

## **Master E3A**

Cours A2 : Systèmes Electroniques Embarqués

Projet : Multicores embarqué pour Big Data et Machine Learning

**DOU Yuhan**  
**FORCIOLI Quentin**  
**GHAOUI Mohamed Anis**  
**TERRACHER Audrey**

Encadré par : **HAMMAMI Omar** et **LE PROVOST Hervé**

Master 2 : Système Embarqué et Traitement de l'Information

Année : 2019-2020



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Méthodologie et approche</b>	<b>3</b>
<b>3</b>	<b>Conception logicielle embarquée</b>	<b>4</b>
3.1	Présentations des algorithmes . . . . .	4
3.1.1	Multiplication de Matrice de Bloc (BMM) . . . . .	4
3.1.2	Estimateur de $\pi$ . . . . .	4
3.1.3	K-means . . . . .	5
3.1.4	Analyse par Composantes Principales . . . . .	5
3.1.5	Coefficient de Pearson . . . . .	6
3.1.6	Décomposition par valeur singulière . . . . .	6
3.2	Implémentation et tests des algorithmes sur ARM . . . . .	6
3.3	Optimisation des algorithmes sur ARM . . . . .	6
3.4	Implémentation sur $\mu$ blaze . . . . .	7
<b>4</b>	<b>Conception Synthèse Haut Niveau</b>	<b>8</b>
4.1	Conception d'une IP avec directives . . . . .	8
4.2	Interfaçage d'une IP avec Axi/Axilite . . . . .	9
4.3	Approche d'implémentation des algorithmes . . . . .	9
4.3.1	Critères de conception d'IP . . . . .	10
<b>5</b>	<b>Implémentation matérielle</b>	<b>11</b>
5.1	Introduction . . . . .	11
5.2	Demo multi-CPU . . . . .	11
5.2.1	Concept et problem . . . . .	11
5.2.2	Utilisation de la BRAM . . . . .	13
5.2.3	Passage de données vers un microblaze . . . . .	14
5.3	IP HLS . . . . .	15
5.3.1	Concept . . . . .	15
5.3.2	Implémentation et clocking . . . . .	15
5.3.3	Performances . . . . .	19
5.4	Amélioration et future design . . . . .	19
<b>6</b>	<b>Performances Mesurées</b>	<b>19</b>
6.1	Performances ARM . . . . .	19
6.2	Performances HLS . . . . .	21
6.3	Performances PC . . . . .	21
<b>7</b>	<b>Conclusion et perspectives</b>	<b>22</b>

## Table des figures

1	Illustration de l'estimation de $\pi$ par la méthode Monte-Carlo . . . . .	5
2	Représentation des projections de PCA . . . . .	6
3	Schéma bloc de l'architecture du $\mu$ blaze . . . . .	7
4	Conception de l'IP : . . . . .	10
5	Occupation des ressources FPGA pour différentes IP . . . . .	11
6	Exemple de design pour le Multi CPU pouvant exécuter un programme sur le FPGA et sur le microblaze. . . . .	12
7	Linker script du microblaze : exécution depuis la DDR et stack et heap étendus	13
8	Block design pour tester la BRAM : . . . . .	14
9	Code du microblaze(droite) et du zynq(gauche) pour l'envoi et la réception de message . . . . .	14
10	Démonstration de la BRAM : le microblaze affiche un message envoyé depuis le Zynq . . . . .	15
11	Exemple d'IP avec ses différents ports . . . . .	16
12	Paramètre du Zynq : ce qu'il faut cocher pour avoir l'interface ACP et la cohé- rence des caches . . . . .	17
13	Exemple de design pour l'IP multiply_block_32 . . . . .	17
14	Écran du timing report permettant de connaître le clock skew . . . . .	19
15	Temps d'exécution sur ARM des différents algorithmes . . . . .	20
16	Un exemple des tailles de codes produit par l'optimisation . . . . .	20
17	Un exemple des temps d'exécution des codes produit par l'optimisation . . . . .	21
18	Temps d'exécution sur PC . . . . .	22

## 1 Introduction

Durant le siècle dernier, les algorithmes sont passés d'une sombre idée à la base de toutes choses faites. Ces algorithmes traitent les données à leur entrée dans des temps parfois constant, consistant et parfois moins déterminés quand ils sont exécutés de manière totalement séquentielle. Avec la récente explosion du Big Data, ces algorithmes doivent traiter de plus en plus des données de plus en plus volumineuse et multi-dimensionnelles. Il est évident qu'une solution parallèle doit être déployée afin de répondre à une telle demande.

On propose alors de concevoir des architectures pouvant effectuer un calcul parallèle utilisant le principe simple qu'une implémentation matérielle est toujours plus rapide, à fréquence égale, qu'une implémentation logicielle. Le problème est que la conception de ces implémentation par des outils de synthèse demande beaucoup de ressources humaine et de réflexions afin de produire un résultat correct et performant. On pense alors à faire usage des outils de synthèse automatisée qui eux vont générer l'implémentation logique de transfert par registres (RTL).

On se place sur une architecture dite hétérogène du fait qu'elle comporte un noyau CPU-ARM et un noyau FPGA sur la même puce, d'où l'appellation *System on Chip*. Ce système est sur la carte *ZedBoard* développer par *Xilinx*. En utilisant les outils proposés par *Xilinx*, on peut effectuer le développement des algorithmes logiciellement ou/et matériellement.

Dans ce document, on procède à l'explication de la méthode et l'approche qu'on a suivi durant l'étude et la conception des algorithmes, la conception des IP HLS qui seront la traduction de ces algorithmes en RTL, l'implémentation matérielle sur la carte et enfin on procède à l'exposition des résultats et une conclusion. On ne présentera pas en détail toutes les implémentations effectués.

## 2 Méthodologie et approche

L'équipe consistant en 4 personnes, 2 personnes sont confiées la tâche de conception des logicielles à implémenter. i.e. Choisir les algorithmes à étudier, choisir le stratagème d'implémentation et la définition des besoins en terme de mécanisme C/C++, de jeu de données et de tests à valider sur PC, ARM et  $\mu$ blaze. Elles passent par une première implémentation qui n'est pas nécessairement optimisée afin de ne pas bloquer les autres parties de l'équipe. Une fois une première implémentation proposée au HLS, l'optimisation de cette implémentation pour ARM commence. Ce processus itératif consiste en l'analyse de l'architecture cible comparée à l'architecture des ordinateurs usuels (PC bureau).

En essayant de maximiser le parallélisme potentiel de l'équipe, Le concepteur HLS procède à l'analyse les implémentations C/C++ proposées par la partie logicielle embarquée, l'adaptation de ces implémentations d'algorithme aux contraintes matérielles sur la carte Zboard. En effet, certains mécanismes logiciels sont simplement impossibles à réaliser en matériel et donc la conception HLS sera une version modifiée afin de créer une IP implémentable et interfaçable dans un premier par ARM puis par  $\mu$ blaze.

En parallèle, le concepteur matériel effectue le développement de projets et cartes modèles : des projets contenant des exemples d'exécution sur ARM, des implémentation d'IP ne faisant rien à part communiquer et de l'implémentation du  $\mu$ blaze.

Enfin, l'équipe logicielle utilisera les cartes développer pour exécuter leurs codes C. Le concepteur HLS se met en cycle de développement itératif pour optimiser les IP HLS. Le concepteur matériel fera un système temporellement hétérogène (multi-horloges).

Il est important de comprendre cette méthodologie car elle est plus importante que les résultats en eux même. En effet, les dépendances inter-équipe sont l'un des goulots les plus contraignant dans le monde de la recherche et l'industrie. On passe trop de temps à communiquer inefficacement et à être bloqué par les autres parties si cette méthodologie n'est pas

respectée. Cet effet est encore plus important dans une équipe aux compétences variées et de domaine hétérogène.

L'équipe est composée par :	DOU Yuhan	Développement Logiciel
	GHAOUI Anis	HLS et chef d'équipe
	FORCIOLI Quentin	Développement Matériel
	TERRACHER Audrey	Développement Logiciel

## 3 Conception logicielle embarquée

On a choisi plusieurs algorithmes de la série Apprentissage Automatique dans SPARK. On les a en début implémenté en C sur PC avec différentes tailles des ensembles de données. Puis, on passe à l'exécution sur ARM et enfin sur  $\mu$ blaze. On citera ces algorithmes sans trop détailler leur implémentations. Les informations complémentaires seront présentées devant le jury lors de la présentation finale.

### 3.1 Présentations des algorithmes

#### 3.1.1 Multiplication de Matrice de Bloc (BMM)

Cet algorithme vise à optimiser la multiplication de matrices en décomposant les grandes matrices en petites sous-matrices. Supposons que l'on va calculer la multiplication de A (MxP) et B (MxP) :

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

avec  $C_{11} = A_{11}.B_{11} + A_{12}.B_{21}$

Dans le cas général sans utiliser de blocs, on utilise trois boucles pour faire la multiplication. Donc avec une taille du bloc fixée, il faut ajouter trois autres boucles pour exécuter le calcul à l'intérieur de chaque bloc. On a testé avec différentes dimensions des matrices et différentes tailles de bloc.

#### 3.1.2 Estimateur de $\pi$

On souhaite calculer la valeur assez précise du nombre irrationnel  $\pi$  par l'estimation de Monte-Carlo. Comme montré sur la figure ci-dessous, on génère des points aléatoires dans d'un carré de côté 2. Ensuite on compte le nombre de points à l'intérieur du cercle encastré dans ce carré. Alors le rapport de la surface de ce cercle à celle de ce carré peut être approximé comme :

$$\frac{S_{cercle}}{S_{carré}} = \frac{\pi R^2}{4R^4} \approx \frac{N_{cercle}}{N_{carré}}$$

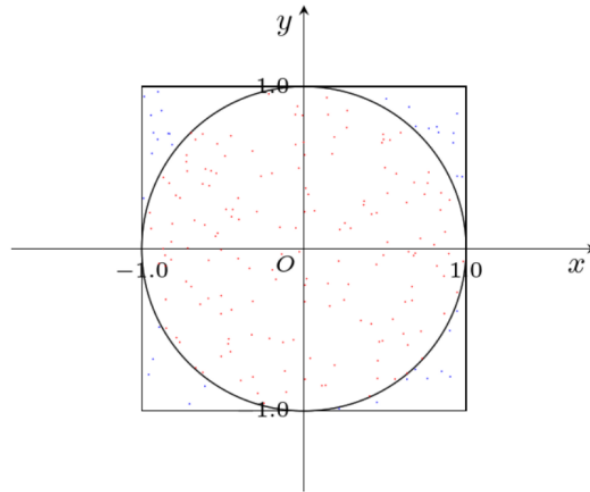


FIGURE 1: Illustration de l'estimation de  $\pi$  par la méthode Monte-Carlo

Donc, la valeur approximative peut être calculée comme :

$$\pi \approx 4 \times \frac{N_{cercle}}{N_{carre}}$$

### 3.1.3 K-means

K-means est un algorithme de clustering qui vise à partitionner un nombre de données en un nombre spécifié de clusters, en fonction de la proximité spatiale des valeurs moyennes. Il y a trois étapes principales :

1. Choisir arbitrairement K centres de clusters.
2. Calculer la distance entre chaque donnée et chaque centre, et assigner les données aux clusters adaptés.
3. Mettre à jour chaque centre selon la moyenne des données dans chaque cluster.

On fait l'itération de ces trois étapes jusqu'à convergence. Mais en pratique, il est mieux de fixer le nombre des itérations afin de pouvoir converger en un temps constant. On commence le programme avec les données à 2 dimensions, après on l'optimise à dimension quelconque afin de s'adapter aux datasets différents. Pour les tests sur PC, on fixe le nombre des itérations à 10.

### 3.1.4 Analyse par Composantes Principales

PCA est un algorithme d'apprentissage automatique qui est utilisé afin de réduire la dimensionnalité au sein d'un ensemble de données tout en conservant autant d'informations que possible. Cette action s'effectue en recherchant un nouvel ensemble de variables appelées composantes, qui constituent les composés des caractéristiques originales décorréliées les unes les autres. Les composantes sont également contraints de telle sorte que le premier composant représente la plus grande variabilité possible dans les données, le deuxième composant la deuxième variabilité la plus importante, et ainsi de suite.

La représentation ci-dessous est un exemple d'application de l'algorithme afin de passer d'un espace 3D à un espace 2D :

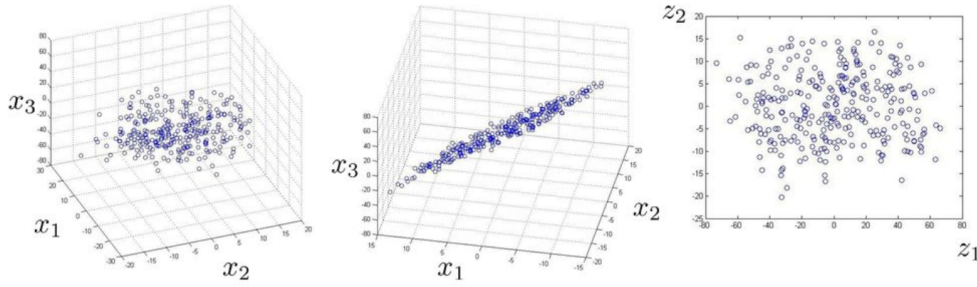


FIGURE 2: Représentation des projections de PCA

### 3.1.5 Coefficient de Pearson

La corrélation est une quantification de la relation linéaire entre des variables continues. Le calcul du coefficient de corrélation de Pearson repose sur le calcul de la covariance entre deux variables continues. Le coefficient de corrélation est ainsi exprimé de la manière suivante :

$$r = \frac{Cov(X, Y)}{\sigma_X \sigma_Y}$$

Avec  $Cov(X, Y)$  la covariance entre les variables  $X$  et  $Y$ , et  $\sigma_X$  et  $\sigma_Y$  les écarts types respectifs de  $X$  et  $Y$ . L'algorithme a été testé avec plusieurs jeux de données avec un nombre d'observations allant de 10 à 20 pour deux variables.

### 3.1.6 Décomposition par valeur singulière

SVD est similaire à PCA. Les deux algorithmes sont utilisés pour la réduction de la dimension de données. Mais avec SVD, on peut obtenir la décomposition de la matrice (données) dans deux directions. Il décompose une matrice  $A$  sous la forme suivante :

$$A = U \cdot \Sigma \cdot V^T$$

avec :

- $A$  : une matrice de dimension  $M \times N$
- $U$  : une matrice de dimension  $M \times M$ , chaque colonne  $U_i$  est un vecteur propre de  $(AA^T)$
- $\Sigma$  : une matrice diagonale de dimension  $M \times N$ , les éléments diagonaux sont  $\sqrt{\lambda_i}$ , avec  $\lambda_i$  la valeur propre correspondante à  $U_i$ . Les éléments sont dans l'ordre décroissant.
- $V^T$  : une matrice de dimension  $N \times N$ , chaque ligne est un vecteur propre de  $(A^T A)$ .

## 3.2 Implémentation et tests des algorithmes sur ARM

Afin d'exécuter les algorithmes sur la carte Zynq, il faut configurer la carte comme le permettent les outils de Vivado. On ne détaillera pas ces configurations (en terme de zone mémoire attribué au code et aux données) sur ce rapport ni les binaires compilés obtenus.

## 3.3 Optimisation des algorithmes sur ARM

En général, il y a trois aspects qui seront pris en compte pour l'optimisation logicielle :

- Délais de branchement
- Utilisation de cache
- Dépendances de données

Maintenant, la plupart des optimisations sont effectuées par le compilateur avec différentes options. Ici pour GNU, il y a quatre niveaux d'optimisation :

- O0 : pas d'optimisation
- O1 : l'optimisation par défaut, qui est faite sur les branchements et les constantes des grandes fonctions qui occupent beaucoup de mémoire
- O2 : l'optimisation au niveau du registre et de l'instruction
- O3 : l'optimisation en déroulant les boucles, utilisant la vectorisation SIMD, faisant `inline` les fonctions, mais prend plus de mémoire et de temps pendant la compilation
- Os : l'optimisation sur la taille du code

Par exemple, BMM est déjà une optimisation pour calculer la multiplication des matrices. Si les blocs sont de taille suffisamment petite pour tenir entièrement dans le cache, il y aura d'échec de cache quelle que soit la méthode de multiplication.

On peut conclure que toutes les options d'optimisation (-O1, -O2, -O3, -Os) par le compilateur peuvent réduire considérablement le temps d'exécution. En comparant la taille du code (segment "text") après les différentes optimisations, on peut voir que la taille du code sera toujours la plus petite si l'on choisit l'option "-Os" qui est spéciale pour l'optimisation de taille. En plus avec "-O3", la taille du code sera plus longue par rapport à d'autres options même si son niveau d'optimisation est le plus haut.

### 3.4 Implémentation sur $\mu$ blaze

Les processeurs Hardcore ont de bonnes performances, mais ne peuvent être utilisés que dans des cartes spécifiques. Les processeurs Softcore (par exemple MicroBlaze ici), en revanche, ont de faibles performances, mais sont très portables sur FPGA.

Le MicroBlaze est un microprocesseur softcore RISC, d'architecture Harvard, entièrement 32 bits. Il présente ainsi 32 registres internes de 32 bits, des bus instructions et données internes, et externes. Le processeur possède 70 options de configuration, parmi lesquelles on trouve : un pipeline (3 ou 5 niveaux), opérateur de division, de décalage, FPU, logique de debug, mémoires cache instructions et données.

L'architecture du cœur Microblaze est représentée par la figure 3 :

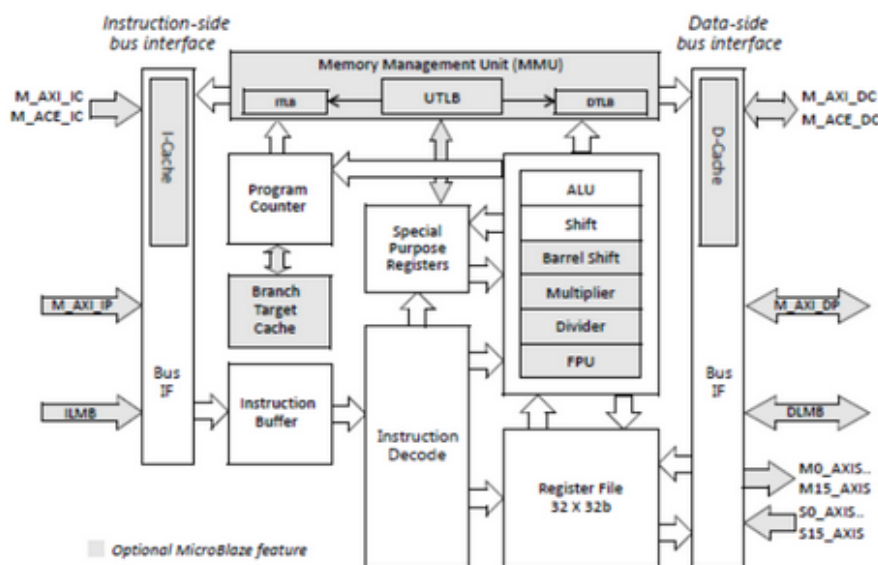


FIGURE 3: Schéma bloc de l'architecture du  $\mu$ blaze



## 4 Conception Synthèse Haut Niveau

Cette conception dite HLS consiste à convertir un code C/C++ en une implémentation RTL d'un code VHDL ou Verilog. La difficulté de cette étape est de comprendre que tout ce qui est en C/C++ n'est pas forcément implémentable ou mal effectué par le trans-compileur. En effet, on tente de passer d'un schéma de flot de contrôle vers un schéma de flot de données. Par exemple, les fonctions récursives sont simplement impossibles à implémenter matériellement. Certains algorithmes itératifs à critères d'arrêt dynamique doivent subir une reconversion. Au final, le but est de générer une IP matérielle dite HLS et qu'elle soit implémentable par l'équipe gérant le matériel.

Dans cette partie, on présente le principe derrière l'outil HLS et quelques-unes des optimisations qu'il propose. Mais aussi, expliquera le raisonnement derrière l'approche de la génération matériellement automatisée.

### 4.1 Conception d'une IP avec directives

Une IP sera encapsulée en elle-même. Ce qui fait que l'implémentation matérielle n'aura aucun contrôle sur ce qui se passe dans l'IP. Afin de pouvoir optimiser l'IP HLS créée, il faut dans un premier temps faire une analyse des dépendances pendant l'exécution de l'algorithme. Cette analyse est faite en 2 temps :

- En premier, automatiquement par Vivado HLS qui possède des outils permettant de détecter des dépendances, d'essayer de les corriger/éliminer ou de suggérer à l'utilisateur d'utiliser les directives proposées.
- En second, l'utilisateur, à l'aide des directives HLS, modifie la synthèse sans modifier le code lui-même.

Les directives les plus intéressantes, pour les algorithmes demandant plusieurs itérations et ayant une implémentation séquentielle tel que Kmeans, sont **FLATTEN**, **PIPELINE** et **UNROLL**.

**Flatten** : Elle consiste à essayer de simplement aplatir la boucle en la rendant en une simple suite d'instructions. Ceci permettra de concevoir un flot de données matériel en RTL. Cette directive n'est possible que si toutes les boucles internes peuvent être aplaties et donc qu'il n'y ait pas de dépendances en données.

**Pipeline** : Applicable aux boucles, fonctions et opérateurs, cette directive divise les opérations en plusieurs étages successifs de conception simple qui à chaque cycle effectuent une opération en 1 cycle. Le pipeline est très efficace pour diminuer la latence des opérateurs mais occupe une plus grande surface sur le FPGA contrairement à une version multi-cycles. Deux aspects importants du pipeline sont : son intervalle d'initialisation (II) qui représente le nombre de cycles après lequel le pipeline peut consommer une donnée ; et la latence i.e. le retard à la sortie.

**Unroll** : Quand un calcul dans une boucle possède des dépendances de données et une latence plus grande que la durée d'une itération, il est conseillé de dérouler la boucle d'un facteur. Par exemple au lieu de traiter les données 1 à 1, on peut traiter n à n. Cette directive peut occuper beaucoup plus de surface FPGA. Il faut donc l'utiliser uniquement quand le nombre d'itération est faible.

Une fois un opérateur/fonction optimisé(e), il faut analyser le nombre d'accès mémoire qu'il effectue en lecture/écriture. Si une mémoire ne fournit qu'un port d'accès, une seule lecture/écriture peut être faite en même temps dans le même cycle (en prenant en compte que

la lecture prend deux cycles selon HLS). Il existe plusieurs stratagèmes pour contrer cela. HLS propose<sup>1</sup> :

**Map** : Permet d'agréger différents blocs mémoires en un seul afin de réduire l'usage des ressources RAM.

**Partition** : Au contraire, cette directive permet de diviser un tableau en plusieurs sous tableau et donc blocs mémoires voir même registre afin d'augmenter le débit lecture/écriture au prix d'une coût matériel plus élevé.

**Reshape** : Permet de redimensionner un bloc mémoire d'un certain facteur de manière entier ou cyclique afin d'avoir un bon compromis entre les deux directives précédentes. Celui-ci est le plus souvent utiliser car il permet d'avoir plusieurs accès mémoire matériels en fonction du nombre de circuits concurrents désirant d'accéder à la même donnée.

## 4.2 Interfaçage d'une IP avec Axi/Axilite

Il est possible d'interfacer une IP matériellement grâce à une panoplie de pointeurs bas niveau. Cette démarche permet d'avoir un contrôle très fin sur les adresses auxquelles le matériel accèdent. Mais par contre, complexifie tellement la gestion matérielle qu'elle devient très chronophage voir impossible pour un humain. Il est donc nécessaire de d'utiliser les interfaces fournis par Xilinx pour la carte ZedBoard, les *Axi* et *Axilite*. Il suffit de définir les ports entrées/sorties de l'IP HLS en tant que maître dans un port Axi et leur données une profondeur (la longueur de la mémoire FIFO). Aussi, il faut définir la fonction *Top-level* comme étant apte à contrôler le bus et donc à recevoir les signaux de synchronisation depuis un CPU (soft ou hard) via l'interface *Axilite*.

Dans les faits, On copie les entrées depuis l'interface du Axi dans des blocs mémoires locaux. Ceci fait que HLS infère des copies en rafale à la place de *memcpy*. Donc, on travaille sur des pointeurs locaux puis on exporte le résultat par Axi output. Il ya donc une abstraction totale de l'interaction mémoire entre une IP HLS et le reste de l'architecture.

## 4.3 Approche d'implémentation des algorithmes

Afin de pourvoir transcrire les algorithmes discutés avec l'équipe logicielle, il d'abord effectuer une analyse de faisabilité en HLS. Car, on ne synthétise pas un algorithme qui ne vaut pas le coût d'être accéléré. Ensuite, on procède à une analyse des besoins matériels pour en déduire les points d'apparition de goulots. Choisir entre la réinterprétation de l'algorithme ou la fragmentation en sous-opérateurs.

Dans la première, on doit réétudier le processus de l'algorithme. Puis, décider si celui-ci est modifiable sans corrompre les résultats ou bien possède une autre implémentation qui puisse satisfaire au mieux les contraintes matérielles. Deux exemples : l'estimation du nombre  $\pi$  par la méthode Monte-Carlo qui ne peut être effectué simplement car la fonction `rand()` n'existe pas en matériel. Il faut alors concevoir un générateur de nombre aléatoire matériel qu'il soit à base d'oscillateur en anneau ou autre. Un deuxième exemple est de modifier le critère d'arrêt de K-means i.e. changer la boucle `while` qui est d'une durée d'exécution indéterminé en boucle `for` qui elle est finie en terme de nombre d'itérations. Plusieurs autres optimisations sont faisables mais il n'y a pas de règles évidentes pour ceci. Il faut soit mener des recherches sur la documentation et les forums ou se fier à son intuition et expérience.

---

1. Xilinx Pragma

Dans la fragmentation en sous-opérateurs, si un algorithme a des parties non dissociables, donc dépendantes au sein d'une boucle, on peut imaginer à le séparer en sous-opérateurs afin d'optimiser ces entités là. Ceci peut offrir un grand gain en performance temporelle mais a deux inconvénients majeurs : Plus de surface par opérateur sera requis à cause de l'implémentation de blocs mémoires "dupliqués" et elle nécessite que la partie développement matériel écrive un protocole d'appel de l'IP plus complexe à développer. Après plusieurs tests, on estime que cette méthode d'optimisation ne sera pas utilisée à cause de contraintes temporelles.

#### 4.3.1 Critères de conception d'IP

Il existe un nombre assez élevé d'optimisation effectuées automatiquement ou manuellement par le logiciel HLS. Un premier facteur limitant lors de la conception est la surface FPGA occupée par l'IP (éléments logiques,DSP,...). Il est simplement impossible d'implémenter une IP qui ne peut pas être portée sur la carte. Donc, il faut limiter l'usage des directive `pipeline` et `unroll` mais aussi les blocs RAM internes. Le *pipeline* possède un critère important qui est l'intervalle d'initialisation. i.e. combien de cycles il y a entre la consommation d'une donnée. Celui-ci est limité par les load/store comem on eput le voir sur la figure 4a, load prend 2 cycles et donc le pipeline est bloqué pend cela. il est difficile de masqué ces chargements et comme pour un CPU, on doit payer le prix soit en temps soi ou bien en surface FPGA.

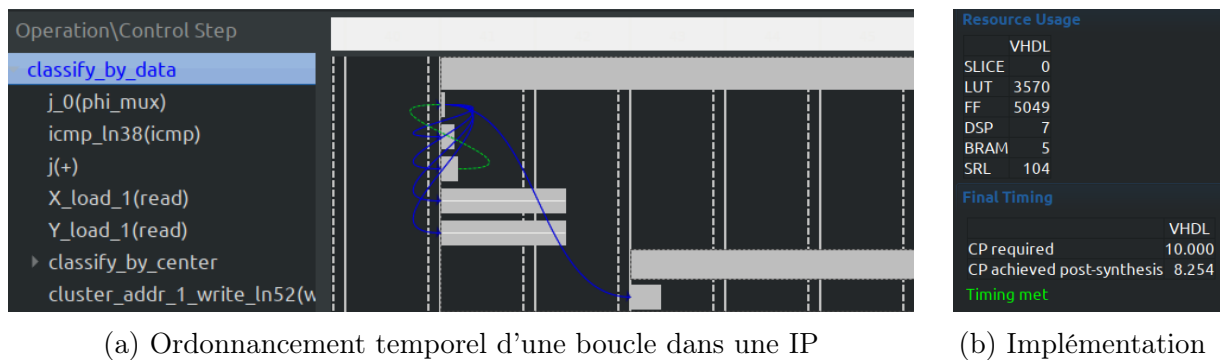


FIGURE 4: Conception de l'IP :

Le deuxième paramètre limitant l'implémentation de l'IP est la fréquence de fonctionnement. HLS prend une période (donc fréquence) objective et "essayera" de synthétiser l'IP demandée à cette fréquence. Il est possible de le forcer à relaxer la synthèse afin de respecter ce critère.

Une solution possible à cette contrainte est d'utiliser des horloges différentes sur la carte qui ce soit par le Zynq ou le **SmartClock** (voir 5.3.2). Les algorithmes SVD et PCA ne sont pas implémentées à cause de la présence de plusieurs séquences de contrôle durs à optimiser dans les contraintes temporelles données. Il est aussi important de noter qu'une IP agit comme un accélérateur matériel non-bloquant pour le CPU. i.e. le CPU est libre pendant tout le temps d'exécution.

La figure 5 montre un résumé de quelques IPs implémentées. Toutes les IP ont pour période de fonctionnement objective : **10 ns**. La figure 4b indique que l'Ip a été synthétisée avec succès et qu'elle respecte les contraintes temporelles imposées.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	4	19	13106	13863	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	2634	-
Register	-	-	1075	-	-
Total	4	19	14181	16497	0
Available	120	8035200	17600	0	0
Utilization (%)	3	23	40	93	0

(a) Pearson

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1186	-
FIFO	-	-	-	-	-
Instance	4	7	3600	5831	-
Memory	3	-	0	0	0
Multiplexer	-	-	-	1070	-
Register	-	-	2236	-	-
Total	7	7	5836	6087	0
Available	120	8035200	17600	0	0
Utilization (%)	5	8	16	45	0

(b) Kmeans

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	7244	-
FIFO	-	-	-	-	-
Instance	4	10	1870	2814	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	2096	-
Register	0	-	6410	224	-
Total	10	10	8280	12378	0
Available	120	8035200	17600	0	0
Utilization (%)	8	12	23	70	0

(c) Matrix Block Mult.

FIGURE 5: Occupation des ressources FPGA pour différentes IP

## 5 Implémentation matérielle

### 5.1 Introduction

Tout l'objet de cette partie est de concevoir une design à l'aide de Vivado pour exécuter les calculs de la partie Software. Elle doit fournir des exemple de matériel pour qu'ils test leur logiciel et est aidée par la partie HLS pour accélérer les calculs. La finalité des design est de permettre d'embarquer les calculs ainsi accélérés pour une utilisation sur par exemple du matériel roulant.

### 5.2 Demo multi-CPU

Une première démo a été conçu pendant le développement des algorithmes jusqu'à l'arrivée des IP HLS. On profite ainsi de l'absence de dépendance avec les autres parties pour explorer des architecture de système.

#### 5.2.1 Concept et problem

Ce design Multi-CPU se propose de juxtaposer aux 2 processeurs ARM embarqués dans le *Zynq*, un softcore *microblaze* implémenté dans la logic programmable (PL/FPGA). L'idée étant de faire tourner des codes sur les 2 processeurs ou d'utiliser le *microblaze* pour du contrôle en utilisant les interruptions.

Un premier design a été réalisé :

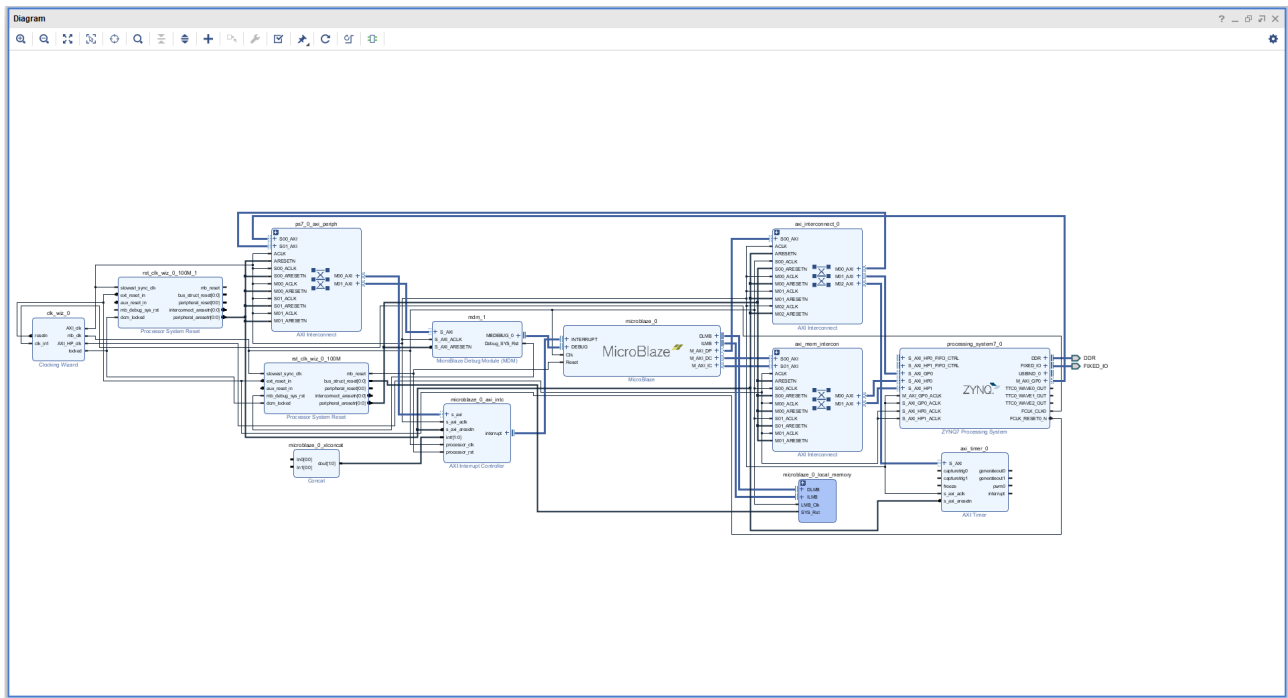


FIGURE 6: Exemple de design pour le Multi CPU pouvant exécuter un programme sur le FPGA et sur le microblaze.

Maintenant que le microblaze est rajouté, il peut être intéressant de pouvoir lui passer des données. Le microblaze est déjà connecté à la RAM du ZYNQ. Donc en théorie, il peut déjà recevoir et envoyer des données. On peut déjà les faire s'exécuter tous les 2 depuis la RAM du ZYNQ (cela permet d'agrandir le heap et le stack pour éviter les dépassement).

### Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

#### Available Memory Regions

Name	Base Address	Size	Add Memory..
microblaze_0_local_memory_ilmb_bram_if_cntlr_Mem_microbl...	0x50	0x1FB0	
ps7_dds_0_HP0_AXI_BASENAME	0x10000000	0x10000000	
ps7_qspi_linear_0	0xFC000000	0x1000000	

#### Stack and Heap Sizes

Stack Size

Heap Size

#### Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_dds_0_HP0_AXI_BASENAME
.init	ps7_dds_0_HP0_AXI_BASENAME
.fini	ps7_dds_0_HP0_AXI_BASENAME
.ctors	ps7_dds_0_HP0_AXI_BASENAME
.dtors	ps7_dds_0_HP0_AXI_BASENAME
.rodata	ps7_dds_0_HP0_AXI_BASENAME
.sdata2	ps7_dds_0_HP0_AXI_BASENAME
.sbss2	ps7_dds_0_HP0_AXI_BASENAME
.data	ps7_dds_0_HP0_AXI_BASENAME
.got	ps7_dds_0_HP0_AXI_BASENAME
.got1	ps7_dds_0_HP0_AXI_BASENAME
.got2	ps7_dds_0_HP0_AXI_BASENAME
.eh_frame	ps7_dds_0_HP0_AXI_BASENAME
.jcr	ps7_dds_0_HP0_AXI_BASENAME
.gcc_except_table	ps7_dds_0_HP0_AXI_BASENAME

FIGURE 7: Linker script du microblaze : exécution depuis la DDR et stack et heap étendus

Une des choses que l'on voudrait pouvoir faire est de passer une grande quantité de données et des instructions au microblaze. Du fait qu'il a accès à la RAM du *Zynq*, il faudrait juste pouvoir lui envoyer des instructions. Si possible dans un emplacement fixe de la mémoire.

### 5.2.2 Utilisation de la BRAM

Pour passer des instructions du *Zynq* au *microblaze*, On décide d'utiliser les BRAM. On crée ainsi un block design qui a en plus du lien avec la DDR du *Zynq*, a aussi des BRAMs. A l'intérieur de celles-ci, on pourra placer des adresses pour les données et des instructions sur ce qu'il faudra en faire.

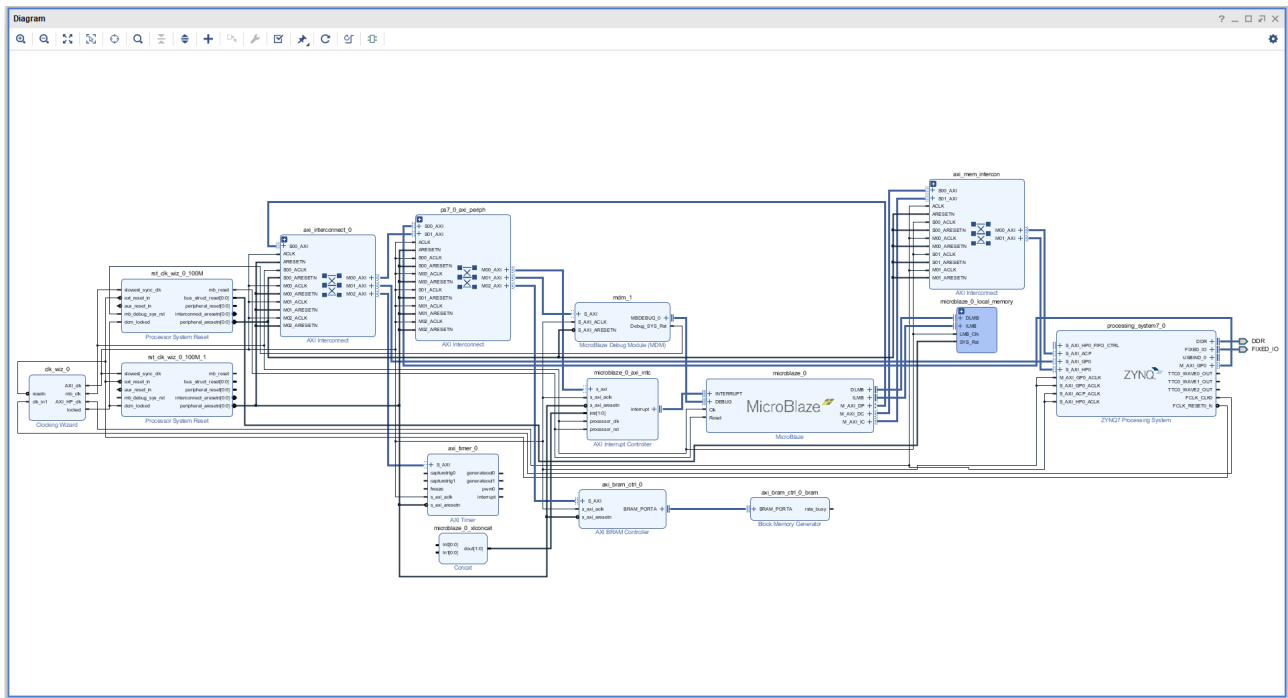


FIGURE 8: Block design pour tester la BRAM :

### 5.2.3 Passage de données vers un microblaze

Une fois cela fait, on a développé 2 applications : une pour le *Zynq* et une pour le *microblaze*.

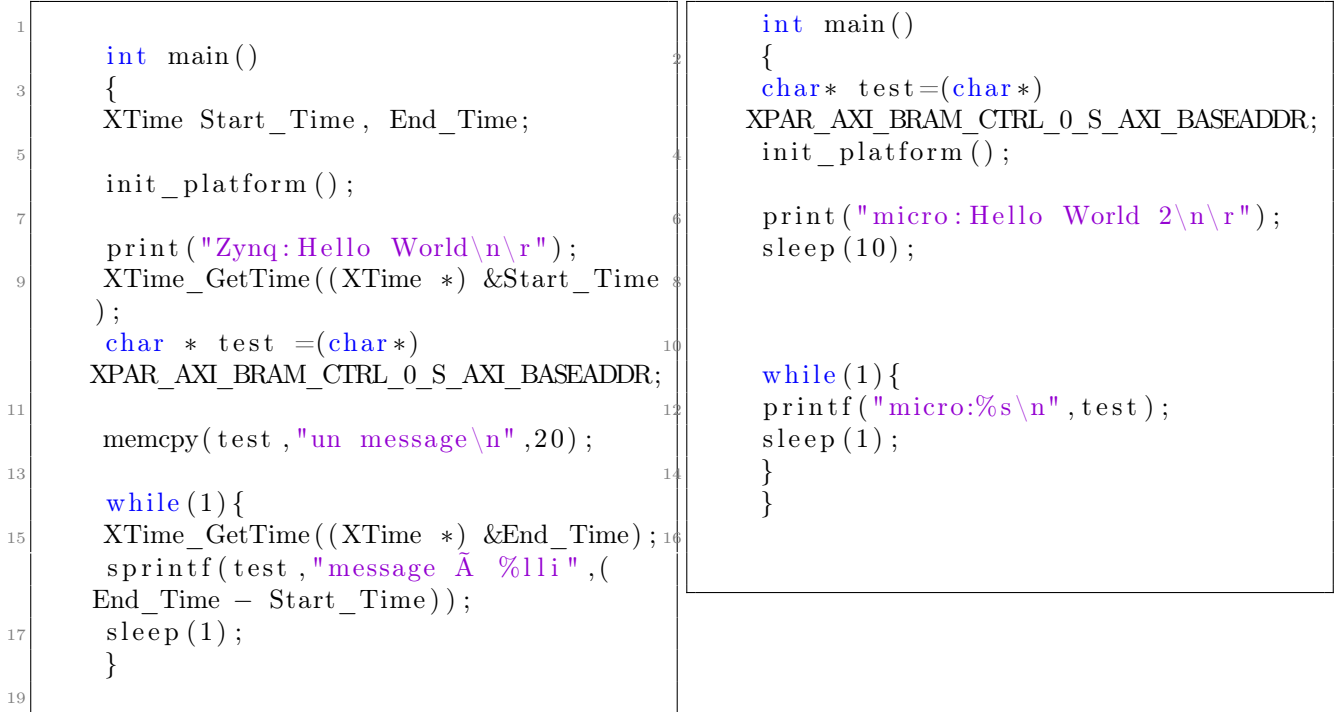
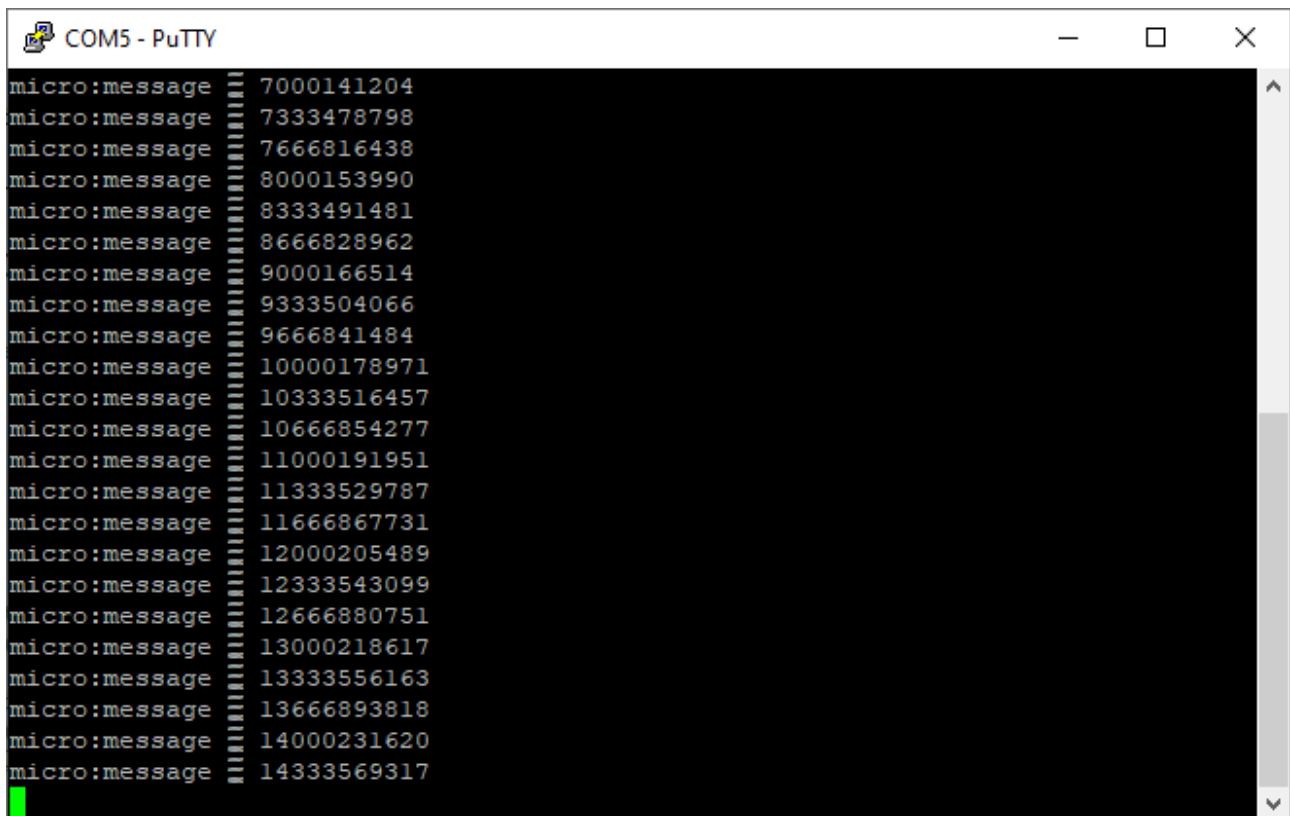


FIGURE 9: Code du microblaze(droite) et du zynq(gauche) pour l'envoi et la réception de message

On peut ainsi facilement passer des adresses du Zynq au microblaze grâce à la BRAM. La

finalité de ce design a été de permettre à l'équipe logicielle de tester leur programme sur la carte. La figure 10 montre l'exécution du codes.



```

micro:message 7000141204
micro:message 7333478798
micro:message 7666816438
micro:message 8000153990
micro:message 8333491481
micro:message 8666828962
micro:message 9000166514
micro:message 9333504066
micro:message 9666841484
micro:message 10000178971
micro:message 10333516457
micro:message 10666854277
micro:message 11000191951
micro:message 11333529787
micro:message 11666867731
micro:message 12000205489
micro:message 12333543099
micro:message 12666880751
micro:message 13000218617
micro:message 13333556163
micro:message 13666893818
micro:message 14000231620
micro:message 14333569317

```

FIGURE 10: Démonstration de la BRAM : le microblaze affiche un message envoyé depuis le Zynq

### 5.3 IP HLS

Dès que la première IP HLS a été finalisée, il a été question de les intégrer dans un design. Donc, un design a été conçu pour les tester.

#### 5.3.1 Concept

Pour accélérer, les programmes que l'on fait tourner sur les *Zynq*, la partie HLS à réaliser des accélérateurs matériels à l'aide des outils *Xilinx*. Ces accélérateurs se présentent comme des IP que l'on peut rajouter dans un bloc design. Il faudra envoyer des données à ces IP pour quelle fasse les calculs à la place du *Zynq* : Beaucoup plus rapidement mais aussi non bloquant pour le CPU.

Ces IP sont commandées par le *Zynq* qui leur donne également des emplacements mémoires comme paramètres. Elles disposent d'accès directe à la mémoire DDR du *Zynq* pour pouvoir charger localement les données et ranger les résultats.

#### 5.3.2 Implémentation et clocking

Une fois les IP mises dans un répertoire communs et que ce répertoire est signalé à *Vivado* comme contenant des IPs, On peut les utiliser dans un block design classique.

**Relier les IPs au Zynq** Les IPs HLS se présentent comme suivant :



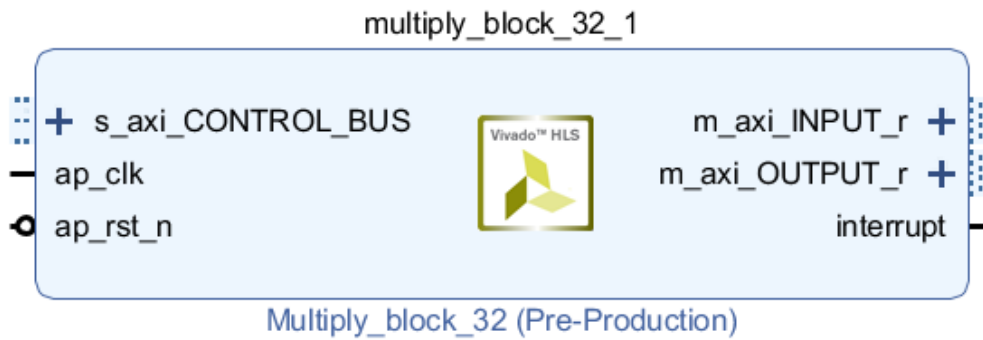


FIGURE 11: Exemple d'IP avec ses différents ports

Elles ont 6 ports :

- Un Port Slave AXILite appelé CONTROL\_BUS qui sert à commander l'IP.
- 2 Ports Master AXI appelés INPUT et OUTPUT qui servent à l'IP pour accéder à des données en mémoire.
- Des ports clk et rst qui permettent à l'IP d'avoir une clock indépendante (on utilisera cela pour avoir de meilleures performances)
- Un port interrupt qui n'a pas été utilisé mais pourrait l'être utilisé avec un microblaze pour déclencher une interruption dès que l'IP finit.

On relie cette IP au Zynq de la manière suivante :

- Les ports INPUT et OUTPUT sont reliés au Slave ACP du ZYNQ pour pouvoir accéder à ces mémoires et avoir la cohérence des caches.(figure 12).
- On relie les ports CONTROL\_BUS au Master GP du Zynq pour pouvoir envoyer des commandes (les commandes étant moins critiques, elles ne passent pas par les interfaces rapides )

Page 17 sur 23

On a fait en sorte dans le bock design que les clocks des IP et la clock du HLS soit indépendantes. Cela permettra de tirer le plus de performance des IP plus tard.

**Fonction pour le SDK** A partir de là, après synthèse, placement et routage du design, on dispose d'un entête spécial par exemple :

```
#include "xmultiply_block_64.h"
```

qui va permettre d'interagir simplement avec une IPS HLS.

Ainsi le code suivant contient 2 fonctions qui permettent d'initialiser une IP HLS(ici *mul64*) et de l'utiliser pour faire un calcul.

```
1 //fonction d'initilisation de L'IP
void init_multiply_block_ip(XMultiply_block_64* mb,XMultiply_block_64_Config*
    mb_c){
3 int status=XMultiply_block_64_CfgInitialize(mb,mb_c);
  XMultiply_block_64_DisableAutoRestart(mb);
5 XMultiply_block_64_InterruptGlobalDisable(mb);
  XMultiply_block_64_InterruptDisable(mb, 1);
7 if(status!=XST_SUCCESS){
  printf("Multiply Block: init_failed \r\n");
9 }
  printf("idle=%lx,ready=%lx,done=%lx\n",XMultiply_block_64_IsIdle(mb),
    XMultiply_block_64_IsReady(mb),XMultiply_block_64_IsDone(mb));
11 printf("succes\n");
  }
13
//fonction de lancement du calcul sur l'IP
15 void multiply_block_hw_call(XMultiply_block_64* mb_p, float* mA, float* mB, float
    * result){
17 //on charge les adresse des donnÃ©es et les sorties
  XMultiply_block_64_Set_in_mA(mb_p, (u32)mA);
19 XMultiply_block_64_Set_in_mB(mb_p, (u32)mB);
  XMultiply_block_64_Set_out_mC(mb_p, (u32)result);
21
//on attend d'Ãªtre prÃªts.
23 while(!XMultiply_block_64_IsReady(mb_p));
25
//on lance
  XMultiply_block_64_Start(mb_p);
27
//on attend d'avoir fini
29 while(!XMultiply_block_64_IsDone(mb_p)){
31 }
  //les rÃ©sultat sont dÃ©jÃ rangÃ©s donc on a fini.
33 return;
35 }
```

Grâce à l'utilisation des interfaces ACP et du fait qu'on ait activé le "*tie off AxUser*", nous n'avons pas besoin de vider le cache puisque la cohérence est maintenue à travers l'AXI et malgré les modifications des données faites par l'IP.

**Optimisation des clock avec les Timings** On peut maintenant chercher à avoir le plus de performances possible avec nos IPs. Pour cela on va utiliser le fait qu’elles aient des clocks indépendantes de celle de l’AXI. On peut ainsi changer leur fréquence séparément du reste du système.

Grâce au *timing report*, on peut connaître après implémentation quelle est la marge que l’on a par rapport au temps critique fixé par la clock. On peut ainsi corriger cette clock itérativement pour aller jusqu’au moment où l’on n’a plus aucune marge.

On a put par exemple monter la clock de l’IP *mul64* de 100 à 130 MHZ.

The screenshot shows the 'Timing' tool interface. The left sidebar lists various timing categories, with 'Intra-Clock Paths' expanded. The main panel displays 'Intra-Clock Paths - HLS\_CLK\_design\_IP\_clk\_wiz\_0\_0'. It shows the clock name 'HLS\_CLK\_design\_IP\_clk\_wiz\_0\_0' and a table of statistics for different path types.

Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	0,144 ns	0,000 ns	0	21220
Hold	0,019 ns	0,000 ns	0	21220
Pulse Width	2,750 ns	0,000 ns	0	9133

FIGURE 14: Écran du timing report permettant de connaître le clock skew

### 5.3.3 Performances

## 5.4 Amélioration et future design

Il s’agirait ensuite de construire un design utilisant et les IP et le *microblaze* pour pouvoir, par exemple utiliser l’IP HLS de block matrix multiplication. Pour faire des multiplication de matrice plus grande en utilisant plusieurs fois l’IP sans avoir besoin de monopoliser le *Zynq*, utilisant ainsi les interruption sur le *microblaze*.

## 6 Performances Mesurées

Dans cette section, on présente les résultats obtenu et validé pour les exécution sur PC standard, ARM, ARM+HLS et *μblaze*.

### 6.1 Performances ARM

L’exécution sur ARM donne les résultats montré en figure 15. On retrouve bien la complexité algorithmique théoriques et que ces algorithmes sont proportionnels à la taille des données sauf pour Bloc-Matrix-Multiplication. qui lui dépend du formatage et du parcours (ijk) de ces données.

<b>Bloc-Matrix-Multiplication</b>	<b>M</b>	<b>P</b>	<b>N</b>	<b>bloysize</b>	<b>nb de cycles</b>	<b>temps (µs)</b>
	40	25	30	3	1694524	5088. 66066
	250	470	316	3	2045606949	6142963. 81081
	250	470	316	7	1740020211	5225285. 91892
	1000	500	200	15	4428594020	13299081. 14114
<b>PiEstimator</b>	<b>nb d'itération</b>				<b>nb de cycles</b>	<b>temps (µs)</b>
	1000				69204	207. 81982
	10000				674408	2025. 24925
	100000				6867433	20622. 92192
	1000000				68545278	205841. 67586
<b>K-means</b>	<b>K (clusters)</b>	<b>D (dimension)</b>	<b>N (nb données)</b>	<b>Itération</b>	<b>nb de cycles</b>	<b>temps (µs)</b>
	3	4	150	20	3729634	11200. 10210
	20	2	3000	20	265058761	795972. 25526
	16	32	1024	20	860685779	2584641. 97898
<b>Pearson</b>	<b>row (nb données)</b>		<b>col (dimension)</b>		<b>nb de cycles</b>	<b>temps (µs)</b>
	10		2		20869	62. 66976
	50		4		244682	734. 78078
	200		10		3627683	10893. 94294
<b>PCA</b>	<b>ligne</b>		<b>colonne</b>		<b>nb de cycles</b>	<b>temps (µs)</b>
	10		2		5680	17. 05706
	50		4		52591	157. 93093
	200		10		2992348	8986. 03003
<b>SVD</b>	<b>ligne</b>		<b>colonne</b>		<b>nb de cycles</b>	<b>temps (µs)</b>
	13		7		215159	646. 12312
	30		35		6042542	18145. 77177
	100		200		666164879	2000495. 13213

FIGURE 15: Temps d'exécution sur ARM des différents algorithmes

On voit aussi le lien entre optimisations de compilation et performance en terme de taille de code et temps sur la figure 16 et 17.

<b>Bloc-Matrix-Multiplication</b>	<b>ordre</b>	<b>optimisation</b>	<b>text</b>
	ijk	-O0	<b>58716</b>
	ikj	-O0	<b>58716</b>
	ikj	-O1	<b>58412</b>
	ikj	-O2	<b>58328</b>
	ikj	-O3	<b>59352</b>
	ikj	-Os	<b>58260</b>
<b>K-means</b>		<b>optimisation</b>	<b>text</b>
		-O0	<b>63104</b>
		-O1	<b>58792</b>
		-O2	<b>58764</b>
		-O3	<b>59324</b>
		-Os	<b>58628</b>
<b>Pearson</b>		<b>optimisation</b>	<b>text</b>
		-O0	<b>59244</b>
		-O1	<b>58548</b>
		-O2	<b>58560</b>
		-O3	<b>61400</b>
		-Os	<b>58436</b>

FIGURE 16: Un exemple des tailles de codes produit par l'optimisation

<b>M</b>	<b>N</b>	<b>P</b>	<b>bloysize</b>	<b>cycles</b>	<b>temps (μs)</b>
1000	500	200	15	<b>4428594020</b>	13299081.14114
1000	500	200	15	<b>3822785564</b>	<b>11479836.52853</b>
1000	500	200	15	<b>531048961</b>	<b>1594741.62462</b>
1000	500	200	15	<b>564442618</b>	<b>1695022.87688</b>
1000	500	200	15	<b>545292506</b>	<b>1637515.03303</b>
1000	500	200	15	<b>809730905</b>	<b>2431624.33934</b>
<b>K</b>	<b>D</b>	<b>N</b>	<b>Itération</b>	<b>cycles</b>	<b>temps (μs)</b>
16	32	1024	20	<b>860685779</b>	<b>2584641.97898</b>
16	32	1024	20	<b>123736936</b>	<b>371582.39039</b>
16	32	1024	20	<b>138956312</b>	<b>417286.22222</b>
16	32	1024	20	<b>139265209</b>	<b>418213.84084</b>
16	32	1024	20	<b>149756615</b>	<b>449719.56456</b>
<b>row (nb données)</b>		<b>col (dimension)</b>		<b>cycles</b>	<b>temps (μs)</b>
200		10		<b>3627683</b>	<b>10893.94294</b>
200		10		<b>695048</b>	<b>2087.23123</b>
200		10		<b>688131</b>	<b>2066.45946</b>
200		10		<b>617540</b>	<b>1854.47447</b>
200		10		<b>805517</b>	<b>2418.96997</b>

FIGURE 17: Un exemple des temps d'exécution des codes produit par l'optimisation

## 6.2 Performances HLS

En mesurant le temps grâce au timer du *Zynq*, on peut évaluer les performance des IPs. Les IPs évaluées sont :

- Block matrix multiplication 32 par 32(*mul32*) et 64 par 64(*mul64*)
- Le coefficient de Pearson (*pearson*)
- L'algorithme kmeans (*kmeans*)

On compare ainsi les temps Software, Hardware et Hardware avec clock amélioré :

exemple d'IP	temps HW (μs) @100MHZ	temps SW (μs)	fréquence améliorée	temps HW amélioré(μs)
mul64	10280.17417	26132.32432	130 MHZ	7042.11712 useconds
mul32	2411.54655	3281.83483	125MHZ	1809.89489
pearson	32.56906	5.25826	115 MHZ	21.19820
kmeans	1297.41742	1527.90390	120MHZ	1082.07808

On a aussi essayé de serrer encore plus les timings en jouant sur la Clock des AXI et sur l'algorithme de placement (Haute performance). Avec les réglages suivants :

- AXI\_CLK=180MHZ
- HLS\_CLK=131MHZ

On a un temps de 6215.74174 μs soit plus de 4 fois moins de temps que la version SW classique.

## 6.3 Performances PC

Pour comparaison, la figure 18 donne les performances en temps d'exécution pour différents paramètres des algorithmes.

BMM	M	P	N	blocksize	temps (us)
	40	25	30	3	73
	250	470	316	3	97875
	250	470	316	7	70285
	1000	500	200	15	172542
Pi Estimator	nb d'itérations				
	1000				77
	10000				467
	100000				4773
	1000000				44501
K-Means	K (clusters)	D (dimension)	N (nb données)	Itération	
	3	4	150	20	9577
	20	2	3000	20	30049
	16	32	1024	20	92520
Pearson	row				col
	10				2
					62
PCA	ligne				colonne
	10				2
					72
SVD	ligne				colonne
	13				7
					6
	30				35
					44
	100				200
					978

FIGURE 18: Temps d'exécution sur PC

Il est évident que ces performances seront meilleures car les PC modernes sont multicœurs et sont cadencées à 3 GHz au minimum avec des optimisations de toutes sortes (exécution dans le désordre, prédiction des branchements, pipelines profonds, cache multi-niveaux...).

## 7 Conclusion et perspectives

Ce projet aura permis de découvrir une architecture hétérogène CPU+FPGA ainsi que les outils de développement qui lui dédié. On a pu implémenté les algorithmes de la série SPARK. L'exploration des possibilité et le choix des algorithmes étaient une étape assez délicate qui demandait l'accord de tous les membres de l'équipe. Ces implémentation en C on étaient exécutées sur PC et ARM avec des plus tentatives d'implémentation sur  $\mu$ blaze mais rendu impossible dans les délais déterminés à cause de problèmes au niveau des programmes de développement.

Certaines IPs HLS conçues montrent une inconsistance/incohérence lors de leurs exécutions ainsi que des différences de résultats, comme pour Kmeans, à cause des différentes normes utilisées et des arrondis. L'interfaçage de ces IPs est correctement assuré par la partie matérielle. On a pu montrer qu'une simple implémentation non optimisée d'une IP HLS est au minimum 4 fois plus rapide, sur la série d'algorithmes que l'on a fait, que l'exécution sur CPU. Il est aussi possible de cadencer les différentes IP du systèmes avec différentes horloges. Cette possibilité rend le système plus hétérogène qu'avant, le complexifiant certes, mais améliorant les performances en termes de temps. Il est aussi à constater que l'usage des IPs consomme plus de ressources s'elles ne sont pas agrégées. i.e. Il est souvent plus performant d'avoir une seule IP qui gèrent toutes l'implémentation RTL si les communications ne sont pas très fréquentes. Dans le cas contraire ceci saturerait l'interface de communication AXI. Enfin, la limite matérielle sur le FPGA reste le plafond physique pour ce genre de solution.

De manière similaire au projet, l'équipe est hétérogène et demandait un certain niveau de coopération. Plusieurs difficultés, concernant les logiciels de développement et un manque de performances des PCs utilisés, ont entravé le bon déroulement du projet. Il faut aussi noté que, malgré tous les efforts mis dans la coopération et communication mais aussi la séparation des

tâches, on a rencontré quelques conflits de point de vu. Ceci est dû aux différentes habitudes de chaque membre de l'équipe.

En perspectives, on prévoit :

- **Implémentation sur  $\mu$ blaze** : On souhaite terminer l'implémentation logicielle des algorithmes sur un  $\mu$ blaze afin de comparer les performances à celles du ARM et de l'IP HLS. Ensuite, on envisage passer à une version multi- $\mu$ -blazes où chacun des CPU exécute soit une partie du calcul ou un autre algorithme.
- **HLS multi-IPs** : même si théoriquement une implémentation multi-IPs de sous-opérateurs semble moins bonne qu'une implémentation d'une IP unique, on souhaite effectuer plusieurs tests sur ces modèles afin d'avoir une base de comparaison.
- **$\mu$ blaze - HLS** On pense aussi avoir une implémentation multi- $\mu$ blaze où chacun serait capable de contrôler une ou plusieurs IP HLS. Ceci répartirait la charge du calcul sur les  $\mu$ blazes et leur IPs mais risque de mener à un encombrement en communication.