View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HAL-CEA

Ordonnancement, assignation et transformations dynamiques de graphe simultanés pour projeter efficacement des applications sur CGRAs

Thomas Peyret, Gwenole Corre, Mathieu Thevenin, Kevin Martin, Philippe Coussy

To cite this version:

Thomas Peyret, Gwenole Corre, Mathieu Thevenin, Kevin Martin, Philippe Coussy. Ordonnancement, assignation et transformations dynamiques de graphe simultanés pour projeter efficacement des applications sur CGRAs. Pascal Felber, Laurent Philippe, Etienne Riviere, Arnaud Tisserand. ComPAS 2014 : conférence en parallélisme, architecture et systèmes, Apr 2014, Neuchatel, Suisse. <hal-00985815>

HAL Id: hal-00985815

https://hal.archives-ouvertes.fr/hal-00985815

Submitted on 30 Apr 2014

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.

ComPAS'2014 : Parallélisme/ Architecture / Système
Neuchâtel, Suisse, du 23 au 25 avril 2014

Ordonnancement, assignation et transformations dynamiques de graphe simultanés pour projeter efficacement des applications sur CGRAs

Thomas Peyret, Gwenolé Corre, Mathieu Thevenin, Kevin Martin, Philippe Coussy

CEA, LIST, Laboratoire Capteurs et Architectures Électroniques
F-91191 Gif-sur-Yvette, France
thomas.peyret@cea.fr, gwenole.corre@cea.fr, mathieu.thevenin@cea.fr

Université de Bretagne-Sud, Lab-STICC
56100 Lorient, France
kevin.martin@univ-ubs.fr, philippe.coussy@univ-ubs.fr

Résumé
Porter une application sur une architecture reconfigurable à gros grain est une tâche complexe qui reste encore souvent réalisée entièrement ou partiellement manuellement. Cet article présente un flot original de synthèse automatisé basé sur des étapes d'ordonnancement et d'assignation simultanées. L'approche proposée parcourt en sens inverse les nœuds du modèle formel extrait à partir du code de l'application compilé pour le transformer dynamiquement uniquement si nécessaire. Les résultats des expériences montrent que l'approche proposée permet une meilleure exploration de l'espace de solution et obtient la meilleure latence dans 90% des cas.

Mots-clés : CGRA, Ordonnancement, Assignation

---

1. Introduction

Durant les vingt dernières années, beaucoup d'Architectures Reconfigurables à Gros Grains (CGRAs) ont été proposées pour accélérer les applications multimédia. Un CGRA, pour *Coarse Grained Reconfigurable Architecture*, est généralement composé d'Unités de calculs (PEs), appelées tuiles, reliées entre-elles par un réseau d'interconnexion. Les architectures de type CGRA présentent un compromis prometteur entre les architectures *many-core* et les Architectures Reconfigurables à Grains Fins (FPGAs) en regroupant les avantages de ces deux domaines [18]. De nombreux travaux sur ces architectures ont été réalisés en faisant varier des paramètres tels que l'homogénéité des tuiles, la présence et/ou la taille des Files de Registres (RFs), le type d'opérateur ($+, \times, -\ldots$), le réseau d'interconnexion (mesh simple, tore, X,...) [17, 2, 15, 12, 8]. La figure 1 donne un exemple classique de CGRA possédant seize tuiles homogènes contenant chacune un opérateur générique, une RF interne et un registre de sortie. Le réseau d'interconnexion est un mesh 2D torique régulier.
Pour exécuter une application sur un CGRA, il faut que ses opérations soient ordonnancées et assignées sur les opérateurs et que ses variables soient placées dans les registres. Ce processus,

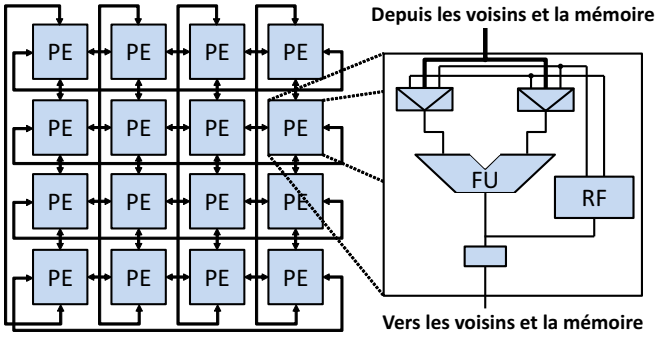

FIGURE 1 – Un CGRA 4 × 4 possédant une RF par tuile et un réseau d'interconnexion de type mesh 2D torique.

appelé « *mapping* », doit respecter les dépendances de contrôles et de données de l'application. Cette tâche complexe ne peut plus être réalisée à la main en raison du nombre croissant d'opérations dans les codes applicatifs. Elle doit donc être automatisée. Pour ce faire, l'application est généralement représentée formellement sous forme de graphe de type « *Control Data Flow Graph* » (CDFG) obtenu post-compilation. Le CGRA est lui aussi généralement abstrait en une description formelle. Son architecture (en particulier son réseau d'interconnexion) contraint très fortement l'obtention de *mappings* dont l'objectif, dans la majeure partie des cas, est de maximiser les performances temporelles de l'application (*i.e.* minimiser la latence et/ou maximiser le débit).

L'ordonnancement et l'assignation sont tous deux connus comme étant des problèmes NP-complets [9] et de très nombreuses approches ont été proposées pour les résoudre. Une solution intuitive consiste à résoudre l'intégralité du « *mapping* » par des méthodes exactes [1, 16]. L'inconvénient de ce type d'approche est qu'elle ne passe pas à l'échelle et se limite donc à des cas d'étude. Pour simplifier le problème, il est possible de séparer ordonnancement et assignation et de tenter de les résoudre de façon séquentielle. Il est dans ce cas possible d'utiliser des méthodes exactes, des heuristiques, des méta-heuristiques ou encore des combinaisons de ces méthodes comme dans [9, 13, 4]. Cependant, comme l'ordonnancement et l'assignation sont des problèmes interdépendants, résoudre l'un sans avoir conscience de l'autre peut conduire au final à ne pas trouver de solution [5].

Pour pallier à ce problème, des approches unifiées ont été développées comme celles dans [11] où l'ordonnancement et l'assignation sont résolus en utilisant une méthode de recuit simulé. Son évolution [3] permet de lever une des restrictions classiquement imposées : utiliser une RF possédant plus d'un registre dans les tuiles du CGRA. Cependant, ce type d'approche converge généralement très lentement vers une solution optimale. Il faut donc trouver un compromis entre une bonne exploration et le temps de synthèse.

Récemment, une nouvelle méthode a été présentée pour résoudre l'ordonnancement et l'assignation en combinant une heuristique et une méthode exacte [6]. Son extension [7] permet d'étendre la méthode pour des CGRAs possédant des RFs de plus de un registre. Elle se base d'une part, sur des transformations statiques du « *Data Flow Graph* » (DFG) qui facilitent le processus de *mapping* en intégrant dans le DFG certaines contraintes architecturales (nombre d'opérateurs, capacité de communication) et temporelles ; et d'autre part, sur l'algorithme de Levi [10] qui permet « de trouver » le DFG dans le modèle d'architecture par résolution exhaustive du problème du sous-graphe commun maximal (MCS). Deux transformations de graphes

sont disponibles : (1) le recalcul, qui duplique une opération pour répartir les opérations filles entre l'original et la copie ; et (2) le routage qui permet d'expliciter une dépendance de donnée dans le temps. Malheureusement, comme elles sont réalisées *a priori*, il est difficile de savoir lesquelles seront véritablement utiles et pertinentes au moment où elles sont faites.
Cet article présente une nouvelle approche permettant de porter des applications sur tout type de CGRA. Elle utilise un algorithme d'ordonnancement et d'assignation simultanés basé sur une heuristique couplée à une méthode exacte et intègre des transformations de graphe dynamiques. L'ordonnancement dérive d'un « *list-scheduling* » appliqué en parcourant le graphe dans un ordre topologique inverse et l'assignation est réalisée par une version incrémentale de l'algorithme de Levi simplifiée par un élagage judicieux. Comme les transformations de graphes sont réalisées dynamiquement et uniquement si besoin, l'espace de solutions est efficacement exploré.
La suite de l'article est organisée comme suit : la section 2 présente notre méthode ; la section 3 décrit le protocole expérimentale et l'analyse les résultats obtenus ; la section 4 conclut.

2. Méthode proposée

Le flot de synthèse proposé est présenté dans la figure 2. Ses entrées sont une spécification fonctionnelle d'une application écrite en langage de haut niveau (C/C++) et un modèle d'architecture CGRA respectant les conditions décrites dans la sous-section suivante. Tout d'abord, le code de l'application est compilé pour extraire un CDFG formel. Le CDFG et le CGRA sont ensuite utilisés pour générer des solutions de *mappings*. L'approche proposée permet d'explorer l'espace des solutions par l'utilisation conjointe de transformations de graphes, d'une heuristique d'ordonnancement et d'une méthode assignation reposant sur une méthode exacte simplifiée par un élagage judicieux. Le CDFG est transformé dynamiquement pour trouver une solution pendant l'ordonnancement et l'assignation. À l'image de ce qui est fait dans [6], deux types de transformations de graphes sont proposés : le routage, qui permet de conserver les dépendances de données, et la scission, qui permet de réduire la pression sur l'interconnexion. Notre algorithme « route » un nœud quand l'ordonnancement échoue et choisit en le routage et la scission si c'est l'assignation qui échoue (voir section 2.2.3). L'objectif de notre flot de synthèse est d'explorer l'espace de conception pour minimiser la latence sous contrainte de ressources.

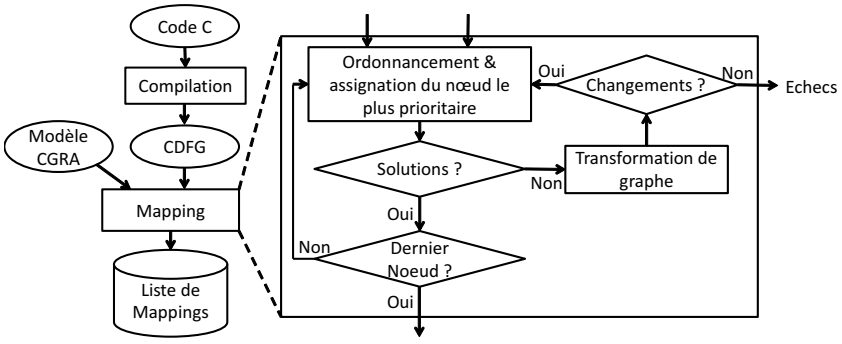

FIGURE 2 – Flot complet et cœur de l'algorithme.

2.1. Modèle d'application et de CGRA

Un CDFG est composé d'un « *Control Flow Graph* » (CFG) et d'un ensemble de *basic blocks* représenté par des DFGs. Un DFG est un graphe bipartite orienté acyclique composé d'arcs et de deux type de nœuds : opération et variable. Un nœud de variable est systématiquement associé au nœud de l'opération qui l'a créé (liaison un-un) et est relié aux consommateurs de son résultat. Dans notre modèle, en complément des nœuds de calcul ($+, \times, -\ldots$), un nouveau type d'opération est défini : la mémorisation. Un nœud de mémorisation (et son nœud de variable associé) peut être ajouté par l'algorithme d'ordonnancement quand cela est nécessaire pour respecter les dépendances de données. L'utilité de ce type d'opération est de rendre explicite les dépendances de données le long des cycles d'ordonnancement (*e.g.* expliciter la dépendance entre le nœud 2 et le nœud 4 sur un cycle dans la figure 3(c) ou entre les nœuds 3 et 8 dans la figure 4).

Le modèle d'architecture du CGRA est un graphe bipartite orienté composé d'arcs et de deux types de nœud : opérateur et registre. L'aspect temporel est implicitement représenté par la connexion entre la sortie d'un registre et l'entrée d'un opérateur. Dans ce modèle, on définit deux sous-types d'opérateur : opérateur de calcul et opérateur de mémorisation. Les opérateurs de calcul représentent la mise en œuvre d'une opération ($+, \times, -\ldots$). Les opérateurs de mémorisation sont associés à un registre et permettent de représenter explicitement l'opération de conservation d'une valeur dans un registre. Les connexions entre les registres et les opérateurs dépendent du réseau d'interconnexion : le modèle prend donc explicitement en compte les contraintes de l'interconnexion au travers des arcs. La RF, si elle est présente, permet de conserver un certain nombre de valeurs au sein d'une tuile. Un exemple de cette représentation est donné dans les figures 3(a) et (b). Ce modèle est très versatile et peut représenter des CGRAs possédant des caractéristiques très variées comme :

- des tuiles homogènes ou hétérogènes ;
- des tuiles avec ou sans RF ;
- la présence ou l'absence de RF partagée entre plusieurs tuiles ;
- un réseau d'interconnexion régulier ou irrégulier ;
- des opérateurs exécutant des opérations sur un ou plusieurs cycles.

Pour permettre l'utilisation de méthodes issues de la théorie des graphes, les équivalences suivantes sont définies entre les deux modèles :

- les nœuds de variable avec les nœuds de registre ;
- les nœuds d'opération de calcul avec les nœuds d'opérateur de calcul ;
- les nœuds d'opération de mémorisation avec les nœuds d'opérateur de mémorisation et de calcul (permettant ainsi à un opérateur d'effectuer un « NOP »).

De cette manière, ces deux modèles sont homomorphiques. Obtenir un *mapping* revient à trouver le graphe du DFG à l'intérieur de celui du CGRA [6, 7]. Une illustration est donnée dans la figure 3 où le DFG de (c) est placé en gris foncé dans le modèle du CGRA de (b).

2.2. Algorithme de mapping

Cette partie présente l'algorithme de mapping que nous proposons. Dans les grandes lignes, l'approche proposée fusionne les étapes d'ordonnancement et d'assignation pour éviter les inconvénients des approches traditionnelles [9, 13, 4, 6, 7]. L'idée est d'ordonnancer un nœud et de vérifier immédiatement s'il existe au moins une solution d'assignation pour l'ensemble des nœuds ordonnancés en s'appuyant sur les solutions trouvées précédemment. Si c'est le cas, l'algorithme passe à l'opération suivante, sinon, le graphe est transformé en fonction du problème rencontré. Une passe dite d'élagage est exécutée à la fin de chaque cycle d'ordonnancement pour diminuer le nombre de solutions partielles. Chacun de ces aspects est maintenant détaillé

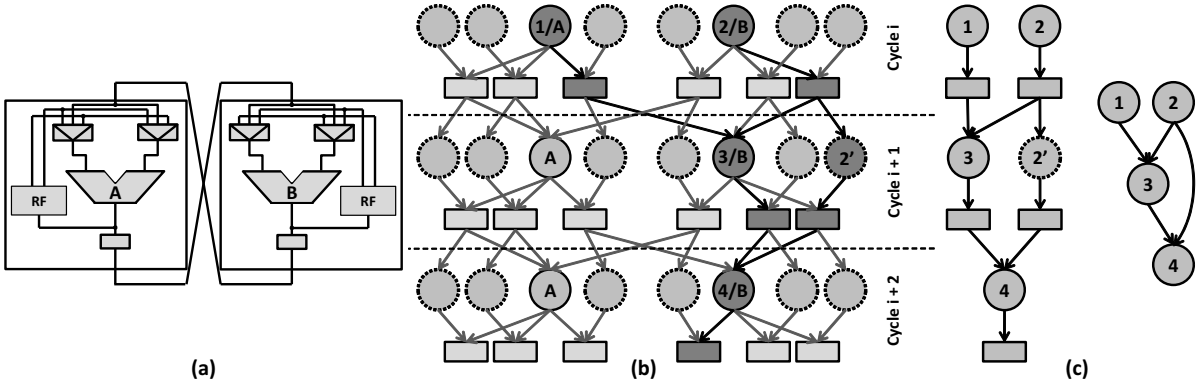

FIGURE 3 – (a) Un CGRA 2×1 possédant deux registres par RF, (b) modèle sous forme de graphe équivalent à (a) sur trois cycles, (c) à droite : DFG initial, (c) à gauche : Modèle sous forme de graphe homomorphique à (b). Un *mapping* possible du DFG de (c) est représenté en gris foncé dans (b). Les mémorisations sont en pointillés et les registres ont une forme rectangulaire.

plus précisément.

2.2.1. Ordonnancement

Notre approche utilise un ordonnancement de type « list scheduling » qui est une heuristique dans laquelle les nœuds ordonnançables sont triés par ordre de priorité. Notre fonction de priorité utilise en premier critère la mobilité des nœuds, comme défini dans [14], puis, pour les nœuds ayant la même mobilité, le nombre d'arcs sortants. En effet, un nœud possédant un nombre d'arcs sortants élevé est *a priori* plus difficile à placer. Commencer par ces nœuds est une stratégie raisonnable pour obtenir la latence la plus petite possible. Par exemple, le nœud 5 de la figure 4(b) possède une priorité plus élevée que les nœuds 3 et 4 à cause de son nombre d'arcs sortants.
Le DFG est ordonnancé en suivant un ordre topologique inverse de parcours. Ainsi, une opération n'est ordonnançable que si tous ses enfants ont déjà été ordonnancés (*e.g.* le nœud 1 dans la figure 4(d) n'est pas encore ordonnançable à cause du nœud 3). Ce choix permet un plus grand nombre de transformations de graphe comme cela est expliqué au paragraphe 2.2.3.

2.2.2. Assignation

Contrairement à la méthode présentée dans [6] et [7], qui utilise une version régulière de l'algorithme de Levi (*i.e.* où l'algorithme cherche des solutions en profondeur d'abord), notre méthode d'assignation est une version incrémentale de l'algorithme de Levi. Elle consiste à considérer l'ensemble des solutions partielles précédentes et pour chacune d'elles à trouver l'intégralité des solutions d'assignation pour le couple de nœuds ordonnancé (opération-variable). En d'autres termes, cet algorithme ajoute le couple de nœuds à placer au sous-graphe considéré précédemment (matérialisé par les nœuds sous le trait pointillé dans la figure 4). Quand le sous-graphe considéré contient l'intégralité des nœuds, le DFG est ordonnancé et assigné. Les solutions trouvées sont alors des *mappings* complets.

2.2.3. Transformations de graphe

Le DFG doit être transformé si un nœud n'est pas ordonnançable (*e.g.* les nœuds de mémorisation 3_1 et 1_1 doivent être ajoutés dans la figure 4(b) et (e) car respectivement les nœuds 3

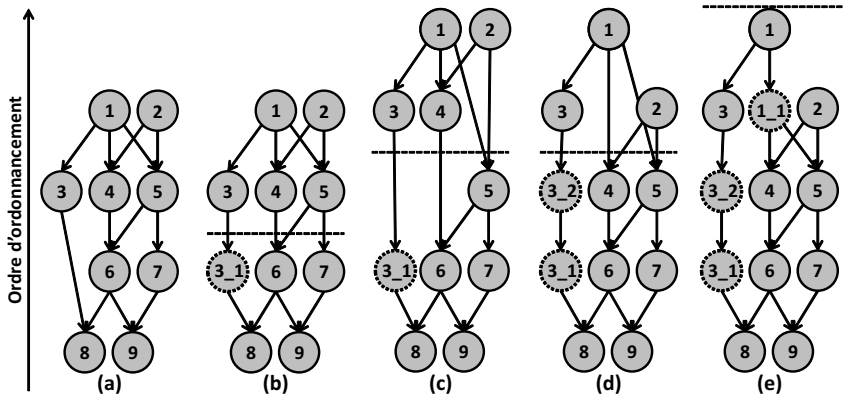

FIGURE 4 – Exemple d'ordonnancement et transformations d'un DFG sur un CGRA de deux tuiles (comme dans la figure 3). La ligne horizontale montre la limite entre les nœuds ordonnancés et non ordonnancés. Les nœuds de mémorisation sont en cercles pointillés. Chaque cercle représente un nœud d'opération et son nœud de variable associé. (a) DFG initial, (b) après le routage du nœud 3, (c) après l'ordonnancement du nœud 5, (d) après avoir re-routé le nœud 3, (e) DFG intégralement ordonnancé.

et 1 ne sont pas ordonnançables) ou si l'algorithme d'assignation, malgré son exhaustivité, ne trouve pas de solution pour le nœud courant.

Contrairement aux travaux présentés dans [6] et [7], où les transformations de graphes sont faites statiquement et *a priori*, l'approche proposée transforme le DFG à la volée, uniquement lorsque cela est véritablement nécessaire. Il existe trois transformations de graphe, illustrées dans la figure 5 :

1. le routage simple, utilisé par exemple si un nœud n'est pas ordonnançable (à cause des dépendances de données) ou quand il n'y a plus d'opérateur de calcul disponible dans l'architecture. Un nœud de mémorisation est alors intercalé pour retarder l'ordonnancement de l'opération courante comme dans la figure 5(c) ;
2. les scissions de nœud, utilisées si un nœud n'a pas d'assignation possible à cause de son nombre d'arcs sortants (*e.g.* un ou plusieurs enfants ne sont pas atteignables au travers du réseau d'interconnexion ou le nombre d'arcs sortants est supérieur aux possibilités de l'architecture). Deux transformations sont alors possibles pour diminuer ce nombre en ajoutant un autre nœud du même type possédant les mêmes prédécesseurs au cycle d'ordonnancement courant et en répartissant les arcs sortants le plus équitablement possible :
   2.1 la scission de nœud d'opération, qui est équivalente au recalcul dans [6] et qui est illustrée dans la figure 5(b) ;
   2.2 la scission de nœud de mémorisation, illustré par le passage de (c) à (d) dans la figure 5.

La scission de nœud est privilégiée si l'architecture possède suffisamment de ressources libres et si le nœud possède strictement plus d'un successeur. Dans les cas contraires, c'est le routage qui est utilisé. Seul l'ordonnancement en sens « inverse » ou « arrière » que nous proposons permet l'utilisation de l'ensemble de ces transformations. En effet, dans le cas d'un ordonnancement avant (utilisant par exemple un tri topologique simple), la seule transformation pertinente est le routage simple car on ne connaît pas les opérations filles qui seront réellement ordonnancées au cycle suivant. La répartition des arcs sortants serait alors purement arbitraire

et ne permettrait pas d'obtenir un véritable gain par rapport au routage simple.

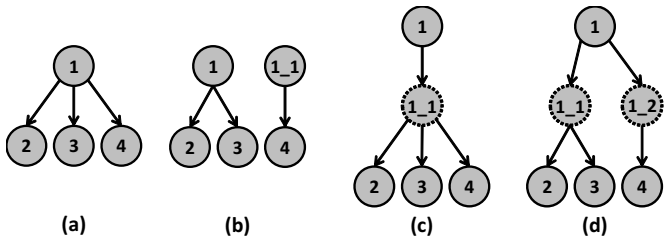

FIGURE 5 – Les différentes transformations de graphes. Les cercles représentent un nœud opération ainsi que son nœud de variable associé. (a) DFG initial, (b) scission d'opération sur le nœud d'opération 1 du DFG en (a), (c) routage simple sur le nœud 1 du DFG (a), (d) scission de mémorisation du nœud 1_1 du DFG en (c).

2.2.4. Élagage

L'étape d'élagage est introduite du fait de l'exhaustivité de l'algorithme de Levi. En effet, sans cela, le nombre de *mappings* partiels pourrait devenir si important qu'il rendrait cette méthode inefficace en termes de temps de calcul ou d'empreinte mémoire (en fonction des dépendances de données et des contraintes architecturales). Cette étape compare et supprime les *mappings* partiels redondants à la fin de chaque cycle d'ordonnancement. Un *mapping* partiel est dit redondant si il utilise exactement les mêmes opérateurs pour effectuer les mêmes opérations qu'un autre « *mapping* » partiel après ordonnancement et assignation de toutes les opérations possibles au cycle courant.

3. Expériences et résultats

Cette section décrit le protocole expérimental et présente les résultats obtenus.

3.1. Protocole expérimental

Notre méthode est intégralement automatisée et implémentée en Java en utilisant l'environnement de développement EMF (Eclipse Modeling Framework). Les CDFGs sont générés à partir des codes des applications en utilisant GCC-4.7.2. Neuf algorithmes issus du traitement du signal et des benchmarks de la « *High Level Synthesis* » (HLS) ont été utilisés pour les expériences, à savoir : une Transformée en Cosinus Discret (DCT-2D), un produit de matrice, une FFT, un calcul de la distance de Manhattan, un filtre à Moyenne Glissante Exponentielle (EMA), une Déconvolution à Fenêtre Glissante (MWD), un filtre trapézoïdal, un masque flou (unsharp mask) et un filtre passe-bas (DC-Filter). Les tests ont été réalisés sur un PC possédant un processeur Intel Xeon et 8 GByte de mémoire vive. Pour obtenir une large gamme de résultats, nous avons fait varier plusieurs contraintes architecturales : la taille du CGRA (3×3 et 4×4), le nombre de registres dans la file de registres (4 et 8) et le nombre de tuiles que les *mappings* ont le droit d'utiliser (1, 2, 3 et 4 tuiles) ; soit seize jeux de contraintes par code applicatif et par méthode.

L'approche proposée dans cet article est comparée à deux autres approches possibles de l'état de l'art. La première, « Méthode 1 », consiste à résoudre l'ordonnancement et l'assignation séparément comme dans la première étape de [9]. Elle utilise un « *list-scheduling* » parcourant les

nœuds du DFG triés par un tri topologique pour les ordonnancer et l'algorithme de Levi pour l'assignation. Le DFG ne peut être transformé que pendant la phase d'ordonnancement en utilisant du re-routage comme expliqué dans le paragraphe 2.2.3. La seconde, « Méthode 2 », réalise des transformations statiques du DFG, *i.e. a priori* pour réaliser l'ordonnancement et essaie de trouver des *mappings* en utilisant l'algorithme de Levi comme proposé dans [6] et [7]. Puisque les méthodes décrites dans [6] et [7] ont montré qu'elles donnent de meilleurs résultats que [11] et [3], nous ne nous comparons pas à ces méthodes dans cet article.
Quatre métriques sont considérées pour déterminer la qualité des différentes approches :

1. le taux de succès : définit comme étant le pourcentage de fois qu'une méthode trouve une solution quand au moins une des trois méthodes a trouvé une solution ;
2. le taux d'obtention de la meilleure latence : définit comme étant le pourcentage de fois que la méthode trouve la meilleure latence parmi les trois méthodes ;
3. la diversité : définit comme étant le nombre de *mappings* différents, c'est-à-dire la capacité d'une méthode à explorer largement l'espace des solutions. Deux *mappings* sont différents l'un de l'autre s'ils n'utilisent pas les mêmes tuiles et/ou s'ils utilisent le réseau d'interconnexion d'une manière différente ;
4. l'efficacité : définit comme étant le débit de *mappings* différents (nombre de *mappings* différents généré par seconde).

Ces deux dernières métriques donnent une indication particulièrement pertinente sur la capacité d'une méthode à transformer le graphe de l'application juste comme il faut. En effet, étant donné le nombre limité de ressources d'un CGRA, l'augmentation du nombre de nœuds du DFG implique une diminution du nombre de possibilités de les placer sur l'architecture. De ce fait, si la méthode conduit ajoute plus de nœuds au graphe que nécessaire, le nombre de solutions partielles que l'algorithme de Levi est capable d'obtenir diminue. De plus, le débit de *mappings* différents permet de savoir si la recherche est efficace ou si l'algorithme perd son temps à essayer des placements qui ne donneront pas d'autres solutions.

3.2. Résultats

Les figures 6 à 9 présentent, pour les différentes métriques, les résultats de compilation pour chacun des codes d'application considérés.
Dans la figure 6, on observe que résoudre les problèmes d'ordonnancement et d'assignation séparément, comme le fait la méthode 1, conduit à un faible taux de succès (environ 37%). Le fait d'effectuer des transformations de graphes statiques, comme la méthode 2, augmente le taux de succès (environ 62%), mais ne donne pas d'aussi bons résultats que notre méthode (environ 99%). Les faibles taux de succès des méthodes 1 et 2 sont dus en grande partie à l'aspect glouton de l'algorithme d'ordonnancement qui, lorsqu'il parcourt les nœuds par un tri topologique, engorge le réseau d'interconnexion du CGRA, empêchant toute assignation.
Concernant le taux d'obtention de la meilleure latence, comme la méthode proposée se base en partie sur une heuristique, il est normal qu'elle ne trouve pas toujours la meilleure latence (comme pour le filtre passe-bas). Cependant, comme illustré dans la figure 7, elle obtient plus de deux fois plus souvent la meilleure latence parmi les trois méthodes testées, soit environ 90% du temps. Quand elle ne trouve pas la meilleure latence, elle l'augmente en moyenne de 1,5 cycles, soit une augmentation d'environ 15% de ces latences.
La figure 8 montre que l'approche proposée dans ce papier permet d'obtenir généralement un nombre plus important de *mappings* différents que les méthodes 1 et 2 (respectivement 3,7 et 2,4 plus grand) et permet donc une meilleure exploration.

De plus, la figure 9 illustre le fait que l'exploration est plus efficace car le temps passé par *mapping* est plus faible comparativement aux autres méthodes (respectivement d'un facteur 2,6 et d'un facteur 2,2).

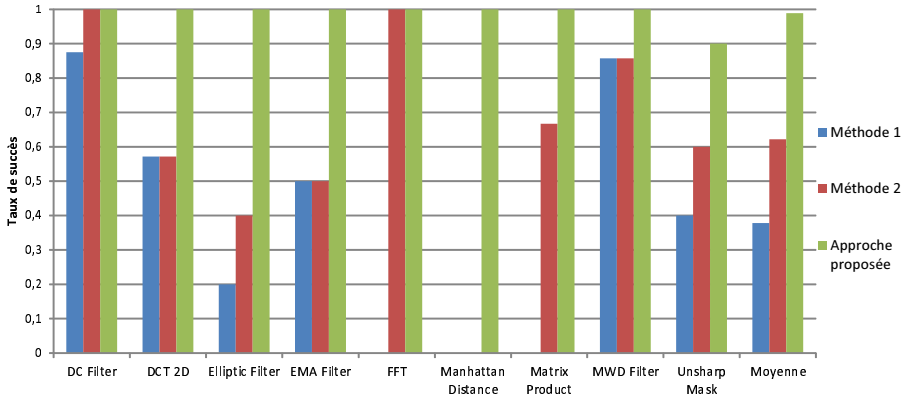

FIGURE 6 – Taux de succès.

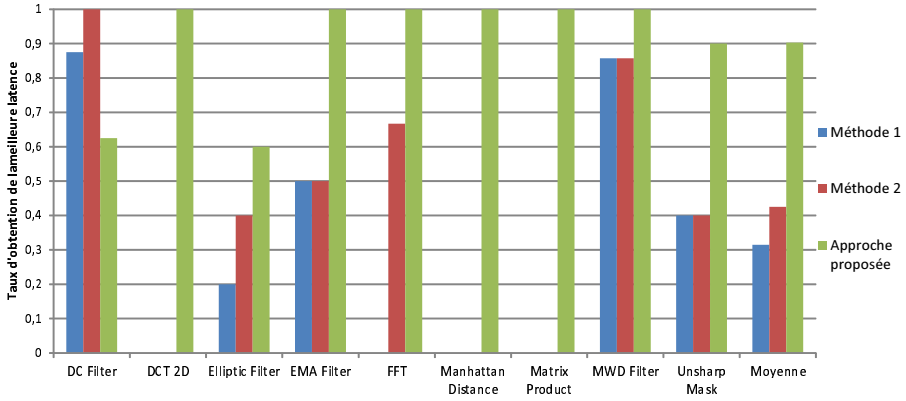

FIGURE 7 – Taux d'obtention de la meilleure latence.

4. Conclusion

Dans cet article, une méthode originale permettant de porter un code d'application écrit en C/C++ sur une architecture générique de type CGRA est présentée. Cette approche, qui repose sur un modèle CDFG de l'application et un modèle formel de l'architecture CGRA, explore l'espace de solution en résolvant simultanément les problèmes d'ordonnancement et d'assignation. Pour cela, les graphes des blocs de base sont parcourus dans un ordre topologique inverse. Ce sens de parcours permet de transformer le graphe à la volée, uniquement lorsque cela est nécessaire. Les expériences montrent que la méthode proposée possède le plus haut taux de succès, trouve 90% du temps la meilleure latence et explore plus efficacement l'espace des solutions par rapport aux méthodes de l'état de l'art.

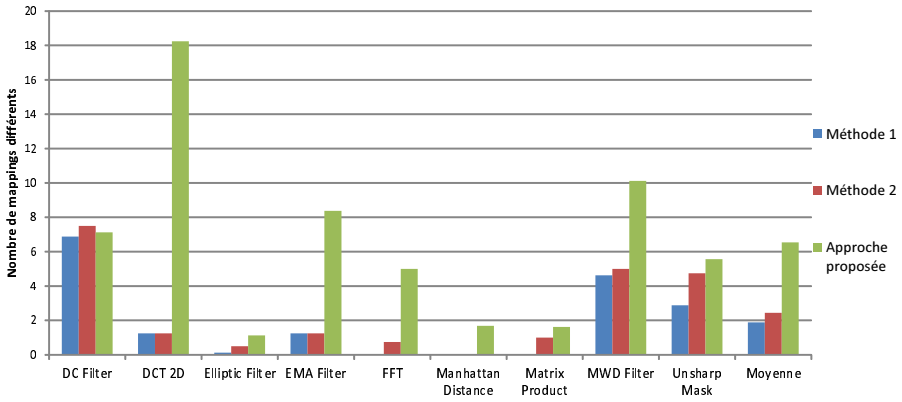

FIGURE 8 – Nombre moyen de *mappings* différents.

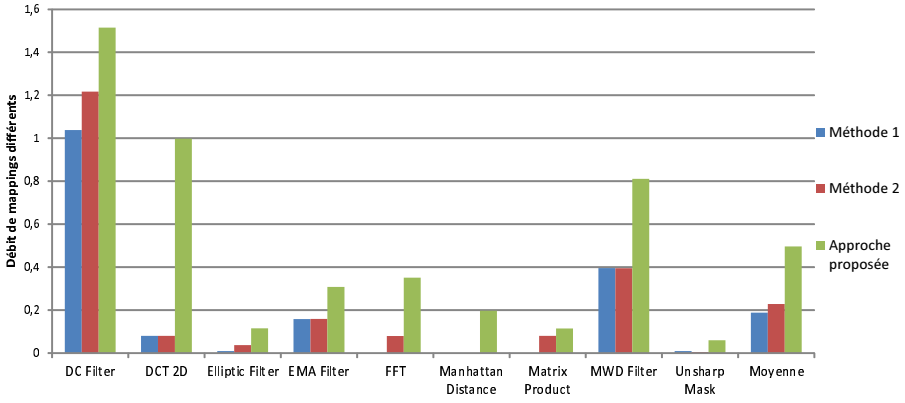

FIGURE 9 – Nombre moyen de *mappings* différents générés par seconde.

Bibliographie

1. Brenner (J. A.), van der Veen (J. C.), Fekete (S. P.), Oliveira Filho (J.) et Rosenstiel (W.). – Optimal Simultaneous Scheduling, Binding and Routing for Processor-like Reconfigurable Architectures. – In *Field Programmable Logic and Applications, International Conference on*, 2006.
2. Campi (F.), Deledda (A.), Mucci (C.), Lodi (A.), Pizzotti (M.), Cirrarelli (L.), Rolandi (P.), Vitkovski (A.) et Vanzolini (L.). – A dynamically adaptive DSP for heterogeneous reconfigurable platforms. – In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, 2007.
3. De Sutter (B.), Coene (P.), Vander Aa (T.) et Mei (B.). – Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. *ACM SIGPLAN Notices*, vol. 43, n7, juin 2008, p. 151.
4. Friedman (S.), Carroll (A.), Van Essen (B.), Ebeling (C.), Hauck (S.) et Ylvisaker (B.). – SPR : an architecture-adaptive CGRA mapping tool. – In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 191–200, 2009.
5. Gajski (D. D.) et Ramachandran (L.). – Introduction to high-level synthesis. *IEEE Design & Test of Computers*, vol. 11, n4, janvier 1994, pp. 44–54.
6. Hamzeh (M.), Shrivastava (A.) et Vrudhula (S.). – EPIMap : using epimorphism to map applications on CGRAs. – In *Design Automation Conference*, pp. 1284–1291, 2012.
7. Hamzeh (M.), Shrivastava (A.) et Vrudhula (S.). – REGIMap : register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). – In *Design Automation Conference*, 2013.
8. Lanuzza (M.), Perri (S.), Corsonello (P.) et Margala (M.). – A new reconfigurable coarse-grain architecture for multimedia applications. – In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, 2007.
9. Lee (G.), Choi (K.) et Dutt (N. D.). – Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *Computer-Aided Design of Intergrated Circuits and Systems, IEEE Transactions on*, vol. 30, n5, 2011, pp. 637–650.
10. Levi (G.). – A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, vol. 9, n4, décembre 1973, pp. 341–352.
11. Mei (B.), Vernalde (S.), Verkest (D.), De Man (H.) et Lauwereins (R.). – DRESC : A retargetable compiler for coarse-grained reconfigurable architectures. – In *Field-Programmable Technology, 2002. (FPT). IEEE International Conference on*, pp. 166–173, 2002.
12. Mei (B.), Vernalde (S.), Verkest (D.), De Man (H.) et Lauwereins (R.). – ADRES : An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. *In : Field Programmable Logic and Application*, éd. par Y. K. Cheung (P.) et Constantinides (G.), pp. 61–70. – Springer Berlin / Heidelberg, 2003.
13. Park (H.), Fan (K.), Mahlke (S. A.), Oh (T.), Kim (H.) et Kim (H.-S.). – Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. – In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.
14. Paulin (P. G.) et Knight (J. P.). – Force-directed scheduling for the behavioral synthesis of ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 8, n6, 1989, pp. 661 – 679.
15. Pillement (S.), Sentieys (O.) et David (R.). – DART : A Functional-Level Reconfigurable Architecture for High Energy Efficiency. *EURASIP Journal on Embedded Systems*, vol. 2008, 2008, pp. 1–13.
16. Raffin (E.), Wolinski (C.), Charot (F.), Kuchcinski (K.), Guyetant (S.), Chevobbe (S.) et Cas-

seau (E.). – Scheduling, binding and routing system for a run-time reconfigurable operator based multimedia architecture. – In *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 168–175, octobre 2010.

17. Singh (H.), Lee (M.-H.), Lu (G.), Kurdahi (F. J.), Bagherzadeh (N.) et Filho (E. M. C.). – MorphoSys : an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, vol. 49, n5, 2000, pp. 465–481.
18. Taylor (M. B.). – Is dark silicon useful ? : harnessing the four horsemen of the coming dark silicon apocalypse. – In *Design Automation Conference*, pp. 1131 – 1136, 2012.