Exploration de l'espace de conception pour la synthèse de communication pour la mise en œuvre d'accélérateurs matériels sur multi-cœurs hétérogènes

Jean-Michel Gorius[1]

[1]Univ Rennes, F-35000 Rennes, France

23 août 2018

Résumé

L'utilisation de SoCs (*System on a Chip*) intégrant un processeur multi-cœurs généraliste et un circuit FPGA (*Field-Programmable Gate Array*) dans les nœuds de calcul modernes permet de répartir les calculs entre processeur et accélérateur matériel. Ce partage des tâches conduit à des problèmes de cohérence des données, les processeurs généralistes utilisant plusieurs niveaux de cache auxquels l'accélérateur n'a qu'un accès restreint. Nous proposons une méthode permettant de générer automatiquement des instructions de gestion de cohérence de cache en utilisant un modèle polyédrique. Cette génération peut être intégrée à des outils de synthèse de haut niveau afin de permettre à l'utilisateur d'utiliser efficacement les interfaces de communication entre CPU et FPGA.

Mots-clés : Accélérateur matériel, Synthèse de haut niveau, Cache, Modèle polyédrique

Référence : Stage effectué du 22 mai 2018 au 20 juillet 2018 à l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, sous la direction de Steven Derrien, Univ Rennes, IRISA, CNRS, Inria, IRISA (UMR 6074), F-35000 Rennes, France et Tomofumi Yuki, Univ Rennes, IRISA, CNRS, Inria, IRISA (UMR 6074), F-35000 Rennes, France.

1 Introduction

Les besoins grandissants de puissance de calcul mais également les problématiques de consommation énergétique tendent à favoriser le développement de nouvelles plateformes matérielles. Celles-ci se distinguent des plateformes de calcul traditionnelles par l'intégration de composants hétérogènes au sein d'une même puce, favorisant ainsi la réduction de la consommation énergétique, au détriment d'une part de modularité mais sans perte de performances. On parle alors de SoC voire même de MPSoC (*Multiprocessor System on a Chip*) dans le cas d'utilisation de processeurs multi-cœurs. Certaines architectures mettent ainsi à disposition de l'utilisateur un ou plusieurs processeurs couplés à un FPGA voire à un GPU (*Graphics Processing Unit*), ce qui permet le développement d'accélérateurs matériels opérant de concert avec les autres éléments du système.

L'utilisation d'accélérateurs matériels à base de tels circuits FPGA est de plus en plus répandue dans les *datacenters* et les infrastructures de *cloud* [10]. En effet, ces accélérateurs associent un niveau de performances élevé à une consommation énergétique très faible en comparaison des processeurs généralistes ou des processeurs graphiques massivement parallèles, ce

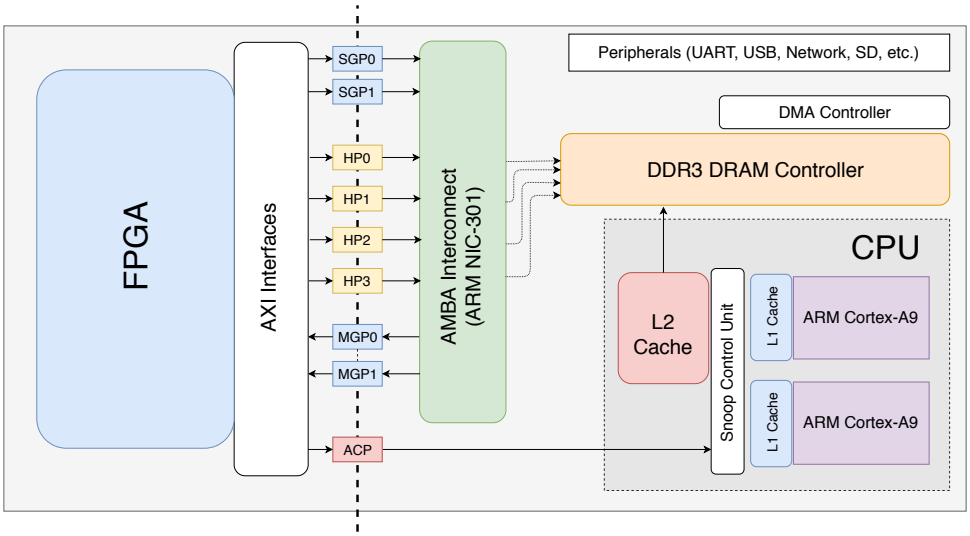


FIGURE 1 – Architecture ZYNQ-7000.

qui en fait des candidats de choix pour un déploiement à grande échelle dans les centres de calcul. La conception de ce type d'accélérateurs est cependant très difficile car elle relève du *design* de circuits et non pas de la programmation. Les outils de synthèse de haut niveau offrent une réponse à cette problématique en permettant la génération automatique d'accélérateurs matériels dédiés directement à partir de spécifications algorithmiques en C++.

L'intégration automatique de ces accélérateurs au sein d'un SoC reste cependant difficile de par la nature hétérogène des composants du système qui interagissent lors des calculs. Il existe des outils tels que Xilinx SDSoC permettant d'automatiser une grande partie de cette étape d'intégration mais ceux-ci ne fonctionnent que de manière exclusivement syntaxique. Ils ne permettent pas d'explorer de manière satisfaisante l'espace de conception, notamment lorsque l'exécution d'un même noyau de calcul est répartie entre l'accélérateur et un ou plusieurs processeurs. En effet, il se pose alors des problèmes de cohérence des données et de synchronisation impossible à gérer de manière statique par les outils de synthèse de haut niveau.

Dans la suite, nous présentons une méthode basée sur *les techniques de compilation polyédrique* permettant de générer automatiquement des instructions de gestion de cohérence de cache pour la synthèse d'accélérateurs matériels pour l'architecture Xilinx ZYNQ-7000. La section 2 donne le contexte général de l'étude en présentant les accélérateurs matériels (section 2.1), en introduisant le concept de synthèse de haut niveau (section 2.2) puis en illustrant le principe de la compilation polyédrique (section 2.3). La section 3 introduit notre méthode de génération de code pour la gestion de la cohérence de cache et discute les difficultés rencontrées lors de son élaboration, et la section 4 présente certains résultats expérimentaux. Enfin, la section 5 conclut ce rapport et donne quelques pistes pour une élaboration future de notre travail.

2 Contexte général

2.1 Accélérateurs matériels

Les FPGAs sont des circuits composés de blocs logiques simples souvent disposés en matrice et reliés par des routes configurables. Les blocs de base de ces circuits sont composés de tables de correspondance numérique auxquelles viennent parfois s'ajouter des composants plus complexes comme des additionneurs. Cette composition modulaire permet de reconfigurer le placement des

composants et le routage entre les blocs afin d'obtenir un circuit dédié à une tâche particulière, autrement dit un accélérateur matériel.

Les accélérateurs matériels tendent à se répandre, notamment dans les infrastructure de calcul des *datacenters* ou des *clouds*, grâce à leur faible consommation énergétique. Ces circuits offrent de très bonnes performances de calcul sur des tâches simples et répétitives, ce qui en fait des candidats de choix pour l'implantation de réseaux de neurones ou de plateformes de HPC (*High Performance Computing*).

Ces nouveaux domaines d'application viennent enrichir les possibilités de conception et ont mené au développement de MPSoCs, des plateformes complexes qui intègrent physiquement plusieurs composants aux côtés de multiples cœurs de processeur sur une même puce électronique. L'architecture ZYNQ-7000 développée par Xilinx est un exemple de MPSoC moderne intégrant des composants très hétérogènes, comme le montre la figure 1.

Cette architecture met à disposition un processeur deux cœurs ARM Cortex-A9, un FPGA ainsi qu'un grand nombre d'interfaces de communication. La figure 1 illustre bien la complexité des interfaces présentes dans un MPSoC, celles-ci variant du bus d'entrée/sortie vidéo à des chemins de données haute performance internes. Les interactions entre les divers composants du système rendent la programmation d'applications visant ces plateformes très ardue. En effet, en plus du développement logiciel ciblant les processeurs généralistes, il faut s'atteler à la conception du circuit à implanter sur le FPGA et de toutes ses interfaces de communication. Afin de faciliter la programmation de ces MPSoCs, de nouveaux outils ont donc vu le jour : *les environnements de synthèse de haut niveau.*

2.2 Synthèse de haut niveau

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_or_top is
    Port ( INA1 : in  STD_LOGIC;
           INA2 : in  STD_LOGIC;
           OA   : out STD_LOGIC;
end and_or_top;

architecture Behavioral of and_or_top is
begin
    OA <= INA1 and INA2;
end Behavioral;
```

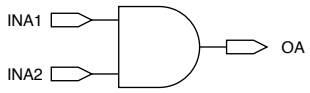


FIGURE 2 – Conception d'une porte AND en VHDL.

La synthèse de haut niveau (*High Level Synthesis* ou HLS) est une méthode de développement qui permet de produire la description, au niveau circuit logique, d'un accélérateur matériel à partir de spécifications en C/C++. Ces outils permettent d'augmenter le niveau d'abstraction auquel le concepteur travaille, ce qui augmente grandement sa productivité. En effet, avant le développement des environnements de HLS, chaque circuit devait être décrit dans un langage spécialisé tel que le VHDL (*VHSIC*[1] *Harware Description Language*). Ces langages sont très proches du matériel et obligent l'utilisateur à définir chaque signal et/ou chaque composant de base manuellement. La figure 2 donne un exemple de circuit simple conçu en VHDL. Les accélérateurs matériels pouvant contenir plusieurs dizaines de milliers de composants, leur description dans un tel langage est très laborieuse et source d'erreurs. Afin d'illustrer le principe

1. VHSIC : *Very High Speed Integrated Circuit*

```
int X[512], Y[512];
int tmp, res = 0;
for(int i = 0; i < 512; ++i) {
    tmp = X[i]*Y[i];
    res += tmp;
}
```

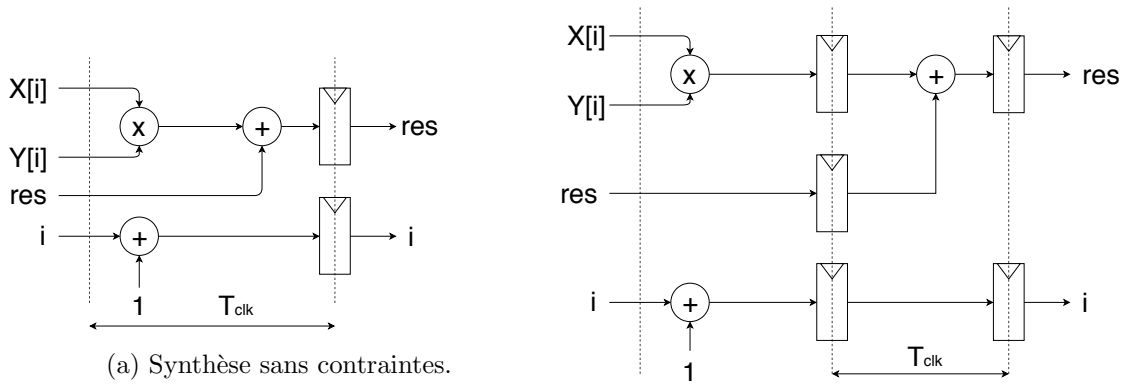
FIGURE 3 – Produit scalaire de deux vecteurs.

et les avantages de la synthèse de haut niveau, nous utiliserons un code réalisant le produit scalaire de deux vecteur (voir figure 3).

Les outils de HLS analysent une spécification algorithmique écrite dans un langage de haut niveau comme le C++ et produisent une description du matériel en VHDL sous forme de machines à états pilotant des chemins de données. Le code de la figure 3 peut être traité de différentes manières en fonction de contraintes spécifiées par l'utilisateur, comme la nature des composants ou la fréquence d'horloge souhaitée. Le comportement par défaut des outils de synthèse de haut niveau revient à exécuter une itération de boucle par cycle d'horloge. Dans ce cas, le chemin de données synthétisé est semblable à celui de la figure 4a. La fréquence de calcul est alors dépendante du temps d'exécution de chacun des opérateurs élémentaires (additionneurs et multiplieur), qui détermine la durée totale d'un cycle T_{clk}. L'utilisateur peut également choisir d'imposer une fréquence d'horloge approchée, ce qui contraindra le type de circuit généré par le compilateur HLS. Dans le cas d'une fréquence souhaitée plus élevée, le circuit de la figure 4b peut être généré. L'augmentation du nombre d'étages de registres permet une fréquence d'exécution plus élevée, bien que la durée totale du calcul puisse être plus longue. Il faut souvent trouver un compromis surface/performances adapté aux applications ciblées.

La génération d'une description matérielle sous contraintes nécessite une connaissance approfondie de la plateforme ciblée afin de gérer au mieux le choix des composants, leur placement et leur routage. Les outils de HLS permettent d'abstraire ces difficultés en laissant le programmeur se concentrer sur les opérations à effectuer sur l'accélérateur plutôt que sur les détails d'implantation matérielle.

Malgré cette abstraction, le concepteur de circuit peut toujours interagir finement au niveau du matériel afin de spécifier des contraintes particulières. En effet, les outils ne font par exemple pas toujours le choix de réutiliser les composants déjà placés pour certaines opérations.



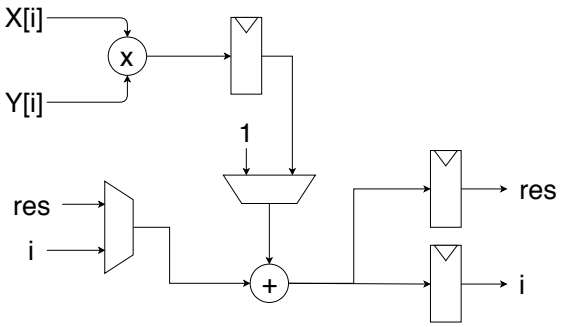
(a) Synthèse sans contraintes.

(b) Contrainte de fréquence d'horloge.

FIGURE 4 – Circuits synthétisés sous diverses contraintes.

```
1  int X[512], Y[512];
2  int tmp, res = 0;
3  #pragma HLS mult=1, adder=1
4  for(int i = 0; i < 512; ++i) {
5      tmp = X[i] * Y[i];
6      res += tmp;
7  }
```

(a) Indication au compilateur sur les composants à utiliser.



(b) Circuit synthétisé utilisant des multiplexeurs.

FIGURE 5 – Utilisation de *pragmas* pour guider la synthèse.

Dans le cas du calcul du produit scalaire, l'incrémentation de la variable de boucle `i` ainsi que l'accumulation dans la variable `res` de la figure 4a utilisent toutes les deux un additionneur. L'utilisateur peut choisir d'indiquer à l'outil de synthèse qu'il ne souhaite utiliser qu'un seul additioneur ainsi qu'un multiplieur à l'aide de directives de préprocesseur. Ces directives se présentent sous la forme de *pragmas* qui indiquent au compilateur HLS un comportement à adopter pour une section de code donnée. Grâce au code de la figure 5a, le programmeur peut amener les outils de HLS à produire un chemin de données semblable à celui présenté figure 5b. D'autres directives permettent d'agir sur le type d'accélérateur généré par le compilateur HLS en modifiant par exemple l'organisation de la mémoire interne du FPGA ou encore les ports utilisés pour accéder à la mémoire externe.

Les exemples précédents ont montré les avantages des outils de HLS dans le cas de la conception de circuits. Ces circuits sont souvent implantés sur FPGA au sein de systèmes plus complexes que sont les SoCs et MPSoCs. Les interactions entre les composantes d'un même SoC sont généralement complexes et nécessitent de synthétiser une logique de communication avancée en plus de la logique de calcul. Afin de simplifier l'intégration de circuits synthétisés dans les SoCs et MPSoCs, des environnements de développement spécialisés ont été mis au point. Ces environnements ajoutent un nouveau niveau d'abstraction en générant de façon automatique les interfaces de communication et de synchronisation entre les éléments d'un même système. Dans la suite, nous nous intéresserons particulièrement à l'environnement SDSoC développé par Xilinx.

SDSoC permet de simplifier le flot de conception d'un accélérateur matériel en prenant en charge la majeure partie des étapes de génération de code et d'exécutables. Cet outil permet de programmer des applications ciblant les architectures Xilinx comme la plateforme ZYNQ-7000 présentée en figure 1. La figure 6 illustre les différentes étapes de transformation réalisées par SDSoC à partir d'une base de code en C++ comportant le code à exécuter sur le processeur ainsi que les spécifications algorithmiques décrivant le comportement de l'accélérateur. Ces deux parties sont séparées par l'outil, qui génère une description VHDL de l'accélérateur ainsi qu'un exécutable ELF. La description VHDL est ensuite transférée sous forme binaire au FPGA tandis que l'exécutable est couplé à un chargeur d'amorçage et un noyau Linux en une image transférée sur une carte SD.

En plus de réaliser une grande partie de la génération du code, SDSoC offre de nouvelles directives de préprocesseur sous la forme de *pragmas* qui permettent à l'utilisateur de paramétrer le type d'interface et les interactions autorisées pour les communications entre la logique programmable du FPGA et le reste du système, notamment en terme d'accès à la mémoire. Dans le cas de l'architecture ZYNQ, ces accès peuvent être réalisés par le biais de deux types de

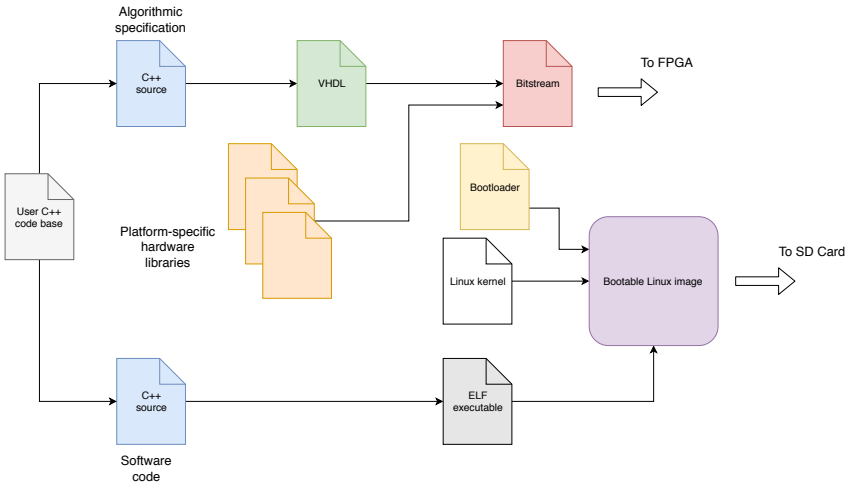


FIGURE 6 – Schéma du flot de conception SDSoC d'un accélérateur matériel et de l'application qui en fait usage.

ports : les ports haute performance *HP0*, *HP1*, *HP2* et *HP3* ainsi que le port *ACP* (*Accelerator Coherency Port*). Comme l'illustre la figure 1, les quatre premiers permettent à un accélérateur implanté sur le FPGA du MPSoC d'accéder directement au contenu de la mémoire principale. Le port ACP permet quant à lui d'accéder à des données en mémoire en garantissant la cohérence avec le processeur, c'est-à-dire en passant par le cache de niveau 2 (L2) et en y chargeant les données requises si celles-ci n'y sont pas déjà présentes. Le port ACP permet d'assurer la synchronisation indispensable entre les données traitées par le FPGA et celles traitées par le CPU lors de tâches partagées.

L'usage de ce seul port de cohérence reste cependant assez restrictif car il ne permet pas d'utiliser pleinement les capacités d'entrées/sorties du FPGA dans le cas de l'exécution d'un même noyau de calcul sur l'accélérateur matériel et les processeurs. En effet, afin d'éviter les problèmes de cohérence, l'utilisateur ne peut faire usage des ports HP sans risques.

Afin d'illustrer cette situation, considérons le cas du calcul de la multiplication de deux matrices A et B par blocs. La figure 7 montre une répartition possible des calculs : l'accélérateur matériel est chargé de calculer $A_{0,0}B_{0,0}$ tandis que le CPU calcule $A_{0,1}B_{1,0}$. Tous deux accumulent le résultat dans une copie locale de $C_{0,0}$. Un problème se pose lors de la réécriture de $C_{0,0}$ en mémoire principale. En effet, si le FPGA n'utilise pas le port ACP pour réécrire sa copie en mémoire, celle-ci va se retrouver dans la DRAM et sera écrasée par l'écriture de la copie présente dans le cache L2.

Afin de pallier ce problème, il suffit d'insérer des évictions de l'intégralité du cache à chaque transfert de données depuis ou vers le FPGA. Cette méthode est cependant très grossière et introduit une forte pénalité lors de l'exécution. En effet, la majeure partie des données présentes dans le cache n'a pas besoin d'être renvoyée en mémoire principale à chaque appel. Afin de pallier cette difficulté, il est nécessaire de mettre en place un mécanisme de gestion plus fine de la cohérence des données. Le respect de la cohérence des données est un aspect central de la conception d'un accélérateur travaillant de concert avec un ou plusieurs processeurs.

Le problème sous-jacent peut être formulé de la manière suivante : comment répartir les accès mémoire entre les différents ports de communication mis à disposition par la logique programmable du FPGA pour maximiser le débit de transfert tout en s'assurant que la cohérences des données est préservée, dans le cas d'un noyau de calcul s'exécutant à la fois sur CPU et accélérateur ?

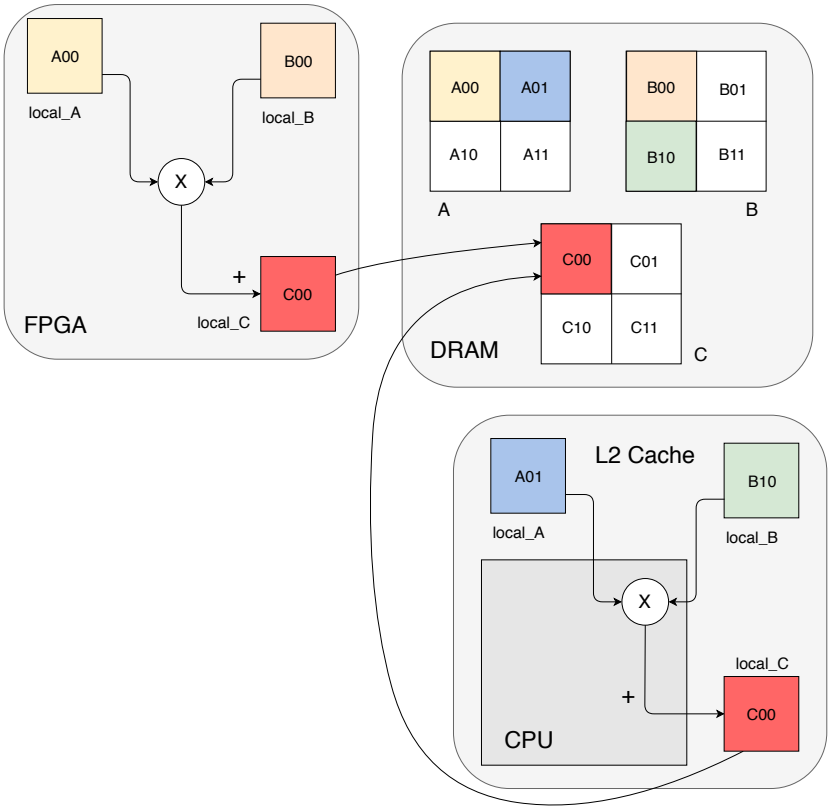


FIGURE 7 – Conflit lors de l'écriture mémoire dans le cas d'un noyau de calcul partagé.

Notre contribution apporte une solution à ce problème en prenant appui sur la théorie des modèles polyédriques et notamment sur l'une de ces applications : la compilation polyédrique. Celle-ci permet d'analyser finement les dépendances de données dans une sous-classe particulière de programmes et d'en déduire les points du code où il est nécessaire d'insérer des instructions de synchronisation des données entre cache CPU et mémoire interne au FPGA. Elle peut notamment être utilisée dans le cadre de l'optimisation de matériel dans le cadre de la synthèse de haut niveau [3]. Dans la section suivante, nous présentons le principe de la compilation polyédrique appliquée à la gestion logicielle du cache processeur à l'aide d'un exemple et introduisons quelques notations utiles.

2.3 Compilation polyédrique

La compilation polyédrique se base sur des modélisations de programmes contenant des boucles imbriquées par des *polyèdres* pour fournir des outils de manipulation de code très précis. Ces modèles permettent par exemple de représenter les dépendances entre les données traitées dans les itérations successives d'un ensemble de boucles. On représente ces dernières par des polyèdres paramétriques ou des *formules de Presburger* qui permettent d'exploiter des propriétés géométriques et combinatoires afin d'optimiser les calculs selon différents critères. Ce modèle de programmes dit *modèle polyédrique* permet par exemple de paralléliser des programmes de manière automatisée [2]. Il existe de nombreux outils permettant de manipuler des représentations polyédriques de programmes dont notamment ISL (*Integer Set Library* [8]) et Barvinok [9], une bibliothèque développée par l'Inria.

Nous nous intéressons à la compilation polyédrique dans le cadre de la génération automatique d'instructions de gestion du cache processeur, dans le but de synchroniser les données

traitées par le CPU et un accélérateur matériel implanté sur FPGA, qui calculent de concert. Ce problème est difficile dans le cas général car l'analyse des dépendances et des accès mémoires se ramènent à un problème indécidable, celui de l'*aliasing* [5]. Dans le cas de boucles dont les bornes et les adresses des zones mémoires accédées peuvent être exprimées linéairement en fonction des variables d'itération, ce problème peut cependant être traité grâce aux modèles polyédriques.

La suite de cette section reprend et adapte une partie du travail de Tavarageri et al. [6] sur la gestion logicielle automatique de cohérence pour des caches multiprocesseur. Après avoir introduit quelques notations, nous illustrerons la représentation des boucles et des dépendances dans le cadre du modèle polyédrique. Nous appuierons notre présentation sur le code donné en figure 8.

```
1 for(int t1 = 0; t1 < tsteps - 1; ++t1)
2     for(int t2 = 0; t2 <= n - 1; ++t2)
3         for(int t3 = 0; t3 <= n - 1; ++t3)
4         S:  A[t2][t3] = 1 + A[t2+1][t3];
```

FIGURE 8 – Boucles imbriquées avec dépendances de données.

2.3.1 Notations

Les notations introduites ici sont classiques de l'étude de la compilation polyédrique et sont reprises dans [6]. Elles sont semblables à celles utilisées dans les environnements interactifs de manipulation de polyèdres comme ISCC [9].

Ensembles : Un ensemble E est défini par $E = \{S[x_1, \ldots, x_m] : c_1 \wedge \ldots \wedge c_p\}$ où S est un identifiant, $[x_1, \ldots, x_m]$ un m-uplet de variables et les c_i des contraintes sur ces variables. Ces ensembles sont utilisés afin de représenter l'espace d'itération d'un groupe de boucles. Dans le cadre de l'exemple de la figure 8, l'espace d'itérations est donné par

$$I^S = \{S[t_1, t_2, t_3] : 0 \leqslant t_1 \leqslant \text{tsteps} - 1 \wedge 0 \leqslant t_2 \leqslant n - 1 \wedge 1 \leqslant t_3 \leqslant n - 1\}.$$

Relations : Une relation R est définie par $R = \{S1[x_1, \ldots, x_m] \mapsto S2[y_1, \ldots, y_n] : c_1 \wedge \ldots \wedge c_p\}$ avec $S1$ et $S2$ identifiants, $[x_1, \ldots, x_m]$ m-uplet de variables d'entrées, $[y_1, \ldots, y_n]$ n-uplet de variables de sortie et les c_i des contraintes sur ces variables. On utilise des relations pour représenter les accès à des tableaux dans des boucles. Les deux accès au tableau A sont par exemple représentés par les relations d'écriture et de lecture suivantes :

$$R_w^S = \{S[t_1, t_2, t_3] \mapsto A[t'_2, t'_3] : t'_2 = t_2 \wedge t'_3 = t_3\}$$
$$R_r^S = \{S[t_1, t_2, t_3] \mapsto A[t'_2, t'_3] : t'_2 = t_2 + 1 \wedge t'_3 = t_3\}.$$

Opérateur d'application : L'*opérateur d'application* d'une relation R à un ensemble E, notée $E' = R(E)$, est définie par $x \in E' \iff \exists y \in E \quad y \mapsto x \in R$. Cet opérateur permet de déterminer la plage de données accédée dans un tableau à partir des relations d'écriture et de lecture et de l'espace d'itérations. En effet, il suffit d'appliquer les relations R_w^S et R_r^S à l'ensemble I^S.

Inverse d'une relation : L'opérateur d'inversion d'une relation R produit une relation R^{-1} telle que les contraintes de R et R^{-1} soient identiques mais que les uplets d'entrée et de sortie soient inversés, c'est-à-dire que

$$\forall x, y \quad x \mapsto y \in R \iff y \mapsto x \in R^{-1}.$$

La section suivante donne un aperçu des possibilités de calcul offertes par la représentation polyédrique des boucles en illustrant le calcul des dépendances de données sur l'exemple de la figure 8.

2.3.2 Représentation des dépendances de données

La compilation polyédrique se sert d'outils de manipulation de polyèdres afin d'effectuer de l'analyse de flot de données. Grâce à ces outils, il est par exemple possible de déterminer l'ensemble des dépendances *Read after Write* (RaW) et *Write after Write* (WaW) dans un ensemble de boucles imbriquées. Ces dépendances sont représentées par des relations de leurs itérations source à leurs itérations cibles.

L'ensemble des dépendances RaW déterminées par Barvinok sur le code de notre exemple est donné par

$$\begin{aligned}\mathcal{D}_{\text{flow}} = \{S[t_1, t_2, t_3] \mapsto S[t'_1 = t_1 + 1, t'_2 = t_2 - 1, t'_3 = t_3] : \\ 0 \leqslant t1 \leqslant \text{tsteps} - 2 \wedge 1 \leqslant t2 \leqslant n - 1 \wedge 1 \leqslant t3 \leqslant n - 1\}.\end{aligned}$$

Cette relation signifie qu'une écriture à un emplacement de A donné à l'itération d'indices (t_1, t_2, t_3) est suivie d'une lecture au même emplacement à l'itération d'indices (t_1+1, t_2-1, t_3).

Dans le cadre de l'exécution d'un noyau de calcul en simultané sur CPU et FPGA, ce type de calcul de dépendances permet de déterminer les zones mémoires présentes dans le cache à mettre à jour avant la prochaine itération afin de garantir la cohérence des accès. La section suivante présente un algorithme s'appuyant sur les possibilités offertes par la compilation polyédrique pour générer automatiquement des instructions de gestion de la cohérence de cache dans le cadre d'une exécution concurrente d'un noyau de calcul sur CPU et FPGA.

3 Contribution

La conception et l'implantation de notre algorithme de génération de code de cohérence de cache se sont déroulées en plusieurs étapes discutées ci-dessous. L'objectif de ce travail était de mettre en place un ensemble de *micro-benchmarks* visant à mesurer la bande passante des interfaces de communication d'un SoC/MPSoC ainsi que l'impact des calculs sur le cache CPU, dans le cadre de l'exécution d'un noyau de calcul à la fois sur le processeur et sur l'accélérateur. Nos expérimentations se sont concentrées sur une application simple, la multiplication de matrices par blocs, afin de mettre en évidence les caractéristiques du système ciblé sans bruiter les mesures par la complexité des calculs. L'ensemble de notre travail s'appuie sur l'architecture Xilinx ZYNQ-7000 (figure 1).

La section 3.1 présente l'environnement Xilinx SDSoC qu'il nous a fallu prendre en main afin de pouvoir travailler avec des SoCs et MPSoCs Xilinx, la section 3.2 illustre les outils de mesure de performances mis en œuvre dans le cadre de nos *micro-benchmarks*, puis la section 3.3 explique les difficultés rencontrées pour la gestion fine du cache pour l'architecture Xilinx ZYNQ-7000. La section 3.4 présente l'algorithme que nous avons développé pour répondre au problème posé en fin de section 2.2. La section 4 donne ensuite quelques résultats expérimentaux.

```cpp
void matmul_block(/* ... */) {
    const int M = BLOCK_SIZE;
    static int block0[M * M] = { 0 };
    static int block1[M * M] = { 0 };
    static int blockOut[M * M] = { 0 };
    /* Sélection des ports d'entrée et copie des données dans la mémoire locale. */
    for(int i = 0; i < M; ++i) {
        for(int j = 0; j < M; ++j) {
            int sum = 0;
            for(int k = 0; k < M; ++k)
#pragma HLS pipeline
                sum += block0[i * M + k] * block1[k * M + j];
            blockOut[i * M + j] += sum;
        }
    }
    /* Sélection du port de sortie et copie des données en mémoire externe. */
}
```

FIGURE 9 – Code C++ de haut niveau.

3.1 Prise en main des outils

Notre travail visant à déployer notre algorithme de génération de code sur une architecture Xilinx ZYNQ-7000, nous avons utilisé l'environnement de synthèse de haut niveau Xilinx SDSoC évoqué en introduction. Cet outil fourni un grand nombre d'utilitaires qui permettent de réduire considérablement le temps de conception d'un circuit ciblant une plateforme MPSoC en automatisant la génération du code VHDL et de l'ensemble de la logique de contrôle des interfaces de communication [7]. Le code de test que nous avons utilisé est présenté en figure 9. Il réalise la multiplication de deux blocs de deux matrices données en entrée sous la forme de pointeurs et accumule le résultat dans un bloc de sortie qui est ensuite réécrit en mémoire externe.

Cet exemple montre bien la complexité des opérations réalisées par SDSoC qui, à partir de spécifications algorithmiques simples en C++, est capable de générer des composants aux interactions très complexes. De plus, l'utilisateur peut s'abstraire de la plateforme ciblée, l'ensemble des opérations d'implantation matérielles étant automatiquement mis en œuvre par l'outil. Afin de réaliser nos premiers essais, nous avons choisi de travailler sur une carte d'expérimentation Zybo développée par Digilent. Cette carte de petite dimension est basée sur l'architecture ZYNQ-7000 et fournit deux cœurs ARM Cortex A9 couplés à un FPGA et à l'ensemble des interfaces présentées en figure 1.

Avant de mettre en place un algorithme de génération de code, il nous a fallu déterminer un moyen fiable de mesurer la bande passante et la latence des communications entre l'accélérateur et la mémoire externe, ainsi que les interactions des accès mémoire avec le cache L2 qui interagit avec le port ACP.

3.2 Mesures de performances

En plus de fournir tous les outils nécessaires à la synthèse d'accélérateurs matériels à partir de spécifications de haut niveau, SDSoC intègre des outils de simulation et de mesure des performances des composants générés. Nous nous sommes particulièrement intéressé à l'un de ces outils : le moniteur de performances AXI.

Ce moniteur de performances est généré sous forme d'un composant générique inséré au-

```
void Xil_L2CacheEnable(void);
void Xil_L2CacheDisable(void);
void Xil_L2CacheInvalidate(void);
void Xil_L2CacheInvalidateLine(unsigned int adr);
void Xil_L2CacheFlushLine(unsigned int adr);
void Xil_L2CacheStoreLine(unsigned int adr)
```

FIGURE 10 – Prototypes d'une partie des fonctions de gestion du cache contenues dans les en-têtes systèmes Xilinx.

tomatiquement dans le circuit synthétisé et relié à l'ensemble des interfaces de communication utilisant le protocole AXI (*Advanced eXtensible Interface*), telles que les ports HP et le port ACP. Il permet d'observer en temps réel l'activité des différents ports de communication mais également l'état du CPU (nombre d'opérations effectuées, charge de travail) et le nombre de *cache miss*. Grâce à ces informations, nous avons pu effectuer de premières mesures. Cependant, nous avons constaté que les données fournies par ce composant n'étaient pas assez précises. En effet, les interactions du moniteur de performances avec les interfaces de communication devenaient visibles dans les relevés lors d'une charge de transfert de données élevée entre la mémoire externe et l'accélérateur matériel. Après une exploration approfondie de la base de code et des bibliothèques fournies par Xilinx, nous avons découvert une méthode non documentée permettant de mesurer au cycle près le temps d'exécution d'une opération, ainsi qu'un moyen de mesurer l'activité du cache de niveau 2 grâce aux compteurs de performance intégrés au contrôleur de cache.

3.3 Gestion fine du cache

Afin de pouvoir mettre en place la génération automatique d'instructions de cohérence de cache, il nous fallait trouver un moyen de gérer très finement le cache L2. Le guide du programmeur ARM Cortex-A pour ARMv7 [1] contient une section dédiée aux opérations de maintenance de cache en assembleur. Le problème de l'accès aux ports de communication du contrôleur de cache s'est alors posé. Il s'avère qu'une autre partie peu documentée du code des bibliothèques Xilinx réalise l'ensemble des opérations dont nous avions besoin. En effet, celle-ci contient des fonctions de contrôle des invalidations et des évictions de zones précises du cache (voir figure 10). Nous avons réalisé plusieurs mesures et comparaisons de durées d'exécution afin de déterminer si chacune de ces fonctions réalisait bien l'opération attendue, ce qui nous a permis de valider le comportement de chacune d'entre elles.

Cette interface de contrôle du cache couplée aux outils de mesure de performances décrits dans la section précédente nous a permis de mettre en place un environnement de *micro-benchmarking* complet, grâce auquel nous avons pu obtenir les résultats présentés dans la section 4.1.

3.4 Génération automatique d'instructions de cohérence de cache

Une fois l'environnement de mesures et de simulation en place, nous nous sommes attelés au développement de notre algorithme de génération automatique d'instructions de cohérence de cache pour le cas de l'exécution d'un noyau de calcul partagé entre processeur et accélérateur matériel. Cette algorithme s'appuie sur les techniques de compilation polyédrique ainsi que sur les opérations mises à jour en section 3.3 afin de répondre au problème formulé à la fin de la section 2.2.

On considère un calcul $\mathcal{C}$. Une exécution concurrente de $\mathcal{C}$ sur CPU et FPGA nécessite de décomposer celui-ci en p sous-calculs $(\mathcal{C}_i)_{1 \leqslant i \leqslant p}$. Soit `computed` un tableau de booléens de taille p tel que `computed[i]` est vrai si et seulement si le bloc $\mathcal{C}_i$ a été calculé. À l'aide des techniques présentées en section 2.3, il est possible de déterminer l'ensemble des dépendances RaW entre les différents $\mathcal{C}_i$ pour construire un graphe de tâches [4]. On génère ensuite un ordonnancement des calculs respectant les contraintes imposées par `dep` en parcourant l'ensemble des $\mathcal{C}_i$. On peut alors répartir les calculs entre le processeur et l'accélérateur matériel en tenant compte de certaines contraintes données par l'utilisateur. En appliquant une technique similaire à celle développée dans [6] et évoquée en section 2.3, il suffit alors de calculer l'ensemble des zones du cache à invalider et/ou à réécrire en mémoire pour chaque bloc exécuté par le FPGA. De cette manière, les calculs effectués par l'accélérateur se comportent comme s'ils étaient soumis à une forme de cohérence de cache. Afin de restreindre le nombre d'opérations à effectuer sur le cache, on peut exclure de la gestion logicielle de la cohérence l'ensemble des données qui transitent vers ou depuis le FPGA par le port ACP, celui-ci étant déjà soumis aux mécanismes matériels de cohérence.

Le pseudo-code de notre algorithme est donné en figure 11. La fonction `depscomputed`(c) renvoie vrai si toutes les dépendances de c ont été traitées et la fonction `dispatch` répartit les calculs entre CPU et accélérateur en fonction de contraintes spécifiées par l'utilisateur. Les fonctions `computeInvalidateSet` et `computeWritebackSet` utilisent les algorithmes développés par Tavarageri et al. [6] pour le calcul des blocs mémoires à invalider et à réécrire en mémoire principale pour un calcul donné. Après exécution de cet algorithme, il suffit d'appliquer une méthode d'insertion d'instructions de cohérence similaire à celle donnée dans [6].

Entrée : Calcul $\mathcal{C}$
Sortie : Répartition des calculs entre CPU et FPGA, ensemble des blocs à invalider après calcul sur FPGA

```
1  Décomposer C en p sous-calculs (C_i)_{1≤i≤p};
2  Soit computed[1..p] tableau donnant l'état de chacun des C_i;
3  Soit dep[1..p] graphe de dépendances calculé à l'aide de D_flow (section 2.3);
4  nToDo ← p;
5  while nToDo > 0 do
6  |  for i ← 1 to p do
7  |  |  if depscomputed(i) then
8  |  |  |  Schedule ← i :: Schedule;
9  |  |  |  nToDo ← nToDo − 1;
10 |  |  end
11 |  end
12 end
13 toComputeCPU, toComputeFPGA ← dispatch(Schedule);
14 for c in toComputeFPGA do
15 |  Voir [6];
16 |  invalidateSet ← computeInvalidateSet(c);
17 |  writebackSet ← computeWritebackSet(c);
18 end
19 return Schedule, invalidateSet, writebackSet;
```

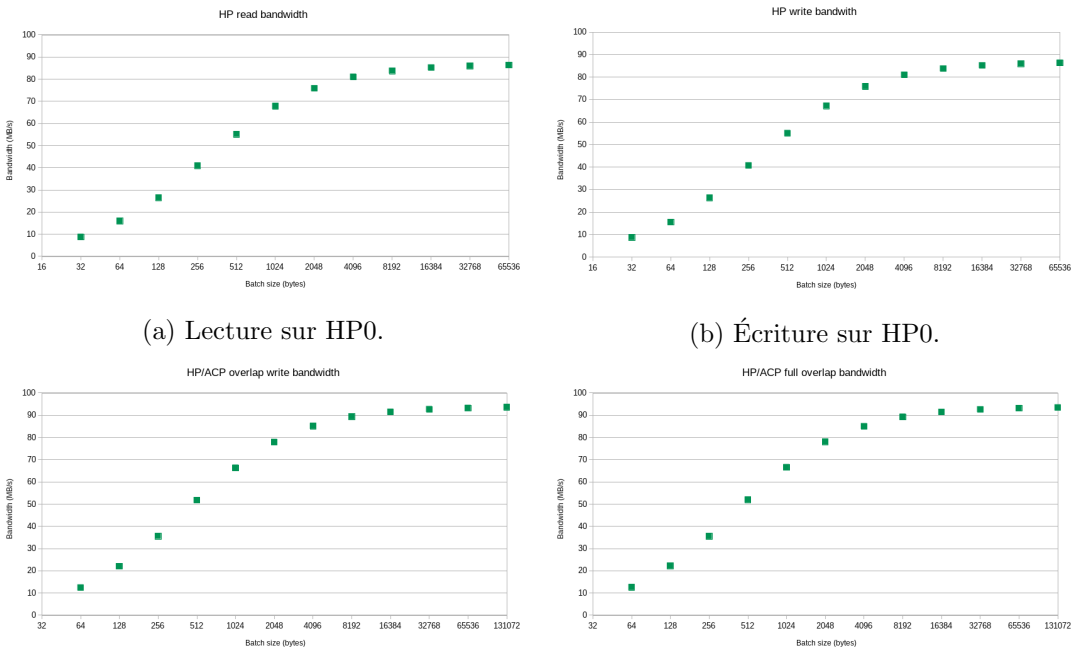
FIGURE 11 – Algorithme de génération d'instructions de cohérence de cache pour une exécution partagée CPU/FPGA.

Nous n'avons pas pu implanter cette algorithme sur une plateforme réelle en raison de certaines contraintes et limites imposées par les outils de synthèse et qui seront développées dans la section suivante.

4 Résultats expérimentaux

Afin de pouvoir définir les contraintes destinées à la fonction `dispatch` introduite dans la section 3.4 et implanté notre algorithme sur un MPSoC, nous avons mesuré la bande passante des interfaces de communication régies par le protocole AXI, à savoir les ports HP0-3 ainsi que le port ACP. Parallèlement à cela, nous avons également tenté de mesurer l'impact de notre méthode sur les durées d'accès à la mémoire. Ces dernières expérimentations nous ont permis de découvrir une limite majeure des outils de HLS dans le cadre de l'allocation mémoire. Les sections suivante présentent tout d'abord l'ensemble des *micro-benchmarks* que nous avons mis en place pour mesurer le débit de transfert entre le CPU et le FPGA (section 4.1). Nous discutons ensuite des limites des outils de synthèse, obstacle principal à la mise en application de notre algorithme sur une plateforme réelle (section 4.2).

4.1 Micro-benchmarks



(a) Lecture sur HP0.

(b) Écriture sur HP0.

(c) Recouvrement partiel entre HP0 et ACP.

(d) Recouvrement total entre HP0 et ACP.

FIGURE 12 – Bande passante mesurée sur les différents ports en fonction de la taille des blocs transmis selon le type de transfert.

L'ensemble de nos *micro-benchmarks* a été réalisé sur une carte Digilent Zybo qui met à disposition de l'utilisateur un MPSoC Xilinx ZYNQ-7000 XC7Z010 intégrant un processeur deux cœurs ARM Cortex-A9 de 650 MHz couplé à un FPGA contenant 28000 cellules logiques et 240 ko de mémoire interne. Cette carte nous a permis de réaliser des mesures indicatives et de mettre en évidence le comportement des divers ports de communication, mais elle est bien

moins performante que les MPSoCs pouvant être implantés dans les centre de calcul haute performance. Les accélérateurs ont été synthétisés en utilisant l'environnement de développement Xilinx SDSoC 2017.4 en mode BareMetal 32 bits, avec une fréquence d'horloge fixée à 100 MHz pour l'accélérateur. Les mesures ont été réalisées grâce aux utilitaires `Xil_Time` discutés en fin de section 3.2. Nous avons conduit une série d'expériences présentées ci-après.

Bande passante sur un port HP : Nous avons tout d'abord déterminé la bande passante des transferts mémoire sur un port HP en lecture ainsi qu'en écriture. Pour ce faire, nous transférons des blocs de données depuis la mémoire externe jusqu'à une mémoire tampon allouée sur le FPGA. Les mesures montrent une augmentation de la bande passante en fonction de la taille des blocs transférés jusqu'à arriver à une bande passante limite. Ce phénomène s'explique par le fait que les échanges de données entre mémoire externe et mémoire interne au FPGA s'effectuent à l'aide d'un protocole de communication (le protocole AXI), dont les délais introduits par les signaux de synchronisation sont d'abord prédominants lors des échanges. On obtient finalement une bande passante maximale de 85 Mo/s tant en lecture (figure 12a) qu'en écriture (figure 12b).

Bande passante sur le port ACP : Les mêmes mesures ont été réalisées sur le port ACP afin d'obtenir une base de comparaison. Les débits obtenus sont encore une fois symétriques, de même que ceux des mesures réalisées lors des expériences décrites ci-dessous. Nous nous contenterons donc de mentionner la bande passante de transfert sans préciser si les échanges de données ont lieu dans le sens de la lecture ou de l'écriture. On observe le même phénomène que dans le cas du port HP, avec une augmentation de la bande passante jusqu'à un seuil, qui est dans ce cas de l'ordre de 100 Mo/s en lecture et en écriture. Les résultats plus élevés que dans l'expérience précédente sont à nuancer. En effet, dans le cadre de nos mesures, le cache L2 est entièrement dédié au chargement des blocs de mémoire à transférer et le processeur n'interfère pas avec les opérations de synchronisation. Dans le cas d'une utilisation réelle, cette bande passante est plus faible et doit être déterminée grâce à une étape de *profiling* afin d'établir les contraintes à définir dans la fonction `dispatch` discutée dans la section 3.4.

Bande passante sur l'ensemble des ports HP simultanément : Dans le cadre de l'exécution d'un noyau de calcul complexe, il est possible que des données soient transférées sur l'ensemble des ports HP simultanément. Afin de vérifier qu'il n'y a pas d'interférences entre ces différents ports, nous avons mesuré la bande passante moyenne totale obtenue lors de transferts pipelinés entre la RAM et la mémoire interne à l'accélérateur. Les mesures montrent un comportement très similaire à celui d'un transfert sur un seul port HP, avec une bande passante limite à 85 Mo/s, et l'analyse de la trace d'exécution montre que les interactions entre les contrôleurs des différents ports n'influent pas significativement sur le débit des données.

Bande passante sur l'ensemble des ports simultanément : Afin d'exécuter de façon concurrente un calcul sur processeur et sur accélérateur matériel, il est nécessaire de transférer une partie des données par le port ACP afin de garantir leur cohérence. Nous avons donc mesuré la bande passante moyenne totale obtenue lors d'un transfert de données parallèle sur l'ensemble des ports HP et ACP, afin de déterminer si les opérations de gestion de la cohérence ont un impact sur la vitesse de transfert globale. De même que précédemment, les résultats indiquent que ces opérations n'ont pas de réelle influence. En effet, on obtient un débit moyen de 90 Mbits/s, ce qui est cohérent avec les mesures effectuée indépendamment sur les ports HP et ACP.

Recouvrement des données sur les ports HP : Une exécution réelle peut amener deux parties des données échangées à se recouvrir partiellement voire totalement et à être transférée sur deux ports HP différents. Nous avons cherché à déterminer si ces recouvrements impactent le débit maximal que l'on peut obtenir sur les ports HP, tout d'abord avec un recouvrement partiel puis avec un recouvrement total. Les mesures montrent que les recouvrement n'ont pas d'effet, et une analyse approfondie du circuit généré indique que SDSoC produit une interface qui duplique les données lors du transfert pour éviter tout conflit. On obtient donc une bande passante maximale de l'ordre de 85 Mo/s dans les deux cas.

Recouvrement des données sur les ports HP et ACP : La question de l'influence des recouvrements se pose également dans le cas de données traitées par le CPU et le FPGA simultanément. Nous avons donc mesuré la bande passante obtenue lors du transfert de deux blocs mémoire avec recouvrement partiel (figure 12c) puis total (figure 12d) sur un port HP et sur ACP. Les résultats obtenus sont similaires à ceux de deux transferts indépendants, l'un sur un port HP et l'autre sur ACP. De même que précédemment, on observe que SDSoC prend soin de générer une interface qui duplique les données pour éviter les conflits. La bande passante moyenne maximale obtenue est de 93 Mo/s.

Il est à noter que les bandes-passantes indiquées ci-avant correspondent aux valeurs maximales obtenues, c'est-à-dire lorsque les données sont découpées et transmises par la logique de contrôle des interfaces en paquets de taille maximale.

Ces *micro-benchmarks* nous ont permis de déterminer des caractéristiques importantes de l'architecture Xilinx ZYNQ-7000. L'implantation de l'algorithme présenté en section 3.4 sur une carte Digilent Zybo a cependant été mise en défaut par le comportement conservatif de SDSoC en matière de gestion de la mémoire qui est développé dans la section suivante.

4.2 Limites des outils

Peu avant la fin du stage, nous avons mis à jour une limitation majeure des outils de synthèse développés par Xilinx dans le cadre de notre objectif d'exécution parallèle CPU/FPGA. En effet, une zone mémoire pouvant faire l'objet d'un transfert vers l'accélérateur matériel doit être allouée en utilisant une interface de programmation mettant à disposition de l'utilisateur principalement deux fonctions d'allocations : `sds_alloc_cacheable` et `sds_alloc_non_cacheable`. Toutes deux allouent une zone contiguë de la RAM, la première la déclarant comme *cachable*, c'est-à-dire qu'une partie de cette zone peut résider dans le cache, tandis que la seconde la déclare comme *non cachable* : toute opération d'accès à cette mémoire se fait entièrement par le contrôleur DRAM et aucune données s'y trouvant n'est chargée dans le cache.

Ce second type de zone mémoire est utile pour les données qui seront exclusivement traitées par le FPGA, car il permet de supprimer les vérifications effectuées par le cache lors d'un transfert sur un port HP. La fonction `sds_alloc_non_cacheable` n'est cependant pas utilisable dans le contexte de données traitées à la fois par le processeur et l'accélérateur, car chaque accès à cette mémoire depuis le CPU entraîne une pénalité non négligeable induite par l'appel au contrôleur mémoire externe. Il faut donc allouer les données partagées en les déclarant comme *cachables*.

Cependant, l'utilisation de la mémoire allouée par `sds_alloc_cacheable` n'est pas sans conséquences. En effet, l'analyse des traces système et les mesures de *profiling* que nous avons effectuées montrent que chaque accès à cette mémoire dans le cadre d'un transfert vers FPGA insère des opérations d'invalidation et d'éviction de l'intégralité du cache L2 avant ledit transfert, afin de garantir la cohérence des données transmises. Bien que souhaitable dans la majorité des cas, cette politique très conservative de gestion de la mémoire est un obstacle majeur à

l'implantation de notre algorithme, celui-ci se basant en partie sur une gestion logicielle fine de la cohérence de cache.

Afin de pallier ce problème, il est nécessaire d'implanter une nouvelle fonction d'allocation qui réalise les mêmes opérations que `sds_alloc_cacheable` sans entraîner une éviction complète du cache lors d'un accès extérieur au processeur. Pour cela, il suffit que nous générions nous-mêmes la logique de contrôle du bloc matériel produit par SDSoC. Ce processus est difficile, car une partie de la logique de génération intégrée à l'outil doit être contournée tout en garantissant le bon fonctionnement des interfaces synthétisées. Cet idée fera l'objet d'investigations futures.

5 Conclusion et travail à venir

La possibilité d'exécuter du code parallèlement sur un processeur généraliste et un accélérateur matériel implanté sur FPGA dans un MPSoC ouvre la porte à de nombreuses applications, notamment dans le domaine du calcul distribué et des *clouds*. Il est crucial de mettre en place des interfaces de communication efficaces entre le CPU et l'accélérateur afin que ceux-ci puissent exécuter du code de façon concurrente rapidement et sans laisser à l'utilisateur le soin de gérer manuellement les erreurs de cohérence pouvant être introduites par ce type d'exécution. Nous avons proposé un algorithme basé sur les techniques de compilation polyédrique permettant de générer des instructions de gestion logicielle de la cohérence de cache pour des données traitées à la fois par le CPU et le FPGA. Cet algorithme n'a cependant pas pu être implanté sur une plateforme réelle à cause des nombreux obstacles introduits par les outils et des limites de leurs interfaces.

Nos mesures de performances nous ont permis de mettre en évidence certaines caractéristiques des interfaces de communication mises à disposition de l'utilisateur par l'architecture Xilinx ZYNQ-7000 par le biais des ports haute performance (HP) et du port de cohérence (ACP). L'exploration de l'espace de conception offert par les outils de synthèse de haut niveau réalisée lors de ce stage sont applicables et généralisables à une gamme d'architecture et de plateformes plus large. Ceci permettrait d'étendre les résultats à des infrastructures de HPC grâce à l'implantation de notre méthode sur des MPSoCs tels que l'UltraScale+ développée par Xilinx. Il serait également envisageable de mettre en œuvre une compression des données avant les transferts dans notre algorithme afin de réduire les limitations introduites par la faible bande passante entre CPU et FPGA mise en évidence par nos mesures.

De manière générale, les techniques de compilation polyédrique offrent une grande variété d'application dans le domaine du développement d'applications à destination des accélérateurs matériels implantés sur FPGA. Ces techniques ont été très développées dans le cadre des GPU et des processeurs généralistes, mais il reste de nombreuses pistes à explorer dans le cas des accélérateurs.

Références

[1] *ARM Cortex-A Series Programmers Guide Version 4.0*. p. 126–128. ARM. 2013.

[2] Uday Bondhugula et al. « A Practical Automatic Polyhedral Parallelizer and Locality Optimizer ». In : *SIGPLAN Not.* 43.6 (juin 2008), p. 101–113. ISSN : 0362-1340.

[3] Roshan Dathathri, Ravi Teja Mullapudi et Uday Bondhugula. « Compiling Affine Loop Nests for a Dynamic Scheduling Runtime on Shared and Distributed Memory ». In : *ACM Trans. Parallel Comput.* 3.2 (juil. 2016), 12 :1–12 :28. ISSN : 2329-4949.

[4] Louis-Noel POUCHET et al. « Polyhedral-based Data Reuse Optimization for Configurable Computing ». In : *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '13. Monterey, California, USA : ACM, 2013, p. 29–38. ISBN : 978-1-4503-1887-7.

[5] G. RAMALINGAM. « The Undecidability of Aliasing ». In : *ACM Trans. Program. Lang. Syst.* 16.5 (1994), p. 1467–1471.

[6] Sanket TAVARAGERI et al. *Automatic Generation of Coherence Instructions for Software-Managed Multiprocessor Caches*. Rapp. tech. OSU-CISRC-1/14-TR03. 2014.

[7] *UG1027 SDSoC Environment User Guide*. Xilinx. 2018.

[8] Sven VERDOOLAEGE. « *isl* : An Integer Set Library for the Polyhedral Model ». In : *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*. 2010, p. 299–302.

[9] Sven VERDOOLAEGE et al. « Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions ». In : *Algorithmica* 48.1 (2007), p. 37–66. URL : `http://barvinok.gforge.inria.fr/`.

[10] Chen ZHANG et al. « Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks ». In : *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA : ACM, 2015, p. 161–170. ISBN : 978-1-4503-3315-3.