Mémoire de recherche

Optimisations C++ en contexte de Synthèse de Haut Niveau sur FPGA

Par

Benjamin QUIEDEVILLE

Encadrement :

Tanguy Risset `tanguy.risset@inria.fr`
Romain Michon `romain.michon@inria.fr`

Organisme :

Inria, laboratoire CITI
56 Boulevard Niels Bohr, 69 603 Villeurbanne

Dates du stage :

15/03/2024 - 31/08/2024

Rapport non confidentiel

Résumé

La principale limitation des traitement numériques audio, intrinsèque au calculs sur processeurs classiques, est la latence que ces traitement introduise. La latence est le temps pris par une quantité de données pour être traitée par l'algorithme implémenté. En audio, cette latence est perçue comme un retard de quelques millisecondes dans les cas les plus favorables et quelques centaines de millisecondes dans les plus mauvaises situations, pouvant gêner la performance musicale. Les FPGA (Field Programmable Gate Array, réseau de portes logiques programmable in situ) se posent comme candidats idéal pour la réduction drastique de la latence. Ces sont des circuit utilisés dans les domines industriels et militaire pour réaliser des calculs temps réel à des fréquences d'échantillonnages du domaine radio (de l'ordre du Mega Hertz), il semblent donc parfaitement adéquats pour des calculs audio de l'ordre de la dizaine de kilo Hertz. Ces FPGA sont cependant beaucoup plus difficiles à programmer que les processeur usuels, ils se programment à l'aide de langage de description matérielle (VHDL, Verilog) et n'offre que peu d'abstraction pour exprimer les algorithmes. Les fabricants de FPGA développent alors des outils de HLS (High Level Synthesis, Synthèse de Haut Niveau) pour exprimer les algorithmes dans un langage plus haut niveau comme Matlab, C ou C++ tout en conservant les avantages de ces circuits. Ce mémoire du Master ATIAM présente donc des travaux dans le cadre du projet Syfala de l'équipe Emeraude du centre Inria de Lyon, encadrés par Tanguy Risset et Romain Michon.

The main limitation of digital audio processing, directly related to the way CPU perform computations, is latency. Latency is the time it takes for a given amount of data to be processed by the implemented algorithm. In audio, this latency is perceived as a delay of between a few milliseconds in the best cases and hundreds of milliseconds in the worst ones and can disturb the musical performance. FPGAs (Field Programmable Gate Array) are therefore great candidates for the reduction of latency. These are circuits that are used in industrial and military situation to perform real time computation at sampling rates in the radio domain (Mega Hertz), they seem fit for audio computation for sampling rate in the tens of kilo Hertz. But FPGAs are way more difficult to program than usual processors, they require to use Hardware Description Languages (HDL) that do not offer enough abstraction to express the algorithms. FPGA companies are then developing HLS (High Level Synthesis) tools to express programs in higher level languages such as Matlab, C or C++ while still providing the same performance gains of these circuits. This ATIAM master's thesis presents results of researches in the context of the Syfala project of the Emeraude Team at INRIA in Lyon, France, supervised by Tanguy Risset and Romain Michon.

Mot clés : Audio, algorithmes temps réel, latence basse, programmation embarquée FPGA, Vitis HLS, optimisations, C++, Faust.

Keywords : Audio, real time algorithms, low latency, embedded programming FPGA, Vitis-HLS, optimisations, C++, Faust.

Remerciements

Je souhaite tout d'abord remercier mes encadrants Tanguy Risset, Romain Michon ainsi que Pierre Cochard, ingénieur sur le projet Syfala pour leur accueil et encadrement durant ce stage, j'ai beaucoup appris de leurs enseignements sur la programmation audio sur FPGA et ai été très honoré de travailler à leurs côtés. Je remercie également Linda Soumari et Cecilia Navarro pour leur accueil dans le laboratoire CITI et leur aide avec les problèmes administrative. Également, je veux remercier Cyrielle Fiolet pour son aide administrative côté IRCAM ainsi que tout le corps enseignant du master ATIAM pour leurs enseignements, ce master a confirmé ma passion pour la programmation audio et je suis maintenant sûr de vouloir poursuivre ma carrière dans ce domaine. Je voudrais aussi remercier Victor, Flore, Mathieu, Ninon, Laura, Cyrielle et Damien pour leur chaleureux accueil dans notre résidence à Villeurbanne durant ce stage. Je souhaite enfin remercier très chaleureusement Romain, Tanguy et Stéphane Letz pour la confiance qu'il m'accorde et de me permettre de débuter au mois de septembre un doctorat sur le langage Faust. Je mûris le projet d'un doctorat depuis le début de mes études supérieures et c'est un honneur de pouvoir prendre part à ce projet.

Table des matières

Table des figures

Liste des tableaux

Liste des Pseudo-codes

Introduction

La latences introduite par les traitements audio numériques est au cœur des recherches d'optimisations. Cette latence est un des principaux paramètres de validité des outils musicaux, elle introduit un délai entre la consigne du musicien et la réponse du dispositif qui, quand il devient trop long (il est devient perceptible à partir de 20 millisecondes environ) perturbe grandement la performance musicale.

Les FPGA (Field Programmable Gate Array, réseau de portes logiques programmable in situ) se posent comme candidats idéal pour la réduction drastique de la latence. Ces sont des circuit utilisés dans les domines industriels et militaire pour réaliser des calculs temps réel à des fréquences d'échantillonnages du domaine radio (de l'ordre du Mega Hertz), il semblent donc parfaitement adéquats pour des calculs audio de l'ordre de la dizaine de kilo Hertz. De plus, une fois programmés, ces circuit requièrent moins de puissance qu'un processeur classique pour réaliser leurs calculs, on peut donc les embarquer facilement dans des dispositifs ayant peu de puissance d'alimentation. Grâce à leur architecture immuable après programmation, les FPGA ont naturellement moins de possibilités de comportements et sont donc moins sujets à des bogues de fonctionnement dus à des erreurs de programmations comme des corruption de mémoire. Ils sont donc fiables dans des environnements critiques ou les erreurs à l'exécution causeraient d'importants dégâts. Ces FPGA sont cependant beaucoup plus difficiles à programmer que les processeur usuels. Ils se programment à l'aide de langage de description matérielle (VHDL, Verilog) et n'offre que peu d'abstraction pour exprimer les algorithmes. Les fabricants de FPGA développent alors des outils de HLS (High Level Synthesis, Synthèse de Haut Niveau) pour exprimer les algorithmes dans un langage plus haut niveau comme Matlab, C ou C++ tout en conservant les avantages de ces circuits. Ce mémoire du Master ATIAM présente donc des travaux dans le cadre du projet Syfala de l'équipe Emeraude du centre Inria de Lyon, encadrés par Tanguy Risset et Romain Michon.

Ce rapport commence par un état de l'art de la programmation audio sur FPGA, une revue des outils de Synthèse de Haut Niveau ainsi qu'une présentation de l'équipe Emeraude et de son projet Syfala en lien avec Vitis HLS, l'outil de synthèse développé par Xilinx. Ensuite, seront abordés la description de la plateforme cible et de l'environnement de travail puis une étude de trois algorithmes servants de support pour l'étude des optimisations. C'est algorithmes sont : un égaliseur paramétrique, un filtre adaptatif NLMS et une simulation de réverbération à plaque par différences finies.

1 État de l'art

1.1 Programmation audio des FPGA

La numérisation des algorithmes de traitement du signal, quand il permettent une réduction de la taille des dispositifs et une meilleure maintenance, viens avec l'ajout d'une latence entrée-sortie [1] du signal dans le dispositif. Cette latence entrée-sortie est en partie provoquée par la conversion analogique-numérique et numérique-analogique et principalement par la mise en mémoire tampon du signal audio pour réduire la charge computationelle. Réduire au minimum la latence des algorithmes est donc au centre de la programmation audio [11]. Dans des situation de programmation embarquée, on peut proposer de remplacer le matériel classique par un FPGA pour ses capacités de parallélisation et sa haute fréquence de calcul.

Inventés par la société Xilinx puis commercialisés à partir de 1985, les FPGA (Field Gate Programmable Array, réseau de portes logiques programmables in situ), sont des circuits se démarquant des technologies traditionnelles de programmation. Ces circuits proposent une architecture de grille de modules dont les connections sont modifiable en fonction du programme à exécuter. Les FPGA sont employés dans des domaines de traitement de signal à haute fréquence d'échantillonnage (domaine radio) ou bien dans le traitement d'image[9], mais en audio, on les trouve également au cœur du protocole Dante pour la transmission de signal, dans certaines tables de mixage numériques professionnelles telle la Midas M32 ou dans certains synthétiseurs numériques tel le Novation Summit, le UDO Super Gemini ou le Waldorf Kyra.

1.2 La Synthèse de Haut niveau (HLS)

Cependant, les langages de description matériel sont très difficiles à utiliser et ne permettent pas le niveau d'abstraction nécessaire pour exprimer efficacement les algorithmes audio. Des outils de Synthèse de Haut Niveau (HLS, High Level Synthesis) ont donc été développés pour programmer les FGPA avec des langages proposant des niveaux d'abstraction plus élevés. Mathworks propose l'outil *HDL Coder* [5] générant du VHDL et du Verilog à partir de scripts Matlab. Intel fabrique des FPGA et distribue son p ropre compilateur et les outils de HLS [4]. Xilinx développe la chaine de compilation Vitis-HLS au sein de l'écosystème Vivado [14] sur lequel se base le projet Syfala. Enfin des travaux sur des alternatives ouvertes de synthèse et création de systèmes sur FPGA sont mis en œuvre pour rendre le domaine accessible [2, 10]. On peut également mentionner le projet visant à compiler sur FPGA des programmes audio temps-réel Pure Data [13]. Il est donc aujourd'hui possible d'implémenter des algorithmes sur un FPGA à l'aide de langages de programmation généralistes, principalement C/C++.

1.3 Le projet Syfala

Le projet Syfala [7, 8] de l'équipe *Emeraude* du centre INRIA de Lyon s'inscrit dans ce domaine en proposant une chaîne d'outil pour développer et compiler des programmes audio sur des circuits FPGA en se basant dans l'environnement Vitis HLS de Xilinx. L'équipe Emeraude (EMbEdded pRogrammable AUDio systEms) [2], dirigée par Tanguy Risset, est une équipe du centre INRIA de Lyon faisant partie du laboratoire CITI (Centre d'Innovation en Télécommunication et Intégration de services), laboratoire collaborant avec l'INSA Lyon et l'INRIA. L'équipe s'oriente principalement sur les implémentations arithmétiques efficaces pour les calculs à haute performances dans des contextes de traitement du signal numérique et d'apprentissage automatique sur FPGA. Le projet Syfala, un des deux principaux de l'équipe avec Flopoco [3], spécialise ces travaux pour le traitement audio sur FPGA.

Le projet a pour objectif de réaliser cette compilation depuis le langage C++ et depuis le langage spécialisé Faust [6] : le compilateur traduit le programme Faust en C++ qui est ensuite

1. La latence entrée-sortie est la durée séparant l'entrée de l'information dans le programme de sa sortie. En traitement audio, c'est le temps écoulé entre l'entrée d'un échantillon audio, et la sortie du résultat.
2. `https://team.inria.fr/emeraude/`
3. `https://www.flopoco.org/`

traduit en VHDL via Vitis-HLS. Le langage *Faust* (Functionnal AUdio STream) principalement développé par Yann Orlarey et Stéphane Letz, est un langage de programmation spécialisé pour la programmation d'algorithme de traitement audio et la création d'instruments de musique numériques. C'est un langage visant à faciliter l'écriture des algorithmes audio, puis à traduire le résultat vers de nombreuses cibles (C++, Rust, Web Assembly...). Le projet Syfala vise entre autres donc à améliorer la compilation de *Faust* vers les FPGA.

Le projet a par ailleurs proposé plusieurs avancées dans l'état de la HLS audio actuelle, notamment le support du protocole Open Sound Control (OSC) offrant un canal de communication entre programmes musicaux, ainsi qu'une interface pour réaliser des algorithmes sur plusieurs échantillons à la fois. Le stage consiste donc en la recherche des optimisations au niveau des algorithmes pour que Vitis-HLS synthétise une description matériel tirant partie des avantages de la structure d'un FPGA.

2 Description de la plateforme cible

2.1 Fonctionnement d'un FPGA

Le FPGA est une plateforme très différente des processeurs usuels que l'on trouve dans les systèmes informatiques. Au lieu de reprogrammer le processeur en lui fournissant une série d'instructions correspondant à notre programme, le FPGA est une matrice de différents modules de calcul et de stockage et le programme vient recomposer la cartographie de ces portes logiques, spécialisant ainsi le matériel pour le programme cible. L'architecture finale du matériel lui permet d'être hautement spécialisé pour une tâche précise qu'il pourra effectuer avec de grandes performances. Dans le cadre d'applications de traitement du signal audio, l'équipe Emeraude a pu réduire la latence entrée-sortie de certains programmes à $11\mu s$ [8] (pour des processeurs usuels, elle est de quelques millisecondes). La programmation d'un FPGA prend donc en compte la gestion des différentes ressources présentes sur le circuit. Ce sont ces ressources qui dictent quels programmes pourront être intégrés sur le matériel et leur quantité varie en fonction du modèle utilisé. Ces ressources sont en réalité des sous-circuits logiques remplissant des rôles distincts dans l'exécution du programme. Ces nombreux modules sont disposés sur une matrice appelée "logic fabric" (tissu logique) et la programmation du circuit consiste à réorganiser le routage des connections entre ces modules On distinguera quatre types de modules principaux sur un FPGA :

- DSP (Digital Signal Processor) : Ce sont des blocs implémentant des fonctions complexes pour le traitement numérique de signaux, il serviront par exemple à calculer des fonctions trigonométrique. Ils sont généralement peu nombreux sur un FPGA, on souhaitera les utiliser judicieusement.
- FF (Flip-Flop, bascules) : Ce sont des modules permettant de réaliser des branchement dans les instruction et des opérations de logique synchrones.
- LUT (Look Up Tables, tables de correspondance) : Ce sont des modules polyvalents pouvant stocker des données, réaliser des opération booléennes complexes et également fournissant des valeurs pré-calculées pour les résultats de certaines fonctions.
- BRAM (Block Random Access Memory) : Ce sont des modules servant uniquement au stockage de données directement sur le tissu logique, ils ont une meilleure capacité que les LUT et offrent une bonne vitesse d'accès.

Enfin, la dernière ressource principale, est la latence du programme. Différente de la latence entrée-sortie, cette latence exprime si le FPGA est en capacité d'effectuer l'algorithme demandé dans le temps imparti par son horloge interne (de l'ordre de la centaine de Mega Hertz). Les mesures de ressources sont données en pourcentage du maximum, si une mesure dépasse 100%, l'algorithme ne peut être implémenté sur le FPGA.

La programmation des FPGA par les langages de description matérielle tel VHDL et Verilog se fait au niveau d'abstraction appelé le RTL. Le RTL, pour Register-Transfert Level (Niveau Transfert de Registre), est le niveau d'abstraction de la description matériel impliquant des registres contenant des signaux et les blocs de logique combinatoire agissant et combinant ces signaux. Dans le contexte de HLS, le RTL est la cible de la première étape de compilation, et une partie d'analyse importante puisque c'est ici que les optimisations principales prennent place.

Certains FPGA sont inclus dans un "Système sur puce" (ou SoC pour *System on Chip*). Un SoC consiste en un système complet disposé sur un unique circuit intégré, à dire un microprocesseur, une mémoire, les dispositifs d'entrée-sortie nécessaires et dans notre situation : un FPGA. Ceci permet l'interaction entre le FPGA et le processeur pour la répartition des calculs, pour le stockage des donnée ou la communication avec des systèmes extérieurs requérant un protocole précis. Il est également possible d'exécuter un système d'exploitation complet sur le processeur et ouvrir d'avantage les possibilités d'interaction. Ce dernier point sera développé plus en détail en 2.2.

Resource	Valeur
Horloge interne	125 MHz
Mémoire DDR3	512 Mo
Mémoire flash (partagée)	16Mo
Fréquence d'échantillonnage	48 kHz
Block DSP	80
Look Up Tables (LUT)	53 200
Flip-Flops (FF)	106 400
Block RAM (BRAM)	630 Ko

TABLE 1 – Caractéristiques du matériel cible, ces grandeurs guideront les optimisations dans le cadre de ce travail.

2.2 Structure globale de la plateforme (exemple avec la zybo Z7020)

Le matériel utilisé lors de ces travaux est une carte de développement distribuée par Digilent, le modèle *Zybo Z7-20*[4] basé sur le FPGA *ZYNQ-7020 Development Board*. Dans cette partie, nous listerons les caractéristique de cette carte précise puisqu'elle est notre cible de déploiement. Si l'on change de FPGA, les caractéristiques ne seront pas les même et les approches pour l'implémentation des algorithmes seront différentes. La Zybo Z7-20 embarque un système sur puce composé d'un FPGA et d'un processeur ARM communiquant entre eux via des canaux dédiés et une mémoire partagée. Le tableau 2.2 regroupe les caractéristiques importantes du FPGA qui servent de guides à la programmation. De ces valeurs nous pouvons déterminer des contrainte de performances pour nos programmes. Par exemple en divisant la fréquence d'horloge par la fréquence d'échantillonnage de travail, nous obtenons le nombre de cycles séparant deux échantillons audio, soit le nombre maximal de cycle que nos traitement doivent prendre pour traiter et produire un échantillon. Nous obtenons ici une valeur de 2604 cycles. Le nombre de cycles que prend un programme à s'exécuter est la latence, nous pouvons donc obtenir un pourcentage de la latence maximale pour nos programmes.

La carte Zybo dispose de 16Mo de mémoire partagée entre le processeur ARM et le FPGA, cette mémoire est très pratique pour stocker une quantité de données conséquente et soulager les ressources du FPGA, mais elle est aussi plus lente d'accès et doit être utilisé de manière optimisée. Les accès à la mémoire partagée se font plus efficacement par ordre d'adresses croissantes et contigües. De même, il est plus intéressant d'accéder directement à une plus grande mémoire et de la rapatrier d'un coup sur le FPGA car cette mémoire est optimisée pour fonctionner comme cela, c'est ce que Xilinx appelle le *Burst d'accès mémoire*.

En figure 1 est visible une représentation schématisée de la structure globale et son utilisation dans le contexte du projet Syfala. On y distingue cinq principaux composants.

— Le FPGA : cellule principale de calcul
— Le codec audio : effectue les conversions analogique-numérique et fournit les échantillons audio au FPGA. Les échantillons sont transmis sous la forme de tableau en C appelés *audio_in* et *audio_out*.
— La mémoire partagée : stocke des valeurs pour le FPGA et sert de transmetteur de données entre le FPGA et le processeur ARM. La mémoire partagée est accessible depuis la HLS par les variables *mem_zone_f* et *mem_zone_i* qui sont deux pointeurs vers les zones mémoire concernées.
— Le micro-processeur ARM : peut initialiser la mémoire de la carte dans un certain état, réduire la charge de travail du FPGA en calculant des informations en amont. Dans le cadre du projet, une version du système d'exploitation *Alpine Linux* a été installée pour ouvrir et communiquer avec des machines externes via un canal Open Sound Control (OSC)[5]
— Le serveur OSC permet une interaction entre le micro-processeur ARM et un programme externe, par exemple Pure Data sur la figure 1.

4. `https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual`
5. `https://opensoundcontrol.stanford.edu/index.html`

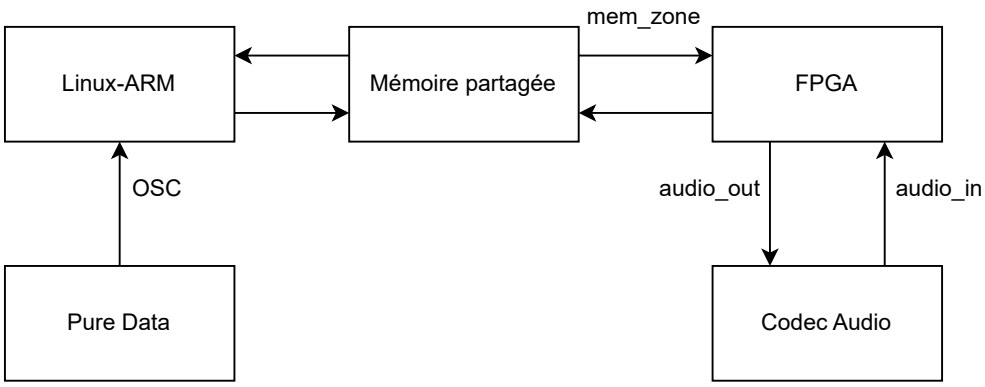

FIGURE 1 – Schéma global de la structure de la plateforme de développement

3 Outils de compilation

3.1 Vitis HLS

L'environnement de Synthèse de haut niveau utilisé dans Syfala se base sur l'outil Visit HLS de Xilinx. C'est une suite complète d'outil pour la programmation des FPGA en C++ comportant autant une interface textuel dans une console ou une interface plus détaillé accompagnant l'Environnement de Développement Intégré (IDE). Le procédé ce compilation se déroule comme suit :

— Vérification de la syntaxe du code source HLS. Vitis HLS utilise une version du compilateur C++ GCC pour contrôler si le code C++ est valide.
— Traduction vers le langage VHDL, langage utilisé pour exprimer le RTL. Certaines optimisations transmises par le programmeur sont appliquées ici.
— Synthèse, le procédé de transformation du schéma RTL en graphe décrivant la hiérarchie des différents modules disponible sur le FPGA pour réaliser les calculs. Il s'agit de l'étape la plus longue de la compilation, pouvant aller jusque plusieurs heure pour les programmes les plus complexes.
— Placement et routage : ici le compilateur viens traduire chaque nœud du graphe synthétisé en une ressource physique sur le tissu logique (c'est le placement) et chaque transfert de signal en une connection entre modules (le routage). Cette étape peut être exécutée plusieurs fois pour raffiner le résultat et optimiser l'exécution.
— Génération du fichier de programmation : c'est la dernière étape, elle génère un fichier contenant toutes les informations pour programmer le FPGA selon la description du RTL. On peut ensuite télé-verser ce fichier sur la carte pour reprogrammer le FPGA et exécuter le programme.

Durant toutes ces étapes, Vitis HLS génère un rapport de compilation, qui représente sa principale interface utilisateur. C'est dans ce rapport que l'on pourra voir les erreur de compilation C++ mais surtout les rapports d'optimisations. C'est le seul endroit où l'on pourra voir si une optimisation spécifiée dans le code de HLS est réalisée convenablement ou si elle pose problème pour la génération du code VHDL. Pour appliquer ces optimisation, Vitis HLS fait utiliser des directives *pragma* dont les principales utilisées sont listées en partie4.1.

Vitis HLS permet en théorie d'utiliser des construction de programmation correspondant au standard C++14. En réalité, beaucoup de ces formulations ne seront pas acceptée lors de la compilation dû aux spécificité de la plateforme concernée. En effet, dans le code HLS, il est impossible de réaliser d'allocation mémoire dynamique par exemple, la taille des objets devant être connue à la compilation. De même, les méthodes de programmation orientées objets et de manière générale les construction très haut niveau sont découragées puisque l'on a moins de contrôle sur les instructions générées par le code, on souhaite que le maximum de ressources du FPGA soient utilisées pour réaliser les calculs et non pour gérer les objets qui réalisent les calculs. Il convient donc d'utiliser un style de programmation plus pragmatique et impérative, ca on ne pourra également pas compter sur les optimisations automatique d'un compilateur C++ classique. Dans notre situation, la quantité de ressources disponibles étant limité, les programmes n'atteindront que rarement la complexité pouvant requérir ces outils. Une autre limite aux constructions autorisées dans le code est liée à la gestions des pointeurs, omniprésents en C++. Bien que Vitis HLS permette l'utilisation d'un pointeur vers un objet, il est impossible d'avoir un pointeur vers un autre pointeur pour échanger par référence la gestion d'un objet (problème qui sera abordé en partie 4.4) ou simplement d'utiliser un pointeur vers une zone mémoire quelconque, aucune mémoire ne peut être allouée dynamiquement.

3.2 Environnement de programmation Syfala

L'interface de développement Syfala est une enveloppe autour de Vitis HLS. Le projet nous fournis un outil en ligne de commande pour appeler facilitant la compilation des programmes. Syfala expose également un outil simple pour compiler un programme à télé-verser sur le processeur ARM embarqué sur le système. Cela ouvre la possibilité d'ouvrir par exemple un canal

OSC entre la carte et un autre programme compatible sur une autre machine pour transmettre des informations en temps réel. Le canal OSC sera utilisé dans la partie 4.2 pour contrôler les paramètres du programme pendant l'exécution.

Avec Syfala, le programmeur peut programmer des traitement avec une fenêtre temporelle d'un échantillon, c'est à dire que le programme accepte en entrée un unique échantillon et non un groupe de 128 (par exemple) comme dans les application classiques sur CPU pour économiser du temps de calcul au détriment de la latence. Cela permet d'obtenir des valeurs de latence entrée-sortie de l'ordre de la micro seconde à une fréquence d'échantillonnage de 48 kHz. Mais dans des situations où le FPGA ne peut réaliser les calculs dans le temps imparti, autrement, si la latence du programme dépasse celle impartie par le matériel (voir 2.2), on peut faire appel à cette même méthode et effectuer les calculs sur plusieurs échantillons en série et faire baisser la latence du programme au détriment de la latence entrée-sortie du programme. Syfala permet d'injecter ce comportement dans le code avec une simple adaptation du code source et un argument à passer en ligne de commande à la compilation, on doit certes réaliser une copie du code source mais nul besoin de tout reconfigurer.

Syfala facilite également l'utilisation de la mémoire partagée en exposant dans le point d'entrée, visible dans le pseudo-code 1, les deux pointeurs *mem_zone_f et mem_zone_i* pointant respectivement vers des zones mémoire constituées de valeurs à virgule flottante 32bits et d'entier 32bits. Ces zones mémoire sont allouées et initialisées par le processeur ARM et d'autres zone de mémoire peuvent être allouées et transmise au code Syfala. Le pseudo-code 1 expose la forme du point d'entrée dans un programme HLS de Syfala avec une phase d'initialisation, d'acquisition des données d'entrée, de traitement et d'écriture des données.

3.3 Environnement de simulation

L'environnement Vitis HLS met à disposition un outil de test des algorithmes C++. Cet outil, la CSIM (C Simulation) est une version du compilateur GCC compilant et exécutant le programme dans un environnement garantissant la validité de l'algorithme HLS. Syfala propose une enveloppe autour de la CSIM pour y ajouter des outils d'import et export de fichiers audio, permettant de contrôler précisément les données d'entrée et de visualiser les résultats du programme. On obtient ainsi un banc de test offrant une excellente répétabilité. Syfala intègre aussi un modèle de CSIM facilitant la création de fichiers de banc de test.

```
#define INPUTS 2
#define OUTPUTS 2

bool initialization = true;

void syfala(
sy_ap_int   audio_in[INPUTS],
sy_ap_int   audio_out[OUTPUTS],

int         arm_ok,
bool*       i2s_rst,
float*      mem_zone_f,
int*        mem_zone_i,

bool        bypass,
bool        mute,
bool        debug
)
{
    #pragma HLS array_partition variable=audio_in type=complete
    #pragma HLS array_partition variable=audio_out type=complete
    #pragma HLS INTERFACE s_axilite port=arm_ok
    #pragma HLS INTERFACE m_axi port=mem_zone_f latency=30 bundle=ram
    #pragma HLS INTERFACE m_axi port=mem_zone_i latency=30 bundle=ram

    *i2s_rst = !arm_ok;

    if (!arm_ok) { return; }

    if (initialization) {
        // initialisation de certaines donnees, ne s'execute que lors du
        // premier appel du point d'entree

        initialization = false;
        return;
    }

    if (bypass || mute) {
        audio_out[0] = 0;
        audio_out[1] = 0;
        return;
    }

    signal_sample_left  = Syfala::HLS::ioreadf(audio_in[0]);
    signal_sample_right = Syfala::HLS::ioreadf(audio_in[1]);

    // algorithme

    Syfala::HLS::iowritef(signal_sample_left,  audio_out[0]);
    Syfala::HLS::iowritef(signal_sample_right, audio_out[1]);
}
```

Pseudo-code 1 – Example de point d'entrée d'un programme Syfala

4 Algorithmes supports des optimisations

4.1 Presentation des mécanismes d'optimisation

Dans cette sous-partie, serons présentées les principales optimisations étudiées durant ce travail. Ces optimisations portent visent principalement à réduire le temps de calcul des boucles, omniprésentes dans les traitements audio. Le chevauchement des différentes itérations de boucle permet d'accélérer

4.1.1 Chevauchement des instructions

C'est une optimisation d'exécution qui est également présente dans les processeurs classiques modernes. Quand plusieurs signaux doivent passer dans une même suite d'instructions, dans le cas d'une boucle par exemple, il est possible de commencer le traitement du signal $n + 1$ avant que le signal n ne soit sorti de la chaîne. Ce phénomène est illustré en figure 2, pour un cas à quatre instructions exécutées séquentiellement. Dans le premier cas, mettons que chaque instruction nécessite un unique cycle d'horloge, la séquence complète prend donc quatre cycles. Traiter plusieurs signaux prend donc un multiple de quatre cycles à compléter. On peut alors remarquer que quand l'instruction A a traité le premier signal, il est possible de la réutiliser directement pour le deuxième signal, nul besoin d'attendre que toute la première séquence soit terminée. Dans l'exemple en figure 2, on passe de huit cycle pour deux itérations à cinq cycles avec le chevauchement, et donc six cycles pour trois itérations. Chaque itération nécessite toujours quatre cycles pour s'exécuter, mais on peut commencer une nouvelle itération tout les cycles.

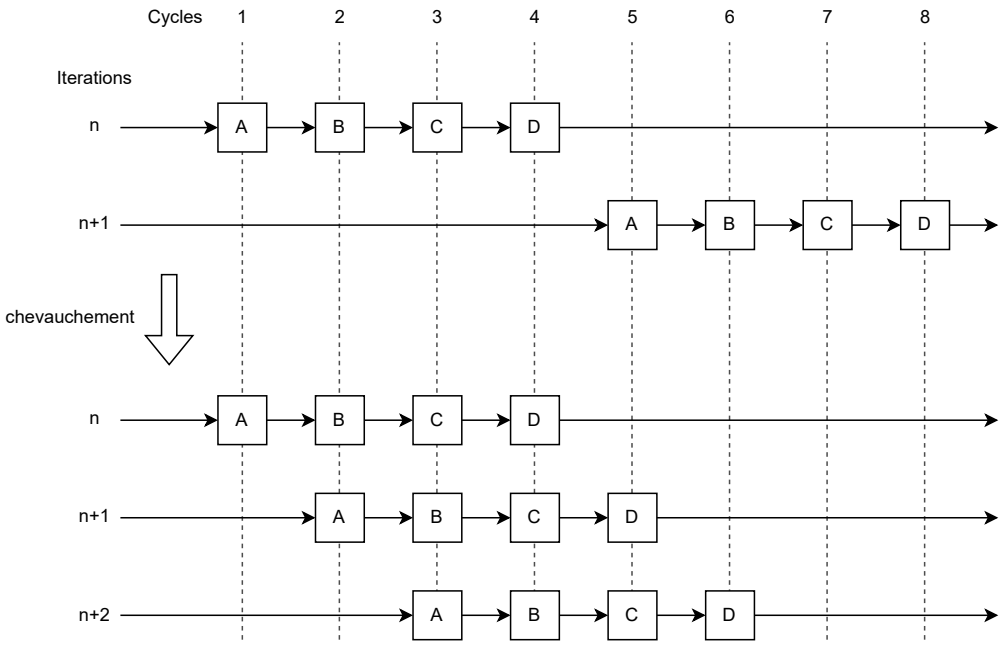

FIGURE 2 – Illustration du chevauchement des instruction pour augmenter le débit d'instructions

Une des conditions importantes pour effectuer ce mécanisme, omise dans l'exemple précédent, est que les différentes itérations doivent être indépendantes. C'est à dire qu'il ne faut pas qu'une instruction nécessite le résultats d'une instruction de l'itération précédente. Dans l'exemple de la figure 4.1.1, on constate que l'instruction A de l'itération $n+1$ nécessite le résultat de l'instruction C de l'itération n. Il est alors obligatoire pour l'exécution d'attendre la production de ce résultat. En conséquence, il est impossible de débuter une nouvelle itération tout les cycle, mais seulement tout les trois cycles. On peut donc modifier l'algorithme pour obtenir les résultats nécessaire le plus tôt possible, voire en amont de la boucle pour limiter au maximum les dépendances entres les itérations de la boucle. Ces dépendances prendront le plus souvent la forme d'un accès en

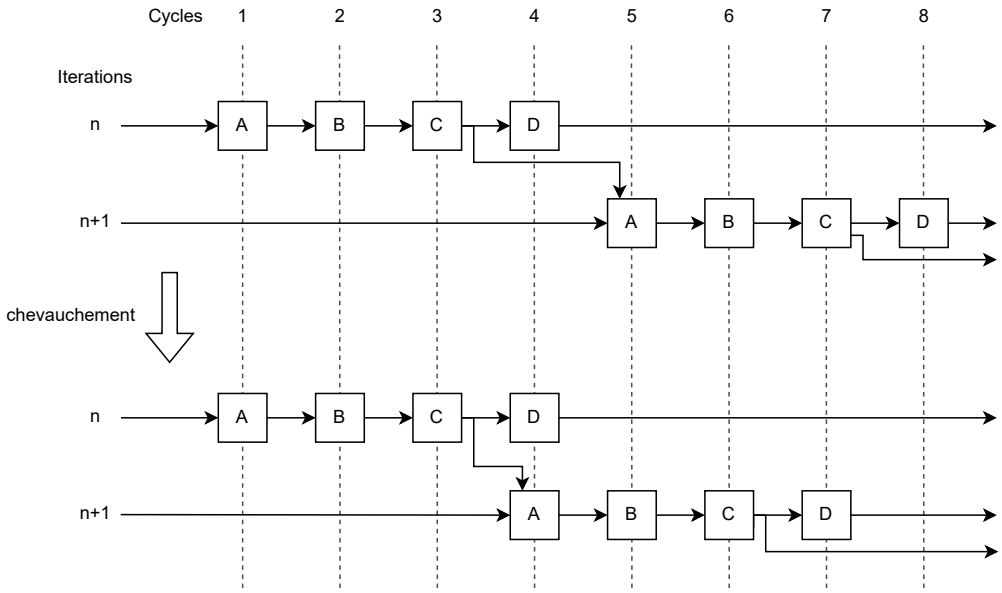

FIGURE 3 – Illustration du chevauchement non idéal lors que les séquences d'instructions ne sont pas indépendantes.

écriture à une zone mémoire qui doit être lue par les itérations suivantes. Le compilateur doit alors conserver l'ordre des accès pour garantir la validité de l'algorithme, il doit alors retarder le début de l'itération suivante. Le nombre de cycles entre deux itérations consécutives est appelé l'Intervalle d'Initialisation (II), c'est cette grandeur qui permet de rendre compte de l'efficacité du chevauchement et qui nous est fournie par Vitis HLS.

Outre les dépendances de résultats précédents, la mémoire peut être aussi un aspect limitant le chevauchement des instructions. En effet, si les itérations on besoin d'accéder à une même zone mémoire en lecture, augmenter le débit d'instructions peut introduire des situation où les cellules de mémoire ne disposent pas d'assez de ports d'accès pour satisfaire toutes les requêtes. Le programme doit alors retarder un accès dans l'attente de disponibilité d'un port de lecture, augmentant l'Intervalle d'Initialisation. Pour résoudre ce problème, on utilisera les méthode de partitionnement de la mémoire développées dans la partie 4.1.3, ou alors on cherchera à réécrire l'algorithme pour agencer différemment les accès mémoire. Il est important toutefois de noter que certains algorithmes ne peuvent naturellement pas d'optimiser avec un Intervalle d'Initialisation de 1.

Cette optimisation s'utilise dans le code source de l'algorithme comme montrée dans le pseudo code 2 en insérant la directive *pragma HLS pileline* en première ligne de la boucle concernée en spécifiant le nombre de cycle souhaité pour l'Intervalle d'Initialisation. Si le compilateur ne peut pas résoudre les conflits présentés plus haut, il visera l'Intervalle d'Initialisation le plus court et l'indiquera dans le rapport de compilation.

```
1       for (int index = 0; index < length; index++) {
2           #pragma HLS pipeline II=2
3
4           // corps de la boucle
5       }
```

Pseudo-code 2 – Chevauchement des itérations boucle en contexte

Le chevauchement des instructions est donc très efficace pour réduire la latence de l'algorithme sans augmenter la consommation de ressources. Cependant, couplé au déroulage de

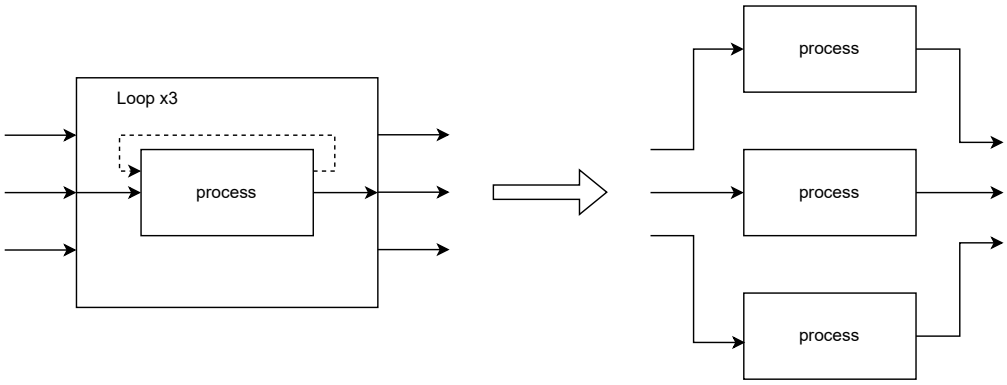

FIGURE 4 – Illustration du déroulage de boucle

boucles présenté en partie 4.1.2, il peut poser certaines contraintes qui augmentent l'allocation de ressources de manière importante.

4.1.2 Déroulage des boucles

L'autre mécanisme d'optimisation principal à disposition lors de la programmation d'un FPGA à l'aide de VITIS HLS est le déroulage des boucles. Sur un FPGA, une boucle est implémentée comme une zone du tissu logique allouée et traversée autant de fois que la boucle le demande, par les différents signaux que l'ont doit traiter. Le déroulage, copie le corps de la boucle autant de fois que le nombre de tour de boucle (ou un nombre spécifié) pour exécuter toutes les itérations en parallèle. Cette copie est une copie physique, on aura donc une forte réduction de la latence au prix d'une augmentation de la consommation des ressources.

Comme pour le chevauchement des instructions, le déroulage d'une boucle nécessite que les itérations soient indépendantes. Le fait qu'une itération nécessite un résultat calculé dans une autre itération invalide cette condition et rend le déroulage impossible. Il est donc important de concevoir l'algorithme pour que toutes les itérations soient indépendantes. Également, le nombre limité de ports d'accès aux blocs de mémoire vient aussi empêcher le déroulage efficace des boucles, on utilisera aussi le partitionnement des données pour faciliter ces accès (voir partie 4.1.3)

Dans le cas de boucles imbriquées, la consommation de ressources provoquées par le déroulage des deux boucles peut être trop important, il est donc judicieux de dérouler seulement la boucle intérieure. Si la boucle intérieure peut se dérouler complètement, il devient possible d'utiliser le chevauchement d'instruction sur la boucle extérieure. Il est important de noter que ces deux mécanismes s'appliquent sur deux boucles différentes, le compilateur refusera de se faire chevaucher des itérations de boucle partiellement déroulées.

Le déroulage de boucle s'utilise comme montré par le pseudo-code 3. La directive *pragma HLS unroll* est ajoutée à la première ligne de la boucle, et le programmeur spécifie le facteur entier de déroulement. Le facteur n'a pas besoin d'être un diviseur du nombre total de boucle. Si ce facteur n'est pas indiqué, la boucle sera déroulée complètement.

Il existe des situation ou Vitis HLS détecte de fausses dépendances entre les itérations. Il est alors possible de spécifier manuellement pour une variable que ses dépendances ne s'étendent qu'à l'intérieur d'une même itération. Pour cela on utilisera la directive *pragma HLS dependance* en spécifiant les variables concernées. Cette directive peut aussi être utilisée pour améliorer le chevauchement d'instruction.

```
1       for (int index = 0; index < length; index++) {
2           #pragma HLS unroll factor=4
3           //corps de la boucle
4       }
```

Pseudo-code 3 – Déroulage de boucle en contexte

4.1.3 Partitionnement des données

Une des difficultés pour le recouvrement d'instruction et le déroulage de boucle se trouve dans les accès mémoire sur le FPGA. Quand le programmeur alloue un tableau dans son programme, elle sera traduite sur le FPGA par l'allocation d'un bloc de mémoire sur le tissu logique. Ces blocs possèdent un nombre limité de ports d'accès en écriture et lecture (voir partie 2.2)et trop d'accès simultanée à ces blocs forceront le compilateur à réorganiser la RTL pour obtenir un programme valide, augmentant donc le nombre de cycle nécessaire. La solution proposée par la HLS est de partitionner ces données, c'est-à-dire de répartir le tableau concerné dans plusieurs blocs mémoire indépendant, chacun avec leur ports d'accès. Ceci permet d'avoir plus d'accès possible depuis deux branches d'une loupe se chevauchant, diminuant l'Intervalle d'Initialisation.

Le partitionnement des tableaux s'utilise avec la directive *pragma HLS array_partition* présentée dans le pseudo-code 4

```
1       float tableau1[100];
2       #pragma HLS array_partition variable=tableau1 type=block factor=4
3
4       int tableau2[100];
5       #pragma HLS array_partition variable=tableau2 type=cyclic factor=4
6
7       uint32_t tableau3[100];
8       #pragma HLS array_partition variable=tableau3 type=complete
```

Pseudo-code 4 – Syntaxte du partitionnement des tableaux de données

On a à notre disposition plusieurs schéma pour partitionner (block, cyclic, complete) illustrés en figure 5. *block* sépare le tableau en blocs contigus, *cyclic* sépare les élément adjacents et les entremêle dans les blocs mémoire, *complete* divise complètement le tableau, donnant un bloc mémoire à chaque valeur du tableau.

Quand les tableaux sont de petites taille (de l'ordre de la dizaine d'éléments), on pourra privilégier le partitionnement complet, cela facilitera grandement le travail du compilateur. Pour de plus grand tableaux, le choix du partitionnement dépendra de la manière dont l'algorithme accède aux données, si deux itérations consécutives d'une boucle accède à deux éléments adjacents du tableau, le partitionnement cyclique permet que ces deux éléments soient dans deux modules de mémoire différents, et donc avec des ports d'accès lecture différents. Si en revanche les itérations de boucle accèdent à des zone plus étendues du tableau, le partitionnement en blocs aura plus de sens puisque deux éléments adjacent seront utilisés par la même itération de boucle.

4.1.4 Choix du type de mémoire

Lors de la compilation, Vitis HLS va déterminer la meilleure manière d'organiser les données en mémoire, c'est-à-dire quel type de module mémoire utiliser dans le RTL. Dans la majorité des cas, Vitis HLS choisira d'utiliser des modules LUT pour leur versatilité, seulement ces modules sont très utilisés la RTL et représentent souvent la ressources limitante pour l'implémentation. On souhaitera alors spécifier manuellement quel type de mémoire utiliser dans certains cas, pour par exemple privilégier les modules BRAM, permettant de contenir beaucoup de données et très rapides d'accès. On spécifie le type de mémoire avec la directive *pragma HLS bin_storage* en indiquant à sur variable elle doit s'appliquer. Le pseudo-code 5 montre un exemple en contexte et présente les argument à fournir.

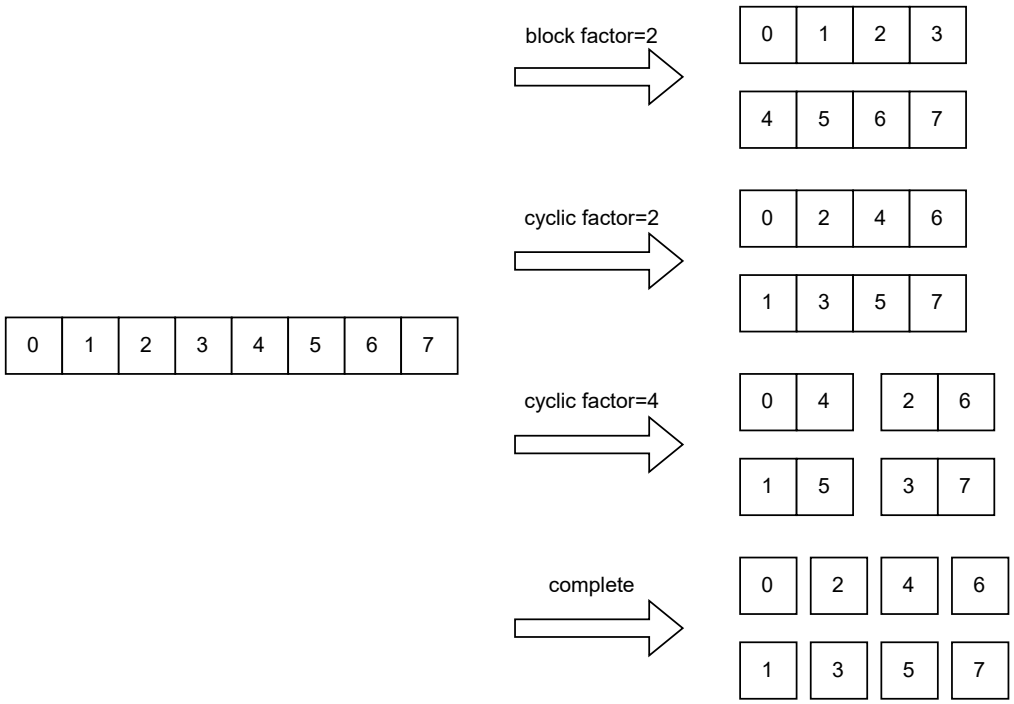

FIGURE 5 – Partitionnement des tableaux

```
1    float tableau[100];
2    #pragma HLS bind_storage variable=tableau type=RAM_2P impl=BRAM
```

Pseudo-code 5 – Syntaxte du choix du type de mémoire pour les variables

- *type* : indique la nature de la mémoire (RAM : Random Access Memory ou ROM : Read Only Memory) et le nombre de ports d'écriture/lecture. La variété de choix dépendra des modules présents sur le FPGA ciblé et certains type peuvent être indisponibles.
- *impl* : indique la méthode d'implémentation dans la RTL. C'est ici que l'on spécifie si la mémoire doit être implémentée dans un module LUT, un module BRAM ou un autre type de mémoire si le modèle de FPGA ciblé en propose d'avantage.

4.2 Egalisation paramétrique

4.2.1 Présentation et utilisation

L'égalisation est un des principaux outils de manipulation sonore en production audio, au delà de la production musicale. Un égaliseur et une suite de filtres réglables altérant le contenu fréquentiel du signal entrant et peut se présenter sous deux formes principales. Si les filtres sont à fréquence fixe et l'utilisateur peux modifier leur gain respectif on parle alors d'égalisation graphique, le contenu spectral du signal est essentiellement découpé en bandes (entre dix et trente dans la majorité des cas)et l'utilisateur altère leur amplitude. Si les fréquences des filtres ne sont pas fixées on parle alors d'égalisation paramétrique, l'utilisateur peut contrôler les fréquences, les amplitudes, le facteur de qualité et parfois le type des filtres. Les filtres d'égaliseur paramétrique sont moins nombreux (moins de dix) mais permette de choisir avec précision quelle partie du spectre altérer.

Dans cette partie nous nous concentrerons sur l'implémentation d'un égaliseur paramétrique, étant plus répandu que la variante graphique. Nous utiliserons pour cela des filtres numérique de la topologie biquadratique en forme directe II, présentée en figure 6 [6]. Ce sont des filtres

6. https://en.wikipedia.org/wiki/Digital_biquad_filter

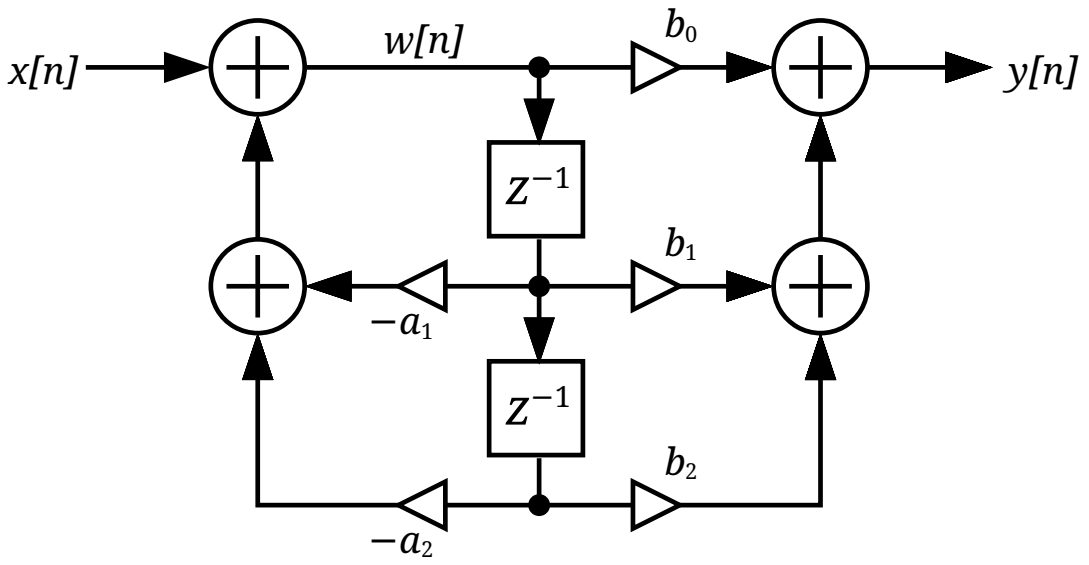

FIGURE 6 – Figure du biquad filter

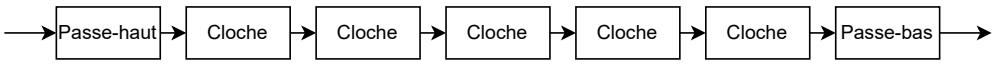

FIGURE 7 – Structure de l'égaliseur paramétrique pour sept filtres en série.

très utilisés dans le filtrage numérique pour leur faible coût computationnel et leur versatilité. Il est possible de programmer ces filtres d'ordre II pour huit réponses en fréquences à l'aide de formules simples [7], nous utiliserons ici les réponses *Passe-bas*, *Passe-haut* et *Cloche*. La formule d'application du filtre pour la forme directe II est exposée en équation 1 et est déduite de la figure 6. Cette forme directe II a été choisie contre la forme directe I ou les formes transposées car elle demande de stocker moins de variables d'état et requiert moins de calculs. De plus, cette forme peut être sujette à des instabilité numériques quand le facteur de qualité atteint des valeurs extrêmes, mais dans le cas d'un simple égaliseur, ce sont des valeurs qui ne sont jamais utilisées.

$$y[n] = b_0 w[n] + b_1 w[n-1] + b_2 w[n-2]\ ,\quad w[n] = x[n] - a_1 w[n-1] - a_2 w[n-2] \tag{1}$$

4.2.2 Implémentation et optimisation

présenter le schéma d'utilisation, avec l'ARM et OSC depuis pure data

L'implémentation avec Vitis HLS de l'égalisation paramétrique avait pour but de fournir un exemple de programme interactif contrôlé à l'aide du protocole OSC et de déterminer combien de filtres biquadratic disposés en série peuvent être compilé sur la carte Zybo Z20 (voir partie 2.2). Il a donc été choisi d'implémenter soixante filtres en série pour constater l'efficacité des optimisations. Le lien avec OSC, comme montré en figure 1, est effectué en travaillant avec le micro-processeur ARM présent sur le SoC. On se servira également de ce processeur pour effectuer les calculs coûteux des coefficients, initialiser les filtres et les transmettre au FPGA via la mémoire partagée.

Le programme complet fonctionne ainsi :
— Le programme sur le processeur ARM initialise les filtres, le FPGA et ouvre le serveur OSC.
— Sur un ordinateur tier, un programme créé avec Pure Data [8] se connecte au serveur OSC et sert d'interface utilisateur pour contrôler le comportement de l'égaliseur en envoyant les paramètres des filtres dans le serveur OSC.

7. Audio EQ cookbook by Robert Bristow-Johnson : `https://www.w3.org/TR/audio-eq-cookbook/`
8. `https://puredata.info`

— Le programme ARM reçois les paramètres par le serveur OSC, calcule les coefficient et met à jour les filtres.
— Le FPGA récupères les filtres sur la mémoire partagée avec le processeur ARM, et les applique sur le signal entrant, le pseudo-code 6 présente l'algorithme de la HLS.

```
1  n_filters = 60
2  filter_array = BiquadFilter[n_filters];
3
4  for (index = 0; index < n_filters; index++) {
5      // recuperation des filtres de la memoire partagee
6      filters[index] = mem_zone_f[adresse_partagee];
7  }
8
9  signal_sample = read_from_audio_input();
10
11 for every filter {
12     // application des filtres
13     filter_process(signal_sample, filter)
14 }
15
16 write_to_output(signal_sample);
```

Pseudo-code 6 – Algorithme d'application de l'égaliseur paramétrique

Les résultats des optimisations pour soixante filtres effectuées sont visibles sur la figure 8. La partie de gauche affiche la consommation de ressources tandis que la partie de droite présente la latence de l'algorithme, les grandeurs sont données en pourcentages. Dans la première situation, aucune optimisation n'est appliquée et la latence est supérieure à 100% : le programme ne peut être compilé. En appliquant le déroulage de boucle et le recouvrement, la latence diminue à 82% (l'algorithme ne permet pas de beaucoup paralléliser), le RTL peut donc être synthétisé correctement. Enfin, en troisième situation, on applique le partitionnement complet du tableau contenant les filtres pour augmenter la quantité d'accès mémoire. On constate une forte augmentation de la consommation de LUT et BRAM due au nombre important de modules de mémoire alloués, mais la latence baisse encore à 75%. Il est important de rappeler que cet égaliseur est composé de soixante filtres ce qui, même pour un égaliseur graphique est plus que nécessaire. On voit donc que ce programme pourrait aisément être inclus comme partie dans un autre programme plus complexe.

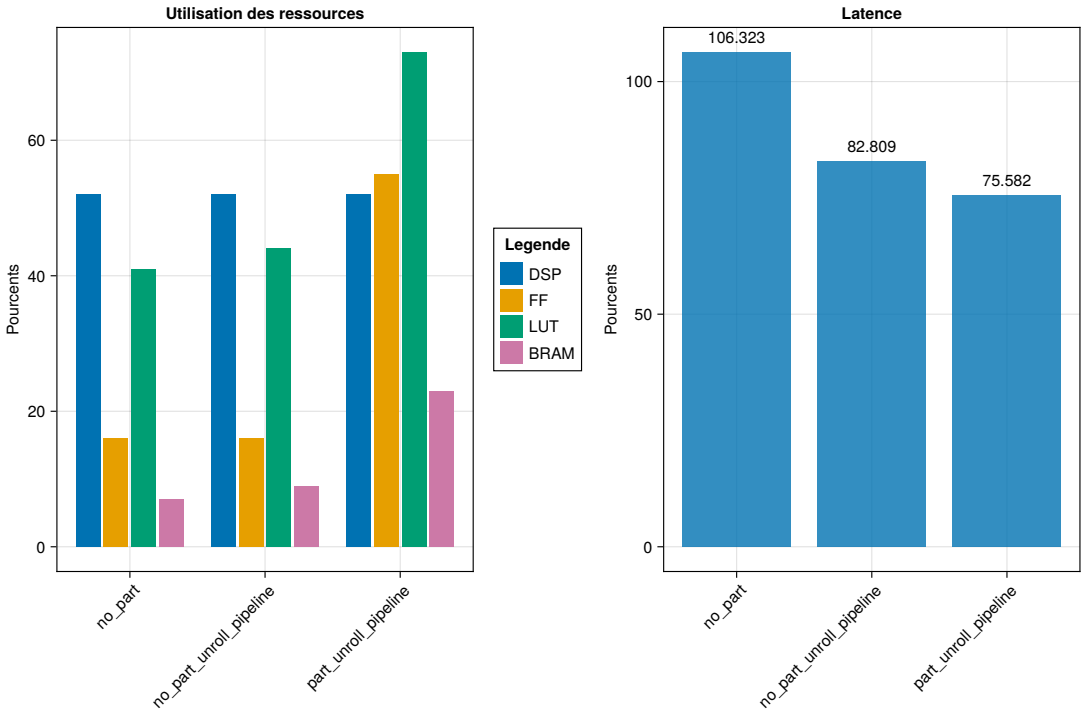

FIGURE 8 – Evolution des performance de l'égaliseur paramétrique. La partie de gauche affiche la consommation de ressources tandis que la partie de droite présente la latence de l'algorithme, les grandeurs sont données en pourcentages. Dans la première situation, aucune optimisation n'est appliquée et la latence est supérieure à 100% : le programme ne peut être compilé. En appliquant le déroulage de boucle et le recouvrement, la latence diminue à 82% (l'algorithme ne permet pas de beaucoup paralléliser), le RTL peut donc être synthétisé correctement. Enfin, en troisième situation, on applique le partitionnement complet du tableau contenant les filtres pour augmenter la quantité d'accès mémoire. On constate une forte augmentation de la consommation de LUT et BRAM due au nombre important de modules de mémoire alloués, mais la latence baisse encore à 75%.

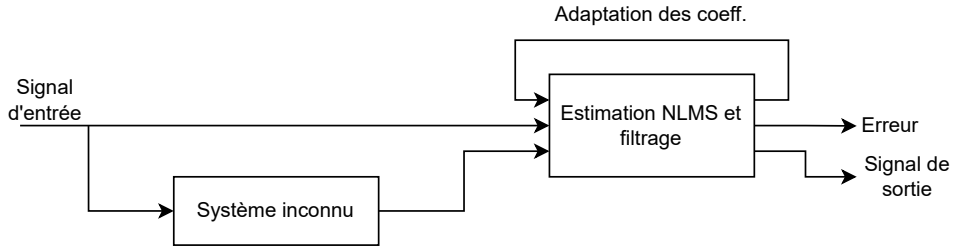

FIGURE 9 – Diagramme du contexte d'utilisation de la NLMS

4.3 Algorithme de NLMS (Normalised Least Mean Square)

4.3.1 Présentation de l'algorithme

Cette sous partie présente la théorie et l'implémentation d'un algorithme de filtre adaptatif utilisant une méthode des moindres carrés appelée la LMS (Least Mean Square). L'algorithme des filtres adaptatifs par LMS est décrite dans [3], il consiste à adapter un filtre à Réponse Impulsionnelle Finie (RIF) en fonction du résultats de sa convolution au signal d'entrée. Cette méthode est utilisée pour identifier le comportement d'un système linéaire inconnu, comme représenté en figure 9.

L'algorithme se déroule comme suit :

— On prend en entrée un échantillon du signal original, ainsi qu'un du signal altéré par le système inconnu à identifier.
— On filtre le signal d'entrée avec le filtre adaptatif et on compare le résultat avec le signal altéré par le système pour établir une erreur.
— On adapte ensuite les coefficients du filtre en fonction de cette erreur ainsi que de l'amplitude du signal d'origine pour la version normalisée. L'étape d'adaptation est présentée dans l'équation 2 où $\mathbf{h}$ est le vecteur des coefficients du filtre, e est l'erreur, $\mathbf{x}$ est le vecteur des échantillons du signal d'origine et μ le pas d'apprentissage.

$$\hat{\mathbf{h}}(n+1) = \hat{\mathbf{h}}(n) + \frac{\mu e^*(n)\mathbf{x}(n)}{\mathbf{x}^H(n)\mathbf{x}(n)} \tag{2}$$

Le dénominateur de l'équation 2 représente la normalisation par rapport à l'amplitude du signal d'origine, sans cet ajout, il est plus difficile de choisit un μ qui garantisse la stabilité de l'algorithme, surtout pour un signal d'entrée avec une grande dynamique d'amplitude.

4.3.2 Implémentation et optimisation

Le pseudo-code 7 présente le pseudo code pour l'algorithme du filtrage adaptatif NLMS.

```
1  filter_coefficients[FILTER_ORDER];
2  input_buffer[FILTER_ORDER];
3
4  system_input_sample  = read_from_audio_input();
5  system_output_sample = read_from_audio_input();
6
7  // suppression de l'echantillon le plus ancien
8  buffer_squared_norm -= input_buffer[end] * input_buffer[end];
9
10 // Decallage du vecteur du signal
11 for index in [FILTER_ORDER - 1 ... 0[
12     input_buffer[index] = input_buffer[index - 1];
```

```
13 }
14
15 // Accueil du nouvel echantillon
16 input_buffer[0] = system_input_sample;
17 buffer_squared_norm += input_buffer[0] * input_buffer[0];
18
19 // convolution du filtre par le signal en memoire
20 // (produit scalaire)
21 for index in [0...FILTER_ORDER[ {
22     //boucle deroulee
23     estimated_output += filter_coefficients[index]
24                             * input_buffer[index];
25 }
26
27 error = system_output_sample - estimated_output;
28
29 // Definition d'un parametre de regularisation
30 // pour eviter une division par 0
31 regularization_parameter = 0.001f;
32
33 // mise a jour des coefficients du filtre en fonction
34 // de l'erreur et de l'amplitude du signal
35 for index in [0...FILTER_ORDER[ {
36     // unrolles & pipelined loop
37     filter_coefficients[index] += input_buffer[index]
38                                     * STEP_SIZE
39                                     * error
40                                     / (buffer_squared_norm
41                                         + regularization_parameter);
42 }
43
44 write_to_audio_output(estimated_output);
45 write_to_audio_output(error);
```

Pseudo-code 7 – Algorithme du filtrage adaptatif par NLMS

Dans cette partie nous nous intéressons à l'implémentation de cet algorithme pour un filtre d'ordre 2048 Le principal point de travail sur cet algorithme se trouve dans le stockage du signal d'entrée pour calculer son amplitude. Ce stockage sera implémenté comme une mémoire à décalage : à chaque nouvel échantillon, on sort le plus ancien du conteneur, on décale tout le contenu dans la mémoire et on accepte le nouvel échantillon. Avec cette méthode, on a toujours accès au 2048 derniers échantillons du signal et donc on peut calculer son amplitude à chaque instant de temps. Cette amplitude sera calculée comme la somme des carrés des échantillons Cependant, on ne calculera pas l'amplitude en parcourant le conteneur et sommant le carré de chaque valeur à chaque appel de la fonction, cela coûte d'implémenter une boucle de calcul pour faire des calculs redondants. En effet, on peut économiser des calculs en conservant la somme d'une itération sur l'autre et, au début d'une nouvelle itération soustraire le carré de l'échantillon le plus ancien et ajouter le carré du nouvel échantillon. On met ainsi à jour l'amplitude du signal en seulement deux multiplications, une soustraction et une addition. On conservera toujours les 2048 échantillon puisque nous en avons besoin pour réaliser le filtrage cependant.

Le choix de la mémoire tampon comme un registre à décalage et non comme un tampon circulaire est liée au fonctionnement du FPGA. Avec un registre à décalage, on peut partitionner finement la mémoire et accéder rapidement à toutes les position pour les mettre à jour. Le tampon circulaire, en revanche nécessite de conserver un indice de position incrémenté à chaque itération et replié au début de la mémoire quand il en dépasse la longueur. Certes l'on économise une boucle pour le décalage des échantillons, mais pour l'application et la mis à jour des coefficients du filtre, il sera nécessaire de manier un indice à incrémenter de plus et de rajouter un test pour le repliement de l'indice. Ces rajout rendent difficile le chevauchement d'instructions

puisqu'il peuvent générer un branchement dans la séquence d'instruction, et donc le nombre de cycles n'est pas connu avec précision. En utilisant cette approche et en ajoutant le chevauchement d'instructions, Vitis HLS n'arrive plus à générer le RTL.

Le partitionnement du filtre et de l'historique du signal permet la parallélisation efficace des boucles d'application du filtre et de mise à jour. La convolution du signal par le filtre, pour un échantillon, est essentiellement le produit scalaire entre les deux vecteur ce qui est hautement parallélisable. Également, la mise à jour des coefficients se fait dans une boucle qui les multiplie respectivement avec les échantillons du signal de manière indépendante, également hautement parallélisable. La figure 10 expose les résultats des optimisations en fonctions du facteur de déroulement et de partitionnement pour quatre situations. Dans la première situation, aucune optimisation n'est réalisée, on observe une très faible consommation de ressources mais une latence de 1204%. En augmentant le facteur de déroulage des boucles d'application du filtre et de mise à jour des coefficients la latence chute brusquement contre une augmentation des ressources. Pour un facteur de 64, la latence continue de descendre mais reste au dessus de 100% tandis que la consommation des modules LUT, utilisée pour implémenter des opérations logiques et stocker les variables dépassent également 100%. Le déroulage de boucle force Vitis HLS à réaliser des copies du RTL des boucles, augmentant grandement les besoin en modules LUT. En appliquant un partitionnement cyclique de facteur 32 (voir partie 4.1.3)sur les deux vecteurs de taille 2048, et en forçant leur stockage dans les modules BRAM (dont on constate l'augmentation de leur utilisation), la consommation des modules LUT est encore réduite et l'on peut même augmenter le facteur de déroulage à 128 avec une latence de 95%, le programme est donc valide.

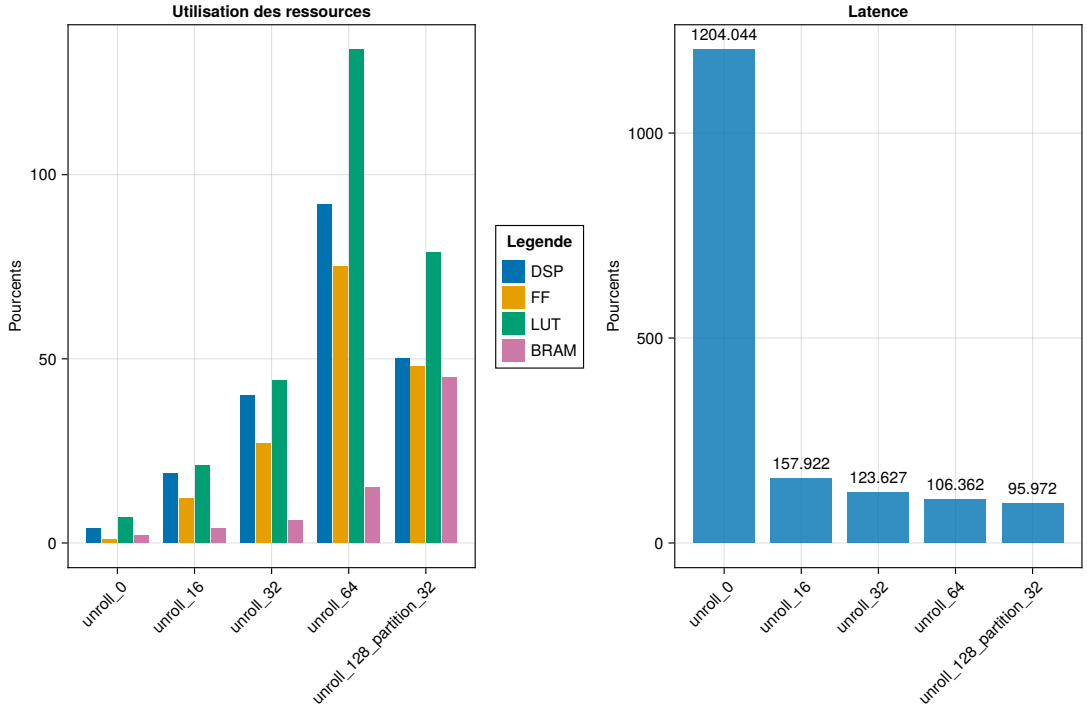

FIGURE 10 – Evolution des performances pour l'algorithme de la NLMS pour quatre configurations de déroulage et de partitionnement. Dans la première situation, aucune optimisation n'est réalisée, on observe une très faible consommation de ressources mais une latence de 1204%. En augmentant le facteur de déroulage des boucles d'application du filtre et de mise à jour des coefficients la latence chute brusquement contre une augmentation des ressources. Pour un facteur de 64, la latence continue de descendre mais reste au dessus de 100% tandis que la consommation des modules LUT, utilisée pour implémenter des opérations logiques et stocker les variables dépassent également 100%. Le déroulage de boucle force Vitis HLS à réaliser des copies du RTL des boucles, augmentant grandement les besoin en modules LUT. En appliquant un partitionnement cyclique de facteur 32 (voir partie 4.1.3)sur les deux vecteurs de taille 2048, et en forçant leur stockage dans les modules BRAM (dont on constate l'augmentation de leur utilisation), la consommation des modules LUT est encore réduite et l'on peut même augmenter le facteur de déroulage à 128 avec une latence de 95%, le programme est donc valide.

4.4 Modélisation physique

4.4.1 Présentation de l'algorithme

Ce dernier algorithme, vise à implémenter une modélisation physique d'une réverbération à plaque L'implémentation se base sur les précédents travaux de S. Bilbao et C. Webb [1, 12] ou la simulation est abordée selon deux approches : l'approche modale et l'approche par différences finies. Nous nous concentrons ici sur l'approche pas différences finies en domaine temporel décrite dans [12]. La méthode consiste en la création d'un maillage de la plaque à simuler et le calcul de constantes dépendant des paramètres d'entrée ainsi que des caractéristiques du matériau simulé. Ces constantes forment ainsi un masque que l'on convolue au maillage à chaque itération de l'algorithme pour obtenir un échantillon audio. Le but de ce travail est donc de mesurer les effets des directives d'optimisation sur la consommation de ressources et la latence pour déterminer si le programme peut être compiler pour les FPGA dont l'équipe Emeraude dispose, et si oui quel est la consommation des ressources, ou si il faut nécessairement un FPGA plus puissant. La théorie derrière cet algorithme se trouve dans [1] et ses implémentations sur CPU, CPU avec instruction vectorielles et GPU sont décrites dans [12]. Nous adapterons ici la version CPU et l'optimiserons pour notre matériel cible.

La formule de comportement d'une plaque décrite par Bilbao est donnée en équation 3 ou u est le vecteur d'état, κ, σ_0 et σ_1 sont des constantes de la simulation déterminant le temps de réverbération et $\sum_q E_q F_q$ représente l'excitation extérieure.

$$u_{tt} = -\kappa^2 \Delta\Delta u - 2\sigma_0 u_t + 2\sigma_1 \Delta u_t + \sum_q E_q F_q \tag{3}$$

Traduite en domaine discret l'équation devient :

$$\delta_{tt} u = -\kappa^2 \delta_\Delta \delta_\Delta u - 2\sigma_0 \delta_t u + 2\sigma_1 \delta_{t-} \delta_\Delta u + \sum_q E_q F_q \tag{4}$$

où δ_t, δ_{tt} et δ_{t-} sont les opérateurs discrets de différence et δ_Δ est une approximation du Laplacien pour le calcul discret.

Pour faciliter l'implémentation de cette relation de récurrence, Bilbao réécrit cette équation sous forme matricielle :

$$\mathbf{u}_{n+1} = \mathbf{B}\mathbf{u}^n + \mathbf{C}\mathbf{u}^{n-1} \tag{5}$$

$$\mathbf{B} = \frac{1}{1+\sigma_0 k}(2\mathbf{I} - \kappa^2 k^2 \mathbf{D}^{(4)} + 2\sigma_1 k \mathbf{D}^{(2)})$$
$$\mathbf{C} = -\frac{1}{1+\sigma_0 k}((1-\sigma_0 k)I + 2\sigma_1 k \mathbf{D}^{(2)})$$

avec k le pas de temps, et les matrices $\mathbf{D}$ implémentant les opérateur de différentiations mentionnés plus haut.

Bilbao indique les manière de calculer le contenu des matrices $\mathbf{B}$ et $\mathbf{C}$, contenant des coefficients constants pour la durée de la simulation, et ces matrices sont par ailleurs creuse, et dans notre situation nous n'avons que six coefficients à conserver. Nul besoin de les allouer sous la forme d'une matrices, nous pouvons les stocker comme des variables séparées.

L'application de l'équation 5 prends la forme d'un masque à deux dimensions à convoluer sur une matrice représentant le maillage discret de la plaque à simuler. Ce masque est montré en figure 11. La figure montre les deux couches séparées du masque puisqu'elle s'applique respectivement sur l'instant n et $n-1$ pour générer une valeur qui sera stocké au centre du masque pour l'instant $n+1$. Le calcul de cette valeur s'effectue par produits scalaires des deux sous-masques avec leur matrice d'état respective.

Une fois les masques convolués complètement avec les matrices d'état et la nouvelle matrice d'état de l'instant $n+1$ calculée, l'échantillon de sortie de la simulation se fait à un indice

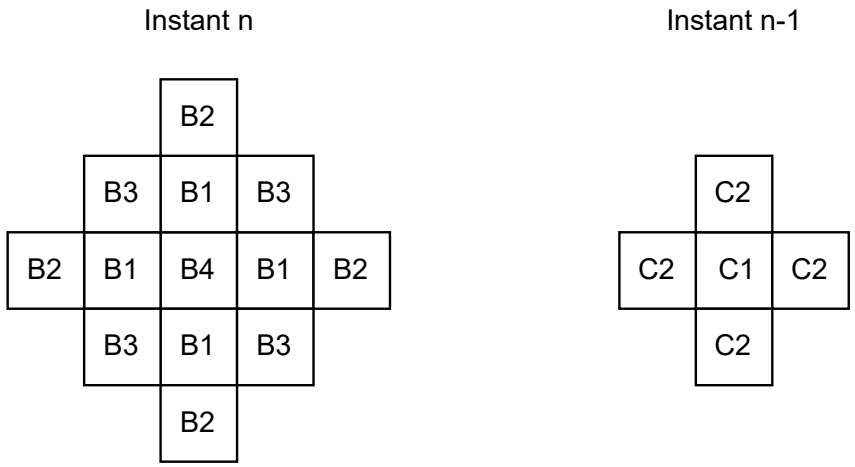

FIGURE 11 – Masques à scanner sur les matrices d'état. La figure montre les deux couches séparées du masque puisqu'elle s'applique respectivement sur l'instant n et $n-1$ pour générer une valeur qui sera stocké au centre du masque pour l'instant $n+1$. Le calcul de cette valeur s'effectue par produits scalaires des deux sous-masques avec leur matrice d'état respective.

arbitraire de la plaque, similaire à un transducteur piezo-électrique sur les vrais dispositifs de réverbération à plaque. Également, lors de la première itération, une impulsion de Dirac sera injectée dans la plaque à un indice arbitraire pour exciter le système. On choisis une impulsion de Dirac pour sa simplicité dans le cadre d'une implémentation sur FPGA, dans la réalité l'impulsion serait plus de la forme d'une Gaussienne ou d'une période de cosinusoïde.

Le pseudo-code du calcul de la nouvelle matrice-d'état est donné en 8, il reprend le code utilisé par Webb dans [12] où u est la matrice d'état réduite sous forme de vecteur de l'instant $n+1$, u1 celle de l'instant n et u2 celle de l'instant $n-1$.

```
1  for(int sampl_index = 0; sample_index < num_samples; ++sample_index) {
2
3      for(int row_index = 2; row_index < (num_rows-1); ++row_index) {
4          for(int col_index = 2; col_index < (num_rows-1); ++col_index) {
5
6              index = row_index*grid_width+col_index;
7
8              u[index] =
9                  B1 * (u1[index-1]
10                     + u1[index+1]
11                     + u1[index-width]
12                     + u1[index+width])
13                 + B2 * (u1[index-2]
14                     + u1[index+2]
15                     + u1[index-2*width]
16                     + u1[index+2*width])
17                 + B3 * (u1[index+width-1]
18                     + u1[index+width+1]
19                     + u1[index-width-1]
20                     + u1[index-width+1])
21                 + B4 * u1[index]
22                 + C1 * (u2[index-1]
23                     + u2[index+1]
24                     + u2[index-width]
```

```
                    + u2[index+width]);
                + C2 * u2[index]
        }
    }

    // Injection de l'impulsion de Dirac
    if (sample_index == 1) {
        u[impulse_indedx] += 1.0;
    }

    // Lecture de la sortie
    out[sample_index] = u[output_index];

    // echange de pointeurs
    ptr = u2;
    u2  = u1;
    u1  = u;
    u   = ptr;
}
```

Pseudo-code 8 – Algorithme de mise à jour des matrices d'état pour la simulation de réverbération à plaque

4.4.2 Implémentation et optimisation

Le premier obstacle de cette implémentation avec Vitis HLS est la taille des vecteur à stocker. En effet, pour une précisions suffisante, il est nécessaire de pouvoir manipuler trois vecteurs (pour les trois instants temporels) comportant chacun plusieurs milliers d'éléments. Ceci ne peux pas se faire efficacement sur le FPGA et ils doivent donc être initialisé par le processeur ARM de la carte puis rangés sur la mémoire partagée. Ensuite, si l'on veut suivre l'implémentation CPU de Webb [12], il faut allouer trois vecteurs d'état et recréer une sorte de registre à décalage dont les éléments sont des pointeurs vers ces vecteurs. Pour calculer un nouvel état du système, il suffit alors de considérer deux des vecteurs comme les instant précédents et d'écrire dans le troisième, et au début de chaque itération, on permute les pointeurs pour réutiliser le vecteur le plus ancien et ainsi éviter des allocation inutiles. Or ceci n'est pas possible avec Vitis HLS, qui ne permet pas d'utiliser des pointeurs de pointeurs où simplement de prendre un pointeurs vers une adresse arbitraire, le RTL n'implémente pas le concept d'adresse pour les modules présent sur le tissu logique et les tailles des objets doivent être connues à la compilation. La solution trouvée à ce problème est d'allouer les trois vecteur sur la mémoire partagée comme une seule grande portion de mémoire, et de former les trois vecteurs d'état comme des décalages du pointeur principal, ce que Vitis HLS autorise. On peut ainsi permuter les décalages dans ce grand tableau pour permuter les vecteurs et éviter les même allocations inutiles, ce que Vitis HLS n'aurait pas permis, n'autorisant pas les allocations dynamiques de mémoire. Cette approche est montrée dans le pseudo-code 9.

```
int offset_0;
int offset_1;
int offset_2;

switch (permutation_index) {
    case 0:
        offset_0 = 0;
        offset_1 = state_vector_size;
        offset_2 = state_vector_size * 2;

    case 1:
        offset_0 = state_vector_size * 2;
        offset_1 = 0;
        offset_2 = state_vector_size;
```

```
15
16      case 2:
17          offset_0 = state_vector_size;
18          offset_1 = state_vector_size * 2;
19          offset_2 = 0;
20  }
21
22  u  = mem_zone_f + offset_0;
23  u1 = mem_zone_f + offset_1;
24  u2 = mem_zone_f + offset_2;
25
26  permutation_index++;
27  if (permutation_index == 3) { permutation_index = 0; }
```

Pseudo-code 9 – Gestion des décalages de pointeurs pour la permutation des vecteurs d'état

Les vecteur d'état sont donc conservés dans la mémoire partagée, qui est beaucoup plus lente d'accès que les modules de mémoire du tissu logique du FPGA, il faudra donc, avant de réaliser les calculs, de rapatrier les données des vecteurs dans les modules mémoire, contredisant le fait de les stocker dans la mémoire partagée en premier lieu. On cherchera donc à seulement rapatrier une sous-partie de la matrice totale, calculer le nouvel état pour cette sous-matrice, et réitérer jusqu'à avoir parcouru toute la nouvelle matrice d'état. Cette approche que l'on appellera ici la version *partitionnée* de l'algorithme nous permet également de réduire le coût en calcul de l'algorithme, la convolution ayant une complexité quadratique, partitionner une grande convolution en plusieurs petites permet de réduire la quantité totale de calculs. On introduit alors un nouveau paramètre : la taille de fenêtre. C'est le paramètre qui détermine la taille de la zone à rapatrier pour effectuer les convolution partielles, on peut aussi en contrôler le nombre et leur répartition, dictant les dimensions de la matrice globale. On a donc une quantité restreinte de valeurs possibles pour les dimensions de la matrice d'état, car elles sont dirigées par les dimensions et le nombre de fenêtres.

Les évolutions des performances sont visibles en figure **??** pour six configurations différentes. Tout d'abord, sans aucune optimisation et avec une grille de 100 par 84, la latence est de 13 811% pour une consommation de ressources quasi-nulle. En deuxième colonne, réduire la taille de la grille diminue grandement la latence (toujours trop élevée) sans grande différence dans l'utilisation de ressources. Trop réduire la taille de la grille altère le son produit et introduit un décalage statique dans le signal ne traduisant alors plus le comportement réel du dispositif. On implémente ensuite l'algorithme partitionné et on observe une explosion de la consommation en ressources, notamment les modules LUT qui atteignent près de 450% des modules disponibles, mais la latence baisse encore. En couplant le partitionnement avec la réduction de la grille on peut atteindre des valeurs de latence d'environ 300%, valeur minimale atteinte durant le stage. Malheureusement la demande en ressources et la latence restent encore trop élevée pour que le RTL puisse être synthétisé.

Les recherches n'ont pas mené à une version de l'algorithme qui puisse être synthétisée sur les cartes FPGA ZyboZ20 du laboratoire. De meilleurs résultats on été obtenu par l'équipe Emeraude avec la simulation par méthode modale utilisant les propriétés de forte parallélisation des FPGA pour synthétiser plusieurs quelques dizaines de milliers de modes de vibration de la plaque vibrante. Ces travaux font partie du projet Syfala.

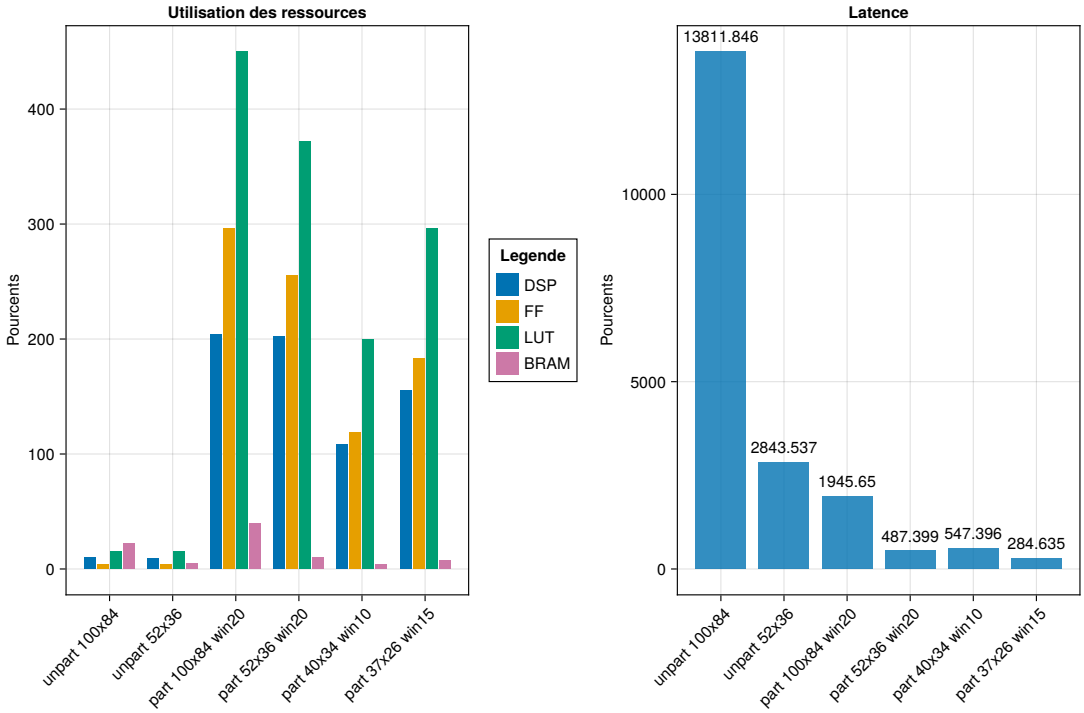

FIGURE 12 – Evolution des performances pour l'algorithme de réverbération à plaque pour six configurations différentes. Tout d'abord, sans aucune optimisation et avec une grille de 100 par 84, la latence est de 13 811% pour une consommation de ressources quasi-nulle. En deuxième colonne, réduire la taille de la grille diminue grandement la latence (toujours trop élevée) sans grande différence dans l'utilisation de ressources. Trop réduire la taille de la grille altère le son produit et introduit un décalage statique dans le signal ne traduisant alors plus le comportement réel du dispositif. On implémente ensuite l'algorithme partitionné et on observe une explosion de la consommation en ressources, notamment les modules LUT qui atteignent près de 450% des modules disponibles, mais la latence baisse encore. En couplant le partitionnement avec la réduction de la grille on peut atteindre des valeurs de latence d'environ 300%, valeur minimale atteinte durant le stage. Malheureusement la demande en ressources et la latence restent encore trop élevée pour que le RTL puisse être synthétisé.

5 Conclusion et perspectives

Dans ce mémoire, nous avons donc étudié les principaux mécanismes d'optimisation mis à disposition par Vitis HLS, l'environnement de Synthèse de Haut Niveau pour FPGA de Xilinx et les avons mis en contexte dans les implémentation de trois algorithmes de traitement audio numériques.

L'égalisation paramétrique a permis de mettre en avant les avancée de Syfala dans la connection de la carte à un logiciel compatible OSC ainsi que du grand nombre de filtre en série que le matériel peut appliquer. Le filtrage adaptatif par méthode NLMS montre également les capacité de filtrage FIR des FPGA de par la nature hautement parallélisable de ces traitements. Enfin, même si l'algorithme de simulation de réverbération à plaque par différence finie ne peut être compilé sur les cartes de développements du laboratoire, la méthode par synthèse modale a donnée de bon résultats et semble être une bonne alternative.

Il restent des mécanismes d'optimisation qui n'ont pas été étudiés par ces algorithmes. La méthode d'inversion de boucles dans le cas de traitements sur plusieurs échantillons n'était pas pertinente ici de par la nature des algorithmes. Également, les structures de données fournies par Vitis HLS ne répondaient pas au besoin des algorithmes étudiés, mais s'appliquent mieux dans le cas d'une convolution partitionnée, également étudiée au Laboratoire par Rémi Jeunehomme, en stage de master à l'INSA.

Il existes plusieurs projets très intéressants pour tirer partie des atouts du projet Syfala comme la création d'une sphère ambisonique d'ordre élevée (HOA) ou l'implémentation d'une table de mixage numérique contrôlée par ordinateur via un serveur OSC.

Références

[1] Stefan Bilbao. A modular percussion synthesis environment. In *proceedings of the 12th International Conference on Digital Audio Effects*, Graz, Austria, 2009.

[2] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Invited : Bambu : an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330, San Francisco, CA, USA, December 2021.

[3] Simon Haykin and Bernard Widrow. *Least-Mean-Square Adaptive Filters*. John Wiley and Sons, Ltd, 2003.

[4] Intel. High level synthesis compiler. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`. Outil pour la traduction de programme C++ en.

[5] Mathworks. Matlab hdl coder. `https://uk.mathworks.com/products/hdl-coder.html`. Boîte à outil pour l'écosystème Matlab pour la traduction en VHDL de scripts Matlab.

[6] Yann Orlarey, Dominique Fober, and Stéphane Letz. FAUST : an efficient functional approach to DSP programming. In Editions DELATOUR FRANCE, editor, *New computational paradigm for computer music*, pages 65–96. 2009.

[7] Maxime Popoff, Romain Michon, Tanguy Risset, Pierre Cochard, Stephane Letz, and Yann Orlarey. Audio dsp to fpga compilation : The syfala toolchain approach. In *proceedings of the 2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Porto, Portugal, 2023.

[8] Maxime Popoff, Romain Michon, Tanguy Risset, Yann Orlarey, and Stéphane Letz. Towards an FPGA-based compilation flow for ultra-low latency audio signal processing. In *proceedings of the SMC-22 - Sound and Music Computing Conference*, saint-Etienne, France, 2022.

[9] I.S. Uzun, A. Amira, and A. Bouridane. FPGA implementation of fast fourier transforme for real-time signal and image processing. *IEE Proceedings - Vision, Image and Signal Processing*, 152 :283–296, 2005.

[10] Trevor Vannoy, Tyler B Davis, Connor Dack, Dustin Sobrero, and Ross K Snider. An open audio processing platform using soc FPGAs and model-based development. In *proceedings of the 147th Convention of the Audio Engineering Society*, New York, 2019.

[11] Yonghao Wang. Low latency audio processing. Master's thesis, School of Electronic Engineering and Computer Science, Queen Mary university of London, 2017.

[12] Craig J Webb and Stefan Bilbao. On the limits of real-time physical modelling synthesis with a modular environment. In *proceedings of the 18th International Conference on Digital Audio Effects*, Trondheim, Norway, 2015.

[13] Clemens Wegener, Sebastian Stang, and Max Neupert. FPGA-accelerated real-time audio in pure data. In *proceedings of the Sound and Music Computing Conference*, Saint-Etienne, France, 2022.

[14] Xilinx. Vitis-hls. `https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html`. Logiciel distribué par AMD proposant un compilateur HLS pour FPGA depuis C++.