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 : **Quelqu'un**

Master 2 : Système Embarqué et Traitement de l'Information

Année : 2019-2020

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 *Xilinix*, 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 μblaze. 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 μblaze.

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 μblaze.

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.

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 μblaze. 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 π

On souhaite calculer la valeur assez précise du nombre irrationnel π 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 :

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

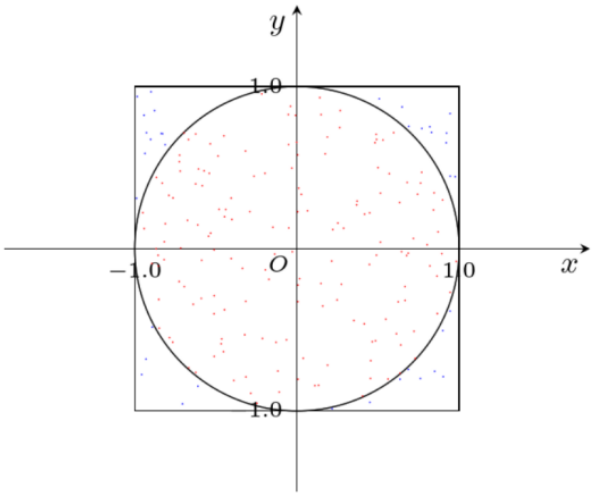

FIGURE 1: Illustration de l'estimation de π par la méthode Monte-Carlo

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 quelque 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 décorrélées les unes les autres. Les composants 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 :

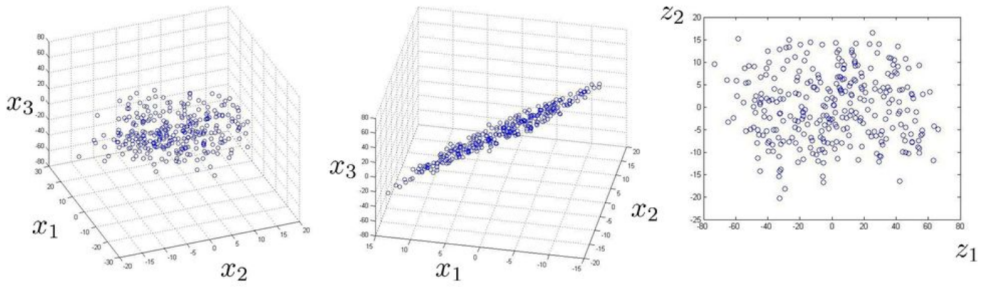

FIGURE 2: Représentation des projections de PCA

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 σ_X et σ_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 :

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

avec :

- A : une matrice de dimension MxN
- U : une matrice de dimension MxM, chaque colonne Ui est un vecteur propre de (AA^T)
- Σ : une matrice diagonale de dimension MxN, les éléments diagonaux sont $\sqrt(\lambda_i)$, avec λ_i la valeur propre correspondante à U_i . Les éléments sont dans l'ordre décroissant.
- V^T : une matrice de dimension NxN, chaque ligne est un vecteur propre de (A^TA).

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 μblaze

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 débug, mémoires cache instructions et données.

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

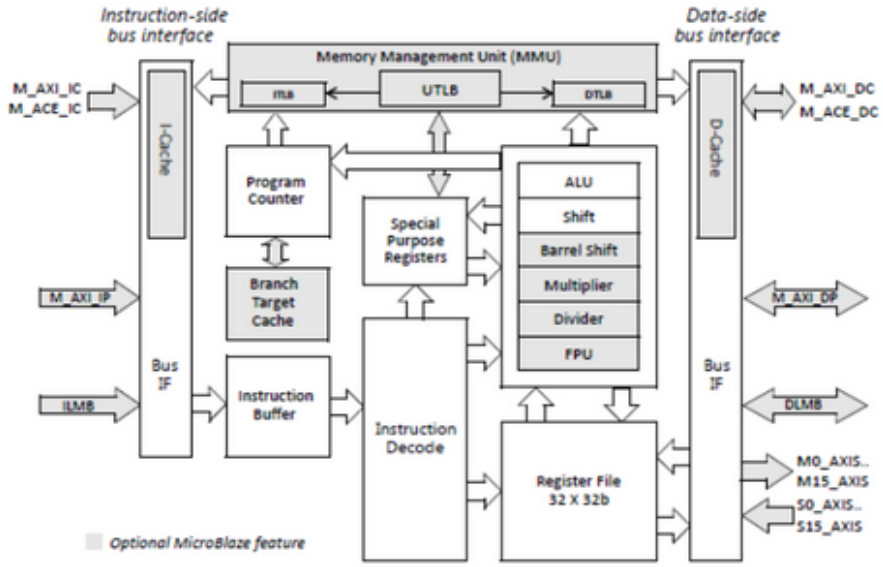

FIGURE 3: Schéma bloc de l'architecture du μblaze

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 forcement implémentable ou mal effectué par le trans-compilateur. En effet, on tente de passer d'un schéma de flot de contrôle vers un schéma de flot de données. Par exemple, le fonctions récursives sont simplement impossibles à implémenter matériellement. Certains algorithmes itératif à critères d'arrêt dynamique doivent subir une reconversion. Au final, le but est de générer une IP matérielle dite HLS et quelle soit implémentable par l'équipe gérant le matériel.

Dans cette partie, on présente le principe dernière 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 donnée 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 important 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 fait 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 [1] :

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

1. Xilinx Pragma

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 π 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.

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 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). La figure 4 montre un résumé de quelques IPs implémentées. Toutes les IP ont pour période de fonctionnement objective : **10** ns.

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	80	35200	17600	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	8087	0
Available	120	80	35200	17600	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	80	35200	17600	0
Utilization (%)	8	12	23	70	0

(c) Matrix Block Mult.

FIGURE 4: Occupation des ressources FPGA pour différentes IP

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.

5 Implémentation matérielle

Multi-horloge

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 5. 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 de ces données.

Bloc-Matrix-Multipication	**M**	**P**	**N**	**blocsize**	**nb de cycles**	**temps (µs)**
	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
PiEstimator	**nb d'itération**				**nb de cycles**	**temps (µs)**
	1000				69204	207.81982
	10000				674408	2025.24925
	100000				6867433	20622.92192
	1000000				68545278	205841.67586
K-means	**K (clusters)**	**D (dimension)**	**N (nb données)**	**Itération**	**nb de cycles**	**temps (µs)**
	3	4	150	20	3729634	11200.10210
	20	2	3000	20	265058761	795972.25526
	16	32	1024	20	860685779	2584641.97898
Pearson	**row (nb données)**		**col (dimension)**		**nb de cycles**	**temps (µs)**
	10		2		20869	62.66976
	50		4		244682	734.78078
	200		10		3627683	10893.94294
PCA	**ligne**		**colonne**		**nb de cycles**	**temps (µs)**
	10		2		5680	17.05706
	50		4		52591	157.93093
	200		10		2992348	8986.03003
SVD	**ligne**		**colonne**		**nb de cycles**	**temps (µs)**
	13		7		215159	646.12312
	30		35		6042542	18145.77177
	100		200		666164879	2000495.13213

FIGURE 5: Temps d'exécution sur ARM des différents algorithmes

6.2 Performances HLS

Les exécutions utilisant les 3 IP réalisées, on obtient les temps d'exécution suivants :

7 Conclusion et perspectives

Ce projet aura permis de découvrir l'architecture hétérogène CPU+FPGA