VILLE DE LIÈGE

Institut de Technologie
Enseignement de Promotion sociale

Année académique 2021 – 2022

Développement d'un codec audio AAC : optimisation de l'algorithme MDCT pour l'architecture ARM

Étudiante :
Laura Binacchi

Lieu de stage :
EVS Broadcast Equipment
Rue du Bois Saint-Jean 13, 4102 Ougrée

Maître de stage :
Bernard Thilmant
Software Engineer

Épreuve intégrée présentée pour l'obtention du diplôme de
BACHELIER.E EN INFORMATIQUE ET SYSTÈMES
FINALITÉ : INFORMATIQUE INDUSTRIELLE

Table des matières

Table des figures

Remerciements

Introduction

Développement d'une solution de software embarqué sur processeur ARM pour encodage audio AAC optimisé aux applications d'EVS :

- Prise de connaissance de l'encodage AAC et de l'environnement EVS qui utilise ce type de format ;
- Prise de connaissance des résultats des optimisations possibles du modèle psycho-acoustique développé par EVS ;
- Développement du code en C ou Assembler pour l'encodage AAC sur plateforme ARM ;
- Test du système et documentation de son implémentation.

Ce travail commencera par une présentation d'EVS et du département dans lequel s'est déroulé mon stage. Parmi les nombreux produits d'EVS, seul le serveur XT sera brièvement présenté puisque c'est spécifiquement pour ce dernier que le codec AAC est développé et optimisé.

Quelques notions théoriques indispensables à la compréhension du travail pratique seront ensuite développées avec une section consacrée au son et sa numérisation et une autre consacrée aux codecs MPEG, à leur fonctionnement et en particulier au fonctionnement du bloc MDCT de l'encodeur AAC.

1 EVS Broadcast Equipment

1.1 Présentation d'EVS et du département R&D

Mon stage s'est déroulé au sein de la société EVS Broadcast Equipment dont la figure 1 représente le logo. EVS est une entreprise d'origine liégeoise devenue internationale. Fondée en 1994 par Pierre L'Hoest, Laurent Minguet et Michel Counson, EVS compte aujourd'hui plus de 600 employés dans plus de 20 bureaux à travers le monde mais son siège principal se situe toujours à Liège.

Figure 1 – Logo de la société EVS Broadcast Equipment[1]

EVS est devenu leader dans le monde du broadcast avec ses serveurs permettant l'accès et la diffusion instantanée des données audiovisuelles enregistrées sur ses serveurs. L'entreprise est également célèbre pour ses ralentis instantanés. Ces technologies sont utilisées pour la production live des plus importants événements sportifs dans le monde : le matériel EVS est notamment utilisé pour la retransmission des Jeux Olympiques depuis 1998.

Plus de 50% des employés d'EVS travaillent en recherche et développement afin de répondre au marché du broadcast en constante évolution. Outre ses solutions techniques innovantes, EVS se différencie de ses concurrents par la proximité entretenue avec les clients en leur proposant des solutions à l'écoute de leurs besoins et en leur offrant un service de support de qualité.

C'est en R&D, dans l'équipe Hardware-Firmware, que s'est déroulé mon stage. Sous la direction de Justin Mannesberg, cette équipe se compose d'une vingtaine d'employés spécialisés en développement embarqué et en développement FPGA. La situation particulière dans laquelle s'est déroulé mon stage, en pleine pandémie de Covid et alors que tous les employés étaient confinés, ne m'a pas permis d'interagir avec beaucoup de membres de l'équipe et ni de pouvoir observer leur travail. Bernard Thilmant (Software Engineer dans l'équipe Hardware-Firmware) a cependant réussi à m'apporter le soutien nécessaire à la bonne réalisation de mon stage : il m'a permis de m'initier au C++, m'a aidée à ne pas me perdre dans les concepts parfois complexes de l'encodage audio et m'a aidée à apporter la rigueur scientifique nécessaire à la réalisation de mon travail. J'ai également pu bénéficier de l'expertise technique de Frédéric Lefranc (Principal Embedded System Architect dans l'équipe Hardware-Firmware) ainsi que du suivi de Justin Mannesberg (Manager de l'équipe Hardware-Firmware).

1.2 Le serveur XT

EVS développe et commercialise de nombreux produits allant des serveurs de production aux interfaces permettant d'exploiter des données audiovisuelles ou de monitorer des systèmes de production[2]. Le serveur de production live XT est un des produits emblématiques d'EVS. Il permet de stocker de grandes quantités de données audiovisuelles et d'y accéder en temps réel afin de répondre aux besoins de la production en live. Par exemple, la remote LSM (*Live Slow Motion* permet d'accéder aux contenus des serveurs XT afin de créer les ralentis pour lesquels EVS est célèbre dans le monde.

FIGURE 2 – Vues avant et arrière (en configuration IP) de l'XT-VIA[2]

Le serveur XT a connu plusieurs versions : XT, XT2, XT2+, XT3 et enfin l'XT-VIA. L'XT-VIA (cf figure 2), la plus récente version du serveur XT, en quelques informations clés[2] :

- offre un espace de stockage de 18 à 54 TB, soit plus de 130h d'enregistrement en UHD-4K ;
- dispose de 2 à plus de 16 canaux selon le format choisi : 2 canaux en UHD-8K (4320p), 6 canaux en UHD-4K (2160p) et plus de 16 canaux en FHD and HD (720p, 1080i, 1080p) ;
- permet une configuration hybride de ses entrées et sorties en IP (10G Ethernet SFP+, 100G en option, ST2022-6, ST2022-7, ST2022-8, ST2110, NMOS IS-04, IS-05, EMBER+, PTP) ou SDI (1.5G-SDI, 3G-SDI et 12G-SDI) ;
- supporte de nombreux formats d'encodage vidéo : UHD-4K (XAVC-Intra et DNxHR), HD/FHD (XAVC-I, AVC-I, DNxHD et ProRes), PROXY (MJPEG et H264) ;
- peut enregistrer 192 canaux audio non compressés et supporte les standards AES et MADI ;
- offre de nombreuses possibilités de connexion avec du matériel EVS ou non.

C'est pour la dernière génération du serveur XT, l'XT-VIA, que le codec AAC est développé. La compression avec perte de données de ce codec permet d'optimiser l'espace occupé par les données audio sans en altérer la qualité perçue. Outre la qualité audio, les performances de l'encodage sont importantes à prendre en compte pour permettre l'enregistrement de plusieurs canaux en parallèle tout en conservant un traitement de l'information qui tienne le temps réel. L'optimisation des performances doit tenir compte de l'architecture de l'XT-VIA : l'architecture ARM Neon remplace l'architecture Intel x86 de ses prédécesseurs avec des différences importantes dans les fonctions intrinsèques.

2 L'encodage audionumérique : généralités

2.1 Le son

2.2 La numérisation d'un signal

3 Les codecs audio

3.1 Définition d'un codec

Un codec est un procédé logiciel composé d'un encodeur (***co**der*) et d'un décodeur (***dec**oder*)[5]. Un codec audio permet donc, d'une part, de coder un signal audio dans un flux de données numériques et, d'autre part, de décoder ces données afin de restituer le signal audio.

Les codecs sont dits avec perte (*lossy*) ou sans perte (*lossless*). Le PCM est par exemple un codec sans perte puisqu'il encode la totalité des informations sonores dans la bande de fréquences humainement audible. Ce type de codec permet de conserver la qualité de l'audio mais nécessite en contrepartie un espace de stockage conséquent, même avec une compression des données.

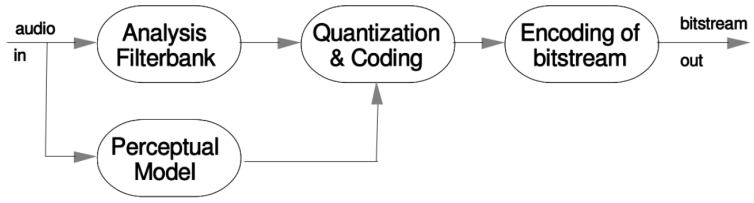
Afin de réduire l'espace de stockage nécessaire, les codecs avec perte permettent de supprimer une partie des données audio. C'est le cas des codecs définis par les normes MPEG dont fait partie le codec AAC.

3.2 Les codecs MPEG

MPEG (*Moving Picture Experts Group*) désigne une alliance de différents groupes de travail définissant des normes d'encodage, de compression et décompression et de transmission de média audio, vidéo et graphiques[6]. Le groupe est actif depuis 1988 et a produit depuis de nombreuses normes.

Les codecs audio qui implémentent les normes MPEG ont pour point commun d'être des codecs avec perte de données basés sur un modèle psychoacoustique. Le premier est le MP3, défini par la norme MPEG-1 Layer-3 ISO/IEC 11172-3 :1993. Le codec AAC est conçu en 1997 pour remplacer le MP3. Il est défini par les normes MPEG-2 partie 7 ISO/IEC 13818-7 :2006[7] et MPEG-4 partie 3 ISO/IEC 14496-3 :2019[8].

Les normes MPEG définissent les grandes lignes de l'encodage et du décodage et le format du conteneur mais pas l'implémentation du codec qui peut de ce fait être plus ou moins performant. Les codecs MPEG sont typiquement composés des blocs suivants :



(a) Schéma fonctionnel de l'encodeur

(b) Schéma fonctionnel du décodeur

Figure 3 – Vue simplifiée d'un codec MPEG basé sur un modèle psychoacoustique[3]

L'encodeur est composé des blocs suivants :

filter bank la banque de filtres décompose le signal temporel d'entrée en différentes composantes fréquentielles

perceptual model le modèle psychoacoustique utilise le signal temporel et/ou sa décomposition fréquentielle pour éliminer les données audio dont l'absence ne nuira pas à la qualité perçue à l'écoute ()

quantization and coding la quantification attribue une valeur numérique aux données du spectre de fréquences : elle sont typiquement codées avec une méthode entropique qui peut être optimisée avec le modèle psychoacoustique

encoding of bitstream les données sont formatées en un flux contenant typiquement le spectre de fréquences codé et des informations supplémentaires permettant l'encodage

Le décodeur a un fonctionnement inverse : le flux de données est décodé (**decoding of bitstream**), les composantes fréquentielles du signal sont retrouvées par l'opération inverse à la quantification (**inverse quantization**) et ces sous-bandes fréquentielles sont finalement rassemblées pour reconstituer le signal temporel (**synthesis filter bank**).

Le fonctionnement du décodeur ne sera pas plus développé dans ce travail car le bloc MDCT fait partie de la banque de filtres de l'encodeur. Le fonctionnement spécifique de l'encodeur AAC sera par contre détaillé dans la section 4.

3.3 Les modèles psychoacoustiques

Le section précédente a défini les codecs MPEG comme étant basés sur un modèle psychoacoustique. La psychoacoustique est une branche de la psychophysique qui étudie la manière dont l'oreille humaine perçoit le son[9]. Cette discipline permet d'améliorer la compression d'un signal audio en éliminant les sons qui sont captés par un microphone mais qui ne peuvent pas être perçus par l'oreille humaine et les avancées dans cette discipline permettent de développer des encodeurs audio de plus en plus performants. Les codecs basés sur un modèle psychoacoustique sont toujours des codecs avec perte puisqu'une partie des informations auditives sera définitivement perdue, ce qui ne nuit pour autant pas à la qualité perçue du son.

L'encodage audionumérique tient déjà compte des seuils de fréquences humainement audibles pour limiter les données audio enregistrées : nous l'avons vu dans la section 2.2, aucun son n'est perçu en-deça de 20Hz ou au-delà de 20kHz. La psychoacoustique permet de mieux dessiner la limite entre ce qui est humainement audible ou non afin d'éliminer un maximum des informations non pertinentes et ainsi augmenter le facteur de compression des données : le facteur de compression des codecs MPEG est environ 15 fois supérieur à celui du CD[4].

Les effets de masque sont au centre des différents modèles psychoacoustiques utilisés pour la compression audio. L'enjeu afin d'obtenir le meilleur taux de compression est de calculer le plus finement possible les seuils de masquage, i.e. la limite entre les informations pertinentes et celles qui peuvent être éliminées. Les effets de masques dans le domaine fréquentiel (*spectral masking effects*) sont parmi les plus utilisés mais il en existe d'autres, e.g. dans le domaine temporel. La figure suivante représente différents effets de masque du domaine fréquentiel :

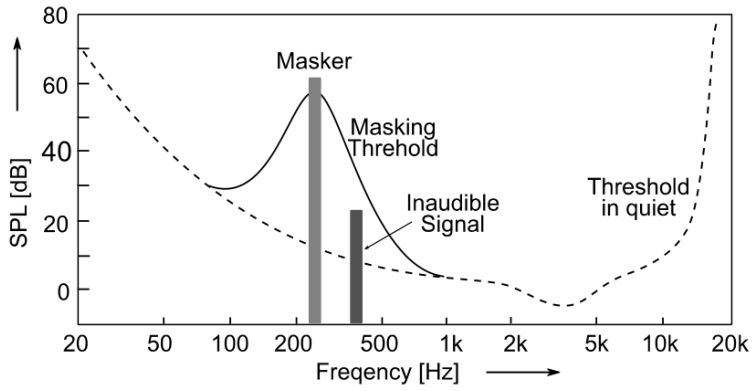


FIGURE 4 – Effets de masque dans le domaine fréquentiel[4]

Les lignes représentent le seuil

Threshold in quiet la ligne en pointillé représente le seuil d'audibilité dans le calme, indépendamment de tout autre élément qui pourrait interférer

Masking threshold

Le calcul du seuil de masquage tient compte[4] :

- des effets de masquage monophonique de la perception non-linéaire des fréquences, plus fine pour les basses fréquences ;
- des effets de masque dans le temps ;
- de l'effet de masque dans une bande de fréquence ou entre bandes de fréquences ;
- l'impact de la tonalité sur le masquage
- masking over time

4 Le codec AAC

4.1 Fonctionnement de l'encodeur AAC

Le codec AAC (Advanced Audio coding) est définit , défini par la norme . Les recherches en psychoacoustique ont permis de développer un algorithme d'encodage plus performant pour l'AAC que pour le MP3 : il permet d'encoder moins données audio tout en gardant la même qualité perçue au décodage[3].

4.2 Le bloc MDCT

5 Environnement de développement

Plateformes : passage de Intel à ARM nécessaire à cause de la lib Ne10 et impossibilité de maintenir une version de référence Intel

Développement remote sur RPI + photo raspberry

CentOS7 parce que utilisé sur tout le matériel linux EVS

Projet en C++ mais qui ressemble fort à du C car implémentation d'un codec -> améliorations possibles : juste du C ?

Le build du projet se fait grâce à un fichier CMake à la racine du projet. Ce fichier présenté dans l'annexe **??** permet d'appeler les CMake du projet *audio_encoding* (projet contenant les MDCT et ses tests) et de la librairie *Ne10*. Les commandes du CMake générant les exécutables de tests seront fournies en annexe à la suite du code source de ces tests.

6 Algorithmes MDCT de référence

La première étape de ce travail consiste à développer des algorithmes de référence afin de valider les différentes MDCT implémentées par la suite. Ces algorithmes de référence sont développés en algorithmique flottante sur base de la formule mathématique de la MDCT. Ils permettent de générer des spectres de fréquence en *float* ou en *integer* afin de valider les données de sortie des MDCT optimisées.

6.1 Description mathématique

La transformation effectuée par la MDCT est donnée par l'équation suivante[10] :

$$X_k = \frac{2}{\sqrt{2N}} \sum_{n=0}^{2N-1} x_n \cos\left[\frac{\pi}{N}\left(N + \frac{1}{2} + \frac{N}{2}\right)\left(k + \frac{1}{2}\right)\right]$$

X_k avec $k \in [0, N[$ pour une fenêtre d'entrée de 2N échantillons

x_n avec $n \in [0, 2N[$: la fenêtre d'entrée

$F : R^{2N} \rightarrow R^N$ la MDCT est une fonction linéaire qui pour 2N nombres réels en entrée produit N nombres réels en sortie

La MDCT a été implémentée avec une fenêtre d'entrée de $2N = 1024$ échantillons. Le bloc de sortie, i.e. le spectre de fréquences de la fenêtre d'entrée, aura donc une taille de 512. Ces valeurs, utilisées à de très nombreux endroits du code, sont rassemblées dans le header `mdct_constants.h` présenté dans l'**annexe B**. Ce fichier contient également d'autres valeurs précalculées sur base de la taille de la fenêtre d'entrée.

La section suivante présente deux implémentations simples de cette formule. Ces implémentations ne pourraient pas être utilisées sans avoir été optimisées car elles seraient beaucoup trop lentes pour un codec qui doit tenir le temps réel sur plusieurs canaux. La complexité de implémentation de cette formule est de $O(N^2)$ opérations (où N est la taille de la fenêtre d'entrée). Cette complexité peut être ramenée à $O(N \log N)$ opérations par une factorisation récursive. La complexité peut également être diminuée en se basant sur une autre transformation, e.g. une DFT (*Discrete Fourier Transorm*) ou une autre DCT (*Discrete Cosine Transform*) : la complexité sera alors de $O(N)$ opérations de *pre-* et *post-processing* en plus de la complexité de la DFT ou de la DCT choisie[10]. C'est cette dernière option qui a été retenue pour ce travail.

6.2 Implémentations des MDCT de référence en *floating point* et *fixed point*

La formule mathématique de la MDCT a été implémentée très simplement en algorithmique flottante avec la possibilité d'obtenir le spectre de fréquences codés en *float*, *double* ou *int32*. L'objectif de ces implémentations est de pouvoir valider les spectres de fréquence calculés par les implémentations optimisées de la MDCT. Les MDCT de référence serviront égaelement à mesurer la précision des MDCT optimisées.

La première implémentation de l'équation de la MDCT est présentée dans l'**annexe C.1**. La fonction développée permet de faire ses calculs et d'obtenir un résultat aussi bien en *float* (32 bits) qu'en *double* (64 bits) grâce à l'utilisation d'un *template*. Le signal temporel a la même précision (*float* ou *double*) que le spectre généré.

La seconde fonction de référence est présentée dans l'**annexe C.2**. Elle permettra de vérifier les résultats des implémentations optimisées en *fixed point*. Tous les calculs ne sont pas fait en algorithmique entière : la fonction fait les mêmes calculs que la fonction de référence en algorithmique flottante (uniquement en *double* cette fois pour garder le plus de précision possible) et transtype le résultat final dans un *integer* de 32 bits qui correspond à une notation Qx.15 signée.

6.3 Validation des algorithmes de référence

Validation avec un code d'exemple qui correspond à un signal sinusoidal (single tone) -> annexes : génération d'un signal single tone **??**+ code qui sort les données. Code pas ici mais renvoi à la section sur la validation des données -> pour le float, on regarde ce qui sort, pour le integer, on vérifie avec calcul q15

Présentation des résultats sous forme de données brutes ou de graphique.

Explication de la lecture des résultats, calcul des bandes de fréquences représentées. Mise en évidence qu'on a bien une seule composante fréquentielle comme attendu pour un signal single tone

+ les résultats auraient été plus précis avec la fonction de fenêtre -> voir améliorations possibles

7 Algorithme MDCT basé sur la FFT

7.1 Optimisations attentues

Choix de l'optimisation de la MDCT basé sur sur une DCT : la FFT -> complexité de O (N log N) (N taille de la fenetre) + O(N) opérations de pré et de post processing.

But : utiliser une FFT déjà optimisée pour n'avoir à optimiser "que" les opérations de pré et de post processing Exemple trouvé sur le site DSP related -> Annexe ? Citation ?

7.2 Implémentation de la MDCT basée sur la FFT de la librairie FFTW3

Code développé à partir de cet exemple en annexe. Il est basé sur la librairie FFTW3 qu'on ne peut pas garder car ne permet pas de travailler en integer -> sera amené à être retravaillé.

Explications des paramètres du code :

- fenêtre de 1024
- pre-twiddle -> 256
- FFT => FFT avec une fenêtre d'entrée réduite et donc une complexité réduite
- post-twiddle -> 512

7.3 Validation

Validé avec un single tone signal + l'algo de référence.

Code d'exemple qui teste l'algo de référence et l'algo FFTW3 avec les mêmes données d'entrée pour comparaison. (on fait la différence pour la précision ? ce n'est peut être pas pertinent de la faire déjà)

Présentation des résultats sous forme de données brutes ou de graphiques.
A ce niveau, pour valider, j'ai aussi essayé de faire la IMDCT mais dû à l'overlap, sur une seule fenetre, on ne sait pas reconstruire le signal -> pas possible de valider comme ça

8 Intégration de la librairie *Ne10*

8.1 Choix de la librairie

La librairie *FFTW3* utilisée pour l'itération précédente de la MDCT ne propose pas de FFT en algorithmique entière. Le passage à une autre librairie était donc nécessaire et le choix s'est porté sur la librairie *Ne10* qui propose différentes FFT en *fixed point*.

Le projet *Ne10* propose toute une série de fonctions mathématiques et physiques de base ainsi que des fonctions de traitement de signal et de traitement d'image. La librairie est spécifiquement développée pour les architectures ARM possédant les opérations SIMD Neon (ARMv7 et ARMv8-A)[11].

Ne10 propose à la fois des fonctions développées en *plain C* et des fonctions optimisées avec les instructions SIMD Neon : les deux types de fonctions seront utilisées pour développer une MDCT optimisée et pour conserver une MDCT de référence en *plain C*. Maintenir une MDCT *plain C* permettra d'avoir une référence pour la mesure des performances de l'algorithme *fixed point* mais pourrait aussi s'avérer utile pour une utilisation de l'encodeur AAC sur une architecture ARM ne possédant pas les instructions Neon.

L'utilisation de la librairie *Ne10* est soumise à la licence *3-Clause BSD*, license permissive qui permet un usage commercial des produits intégrant la librairie et qui ne contraint pas à en distribuer le code source[12].

8.2 Implémentation de la MDCT basée sur la FFT *Ne10* en *float 32*

La librairie *Ne10* s'installe simplement en suivant les instructions données par la documentation : clone du projet GitHub, run du CMake et build du projet[11]. La librairie ne peut cependant être installée que sur une plateforme Linux, Android ou iOS reposant sur une architecture ARM. À partir de ce moment, il n'est donc plus possible de maintenir une implémentation de référence de la MDCT pour une architecture Intel.

Ne10 propose des algorithmes de FFT *real to complex* et *complex to complex* en *floating point* (32 bits) ou en *integer* (32 bits et 16 bits). L'objectif est évidemment de passer toute la MDCT en *integer* mais pour un premier test de la librairie, l'algorithme de la section précédente a tout d'abord été repris en remplaçant la FFT de *FFTW3* par la fonction `ne10_fft_c2c_1d_float32_neon` de *Ne10* : FFT en *complex to complex* en *float 32*, i.e. l'entrée et la sortie de la FFT sont représentées sous forme de tableaux de nombres complexes codés en *float* sur 32 bits.

L'annexe **??** présente le code de cette implémentation. L'annexe **??** montre que ce code est construit sur le même modèle que le code utilisant la librairie *FFTW3*. La classe contient la configuration de la FFT et les tableaux contenant les données d'entrée, les données de sortie et les facteurs de twiddling. La classe définit trois fonctions publiques : le constructeur, le destructeur et la fonction MDCT.

L'annexe **??** montre l'initialisation de la MDCT dans le constructeur de la classe. Le constructeur initialise les tableau de facteurs de twiddling de le même manière que l'algorithme basé sur la FFT de *FFTW3*. La configuration de la FFT de *Ne10* se fait conformément au code d'exemple donné par la documentation de la librairie[11] avec en paramètre la taille de la FFT qui correspond au quart de la taille de la fenêtre d'entrée.

Le destructeur présenté à l'annexe **??** permet de libérer la mémoire allouée pour la FFT en appelant la fonction adéquate de la librairie *Ne10*.

La fonction MDCT présentée dans l'annexe **??**, comme pour l'algorithme précédent :

- effectue les opérations de *pre-processing* ou *pre-twiddling* : ce sont les mêmes que celle de la MDCT *FFTW3* en arithmétique *floating point* sur 32 bits ;
- appelle l'algorithme de FFT : la FFT de *Ne10* prend en paramètres les tableaux contenant les données d'entrée et de sortie, la configuration de la FFT et un *integer* à `0` pour réaliser la FFT ou à `1` pour réaliser l'opération inverse ;
- effectue les opérations de *post-processing* ou *post-twiddling* : ici aussi les mêmes que celles de la MDCT *FFTW3* en arithmétique *floating point* sur 32 bits.

L'algorithme développé ici ne diffère donc pas de l'algorithme présenté à la section précédente. Son développement est trivial mais il permet de tester et de valider le fonctionnement de la librairie *Ne10*.

8.3 Validation

L'utilisation de la librairie *Ne10* est validée par comparaison du spectre de fréquences qu'elle génère avec les spectres générés par la MDCT de référence en *double* et par la MDCT basée sur *FFTW3* également en *double*. Les MDCT sont appelée en *double* pour plus de précision. La comparaison aurait pu se faire sur base de tous les algorithmes de MDCT en *float* sur 32 bits mais le choix qui a été fait ici permet en plus de vérifier que la perte de précision entre le *float* et le *double* soit acceptable.

Le code source permettant de comparer les trois MDCT en *floating point* est présenté à l'annexe **??**. Il sera expliqué en détail dans la section 11.1 consacrée à la validation des données des MDCT. Compilé avec les commandes CMake de l'annexe **??**, le code produit un exécutable permettant de comparer les spectres de fréquence produits par les trois MDCT *floating point* en les affichant en console.

Figure 5

En redirigeant les données sorties en console vers un fichier texte, il est possible d'exploiter ces données sous forme de graphique. (voir figure 5) TODO : explication du graphique

TODO : présentation de la précision

8.4 Performances

Une fois l'utilisation de la librairie *Ne10* validée, Mesure de la différence de performance entre différentes FFT : code en annexe

FFT Ne10 f32 plain C average run time : 5632.06 ns standard deviation : 4.38439e-09 ns
FFT Ne10 f32 Neon average run time : 3115.18 ns standard deviation : 2.81769e-06 ns
FFT Ne10 i32 plain C average run time : 10723.1 ns standard deviation : 1.77145e-06 ns
FFT Ne10 i32 Neon average run time : 3455.78 ns standard deviation : 5.35823e-07 ns
FFT Ne10 i16 plain C average run time : 9557.52 ns standard deviation : 1.69145e-07 ns
FFT Ne10 i16 Neon

9 Algorithme MDCT en arithmétique fixed point

Le passage d'une arithmétique flottante à une arithmétique entière est une des optimisations envisagées par l'analyse préalable à ce travail. Le bénéfice attendu est double : l'arithmétique entière est généralement plus rapide et un bloc MDCT en arithmétique entière permettrait de mieux intégrer le bloc MDCT à l'ensemble de l'encodeur.

La première de ces attentes n'a pas pu être rencontrée dans ce travail. En effet, l'architecture ARMv7 du Raspberry supporte de nombreuses instructions en *floating point* sur 32 bits[13]. Or, plusieurs instructions *fixed point* sont souvent nécessaires pour remplacer une seule instruction en *floating point*, e.g. une multiplication en float est remplacée par une multiplication et une ou plusieurs rotations de bits. Plutôt que de gagner en temps d'exécution, le passage en *fixed point* a ralenti la MDCT.

Cependant, le passage en *fixed point* permet d'économiser un transtypage du *integer* vers le *float* en entrée de la MDCT et inversement en sortie. Avec un temps d'exécution au moins équivalent en *fixed point* qu'en *floating point*, le passage en *fixed point* permettrait donc tout de même d'améliorer les performances de l'ensemble de l'encodeur AAC.

Enfin, les données reçues à l'entrée de la MDCT sont codées en *integer* sur 16 bits alors que les *float* sont codés au minimum sur 32 bits. Garder le maximum de données et d'opérations en 16 bits permettrait de gagner en performance au moment de l'utilisation des opérations SIMD.

9.1 Arithmétique fixed point

La représentation *fixed point* est une alternative au *float* pour le codage des nombres décimaux[14]. Le principe est de réserver un certain nombres de bits pour coder la partie entière et un autre nombre de bits pour coder la partie décimale du nombre. Ce travail utilise deux notations pour la représentation en *fixed point*, toujours signées :

- Qm où m est le nombre de bits réservés aux décimales, e.g. une notation Q15 sur 16 bits permet de représenter un nombre signé ne contenant que des décimales et pas de partie entière (1 bit est réservé pour le signe) ;
- $Qx.y$ où x est le nombre de bits réservés à la partie entière et y le nombre de bits réservés à la partie décimale, e.g. une notation Q1.15 équivaut à une notation Q15 sur 16 bits.

Pour passer la MDCT en algorithmique *fixed point*, il faut tout d'abord prêter attention à choisir la représentation adéquate. Par exemple, les données d'entrée de la MDCT sont comprises entre 0.9 et -0.9. Elles peuvent donc être représentées en Q15. Si elles avaient été comprises ente 1 et -1, la conversion à une représentation Q15 aurait produit un dépassement sur l'une des valeurs limites et par conséquent une perte d'information.

Des dépassements peuvent également se produire lors des opérations arithmétiques :

- une **addition** ou une **soustraction** peut causer un dépassement d'1 bit, e.g. la somme d'un nombre sur 32 bits et d'un nombre sur 16 bits nécessite potentiellement 33 bits pour être codée ;
- le résultat d'une **multiplication** ou d'une **division** peut devoir être codé sur un nombre de bits équivalent à la somme des nombres de bits composants les deux nombres multipliés ou divisés, e.g. le produit d'un nombre de 16 bits multiplié par un nombre de 32 bits nécessite potentiellement 48 bits pour être codé.

Ces dépassements sont théoriques. En fonction des données réelles à traiter, il est possible de ne pas respecter à la lettre les règles énoncées plus haut. C'est le cas par exemple avec les facteurs de *twiddling* dont on sait qu'il valent au maximum un quart de la valeur d'un sinus ou d'un cosinus et qui sont codés en Q15 : il est alors certain que l'addition de deux de ces nombres ne causera pas de dépassement.

Là où l'implémentation en *floating point* était triviale, puisqu'elle demandait simplement de reprendre le code d'exemple fourni et de l'adapter à l'utilisation de la librairie *Ne10*, l'implémentation en *fixed point* devient plus complexe : il faut prêter attention à coder les nombres dans les bonnes ranges et implémenter les différentes opérations arithmétiques de sorte à ne pas causer de dépassements.

9.2 Implémentation de la MDCT basée sur la FFT Ne10 en arithmétique *fixed point*

L'**annexe J** présente l'implémentation de la MDCT *Ne10 fixed point* en *plain C*. La classe `mdct_ne10_i32_c` a la même structure que l'implémentation en *floating point*. Le header présenté dans l'**annexe J.1** montre que seul le type des données a changé :

- La configuration et les tableaux d'entrée et de sortie de la FFT sont définis avec les types *int32*, et non plus *float32*, de la librairie *Ne10* ;
- Le tableau de facteurs de *twiddling* passe du `float` au `int16_t` ;
- La fonction MDCT prend en paramètres un signal temporel en `int16_t` et renvoie un spectre en `int32_t` au lieu des tableaux de `float`.

9.2.1 Utilisation de la FFT *Ne10* en *int 32*

L'utilisation de la librairie *Ne10* en *integer* est très peu différente de son utilisation en *float* :

- La configuration est initialisée dans le constructeur de la classe `mdct_ne10_i32_c` (**annexe J.2**) en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée.
- L'espace alloué à la configuration est libéré dans le destructeur de la classe `mdct_ne10_i32_c` (**annexe J.3**) avec la fonction appropriée de la librairie *Ne10* ;
- La fonction de FFT est appelée avec en paramètres le tableau contenant les données d'entrée, le tableau dans lequel sera calculé le résultat de la FFT, la configuration préalablement initialisée, un *integer* qui indique à la fonction de réaliser la FFT et non son opération inverse et, en plus de la FFT en *float*, un facteur de mise à échelle mis ici à 0.

Il est à noter que l'initialisation du facteur de mise à échelle de la fonction de FFT n'est pas documentée dans la librairie *Ne10*. L'effet de ce facteur sur la FFT n'est pas non plus indiqué. J'ai initialisé ce facteur à 0 après avoir testé la FFT de *Ne10* dans le but d'obtenir les mêmes valeurs qu'en *float* mises à une échelle Q15.

La documentation incomplète de la librairie a été une des difficultés de ce travail. En dehors des quelques codes d'exemples disponibles, il est très compliqué de savoir quelles données sont attendues et sont produites par les fonctions de *Ne10*. Il est également très difficile de trouver des ressources externes sur l'utilisation de *Ne10*.

Le manque de documentation est également ce qui a empêché l'utilisation de la FFT en *int16* plutôt qu'en *int32*. Des opérations en 16 bits pour tout le bloc MDCT auraient été plus performantes. Malheureusement, si la documentation de *Ne10* dit bien travailler en Q15 pour du 16 bits et en Q31 pour du 32 bits, elle ne dit pas quelle *headroom* prévoir, quelles *ranges* de valeurs sont acceptées, si les opérations saturent ou non, etc. En testant la FFT 16 bits avec des données codées en Q15, trop de valeurs étaient fausses. Je n'ai pas non plus réussi à obtenir de résultat satisfaisant en essayant de jouer avec le facteur de mise à échelle.

Pour ne pas produire de dépassement dans la fonction de FFT, il a fallu prévoir 8 bits de *headroom* pour la FFT. Sur des données codées sur 16 bits, cela signifie qu'il ne resterait plus que 8 bits de données utiles pour le signal, dont 1 bit pour le signe. Cette perte de précision trop importante a forcé l'utilisation de la FFT en 32 bits.

9.2.2 Initialisation des *twiddle factors*

Le tableau de facteurs de `twiddle` est initialisé dans le constructeur de la classe `mdct_ne10_i32_c` (**annexe J.2**). Ses valeurs sont calculées en *double* puis converties en *integer* (représentation Q15).

Il aurait été posible de convertir les opérations d'initialisation en *fixed point* mais l'optimisation du constructeur n'est pas nécessaire. En effet, puisque la taille de fenêtre d'entrée ne varie pas, l'appel de l'initialisation ne sera fait qu'une fois.

9.2.3 Opérations de *pre-twiddling* et de *post-twiddling*

L'essentiel du tavail pour cette implémentation a été le passage des opérations de *pre-* et de *post-processing* en arithmétique *fixed point*. Le résultat de ce travail est présenté dans l'**annexe J.3** qui présente la fonction MDCT basée sur la FFT de *Ne10* en *plain C*.

Les opérations de *pre-twiddling* transforment le signal temporel et les facteurs de *twiddling*, tous deux en représentation Q15, en un nombre en représentation Q1.23 sur 32 bits à donner en entrée de la FFT de *Ne10* en *int32*. La représentation Q1.23 est celle qui permet de garder le maximum de précision tout en s'assurant de ne pas produire de dépassement dans la FFT en gardant 8 bits de *headroom*.

Le *post-twiddling* récupère les données en Q9.23 produites par la FFT de *Ne10* : la précision de 23 décimales donnée en entrée, le bit de signe et les 8 bits de *headroom*. En combinaison avec les facteurs de *twiddling* codés en Q15, elles sont transformées en un spectre de fréquences codé en Q9.15 sur 32 bits.

Les opérations de *pre-processing* suivent toujours le même schéma : les parties réelles ou imaginaires des données d'entrée de la FFT sont la somme de deux produits d'un facteur de *twiddling* et de la somme de deux échantillons du signal temporel. Voici en exemple la transformation de la première opération de l'arithmétique *floating point* vers l'arithmétique *fixed point* :

$$fft\_in[i/2].r = r0 * c + i0 * s;$$

devient

$$fft\_in[i/2].r = (((r0 * c) + 64) >> 7) + (((i0 * s) + 64) >> 7);$$

où $r0 = time\_signal[MDCT_M32 - 1 - i] + time\_signal[MDCT_M32 + i];$

et $i0 = time\_signal[MDCT_M2 + i] - time\_signal[MDCT_M2 - 1 - i];$

r0 est la somme de deux échantillons du signal temporel codés en Q1.15 : il correspond donc à une notation **Q2.15** ici codée sur 32 bits plutôt que de perdre un bit de précision pour garder la valeur sur 16 bits ;

c *twiddle factor*, est codé en Q1.15, mais on sait que le facteur de mise à échelle pour une fenêtre de 1024 échantillons temporels l'a réduit à $\frac{1}{4}$ de la range possible du Q1.15, noté **Q1.15/4** ;

r0*c le produit d'un nombre en Q2.15 avec un nombre Q1.15 est théoriquement un Q2.30 mais tient en pratique sur du **Q1.30/2** puisque le *twiddle factor* est en Q1.15/4 ;

((r0*c)+64)»7 le résultat en Q1.30/2 est ramené à du **Q1.23/2** par une opération de shift avec arrondi ($64 = 2^6$, ajouter un bit en $7^{\text{ème}}$ position en partant du LSB permet d'arrondir la valeur du $8^{\text{ème}}$ bit) ;

((i0*s)+64)»7 l'opération est équivalente à la précédente et est donc également codée en **Q1.23/2** ;

(((r0*c)+64)»7) + (((i0*s)+64)»7) l'addition de deux Q1.23/2 donne un **Q1.23**, codé sur 32 bits, il permet de conserver les 8 bits de *headroom* nécessaire à la FFT.

Les opérations de *post-processing* suivent elles aussi toujours le même schéma : les valeurs du spectre de fréquences sont calculées en additionnant deux produits d'un facteur de *twiddling* et d'une donnée de sortie de la FFT. Voici en exemple la conversion de la première opération de *post-twiddling* en *fixed point* :

$$spectrum[i] = -r0 * c - i0 * s;$$

devient

$$spectrum[i] = ((((-r0 + 128) >> 8) * c + 16384) >> 15) - ((((i0 + 128) >> 8) * s + 16384) >> 15);$$

où $r0$ et $i0$ sont les parties réelle et imaginaire des données de sortie de la FFT

r0 donnée de sortie de la FFT, est codé en **Q9.23** ;

(-r0+128)»8 $r0$ est ramené en **Q9.15** par un *shift* de 8 avec arrondi ;

c *twiddle factor*, est codé en Q1.15, mais on sait que le facteur de mise à échelle pour une fenêtre de 1024 échantillons temporels l'a réduit à $\frac{1}{4}$ de la range possible du Q1.15, noté **Q1.15/4** ;

((-r0+128)»8)*c la multiplication d'un Q9.15 par un Q1.15/4 donne un Q10.30/4 ou **Q9.30/2** ;

(((-r0+128)»8)*c+16384)»15 le résultat en Q9.30/2 est ramené à du **Q9.15/2** par un *shift* de 15 avec arrondi ;

(((i0+128)»8)*s+16384)»15 le second terme effectue les mêmes opérations pour un résultat en **Q9.15/2** ;

((((-r0+128)»8)*c+16384)»15) - ((((i0+128)»8)*s+16384)»15) le différence de deux Q9.15/2 est un Q10.15/2 ou **Q9.15**.

9.3 Performances

Les performances de la MDCT *fixed point* en *plain C* sont moins bonnes que celles de l'implémentation en *floating point* :

- La MDCT *Neon float 32 plain C* a un temps d'exécution moyen de 9 µs ;
- La MDCT *Neon integer 32 plain C* a un temps d'exécution moyen de 15 µs.

Les détails de la mesure des performances sont exposés dans la section 11.2.

Le temps d'exécution notablement plus conséquent s'explique par le fait que l'architecture ARMv7 supporte de nombreuses opérations *floating point* sur 32 bits. Contrairement à des processeurs plus anciens, l'arithmétique *fixed point* n'est donc pas plus rapide sur l'ARMv7.

De plus, la section 9.2 a montré qu'une seule instruction en *floating point* était remplacée par plusieurs opérations *fixed point*. Une multiplication est, par exemple, remplacée par une multiplication, une rotation et une addition pour l'arrondi. Puisque les opérations sur des *integer* ne sont pas plus rapides que sur des *float*, il n'est donc pas étonnant que la MDCT *fixed point* soit plus lente que la MDCT *floating point*.

Cette explication est cohérente avec les mesures des performances des différentes FFT (section 8.4) : la FFT de *Ne10* en *int 32 plain C* a un temps d'exécution moyen de 11 µs contre 6 µs pour la FFT *Neon float 32 plain C* et 5 µs pour la FFT *FFTW3 float 32*.

10 Optimisations à l'architecture ARM

10.1 Spécificités de l'architecture ARMv7

Globalement reprend les tutos ARM

10.2 Utilisation des fonctions Neon SIMD (intrinsic)

Les opérations SIMD permettent de faire plusieurs opérations en une fois où l'algo normal n'en fait qu'une à la fois. L'algo SIMD permet de faire plusieurs modifications à la fois -> il faut ranger les données de manière à pouvoir l'appliquer facilement (fonction fenêtre -> tri des données ?)

Plus les données sur lesquelles on travaille sont petites, plus on peut faire d'opérations en parallèle -> il faut voir si la perte de précision est ok. -> mettre une représentation graphique, c'est beaucoup plus simple à comprendre. D'où l'intérêt de passer à du 16 bits, plutôt que de rester en 32 bits et il faut absolument éviter le 64 bits (aucun intérêt).

Attention au flag pour la compilation + au header (accès aux intrinsics pas activé par défaut)

11 Analyse des résultats

11.1 Validation des données

Les sections précédentes ont montré que la les différentes itérations de la MDCT développées ont été validées à chaque étape. Cette validation consiste essentiellement en une vérification manuelle des données de sortie de la MDCT : avec un signal sinusoïdal connu en entrée, il est facile de vérifier que l'analyse fréquentielle ne contient bien qu'une seule composante fréquentielle, que la vérification se fasse en lisant les données brutes à la sortie de l'algorithme ou par une analyse graphique de celle-ci.

Ces tests auraient pu être améliorés en automatisant la vérification, e.g. en générant une fois les données de référence attendues pour permettre le développement d'un code de test qui compare automatiquement les données de références avec les données à vérifier. En effet, devoir relancer les tests et vérifier les données à chaque fois qu'une modification est faite dans le code peut s'avérer laborieux et mettre en place des tests automatiques aurait permis de gagner un temps précieux.

11.2 Gain en performance

La mesure des performances a pour but de valider le bloc MDCT avant de l'intégrer au codec AAC. Le cahier des charges du stage ne contenait pas d'objectif à atteindre en terme de performances, ni absolu (e.g. un temps d'exécution maximal à respecter dans des conditions données), ni relatif (e.g. gagner un certain pourcentage de performances par rapport à une MDCT de référence).

L'objectif en terme de temps d'exécution n'étant pas fini, il a été décidé de tenter de gagner le maximum de performances sur le temps de mon stage. Le critère de réussite est dès lors d'obtenir des performances au moins équivalentes pour la version finale de la MDCT que pour ses itérations précédentes.

Le temps d'exécution de la MDCT *fixed point* doit évidemment être inférieur au temps d'exécution de l'algorithme de référence puisque celui-ci ne contient aucune optimisation. Ce temps peut toutefois être équivalent au temps d'exécution de la MDCT *floating point* : à performances équivalentes, l'algorithme *fixed point* rendra tout de même l'encodeur AAC plus performant en économisant les transtypages *integer-floating point* à l'entrée et à la sortie du bloc MDCT. En effet, les données sont reçues par la MDCT en *integer* et devront être traitées en *integer* par le bloc de quantification à la sortie de la MDCT.

Le temps d'exécution des différentes itérations de la MDCT a été mesuré sur base du code de l'**annexe TODO** compilé par les commandes CMake présentées dans l'annexe l'**annexe TODO**. Le code permet de générer plusieurs exécutables en fonction de la variable de préprocesseur définie à la compilation, afin de pouvoir tester :

- La MDCT *Ne10 float 32 plain C* ;
- La MDCT *Ne10 integer 32 plain C* ;
- La MDCT *Ne10 integer 32 neon* ;
- La MDCT de référence en *float 32*.

Le code de l'**annexe TODO**, compilé avec les commandes de l'**annexe TODO** génère un exécutable permettant de mesurer le temps moyen d'exécutions de la MDCT *FFTW3 float 32*.

Les résultats présentés dans la table 1 ont été mesurés sur 10 000 000 d'exécutions. Afin de ne pas introduire d'aléatoire dans ces mesures, les MDCT ont toutes été testées avec le même signal temporel en entrée : un signal sinusoïdal à 440Hz. Les mesures de performances ont été prises avec des exécutables compilés en mode *release* avec l'option `CMAKE_BUILD_TYPE=RELEASE`.

MDCT	**Temps d'exécution moyen (ns)**	**Écart type (ns)**
Ne10 i32 Neon	5796.35	0.02259×10^{-6}
Ne10 i32 C	14939.7	0.93554×10^{-6}
Ne10 f32	9020.1	0.02251×10^{-6}
FFTW3 f32	7446.84	1.29403×10^{-6}
Reference f32	29212.6×10^{3}	0.77552×10^{-6}

Table 1 – Test de performance des algorithmes MDCT

Les résultats montrent que la MDCT optimisée avec les instructions SIMD Neon est bien plus rapide que les autres MDCT développées avec un temps d'exécution moyen de 5796.35 ns. L'objectif du stage est donc bien atteint.

L'algorithme MDCT de référence est le plus lent de tous avec un temps d'exécution moyen de 29 212.6 µs. Ce résultat est tout à fait normal puisque cet algoritme a été développé sans optimisation particulière.

La MDCT *fixed point* en *plain C* est la plus lente des MDCT optimisées avec un temps d'exécution moyen de 14 939.7 ns. La section 9.3 permet de comprendre en quoi ce résultat est cohérent puisque l'architecture ARMv7 supporte de nombreuses instructions en *float* sur 32 bits et que l'arithmétique *fixed point* nécessite souvent plusieurs opérations là où une seule est nécessaire en *floating point*.

Ces résultats montrent que l'implémentation des fonctions SIMD Neon était nécessaire afin d'obtenir des performances acceptables. Sans l'implémentation d'une MDCT optimisée avec ces instructions, l'utilisation d'une MDCT *fixed point* aurait été compromis par les résultats insatisfaisants de l'implémentation *plain C*.

Enfin, pour les implémentation plain C, la MDCT *Ne10* est un peu plus lente que la MDCT *FFTW3* avec un temps d'exécution moyen de 9020.1 ns contre 7446.84 ns. Ce résultat est cohérent avec les performances des différentes FFT mesurées à la section 8.4 : la FFT de *FFTW3* en *float 32* est en effet environ 2 µs plus rapide que la FFT de *Ne10* en *float 32 plain C*.

11.3 Perte de précision

12 Améliorations possibles

- Fonction fenêtre intégrée aux opérations de pre twiddling
- Quantification intégrée au post twiddling
- tests automatisés
- code en C
- tests de performances plus poussés avec comparaison avec un algo existant

Conclusion

Sur base du cahier des charges de début de stage, il a été décidé que mon travail s

Références

[1] EVS Website, "Page d'accueil d'EVS Broadcast Equipment." [https://evs.com], consulté le 21 avril 2022.

[2] EVS Website, "Page de présentation des produits commercialisés par EVS Broadcast Equipment." [https://evs.com/products], consulté le 21 avril 2022.

[3] K. Brandenburg, "Mp3 and aac explained," *AES 17th International Conference on High Quality Audio Coding*, 1999.

[4] J. Herre and S. Dick, "Psychoacoustic models for perceptual audio coding—a tutorial review," *Applied Sciences*, vol. 9, p. 2854, 07 2019.

[5] Wikipedia, "Codec." [https://en.wikipedia.org/wiki/Codec], consulté le 2 mai 2022.

[6] Wikipedia, "Moving picture experts group." [https://en.wikipedia.org/wiki/Moving_Picture_Experts_Group], consulté le 30 mars 2022.

[7] "Information technology – Generic coding of moving pictures and associated audio information – Part 7 : Advanced Audio Coding (AAC)," standard, International Organization for Standardization, 2006. [https://www.iso.org/standard/43345.html].

[8] "Information technology – Coding of audio-visual objects – Part 3 : Audio," standard, International Organization for Standardization, 2019. [https://www.iso.org/standard/76383.html].

[9] Wikipedia, "Psychoacoustics." [https://en.wikipedia.org/wiki/Psychoacoustics], consulté le 2 mars 2022.

[10] Wikipedia, "Modified discrete cosine transform." [https://en.wikipedia.org/wiki/Modified_discrete_cosine_transform], consulté le 17 septembre 2021.

[11] Project Ne10 Website, "Documentation du projet Ne10." [http://projectne10.github.io/Ne10/doc/], consulté le 9 mai 2022.

[12] "The 3-Clause BSD License." [https://opensource.org/licenses/BSD-3-Clause], consulté le 9 mai 2022.

[13] "Instruction Set Assembly Guide for Armv7 and earlier Arm architectures – Version 2.0 – Reference Guide," 2019. [https://developer.arm.com/documentation/100076/0200].

[14] E. Oberstar, "Fixed-Point Representation & Fractional Math Revison 1.2," 08 2007.

Liste des annexes

A CMake principal

Fichier CMake principal placé à la racine du projet. Il permet de compiler :

- le projet *audio_encoding* contenant les différentes MDCT et leurs tests : les commandes CMake de ce sous projet sont présentées dans les annexes suivantes sous le code qu'elles permettent de compiler ;
- la librairie *Ne10* : les variables suivantes sont initialisées conformément aux recommendations de la documentation pour la compilation de la librairie :
  - NE10_LINUX_TARGET_ARCH est initialisée à armv7 (l'architecture du Raspberry Pi 4) ;
  - GNULINUX_PLATFORM est initialisée à ON ;
  - BUILD_DEBUG est initialisée à ON si le projet est compilé en mode *debug*.

```
cmake_minimum_required(VERSION 3.13)

set(NE10_LINUX_TARGET_ARCH armv7)
set(GNULINUX_PLATFORM ON)
if (CMAKE_BUILD_TYPE STREQUAL "DEBUG")
    set(BUILD_DEBUG ON)
endif (CMAKE_BUILD_TYPE STREQUAL "DEBUG")

add_subdirectory(audio_encoding)
add_subdirectory(Ne10)
```

B Valeurs constantes des MDCT

Le fichier `mdct_constants.h` rassemble les valeurs constantes des MDCT pour une fenêtre d'entrée de 1024 échantillons.

```
// Sampling frequency: 48kHz
#define FS                  48000

// Window length and derived constants
#define MDCT_WINDON_LEN     1024
#define MDCT_M              (MDCT_WINDON_LEN>>1)     // spectrum size
#define MDCT_M2             (MDCT_WINDON_LEN>>2)     // fft size
#define MDCT_M4             (MDCT_WINDON_LEN>>3)
#define MDCT_M32            (3*(MDCT_WINDON_LEN>>2))
#define MDCT_M52            (5*(MDCT_WINDON_LEN>>2))
```

C Algorithmes de référence

C.1 MDCT de référence en *float*

Algorithme de référence basé sur la formule mathématique de la MDCT. Le template permet de réaliser les calculs en *float* ou en *double*.

```
#include <cmath>

#include "mdct_constants.h"

template<typename FLOAT>
void ref_float_mdct(FLOAT *time_signal, FLOAT *spectrum)
{
    FLOAT scale = 2.0 / sqrt(MDCT_WINDON_LEN);
    FLOAT factor1 = 2.0 * M_PI / static_cast<FLOAT>(MDCT_WINDON_LEN);
    FLOAT factor2 = 0.5 + static_cast<FLOAT>(MDCT_M2);
    for (int k = 0; k < MDCT_M; ++k)
    {
        FLOAT result = 0.0;
        FLOAT factor3 = (k + 0.5) * factor1;
        for (int n = 0; n < MDCT_WINDON_LEN; ++n)
        {
            result += time_signal[n] * cos((static_cast<FLOAT>(n) + factor2) * factor3);
        }
        spectrum[k] = scale * result;
    }
}
```

C.2 MDCT de référence en *integer*

Algorithme de référence basé sur la formule mathématique de la MDCT. Le spectre est calculé en *double* puis converti en *integer* sur 32 bits en représentation Q15.

```
#include <cassert>

#include "ref_mdct.h"

void ref_int_mdct(int16_t *time_signal, int32_t *spectrum)
{
    double scale = sqrt(MDCT_WINDON_LEN) / 2.0;      // MDCT scale (2/sqrt(WIN_LEN)) + Q15 scale
    double factor1 = 2.0 * M_PI / MDCT_WINDON_LEN;
    double factor2 = 0.5 + MDCT_M2;
    for (int k = 0; k < MDCT_M; ++k)
    {
        double result = 0.0;
        double factor3 = (k + 0.5) * factor1;
        for (int n = 0; n < MDCT_WINDON_LEN; ++n)
        {
            result += time_signal[n] * cos((n + factor2) * factor3);
        }
        assert(round(result*scale) == static_cast<int32_t>(round(result*scale)));
        spectrum[k] = static_cast<int32_t>(round(result/scale));
    }
}
```

D Génération d'un signal sinusoïdal

D.1 Génération d'un signal sinusoïdal en *float*

Code de génération d'un signal sinusoïdal en *float* ou en *double*.

```
#include <cmath>

template<typename FLOAT>
void sin_float(FLOAT *out, int n_samples, double amplitude,
    double frequency, double phase_shift, int sampling_frequency)
{
    FLOAT omega = 2.0 * M_PI * frequency / static_cast<FLOAT>(sampling_frequency);
    for (int i = 0; i < n_samples; ++i)
    {
        out[i] = amplitude * sin(static_cast<FLOAT>(i) * omega + phase_shift);
    }
}
```

D.2 Génération d'un signal sinusoïdal en *integer*

La génération du signal sinusoïdal en *integer* fait appel à la génération du signal sinusoïdal en *double* avant de convertir le résultat en *integer* (représentation Q15).

```
#include <cstring>

#include "sin_wave.h"

void sin_int(int16_t *out, int n_samples, double amplitude,
    double frequency, double phase_shift, int sampling_frequency)
{
    double scale = 1.0;
    if (abs(amplitude) < 1.0) scale *= amplitude;

    double *temp_sin = static_cast<double *>(malloc(n_samples*sizeof(double)));
    memset(temp_sin, 0, n_samples*sizeof(double));

    sin_float<double>(temp_sin, n_samples, scale, frequency, phase_shift, sampling_frequency);

    for (int i = 0; i < n_samples; ++i)
    {
        out[i] = static_cast<int16_t>(temp_sin[i]*pow(2.0, 15.0));
    }
}
```

E Implémentation de la MDCT basée sur la FFT de *FFTW3*

E.1 Header

Header de la classe `mdct_fftw3_f32` : MDCT basée sur la FFT de la librairie *FFTW3* en *float* (32 bits). La classe contient les structures de données `fft_in` et `fft_out`, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`fft_plan`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#include <fftw3.h>

#include "mdct_constants.h"

class fftw3_mdct_f32
{
    private:
    fftwf_plan fft_plan;       // FFT configuration
    fftwf_complex *fft_in;     // FFT input buffer
    fftwf_complex *fft_out;    // FFT output buffer
    float twiddle[MDCT_M];

    public:
    fftw3_mdct_f32();
    ~fftw3_mdct_f32();
    void mdct(float *time_signal, float *spectrum);
    void imdct(float *spectrum, float *time_signal);
};
```

E.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_fftw3_f32` :

- Le tableau de `twiddle` est initialisé en *float* sur 32 bits ;
- La FFT de *FFTW3* est initialisée en une dimension (pour l'audio) avec la taille de la FFT réduite à un quart de la taille de la fenêtre d'entrée par le pre-processing et avec l'option `FFTW_MEASURE` plus lente à l'initialisation mais qui permet d'optimiser le temps d'exécution de la FFT ;
- Les tableaux contenant les données d'entrée (`fft_in`) et de sortie (`fft_out`) de la FFT sont alloués dynamiquement avec la fonction de *FFTW3* et ils sont passé en paramètre à la configuration de la FFT.

```
#include <cmath>

fftw3_mdct_f32::fftw3_mdct_f32()
{
    float alpha = M_PI / (8.f * MDCT_M);
    float omega = M_PI / MDCT_M;
    float scale = sqrt(sqrt(2.f / MDCT_M));

    for (int i = 0; i < MDCT_M2; ++i)
    {
        float x = omega*i + alpha;
        twiddle[2*i] = scale * cos(x);
        twiddle[2*i+1] = scale * sin(x);
    }
```

```
    fft_in = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
    fft_out = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
    fft_plan = fftwf_plan_dft_1d(MDCT_M2, fft_in, fft_out, FFTW_FORWARD, FFTW_MEASURE);
}
```

E.3 Destructeur

Destructeur de la classe `mdct_fftw3_f32` qui permet de libérer la mémoire allouée aux tableaux d'entrée et de sortie de la FFT et à sa configuration avec les fonctions appropriées fournies par la librairie *FFTW3*.

```
fftw3_mdct_f32::~fftw3_mdct_f32()
{
    fftwf_destroy_plan(fft_plan);
    fftwf_free(fft_in);
    fftwf_free(fft_out);
}
```

E.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *FFTW3* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettent de réduire la fenêtre d'entrée de la FFT ;
- Appel de la fonction FFT de *FFTW3* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettent de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle*.

```
void fftw3_mdct_f32::mdct(float *time_signal, float *spectrum)
{
    float *cos_tw = twiddle;
    float *sin_tw = cos_tw + 1;

    /* odd/even folding and pre-twiddle */
    float *xr = (float *)fft_in;
    float *xi = xr + 1;

    for (int i = 0; i < MDCT_M2; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] + time_signal[MDCT_M32+i];
        float i0 = time_signal[MDCT_M2+i] - time_signal[MDCT_M2-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        xr[i] = r0*c + i0*s;
        xi[i] = i0*c - r0*s;
    }

    for(int i = MDCT_M2; i < MDCT_M; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] - time_signal[-MDCT_M2+i];
        float i0 = time_signal[MDCT_M2+i] + time_signal[MDCT_M52-1-i];

        float c = cos_tw[i];
```

```
        float s = sin_tw[i];

        xr[i] = r0*c + i0*s;
        xi[i] = i0*c - r0*s;
    }

    /* complex FFT of size MDCT_M2 */
    fftwf_execute(fft_plan);

    /* post-twiddle */
    xr = (float *)fft_out;
    xi = xr + 1;

    for (int i = 0; i < MDCT_M; i += 2)
    {
        float r0 = xr[i];
        float i0 = xi[i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        spectrum[i] = -r0*c - i0*s;
        spectrum[MDCT_M-1-i] = -r0*s + i0*c;
    }
}
```

F Validation de la MDCT *FFTW3* en *float 32*

F.1 Code source

Test de la MDCT basée sur la FFT de *FFTW3* avec un signal d'entrée sinusoïdal à 200Hz :

- Génération et affichage d'un signal sinusoïdal à 200Hz ;
- Calcul et affichage du spectre de fréquences de ce signal ;
- Opération inverse de la MDCT et affichage du signal temporel calculé à partir du spectre.

```
#include <iomanip>
#include <iostream>

#include <cstring>

#include "mdct_constants.h"
#include "fftw3_mdct_f32.h"
#include "sin_wave.h"

/**
* @brief MDCT algorithm calling the FFT of the fftw3 library
* Code based on https://www.dsprelated.com/showcode/196.php
*/
int main(void)
{
    float time_in[MDCT_WINDON_LEN];         // input time signal
    sin_float(time_in, MDCT_WINDON_LEN, 0.9, 200.0, 0.0, FS);

    float time_out[MDCT_WINDON_LEN];        // output time signal (generated by the IMDCT)
    memset(time_out, 0, MDCT_WINDON_LEN*sizeof(float));

    float spectrum[MDCT_M];     // frequency spectrum
    memset(spectrum, 0, MDCT_M*sizeof(float));

    fftw3_mdct_f32 fftw3_mdct;
    fftw3_mdct.mdct(time_in, spectrum);
    fftw3_mdct.imdct(spectrum, time_out);

    for (int i = 0; i < MDCT_WINDON_LEN; ++i)
    {
        std::cout << "time_in[" << std::setw(4) << i << "]" << std::setw(12) << time_in[i]
            << "␣|␣time_out[" << std::setw(4) << i << "]" << std::setw(12) << time_out[i]
            << std::endl;
    }
    std::cout << std::endl;

    for (int i = 0; i < MDCT_M; ++i)
    {
        std::cout << "spectrum[" << std::setw(4) << i << "]"
            << std::setw(12) << spectrum[i] << std::endl;
    }
    std::cout << std::endl;

    return 0;
}
```

F.2 Compilation

Commandes CMake permettant de compiler le code d'exemple.

```
# MDCT using the fftw3 library f32
add_executable(fftw3_mdct_f32 test/validation/fftw3_example.cpp
    src/fftw3_mdct_f32.cpp src/sin_wave.cpp)
target_link_libraries(fftw3_mdct_f32 fftw3f)
```

G Implémentation de la MDCT basée sur la FFT de *Ne10* en *floating point*

G.1 Header

Header de la classe `mdct_ne10_f32_c` : MDCT basée sur la FFT de la librairie *Ne10* en *float* (32 bits). La classe contient les structures de données `fft_in` et `fft_out`, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`cfg`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_f32_c
{
    private:
    ne10_fft_cfg_float32_t cfg;                                                    // Ne10 configuration
    ne10_fft_cpx_float32_t fft_in[MDCT_M2] __attribute__((aligned(16)));   // Ne10 FFT input buffer
    ne10_fft_cpx_float32_t fft_out[MDCT_M2] __attribute__((aligned(16)));  // Ne10 FFT output buffer
    float twiddle[MDCT_M] __attribute__((aligned(16)));                            // twiddle factors

    public:
    ne10_mdct_f32_c();
    ~ne10_mdct_f32_c();
    void mdct(float *time_signal, float *spectrum);
};
```

G.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_f32_c` :

- Le tableau de `twiddle` est initialisé en *float* sur 32 bits ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *float 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée.

```
ne10_mdct_f32_c::ne10_mdct_f32_c()
{
    float alpha = M_PI / (8.0 * static_cast<float>(MDCT_M));
    float omega = M_PI / static_cast<float>(MDCT_M);
    float scale = sqrt(sqrt(2.0 / static_cast<float>(MDCT_M)));
    for (int i = 0; i < MDCT_M2; ++i)
    {
        float x = omega * i + alpha;
        twiddle[2*i] = static_cast<float>(scale * cos(x));
        twiddle[2*i+1] = static_cast<float>(scale * sin(x));
    }

    cfg = ne10_fft_alloc_c2c_float32_c(MDCT_M2);
}
```

G.3 Destructeur

Destructeur de la classe `mdct_ne10_f32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_f32_c::~ne10_mdct_f32_c()
{
    ne10_fft_destroy_c2c_float32(cfg);
}
```

G.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *float 32* et en *plain C* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettent de réduire la fenêtre d'entrée de la FFT ;
- Appel de la fonction FFT de *Ne10* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettent de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle*.

```
void ne10_mdct_f32_c::mdct(float *time_signal, float *spectrum)
{
    // pre-twiddling
    float *cos_tw = twiddle;
    float *sin_tw = cos_tw + 1;
    for (int i = 0; i < MDCT_M2; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] + time_signal[MDCT_M32+i];
        float i0 = time_signal[MDCT_M2+i] - time_signal[MDCT_M2-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        fft_in[i/2].r = r0*c + i0*s;
        fft_in[i/2].i = i0*c - r0*s;
    }

    for (int i = MDCT_M2; i < (MDCT_M); i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] - time_signal[-MDCT_M2+i];
        float i0 = time_signal[MDCT_M2+i] + time_signal[MDCT_M52-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        fft_in[i/2].r = r0*c + i0*s;
        fft_in[i/2].i = i0*c - r0*s;
    }

    // FFT
    ne10_fft_c2c_1d_float32_c(fft_out, fft_in, cfg, 0);

    // post-twiddling
    for (int i = 0; i < (MDCT_M); i += 2)
    {
        float r0 = fft_out[i/2].r;
```

```
        float i0 = fft_out[i/2].i;

        float c = cos_tw[i];
        float s = sin_tw[i];

        spectrum[i] = -r0*c - i0*s;
        spectrum[(MDCT_M)-1-i] = -r0*s + i0*c;
    }
}
```

H Mesure des performances des FFT de *Ne10*

H.1 Code source

Code permettant de tester la vitesse d'exécution moyenne de différentes FFT proposées par la librairie *Ne10*. La moyenne est calculée sur 10 000 000 exécutions. Les données d'entrée de la FFT sont générées aléatoirement et sont différentes pour chaque exécution. Les variables de préprocesseur définies à la compilation permettent sur base du même code de mesurer le temps d'exécution moyen avec écart type :

- de la FFT *complex to complex* en *float 32* en *plain C* ou avec les optimisations Neon ;
- de la FFT *complex to complex* en *integer 32* en *plain C* ou avec les optimisations Neon ;
- de la FFT *complex to complex* en *integer 16* en *plain C* ou avec les optimisations Neon.

```
#include <iomanip>
#include <iostream>
#include <limits>

#include <cmath>
#include <cstring>

#include "mdct_constants.h"
#include "Timers.h"
#include "NE10.h"

#ifdef F32          // 32 bits floating point arithmetic

#define INPUT_RANGE             1.8
#define INPUT_DATA              ne10_fft_cpx_float32_t
#define OUTPUT_DATA             ne10_fft_cpx_float32_t
#define FFT_CONFIG              ne10_fft_cfg_float32_t
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_float32

#ifdef NEON
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_float32_neon
#define PERFORM_FFT             ne10_fft_c2c_1d_float32_neon
#else
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_float32_c
#define PERFORM_FFT             ne10_fft_c2c_1d_float32_c
#endif

#elif I32           // 32 bits fixed point arithmetic

#define INPUT_RANGE             std::numeric_limits<int16_t>::max()*2
#define INPUT_DATA              ne10_fft_cpx_int32_t
#define OUTPUT_DATA             ne10_fft_cpx_int32_t
#define FFT_CONFIG              ne10_fft_cfg_int32_t
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_int32

#ifdef NEON
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int32_neon
#define PERFORM_FFT             ne10_fft_c2c_1d_int32_neon
#else
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int32_c
#define PERFORM_FFT             ne10_fft_c2c_1d_int32_c
#endif
```

```
#else                // 16 bits fixed point arithmetic

#define INPUT_RANGE              std::numeric_limits<int16_t>::max()*2
#define INPUT_DATA               ne10_fft_cpx_int16_t
#define OUTPUT_DATA              ne10_fft_cpx_int16_t
#define FFT_CONFIG               ne10_fft_cfg_int16_t
#define ALLOC_CONFIG             ne10_fft_alloc_c2c_int16
#define DESTROY_CONFIG           ne10_fft_destroy_c2c_int16

#ifdef NEON
#define PERFORM_FFT              ne10_fft_c2c_1d_int16_neon
#else
#define PERFORM_FFT              ne10_fft_c2c_1d_int16_c
#endif

#endif

#define RUNS             10000000
#define FFT_SCALE_FLAG   0

int main()
{
    // print which FFT will be tested
#ifdef F32
#ifdef NEON
    std::cout << "FFT Ne10 f32 Neon" << std::endl;
#else
    std::cout << "FFT Ne10 f32 plain C" << std::endl;
#endif

#elif I32
#ifdef NEON
    std::cout << "FFT Ne10 i32 Neon" << std::endl;
#else
    std::cout << "FFT Ne10 i32 plain C" << std::endl;
#endif
#else
#ifdef NEON
    std::cout << "FFT Ne10 i16 Neon" << std::endl;
#else
    std::cout << "FFT Ne10 i16 plain C" << std::endl;
#endif
#endif

    // seed the random
    srand(static_cast<unsigned>(time(0)));

    // initialize the configuration
    FFT_CONFIG cfg = ALLOC_CONFIG(MDCT_M2);

    // start the loop executing the FFTs
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        // initialize an empty spectrum
        OUTPUT_DATA spectrum[MDCT_M2] __attribute__((aligned(16)));
        memset(&spectrum, 0, (MDCT_M2)*sizeof(OUTPUT_DATA));
```

```
        // generate random input data
        INPUT_DATA time_signal[MDCT_M2] __attribute__((aligned(16)));
        for (int i = 0; i < MDCT_M2; ++i)
        {
            time_signal[i].r = INPUT_RANGE * rand() / RAND_MAX - INPUT_RANGE / 2;
            time_signal[i].i = INPUT_RANGE * rand() / RAND_MAX - INPUT_RANGE / 2;
        }

        // perfom the FFT and measure the run time
        EvsHwLGPL::CTimers timer;
        timer.Start();
#ifdef F32
        PERFORM_FFT(time_signal, spectrum, cfg, 0);
#else
        PERFORM_FFT(time_signal, spectrum, cfg, 0, FFT_SCALE_FLAG);
#endif
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }

    // clean
    DESTROY_CONFIG(cfg);

    // compute the average
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time:    " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << " ns" << std::endl;

    return 0;
}
```

H.2 Compilation

Commandes CMake utilisées pour générer les exécutables permettant de mesurer le temps d'exécution de différentes FFT proposées par la librairie *Ne10*. En fonction des variables de préprocesseur définies, les exéctables suivants sont générés :

- `run_fft_f32_c` est généré si la variable `F32` est définie pour mesurer le temps d'exécution de la FFT *float 32 plain C* ;
- `run_fft_i32_c` est généré si la variable `I32` est définie pour mesurer le temps d'exécution de la FFT *integer 32 plain C* ;
- `run_fft_i16_c` est généré par défaut pour mesurer le temps d'exécution de la FFT *integer 16 plain C* ;
- `run_fft_f32_neon` est généré si les variables `F32` et `NEON` sont définies pour mesurer le temps d'exécution de la FFT *float 32* avec optimisations *Neon* ;
- `run_fft_i32_neon` est généré si les variables `I32` et `NEON` sont définies pour mesurer le temps d'exécution de la FFT *integer 32* avec optimisations *Neon* ;

- `run_fft_i16_neon` est généré si la variable NEON est définie pour mesurer le temps d'exécution de la FFT *integer 16* avec optimisations *Neon*.

```
# Ne10 FFT performance (float32 plain C)
add_executable(run_ne10_fft_f32_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_f32_c PUBLIC -DF32)
target_link_libraries(run_ne10_fft_f32_c NE10)

# Ne10 FFT performance (int32 plain C)
add_executable(run_ne10_fft_i32_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i32_c PUBLIC -DI32)
target_link_libraries(run_ne10_fft_i32_c NE10)

# Ne10 FFT performance (int16 plain C)
add_executable(run_ne10_fft_i16_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i16_c PUBLIC -DI16)
target_link_libraries(run_ne10_fft_i16_c NE10)

# Ne10 FFT performance (float32 with neon optimizations)
add_executable(run_ne10_fft_f32_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_f32_neon PUBLIC -DF32 -DNEON)
target_link_libraries(run_ne10_fft_f32_neon NE10)

# Ne10 FFT performance (int32 with neon optimizations)
add_executable(run_ne10_fft_i32_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i32_neon PUBLIC -DI32 -DNEON)
target_link_libraries(run_ne10_fft_i32_neon NE10)

# Ne10 FFT performance (int16 with neon optimizations)
add_executable(run_ne10_fft_i16_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i16_neon PUBLIC -DI16 -DNEON)
target_link_libraries(run_ne10_fft_i16_neon NE10)
```

I Mesure des performances des FFT de *FFTW3*

I.1 Code source

Code permettant de tester la vitesse d'exécution moyenne de la FFT en *float 32* de la librairie *FFTW3*. La moyenne est calculée sur 10 000 000 exécutions. Les données d'entrée de la FFT sont générées aléatoirement et sont différentes pour chaque exécution.

```
#include <iomanip>
#include <iostream>

#include <cmath>

#include <fftw3.h>

#include "mdct_constants.h"
#include "Timers.h"

#define RUNS 10000000

int main()
{
    // print which FFT will be tested
    std::cout << "FFT FFTW3 f32 plain C" << std::endl;

    // seed the random
    srand(static_cast<unsigned>(time(0)));

    // start the loop executing the FFTs
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        // initialize an empty spectrum
        fftwf_complex *fft_out = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);

        // generate random input data
        fftwf_complex *fft_in = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
        float *x = (float *)fft_in;
        for (int i = 0; i < MDCT_M2; ++i)
        {
            x[i] = 1.8f * rand() / RAND_MAX - 1.8f / 2.0f;
        }

        // initialize the configuration
        fftwf_plan fft_plan = fftwf_plan_dft_1d(MDCT_M2, fft_in, fft_out,
            FFTW_FORWARD, FFTW_MEASURE);

        // perfom the FFT and measure the run time
        EvsHwLGPL::CTimers timer;
        timer.Start();
        fftwf_execute(fft_plan);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();

        // clean
        fftwf_destroy_plan(fft_plan);
        fftwf_free(fft_in);
        fftwf_free(fft_out);
```

```
    }

    // compute the average
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time:    " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << " ns" << std::endl;

    return 0;
}
```

I.2 Compilation

Commandes CMake utilisées pour générer l'exécutable permettant de mesurer le temps d'exécution de la FFT *float 32* de la librairie *FFTW3*.

```
# FFTW3 FFT performance (float32)
add_executable(run_fftw3_fft_f32 test/performance/run_fftw3_fft_f32.cpp src/Timers.cpp)
target_link_libraries(run_fftw3_fft_f32 fftw3f)
```

J Implémentation de la MDCT basée sur la FFT de *Ne10* en *fixed point*

J.1 Header

Header de la classe `mdct_ne10_i32_c` : MDCT basée sur la FFT de la librairie *Ne10* en *integer* (32 bits). La classe contient les structures de données `fft_in` en représentation Q1.15 et `fft_out` en Q9.15, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`cfg`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_i32_c
{
    private:
    ne10_fft_cfg_int32_t cfg;                                          // Ne10 configuration
    ne10_fft_cpx_int32_t fft_in[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT input buffer
                                                                       //   Q1.15
    ne10_fft_cpx_int32_t fft_out[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT output buffer
                                                                       //   Q9.15
    int16_t twiddle[MDCT_M] __attribute__((aligned(16)));              // MDCT twiddle factors

    public:
    ne10_mdct_i32_c();
    ~ne10_mdct_i32_c();
    void mdct(int16_t *time_signal, int32_t *spectrum);
};
```

J.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_i32_c` :

- Le tableau de `twiddle` est initialisé en *double* puis converti en *integer* (représentation Q15) ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée.

```
ne10_mdct_i32_c::ne10_mdct_i32_c()
{
    // intialize the twiddling factors
    double alpha = M_PI / (8.0*MDCT_M);
    double omega = M_PI / MDCT_M;
    double scale = sqrt(sqrt(2.0 / static_cast<double>MDCT_M));
    for (int i = 0; i < MDCT_M2; ++i)
    {
        double x = omega * i + alpha;
        twiddle[2*i] = static_cast<int16_t>(cos(x)*scale*pow(2.0, 15.0));
        twiddle[2*i+1] = static_cast<int16_t>(sin(x)*scale*pow(2.0, 15.0));
    }

    // initialize the Ne10 FFT configuration
    cfg = ne10_fft_alloc_c2c_int32_c(MDCT_M2);
}
```

J.3 Destructeur

Destructeur de la classe `mdct_ne10_i32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_i32_c::~ne10_mdct_i32_c()
{
    ne10_fft_destroy_c2c_int32(cfg);
}
```

J.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *integer 32* et en *plain C* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettant de réduire la fenêtre d'entrée de la FFT sont faites en algorithmique *fixed point* ;
- Appel de la fonction FFT de *Ne10* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettant de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle* sont faites en algorithmique *fixed point*.

```
void ne10_mdct_i32_c::mdct(int16_t *time_signal, int32_t *spectrum)
{
    // pre-twiddling
    // fft_in = (Q1.15 + Q1.15) * Q1.15/4 + (Q1.15 + Q1.15) * Q1.15/4
    //          1/4 Q1.30 + 1/4 Q1.30 + 1/4 Q1.30 + 1/4 Q1.30 -> Q1.30
    //          >>7 -> Q1.23 + 8 bits reserved for the FFT
    int16_t *cos_tw = twiddle;
    int16_t *sin_tw = cos_tw + 1;
    for (int i = 0; i < MDCT_M2; i += 2)
    {
        int32_t r0 = static_cast<int32_t>(time_signal[MDCT_M32-1-i]) + time_signal[MDCT_M32+i];
        int32_t i0 = static_cast<int32_t>(time_signal[MDCT_M2+i]) - time_signal[MDCT_M2-1-i];

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        fft_in[i/2].r = (((r0*c)+64)>>7) + (((i0*s)+64)>>7);
        fft_in[i/2].i = (((i0*c)+64)>>7) - (((r0*s)+64)>>7);
    }

    for (int i = MDCT_M2; i < MDCT_M; i += 2)
    {
        int32_t r0 = static_cast<int32_t>(time_signal[MDCT_M32-1-i]) - time_signal[-MDCT_M2+i];
        int32_t i0 = static_cast<int32_t>(time_signal[MDCT_M2+i]) + time_signal[MDCT_M52-1-i];

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        fft_in[i/2].r = (((r0*c)+64)>>7) + (((i0*s)+64)>>7);
        fft_in[i/2].i = (((i0*c)+64)>>7) - (((r0*s)+64)>>7);
    }

    // perform the FFT
    ne10_fft_c2c_1d_int32_c(fft_out, fft_in, cfg, 0, 0);
```

```
    // post-twiddling
    // spectrum = Q9.23>>8 * Q1.15/4 + Q9.23>>8 * Q1.15/4
    //          = Q9.15 * Q1.15/4 + Q9.15 * Q1.15/4
    //          = Q10.30/4 + Q10.30/4
    //          = Q11.30/4
    //          = Q9.30 >> 15 = Q9.15
    for (int i = 0; i < MDCT_M; i += 2)
    {
        int32_t r0 = fft_out[i/2].r;
        int32_t i0 = fft_out[i/2].i;

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        spectrum[i] = ((((-r0+128)>>8)*c+16384)>>15) - ((((i0+128)>>8)*s+16384)>>15);
        spectrum[MDCT_M-1-i] = ((((-r0+128)>>8)*s+16384)>>15) + ((((i0+128)>>8)*c+16384)>>15);
    }
}
```

K Implémentation de la MDCT basée sur la FFT de *Ne10* en *fixed point* avec optimisations *Neon*

K.1 Header

Header de la classe `mdct_ne10_i32_neon` : MDCT basée sur la FFT de la librairie *Ne10* en *integer* (32 bits) optimisée par l'utilisation des opérations SIMD Neon. La classe contient les structures de données `fft_in` en représentation Q1.15 et `fft_out` en Q9.15, les tableaux de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`cfg`). Contrairement aux autres implémentations, les facteurs de twiddle ne sont pas rassemblés dans un seul tableau. Les tableaux de facteurs de *pre-twiddling* et de *post-twiddling* sont séparés car ils sont utilisés en 16 bits pour le *pre-twiddling* et en 32 bits pour le *post-twiddling*. Chacun de ses tableaux est séparé en deux car pour que chaque moitié puisse être initialisée dans un ordre qui facilite l'utilisation des opérations SIMD. L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include <arm_neon.h>
#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_i32_neon
{
private:
    ne10_fft_cfg_int32_t cfg;                                          // Ne10 configuration
    ne10_fft_cpx_int32_t fft_in[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT input buffer
    ne10_fft_cpx_int32_t fft_out[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT output buffer
    int16_t pretwiddle_start[MDCT_M2] __attribute__((aligned(16)));    // pre-twiddle factors
    int16_t pretwiddle_end[MDCT_M2] __attribute__((aligned(16)));      // second half is stored
                                                                       //   in reversed order
    int32_t posttwiddle_start[MDCT_M2] __attribute__((aligned(16)));   // post-twiddle factors
    int32_t posttwiddle_end[MDCT_M2] __attribute__((aligned(16)));     // second half is stored
                                                                       //   in reversed order

public:
    ne10_mdct_i32_neon();
    ~ne10_mdct_i32_neon();
    void mdct(int16_t *time_signal, int32_t *spectrum);
};
```

K.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_i32_neon` :

- Le tableau de `twiddle` est initialisé en *double* puis converti en *integer* (représentation Q15 en 16 bits pour le *pre-twiddling* et en 32 bits pour le *post-twiddling*) : la première moitié des tableaux est rangée à l'endroit dans les tableaux `pretwiddle_start` et `posttwiddle_start` tandis que la seconde est rangée à l'envers dans les tableaux `pretwiddle_end` et `posttwiddle_end` ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée avec la fonction adaptée pour l'exécution d'une FFT optimisée avec les instructions SIMD Neon.

```
ne10_mdct_i32_neon::ne10_mdct_i32_neon()
{
    double alpha = M_PI / (8.0*MDCT_M);
    double omega = M_PI / MDCT_M;
    double scale = sqrt(sqrt(2.0 / static_cast<double>MDCT_M));
    for (int i = 0; i < MDCT_M4; ++i)
    {
        double start = omega * (i) + alpha;
        double end = omega * (i+MDCT_M4) + alpha;
        double cos_start = cos(start);
        double sin_start = sin(start);
        double cos_end = cos(end);
        double sin_end = sin(end);
        pretwiddle_start[2*i] = static_cast<int16_t>(cos_start*scale*pow(2.0, 15.0));
        pretwiddle_start[2*i+1] = static_cast<int16_t>(sin_start*scale*pow(2.0, 15.0));
        pretwiddle_end[MDCT_M2-2*i-2] = static_cast<int16_t>(cos_end*scale*pow(2.0, 15.0));
        pretwiddle_end[MDCT_M2-2*i-1] = static_cast<int16_t>(sin_end*scale*pow(2.0, 15.0));
        posttwiddle_start[2*i] = static_cast<int32_t>(cos_start*scale*pow(2.0, 31.0));
        posttwiddle_start[2*i+1] = static_cast<int32_t>(sin_start*scale*pow(2.0, 31.0));
        posttwiddle_end[MDCT_M2-2*i-2] = static_cast<int32_t>(cos_end*scale*pow(2.0, 31.0));
        posttwiddle_end[MDCT_M2-2*i-1] = static_cast<int32_t>(sin_end*scale*pow(2.0, 31.0));
    }

    cfg = ne10_fft_alloc_c2c_int32_neon(MDCT_M2);
}
```

K.3 Destructeur

Destructeur de la classe `mdct_ne10_i32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_i32_neon::~ne10_mdct_i32_neon()
{
    ne10_fft_destroy_c2c_int32(cfg);
}
```

K.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *integer 32* avec utilisation des instructions SIMD Neon :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettant de réduire la fenêtre d'entrée de la FFT sont faites en algorithmique *fixed point*. L'utilisation des fonctions SIMD permet d'effectuer quatre opérations en parallèle afin de réduire le temps d'exécution. Les facteurs de *pre-twiddling* en 16 bits transforment le signal d'entrée en *16 bits* en un tableau d'entrée de la FFT en 32 bits ;
- Appel de la fonction FFT de *Ne10* optimisée par l'utilisation des instructions SIMD Neon ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettant de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle* sont faites en algorithmique *fixed point*. Les fonctions SIMD permettent d'effectuer deux ou quatre opérations en parallèle afin de réduire le temps d'exécution.

```
void ne10_mdct_i32_neon::mdct(int16_t *time_signal, int32_t *spectrum)
{
    // see the twiddling_loops.nlsx file for more details

    // for i from 0 to 254, step 2

    // r[  0 -> 127, pas 1] =  time_signal[ 767 ->  513, pas 2] * c[  0 -> 127, pas 1]
    //                       +  time_signal[ 768 -> 1022, pas 2] * c[  0 -> 127, pas 1]
    //                       +  time_signal[ 256 ->  510, pas 2] * s[  0 -> 127, pas 1]
    //                       + -time_signal[ 255 ->    1, pas 2] * s[  0 -> 127, pas 1]

    // i[  0 -> 127, pas 1] =  time_signal[ 256 ->  510, pas 2] * c[  0 -> 127, pas 1]
    //                       + -time_signal[ 255 ->    1, pas 2] * c[  0 -> 127, pas 1]
    //                       + -time_signal[ 767 ->  513, pas 2] * s[  0 -> 127, pas 1]
    //                       + -time_signal[ 768 -> 1022, pas 2] * s[  0 -> 127, pas 1]

    // fft_in[i/2].r = time_signal[M32-1-i]    * cos_tw[i] + time_signal[M32+i]     * cos_tw[i]
    //               + time_signal[M2+i]       * sin_tw[i] + (-time_signal[M2-1-i]) * sin_tw[i]

    // fft_in[i/2].i = time_signal[M2+i]       * cos_tw[i] + (-time_signal[M2-1-i]) * cos_tw[i]
    //               + (-time_signal[M32-1-i]) * sin_tw[i] + (-time_signal[M32+i])  * sin_tw[i]


    // for i from 510 to 256, step 2

    // r[255 -> 128, pas 1] =  time_signal[ 257 ->  511, pas 2] * c[255 -> 128, pas 1]
    //                       + -time_signal[ 254 ->    0, pas 2] * c[255 -> 128, pas 1]
    //                       +  time_signal[ 766 ->  512, pas 2] * s[255 -> 128, pas 1]
    //                       +  time_signal[ 769 -> 1023, pas 2] * s[255 -> 128, pas 1]

    // i[255 -> 128, pas 1] =  time_signal[ 766 ->  512, pas 2] * c[255 -> 128, pas 1]
    //                       +  time_signal[ 769 -> 1023, pas 2] * c[255 -> 128, pas 1]
    //                       + -time_signal[ 257 ->  511, pas 2] * s[255 -> 128, pas 1]
    //                       +  time_signal[ 254 ->    0, pas 2] * s[255 -> 128, pas 1]

    // fft_in[i/2].r = time_signal[M32-1-i]    * cos_tw[i] + (-time_signal[-M2+i]) * cos_tw[i]
    //               + time_signal[M2+i]       * sin_tw[i] + time_signal[M52-1-i]  * sin_tw[i]

    // fft_in[i/2].i = time_signal[M2+i]       * cos_tw[i] + time_signal[M52-1-i] * cos_tw[i]
    //               + (-time_signal[M32-1-i]) * sin_tw[i] + time_signal[-M2+i]   * sin_tw[i]

    for (int i = 0; i < MDCT_M2; i += 8)
    {
        // tx.val[0] -> odd indexes
        // tx.val[1] -> even indexes
        int16x4x2_t t1 = vld2_s16(time_signal+MDCT_M2+i);
        int16x4x2_t t2 = vld2_s16(time_signal+MDCT_M2-8-i);
        int16x4x2_t t3 = vld2_s16(time_signal+MDCT_M32+i);
        int16x4x2_t t4 = vld2_s16(time_signal+MDCT_M32-8-i);

        t2.val[0] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t2.val[0]));
                                            // reverse the t2 even values: 0 2 4 6 -> 6 4 2 0
        t2.val[1] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t2.val[1]));
                                            // reverse the t2 odd values : 1 3 5 7 -> 7 5 3 1
        t4.val[0] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t4.val[0]));
                                            // reverse the t4 even values: 0 2 4 6 -> 6 4 2 0
        t4.val[1] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t4.val[1]));
                                            // reverse the t4 odd values : 1 3 5 7 -> 7 5 3 1
```

```
        // x_tw.val[0] -> cos twiddle
        // x_tw.val[1] -> sin twiddle
        int16x4x2_t start_tw = vld2_s16(pretwiddle_start+i);
        int16x4x2_t end_tw = vld2_s16(pretwiddle_end+i);

        // start.val[0] -> real part
        // start.val[1] -> imaginary part
        int32x4x2_t start;
        start.val[0] = vshrq_n_s32(
            vaddq_s32(
                vaddq_s32
                    vmull_s16(t4.val[1], start_tw.val[0]),
                    vmull_s16(t3.val[0], start_tw.val[0])),
                vaddq_s32
                    vmull_s16(t1.val[0], start_tw.val[1]),
                    vmull_s16(vneg_s16(t2.val[1]), start_tw.val[1])))
                , 7);
        start.val[1] = vshrq_n_s32
            vaddq_s32(
                vaddq_s32
                    vmull_s16(t1.val[0], start_tw.val[0]),
                    vmull_s16(vneg_s16(t2.val[1]), start_tw.val[0])),
                vaddq_s32
                    vmull_s16(vneg_s16(t4.val[1]), start_tw.val[1]),
                    vmull_s16(vneg_s16(t3.val[0]), start_tw.val[1])))
                , 7);

        // end.val[0] -> real part
        // end.val[1] -> imaginary part
        int32x4x2_t end;
        end.val[0] = vshrq_n_s32
            vaddq_s32(
                vaddq_s32(vmull_s16(t1.val[1], end_tw.val[0]),
                vmull_s16(vneg_s16(t2.val[0]), end_tw.val[0])),
            vaddq_s32
                vmull_s16(t4.val[0], end_tw.val[1]),
                vmull_s16(t3.val[1], end_tw.val[1])))
            , 7);
        end.val[1] = vshrq_n_s32
            vaddq_s32(
                vaddq_s32
                    vmull_s16(t4.val[0], end_tw.val[0]),
                    vmull_s16(t3.val[1], end_tw.val[0])),
            vaddq_s32
                vmull_s16(vneg_s16(t1.val[1]), end_tw.val[1]),
                vmull_s16(t2.val[0], end_tw.val[1])))
            , 7);

        // reverse the end part
        end.val[0] = (int32x4_t)vrev64q_s32(end.val[0]);
        end.val[0] = vcombine_s32(vget_high_s32(end.val[0]), vget_low_s32(end.val[0]));
        end.val[1] = (int32x4_t)vrev64q_s32(end.val[1]);
        end.val[1] = vcombine_s32(vget_high_s32(end.val[1]), vget_low_s32(end.val[1]));

        // store the result
        vst2q_s32((int32_t *)fft_in+i, start);
        vst2q_s32((int32_t *)(fft_in+MDCT_M2-4-i/2), end);
    }
```

```
    // perform the FFT
    ne10_fft_c2c_1d_int32_neon(fft_out, fft_in, cfg, 0, 0);

    // post-twiddling
    for (int i = 0; i < MDCT_M2; i += 8)
    {
        // load the fft ouput and reverse the end part
        // fft_out_x.val[0] -> real part
        // fft_out_x.val[1] -> imaginary part
        int32x4x2_t fft_out_start = vld2q_s32((int32_t *)fft_out+i);
        int32x4x2_t fft_out_end = vld2q_s32((int32_t *)fft_out+MDCT_M-8-i);
        fft_out_end.val[0] = (int32x4_t)vrev64q_s32(fft_out_end.val[0]);
        fft_out_end.val[0] = vcombine_s32(vget_high_s32(fft_out_end.val[0]),
                                          vget_low_s32(fft_out_end.val[0]));
        fft_out_end.val[1] = (int32x4_t)vrev64q_s32(fft_out_end.val[1]);
        fft_out_end.val[1] = vcombine_s32(vget_high_s32(fft_out_end.val[1]),
                                          vget_low_s32(fft_out_end.val[1]));

        // load the twiddle factors
        // x_tw.val[0] -> cos twiddle
        // x_tw.val[1] -> sin twiddle
        int32x4x2_t start_tw = vld2q_s32(posttwiddle_start+i);
        int32x4x2_t end_tw = vld2q_s32(posttwiddle_end+i);

        int32x4x2_t spectrum_start;
        spectrum_start.val[0] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[0]), start_tw.val[0]),
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[1]), start_tw.val[1])), 8);
        spectrum_start.val[1] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[0]), end_tw.val[1]),
            vqrdmulhq_s32(fft_out_end.val[1], end_tw.val[0])), 8);

        int32x4x2_t spectrum_end;
        spectrum_end.val[0] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[0]), end_tw.val[0]),
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[1]), end_tw.val[1])), 8);
        spectrum_end.val[1] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[0]), start_tw.val[1]),
            vqrdmulhq_s32(fft_out_start.val[1], start_tw.val[0])), 8);

        spectrum_end.val[0] = (int32x4_t)vrev64q_s32(spectrum_end.val[0]);
        spectrum_end.val[0] = vcombine_s32(vget_high_s32(spectrum_end.val[0]),
                                           vget_low_s32(spectrum_end.val[0]));
        spectrum_end.val[1] = (int32x4_t)vrev64q_s32(spectrum_end.val[1]);
        spectrum_end.val[1] = vcombine_s32(vget_high_s32(spectrum_end.val[1]),
                                           vget_low_s32(spectrum_end.val[1]));

        // store the result
        vst2q_s32((int32_t *)spectrum+i, spectrum_start);
        vst2q_s32((int32_t *)(spectrum+MDCT_M-8-i), spectrum_end);
    }
}
```

L

L.1 Code source

L.2 Code source

M Mesure des performances des MDCT *Ne10* et MDCT de référence

M.1 Code source

Code permettant de mesurer les performances des MDCT de *Ne10* en *float 32 plain C*, *integer 32 plain C*, *integer 32* avec optimisation Neon et de la MDCT de référence en *double* (en fonction de la variable de préprocesseur définie à la compilation). Le code mesure et affiche le temps d'exécution moyen et l'écart type. Ces informations sont calculées sur un nombre d'exécution donné en paramètre à l'exécutable. Les MDCT sont testées avec le même signal sinusoïdal en entrée dont la valeur par défaut est de 200Hz.

```
#include <iostream>
#include <cstring>

#include "args_parser.h"
#include "mdct_constants.h"
#include "sin_wave.h"
#include "Timers.h"


#ifdef FIXED_POINT_C     // fixed point arithmetic

#include "ne10_mdct_i32_c.h"

#define INPUT_DATA               int16_t
#define OUTPUT_DATA              int32_t
#define GENERATE_SIN             sin_int
#define MDCT                     ne10_mdct_i32_c

#elif FIXED_POINT_NEON   // fixed point arithmetic

#include "ne10_mdct_i32_neon.h"

#define INPUT_DATA               int16_t
#define OUTPUT_DATA              int32_t
#define GENERATE_SIN             sin_int
#define MDCT                     ne10_mdct_i32_neon

#elif FLOATING_POINT     // floating point arithmetic

#include "ne10_mdct_f32_c.h"

#define INPUT_DATA               float
#define OUTPUT_DATA              float
#define GENERATE_SIN             sin_float
#define MDCT                     ne10_mdct_f32_c

#else                    // reference algorithm in floating point arithmetic

#include "ref_mdct.h"

#define INPUT_DATA               double
#define OUTPUT_DATA              double
#define GENERATE_SIN             sin_float

#endif
```

```
/**
 * @brief Run the MDCT on a single frame x times
 * the signal is a single tone configurable via the --sin parameter (200Hz by default)
 * the number of runs is setted by the --run parameter (1 by default)
 */
int main(int argc, char **argv)
{
    // initialize the parameters
    params p;
    try
    {
        p = parse_args(argc, argv);
    }
    catch(const std::runtime_error &err)
    {
        std::cerr << err.what() << std::endl;
        usage();
        return 1;
    }

    // print which MDCT will be tested
#ifdef FIXED_POINT_C
    std::cout << "MDCT Ne10 i32 plain C: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#elif FIXED_POINT_NEON
    std::cout << "MDCT Ne10 i32 Neon: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#elif FLOATING_POINT
    std::cout << "MDCT Ne10 f32 plain C: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#else
    std::cout << "Reference MDCT: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#endif

    // generate the time signal
    INPUT_DATA time_signal[MDCT_WINDON_LEN] __attribute__((aligned(16)));
    memset(&time_signal, 0, MDCT_WINDON_LEN*sizeof(INPUT_DATA));
    GENERATE_SIN(time_signal, MDCT_WINDON_LEN, 0.9, p.frequency, 0.0, FS);

    // initialize an empty spectrum
    OUTPUT_DATA mdct_spectrum[MDCT_M] __attribute__((aligned(16)));
    memset(&mdct_spectrum, 0, MDCT_M*sizeof(OUTPUT_DATA));

    // perform the MDCT x times
    EvsHwLGPL::CTimers timer;
    int64_t *runtimes = static_cast<int64_t *>(malloc(p.runs * sizeof(int64_t)));

#ifdef REF
    for (unsigned i = 0; i < p.runs; ++i) {
        timer.Start();
        ref_float_mdct<INPUT_DATA>(time_signal, mdct_spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }
```

```
#else
    MDCT ne10_mdct;
    for (unsigned i = 0; i < p.runs; ++i) {
        timer.Start();
        ne10_mdct.mdct(time_signal, mdct_spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }
#endif

    // compute the average run time
    double avg = 0.0;
    for (unsigned i = 0; i < p.runs; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(p.runs);
    std::cout << "average run time: " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (unsigned i = 0; i < p.runs; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(p.runs);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << std::endl;

    // clean
    free(runtimes);

    return 0;
}
```

M.2 Compilation

Commandes CMake pour la compilation des différents exécutables de tests de performance des MDCT :

- la variable `FLOATING_POINT` permet de compiler l'exécutable `run_mdct_f32_c` pour tester la MDCT basée sur la FFT de *Ne10* en *float 32 plain C* ;
- la variable `FIXED_POINT_C` permet de compiler l'exécutable `run_mdct_i32_c` pour tester la MDCT basée sur la FFT de *Ne10* en *integer 32 plain C* ;
- la variable `FIXED_POINT_NEON` permet de compiler l'exécutable `run_mdct_i32_neon` pour tester la MDCT basée sur la FFT de *Ne10* en *integer 32* avec optimisations Neon ;
- la variable `REF` permet de compiler l'exécutable `run_mdct_ref` pour tester la MDCT de référence en *double* ;

```
# Ne10 f32 x times on a single frame
add_executable(run_ne10_mdct_f32_c test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_f32_c.cpp)
target_compile_definitions(run_ne10_mdct_f32_c PUBLIC -DFLOATING_POINT)
target_link_libraries(run_ne10_mdct_f32_c NE10)

# Ne10 i32 plain C x times on a single frame
add_executable(run_ne10_mdct_i32_c test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_i32_c.cpp)
target_compile_definitions(run_ne10_mdct_i32_c PUBLIC -DFIXED_POINT_C)
target_link_libraries(run_ne10_mdct_i32_c NE10)
```

```
# Ne10 i32 neon x times on a single frame
add_executable(run_ne10_mdct_i32_neon test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_i32_neon.cpp)
target_compile_definitions(run_ne10_mdct_i32_neon PUBLIC -DFIXED_POINT_NEON)
target_link_libraries(run_ne10_mdct_i32_neon NE10)

# Ref MDCT x times on a single frame
add_executable(run_ref_mdct test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ref_mdct.cpp)
target_compile_definitions(run_ref_mdct PUBLIC -DREF)
target_link_libraries(run_ref_mdct NE10)
```

N Mesure des performances de la MDCT *FFTW3* en *float 32*

N.1 Code source

Code permettant de mesurer les performances de la MDCT basée sur la FFT de *FFTW3* en *float 32*. Le code mesure et affiche le temps d'exécution moyen et l'écart type. Ces informations sont calculées sur 10 000 000 exécutions. La MDCT est testée avec le même signal sinusoïdal en entrée dont la valeur est définie à 440Hz.

```
#include <iostream>
#include <cstring>

#include "fftw3_mdct_f32.h"
#include "sin_wave.h"
#include "Timers.h"

#define RUNS        10000000
#define FREQUENCY   440.0

int main()
{
    // print which MDCT will be tested
    std::cout << "MDCT␣FFTW3␣f32␣plain␣C:␣"
        << RUNS << "␣runs␣with␣a␣single␣tone␣signal␣(" << FREQUENCY << "Hz)" << std::endl;

    // generate the time signal
    float time_signal[MDCT_WINDON_LEN];
    sin_float(time_signal, MDCT_WINDON_LEN, 0.9, FREQUENCY, 0.0, FS);

    // initialize an empty spectrum
    float spectrum[MDCT_M];
    memset(spectrum, 0, MDCT_M*sizeof(float));

    // initialize the configuration
    fftw3_mdct_f32 fftw3_mdct;

    // perform the MDCT
    EvsHwLGPL::CTimers timer;
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        timer.Start();
        fftw3_mdct.mdct(time_signal, spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }

    // compute the average run time
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average␣run␣time:␣" << avg << "␣ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard␣deviation:␣" << dev << std::endl;
```

```
    // clean
    free(runtimes);

    return 0;
}
```

N.2 Compilation

Commandes CMake pour la compilation de l'exécutable de tests de performance de la MDCT basée sur la FFT de *FFTW3* en *float 32*.

```
# FFTW3 f32
add_executable(run_fftw3_mdct_f32 test/performance/run_fftw3_mdct_f32.cpp
    src/fftw3_mdct_f32.cpp src/sin_wave.cpp src/Timers.cpp)
target_link_libraries(run_fftw3_mdct_f32 fftw3f)
```