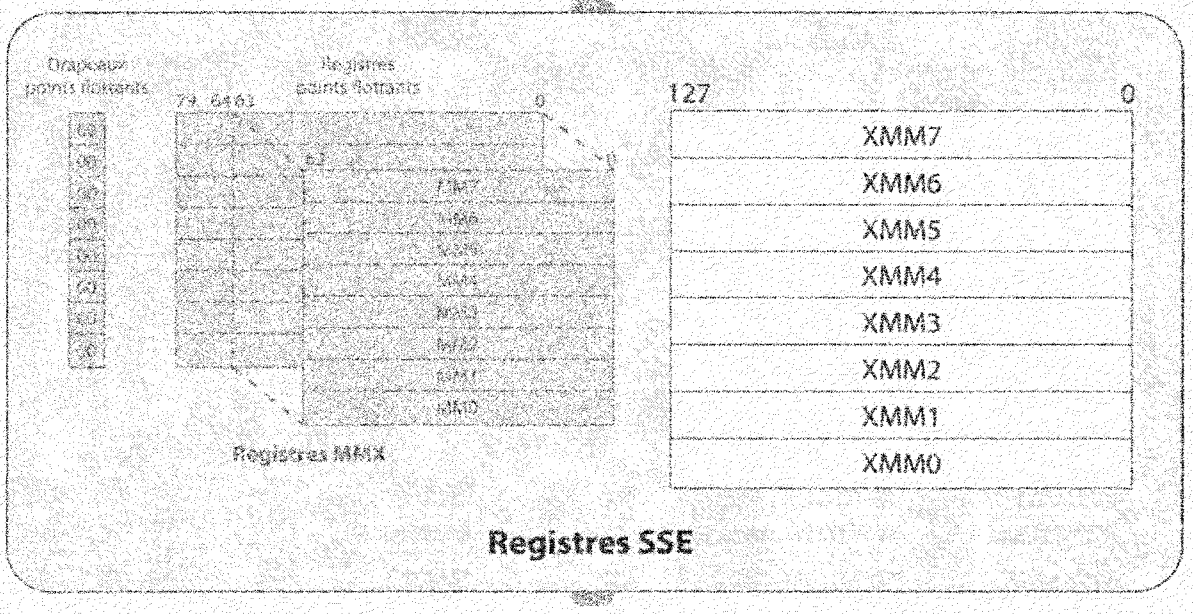

Figure 1-10: Registres SSE.

Suite au jeu d'instruction SSE, plusieurs autres jeux d'instructions ont étés ajoutés : SSE2, SSE3, SSSE3, SSE4.1 et SSE4.2. Ces jeux d'instructions ajoutent de nouvelles opérations en valeur entières et en point flottant utilisant les huit registres ajoutés avec le jeu d'instruction SSE.

1.4 Programmation

Pour effectuer une tâche sur un ordinateur, il est nécessaire de lui indiquer comment faire. Le seul langage qu'un processeur comprend est son langage machine. Le langage machine est très difficile à comprendre par les humains. Pour pallier à ce problème, le langage assembleur est né. Il fait correspondre une instruction assembleur à une instruction machine. Malgré le fait qu'un humain peut plus aisément comprendre ce langage que le langage machine, il reste très complexe. Plusieurs langages de programmation ont donc vu le jour, ayant tous pour but de permettre de coder plus simplement pour un humain. Les

instructions dans un langage de programmation haut niveau peuvent, et c'est souvent le cas, se traduisent en plusieurs instructions assembleurs ou machine. Chaque langage apporte des avantages (et inconvénients) sur certains aspects de la programmation. Ils peuvent être classés en plusieurs paradigmes, en fonction de la manière dont ils gèrent le parallélisme.

1.4.1 Paradigme impératif

Ce paradigme se caractérise par le fait que les programmes s'exécutent selon l'ordre des instructions qui le compose et qui changent son état. Dans cette catégorie, on retrouve la programmation procédurale, la programmation structurée et la programmation objet. Le C, le C++, le Pascal, le Delphi, le C# et plusieurs autres langages en sont des exemples.

Dans ce paradigme, il incombe aux programmeurs d'indiquer explicitement le parallélisme et la synchronisation. Certains langages ont des mots clé ou des fonctions qui aident les programmeurs, mais rien qui est fait de manière complètement transparente.

1.4.2 Paradigme évènementiel

Ce paradigme se caractérise par le fait que les programmes s'exécutent selon l'ordre des évènements qu'il reçoit. Un programme de ce paradigme se caractérise par deux parties distinctes : la détection des évènements et le traitement des évènements. Les diagrammes de Pétri (Petri, 1962) sont un moyen de décrire ce paradigme qui peut être interprété comme étant contrôlé par le flux de données. La plupart des librairies d'interfaces usagers entrent dans cette catégorie. Souvent, les langages impératifs sont utilisés pour fabriquer les programmes évènementiels.

Il est possible que le langage ou l'outil de génération de code permette que chaque évènement soit traité de manière parallèle aux autres évènements. Durant le traitement d'un évènement, il incombe aux programmeurs d'indiquer explicitement le parallélisme. La synchronisation entre les évènements est prise en charge. Cependant, la synchronisation qui doit être effectué à l'intérieur du traitement d'un évènement doit être faite explicitement par le programmeur.

1.4.3 Paradigme data-parallèles

Ce paradigme se caractérise par le fait que le parallélisme s'exprime en fonction d'opérations parallèles sur des structures de données.

Dans ce paradigme, il incombe aux programmeurs d'indiquer explicitement la synchronisation. Cependant, le parallélisme est fait de manière complètement transparente. Les programmes data-parallèles sont écris dans des langages spécialisés, et ne sont souvent disponible que pour une, ou un petit nombre, d'architectures matérielles. On peut citer High Performance Fortran (HPF) (Koelbel, 2010), C//, C*, Cilk, IDOLE, L, PARALLAXIS-III, Scientific Vector Language (SVL) (Santavy et Labute, 2010) et ZPL (University of Washington, 2010) comme exemples. Selon l'analyse de (Hammarlund et Lisper, 1993), les langages data-parallèles et les techniques d'exécutions parallèles sont principalement des extensions de langages séquentiels existant.

Le data-parallélisme est la forme principale de parallélisme dans les calculs scientifiques (biologie, chimie, géologie, ingénierie, etc.).

1.4.4 Paradigme fonctionnel

Ce paradigme se caractérise par le fait que les langages fonctionnels ne comportent que des affectations et des appels de fonctions. Une fonction peut être appelée aussitôt que tous ses arguments sont connus. On peut citer Sisal (Streams and Iteration in a Single Assignment Language), LISP (McCarthy, 1960) et NESL (Blelloch, 2010) comme exemples de langages fonctionnels.

La synchronisation et le parallélisme peuvent être pris en charge implicitement ou laissé à la charge du programmeur.

1.5 Problématique

Comme indiqué en 1.1, les logiciels demandent de plus en plus de puissance de calculs. Cette puissance de calcul peut provenir de l'utilisation des instructions data-parallèles disponibles dans les processeurs modernes (voir 1.3). Cependant, il existe peu de moyens d'utiliser ces instructions de manière simple et efficace. Trois techniques existent pour inclure ces instructions data-parallèles dans un programme. Il est possible d'utiliser le langage assembleur, les fonctions intrinsèques en langage C ou laisser le compilateur effectuer le travail.

1.5.1 Le langage assembleur

Pour coder en langage assembleur, il est nécessaire d'avoir une connaissance approfondie des jeux d'instructions des processeurs. Il faut comprendre le fonctionnement de la mémoire vive, des mémoires caches et des registres. Il faut être en mesure d'effectuer l'alignement des données en mémoire et d'éviter le dépassement de capacité lors du

traitement des tableaux. Le dépassement de capacité arrive lorsque la taille des tableaux n'est pas un multiple du nombre de données qui entrent dans un registre. Lors de calculs important, il faut effectuer des transferts mémoire ou utiliser la pile comme une zone de stockage temporaire car le nombre de registre disponible ne sera pas suffisant.

L'utilisation des instructions data-parallèles en assembleur ont les mêmes problématiques que tout autre code en langage assembleur, mais demandent d'aligner les données en mémoire à seize octets en plus. Autrement dit, il faut que l'adresse des données soit divisible par seize sans reste.

Le Code 1-1 montre comment l'addition des valeurs d'un vecteur d'entier peut être faite. Dans cet exemple, la définition des variables et constantes a été omise, ainsi que la pré-lecture des données en mémoire.

```
/* Additionne les entiers du vecteur x et place le résultat dans la
 * variable y.
 *
 * Étapes :
 *   1) assignation globales
 *   2) On additionne, un à la fois, les entiers de 4 octets jusqu'à ce
 *      que le vecteur soit aligné à un 16 octets en mémoire.
 *   3) On additionne en block de 4 entiers de 4 octets à la fois.
 *   4) On additionne les sommes partielles.
 *   5) On additionne les entiers restant.
 *   6) On place le résultat dans la variable y en mémoire.
 */

/*** 1) assignation globales ***/

/* pointe sur la donnée de x ou l'on est rendu */
lea esi, x

/* pointe sur ou il faut sauvegarder le résultat */
lea edi, y

/* calcul la fin du tableau */
mov edx, esi
add edx, x_size

/* y = 0; */
xor ebx, ebx

/* 2) On additionne, un à la fois, les entiers de 4 octets jusqu'à ce
 *    que le vecteur soit aligné à un 16 octets en mémoire
 */

/* calcul si on est aligné au 16 bit */
mov ecx, esi
and ecx, 0x0000000f
cmp ecx, 0
jz  forle
sub ecx, 16

/* tant qu'on n'est pas aligné */
forlb :

/* effectue une addition */
add ebx, [esi]

/* change a l'entier suivant */
add esi, 4

/* valide si on est aligné maintenant */
add ecx, 4
cmp ecx, 0
jnz forlb
```

```
/* fin de la boucle */
forle:

/*** 3) On additionne en block de 4 entiers de 4 octets à la fois ***/

/* initialise l'endroit des résultats partiels */
pxor xmm0, xmm0

/* calcul le dernier endroit aligné du tableau */
mov eax, edx
and eax, 0xfffffff0

/* valide qu'il y a de quoi a calculer */
cmp esi, eax
jge for2e

/* déroulement de boucle pour le premier seulement */
/* y += x[i-3] + x[i-2] + x[i-1] + x[i] */
paddd xmm0, [esi]

/* début de la boucle */
for2b:

/* place pour le prochain calcul */
add esi, 16
cmp esi, eax
jge for2e

/* y += x[i-3] + x[i-2] + x[i-1] + x[i] */
paddd xmm0, [esi]

/* debute une autre boucle */
jmp for2b

/* fin de la boucle */
for2e :

/*** 4) On additionne les sommes partielles ***/
push   esp
movd   [esp], xmm0
psrldq xmm0,  4
push   esp
movd   [esp], xmm0
psrldq xmm0,  4
push   esp
movd   [esp], xmm0
psrldq xmm0,  4
push   esp
movd   [esp], xmm0
add    ebx,   [esp]
add    esp,   4
add    ebx,   [esp]
add    esp,   4
```

```
add     ebx,    [esp]
add     esp,    4
add     ebx,    [esp]
add     esp,    4

/*** 5) On additionne les entiers restant ***/

/* calcul les entiers restant */
for3b :

/* est-ce que c'est terminé ? */
cmp esi, edx
jge for3e

/* effectue une addition */
add ebx, [esi]

/* change a l'entier suivant */
add esi, 4
jmp for3b

/* fin de la boucle */
for3e:

/*** 6) On place le résultat dans la variable y en mémoire ***/
mov [edi], ebx
```

Code 1-1 : Addition d'un vecteur d'entier en assembleur.

Voyant tout ce qu'implique la programmation des instructions data-parallèles en langage assembleur, il est aisé de comprendre pourquoi peu de gens l'apprécient et encore moins l'utilise. Elle reste néanmoins utilisée lorsqu'un gain de performance est obligatoire, et que les autres moyens plus simples ont échoués à l'atteinte des objectifs.

1.5.2 Les fonctions intrinsèques

Une autre avenue pour utiliser les instructions data-parallèles est l'utilisation d'un compilateur qui inclut des fonctions intrinsèques pour travailler avec les instructions data-parallèles. Une fonction intrinsèque est une fonction dans le langage C, dont l'implémentation est assurée par le compilateur, qui ne provient pas d'une librairie de code externe. Le compilateur en possède une connaissance approfondie et est en mesure de

l'optimiser. Ces fonctions sont pour l'essentiel un remplacement direct des instructions assembleurs. Par exemple, la fonction intrinsèque `_mm_add_pi32` en C est l'équivalent de l'instruction `PADDD` en assembleur. Il faut utiliser un type de donnée spécifique pour ces opérations. Ceci implique qu'il faut transférer les données de travail du type choisis (entier trente-deux bits par exemple) vers ce type. Cette opération correspond au transfert de données de la mémoire à un registre du processeur. Ensuite, l'appel de la fonction peut avoir lieu. Finalement, il faut transférer le résultat vers l'endroit désiré, représentant le déplacement des données du registre vers la mémoire.

Donc, ces fonctions aident à écrire le code, mais tout comme le langage assembleur, elles ont plusieurs défauts. Nous nommerons seulement l'utilisation d'un type de donné particulier et l'alignement de données en mémoire. Le Code 1-2 est le même que le Code 1-1, mais en langage C avec l'utilisation des fonctions intrinsèques.

```
/* Additionne les entiers du vecteur x et place le résultat dans la
 * variable y.
 */

/* 1) Assignation globales */

/* pointe sur la donnee de x ou l'on est rendu */
const unsigned int * px = x;

/* calcul la fin du tableau */
const unsigned int *const pxend = *(&x + 1);

/* y = 0; */
y = 0;

/* 2) On additionne, un à la fois, les entiers de 4 octets jusqu'à ce que
 *    le vecteur soit aligné à un 16 octets en mémoire
 */

/* tant qu'on n'est pas aligné */
while (reinterpret_cast<unsigned int>(px) & 0x0000000f)
{
    y += *px++;
}

/* 3) On additionne en block de 4 entiers à la fois */

/* initialise l'endroit des résultats partiels */
__m128i tmp1 = {0};

/* calcul le dernier endroit aligné du tableau */
const unsigned int *const pxenda =
    reinterpret_cast<unsigned int *>(
        reinterpret_cast<unsigned int>(pxend) & 0xfffffff0
    );

/* Tant qu'il y a de quoi a calculer en block */
while (px < pxenda)
{
    /* y += x[i-3] + x[i-2] + x[i-1] + x[i] */
    __m128i tmp2;
    tmp2 = _mm_setr_epi32(*px++, *px++, *px++, *px++);
    tmp1 = _mm_add_epi32(tmp1, tmp2);
}

/* 4) On additionne les sommes partielles */

for (size_t i = 0; i < 4; ++i)
{
    y += tmp1.m128i_u32[i];
}

/* 5) On additionne les entiers restant */
```

```
while (px < pxend)
{
    y += *px++;
}
```

Code 1-2 : Addition d'un vecteur d'entier en langage C en utilisant les fonctions intrinsèques.

1.5.3 Optimisation par le compilateur

La troisième option est d'opter pour un compilateur optimisant qui sait comment utiliser les instructions data-parallèles. Présentement il est aisé de trouver un compilateur qui est capable d'utiliser une partie des instructions data-parallèles. Le Code 1-3 montre la même opération qu'au Code 1-1 et au Code 1-2 qui serait optimisé par un compilateur optimisant.

```
/* Additionne les entiers du vecteur x et place le résultat dans la
 * variable y.
 */

y = 0;
for (int i = 0; i < X_SIZE; ++i)
{
    y += x[i];
}
```

Code 1-3 : Addition d'un vecteur d'entier en langage C.

Compilateur	Jeux d'instructions IUDM supportés
Microsoft	MMX, SSE, SSE2
Intel	MMX, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2
gcc	MMX, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2

Tableau 1-2 : Jeux d'instructions IUDM supportés par les compilateurs.

En plus de ne pas toujours supporter tous les jeux d'instructions IUDM disponibles comme le montre le Tableau 1-2, ils ne sont pas capables d'optimiser le code dans toutes les circonstances. Le document (Levicki, 2012) montre que le compilateur d'Intel est capable

d'optimiser le Code 1-4 mais pas le Code 1-5, deux codes qui sont presque identiques. La différence est en caractères gras dans le code.

```
for (short i = 0; i < SIZE; ++i)
{
    // corps de la boucle
}
```

Code 1-4 : Boucle optimisée par le compilateur d'Intel.

```
for (unsigned short i = 0; i < SIZE; ++i)
{
    // corps de la boucle
}
```

Code 1-5 : Boucle non optimisée par le compilateur d'Intel.

L'utilisation d'un compilateur optimisant qui inclus l'utilisation d'une partie des instructions data-parallèles est mieux que celle d'un compilateur qui ne les utilise pas, mais ceci n'est pas l'idéal et n'est parfois pas suffisant.

1.5.4 Besoin

Il est maintenant clair qu'un outil permettant au programmeur d'utiliser la puissance offerte par les instructions data-parallèles de manières simple et efficace est pertinent et nécessaire. Cet outil devrait utiliser les instructions data-parallèles partout où c'est pertinent. Le programmeur ne devrait pas avoir à fournir d'effort supplémentaire.

Afin de résoudre cette problématique, nous proposons un ensemble d'instructions data-parallèles qui seront intégrées au langage psC. Le paradigme supporté par le langage psC étant parallèle, en non séquentiel, fait de ce langage de programmation de FPGA le candidat idéal pour y intégrer des instructions data-parallèles.

Intel offre une implémentation de valarray qui utilise les instructions SSE. L'objectif étant d'évaluer la faisabilité d'utiliser un langage pour programmation matérielle pour générer le code pour FPGA, cette approche n'a pas été investiguée.

2

ALGORITHMES

Les logiciels, où applications, sont des ensembles d'algorithmes ordonnés de manière à effectuer une ou plusieurs tâches précises. Pour être en mesure d'améliorer la performance, il est nécessaire de comprendre et classifier ces algorithmes. Ceci, afin de mieux définir les instructions data-parallèles pertinentes et utiles.

2.1 Classification des algorithmes

Comme vu en 1.2, il existe plusieurs architectures matérielles. L'existence de ces architectures matérielles provient du fait que les algorithmes sont différents, et que certains sont plus simple à implémenté sur certaines architectures.

On présente ici une classification des algorithmes qui se base sur la dépendance entre les données. Ceci nous permet de déterminé ceux qu'il est possible paralléliser sur l'architecture PC et de mieux les connaître. Pour définir le comportement des algorithmes, on utilise les termes bloc d'instructions et bloc de données. Un bloc d'instructions est un ensemble d'instructions ordonnées d'une manière précise pour effectuer une tâche. Un bloc d'instructions ne peut pas être divisé. Il doit être vu comme monolithique. Un bloc de

données est constitué de plusieurs données, toutes traitées simultanément par un bloc d'instructions.

2.1.1 Algorithmes synchrones

Les algorithmes synchrones peuvent se composer d'un ou plusieurs blocs d'instructions. Cependant, il n'y a qu'un seul chemin d'exécution possible. De plus, il ne doit pas y avoir de corrélation entre les données pour que chacun des blocs de données puissent être traités indépendamment des autres. Lorsque l'architecture matérielle le permet, les blocs de données peuvent être traités de manière parallèle. Ces algorithmes sont des candidats parfaits pour un traitement via une architecture IUDM car ils peuvent être traités avec une séquence de blocs d'instructions.

Voici un exemple algorithme synchrone : considérant `A`, `B` et `C` comme des vecteurs, il est possible d'exécuter l'algorithme montré au Code 2-1 en utilisant `i` comme index de bloc de données. La Figure 2-1 l'illustre.

```
C[i] = A[i] + B[i];
```

Code 2-1 : Algorithme synchrone.

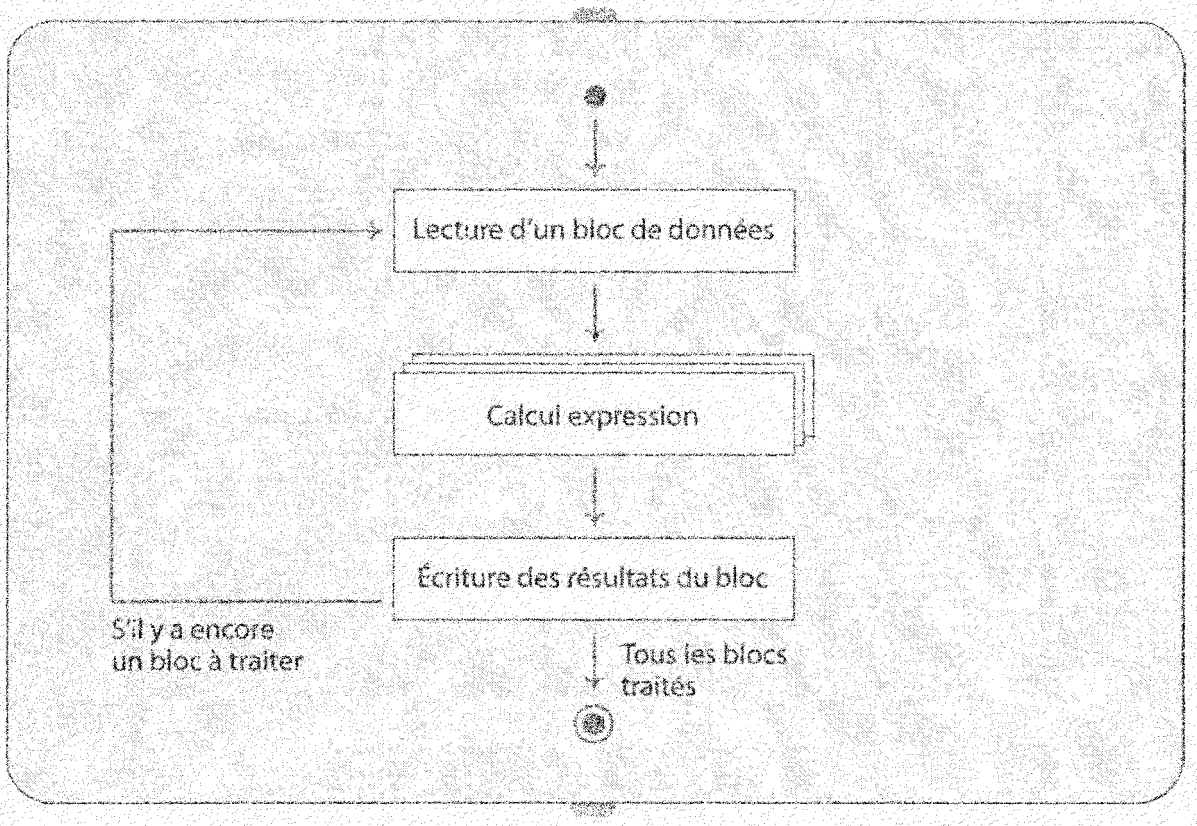

Figure 2-1: Algorithme synchrone.

2.1.2 Algorithmes légèrement synchrones

Les algorithmes légèrement synchrones sont similaires aux algorithmes synchrones à l'exception d'un seul point. Le chemin d'exécution entre les blocs d'instructions varie en fonction du bloc de données traité. Le meilleur moyen pour expliquer est via un exemple. En reprenant l'exemple précédant et en changeant le calcul, nous obtenons un algorithme légèrement synchrone (voir le Code 2-2). Il y a plusieurs chemins différents entre les blocs d'instructions en fonction de la comparaison `A[i] < 0`.

```
C[i] = (A[i] < 0) ? (A[i] | B[i]) : (A[i] & B[i]);
```

Code 2-2 : Algorithme légèrement synchrone.

Contrairement aux les algorithmes synchrones, il est impossible de les traiter en une seule séquence de blocs d'instructions sous une architecture IUDM. Il est nécessaire d'effectuer des opérations à priori ou à postériori supplémentaires. Les opérations à priori servent à détecter lequel des chemins d'exécutions doit être emprunté, pour ensuite effectuer les blocs d'instructions s'y trouvant (voir la Figure 2-2). Cette approche exige que le chemin d'exécution convienne à toutes les données d'un bloc de données. Pour le cas à postériori, tous les chemins d'exécutions sont fait et on choisit le bon résultat pour chaque donné du bloc de données par la suite (voir la Figure 2-3). Ceci implique d'effectuer toujours tous les blocs d'instructions, qu'il soit nécessaire ou pas au résultat final. Enfin, il est possible d'effectuer les deux techniques en simultané (voir la Figure 2-4). Aucune de ces solutions n'est optimale car il y a du travail supplémentaire qui est requis dans tous les cas.

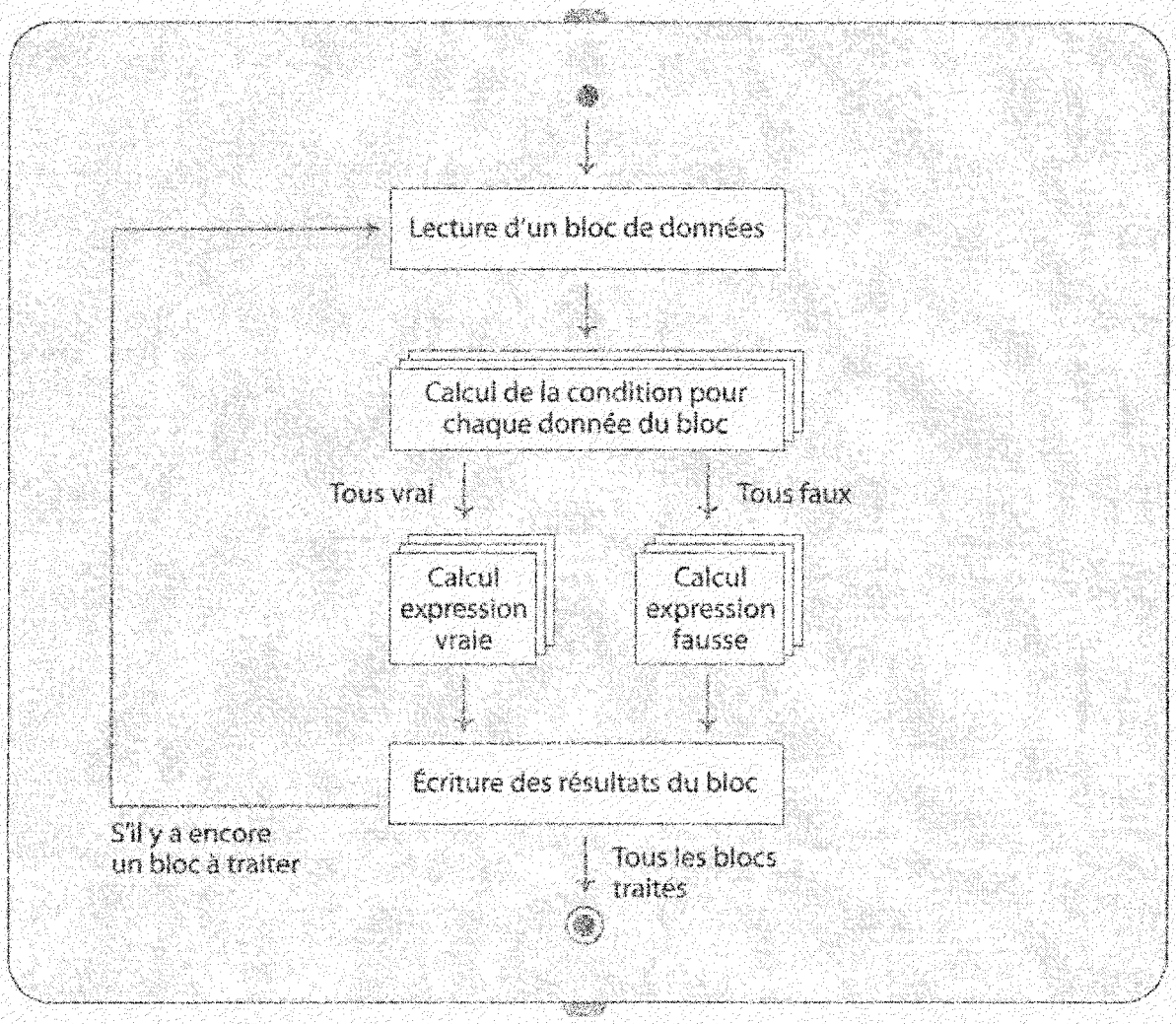

Figure 2-2: Algorithme légèrement synchrone - opération à priori.

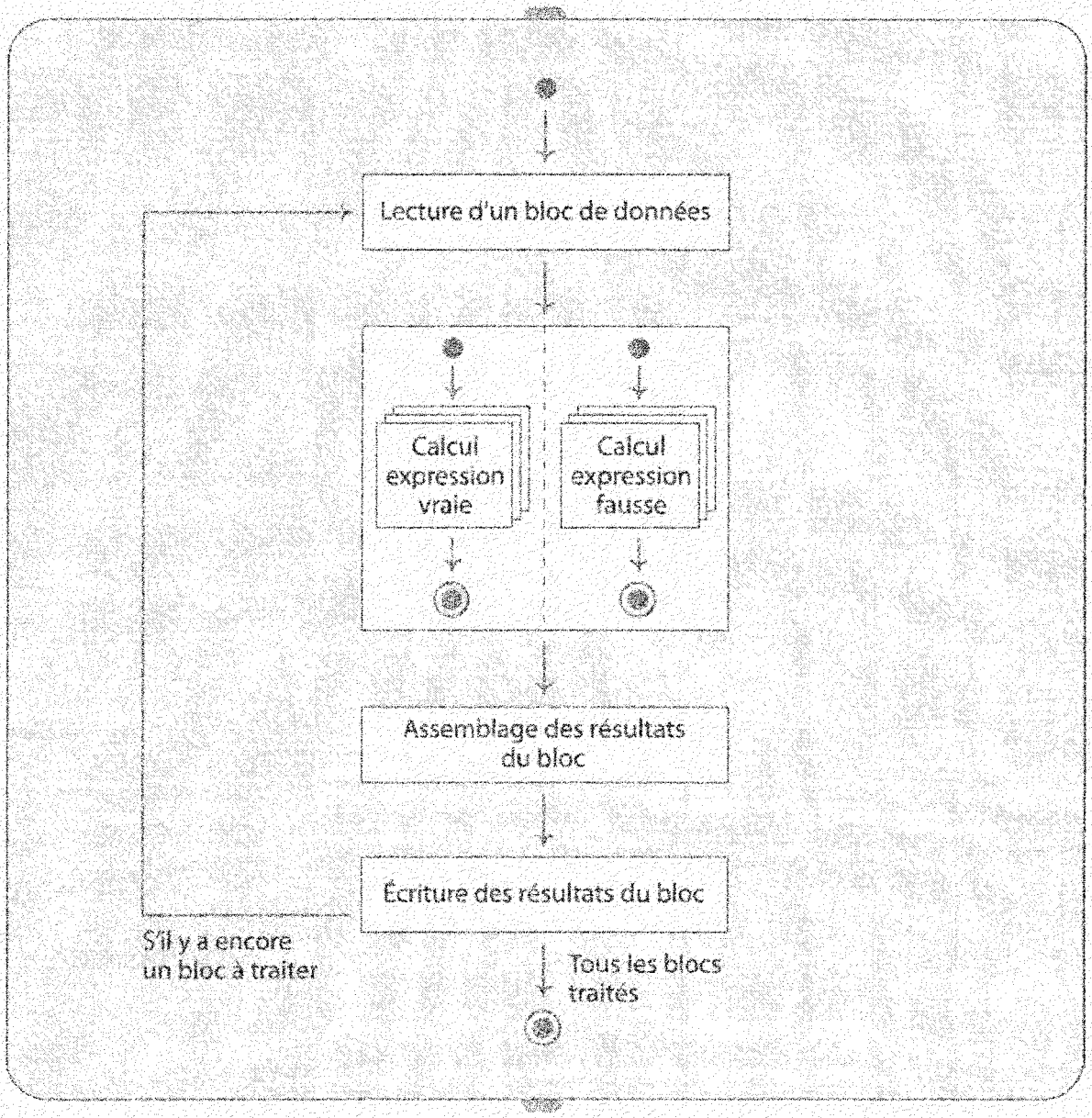

Figure 2-3: Algorithme légèrement synchrone - opération à postériori.

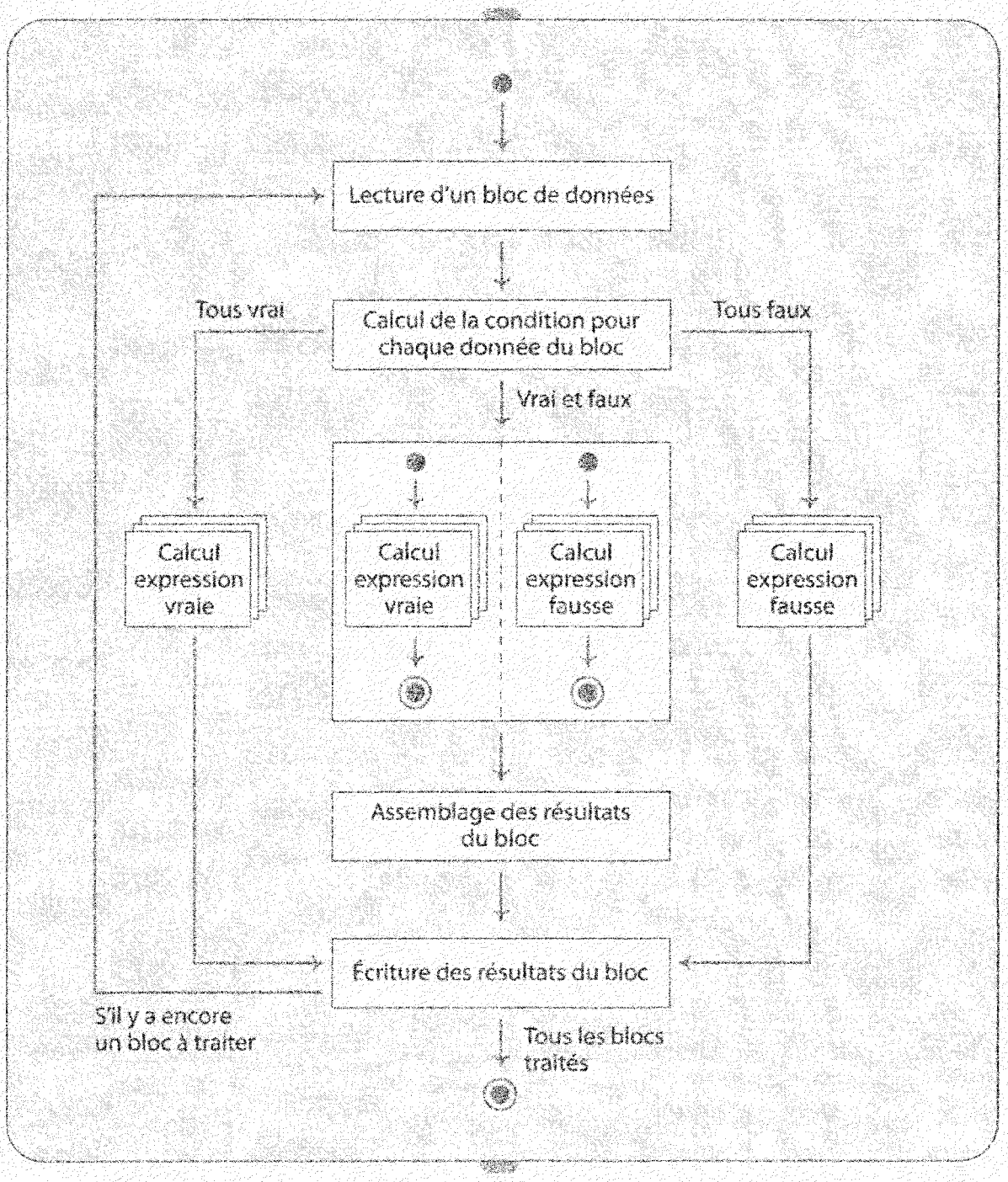

Figure 2-4: Algorithme légèrement synchrone - opérations à priori et à postériori.

Ces algorithmes sont de bons candidats pour les architectures IMDM. Sur ces architectures, chaque bloc peut être est effectué sur une unité de traitement différente.

2.1.3 Algorithmes asynchrones

Les algorithmes asynchrones ont comme caractéristique que leurs données ne sont pas indépendantes. Il faut donc avoir une connaissance particulière de l'algorithme pour synchroniser l'accès aux données et paralléliser les parties qui peuvent l'être.

Historiquement, tous les programmes sont traités comme des algorithmes asynchrones car les architectures matérielles n'avaient pas la possibilité d'améliorer les performances de manière automatique. Généralement, il n'y a pas de parallélisme mis de l'avant et un seul chemin d'exécution est actif à un temps précis. Ce sont les algorithmes que les programmeurs trouvent les plus simples à comprendre, concevoir, implanter et maintenir (voir la Figure 2-5).

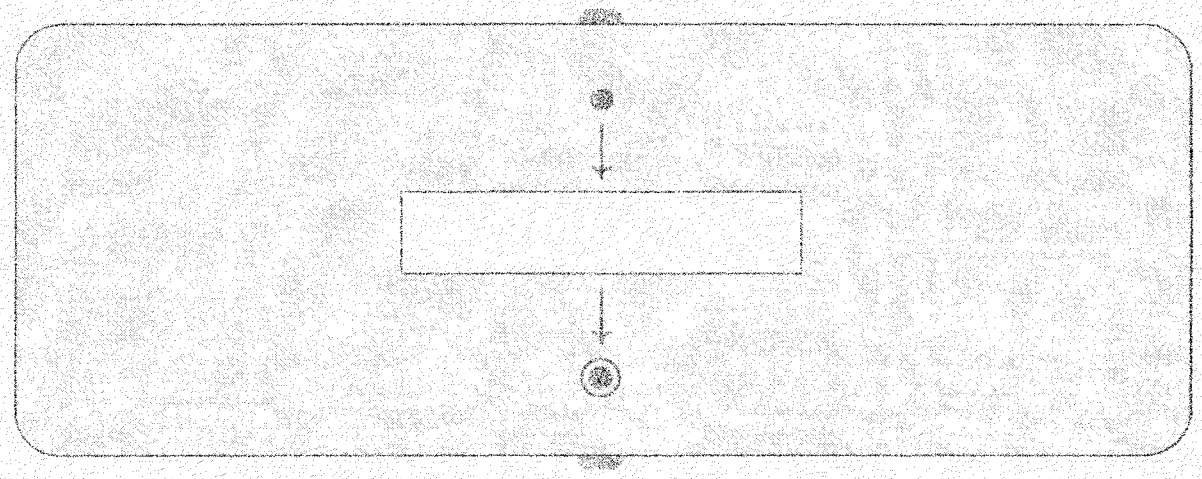

Figure 2-5: Algorithme asynchrone.

Ils sont de bons candidats pour les architectures IUDU.

2.2 Instructions souhaitables

Pour être viables, les langages inclus dans le paradigme de programmation data-parallèles doivent supporter les instructions traditionnelles disponibles dans le paradigme de programmation impératif, mais ils doivent en plus offrir plusieurs instructions qui ne sont pas disponibles dans les autres paradigmes de programmations. Ces instructions sont des instructions qui traitent des données en parallèles. Ces instructions sont décrites dans les sections suivantes.

Pour simplifier l'écriture, nous utiliserons la notion de groupe de données. Un groupe de données est soit un vecteur ou une matrice contenant les données. Toutes les données d'un groupe de donnée sont du même type.

2.2.1 Le type « index »

Un nouveau type `index` est utilisé pour les instructions data parallèles proposées. Les variables de type index permettent d'identifier des plages de valeurs. Par exemple, pour indiquer d'effectuer une opération sur tous les éléments de la première dimension de tableaux, un index est utilisé, comme le montre le Code 2-3.

```
/* Exemple d'utilisation du type index.
 */

/* déclaration */
int32 M[10];
int32 N[10];
int32 V[10];

index i;

/* Effectuer ici du travail pour assigner des valeurs. */

/* calcul qui utilise l'index */
V[i] = M[i] + N[i];
```

Code 2-3 : Utilisation du type index.

2.2.2 Instructions d'affectations

Une instruction d'affectation permet d'assigner un groupe de données à un endroit précis d'un autre groupe de données. Par exemple, envoyer les données d'un vecteur dans une rangée d'une matrice. Le Code 2-4 montre comment cette instruction pourrait être effectuée dans un langage hypothétique. De plus, la combinaison avec les instructions de sectionnement permet d'extraire un sous-groupe de données à partir d'un groupe de données et de les insérer dans un autre groupe de données.

```
/* Déclaration du vecteur de taille T et de la matrice de taille T x T,
 * tous deux de type entier à 32 bit.
 */
int32 V[T];
int32 M[T][T];

/* Effectuer ici du travail pour assigner des valeurs. */

/* Effectue des communications. */
index i;
M[2][i] = V[i];  // rangée
M[i][2] = V[i];  // colonne
M[i][i] = V[i];  // diagonale
```

Code 2-4 : Instruction de communication.

2.2.3 Instructions de sectionnement

Du moment que l'on parle d'un groupe de donnée, il devient nécessaire de pouvoir référer à un sous-ensemble du groupe de données comme s'il était lui-même un groupe de données. Cette capacité d'extraire une partie du groupe de donnée se nomme sectionnement. Par exemple, si une fonction qui effectue un travail avec un vecteur et que nous avons une matrice de deux dimensions, il faut être en mesure d'exécuter la fonction sur une colonne de la matrice. Le Code 2-5 montre comment ceci pourrait être fait dans un langage hypothétique.

```
/* déclaration d'une matrice A de taille 4x8 et de type entier à 32 bit.
 */
int32 A[4][8];

/* Effectuer ici du travail avec A pour lui assigner des valeurs. */

/* Indique quel est le sectionnement et récupère l'élément (Colonne 2),
 * via un tableau de référence pour ne pas avoir de copie.
 */
index i;
int32 & SGA[8] = A[2][i];

/* Utiliser le sous-groupe comme étant un vecteur de taille 8 */
```

Code 2-5 : Instruction de sectionnement.

2.2.4 Instructions conditionnelles

Certains algorithmes nécessitent qu'un calcul soit appliqué à seulement une partie des éléments d'un groupe de données. Lorsque ceci se produit, il est nécessaire d'effectuer un test sur les données elles-mêmes pour décider si l'opération les affecte. Cette capacité se nomme instruction conditionnelle. Par exemple, des instructions permettant d'effectuer l'inverse des éléments différents de zéro. Le Code 2-6 montre comment ceci pourrait être fait dans un langage hypothétique.

```
/* déclaration d'un vecteur V de taille T et de type entier à 32 bit. */
int32 V[T];

/* Effectuer ici du travail avec V pour lui assigner des valeurs. */

/* Effectue l'inverse de tous les éléments non nul. */
index i;
V[i] = (V[i] == 0) ? V[i] : (1 / V[i]);
```

Code 2-6 : Instruction conditionnelle.

2.2.5 Instructions de réductions

Les instructions de réduction consistent à diminuer le nombre de dimensions d'un groupe de données. Par exemple, passer d'une matrice à cinq dimensions à une matrice à quatre dimensions. La réduction est dite complète lorsque le groupe devient un scalaire. Sinon elle est qualifiée de partielle. Il existe plusieurs techniques qui peuvent être utilisés pour effectuer cette réduction. Par exemple, l'élément du groupe de dimension moindre pourrait correspondre à la somme de tous les éléments correspondant dans le groupe de dimensions plus importante. Ce pourrait également être le plus petit élément, le plus grand, la variance, la covariance, l'écart type, etc. Le Code 2-7 montre comment une réduction partielle utilisant la valeur minimale pourrait être effectuée dans un langage hypothétique.

```
/* déclaration d'une matrice M de taille 4x8 et de type entier à 32 bit.
 */
int32 M[4][8];

/* Effectuer ici du travail avec M pour lui assigner des valeurs. */

/* Effectue la réduction vers un vecteur de taille 4. */
int32 V[4];
index i;
V = min(M[i]);
```

Code 2-7 : Instruction de réduction.

2.2.6 Instructions vectorielles

Une instruction vectorielle est une instruction qui s'exécute sur chacune des données d'un groupe de données. Il doit être transparent pour l'usager qu'il utilise une instruction scalaire ou vectorielle. L'instruction scalaire traitera une donnée, l'instruction vectorielle traitera un groupe de données. Par exemple, si un langage offre la possibilité d'écrire le Code 2-8 où `A`, `B` et `C` sont des scalaires, le même code devrait pouvoir être utilisé lorsque `A`, `B` et `C` sont des groupes de données de même taille.

```
A = B + C
```

Code 2-8 : Instruction vectorielle.

L'utilisation d'un langage impératif, contraint à l'utilisation d'une boucle (voir le Code 2-8). Il va sans dire que l'optimisation du code machine généré, qu'il soit axé sur le temps d'exécution ou la taille du code, n'est pas garantie.

```
for (int i = 0; i < DATA_SIZE; ++i)
{
    A[i] = B[i] + C[i];
}
```

Code 2-9 : Instruction vectorielle en langage C.

2.3 Solution selon valarray

Valarray est le nom donné à un ensemble d'objets et fonctions faisant partie du standard C++. Il offre la possibilité d'effectuer plusieurs des opérations souhaitables directement en langage C++. Cependant, comme le montre le document (AFNOR, 2000), valarray n'est pas parfait du point de vue de la syntaxe. De plus il ne supporte pas les tableaux à plusieurs dimensions naturellement. Il est néanmoins utile de regarder comment

il adresse chacune des instructions souhaitées pour la création, l'affectation et l'accès aux données, afin de s'en inspirer dans le cadre du travail.

2.3.1 Construction

Il est possible de créer un objet valarray de plusieurs manières, comme le démontre le Code 2-10. Il est intéressant de noter la création par copie (en gras) qui utilise une syntaxe vectorielle simple.

```
/* Création d'un vecteur de int. */
valarray<int> v1;

/* Création d'un vecteur de taille précise (8). */
valarray<int> v2(8);

/* Création d'un vecteur de taille précise (8), dont les éléments sont
 * tous initialisés à une même valeur (3).
 */
valarray<int> v3(3, 8);

/* Création d'un vecteur de taille précise (8), dont les éléments sont
 * initialisés à des valeurs données.
 */
static const int initval[] = {0, 1, 2, 3, 4, 5, 6, 7};
valarray<int> v4(initval, 8);

/* Création d'une copie d'un vecteur */
valarray<int> v5(v4);
valarray<int> v6 = v4;
```

Code 2-10 : Valarray – construction.

2.3.2 Instructions d'affectation

La première manière d'affecter des valeurs à un valarray est lors de sa création. Le Code 2-10 montre comment affecter tous les indices à la même valeur et à des valeurs différentes dans cette situation.

En dehors de la création, il est possible d'affecter un scalaire à chacun des éléments via l'opérateur égal. Pour l'accès à un seul élément (un scalaire) en dehors de la création, l'opérateur crochet est utilisé. Le Code 2-11 montre comment les accès en lecture et en écriture fonctionnent.

```
/* Affectation de la valeur 'a' à tous les éléments d'un valarray 'v' */
v = a

/* Affectation de la valeur 'a' à un élément 'i' d'un valarray 'v' */
v[i] = a

/* Extraction d'une valeur 'a' d'un élément 'i' d'un valarray 'v' */
a = v[i];
```

Code 2-11 : Valarray – affectation et extraction d'un scalaire.

Enfin, il est possible d'affecter des groupes de données à des endroits précis en utilisant l'opérateur crochet et des objets explicitement fabriqués pour ce faire. Ces cas seront couverts dans la section 2.3.3.

2.3.3 Instructions de sectionnement

Valarray offre deux objets pour effectuer le sectionnement; `slice` et `gslice`. Le premier est utilisé pour transformer un index à deux dimensions vers l'index unique du valarray. Le second, plus complexe, est utilisé pour transformer un index à dimensions plus élevé vers l'index unique de valarray. L'application de ces objets à un valarray donne un objet intermédiaire qui peut être utilisé pour créer un nouvel objet valarray ou utilisé à la place d'un valarray dans certaines instructions. Le Code 2-12 montre des exemples.

```
/* Multiplie les éléments 1, 3 et 5 du valarray 'v' par 10 */
v[std::slice(1,3,2)] *= std::valarray<int>(10, 3);

/* Affecte 99 aux éléments 0, 3, 6 du valarray 'v' */
v[std::slice(0,3,3)] = 99;

/* Affecte les éléments 0, 3 et 6 du valarray 'v2' et les affectent aux
 * éléments 0, 4, 8 du valarray 'v2'
 */
v1[std::slice(0,3,3)] = v2[std::slice(0,3,4)] = 99;
```

Code 2-12 : Valarray – Instructions de sectionnement.

2.3.4 Instructions conditionnelles

L'instruction conditionnelle n'existe pas avec valarray. Cependant, il est possible d'obtenir un résultat similaire en utilisant un opérateur de comparaison avec un valarray. L'objet fabriqué par cette opération est un `valarray<bool>` qui contient `true` pour chaque élément pour lequel la condition est vraie. Appliqué au valarray via l'opérateur crochet, il sélectionne les éléments pour lesquels il est à `true`. Il est possible d'inverser un `valarray<bool>` avec l'opérateur de négation logique. Enfin, une opération peut être effectuée à un sous-groupe et affecter à un endroit précis. Le Code 2-13 en montre l'utilisation.

```
/* Effectue l'opération v = v > 25 ? 25 : -v; avec valarray */
valarray<int> v(50);
for (int i = 0; i < v.size(); ++i)
  v[i] = i;

/* Sélectionne les éléments plus grand que 25 du valarray 'v' */
valarray<bool> greater25 = v > 25;

/* Affecte 25 à tous les éléments de 'v' qui sont plus grand que 25 */
v[greater25] = 25;

/* Affecte '-v' aux autres éléments de 'v' */
v[!greater25] = -std::valarray<int>(v[!greater25]);
```

Code 2-13 : Valarray – Instructions conditionnelles.

2.3.5 Instructions de réductions

Valarray n'offre qu'une sélection limitée de réductions listé au Tableau 2-1. Pour toute autre instruction de réduction, il est nécessaire que le programmeur fasse preuve d'originalité dans l'utilisation des `slice` et de la fonction `apply`. La somme de travail pour le programmeur serait moindre si plus d'opérations de réductions seraient offertes.

Opération	Description
size	La taille du valarray.
sum	La somme des éléments du valarray.
min	L'élément le plus petit contenu dans le valarray.
max	L'élément le plus grand contenu dans le valarray.

Tableau 2-1 : Valarray – instructions de réduction.

2.3.6 Instructions vectorielle

Valarray supporte certaines instructions vectorielles via des fonctions spécialisées prenant un valarray en paramètre. Le Tableau 2-2 indique toutes les opérations vectorielles supportées. Chacune de ces opérations a comme paramètre un valarray. Lorsqu'elle nécessite plus d'un paramètre, il peut être soit un scalaire qui est réutilisé pour chaque élément, soit un valarray de taille identique dont chaque élément est utilisé avec l'élément réciproque de l'autre valarray. De plus, la fonction membre `apply` permet d'appliquer une fonction fournie par l'usager à chaque élément du valarray.

```
/* Additionne 3 à tous les éléments d'un valarray 'v' */
v += 3;

/* Divise par 5 tous les éléments d'un valarray 'v' */
v /= 5;

/* Effectue la puissance d'un valarray 'v' par 2 */
v = pow(v, 2);

/* Effectue la puissance d'un valarray 'v' par un autre valarray 'p' */
v = pow(v, p);

/* Applique la fonction fn à chaque éléments du valarray 'v' */
v = v.apply(fn);
```

Code 2-14 : instructions vectorielles de valarray.

Fonction	Description
abs	Valeur absolue
asin	Arc sinus
acos	Arc cosinus
atan	Arc tangente
atan2	Arc tangente à deux arguments
cos	Cosinus
cosh	Cosinus hyperbolique
exp	Exponentiel naturel
log	Logarithme naturel
log10	Logarithme à base 10
pow	Puissance
sin	Sinus
sinh	Sinus hyperbolique
sqrt	Racine carrée
tan	Tangente
tanh	Tangente hyperbolique
+ (unaire)	Ne change pas la valeur
- (unaire)	Inverse le signe
+ (binaire)	Addition
- (binaire)	Soustraction
~	Inverse les bits qui composent le nombre
*	Multiplication
/	Division
%	Reste de la division (modulo)
^	Ou exclusif binaire
&	Et binaire
|	Ou inclusif binaire
<<	Décalage à gauche des bits de chaque élément
>>	Décalage à droite des bits de chaque élément
==	Égalité
!=	Non égalité, différent
<	Plus petit
>	Plus grand
<=	Plus petit ou égal
>=	Plus grand ou égal
||	Ou logique
&&	Et logique

Tableau 2-2 : Valarray – instructions vectorielles.

2.4 Solution selon Blitz++

Blitz++ est également une librairie conçue pour le langage C++. Comme valarray, elle ajoute la possibilité d'effectuer les opérations souhaitables. Les opérations sont seulement au niveau de la syntaxe, et aucune optimisation n'est effectuée pour prendre en charge les possibilités offertes par le matériel, sauf ce que le compilateur réussira à optimiser. À l'instar de valarray, son étude est une bonne source d'inspiration pour la solution proposée. La classe sur laquelle est basée la librairie se nomme `Array`.

2.4.1 Construction

La création d'un objet `Array` s'effectue principalement via l'une des deux formes démontrées au Code 2-15. Via la seconde manière, celle utilisant l'objet `Range`, il est possible de créer des objets dont les index ne débutent pas à zéro. Il est également possible de créer des objets dont la structure mémoire est ordonnée comme en Fortran, où la colonne est majeure, au lieu de la manière C, où rangée est majeur. Un `Array` ne peut pas recevoir de valeur initialement, sauf si c'est le résultat d'une opération qui produit un `Array`.

```
/* Création d'une matrice tridimensionnelle de taille NxNxN de float */
blitz::Array<float,3> A(N,N,N);

/* Création d'une matrice 5x5 dont les indices sont -2 à 2 et 1 à 5. */
blitz::Array<float,2> A(blitz::Range(-2,2), blitz::Range(1, 5));
```

Code 2-15 : Blitz++ – construction.

2.4.2 Affectation et extraction de valeurs

La première manière d'affecter des valeurs à un `Array` de Blitz++ est d'utiliser l'opérateur « virgule » pour assigner directement une matrice. Le Code 2-16 montre l'utilisation de cette technique. Il est possible d'affecter la valeur d'un scalaire à tous les

éléments via l'opérateur égal. Pour l'accès à un seul élément (un scalaire), l'opérateur parenthèse (et non l'opérateur crochet comme avec valarray et les tableaux) est utilisé. Le Code 2-16 montre les accès en lecture et en écriture.

```
/* Affectation de la valeur 'a' à l'array 'v' */
v = a;

/* Affectation de la matrice identité à 'v' qui est de taille 3x3 */
v = 1, 0, 0,
    0, 1, 0,
    0, 0, 1;

/* Extraction d'une valeur à la position 'i' 'j' d'un Array 'v' et
 * assignation à la position 'k' 'l'
 */
v(k, l) = v(i, j);
```

Code 2-16 : Blitz++ – affectation et extraction d'un scalaire.

Avec Blitz++ il existe un élément nommé « placeholder ». On peut le traduire en français par index. C'est un élément qui a comme valeur sa position dans la dimension où il se trouve. Pour mieux comprendre, voir le Code 2-17 ou la documentation de Blitz++. Dans le code qui suit, `firstIndex` est un type prédéfini permettant de déclarer un index pour la première dimension. Il existe des types similaires pour les autres dimensions : `secondIndex`, `thirdIntex`, ...

```
/* Affectation des valeurs 0, 1, 2 et 3 à un Array 'v' d'une seule
 * dimention de taille 4.
 */

/* déclaration */
v(4);

/* Assignation manuelle */
v = 0, 1, 2, 3;

/* L'équivalent avec un index */
firstIndex i;
v = i;
```

Code 2-17 : Blitz++ - index.

2.4.3 Instructions d'affectations

Blitz++ offre les instructions d'affectations via l'opérateur parenthèse (*()*). Cet opérateur permet de choisir un sous-groupe d'un *Array* et d'y extraire ou affecter des valeurs. La fonction *extractComponent* permet également d'aller chercher un sous-ensemble.

2.4.4 Instructions de sectionnement

Blitz++ offre une manière simple d'effectuer le sectionnement. On utilise l'opérateur parenthèses comme pour l'extraction d'un scalaire, mais on remplace au moins un des scalaires par un objet *Range*. Le Code 2-18 montre un exemple.

```
/* Va chercher une partie d'un plan deux dimension dans un array 'v' de
 * trois dimension.
 * Éléments 2 à 5 de la première dimension
 * Éléments 4 à 8 de la seconde dimension
 * Plan situé à l'élément 3 de la troisième dimension.
 */
v(Range(2,5), Range(4,8), 3);
```

Code 2-18 : Instructions conditionnelles de Blitz++.

Blitz++ offre d'autres manières d'effectuer le sectionnement, mais ces manières sont toutes plus complexes que celle montré ci-haut et elle les inclut toutes. Nous ne discuterons donc pas des autres méthodes.

2.4.5 Instructions conditionnelles

L'instruction conditionnelle est fournie via la fonction `where`. Cette fonction demande trois paramètres. Le premier est la condition. Le second est l'action à effectuer si la condition est vraie. Le troisième est l'action dans le cas où la condition est fausse. Le Code 2-19 en montre l'utilisation.

```
/* Effectue l'opération v = v > 25 ? 25 : -v; avec Blitz++ */
v = where(v > 25, 25, -v));
```

Code 2-19 : Blitz++ – Instruction conditionnelle.

2.4.6 Instructions de réductions

Blitz++ offre une sélection de réductions plus importante que celle offerte par valarray. Le Tableau 2-3 en fait la liste. Pour toute autre instruction de réduction, il est nécessaire que le programmeur développe la fonctionnalité avec les outils fournis.

Opération	Description
size	Quantité d'éléments
count	Le nombre de fois que l'expression passée est vraie
sum	Somme des éléments
product	Produit des éléments
mean	Moyenne des éléments
min	Élément le plus petit
max	Élément le plus grand
minIndex	L'index de l'élément le plus petit
maxIndex	L'index de l'élément le plus grand
any	Vrai si l'expression passée est vraie pour au moins un élément
all	Vrai si l'expression passée est vraie pour tous les éléments
first	Le premier élément pour lequel l'expression passée est vraie
last	Le dernier élément pour lequel l'expression passée est vraie
extent	Taille de la dimension
dimensions	Nombre de dimension de la structure
stride	Pas de la dimension

Tableau 2-3 : Blitz++ – Instructions de réduction.

2.4.7 Instructions vectorielles

Blitz++ supporte certaines instructions vectorielles via des fonctions spécialisées prenant un `Array` en paramètre. Le Tableau 2-4 indique une partie des opérations vectorielles supportées. Toutes les opérations du Tableau 2-2 sont également supportées. Lorsqu'elle nécessite plus d'un paramètre, il peut être soit un scalaire qui est réutiliser pour chaque élément du `Array`, soit un `Array` de taille identique dont chaque élément est utilisé avec l'élément correspondant du l'autre `Array`, soit un index. Si l'opération désirée n'est pas fournie, Blitz++ offre plusieurs macros qui permettent de transformer une fonction qui accepte des scalaires en une fonction qui accepte les paramètres mentionnés ci-haut.

De plus, des fonctions pour traiter des dérivées centrales, dérivées avant, dérivées arrière, Laplaciens, Gradians, Jacobiens, Curl, Divergences et dérivées mixtes sont offertes. Une panoplie de possibilités sur les nombres semi-aléatoires est également disponible. Il est

possible de choisir la précision des nombres, le germe et le type de génération (Uniforme, Normale, Exponentielle, Gamma, et autres).

```
/* Additionne 3 à tous les éléments d'un array 'v' */
v += 3;

/* Divise par 5 tous les éléments d'un array 'v' */
v /= 5;

/* Effectue l'arrondissement supperieur pour tous les éléments
 * d'un array 'v'
 */
v = ceil(v);
```

Code 2-20 : Blitz++ – instruction vectorielle.

Fonction	Description
convolve	Calcul la convolution une dimension
arg	Argument d'un nombre complexe
ceil	Arrondis à l'entier supérieur
floor	Arrondis à l'entier inférieur
cexp	Exponentielle
conj	Conjugué d'un nombre complexe
cbrt	Racine cubique
expm1	Exponentielle moins un
erf	Fonction d'erreur
erfc	Fonction d'erreur complémentaire
ilogb	Exposant non biaisé
itrunc	Arrondissement à l'entier envers zéro
j0	Bessel, premier type, ordre zéro
j1	Bessel, premier type, ordre un
lgamma	Logarithme naturel de la fonction gamma
logb	Exposant non baisé de l'exposant
logp1	Logarithme plus un
nearest	Arrondis à l'entier le plus près
rsqrt	Racine carré réciproque
y0	Bessel, second type, ordre zéro
y1	Bessel, second type, ordre un
remainder	Reste de la division
hypot	Hypoténuse
scalb	Calcul de « x + 2 * y »

Tableau 2-4 : Blitz++ – liste partielle des instructions vectorielles.

2.5 Solution proposé

Les solutions étudiées qui sont des librairies C ou C++ laissent au compilateur le soin d'effectuer le parallélisme et d'utiliser les instructions data-parallèles du processeur. . Pour les autres solutions étudiées (HPF, C//, C*, Cilk, IDOLE, L, PARALLAXIS-III, SVL, ZPL, ...), soit qu'elles ne sont pas disponibles pour l'architecture PC, soit qu'elles n'offrent pas les opérations data-parallèles qui utilisent les instructions data-parallèles.

Il est donc nécessaire d'avoir quelque chose de nouveau pour répondre à ce critère. Notre solution est basée sur le langage psC, dont la syntaxe est similaire à celle du C. Il peut être utilisé sous plusieurs architectures logicielles incluant celles d'Intel et matérielles, les FPGA. Il s'agit d'un langage 100% parallèle, ce qui facilite la programmation d'algorithmes data-parallèles.

Cependant, la syntaxe actuelle ne supporte pas les opérations data-parallèles. Une partie importante de ce travail de recherche a consisté à définir et intégrer au compilateur psC une partie des éléments de syntaxe requis pour les instructions data-parallèles. Il a été utile de s'inspirer de valarray et Blitz++ pour définir ces opérations pour le langage.

2.5.1 Construction

Pour rester le plus simple possible, tout en conservant une syntaxe similaire à celle du langage C, la déclaration d'une structure data-parallèle est simplement la déclaration d'un tableau. La raison est qu'il est naturel pour les humains de se représenter les structures data-parallèles comme des tableaux. Lors de la création, il est possible de créer un tableau avec ou sans initialisation. La syntaxe de la déclaration est identique à celle d'un tableau C.

Voir le Code 2-21 pour des exemples de créations sans initialisations. S'il y a initialisation, la déclaration est suivie d'un signe égal et d'un choix d'initialiseur parmi:

- Scalaire: Un seul élément du type du tableau est présent. Tous les éléments du tableau sont alors initialisés avec cet élément. Voir le Code 2-22 pour un exemple.
- Tableau de taille identique : Chaque élément du tableau prend la valeur correspondante dans le tableau d'initialisation. Voir le Code 2-23 pour des exemples.
- Tableau d'initialisation plus grand : Chaque élément du tableau prend la valeur correspondante dans le tableau d'initialisation. Les éléments de surplus sont tout simplement ignorés. Voir le Code 2-24 pour un exemple.
- Tableau d'initialisation plus petit : Chaque élément du tableau prend la valeur correspondante dans le tableau d'initialisation tant qu'il y en a. Les éléments n'ayant pas d'éléments correspondants prennent la valeur par défaut du type du tableau. Voir le Code 2-25 pour un exemple.

```
/* tableau de 8 éléments de type int */
int s1[8];

/* tableau de 8x3 éléments de type double */
double s2[8][3];

/* tableau de 5x6x7 éléments de type byte */
byte s3[5][6][7];
```

Code 2-21 : psC – déclaration sans initialisation.

```
/* tableau de 10 éléments de type int initialisé à la valeur 2
 * s4 = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
 */
int s4[10] = 2;
```

Code 2-22 : psC – Déclaration avec initialisation via un scalaire.

```
/* tableau initialisés avec un autre tableau de même type et taille
 * s4 = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
 * s5 = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
 */
int s5[10] = s4;
```

Code 2-23 : psC – Déclaration avec initialisation via un tableau de même taille.

```
/* tableau initialisé avec un autre tableau de même type mais plus grand
 * s4 = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
 * s9 = {2, 2}
 */
int s9[2] = s4;
```

Code 2-24 : psC – Déclaration avec initialisation via un tableau plus grand.

```
/* tableau initialisé avec un autre tableau de même type mais plus petit
 * s4  = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
 * s10 = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
 */
int s10[20] = s4;
```

Code 2-25 : psC – Déclaration avec initialisation via un tableau plus petit.

2.5.2 Affectation et extraction de valeurs

Pour conserver un langage similaire au C, l'extraction d'une valeur sera la même que celle utilisé pour les tableaux du langage C. Le Code 2-26 montre un exemple de l'extraction d'une valeur.

Il est souvent nécessaire d'obtenir plus d'une valeur. Dans ces cas, le langage C ne fournis pas d'aide car il ne supporte pas cette fonctionnalité. Une approche similaire à celle de Blitz++ est donc utilisée. Tel que présenté au Code 2-27, une liste de valeurs, séparé par

des virgules, est utilisée pour extraire plusieurs valeurs simultanément. Une paire de crochet vide indique que toutes les valeurs sont demandées.

L'affectation utilise les indices de la même manière que l'extraction. Le Code 2-28 montre un exemple.

```
/* Extraction d'une valeur, à l'index 4
 * t1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
 * a  = 4
 */
int a = t1[4];

/* t2 = {00, 01, 02, 03, 04
 *       10, 11, 12, 13, 14
 *       20, 21, 22, 23, 24
 *       30, 31, 32, 33, 34
 *       40, 41, 42, 43, 44
 *       50, 51, 52, 53, 54}
 * b  = 33
 */
int b = t2[3][3];
```

Code 2-26 : psC – Extraction d'un scalaire.

```
/* Extraction des valeurs aux indices 2, 4 et 6
 * t1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
 * r1 = {2, 4, 6}
 */
int r1[] = t1{2, 4, 6};

/* Extraction des valeurs aux rangés 2 et 4, colonnes 0 et 3
 * t2 = {00, 01, 02, 03, 04
 *       10, 11, 12, 13, 14
 *       20, 21, 22, 23, 24
 *       30, 31, 32, 33, 34
 *       40, 41, 42, 43, 44
 *       50, 51, 52, 53, 54}
 * r2 = {20, 23
 *       40, 43}
 */
int r2[2][2] = t2[2, 4][0, 3];
```

Code 2-27 : psC – Extraction d'un tableau.

```
/* affectation d'une valeur à l'index 4
 * t1 = {?, ?, ?, ?, 5, ?, ?}
 */
t1[4] = 5;

/* affectation d'une valeur à l'index 2,2
 * t2 = {?, ?, ?, ?, ?,
 *       ?, ?, ?, ?, ?,
 *       ?, ?, 3, ?, ?,
 *       ?, ?, ?, ?, ?,
 *       ?, ?, ?, ?, ?}
 */
t2[2][2] = 3;

/* affectation de plusieurs valeurs simultanément
 * t3 = {0, 1, 2}
 * t4 = {?, ?, 0, ?, 1, ?, 2}
 */
t4[2, 4, 6] = t3;
```

Code 2-28 : psC – Affectation.

2.5.3 Instructions de sectionnement

Dans le langage psC, il existe les mots clé `to` et `step` qui permettent d'obtenir des valeurs partant d'un nombre jusqu'à un autre en effectuant des pas d'une taille prédéfinie. Les valeurs créées peuvent être placé dans un tableau si désiré ou utilisé directement. En utilisant le résultat pour indiquer les indices, il est donc possible d'effectuer le sectionnement. Le Code 2-29 montre des exemples de tranches.

```
// Tableau de sectionnement
// s = {3, 5, 7, 9}
s = 3 to 9 step 2;

// Sectionnement du tableau t1
// t1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
// t2 = {3, 5, 7, 9}
t2 = t1[3 to 9 step 2];
t2 = t1[s];

// Sectionnement du tableau t3
// t3 = {?, ?, ?, 5, ?, 5, ?, 5, ?, 5}
t3[3 to 9 step 2] = 5;
t3[s] = 5;
```

Code 2-29 : psC – Opération de sectionnement.

2.5.4 Instructions conditionnelles

Il existe deux manières en psC pour effectuer des opérations conditionnelles. La première est l'utilisation de l'opérateur ternaire. Sa syntaxe est la même que celle du langage C. La condition booléenne est calculée pour chaque élément du tableau et l'expression correspondante choisis de manière individuelle. Ceci implique que tous les éléments du tableau n'utiliseront pas nécessairement tous la même expression. Le Code 2-30 montre un exemple d'utilisation de l'opérateur ternaire.

```
// t1 = { 0,  1,  2,  3,  4,  5,  6,  7,  8,  9}
// t2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}
// t3 = {10,  1, 12,  3, 14,  5, 16,  7, 18,  9}
t3 = (t1 & 1 == 1) ? t1 : t2;
```

Code 2-30 : psC – Opération ternaire.

La seconde manière, est d'utiliser un autre tableau pour sélectionner les éléments qui vont être utilisés ou affectés. Le Code 2-31 montre un exemple de cette technique.

```
// t1 = { 0,  1,  2,  3,  4,  5,  6,  7,  8,  9}
// t2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19}
// t3 = {10,  1, 12,  3, 14,  5, 16,  7, 18,  9}
c = [t1 & 1 == 1];
t3 = t1;
t3[c] = t2[c];
```

Code 2-31 : psC – Opération équivalente à l'opération ternaire.

2.5.5 Instructions de réduction

Blitz++ offre une sélection de réductions plus importante que celle offerte par valarray et nous servira de référence. Le Tableau 2-3 donne la liste des instructions supportées par psC. Pour toute autre instruction de réduction, il est nécessaire que le programmeur développe la fonctionnalité avec les outils fournis.

opération	Description
size	Quantité d'éléments
count	Le nombre de fois que l'expression passée est vraie
sum	Somme des éléments
product	Produit des éléments
mean	Moyenne des éléments
min	Élément le plus petit
max	Élément le plus grand
any	Vrai si l'expression passée est vraie pour au moins un élément
all	Vrai si l'expression passée est vraie pour tous les éléments
first	Le premier élément pour lequel l'expression passée est vraie
last	Le dernier élément pour lequel l'expression passée est vraie
extent	Taille de la dimension
dimensions	Nombre de dimension de la structure
stride	Pas de la dimension
transpose	Change l'ordre des dimensions
convolve	Convolution spatiale une dimension

Tableau 2-5 : psC – Instructions de réduction.

2.5.6 Instructions vectorielles

Il y a les opérations unaires, les opérations binaires et les fonctions. Une opération unaire n'a besoin que d'un seul tableau qui le suit et effectue son opération sur lui. La

négation en est un exemple d'une opération unaire. Le résultat est un tableau de taille identique mais dont l'opération a été appliquée à chaque élément. Le Tableau 2-6 liste les opérations unaires de psC.

Opération	Description
-	Inversion de signe
~	Inversion binaire
!	Négation logique
++	Incrémente et assigne (x = x + 1)
--	Décrémente et assigne (x = x – 1)

Tableau 2-6 : psC – Opérations unaires.

Une opération binaire est une opération qui nécessite deux tableaux de tailles identiques, de part et d'autre de l'opérateur. Il existe cependant une exception, l'un des opérandes peut être un scalaire. Dans ce cas, il est vu comme étant un tableau de la taille ayant la valeur du scalaire pour tous ses éléments Le résultat est soit un tableau de taille et type identique dont l'opération a été appliquée à chaque élément ou un tableau de taille identique de type booléen qui indique pour chaque élément si l'expression booléenne est vraie ou fausse pour chaque élément. Le Tableau 2-7 liste les opérations binaires supportées. Pour chaque opérateur binaire qui n'est pas une opération booléenne, il existe également la version d'affectation calculé. L'affectation calculée consiste en la concaténation de l'opérateur égal et d'un autre opérateur. La valeur qui reçoit le résultat est également utilisée comme premier tableau pour effectuer l'opération. En exemple, l'affectation de l'opération d'addition `A = A + B` est équivalente l'affectation calculée suivante `A += B`.

Opération	Description
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo
^	Ou exclusif binaire
|	Ou inclusif binaire
&	Et binaire
<<	Décalage binaire vers la gauche
>>	Décalage binaire vers la droite
<	Plus petit
>	Plus grand
<=	Plus petit ou égal
>=	Plus grand ou égal
==	Égal
!=	Différent
&&	Et logique
||	Ou logique

Tableau 2-7 : psC – Opérations binaires.

Le langage C offre une panoplie d'opérations mathématiques via la bibliothèque standard. De plus, plusieurs bibliothèques spécialisées existent qui augmentent cette offre. Le langage psC offrira les fonctions du Tableau 2-8, qui sont présentement supportées en opérations non data-parallèle.

Opération	Description
abs	Valeur absolue
ceil	Arrondissement à l'entier supérieur
floor	Arrondissement à l'entier inférieur
sqrt	Racine carré
cbrt	Racine cube
exp	Exponentiation
log	Logarithme à base 2
log10	Logarithme à base 10
lg	Logarithme naturel
cos	Cosinus
tan	Tangente
sin	Sinus
acos	Cosinus inverse
atan	Tangente inverse
asin	Sinus inverse
cosh	Cosinus hyperbolique
sinh	Sinus hyperbolique
tanh	Tangente hyperbolique
pow	Puissance
isalpha	Le caractère est une lettre
isdigit	Le caractère est un nombre
isalnum	Le caractère est un nombre ou une lettre
tolower	Transforme le caractère en minuscule
toupper	Transforme le caractère en majuscule
erf	Fonction d'erreur
erfc	Fonction d'erreur complémentaire

Tableau 2-8 : psC – Fonctions data-parallèles.

Le but visé étant d'utiliser les instructions data-parallèles disponible dans les processeurs, des fonctions utilisant les fonctions spécialisées sont offertes. Ces fonctions sont particulièrement utiles pour les applications multimédia. Le Tableau 2-9 liste ces fonctions.

Opération	Description
adds	Addition saturée
subs	Soustraction saturée
mulh	Multiplication de la moitié haute des entiers
mulex	Effectue la multiplication avec retour qui évite de tronquer la réponse (type de retour plus grand).

Tableau 2-9 : psC – Opérations multimédia.

3

RÉALISATION

3.1 Choix du langage

Le langage choisi est le psC. C'est un langage dont la syntaxe est basée sur celle du langage C. Cependant, il possède plusieurs mots clé et éléments de syntaxe qui lui permettent d'agir comme langage de description matériel. Il ne contient pas d'instructions data-parallèles, mais il possède d'autres caractéristiques qui vont aider lors de la mise en œuvre. Les deux caractéristiques qui nous intéressent sont décrites ici-bas.

Premièrement, c'est un langage fortement typé. Ceci veut dire que le type de données contenu dans une variable est connu lors de la compilation et ne peut pas changer en cours d'exécution. Ça implique également que la taille en mémoire de chaque variable est connue lors de la compilation. Ceci n'empêche pas d'affecter une valeur d'un autre type à une variable. Des conversions (« type cast ») similaires à celles du langage C sont disponibles.

Deuxièmement, il n'y a pas d'allocation dynamique. Toute la mémoire nécessaire est connue lors de la compilation. Cette mémoire est allouée lors du démarrage de l'application. Pour les tableaux, la taille maximale est allouée. S'il n'y a pas assez de mémoire de disponible, l'application ne démarre tout simplement pas. Ceci permet de

prendre des raccourcis lors de la mise en œuvre car aucune validation d'échec d'allocation ou de taille n'est nécessaire lors de l'exécution. L'information est disponible lors de la compilation. Cependant, cette contrainte pourra être enlevée dans une version future.

3.2 Historique du langage psC

La programmation des composantes physiques via les langages de descriptions matériels (ABEL, Verilog HDL, VHDL, ...) est complexe. Les techniques de programmations et les algorithmes sont différents de ceux supportés par les langages de programmation de logiciels. Les programmeurs logiciels sont familier avec la programmation « séquentielle » mais peu avec la programmation « parallèle », requise pour programmer ou décrire le matériel. Ces techniques de programmation se rapprochent plus de la conception électronique que de la programmation.

Un langage de description matérielle de haut niveau (HL-HDL), qui permettrait au programmeur logiciel de programmer ou décrire le matériel avec des techniques qui lui sont familières était nécessaire. C'est ce qui a motivé le développement de la première version du langage, appelée K3. Il était basé sur le langage de programmation LISP. Cependant, le langage supportait naturellement le parallélisme. Toutes les instructions d'une fonction étaient exécutées en parallèle, tout comme dans les puces électroniques.

Le K3 est basé sur le langage LISP, un lange fonctionnel peu utilisé. Suite à des consultations réalisés pas Novakod Technologies, il a été déterminé que la syntaxe devrait être basé sur le langage C. La première étape a donc été de transformer la syntaxe K3 en une syntaxe similaire au C, qui deviendra psC. Durant cette étape, aucune nouvelle

instruction n'est introduite, l'arbre syntaxique interne n'est pas modifié, seulement la syntaxe du langage. Les instructions data-parallèles ont été ajoutées pendant la migration vers le psC.

3.3 Description de la syntaxe du psC

Chacun des points suivants expliquent les concepts du langage psC.

3.3.1 Composant

Un composant en psC est l'équivalent d'une classe en C++. C'est une entité qui permet de regrouper les variables et les fonctions interdépendantes. Un composant est soit hiérarchique, formé de signaux et d'autres composants, ou encore un composant exécutable, formés de variables et de fonctions. Par exemple, un additionneur de quatre nombres, formés de trois composants interconnectés est un composant hiérarchique, voir la Figure 3-1.

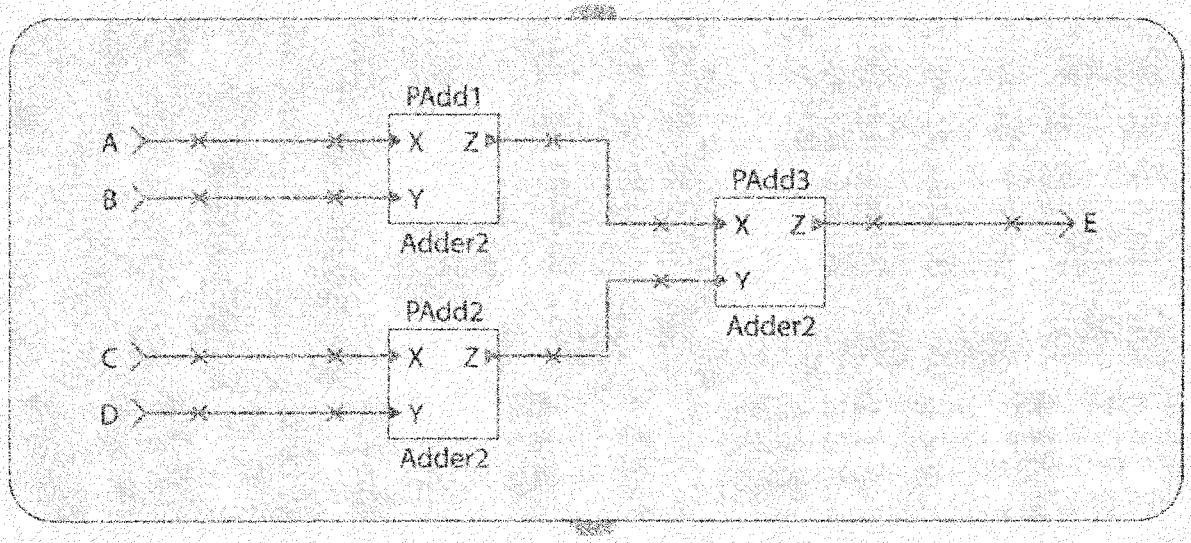

Figure 3-1: psC – additionneur à quatre nombres en graphique.

Le composant décrit par le Code 3-1 est un composant exécutable. Il comprend une seule fonction `ExecuteAdd(0)`, incluant une instruction d'assignation. La fonction est exécutée s'il y a un événement sur les ports X ou Y. Le résultat apparaît sur le port de sortie, accompagné d'un événement.

```
component Adder2 (
  in  active int X,
  in  active int Y,
  out active int Z)
{
  ExecuteAdd(0) on X, Y
  {
    Z := X + Y;
  }
};
```

Code 3-1 : psC – additionneur.

3.3.2 Ports et signaux

Le Code 3-2 en langage psC correspond à la représentation graphique de la Figure 3-1.

```
component Adder4(
  in  int A,
  in  int B,
  in  int C,
  in  int D,
  out int E)
{
  component Adder2(
    in  active int X,
    in  active int Y,
    out active int Z)
    {
      /* Effectue l'addition lors d'un évènement sur X ou Y */
      ExecuteAdd(0) on X, Y
      {
        /* les ":" sont utilisé pour effectuer un évèvement de sortie */
        Z := X + Y;
      }
    };

    // déclarations des processus
    Adder4::Adder2  PAdd1;
    Adder4::Adder2  PAdd2;
    Adder4::Adder2  PAdd3;

    // déclarations des signaux
    int signal0 = {A, PAdd1.X};
    int signal1 = {B, PAdd1.Y};
    int signal2 = {C, PAdd2.X};
    int signal3 = {D, PAdd2.Y};
    int signal4 = {PAdd1.Z, PAdd3.X};
    int signal5 = {PAdd2.Z, PAdd3.Y};
    int signal6 = {PAdd3.Z, E};
};
```

Code 3-2 : psC – additionneur à quatre nombres en texte.

Le code comprend la déclaration des composants, ici `Adder2`, des processus et des signaux. Un signal est un lien qui transporte les données d'un composant à un autre. Chaque signal comprend un port de sortie et un ou plusieurs ports d'entrées, par exemple, les signaux `signalx` dans le Code 3-1

Les mots clé `in` et `out` indiquent la direction. Le mot clé `active` indique qu'une donnée supplémentaire, un événement, est transportée en plus de la donnée. Cette

information est utilisée pour déclencher l'exécution des fonctions associées à ce signal. À l'inverse, le mot clé `passive` indique l'absence d'événement. L'opérateur deux points (:) peut être utilisé sur les signaux actifs pour envoyer un évènement.

3.3.3 Type

Le langage psC supporte les types usuels ainsi que des types « point fixe ». Le Tableau 3-1 en fait la liste et les décrits. Le langage supporte également les types définis par l'utilisateur, le mot clé `typedef` permet de définir les nouveaux types.

Type	Description
bool	Définit un type logique dont les valeurs sont soit `true` ou `false`.
bit	Définit un entier non signé de 1 bit.
ubyte	Définit un entier non signé de 8 bits.
ushort	Définit un entier non signé de 16 bits.
uint	Définit un entier non signé de 32 bits.
ulong	Définit un entier non signé de 64 bits.
byte	Définit un entier de 8 bit.
short	Définit un entier de 16 bit.
int	Définit un entier de 32 bit.
long	Définit un entier de 64 bit.
fix8	Définit un nombre à point fixe de 8 bit.
fix16	Définit un nombre à point fixe de 16 bit.
fix32	Définit un nombre à point fixe de 32 bit.
fix64	Définit un nombre à point fixe de 64 bits.
float	Définit un nombre à point flottant de 32 bits.
double	Définit un nombre à point flottant de 64 bits.
time	Définit un entier de 64 bit utilisé pour la représentation du temps.

Tableau 3-1 : types du psC.

3.3.4 Fonctions

Une fonction est une série d'instructions effectuées en parallèle lorsqu'une condition spécifique survient. Ils sont l'équivalent des fonctions membres d'une classe en C++. Le langage psC supporte des fonctions définies par l'utilisateur, par exemple la

fonction `ExecuteAdd(0) on X, Y` de l'additionneur précédent. Le mot clé `on` indique sur quel événement d'entrée la fonction est exécutée. Il existe également des fonctions prédéfinies, listées au Tableau 3-2

Type	Description
start	Effectue l'opération au démarrage du programme. Analogue au constructeur d'une classe en C++.
stop	Effectue l'opération à l'arrêt du composant. Analogue au destructeur d'une classe en C++.
always	Effectue l'opération à chaque cycle.
timerEnd	Effectue l'opération lorsque la minuterie termine.

Tableau 3-2 : Fonctions prédéfinies.

3.3.5 Fonctions intrinsèques

En plus des fonctions définies par l'utilisateur, le psC comprend plusieurs fonctions intrinsèques, correspondants aux fonctions des libraires du langage C. Dans sa version actuelle, il n'est pas possible pour le programmeur de définir de nouvelles fonctions intrinsèques. Cette limitation est compensée par la possibilité de définir des fonctions internes aux composants, tels que décrit à la section précédente.

3.3.6 Opérateur

Le langage psC supporte les mêmes opérateurs que le langage C (voir le Tableau 3-3).

Catégorie	Opérateur	Exemple
Arithmétique	Addition	A+B
	Soustraction	A-B
	Multiplication	A*B
	Division	A/B
	Inverse	-A
	Modulo	A%B
	Pré-incrément	++A
	Post-incrément	A++
	Pré-décrément	--A
	Post-décrément	A--
Comparaison	Égal	A==B
	Différent	A!=B
	Plus grand	A>B
	Plus petit	A<B
	Plus grand ou égal	A>=B
	Plus petit ou égal	A<=B
Logique	Inversion	!A
	Et	A&&B
	Ou	A||B
Binaire	Inversion	~A
	Et	A&B
	Ou	A|B
	Ou exclusif	A^B
	Déplacement à gauche	A<<B
	Déplacement à droite	A>>B
Affectation	Affectation	A=B
Composé	Addition et affectation	A+=B
	Soustraction et affectation	A-=B
	Multiplication et affectation	A*=B
	Division et affectation	A/=B
	Modulo et affectation	A%=B
	Et binaire et affectation	A&=B
	Ou binaire et affectation	A|=B
	Ou exclusif binaire et affectation	A^=B
	Déplacement à gauche et affectation	A<<=B
	Déplacement à droite et affectation	A>>=B
Adressage	Déplacement	A[B]
	Indirection	*A
	Référence	&A

Tableau 3-3 : Opérateurs de psC.

3.3.7 Transtypage

Comme dans le langage C++, le transtypage existe sous deux formes, implicite et explicite. Le transtypage implicite ne requiert aucune information de la part du développeur et permet de modifier les types similaires. Le transtypage explicite identique à celui au langage C. Le Code 3-3 montre un exemple de transtypage en psC.

```
varType1 = (type1)varType2;
```

Code 3-3 : psC – transtypage.

3.4 Ajout des instructions data-parallèles

La grammaire du psC contient presque tout ce qui est nécessaire pour supporter les opérations data-parallèles. Voici ce qui est manquant :

- les règles de grammaires qui permettent de travailler directement avec des tableaux,
- le type `index`, avec les règles de grammaires qui s'y rattachent,
- des fonctions spécialisé pour du traitement data-parallèles.

Chacun de ces points va être traité en détail dans les paragraphes qui suivent.

3.4.1 Ajout de la grammaire data-parallèles sur les tableaux

Dans le langage, les opérations et les fonctions s'attendent à travailler sur des scalaires. Par exemple, dans l'expression `A + B`, `A` et `B` doivent être des scalaires. Avec l'ajout data-parallèle dans la syntaxe, ces variables peuvent maintenant être des tableaux de même type et de même taille.

Les fonctions du psC peuvent maintenant être appelées avec un tableau en argument. L'opération effectuée par la fonction s'effectue sur chacun des éléments du tableau, de manière indépendante et parallèle.

3.4.2 Ajout du type index

Pour plusieurs opérations data-parallèles, il est important de connaître comment les indices sont liés ensemble. Par exemple, pour effectuer la transposé d'une matrice, on inverse les indices lors de l'affectation (voir le Code 3-4). Pour être capable d'indiquer de telles opérations, le type index est utilisé.

```
index i;
index j;
A[i][j] = B[j][i];
```

Code 3-4 : psC – type index explicite.

3.4.3 Ajout de fonctions spécialisées data-parallèles

Comme décrit en 2.2, plusieurs instructions spécialisées devraient être disponibles en psC. Durant ce travail, seulement celles nécessaires ont été implémentés. Le Tableau 3-4 en fait la liste. Dans ce tableau, `T` représente un tableau à une dimension, `i` un index et `s` un scalaire.

Opérateur	Résultat	Description
min(T1, T2)	T3	Retourne la valeur minimale pour chaque indice de T1 et T2.
sum(T)	s	Retourne la somme de tous les éléments de T.
T1 + T2	T3	Retourne l'addition de chaque indice de T1 et T2.
T1 * T2	T3	Retourne la multiplication de chaque indice de T1 et T2.
T1[i]	s	Retourne la valeur de T1 à l'indice i.
T1[i1] = T2[i2]	T1[i1]	Affecte la valeur de T2 à l'indice i2 dans T1 à l'indice i1 et retourne cette valeur.
i1 + i2	i3	Additionne deux indices pour en former un troisième.
T1[s1]	s2	Retourne la valeur de T1 à l'indice s1.
i + s	i2	Additionne un scalaire à un indice pour en former un nouveau.

Tableau 3-4 : Opérations data-parallèles implémentées.

3.5 Compilation

Lors de la compilation, il faut détecter ce qui est une instruction data-parallèle parmi toutes les instructions. Une foi détecté, il faut générer l'arbre abstrait interne de manière appropriée.

Étudions l'opération psC montré au Code 3-5. Cette opération va s'effectuer chaque fois que le signal `exec` sera activé.

```
DoMult(0) on exec
{
  /* int Z
   * int A
   * int B
   */
  Z := A % B;

  /* int Y[32]
   * int X[32][12]
   * int B[32]
   */
  Y[i] := sum(X[i][j]) + B[i];
}
```

Code 3-5 : psC – index type.

Cette opération donnera deux résultats, `Z` et `Y`. Les deux résultats se calculent simultanément. Ceci est le parallélisme supporté par le langage psC. Le premier résultat, `Z`, provient d'une opération scalaire. Il est le résultat du modulo du scalaire `A` par le scalaire `B`. Le second résultat est le résultat d'une opération data-parallèle.

La détection du data parallélisme est effectué dans l'arbre abstrait interne, lorsqu'un nœud contient une fonction (ou un opérateur) et qu'au moins un des enfants de la fonction (ou opérateur) est un tableau. Tout ce qui se situe sous ce nœud fait partie d'un calcul data-parallèle et traité comme tel. Tout le reste est traité par l'engin scalaire. Voir la Figure 3-2, dans laquelle la somme des éléments du tableau Z est effectuée en parallèle (`sum(Z[i])`.

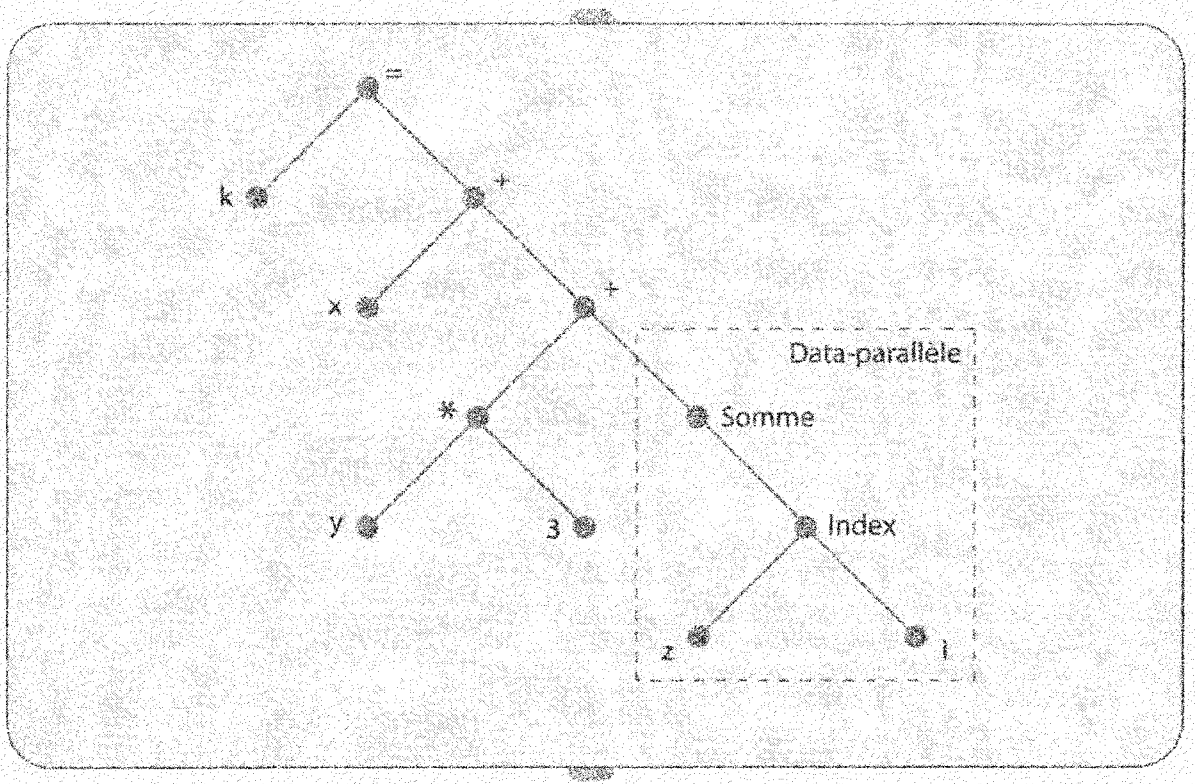

Figure 3-2: Algorithme synchrone.

3.5.1 Validation

Chaque nœud de l'arbre abstrait interne identifié comme data-parallèle est validé.

La première validation est au niveau du type des opérandes. Si le type n'est pas exactement le même, des opérations de transtypage implicite sont ajoutées entre l'opérande et l'opérateur. S'il n'est pas possible d'en ajouter pour obtenir le résultat souhaité, une erreur de compilation est indiquée.

La seconde validation est la taille des opérandes. L'un des opérandes doit être un tableau. Les autres opérandes peuvent être des scalaires ou des tableaux. Les tableaux doivent être de tailles identiques. S'ils ne le sont pas, une erreur de compilation est générée. Les scalaires, quant à eux, sont transformés en en un tableau de la bonne taille, ayant la valeur scalaire pour chacun de ses éléments.

Si aucune erreur n'est générée, nous somme prêt à passer à l'étape suivante, la génération.

3.6 Génération de code assembleur

3.6.1 Architecture MMX et SSE

Le travail consiste à générer du code assembleur utilisant les registres et opérations SSE. Cette section présente un sommaire de l'architecture SSE et des instructions SSE utilisées. La Figure 3-3 montre schématiquement les registres MMX et SSE. Le Tableau 3-5 liste les opérations qui seront utilisées.

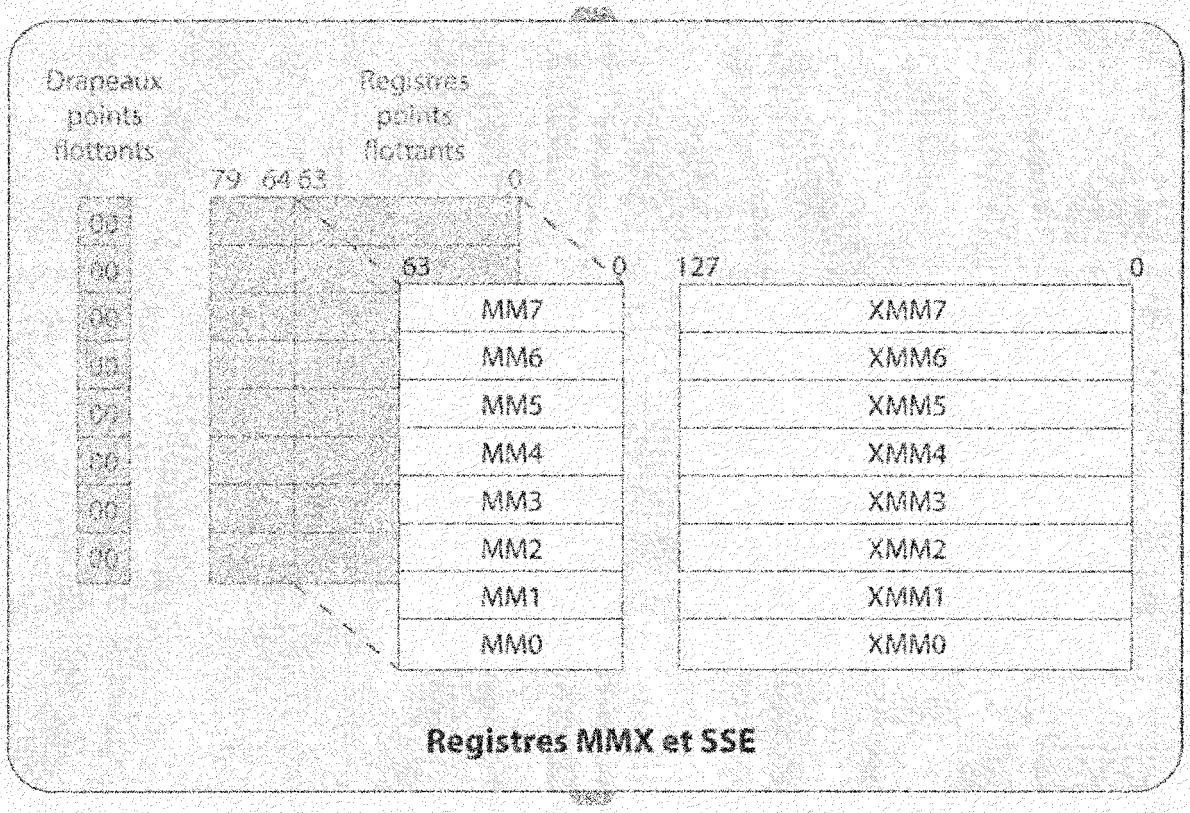

Figure 3-3: Registres MMX et SSE.

Opération	Description
addpd	Addition de valeurs à point flottant double précision, d'un registre xmm ou de la mémoire à un registre xmm
addps	Addition de valeurs à point flottant simple précision, d'un registre xmm ou de la mémoire à un registre xmm
emms	Place les registres dans un état pour utilisation par l'engin de calcul à point flottant x87
haddpd	Addition horizontale de valeurs à point flottant double précision, d'un registre xmm ou de la mémoire à un registre xmm
haddps	Addition horizontale de valeurs à point flottant simple précision, d'un registre xmm ou de la mémoire à un registre xmm
movdqa	Déplace 128 bits de la mémoire à un registre xmm ou l'inverse
movq	Déplace 64 bits de la mémoire à un registre mmx ou xmm ou l'inverse
mulpd	Multiplication de valeurs à point flottant double précision, d'un registre xmm ou de la mémoire à un registre xmm
mulps	Multiplication de valeurs à point flottant simple précision, d'un registre xmm ou de la mémoire à un registre xmm
pand	Et binaire
pminub	Minimum entre les entiers non signé contenu dans un registre mmx ou xmm et ceux d'un registre mmx, xmm ou de la mémoire
por	Ou binaire
prefetchnta	Instruction de pré-chargement des caches non temporelles
prefetcht0	Instruction de pré-chargement des caches temporelles
pshufhw	Mouvement de données de 16bits à dans la partie haute d'un registre xmm
pshuflw	Mouvement de données de 16bits à dans la partie basse d'un registre xmm
psrld	Déplacement à droite de valeurs de 16 bits simultanément
punpckhqdq	Garde dans un registre xmm cible les parties haute de ce registre et d'un autre registre xmm ou de la mémoire
pxor	Ou exclusif binaire
unpckhps	Garde dans un registre xmm cible les parties hautes de ce registre et d'un autre registre xmm ou de la mémoire
unpcklps	Garde dans un registre xmm cible les parties basses de ce registre et d'un autre registre xmm ou de la mémoire

Tableau 3-5 : Opérations MMX et SSE utilisés.

3.6.2 Principe

La génération du code est effectuée par patrons. Un patron est un ou plusieurs nœuds organisés d'une manière donné ayant des opérandes de types spécifiques. Par exemple, la somme des éléments d'un tableau d'entiers de huit bits est un patron. Des patrons peuvent être basés sur des critères très complexes.

Quand plusieurs patrons peuvent s'utiliser à un nœud donné, celui choisi sera celui qui transforme le plus grand nombre de nœuds en instructions machines en même temps.

Les instructions générés par un patron sont imbriqués les unes dans les autres selon l'arbre abstrait interne. Par exemple : un patron qui effectue une addition pourra inclure, au sein de ses instructions, les instructions produite par un patron de multiplication si l'un de ses opérandes est le résultat d'une multiplication.

Un patron de base a été fabriqué pour chaque type de nœud différent utilisé par les applications utilisées pour mesurer les performances. Ensuite, basé sur les résultats obtenus, des patrons plus spécifiques, qui incorporent plus de nœuds, ont étés fabriqués de la manière la plus optimales. Chacun d'eux est décrit plus en détail dans les sections qui suivent. Ils ont été inspirés par des documents d'Intel dont (Intel Corporation, 1999, az), (Intel Corporation, 2010, bj), (Intel Corporation, 2010, bk) et (Intel Corporation, 1999, be).

3.6.3 Patron : index

Les index sont utilisés pour effectuer le bouclage. Le bouclage est l'action d'exécuter les mêmes instructions plusieurs fois. Une autre possibilité existe pour obtenir ce résultat, c'est la récursivité. Elle n'a pas été utilisée car il n'y a pas d'instructions spécifiques pour utiliser la récursivité dans les processeurs et la mise en œuvre demande plus d'instructions que le bouclage. Ceci implique des performances moindres.

Pourquoi utiliser le patron de bouclage ? Car il est utilisé de manière implicite dans presque tous les algorithmes data-parallèles. À moins que toutes les données nécessaires pour traiter la taille d'un vecteur entre dans un registre SSE, il sera nécessaire de boucler.

Le patron de bouclage utilisé est montré au Code 3-6. Ce patron est rapide, demande qu'un seul registre et peu d'instructions. Il est possible de l'imbriquer à l'intérieur de lui-même en l'entourant d'un simple `push ecx` avant et d'un `pop ecx` après. La constante `PROCESSED_SIZE` représente la taille des données qui ont été traités à l'intérieur de la boucle. Généralement, ce sera seize bytes sur les architectures utilisés, ce qui correspond à la taille des registres SSE. La constante `ARRAY_SIZE` est la taille du tableau à traité. Le marqueur est nommé en fonction du nom de l'index.

```
/* Place le contenu de ecx à zoré */
xor ecx, ecx

/* Marqueur de bouclage */
lbl_i:

/* placer ici les instructions répétés */

/* Incrément du compteur */
add ecx, PROCESSED_SIZE

/* Validation de fin de boucle */
cmp ecx, ARRAY_SIZE

/* Branchement si la boucle n'est pas terminée */
jl lbl_i
```

Code 3-6 : psC – patron index.

3.6.4 Patron : chargement d'une donnée

Le chargement de données contenues dans un vecteur à traiter avec les instructions data-parallèles est le même, peu importe le type de la donnée.

Premièrement, le pointeur sur le début des données est placé dans un registre du processeur. Ce registre est ajusté au besoin quand plusieurs bouclages sont imbriqués. Deuxièmement, l'instruction `movaps` est utilisée pour charger les données dans un registre

XMM. Les données doivent être alignées à seize bits. Le Code 3-7 montre l'opération complète pour deux niveaux d'imbrication de boucle. Le registre xmm0 est rempli avec les données du vecteur A qui n'a qu'une seule dimension. Le registre xmm1 est rempli avec les données du vecteur B, qui est un vecteur de deux dimensions dont la première est égal à celle de A.

```
/* Chargement de l'adresse des données dans des registres */
mov esi, A
mov eax, B

/* début de bouclage sur i */
xor ecx, ecx
lbl_i :

/* chargement des données de A */
movaps xmm0, xmmword ptr [esi+ecx*4]

/* début de bouclage sur j */
push ecx
xor ecx, ecx
lbl_j :

/* chargement des données de B */
movaps xmm1, xmmword ptr [eax+ecx*4]

/* ...traitement des données ici... */

/* fin de bouclage sur j */
add ecx, 4
cmp ecx, B_SIZE
jl lbl_j
pop ecx

/* fin de bouclage sur i, avec ajustement pour pointeur de données pour B
 */
add eax, B_2ND_SIZE
add ecx, 4
cmp ecx, A_SIZE
jl lbl_i
```

Code 3-7 : psC – Patron chargement de données alignées.

3.6.5 Patron : chargement de données non-alignées

Lorsqu'il n'est pas possible d'avoir des données alignées, l'instruction `movups` peut être utilisée de la même manière que `movaps`. Elle est seulement moins performante.

3.6.6 Patron : sauvegarde d'une donnée

Le patron de sauvegarde des données est basé sur la même logique que celui de chargement des données. La différence est l'ordre des opérandes de l'instruction `movaps`. Au lieu d'effectuer le mouvement des données de la mémoire à un registre, elle est déplacée d'un registre à la mémoire.

3.6.7 Patron : minimum d'entiers de huit bits non signés

L'instruction `pminub` est une instruction data-parallèle présente dans les processeurs. Lorsqu'un nœud demande un minimum et que ses opérandes sont des entiers de huit bits non signés, il suffit d'appeler cette instruction, comme le montre le Code 3-8. Dans cet exemple, `xmm0` et `xmm1` sont les registres où se trouvent les opérandes.

```
pminub xmm0, xmm1
```

Code 3-8 : psC – Patron minimum

3.6.8 Patron : multiplication de points flottants à trente-deux bits

L'instruction `mulps` est une instruction data-parallèle présente dans les processeurs. Lorsqu'un nœud demande une multiplication et que ses opérandes sont des points flottants de trente-deux bits (`float`), il suffit d'appeler cette instruction, comme le montre le Code 3-9. Dans cet exemple, `xmm0` et `xmm1` sont les registres où se trouvent les opérandes.

```
mulps xmm0, xmm1
```

Code 3-9 : psC – Patron multiplication parallèle valeur points flottant de simple précision.

3.6.9 Patron : multiplication de points flottants à soixante-quatre bits

Les points flottant à soixante-quatre bits, communément appelé `double`, sont souvent utilisés par les calculs scientifiques où la précision est importante. L'instruction `mulpd` est l'équivalent de l'instruction `mulps` vu précédemment. Elle s'utilise de la même manière. La différence est qu'elle effectue deux multiplications en parallèle au lieu de quatre. Le Code 3-10 montre son utilisation.

```
mulpd xmm0, xmm1
```

Code 3-10 : psC – Patron multiplication parallèle valeur points flottant de double précision.

3.6.10 Patron : addition de points flottants à soixante-quatre bits

Cette opération est, tout comme la multiplication, une instruction tellement de base qu'il existe une instruction pour l'effectuer, `addpd`. Le Code 3-11 montre l'utilisation. Si le premier opérande est une sommation, et que le second opérande ne l'est pas, les opérandes sont inversés. Ceci permet d'utiliser le registre de résultat comme registre de sommation de la somme, et d'éviter une instruction de remise à zéro de ce registre et une instruction d'addition. Le Code 3-12 effectue l'opération selon l'ordre naturel, c'est-à-dire que l'addition est effectuée après la somme. Le Code 3-13 quant à lui effectue l'addition au départ. Les différences sont placées en caractères gras.

```
addpd xmm0, xmm1
```

Code 3-11 : psC – Patron addition parallèle valeur points flottant de simple précision.

```
/* dans l'initialisation de la boucle externe */
mov eax, B
mov esi, A

// dans l'initialisation de la boucle interne
pxor xmm0, xmm0

// dans la boucle interne
addpd xmm0, xmmword ptr [esi+ecx*2]

// après la fin de la boucle interne
addpd xmm0, xmmword ptr [eax+ecx+2]
```

Code 3-12 : psC – sum(A[i][j]) + B[i].

```
// dans l'initialisation de la boucle externe
mov eax, B
mov esi, A

// dans l'initialisation de la boucle interne
movapd xmm0, xmmword ptr [eax+ecx*2]

// dans la boucle interne
addpd xmm0, xmmword ptr [esi+ecx*2]
```

Code 3-13 : psC – B[i] + sum(A[i][j]).

3.6.11 Patron : somme de points flottant trente-deux bits, résultats scalaire

Ce patron demande de placer des instructions dans la partie d'initialisation d'une boucle, et dans la boucle elle-même. Il requiert un bouclage pour s'exécuter, comme mentionné dans le Code 3-14. Le registre `xmm0` est utilisé pour y placer le résultat de la somme. Le première donnée (bits zéro à trente-et-un) sera la valeur scalaire. Le registre `esi` pointe sur les données à sommer.

```
/* dans l'initialisation de la boucle */
pxor  xmm0, xmm0

/* ... */

/* dans la boucle */
addps xmm0, xmmword ptr [esi+ecx*4]

/* ... */

/* après la fin de la boucle */
haddps xmm0, xmm0
haddps xmm0, xmm0
```

Code 3-14 : psC – Patron somme de points flottant à simple précision.

3.6.12 Patron : multiplication matricielle de points flottant soixante-quatre bits

Ce patron est le plus complexe jusqu'à présent. L'utilisation d'un double bouclage est nécessaire.

Le patron demande que le premier opérande soit une matrice et que le second soit un vecteur. Ce patron possède un nombre caractéristique. Pour le trouver, l'opération suivante est utilisée : le nombre de bits disponible dans un registre (cent-vingt-huit bits), divisé par la taille des opérandes (soixante-quatre bits). Les tailles (première et seconde dimension de la matrice et celle du vecteur) doivent être un multiple de ce nombre caractéristique.

Ce patron utilise les registres suivants. Le registre *ecx* est utilisé pour le bouclage. Le registre *esi* est utilisé comme pointeur sur les données de la matrice. Le registre *eax* est utilisé comme pointeur des données du vecteur. Le registre *xmm0* est utilisé comme registre de sommation. Les registres *xmm1* à *xmmX*, où X est le nombre caractéristique, sont utilisés comme registres de multiplication. Le Code 3-15 illustre ce patron.

```
/* initialisation de la boucle externe */

/* ... */

/* initialisation de la boucle interne */
pxor xmm0, xmm0

/* ... */

/* dans la boucle interne */
movapd xmm1, xmmword ptr [eax+ecx*2]
movapd xmm2, xmm1
mulpd xmm1, xmmword ptr [esi+ecx*2]
mulpd xmm2, xmmword ptr [esi+ecx*2+MATRIX_2ND_SIZE_BITS]
haddps xmm1, xmm2
addps xmm0, xmm1
```

Code 3-15 : psC – Patron multiplication matriciel de points flottant à double précision.

3.6.13 Patron : transformation d'un scalaire de huit bits non signé en un vecteur

Les instructions de déplacement *pshufhw* et *pshuflw* sont utilisé, comme montré par le Code 3-17, pour transformé un scalaire en vecteur. Cependant, ces instructions travaillent avec des entiers de seize bits et non de huit bits. Pour compenser ce fait, d'autres instructions sont utilisés, ainsi qu'un masque placé de manière statique dans le code (voir Code 3-16). Le scalaire est placé dans un registre SSE préalablement (*xmm0* dans ce cas). Suites aux instructions, chacun des octets présents dans le registre de résultat (*xmm1* dans ce cas-ci) seront égal au scalaire.

```
__declspec(align(16)) static const unsigned __int64 mask[] =
{0xff00ff00ff00ff00, 0xff00ff00ff00ff00};
```

Code 3-16 : psC – Masque pour transformation d'entier huit bits en vecteur.

```
pshufhw xmm1, xmm0, 0xF5
pshuflw xmm1, xmm1, 0xF5
pand    xmm1, xmm2
movdqa  xmm3, xmm1
psrld   xmm1, 8
por     xmm1, xmm3
```

Code 3-17 : psC – Patron transformation d'entier huit bits en vecteur.

4

RÉSULTAT

4.1 Présentation des tests

Trois algorithmes classiques, utilisé dans des domaines différents, ont été choisis pour valider les performances et mesurer les gains. Ces trois algorithmes sont le filtre à réponse impulsionnelle finie, la saturation alpha et la multiplication matricielle. Chacun de ces algorithmes est traité individuellement dans les sections suivantes. Suivra un sommaire des résultats.

Pour chacun, plusieurs mises en œuvre ont été utilisées afin de voir le gain versus l'effort. Le Tableau 4-1 liste les mises en œuvre effectuées.

Mises en œuvre
Langage C, compilateur Microsoft
Langage C, compilateur Intel
Assembleur, avec instruction MMX, SSE, SSE2 et SSE3
Langage psC

Tableau 4-1 : mises en œuvre de validation

Pour obtenir le temps de calcul, les fonctions du compteur haute performance des processeurs ont été utilisées. La fonction `QueryPerformanceFrequency` donne le nombre de tics effectué en une seconde. La fonction `QueryPerformanceCounter` donne un nombre de tics. En effectuant l'appel de cette fonction avant et après le calcul, il est possible de

connaître le nombre exact de tics passé au cours de l'appel. En divisant ce nombre par la fréquence, on obtient le temps utilisé en seconde.

De plus, la priorité du processus est placé à temps réel. C'est la classe de priorité la plus importante qu'il est possible d'avoir. Le thread est placé comme étant critique, la priorité la plus élevé possible. Avec ces priorités, l'ordinateur ne répond plus durant l'exécution des calculs. Toute la puissance de calcul étant utilisée par le thread.

Les tests sont effectués douze fois, avec les mêmes valeurs. Les deux premières fois sont ignorées. Les temps des dix autres fois sont additionnés pour obtenir la valeur de temps utilisé pour la comparaison des différentes techniques.

4.2 Filtre à réponse impulsionnelle finie

Le filtre à réponse impulsionnelle finie (RIF) est un filtre numérique utilisé en traitement du signal. Il est basé uniquement sur les valeurs d'entrée. C'est-à-dire qu'il ne possède pas de rétroaction. Il est une moyenne pondéré des termes d'entré. La Figure 4-1 l'illustre. L'implémentation du filtre RIF est faite avec des valeurs à point flottant de 32 bits.

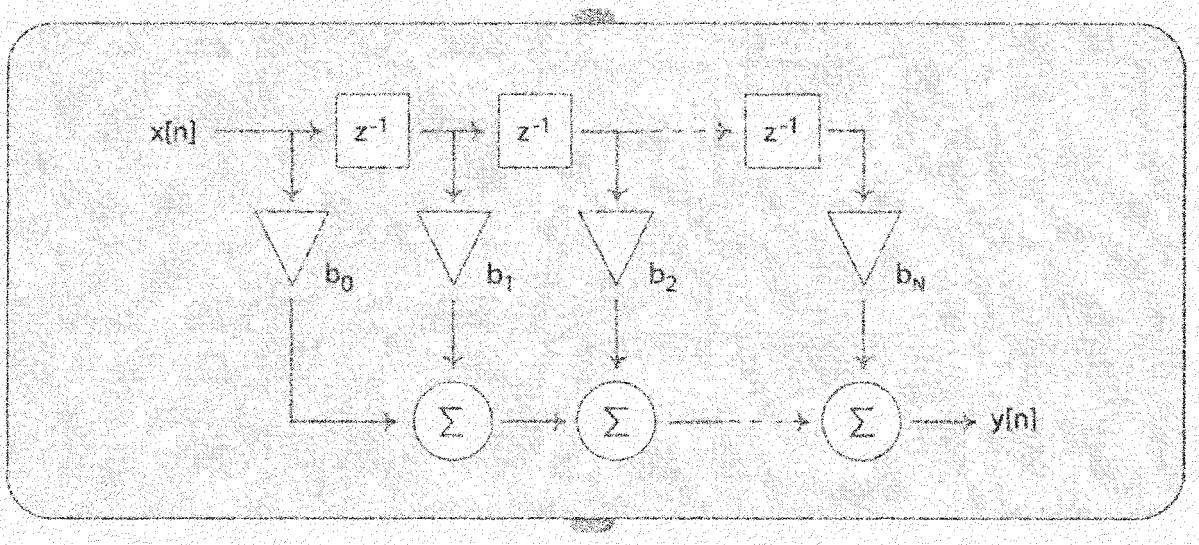

Figure 4-1: Filtre à réponse impulsionnelle finie.

Les entrées dans l'algorithme sont :

- Un vecteur d'entré `a_inputs` contenant les valeurs sur lesquelles le filtre sera appliqué. Sa taille doit être d'au minimum de celle du vecteur de sortie plus celle du vecteur des valeurs de pondérations. Les valeurs supplémentaires doivent être placées avant les entrées, et représentent le passé. Si le passé n'est pas connu, ou est ignoré, ces valeurs seront toutes des zéro.

- Un vecteur des valeurs de pondération `a_taps` à utiliser par le filtre. L'index du tableau est utilisé comme indice de temps.

- La taille du vecteur de valeurs pondérées `FILTER_ORDER`. Pour ce travail, seulement la valeur 16 a été utilisée. L'indice le plus bas (0) correspond au

temps le plus loin et l'indice le plus élevé (15) correspond au temps le plus proche. Ceci dans le but de simplifier le code en évitant des calculs d'index.

- Un vecteur de sorti `a_outputs` dans lequel les résultats seront placés.
- Le nombre de valeur de sortie `a_outputsLength` désiré.

4.2.1 Implantation C

L'implantation en langage C est simple. Elle consiste en l'exécution d'une boucle qui effectue la somme pondéré, `n` fois. Le Code 4-1 en est l'implémentation.

```
void fir(
  float * a_inputs,
  float * a_taps,
  float * a_outputs,
  int a_outputsLength)
{
  for (int n = 0; n < a_outputsLength; ++n)
  {
    float accummulator = 0.0;
    for (int m = 0; m < FILTER_ORDER; ++m)
    {
      accummulator += a_taps[m] * a_inputs[n + m + 1];
    }
    a_outputs[n] = accummulator;
  }
}
```

Code 4-1 : Filtre RIF en C

4.2.2 Implantation Assembleur

Le code assembleur est similaire à la boucle en langage C. Cependant, il est beaucoup plus complexe. Le code est basé sur la documentation d'Intel (Intel Corporation, 1999, az).

L'implantation pose une contrainte sur la taille du vecteur de sortie `a_outputsLength`, elle doit être divisible par quatre. Les instructions SSE ont été utilisées.

Ceci demande d'avoir les vecteurs (entré, valeurs pondérés et sortie) accessibles à une adresse mémoire divisible par 16.

Le Code 4-2 montre comment cette opération est effectuée. Pour l'algorithme de filtre RIF.

```
bool FilterSSE(
  float * a_inputs,
  float * a_taps,
  float * a_outputs,
  int a_outputsLength)
{
  /* Alloue l'espace nécessaire pour les valeurs pondérés alignés */
  float * taps = (float*)_aligned_malloc(
    sizeof(float) * (FILTER_ORDER + 4) * 4, 16);

  /* Place des zéro partout */
  memset(taps, 0, sizeof(float) * (FILTER_ORDER + 4) * 4);

  /* copie les valeurs, non décalées */
  memcpy(taps, a_taps, sizeof(float) * FILTER_ORDER);

  /* copie les valeurs, décalées de 1 */
  memcpy(taps + 1 + 1 * (FILTER_ORDER + 4), a_taps,
    sizeof(float) * FILTER_ORDER);

  /* copie les valeurs, décalées de 2 */
  memcpy(taps + 2 + 2 * (FILTER_ORDER + 4), a_taps,
    sizeof(float) * FILTER_ORDER);

  /* copie les valeurs, décalées de 3 */
  memcpy(taps + 3 + 3 * (FILTER_ORDER + 4), a_taps,
    sizeof(float) * FILTER_ORDER);
```

Code 4-2 : Alignement de valeurs pondérées.

Maintenant, il est temps d'entrer réellement dans le code assembleur. La première étape est de préparer les registres utilitaires pour pointer aux bons endroits mémoire et d'avoir les compteurs initialisés correctement.

```
mov     ecx, a_outputsLength  /* place le nombre de données à traiter */
sal     ecx, 2                /* calcul le nombre de byte necessaire */
mov     esi, a_inputs       /* place le vecteur d'entrée */
add     esi, ecx            /* calcul le pointeur de fin d'entree */
mov     edx, taps           /* place le pointeur sur les valeurs pondérés */
add     esi, 64             /* les 16 dernieres valeurs */
mov     edi, a_output       /* place le vecteur de sortie */
add     edi, ecx            /* calcul le pointeur de fin de sortie */
neg     ecx                 /* utilisé comme offset de départ */
```

Code 4-3 : Initialisation registre pour RIF.

Le Code 4-4 effectue le calcul complet. Il correspond aux boucles `for` imbriqués dans le Code 4-1.

```
movaps      xmm0, [esi + ecx]       /* xmm0 = in[n+3:n+0] */
mov         eax, edx                /* pointeur aux taps */
add         eax, 64                 /* pointeur à c0_0 */
movaps      xmm4, [eax + 160]       /* xmm4 = c2_0 */
movaps      xmm2, [eax + 240]       /* xmm2 = c3_0 */
movaps      xmm1, xmm0              /* xmm1 = in[n+3:n+0] */
mulps       xmm0, [eax + 80]        /* xmm0 = in[n+3:n+0] * c1_0 */
mulps       xmm4, xmm1              /* xmm4 = in[n+3:n+0] * c2_0 */
mulps       xmm2, xmm1              /* xmm2 = in[n+3:n+0] * c3_0 */
loop1:
prefetcht0 [esi + ecx + 16]         /* placement en cache des données */
movaps      xmm6, xmm0              /* xmm6 = in[n+3:n+0] * c1_0 */
movaps      xmm0, [esi + ecx - 16] /* xmm0 = in[n-1:n-4] */
movaps      xmm5, xmm4              /* xmm5 = in[n+3:n+0] * c2_0 */
movaps      xmm4, [eax - 16]        /* xmm4 = c0_1 */
movaps      xmm3, xmm2              /* xmm3 = in[n+3:n+0] * c3_0 */
movaps      xmm7, [eax + 240 - 16] /* xmm7 = c3_1 */
mulps       xmm4, xmm1              /* xmm4 = in[n+3:n+0] * c0_1 */
movaps      xmm1, xmm0              /* xmm1 = in[n-1:n-4] */
mulps       xmm0, [eax + 80 - 16]  /* xmm0 = in[n-1:n-4] * c1_1 */
movaps      xmm2, xmm4              /* xmm2 = in[n+3:n+0] * c0_1 */
movaps      xmm4, [eax + 160 - 16] /* xmm4 = c2_1 */
mulps       xmm7, xmm1              /* xmm7 = in[n-1:n-4] * c3_1 */
addps       xmm6, xmm0              /* xmm6 = in[n+3:n+0] * c1_0 +
                                     *        in[n-1:n-4] * c1_1 */
movaps      xmm0, [esi + ecx - 32] /* xmm0 = in[n-5:n-8] */
mulps       xmm4, xmm1              /* xmm4 = in[n-1:n-4] * c2_1 */
addps       xmm5, xmm4              /* xmm5 = in[n+3:n+0] * c2_0 +
                                    /*        in[n-1:n-4] * c2_1 */
movaps      xmm4, [eax - 32]        /* xmm4 = c0_2 */
addps       xmm3, xmm7              /* xmm3 = in[n+3:n+0] * c3_0 +
                                     *        in[n-1:n-4] * c3_1 */
movaps      xmm7, [eax + 240 - 32] /* xmm7 = c3_2 */
mulps       xmm4, xmm1              /* xmm4 = in[n-1:n-4] * c0_2 */
movaps      xmm1, xmm0              /* xmm1 = in[n-5:n-8] */
mulps       xmm0, [eax + 80 - 32]  /* xmm0 = in[n-5:n-8] * c1_2 */
addps       xmm2, xmm4              /* xmm2 = in[n+3:n+0] * c0_1 +
                                     *        in[n-1:n-4] * c0_2 */
movaps      xmm4, [eax + 160 - 32] /* xmm4 = c2_2 */
mulps       xmm7, xmm1              /* xmm7 = in[n-5:n-8] * c3_2 */
addps       xmm6, xmm0              /* xmm6 = in[n+3:n+0] * c1_0 +
                                     *        in[n-1:n-4] * c1_1 +
                                     *        in[n-5:n-8] * c1_2 */
movaps      xmm0, [esi + ecx - 48] /* xmm0 = in[n-9:n-12] */
mulps       xmm4, xmm1              /* xmm4 = in[n-5:n-8] * c2_2 */
addps       xmm5, xmm4              /* xmm5 = in[n+3:n+0] * c2_0 +
                                     *        in[n-1:n-4] * c2_1 +
                                     *        in[n-5:n-8] * c2_2 */
movaps      xmm4, [eax - 48]        /* xmm4 = c0_3 */
addps       xmm3, xmm7              /* xmm3 = in[n+3:n+0] * c3_0 +
                                     *        in[n-1:n-4] * c3_1 +
                                     *        in[n-5:n-8] * c3_2 */
movaps      xmm7, [eax + 240 - 48] /* xmm7 = c3_3 */
```

```
mulps     xmm4, xmm1              /* xmm4 = in[n-5:n-8] * c0_3 */
movaps    xmm1, xmm0              /* xmm1 = in[n-9:n-12] */
mulps     xmm0, [eax + 80 - 48]   /* xmm0 = in[n-9:n-12] * c1_3 */
addps     xmm2, xmm4              /* xmm2 = in[n+3:n+0] * c0_1 +
                                  *        in[n-1:n-4] * c0_2 +
                                  *        in[n-5:n-8] * c0_3 */
movaps    xmm4, [eax + 160 - 48] /* xmm4 = c2_3 */
mulps     xmm7, xmm1              /* xmm7 = in[n-9:n-12] * c3_3 */
addps     xmm6, xmm0              /* xmm6 = in[n+3:n] * c1_0 +
                                  *        in[n-1:n-4] * c1_1 +
                                  *        in[n-5:n-8] * c1_2 +
                                  *        in[n-9:n-12] * c1_3 */
movaps    xmm0, [esi + ecx - 64] /* xmm0 = in[n-13:n-16] */
mulps     xmm4, xmm1              /* xmm4 = in[n-9:n-12] * c2_3 */
addps     xmm5, xmm4              /* xmm5 = in[n+3:n] * c2_0 +
                                  *        in[n-1:n-4] * c2_1 +
                                  *        in[n-5:n-8] * c2_2 +
                                  *        in[n-9:n-12] * c2_3 */
movaps    xmm4, [eax - 64]        /* xmm4 = c0_4 */
addps     xmm3, xmm7              /* xmm3 = in[n+3:n] * c3_0 +
                                  *        in[n-1:n-4] * c3_1 +
                                  *        in[n-5:n-8] * c3_2 +
                                  *        in[n-9:n-12] * c3_3 */
mulps     xmm4, xmm1              /* xmm4 = in[n-9:n-12] * c0_4 */
movaps    xmm1, xmm0              /* xmm1 = in[n-13:n-16] */
mulps     xmm0, [eax + 80 - 64]   /* xmm0 = in[n-13:n-16] *  */c1_4
movaps    xmm7, [eax + 240 - 64] /* xmm7 = c3_4 */
addps     xmm2, xmm4              /* xmm2 = in[n+3:n] * c0_1 +
                                  *        in[n-1:n-4] * c0_2 +
                                  *        in[n-5:n-8] * c0_3 +
                                  *        in[n-9:n-12] * c0_4 */
movaps    xmm4, [eax + 160 - 64] /* xmm4 = c2_4 */
mulps     xmm7, xmm1              /* xmm7 = in[n-13:n-16] * c3_4 */
addps     xmm6, xmm0              /* xmm6 = in[n+3:n] * c1_0 +
                                  *        in[n-1:n-4] * c1_1 +
                                  *        in[n-5:n-8] * c1_2 +
                                  *        in[n-9:n-12] * c1_3 +
                                  *        in[n-13:n-16] * c1_4 */
mulps     xmm4, xmm1              /* xmm4 = in[n-13:n-16] * c2_4 */
addps     xmm5, xmm4              /* xmm5 = in[n+3:n] * c2_0 +
                                  *        in[n-1:n-4] * c2_1 +
                                  *        in[n-5:n-8] * c2_2 +
                                  *        in[n-9:n-12] * c2_3 +
                                  *        in[n-13:n-16] * c2_4 */
movaps    xmm0, [esi + ecx + 16] /* xmm0 = in[n+3:n] */
addps     xmm3, xmm7              /* xmm3 = in[n+3:n] * c3_0 +
                                  *        in[n-1:n-4] * c3_1 +
                                  *        in[n-5:n-8] * c3_2 +
                                  *        in[n-9:n-12] * c3_3 +
                                  *        in[n-13:n-16] * c3_4 */
movaps    xmm4, xmm6              /* xmm4 = in[n+3:n] * c1_0 +
                                  *        in[n-1:n-4] * c1_1 +
                                  *        in[n-5:n-8] * c1_2 +
```

```
                                    *          in[n-9:n-12] * c1_3 +
                                    *          in[n-13:n-16] * c1_4 */
movaps      xmm1, xmm5              /* xmm1 = in[n+3:n] * c2_0 +
                                    *          in[n-1:n-4] * c2_1 +
                                    *          in[n-5:n-8] * c2_2 +
                                    *          in[n-9:n-12] * c2_3 +
                                    *          in[n-13:n-16] * c2_4 */
unpcklps    xmm4, xmm3              /* xmm4 = 3.1, 6.1, 3.0, 6.0 */
unpckhps    xmm5, xmm2              /* xmm5 = 2.3, 5.3, 2.2, 5.2 */
mov         eax,  edx               /* pointeur aux taps */
add         eax,  64                /* pointeur à c0_0 */
unpcklps    xmm1, xmm2              /* xmm1 = 2.1, 5.1, 2.0, 5.0 */
movaps      xmm2, [eax + 240)       /* xmm2 = c3_0 */
unpckhps    xmm6, xmm3              /* xmm6 = 3.3, 6.3, 3.2, 6.2 */
addps       xmm6, xmm4              /* xmm6 = 3.1+3.3, 6.1+6.3,
                                    *          3.0+3.2, 6.0+6.2 */
movaps      xmm4, [eax + 160]       /* xmm4 = c2_0 */
addps       xmm5, xmm1              /* xmm5 = 2.1+2.3, 5.1+5.3,
                                    *          2.0+2.2, 5.0+5.2 */
movaps      xmm7, xmm6              /* xmm7 = 3.1+3.3, 6.1+6.3,
                                    *          3.0+3.2, 6.0+6.2 */
movaps      xmm1, xmm0              /* xmm1 = in[n+3:n] */
unpcklps    xmm7, xmm5              /* xmm7 = 2.0+2.2, 3.0+3.2,
                                    *          5.0+5.2, 6.0+6.2 */
mulps       xmm2, xmm1              /* xmm2 = in[n+3:n] * c3_0 */
unpckhps    xmm6, xmm5              /* xmm6 = 2.1+2.3, 3.1+3.3,
                                    *          5.1+5.3, 6.1+6.3 */
mulps       xmm0, [eax + 80]        /* xmm0 = in[n+3:n] * c1_0 */
addps       xmm7, xmm6              /* xmm7 = 2.x, 3.x, 5.x, 6.x
                                    *       = out[n+3], out[n+2],
                                    *          out[n+1], out[n] */
mulps       xmm4, xmm1              /* xmm4 = in[n+3:n] * c2_0 */
movntps     [edi+ecx], xmm7         /* sauvegarde des résultats */
add         ecx,  16                /* chargement des valeurs suivantes */
jnz         loop1                   /* bouclage */
```

Code 4-4 : Filtre RIF en assembleur.

4.2.3 Implantation psC

Enfin, le Code 4-5 montre ce qui est nécessaire d'écrire dans le langage psC.

```
component FIR {
  in  active  floatVector_100000 a_inputs,
  in  passive floatVector_100000 a_taps,
  out active  floatVector_100000 a_outputs)
{
  DoFIR(0) on a_inputs
  {
    index i;
    index j;
    a_outputs[i] := sum(a_taps[j] * a_inputs[i + j + 1]);
  }
}
```

Code 4-5 : Filtre RIF en psC.

4.2.4 Résultats

Chaque implantation a été testée avec un million de valeurs désiré dans le vecteur de sortie. Le Tableau 4-2 montre les résultats. Les itérations 1 et 2 sont données à titre informatif uniquement, ils n'entrent pas dans le total. Comme tous s'y attendaient, le compilateur d'Intel obtient un résultat plus performant que celui de Microsoft, et le travail d'un expert donne un résultat encore plus performant. Le psC quand à lui, se situe entre le résultat obtenu par Intel et celui d'un expert, démontrant la validité de l'approche.

Itération	Microsoft	Intel	Assembleur	psC
1	*0.259819*	*0.031590*	*0.009*	*0.016774*
2	*0.130489*	*0.030290*	*0.009*	*0.016796*
3	0.130429	0.030260	0.009	0.016749
4	0.131751	0.030249	0.009	0.016772
5	0.130475	0.030253	0.009	0.016900
6	0.130598	0.030252	0.009	0.016985
7	0.130418	0.030253	0.009	0.016733
8	0.130348	0.030256	0.009	0.016886
9	0.130494	0.030254	0.009	0.016742
10	0.130606	0.030256	0.009	0.016770
11	0.130985	0.030255	0.009	0.016781
12	0.130963	0.030263	0.009	0.016930
Total	**1.307066**	**0.302552**	**0.098018**	**0.168248**

Tableau 4-2 : Résultats filtre FIR.

4.3 Saturation alpha

La saturation alpha est une opération du domaine du traitement de l'image. Pour la comprendre, il est nécessaire d'effectuer une mise en contexte. Un pixel est un point visible d'un utilisateur. C'est le plus petit artefact visible dans une image. Il peut être décrit de plusieurs manières, mais deux caractéristiques principales sont toujours présentes, que ce soit implicitement ou explicitement, sont la couleur et la transparence.

La représentation qui nous intéresse représente un pixel avec un encodage de trois couleurs primaires (rouge, vert et bleu). Chaque couleur primaire est représenté avec 8 bits. Enfin, 8 bits sont utilisé pour représenter une valeur alpha. Cette valeur sera utilisée comme valeur de saturation, du nom de la technique saturation alpha.

L'opération de saturation est conceptuellement simple. Pour chaque couleur primaire, la valeur finale est le minimum entre la couleur primaire originale ou la valeur alpha. La Figure 4-2 montre des exemples.

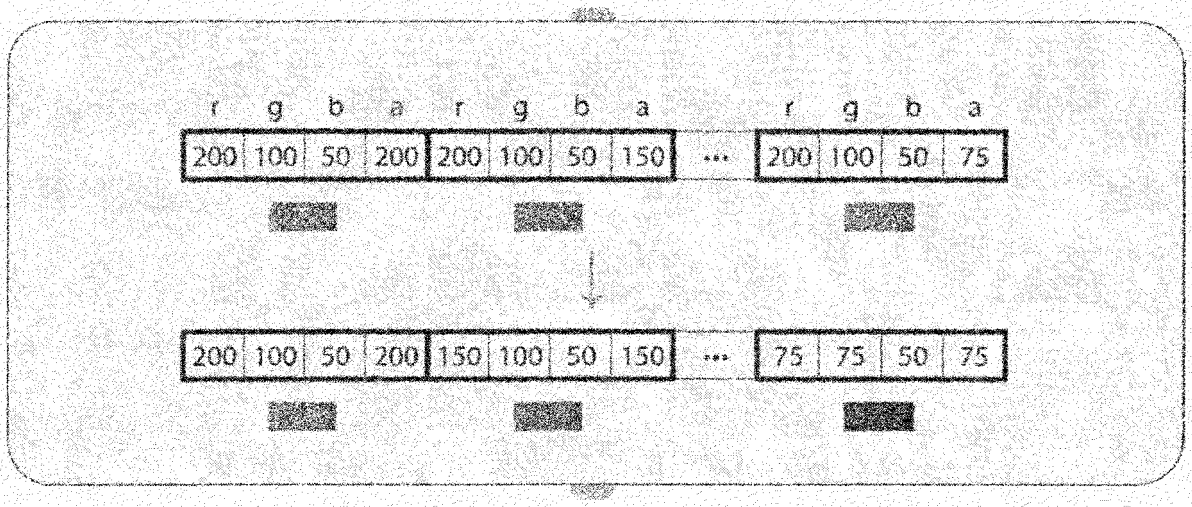

Figure 4-2: Exemples de saturation alpha.

Les entrées dans l'algorithme sont :

- Un vecteur d'entré `a_inputs` contenant les valeurs des pixels de la source.
- Un vecteur de sorti `a_outputs` dans lequel les résultats seront placés.
- Le nombre pixel `a_length` à traiter.

4.3.1 Implantation C

L'implantation en langage C est simple. Elle consiste en l'exécution d'une boucle qui effectue le travail, `a_length` fois. Le Code 4-6 en est l'implémentation.

```
typedef unsigned char Uint8;

typedef struct rgba
{
  Uint8 r;
  Uint8 g;
  Uint8 b;
  Uint8 a;
} rgba;

void alpha_saturation(
  rgba * a_inputs,
  rgba * a_outputs,
  int a_length)
{
  for (int i = 0; i < a_length; ++i)
  {
    a_outputs[i].a = a_inputs[i].a;
    a_outputs[i].r = min(a_inputs[i].r, a_inputs[i].a);
    a_outputs[i].g = min(a_inputs[i].g, a_inputs[i].a);
    a_outputs[i].b = min(a_inputs[i].b, a_inputs[i].a);
  }
}
```

Code 4-6 : Saturation alpha en C.

4.3.2 Implantation Assembleur

Comme pour le filtre RIF, la mise en œuvre est une boucle similaire, mais plus complexe, étant en langage assembleur. Le nombre de pixel traité doit être divisible par quatre. Les instructions SSE2 ont été utilisées, ce qui implique d'avoir des adresse mémoire divisible par 16, tout comme le filtre FIR. Le Code 4-7 est l'implantation de la boucle de calcul.

```
/* début de bouclage */
forloop:
prefetchnta [esi + 16*2]

/* calcul */
movq        mm0, mmword ptr [esi]      /* chargement A et B */
pshufw      mm1, mm0, 0xF5             /* en caulcul du alpha A et B */
pand        mm1, mm7                   /* en caulcul du alpha A et B */
movq        mm3, mmword ptr [esi + 8] /* pixel C et D */
pshufw      mm4, mm3, 0xF5             /* en caulcul du alpha C et D */
movq        mm2, mm1                   /* en caulcul du alpha A et B */
psrld       mm1, 8                     /* en caulcul du alpha A et B */
por         mm1, mm2                   /* alpha A et B */
pminub      mm0, mm1                   /* resultat A et B */
movq        mmword ptr [edi], mm0      /* sauvegarde A et B */
pand        mm4, mm7                   /* en caulcul du alpha C et D */
movq        mm5, mm4                   /* en caulcul du alpha C et D */
psrld       mm4, 8                     /* en caulcul du alpha C et D */
por         mm4, mm5                   /* alpha C et D */
pminub      mm3, mm4                   /* resultat C et D */
movq        mmword ptr [edi + 8], mm3 /* sauvegarde C et D*/

/* fin de bouclage */
add         eax, 16
cmp         eax, ecx
jl          forloop
```

Code 4-7 : Saturation alpha en assembleur.

4.3.3 Implantation psC

Enfin, le Code 4-8 montre ce qui est nécessaire d'écrire dans le langage psC.

```
component AlphaSaturation (
  in active rgba_100000 a_inputs,
  in passive index a_length,
  out active rgba_100000 a_outputs)
{
  DoWork(0) on cnt
  {
    index i;
    a_outputs[a_length][i] :=
      min(a_inputs[a_length][i], a_inputs[a_length][3]);
  }
}
```

Code 4-8 : Saturation alpha en psC.

4.3.4 Résultats

Chaque implantation a été testée avec un cent mille pixels. Le Tableau 4-3 montre les résultats. Les itérations 1 et 2 sont listé à titre informatif uniquement, ils n'entrent pas dans le total. Pour les compilateurs C, le compilateur d'Intel obtient un résultat moins performant que celui de Microsoft. Le travail d'un expert donne encore le résultat le plus performant. Le compilateur psC produit un code aussi performant que celui de l'expert.

Itération	Microsoft	Intel	Assembleur	psC
1	*0.001291*	*0.002231*	*0.000289*	*0.000309*
2	*0.001064*	*0.002036*	*0.000256*	*0.000288*
3	0.001104	0.002034	0.000264	0.000262
4	0.001063	0.002063	0.000256	0.000262
5	0.001060	0.002050	0.000256	0.000262
6	0.001055	0.002032	0.000257	0.000262
7	0.001087	0.002031	0.000302	0.000261
8	0.001060	0.002033	0.000255	0.000261
9	0.001052	0.002051	0.000256	0.000273
10	0.001091	0.002032	0.000257	0.000262
11	0.001054	0.002032	0.000265	0.000261
12	0.001059	0.002037	0.000256	0.000262
Total	**0.010666**	**0.020394**	**0.002625**	**0.002658**

Tableau 4-3 : Résultats saturation alpha.

4.4 Multiplication matricielle

La multiplication matricielle est une opération utilisée dans plusieurs domaines, dont le traitement de l'image et les calculs scientifiques. Elle consiste en la somme de la multiplication des éléments d'une rangée d'une première matrice par les éléments d'une colonne d'une seconde matrice pour fabriquer une troisième matrice. Enfin, une dernière

matrice est ajoutée à ce résultat. Pour un élément i,j de la matrice résultat Y, la valeur sera l'élément i,j la matrice B additionné de la somme sur k de $A_{i,k}$ par $X_{k,j}$. Le tout est connu sous la formule $Y = AX+B$. La Figure 4-3 en fait l'illustration.

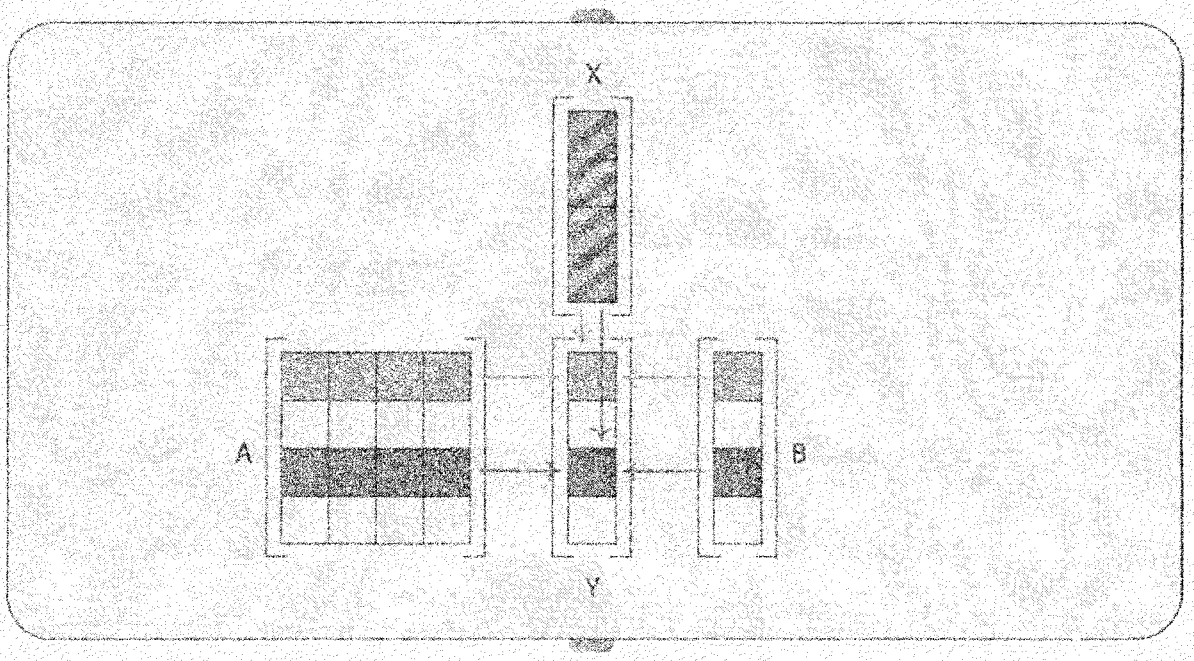

Figure 4-3: Exemples de multiplication matricielle.

Pour l'exemple, les indices i et j vont varier de 1 à 10, et l'indice k reste libre.

Les entrées dans l'algorithme sont :

Trois vecteurs d'entrée `a_A`, `a_B` et `a_X`, correspondant aux trois tableaux sources.

- Un vecteur de sorti `a_Y` dans lequel les résultats seront placés.
- Le nombre pixel `a_count` à traiter.

4.4.1 Implantation C

L'implantation en langage C consiste en trois boucles imbriquées. Les deux boucles extérieures font varier les indices `i` et `j` pour balayer toutes les positions de la matrice résultante. La boucle interne permet de balayer les matrices d'entrées pour effectuer la somme des multiplications. Le Code 4-9 en est l'implémentation.

```
for (Uint32 k = 0; k < count; ++k)
{
  const Uint32 Xindex = k * 10;
  for (Uint32 i = 0; i < 10; ++i)
  {
    const Uint32 index = Xindex + i;
    const Uint32 Aindex = i * 10;

    double accum = B[index];
    for (Uint32 j = 0; j < 10; ++j)
    {
      accum += A[Aindex + j] * X[Xindex + j];
    }
    Y[index] = accum;
  }
}
```

Code 4-9 : Multiplication matricielle en C.

4.4.2 Implantation Assembleur

Comme pour les autres exemples, la mise en œuvre en assembleur suit la même logique que celle en langage C, mais est plus complexe. Les instructions SSE2 ont été utilisées, ce qui implique d'avoir des adresse mémoire divisible par 16, tout comme le filtre FIR. Le Code 4-10 montre le traitement principal en langage assembleur.

```
/* edi: pointeur sur le tableau de résultats (Y).
 * esi: pointeur sur le tableau des données (X).
 * esp+8: taille d'une rangée.
 * esp+4: pointeur sur la fin du tableau des résultats.
 * edx: pointeur sur le tableau B.
 */

/* Pour chaque matrice X */
count_loop:
mov     eax, [esp+8]
add     eax, edi
mov     [esp], eax
mov     eax, A

/* Effectue le calcul matriciel */
big_loop:
movsd   xmm7, [edx]         /* initialise les sommes partielles */
add     edx, 8              /* prépare la rangée suivante
xor     ecx, ecx            /* réinitialise le compteur */

small_loop:
movaps  xmm0, [esi+ecx*8] /* Charge deux valeur de la rangée */
mulpd   xmm0, [eax+ecx*8] /* Multiplie par les taps correspondant */
addpd   xmm7, xmm0          /* Accumule */
/* Valeurs suivantes de la rangée */
add     ecx, 2
cmp     ecx, 10
jl      small_loop

/* Addition des deux somme partielles */
movaps  xmm6, xmm7
unpckhpd xmm6, xmm6
addpd   xmm7, xmm6
/* Sauvegaqrde du résultat */
movsd   [edi], xmm7

/* Rangée suivante */
add     edi, 8
add     eax, [esp+8]
cmp     edi, [esp]
jl      big_loop

/* Matrice suivante */
cmp     edi, [esp+4]
jl      count_loop
```

Code 4-10 : Multiplication matricielle en assembleur.

4.4.3 Implantation psC

Enfin, le Code 4-11 montre ce qui est nécessaire d'écrire dans le langage psC.

```
component matrix (
  in passive mat_10_10 A,
  in passive vec_100000_10 X,
  in passive vec_100000_10 B,
  in active index cnt,
  in passive index i,
  in passive index j,
  out active vec_100000_10 Y)
{
  DoMult(0) on cnt
  {
    Y[cnt][i] := sum(A[i][j] * X[cnt][j]) + B[cnt][i];
  }
}
```

Code 4-11 : Multiplication matricielle en psC.

4.4.4 Résultats

Chaque implantation a été testée avec une matrice A de 10 par 10, et une matrice X et B de 10 par un million. Le Tableau 4-4 montre les résultats. Les itérations 1 et 2 sont données à titre informatif uniquement, ils n'entrent pas dans le total. Pour les compilateurs C, le compilateur d'Intel obtient un résultat moins performant que celui de Microsoft. Le travail d'un expert donne encore le résultat le plus performant. Le psC quand à lui, se situe entre le résultat obtenu par le compilateur le plus performant et celui d'un expert, mais très proche de celui des compilateurs C. Il reste donc place à amélioration dans la génération de code de cet algorithme.

Itération	Microsoft	Intel	Assembleur	psC
1	*0.041069*	*0.044026*	*0.018980*	*0.036165*
2	*0.035482*	*0.037763*	*0.01876*	*0.035022*
3	0.034776	0.038610	0.018739	0.03516
4	0.035350	0.037755	0.018777	0.034629
5	0.034756	0.038128	0.018716	0.034743
6	0.034941	0.040748	0.019187	0.034703
7	0.035127	0.040702	0.018782	0.034876
8	0.034987	0.039064	0.018798	0.034682
9	0.034779	0.037979	0.018722	0.034929
10	0.035211	0.037933	0.018774	0.034511
11	0.035370	0.038339	0.018780	0.034583
12	0.034811	0.038193	0.018949	0.035353
Total	**0.350108**	**0.387450**	**0.188224**	**0.348168**

Tableau 4-4 : Résultats multiplication matricielle.

5

CONCLUSION

Ce travail se voulait une preuve qu'il était possible d'avoir un outil pour les programmeurs qui leurs permettrait d'utiliser la puissance offerte par les instructions data-parallèles de manières simple et efficace. Cet outil devrait utiliser les instructions data-parallèles partout où c'est pertinent et le programmeur ne devrait pas avoir à fournir d'effort supplémentaire pour ce faire. Le langage psC, initialement destiné à la programmation des FPGA, s'est avéré efficace pour la programmation des processeurs modernes et l'utilisation des instructions data-parallèles du processeur. L'utilisation d'un langage destiné à la programmation matérielle, pour la programmation data parallèle des processeurs est un concept unique.

En observant les résultats, on note que le celui obtenu par le code psC est toujours entre celui obtenu par un expert (assembleur) et celui obtenu par les compilateurs C. Ceci démontre qu'il est possible de générer du code efficace à partir d'un langage HL-HDL. Le Tableau 5-1 montre un comparatif des performances et la dernière ligne indique la performance relative globale par rapport à l'implémentation en assembleur. Il aurait été intéressant d'effectuer la comparaison avec l'implémentation de valarray d'Intel.

Test	Microsoft	Intel	Assembleur	psC
Filtre RIF	1.307066 7%	0.302552 32%	0.098018 100%	0.168248 58%
Saturation alpha	0.010666 25%	0.020394 13%	0.002625 100%	0.002658 99%
Multiplication matricielle	0.350108 54%	0.387450 49%	0.188224 100%	0.348168 54%

Tableau 5-1 : Sommaire des résultats.

En conclusion, ce travail de recherche a démontré que le programmeur peut exprimer l'algorithme de son choix dans un langage HL-HDL et l'utiliser sur PC. Le code généré bénéficie des instructions data-parallèles pour améliorer les performances. Le code psC est même plus simple que l'implémentation en C. Le gain en performance de l'implémentation psC est présent pour tous les cas étudié, et se rapproche de l'implémentation assembleur qui est le maximum atteignable.

La version actuelle pourrait être améliorée de deux manières. Premièrement, son intégration à la programmation classique, comme le langage C, devrait être améliorée. De plus, d'autres « patrons » pourraient être développés afin de supporter d'avantage de cas d'usages et d'opérations data-parallèles.

En final, nous pouvons conclure que la sémantique naturellement parallèle du langage psC permet, grâce à ses nouvelles instructions data-parallèles, de programmer efficacement le matériel (FPGA) et les processeurs actuels avec architecture data-parallèle..

BIBLIOGRAPHIE

[Advanced Micro Devices, 2000] Advanced Micro Devices, mars 2000. AMD Extensions to the 3DNow! and MMX Instruction Sets Manual.

[AFNOR, 2000] AFNOR, avril 2000. Fixing valarray for RealWorld Use. International standardization working group for the programming language C++, Tokyo.

[Beekley, 2005] Beekley, J. , mars 2005. AN501: Latency Settings and their Impact on Memory Performance. Corsair Memory.

[Blelloch, 1994] Blelloch, G. E., juillet 1994. Reading List on Parallel Programming Languages.

[Blelloch, 2010] Blelloch, G. E., décembre 2010. NESL: A Parallel Programming Language. http://www.cs.cmu.edu/~scandal/nesl.html.

[Blelloch et al., 1993] Blelloch, G. E., Chatterjee, S. , Hardwick, J. C., Sipelstein, J. et Zagha, M. , 1993. Implementation of a Portable Nested Data-Parallel Language. 102-111, 4th, San Diego.

[Boug´e et al., 1996] Boug´e, L. , Cachera, D. , Le Guyadec, Y. , Utard, G. et Virot, B. , 1996. Formal validation of data-parallel programs: a two-component assertional proof system for a simple language. Institut National de Recherche en Informatique et en Automatique.

[Brian W. Kernighan, 1988] Brian W. Kernighan, D. Ritchie, 1988. The C Programming Language, Second Edition, 2nd. Prentice Hall, 274 p.

[Conte et al., 2000] Conte, E. , Tommesani, S. et Zanichelli, F. , 2000. The long and winding raod to high-performance image processing with MMX/SSE. Dipartimento di Ingegneria dell'Informazinoe, Università di Parma, 302-310.

[Darlington et al., 1995] Darlington, J. , Guo, Y. et To, H. Wing, juillet 1995. Structured Parallel Programming: Theory meets Practice. London.

[Dekeysek et al., 2010] Dekeysek, J. , Kokoszko, B. , Levaire, J. et Marquet, P. , septembre 2010. IDOLE: Irregular structures in data-parallel languages. http://www.lifl.fr/west/idole/.

[Flynn, 1972] Flynn, M. J., 1972. Some Computer Organizations and Their Effectiveness. IEEE Trans. Comput., C-21, 948.

[Flynn, 1972] Flynn, M. J., 1972. Some Computer Organizations and Their Effectiveness. IEEE Trans. Comput., C-21, 948.

[Gordon Willhofl, 1995] Gordon Willhofl, R. , 1995. DPL: A Data Parallel Language for the Expression and Execution of General Parallel Algorithms. APL Quote Quad, 26, 1, 12-13.

[Graham, 1995] Graham, P. , 1995. ANSI Common LISP. Prentice Hall, 432 p.

[Hammarlund et Lisper, 1993] Hammarlund, P. et Lisper, B. , 1993. Data Parallel Programming: A Survey and a Proposal for a New Model. Departement of Teleinformatics, Royal Institute of Technology.

[Hatcher et al., 1991] Hatcher, P. J., Quinn, M. J., Anderson, R. J., Lapadulal, A. J., Seevers, B. K. et Bennett, A. F., 1991. Architecture-Independent Scientific Programming in Dataparallel C: Three Case Studies. 208-217, Principles & Practice of Parallel Programming, Williamsburg.

[Hillis et Guy L. Steele, 1986] Hillis, W. Daniel et Guy L. Steele, J. , 1986. Data Parallel Algorithms. Communications of the ACM, 29, 12, 1170-1183.

[Institute for System Programming, 2000] Institute for System Programming, 2000. The mpC Programming Language Specification. Russian Academy Of Sciences.

[Intel Corporation, 1966, a] Intel Corporation, mars 1966. Using MMX Instructions to Implement Bilinear Interpolation of Video RGB Values.

[Intel Corporation, 1996, b] Intel Corporation, mars 1996. Advanced Procedural Texturing Using MMX Technology.

[Intel Corporation, 1996, c] Intel Corporation, mars 1996. AGP and 3D Graphics Software.

[Intel Corporation, 1996, d] Intel Corporation, mars 1996. An Efficient Vector/Matrix Multiply Routine using MMX Technology.

[Intel Corporation, 1996, e] Intel Corporation, mars 1996. Color Conversion from YUV12 to RGB Using Intel MMX Technology.

[Intel Corporation, 1996, f] Intel Corporation, mars 1996. How to use floating-point or MMX instructions in Ring 0 or a VxD under Windows 95.

[Intel Corporation, 1996, g] Intel Corporation, mars 1996. Implementing Fractals with MMX Technology.

[Intel Corporation, 1996, h] Intel Corporation, mars 1996. Intel Architecture MMX Technology - Programmer's Reference Manual.

[Intel Corporation, 1996, i] Intel Corporation, mars 1996. MMX Technology Developers Guide.

[Intel Corporation, 1996, j] Intel Corporation, mars 1996. MMX Technology for 3D Rendering.

[Intel Corporation, 1996, k] Intel Corporation, mars 1996. MMX Technology Technical Overview.

[Intel Corporation, 1996, l] Intel Corporation, mars 1996. New EMMS Usage Guidelines.

[Intel Corporation, 1996, m] Intel Corporation, mars 1996. Survey of Pentium Processor Performance Monitoring Capabilities & Tools.

[Intel Corporation, 1996, n] Intel Corporation, mars 1996. Using MMX Instructions for 3D Bilinear Texture Mapping.

[Intel Corporation, 1996, o] Intel Corporation, mars 1996. Using MMX Instructions for Procedural Texture Mapping.

[Intel Corporation, 1996, p] Intel Corporation, mars 1996. Using MMX Instructions in a Fast iDCT Algorithm for MPEG Decoding.

[Intel Corporation, 1996, q] Intel Corporation, mars 1996. Using MMX Instructions to Compute a 16-Bit Vector.

[Intel Corporation, 1996, r] Intel Corporation, mars 1996. Using MMX Instructions to Compute the 2x2 Haar Transform.

[Intel Corporation, 1996, s] Intel Corporation, mars 1996. Using MMX Instructions to Compute the Absolute Difference in Motion Estimation.

[Intel Corporation, 1996, t] Intel Corporation, mars 1996. Using MMX Instructions to Compute the L1 Norm Between Two 16-bit Vectors.

[Intel Corporation, 1996, u] Intel Corporation, mars 1996. Using MMX Instructions to Compute the L2 Norm Between Two 16-Bit Vectors.

[Intel Corporation, 1996, v] Intel Corporation, mars 1996. Using MMX Instructions to Convert 24-Bit True Color Data to 16-Bit High Color.

[Intel Corporation, 1996, w] Intel Corporation, mars 1996. Using MMX Instructions to Convert RGB To YUV Color Conversion.

[Intel Corporation, 1996, x] Intel Corporation, mars 1996. Using MMX Instructions to Get Bits From a Data Stream.

[Intel Corporation, 1996, y] Intel Corporation, mars 1996. Using MMX Instructions to Implement 2D Sprite Overlay.

[Intel Corporation, 1996, z] Intel Corporation, mars 1996. Using MMX Instructions to implement 2X 8-bit Image Scaling.

[Intel Corporation, 1996, aa] Intel Corporation, mars 1996. Using MMX Instructions to Implement a 1/3T Equalizer.

[Intel Corporation, 1996, ab] Intel Corporation, mars 1996. Using MMX Instructions to Implement a Column Filter.

[Intel Corporation, 1996, ac] Intel Corporation, mars 1996. Using MMX Instructions to Implement a Modem Baseband Canceler.

[Intel Corporation, 1996, ad] Intel Corporation, mars 1996. Using MMX Instructions to Implement a Modem Passband Canceler.

[Intel Corporation, 1996, ae] Intel Corporation, mars 1996. Using MMX Instructions to Implement a Row Filter Algorithm.

[Intel Corporation, 1996, af] Intel Corporation, mars 1996. Using MMX Instructions to Implement a SchurWeiner Filter.

[Intel Corporation, 1996, ag] Intel Corporation, mars 1996. Using MMX Instructions to Implement a Synthesis Sub-Band Filter for MPEG Audio Decoding.

[Intel Corporation, 1996, ah] Intel Corporation, mars 1996. Using MMX Instructions to Implement a Video Loop Filter.

[Intel Corporation, 1996, ai] Intel Corporation, mars 1996. Using MMX Instructions to Implement Audio Echo Sound Effects.

[Intel Corporation, 1996, aj] Intel Corporation, mars 1996. Using MMX Instructions to Implement Data Alignment.

[Intel Corporation, 1996, ak] Intel Corporation, mars 1996. Using MMX Instructions to Implement Median Filter.

[Intel Corporation, 1996, al] Intel Corporation, mars 1996. Using MMX Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback.

[Intel Corporation, 1996, am] Intel Corporation, mars 1996. Using MMX Instructions to Implement the G.728 Codebook Search.

[Intel Corporation, 1996, an] Intel Corporation, mars 1996. Using MMX Instructions to Implement the Gouraud Shading Algorithm.

[Intel Corporation, 1996, ao] Intel Corporation, mars 1996. Using MMX Instructions to Implement the Levinson-Durbin Algorithm.

[Intel Corporation, 1996, ap] Intel Corporation, mars 1996. Using MMX Instructions to Implement Viterbi Decoding.

[Intel Corporation, 1996, aq] Intel Corporation, mars 1996. Using MMX Instructions to Perform 16-Bit x 31-Bit Multiplication.

[Intel Corporation, 1996, ar] Intel Corporation, mars 1996. Using MMX Instructions to Perform 3D Geometry Transformations.

[Intel Corporation, 1996, as] Intel Corporation, mars 1996. Using MMX Instructions to Perform Complex 16-Bit FFT.

[Intel Corporation, 1996, at] Intel Corporation, mars 1996. Using MMX Instructions to Perform Simple Vector Operations.

[Intel Corporation, 1996, au] Intel Corporation, mars 1996. Using MMX Instructions to Transpose a Matrix.

[Intel Corporation, 1996, av] Intel Corporation, mars 1996. Using MMX Instructuions to Implement Alpha Blending.

[Intel Corporation, 1996, aw] Intel Corporation, mars 1996. Using MMX Technology Instructions to Compute a 16-Bit FIR Filter.

[Intel Corporation, 1996, ax] Intel Corporation, mars 1996. Using MMX Technology Instructions to Implement a 2/3T Equalizer.

[Intel Corporation, 1996, ay] Intel Corporation, mars 1996. Using MMX™ Technology Instructions to Compute a 16-Bit FIR Filter.

[Intel Corporation, 1999, az] Intel Corporation, janvier 1999. 32-bit Floating Point Real & Complex 16-Tap FIR Filter Implemented Using Streaming SIMD Extensions. Intel Corporation.

[Intel Corporation, 1999, ba] Intel Corporation, janvier 1999. Antialiasing Implemented Using Streaming SIMD Extensions.

[Intel Corporation, 1999, bb] Intel Corporation, janvier 1999. Data Alignment and Programming Issues for the Streaming SIMD Extensions.

[Intel Corporation, 1999, bc] Intel Corporation, janvier 1999. FIR and IIR Filtering using Streaming SIMD Extensions.

[Intel Corporation, 1999, bd] Intel Corporation, janvier 1999. Phong Equation Using Streaming SIMD Extensions.

[Intel Corporation, 1999, be] Intel Corporation, janvier 1999. RGB Alpha Saturation Using Streaming SIMD Extensions.

[Intel Corporation, 1999, bf] Intel Corporation, janvier 1999. Software Conventions for the Streaming SIMD Extensions.

[Intel Corporation, 1999, bg] Intel Corporation, juin 1999. Streaming SIMD Extensions - Matrix Multiplication.

[Intel Corporation, 2009, bh] Intel Corporation, novembre 2009. Intel 64 and IA-32 Architectures Optimization Reference Manual.

[Intel Corporation, 2010, bi] Intel Corporation, mars 2010. Intel 64 and IA-32 Architectures Software Developer's Manual.

[Intel Corporation, 2010, bj] Intel Corporation, août 2010. Intel C++ Compiler 11.1 User and Reference Guides. Intel Corporation, http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/win/compiler_c/index.htm.

[Intel Corporation, 2010, bk] Intel Corporation, septembre 2010. MMX Technology Manuals and Application Notes. http://software.intel.com/en-us/articles/mmxt-technology-manuals-and-application-notes/.

[Jouret, 1991] Jouret, G. K., 1991. Compiling Functional Languages for SIMD Architectures. Department of Computing, Imperial College of Science, Technology & Medicine, 79-86.

[Keller et Chakravarty] Keller, G. et Chakravarty, M. M. T., s.d. On the Distributed Implementation of Aggregate Data Structures by Program Transformation.

[Klaiber et Levy, 1994] Klaiber, A. C. et Levy, H. M., 1994. A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs. 94-105, Proceedings of the 21st Annual International, Chicago, IEEE.

[Koelbel, 2010] Koelbel, C. , décembre 2010. . http://hpff.rice.edu/index.htm.

[Koelbel et Mehrotra, 1991] Koelbel, C. et Mehrotra, P. , 1991. Programming Data Parallel Algorithms on Distributed Memory Machines Using Kali. 414-423, Principles and Practice of Parallel Programming, Williamsburg.

[Kotz, 1995] Kotz, D. , 1995. A Data-Parallel Programming Library For Education (DAPPLE). 76-81, International Symposium on Computer Architecture, Santa Margherita Ligure, IEEE.

[Kumar, 1998] Kumar, A. , 1998. SSE2 Optimization – OpenGL Data Stream Case Study. Mt. Prospect.

[Levicki, 2012] Levicki, I. M., juin 2012. Array Clipping - Intel Software Network. http://software.intel.com/en-us/articles/array-clipping/.

[Lischner, 2003] Lischner, R. , 2003. C++ In a Nutshell. O'Reilly Media, 810 p.

[Loidl et al., 2003] Loidl, H. , Rubio, F. , Scaife, N. , Hammond, K. , Horiguchi, S. , Klusik, U. , Loogen, R. , Michaelson, G. , A, R. Pen, Priebe, S. , N, A. J. Rebo et Trinder, P. , 2003. Comparing Parallel Functional Languages: Programming and Performance. Kluwer Academic Publishers.

[Mannhart et Gautier, 1998] Mannhart, N. et Gautier, T. , juin 1998. Parallelism and Aldor: a first report of ∏it.

[McCarthy, 1960] McCarthy, J. , 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM.

[Microsoft, 2010] Microsoft, août 2010. /arch compiler option. Microsoft, http://msdn.microsoft.com/en-us/library/7t5yh4fd.aspx.

[Mittal et al., 1997] Mittal, M. , Peleg, A. et Weiser, U. , 1997. MMX Technology Architecture Overview. Intel Technology Journal, Q3, 1-12.

[Novakod Technologies Inc, 2005] Novakod Technologies Inc, 2005. K3 Programmer's reference manual. Saguenay, Québec.

[Novakod Technologies Inc, 2005] Novakod Technologies Inc, 2005. psC Programmer's reference manual. Saguenay, Québec.

[Peleg et al., 1997] Peleg, A. , Wilkie, S. et Weiser, U. , 1997. Intel MMX for Multimedia PCs. Communications of the ACM, 40, 1, 25-38.

[Petri, 1962] Petri, C. Adam, 1962. Communication with automata.

[Prins et Palmer, 1993] Prins, J. F. et Palmer, D. W., 1993. Transforming High-Level Data-Parallel Programs into Vector Operations. 119-128, Principles and Practice of Parallel Programming, San Diego, Association for Computing Machinery.

[Ramirez et al., 2001] Ramirez, A. , Larriaba-Pey, J. L. et Valero, M. , 2001. Instruction Fetch Architectures and Code Layout Optimizations. Proceedings of the IEEE, 89, 11, 1588-1609.

[Santavy et Labute, 2010] Santavy, M. et Labute, P. , décembre 2010. . http://www.chemcomp.com/journal/svl.htm.

[Schaffer et al., 2002] Schaffer, R. , Merker, R. et Catthoor, F. , 2002. Systematic Design of Progrmas with Sub-Word Parallelism. 1-6, Proceedings of the International Conference on Parallel Computing in Electrical Engineering, IEEE.

[Stalling, 2003] Stalling, W. , 2003. Computer Organization & Architecture. Prentice Hall, Upper Saddle River, New Jersey.

[Stallman, 2003] Stallman, R. M., octobre 2003. Using the GNU Compiler Collection. Boston.

[Stroustrup, 1997] Stroustrup, B. , 1997. The C++ Programming Language, 3rd Edition. Addison Wesley Publishing Company, Reading, Massachusetts.

[Subhlok et Yang, 1997] Subhlok, J. et Yang, B. , 1997. A New Model for Integrated Nested Task and Data Parallel Programming. 1-12, Principles and Practice of Parallel Programming, Las Vegas, Association for Computing Machinery.

[University of Washington, 2010] University of Washington, décembre 2010. ZPL. University of Washington, http://www.cs.washington.edu/research/zpl/home/index.html.

[Veldhuizen, 2006] Veldhuizen, T. , mars 2006. Blitz++ User's Guide. Free Software Foundation.

[Wikipedia, 2010] Wikipedia, septembre 2010. Von Neumann architecture. http://en.wikipedia.org/wiki/Von_Neumann_architecture.

[Wikipedia, 2012] Wikipedia, juin 2012. Parallel computing. http://en.wikipedia.org/wiki/Parallel_computing.

[WillAofl, 1995] WillAofl, R. Cordon, mai 1995. DPL: A Data Parallel Language for the Expression and Execution of General Parallel Algorithms. Endicott.