Rapport de stage

Accélération matérielle de la recherche de la clique de poids maximum.

été 2021.

LEOTHAUD DYLAN

1 INTRODUCTION

Il y a de plus en plus d'électronique, dans les téléphones ou la domotique par exemple. Cependant dans certain cas, comme pour des calculs spécifiques ou des systèmes avec des contraintes supplémentaire, d'efficacité ou de taille, et pour cela une architecture spécialisée peut être la meilleure solution.

La synthèse de haut-level (High Level Synthesis, HLS) est un outil prenant un programme en entrée, dans notre cas, écrit en C/C++, et renvoyant une description de circuit matériel.

Les circuits générés par la HLS et exécuté sur FPGA sont généralement utilisé pour des calculs intense et répétitif comme pour appliquer un filtre sur une image par exemple. Pour de telles applications, la HLS est capable d'exploiter du parallélisme au niveau des instruction et ainsi d'augmenter l'efficacité des calculs ainsi que l'efficacité énergétique du circuit.

Cependant, pour des applications moins régulières, c'est-à-dire avec un flot de contrôle plus complexe ou des dépendances à la mémoire qui sont dynamiques, la HLS atteint ses limites et les performances du matériel généré sont réduite.

Hors, lorsque plusieurs programmes voire plusieurs partie du même programme ne s'exécutent pas en même temps, il est possible de les fusionner pour obtenir une architecture plus petite effectuant les mêmes calculs avec une efficacité similaire.

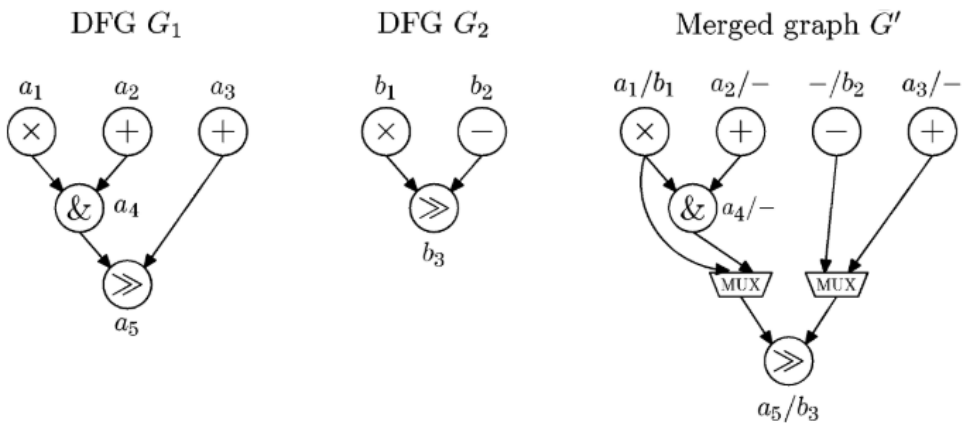

Fig. 1. Exemple de fusion de chemin de données donné dans [1], le dernier graphe est la fusion des deux premiers.

La figure 1 donne un exemple de la fusion de deux chemins de données, on voit que le chemin fusionné est capable d'effectuer les calculs des deux chemins de données grâce aux multiplexeurs qui sont moins coûteux qu'un multiplieur par exemple.

Pour la fusion de deux chemins de données, l'algorithme de Moreano donne une solution optimale. [1] mais s'effectue en un temps exponentiel à cause de la recherche d'un clique de poids maximum dans un graphe non-orienté pondéré, ce qui est un problème NP-complet.

Ainsi, bien que la construction du graphe pondéré ou la reconstruction du chemin de donnée fusionné peut prendre du temps, le plus long de l'algorithme est de trouver la clique de poids maximum sachant qu'un graphe à n sommets peut avoir jusqu'à $3^{n/3}$ cliques maximales. [2]

Ainsi, lors de l'utilisation de la HLS, les fusions de chemins de données peuvent permettre d'obtenir des circuits de surfaces plus petites quand il y a beaucoup de zones de calcul mutuellement exclusives.

L'objectif est alors d'accélérer la recherche de clique de poids maximal afin de pouvoir fusionner plus de chemin de données ou des chemins de données plus grands malgré l'explosion du temps de calcul.

Nous allons implémenter un algorithme de branch and bound, bien que cela ne semble pas adapté à un circuit matériel, puis l'adapter et le rendre plus efficace avec la HLS.

Cela se fera grâce à un circuit matériel implémenté sur un FPGA (Field Programmable Gate Arrays) et généré à partir d'un code C++ transformé par la synthèse de haut niveau (HLS).

2 ÉTAT DE L'ART

Dans cette partie nous allons faire un tour d'horizon des notions utiles et des publications importantes.

2.1 Datapath merging

Un chemin de données est un ensemble de composants et d'arcs représentant les dépendances de données entre ces composants. La figure 2 donne un exemple de chemin de données.

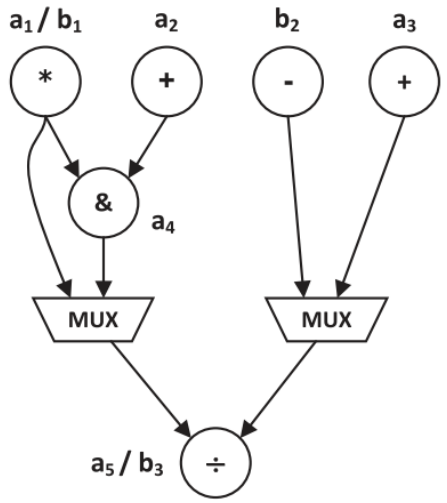

Fig. 2. Exemple de chemin de données provenant de [3]

Le problème de fusion de chemin de données (datapath merging) prend plusieurs chemins de données en entrée et cherche un chemin de données pouvant effectuer les mêmes calcul que chacune des entrées tout en étant le plus petit ou le moins coûteux possible.

Dans [1], Moreano et al. proposent un algorithme de fusion de chemins de données. Cette fusion est optimale dans le cas de la fusion de deux chemins de données car effectue la fusion de l'ensemble de sommets de poids le plus grand possible.

Pour cela, il recherche parmi toutes les fusions de composants et d'arcs possible celle qui a le gain le plus important.

Cela se fait en plusieurs étapes, rechercher toutes les fusions de composants et d'arcs possibles, fabriquer un graphe de compatibilité avec ces dernières, deux fusions ne sont pas compatibles si elles fusionnent un même composant d'un chemin de données à deux composants différent du second. Pour finir, en recherchant la clique de poids maximum, on obtient le meilleur ensemble de fusion compatibles entre elles possible, il reste alors à reconstruire un chemin de données en effectuant ces fusions.

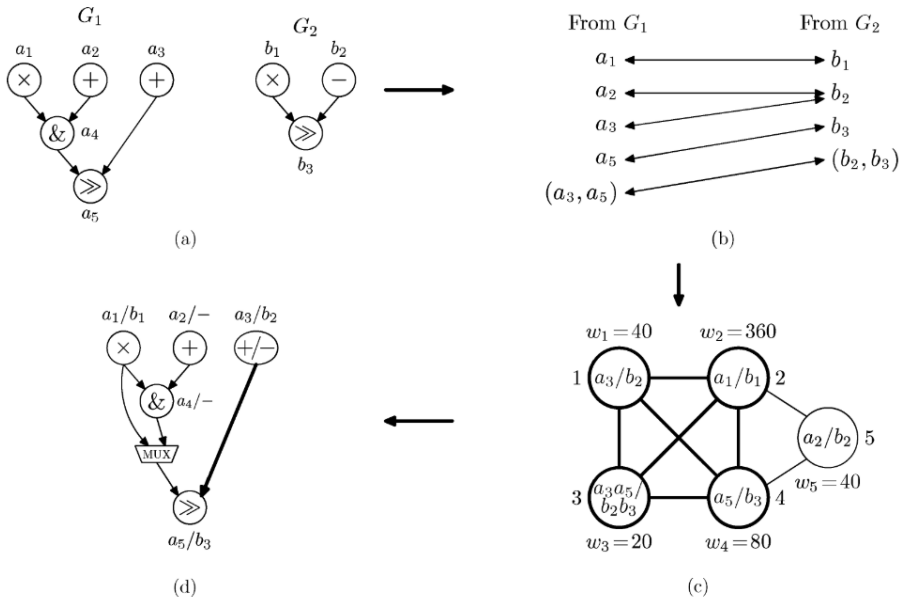

Fig. 3. Exemple de fusion de chemin de données donné dans [1], en détaillant les différentes étapes de l'algorithme.

L'algorithme peut alors se décomposer en 4 étapes majeures. En entrée, on a deux chemins de données (3.a), on recherche toutes les fusions de sommets et d'arcs possibles (3.b), on en crée un graphe de compatibilité des fusions (3.c). Puis on cherche la clique de poids maximales de ce graphe afin de recréer un chemin de données (3.d) en effectuant les fusions représenter par les sommets de la clique.

Le graphe de compatibilité est un graphe pondéré où les poids des sommets représentent le gain en coût ou en surface d'une fusion. Si la fusion représente une fusion d'arcs, alors le poids du sommet est le coût du multiplexeur qui aurait été utilisé sans cette fusion. Si la fusion représente une fusion de composants, alors le poids du sommet est le coût du composant car on en utilise qu'un seul au lieu de deux, auquel on retire le coût des multiplexeurs nécessaire pour choisir les entrées.

2.2 Problème de la clique

Dans un graphe pondéré non-orienté $G = (V, E, w)$ où V est un ensemble d'élément appelé sommets, E est un ensemble de pair d'éléments de V appelé arcs et $w : V \mapsto \mathbb{N}$ est la fonction pondérant les sommets, une clique C est un sous ensemble de V dont tous les éléments sont reliés par des arcs.

La recherche de la clique de poids maximum dans un graphe est un problème NP-complet.

2.2.1 Algorithme de Bron-Kerbosch.

Un des algorithmes d'énumération des cliques maximales les plus connus est celui de Bron-Kerbosch (1973), décrit dans [4], il se base sur un principe simple qui est que si on fixe un sommet du graphe, une clique maximale soit le contient, soit ne le contient pas et donc essaies les deux possibilités.

Cet algorithme a une complexité en $O(3^{n/3})$ où n est le nombre de sommets du graphe. Cette complexité est optimale dans le pire cas car un graphe à n sommets contient au plus $3^{n/3}$ cliques maimales. [2]

Bien que la complexité dans le pire cas soit optimale, cet algorithme peut être amélioré dans les autres as, par exemple, une première version ajoute un choix de pivot en partant du principe que si une clique maximale ne contient pas un certain sommet, alors elle en contient un qui n'est pas un de ses voisins, sinon on pourrait l'étendre en lui ajoutant ce sommet, ce qui contredirais le caractère maximal de cette clique.

L'algorithme de Tomita propose une autre amélioration qui consiste à faire le choix de pivot qui minimise le nombre d'appel récursif.

2.2.2 Algorithme d'Östergård.

L'outil de recherche de cliques maximales le plus rapide est le cliquer [5], il est basé sur un algorithme de Patric Östergård. [6]

Cet algorithme est similaire au précédent mais rajoute le fait que lors de la recherche de clique de poids maximum dans un graphe G, si on connaît le poids p de la clique de poids maximum du graphe G auquel on a retiré un sommet u, alors le poids de la plus grande clique de G est plus grand que p et plus petit que $p + w(u)$ où $w(u)$ est le poids du sommet u.

Il recherche alors la clique de poids maximum en travaillant sur des sous-graphes de G auquel il ajoute au fur et à mesure un sommet à chaque étape et considère uniquement les cliques contenant ce sommet car celles ne le contenant pas auront déjà été traitées.

2.3 La synthèse de haut niveau

La synthèse de haut niveau (HLS) est un outil créant une description d'un circuit matériel ayant le même comportement qu'un programme écrit en C ou C++.

Cela se fait en plusieurs étapes, trouver les différents modules du programme, expliciter ces modules sous formes de portes logiques, puis ces portes logiques par des bloc rams (BRAMs), flips-flop (FFs) et lookups table (LUTs) avant d'être placé sur le FPGA. Et cela en passant par plusieurs phases d'optimisation.

On connait alors le comportement du circuit précis au cycle d'horloge près. Le nombre de cycle d'horloge nécessaire à l'exécution d'une fonction ou d'une itération d'une boucle est appelé la latence.

On dit qu'une boucle est pipeliné si l'itération suivante commence avant la fin de la précédente, le nombre de cycle d'horloge nécessaire avant le début de la prochaine itération est appelé intervalle d'initialisation (II).

Certaines contraintes limitent le II. D'une part, les ressources, si il n'y a pas assez de ressources matérielles pour faire tous les calculs en simultanés, il faudra nécessairement plusieurs cycles pour effectuer le calcul. D'autre part la distance de dépendances aux données, en effet, si pour effectuer

un calcul il faut un certains plusieurs étapes et connaître le résultat du calcul précédent, il sera alors nécessaire d'attendre la fin de ce dernier pour en commencer un nouveau.

La spéculation [10], est un mécanisme permettant d'augmenter artificiellement cette distance de dépendance. Si le programme est la forme d'un branchement avec un cas rapide et un cas lent, elle consiste à toujours supposer prendre le cas rapide et à annuler si on s'est trompé. Ainsi, le temps d'exécution n'est pas affecté s'il fallait prendre le cas lent mais est plus rapide s'il fallait prendre le cas rapide car le calcul a été commencé à plus tôt.

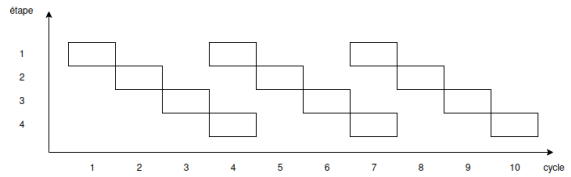

Fig. 4. Exemple de trace d'une boucle de latence 4 et de II=3 sur 3 itérations.

3 ACCÉLÉRATION DE LA RECHERCHE DE CLIQUE DE POIDS MAXIMUM.

3.1 Hypothèses sur l'algorithme

J'ai choisi d'implémenter l'algorithme de Tomita qui est présenté comme le meilleur algorithme, celui de référence selon [7], [8] par exemple, même si on ne connaît pas selon quels critères il serait meilleur que les autres.

Cet algorithme n'est initialement pas adapté à la HLS et nous allons alors devoir le modifier ce qui simplifiera le circuit hardware et l'efficacité d'une itération même si cela va augmenter le nombre d'itération, correspondant au nombre d'appels récursif, de l'algorithme.

input : C : clique en train d'être construite,
P : ensemble des sommets possible à ajouter,
X : ensemble des sommets exclus qui ne pourront pas être ajouté.
if $P = \emptyset$ *et* $X = \emptyset$ **then**
  Déclarer R comme une clique maximale.
else if $P \neq \emptyset$ **then**
  Choisir le sommet u dans $P \cup X$ ayant le plus de voisin dans P.
  foreach $v \in P \backslash N(u)$ **do**
    Tomita($C \cup \{v\}, P \cap N(v), X \cap N(v)$)
    $P = P \backslash \{v\}$
    $X = X \cup \{v\}$
  end

Algorithm 1: Algorithme de Tomita

Dans cet algorithme, $N(v)$ représente l'ensemble des voisins du sommet v dans le graphe.

Cependant cette algorithme pose plusieurs problèmes :

- Cet algorithme est récursif, ce qui n'est pas permis en HLS. [9]
- Le choix du sommet u nécessite un grand nombre d'accès mémoire pour parcourir tout les sommets de $P \cup X$ et trouver celui ayant le plus de voisins dans P, cela est lent, car même si plusieurs accès mémoire peuvent être fait en parallèle, un accès mémoire se fait en deux

cycles d'horloge, le premier pour choisir l'adresse dans le bloc ram et le second pour lire la mémoire.
- Parcourir tous les sommets de $P \backslash N(u)$ peut être lent aussi en fonction de la représentation mémoire des ensembles dont celle qui a été retenue ici.

On pourra ensuite encore modifier l'algorithme afin de ne pas trouver toutes les cliques maximales mais une clique de poids maximum, ce qui nous permettra d'ajouter des conditions d'arrêt en fonction du poids ce qui n'est pas présent dans l'algorithme initial.

On sait de plus que la boucle de calcul n'aura pas un II=1 car il faut connaître le résultat du calcul pour commencer la suite.

Enfin, il nous sera possible de paralléliser différents calculs pour obtenir des temps d'exécution plus court en profitant du fait que sur un circuit matériel tout se fait réellement en parallèle.

3.1.1 *Représentation mémoire des différents éléments.*

La mémoire disponible est sous forme de bloc ram (BRAM) contenant des mots de 32 bits et deux actions (lecture ou écriture) peuvent être effectuées en simultanée.

On a choisit une taille maximale pour les graphes de 256 sommets et on note $N = 256$.

Les ensembles

Pour économiser le plus de mémoire possible, étant donner qu'il faut que l'utilisation mémoire ne dépende pas des éléments contenue dans l'ensemble, un ensemble est représenté par N bits tels que le $k^{\text{ème}}$ bit vaut 1 si et seulement si le $k^{\text{ème}}$ éléments appartient à l'ensemble. On sépare ces N bits en $N/32 = 8$ mots de 32 bits, eux même dispatchés en 4 BRAMs, ainsi, un ensemble entier peut être lu en une seule fois.

Le graphe

Le graphe est représenté sous forme de matrice d'adjacence, un tableau de N ensembles représentant les voisins de chaque sommets. Même si le graphe à moins que N sommets cela n'est pas dérangeant pour l'algorithme étant donné que les sommets inexistant sont représenté n'ayant pas de voisin et de poids nul.

Le nombre de sommet est uniquement utile à l'initialisation où pour un graphe à n sommets, numéroté de 0 à $n-1$, on a $C = X = \emptyset$ et $P = [\![0, n-1]\!]$. Initialisé avec ces ensembles évites des calculs inutile sur les sommets inexistant bien que cela ne change pas la correction de l'algorithme.

3.1.2 *Récursivité.*

Les algorithmes récursifs sont interdits lors de l'utilisation de la HLS. [9]

Dans cet algorithme, la particularité est que le résultat des appels récursif ne sont pas nécessaire pour continuer l'algorithme, on peut même effectuer les différents appels récursif dans un ordre quelconque voir après l'itération effectuant ces appels.

Une solution simple et fonctionnel pour éviter les appels récursifs, est de planifier les prochains appels en ajoutant les arguments de la fonction à une pile et de transformer les appels récursif en une boucle tant que cette pile est non vide.

3.1.3 *Choix du pivot.*

Choisir un pivot u dans $P \cup X$ tel que $|P \cap N(u)|$ nécessite un grand nombre d'accès mémoire ce qui augmente de beaucoup la latence de la boucle itérant sur la pile.

L'algorithme de Tomita fait ce choix de pivot car c'est celui qui minimise le nombre d'appel récursif.

Pour diminuer la latence créé par cela, on fait le choix de prendre comme pivot un élément quelconque de P bien que cela augmente le nombre d'itération de la boucle itérant sur la pile, le temps de calcul de la boucle diminue et et le nombre d'itération sera également diminué plus tard en ajoutant des conditions sur le poids de la clique. En effet, contrairement à l'algorithme de Tomita, on ne cherche pas toutes les cliques maximales mais une clique de poids maximum. De plus, l'ensemble X devient également inutile.

3.1.4 ***Parcours de tous les sommets de*** $P\backslash N(u)$.

À cause de la représentation des ensembles, parcourir tous les sommets de $P\backslash N(u)$ est lent et prend le même temps quelque soit son cardinal. En effet, il faut parcourir tous les entiers de 0 à $N-1$ et vérifier s'ils appartiennent à P et à $N(u)$, ce qui augment également la latence pour beaucoup d'appels où l'ensemble P est de cardinal faible.

La solution choisit ici est de ne pas parcourir tous les sommets de cet ensemble mais à la place d'empiler deux appels, soit en choisissant d'ajouter le sommet u à la clique, sans en choisissant de ne pas l'ajouter.

Pour garder le fait que si on décide de ne pas choisir le sommet u alors on sait que une clique maximale contient un sommet non voisin de u, on ajoute un masque et on choisir le pivot dans $P\backslash masque$ ce que prend le même temps de calcul que choisir le pivot dans P.

3.1.5 ***Conditions d'élagage***.

Nous allons ici ajouter des condition pour élaguer l'arbre de recherche de clique de poids maximum, cela pourrait être ajouté à l'algorithme initial, cependant il a pour but de trouver toutes les clique maximale et non une clique de poids maximum comme le nécessite l'algorithme de datapath merging.

Si le poids total $w(C) + w(P)$ est plus petit que le poids de la plus grande clique déjà trouvé jusqu'à maintenant, il est inutile de continuer cette exploration puisque l'on sait déjà que l'on ne trouvera pas de clique assez grande.

On obtient alors un algorithme ayant une latence de 6 cycles et également un II=6.

Le II s'explique par le fait que l'on a choisit une pile pour stocker les différents arguments et on a donc besoin d'attendre la fin de l'itération précédente pour commencer la suivante.

Cependant, on sait que la pile n'aura jamais plus de n éléments où n est le nombre de sommet du graphe, grâce au parcours en profondeur tandis que si on remplaçait la pile par une file, on change le parcours en un parcours en largeur qui peut donc avoir jusqu'à 2^n itérations de prévu, ce qui nécessiterais beaucoup de mémoire pour $n = 256$.

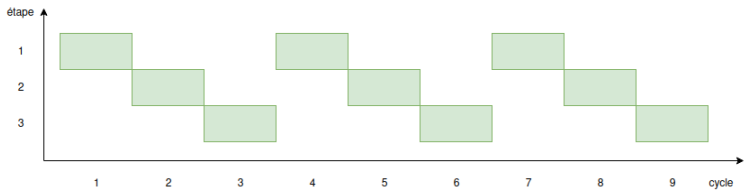

Fig. 5. Exemple de trace similaire à celle obtenue mais avec II=3.

L'exécution de notre boucle principale est alors similaire à celle de 5 mais se déroulant sur deux fois plus de cycles d'horloge, on voit alors que non seulement on perd du temps car on ne commence pas une itération à chaque cycle mais également que le circuit matériel est sous utilisé. en effet,

bien qu'une partie du matériel peut être utile à chaque étape de la boucle, cela ne sera pas le cas pour tout et il y a donc un coût matériel inutilisé dans ce circuit.

3.1.6 *État actuel*.

On a donc à ce point implémenté l'algorithme de référence en le modifiant pour obtenir d'une part une version plus adaptée à un circuit électronique et d'autre part en le spécialisant à la recherche d'une clique de poids maximale au lieu d'énumérer toutes les cliques maximales au sens de l'inclusion, ce qui est plus adapté à notre problème.

On remarque alors qu'il semble difficile en continuant comme ça de diminuer la latence jusqu'à 1 car dépiler et empiler prennent 2 cycles chacun et il est également difficile de pipeliner car il faut attendre le résultat de l'itération précédente pour commencer la suivante.

Cela a donc 2 conséquences, la première est la vitesse d'exécution, pour un même algorithme, avoir II=1 signifie que l'on commence une itération à chaque cycle de l'horloge ce qui augmente grandement la vitesse d'exécution. La seconde est que le circuit est utilisé que en partie à chaque fois et que chacune de ces parties est utilisé un cycle d'horloge sur six.

Une solution est d'effectuer des calculs indépendant en parallèle, ainsi, puisqu'il faut attendre la fin d'une itération pour commencer la suivante, si on en commence plusieurs indépendantes, on peut les alterner pour utiliser tout le circuit, il faut alors obtenir plusieurs calcul indépendant pour pouvoir accélérer l'exécution.

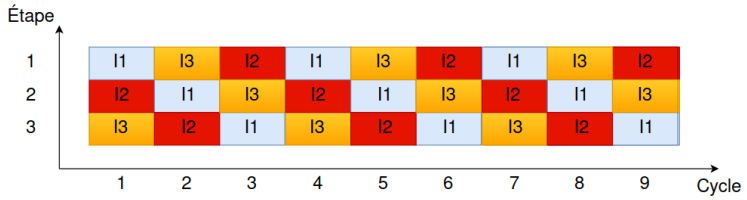

Fig. 6. Exemple de trace avec calculs entrelacés, latence=3 et II=1.

On a alors dans 6, contrairement à 5, un circuit qui est tout le temps utilisé et en plus, puisqu'on effectue plusieurs itérations en parallèle, on obtient un algorithme qui semble plus rapide.

3.1.7 *Pré-découpage du problème initial*.

On sait déjà qu'il existe une clique de poids maximale contenant au moins un sommet puisque les poids des sommets sont positifs.

Ainsi pour un graphe de sommets numéroté de 0 à $n-1$, il existe $k \in [\![0, n-1]\!]$ telle qu'une clique de poids maximale contient le sommet k et que tous les autres sommet de la clique soit dans $[\![0, k-1]\!]$.

On peut donc découper la recherche de clique de poids maximale en n appels indépendant, où pour k entre 0 et $n-1$ $C = \{k\}, P = [\![0, k-1]\!] \cap N(k)$ et $masque = \emptyset$.

On obtient alors n appels indépendant et il nous en fallait 6 pour utilisé le circuit complètement, même si 8 seront utilisés en pratique pour simplifier les calculs de divisions qui deviennent des décalages de bits et de modulo qui deviennent des & bits à bits.

où les sommets du graphe sont numéroté de 0 à $n-1$, $N(u)$ représente les voisins du sommet u et N_e est le nombre de calcul à entrelacer.

On a alors toujours une boucle de latence 6 cependant grâce au fait qu'il soit exécuté sur un circuit électronique, la boucle est pipelinée et on a II=1.

On a maintenant un algorithme ayant un fil d'exécution utilisé le plus possible et faisant les calculs 8 par 8. Cependant grâce au fait que l'on crée un circuit électronique dédié, on peut accélérer les calculs en ayant plusieurs fils d'exécution en parallèle. En revanche, contrairement aux calculs sur le même fil d'exécution, les accès simultanés à la réserve de calculs peuvent poser des problèmes.

3.1.8 ***Partage de la réserve.***

La réserve de calculs à effectuer est utile pour les différents calcul à entrelacer, ainsi tant qu'il y a des calculs à effectuer, le fil d'exécution est toujours presque plein. Cependant, chaque partie du circuit étant utilisé par un seul calcul à la fois, il n'y pas de risque d'accès simultané à cette réserve, ce qui n'est plus vrais avec plusieurs fils d'exécution.

Une première solution, la plus simple serait d'avoir une réserve par fil d'exécution et de répartir les appels initiaux entre ces dernières. Cependant, le temps d'exécution de certains appels est beaucoup plus court que pour d'autres et cela ne peut pas être su à l'avance car dépend d'une part des voisins de chaque sommet mais également des appels effectué précédemment à cause de la condition d'arrêt.

Une autre solution, plus compliqué à implémenter mais qui évite ce problème est d'avoir une seule réserve et un arbitre afin qu'un seul fil d'exécution puisse lire dans la réserve à la fois, ainsi, il est possible que de temps en temps les fils d'exécutions perdent quelques cycles, cependant on sait qu'il y aura toujours au un fil d'exécution qui a le droit de lire dans la réserve et qui sera donc complet. On

sait donc que malgré ces cycles perdu, avoir plusieurs fils d'exécution accélère strictement les calculs.

```
reserve = pile vide.
S = matrice de N_threads × N_e pile vide.
w_max = matrice de N_threads × N_e 0. // Poids de la plus grande pile trouvée.
C_max = matrice de N_threads × N_e ensemble vide. // Clique maximale trouvée
for k ∈ [[0, n − 1]] do
    empiler ({k}, [[0, k − 1]] ∩ N(k), ∅) dans reserve.
end
arbitre = 0
while reserve est non vide ou que toutes les piles de S sont non vides do
    for i ∈ [[0, N_threads − 1]] do
        // Toutes les itérations se font en parallèle.
        for j ∈ [[0, N_e − 1]] do
            // Une itération commence à chaque cycle.
            if S[i][j] est vide et arbitre = i then
                (C, P, masque) = dépiler reserve.
                empiler (C, P, masque) dans S[i][j].
            else if S[i][j] est non vide then
                (C, P, masque) = dépiler S[i][j].
                if w(P) + w(C) > w_max[i][j] then
                    if P = ∅ then
                        w_max[i][j] = w(C)
                        C_max[i][j] = C
                    else
                        Choisir u ∈ P\masque. if masque = ∅ then
                            empiler (C, P\{u}, N(u)) dans S[i][j]
                        else
                            empiler (C, P\{u}, masque) dans S[i][j]
                        empiler (C ∪ {u}, P\{u}, ∅) dans S[i][j].
        end
    end
    arbitre = (arbitre + 1)%N_threads
end
Rendre la plus grande clique de C_max
```

Algorithm 2: Algorithme modifié.

4 IMPLÉMENTATION PRATIQUE

On souhaite alors implémenter cet algorithme mais de manière à avoir un circuit matériel efficace. Pour un unique fil d'exécution, il est possible d'implémenter l'algorithme tel quel sans problème. Cependant, dès lors que l'on souhaite en avoir plusieurs, le flot de contrôle devient compliqué et la HLS ne crée pas les circuits les plus performants possible. Un autre problème est que si l'on essaie de lancer en parallèle plusieurs fonctions correspondant aux corps de la boucle, la HLS garde un flot de données synchrone et donc attend que tous ses appels soit terminé pour commencer les suivants, alors qu'il est possible d'en commencer un nouveau dès que l'un d'eux terminent.

Une solution à cela est d'avoir plusieurs coeurs de calcul indépendant qui seront transformés par la HLS indépendement les uns des autres et qui s'exécuteront en parallèle sur le FPGA tout en communiquant entre eux.

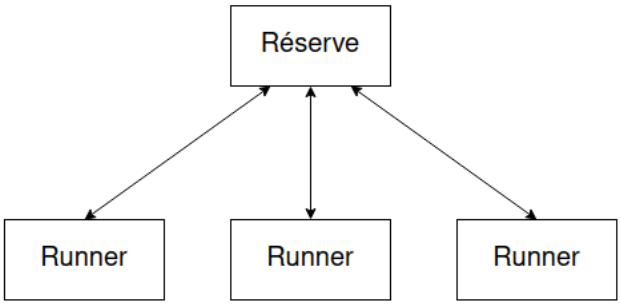

Fig. 7. Schéma du circuit visé pour 3 runners.

Le circuit matériel obtenue est alors similaire à celui de 7 où la réserve et chaque runner sont transformé indépendamment par la HLS.
On a alors comme fonctionnement :

Tant que la réserve est non vide, elle donne un calcul à effectuer à un runner disponible, puis récupère la plus grande clique obtenue lorsque ce calcul est terminé.

5 EXPÉRIMENTATION

Nous avons tester notre algorithme de recherche de clique de poids maximum sur différents graphes afin de comparer l'efficacité de notre accélérateur matériel avec l'efficacité du cliquer.

Cependant, le cliquer est un algorithme monothreadé alors qu'il semble également possible de paralléliser son exécution. Pour cela nous allons implémenter le même alorithme que celui du cliquer et le paralléliser afin de voir à quel point il pourrait en être parallélisé.

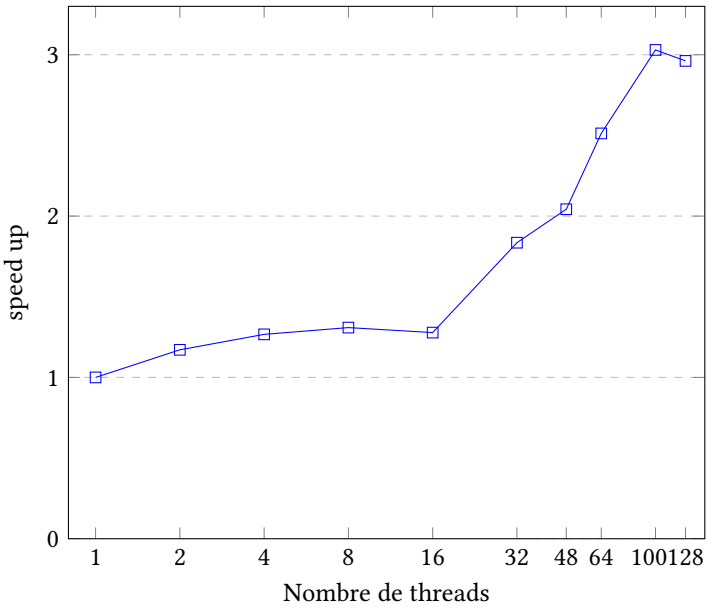

Fig. 8. Accélération potentielle du cliquer en fonction du nombre de thread

L'algorithme décrit en 2.2.2 peut se résumer au fait de trier les sommets du graphes en numérotant $[\![1, n]\!]$ puis rechercher pour tout k dans $[\![1, n]\!]$ la plus grande clique du sous-graphe de sommets $[\![1, k]\!]$ contenant le sommet k, puis il faut prendre la plus grande clique trouvée.

Ainsi, l'ordre dans laquelle ses calculs sont effectués change uniquement le temps de calcul et non le résultat. Il est donc possible d'effectuer plusieurs de ses calculs simultanément afin de paralléliser cet algorithme.

Le graphique de la figure 8, montre en moyenne le rapport entre le temps de calcul pour l'algorithme monothreadé avec le temps de calcul pour l'algorithme avec plusieurs threads en fonction du nombre de threads.

On voit alors qu'il serait possible de triplé la vitesse du cliquer.

Pour mesurer l'efficacité de notre accélérateur, nous allons comparer la vitesse de calcul de cliques de poids maximum pour certains graphe du benchmark DIMACS à la vitesse de calcul du cliquer.

Nous allons regarder la vitesse de calcul pour plusieurs quantités de runners différent afin de comprendre l'impacte de la parallélisation de cet algorithme.

Malheureusement, le temps de synthèse et de compilation étant de plusieurs heures, les résultats seront ici que partiel.

	cliquer	1 runner
brock200_1	797.99	245.64
brock200_2	17.01	272.9
C125.9	3871.29	1986.83
C-fat200-1	3.86	176.83
C-fat200-2	4.85	2148.71
hamming8-2	21.88	464.97
hamming8-4	2.18	223.99
MANN-a9	23.07	2222.42
san200_0.7_1	27518.93	2403.23

Table 1. Temps moyen en ms pour des graphes du benchmark DIMACS

Le tableau 1 donne ainsi le temps moyen de calcul des différentes cliques de poids maximum en ms pour le cliquer et pour notre accélérateur avec un runner. On remarque qu'avec un seul runner certaines recherches de clique sont plus lente et d'autres plus rapide mais il semblerait que le temps minimal soit plus grand, certainement à cause de la communication avec le FPGA qui est plus lente que de rester sur le processeur.

Il faudrait pour obtenir des résultats plus complet, effectuer des tests pour plusieurs quantité de runners. Le nombre maximal possible dépend d'une part des composants disponible (BRAM, LUT, FF), dans notre cas cela nous permettrait de placer 38 runners. Mais également du routage, car lors du placement des composants sur la carte, il faut créer les chemins sur lesquels seront transmisent les données et si la carte est très remplie cela peut être compliqué.

6 CONCLUSION

Ainsi, on a cherché à accélérer la recherche de clique de poids maximum sur un FPGA en utilisant la HLS. Cette accélération a pour but d'accélérer la fusion de chemin de données qui est elle même utilisé par la HLS pour rendre les circuits matériels plus petit.

Pour cela nous avons réarrangé un algorithme de branch and bound existant en essayant de le paralléliser au maximum afin de le rendre efficace pour implémentation sur FPGA bien que cela ne soit pas l'utilisation usuelle des FPGAs car nécessite peu de calcul et beaucoup d'accès mémoire.

Il est pourtant ainsi possible d'avoir des accélérateurs matériels efficaces pour cette recherche de clique.

De plus, il serait possible par la suite pour améliorer cet accélérateur de spéculer sur l'algorithme de branch and bound en supposant que l'on va continuer d'avancer dans l'arbre de recherche, et annuler cela s'il faut remonter.

Le travail effectué dans ce stage à ouvert des pistes de choses qui pourrait être fait dans le futur :

- Lors de l'algorithme de Moreano et al. pour la fusion de chemins de données, les chemins de données sont fusionnés 2 à 2. Cependant, il serait possible d'adapter l'algorithme afin de créer un graphe de compatibilité de tout les chemins de données en même temps pour obtenir la meilleure fusion finale en une seule étape car le résultat de l'algorithme actuel dépend de l'ordre dans lequel est fusionné les chemins de données.
- Il serait également possible en ajoutant des informations au graphe de compatibilités de modifier la recherche de clique de poids maximum en une recherche de clique de poids maximum telle que le chemin de données fusionnées ne contiennent pas de cycle, car un chemin de données contenant un cycle n'est pas valable.

- La procédure suivie dans ce stage afin d'obtenir une accélération de l'algorithme de recherche de clique de poids maximum semble pouvoir être valide pour beaucoup d'algorithme de branch and bound. Ainsi, il semblerait possible de généralisé le travail effectué ici.

REFERENCES

[1] Moreano, N., Bonn, E., De Souza, C., Araujo, G. (2005). Efficient datapath merging for partially reconfigurable architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(7), 969–980. https://doi.org/10.1109/TCAD.2005.850844

[2] Erdös, P. (1966). On cliques in graphs. Israel Journal of Mathematics, 4(4), 233–234. https://doi.org/10.1007/BF02771637

[3] Fazlali, M., Fallah, M. K., Hosseinpour, N., & Katanforoush, A. (2019). Accelerating datapath merging by task parallelisation on multicore systems. International Journal of Parallel, Emergent and Distributed Systems, 34(5), 615–628. https://doi.org/10.1080/17445760.2018.1552957

[4] Conte, A. (2013). Review of the Bron-Kerbosch algorithm and variations.

[5] Korkeakoulu, T. (2003). CLIQUER USER ' S GUIDE Version 1 . 0 HELSINKI UNIVERSI1Y OF TECHNOLOGY.

[6] Östergård, P. R. J. (2002). A fast algorithm for the maximum clique problem. Discrete Applied Mathematics, 120(1–3), 197–207. https://doi.org/10.1016/S0166-218X(01)00290-6

[7] Bernard, J., & Seba, H. (2017). Résolution de problèmes de cliques dans les grands graphes État de l ' art La compression de graphes. 2013, 37–48.

[8] Xin, H., Zhong, M., Liu, Q., & Wang, M. (2020). List all maximal cliques of an undirected graph: A parallable algorithm. IOP Conference Series: Materials Science and Engineering, 790(1). https://doi.org/10.1088/1757-899X/790/1/012076

[9] Ryan Kastner, Janarbek Matai, S. N. (2018). Parallel Programming for FPGAs, The HLS Book. Book, 53(9).

[10] Derrien, S., Marty, T., Rokicki, S., Yuki, T., Derrien, S., Marty, T., Rokicki, S., Yuki, T., Speculative, T., Pipelining, L., Derrien, S., Marty, T., Rokicki, S., Yuki, T. (2020). Toward Speculative Loop Pipelining for High-Level Synthesis To cite this version : HAL Id : hal-02949516 Toward Speculative Loop Pipelining for High-Level Synthesis.