RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique

Département d'Electronique

Mémoire de master en électronique

Accélération matérielle pour le calcul numérique sur systèmes embarqués

---

Imene DJELLAD

Sous la direction de
Dr. Mourad ADNANE

Présenté et soutenu publiquement le 21/06/2017

Composition du Jury :

Promoteur	M. Nicolas FARRUGIA	MCF.	IMT Atlantique
Co-promoteur	M. Mourad ADNANE	Dr.	ENP
Président	M. Adel BELOUCHRANI	Prof.	ENP
Examinateur	M. Rabah SADOUN	Dr.	ENP

ENP 2017

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique

Département d'Electronique

Mémoire de master en électronique

Accélération matérielle pour le calcul numérique sur systèmes embarqués

---

Imene DJELLAD

Sous la direction de
Dr. Mourad ADNANE

Présenté et soutenu publiquement le 21/06/2017

Composition du Jury :

Promoteur	M. Nicolas FARRUGIA	MCF.	IMT Atlantique
Co-promoteur	M. Mourad ADNANE	Dr.	ENP
Président	M. Adel BELOUCHRANI	Prof.	ENP
Examinateur	M. Rabah SADOUN	Dr.	ENP

ENP 2017

Remerciements

Ce travail n'aurait pas pu être possible sans l'aide et les conseils de nombreuses personnes.

Tout d'abord, je voudrai remercier ma famille et toute personne qui m'a aidé de près ou de loin durant cette période.

Ensuite, je souhaiterai remercier mon encadrant, enseignant et superviseur Dr. Mourad ADNANE pour son soutien.

Je voudrai aussi remercier mes encadrants Pr. Nicolas FARRUGIA et Ghouti BOUCLI HACENE sans lesquels ce stage n'aurait pas pu avoir lieu.

Je suis très reconnaissante envers tous les membres constituants le jury : and Dr. Mourad ADNANE mon encadrant de m'avoir fait l'honneur d'examiner mon mémoire de Master.

ملخص

في هذا الموجز، سوف نرى طريقة جديدة لإنشاء تراكب بفضل اللغات عالية المستوى من أجل تسريع ماديا حساب مصفوفة.

.

الكلمات الدالة : Overlay, FPGA, SDSoc, MMULT.

Abstract

In this thesis, we will see a new way to create Overlays thanks to high-level languages, in order to materially accelerate the matrix calculation.

Key words : Overlay, FPGA, SDSoc, MMULT.

Résumé

Dans ce mémoire, nous verrons un nouveau moyen de créer des Overlays grâce aux langages de haut niveau et cela dans le but d'accélérer matériellement le calcul matriciel.

Mots clés : Overlay, FPGA, SDSoc, MMULT.

Table des matières

Liste des tableaux

Liste des figures

Liste des abréviations

FPGA Field-Programmable Gate Array
HLS High Level synthesis
RTL Register Transfert Logic
PYNQ Python productivity for Zynq
VHDL VHSIC Hardware Description Language
VHSIC Very High Speed Integrated Circuit
AXI Advanced eXtensible Interface
CBL Configurable Logic Blocks
IOB Input/Output Buffers
SOC System On Chip
API Application Programming Interface
PL Programmable Logic
ABI Application Binary Interface
CFFI C Foreign Function Interface
SD Secure Digital
QSPI Quad Serial Peripheral Interface
DDR Double Data Rate
SDRAM Synchronous Dynamic Random Access Memory
HDMI High Definition Multimedia Interface
USB Universal Serial Bus
IMT Institut Mines Télécom
UART Universal Asynchronous Receiver Transmitter
SPI Serial Peripheral Interface
IIC Inter-Integrated Circuit
PS Processing System
MMIO Memory-mapped Input/Output
GPIO General Purpose Input/Output
BRAM Block Ram
RAM Random Access Memory
LED Light-Emitting Diode
DMA Direct Access Memory
TCL Tool Command Language
PE Processing Element
AMBA Advanced Microcontroller Bus Architecture

Contexte et introduction

Contexte

IMT Atlantique (ex :Télécom Bretagne) est une grande école d'ingénieur publique française dont le niveau national est élevé et reconnu, elle est également un centre de recherche international dans les sciences et technologies de l'information et fait partie de l'Institut Mines-Télécom et de l'Université européenne de Bretagne. De plus, elle est partenaire avec plus de 50 établissements d'enseignement supérieur et de recherche en Europe et dans le monde entier et elle collabore avec de nombreux instituts réputés tel que le MIT (Massachusetts Institute of Technology).

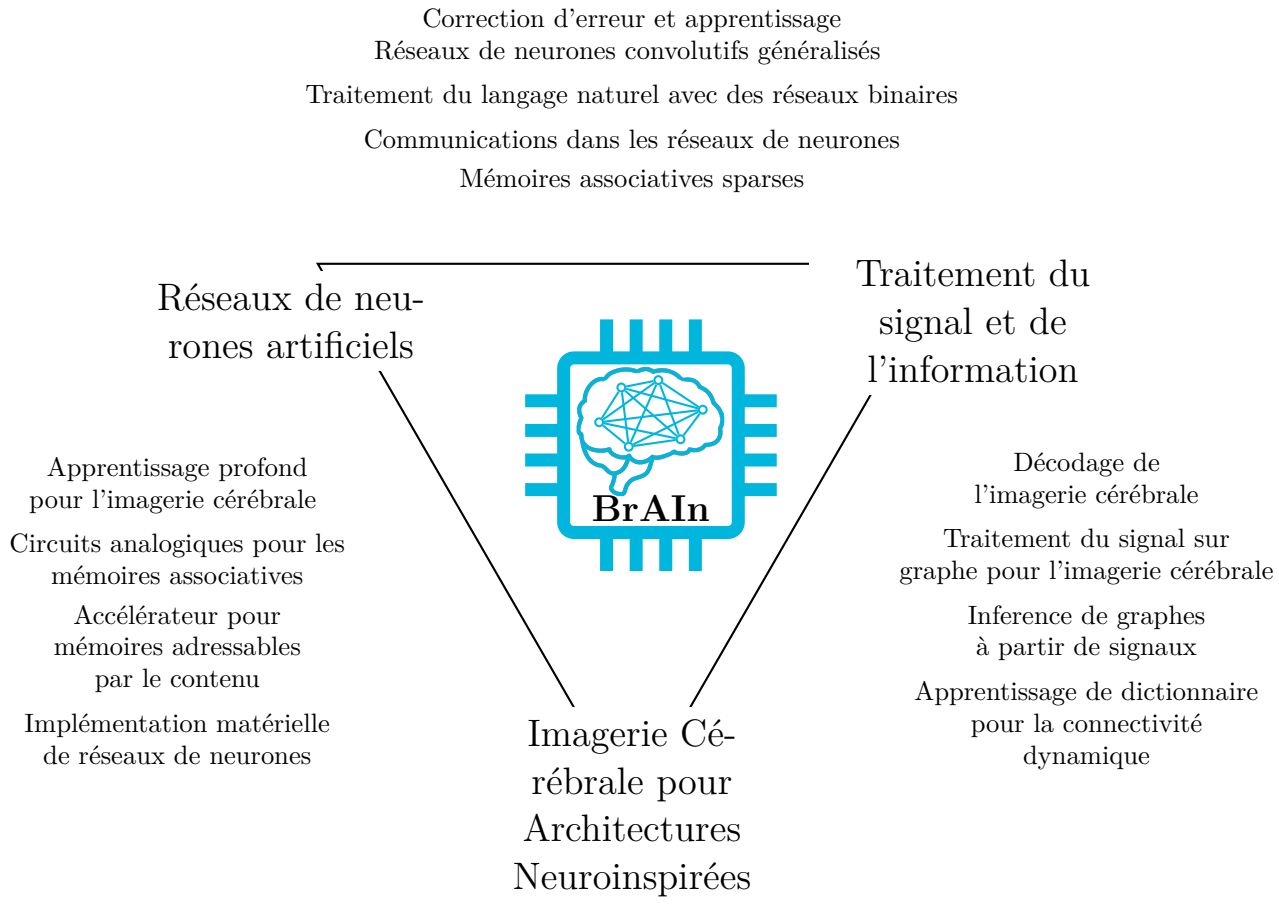
Grâce à son niveau international, l'école accueille des milliers d'étudiants venant de 40 pays désireux de poursuivre des formations d'ingénieur, de master recherche, de master spécialisé et de doctorat.

IMT Atlantique dispose de neuf départements d'enseignement et de recherche :
- Électronique (Brest) où j'ai effectué mon stage.
- Informatique (Brest).
- Image traitement de l'information (Brest).
- Langues culture internationale (Brest).
- Logique des usages, sciences sociales sciences de l'information (Brest, Rennes).
- Micro-ondes (Brest).
- Optique (Brest).
- Réseaux, sécurité et multimédia (Rennes).
- Signal communications (Brest).

Le département électronique met en œuvre des moyens modernes de conception des composants de l'électronique. Il dispose des outils informatiques et du matériel de mesure très performants pour donner aux étudiants et aux chercheurs une formation adaptée aux réalités de la micro-électronique.

L'équipe BrAIn (Brain Inspired Artificial Inteligence) du département d'électronique de l'IMT Atlantique travaille à la frontière de plusieurs domaines, tels que l'informatique, l'électronique, la psychologie et les neurosciences, pour établir des modèles à la fois efficaces et biologiquement plausibles.
Le laboratoire a fondé le projet NEUCOD (pour Neural Coding) qui vise à identifier

et exploiter les analogies observées entre les propriétés du cortex cérébral, et celles des décodeurs correcteurs d'erreurs modernes. Les sujets les plus répandus en informatique dans l'équipe sont : les réseaux à clique dits Gripon-Berrou, le traitement de signal sur graphe, l'analyse des EEG et les modèles neuronaux en général.



Introduction générale

Lorsque nous plongeons dans l'histoire des techniques de développement FPGA depuis qu'ils ont été introduits au milieu des années 1980, nous remarquons une tendance croissante à se développer à un niveau d'abstraction plus élevé ; Le nouvel environnement de développement Xilinx SDSoC continue cette tendance.

Au début, les FPGA ont été développés schématiquement en utilisant des portes logiques standard et des bascules pour implémenter les fonctions requises dans le périphérique. Cependant, cette approche présente plusieurs limitations sévères lorsqu'il s'agit de décrire de grandes conceptions et des fonctionnalités complexes, et pour la vérification. Il est beaucoup plus facile de décrire les fonctionnalités dont nous avons besoin à un niveau supérieur et de permettre à la chaîne d'outils de déterminer l'implémentation du niveau de la porte. C'est là que les langages de description matérielle tels que Verilog et VHDL ont apporté un grand avantage car ils nous ont permis de programmer à un niveau d'abstraction plus élevé où nous décrivons le comportement que nous voulons implémenter et les outils de synthèse déterminent la logique réelle.

Bien sûr, avec les HDL, nous devons coder de manière spécifique et avec des contraintes sur les commandes supportées si nous souhaitons que le code soit synthétisable. En conséquence, beaucoup de développement HDL est effectué au niveau de transfert de registre (RTL). Alors que les HDL ont été le pilier d'un grand nombre d'années, l'augmentation des capacités, de la taille et des performances des périphériques signifie que le développement de modèles basés sur FPGA et SoCs programmables utilisant un HDL est maintenant confronté aux mêmes problèmes que la conception schématique.

Tout cela s'avère compliqué lorsque nous ajoutons des SoC comme Zynq. Nous voulons nous assurer que nous pouvons maximiser les performances SoC en accélérant les fonctions dans le cadre PL (logique programmable) du périphérique. Cependant, nous ne pourrons peut-être pas savoir avant d'être avancés dans le développement si nous avons choisi toutes les bonnes fonctions pour l'accélération. Les changements à un stade ultérieur nécessiteront un temps de développement supplémentaire (indésirable et potentiellement coûteux).

SDSoC nous permet non seulement d'augmenter considérablement le niveau d'abstraction car il s'appuie sur le code généré par Vivado HLS, mais il crée également un environnement de développement étroitement intégré, qui nous permet également de concevoir efficacement l'application SoC et d'atteindre nos objectifs de performance système en accélérant les fonctions sélectionnées dans le côté PL de l'appareil. Le déplacement des fonctions dans le PL à partir du PS (système processeur) est très simple à faire avec l'environnement de développement SDSoC une fois que nous avons déterminé ceux que nous souhaitons déplacer. Ensuite, le processus de génération crée l'image de démarrage en un seul clic.

Pour cela, dans ce projet nous allons voir un exemple de développement d'Overlay grâce à l'outil SDSoC, son intégration dans la PYNQ et des tests de validation.

Dans le chapitre 1, nous verrons comment gérer l'outil de développement SDSoC et les moyens utilisés pour optimiser au maximum les ressources et les temps de calculs.
Nous verrons aussi un exemple crée à partir de cet outil permettant d'effectuer la multiplication de matrices avec de grandes performances.
Dans le chapitre 2, nous verrons les résultats obtenus grâce à l'Overlay crée et son implémentation.

Chapitre 1

Création de l'Overlay MMULT

1.1 Introduction

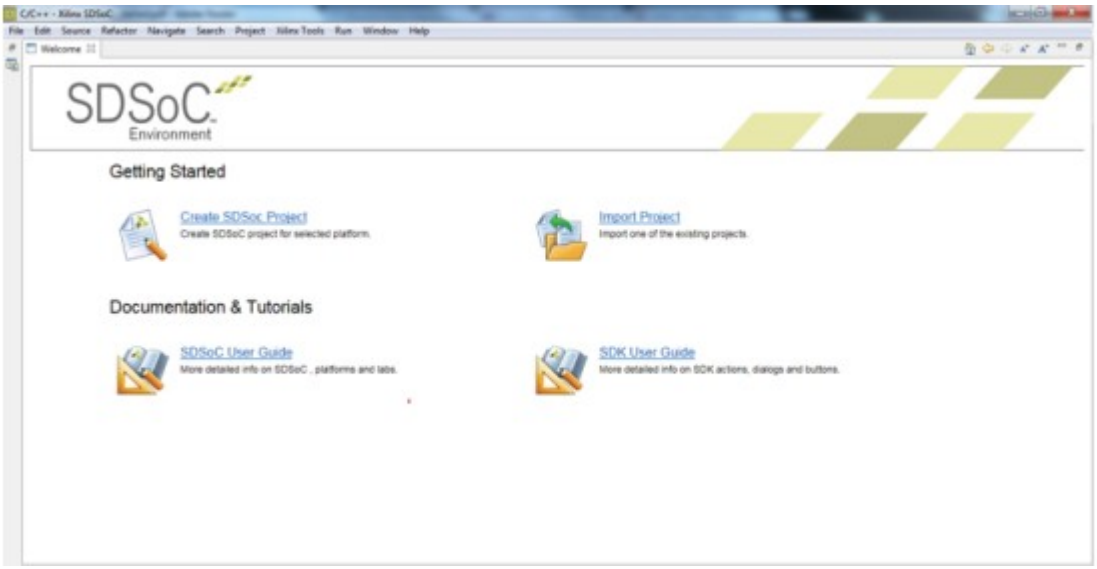
L'un des avantages du système PYNQ est que nous pouvons intégrer des Overlays matérielles pour le Zynq Z-7000 SoC de PYNQ et les utiliser avec facilité dans un environnement de programmation Python. Il est assez simple de créer et d'intégrer un Overlay matériel. Cependant, nous devons encore développer un Overlay avec les fonctions que nous désirons. Idéalement, pour continuer à tirer parti des avantages du système PYNQ de haut niveau, nous voulons développer les Overlays à l'aide d'une approche similaire de haut niveau.
La façon traditionnelle de développer des Overlays matériels pour le FPGA dans le SoC de Zynq consiste à utiliser Vivado, parfois combiné avec Vivado HLS pour implémenter des fonctions complexes définies en C ++.
L'environnement de développement Xilinx SDSoC nous permet de créer des applications qui fonctionnent sur les processeurs ARM Cortex-A9 de Zynq SoC (le système PS ou processeur) et la logique programmable (PL). Nous pouvons déplacer des fonctions entre les deux quand nous souhaitons accélérer les parties du design.

Ce que cela signifie pour le système PYNQ, c'est que nous pouvons utiliser SDSoC pour créer un Overlay matériel à l'aide de Vivado HLS, puis le faire interagir avec l'interface en C (CFFI) de Python. En théorie, cette approche nous permet de créer des Overlays matériels sans avoir besoin d'écrire une ligne de HDL.
Dans ce chapitre, nous allons parler de SDSoC tout en explorant son fonctionnement, son utilisation et le processus de développement d'applications en C ++ pour notre conception Zynq. Nous verrons alors, une nouvelle approche pour le développement d'Overlays.

1.2 Présentation de SDSoC

L'outil de développement SDSoC nous permet de développer notre application en C ++. Au cours de ce développement, nous essayons d'anticiper la demande pour identi-

fier les fonctions à accélérer dans le côté PL de l'appareil, nous pouvons alors construire le système et générer l'image de carte SD.
Une fois notre application est implémentée sur notre carte de développement, nous pouvons analyser les performances et optimiser les fonctions d'accélération si nécessaire.



Interface de l'outil SDSoC
[1]

1.3 Utilisation de l'outil SDSoC pour la création de l'Overlay MMULT

La première étape dans l'utilisation de SDSoC est de créer une plateforme SDSoC. Une plateforme SDSoC requiert à la fois une définition matérielle et une définition logicielle. Nous pouvons créer la définition matérielle à partir de Vivado. La base de conception PYNQ servira de base parce que nous voulons nous assurer que les paramètres PS sont corrects. Cependant, pour libérer des ressources dans le PL pour SDSoC, nous allons éliminer certaines des fonctions logiques.
Donc, nous allons utiliser la conception de base, supprimer la logique dont nous n'avons pas besoin et conserver le matériel dont nous avons besoin.

Nous devons créer un nouveau projet dans l'outil SDSoC et suivre les étapes permettant de nommer le projet, de sélectionner la carte de développement et de sélectionner le système d'exploitation utilisé.

Une fois que la plateforme a été créée dans SDSoC, nous pouvons profiter des bibliothèques HLS prises en charge pour créer l'application que nous souhaitons. SDSoC générera automatiquement le fichier bit requis et le fichier TCL dans la compilation.

Cependant, dans ce cas, nous avons également besoin des fichiers C générés par SDSoC pour l'interface avec la fonction accélérée dans le PL Zynq. Nous faisons cela en utilisant une bibliothèque partagée, que nous pouvons appeler depuis l'environnement Python. Nous pouvons créer une bibliothèque partagée en cochant l'option lorsque nous créons un nouveau projet SDSoC.

Pour la bibliothèque partagée, nous devrons connaître les noms des fonctions qui y sont contenues. Ces fonctions seront renommées par SDSoC pendant la procédure de construction et nous devrons utiliser ces noms modifiés dans l'interface Python CFFI car c'est ce qui est inclus dans la bibliothèque partagée.

Au début, nous allons travailler dans le côté PS du Zynq. Nous allons vérifier que tout marche en testant sur la partie PS notre code puis nous pourrons le déplacer vers le PL.

Le projet est composé des dossiers suivants :

- Fonctions matérielles SDSoC : c'est là que nous trouverons par la suite les fonctions que nous avons transférées dans le matériel.
- Include : l'extension de ce fichier montrera tous les fichiers d'en-tête C ++ utilisés dans la construction.
- Src : les codes sources utilisés dans le projet.

Nous allons alors commencer par nous assurer que tout est bien configuré avec non seulement notre installation et notre environnement SDSoC, mais aussi notre plan de développement consiste à créer la démo afin qu'elle s'exécute sur le côté PS de l'appareil.

La construction du projet ne devrait pas durer trop longtemps et fait apparaitre les éléments suivants dans le projet Explorer :

- Binaries - Dans ce cas, nous trouvons les fichiers Extensible Linker File (ELF) créés à partir du processus de compilation du logiciel.
- Archives - Nous trouverons les fichiers objet, qui sont liés pour créer les fichiers binaires.
- SDRRelease - Cela contient nos fichiers de démarrage et ne sera pas présent avant de faire une compilation.

Maintenant, nous avons construit la première démo, qui s'exécute uniquement sur le Zynq PS.

Avec la construction complète, nous copions tous les fichiers sous le dossier SDRelease -> sd_card sous l'explorateur de projet sur une carte SD et nous l'insérons dans la PYNQ. Avec un programme terminal connecté à la PYNQ et la séquence de démarrage terminée, nous devons exécuter le programme.

Nous allons maintenant déplacer la fonction de multiplication dans le côté PL (logique programmable) du Zynq SoC.

L'étape suivante consiste à décharger notre fonction au côté PL du Zynq SoC. Cela se fait en sélectionnant la fonction d'intérêt et cliquer sur Toggle HW / SW pour la faire passer du hardware au software.

Une fois que nous l'avons fait, SDSoC linker appellera automatiquement Vivado HLS et Vivado pour implémenter les fonctions du côté PL des SoC. Il créera également les pilotes pertinents dans le logiciel pour supporter ce passage au matériel.

En réalité, le déchargement de la fonction sur le côté PL du périphérique devient transparent, sauf que nous obtenons une augmentation de performance significative.

SDSoC accélère les fonctions du côté PL (logique programmable) du Zynq SoC en utilisant quelque chose appelé «cadre de connectivité», qui décrit les connexions logiques et physiques entre les côtés PL et PS (processeur) du périphérique. Sans surprise, SDSoC inclut une API permettant des transferts dans ce cadre.

Mais comment l'outil sait-il quelles connexions logiques et physiques sont disponibles ? SDSoC réalise cela en utilisant deux définitions de plateforme : l'une définit le matériel et l'autre le logiciel. Dans la plateforme matérielle, nous trouverons la définition de la plateforme de base créée dans Vivado. Nous verrons donc :

- Horloges - Toutes les horloges utilisées dans la plate-forme SDSoC doivent provenir des horloges du processeur
- Réinitialisation - Le nombre de réinitialisations disponibles
- Interruptions - Le nombre d'interruptions disponibles
- AXI - Le nombre de connexions AXI et AXI-Streaming disponibles

Cela signifie que nous pouvons développer une plateforme de base en utilisant Vivado, en utilisant des périphériques personnalisés, puis en exportant la plateforme vers SDSoC.

SDSoC nous permet donc de :

- Employer Vivado HLS pour générer une logique pour le côté PL du Zynq SoC
- Analyser les communications vers et depuis la fonction
- Établir un réseau de communication AXI basé sur l'analyse ci-dessus
- Générer une fonction stub du logiciel pour accélérer la fonction

C'est cette fonction stub qui est réellement appelée au lieu de la fonction accélérée. Alors que les interfaces du logiciel avec la fonction stub restent identiques, sa fonctionnalité est très différente. La fonction stub utilise la structure de connectivité pour initialiser et envoyer / recevoir des données vers et depuis le matériel PL où réside la fonction accélérée.

La façon dont fonctionne le cadre de connectivité est vraiment passionnante. Il utilise des appels d'API logiciels indépendants de la mise en œuvre pour synchroniser les transferts de données vers et depuis le PL. Lorsque le code est construit, ces appels sont ensuite traduits aux pilotes corrects en fonction de la configuration du réseau AXI créé.

La fonction mmult () est implémentée en C ++, mais elle contient également un certain nombre de pragmas. Un pragma est une directive qui indique à SDSoC comment il doit traiter son entrée. L'exploration de ces pragmas et leur rôle dans l'amélioration de la performance constituent un excellent point de départ pour examiner la synthèse des niveaux élevés (HLS) et la manière dont elle est utilisée dans SDSoC et Vivado HLS.

Nous utilisons pragmas pour contrôler les optimisations et le comportement de synthèse dans SDSoC de la même manière que nous les utilisons lors du développement de modèles en utilisant des méthodes de codage SoC / FPGA standard.

1.4 Role des Pragmas utilisées dans le code

- pragma HLS INLINE self - La commande INLINE supprime toute hiérarchie dans une fonction, ce qui permet des optimisations à travers les limites. Ceci est semblable à la définition de l'option de la hiérarchie aplatie dans la synthèse HDL.
- pragma HLS PIPELINE II = 1 - Le pipeline est la façon dont les ingénieurs FPGA ont traditionnellement augmenté la fréquence de fonctionnement d'un circuit tout en augmentant la latence. Le pipeline SDSoC se comporte de manière similaire, ce qui nous permet de faire fonctionner des pipelines ou des boucles. Dans le code d'exemple ci-dessus, le pragma du pipeline est situé avant la boucle finale pour s'assurer que HLS déroule la boucle la plus interne et effectue toutes les multiplications à la fois. Dans ce pragma, le «II» immédiatement suivant fait référence à «intervalle d'initiation». Ce paramètre définit un nombre cible de cycles d'horloge de la fonction ou de la boucle pour le traitement d'une nouvelle entrée. Le réglage de ce paramètre sur 1 signifie que la fonction ou la boucle traitera une nouvelle entrée à chaque cycle d'horloge. Dans cet exemple, le résultat est une multiplication parallèle. HLS tentera d'atteindre cette cible spécifiée. Si cela n'est pas possible, il créera un design avec l'intervalle d'initiation le plus bas possible et émettra un avertissement, ce qui nous permettra d'analyser et d'optimiser la conception pour tenter d'atteindre l'intervalle d'initiation requis.
- pragma HLS array_partition - Cette pragma partitionne un tableau en petits éléments (en utilisant deux pragmas pour l'entrée a et l'entrée b). Un réseau partitionné permet plusieurs lectures et écritures simultanées, ce qui augmente potentiellement le débit. Le partage de tableaux est important car nous avons déroulé la dernière boucle "for" et effectuons simultanément toutes les multiplications dans le matériel PL de Zynq SoC. Le "facteur" de partitionnement de 16 spécifie que 16 tableaux doivent être créés tandis que le paramètre "dim" définit la dimension du tableau à diviser. La valeur spécifiée de 2 crée un tableau bidimensionnel.

L'utilisation de ces pragmas démontre une solution optimale pour cet exemple.

Pour tirer le meilleur parti de SDSoC, nous devons vraiment comprendre comment nous codons efficacement pour HLS (synthèse de haut niveau). Comprendre la façon dont nous codeons pour HLS nous permet de développer notre algorithme de haut

niveau, puis de le traduire à RTL, en économisant beaucoup de temps sur la conversion de C ++ / System C vers RTL. Cela nous permet également de vérifier notre algorithme au niveau C et la vérification à ce niveau est plus rapide car nous n'avons pas besoin de compte pour les cycles delta.

1.5 Fonctionnement de l'outil Vivado HLS

HLS fonctionne en trois phases principales :

- Planification - pendant cette phase, les opérations effectuées pendant chaque cycle d'horloge sont assignées. Pour ce faire, HLS a des informations sur la fréquence de fonctionnement et le périphérique cible. Il planifie l'ordre des opérations en utilisant ces informations. Bien sûr, la détente des conditions de synchronisation ou l'utilisation d'un dispositif de performances supérieures entraînent une mise à jour de la planification HLS pour inclure plus d'opérations dans ce cycle d'horloge. C'est dans cette phase que le pragma utilisé pour contrôler la synthèse et l'optimisation est appliqué.
- Liaison - dans cette phase, les opérations identifiées dans l'opération de planification sont liées aux ressources disponibles dans la logique programmable.
- Extraction de la logique de contrôle - Dans cette phase, la logique de contrôle requise est extraite et les machines d'état sont construites pour contrôler le flux de mise en œuvre au besoin.

Après avoir compris les trois étapes du HLS, nous devons mieux comprendre comment HLS synthétise notre code, en gardant à l'esprit ce qui finira par se transformer en matériel.

Les trois phases HLS ci-dessus sont appliquées aux fonctions C du système C ++. En tant que tel, l'E/S de la fonction synthétisée est déterminée par les paramètres passés et retournés à partir de la fonction. Cependant, il existe un certain nombre d'autres comportements intéressants qui doivent être pris en considération. La première est que si la fonction contient des appels vers d'autres fonctions, ces autres fonctions seront synthétisées. En tant que tel, ils doivent être écrits pour soutenir HLS.
Comme HLS synthétise les fonctions par défaut, il laissera les boucles déroulées et les tableaux seront synthétisés dans les mémoires. C'est pourquoi nous avons examiné plus haut comment nous pourrions optimiser à la fois les mémoires et les boucles en utilisant pragmas pour obtenir les meilleures performances.

Lorsque nous utilisons HLS dans SDSoC, il existe aussi quelques règles que nous devons suivre. La fonction à accélérer doit être dans son propre fichier. (Les fonctions appelées peuvent être dans le même fichier, par exemple). Le code qui appelle la fonction doit être dans un fichier distinct.

Bien sûr, il existe d'autres contraintes lors de l'utilisation de SDSoC (ou HLS). L'un

d'entre eux est similaire aux langues HDL en ce sens qu'il existe des constructions qui peuvent être synthétisées et celles qui ne peuvent pas l'être. Par exemple, nous ne pouvons pas utiliser les variables globales, qui sont partagées entre le logiciel et la fonction à accélérer. Nous ne pouvons pas surcharger les fonctions. Les fonctions doivent contenir au moins un argument et les paramètres de retour doivent correspondre à 32 bits.

Lors de l'écriture de fonctions C traditionnelles, il est souvent fréquent d'allouer et de libérer de la mémoire système à l'aide des fonctions malloc (), alloc () et free (). Cette technique de codage pose des problèmes de synthèse car nous devons garantir que les exigences de mémoire de la fonction à synthétiser sont limitées. La mémoire non consolidée ne peut pas être synthétisée à l'aide de ressources matérielles finies.
Nous pouvons utiliser des pointeurs dans le code que nous avons l'intention de synthétiser, mais nous devons suivre certaines règles de base si elles doivent être synthétisées efficacement. La première et la plus importante règle concerne le casting. Si nous voulons pouvoir utiliser HLS, nous pouvons lancer entre des types C natifs. Cependant, nous ne pouvons pas utiliser des astuces générales. Il est permis d'utiliser des pointeurs dans la liste des paramètres de fonction, d'utiliser l'arithmétique des pointeurs et d'utiliser des pointeurs vers des tableaux.

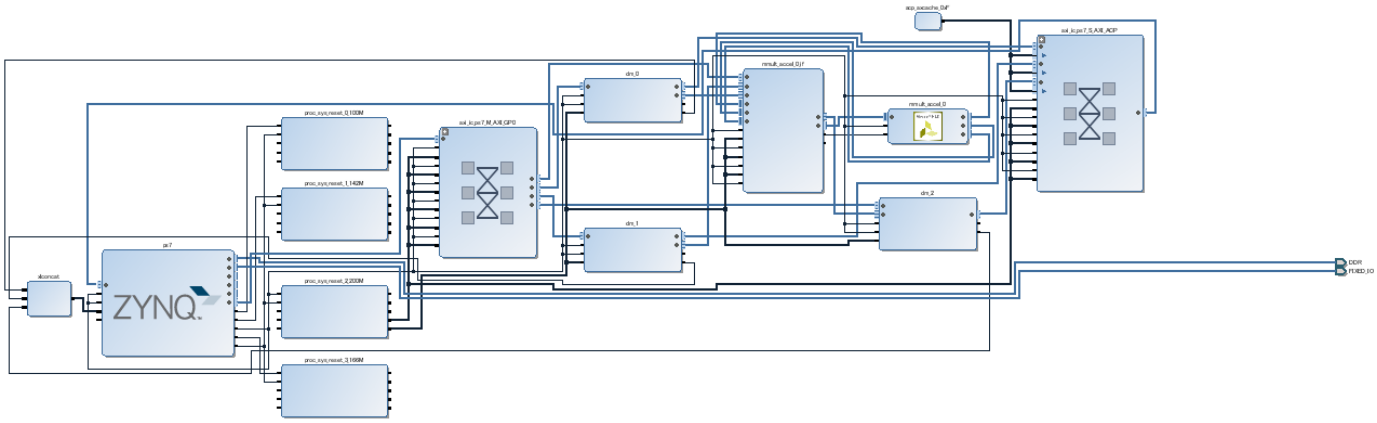
Nous devons également nous assurer que nous n'utilisons pas de fonctions récursives dans la fonction que nous souhaitons accélérer en utilisant HLS. Une fonction récursive est celle qui s'appelle soit un nombre fini, soit un nombre infini de fois.

Outre les styles de codage qui permettent la synthèse, il existe également des styles de codage qui permettent une meilleure optimisation. Par exemple, une boucle avec des limites variables peut empêcher l'optimisation car HLS ne peut pas déterminer la latence de la boucle. Nous pouvons résoudre ce problème en utilisant une macro assert dans le code C pour fournir le nombre maximum de boucles. Il convient de noter ici que nous ne pouvons pas dérouler une boucle de limites variables car HLS ne sait pas combien de matériel créer. Dans de tels cas, nous souhaitons examiner les moyens de réécrire la fonction en fonction d'un nombre fixe d'itérations.

Les boucles offrent une excellente zone d'optimisation. En utilisant des boucles imbriquées, nous pouvons réduire les temps de calculs en déroulant les boucles et crée du matériel dédié pour chaque itération en effectuant tous les calculs simultanément, ce qui entraînera la plus faible empreinte logique . Bien sûr, la performance croissante exige plus de ressources, qui peuvent ou non être autorisées selon les ressources disponibles.

Pour aplatir les boucles, ils doivent être parfaites ou semi parfaites. Il n'y a qu'une seule différence entre les deux. Une boucle parfaite a des limites définies alors qu'une boucle semi-parfaite permet à la boucle externe d'être variable.

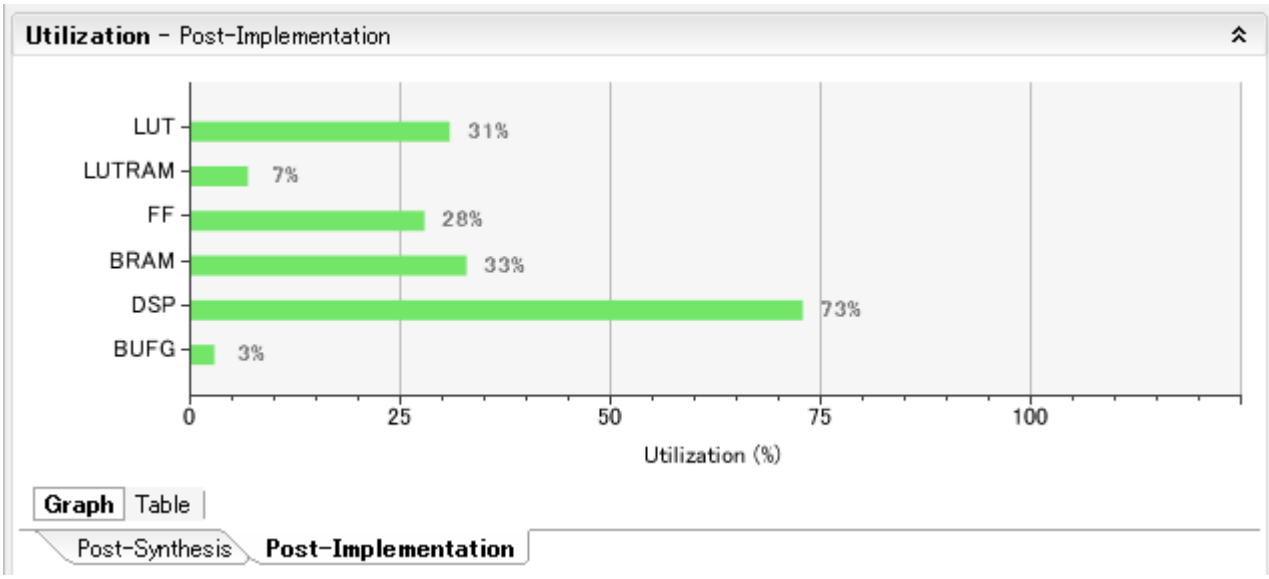
A la fin nous avons généré la construction matérielle suivante dans Vivado 1.2 :



Block design de l'Overlay MMULT tiré du logiciel Vivado
[2]

De toute évidence, nous pouvons voir l'ajout du matériel accéléré.

Tout ce qu'il faut maintenant est de télécharger le bit, tcl, et donc les fichiers vers le PYNQ, puis d'écrire un cahier pour les mettre au travail.
Nous pouvons voir sur la figure 1.3, le pourcentage de ressources utilisées.



Rapport d'utilisation des ressources de la PYNQ
[2]

1.6 Conclusion

En résumé nous avons :

- éliminé les éléments dont nous n'avions pas besoin dans l'architecture de base ;
- exporté l'architecture minimisé vers SDSoc ;
- écrit un programme optimisé en C++ permettant d'effectuer la multiplication de deux matrices ;
- compilé le tout en laissant à SDSoc le soin de générer les fichiers nécessaires à l'implémentation sur PYNQ.

Chapitre 2

Simulations et analyses

2.1 Introduction

Après avoir généré les fichiers bit et tcl ainsi que la bibliothèque partagée et les avoir intégrés dans la PYNQ, on écrit les fonctions permettant d'interagir avec l'accélérateur mmult et on analyse les résultats obtenus.
Dans ce chapitre, nous allons analyser le comportement de l'accélérateur et le comparer avec la multiplication sans accélération et nous allons exposer la performance de notre Overlay.

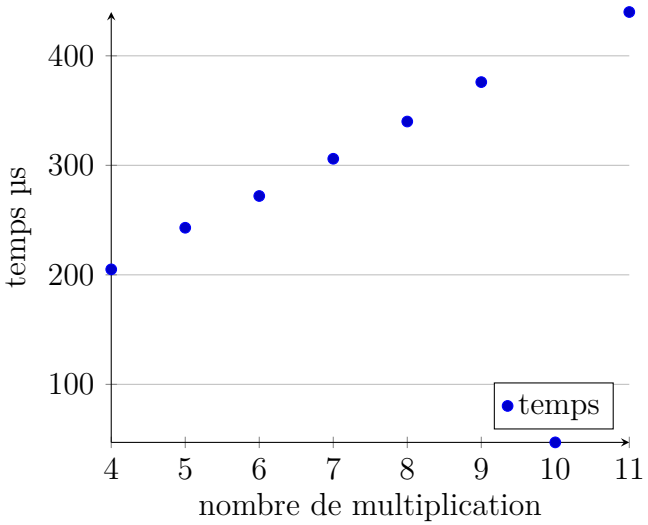
2.2 Simulations

Nous allons étudier les temps de calculs donnés par l'accélérateur MMULT et les temps de calculs sans accélérateur, soit directement sur le processeur de la PYNQ et les comparer. MMULT permet de calculer la multiplication de matrices. Dans les exemples suivants, nous avons généré des matrices de données aléatoires grâce à python et nous avons étudié les temps pris pour le calcul suivant le nombre de multiplications de matrices de mêmes tailles.

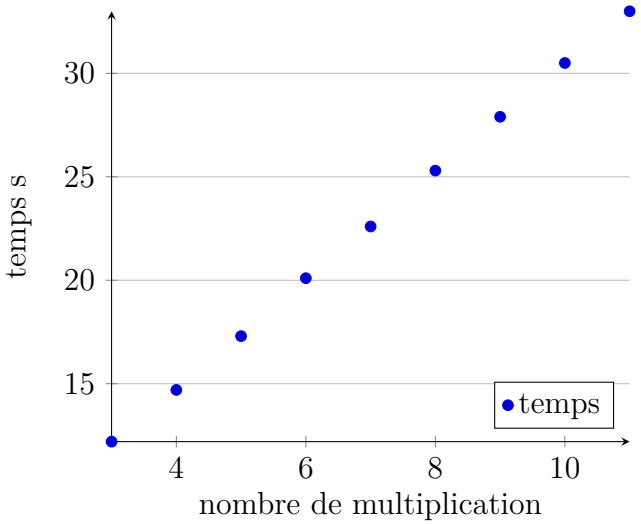
2.3 Matrice de taille 256 X 256

Nous allons commencer par une matrice de taille 256 X 256 et voir le temps mis en hardware (en utilisant l'accélérateur MMULT) et en software (sans accélérateur) selon le nombre de multiplications.
Le graphe 2.1 illustre le temps de calcul pour l'accélérateur de MMULT en fonction du nombre de multiplications. On voit bien que le temps augmente en fonction du nombre de multiplications, ce qui est tout à fait normal, le temps augmente avec le nombre de calcul à effectuer. Par contre, le temps pris pour ces calculs est de l'ordre des micro secondes.

Sur le graphe 2.1, on voit bien que sans accélération, ce temps est de l'ordre des secondes, MMULT est donc un accélérateur assez performant.

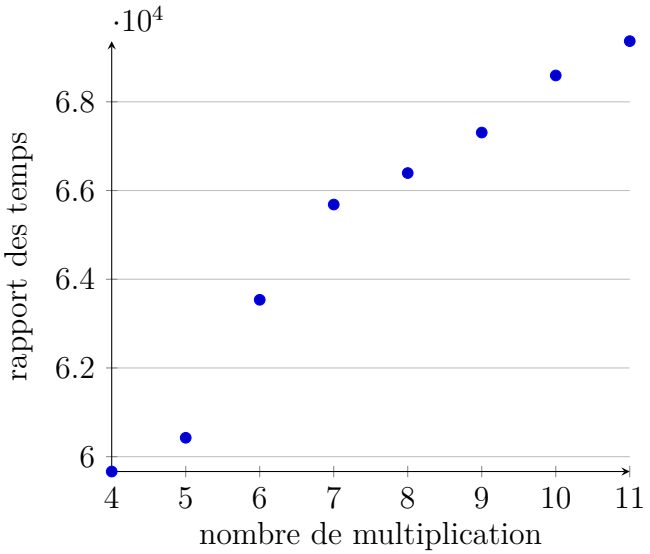


Temps de calcul pour l'accélérateur de multiplication en fonction du nombre de multiplications pour une matrice 256 X 256



Temps de calcul sans accélération en fonction du nombre de multiplications pour une matrice 256 X 256

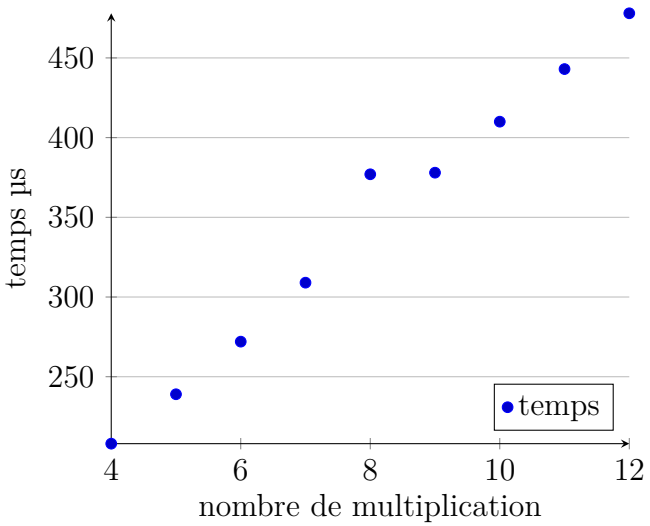
Dans le graphe 2.3, on compare le temps pris avec et sans accélération pour effectuer les multiplications de matrices. On trace le ratio du temps pris sans accélération sur le temps pris avec accélération. On peut voir grâce à la courbe croissante que l'accélérateur est très performant quel que soit le nombre de multiplication mais ce qui est encore plus intéressant est que sa performance augmente avec le nombre de multiplication. Le temps qu'il prend ne se démultiplie pas, il augmente légèrement, ce qui est très intéressant.



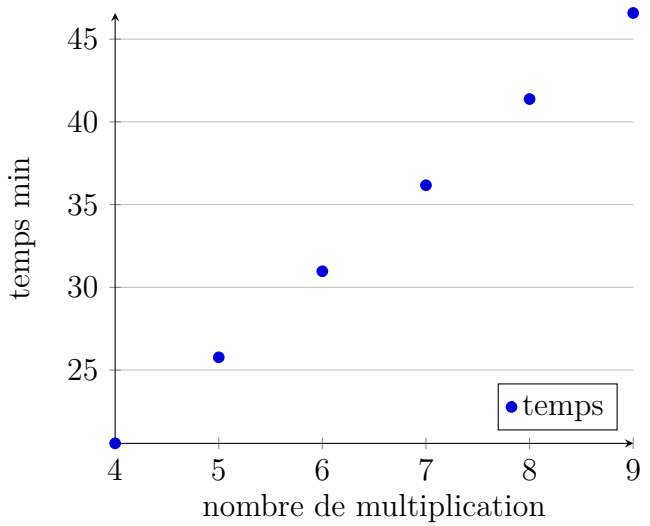
Rapports de temps de calcul software/hardware en fonction du nombre de multiplications pour une matrice 256 X 256

2.4 Matrice de taille 1024 X 1024

Nous allons maintenant monter d'un cran de difficulté en augmentant la taille de la matrice à 1024 X 1024 et voir le temps mis en hardware (en utilisant l'accélérateur MMULT) et en software (sans accélérateur) selon le nombre de multiplications.
Le graphe 2.4 illustre le temps de calcul pour l'accélérateur de MMULT en fonction du nombre de multiplications. On voit bien que le temps augmente en fonction du nombre de multiplications, ce qui est tout à fait normal, le temps augmente avec le nombre de calcul à effectuer. Par contre, le temps pris pour ces calculs est encore de l'ordre des micro secondes malgré l'augmentation considérable de la taille des matrices.
Sur le graphe 2.5, on voit bien que sans accélération, ce temps a beaucoup augmenté, il est passé des secondes au minutes. Vu les résultats obtenus, il est clairement inutile de tracer le ratio des temps car l'accélérateur est largement performant et incomparable au calcul sur processeur sans accélération.



Temps de calcul pour l'accélérateur de multiplication en fonction du nombre de multiplications pour une matrice 1024 X 1024



Temps de calcul sans accélération en fonction du nombre de multiplications pour une matrice 1024 X 1024

Taille de la matrice	Meilleur temps hardware	Meilleur temps software
32 X 32	514 µs	55.4 ms
64 X 64	515 µs	442 ms
128 X 128	515 µs	4.29 s
256 X 256	518 µs	35.8 s
512 X 512	519 µs	7min 44s

Comparaison entre les temps hardware et software pour différentes tailles de matrices et un nombre de multiplications fixé et égale à 15

2.5 Conclusion

Nous pouvons conclure que l'accélérateur MMULT est très performant et que son intérêt grandit exponentiellement avec la taille des matrices utilisées.

Chapitre 3

Conclusion

Dans ce travail, nous avons vu une autre manière de créer les Overlays et ceci grâce à l'outil SDSoC.
Nous avons réussi à obtenir de grandes performances pour les mutiplications de matrices. En effet, les temps de calculs ont été considérablement réduits grâce à l'accélération matérielle de la fonction de multiplication.
Nous voyons alors l'intérêt de l'accélération matérielle.

Les perspectives de ce travail seraient d'utiliser cet accélérateur matériel afin d'accélérer les calculs des réseaux de neurones.

Bibliographie

[1] *SDSoC, step by step : build a sample design*. URL : http://www.epdtonthenet.net/article/116024/SDSoC--step-by-step--build-a-sample-design.aspx.

[2] *Overlay MMULT*. URL : https://github.com/tkat0/pynqmmult.

[3] *PYNQ*. URL : https://forums.xilinx.com/t5/Xcell-Daily-Blog/Python-Zynq-PYNQ-which-runs-on-Digilent-s-new-229-pink-PYNQ-Z1/ba-p/726277.

[4] *FPGA*. URL : http://jjmk.dk/MMMI/PLDs/FPGA/fpga.htm.

[5] *Average categorize some mnist digits*. URL : http://write-up.semantic-db.org/195-average-categorize-some-mnist-digits.html.

[6] *Réseau de neurones artificiels*. URL : https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels.

[7] Amer Baghdadi. *Systèmes embarqués*. URL : https://formations.telecom-bretagne.eu/fad/course/index.php.

[8] Adam Taylor. *Exploring PYNQ's Base PL*. URL : https://forums.xilinx.com/t5/Xcell-Daily-Blog/Adam-Taylor-s-MicroZed-Chronicles-Part-157-Exploring-PYNQ-s-Base/ba-p/734085.

[9] *AMBA AXITM and ACETM Protocol Specification*. URL : http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf.

[10] *TensorFlow*. URL : https://en.wikipedia.org/wiki/TensorFlow.

[11] *Vivado*. URL : https://en.wikipedia.org/wiki/Xilinx_Vivado.