PR310 - Projet avancé en systèmes embarqués

Algorithmes de tri par codesign

Date: 27 / 01 / 2023

BOUT Antoine

CORNUEZ Xavier

KEOMANYVONG Hannah

MARTIN MOYANO Adolfo

Encadré par : B. LE GAL

Table des matières

Introduction

Ce rapport rend compte du travail effectué dans le cadre du projet avancé système embarqué. Ce projet porte sur l'étude d'algorithmes de tri et sur leur implémentation matérielle. Ce rapport détaille les études de plusieurs algorithmes ainsi que des optimisations que nous avons apportées. Nous avons travaillé aussi bien sur des améliorations logicielles que matérielles ciblant un FPGA.

Tri par insertion

1. Principe du tri par insertion :

Le tri par insertion consiste à comparer chaque élément du tableau aux éléments précédents qui ont déjà été triés, et effectuer une permutation des données si elles ne sont pas dans l'ordre croissant. Le fonctionnement de ce tri est plus facilement visualisable à l'aide de son code en langage C ci-dessous.

Pour le tri du tableau table de taille N, on réalise le tri par insertion à l'aide du code ci-dessous :

```
for (int i=1 ; i <= N-1; i++)
    {
    j=i;
      while (j > 0 && table[j-1] > table[j])
        {
            tmp = table[j];
            table[j] = table[j-1];
            table[j-1] = tmp;
            j--;
        }
    }
```

Figure 1 Code C du tri par insertion d'un tableau table de taille N

L'algorithme est composé de deux boucles imbriquées. La 1ère boucle incrémentant l'entier i permet de sélectionner l'élément du tableau que l'on souhaite trier. La 2ème boucle décrémentant l'entier j permet d'effectuer la comparaison de l'élément à trier avec les éléments du tableau triés précédemment. Ainsi, si l'élément est inférieur à des données déjà triées, celui-ci sera inséré au bon endroit par une série de permutations 2 par 2.

<u>Complexité de l'algorithme de tri:</u>

Le nombre maximal de comparaisons pour le tri d'un tableau de taille n est de $C(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{(n-1)*n}{2}$. Ainsi, la complexité du tri par insertion est de $O(n^2)$.

En effet, étant donné le principe de l'algorithme, le nombre de comparaisons augmente de manière quadratique. Ainsi, il n'est approprié seulement pour le tri d'un tableau avec peu de données. Cependant, les performances de ce tri pour un tableau qui est relativement trié s'améliorent. On obtient une complexité de O(k*n), pour un tableau possédant un élément qui doit être déplacé de k cases pour être trié.

Remarque : Ce tri possède la particularité de pouvoir commencer son exécution avant de recevoir toutes les données du tableau. En effet, le tri par insertion commence dès la réception d'au moins deux données.

Dans cette partie, on s'intéresse aux divers moyens d'optimiser le tri par insertion en utilisant des approches logicielles et matérielles.

2. Optimisations logicielles :

Dans un premier temps, on cherche à optimiser les performances du tri par insertion d'un point de vue strictement logiciel. On s'intéresse à diverses pistes d'optimisation telles que la gestion des pires cas (tableau dans l'ordre décroissant), et les échanges de données au sein du tableau (accès à la mémoire). De plus, on s'intéresse à l'éventuelle parallélisation de cet algorithme à l'aide de OpenMP.

a. Optimisation du tri par insertion : prétraitement du tableau.

Principe

Une première approche pour optimiser les performances de cet algorithme est de traiter d'abord les pires scénario. Dans le cas où le minimum du tableau, de taille n, est placé en dernière position n-1, notre programme doit réaliser n-1 comparaisons et échanges de données 2 à 2 pour placer le minimum en position 0. Le pire scénario est donc lorsque le tableau est trié dans l'ordre décroissant, on doit alors réaliser les (n-1)*n/2 comparaisons. La première optimisation visant à réduire le nombre de comparaisons effectuées correspond à faire un prétraitement du tableau. En effet, avant d'effectuer le tri par insertion du tableau, on peut modifier l'ordre des données dans celui-ci pour réduire le temps d'exécution global du tri.

Une solution trouvée a été de comparer les données du tableau 2 par 2 à partir de ses extrémités, et les échanger si elles ne sont pas dans le bon ordre. Ainsi, on réalise un premier traitement du tableau de complexité O(n/2) échangeant les valeurs aux extrémités, puis on réalise finalement le tri par insertion conventionnel sur ce tableau. Dans le pire cas (tableau dans l'ordre décroissant), le nombre de comparaisons serait alors réduit à (n/2)+(n-1), soit O(3/2 n).

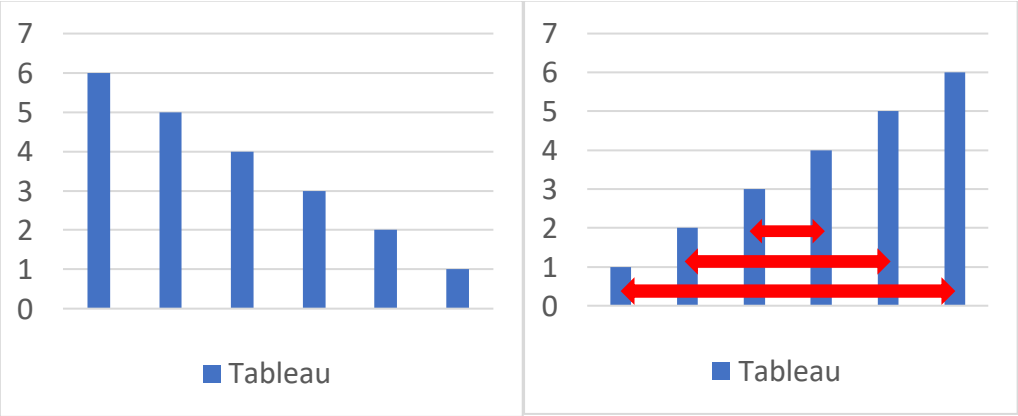

Figure 2 Schéma permettant de visualiser les permutations réalisées avant le tri par insertion

```
int start=0;
  int end=N-1;
  while(start<end)
  {
    if (table[start]>table[end]) //swap
    {
        tmp = table[start];
        table[start] = table[end];
        table[end] = tmp;
    }
```

Figure 3 Code C de l'optimisation exécutée avant le tri par insertion d'un tableau table de taille N

Cette solution permet d'améliorer les performances de ce pire cas du tri par insertion et, de manière générale, améliore les performances du tri pour des tableaux possédant des valeurs mal placées en fin de tableau. Cependant, elle dégrade les performances pour le tri d'un tableau qui est relativement bien trié. Il faut ainsi vérifier expérimentalement le gain en performances pour des tableaux générés aléatoirement.

Mesure des performances

On réalise une mesure des performances de l'optimisation logicielle implémentée pour des tableaux aléatoires de tailles différentes. Le graphique ci-dessous montre la durée moyenne du tri pour différentes tailles de tableaux avec des données int32_t. Pour chaque taille de tableau, on réalise entre 4096 et 1024 tris de tableaux indépendants et générés aléatoirement, et on effectue la moyenne de tous les temps d'exécution. Sur le graphique, la courbe en rouge prend en compte le temps de modification du tableau et de tri de celui-ci.

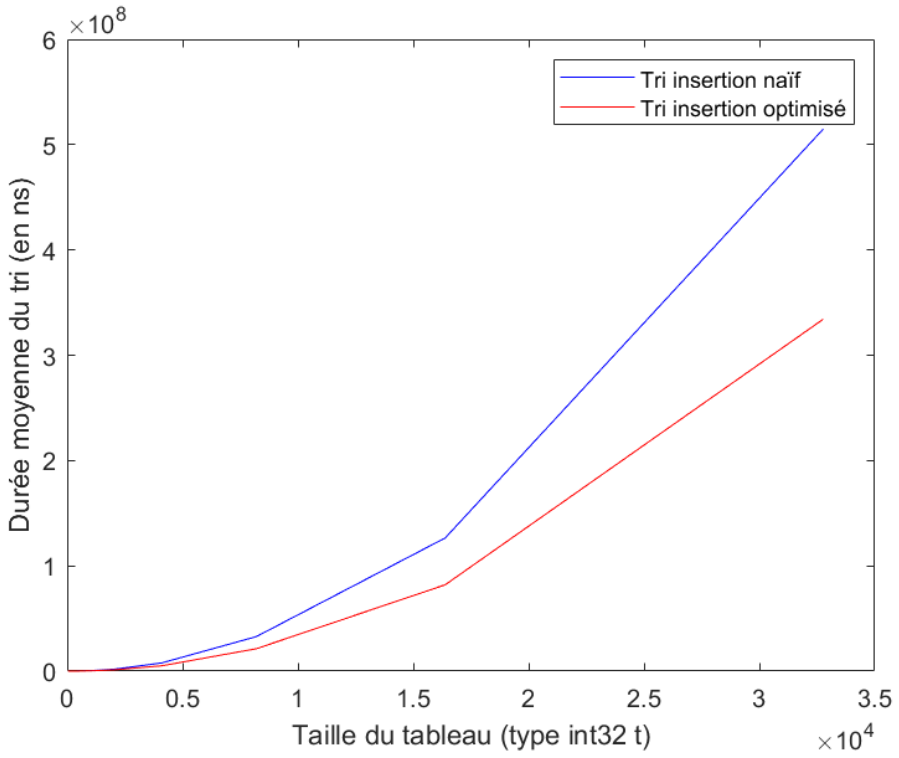

Figure 4 Graphique montrant l'amélioration des performances avec l'optimisation de prétraitement du tableau (en rouge)

La mesure des performances permet de vérifier cette optimisation de l'algorithme de tri avec une diminution évidente du temps nécessaire pour le tri des tableaux. En effet, pour un tableau de 32 768 données, on a une diminution moyenne du temps d'exécution de l'ordre de 40%.

b. Optimisation du tri par insertion : gestion des opérations de lecture et écriture sur la mémoire

Traditionnellement, le tri par insertion réalise des échanges 2 à 2 de chaque donnée du tableau qui est triée. Un moyen d'améliorer les performances du tri est de considérer que le tri par insertion réalise un décalage des données du tableau et non pas des échanges 2 à 2. Le code sur la figure ci-dessous illustre cette idée. Contrairement aux échanges de données sur le code Figure 1, dans cette boucle while, les données du tableau sont décalées et la donnée triée est placée finalement lorsque la boucle while finit de s'exécuter.

```
for (int i=1 ; i <= N-1; i++)
    {
    j=i;
    tmp = table[j];
      while (j > 0 && table[j-1] > tmp)
        {
            table[j] = table[j-1];
            j--;
        }
    table[j] = tmp;
    }
```

Figure 5 Code C du tri par insertion d'un tableau table de taille N avec optimisation des accès mémoire

Cette optimisation permet de réduire les accès mémoire en réduisant le nombre d'opérations de lecture et écriture. Ceci entraine une amélioration significative des performances du tri par insertion.

Mesure des performances

On réalise une mesure des performances de cette optimisation pour des tableaux aléatoires de grande taille. Dans ce cas, on effectue seulement deux tris pour chaque taille de tableau. Le temps d'exécution des tris sont représentés sur la figure ci-dessous.

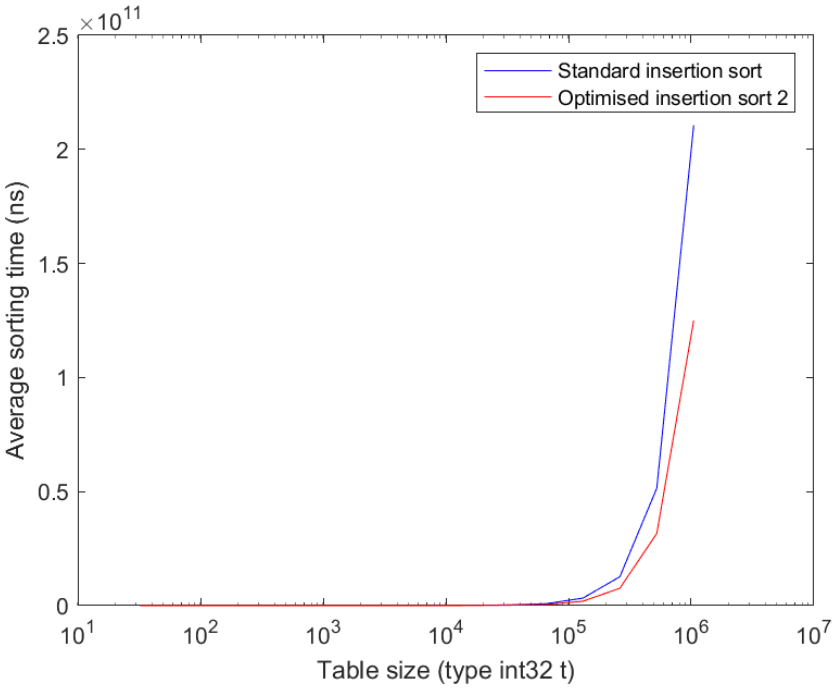

Figure 6 Graphique montrant l'amélioration des performances avec la diminution des accès à la mémoire

On obtient une amélioration des performances pour obtenir une réduction du temps d'exécution de 40% pour les tableaux de 1 048 580 données.

c. Optimisation par parallélisation du tri par insertion

Principe

Le tri par insertion trie les données une à une depuis la 1ère position 0 jusqu'à la dernière position n-1. Ainsi, pour des tableaux de taille importante, l'algorithme trie seulement la première partie du tableau pendant un certain temps. L'optimisation réalisée dans cette partie exploiterait ce temps de tri de la 1ère partie du tableau pour commencer à trier la 2ème moitié du tableau en parallèle.

L'algorithme de tri se compose alors de 2 étapes :

- Etape 1 : tri par insertion des 2 moitiés du tableau de manière indépendante
- Etape 2 : insertion des données triées de la 2ème moitié dans la 1ère moitié pour obtenir le tableau final trié dans l'ordre croissant

En triant la 2ème partie du tableau en parallèle, l'insertion de ces données dans la première partie du tableau nécessite au maximum N/2 permutations, au lieu de N pour le tri par insertion standard.

La parallélisation logicielle de ce tri se fait à l'aide de l'API OpenMP. En effet les pragma ''#pramga omp parallel sections'' et « pragma omp section » permettent d'exécuter deux boucles for de tri en parallèle. Cependant, cette parallélisation ne serait uniquement efficace pour tableaux de très grande taille. La parallélisation implique la mise en place de threads et une architecture plus complexe qui nuit aux performances du tri.

Mesure des performances

Sur la figure ci-dessous sont représentées les performances de la parallélisation et des implémentations précédentes. Une mesure des performances ne permet pas d'obtenir l'effet souhaité pour des tableaux de grande taille. Lorsque les performances sont meilleures, ce gain en performances reste marginal.

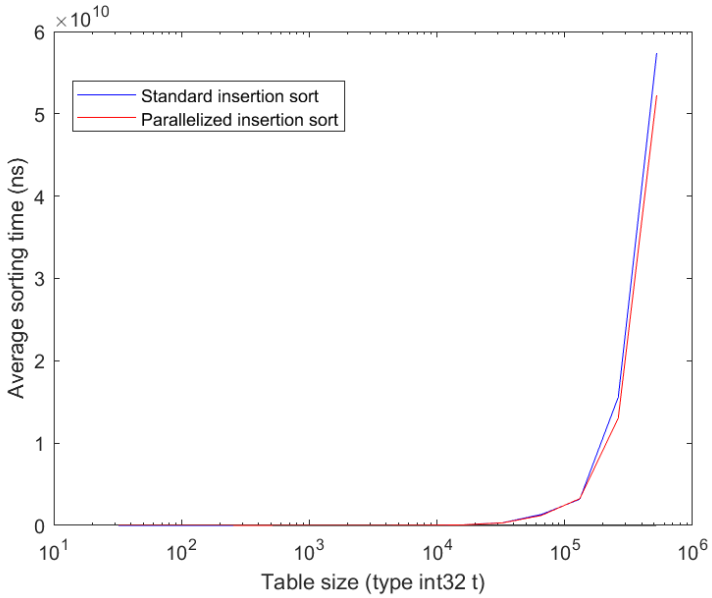

Figure 7 Graphique montrant les performances de la parallélisation du tri par insertion

De plus, cette optimisation par la parallélisation nuit à la performance du tri des tableaux de petite taille. Comme le montre la figure ci-dessous, pour des tableaux de taille inférieure à 8 192, le tri par insertion parallélisé est plus lent que le tri par insertion standard. La mise en place de threads par l'API OpenMP est à l'origine de ce phénomène.

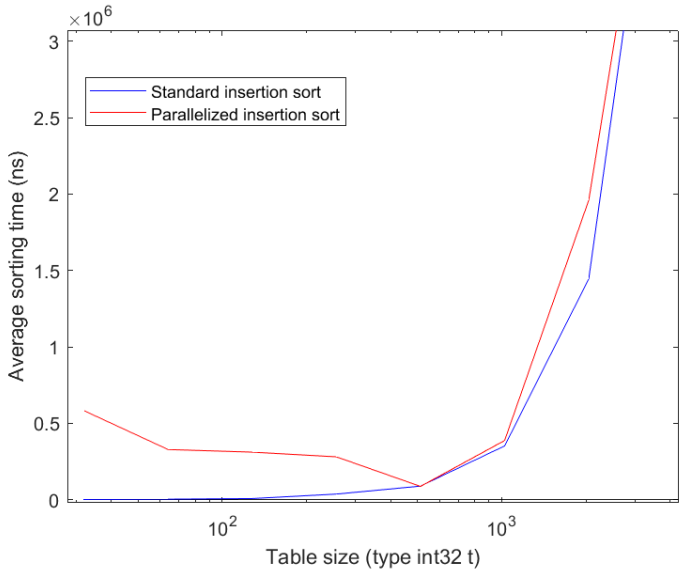

Figure 8 Graphique montrant la perte en performances de la parallélisation du tri par insertion pour des tableaux de faible taille.

Comparaison avec autres algorithmes de tri

Malgré les optimisations réalisées, on observe que le temps d'exécution de notre algorithme de tri par insertion reste très supérieur à d'autres algorithmes de tri tel que le tri radix (voir figure ci-dessous).

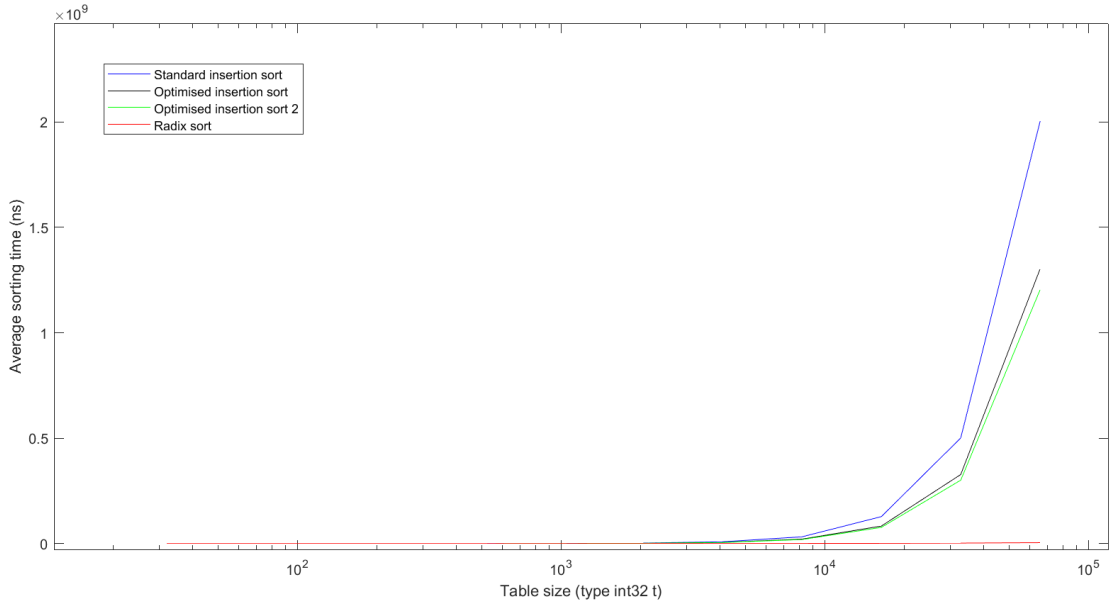

Figure 9 Graphique comparant les performances des algorithmes de tri par insertion optimisées et du tri radix.

En effet, la perte en performances du tri par insertion augmente drastiquement en fonction de la taille du tableau, car le tri par insertion possède tout de même une complexité quadratique. Cette divergence en termes de performances est visible à partir de tableaux de taille supérieure à 10^3 éléments.

3. Optimisations matérielles :

Dans cette partie, on s'intéresse à l'outil Vitis HLS de Xilinx qui permet de faire du codesign hardware/software. Cet outil permet de réaliser la synthèse d'une architecture en VHDL à partir de notre algorithme de tri en langage C. Pour guider l'outil vers une architecture optimisant les performances ou le nombre de ressources utilisées, on utilisera notamment des ''pragma''. Les pragma sont des directives données à Vitis HLS pour effectuer la synthèse RTL.

Le but est donc d'implémenter cet algorithme de tri sur puce FPGA avec une horloge de 100MHz. Le FPGA choisi est le ''xc7a100t-csg324-1'' présent sur la carte d'évaluation Artix-7. Il dispose de 63 400 LUTs (6-input), 126 800 Flip-Flops, et 4 860 Kb en BRAM. On réalise des tri de tableaux de 128 entiers de type int32_t. Pour cela, on synthétise l'architecture représentée sur le schéma ci-dessous. Les données sont communiquées entre le PC et le FPGA par liaison série UART à l'aide des blocs RCV et SND. Ces octets échangés par le PC sont convertis en données 32 bits à l'aide des blocs BYTES_TO_WORDS et WORDS_TO_BYTES. Les données 32 bits sont stockées dans le block DATA_SORTER qui trie le tableau de données reçues par UART_RX, puis le renvoi vers la liaison UART_TX.

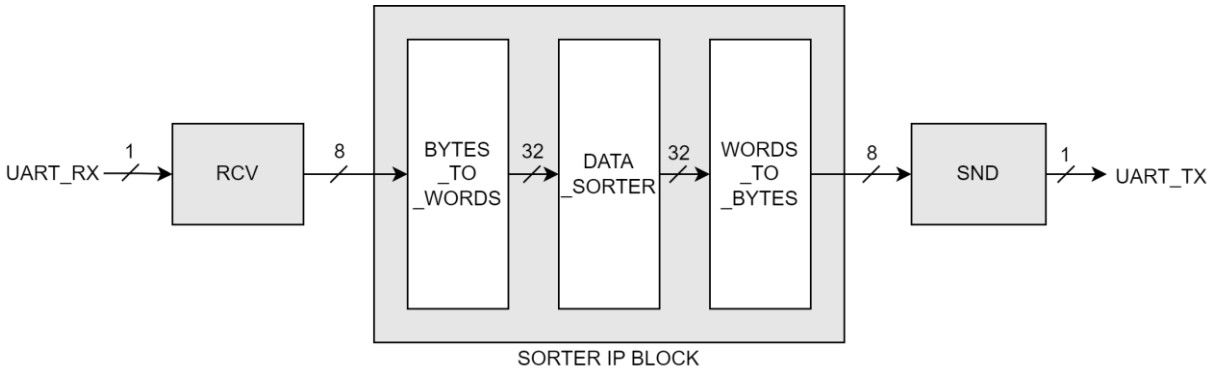

Figure 10 Schéma simplifié de l'architecture de notre système de tri sur FPGA avec block IP.

a. Synthèse de la fonction de tri naïve

Dans un premier temps, on réalise la synthèse de notre architecture avec la fonction de tri naïve, ne présentant aucune optimisation. Les résultats de cette synthèse serviront alors de modèle de référence pour évaluer ensuite les optimisations.

On obtient alors les résultats de synthèse indiqués ci-dessous.

Modules & Loops	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
stream_sorter	50322	5.030E5	-	50323	-	no	2	0	448	857	0
stream_sorter_Pipeline_BYTES2WORDS_LOOP	514	5.140E3	-	514	-	no	0	0	46	107	0
stream_sorter_Pipeline_receive_table	130	1.300E3	-	130	-	no	0	0	19	75	0
stream_sorter_Pipeline_send_sorted_table	130	1.300E3	-	130	-	no	0	0	11	75	0
stream_sorter_Pipeline_VITIS_LOOP_177_1	383	3.830E3	-	383	-	no	0	0	100	128	0
stream_sorter_Pipeline_WORDS2BYTES_LOOP	514	5.140E3	-	514	-	no	0	0	39	111	0
insertion_sort	49022	4.900E5	386	-	127	no	-	-	-	-	-

Figure 11 Résultats de synthèse HLS du tri insertion naïf

La réception et l'envoi des données du tableau nécessitent chacuns 130 cycles. Le tri est représenté par la boucle « insertion_sort » qui possède une latence de 49 022 cycles. La latence totale du tri est assez élevée, elle impose un tri du tableau en 493 us. Cependant le nombre de ressources utilisées

est très faible pour la puce FPGA choisie, on utilise 1% des LUTs et moins de 1% des bascules Flip-Flops disponibles. Les 128 données du tableau sont stockées sur 2 block RAM (BRAM) du FPGA.

b. Parallélisation de l'accès à la mémoire: pragma array_partition

Une limitation de notre architecture de tri naïve est l'utilisation de BRAM pour stocker les données du tableau à trier. En effet, cette décision permet de limiter le nombre de ressources nécessaire pour stocker le tableau. Cependant, l'accès séquentiel aux données de la mémoire augmente significativement la latence de notre architecture. Le tri par insertion doit réaliser de nombreux accès à la mémoire en lecture puis écriture (permutations 2 à 2).

Un moyen d'optimiser les performances de notre algorithme de tri consiste donc à paralléliser l'accès à la mémoire. Cette stratégie vise à augmenter la latence en réalisant des accès simultanés aux données du tableau qui ne sont plus stockées dans une seule mémoire. Pour cela, on utilise le pragma : « #pramga HLS ARRAY_PARTITION dim=1 type=complete variable=table ». Ainsi les données du tableau à trier sont stockées dans la logique du FPGA avec les bascules Flip-Flop et LUT, et permettent un accès simultanée.

Figure 12 Illustration du fonctionnement du pragma array_partition complete

On obtient alors les résultats de synthèse ci-dessous :

Modules & Loops	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
stream_sorter	18064	1.810E5	-	18065	-	no	0	0	16846	7504	0
stream_sorter_Pipeline_BYTES2WORDS_LOOP	514	5.140E3	-	514	-	no	0	0	46	107	0
stream_sorter_Pipeline_receive_table	130	1.300E3	-	130	-	no	0	0	4114	75	0
stream_sorter_Pipeline_VITIS_LOOP_217_1	129	1.290E3	-	129	-	no	0	0	4179	5028	0
stream_sorter_Pipeline_send_sorted_table	130	1.300E3	-	130	-	no	0	0	43	748	0
stream_sorter_Pipeline_WORDS2BYTES_LOOP	514	5.140E3	-	514	-	no	0	0	39	111	0
insertion_sort	16764	1.680E5	132	-	127	no	-	-	-	-	

Figure 13 Résultats de synthèse HLS pour le tri insertion naïf optimisé

La réception et l'envoi des données du tableau nécessitent chacune 130 cycles. Le tri possède à présent une latence de 16 764 cycles. On remarque une diminution de la latence de 66% par rapport à la synthèse précédente qui n'utilisait pas de pragma. Ceci permet un tri plus rapide du tableau en 170 us. Cependant, le nombre de ressources utilisées est considérablement supérieur(13% des Flip-Flops et 11% LUTs). Ceci est cohérent avec notre stratégie de stockage du tableau sans BRAM.

c. Approche matérielle de la parallélisation du tri par insertion

On souhaite à présent utiliser une autre approche matérielle pour réaliser un tri par insertion parallélisé sur puce FPGA. Ce tri par insertion débute dès la réception des premiers éléments du tableau par FPGA. En effet, ce tri ne nécessite pas le stockage de l'intégralité du tableau pour commencer son exécution.

Principe

Ce tri d'un tableau de taille N utilise un ensemble de N cellules qui sont placées en série. Chaque cellule possède une entrée 32bits connectée à la cellule précédente, et une sortie 32bits connectée à la cellule suivante. Par exemple, l'entrée de la cellule 1 correspond à la sortie de la cellule 0, et la sortie de la cellule 1 correspond à l'entrée de la cellule 2 (voir schéma ci-dessous). A chaque cellule i, on associe aussi une valeur enregistrée table_sorted[i], qui correspondra à la valeur trié du tableau en fin d'exécution.

Le tri débute lors de la réception de la 1ère donnée du tableau sur la file (FIFO). Celle-ci est enregistrée sur la cellule 0. Ensuite, lors de la réception de la 2ème donnée à trier, celle-ci est affectée en entrée à la cellule 0. La cellule 0 compare alors ces 2 données, et fournit en sortie la donnée supérieure, puis enregistre la donnée inférieure. De cette façon, parmi les premières 2 données à trier, la valeur supérieure se propage vers la cellule 1. Il est en de même pour le reste de données. Celles-ci sont introduites séquentiellement dans le système de cellules par la cellule 0, lors de la réception. Ensuite, elles se propagent au fil des cellules pour obtenir un tableau table_sorted trié dans l'ordre croissant en fin d'exécution. Les données de valeur plus importante se propagent vers la cellule N-1, et les données de valeur faible restent vers la cellule 0.

Lorsque toutes les données ont été reçues et introduites dans le système, la donnée table_sorted[0] de la cellule 0 correspond alors au minimum et peut être renvoyée comme 1ère valeur du tableau trié. Pendant ce temps, les données continuent à se propager sur les autres cellules pour finir le tri. Ainsi, on envoie séquentiellement les données 0 à N-1 du tableau trié.

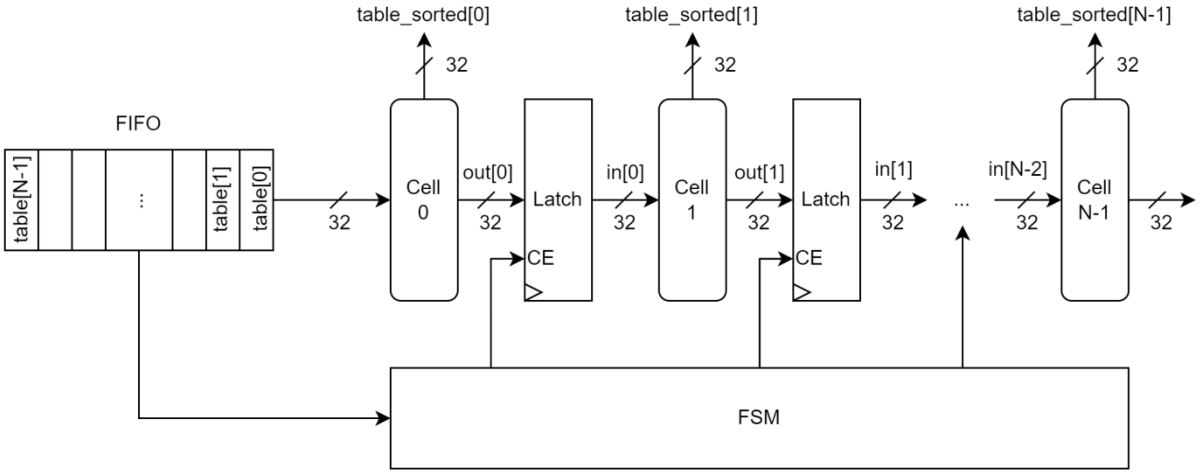

Figure 14 Schéma représentant le système de cellules pour le tri du tableau à partir d'une file de données

Ce fonctionnement correspond bien au tri par insertion parallélisé. Pour un tableau étant complétement trié sauf pour un élément i, les données i+1 à N-1 sont décalées par propagation tout au long des cellules. L'élément i est ensuite inséré au bon emplacement pour obtenir un tableau trié

qui se propage alors jusqu'à atteindre la cellule N-1. Dans le cas d'un tableau aléatoire, on a ce même phénomène de tri par insertion qui a lieu avec les données qui sont insérées de manière parallèle.

La cellule vue sur le schéma précédent peut être conçue comme indiquée sur le schéma ci-dessous. Elle est composée principalement d'un comparateur 32 bits, bascules permettant d'enregistrer la valeur table_sorted[i] correspondant à la donnée triée du tableau, et des multiplexeurs permettant la propagation des données à trier.

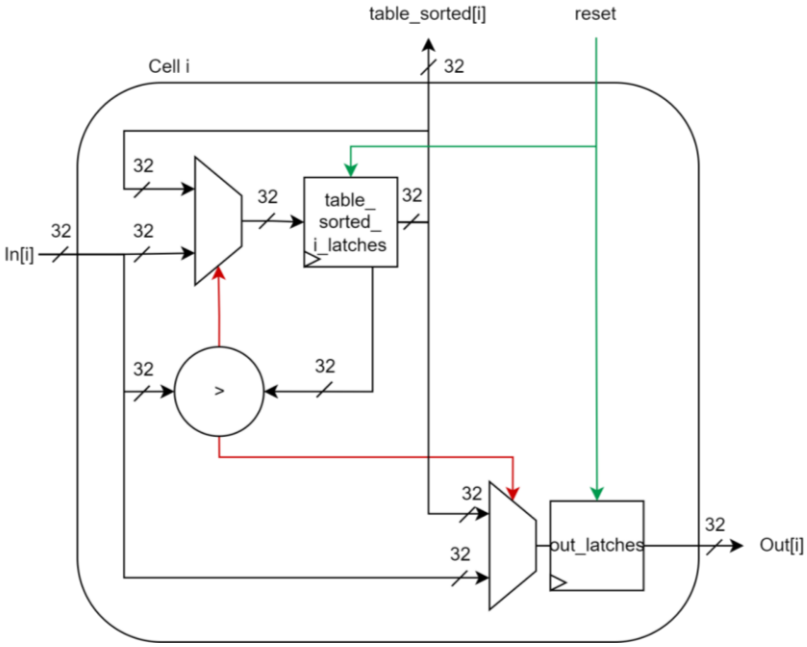

Figure 15 Schéma de la cellule

Synthèse

Bien que ce système puisse être réalisé avec le langage de description VHDL, on a souhaité utiliser le Vitis HLS pour synthétiser une architecture fournissant le même fonctionnement décrit précédemment. En effet, on décrit ce même fonctionnement en langage C, et on utilise les pragma adaptés pour guider la synthèse faite par l'outil. Dans ce cas, il s'agit des pragma « array partition » et « unroll ». Après la vérification du code de tri écrit langage C, on réalise la synthèse correspondante avec Vitis HLS. Le fonctionnement de ce bloc IP de tri a été vérifié sur la carte d'évaluation.

Modules & Loops	Latency(cycles)	Latency(ns)	Interval	Pipelined	BRAM	DSP	FF	LUT	URAM
stream_sorter	1300	1.300E4	1301	no	0	0	16680	21344	0
stream_sorter_Pipeline_BYTES2WORDS_LOOP	514	5.140E3	514	no	0	0	46	107	0
data_sorter_insertion_FIFO_parallel_final_int_128_s	265	2.650E3	265	no	0	0	16386	20906	0
data_sorter_insertion_FIFO_parallel_final_int_128_Pipeline_insert_table	130	1.300E3	130	no	0	0	8206	2452	0
data_sorter_insertion_FIFO_parallel_final_int_128_Pipeline_queue_sort	130	1.300E3	130	no	0	0	8140	3025	0
stream_sorter_Pipeline_WORDS2BYTES_LOOP	514	5.140E3	514	no	0	0	39	111	0

Figure 16 Résultats de synthèse HLS du système de cellules permettant le tri par insertion parallélisé

Cette approche nécessite un nombre de ressources plus important que l'optimisation matérielle précédente. Cependant celui-ci reste comparable et implémentable sur la cible FPGA. On obtient une architecture avec des performances significativement améliorées, le tri est réalisé en seulement 265 cycles.

d. Comparaison du tri par insertion matériel avec Vitis HLS

Le tableau ci-dessous rassemble les résultats de synthèse des architectures étudiées dans les parties précédentes. Ces résultats sont aussi illustrés sur le graphique ci-après.

Type de tri	Latence	Nombre de ressources
a) Tri naïf	130 (receive_table) +49 022 (insertion_sort) +130 (send_sorted_table) =49 282 cycles	• 448 FF(<1%) • 857 LUT(1%) • 2 BRAM
b) Tri avec partition complète du tableau	130 (receive_table) +16 764 (insertion_sort) +130 (send_sorted_table) =17 024 cycles	• 16 846 FF(13%) • 7 504 LUT(11%)
c) Tri parallélisé par système de cellules	130 (insertion_tab) +130 (queu_sort) +5 =265 cycles	• 16 680 FF (12%) • 21 344 LUT (32%)

Tableau 1 Comparaison des architectures de tri par insertion pour un tableau de 128 données en int32_t, clock 100MHz, FPGA xc7a100t-csg324-1

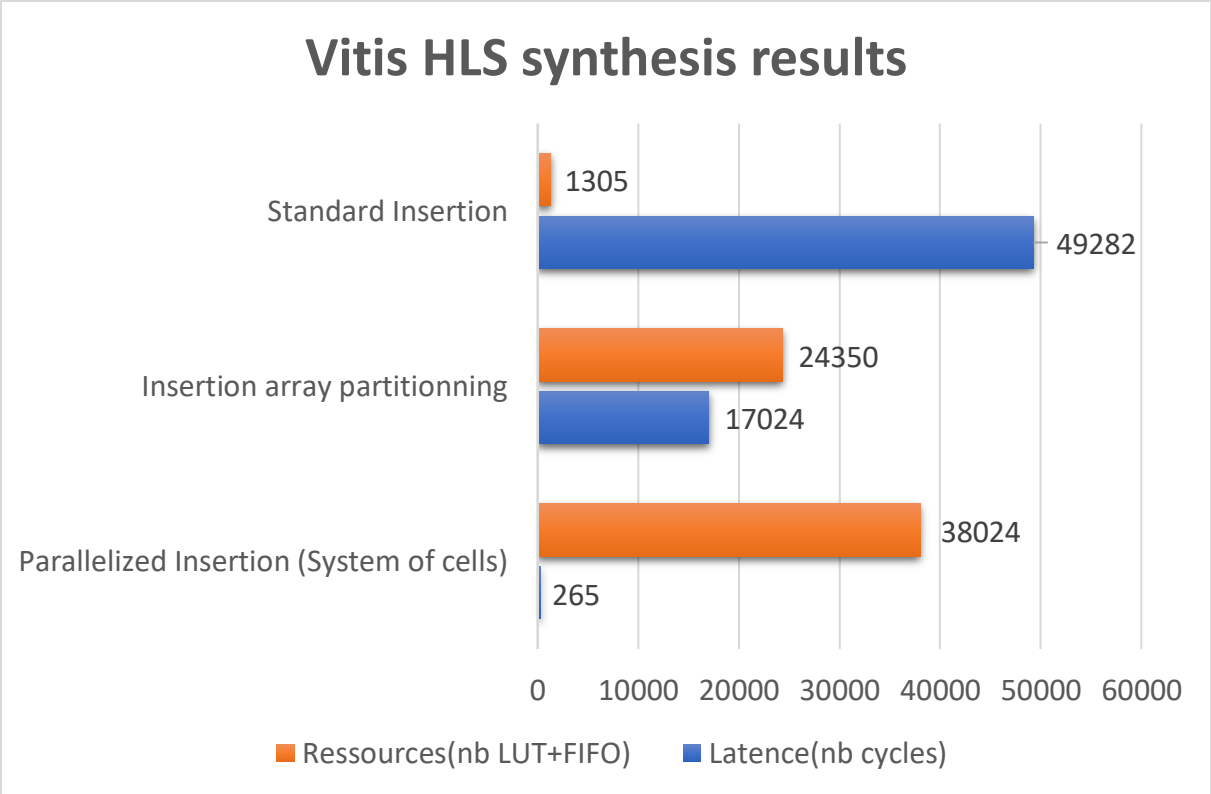

Figure 17 Graphique représentant la latence et le nombre de ressources utilisées pour les 3 implémentations étudiées

On observe un compromis entre le nombre de ressources utilisées pour synthétiser l'architecture et la latence du système. Dans un premier temps, Vitis HLS synthétise une architecture qui optimise le nombre de ressources utilisées avec une version standard du tri par insertion. En effet, celle-ci utilise des BRAM pour le stockage du tableau. La deuxième architecture avec le pragma array partition permet de se franchir de la limitation temporelle imposée par l'accès séquentiel à une

mémoire. Les données du tableau sont alors accessibles de manière simultanée. Ainsi, la latence de ce système diminue de 66%, mais on utilise plus de 18 fois plus de ressources matérielles LUT et Flip-Flops (pour le stockage du tableau). Finalement, la dernière approche utilisant un système de cellules comparatrices permet une diminution très significative du temps nécessaire pour trier le tableau. Cette approche permettant de paralléliser le tri par insertion offre de meilleures performances. Cependant ce système plus complexe, nécessitant une description plus précise de l'architecture par code C, nécessite davantage d'éléments logiques pour sa synthèse.

Tri à bulle

1. Principe du tri à bulle :

Le tri à bulle consiste à comparer répétitivement 2 éléments consécutifs du tableau et à les permuter s'ils sont dans le désordre. Voici un code en C de l'algorithme de tri à bulle.

```
for(int32_t y = 0; y < N - 1; y += 1 )
{
    last = table[0];
    for(int32_t x = 1; x < N; x += 1 )
    {
        newv = table[x];
        if( last > newv )
        {
            table[x-1] = newv;
        }else{
            table[x-1] = last;
            last       = newv;
        }
    }
    table[MAX_SIZE-1] = last;
}
```

Figure 18 : Code C du tri à bulle pour un tableau de taille N

L'algorithme est composé de deux boucles imbriquées. La boucle externe indique le nombre de fois qu'il faut parcourir le tableau (N-1 fois pour un tableau de taille N). Dans la boucle interne, on parcourt le tableau en permutant les éléments 2 à 2 lorsqu'ils sont désordonnés.

<u>Complexité de l'algorithme :</u>

Le nombre maximal de comparaisons pour le tri d'un tableau de taille n est de $C(n) = \sum_{i=0}^{n-2}\sum_{j=1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-1) = (n-1)^2$. Ainsi, la complexité du tri par insertion est O(n²).

2. Optimisations matérielles sur Vitis HLS :

a. Synthèse de la fonction de tri à bulle naïve.

Comme dans la partie précédente, on réalise dans un premier temps la synthèse de notre architecture avec la fonction de tri naïve avec Vitis HLS. Les figures ci-dessous montrent les résultats de synthèse pour différentes tailles de tableau.

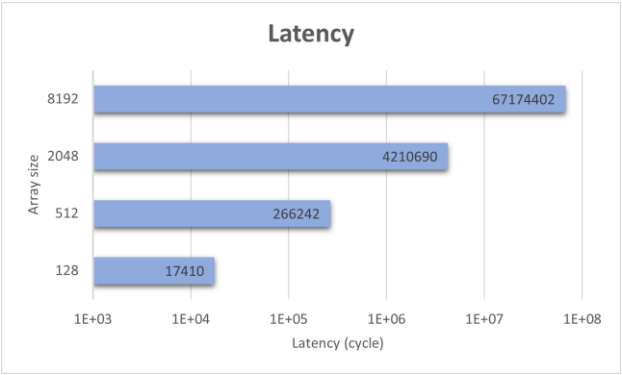

Figure 19 : Latence en fonction de la taille du tableau

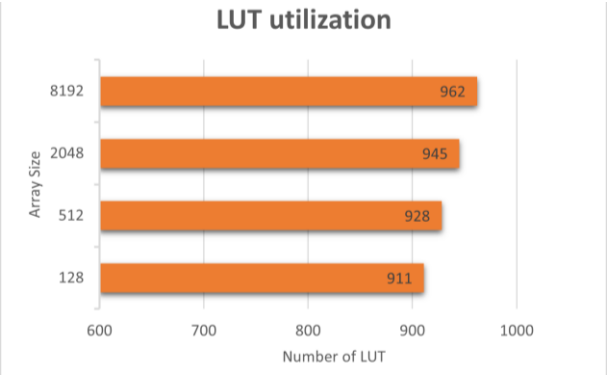

Figure 20 : LUTs utilisées en fonction de la taille du tableau

Le nombre de cycles necéssaires pour trier est quasiment égal au nombre de comparaisons. Il est donc très grand au vu de la complexité de l'algorithme. Le nombre de LUT reste quasiment constant quand la taille du tableau augmente (on utilise 1% des LUTs). Cette approche naïve utilise très peu de ressources.

b. Synthèse avec parallélisation du tri

Pour réduire le temps de tri, on peut adopter une approche diviser pour régner. On décide de diviser le tableau en plusieurs parties de même taille. On trie ensuite chaque sous-tableau avec l'algorithme de tri à bulle et on les fusionne en un unique tableau trié.

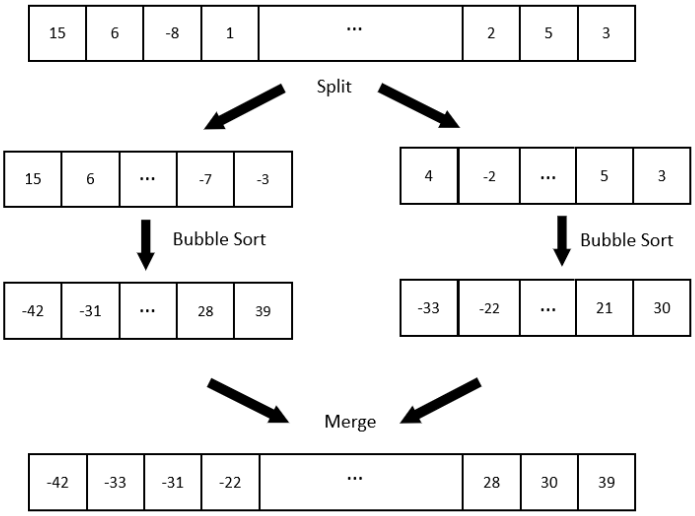

Figure 21 : Principe de l'optimisation du tri à bulle

Prenons l'exemple où l'on coupe le tableau en 2. Si n est la taille du tableau, on trie deux tableaux de taille n/2. Le nombre de comparaisons est alors $C(n) = 2 \times (\frac{n}{2} - 1)^2 \approx \frac{n^2}{2}$. Le nombre de comparaisons pour la fusion est linéaire et est donc négligé. On devrait alors gagner un facteur 2 sur la latence par rapport à la version naïve. Cependant il est possible de paralléliser les tris des deux sous tableaux, on gagnerait encore un facteur 2 sur la latence.

Pour paralléliser ces tris, il faut être en mesure d'effectuer plusieurs accès mémoires simultanément. Il faut donc stocker nos sous-tableaux dans différents blocs mémoires. Pour cela, on utilise le pragma suivant dans notre code C :

```
#pragma HLS ARRAY_PARTITION variable=table1 type=block factor=2
```

Dans cet exemple, le tri du tableau devrait alors être 4 fois plus rapide que dans la version naïve.

Nous avons fait la synthèse de 2 algorithmes avec cette approche. Celui vu dans l'exemple ci-dessus et un qui le coupe le tableau en 4. En suivant le même raisonnement, on s'attend à ce que ce dernier algorithme soit 16 fois plus rapide que la version naïve. Lorsqu'on a 4 sous tableaux, l'étape de fusion se fait en plusieurs étapes comme sur la figure ci-dessous.

Pour réaliser les étapes de fusion on utilise une zone mémoire de la même taille que le tableau d'entrée, on fait alors des fusions successivement d'un tableau à l'autre (voir figure ci-dessous).

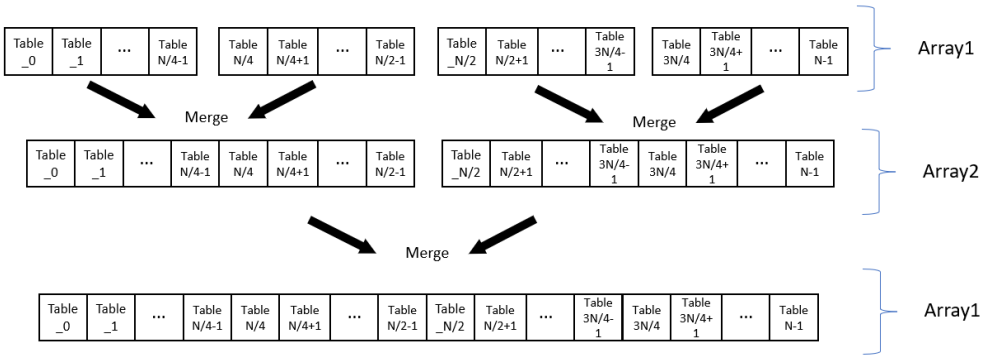

Figure 22 : Fusion de 4 sous-tableaux

Voici les résultats de synthèse obtenus avec Vitis HLS.

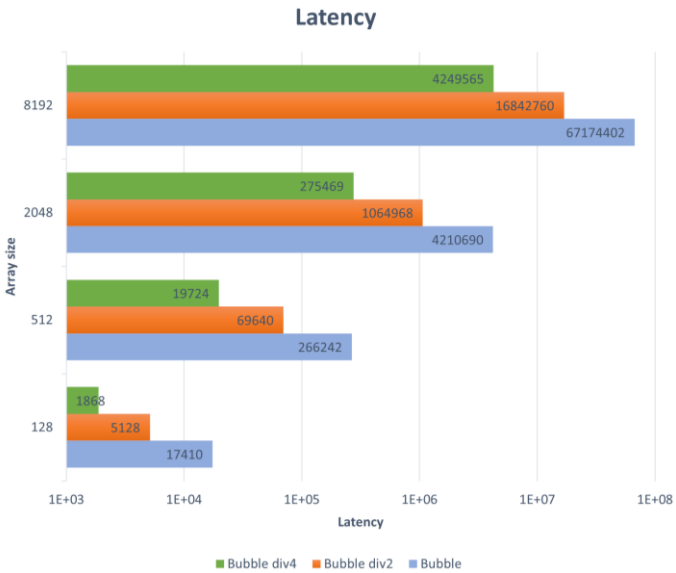

Figure 23 : Latence en fonction de la taille du tableau

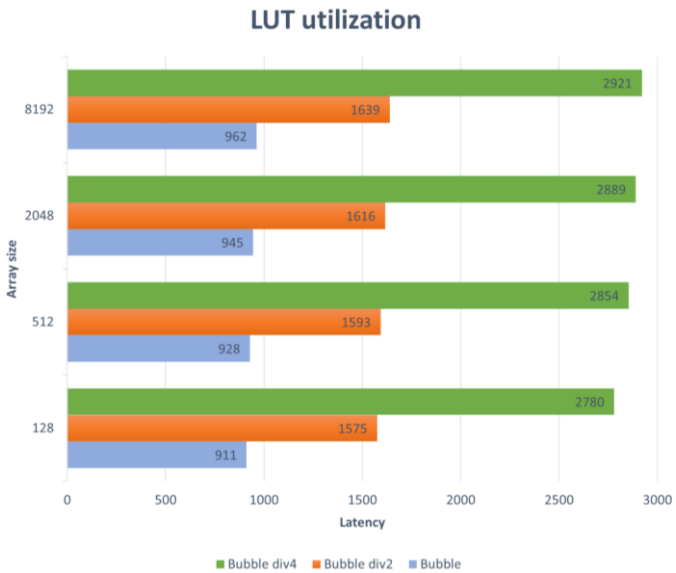

Figure 24 : nombre de LUTs en fonction de la taille du tableau

On observe bien la diminution de la latence comme attendue. Cependant on utilise plus de ressources avec cette approche. En effet, on veut trier les sous-tableaux en parallèle, il faut donc instancier autant de « trieur à bulle » que l'on a de sous tableaux.

3. Implémentation complète en VHDL

Par ailleurs, on a travaillé sur l'implémentation complète du tri en VHDL sur Vivado, afin de comparer les performances avec la solution obtenue sur Vitis HLS.

a. Description de l'architecture

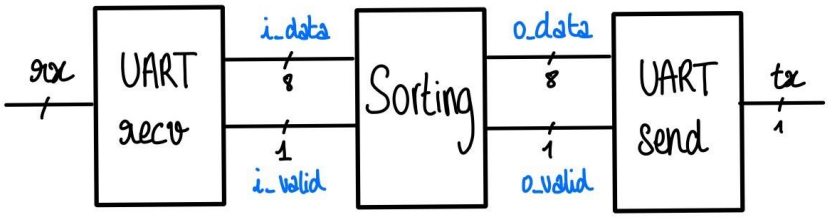

Figure 25 : Schéma global de l'implémentation hardware du tri bulle

Les données à trier sont reçus via le port UART de la carte NEXYS4. Elles sont triées dans le bloc sorting. Puis, les données triées sont renvoyées par le port UART. On utilise les modules de M.Bornat pour la communication UART (**UART_recv** et **UART_fifo_send**).

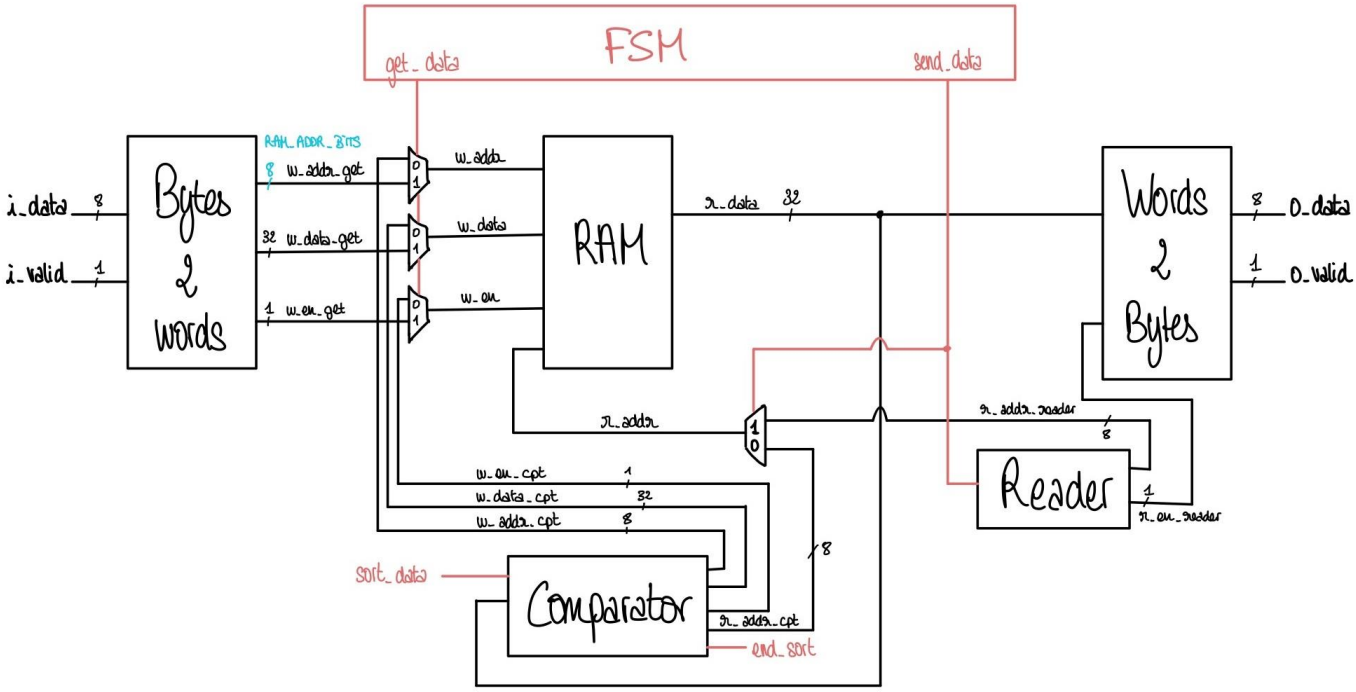

Figure 26 : Schéma du bloc Sorting

Concernant la partie pour trier, elle est composée de plusieurs blocs dont la description est ci-dessous:

- **Bytes_to_words** est le module en entrée qui reçoit les données à trier via le module UART Rx. Cependant, **UART_recv** renvoie des datas sur 8 bits. Le rôle de ce module est donc de reconstitué la donnée sur 32 bits.

- **Words_to_bytes** est le module en sortie complémentaire à **Bytes_to_words**. Il envoie les données sur 32 bits par paquet de 8 bits.

- **RAM** est la mémoire dans laquelle les données sont stockées.

- **Comparator** est le bloc représentant l'algorithme de tri bulle.

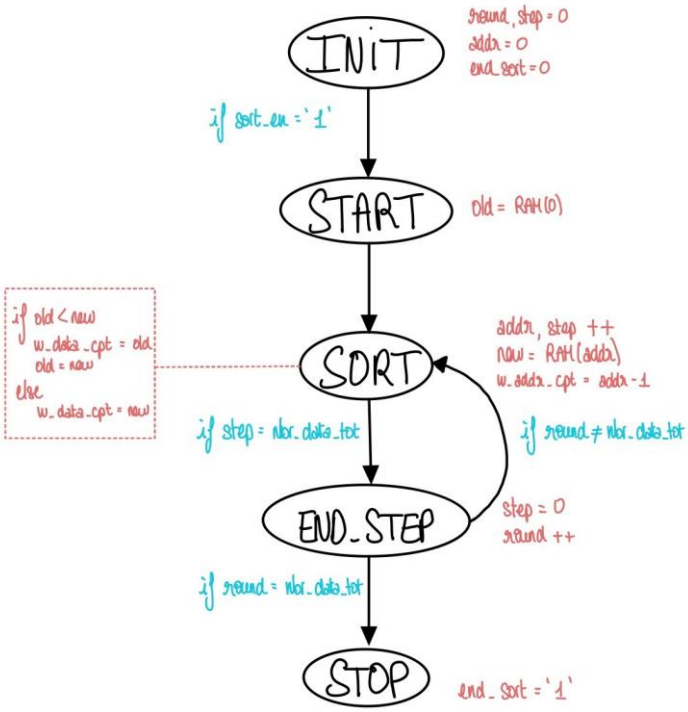

Figure 27 : Schéma du module Comparator

- **Reader** est utilisé pour l'envoi des données triées. Il permet de lire dans la mémoire les datas en envoyant l'adresse de la donnée à lire.

- **FSM** est la machine d'état qui ordonne les tâches à effectuer. Il y a quatre étapes représentées sur le schéma suivant :

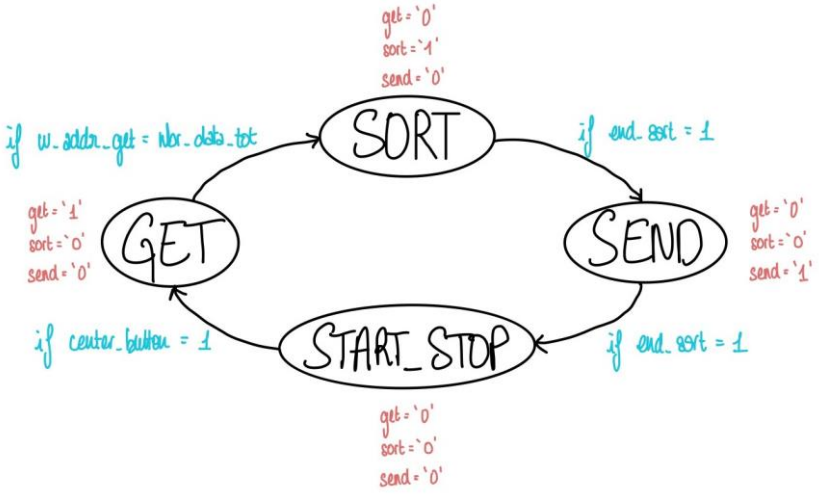

Figure 28 : Schéma de la FSM

- Durant l'état GET, les données à trier sont stockées dans la mémoire **RAM**.
- Les données sont triées lors de l'étape SORT selon l'algorithme de tri choisi, ici le tri bulle.
- Une fois le tri terminé, il y a l'état SEND. On va lire les données triées dans le RAM et les envoyées.

b. Analyse des performances

On a comparé les performances du tri bulle sur Vivado avec celles sur Vitis. On a relevé le temps de tri ainsi que le nombre de ressources utilisées pour le tri de 128 données.

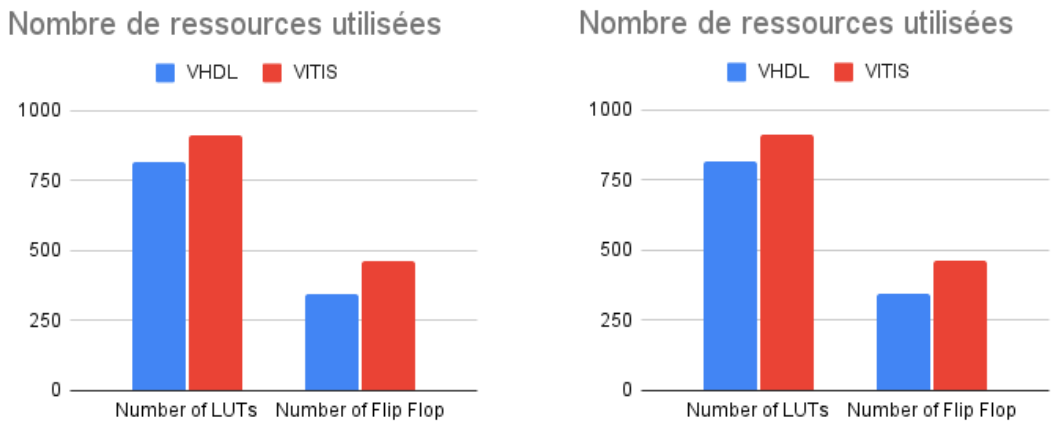

Figure 29 : Comparaison des performances entre Vivado et Vitis pour un tri de 128 données

On remarque que l'algorithme sur Vivado utilise moins de ressources matérielles.

Le tri bulle nécessite environ 17 000 cycles d'horloge pour les deux programmes. Cependant, la fréquence de l'horloge sur Vitis est supérieure.

Les performances de l'implémentation complète en VHDL ne sont pas meilleures que celles de Vitis. Son développement est en plus beaucoup plus long. Ainsi, l'implémentation complète en VHDL de l'algorithme de tri bulle n'est pas rentable comparé au développement sur Vitis.

Aucune optimisation n'a été apporté sur Vivado par manque de temps.

Tri Fusion

Nous avons vu que plus on divise le tableau plus on réduit le nombre de comparaisons. Le principe du tri fusion est de diviser le tableau le plus possible, jusqu'à obtenir des sous-tableaux de taille 1. On fusionne alors ces sous-tableaux en plusieurs étapes pour construire le tableaux trié (voir figure ci-dessous). Le nombre de comparaisons est de l'ordre de $N \times log(N)$.

Pour réaliser ce tri, on utilise un deuxième tableau de la même taille que le tableau d'entrée, on fait alors des fusions successivement d'un tableau à l'autre (voir figure ci-dessous), cette méthode simplifie grandement l'implémentation de l'algorithme mais utilise deux fois plus mémoire qu'un tri en place.

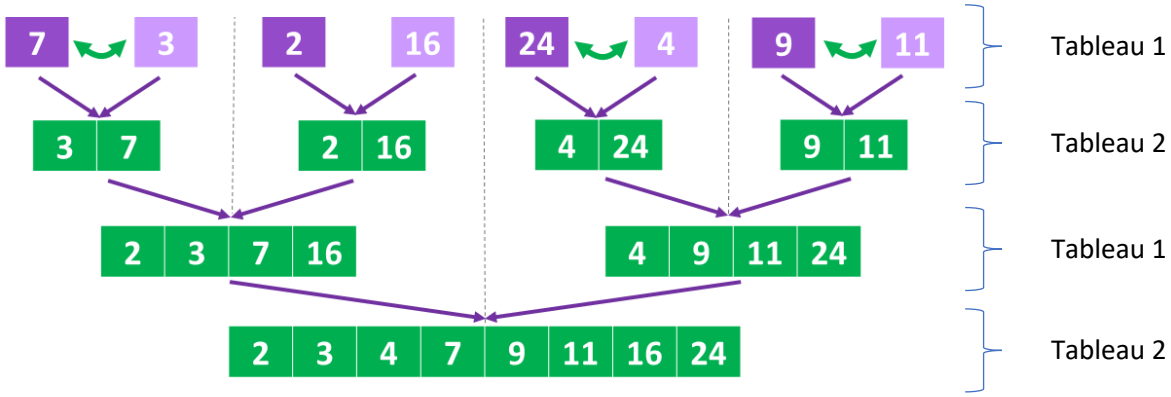

Figure 30 : Principe du tri fusion

Voici les résultats de synthèse obtenus avec Vitis HLS :

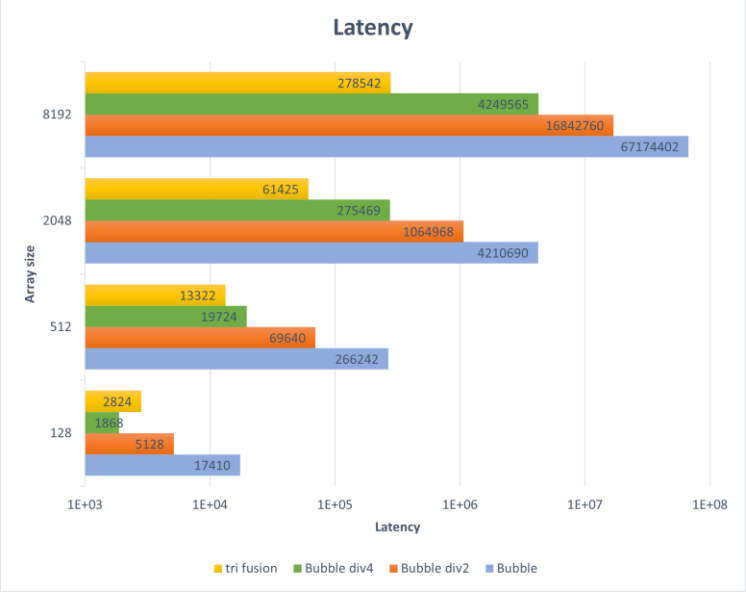

Figure 31 : Latence en fonction de la taille du tableau

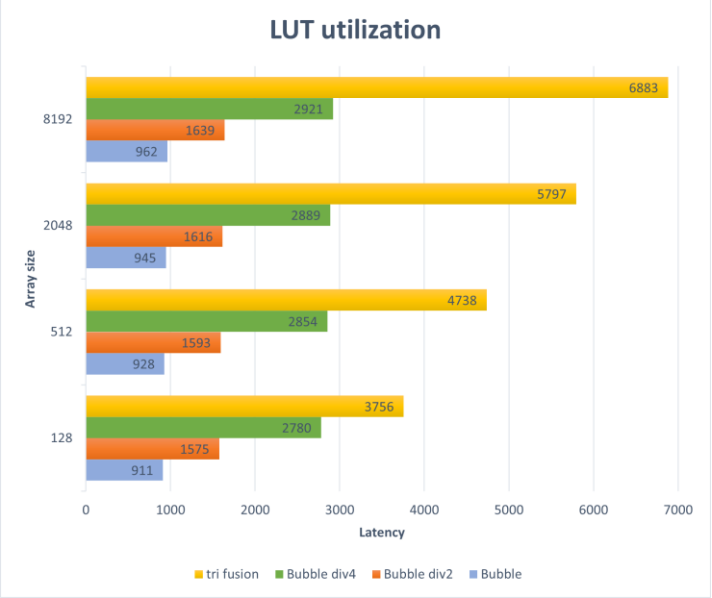

Figure 32 : nombre de LUTs en fonction de la taille du tableau

On observe une forte diminution de latence pour les grands tableaux. L'évolution de la latence lorsque la taille du tableau augmente est en accord avec la complexité de l'algorithme. Les ressources utilisées sont plus importantes que pour les tris à bulle et dépendent de la taille du tableau.

Une piste d'amélioration de cette implémentation est de paralléliser les fusions des sous-tableaux. Cela réduirait fortement le temps de tri.

Tri radix binaire

1. Principe :

Le tri radix binaire consiste à séparer les valeurs d'un tableau en deux sous-tableau en considérant uniquement leur bit de poids fort. Le premier sous-tableau contient les valeurs ayant un bit de poids fort à 1 et le deuxième ayant leur bit de poids fort à 0, le premier tableau contient les valeurs hautes et le deuxième contient les valeurs basses. Ensuite, chaque sous-tableau est séparé en deux sous-sous-tableau de la même manière selon le deuxième bit de poids fort. Le même procédé est réalisé pour chaque bits des valeurs du tableau. A la fin de l'algorithme, les valeurs sont tirées par ordre croissant.

Cet algorithme fonctionne pour des valeurs entières non-signées. Pour l'utiliser sur des valeurs signées, il suffit d'échanger les premiers sous-tableaux (bit de singe à 0 au-dessus de bit de signe à 1) puis de continuer l'algorithme de la même manière. Il en va de même pour des nombres flottants de type float ou double.

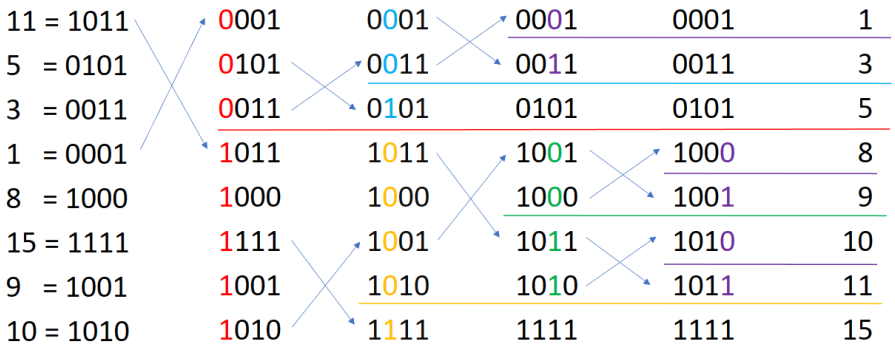

Figure 33 : Exemple de tri radix binaire appliqué à un tableau de valeurs sur 4 bits

L'avantage de cet algorithme est qu'il présente une complexité algorithmique linéaire. En effet, pour un tableau de N valeurs codées sur n bits, chaque valeur sera testée uniquement n fois. On a donc N*n comparaisons au total.

L'inconvénient de cet algorithme est que la taille de chacun des sous-tableaux créés ne peut pas être connues à l'avance, elle dépend des valeurs, ce qui complique l'implémentation matérielle.

2. Implémentation logicielle :

a. Algorithme récursif

Afin de valider l'algorithme et d'en estimer les performances, une version logicielle récursive a d'abord été développée. Celle-ci se présente en une fonction récursive recevant le pointeur du tableau à trier, sa taille, ainsi qu'un masque. A l'itération i, le masque ne contient que le $i^{ème}$ bit de

poids fort à un. La fonction compare alors chaque valeur depuis le début à la recherche de la première valeur ayant un 1 au $i^{ème}$ bit de poids fort. Une fois cette valeur trouver, elle part de la fin du tableau pour trouver la dernière valeur ayant un 0 au $i^{ème}$ bit de poids fort. Ces deux valeurs sont interverties puis le parcours du tableau continue en intervertissant les valeurs devant l'être. La fonction est ensuite appelée récursivement sur la première partie du tableau puis sur la deuxième en appliquant un décalage à droite au masque.

Cette fonction récursive est appelée par une pseudo-fonction ne prenant en argument que le pointeur du tableau et sa taille. Elle ne fait alors qu'appeler la fonction récursive en initialisant le masque avec uniquement un 1 au bit de poids fort.

Afin d'adapter cet algorithme à des valeurs entières signées, une autre pseudo-fonction a été développée. Elle tri d'abord le tableau selon le bit de signe puis elle fait appel à la fonction récursive pour entiers non-signés avec un masque décalé une première fois et en faisant un cast sur le tableau. Pour trier des nombres flottants, une dernière pseudo-fonction est utilisée. Elle appelle simplement la fonction de tri sur entiers signés en castant le tableau.

Afin d'améliorer les performances, la fonction récursive termine prématurément lorsque la taille du tableau à trier est inférieure à 2. En effet, il est alors plus rapide de comparer directement ces deux valeurs que de continuer à comparer chacun de leurs bits un à un. Et il est évidemment inutile de trier un tableau de 1 ou 0 valeur. Ces conditions d'arrêt supplémentaires permettent d'économiser des calculs inutiles facilement.

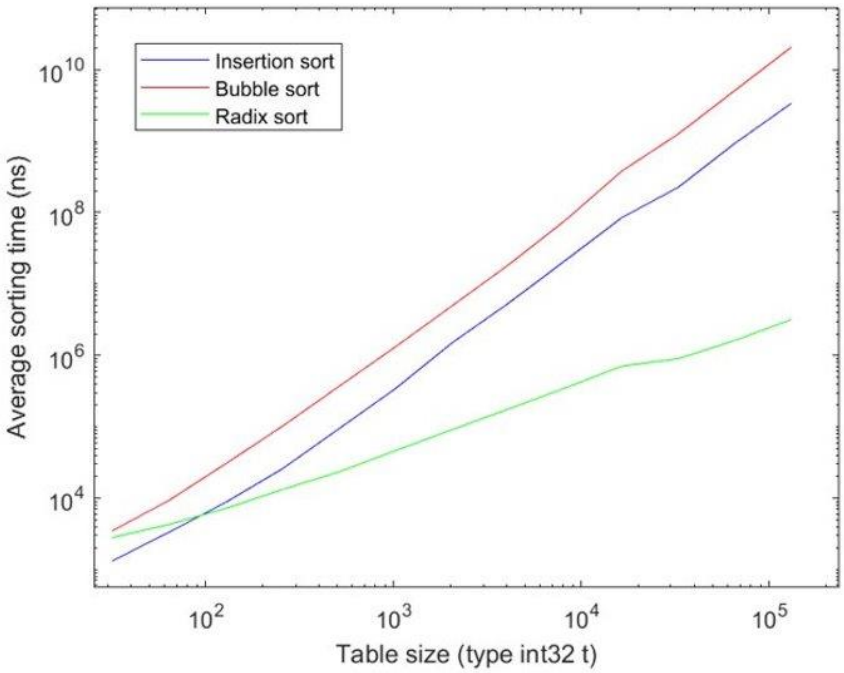

Figure 34 : Comparaison des performances entre les différents tris sur des tableaux de types int32_t

Le graphique ci-dessus présente les performances temporelles obtenues avec les tris par insertion, bulle et radix binaire réalisé avec son algorithme récursif. Comme le montre le graphique ci-dessus, le tri radix binaire est bien plus performant que le tri par insertion ou que le tri bulle, particulièrement pour les tableaux de grandes tailles.

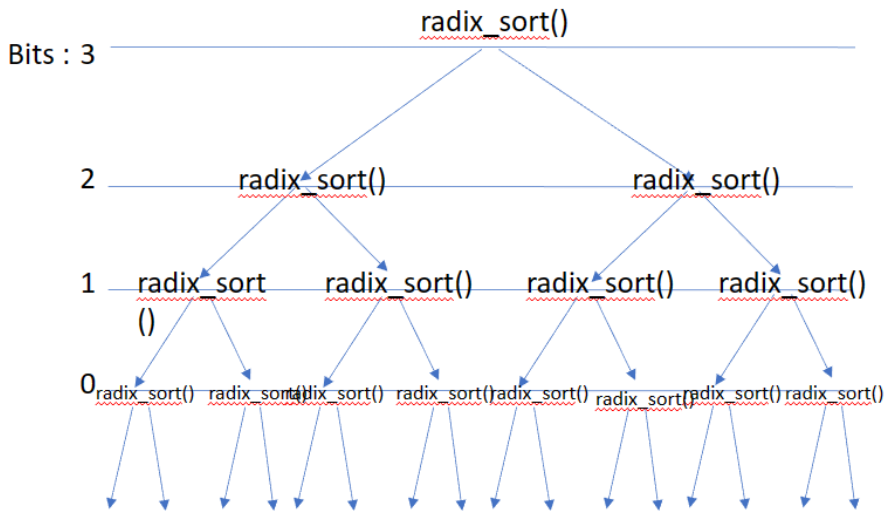

Figure 35 :Arbre d'appels récursifs du tri radix récursif

b. Algorithme itératif

L'algorithme récursif a permis de vérifier et de comparer les performances du tri radix binaire par rapports aux autres algorithmes développés par le groupe en implémentation logicielle. Le but est alors de pouvoir l'implémenter en matériel. Cependant, un algorithme récursif n'est pas synthétisable par l'outil Vitis HLS. Il est alors nécessaire de le décrire de manière itérative avant de le synthétiser en matériel.

Afin de décrire l'algorithme de manière itérative, synthétisable par Vitis HLS, le point de départ est l'algorithme récursif. Le but est alors de stocker toutes les variables nécessaires à simuler les appels récursifs dans une boucle. L'algorithme récursif a une profondeur maximum de 32 (pour des valeurs sur 32 bits). Il descend toujours l'arbre de d'appels par la gauche puis passe sur la branche de droite avant de remonter. Ainsi, il faut garder en mémoire le coté emprunté par chaque appel et les bornes du sous-tableau traité par chaque appel. Il y a donc 3 tableaux de 32 valeurs à conserver en mémoire : coté emprunté par l'appel, borne haute de l'appel, borne basse de l'appel. Cela ne représente donc pas un surcoût de mémoire trop conséquent.

C'est en partant de ce constat que nous avons tenté de traduire l'algorithme récursif en un algorithme itératif. Malheureusement, nous n'avons pas eu le temps de débugger l'algorithme itératif pour qu'il fonctionne correctement. Nous n'avons donc pas pu essayer de l'implémenter en matériel avec Vitis HLS.

Conclusion

Ce projet nous a permis d'étudier différentes méthodes d'optimisation du tri d'un tableau. Pour cela, on a étudié 4 algorithmes de tri : le tri par insertion, le tri à bulle, le tri fusion, et le tri radix binaire. Ces algorithmes de complexités différentes offrent des performances variées. Des optimisations logicielles et matérielles ont été réalisées pour réduire le temps d'exécution du tri avec ces algorithmes. Pour cela, on a utilisé divers outils tels que l'API OpenMP pour le multithreading, l'outil de synthèse Vitis HLS, et une description complète du système en langage VHDL.

Bibliographie

http://ijcset.com/docs/IJCSET13-04-05-068.pdf

https://faculty.cs.niu.edu/~mcmahon/CS241/Notes/Sorting_Algorithms/insertion_sort.html

https://cseweb.ucsd.edu/~jmatai/presentations/FPGA2016_Resolve.pdf

https://people.cs.rutgers.edu/~venugopa/parallel_summer2012/bitonic_overview.html