Filière Électronique - PR310

Rapport de projet

Algorithmes de tri

Élèves :
Jérémy CHESLET
Léo GUESNERIE
Antonin TRILLOT

Professeur :
Bertrand LE GAL
Jérémie CRENNE

Janvier 2022

Table des matières

1 Introduction

Dans le cadre du module PR310, un projet avancé de la spécialisation systèmes embarqués a été réalisé au cours de 10 séances de 3h. Le sujet était porté sur l'étude est l'optimisation d'algorithmes de tri de nombres sur différentes architectures. Il traitait particulièrement la parallélisation des algorithmes pour accélérer leurs exécutions. Le choix des objectifs, des algorithmes ainsi que des cibles matérielles nous étaient laissées en fonction de nos appétences. En effet, l'objectif de ce projet était avant tout d'explorer des solutions tris efficaces.

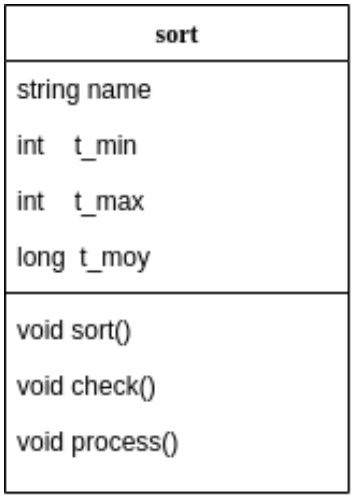
2 Première approche

La première chose que nous avons fait a été de regarder les différents algorithmes de tri existant. Nous les avons alors programmé afin de pouvoir les comparer entre eux.

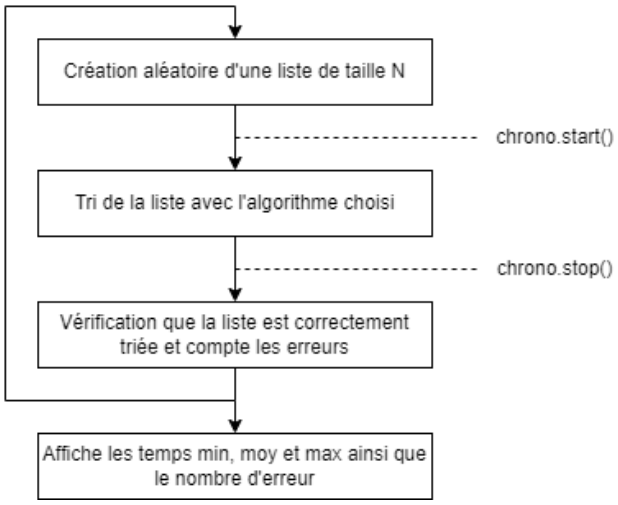
2.1 Protocole de test

Afin de comparer facilement les algorithmes de tri, nous avons créé en C++ un environnement de test. Cet environnement est basé sur une classe nommé `sort` contenant une méthode `sort` qui trie la liste, une méthode de vérification `check` et une méthode `process` qui est appelle les autres méthodes comme montré dans le schéma. Cette classe contient aussi des attributs: temps minimum, temps moyen et temps maximum. Ceux-ci sont mis à jour dans le process.

Le protocole que nous avons choisi fonctionne comme ceci. Premièrement, nous créons une liste de nombre entier (int codé sur 4 octets) de taille N. Cette taille est en argument de la méthode et peut donc être modifiée en fonction des expérimentations que nous voulons mener. Nous appelons ensuite la méthode `sort` afin de trier la liste. Nous mesurons aussi le temps d'exécution de ce tri et mettons à jour les valeurs de temps en conséquence. Une fois que la liste est triée, nous vérifions que les nombres de la liste de sortie sont bien positionnés du plus petit au plus grand. Si cela n'est pas vrai, on incrémente un compteur. On recommence ce process une cinquantaine de fois puis on affiche les résultats dans le terminal.



(a) Classe sort



(b) Fonctionnement du process

Ce protocole permet donc de valider le fonctionnement d'un tri pour une taille de liste donnée. Cela n'étant pas très pratique pour récupérer les résultats, nous avons créer des commandes permettant de faire appel facilement au protocole.

```
antonin@Nashi:~/Enseirb/Projet_S9/sorting-2021$ ./bin/main -h
Commande disponible
   -c [sort] [len_min] [len_max] [step] :variation de la taille de la liste a trier
   -a [len] : chaque sort passe pour une liste de taille len

       sort: 1- Bubble sort
       sort: 2- Bubble sort CPU
       sort: 3- Shaker sort
       sort: 4- Shaker sort CPU
       sort: 5- Merge sort
       sort: 6- Quick sort
       sort: 7- Quick sort CPU
       sort: 8- Quick sort SIMD
       sort: 9- std sort
```

FIGURE 2 – Commande disponible pour le benchmarking

La première commande $-c$ permet de calculer pour une type de tri les temps d'exécutions pour différentes tailles de liste. Nous choisissons alors l'algorithme, la taille minimal, la taille maximal ainsi que le pas et le programme nous donne les résultats.

```
antonin@Nashi:~/Enseirb/Projet_S9/sorting-2021$ ./bin/main -c 5 100 1000 200
Sorting with      Merge sort len = 100    ...      ended with : 0 error(s).
Sorting with      Merge sort len = 300    ...      ended with : 0 error(s).
Sorting with      Merge sort len = 500    ...      ended with : 0 error(s).
Sorting with      Merge sort len = 700    ...      ended with : 0 error(s).
Sorting with      Merge sort len = 900    ...      ended with : 0 error(s).


TAB OF LEN VARIATION FOR Merge sort
            LEN      T_MIN (ns)      T_MOY (ns)      T_MAX (ns)
            100           19009           24572           50017
            300           68591          122361          206396
            500          182478          225191          289321
            700          212592          261889          404541
            900          495934          536659          572602
```

FIGURE 3 – Commande -c

La seconde commande $-a$ permet de calculer les temps d'exécutions de tous les algorithmes implémentés pour une taille de liste que nous lui donnons.

```
antonin@Nashi:~/Enseirb/Projet_S9/sorting-2021$ ./bin/main -a 1000
Sorting with     Bubble sort len = 1000    ...      ended with : 0 error(s).
Sorting with Bubble sort CPU len = 1000    ...      ended with : 0 error(s).
Sorting with     Shaker sort len = 1000    ...      ended with : 0 error(s).
Sorting with Shaker sort CPU len = 1000    ...      ended with : 0 error(s).
Sorting with      Merge sort len = 1000    ...      ended with : 0 error(s).
Sorting with      Quick sort len = 1000    ...      ended with : 0 error(s).
Sorting with  Quick sort CPU len = 1000    ...      ended with : 0 error(s).
Sorting with Quick sort SIMD len = 1000    ...      ended with : 0 error(s).
Sorting with        Std sort len = 1000    ...      ended with : 0 error(s).


TAB OF SORT FOR LEN = 1000
            LEN          T_MIN (ns)          T_MOY (ns)          T_MAX (ns)
         Bubble sort             1169344             1400662             5594842
      Bubble sort CPU              198966              315281             4576847
         Shaker sort             1409516             1489889             2032787
      Shaker sort CPU             1632994             1854121             4207305
          Merge sort              103422              138793             1208618
          Quick sort               76650               88758              234198
       Quick sort CPU              755205              891093             3024446
      Quick sort SIMD               19912               20474               30808
            Std sort               40048               51985              208505
```

FIGURE 4 – Commande -h

2.2 Algorithmes étudiés

Tri à bulles

Un des algorithmes les plus simple à comprendre et à implémenter est le tri a bulle. Le principe de ce tri est assez simple: faire remonter vers la fin de la liste les chiffres les plus grand. Pour ce faire, l'algorithme compare les nombres deux à deux et échange leurs places si le premier est plus grand que le second. Voici un exemple pour une liste de 5 nombres:

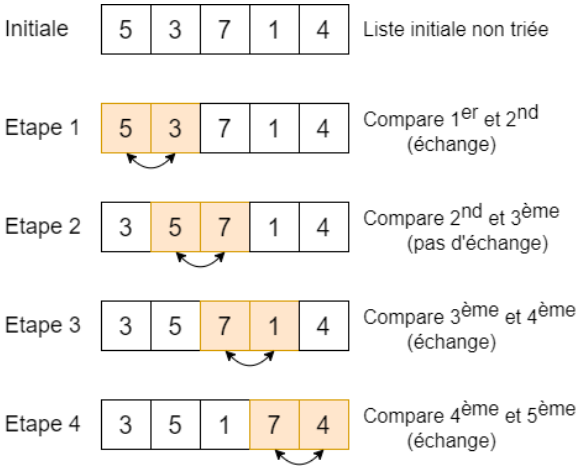


FIGURE 5 – Fonctionnement du tri à bulle

Une fois que l'on arrive à la fin de la liste, on recommence au début et cela jusqu'à ce qu'il n'y ai plus d'échange.

Maintenant que le tri est implémenté, nous l'avons fait fonctionné pour différentes tailles de liste et avons noté le temps d'exécution de ce tri. Une amélioration possible de ce tri est de ne pas revenir comparer le début de la liste lorsque nous somme arrivé au bout mais de faire demi tour. Ce tri spécifique, appelé shaker, permet d'améliorer les temps d'accès à la mémoire, notamment pour les listes de grande taille.

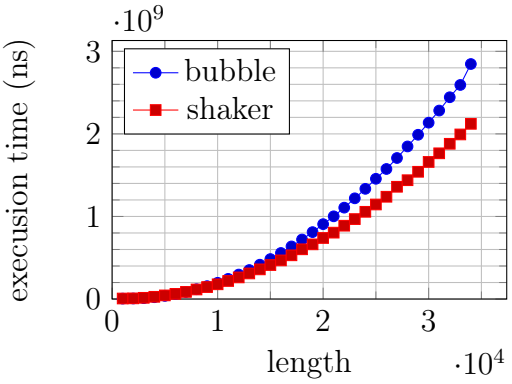


FIGURE 6 – Temps d'exécution des algorithmes bubble et shaker en fonction de la taille de la liste

On observe une courbe exponentiel ce qui correspond à la complexité de ce tri qui est en $\mathcal{O}(n^2)$.

3 Optimisation sur CPU

3.1 Optimisation d'algorithme

Une méthode pour diminuer le temps de tri de certains algorithmes est de modifier ceux-ci pour qu'ils se rapprochent de l'adage « diviser pour régner ». En effet, trier plusieurs petites listes avec un algorithme d'une complexité quadratique comme le tri à bulle en $\mathcal{O}(n^2)$ sera plus rapide que trier une grande liste d'une seule traite. Partitionner le tri permet ainsi de fortement diminuer le temps du tri, surtout pour les longues listes. On peut retrouver le résultat obtenu pour un tri à bulle sur la figure **??**, où le vecteur à trier à été divisé par 2 et par 10.

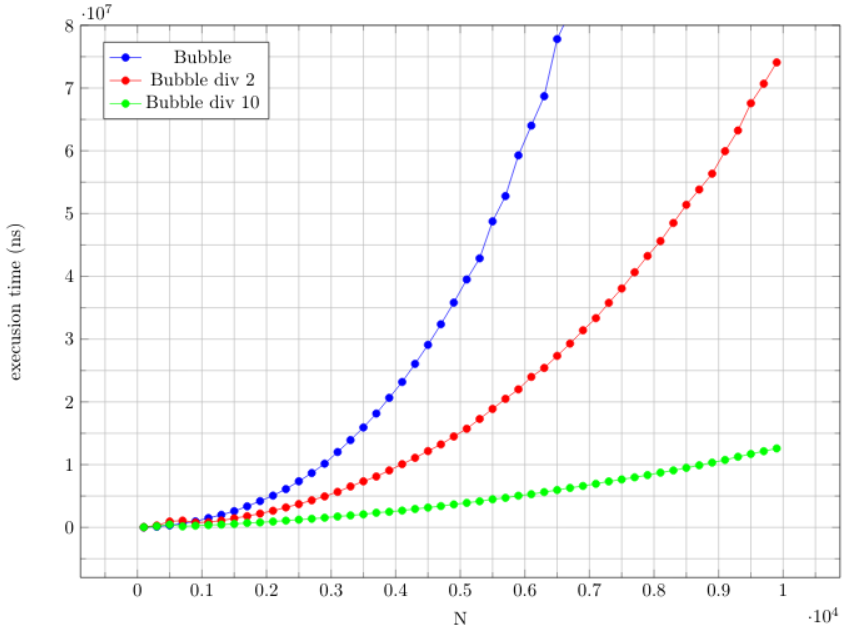


FIGURE 7 – Résultat du partitionnage du tri avec le tri à bulle

Pour ce faire, il suffira d'appliquer le tri choisi sur chaque sous-ensembles du tableau initial, puis de fusionner chacun des sous-ensembles triés dans un tableau final. Si on divise au maximum le tableau initial, c'est-à-dire jusqu'à n'avoir que des sous-ensemble d'un élément, on va alors faire un tri fusion et obtenir une complexité en $\mathcal{O}(n \log n)$.

En plus de diminuer la complexité des algorithmes modifiés, cette optimisation permet aussi, sans forcément sans rendre compte, d'améliorer l'utilisation de la mémoire cache. En effet, trier des ensembles de données plus petit revient à diminuer le nombre de données à sauvegarder dans la mémoire cache du CPU et donc diminue le nombre d'accès requis à la mémoire RAM. Idéalement, il faudrait que le tableau ne soit charger qu'une seule fois en mémoire cache pendant le tri et il faut donc diviser le tableau initial en fonction de la taille des mémoires caches du CPU sur lequel tourne l'algorithme.

3.2 *Multithreading*

Une des meilleurs optimisation que l'on peut faire est de paralléliser les calculs. En effet, toutes les opérations de comparaison sont indépendantes et peuvent donc être fait en même

temps. Un CPU possède plusieurs coeurs de calcul qui peuvent fonctionner simultanément. Pour utiliser ces différents coeurs de calcul, nous utilisons l'API OpenMP.

Pour paralléliser le calcul, nous séparons la liste à trier en plusieurs sous-listes. Chaque sous-liste est ensuite triées sur un coeur de calcul différent. Il s'agira ensuite de fusionner les sous-listes sur un coeur pour obtenir la liste initiale triée.

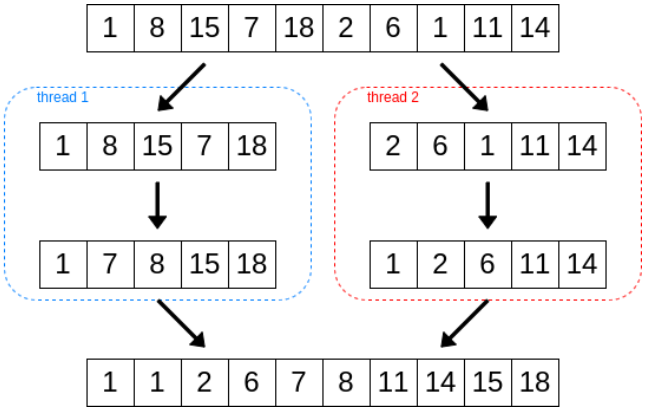
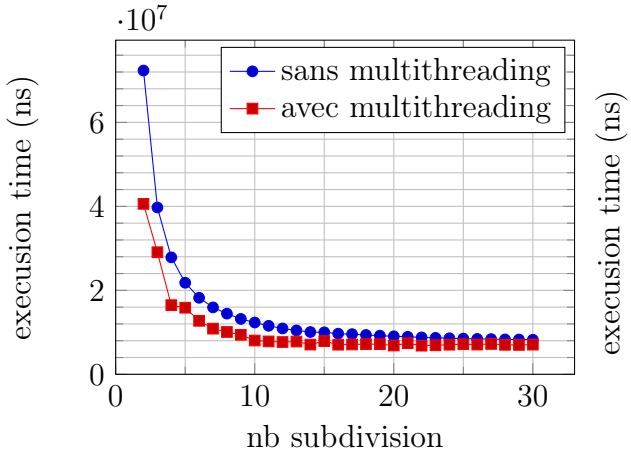
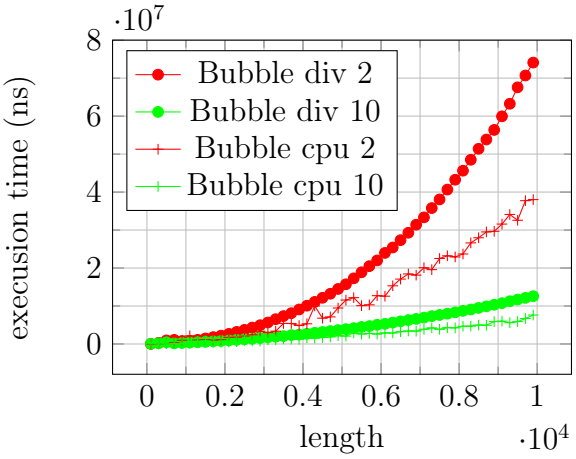


FIGURE 8 – Multithreading

Nous avons utilisé cette technique sur plusieurs algorithmes et notamment sur le tri à bulle. Étant le tri le moins efficace, il est intéressant de voir combien il est avantageux de paralléliser le problème sur un algorithme qui fait un nombre important d'opérations. Nous avons alors réalisé plusieurs mesures. La première est de faire varier le nombre de subdivision pour une taille de liste fixe (ici 10^3) Nous obtenons alors la figure 9.1. On remarque que le fait de diviser la liste en sous-liste est plus efficace que d'avoir une seul liste. Le multithreading permet ensuite d'accélérer encore plus le temps de calcul. La seconde mesure est de faire varier la longueur de la liste avec une subdivision fixe, on obtient la figure 9.2.



(a) Temps d'exécution en fonction du nombre de subdivision avec et sans multithreading



(b) Temps d'exécution en fonction de la taille

FIGURE 9 – Simulation

3.3 SIMD

Nous avons vu au cours de la partie précédente des manières d'améliorer les performances temporelles de nos algorithmes de tri par multithreading, augmentant par la même occasion les ressources nécessaires à leur exécution. L'objectif de cette nouvelle partie sera elle aussi d'augmenter les performances de nos algorithmes de tri mais en n'augmentant pas les ressources utilisées. On se servira pour cela d'instructions intrinsèques ou SIMD (Simple Instruction Multiple Data). SIMD permet de gagner en performances en parallélisant des instructions similaires réalisées sur plusieurs données.

3.3.1 Jeux d'instruction

Il a été choisi ici d'utiliser des jeux d'instructions compatibles avec le plus d'architectures possible, on utilisera donc AVX, AVX2 et BMI2 qui sont compatibles avec quasiment 100% des CPUs sortis après 2010. AVX2 permet de travailler sur des vecteurs de données de 256 bits qu'on peut remplir à partir de données contiguës ou non de la mémoire. On peut ensuite ramener ces vecteurs en mémoire à l'aide d'instructions de déchargement. AVX2 autorise aussi à l'aide de l'instruction permutevar8x32 de réaliser des permutations de mots de 32 bits à l'intérieur d'un vecteur. Un des désavantages de ce jeu d'instruction est qu'il est peu compatible avec BMI2. En effet, étant donné que les masques et les résultats de comparaison sont renvoyés dans des vecteurs de 256 bits en AVX2 et dans des int pour BMI2 il va nous falloir faire de nombreuses conversions pour faire passer les données d'un masque d'un jeu d'instruction à l'autre car certaines instruction BMI2 n'ont pas d'équivalent. C'est le cas des instruction pext et pdep qui nous seront très utilises qui permette d'extraire ou stocker de manières contiguës des données non contiguës en fonction d'un masque.

3.3.2 Algorithme

Le tri rapide est un des algorithmes les plus utilisé dans l'industrie pour ces performances sur des listes "moyennement longues", c'est l'algorithme qu'on se propose ici d'optimiser à l'aide d'instructions SIMD. L'algorithme de tri rapide en lui-même n'est pas modifié, seule la fonction de partitionnement à été adaptée, c'est à dire la fonction chargée de placer à gauche du pivot les valeurs inférieures à ce dernier. En effet, les membres du tableau à trier qui étaient précédemment comparés au pivot et déplacés séquentiellement et un par un vont maintenant subir les mêmes étapes mais 8 par 8. On présente l'algorithme de partitionnement SIMD dans l'Algortihm 1

Cet algorithme est similaire en beaucoup de points à l'algorithme de partitionnement scalaire. On créé des pointeurs qui démarrent à chaque extrémité de notre liste à trier et qui vont nous permettre de garder une trace d'où stocker les données (gauche/droite) et où charger les données (ind_gauche/droite), la boucle se termine lorsque les indices de chargement ont une différence inférieure à la taille de nos vecteurs SIMD. Démarre alors un algorithme de partitionnement séquentiel pour les quelques données restantes. Le partitionnement en lui même est réalisé par la fonction Partition_vecteur dont le pseudo-code est fourni en Algorithm 2. Le fait de conserver en mémoire les vecteurs contenant les valeurs des extrémités de chargement permet d'éviter d'écrire par dessus des valeurs qui ne sont pas encore triées. L'implémentation AVX2/BMI2 de cet algorithme est fournie en annexe.

Dans cet algorithme nous permutons les éléments de la liste à partionner pour faire en

Algorithm 1 Partitionnement

```
1:  procedure PARTITION(liste)
2:      [vect_pivot, ind_pivot] ← choix_pivot(liste)
3:      swap(ind_pivot, longueur_liste-1)
4:      ind_gauche ← 0
5:      ind_droite ← longueur_liste
6:      droite ← longueur_liste
7:      gauche ← 0
8:      vect_gauche ← load(liste, ind_gauche)
9:      vect_droite ← load(liste, ind_droite)
10:     while ind_gauche+8 < ind_droite do
11:         if (ind_gauche - gauche)<=(droite - ind_droite) then
12:             [gauche, droite] ← Partition_vecteur(vect_gauche, vect_pivot, gauche, droite)
13:             ind_gauche ← ind_gauche + 8
14:             vect_gauche ← load(liste, ind_gauche)
15:         else
16:             [gauche, droite] ← Partition_vecteur(vect_droite, vect_pivot, gauche, droite)
17:             ind_droite ← ind_droite − 8
18:             vect_droite ← load(liste, ind_droite)
19:     gauche ← Partition_sequentielle(val_gauche, val_droite, gauche, droite)
20:     swap(gauche,longueur_liste-1)
21:     return gauche
```

Algorithm 2

```
1:  procedure PARTITION_VECTEUR(vect, vect_pivot, gauche, droite)
2:      mask_sup ← cmp_SIMD(vect, pivot_vect)
3:      if mask_sup == 0x00 then
4:          store(val, gauche)
5:          gauche ← gauche + 8
6:      else if mask_sup == 0xFF then
7:          store(val, droite)
8:          droite ← droite − 8
9:      else
10:         shuffle_mask ← get_shuffle_mask(mask_sup)
11:         val ← permute(val, shuffle_mask)
12:         mask_sup ← cmp_SIMD(vect, pivot_vect)
13:         store_mask(val, droite, mask_sup)
14:         store_mask(val, gauche, not(mask_sup))
15:     return [gauche, droite]
```

sorte que les éléments inférieurs au pivot soient placés au début du vecteur et ceux supérieurs au pivot soient placés en queue du vecteur. Cette étape est nécéssaire car contrairement à AVX512, AVX2 ne possède pas d'instruction pour réaliser un stockage contiguës de bits non contiguës. L'implémentation AVX2/BMI2 de cet algorithme est fourni en annexe avec quelques annotations.

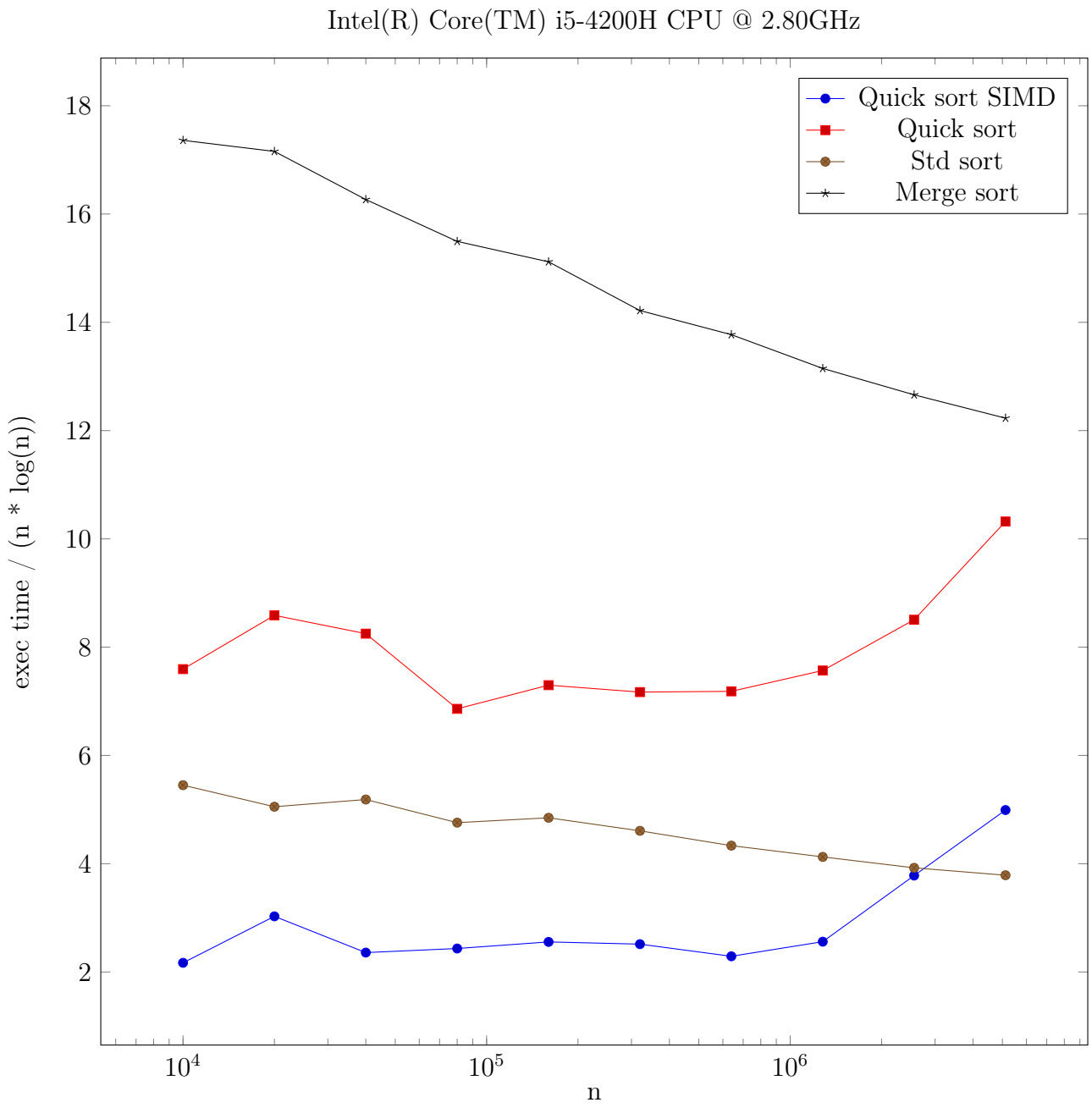
Afin d'accélérer encore plus notre algorithme de tri rapide SIMD on se propose de le mixer avec un algorithme de tri bitonique SIMD pour le cas des petites listes, étant donné que le tri bitonique est beaucoup plus adapté pour des listes de très petite taille (<256 éléments). On modifie ainsi notre algorithme de tri rapide afin traiter le cas des listes de taille inférieure à 256 éléments.

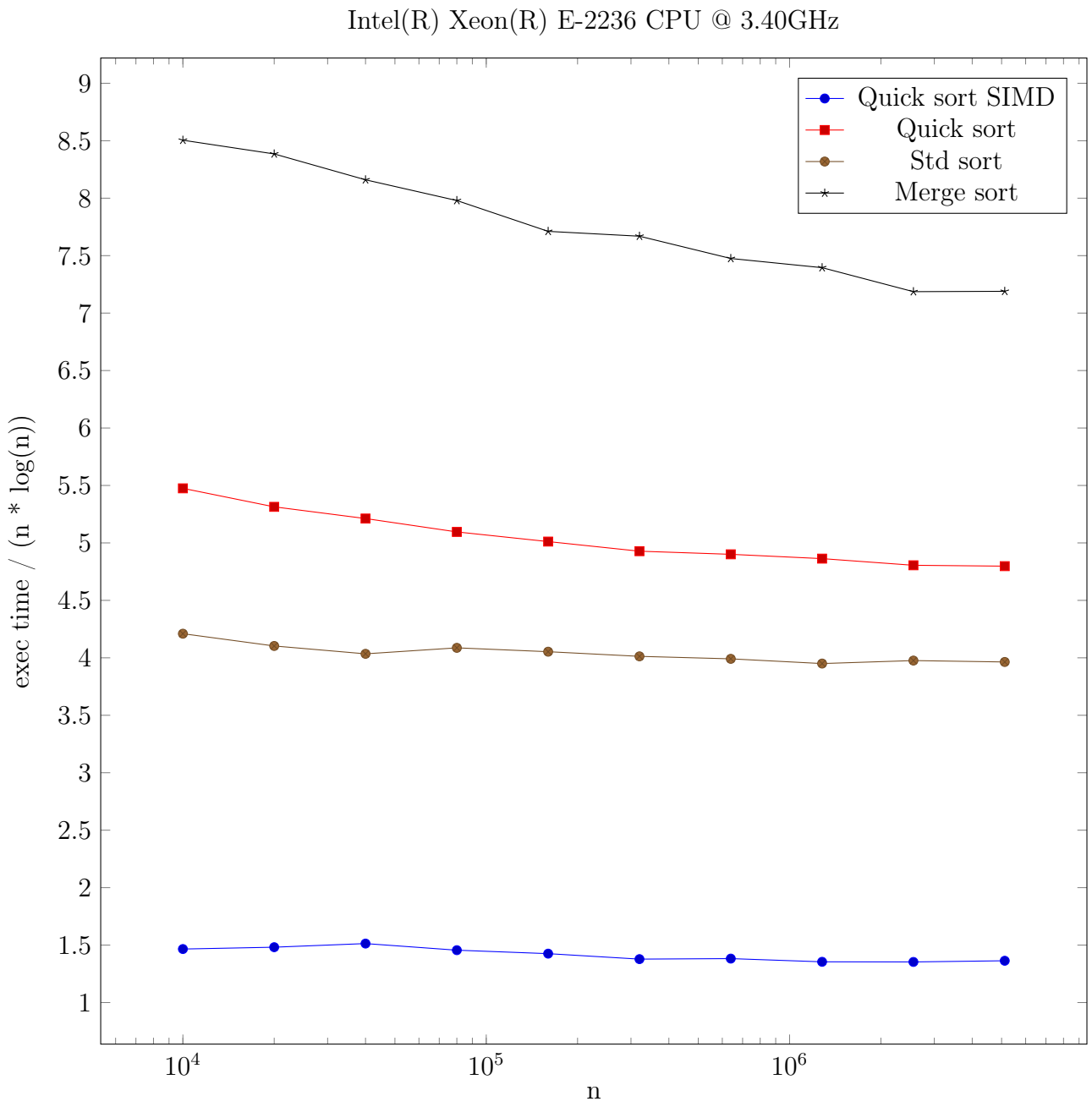
Algorithm 3

```
1: procedure TRI_RAPIDE_SIMD(liste, longueur_liste)
2:     if longueur_liste < 256 then
3:         Tri_bitonique(liste, longueur_liste)
4:     else
5:         ind_pivot ← Partitionnement_SIMD(liste, longueur_liste)
6:         Tri_rapide_SIMD(liste, ind_pivot)
7:         Tri_rapide_SIMD(liste[pivot], longueur_liste - ind_pivot)
8:     return liste
```

3.3.3 Résultats

Nous avons décidé de comparer notre algorithme avec trois algorithmes qui nous semblaient pertinent : nos implémentations de tri-fusion et tri-rapide ainsi que std::sort. Le dernier est un algorithme de tri natif du C++ qui utilise une sorte de tri rapide fusionné avec un heap sort. Pour comparer les performances de ces algorithmes, il a été choisi deux architectures CPU différentes, une plutôt récente (2019) Intel(R) Xeon(R) E-2236 CPU @ 3.40GHz et une très ancienne (2013) Intel(R) Core(TM) i5-4200H CPU. On testera la capacité de chaque algorithme de trier des listes d'entiers générés aléatoirement. On représente en figure X et X les temps moyen d'éxécution de chaque algorithme divisé par n*log(n) avec n la longueur de la liste triée.


Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz
Quick sort SIMD
Quick sort
Std sort
Merge sort
exec time / (n * log(n))
n
18
16
14
12
10
8
6
4
2
10^4
10^5
10^6



Sur l'architecture la plus ancienne, on remarque que notre implémentation du tri rapide en SIMD est plus performante que std::sort pour des listes de petite taille. Sur l'architecture plus récente le tri rapide SIMD est plus performant quelle que soit la taille du tableau, on trouve en moyenne un facteur 3 d'efficacité entre notre implémentation du tri rapide en SIMD et std::sort. Ainsi notre implémentation du tri rapide en SIMD est concluante, et elle permet de plus de nous rendre compte des progrès des architectures CPU vers l'utilisation d'instructions SIMD. Une manière d'améliorer les performances de notre tri serait de le passer en AVX512 qui permet notamment d'utiliser des vecteurs de taille deux fois plus élevé et qui possède une instruction très intéressante de "compress-store" qui permettrait d'éviter notre étape de permutation des valeurs à l'intérieur d'un vecteur pendant le partitionnement.

4 Module de tri sur FPGA

Une partie de ce projet a été dédié à la conception d'architectures de tri sur FPGA. L'objectif était de voir les enjeux d'un trieur sur FPGA, les possibilités que cela offrait mais aussi et surtout de nous essayer à la conception d'architecture avec la synthèse de haut niveau (*High Level Synthesis*, HLS) de *Vivado HLS*.

4.1 *High Level Synthesis*

Afin de gagner du temps dans la conception et pour pouvoir générer plusieurs architectures dans un faible temps imparti, il a été décidé de concevoir nos architectures tris avec la synthèse de haut niveau. Étant donné que des trieurs ont déjà été développé en C++ pour tourner sur CPU, il suffira alors de proposer une architecture de tri cohérente sur FPGA à décrire en C++ puis d'apdater un algorithme de tri déjà programmé et d'y rajouter quelques `pragma HLS` pour tordre le bras au compilateur afin d'obtenir une solution optimisée.

4.2 Protocole de test

Le protocole choisi pour vérifier les architectures générés est simple : il ne fera que vérifier le fonctionnement du tri implémenté. En effet, l'IDE *Vivado HLS* fournit le temps de latence à chaque niveau de l'architecture générée. Cette première mesure sera amplement suffisante dans un premier temps, mais une autre méthode serait de compter le nombre de cycle d'horloge qu'il s'est écoulé entre le début et la fin du tri. Étant donné l'avancement que nous avons eu sur cette partie, ce type de compteur n'a pas été intégré.

Pour vérifier le tri, une architecture de test a été fournie afin de concentrer le travail sur le développement du trieur. Cette architecture est un *wrapper* qui permet de récupérer un tableau de données non triées depuis le port UART du FPGA puis de renvoyer ce tableau de donnée après le tri. Si tout ce passe comme souhaité, ce tableau doit être trié. Avec cette architecture vient un programme écris en C++ qui permet de générer un tableau de données non triées, de le transférer au FPGA par l'UART, de récupérer le tableau « trié » et enfin de vérifier le fonctionnement du tri.

4.3 Architecture envisagée

Stratégie de tri

Comme nous sommes sur FPGA, c'est à nous de décider les ressources que nous souhaitons investir dans le trieur. Il s'agit de choisir une architecture qui prend en compte les ressources mis à disposition. En se basant sur le principe de *diviser pour régner*, on propose une architecture basée sur plusieurs coeurs de tri pour diminuer au maximum le temps nécessaire pour trier le tableau. Ainsi, au plus simple, l'architecture envisagée est celle présentée à la figure 10, avec deux coeurs de tri. La taille des tableaux à trier est définie lors de la synthèse du module globale sous *Vivado HLS*.

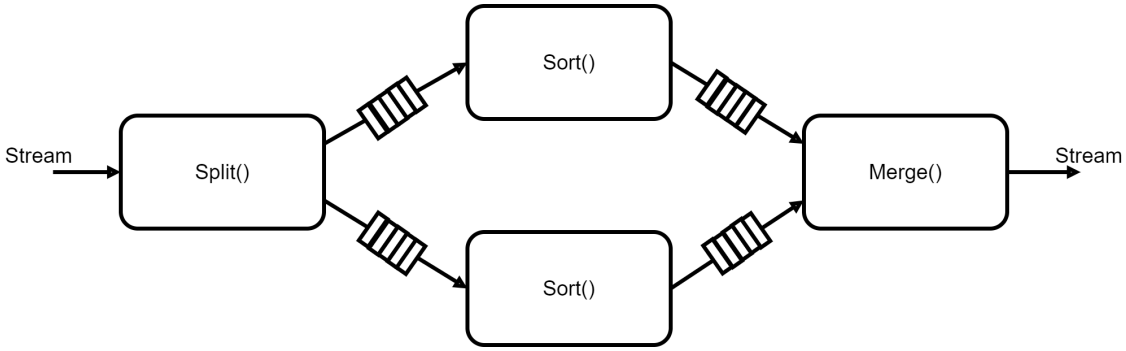


FIGURE 10 – Architecture de tri envisagée sur FPGA

Cette architecture est divisée en 3 parties :

1. Un module `Split()` pour diviser le tableau en entrée en sous-tableaux de plus petites tailles.
2. Des modules `Sort()` pour trier un sous-tableau.
3. Un module `Merge()` pour fusionner les sous-tableaux triés en un unique tableau trié.

Ces parties sont reliées entre elles par des FIFOs afin de fluidifier le fonctionnement du tri. Ces FIFOs sont notamment très utiles entre le module `Split()` et les modules `Sort()` si le tri implémenté permet de commencer à trier les sous-tableaux avant d'avoir reçu leurs derniers éléments.

Cette architecture est donc notre base pour trier sur FPGA. Il suffira alors de définir un tri dans le module `Sort()` puis de dupliquer le module. Plus il y a de modules de tri instanciés, plus le temps de tri sera faible, mais plus la consommation de ressource sera importante. Augmenter le nombre de sous-tableaux à trier va également augmenter la complexité, *i.e.* le coût matériel, et potentiellement la latence du module `Merge()` vu que ce dernier devra manipuler plus de flux de données en même temps.

Module de séparation de flux

Le module `Split()` est l'entrée de l'architecture. Il répartie le flux de données d'entrée, qui correspond aux données à trier, dans plusieurs tableaux équitablement. Cette partie de l'architecture se doit d'être pipeline pour délivrer les données le plus rapidement possible au second étage. La latence visée est donc proportionnelle la longueur du tableau à trier et il devrait même ni avoir qu'un cycle d'horloge par données traitées.

Module de tri

Il s'agit de cette partie qui effectue vraiment le tri. C'est vraisemblablement la latence de ce module qui va impacter la latence du trieur au global, le tri à choisir découle donc de la latence souhaitée, sans oublié les ressources à consommer. Plusieurs stratégies sont alors disponible, un tri peu coûteux en ressources mais lent comme un tri à bulle, qui peut être un bon choix dès lors qu'on instancie plusieurs module `Sort()`, ou bien un tri rapide à latence fixe mais très coûteux dès lors qu'on souhaite trier des longues listes comme un tri bitonique.

Il n'y a pas de solutions miracles surtout vu qu'on n'implémente pas un tri pour une application précise, et il a été décidé d'implémenter un tri simple, peu coûteux et un peu plus performant qu'un tri à bulle : une variante du tri par insertion. Un des principaux avantages de ce tri dans notre situation est qu'il peut commencer à trier sans avoir reçu l'entièreté du tableau à trier. Bien qu'en moyenne pour des données inconnus, la complexité du tri est en $\mathcal{O}(n^2)$, lorsque les données en entrées peuvent être plus ou moins prédites, la complexité du tri peut s'approcher de $\mathcal{O}(n)$ soit mieux qu'un classique tri fusion sur CPU.

Il y a également possibilité d'implémenter différents types de tri dans chaque module de tri instancié. Cela permettrait d'accélérer le tri dans le cas où les données à trier seraient plus ou moins connu. Il faudra juste ajouter une étape dans le module `split()` pour savoir où chaque donnée doit être envoyée.

Module de fusion

Le module `Merge()` est la sortie de l'architecture. Il effectue la fusion des tableaux triés et envoie les éléments triés sous forme de flux au module qui serait mis en aval. Tout comme le module d'entrée `Split()`, la latence de ce bloc doit être proportionnelle au nombre de données à trier, et au mieux on doit sortir une donnée par cycle d'horloge.

Comme explicité plus haut, la complexité de ce module dépend directement du nombre de flux à gérer en entrée. Il se peut que si le nombre de flux est trop élevé, le module ne puisse pas sortir une nouvelle donnée à chaque cycle d'horloge. Une solution serait de modifier légèrement l'architecture initiale pour mettre en cascade plusieurs module de fusion.

4.4 Résultats et performances

Les performances qui seront données ont été obtenu en synthétisant des architectures qui tries 128 entiers non signés sur 8 bits, avec une période d'horloge visée de 10ns et sur une cible Nexys A7 50T. En réalité, de nombreuses autres architectures ont été synthétisées avec d'autres paramètres ce qui nous a permis de voir l'un des énormes avantages de la synthèse de haut niveau car cela aurait pris un temps considérable à faire avec les langages de descriptions classiques.

Architecture synthétisée

Concernant l'architecture globale en elle-même, l'objectif n'a pas été atteint complètement. En effet, l'architecture obtenue se rapproche de celle de la figure 11, c'est-à-dire sans les FIFOs entre l'étage 1 et 2. Cela signifie que les modules `Sort()` ne pourront commencer à trier que quand les trieurs auront reçu le tableau en entier. On perd donc un des avantages du tri par insertion implémenté.

Concernant les différents modules, l'objectif a été atteint pour le module `Split()` , le module `Merge()` et le module `Sort()`, si on omet le problème de FIFO en entrée de celui-ci.

On notera quand même qu'obtenir le résultat du module de fusion a été compliqué contrairement à ce que nous nous attendions. Notamment, si ce module avait été écris à la main en VHDL, le résultat souhaité aurait été obtenu assez rapidement et sans trop d'effort, ce qui ne s'est pas passé avec la synthèse de haut niveau.

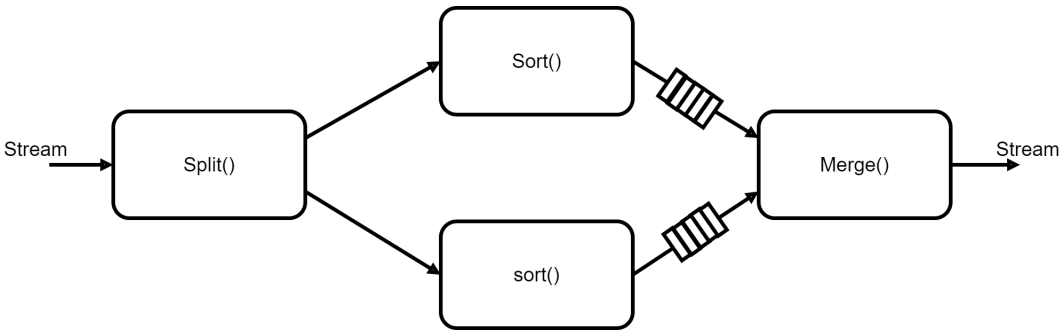


FIGURE 11 – Architecture de tri synthétisée sur FPGA

Coûts matériels

L'un des avantages matériels de l'architecture synthétisée est qu'elle consomme plutôt peu de ressources au global mais surtout que les ressources consommées pour chaque module sont peu dépendante de la taille du tableau à trier. En effet, dans le cas où le nombre de données à trier augmente, c'est surtout le nombre de bloc RAM consommé, ou tout autre élément de mémorisation, qui va augmenter. C'est un problème sur FPGA étant donné que les blocs RAM sont plutôt cher, mais c'est un problème indépendant de notre solution. En effet, si l'objectif de l'architecture est de trier une grande liste, il faudra nécessairement utilisé des blocs RAM dans l'architecture et le nombre de blocs consommés sera directement liés à la taille de la liste à trier.

Il est également intéressant de voir que le tri d'entiers codés sur des longueurs différentes change linéairement le nombre de ressources consommées comme on peut le voir sur la figure 12. Ce résultat n'était pas vraiment attendu, mais surtout, sans la synthèse de haut niveau, nous n'aurions pas eu le temps de constater un tel résultat.

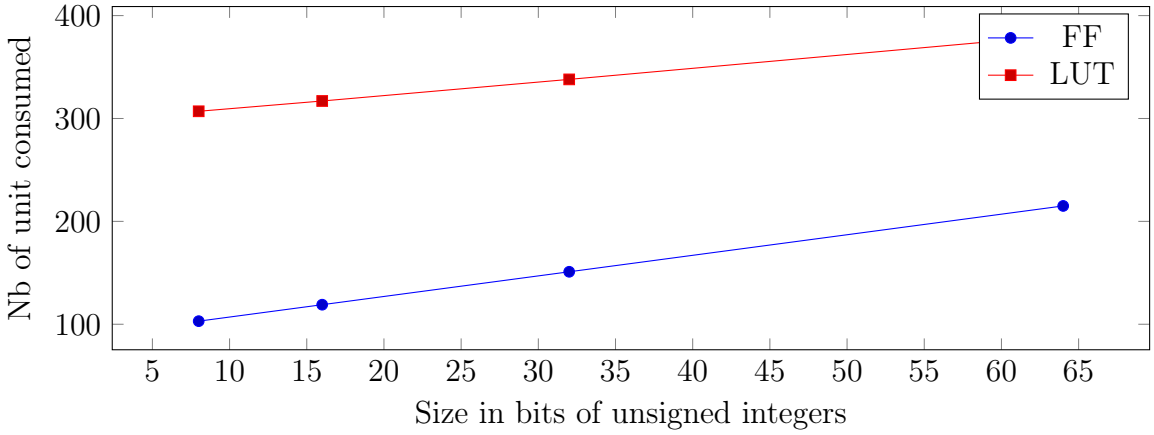


FIGURE 12 – Ressources consommées par un module `Sort()` en fonction de la taille des entiers non signés en bit

Performances temporelles

On retrouve la latence de chacun des modules dans le tableau 1.

Module	`Split()`	`Sort()`	`Merge()`
Latence (cycle d'horloge)	130	192 - 2208	133

TABLE 1 – Latence de chaque module de l'architecture

On voit bien que l'objectif temporel des modules d'entrée et de sorties ont été atteint, mais en réalité, ces latences n'ont que peu d'impact comparées au temps qu'il faut pour trier les sous-tableaux. La complexité et lenteur du tri par insertion se fait rapidement ressentir.

Axes d'amélioration

Il y a de nombreuses pistes d'amélioration pour cette architecture. Déjà, comme évoqué plusieurs fois et si le temps l'avait permis, implémenter différents tris pour le module `Sort()`, tel que le tri bitonique, aurait été intéressant et cela aurait permis de comparer nos solutions. Il y a aussi l'idée de diviser la liste d'entiers à trier en plusieurs sous liste, mais sans instancier plus de module `Sort()`. Cela viendrait avec un module de gestion mémoire qui organiserait la fusion des listes pour reconstruire la liste finale triée. La figure 13 montre le schéma bloc de l'architecture qui découlerai de cette idée.

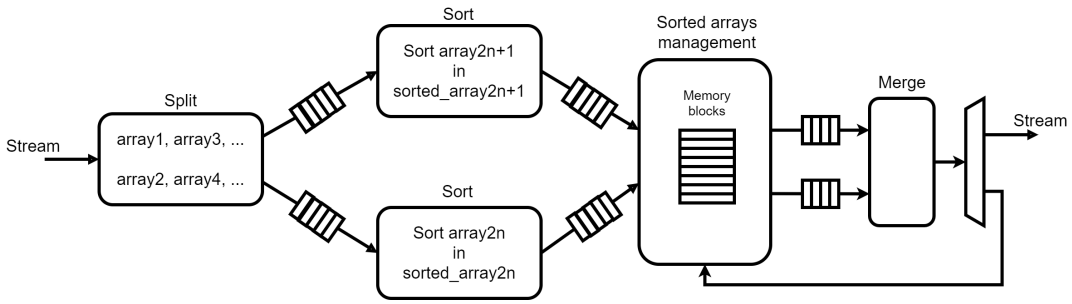


FIGURE 13 – Architecture de tri sur FPGA avec un module de gestion mémoire

5 Conclusion

A travers ce projet, nous avons pu découvrir et développer des solutions pour optimiser des algorithmes. La liberté qui nous était offerte pour le choix de la technologie et des optimisations à appliquer nous ont permis de toucher à plusieurs points. Cela a permis d'approfondir certains connaissances que nous avions vu au cours de ce dernier semestre, comme la parallélisation de processus avec `OpenMP`.

Concernant la synthèse de haut niveau avec *Vivado HLS*, on a pu découvrir que, certes il est très rapide et étonnamment efficace de générer des architectures avec la synthèse de haut niveau, mais dès qu'il s'agit de générer l'architecture souhaitée et optimisée comme prévue, c'est une toute autre histoire. Bien que l'avancement a été ralenti par notre inexpérience en développement HLS, on en ressort avec de nouvelles compétences. On reste tout de même déçu de na pas avoir commencer à travailler plus tôt sur cette partie, ce qui aurait permis d'approfondir encore plus la HLS.

6 Annexes

```
int part_SIMD_int(int* tab,int len){


    int ptr_left=0;
    int ptr_left_next=0;

    int ptr_right=len-1;
    int ptr_right_next=len-1;

    int ind_pivot=rand() % ( len );
    const __m256i pivot=_mm256_set1_epi32(tab[ind_pivot]);


    int temp=tab[ind_pivot];
    tab[ind_pivot]=tab[len-1];
    tab[len-1]=temp;
    ind_pivot=len-1;

    __m256i val_left=_mm256_loadu_si256((__m256i*)&tab[ptr_left]);
    __m256i val_right=_mm256_loadu_si256((__m256i*)&tab[ptr_right-8]);
    __m256i val;


    while (ptr_left+8<ptr_right){

        if ((ptr_left-ptr_left_next)<=(ptr_right_next-ptr_right)){
            val=val_left;
            ptr_left+=8;
            val_left=_mm256_loadu_si256((__m256i*)&tab[ptr_left]);
        }
        else {
            val=val_right;
            ptr_right-=8;
            val_right=_mm256_loadu_si256((__m256i*)&tab[ptr_right-8]);
        }
        part8_SIMD_int(tab, val, &ptr_left_next, &ptr_right_next, pivot);
    }


    _mm256_maskstore_epi32(&tab[ptr_left_next],
        _mm256_loadu_si256((__m256i*)&movemask[(ptr_right-ptr_left)*8]),
                            val_left);

    ptr_left_next+=seq_sort(&tab[ptr_left_next],ptr_right-ptr_left,
                             tab[ind_pivot]);


    temp=tab[ind_pivot];
    tab[ind_pivot]=tab[ptr_left_next];
    tab[ptr_left_next]=temp;



    return ptr_left_next;
}
```

```
inline void part8_SIMD_int(int* tab,__m256i val, int* ptr_left_next,
                           int* ptr_right_next, __m256i pivot){
    const __m256i mask_max= _mm256_set1_epi32(-1);
    __m256i mask_high =_mm256_cmpgt_epi32( val,  pivot);

    //mask_high==0 0 0 0 0 0 0 0
    if (_mm256_testz_ps((__m256)mask_high,(__m256)mask_max)) {
        _mm256_maskstore_epi32(&tab[*ptr_left_next], mask_max, val);
        *ptr_left_next+=8;

    }

    //mask_high== -1 -1 -1 -1 -1 -1 -1 -1
    else if (_mm256_testc_ps  ((__m256)mask_high,(__m256)mask_max)) {
        _mm256_maskstore_epi32(&tab[*ptr_right_next-8], mask_max, val);
        *ptr_right_next-=8;
    }


    else{

        int mask_high_32 = _mm256_movemask_epi8(mask_high);
        unsigned int count_high=__builtin_popcount((long int)mask_high_32)/4;
        __m256i mask_low=_mm256_andnot_si256(mask_high, mask_max);
        int mask_low_32=_mm256_movemask_epi8(mask_low);
        unsigned int count_low=8-count_high;

        int mask_32_low=_pext_u32(1985229328,mask_low_32);
        int mask_32_high= _pext_u32(1985229328,mask_high_32);

        mask_32_low += mask_32_high<<(count_low<<2);


        int a = _pdep_u32(mask_32_low,252645135);
        int b = _pext_u32(mask_32_low, 4294901760);
        b=_pdep_u32(b,252645135);
        __m256i temp_256=_mm256_set_epi32(0,0,0,0,0,0,b,a);
        __m128i temp_128=(__m128i)_mm256_castps256_ps128((__m256)temp_256);
        __m256i shufmask = _mm256_cvtepu8_epi32(temp_128);


        val=_mm256_permutevar8x32_epi32 (val, shufmask);
        _mm256_storeu_si256  ((__m256i*)&tab[*ptr_right_next-8], val);
        _mm256_storeu_si256  ((__m256i*)&tab[*ptr_left_next], val);

        *ptr_left_next+=8-count_high;
        *ptr_right_next-=count_high;
    }


}
```