Rapport de projet

PR310

3ème année

Algorithmes de tri sur cible embarquée

Auteurs
Denis Shemonaev
Romain Beaubois
Vincent Dubreuil
Mathieu Pillet

Professeur encadrant
Jérémie Crenne
Bertrand Legal

Table des matières

1 Introduction **2**

2 Cahier des charges **2**
2.1 Algorithmes 2
2.2 Cibles 4
2.3 Processus de test 4
2.4 Comparaison des solutions 5

3 Solution CPU **5**
3.1 Environnement de test et de développement 5
3.2 x86 Intel i7 6
3.3 ARM Cortex A9 7
3.4 Comparaison 8

4 Solution FPGA **9**
4.1 Environnement de test et de développement 9
4.2 Protocole de communication AXI 10
4.3 Zynq 7020 11

5 Solution GPU **15**
5.1 Tri radix parallélisé 15
5.2 Tri mixte CPU/GPU 16
5.3 Comparaison 16

6 Comparaison des solutions **17**

7 Conclusion **18**

1 Introduction

Ce rapport rend compte du travail effectué dans le cadre du module PR310 qui correspond au projet avancé de l'option de spécialisation systèmes embarqués. Il traite de l'étude de tris de données et plus précisément à l'étude de différentes solutions de tri et notamment les solutions de tri parallélisables. Le but de ce projet était avant tout d'explorer les différentes solutions possibles avec un regard critique et de les comparer entre elles ainsi qu'à des référentiels comme les fonctions proposées par les bibliothèques standard. Le point de départ est donc la définition d'un cahier des charges qui permettra de définir les conditions de tests ainsi que les solutions étudiées mais surtout de poser des limites à ce sujet qui est très vaste. Le présent rapport détaillera donc les différentes étapes et réalisations de ce projet en débutant par la définition du cahier des charges, suivie des solutions développées et de leurs résultats et enfin une comparaison de l'ensemble.

2 Cahier des charges

2.1 Algorithmes

Bubble

Le tri à bulle est un des tris les plus simples algorithmiquement parlant étant donné qu'il revient à comparer deux éléments voisins et à les inverser si le critère de tri n'est pas respecté. C'est donc un tri quadratique qui se révèle efficace uniquement pour de petits ensembles de données.

- Algorithme de tri simple
- Complexité temporelle en $O(n^2)$
- En place

Selection

Le tri par sélection, quadratique également, est un autre algorithme naïf qui trie les données en les sélectionnant du minimum au maximum.

- Algorithme de comparaison
- Utilisation de deux sous tableaux
- Complexité temporelle moyenne en $O(n^2)$
- En place

Insertion

Le tri par insertion trie les données en les insérant une à une dans un tableau vide ou déjà trié. Il est également quadratique, ce qui le rend inefficace pour un nombre important de données.

- Algorithme de comparaison
- Complexité temporelle moyenne en $O(n^2)$
- En place

Quick

Ce tri est basé sur la méthode récursive "diviser pour régner", même s'il peut être implémenté en itératif. Sa simplicité est son efficacité en fait une très bonne solution pour trier des ensembles volumineux de données.

- Algorithme de comparaison
- Complexité temporelle moyenne en O(n log(n))
- Pire complexité temporelle en $O(n^2)$
- Complexité spatiale moyenne en O(log(n))

Tim

Le tri Tim est un algorithme de tri récent qui se veut être plus adapté aux données réelles qui sont souvent partiellement triées. Il se base sur l'utilisation de deux algorithmes afin d'exécuter plusieurs tris en même temps sur des ensembles réduits qui auraient donc une chance d'être déjà partiellement triées voire complètement triées, ce qui correspond à un cas favorable pour le tri par insertion. L'algorithme de merge étant ainsi là pour ensuite fusionner les résultats des différents tris réalisés.

- Algorithme hybride (merge et insertion)
- Adapté aux données avec portions triées
- Complexité temporelle moyenne en O(n log(n))

Radix

Ce tri se démarque par l'absence de comparaison, le radix trie en balayant successivement les chiffres qui composent un nombre pour le trier.

- Algorithme de distribution
- Complexité temporelle moyenne en O(n log(n))
- Complexité spatiale moyenne O(n)

Bitonique

Le tri bitonique se démarque par sa structure : c'est un réseau de tri fortement parallélisable. En effet, à chaque étape "séquentielle" du tri, tous les éléments sont comparés paire a paire. Si les ressources spatiales et matérielles étaient illimitées, ce réseau ne nécessiterait que $O(\log_2^2 n)$ étapes pour trier n valeurs. Cependant, cet avantage est contre-balancé par un fort coût spatial afin de déployer l'intégralité du réseau : il est nécessaire d'accéder aux n éléments en parallèle ainsi que de disposer d'assez de comparateurs.

- Réseau de tri
- Complexité temporelle moyenne en $O(\log^2(n))$
- Complexité spatiale moyenne $O(n \log^2(n))$
- Peu flexible

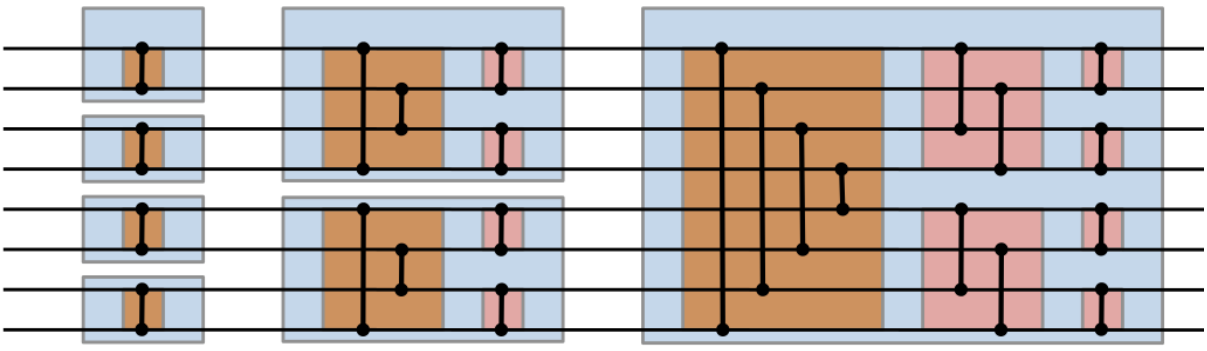

FIGURE 2.1 – Réseau de tri bitonique à 8 entrées

2.2 Cibles

Les algorithmes d'études étant définis, on peut effectuer le choix des cibles selon des critères de sélection jugés pertinents.

2.2.1 Critères de sélection

La liste des critères de sélection des cibles est donc la suivante.

- **Accessibilité** : facilité à se procurer la cible.
- **Technologies et architectures différentes** : FPGA, CPU (ARM, x86, MIPS, RISC) ou GPU.
- **Familiarité avec la cible** : connaissance de la cible pour un gain en temps de développement.

2.2.2 Cibles retenues

Les cibles alors retenues sont donc les suivantes :

- **CPU** : ARM Cortex A9, x86 Intel i7
- **FPGA** : Zynq 7020
- **GPU** : Nvidia Jetson TK1

Le processeur ARM Cortex A9 ainsi que le Zynq 7020 ont pour support la carte Pynq Z2 de chez Xilinx qui implémente également une connexion entre le CPU et le FPGA à l'aide d'un bus AXI. Le processeur x86 Intel i7 est associé à un ordinateur portable récent. Le GPU se trouve quant à lui intégré sur une carte d'un kit de développement Nvidia que nous avons déjà utilisée au cours du module HPEC cette année.

2.3 Processus de test

Le processus de test a pour but de donner des résultats fiables, réalistes et représentatifs. Pour cela, on choisit tout d'abord de faire des tests sur un nombre de données à trier différent afin de faire apparaître la complexité temporelle des algorithmes et pouvoir comparer ce critère. Pour avoir faire apparaître le pire et le meilleur cas des algorithmes, on choisit de changer les données testées plusieurs fois. Et enfin, pour avoir un résultat plus précis sur le temps moyen de tri, on choisit de lancer plusieurs fois le même tri pour le même jeu de données. De plus, ce dernier critère permet de limiter l'écart entre les différentes mesures que l'on peut avoir avec la cible x86 à cause des appels systèmes de l'OS liés au support ou encore à la l'utilisation du turbo boost du processeur. Le schéma suivant illustre le procédé de test adopté.

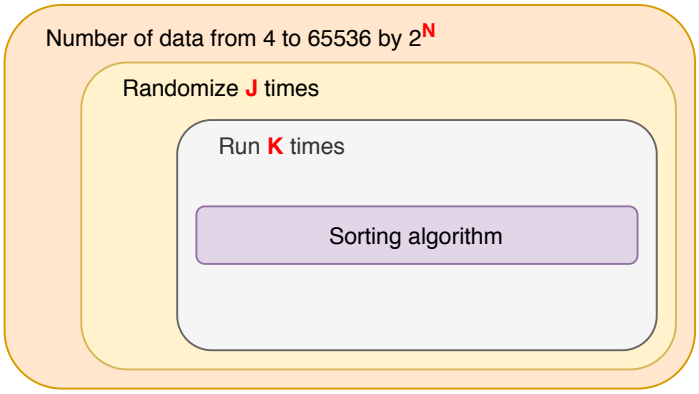

FIGURE 2.2 – Procédé de test des algorithmes

Ainsi, pour la suite de ce rapport, on prendra compte des définitions suivante :

- NB_DATA : nombre de données testées
- AVERAGING : nombre de répétitions du test pour un algorithme avec le même jeu de données
- RANDOM SETS : nombre de changements de jeux de données pour un algorithme

2.4 Comparaison des solutions

Afin de pouvoir comparer les différentes solutions il est nécessaire d'avoir des critères représentatifs et cohérents par rapport aux systèmes embarqués. Les critères retenus sont les suivants.

- Consommation moyenne de la cible
- Utilisation des ressources disponibles
- Temps de calcul
- Flexibilité/portabilité

3 Solution CPU

Cette partie aborde le protocole de test et les résultats obtenus sur des cibles CPU ARM et x86. Pour le processeur x86, un ordinateur récent de la marque MSI à été utilisé tandis que le processeur ARM est celui présent sur la Pynq.

3.1 Environnement de test et de développement

Afin de tester les algorithmes, un wrapper en C++ à été mis en place pour une raison de simplicité. Une classe `Comparator` est utilisée pour chronométrer le temps mis par un algorithme pour trier les données avec la méthode `process`. Une méthode `check` permet de vérifier que les données obtenues en sortie sont bien triées.

C++

```
class Comparator {
private:
    virtual unsigned int* sort(unsigned int data[], int len) = 0;
public:
    int process(unsigned int data[], int len);
    void check(unsigned int data[], int len);
};
```

Le protocole de test explicité dans la partie 2.3 est implémenté par le module `benchmark`. Celui-ci déroule le protocole avec l'aide de trois fonctions :

- `randomize` qui permet de mélanger un tableau de données.
- `print_progress` qui affiche une barre de progression du test.
- `runBenchmark` qui lance la procédure de test et inscrit les résultats dans un fichier CSV.

Pour rendre le test plus ludique et plus pratique, un script Bash permet d'implémenter une interface en ligne de commande simple (voir figure 3.1) qui permet de choisir l'algorithme à tester puis affiche le résultat avec des courbes en utilisant gnuplot.

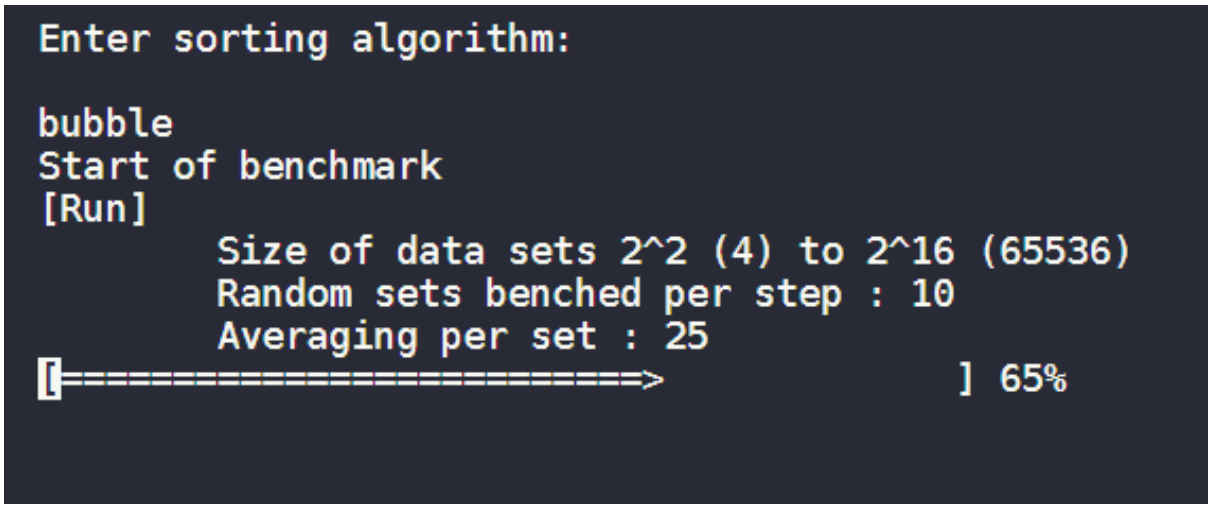

FIGURE 3.1 – Interface utilisateur

3.2 x86 Intel i7

Le tableau ci-dessous donne les temps de tri en µs obtenus pour les différentes algorithmes avec le CPU x86 de chez Intel.

	NB_DATA = 256			NB_DATA = 8192			NB_DATA = 65536		
Algorithme	Moy	Max	Min	Moy	Max	Min	Moy	Max	Min
Radix	3	23	3	75	138	66	**538**	1,092	519
Tim	**<1**	25	<1	**65**	529	48	623	3,758	482
Quick	2	52	2	161	825	139	1,528	3,987	1,406
Bitonique	**57**	531	47	272	915	236	1,357	17,507	1,106
libc qsort	21	110	15	310	1,403	257	2,961	14,933	2,267
Insertion	**<1**	3	<1	274	7,335	7	**16,135**	**406,540**	**57**
Selection	18	42	17	15,085	16,123	15,001	973,225	1,013,822	960,291
Bubble	21	49	20	**17,491**	57,469	14,718	**1,211,633**	4,598,438	1,016,124

Tableau 1 – Tableau récapitulatif des résultats obtenus sur CPU x86 Intel i7-8565U, 1,8 GHz (standard), 4,6 GHz (turbo), 4 coeurs physiques, cache 8 Mo

Ces premiers résultats nous permettent de faire un premier constat sur l'impact du choix de l'algorithme. En effet, on voit bien ici qu'il n'y a pas un seul algorithme toujours plus rapide que les autres mais bien un algorithme plus rapide pour un certain nombre données. On voit donc par exemple que le Tim est le plus rapide pour 256 et 8192 éléments mais que pour 65536 c'est le radix qui présente un temps de tri moyen plus faible.

On peut également noté que ces résultats illustrent bien la complexité temporelle des tris quadratiques puisque lorsqu'on compare l'évolution du temps de tri par rapport au nombre de données on voit bien que le sélection ou bubble ont des résultats qui augmentent beaucoup plus que les autres.

A titre de comparaison avec une référence, on peut voir que certains des algorithmes que nous avons développés se trouvent être plus efficace que le tri proposé par la libc standard. On peut penser que cela est dû au tri de la libc qui est peut-être plus flexible au détriments des performances.

Finalement, on peut aussi noter que l'insertion mais bien en valeur la notion de meilleur et pire cas quand on compare la grande différence qu'il y a entre son temps maximum de tri et son temps minimal, différence qui est aussi présente en ses extremums et sa moyenne.

3.3 ARM Cortex A9

Le tableau ci-dessous donne les temps de tri en µs obtenus pour les différentes algorithmes avec le CPU ARM. Pour cette cible, nous n'avons pas pu obtenir de résultats pour 65536 données et ce même en essayant les différentes options de compilation possibles (O2 et O3).

	NB_DATA = 256			**NB_DATA = 8192**		
Algorithme	Moy	Max	Min	Moy	Max	Min
Radix	146	152	146	**4,333**	4,335	4,331
Tim	**69**	173	65	4,815	8,418	4,665
Quick	134	149	128	6,926	7,674	6,613
Bitonique	311	414	307	23,794	30,832	23,495
Insertion	612	643	596	611,342	616,139	605,223
Selection	1,154	1,155	1,152	1,137,863	1,137,813	
Bubble	**1,770**	1,807	1,714	**1,191,771**	1,911,383	1,161,968

Tableau 2 – Tableau récapitulatif des résultats obtenus sur CPU ARM Cortex A9

Bien que l'architecture de CPU soit différente, on retrouve les mêmes résultats pour ce qui est de l'algorithme le plus rapide qui est le radix pour un nombre d'éléments plus élevé. On remarque également que les temps de tri sont supérieurs à ceux sur x86 avec par exemple un radix à peu près 57 fois plus rapide en moyenne sur x86 pour le même nombre de données (8192 données). On peut également relevé que pour ce CPU on trouve un tri par insertion plus lent que le bitonique pour 256 données, ce qui n'était pas le cas pour le x86. De plus, on peut voir qu'avec le processeur ARM le tri par insertion présente un écart beaucoup plus faibles entre ses extremums et entre ces derniers et sa moyenne.

Au vu du temps de simulation que l'on obtient pour tester un nombre de données élevé avec ainsi que la résolution tardive d'un soucis lié à la taille de heap/stack de la Pynq qui bloquait les tests lorsqu'on avait plus de 32728 éléments, nous avons choisit de ne simuler un nombre de données élevé que pour les deux algorithmes que nous souhaitions accélérer avec l'utilisation du FPGA à savoir le bitonique et le radix.

3.4 Comparaison

Les courbes des figures 3.2 et 3.3 représentent le temps d'exécution en fonction du nombre d'éléments à trier. On peut constater que pour les algorithmes Radix et Insertion la durée d'exécution maximale se distingue beaucoup sur le CPU x86 du temps d'exécution moyen. Cela est en partie dû aux appels systèmes effectués durant le benchmark, ce qui ralenti l'exécution des algorithmes. En changeant l'évolution des données par un incrément constant de 256 nous avons obtenus une courbe qui présente des pics réguliers pour le maximum de temps de tri, on pourrait également penser que ces pics sont dûs à l'activation du turbo boost qui s'active pendant une certaine durée (augmentant les performances) mais qui s'arrête ensuite pour permettre au processeur de refroidir (chute des performances qui pourrait donner un pic). Cette différence n'est pas présente sur le CPU de la PYNQ car l'ensemble des ressources sont dédiées aux algorithmes grâce à l'absence d'OS et on peut noter également l'absence de turbo boost pour le processeur ARM.

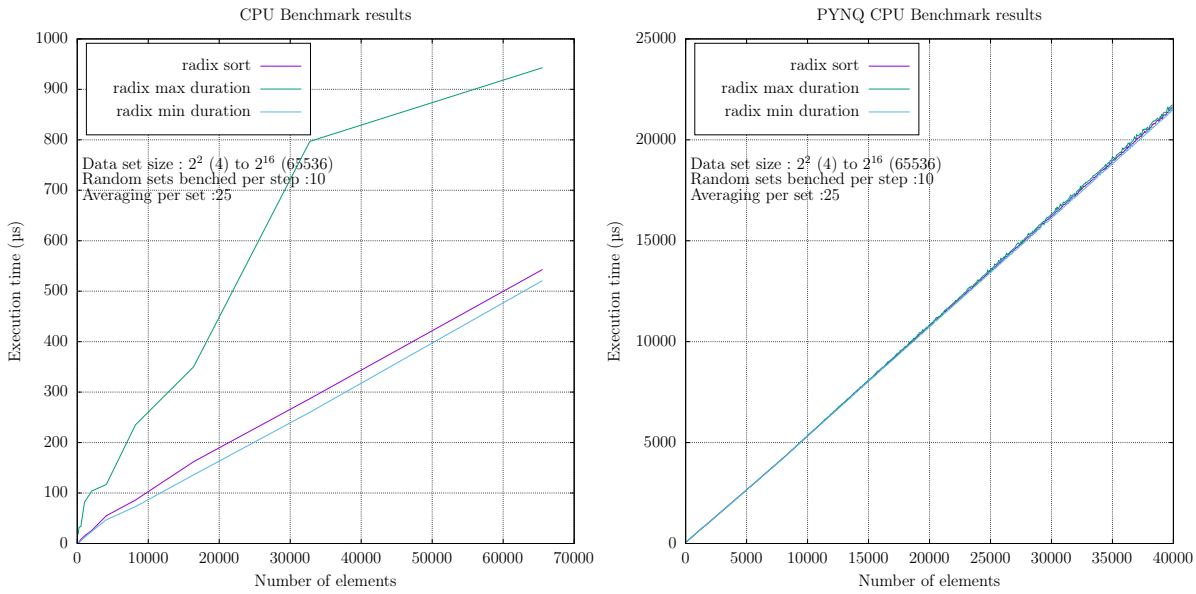

FIGURE 3.2 – Résultats tri radix

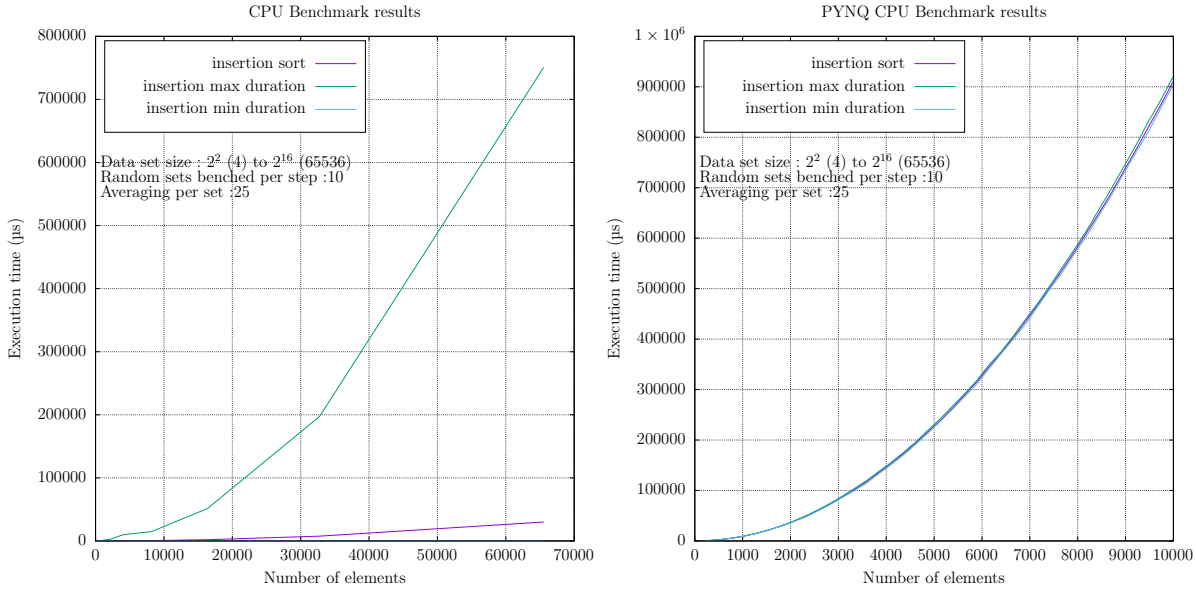

FIGURE 3.3 – Résultats tri insertion

4 Solution FPGA

Nous avons également sélectionne une cible FPGA afin d'y mettre en place les performances des algorithmes de tri, et de comparer les performances, la consommation et le coût de cette solution face aux autres.

4.1 Environnement de test et de développement

La carte cible est une Pynq-Z2, disposant d'un Zynq 7020 de Xilinx. Cette puce contient deux processeurs ARM A9 accompagne d'un FPGA de 1.3 millions de portes.

Le processeur est donc celui de la partie précédente, pour mesurer les performances des algorithmes sur CPU.

Cette partie se concentre donc sur l'accélération du tri par le FPGA. Pour cela, deux méthodes seront utilisées : la réalisation complète du tri sur le FPGA, ainsi qu'une coopération entre le CPU et le FPGA. Les données seront donc stockées sur le CPU, envoyées au FPGA puis récupérées triées. La communication des données entre le processeur et le FPGA se fera via un bus AXI.

4.2 Protocole de communication AXI

4.2.1 Architecture de la communication

Le bus AXI est celui utilisé au sein des puces Xilinx pour gérer la plupart des communications internes à la puce. Nous avons donc choisi d'utiliser différentes implémentations de ce bus afin de mettre en place une communication rapide entre le processeur et l'accélérateur de tri.

On peut voir sur la figure 4.1 une représentation des communications entre le processeur (Processing System, ou PS) et le FPGA (Processing Logic, ou PL). Afin d'avoir des communications rapides et n'interrompant pas le CPU lors des transferts de données, la mise en place d'un DMA (Direct Memory Access) était indispensable. Le processeur échange donc les données avec le DMA via un bus AXI-Lite, version simplifiée mais restant adressée du protocole AXI. Du côté de l'accélérateur, on choisit d'utiliser un AXI-Stream pour sa simplicité : le module se contente de lire une file de données à travers le DMA pour recevoir les données, et d'en remplir une en sortie afin de les remettre au CPU. Il suffit alors du côté du CPU de lire les données d'un tableau à partir de l'adresse de départ du buffer de réception.

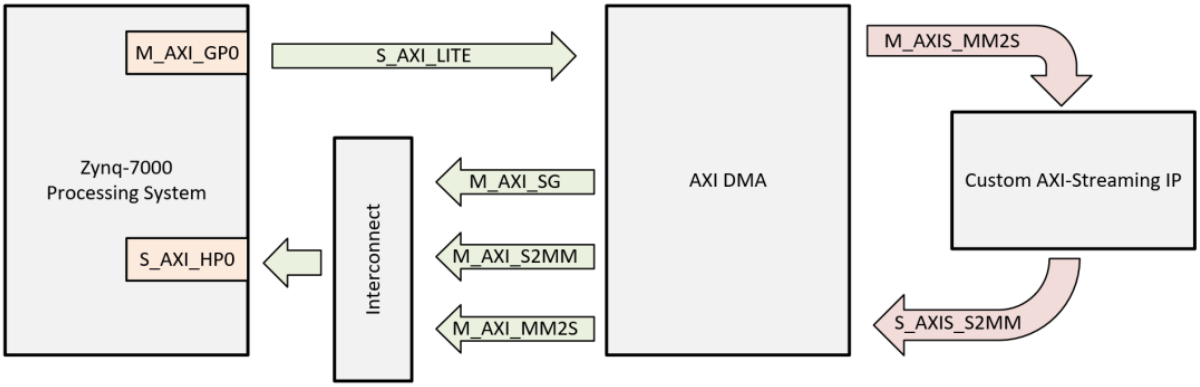

FIGURE 4.1 – Schéma de la communication AXI mise en place

4.2.2 Performances

Afin de déterminer si l'accélération pouvait être avantageuse, il était tout d'abord nécessaire de s'assurer que le temps de communication PS/PL était plus court que le temps de tri sur le CPU seul, auquel cas aucun accélérateur ne serait intéressant. Nous avons donc mesurer ce temps en plaçant un simple module de FIFO à la place de l'accélérateur, comme on peut le voir sur la figure 4.2 ci-dessous.

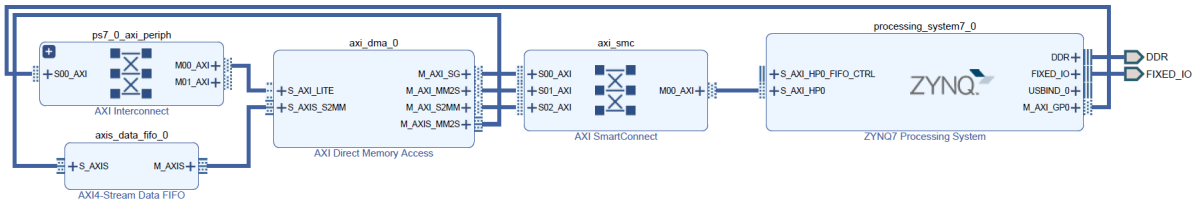

FIGURE 4.2 – Block design de mesure de débit du bus AXI

Depuis le CPU, on mesure donc le temps entre l'envoie de la première donnée et la réception de la dernière donnée, chaque donnée étant un entier sur 4 octets et envoyée dans un paquet de 2048 entiers. On obtient les résultats suivants :

NB_DATA	**2048**	**8192**	**65536**	**131072**
Temps (µs)	6512	6177	3384	3361
Débit (Mbit/s)	10.1	42.4	619.7	1247.9
Nombre de paquets	1	4	32	64

Tableau 3 – Mesures de performances du bus AXI

On voit que le débit augmente fortement avec la quantité de données envoyées, ce qui paraît intuitif au premier abord. En revanche, le temps de communication diminue avec la quantité de données envoyée, ce qui n'est cette fois pas intuitif car on s'attendrait normalement à un temps de communication du type $t = quantite/debit + latence$, avec la latence représentant un coût fixe indépendant de la quantité de données envoyée. Ce phénomène n'a pas été clairement identifié, il faudrait pour cela approfondir notre maîtrise du bus AXI, du DMA et vérifier le code Assembleur généré par le compilateur pour vérifier d'où vient cette optimisation.

On peut tout de même conclure que pour trier 65k éléments, l'intégration d'un accélérateur peut être intéressante : le temps de communication est plus court que le tri de 8k éléments sur le CPU seul, il y a donc place à un gain de performance en mettant le FPGA à contribution. Pour comparaison, le tri de 65k éléments via le tri radix sur le processeur ARM prenait 28,795 µs, et 353,574 µs pour le tri bitonique.

4.3 Zynq 7020

Pour des questions de temps de développement mais aussi de simplicité nous avons décidé d'utiliser Vivado HLS pour générer les modules de tris pour le FPGA. En effet, ce logiciel nous permettait de synthétiser les modules à partir des algorithmes en C développés pour les tests sur la partie CPU mais également d'essayer d'optimiser les algorithmes avec des directives de pipeline ou unroll. Les algorithmes ayant fait l'objet d'une implémentation FPGA sont le bitonique et le radix. Le bitonique étant un réseau de tri il correspond à un algorithme qui se prête bien à la parallélisation alors que le radix quant à lui présente des étapes qui se prêtent peu à ce genre d'implémentation. Toutefois, le radix implémenté présente quelques modifications permettant d'améliorer le parallélisme avec une augmentation de complexité spatiale pour permettre la mise en parallèle de certaines étapes. L'algorithme est aussi adapté pour être plus efficace sur FPGA avec l'utilisation de la base hexadécimal pour le tri qui fait appel à des décalages et masquages pour le tri au lieu de modulos sur les chiffres (base décimal).

4.3.1 Tri radix sur FPGA

Le tableau ci-dessous correspond aux résultats obtenus avec l'implémentation du tri radix sur FPGA. L'utilisation des ressources correspond au module de tri uniquement, le temps prend en compte le temps de tri auquel s'additionne le temps de communication et enfin la puissance et le chemin critique correspondent à l'implémentation complète (design complet avec bus AXI).

Nb données	**Temps**(µs)	**Conso** (W)	**LUT**	**FF**	**BRAM**	**WNS** (ns)
256	6496	1,352	3293 (6%)	1417 (1%)	3%	2,11
8192	5281	1,392	3328 (6%)	1467 (1%)	14%	1,927
32768	9752	1,48	3369 (6%)	1487 (1%)	48%	1,245
65536	19459	1,609	3386 (6%)	1496 (1%)	94%	0,355

Tableau 4 – Tableau récapitulatif des résultats du tri radix sur le FPGA Zynq 7020

Ce que l'on tout d'abord donner c'est l'évolution du temps total qui présente une valeur plus basse pour un nombre de données plus grand à 8192, cette valeur est expliquée par un changement de taille de paquet pour la communication avec le bus AXI. Ainsi, on voit que le taille du paquet a un impact conséquent sur les performances et elle doit donc être adaptée au nombre de données envoyées de façon à avoir le plus grand paquet possible.

Pour ce qui est de la consommation, on note qu'elle est basse comparer à celle d'un processeur x86 et qu'elle augmente avec le nombre de données à traiter. Cette augmentation est liée à l'utilisation de signaux et blocs ram supplémentaires.

On remarque également que l'évolution des LUT utilisées en fonction du nombre de données est très faible, en effet avec cet algorithme le nombre de données ne joue pas vraiment et n'augmente qu'à cause des instances supplémentaires crées.

Enfin, le chemin critique de l'implémentation diminue avec le nombre de données de tel façon que pour un nombre d'éléments plus grand que 65536 il va rapidement devenir difficile de tenir les exigences temporelles imposées par l'horloge actuelle.

Ainsi, au vu des résultats obtenus on peut penser qu'il serait difficiles de faire une implémentations de plusieurs blocs de tri radix sur FPGA pour un grand nombre de données principalement à cause des conflits sur les accès aux blocs RAM ou encore au niveau du chemin critique.

4.3.2 Tri bitonique sur FPGA

Le tri bitonique présente l'avantage théorique de favoriser la mise en parallèle des comparaisons. En effet, à chaque étape, on peut effectuer, pour n données à comparer, $n/2$ comparaisons/échange d'indice en parallèle, pour $O(\log_2^2 n)$ étapes. Complètement déployé, le réseau nécessite donc $O(n \log_2^2 n)$ comparateur, soit une dizaine de millions pour trier 65k éléments. Il est donc évident qu'un tel réseau ne peut pas être déployé sur notre cible.

Si l'on se limite au nombre de comparateurs nécessaire en même temps, on arrive à "seulement" 32k comparateurs, cependant il reste impensable de réaliser une telle architecture, le FPGA disposant de 50k LUT6, et les signaux de contrôles deviendraient

beaucoup trop contraignants. Plusieurs approches on donc été utilisées pour tenter d'implémenter ce réseau.

La première approche est de déployer le réseau partiellement : on déploie seulement une partie des comparateurs à chaque étape afin de limiter la complexité spatiale. Cela a cependant pour conséquence d'augmenter fortement la complexité temporelle. Une première implémentation reprend simplement l'algorithme C, séquentiel, utilisé précédemment, tandis qu'une deuxième tente de mieux utiliser les ressources disponible au travers de directives HLS.

Version	**Temps** (µs)	**Conso** (W)	**LUT**	**FF**	**BRAM**	**WNS (ns)**
HLS 1	156,891	1.5	6,652 (12%)	9,191 (8.6%)	66 (47%)	1.2
HLS 2	193,305	1.5	24,456 (46%)	20,748 (19%)	66 (47%)	-0.1
CPU	353,574	1.3	-	-	-	-

Tableau 5 – Résultats du tri bitonique sur le FPGA Zynq 7020, 65536 éléments

On voit donc dans ces résultats que dans sa version la plus rapide, l'utilisation du FPGA permet de gagner un facteur deux comparativement au CPU seul. Cependant, la version 1 ne tire avantage d'aucune directive HLS, il est donc naturel de penser qu'en ayant une description plus précise et efficace, le facteur de gain peut devenir beaucoup plus important.

La version 2 quand à elle illustre une mauvaise utilisation des directives HLS : elle utilise plus de ressources que la version 1, ce qui était désiré, mais reste plus lente, ce qui n'est évidemment pas le but attendu. Il aurait fallut mieux utiliser les directives et regarder plus en détail l'architecture générée par l'outil afin de l'améliorer, car beaucoup de facteurs peuvent entrer en jeu ici, comme l'augmentation du nombre de signaux de contrôles ou une mauvaise répartition des données en mémoire empêchant un accès parallèle. La description du tri bitonique sur FPGA étant affaire de compromis, il est donc important de tous les décrire de manière cohérente sans que l'outil puisse faire des compromis contradictoires du point de vue des performances.

La deuxième approche est de faire coopérer le CPU et le FPGA dans le travail de tri des données : le tri bitonique étant le plus efficace temporellement lorsqu'il est totalement déployé, on peut en déployer un complet plus petit, et utiliser le CPU pour préparer ou fusionner les données renvoyées par le réseau sur FPGA. On choisit dans notre cas d'envoyer des paquets de données triés entre eux au FPGA grâce à une variante du quick sort permettant de séparer l'ensemble initial en sous-ensembles de taille fixe et sans intersection. Le réseau bitonique doit donc s'occuper de trier ces sous-ensembles successivement, les données renvoyées au CPU étant alors toutes triées.

La séparation en sous-ensembles d'éléments triés entre eux étant de plus en plus coûteuse à mesure que le nombre de sous-ensembles augmente – le tri étant complet lorsque le nombre de sous-ensembles est égale au nombre d'éléments –, on souhaite donc créer un tri bitonique, pipeliné, de plus grande taille possible.

On crée donc un réseau bitonique de taille 16 et 32, en séparant sur le CPU en sous-ensembles de taille correspondante. On obtient alors les résultats suivants :

Taille	**Temps** (µs)	**Conso** (W)	**LUT**	**FF**	**BRAM**	**WNS** (ns)
8	44,731	1.5	7,337 (14%)	9,605 (9.0%)	2 (1%)	0.2
16	31,208	1.5	9,066 (17%)	10,263 (9.5%)	2 (1%)	-0.9
32	24,650	1.6	14,629 (28%)	11,556 (11%)	2 (1%)	-3.2

Tableau 6 – Résultats du tri bitonique mixte CPU/FPGA pour 65536 éléments

Tout d'abord, on voit que le réseau bitonique de taille 32 n'est pas utilisable dans les contraintes de temps imposées et ne marche pas à tous les coups lors des tests, et ce malgré un WNS indiqué à 4 ns post synthèse HLS. On note cependant que lorsque ce module fonctionne, les temps de traitement atteints sont beaucoup plus faibles que ceux du FPGA seul, ce qui semble indiqué qu'une approche mixte de deux algorithmes est bonne.

Les modules avec un réseau de taille 16 et 8 sont, comme prévu, plus lents mais restent bien plus intéressants aussi bien pour leur complexité spatiale que pour leur temps d'exécution que les implémentations CPU ou FPGA seules.

Dans la continuité de ces résultats et vu la place occupé par un tri bitonique de taille 16 ($\sim$ 3k LUTs sans le bus AXI et le DMA) et 8 ($\sim$ 1.5k LUTs dans les même conditions), on peut imaginer accélérer la séparation des données sur le FPGA également et mettre plusieurs réseaux en parallèle, ou passer par une fusion des données en sortie plutôt qu'un pré-tri. Il est également possible de raffiner les architectures en passant directement en HDL afin d'encore augmenter les performances des modules.

4.3.3 Comparaison entre bitonique et radix

Bien qu'étant adapté au parallélisme, le tri bitonique n'atteint pas les performances du radix dans notre test. Cependant, il est possible d'y parvenir en dévouant plus de temps à l"optimisation de tri bitonique, en effet les performances de ce tri sont très dépendantes du déploiement du réseau qui est une étape assez compliquée à bien décrire avec HLS. Le bitonique serait alors une solution plus efficace et beaucoup moins gourmande en mémoire que le radix.

4.3.4 Comparaison avec résultats CPU

Dans le cas du tri radix pour 65536 éléments on voit que l'accélération FPGA souhaitée est bien réalisée étant donné que le CPU réalise le tri en 28795 µs contre 19459 µs avec l'accélération FPGA, ce qui donne une diminution du temps total de 48% pour ce test. Globalement, on note que même si peu d'étapes sont parallélisées on peut obtenir un gain de performances avec l'utilisation du FPGA. Il en est de même pour le tri bitonique pour lequel l'accélération est d'autant plus grande de par la structure parallélisable de ce tri. Toutefois, le temps de communication qui apparaît avec l'utilisation de l'accélération FPGA reste conséquent et devient une source de désaccéleration pour un faible nombre de données.

5 Solution GPU

Finalement, nous avons désiré essayer quelques solutions sur processeur graphique en utilisant les cartes de développement Jetson TK1 précédemment utilisée dans un autre projet.

Une première idée se base sur un tri radix arrangé pour du parallélisme. Une deuxième solution utilise le CPU pour effectuer un tri préliminaire avant la parallélisation.

5.1 Tri radix parallélisé

Le but de cette solution était à l'origine d'effectuer l'ensemble du tri des données uniquement sur le GPU, en comptant tout de même le temps de transfert des données entre le CPU et le GPU. La plupart des algorithmes classiques ne sont cependant pas adaptés à ce genre de configuration à cause des multiples écritures en mémoire à une même adresse. Le tri radix fait cependant exception à ce problème car il effectue un et un seul déplacement de chaque donnée pour chaque tri partiel de radix. C'est pour cette raison que nous avons tenté de l'adapter à une architecture multi-coeur.

Originellement, le tri radix se base sur la comparaison des chiffres des données en base décimale. Pour des données possédant au maximum 8 chiffres par exemple, il faut effectuer 8 tris partiels pour chaque digit. Dans notre cas d'application, nous avons cependant utilisé la base 256 pour comparer 8 bits à la fois, ce qui raccourci l'algorithme à 4 étapes de tris partiels (avec des données sur 32 bits).

Pour chaque tri partiel nous avons alors les étapes suivantes à réaliser :

- On commence par dénombrer chaque chiffre (en base 256) présent dans le jeu de données présent. Il faut donc un tableau de la taille de la base utilisée, ici 256.
- En connaissant les occurrences de chaque chiffre, on peut calculer les indices dans le tableau de données où il faut commencer à écrire à nouveau les nombres contenant ces chiffres pour obtenir un ensemble trié. Cette opération se résume à des sommes des occurrences précédemment comptées.
- La dernière étape consiste à écrire les données dans un nouvel ordre partiellement trié par rapport au chiffre regardé en utilisant les indices précédemment calculés.

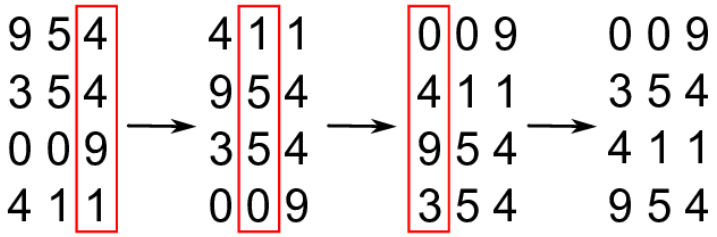

FIGURE 5.1 – Étapes d'un tri radix en base 10 sur 3 chiffres

Nous avons alors choisit de paralléliser ce tri en divisant l'espace mémoire des données en parties continues de tailles équivalentes entre les différents threads :

- Dans la première étape, chaque thread dénombre les chiffres des données présentes dans son sous-espace fixé de mémoire de départ. Chaque thread doit donc utiliser un tableau de 256 valeurs pour dénombrer les 256 chiffres différents.

- La deuxième étape consiste à calculer les indices d'écriture des données pour chaque chiffre pour chaque thread en utilisant l'ensemble des dénombrements précédents de tous les threads. Cette étape se résume à des additions récursives en Zig-Zag dans un tableau à deux dimensions de taille $B \times T$ où B est la base utilisée (ici 256), et T est le nombre de threads utilisés (typiquement compris entre 64 et 4096). La complexité de ce calcul est donc en $O(B \times T)$. Comme ce calcul est difficilement parallélisable et que sa complexité est indépendante du nombre de données triées, nous avons choisit de l'exécuter sur un unique thread.
- Pour la dernière étape, chaque thread déplace les données présentes dans la partie de mémoire initialement dédiée. L'adresse d'arrivée de ce déplacement n'est cependant à priori pas dans l'espace d'adressage d'origine du thread. Les threads doivent donc effectuer des écritures croisées dans un même tableau en mémoire.

Le problème de cette solution est qu'elle implique des écritures croisées dans l'espace d'adressage des données entre les threads.

5.2 Tri mixte CPU/GPU

La deuxième solution consiste à effectuer une partie du tri sur le CPU avant d'envoyer les données au GPU.

La première partie du tri sur CPU permet de diviser l'ensemble des données en paquets de données triés entre eux. Après cette opération, on peut envoyer les données au GPU qui trie chaque paquet individuellement.

La division en paquets triés est effectué en utilisant un algorithme de calcul de médianes basé sur un quick-sort modifié. Après envoie des données , on peut choisir n'importe quel autre algorithme de tri coté GPU, suivant la configuration initiale des données. Nous avons seulement testé avec les algorithmes radix-sort et quick-sort.

5.3 Comparaison

Afin de mesurer l'efficacité des solutions développées, nous avons mesuré les temps minimum, maximum et moyens pour des tris de 256, 8192 et 65536 données avec un lissage des mesures sur 100 jeux de données différents. Nous avons mesuré le tri radix sur CPU, le tri radix sur GPU, le tri radix sur GPU avec séparation préliminaire des données, et le tri rapide avec séparation préliminaire des données en variant le nombre de threads utilisés. Ces résultats sont reportés dans la table 7.

		NB_DATA = 256	**NB_DATA = 8192**	**NB_DATA = 65536**
Algorithme	Threads	Moy	Moy	Moy
Radix CPU	1	18	503	4,964
Radix GPU	64	17,052	22,822	49,071
Split+Radix	512	NC	46,454	19,320
Split+Quick	128	1,205	9,843	73,419
Split+Quick	256	2,365	8,372	42,578
Split+Quick	512	NC	13,022	35,603
Split+Quick	1,024	NC	32,843	62,369

Tableau 7 – Tableau récapitulatif des résultats obtenus sur GPU

On remarque en premier que le tri radix totalement sur GPU est particulièrement inefficace, comparé au simple tri radix sur CPU. Le problème vient très probablement de la restriction des accès à la mémoire entre les différents threads. L'autre solution basée sur un pré-triage sur CPU reste également plus lente qu'un simple tri sur CPU, même en changeant le nombre de threads utilisés.

On observe d'ailleurs une amélioration du temps de temps calcul avec l'augmentation du nombre threads, jusqu'à un certain seuil où la tendance s'inverse. On remarque par ailleurs que pour des jeux de données plus gros, l'emploi de plus de threads est plus efficace.

Concernant la mesure de temps, on mesure également que plus de 75% du temps est utilisé par le GPU (sans même compter les temps de transfert). L'inefficacité temporelle de ces solutions viens donc des codes des kernels. Le problème viens très probablement de la restriction des accès à la mémoire (de taille importante) entre les threads.

6 Comparaison des solutions

Cette partie va donc traiter d'une comparaison globale entre les différentes solutions abordées en se basant sur les critères définis par le cahier des charges dont les grandes lignes sont récapitulées par le tableau 8.

	Conso	**Prix**	**Temps de tri (meilleur)**	**Flexibilité**
CPU ARM	faible	faible	élevé	élevée
CPU x86	élevée	élevé	très faible	élevée
FPGA	faible	moyen	faible	faible
GPU	faible	moyen	élevé	moyenne

Tableau 8 – Comparaison des résultats pour les différentes solutions étudiées

Comme on pouvait s'en douter, il n'y a pas de solution miracle qui va répondre parfaitement à tous les critères mais plutôt une solution qui proposera le bon compromis pour l'application qui l'utilise. Effectivement, pour ce qui serait d'un système embarqué ne nécessitant pas de performances trop strictes le processeur ARM représente une bonne solution de par son prix intéressant, sa faible consommation, flexibilité et même sa taille, c'est d'ailleurs pour cette raison que l'on trouve de plus en plus cette architecture de nos jours.

Un CPU x86 dernière génération de chez Intel ne correspondrait pas vraiment pour une application embarqué de par sa forte consommation et son prix auquel on pourra ajouter son manque de portabilité. En effet, ce type de technologie de précision est difficilement exploitable sans l'utilisation d'un OS qui implique de ce fait d'avoir des périphériques supplémentaires et qui apporte également tous les inconvénients qui viennent avec un OS (même temps réel dur).

Pour ce qui est de la solution FPGA, elle se trouve être très prometteuse au niveau du rapport performance/consommation, le parallélisme de cette technologie étant dans bien des cas un bon moyen d'augmenter les performances d'un système. Toutefois, on se retrouve vite confronté à un problème de flexibilité de l'architecture qui a tendance à être plutôt spécifique à une application. Il faut également ajouter à ceci la gestion des problèmes liés à la communication ou à la programmation qui bien que de plus en

plus performants et accessibles nécessitent du temps pour être efficacement paramétrer. De plus, la programmation bien qu'étant plus rapide et facilité avec des outils comme HLS, elle reste néanmoins compliquée à faire correspondre exactement aux attentes et le passage directement par le VHDL ou Verilog nécessite un temps de développement souvent plus élevé et une bonne connaissance du circuit.

Enfin, la solution GPU bien qu'au premier abord idéale s'est trouvée un peu difficile à exploiter par manque de temps. Effectivement, à l'instar de la partie FPGA elle nécessite d'être bien paramétrée pour être efficace. Cependant, elle peut rester viable pour une application embarqué au vu de sa consommation et du prix.

Un autre critère que nous n'avons pris en compte dans cette comparaison mais qui sera totalement justifié serait la complexité de programmation et ainsi le temps de développement qui en découle. Comme rapidement évoquée au-dessus, une solution utilisant un FPGA bien que efficace vis à vis de nos critères peut se révéler lourde en temps de développement augmentant ainsi le prix lié à cette solution. Nous l'avons notamment constaté avec le tri bitonique qui pourrait être extrêmement efficace mais qui nécessite un paramétrage pointu que l'on peine à atteindre avec Vivado HLS ou bien avec la configuration du bus AXI qui bien que très performant nous a été très coûteux en temps. Ainsi, on pourrait ajouter cette dimension à nos critères pour rendre la comparaison plus réalistes et parlante dans le cas du développement d'un produit par une entreprise où le temps développement est synonyme de coût salariale.

7 Conclusion

Ce projet avec un sujet très ouvert était l'occasion d'explorer différentes solutions et implémentations pour un même problème, le tout en étant capable de se fixer des limites et un cadre de travail pour définir et répartir ce les tâches à réaliser. C'était également l'occasion d'approfondir nos connaissances et renforcer nos compétences avec les différentes cibles et logiciels utilisés tels que Vivado HLS pour la génération de modules pour FPGA, le développement en langage C avec l'utilisation des bibliothèques tels que OpenMP ou encore l'utilisation de CUDA pour la programmation de coeurs GPU. Nous avons également pu découvrir et utiliser un nouveau protocole de communication qu'est le bus AXI ainsi qu'un nouvel outil de développement qu'est Vivado SDK. Ces deux derniers nous ayant permis d'établir une connexion entre un FPGA et un CPU sur la même carte.

De plus, ce projet était très enrichissant au niveau de la réflexion apportée par l'étude de différents algorithmes de tri et de leur optimisation, tout comme la tentative de création d'algorithmes en combinant et modifiant des algorithmes existants.

Finalement, ce projet était une très bonne conclusion à ce semestre puisqu'il nous a permis de travailler en groupe et en autonomie tout en revoyant et utilisant les différentes compétences et connaissances acquises tout au long de notre formation pour développer des solutions fonctionnelles pour le tri de données.