View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by HAL-Rennes 1

Application de la fusion de chemins de données à la synthèse d'architectures parallèles

Clément Guy

To cite this version:

Clément Guy. Application de la fusion de chemins de données à la synthèse d'architectures parallèles. Architectures Matérielles [cs.AR]. 2010. <dumas-00530700>

HAL Id: dumas-00530700

https://dumas.ccsd.cnrs.fr/dumas-00530700

Submitted on 29 Oct 2010

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

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

Université de Rennes 1
IFSIC
MRI parcours P1

Rapport de stage

Application de la fusion de chemin de données à la synthèse d'architectures parallèles

par

Clément Guy

Encadrant : Steven Derrien (Équipe CAIRN - INRIA)

Rennes, 2010

Table des matières

Introduction 1

1 État de l'art 3
- 1.1 Algorithmes sources et architectures cibles 3
  - 1.1.1 Nids de boucles à contrôle statique 3
  - 1.1.2 Réseaux réguliers de processeurs 4
- 1.2 Ordonnancement et allocation de nids de boucles 5
  - 1.2.1 Modèle polyédrique 5
  - 1.2.2 Ordonnancement et allocation avec le modèle polyédrique 7
  - 1.2.3 Environnement MMAlpha 9
- 1.3 Partage de ressources 9
  - 1.3.1 Partitionnement et temps multi-dimensionnel 10
  - 1.3.2 Fusion de chemins de données 12
  - 1.3.3 Partage de ressources au sein de réseaux réguliers 15

2 Contribution 17
- 2.1 Flot de synthèse 17
- 2.2 Polyhedral Data-Path Graph 17
  - 2.2.1 Définitions 19
  - 2.2.2 Exemple de PDPG détaillé 19
- 2.3 Partage de ressources au sein de réseaux réguliers 21
  - 2.3.1 Graphe de compatibilité 21
  - 2.3.2 Clique de poids maximal et PDPG fusionné 25
  - 2.3.3 Détection des cycles illégaux 26
  - 2.3.4 Cycles illégaux provoqués par plus de deux fusions 27
- 2.4 Perspectives 28

3 Mise en œuvre 31
- 3.1 Outils utilisés 31
  - 3.1.1 Librairies polyédriques 31
  - 3.1.2 Méta-modèles et EMF 31
- 3.2 Implémentation 31
  - 3.2.1 Méta-modèles 32
  - 3.2.2 Construction du graphe de compatibilité 33
  - 3.2.3 Calcul de la clique de poids maximal et construction du graphe fusionné 35
- 3.3 Travaux futurs 36
  - 3.3.1 Traitement des cycles illégaux 36
  - 3.3.2 Tests 37
  - 3.3.3 Liaison PDPG-Datapath Model 37
  - 3.3.4 Liaison MMAlpha-PDPG 38
  - 3.3.5 Implémentations d'optimisations et de variantes 38

Conclusion 39

Bibliographie 41

Introduction

Le marché du système embarqué est le théâtre d'une véritable course à tous les niveaux (performance, miniaturisation, consommation électrique). En effet que ce soit pour le multimédia portable (consoles portables, appareils photos et caméras numériques...) ou pour les télécoms (émetteurs-récepteurs et téléphones 3 et bientôt 4G), la très forte concurrence oblige les systèmes à être de plus en plus performants (meilleure résolution, meilleure transmission...) et à avoir une plus grande autonomie, tout en conservant, voir en diminuant, leur taille. La plupart de ces systèmes embarqués ont un point commun : ils doivent exécuter des fonctions lourdes et répétitives dues au traitement d'images ou du signal. Il s'agit généralement de traitements successifs sur des matrices, utilisant des boucles imbriquées, qui si elles sont exécutées sur le processeur générique embarqué, l'alourdissent et le ralentissent.

Un exemple de ces fonctions dites critiques est l'algorithme du filtre de Kalman [6]. Ce dernier est utilisé en traitement de signal pour estimer les paramètres d'un système dynamique à partir de mesures bruitées et utilise une succession de calculs sur des matrices. Par exemple un récepteur aux normes LTE (*Long Term Evolution*, normes qui doivent assurer le passage de la 3G à la 4G) actuelles doit réaliser entre 83 et 2000 filtres de Kalman en parallèle en 0.13 ms, afin d'estimer les paramètres du canal de propagation radio entre l'antenne-relais et le téléphone. Il est nécessaire que cette estimation ait lieu après la propagation, donc au niveau du récepteur, à l'intérieur du téléphone. Il est donc nécessaire d'avoir une puce réalisant ces calculs de manière massivement parallèle avec une consommation et une surface minimale.

Les circuits spécialisés apportent une solution à ces besoins en constante augmentation. Qu'ils soient reconfigurables *(Field-Programmable Gate Array)* ou non *(Application Specific Integrated Circuit)*, ces circuits sont chargés d'exécuter les fonctions les plus coûteuses du système de manière spécialisée. Ce semble être une réponse optimale aux problèmes de l'embarqué, puisque un circuit conçu de manière spécifique pour l'une de ces fonctions est à la fois plus performant mais aussi plus petit et moins consommateur qu'un processeur générique.

Mais la difficulté se pose dans la conception de ces circuits spécifiques à partir de l'algorithme de la fonction critique. En effet, extraire le parallélisme d'un tel algorithme afin de concevoir un circuit n'est déjà pas une chose aisée, mais il faut ensuite minimiser le coût matériel du circuit en limitant la surface de silicium, par exemple en évitant la multiplication de composants. Ce qui conduit la conception sous deux contraintes opposées : d'un côté celle d'exécuter le plus rapidement possible l'algorithme, et donc de paralléliser l'exécution au maximum, et de l'autre celle de limiter le coût de la puce obtenue.

Ce sont ces contraintes qui conduisent la conception automatique de circuits dédiés, ou synthèse de haut-niveau (HLS pour *High-Level Synthesis*), dont le but est d'obtenir à partir d'un algorithme décrit dans un langage de programmation de haut-niveau (C, Matlab...) une description matérielle d'une puce exécutant de manière spécifique cet algorithme. Le partage de ressources s'intéresse au sein de la HLS à la réduction de la surface de silicium utilisée par un circuit, et donc de son coût, par la réutilisation de composants matériels.

C'est dans le cadre de la HLS que se situe ce stage, dont le but était d'obtenir un outil de conception automatique de circuits dédiés pour des algorithmes contenant des boucles imbriquées, permettant de partager les ressources matérielles au sein de ces circuits pour en diminuer le coût et la taille.

Un premier chapitre de ce rapport est dédié à un état de l'art de différentes techniques existantes, à la fois dans le domaine de la synthèse de haut-niveau pour les boucles imbriquées et dans le domaine du partage de ressources. Le deuxième chapitre présente le travail effectué au cours du stage, principalement la définition des *Polyhedral Data-Path Graphs* (en Section 2.2) et l'adaptation d'un algorithme de partage de ressources à ces graphes (en Section 2.3). Enfin le dernier chapitre présente l'implémentation des PDPG et de l'algorithme.

Chapitre 1

État de l'art

La synthèse de haut-niveau (HLS) a développé de nombreuses techniques visant à produire des accélérateurs matériels, ou circuits dédiés, à partir d'algorithmes et de programmes, qu'elles soient consacrées aux nids de boucles ou non. Ce chapitre introduit les nids de boucles à contrôle statique (1.1.1), pour lesquels nous avons chercher à développer une nouvelle méthode de HLS minimisant la surface des réseaux réguliers (1.1.2). Il présente aussi certaines de ces techniques, en rapport avec notre travail, notamment le modèle polyédrique (1.2.1), la méthode des sommets (1.2.2) et plusieurs méthodes de partage de ressources, à un niveau d'abstraction élevé (1.3.1) ou au contraire directement au niveau des unités fonctionnelles (1.3.2).

1.1 Algorithmes sources et architectures cibles

Il est généralement admis qu'un programme passe 80% de son temps dans 20% de son code et que ces 20% sont les boucles du programme. Obtenir une architecture dédiée à l'accélération d'un ensemble de boucles imbriquées, ou nids de boucles, est donc particulièrement intéressant puisque ce dernier concentre une grande partie du temps de calcul d'une application. Or, lorsque l'on cherche à exécuter efficacement ces nids de boucles afin d'accélérer un programme, une des solutions possibles est de chercher à obtenir un circuit spécifique à l'exécution de ces tâches critiques. Cette section s'attache à présenter les nids de boucles à contrôle statique, qui sont une forme de nids de boucles particulières dans une première partie et les réseaux réguliers, qui sont des architectures développées spécifiquement pour l'accélération des nids de boucles dans une deuxième.

1.1.1 Nids de boucles à contrôle statique

Les nids de boucles à contrôle statique (ou ACL pour Affine Control Loop) sont des nids de boucles dont chaque boucle possède un indice entier (dans $\mathbb{Z}$) borné par des contraintes affines qui ne dépendent que de constantes, de paramètres ou d'indices de boucles plus externes [12]. Par exemple, le programme de multiplication de matrices présenté en figure 1.1, possède trois boucles imbriquées, dont les indices `i`, `j` et `k` ne dépendent que du paramètre N (la taille des matrices).

Chaque instruction d'un nid de boucles possède un vecteur d'itération $\overrightarrow{x} = \{x_1, x_2...x_n\}$, où n est la profondeur de l'instruction dans le nid de boucles, c'est-à-dire le nombre de boucles imbriquées englobant l'instruction, et où x_i correspond à l'indice de la boucle de profondeur i. Ici, x_i est donc un entier dont la valeur est contrainte par des équations linéaires. L'ensemble des contraintes sur les indices x_i du vecteur d'indices $\overrightarrow{x}$ définissent un sous-ensemble d'un espace à n dimensions ou domaine. Ce domaine contient l'ensemble des points entiers correspondant aux itérations de l'instruction. Par exemple le vecteur d'itération $\overrightarrow{x} = i, j, k$ du nid de boucles de la figure 1.1 et les contraintes qui lui sont associées définissent un domaine en 3 dimensions. Dans ce domaine, le point de coordonnées $\{0, 0, 0\}$ correspond à la première itération du programme et donc à l'exécution des instructions : `tmp1[0][0] = a[0][0]*b[0][0];` et `tmp2[0][0] = tmp2[0][0]+tmp1[0][0];`.

```
#define N = 100 //Parametre

int[][] matmult( int a[][], int b[][]) {
  int i,j,k;
  int c[N][N];
  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      tmp2[i][j] = 0
      for (k=0; k<N; k++){
        tmp1[i][j] = a[i][k]*b[k][j];
        tmp2[i][j] = tmp2[i][j]+tmp1[i][j];
      }
      c[i][j] = tmp2[i][j]
    }
  }
  return c;
}
```

FIG. 1.1: Multiplication de matrices en C

1.1.2 Réseaux réguliers de processeurs

Les architectures formées par des réseaux réguliers, ou architectures systoliques, sont des architectures massivement parallèles, où la structure de tous les processeurs est similaire et où les connexions sont uniquement locales [11]. Elles prennent la plupart du temps la forme d'une grille ou d'une ligne de processeurs où les communications avec l'extérieur sont assurées par les processeurs situés sur les côtés de la grille.

Elles sont particulièrement efficaces pour exécuter de manière parallèle les nids de boucles, puisqu'à chaque itération du nid de boucles la même suite d'opérations est exécutée, souvent sur des données calculées à l'itération précédente. C'est exactement le schéma des réseaux réguliers, où les données d'entrée (ou paramètres) entrent d'un côté de la grille, subissent un traitement dans la première ligne de processeurs, le résultat étant ensuite passé à la seconde ligne de processeurs et ainsi de suite.

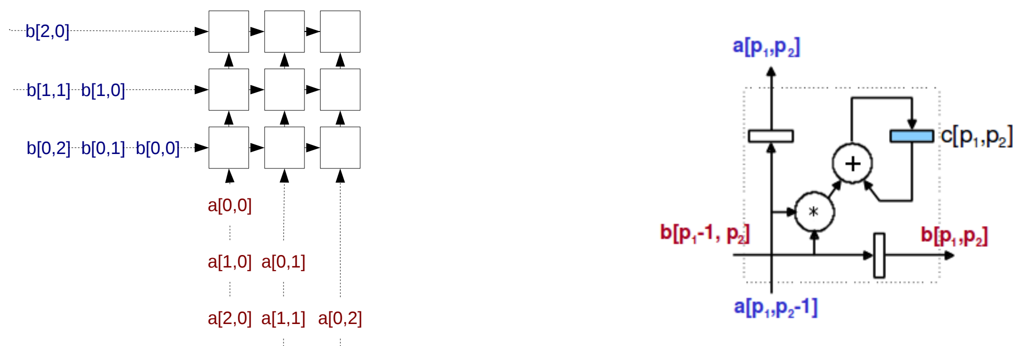


FIG. 1.2: Réseau régulier réalisant une multiplication de matrices et processeur de multiplication-accumulation

La figure 1.2 présente un exemple d'une telle architecture. Il s'agit d'un réseau de processeurs réalisant une multiplication de matrices, et d'un exemple de la structure des processeurs du réseau. Les deux matrices a et b entrent progressivement dans le réseau, par les bords du bas et de gauche. Chaque processeur récupère les éléments des matrices venant de ses voisins du bas et de gauche, les multiplie, les additionne avec le contenu de son registre et stocke le résultat, puis transmet les éléments des matrices à ses voisins du haut et de droite. Ainsi la matrice a voyage de bas en haut dans le réseau et la matrice b de gauche à droite.

Ainsi, à $t = 0$ seul le processeur de coordonnées $(0, 0)$ est actif, il est en effet le seul à avoir reçu des éléments des matrices ($a[0, 0]$ et $b[0, 0]$). Il les multiplie, les additionne avec le contenu de son registre (0) et stocke le résultat, avant d'envoyer $a[0, 0]$ à son voisin du haut et $b[0, 0]$ à son voisin de droite. À

$t = 1$, les processeurs de coordonnées $(0, 1)$ et $(1, 0)$ sont également actifs. Ils ont reçu un élément de matrice par le processeur de coordonnées $(0, 0)$ et un élément par les bords du réseau. Les processeurs actifs multiplient les éléments de matrices reçus, additionnent le résultat au contenu de leur registre et stockent le résultat de cette addition, puis ils envoient les éléments de matrices reçus à leurs voisins, et ainsi de suite.

1.2 Ordonnancement et allocation de nids de boucles

Pour obtenir une architecture dédiée à l'exécution d'un programme contenant des nids de boucles, on va chercher à ordonnancer et à allouer chaque opération des nids de boucle à un instant et un opérateur matériel précis afin de pouvoir spécifier un circuit. Le modèle polyédrique et l'utilisation de techniques comme la méthode des sommets permettent d'obtenir cet ordonnancement et cette allocation. Une fois ces étapes franchies, il faut ensuite générer le circuit correspondant. Ce circuit se présente généralement sous la forme d'un réseau de processeurs, particulièrement adapté à l'exécution parallèle de nids de boucles.

Cette section s'intéresse dans une première partie au modèle polyédrique, qui permet d'exprimer des algorithmes contenant des nids de boucles sous forme de système d'équations récurrentes affines. La deuxième partie introduit la méthode des sommets, qui permet d'obtenir un ordonnancement et une allocation pour ces systèmes d'équations et enfin une troisième partie présentera l'environnement MMAlpha qui implémente la méthode des sommets.

1.2.1 Modèle polyédrique

Le modèle polyédrique est une représentation mathématique qui permet, entre autres, d'exprimer les nids de boucles en fonction des contraintes sur les indices des boucles. Cette partie présente ce modèle et s'intéresse plus particulièrement au cas des polyèdres entiers. En effet ceux-ci permettent d'utiliser la programmation linéaire entière pour obtenir des ordonnancements et allocations afin de concevoir une architecture parallèle exécutant le nid de boucles exprimé [1].

Polyèdres entiers

Un polyèdre rationnel est un ensemble de la forme $\{z \in \mathbb{Q}^n | Qz \geq q\}$ où Q est une matrice entière et q un vecteur entier [12]. C'est à dire un ensemble de points dans un espace à n dimensions délimité par des contraintes linéaires. Un polyèdre entier ou domaine polyédrique est l'ensemble des points entiers contenus dans un polyèdre rationnel : $\{z \in \mathbb{Z}^n | Qz \geq q\}$. C'est-à-dire l'ensemble des points du polyèdre rationnel dont les coordonnées dans l'espace sont des entiers.

Un polyèdre peut être représenté de différentes manières :

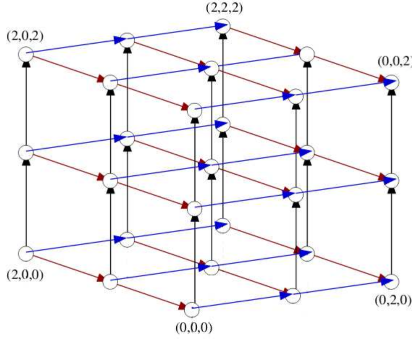
- Géométriquement, de la manière la plus intuitive, pour les polyèdres possédant suffisamment peu de dimensions.
- Par un ensemble d'inégalités linéaires, ou contraintes.
- Par un ensemble de constructeurs. Un constructeur est une combinaison linéaire des sommets et des rayons du polyèdre, forme sous laquelle on peut représenter les points d'un polyèdre.

Le programme de multiplication de matrices présenté plus haut (figure 1.1) permet d'obtenir un domaine polyédrique pour chacune des instructions. La figure 1.3 présente deux représentations (géométrique avec $N = 2$ et ensemble de contraintes) du polyèdre correspondant aux instructions qui forment le corps du nid de boucles (lignes 10 et 11 de la figure 1.3). En effet celles-ci sont exécutées à la même itération, leur domaine est donc le même.

Opérations sur les polyèdres

Il est possible de réaliser un certain nombre d'opérations sur les polyèdres. Ce sont des ensembles, on peut donc leur appliquer les opérations ensemblistes classiques (union, intersection, différence, projection) avec les résultats habituels. Il existe également des algorithmes qui leur sont spécifiques. Deux d'entre eux nous ont servis au cours du stage.

L'algorithme de Chernikova [7] permet de passer d'un ensemble de contraintes linéaires définissant un polyèdre à sa représentation sous forme de sommets et de rayons. Il est utilisé dans la méthode des sommets (voir 1.2.2) qui permet d'ordonnancer et d'allouer des opérations représentées par un polyèdre.



$$\mathcal{D}_{tmp1} = \mathcal{D}_{tmp2} = \begin{cases} 0 \leq i \leq N \\ 0 \leq j \leq N \\ 0 \leq k \leq N \end{cases}$$

FIG. 1.3: Domaine polyédrique de la multiplication ($\mathcal{D}_{tmp1}$) et de l'addition ($\mathcal{D}_{tmp2}$) de la multiplication de matrices sous deux formes différentes

Le polynôme d'Ehrhart [7] associé à un polyèdre entier est un polynôme exprimant le nombre de points au sein du polyèdre, en fonction de ses paramètres s'il en a. Nous l'utilisons dans notre méthode de partage de ressources (voir 2.3.1).

Systèmes d'équations récurrentes affines

Les équations récurrentes sont utilisées pour modéliser des algorithmes dans le but de les paralléliser. Les systèmes d'équations récurrentes affines (ou SARE pour System of Affine Reccurrent Equations) paramétrés sont des ensembles finis d'équations de la forme

$$z \in \mathcal{D}, p \in \mathcal{P} \Rightarrow V_i(z,p) = f(..., V_j(I(z,p)), ...)$$

où

- V_i et V_j sont des variables appartenant à l'ensemble $\mathbf{V}$. Chaque variable $V_i \in \mathbf{V}$ peut-être vue comme une fonction $V_i : \mathcal{D}_{V_i} \rightarrow X$ où $\mathcal{D}_{V_i}$ est le domaine de V_i et X un ensemble quelconque ($\mathbb{Z}$ ou $\mathbb{R}$, par exemple).
- $\mathcal{D}$ est le domaine paramétré de l'équation, avec $\mathcal{D} \subseteq \mathcal{D}_{V_i}$.
- $I(z,p) = D \begin{pmatrix} z \\ p \end{pmatrix} + d$ est une fonction affine de $\mathbb{Z}^{d_{V_j}}$ vers $\mathbb{Z}^{d_{V_i}}$ appelée fonction de dépendance.
- $p \in \mathcal{P}$ est un vecteur des paramètres de taille du système.
- f est une fonction quelconque de $\mathbf{V}^n$ vers $\mathbf{V}$.

Les équations sont dites récurrentes parce qu'elles expriment des dépendances entre leurs diverses variables, et peuvent donc servir à décrire une boucle, et le système est dit paramétré parce qu'il prend en compte $\mathcal{P}$, un ensemble de paramètres qui influe sur son domaine. Les SARE sont un formalisme très puissant qui permet d'exprimer un grand nombre d'algorithmes. Nous nous intéresserons à la représentation des nids de boucles à contrôle statique par les SARE.

Dans le cas d'un nid de boucles à contrôle statique, le domaine $\mathcal{D}$ des équations est un polyèdre entier ($D \subseteq \mathbb{Z}^n$) défini par les contraintes linéaires sur le vecteur d'indices, de même que les domaines $\mathcal{D}_{V_i}$ des variables V_i (les indices des boucles sont exprimés sur $\mathbb{Z}$). Le programme de multiplication de matrices présenté en figure 1.1 a pour SARE le système présenté en figure 1.4, où l'addition et la multiplication sont les fonctions f et $\mathcal{D}$ est le polyèdre convexe entier représentant le domaine d'itération du nid de boucles.

$$\begin{aligned} tmp1(i,j,k,N) &= a(i,j,k,N) * b(i,j,k,N) \\ tmp2(i,j,k,N) &= \begin{cases} k \leq 0, & 0 \\ k > 0, & tmp2(i,j,k-1,N) + tmp1(i,j,k,N) \end{cases} \\ c(i,j) &= tmp2(i,j,N,N) \end{aligned}$$

FIG. 1.4: Système d'équations récurrentes affines du programme matmult

Polyhedral Reduced Dependencies Graph

On peut, à partir d'un SARE, obtenir un PRDG (pour *Polyhedral Reduced Dependencies Graph*). Il s'agit d'un graphe orienté $G = (W, E)$ tel que :

- Un sommet $w_i \in W$ correspond à une variable $V_i \in \mathbf{V}$ du SARE d'origine. w_i possède un domaine polyédrique $\mathcal{D}_{w_i} = \mathcal{D}_{V_i}$, une fonction $f_{w_i} : \mathcal{D}_{V_i} \to X$ et des ports d'entrées, correspondant aux paramètres de la fonction et de sorties, correspondant aux résultats de la fonction.
- Il y a un arc $e = (w_i, p_i, w_j, p_j) \in E$ si il y a une dépendance de données entre le port p_i du sommet w_i et le port p_j du sommet w_j. e possède une fonction de dépendance f_e de $\mathbb{Z}^{d_{V_j}}$ vers $\mathbb{Z}^{d_{V_i}}$ correspondant à la fonction I du SARE d'origine.

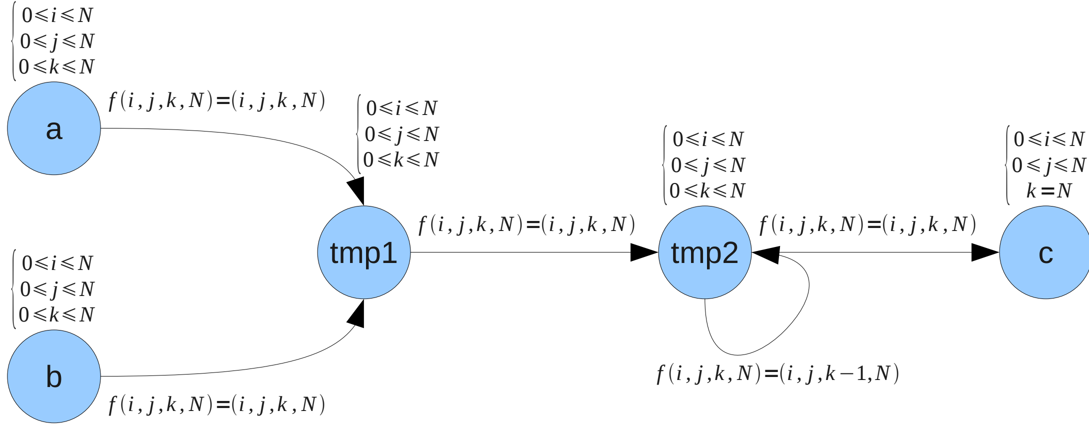


FIG. 1.5: PRDG de la multiplication de matrices

Par exemple, le PRDG obtenu à partir du SARE de la multiplication de matrices de la figure 1.1 est présenté en figure 1.5. Le domaine de chaque sommet est présenté sous forme de contraintes linéaires au-dessus du sommet. Chaque arc porte sa fonction de dépendance. Il y a une dépendance cyclique sur le sommet $tmp2$ qui représente l'accumulation des résultats de la multiplication. La fonction de dépendance de l'arc représente bien l'affectation de la somme de $tmp2(i, j, k-1, N)$, c'est-à-dire de la valeur de $tmp2$ calculé à l'itération précédente, et de $tmp1$ calculé au cycle courant, à $tmp2(i, j, k, N)$, c'est-à-dire $tmp2$ du cycle courant.

1.2.2 Ordonnancement et allocation avec le modèle polyédrique

Une fois un nid de boucles de profondeur n modélisé par un SARE, on va chercher à obtenir une architecture systolique l'exécutant. Afin que chaque opération de l'algorithme soit exécutée sur un processeur donné à un instant donné il faut trouver :

- Une fonction d'*ordonnancement* t_{V_i} de $\mathcal{D}_{V_i}$ vers $\mathbb{N}^m$, pour chaque $V_i \in \mathbf{V}$, qui associe un instant d'exécution à chaque opération, et donc à chaque point d'itération, avec $m = 1$, ou plus dans le cas d'un temps multi-dimensionnel (voir 1.3.1).
- Une fonction d'*allocation* p_{V_i} de $\mathcal{D}_{V_i}$ vers $\mathbb{N}^{n-m}$, pour chaque $V_i \in \mathbf{V}$. Cette fonction associe à chaque opération le vecteur de coordonnées d'un processeur dans un réseau de processeurs de dimension $n - m$.

La fonction d'ordonnancement doit respecter les deux contraintes suivantes :

$$\begin{cases} \forall (V_i, \mathcal{D}_{V_i}) \in \mathbf{V}, \vec{x} \in \mathcal{D}_{V_i} \Rightarrow t_{V_i}(\vec{x}) \geq 0 \\ \forall (V_i, V_j, \mathcal{D}, I), \vec{x} \in \mathcal{D} \Rightarrow t_{V_i}(\vec{x}) > t_{V_j}(I(\vec{x})) \end{cases} \tag{1.1}$$

où $\vec{x} = (z, p)$ est le vecteur d'itération, éventuellement paramétré par p, avec z un vecteur à n composantes.

Ces contraintes imposent de ne pas attribuer d'instant d'exécution négatif ($t_{V_i}(\vec{x}) \geq 0$), et de respecter les dépendances de causalité entre les opérations, c'est à dire d'attribuer à V_i un instant d'exécution strictement supérieur à celui de V_j si elle dépend de cette dernière ($t_{V_i}(\vec{x}) > t_{V_j}(I(\vec{x}))$). En effet les opérandes d'une opération doivent être disponibles, et donc avoir été calculés, pour qu'exécuter l'opération soit légal.

Trouver un ordonnancement optimal sur des domaines d'itération extrêmement larges demanderait une énumération longue et coûteuse des points de ces domaines. Parcourir l'ensemble des points d'un polyèdre ne serait de plus pas possible dans le cas de polyèdres paramétrés, par exemple dans le cas d'une multiplication de matrices dont la taille est inconnue.

Une des solutions pour éviter cela est d'utiliser une fonction d'ordonnancement qui associe à chaque calcul un instant d'exécution en fonction de sa position. On s'intéressera plus particulièrement à une fonction linéaire, tout d'abord parce qu'elle permettra d'utiliser des méthodes de programmation linéaire entière et ensuite parce qu'elle facilitera la parallélisation et la synthèse d'architecture.

Il s'agit donc de trouver une fonction linéaire de la forme :

$$t_{V_i}(\vec{x}) = A_{V_i}\vec{x} + \alpha_{V_i} \tag{1.2}$$

De la même manière, on peut chercher une fonction d'allocation de $\mathcal{D}_{V_i}$ vers $\mathbb{N}^m$. En général m est égal à $n-1$, p est donc une projection linéaire de l'espace d'itération de la forme :

$$p_{V_i}(\vec{x}) = \beta_{V_i}\vec{x} \tag{1.3}$$

où β est un vecteur de même dimension que $\vec{x}$, dont l'un des composants est égal à 0 (projection linéaire).

Plusieurs méthodes ont été développées afin de trouver des ordonnancements et des allocations. On peut notamment citer la méthode dite "des sommets" développée par Rajopadhye et al. et Quinton et Van Dongen [1], [12] et la méthode de Farkas développée par Feautrier [4], [2].

Méthode des sommets

La méthode des sommets s'appuie sur la propriété selon laquelle si une contrainte est vraie pour tous les sommets d'un polyèdre alors elle est vraie en tout point du polyèdre et réciproquement [12]. On va donc chercher à résoudre un programme linéaire où l'on minimisera un paramètre associé à la fonction d'ordonnancement (temps d'exécution total, délai entre opérateurs...) ou à la fonction d'allocation (taille ou forme de l'architecture) et où les contraintes que doivent respecter ces fonctions seront exprimées sur les sommets du polyèdre.

L'algorithme de Chernikova permet, entre autres, de calculer l'ensemble des sommets d'un polyèdre, sous la forme de n-uplets, n étant le nombre de dimensions du polyèdre. On peut ensuite appliquer les contraintes nécessaires à ces points, pour obtenir un ensemble d'équations linéaires.

Par exemple, les sommets du polyèdre représentant le domaine d'itération de l'addition (ou de la multiplication) de la multiplication de matrice sont les huit points $(0,0,0), (0,0,N), ..., (N,N,N)$. Les contraintes de dépendances de l'équation 1.1 s'exprimeront donc comme suit sur les sommets :

$$\left\{\begin{array}{lcl} t(0,0,N) & > & t(0,0,N-1) \\ t(0,N,N) & > & t(0,N,N-1) \\ ... & & \\ t(N,N,N) & > & t(N,N,N-1) \end{array}\right. \tag{1.4}$$

De même, si l'on cherche à obtenir un réseau régulier de processeurs, on va établir des contraintes sur l'*allocation* et l'*ordonnancement* de manière à ce que les données (le contenu des matrices a et b) entrent par les côtés du réseau et circulent de processeur en processeur le long des axes i et j :

$$\left\{\begin{array}{lcl} p(0,N) & > & p(0,N-1) \\ p(N,0) & > & p(N-1,0) \\ p(N,N) & > & p(N-1,N) \\ p(N,N) & > & p(N,N-1) \end{array}\right. \left\{\begin{array}{lcl} t(0,N) & > & t(0,N-1) \\ t(N,0) & > & t(N-1,0) \\ t(N,N) & > & t(N-1,N) \\ t(N,N) & > & t(N,N-1) \end{array}\right. \tag{1.5}$$

Avec ces contraintes, une projection linéaire intuitive pour obtenir p est la projection le long de l'axe k. On obtient ainsi un réseau de processeurs de N sur N, où chaque processeur exécute une

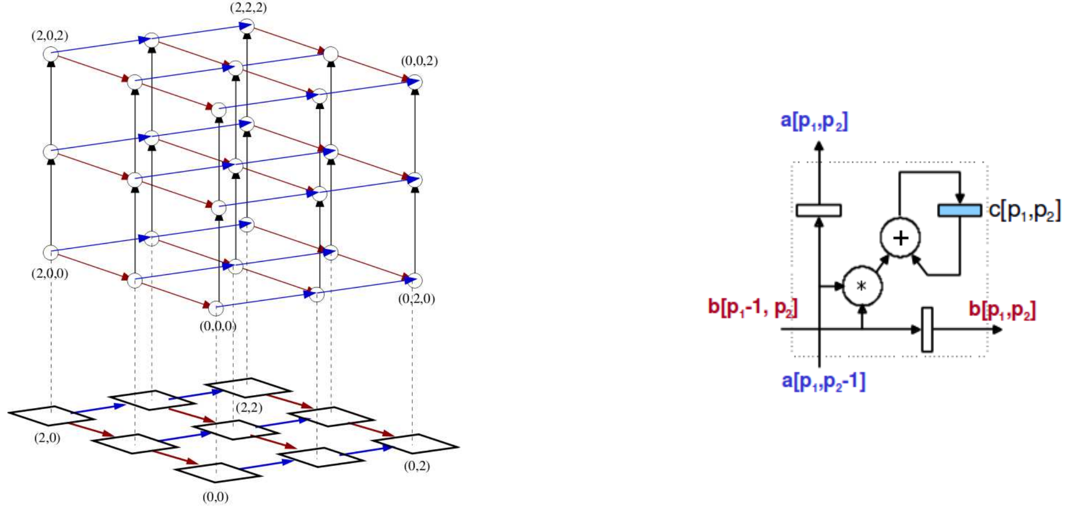


FIG. 1.6: Projection sur un réseau de processeurs et processeur de multiplication-accumulation

multiplication-accumulation (MAC) à l'instant τ et transmet ses données d'entrée à l'instant $\tau+1$ à son voisin (figure 1.6).

On peut ensuite chercher à résoudre le problème de programmation linéaire entière consistant à trouver une fonction $t(\vec{x})$ minimisant le temps total d'exécution ou la latence entre chaque opération par exemple, tout en respectant ces contraintes. Si on cherche à minimiser le temps total d'exécution, on peut poser une fonction affine $T(p) = \Gamma p + \gamma$ qui en sera la borne supérieure. Ainsi on pose la contrainte suivante pour tout sommet $(V_i, \mathcal{D}_{V_i}) \in \mathbf{V}$:

$$\vec{x} = (z, p) \in \mathcal{D}_{V_i} \Rightarrow t_{V_i}(\vec{x}) \leq T(p) \tag{1.6}$$

Et on cherche à résoudre le problème de programmation linéaire consistant à minimiser T en respectant les contraintes de dépendances et d'allocation (équations 1.4 et 1.5), ainsi que celles concernant T.

Dans le cas de l'exemple de la multiplication de matrices, un ordonnancement possible respectant ces contraintes est $t(i,j,k) = i + j + k$. Il correspond à l'exécution du réseau régulier présenté en 1.1.2.

1.2.3 Environnement MMAlpha

MMAlpha est un environnement de programmation qui implémente la méthode des sommets. Il est développé en Mathematica et en C et utilise la librairie polyédrique *polylib* (voir 3.1.1). MMAlpha permet d'écrire et de manipuler des programmes en langage Alpha. Il est possible, entre autres, d'ordonnancer et d'allouer les opérations d'un programme Alpha grâce à la méthode des sommets. On peut à partir de cet ordonnancement et de cette allocation obtenir la description matérielle d'un réseau régulier en langage VHDL *(Very high speed integrated circuit Hardware Description Language)*.

Langage Alpha

Le langage Alpha est un langage fonctionnel dédié à la synthèse de réseaux réguliers. Un programme Alpha est un ensemble d'équations uniformes récurrentes *(uniform reccurence equation)*, un sous-ensemble des équations récurrentes affines. Chaque équation est à *affectation unique* et définit une variable sur un domaine multi-dimensionnel. Il est possible d'obtenir un programme Alpha à partir d'un programme C en transformant chaque instruction en instruction à *affectation unique* puis en déterminant les domaines de chaque variable.

1.3 Partage de ressources

Un circuit dédié se doit d'avoir un prix et une consommation électrique les plus faibles possibles. Ces deux paramètres sont en lien direct avec la surface de celui-ci. En effet, plus une puce est grande,

plus elle utilise de silicium et donc plus elle coûte cher, et plus elle nécessite de courant. Or la fonction d'allocation p obtenue par la méthode des sommets (cf 1.2.2) peut donner des réseaux de processeurs très importants, souvent trop grands. Ainsi l'allocation d'une multiplication de matrices de 100 par 100, donnerait un réseau de processeurs de mêmes dimensions, c'est à dire un réseau de 10000 processeurs. La surface et le coût d'un tel réseau sont bien trop importants, et feraient perdre tout l'intérêt d'un circuit dédié. On peut chercher à réduire la surface d'un tel circuit en restant au niveau d'abstraction proposé par le modèle polyédrique, mais on peut aussi choisir de descendre plus bas, pour gagner de la surface au niveau même des opérateurs matériels. Cette section présente les deux méthodes et s'intéresse plus particulièrement à la fusion de chemin de données présentée par Moreano, Borin, de Souza et Araujo dont nous nous sommes inspirés pour le partage de ressources au sein de réseaux réguliers.

1.3.1 Partitionnement et temps multi-dimensionnel

Le partitionnement est une technique permettant, une fois l'allocation des opérations réalisée, de diminuer la surface nécessaire pour un réseau régulier. Il consiste à considérer le réseau donné par la fonction d'allocation p comme un ensemble de processeurs virtuels et chercher à le découper en *tuiles* régulières de processeurs virtuels.

Une fois une fonction d'allocation p obtenue, à chaque opération sont allouées les coordonnées d'un processeur virtuel dans une grille de processeurs. Le partitionnement consiste à découper cette grille en τ *tuiles*, régions de γ processeurs virtuels voisins.

On peut ensuite, soit avoir un réseau de γ processeurs physiques qui exécute les *tuiles* séquentiellement, on parle alors de partitionnement LPGS (Localement Parallèle Globalement Séquentiel), soit avoir un réseau de τ processeurs physiques, chaque processeur physique visitant un à un les processeurs virtuels de la tuile qui lui est attribuée, on parle alors de partitionnement LSGP (Localement Séquentiel Globalement Parallèle). La première solution, multiplie le temps d'exécution et divise la surface du circuit par τ, le nombre de *tuiles* dans le réseau virtuel, la seconde par γ, le nombre de processeurs dans une tuile.

La figure 1.7 présente un exemple de réseau virtuel de vingt-quatre processeurs, où chaque processeur est activé au même moment et fonctionne en parallèle des autres. On peut découper ce réseau en quatre *tuiles* de six processeurs afin de diminuer sa surface. Le réseau physique issu du partitionnement LPGS (1.8) possède six processeurs qui exécutent les tâches d'une *tuile* de manière parallèle (Localement Parallèle), mais le réseau exécute les *tuiles* les unes après les autres (Globalement Séquentiel). A l'inverse, le réseau physique issu du partitionnement LSGP (1.9) possède quatre processeurs, chacun exécutant une à une les tâches de la *tuile* qui lui est attribuée (Localement Séquentiel), mais tous le faisant en parallèle des autres (Globalement Parallèle).

Ordonnancement dense

Darte, Schreiber, Rau et Vivien ont présenté une technique cherchant à "saturer" le matériel, c'est à dire à empêcher les processeurs du réseau d'être inactifs [3]. En effet après l'allocation et l'ordonnancement par la méthode des sommets, on peut obtenir une architecture où chaque processeur n'est actif que périodiquement, par exemple un cycle sur deux. Dans un tel cas, où les processeurs effectuent le même traitement et où la moitié d'entre eux est inactive à un cycle et l'autre moitié au cycle suivant, il est logique d'essayer de réduire la surface de l'architecture de moitié en rendant chaque processeur actif à tous les cycles.

Les auteurs cherchent d'abord une fonction d'allocation p qui alloue à chaque opération les coordonnées d'un processeur virtuel dans une grille de dimension $n-1$ et de forme $V = (V_1, ...V_{n-1})$, où n est la profondeur du nid de boucles considéré. Cette grille est ensuite découpée en *clusters* (équivalents aux *tuiles*). Chaque *cluster* est associé à un processeur physique d'un réseau de dimension $n-1$ également et de forme $P = (P_1, ..., P_{n-1})$. Ce processeur physique est chargé d'effectuer tous les calculs alloués aux processeurs virtuels de son cluster. Il s'agit d'un partitionnement LSGP (le réseau physique possède τ processeurs). L'ensemble des *clusters* doit couvrir toute la grille de processeurs virtuels. La forme d'un *cluster*, C est définie par :

$$C = (C_1, ..., C_{n-1}),$$

où $C_i = \lceil V_i/P_i \rceil$

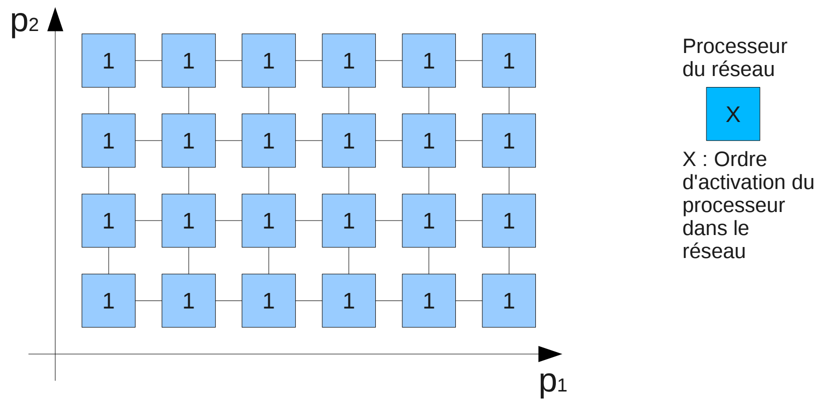


Fig. 1.7: Réseau virtuel avant partitionnement

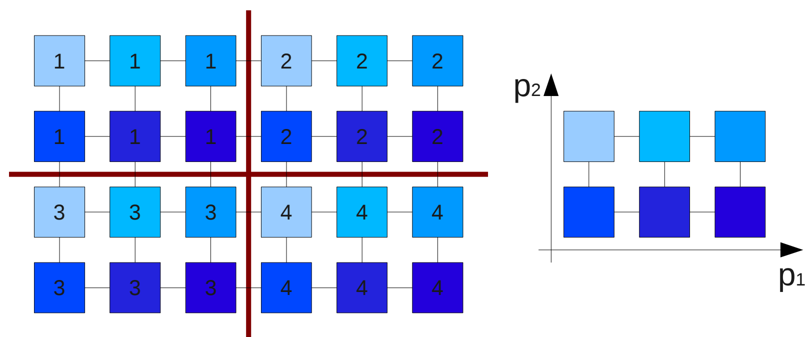


Fig. 1.8: Partitionnement LPGS en quatre tuiles

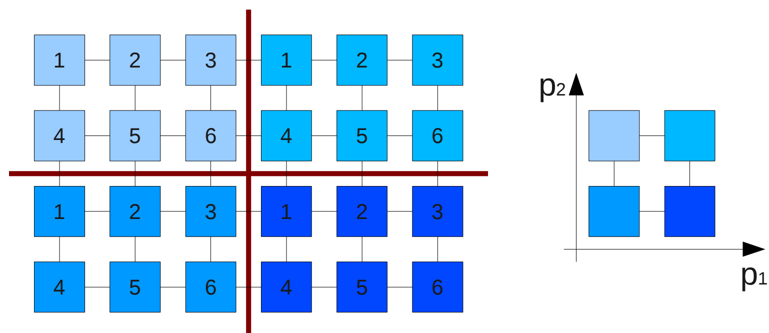


Fig. 1.9: Partitionnement LSGP en quatre tuiles

Les auteurs ont déterminé qu'un ordonnancement t est dense si et seulement si t respecte la contrainte suivante, et ce pour n'importe quelle permutation entre les éléments de C, accompagnée de la même permutation aux $n-1$ premiers éléments de t :

$$t = (k_1, k_2C_1, k_3C_1C_2, ..., k_nC_1...C_{n-1}) \tag{1.7}$$

avec $k_n = \pm 1$ et $k_i \wedge C_i = 1$ où $x \wedge y$ est le plus grand diviseur commun de x et y.

Il est possible d'ajouter des contraintes linéaires supplémentaires sur l'ordonnancement ainsi obtenu. Pour cela, on calcule par programmation linéaire les bornes pour les éléments de t, puis on énumère les valeurs possibles pour les k_i qui sont dans ces bornes et qui satisfont la contrainte $k_i \wedge C_i = 1$. Puis toutes les permutations possibles entre les éléments de C (et donc de t) sont essayées. Chaque valeur de k_i donne un ordonnancement dense, parmi ceux-ci, seuls les ordonnancements satisfaisant l'ensemble du système de contraintes linéaires sont conservés. Enfin l'un d'entre eux est sélectionné en fonction d'une mesure du temps total d'exécution et d'une estimation de la surface finale du circuit.

Cette méthode peut sembler difficile à mettre en place, en effet, contrairement aux contraintes linéaires, les contraintes de formes $k_n = \pm 1$ et $k_i \wedge C_i = 1$ n'entrent pas dans le cadre de la programmation linéaire entière, et ne sont donc pas utilisables avec un solveur classique. De plus l'énumération des valeurs possibles des coefficients k_i et des différentes permutations possibles ($!n$ où n est le nombre de dimensions du *cluster*) peut se montrer coûteuse dès lors que le nombre de dimensions des domaines d'itération augmente.

Temps multi-dimensionnel

Les ordonnancements multi-dimensionnels sont des ordonnancements où la fonction de temps associe à chaque opération des coordonnées en plusieurs dimensions. Chacun de ces instants multi-dimensionnels est comparable aux autres par l'ordre lexicographique. Ainsi, l'instant $t_1 = (1, 1, 2)$ est plus grand que l'instant $t_2 = (1, 1, 0)$ et plus petit que l'instant $t_3 = (2, 0, 0)$. Le temps est divisé en plusieurs dimensions qu'on peut voir comme des heures, minutes, secondes, sauf qu'il n'y a pas un nombre fixe de "minutes" dans une "heure" ou de "secondes" dans une "minute". $t_4 = (0, x)$ est toujours plus petit que $t_5 = (1, 0)$, et ce quelque soit x.

Le temps multi-dimensionnel peut être utilisé de deux manières. Soit pour obtenir un ordonnancement linéaire valide alors qu'il n'y a pas de solution uni-dimensionnelle au problème d'ordonnancement [5]. Soit pour diminuer la surface d'un circuit en augmentant le temps de calcul. Il s'agit alors d'une forme particulière de partitionnement, où on ajoute une dimension au temps pour en enlever une au réseau de processeurs. On peut passer ainsi d'une grille à une ligne de processeurs, par exemple.

Si on reprend l'exemple de la figure 1.7, on pourrait exécuter les opérations du réseau virtuel sur une ligne de six processeurs avec un temps à deux dimensions. A chaque ligne de processeurs de la grille correspondrait une unité de la première dimension du temps, et à chaque opération d'un processeur au sein d'une ligne correspondrait une unité de la deuxième dimension du temps. La première opération d'un processeur de la première ligne serait exécutée à $t = (1, 1)$, la troisième opération d'un processeur de la deuxième ligne à $t = (2, 3)$.

1.3.2 Fusion de chemins de données

Des solutions ont été mises au point pour diminuer la surface d'architectures multi-modes ou reconfigurables dédiées à plusieurs applications ou à une application contenant plusieurs tâches critiques à accélérer, directement au niveau des opérateurs matériels. L'une d'elle est la fusion de chemin de données. Elle consiste à utiliser un composant commun à plusieurs fonctions ou opérations plutôt que d'en avoir un par opération, à condition que ce composant n'ait pas besoin d'être utilisé au même moment, c'est-à-dire que les opérations soient mutuellement exclusives par rapport au temps. Ainsi, en étudiant les fonctions pour lesquelles on veut concevoir un circuit, on peut rechercher les opérations ou motifs d'opérations répétés pour définir des opérateurs matériels partagés.

La méthode présentée en [8] par Moreano, Borin, de Souza et Araujo, tout comme celle de Wolinski, Kuchcinski, Raffin et Charot [14], utilise un graphe de compatibilité afin de déterminer les meilleurs choix de fusion d'opérateurs au sein d'un ensemble de chemins de données. Pour cela, elle se base sur les graphes de flot de données (ou DFG pour *Data-Flow Graphs*) des programmes pour lesquels on cherche à générer un circuit dédié. Un graphe de flot de données est un graphe orienté $G = (V, E)$ où :

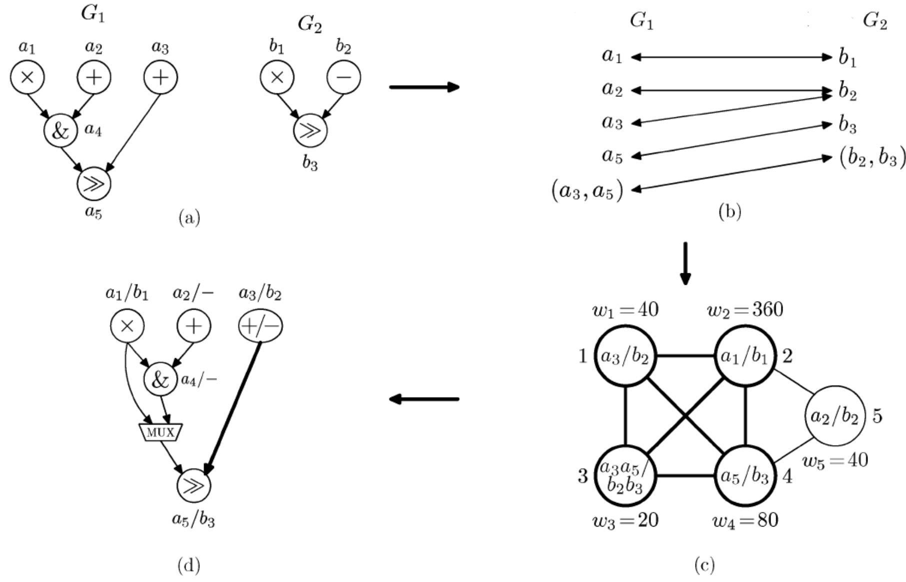


FIG. 1.10: Étapes de fusion de DFG (extrait de [8])

- Un sommet $v \in V$ correspond à une opération ou une variable. Chaque sommet a un ensemble de ports d'entrée.
- Un arc $e = (u, v, p) \in E$ correspond à un transfert de données du sommet u au port p du sommet v.

Un tel graphe peut être extrait à la compilation d'une application. Un chemin de données est obtenu par l'ordonnancement d'un graphe de flot de données. Il s'agit d'un DFG où chaque opération possède un instant d'exécution et dans lequel les registres ont été placés. Il décrit l'ensemble des calculs que doit réaliser le circuit.

La figure 1.10 donne un exemple de fusion de deux DFG. On peut voir en 1.10(a) les DFG G_1 et G_2 d'origine et en 1.10(d) le graphe fusionné obtenu à la fin du processus, dans lequel les sommets a_1 et b_1, a_3 et b_2 et a_5 et b_3 ont été fusionnés. Les figures 1.10(b) et 1.10(c) présentent les deux étapes intermédiaires nécessaires à la fusion de DFG qui sont détaillées plus bas.

Graphe de compatibilité

Le graphe de compatibilité G_c obtenu à partir de deux DFG $G_i = (V_i, E_i)$ et $G_j = (V_j, E_j)$, est un graphe pondéré non-orienté $G_c = (V_c, E_c)$, où :

- Chaque sommet $v_c \in V_c$ de poids w_c correspond à une *fusion* possible entre
  - $v_i \in V_i$ et $v_j \in V_j$ telle que les sommets v_i et v_j sont fusionnables.
  - $e_i = (u_i, v_i, p_i) \in E_i$ et $e_j = (u_j, v_j, p_j) \in E_j$ telle que les arcs e_i et e_j sont fusionnables.
- Il y a une arête $e_c = (u_c, v_c) \in E_c$ si les *fusions* représentés par u_c et v_c sont compatibles
- Le poids ω_c d'un sommet v_c est le gain de surface obtenu par la fusion des deux opérations/arcs de v_c.

La figure 1.10(c) présente le graphe de compatibilité obtenu pour les deux DFG de la figure 1.10(a). Chaque sommet correspond à une *fusion* d'opérateurs ou d'arcs possible et les arcs relient les *fusions* compatibles entre elles.

Sommets et arcs fusionnables : Deux sommets $v_i \in G_i$ et $v_j \in G_j$ sont fusionnables si $O(v_i) \cap O(v_j) \neq \emptyset$, avec $O(v)$ l'ensemble des opérateurs matériels disponibles pouvant réaliser l'opération représentée par v, c'est-à-dire si il existe un opérateur matériel dans la bibliothèque de composants mise à disposition capable d'effectuer l'opération correspondant à v_i et l'opération correspondant à v_j.

Deux arcs $e_i = (u_i, v_i, p_i)$ et $e_j = (u_j, v_j, p_j)$ sont fusionnables si :

- $O(u_i) \cap O(u_j) \neq \emptyset$, c'est à dire qu'il existe un opérateur matériel capable d'effectuer l'opération correspondant à v_i et l'opération correspondant à v_j

- $O(v_i) \cap O(v_j) \neq \emptyset$, c'est à dire qu'il existe un opérateur matériel capable d'effectuer l'opération correspondant à u_i et l'opération correspondant à u_j
- les ports p_i et p_j correspondent

Deux ports p_i et p_j correspondent si ils occupent la même position sur leur sommet respectif, ou si un de leurs sommets cibles (v_i ou v_j) correspond à une opération commutative.

Sur la figure 1.10(b) on peut voir qu'il existe un même opérateur commun à a_1 et b_1, un autre à a_2 et b_2, et ainsi de suite, et que comme il existe des opérateurs communs à a_3 et b_2 et à a_5 et b_3, il est possible de fusionner les arcs (a_3, a_5) et (b_2, b_3).

Compatibilité des fusions : Deux sommets v_c et u_c du graphe de compatibilité G_c sont compatibles si ils ne représentent pas deux *fusions* possibles du même sommet. Deux *fusions* de sommets sont compatibles tant qu'aucun des sommets concernés par ces fusions ne sont en commun. Une *fusion* d'arcs est compatible avec les mêmes sommets que les *fusions* de sommets dont elle dépend, c'est-à-dire, les *fusions* concernant ses sommets sources et cibles.

On peut voir sur la figure 1.10(c) que les *fusions* correspondant aux sommets 1 (a_3/b_2) et 5 (a_2/b_2) ne sont pas compatibles (il n'y a pas d'arête entre elles). Elles concernent en effet toutes les deux le sommet b_2. Comme la *fusion* correspondant au sommet 2 (a_3a_5/b_2b_3) dépend de celle du sommet 1, celle-ci n'est pas compatible non plus avec la *fusion* du sommet 5, c'est pourquoi ce dernier ne possède aucune arête le reliant aux sommets 1 et 2.

Poids des sommets : Le poids ω_c d'un sommet représente le gain de surface apporté par la fusion, il est égal à la différence entre la surface occupée avant la fusion et la surface occupée après la fusion. Avec $S(x)$ la surface de l'opérateur matériel réalisant x et op_V le plus petit opérateur matériel réalisant chacune des opérations associés aux sommets de l'ensemble V, on a :

- Si v_c représente une *fusion* u/v de sommets, le gain est égal à la somme des surfaces des opérateurs, moins la surface de l'opérateur qui les remplace et du multiplexeur placé en entrée de cet opérateur. En effet, si un opérateur réalise deux opérations différentes, un multiplexeur est nécessaire pour sélectionner les opérandes en fonction de l'opération à effectuer.

$$\omega_c = S(op_u) + S(op_v) - (S(op_{\{u,v\}}) + S(multiplexeur)) \tag{1.8}$$

- Si v_c représente une *fusion* $(u_i, v_i, p_i)/(u_j, v_j, p_j)$ d'arcs, le gain est égal à la surface du multiplexeur enlevé entre les opérateurs u_i/u_j et v_i/v_j, qui est remplacé par un unique fil.

$$\omega_c = S(multiplexeur) \tag{1.9}$$

Ainsi, sur l'exemple de la figure 1.10, on peut voir que l'utilisation d'un seul multiplieur au lieu de deux apporte un gain de 360, que la suppression d'un multiplexeur apporte un gain de 20 et ainsi de suite.

Clique de poids maximal et fusion de graphes

Le graphe de compatibilité exprime toutes les combinaisons possibles de *fusions* entre les différents opérateurs de deux DFG. Il faut ensuite sélectionner la meilleure combinaison afin que la surface du circuit dérivé de la fusion des DFG soit la plus petite possible. Pour cela un calcul de la clique de poids maximale est effectué sur le graphe de compatibilité. Cette clique contiendra l'ensemble des sommets retenus pour la fusion de graphes. Puisque la clique correspond à la réduction de surface maximum, le graphe fusionné obtenu sera celui avec la plus petite surface.

Une clique est un ensemble de sommets deux-à-deux adjacents, c'est à dire que chaque sommet d'une clique est connecté à chaque autre (la clique définit un sous-graphe complet). La clique de poids maximal sélectionne donc l'ensemble de *fusions compatibles* dont le poids cumulé est le plus grand, c'est à dire qui assureront la plus grande réduction de surface du circuit, puisque le poids des sommets représente le gain de surface apporté par les *fusions*.

Le problème de la clique de poids maximal est connu pour être NP-Complet. Les auteurs de [8] ont utilisé l'algorithme présenté en [10] auquel ils ont ajouté une heuristique qui interrompt l'exécution après un temps polynomial en la cardinalité de V_c.

La figure 1.10(c) présente le graphe de compatibilité obtenu à partir des deux DFG de la figure 1.10(a), et, en gras, la clique de poids maximal de ce graphe. Les sommets 1, 2, 3 et 4 sont tous inter-connectés

et la somme de leur poids est supérieure au poids de chacune des autres cliques du graphe (500, contre 480 pour la clique formée des sommets 2, 4 et 5).

On peut enfin construire le graphe fusionné à partir des deux DFG originaux et de la clique de poids maximum trouvée. Cette méthode peut être utilisée successivement sur plusieurs DFG, en fusionnant le premier graphe résultat de la fusion de deux DFG avec un troisième, et ainsi de suite.

La figure 1.10(d) présente un tel DFG fusionné à partir des deux DFG de la figure 1.10(a), on passe ainsi de 8 opérateurs et 6 connexions à 6 opérateurs (dont un multiplexeur) et 6 connexions.

Cependant les DFG d'algorithmes contenant des nids de boucles peuvent se révéler très grands. En effet pour chaque instruction du corps du nid de boucles on aura autant de sommets que d'itérations du nid de boucles. Fusionner deux DFG d'aussi grande taille mènerait à une explosion combinatoire du graphe de compatibilité.

1.3.3 Partage de ressources au sein de réseaux réguliers

Les nids de boucle forment les parties des programmes qui peuvent se montrer les plus consommatrices en temps. C'est pourquoi accélérer l'exécution de ces nids de boucles est prioritaire lorsque l'on veut améliorer les performances d'un système embarqué. Pour cela, le modèle polyédrique apporte des solutions intéressantes dans la parallélisation de nids de boucles. Cette représentation permet d'obtenir un réseau de processeurs en formulant des problèmes de programmation linéaire entière, à partir des contraintes définissant le domaine d'itération du nid de boucles et des dépendances entre les données. Elle permet également d'ajouter des contraintes afin de limiter la surface ou le temps d'exécution du circuit conçu. Les architectures régulières s'adaptent particulièrement bien à la parallélisation de nids de boucles, et les contraintes pour les obtenir sont simples à définir.

Cependant l'obtention d'un réseau de processeurs exécutant de manière spécifique l'algorithme désiré n'est pas toujours suffisant. Non content d'accélérer les calculs, nous voudrions diminuer le coût et la consommation du circuit dédié. Pour cela, limiter la surface de silicium de ce circuit semble primordial, puisqu'elle influe sur ces deux facteurs.

À cette fin, différentes méthodes de partage de ressources ont été mises au point dans le domaine de la synthèse de haut-niveau, dont certaines sont présentées ici. Malheureusement, celles-ci ne se prêtent pas toujours au partage de ressources au sein de réseaux réguliers.

Contrairement aux méthodes de partitionnement présentées en 1.3.1, la méthode de fusion de chemins de données (présentée en 1.3.2) ne semble pas adéquate pour le partage de ressources au sein des réseaux réguliers. En effet les réseaux réguliers dérivés grâce au modèle polyédrique ont de nombreux processeurs similaires et peuvent être très étendus. Chercher à obtenir un graphe de compatibilité sur de telles architectures mèneraient à une explosion combinatoire due à la fois au très grand nombre de sommets et à leur compatibilité intrinsèque, qui rendrait impossible le calcul de la clique de poids maximal.

Chapitre 2

Contribution

Face aux limitations des méthodes de fusion de chemins de données, il apparaît nécessaire de développer des techniques de partage de ressources à grain fin adaptées aux nids de boucles et pouvant exploiter la régularité de telles structures. Pour cela, nous avons cherché à mettre à profit le modèle polyédrique en conservant les informations concernant l'ordonnancement et l'allocation sous forme de polyèdres pendant la phase de partage de ressources. La première section de ce chapitre situe notre travail au sein d'un flot de synthèse de haut-niveau classique. La deuxième présente les *Polyhedral Data-Path Graphs* (ou PDPG), qui conserve ces informations, et la troisième une adaptation de l'algorithme de Moreano, Borin, de Souza et Araujo à ces graphes. Enfin, la quatrième introduit les possibilités d'optimisations que nous aborderons au cours des deux mois restants.

2.1 Flot de synthèse

La figure 2.1 présente un flot de HLS pour les réseaux réguliers simplifié. A partir d'un programme C décrivant l'algorithme pour lequel on veut un circuit dédié, on construit le SARE modélisant cet algorithme. Une fonction de temps et une fonction d'allocation sont calculées pour chacune des équations du SARE. À partir de ces fonctions, il est possible d'obtenir un PDPG décrivant le réseau régulier correspondant sur lequel on peut utiliser des techniques de partage de ressources afin de diminuer la surface de ce réseau, éventuellement après avoir pratiqué un partitionnement (1.3.1). L'étape suivante est de générer une description matérielle (en VHDL) du chemin de données du réseau. À partir de cette description et du PDPG obtenu plus tôt on peut également générer la description matérielle du contrôleur du circuit. C'est le chemin de données qui exécute tous les calculs, alors que le contrôleur active les mémoires et sélectionne les entrées des multiplexeurs afin de stocker ou d'acheminer les bonnes données. On peut enfin fournir ces deux descriptions à un outil de synthèse afin de programmer un FPGA, ou de directement synthétiser le circuit.

Nous ne nous intéressons ici qu'à une partie de ce (vaste) programme, plus précisément à celle qui s'attache au partage de ressources. Les étapes en vert sur la figure 2.1 sont celles sur lesquels nous avons travaillé. Il s'agit de la création d'un PDPG à partir d'un ordonnancement et d'une allocation donnés, de l'application de l'algorithme de partage de ressources adapté de celui de Moreano et al. et enfin de la génération d'une description matérielle du chemin de données correspondant à ce PDPG.

2.2 Polyhedral Data-Path Graph

Le but des *Polyhedral Data-Path Graphs* (ou PDPG) est de conserver les informations concernant l'allocation et l'ordonnancement des opérations au sein d'un programme ou d'une fonction sous forme polyédrique. En effet, il apparait plus simple de comparer deux domaines polyédriques appartenant à deux sommets qu'un ensemble d'autant de sommets que de points dans ces deux domaines.

Un *Polyhedral Data-Path Graph* (ou PDPG) est un graphe de chemin de données d'un programme dans lequel chaque sommet représente non pas une opération mais un ensemble d'opérations. Il s'agit en fait d'une forme de PRDG particulière (voir 1.2.1) dans laquelle les domaines associés aux sommets représentent un ensemble d'opérations du type du sommet à réaliser. Ces domaines sont définis sur

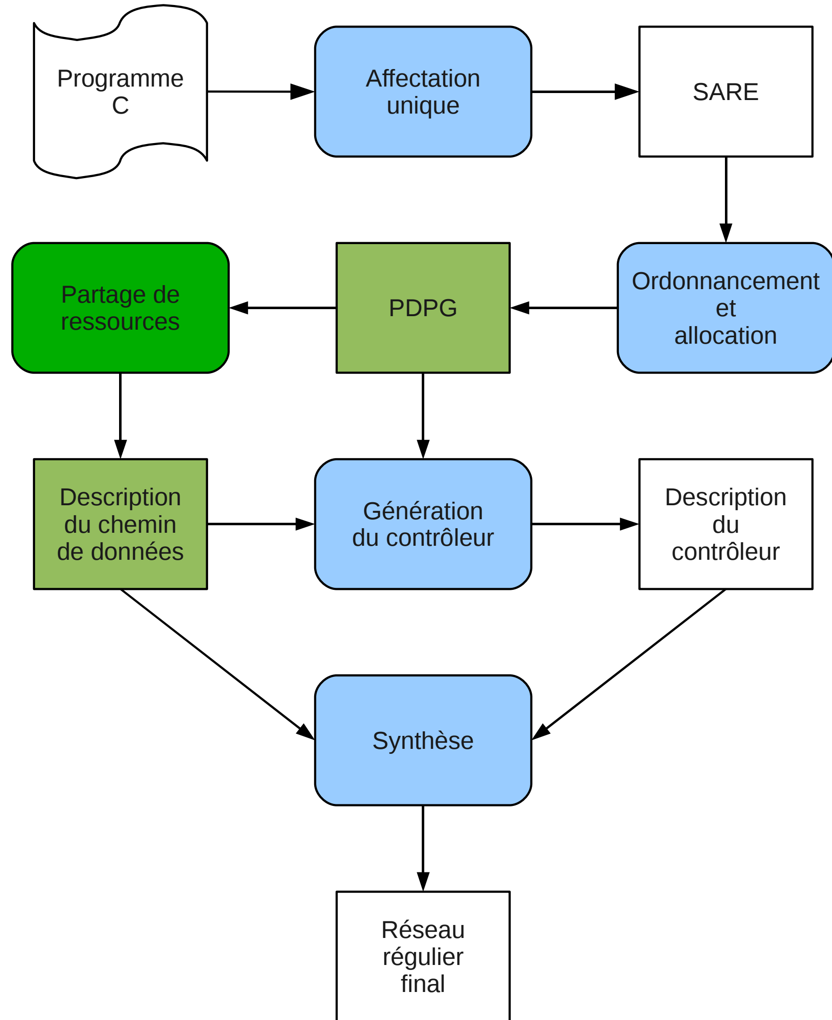


FIG. 2.1: Flot de synthèse de réseaux réguliers

les dimensions du temps et des processeurs, et non sur les indices des boucles comme dans l'exemple présenté 1.2.1.

Par exemple un sommet peut représenter un ensemble d'additions effectuées en parallèle par plusieurs processeurs sur une période de temps donnée. L'ensemble de ces opérations est représentée par un polyèdre dont chaque point représente une opération. Ce polyèdre a autant de dimensions qu'il y en a au réseau de processeurs et au temps. En général, le polyèdre sera donc en trois dimensions : deux pour l'espace du réseau de processeurs et une pour le temps. Chaque point du polyèdre indiquant sur quel processeur et à quel instant exécuter l'opération qui lui correspond.

La figure 2.2 donne un exemple d'un tel domaine polyédrique. Celui-ci n'a qu'une dimension de processeurs (p) et une dimension pour le temps (t). Ce domaine est associé à un sommet de PDPG représentant un ensemble d'additions. Chacun des points du domaine représente alors une addition particulière. La projection d'un point le long de l'axe du temps (sur l'axe des processeurs) donne le processeur sur lequel l'addition doit être exécutée et sa projection le long de l'axe des processeurs (sur l'axe du temps) donne l'instant auquel l'addition est effectuée. La projection du domaine le long de l'axe (ou des axes) du temps donne la forme du réseau régulier correspondant.

Un PDPG est donc, comme pour un chemin de données, la version ordonnancée et allouée d'un graphe de flot de données (DFG) dont chaque sommet représenterait un ensemble d'opérations. Pour une opération qui n'est pas au sein d'une boucle ou d'un nid de boucles, le polyèdre associé sera composé

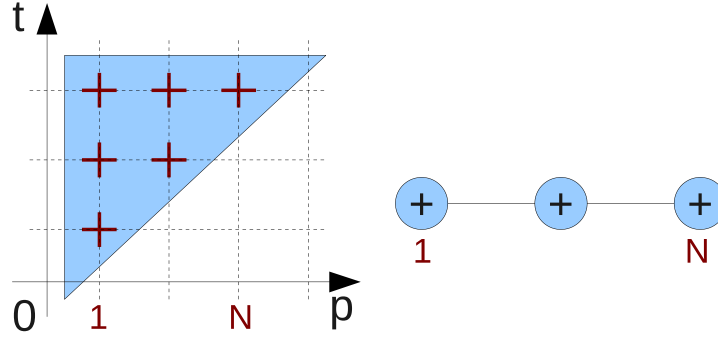


FIG. 2.2: Domaine d'un sommet de PDPG et réseau correspondant

d'un seul point, une telle opération n'étant effectuée qu'une seule fois sur un opérateur donné. Au contraire les opérations contenues dans une boucle ou un nid de boucles seront associées à des sommets dont les polyèdres contiennent autant de points que d'itérations de la boucle ou du nid de boucles.

Cette section définit plus précisément les PDPG en 2.2.1 avant d'en donner un exemple détaillé en 2.2.2.

2.2.1 Définitions

Un PDPG est un graphe orienté $G = (V, E)$ où :

- Un sommet $v \in V$ représente un ensemble d'opérations ou d'entrées/sorties associé à un domaine polyédrique $\mathcal{D}_v$. S'il s'agit d'un ensemble d'opérations, v possède un opérateur définissant les opérations réalisées ainsi que des ports d'entrées et de sorties typés. S'il s'agit d'un ensemble d'entrées (respectivement de sorties) le sommet possède un port de sortie (respectivement des ports d'entrée) typé, ainsi que le nom de la variable concernée. Le domaine $\mathcal{D}_v$ est un polyèdre dont les dimensions représentent le réseau de processeurs sur lequel sont effectués les opérations et le temps. Chaque point de $\mathcal{D}_v$ correspond à l'exécution d'une opération, ou d'une entrée/sortie sur un processeur donné à un instant donné.
- Un arc $e = (u, p_u, v, p_v) \in E$ indique un transfert de données depuis le port de sortie p_u du sommet u vers le port d'entrée p_v du sommet v. Il possède également une *fonction de dépendance* $\phi_e(i) = (\alpha i + a)$ de $\mathcal{D}_e$ vers $\mathcal{D}_v$ où i est un vecteur d'indices de $\mathbf{D}_p$, l'espace des processeurs et un domaine $\mathcal{D}_e = \mathcal{D}_u \cap \phi_e(\mathcal{D}_v)$. L'arc e représente donc le transfert de données d'un ensemble de processeurs de $\mathcal{D}_e$ aux processeurs de $\mathcal{D}_v$.

L'espace des processeurs, $\mathbf{D}_p$, est l'ensemble des dimensions attribuées au réseau régulier. Si $\mathbf{D}_p$ n'a qu'une dimension, le réseau a la forme d'une ligne, s'il en a deux, on parle de grille. La projection d'un point d'un domaine sur $\mathbf{D}_p$ indique le processeur au sein duquel l'opération associée au point sera effectuée.

Le domaine du temps, $\mathbf{D}_t$, est l'ensemble des dimensions attribuées au temps. Il n'y en a en général qu'une, mais il est possible d'en avoir plus (voir 1.3.2). La projection d'un point d'un domaine sur $\mathbf{D}_t$ indique l'instant auquel l'opération associée au point sera effectuée.

2.2.2 Exemple de PDPG détaillé

La figure 2.3 présente G_{mm}, le PDPG de la multiplication de matrices de la figure 1.1. En vert clair sont représentés les sommets d'entrées des matrices a et b, en vert foncé le sommet de la constante 0, en jaune les sommets des opérations ($+$, $*$, $R1$, $R2$ et $R3$) et en orange le sommet de sortie de la matrice résultat c.

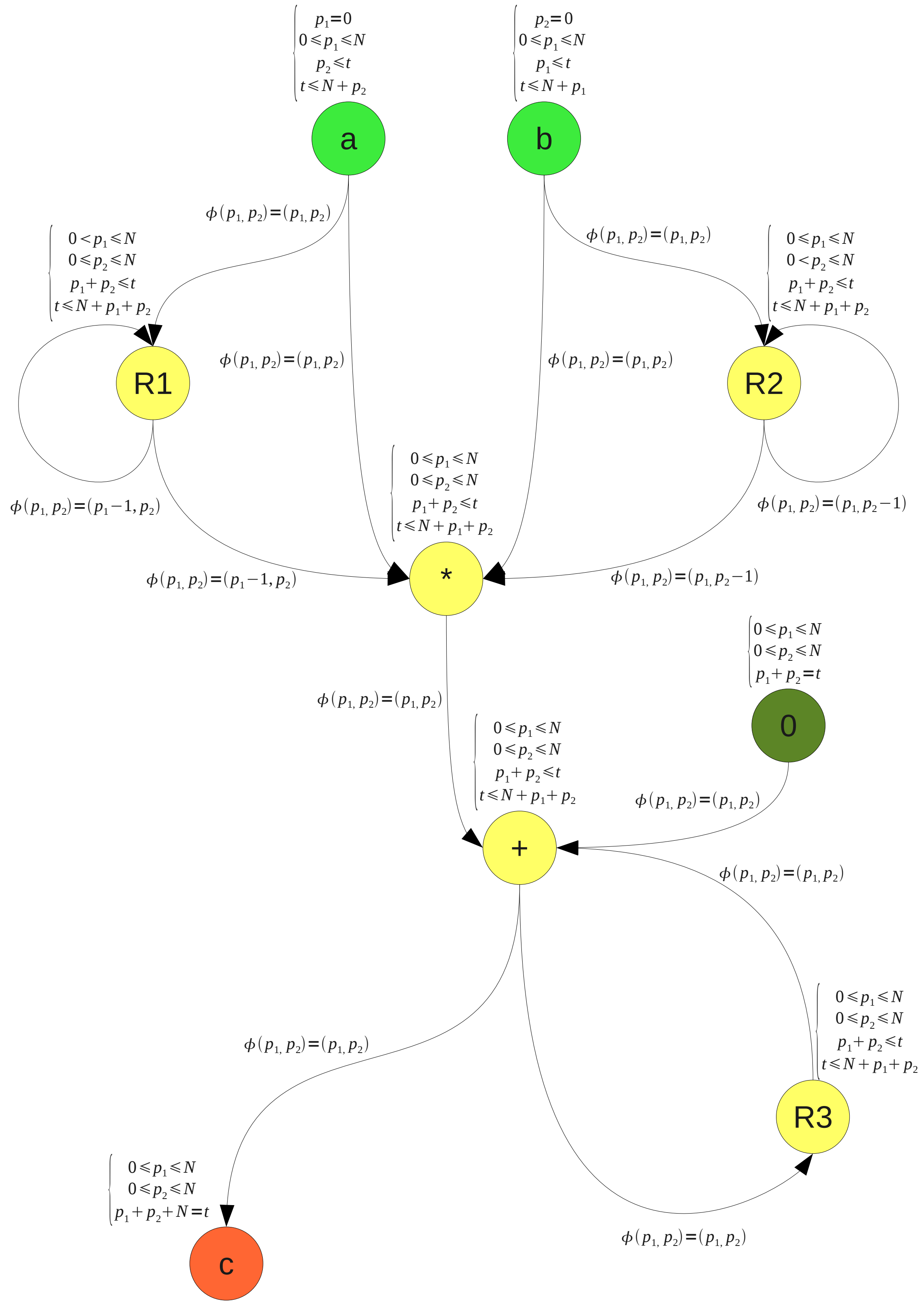


FIG. 2.3: G_{mm} : PDPG de la multiplication de matrices

Ici, $\mathbf{D}_p$ a deux dimensions p_1 et p_2 et $\mathbf{D}_t$ n'en a qu'une, t. Les domaines associés aux sommets de G_{mm} sont présents sous la forme de contraintes linéaires. Pour des raisons de place les domaines associés aux arcs ne sont pas présentés, il suffit de se rappeler qu'ils sont égaux à l'intersection du domaine de leur sommet source et l'image de celui de leur sommet cible par leur fonction de dépendance. On peut constater que le réseau régulier correspondant à G_{mm} est un réseau de forme carré de N sur N processeurs. En effet les deux dimensions du réseau (p_1 et p_2) sont toujours comprises entre 0 et N.

Les additions, multiplications et les stockages des accumulations dans les registres ont le même domaine. Les opérateurs concernés par ce domaine sont actifs de $t = p_1 + p_2$ à $t = p_1 + p_2 + N$, c'est à dire à partir du moment où t est égal à la somme de leurs indices de coordonnées, et pendant N cycles du réseau.

Les entrées sont limitées à un domaine contenant uniquement les processeurs de l'un des bords ($p_1 = 0$ pour a et $p_2 = 0$ pour b). Les éléments des matrices voyagent ensuite à travers le réseau, stockés un cycle dans le registre $R1$ ou $R2$ d'un processeur avant de passer à son voisin. C'est ce à quoi correspondent les fonctions de dépendance sur les arcs entre les sommets $R1$ et $R2$ et $*$ et sur les arcs qui bouclent sur $R1$ et $R2$. Elles expriment des dépendances par rapport au processeur précédent sur l'axe p_1 ($\phi(p_1, p_2) = (p_1 - 1, p_2)$) ou l'axe p_2 ($\phi(p_1, p_2) = (p_1, p_2 - 1)$) qui correspondent au passage d'un élément de matrice d'un processeur à l'autre.

La constante 0 n'est active que durant le premier cycle où l'additionneur auquel elle est connectée est actif ($t = p_1 + p_2$), cycle pendant lequel le registre $R3$ ne contient pas encore de valeur, afin de réaliser la première addition de l'accumulation.

Les sorties enfin ne sont actives qu'au dernier cycle où l'additionneur auquel elles sont connectées est actif, afin de récupérer le résultat de l'accumulation, qui est l'élément de la matrice c de même coordonnées que la sortie.

Ceci correspond bien au comportement du réseau régulier effectuant une multiplication de matrices décrit en 1.1.2.

2.3 Partage de ressources au sein de réseaux réguliers

Cette section présente l'adaptation de l'algorithme de Moreano et al. (cf 1.3.2) au partage de ressources au sein d'un réseau régulier. Elle est donc présentée comme la section 1.3 afin de mettre en avant les différences découlant de cette adaptation. Tout d'abord l'algorithme cherche à sélectionner la meilleure combinaison de *fusions* au sein d'un même PDPG, et non de deux DFG, en effet on peut s'assurer de l'exclusion mutuelle dans le temps de deux sommets, puisque chacun d'eux possède un domaine d'existence. C'est aussi pour cette raison que l'algorithme autorise la fusion d'autant de sommets ou d'arcs que possible, et non seulement de deux. Enfin, une fusion de sommets ne représente pas le partage d'un opérateur matériel entre plusieurs opérations, mais le partage d'ensembles d'opérateurs entre plusieurs ensembles d'opérations. Ainsi si une partie des processeurs d'un réseau régulier a deux additionneurs qui ne sont jamais utilisés au même moment le partage de ressources permettra la fusion de ces additionneurs.

2.3.1 Graphe de compatibilité

Tout comme l'algorithme de Moreano et al., notre algorithme se base sur la construction d'un graphe de compatibilité représentant les différentes *fusions* possibles au sein du PDPG. Un graphe de compatibilité correspondant au PDPG G est un graphe pondéré non-orienté $G_c = (V_c, E_c)$ tel que :

- Un sommet $v_c \in V_c$ correspond à :
  - Une *fusion* possible de l'ensemble de sommets $U \subseteq V$.
  - Une *fusion* possible de l'ensemble d'arcs $F \subseteq E$.
- Il y a un arc $e_c = (v_1, v_2) \in E_c$ si les *fusions* représentées par v_1 et v_2 sont compatibles.

Sommets et arcs fusionnables

Il convient de définir ce qu'est une fusion possible au sein du PDPG $G = (V, E)$ et du graphe de compatibilité $G_c = (V_c, E_c)$ correspondant.

Une fusion entre sommets revient à mettre en commun un opérateur matériel capable de réaliser toutes les opérations de ces sommets.

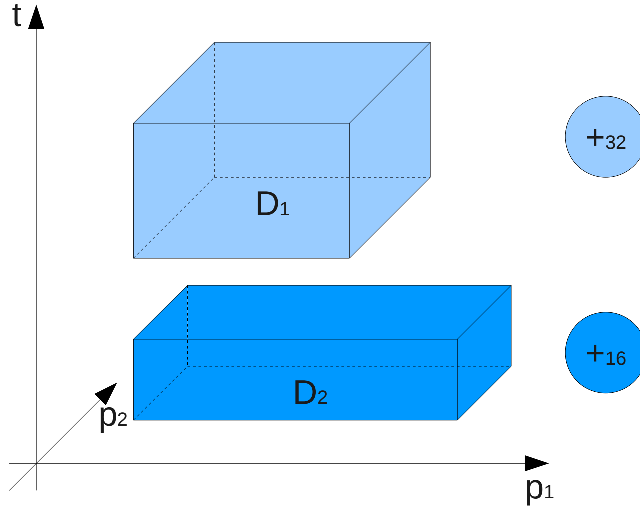


FIG. 2.4: Deux sommets fusionnables et leurs domaines

Une fusion entre arcs correspond à supprimer un ou plusieurs multiplexeurs amenés par les fusions de sommets. En effet si un opérateur est mis en commun entre plusieurs opérations, les différents opérandes en entrée doivent être sélectionnés selon l'opération à réaliser. Les différents fils connectant les opérandes à l'opérateur sont branchés sur un multiplexeur qui en sélectionne un en fonction d'un signal de contrôle. Si l'on fusionne les arcs, on peut supprimer, ou diminuer la taille du multiplexeur chargé de sélectionner l'opérande.

On peut fusionner un ensemble de sommets $U \subseteq V$ si $\forall u, v \in U$:

- $\mathcal{D}_u \cap \mathcal{D}_v = \emptyset$
- $O(u) \cap O(v) \neq \emptyset$, avec $O(v)$ l'ensemble des opérateurs matériels disponibles pouvant réaliser l'opération associée à v
- La fusion de u et v ne crée pas de cycles illégaux (voir 2.3.3)

C'est à dire si l'ensemble des domaines des sommets de U sont disjoints, en effet on ne peut partager un opérateur matériel qu'entre deux opérations qui ont lieu à des instants différents, et si il existe un opérateur matériel capable d'effectuer toutes les opérations représentées par les sommets de U.

On peut fusionner un ensemble d'arcs $F \subseteq E$ si $\forall e_1 = (u_1, p_{u_1}, w_1, p_{w_1}), e_2 = (u_2, p_{u_2}, w_2, p_{w_2}) \in F$:

- $\mathcal{D}_{u_1} \cap \mathcal{D}_{u_2} = \emptyset$
- $\mathcal{D}_{v_1} \cap \mathcal{D}_{v_2} = \emptyset$
- p_{w_1} et p_{w_2} correspondent
- $\phi_{e_1} = \phi_{e_2}$

C'est à dire, si tous les sommets sources des arcs de F peuvent être fusionnés et si tous les sommets cibles des arcs peuvent être fusionés, si leur ports correspondent et si leur fonction de dépendance est la même. On ne peut supprimer un multiplexeur sélectionnant un fil parmi deux et le remplacer par un unique fil, que si ces deux fils ont la même origine et la même destination et que si ils sont branchés sur la même entrée de cette destination, ou que l'ordre des entrées n'importe pas. Deux ports p_{w_1} du sommet v_1 et p_{w_2} du sommet v_2 correspondent s'ils occupent la même position sur leur sommet respectif ou que v_1 ou v_2 représente une opération commutative à deux entrées.

Ainsi si le sommet (ou l'arc) 1 est fusionnable avec les sommets (ou les arcs) 2 et 3 et que ceux-ci sont également fusionnables, le graphe de compatibilité contiendra non-seulement les trois sommets 1/2, 1/3 et 2/3 mais également le sommet 1/2/3. Il est obligatoire que chacun de ces sommets soit présent, et pas uniquement le sommet 1/2/3, qui peut sembler plus intéressant au premier abord, puisqu'il permet une *fusion* plus importante. En effet chaque *fusion* peut offrir des compatibilités différentes avec d'autres *fusions*.

La figure 2.4 donne un exemple de deux sommets fusionnables. Leurs domaines ($\mathcal{D}_1$ et $\mathcal{D}_2$) sont dis-

joints et ils représentent tous deux une addition, l'une sur des données de 32 bits, l'autre sur des données de 16 bits. Il existe un opérateur capable de réaliser ces deux opérations : un additionneur 32 bits.

Compatibilité des sommets

La compatibilité de deux sommets désigne la possibilité d'effectuer les deux fusions représentées par les deux sommets dans le circuit final.

Deux *fusions* de sommets représentées par v_1 et v_2 sont compatibles si elles n'ont aucun sommet en commun ($U_1 \cap U_2 = \emptyset$ avec U_1 l'ensemble des sommets de v_1 et U_2 l'ensemble des sommets de v_2) et si elles ne créent pas de cycles illégaux (voir 2.3.3). Un sommet v_1 menant à la fusion de s_1 et s_2 et un sommet v_2 menant à la fusion de s_1 et s_3 ne sont pas compatibles. Si il est possible de fusionner s_2 et s_3 il existera alors un sommet v_3 représentant la fusion de s_1, s_2 et s_3, sinon il est impossible de réaliser à la fois la fusion de s_1 et s_2 et celle de s_1 et s_3.

Une *fusion* de sommet représentée par v_v et une *fusion* d'arcs représentée par v_e sont compatibles si v_v est l'une des *sources* ou l'une des *cibles* de v_e ou si v_v est compatible avec l'ensemble des *sources* et *cibles* de v_e. En effet, une fusion d'arcs e_1/e_2 est rendue possible par la fusion des sommets sources et des sommets cibles de e_1 et e_2. Si ces fusions sont incompatibles avec une autre fusion, il en va de même pour la fusion des arcs.

Deux *fusions* d'arcs représentées par v_1 et v_2 sont compatibles si l'ensemble des *sources* et *cibles* de v_1 est compatible avec l'ensemble des *sources* et *cibles* de v_2.

Poids des sommets

Le poids $\omega(v_c)$ d'un sommet v_c représente le gain de surface amené par la fusion qui correspond à v_c. Il est égal à la différence entre la surface occupée par les opérations de v_c sans fusion ($\omega^{sf}(v_c)$) et celle occupée par ces mêmes opérations avec la fusion ($\omega^f(v_c)$).

$$\omega(v_c) = \omega^{sf}(v_c) - \omega^f(v_c)$$

Dans la suite :

- op_V pour V un ensemble de sommets, est l'opérateur matériel le moins coûteux pouvant effectuer toutes les opérations associées aux éléments de V.
- $S(o)$ pour o un opérateur matériel, est la surface occupée par o.
- $P_t(\mathcal{D})$ pour tout domaine $\mathcal{D}$ est la projection de ce domaine le long de l'axe (ou des axes) du temps, et donc sur $\mathbf{D}_p$. Cette projection reste elle-même un domaine sur les dimensions du réseau de processeurs.
- $Ehr(\mathcal{D})$ pour tout domaine $\mathcal{D}$ est le nombre de points entier de $\mathcal{D}$.
- mux_x est un multiplexeur à x entrées.

Fusion de sommets : Si v_c représente une *fusion* de l'ensemble de sommets V la surface qui correspond au sommet $v_c \in G_c$ sans partage de ressources ($\omega^{sf}(v_c)$) est la somme des surfaces occupées par chacun des ensembles d'opérations représentés par les sommets $v \in G$ qui appartiennent à v_c. Pour tout $v \in v_c$, la surface occupée est égale à la surface de l'opérateur matériel le moins coûteux pouvant effectuer l'opération associée à v multipliée par le nombre de processeurs exécutant cette opération, c'est-à-dire le nombre de points dans la projection de $\mathcal{D}_v$ sur $\mathbf{D}_p$.

$$\omega^{sf}(v_c) = \sum_{v \in V} S(op_v) * Ehr(P_t(\mathcal{D}_v)) \tag{2.1}$$

La surface correspondant à v_c, après partage de ressources ($\omega^f(v_c)$) est la somme des :

- Surfaces occupées par les opérateurs partagées entre deux ou plus sommets de G. Pour un ensemble de sommets $U \in V$ tel que $\forall u \in U, P_t(\mathcal{D}_u) \neq \emptyset$, celle-ci est égale au poids de l'opérateur matériel le moins coûteux capable d'effectuer l'ensemble des opérations associées aux sommets de U multipliée par le nombre de processeurs exécutant ces deux opérations, c'est-à-dire le nombre de points dans l'intersection des projections de $\mathcal{D}_u$ sur $\mathbf{D}_p$ $\forall u \in U$.

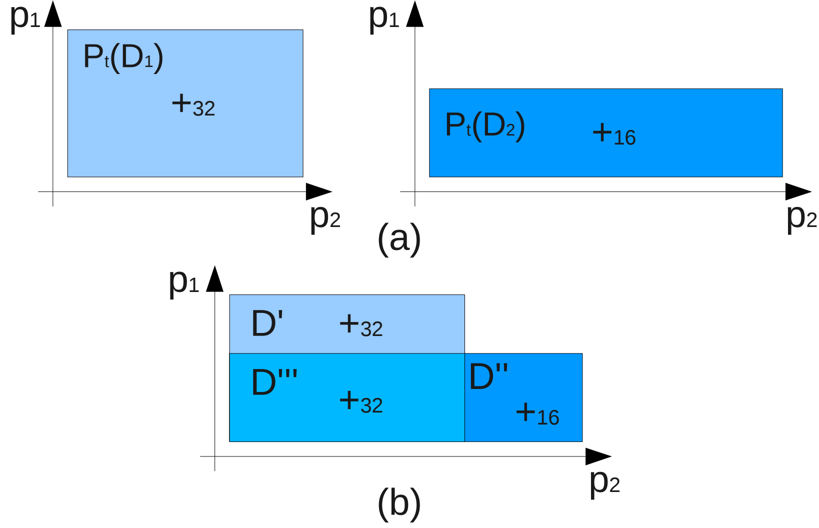


Fig. 2.5: Projections sur l'espace des processeurs des domaines de la figure 2.4

- Surfaces occupées par les opérateurs qui ne sont pas partagés. Pour un sommet $v \in v_c$ celle-ci est égale à la surface de l'opérateur matériel le moins coûteux pouvant effectuer l'opération associée à v multipliée par le nombre de processeurs exécutant *uniquement* cette opération, c'est-à-dire le nombre de points dans la différence entre la projection de $\mathcal{D}_v$ sur $\mathbf{D}_p$ et la projection de $\mathcal{D}_u, \forall u \in v_c, u \neq v$ sur $\mathbf{D}_p$.
- Surfaces des multiplexeurs en entrée des opérateurs partagés. Pour un ensemble de sommets $U \in V$ tel que $\forall u \in U, P_t(\mathcal{D}_u) \neq \emptyset$, elle est égale à la surface d'autant de multiplexeurs à $|U|$ entrées que de ports d'entrée sur les sommets fusionnés multipliée par le nombre de processeurs effectuant toutes les opérations correspondant aux sommets de U, c'est-à-dire le nombre de points dans l'intersection des projections de $\mathcal{D}_u$ sur $\mathbf{D}_p$ $\forall u \in U$.

$$\begin{aligned}\omega^f(v_c) = & \sum_{\substack{U \subseteq V \\ \bigcap\limits_{\forall u \in U} P_t(\mathcal{D}_u) \neq \emptyset}} \left(\left(S(mux_{|U|}) + S(op_U)\right) * Ehr\left(\bigcap_{\forall u \in U} P_t(\mathcal{D}_u) \right) \right) \\ & + \sum_{\substack{\forall v,w \in V \\ v \neq w}} \left(S(op_v) * Ehr(P_t(\mathcal{D}_v) \setminus P_t(\mathcal{D}_w))\right)\end{aligned} \quad (2.2)$$

La figure 2.5 présente les projections sur l'espace des processeurs des domaines ($\mathcal{D}_1$ et $\mathcal{D}_2$) de la figure 2.4 sans fusion (2.5(a)) et après fusion (2.5(b)). Le poids du sommet v_c du graphe de compatibilité qui représente la fusion de ces deux sommets est égal à la différence entre la surface occupée sans fusion et celle occupée avec fusion. La surface sans fusion est égale au coût d'un additionneur 32 bits multiplié par le nombre de points dans le domaine $P_t(\mathcal{D}_1)$ plus le coût d'un additionneur 16 bits multiplié par le nombre de points dans le domaine $P_t(\mathcal{D}_2)$. La surface occupée avec fusion est égale à la somme de :

- Du coût d'un additionneur 32 bits multiplié par le nombre de points dans le domaine commun ($\mathcal{D}'''$)
- Du coût d'un additionneur 32 bits multiplié par le nombre de points dans le domaine $\mathcal{D}'$
- Du coût d'un additionneur 16 bits multiplié par le nombre de points dans le domaine $\mathcal{D}''$
- Du coût d'un multiplexeur 32 bits à deux entrées multiplié par le nombre de points dans le domaine commun ($\mathcal{D}'''$)

Sans fusion, les processeurs de la partie du réseau régulier final correspondant à $\mathcal{D}'''$ posséderaient un

additionneur 32 bits et un additionneur 16 bits. Après la fusion, ils ne possèderont plus qu'un additionneur 32 bits. Le gain est donc égal au coût de l'ensemble des additionneurs 16 bits retirés moins l'ensemble des multiplexeurs 32 bits à deux entrées ajoutés.

Fusion d'arcs : Si v_c représente une *fusion* de l'ensemble d'arcs E, la surface qui correspond au sommet $v_c \in G_c$ sans partage de ressources est la somme des surfaces occupées par les multiplexeurs en entrée des sommets cibles des arcs $e \in E$. Elle est égale à la surface d'un multiplexeur à $|E|$ entrées multipliée par le nombre de processeurs cibles.

$$\omega^{sf}(v_c) = \sum_{\substack{F \subseteq E \\ \bigcap_{\forall f \in F} P_t(\mathcal{D}_f) \neq \emptyset}} \left(S(mux_{|F|}) * Ehr\left(\bigcap_{\forall f \in F} P_t(\mathcal{D}_f) \right) \right) \tag{2.3}$$

Après partage de ressources, il ne reste plus de multiplexeurs (tous les fils ont été fusionnés), la surface occupée est donc considérée comme nulle.

$$\omega^{f}(v_c) = 0 \tag{2.4}$$

2.3.2 Clique de poids maximal et PDPG fusionné

Le calcul de la clique de poids maximal s'effectue de la même manière que celle présentée par Moreano et al. en [8]. En effet les modifications apportées touchent à la construction du graphe de compatibilité, mais pas à la sélection de la clique, une fois celui-ci construit.

Par contre la fusion des sommets et des arcs de la clique diffère. La fusion de deux sommets (respectivement arcs) entre eux résulte en la création non pas d'un sommet fusionné mais de jusqu'à trois sommets (respectivement arcs). Un pour la partie du réseau qui effectuera les opérations des deux sommets d'origine, et deux pour les parties du réseau qui n'effectueront que les opérations d'un sommet d'origine. Le PDPG fusionné obtenu en fin de processus contiendra donc vraisemblablement plus de sommets et d'arcs que celui d'origine, mais la surface de son réseau régulier sera inférieure.

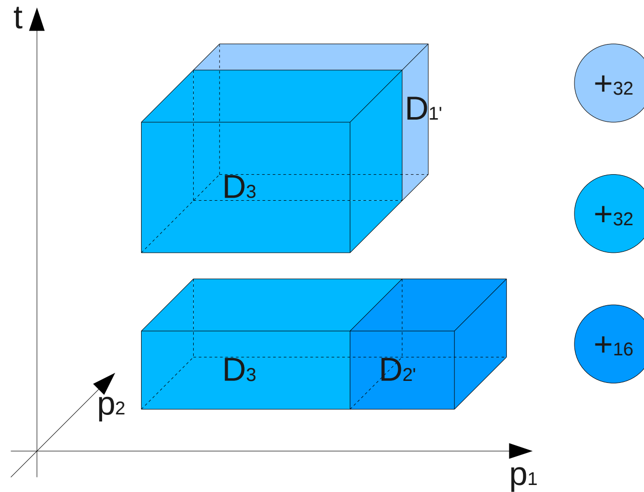


FIG. 2.6: Sommets résultant de la fusion et leurs domaines

La figure 2.6 présente les sommets obtenus après fusions des deux sommets de la figure 2.4, ainsi que leurs domaines. On obtient bien trois sommets. Les domaines $\mathcal{D}_{1'}$ et $\mathcal{D}_{2'}$ correspondent aux parties non-partagées des projections des domaines de départ ($\mathcal{D}_1$ et $\mathcal{D}_2$). Le domaine $\mathcal{D}_3$ est une union de deux polyèdres correspondant aux parties partagées des projections des domaines de départ.

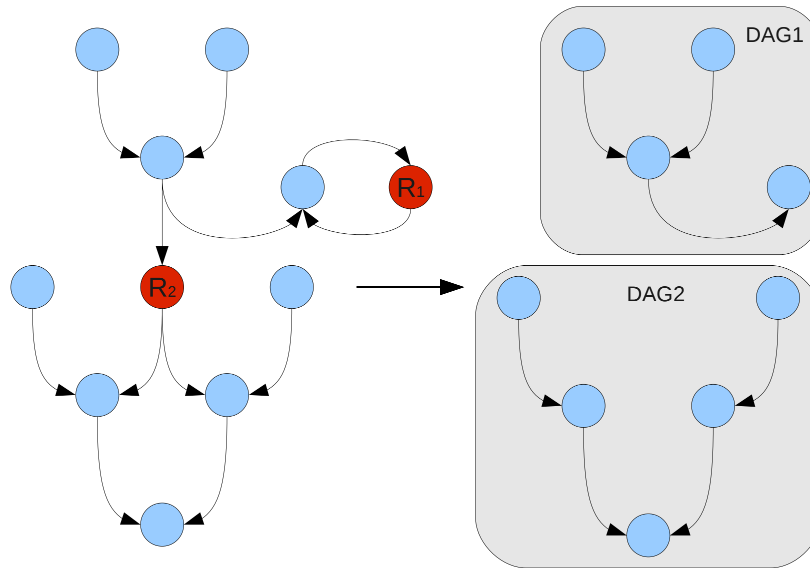


Fig. 2.7: Divison d'un PDPG en DAG par suppression des registres

2.3.3 Détection des cycles illégaux

Les outils de synthèse matérielle sont utilisés pour configurer un FPGA *(Field-Programmable Gate Array)* ou synthétiser un circuit synchrone à partir de langage de description matérielle comme le VHDL. Ils doivent détecter les erreurs ou incohérences dans la description qui rendraient un circuit non-fonctionnel. Une de ces incohérence est l'existence de cycles entre opérateurs dans lesquels il n'y a pas de registre. Pour générer un circuit synchrone les outils de synthèse doivent pouvoir mesurer le chemin critique, le chemin le plus long entre deux registres, ce qui n'est pas possible avec un cycle sans registre.

Même si les cycles créés par le partage de données sont des cycles virtuels, puisqu'ils ne sont en réalité que deux circuits superposés, la plupart des outils de synthèse ne sont pas capables de le détecter.

C'est pourquoi nous nous sommes attachés à détecter les cycles illégaux, ou considérés comme illégaux, que pourrait amener le partage de ressources. On peut discerner trois types de cycles illégaux provoqués par le partage de ressources :

- Les cycles provoqués par une seule fusion de sommets, dans ce premier cas, assez simple à détecter, la fusion considérée devient illégale, et le graphe de compatibilité ne contiendra pas de sommet correspondant.
- Les cycles provoqués par deux fusions, dans ce cas, le cycle ne peut être détecté qu'après la création de toutes les fusions de sommets et les fusions qui le provoquent restent légales, elles sont uniquement incompatibles
- Les cycles provoqués par plus de deux fusions, ici la détection intervient au même moment que pour le deuxième, mais le traitement est plus complexe.

Division du PDPG en DAG

Pour détecter les cycles illégaux, on réduit le PDPG d'origine en le divisant en DAG *(Directed Acyclic Graph)* non-connexes représentant des chemins de données combinatoires (sans registre). Pour cela, on supprime les sommets de stockage (dont l'opérateur est une mémoire) ainsi que les arcs dont ils sont la cible ou la source, comme le montre la figure 2.7. Puisqu'un PDPG est un chemin de données, il ne doit, à l'origine, contenir aucun cycle illégal (sans registre). Aucun des graphes obtenus en supprimant les sommets de stockage ne peut donc contenir de cycle. C'est à partir de ces DAG que nous pouvons détecter les cycles illégaux.

Cycles illégaux provoqués par une fusion

Une unique fusion ne peut provoquer de cycle illégal qu'à l'intérieur d'un même DAG. En effet si la fusion ne concerne que des sommets appartenant à des DAG différents, ces sommets sont tous séparés par au moins un registre au sein du PDPG d'origine, et il n'y a alors pas de possibilités de cycle illégal. Par contre si un sommet d'un DAG est fusionné avec l'un de ses successeurs, il n'y aura pas de mémoire dans le cycle ainsi créé, qui sera donc illégal (la figure 2.8 en donne un exemple). Pour éviter cela, il suffit donc de ne pas autoriser de fusion comprenant un sommet et un de ses successeur au sein d'un même DAG.

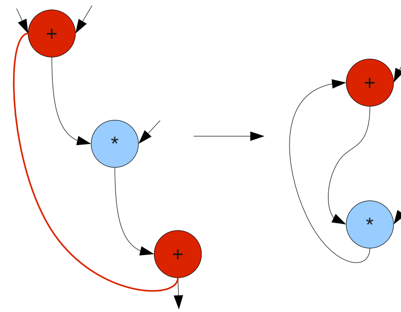


FIG. 2.8: Création d'un cycle illégal par la fusion de deux sommets d'un même DAG

Cycles illégaux provoqués par deux fusions

Un cycle illégal peut survenir suite à deux fusions entre deux mêmes DAG, comme le montre la figure 2.9. Il est dans ce cas possible de détecter le cycle en opérant un tri topologique sur les deux DAG concernés. Si les deux fusions conservent l'ordre topologique dans le graphe, elle ne créent pas de cycle illégal. Sinon, les fusions créent ensemble un cycle illégal, on ne peut donc pas les sélectionner ensemble au sein d'une clique. Elles sont incompatibles, il n'y a pas d'arcs entre les deux sommets les représentant au sein du graphe de compatibilité.

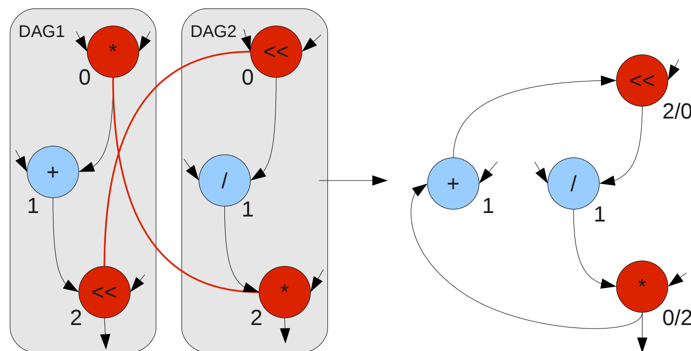


FIG. 2.9: Création d'un cycle illégal par deux fusions

2.3.4 Cycles illégaux provoqués par plus de deux fusions

Les cycles illégaux provoqués par plus de deux fusions sont un problème délicat qui ne nous est apparu que tardivement, c'est pourquoi il est traité séparément.

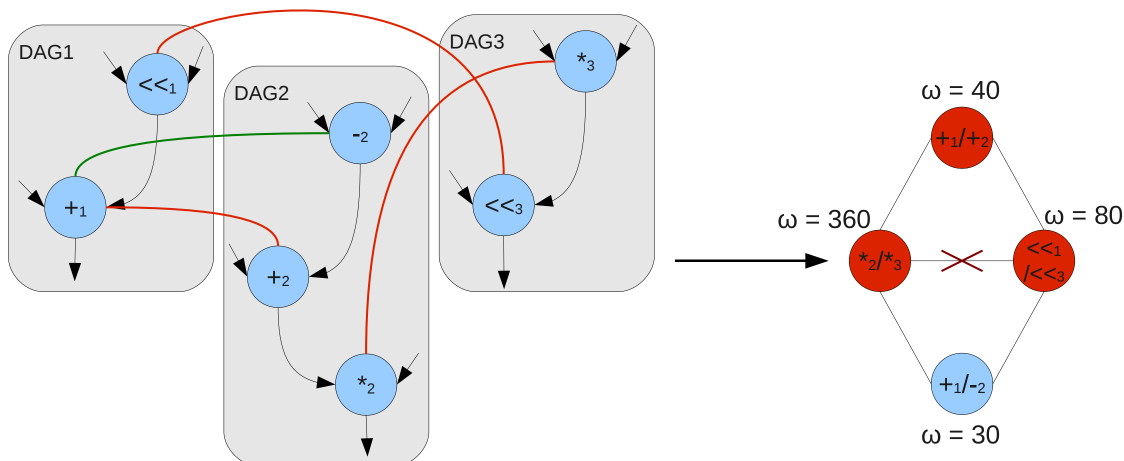


FIG. 2.10: Création d'un cycle illégal par trois fusions et graphe de compatibilité correspondant

S'il est possible de détecter ces cycles illégaux de la même manière que ceux provoqués par deux fusions, on peut leur appliquer le même traitement. En effet quand deux fusions provoquent ensemble un cycle illégal, elles sont incompatibles, mais quand trois (ou plus) fusions provoquent un cycle illégal, elles ne sont pas incompatibles deux à deux. N'importe quelle paire prise séparément parmi les fusions qui provoquent le cycle illégal est compatible et légale, c'est l'ensemble qui est illégal. Il y a plusieurs possibilités pour empêcher la création d'un cycle illégal malgré tout.

La première consiste à "rendre" incompatibles deux fusions pour briser le cycle. Mais il faut alors choisir de manière arbitraire les deux fusions qui seront rendues incompatibles, en risquant de briser une clique. La figure 2.10 présente un tel cas de figure. Dans le graphe de compatibilité les trois fusions provoquant un cycle illégal sont représentées en rouge. La clique de poids maximal est la clique composée par ces trois sommets, on ne peut donc l'accepter. Mais si un arc est choisi arbitrairement pour empêcher de sélection d'une clique créant un cycle illégal, on risque de supprimer l'arc reliant les sommets $*_2/*_3$ et $<<_1$ / $<<_3$, interdisant par là-même la deuxième clique de poids maximal ($*_2/*_3$, $<<_1$ / $<<_3$ et $+_1/-_2$) alors qu'elle est totalement légale. Cette solution revient à l'implémentation d'une fusion itérative telle qu'elle est présentée en 2.4.

La deuxième solution serait d'utiliser un algorithme rendant toutes les cliques du graphe de compatibilité et ensuite d'éliminer celles provoquant un cycle illégal, afin de choisir la clique de poids maximal parmi celles restantes. C'est une solution coûteuse, puisque le nombre de cliques risque d'exploser de manière combinatoire, incluant les plus petites d'entre elles, uniquement constituées de deux sommets reliés par un arc.

2.4 Perspectives

Il reste possible d'apporter des améliorations ou des variantes à l'algorithme présenté ici. En effet, si celui-ci s'intéresse au cas général du partage de ressources au sein des réseaux réguliers, certains cas particuliers intéressants gagneraient peut-être à être traités autrement.

Une première optimisation possible serait la détection de décalages temporels permettant de gagner en surface. Il est en effet possible que, de par l'ordonnancement et l'allocation, deux domaines se chevauchent dans le temps, interdisant le partage de ressources, alors que ce chevauchement peut être négligeable par rapport à la durée d'une exécution du programme. On peut avoir intérêt dans ce cas à retarder l'exécution d'un délai suffisant pour que les domaines ne se chevauchent plus et qu'il soit possible de partager les opérateurs matériels.

L'un des problème les plus importants auquel peut être confronter notre algorithme est une explosion combinatoire du graphe de compatibilité qui rendrait impossible le calcul de la clique de poids maximum.

Il est plus que probable que celle-ci survienne sur des PDPG suffisamment grands, c'est-à-dire, tirés de fonctions critiques suffisamment longues. Pour éviter cela, il sera donc nécessaire de définir un algorithme ne construisant pas le graphe de compatibilité complet, ou ne recherchant pas la clique de poids maximal directement sur ce dernier.

Une possibilité est de construire la fusion de manière incrémentale, de la même manière qu'en [8]. Pour cela, on pourrait n'autoriser les fusions à ne considérer que deux sommets à la fois, construire le graphe de compatibilité correspondant et définir des heuristiques afin de sélectionner les fusions conservées. La plus simple d'entre elles est bien sur de calculer la clique de poids maximal sur le graphe de compatibilité courant. On peut ensuite construire un graphe fusionné et chercher à nouveau des fusions de paires de sommets et continuer jusqu'à trouver un point fixe. Cette méthode permettrait d'éviter l'explosion combinatoire du nombre de sommets et d'arcs du graphe de compatibilité, mais ne garantirait pas un résultat optimal puisque certaines fusions de deux sommets peuvent en interdire d'autres qui se seraient révélées plus intéressantes.

Une autre possibilité permettant elle de trouver un résultat optimal serait de calculer tous les ensembles de fusions compatibles par paires de sommets et pour chacun d'entre eux construire le graphe fusionné et recommencer. Le nombre de graphes exploserait mais permettrait de ne traiter qu'un graphe de compatibilité réduit à la fois.

Chapitre 3

Mise en œuvre

Ce chapitre présente la mise en œuvre des PDPG et de l'algorithme présentés dans le chapitre précédent (respectivement section 2.2 et 2.3). Celle-ci est réalisée en JAVA et se base sur plusieurs outils déjà existants et utilisés au sein de l'équipe CAIRN. Une première section (3.1) présente brièvement les outils utilisés pour l'implémentation, une deuxième détaille cette dernière (3.2) et enfin la section 3.3 introduit les travaux restants à réaliser.

3.1 Outils utilisés

3.1.1 Librairies polyédriques

ISL *(Integer Set Library)* [13] et *polylib* [7] sont deux librairies permettant de manipuler et de transformer des polyèdres. Elles implémentent la majorité des opérations classiques sur les polyèdres (union, intersection...) ainsi que le calcul des polynômes d'Ehrhart et la transformation d'une représentation d'un polyèdre sous forme de contraintes à celle sous forme de sommets et de rayons par l'algorithme de Chernikova.

3.1.2 Méta-modèles et EMF

Les méta-modèles, ou modèles de modèles, sont un outil puissant permettant, entres autres, de définir des architectures logicielles. EMF *(Eclipse Modeling Framework)* est une infrastructure intégrée à *Eclipse* qui permet de créer et de manipuler des méta-modèles répondant au standard MOF *(Meta-Object Facility)* de l'OMG. Les fichiers *ecore* sur lesquels se base EMF permettent de créer des méta-modèles simplement à l'aide d'un éditeur graphique ou d'un diagramme UML, puis de générer les classes JAVA et une partie des méthodes associées automatiquement. Nous nous sommes basés sur ces outils pour implémenter notre adaptation de l'algorithme de Moreano, Borin, de Souza et Araujo.

Datapath Model

Le méta-modèle *Datapath Model* est un méta-modèle de circuit matériel développé au sein de l'infrastructure de compilation GECOS *(GEneric COmpiler Suite)* de l'équipe CAIRN et dont le but est de faciliter la génération de description matérielle en VHDL. Il permet de décrire n'importe quelle architecture matérielle et nous l'utilisons afin de décrire les réseaux réguliers et leurs composants matériels.

Polyhedron Model

Ce méta-modèle qui décrit les *Polyhedral Domain*, des unions de polyèdres, a été développé dans l'équipe CAIRN également. Un *Polyhedral Domain* possède un ensemble de polyèdres, eux-mêmes décrits comme des systèmes de contraintes linéaires, ainsi qu'un ensemble d'indices et de paramètres.

3.2 Implémentation

Cette section présente l'implémentation proprement dite des PDPG et de l'algorithme. Une première partie présente les méta-modèles que nous avons définis et les deux suivantes détaillent la mise en œuvre de

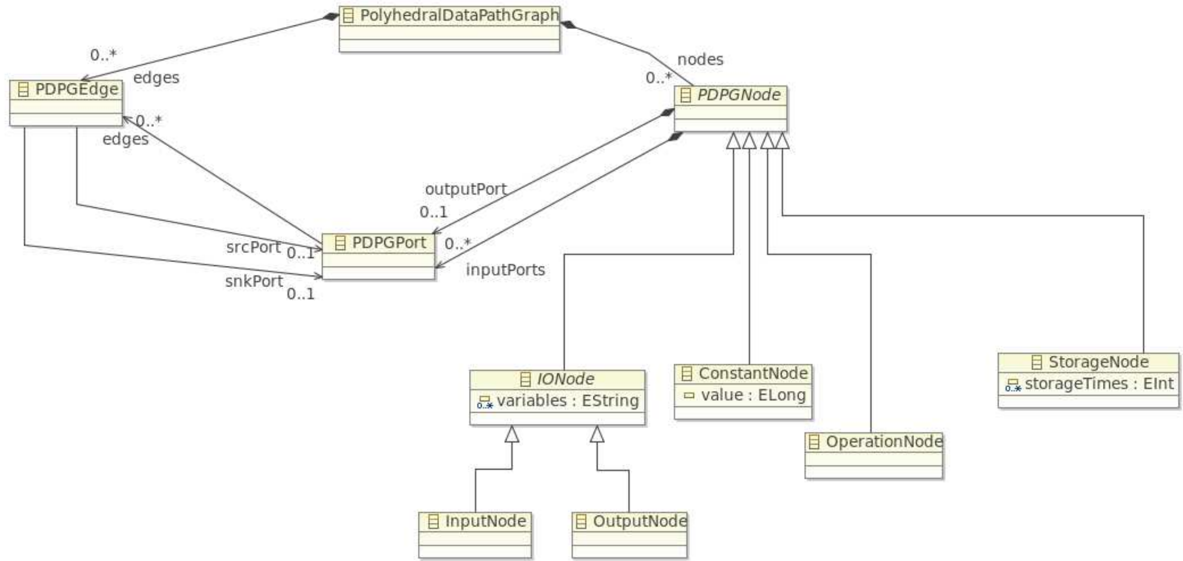


FIG. 3.1: Diagramme UML des PDPG

l'algorithme (construction du graphe de compatibilité, calcul de la clique de poids maximal et construction du graphe fusionné).

3.2.1 Méta-modèles

Nous avons défini trois méta-modèles afin de structurer notre implémentation. Le premier et le plus important d'entre eux est celui des PDPG. Le second est le méta-modèle des composants, qui permet de définir la bibliothèque d'opérateurs matériels disponibles pour la fusion, et le troisième est le méta-modèle du graphe de compatibilité.

Méta-modèles des PDPG

Le méta-modèle principal est celui décrivant les PDPG, autour desquels s'articule tout le reste de l'algorithme de partage de ressources dans les réseaux réguliers. Le méta-modèle définit les différents types de sommets (d'entrées, de sorties, de stockage, d'opérations...) et les informations qui leur sont liés telles que les ports ou domaines, représenté grâce au méta-modèle *Polyhedron Model*, ainsi que les arcs et leurs fonctions de dépendance. Comme celles-ci sont uniformes (de la forme $\phi(\vec{i}) = \vec{i} + \vec{\alpha}$) elles sont représentées par un vecteur ($\vec{\alpha}$) d'entiers.

Méta-modèle des composants

Le méta-modèle des composants définit les types et opérateurs matériels disponibles pour la création du réseau régulier. Pour définir si deux sommets sont compatibles, le programme vérifie qu'il existe un composant dans la bibliothèque capables d'effectuer chacune des opérations des sommets. Chaque composant possède la description d'un chemin de données (définie grâce à *Datapath Model*), ainsi que la liste des opérateurs ou types qu'il supporte, selon qu'il s'agit d'un opérateur, d'une mémoire, d'entrées-sorties, etc.

Chaque élément d'un PDPG correspond à un élément matériel (unités fonctionnelles, mémoires, fils...) au sein d'un circuit. Il existe donc un composant qui décrit l'architecture et le comportement de cet élément.

- Un sommet d'entrée/sortie correspond à un ensemble de pattes d'entrées ou de sorties qui permettent de connecter le circuit à d'autres composants.
- Un sommet d'opération correspond à un ensemble d'opérateurs matériels d'un type donné (additionneurs, multiplieurs...).

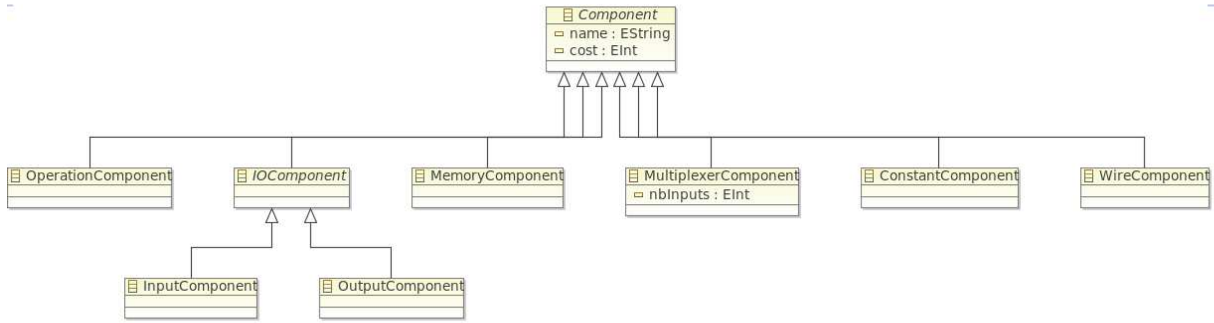


FIG. 3.2: Diagramme UML des composants

- Un sommet de stockage correspond à un ensemble de mémoires (registres, RAM...). Dans ce cas, le sommet connaît la durée, exprimée sur $\mathbf{D}_t$, le domaine du temps, pendant laquelle il doit conserver les données. En effet dans le cas d'une file de registres, le nombre d'étages de la file est égal au temps que mettent les données à traverser la file.
- Un arc correspond à un ensemble de fils connectant les éléments matériels correspondant aux sommets source et cible. La fonction de dépendance de l'arc et les coordonnées de l'élément source indique à quel élément cible le fil doit être connecté. Par exemple, si la fonction de dépendance est $\phi_e(p_1, p_2) = (p_1 + 1, p_2)$, chaque fil correspondant à e partira de l'un des opérateurs sources et sera connecté à un opérateur cible dans le processeur suivant sur l'axe p_1.
- Un port correspond à un ensemble de connexions entre un ensemble d'opérateurs et un ensemble de fils. Les ports d'entrées connectés à plusieurs arcs sont un cas particuliers. Ils correspondent à un multiplexeur, qui doit sélectionner l'un des fils correspondant à l'un des arcs en fonction du contrôle.

Méta-modèle des graphes de compatibilité

Le méta-modèle du graphe de compatibilité définit la structure d'un graphe de compatibilité, les informations contenues par les sommets et les méthodes permettant de connaitre la compatibilité entre deux sommets. Un sommet du graphe de compatibilité connaît l'ensemble des sommets (respectivement des arcs) du PDPG concernés par sa fusion, ainsi que l'ensemble des sommets (respectivement des arcs) résultant de leur fusion. Il possède également un poids (calculé selon la surface des opérateurs matériels correspondant) et un domaine qui est l'union des domaines des sommets ou des arcs concernés par sa fusion. Les sommets représentant les fusions d'arcs possèdent également un numéro de port, indiquant le port d'entrée sur lequel l'arc résultant de la fusion doit être connecté et des références vers les fusions de sommets qui contiennent les sommets sources et cibles de leurs arcs.

3.2.2 Construction du graphe de compatibilité

Au cours de la construction du graphe de compatibilité, toutes les fusions de sommets possibles sont d'abord trouvées, avant de faire de même pour les fusions d'arcs. Après quoi le poids de chaque sommet du graphe de compatibilité est calculé, avant de connecter les sommets compatibles.

Sommets et arcs fusionnables

Le programme parcourt l'ensemble des sommets du PDPG original. Pour chaque sommet s_1, il parcourt l'ensemble des sommets du PDPG et des fusions déjà réalisées. Pour chaque sommet ou fusion de sommets s_2, il compare les domaines de s_1 et s_2. Si leur intersection est vide, il cherche dans la bibliothèque de composants un composant capable de réaliser les opérations associées à s_1 et à s_2. S'il n'en existe aucun, s_1 et s_2 ne sont pas fusionnables, sinon il reste à vérifier que l'un deux sommets n'est pas ou ne contient pas (dans le cas d'une fusion de sommet) un successeur de l'autre dans un DAG (voir 2.3.3). Dans ce cas le programme crée un nouveau sommet du graphe de compatibilité représentant la fusion de s_1 et s_2.

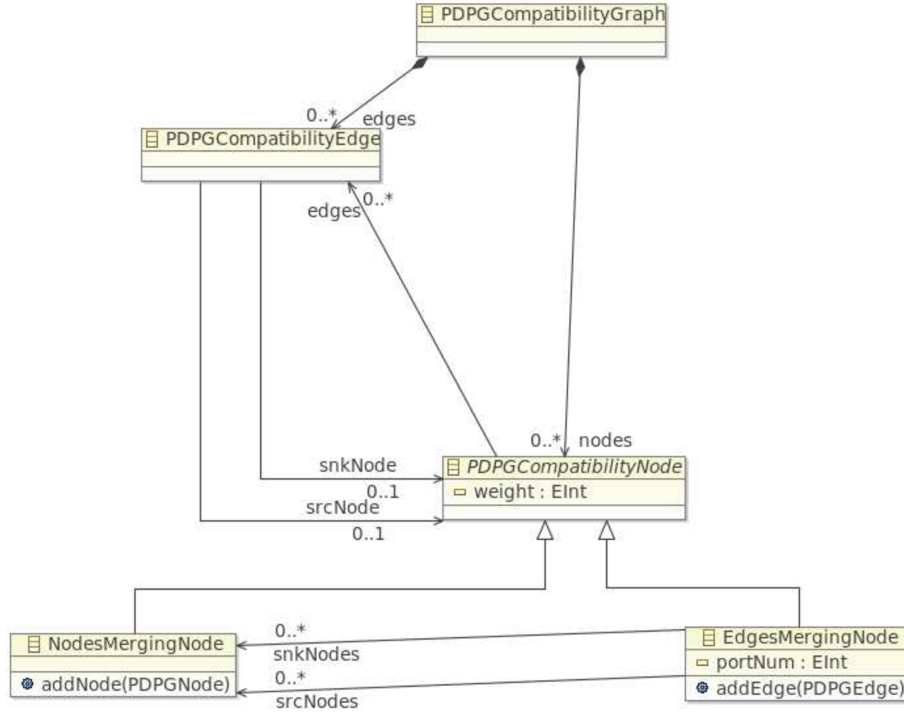


FIG. 3.3: Diagramme UML des graphes de compatibilité

Une fois toutes les fusions de sommets trouvées, le programme parcourt l'ensemble des arcs du PDPG. Pour chaque paire d'arcs, il compare les sommets sources et cibles. Si les deux sommets sources sont égaux, ou qu'ils appartiennent tous deux à un même sommet du graphe de compatibilité, et qu'il en va de même pour les deux sommets cibles, le programme compare les fonctions de dépendance. Si elle sont égales, le programme vérifie ensuite que les ports correspondent. Pour cela, ils doivent se trouver à la même position sur leurs sommets respectifs. Pour l'instant la commutativité de l'opérateur du sommet cible n'est pas prise en compte, à cause de difficultés provoquées dans la construction du PDPG fusionné.

Calcul du poids

Comme expliqué en 2.3.1, le poids de chaque fusion de sommets dépend du nombre de processeurs exécutant les opérations des sommets fusionnés, c'est-à-dire du nombre de points contenus dans la projection sur $\mathbf{D}_p$ de leurs domaines. Les intersections des projections donnent les ensemble de processeurs partagés et les différences des projections les ensembles de processeurs qui conservent leurs opérateurs (ils ne partagent aucune opération avec les autres processeurs). Il est donc nécessaire de calculer les domaines de chacun des nouveaux sommets utilisés pour connaître le nombre de composants matériels utilisés, puisqu'ils correspondent aux intersections et aux différences des domaines des sommets concernés par la fusion. Pour cela, nous créons les nouveaux sommets de manière itérative.

Pour chaque fusion, le programme prend deux sommets de la fusion, crée tous les nouveaux sommets possibles à partir de leurs domaines (au maximum trois, la partie commune et chaque partie qui n'est pas commune). Si des nouveaux sommets sont créés il retire la paire des sommets de la fusion et ajoute les sommets nouvellement créé. Il sélectionne une nouvelle paire et continue jusqu'à atteindre un point fixe. En effet après un certain nombre d'itérations, toutes les combinaisons ont été essayées et il ne reste plus dans la liste que les sommets créés qui n'ont aucune partie commune. Il n'y a donc plus de sommets ajoutés ou enlevés de la liste. Après avoir essayé toutes les paires possibles de la liste sans que celle-ci ne soit modifiée, le point fixe est atteint.

Par exemple la figure 3.4(a) présente trois sommets fusionnables et la projection de leurs domaines sur l'espace des processeurs. La figure 3.4(b) présente une première étape qui a mené à la création des sommets intermédiaires à partir des sommets 1 et 2. On va ensuite créer les sommets à partir de 3 et de

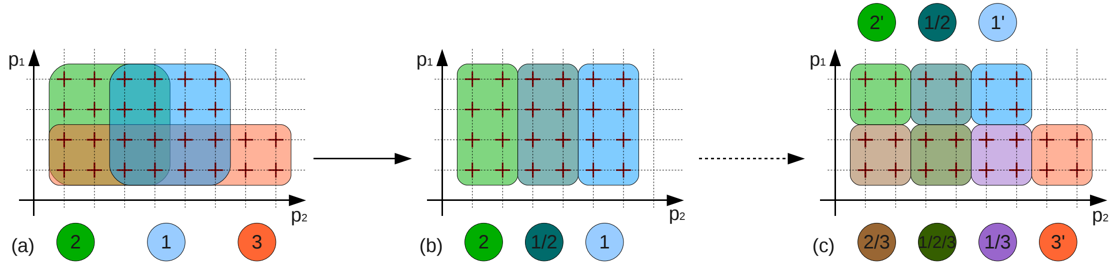


FIG. 3.4: Étapes dans la création des sommets résultants de la fusion

l'un des sommets créés et ainsi de suite jusqu'à obtenir les sept sommets présentés en 3.4(c) .

Le domaine d'un nouveau sommet s'_1 qui représente la partie non-partagée du réseau du sommet s_1 avec le sommet s_2 a pour "base" la différence entre la projection sur l'espace des processeurs de $\mathcal{D}_1$ et celle de $\mathcal{D}_2$ (voir figure 3.5). Il faut ensuite rajouter la dimension (ou les dimensions) du temps afin de récupérer un domaine complet, contenant les indications à la fois sur les processeurs, mais aussi sur les instants auxquels doivent être exécutées les opérations. On obtient alors un domaine infini sur les dimensions du temps (voir figure 3.6(a)). Pour obtenir le domaine final $\mathcal{D}'_1$ de s'_1 il suffit de faire l'intersection de ce domaine infini avec $\mathcal{D}_1$ (figure 3.6(b)).

Le domaine du nouveau sommet $s_{1/2}$ qui représente la partie partagée du réseau entre s_1 et s_2 est l'union de $\mathcal{D}_1$ et de $\mathcal{D}_2$ moins $\mathcal{D}'_1$ et $\mathcal{D}'_2$.

Une fois tous les sommets créés et leurs domaines calculés, il suffit de projeter les domaines sur l'espace des processeurs et de calculer le nombre de points à l'aide des polynômes d'Ehrhart dans ces projections, pour ensuite le multiplier par la surface de l'opérateur correspondant pour obtenir le poids correspondant à une partie de la fusion. La somme de chacun de ces poids et de ceux des multiplexeurs donne le poids du sommet dans le graphe de compatibilité.

De la même manière, pour calculer le poids des fusions d'arcs, on construit les arcs fusionnés. Chaque nouvel arc e étant associé aux arcs du PDPG d'origine dont il est issu, il suffit de compter le nombre de ces arcs d'origine pour connaître le nombre d'entrées du multiplexeur supprimé par la création de e. Grâce au nombre d'entrées et à la taille en bits des données qui transitent par les fils, on connaît le coût d'un multiplexeur, qui multiplié par le nombre de points dans la projection de $\mathcal{D}_e$ sur $\mathbf{D}_p$ donne une partie du poids du sommet du graphe de compatibilité. La somme des parties associées à chacun des nouveaux arcs résultant d'une fusion d'arcs donne le poids du sommet de cette fusion.

3.2.3 Calcul de la clique de poids maximal et construction du graphe fusionné

Le calcul de la clique de poids maximal se fait grâce à une version en JAVA de l'algorithme utilisé par Moreano et al. [10]. Cette version a été adaptée pour les graphes de compatibilité tels qu'ils sont définis dans le méta-modèle et ne comporte pas l'heuristique d'arrêt utilisée en [8].

Bien que les sommets et les arcs fusionnés ait été construits pour calculer le poids des sommets du graphe de compatibilité, il reste quelques étapes avant d'avoir le PDPG fusionné. Il faut en effet connecter tous les sommets sélectionnés dans la clique de poids maximal avec les sommets qui n'ont pas été fusionnés. Pour cela, nous avons implémenté un *visiteur* qui crée un nouveau PDPG et qui parcourt l'ensemble du PDPG d'origine. Pour chaque chaque sommet, si le sommet a été fusionné, il intègre les nouveaux sommets au nouveau PDPG, sinon il le copie dans le nouveau PDPG. Il fait de même pour les arcs, à la différence que quand un arc a subi une fusion, il faut retrouver son sommet source et son sommet cible, qui résultent eux-mêmes d'une fusion. Pour cela le programme compare le domaine du nouvel arc à ceux des sommets issus des fusions du sommet source et du sommet cible. Le domaine du nouvel arc doit être contenu dans celui de son sommet source et dans l'image par sa fonction de dépendance de son sommet cible.

FIG. 3.5: Deux projections de domaines ($\mathcal{D}'_1$ et $\mathcal{D}'_2$) sur l'espace des processeurs (a) et leur différence (b)

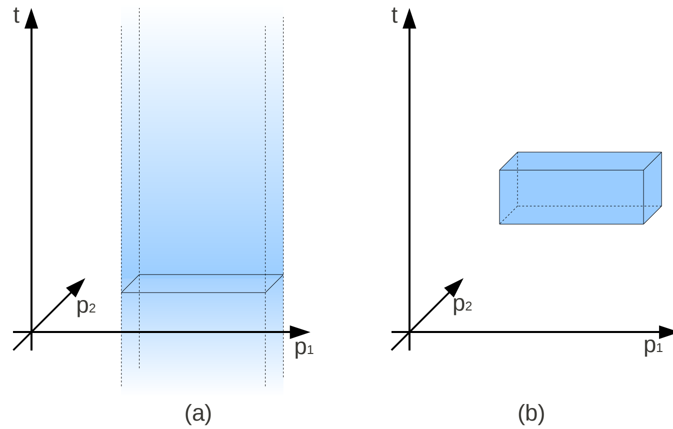


FIG. 3.6: Domaine infini (a) et son intersection avec $\mathcal{D}_1$ (b)

3.3 Travaux futurs

Il reste encore des choses à faire avant que l'algorithme ne soit intégré à un flot de synthèse complet. La première d'entre elles est d'implémenter la détection et le traitement des cycles illégaux. Il faudra ensuite tester l'algorithme sur différents exemples et implémenter la liaison entre les PDPG fusionnés et les *Datapath Model* et entre MMAlpha et les PDPG, puis éventuellement mettre en place différentes optimisations et variantes, en fonction des résultats des tests.

3.3.1 Traitement des cycles illégaux

Actuellement l'implémentation ne détecte que les cycles illégaux provoqués par une seule fusion. Pour les autres cas, il faut étudier les différentes possibilités de détection et de traitement de ces cycles.

La première est de modifier l'algorithme de calcul de clique de poids maximal afin qu'il prenne en compte la création de cycles illégaux et refuse les cliques en provoquant. L'algorithme utilise une méthode de *branch & bound* et il semble à première vue difficile d'appliquer cette modification, puisqu'il est possible de ne détecter un cycle illégal qu'après l'élimination de certaines solutions qui auraient pu contenir la clique légale de poids maximal.

La deuxième est d'utiliser un algorithme capable de calculer toutes les cliques, pour d'éliminer les cliques illégales et ensuite seulement sélectionner la clique légale de poids maximal.

La dernière, et celle qui semble la plus simple à mettre en place, est celle qui consiste à fusionner itérativement les sommets par paires plutôt que par ensemble. Comme expliqué en 2.3.3, on peut détecter et empêcher assez simplement les cycles illégaux provoqués par uniquement deux fusions, la fusion itérative règlerait donc ce problème.

3.3.2 Tests

Bien que l'application semble fonctionner, il est nécessaire d'établir des résultats expérimentaux. Ceux-ci nous permettront de mesurer l'efficacité de notre méthode et ses limites, telles que l'explosion combinatoire du graphe de compatibilité avec des PDPG trop grands. Il s'agira donc d'essayer de placer une taille (en nombre de sommets et d'arcs) sur ce "trop grand". Il existe déjà plusieurs algorithmes de traitement d'image ou de signal sur lesquels nous prévoyons de tester notre application.

- Les multiplications de matrices successives sont un motif assez courant dans les applications de traitement d'image ou de signal, et il s'agit d'un algorithme simple, qui ne devrait pas poser trop de difficultés à modéliser sous forme d'un PDPG. De plus il paraît évident que le partage de ressources est possible, puisqu'il s'agit d'opérations similaires, utilisant donc les mêmes opérateurs.
- Certaines applications d'images effectuent des traitements sur des zones de plus en plus grandes d'images, commençant par un carré de $n * n$ pixels, puis à un carré deux fois plus grand et ainsi de suite jusqu'à couvrir l'image entière. Les traitements sont toujours les mêmes et sont effectués par des processeurs spécialisés sur chaque pixel. Il est donc intéressant de réduire la surface du réseau de processeurs de la taille de l'image à celle du plus petit carré utilisé.
- Les filtres numériques FIR utilisent généralement un schéma répété de séquences de multiplication-accumulation, suivie d'une étape de calcul d'un coefficient rectificateur. Il y a là encore une régularité qui pourrait être exploitée.

Le filtre de Kalman est utilisé dans le traitement de signal et les télécommunications. Il estime l'état d'un système dynamique à partir de mesures bruitées. Il s'applique aussi bien dans le cas d'un radar que d'une communication entre une antenne-relais et un téléphone portable.

L'algorithme traite plusieurs matrices au sein des équations suivantes :

$$\begin{array}{lcl} \hat{x}_{k+1/k} & = & \Phi\hat{x}_{k/k} \\ P_{k+1/k} & = & \Phi P_{k/k}\Phi^t + b\beta b^t, P_{0/0} = I \\ V_{k+1} & = & h_k P_{k+1/k} h_k^t + 1 \\ K_{k+1} & = & P_{k+1/k} h_k^t V_{k+1}^{-1} \\ \hat{x}_{k+1/k+1} & = & \hat{x}_{k+1/k} + K_{k+1}\left[\tilde{y}_{k+1} - h_{k+1}\hat{x}_{k+1/k}\right] \\ P_{k+1/k+1} & = & P_{k+1/k} - K_{k+1} h_{k+1} P_{k+1/k} \end{array}$$

où $k = 0, 1, 2, ...$ représente le temps discret, $\tilde{y}_k$ est le vecteur de mesure de taille M, Φ est une matrice connue. $\hat{x}_{t/t'}$ est la prédiction de l'état x au temps t à partir des mesures obtenues jusqu'au temps t'. $P_{k+1/k}$ est la covariance de l'estimation d'erreur et $P_{k+1/k+1}$ est la covariance de la prédiction d'erreur. Enfin K_{k+1} est le gain de Kalman et h_k est un vecteur de réponse impulsionnelle [9].

Comme on peut le voir, le filtre de Kalman manipule de nombreuses matrices, ce qui se traduit en langage de programmation par de nombreux nids de boucle. Tel qu'il est spécifié dans les normes de téléphonie LTE *(Long-Term Evolution)*, il doit effectuer 13088 multiplications-accumulations à chaque pas, soit pour des multiplications de matrices, soit pour des multiplications de vecteurs. Encore une fois, on voit l'intérêt qu'on aurait à ne pas avoir un réseau contenant 13088 multiplieurs et autant d'additionneurs.

Il existe déjà plusieurs travaux traitant de la création d'un circuit dédié au filtre de Kalman, comme [9] qui présente l'utilisation du modèle polyédrique appliquée au filtre de Kalman au sein de MMAlpha. Les auteurs ont dérivé de l'ordonnancement obtenu automatiquement grâce à MMAlpha une architecture pour le filtre de Kalman qui se montre plus performante que les autres architectures déjà proposées. Mais il semble qu'aucune architecture réaliste pour le filtre de Kalman n'ait encore été obtenue de manière complètement automatique.

3.3.3 Liaison PDPG-Datapath Model

Afin d'obtenir une description matérielle en VHDL en fin du flot de synthèse, dans le but de programmer un FPGA ou de synthétiser un circuit, il faut réaliser la liaison entre le PDPG fusionné, résultat du partage de ressources, et le méta-modèle de chemin de données *Datapath Model*. Chaque composant défini grâce au méta-modèle présenté en 3.2.1 possède déjà la description de son chemin de données. Il faut donc parcourir les projections des domaines de chaque sommet du PDPG sur $\mathbf{D}_p$, et pour chaque point, ajouter le chemin de données du composant correspondant au sommet. Puis pour chaque arc, il

faut ajouter des fils en fonction de la fonction de dépendance ϕ. A partir de la description de chemin de données obtenue, il sera alors possible de générer la description en VHDL correspondante.

3.3.4 Liaison MMAlpha-PDPG

La construction d'un PDPG à la main est longue et complexe. En effet il faut ordonnancer et allouer les opérations, afin de calculer les domaines de chaque sommet et les fonctions de dépendances de chaque arc. Il serait bien plus intéressant de profiter d'un outil déjà existant comme MMAlpha. Une liaison JAVA-Mathematica ayant déjà été mise au point et permettant de faire des appels à Mathematica depuis des programmes JAVA, il faudra implémenter la construction d'un PDPG à partir des résultats de l'ordonnancement et de l'allocation réalisée par MMAlpha.

3.3.5 Implémentations d'optimisations et de variantes

En fonction des résultats expérimentaux, il pourra s'avérer utile de mettre en place des variantes ou des optimisations, comme celles présentées en 2.4. Elles permettront entre autres de faire face au problème de l'explosion combinatoire qui risque de se présenter passé une certaine taille de PDPG d'entrée.

Conclusion

La synthèse de circuits dédiés à l'accélération de nids de boucles est encore un problème ouvert dans le domaine de la synthèse de haut-niveau. En effet ceux-ci doivent respecter des contraintes opposées fortes. D'une part, on cherche à avoir une accélération, et donc une parallélisation, maximale pour les parties les plus consommatrices en temps de calcul des programmes, et d'autre part on souhaite que le circuit réalisant cette accélération soit le plus petit et le moins consommateur possible, afin de l'intégrer à des systèmes multimédias portables (téléphones, appareils photos numériques...).

Ce rapport a présenté plusieurs techniques de synthèse de haut-niveau dédiées soit à l'accélération de nids de boucles, soit au partage de ressources. Nous nous sommes basés sur ces méthodes pour proposer une nouvelle technique de partage de ressources au sein des réseaux réguliers, circuits massivement parallèles destinés à l'accélération des nids de boucles.

Celle-ci adapte un algorithme déjà existant réalisant la fusion de chemin de données de tâches critiques, à partir d'un graphe de compatibilité sur lequel est calculé une clique de poids maximal. Une adaptation était nécessaire pour pouvoir appliquer cette méthode basée sur la fusion opérateur à opérateur aux nids de boucle. En effet ceux-ci contiennent un trop grand nombre d'opérateurs identiques, qui amène à l'explosion combinatoire du graphe de compatibilité, sur lequel il ne serait alors plus possible de calculer la clique de poids maximal.

Pour palier ce problème, nous avons exploité la régularité intrinsèque des nids de boucles, à travers le modèle polyédrique, un outil mathématique puissant qui permet de représenter et de manipuler aisément les nids de boucles sous forme de polyèdres. A partir de cette représentation, il existe des méthodes, telles que la méthode des sommets, calculant une fonction d'ordonnancement et une fonction d'allocation, de manière à ce que chaque opération ait un instant et un opérateur d'exécution.

Pour fusionner des chemins de données polyédriques nous avons défini un graphe conservant les informations sur l'ordonnancement et l'allocation des opérations d'un nid de boucles sous forme de polyèdre. A partir de ce graphe nous construisons un graphe de compatibilité, indiquant toutes les fusions d'opérateurs possibles, et leur compatibilité, sur lequel nous résolvons un problème de clique de poids maximal afin de sélectionner la combinaison de fusions qui minimise la surface du circuit final.

L'implémentation est en cours, mais n'a pas encore pu être testée, à cause du problème posé par la création de cycles illégaux dans un circuit par la fusion de plusieurs sommets. Ce problème n'est pas apparu plus tôt puisqu'il découle de notre décision d'autoriser la fusion de plus de deux opérateurs à la fois, chose impossible dans l'algorithme original. Nous étudions actuellement plusieurs solutions possibles pour le résoudre. Nous pourrons ensuite intégrer notre algorithme au sein d'un flot de synthèse de haut-niveau complet utilisant les outils de l'équipe CAIRN, afin de générer la description matérielle d'un circuit à partir d'un programme C.

Bibliographie

[1] Stephan Balev, Patrice Quinton, Sanjay V. Rajopadhye, and Tanguy Risset. Linear programming models for scheduling systems of affine recurrence equation - a comparative study -. *Publication interne IRISA*, 1161, 1998.

[2] Alain Darte, Yves Robert, and Frédéric Vivien. Parallelism detection in nested loops. In *Scheduling And Automatic Parallelization*, pages 216–226. 2000.

[3] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems*, 7(1) :159–172, 2002.

[4] Paul Feautrier. Some efficient solutions to the affine scheduling problem Part I One-dimensional Time. 1993.

[5] Paul Feautrier. Some efficient solutions to the affine scheduling problem Part II Multidimensional Time. 1993.

[6] S. Y. Kung. *VLSI array processors.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[7] Vincent Loechner, Bd. S. Brant, and F-Illkirch. PolyLib : A Library for Manipulating Parameterized Polyhedra, 1999.

[8] Nahri Moreano, Edson Borin, Cid C. de Souza, and Guido Araujo. Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(7) :969–980, 2005.

[9] Aurelien L. T. Mozipo, Daniel Massicotte, Patrice Quinton, and Tanguy Risset. Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha. In *International Conference on Parallel Computing in Electrical Engineering*, pages 201–206, 1998.

[10] Patric R. J. Östergård. A new algorithm for the maximum-weight clique problem. *Nordic J. of Computing*, 8(4) :424–436, 2001.

[11] Patrice Quinton and Yves Robert. *Algorithmes et architectures systoliques.* MASSON, 1989.

[12] Sanjay V. Rajopadhye. Dependence Analysis and Parallelizing Transformations. In *The Compiler Design Handbook*, pages 329–372. 2002.

[13] Sven Verdoolaege. Integer Set Library : Manual, 2010.

[14] Christophe Wolinski, Krzysztof Kuchcinski, Erwan Raffin, and François Charot. Architecture-Driven Synthesis of Reconfigurable Cells. In *DSD*, pages 531–538, 2009.