Rapport final : EVEEX

Encodage Vidéo ENSTA Bretagne Expérimental

A. Froehlich G. Leinen J.N. Clink H. Saad
H. Questroy

05-01-2021

Informations

Membres du groupe : Guillaume Leinen, Jean-Noël Clink, Hussein Saad, Alexandre Froehlich, Hugo Questroy

Encadrants : Pascal Cotret, Jean-Christophe Le Lann, Joël Champeau

Code disponible sur GitHub ==> https://github.com/EVEEX-Project

Eportfolio Mahara disponible => https://mahara.ensta-bretagne.fr/view/group views.php?group=348

Abstract

Résumé

Remerciements

— Nos encadrants **Pascal Cotret**, **Jean-Christophe Le Lann** et **Joël Champeau**, qui nous aident à définir les objectifs à atteindre, nous prêtent le matériel nécessaire en particulier les cartes FPGA, ainsi qu'a résoudre des problèmes théoriques.
— **Enjoy Digital**, société créée par un Alumni Ensta-Bretagne, et son produit Litex qui nous sera très utile sur l'implémentation hardware.
— Le site **FPGA4students** pour ses tutoriels VHDL/Verilog.
— **Jean-Christophe Leinen** pour ses conseils sur les méthodes Agiles.

Introduction

Dans son rapport annuel sur l'usage d'internet, Cisco met en exergue l'importance du trafic vidéo dans l'internet mondial [1] : la part du streaming dans le débit mondial ne fait qu'augmenter, les définitions de lecture augmentent elles aussi, ainsi que la part de "live" très gourmands en bande passante.

Dans cette perspective, il est clair que l'algorithme de compression utilisée pour compresser un flux vidéo brut à toute son importance, le moindre % de bande passante économisée permettant de libérer plusieurs TB/s de bande passante. Ces codecs sont relativement peu connu du grand public, en voici quelques uns:

— MPEG-4 (H.264) : il s'agit d'un des codecs les plus connu, car il génère des fichiers d'extension .mp4 et est embarqué dans un grand nombre d'appareils. Il est important de savoir que, comme le H.265, ce codec est **protégé** par un brevet, et les services et constructeurs souhaitant utilisé cet algorithme ou un connecteur basé sur l'algorithme doivent reverser des royalties à MPEG-LA [2] (*La fabrication d'un connecteur display-port coutant 0.20$ de license au constructeur*), coalitions de plusieurs entreprises du numériques comme sony, panasonic ou encore l'université de Columbia.

— VPx : appartenant à l'origine à One-technologie, l'entreprise fut racheté par Google à la suite. Les codecs VP (dernière version VP9) sont ouvert et sans royalties. Ce codec est plutôt performant en terme de compression mais son encodage est lent [3], avec sur un core i7 d'intel en 720p une vitesse de compression proche de 2 images/s, occasionnant des coûts non négligeable en puissance informatique pour les entreprises productrices de contenu (comme Netflix).

— H.265 : l'un des codecs les plus récents, il permet une réduction significative de la bande passante nécessaire, notamment pour le streaming, mais est aussi lent à l'encodage, et demande des royalties.

Vous l'aurez constaté, les codecs les plus actuels sont souvent détenus par des entreprises du secteur. Pour limiter les coûts annexes pour les entreprises, un consortium s'est créé en 2015, a but non lucratif, afin de développer un codec libre de droit aussi efficace que les autres : l'**Aliance for Open-Media** [4]. On compte la plupart des acteurs du secteur dans ce consortium, notamment l'arrivée des acteurs du streaming comme Hulu ou Netflix. Leur création, le codec AV1, basé sur VP9, est donc libre de droit, et est notamment très employé dans le streaming vidéo. Il a l'avantage de proposer une compression 30% plus forte que le H265 [5], mais occasionne par les différentes bibliothèques utilisées une utilisation des ressources informatiques (puissance CPU) bien plus importante, aussi bien du coté encodeur que décodeur. La transition vers l'AV1 sur les grandes plateformes vidéos (Netflix, Youtube) n'est pas encore réalisé mais bien en cours de planification.

Ce besoin en ressources CPU devient donc critique et un point économiquement important pour les entreprises du secteur vidéo. A l'heure actuelle, l'architecture

PC (jeu d'instruction x64, x86) reste la plus utilisé dans l'informatique moderne, mais cela pourrait changer d'ici quelques années. En effet, les architectures a but embarqué on fait de gros progrès ces dernières années, au point que même un géant du secteur comme Apple décide de basculer tout ces ordinateurs vers une architecture ARM.

Ces architectures embarqués, plus récentes, possèdent en effet des caractéristiques de consommation et de performances qui les rendent très intéressantes pour le traitement vidéo. Voyons ensemble quelques architectures et technologies actuelles :

Architecture ARM

Ce jeu d'instruction est très utilisé dans les appareils mobiles comme les smartphones ou tablettes. Il a l'avantage de proposer un jeu d'instruction beaucoup plus simple, ce qui permet notamment des **performances** en matières de **consommation d'énergie** très intéressantes en mobilité. En revanche, **l'architecture est**, tout comme les jeux d'instructions pc plus anciens, **sous licence** également (x64 pour AMD, x86 pour Intel). Les *SOC* ARM **embarquent** tout les composants nécessaires au fonctionnement du système (CPU, GPU, DSP, gestions des I/O) sur une seule puce ce qui rend les systèmes compacts. Le jeu d'instructions réduit rend en revanche l**'inter compatibilité** entre x64/86 et architectures de type RISC (Reduced Instruction Set Computer) comme l'ARM ou le riscV.

Architecture RISCV

Le jeu d'instruction RISCV est très proche de l'ARM, qui est aussi un RISC. La différence profonde se situe dans l'adoption de la technologie ARM dans l'écrasante majorité des applications, ce qui implique une faible quantité de code existant pour RISCV. En Revanche, RISCV possède un avantage notable : elle est complètement Open-source et libre de droits.

Processeurs Field Programmable Gate Array (FPGA)

L'architecture FPGA est complément différente des autres cités précédemment. Un processeur ARM ou x86 est gravé et inaltérable dans son fonctionnement, il n'est pas possible de modifier le "hardware" c'est à dire les branchements des registres au sein même de la puce. Avec les FPGA, on gagne cette possibilité et c'est tout l'intérêt. La quasi-totalité du processeur est ainsi "reprogrammable" au niveau matériel. Cela implique beaucoup de choses dont voici une partie :

— Comme on peut reprogrammer les portes logiques qui le constitue, il est possible d'intégrer un **très fort parallélisme** au sein des calculs : les portes logiques peuvent être repartis en **autant d'unités de traitement qu'on le souhaite** pourvu qu'on a assez de silicium. Si un design consomme 500 portes et qu'on en disposent de 5000, on peut tout a fait séparer le calcul en 10 cœurs alors qu'un processeur conventionnel à 4

cœurs sera limité à 4 unités de traitement, et ce peut importe le niveau de charge de ces cœurs.
- Un code implémenté sur FPGA est par définition **optimisé pour le matériel** puisque l'on définit les branchements processeurs en fonction du code exécuté. A contrario il y aura un processus de routage important dans une architecture "gravé dans le marbre". En fait, avec un FPGA, il est possible de développer des Application Specific Integrated Circuit (ASIC), massivement employés dans la réalisation de taches simples et répétitives comme le minage de cryptomonnaies.
- En revanche, l'appréhension d'une tel technologie est loin d'être aisé, le **développement** avec des Hardware Description Langage (HDL) tel que le Verilog ou le VHDL est **loin d'être facile** et la formation est longue.
- Aussi, la **synthèse FPGA** pour passer du code au branchement de portes logiques est **longue** et nécessite des ***Toolchains*** de développement **lourde** (Vivado de Xilinx pèse 50 go sur un disque dur)

Vous l'aurez compris, l'implémentation matérielle d'un algorithme de compression vidéo est un point essentiel du partage de la bande passante mondiale.

En cela, afin de répondre à cette problématique nouvelle, l'ENSTA Bretagne voudrait développer un algorithme de compression vidéo, qui soit **open-source** et doté de performances convaincantes (définies par la suite), puis implémenter cet algorithme sur un système embarqué. La réalisation et l'implémentation de cet algorithme constitue notre travail, et se nomme **EVEEX (projet Encodage Vidéo ENSTA Bretagne Expérimental)**.

Le projet EVEEX

définition des exigences

Afin de définir clairement nos objectifs pour ce projet, il est primordial de définir les exigences, qu'elles soient fonctionnelles ou physiques (programmation).

Nous l'avons abordée dans l'introduction, la problématique des codecs vidéos est **essentielle** dans la gestion de la bande passante globale et de l'impact énergétique d'Internet. Pour permettre une amélioration collaborative et un accès universel, il est donc primordial que le projet soit **open-source** et libre de droit. Cet algorithme doit permettre l'extraction d'un flux vidéo, provenant par exemple d'une caméra, la compression de celui-ci, la mise en forme des données compressées, l'envoi de ces données à travers le réseau, le décodage des données reçues via le réseau, la décompression des données compressées ainsi que l'affichage de celles-ci.

Enfin, l'algorithme doit être développé dans un langage permettant une implémentation en embarqué que ce soit sur FPGA ou sur une autre technologie. Nous verrons plus tard que le choix du language est un point crucial dans la réalisation du projet.

Nous avons donc défini un certain nombre d'exigences avec les performances attendues lorsqu'elles sont pertinentes, ainsi que notre certitude quand à la réalisation de ces exigences.

Les points de vocabulaire au niveau de exigences seront définis par la suite dans le rapport, dans un glossaire.

Numéro identifiant l'exigence	Exigence	Performances attendues
1	Le projet doit être intégralement open-source et accessible gratuitement (exigence non fonctionnelle)	None
2.1	L'algorithme doit pouvoir recevoir un flux photos et vidéo "brut" et le convertir en un format exploitable	Conversion d'un flux RGB en flux YUV/YCbCr
2.2.1	L'algorithme doit compresser des données brutes	Dans un premier temps, performances analogues au MPEG-1 => 20:1 pour une photo, 100:1 pour vidéo

Numéro identifiant l'exigence	Exigence	Performances attendues
2.2.2	L'algorithme doit décompresser des données compressées	Performances identiques ou supérieures à celles de l'encodage
2.2.3	L'algorithme doit compresser les données d'une manière originale (pas une copie de MPEG)	None
2.3.1	L'algorithme doit pouvoir formater les données compressées afin qu'elles puissent être envoyées via un réseau.	None
2.3.2	L'algorithme doit pouvoir recevoir les données par le réseau et les comprendre	None
2.4.1	L'algorithme doit permettre un affichage d'une image décodée	Affichage VGA sur la carte FPGA
3.1	L'algorithme doit pouvoir s'exécuter sur une carte FPGA	Identiques ou supérieures à la version PC de l'algorithme
4.1	L'algorithme implémenté sur FPGA doit induire une faible consommation électrique	Inférieures à la consommation d'un PC exécutant l'algorithme. (<30W)

On rajoute à ces exigences les fonctions définissant les relations entre l'algorithme et les acteurs externes. Pour cela, la méthode du diagramme en pieuvre est utilisée. Elle permet d'illustrer clairement les fonctions accomplies par le système (ie l'algorithme EVEEX).

Maîtrise de l'algorithme

Un point clé dans la réussite du projet est la maîtrise de l'algorithme en lui même. Les choix qui seront fait sur cet algorithme seront déterminant en termes de performances.

Pour cet algorithme de traitement des données, nous nous sommes basés sur le MJPEG, car il est relativement simple à appréhender. **L'objectif de notre**

algorithme est de compresser l'image de référence le plus possible (ie avoir le meilleur taux de compression), et de faire cela le plus rapidement et le plus efficacement possible.

Le fonctionnement global de l'algorithme est détaillé dans le diagramme en blocs fourni en annexe. Il est composé de 3 phases principales. Pour chacune de ces 3 étapes (encodage, passage réseau et décodage), nous ferons un *zoom* sur le diagramme que nous avons effectué (par souci de simplicité).

L'image, au format RGB (que sort nativement la plupart des cameras), est tout d'abord **convertie au format chrominance/luminance (YUV)**.

Ensuite, l'image est découpée en **macroblocs** de 16x16 pixels. En réalité, comme une image RGB contient 3 canaux de couleur, les macroblocs sont en fait de taille 16x16x3, mais, par abus de langage, et par souci de simplicité, nous dirons simplement qu'ils ont une taille de 16x16 (ou NxN dans le cas général). Cette taille de macroblocs n'est pas arbitraire. En effet, nous avons déterminé **empiriquement** que, pour notre prototype, **et pour des images pré-existantes en 480p (720x480 pixels) ou alors générées aléatoirement**, les macroblocs 16x16 étaient ceux qui produisaient les meilleurs taux de compression parmi les tailles standards de macroblocs, à savoir 8x8, 16x16 et 32x32 pixels. Cette décomposition en macroblocs permet de faciliter le traitement de l'image et de paralléliser les tâches. De plus, nous nous différencierons des autres algorithmes existants en rendant cette taille de macroblocs **variable** en fonction du contenu du macrobloc. Par exemple, si un macrobloc présente un taux de contraste élevé, on réduit sa taille, alors que si c'est un aplat de couleur, on l'augmente. Cela permettra (a priori) d'améliorer le taux de compression.

Après cette étape, on applique diverses transformations **à chacune de ces matrices-macroblocs YUV** afin de les compresser. Ces transformations font partie de **l'étape d'encodage**.

- Une Transformation en Cosinus Discrète, ou **DCT [5]**, qui est une transformation linéaire et **réversible** qui va permettre de **concentrer** les données du macrobloc YUV dans la diagonale de l'image de sortie (la diagonale "nord-ouest / sud-est"). Ainsi, en-dehors de cette zone, les composantes de l'image (après application de la DCT) seront relativement faibles en valeur absolue, ce qui sera **très pratique** lors des étapes suivantes.
- On effectue ensuite **une linéarisation en zigzag** du macrobloc DCT ainsi généré. Cela signifie simplement que l'on va découper les 3 canaux 16x16 du macrobloc DCT en 3 vecteurs-listes de longueur 16x16 = 256. **On passe donc d'un array à 2 dimensions à un array en une seule dimension.** Ce découpage va se faire selon les 2x16-1 = 31 diagonales "sud-ouest / nord-est" de chacun des 3 canaux du macrobloc DCT (cf. image ci-dessous). Ce découpage, en conjonction avec la DCT (cf. étape précédente) est ici **extrêmement commode**, puisque l'on se retrouve avec des listes qui, en leur "centre", ont des valeurs repré-

sentatives non-négligeables, et puis, partout ailleurs, ces valeurs seront moindres.

— On effectue maintenant l'étape de seuillage, aussi appelée **quantization**. Cette opération consiste à ramener à zéro tous les éléments des 3 listes (issues de la linéarisation en zigzag) qui sont inférieurs **(en valeur absolue)** à un certain seuil, appelé *threshold* (ou *DEFAULT_QUANTIZATION_THREASHOLD* dans le code). Comme énoncé précédemment, la plupart des valeurs de ces 3 listes seront relativement faibles, donc appliquer ce seuillage va nous permettre d'avoir en sortie 3 listes avec **beaucoup de zéros**. Le seuil a ici été déterminé empiriquement, à partir d'une série de tests sur des images-macroblocs générées aléatoirement. **On a choisi *threshold* = 10, car il s'agissait de la valeur maximale qui permet quand même d'avoir une bonne qualité d'image en sortie.** Il est important de noter que cette étape de seuillage est **irréversible**.

— On passe ensuite à l'étape de la **RLE** (Run-Length Encoding). Cette étape consiste à regrouper de manière synthétique (dans des tuples, aussi appelés *tuples RLE*) les séries de zéros obtenues après l'étape de la quantization. Concrètement, si dans une liste seuillée on a 124 zéros puis un 5.21 (par exemple), d'abord 5.21 est arrondi à l'entier le plus proche (ici 5), puis cette série de 125 entiers sera stockée dans le tuple (124, 5). Plus généralement, si l'on a le tuple RLE "(U, V)", cela signifie que l'on a U zéros puis l'entier **non-nul** V. Ainsi, chaque macrobloc sera décrit de manière **extrêmement synthétique** par une liste de tuples RLE. **L'image finale, étant décomposée en une série de macroblocs, sera alors une liste de listes de tuples RLE.**

La partie suivante concerne le formatage des données. On utilise pour cela un **arbre binaire de Huffman** qui permet à la fois de compresser et de formater les données selon une trame précise. On appellera la trame à transmettre un **bitstream**.

```
Encoded string :
10101001100110000110011011011011100100110101000010100010010111
10101111100000101011001110101110000001001001101011111110101110
10100111110011000101111101111010101100101110110011001001001101 1
11000011111001000010
String decoded back : le chic de l'ensta bretagne sur la compression vide
```

L'arbre se base sur la récurrence des caractères afin de les ordonner et d'adresser à chaque caractère un mot binaire. Les "caractères" correspondent ici en fait à des tuples RLE. **L'idée est que, plus un tuple RLE apparaîtra souvent dans la frame RLE, moins le mot binaire qui lui est associé aura une taille élevée.** Les correspondances tuple RLE / mot binaire sont indiquées dans un dictionnaire, appelé **dictionnaire de Huffman**.

Après application de l'algorithme de Huffman **à la frame entière**, on se retrouve donc avec un dictionnaire de Huffman, ainsi qu'une frame RLE **prête à être**

encodée. Le dictionnaire de Huffman est ensuite converti en bitstream (ici une chaîne de caractères de "0" et de "1").

La raison pour laquelle on considère un bitstream, **envoyé d'un client à un serveur**, est parce que l'on veut simuler le transfert de données compressées d'un ordinateur à un autre, qui correspondront idéalement d'ici la fin de l'année à un flux vidéo compressé.

Comme rappelé en début de partie, l'objectif est d'avoir le meilleur taux de compression, c'est-à-dire que l'on veut minimiser la taille du bitstream total (envoyé du client au serveur) par rapport à la taille originale de l'image (en bits).

L'envoi du bitstream total se fera en **4 étapes** :

1. On envoie l'en-tête de la frame (ou le **header**), qui va contenir les métadonnées générales de l'image considérée : l'identifiant de la frame (frame_id), le type de message (HEADER_MSG, ie 0), la largeur de l'image (img_width), la hauteur de l'image (img_height) et la taille des macroblocs (macroblock_size).
2. On envoie ensuite le dictionnaire de Huffman encodé, **paquet par paquet**. Chaque paquet, de taille inférieure ou égale à *bufsize* (ici 4096 octets), contiendra les métadonnées du paquet (frame_id, type_msg = DICT_MSG = 1, index et packet_size), ainsi que ses données utiles, à savoir la partie du dictionnaire encodé que l'on veut envoyer. Chacun de ces paquets forment ce qu'on a appelé dans le code la partie **dict** du bitstream.
3. On envoie ensuite les paquets associés à chacun des macroblocs, qui seront également de taille limitée (<= bufsize). De même, chacun des paquets envoyés contiendra les métadonnées du paquet (frame*id, type_msg = BODY_MSG = 2, macroblock_number, index et packet_size) ainsi que la partie du bitstream associé à un macrobloc RLE,* ***encodé entre-temps depuis la frame RLE via l'algorithme de Huffman*** *(fonction d'encodage). Chacun de ces paquets forment le _corps* du bitstream, aussi appelé **body** dans le code.
4. Enfin, on envoie le message de fin, aussi appelée la queue du message (ou **tail**), qui est simplement là pour signaler que l'on arrive à la toute fin du bitstream. Ce message de fin contient seulement frame_id et type_msg (HEADER_MSG, ie 3).

Il est important de noter que, comme on veut également optimiser les performances temporelles de cet algorithme, il est primordial que l'on puisse convertir la frame RLE en bitstream ET envoyer ce dernier au serveur le plus rapidement possible. Ainsi, nous avons jugé intéressant de générer le bitstream dans un buffer via un thread en parallèle. Le thread principal n'aura alors qu'à extraire les paquets à envoyer de ce buffer, sans avoir à perdre de temps à les convertir. De même, le thread "écrivain" n'aura pas à perdre de

temps à attendre que le client envoie le paquet puis reçoive le message de retour du serveur (cf. diagramme).

Maintenant que le serveur a reçu l'entièreté du bitstream associé à l'image compressée, on va pouvoir commencer l'étape de **décodage**, qui constitue la troisème et dernière étape de notre algorithme. **Il s'agit en fait de l'étape d'encodage, mais effectuée dans l'ordre inverse.** La seule étape qui ne réapparaît pas au décodage est la **quantization**, ce qui est logique puisqu'il s'agit d'une étape irréversible. En effet, si une valeur a été seuillée (ie ramenée à zéro), on n'a - à ce stade - aucun moyen de savoir quelle était sa valeur initiale avant le seuillage.

Puis, finalement, après avoir décodé l'image au format YUV, on la convertit au format RGB.

En ce qui concerne les performances de cet algorithme, pour une image typique en 480p, notre algorithme s'effectue en une vingtaine de secondes en moyenne, et a des taux de compression variant entre 10:1 et 5:1 en moyenne. Ces taux de compression, *bien qu'améliorables*, sont toutefois assez satisfaisants, dans la mesure où les taux de compression d'algorithmes pré-existants (tels que le MPEG-2) varient typiquement entre 20:1 et 5:1 pour des images "classiques". Voici quelques statistiques de performances liées à notre algorithme :

Nous avons également mis en place une alternative à la DCT, la **iDTT** (integer Discrete Tchebychev Transform). Cette transformation va considérer (en entrée ET en sortie) des tableaux d'entiers, et non de flottants, comme le fait la DCT. Par rapport à la DCT, cette transformation est un tout petit peu plus précise (ce qui se traduit concrètement par une qualité d'image un peu plus élevée), mais il s'avère que le temps de calcul est bien plus élevé que pour la DCT classique. Voici quelques statistiques de performances liées à la version alternative de notre algorithme qui utilise la iDTT :

Nous avons implémenté cette méthode supplémentaire afin de sortir un peu des sentiers battus et de voir ce que l'on pouvait faire (ou optimiser) avec des méthodes entières (et non flottantes comme avec la DCT). **Comme les performances temporelles de la DCT surpassent largement celles de la iDTT, nous continuerons évidemment à nous focaliser principalement sur la DCT.**

Développement de l'algorithme

Le développement de cet algorithme s'est effectué en 3 phases : langage de haut niveau, puis de bas niveau pour finir par un langage d'un peu plus haut niveau. Voyons ensemble ces étapes :

Haut Niveau : code en python

La première étape consista en un développement orienté objet, dans un langage de haut niveau. Nous avons choisi le python car nous avions tous ou presque

de bonnes connaissances dans ce langage. La programmation objet nous a été utile pour appréhender les types et structures non natives nécessaire au fonctionnement de plusieurs blocs, notamment l'encodage de Huffman. Un autre avantage du python réside dans l'affectation en mémoire des variables dynamique et automatique.

Le développement en python de l'algorithme était relativement simple, et ce même en utilisant peu de bibliothèque externes (nous voulions garder une maîtrise sur le code, et éviter un effet "boite noire"). Les principales difficultés furent les suivantes :

— La conversion de RGB vers YUV a nécessité quelques recherches : les matrices de passages que nous trouvions avaient des coefficients différents et nous obtenions des résultats colorimétriquement problématiques. Nous avons néanmoins finis par trouver une source convenable et nous sommes passés à la suite.

—

Nous avons fait le choix de proposer un "package" de cet algorithme en python, afin de pouvoir l'importer facilement sur une machine ainsi que toute les bibliothèques nécessaire a son exécution.

Le code fonctionnant, passons à quelques statistiques et performances sur l'algorithme.

Bas niveau : code en C

A la suite du python, nous voulions améliorer les performances de l'algorithme. Notre choix c'est donc vite porté sur un language plus bas niveau comme le c, avec absence d'interpreteur. Cependant le code C présente quelques problèmes :

— l'allocation en mémoire n'est pas automatique (faite via malloc())
— les types sont peu nombreux et la création de classes (struct) difficile

Néanmoins, nous étions confiant sur la possibilité de réaliser le code, d'autant plus que nous disposions d'un guide pour faire de la programmation orienté objet en c. Nous avons donc commencé la construction du code.

Le temps passa et très vite le code devenu incontrôlable. La quantité de type non natifs produits devenait relativement important, le code augmentait très vite en volume et la structure de programmation devenu obfusqué et incompréhensible. On dépasse maintenant les 7000 lignes de code...

Ce code ne fait pas rien, la plupart des fonctions de l'encodeur sont implémentés. Le problème réside dans la liaison entre ces blocs de traitements (par exemple la liaison encodeur/huffman). De plus, le grand nombre de types et d'objets créés augmentait les fuites de mémoires et il fallu passer plus de 2 semaines pour boucher toutes les fuites par Valgrind.

Quelques performances en C :

langages alternatif : Golang (Go)

Devant l'échec du code en C, nous nous sommes orientés vers un langage plus facile à écrire, tout en disposant de performances équivalentes au c. Nous nous sommes donc orienté sur un langage poussé par Google et adopté par beaucoup d'entreprises du numériques : Go

Voici un aperçu des qualités et défauts du langage :

Avantages:

- la syntaxe est plus facile à lire que le c, notamment la gestion des array (slice) et la définition de méthode de "struct" se rapprochant de la POO
- Garbage collector intégré au langage (le go donne la localisation d'une erreur et est beaucoup plus verbeux sur le debug que le c et GCC)
- Outils de profiling du code intégré et bibliothèque de tests unitaires intégré.
- Beaucoup de garde fou : il n'y a pas de runtime error, juste des compilations d'erreurs
- Cross-compilation existante pour le langage.

inconvénients:

- Taille du binaire conséquente (un hello world de 2Mo ça fait mal)
- **cross-compilation impossible en riscV32 bits**, ainsi que la HLS via vivado.

Concernant la programmation en GO, l'encodeur fonctionne entièrement, cependant par faute de temps, le décodeur ainsi que le socket de transmission réseau ne fonctionne pas encore (70%).

Le Go possède comme indiqué précédemment un outil de profiling très évolué, donc voyons ensemble les conclusions que l'on peut en tirer :

On constate que la fonction de calcul du cosinus (qui intervient dans le calcul d'une DCT) consomme énormément de ressources, nous avons donc eu l'idée de passer par un développement de Taylor (rang 2).

Là, on constate qu'on accélère considérablement le calcul du cosinus , ce qui nous prouve l'utilité du développement de Taylor.

La problématique du VHDL

Dans nos plans initiaux, nous cherchions a

Implémentation sur l'embarqué

Alternative 1 : RISCV

Alternative 2 : ARM

Déroulement agile du projet

Le projet a été mené dans un cadre agile comme bon nombre de projet à court et moyen termes et a équipe réduite en informatique. La première étape a été la définition des rôles au sein du projet :

- ScrumMaster : Guillaume Leinen. Il est garant du respect de la méthode agile, ainsi que du rythme de production en sprint. Il n'est pas le chef de projet mais plutôt un coordinateurs des différents membres.
- Product Owner : Alexandre Froehlich. Il a la charge de vérifier que ce qui est réalisé est utile au projet final et rentre dans le cahier des charges fixé précédemment. Il est plutôt présent chez le client en temps normal, mais ce rôle a plutôt été rempli par M. Le Lann ainsi que M. Cotret.

Au début, nous avons choisi de répartir les taches selon 2 catégories :

- Une partie "software" constituée d'Alexandre, Hugo et Jean-Noël. Leur travail consiste à se focaliser sur la construction de l'algorithme et son développement dans plusieurs langages.
- Une partie "hardware" constituée de Guillaume et Hussein. Ils se focalisent sur l'implémentation matérielle de l'algorithme, et la gestion du FPGA et de ces constituants (RAM, I/O).

Mais très vite, cette frontière entre software et hardware finissait par ne plus avoir beaucoup de sens. Il devenait impossible de travailler le hardware sans avoir le software en tête et inversement. La distribution des taches s'est donc plus faite en fonction des appétences de chacun et des connaissances.

Concernant les sprints eux-mêmes, nous nous sommes orientés sur des sprints de **2 semaines**, avec un objectif de release (programme, documentation, fonctions supplémentaires, etc) **tous les 3 sprints**. Nous évaluons chaque tâche par un **système de points**, prenant en compte : la difficulté de la tâche, la longueur prévue, ou le nombre de personnes impliquées dans celle-ci. Cependant là aussi la rigidité d'un système de sprint aux 2 semaines commençait à peser sur la productivité, et les sprints vers la fin du projet étaient plus proche de la semaine voir de la demi journée.

Afin de planifier l'activité ainsi que de garder une trace de ce qui a été fait, nous nous sommes orientés vers une solution de méthode agile basé sur Github appelé "Zenhub".

Sur ce service, les taches sont regroupés en Issues comme lorsque l'on remonte un bug à un développeur, la différence étant qu'on peut facilement pipeliné la réalisation des issues, et les regroupés en différentes catégories notamment des milestones, qui correspondent au sprint (*Les sprints sont arrivés en tant que tel*

mais vers la fin du projet). De plus, ce formalisme permet d'extraire quantité d'informations et de statistiques de performance dont voici la principale :

Le **"Velocity tracking"** permet, via un système de points de notation des issues, de voir facilement l'étendue du travail réalisé au sein d'un sprint. Les sprints terminés sont grisés. On constate une périodicité, due notamment à la release tous les 3 sprints. On rajoute des issues au fur et a mesure des idées de tout le monde.

Nous nous sommes servis des descriptions des issues pour conserver les user-stories. entre autre, les points intéressants pour une user-story furent les suivants :

- **difficultés rencontrés** : quelles ont été les sources de difficulté dans le travail du ou des personnes réalisant l'issue. Cela permet de rafraîchir l'affectation des taches en fonctions des compétences de chacun et de la confiance en la réalisation de la tache.
- **travail réalisé** : L'issue a tel été réalisé en partie ? en totalité ?
- **perspectives futures** : ce qui va découler de la réalisation de la tache.

Nous complétions ces user-stories sous forme écrite par 15 minutes de démonstration en fin de chaque journée pour pouvoir montrer à tout le monde le travail réalisé.

Pour un aperçu plus convivial et plus chronologique du déroulé du projet, un portfolio est disponible sur Mahara à l'adresse suivante :

Conclusion

La démarche agile a été plus que nécessaire dans ce projet. En effet nous avons rencontrés plusieurs branches qui se sont avérés mortes ou sans issues a court terme. La démarche agile nous a permis de rebondir notamment lors de la fin du code c ou de l'implémentation fpga.

Nous nous sommes aperçu que

Annexes

Bibliographie

[**1**] https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[**2**] mpeg-la royalties https://www.businesswire.com/news/home/20150305006071/en/MPEG-LA-Introduces-License-DisplayPort

[**3**] encodage lent VP9 https://groups.google.com/a/webmproject.org/g/codec-devel/c/P5je-wvcs60?pli=1

[**4**] http://aomedia.org/about/

[**5**] https://www.lesnumeriques.com/tv-televiseur/av1-nouveau-standard-video-adopte-a-unanimite-n73213.html

[**6**] 2014. *Vivado Design Suite Tutorial : High-Level Synthesis*. [ebook] Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug871-vivado-high-level-synthesis-tutorial.pdf [Accessed 15 October 2020].

[**7**] LiteX-hub - Collaborative FPGA projects around LiteX. Available at: https://github.com/litex-hub [Accessed 1 December 2020].

[**8**] Zenhub.com - project Management in Github. Available at: https://www.zenhub.com/ [Accessed 15 october 2020]

[**9**] Pilai, L., 2003. *Huffman Coding*. [pdf] Available at: https://www.xilinx.com/support/documentation/application_notes/xapp616.pdf [Accessed 1 December 2020].

Glossaire

Flux-Vidéo: Processus d'envoi, réception et lecture en continu de la vidéo.

Open-Source: Tout logiciel dont les codes sont ouverts gratuitement pour l'utilisation ou la duplication, et qui permet de favoriser le libre échange des savoirs informatiques.

RGB: Rouge, Vert, Bleu. C'est un système de codage informatique des couleurs. Chaque pixel possède une valeur de rouge, une de vert et une de bleu.

FPGA: Une puce FPGA est un circuit imprimé reconfigurable fonctionnant à base de portes logiques.

Macrobloc (ou Macroblock) : Une partie de l'image de taille 16x16 pixels.

Bitstream: Flux de données en binaire.

VHDL: Langage de description de matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique.

IDE: Environnement de Développement.

Frames: Images qui composent une vidéo. On parle de FPS (Frames Per Second) pour mesurer la fréquence d'affichage.

HLS: High Level Synthesis. Outil logiciel permettant de synthétiser du code haut niveau en un code de plus bas niveau.

SoC: System on Chip. Puce de sillicium intégrant plusieurs composants, comme de la mémoire, un processeur, et un composant de gestion d'entrées/sorties.